Lecture 16

Linear classifiers and backpropagation

Linda Shapiro Nov 20, 2025

Administrative

A4 is out

- Due Nov 25

Project

- Written Reports due Dec 8
- Oral Reports (5 min per group) on Zoom 4:30 Dec 10 for *at least* 2 hours

Administrative

No Recitation this friday

Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

1950s Age of the Perceptron

1957 The Perceptron (Rosenblatt)

1969 Perceptrons (Minsky, Papert)

1980s Age of the Neural Network

1986 Back propagation (Hinton)

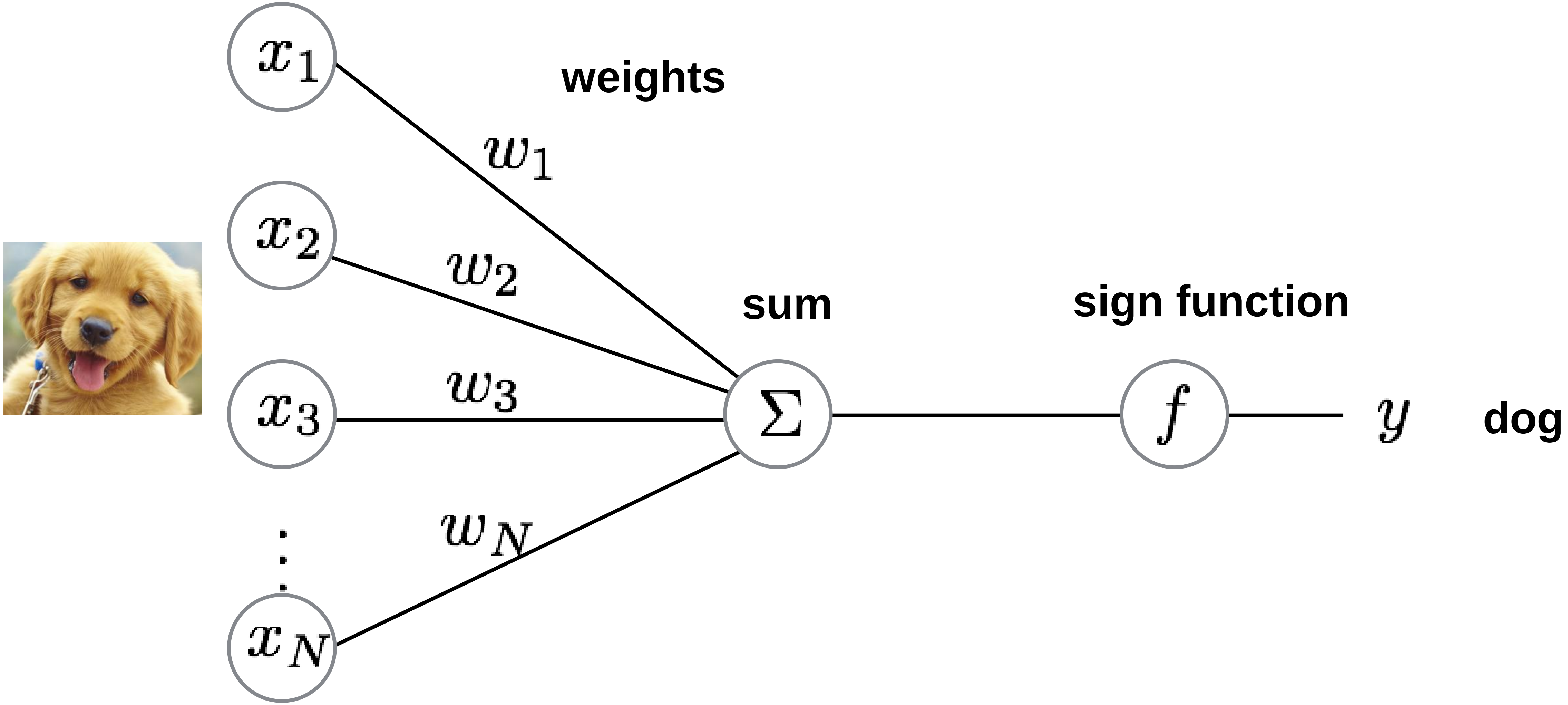
1990s Age of the Graphical Model

2000s Age of the Support Vector Machine

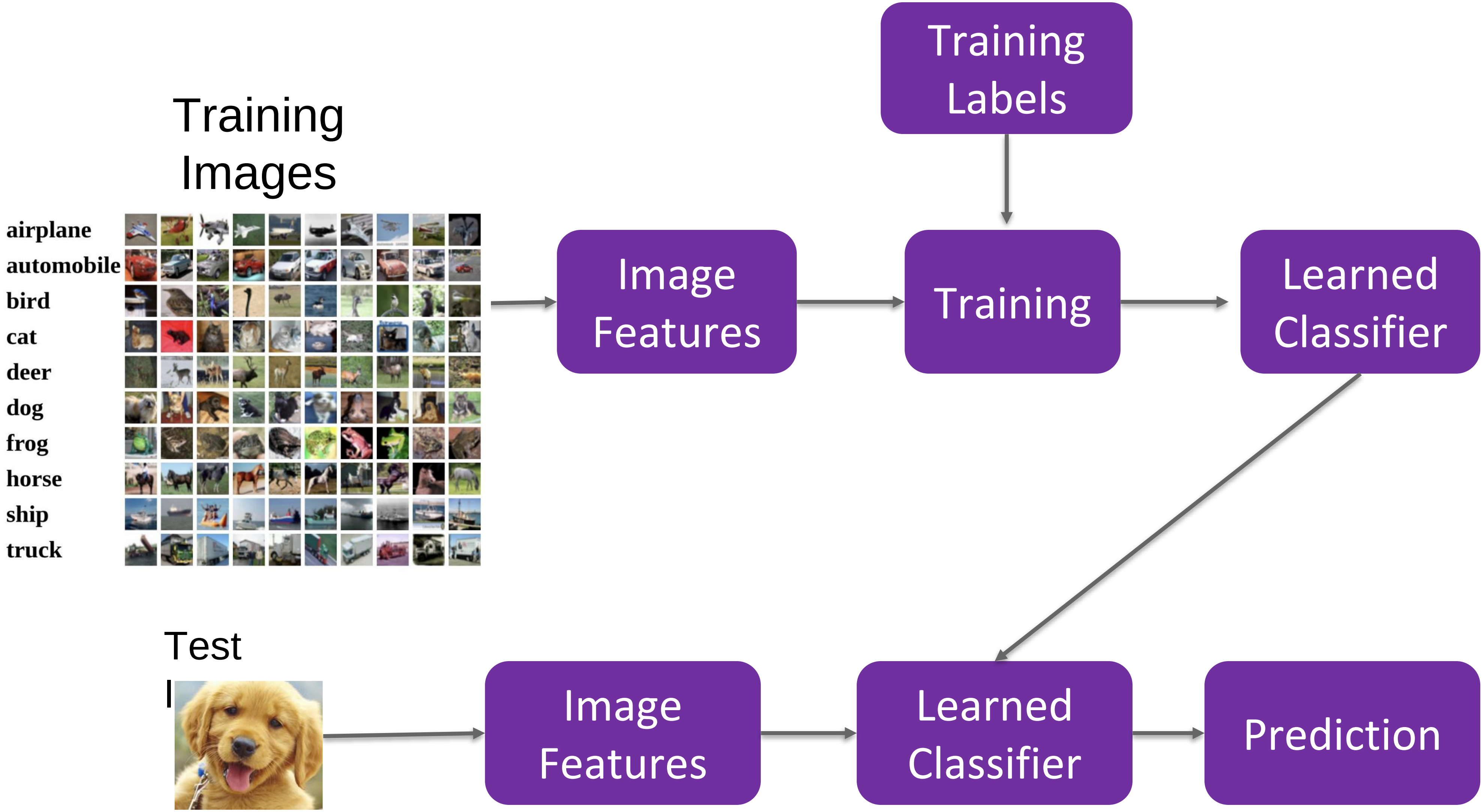
2010s Age of the Deep Network

deep learning = known algorithms + computing power + big data

Perceptron: for image classification



Let’s revisit our simple recognition pipeline to explain where perceptrons fit in



Remember we can featurize images into a vector

Image
Vector

Raw pixels
Raw pixels + (x,y)
PCA
LDA
BoW
BoW + spatial pyramids

Recall: we can featurize images into a vector

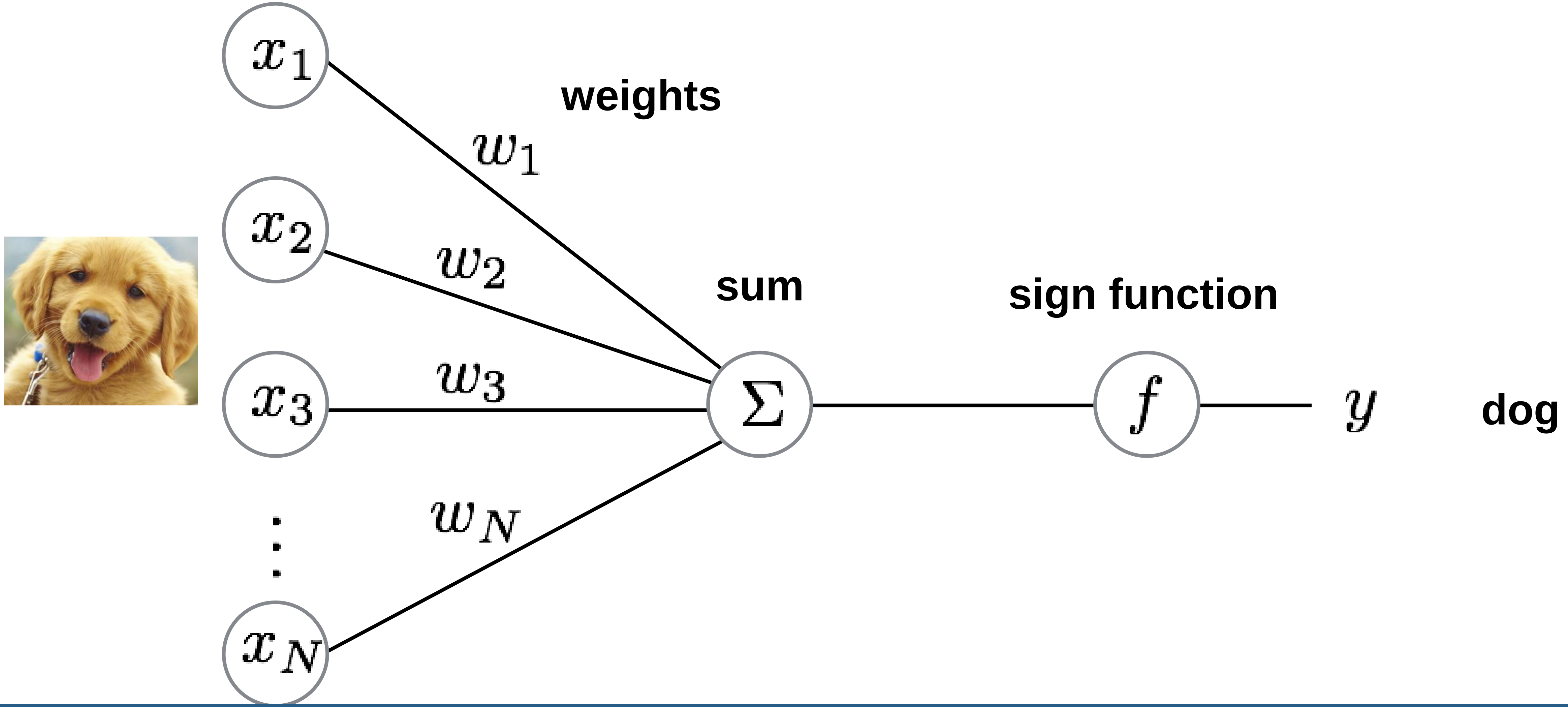
Raw pixels
Raw pixels + (x,y)
PCA
LDA
BoW
BoW + spatial pyramids

Image
Vector

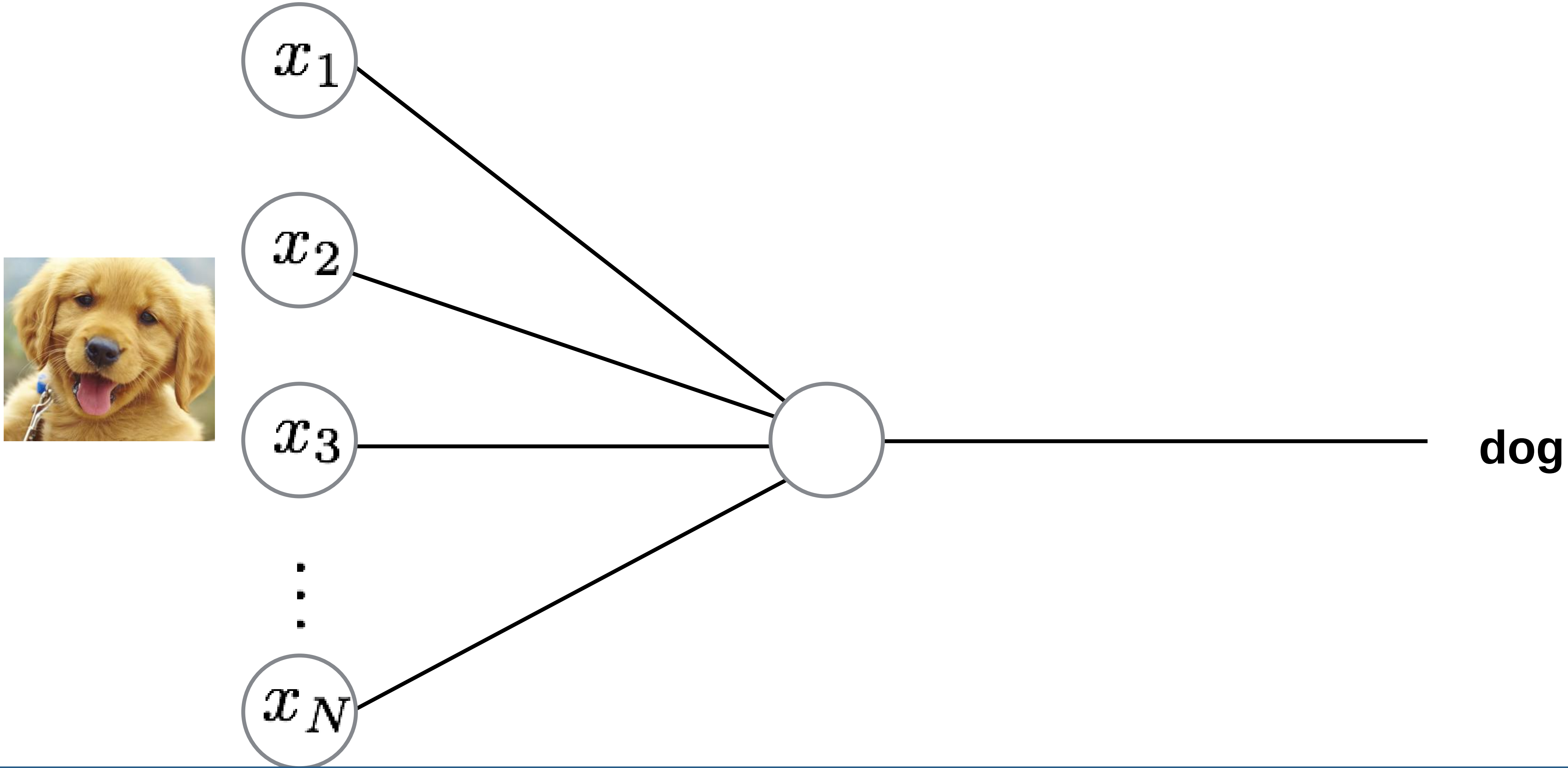
x_1
x_2
x_3
...
...
...
...
...
...
...
...
...

x_n

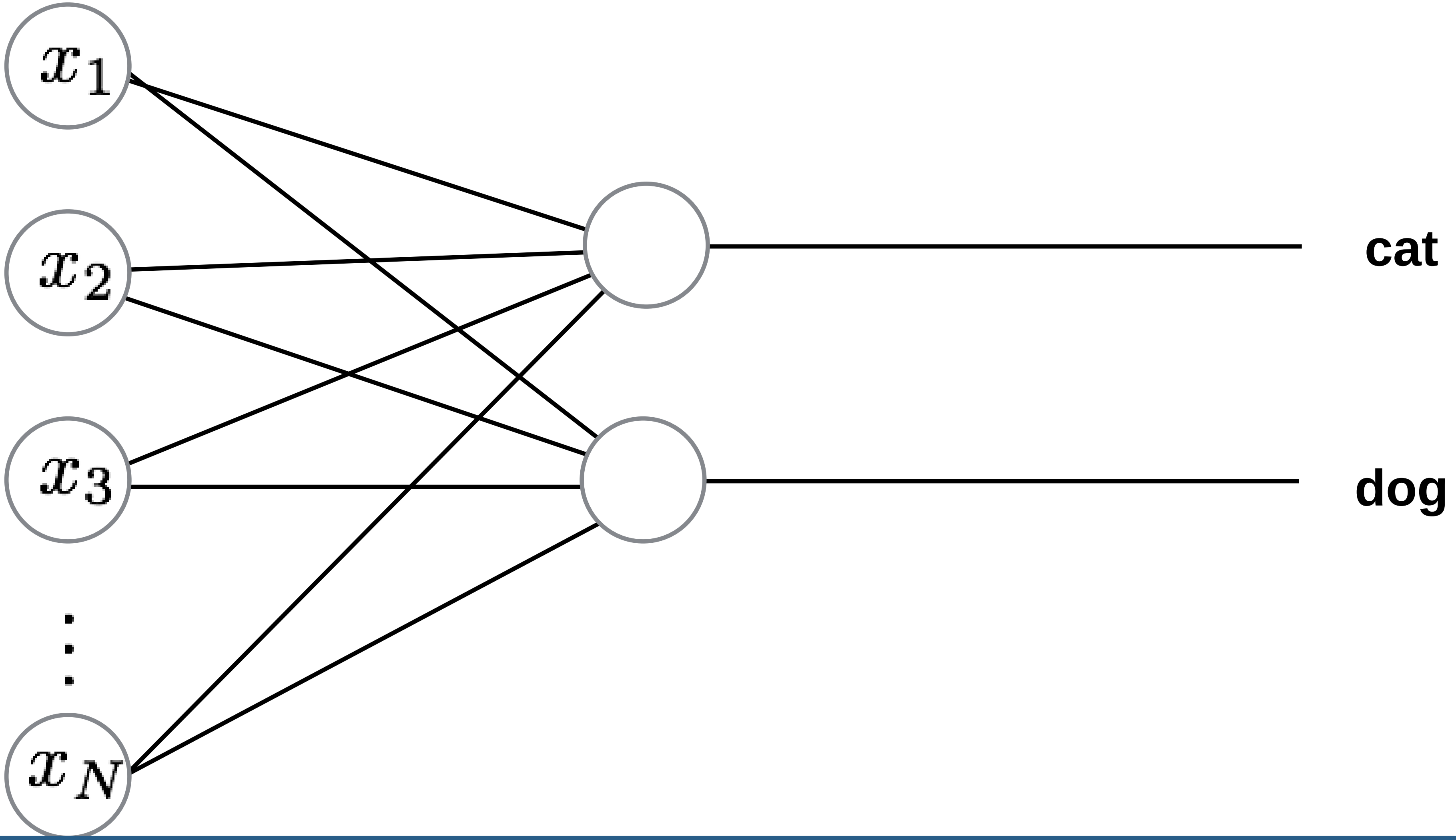
Perceptrons are a simple transformation that converts feature vectors into recognition scores



