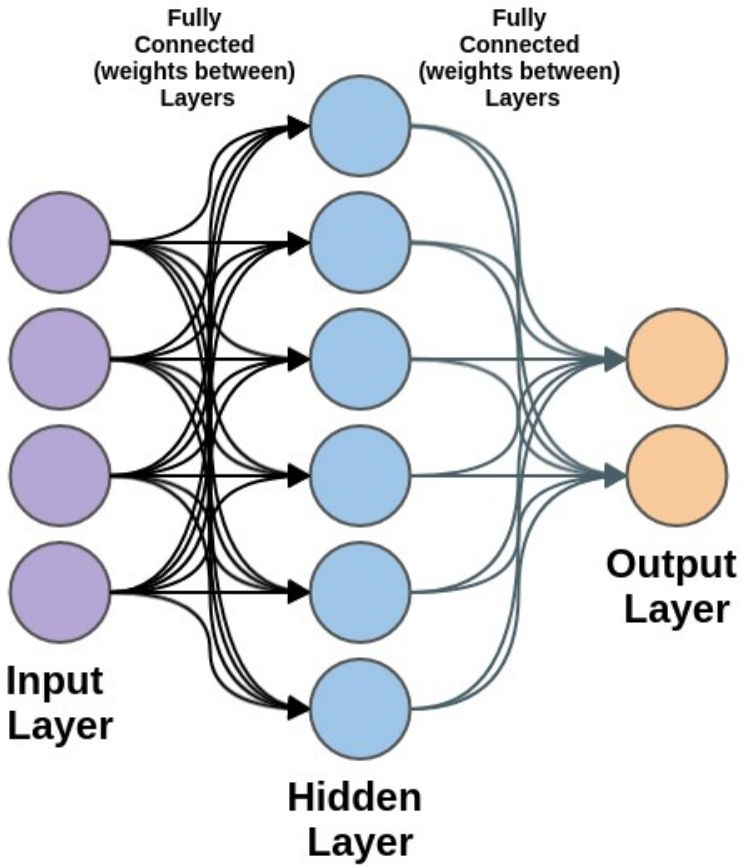
Multi-Class Classification

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

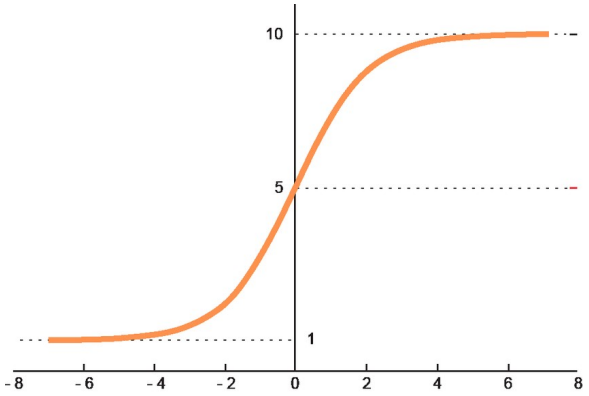
softmax([1,2,7,3,2]):

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.

