Do not distribute course material

You may not and may not allow others to reproduce or distribute lecture notes and course materials publicly whether or not a fee is charged.

Topic 6
Neural Networks Continued

INTRODUCTION TO MACHINE LEARNING

PROF. LINDA SELLIE

Some of these slides are from Prof. Rangan

Outline

- ❑ Introduction to neurons
- ❑ Nonlinear classifiers from linear features
- ❑ Neural networks notation
- ❑ Pseudocode for prediction
- ❑ Training a neural network
- ❑ Implementing gradient descent for neural networks
  - Vectorization
  - Pseudocode
- ❑ Preprocessing
- ❑ Initialization
- ❑ Activations

The batch gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$\Delta W^{(\ell)} = 0$

$\Delta b^{(\ell)} = 0$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)}$ $\mathbf{z}^{(\ell)}$

run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through n_ℓ

Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level $\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$

$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Random Initialization of Parameters

```
nn_structure = [3, 4, 3]
```

What are the dimensions of $W^{(1)}$?

a) 1x1
b) 3x4
c) 4x3
d) 3x3

Random Initialization of Parameters

```
nn_structure = [3, 4, 3]
```

$W^{(1)}$

```
[[0.65, 0.18, 0.89],
 [0.06, 0.74, 0.72],
 [0.92, 0.29, 0.71],
 [0.39, 0.41, 0.10]]
```

$b^{(1)}$

```
0.07,
0.35,
0.60,
0.50
```

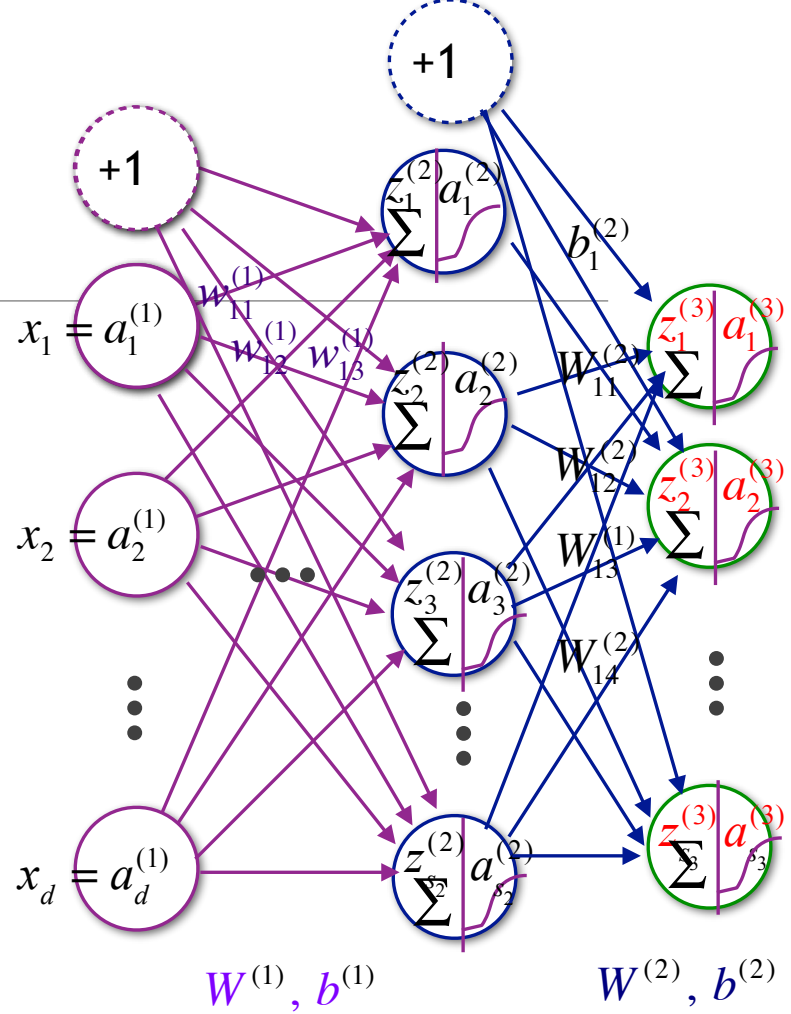
$W^{(2)}$

```
[[0.45, 0.64, 0.10, 0.82],
 [0.30, 0.25, 0.39, 0.36],
 [0.78, 0.60, 0.16, 0.07]]
```

$b^{(2)}$

```
0.14,
0.64,
0.62
```

```
def setup_and_init_weights(nn_structure):
    W = {}
    b = {}
    for l in range(1, len(nn_structure)):
        W[l] = r.random_sample((nn_structure[l], nn_structure[l-1]))
        b[l] = r.random_sample((nn_structure[l],))
    return W, b
```



Array - we represent it as column
Numpy represents it as a row

The gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$$\Delta W^{(\ell)} = 0$$
$$\Delta b^{(\ell)} = 0$$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)}$ $\mathbf{z}^{(\ell)}$
run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through n_ℓ
Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level $\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)}\left(a^{(\ell)}\right)^T$

$$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Initialization of △W and △b to zero

nn_structure = [3, 4, 3]

```
[[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]]
```

$\triangle W^{(1)}$

```
[0.,
 0.,
 0.,
 0.]
```

$\triangle b^{(1)}$

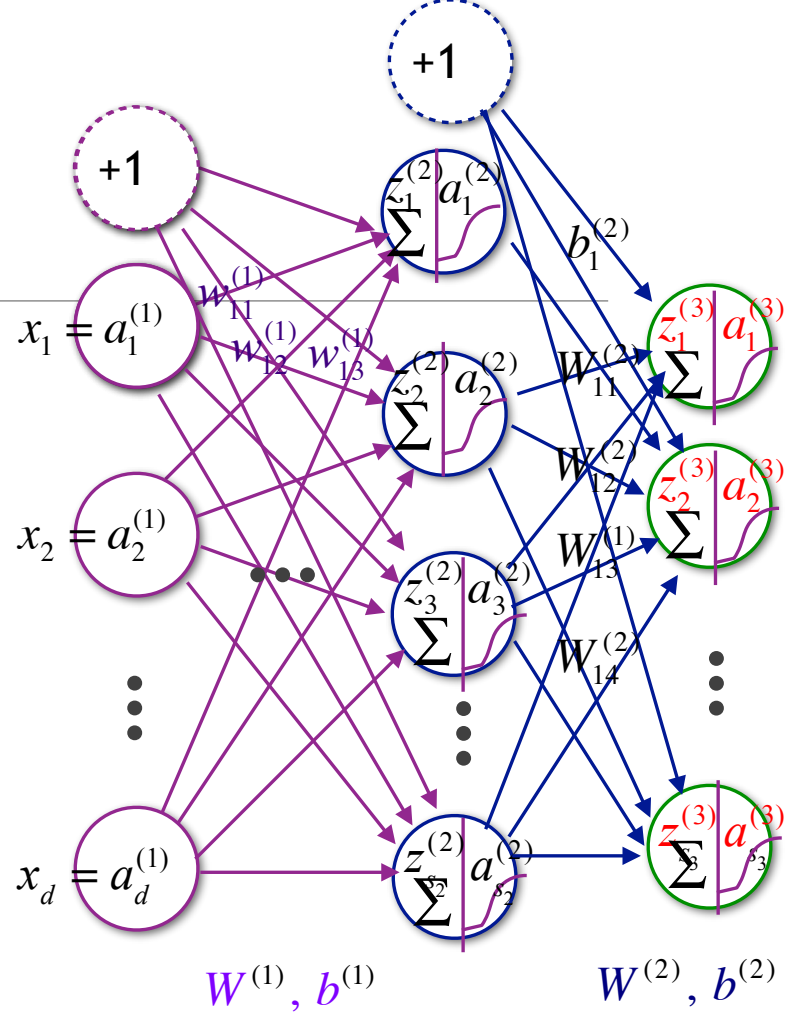
```
[[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]]
```