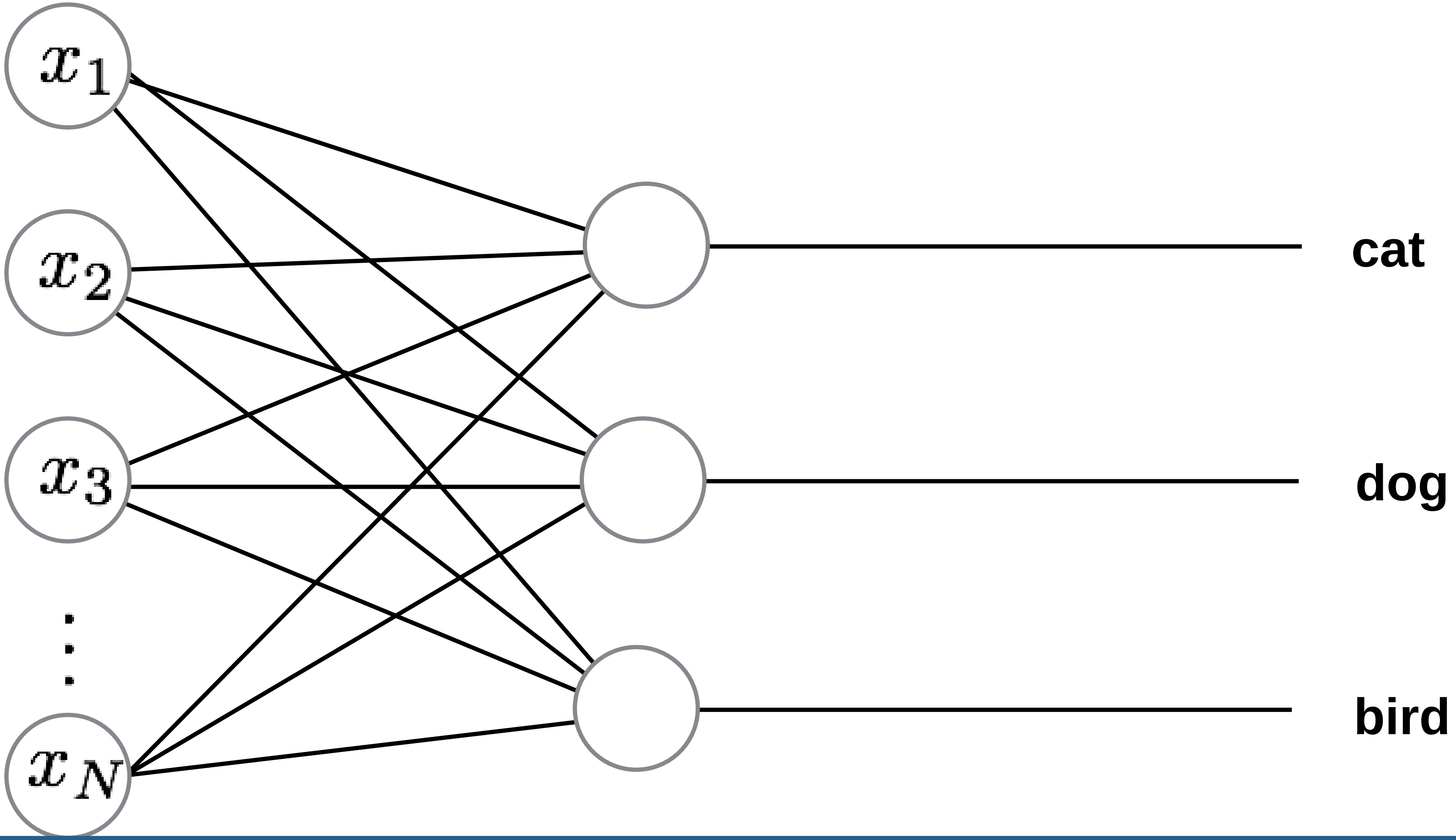
Perceptron: simplified view with one perceptron (produces 1 score for one category)



Perceptron: simplified view with two perceptrons (produces 2 scores with 2 categories)



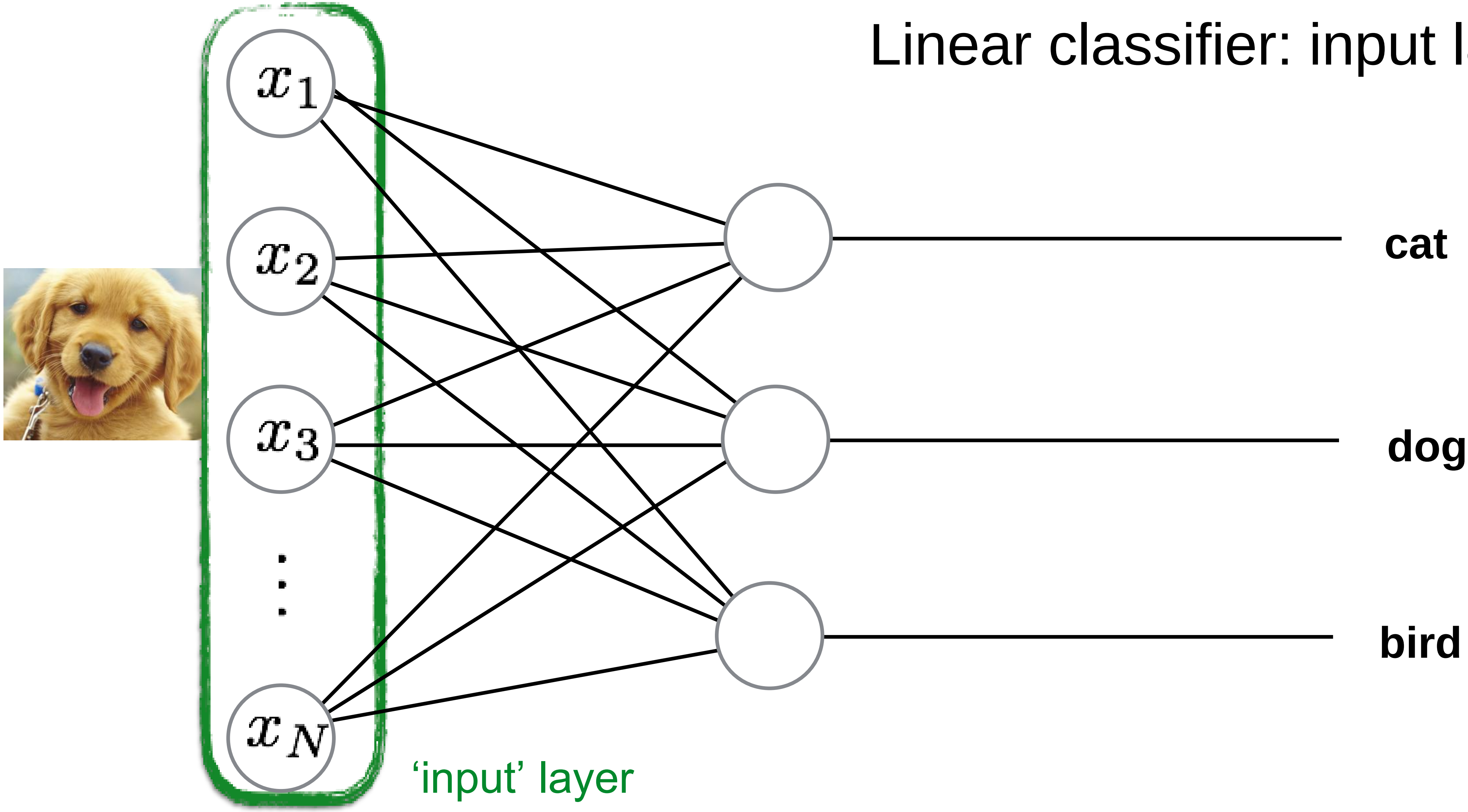
Linear classifier is a set of perceptrons produces one score for every category



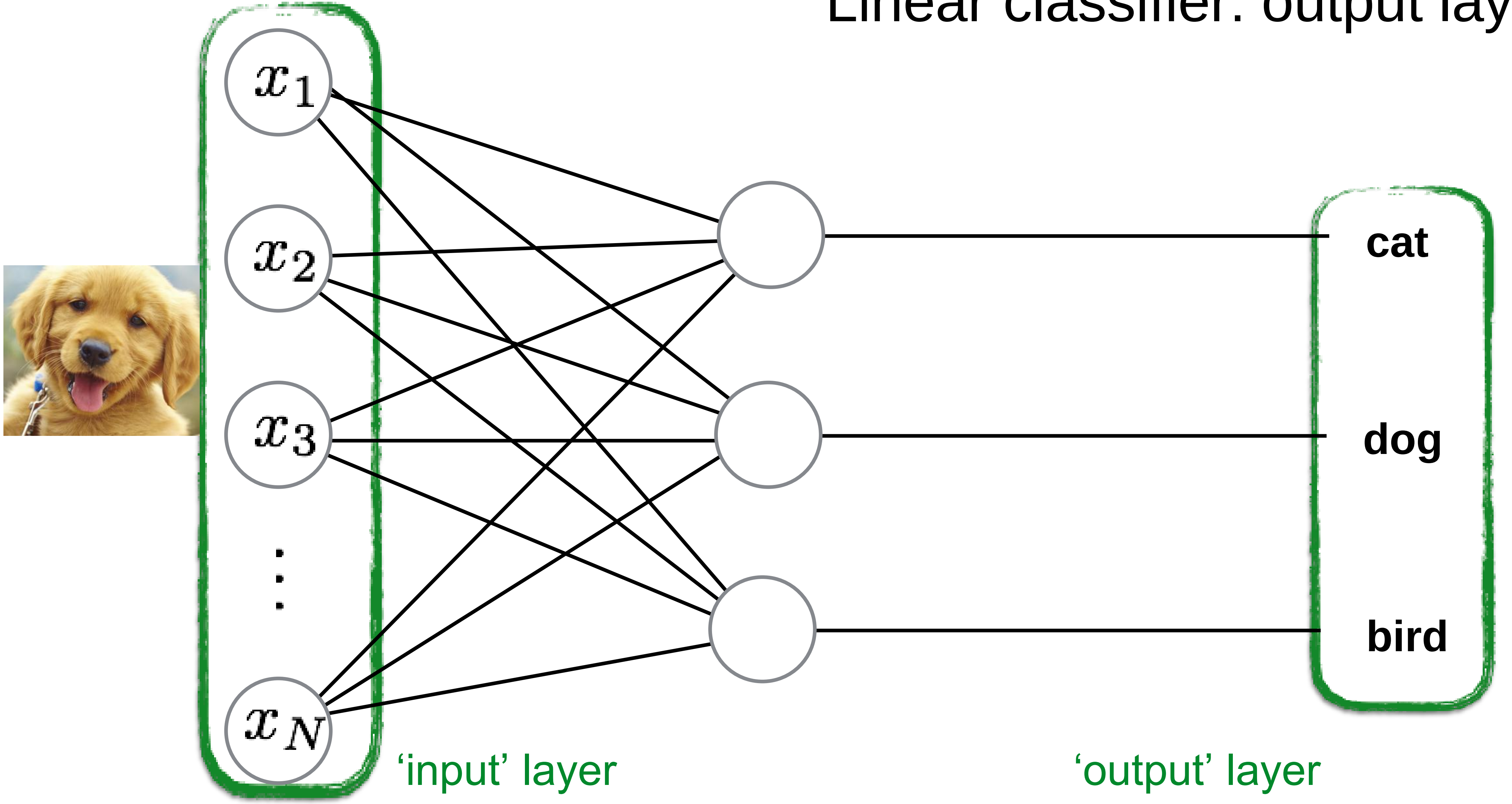
Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

