Machine Learning: Linear Classification & Optimization

ROB 102: Introduction to AI & Programming

Lecture 12

2021/11/29

Project 4:Machine Learning

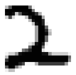
Implement three machine learning algorithms to classify images from the MNIST dataset.

1. Nearest neighbors
2. **Linear Classifier** (Today!)
3. Neural Network

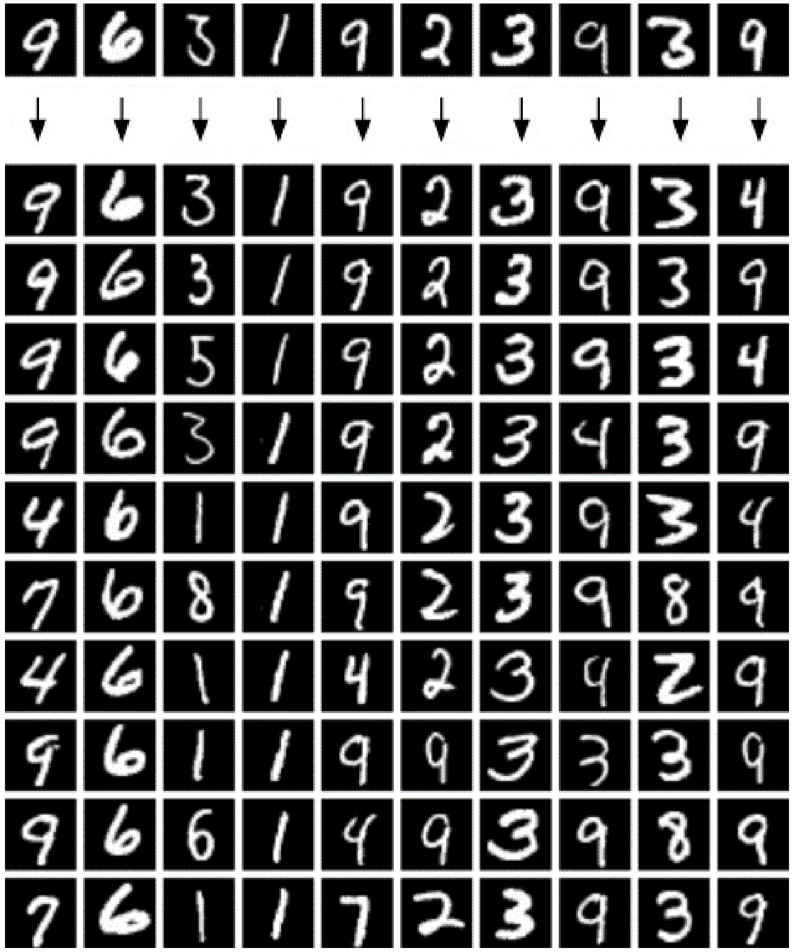
Last time…

We saw the (k-)Nearest Neighbor algorithm.

We assumed an image is numerically close to other images in the same class.

distance( , )

At training time, we saved ALL our training data, and calculated distances at test time.



Summary: Nearest Neighbors

Pros:

+ Straight-forward to implement
+ No training necessary
+ "Pretty good" for many problems

Cons:

– Requires a lot of memory
– Expensive at test time
– Distance isn't always a good indicator of class similarity
– We need many training examples to make a good classifier for high dimensions (like images!)

Class Scores

What if we had a function that told us how “two-like” an image is?

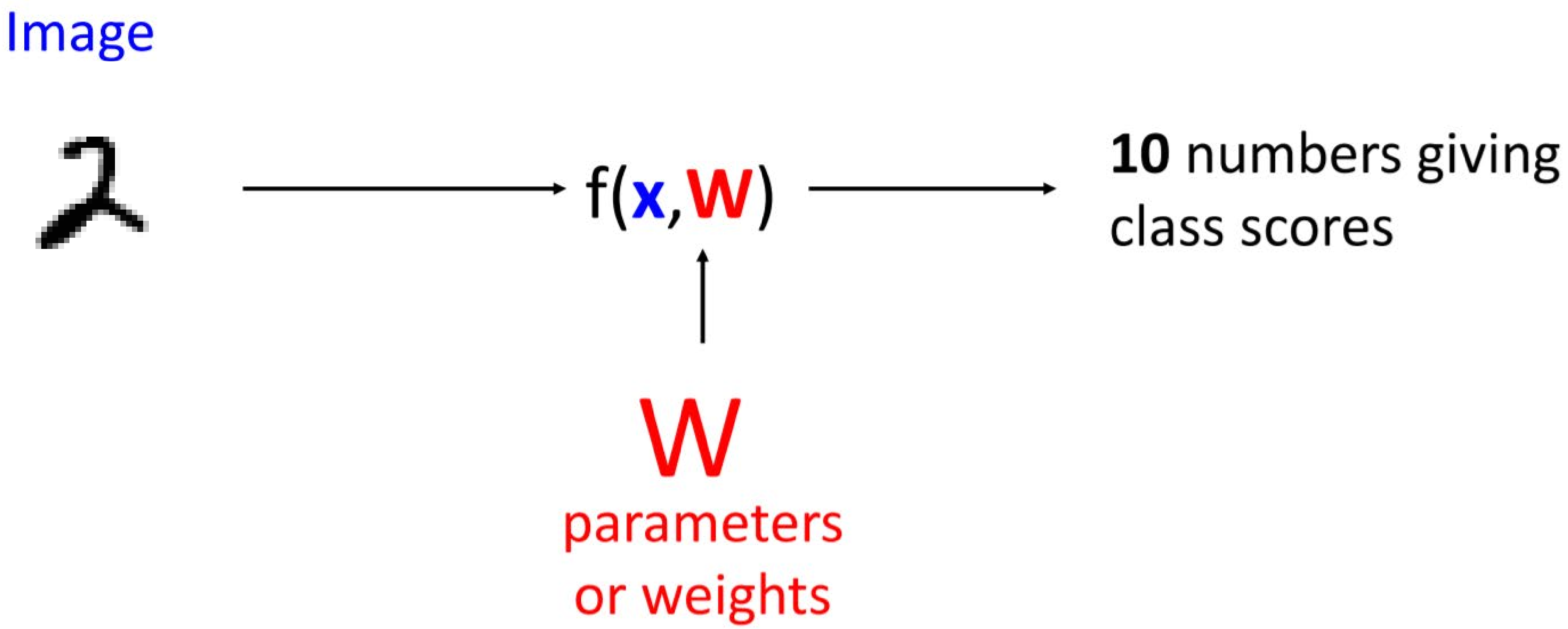
$f_2($  $) = \text{score}$

The score should be HIGH the image is a two and LOW if the image is probably not a two.

Most modern machine learning algorithms do classification like this.

Class Scores

Now, we only need to learn the parameters of a function using our training data. We don't need to save the training data anymore.



Image

$f(\mathbf{x},\mathbf{W})$

10 numbers giving class scores

W

parameters or weights

Recall: Machine Learning Algorithm

Training time:

Find a function $f(X)$ which does well at classifying training data.

Testing time:

Use $f(X)$ to classify new data.

Recall: Machine Learning Algorithm

Training time:

Find a function $f(X)$ which does well at classifying labelled data.

Testing time:

Use $f(X)$ to classify new data.

How do we choose what $f(X)$ should look like?

How do we learn $f(X)$?

Recall: Machine Learning Algorithm

Training time:

Find a function f(X) which does well at classifying labelled data.

Testing time:

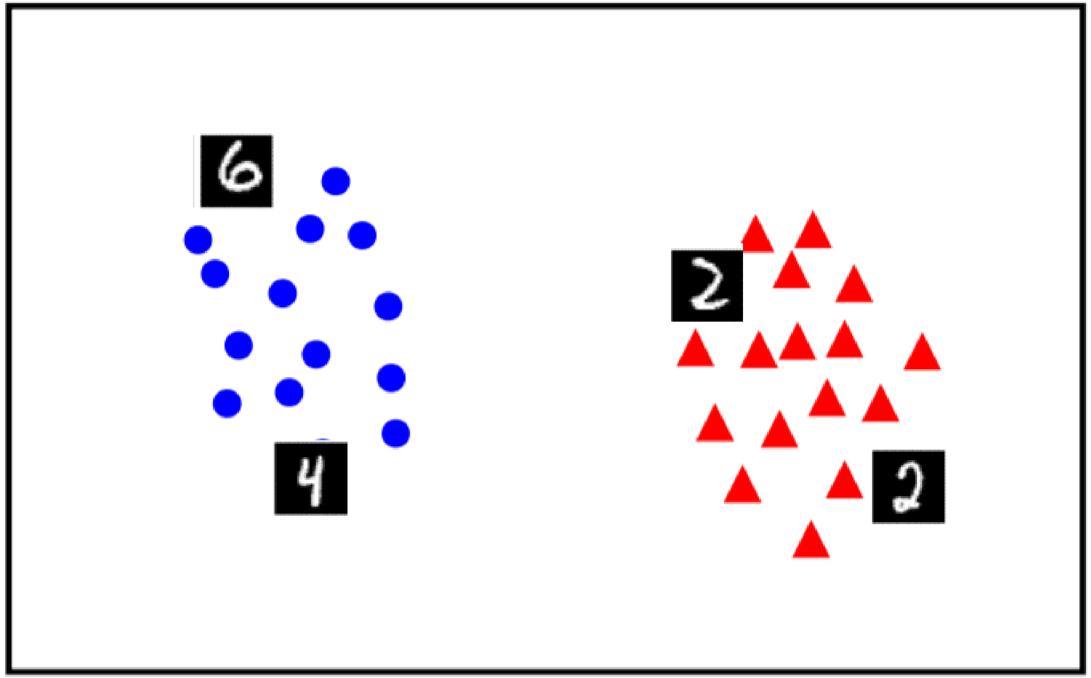
Use f(X) to classify new data.

How do we choose what $f(X)$ should look like?

- Next: Use a linear function

Linear Classification

Let’s look at a 2-D example. We want to classify our points into two categories:



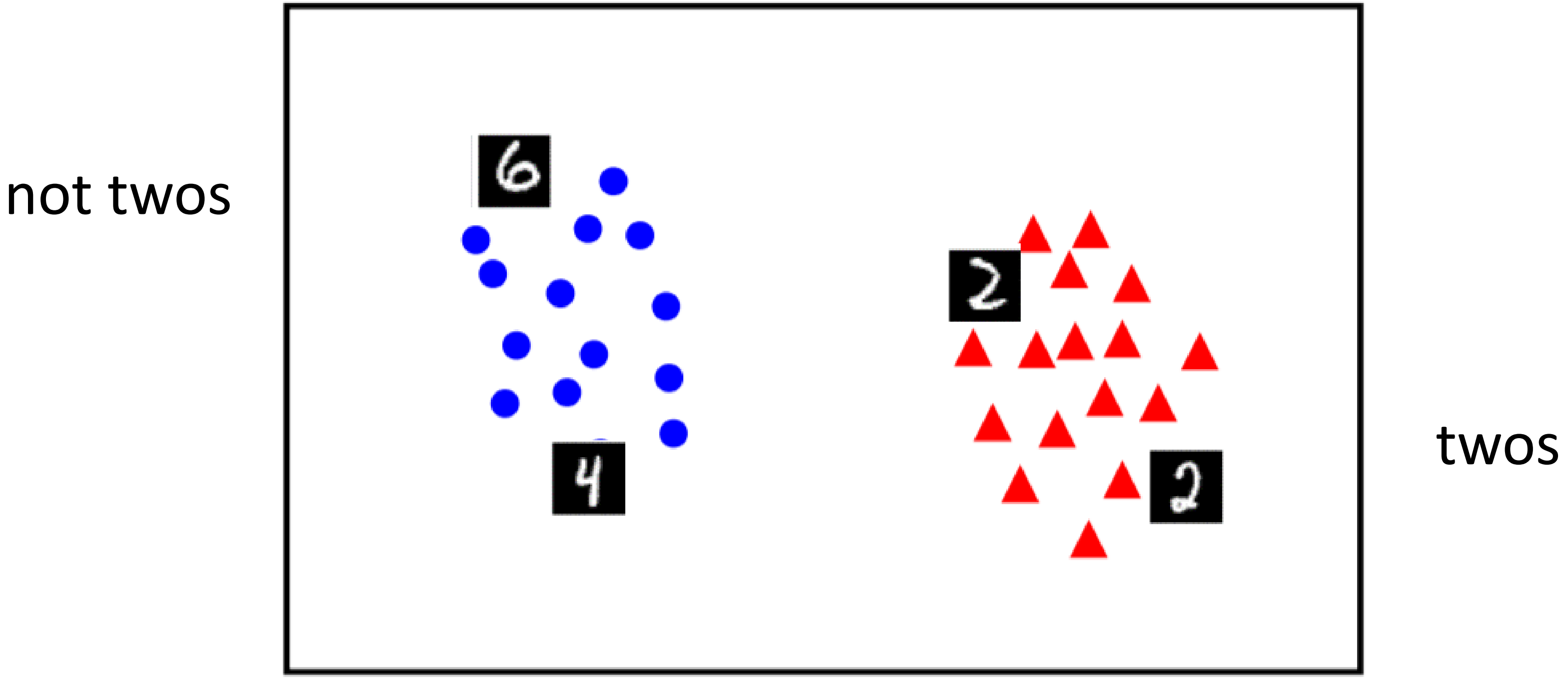
not twos

twos

Example Credit: (link)

Linear Classification

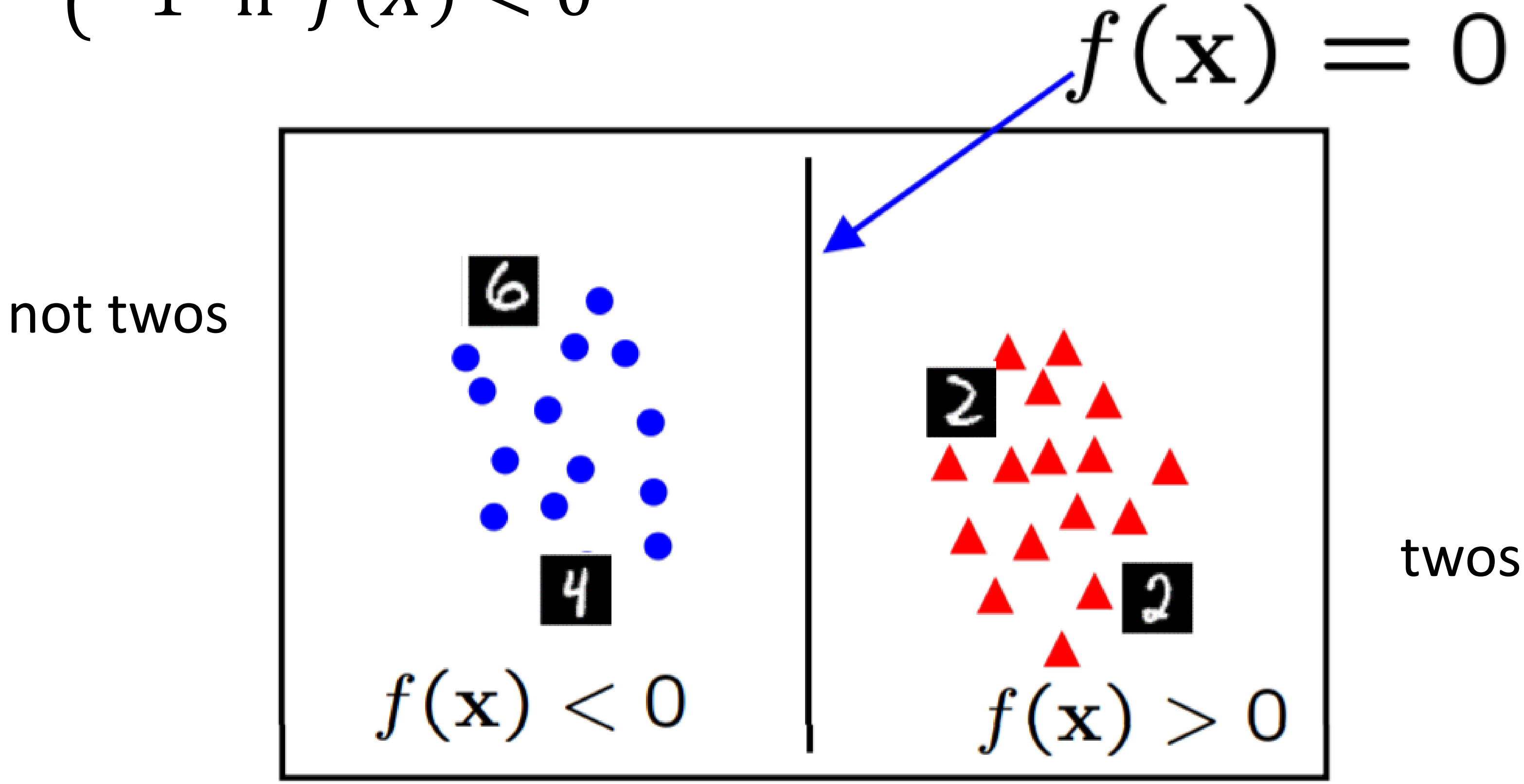
We need a function, $f(X)$ which will help us find a label, y_{pred}, which is 1 if the image is of a two, and -1 if it is not a two.



Example Credit: (link)

Linear Classification

$$y_{pred} = \begin{cases} 1 & \text{if } f(X) \geq 0 \\ -1 & \text{if } f(X) < 0 \end{cases}$$

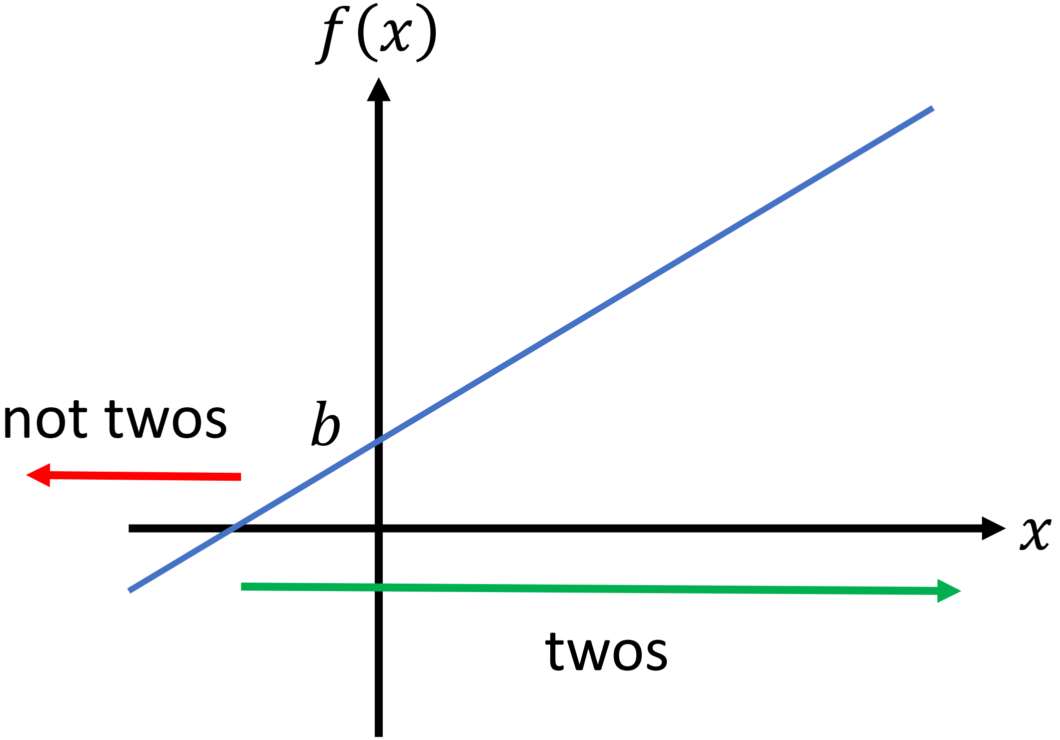


Example Credit: (link)

Linear Classification

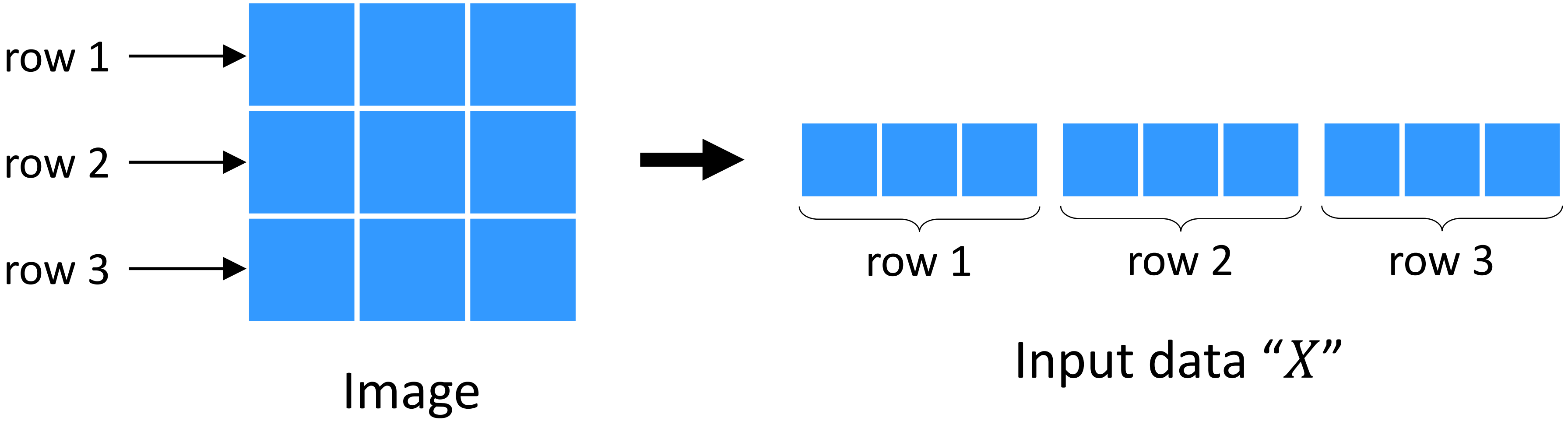
What does this look like in code? Remember, our plan is to use a linear function. In 1-D, this looks like this:

$$f(x) = wx + b$$



Images as Vectors

We will rearrange our images into long vectors by flattening them. The vectors will have length W*H.