$\triangle W^{(2)}$

```
[0.,
 0.,
 0.]
```

$\triangle b^{(2)}$

```
def init_tri_values(nn_structure):
    tri_W = {}
    tri_b = {}
    for l in range(1, len(nn_structure)):
        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-1]))
        tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```



The gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$\Delta W^{(\ell)} = 0$

$\Delta b^{(\ell)} = 0$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)}$ $\mathbf{z}^{(\ell)}$

run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through n_ℓ

Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level

$$\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Forward Propagation

nn_structure = [3, 4, 3]

- pseudocode:

$a^{(1)} = x$

for ℓ = 1 to L do

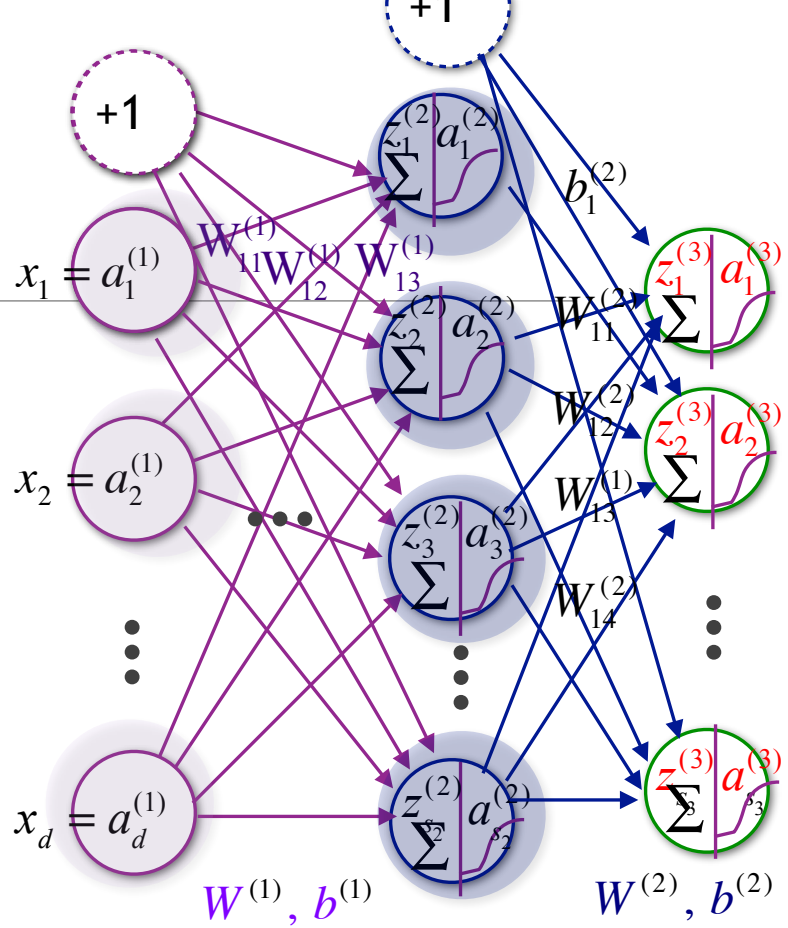
$z^{(\ell+1)} = W^{(\ell)} a^{(\ell)} + b^{(\ell)}$

$a^{(\ell+1)} = f(z^{(\ell+1)})$

$\hat{y} = a^{(n_L)}$

What is the dimension of $a^{(2)}$?

a) Vector of size 1
b) Vector of size 2
c) Vector of size 3
d) Vector of size 4
e) Vector of size 5
f) Vector of size 6



Forward Propagation

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)} \quad x = [1,\ 1,\ 1]^T$$

$$a^{(l+1)} = f(z^{(l+1)}) = f(W^{(l)} a^{(l)} + b^{(l)})$$

- pseudocode:

$a^{(1)} = x$

for ℓ = 1 to L do

$z^{(\ell+1)} = W^{(\ell)} a^{(\ell)} + b^{(\ell)}$

$a^{(\ell+1)} = f(z^{(\ell+1)})$

$\hat{y} = a^{(n_L)}$

$W^{(1)}\ a^{(1)} + b^{(1)} = z^{(2)}$

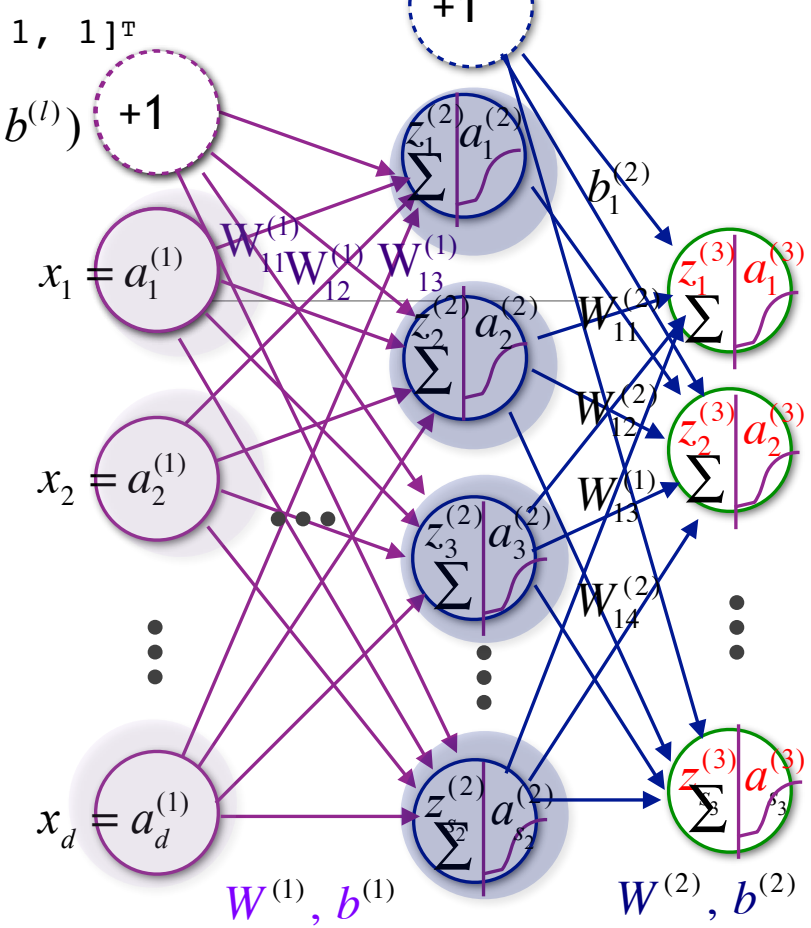
```
[[0.65, 0.18, 0.89],  [1      [0.07,    [1.80,
 [0.06, 0.74, 0.72],   1   +    0.35,  =   1.88,
 [0.92, 0.29, 0.71],   1]       0.60,      2.53,
 [0.39, 0.41, 0.10]]            0.50]      1.39]
```

$f(z^{(2)}) = a^{(2)}$

```
f( [1.80,    )  =  [0.86,
     1.88,          0.87,
     2.53,          0.93,
     1.39]          0.80]
```

```
def f(x):
    return 1 / (1 + np.exp(-x))
```

```
def feed_forward(x, W, b):
    a = {1: x}
    z = { }
    for l in range(1, len(W) + 1): # for each layer
        node_in = a[l]
        z[l+1] = W[l].dot(node_in) + b[l]  #  z^(l+1) = W^(l) a^(l) + b^(l)
        a[l+1] = f(z[l+1])  #  a^(l+1) = f(z^(l+1)) = f(W^(l) a^(l) + b^(l))
    return a, z
```



Forward Propagation

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$

$$a^{(l+1)} = f(z^{(l+1)}) = f(W^{(l)} a^{(l)} + b^{(l)})$$

- pseudocode:

$a^{(1)} = x$

for ℓ = 1 to L do

$z^{(\ell+1)} = W^{(\ell)} a^{(\ell)} + b^{(\ell)}$

$a^{(\ell+1)} = f(z^{(\ell+1)})$

$\hat{y} = a^{(n_L)}$

