

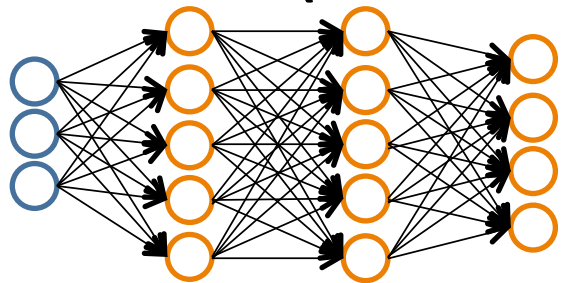


Machine Learning

Neural Networks: Learning

Cost function

Neural Network (Classification)



Layer 1 Layer 2 Layer 3 Layer 4

2 types of classification problems:

Binary classification

$y = 0$ or 1

1 output unit

$$h_0 \in \mathbb{R}$$

SL=1, K=number
of unit in output
layer = 1

$\frac{1}{2} \rightarrow \text{circle} \rightarrow h_0 \in \mathbb{R}$

$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

m training examples

$L =$ total no. of layers in network

L=4 here

$s_l =$ no. of units (not counting bias unit) in layer l

S1=3, S2=5, S4=SL=4

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

K classes

pedestrian car motorcycle truck

K output units

$$h_0 \in \mathbb{R}^K$$

SL=K, K>= 3

Cost function

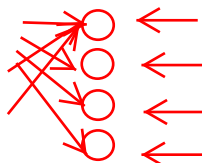
Logistic regression:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

Neural network h is a k dimensional vector, h_i is the ith element of the vector that is output by neural network

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

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



So if I have four output units, that is if the final layer of my neural network has four output units, then this is a sum from k equals one through four of basically the logistic regression algorithm's cost function but summing that cost function over each of my four output units in turn.

it's summing over these terms θ_{ji} for all values of j and i . Except that we don't sum over the terms corresponding to these bias values like we have for logistic progression. We don't sum over the terms responding to where i is equal to 0, that is because when we're computing the activation of a neuron, we have terms like these. And so the values with a zero there, that corresponds to something that multiplies into an x_0 or an a_0 . And so this is like a bias unit and by analogy to what we were doing for logistic progression, we won't sum over those terms in our regularization term because we don't want to regularize them and string their values as zero. But this is just one possible convention, and even if you were to sum over i equals 0 up to S_I , it would work about the same and doesn't make a big difference. But maybe this convention of not regularizing the bias term is just slightly more common.

$$\frac{\theta_{i0}}{a_0} x_0 + \theta_{i1} x_1 + \dots$$



Machine Learning

Neural Networks: Learning

Backpropagation algorithm

Gradient computation

Gradient computation

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

➔ $\min_{\Theta} J(\Theta)$

Need code to compute:

→ $-J(\Theta)$

$$\rightarrow - \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \leftarrow$$

$$\textcircled{1}^{(k)}_{ij} \in \mathbb{R}$$

Gradient computation

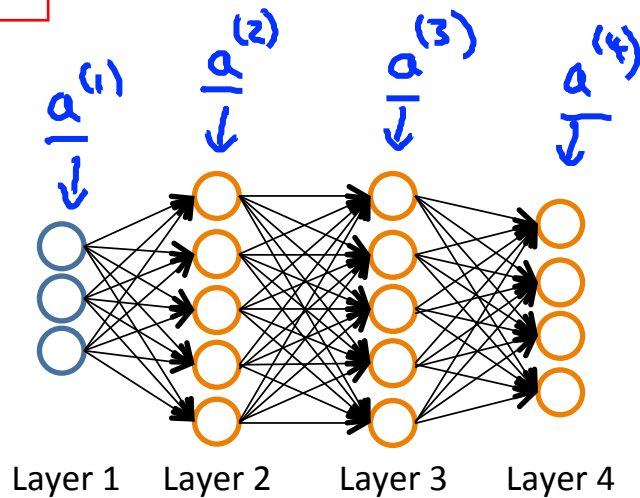
The first thing we do is we apply forward propagation in order to compute whether a hypotheses actually outputs given the input.

Given one training example (x, y) :

Forward propagation:

$$\begin{aligned} &\rightarrow \underline{a^{(1)}} = \underline{x} \\ &\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)} \\ &\rightarrow a^{(2)} = g(z^{(2)}) \quad (\text{add } \underline{a_0^{(2)}}) \\ &\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)} \\ &\rightarrow a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\ &\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)} \\ &\rightarrow \underline{a^{(4)}} = \underline{h_{\Theta}(x)} = g(z^{(4)}) \end{aligned}$$

Next, in order to compute the "derivatives", we're going to use an algorithm called "back propagation".



Back propagation comes from the fact that we start by computing the delta term for the output layer and then we go back a layer and compute the delta terms for the third hidden layer and then we go back another step to compute delta 2 ..

Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l .

For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

= difference between hypothesis and value of y

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)})$$

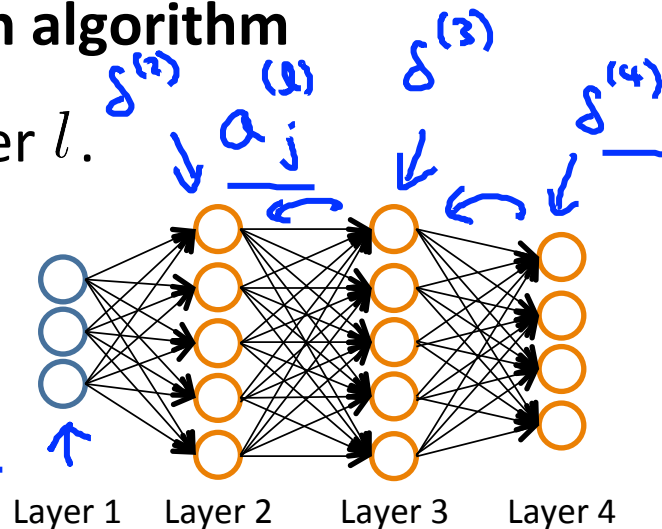
(No $\delta^{(1)}$)

there is no delta 1 term

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

if ignore regularization

(ignore λ ; if $\lambda = 0$)



$\begin{bmatrix} 1 \\ a3_1 \\ a3_2 \\ x \\ x \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1-a3_1 \\ 1-a3_2 \\ 1-x \\ 1-x \end{bmatrix}$
	\cdot

these deltas are going to be used as accumulators that will slowly add things in order to compute these partial derivatives.