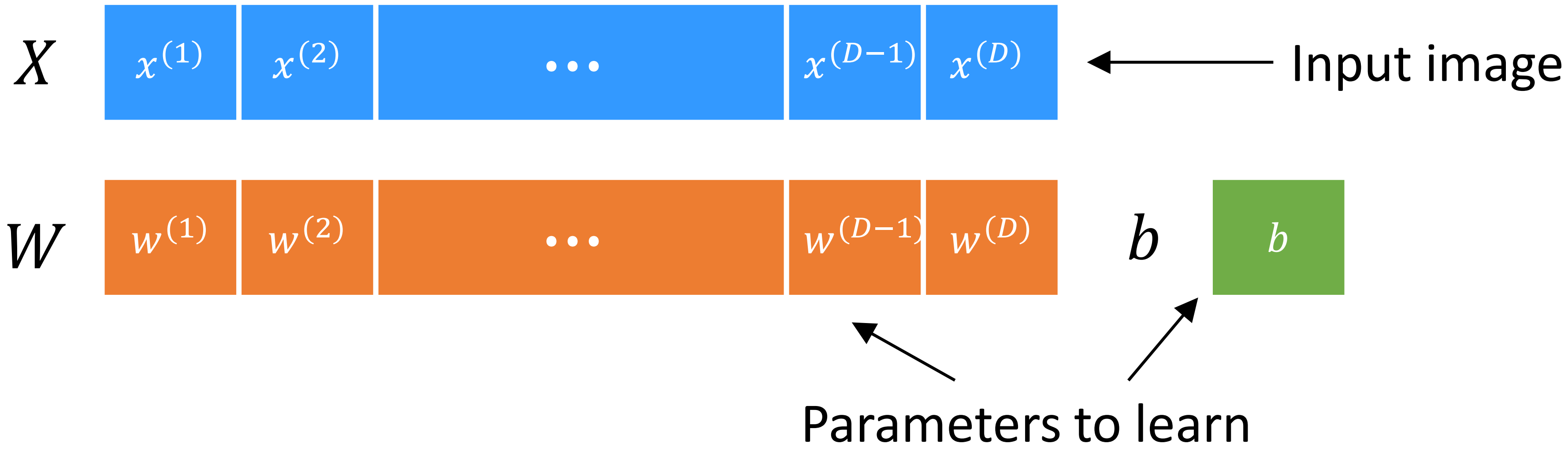


For MNIST images, vectors will have length 28*28 = 784.

Linear Classification

For our image, with dimension D = 28*28:

$$f(X) = w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + \cdots + w^{(D-1)}x^{(D-1)} + w^{(D)}x^{(D)} + b$$



X $x^{(1)}$ $x^{(2)}$ $\cdots$ $x^{(D-1)}$ $x^{(D)}$ ← Input image

W $w^{(1)}$ $w^{(2)}$ $\cdots$ $w^{(D-1)}$ $w^{(D)}$ b b

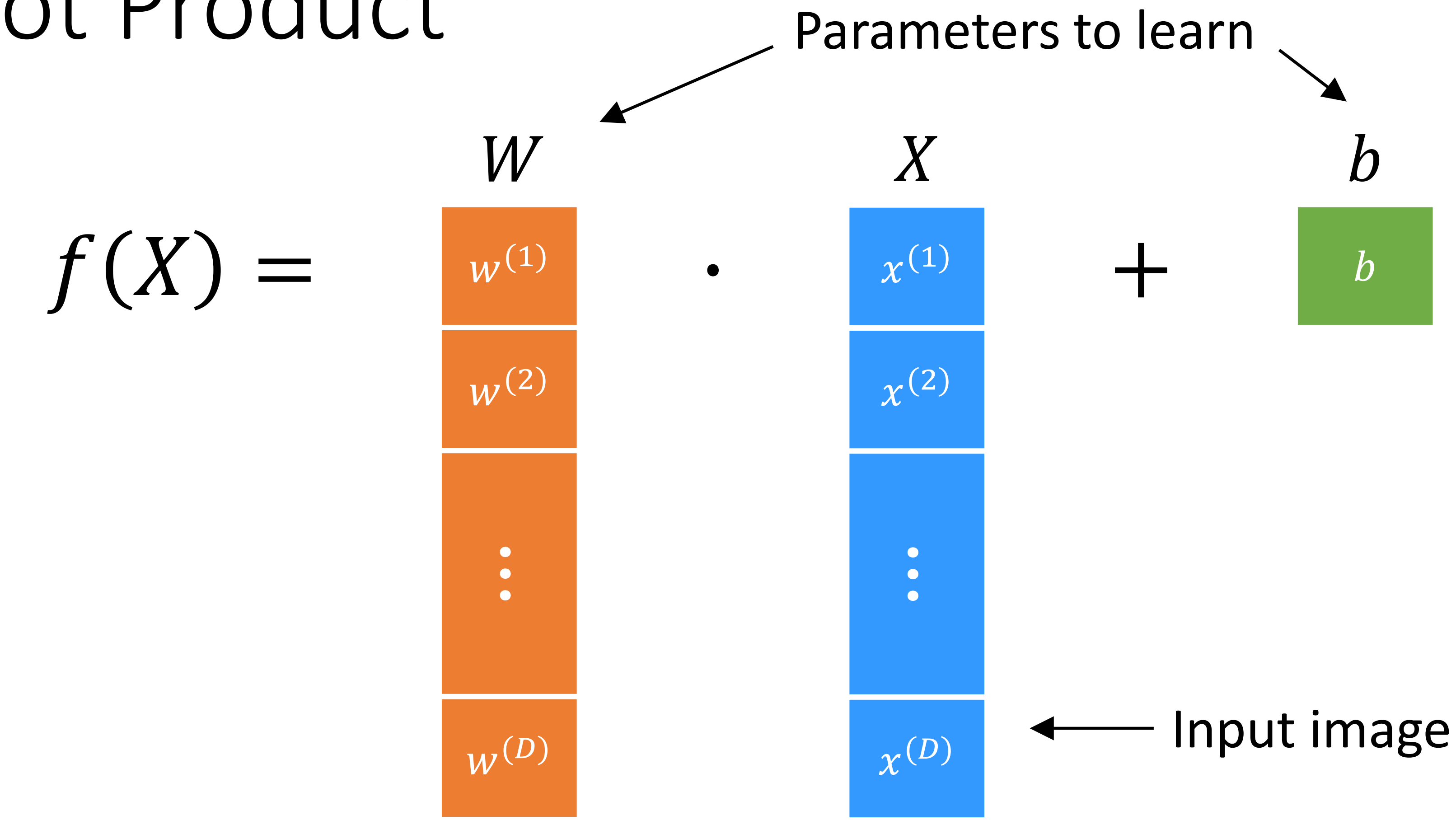
Parameters to learn

Dot Product

A convenient way of writing this is called the dot product of vectors:

$$f(X) = W \underset{\substack{\uparrow \\ \text{Dot product}}}{\cdot} X + b$$

Dot Product



Parameters to learn

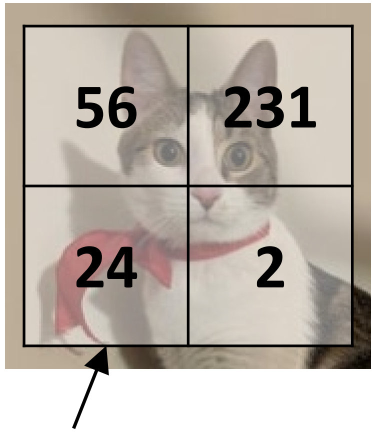
$$f(X) = \overset{W}{\begin{bmatrix} w^{(1)} \\ w^{(2)} \\ \vdots \\ w^{(D)} \end{bmatrix}} \cdot \overset{X}{\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(D)} \end{bmatrix}} + \overset{b}{\boxed{b}}$$

Input image

$$= w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + \cdots + w^{(D)}x^{(D)} + b$$

Example: Dot Product

Image:



Pixel values

Parameters:

W

0.2

-0.5

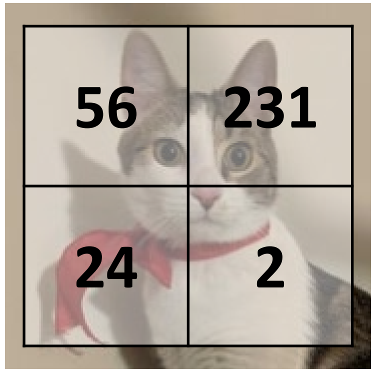
0.1

2.0

b

1.1

Example: Dot Product



$$\overset{W}{\begin{bmatrix}0.2\\-0.5\\0.1\\2.0\end{bmatrix}} \cdot \overset{X}{\begin{bmatrix}56\\231\\24\\2\end{bmatrix}} + \overset{b}{\begin{bmatrix}1.1\end{bmatrix}}$$

$$= 0.2 \cdot 56 + -0.5 \cdot 231 + 0.1 \cdot 24 + 2.0 \cdot 2 + 1.1$$

$$= -96.8$$

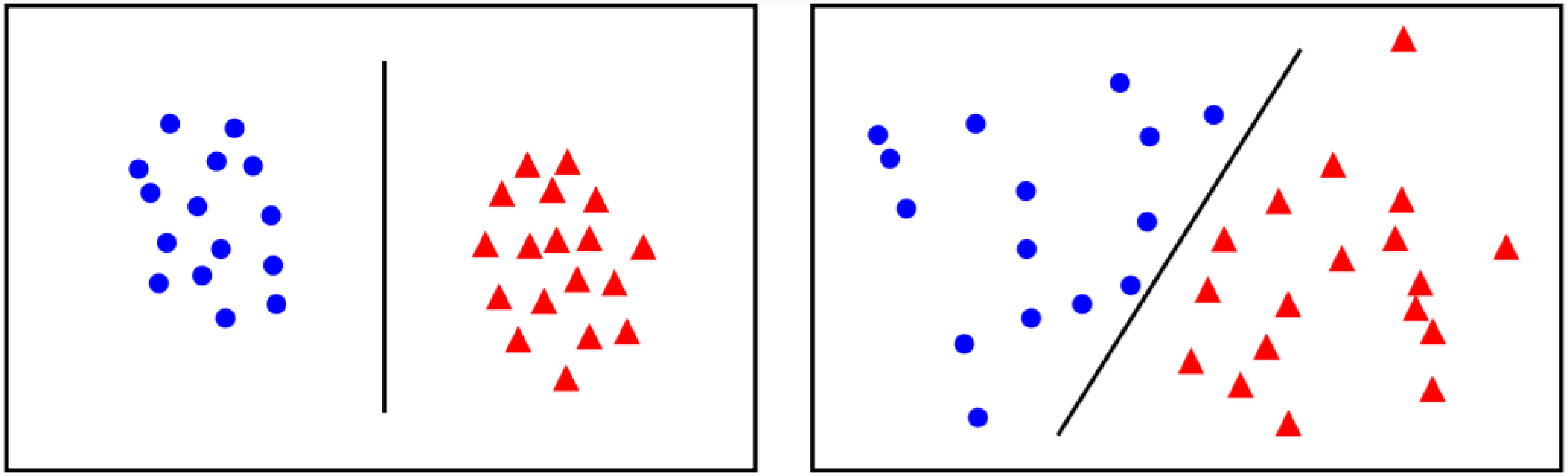
Linear separability

Let’s go back and take a look at our choice of model, $f(X)$. What does it mean to use a linear model?

$$f(X) = W \cdot X + b$$

Linear separability

If we can draw a straight line through our data, then we say it is <u>linearly separable</u>.



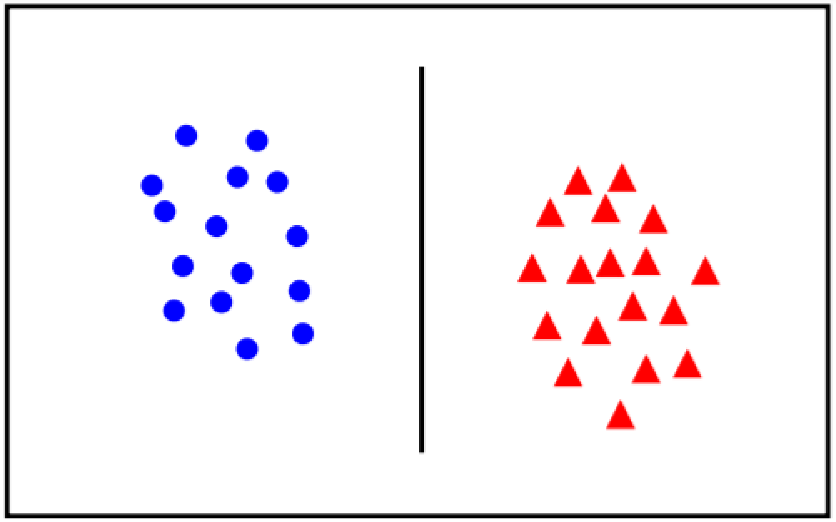
Linearly separable

Linearly separable

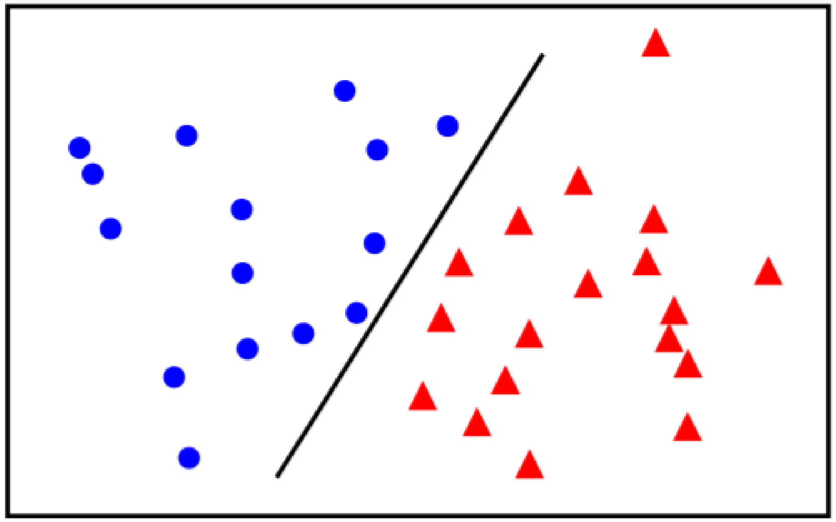
Example Credit: (link)

Linear separability

If we can draw a straight line through our data, then we say it is <u>linearly separable</u>.



Linearly separable



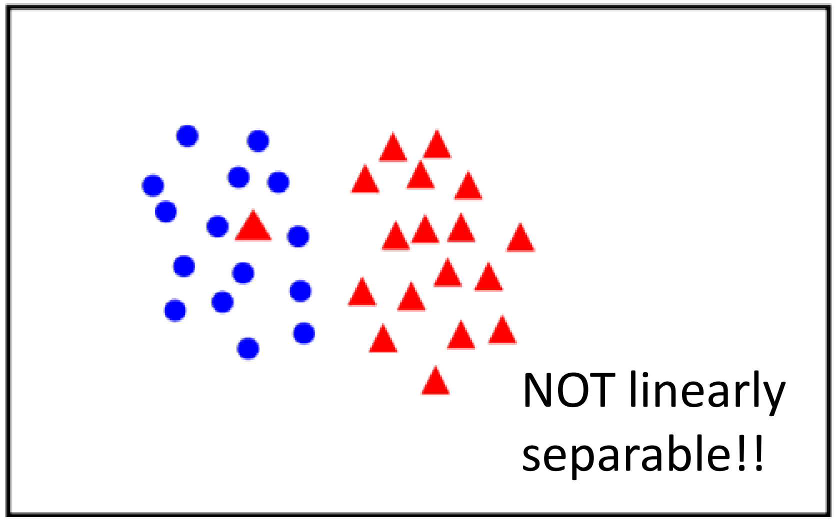
Linearly separable

Linear classifier assumption: The data is <u>linearly separable</u>.

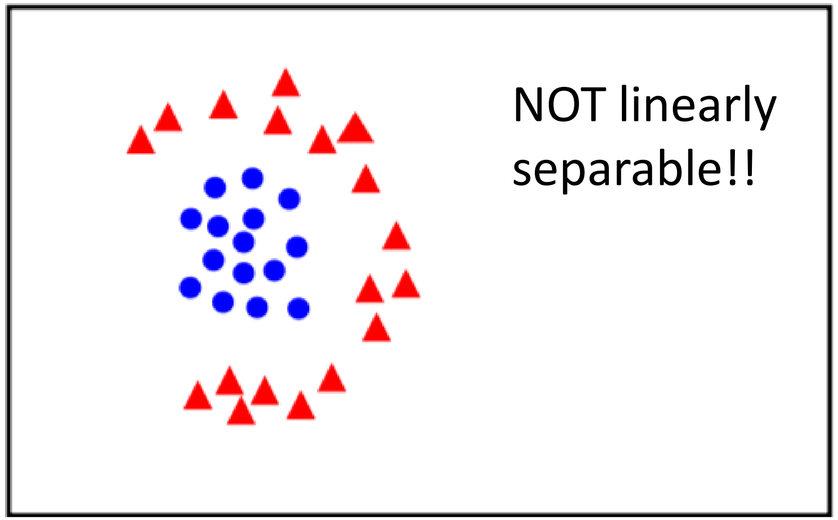
Example Credit: (link)

Linear separability

The problem with assumptions…



NOT linearly separable!!



NOT linearly separable!!

“All models are wrong, but some models are useful.”

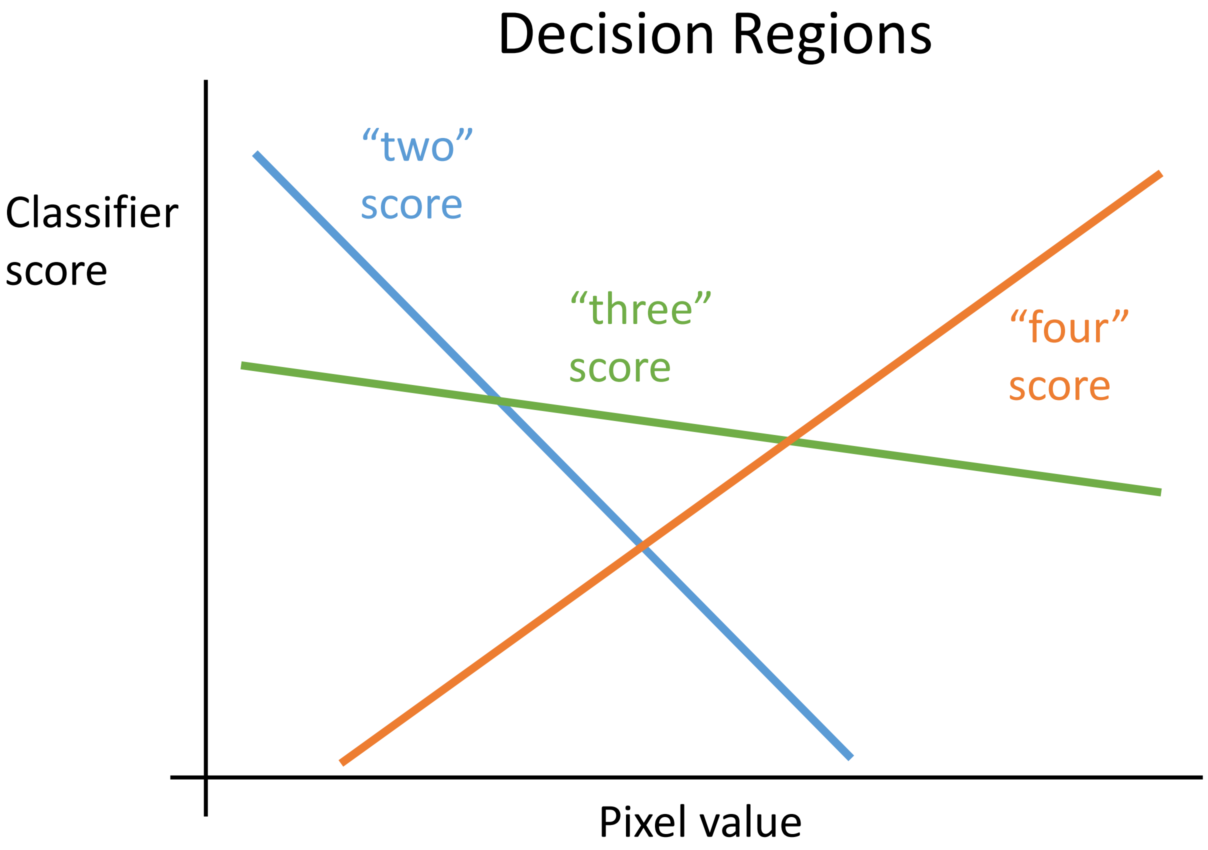
Translation: Let’s assume our linear model is “good enough”

Example Credit: (link)

Multiple Classes

In MNIST we have 10 classes, so we learn 10 classifiers (10 weight vectors).