```
W(2)
 [[0.45, 0.64, 0.10, 0.82],
  [0.30, 0.25, 0.39, 0.36],
  [0.78, 0.60, 0.16, 0.07]]
```

$W^{(2)} a^{(2)} + b^{(2)} = z^{(3)}$

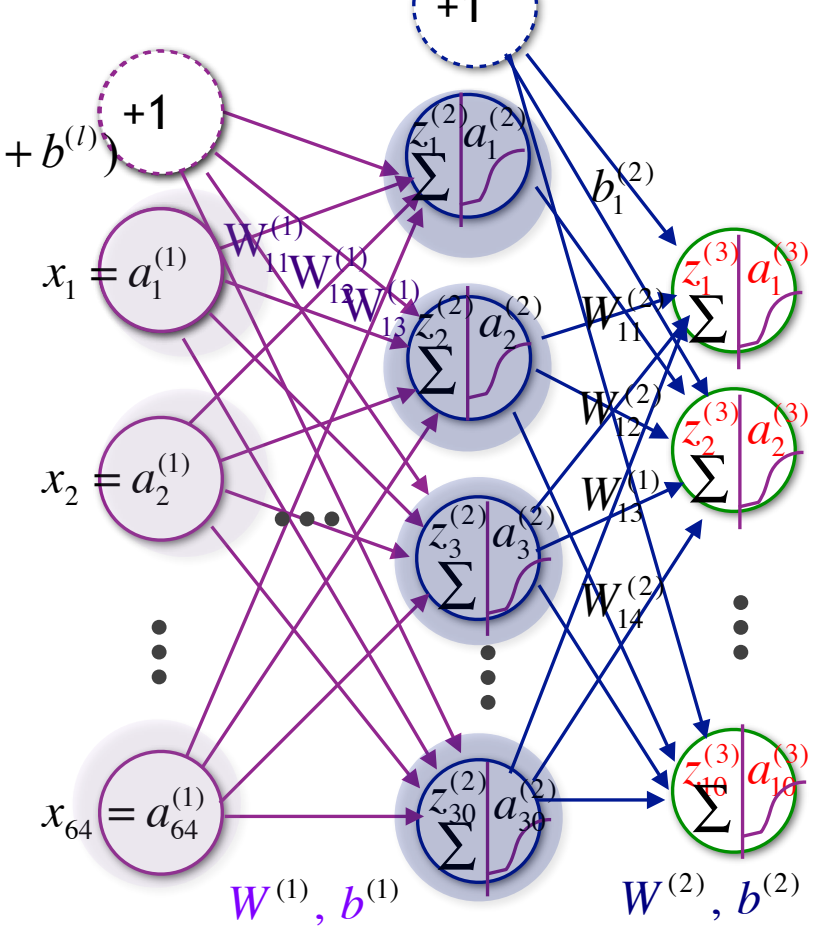
```
[0.86,   0.14,     [2.66,
 0.87,   0.64,  =   1.76,
 0.93,   0.62       2.01]
 0.80]
```

$f(z^{(3)}) = a^{(3)}$

```
f([2.66,  ) = [0.93
   1.76,       0.85
   2.01]       0.88 ]
```

```
def f(x):
    return 1 / (1 + np.exp(-x))
```

```
def feed_forward(x, W, b):
    a = {1: x}
    z = { }
    for l in range(1, len(W) + 1): # for each layer
        node_in = a[l]
        z[l+1] = W[l].dot(node_in) + b[l]  #  z^(l+1) = W^(l) a^(l) + b^(l)
        a[l+1] = f(z[l+1])  #  a^(l+1) = f(z^(l+1)) = f(W^(l) a^(l) + b^(l))
    return a, z
```



The gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$$\Delta W^{(\ell)} = 0$$

$$\Delta b^{(\ell)} = 0$$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)} \mathbf{z}^{(\ell)}$

run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through nℓ

Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level $\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$

$$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Definitions:

$$\frac{\partial J}{\partial W^{(\ell)}} = \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\frac{\partial J}{\partial b^{(\ell)}} = \delta^{(\ell+1)}$$

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

$$\frac{\partial J}{\partial W_{ij}^{(\ell)}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$$

$$\frac{\partial J}{\partial b_i^{(\ell)}} = \delta_i^{(\ell+1)}$$

Calculating δ

$$\delta^{(n_\ell)} = -(y - a^{(n_\ell)}) \bullet f'(z^{(n_\ell)})$$

$$\delta^{(\ell)} = \left(\left(W^{(\ell)} \right)^T \delta^{(\ell+1)} \right) \bullet f'(z^{(\ell)})$$

```
def f_deriv(x):
    return f(x) * (1 - f(x))
```

```
def calculate_out_layer_delta(y, a_out, z_out):
   return -(y-a_out) * f_deriv(z_out)
```

$$-\left(y - a^{(3)}\right) \bullet f'(z^{(3)})$$

$$-\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0.935 \\ 0.854 \\ 0.882 \end{pmatrix} \bullet f'\begin{pmatrix} 2.66, \\ 1.76, \\ 2.01 \end{pmatrix} = -\begin{pmatrix} 0.065 \\ 0.146 \\ 0.118 \end{pmatrix} \bullet \begin{pmatrix} 0.06087702 \\ 0.12500965 \\ 0.10390214 \end{pmatrix} = \delta^{(3)}$$

$$\begin{matrix} -0.00396415 \\ -0.01830894 \\ -0.01223682 \end{matrix}$$

Calculating δ

$$\delta^{(n_\ell)} = -(y - a^{(n_\ell)}) \bullet f'(z^{(n_\ell)})$$

$$\delta^{(\ell)} = \left(\left(W^{(\ell)} \right)^T \delta^{(\ell+1)} \right) \bullet f'(z^{(\ell)})$$

```
def f_deriv(x):
    return f(x) * (1 - f(x))
```

```
def calculate_hidden_delta(delta_plus_1, w_l, z_l):
   return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

$$\left(W^{(2)T} \quad \delta^{(3)} \right) \bullet \quad f'(z^{(2)}) \quad = \left(W^{(2)T} \delta^{(3)} \right) \quad f'(z^{(2)}) \quad = \quad \delta^{(2)}$$

$$\left(\begin{pmatrix} [0.453 & 0.304 & 0.776] \\ [0.637 & 0.246 & 0.600] \\ [0.996 & 0.390 & 0.164] \\ [0.820 & 0.361 & 0.070] \end{pmatrix} \begin{pmatrix} -0.004 \\ -0.018 \\ -0.012 \end{pmatrix} \right) \bullet f' \begin{pmatrix} 1.798 \\ 1.875 \\ 2.530 \\ 1.389 \end{pmatrix} = \begin{pmatrix} -0.017 \\ -0.014 \\ -0.013 \\ -0.011 \end{pmatrix} \bullet \begin{pmatrix} 0.122 \\ 0.115 \\ 0.068 \\ 0.160 \end{pmatrix} = \begin{pmatrix} -0.002 \\ -0.002 \\ -0.001 \\ -0.002 \end{pmatrix}$$

The gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$\Delta W^{(\ell)} = 0$

$\Delta b^{(\ell)} = 0$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)}$ $\mathbf{z}^{(\ell)}$

run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through n_ℓ

Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level

$$\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Definitions:

$$\frac{\partial J}{\partial W^{(\ell)}} = \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\frac{\partial J}{\partial b^{(\ell)}} = \delta^{(\ell+1)}$$

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

$$\frac{\partial J}{\partial W_{ij}^{(\ell)}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$$

$$\frac{\partial J}{\partial b_i^{(\ell)}} = \delta_i^{(\ell+1)}$$

Computing

$$\delta^{(n_\ell)} = -(y - a^{(n_\ell)}) \bullet f'(z^{(n_\ell)}) \qquad \Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\delta^{(\ell)} = \left(\left(W^{(\ell)}\right)^T \delta^{(\ell+1)}\right) \bullet f'(z^{(\ell)}) \qquad \Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

```
def f_deriv(x):
    return f(x) * (1 - f(x))
```

```
def calculate_out_layer_delta(y, a_out, z_out):
    return -(y-a_out) * f_deriv(z_out)
```