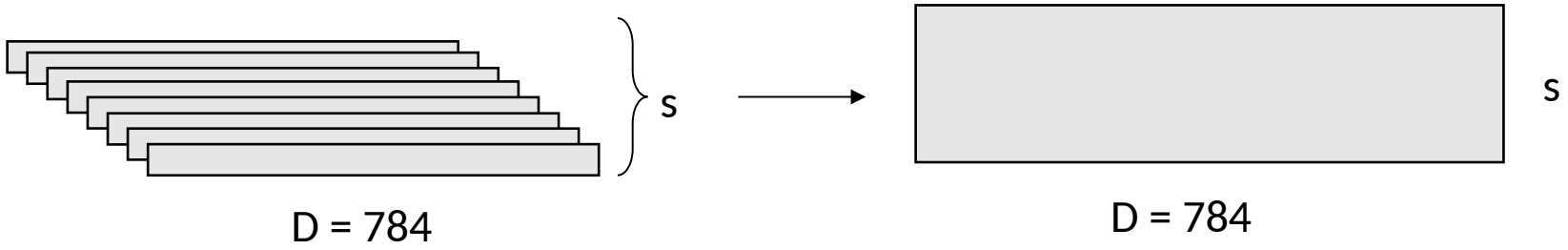


A Simple Example

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

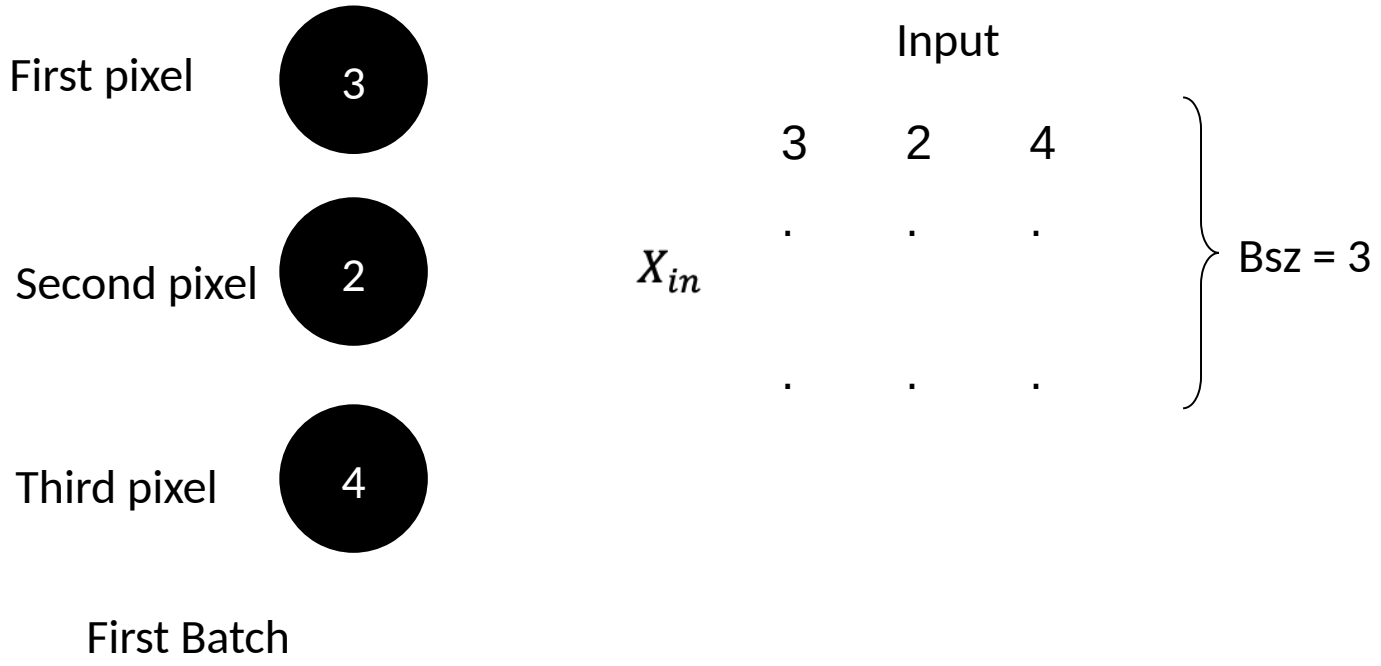
Mini-batch for Machine Learning

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

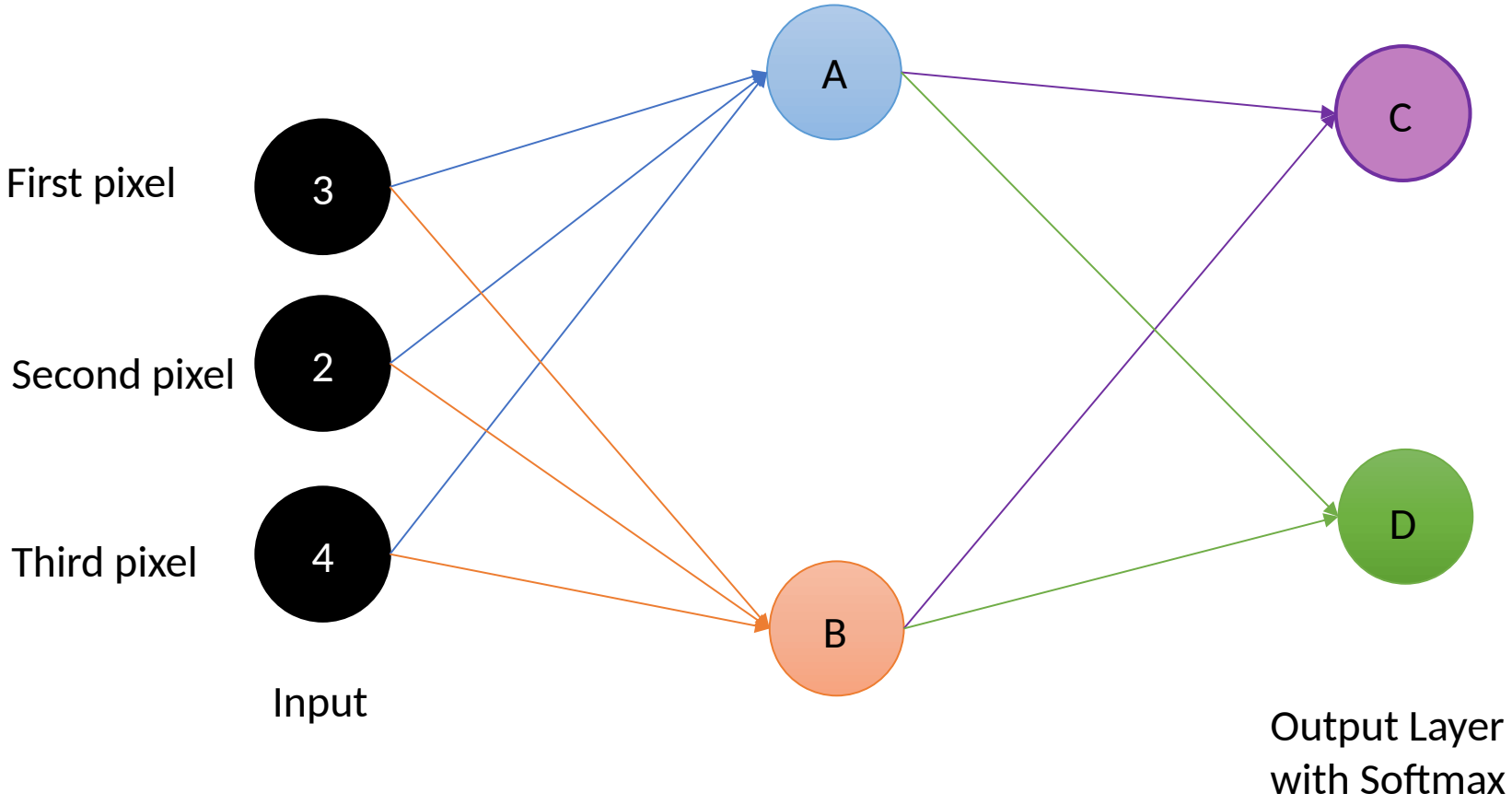


- When s is very large RAM overload
- In HW4, we will use a batch size of 128

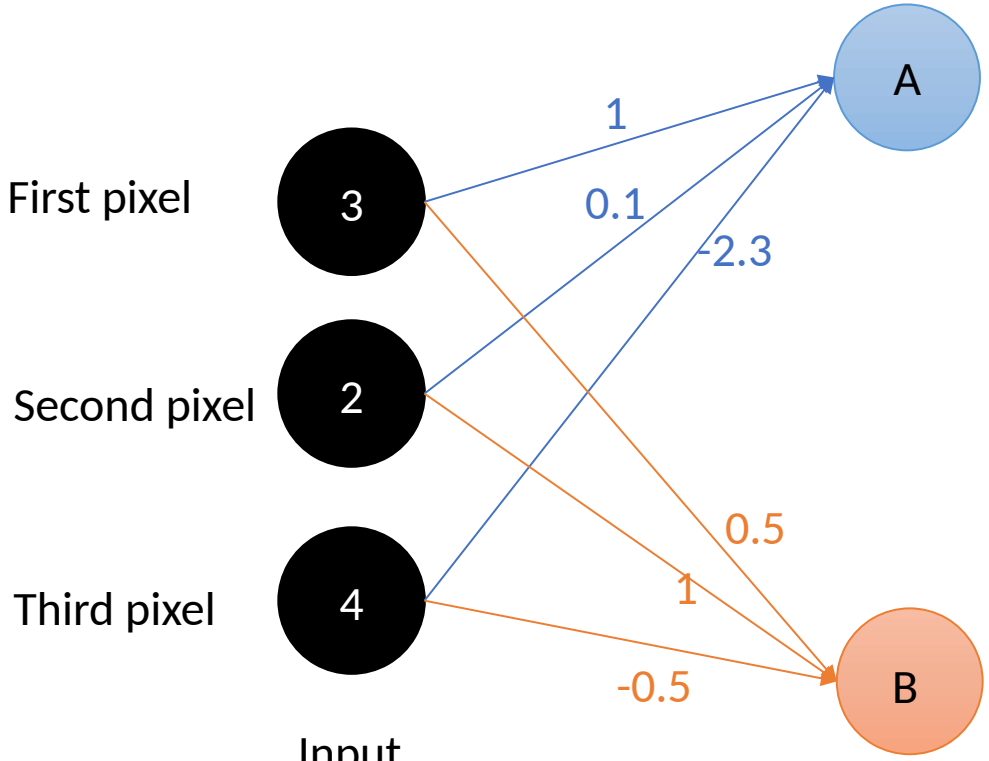
Neural Network Easy Example



Neural Network Easy Example



Neural Network Easy Example



Input

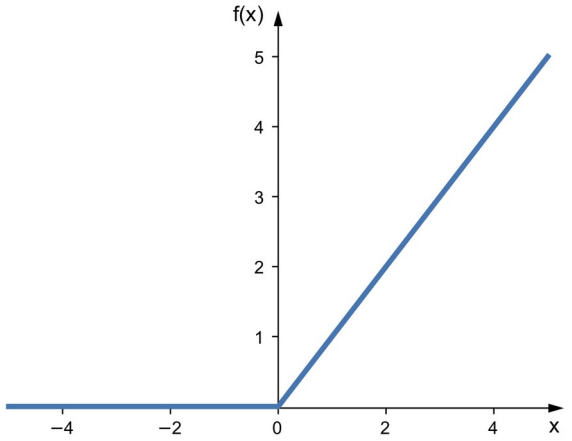
	3	2	4
X_{in}	.	.	.
	.	.	.

1-st Layer (ReLU)

