Neural networks: learning

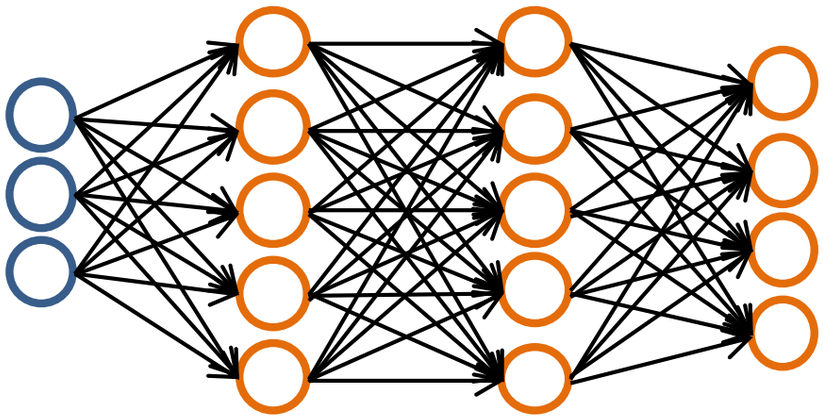
Andrew Ng

Neural networks:

learning

cost function

Neural Network (Classification)



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

$L =$ total no. of layers in network

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

Binary classification

$y = 0$ or 1

1 output unit

Multi-class classification (K classes)

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

pedestrian car motorcycle truck

K output units

Cost function

Logistic regression:

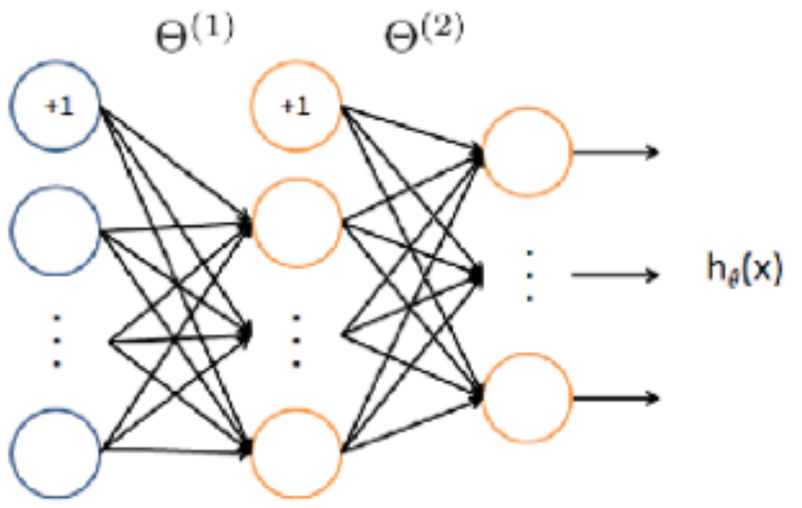
$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Neural network:

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

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

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$



Neural networks: learning

Backpropagation algorithm

Gradient computation

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

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_j^{(l)})^2$$

$$\min_{\Theta} J(\Theta)$$

Need code to compute

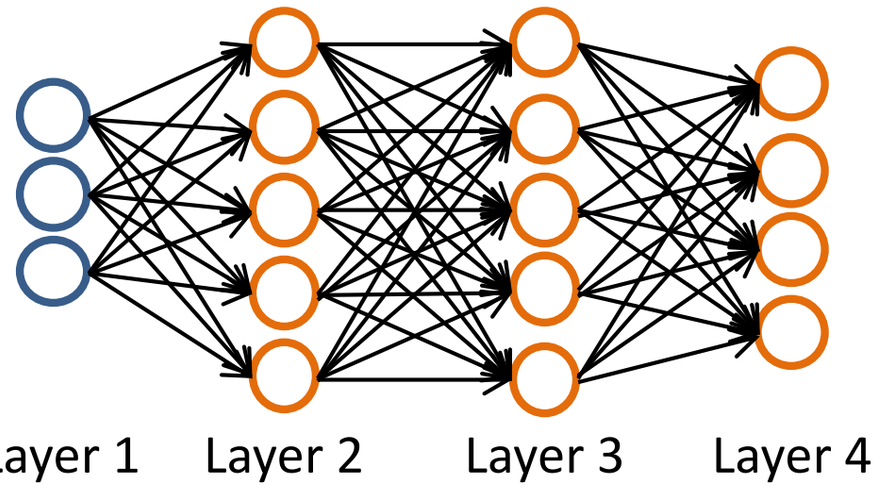
$$J(\Theta) \qquad \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

Gradient computation

Given one training example (x, y):

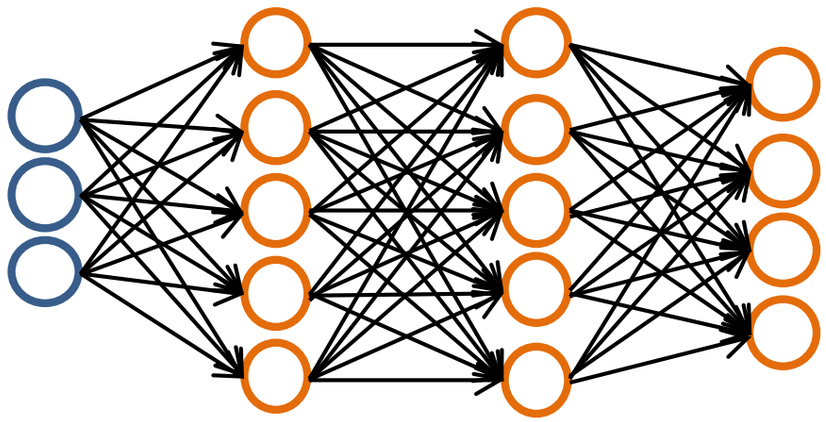
Forward propagation:

