⇒ Backpropagation intuition

→ Backpropagation is actually doing something very similar to forward propagation, but backwards!

→ The cost func^n for our NN is:

$$J(\theta) = -\frac{1}{m}\sum_{t=1}^{m}\sum_{k=1}^{K}\left[y_k^t \log\left(h_\theta(x^t)\right)_k + (1-y_k^t)\log\left(1-(h_\theta(x^t))_k\right)\right]$$

$$+\frac{\lambda}{2m}\sum_{L=1}^{L-1}\sum_{i=1}^{s_L}\sum_{j=1}^{s_{L+1}}\left(\theta_{ji}^{L}\right)^2$$

→ If we consider simple non-multiclass classification (K=1), & disregard regularization, the cost is computed with:

cost(t)

$$J(\theta) = y \log\left(h_\theta(x)\right) + (1-y)\log\left(1-h_\theta(x)\right)$$

for one training example.

* We can think about a δ term as the "error" of cost for the activation value associated with a unit.

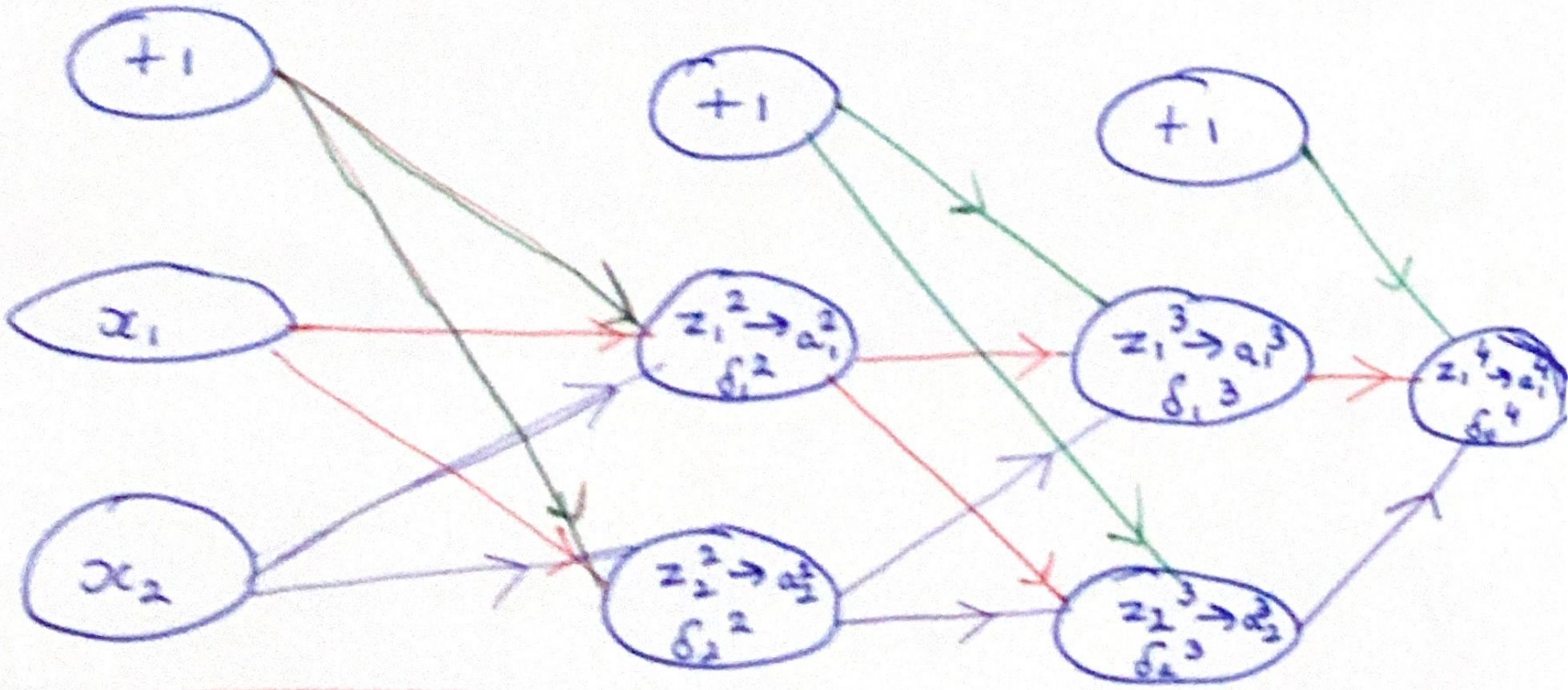
→ More formally,

$$\delta_j^l = \frac{\partial J(\theta)}{\partial z_j^l}$$

* For the output layer $\boxed{\delta^L = a - y}$

We propagate these values backwards.

eg:



$\delta_1^4 = y - a^4$ ∴ $\delta_1^3 = \theta_{11}^3 \delta_1^4$

$$\delta_2^{\,3} = \theta_{12}^{\,3}\ \delta_1^{\,4}$$

→ Continuing to prev. layers :

$$\delta_1^{\,2} = \theta_{11}^{\,2}\ \delta_1^{\,3} + \theta_{21}^{\,2}\ \delta_2^{\,3}$$

& $$\delta_2^{\,2} = \theta_{12}^{\,2}\ \delta_1^{\,3} + \theta_{22}^{\,2}\ \delta_2^{\,3}$$

