-w5- Backpropagation Algorithm-

Aslında bunun vein değil. Bunu sadece Gradient termsü bulmak için kullandık!

An algorithm for minimizing the Cost Function!

→ Geçen derste $J(\Theta)$ formunu yazdık.

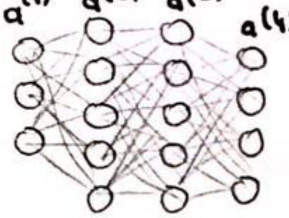
- Yapmamız gerekeni Find parameters Θ minimizes $J(\Theta)$
- GDA veya Adv. Opt. Alg.s kullanabilmek için aşağıdaki iki kavramı hesaplayabiliyor olmalıyız:
  - $\boxed{J(\Theta)}$ ✓Bunu biliyoruz.
  - $\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)}$ where $\Theta_{ij}^{(l)} \in \mathbb{R}$

Bunu nasıl hesaplarız ona bakacağız:

Let's assume we have only one training ex: (x, y):

- Let's look at the sequence of calculations:
  - **First: Forward Propagation** ile inputa karşılık gelen $h(x)$'i bulunuz:

$a^{(1)}$ $a^{(2)}$ $a^{(3)}$ $a^{(4)}$



Forward Propagation

→ $a^{(1)} = x$ → $z^{(2)} = \Theta^{(1)} \cdot a^{(1)}$ → $a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

→ $z^{(3)} = \Theta^{(2)} \cdot a^{(2)}$ → $a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

→ $z^{(4)} = \Theta^{(3)} \cdot a^{(3)}$ → $a^{(4)} = h_\Theta(x) = g(z^{(4)})$

Remember, $a^{(2)} = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ a_4^{(2)} \\ a_5^{(2)} \end{bmatrix}$

- **Next: Backpropagation Algorithm** in order to compute the derivative terms or Gradients:

Yeni bir kavram tanımlıyoruz "$\delta_J^{(l)}$" her node için tanımlı. l. layerdaki J. unitin "Activation Error"ü ($a_J^{(l)}$ l. layer J. unitin activationı)

For each output unit (layer 4): $\boxed{\delta_J^{(4)} = a_J^{(4)} - y_J = (h(x))_J - y_J}$

vectorized form $\boxed{\delta^{(4)} = a^{(4)} - y}$

Then for other layers:

g': der. of the activation func. vector of ones.

$\boxed{\delta^{(3)} = (\Theta^{(3)})^T \cdot \delta^{(4)} .* g'(z^{(3)})}$ where $\boxed{g'(z^{(3)}) = a^{(3)} .* (1 - a^{(3)})}$

$\boxed{\delta^{(2)} = (\Theta^{(2)})^T \cdot \delta^{(3)} .* g'(z^{(2)})}$ where $\boxed{g'(z^{(2)}) = a^{(2)} .* (1 - a^{(2)})}$

$\boxed{\text{No } \delta^{(1)} \text{ term! Input'un Error'u olmaz.}}$

Finally: $\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_J^{(l)} \cdot \delta_i^{(l+1)}}$ → (ignoring reg. terms or assuming $\lambda = 0$)

W5 - Backpropagation Algorithm II

Geçen sayfada anlatılanlar biraraya getirilip, Backpropagation Algorithmin derivative terms hesaplamak için nasıl kullanıldığına bakılırsa:

- Training Set: $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j) (Used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)
- For $i = 1 : m$
  - Set $a^{(1)} = x^{(i)}$
  - Perform Forw. Prop. to compute $a^{(l)}$ for $l = 2, 3, \ldots L$
  - Using $y^{(i)}$, compute $\boxed{\delta^{(L)} = a^{(L)} - y^{(i)}}$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ (there is no $\delta^{(1)}$)
  - $\underbrace{\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \cdot \delta_i^{(l+1)}}$

