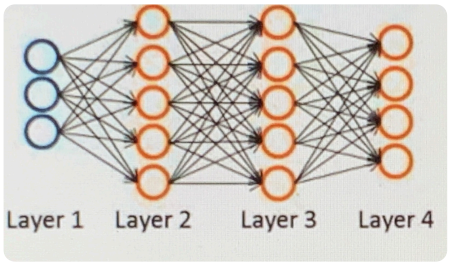
Stanford ML

WEEK 5 :

Neural Network Cost Function

CLASSIFICATION



- $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
- L = total no. of layers
- s_l = no. of units in layer l
  - ↳ NOT INCLUDING BIAS UNIT

BINARY

* $y \in \{0, 1\}$
* Only 1 OUTPUT unit
  - ↳ $h_\Theta(x) \in \mathbb{R}$
  - ↳ $S_L = 1$ and $K = 1$

MULTI-CLASS

* $y \in \mathbb{R}^K$ e.g. (CAR, TRUCK)

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

* K output units $\therefore S_L = K$
* $h_\Theta(x) \in \mathbb{R}^K$; $S_L = K$ $(K \geq 3)$
  - ↳ Only need to use 1 vs. all method if $K \geq 3$

- **J - Logistic Regression :**

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

↳ θ_0 parameter is not regularized by convention

J - NEURAL NETWORK

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

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

SUM OVER OUTPUT

OUTPUT UNITS

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left\{\Theta_{ji}^{(l)}\right\}^2\Bigg\}$$

REGULARIZATION

still not including BIAS UNIT

first layer

not regularized

- The number of columns in our current Θ matrix is equal to the number of nodes in our current layer (INCLUDING BIAS)
- The number of rows in our current theta matrix is equal to the number of nodes in the NEXT LAYER (EXCLUDING BIAS)

Backpropagation Algorithm

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

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left\{\Theta_{ji}^{(l)}\right\}^2$$

- In order to minimize the cost function we need to compute
  - → $J(\Theta)$
  - → $\partial / \partial \Theta_{ij}^{(l)} \; J(\Theta)$

how do we compute this?

Gradient Computation → USE BACKPROPAGATION

- Given one training example (x, y):

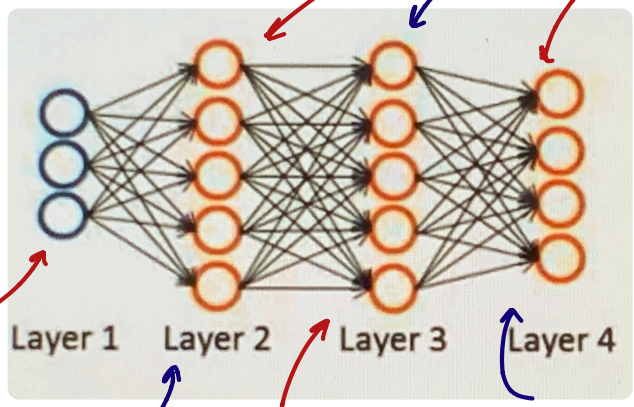
① FORWARD PROPAGATION:

* $a^{(1)} = x \;;\; z^{(2)} = \Theta^{(1)} a^{(1)}$

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

* $a^{(3)} = g(z^{(3)})$ { add $a_0^{(3)}$ } ; $z^{(4)} = \Theta^{(3)} a^{(3)}$

* $a^{(4)} = h_\Theta(x) = g(z^{(4)})$



② BACKPROPAGATION :

* *Intuition :* ACTIVATION

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

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

- $\delta_j^{(4)} = a_j^{(4)} - y_j = \{h_\Theta(x)\}_j - y_j$

$\underline{\delta}^{(4)} = \underline{a}^{(4)} - \underline{y} = \{h_\Theta(x)\} - \underline{y}$ } VECTORIZED

- $\delta^{(3)} = [\Theta^{(3)}]^T \delta^{(4)} \; .* \; g'(z^{(3)})$ ' = DERIVATIVE AT $z^{(3)}$

↑ element-wise multiplication

- $\delta^{(2)} = \Theta^{(2)T} \delta^{(3)} \; .* \; g'(z^{(2)})$

- NO $\delta^{(1)}$ TERM

$g'(\underline{z}^{(i)}) = \underline{a}^{(i)} \; .* \; (1 - \underline{a}^{(i)})$

∴ Ignoring the regularization term ($\lambda = 0$)

