Deep Learning

First classifier: **Nearest Neighbor**

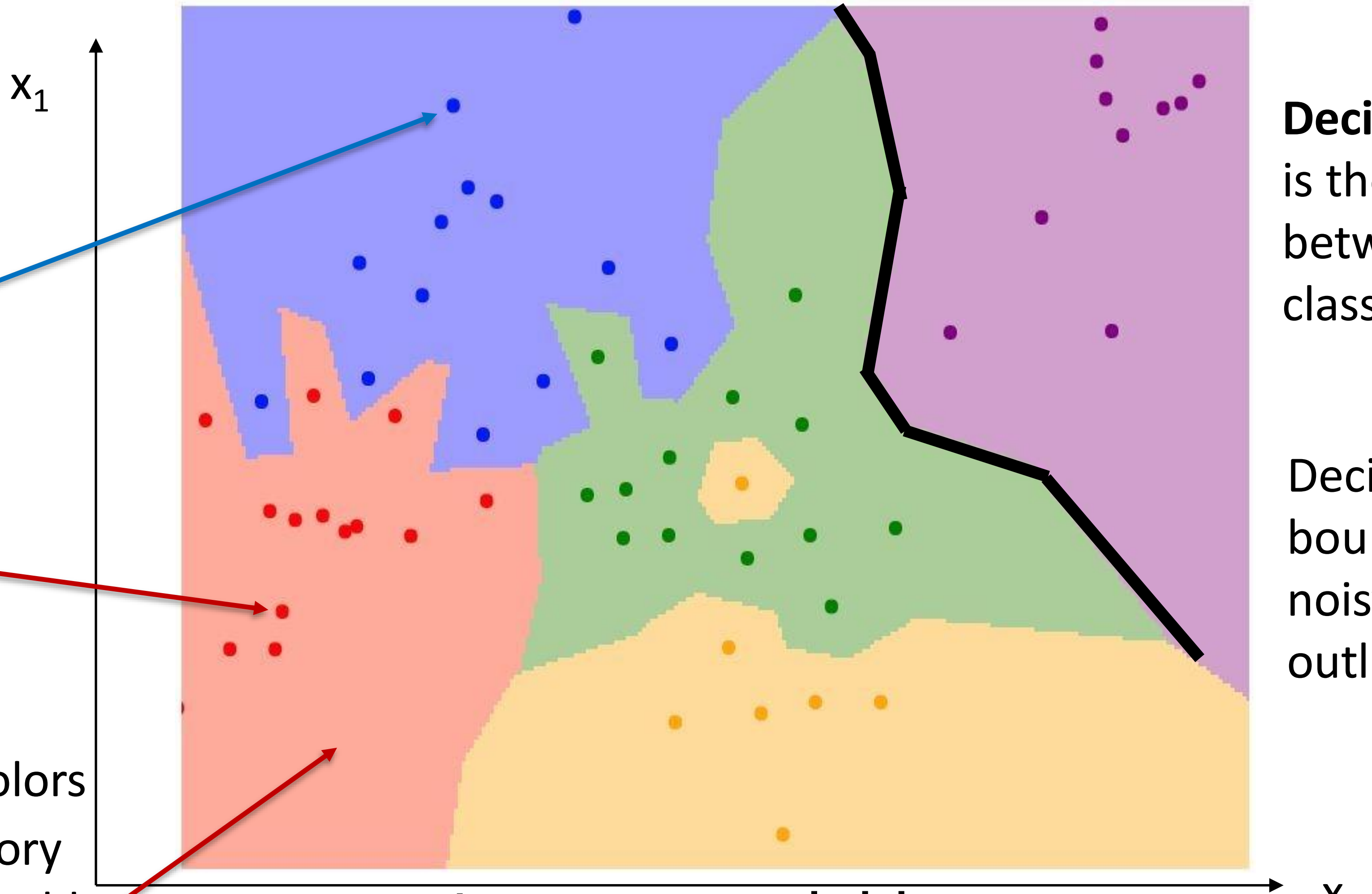
```
def train(images, labels):
  # Machine learning!
  return model
```

