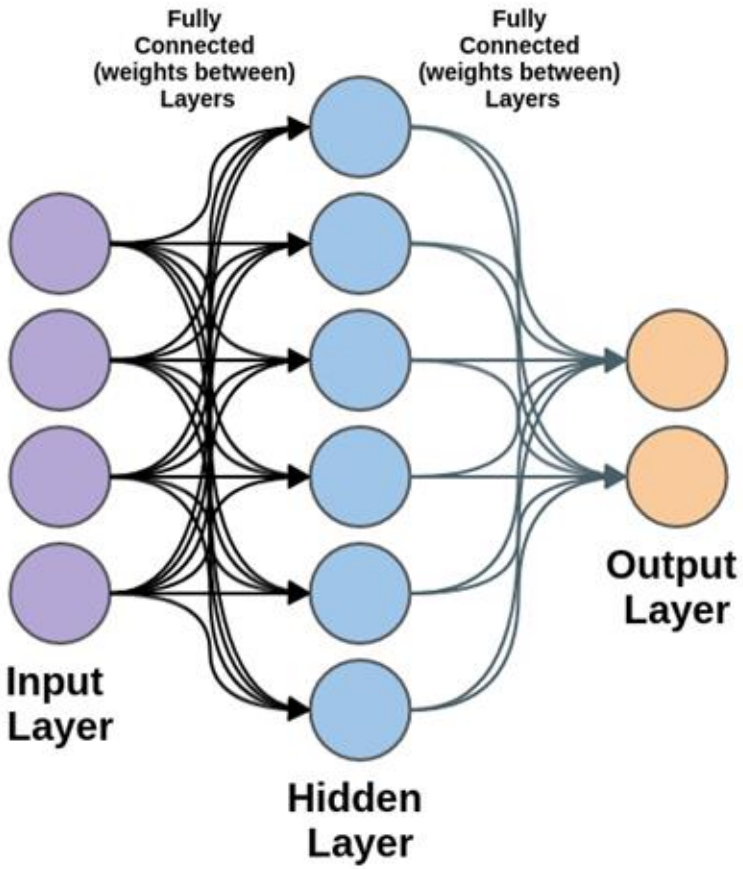
Multi-Class Classification

Solution

- Traditional Method: 1-vs-other method
  - Too slow. If we have n-classes, we need to train n models
  - Performance is not great, because the sample size is different for positive and negative classes
- Multiple Neurons
  - Use n output neuron to correspond n classes.
  - Easy, fast, and robust
  - Problem: how to model the probability? The values in the neural network can be negative or greater than 1.



Softmax: normalized exponential

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Input: vector of reals

Output: **probability** distribution

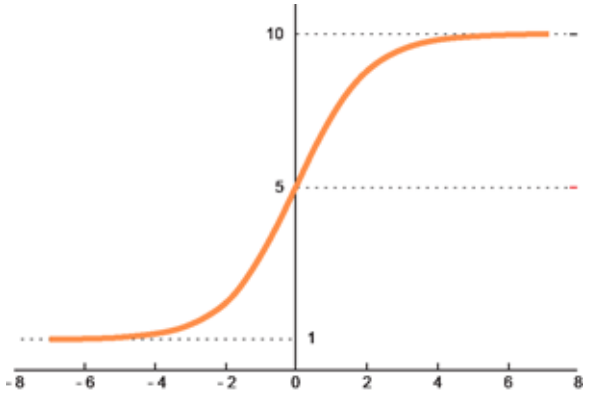
softmax([1,2,7,3,2]):

Calculate e^x: [2.72, 7.39, 1096.63, 20.09, 7.39]

Calculate sum(e^x): 2.72+7.39+1096.63+20.09+7.39 = 1134.22

Normalize: e^x/sum(e^x) = [0.002, 0.007, 0.967, 0.017, 0.007]

Result is a vector of reals.

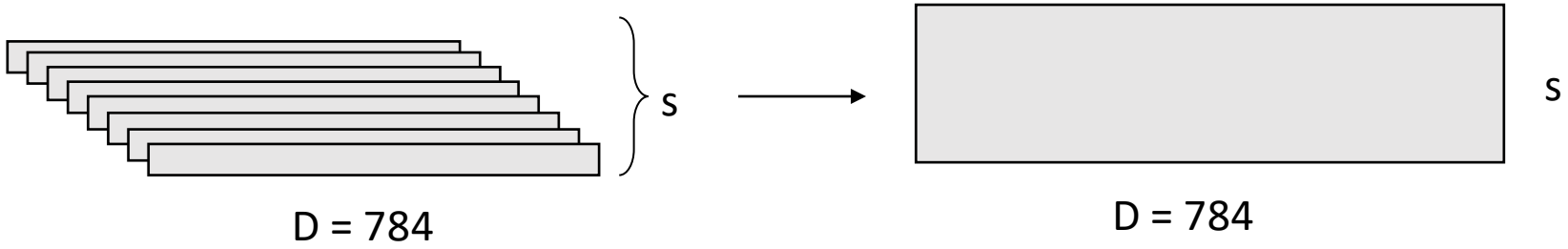


A Simple Example

Here, we will go over a simple 2-layer neural network (no bias).

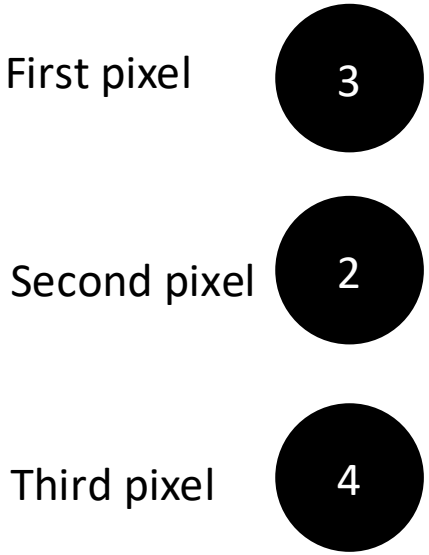
Mini-batch for Machine Learning

- Let's say we have S images of size 28x28
- We use a matrix to represent data.



- When s is very large RAM overload
- In HW4, we will use a batch size of 128

Neural Network Easy Example



First Batch

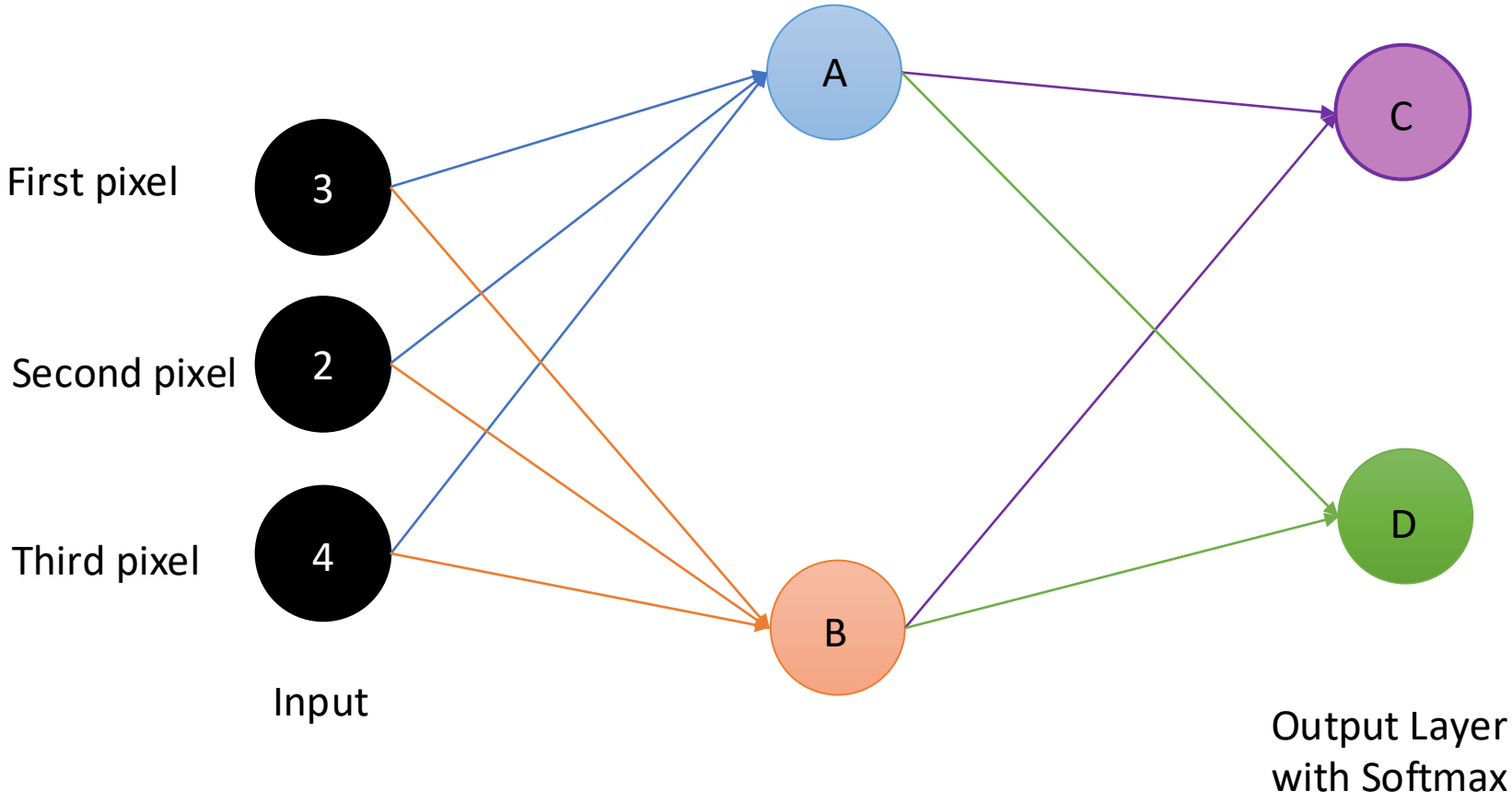
Input

X_{in}

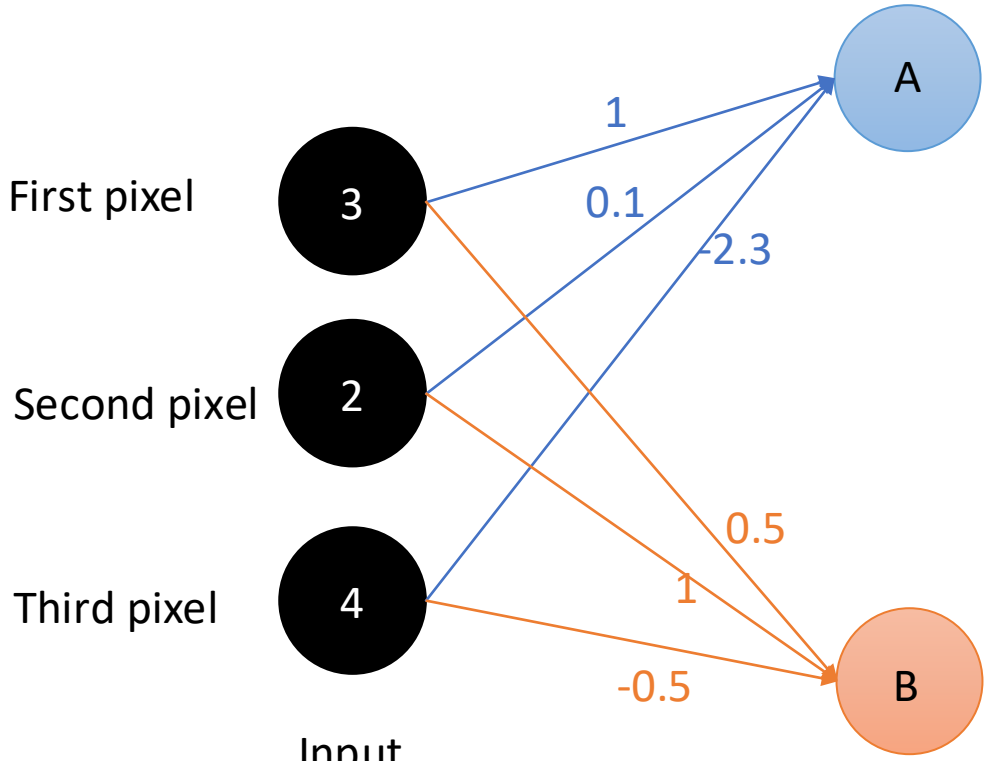
3	2	4
.	.	.
.	.	.

Bsz = 3

Neural Network Easy Example



Neural Network Easy Example



Input

X_{in}

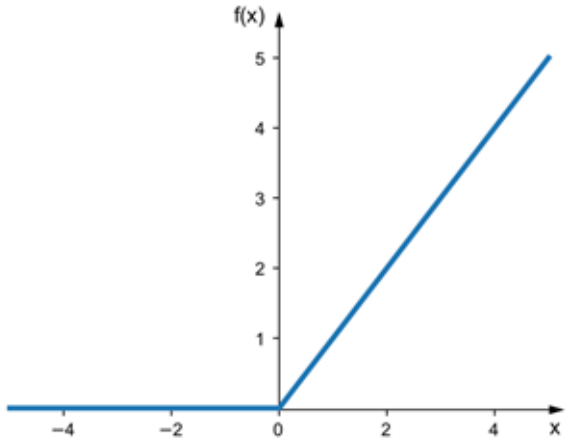
3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