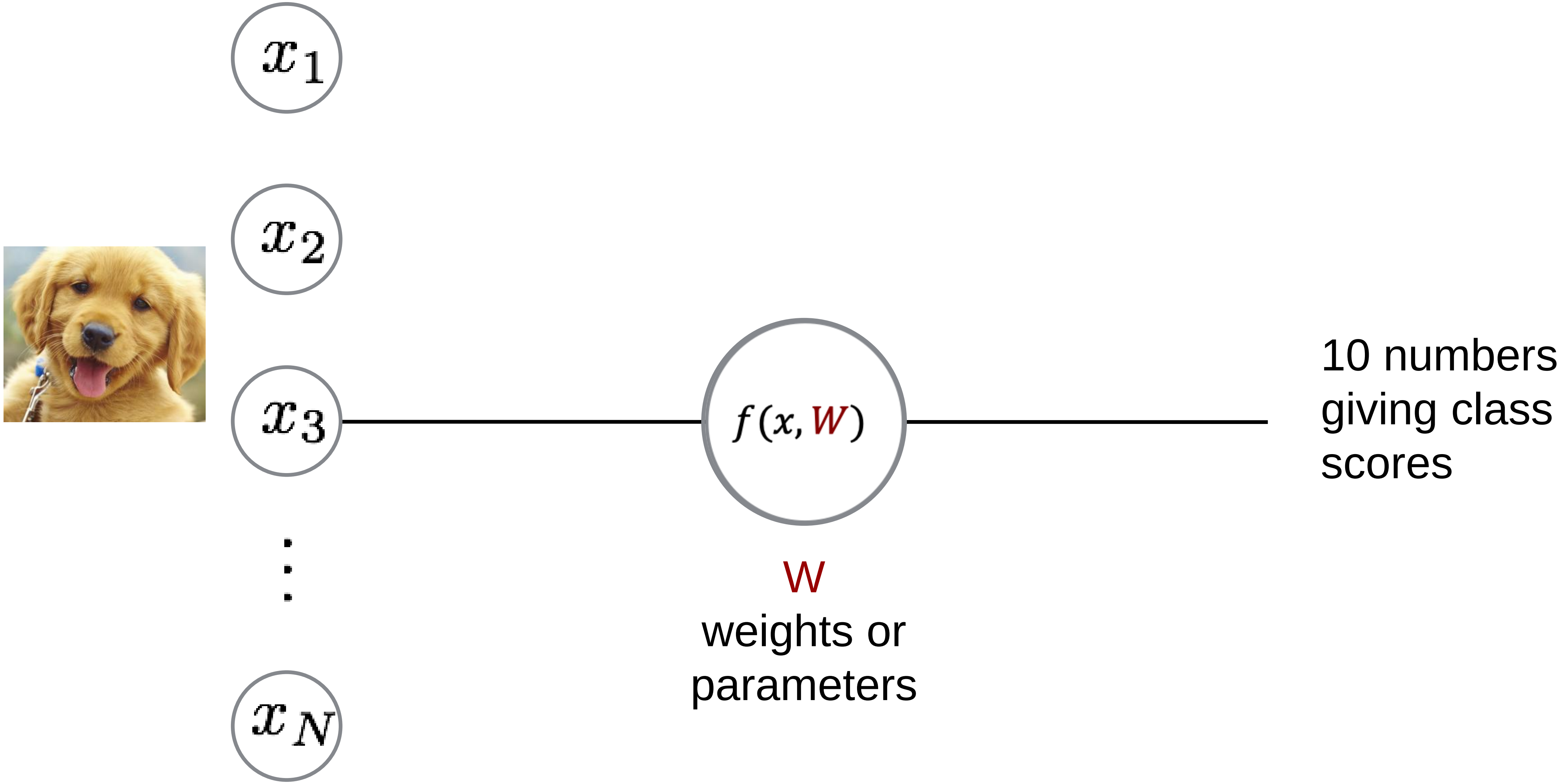
Linear classifier: input layer



Linear classifier: output layer



Linear classifier: mathematical formulation



Linear classifier: mathematical formulation with RGB features

x_1

$f(x, W) = Wx$

$x = 3072 \times 1$

$W = ?$

Q. What is the shape of W?

x_2

x_3

$f(x, W)$

10 numbers giving class scores

(32x32x3)
3072 dimensional vector

$\vdots$

W
weights or parameters

x_N

Linear classifier: mathematical formulation with RGB features

x_1

x_2

x_3

⋮

x_N

(32x32x3)
3072 dimensional vector

$f(x, W) = Wx$

x = 3072 x 1

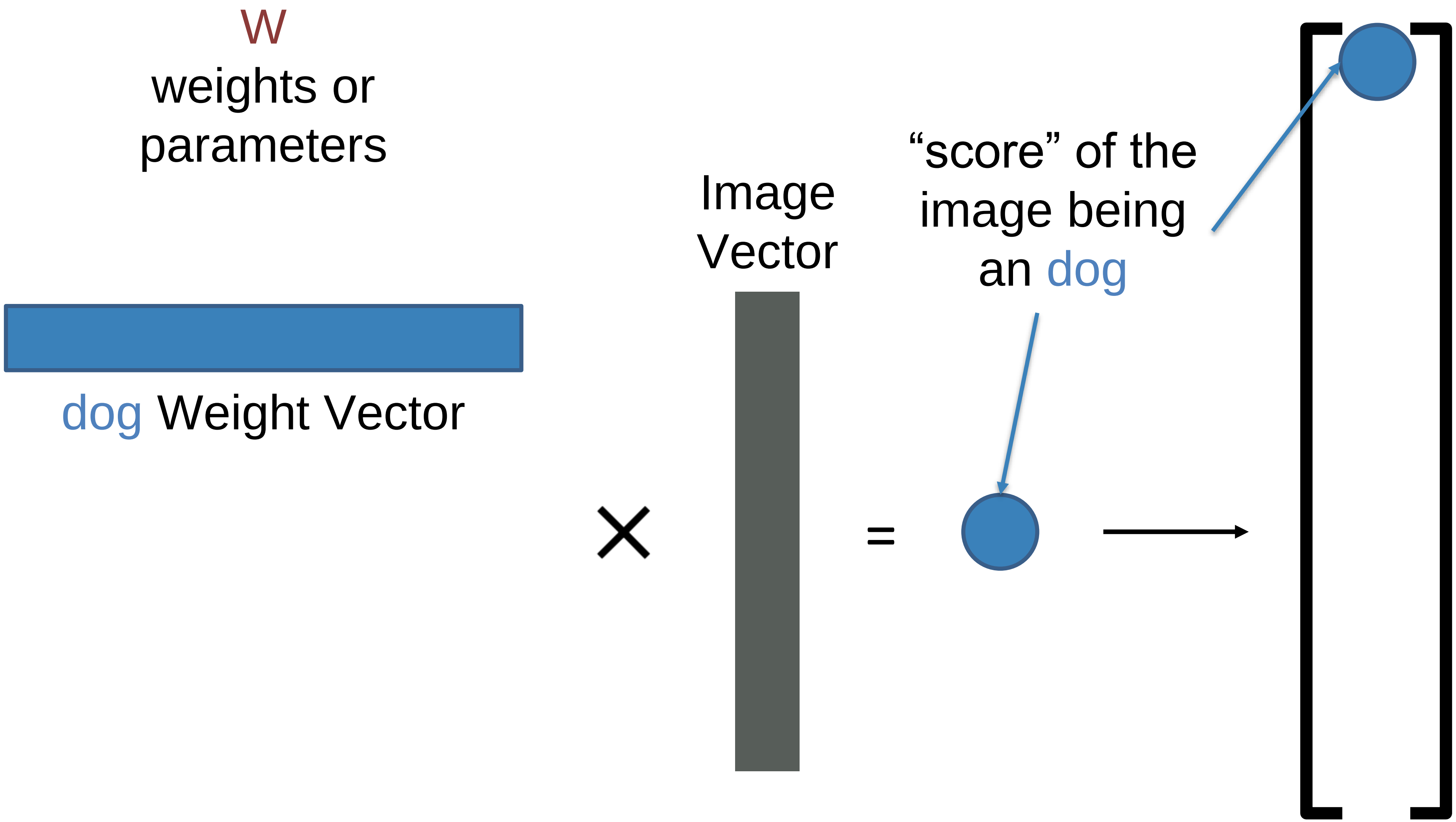
W = 10 x 3072

$f(x, W)$

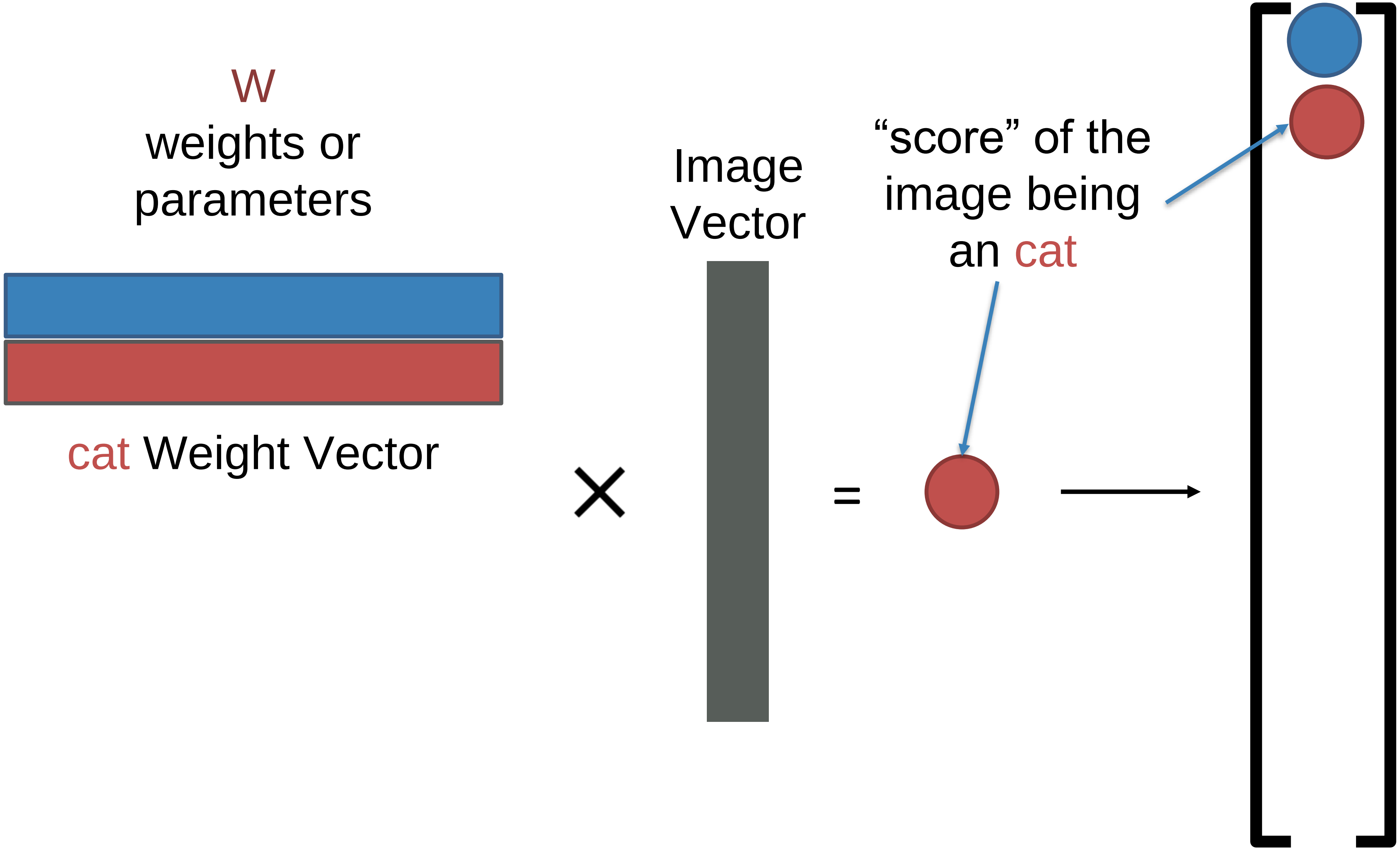
W
weights or parameters

10 numbers giving class scores

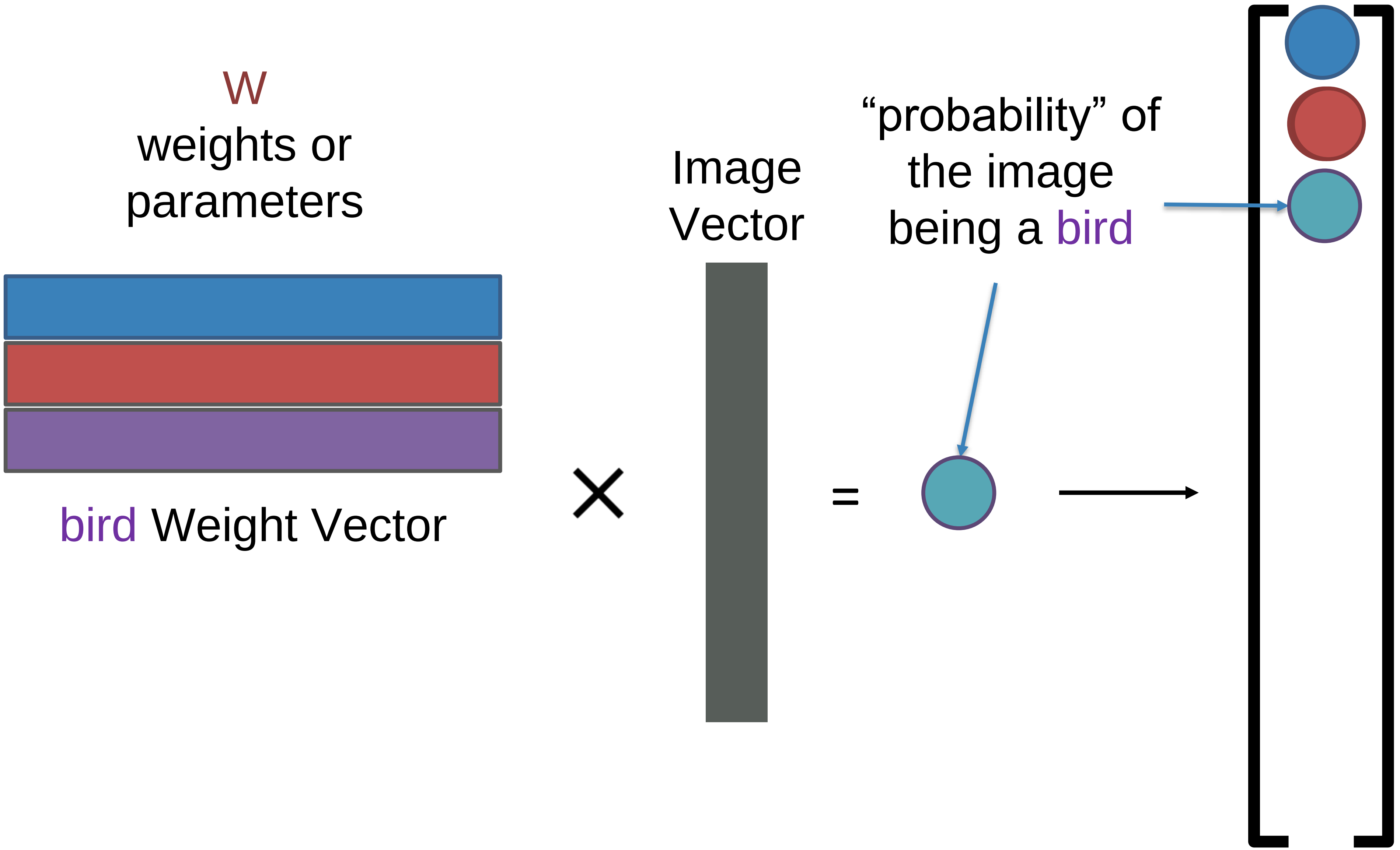
Linear classifier: function visualized



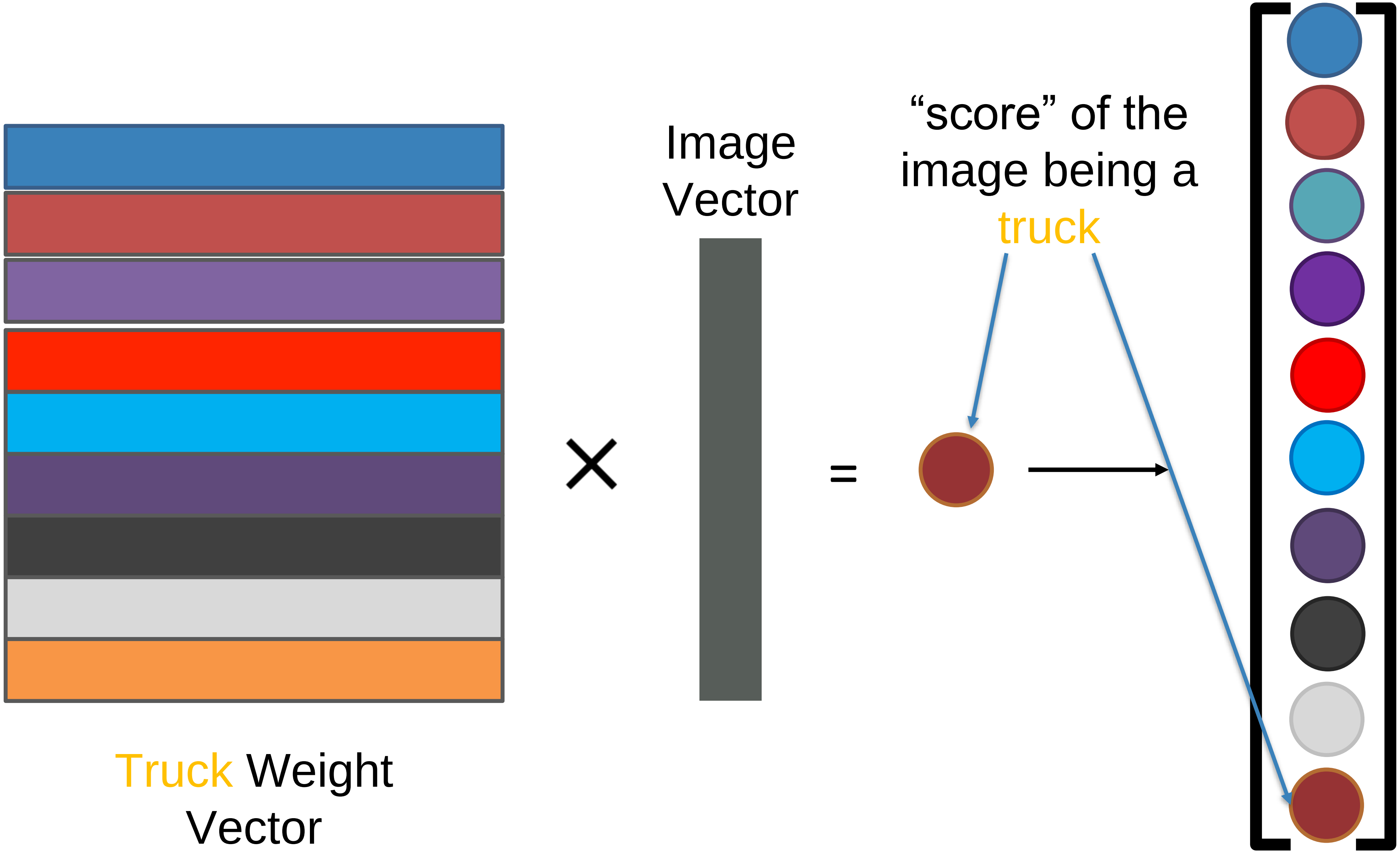
Linear classifier: function visualized



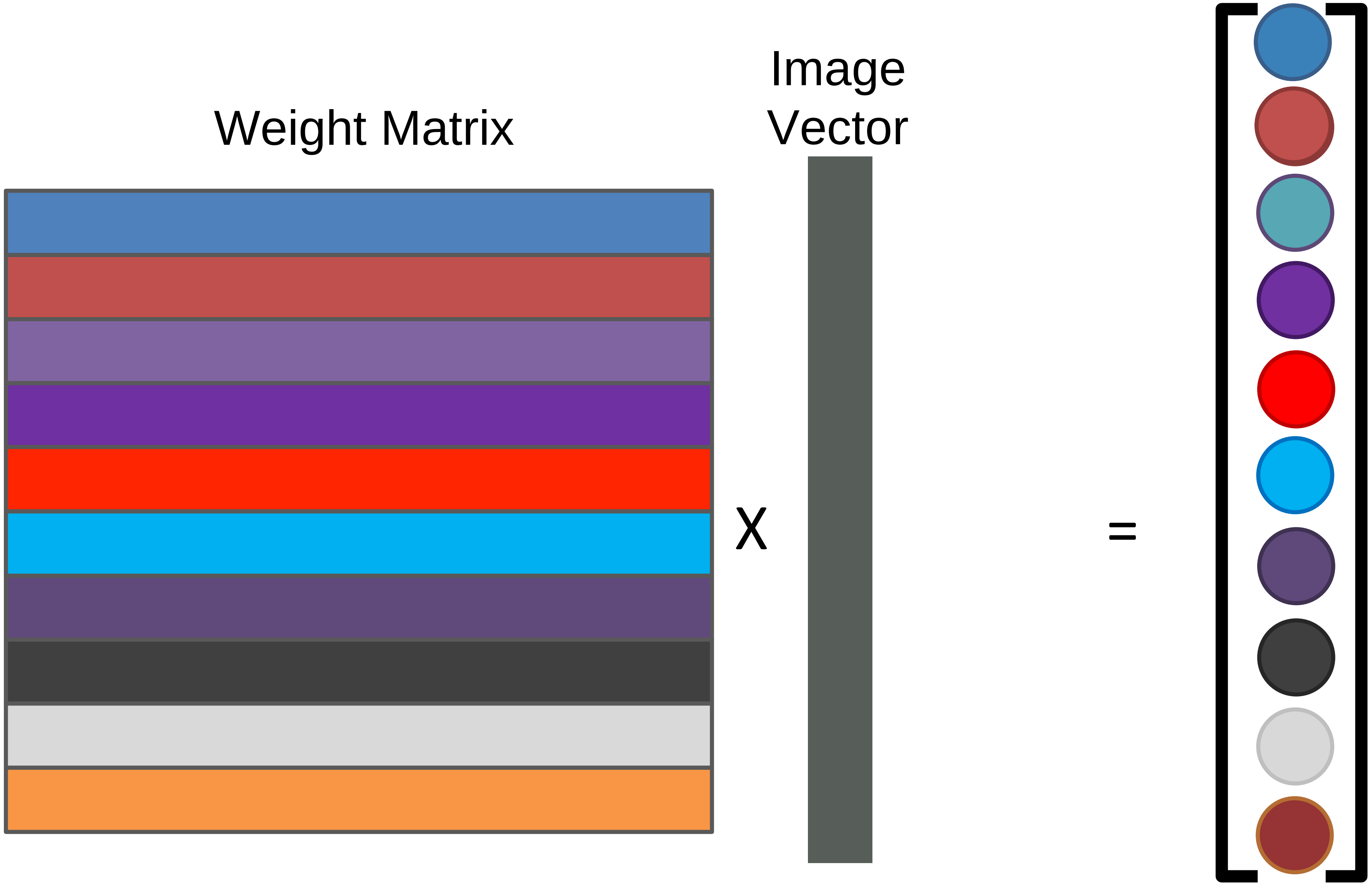
Linear classifier: function visualized



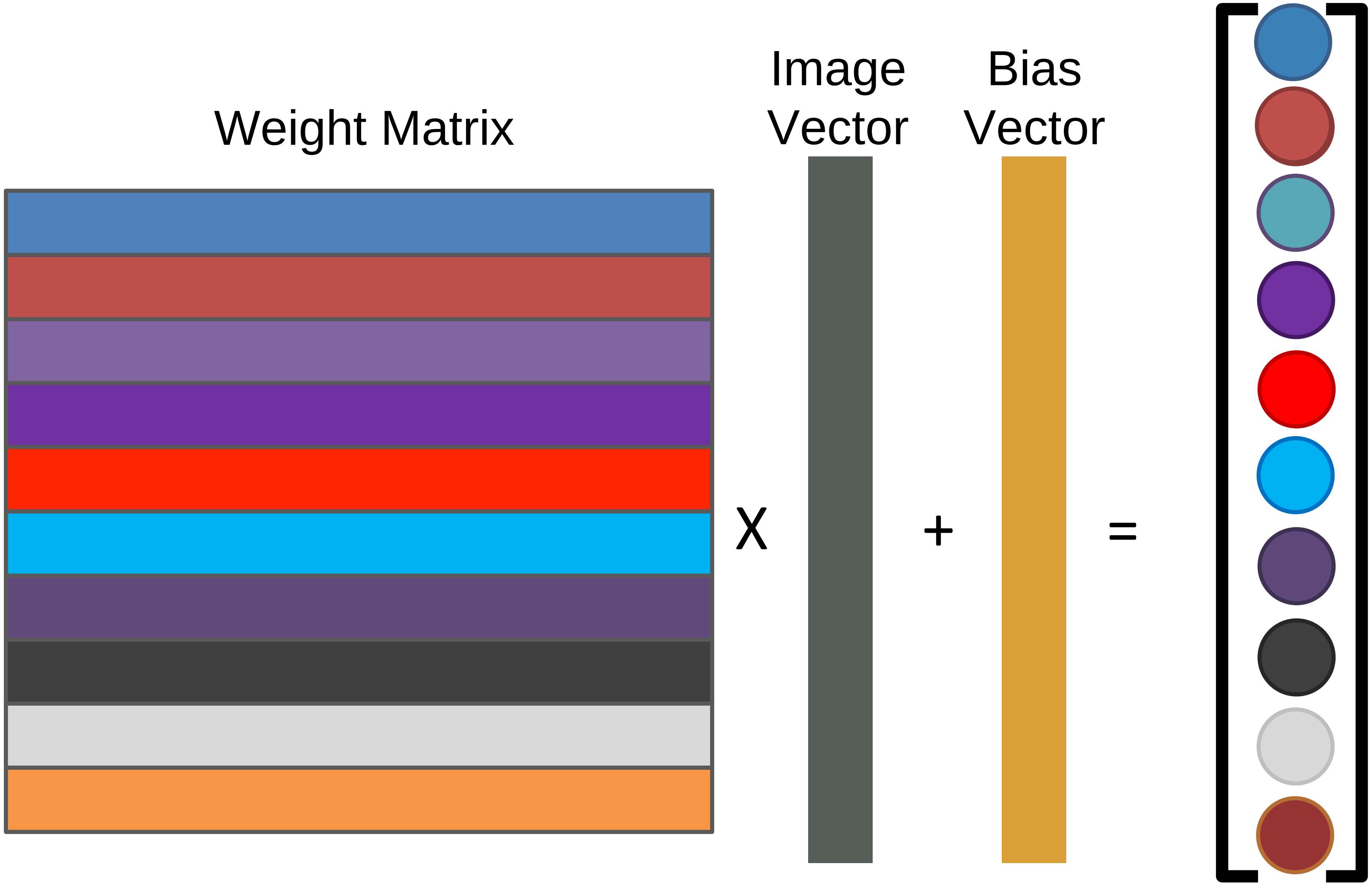
Linear classifier: function visualized



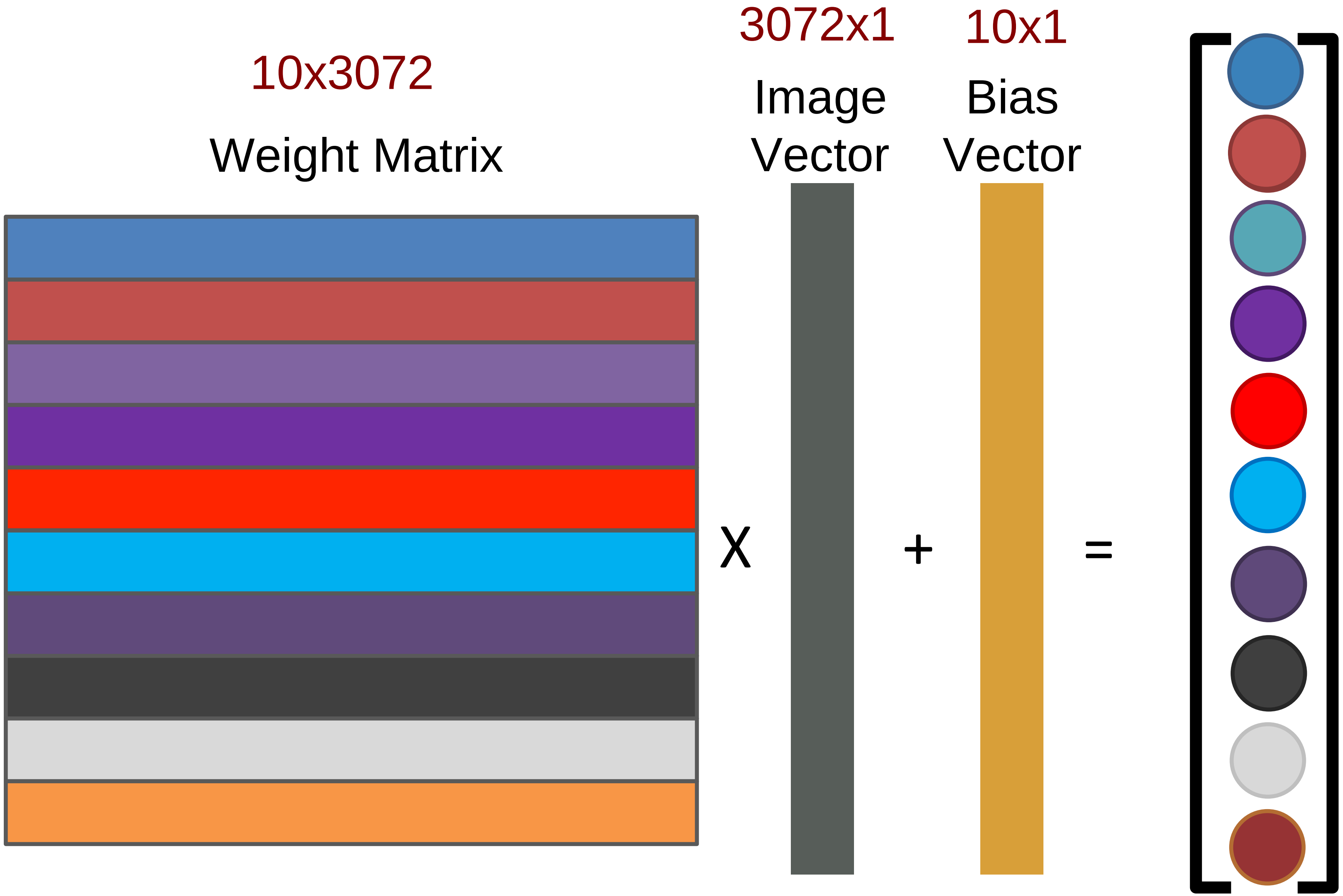
Linear classifier: function visualized



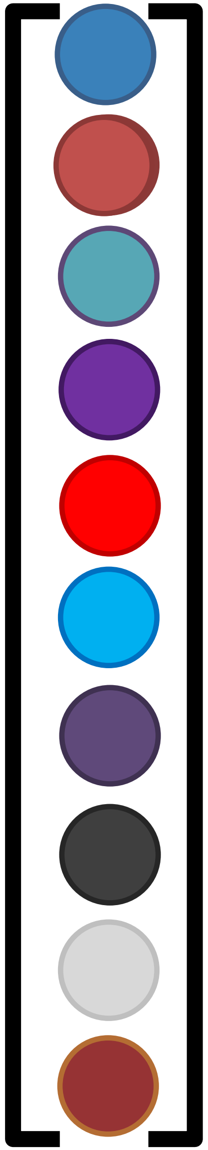
Linear classifier: bias vector



Linear classifier: size



Linear classifier: Making a classification



234
94
-104
22
1
27.3
-13
-123
99
-82

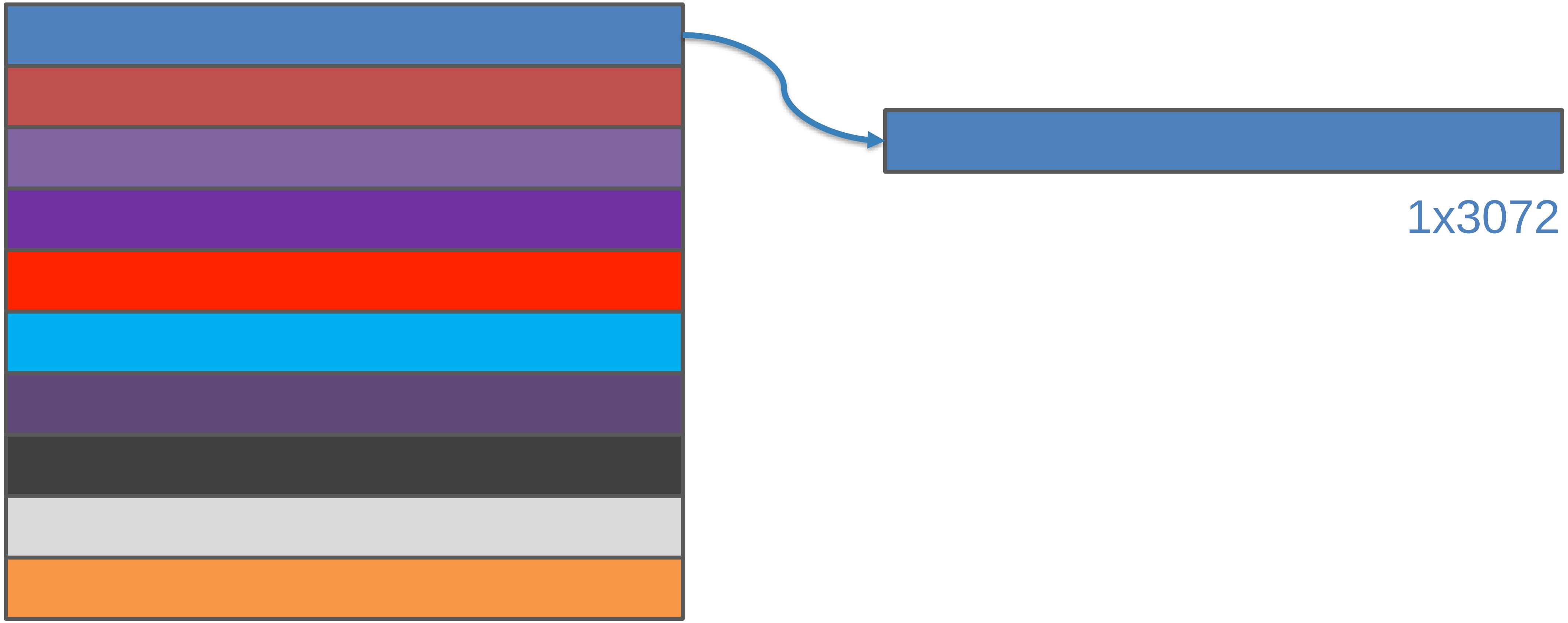
$\xrightarrow{argmax}$ **dog**

Interpreting the weights

- Assume our weights are trained on the CIFAR 10 dataset with raw pixels:

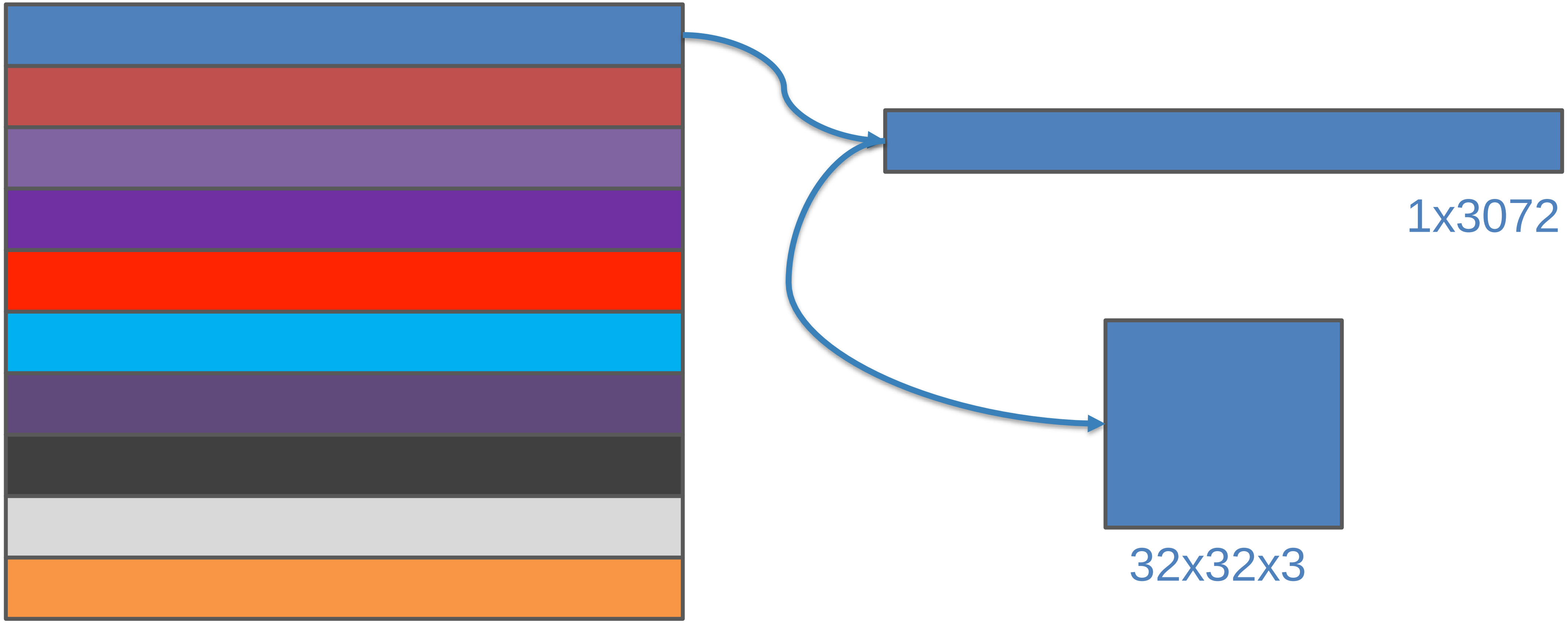
Interpreting the weights as templates

Let us look at each row of the weight matrix



Interpreting the weights as templates

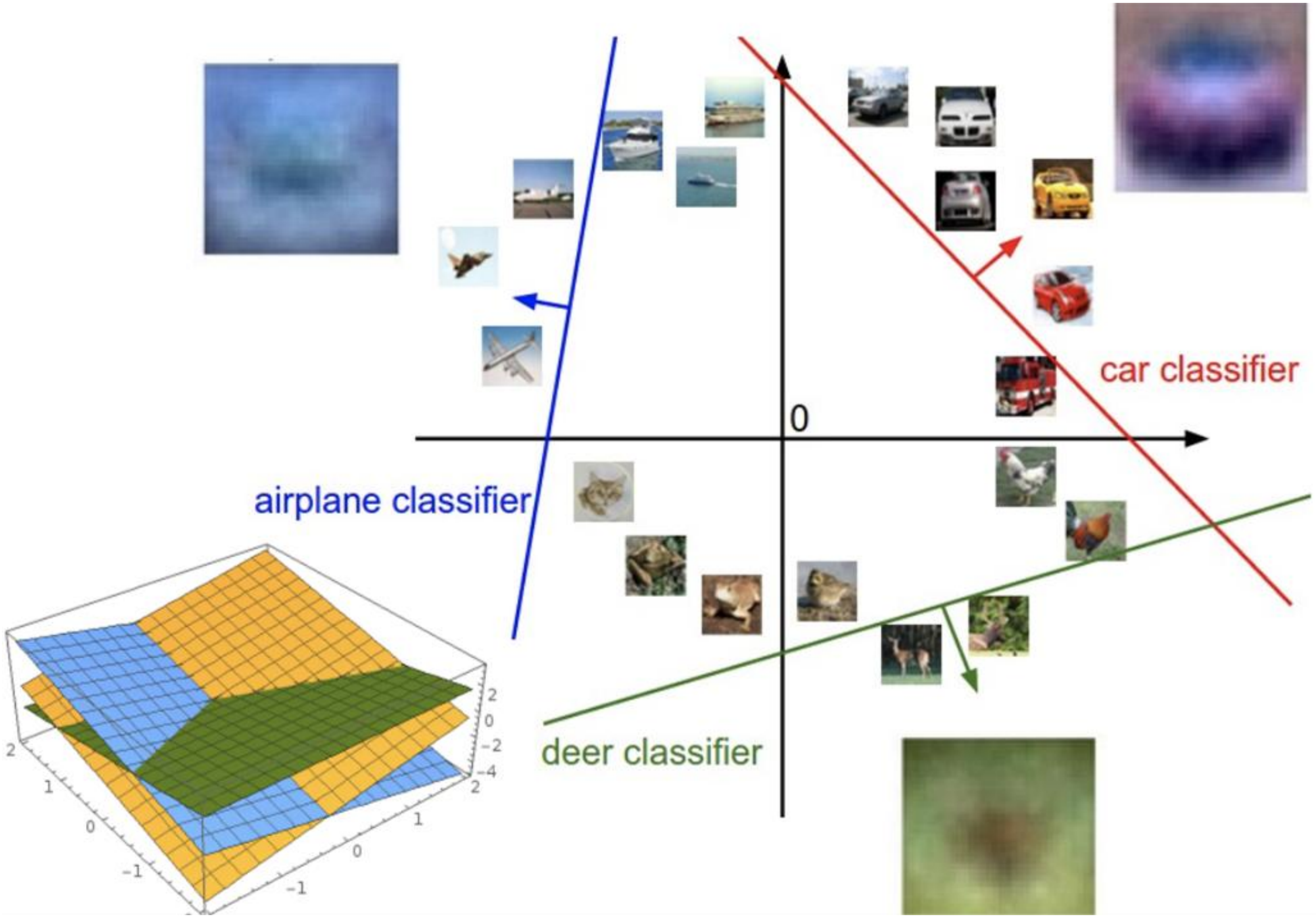
We can reshape the vector back into the shape of an image



Let's visualize what the templates look like

We can reshape the row back to the shape of an image

Interpreting the weights geometrically



- Assume the image vectors are in 2D space to make it easier to visualize.

Plot created using Wolfram Cloud

Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

Training linear classifiers

We need to learn how to pick the weights in the first place.

Formally, we need to find *W* such that