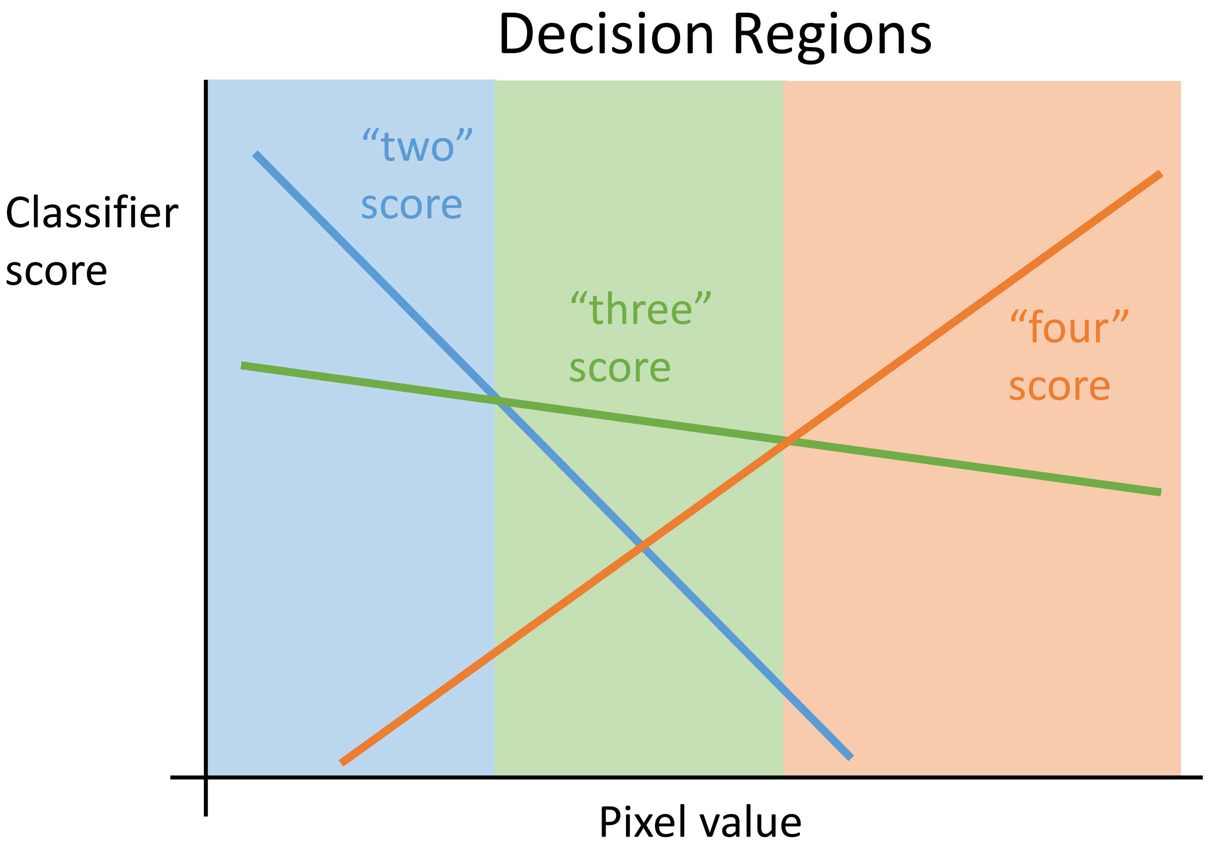
Each classifier gives a score for a class. We predict that the image belongs to the class that gives it the highest score.



Multiple Classes

In MNIST we have 10 classes, so we learn 10 classifiers (10 weight vectors).

Each classifier gives a score for a class. We predict that the image belongs to the class that gives it the highest score.



Multiple Classes

If we have K classes, we can find K linear functions (K weight vectors and biases):

$$f_1(X) = W_1 \cdot X + b_1$$

$$f_2(X) = W_2 \cdot X + b_2$$

$$\vdots$$

$$f_K(X) = W_K \cdot X + b_K$$

Linear Classifier Algorithm

Training: Find weights W and bias b that do well at classifying training data

Testing: For each class i, do:

$$f_i(X) = W_i \cdot X + b_i$$

Assign label:

$$y_{pred} = \underset{i}{\operatorname{argmax}} f_i(X)$$

airplane	-3.45	-0.51	3.42
automobile	-8.87	y_{pred} $\boxed{\mathbf{6.04}}$	4.64
bird	0.09	5.31	2.65
cat	**2.9**	-4.22	5.1
deer	4.48	-4.19	2.64
dog	y_{pred} $\boxed{8.02}$	3.58	5.55
frog	3.78	4.49	**-4.34**
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	y_{pred} $\boxed{6.14}$

Image Credit: Johnson (link)

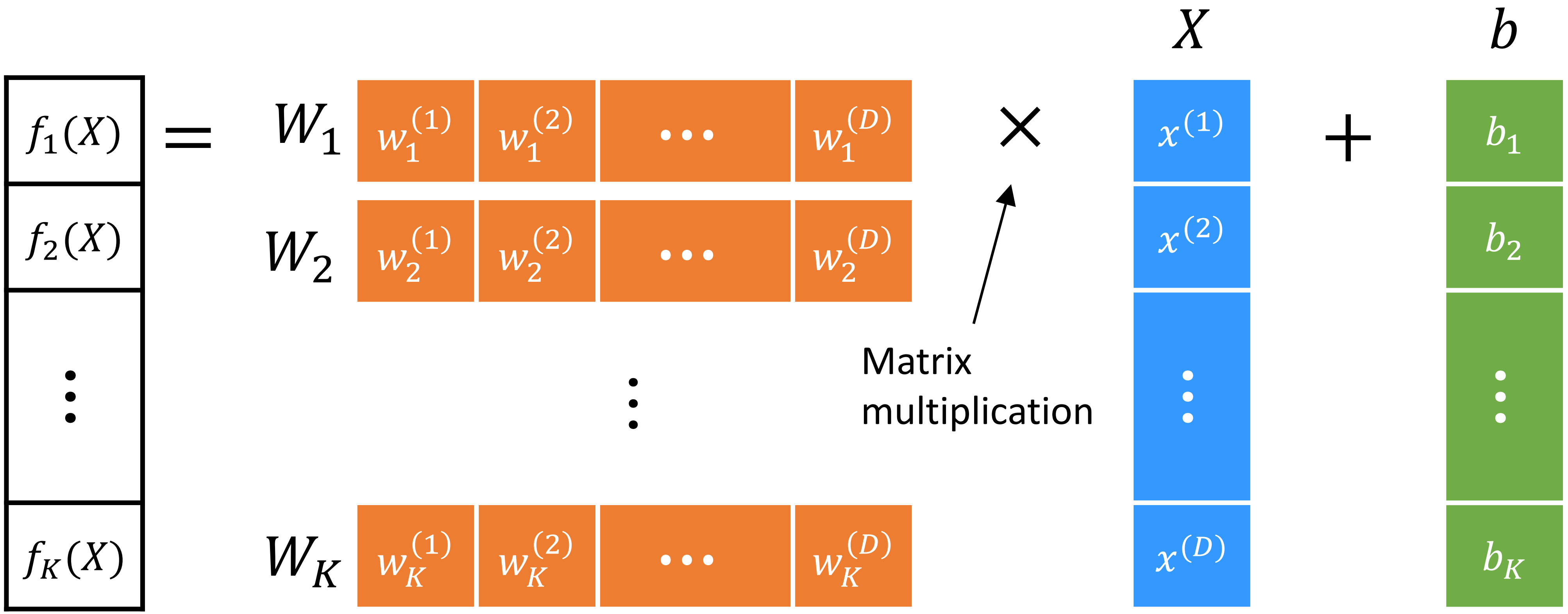
Matrix Multiplication

For the multiple class case, we need to perform K dot products between the weight vectors and images.

$$\begin{bmatrix} f_1(X) \\ f_2(X) \\ \vdots \\ f_K(X) \end{bmatrix} = \begin{bmatrix} W_1 \cdot X + b_1 \\ W_2 \cdot X + b_2 \\ \vdots \\ W_K \cdot X + b_K \end{bmatrix}$$

Matrix multiplication allows us to do this operation in one step.

Matrix Multiplication



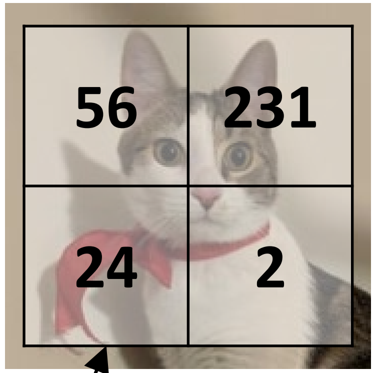
$$\begin{bmatrix} f_1(X) \\ f_2(X) \\ \vdots \\ f_K(X) \end{bmatrix} = \begin{array}{c} W_1 \\ W_2 \\ \\ W_K \end{array} \begin{bmatrix} w_1^{(1)} & w_1^{(2)} & \cdots & w_1^{(D)} \\ w_2^{(1)} & w_2^{(2)} & \cdots & w_2^{(D)} \\ & \vdots & & \\ w_K^{(1)} & w_K^{(2)} & \cdots & w_K^{(D)} \end{bmatrix} \times \overset{X}{\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(D)} \end{bmatrix}} + \overset{b}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}}$$

Matrix multiplication

Matrix multiplication takes the dot product between each row of the first matrix with each column of the second matrix.

Example: Matrix Multiplication

Image:



Pixel values

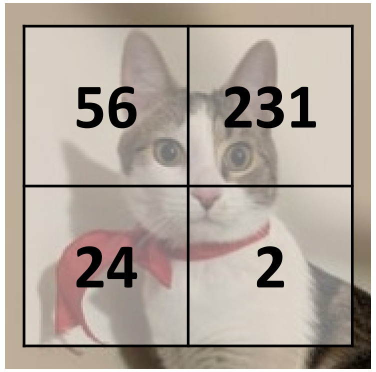
Parameters:

W_{cat}	0.2	-0.5	0.1	2.0
W_{dog}	-0.2	0.0	0.4	1.3
W_{bird}	1.1	-0.7	0.6	-0.2

b_{cat} 1.1 b_{dog} -0.3

b_{bird} 0.9

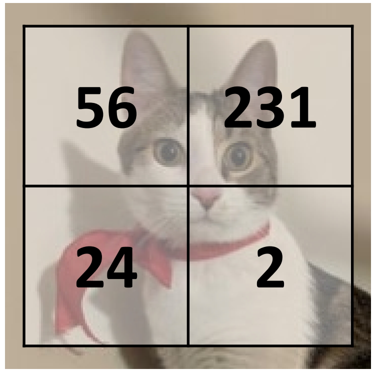
Example: Linear Classification



$$\begin{array}{c} W_{cat} \\ W_{dog} \\ W_{bird} \end{array} \begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ -0.2 & 0.0 & 0.4 & 1.3 \\ 1.1 & -0.7 & 0.6 & -0.2 \end{bmatrix} \times \overset{X}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \begin{bmatrix} 1.1 \\ -0.3 \\ 0.9 \end{bmatrix} \begin{array}{c} b_{cat} \\ b_{dog} \\ b_{bird} \end{array}$$

$$\begin{array}{|c|} \hline f_{cat}(X) \\ \hline f_{dog}(X) \\ \hline f_{bird}(X) \\ \hline \end{array}$$

Example: Linear Classification



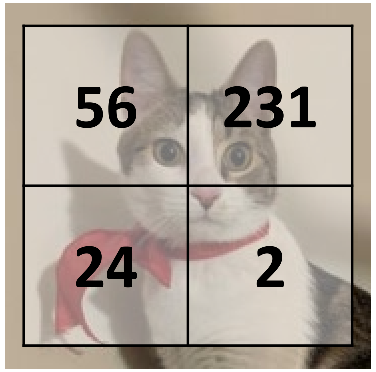
$$\begin{array}{c|cccc} W_{cat} & 0.2 & -0.5 & 0.1 & 2.0 \\ W_{dog} & -0.2 & 0.0 & 0.4 & 1.3 \\ W_{bird} & 1.1 & -0.7 & 0.6 & -0.2 \end{array} \times \overset{X}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \begin{array}{c|c} 1.1 & b_{cat} \\ -0.3 & b_{dog} \\ 0.9 & b_{bird} \end{array}$$

$f_{cat}(X)$	-96.8
$f_{dog}(X)$	
$f_{bird}(X)$	

$$f_{cat}(X) = 0.2 \cdot 56 + -0.5 \cdot 231 + 0.1 \cdot 24 + 2.0 \cdot 2 + 1.1$$

$$= -96.8$$

Example: Linear Classification



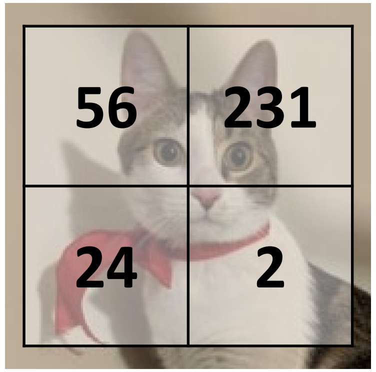
$$\begin{array}{c} W_{cat} \\ W_{dog} \\ W_{bird} \end{array} \begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ -0.2 & 0.0 & 0.4 & 1.3 \\ 1.1 & -0.7 & 0.6 & -0.2 \end{bmatrix} \times \overset{X}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \begin{bmatrix} 1.1 \\ -0.3 \\ 0.9 \end{bmatrix} \begin{array}{c} b_{cat} \\ b_{dog} \\ b_{bird} \end{array}$$

$f_{cat}(X)$	-96.8
$f_{dog}(X)$	0.7
$f_{bird}(X)$	

$$f_{dog}(X) = -0.2 \cdot 56 + 0.0 \cdot 231 + 0.4 \cdot 24 + 1.3 \cdot 2 + -0.3$$

$$= 0.7$$

Example: Linear Classification



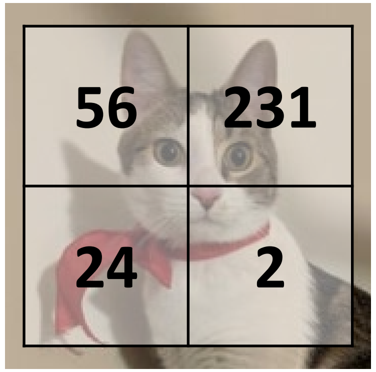
$$\begin{array}{c|cccc} W_{cat} & 0.2 & -0.5 & 0.1 & 2.0 \\ W_{dog} & -0.2 & 0.0 & 0.4 & 1.3 \\ W_{bird} & 1.1 & -0.7 & 0.6 & -0.2 \end{array} \times \overset{X}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \begin{array}{c|c} 1.1 & b_{cat} \\ -0.3 & b_{dog} \\ 0.9 & b_{bird} \end{array}$$

$f_{cat}(X)$	-96.8
$f_{dog}(X)$	0.7
$f_{bird}(X)$	-85.2

$$f_{bird}(X) = 1.1 \cdot 56 + -0.7 \cdot 231 + 0.6 \cdot 24 + -0.2 \cdot 2 + 0.9$$

$$= -85.2$$

Example: Linear Classification



$$\begin{array}{c|cccc} W_{cat} & 0.2 & -0.5 & 0.1 & 2.0 \\ W_{dog} & -0.2 & 0.0 & 0.4 & 1.3 \\ W_{bird} & 1.1 & -0.7 & 0.6 & -0.2 \end{array} \times \overset{X}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \begin{array}{c|c} 1.1 & b_{cat} \\ -0.3 & b_{dog} \\ 0.9 & b_{bird} \end{array}$$

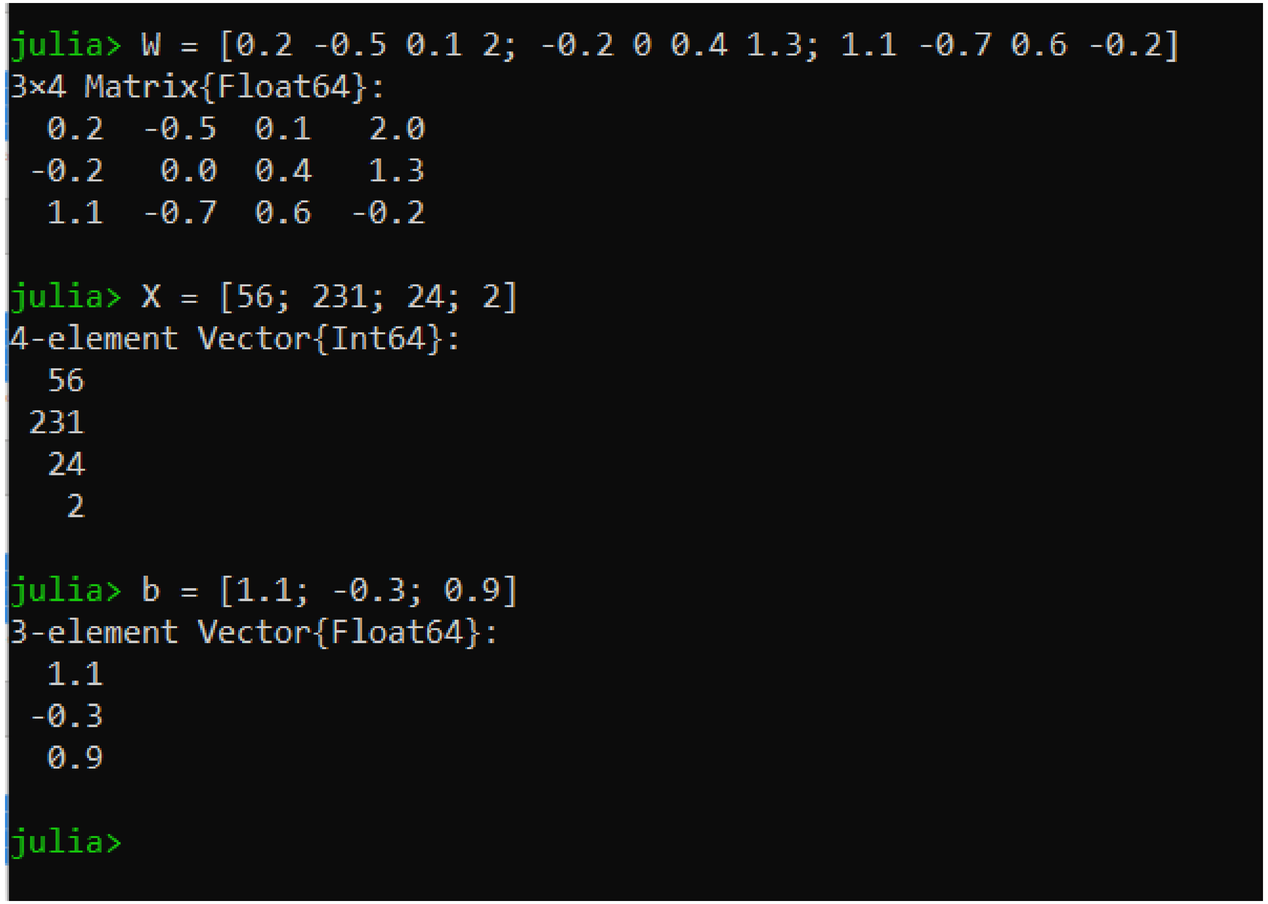
$f_{cat}(X)$ = -96.8

$f_{dog}(X)$ = $\boxed{0.7}$

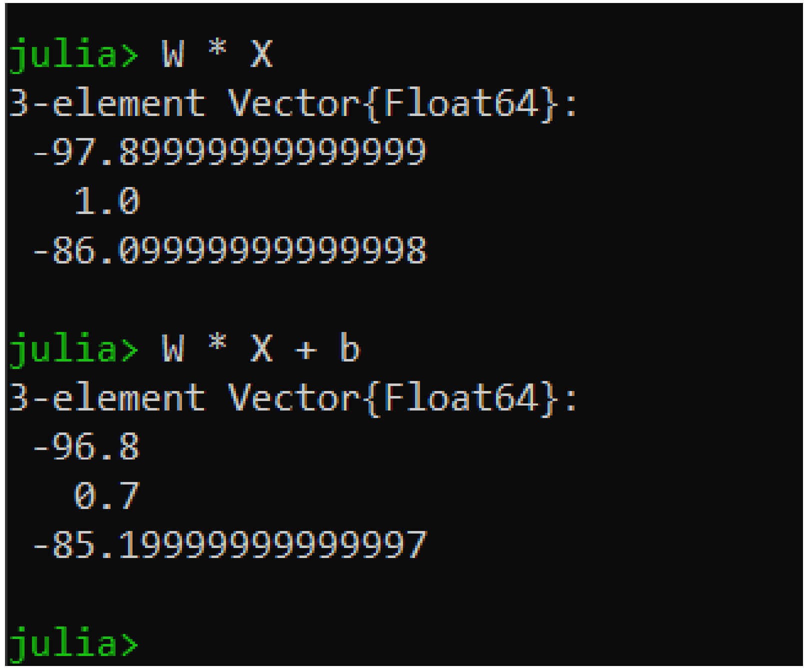
$f_{bird}(X)$ = -85.2

Prediction: Dog!