→ Memorize all data and labels

```
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

→ Predict the label of the most similar training image

Example



1-nearest neighbor

Points are training examples; colors give training labels

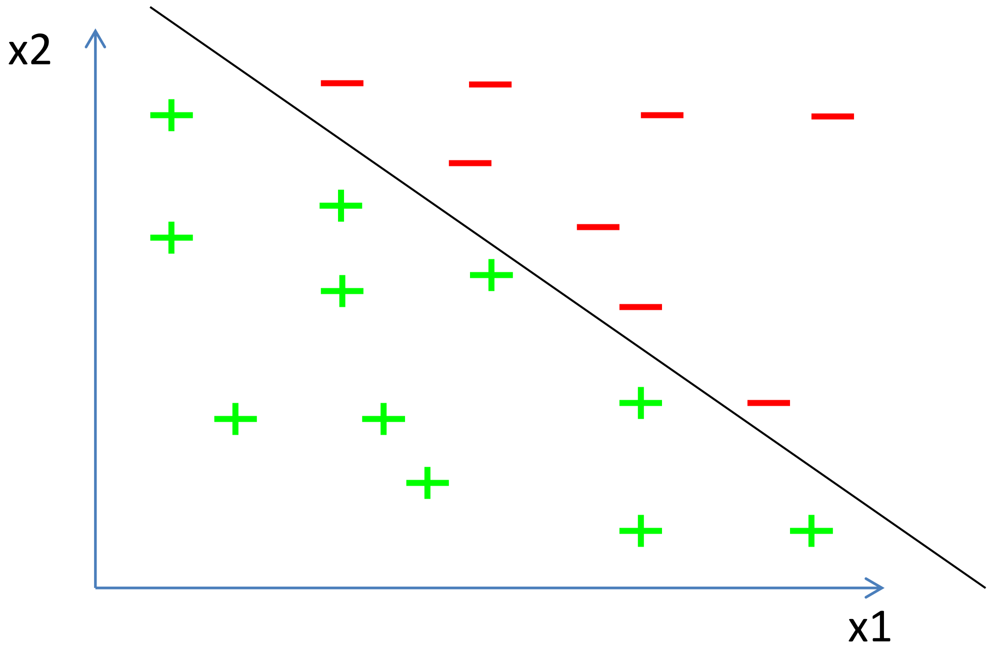
Background colors give the category a test point would be assigned

Decision boundary is the boundary between two classification regions

Decision boundaries can be noisy; affected by outliers

Linear classifiers : Motivation

- kNN produce decision boundaries by calculating them during prediction.
- Can we define a (simple) function during training to define decision boundaries directly?



Parametric Approach: Linear Classifier

$$f(x,W) = Wx$$

Image

Array of **32x32x3** numbers
(3072 numbers total)

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

W

parameters
or weights

10 numbers defining
class scores

Parametric Approach: Linear Classifier

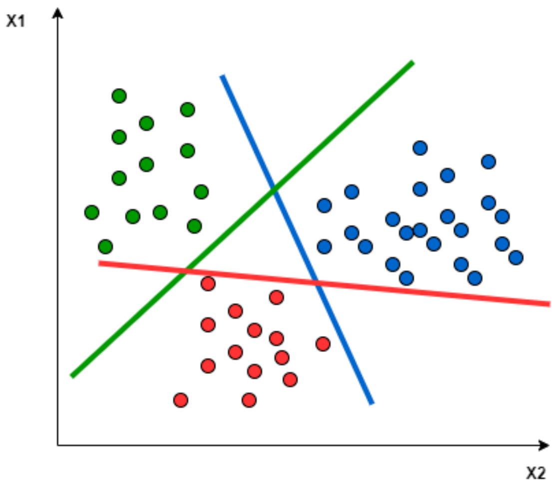
$\mathbf{w}_1 \cdot \mathbf{x} = w_{1,1} * x_1 + \ldots + w_{1,3072} * x_{3072}$

Shape: (10,3072)

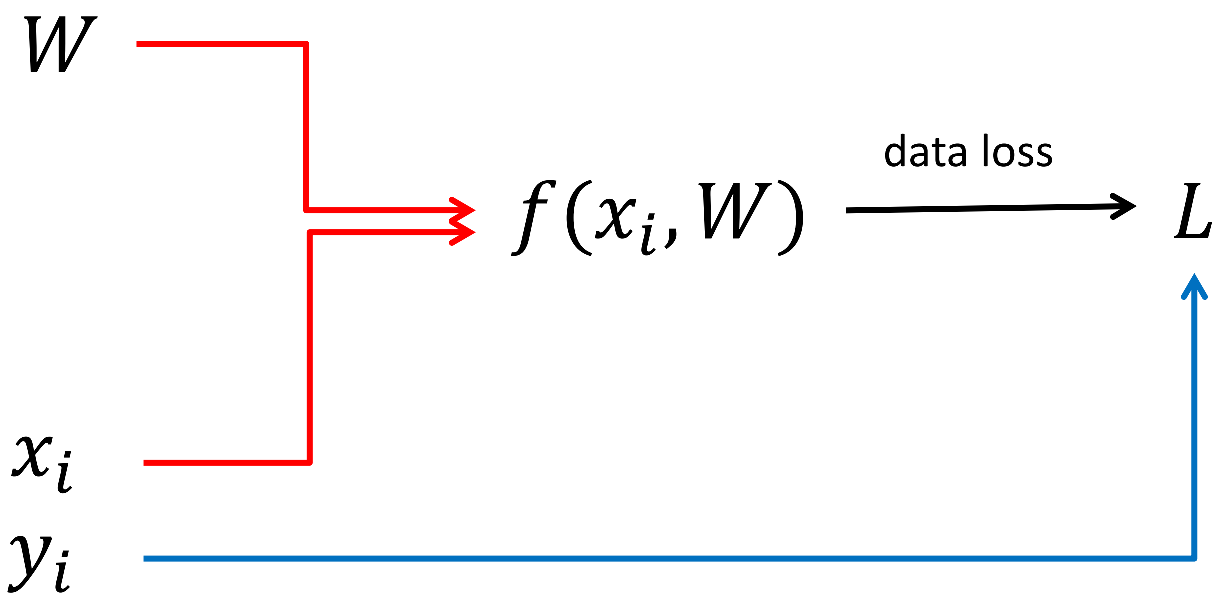
$$f(x,W) = Wx+b$$

Shape: (10,1)

Shape: (3072,1)

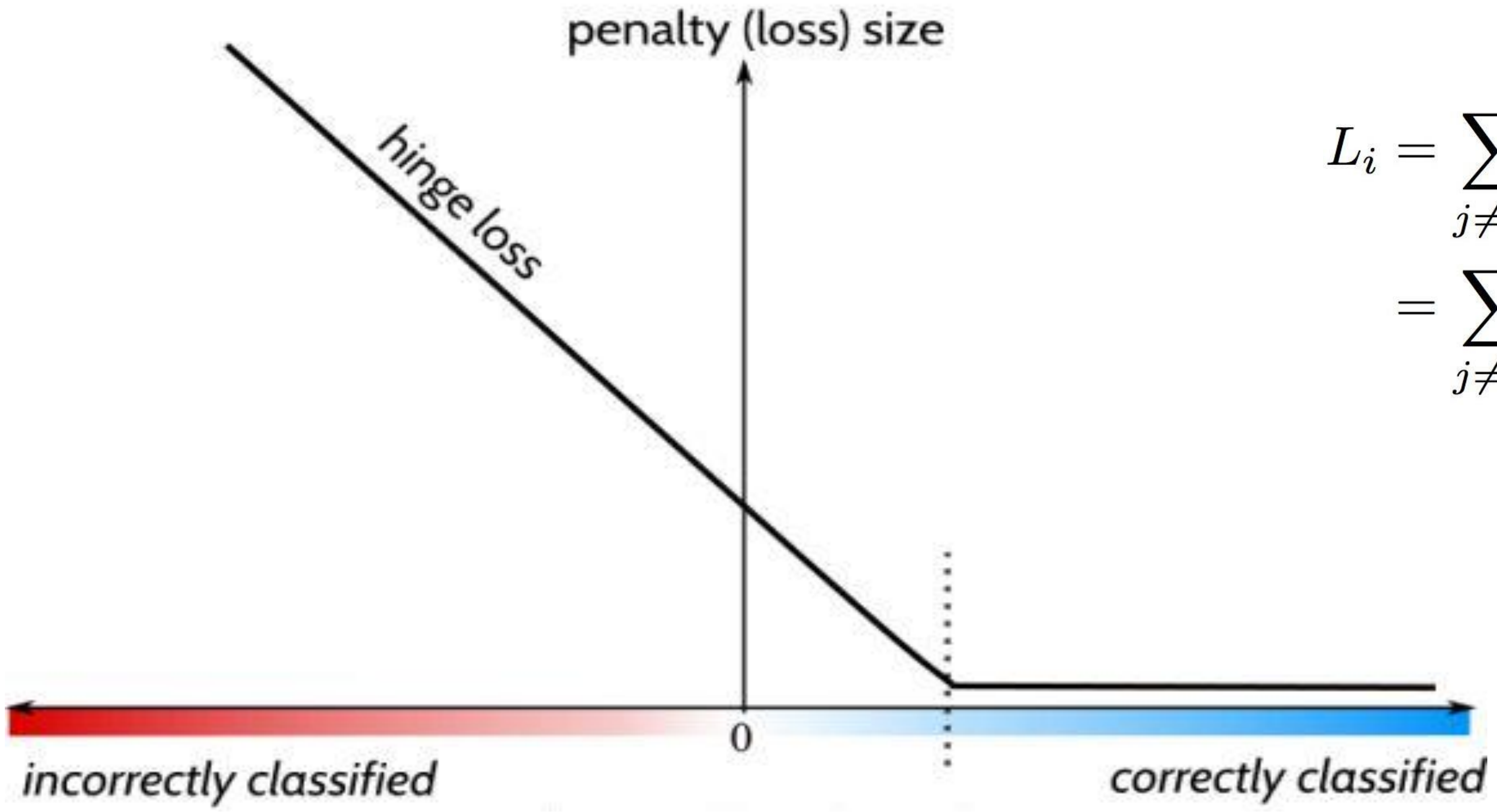


Loss Function



L: Metric to assess what loss of data classification our model incurs

Hinge loss

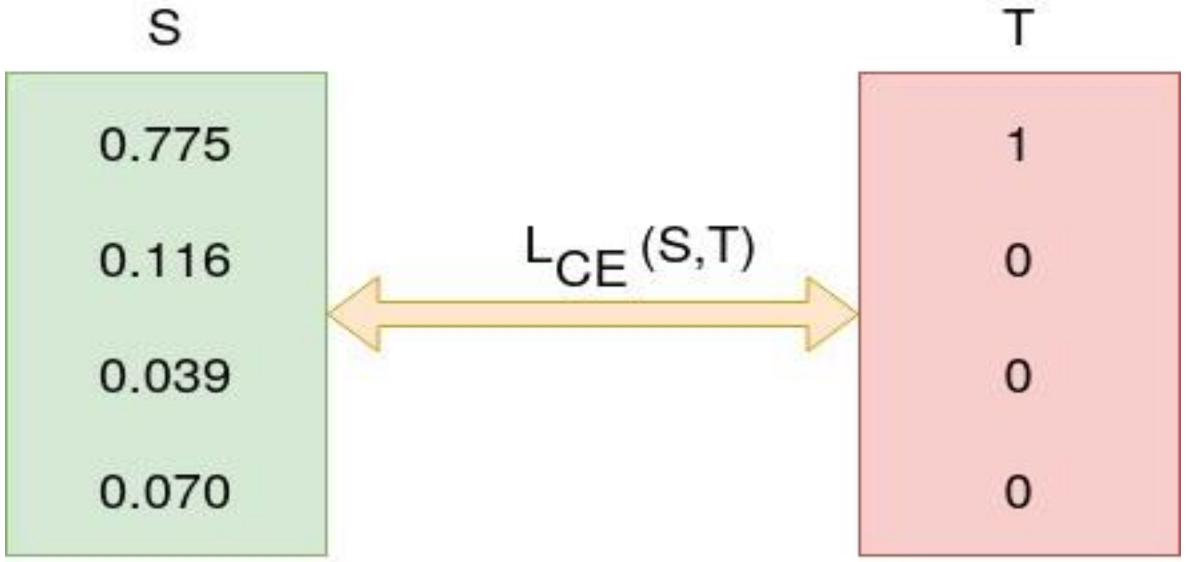


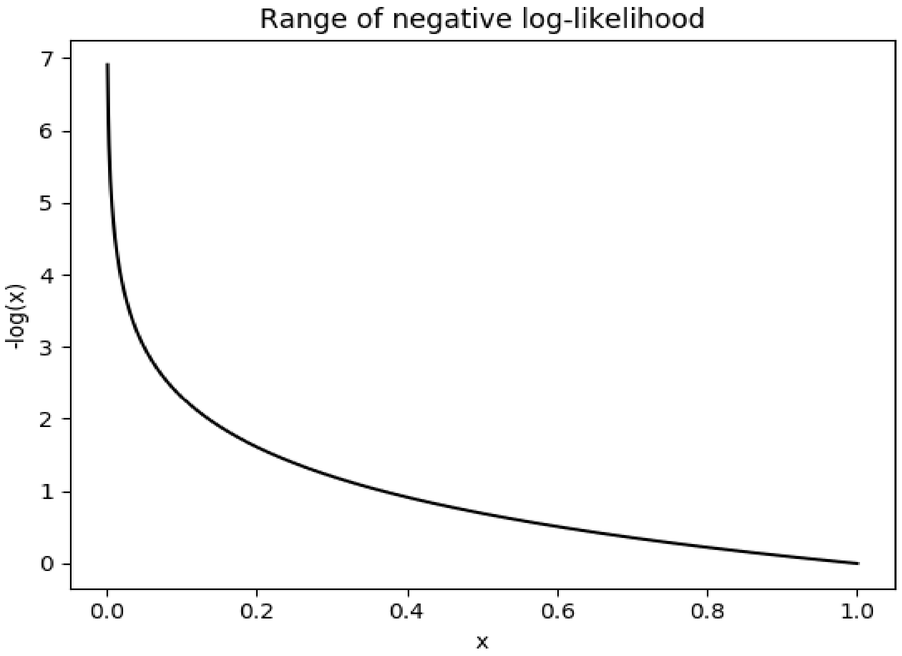
$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

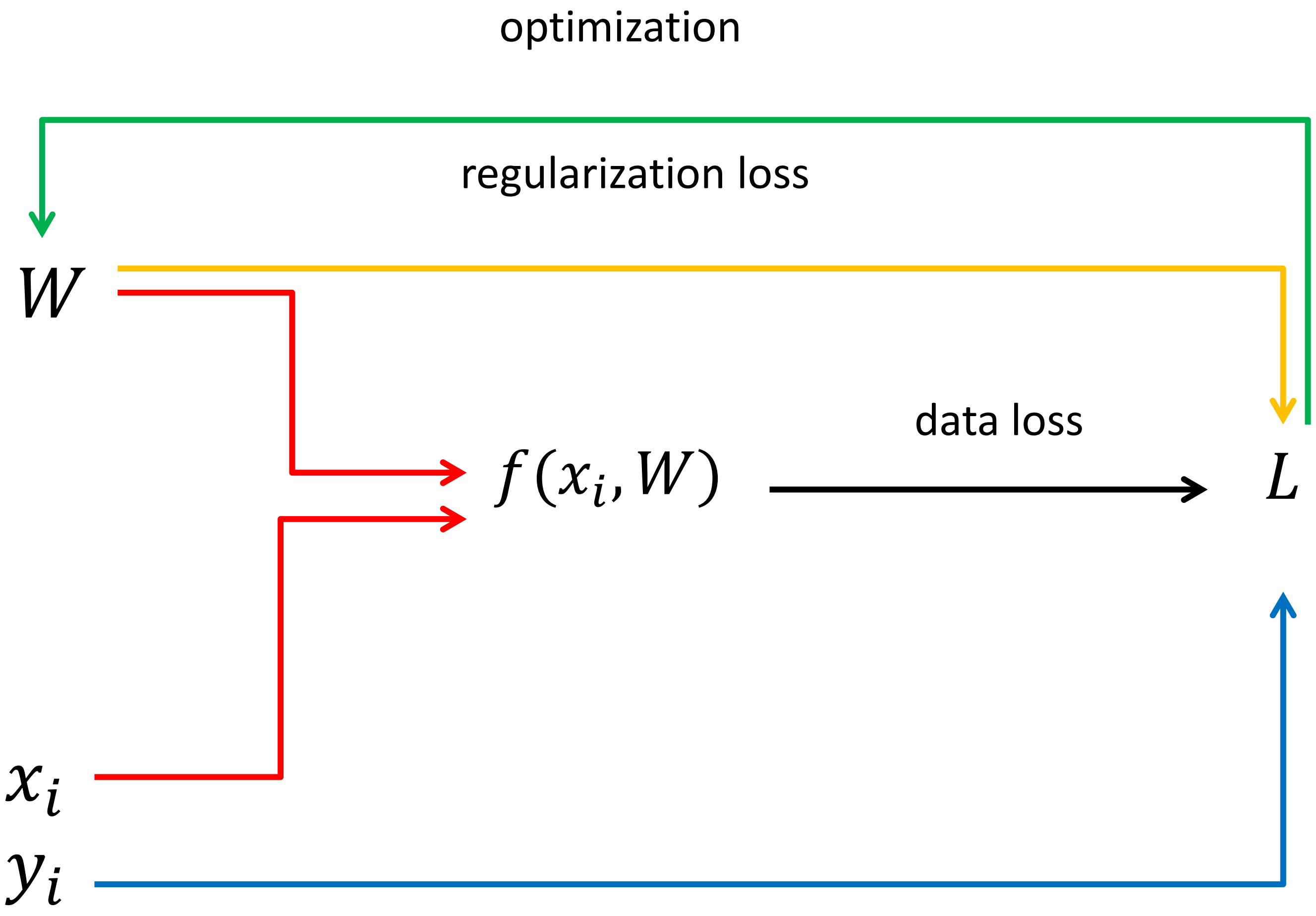
Cross-Entropy Loss

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$





Linear Classifier

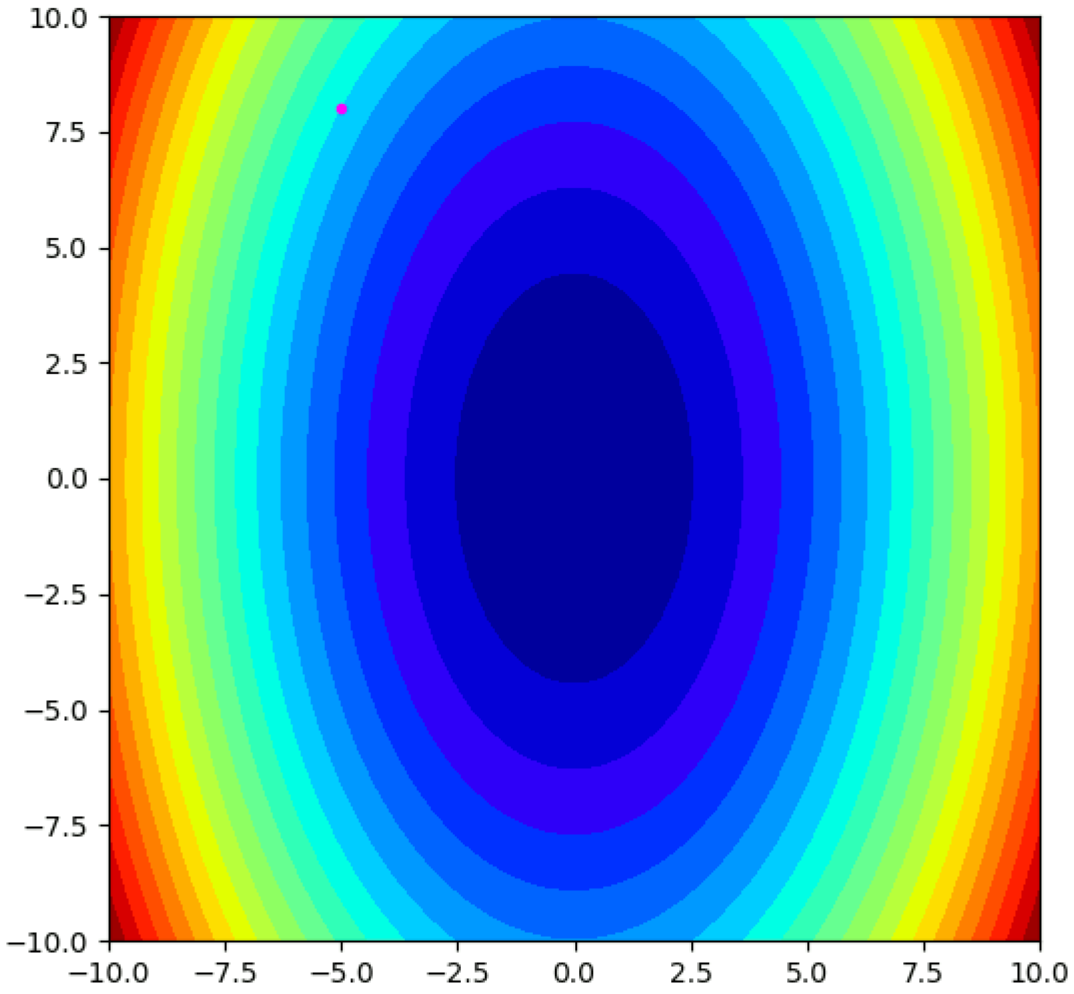


Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:
- Weight initialization method
- Number of steps
- Learning rate

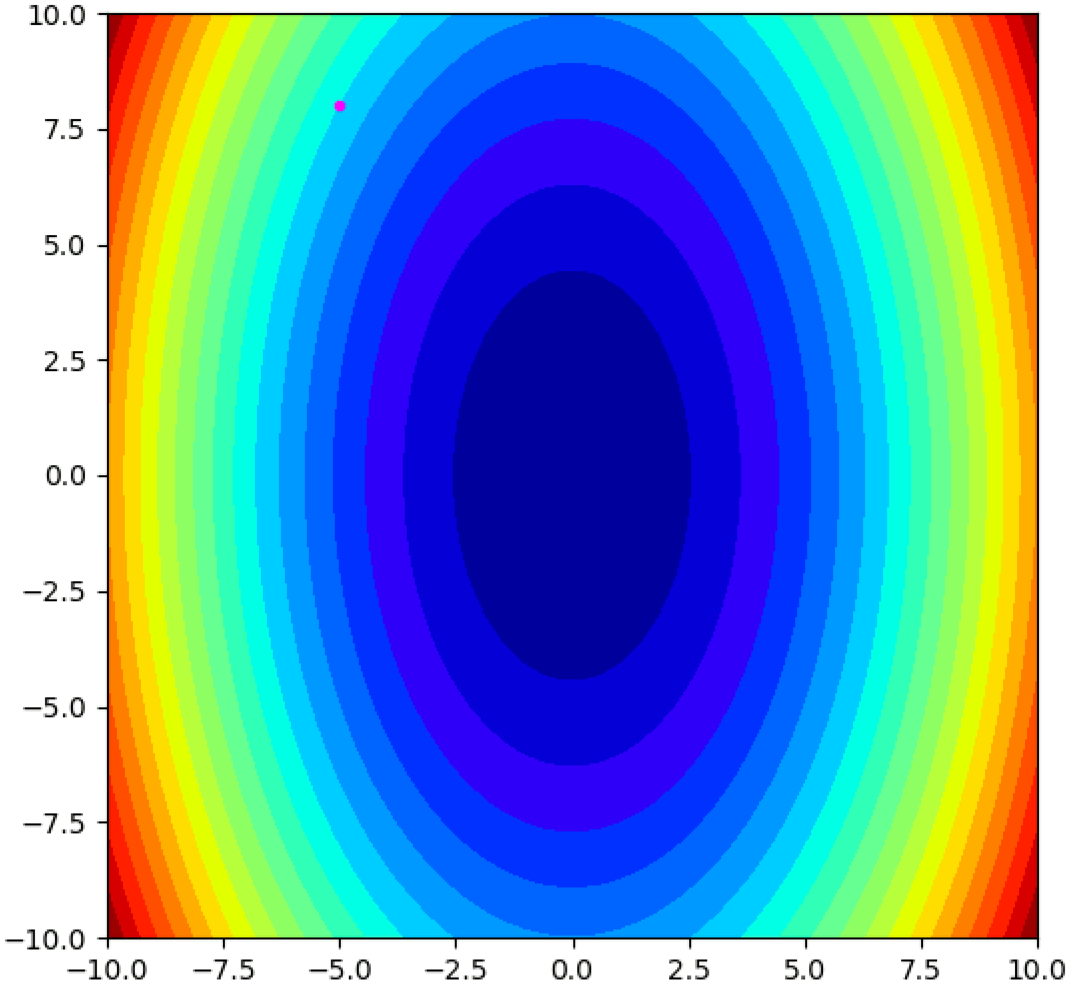


Problems with SGD

Gradients are calculated from minibatches → they can be **noisy**

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

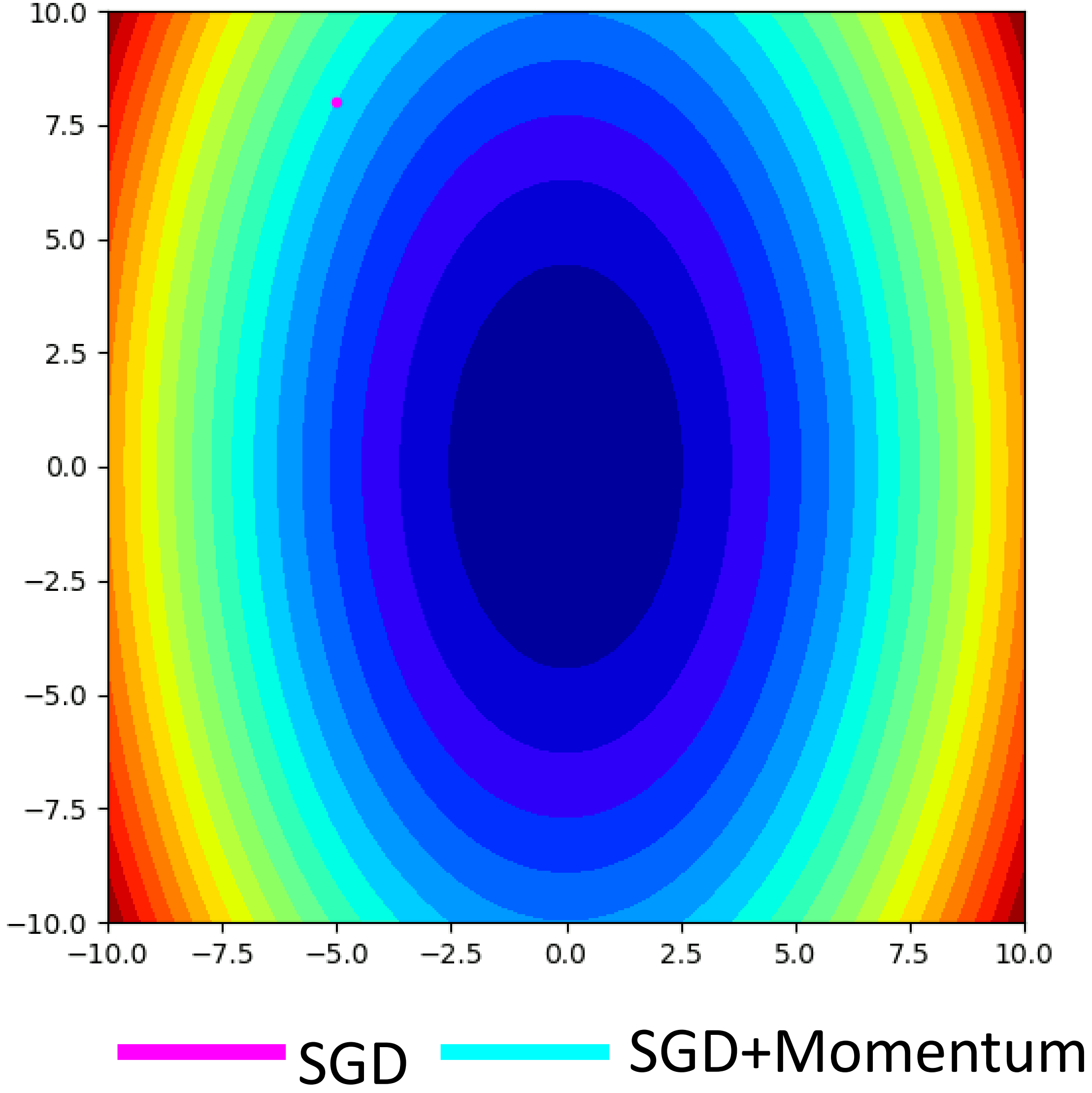


SGD + Momentum

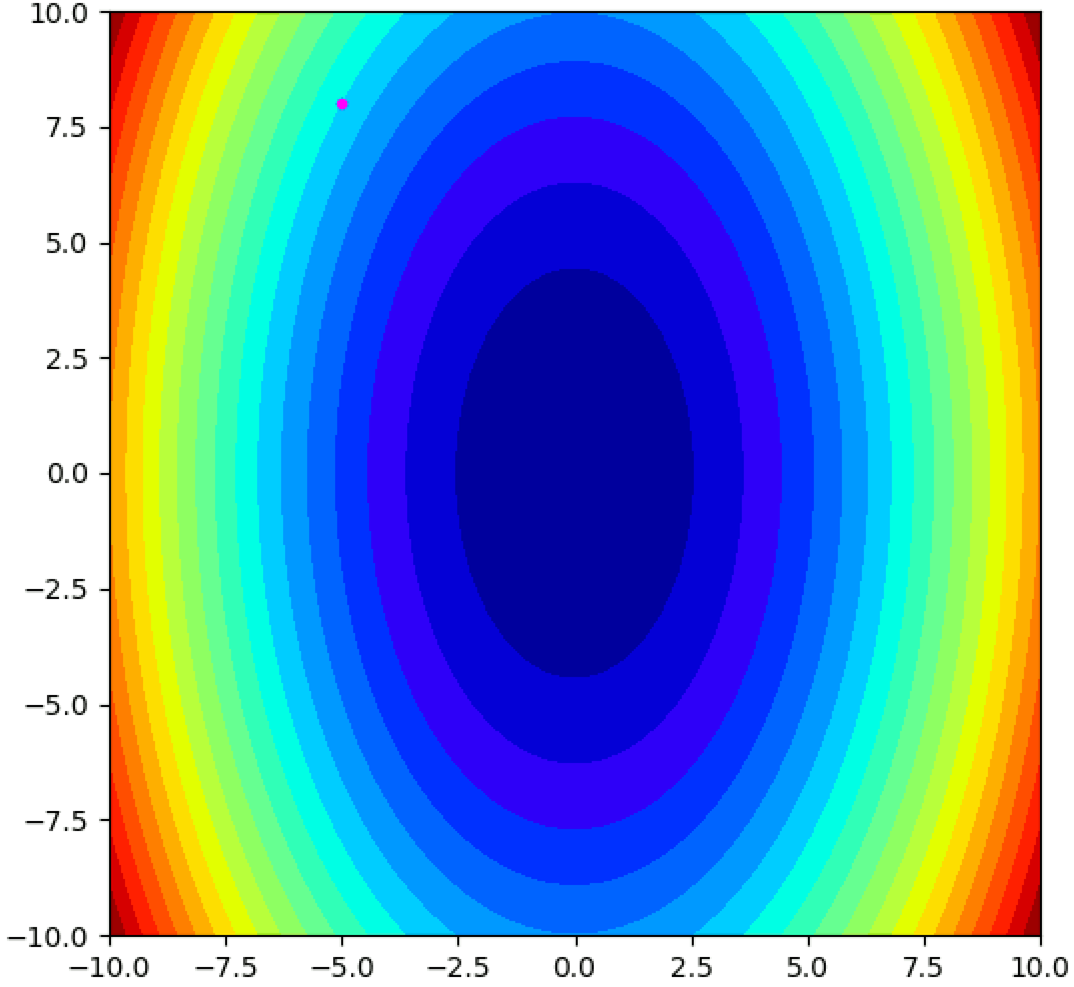
Local Minima

Saddle points

Gradient Noise



Adam



SGD

SGD+Momentum

RMSProp

Adam

Summary

1. Use **Linear Models** for image classification problems

$$s = f(x; W) = Wx$$