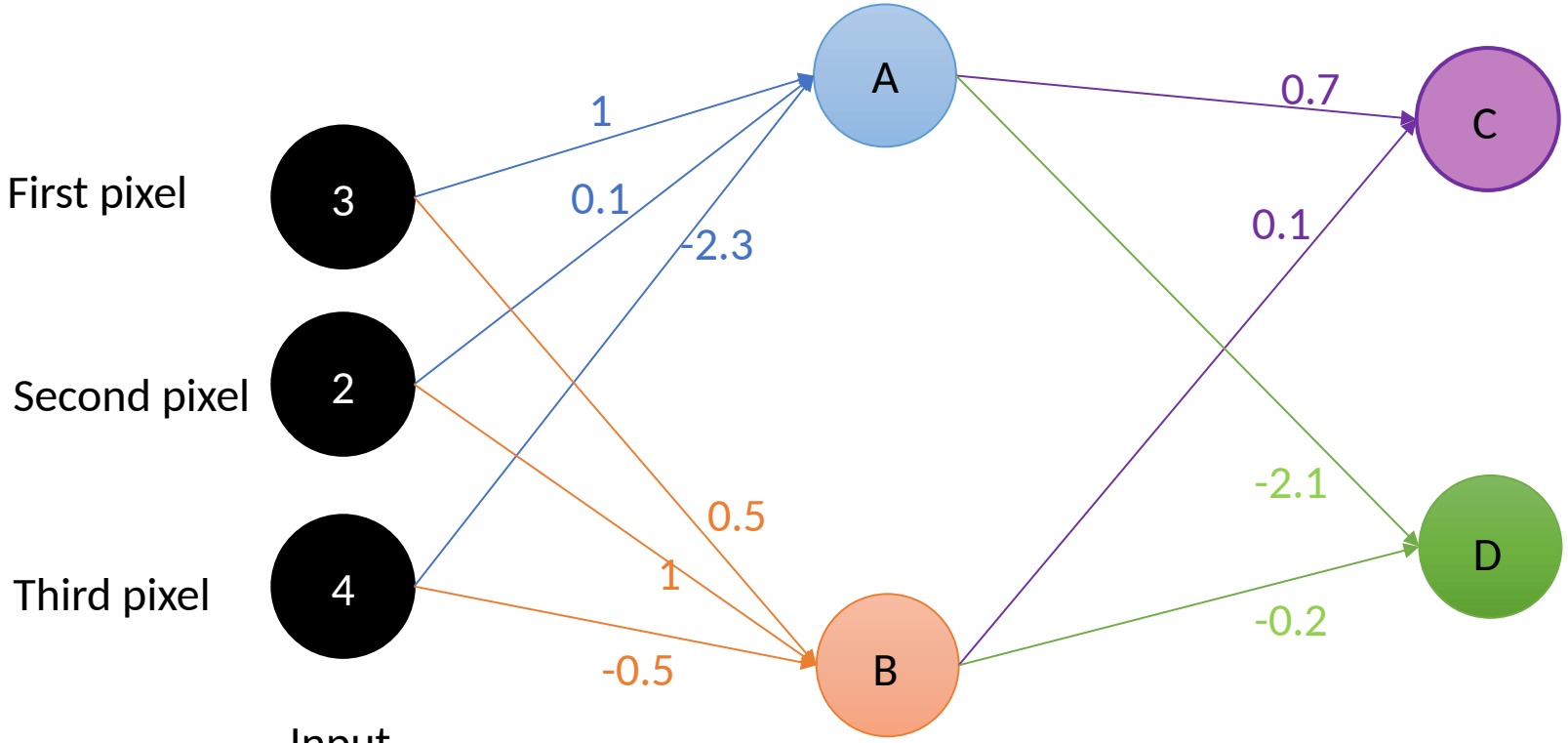
w_1

	A	B
	1	0.5
	0.1	1
	-2.3	-0.5

Weights for level 1



Neural Network Easy Example



Input

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

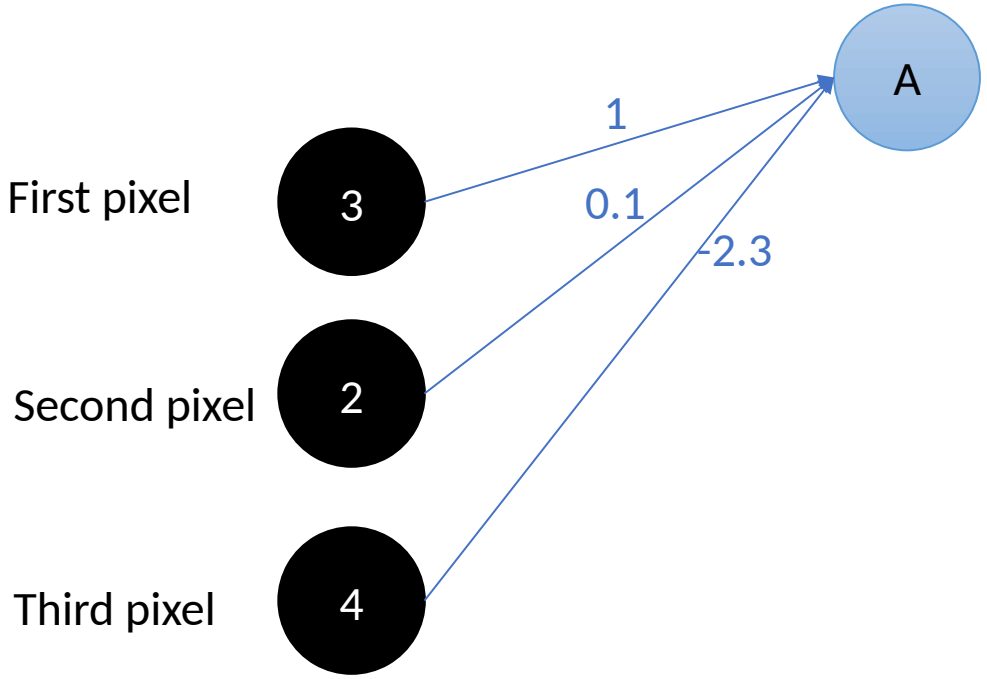
1	0.5
0.1	1
-2.3	-0.5

Output Layer with Softmax

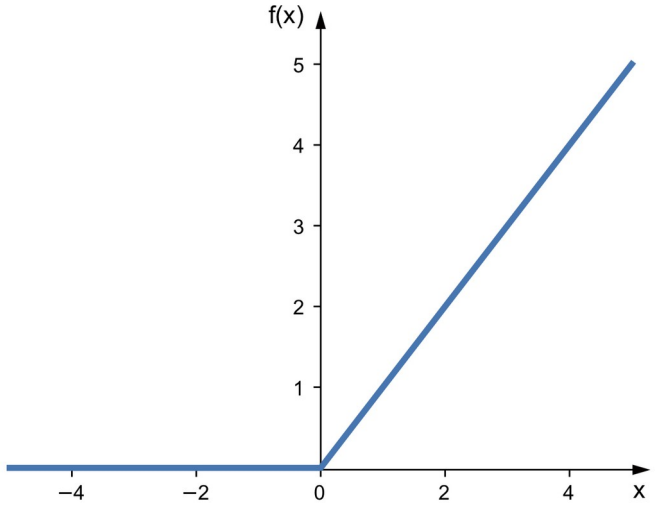
w_2

0.7	-2.1
0.1	-0.2

1st Layer with ReLU

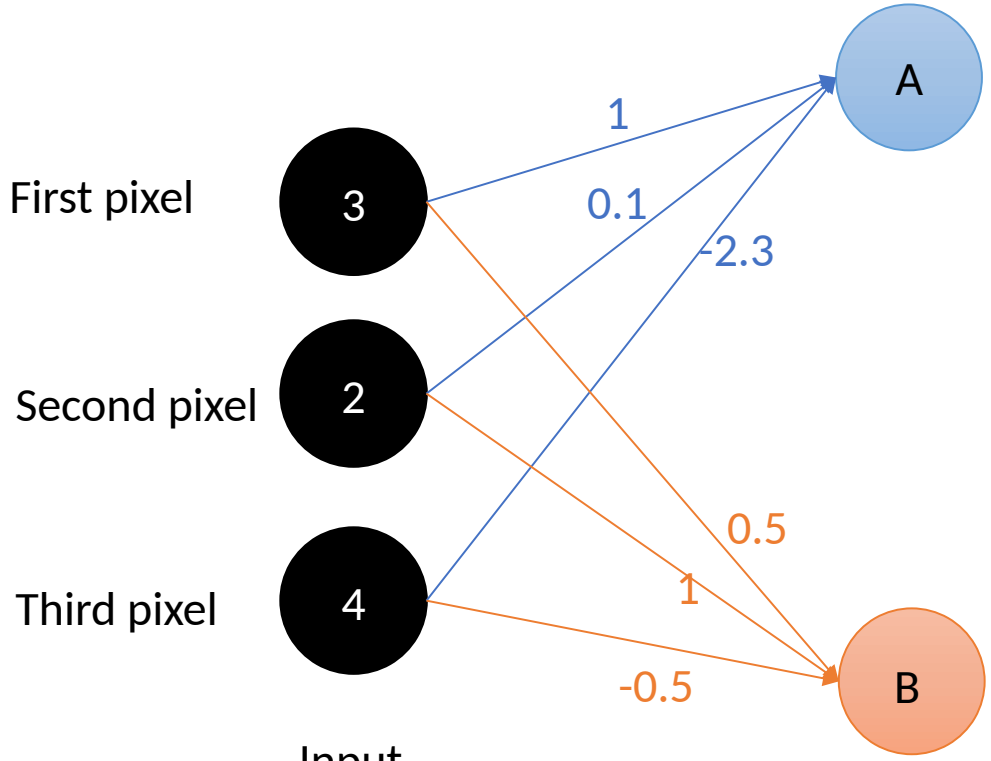


$$Output_A = 3 * 1 + 2 * 0.1 + 4 * -2.3 = -6.0$$
$$Output_A = 0\ (after\ RELU)$$



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

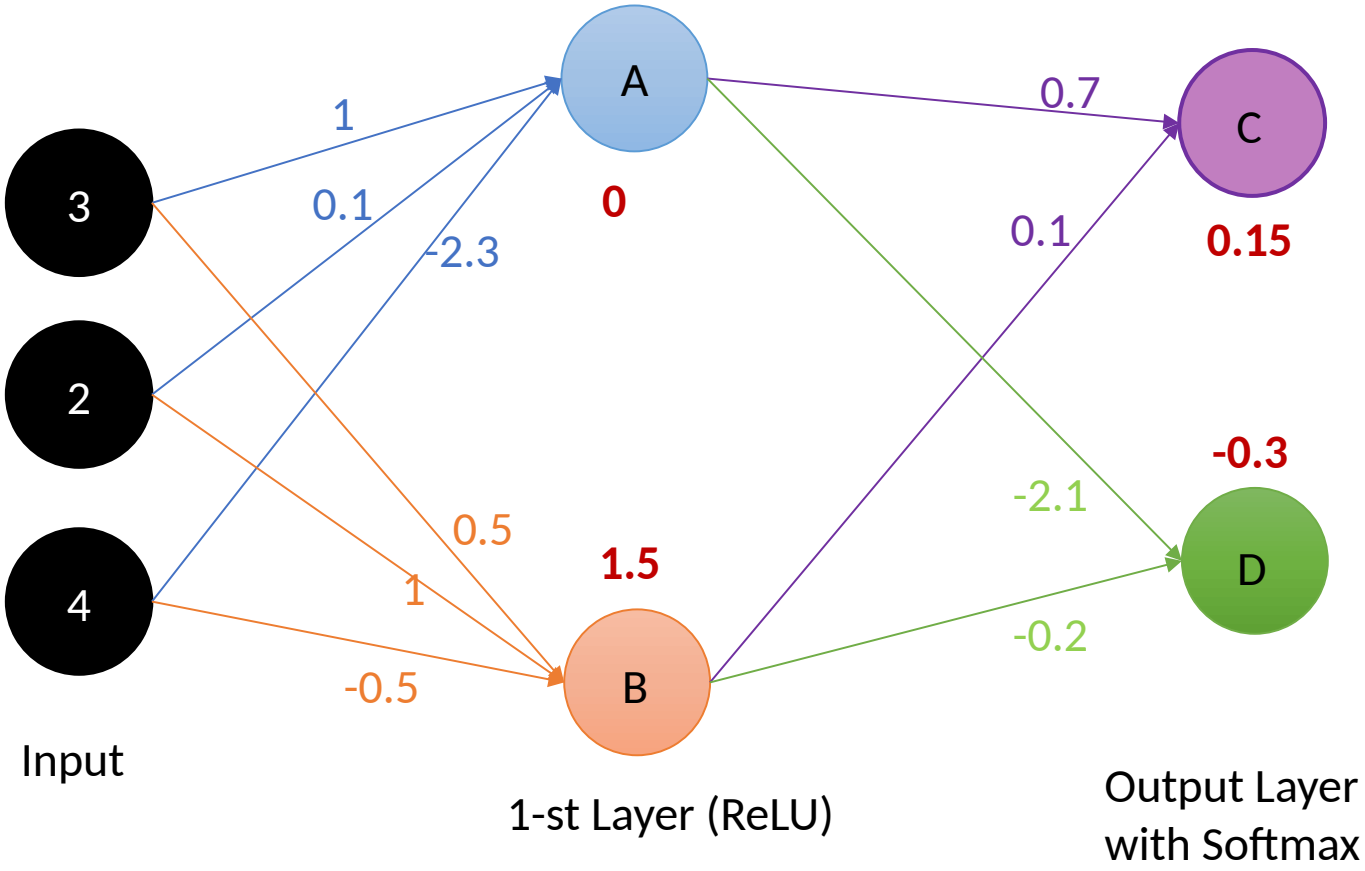
1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

O_1 is outputs from layer 1

Neural Network Easy Example

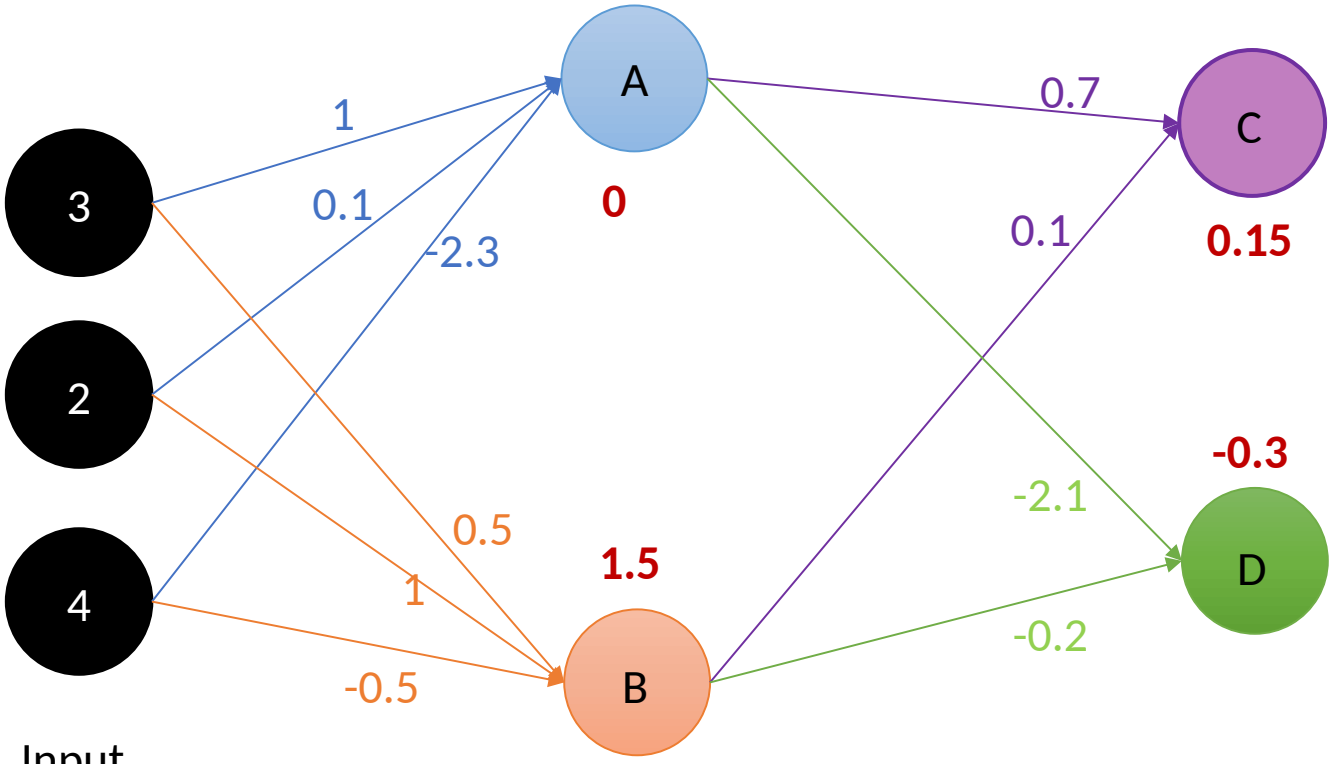


$$output_C = \frac{e^{0.15}}{e^{0.15} + e^{-0.3}} \approx \frac{1.16}{1.16 + 0.74} = 0.61$$

