CSE 455
SVMs and Neural Nets

Linda Shapiro
Professor of Computer Science & Engineering
Professor of Electrical Engineering

Kernel Machines

- A relatively new learning methodology (1992) derived from statistical learning theory.
- Became famous when it gave accuracy comparable to neural nets in a handwriting recognition class.
- Was introduced to computer vision researchers by Tomaso Poggio at MIT who started using it for face detection and got better results than neural nets.
- Has become very popular and widely used with packages available.

Support Vector Machines (SVM)

- Support vector machines are learning algorithms that try to find a hyperplane that separates the different classes of data the most.

- They are a specific kind of kernel machines based on two key ideas:

  - maximum margin hyperplanes

  - a kernel 'trick'

The SVM Equation

- $y_{SVM}(x_q) = \underset{c}{\text{argmax}} \sum_{i=1,m} \alpha_{i,c} K(x_i,x_q)$

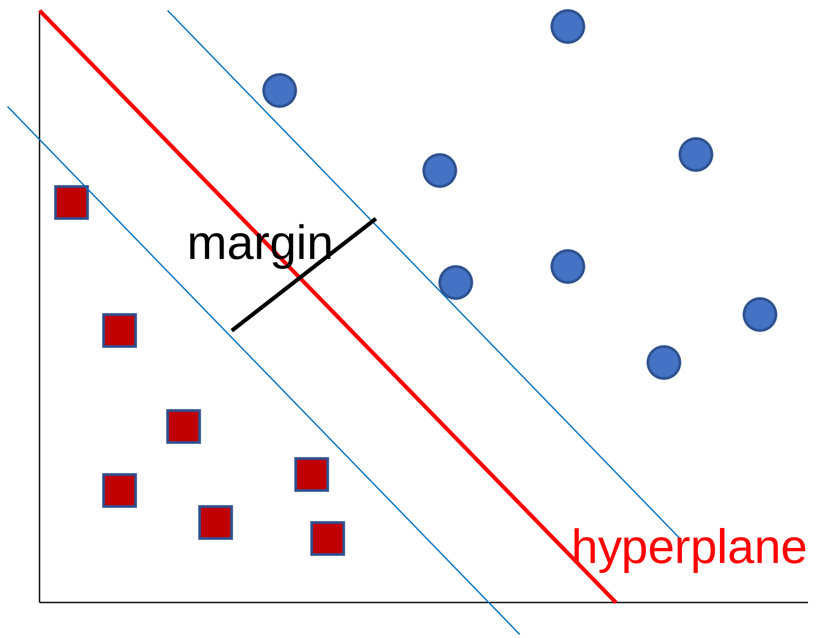
- x_q is a query or unknown object
- c indexes the classes
- there are m support vectors x_i with weights $\alpha_{i,c}$, i=1 to m for class c
- K is the kernel function that compares x_i to x_q

*** This is for multiple class SVMs with support vectors for every class; we'll see a simpler equation for 2 class.

Maximal Margin (2 class problem)

In 2D space,
a hyperplane is
a line.

In 3D space,
it is a plane.

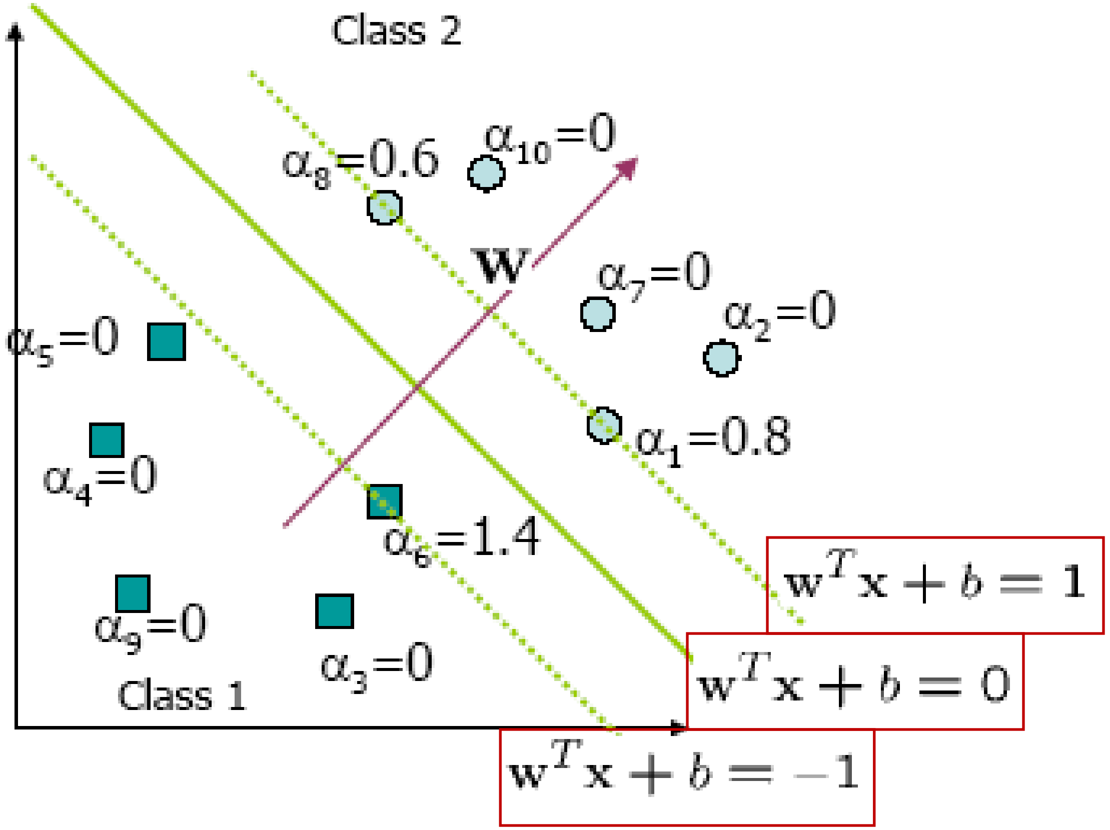


Find the hyperplane with maximal margin for all the points. This originates an optimization problem which has a unique solution.

Support Vectors

- The weights α_i associated with data points are zero, except for those points closest to the separator.
- The points with nonzero weights are called the support vectors (because they hold up the separating plane).
- Because there are many fewer support vectors than total data points, the number of parameters defining the optimal separator is small.

A Geometric Interpretation



Kernels

- A kernel is just a similarity function. It takes 2 inputs and decides how similar they are.

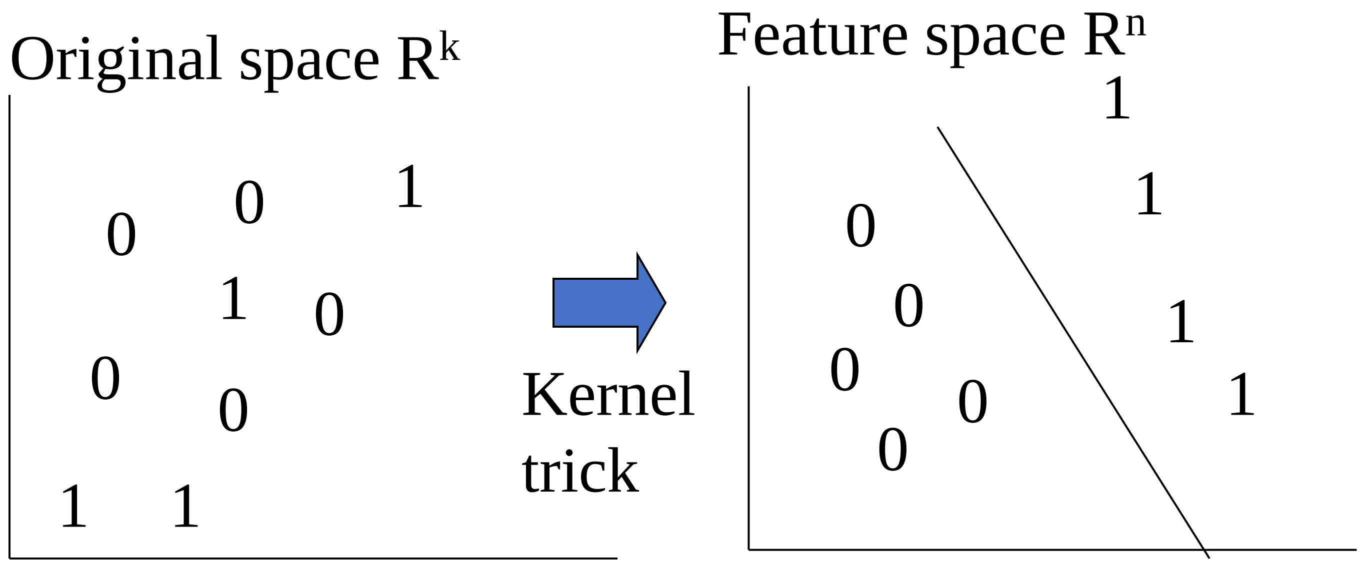
- Kernels offer an alternative to standard feature vectors. Instead of using a bunch of features, you define a single kernel to decide the similarity between two objects.

Kernels and SVMs

- Under some conditions, every kernel function can be expressed as a dot product in a (possibly infinite dimensional) feature space (Mercer’s theorem)
- SVM machine learning can be expressed in terms of dot products.
- So SVM machines can use kernels instead of feature vectors.

The Kernel Trick

The SVM algorithm implicitly maps the original data to a feature space of possibly infinite dimension in which data (which is not separable in the original space) becomes separable in the feature space.

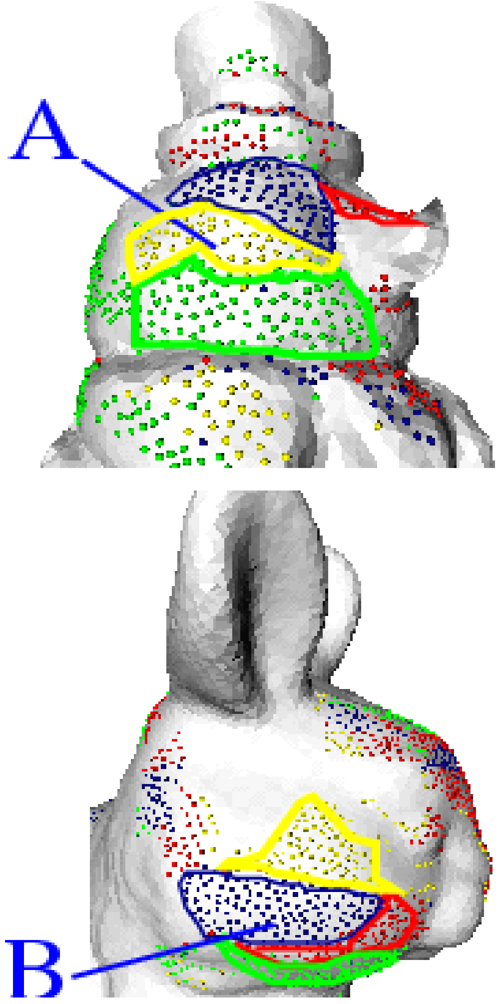


Kernel Functions

- The kernel function is designed by the developer of the SVM.
- It is applied to pairs of input data to evaluate dot products in some corresponding feature space.
- Kernels can be all sorts of functions including polynomials and exponentials.
- Simplest is just the plain dot product: xi•xj
- The polynomial kernel $K(xi,xj) = (xi \bullet xj + 1)^p$, where p is a tunable parameter.

Kernel Function used in our 3D Computer Vision Work

- $k(A,B) = \exp(-\theta^2_{AB}/\sigma^2)$
- A and B are shape descriptors (big vectors).
- θ is the angle between these vectors.
- σ^2 is the "width" of the kernel.



What does SVM learning solve?

- The SVM is looking for the best separating plane in its alternate space.

- It solves a quadratic programming optimization problem

$$\underset{\alpha}{argmax} \ \sum_j \alpha_j \ - 1/2 \sum_{j,k} \alpha_j \alpha_k y_j y_k (\mathbf{x}_j \bullet \mathbf{x}_k)$$

subject to $\alpha_j > 0$ and $\Sigma \alpha_j y_j = 0$.

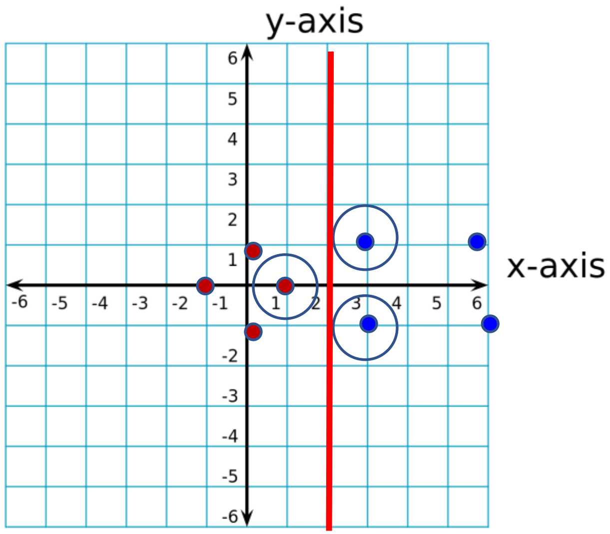
- The equation for the separator for these optimal α_j is

$$h(\mathbf{x}) = \text{sign}(\sum_j \alpha_j y_j (x \bullet x_j) - b)$$

Simple Example of Classification

- $K(A,B) = A \bullet B$
- known positive class points {(3,1),(3,-1),(6,1),(6,-1)}
- known negative class points {(1,0),(0,1),(0,-1),(-1,0)}
- support vectors: s = {(1,0),(3,1),(3,-1)} with weights α = -3.5, .75, .75
- classifier equation: $f(x) = \text{sign}(\Sigma_i [\alpha_i * K(s_i, x)] - b)$

b=2



$f(1,1) = \text{sign}(\Sigma_i\ \alpha_i\ s_i \bullet (1,1) - 2)$

$= \text{sign}(.75*(3,1) \bullet (1,1) + .75*(3,-1) \bullet (1,1) + (-3.5)*(1,0) \bullet (1,1) - 2)$

$= \text{sign}(1 - 2) = \text{sign}(-1) = -$ negative class

CORRECT

Time taken to build model: 0.15 seconds

Correctly Classified Instances	319	83.5079 %
Incorrectly Classified Instances	63	16.4921 %
Kappa statistic	0.6685	
Mean absolute error	0.1649	
Root mean squared error	0.4061	
Relative absolute error	33.0372 %	
Root relative squared error	81.1136 %	
Total Number of Instances	382	

