ISTA 421 + INFO 521
Introduction to Machine Learning

Lecture 25:
Neural Networks II – The Backpropagation Algorithm

Clay Morrison
claytonm@email.arizona.edu
Harvill 437A
Phone 621-6609

22 November 2017

1

Rest of the Semester

- Mon, Nov 20 – Intro to NN
- Wed, Nov 22 – NN 2: Backpropagation
- Mon, Nov 27 – NN 3: Sparse Autoencoders
- Wed, Nov 29 – Clustering: K-Means
- Mon, Dec 4 – Clustering: Gaussian Mixture Model
- Wed, Dec 6 – Principle Components Analysis

- Homework 5 (NN) due Thurs, Dec 7
- Final Project due Mon, Dec 11

- **Reminder: Please fill out course evaluation (TCE)**

2

Neural Networks and Deep Learning

Homework 5 and a Note about Notation

- **Tutorial:** Andrew Ng's *Unsupervised Feature Learning and Deep Learning* (UFLDL) tutorial on building a **sparse autoencoder** for classifying handwriting
  - Main focus: **http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial**
    - The first section on the Sparse Autoencoder is what we'll focus on.
  - There is also the newer UFLDL tutorial: http://ufldl.stanford.edu/tutorial/
- **Notation:**
  - The UFLDL tutorial (and in many cases in the neural network literature), matrices representing weights going from one layer to the next do so by having column indices for the *from* (source) and row indices for the *to* (destination) weight links.
  - Also, superscripts in parentheses are used to represent individuals in training data or layer index in the network; subscripts used to index into the elements of a vector/matrix.

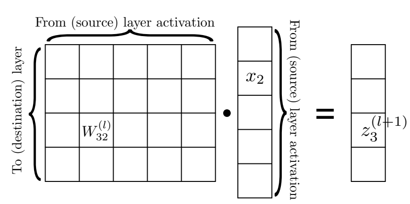
The i^{th} input output value

$(x^{(i)}, y^{(i)})$

The 3^{rd} value of the i^{th} input vector

$x_3^{(i)}$

scalar! (note the subscript)



vectors!

$z^{(l+1)} = W^{(l)}x$

matrix!

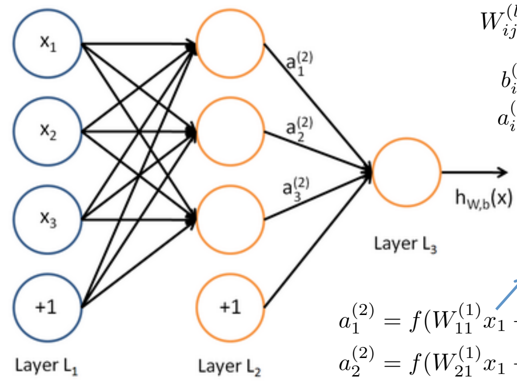
Feed Forward MLP Calculation (Forward Propagation)

n_l = number of layers = 3

$W_{ij}^{(l)}$ = weight between unit j in layer l and unit i in layer $l+1$

$b_i^{(l)}$ = bias for unit i in layer $l+1$

$a_i^{(l)}$ = activation (output value) of unit i in layer l



$a_i^{(1)} = x_i$

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

$z_i^{(l)} = \sum_{j=1}^{n} W_{ij}^{(l-1)} x_j + b_i^{(l-1)}$ and $a_i^{(l)} = f(z_i^{(l)})$

SISTA 5

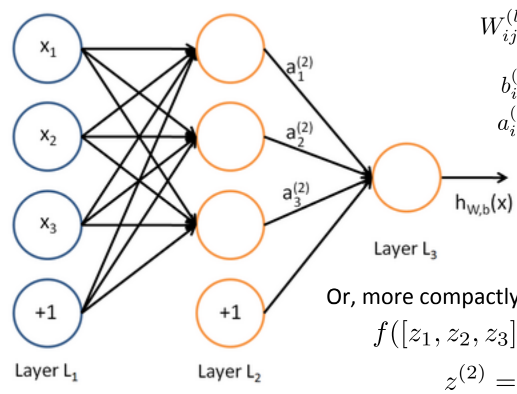
Feed Forward MLP Calculation (Forward Propagation)

n_l = number of layers = 3

$W_{ij}^{(l)}$ = weight between unit j in layer l and unit i in layer $l+1$

$b_i^{(l)}$ = bias for unit i in layer $l+1$

$a_i^{(l)}$ = activation (output value) of unit i in layer l

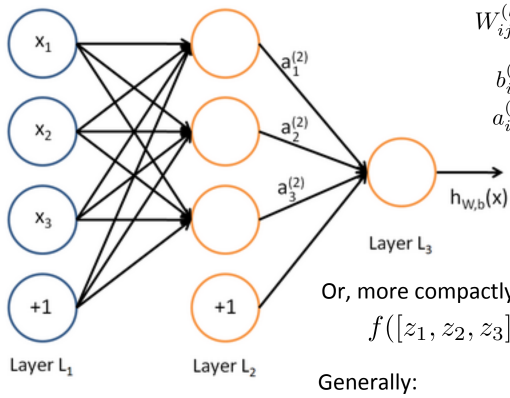


Or, more compactly, if we make *f*(•) a vector function:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$
$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$$