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

BACKPROPAGATION

must do seperately for each training example

Overall:

- Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- Set $\Delta_{ij}^{(l)} = 0 \quad \forall\, i, j, l$ {used to compute DERIV.}
  - ↳ ACCUMULATOR
- Loop through training set
  - $a^{(1)} = x^{(i)}$
  - forward propagation for $a^{(l)}$ $l = 2, 3, \ldots, L$
  - Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$ ↳ TARGET LABEL
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  - $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a^{(l)} \delta_i^{(l+1)}$
    - ↳ $\Delta^{(l)} := \Delta^{(l)} \delta^{(l+1)} (a^{(l)})^T$ } VECTORIZED
- Come out of for loop

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

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

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

$\delta_j^{(l)}$ = "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

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

↳ without regularization

Implementing Backpropagation

UNROLLING Θ MATRIX $\longrightarrow$ VECTORS

- Allows advanced optimization methods to be used

CODE

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

theta: $\mathbb{R}^{n+1}$ VECTORS

POINTER

initial theta values

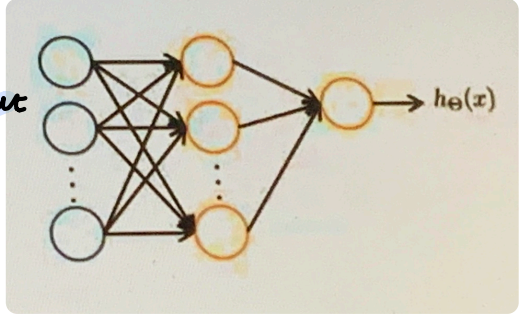
- Neural Network ($L = 4$)

* $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ – matrices (Theta1, Theta2, Theta3)
* $D^{(1)}, D^{(2)}, D^{(3)}$ – matrices (D1, D2, D3)

$\longrightarrow$ "UNROLL Θ, D" into VECTORS

Example:

- $s_1 = 10, s_2 = 10, s_3 = 1$ | input, hidden, output



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

UNROLLING:

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

ROLLING:

$\Theta^{(1)} \in \mathbb{R}^{10 \times 11}$ $\longrightarrow$ 110 elements

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

$\Theta^{(3)} \in \mathbb{R}^{1 \times 11}$

$\Theta \in \mathbb{R}$

Learning Algorithm

1.) Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

2.) Unroll to get initialTheta to pass to fminunc(@costFunction, initialTheta, options)

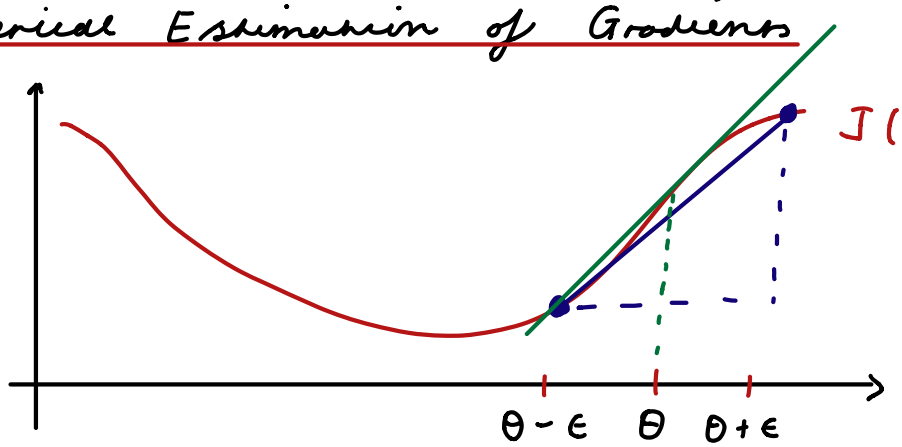
3.) Implement cost function

```
function [jval, gradientVec] = costFunction(thetaVec)
    From thetaVec, get Θ(1), Θ(2), Θ(3).   } USE RESHAPE
    Use forward prop/back prop to compute D(1), D(2), D(3) and J(Θ)
    Unroll D(1), D(2), D(3) to get gradientVec.
```

Gradient Checking

- Finding bugs in the NEURAL NET
- It is possible to write a BUGGY, but WORKING ANN.