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.722	0.056	0.925	0.722	0.811	0.833	cal
	0.944	0.278	0.78	0.944	0.854	0.833	dor
W Avg.	0.835	0.17	0.851	0.835	0.833	0.833	

=== Confusion Matrix ===

```
  a   b   <-- classified as
 135  52 |   a = cal
  11 184 |   b = dor
```

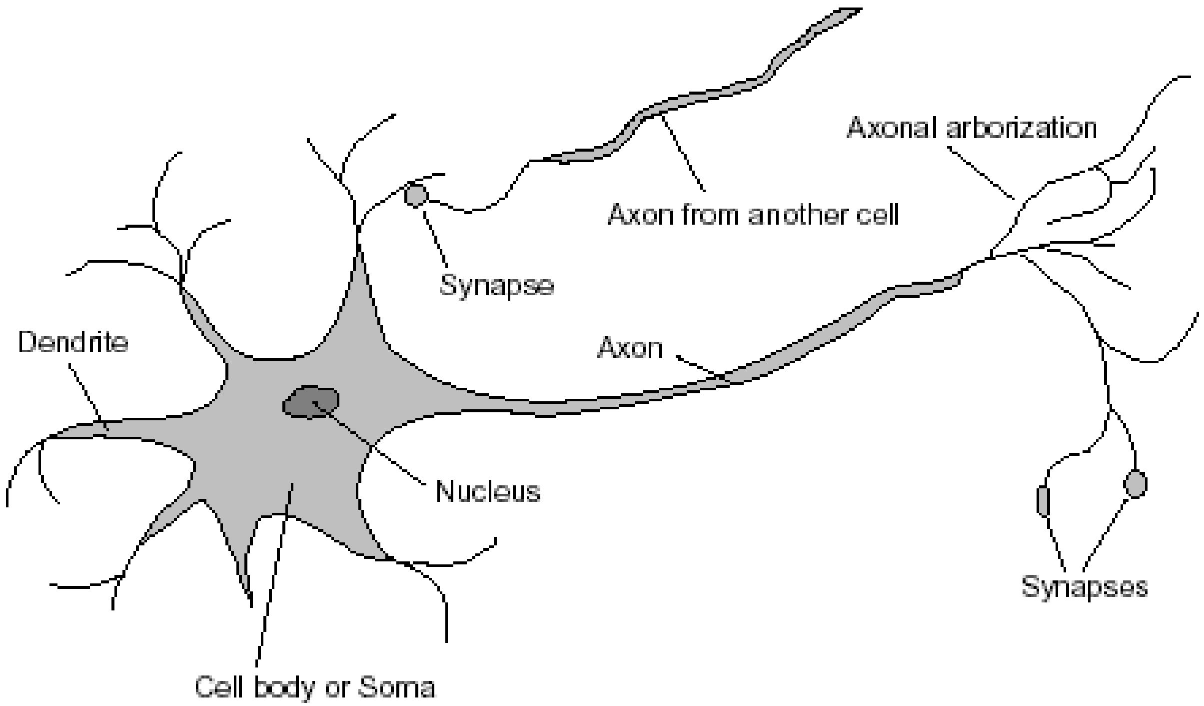
Neural Net Learning

- Motivated by studies of the brain.
- A network of “artificial neurons” that learns a function.
- Doesn’t have clear decision rules like decision trees, but highly successful in many different applications. (e.g. face detection)
- We use them frequently in our research.
- I’ll be using algorithms from

http://www.cs.mtu.edu/~nilufer/classes/cs4811/2016-spring/lecture-slides/cs4811-neural-net-algorithms.pdf

Brains

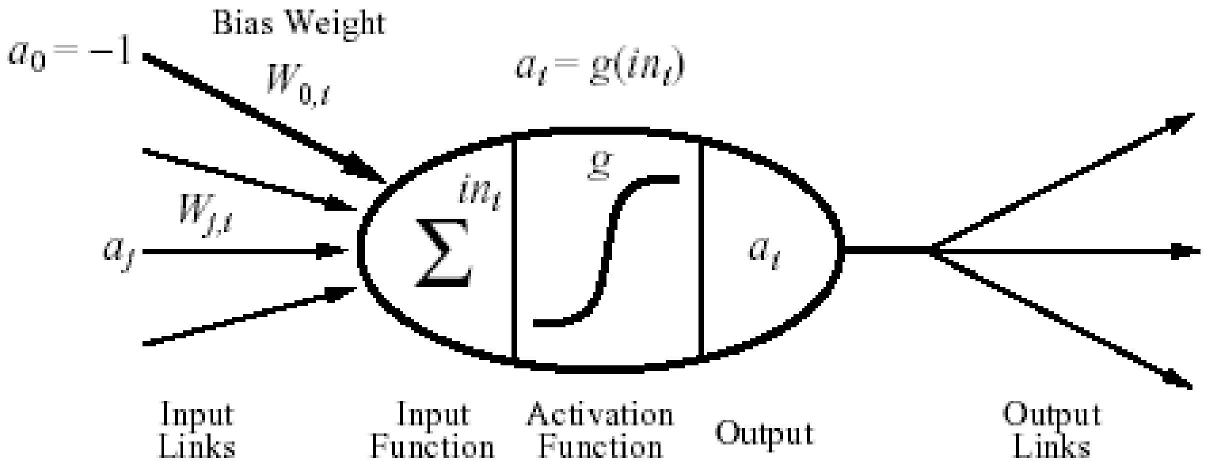
10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



McCulloch–Pitts "unit"

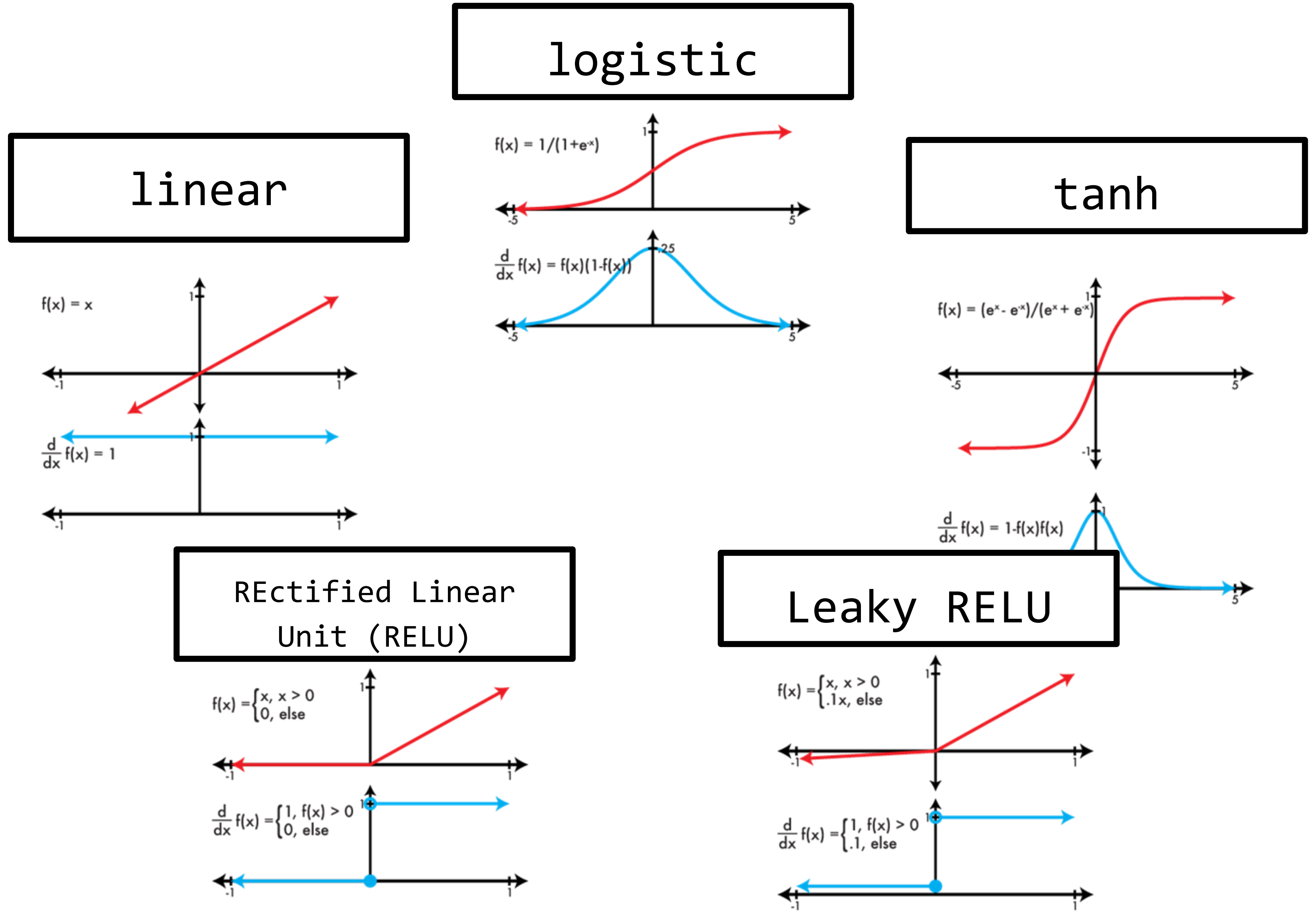
Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$

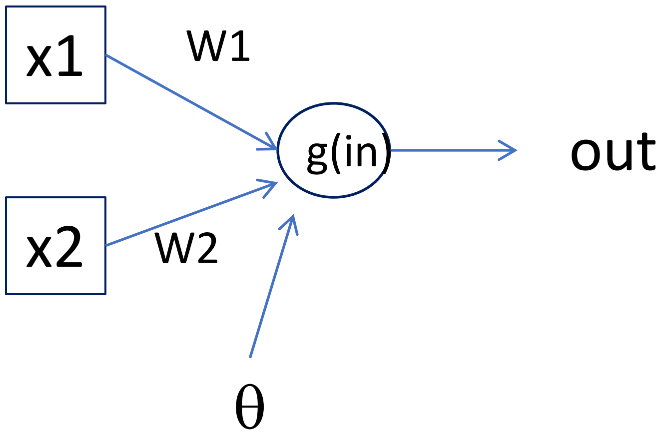


A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Common activation functions φ



Simple Feed-Forward Perceptrons



$in = (\sum W_j x_j) + \theta$

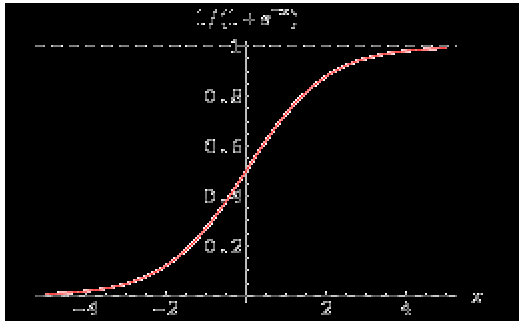
$out = g[in]$

g is the activation function

It can be a step function:
g(x) = 1 if x >=0 and 0 (or -1) else.

It can be a sigmoid function:
g(x) = 1/(1+exp(-x)).

The sigmoid function is differentiable and can be used in a gradient descent algorithm to update the weights.



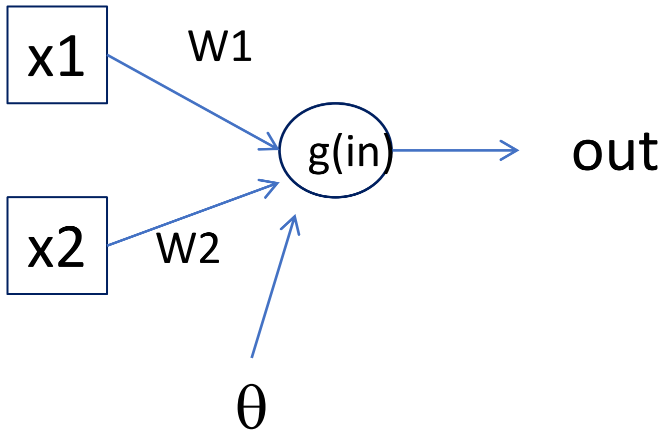
and other things...

Gradient Descent

takes steps proportional to the **negative** of the gradient of a function to find its local minimum

- Let **X** be the inputs, y the class, **W** the weights
- in = $\sum W_j x_j$
- Err = y – g(in)
- E = ½ Err^2 is the squared error to minimize
- $\partial E/\partial W_j$ = Err * $\partial Err/\partial W_j$ = Err * $\partial/\partial W_j$(g(in))(-1)
- = -Err * g'(in) * x_j
- The update is W_j <- W_j + α * Err * g'(in) * x_j
- α is called the learning rate.

Simple Feed-Forward Perceptrons




```
repeat
  for each e in examples do
    in = (∑ W_j x_j) + θ
    Err = y[e] – g[in]
    W_j = W_j + α Err g'(in) x_j[e]
until done
```

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

Note1: when g is a step function, the g'(in) is removed.