→ So, backpropagation algo :

* Calculates the δ,

* And these δ values are the weighted sum of the next layer's delta values.

* δ^l corresponds to nodes & Δ^L corresponds to weights

⇒ Implementation note: Unrolling/Flattening parameters (matrices)

→ With neural networks, we are working with sets of matrices:

$$\Theta^1, \Theta^2, \Theta^3, \ldots$$

$$D^1, D^2, D^3, \ldots$$

→ In order to use optimizing func^{n}s such as fminunc(), we'll want to "unroll" or "flatten" all the elements into a one long vector:

```
thetaVector = [Theta1(:); Theta2(:); Theta3(:); ...]

deltaVector = [D1(:); D2(:); D3(:); ...]
```

→ We can get data back of our original matrices by using a bit of slicing & "reshape" func^n.

eg: if dim (Theta1) = 10X11

, dim (Theta2) = 10X11

& dim (Theta3) = 1X11

then we can get back our original matrices as:

Theta 1 = reshape (thetaVector (1:110), 10,11)

Theta 2 = reshape (thetaVector (111:220), 10,11)

Theta3 = reshape (thetaVector (221:231), 1,11)

→ By doing this, our learning algo for the above eg. becomes:

Having initial parameters θ^1, θ^2, θ^3

Unroll/flatten to get "initial Theta" to pass to

fminunc (@ Cost Function, initialTheta, options)

function [jval, gradientVec] = costFunction(thetaVec)

