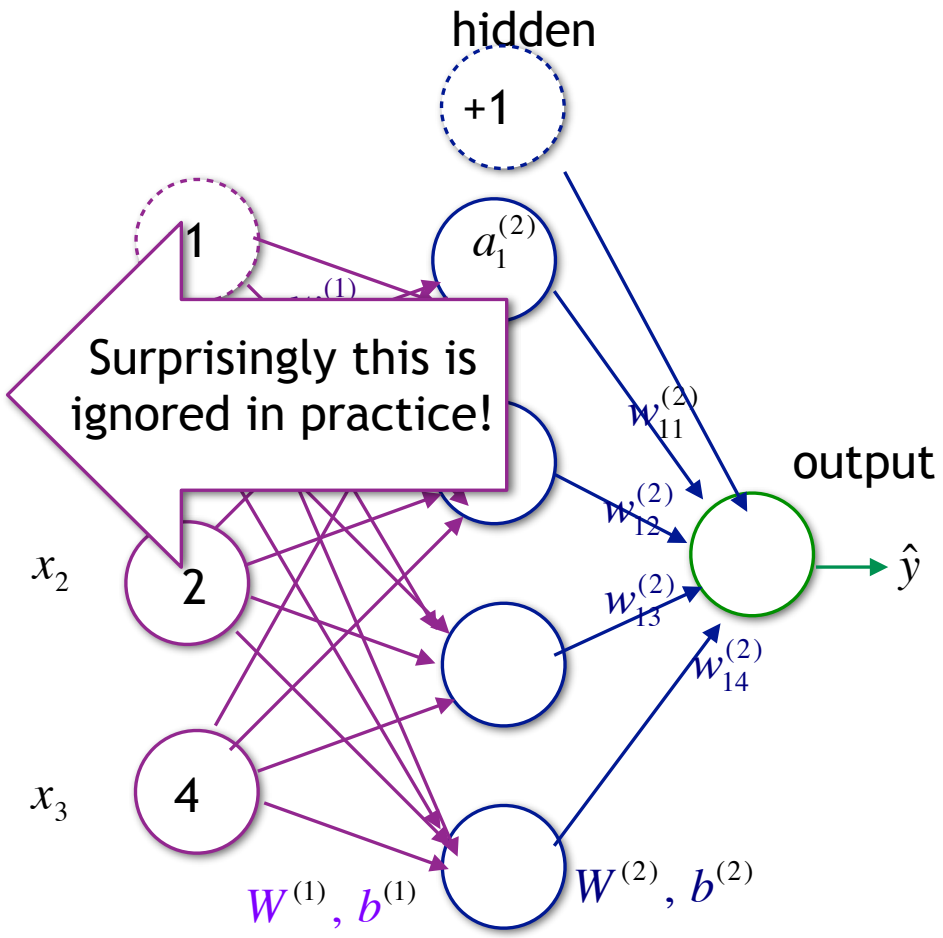
http://www.offconvex.org/2016/12/20/backprop/

How do we train a neural network?

FIRST SCALE THE TARGET VALUES [0,1] FOR SIGMOID ACTIVATION

Problems!

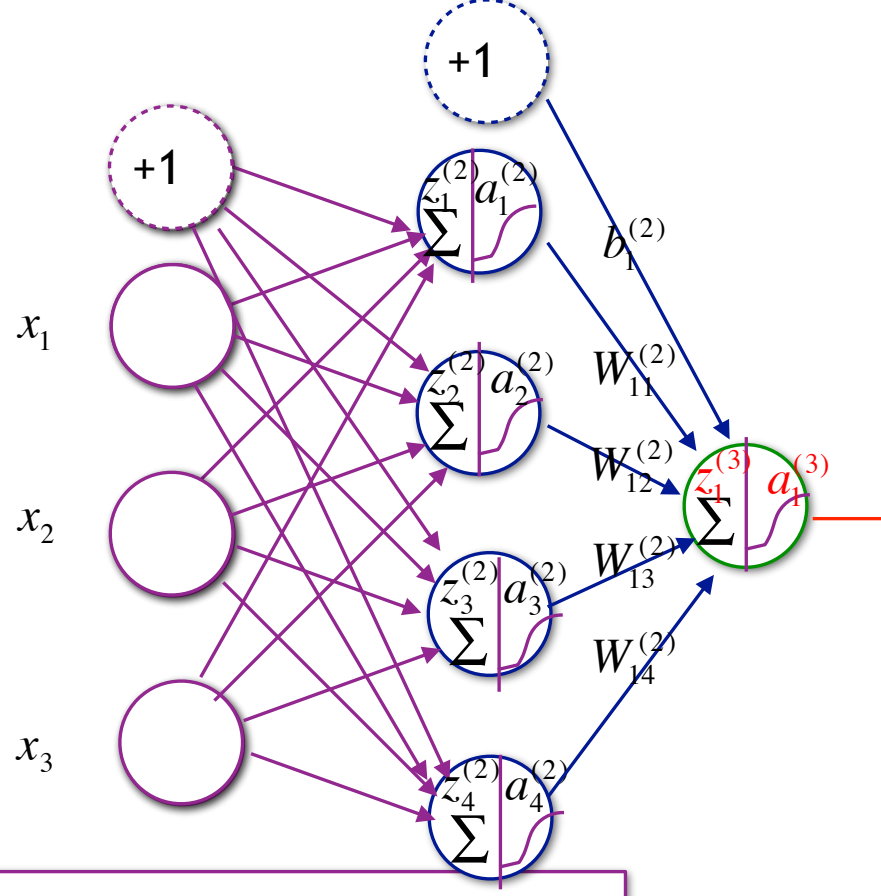
- We would like to use gradient descent to update the weights…
- Gradient descent worked before because we had a convex function… we no longer can assume our function is convex
- Taking the derivative of the entire neural network is too time consuming
- Solutions?
  - We can find the derivative for one level when the other levels are fixed
  - We can try different initial weights and retrain with different initial configurations



Training a neural network

- Similar to logistic regression and linear regression, we use the training examples and the derivative of the cost function to move toward an optimal solution
- We will first create a stochastic algorithm where we adapt the weights after observing only one training example.
- If the neural network correctly identifies the example - no changes are made to the weights
- If the neural network incorrectly identifies the example - the weights are updated the reduce the error
- How do you think we can update the weights?
  - The weights on the output level are straightforward if we knew the cost function

$W_{\text{new}} = W_{\text{old}} - \alpha \nabla \text{error}$ $\quad W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \alpha \dfrac{\partial J(W,b)}{\partial W_{ij}^{(\ell)}} \quad b_i^{(\ell)} = b_i^{(\ell)} - \alpha \dfrac{\partial J(W,b)}{\partial b_i^{(\ell)}}$

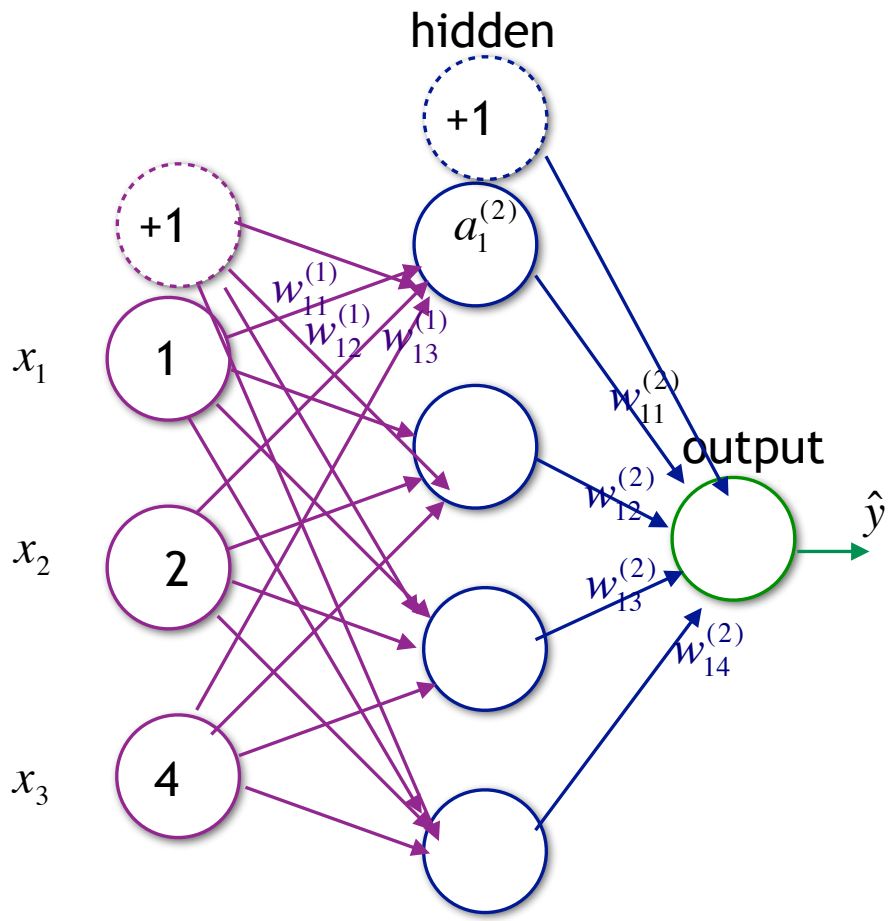


Gradient descent algorithm depends on the cost function and the structure of the network