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



Gradient computation: Backpropagation algorithm

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



For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j \quad \text{vectorized: } \delta^{(4)} = a^{(4)} - y$$

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

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)}) \qquad g'(z^{(l)}) = a^{(l)} .* (\vec{1} - a^{(l)})$$

Gradient computation

$$\min_{\Theta} J(\Theta)$$

Need code to compute

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

Gradient computation:

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) = \frac{1}{m} \sum_{k=1}^{m} (a_j^{(l)(k)} \delta_i^{(l+1)(k)}) \text{ (if } \lambda = 0)$$

Gradient Computation

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

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

For $k = 1$ to m

Set $a^{(1)} = x^{(k)}$

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

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

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

for all l, i, j

$$\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

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

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

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) = \frac{1}{m} \sum_{k=1}^{m} (a_j^{(l)(k)} \delta_i^{(l+1)(k)})$$

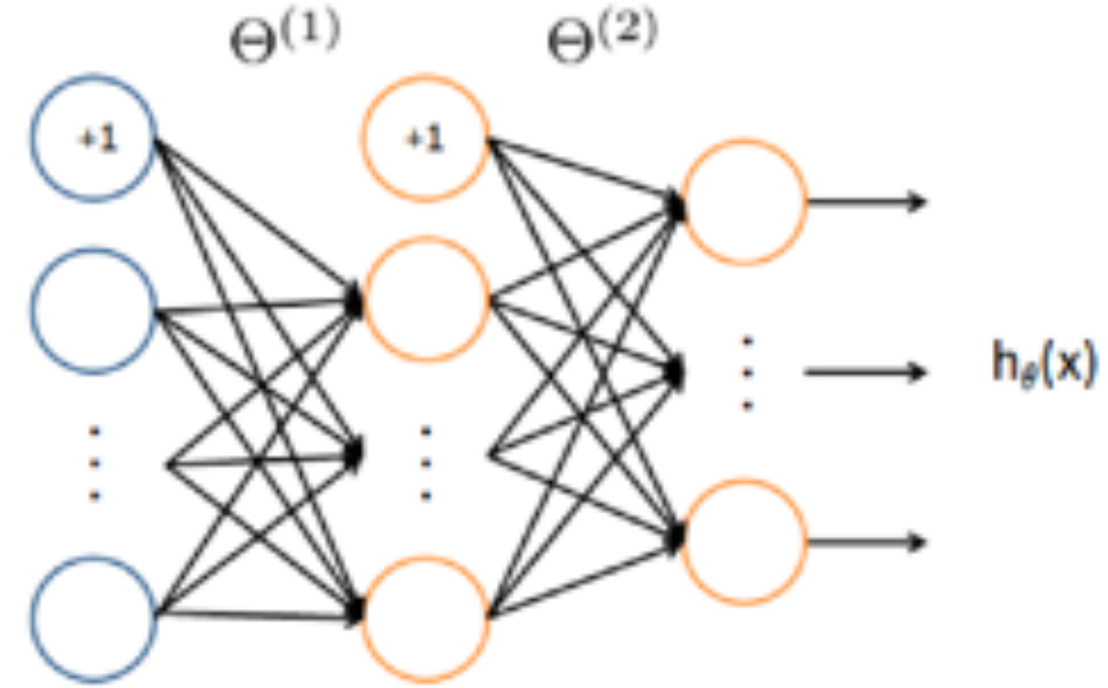
vectorized, for all l

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

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

Neural networks: learning

Gradient computation example



$a^{(1)} = x$ (add $a_0^{(1)}$)