From thetaVec, get Θ^1, Θ^2, Θ^3 back using reshape

Use forward prop & backward prop to compute

D^1, D^2, D^3 & $J(\Theta)$

Unroll/Flatten D^1, D^2, D^3 to get gradientVec

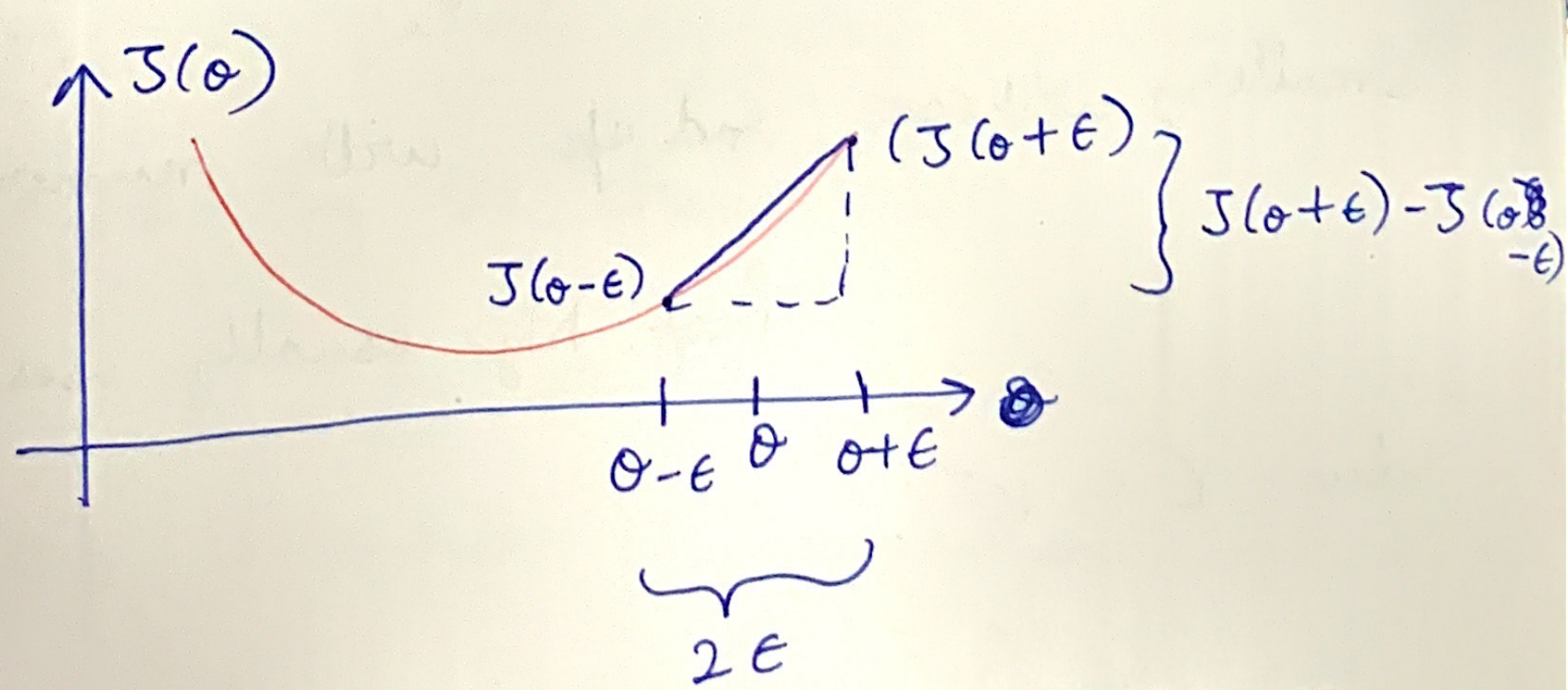
⇒ Gradient checking

→ Backpropagation can be messy! It is easy to miss ~~look over~~ some imp. details and while implementing it ~~is~~ ~~easy~~ & face bugs.

~~eg:~~ This may mean that

→ Gradient checking will assure that our backpropagation works as intended.

→ Let $J(\theta)$ be a single valued func^n only dependent on θ. Supp. $J(\theta)$ looks like:



If $\epsilon \to 0$ ∴ $\frac{dJ(\theta)}{d\theta} \approx \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$

* This is the 2 sided difference & is more accurate than a 1 sided difference $\left[\frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J(\theta^{0} + \epsilon) - \partial J(\theta)}{\epsilon}\right]$

* With multiple theta matrices, we can approximate the derivative wrt θ_j as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} \approx \frac{J(\theta_1, \ldots, \theta_j + \epsilon, \ldots, \theta_m) - J(\theta_1, \ldots, \theta_j - \epsilon, \ldots, \theta_m)}{2\epsilon}$$

* A good small value for ϵ guarantees the math above to become true. If the value be much smaller, will cause end up with numerical problems. Prof. Ng usually uses the value $\epsilon = 10^{-4}$.

* Using the prev. algs, we can manually (or rather laboriously) calculate the value of $\frac{\partial J(\theta)}{\partial \theta_{ij}^{l}}$ for each & every weight in our network!