$= -(y - a^{n_\ell}) \bullet f'(z^{n_\ell})$

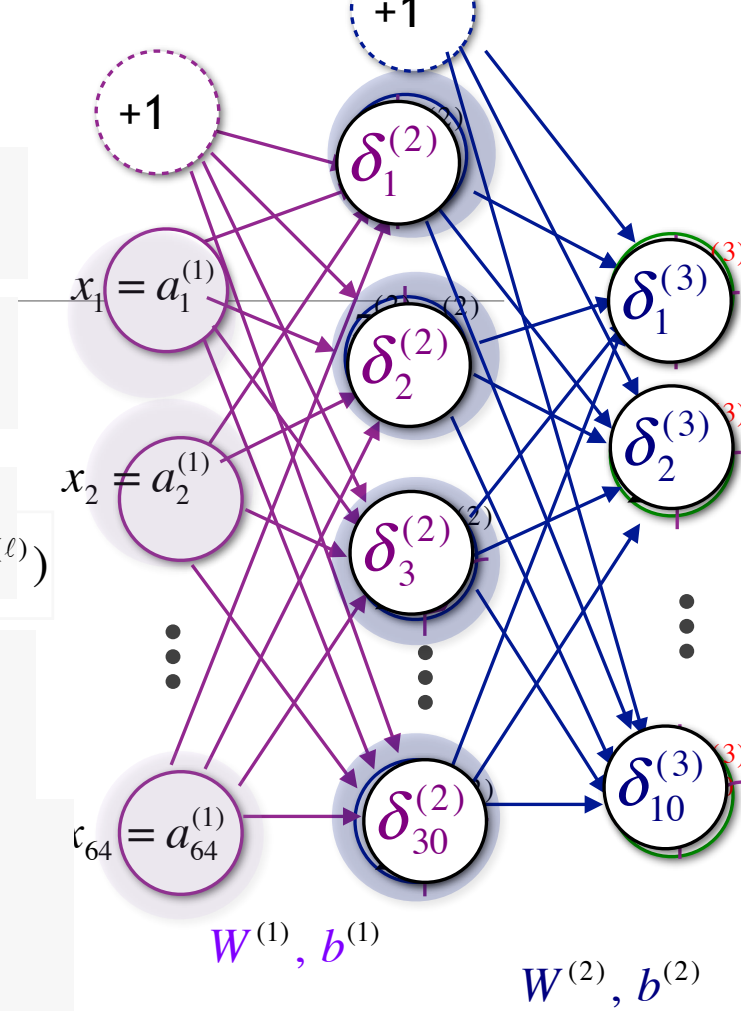
```
def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

$= \left(\left(W^{(\ell)}\right)^T \delta^{(\ell+1)}\right) \bullet f'(z^{(\ell)})$

```
for i in range(len(y)):
    delta = {}
    a, z = feed_forward(X[i, :], W, b)
    for l in range(len(nn_structure), 0, -1):
        if l == len(nn_structure):
            delta[l] = calculate_out_layer_delta(y[i,:], a[l], z[l])
        else:
            if l > 1:
                delta[l] = calculate_hidden_delta(delta[l+1], W[l], z[l])
    tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.transpose(a[l][:,np.newaxis]))
    tri_b[l] += delta[l+1]
```

$\# \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$

$\# \Delta b^{(\ell)} + \delta^{(\ell+1)}$



The gradient descent algorithm

Algorithm and code adapted from http://adventuresinmachinelearning.com/neural-networks-tutorial/

Randomly initialize the weights for each layer: $W^{(\ell)}$

While iterations < iteration limit:

$\Delta W^{(\ell)} = 0$

$\Delta b^{(\ell)} = 0$

For i = 1 to N:

run forward propagation and save for each level, the values $a^{(\ell)}$ $\mathbf{z}^{(\ell)}$

run back-propagation to calculate $\delta^{(\ell)}$ values for levels 2 through n_ℓ

Update $\Delta W^{(\ell)}$ $\Delta b^{(\ell)}$ for each level

$$\Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

Perform a gradient descent step

$$W^{(\ell)} = W^{(\ell)} - \alpha \frac{1}{N} \Delta W^{(\ell)}$$

$$b^{(\ell)} = b^{(\ell)} - \alpha \frac{1}{N} \Delta b^{(\ell)}$$

Definitions:

$$\frac{\partial J}{\partial W^{(\ell)}} = \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\frac{\partial J}{\partial b^{(\ell)}} = \delta^{(\ell+1)}$$

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

$$\frac{\partial J}{\partial W_{ij}^{(\ell)}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$$

$$\frac{\partial J}{\partial b_i^{(\ell)}} = \delta_i^{(\ell+1)}$$

Computing

$$\delta^{(n_\ell)} = -(y - a^{(n_\ell)}) \bullet f'(z^{(n_\ell)}) \qquad \Delta W^{(\ell)} = \Delta W^{(\ell)} + \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

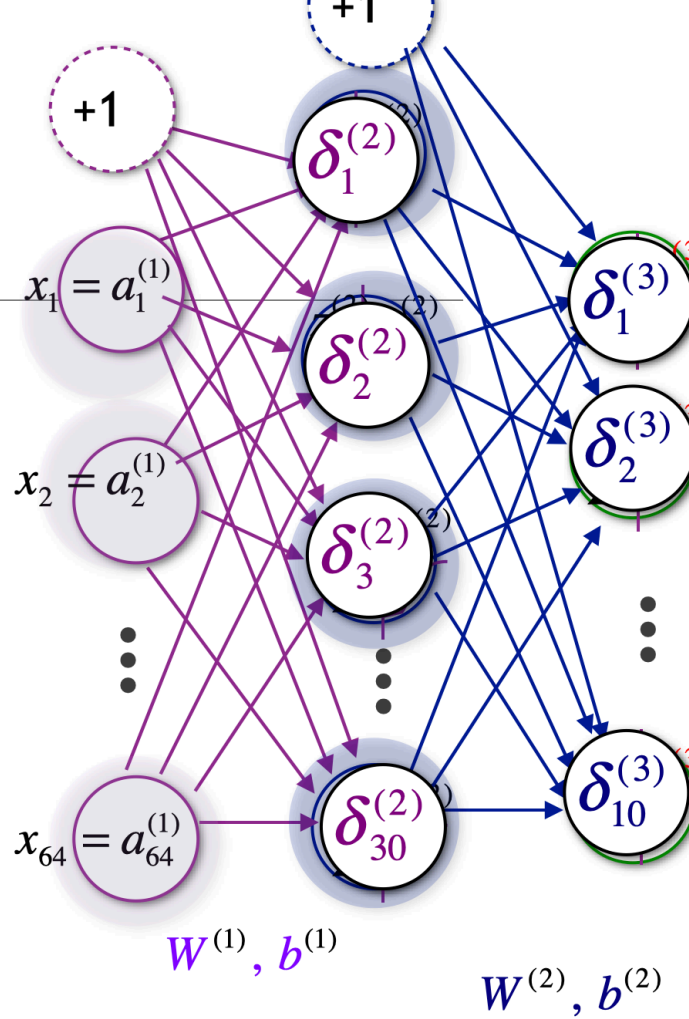
$$\delta^{(\ell)} = \left(\left(W^{(\ell)}\right)^T \delta^{(\ell+1)}\right) \bullet f'(z^{(\ell)}) \qquad \Delta b^{(\ell)} = \Delta b^{(\ell)} + \delta^{(\ell+1)}$$