Matrix Multiplication in Julia



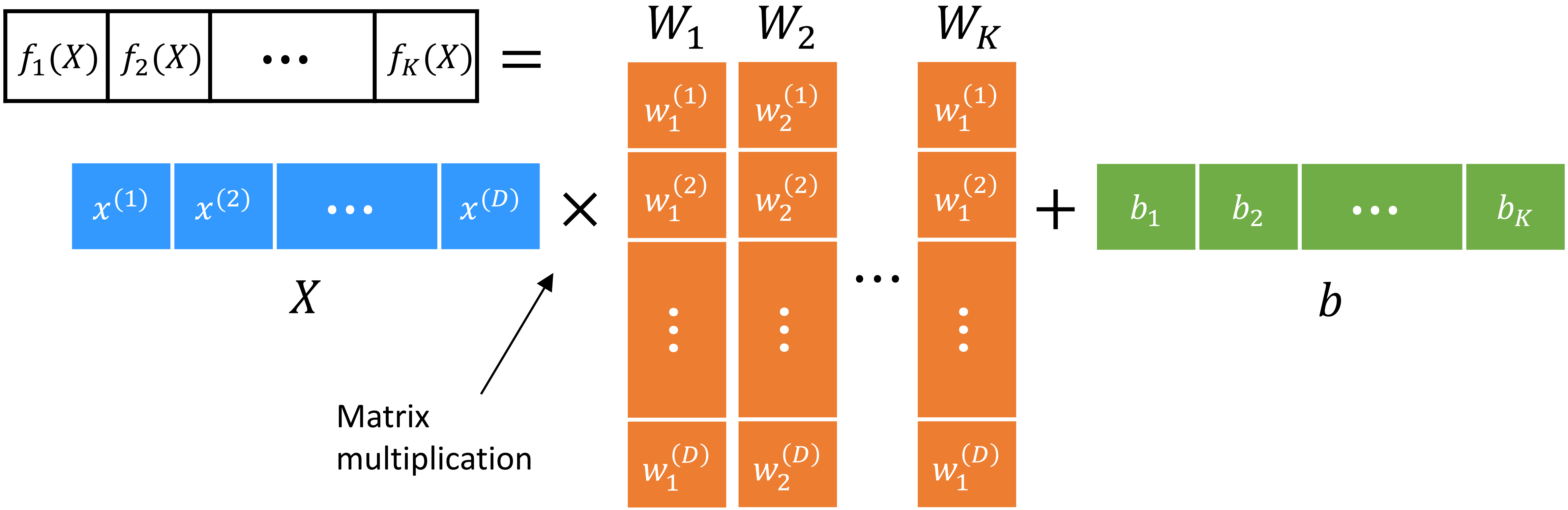
```
julia> W = [0.2 -0.5 0.1 2; -0.2 0 0.4 1.3; 1.1 -0.7 0.6 -0.2]
3×4 Matrix{Float64}:
  0.2  -0.5  0.1   2.0
 -0.2   0.0  0.4   1.3
  1.1  -0.7  0.6  -0.2

julia> X = [56; 231; 24; 2]
4-element Vector{Int64}:
  56
 231
  24
   2

julia> b = [1.1; -0.3; 0.9]
3-element Vector{Float64}:
  1.1
 -0.3
  0.9

julia>
```



```
julia> W * X
3-element Vector{Float64}:
 -97.89999999999999
   1.0
 -86.09999999999998

julia> W * X + b
3-element Vector{Float64}:
 -96.8
   0.7
 -85.19999999999997

julia>
```

The * operation between two matrices is a matrix multiplication in Julia.

Project 4: Matrix Multiplication

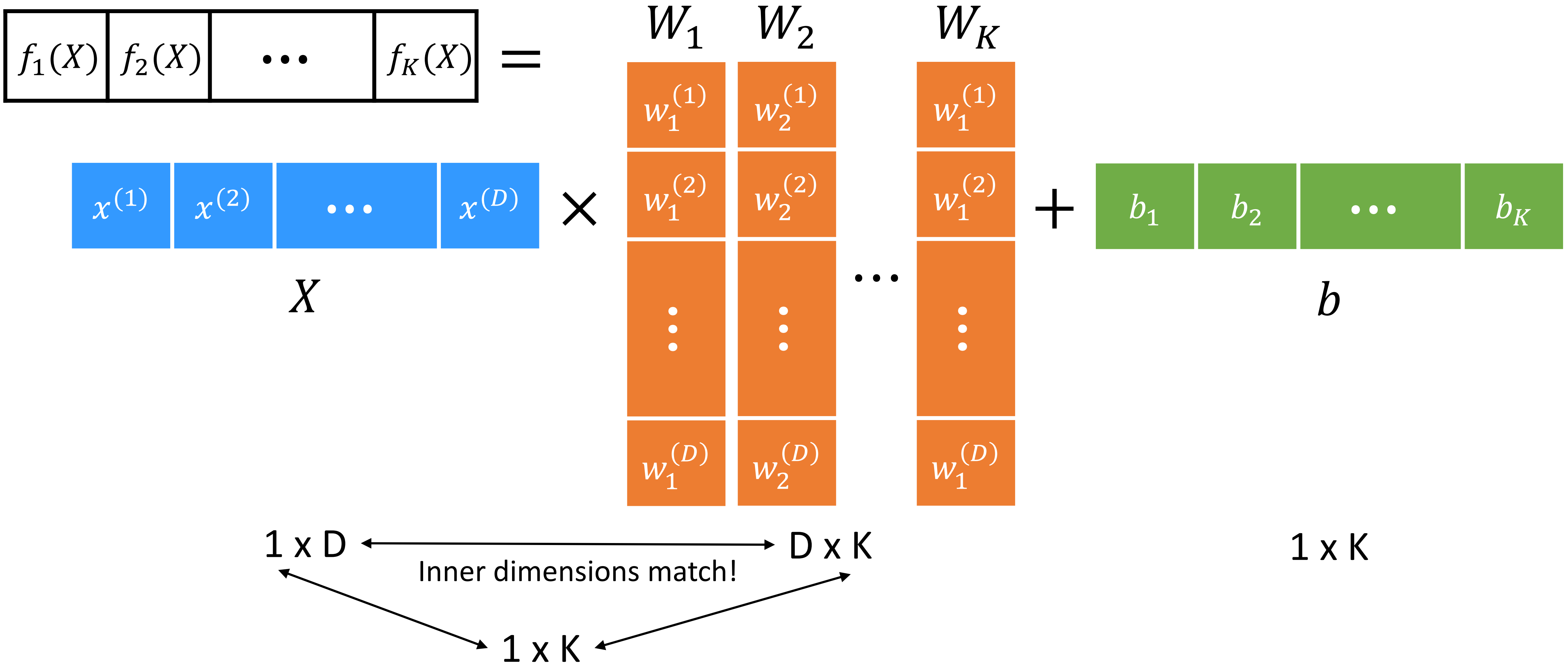


Matrix multiplication

Note: This will give us the same answer, $X \cdot W_i$ with instead of $W_i \cdot X$. Project 4 uses this representation (images stacked in rows).

Matrix Shapes

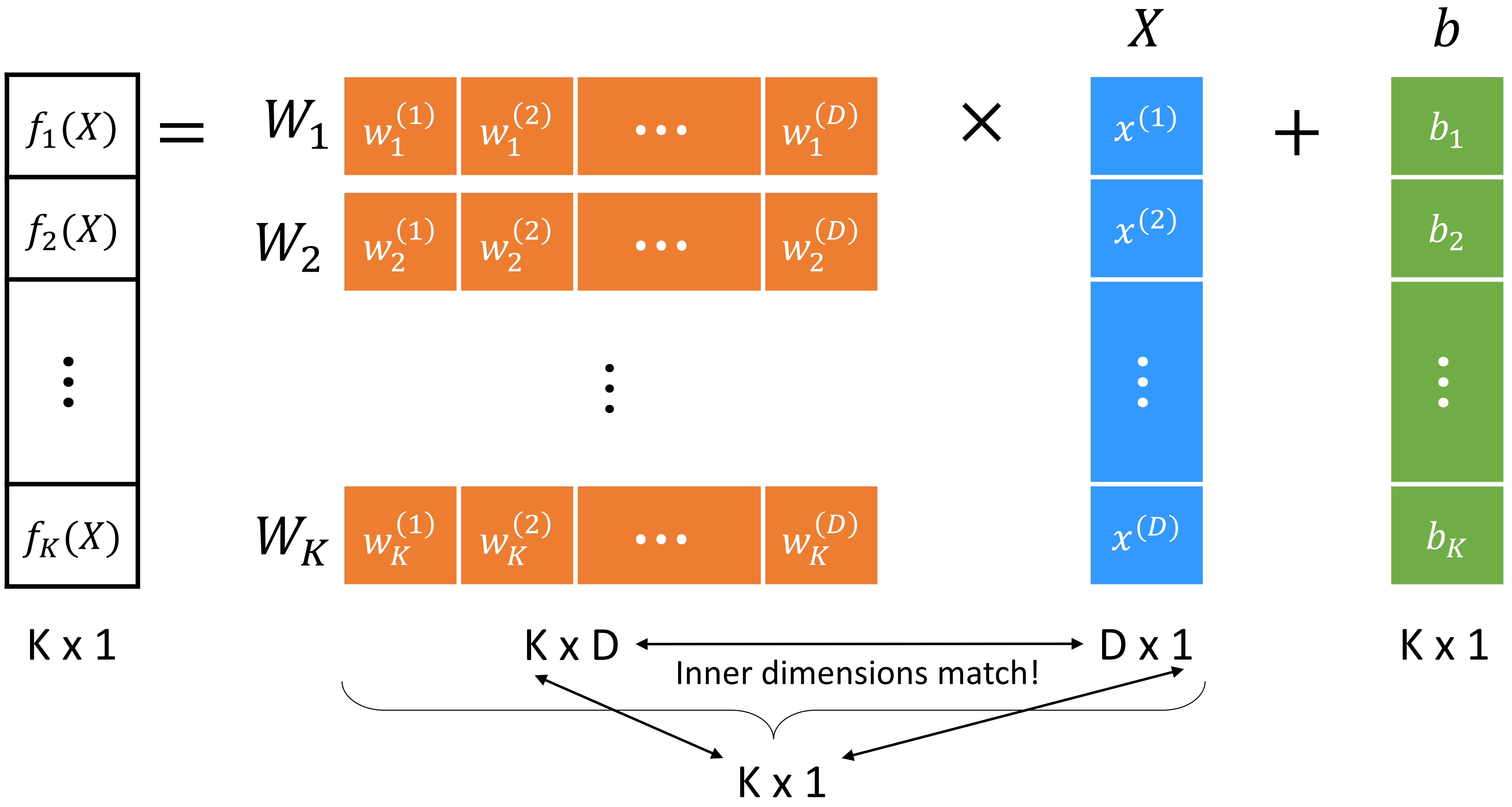
We write matrix shape as:
#rows x #columns
Or: (rows, columns)



$$\begin{bmatrix} f_1(X) & f_2(X) & \cdots & f_K(X) \end{bmatrix} = \underbrace{\begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(D)} \end{bmatrix}}_{X} \times \begin{array}{cccc} W_1 & W_2 & & W_K \\ \hline w_1^{(1)} & w_2^{(1)} & & w_1^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \cdots & w_1^{(2)} \\ \vdots & \vdots & & \vdots \\ w_1^{(D)} & w_2^{(D)} & & w_1^{(D)} \end{array} + \underbrace{\begin{bmatrix} b_1 & b_2 & \cdots & b_K \end{bmatrix}}_{b}$$

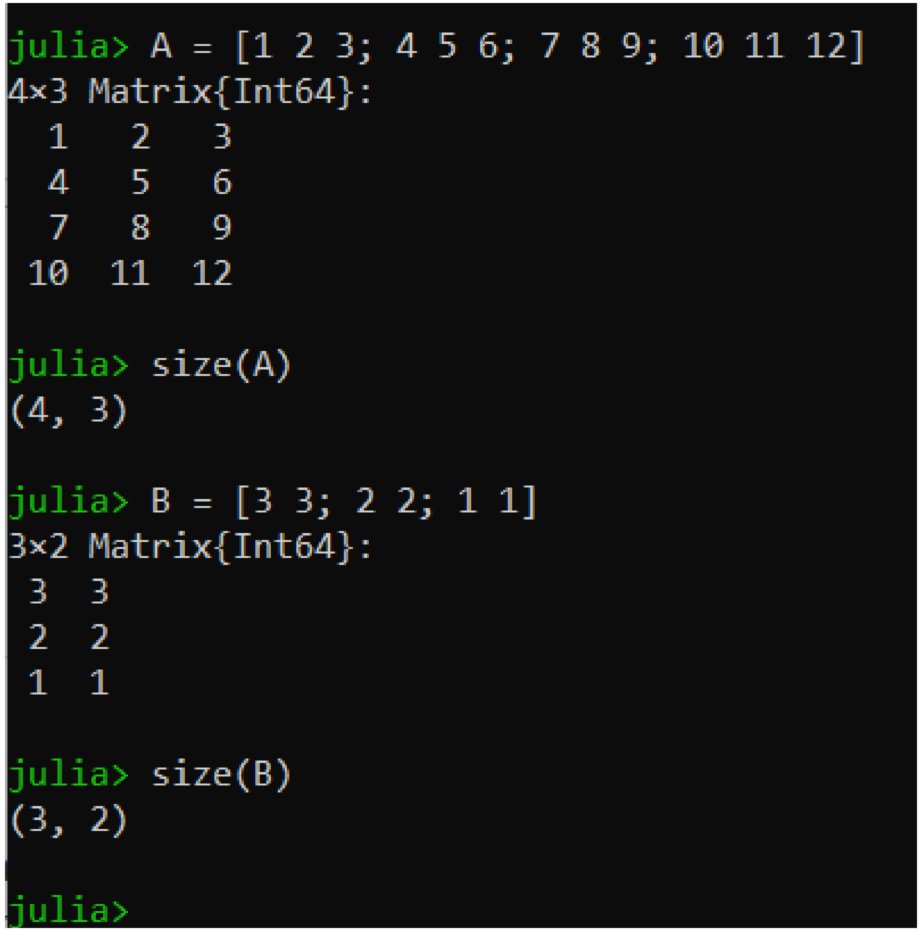
Matrix Shapes

We write matrix shape as:
#rows x #columns
Or: (rows, columns)

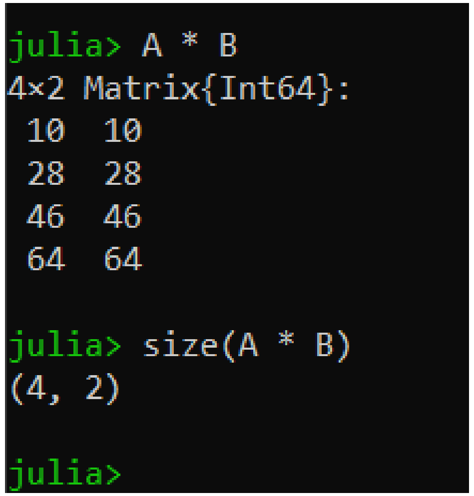


$$\begin{bmatrix} f_1(X) \\ f_2(X) \\ \vdots \\ f_K(X) \end{bmatrix} = \begin{array}{c} W_1 \\ W_2 \\ \\ W_K \end{array} \begin{bmatrix} w_1^{(1)} & w_1^{(2)} & \cdots & w_1^{(D)} \\ w_2^{(1)} & w_2^{(2)} & \cdots & w_2^{(D)} \\ & \vdots & & \\ w_K^{(1)} & w_K^{(2)} & \cdots & w_K^{(D)} \end{bmatrix} \times \overset{X}{\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(D)} \end{bmatrix}} + \overset{b}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}}$$

Matrix Multiplication in Julia



```
julia> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12

julia> size(A)
(4, 3)

julia> B = [3 3; 2 2; 1 1]
3×2 Matrix{Int64}:
 3  3
 2  2
 1  1

julia> size(B)
(3, 2)

julia>
```



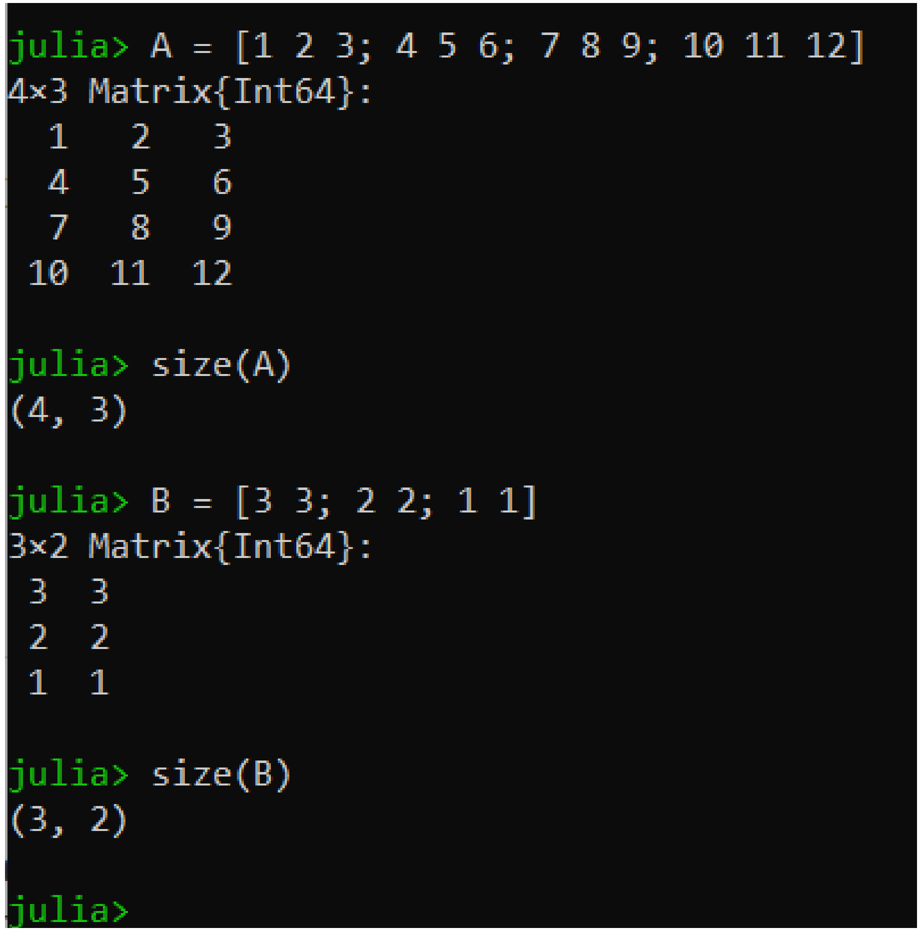
```
julia> A * B
4×2 Matrix{Int64}:
 10  10
 28  28
 46  46
 64  64

julia> size(A * B)
(4, 2)

julia>
```

Legal!