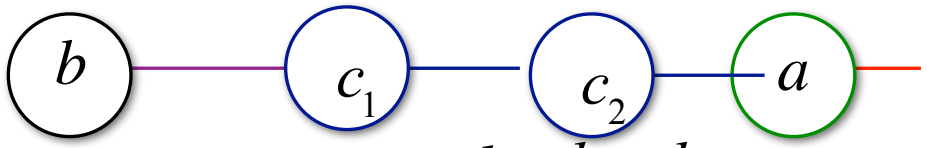
Goal

- Given
  - The target for the output unit y
  - The input is **x**
  - The output produced by the neural network is ŷ
- "Train" the network by performing gradient descent:
  - If the output is correct, don't change the weights
  - If the output is wrong, change weights:
    - If output is low, increment weights
    - If output is high, decrement weights

$$W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \alpha \frac{dJ}{dW_{ij}^{(\ell)}} \qquad b_i^{(\ell)} = b_i^{(\ell)} - \alpha \frac{dJ}{db_i^{(\ell)}}$$

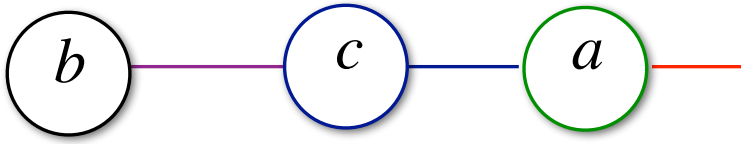


Basic Facts:



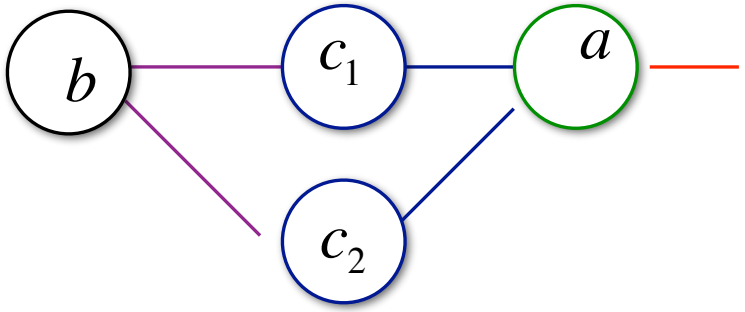
$$\frac{da}{db} = \frac{da}{dc_2}\frac{dc_2}{dc_1}\frac{dc_1}{db}$$

values propagate this way

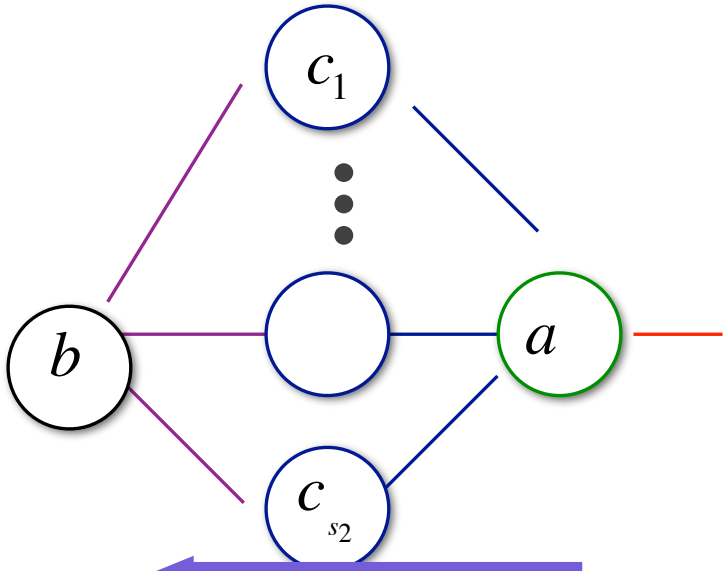


$$\frac{da}{db} = \frac{da}{dc}\frac{dc}{db}$$

values propagate this way



$$\frac{da}{db} = \frac{da}{dc_1}\frac{dc_1}{db} + \frac{da}{dc_2}\frac{dc_2}{db}$$



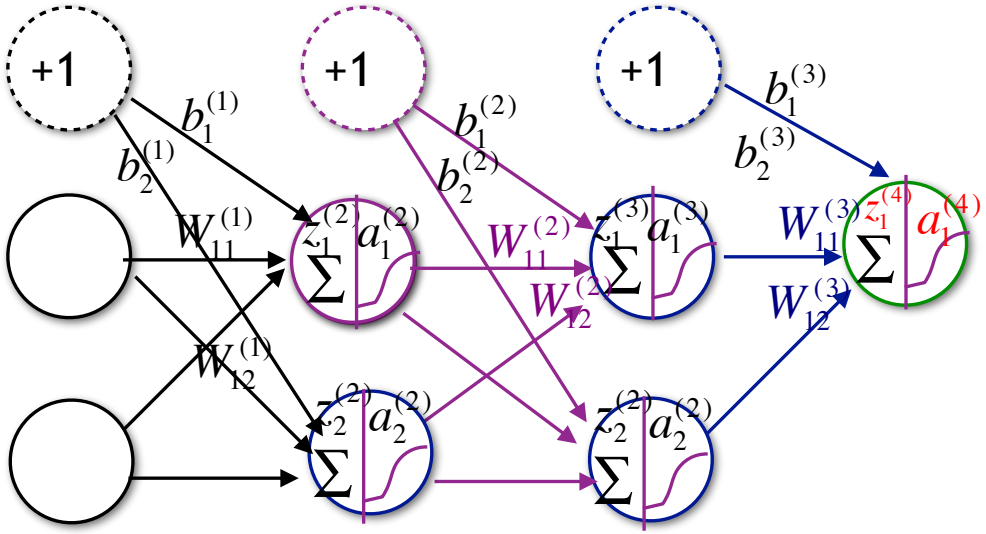
values propagate this way

$$\frac{da}{db} = \sum_{i=1}^{s_\ell} \frac{da}{dc_i}\frac{dc_i}{db}$$

We update the weights as we did for logistic regression and linear regression In neural networks

Due the the many layers in neural networks we need to repeatedly apply the chain rule: again and again and again …

More Basic Facts



$$\hat{y}_1 = a_1^{(n_\ell)} = f(z_1^{(n_\ell)}) \qquad J(W,b,x,y) = \frac{1}{2}(y-\hat{y})^2 \qquad \frac{dJ}{d\hat{y}} = -(y-\hat{y})$$

$$a_j^{(k)} = f(z_j^{(k)}) = \frac{1}{1+e^{-z_j^{(k)}}}$$

$$z_j^{(k)} = \sum_{i=1}^{s_{k-1}} W_{ji}^{(k-1)} a_i^{(k-1)} + b_j^{(k-1)} \qquad \frac{dz_j^{(k)}}{dW_{ji}^{(k-1)}} = a_i^{(k-1)} \qquad \frac{dz_j^{(k)}}{db_j^{(k-1)}} = 1$$

$$f(z) = \frac{1}{1+e^{-z}} \qquad \frac{df(z)}{dz} = f(z)(1-f(z))$$

How to update the weights…

$W_{\text{new}} = W_{\text{old}} - \alpha \nabla \text{error}$

$$W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial W_{ij}^{(\ell)}}$$

$$b_i^{(\ell)} = b_i^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial b_i^{(\ell)}}$$

$$J(W, b, x, y) = \frac{1}{2}(y - \hat{y})^2$$

$$z_j^{(k)} = \sum_{i=1}^{s_{k-1}} W_{ji}^{(k-1)} a_i^{(k-1)} + b_j^{(k-1)}$$

$$\frac{dz_j^{(k)}}{dW_{ji}^{(k-1)}} = a_i^{(k-1)}$$

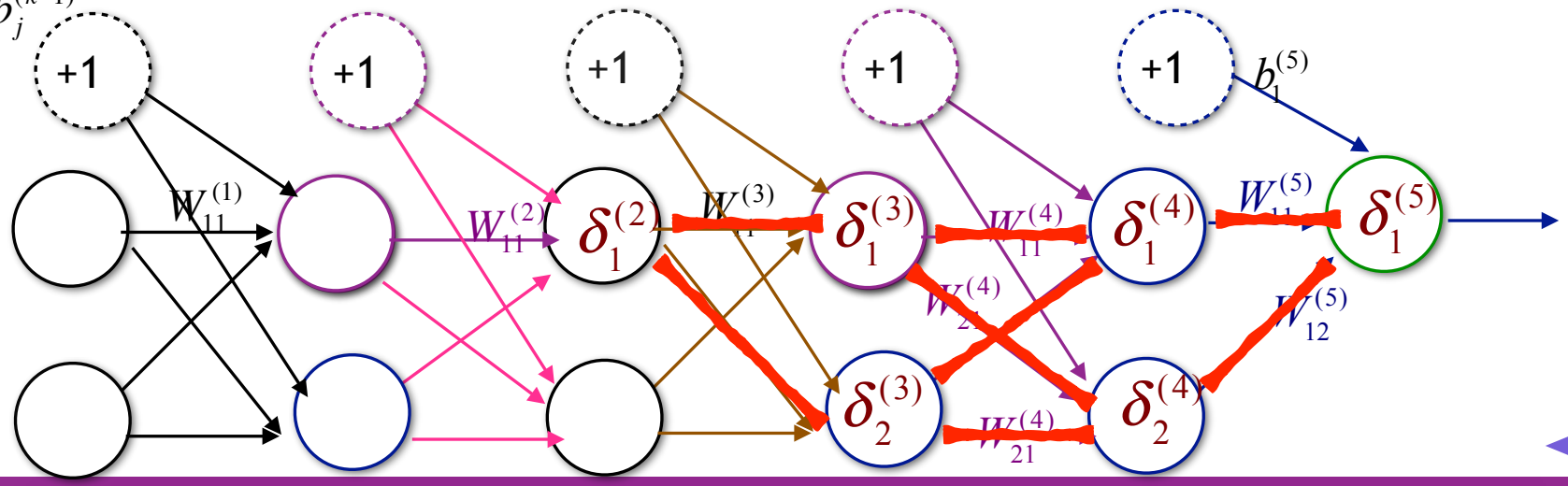
$$\delta_j^{(\ell)} = \frac{dJ}{dz_j^{(\ell)}}$$