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



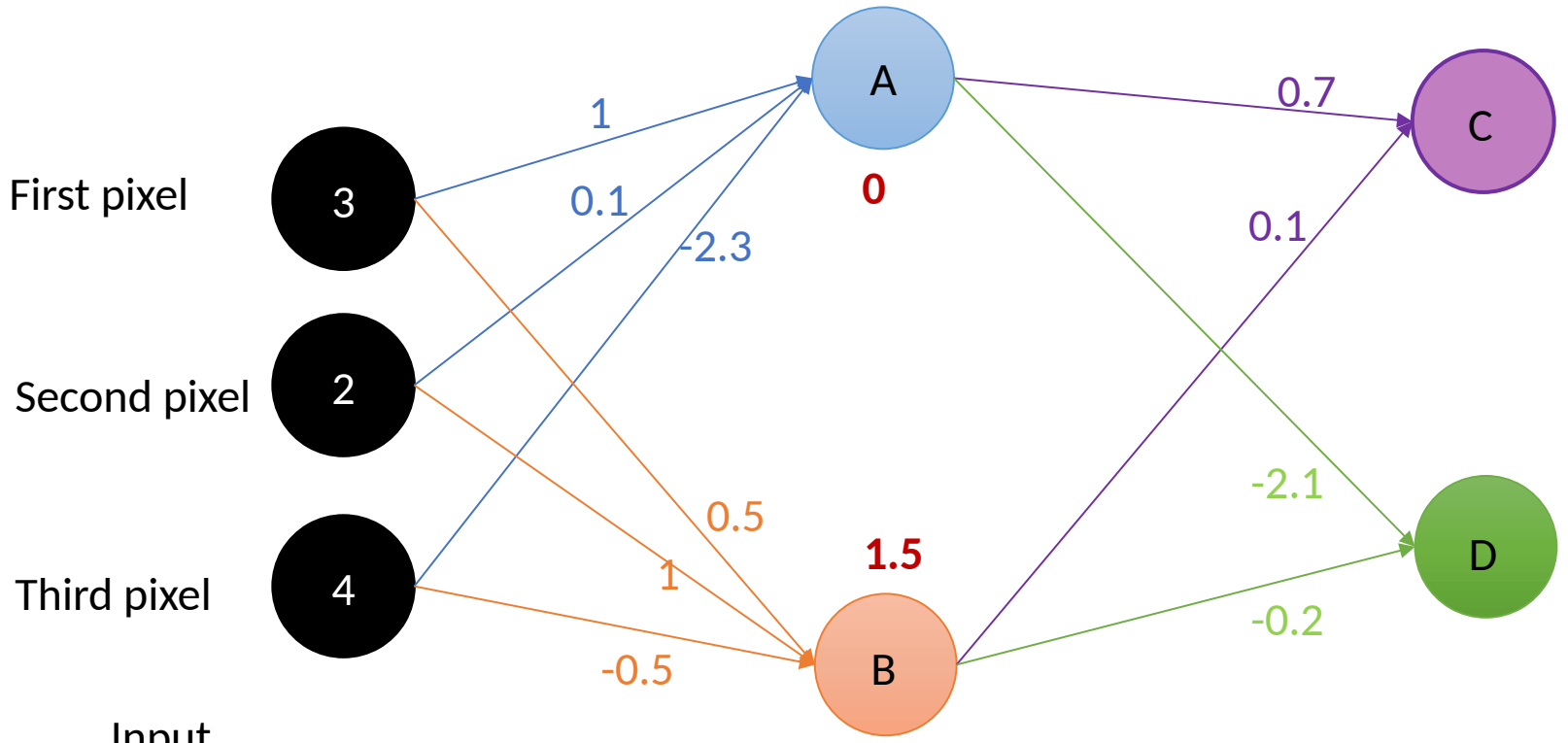
Output O_2 at layer 2

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

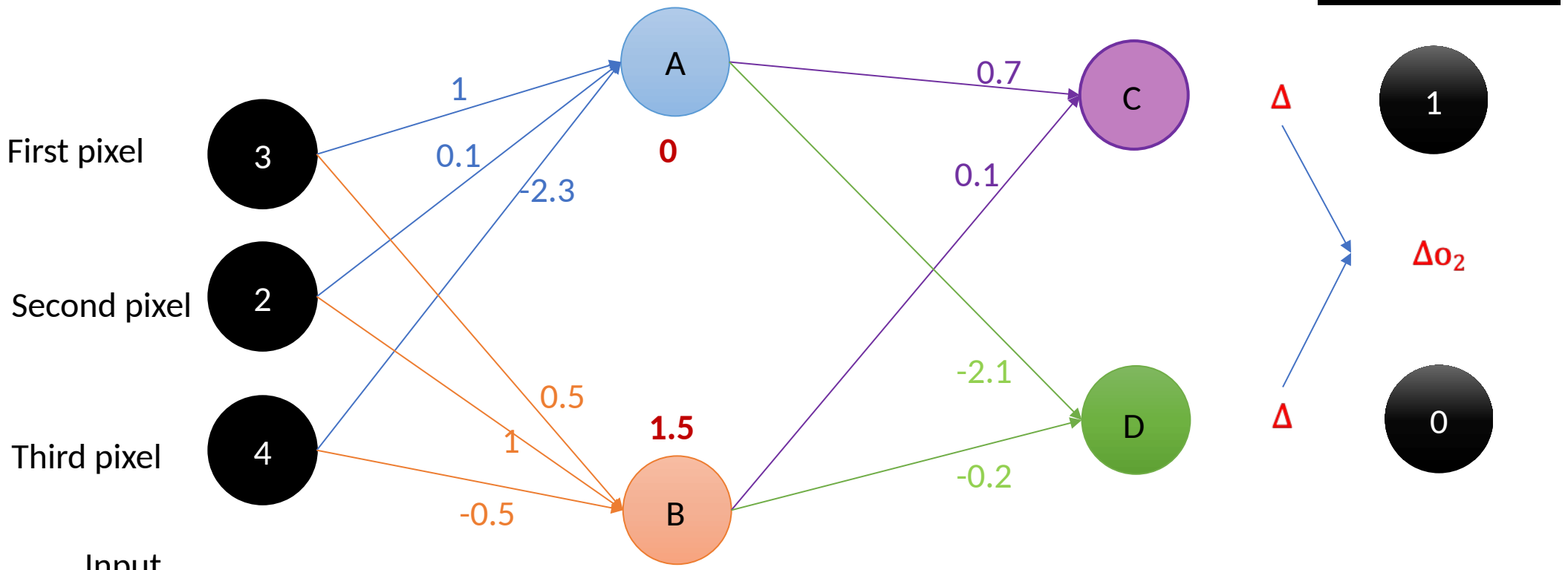
w_2

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

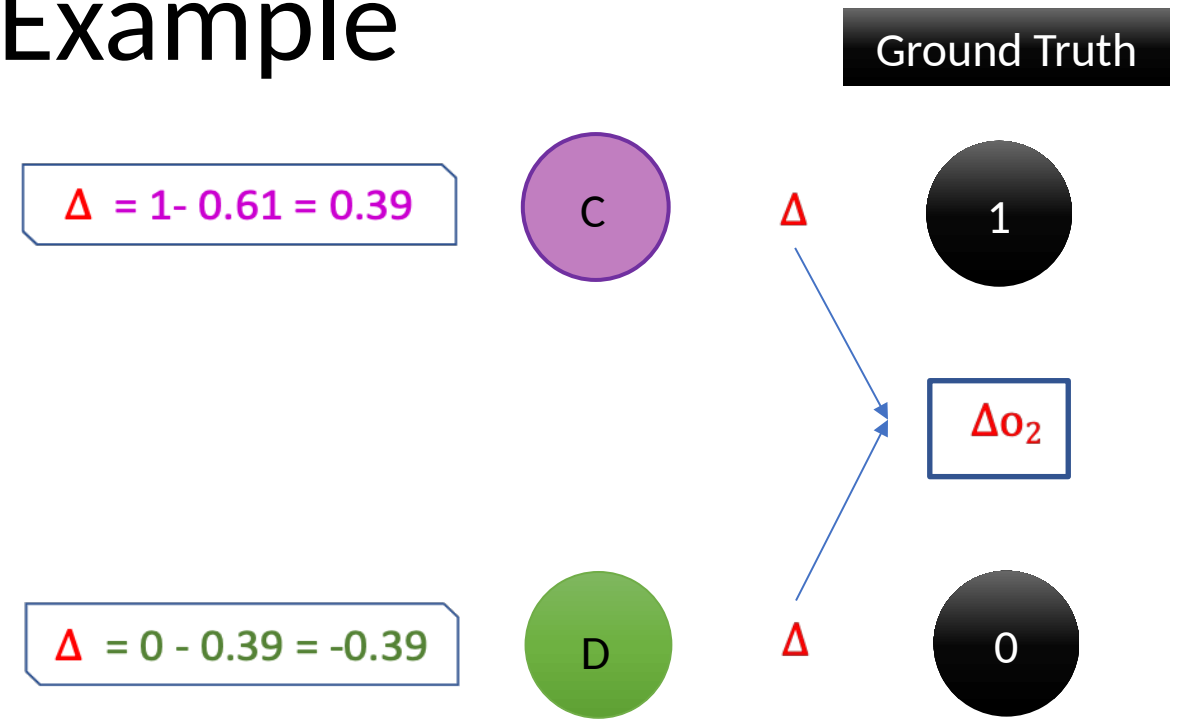
w_2

0.7	-2.1
0.1	-0.2

o_2

0.61	0.39
.	.
.	.

Neural Network Easy Example



$\Delta o_2 =$ [.39 -.39]

o_2

0.61	0.39
.	.
.	.

What to do with this error?

What to do with this error?

Backpropagate it to update the weights!

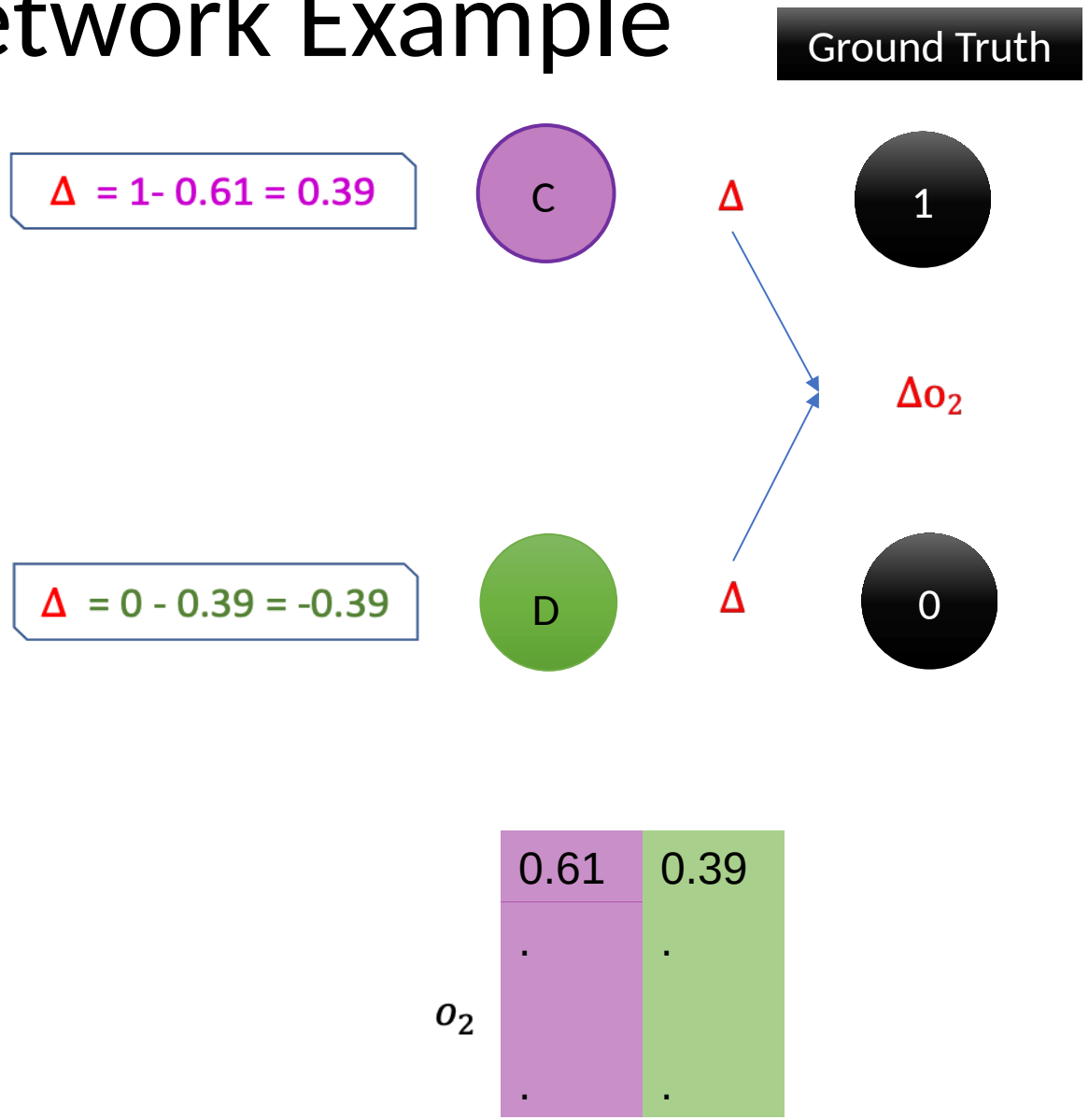
How? We will be doing this (in a few slides).

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

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

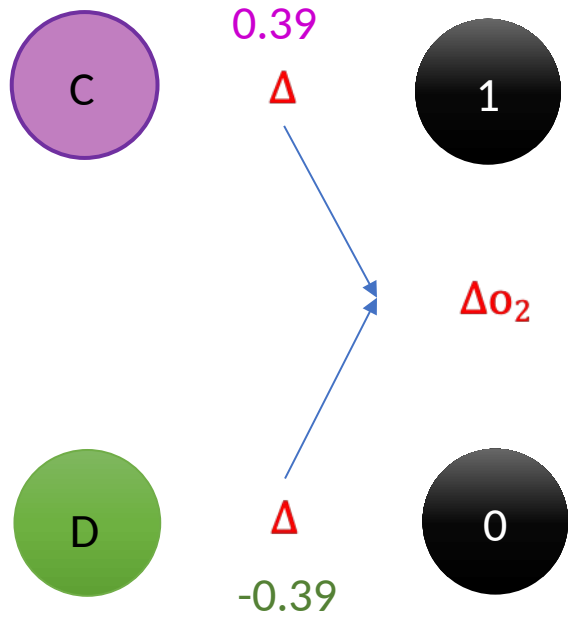
$$\Delta w_t = -\frac{\partial E}{\partial w_t} = o_1^T \, g'(o_2) \circ \Delta o_2$$

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

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

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

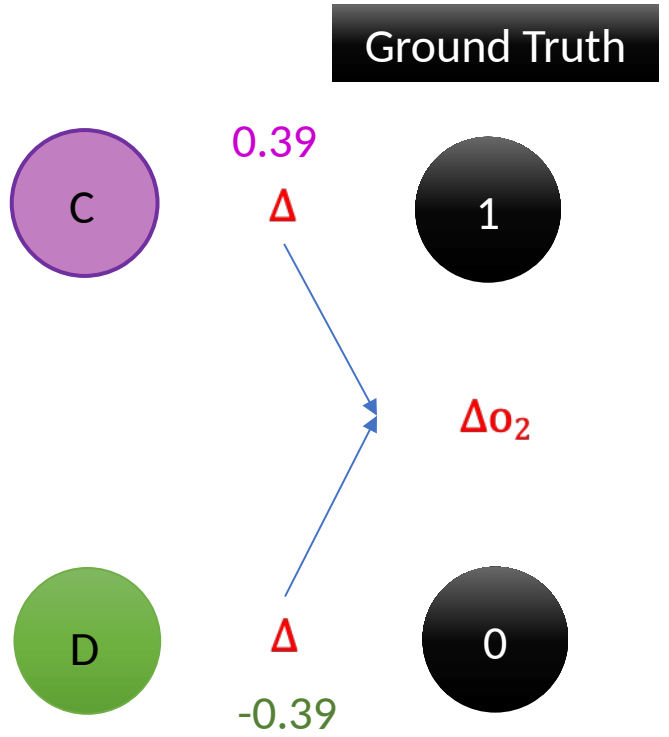
0.39	-0.39
.	.
.	.

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

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.

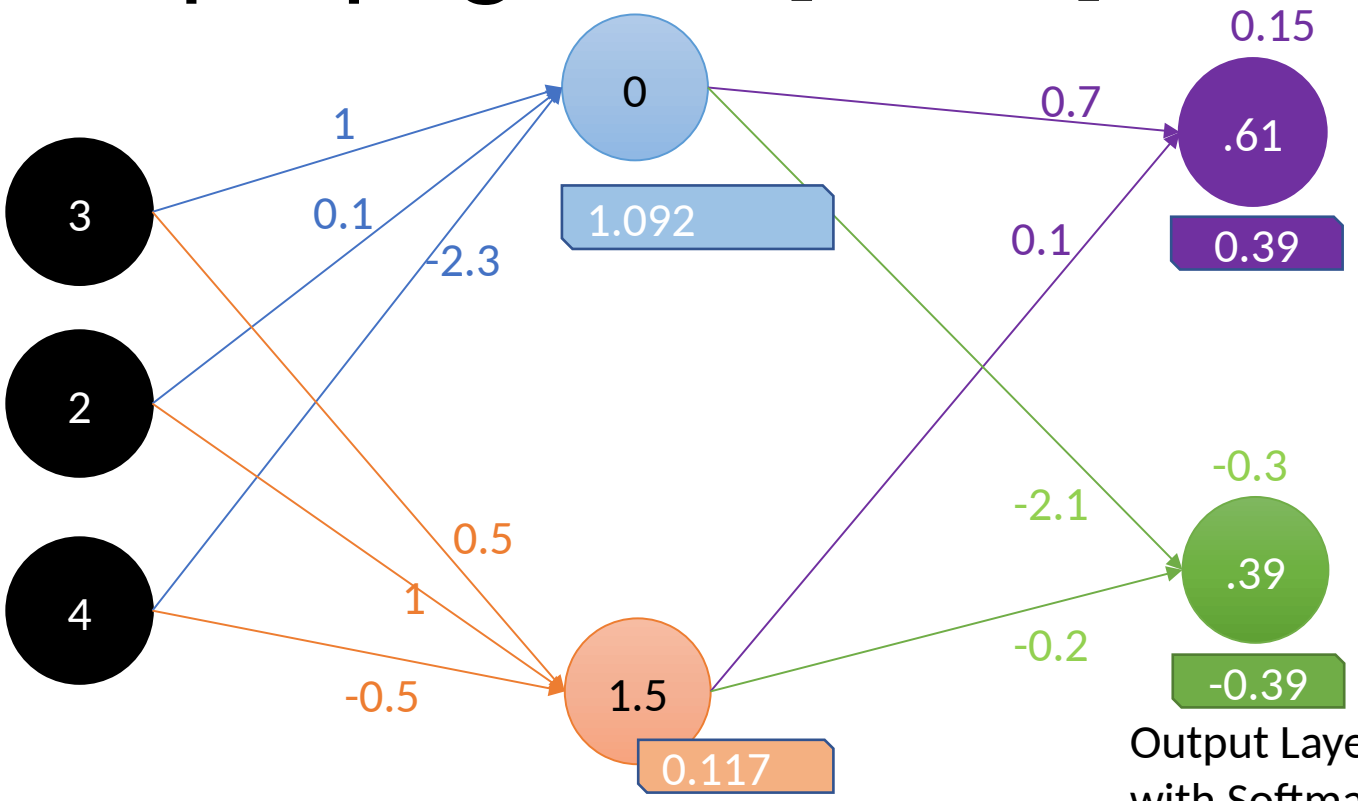
Now to get Δo_1 multiply Δo_2 by W_2^T.