Backpropagation algorithm

→ Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

Set $\underline{\Delta_{ij}^{(l)}} = 0$ (for all l, i, j).

(used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to m ← $(\underline{x^{(i)}}, \underline{y^{(i)}})$

Set $\underline{a^{(1)}} = \underline{x^{(i)}}$

→ Perform forward propagation to compute $\underline{a^{(l)}}$ for $l = \underline{2}, \underline{3}, \dots, \underline{L}$

→ Using $\underline{y^{(i)}}$, compute $\underline{\delta^{(L)}} = \underline{a^{(L)}} - \underline{y^{(i)}}$

→ Compute $\underline{\delta^{(L-1)}}, \underline{\delta^{(L-2)}}, \dots, \underline{\delta^{(2)}}$

~~set~~ there is no delta 1 term

→ $\underline{\Delta_{ij}^{(l)}} := \underline{\Delta_{ij}^{(l)}} + \underline{a_j^{(l)}} \underline{\delta_i^{(l+1)}}$ ← Δ

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

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

→ $\underline{D_{ij}^{(l)}} := \frac{1}{m} \underline{\Delta_{ij}^{(l)}}$ if $\underline{j = 0}$ bias terms

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Suppose you have two training examples $(\mathbf{x}^{(1)}, \mathbf{y}^{(1)})$ and $(\mathbf{x}^{(2)}, \mathbf{y}^{(2)})$. Which of the following is a correct sequence of operations for computing the gradient? (Below, FP = forward propagation, BP = back propagation).

- ☐ FP using $\mathbf{x}^{(1)}$ followed by FP using $\mathbf{x}^{(2)}$. Then BP using $\mathbf{y}^{(1)}$ followed by BP using $\mathbf{y}^{(2)}$.
- ☐ FP using $\mathbf{x}^{(1)}$ followed by BP using $\mathbf{y}^{(2)}$. Then FP using $\mathbf{x}^{(2)}$ followed by BP using $\mathbf{y}^{(1)}$.
- ☐ BP using $\mathbf{y}^{(1)}$ followed by FP using $\mathbf{x}^{(1)}$. Then BP using $\mathbf{y}^{(2)}$ followed by FP using $\mathbf{x}^{(2)}$.
- ☒ FP using $\mathbf{x}^{(1)}$ followed by BP using $\mathbf{y}^{(1)}$. Then FP using $\mathbf{x}^{(2)}$ followed by BP using $\mathbf{y}^{(2)}$.