2. Use **Loss Functions** to express preferences over different selection of weights

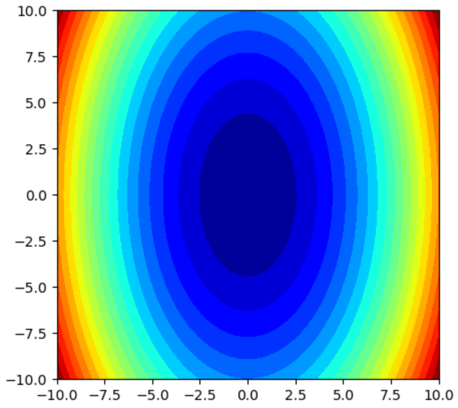
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$ Softmax

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$ SVM

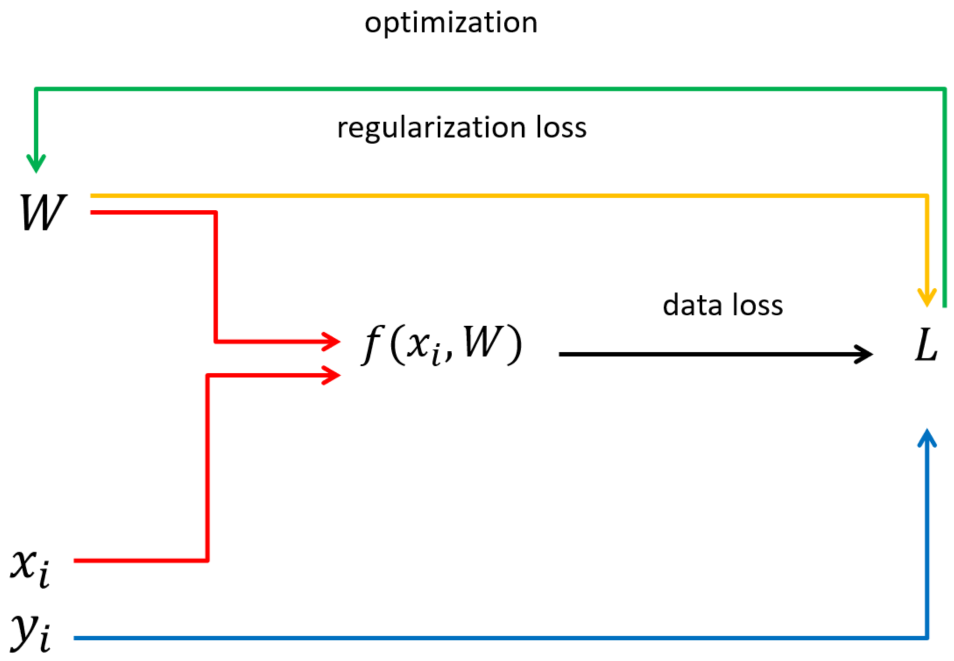
$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$

3. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

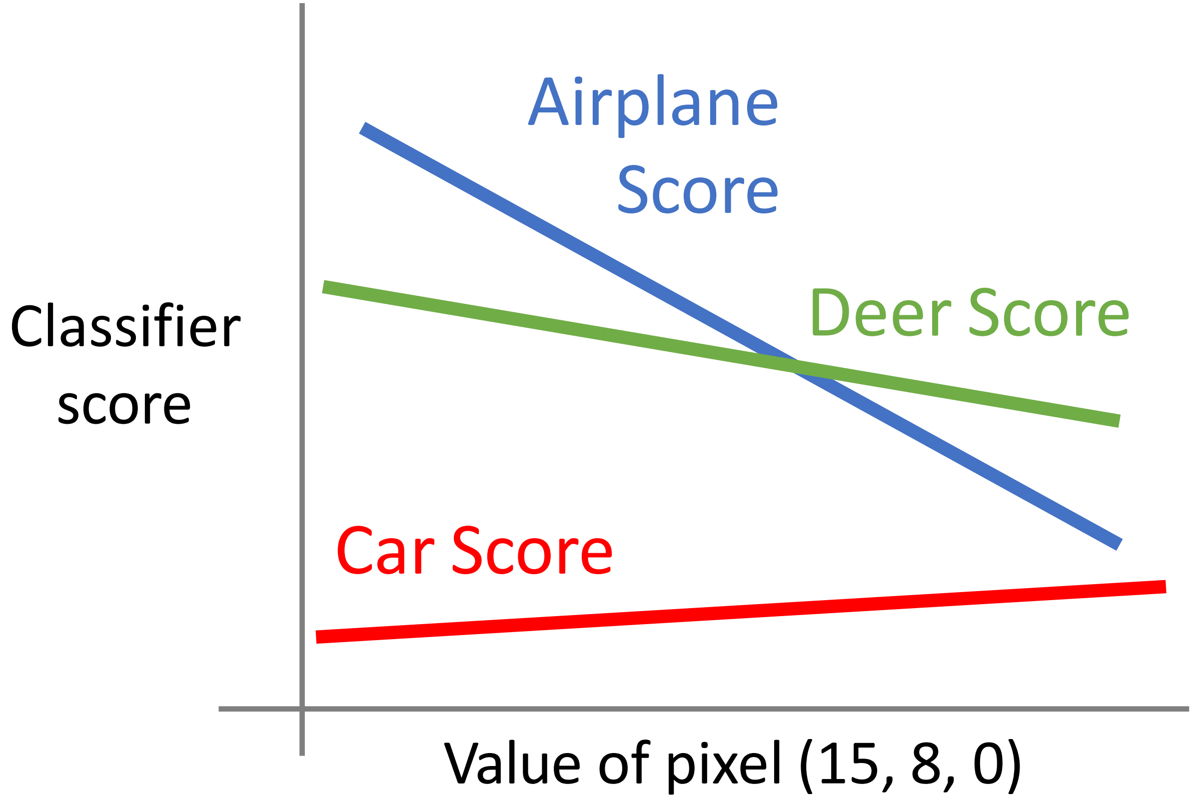
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```



Cifar10 Linear Classifier



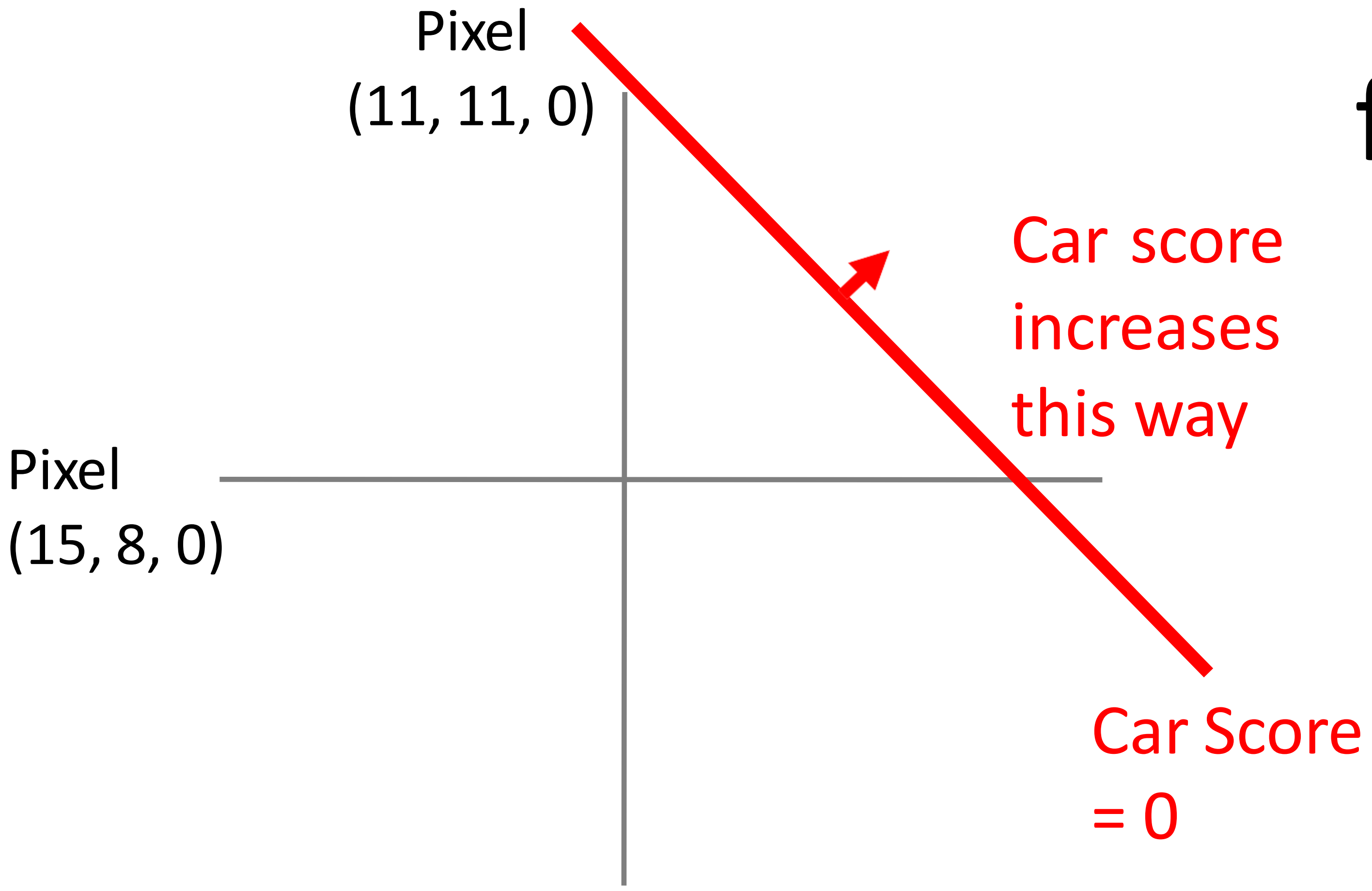
Geometric Interpretation: Linear Classifier



$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

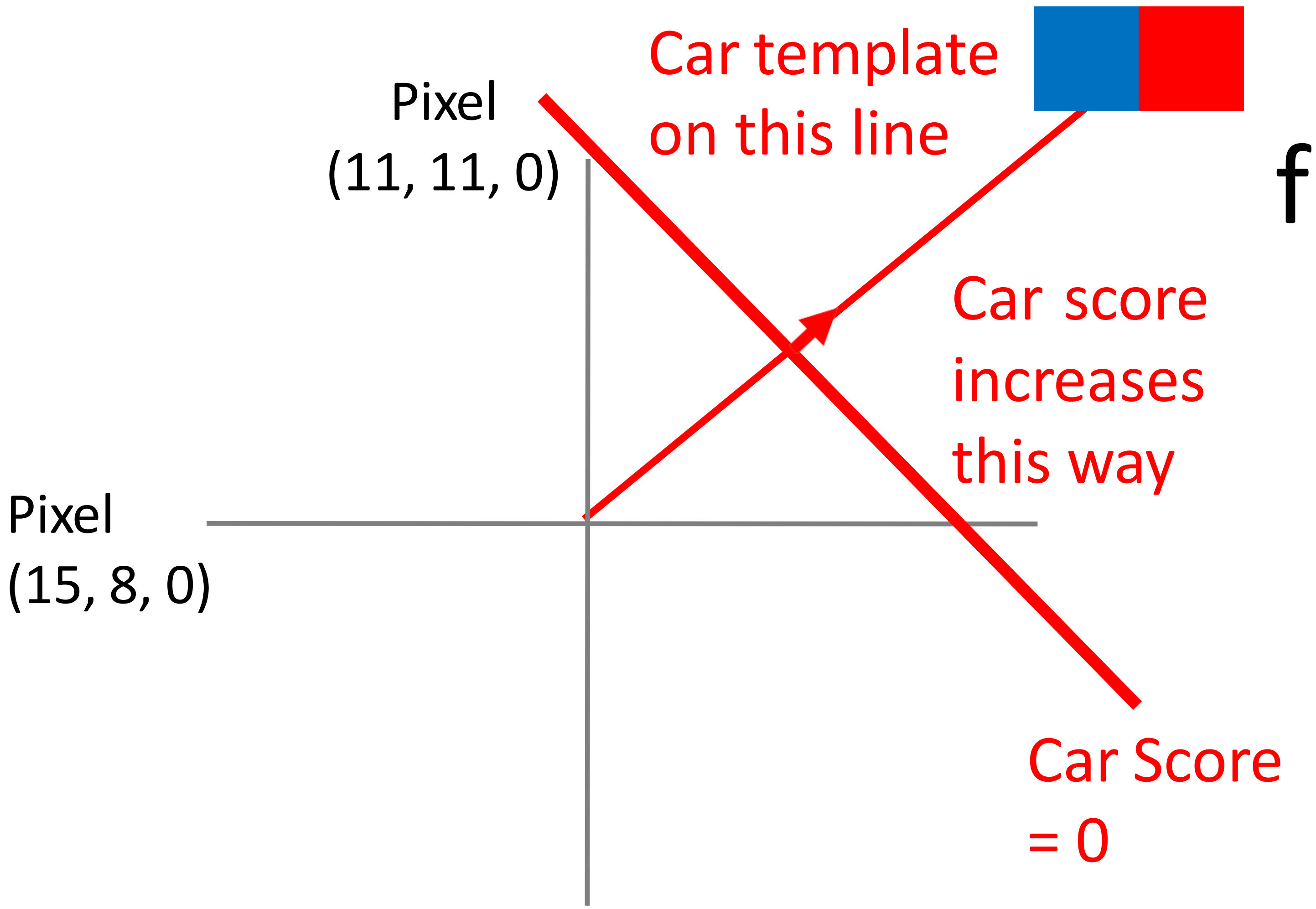
Geometric Interpretation: Linear Classifier



$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

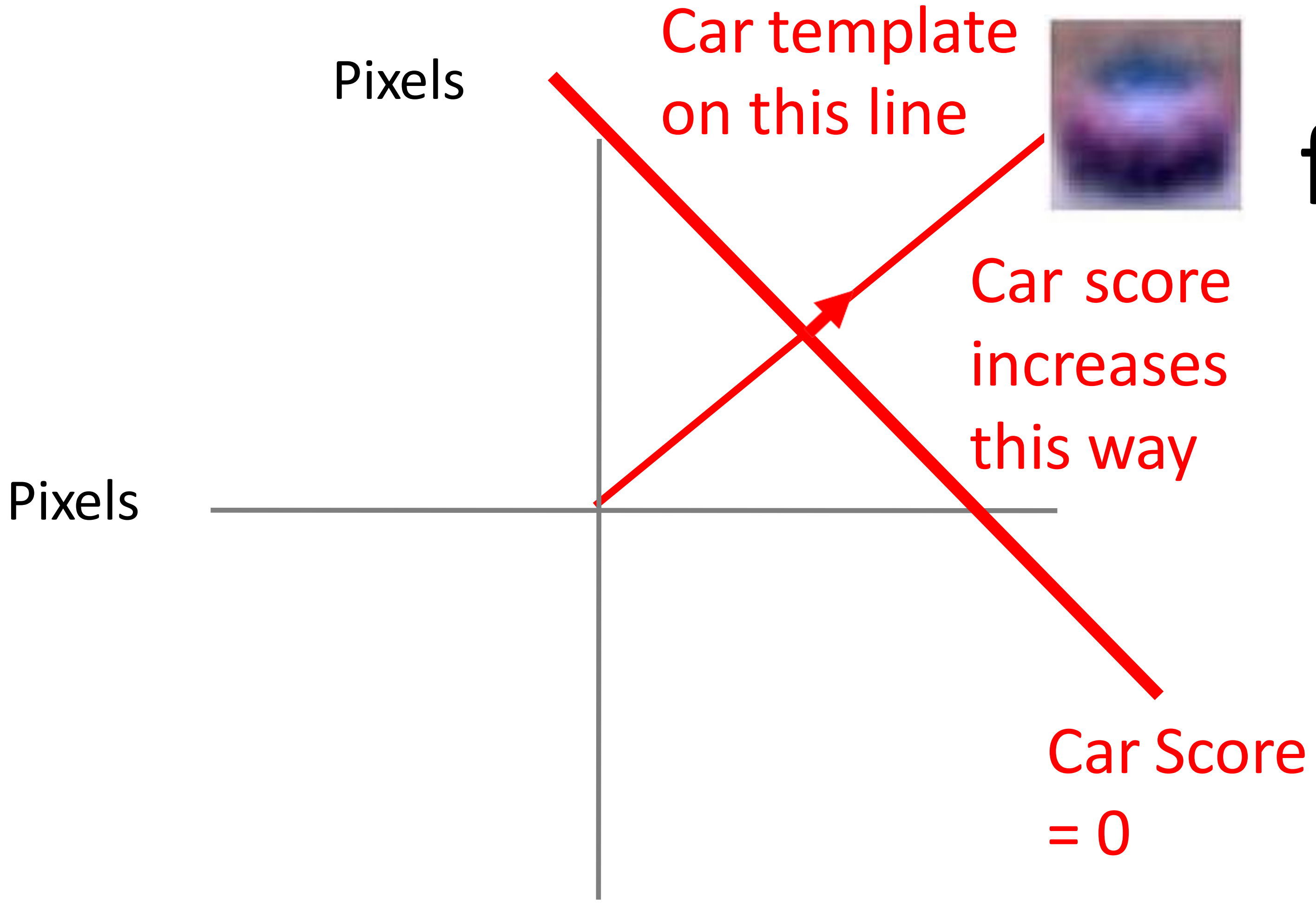
Geometric Interpretation: Linear Classifier



$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

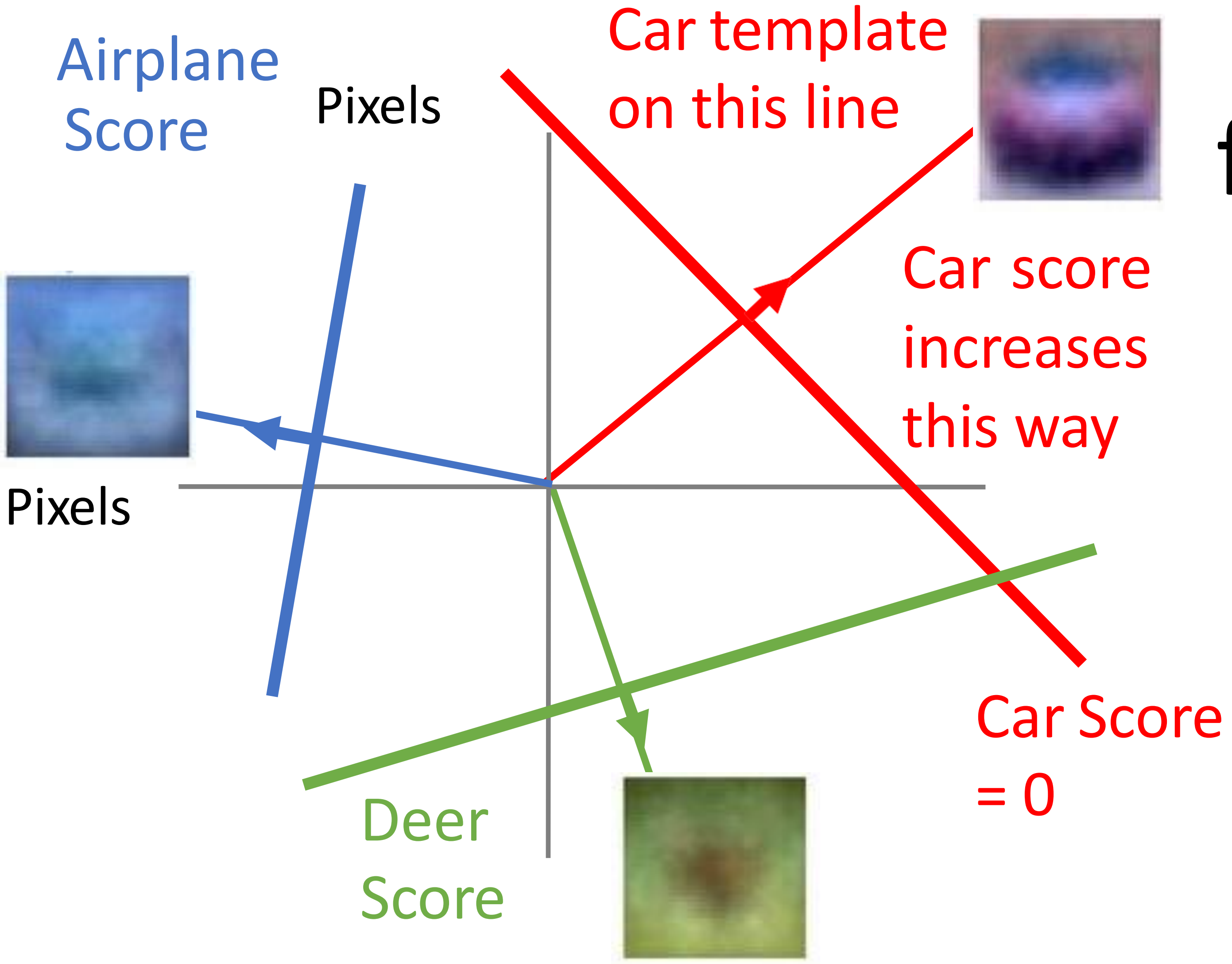
Geometric Interpretation: Linear Classifier



$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

Geometric Interpretation: Linear Classifier

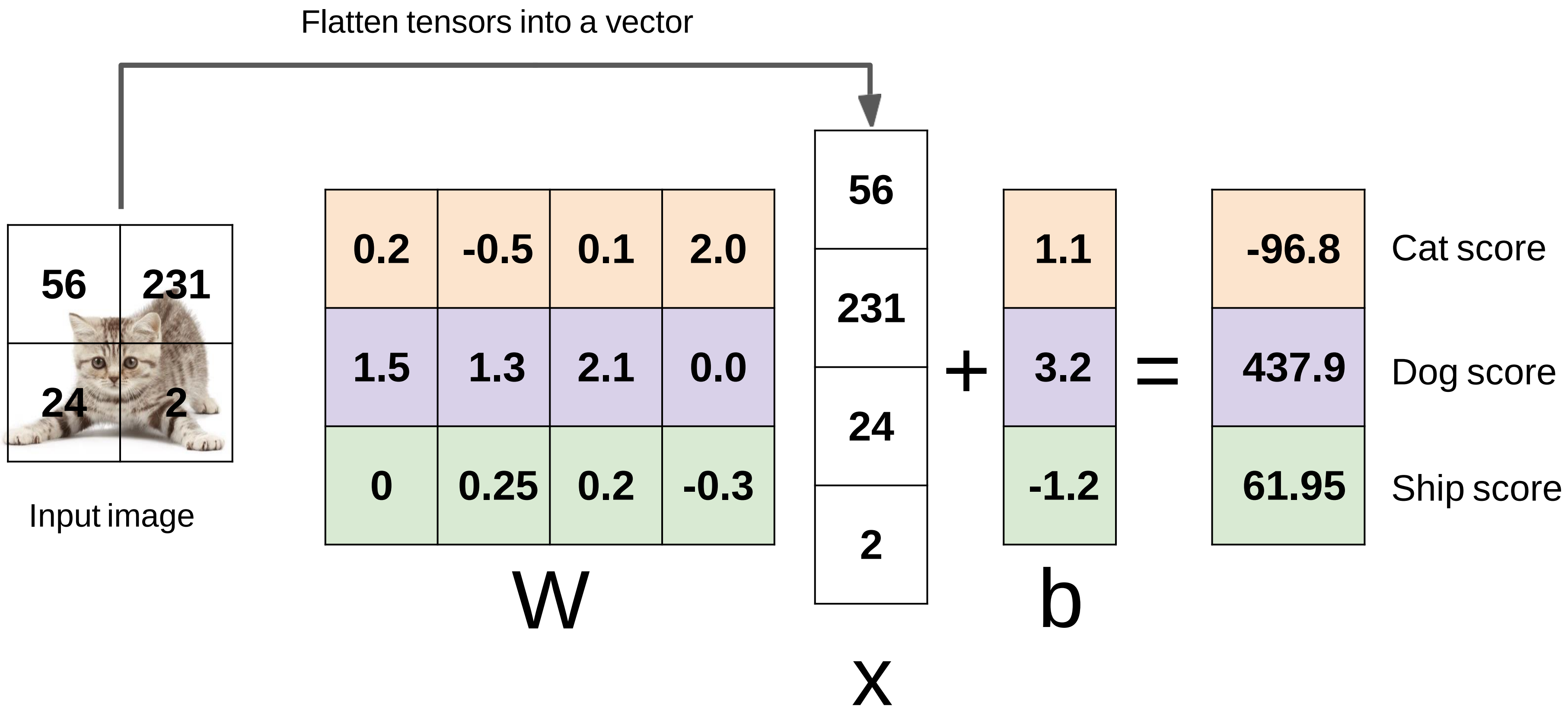


$$f(x,W) = Wx + b$$

Array of **32x32x3** numbers
(3072 numbers total)

Visual Interpretation: Linear Classifier

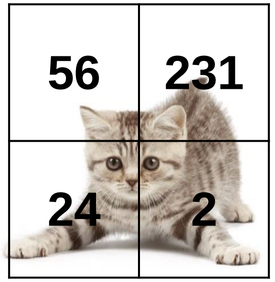
Linear classifier has one “template” per category



Visual Interpretation: Linear Classifier

A single template cannot capture multiple modes of the data

Flatten tensors into a vector



Input image

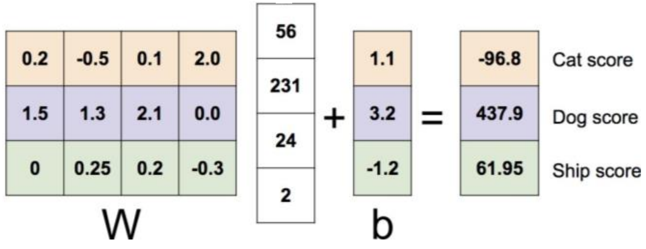
$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W} \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix} \begin{matrix} \text{Cat score} \\ \text{Dog score} \\ \text{Ship score} \end{matrix}$$

e.g. horse template has 2 heads

Linear Classifiers

Algebraic Interpretation

$f(x,W) = Wx$

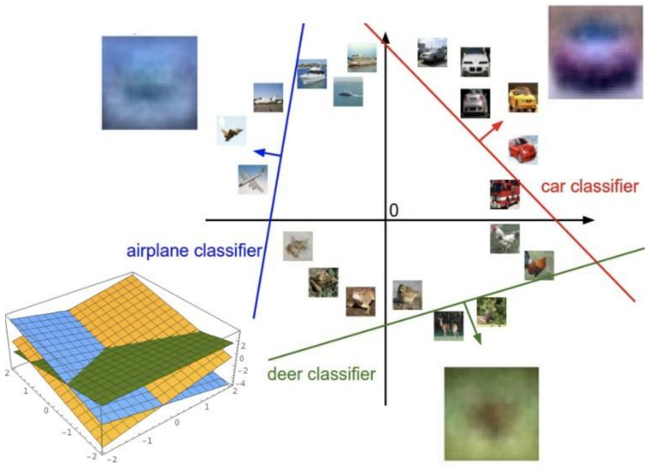


Visual Interpretation

One template
per class

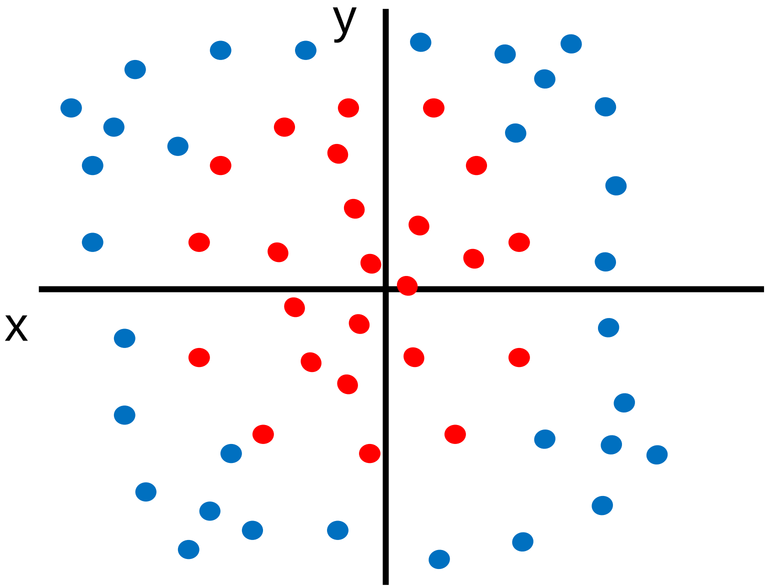
Geometric Interpretation

Hyperplanes
separating space



Linear Classifiers shortcomings

Geometric Viewpoint

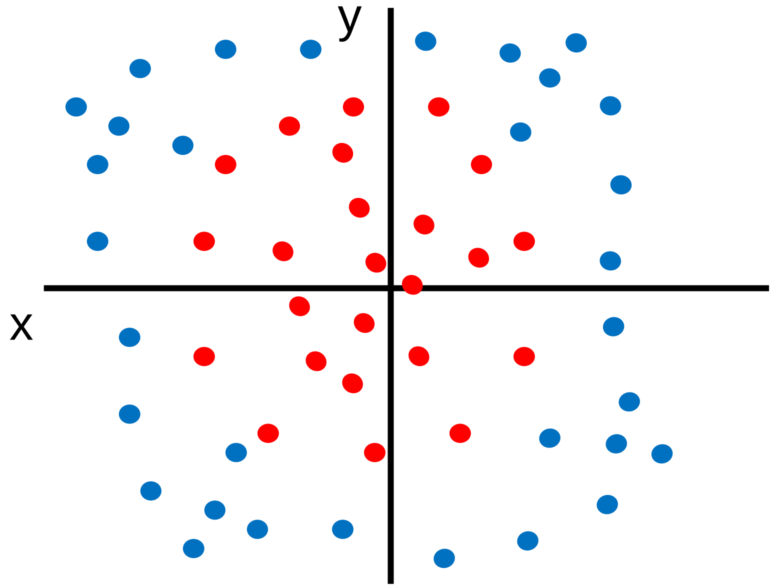


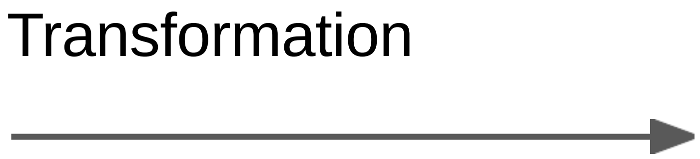
