Lecture#7

Artificial Neural Networks

Linear Classifier

- We will begin by implementing a linear classifier
- It will have two major components:
- A score function that maps the data to categories
- A loss function that calculates the difference between predicted categories and actual categories in the dataset
- The loss function will be used for training the classifier

Score Function

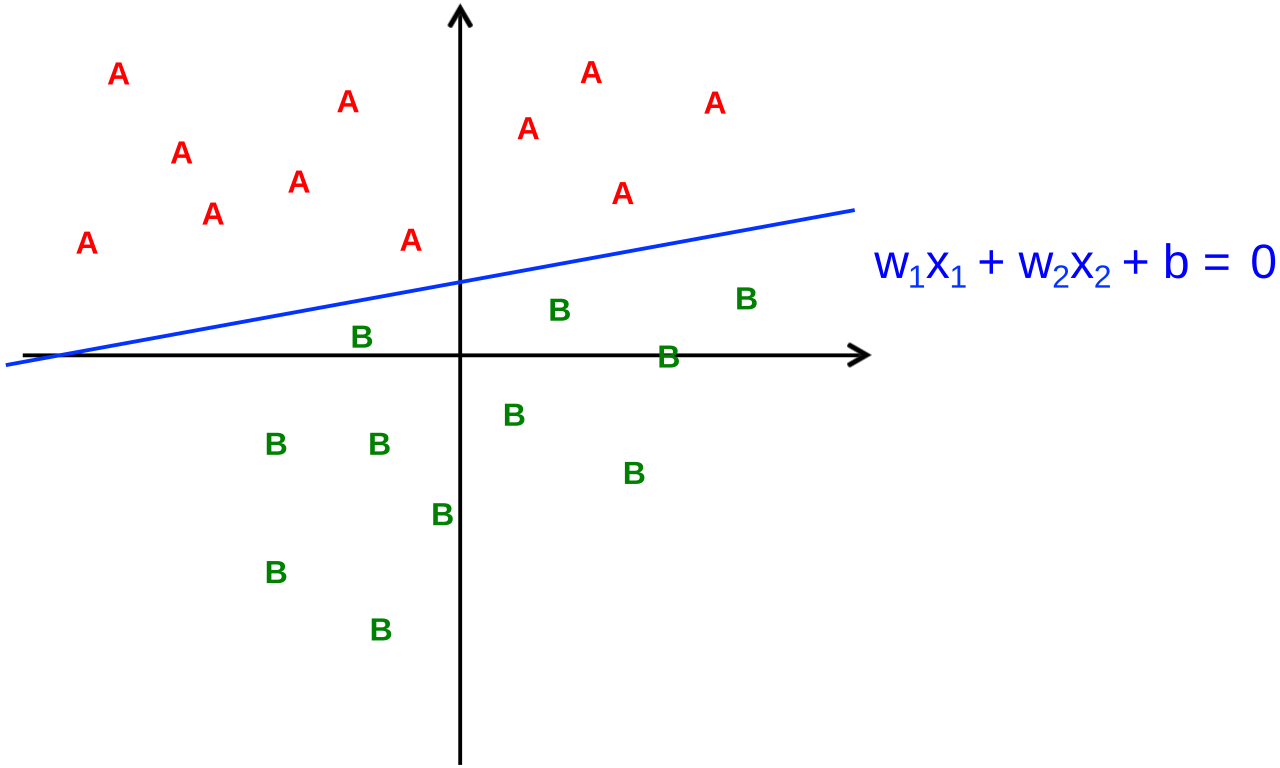
Score Function

- We have a linear function:

$$w_1x_1 + w_2x_2 + \ldots + w_nx_n + b = 0$$

- X is the input data, with one value x_i for each attribute
- Each attribute is multiplied by a weight w_i
- And finally a bias b is added
  - So the linear function doesn't have to cross the origin
- The linear function is used to separate categories:

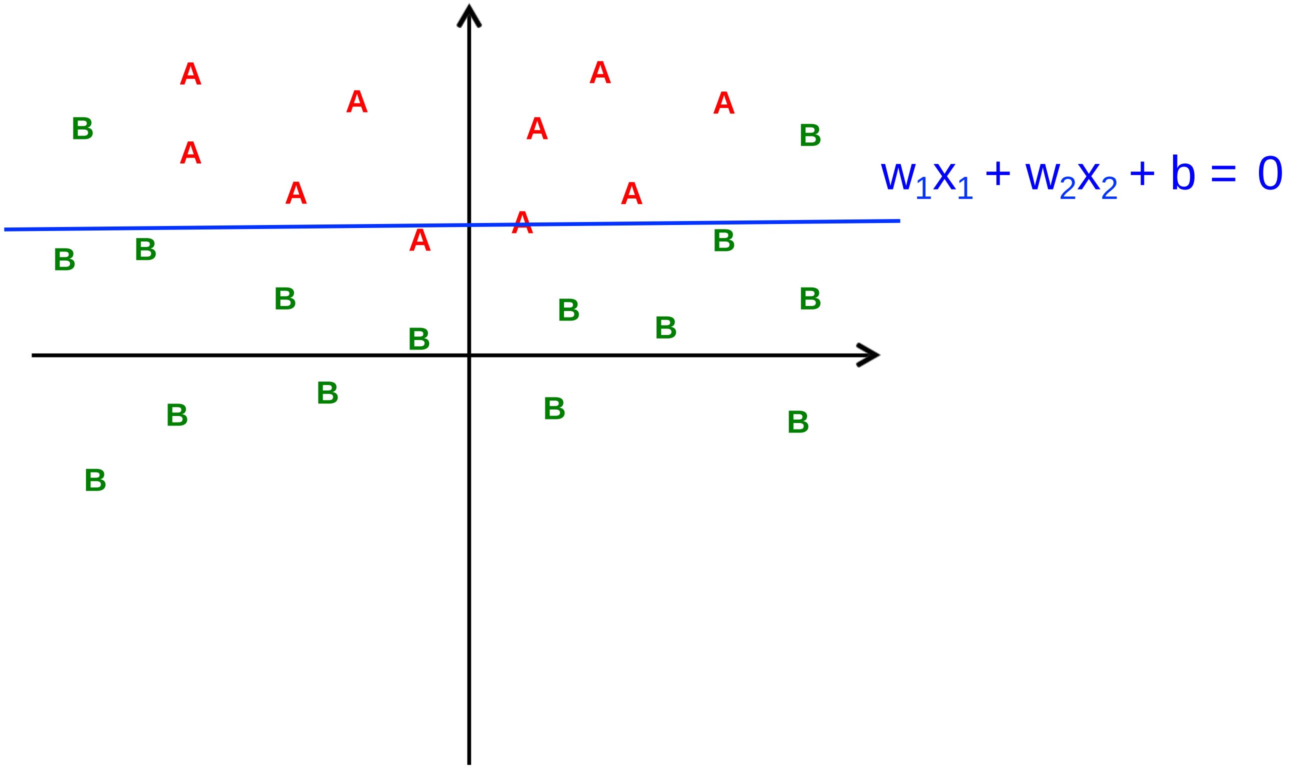
Score Function



Linear Separation

- As the name implies, the linear classificer can only separate linearly separable categories
- It will never be 100% accurate if we have a dataset that looks like this:

Linear Separation



Score Function

- If we calculate the score function:

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b = 0$$

- … for an instance we see the confidence that the example belongs to the category
  - Higher values = more confidence
- This is our score function!
- What if we have more than one category?

Mutiple Categories

- If we have two or more categories, we need one linear function for each category:

$$w_{11}x_{11} + w_{12}x_{12} + \ldots + w_{1n}x_{1n} + b_1 = 0$$
$$w_{21}x_{21} + w_{22}x_{22} + \ldots + w_{2n}x_{2n} + b_2 = 0$$
$$\ldots$$
$$w_{k1}x_{k1} + w_{k2}x_{k2} + \ldots + w_{kn}x_{kn} + b_k = 0$$

- The most efficient way to calculate the score function is to use matrix/vector operations:

Score Function

- The weights can be seen as a matrix:

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1n} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2n} \\ \dots & & & & \\ w_{k1} & w_{k2} & w_{k3} & \dots & w_{kn} \end{bmatrix}$$

- … and the bias and example as column vectors:

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_k \end{bmatrix} \qquad \boldsymbol{x_i} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

Score Function

- Calculating the score function is then a matrix-vector multiplication plus addition:

$$f(\boldsymbol{x_i}, \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{W}\boldsymbol{x_i} + \boldsymbol{b}$$

- This produces a vector with one confidence value for each category
- The example is classified as the category with the highest confidence:

$$y_{pred} = argmax(\boldsymbol{scores})$$

How it works

- Assume we have two categories and three inputs:

$$\boldsymbol{W}\boldsymbol{x_i} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \end{bmatrix}$$

- … and with the bias vector:

$$\boldsymbol{W}\boldsymbol{x_i} + \boldsymbol{b} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2 \end{bmatrix}$$

- This is actually the dot-product of $\mathbf{x_i}$ with each row in **W**
- Number of columns in **W** must be equal to the number of components in $\mathbf{x_i}$