Correct

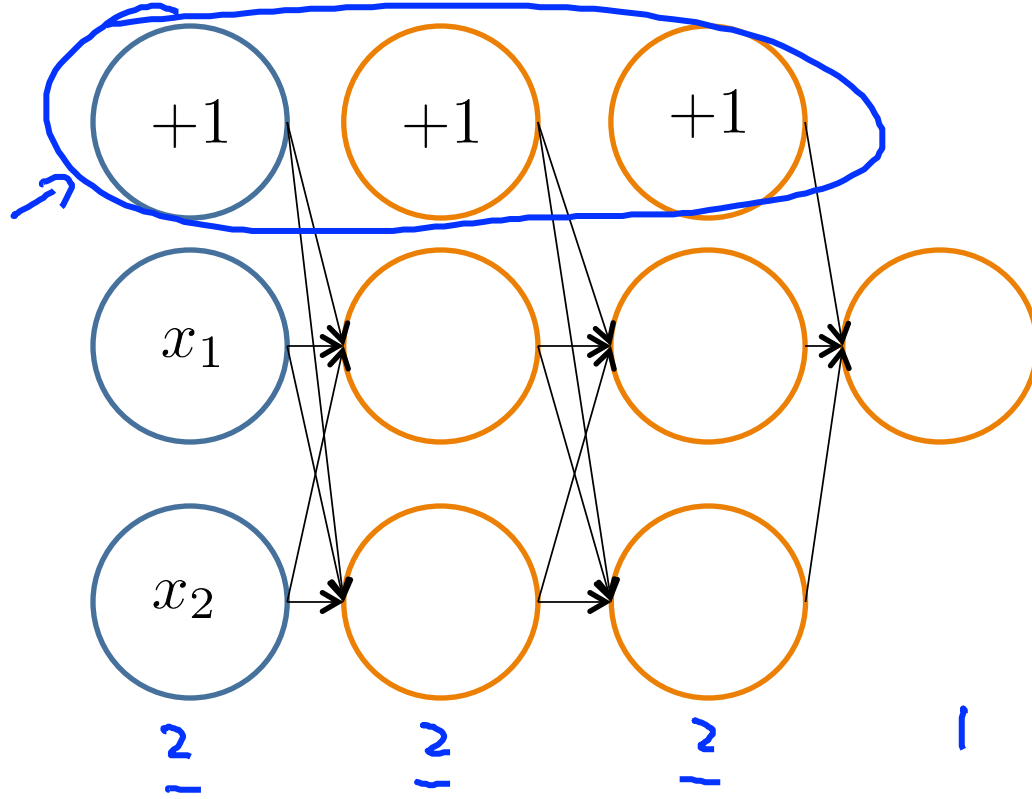


Machine Learning

Neural Networks: Learning

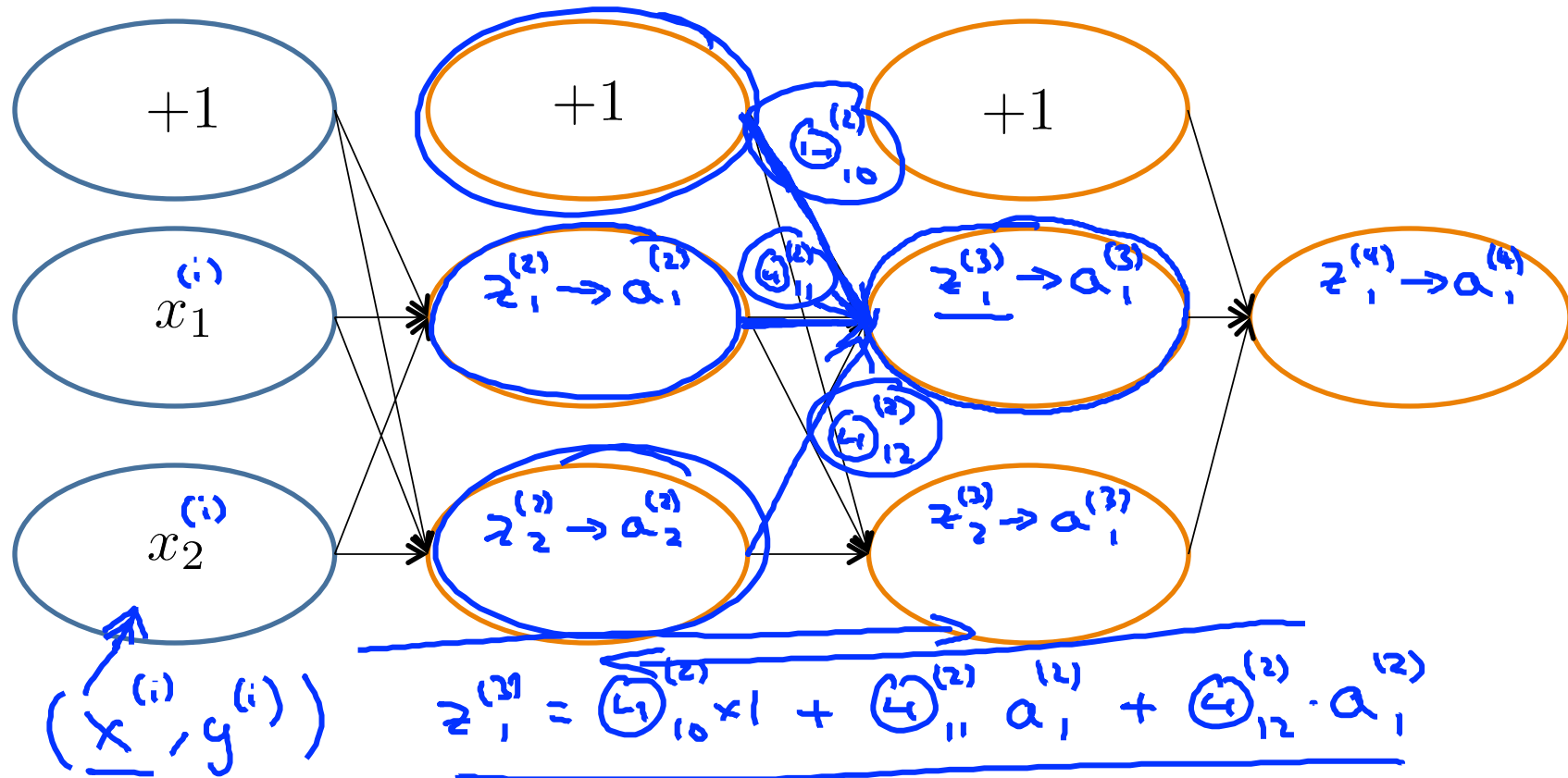
Backpropagation intuition

Forward Propagation



Here's a neural network with two input units that is not counting the bias unit, and two hidden units in this layer, and two hidden units in the next layer. And then, finally, one output unit. Again, these counts two, two, two, are not counting these bias units on top. In order to illustrate forward propagation, I'm going to draw this network a little bit differently.

Forward Propagation



And in particular I'm going to draw this neuro-network with the nodes drawn as these very fat ellipsis, so that I can write text in them. When performing forward propagation, we might have some particular example. Say some example $x^{(i)}$ comma $y^{(i)}$. And it'll be this $x^{(i)}$ that we feed into the input layer. So this maybe $x^{(i)}_2$ and $x^{(i)}_2$ are the values we set the input layer to. And when we forward propagated to the first hidden layer here, what we do is compute $z^{(2)}_1$ and $z^{(2)}_2$. So these are the weighted sum of inputs of the input units. And then we apply the sigmoid of the logistic function, and the sigmoid activation function applied to the z value. Here's are the activation values. So that gives us a $(2)_1$ and a $(2)_2$. And then we forward propagate again to get here $z^{(3)}_1$. Apply the sigmoid of the logistic function, the activation function to that to get a $(3)_1$. And similarly, like so until we get $z^{(4)}_1$. Apply the activation function. This gives us a $(4)_1$, which is the final output value of the neural network.

'''
Let's erase this arrow to give myself some more space. And if you look at what this computation really is doing, focusing on this hidden unit, let's say. We have to add this weight. Shown in magenta there is my weight $\theta^{(2)}_{10}$, the indexing is not important. And this way here, which I'm highlighting in red, that is $\theta^{(2)}_{11}$ and this weight here, which I'm drawing in cyan, is $\theta^{(2)}_{12}$. So the way we compute this value, $z^{(3)}_1$ is, $z^{(3)}_1$ is as equal to this magenta weight times this value. So that's $\theta^{(2)}_{10} \times 1$. And then plus this red weight times this value, so that's $\theta^{(2)}_{11}$ times $a^{(2)}_1$. And finally this cyan weight times this value, which is therefore plus $\theta^{(2)}_{12}$ times $a^{(2)}_1$. And so that's forward propagation. And it turns out that as we'll see later in this video, what backpropagation is doing is doing a process very similar to this. Except that instead of the computations flowing from the left to the right of this network, the computations since their flow from the right to the left of the network. And using a very similar computation as this. And I'll say in two slides exactly what I mean by that.