```
while cnt < iter_num:
    tri_W, tri_b = init_tri_values(nn_structure)

    for i in range(len(y)):
        delta = {}
        a, z = feed_forward(X[i, :], W, b)
        for l in range(len(nn_structure), 0, -1):
            if l == len(nn_structure):
                delta[l] = calculate_out_layer_delta(y[i,:], a[l], z[l])
            else:
                if l > 1:
                    delta[l] = calculate_hidden_delta(delta[l+1], W[l], z[l])
            tri_W[l] += np.dot(delta[l+1][:,np.newaxis], np.transpose(a[l][:,np.newaxis]))
            tri_b[l] += delta[l+1] # Δb^(ℓ) + δ^(ℓ+1)

    # perform the gradient descent step for the weights in each layer
    for l in range(len(nn_structure) - 1, 0, -1):
        W[l] += -alpha * (1.0/N * tri_W[l])
        b[l] += -alpha * (1.0/N * tri_b[l])

    cnt += 1
```

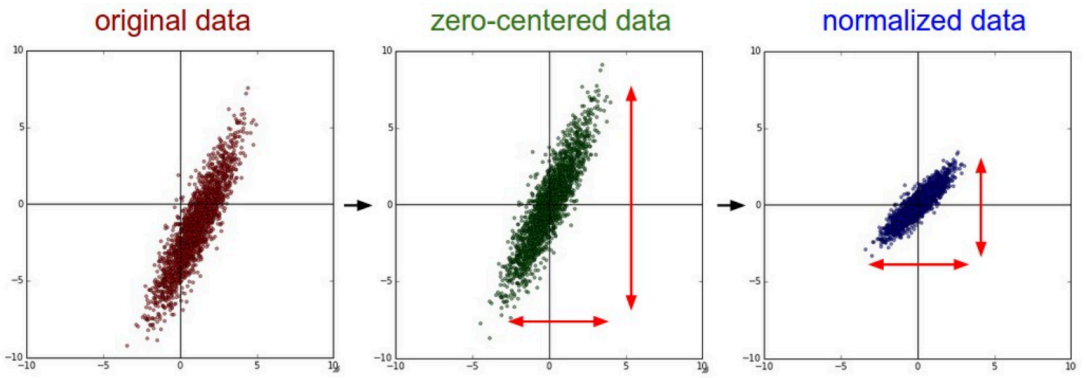


Outline

- ❑ Introduction to neurons
- ❑ Nonlinear classifiers from linear features
- ❑ Neural networks notation
- ❑ Pseudocode for prediction
- ❑ Training a neural network
- ❑ Implementing gradient descent for neural networks
  - Vectorization
- ❑ Preprocessing
- ❑ Initialization
- ❑ Activations

Preprocessing

To speed up learning



https://cs231n.github.io/neural-networks-2/

- **Mean subtraction**: For ever feature subtract the mean, thus zero centering the data

```
X= [[ 2  2  3]
    [-1  4  6]
    [ 2  0  9]]
```

  - X -= np.mean(X, axis = 0)

```
[[ 1.  0. -3.]
 [-2.  2.  0.]
 [ 1. -2.  3.]]
```

  - For image data, we can subtract a the average pixel intensity
  - X -= np.mean(X)

```
[[-1. -1.  0.]
 [-4.  1.  3.]
 [-1. -3.  6.]]
```

- **Normalization**: make all the features are roughly the same scale
  - X/=np.std(X, axis=0)

```
[[ 0.7  0.  -1.2]
 [-1.4  1.2  0. ]
 [ 0.7 -1.2  1.2]]
```

  - min/max scaling Or make the min and max for each feature -1 and 1 (only do this if each feature should be equally important)

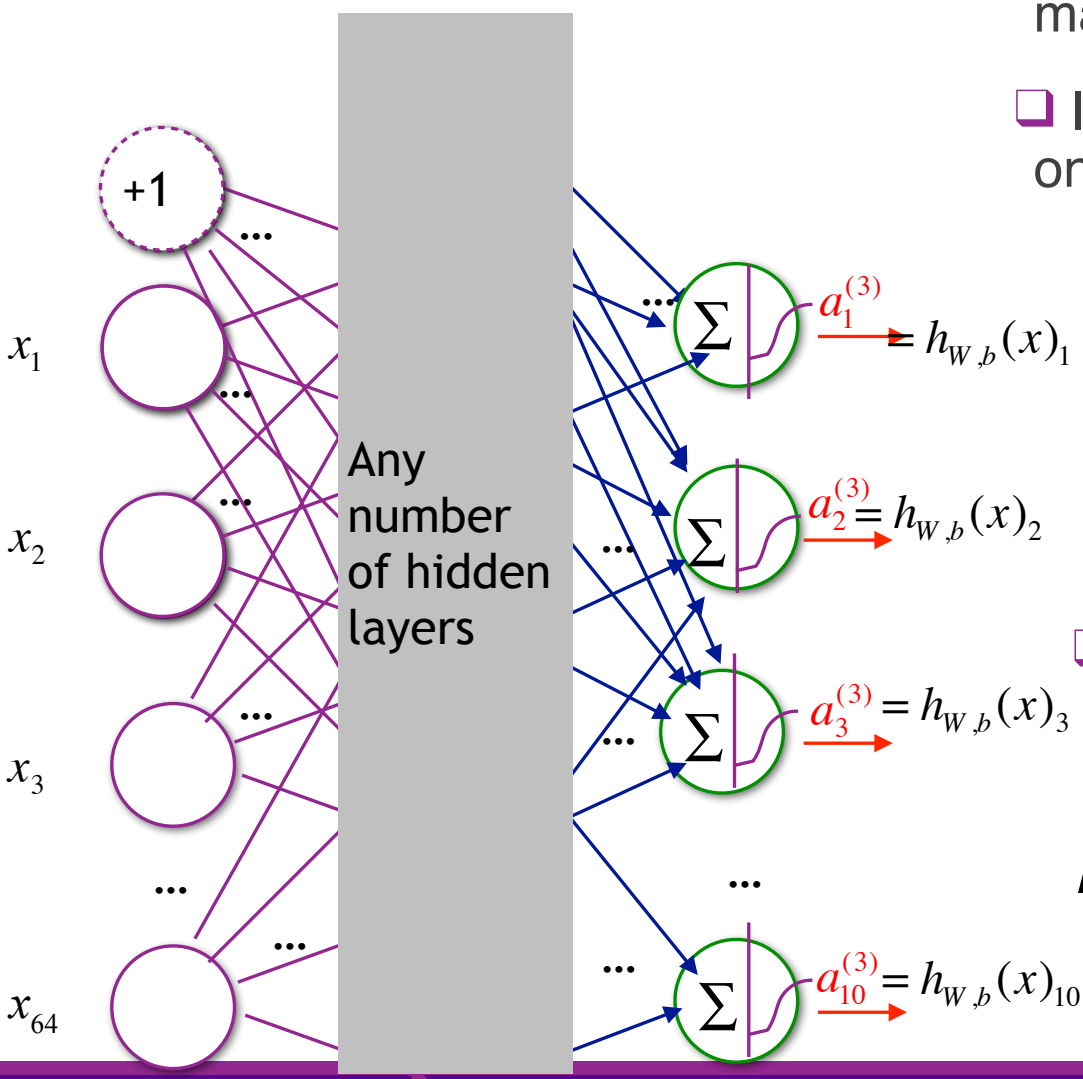
```
X= [[ 2  2  3]
    [-1  4  6]
    [ 2  0  9]]
```