$$\min_{W} \mathrm{Loss}(y, \hat{y})$$

Where y is the true label, $\hat{y}$ is the model's predicted label.

All we have to do is define a loss function!

Given training data:

x	y
[2 3]	7
[5 1]	11
[8 8]	24

$$y = wx$$

What do you think is a good approximation weight parameter for this data point?

W = [? ?]

Given training data:

$$y = wx$$

x	y
[2 3]	7
[5 1]	11
[8 8]	24

What do you think is a good approximation weight parameter for this data point?

W = [2 1]

Properties of a loss function

Given several training examples: $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$

and a perceptron: $\hat{y} = wx$

where x is image and y is (integer) label (0 for dog, 1 for cat, etc)

A loss function $\mathrm{L_i}(y_\mathrm{i}, \hat{y}_\mathrm{i})$ tells us how good is our current classifier

- When the classifier predicts correctly, the loss should be low
- When the classifier makes mistakes, the loss should be high

Properties of a loss function

Given several training examples: $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$

and a perceptron: $\hat{y} = wx$

where x is image and y is (integer) label (0 for dog, 1 for cat, etc)
Loss over the entire dataset is an average of loss over examples

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i(y_i, \hat{y}_i)$$

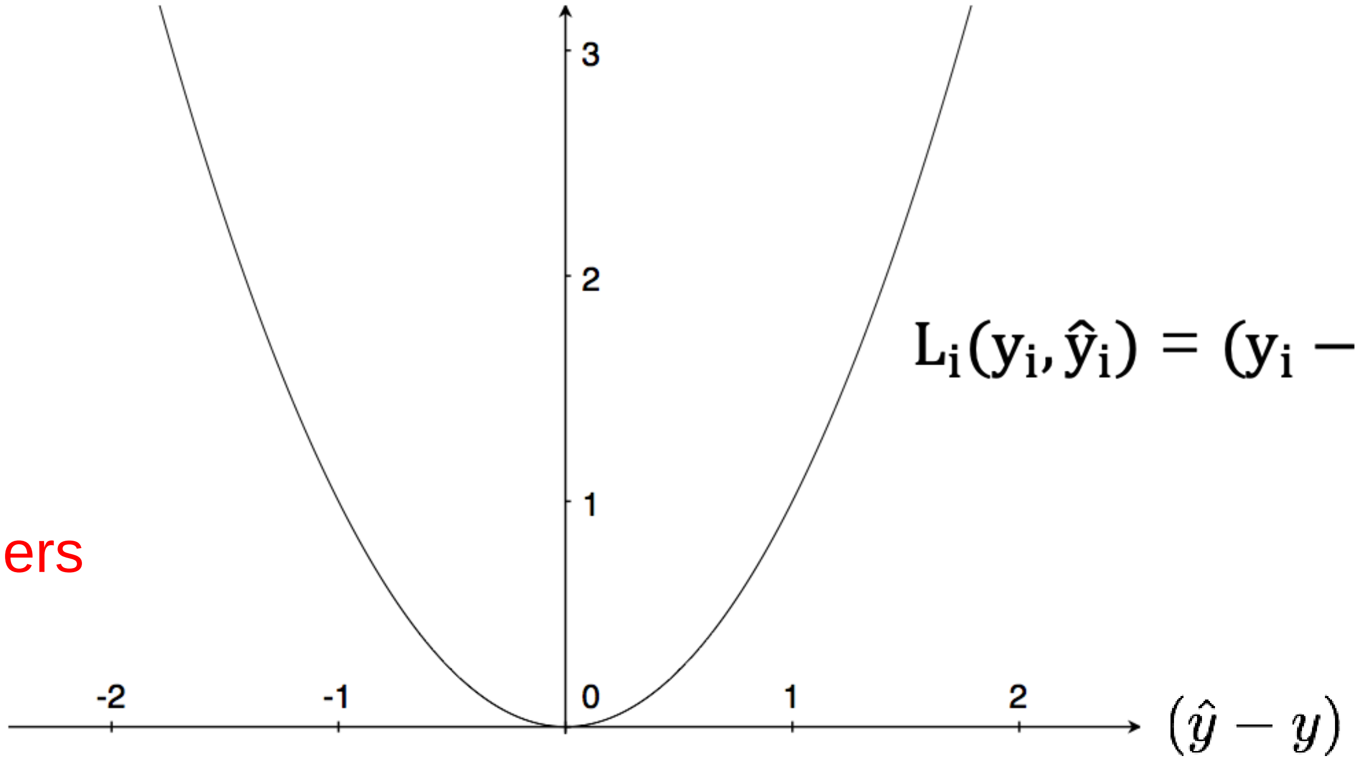
How do we choose the loss function L_i ?

YOU get to chose the loss function!

(some are better than others depending on what you want to do)

Squared Error (L2)

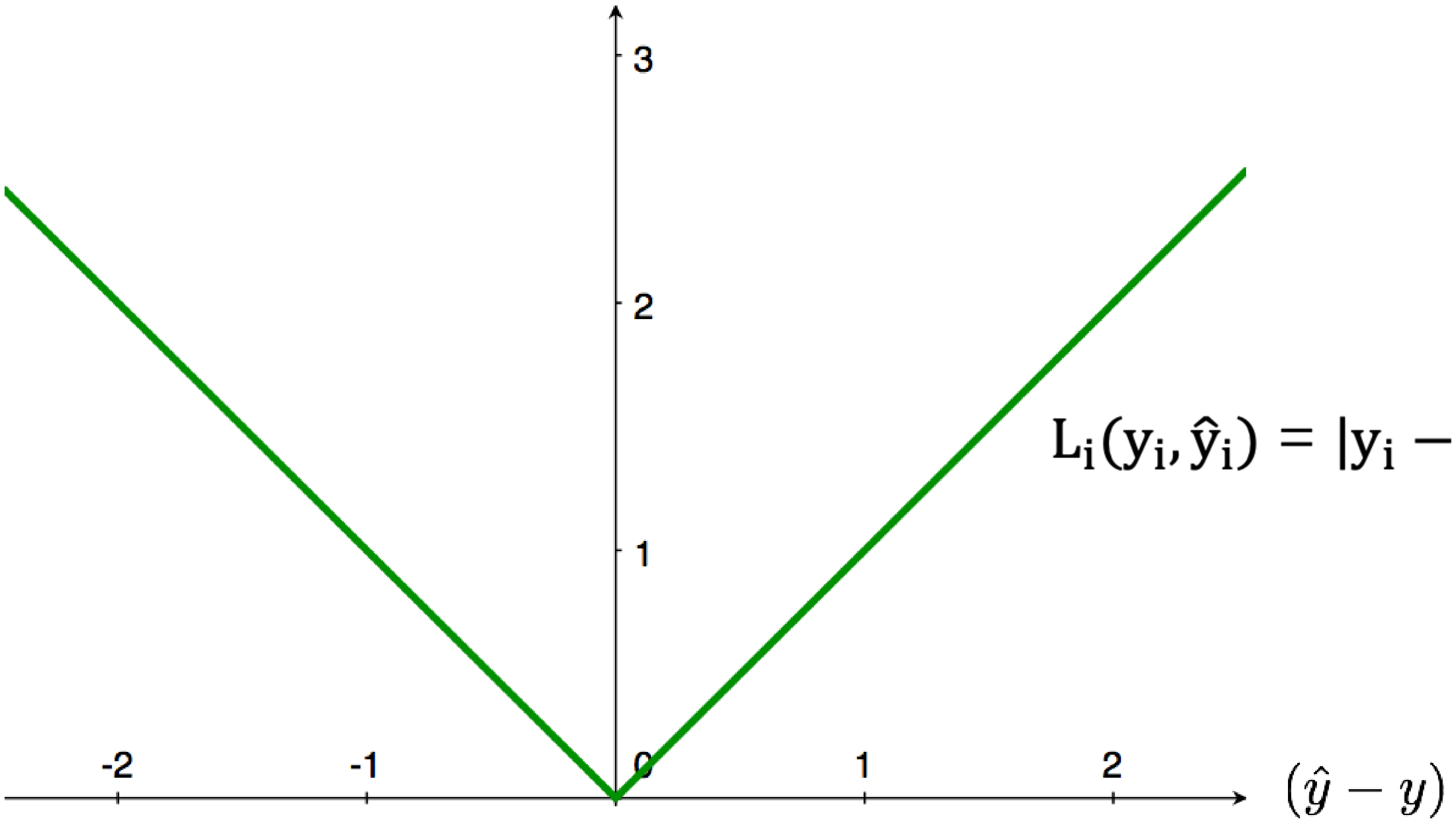
(a popular loss function) ((why?))



$$L_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

Not robust to outliers

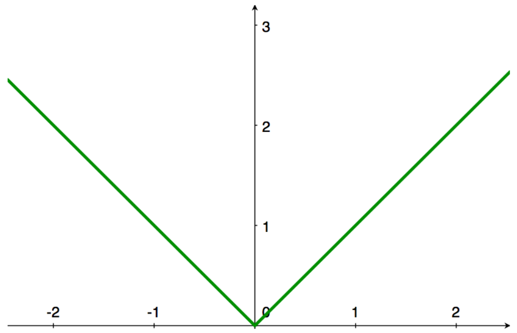
L1 loss



$$L_i(y_i, \hat{y}_i) = |y_i - \hat{y}_i|$$

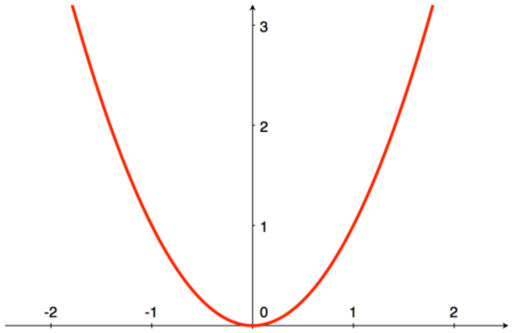
L1 Loss

$$L_i(y_i, \hat{y}_i) = |y_i - \hat{y}_i|$$



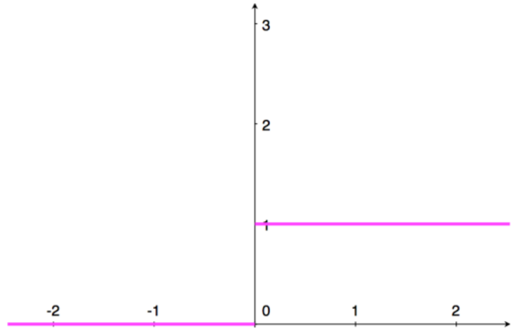
L2 Loss

$$L_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$



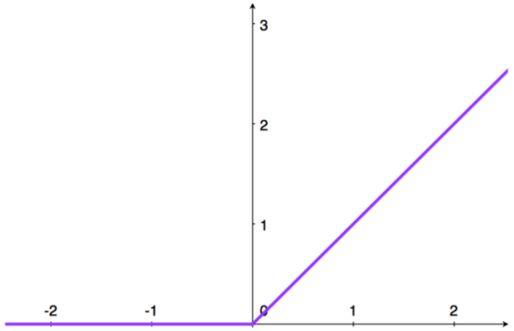
Zero-One Loss

$$L_i(y_i, \hat{y}_i) = 1||y_i \neq \hat{y}_i||$$



Hinge Loss (only if y ranges [0,1])

$$L_i(y_i, \hat{y}_i) = \max(0,\ 1 - y_i\,\hat{y}_i)$$



Softmax Classifier (Multinomial Logistic Regression)