What is backpropagation doing?

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_{\Theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

(x⁽ⁱ⁾, y⁽ⁱ⁾)

Focusing on a single example x⁽ⁱ⁾, y⁽ⁱ⁾, the case of 1 output unit, and ignoring regularization (λ = 0),

Note: Mistake on lecture, it is supposed to be 1-h(x).

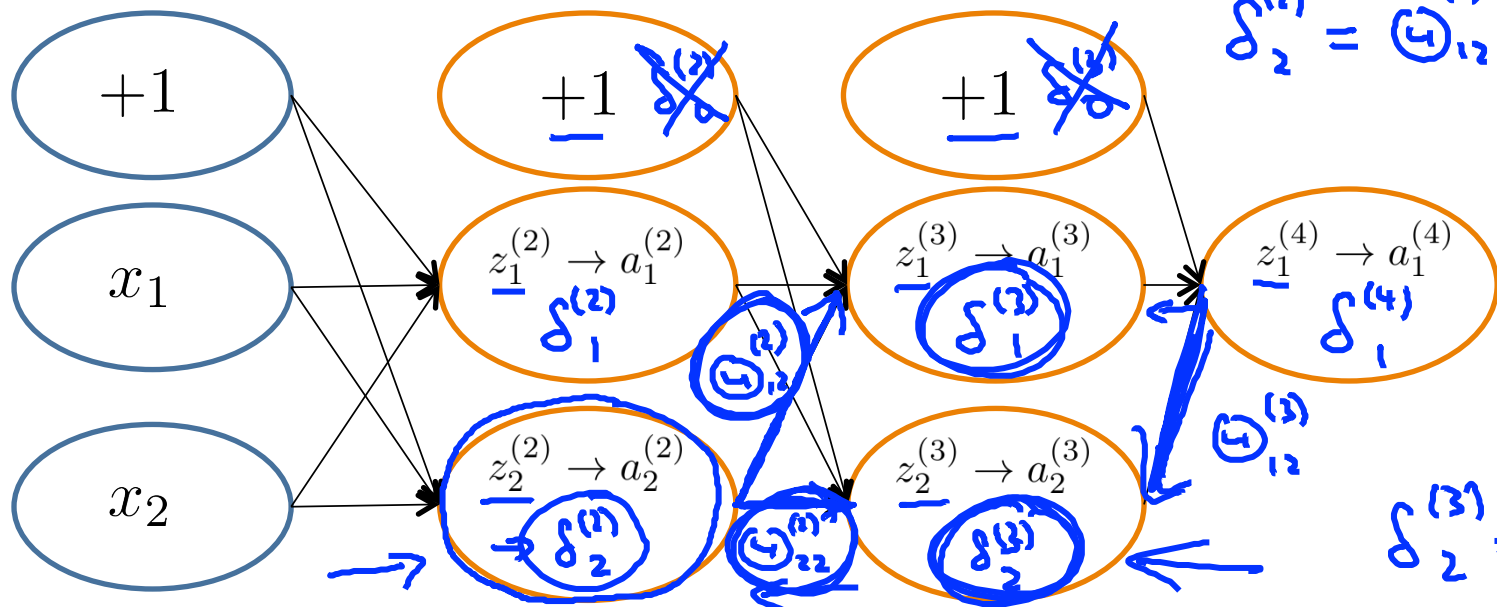
$$\text{cost}(i) = y^{(i)} \log h_{\Theta}(x^{(i)}) + (1 - y^{(i)}) \log h_{\Theta}(x^{(i)})$$

(Think of cost(i) ≈ (h_Θ(x⁽ⁱ⁾) - y⁽ⁱ⁾)²)

I.e. how well is the network doing on example i?

. To better understand what backpropagation is doing, let's look at the cost function. It's just the cost function that we had for when we have only one output unit. If we have more than one output unit, we just have a summation you know over the output units indexed by k there. If you have only one output unit then this is a cost function. And we do forward propagation and backpropagation on one example at a time. So let's just focus on the single example, $x^{(i)}$ $y^{(i)}$ and focus on the case of having one output unit. So $y^{(i)}$ here is just a real number. And let's ignore regularization, so λ equals 0. And this final term, that regularization term, goes away. Now if you look inside the summation, you find that the cost term associated with the training example, that is the cost associated with the training example $x^{(i)}$, $y^{(i)}$. That's going to be given by this expression. So, the cost to live off example i is written as follows. And what this cost function does is it plays a role similar to the squared error. So, rather than looking at this complicated expression, if you want you can think of cost of i being approximately the square difference between what the neural network outputs, versus what is the actual value. Just as in logistic regression, we actually prefer to use the slightly more complicated cost function using the log. But for the purpose of intuition, feel free to think of the cost function as being the sort of the squared error cost function. And so this $cost(i)$ measures how well is the network doing on correctly predicting example i .

Forward Propagation



$\rightarrow \delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l).

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ (for $j \geq 0$), where

$$\text{cost}(i) = y^{(i)} \log h_{\Theta}(x^{(i)}) + (1 - y^{(i)}) \log h_{\Theta}(x^{(i)})$$

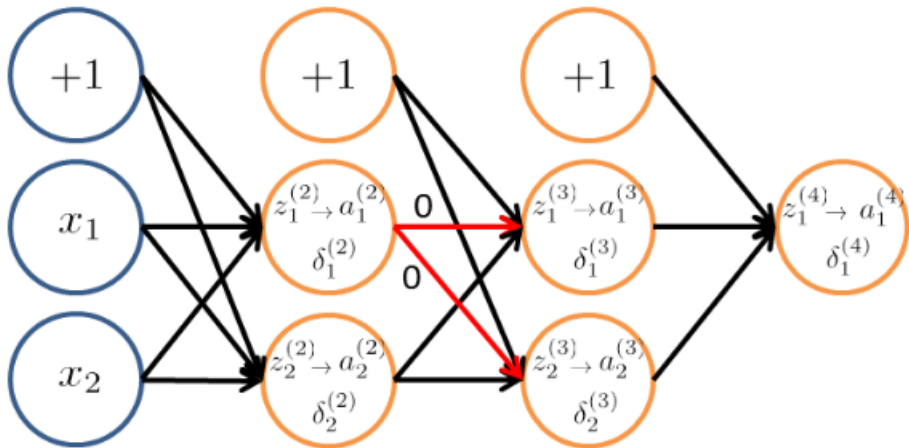
*backpropagation is computing these delta superscript l subscript j terms. And we can think of these as the error of the activation value that we got for unit j in the layer, in the l th layer.