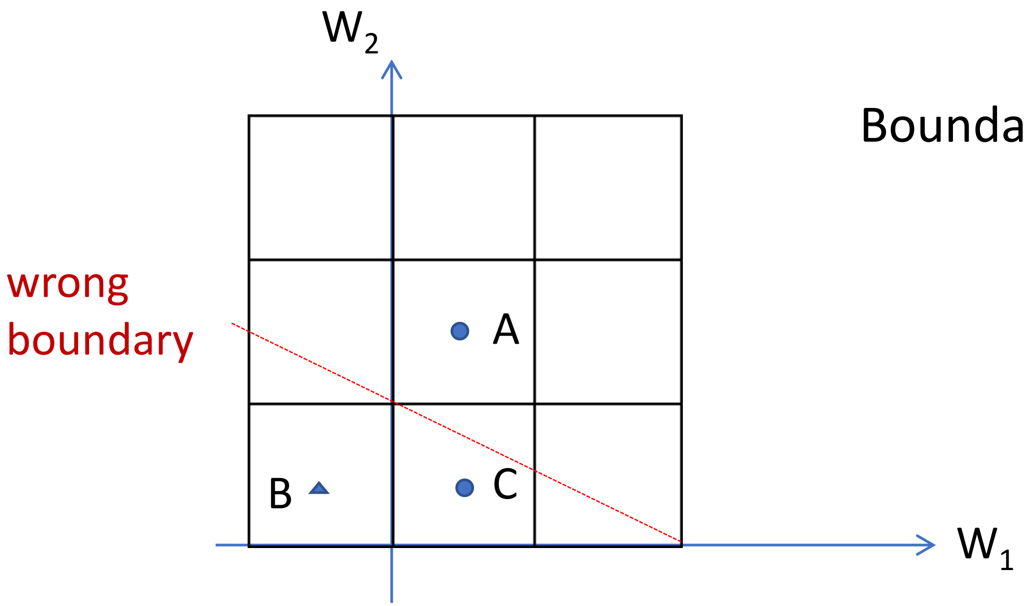
Note2: later in back propagation, Err * g'(in) will be called Δ

We'll let g(x) = 1 if x >=0 else -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Learning

Examples:
A=[(.5,1.5),+1],
B=[(-.5,.5),-1],
C=[(.5,.5),+1]
Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

repeat
  for each e in examples do
    in = $(\sum W_j x_j) + \theta$
    Err = y[e] – g[in]
    $W_j = W_j + \alpha$ Err $g'(in)$ $x_j[e]$
until done

A=[(.5,1.5),+1]
in = .5(1) + (1.5)(2) -2 = 1.5
g(in) = 1; Err = 0; NO CHANGE

B=[(-.5,.5),-1]
In = (-.5)(1) + (.5)(2) -2 = -1.5
g(in) = -1; Err = 0; NO CHANGE

C=[(.5,.5),+1]
in = (.5)(1) + (.5)(2) – 2 = -.5
g(in) = -1; Err = 1-(-1)=2

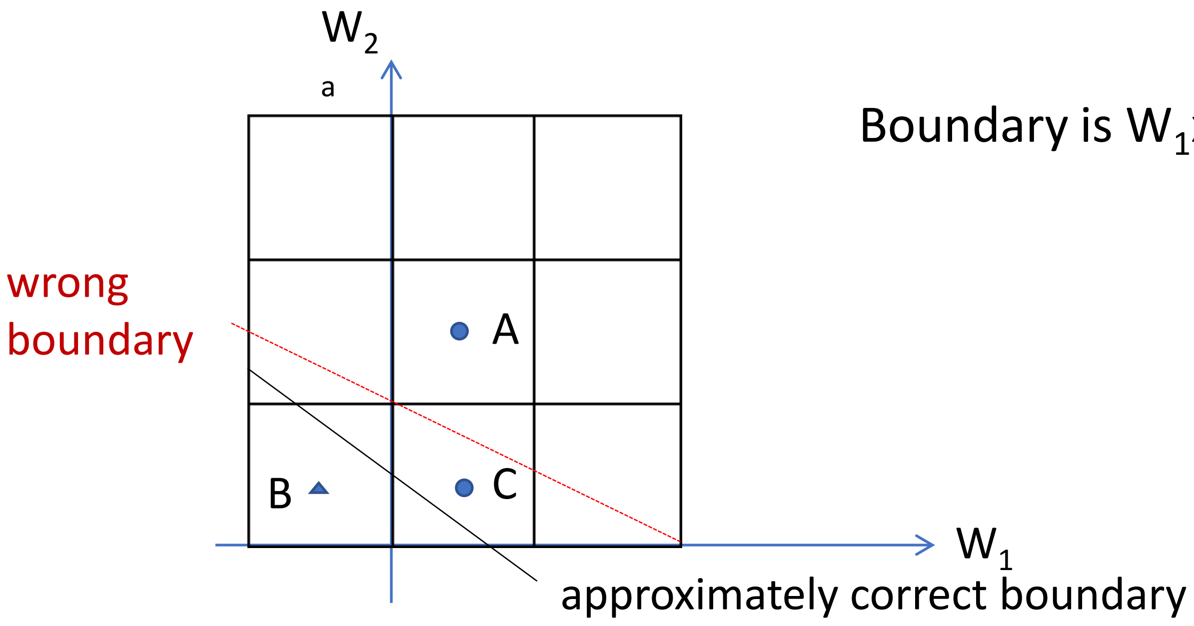
Let α=.5

W1 <- W1 + .5(2) (.5)   leaving out g'
  <- 1 + 1(.5) = 1.5
W2 <- W2 + .5(2) (.5)
  <- 2 + 1(.5) = 2.5
θ <- θ + .5(+1 – (-1))
θ <- -2 + .5(2) = -1

Graphically

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$



Boundary is $W_1x_1 + W_2x_2 + \theta = 0$

Back Propagation

- Simple single layer networks with feed forward learning were not powerful enough.
- Could only produce simple linear classifiers.
- More powerful networks have multiple hidden layers.
- The learning algorithm is called back propagation, because it computes the error at the end and propagates it back through the weights of the network to the beginning.

The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
  $w_{i,j} \leftarrow$ a small random number
repeat
  for each example (**x**,**y**) **in** *examples* **do**
    /* Propagate the inputs forward to compute the outputs. */
    for each node i in the input layer **do**   // Simply copy the input values.
      $a_i \leftarrow x_i$
    for $l = 2$ to L **do**   // Feed the values forward.
      for each node j in layer l **do**
        $in_j \leftarrow \sum_i w_{i,j}\, a_i$
        $a_j \leftarrow g(in_j)$
    for each node j in the output layer **do**   // Compute the error at the output.
      $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
    /* Propagate the deltas backward from output layer to input layer */
    for $l = L - 1$ **to** 1 **do**
      for each node i in layer l **do**
        $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$ // "Blame" a node as much as its weig
    /* Update every weight in network using deltas. */
    for each weight $w_{i,j}$ in *network* **do**
      $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.
until some stopping criterion is satisfied

return *network*

Let's break it into steps.

Initialize

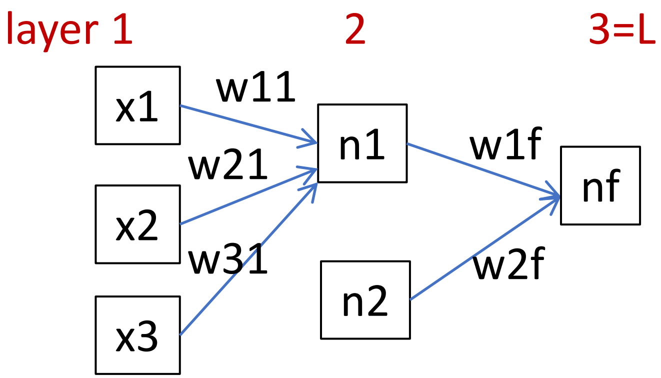
The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(*examples, network*)
returns a neural network

inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

for each weight $w_{i,j}$ in *network* do
$w_{i,j} \leftarrow$ a small random number



Forward Computation

repeat

    for each example (**x**,**y**) **in** *examples* **do**

        /* Propagate the inputs forward to compute the outputs. */

        for each node i in the input layer **do** // Simply copy the input values.

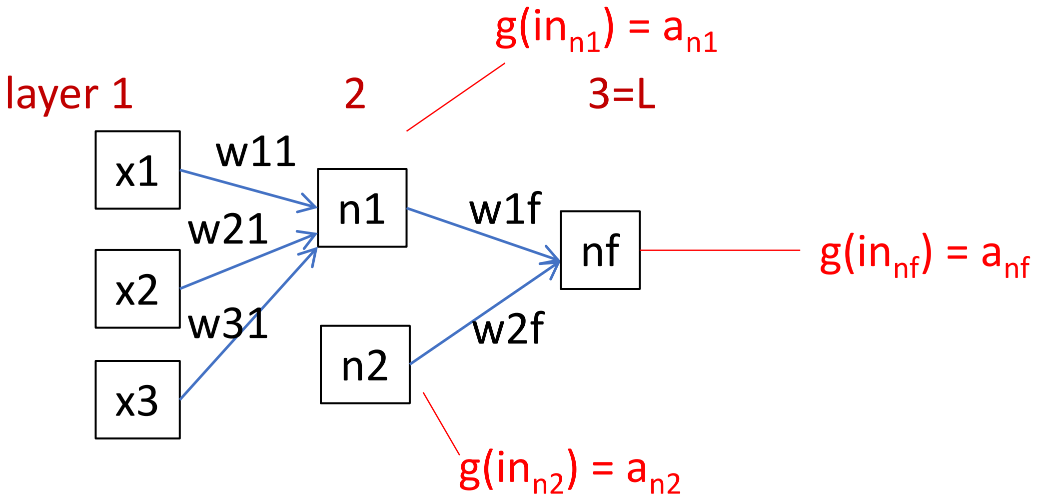
            $a_i \leftarrow x_i$

        for $l = 2$ to L **do** // Feed the values forward.

            for each node j in layer l **do**

                $in_j \leftarrow \sum_i w_{i,j}\, a_i$

                $a_j \leftarrow g(in_j)$

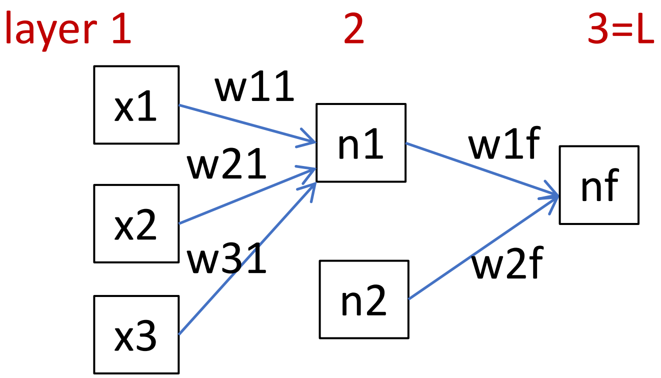


Backward Propagation 1

for each node j in the output layer **do** // Compute the error at the output.

$$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$$

- Node nf is the only node in our output layer.
- Compute the error at that node and multiply by the derivative of the weighted input sum to get the change delta.

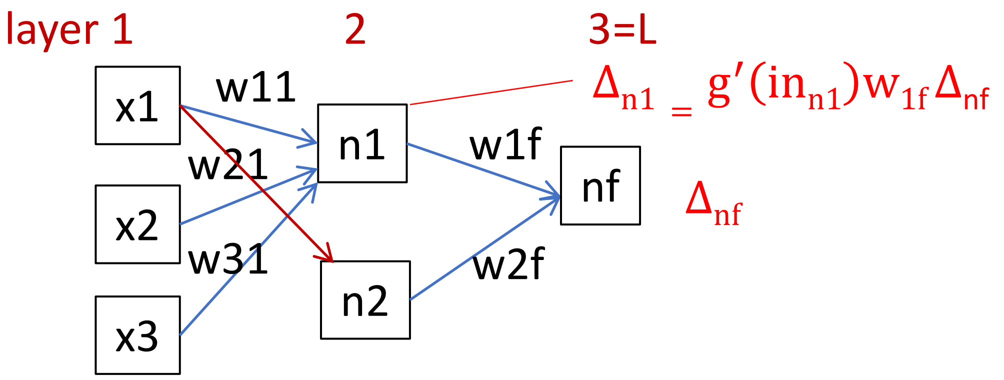


$\Delta_{nf} = g'(in_{nf}) * (y_{nf} - a_{nf})$

Backward Propagation 2

/* Propagate the deltas backward from output layer to input layer */
for $l = L - 1$ **to** 1 **do**
    for each node i in layer l **do**
        $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$ // "Blame" a node as much as its weight

- At each of the other layers, the deltas use
  - the derivative of its input sum
  - the sum of its output weights
  - the delta computed for the output error

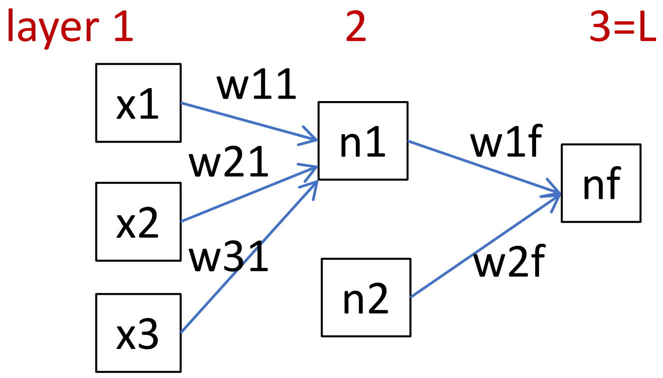


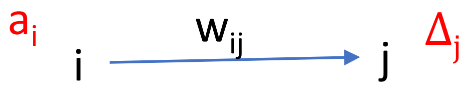
If there were two output nodes, there would be a summation.

Backward Propagation 3

/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in *network* **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$ // Adjust the weights.

Now that all the deltas are defined, the weight updates just use them.





Back Propagation Summary

- Compute delta values for the output units using observed errors.
- Starting at the output-1 layer
  - repeat
    - propagate delta values back to previous layer
    - till done with all layers
  - update weights for all layers

- This is done for all examples and multiple epochs, till convergence or enough iterations.