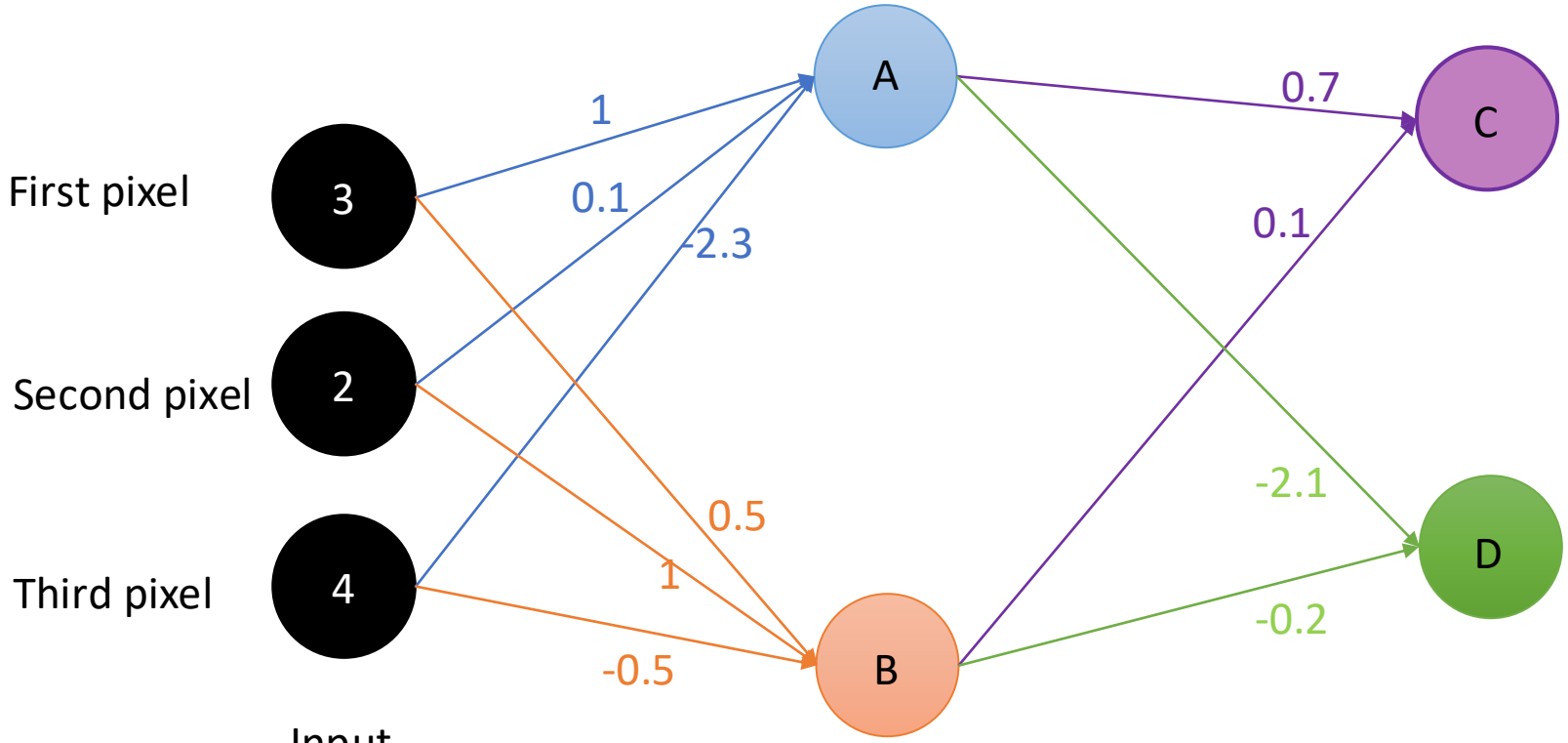
w_1

1	0.5
0.1	1
-2.3	-0.5
A	B

Weights for level 1



Neural Network Easy Example



Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

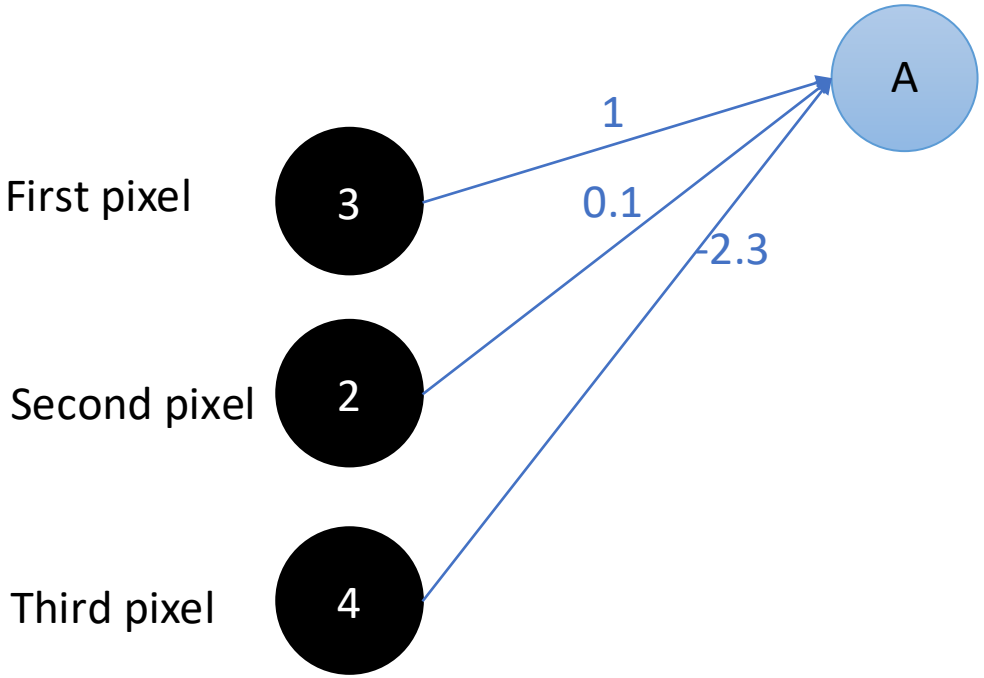
1	0.5
0.1	1
-2.3	-0.5

Output Layer with Softmax

w_2

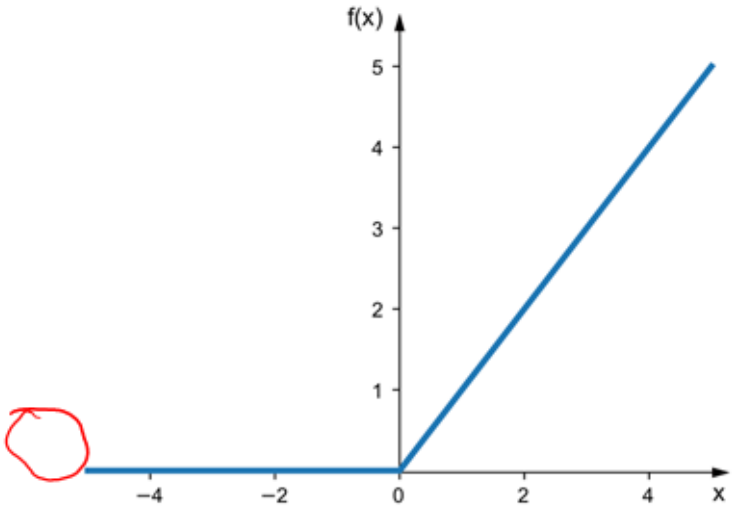
0.7	-2.1
0.1	-0.2

1st Layer with ReLU



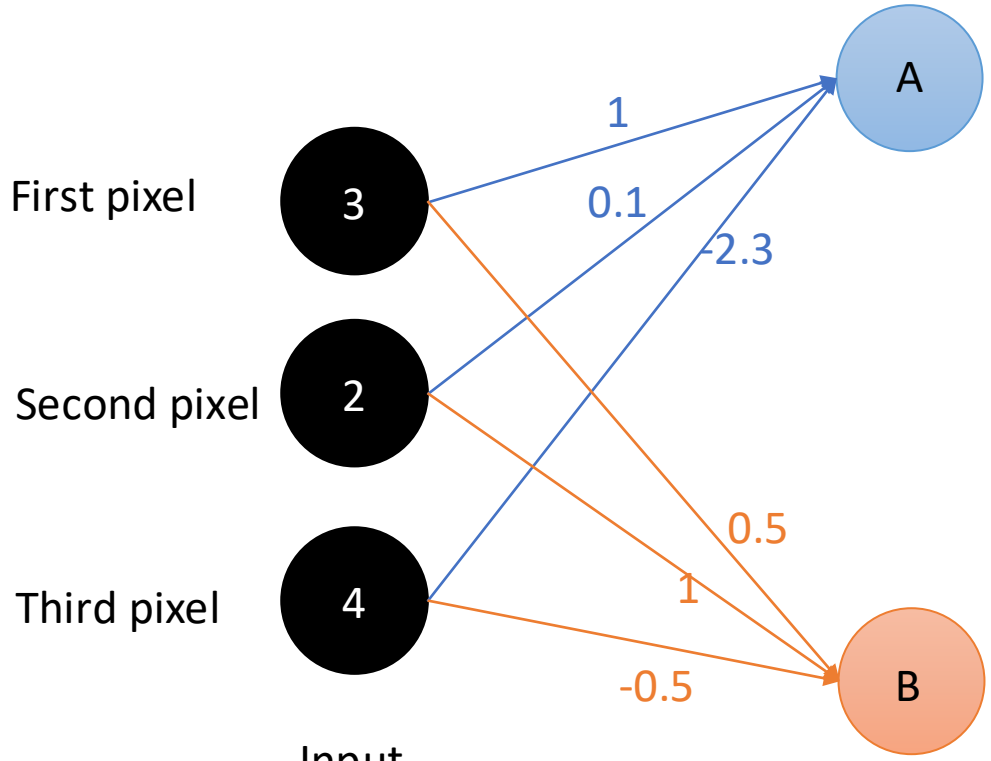
$$Output_A = 3 * 1 + 2 * 0.1 + 4 * -2.3 = -6.0$$

$$Output_A = 0 \ (after \ RELU)$$



RELU

Neural Network Easy Example



$A = 0$

$B = 3 * 0.5 + 2 * 1 + 4 * -0.5 = 1.5$

Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

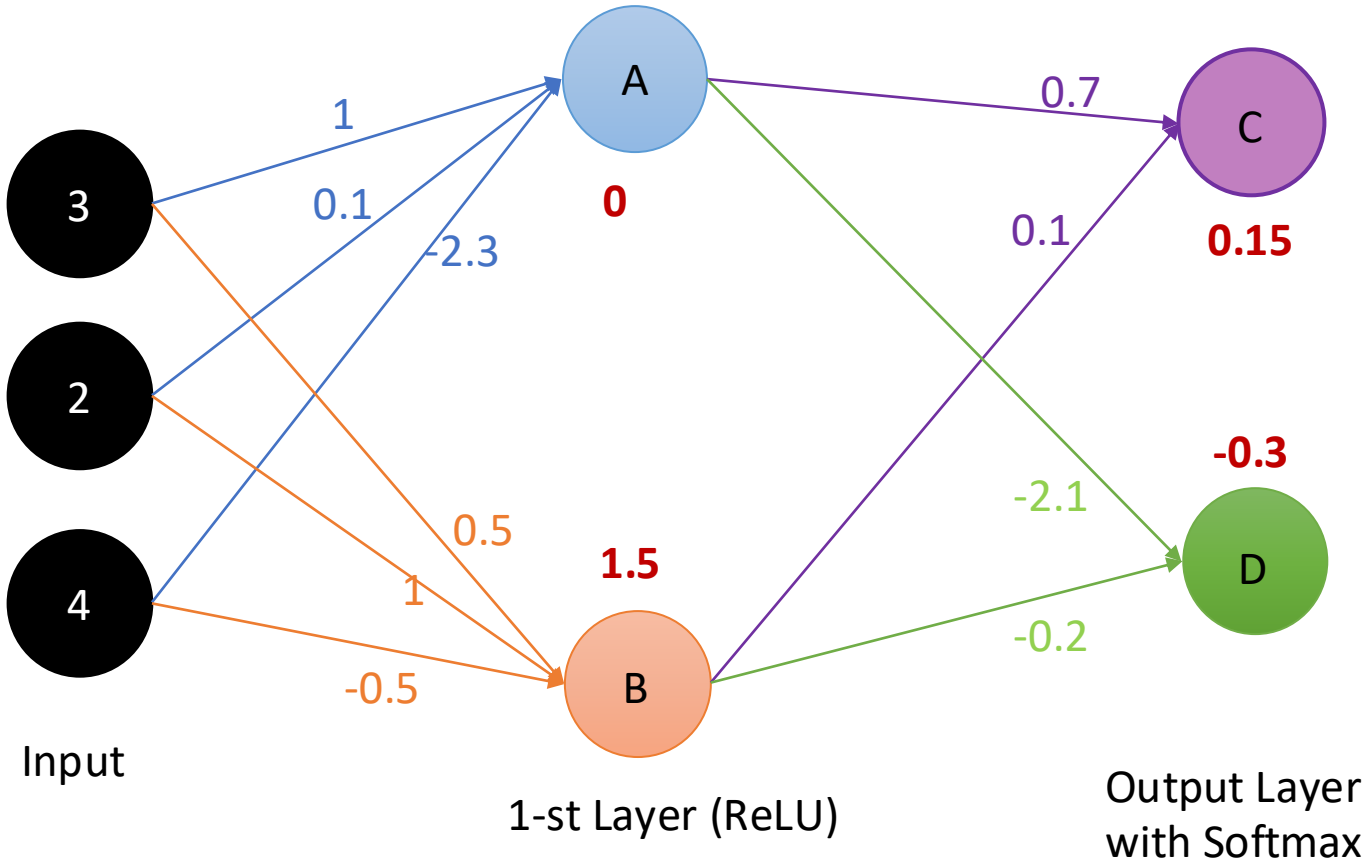
1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

O_1 is outputs from layer 1

Neural Network Easy Example

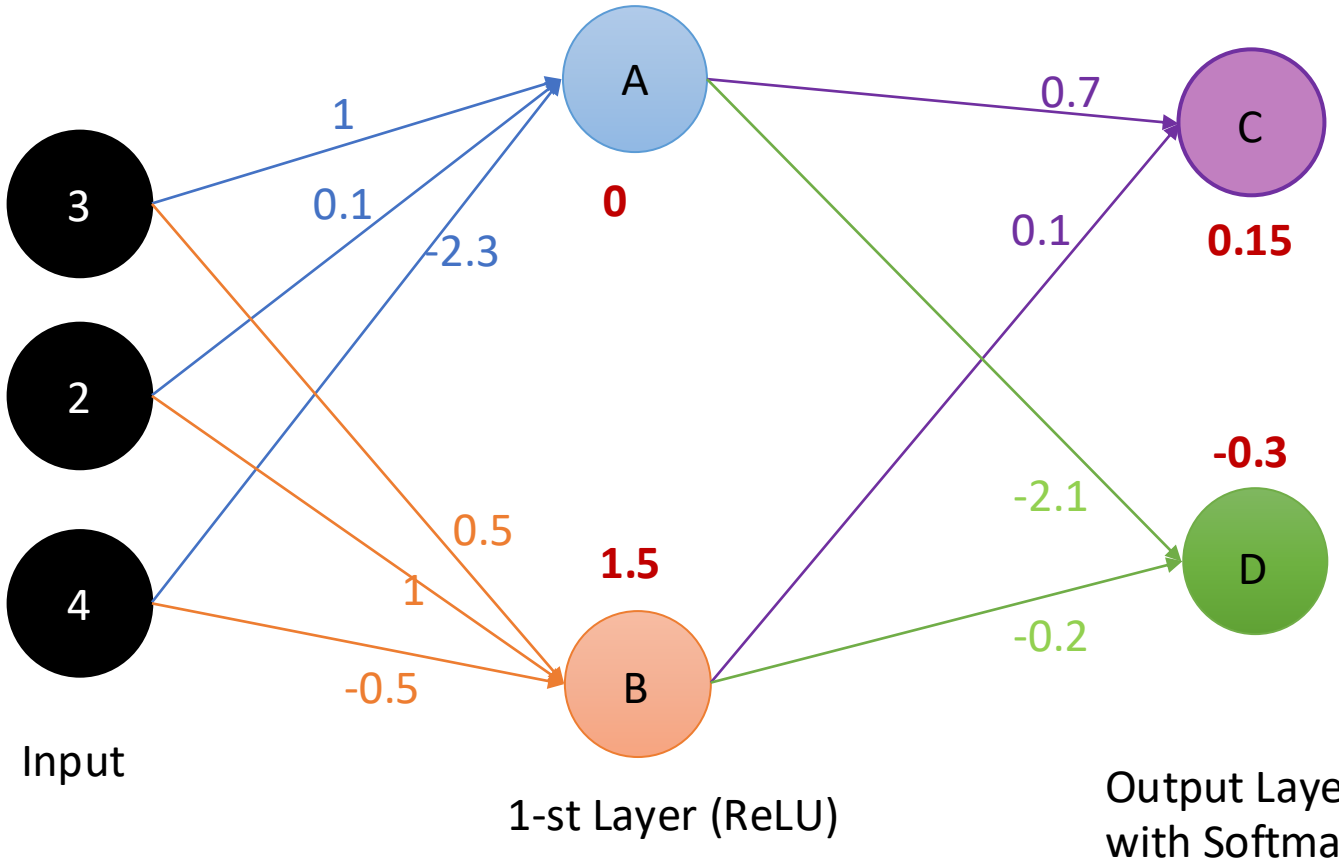


$$output_C = \frac{e^{0.15}}{e^{0.15} + e^{-0.3}} \approx \frac{1.16}{1.16 + 0.74} = 0.61$$