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

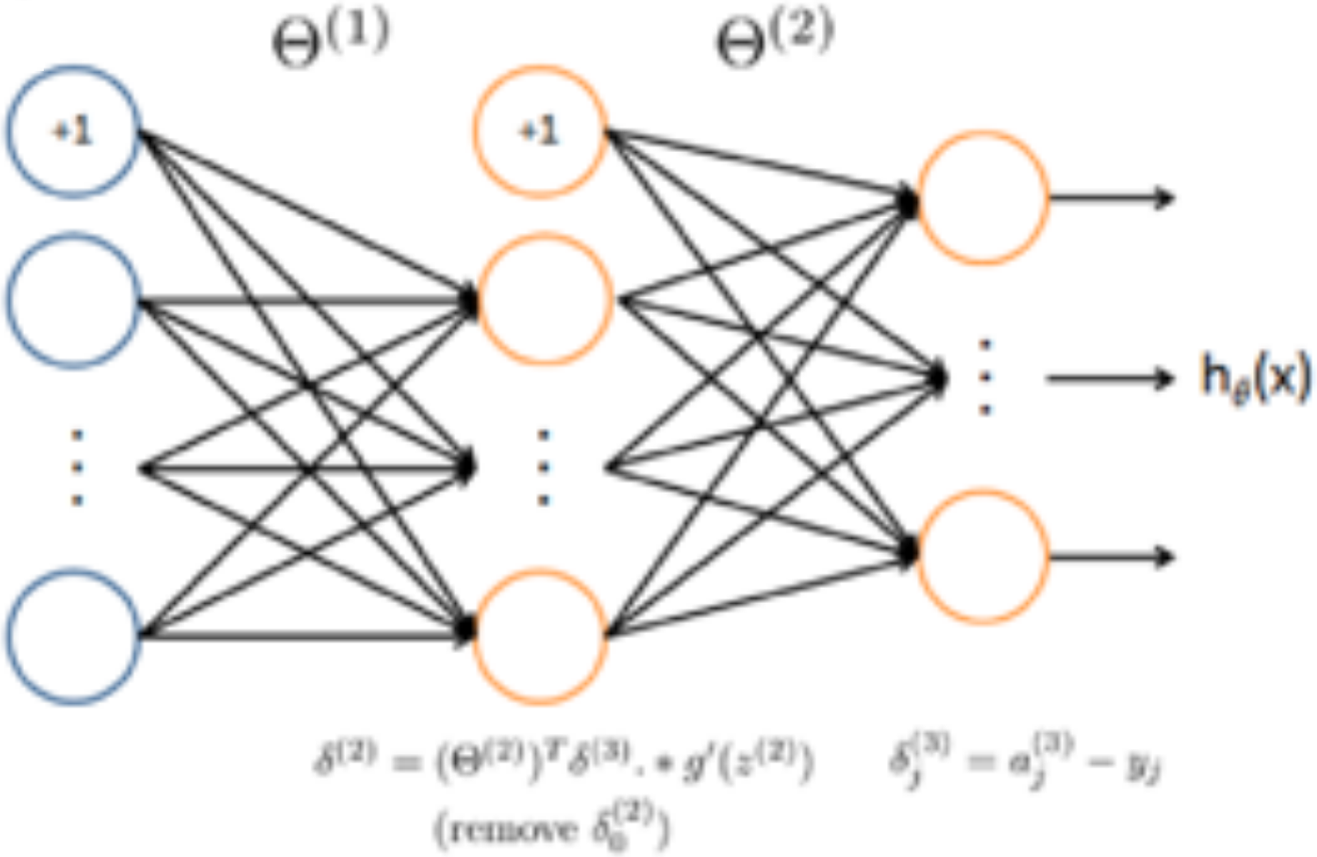
$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)}) = h_\theta(x)$

forward propagation

backward propagation



$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$ (remove $\delta_0^{(2)}$)

$\delta_j^{(3)} = a_j^{(3)} - y_j$

Input Layer Hidden Layer Output Layer

```
input_layer_size = 3
hidden_layer_size = 5
num_labels = 3
m = 5
```

$X : 5 \times 4$

$y : 5 \times 3$

$\Theta^{(1)} : 5 \times 4$

$\Theta^{(2)} : 3 \times 6$

$$X = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} & x_3^{(3)} \\ 1 & x_1^{(4)} & x_2^{(4)} & x_3^{(4)} \\ 1 & x_1^{(5)} & x_2^{(5)} & x_3^{(5)} \end{pmatrix} \quad y = \begin{pmatrix} y_1^{(1)} & y_2^{(1)} & y_3^{(1)} \\ y_1^{(2)} & y_2^{(2)} & y_3^{(2)} \\ y_1^{(3)} & y_2^{(3)} & y_3^{(3)} \\ y_1^{(4)} & y_2^{(4)} & y_3^{(4)} \\ y_1^{(5)} & y_2^{(5)} & y_3^{(5)} \end{pmatrix}$$

$$\Theta^{(1)} = \begin{pmatrix} \Theta_{1,0}^{(1)} & \Theta_{1,1}^{(1)} & \Theta_{1,2}^{(1)} & \Theta_{1,3}^{(1)} \\ \Theta_{2,0}^{(1)} & \Theta_{2,1}^{(1)} & \Theta_{2,2}^{(1)} & \Theta_{2,3}^{(1)} \\ \Theta_{3,0}^{(1)} & \Theta_{3,1}^{(1)} & \Theta_{3,2}^{(1)} & \Theta_{3,3}^{(1)} \\ \Theta_{4,0}^{(1)} & \Theta_{4,1}^{(1)} & \Theta_{4,2}^{(1)} & \Theta_{4,3}^{(1)} \\ \Theta_{5,0}^{(1)} & \Theta_{5,1}^{(1)} & \Theta_{5,2}^{(1)} & \Theta_{5,3}^{(1)} \end{pmatrix} \quad \Theta^{(2)} = \begin{pmatrix} \Theta_{1,0}^{(2)} & \Theta_{1,1}^{(2)} & \Theta_{1,2}^{(2)} & \Theta_{1,3}^{(2)} & \Theta_{1,4}^{(2)} & \Theta_{1,5}^{(2)} \\ \Theta_{2,0}^{(2)} & \Theta_{2,1}^{(2)} & \Theta_{2,2}^{(2)} & \Theta_{2,3}^{(2)} & \Theta_{2,4}^{(2)} & \Theta_{2,5}^{(2)} \\ \Theta_{3,0}^{(2)} & \Theta_{3,1}^{(2)} & \Theta_{3,2}^{(2)} & \Theta_{3,3}^{(2)} & \Theta_{3,4}^{(2)} & \Theta_{3,5}^{(2)} \end{pmatrix}$$