- It allows us to treat the outputs of a model as probabilities for each class
- common way of measuring distance between probability distributions is Kullback-Leibler (KL) divergence:

$$D_{KL} = \sum_{y} P(y) \log \frac{P(y)}{Q(y)}$$

- where *P* is the ground truth distribution and *Q* is the model's output score distribution

Softmax Classifier (Multinomial Logistic Regression)

KL divergence:

$$D_{KL} = \sum_y P(y) \, log \frac{P(y)}{Q(y)}$$

In our case, *P* is only non-zero for correct class

For example, consider the case we only have 3 classes:

$$\boxed{\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}} \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array}$$

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

KL divergence:

$$D_{KL} = \sum_y P(y)\, log \frac{P(y)}{Q(y)}$$

$$= -\log Q(y) \text{ when } y \; = \; dog$$

$$= -\log Prob[f(x_i, W) = y_i]$$

$$\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{matrix}\text{dog}\\\text{cat}\\\text{bird}\end{matrix}$$

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

Remember our linear classifier: $\hat{y} = wx$

There are no limitations on the output space. Meaning that the model can output <0 or >1

$$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix}$$

model outputs

$$\begin{array}{cl} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array} \end{array}$$

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

We need a mechanism to convert or normalize the output into probability range [0, 1]

Solution: SOFTMAX: $Prob[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$

$$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix}$$

model outputs

$$\begin{array}{cl} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array} \end{array}$$

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

We need a mechanism to convert or normalize the output into probability range [0, 1]

Solution: SOFTMAX: $$Prob[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$$

$$\begin{bmatrix}3.2\\5.1\\-1.7\end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix}24.5\\164\\0.18\end{bmatrix} \qquad \xleftrightarrow{\text{loss}} \qquad \begin{array}{cl}\begin{bmatrix}1\\0\\0\end{bmatrix} & \begin{array}{l}\text{dog}\\\text{cat}\\\text{bird}\end{array}\end{array}$$

model outputs

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

We need a mechanism to convert or normalize the output into probability range [0, 1]

Solution: SOFTMAX: $Prob[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$

$$\begin{bmatrix}3.2\\5.1\\-1.7\end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix}24.5\\164\\0.18\end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix}0.13\\0.87\\0.00\end{bmatrix} \xleftrightarrow{\text{loss}} \begin{bmatrix}1\\0\\0\end{bmatrix} \begin{array}{l}\text{dog}\\\text{cat}\\\text{bird}\end{array}$$

model outputs — probabilities — correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

In this case, what is the loss:

$$L_i = ??$$

$$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 24.5 \\ 164 \\ 0.18 \end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix} 0.13 \\ 0.87 \\ 0.00 \end{bmatrix} \xleftrightarrow{\text{loss}} \begin{array}{c} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \end{array} \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array}$$

model outputs

probabilities

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

In this case, what is the loss:

Probability model computed for the correct class (dog)

$$L_i = -\log(0.13) = 2.04$$

$$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 24.5 \\ 164 \\ 0.18 \end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix} 0.13 \\ 0.87 \\ 0.00 \end{bmatrix} \xleftrightarrow{\text{loss}} \begin{array}{cc} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array} \end{array}$$

model outputs

probabilities

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

what is the minimum and maximum values that the loss can be?

$$\begin{bmatrix}3.2\\5.1\\-1.7\end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix}24.5\\164\\0.18\end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix}0.13\\0.87\\0.00\end{bmatrix} \xleftrightarrow{\text{loss}} \begin{array}{cl} \begin{bmatrix}1\\0\\0\end{bmatrix} & \begin{array}{l}\text{dog}\\\text{cat}\\\text{bird}\end{array}\end{array}$$

model outputs

probabilities

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

At initialization, all the weights will be random. In this case, we can assume that the outputs will have the same probabilities, then what will the initial loss be?

$$\begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 24.5 \\ 164 \\ 0.18 \end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix} 0.13 \\ 0.87 \\ 0.00 \end{bmatrix} \xleftrightarrow{\text{loss}} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{array}{l} \text{dog} \\ \text{cat} \\ \text{bird} \end{array}$$

model outputs

probabilities

correct outputs

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log Prob[f(x_i, W) == y_i]$$

At initialization, all the weights will be random. In this case, we can assume that the outputs will have the same probabilities, then what will the initial loss be given C=10 classes?

$$L_i = -\log\left(\frac{1}{C}\right) = \log(C) = \log(10) = 2.03$$

$$\begin{bmatrix}3.2\\5.1\\-1.7\end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix}24.5\\164\\0.18\end{bmatrix} \xrightarrow{\text{norm}} \begin{bmatrix}0.13\\0.87\\0.00\end{bmatrix} \xleftrightarrow{\text{loss}} \begin{bmatrix}1\\0\\0\end{bmatrix}\begin{array}{l}\text{dog}\\\text{cat}\\\text{bird}\end{array}$$

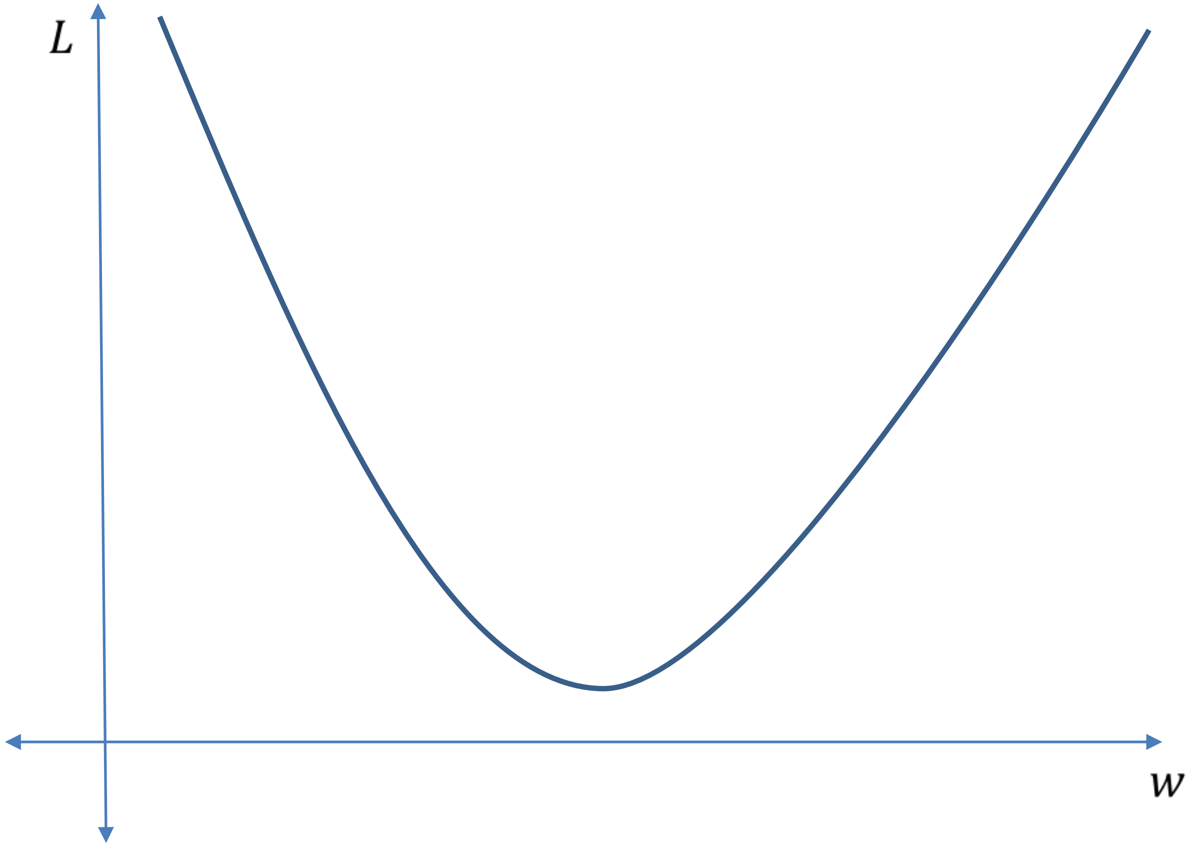
model outputs — probabilities — correct outputs

Today's agenda

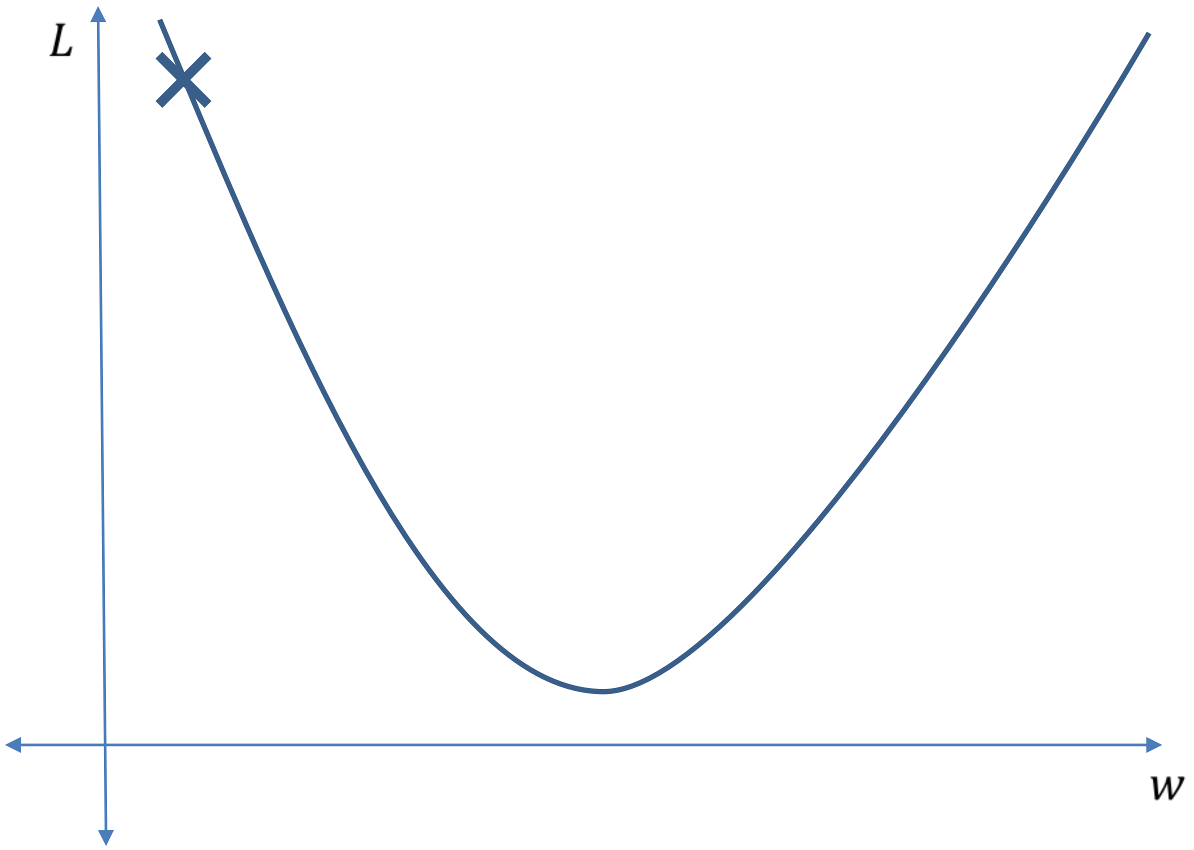
- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

How do we find the weights that minimize the loss?

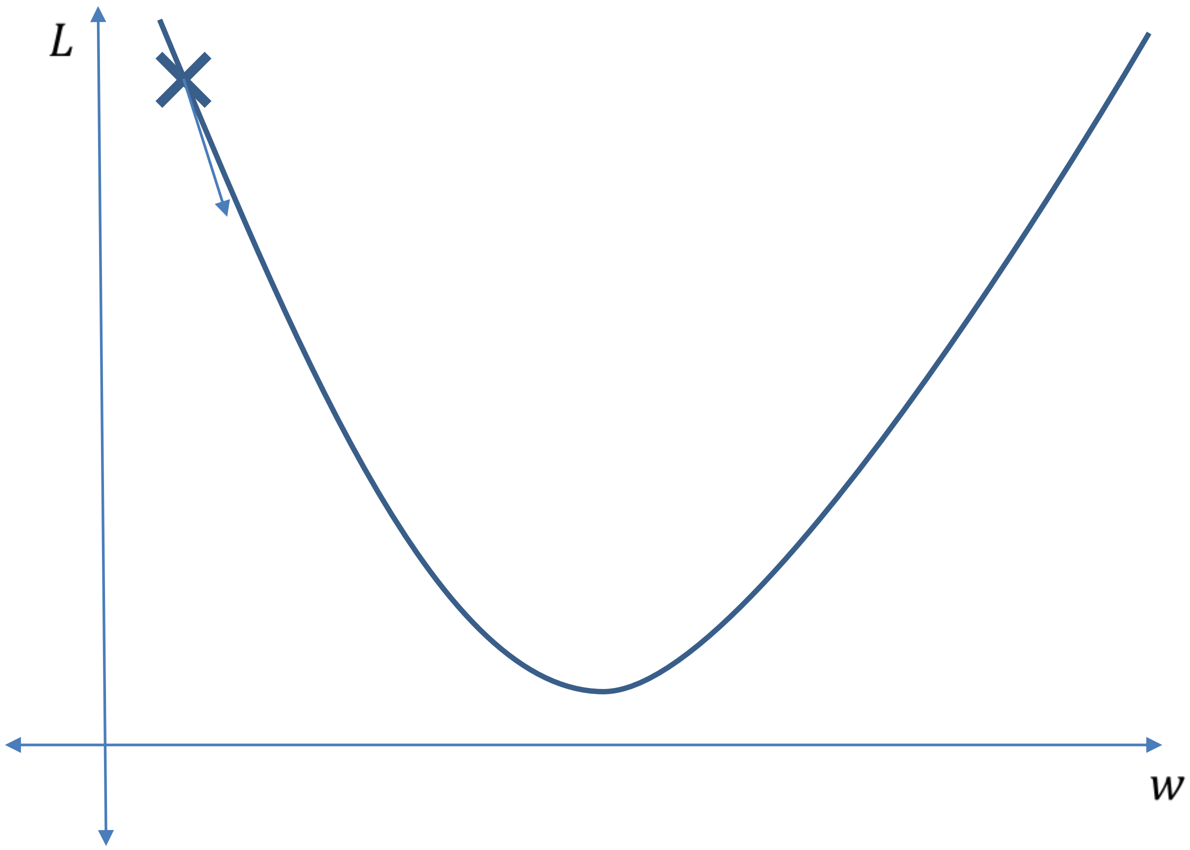
Gradient descent visualized: Minimizing loss



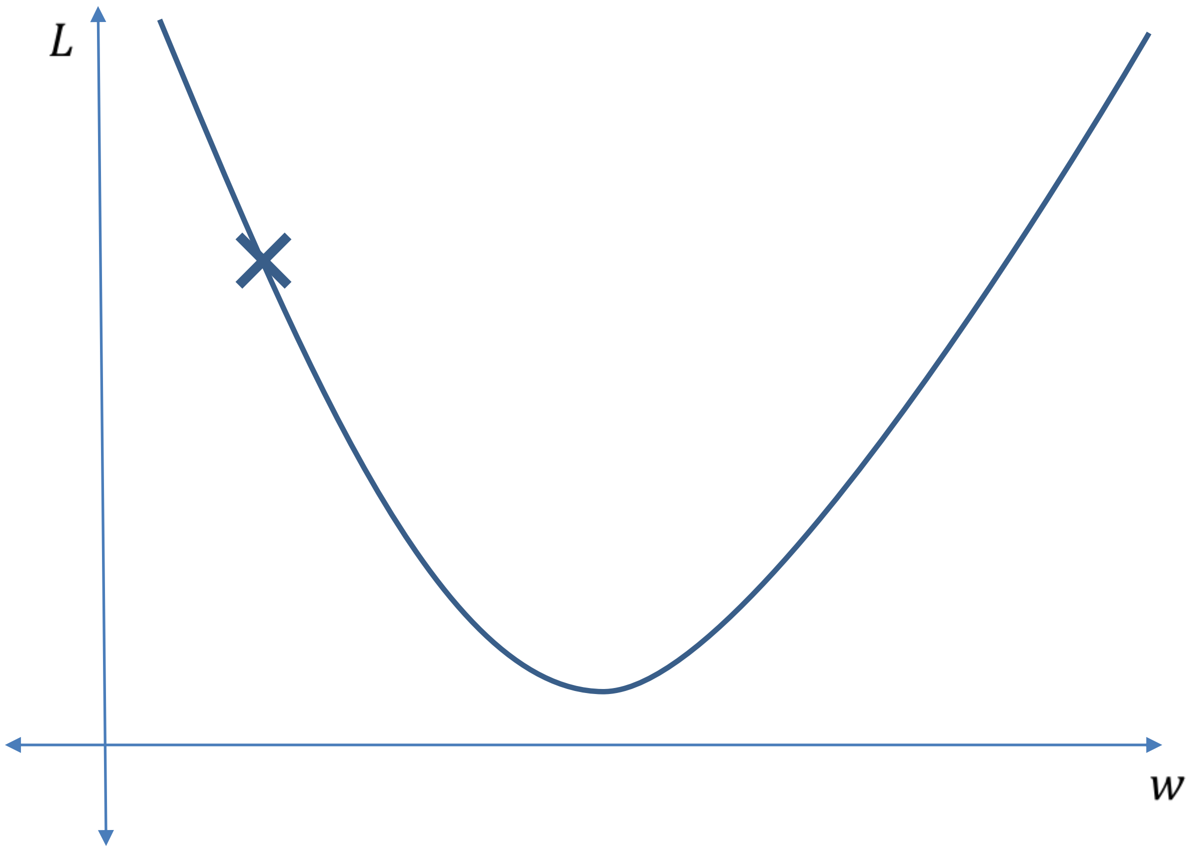
Gradient descent visualized: Minimizing loss



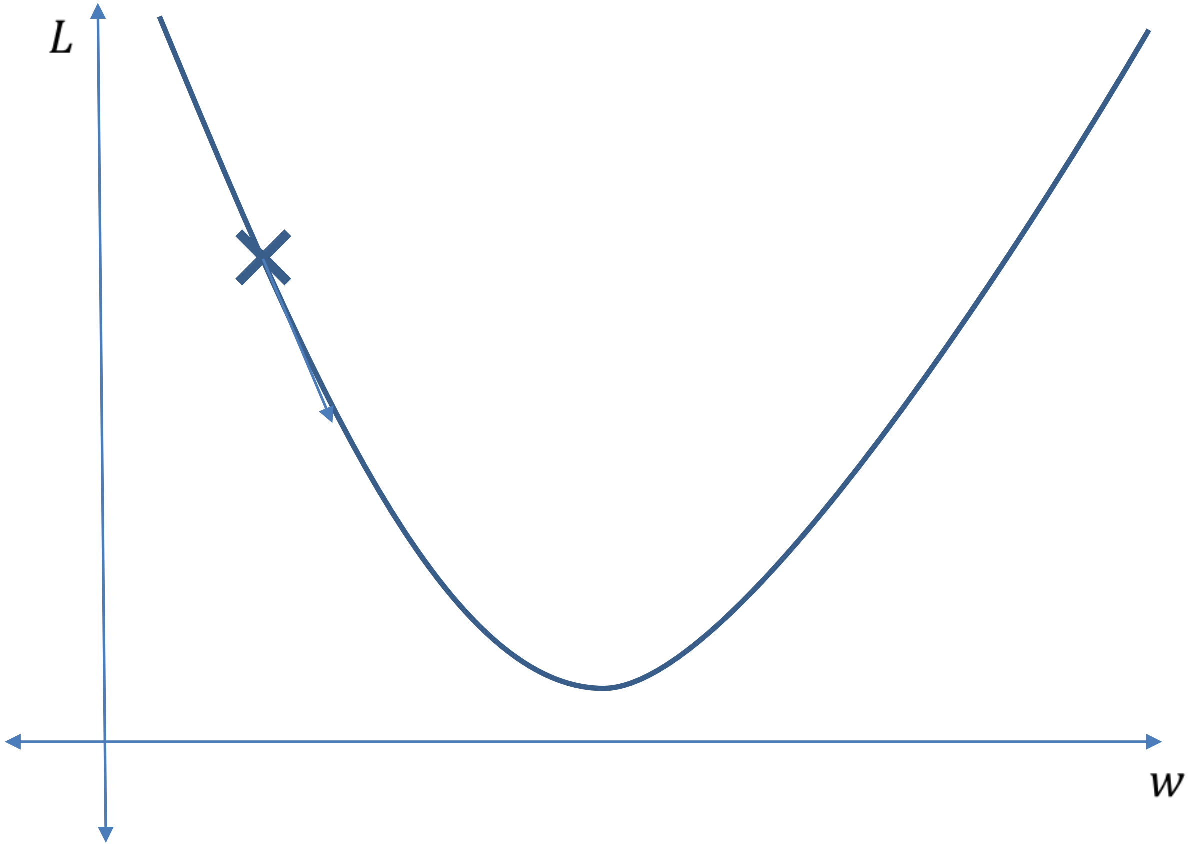
Gradient descent visualized: Minimizing loss