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

end

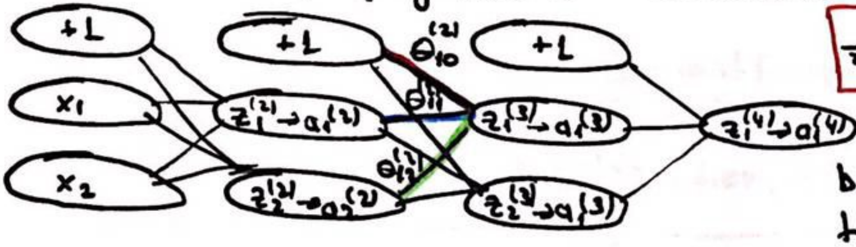
- $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

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

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

-W5- Backpropagation Intuition-

- Backpropagation'ın mekaniğini daha iyi kavrayabilmek için öncelikle forward-propagation'a bakalım.

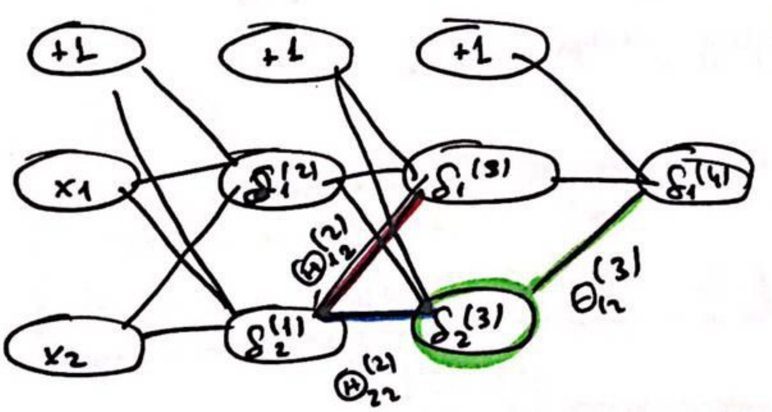


$$\boxed{z_1^{(3)} = \Theta_{10}^{(2)}\cdot 1 + \Theta_{11}^{(2)}\cdot a_1^{(2)} + \Theta_{12}^{(2)}\cdot a_2^{(2)}}$$

Backpropagation da buna benzer çalışıyor fakat computation sağdan sola doğru oluyor.

- $J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m} cost(i)$ gibi düşünülebilir. cost(i) burada single training ex.'ın hatası! Normalde logarithmic formda fakat intuition dan $cost(i) \approx (h(x) - y(i))^2$ gibi düşünebiliriz.

- $\delta_j^{(l)}$ = "error" of cost for $a_j^{(l)}$ (unit j in layer l)
  - Formally $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$ (for $j \geq 0$).



$$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$$

$$\delta_2^{(2)} = \underline{\Theta_{12}^{(2)}\cdot\delta_1^{(3)}} + \underline{\Theta_{22}^{(2)}\cdot\delta_2^{(3)}}$$

$$\delta_2^{(3)} = \Theta_{12}^{(3)}\cdot\delta_1^{(4)}$$

- Bazen bias units için de δ'lar hesaplanır ancak kullanılmaz çünkü anlamsız!

W5 - Implementation Note: Unrolling Parameters from matrices to vectors.

- Bu unrolling olayına advanced optimization algoritmaları için gerek olacak
  Çünkü bunlar input olarak vektörleri alıyor!

Advanced Optimization

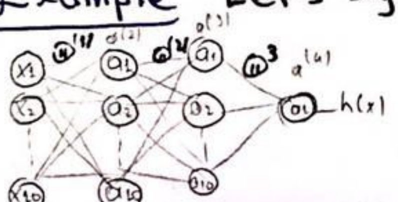
function [jVal, gradient] = costFunction(theta)
$\hookrightarrow \mathbb{R}^{n+1}$ vector $\rightarrow \mathbb{R}^{n+1}$ vectors!
⋮
optTheta = fminunc(@costFunction, initialTheta, options)

- Yukarıdaki rutinler thetaların ve gradientlerin vector olduğunu assume eder, log. reg. için zaten böyleydi ama NNs için bunlar (parameters) matris.