For every training example $(x^{(i)}, y^{(i)})$

```
input_layer_size = 3
hidden_layer_size = 5
num_labels = 3
m = 5
```

$X : 5 \times 4$

$y : 5 \times 3$

$\Theta^{(1)} : 5 \times 4$

$\Theta^{(2)} : 3 \times 6$

forward:

$a^{(1)} = x^{(1)} : (4,1)$

$z^{(2)} = \Theta^{(1)} . a^{(1)} : (5,1)$

$a^{(2)} = g(z^{(2)}) : (5,1)$

add $a_0^{(2)} \rightarrow a^{(2)} : (6,1)$

$z^{(3)} = \Theta^{(2)} a^{(2)} : (3,1)$

$a^{(3)} = g(z^{(3)}) : (3,1)$

backward:

$\delta^{(3)} = a^{(3)} - y^{(1)} : (3,1)$

$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)}) : (6,1)$

$g'(z^{(2)}) = a^{(2)} .* (\overrightarrow{1} - a^{(2)}) : (6,1)$

remove $\delta_0^{(2)} \rightarrow \delta^{(2)} : (5,1)$

$\Delta^{(1)} = \Delta^{(1)} + \delta^{(2)} (a^{(1)})^T : (5,4)$

$\Delta^{(2)} = \Delta^{(2)} + \delta^{(3)} (a^{(2)})^T : (3,6)$

1D and 2D arrays in Python

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

```
a = np.array([1,2,3])

a.shape
(3,)

b = np.array([4,5])

b.shape
(2,)

np.matmul(a[:, np.newaxis],b[np.newaxis, :])
array([[ 4,  5],
       [ 8, 10],
       [12, 15]])
```

Vectorized implementation

$$X = \begin{bmatrix} 1 & — & x^{(1)} & — \\ 1 & — & x^{(2)} & — \\ & \vdots & & \\ 1 & — & x^{(5)} & — \end{bmatrix} \quad \Theta^{(1)} = \begin{bmatrix} — & \Theta_1^{(1)} & — \\ — & \Theta_2^{(1)} & — \\ & \vdots & \\ — & \Theta_5^{(1)} & — \end{bmatrix} \quad (\Theta^{(1)})^T = \begin{bmatrix} | & | & & | \\ \Theta_1^{(1)} & \Theta_2^{(1)} & \dots & \Theta_5^{(1)} \\ | & | & & | \end{bmatrix}$$

$$X(\Theta^{(1)})^T = \begin{bmatrix} x^{(1)}\Theta_1^{(1)} & x^{(1)}\Theta_2^{(1)} & \dots & x^{(1)}\Theta_5^{(1)} \\ x^{(2)}\Theta_1^{(1)} & x^{(2)}\Theta_2^{(1)} & \dots & x^{(2)}\Theta_5^{(1)} \\ & \vdots & & \\ x^{(5)}\Theta_1^{(1)} & x^{(5)}\Theta_2^{(1)} & \dots & x^{(5)}\Theta_5^{(1)} \end{bmatrix}$$

$$g(X(\Theta^{(1)})^T) = \begin{bmatrix} a_1^{(1)} & a_2^{(1)} & \dots & a_5^{(1)} \\ a_1^{(1)} & a_2^{(1)} & \dots & a_5^{(1)} \\ & \vdots & & \\ a_1^{(1)} & a_2^{(1)} & \dots & a_5^{(1)} \end{bmatrix} \quad g(X(\Theta^{(1)})^T) = \begin{bmatrix} — & a^{(1)} & — \\ — & a^{(2)} & — \\ & \vdots & \\ — & a^{(5)} & — \end{bmatrix}$$

Vectorized implementation

```
def forward_propagate(X, theta1, theta2):
    m = X.shape[0]

    a1 = np.hstack([np.ones([m, 1]), X])
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])
    z3 = np.dot(a2, theta2.T)
    h = sigmoid(z3)

    return a1, z2, a2, z3, h
```

Neural networks: learning

Implementation note: Unrolling parameters

Advance optimization

scipy.optimize.minimize¶

scipy.optimize.**minimize**(*fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None*) [source]

Minimization of scalar function of one or more variables.

Parameters: **fun** : *callable*

The objective function to be minimized.

```
fun(x, *args) -> float
```

where x is an 1-D array with shape (n,) and *args* is a tuple of the fixed parameters needed to completely specify the function.

x0 : *ndarray, shape (n,)*

Initial guess. Array of real elements of size (n,), where 'n' is the number of independent variables.

args : *tuple, optional*

Extra arguments passed to the objective function and its derivatives (*fun*, *jac* and *hess* functions).