Numerical Estimation of Gradients



$\Theta \in \mathbb{R}$

— ACTUAL GRAD

— APPROX. GRAD

$$\therefore \quad \frac{d}{d\theta} J(\theta) \approx \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$$

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

TWO SIDED DIFFERENCE

- IMPLEMENT:

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

- Consider when $\theta \in \mathbb{R}^n$ (vector)
  - Unrolled $(\theta^{(1)}, \theta^{(2)}, \theta^{(3)})$

$$\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{J(\theta_1, \theta_2, \ldots \theta_i + \epsilon, \ldots \theta_n) - J(\theta_1, \theta_2, \ldots \theta_i - \epsilon, \ldots \theta_n)}{2\epsilon}$$

IMPLEMENT

Unrolled theta $(\theta^{(1)}; \theta^{(2)}; \theta^{(3)})$

```
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;
```

* CHECK THAT gradApprox ≈ DVec

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.

↳ BEFORE TRAINING NETWORK = GRADIENT DESCENT

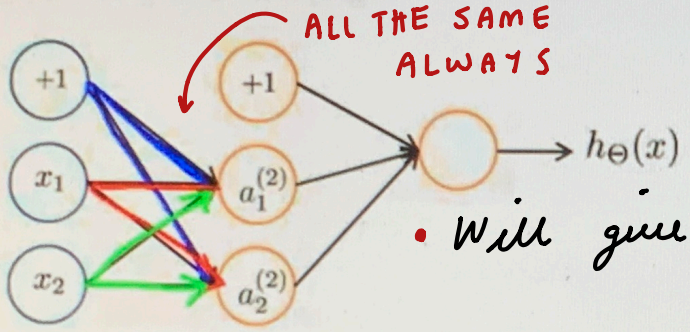
Important:

- 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.

Initial Θ Value

- Set `initialTheta = zeros(n,1)` DOES NOT WORK when training ANNs

Zero initialization



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

- Will give $a_1^{(2)} = a_2^{(2)}$ and $\delta_1^{(2)} = \delta_2^{(2)}$

- Overall:

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J = \frac{\partial}{\partial \Theta_{02}^{(1)}} J \Rightarrow \Theta_{01}^{(1)} = \Theta_{02}^{(1)} \neq 0$$

∴ After each update, parameters corresponding to inputs going into each of 2 hidden units are identical

- All units therefore end up representing a single feature

∴ **RANDOM INITIALIZATION ⟶ SYMMETRY BREAKING**

* Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$

$\in \mathbb{R}^{10\times 11}$ $[0,1]$ → $\in \mathbb{R}^{10\times 11}$ $[-\epsilon, \epsilon]$

10 x 11

1 x 11

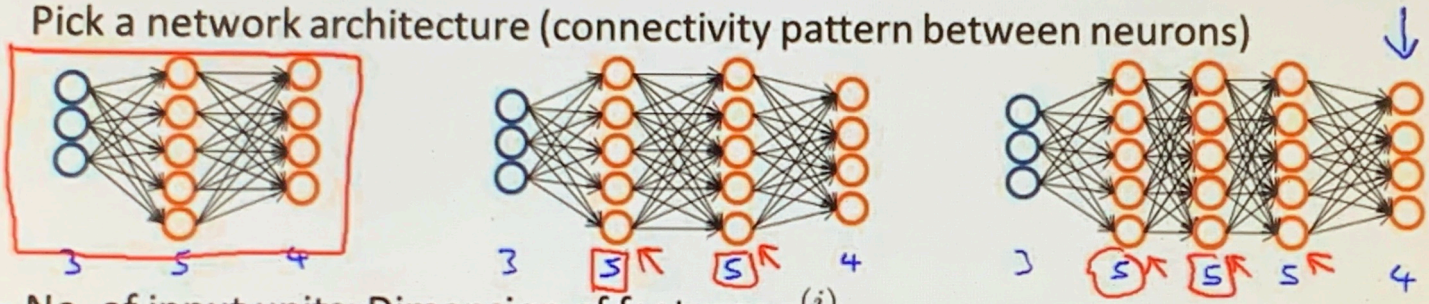
```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;

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

Putting it all Together

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)

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

~~$y = 5$~~

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

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
```

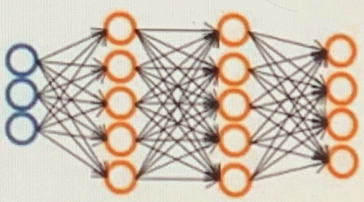
Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$

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

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

}

COMPUTE $\partial / \partial \Theta_{ij}^{(l)} J(\Theta)$



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.