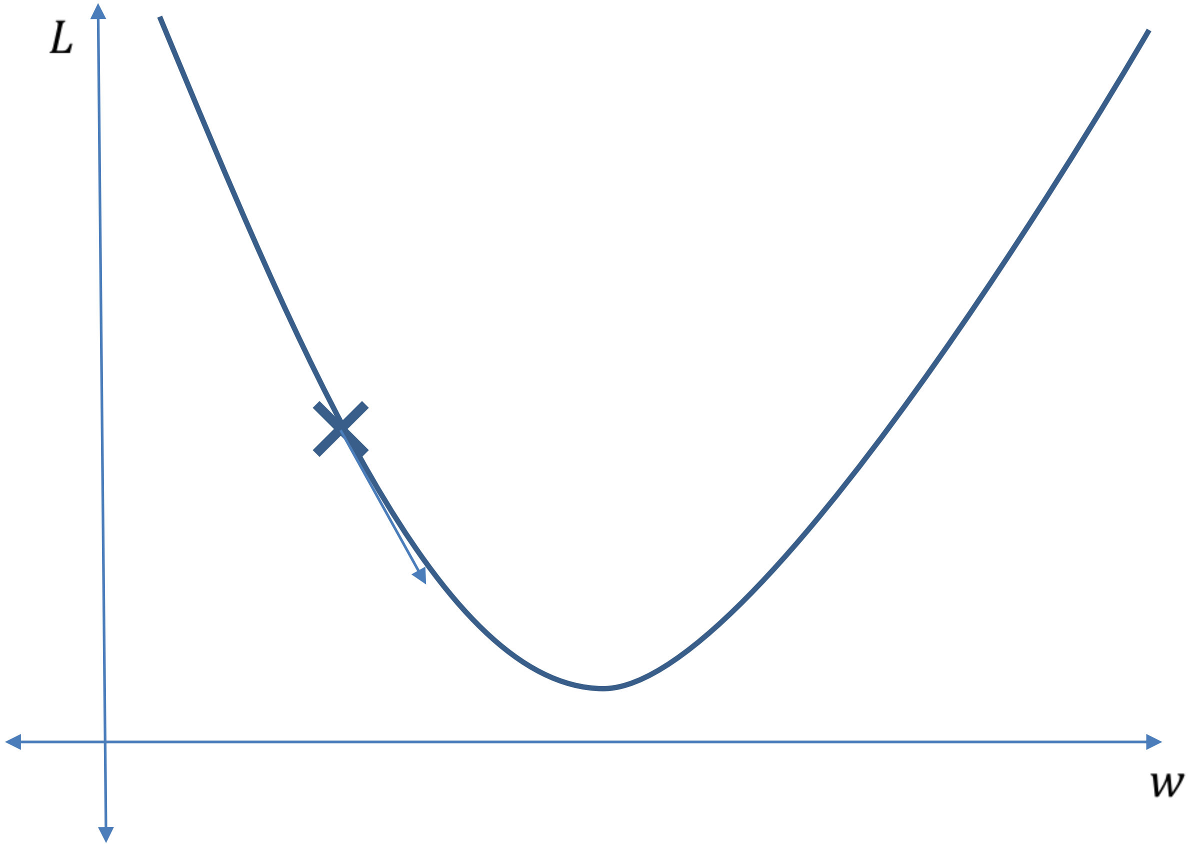
Gradient descent visualized: Minimizing loss



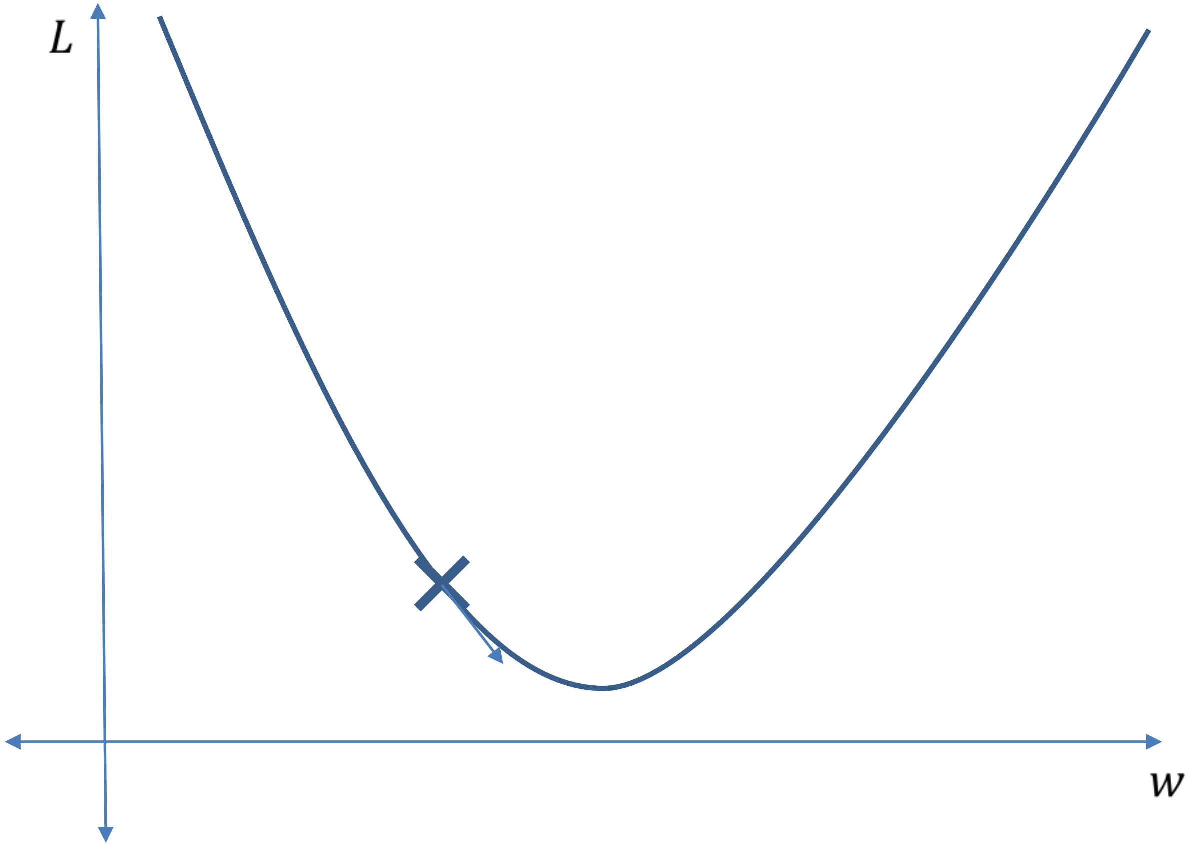
Gradient descent visualized: Minimizing loss



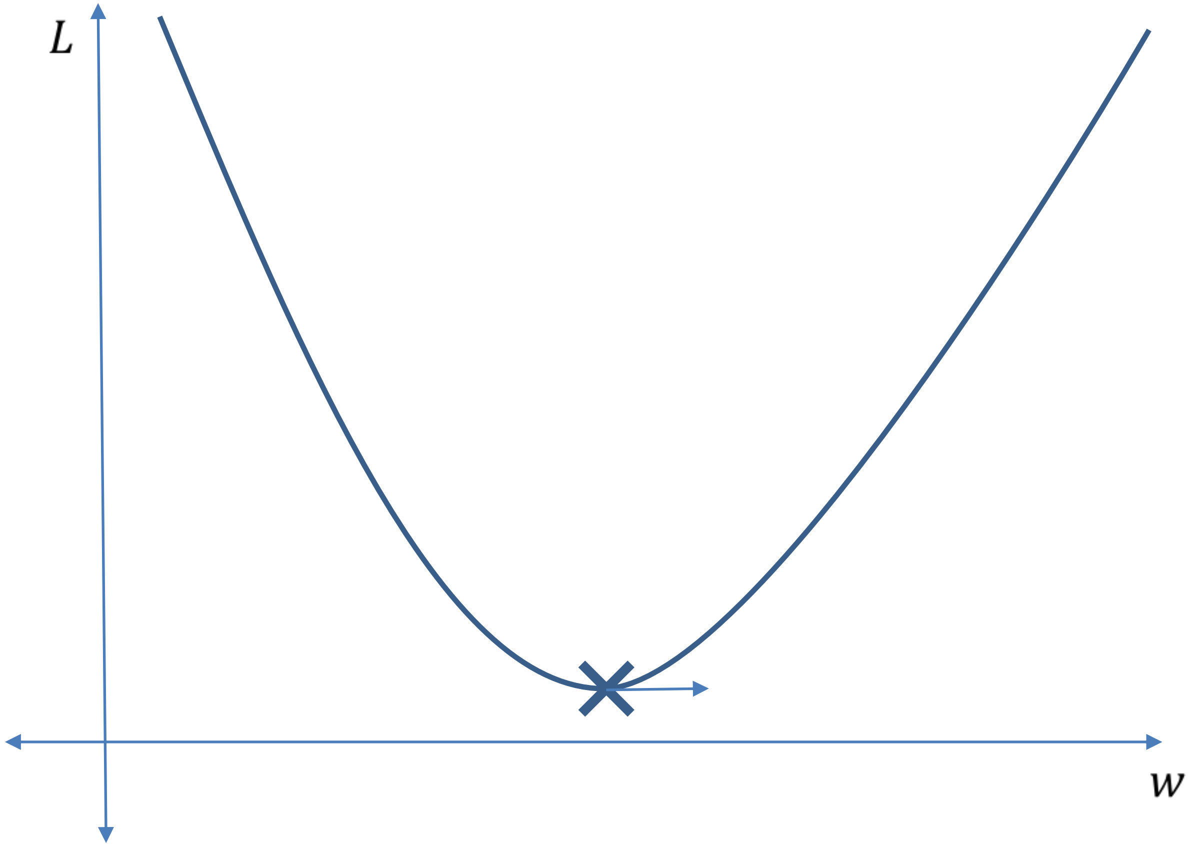
Gradient descent visualized: Minimizing loss



Gradient descent visualized: Minimizing loss



Gradient descent visualized: Minimizing loss



Gradient Descent Pseudocode

for _ in {0,…,num_epochs}:

$$L = 0$$

for x_i, y_i in data:

$$\hat{y}_i = f(x_i, W)$$

$$L \mathrel{+}= L_i(y_i, \hat{y}_i)$$

$$\frac{dL}{dW} = ???$$

$$W \coloneqq W - \alpha \frac{dL}{dW}$$

Partial derivative of loss to update weights

Given training data point (***x***, ***y***), the linear classifier formula is: $\hat{y} = Wx$

Let's assume that the correct label is class ***k***, implying ***y=k***

$$\text{Loss} = L(\hat{y}, y) = -\log \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$$

$$= -\hat{y}_k + \log \sum_j e^{\hat{y}_j}$$

Calculating the gradient is hard, but we can use the chain rule to make it simpler

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}$$

Partial derivative of loss to update weights

Given training data point (***x***, ***y***), the linear classifier formula is: $\hat{y} = Wx$

Let's assume that the correct label is class ***k***, implying ***y=k***

$$\text{Loss} = -\hat{y}_k + \log \sum_j e^{\hat{y}_j}$$

Now, we want to update the weights ***W*** by calculating the direction in which to change the weights to reduce the loss: $\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}$

We know that $\frac{d\hat{y}}{dW} = x$, but what about $\frac{dL}{d\hat{y}}$?

Partial derivative of loss to update weights

$$L = -\hat{y}_k + \log \sum_j e^{\hat{y}_j}$$

To calculate $\frac{dL}{d\hat{y}}$, we need to consider two cases:

Case 1: $$\frac{dL}{d\hat{y}_k} = -1 + \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$$

Case 2: $$\frac{dL}{d\hat{y}_{l \neq k}} = \frac{e^{\hat{y}_l}}{\sum_j e^{\hat{y}_j}}$$

Partial derivative of loss to update weights

Putting it all together:

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}$$

$$\frac{dL}{dW} =. \begin{bmatrix} \frac{e^{\hat{y}_0}}{\sum_j e^{\hat{y}_j}} \\ \dots \\ \dots \\ -1 + \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} \\ \dots \\ \dots \\ \frac{e^{\hat{y}_{3071}}}{\sum_j e^{\hat{y}_j}} \end{bmatrix} \quad x$$

Gradient Descent Pseudocode

```
for _ in {0,…,num_epochs}:
```

$\overline{L} = 0$

for x_i, y_i in data:

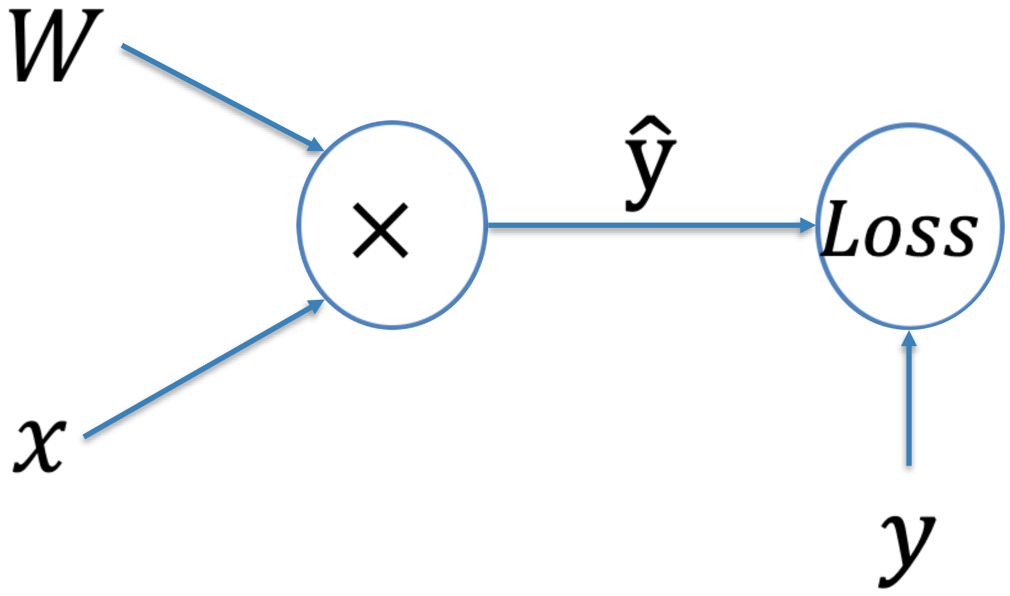
$\hat{y}_i = f(x_i, W)$

$L \mathrel{+}= L_i(y_i, \hat{y}_i)$

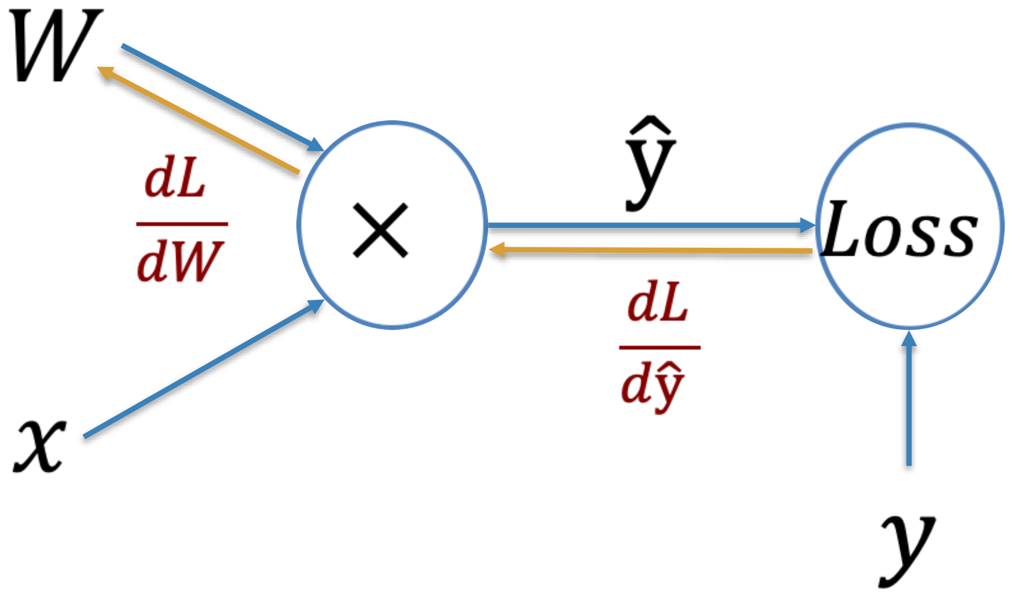
$\frac{dL}{dW} = \textit{We know how to calculate this now!}$

$W := W - \alpha \frac{dL}{dW}$

Backprop – another way of computing gradients



Backprop – another way of computing gradients



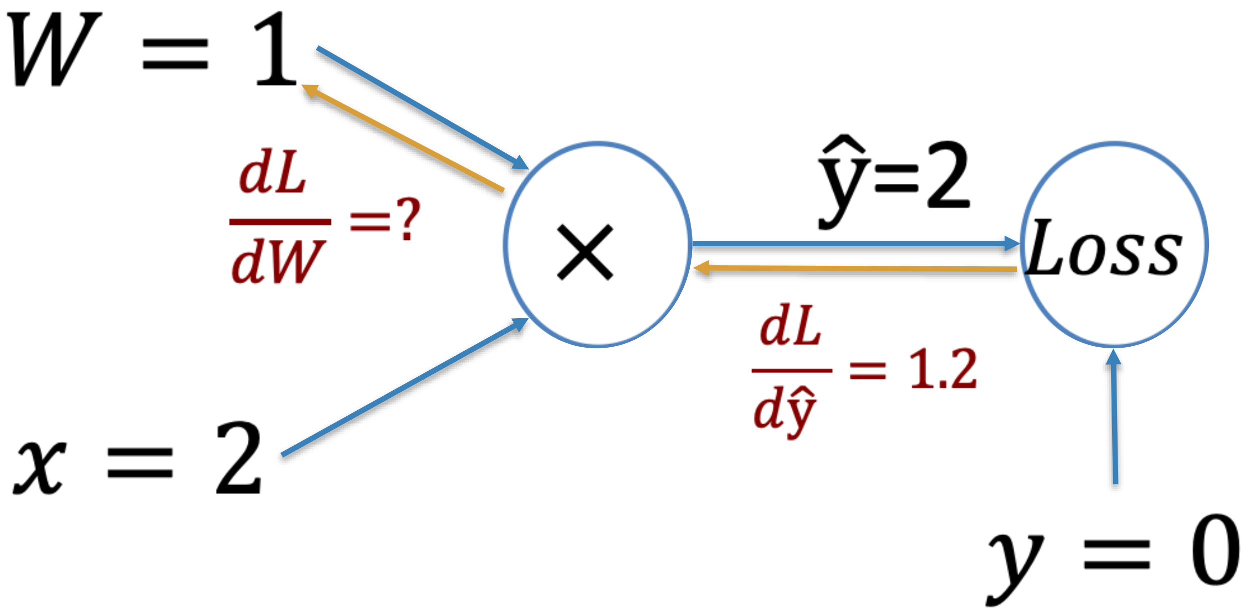
$$\hat{y} = Wx$$
$$L = Loss(\hat{y}, y)$$

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}$$

Key Insight:

- visualize the computation as a graph
- Compute the forward pass to calculate the loss.
- Compute all gradients for each computation backwards

Backprop example in 1D:



We know the chain rule

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW}$$
$$= \frac{dL}{d\hat{y}}x$$
$$= 1.2x$$
$$= 1.2 \times 2$$
$$= 2.4$$

●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

Let's break it into steps.