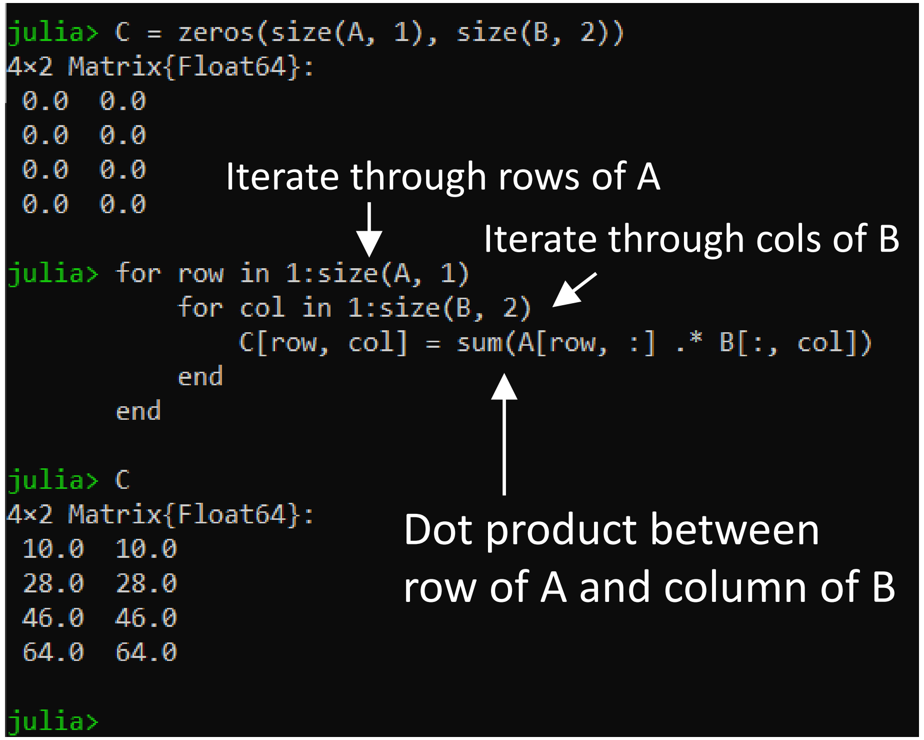
The inner dimensions match:

(4, 3) x (3, 2) → (4, 2)

Matrix Multiplication in Julia



```
julia> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12

julia> size(A)
(4, 3)

julia> B = [3 3; 2 2; 1 1]
3×2 Matrix{Int64}:
 3  3
 2  2
 1  1

julia> size(B)
(3, 2)

julia>
```



```
julia> C = zeros(size(A, 1), size(B, 2))
4×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
 0.0  0.0

julia> for row in 1:size(A, 1)
           for col in 1:size(B, 2)
               C[row, col] = sum(A[row, :] .* B[:, col])
           end
       end

julia> C
4×2 Matrix{Float64}:
 10.0  10.0
 28.0  28.0
 46.0  46.0
 64.0  64.0

julia>
```

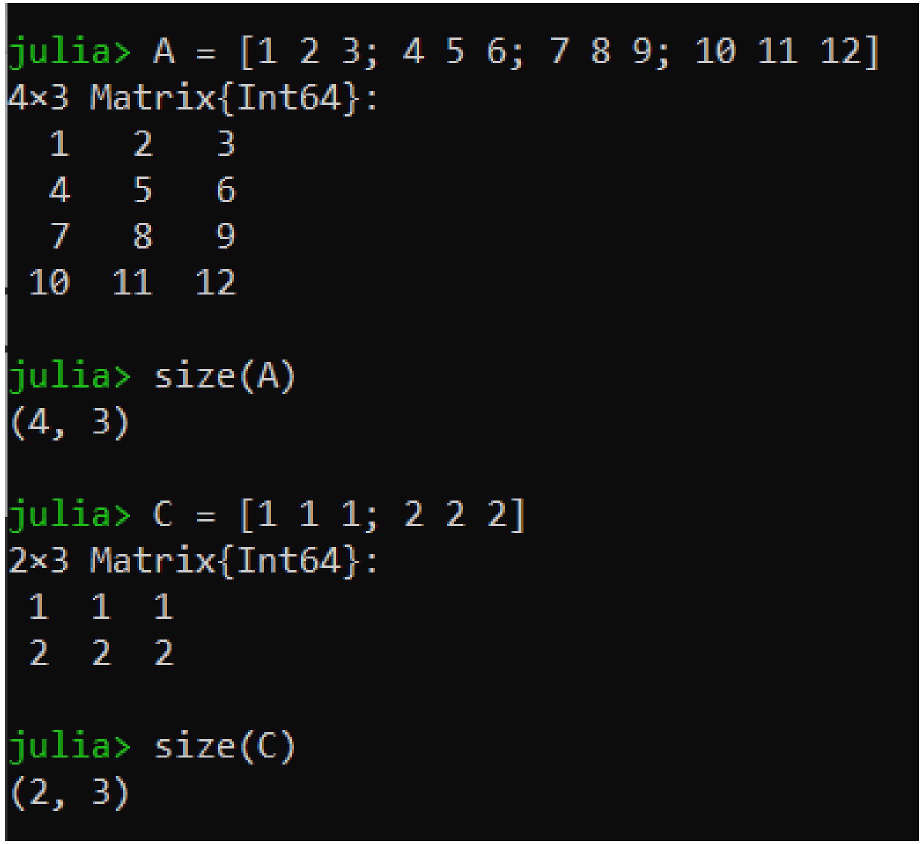
Iterate through rows of A

Iterate through cols of B

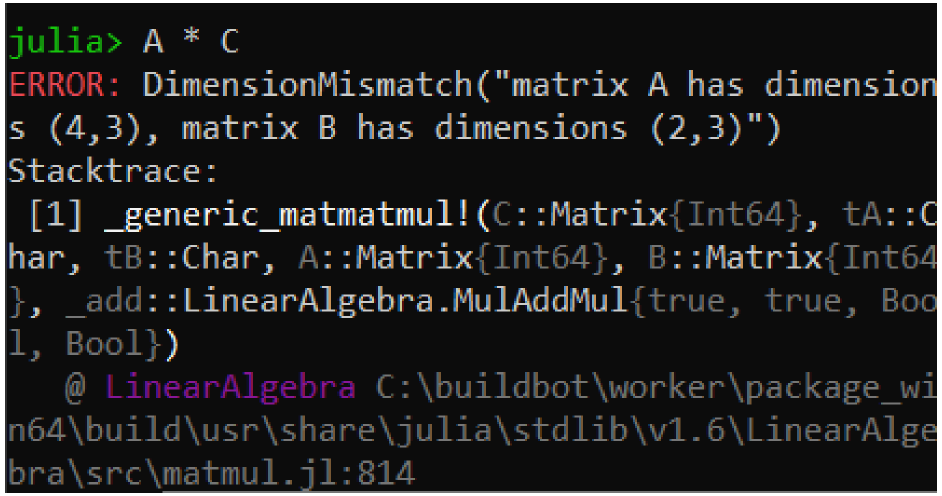
Dot product between row of A and column of B

We get the same answer if we do dot products between the rows of A and the columns of B!

Matrix Multiplication in Julia



```
julia> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12

julia> size(A)
(4, 3)

julia> C = [1 1 1; 2 2 2]
2×3 Matrix{Int64}:
 1  1  1
 2  2  2

julia> size(C)
(2, 3)
```



```
julia> A * C
ERROR: DimensionMismatch("matrix A has dimension
s (4,3), matrix B has dimensions (2,3)")
Stacktrace:
 [1] _generic_matmatmul!(C::Matrix{Int64}, tA::C
har, tB::Char, A::Matrix{Int64}, B::Matrix{Int64
}, _add::LinearAlgebra.MulAddMul{true, true, Boo
l, Bool})
   @ LinearAlgebra C:\buildbot\worker\package_wi
n64\build\usr\share\julia\stdlib\v1.6\LinearAlge
bra\src\matmul.jl:814
```

Illegal ☹

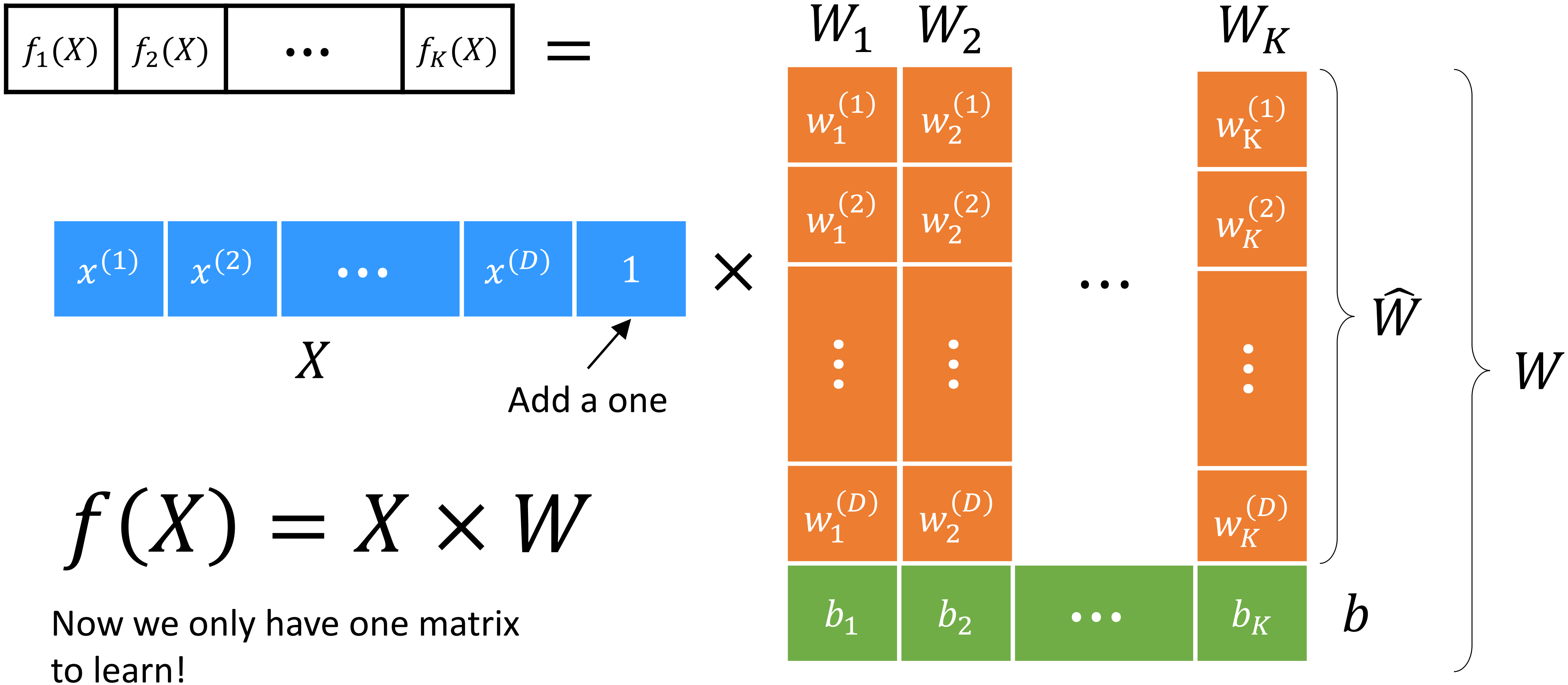
The inner dimensions don’t match:

(4, 3) x (2, 3) → Fails!

One more trick…

$$\begin{bmatrix} f_1(X) & f_2(X) & \cdots & f_K(X) \end{bmatrix} = \underset{X}{\begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(D)} \end{bmatrix}} \times \underbrace{\begin{array}{cccc} W_1 & W_2 & & W_K \\ w_1^{(1)} & w_2^{(1)} & & w_K^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \cdots & w_K^{(2)} \\ \vdots & \vdots & & \vdots \\ w_1^{(D)} & w_2^{(D)} & & w_K^{(D)} \end{array}}_{\widehat{W}} + \underset{b}{\begin{bmatrix} b_1 & b_2 & \cdots & b_K \end{bmatrix}}$$

One more trick…



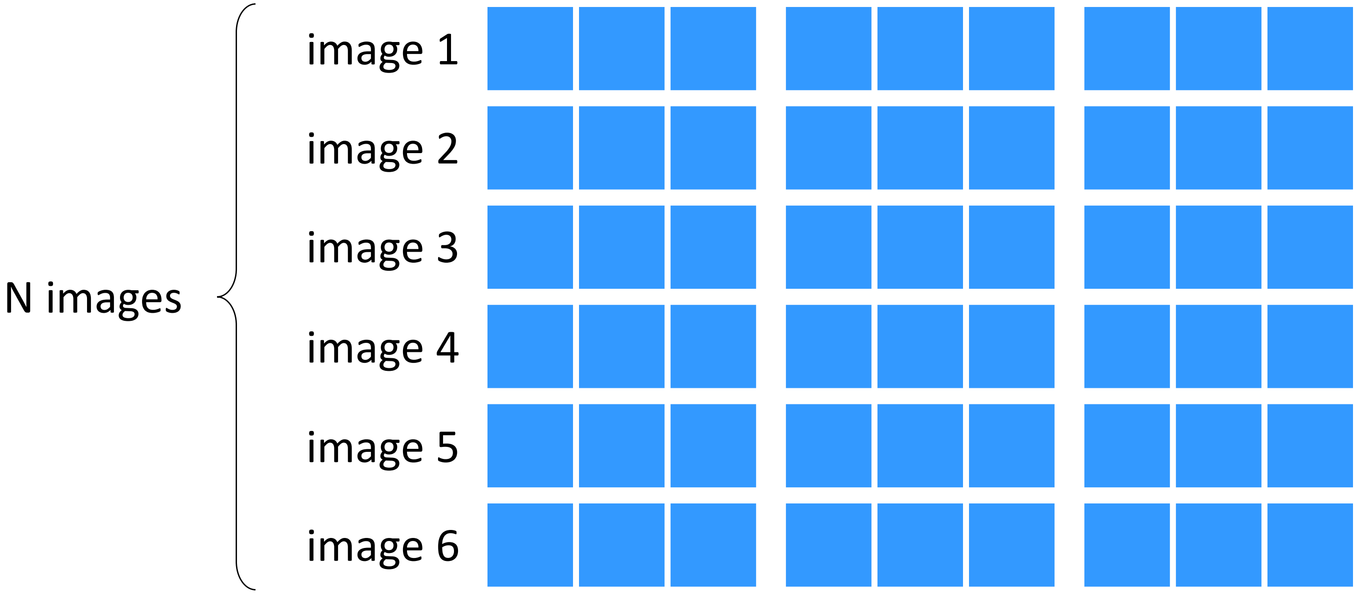
$$f(X) = X \times W$$

Now we only have one matrix to learn!

**but (D+1) x K parameters

Images as Matrices

When we have multiple images, we will stack them up into a big matrix. For N images, the matrix will have size Nx(W*H).



Exercise: Can you calculate the class scores for all the images using one matrix multiplication?

MNIST has 60,000 training images. The training image matrix will have size 60,000x784.

Training the model

How do we find $\boldsymbol{W}$?

$$f(X) = X \times \boldsymbol{W}$$

We will use optimization!

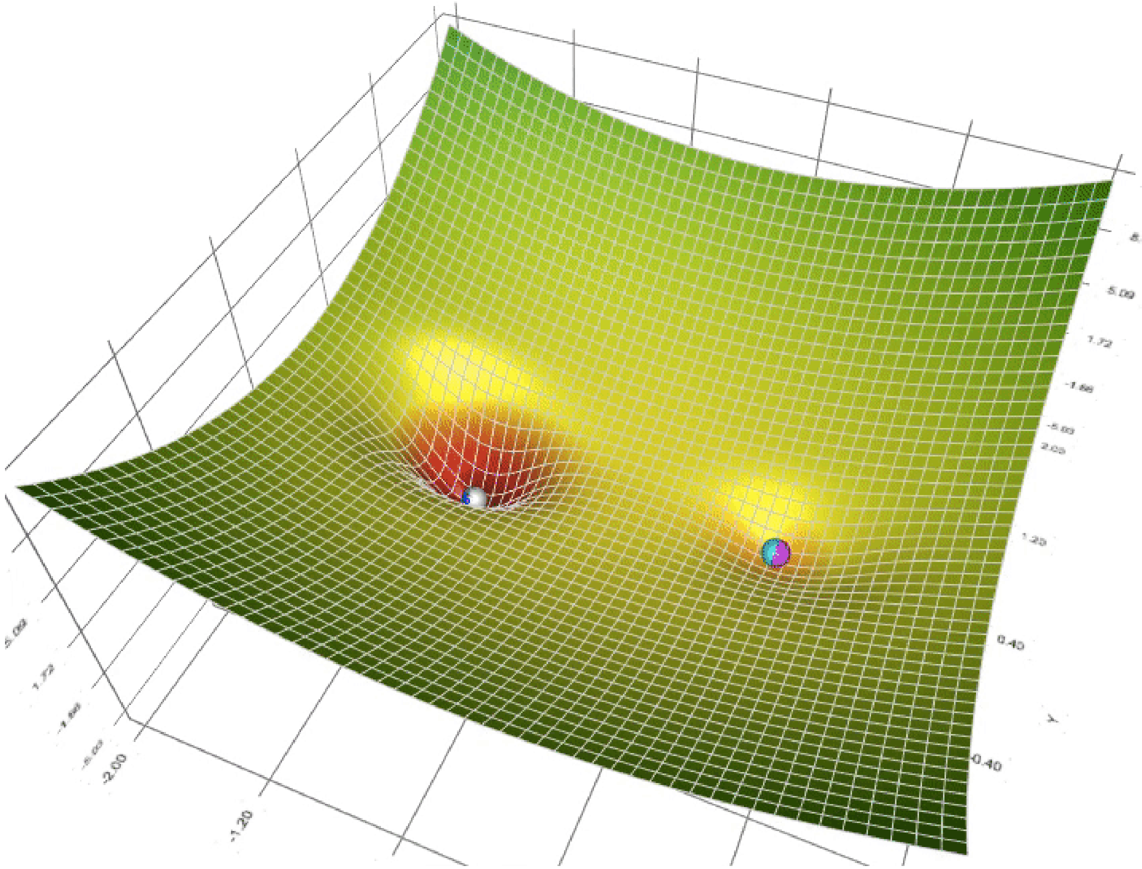
Gradient Descent Optimization

An optimization algorithm helps us find the best *local* (lowest or highest) value of a function.

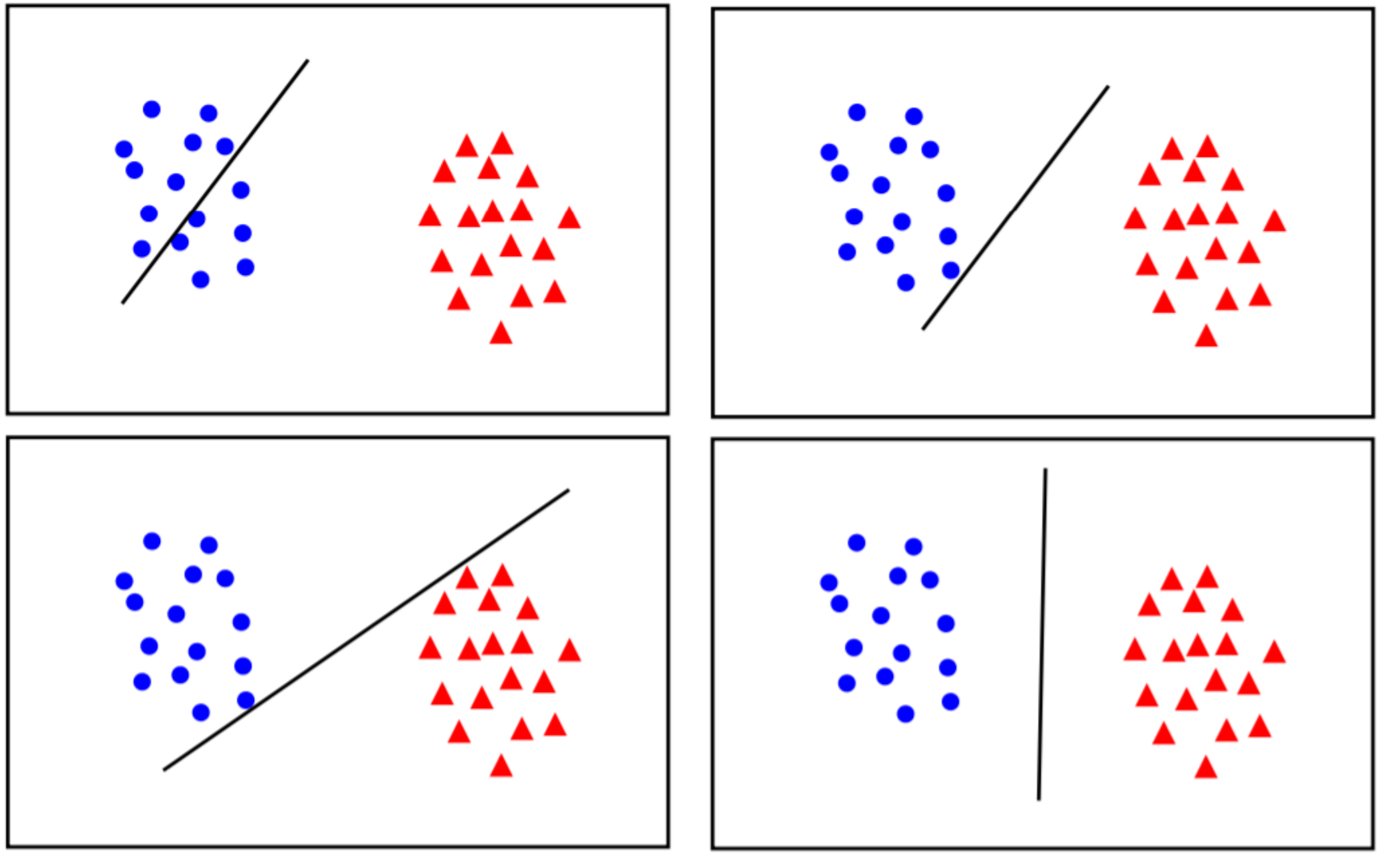
We will use an algorithm called **gradient descent** to find the weights.

First, we need something to optimize.

(Link to GIF)



How can we train a linear classifier?