*More formally, what the delta terms actually are is this, Partial derivatives with respect to these things of the cost function. And if we could go inside the neural network and just change those z_{lj} values a little bit, then that will affect these values that the neural network is outputting. And that will end up changing the cost function.

*And so they're a measure of how much would we like to change the neural network's weights, in order to affect these intermediate values of the computation. So as to affect the final output of the neural network $h(x)$ and therefore affect the overall cost.

*What backpropagation is doing: For the output layer, the first set's this delta term, $\delta^{(4)}_1$, as $y^{(i)}$ if we're doing forward propagation and back propagation on this training example i . That says $y^{(i)}$ minus $a^{(4)}_1$. So this is really the error, right? It's the difference between the actual value of y minus what was the value predicted, and so we're gonna compute $\delta^{(4)}_1$ like so. Next we're gonna do, propagate these values backwards. I'll explain this in a second, and end up computing the delta terms for the previous layer. We're gonna end up with $\delta^{(3)}_1$. $\delta^{(3)}_2$. And then we're gonna propagate this further backward, and end up computing $\delta^{(2)}_1$ and $\delta^{(2)}_2$. Now the backpropagation calculation is a lot like running the forward propagation algorithm, but doing it backwards. So here's what I mean. Let's look at how we end up with this value of $\delta^{(2)}_2$. So we have $\delta^{(2)}_2$. And similar to forward propagation, let me label a couple of the weights. So this weight, which I'm going to draw in cyan. Let's say that weight is $\theta^{(2)}_{12}$, and this one down here when we highlight this in red. That is going to be let's say $\theta^{(2)}_{22}$. So if we look at how $\delta^{(2)}_2$ is computed, how it's computed with this note. It turns out that what we're going to do, is gonna take this value and multiply it by this weight, and add it to this value multiplied by that weight. So it's really a weighted sum of these delta values, weighted by the corresponding edge strength. So completely, let me fill this in, this $\delta^{(2)}_2$ is going to be equal to, $\theta^{(2)}_{12}$ is that magenta lay times $\delta^{(3)}_1$. Plus, and the thing I had in red, that's $\theta^{(2)}_{22}$ times $\delta^{(3)}_2$. So it's really literally this red wave times this value, plus this magenta weight times this value. And that's how we wind up with that value of delta. And just as another example, let's look at this value. How do we get that value? Well it's a similar process. If this weight, which I'm gonna highlight in green, if this weight is equal to, say, $\delta^{(3)}_{12}$. Then we have that $\delta^{(3)}_2$ is going to be equal to that green weight, $\theta^{(3)}_{12}$ times $\delta^{(4)}_1$. And by the way, so far I've been writing the delta values only for the hidden units, but excluding the bias units. Depending on how you define the backpropagation algorithm, or depending on how you implement it, you know, you may end up implementing something that computes delta values for these bias units as well. The bias units always output the value of plus one, and they are just what they are, and there's no way for us to change the value. And so, depending on your implementation of back prop, the way I usually implement it. I do end up computing these delta values, but we just discard them, we don't use them.

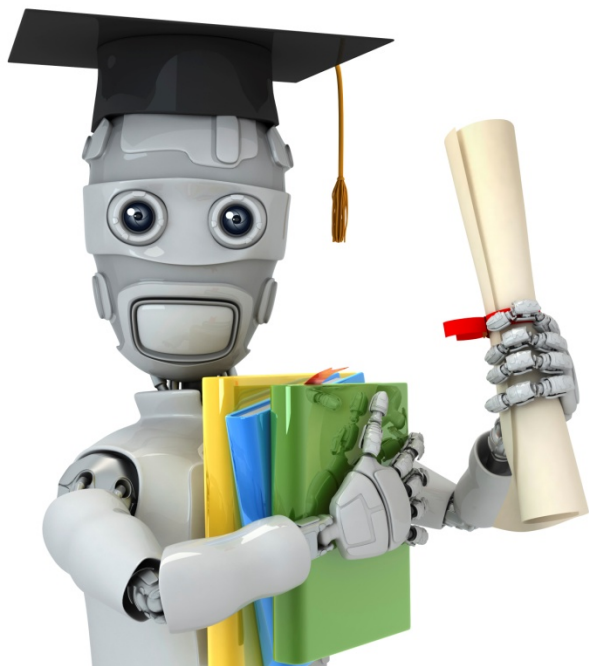
Consider the following neural network:



Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running backpropagation, what can we say about the value of $\delta_1^{(3)}$?

- ☐ $\delta_1^{(3)} > 0$
- ☐ $\delta_1^{(3)} = 0$ only if $\delta_1^{(2)} = \delta_2^{(2)} = 0$, but not necessarily otherwise
- ☐ $\delta_1^{(3)} \leq 0$ regardless of the values of $\delta_1^{(2)}$ and $\delta_2^{(2)}$
- ☒ There is insufficient information to tell

Correct



Neural Networks: Learning

Implementation note: Unrolling parameters

The advantage of the matrix representation is that when your parameters are stored as matrices it's more convenient when you're doing forward propagation and back propagation and it's easier when your parameters are stored as matrices to take advantage of the, sort of, vectorized implementations.