Softmax function applied to output

$$output_D = \frac{e^{-0.3}}{e^{0.15} + e^{-0.3}} \approx \frac{0.74}{1.16 + 0.74} = 0.39$$

Neural Network Easy Example



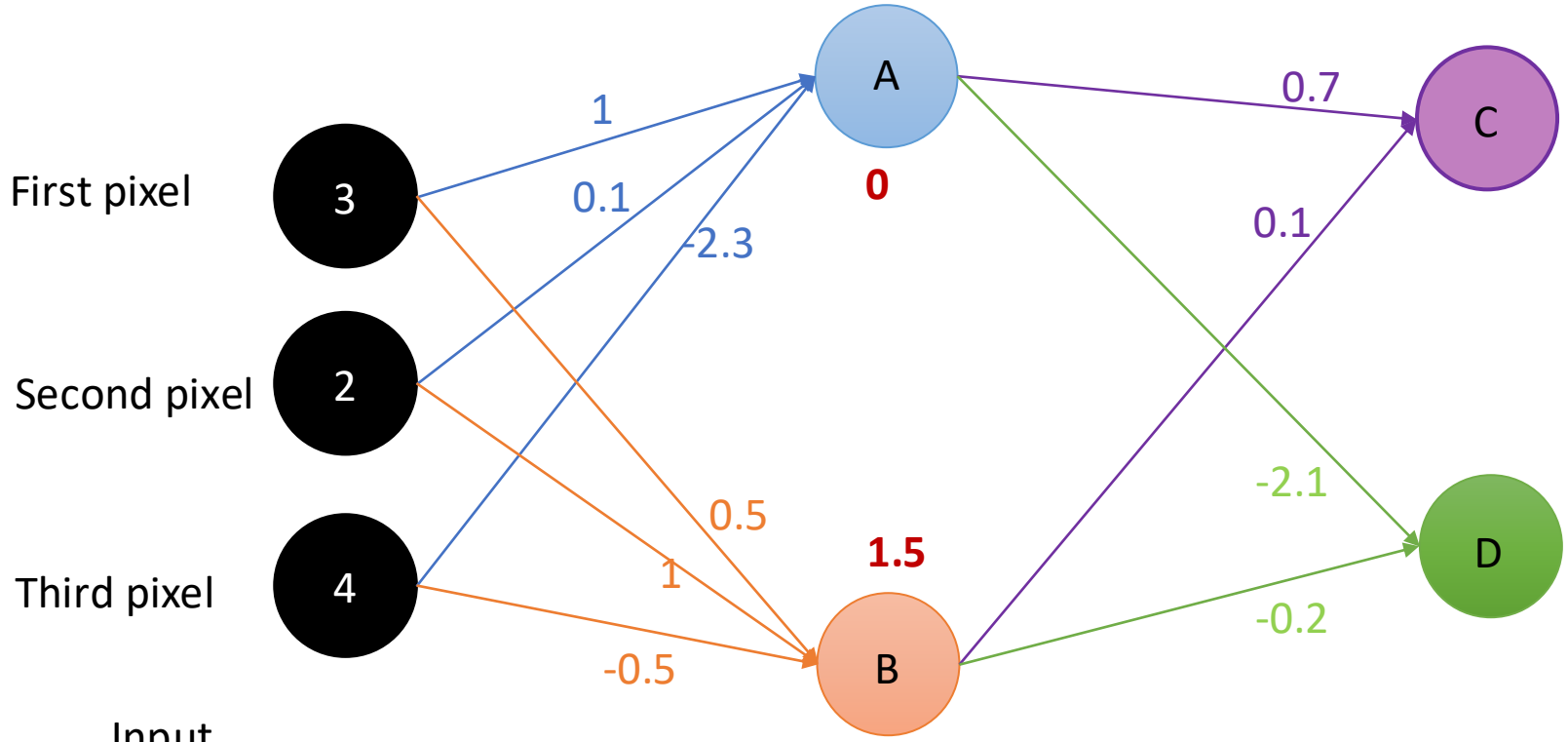
Output O_2 at layer 2

o_2

0.61	0.39
.	.
.	.

Neural Network Easy Example

Key:
w_i is weights at layer i
o_i is outputs at layer i



Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

Output Layer with Softmax

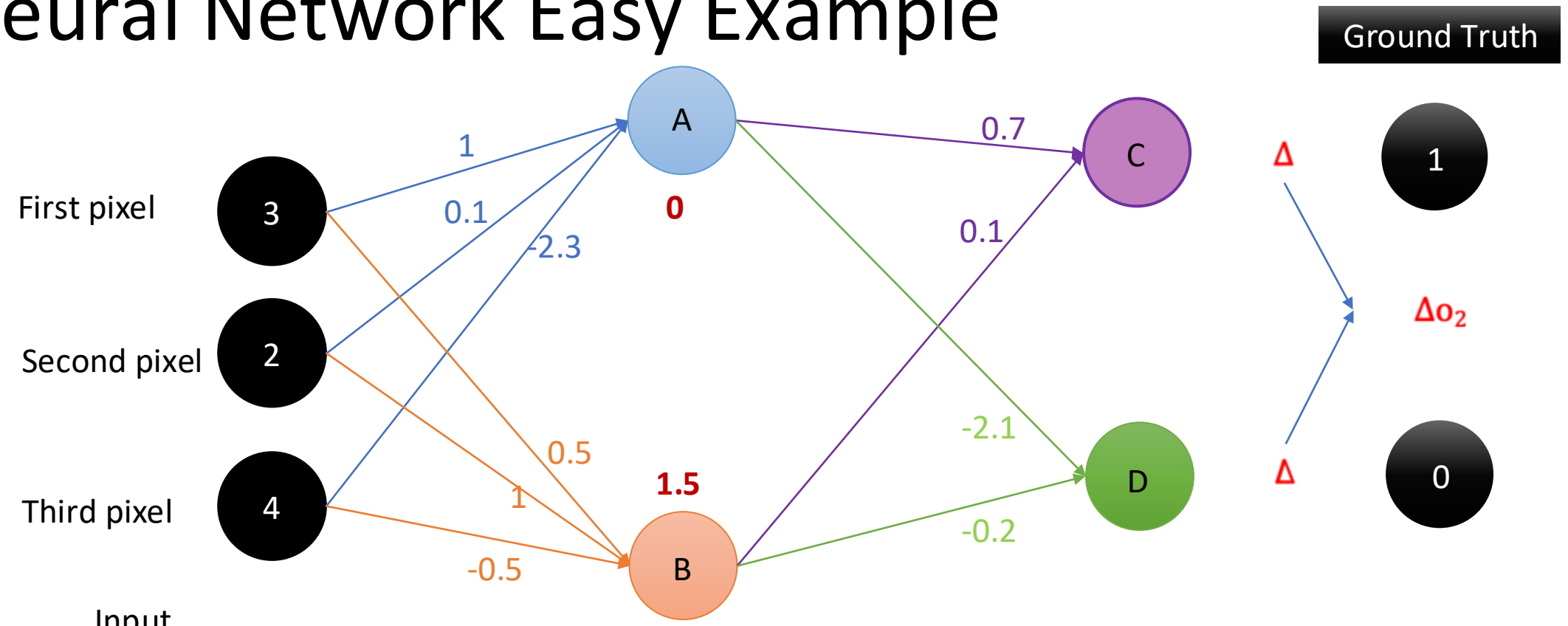
w_2

0.7	-2.1
0.1	-0.2

o_2

0.61	0.39
.	.
.	.

Neural Network Easy Example



Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

Output Layer with Softmax

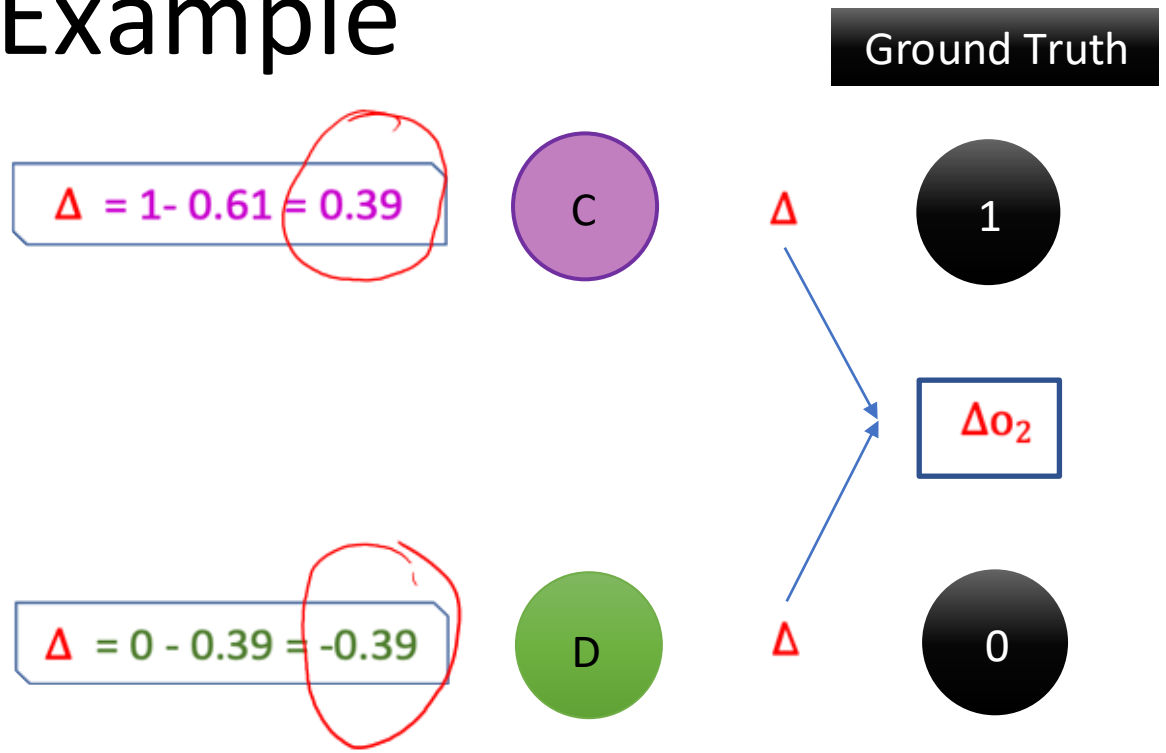
w_2

0.7	-2.1
0.1	-0.2

o_2

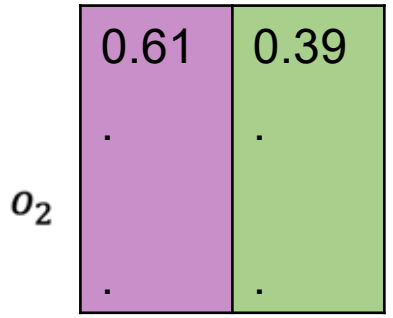
0.61	0.39
.	.
.	.

Neural Network Easy Example



This is the Error or Loss

Δo_2 = [.39 -.39]



What to do with this error?

What to do with this error?

Backpropagate it to update the weights!

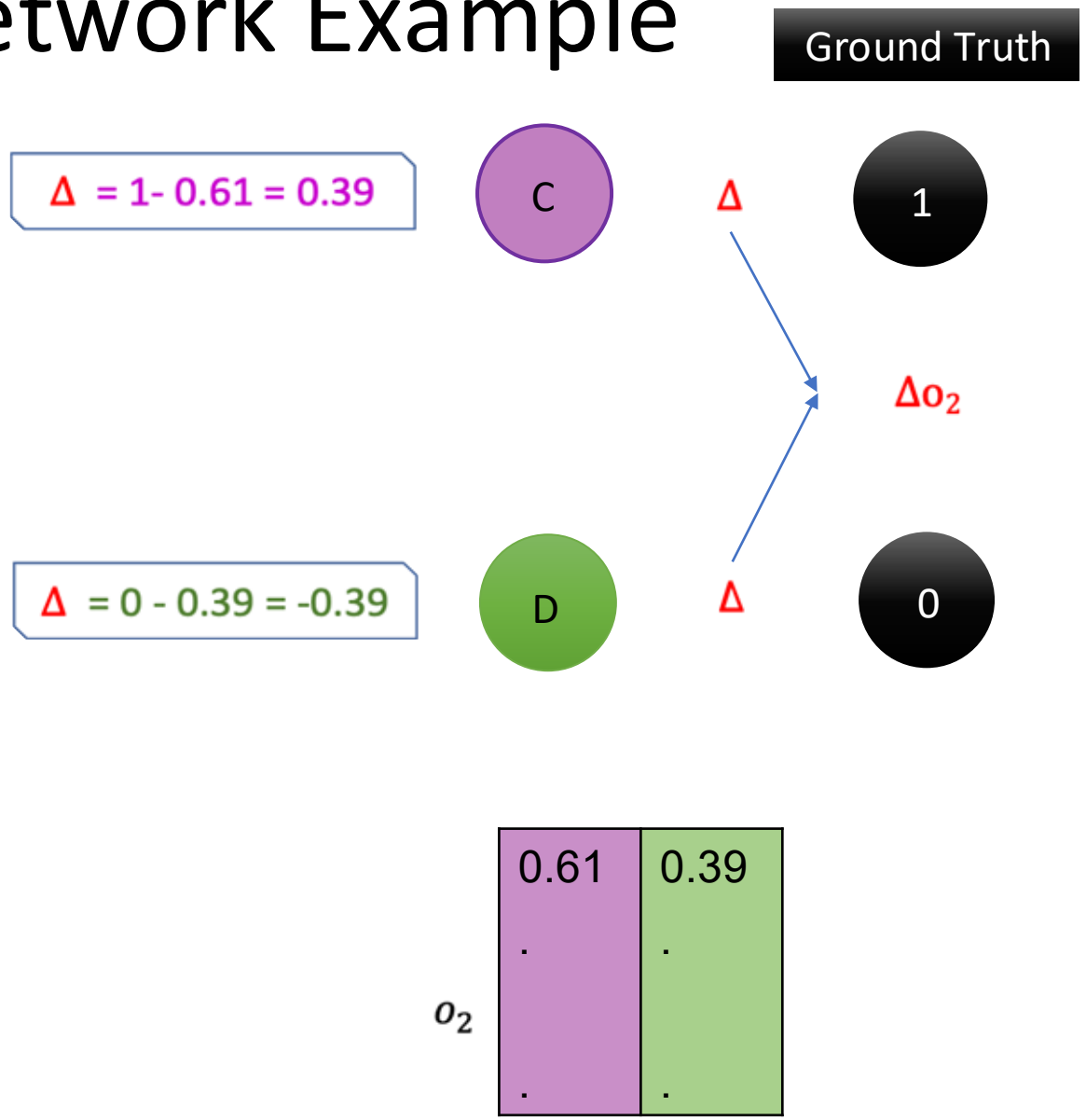
How? We will be doing this (in a few slides).

$\delta o1 = (W_2^T) \cdot \delta o2 \odot g'(z1)$

So what we'll do is:

- Take the error from the next layer (δo_2)
- Push it backward through the weights (W_2^T)
- Multiply by the activation derivative.