Time taken to build model: 16.2 seconds

Correctly Classified Instances	307	80.3665 % (did not boost)
Incorrectly Classified Instances	75	19.6335 %
Kappa statistic	0.6056	
Mean absolute error	0.1982	
Root mean squared error	0.41	
Relative absolute error	39.7113 %	
Root relative squared error	81.9006 %	
Total Number of Instances	382	

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.706	0.103	0.868	0.706	0.779	0.872	cal
	0.897	0.294	0.761	0.897	0.824	0.872	dor
W Avg.	0.804	0.2	0.814	0.804	0.802	0.872	

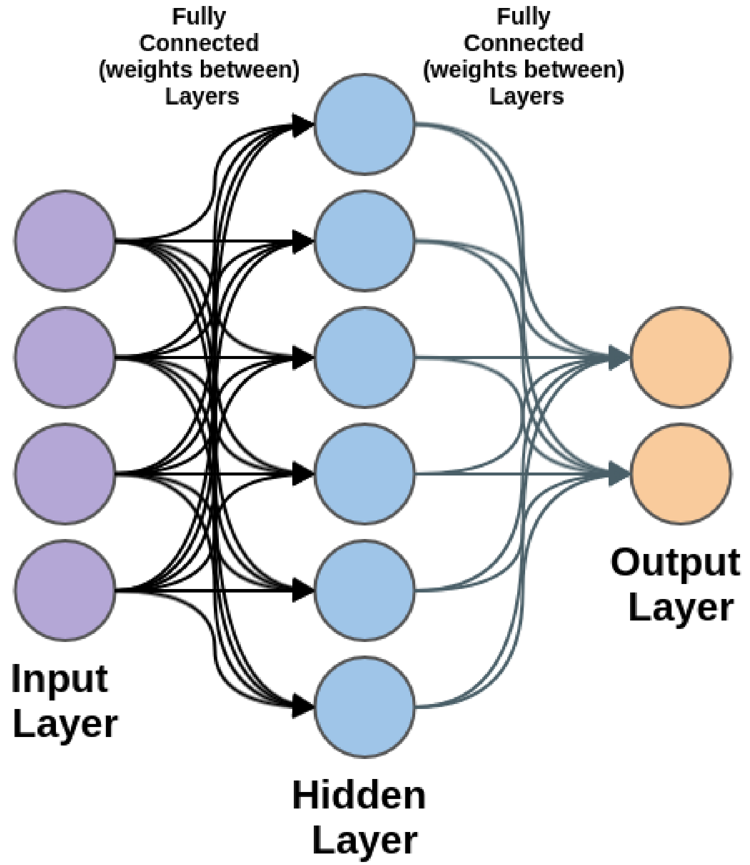
=== Confusion Matrix ===

```
   a   b   <-- classified as
 132  55 |   a = cal
  20 175 |   b = dor
```

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

Z

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

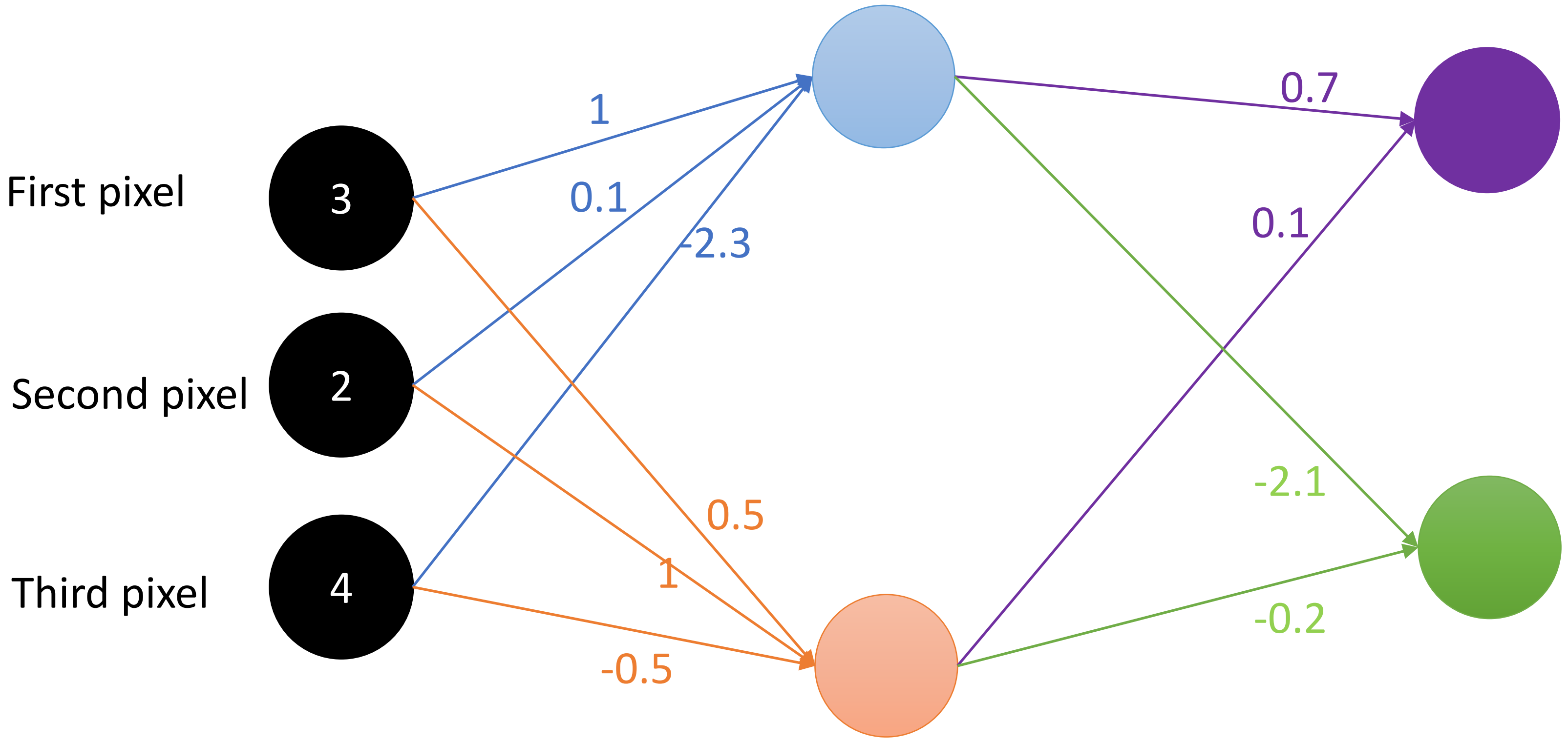
Here, we will go over a simple 2-layer neural network (no bias).

Mini-batch for Machine Learning

- We use a matrix to represent data.
- If there are 10,000 images, and each image contains 784 features, we can use a 10,000 x 784 matrix to represent the whole dataset.
- Hard to load a large dataset at once; so, we can split the dataset into smaller batches.
- For instance, in homework 5, we use batch size 128. Then, each batch contains 128 images, and the corresponding data is stored in a 128 x 784 matrix.
- Then, we can feed batches one-by-one to the ML model, and train it for each batch.

Neural Network Easy Example

Here, we use batch size of 4, and we only visualize the first sample for simplicity.



Input Layer

1-st Layer (ReLU)

Output Layer with Softmax

X_{in}

3	2	4
.	.	.
.	.	.

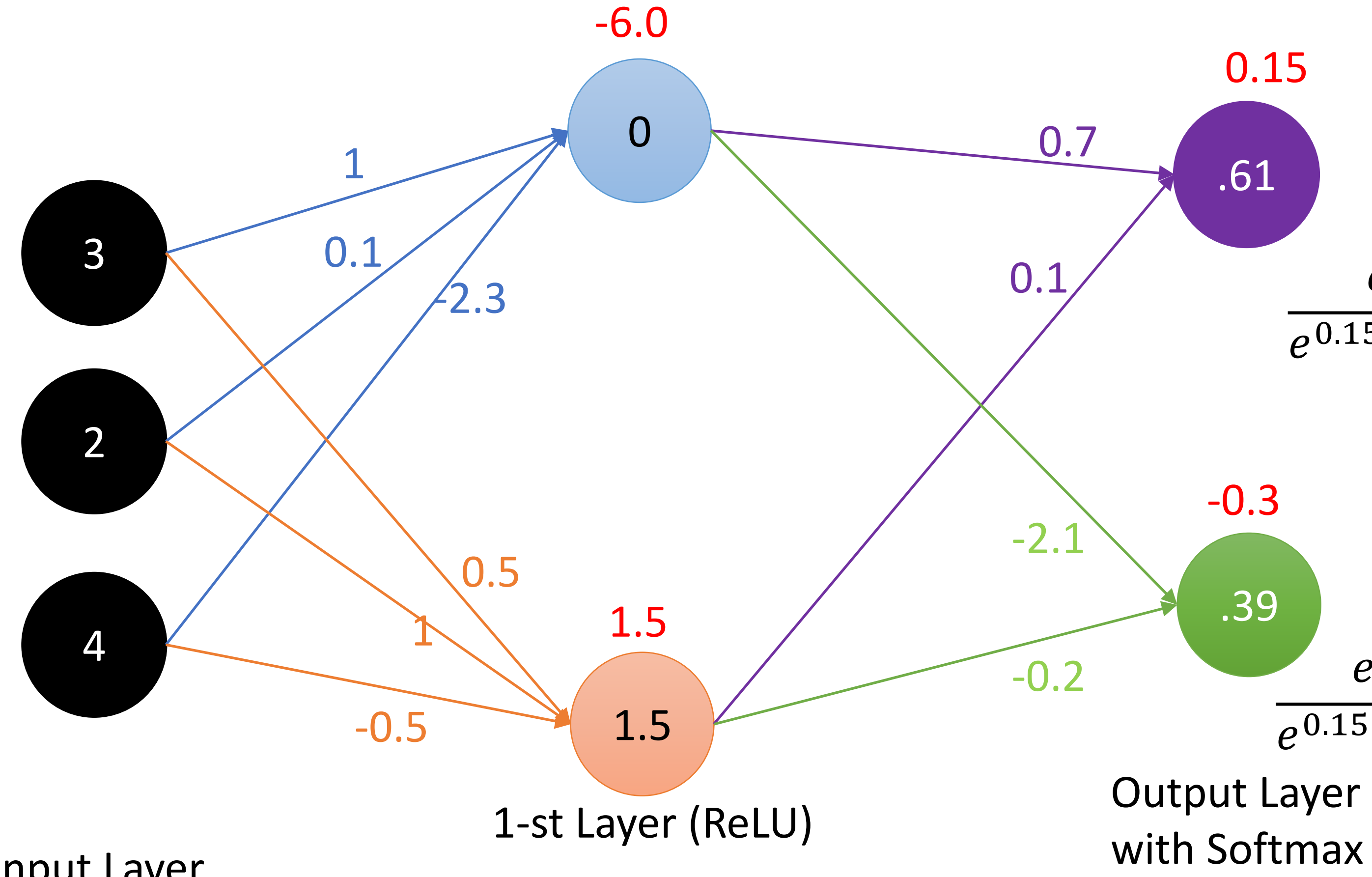
w_1

1	0.5
0.1	1
-2.3	-0.5

w_2

0.7	-2.1
0.1	-0.2

[Example] Forward Pass



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

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

Input Layer

X_{in}

3	2	4
.	.	.
.	.	.

w_1

1	0.5
0.1	1
-2.3	-0.5

o_1

0	1.5
.	.
.	.

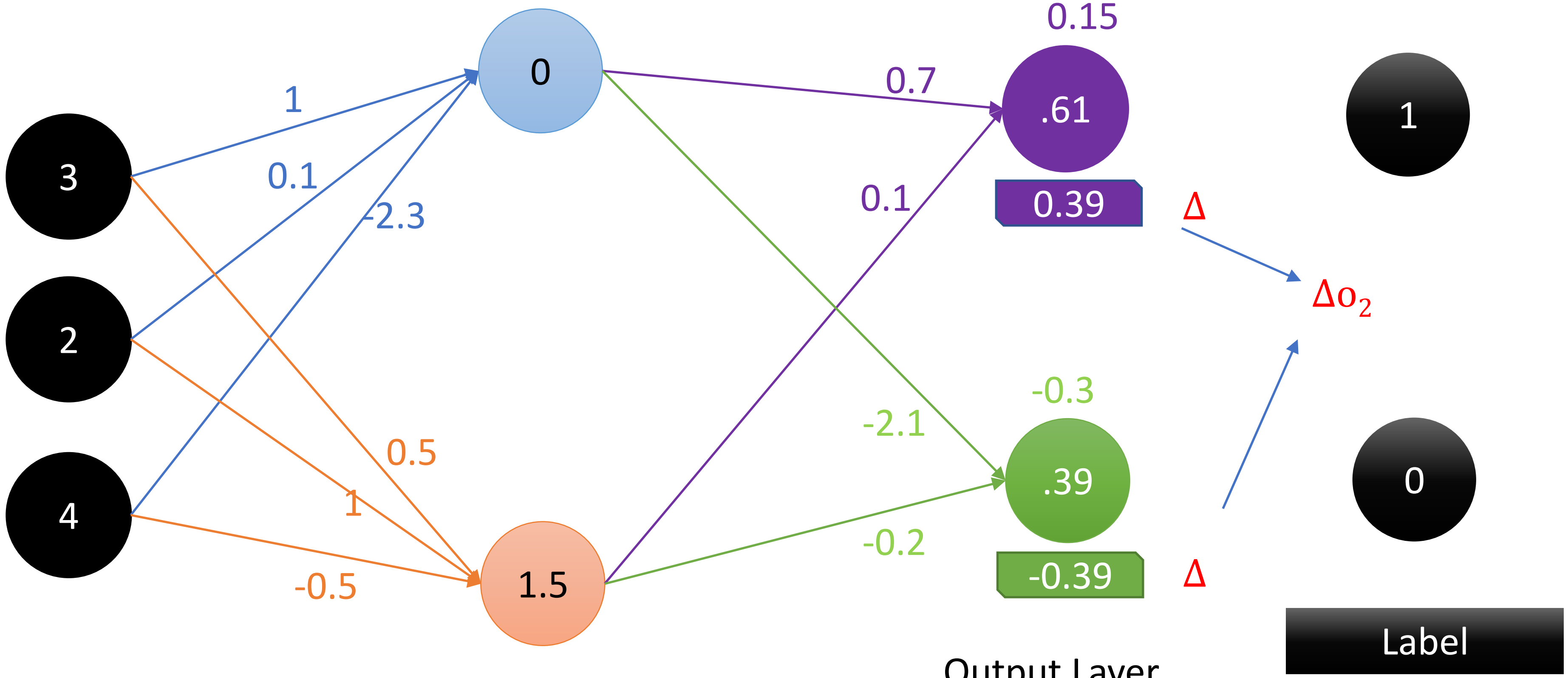
w_2

0.7	-2.1
0.1	-0.2

o_2

0.61	0.39
.	.
.	.