Whereas in contrast the advantage of the vector representation, when you have like θ_{Vec} or D_{Vec} is that when you are using the advanced optimization algorithms. Those algorithms tend to assume that you have all of your parameters unrolled into a big long vector. And so with what we just went through, hopefully you can now quickly convert between the two as needed.

Advanced optimization

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

Handwritten annotations: \mathbb{R}^{n+1} (under gradient), \mathbb{R}^{n+1} (vectors) (under theta), and an arrow pointing from initialTheta to the \mathbb{R}^{n+1} (vectors) label.

Neural Network (L=4):

→ $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)

→ $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ - matrices (D1, D2, D3)

“Unroll” into vectors

how to take this matrices and unroll it into vectors in a format so that we can feed into

Example

$$s_1 = 10, s_2 = 10, s_3 = 1$$

$$\rightarrow \Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$$

$$\rightarrow D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$$

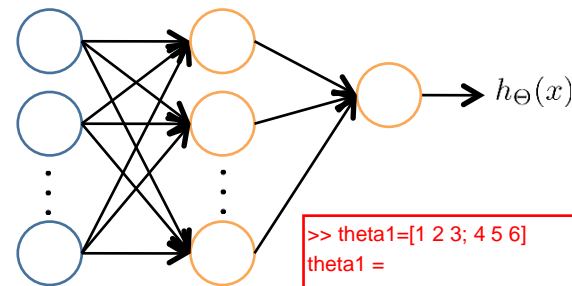
$$\rightarrow \text{thetaVec} = [\text{Theta1}(:); \text{Theta2}(:); \text{Theta3}(:)] ;$$

$$\rightarrow \text{DVec} = [\text{D1}(:); \text{D2}(:); \text{D3}(:)] ;$$

$$\text{Theta1} = \text{reshape}(\text{thetaVec}(1:110), 10, 11) ;$$

$$\rightarrow \text{Theta2} = \text{reshape}(\text{thetaVec}(111:220), 10, 11) ;$$

$$\rightarrow \text{Theta3} = \text{reshape}(\text{thetaVec}(221:231), 1, 11) ;$$



```
>> theta1=[1 2 3; 4 5 6]
theta1 =
     1     2     3
     4     5     6
>> theta2=[7 8 9; 10 11 12]
theta2 =
     7     8     9
    10    11    12
>> thetavec=[theta1(:);theta2(:)]
thetavec =
     1
     4
     2
     5
     3
     6
     7
    10
     8
    11
     9
    12
>> A1=reshape(thetavec(1:6),2,3)
A1=1     2     3
```

Learning Algorithm

- Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
- Unroll to get `initialTheta` to pass to
- `fminunc(@costFunction, initialTheta, options)`

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

- From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$. *reshape*
- Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ $J(\Theta)$
and $D^{(1)}, D^{(2)}, D^{(3)}$
Unroll _____ to get gradientVec.

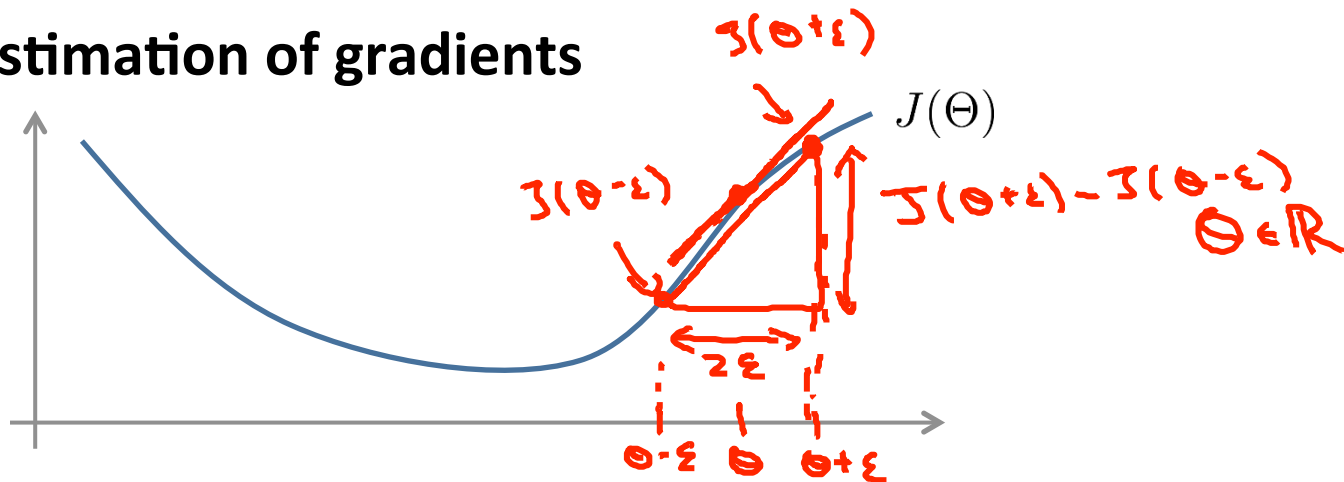


Neural Networks: Learning

Gradient checking

There's an idea called gradient checking that eliminates almost all of these problems. So, today every time I implement back propagation or a similar gradient to a [INAUDIBLE] on a neural network or any other reasonably complex model, I always implement gradient checking. And if you do this, it will help you make sure and sort of gain high confidence that your implementation of forward and back propagation or whatever is 100% correct. And from what I've seen this pretty much eliminates all the problems associated with a sort of a buggy implementation as a back propagation.