Some training data can't be separated with a hyperplane

Visual Viewpoint

One template per class: Can't recognize different modes of a class

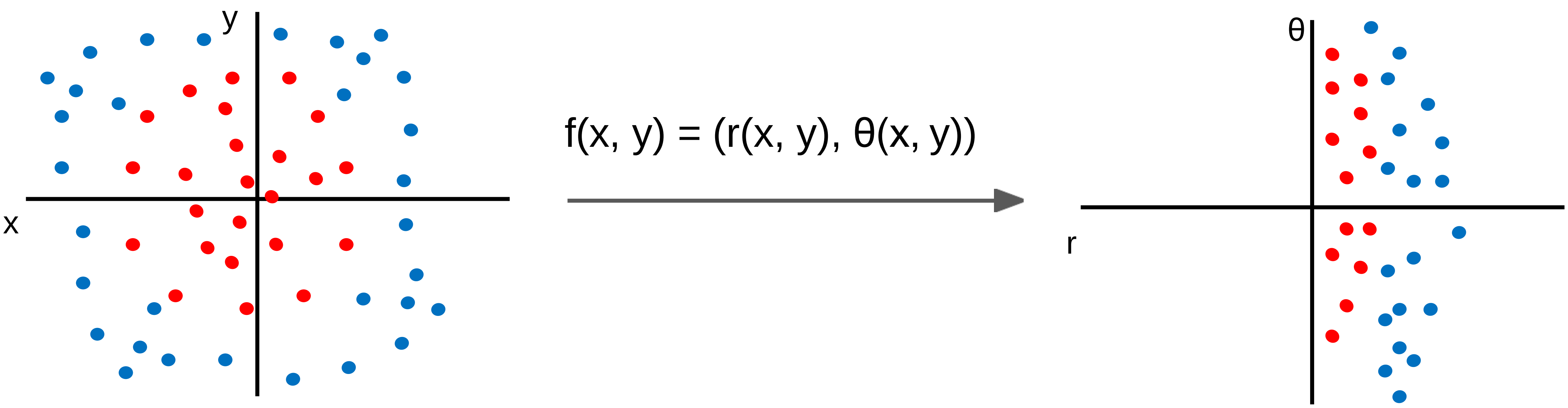
Apply Transformations





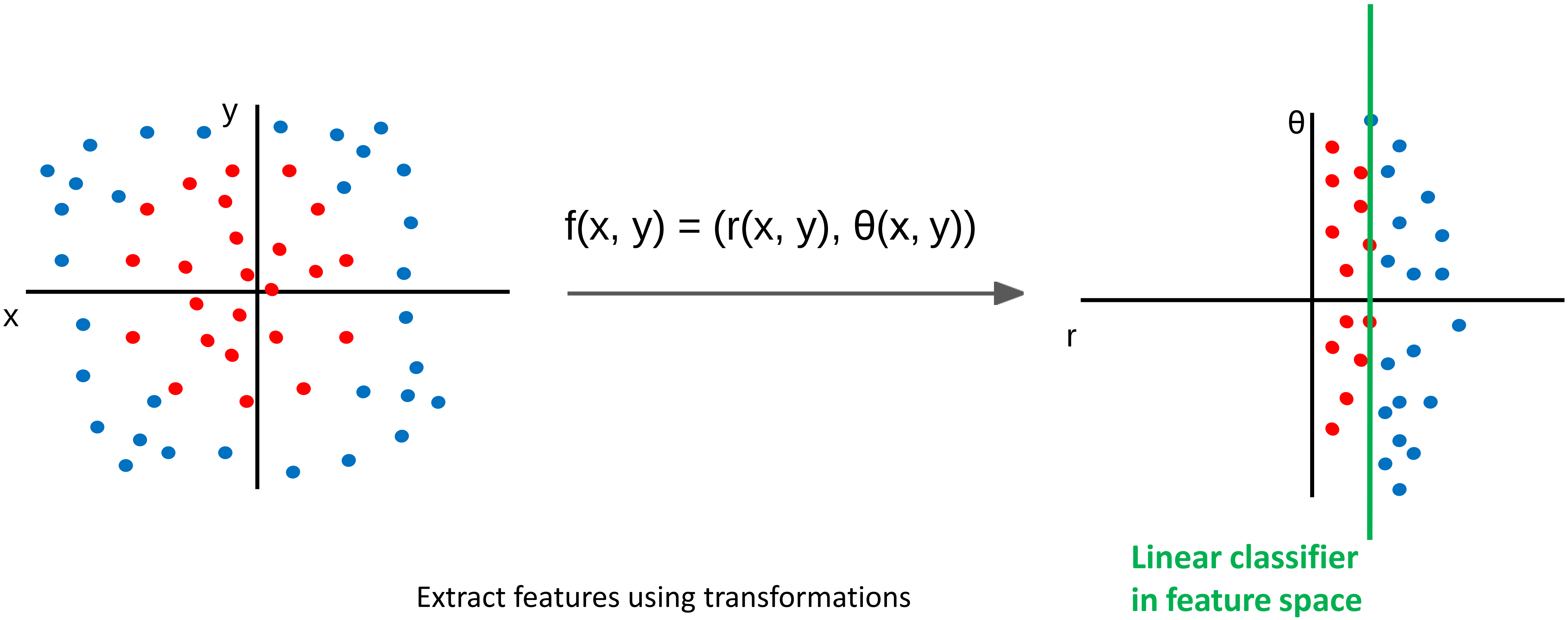
Extract features using transformations

Apply Transformations



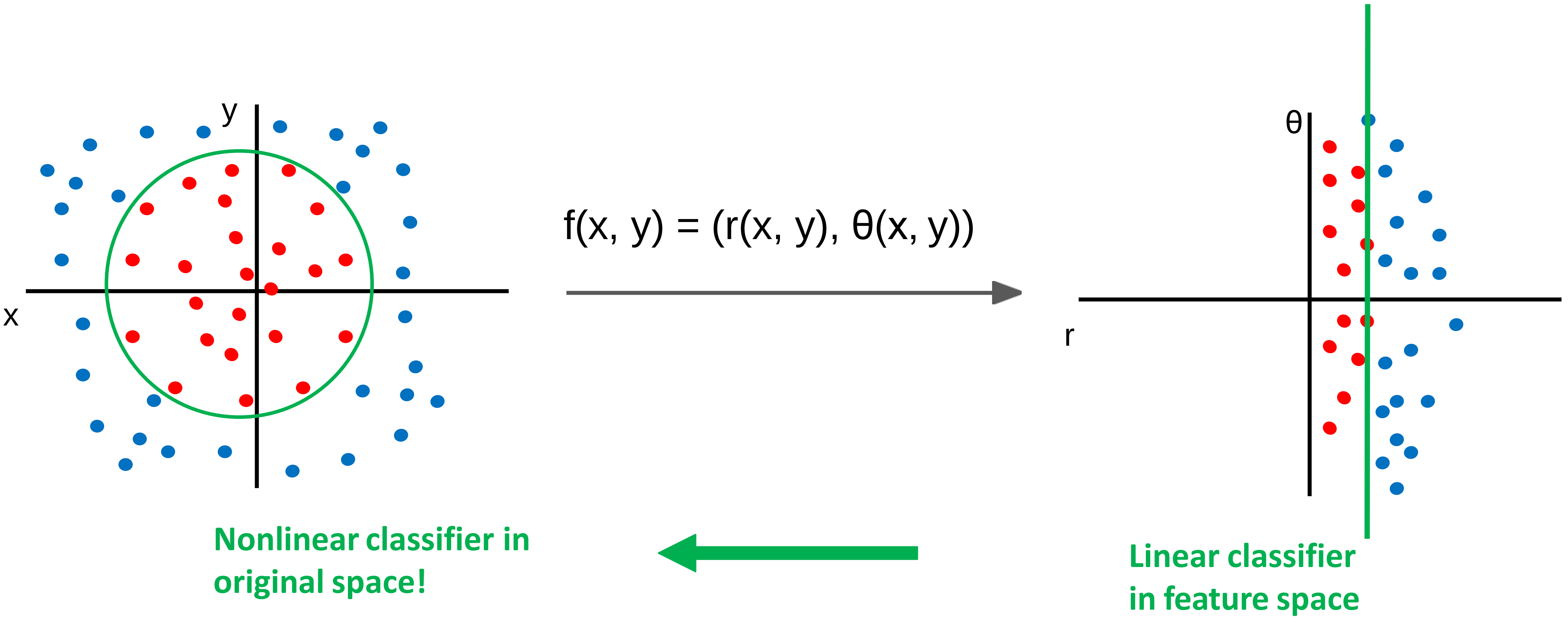
Extract features using transformations

Apply Transformations

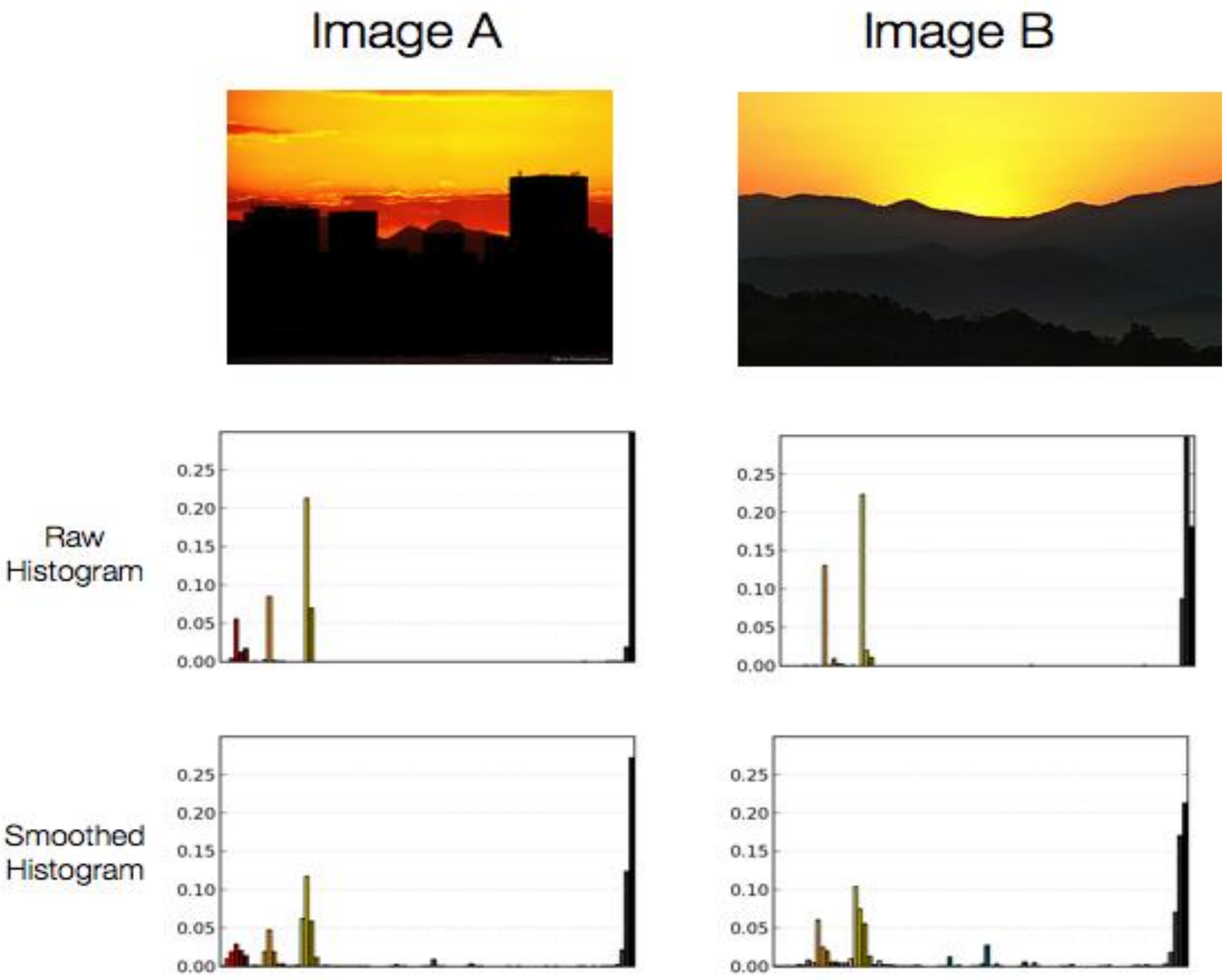


Extract features using transformations

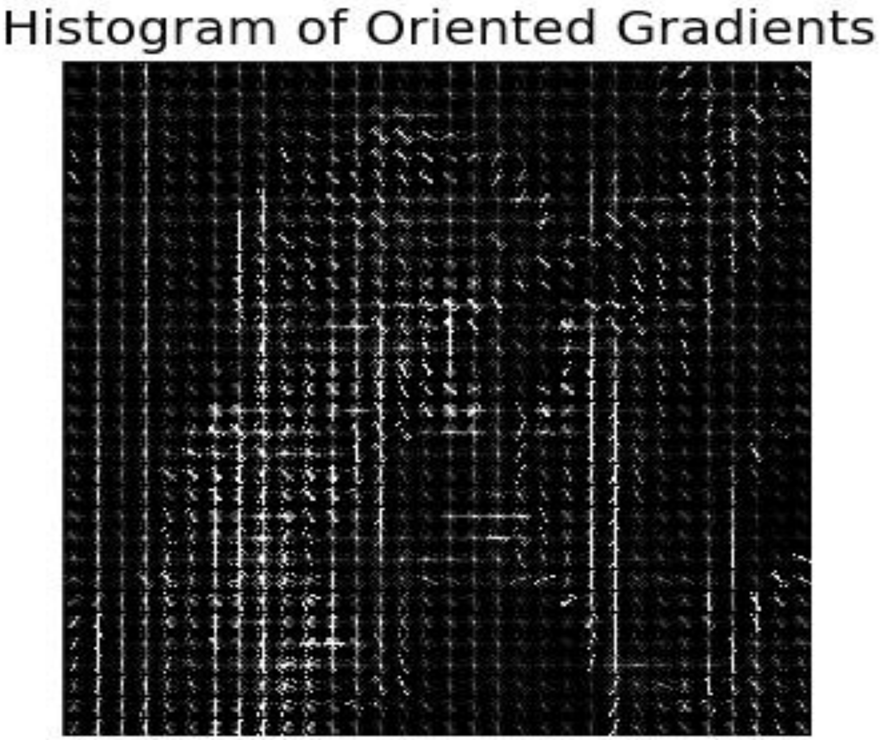
Apply Transformations



Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)



Lowe, “Object recognition from local scale-invariant features”, ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

Image Feature Aggregation

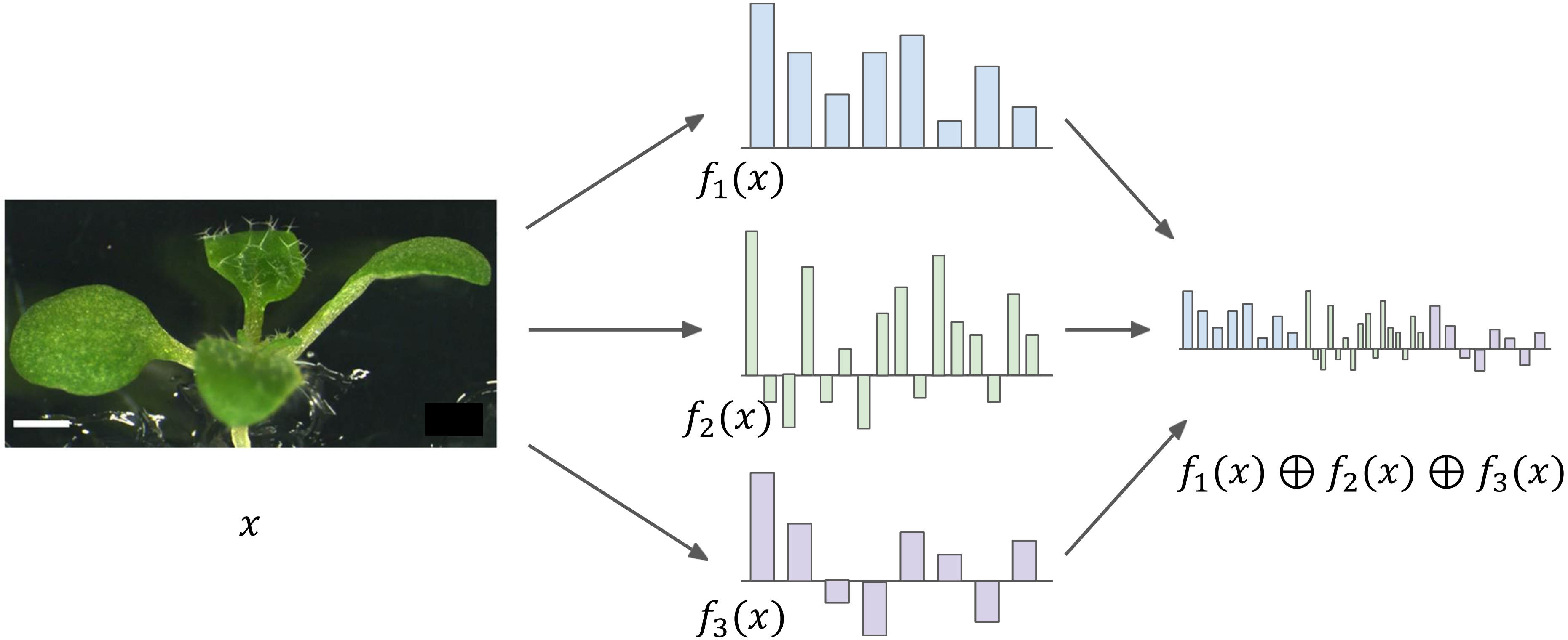
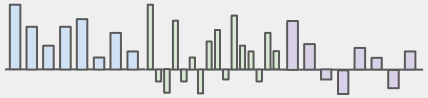


Image Features

Feature Extraction



f

10 numbers giving
scores for classes

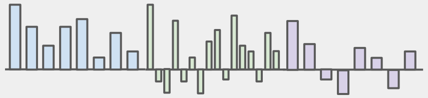
+

Class Label

training

Image Features

Feature Extraction



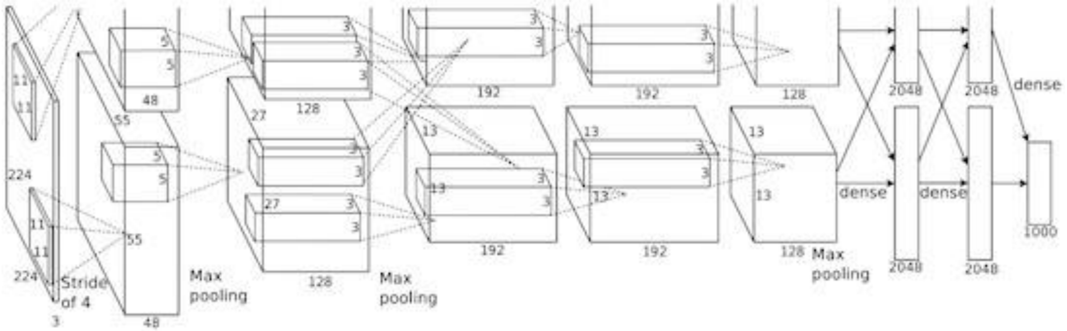
f

10 numbers giving scores for classes

\+

Class Label

training



10 numbers giving scores for classes

training

Neural Networks

(**Before**) Linear score function:

$$f = Wx$$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$$W_2 \in \mathbb{R}^{C \times H} \quad W_1 \in \mathbb{R}^{H \times D} \quad x \in \mathbb{R}^D$$

Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

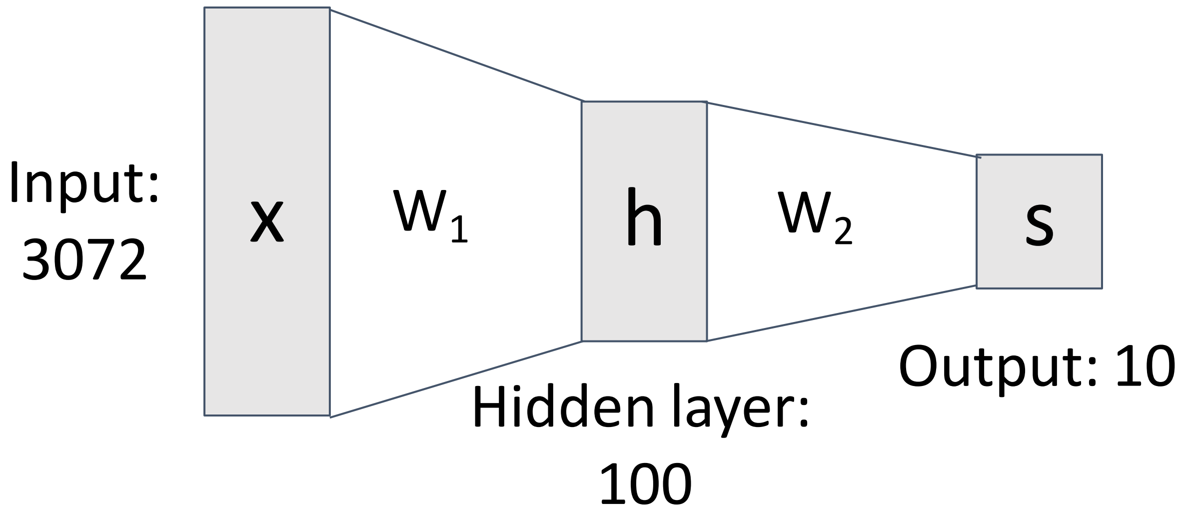
$$W_3 \in \mathbb{R}^{C \times H_2} \quad W_2 \in \mathbb{R}^{H_2 \times H_1} \quad W_1 \in \mathbb{R}^{H_1 \times D} \quad x \in \mathbb{R}^D$$

Neural Networks

$$f = Wx$$
$$f = W_2 \max(0, W_1 x)$$

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



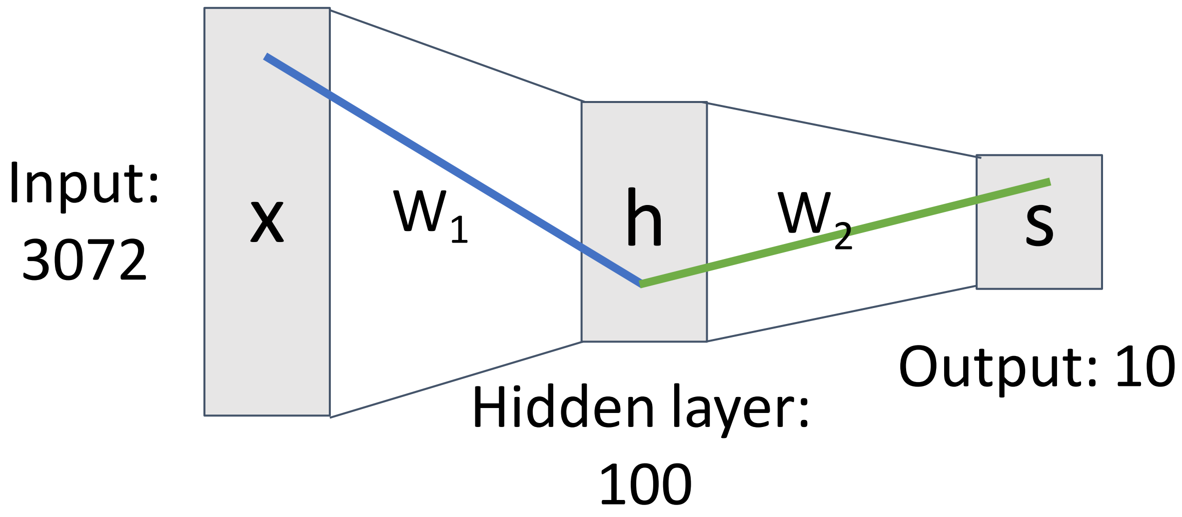
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

Element (i, j) of W_1 gives the effect on h_i from x_j



Element (i, j) of W_2 gives the effect on s_i from h_j

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

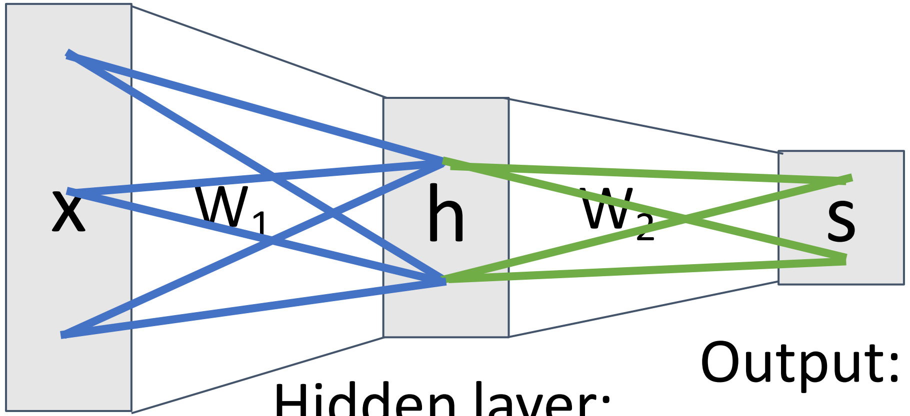
(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

Element (i, j) of W_1 gives the effect on h_i from x_j

All elements of x affect all elements of h

Input: 3072