- L=4 olan bir NN için
  - Parameters: $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)
  - Gradients: $D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D1, D2, D3)

- Bu videoda amaç NNs için Parameters ve Gradients'i nasıl matris formundan vector forma unroll edebileceğimizi öğrenmek:

Example Let's say we have NN with $s_1=10, s_2=10, s_3=10, s_4=1$



- $\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$ Parameters (Matrix)
- $D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$ Gradients (Matrix)

Unrolling

```
ThetaVec = [Theta1(:); Theta2(:); Theta3(:)];
DVec     = [D1(:); D2(:); D3(:)]
```

Bu şekilde matrisler açılıp ucuca eklenir ve sütun vektörleri elde edilmiş olur!

Roll Back

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

Where to use Unrolling / Rolling Parameters

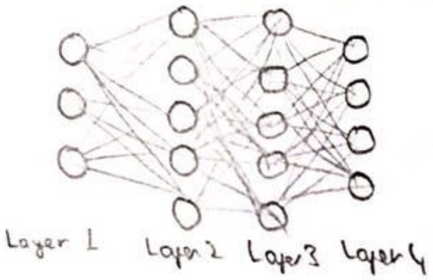
Neden? Çünkü Forward and Backprop. ve J(Θ) calculations is easier by matrices but opt. alg. need vector forms.

- Let's say we have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
- Unroll them to get "initialTheta" to pass to: fminunc(@costFunction, initialTheta, options)
- function [jVal, gradientVector] = costFunction(thetaVec)
  - From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ use forward/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$
  - Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVector

-W5- NNs Cost Function

- NNs are one of the most powerful Learning Algorithms we have today.
- Bu derslerde NN parametersi'ı training set'e fit etmeyi göreceğiz
  - We gonna focus on the application of NNs to classification problems. Yani başka uygulamalarda da kullanılabilir demekki.

Neural Network (Classification)



- Training Set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)})\}$
- m: # of training examples
- L: # of layers in network $(L = 4)$
- s_l: # of units (w/o bias) in layer l $(s_1 = 3, s_2 = 5 \dots)$

→ We are gonna consider 2 Types of Classification Problems:

Binary Classification:

- $y = 0$ or 1 $(h(x))$
- 1 output unit computes $h(x)$
- $h_\Theta(x) \in \mathbb{R}$ • $s_L = 1$ • $K = 1$

Multi-class Classification (K classes):

- $y \in \mathbb{R}^K$ $(h(x))$ E.g. $\begin{bmatrix}1\\0\\0\end{bmatrix}$ car $\begin{bmatrix}0\\1\\0\end{bmatrix}$ truck $\begin{bmatrix}0\\0\\1\end{bmatrix}$ bicycle
- $s_L = K$
- K output units (K classes)

Cost Function for NN: Generalization of the one we used for Log. Reg.

Cost Func. for Log. Reg:

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

For NN:

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

- $\boxed{h_\Theta(x) \in \mathbb{R}^K \text{ (K=1 if binary classification)}}$ • $\boxed{(h_\Theta(x))_i = i^{th} \text{ output}}$

(1) Instead of having 1 log. reg. output unit we have K of them.

(2) Basically we are summing log. reg. cost function over each of my four outputs. Her unit için y_k ile $h(x)_k$ karşılaştırılıyor.

Adv. Opt. Methods can

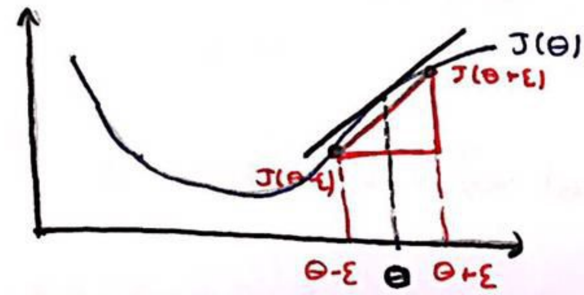
- $J(\Theta)$
- Partial Derivative Terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l — l here # of layers

W5 - NNs: Gradient Checking -

Backpropagation is a little tricky. Bazen işe yarıyormuş gibi görünür hatta $J(\Theta)$ giderek azalır ama sonuçta $J(\Theta)$'yı minimize edemeden durabilir (backprop alg. daki buglardan dolayı)

Gradient Checking ile Forw/Back prop. implementationımızın 100% doğru olduğundan emin olabiliriz.