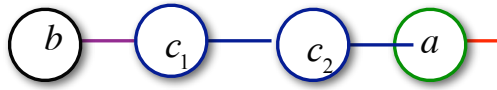
$$\frac{\partial J}{\partial W_{11}^5} = \frac{\partial J}{\partial z_1^5}\frac{\partial z_1^5}{\partial W_{11}^5} = \delta_1^{(5)}\frac{\partial z_1^5}{\partial W_{11}^5} = -(y - f(z_1^5))f'(z_1^5)\, a_1^{(4)}$$

$$\frac{\partial J}{\partial W_{11}^4} = \frac{\partial J}{\partial z_1^4}\frac{\partial z_1^4}{\partial W_{11}^4} = \delta_2^{(4)}\frac{\partial z_1^4}{\partial W_{11}^4}$$

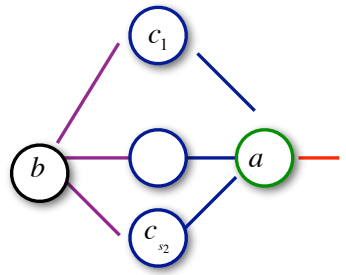
$$\frac{\partial J}{\partial W_{11}^3} = \frac{\partial J}{\partial z_1^3}\frac{\partial z_1^3}{\partial W_{11}^4} = \delta_1^{(3)}\frac{\partial z_1^3}{\partial W_{11}^4}$$

$$\frac{\partial J}{\partial W_{11}^2} = \delta_1^{(2)}\frac{\partial z_1^2}{\partial W_{11}^4} = \delta_1^{(2)}\frac{\partial z_1^2}{\partial W_{11}^4}$$





$$\frac{da}{db} = \frac{da}{dc_2}\frac{dc_2}{dc_1}\frac{dc_1}{db}$$



$$\frac{da}{db} = \sum_{i=1}^{s_\ell} \frac{da}{dc_i}\frac{dc_i}{db}$$

values propagate this way

What would the formula be if the the number of node ("width") was 10?

Determining the derivatives:

Definitions: $J(W,b,x,y)=\frac{1}{2}(y-\hat{y})^2 \qquad \hat{y}=h_{W,b}(x)$

$$\hat{y}_j = a_j^{(n_\ell)} = f(z_j^{(n_\ell)}) \qquad a_j^{(k)} = f(z_j^{(k)}) \qquad z_j^{(k)} = \sum_{i=1}^{s_{k-1}} W_{ji}^{(k-1)} a_i^{(k-1)} + b_j^{(k-1)} \qquad \frac{dz_j^{(k)}}{dW_{ji}^{(k-1)}} = a_i^{(k-1)} \qquad \boxed{\delta_j^{(\ell)} = \frac{dJ}{dz_j^{(\ell)}}}$$

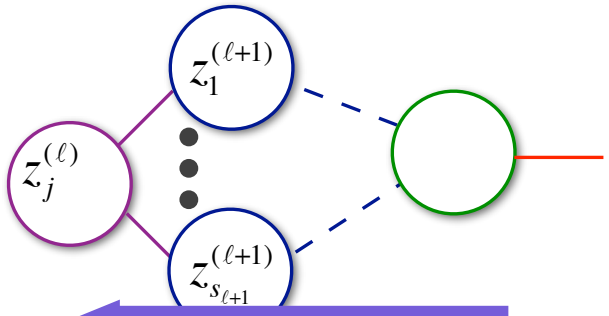
For the output layer: $\delta_j^{(n_\ell)} = \frac{dJ}{dz_j^{n_\ell}} = -(y - f(z_j^{n_\ell}))f'(z_j^{n_\ell})$

The derivatives: $\frac{dJ}{dW_{ij}^{\ell}} = \frac{dJ}{dz_i^{l+1}}\frac{dz_i^{l+1}}{dW_{ij}^{\ell}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$

For the other layers:

$$\delta_j^{(\ell)} = \frac{dJ}{dz_j^{\ell}} = \sum_{i=1}^{s_{\ell+1}} \frac{dJ}{dz_i^{\ell+1}}\frac{dz_i^{\ell+1}}{dz_j^{\ell}} = \sum_{i=1}^{s_{\ell+1}} \delta_i^{(\ell+1)} \frac{dz_i^{(\ell+1)}}{dz_j^{(\ell)}} = \sum_{i=1}^{s_{\ell+1}} \delta_i^{(\ell+1)} W_{ij}^{(\ell)} f'(z_j^{(\ell)})$$

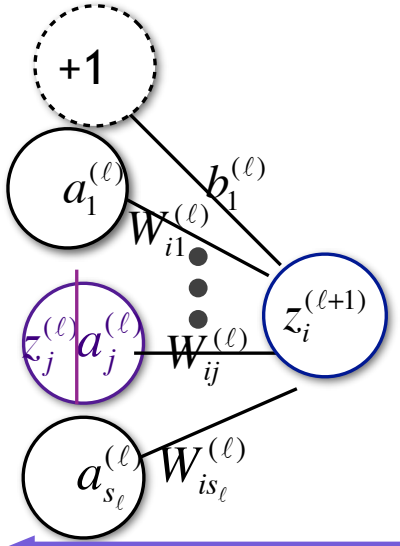
$$\frac{dJ}{db_i^{\ell}} = \frac{dJ}{dz_i^{l+1}}\frac{dz_i^{l+1}}{db_i^{\ell}} = \delta_i^{(\ell+1)}$$



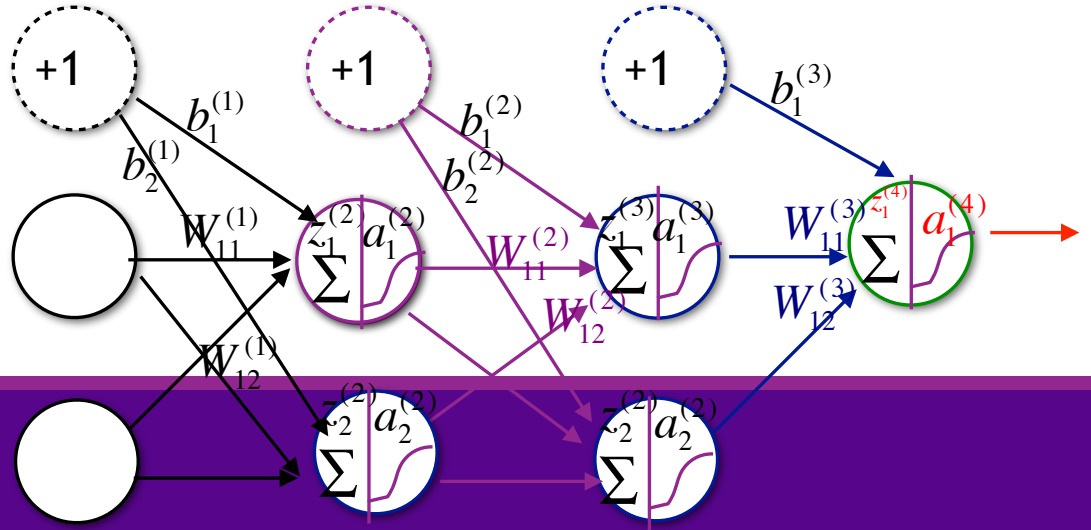
values propagate this way

$$z_i^{(\ell+1)} = \sum_{k=1}^{s_\ell} W_{ik}^{\ell} a_k^{\ell} + b_k^{\ell} = \sum_{k=1}^{s_\ell} W_{ik}^{\ell} f(z_k^{\ell}) + b_k^{\ell}$$

$$\frac{dz_i^{(\ell+1)}}{dz_j^{(\ell)}} = W_{ij}^{\ell}\frac{da_j^{(\ell)}}{dz_j^{(\ell)}} = W_{ij}^{\ell} f'(z_j^{(\ell)}) \qquad \frac{da_j^{(\ell)}}{dz_j^{(\ell)}} = f'(z_j^{(\ell)})$$