Back to our Neural Network Example



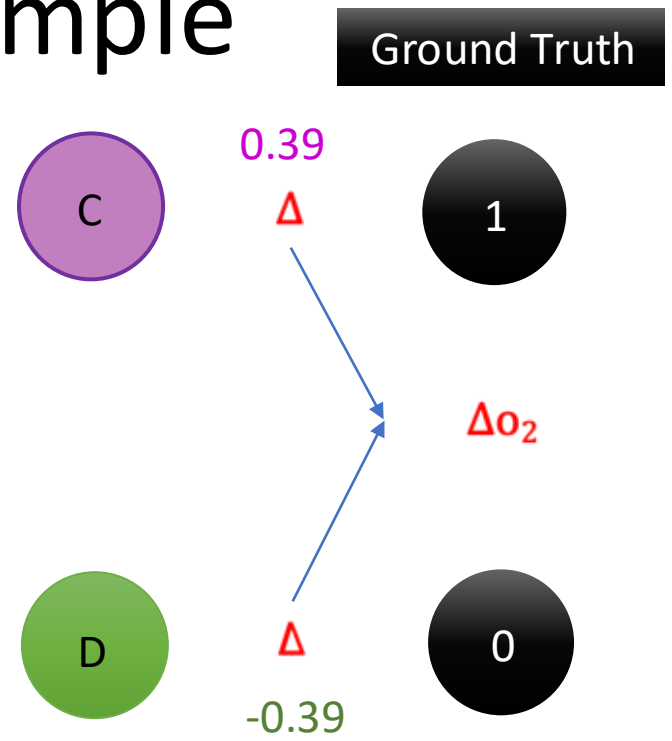
Back to our Neural Network Example

$$\Delta w_t = -\frac{\partial E}{\partial w_t} = o_1^T \, g'(o_2) \circ \Delta o_2$$
$$w_{t+1} = w_t + \alpha \, \Delta w_t$$

Assume g'(.) = 1

"∘" represents elementwise multiplication for matrix

We need to compute this for t=1 and t-2



o_2

0.61	0.39
.	.
.	.

Neural Network Easy Example

$g'(o_2) \circ \Delta o_2$

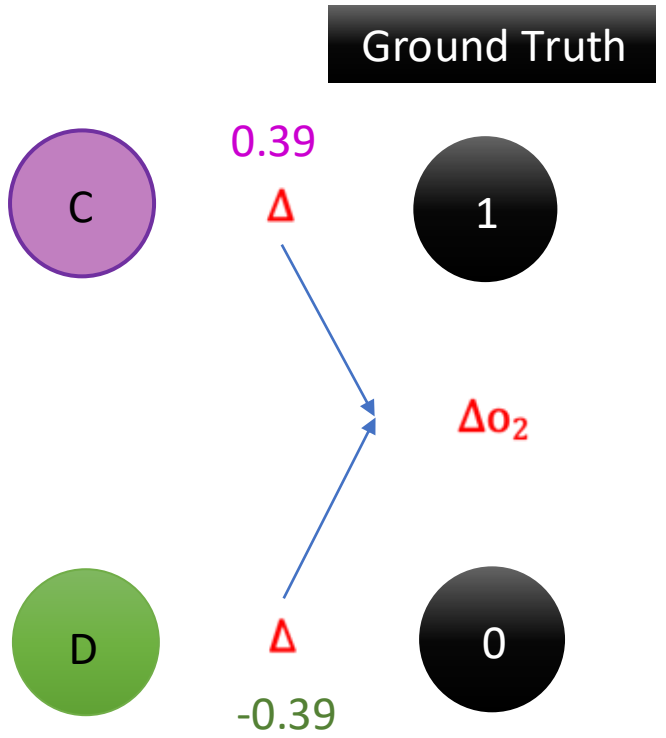
0.39	-0.39
.	.
.	.

$\Delta w_2 = o_1^T g'(o_2) \circ \Delta o_2$

0	0
0.585	-0.585

$\Delta o_1 = g'(o_2) \circ \Delta o_2 w_2^T$

1.092	0.117
.	.
.	.



"∘" represents elementwise multiplication for matrix

Assume g'(.) = 1

$$\Delta w_t = -\frac{\partial E}{\partial w_t} = o_1^T g'(o_2) \circ \Delta o_2$$

$$w_{t+1} = w_t + \alpha \Delta w_t$$

Recall that o_1 was [0 1.5] and note that 1.5 * .39 = .585.

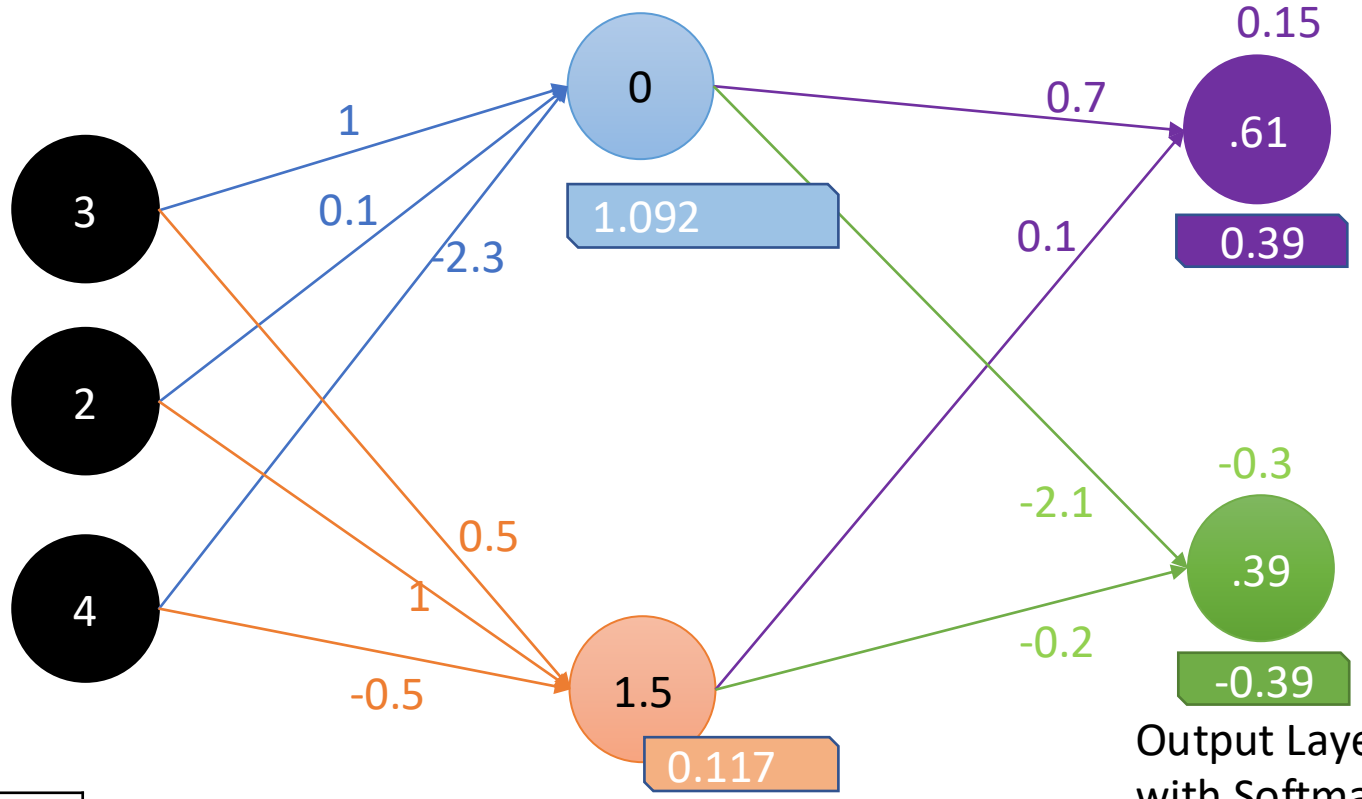
Now to get Δo_1 multiply Δo_2 by W_2^T.

W_2^T =
0.7 −0.1
−0.21 −0.2]

o_2

0.61	0.39
.	.
.	.

Backpropagation [Cont.]



Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

w_2

0.7	-2.1
0.1	-0.2

o_2

0.61	0.39
.	.
.	.

$g'(o_2) \circ \Delta o_2$

0.39	-0.39
.	.
.	.

$\Delta w_2 = o_1^T g'(o_2) \circ \Delta o_2$

0	0
0.585	-0.585

$\Delta o_1 = g'(o_2) \circ \Delta o_2 w_2^T$

1.092	0.117
.	.
.	.

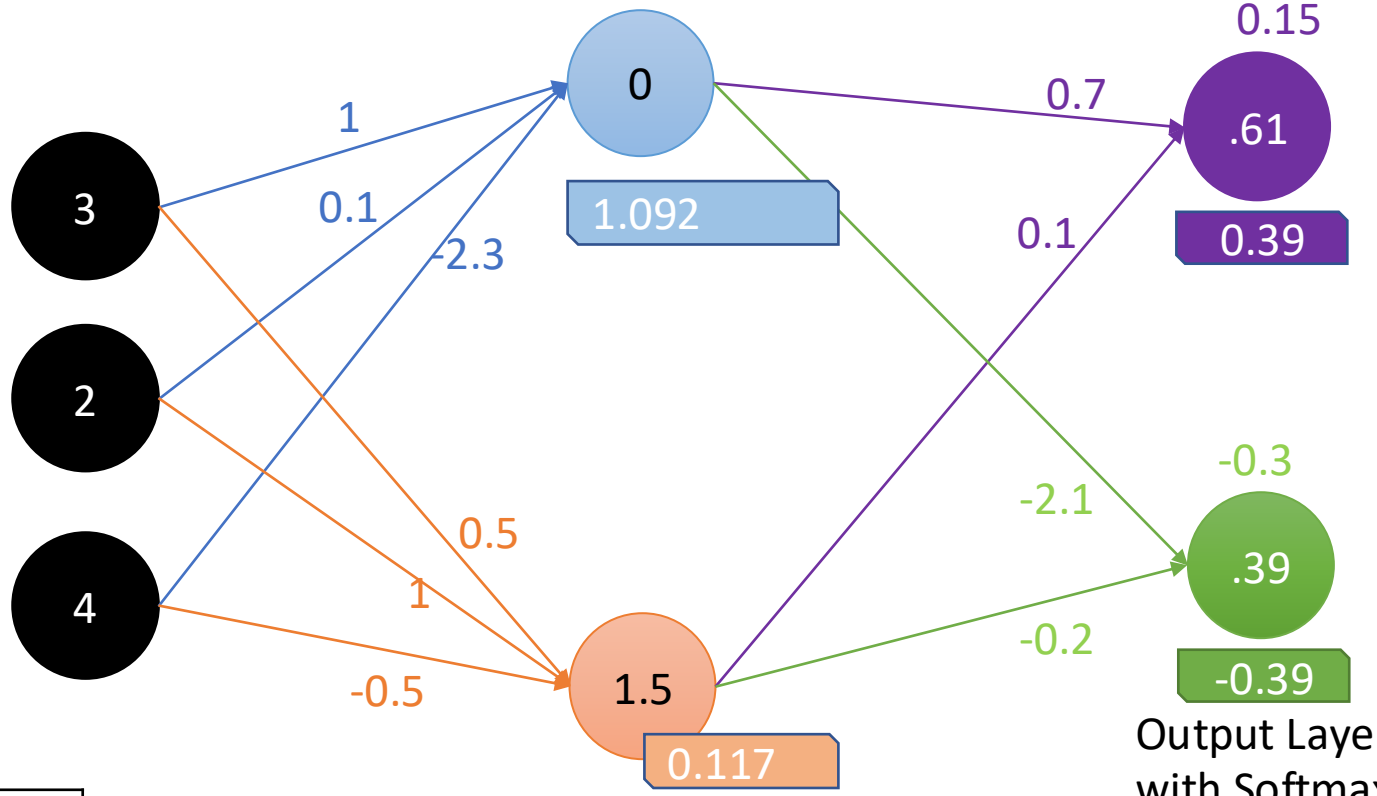
Backpropagation [Cont.]

$g'(o_1) \circ \Delta o_1$

0	0.117
.	.
.	.

$\Delta w_1 = o_0^T g'(o_1) \circ \Delta o_1$

0	0.351
0	0.234
0	0.468



$g'(o_2) \circ \Delta o_2$

0.39	-0.39
.	.
.	.

$\Delta w_2 = o_1^T g'(o_2) \circ \Delta o_2$

0	0
0.585	-0.585

$\Delta o_1 = g'(o_2) \circ \Delta o_2 w_2^T$

1.092	0.117
.	.
.	.

Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

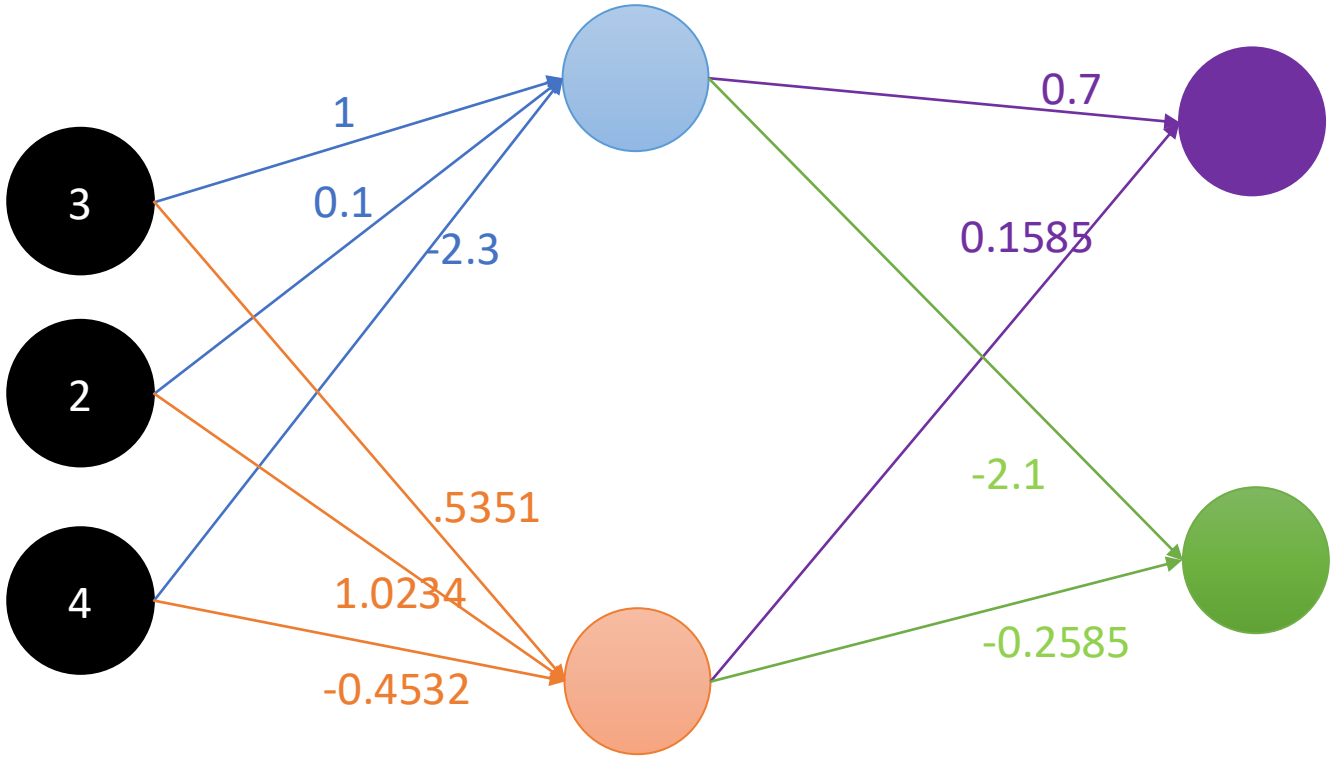
w_2

0.7	-2.1
0.1	-0.2

o_2

0.61	0.39
.	.
.	.