SISTA 6

Feed Forward MLP Calculation (Forward Propagation)



n_l = number of layers = 3

$W_{ij}^{(l)}$ = weight between unit j in layer l and unit i in layer $l+1$

$b_i^{(l)}$ = bias for unit i in layer $l+1$

$a_i^{(l)}$ = activation (output value) of unit i in layer l

Or, more compactly, if we make *f*(•) a vector function:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

Generally:

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)} \qquad \text{where } a^{(1)} = x$$

$$a^{(l+1)} = f(z^{(l+1)})$$

SISTA 7

Gradient Descent

- Given a loss function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \Big(\underbrace{\| h_\theta(x^{(i)}) - y^{(i)} \|} \Big)^2$$

L_2 norm (i.e., Euclidean distance)
when h_θ and y are vectors; when scalars, just subtract

- *m* is the number of examples and *h* some function of parameters *θ*
- Gradient descent updates the parameters in steps:

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

SISTA 8

Backpropagation Algorithm
(for fully connected feedforward network)

training pairs: $\{(x^{(1)}, y^{(1)}), (x^{(m)}, y^{(m)})\}$

Define the loss

Neural networks can be prone to overfitting, so regularize

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\underbrace{\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2}_{\text{Implementation note: use numpy.linalg.norm()}}\right)\right] + \underbrace{\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2}_{\text{weight decay term}}$$

Goal is to minimize $J(W, b)$

weight decay term
(note that reg term typically not over bias terms)

One iteration of gradient descent updates parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} \qquad \frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \left[\frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial W_{ij}^{(l)}}\right] + \lambda W_{ij}^{(l)}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(W,b)}{\partial b_i^{(l)}} \qquad \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial b_i^{(l)}}$$

SISTA 9

Backpropagation Algorithm
(for fully connected feedforward network)

Partial derivatives of cost function for single example (x,y) $\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}}$ $\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}}$

Backprop Core (in words):

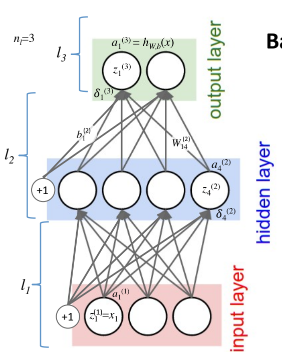
Given training example (x,y) …

Run "forward pass" to compute all activations throughout network

For each node i in layer l, compute "error term" $\delta_i^{(l)}$ Measures how much node was "responsible" for any errors in output

For an output node, can directly measure the difference between network's activation and the true target value, to define $\delta_i^{(n_l)}$

For hidden units, compute $\delta_i^{(l)}$ based on weighted average of the error terms of the nodes that use $a_i^{(l)}$ as an input.



Backprop Core (in more detail):

(1) Perform feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}

(2) For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}}\frac{1}{2}\|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

Indices are correct! i is layer l, j is layer l+1

(3) For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$
For each node I in layer l, set $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'(z_i^{(l)})$

(4) Compute the desired partial derivatives, which are given as:

$$\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}} = a_j^{(l)}\delta_i^{(l+1)} \qquad \frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

SISTA 10

Backpropagation Algorithm
(for fully connected feedforward network)

Partial derivatives of cost function for single example (x,y): $\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}}$ $\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}}$

Backprop Core (Vectorized):

'•' denotes element-wise product (Hadamard product)
$f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$

(1) Perform feedforward pass

(2) For the output layer (layer n_l), set $\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$

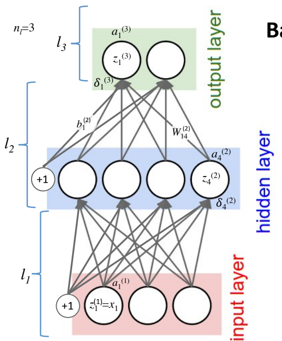
(3) For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$, set $\delta^{(l)} = \left((W^{(l)})^\top \delta^{(l+1)}\right) \bullet f'(z^{(l)})$

(4) Compute the desired partial derivatives, which are given as:

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^\top$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}$$

Yes, this *is* an outer product!!
Implementation note: look at `numpy.linalg.outer()`

If sigmoid activation: $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$ — store during forward pass



Backprop Core (in more detail):

(1) Perform feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}

(2) For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

(3) For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$
For each node I in layer l, set $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}\right) f'(z_i^{(l)})$

(4) Compute the desired partial derivatives, which are given as:

$$\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \qquad \frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

SISTA 11

Backpropagation Algorithm
(for fully connected feedforward network)

Partial derivatives of cost function for single example (x,y): $\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}}$ $\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}}$

Backprop Core (Vectorized):

'•' denotes element-wise product (Hadamard product)
$f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$