Numerical estimation of gradients



$$\frac{d}{d\Theta} J(\Theta) \approx$$

$$\frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

$$\epsilon = 10^{-4}$$

~~$$\frac{J(\Theta + \epsilon) - J(\Theta)}{\epsilon}$$~~

Implement: gradApprox = (J(theta + EPSILON) - J(theta - EPSILON)) / (2*EPSILON)

Parameter vector θ

→ $\theta \in \mathbb{R}^n$ (E.g. θ is “unrolled” version of $\underline{\Theta^{(1)}}$, $\underline{\Theta^{(2)}}$, $\underline{\Theta^{(3)}}$)

→ $\theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n]$

→ $\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$

→ $\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon}$

⋮

→ $\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}$

```

for i = 1:n, ←
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    / (2*EPSILON);
end;

```



$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i + \epsilon \\ \vdots \\ \theta_n \end{bmatrix} \rightarrow \theta_i - \epsilon$

$\frac{2}{2\theta_i} J(\theta)$

Check that gradApprox \approx DVec ←

↑
From back prop.

Implementation Note:

- - Implement backprop to compute DVec (unrolled $D^{(1)}$, $D^{(2)}$, $D^{(3)}$).

- - Implement numerical gradient check to compute gradApprox.
- - Make sure they give similar values.
- - Turn off gradient checking. Using backprop code for learning.


Important:

- - Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(...)`) your code will be very slow.



Machine Learning

Neural Networks: Learning

Random initialization

Initial value of Θ

For gradient descent and advanced optimization method, need initial value for Θ .

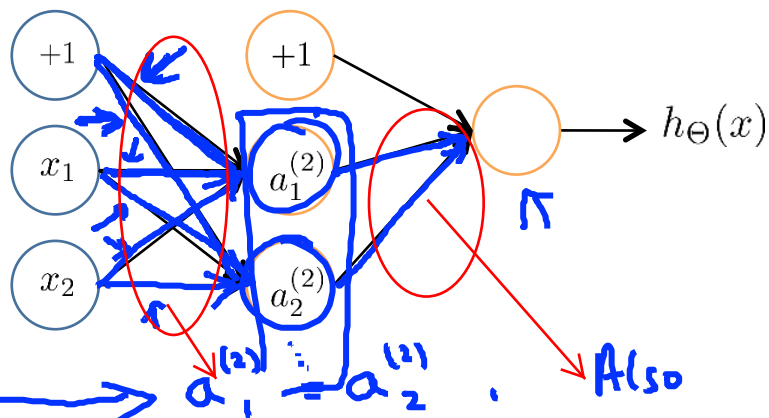
```
optTheta = fminunc(@costFunction,  
    initialTheta, options)
```

Consider gradient descent

Set initialTheta = zeros(n,1) ?

Whereas this worked okay when we were using logistic regression, initializing all of your parameters to zero actually does not work when you are training on your own network.

Zero initialization



$$\rightarrow \Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l.$$

$$\frac{\partial}{\partial \Theta_{0,1}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{0,1}^{(1)}} J(\Theta)$$

$$\Theta_{0,1}^{(1)} = \Theta_{0,2}^{(1)}$$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

$$\underline{a_1^{(2)} = a_2^{(2)}}$$

Random initialization: Symmetry breaking

→ Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

Random 10x11 matrix (betw. 0 and 1)

→ Theta1 = rand(10, 11) * (2 * INIT_EPSILON)
- INIT_EPSILON; [-ε, ε]

→ Theta2 = rand(1, 11) * (2 * INIT_EPSILON)
- INIT_EPSILON;

this epsilon here has nothing to do with the epsilon that we were using when we were doing gradient checking

Consider this procedure for initializing the parameters of a neural network:

1. Pick a random number $r = \text{rand}(1,1) * (2 * \text{INIT_EPSILON}) - \text{INIT_EPSILON}$;
2. Set $\Theta_{ij}^{(l)} = r$ for all i, j, l .

Does this work?

- ☐ Yes, because the parameters are chosen randomly.
- ☐ Yes, unless we are unlucky and get $r=0$ (up to numerical precision).
- ☐ Maybe, depending on the training set inputs $x(i)$.
- ☒ No, because this fails to break symmetry.

Correct



Machine Learning

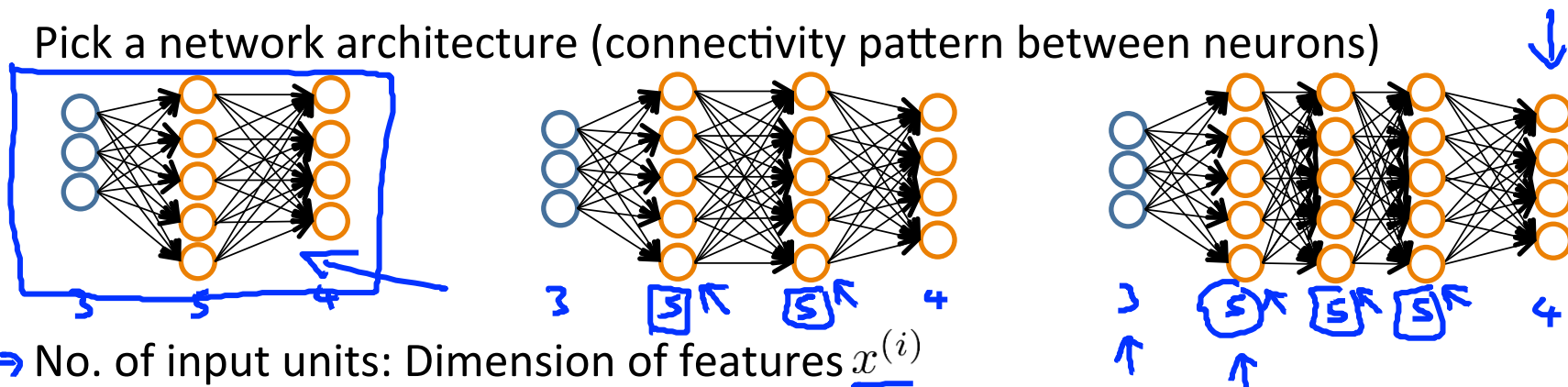
Neural Networks: Learning

Putting it together

And usually the number of hidden units in each layer will be maybe comparable to the dimension of x , comparable to the number of features, or it could be anywhere from same number of hidden units of input features to maybe so that three or four times of that.