```
[[ 1.  0. -3.]
 [-2.  2.  0.]
 [ 1. -2.  3.]]
```

Often only zero center for image data (and don't normalize)

https://cs231n.github.io/neural-networks-2/

Multiple Output Units: One-vs-All



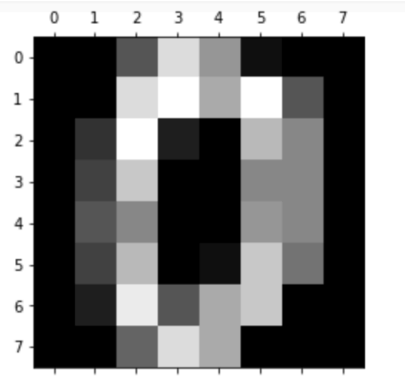
- Instead of just computing one output, we can compute as many as we would like
- If we were classifying digits, we could have ten outputs, one for each digit - using a 1 of k representation

$$y = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ If } y = 0 \qquad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \text{ If } y = 1 \quad \cdots \qquad y = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \text{ If } y = 9$$

- For our hypothesis $h_{W,b}(x) \in \mathbb{R}^K$ we want:

$$h_{W,b}(x) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad h_{W,b}(x) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \cdots \quad h_{W,b}(x) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Pre-processing steps



- One-vs-all

```
def convert_y_to_vect(y):
    y_vect = np.zeros((len(y), 10))
    for i in range(len(y)):
        y_vect[i, y[i]] = 1
    return y_vect
```

```
[9
 9
 4
 3
 …]

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
                                …]]
```

For each feature. Find the mean and standard deviation of the feature and then subtract the mean and divide by the standard deviation.
x'=(x – mean) / standard_deviation

Mean of feature 1 is 0
Mean of feature 2 is 3

- Scale the date set

```
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
```

```
0.   0.   5. 13.   9.   1.   0.   0.
0.   0. 13. 15. 10. 15.   5.   0.
0.   3. 15.   2.   0. 11.   8.   0.
0.   4. 12.   0.   0.   8.   8.   0.
0.   5.   8.   0.   0.   9.   8.   0.
0.   4. 11.   0.   1. 12.   7.   0.
0.   2. 14.   5. 10. 12.   0.   0.
0.   0.   6. 13. 10.   0.   0.   0.