How it works

- We don't even need to split the input data **X** into columns
- When calculating a product between matrices **W** and **X**, we can see **X** as a bunch of lined up column vectors:

$$\boldsymbol{WX} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \end{bmatrix} \begin{bmatrix} x_{21} \\ x_{22} \\ x_{23} \end{bmatrix}$$

- This results in a new matrix:

$$\boldsymbol{WX} = \begin{bmatrix} w_{11}x_{11} + w_{12}x_{12} + w_{13}x_{13} & w_{11}x_{21} + w_{12}x_{22} + w_{13}x_{23} \\ w_{21}x_{11} + w_{22}x_{12} + w_{23}x_{13} & w_{21}x_{21} + w_{22}x_{22} + w_{23}x_{23} \end{bmatrix}$$

How it works

- The bias vector **b** is then added to each column:

$$\boldsymbol{WX} + \boldsymbol{b} = \begin{bmatrix} w_{11}x_{11} + w_{12}x_{12} + w_{13}x_{13} + b_1 & w_{11}x_{21} + w_{12}x_{22} + w_{13}x_{23} + b_1 \\ w_{21}x_{11} + w_{22}x_{12} + w_{23}x_{13} + b_2 & w_{21}x_{21} + w_{22}x_{22} + w_{23}x_{23} + b_2 \end{bmatrix}$$

- Now we have a matrix where each column is a score vector for an example $\mathbf{x_i}$ in **X**
- Taking *argmax* for each column produces a row vector with the predicted category for each example:

$$\boldsymbol{Y_{pred}} = \begin{bmatrix} argmax(\boldsymbol{scores_1}) & argmax(\boldsymbol{scores_2}) \end{bmatrix}$$

Simple example

Image is converted to pixel
vector (only 4 pixels used)

W				x_i		b		scores	
0.2	-0.5	0.1	2.0	56	+	1.1	=	-96.8	cat score
1.5	1.3	2.1	0.0	231		3.2		**437.9**	dog score
0	0.25	0.2	-0.3	24		-1.2		60.75	rabbit score
				2					

This is clearly a dog...

The weights need to be modified
(learned) to produce correct output!

Simple example

Image is converted to pixel vector (only 4 pixels used)

W

0.8	0.5	0.1	2.0
0.2	-0.1	2.1	0.0
0	0.15	0.2	-0.3

x_i

56
231
24
2

\+

b

1.1
3.2
-1.2

=

scores

167.8	cat score
41.7	dog score
37.65	rabbit score

Now we get correct output!

How can we automatically learn weights from training data?

Loss Function

Loss Function

- First, we need to define a loss function
  - Sometimes called cost function or objective
- The loss function measures how happy we are with the result
- The first set of weights gave a poor prediction – we are not happy
- The second set of weights gave a good prediction – we are happy!
- The loss will be high for bad predictions, and low for good predictions
- There are many loss functions, but we will focus on Softmax

Softmax

- Softmax calculates the normalized probabilities for belonging to each category
- This is then combined to a single loss value: cross-entropy loss

Softmax

- The loss L_i is calculated as:

$$L_i = -log\left(\frac{e^{f_{yi}}}{\sum_j e^{f_j}}\right)$$

- We calculate the log probability for the correct category e^{fyi} and normalize by dividing with the sum of log probabilities for all categories
- Finally we calculate the negative natural logarithm of the normalized log probability for the correct class

Example

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

x_i

-15
22
-44
56

+

b

0.0
0.2
-0.3

=

scores	→	e^{f_j}	→	normalize
-2.85		0.058		0.016
0.86		2.36		0.632
0.28		1.32		0.353

y_i = 2

Σ = 1.0

loss = -log(0.353) = **1.04**

Matrix Multiplication

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

$\mathbf{x_i}$

-15
22
-44
56

=

= dot-product of row 1 in W and column X_i
= dot-product of row 2 in W and column X_i
= dot-product of row 3 in W and column X_i

Matrix Multiplication

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

$\mathbf{x_i}$

-15
22
-44
56

=

= 0.01 * -15 - 0.05 * 22 + 0.1 * -44 + 0.05 * 56 = **-2.85**
= 0.7 * -15 + 0.2 * 22 + 0.05 * -44 + 0.16 * 56 = **0.66**
= 0 * -15 - 0.45 * 22 - 0.2 * -44 + 0.03 * 56 = **0.58**

Matrix Addition

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

$\mathbf{x_i}$

-15
22
-44
56

=

-2.85
0.66
0.58

\+

b

0.0
0.2
-0.3

=

scores

-2.85
0.86
0.28

Simply add each element of vector **b**

Numerical Stability

- If we have very high scores, calculating e^{f_j} and then sum all the values can lead to numerical problems
- The sum can blowup, i.e. we get outside the range of *double*
- This can be solved by shifting all scores so that the highest score is 0:
  - Find *max(scores)*
  - Subtract *max(scores)* for each *score*

Regularization

- Suppose we have a perfect set of weights: loss = 0.0
- The problem is that this set might not be unique!
- There can be multiple sets of weights that give the same loss
- To distinct between two such sets, we extend the loss function with a regularization penalty:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Regularization

- The most common one is the L2 norm, which penalizes large weights
  - Large weights can lead to numerical overflow…
  - Small weights improve generalization and reduces overflow
- The L2 norm is calculated as the squared sum of all weights:

$$R(W) = \sum_k \sum_l w_{k,l}^2$$

- The lambda parameter is called the reqularization strength, and is typically set to a low value such as 0.01

Example

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

→

Squared W

0.0001	0.0025	0.01	0.0025
0.49	0.04	0.0025	0.0256
0.0	0.2025	0.04	0.0009

L2 norm = sum of all squared W
= **0.8166**

Example

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

$\mathbf{x_i}$

-15
22
-44
56

+

b

0.0
0.2
-0.3

y_i

2

Loss = Data loss + regularization loss
= 1.04 + 0.01 * 0.8166 = **1.048**

Example

W

1.00	2.00
2.00	-4.00
3.00	-1.00

b

0.00
0.50
-0.50

Squared W

1.0	4.0
4.0	16.0
9.0	1.0

sum: 35

λ: 0.01

X		y	scores			L
0.50	0.40	0	1.30	-0.10	0.60	0.56
0.80	0.30	0	1.40	0.90	1.60	1.04
0.30	0.80	0	1.90	-2.10	-0.40	0.11
-0.40	0.30	1	0.20	-1.50	-2.00	1.96
-0.30	0.70	1	1.10	-2.90	-2.10	4.06
-0.70	0.20	1	-0.30	-1.70	-2.80	1.68
0.70	-0.40	2	-0.10	3.50	2.00	1.72
0.50	-0.60	2	-0.70	3.90	1.60	2.40
-0.40	-0.50	2	-1.40	1.70	-1.20	3.00

mean: 1.84

Data loss: 1.84

Regularization loss: 0.35

Total loss: 2.19

Optimization

Optimization

- The loss function quantifies the quality of a set of weights
- The goal of optimization, or learning, is to find a set of weights that minimizes the loss function
- This can of course be done with random search or hill climbing, but it will most likely take ages to find a good set of weights
- Instead we can compute the best direction using the gradient of the loss function!

Gradient

- The task is to computer the best direction in which we should change the weights
- This direction turns out to be related to the gradient of the loss function
- The gradient is a vector of slopes (derivatives) for each dimension in the input space
- Mathematically, the derivative of a 1-D function with respect to its (single) input is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

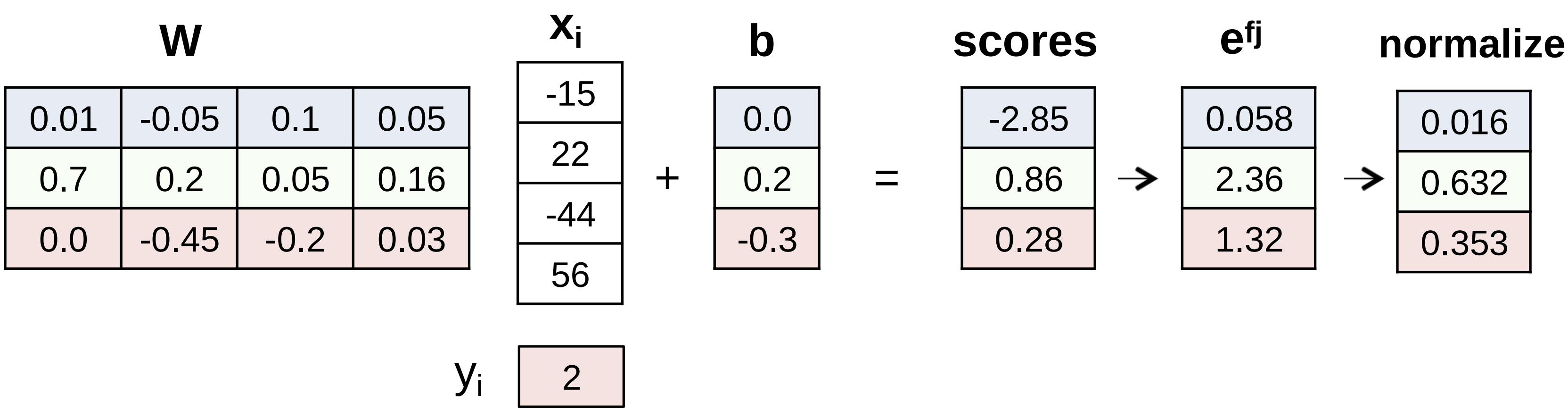
Gradient

- If we have a function that takes a vector of numbers instead of a single number, the derivatives are called partial derivatives
- The gradient is simply the vector of partial derivatives in each input dimension
- We can do this in two ways:
  - Numerical gradient: slow and approximate
  - Analytic gradient: fast and exact but error-prone
- Since speed is important, we will focus on the analytic gradient