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



→ No. of input units: Dimension of features $x^{(i)}$

→ No. output units: Number of classes

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

$$y \in \{1, 2, 3, \dots, 10\}$$

~~$y=5$~~



$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\text{OR} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Training a neural network

- 1. Randomly initialize weights
- 2. Implement forward propagation to get $h_{\Theta}(x^{(i)})$ for any $x^{(i)}$
- 3. Implement code to compute cost function $J(\Theta)$
- 4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

→ for $i = 1:m$ { $(x^{(1)}, y^{(1)})$ $(x^{(2)}, y^{(2)})$, ..., $(x^{(m)}, y^{(m)})$ }

→ Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$

(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \dots, L$).

$$\Delta^{(2)} := \Delta^{(2)} + \delta^{(L)} (a^{(2)})^T$$

compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.



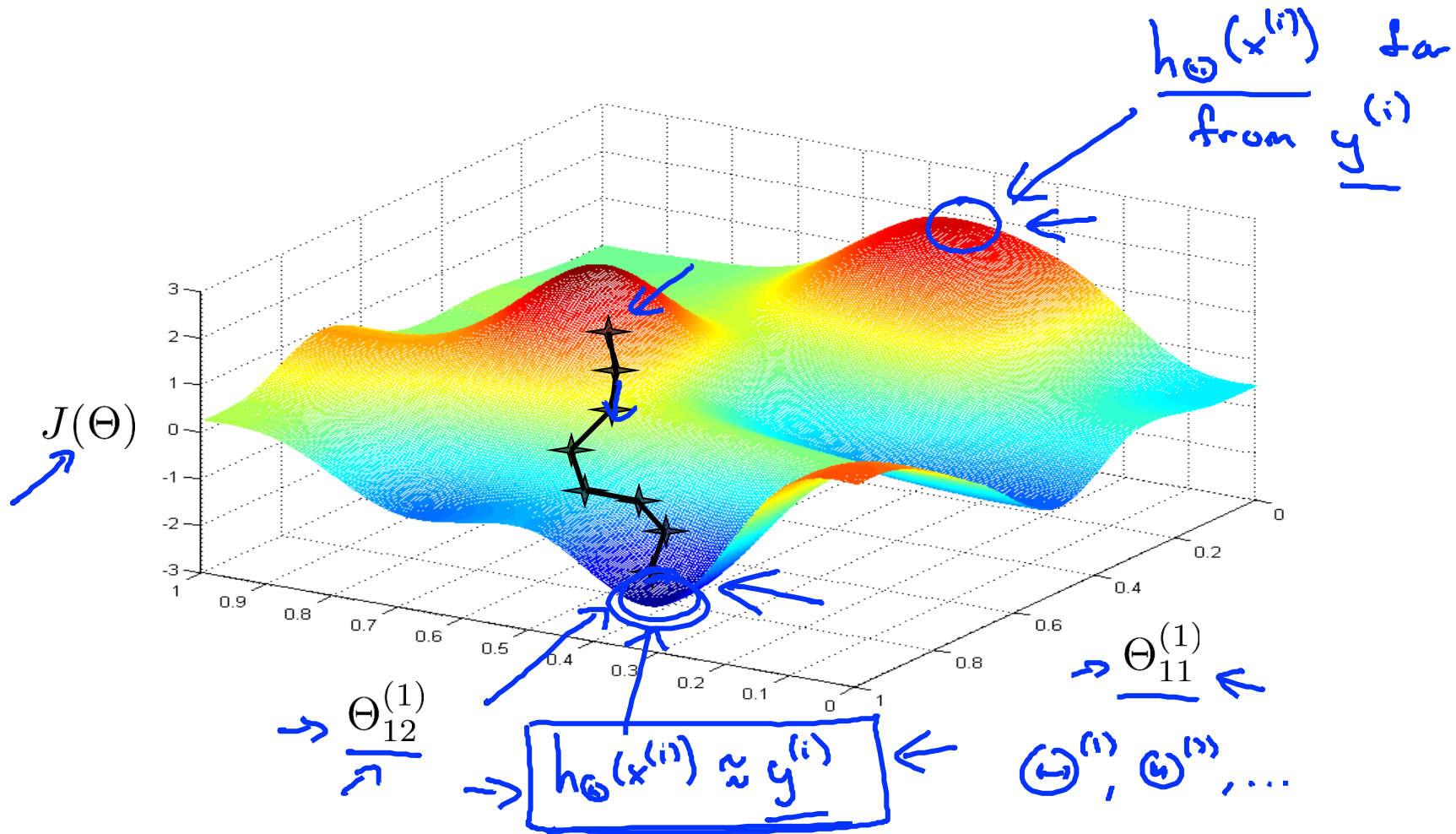
Training a neural network

- 5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.
- Then disable gradient checking code.
- 6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

$\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

$J(\Theta)$ — non-convex

for neural networks, this cost function j of θ is non-convex, or is not convex and so it can theoretically be susceptible to local minima, and in fact algorithms like gradient descent and the advance optimization methods can, in theory, get stuck in local optima, but it turns out that in practice this is not usually a huge problem and even though we can't guarantee that these algorithms will find a global optimum, usually algorithms like gradient descent will do a very good job minimizing this cost function j of θ and get a very good local minimum, even if it doesn't get to the global optimum.



Suppose you are using gradient descent together with backpropagation to try to minimize $J(\Theta)$ as a function of Θ . Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

- ☐ Plot $J(\Theta)$ as a function of Θ , to make sure gradient descent is going downhill.
- ☐ Plot $J(\Theta)$ as a function of the number of iterations and make sure it is increasing (or at least non-decreasing) with every iteration.
- ☒ Plot $J(\Theta)$ as a function of the number of iterations and make sure it is decreasing (or at least non-increasing) with every iteration.

Correct

- ☐ Plot $J(\Theta)$ as a function of the number of iterations to make sure the parameter values are improving in classification accuracy.



Machine Learning

Neural Networks: Learning

Backpropagation
example: Autonomous
driving (optional)