- Numerical Estimation of Gradients -



- Assume I have a cost function and $\Theta \in \mathbb{R}$.

Amacım numeric olarak Θ noktasında $J(\Theta)$'nın eğimini bulabilecek bir yaklaşım elde etmek.

$$\boxed{\frac{d}{d\Theta} J(\Theta) \approx \frac{J(\Theta+\varepsilon) - J(\Theta-\varepsilon)}{2\varepsilon}}$$

→ $\varepsilon \cong 10^{-4}$, zaten ε küçüldükçe $\approx$ giderek $=$'e yaklaşır. Türevin tanımı zaten bu.

Implement Code:

```
gradApprox = (J(theta+EPSILON) - J(theta-EPSILON)/(2*EPS.)
```

- General Case when Θ is a vector Parameter -

- $\Theta \in \mathbb{R}^n$ (E.g. Θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
  - $\Theta = [\Theta_1, \Theta_2, \Theta_3, \dots \Theta_n]$

- $$\frac{\partial}{\partial \Theta_1} J(\Theta) \approx \frac{J(\Theta_1+e, \Theta_2, \Theta_3, \dots, \Theta_n) - J(\Theta_1-e, \Theta_2, \Theta_3, \dots \Theta_n)}{2e}$$

- $$\frac{\partial}{\partial \Theta_2} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2+e, \Theta_3, \dots \Theta_n) - J(\Theta_1, \Theta_2-e, \Theta_3, \dots, \Theta_n)}{2e}$$

  ⋮

- $$\frac{\partial}{\partial \Theta_n} J(\Theta) \approx \frac{J(\Theta_1, \Theta_2, \dots, \Theta_n+e) - J(\Theta_1, \Theta_2, \dots, \Theta_n-e)}{2e}$$

↳ Derivative terms numerically computed.

W5 - NNs: Gradient Checking II -

- Biraz evvel açıklanan general case için aşağıdaki kod kullanılır:

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

$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i + \varepsilon \\ \vdots \\ \theta_n \end{bmatrix}$ → Thetaplus bu oldu.

gradApprox(i) $\approx \frac{\partial}{\partial \theta_i} J(\theta)$

- We implement this for loop to check that $\boxed{\text{gradApprox} \approx \text{Dvec}}$

from for loop (gradApprox) — from backpropagation (Dvec)

- Backpropagation ile Gradients hesaplamak relatively efficient. Numeric sonuç ile bunu karşılaştırarak backprop.'un doğru çalıştığından emin olabilirim.

- Putting Everything Together: Implementation -

- Implement backpropagation to compute $\boxed{\text{DVec}}$ (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$)
- Implement numerical gradient checking to compute $\boxed{\text{gradApprox.}}$
  - Make sure $\boxed{\text{gradApprox} \approx \text{DVec}}$
- Turn off gradient checking before seriously training the network. Because numerical gradient checking code is computationally expensive.

Yani anladığım kadarıyla sadece 1 tur için yani 1 thetaVec için numeric ve backprop ile elde edilen Grads karşılaştırılır. doğruluğundan emin olduktan sonra backprop. kullanılmaya devam edilir.

Important: Be sure to disable your gradient checking code before training the classifier. If you run gradient computation on every iteration of GDA (or in the inner loop of costFunc(...)) your code will be very slow.