Analytic Gradient

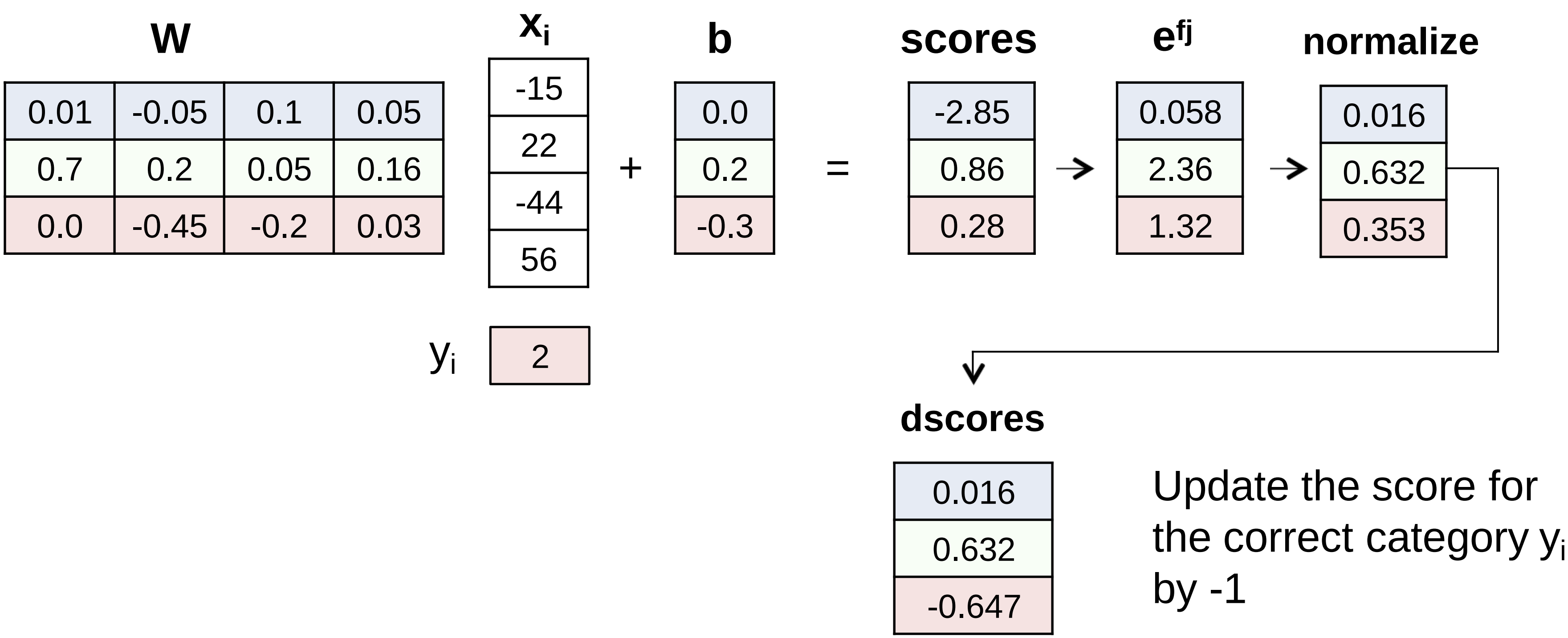
- To find the analytic gradient, we need to derive a formula for the gradient using our math skills
- Luckily, the loss functions we use are well known and we don't have to find the formula on our own
- Depending on the loss function, the formula can be quite complex to implement
- How can we implement the gradients formula for Softmax?

Softmax Gradients



This is what we have already done
when calculating loss

Softmax Gradients



Update the score for the correct category y_i by -1

Softmax Gradients

dscores

0.016
0.632
-0.647

$\mathbf{x}_i^T$

-15	22	-44	56

=

dW

-0.23	0.34	-0.68	0.87
-9.47	13.89	-27.77	35.35
9.70	-14.23	28.45	-36.21

Multiply **dscores** with the transpose of $\mathbf{x}_i$

Multiply column and row vector

dscores

0.016
0.632
-0.647

x_i^T

-15	22	-44	56

=

= 0.016 * -15 = **-0.23**	= 0.016 * 22 = **0.34**	= 0.016 * -44 = **-0.68**	= 0.016 * 56 = **0.87**
= 0.632 * -15 = **-9.47**	= 0.632 * 22 = **13.89**	= 0.632 * -44 = **-27.77**	= 0.632 * 56 = **35.35**
= -0.647 * -15 = **9.70**	= -0.647 * 22 = **-14.23**	= -0.647 * -44 = **28.45**	= -0.647 * 56 = **-36.21**

$M_{0,0} = dscores_0 * X^T_{i\ 0}$
$M_{0,1} = dscores_0 * X^T_{i\ 1}$
...

Softmax Gradients

dscores				dB
0.016	=	0.016	=	0.016
0.632		0.632		0.632
-0.647		-0.647		-0.647

Sum the values of all rows
in **dscores** into a new vector

Softmax Gradients

dW

-0.23	0.34	-0.68	0.87
-9.47	13.89	-27.77	35.35
9.70	-14.23	28.45	-36.21

dB

0.016
0.632
-0.647

Now we have the gradients!

What if we have multiple input examples?

Multiple training examples

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

x_i

-15	8
22	-12
-44	14
56	-5

y_i

2	1

+

b

0.0
0.2
-0.3

=

scores

-2.85	1.83
0.86	3.30
0.28	2.15

e^{f_j}

0.058	6.23
2.36	27.11
1.32	8.59

→

normalize

0.016	0.149
0.632	0.647
0.353	0.205

Multiple training examples

dscores

0.016	0.149
0.632	-0.353
-0.647	0.205

y_i	2	1

Update the score for the correct categories y_i by -1

Multiple training examples

dscores

0.0078	0.074
0.316	-0.177
-0.323	0.102

Divide by number of training examples (2 in this case)

Multiple training examples

dscores

0.0078	0.074
0.316	-0.177
-0.323	0.102

$\mathbf{x_i}^T$

-15	22	-44	56
8	-12	14	-5

=

dW

0.479	-0.722	0.701	0.061
-6.147	9.063	-16.359	18.556
5.669	-8.341	15.659	-18.617

Multiply **dscores** with the transpose of $\mathbf{x_i}$

Softmax Gradients

dscores

0.0078	0.074
0.316	-0.177
-0.323	0.102

=

0.0078+0.074
0.316-0.177
-0.323+0.102

=

dB

0.082
0.139
-0.221

Sum the values of all rows
in **dscores** into a new vector

Regularization

- We also need to add a regularization factor to the weight gradients **dW**
- This is done by adding the weight matrix **W** scaled by lambda/2 to **dW**
- Let’s go back to our first example with a single training example:

Regularization   Factor

dW

-0.23	0.34	-0.68	0.87
-9.47	13.89	-27.77	35.35
9.70	-14.23	28.45	-36.21

+

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

$* \lambda * 0.5$ =

$dW + W * \lambda * 0.5$

=

-0.2317	0.3396	-0.6793	0.8654
-9.4639	13.8866	-27.7709	35.3459
9.6992	-14.2277	28.4500	-36.2102

dB is not changed

Weights Upgrades

- The weights are upgraded by subtracting **dW** multiplied by a learning rate
- The learning rate is typically set to a low value such as 0.1 or 0.05
- The best learning rate for each dataset has to be discovered by trial and error…

Weights Upgrades

W

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

-

dW

-0.2317	0.3396	-0.6793	0.8654
-9.4639	13.8866	-27.7709	35.3459
9.6992	-14.2277	28.4500	-36.2102

*** 0.1** =

newW

=

0.033	-0.084	0.168	-0.037
1.646	-1.189	2.827	-3.375
-0.970	0.973	-3.045	3.651

Bias Upgrades

b		**dB**			**newB**
0.0		0.082			-0.002
0.2	-	0.139	*** 0.1**	=	0.137
-0.3		-0.221			-0.235

If we calculate the loss, it has decreased
from **1.04** to **0.48**

Back to our previous example

W

1.00	2.00
2.00	-4.00
3.00	-1.00

b

0.00
0.50
-0.50

X

0.50	0.40
0.80	0.30
0.30	0.80
-0.40	0.30
-0.30	0.70
-0.70	0.20
0.70	-0.40
0.50	-0.60
-0.40	-0.50

y

0
0
0
1
1
1
2
2
2

scores

1.30	-0.10	0.60
1.40	0.90	1.60
1.90	-2.10	-0.40
0.20	-1.50	-2.00
1.10	-2.90	-2.10
-0.30	-1.70	-2.80
-0.10	3.50	2.00
-0.70	3.90	1.60
-1.40	1.70	-1.20

L

0.56
1.04
0.11
1.96
4.06
1.68
1.72
2.40
3.00

mean 1.84

Let's calculate the gradients!