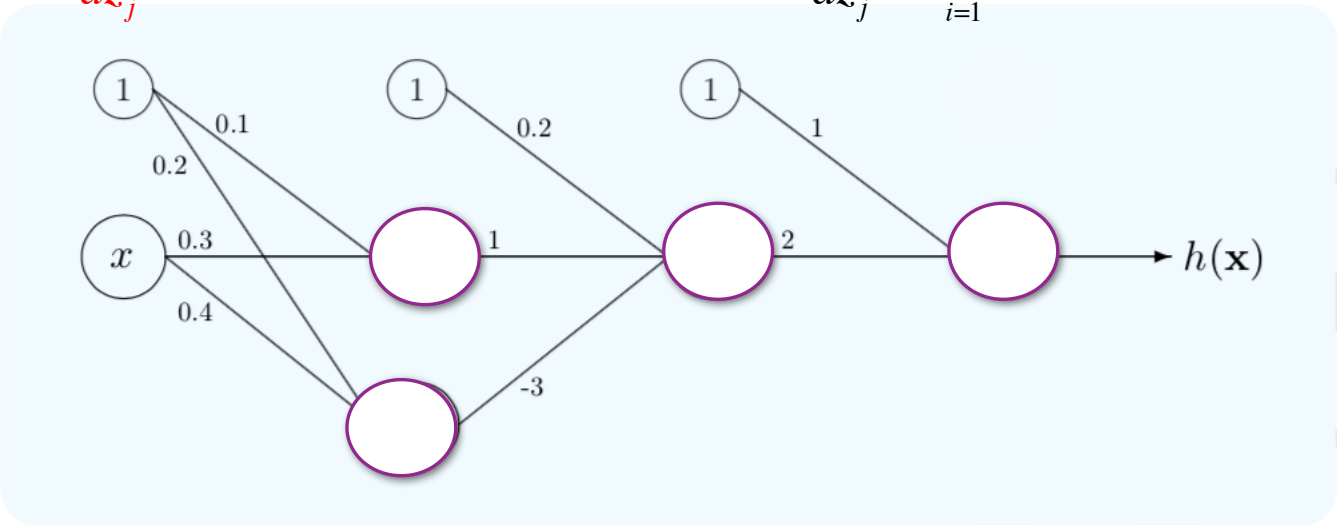


values propagate this way



$$\delta_j^{(n_\ell)} = \frac{dJ}{dz_j^{n_\ell}} = -(y - f(z_j^{n_\ell}))f'(z_j^{n_\ell}) \qquad \delta_j^{(\ell)} = \frac{dJ}{dz_j^{\ell}} = \sum_{i=1}^{s_{\ell+1}} \delta_i^{(\ell+1)} W_{ij}^{(\ell)} f'(z_j^{(\ell)})$$

$$\frac{df(z)}{dz} = f(z)(1 - f(z)) \qquad \frac{dJ}{dW_{ij}^{\ell}} = \delta_i^{(\ell+1)} a_j^{(\ell)} \qquad \frac{dJ}{db_i^{\ell}} = \delta_i^{(\ell+1)}$$



$$W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial W_{ij}^{(\ell)}} \qquad b_i^{(\ell)} = b_i^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial b_i^{(\ell)}}$$

$W^{(1)} =$ $\qquad W^{(2)} =$ $\qquad W^{(3)} =$

$b^{(1)} =$ $\qquad b^{(2)} =$ $\qquad b^{(3)} =$

If x = 2 and y =1, forward propagation gives:

$x = a^{(1)}$ $\qquad z^{(2)}$ $\qquad a^{(2)}$ $\qquad z^{(3)}$ $\qquad a^{(3)}$ $\qquad z^{(4)}$ $\qquad a^{(4)}$

$\delta^{(4)}$ $\qquad \delta^{(3)}$ $\qquad \delta^{(2)}$

$\frac{\partial J}{\partial W^{(1)}} =$ $\qquad \frac{\partial J}{\partial W^{(2)}} =$ $\qquad \frac{\partial J}{\partial W^{(3)}} =$

Similar to Learning from Data example

Implementing Gradient Descent!

We will use the calculations we just derived in our algorithm

Matrix-Vector Notation: $\delta_j^{(\ell)} = \dfrac{dJ}{dz_j^{(\ell)}}$

Hadamard product: element-wise product operator

$$a = b \bullet c \qquad a_i = b_i c_i \qquad f'([z_1, z_2, z_3]^T) = [f'(z_1), f'(z_2), f'(z_3)]^T$$

Our formulas

Vectorized formulas

The output layer:

$$\delta_j^{(n_\ell)} = \frac{dJ}{dz_j^{n_\ell}} = -(y - f(z_j^{n_\ell}))f'(z_j^{n_\ell}) \qquad \delta^{(n_\ell)} = -(y - f(z^{n_\ell})) \bullet f'(z^{n_\ell})$$

The other layers:

$$\delta_j^{(\ell)} = \frac{dJ}{dz_j^{\ell}} = \sum_{i=1}^{s_{\ell+1}} \delta_i^{(\ell+1)} W_{ij}^{(\ell)} f'(z_j^{(\ell)}) \qquad \delta^{(\ell)} = \left(\left(W^{(\ell)}\right)^T \delta^{(\ell+1)}\right) \bullet f'(z^{(\ell)})$$

$$W^{(\ell)} = \begin{bmatrix} W_{11}^{(\ell)} & W_{12}^{(\ell)} & \cdots & W_{1s_\ell}^{(\ell)} \\ W_{21}^{(\ell)} & W_{22}^{(\ell)} & & W_{2s_\ell}^{(\ell)} \\ \vdots & & & \vdots \\ W_{s_{\ell+1}1}^{(\ell)} & W_{s_{\ell+1}2}^{(\ell)} & \cdots & W_{s_{\ell+1}s_\ell}^{(\ell)} \end{bmatrix}$$

$$\left(W^{(\ell)}\right)^T = \begin{bmatrix} W_{11}^{(\ell)} & W_{21}^{(\ell)} & \cdots & W_{s_{\ell+1}1}^{(\ell)} \\ W_{12}^{(\ell)} & W_{22}^{(\ell)} & & W_{s_{\ell+1}2}^{(\ell)} \\ \vdots & & & \vdots \\ W_{1s_\ell}^{(\ell)} & W_{2s_\ell}^{(\ell)} & \cdots & W_{s_{\ell+1}s_\ell}^{(\ell)} \end{bmatrix}$$

$$\delta^{(\ell)} = \begin{bmatrix} \delta_1^{(\ell)} \\ \delta_2^{(\ell)} \\ \vdots \\ \delta_{s_\ell}^{(\ell)} \end{bmatrix} = \begin{bmatrix} W_{11}^{(\ell)} & W_{21}^{(\ell)} & \cdots & W_{s_{\ell+1}1}^{(\ell)} \\ W_{12}^{(\ell)} & W_{22}^{(\ell)} & & W_{s_{\ell+1}2}^{(\ell)} \\ \vdots & & & \vdots \\ W_{1s_\ell}^{(\ell)} & W_{2s_\ell}^{(\ell)} & \cdots & W_{s_{\ell+1}s_\ell}^{(\ell)} \end{bmatrix} \begin{bmatrix} \delta_1^{(\ell+1)} \\ \delta_2^{(\ell+1)} \\ \vdots \\ \delta_{s_{\ell+1}}^{(\ell+1)} \end{bmatrix} \bullet \begin{bmatrix} f'(z_1^{(\ell)}) \\ f'(z_2^{(\ell)}) \\ \vdots \\ f'(z_{s_\ell}^{(\ell)}) \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{s_{\ell+1}} W_{i1}^{(\ell)} \delta_i^{(\ell+1)} \\ \sum_{i=1}^{s_{\ell+1}} W_{i2}^{(\ell)} \delta_i^{(\ell+1)} \\ \vdots \\ \sum_{i=1}^{s_{\ell+1}} W_{is_{\ell+1}}^{(\ell)} \delta_i^{(\ell+1)} \end{bmatrix} \bullet \begin{bmatrix} f'(z_1^{(\ell)}) \\ f'(z_2^{(\ell)}) \\ \vdots \\ f'(z_{s_\ell}^{(\ell)}) \end{bmatrix}$$

Matrix-Vector Notation:

Definition: $\delta_j^{(\ell)} = \dfrac{dJ}{dz_j^{(\ell)}}$ $\quad \dfrac{dJ}{dW_{ij}^{\ell}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$ $\quad \dfrac{dJ}{db_i^{\ell}} = \delta_i^{(\ell+1)}$

Hadamard product: element-wise product operator $a = b \bullet c$ $\quad a_i = b_i c_i$ $\quad f'([z_1, z_2, z_3]^T) = [f'(z_1), f'(z_2), f'(z_3)]^T$

The derivatives:

Our formulas

$$\frac{dJ}{dW_{ij}^{\ell}} = a_j^{(\ell)} \delta_i^{(\ell+1)}$$

$$\frac{dJ}{db_i^{\ell}} = \delta_i^{(\ell+1)}$$

Vectorized formulas

$$\frac{dJ}{dW^{\ell}} = \delta^{(\ell+1)} \left(a^{(\ell)}\right)^T$$

$$\frac{dJ}{db^{\ell}} = \delta^{(\ell+1)}$$

$$a^{(\ell)} = \begin{bmatrix} a_1^{(\ell)} \\ a_2^{(\ell)} \\ \vdots \\ a_{s_\ell}^{(\ell)} \end{bmatrix} \qquad \delta^{(\ell+1)} = \begin{bmatrix} \delta_1^{(\ell+1)} \\ \delta_2^{(\ell+1)} \\ \vdots \\ \delta_{s_{\ell+1}}^{(\ell+1)} \end{bmatrix}$$

$$\frac{dJ}{dW^{(\ell)}} = \begin{bmatrix} \frac{dJ}{dW_{11}^{(\ell)}} & \frac{dJ}{dW_{12}^{(\ell)}} & \cdots & \frac{dJ}{dW_{1s_\ell}^{(\ell)}} \\ \frac{dJ}{dW_{21}^{(\ell)}} & \frac{dJ}{dW_{22}^{(\ell)}} & & \frac{dJ}{dW_{2s_\ell}^{(\ell)}} \\ \vdots & & & \vdots \\ \frac{dJ}{dW_{s_{\ell+1}1}^{(\ell)}} & \frac{dJ}{dW_{s_{\ell+1}2}^{(\ell)}} & \cdots & \frac{dJ}{dW_{s_{\ell+1}s_\ell}^{(\ell)}} \end{bmatrix} = \begin{bmatrix} \delta_1^{(\ell+1)} \\ \delta_2^{(\ell+1)} \\ \vdots \\ \delta_{s_{\ell+1}}^{(\ell+1)} \end{bmatrix} \begin{bmatrix} a_1^{(\ell)} & a_2^{(\ell)} & \cdots & a_{s_\ell}^{(\ell)} \end{bmatrix}$$

Implementing the gradient step

$W_{\text{new}} = W_{\text{old}} - \alpha \nabla \text{error}$

$$W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial W_{ij}^{(\ell)}}$$

$$b_i^{(\ell)} = b_i^{(\ell)} - \alpha \frac{\partial J(W, b, \mathbf{x}, y)}{\partial b_i^{(\ell)}}$$

Going from ***stochastic*** gradient descent to ***batch*** gradient descent:

Vectorized formulas

$$\frac{dJ(W,b,x,y)}{dW^{\ell}} = \delta^{(\ell+1)}\left(a^{(\ell)}\right)^T$$

$$\frac{dJ(W,b,x,y)}{db^{\ell}} = \delta^{(\ell+1)}$$

$$\frac{dJ(W,b)}{dW^{(\ell)}} = \frac{1}{N}\sum_{i=1}^{N}\frac{dJ(W,b,x,y)}{dW^{(\ell)}}$$

$$\frac{dJ(W,b)}{db^{(\ell)}} = \frac{1}{N}\sum_{i=1}^{N}\frac{dJ(W,b,x,y)}{db^{(\ell)}}$$

Keeping track of a term that is summing up the values:

$$\Delta W^{(\ell)} = \sum_{i=1}^{N}\frac{dJ(W,b,x,y)}{dW^{(\ell)}}$$

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

Keeping track of a term that is summing up the values:

$$\Delta b^{(\ell)} = \sum_{i=1}^{N}\frac{dJ(W,b,x,y)}{db^{(\ell)}}$$

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

The batch gradient descent algorithm

Algorithm and code adapted from https://github.com/adventuresinML/adventures-in-ml-code/blob/master/neural_network_tutorial.py

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]

$W^{(1)}$

```
[[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
```

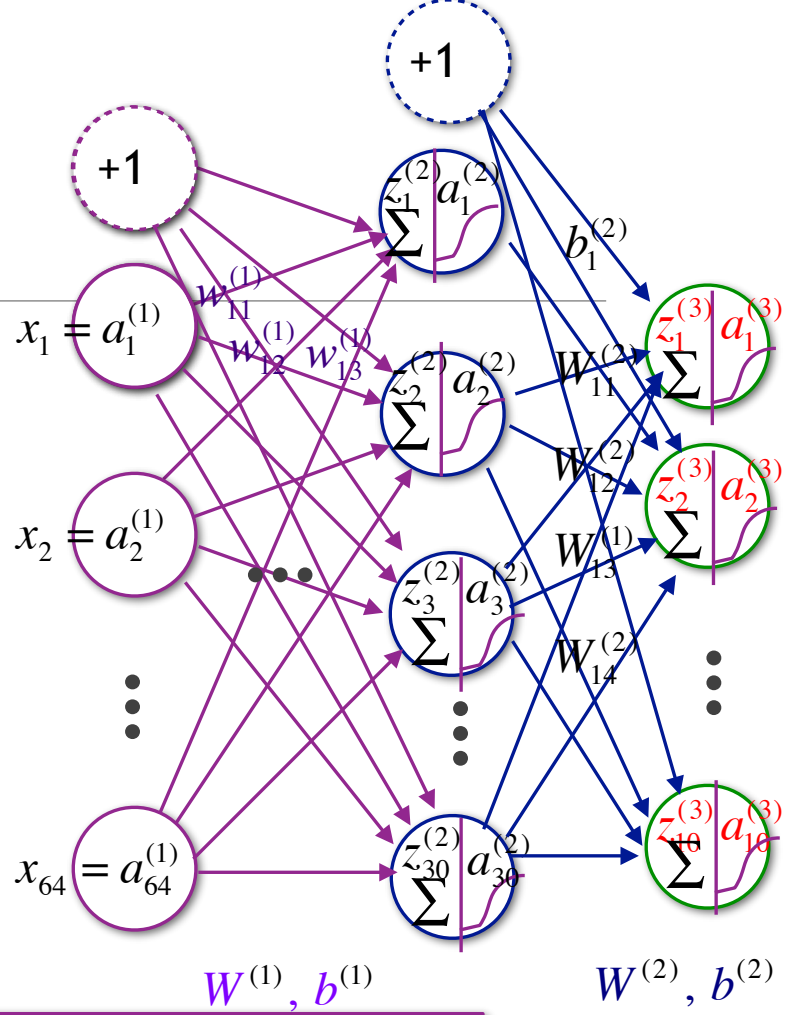
$W^{(2)}$

```
[[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 will represent it as column
Numpy represents it as a row

The gradient descent algorithm

Algorithm and code adapted from https://github.com/adventuresinML/adventures-in-ml-code/blob/master/neural_network_tutorial.py

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.]]
```

$\nabla W^{(1)}$

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

$\nabla b^{(1)}$

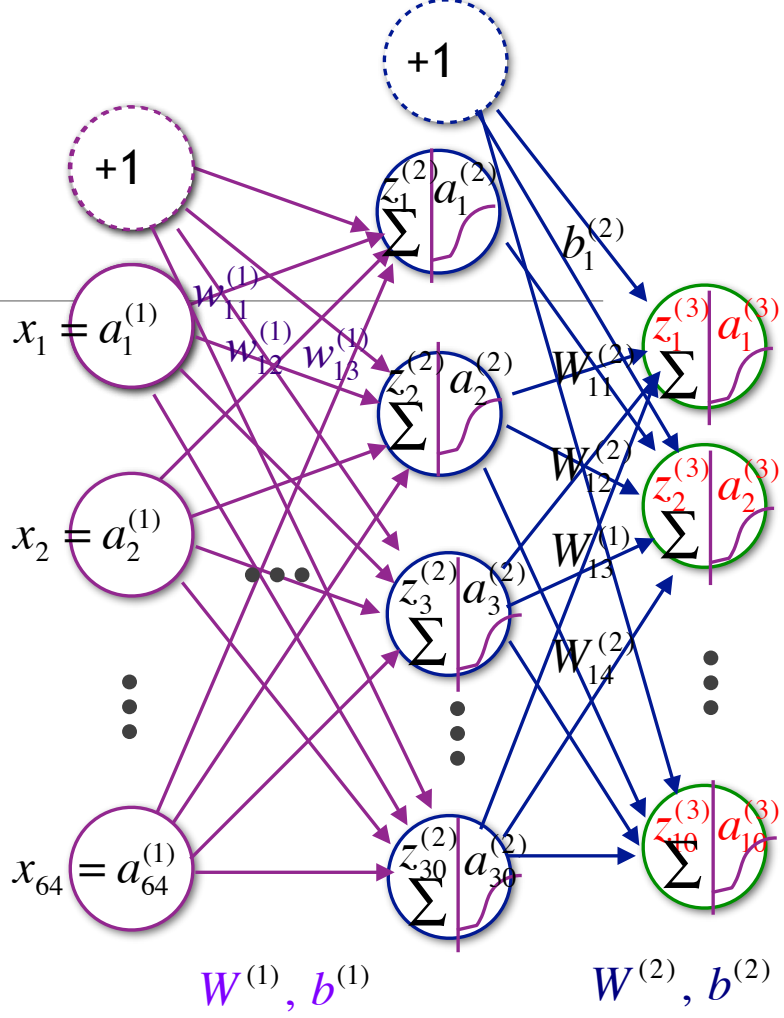
```
[[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]]
```