Hidden layer: 100

Output: 10

Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

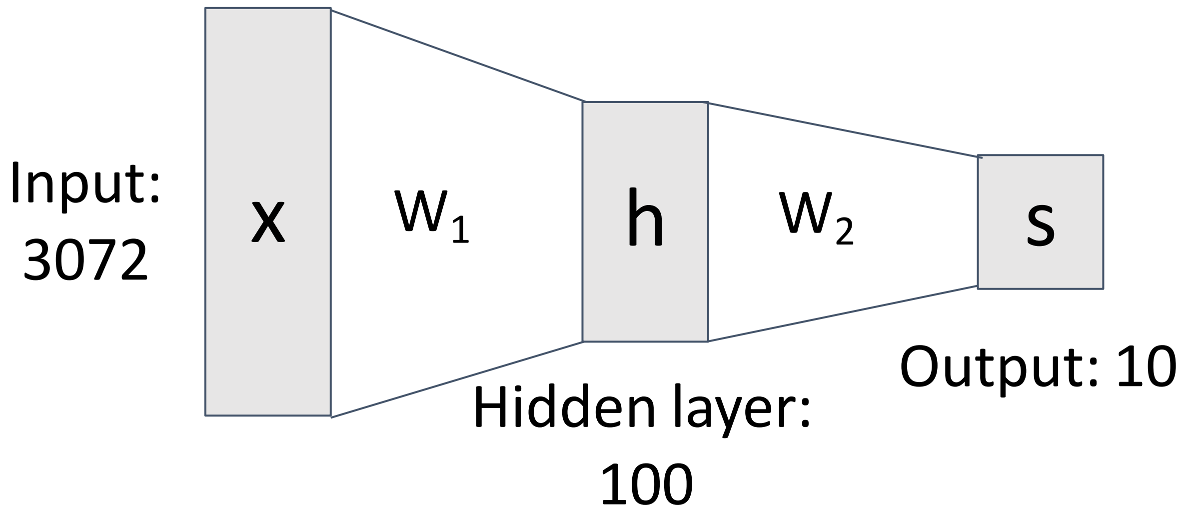
Fully-connected neural network
Also "Multi-Layer Perceptron" (MLP)

Neural Networks

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network

Linear classifier: One template per class

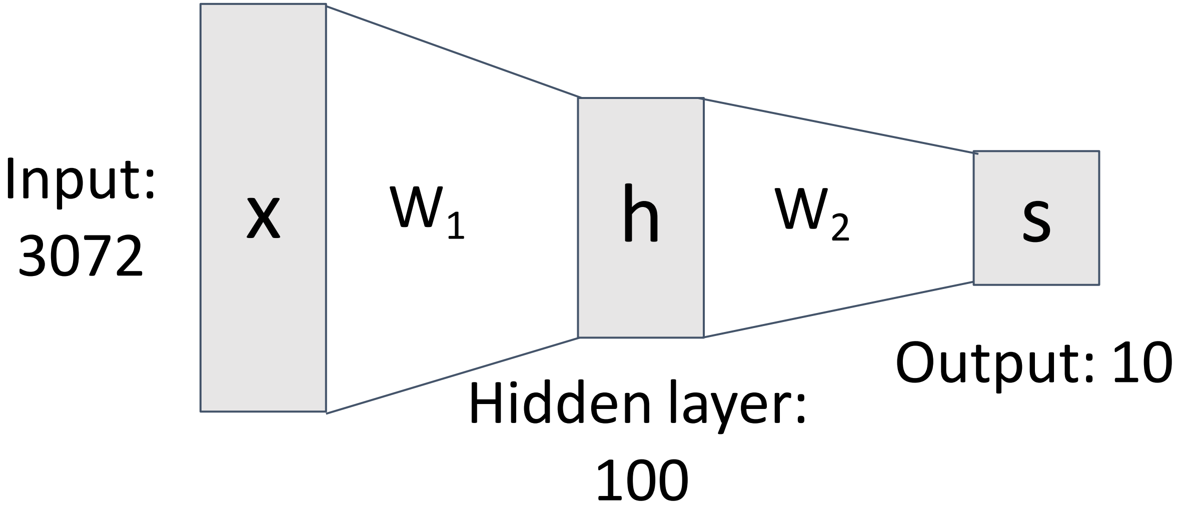


$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



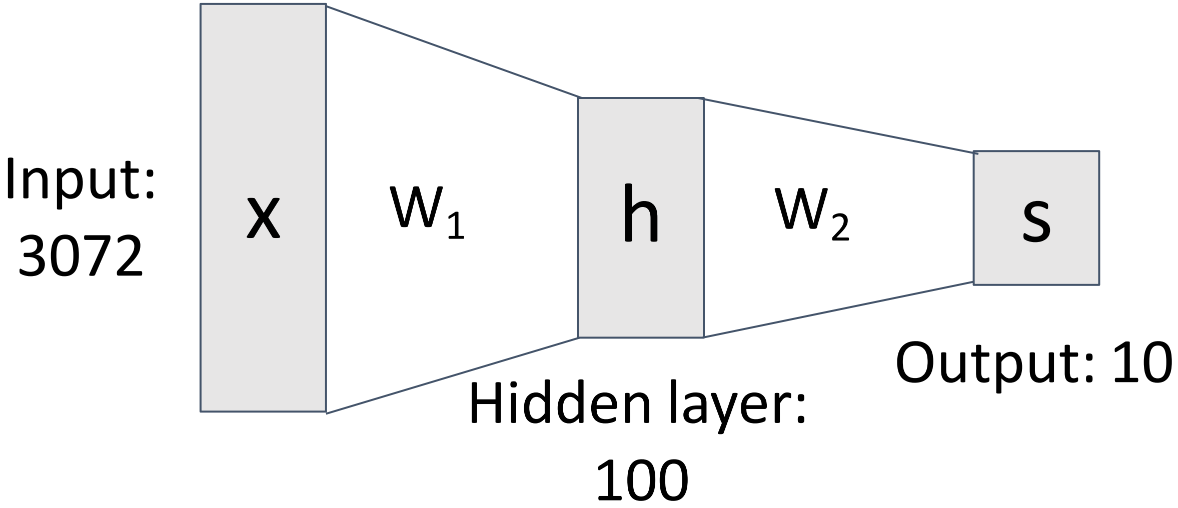
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Can use different templates to cover multiple modes of a class

(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



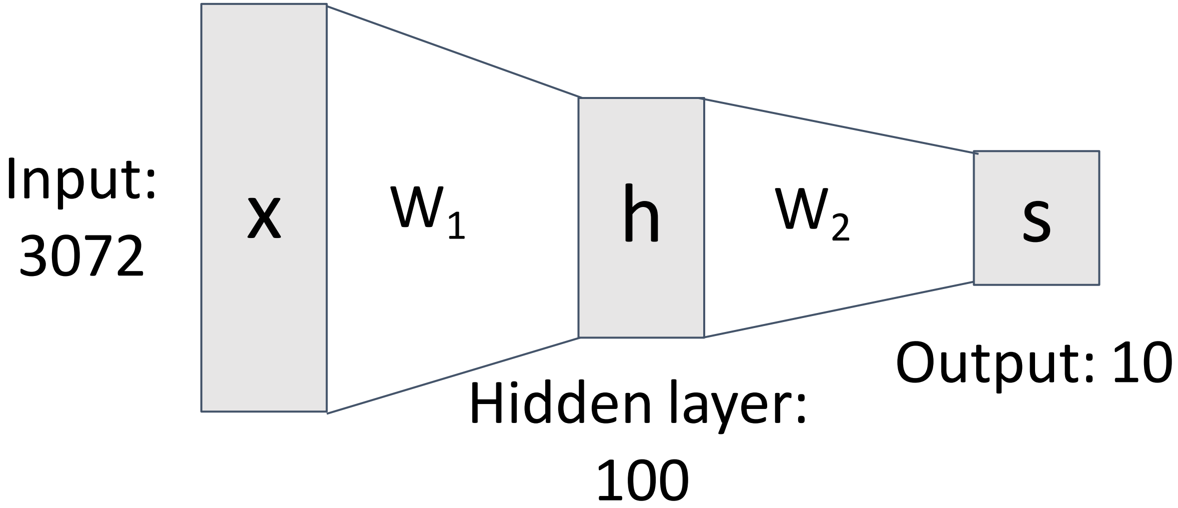
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

Most templates not interpretable

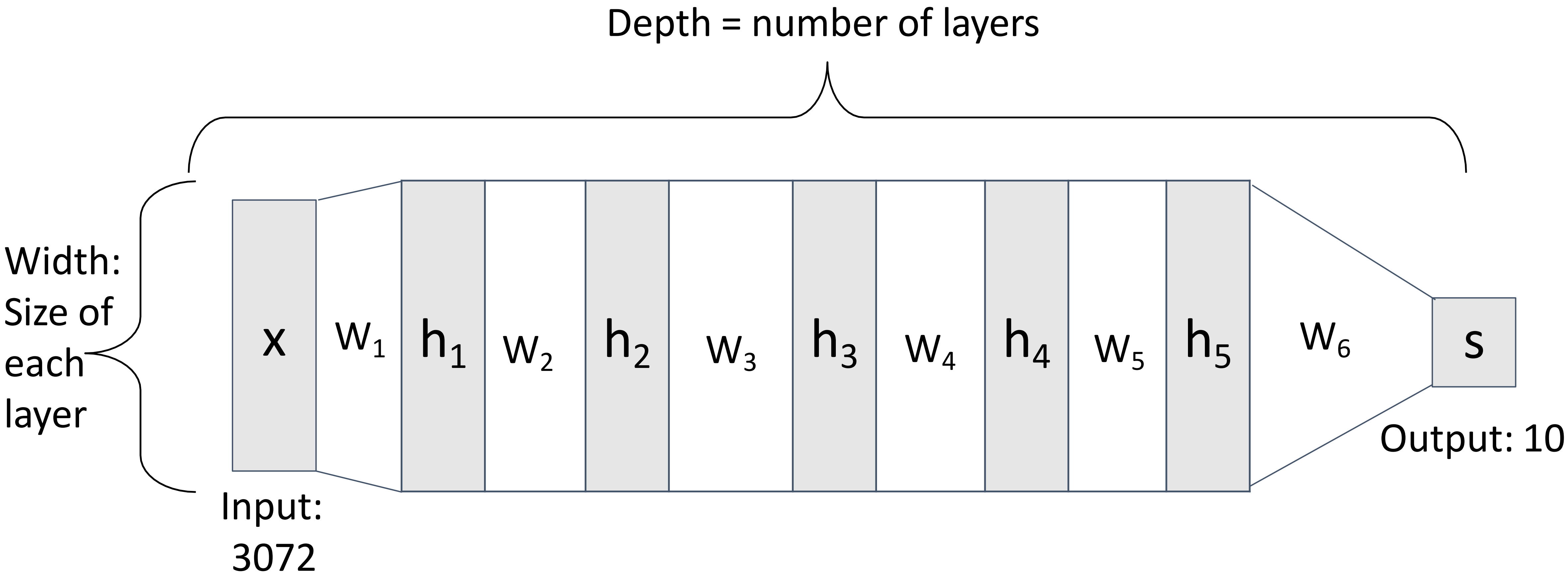
(**Before**) Linear score function:

(**Now**) 2-layer Neural Network



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Deep Neural Networks



$$s = W_6 \max(0, W_6 \max(0, W_5 \max(0, W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))))$$

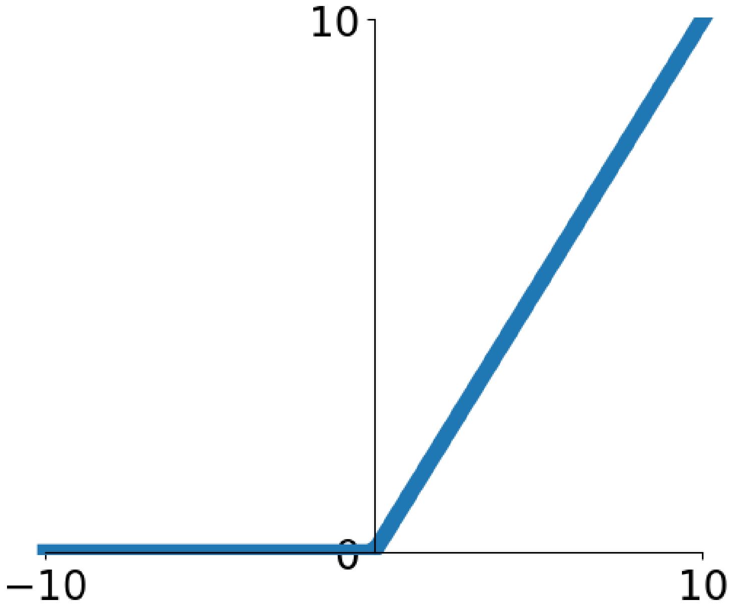
Activation Functions

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

The function $ReLU(z) = max(0, z)$ is called “Rectified Linear Unit”

This is called the **activation function** of the neural network



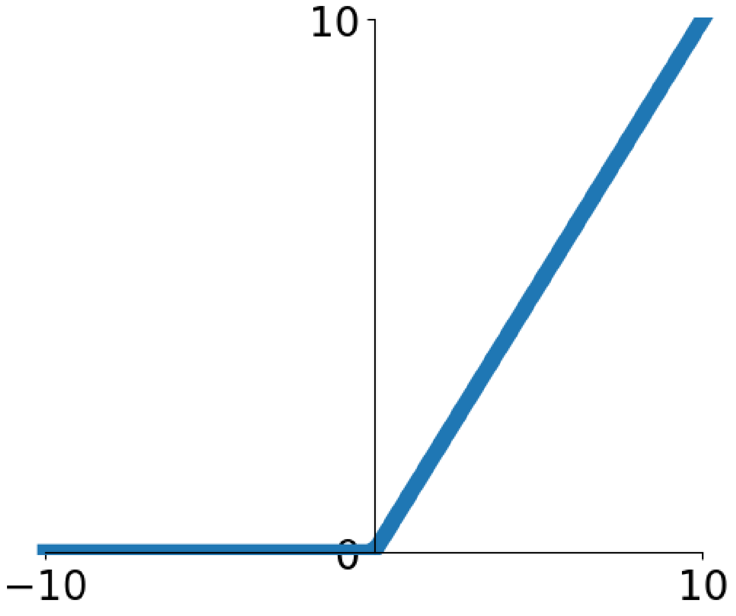
Activation Functions

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

The function $ReLU(z) = max(0, z)$ is called "Rectified Linear Unit"

This is called the **activation function** of the neural network



Without activation function:

$$s = W_2 W_1 x$$

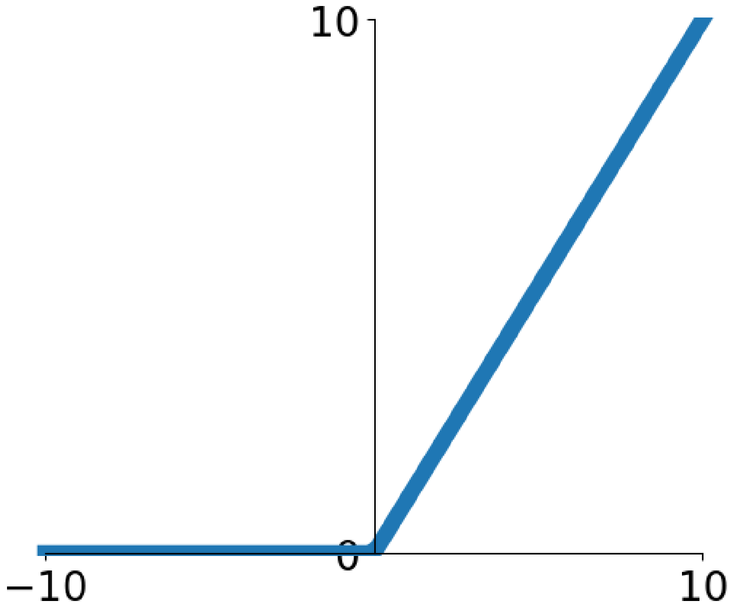
Activation Functions

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

The function $ReLU(z) = max(0, z)$ is called "Rectified Linear Unit"

This is called the **activation function** of the neural network



Without activation function:

$$s = W_2 W_1 x$$

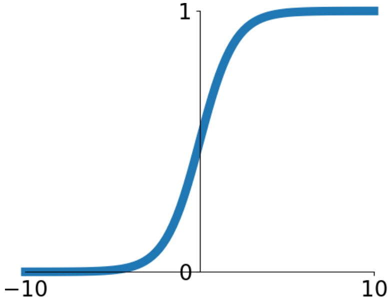
$$W_3 = W_2 W_1 \in \mathbb{R}^{C \times H} \quad s = W_3 x$$

→ Linear classifier

Activation Functions

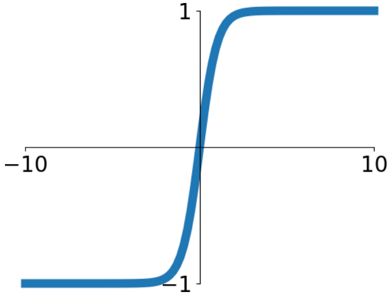
Sigmoid

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



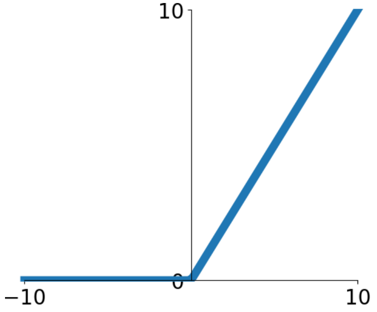
tanh

$\tanh(x)$