[Example] Ground Truth and Loss



Input Layer

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

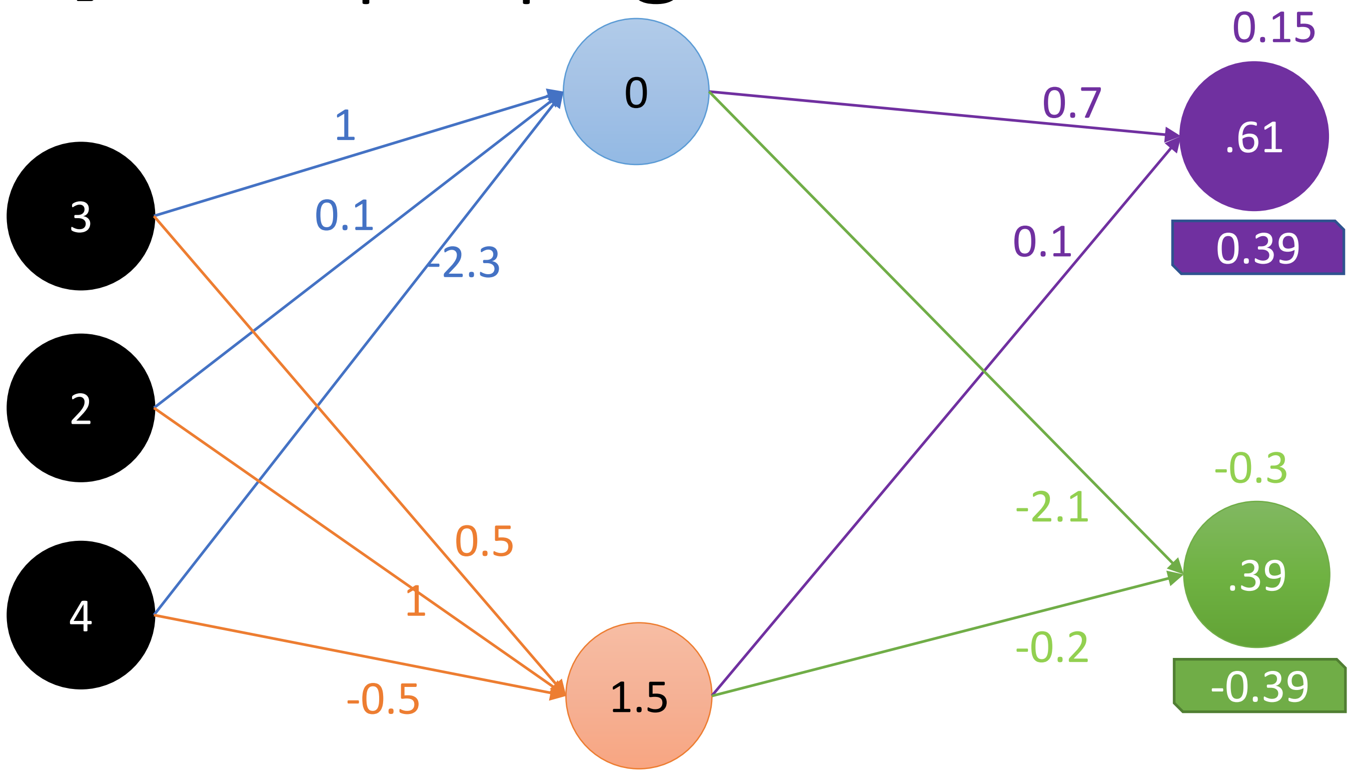
0.61	0.39
.	.
.	.

[Example] Backpropagation

"∘" represents elementwise multiplication for matrix

Assume g'(.) = 1

We use Δw_2 to represent the weight gradient for layer 2.



Input Layer

X_{in}

3	2	4
.	.	.
.	.	.

w_1

1	0.5
0.1	1
-2.3	-0.5

1-st Layer (ReLU)

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
.	.
.	.

Backpropagation [Cont.]

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

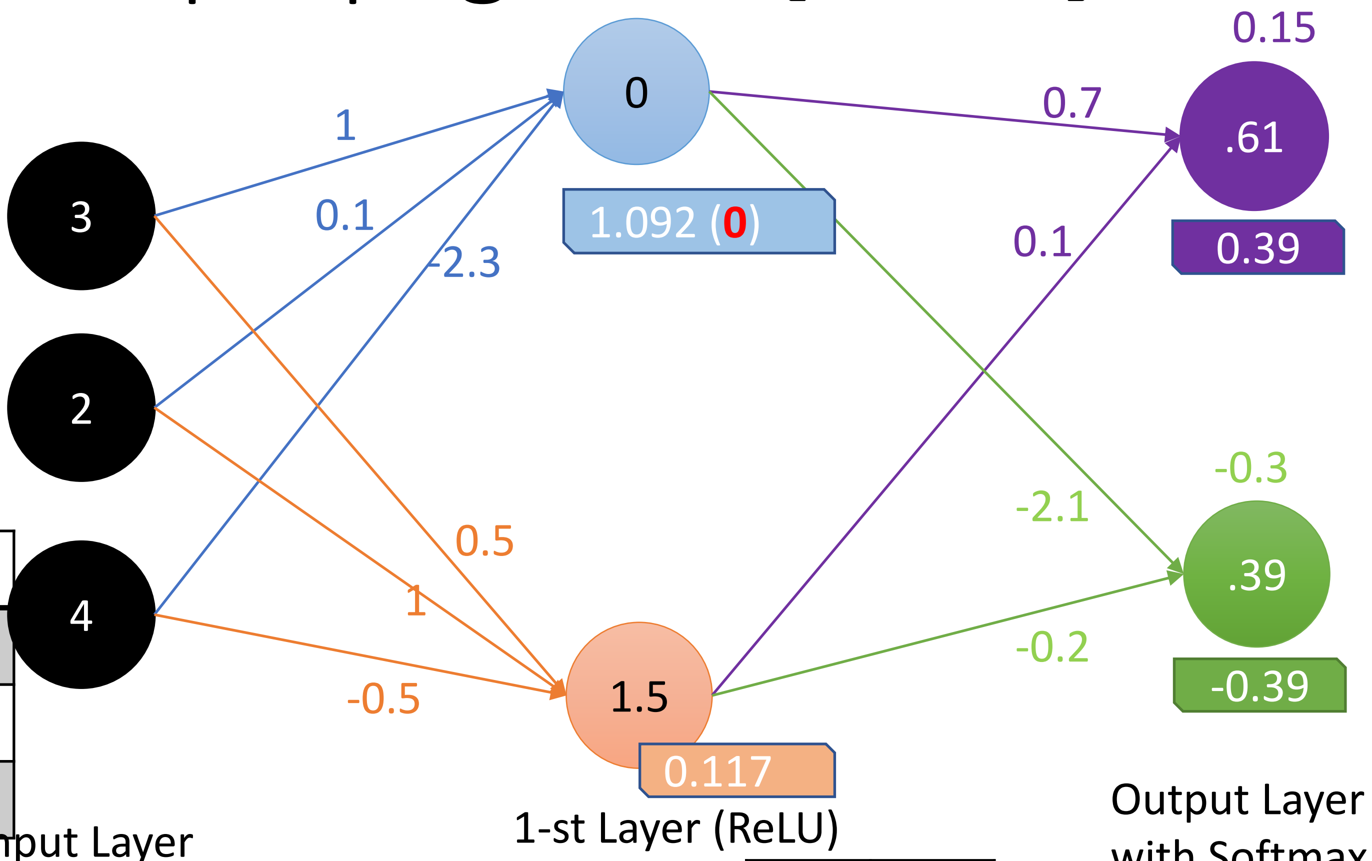
0	0.117
.	.
.	.

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

0	0.351
0	0.234
0	0.468

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

0.0585	0.117	-0.0585
.	.	.
.	.	.



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
.	.
.	.

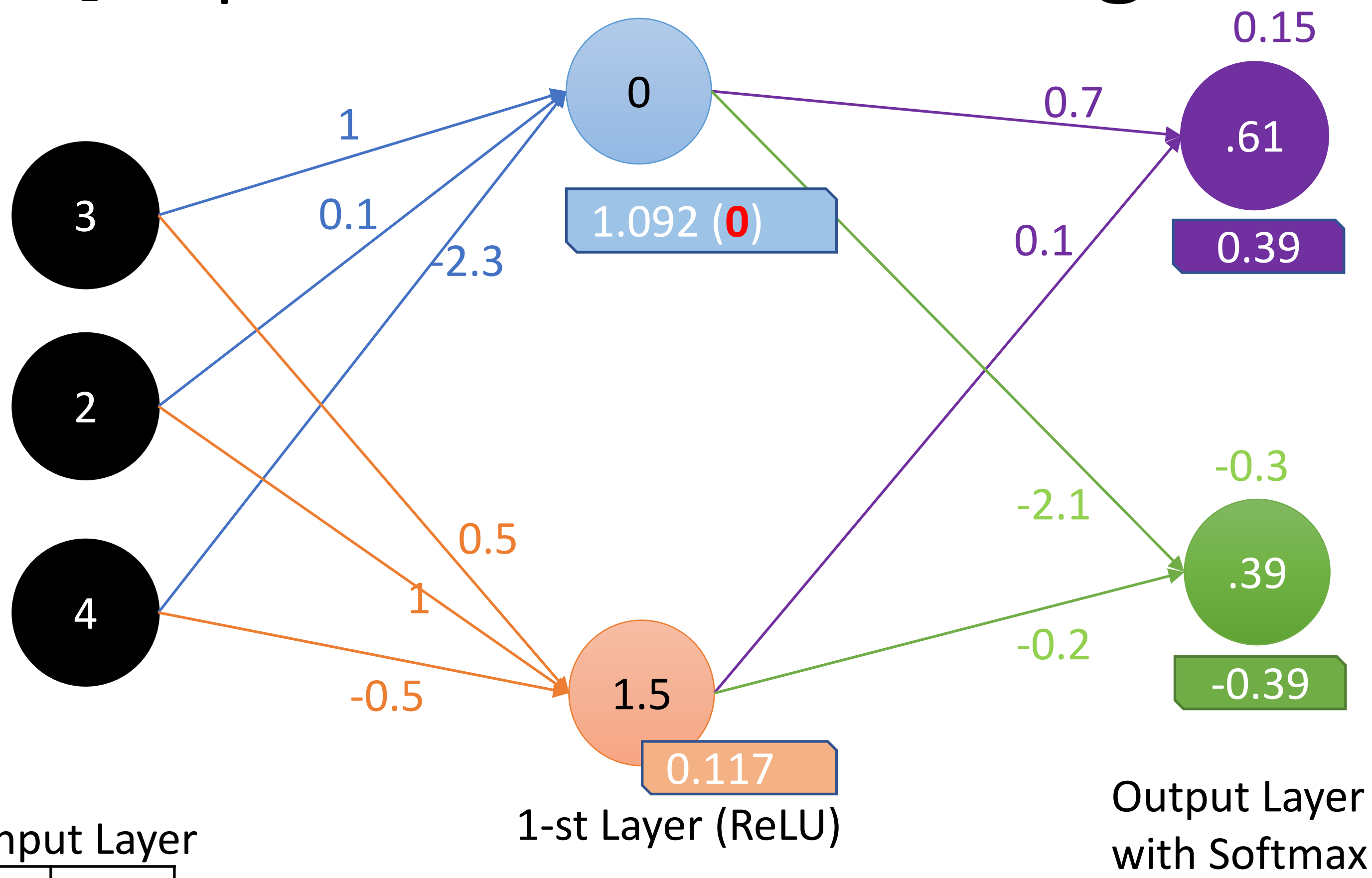
[Example] Update with Learning Rate 0.1

$\Delta w_1 = o_0^T g'(\text{o}_1) \circ \Delta \text{o}_1$

0	0.351
0	0.234
0	0.468

$w_1 = w_1 + \alpha \Delta w_1$

1	.5351
0.1	1.0234
-2.3	-0.4532



$\Delta w_2 = o_1^T g'(\text{o}_2) \circ \Delta o_2$

0	0
0.585	-0.585

$w_2 = w_2 + \alpha \Delta w_2$

0.7	-2.1
0.1585	-0.2585

X_{in}

3	2	4
.	.	.
.	.	.

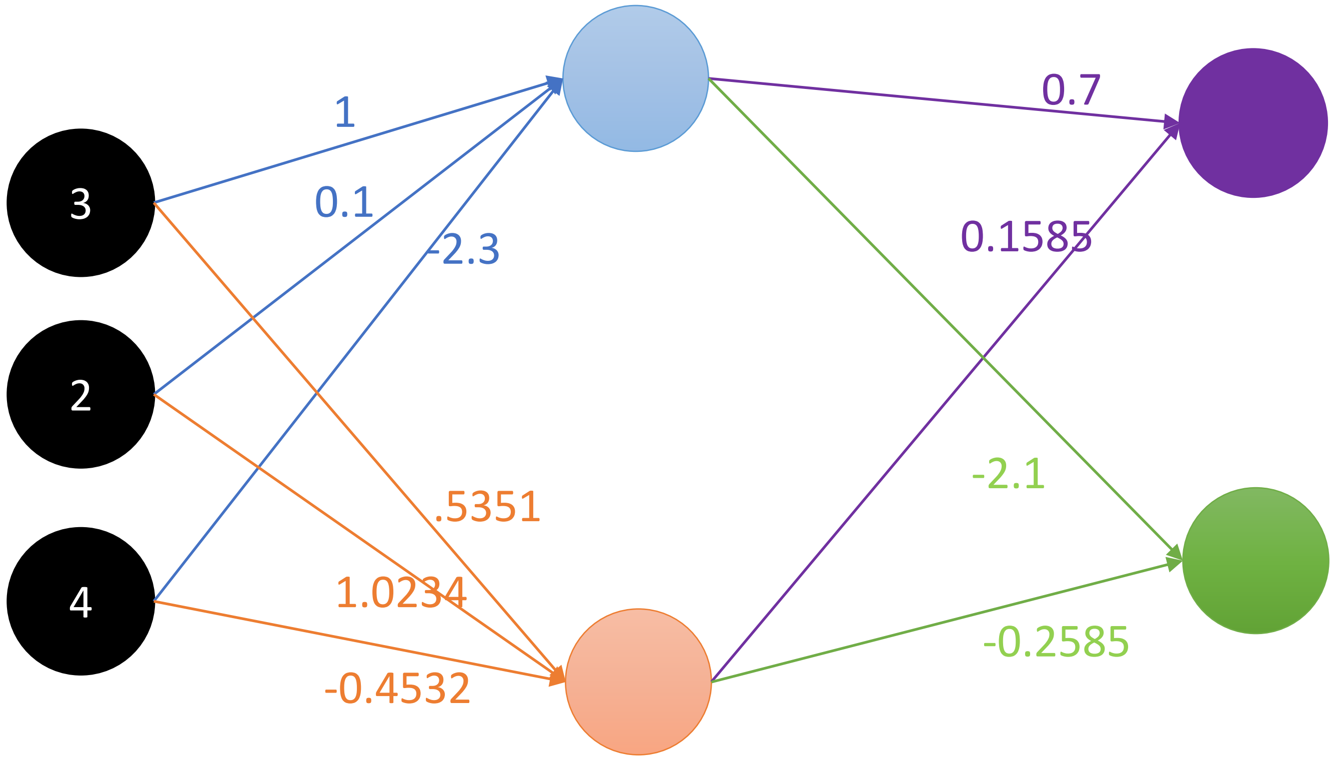
w_1

1	0.5
0.1	1
-2.3	-0.5

w_2

0.7	-2.1
0.1	-0.2

[Example] Done



Input Layer

X_{in}

3	2	4
.	.	.
.	.	.