$\nabla W^{(2)}$

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

$\nabla b^{(2)}$

```
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 https://github.com/adventuresinML/adventures-in-ml-code/blob/master/neural_network_tutorial.py

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

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

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

- 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
 [0.06, 0.74, 0.72],   1
 [0.92, 0.29, 0.71],   1]
 [0.39, 0.41, 0.10]]
```

$+$

```
[0.07,
 0.35,
 0.60,
 0.50]
```

$=$

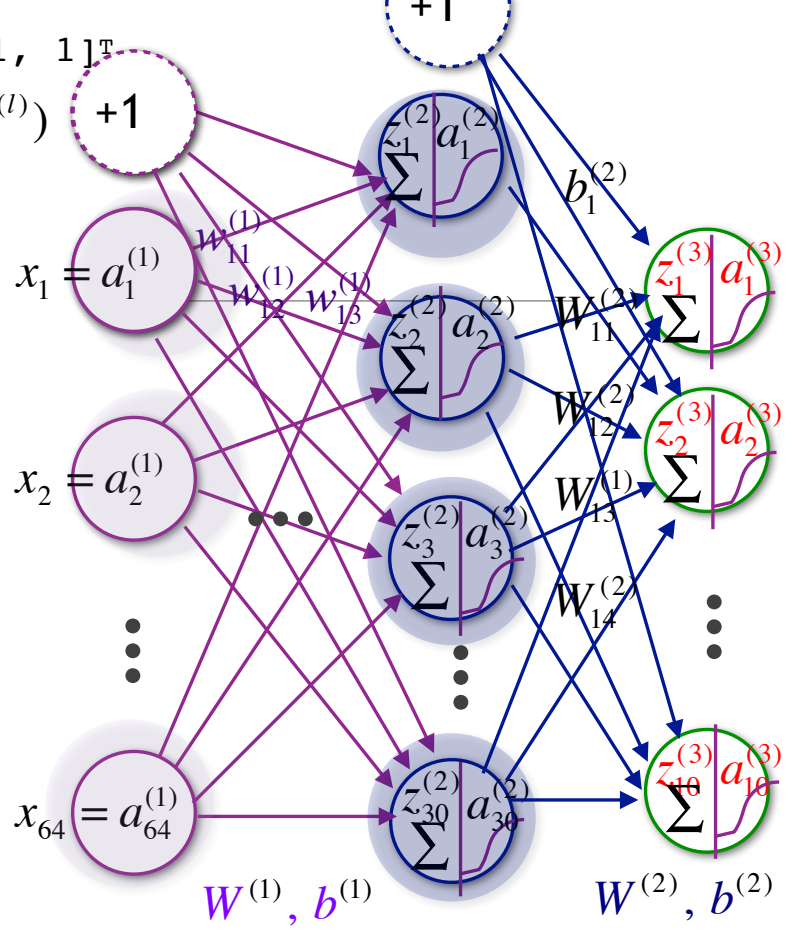
```
[1.80,
 1.88,
 2.53,
 1.39]
```

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

$$f\left(\begin{matrix}[1.80,\\ 1.88,\\ 2.53,\\ 1.39]\end{matrix}\right) = \begin{matrix}[0.86,\\ 0.87,\\ 0.93,\\ 0.80]\end{matrix}$$

```
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

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

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

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

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

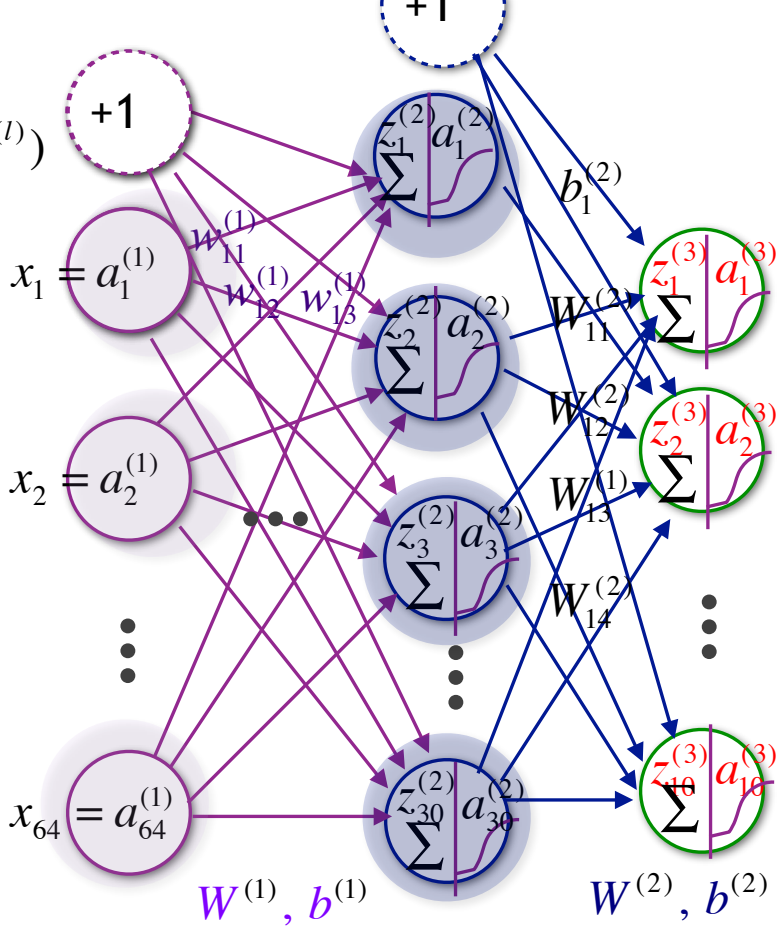
```
W(2)
 [[0.45, 0.64, 0.10, 0.82],[0.86,   0.14,
  [0.30, 0.25, 0.39, 0.36], 0.87, +  0.64,  =  [2.66,
  [0.78, 0.60, 0.16, 0.07]] 0.93,    0.62       1.76,
                            0.80]                2.01]
```

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

```
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 https://github.com/adventuresinML/adventures-in-ml-code/blob/master/neural_network_tutorial.py

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

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

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

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)
```

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

$$-\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0.935 \\ 0.854 \\ 0.882 \end{pmatrix} \bullet f'\begin{pmatrix} 2.66, \\ 1.76, \\ 2.01 \end{pmatrix} = -\begin{pmatrix} 0.065 \\ 0.146 \\ 0.118 \end{pmatrix} \bullet \begin{pmatrix} 0.06087702 \\ 0.12500965 \\ 0.10390214 \end{pmatrix} = \begin{matrix} -0.00396415 \\ -0.01830894 \\ -0.01223682 \end{matrix}$$

$$\delta^{(3)}$$

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 https://github.com/adventuresinML/adventures-in-ml-code/blob/master/neural_network_tutorial.py

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:

$$\frac{dJ}{dW^{\ell}} = \delta^{(\ell+1)}\left(a^{(\ell)}\right)^T$$

$$\frac{dJ}{db^{\ell}} = \delta^{(\ell+1)}$$

$$\delta_j^{(\ell)} = \frac{dJ}{dz_j^{(\ell)}}$$

$$\frac{dJ}{dW_{ij}^{\ell}} = \delta_i^{(\ell+1)} a_j^{(\ell)}$$

$$\frac{dJ}{db_i^{\ell}} = \delta_i^{(\ell+1)}$$

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)
```

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

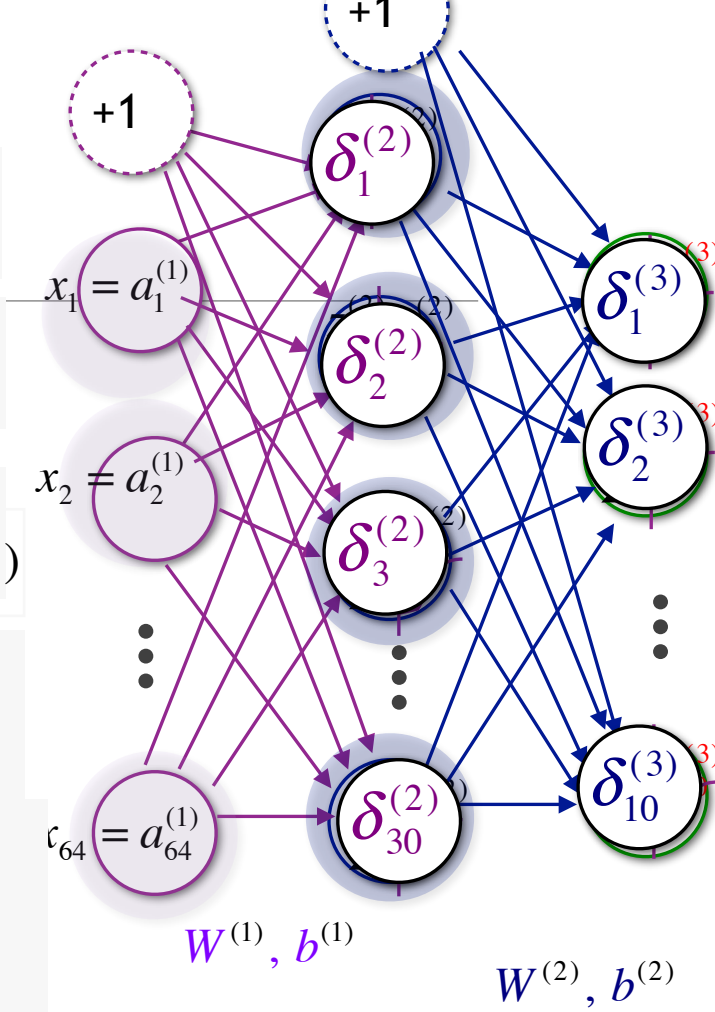
```
def calculate_hidden_delta(delta_plus_1, w_l, z_l):
    return np.dot(np.transpose(w_l), delta_plus_1) * f_deriv(z_l)
```