Data loss: 1.84

Regularization loss: 0.35

Total loss: 2.19

Example - iteration 0

W

1.00 -0.20	2.00 0.07
2.00 0.24	-4.00 -0.27
3.00 -0.01	-1.00 0.19

b

0.00 0.15
0.50 0.04
-0.50 -0.19

X		y	scores			L
0.50	0.40	0	1.30	-0.10	0.60	0.56
0.80	0.30	0	1.40	0.90	1.60	1.04
0.30	0.80	0	1.90	-2.10	-0.40	0.11
-0.40	0.30	1	0.20	-1.50	-2.00	1.96
-0.30	0.70	1	1.10	-2.90	-2.10	4.06
-0.70	0.20	1	-0.30	-1.70	-2.80	1.68
0.70	-0.40	2	-0.10	3.50	2.00	1.72
0.50	-0.60	2	-0.70	3.90	1.60	2.40
-0.40	-0.50	2	-1.40	1.70	-1.20	3.00

mean 1.84

Data loss: 1.84

Regularization loss: 0.35

Total loss: 2.19

Example - iteration 1

W

1.02 -0.20	1.99 0.07
1.98 0.24	-3.97 -0.27
3.00 -0.01	-1.02 0.19

b

-0.02 0.15
0.50 0.04
-0.48 -0.18

X		y	scores			L
0.50	0.40	0	1.29	-0.10	0.61	0.56
0.80	0.30	0	1.40	0.89	1.61	1.04
0.30	0.80	0	1.89	-2.09	-0.40	0.11
-0.40	0.30	1	0.17	-1.49	-1.99	1.93
-0.30	0.70	1	1.07	-2.88	-2.09	4.01
-0.70	0.20	1	-0.33	-1.68	-2.79	1.65
0.70	-0.40	2	-0.10	3.47	2.03	1.68
0.50	-0.60	2	-0.70	3.87	1.63	2.35
-0.40	-0.50	2	-1.42	1.69	-1.17	1.96

mean 1.81

Data loss: 1.81

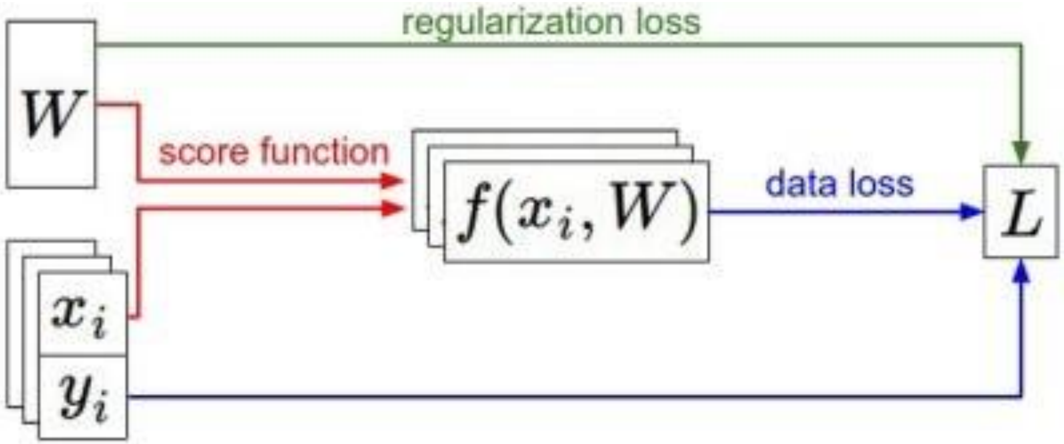
Regularization loss: 0.35

Total loss: 2.16

Gradient Descent

- The procedure of repeatedly evaluating the gradients and perform weights updates is call Gradient Descent
- It is the most common way of optimizing/training linear classifiers, and also Neural Networks which we will look into shortly
- We can also train on batches of the training examples instead of all examples
  - Mini-batch Gradient Descent
- Or we can train on one example at a time
  - Stochastic Gradient Descent

Overview of information flow



Linear Softmax classifier

- Now we have a complete linear Softmax classifier
- Let's see how well it works on the example data:

X		y
0.50	0.40	0
0.80	0.30	0
0.30	0.80	0
-0.40	0.30	1
-0.30	0.70	1
-0.70	0.20	1
0.70	-0.40	2
0.50	-0.60	2
-0.40	-0.50	2

Linear Softmax classifier

λ: 0.01
Lrate: 1.0

Iteration	Loss	Accuracy	
0	2.19	2/9	22.2%
1	1.91	2/9	22.2%
2	1.67	2/9	22.2%
3	1.49	3/9	33.3%
4	1.34	4/9	44.4%
5	1.22	5/9	55.6%
6	1.11	6/9	66.7%
7	1.03	6/9	66.7%
8	0.96	7/9	77.8%
9	0.90	7/9	77.8%
10	0.85	7/9	77.8%
11	0.81	7/9	77.8%
12	0.77	7/9	77.8%
13	0.74	7/9	77.8%
14	0.71	7/9	77.8%
15	0.69	7/9	77.8%
16	0.67	7/9	77.8%
17	0.66	8/9	88.9%
18	0.64	8/9	88.9%
19	0.63	9/9	100%

Iris dataset

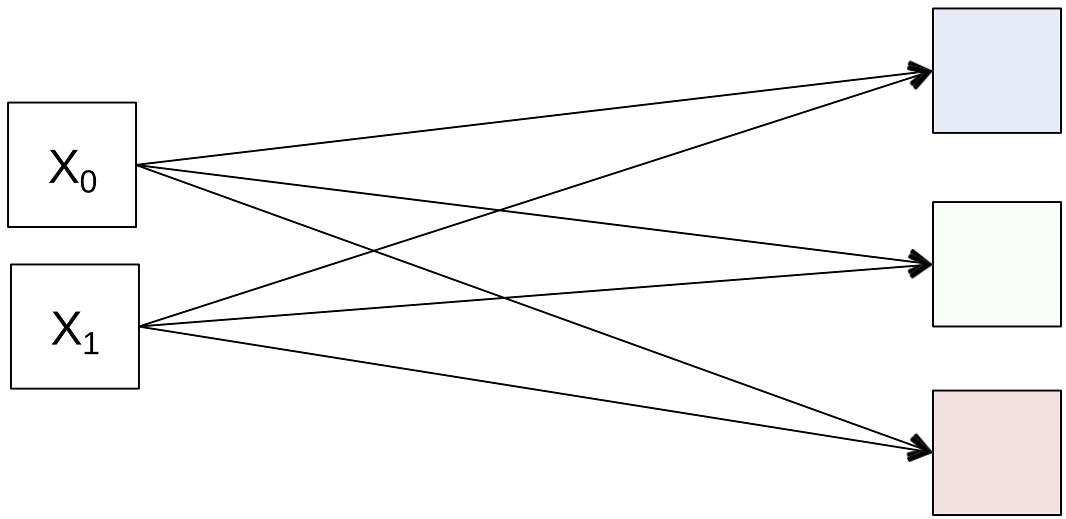
Iteration	Loss
0	1.0711
40	0.6935
80	0.5791
120	0.4842
160	0.4052
200	0.3655
240	0.3603
280	0.3591
300	0.3592

Final Result		
Loss:	0.3591	
Accuracy	147/150	98%

λ: 0.01
Lrate: 0.1
Iterations: 300

How can we expand this into a Neural Network?