Advance optimization

jac : *{callable, '2-point', '3-point', 'cs', bool}, optional*

Method for computing the gradient vector. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr. If it is a callable, it should be a function that returns the gradient vector:

```
jac(x, *args) -> array_like, shape (n,)
```

where x is an array with shape (n,) and *args* is a tuple with the fixed parameters. Alternatively, the keywords {'2-point', '3-point', 'cs'} select a finite difference scheme for numerical estimation of the gradient. Options '3-point' and 'cs' are available only to 'trust-constr'. If *jac* is a Boolean and is True, *fun* is assumed to return the gradient along with the objective function. If False, the gradient will be estimated using '2-point' finite difference estimation.

options : *dict, optional*

A dictionary of solver options. All methods accept the following generic options:

maxiter : *int*

Maximum number of iterations to perform.

disp : *bool*

Set to True to print convergence messages.

Unrolling parameters

```
def backprop(theta, input_size, hidden_size,
                    num_labels, X, y, reg):
   ...
   return jVal, gradient


fmin = minimize(fun=backprop, x0=params,
                args=(input_size, hidden_size,
                      num_labels, X, y, reg),
method='TNC', jac=True,
options={'maxiter': 70})
```

"theta" and "gradient" are vectors but in neural networks (L=4):

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

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

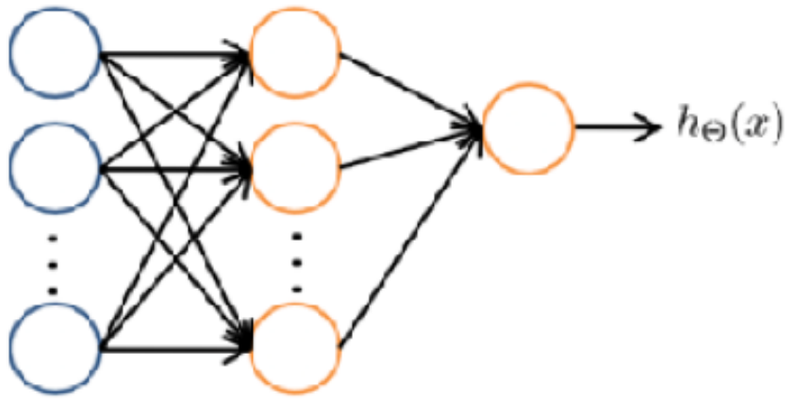
"Unroll" matrices into vectors

Example

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

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

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




```
thetaVec = np.concatenate((np.ravel(Theta1),
                           np.ravel(Theta2),
                           np.ravel(Theta3))


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

Learning algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
Unroll to get **`initialTheta`** to pass to

`minimize(backprop, initialTheta)`

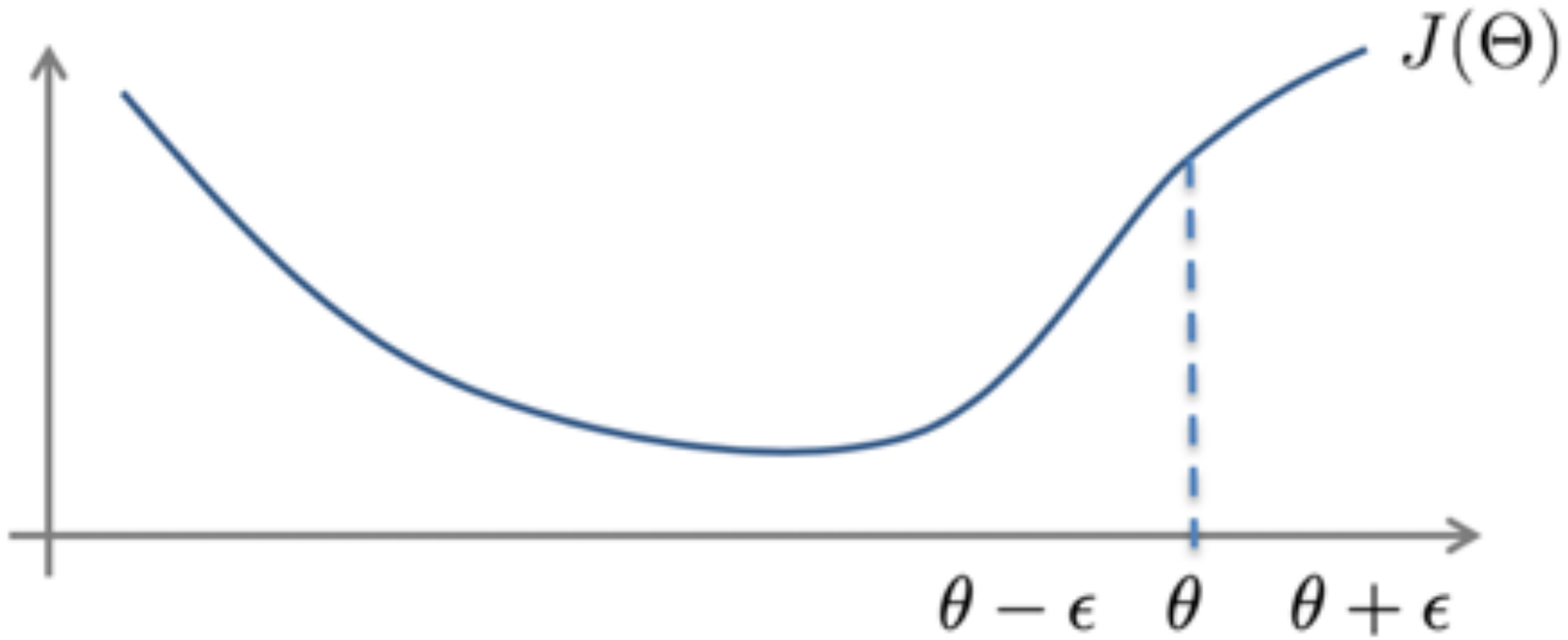
`def = backprop(thetaVec)`

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

Neural networks: learning

Gradient checking

Numerical estimation of gradients



$$\frac{d}{d\Theta} J(\Theta) = \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon} \qquad \epsilon = 10^{-4}$$

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 $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

$\theta = \theta_1, \theta_2, \theta_3, \ldots, \theta_n$

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

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

$$\vdots$$

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