- Initialize

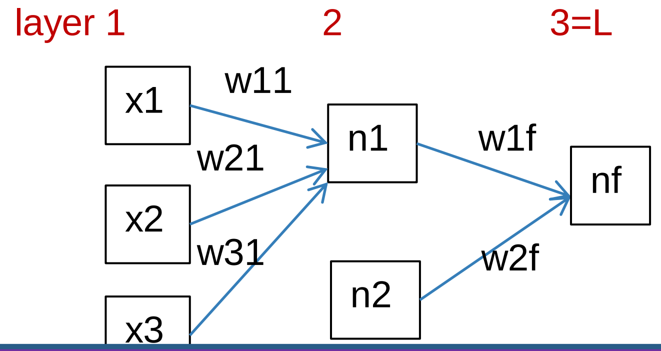
The backpropagation algorithm

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

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

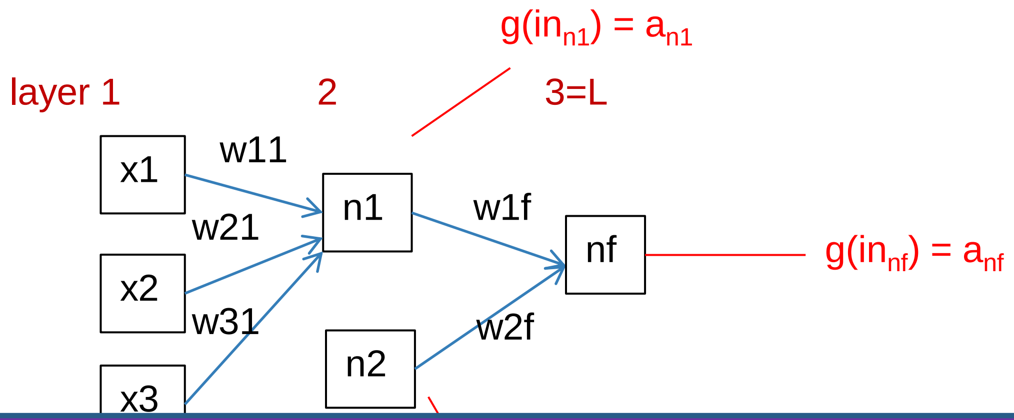
inputs:
examples, a set of examples, each with input vector **x** and output vector **y**.
network, a multilayer network with L layers, weights $W_{j,i}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

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**
        $in_j \leftarrow \sum_i w_{i,j}\, a_i$
        $a_j \leftarrow g(in_j)$

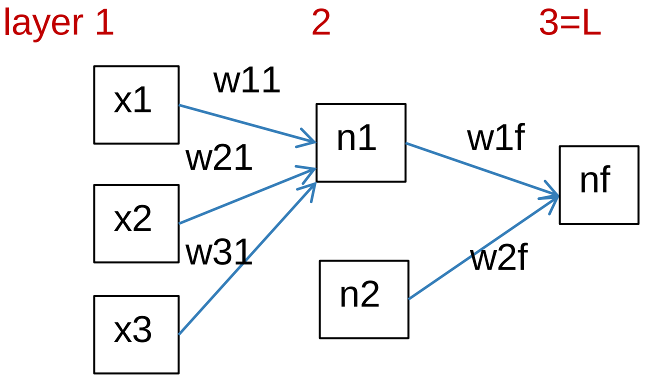


●Backward Propagation 1

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.

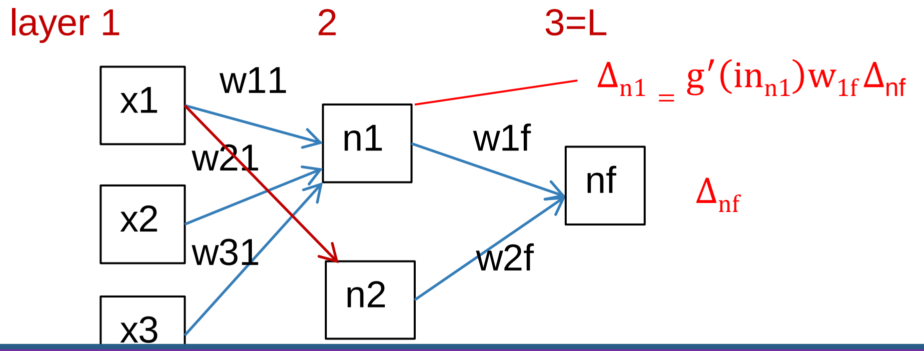


$\Delta_{nf} = g'(in_{nf}) * (y_{nf} - a_{nf})$

●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

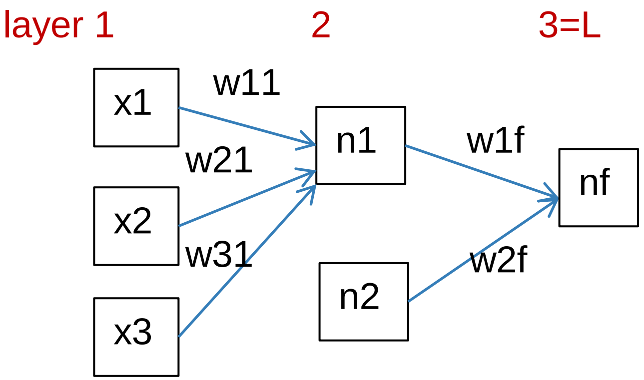


If there were two output nodes, there would be a summation.

●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.

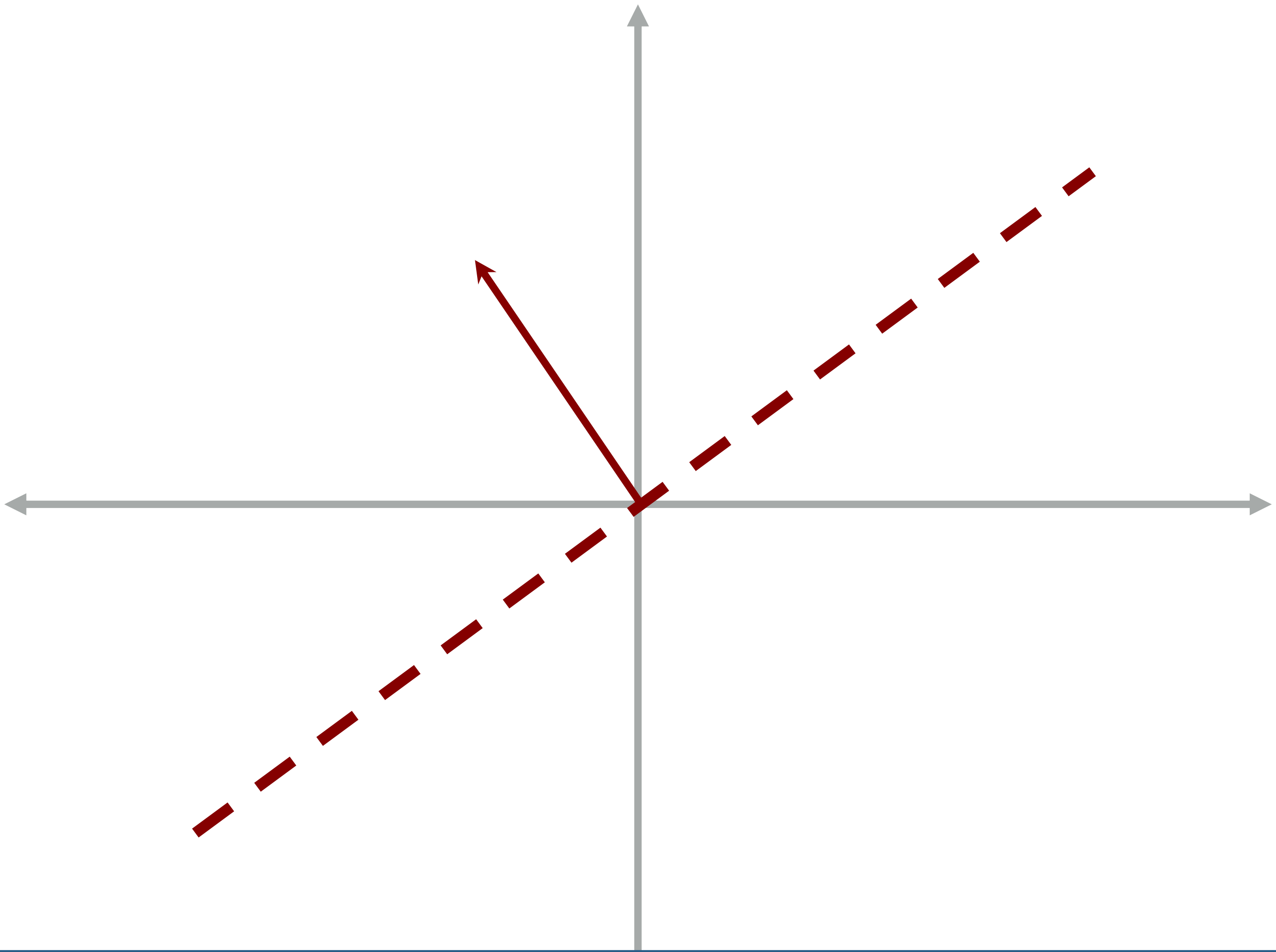


Back Propagation Summary

- 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

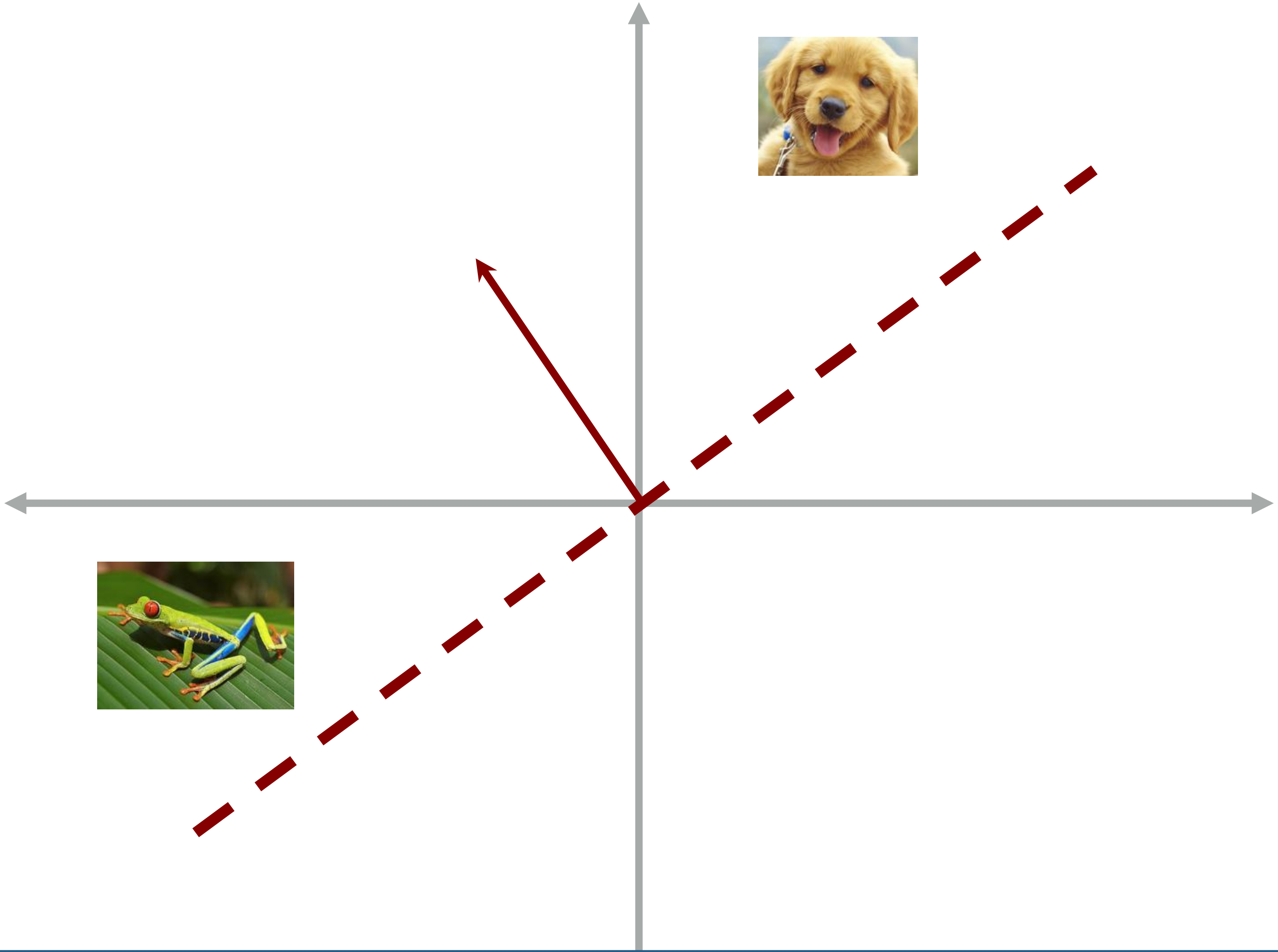
- This is done for all examples and multiple epochs, till convergence or enough iterations.

Interpreting the weights geometrically



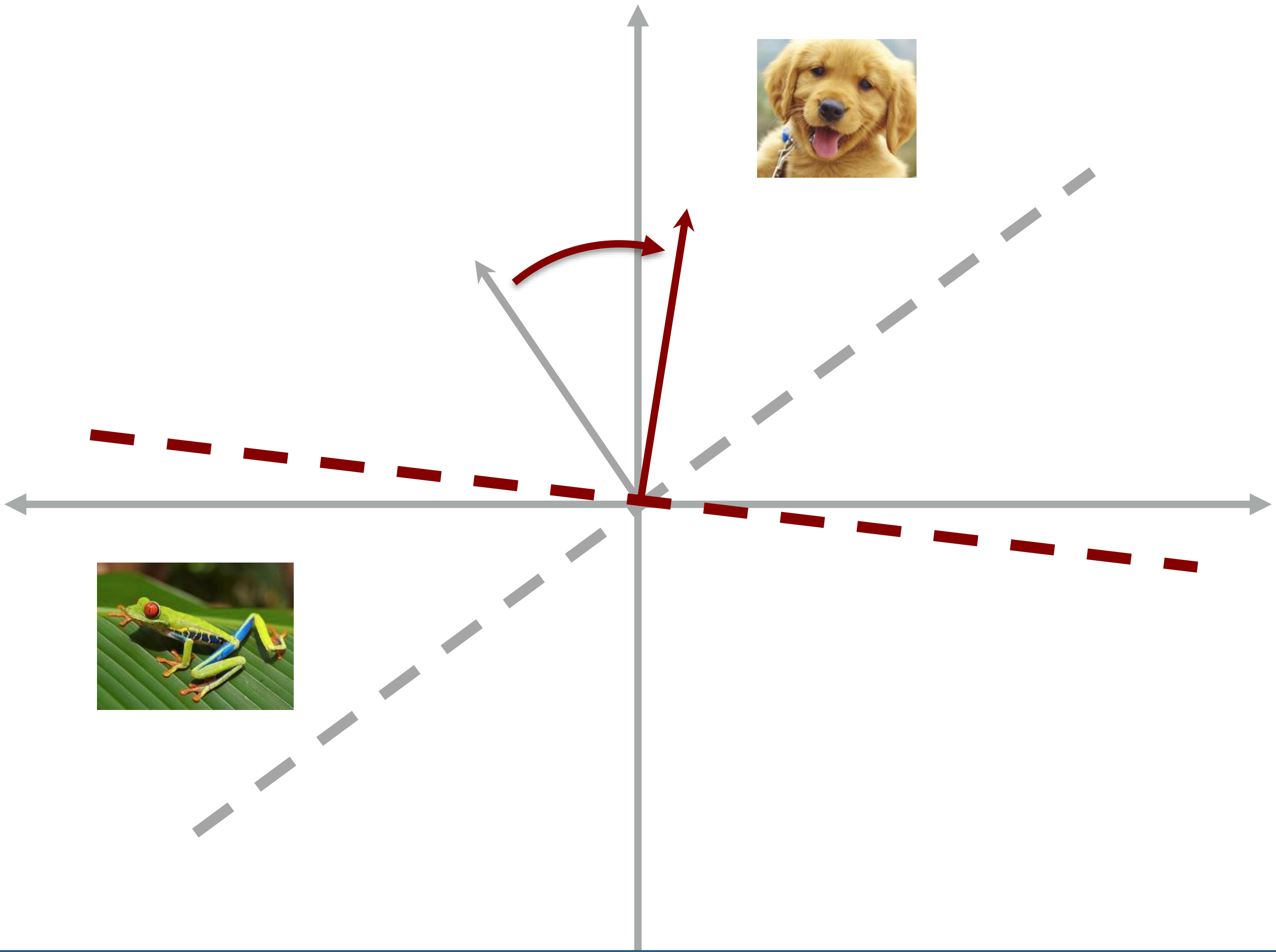
- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly

Interpreting the weights geometrically



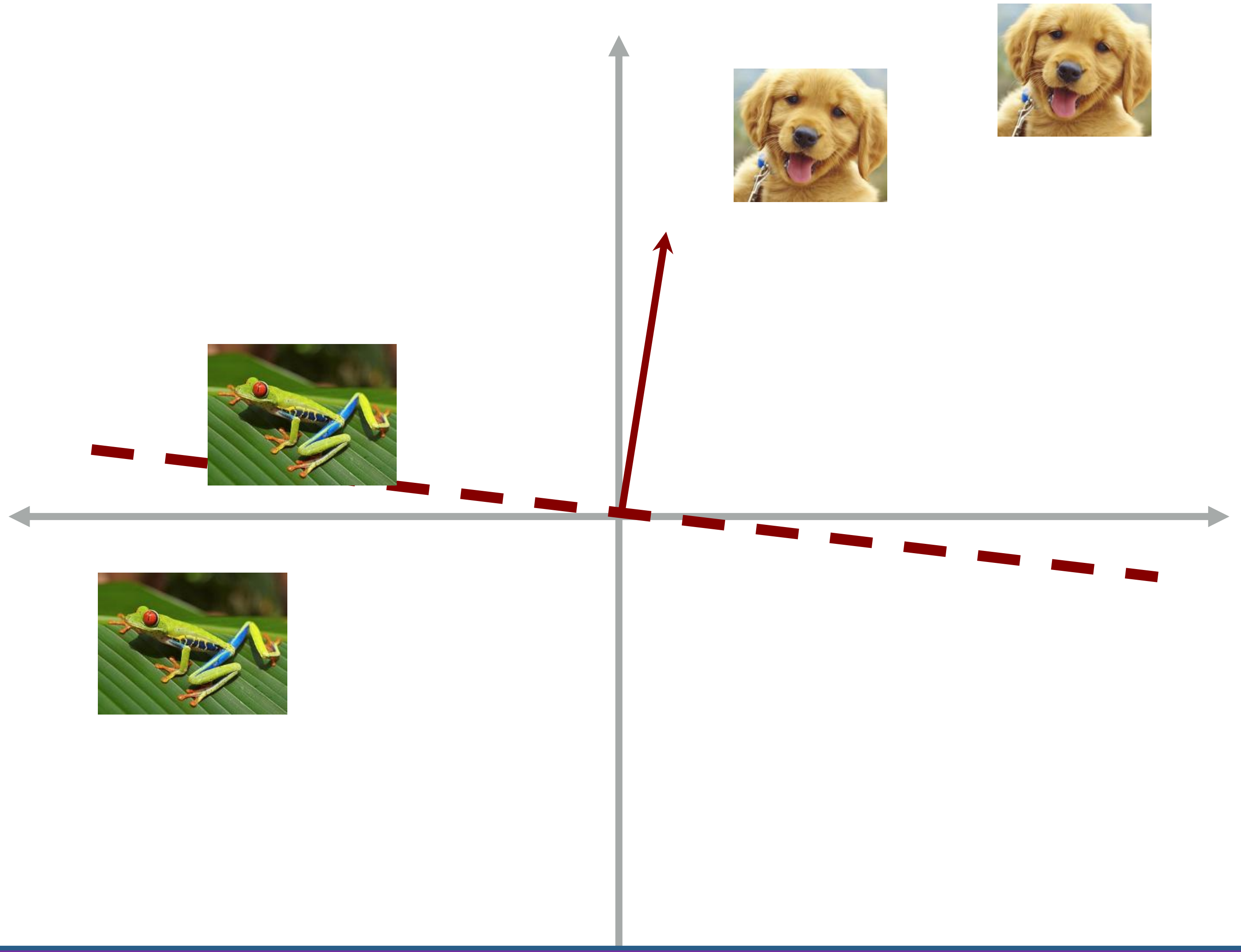
- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two data points