We can see why we need backpropagation & why it is so efficient.

Backpropagation algs

Backpropagation works by computing the gradient of the loss func^{n} wrt each weight by the chain rule.

By iterating backwards from the last layer, it computes the gradient one layer at a time using the gradients of the next layer (Rem. we're iterating backwards!), it removes the redundant calculations of the intermediate terms.

* Backpropagation can is a clear example of dynamic programming!

* Manually calculating gradient can be slow but it helps us in stress testing our backpropagation algo, to check whether backprop is calculating the right gradient values or not.

* We can implement this "slow" computation as follows:

Let there be n no. of parameters in our cost function i.e. $J(\theta) = J(\theta_1, \ldots, \theta_n)$

```
epsilon = 1e^-4;
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) += epsilon;
    thetaMinus = theta;
    thetaMinus(i) -= epsilon;
    gradApprox(i) = [J(thetaPlus) - J(thetaMinus)] / 2*epsilon;
end
```

→ We earlier saw how we can compute the "D" matrix using backpropagation.

So, once we compute our gradApprox vector, we can check that whether $D == gradApprox$ or not.

→ Once, you've verified that backprop is working correctly; don't forget to turn off the code for computing gradApprox (As it is very computationally expensive)

→ Initial value of θ : For gradient descent & advanced optimization method, we need to set some initialTheta value

→ Zero initialization : Initializing all theta weights to 0 doesn't work with neural networks. When we back propagate, all nodes will update to the same value repeatedly resulting in same values of $\frac{\partial J(\theta)}{\partial \theta_{ij}^{l}}$ for each iteration of gradient descent.

And if the value of all the weights does not change are same, that means as all hidden layers are computing the same exact feature. which will make this prevents our NN is pushing out complex hypothesis.

→ Random initialization : Symmetry breaking :

* Initialize each Θ_{ij}^{l} to a random value in ~~[0, ε]~~

~~i.e.~~ $[-\epsilon_{init}, \epsilon_{init}]$ (i.e. $-\epsilon_{init} \leq \Theta_{ij}^{l} \leq \epsilon_{init}$)

* ~~Explain~~

$$\Theta_{\text{●●}}^{l} = 2\,\epsilon_{init} \times \underbrace{\text{rand}(s_{l+1}, \text{●} (s_l + 1))}_{\text{Max value} = 1,\ \text{Min} = 0} - \epsilon_{init}$$

The above expression keeps in check that values of Θ^{l} lies in the range $[-\epsilon_{init}, \epsilon_{init}]$

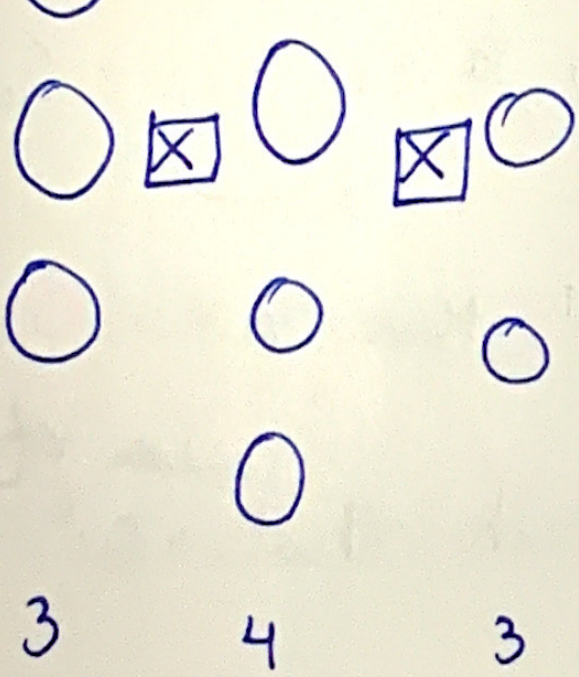
* Typically, we take ϵ as a small value like 0.1.