1-st Layer (ReLU)

w_1

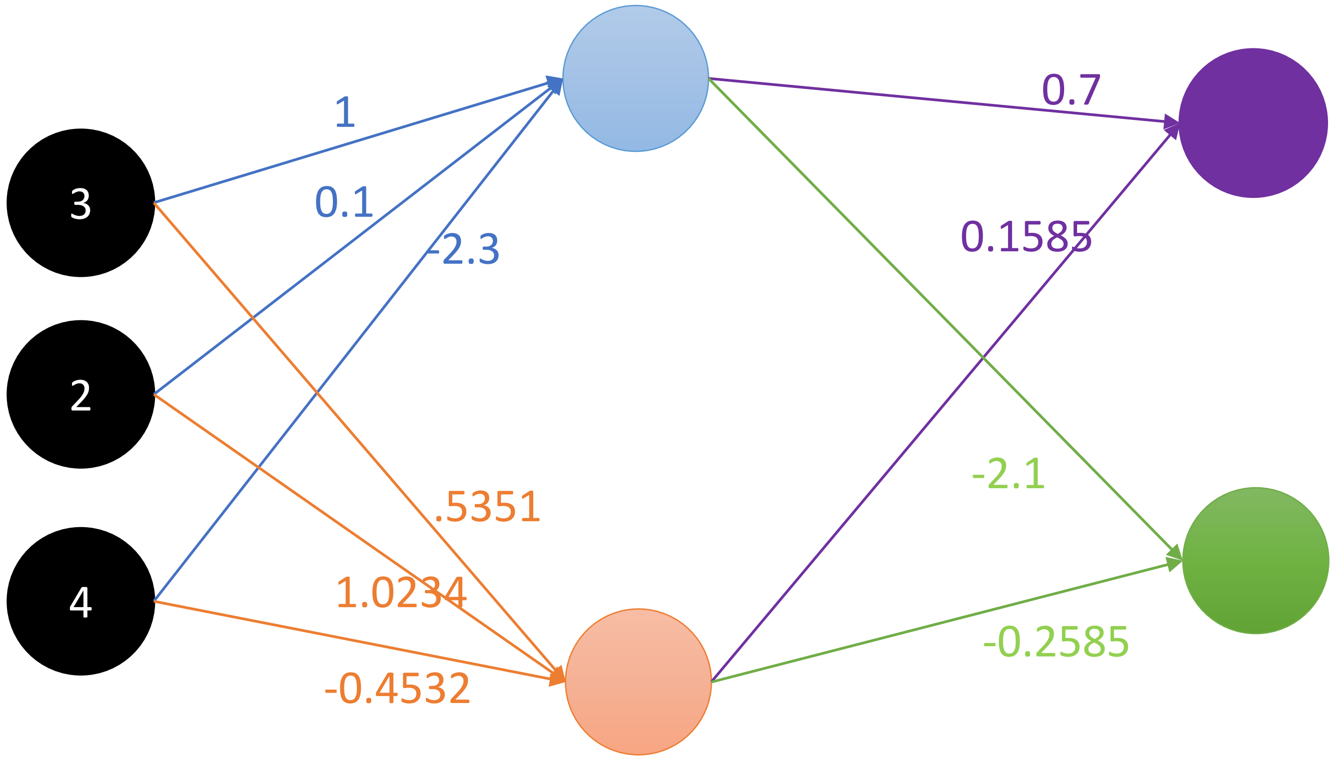
1	0.5351
0.1	1.0234
-2.3	-0.4532

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

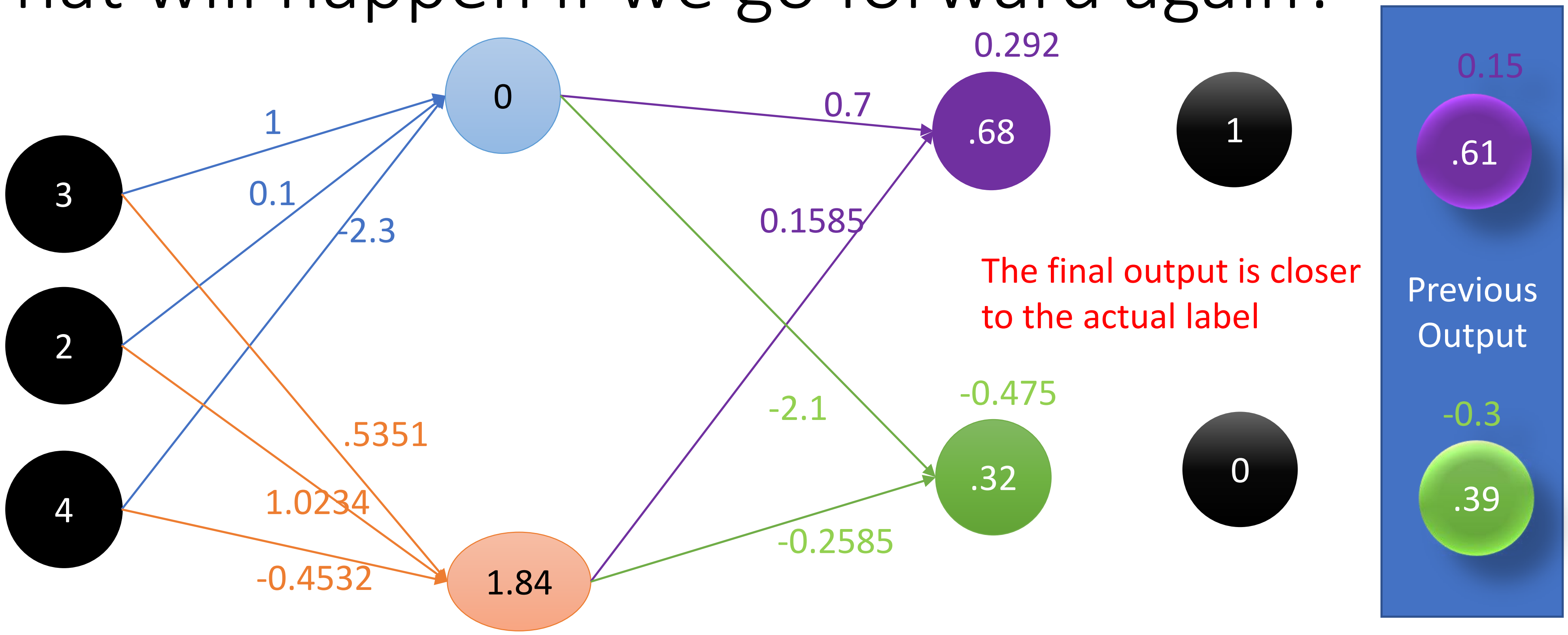
1	0.5351
0.1	1.0234
-2.3	-0.4532

Output Layer with Softmax

w_2

0.7	-2.1
0.1585	-0.2585

Think: What will happen if we go forward again?



Input Layer

1-st Layer (ReLU)

Output Layer with Softmax

Label

X_{in}

3	2	4
.	.	.
.	.	.

w_1

1	0.5351
0.1	1.0234
-2.3	-0.4532

w_2

0.7	-2.1
0.1585	-0.2585

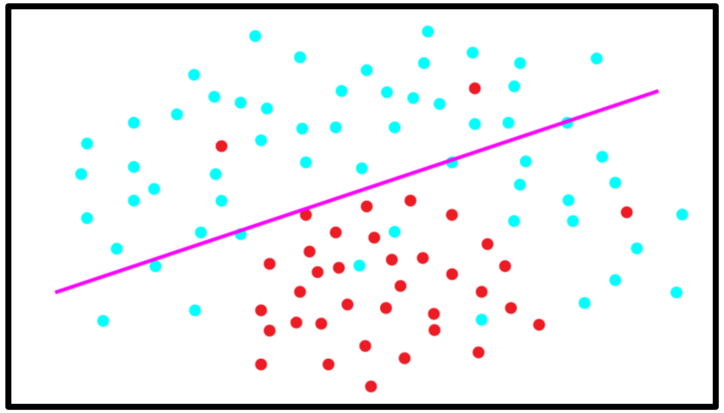
Tricks for Neural Network

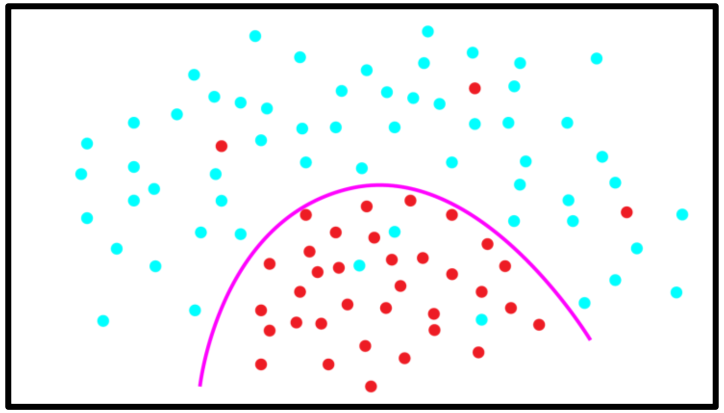
Problem: Under and Overfitting

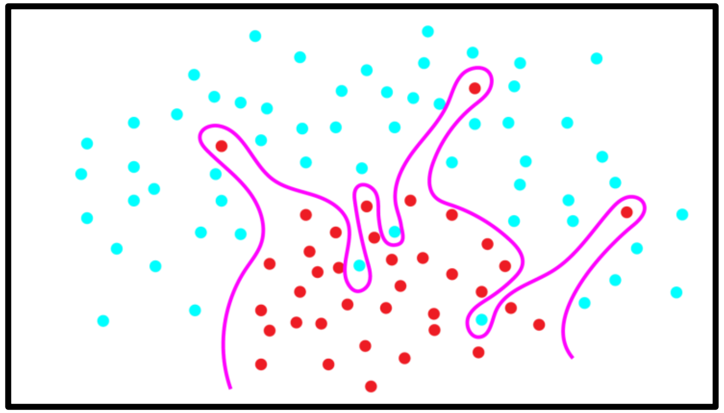
Underfitting: model not powerful enough, too much bias

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

Want the happy medium, how?







Weight decay: neural network regularization

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

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:

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

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

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:

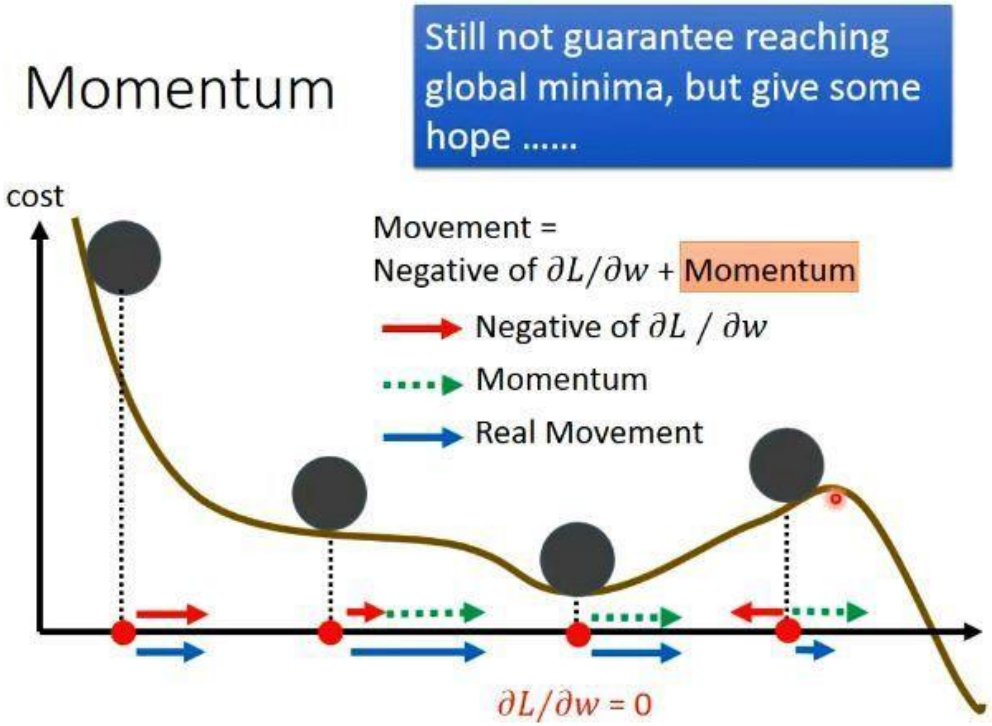
$$\Delta w_t = -\partial/\partial w_t \, L(w_t)$$