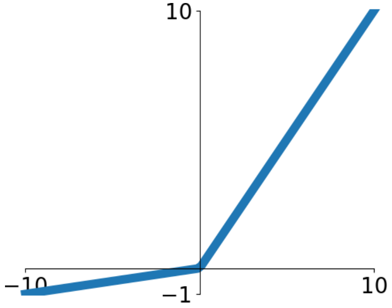
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

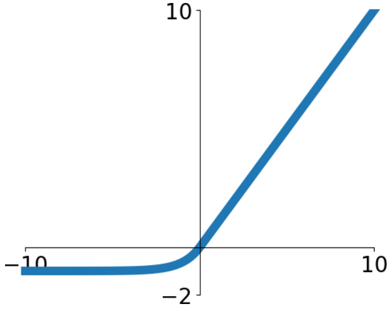


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

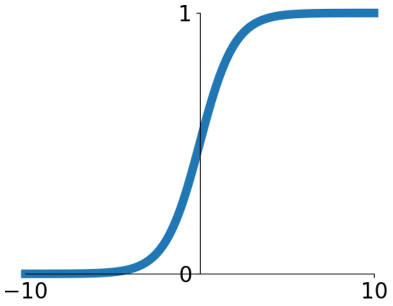
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

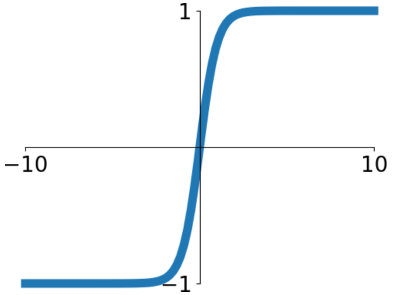
Sigmoid

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



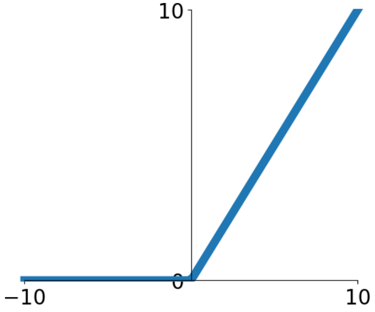
tanh

$$\tanh(x)$$



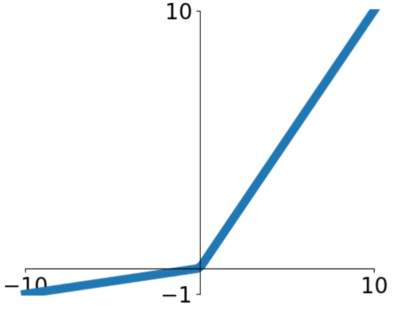
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

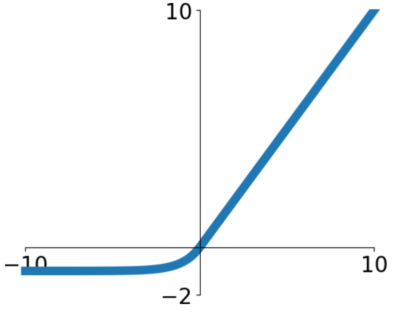


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

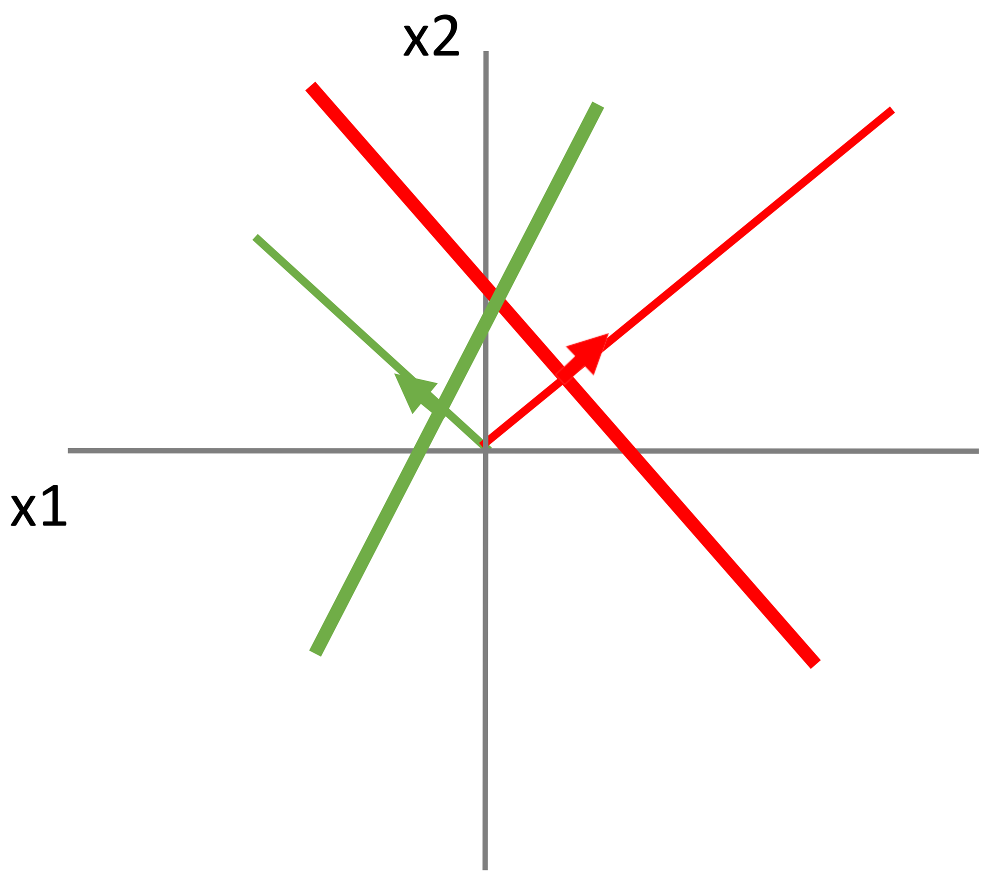
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



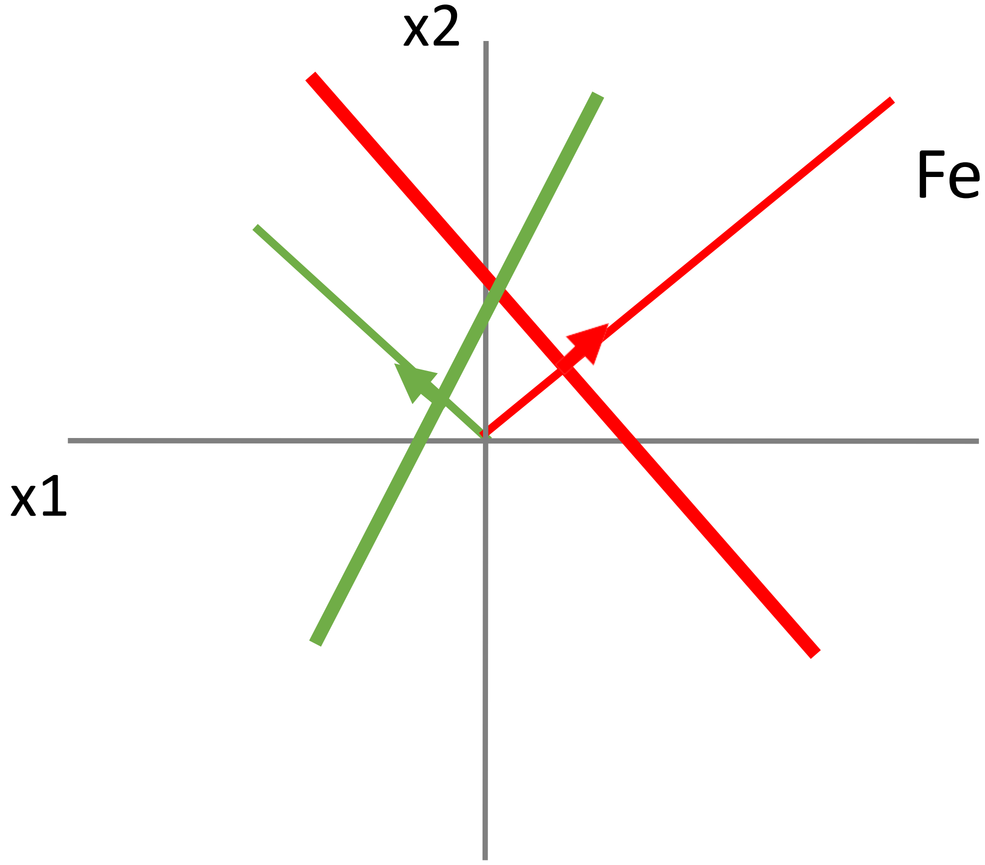
Feature Transform

Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

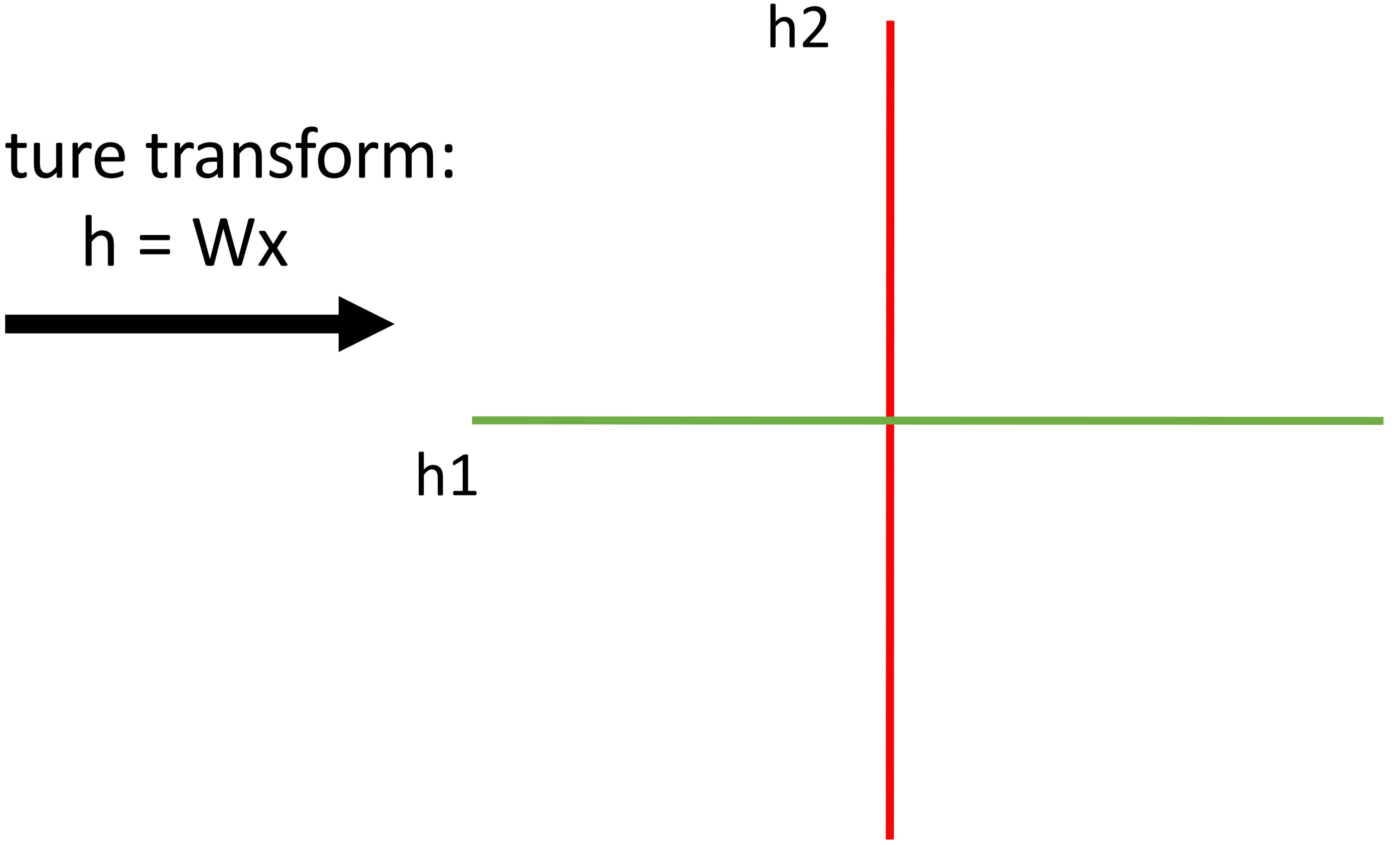


Feature Transform

Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

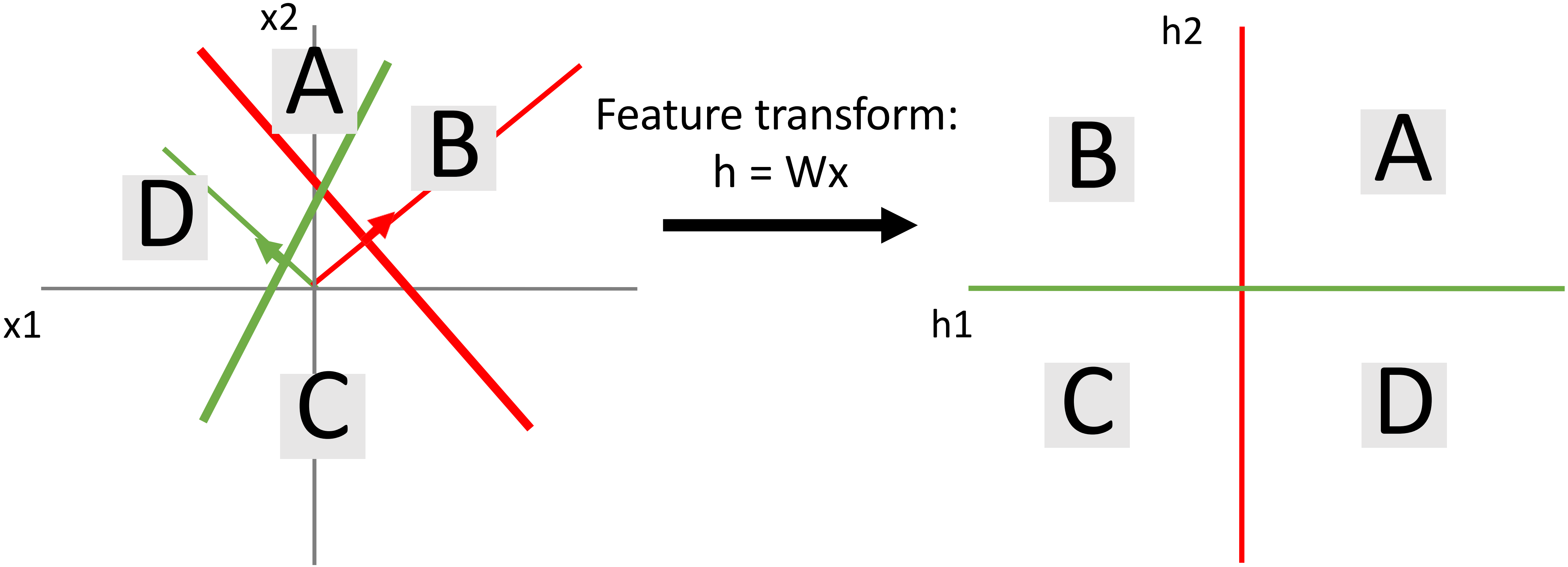


Feature transform:
h = Wx



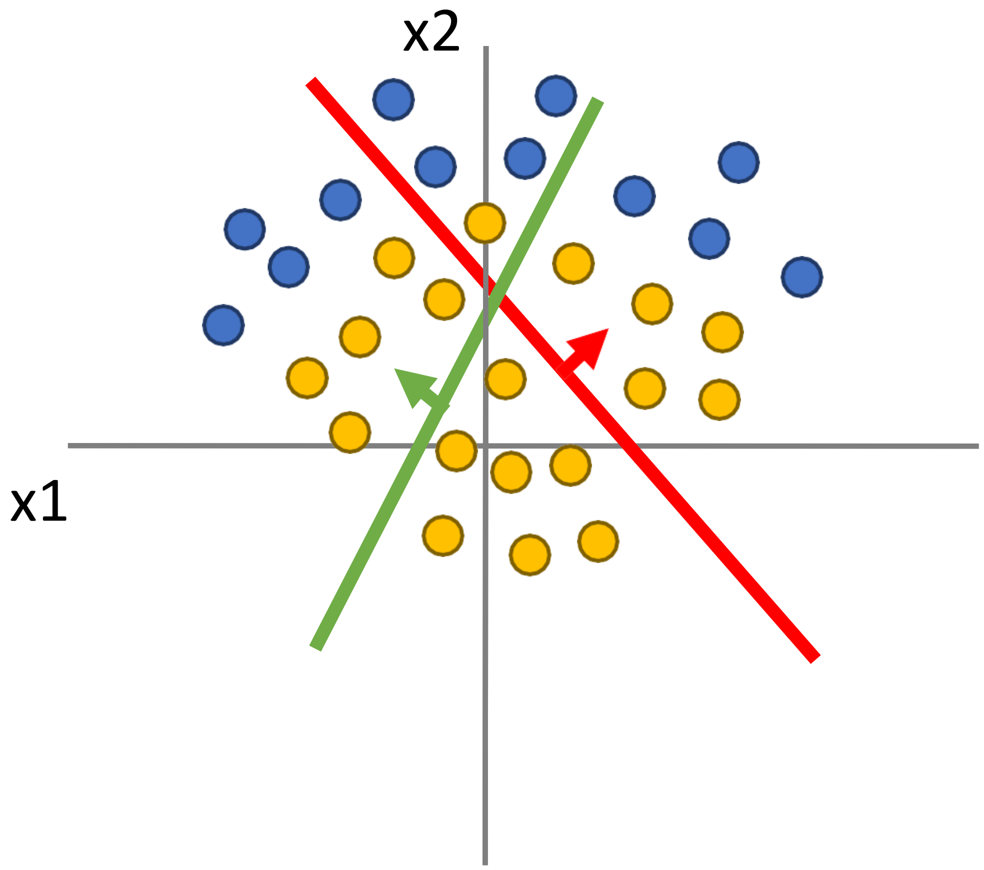
Feature Transform

Consider a linear transform: h = Wx
Where x, h are both 2-dimensional



Feature Transform

Points not linearly separable in original space

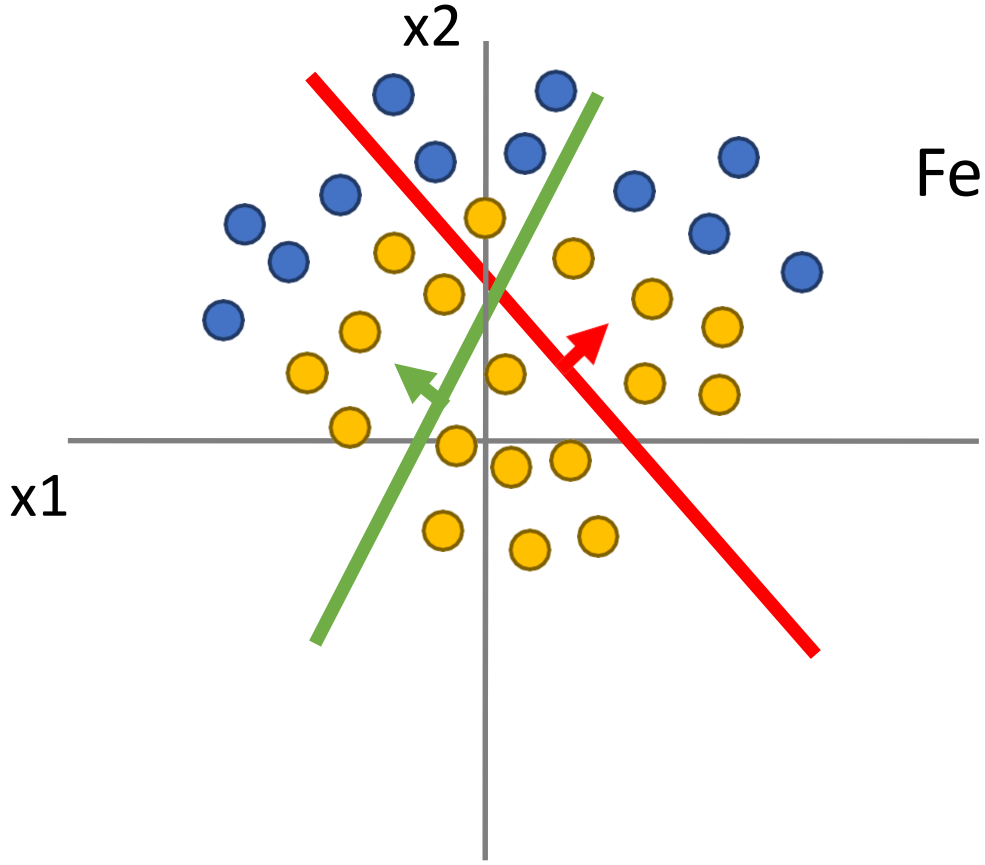


Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

Feature Transform

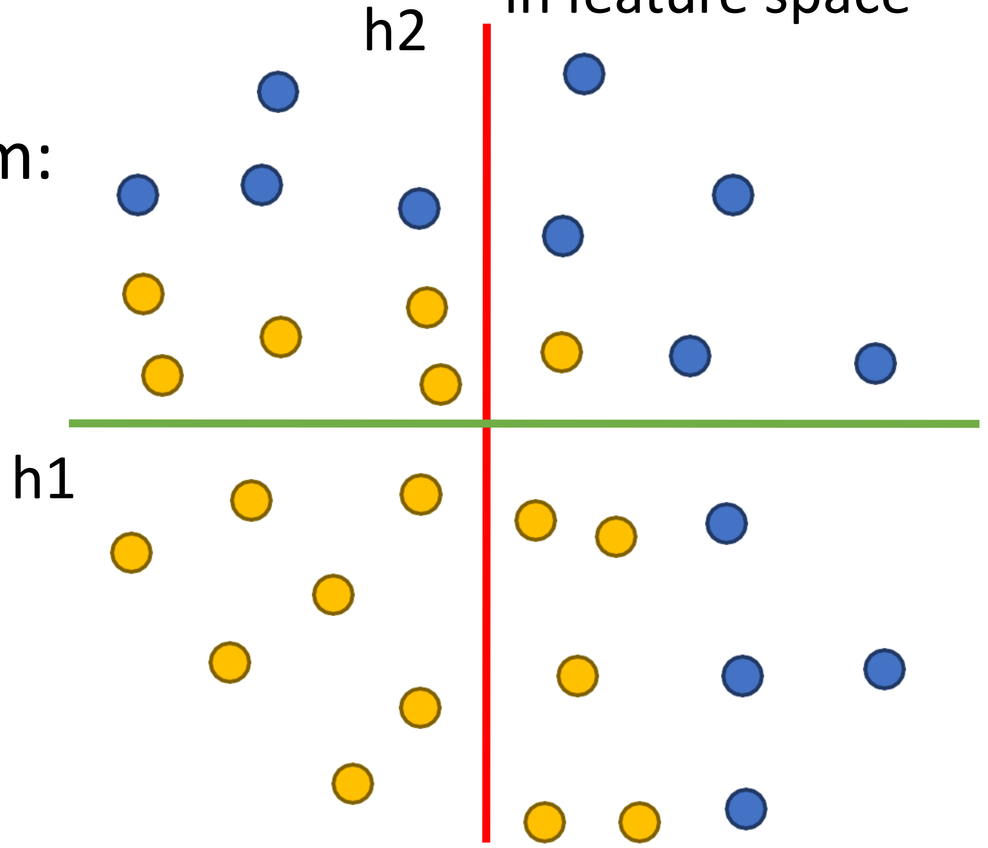
Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

Points not linearly separable in original space



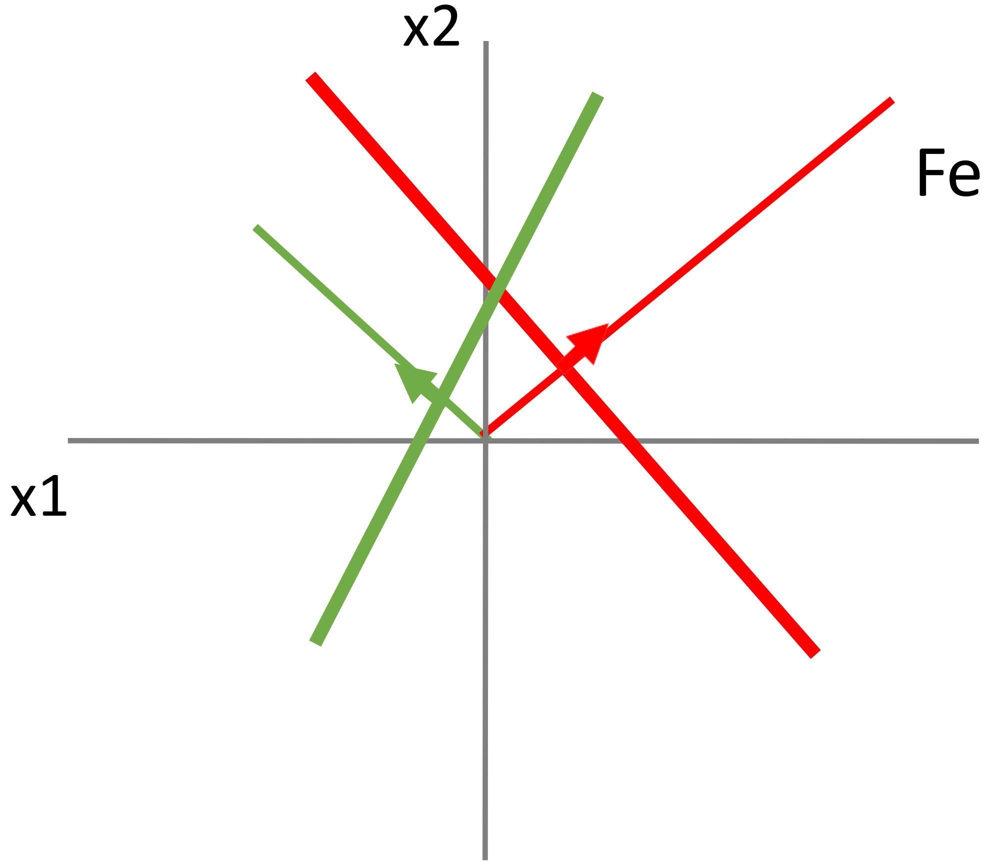
Feature transform:
h = Wx

Not linearly separable in feature space

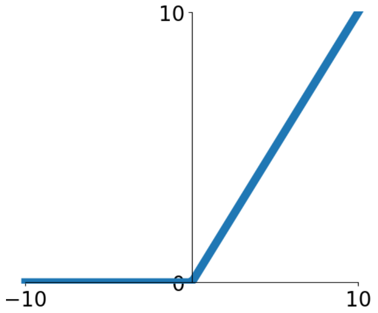


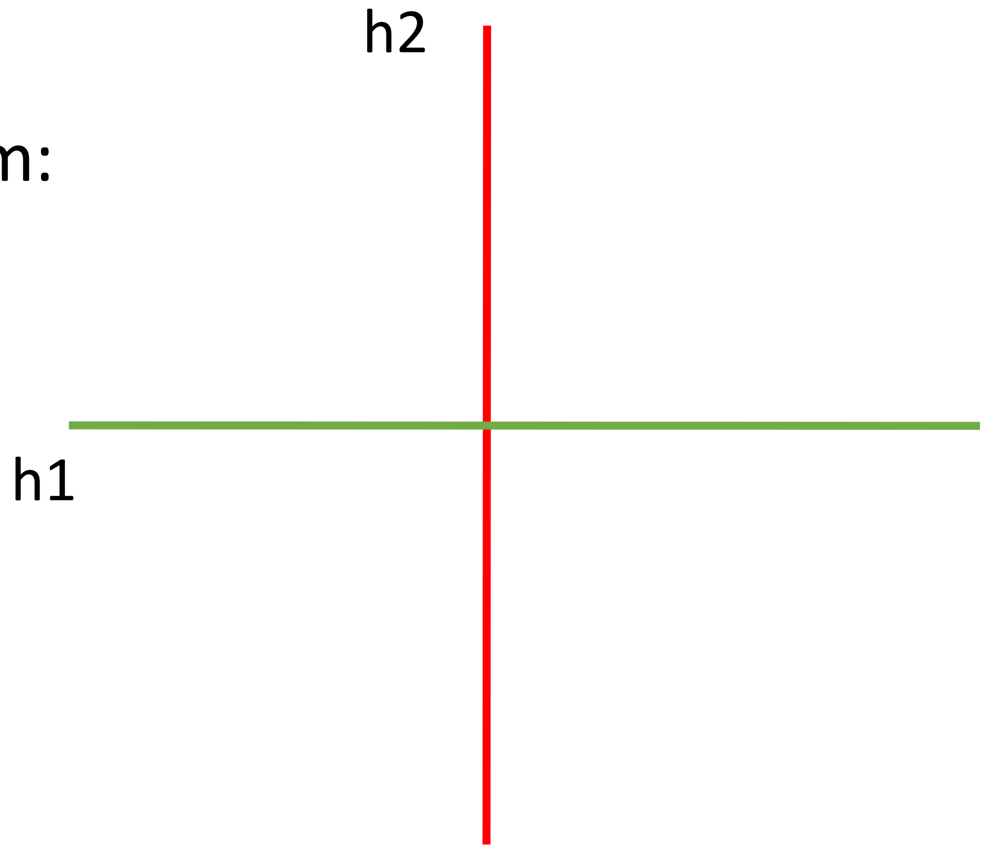
Feature Transform

Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional



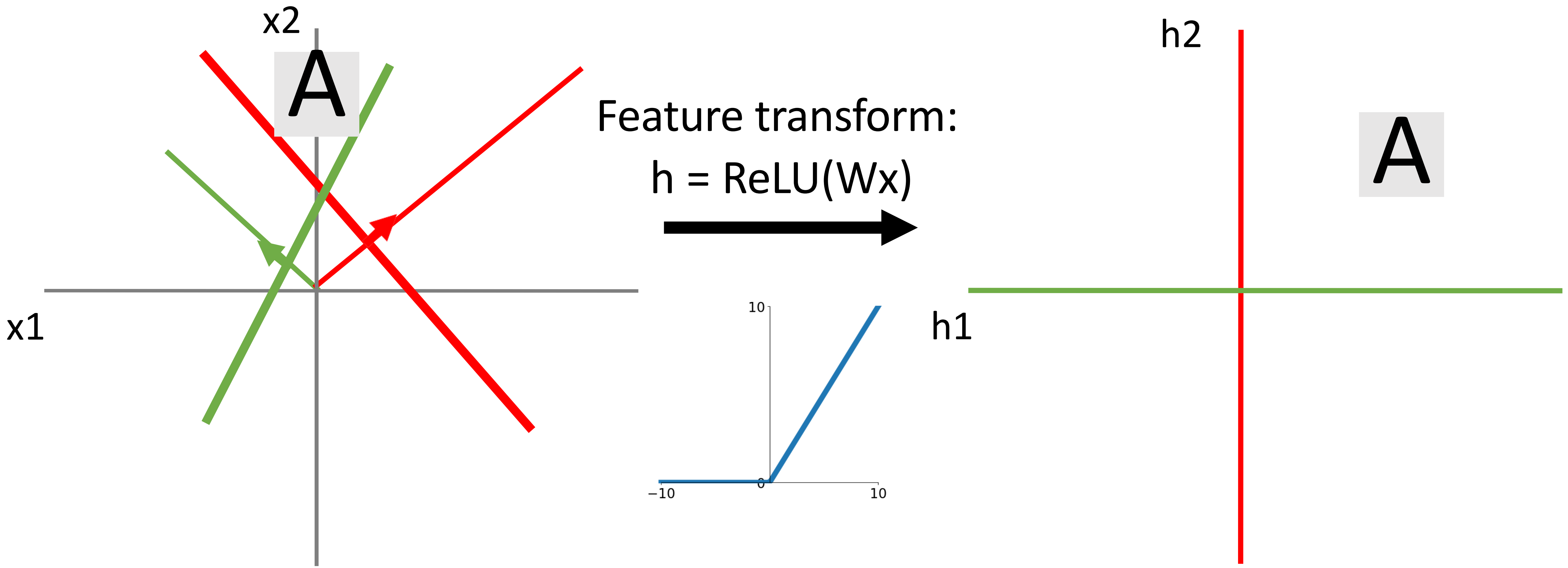
Feature transform:
h = ReLU(Wx)





Feature Transform

Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

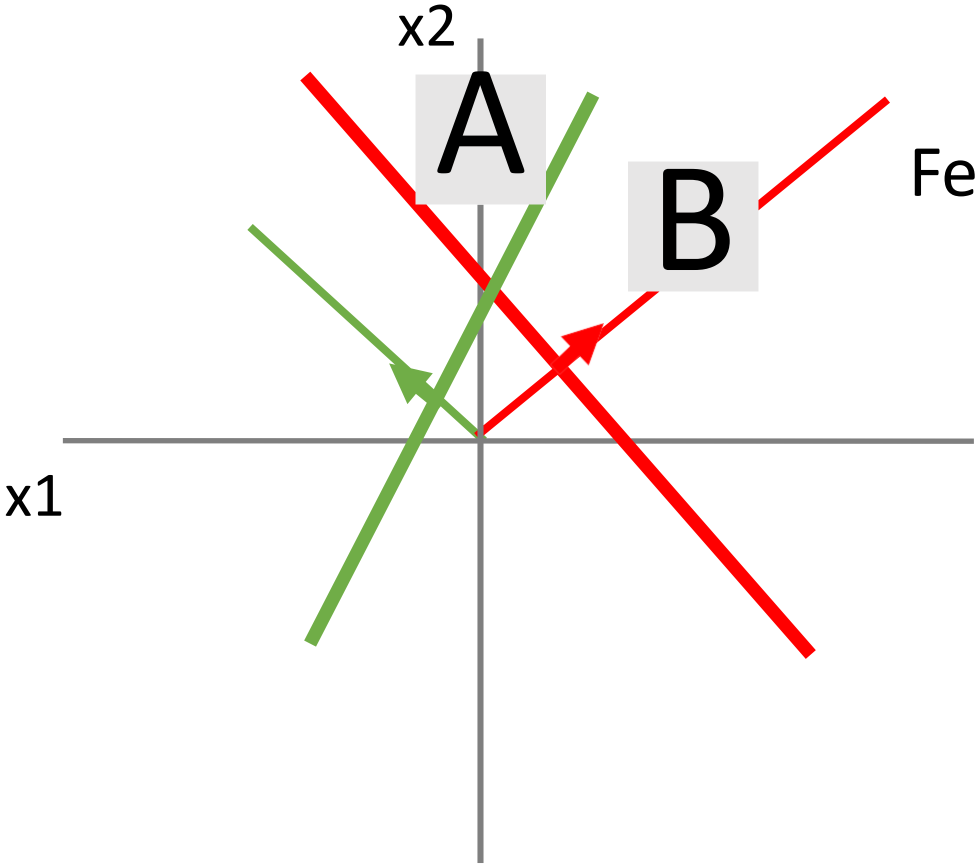


Feature Transform

Consider a neural net hidden layer:

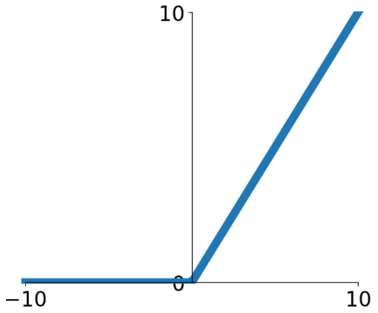
h = ReLU(Wx) = max(0, Wx)

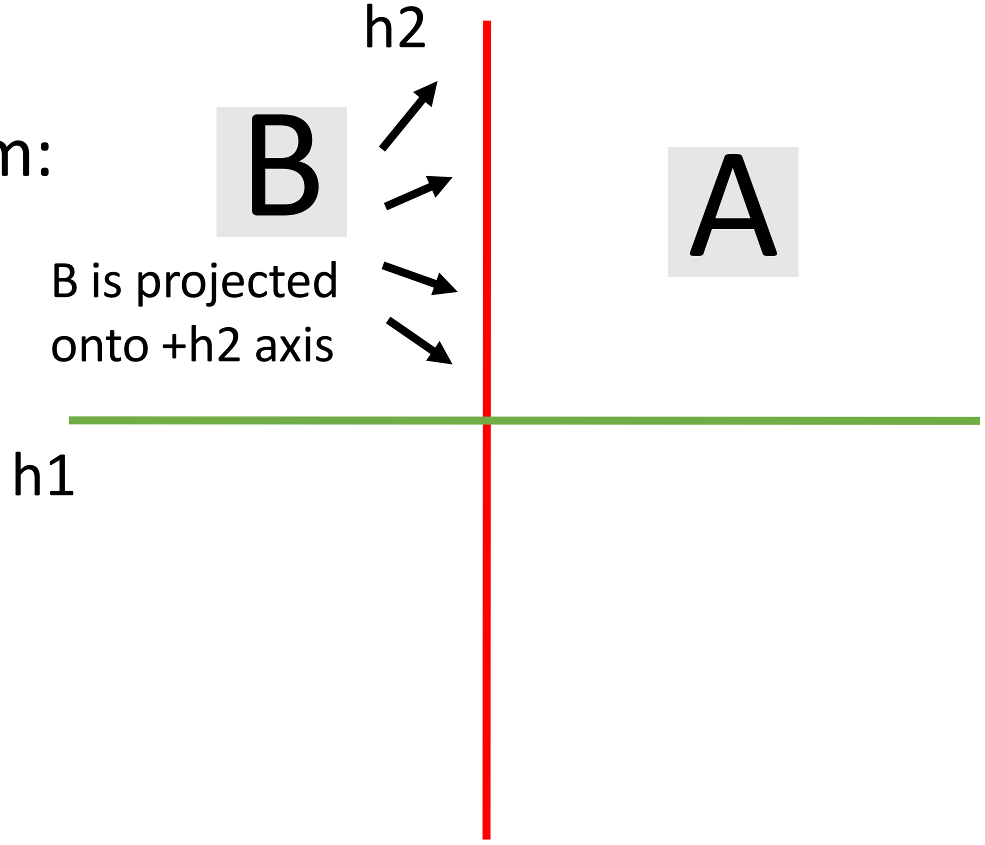
Where x, h are both 2-dimensional



Feature transform:

h = ReLU(Wx)