-W5- Random Initialization -

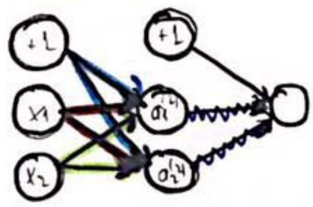
Initial Value of Θ : For GDA and Advanced Optimization Algorithms we need initial value for Θ.

optTheta = fminunc(@costFunction, initialTheta, options)

- Consider gradient descent algorithm,
  - Setting initialTheta = zeros(n,1) has worked for logistic regression but it does not work we are training a Neural Network!

Zero Initialization Example: Aşagidaki NN'i eğitmeye çalışıyoruz diyelim.

- $\Theta_{ij}^{(l)} = 0$ for all i, j, l olarak initialize edilirse;



→ h(x)

→ Tüm training examples için $\boxed{a_1^{(2)} = a_2^{(2)}}$

→ Ayrıca a1 ve a2'den çıkan $\Theta^{(2)}$'nin weightlerde yani $\Theta_1^{(2)}$ ve $\Theta_2^{(2)}$ de 0 olduğu için, $\boxed{\delta_1^{(2)} \text{ ve } \delta_2^{(2)} \text{ aynı!}}$

→ Ayrıca $\boxed{\frac{\partial}{\partial \Theta_{10}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{20}^{(1)}} J(\Theta)}$ bu da şu demek GDA updatelerden sonra bile $\boxed{\Theta_{10}^{(1)} = \Theta_{20}^{(1)}}$ olacak. Aynı renkli weights eşit olacak!

Sonuçta bu da şu demek sonuçta $\boxed{a_1^{(2)} = a_2^{(2)}}$ olacak! Redundant representation. 50 tane hidden unit olsa dahi hepsi aynı sonucu verecek.

Solution: Random Initialization: Symmetry Breaking

- Initialize $\Theta_{ij}^{(l)}$ to a random value in [-A, A] (i.e. $-A \le \Theta_{ij}^{(l)} \le A$)

random 10x11 matrix (between 0-1)

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

A usually small around zero

Peki zero initialization neden log. reg. sorun değildi? Çünkü cost function farklı olduğu için Gradda farklıydı ve her grad için x_j etkili oluyordu bu yüzden farklı sonuçlar geliyordu ama NN için grads backprop ile δ ile hesaplanır ve aynı ~~gelecekler~~

W5 - Putting All Together -

- How do we implement learning algorithm for NNs.

- When training a NN first thing we should do is pick a network architecture:
  - → # of input units: Dimension of features $x^{(i)}$
  - → # of output units: Number of classes

Reminder: Don't forget to write y in vector form like $y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ 6 class varsa ve bu output 3 demekse

→ Reasonable default: Using a single hidden layer, or if >1 hidden layer have same # of hidden units in every layer (usually the more the better) → more computationally expensive but rather than that it's a good thing.

Ex: 3-5-4, 3-5-5-4, 3-5-5-5-4.

Training a NN:

1) Randomly initialize weights (small values near zero)
2) Implement forward propagation to get $h(x^{(i)})$ for any $x^{(i)}$
3) Implement code to compute $J(\Theta)$
4) Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

- To implement backpropagation we usually use a for loop (advanced method var ama şimdi gerek yok)

```
for i=1:m {
    Perform forward/back prop. using example (x^(i), y^(i))
    Get all activations a^(l) and all delta terms δ^(l) for l=2,...,L
    Δ^(l) := Δ^(l) + δ^(l+1) . (a^(l))^T
    ---
}
---
compute ∂/∂Θ_ij^(l) J(Θ)
```

5) Use gradient checking to compare $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ computed using backprop gained vs. using numerical estimate of gradient of $J(\Theta)$. Then disable gradient checking code.

6) Using gradient descent algorithm or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ. → $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

(!) $J(\Theta)$ is NON-CONVEX for Neural Networks so GDA or other Adv.Opt.Alg can get stuck in a local optima. But in practice this's not usually a huge problem. Yani genelde iyi çalışır. Globali bulamasa bile yaklaşır iyi yanına