Current network layout

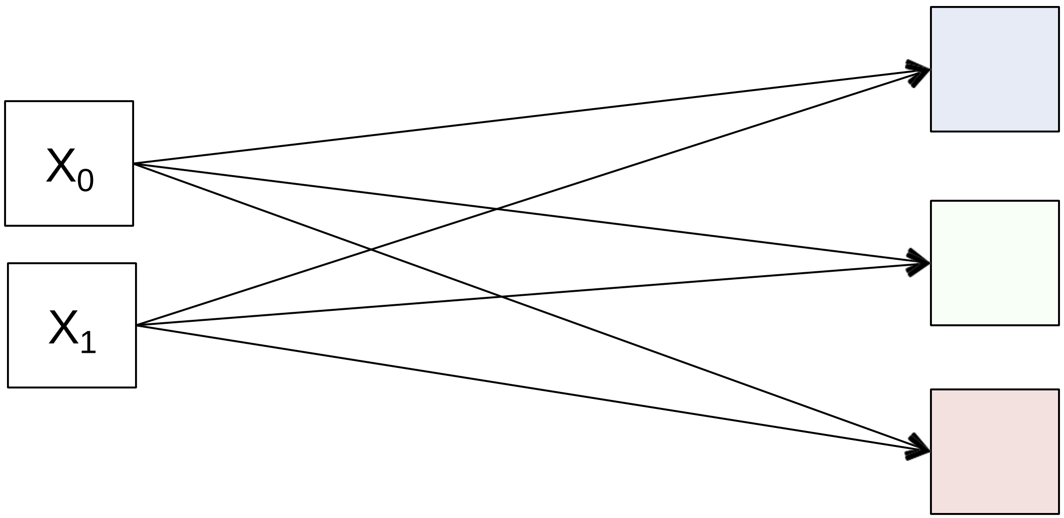


We have a single layer,
the output layer

Inputs

Softmax output
layer

Current network layout

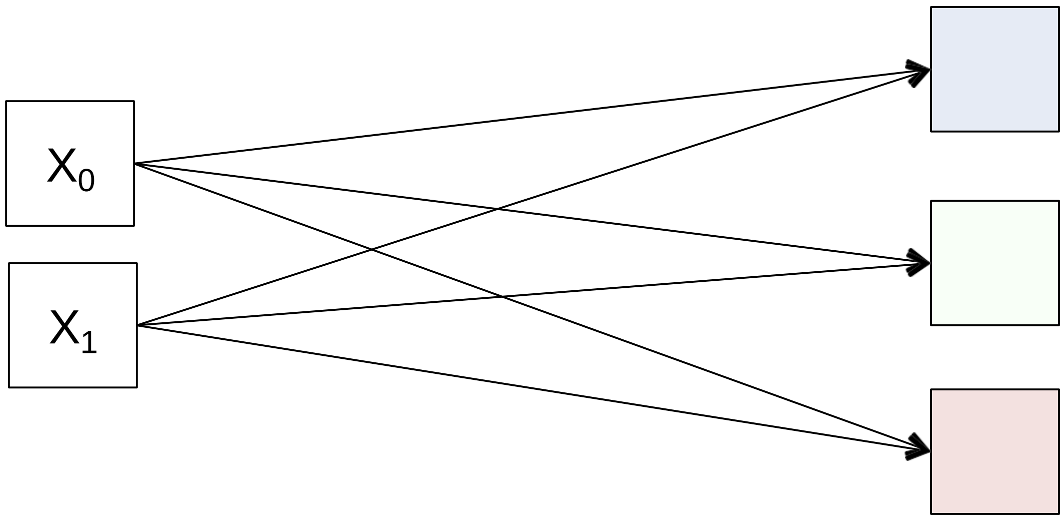


Each node in the network is called a Unit

Inputs

Softmax output layer

Current network layout

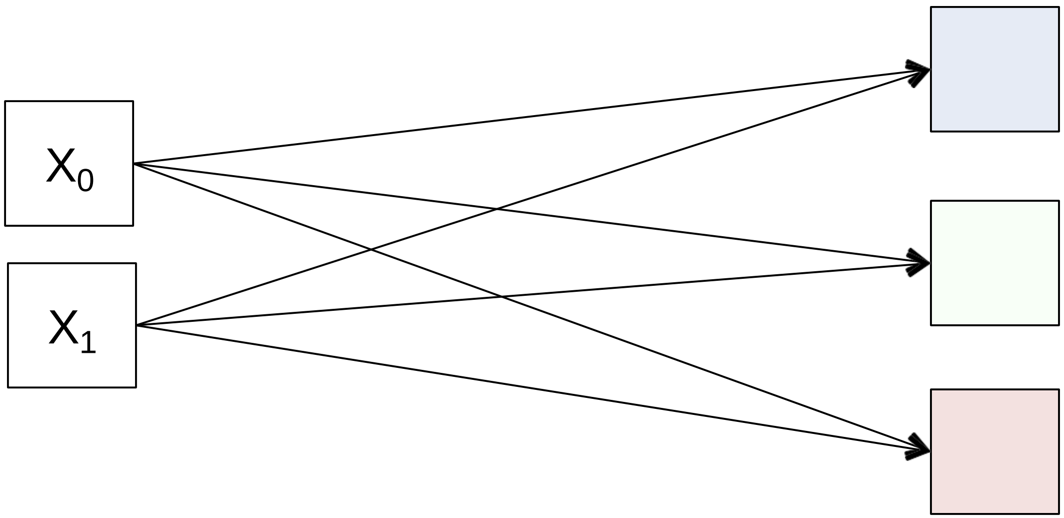


We have a network with two input units and three output units

Inputs

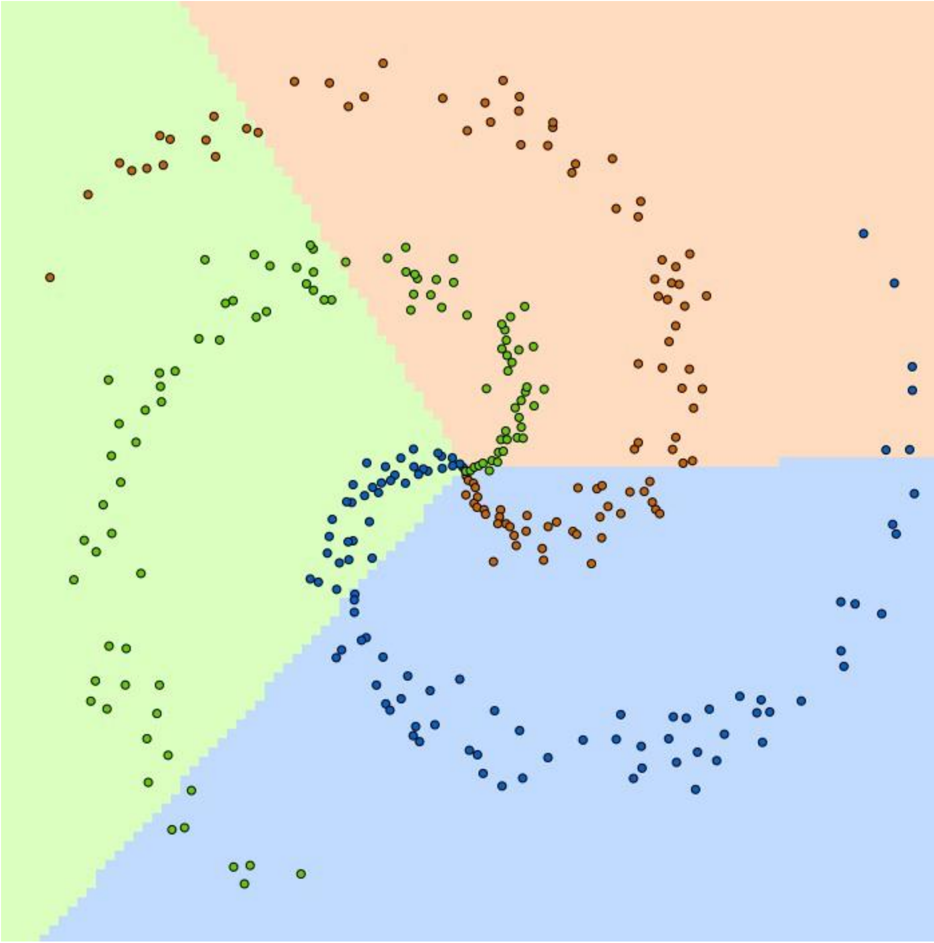
Softmax output layer

Limitations

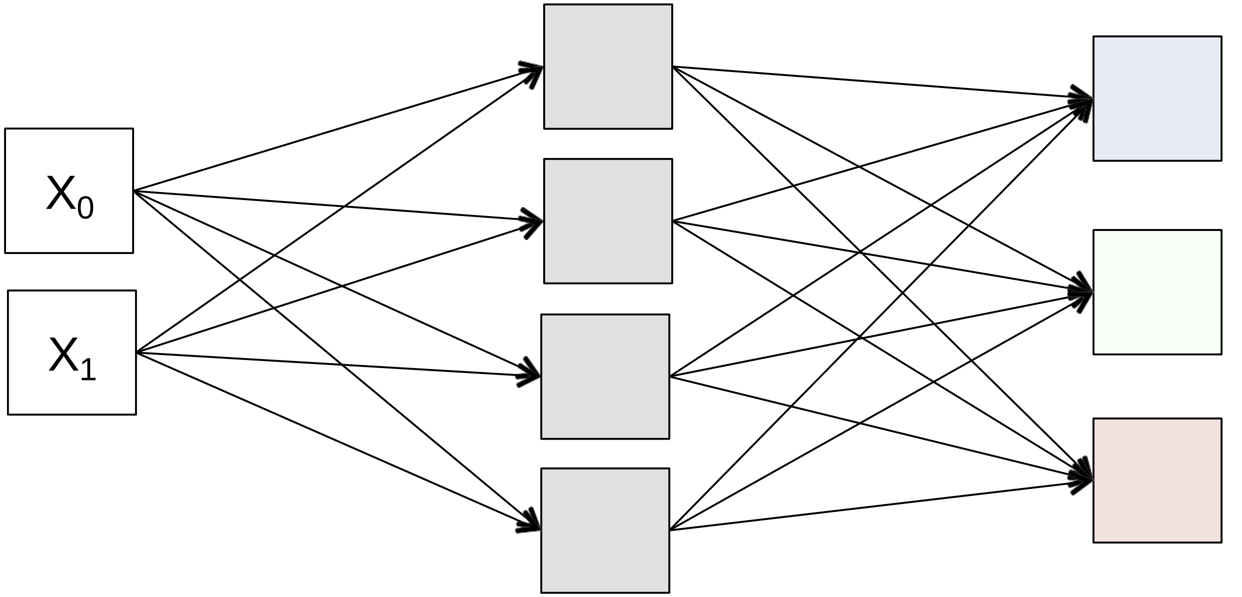


Even if this is a quite powerful classifier, it can only handle categories that are linearly separable!