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

```
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]
```

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

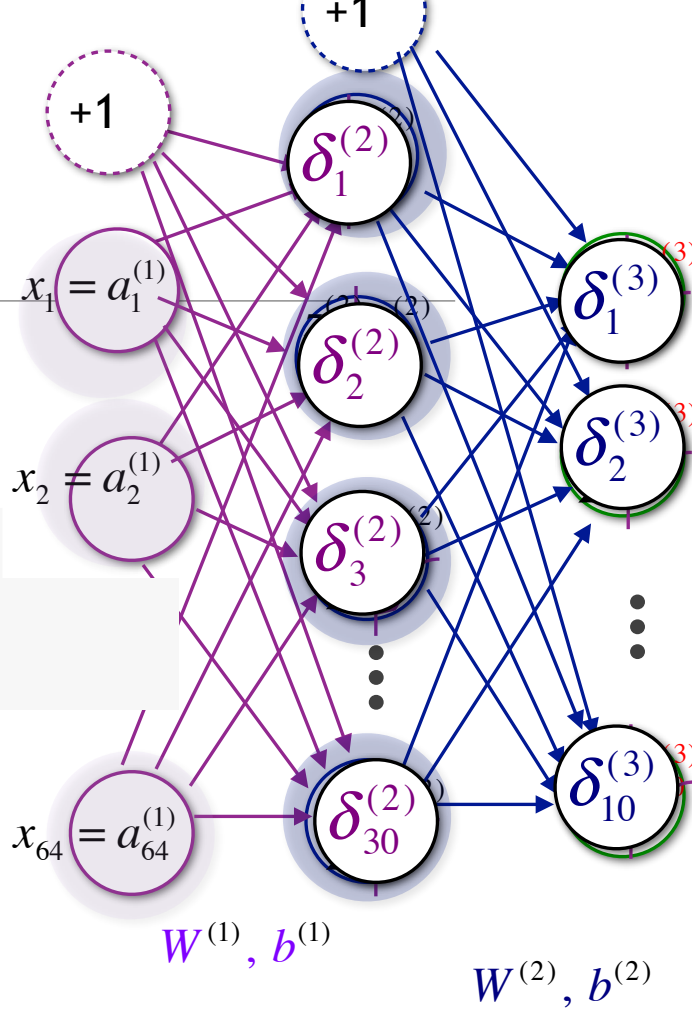
$\# \Delta b^{(\ell)} + \delta^{(\ell+1)}$



Computing

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

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



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

```
while cnt < iter_num:
    tri_W, tri_b = init_tri_values(nn_structure)
    avg_cost = 0
```

for i in range(len(y)):
  Code from the previous slide

```
    # 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])
    # complete the average cost calculation
    avg_cost = 1.0/N * avg_cost
    avg_cost_func.append(avg_cost)
    cnt += 1
```

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

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

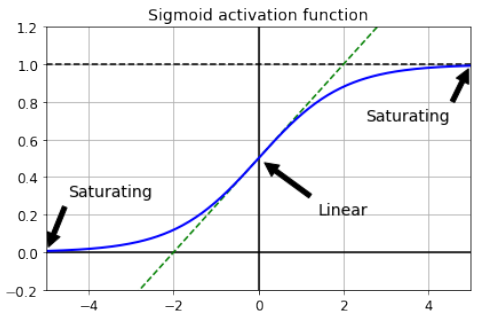
Exploding gradients problem: the gradients can get larger - too large. (This is mostly a problem in recurrent neural networks. Recurrent neural networks are not covered in this class)

Another problem is layers learn at different speeds

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)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Activations

deep neural nets can have saturation problems with sigmoid and tan

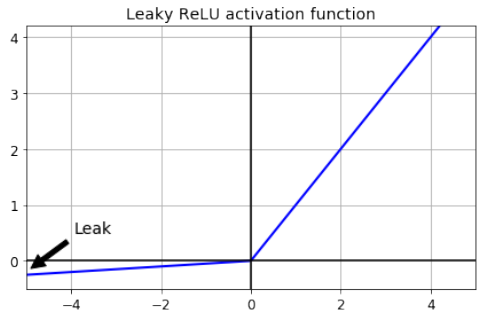


max(0,z)

ReLU - doesn't saturate, suffers from dying ReLU problem

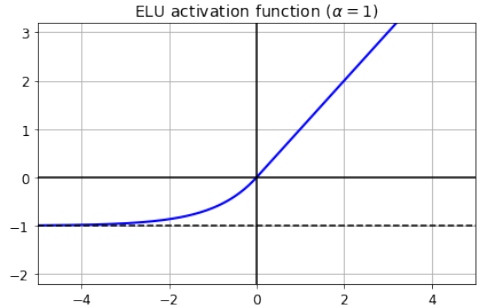
Leaky ReLU - typically hyper parameter $\alpha = 0.01$. This prevents leaky ReLU from dying. (Some results show $\alpha = 0.2$ is better)

max(αz,z)



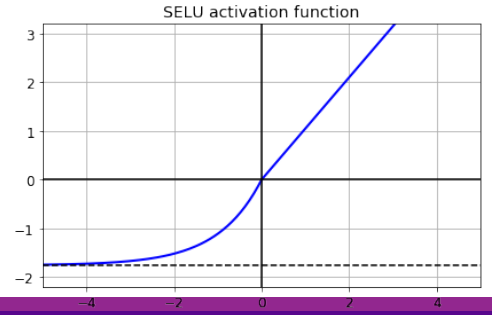
ELU (exponential linear unit) - typically hyper parameter $\alpha = 1$. Takes on negative values to prevent vanishing gradient, non zero gradient so doesn't have dying unit problem, smooth everywhere. However it is slower than ReLU and leaky ReLU due to the exponential function but often has faster convergence

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

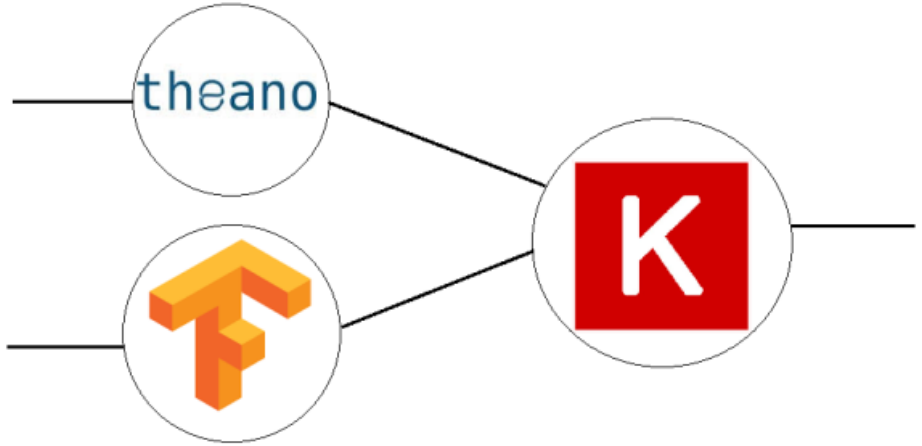


SELU (scaled exponential unit)

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



Keras Package



Keras
High-level language

Low-level backend
(e.g. Tensorflow, Theano, CNTK, …)

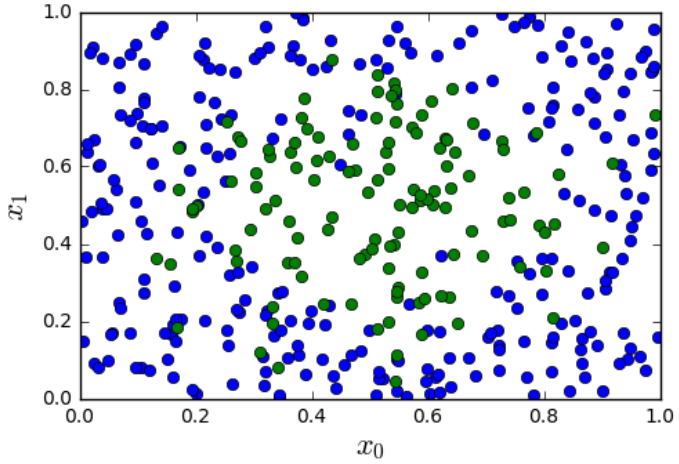
- High-level neural network language
- Wrapper around a complex backend
  - Much simpler than raw backend language (e.g. TensorFlow, or Theano)
  - Very fast coding
  - Uniform language for all backend
- But…
  - Slightly less flexible
  - Not as fast sometimes
- In this class, we use Keras