```
for i in range(len(theta)):
  thetaPlus = theta
  thetaPlus[i] = thetaPlus[i] + EPSILON
  thetaMinus = theta
  thetaMinus[i] = thetaMinus[i] – EPSILON
  gradApprox[i] = (J(thetaPlus) – J(thetaMinus))
                  /(2*EPSILON)
```

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

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

Neural networks: learning

Random initialization

Initial value of θ

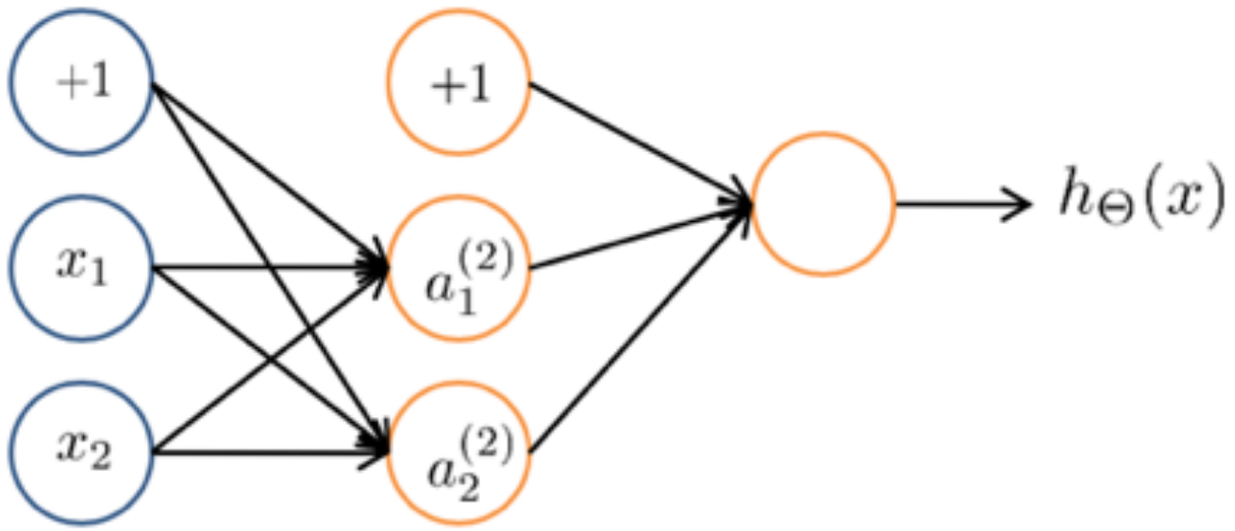
For gradient descent and advanced optimization method, need initial value for θ.

```
optTheta = minimize(backprop, initialTheta)
```

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

Zero initialization



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

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)} \qquad a_1^{(2)} = a_2^{(2)} \qquad \delta_1^{(2)} = \delta_2^{(2)}$$

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

and again: $\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

After each update, parameters corresponding to inputs going into each of two hidden units keep being identical. Every hidden unit is computing the same combination of inputs, i.e., the same “hidden feature”

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.

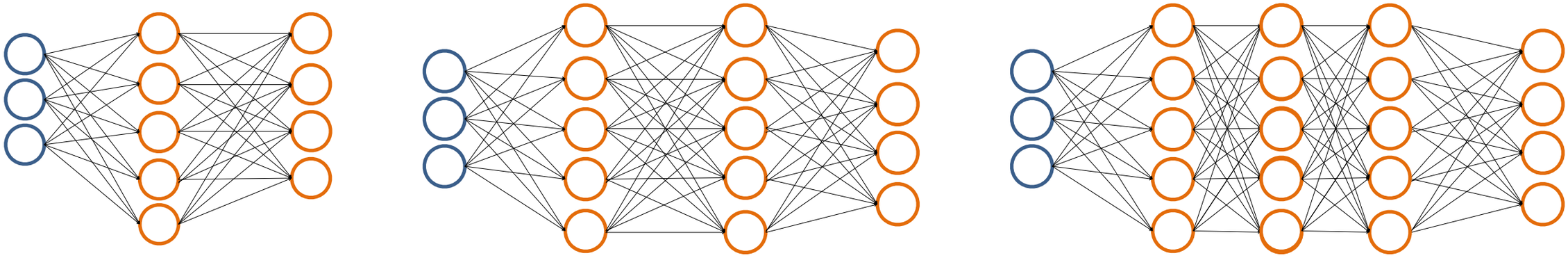
```
Theta1 = np.random.random((10,11))*(2*INIT_EPSILON)
         - INIT_EPSILON