Limitations



Layered network

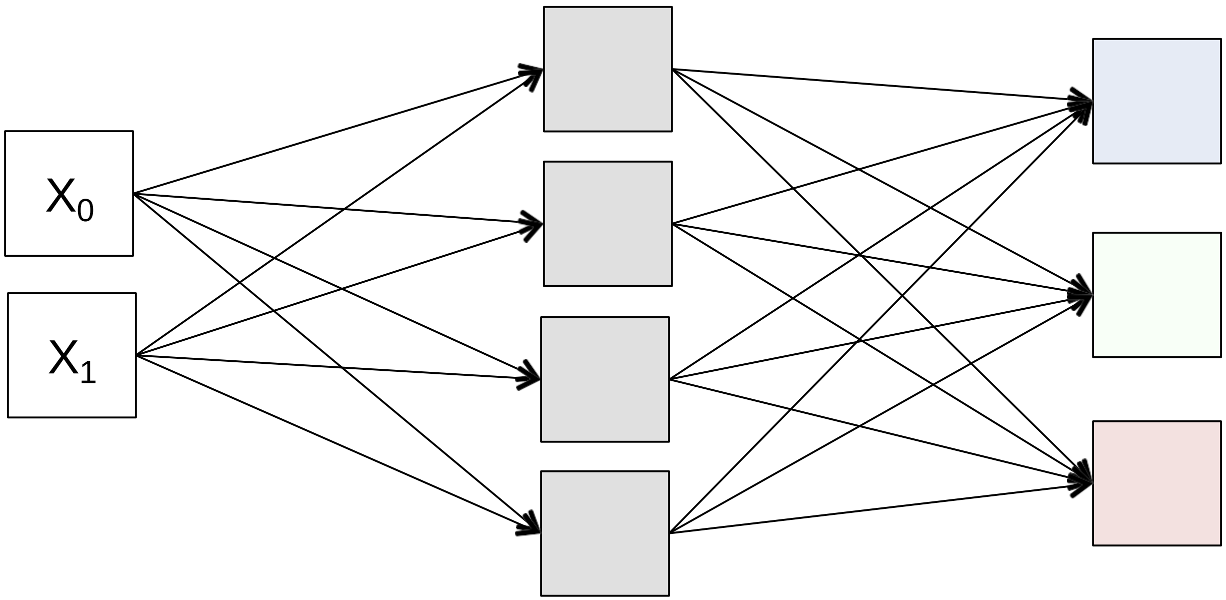


Inputs

Expand with a layer of hidden nodes

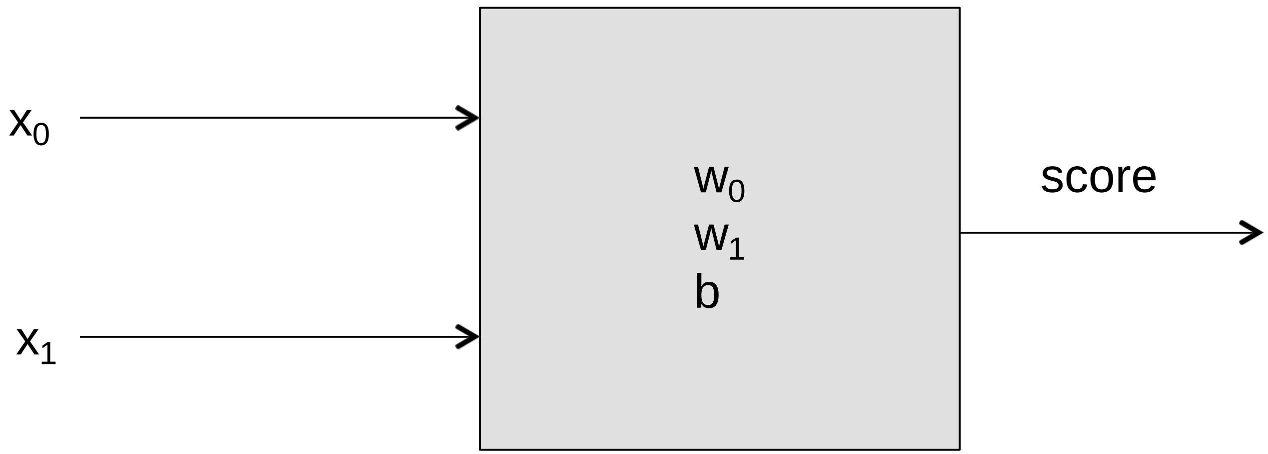
Softmax output layer

Layered network



The layered (neural) network can learn categories that are not linearly separable!

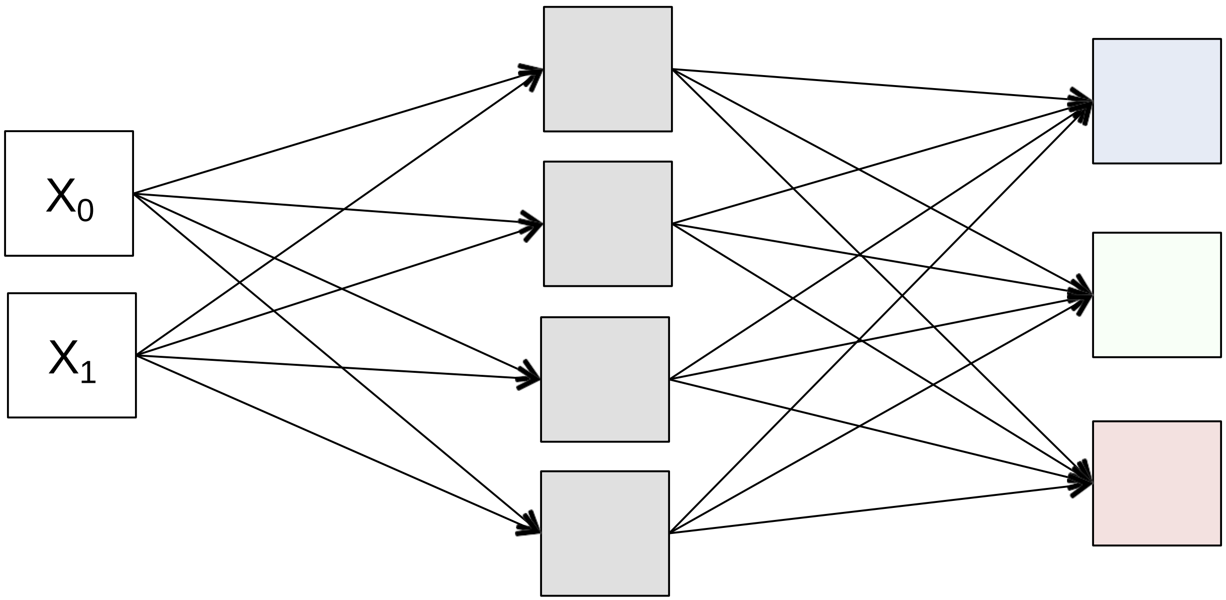
Unit



Each unit has its own set of inputs, a weight for each input and a bias.

The output (score) can act as input to units in another layer.

Score Function



The input data x is the input to the hidden layer

The scores of the hidden layer is input to the output layer

Hidden Layer Units

- In the output layer we used the Softmax function
- In the hidden layer we need a slightly different type of activation function
- There is a wide range we can choose from:
  - Sigmoid
  - Tanh
  - ReLU
  - …
- Here, we will use the ReLU function