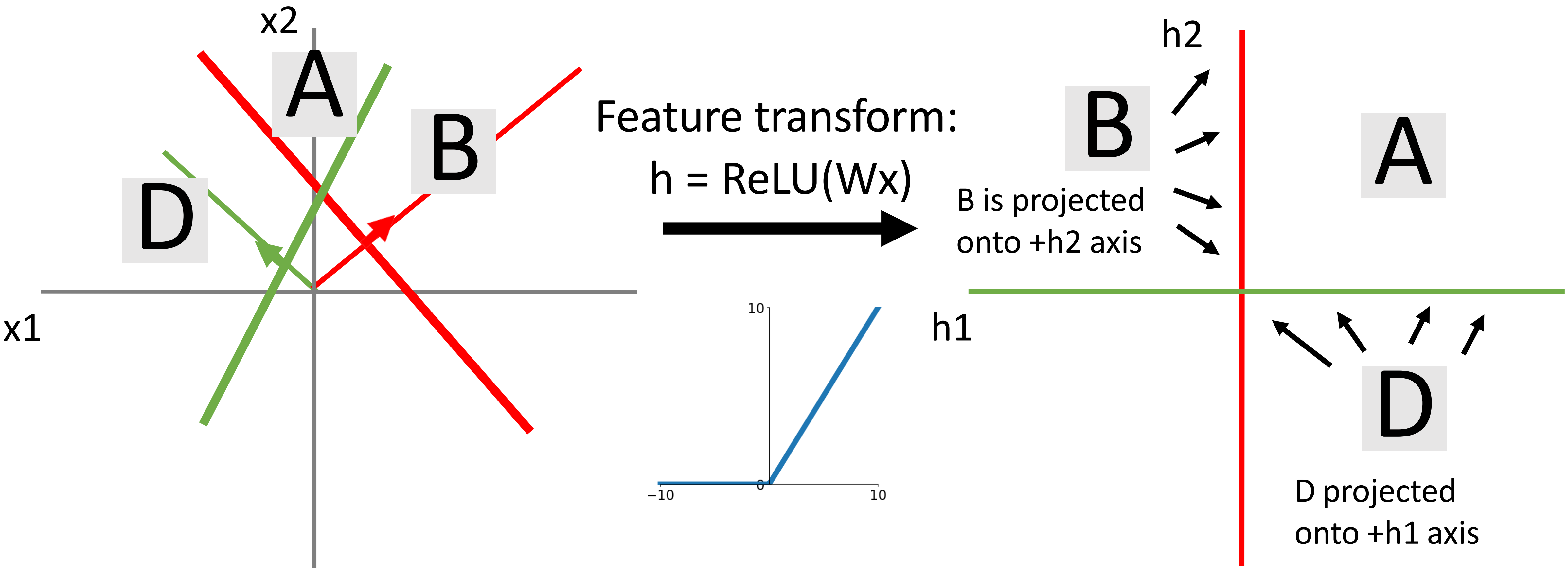


Feature Transform

Consider a neural net hidden layer:

h = ReLU(Wx) = max(0, Wx)

Where x, h are both 2-dimensional

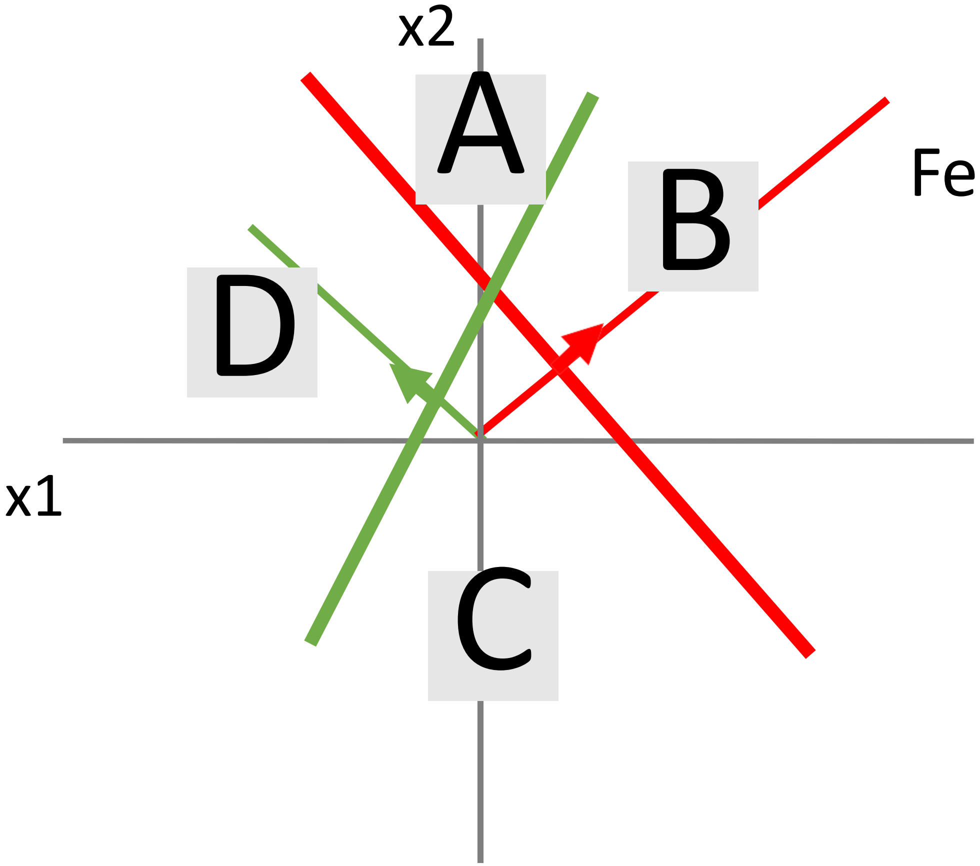


Feature Transform

Consider a neural net hidden layer:

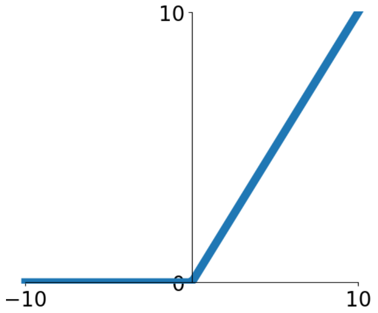
$h = \text{ReLU}(Wx) = \max(0, Wx)$

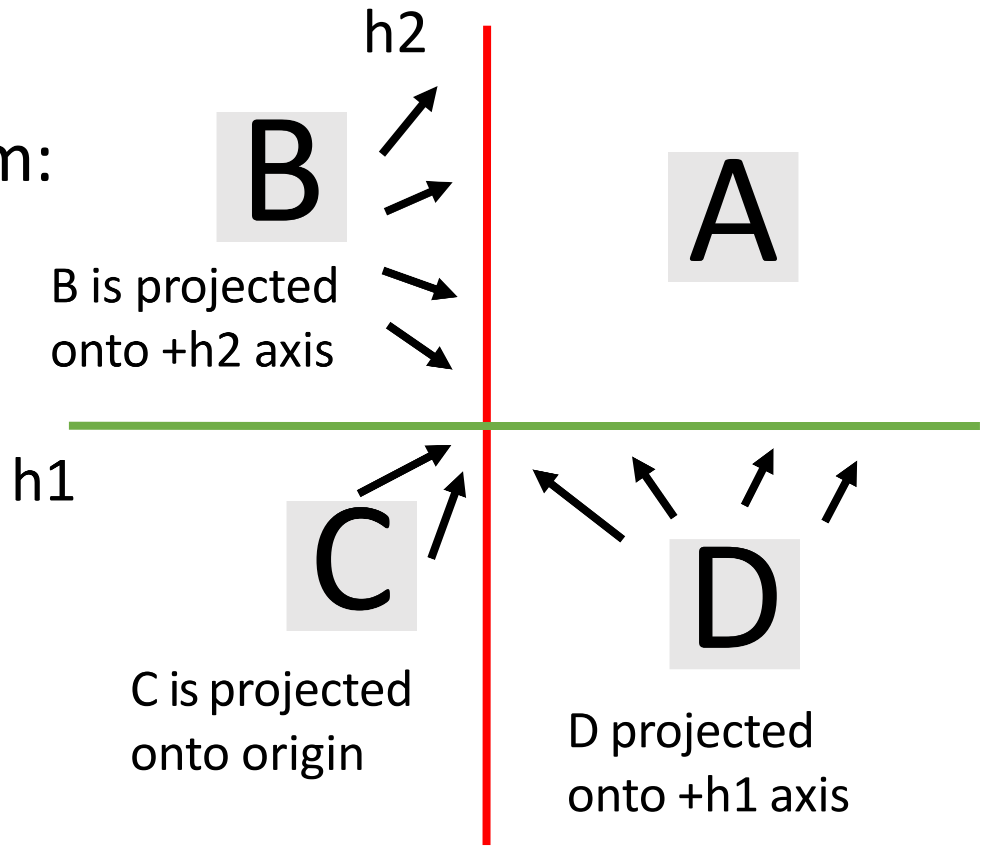
Where x, h are both 2-dimensional



Feature transform:

$h = \text{ReLU}(Wx)$





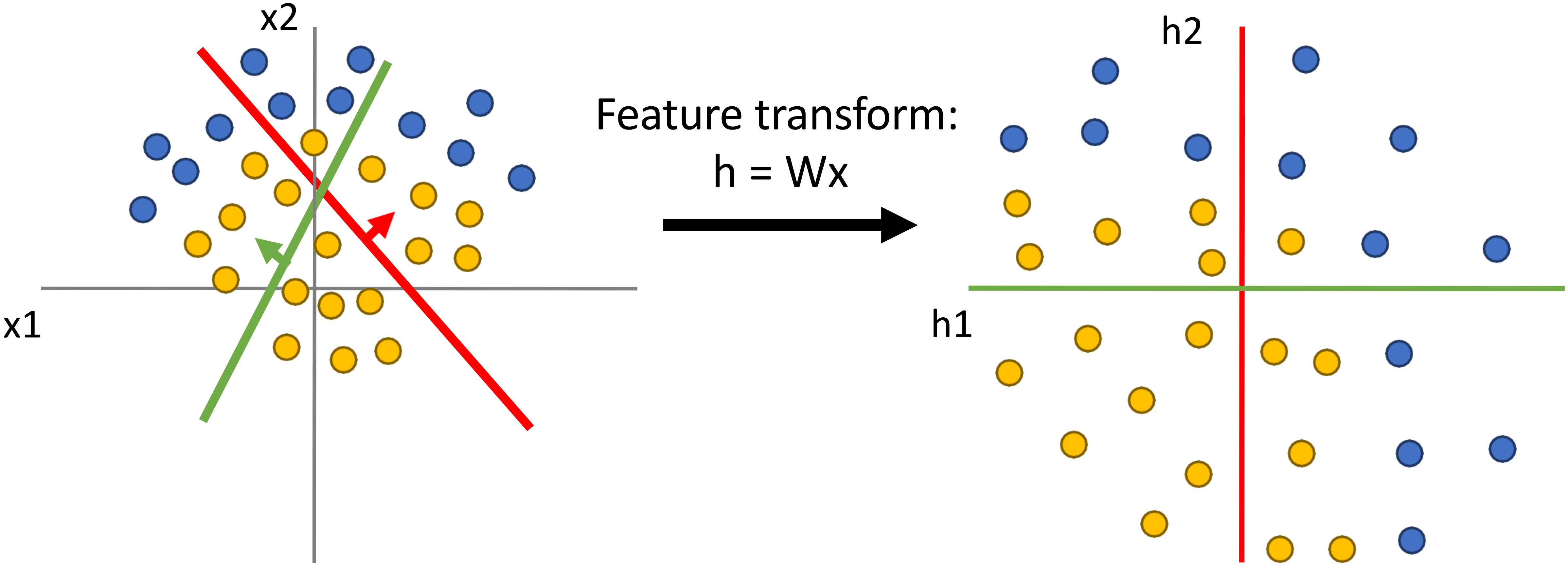
Feature Transform

Points not linearly separable in original space

Consider a neural net hidden layer:

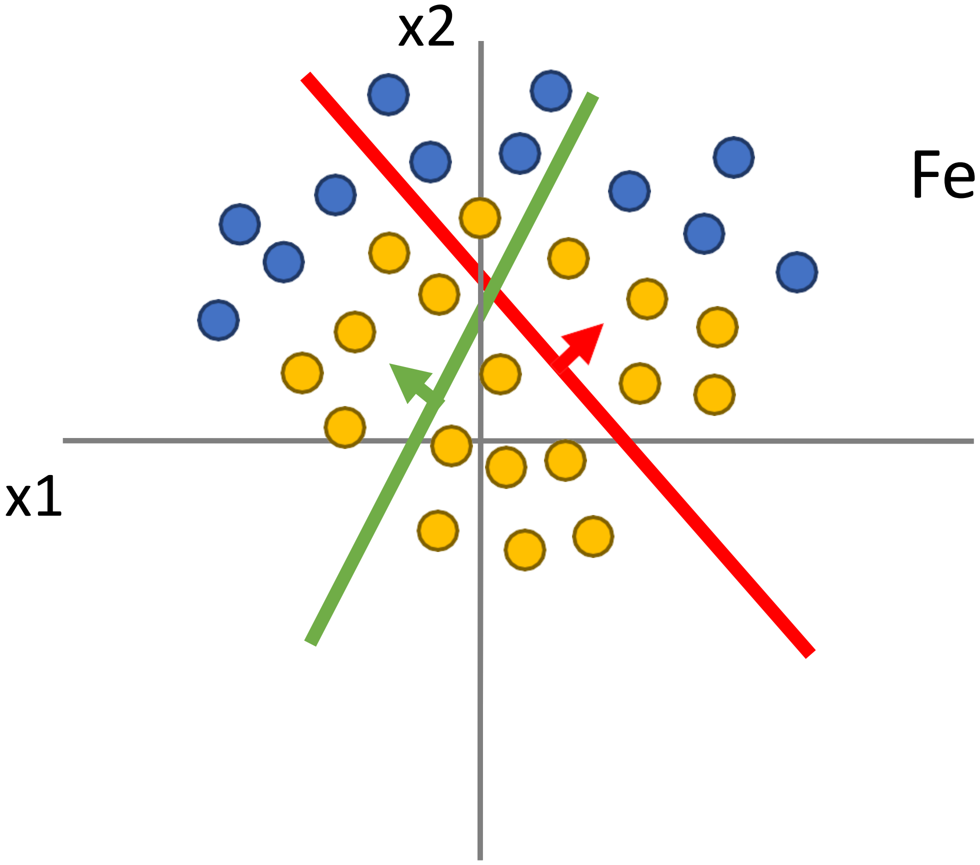
$h = \text{ReLU}(Wx) = \max(0, Wx)$

Where x, h are both 2-dimensional



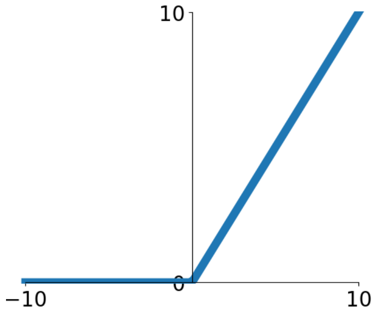
Feature Transform

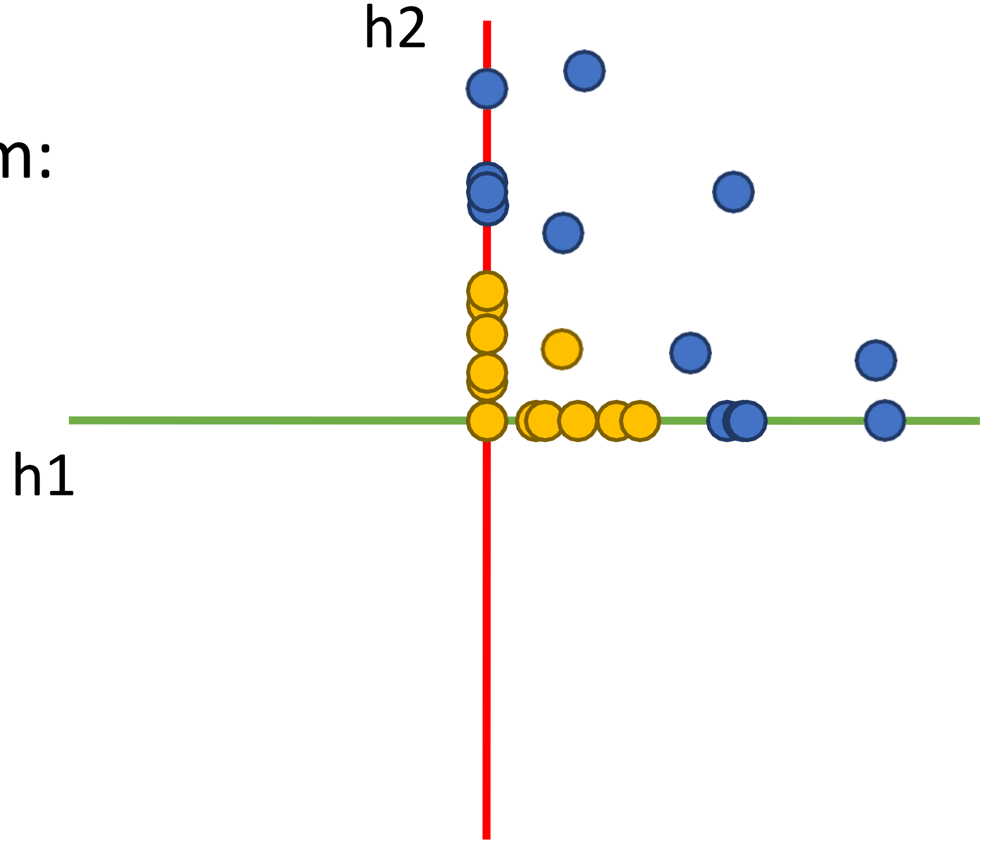
Points not linearly separable in original space



Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

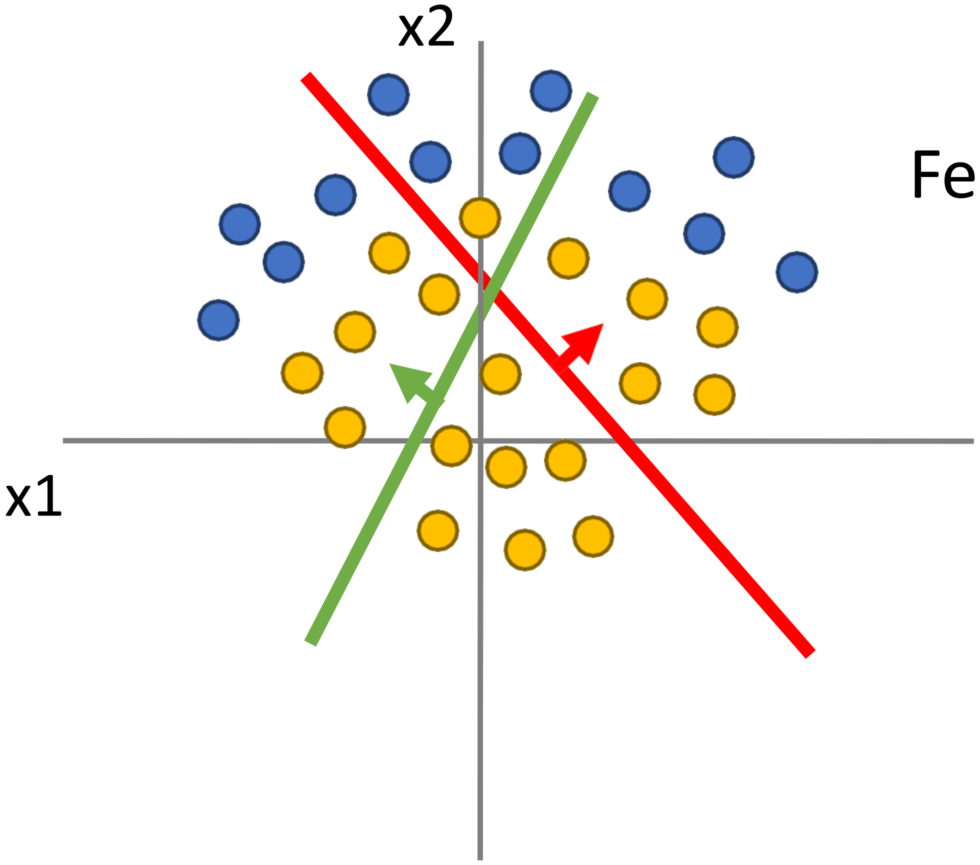
Feature transform:
h = ReLU(Wx)





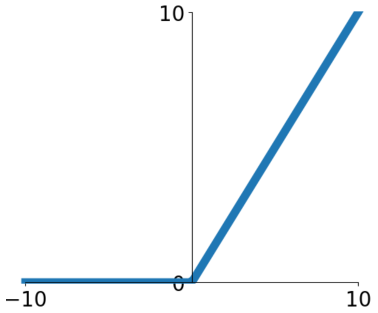
Feature Transform

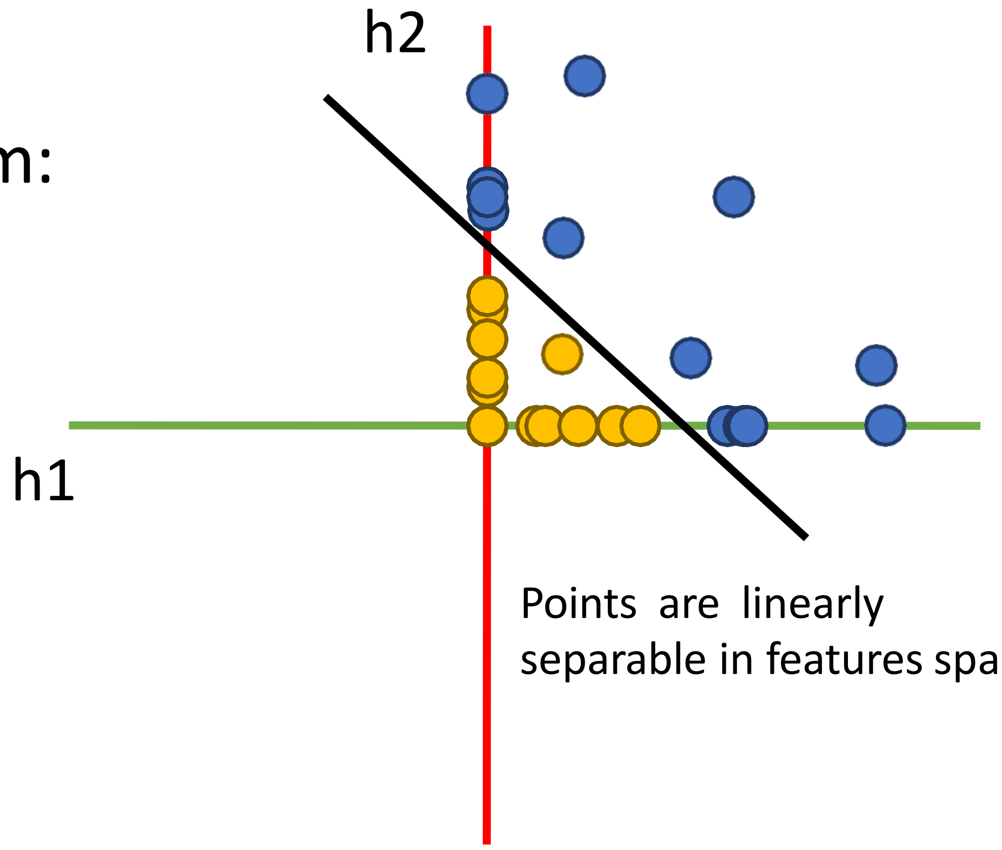
Points not linearly separable in original space



Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

Feature transform:
h = ReLU(Wx)





Points are linearly separable in features space!

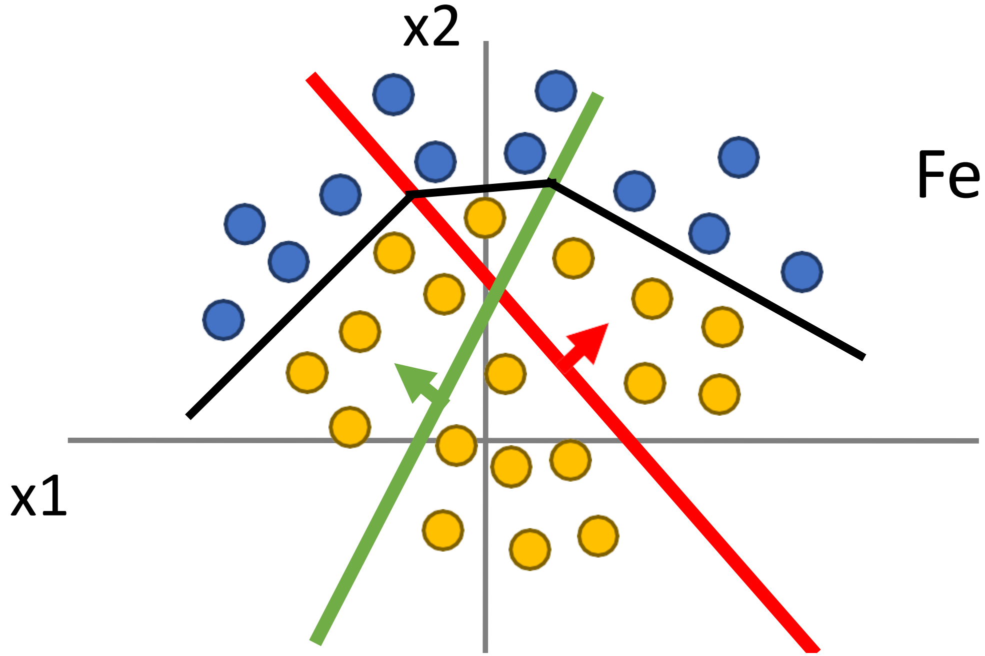
Feature Transform

Consider a neural net hidden layer:

h = ReLU(Wx) = max(0, Wx)

Where x, h are both 2-dimensional

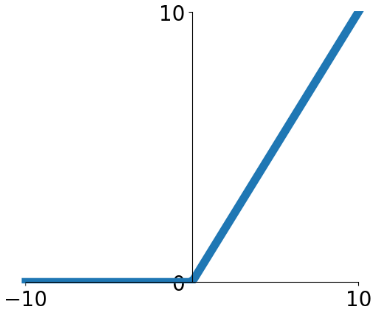
Points not linearly separable in original space

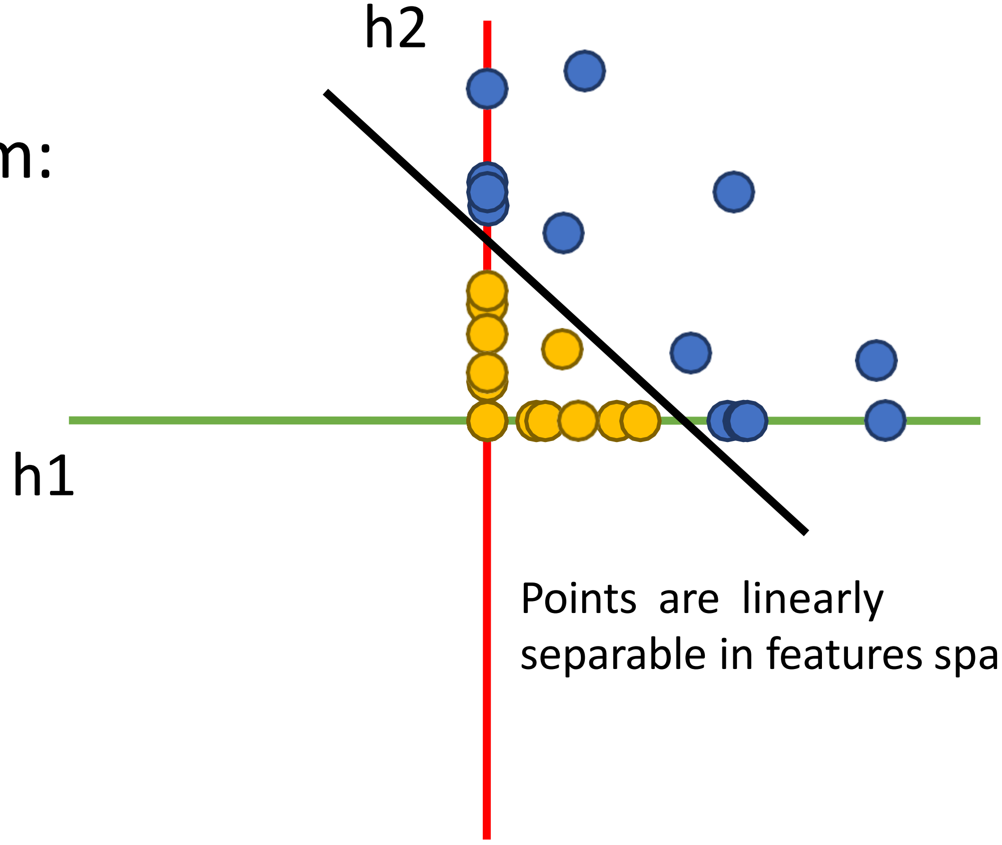


Linear classifier in feature space gives nonlinear classifier in original space

Feature transform:

h = ReLU(Wx)

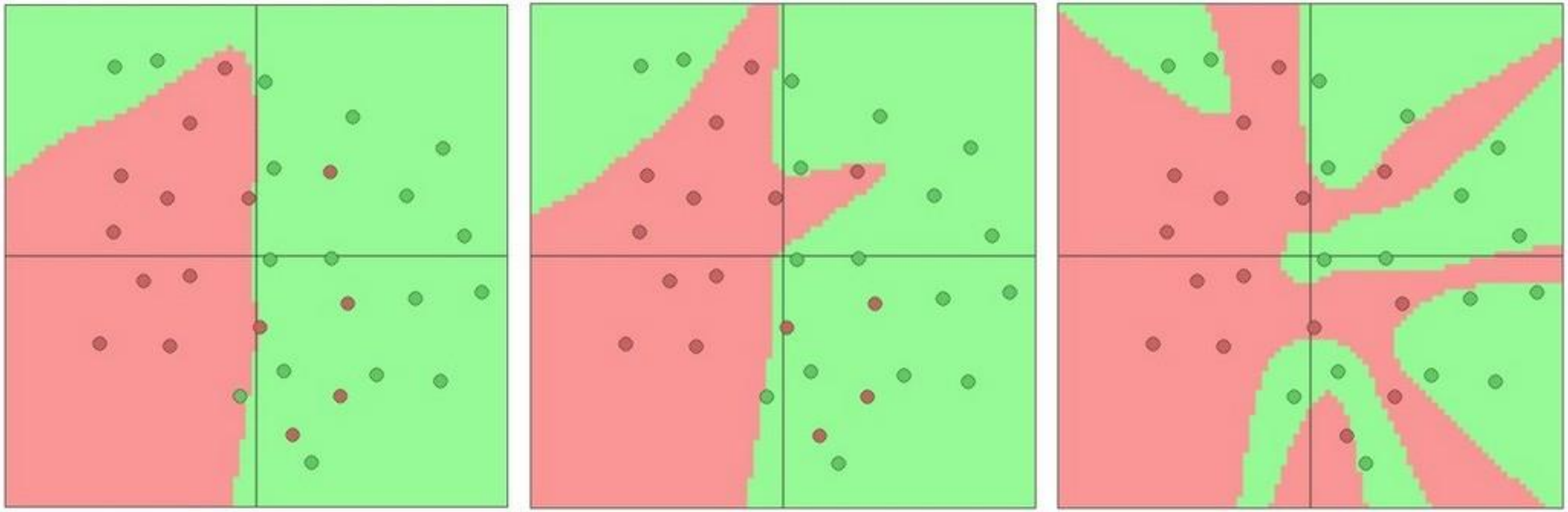




Points are linearly separable in features space!

Setting the number of layers and their sizes

3 hidden units | 6 hidden units | 20 hidden units



More hidden units = more capacity

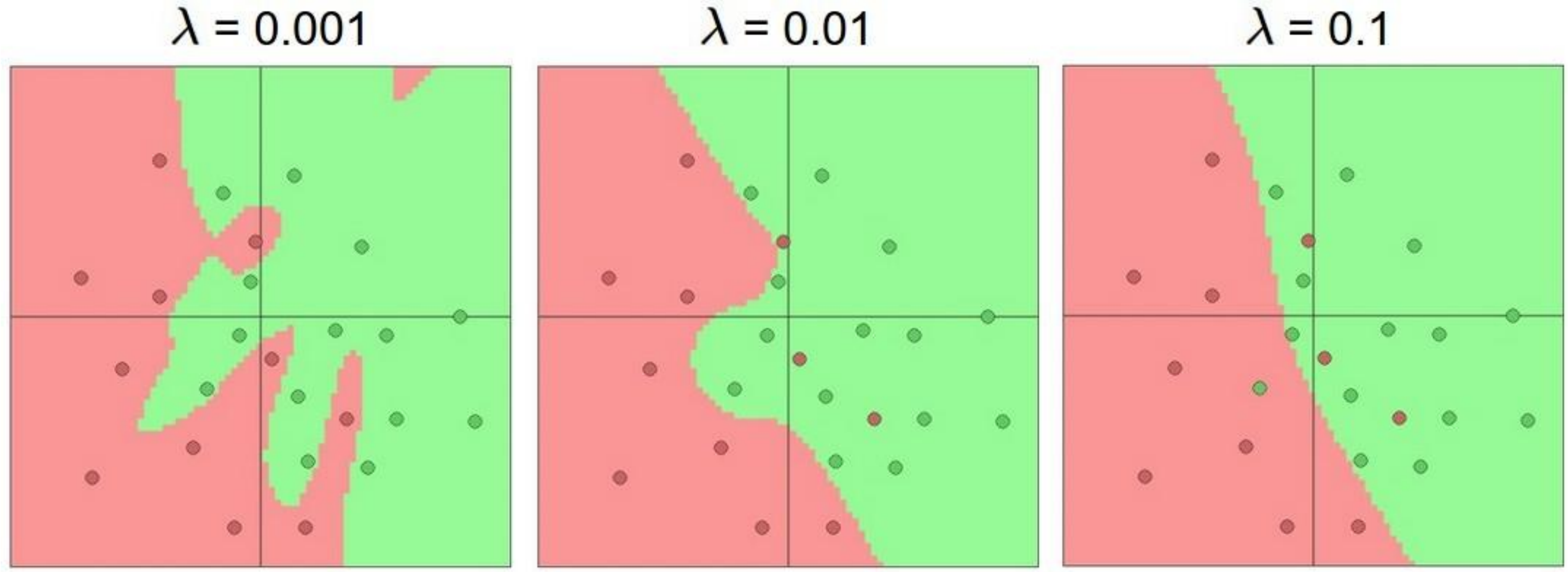
Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

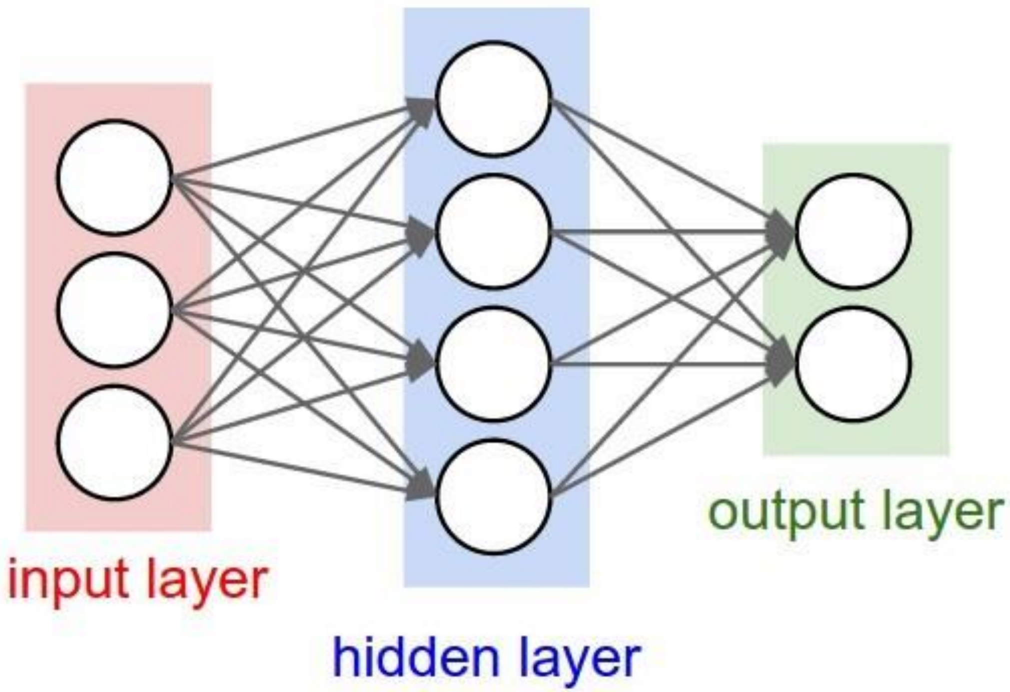
Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Regularization with constant number of layers



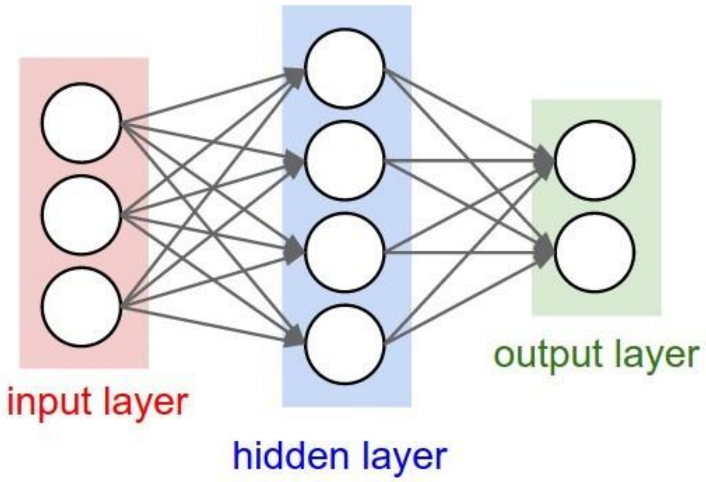
Neural Net sample code