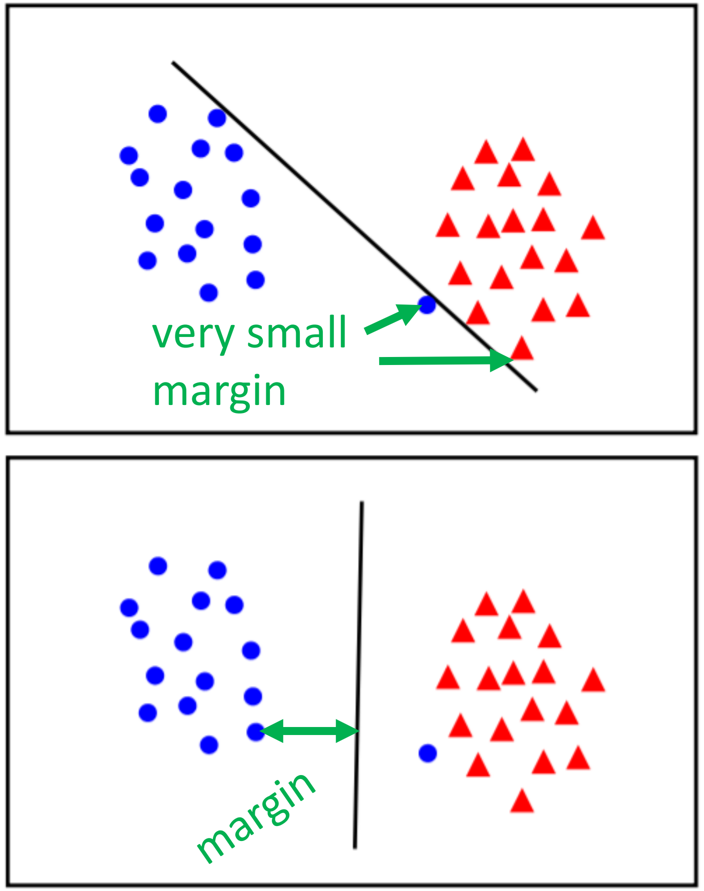


Which of these classifiers is better?

Recall: Overfitting

We would like our model to perform well on new data, even if that sacrifices performance on training data.

How can we train a linear classifier?



Idea: We want to find a function which maximizes a margin.

This will make our algorithm more stable to perturbations in the input.

We might misclassify an example or two in our training set, but hopefully we'll get a function which is better at classifying new images.

Loss Function

A **loss function** tells us how good a model is at classifying an image.

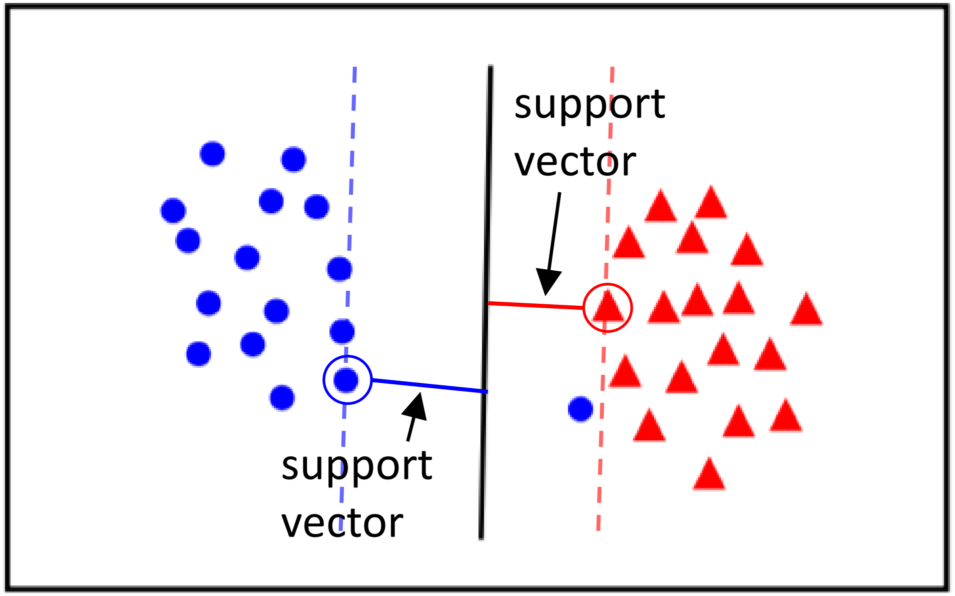
The loss is **LOW** if we’re doing a good job at classifying, and **HIGH** if we’re doing a bad job.

Our goal is to find weights W that *minimize* the loss. This is called optimization. (More on that later).

Support Vector Machine (SVM) Loss

Idea: The score for the correct class of an image should be higher than the other scores by some margin.

Aside: Why is it called "support vector machine"?



SVM Loss

Goal: Images should be classified correctly by at least a margin Δ.

Say for some image X, its correct label is y with score $f_y(X)$. For all classes $i \neq y$, we want:

$$\text{Correct class score} \longrightarrow f_y(X) \geq \overset{\substack{\text{Incorrect class score} \\ \downarrow}}{f_i(X)} + \Delta$$

In other words, the score for some incorrect class, $f_i(X)$ is less than the correct class score by at least Δ.

SVM Loss

If the margin condition is met, then we're happy! We will set $L_i = 0$.
Otherwise, the loss will be the amount by which we are off:

Incorrect class score

$$L_i = (f_i(X) + \Delta\ ) - f_y(X) \longleftarrow$$

Correct class score

The total loss is the sum of the losses for each class that is not correct:

$$L = \sum_{\forall\, i\, \backslash y} \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

Note: $\forall\, i\ \backslash\ y$ means "for all values of i except y"

SVM Loss

If the margin condition is met, then we're happy! We will set $L_i = 0$. Otherwise, the loss will be the amount by which we are off:

Incorrect class score

$$L_i = \left(f_i(X) + \Delta\right) - f_y(X) \longleftarrow$$

Correct class score

The total loss is the sum of the losses for each class that is not correct:

$$L = \sum_{\forall\, i \setminus y} \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

In English: For each incorrect class, add its loss to the total, if it wasn't less than the correct class by the margin.

Example: SVM Loss

$$L_i = \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

Exercise: Find the SVM Loss for each image.

classes	scores		
cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Image Credit: Johnson (link)

Example: SVM Loss

$$L_i = \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

2.9

Exercise: Find the SVM Loss for each image.

$\Delta = 1$

car: $5.1 + 1 = 6.1 > 3.2$

$L_{car} = 6.1 - 3.2 = \mathbf{2.9}$

frog: $-1.7 + 1 = -0.7 < 3.2$

$L_{frog} = \mathbf{0}$

$L = \mathbf{2.9} + \mathbf{0} = 2.9$

Image Credit: Johnson (link)

$$L_i = \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

Example: SVM Loss

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0	

Exercise: Find the SVM Loss for each image.

$\Delta = 1$

cat: 1.3 + 1 = 2.3 < 4.9

$L_{cat} = \mathbf{0}$

frog: 2.0 + 1 = 3.0 < 4.9

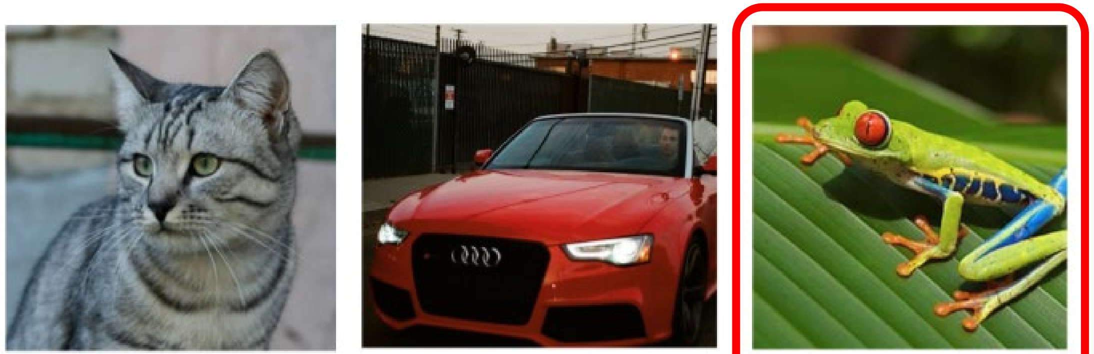
$L_{frog} = \mathbf{0}$

$L = \mathbf{0} + \mathbf{0} = 0$

Image Credit: Johnson (link)

Example: SVM Loss

$$L_i = \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0	12.9

Exercise: Find the SVM Loss for each image.

$\Delta = 1$

cat: 2.2 + 1 = 3.2 > -3.1

L_{cat} = 3.2 - -3.1 = **6.3**

car: 2.5 + 1 = 3.5 > -3.1

L_{frog} = 3.5 - -3.1 = **6.6**

L = **6.3** + **6.6** = 12.9

Image Credit: Johnson (link)

Example: SVM Loss

$$L_i = \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

Exercise: Find the SVM Loss for each image.

$\Delta = 1$

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0	12.9

Total Loss:

= (2.9 + 0 + 12.9) / 3

= 5.27

Image Credit: Johnson (link)

How can we train a linear classifier?

Goal: Find a set of weights W that minimizes the loss.

Setting the weights

Idea #1: Random Search

Until we're happy with the performance, do:

1. Set weights randomly
2. Check how well the classifier does

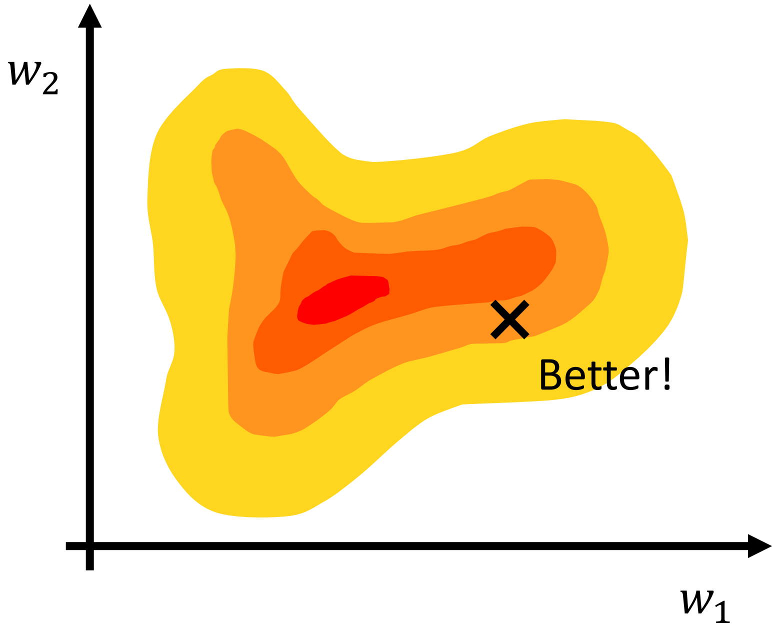
Red = Loss function is low

Setting the weights

Idea #1: Random Search

Until we're happy with the performance, do:

1. Set weights randomly
2. Check how well the classifier does



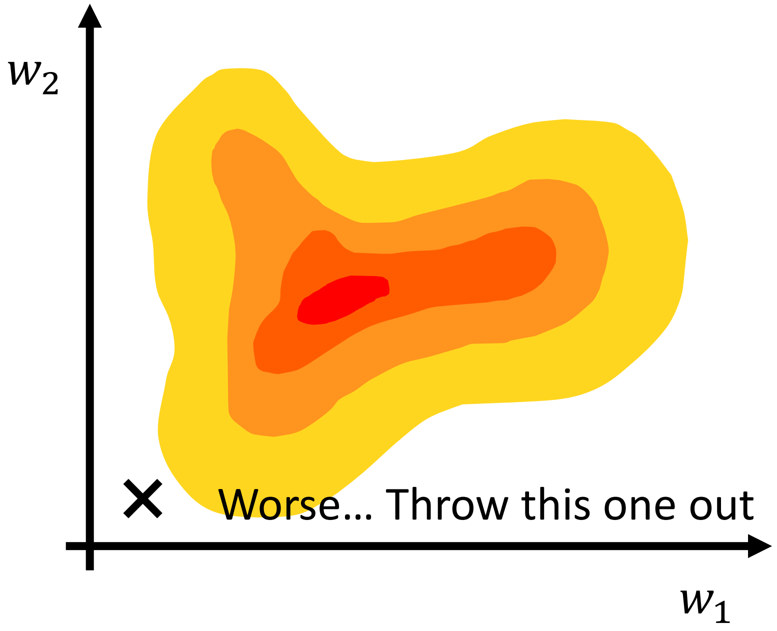
Red = Loss function is low

Setting the weights

Idea #1: Random Search

Until we’re happy with the performance, do:

1. Set weights randomly
2. Check how well the classifier does



Red = Loss function is low

Setting the weights

Idea #1: Random Search

Until we're happy with the performance, do:

1. Set weights randomly
2. Check how well the classifier does

Red = Loss function is low

Bad idea! We have 785x10 different numbers to find. This will take forever.

Setting the weights

Imagine you’re hiking (blindfolded!). You want to get to the lowest point of the hill.

Idea:

1. Try to take a step.
2. Stay there if you get lower than your current location.
3. Repeat!

Setting the weights

Idea #2: Random Local Search

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights

Red = Loss function is low

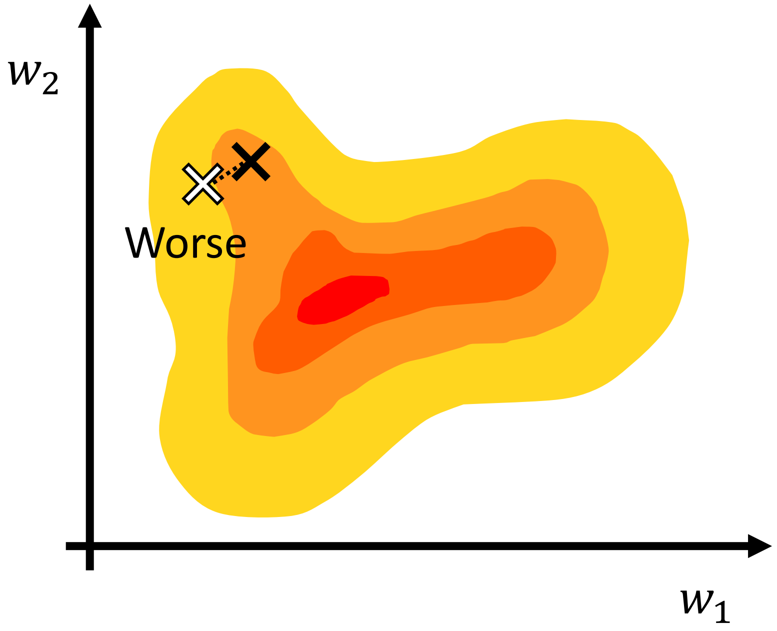
Setting the weights

Idea #2: Random Local Search

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights



Red = Loss function is low

Setting the weights

Idea #2: Random Local Search

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights

Red = Loss function is low

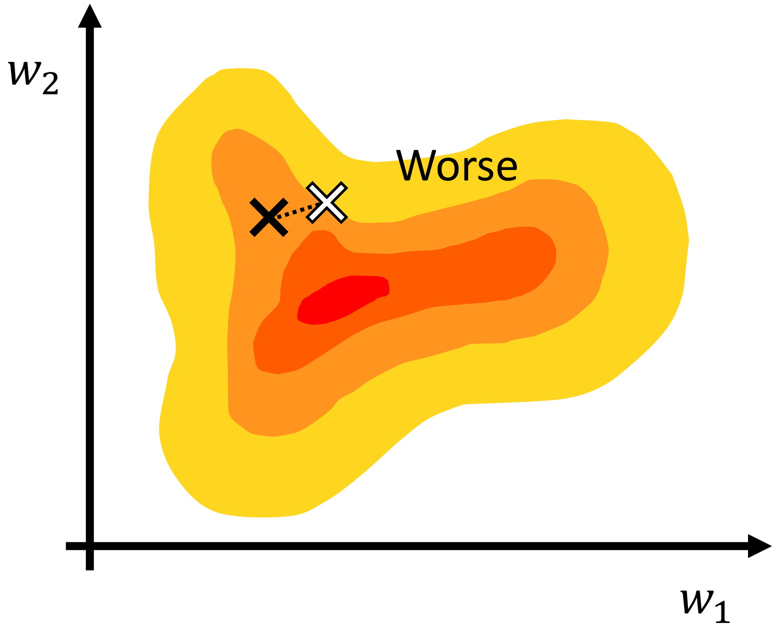
Setting the weights

Idea #2: Random Local Search

Set the weights randomly

Until we’re happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights



Red = Loss function is low

Setting the weights

Idea #2: Random Local Search

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights

Red = Loss function is low

Setting the weights

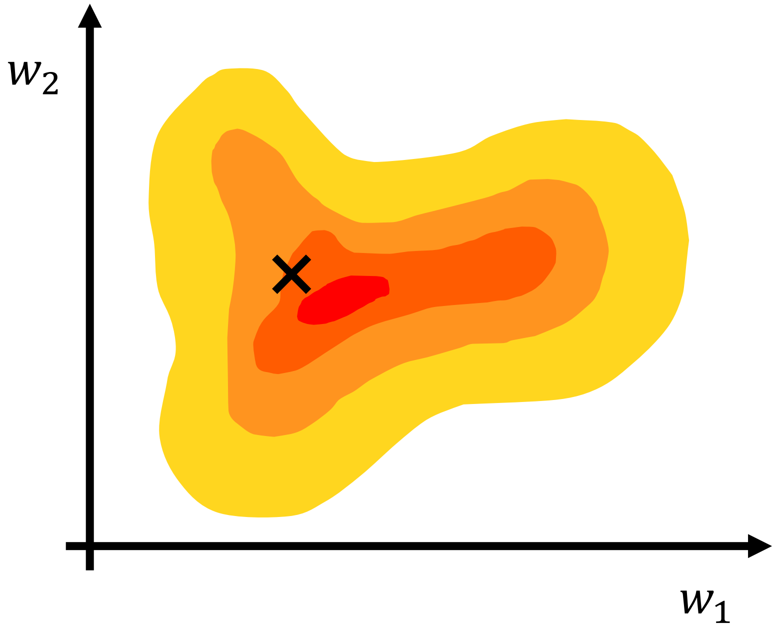
Idea #2: Random Local Search

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in a random direction
2. If the loss improves, update the weights

Better, but still not great. It will be hard to find the right direction to step.

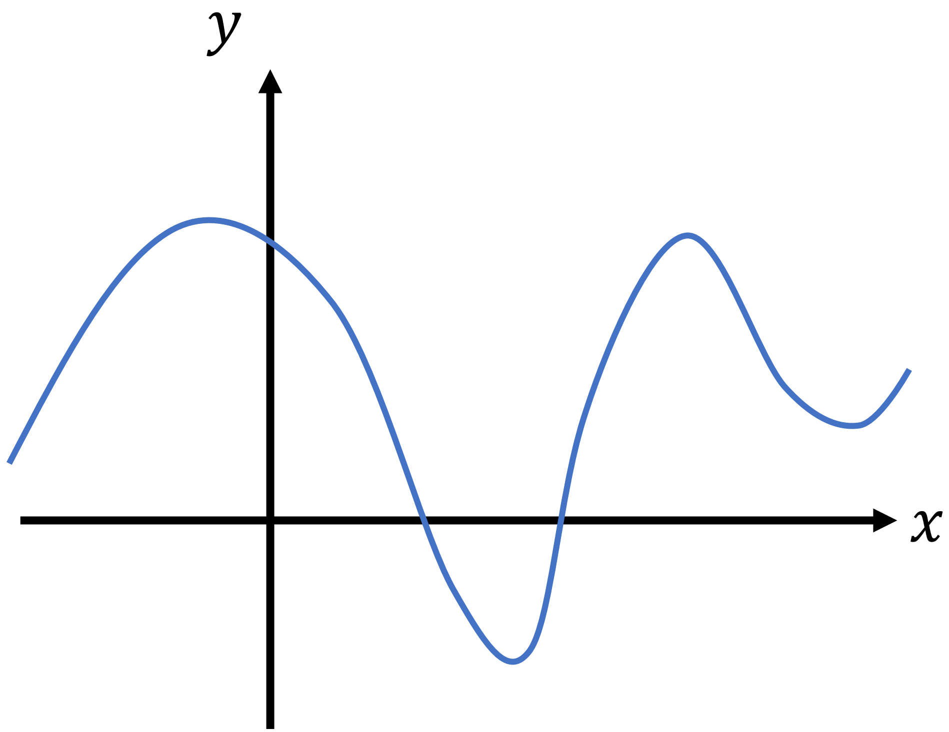


Red = Loss function is low

Rate of Change

Can we do better?

Yes! With Calculus ☺

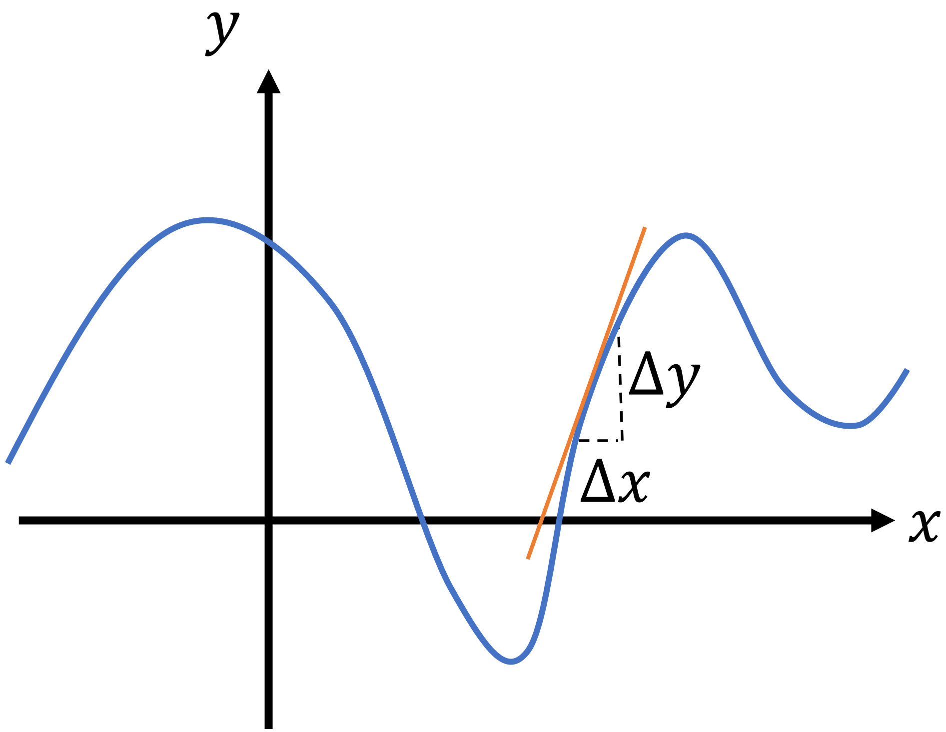


Rate of Change

The slope at a point is given by:

$$\text{slope} = \frac{\Delta y}{\Delta x}$$

This is the **rate of change** of y. It tells us how y changes if we change x a little.