Now we backpropagate with a learning rate of 0.1



$$w_1 = w_1 + \alpha \Delta w_1$$

$$w_2 = w_2 + \alpha \Delta w_2$$

Input Layer

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

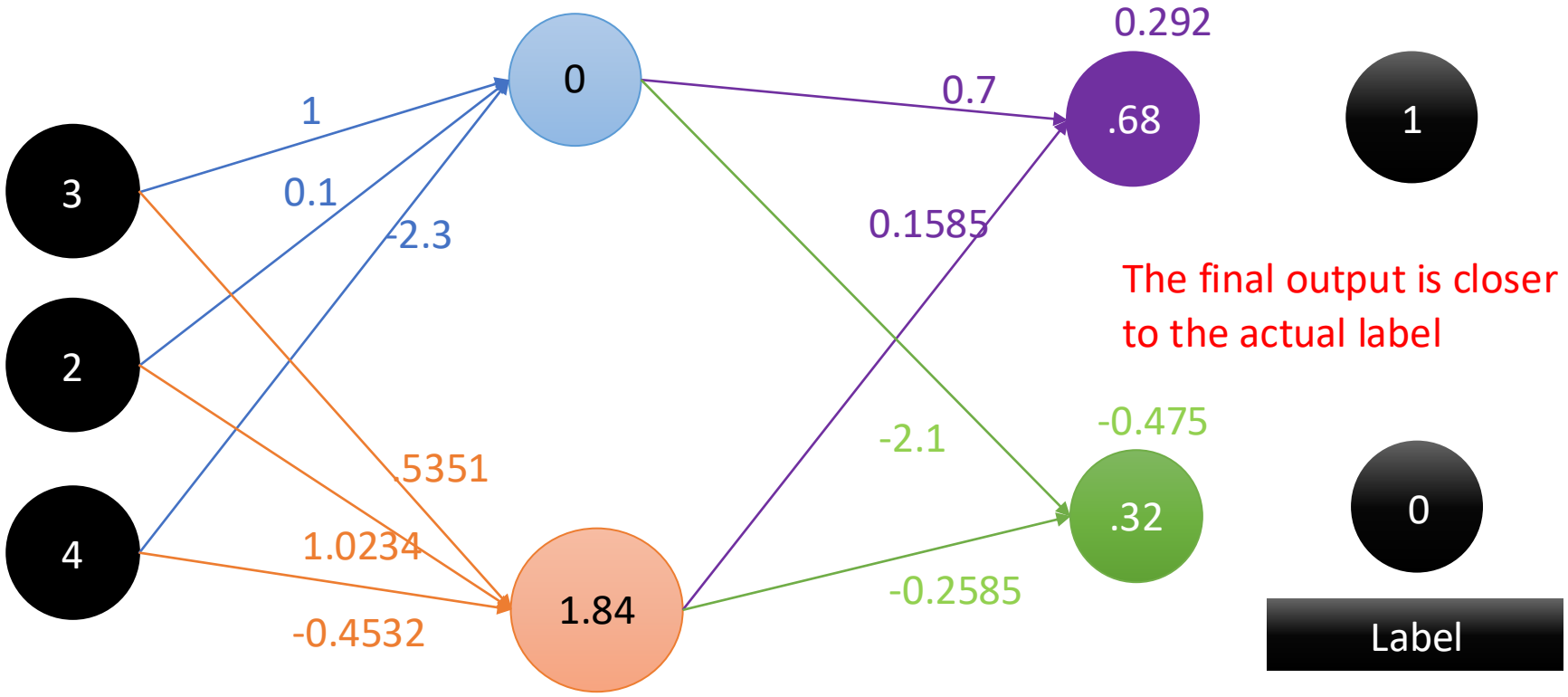
1	0.5351
0.1	1.0234
-2.3	-0.4532

Output Layer with Softmax

w_2

0.7	-2.1
0.1585	-0.2585

Think: What will happen if we go forward again?



Input Layer

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

1	0.5351
0.1	1.0234
-2.3	-0.4532

Output Layer with Softmax

w_2

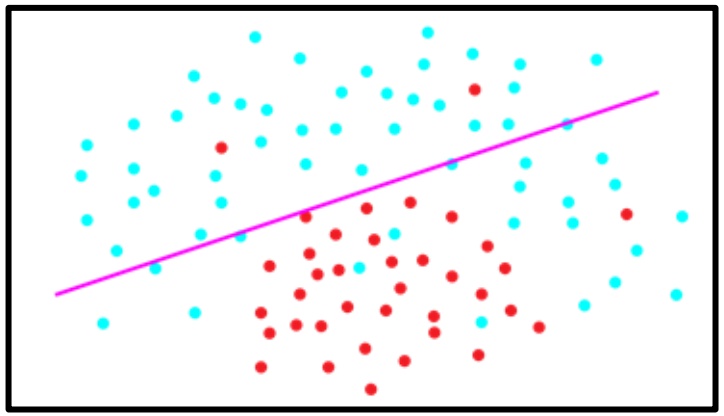
0.7	-2.1
0.1585	-0.2585

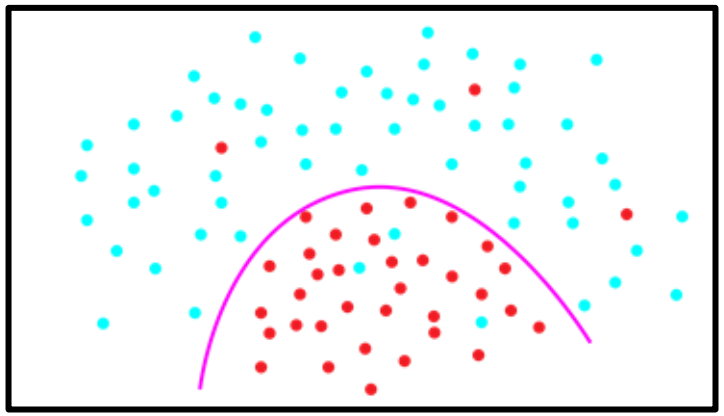
Tricks for Training Neural Networks

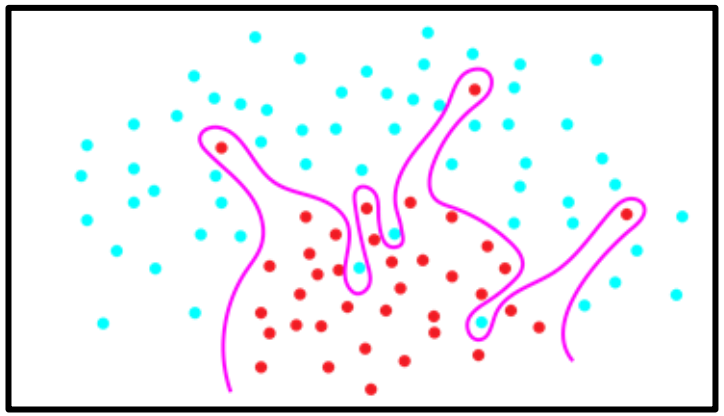
Problem: Under and Overfitting

Underfitting: model not powerful enough, too much bias

Overfitting: model too powerful, fits to noise, doesn't generalize well







Weight decay: neural network regularization

We want the weights to be close to 0.

We use Δw_t to represent the weight gradient for timepoint **t** (the current step).

Let L be the "loss" function; (e.g. $L = |y - g(in)|$, $L = (y - g(in))^2$, etc.)

λ is a regularization parameter (for decay)

Higher: more penalty for large weights, less powerful model

Lower: less penalty, more overfitting

Before:

$\Delta w_t = -\partial/\partial w_t \, L(w_t)$

$w_{t+1} = w_t + \alpha \, \Delta w_t$

Now:

$w_{t+1} = w_t - \alpha[\partial/\partial w_t \, L(w_t) + \lambda w_t] = w_t - \alpha[-\Delta w_t + \lambda w_t]$

$= w_t - \alpha \, \partial/\partial w_t \, L(w_t) - \alpha \, \lambda w_t = w_t + \alpha \, \Delta w_t - \alpha \, \lambda w_t$

Subtract a little bit of weight every iteration

Momentum: speeding up SGD

If we keep moving in same direction we should move further every round

Before:

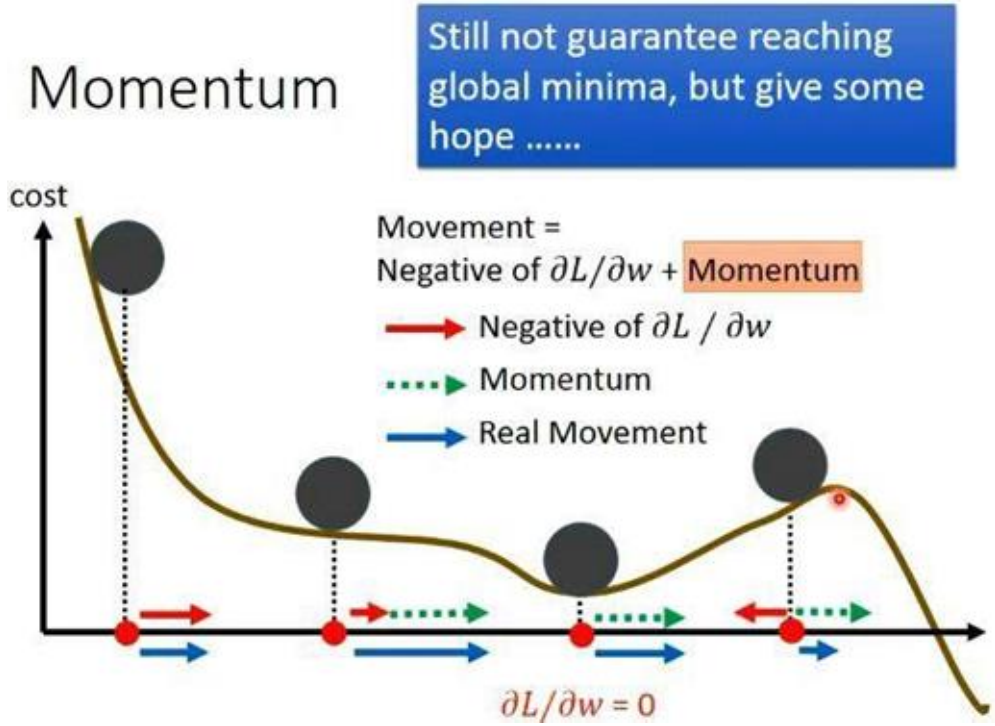
$$\Delta w_t = -\partial/\partial w_t \, L(w_t)$$

Now:

$$\Delta w_t = -\partial/\partial w_t \, L(w_t) + m\Delta w_{t-1}$$

$$w_{t+1} = w_t + \alpha \, \Delta w_t$$

Side effect: **smooths** out updates if gradient is in different directions



Δw_{t-1} represent the gradient calculated in the previous step.

NN updates with weight decay and momentum

$$\Delta w'_t = -\partial/\partial w_t L(w_t) - \lambda w_t + m\Delta w'_{t-1}$$

Gradient of loss

Weight decay

Momentum

$$w_{t+1} = w_t + \alpha \Delta w'_t$$

Learning rate

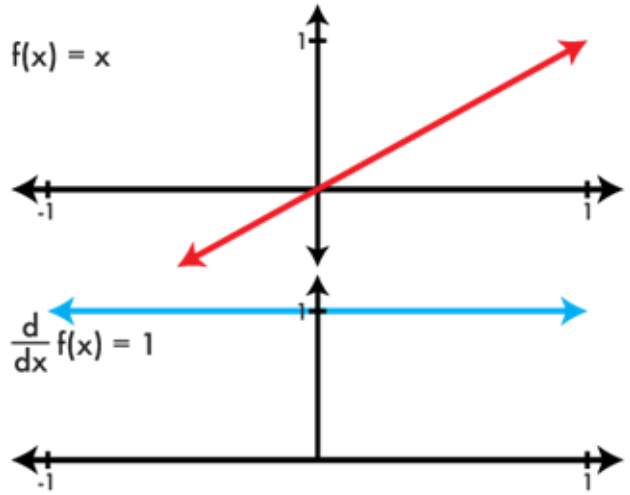
Dropout

Randomly eliminate some nodes during training

Activations

Linear Activation

- $$g(x) = x$$
  $$g'(x) = 1$$



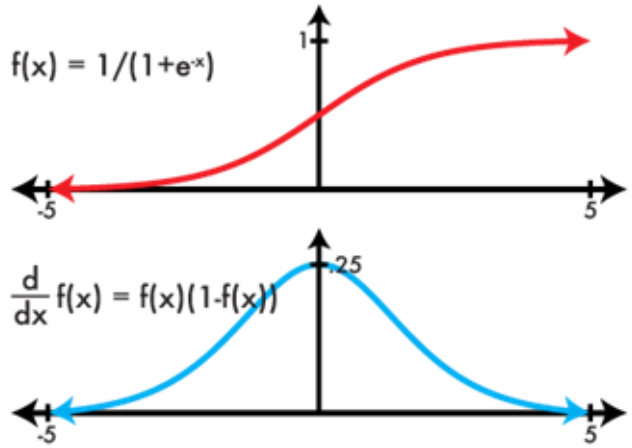
- Only offers linear effects.
- For a 2-layer NN with linear activations for both layers.

$$f(X) = g(g(Xw_1)w_2) = Xw_1w_2 = Xw$$

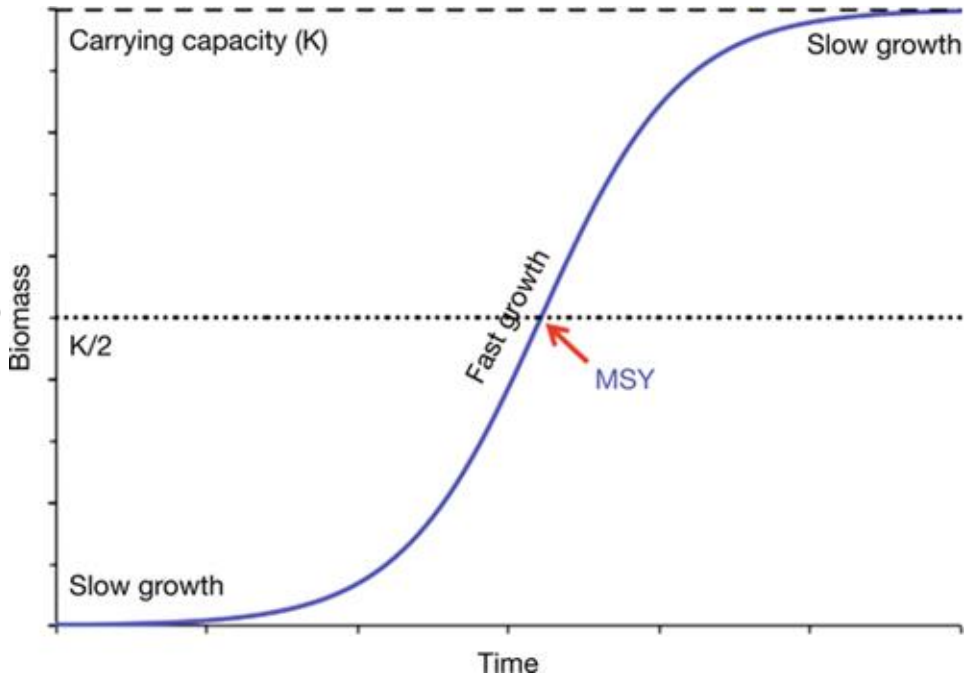
- Not so great, need Non-Linear activations to learn more complex data distribution.

Logistic Activation

- $$g(x) = \frac{1}{1 + e^{-x}}$$
$$g'(x) = g(x)g(1 - x)$$



- Aka Sigmoid function (S-shape)
- Used in Logistic regression.
- The result is in range (0, 1),
- It can represent probability.
- A special case of logistic growth (population model).