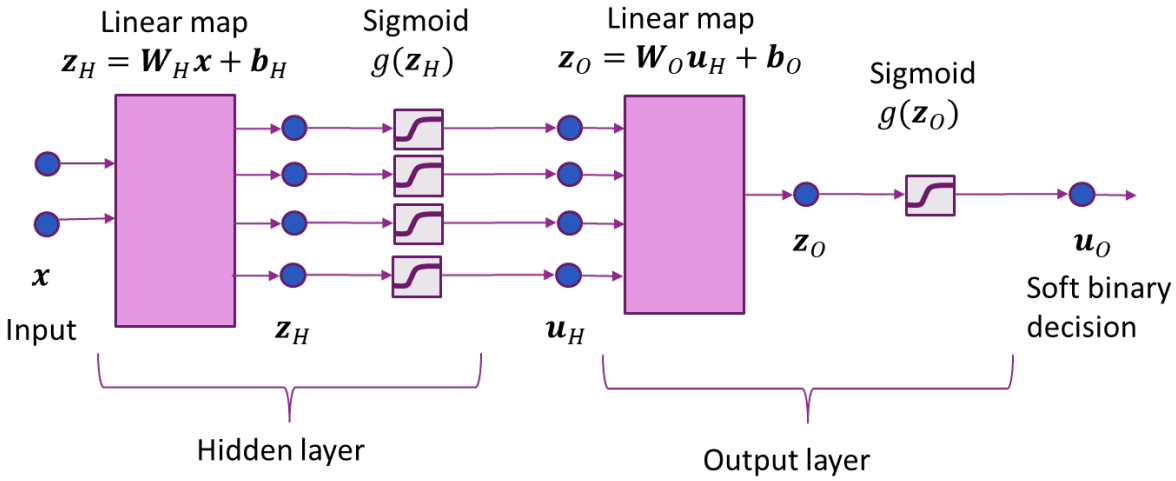
Keras Recipe

- Step 1. Describe model architecture
  - Number of hidden units, output units, activations, …
- Step 2. Select and optimizer (see http://ruder.io/optimizing-gradient-descent/ and https://keras.io/optimizers/)
- Step 3. Select a loss function and compile the model (see https://keras.io/losses/)
- Step 4. Fit the model
- Step 5. Test / use the model

Synthetic Data Example

- Try a simpler two-layer NN
  - Input = 2 dim
  - 4 hidden units
  - 1 output unit (binary classification)





Step 0: Import the Packages

- Install a deep learning backend: Tensorflow, Theano, CNTK, …
- Then install Keras

```
import keras
```

```
Using TensorFlow backend.
```

Step 1: Define Model

```
from keras.models import Model, Sequential
from keras.layers import Dense, Activation
```

```
import keras.backend as K
K.clear_session()
```

```
nin = nx  # dimension of input data
nh = 4    # number of hidden units
nout = 1  # number of outputs = 1 since this is binary
model = Sequential()
model.add(Dense(nh, input_shape=(nx,), activation='sigmoid', name='hidden'))
model.add(Dense(1, activation='sigmoid', name='output'))
```

- ❑Load modules for layers
- ❑Clear graph
- ❑Build model
  - This example: dense layers
  - Give each layer a dimension, name & activation

Step 2, 3: Select and Optimizer & Compile

```
from keras import optimizers

opt = optimizers.Adam(lr=0.01, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

- Adam optimizer generally works well for most problems
  - In this case, had to manually set learning rate
- Use binary cross-entropy loss (this is the loss function we used for logistic regression)
- Metrics indicate what will be printed in each epoch
- Learn more at https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Step 4: Fit the Model

```
model.fit(X, y, epochs=10, batch_size=100)
```

```
Epoch 1/10
400/400 [==============================] - 0s - loss: 0.8047 - acc: 0.3900
Epoch 2/10
400/400 [==============================] - 0s - loss: 0.7695 - acc: 0.3900
Epoch 3/10
400/400 [==============================] - 0s - loss: 0.7428 - acc: 0.3900
Epoch 4/10
400/400 [==============================] - 0s - loss: 0.7223 - acc: 0.3900
Epoch 5/10
400/400 [==============================] - 0s - loss: 0.7027 - acc: 0.4000
Epoch 6/10
400/400 [==============================] - 0s - loss: 0.6895 - acc: 0.5650
Epoch 7/10
400/400 [==============================] - 0s - loss: 0.6814 - acc: 0.6100
Epoch 8/10
400/400 [==============================] - 0s - loss: 0.6756 - acc: 0.6100
Epoch 9/10
400/400 [==============================] - 0s - loss: 0.6720 - acc: 0.6100
Epoch 10/10
400/400 [==============================] - 0s - loss: 0.6694 - acc: 0.6100
```

- Use keras fit function
  - Specify number of epoch & batch size
- Prints progress after each epoch
  - Loss = loss on training data
  - Acc = accuracy on training data

Fitting the Model with Many Epochs

- This example requires large number of epochs (~1000)
- Do not want to print progress on each epoch
- Rewrite code to manually print progress

```
nit = 20   # number of training iterations
nepoch_per_it = 50  # number of epochs per iterations

# Loss, accuracy and epoch per iteration
loss = np.zeros(nit)
acc = np.zeros(nit)
epoch_it = np.zeros(nit)

# Main iteration loop
for it in range(nit):

    # Continue the fit of the model
    init_epoch = it*nepoch_per_it
    model.fit(X, y, epochs=nepoch_per_it, batch_size=100, verbose=0)

    # Measure the loss and accuracy on the training data
    lossi, acci = model.evaluate(X,y, verbose=0)
    epochi = (it+1)*nepoch_per_it
    epoch_it[it] = epochi
    loss[it] = lossi
    acc[it] = acci
    print("epoch=%4d loss=%12.4e acc=%7.5f" % (epochi,lossi,acci))
```

```
epoch=  50 loss=  6.6854e-01 acc=0.61000
epoch= 100 loss=  6.6702e-01 acc=0.61000
epoch= 150 loss=  6.5264e-01 acc=0.61000
epoch= 200 loss=  5.9691e-01 acc=0.53500
epoch= 250 loss=  5.4305e-01 acc=0.70500
epoch= 300 loss=  4.8620e-01 acc=0.79000
epoch= 350 loss=  4.1364e-01 acc=0.86250
epoch= 400 loss=  3.6114e-01 acc=0.86250
epoch= 450 loss=  3.3093e-01 acc=0.86750
epoch= 500 loss=  3.1383e-01 acc=0.86750
epoch= 550 loss=  3.0321e-01 acc=0.87250
epoch= 600 loss=  2.9631e-01 acc=0.88000
epoch= 650 loss=  2.9159e-01 acc=0.87750
epoch= 700 loss=  2.8804e-01 acc=0.88250
epoch= 750 loss=  2.8534e-01 acc=0.88750
epoch= 800 loss=  2.8322e-01 acc=0.88250
epoch= 850 loss=  2.8132e-01 acc=0.88750
epoch= 900 loss=  2.7995e-01 acc=0.89000
epoch= 950 loss=  2.7846e-01 acc=0.88500
epoch=1000 loss=  2.7721e-01 acc=0.89000
```

Recap: MNIST data

- Classic MNIST problem:
  - Detect hand-written digits
  - Each image is 28 x 28 = 784 pixels
- Dataset size:
  - 50,000 training digits
  - 10,000 test
  - 10,000 validation (not used here)
- Can be loaded with sklearn and many other packages

Simple MNIST Neural Network

- 784 inputs, 100 hidden units, 10 outputs

```
nin = X.shape[1]  # dimension of input data
nh = 100     # number of hidden units
nout = int(np.max(y)+1)    # number of outputs = 10 since there are 10 classes
model = Sequential()
model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(nout, activation='softmax', name='output'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 100)               78500
_________________________________________________________________
output (Dense)               (None, 10)                1010
=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
```

Fitting the Model

- Run for 20 epochs, ADAM optimizer, batch size = 100
- Final accuracy = 0.972
- Not great, but much faster than SVM. Also CNNs we study later do even better.

```
opt = optimizers.Adam(lr=0.001) # beta_1=0.9, beta_2=0.
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
model.fit(Xtr, ytr, epochs=10, batch_size=100, validation_data=(Xts,yts))
```

```
Epoch 7/10
50000/50000 [==============================] - 3s - loss: 0.0474 - acc: 0.9868 - val_loss: 0.0886 - val_ac
c: 0.9717
Epoch 8/10
50000/50000 [==============================] - 3s - loss: 0.0440 - acc: 0.9884 - val_loss: 0.0875 - val_ac
c: 0.9718
Epoch 9/10
50000/50000 [==============================] - 2s - loss: 0.0393 - acc: 0.9903 - val_loss: 0.0872 - val_ac
c: 0.9732
Epoch 10/10
50000/50000 [==============================] - 3s - loss: 0.0381 - acc: 0.9901 - val_loss: 0.0875 - val_ac
c: 0.9718
```