Interpreting the weights geometrically



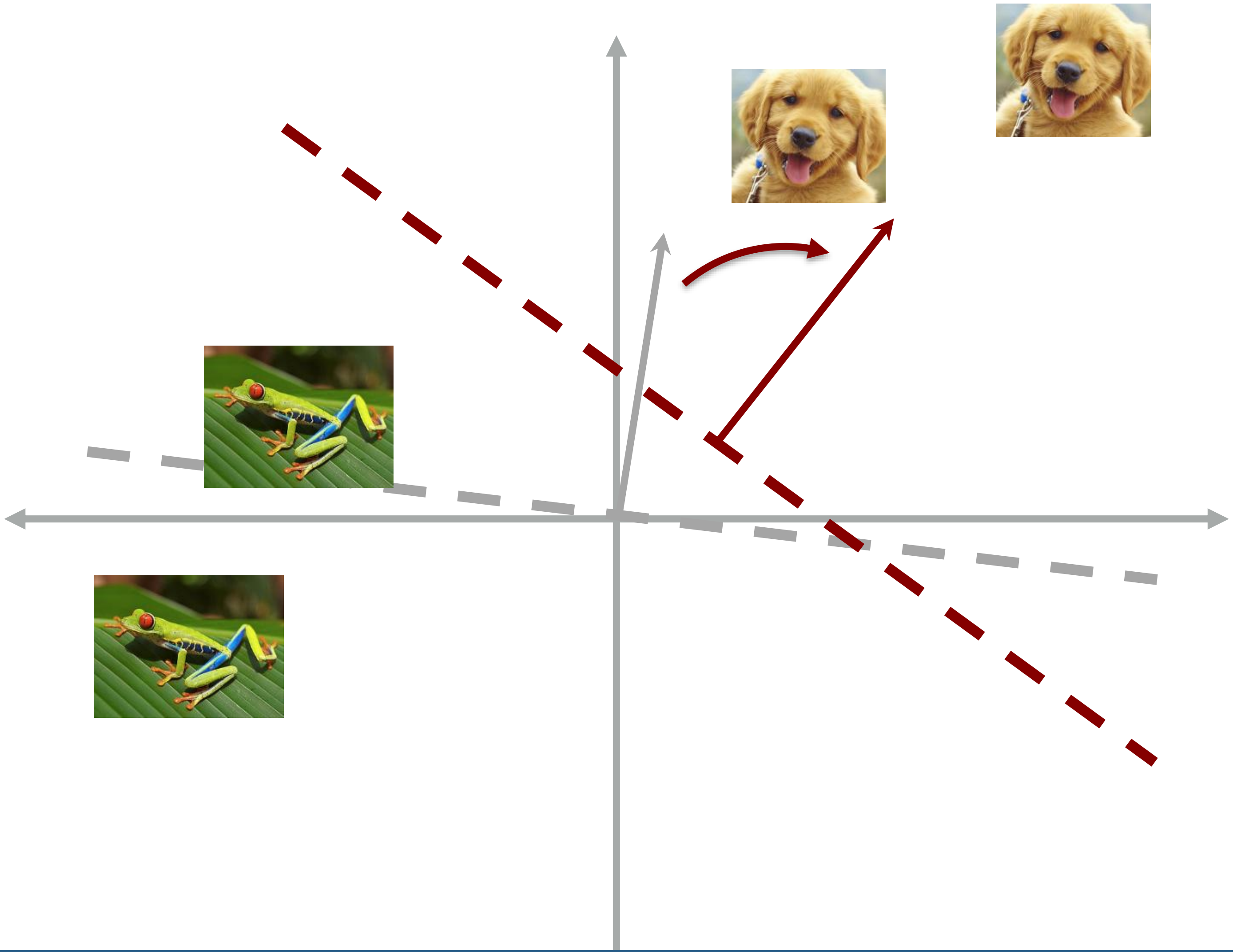
- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two data points
- Update the weights

Interpreting the weights geometrically



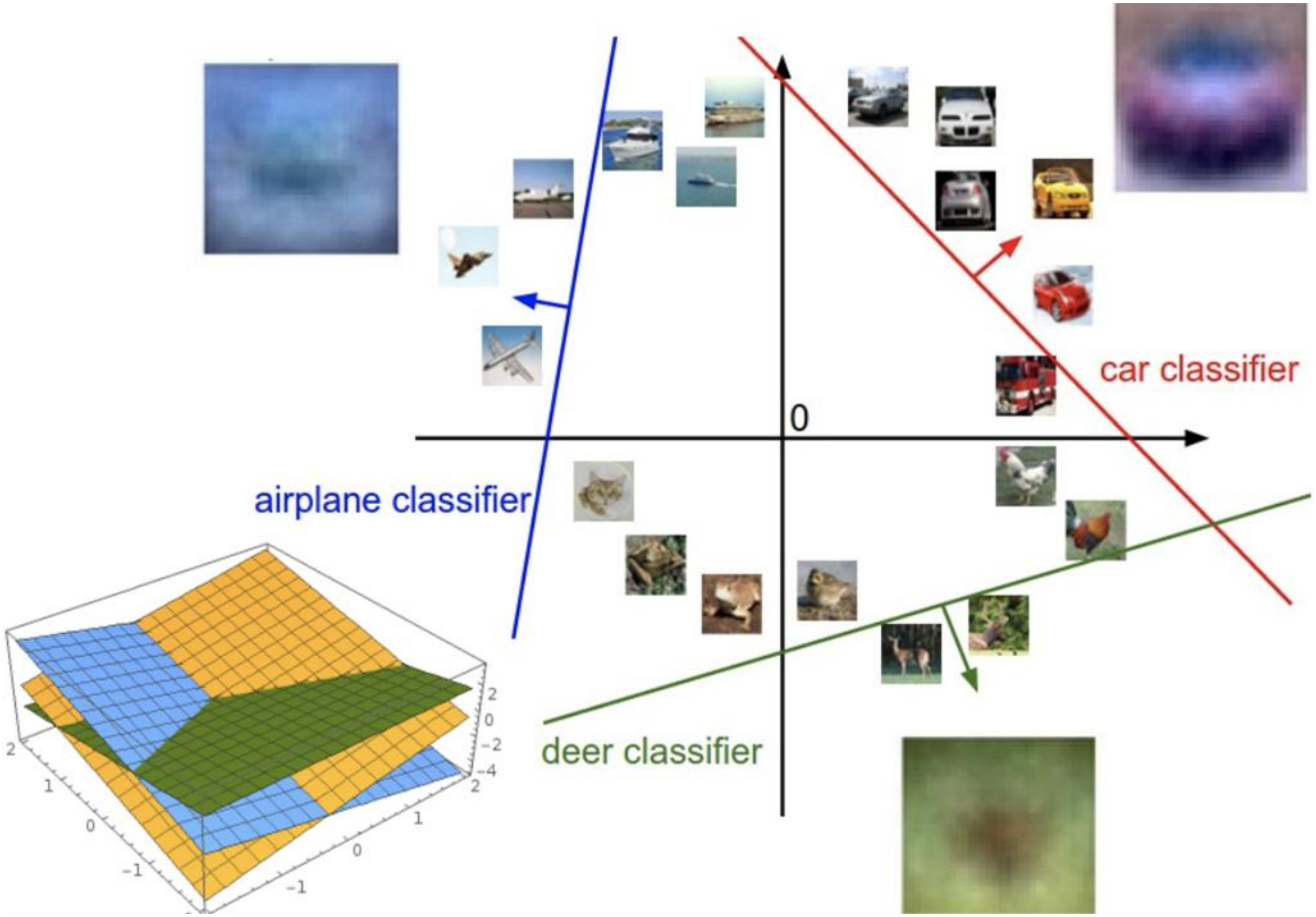
- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two more data points
- Update the weights

Interpreting the weights geometrically



- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two more data points
- Update the weights

Interpreting the weights geometrically

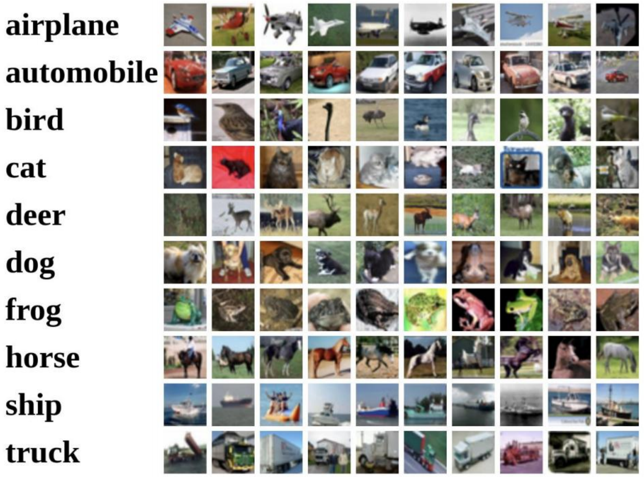


Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks

A simple recognition pipeline

Training Images



Training Labels

Image Features

Training

Learned Classifier

Test

I

Image Features

Learned Classifier

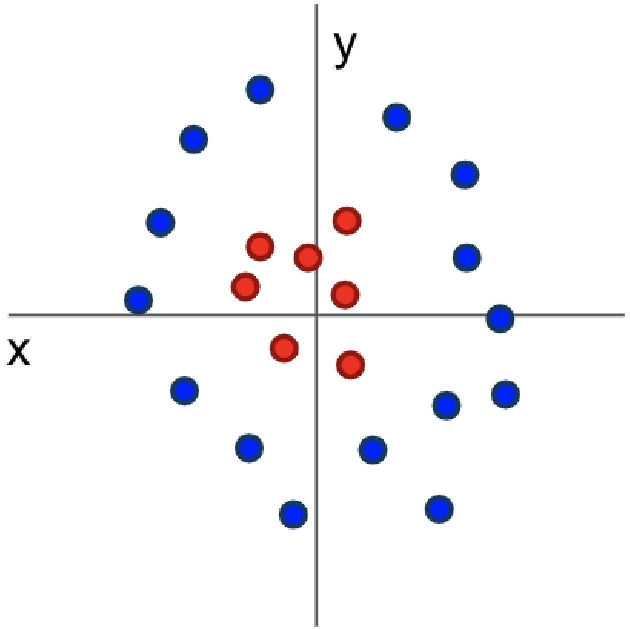
Prediction

Recall: we can featurize images into a vector

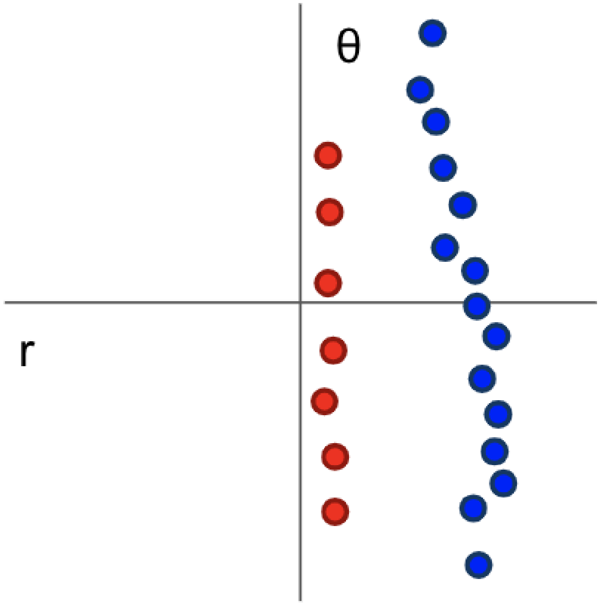
Raw pixels
Raw pixels + (x,y)
PCA
LDA
BoW
BoW + spatial pyramids

Image Vector

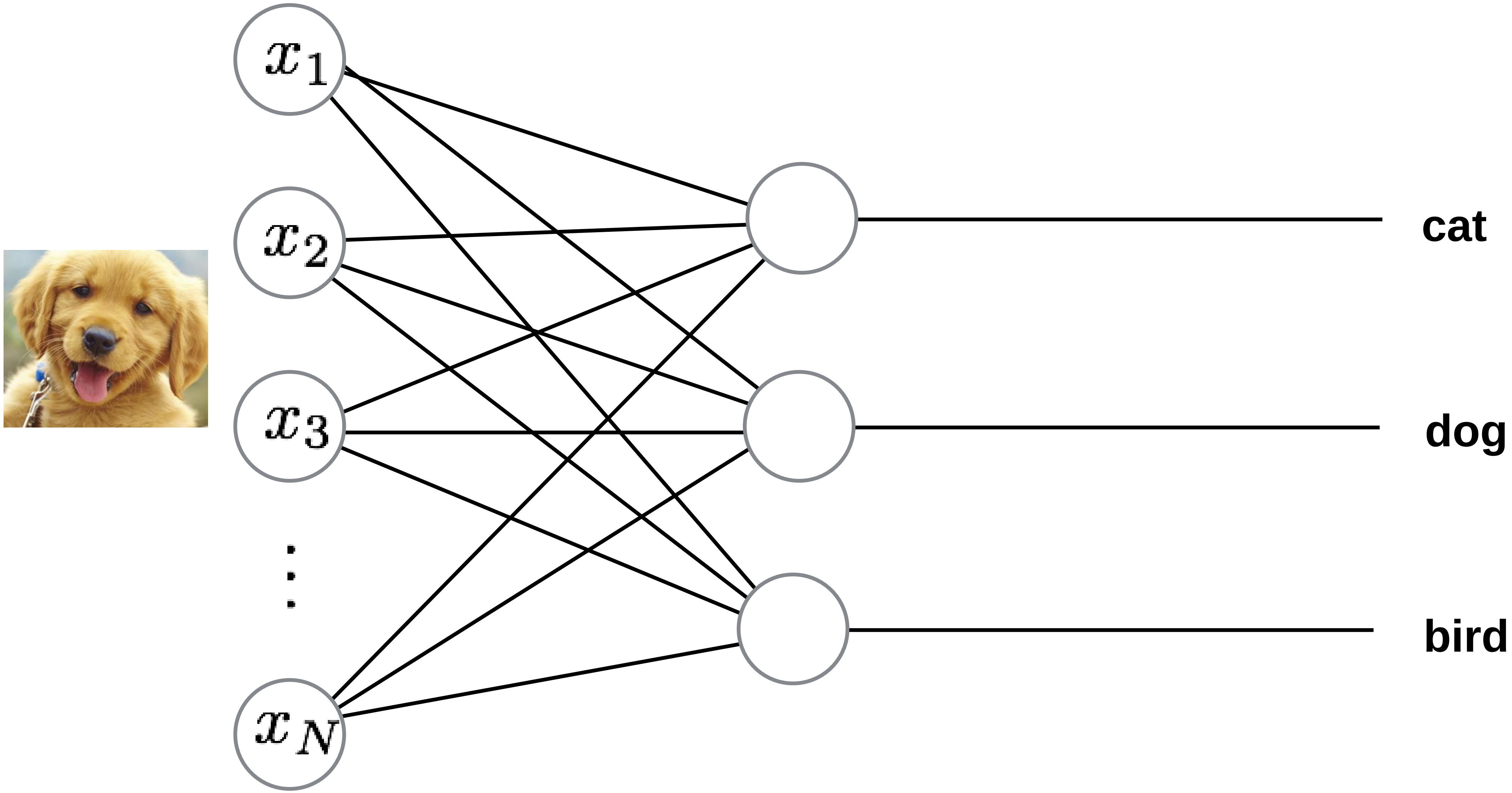
Features sometimes might not be linearly separable



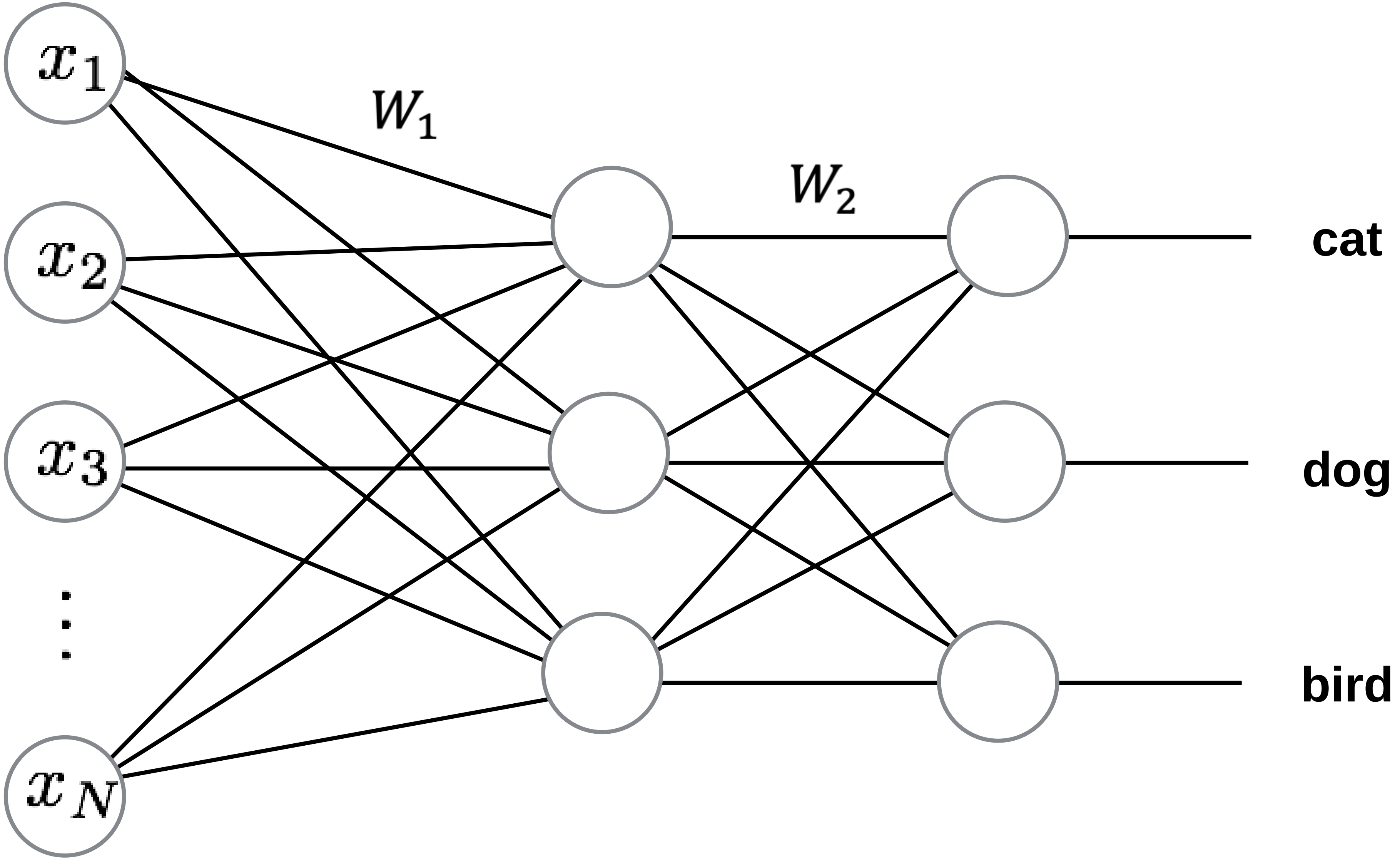
$$f(x,y) = (r(x,y), \theta(x,y))$$



Remember our linear classifier



Let's change the features by adding another layer



2-layer network: mathematical formula

- Linear classifier: $y = Wx$
- 2-layer network: $y = W_2 \max(0, W_1 x)$
- 3-layer network: $y = W_3 \max(0, W_2 \max(0, W_1 x))$

The number of layers is a new hyperparameter!

2-layer network: mathematical formula

- Linear classifier: $y = Wx$
- 2-layer network: $y = W_2 \max(0, W_1 x)$

We know the size of *x* = 1 x 3072 and y = 10 x 1, so what are ***W1*** and ***W2***

$$W_1 = h \times 3072 \qquad W_2 = 10 \times h$$

h is a new hyperparameter!

2-layer network: mathematical formula

- Linear classifier: $y = Wx$
- 2-layer network: $y = W_2 \max(0, W_1 x)$

Why is the max(0, _) necessary? Let's see what happen when we remove it:

$$y = W_2 W_1 x = \mathrm{W}x$$

Where: $\mathrm{W} = W_2 W_1$

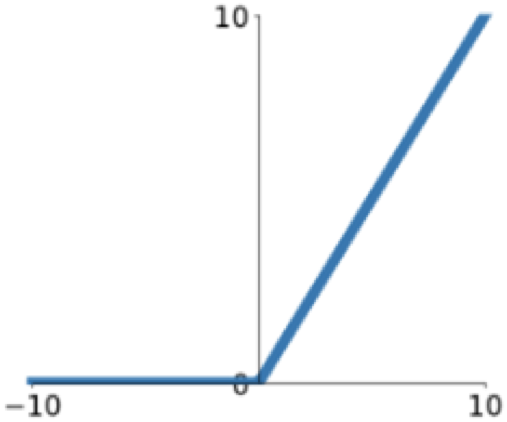
Activation function

The non-linear max function allows models to learn more complex transformations for features.

Choosing the right activation function is another new hyperparameter!

ReLU

$$\max(0, x)$$



2-layer neural network performance

- ~40% accuracy on CIFAR-10 test
  - Best class: Truck (~60%)
  - Worst class: Horse (~16%)

- Check out the model at: **https://tinyurl.com/cifar10**

Next lecture

Deep Learning