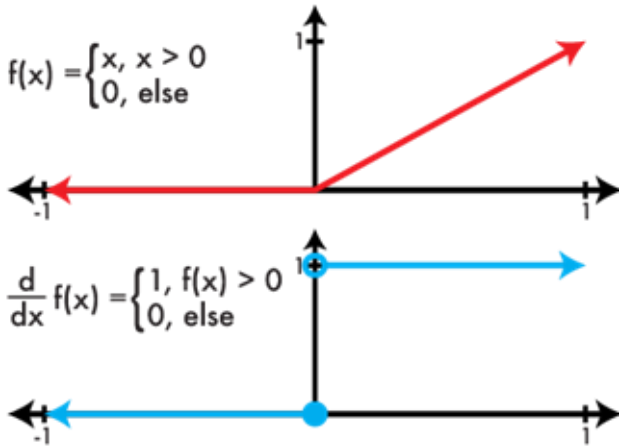
Do you see the problem with Sigmoidal activations?

Do you see the problem with Sigmoidal activations?

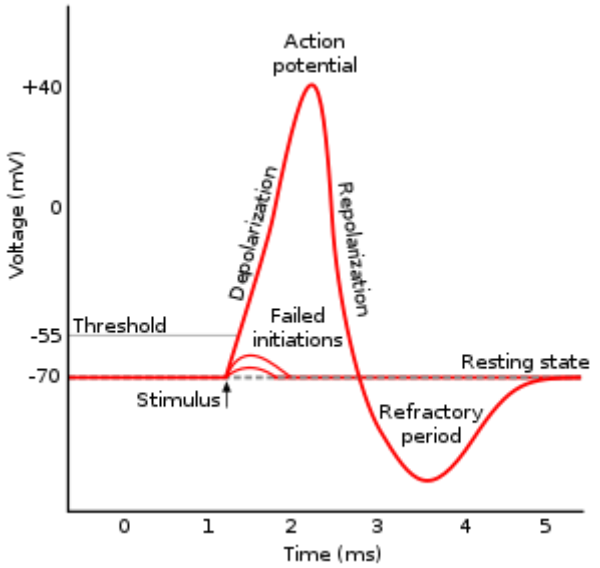
Vanishing gradients!

ReLU Activation

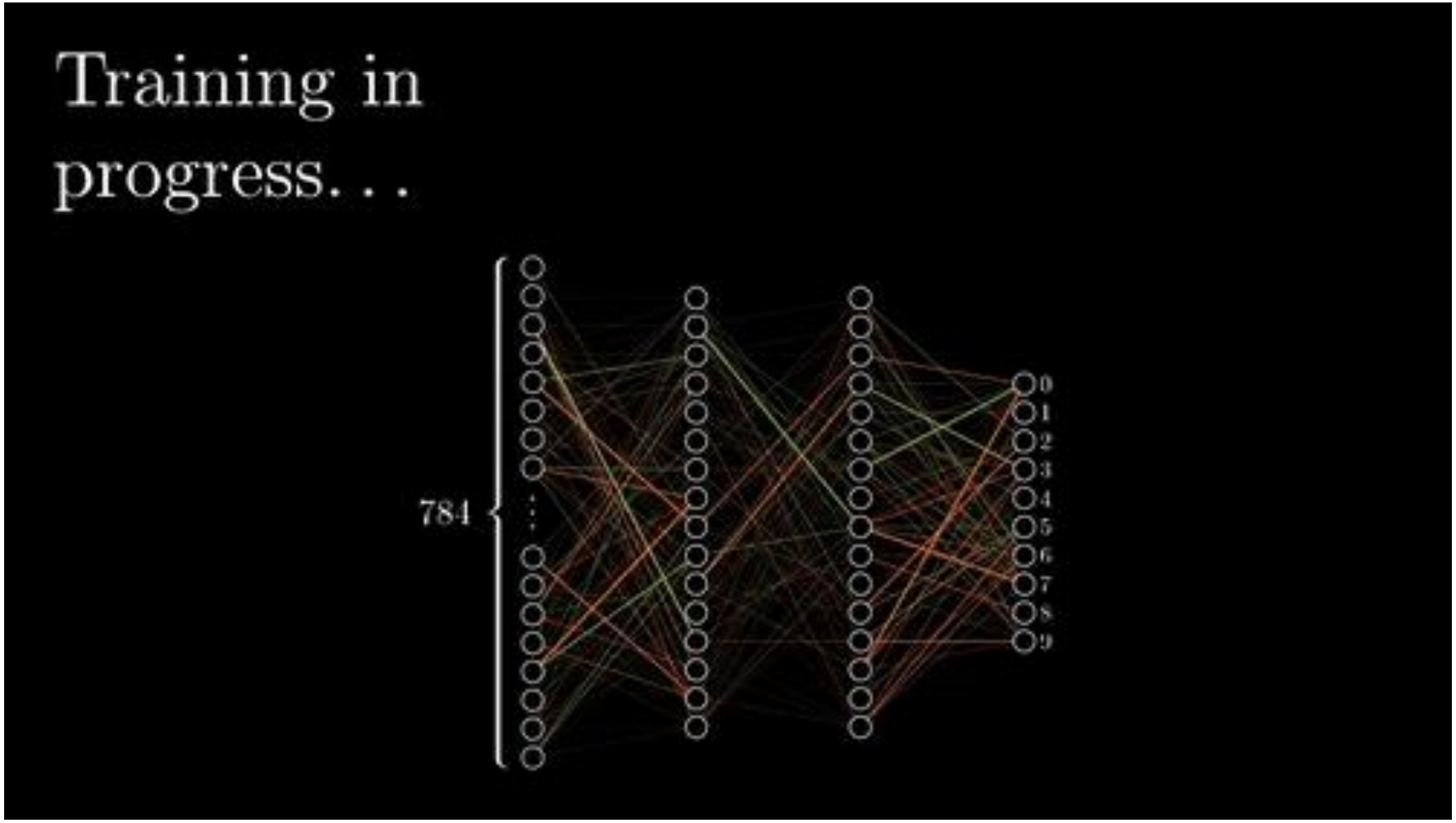
- $$g(x) = \max(0, x)$$
$$g'(x) = \mathbf{1}_{g(x)>0}$$



- Rectified linear unit
- ***Fast!*** In backpropagation, 1 when positive, 0 otherwise.
- Optimizes important (positive) values and ignore the others.
- Analog to neurons
- Information loss is small (other neurons will carry information)

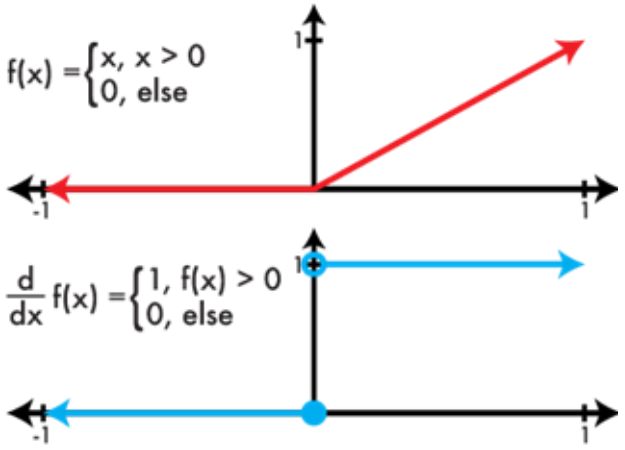


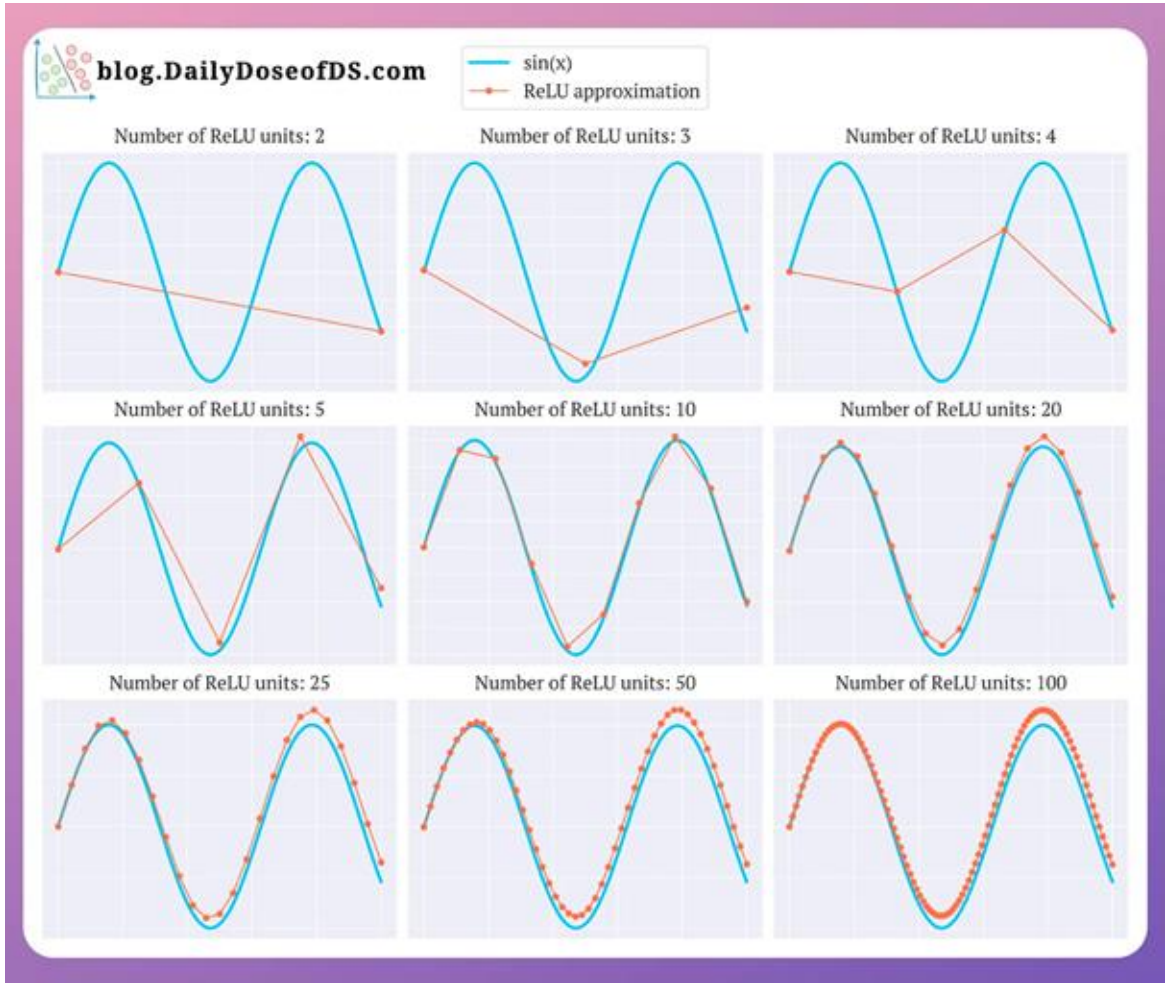
Visualization with ReLU



https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw

Why ReLU provides non-linearity?

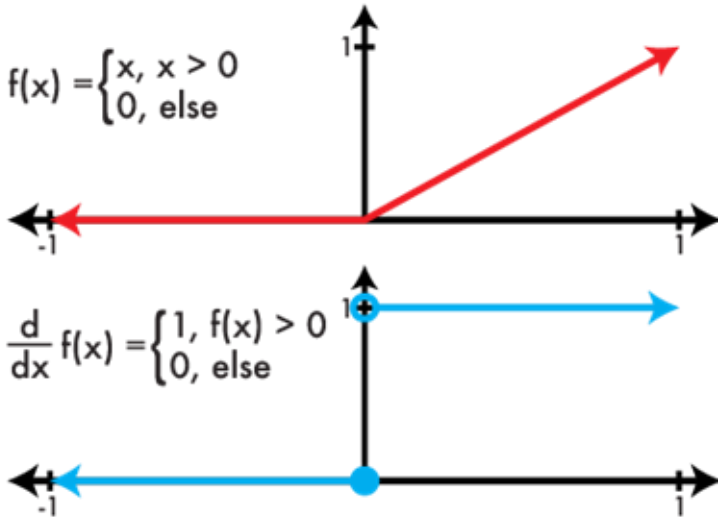




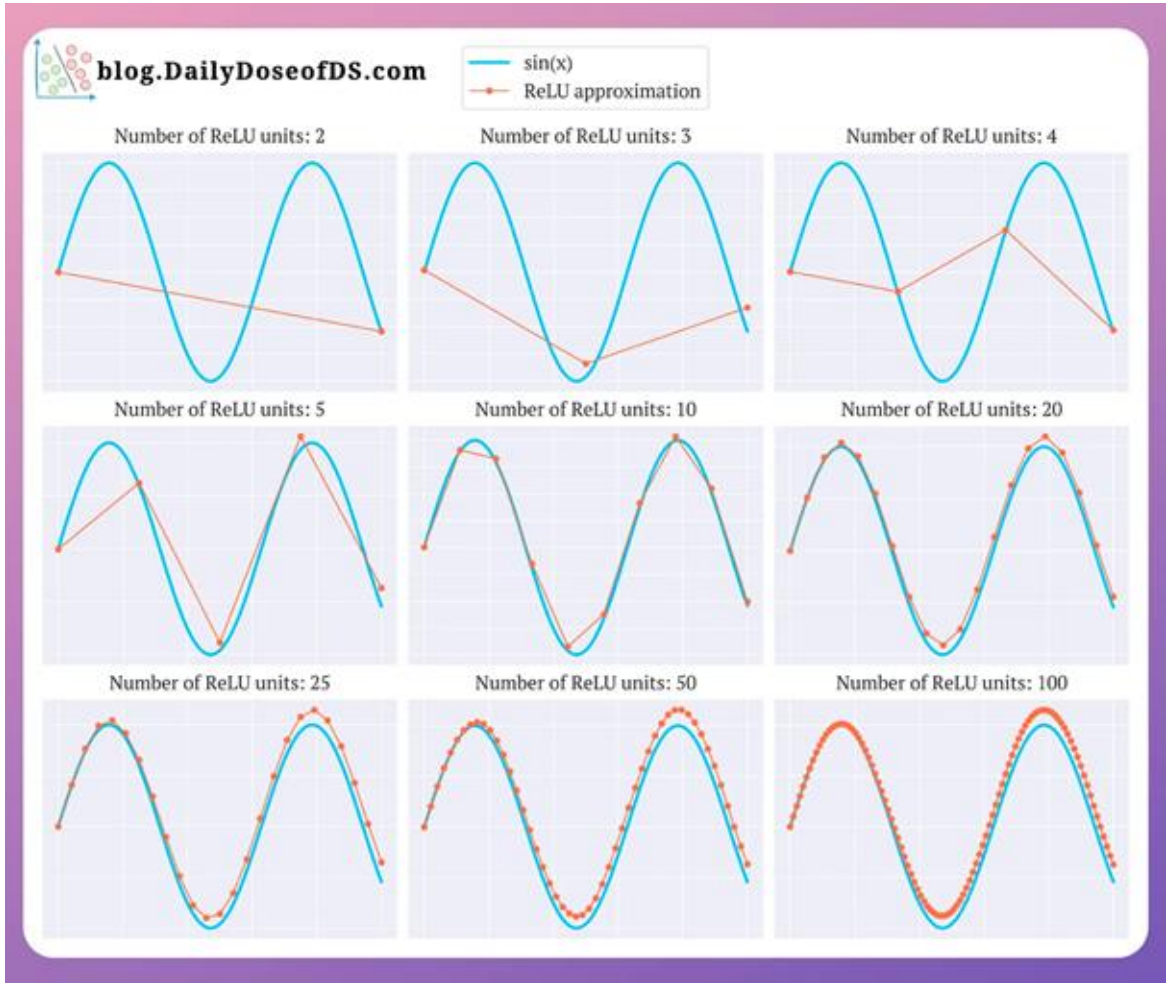
Why ReLU provides non-linearity?