Rate of Change

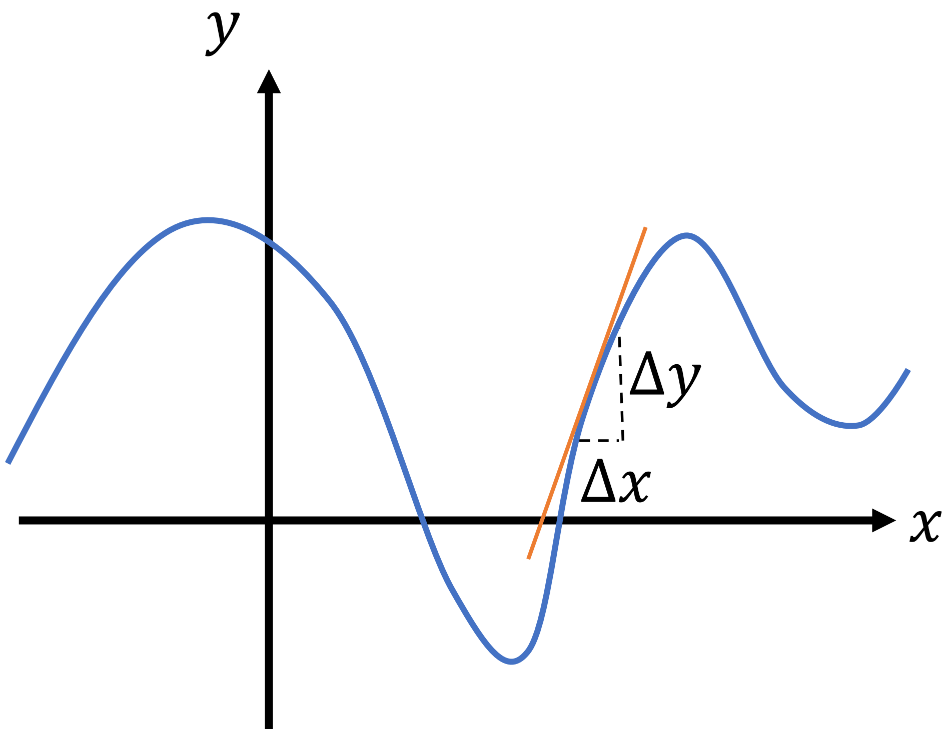
The slope at a point is given by:

$$\text{rate of change} = \frac{\Delta y}{\Delta x}$$

Recall from calculus:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

gradient



Gradient

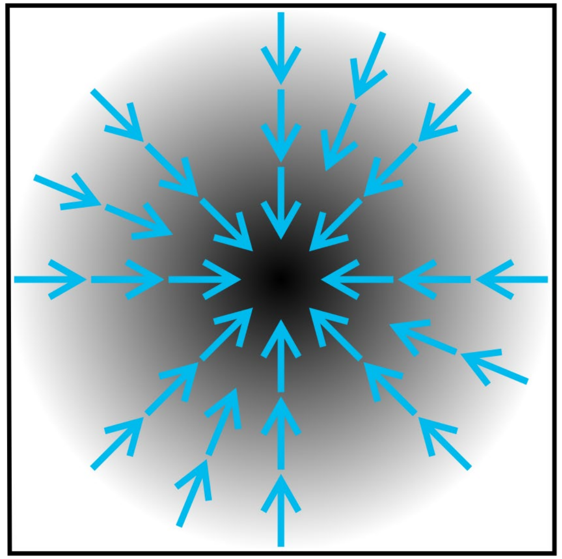
Recall from calculus:

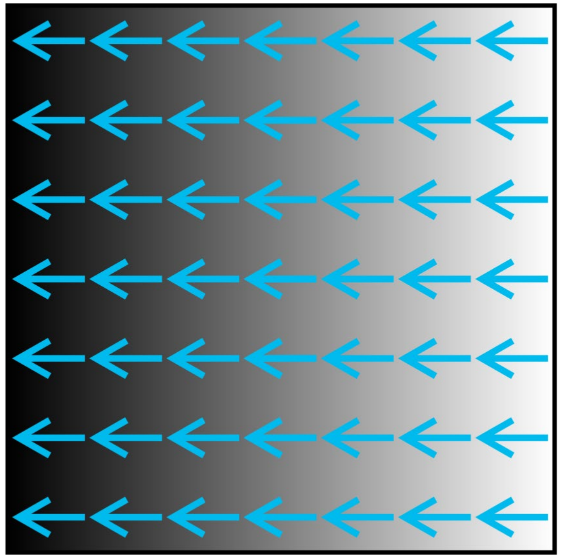
$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

In 2D, the gradient is the direction and rate of fastest increase.

gradients in 2D → $\left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right]$

partial derivatives





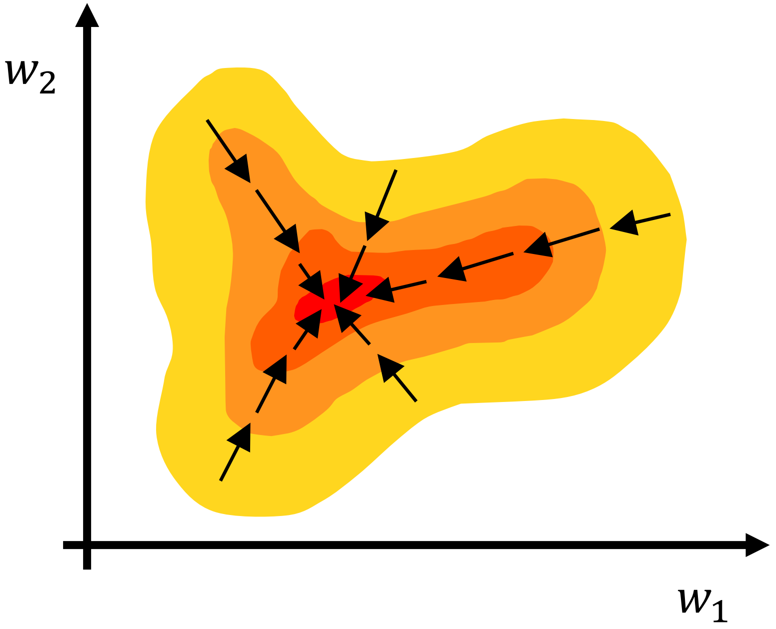
Gradient

Recall from calculus:

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

In 2D, the gradient is the direction and rate of fastest increase.

gradients in 2D $\longrightarrow \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}\right]$



Red = Loss function is low

Idea: We can use the gradient of the loss function to figure out how to update our weights!

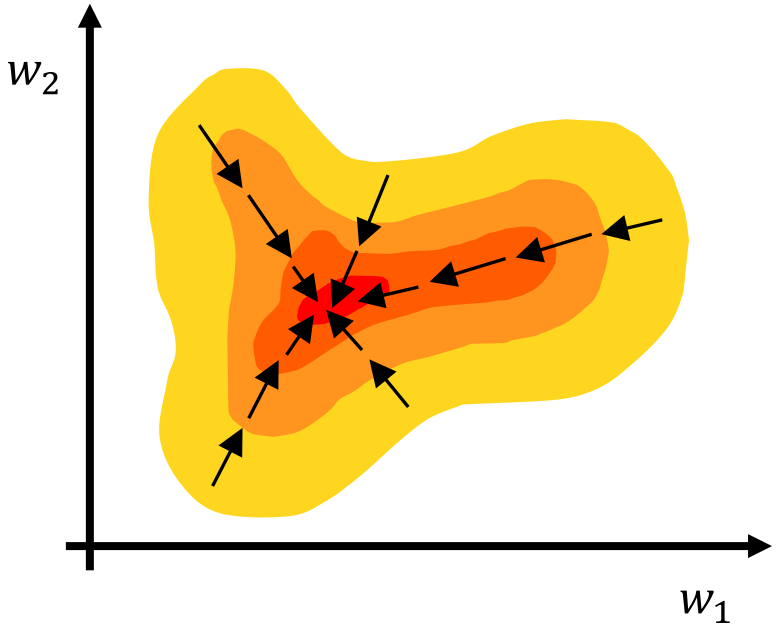
Setting the weights: Gradient Descent

Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient



Red = Loss function is low

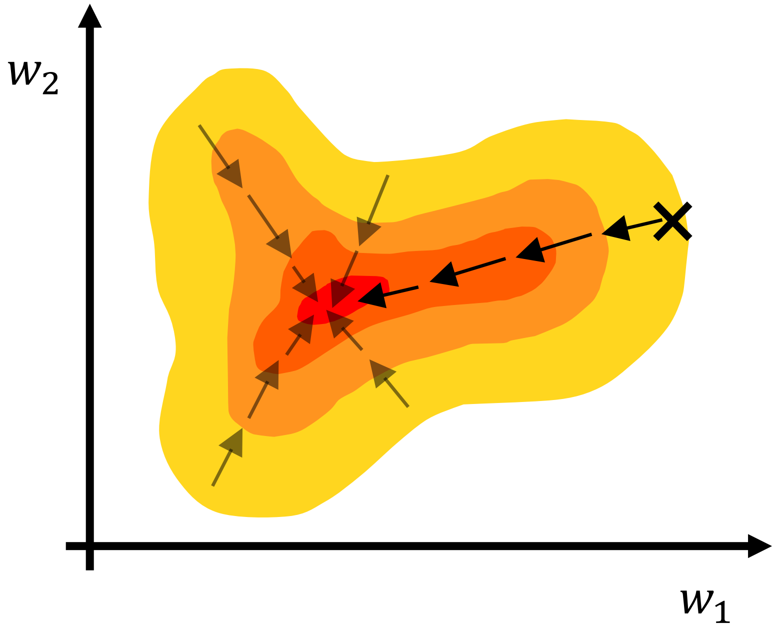
Setting the weights: Gradient Descent

Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient



Red = Loss function is low

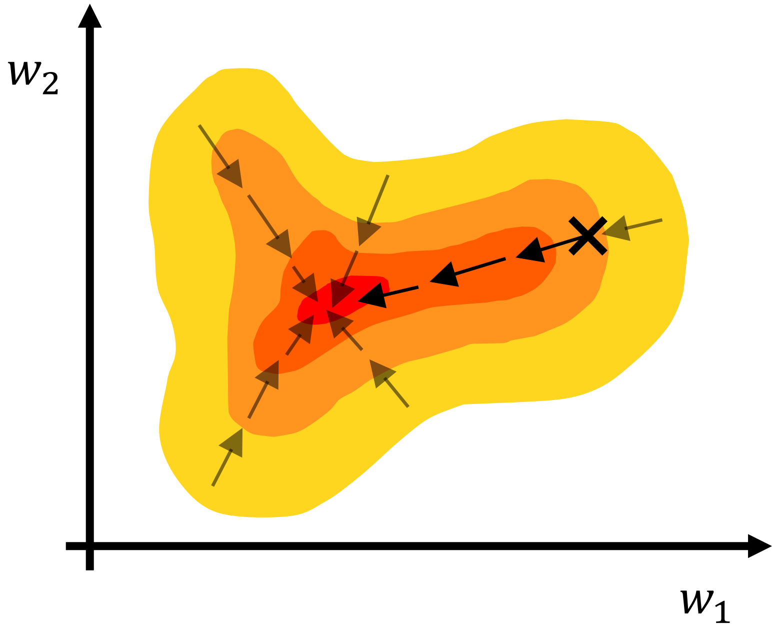
Setting the weights: Gradient Descent

Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient



Red = Loss function is low

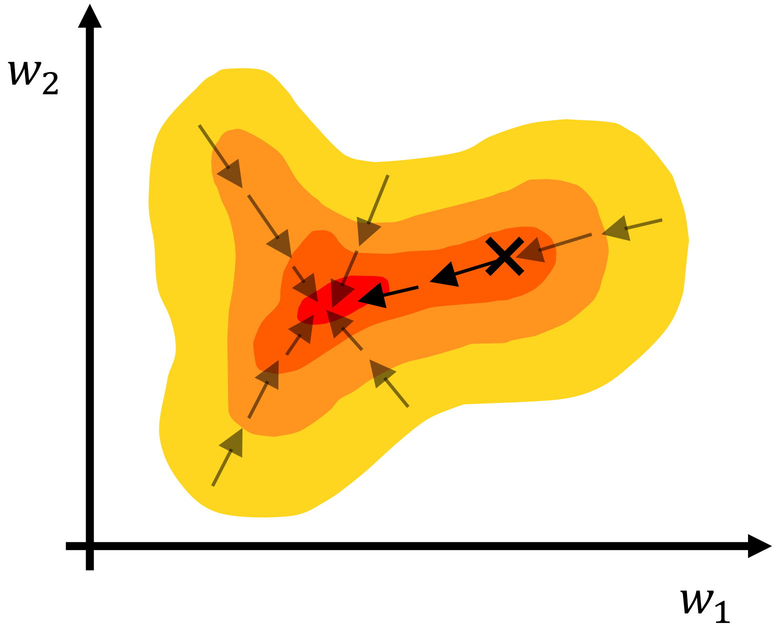
Setting the weights: Gradient Descent

Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient



Red = Loss function is low

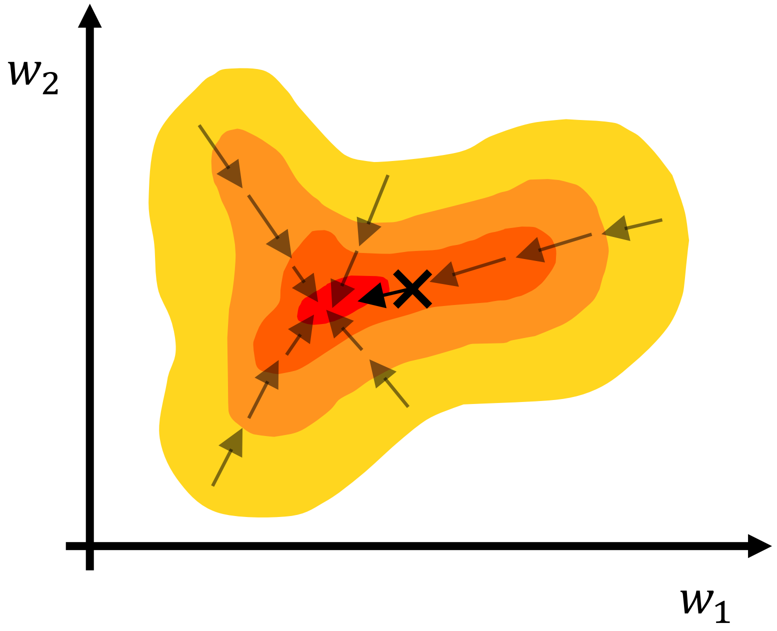
Setting the weights: Gradient Descent

Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient



Red = Loss function is low

Setting the weights: Gradient Descent

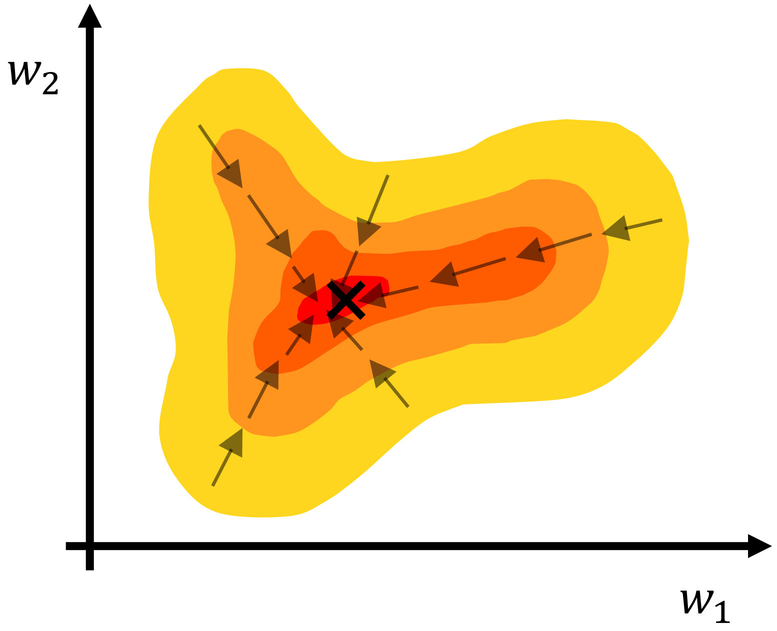
Idea #3: Following the gradient

Set the weights randomly

Until we're happy with the performance, do:

1. Take a small step in the direction of the gradient

Better! And we can compute the gradients ahead of time because we know how to do the derivatives.



Red = Loss function is low

Optimization using Gradient Descent

Gradient descent algorithm:

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W)
    W = W - step_size * loss_grad
```

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W)
    W = W - step_size * loss_grad
```

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W)
    W = W - step_size * loss_grad
```

For a fixed number of iterations…

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W, y)
    W = W - step_size * loss_grad
```

For a fixed number of iterations…

loss function

images

labels

Calculate the gradient of the loss w.r.t. W.

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W, y)
    W = W - step_size * loss_grad
```

loss function

images

labels

For a fixed number of iterations…

Calculate the gradient of the loss w.r.t. W.

Update the weight in the opposite direction of the gradient

Gradient is the direction of maximum *increase*, and we want the loss to *decrease*, so we use the *negative* of the gradient.

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

loss function

images

labels

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W, y)
    W = W - step_size * loss_grad
```

For a fixed number of iterations…

Calculate the gradient of the loss w.r.t. W.

Update the weight in the opposite direction of the gradient

How do we calculate the gradient?

How do we pick the step size?

Computing the Gradient

We can take the gradient of the loss with respect to each weight:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_1^{(1)}} & \cdots & \frac{\partial L}{\partial w_K^{(1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_1^{(D)}} & \cdots & \frac{\partial L}{\partial w_K^{(D)}} \end{bmatrix}$$

The gradient of the loss with respect to the weights is a matrix with the same size as the weights (D x K).

Computing the Gradient

We can take the gradient of the loss with respect to each weight:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_1^{(1)}} & \cdots & \frac{\partial L}{\partial w_K^{(1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_1^{(D)}} & \cdots & \frac{\partial L}{\partial w_K^{(D)}} \end{bmatrix}$$

We know the loss function, so we can compute the gradient beforehand. You are given a function to compute the gradients in P4!

Computing the Gradient

You are given a function to compute the gradients in P4!