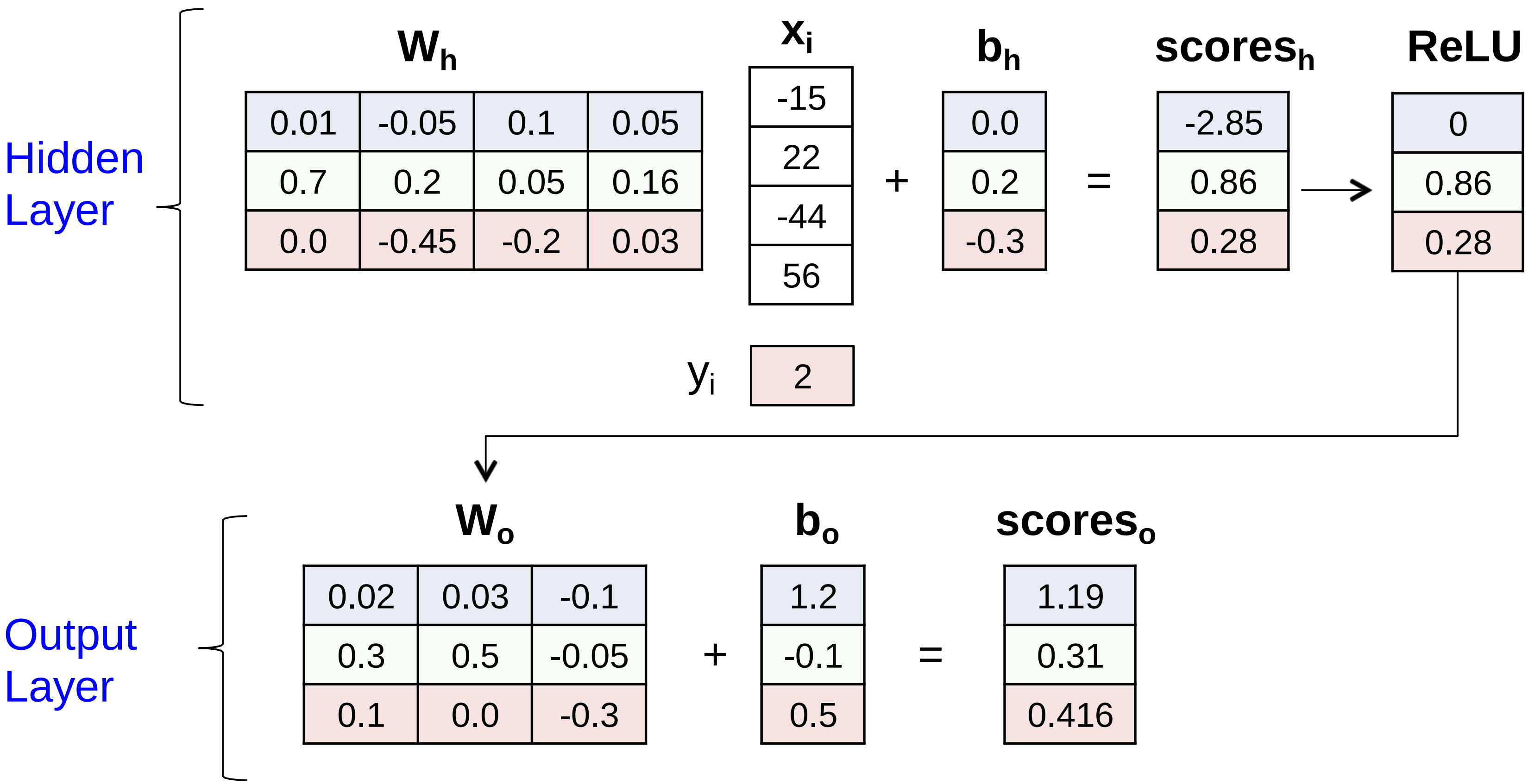
ReLU

- The ReLU (Rectified Linear Unit) calculates the function:

$$f(x) = max(0, x)$$

- First, the weighted sum of the inputs plus the bias is calculated (as we've done before)
- Then, the activation function is applied on the result

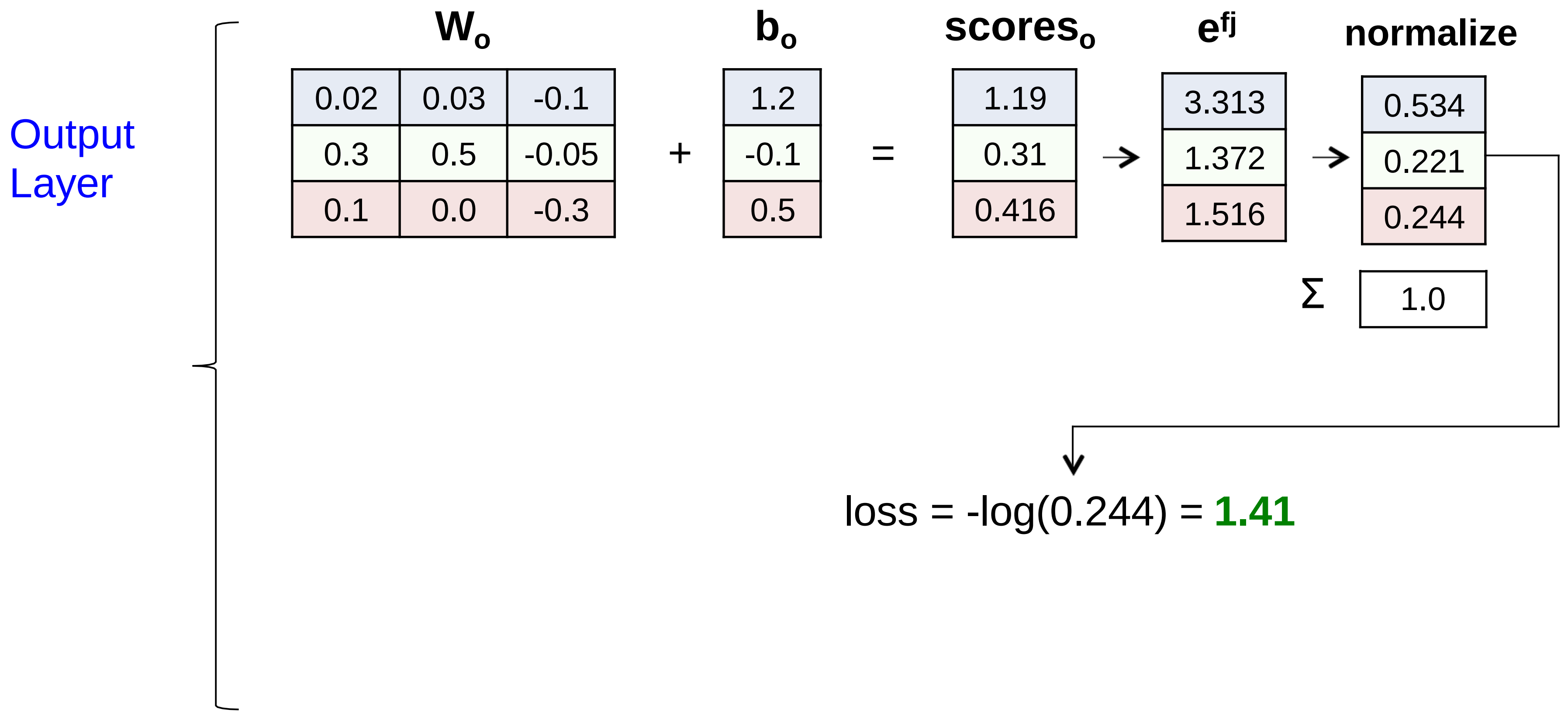
Score Function



Loss Function

- The loss function/gradients are slightly more complex
- We need to calculate the loss and gradients for the output layer first (in the same way as we did before)
- The gradients are then backpropagated into the hidden layer
- The loss for both layers are summed