The function is not linear if it does not satisfy the superposition principles: 1) additivity, 2) homogeneity

1) F (x1 + x2) = F(x1) + F(x2)
2) F(ax) = aF(x)

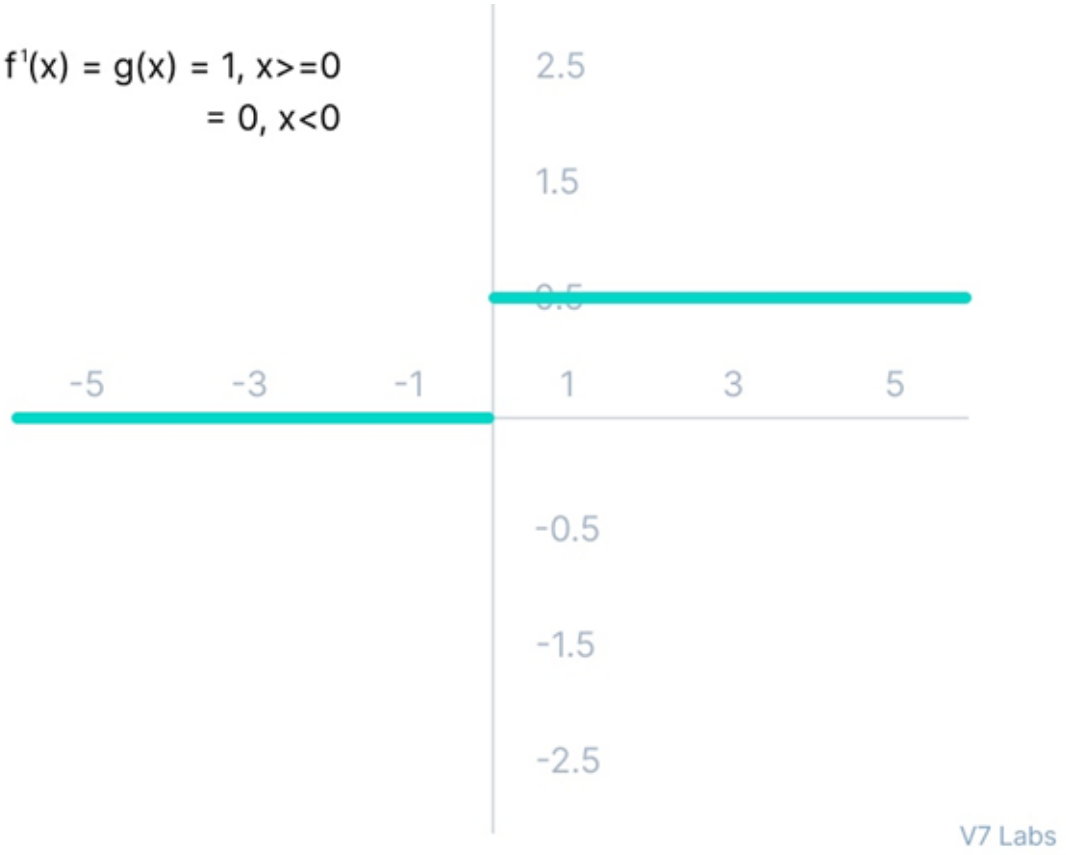


Why ReLU provides non-linearity?

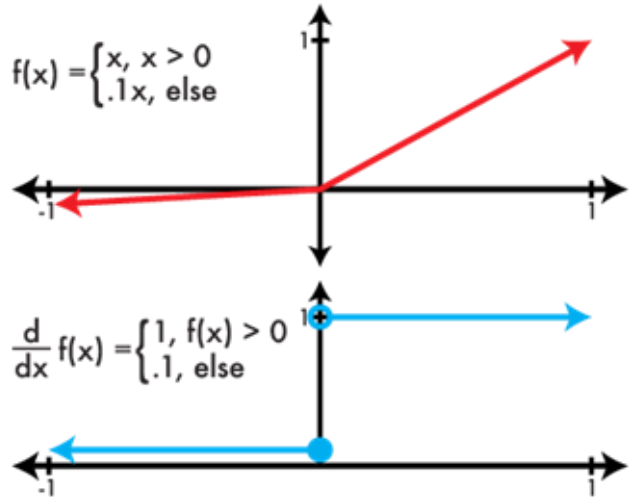


Taken from https://www.blog.dailydoseofds.com/p/a-visual-and-intuitive-guide-to-what#:~:text=The%20core%20point%20to%20understand,of%20a%20non%2Dlinear%20curve.

The Dying ReLU Problem

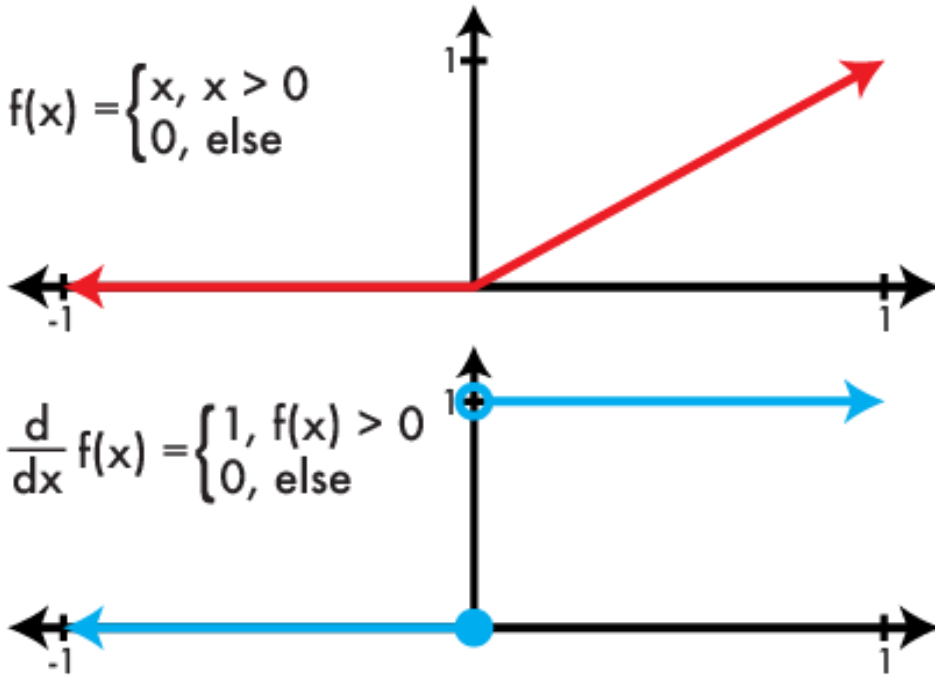


LeakyReLU Activation



- No information loss (compared to ReLU)
- Solves “dying ReLU” problem (i.e. all neurons output 0)
- Similar to ReLU, pays less attention to less important neurons
- Not always better than ReLU

Do you see any other issues with ReLU?



Do you see any other issues with ReLU?

- Exploding gradients: Exact opposite phenomena we had with Sigmoid (vanishing gradients)

Do you see any other issues with ReLU?

- Exploding gradients: Exact opposite phenomena we had with Sigmoid (vanishing gradients)
  - Do gradient clipping, or layer-wise normalization

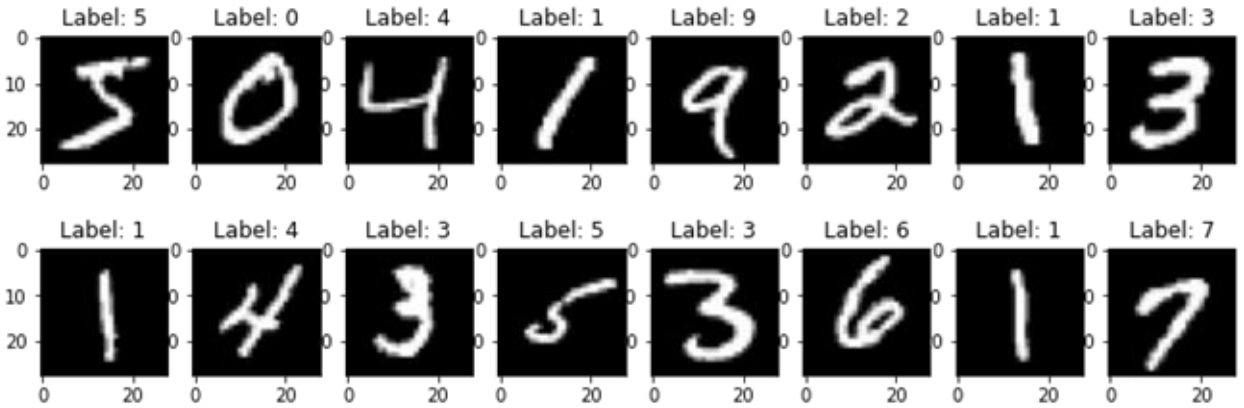
Homework 4
Neural Network

MNIST: Handwriting recognition

50,000 images of handwriting

Input: 28 x 28 x 1 (grayscale) 784 pixel values

Numbers 0-9 10 classes



Train a linear softmax model

Goal: > 95% accuracy on MNIST

Functions You need to Code

Functions You need to Code (**src/hw4/classifier.py**)

```
def activate_matrix(matrix m, ACTIVATION a)
def gradient_matrix(matrix m, ACTIVATION a, matrix d)
def forward_layer(layer *l, matrix in)
def backward_layer(layer *l, matrix delta)
def update_layer(layer *l, double rate, double momentum, double decay)
```

Run Experiments and Write a Report (**hw4.pdf**)

Play around with tryhw4.py file, and answer the questions.

Save your question to a PDF file and submit to Canvas for grading.

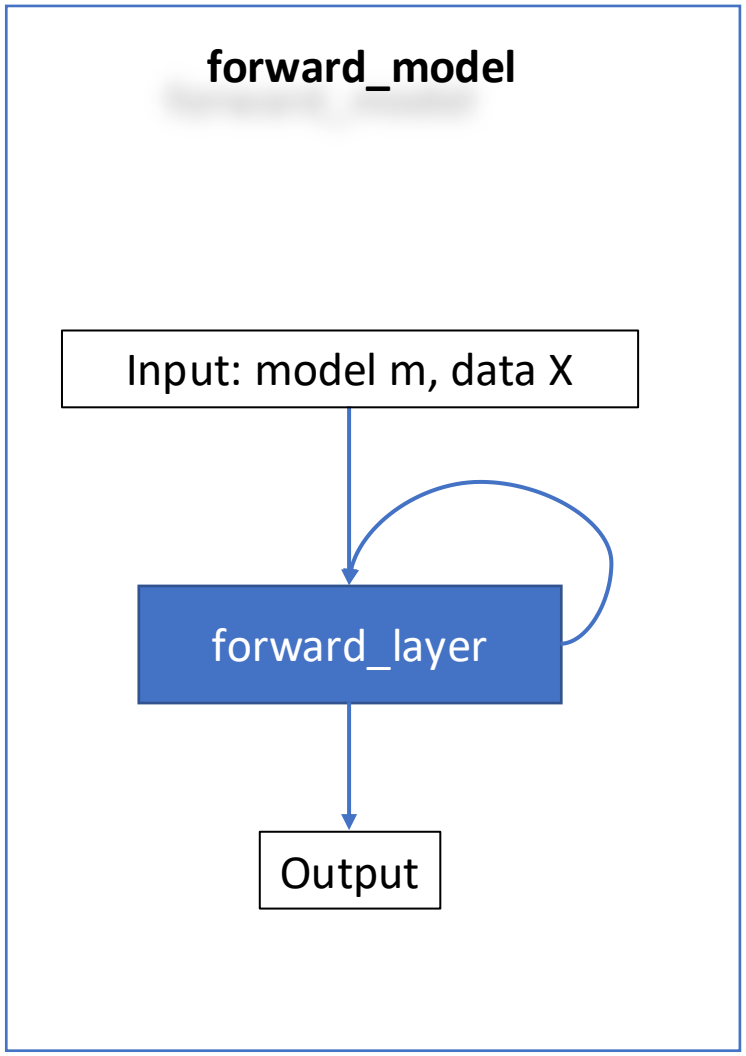
Important Data Structure (classifier.py)

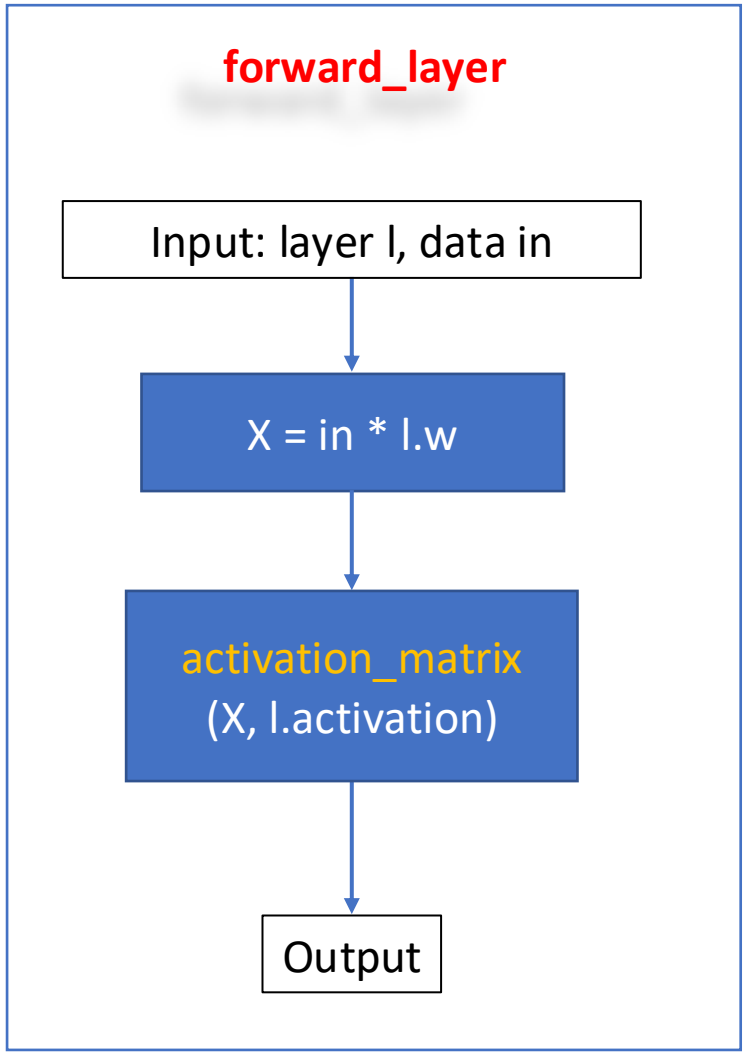
```
class layer:
    def __init__(self):
        self.inp = None         # saved input to this layer
        self.w = None           # current weights
        self.dw = None          # current weight gradient
        self.v = None           # previous weight update / momentum term
        self.out = None         # saved output from this layer
        self.activation = ""    # activation type

class model:
    def __init__(self):
        self.n = 0
        self.layers = []
```

Useful Matrix manipulation functions (src/matrix.py)