W_2^T =
0.7 −0.1
−0.21 −0.2]

o_2

0.61	0.39
.	.
.	.

Backpropagation [Cont.]



X_{in}

3	2	4
.	.	.
.	.	.

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
.	.
.	.

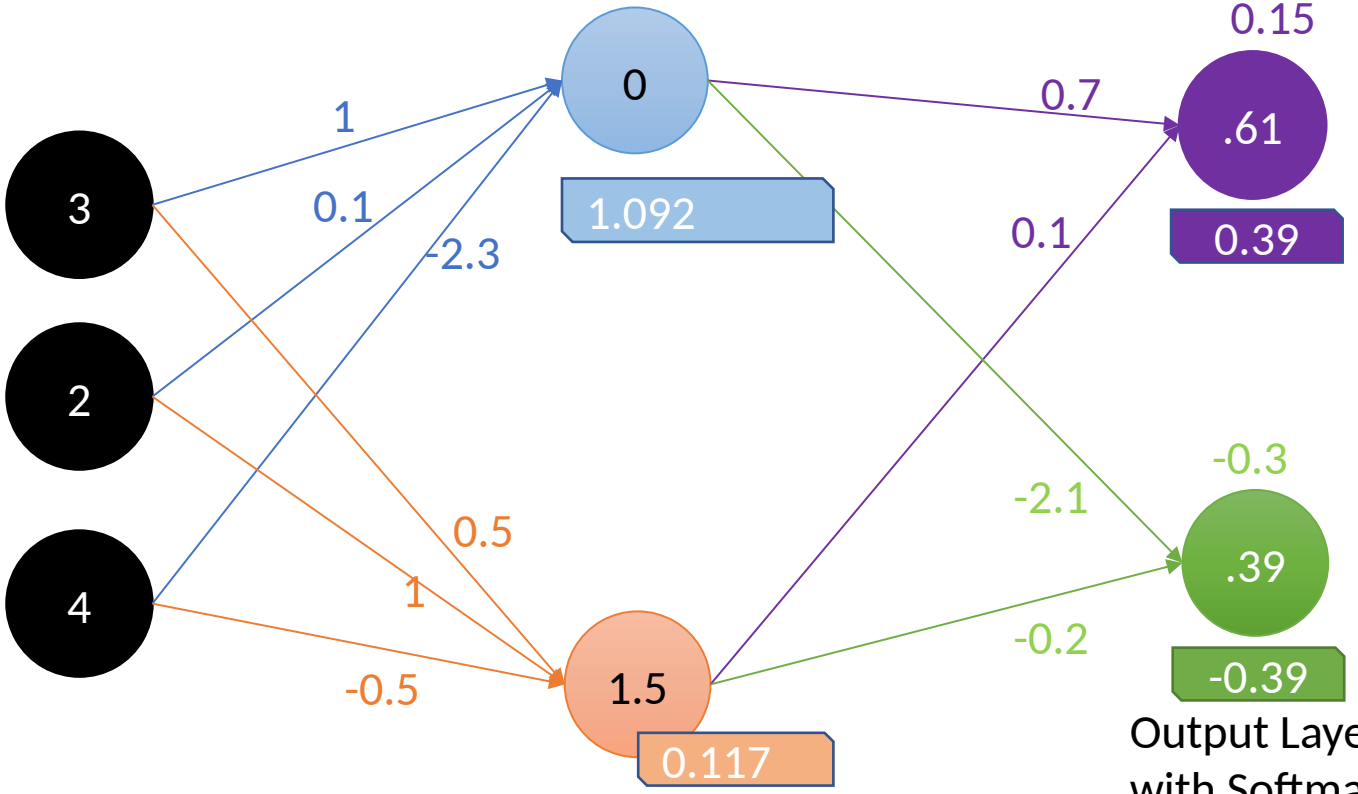
Backpropagation [Cont.]

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

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



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
.	.
.	.

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

0.39	-0.39
.	.
.	.

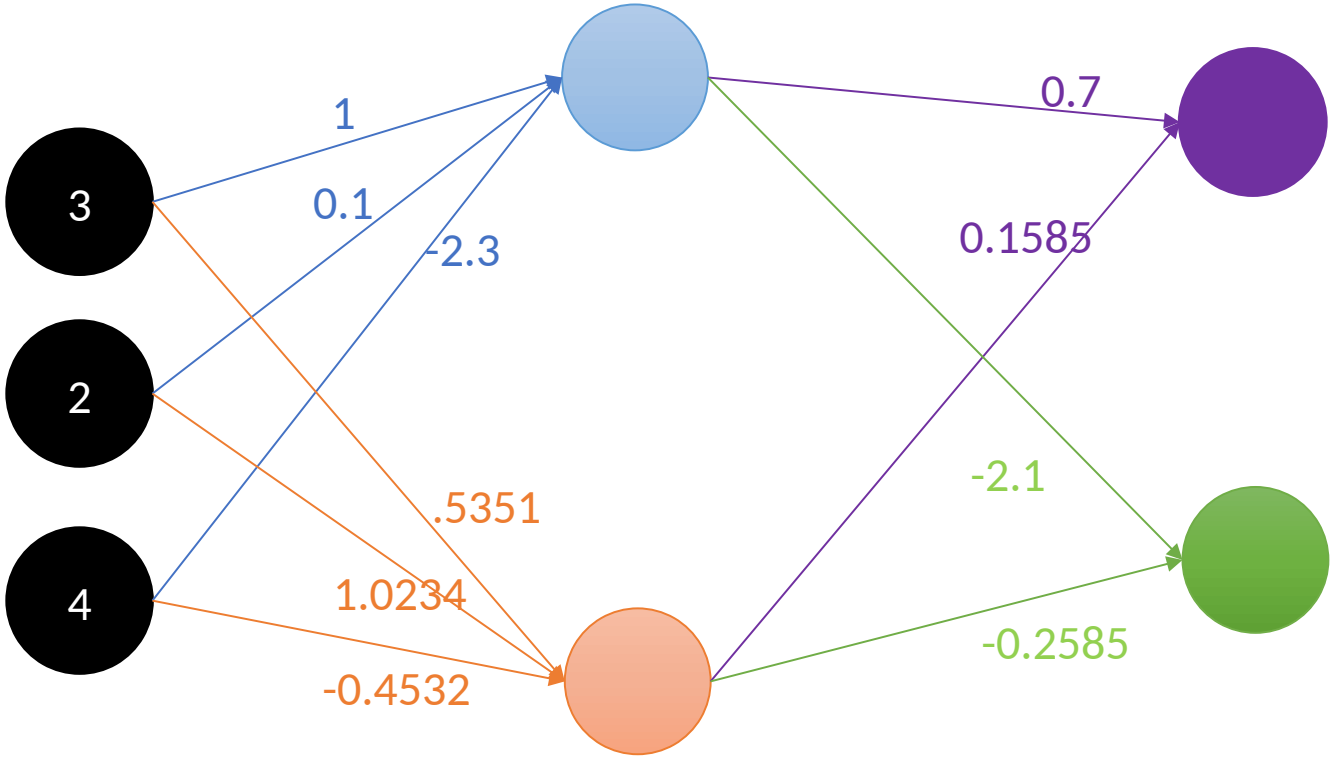
$\Delta w_2 = o_1^T g'(o_2) \circ \Delta o_2$

0	0
0.585	-0.585

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

1.092	0.117
.	.
.	.

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

	3	2	4
X_{in}	.	.	.
	.	.	.

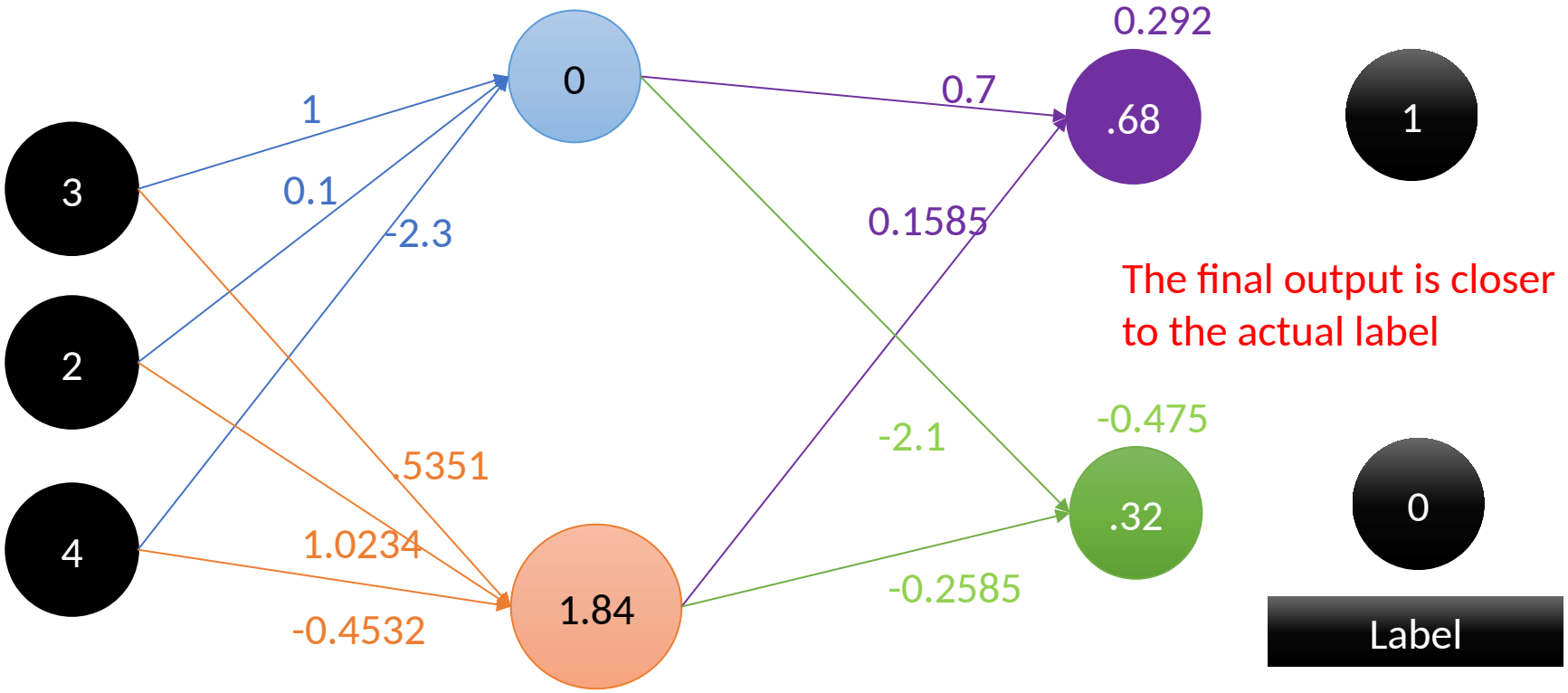
1-st Layer (ReLU)

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

	3	2	4
X_{in}	.	.	.
	.	.	.