Theta2 = np.random.random((1,11))*(2*INIT_EPSILON)
                  - INIT_EPSILON
```

Neural networks: learning

Putting it together

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



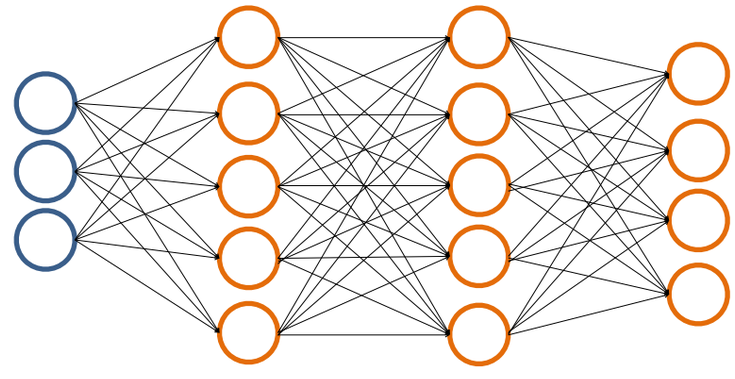
No. of input units: Dimension of features $x^{(i)}$
No. output units: Number of classes
Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

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

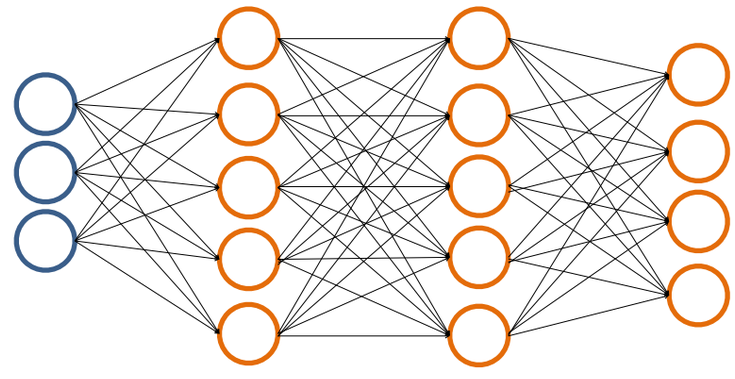
```
for i = 1:m
```