 0.    -0.335 -0.043  0.274 -0.664 -0.844 -0.41  -0.125
-0.059 -0.624  0.483  0.76  -0.058  1.128  0.88  -0.13
-0.045  0.111  0.896 -0.861 -1.15   0.515  1.906 -0.114
-0.033  0.486  0.47  -1.5   -1.614  0.076  1.542 -0.047
 0.     0.765  0.053 -1.448 -1.737  0.044  1.44   0.
-0.061  0.811  0.63  -1.122 -1.066  0.661  0.818 -0.089
-0.035  0.742  1.151 -0.869  0.11   0.538 -0.757 -0.21
-0.024 -0.299  0.087  0.208 -0.367 -1.147 -0.506 -0.196
```

Outline

- ❑ Introduction to neurons
- ❑ Nonlinear classifiers from linear features
- ❑ Neural networks notation
- ❑ Pseudocode for prediction
- ❑ Training a neural network
- ❑ Implementing gradient descent for neural networks
  - Vectorization
- ❑ Preprocessing
- ❑ Initialization
- ❑ Activations

https://datascience.stackexchange.com/questions/13061/when-to-use-he-or-glorot-normal-initialization-over-uniform-init-and-what-are

Initialization

in back propagation we propagate the error gradient from the output layer to the input layer.

Vanishing gradient problem: the gradients can get smaller as they propagate towards the front of the network - and thus they don't get change

Exploding gradients problem: the gradients can get larger - too large (mostly a problem in recurrent neural networks)

Another problem is layers learn at different speeds

Scale and center the data first

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

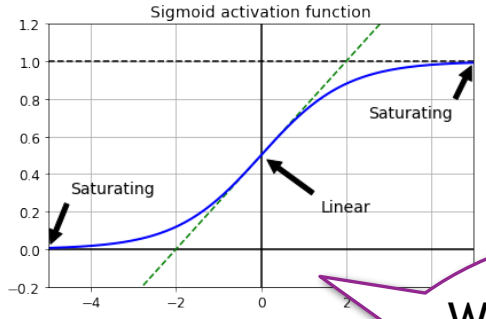
https://www.deeplearning.ai/ai-notes/initialization/

Outline

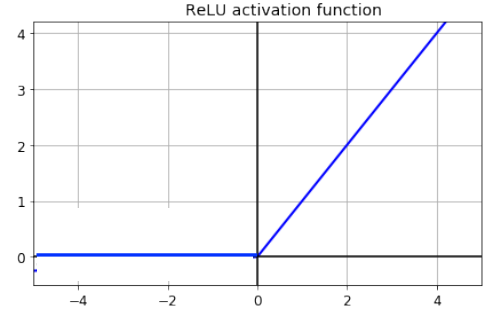
- ❑ Introduction to neurons
- ❑ Nonlinear classifiers from linear features
- ❑ Neural networks notation
- ❑ Pseudocode for prediction
- ❑ Training a neural network
- ❑ Implementing gradient descent for neural networks
  - Vectorization
- ❑ Preprocessing
- ❑ Initialization
- ❑ Activations

Activations

deep neural nets can have saturation problems with sigmoid and tan



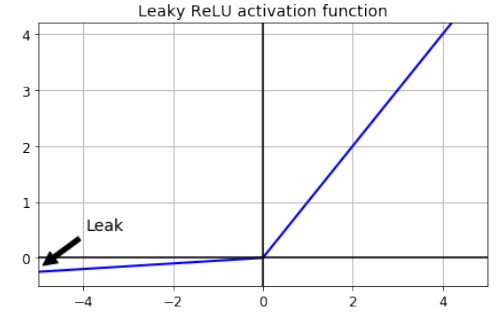
max(0,z)



What happens if z=10, or -10? What is the

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial W_{ij}}$$

max(αz,z)

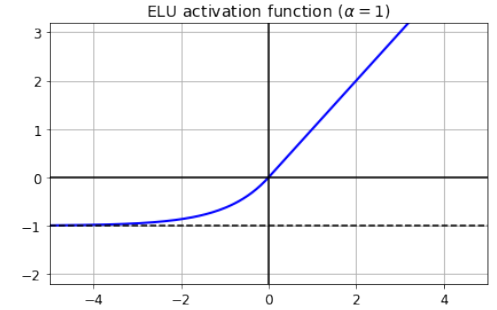


ReLU - doesn't saturate in +region, suffers from dying ReLU problem

Leaky ReLU - typically hyper parameter $\alpha = 0.01$. This prevents leaky ReLU from dying. (Some results show $\alpha = 0.2$ is better)

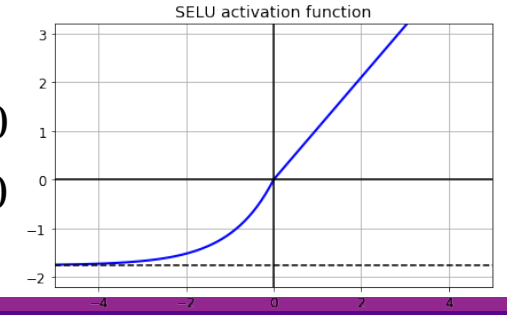
ELU (exponential linear unit) - typically hyper parameter $\alpha = 1$. Takes on negative values to prevent vanishing gradient, non zero gradient so doesn't have dying unit problem, smooth everywhere. However it is slower than ReLU and leaky ReLU due to the exponential function but often has faster convergence

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$



SELU (scaled exponential unit)

$$\text{SELU}_\alpha(z) = \lambda \begin{cases} \alpha(\exp(z) - 1) & \text{if } z > 0 \\ z & \text{if } z \leq 0 \end{cases}$$

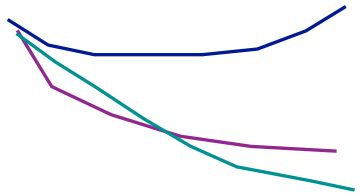


Hyperparameter testing