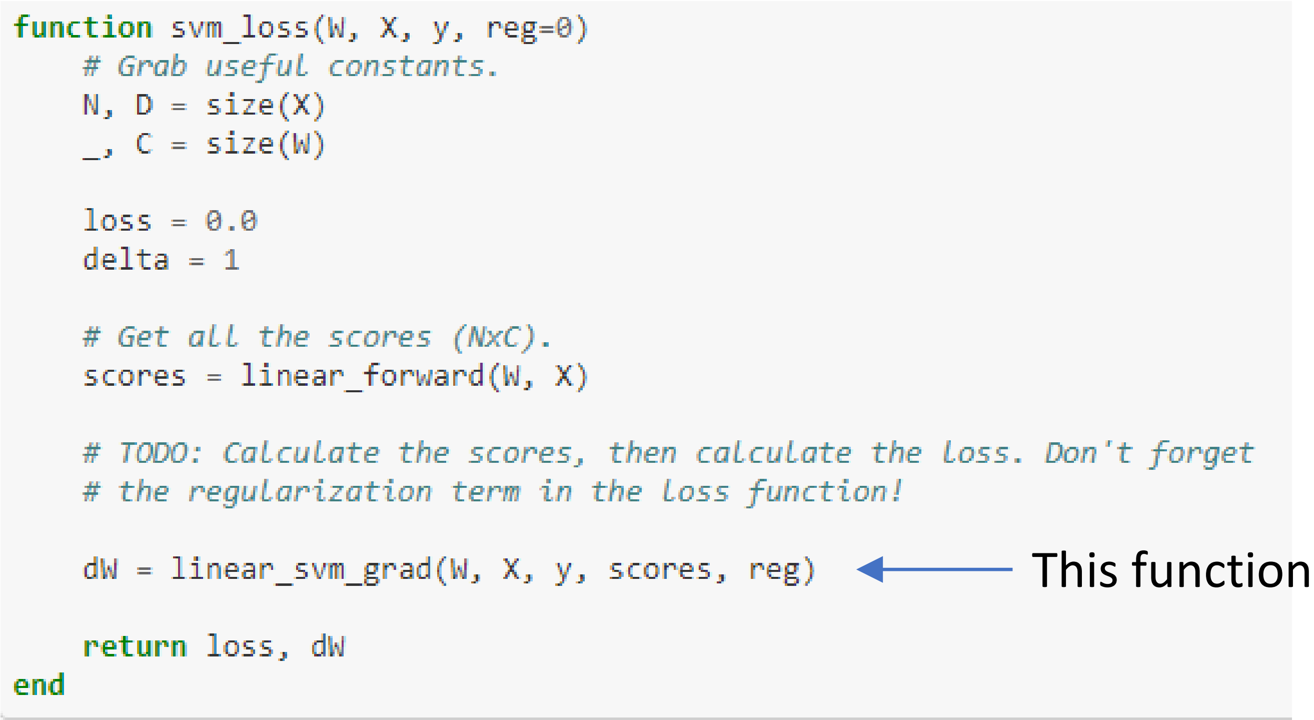
If you want to derive them, check out the notes for Stanford course CS231n. (link)

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

Computing the Gradient in Julia

The provided gradient function is called in the function that computes the loss



```julia
function svm_loss(W, X, y, reg=0)
    # Grab useful constants.
    N, D = size(X)
    _, C = size(W)

    loss = 0.0
    delta = 1

    # Get all the scores (NxC).
    scores = linear_forward(W, X)

    # TODO: Calculate the scores, then calculate the loss. Don't forget
    # the regularization term in the loss function!

    dW = linear_svm_grad(W, X, y, scores, reg)

    return loss, dW
end
```

This function computes gradients

Optimization using Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(D, K) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W, y)
    W = W - step_size * loss_grad
```

loss function

images

labels

For a fixed number of iterations…

Calculate the gradient of the loss w.r.t. W.

Update the weight in the opposite direction of the gradient

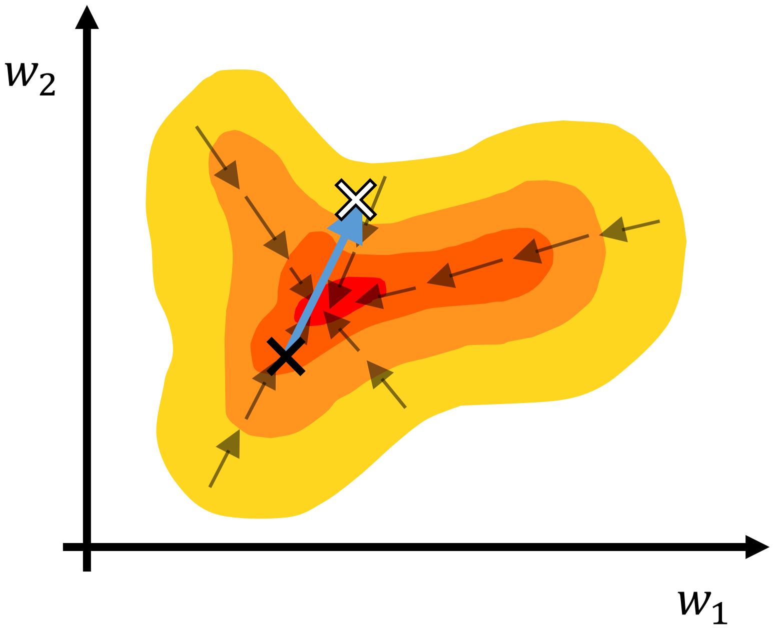
How do we calculate the gradient?

How do we pick the step size?

Learning Rate

The step size, which tells us how big of a step to take along the gradient, is called the **learning rate**.

If we take a huge step, we might overshoot, and get farther away from the minimum.

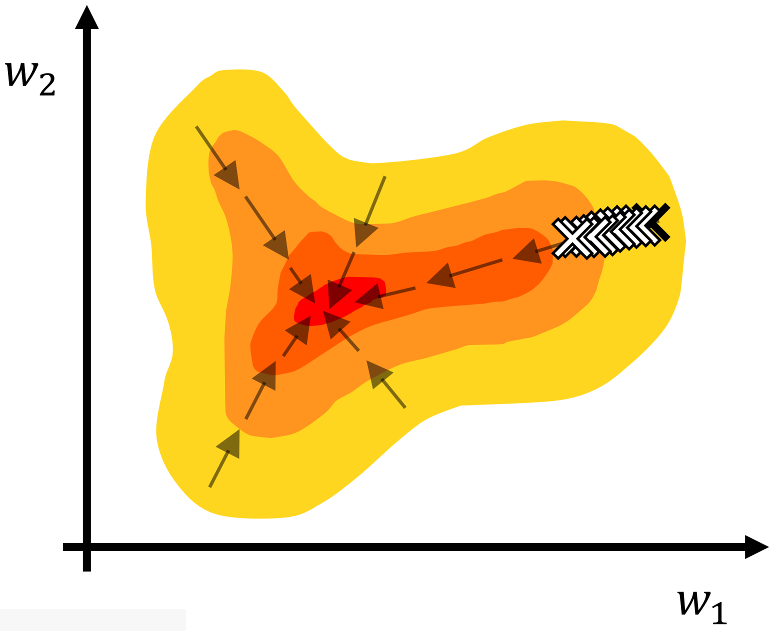


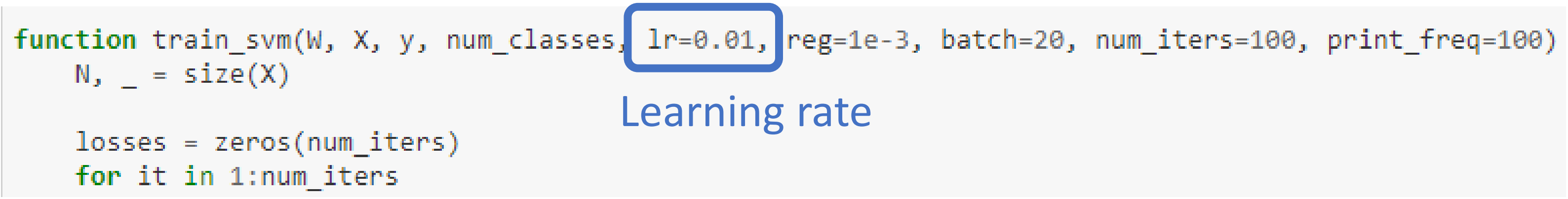
Learning Rate

The step size, which tells us how big of a step to take along the gradient, is called the **learning rate**.

If we take a small step, it will take a long time to find good weights.

The learning rate is a **hyperparameter** we need to set!





```
function train_svm(W, X, y, num_classes, lr=0.01, reg=1e-3, batch=20, num_iters=100, print_freq=100)
    N, _ = size(X)

    losses = zeros(num_iters)
    for it in 1:num_iters
```

Learning rate

Mini-Batch Gradient Descent

Gradient descent algorithm:

Initialize weights to small values drawn from a normal distribution

```
W = random_normal(K, D) * eps
for iteration in 1:N do:
    loss_grad = SVM_grad(SVM_loss, X, W, y)
    W = W - step_size * loss_grad
```

For a fixed number of iterations…

loss function

images

labels

Calculate the gradient of the loss w.r.t. W.

Update the weight in the opposite direction of the gradient

Compute the average loss for all images

That sounds hard…

Mini-Batch Gradient Descent

Instead of computing the loss for every single training image at every iteration (expensive!!) we will only compute loss for a small randomly selected **batch** of data.

We will assume that this gives us a reasonable estimate of what the loss would look like over all the data.

In P4.2, your functions should accept batches of image data, instead of one image. The image matrix X will have shape (N, D) where N is batch size.

Regularization

Let’s say we have a set of weights which classify all the images with 100% accuracy.

$$f(X) = W \times X$$

Any scalar multiplication of the weight matrix will also classify the images perfectly. There are infinite of these matrices!!

$$f(X) = \alpha W \times X$$

That’s going to make it hard to find good weights.

Regularization

The solution to this is to add an additional part to our loss function which tries to classify images correctly while keeping the weights small.

We'll add a regularization loss to the overall loss function:

$$L_{reg}(W) = \alpha \sum_{i=1}^{D} \sum_{j=1}^{K} \left(w_j^{(i)}\right)^2$$

Regularization coefficient
(need to tune this)

Sum of all the squared weights in the weight matrix

Regularization

The solution to this is to add an additional part to our loss function which tries to classify images correctly while keeping the weights small.

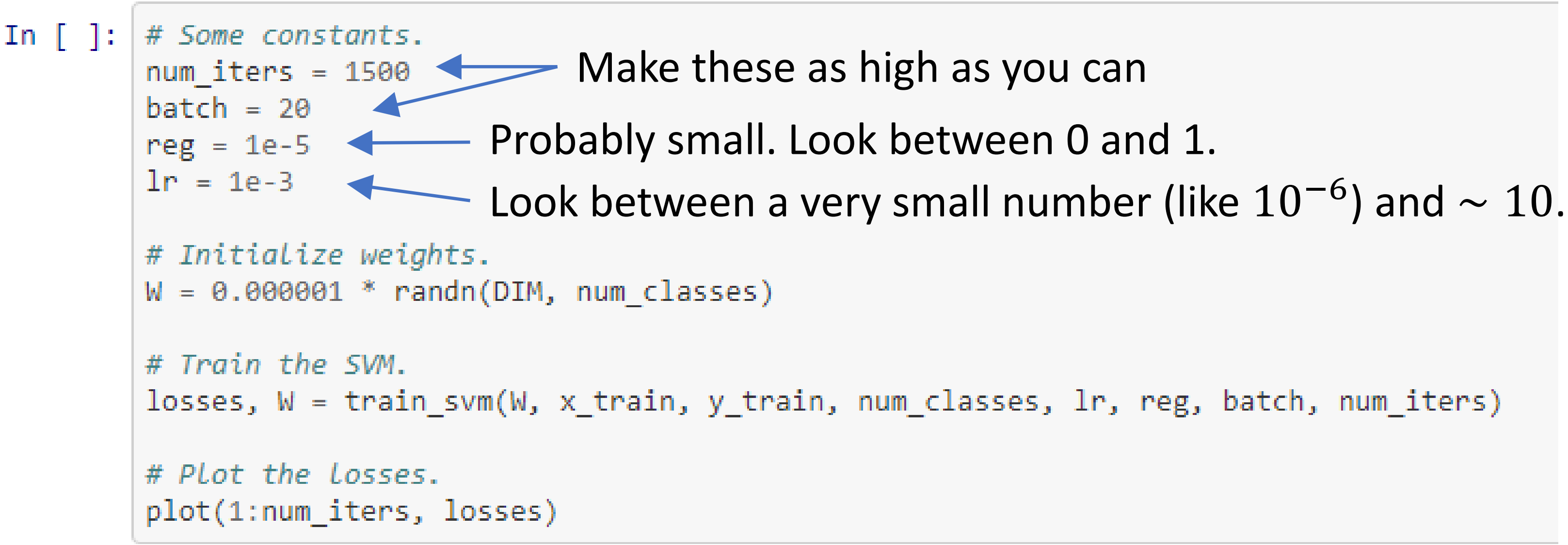
We'll add a regularization loss to the overall loss function:

$$L_{reg}(W) = \alpha \sum_{i=1}^{D} \sum_{j=1}^{K} \left(w_j^{(i)}\right)^2$$

$$L(X, W, y) = L_{SVM}(W, X, y) + L_{reg}(W)$$

Tuning Hyperparameters

Play with the parameters in the notebook!



```
In [ ]: # Some constants.
        num_iters = 1500
        batch = 20
        reg = 1e-5
        lr = 1e-3

        # Initialize weights.
        W = 0.000001 * randn(DIM, num_classes)

        # Train the SVM.
        losses, W = train_svm(W, x_train, y_train, num_classes, lr, reg, batch, num_iters)

        # Plot the losses.
        plot(1:num_iters, losses)
```

Make these as high as you can

Probably small. Look between 0 and 1.

Look between a very small number (like 10^{-6}) and ~ 10.

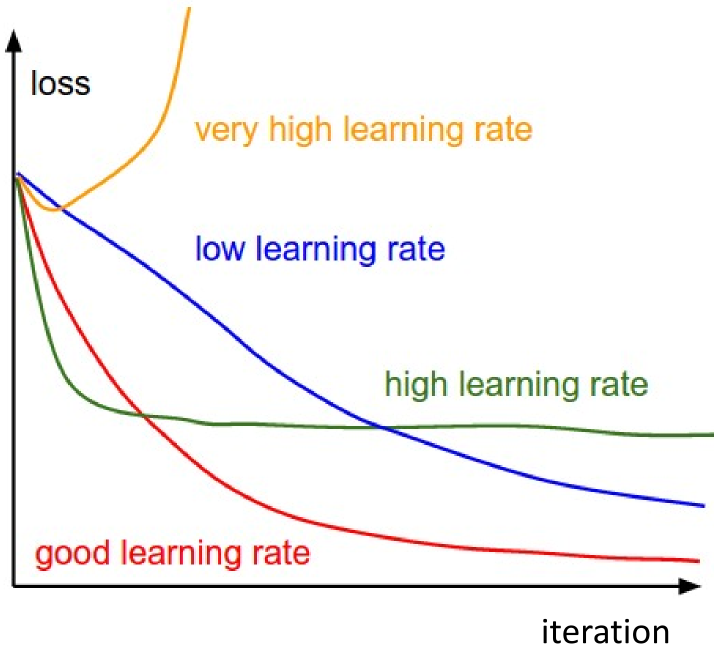
More things to try:

- Search on a log scale to get an idea of the right place to look (…0.01, 0.1, 1, 10…)
- Search coarse to fine.
- In early iterations, use a higher learning rate. Decrease it in later iterations (implement a learning rate schedule)

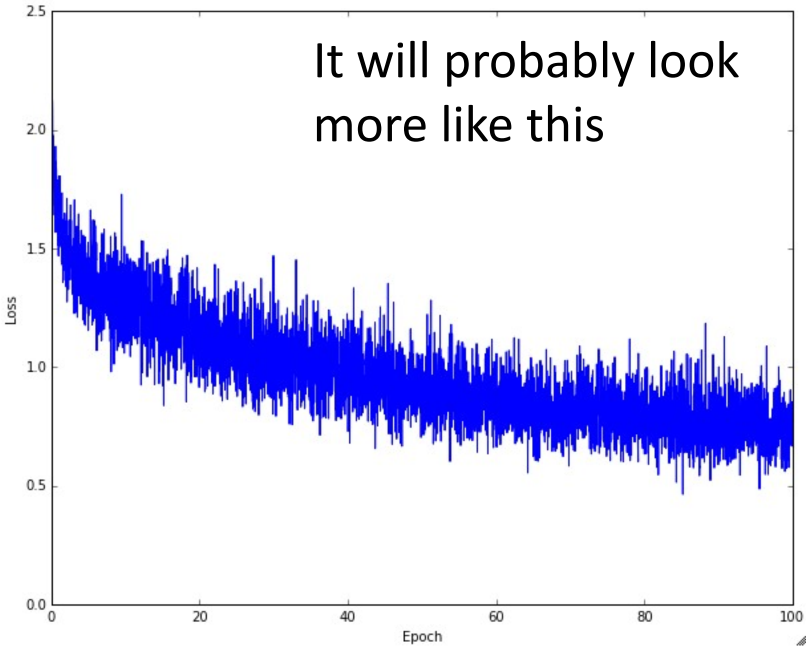
Maybe even try implementing a validation fold or cross validation to find `reg` and `lr`.

Tuning Hyperparameters

The loss curve is a plot of loss over time. It’s a good way to check your machine learning algorithm is learning.



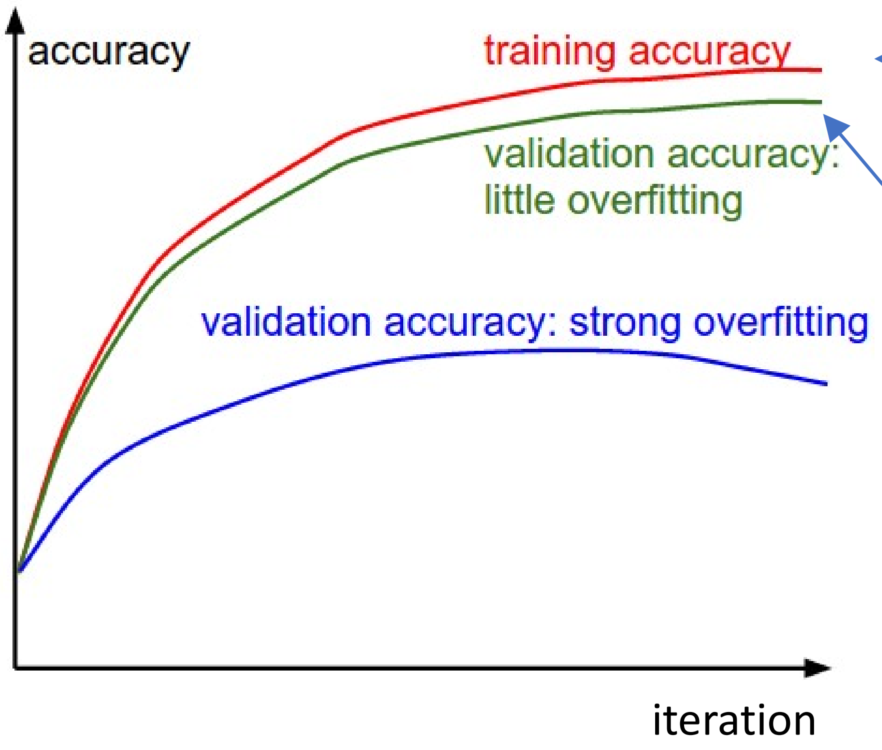
Ideally the loss curve should look like this



It will probably look more like this

Tuning Hyperparameters

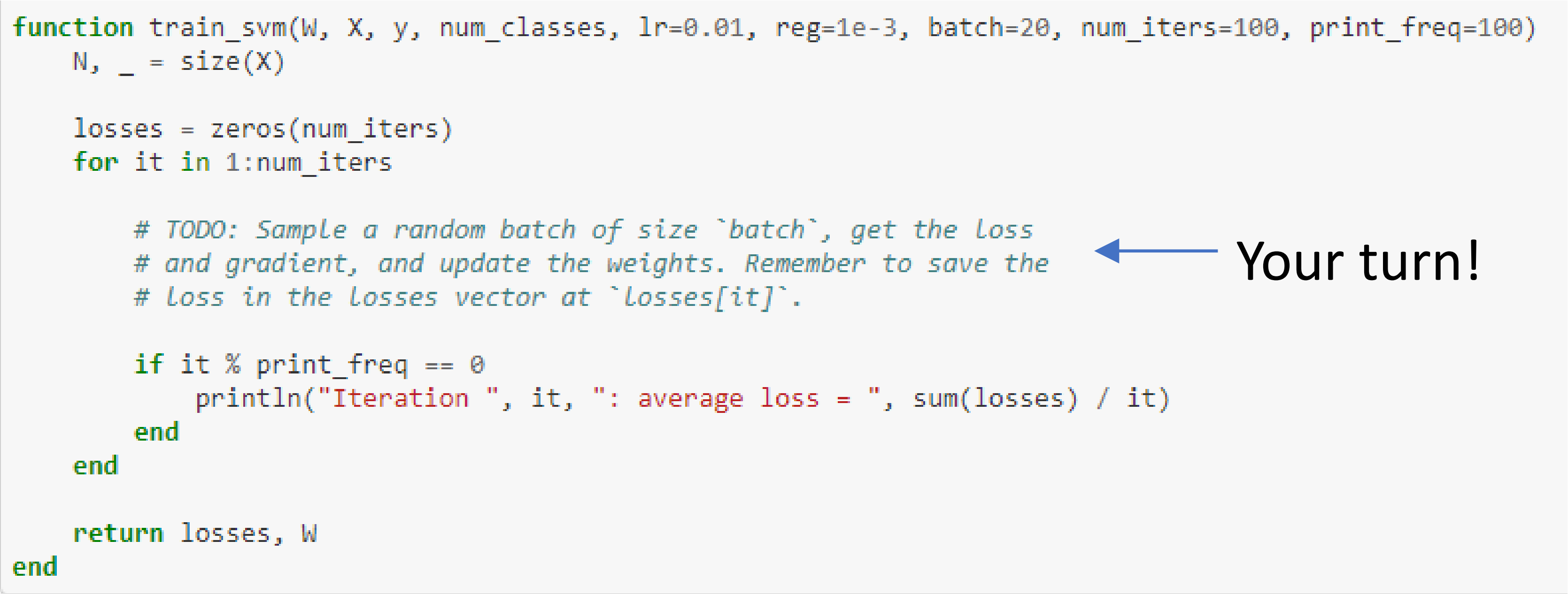
You can also look at the training and validation accuracies to see how your training is going



Training accuracy should be the highest (otherwise you may have a bug)

Validation accuracy should be lower, but still good

P4.2: Linear classifiers



```
function train_svm(W, X, y, num_classes, lr=0.01, reg=1e-3, batch=20, num_iters=100, print_freq=100)
    N, _ = size(X)

    losses = zeros(num_iters)
    for it in 1:num_iters

        # TODO: Sample a random batch of size `batch`, get the loss
        # and gradient, and update the weights. Remember to save the
        # loss in the losses vector at `losses[it]`.

        if it % print_freq == 0
            println("Iteration ", it, ": average loss = ", sum(losses) / it)
        end
    end

    return losses, W
end
```

Your turn!

Next time: Neural networks!