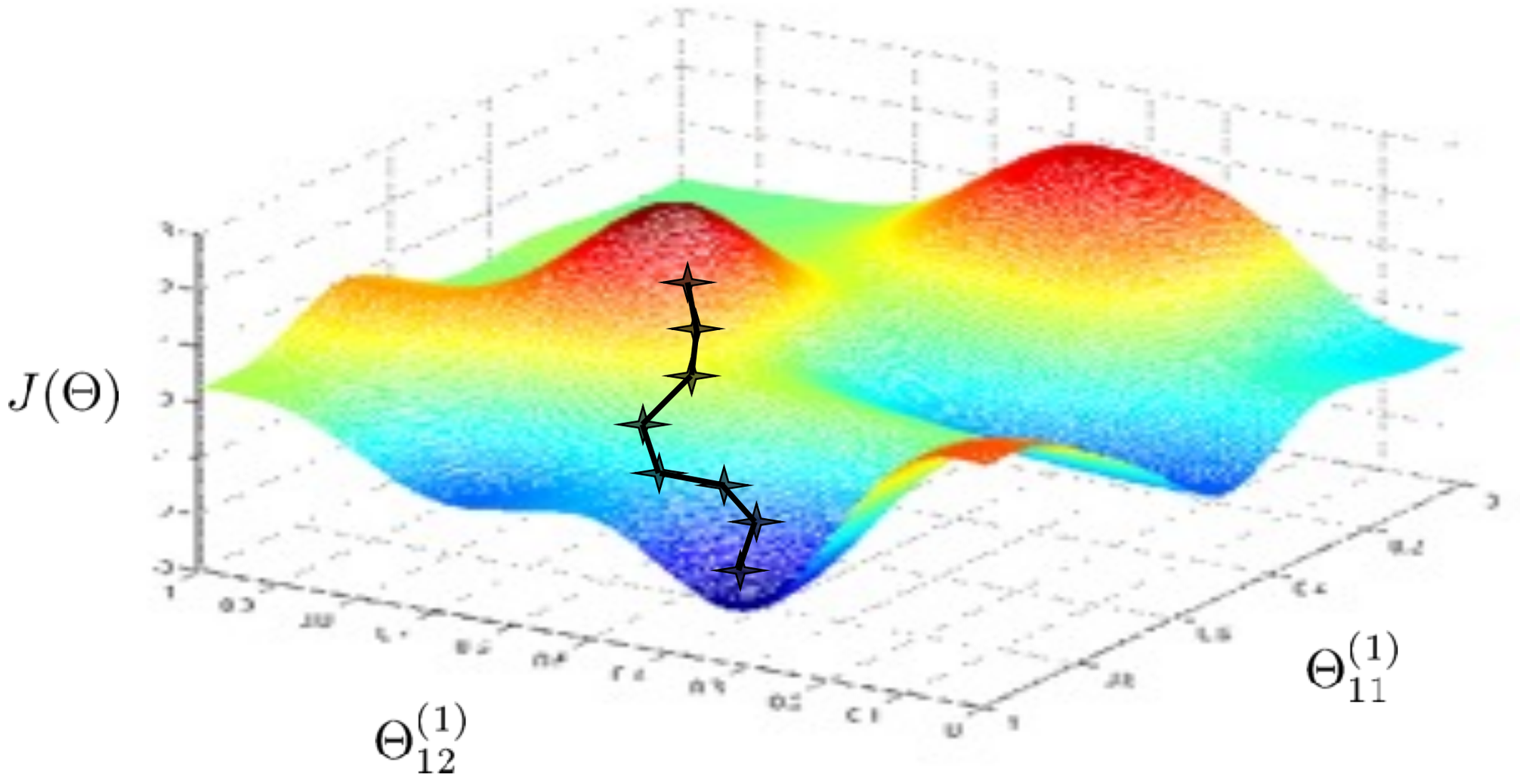
Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$)



Training a neural network

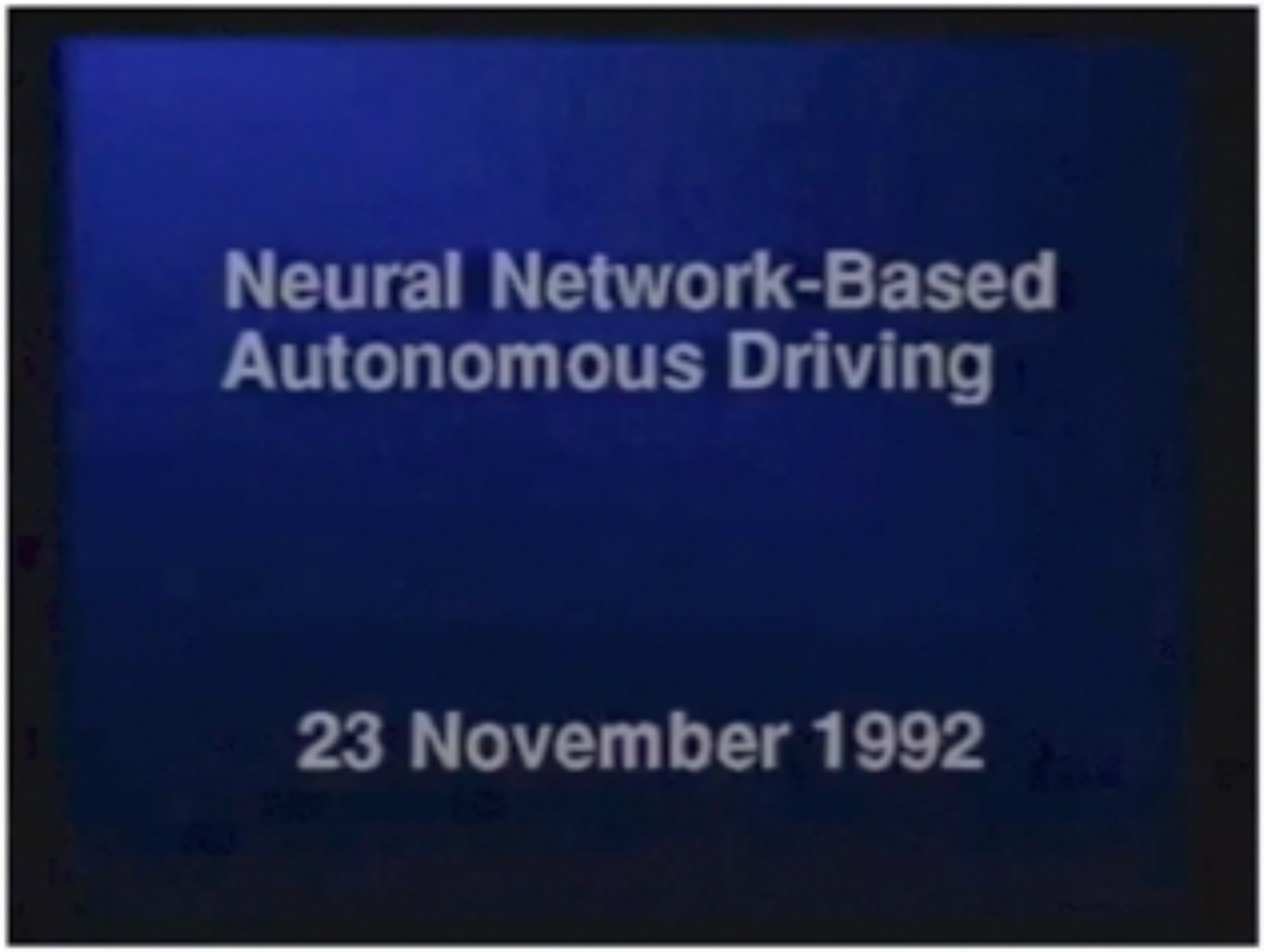
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$
   Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try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




$J(\Theta)$
$\Theta_{12}^{(1)}$
$\Theta_{11}^{(1)}$

Neural networks: learning

Backpropagation example: Autonomous driving



[Courtesy of Dean Pomerleau]