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

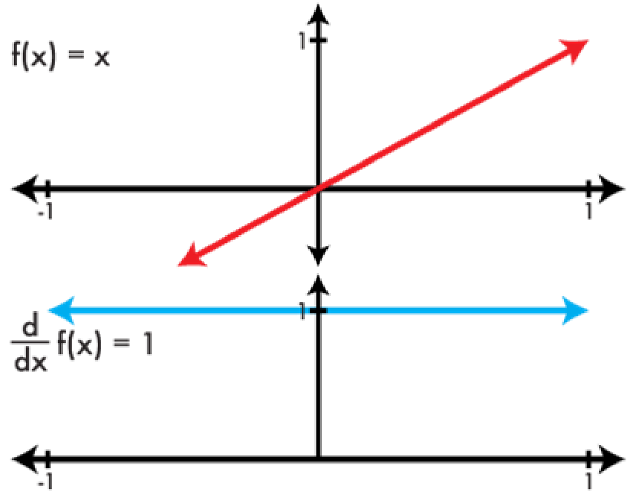
Momentum

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

Learning rate

Activations

Linear Activation



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

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

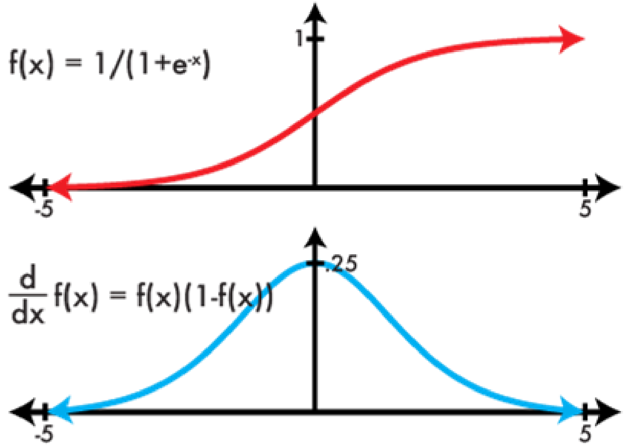
$$f(X) = g(g(Xw_1)w_2) = Xw_1w_2 = Xw$$

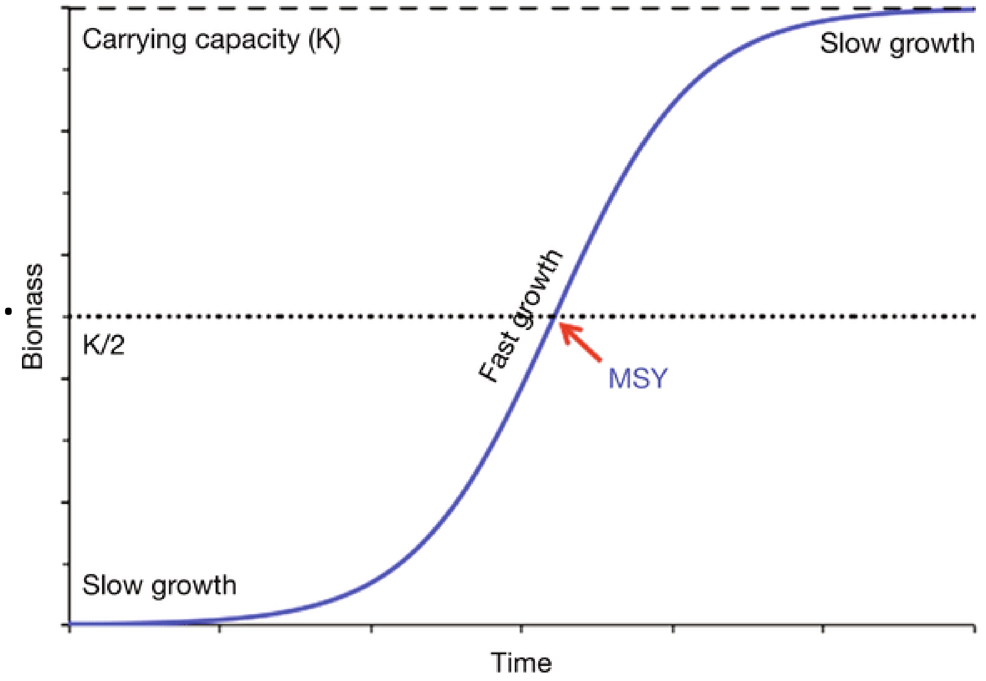
- Not so great, need Non-Linear activations to learn more complex data distribution.

Logistic Activation

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

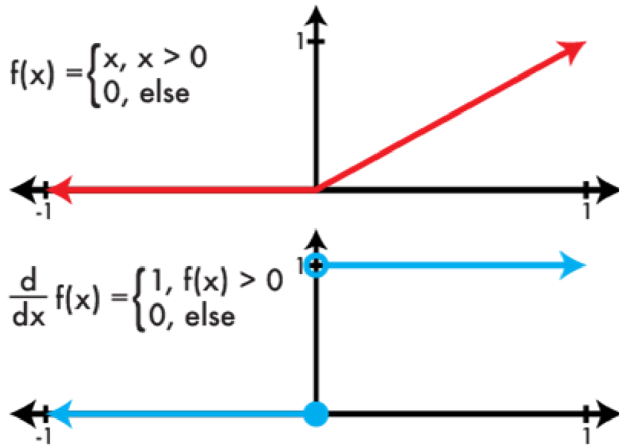
- Aka Sigmoid function (S-shape)
- Used in Logistic regression.
- The result is in range (0, 1),
- It can represent probability.
- A special case of logistic growth (population model).



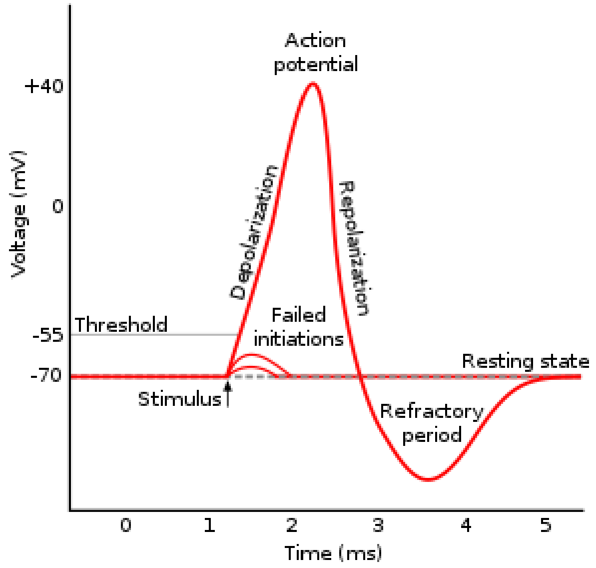


ReLU Activation

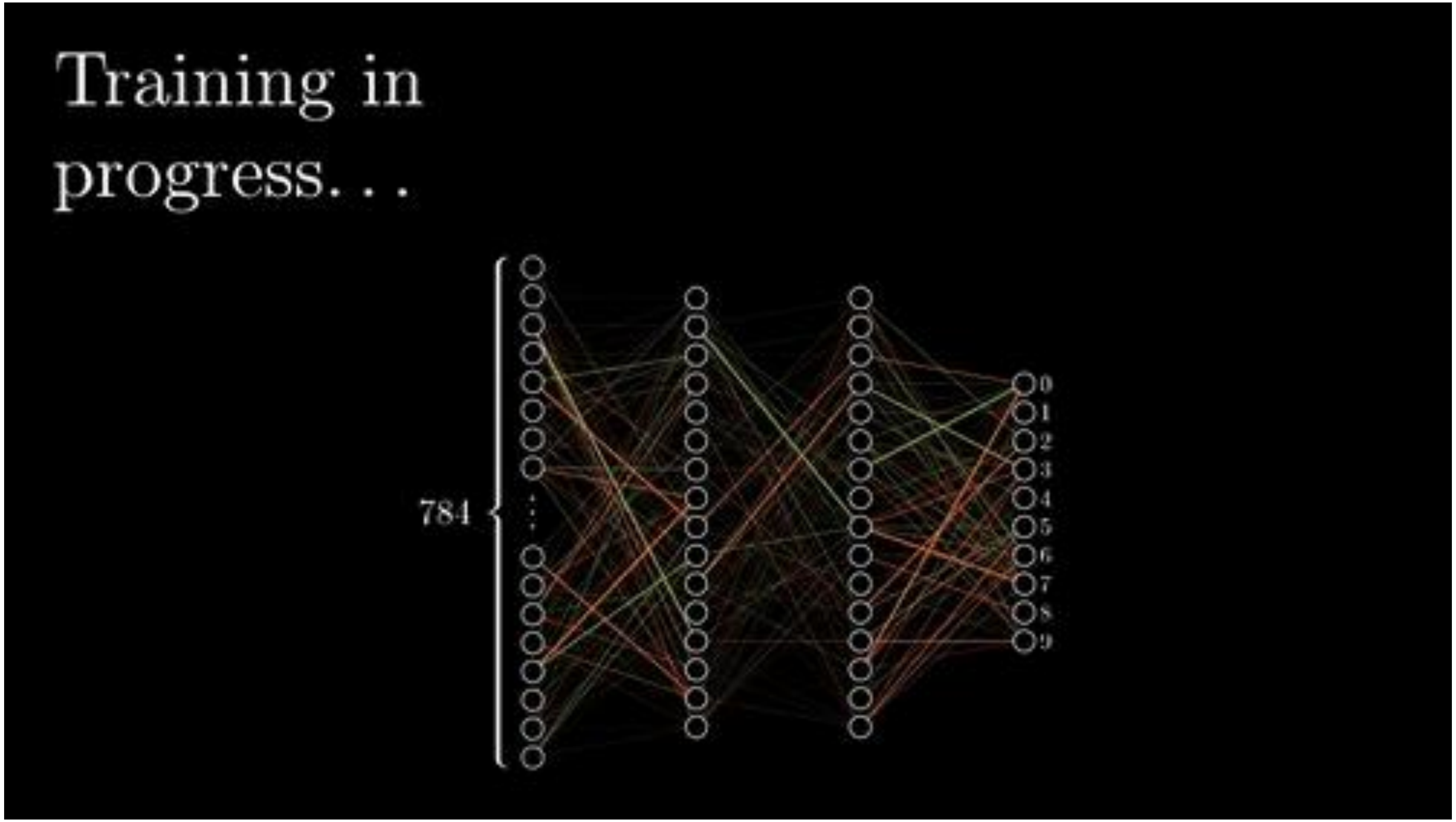
$$g(x) = \max(0, x)$$
$$g'(x) = \mathbf{1}_{g(x)>0}$$



- 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)

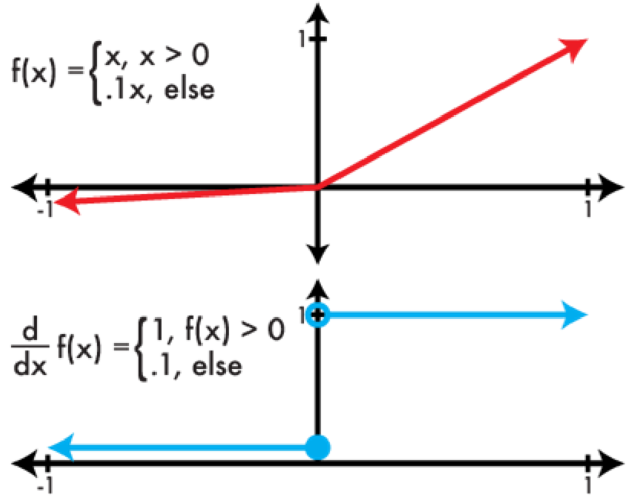


Visualization with ReLU



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

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

CSE 455 Homework 5

Neural Network

Due: 05/28

MNIST: Handwriting recognition

50,000 images of handwriting

28 x 28 x 1 (grayscale)

Numbers 0-9

10 class softmax regression

Input is 784 pixel values

Train the model

> 95% accuracy

Functions You need to Code

Functions You need to Code (**classifier.c**)

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

Run Experiments and Write a Report (**hw5.pdf**)

Play around with tryhw5.py file, and answer the questions.

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

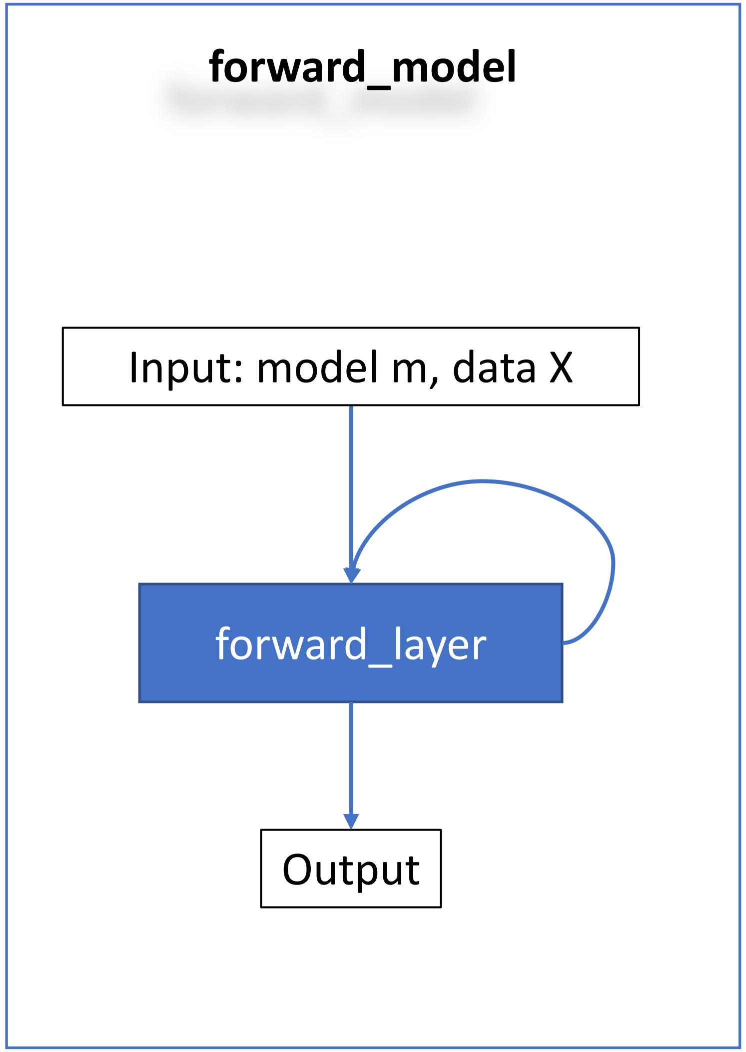
Important Data Structure (image.h)