(1) Perform feedforward pass

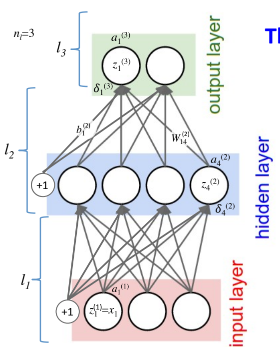
(2) For the output layer (layer n_l), set $\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$

(3) For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$, set $\delta^{(l)} = \left((W^{(l)})^\top \delta^{(l+1)}\right) \bullet f'(z^{(l)})$

(4) Compute the desired partial derivatives, which are given as:

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)} (a^{(l)})^\top$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}$$

If sigmoid activation: $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$ — store during forward pass



The Complete Backprop Algorithm as Gradient Descent:

(1) Set $\Delta W^{(l)} := 0,\ \Delta b^{(l)} := 0$ for all l (note that 'ΔW' and 'Δb' are single variables)

(2) For i = 1 to m,

(a) Use backpropagation (Backprop Core) to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ $\nabla_{b^{(l)}} J(W,b;x,y)$

(b) Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$

(c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$

(3) Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \underbrace{\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right]}_{\text{gradient}} \qquad b^{(l)} = b^{(l)} - \alpha \underbrace{\left[\frac{1}{m}\Delta b^{(l)}\right]}_{\text{gradient}}$$

Repeat (2) and (3) until "convergence"

Initializing Parameters

- If all parameters start off at identical values (e.g., 0), then all hidden layer units would learn the same function of the input.
- Random initialization breaks symmetry.
- Often sample small random values near zero:

$$\mathcal{N}(0, \epsilon^2) \quad (\text{e.g., } \epsilon = 0.01)$$

- Another heuristic:

$$Uniform\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}\right]$$

SISTA 13

Improvements to Gradient Descent

$$W^{(l)} = W^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right] \qquad b^{(l)} = b^{(l)} - \alpha\left[\frac{1}{m}\Delta b^{(l)}\right]$$

- Automatically adapting α
- Newton's method(s): using Hessian information
- L-BFGS
- Conjugate Gradient

What you need to provide:

A function that returns the cost ($J(\theta)$) and grad ($\nabla_\theta J(\theta)$)

Initial theta

```
J = lambda x: your_fn(x, vis_size, hid_size, lambda_, data)
scipy.optimize.minimize(fun=J, x0=theta, method='L-BFGS-B')
```

14

Debugging: Gradient Checking

Numerically checking derivatives

Recall mathematical definition of the derivative (assuming *J* is fn of *θ*):

$$\frac{\partial J(\theta)}{\partial \theta} = \lim_{\epsilon \to 0} \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$$

At any specific value of *θ* we can numerically approximate the derivative by:

$$\frac{J(\theta + \mathrm{EPSILON}) - J(\theta - \mathrm{EPSILON})}{2 \times \mathrm{EPSILON}}$$

In practice, set EPSILON to a small constant, such as 10^{-4} = 0.0001.

You can use this numerical approximation to compare against a function $g(\theta) = \frac{\partial J(\theta)}{\partial \theta}$ used to compute the gradient.

$$g(\theta) \approx \frac{J(\theta + \mathrm{EPSILON}) - J(\theta - \mathrm{EPSILON})}{2 \times \mathrm{EPSILON}}$$

With EPSILON = 10^{-4}, you'll usually get a similarity between g(*θ*) and your numerical approximation to at least 4 significant digits.

SISTA 15

Debugging: Gradient Checking

Numerically checking derivatives $g(\theta) \approx \frac{J(\theta + \mathrm{EPSILON}) - J(\theta - \mathrm{EPSILON})}{2 \times \mathrm{EPSILON}}$

$$g(\theta) = \frac{\partial J(\theta)}{\partial \theta}$$

The above approximates the gradient for a single parameter *θ*.
If we are computing the derivative of each component parameter *i* in vector *θ*, then we can modify the above estimate by varying the specific parameter value by EPSILON:

$$g_i(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i \\ \vdots \\ \theta_N \end{bmatrix} \qquad \mathbf{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Let $\boldsymbol{\theta}^{(i+)} = \boldsymbol{\theta} + \mathrm{EPSILON} \times \mathbf{e}_i$

$\boldsymbol{\theta}^{(i-)} = \boldsymbol{\theta} - \mathrm{EPSILON} \times \mathbf{e}_i$

The comparison then becomes:

$$g_i(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}^{(i+)}) - J(\boldsymbol{\theta}^{(i-)})}{2 \times \mathrm{EPSILON}}$$

The goal of each iteration of the Backpropagation Algorithm is to compute the gradient

$$W^{(l)} = W^{(l)} - \alpha \underbrace{\left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]}_{\substack{\text{gradient} \\ \nabla_{W^{(l)}} J(W,b)}} \qquad b^{(l)} = b^{(l)} - \alpha \underbrace{\left[\frac{1}{m} \Delta b^{(l)} \right]}_{\substack{\text{gradient} \\ \nabla_{b^{(l)}} J(W,b)}}$$

The above numerical gradient estimation can be used to numerically compute the derivatives of $J(W,b)$ and compare to the computations of $\nabla_{W^{(l)}} J(W,b)$, $\nabla_{b^{(l)}} J(W,b)$