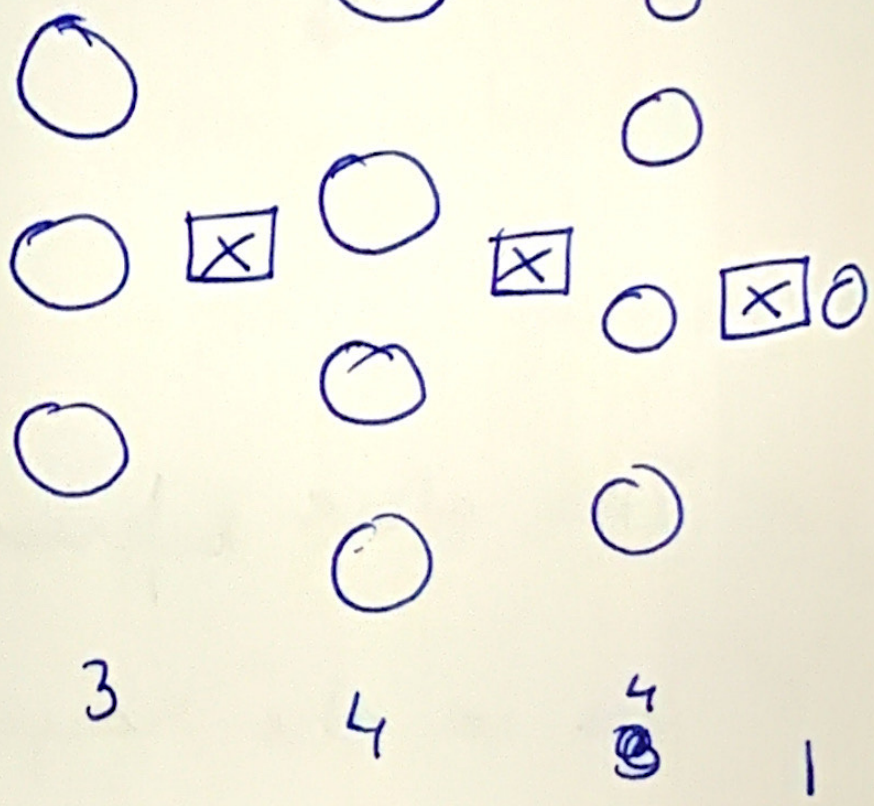
Because, if the value is too large, then our sigmoid activation keep putting out near to 1 values for each output (because of how large our weighted sum becomes).

Putting it together

⇒ training a neural network

→ Pick a network architecture (connectivity pattern b/w neurons)





→ No. of input units = dimensions of features x^i

→ No. of output units = no. of classes

→ No. of hidden units per layer: Usually more the better. (Can increase computational complexity)

→ Rule: If more than 1 hidden layer, then assign the same no. of units in every

⇒ Training a neural network

1) Randomly initialize the weights.

2) Implement forward propagation to get $h_\theta(x^i)$ for any x^i out of our training examples.

3) Implement code to compute cost function $J(\theta)$

4) Implement backprop to compute partial derivatives $\frac{\partial}{\partial \theta_{ij}^l} J(\theta)$

→ When we perform forward & backward propagation, we loop on every training example:

```
for i = 1 : m {
    Forward propagation on (x^i, y^i) // Get a^(l) term
    Back propagation on (x^i, y^i) // Get δ terms
    Compute Δ := Δ^l + δ^(l+1) (a^l)^T
}
```

5) Use gradient checking to compare $\frac{\partial J(\theta)}{\partial \Theta_{ij}^{l}}$

computed using backpropogation vs. using numerical ~~estimate~~ of gradient of $J(\theta)$.

Then disable gradient checking

6) Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\theta)$ as a function of parameters θ.

* In neural nets, $J(\theta)$ is a non convex fun^{n} s thus we can end up in a local ~~minimum~~ minimum instead but it ~~works~~ even that might work very well for our problem.