1-st Layer (ReLU)

w_1

1	0.5351
0.1	1.0234
-2.3	-0.4532

Output Layer with Softmax

w_2

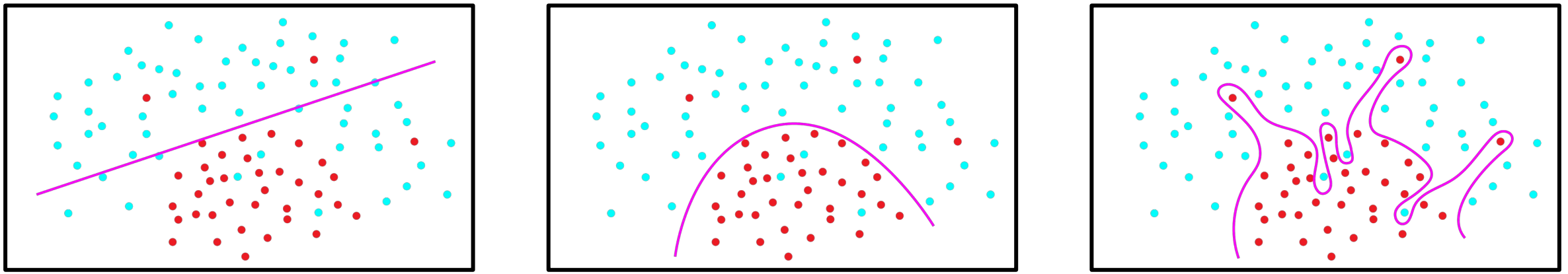
0.7	-2.1
0.1585	-0.2585

Tricks for Training Neural Networks

Problem: Under and Overfitting

Underfitting: the model fails to fit the training data

Overfitting: good training fit, poor generalization to new data



Weight decay: neural network regularization

Δw_t denotes a negative-gradient update direction: $-\nabla L(w_t)$.

We want the weights to be close to 0.

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:

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

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

Shrink weights toward zero each iteration

Momentum: speeding up SGD

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

Before:

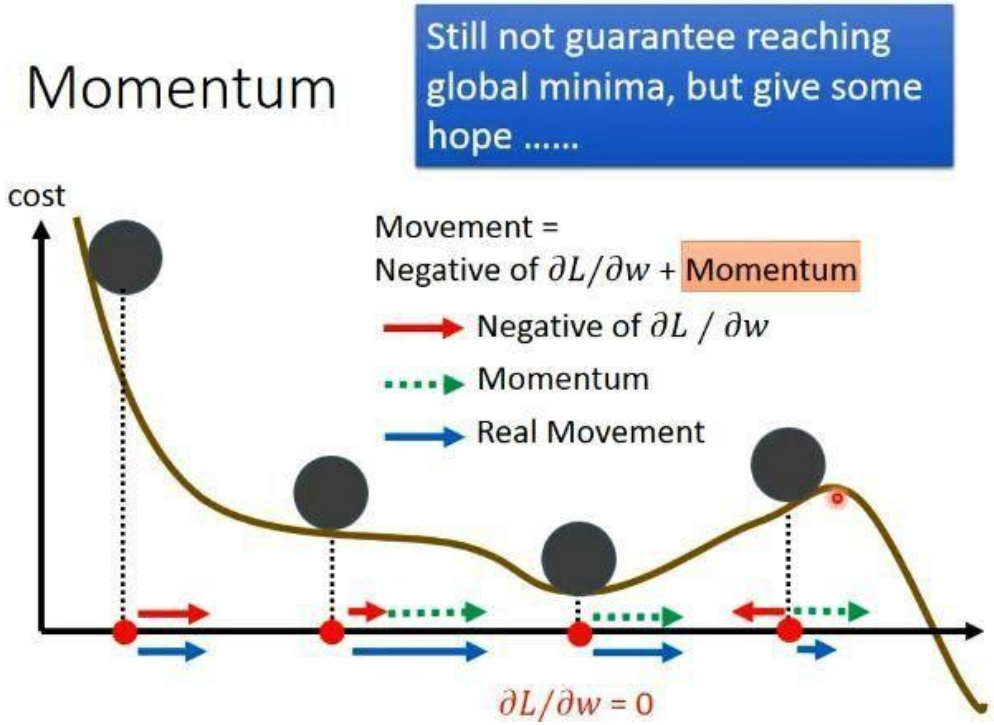
$$\Delta w_t = -\partial/\partial w_t \, L(w_t)$$

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} is the previous momentum velocity, including its history.

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

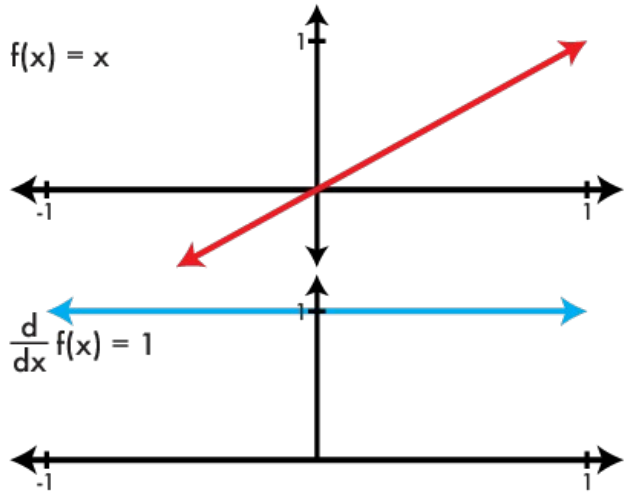
Randomly mask some activations during training.

At test time, disable dropout and use all units.

Activations

Linear Activation

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



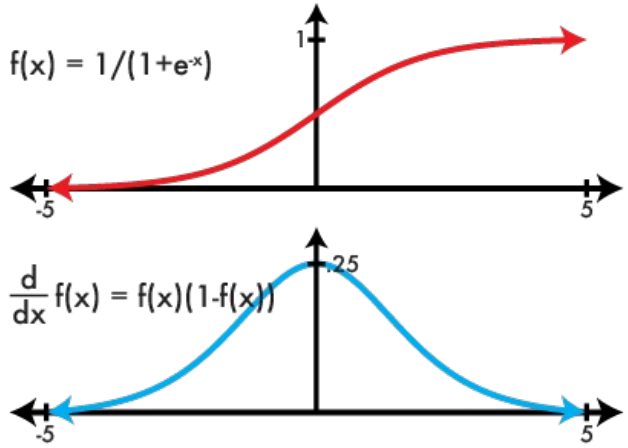
- Only offers linear effects.
- For a 2-layer NN with linear activations for both layers.

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

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

Logistic Activation

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



- Also called the sigmoid activation
- Output is in (0, 1)
- Used for binary class probabilities
- Local derivative is at most 1/4
- From saved output m: derivative = m(1 − m)

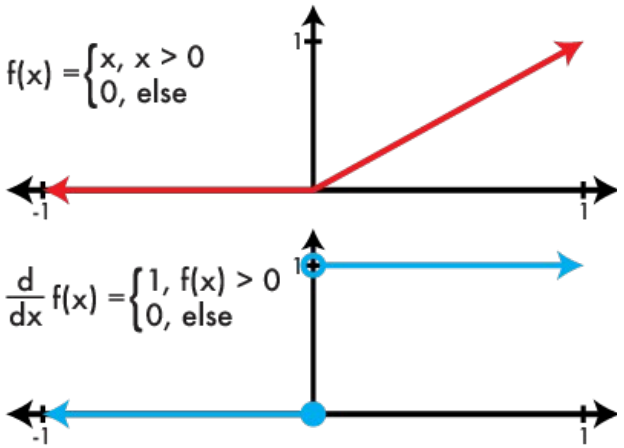
Do you see the problem with Sigmoidal activations?

Do you see the problem with Sigmoidal activations?

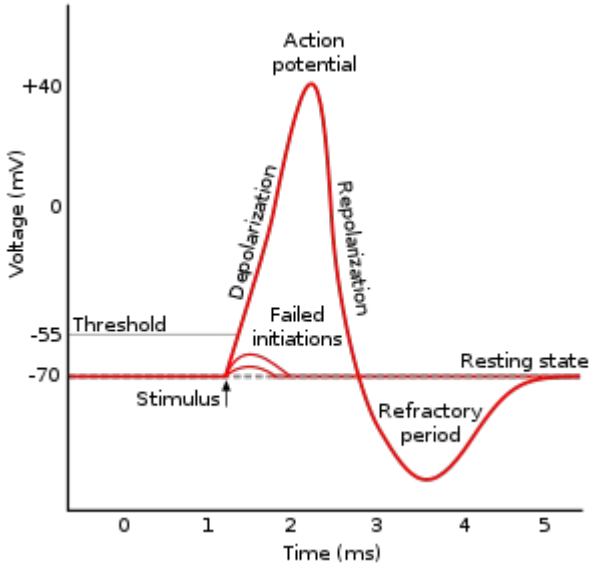
Vanishing gradients!

ReLU Activation

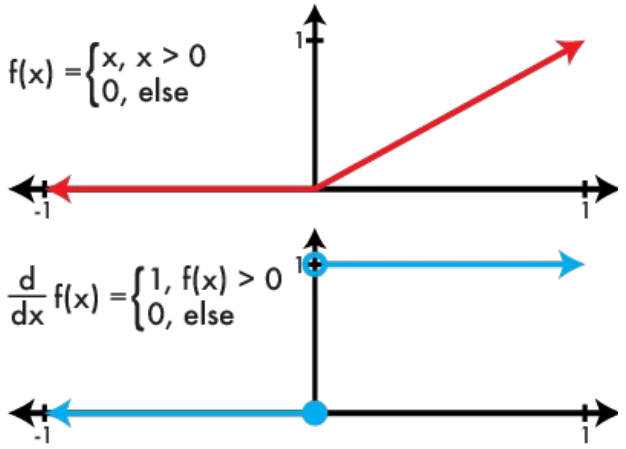
$$g(x) = \max(0, x)$$
$$g'(x) = 1 \text{ if } x > 0; \text{ otherwise } 0$$



- Simple thresholding is fast to compute
- Positive inputs keep derivative 1
- Negative inputs have derivative 0
- Negative values are discarded, not ranked by importance
- No general guarantee that information loss is small



Why ReLU provides non-linearity?

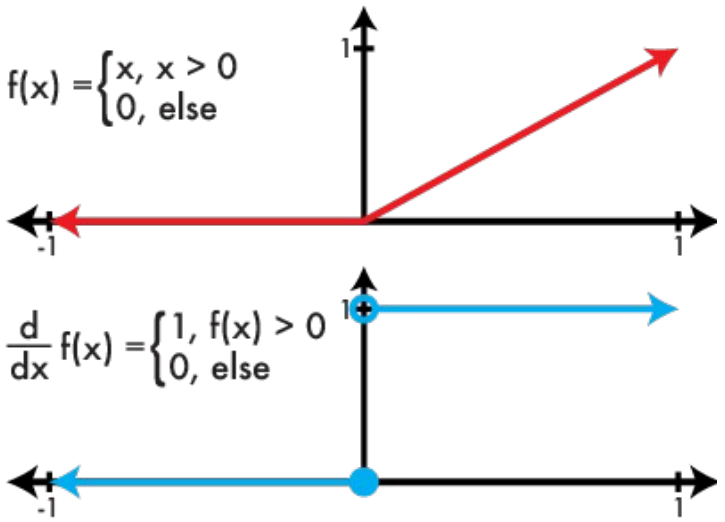




Why ReLU provides non-linearity?

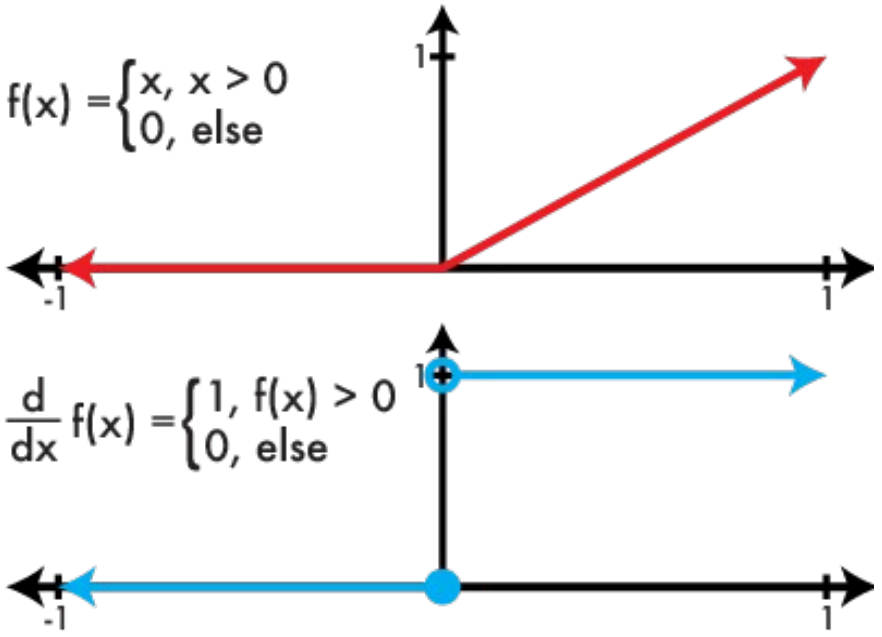
The function is not linear if it does not satisfy the superposition principles: 1) additivity, 2) homogeneity

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

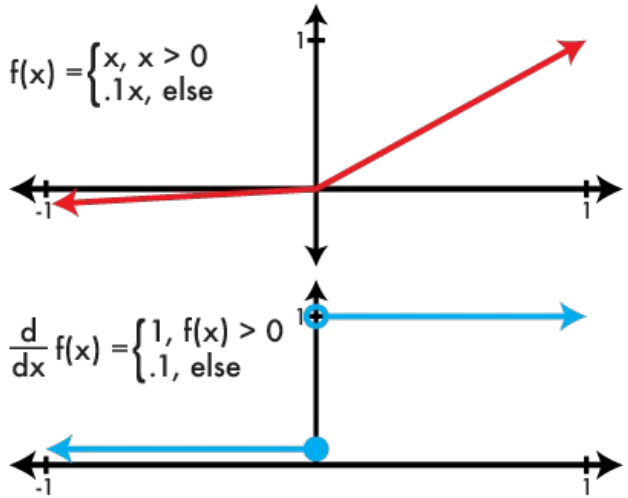


The Dying ReLU Problem

A unit can become inactive on all training examples,
so its ReLU derivative stays zero.