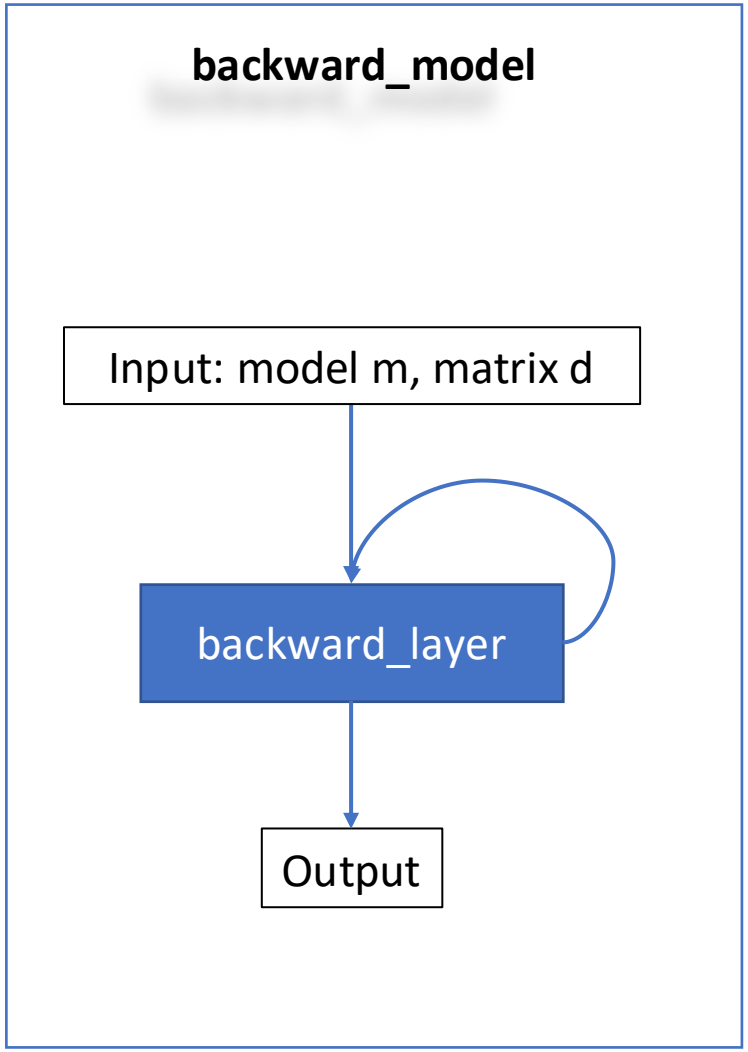
```
matrix_mult_matrix(a, b)        # matrix multiplication: a @ b
transpose_matrix(m)             # transpose
axpy_matrix(a, x, y)            # a * x + y
copy_matrix(m)                  # copy a matrix
make_matrix(rows, cols)         # create a matrix
```

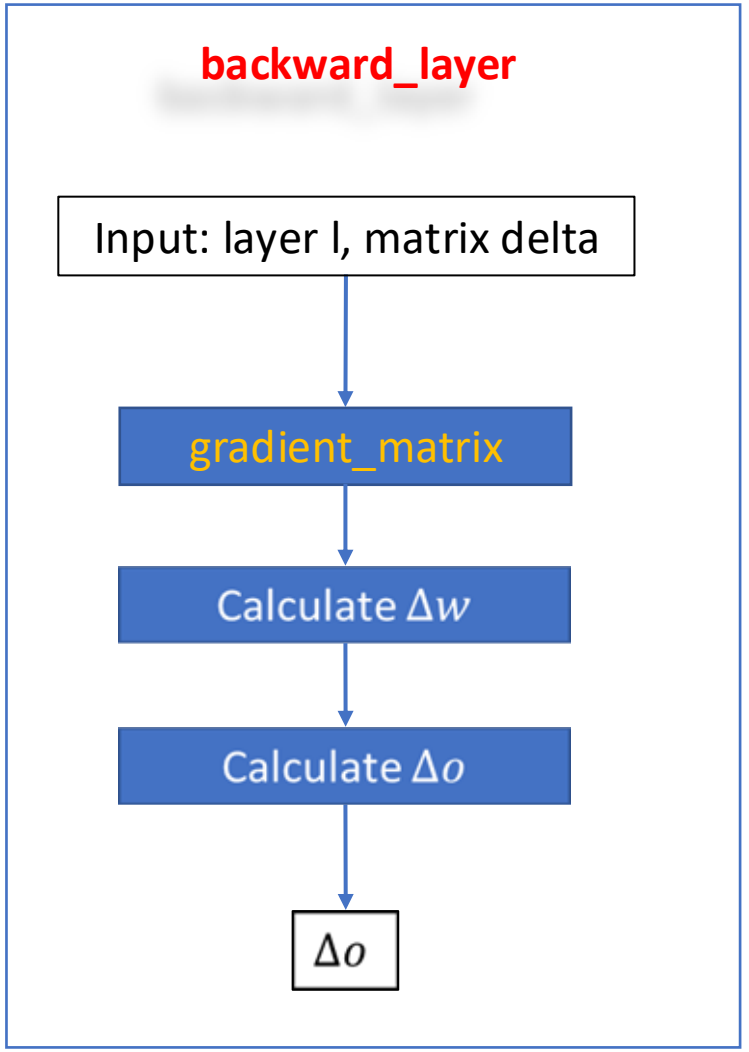
Forward Pass in Homework





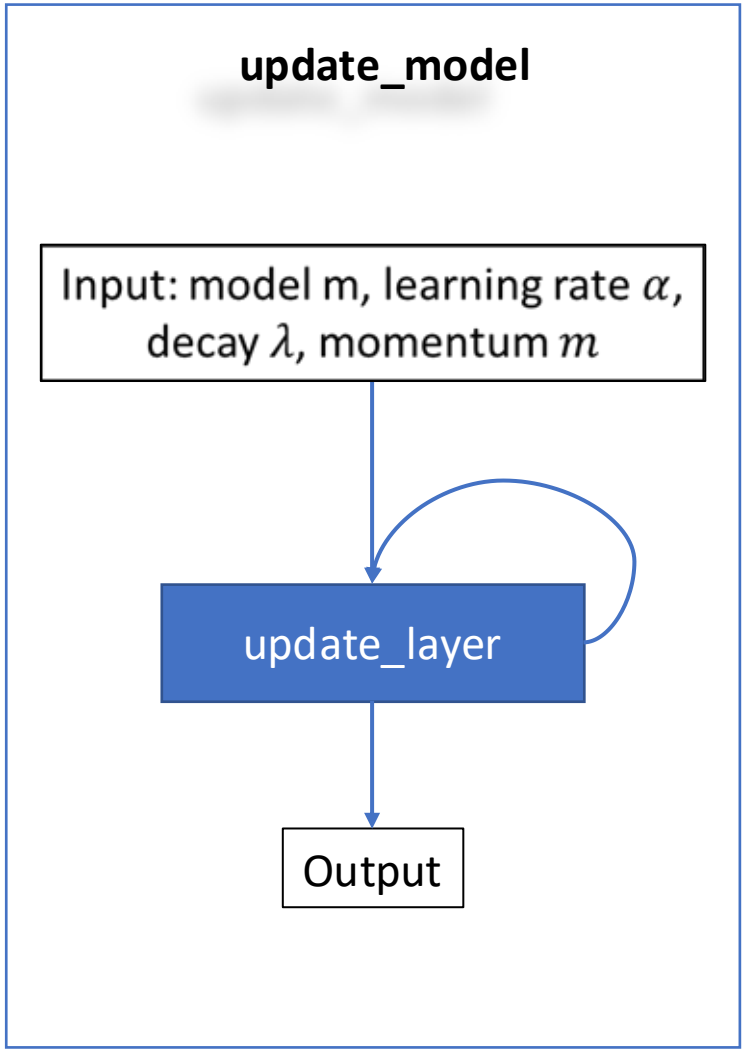
Backward Pass in Homework

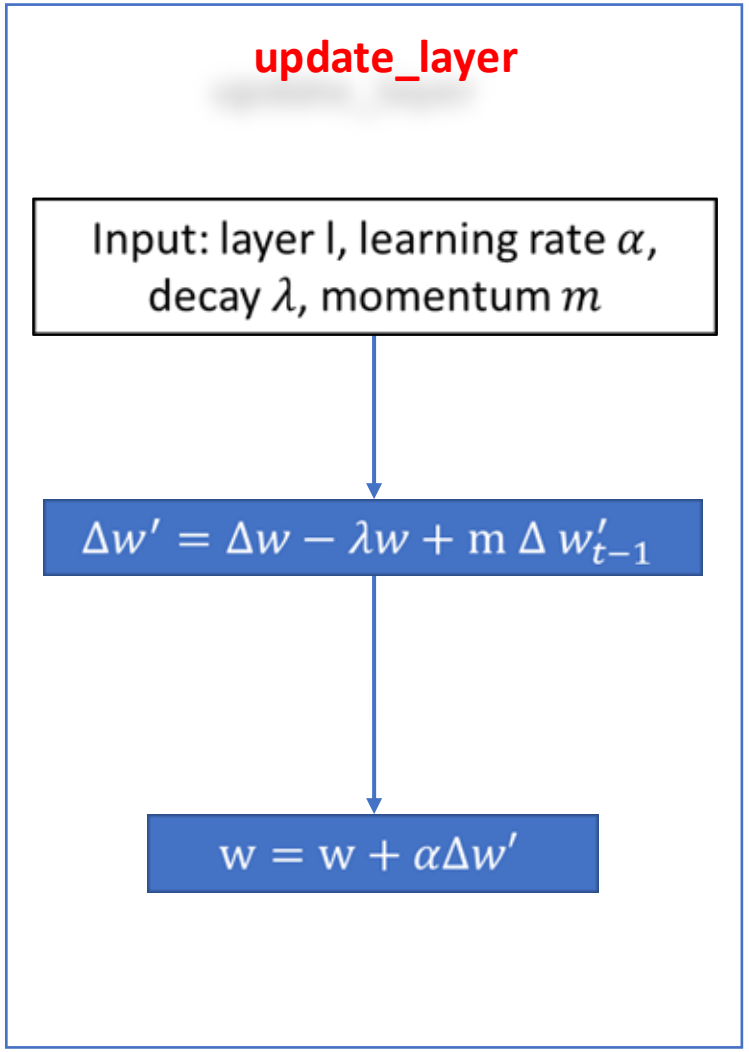




Weight Update in Homework

$\Delta w'_{t-1}$ represent the regularized gradient from the previous step. In the code, we use "l->v" to store this value.





TODO def activate_matrix(matrix m, ACTIVATION a)

Apply activation “a” to the matrix “m”

```
# Run an activation function on each element in a matrix,
# modifies the matrix in place
# matrix m: Input to activation function
# ACTIVATION a: function to run
def activate_matrix(m, a):
    for i in range(m.rows):
        sum_val = 0.0
        for j in range(m.cols):
            x = m.data[i][j]
            if a == 'LOGISTIC':
                pass # TODO
            elif a == 'RELU':
                pass # TODO
            elif a == 'LRELU':
                pass # TODO
            elif a == 'SOFTMAX':
                pass # TODO
            sum_val += m.data[i][j]

        if a == 'SOFTMAX':
            pass # TODO: have to normalize by sum if we are using SOFTMAX
```

TODO `void gradient_matrix(matrix m, ACTIVATION a, matrix d)`

Calculate g'(m) * d, and store in-place to matrix d.
The matrix "m" is the output of a layer, and matrix "d" is the Δ of output.

```
# Calculates the gradient of an activation function and multiplies it into
# the delta for a layer
# matrix m: an activated layer output
# ACTIVATION a: activation function for a layer
# matrix d: delta before activation gradient
def gradient_matrix(m, a, d):
    for i in range(m.rows):
        for j in range(m.cols):
            x = m.data[i][j]
            # TODO: multiply the correct element of d by the gradient
```

TODO `matrix forward_layer(layer *l, matrix in)`

Given the input data “in” and layer “l”, calculate the output data.

```
# Forward propagate information through a layer
# layer *l: pointer to the layer
# matrix in: input to layer
# returns: matrix that is output of the layer
def forward_layer(l, inp):
    l.inp = inp  # Save the input for backpropagation

    # TODO: fix this! multiply input by weights and apply activation function.
    out = make_matrix(inp.rows, l.w.cols)

    free_matrix(l.out)  # free the old output
    l.out = out         # Save the current output for gradient calculation
    return out
```

TODO matrix backward_layer(layer *l, matrix delta)

Given the layer "l" and delta, perform backward step:
- 1.4.1: Calculate the delta after considering the activation
- 1.4.2: Calculate Δw
- 1.4.3: Calculate and Return Δo (aka "dx").

```
# Backward propagate derivatives through a layer
# layer *l: pointer to the layer
# matrix delta: partial derivative of loss w.r.t. output of layer
# returns: matrix, partial derivative of loss w.r.t. input to layer
def backward_layer(l, delta):
    # 1.4.1
    # delta is dL/dy
    # TODO: modify it in place to be dL/d(xw)


    # 1.4.2
    # TODO: then calculate dL/dw and save it in l->dw
    free_matrix(l.dw)
    dw = make_matrix(l.w.rows, l.w.cols) # replace this
    l.dw = dw


    # 1.4.3
    # TODO: finally, calculate dL/dx and return it.
    dx = make_matrix(l.inp.rows, l.inp.cols) # replace this

    return dx
```

TODO `void update_layer(layer *l, double rate, double momentum, double decay)`

Given a layer “l”, learning rate, momentum, and decay rate, Update the weight (i.e. l->w)

```
    # Update the weights at layer l
    # layer *l: pointer to the layer
    # double rate: learning rate
    # double momentum: amount of momentum to use
    # double decay: value for weight decay
    def update_layer(l, rate, momentum, decay):
        # TODO:
        # Calculate Δw_t = dL/dw_t - λw_t + mΔw_{t-1}
        # save it to l->v


        # Update l->w


        # Remember to free any intermediate results to avoid memory leaks
        pass
```

Functions You Need to Know before Experiments

For simplicity, we already filled the following functions for you. You should read and understand these functions before running experiments.

```
def make_layer(int input, int output, ACTIVATION activation)
def forward_model(model m, matrix X)
def backward_model(model m, matrix dL)
def update_model(model m, double rate, double momentum, double decay)
def accuracy_model(model m, data d)
def cross_entropy_loss(matrix y, matrix p)
def train_model(model m, data d, int batch, int iters, double rate, double momentum, double decay)
```

Get the Data

1. Download, Unzip, and Prepare the MNIST Dataset

```
wget https://pjreddie.com/media/files/mnist_train.tar.gz
wget https://pjreddie.com/media/files/mnist_test.tar.gz
tar xzf mnist_train.tar.gz
tar xzf mnist_test.tar.gz
find train -name \*.png > mnist.train
find test -name \*.png > mnist.test
```

2. Download, Unzip, and Prepare the CIFAR-10 Dataset

```
wget http://pjreddie.com/media/files/cifar.tgz
tar xzf cifar.tgz
find cifar/train -name \*.png > cifar.train
find cifar/test -name \*.png > cifar.test
```

Experiments (Write Your Answers to **hw4.pdf**)

1. Coding and Data preparation
2. MNIST Experiments
    1. Linear Softmax Model (1-layer)
        1. Run the basic model
        2. Tune the learning rate
        3. Tune the decay
    2. Neural Network (2-layer NNs and 3-layer NNs)
        1. Find the best activation
        2. Tune the learning rate
        3. Tune the decay
        4. Tune the decay for 3-layer Neural Network
3. Experiments for CIFAR-10
    1. Neural Network (3-layer NNs)
        1. Tune the learning rate and decay