Loss Function



Gradients

Output Layer

W_o

0.02	0.03	-0.1
0.3	0.5	-0.05
0.1	0.0	-0.3

+

b_o

1.2
-0.1
0.5

=

$scores_o$

1.19
0.31
0.416

→ e^{fj}

3.313
1.372
1.516

→ **normalize**

0.534
0.221
0.244

Σ 1.0

dscores

0.534
0.221
-0.756

Update the score for the correct category y_i by -1

Gradients

Output Layer

W_o + b_o = $scores_o$ → e^{f_j} → normalize

W_o

0.02	0.03	-0.1
0.3	0.5	-0.05
0.1	0.0	-0.3

b_o

1.2
-0.1
0.5

$scores_o$

1.19
0.31
0.416

e^{f_j}

3.313
1.372
1.516

normalize

0.534
0.221
0.244

Σ 1.0

dscores

0.534
0.221
-0.756

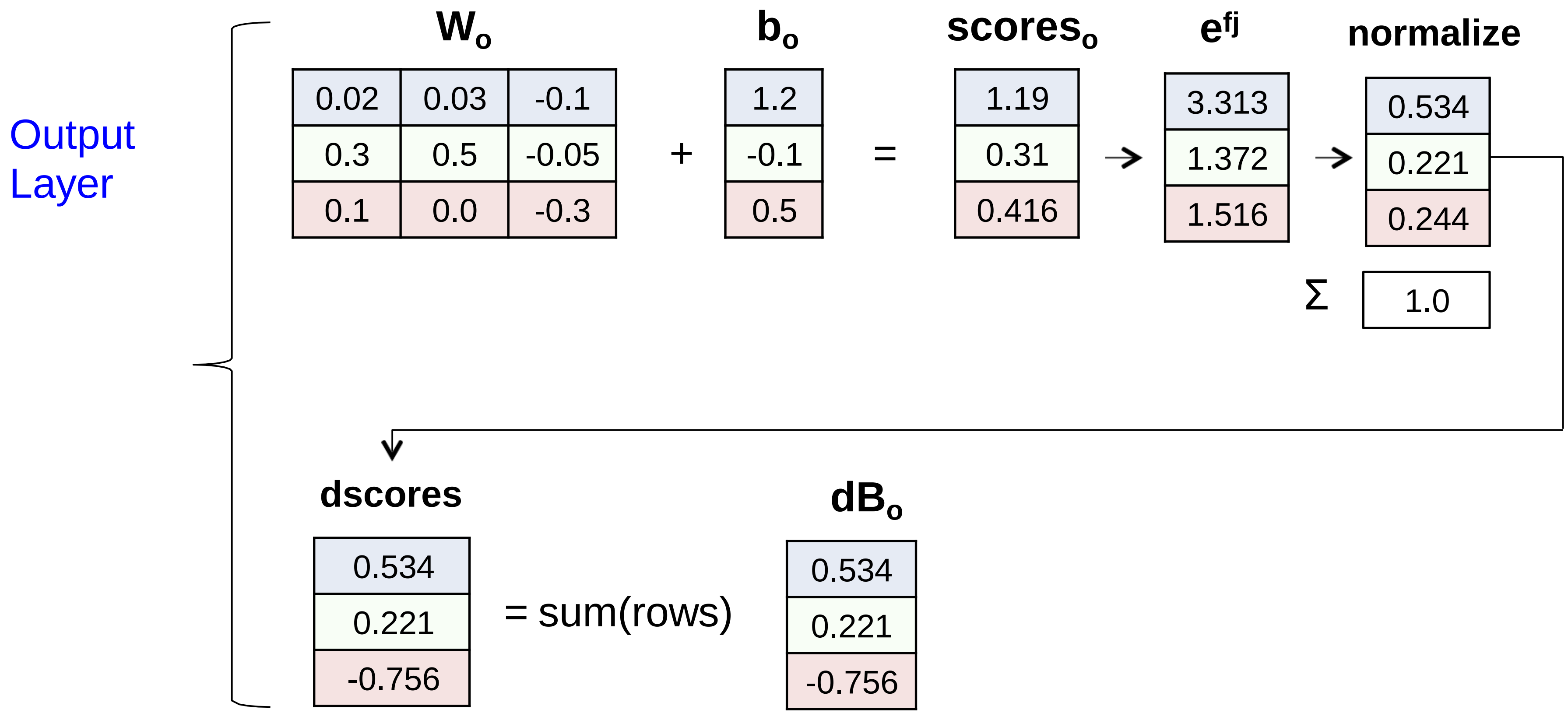
x_i^T

0	0.86	0.28

= dW_o

0.0	0.459	0.150
0.0	0.190	0.062
0.0	-0.650	-0,212

Gradients



Output
Layer
W_o
0.02 0.03 -0.1
0.3 0.5 -0.05
0.1 0.0 -0.3
+
b_o
1.2
-0.1
0.5
=
$scores_o$
1.19
0.31
0.416
e^{fj}
3.313
1.372
1.516
normalize
0.534
0.221
0.244
Σ
1.0
dscores
0.534
0.221
-0.756
= sum(rows)
dB_o
0.534
0.221
-0.756

Loss Function

Hidden Layer

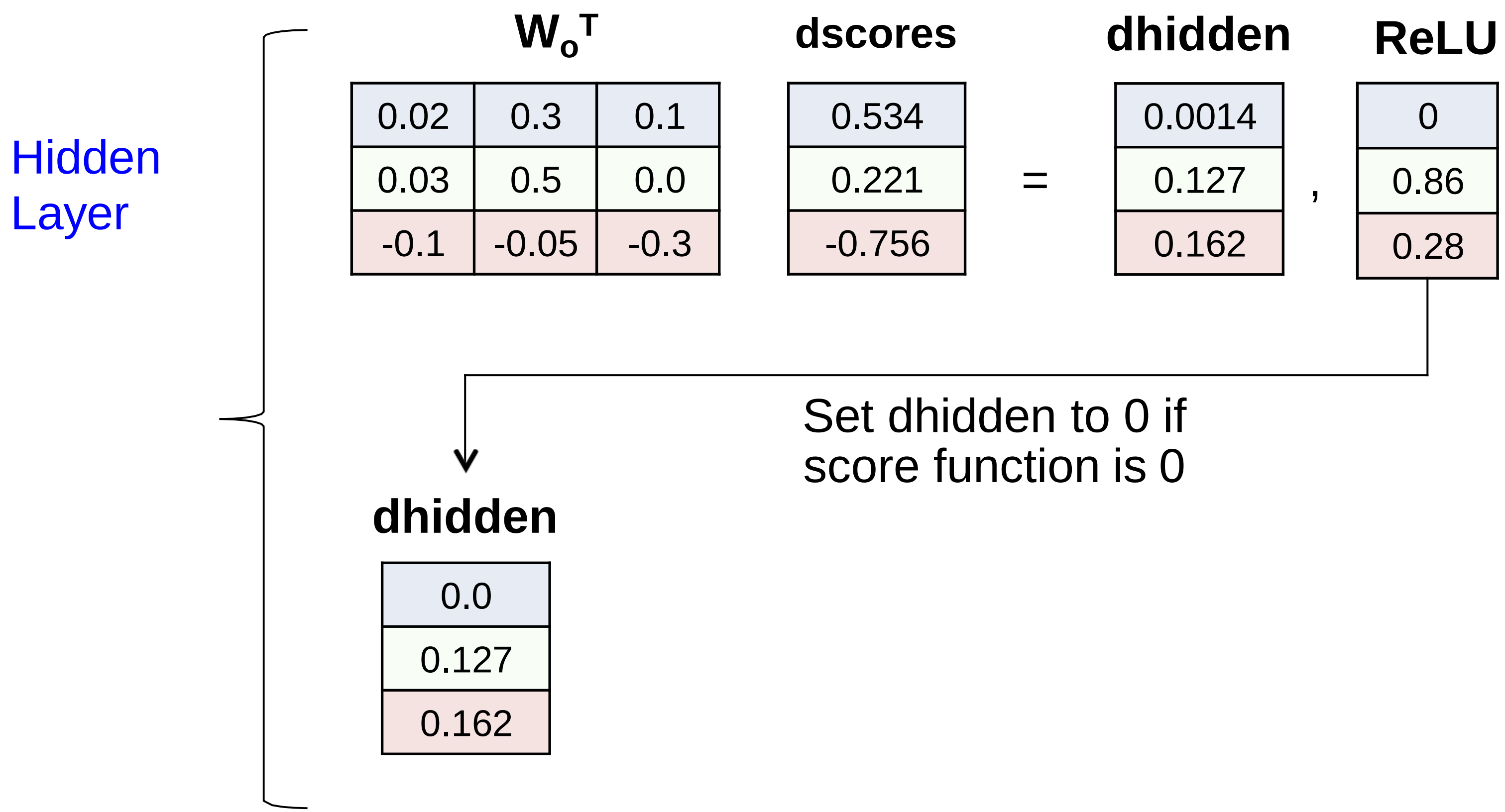
W_h

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

Loss is the L2 regularization = sum of all squared W_{ij}

loss = λ * 0.817 * 0.5 = **0.00408**

Gradients


Hidden
Layer
W$_o^T$
0.02
0.3
0.1
0.03
0.5
0.0
-0.1
-0.05
-0.3
dscores
0.534
0.221
-0.756
=
dhidden
0.0014
0.127
0.162
,
ReLU
0
0.86
0.28
Set dhidden to 0 if
score function is 0
dhidden
0.0
0.127
0.162

Gradients

Hidden Layer

W_o^T · dscores = dhidden , ReLU

W_o^T

0.02	0.3	0.1
0.03	0.5	0.0
-0.1	-0.05	-0.3

dscores

0.534
0.221
-0.756

=

dhidden

0.0014
0.127
0.162

,

ReLU

0
0.86
0.28

dhidden

0.0
0.127
0.162

x_i^T

-15	22	-44	56

=

dW_h

0.0	0.0	0.0	0.0
-1.90	2.79	-5.59	7.11
-2.43	3.56	7.13	9.07

Gradients

Hidden Layer

W_o^T

0.02	0.3	0.1
0.03	0.5	0.0
-0.1	-0.05	-0.3

dscores

0.534
0.221
-0.756

=

dhidden

0.0014
0.127
0.162

,

ReLU

0
0.86
0.28

dhidden

0.0
0.127
0.162

= sum(rows)

dB_h

0.0
0.127
0.162

Regularization

- Regularization is added to loss and gradients in the output and hidden layer as before
- The total loss is the loss for the output plus the loss for the hidden layer

Weights Upgrades

W_h

0.01	-0.05	0.1	0.05
0.7	0.2	0.05	0.16
0.0	-0.45	-0.2	0.03

-

dW_h

0.0	0.0	0.0	0.0
-1.90	2.79	-5.59	7.11
-2.43	3.56	7.13	9.07

*** 0.1** =

$newW_h$

=

0.01	-0.05	0.10	0.05
0.89	-0.08	0.61	-0.55
0.24	-0.81	0.51	-0.88

Bias Upgrades

b_h		dB_h		
0.0		0.0		
0.2	-	0.127	*** 0.1**	=
-0.3		0.162		

	$newB_h$
	0.0
=	0.19
	-0.32

Summary

- The linear classifier has now been extended to contain a hidden layer with ReLU nodes
- The hidden layer enables the classifier to learn categories that are not linearly separable

Non-linearly separable categories