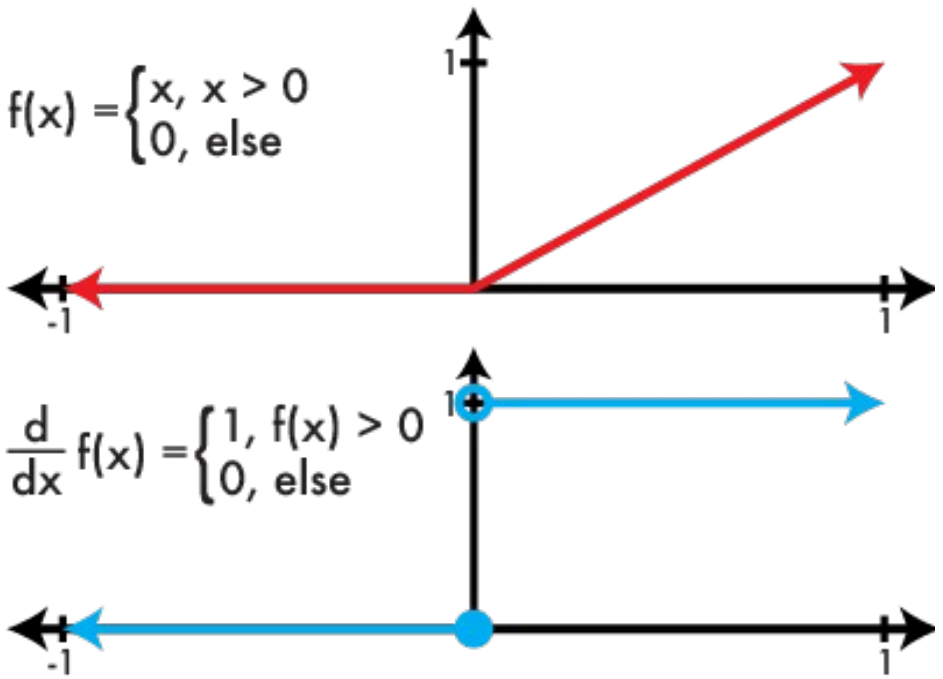


LeakyReLU Activation



- Keeps a nonzero slope for negative inputs
- Helps avoid units stuck with zero local gradient
- HW4 uses a negative-region slope of 0.1
- Not always better than ReLU

Do you see any other issues with ReLU?



Do you see any other issues with ReLU?

- Gradients can still explode through products of weight matrices, even though ReLU derivatives are 0 or 1.

Do you see any other issues with ReLU?

- Gradients can still explode through products of weight matrices.
  - Use suitable initialization and learning rates; clip large gradients when needed.

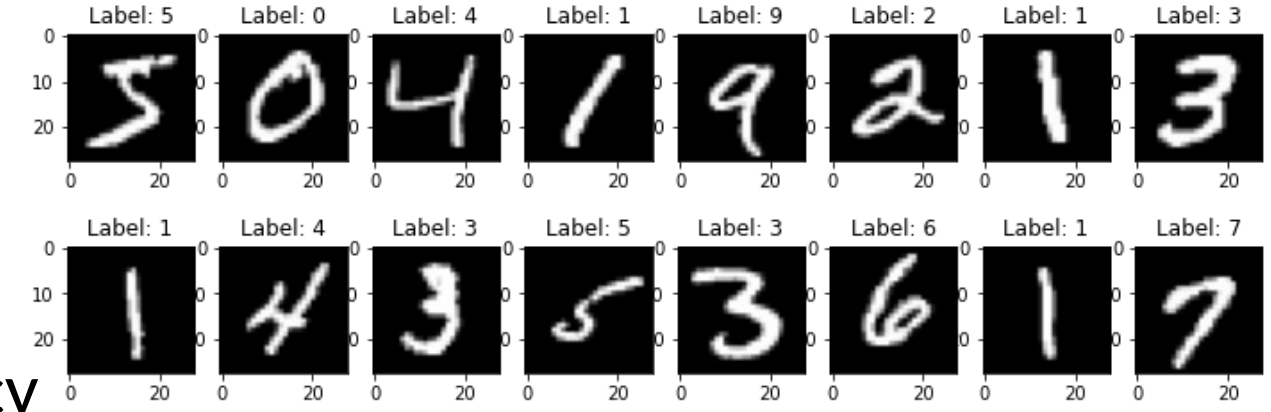
Homework 4
Neural Network

MNIST: Handwriting recognition

60,000 training images; 10,000 test images

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

Numbers 0-9 10 classes



Train a linear softmax model

Compare training and test accuracy

Functions You need to Code

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

```
def activate_matrix(m, a):
def gradient_matrix(m, a, d):
def forward_layer(l, inp):
def backward_layer(l, delta):
def update_layer(l, rate, momentum, decay):
```

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

Run tryhw4.py and answer the experiment questions.

Submit classifier.py and hw4.pdf to Canvas under Homework 4.

Important Data Structure (classifier.py)

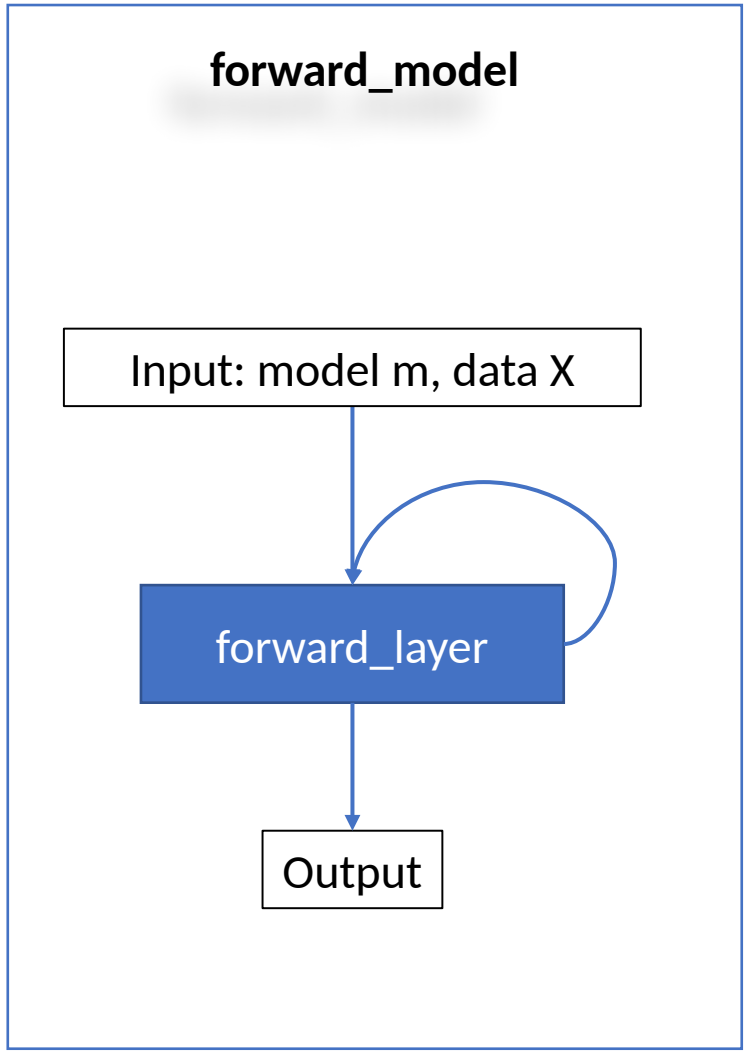
```
class layer:
    def __init__(self):
        self.inp = None         # saved input to this layer
        self.w = None           # current weights
        self.dw = None          # negative batch-sum loss gradient
        self.v = None           # momentum velocity
        self.out = None         # saved output from this layer
        self.activation = ""    # activation type

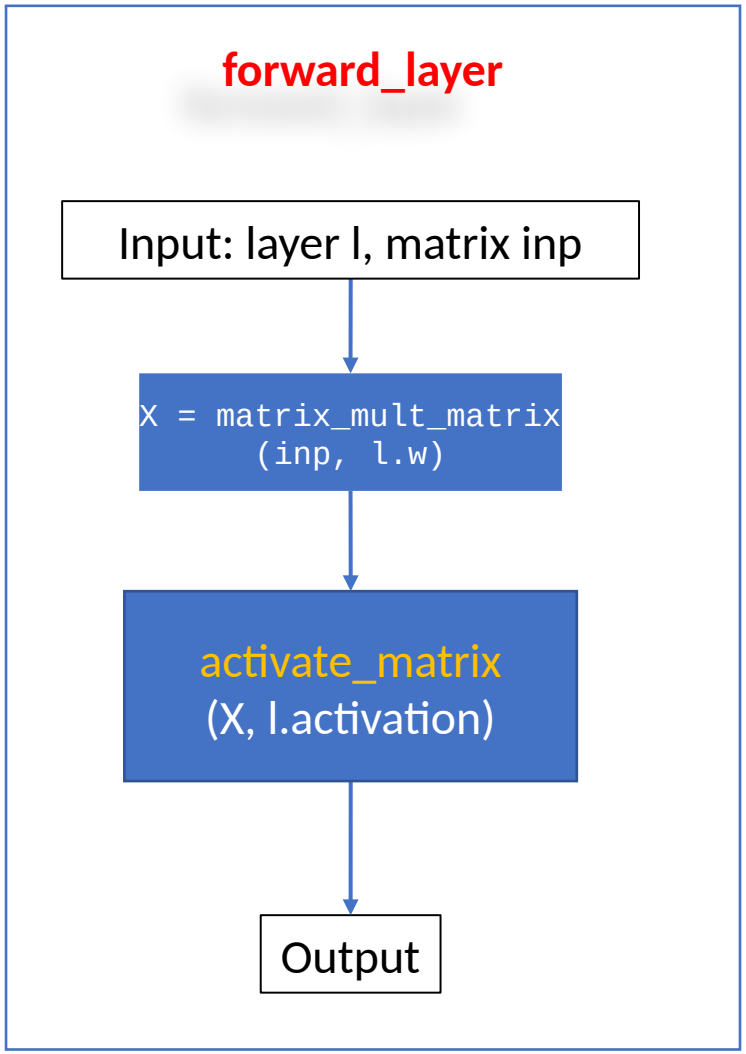
class model:
    def __init__(self):
        self.n = 0
        self.layers = []
```

Useful Matrix manipulation functions (src/matrix.py)