In batch gradient descent cost function should go down

Babysit models (if you don't have the ability to try too many parameters)

Sample at random the many different hyper-parameters:



uniformly at random examples:

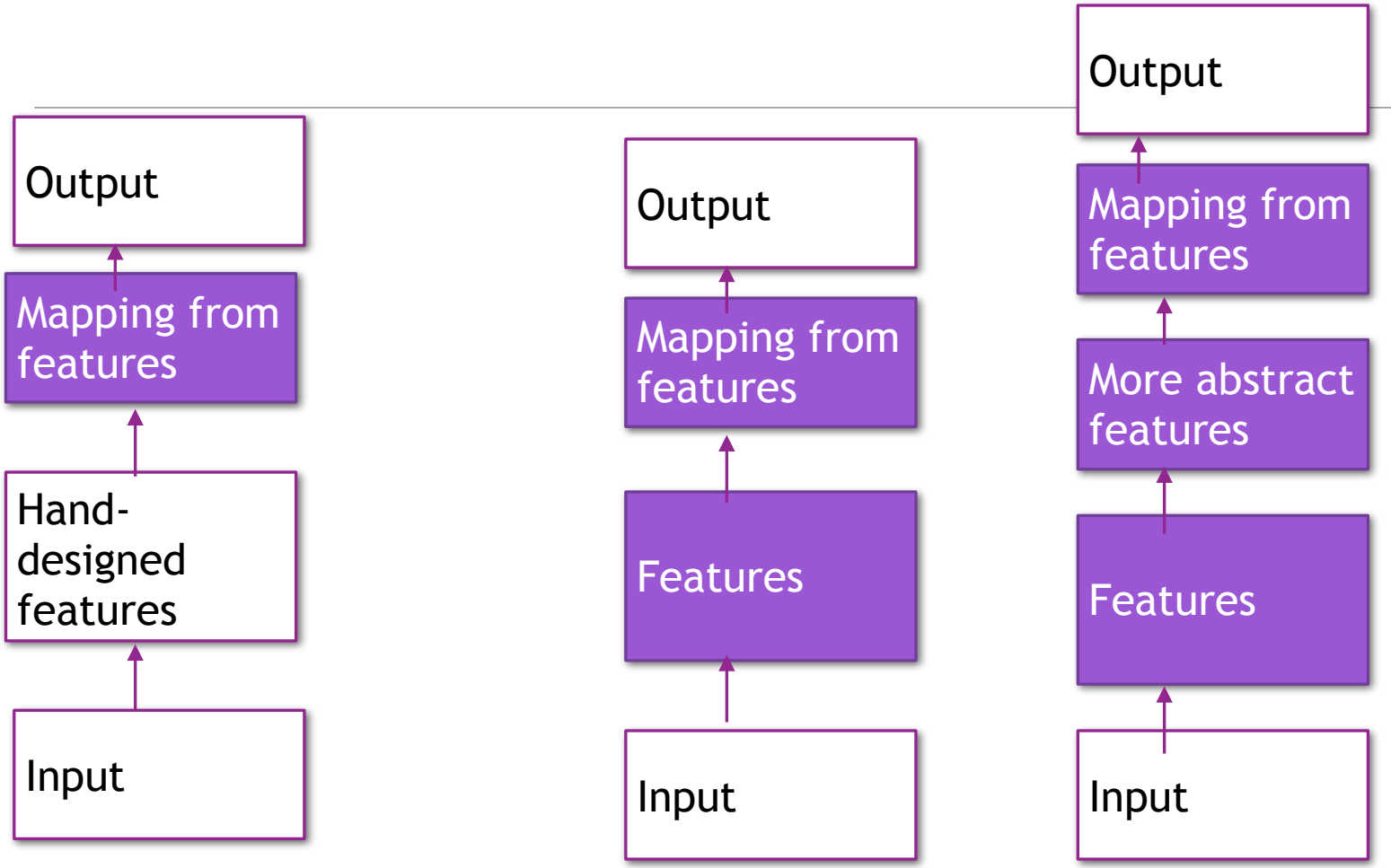
- # hidden units e.g. 20-100
- # layers 2-6

Log scale example:

- If you want $\alpha \in [0.00001, 10]$ then choose uniformly at random in the range $r \in [\log_{10} 0.00001, \log_{10} 10]$ then $\alpha = 10^r$

Learn more at: https://cs231n.github.io/neural-networks-3/

"As of 2016, a rough rule of thumb is that a supervised deep learning algorithm will generally achieve acceptable performance with around 5,000 labeled examples per category, and will match or exceed human performance when trained with a dataset containing at least 10 million labeled examples." Ian Goodfellow, Yoshua Bengio and Aaron Courville



"multiple levels of composition"

Logistic regression
Linear regression, etc

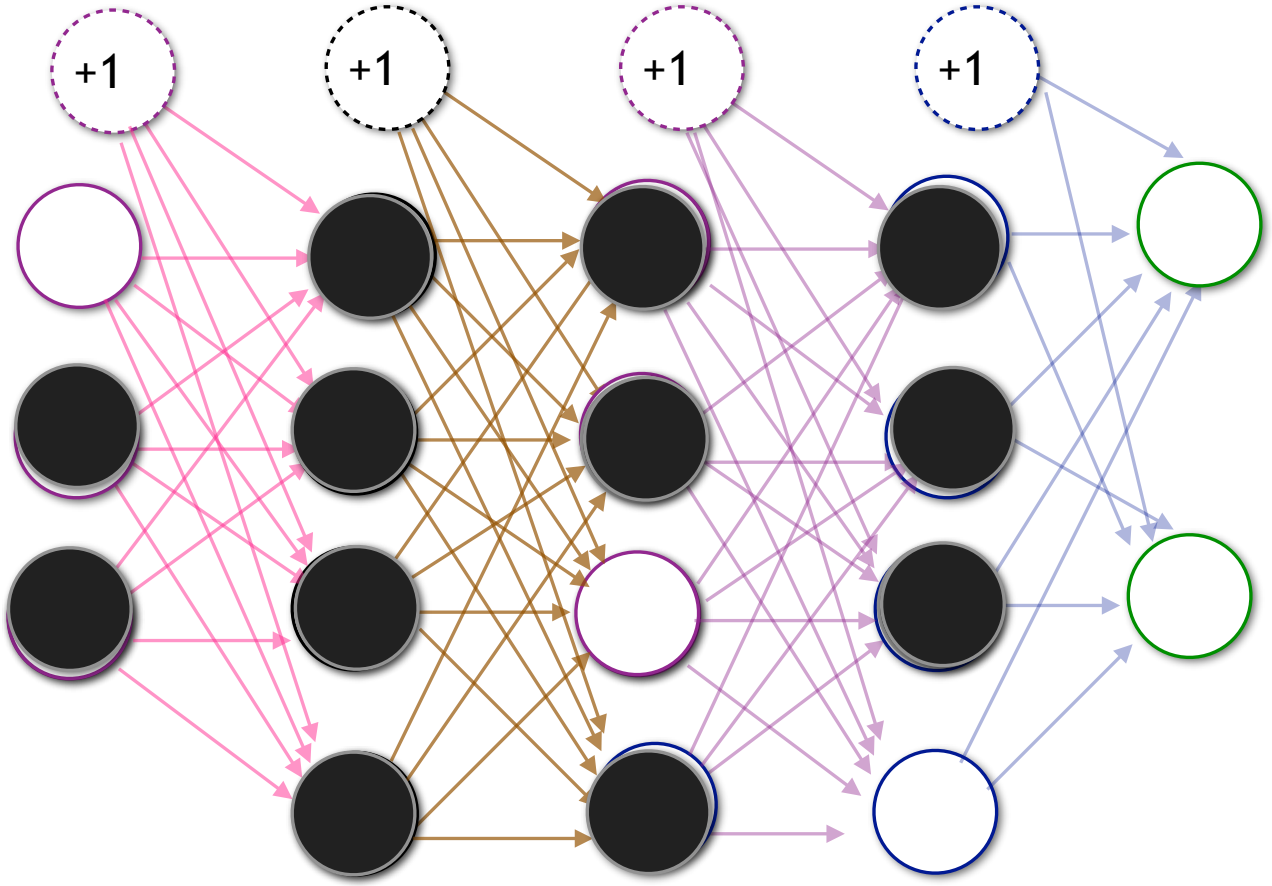
Deep learning
Representation learning

Graph modified from Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville

Regularization: Dropout

Each iteration, randomly set some activations of hidden neurons to 0 (typically 50%)

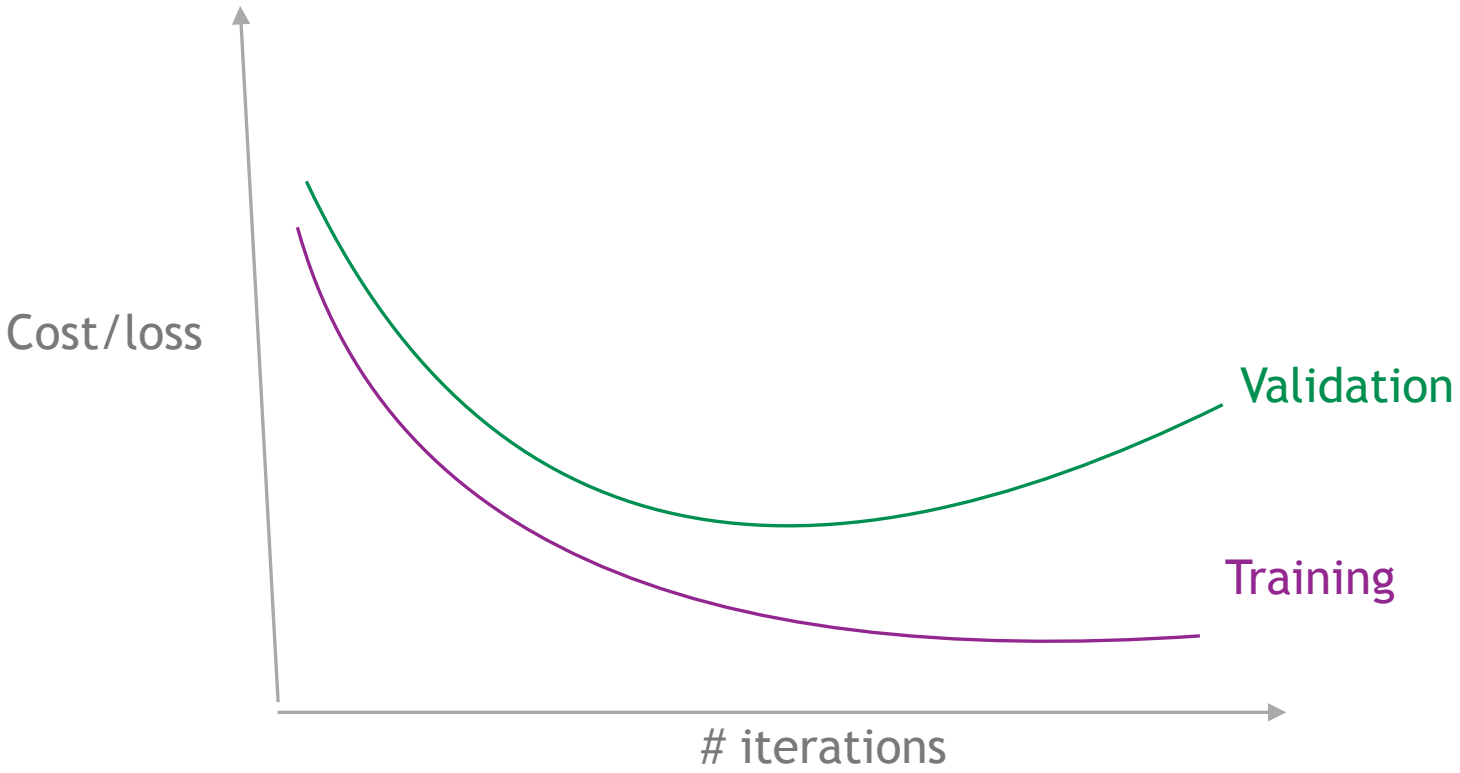
Require the network to have multiple ways to predict(i.e. cannot rely on any 1 node or path through the network)



Regularization: Early stopping

Typically wait until it is above the min for some time.

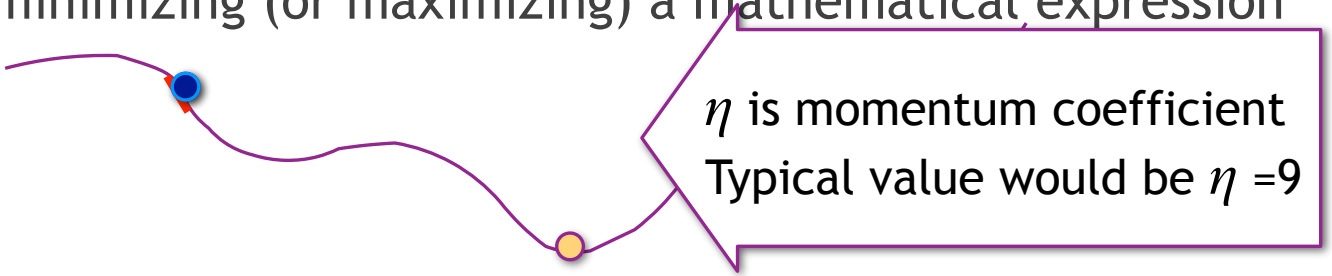
Stop training before it is overfitting



Optimizers

Our goal is to minimizing our **loss function** (to improve prediction performance)

Optimization is minimizing (or maximizing) a mathematical expression



- **momentum**: update rule is a sum of previous update rules and the current derivative. Previous updates accumulate to influence the current direction (e.g. how gravity works on a ball)

$$m^{\text{new}} = \eta \cdot m^{\text{old}} - \alpha \nabla_\theta J(\theta)$$
$$\theta^{\text{new}} = \theta^{\text{old}} + m^{\text{new}}$$

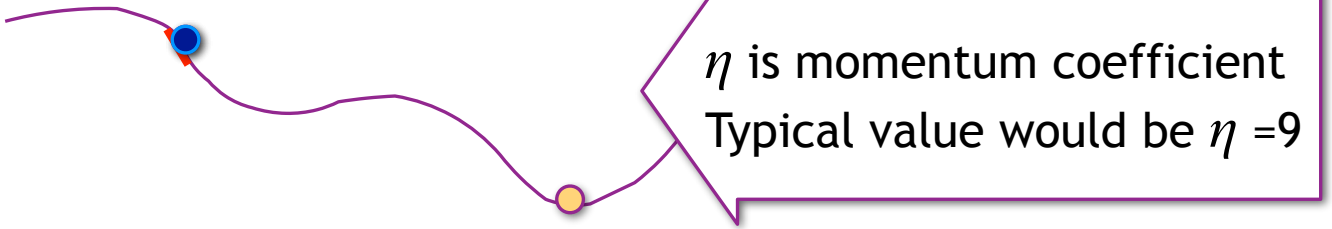
where m^{new} is the **momentum factor**, η is the **coefficient of momentum** (e.g. how much of the previous "momentum" we **carry forward**)

See momentum in action: https://distill.pub/2017/momentum/

Optimizers

Our goal is to minimizing our **loss function** (to improve prediction performance)

Optimization is minimizing (or maximizing) a mathematical expression



- **momentum**: update rule is a sum of previous update rules and the current derivative. Previous updates accumulate to influence the current direction (e.g. how gravity works on a ball)

$$m^{\text{new}} = \eta \cdot m^{\text{old}} - \alpha \nabla_\theta J(\theta)$$
$$\theta^{\text{new}} = \theta^{\text{old}} + m^{\text{new}}$$

where m^{new} is the **momentum factor**, η is the **coefficient of momentum** (e.g. how much of the previous "momentum" we **carry forward**)

See momentum in action: https://distill.pub/2017/momentum/