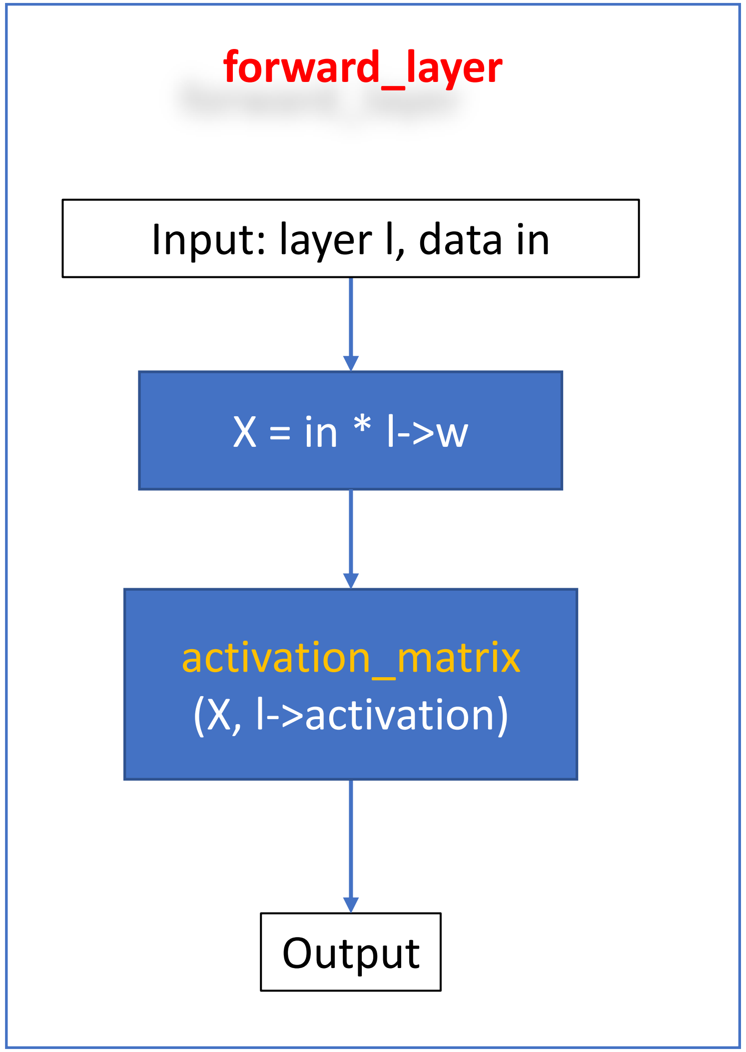
```
typedef enum{LINEAR, LOGISTIC, RELU, LRELU, SOFTMAX} ACTIVATION;

typedef struct {
    matrix in;              // Saved input to a layer
    matrix w;               // Current weights for a layer
    matrix dw;              // Current weight updates
    matrix v;               // Past weight updates (for use with momentum)
    matrix out;             // Saved output from the layer
    ACTIVATION activation;  // Activation the layer uses
} layer;

typedef struct {
    layer *layers;
    int n;
} model;
```

Useful Matrix manipulation functions (matrix.c)

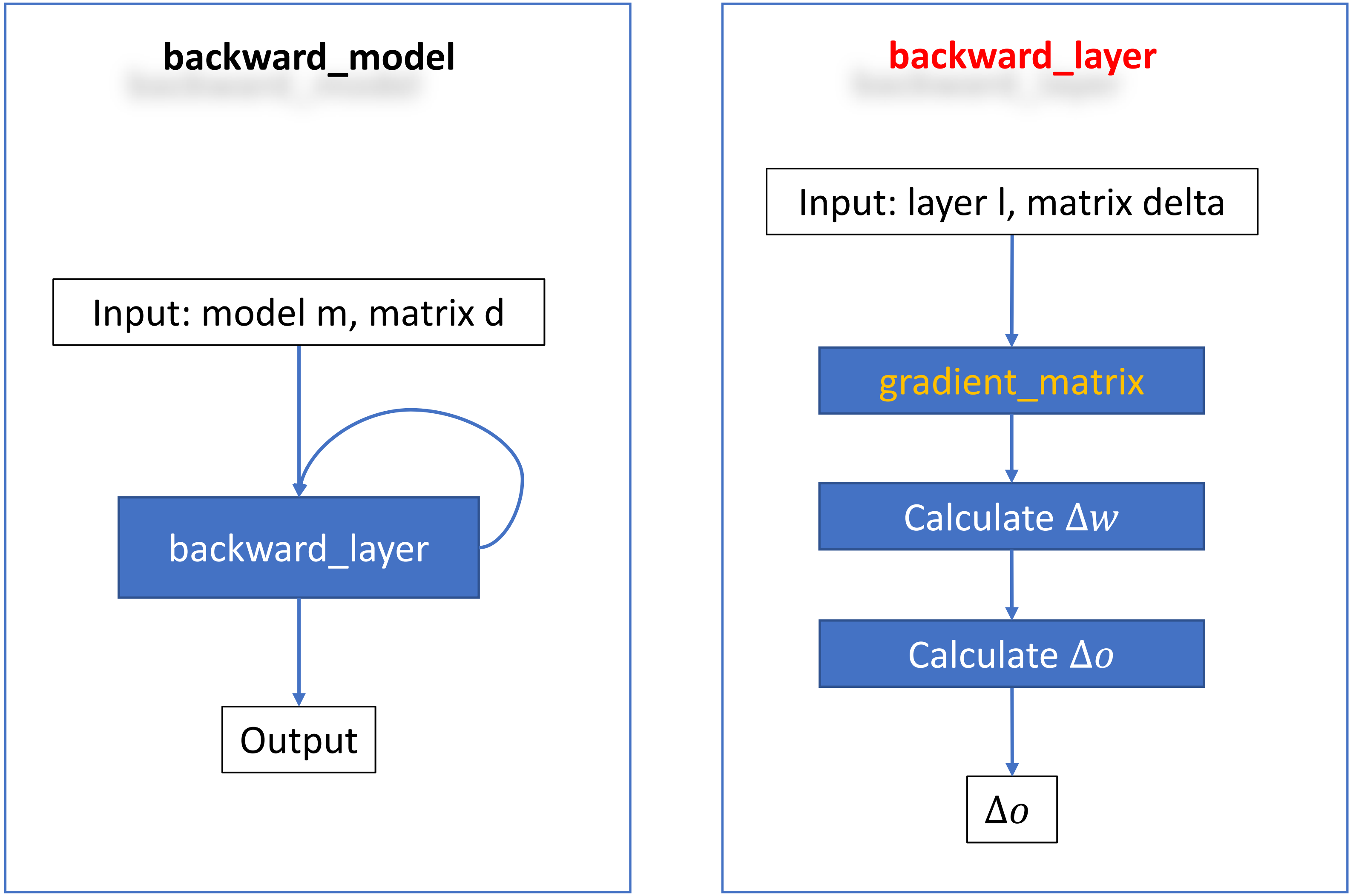
```
matrix matrix_mult_matrix(matrix a, matrix b);
matrix transpose_matrix(matrix m);
matrix axpy_matrix(double a, matrix x, matrix y); // a * x + y
```

Forward Pass in Homework



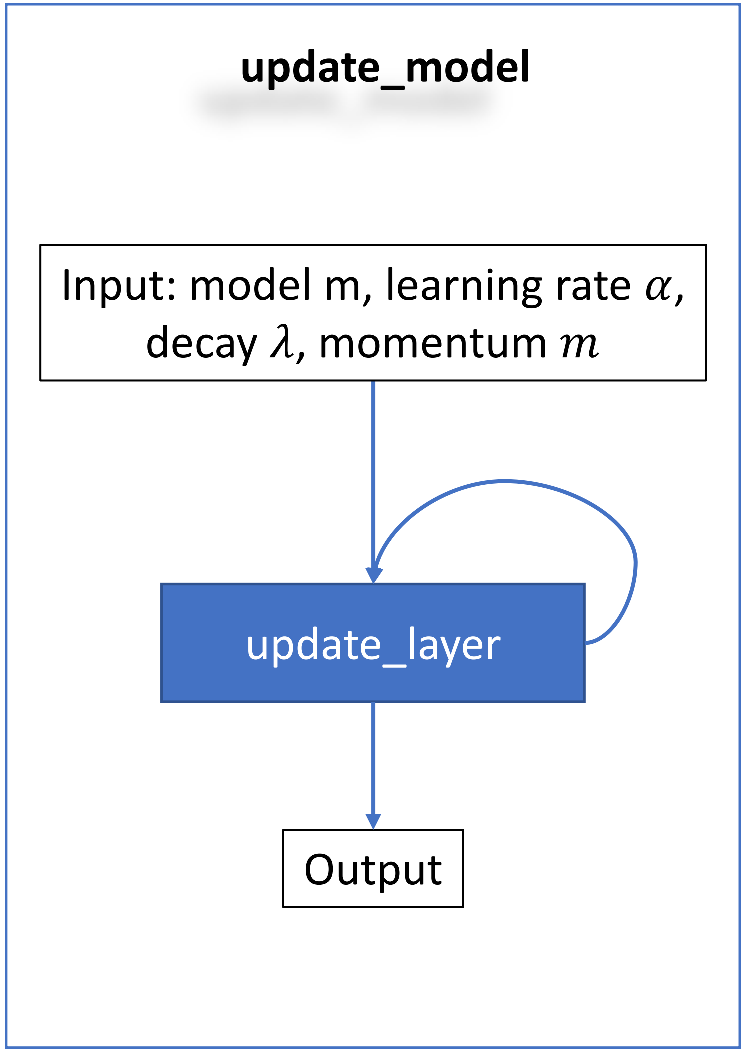


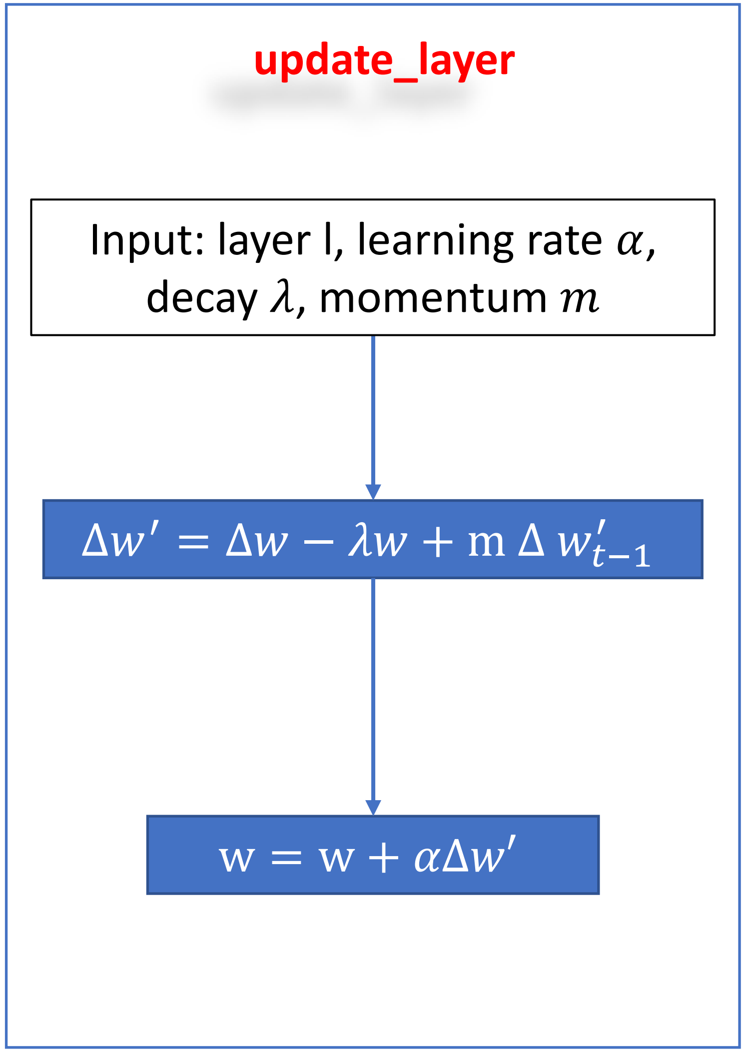
Backward Pass in Homework



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 `void activate_matrix(matrix m, ACTIVATION a)`

Apply activation "a" to the matrix "m"

```
for(i = 0; i < m.rows; ++i){
    double sum = 0;
    for(j = 0; j < m.cols; ++j){
        double x = m.data[i][j];
        if(a == LOGISTIC){
            // TODO  m.data[i][j] should equals 1 / (1 + exp(-x));
        } else if (a == RELU){
            // TODO  m.data[i][j] should equals x if x > 0; otherwise, it should equal 0
        } else if (a == LRELU){
            // TODO  m.data[i][j] should equals x if x > 0; otherwise, it should equal 0.1 * x.
        } else if (a == SOFTMAX){
            // TODO  m.data[i][j] should equals exp(x) here, and we will normalize it later.
        }
        sum += m.data[i][j];
    }
    if (a == SOFTMAX) {
        // TODO: have to normalize by sum if we are using SOFTMAX
         // for all the possible j, we should normalize it as m.data[i][j] /= sum;
    }
}
```

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.

```
int i, j;
for(i = 0; i < m.rows; ++i){
    for(j = 0; j < m.cols; ++j){
        double x = m.data[i][j];
        // TODO: multiply the correct element of d by the gradient
        // if a is SOFTMAX or a is LINEAR, we should do nothing (multiply by 1)
        // if a is LOGISTIC,  d.data[i][j] should times x * (1.0 - x);
        // if a is RELU and x <= 0, d.data[i][j] should be zero
        // if a is LRELU and x <= 0, d.data[i][j] should multiple 0.1
    }
}
```

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

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

```
l->in = in;  // Save the input for backpropagation

// TODO: multiply input by weights and apply activation function.
// Calculate out = in * l->w (note: matrix multiplication here)
// Then, apply activate_matrix function to out with l->activation

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”).

```
// delta is Δout
// TODO: modify it in place to be “g'(out) * delta” out with // gradient_matrix function.
// You can use gradient_matrix function with “l->out” and “l->activation” to “delta”

// TODO: then calculate dL/dw and save it in l->dw
free_matrix(l->dw);
// Calculate xt as the transpose matrix of “l->in”
// Calculate dw as xt times delta (matrix multiplication)
// free matrix xt to avoid memory leak
l->dw = dw;

// TODO: finally, calculate dL/dx and return it. (Similar to 1.4.2. Care memory leak)
// Calculate dx = delta * (l->w)^T,   where * is matrix multiplication and ^T is matrix transpose
return dx;
```

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

    // Calculate Δw_t = dL/dw_t - λw_t + mΔw_{t-1}
    // save it to l->v
    // Note that You can use axpy_matrix to perform the matrix summation/subtraction



    // Update l->w
    // l->w = rate * l->v + l->w
```

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

Note the multiplication and summation in this slides all mean matrix multiplication or matrix summation.

Functions You Need to Know before Experiments

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

```
layer make_layer(int input, int output, ACTIVATION activation)
matrix forward_model(model m, matrix X)
void backward_model(model m, matrix dL)
void update_model(model m, double rate, double momentum, double decay)
double accuracy_model(model m, data d)
double cross_entropy_loss(matrix y, matrix p)
void 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 **hw5.pdf**)

1. Coding and Data prepare
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