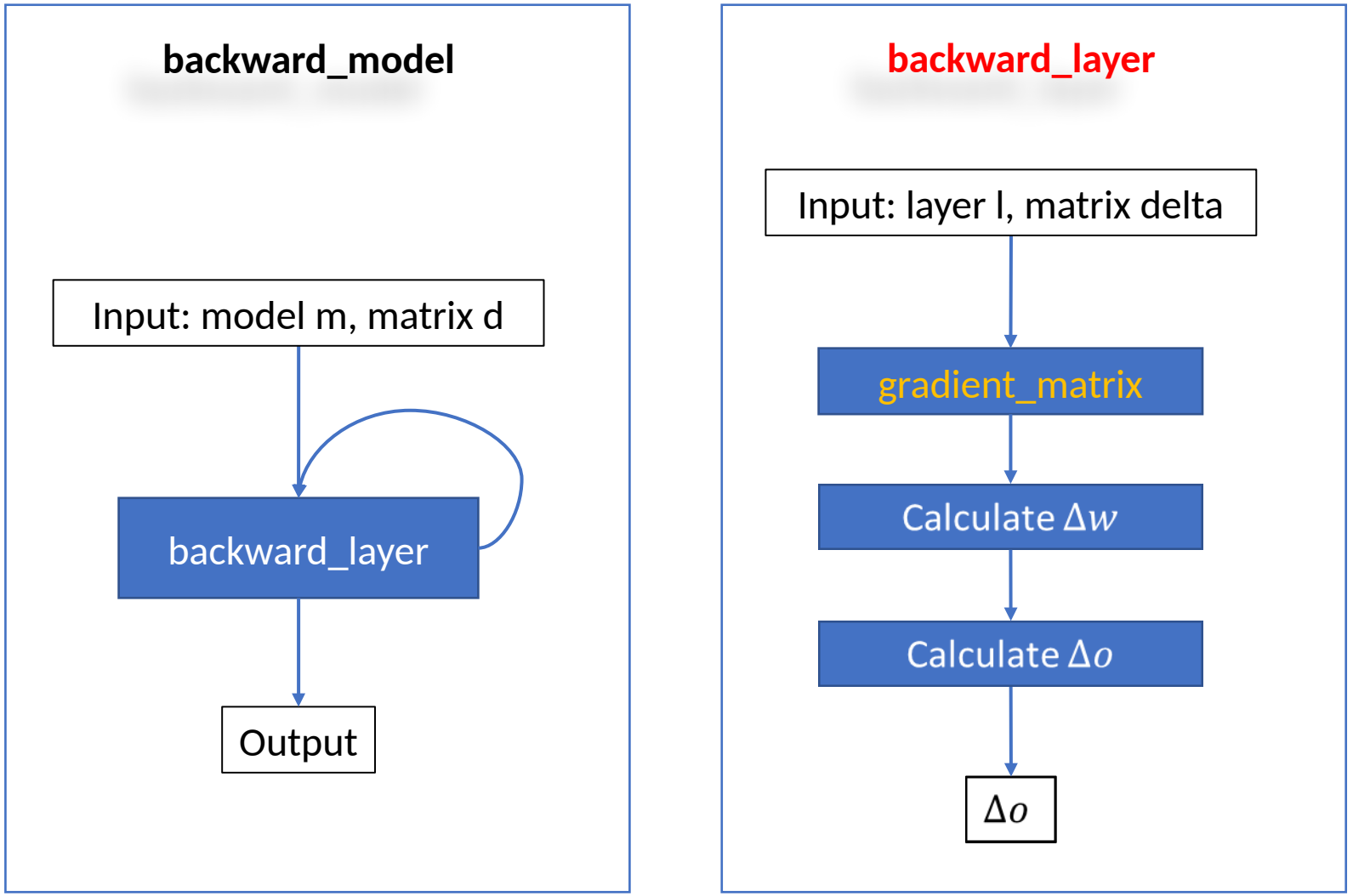
```
matrix_mult_matrix(a, b)      # matrix multiplication: a @ b
transpose_matrix(m)           # transpose
axpy_matrix(a, x, y)          # a * x + y; returns a new matrix
copy_matrix(m)                # copy a matrix
make_matrix(rows, cols)       # create a matrix
```

Forward Pass in Homework



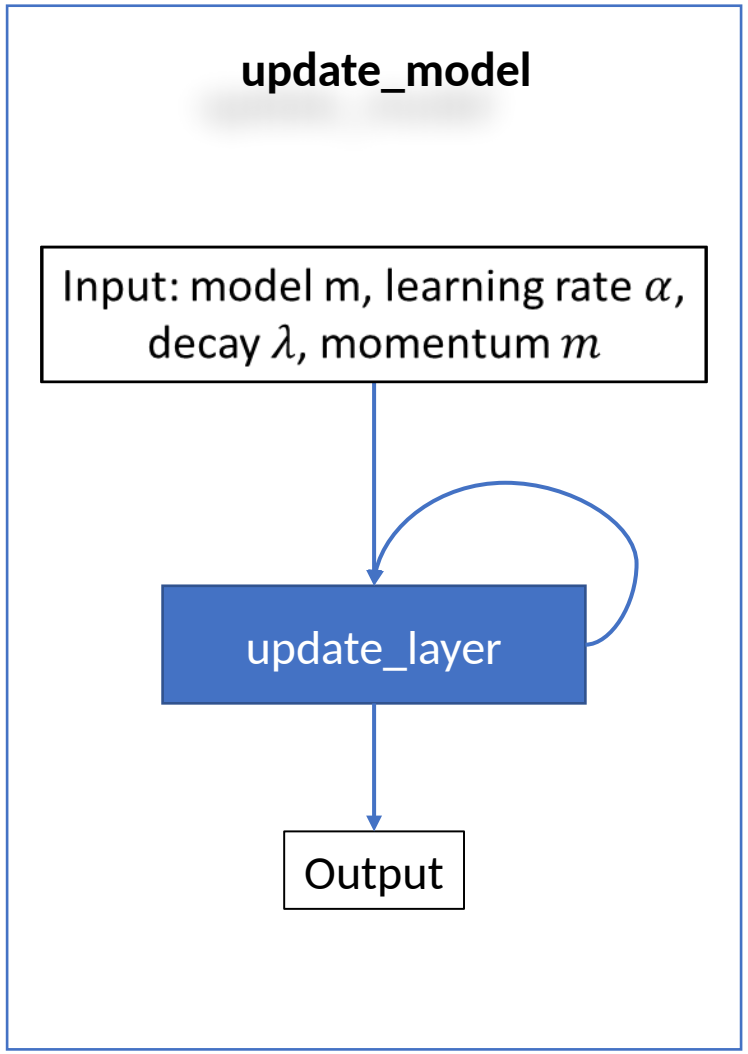


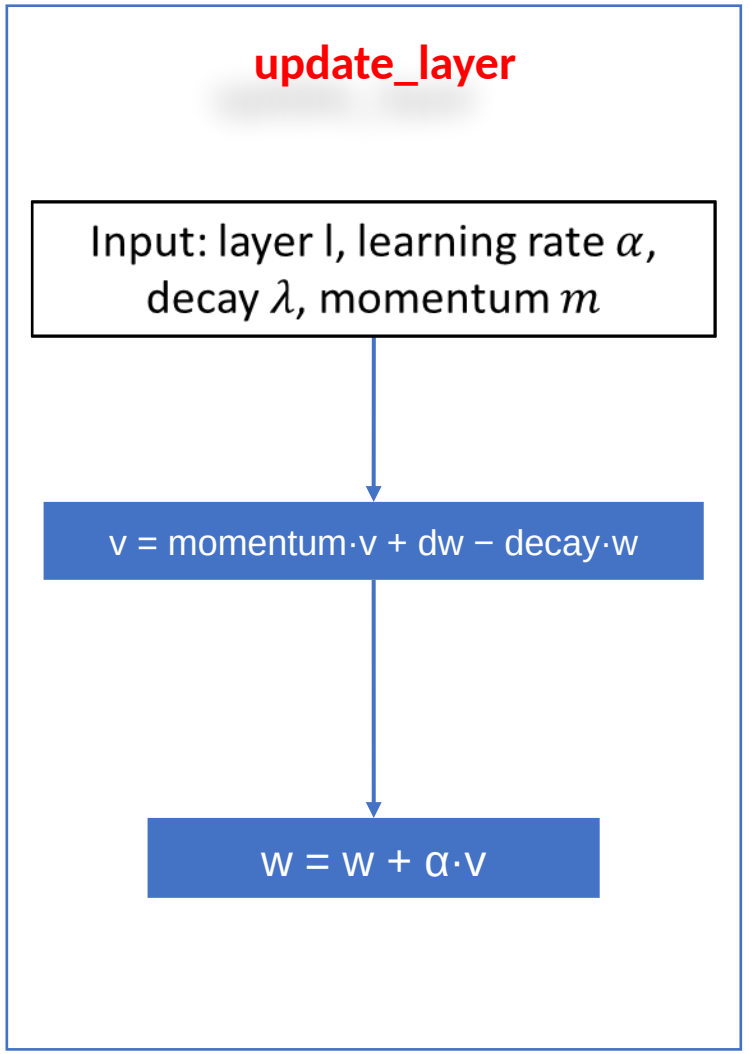
Backward Pass in Homework



Weight Update in Homework

l.dw is the negative batch-sum loss gradient.
l.v stores the momentum velocity.
During training: α = rate / batch.





TODO def activate_matrix(m, a):

Apply activation “a” to the matrix “m”

```
def activate_matrix(m, a):
    # Modify m in place; one sample per row.
    for i in range(m.rows):
        # SOFTMAX: subtract this row's max for stability.
        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 (negative slope 0.1)
            elif a == 'SOFTMAX':
                pass  # TODO: exponentiate shifted logits
        if a == 'SOFTMAX':
            pass  # TODO: normalize this row
```

TODO `def gradient_matrix(m, a, d):`

m holds activated outputs; modify d in place.
SOFTMAX uses the already combined loss signal.

```
def gradient_matrix(m, a, d):
    # m: post-activation outputs
    # d: incoming update signal, modified in place
    for i in range(m.rows):
        for j in range(m.cols):
            x = m.data[i][j]
            # TODO: multiply d.data[i][j] by local derivative
            # LOGISTIC: use x * (1 - x)
            # RELU / LRELU: use the matching slope
            # LINEAR / SOFTMAX: leave d unchanged
```

TODO `def forward_layer(l, inp):`

Given input matrix inp and layer l, calculate the output.

```
def forward_layer(l, inp):
    l.inp = inp  # save input for backpropagation

    # TODO: matrix-multiply inp by l.w, then activate
    out = make_matrix(inp.rows, l.w.cols)  # replace

    l.out = out  # save output for activation derivative
    return out
```

TODO def backward_layer(l, delta):

1. Apply the local activation derivative.
2. Store the weight-update direction in l.dw.
3. Return the signal for the previous layer.

```
def backward_layer(l, delta):
    # During training, the output signal is truth - p.

    # 1.4.1 TODO: apply activation derivative in place

    # 1.4.2 TODO: calculate and store l.dw
    # This is a negative batch-sum loss gradient.
    dw = make_matrix(l.w.rows, l.w.cols)  # replace
    l.dw = dw

    # 1.4.3 TODO: propagate to the previous layer
    dx = make_matrix(l.inp.rows, l.inp.cols)  # replace
    return dx
```

TODO def update_layer(l, rate, momentum, decay):

Update l.v and l.w using the signed direction l.dw.
Training passes effective rate = rate / batch.

```
def update_layer(l, rate, momentum, decay):
    # l.dw: negative batch-sum loss gradient
    # l.v: accumulated momentum velocity

    # TODO: combine l.dw, -decay*l.w, and momentum*l.v
    # Save the current velocity in l.v.

    # TODO: add rate*l.v to l.w
    pass

# train_model already calls update_model(rate / batch, ...)
# Keep the plus sign; do not average the gradient twice.
```

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(input, output, activation):
def forward_model(m, X):
def backward_model(m, dL):
def update_model(m, rate, momentum, decay):
def accuracy_model(m, d):
def cross_entropy_loss(y, p):
def train_model(m, d, batch, iters, rate, momentum, decay):
```

Get the Data

1. MNIST PNGs: extract into the repository root

Download the MNIST PNG version from Kaggle.

```
# From the repository root:
cd mnist-png
find train -name '*.png' > mnist.train
find test -name '*.png' > mnist.test
python ../tryhw4.py
```

2. CIFAR-10 PNGs: keep the nested cifar10/cifar10 layout

Download the CIFAR-10 PNG version from Kaggle.

```
# From the repository root:
cd cifar10
find cifar10/train -name '*.png' > cifar.train
find cifar10/test -name '*.png' > cifar.test
python ../tryhw4.py  # change script loading to cifar.*
```

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

1. Coding and data preparation
2. MNIST Experiments
    1. Linear Softmax Model (1-layer)
        1. Run baseline; report training and test accuracy
        2. Learning rate: 10^{1}, 10^{0}, 10^{-1}, 10^{-2}, 10^{-3}
        3. Decay: powers of 10 from 10^{0} to 10^{-5}
    2. Neural networks (2-layer and 3-layer)
        1. Compare hidden activations; keep SOFTMAX output
        2. Tune the learning rate
        3. Compare decay with 0; 3-layer sweep: 10^{-4} to 10^{0}
        4. 3-layer: inputs → 64 → 32 → outputs; 3000 iterations
3. Experiments for CIFAR-10
    1. 3-layer NN; tune learning rate and decay
        1. Report best train/test accuracy and all settings tried