```
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

Neural Net sample code

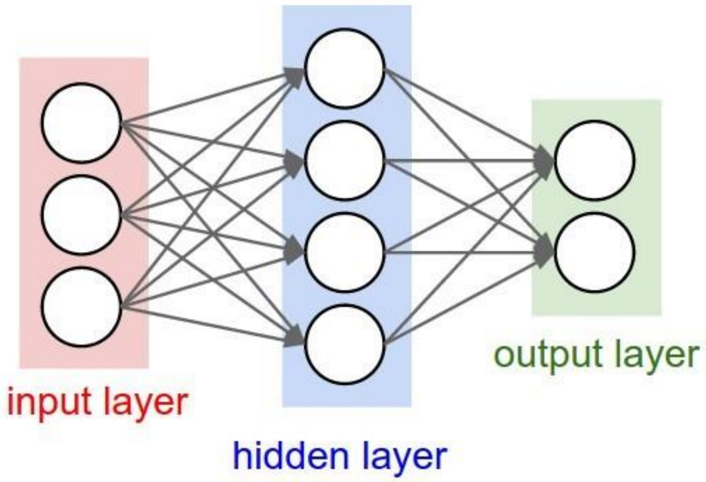


Initialize weights and data

```
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

Neural Net sample code



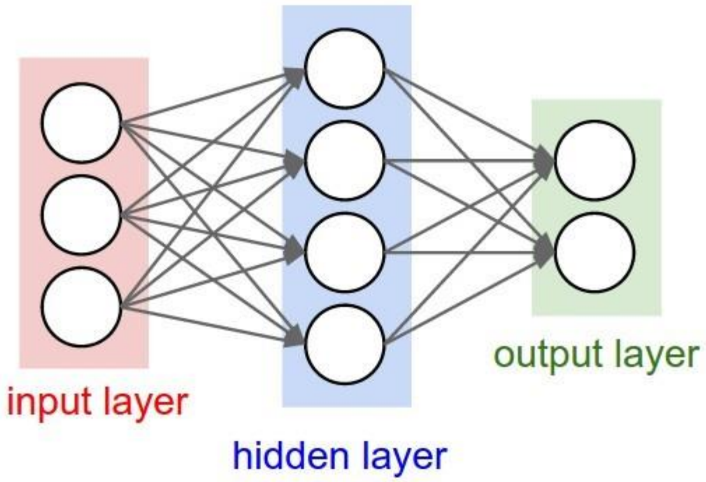
Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

```
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

Neural Net sample code



Initialize weights and data

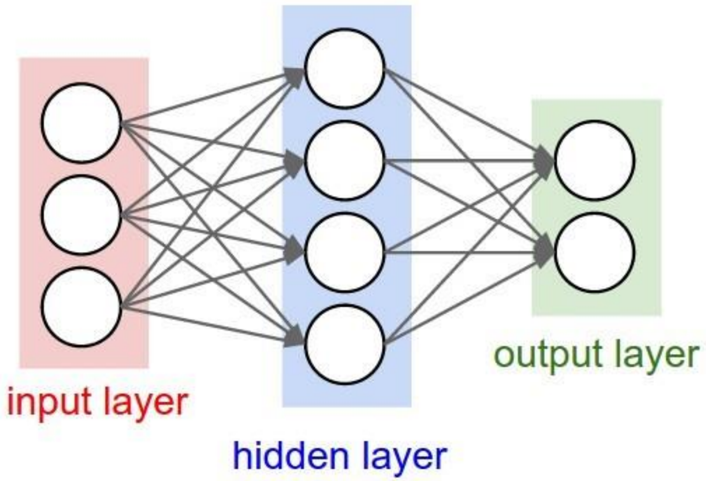
Compute loss (sigmoid activation, L2 loss)

Compute gradients

```
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

Neural Net sample code



Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients

SGD step

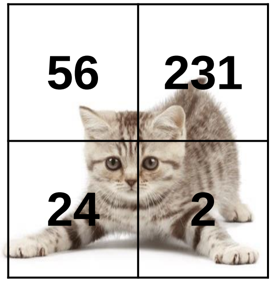
```python
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

Spatial Information

$$f = W_2 \max(0, W_1 x)$$

Flatten lattice into vector

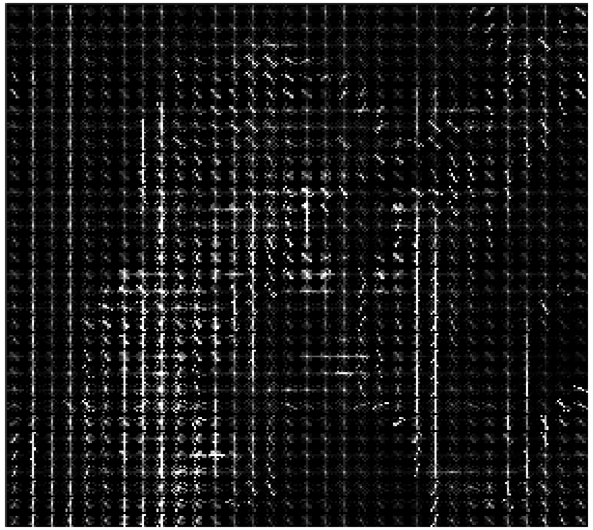


Input image
(2, 2)

56
231
24
2

(4,)

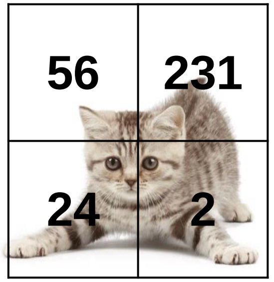
Histogram of Oriented Gradients



Spatial Information

$$f = W_2 \max(0, W_1 x)$$

Flatten lattice into vector



Input image
(2, 2)

Problem: So far our neural networks don't respect the spatial structure of images

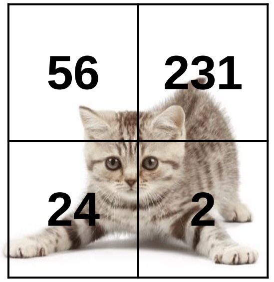
56
231
24
2

(4,)

Spatial Information

$$f = W_2 \max(0, W_1 x)$$

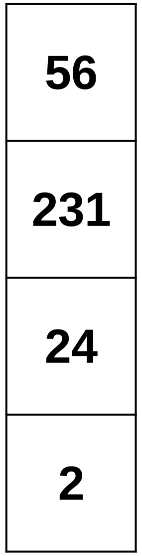
Flatten lattice into vector



Input image
(2, 2)

Problem: So far our neural networks don’t respect the spatial structure of images

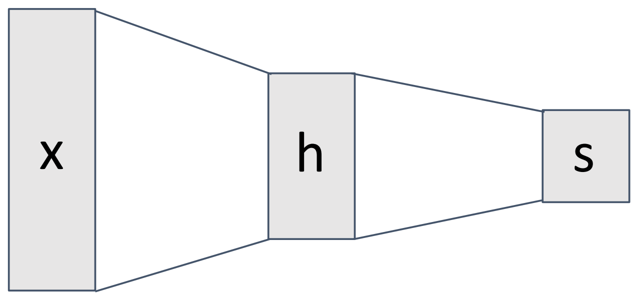
Solution: Define new computational operators



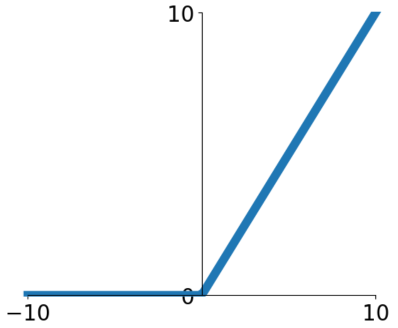
(4,)

Components of a Fully-Connected Network

Fully-Connected Layers

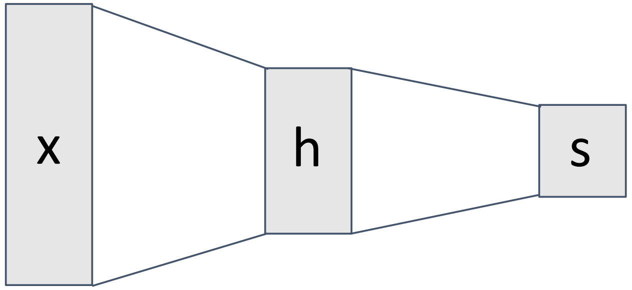


Activation Function

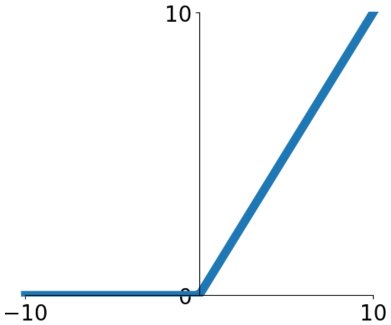


Components of a Convolutional Network

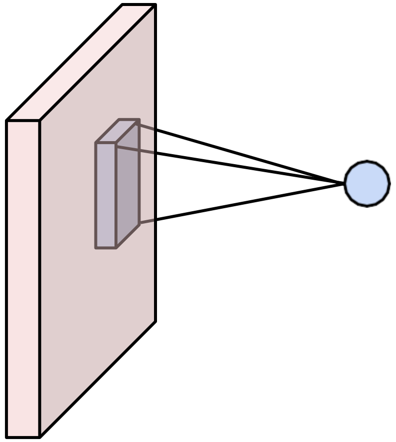
Fully-Connected Layers



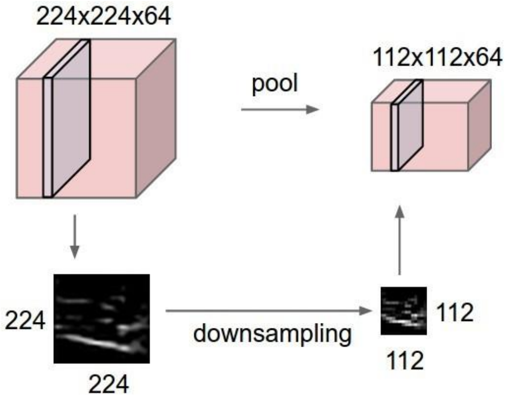
Activation Function



Convolution Layers



Pooling Layers

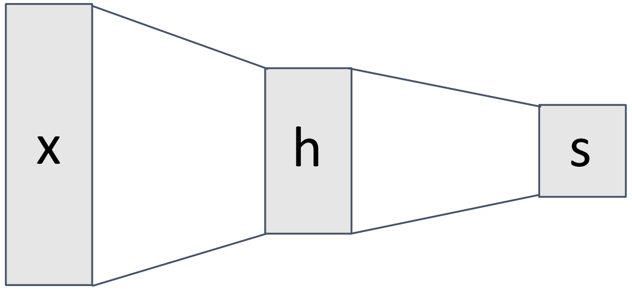


Normalization

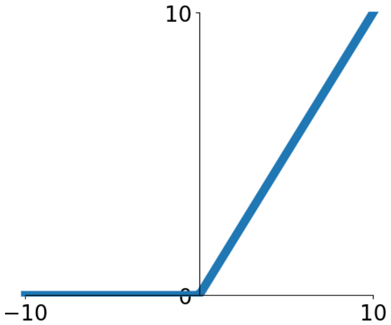
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Components of a Convolutional Network

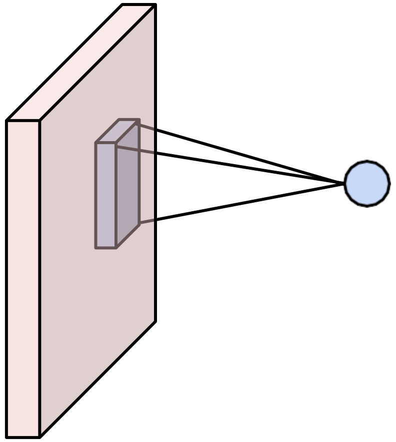
Fully-Connected Layers



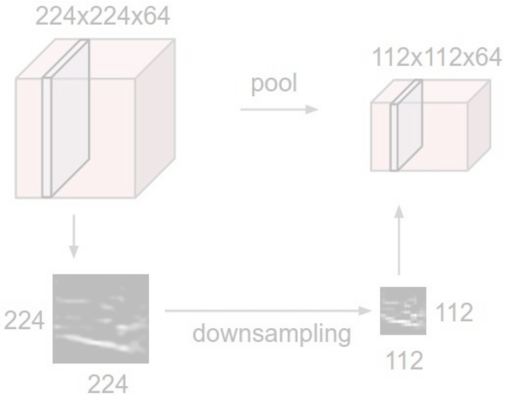
Activation Function



Convolution Layers



Pooling Layers

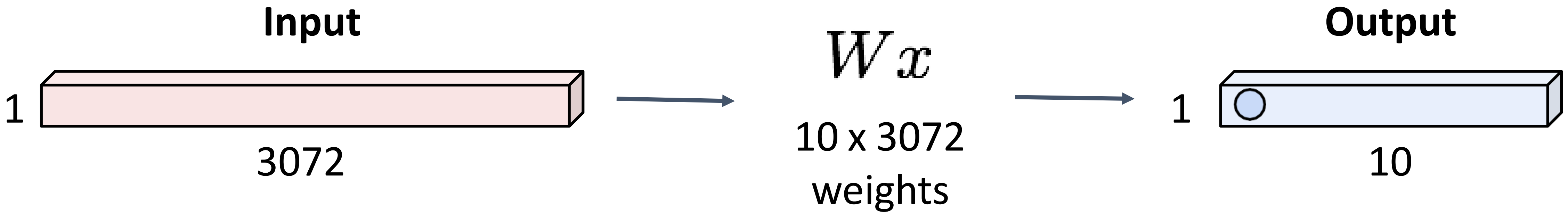


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

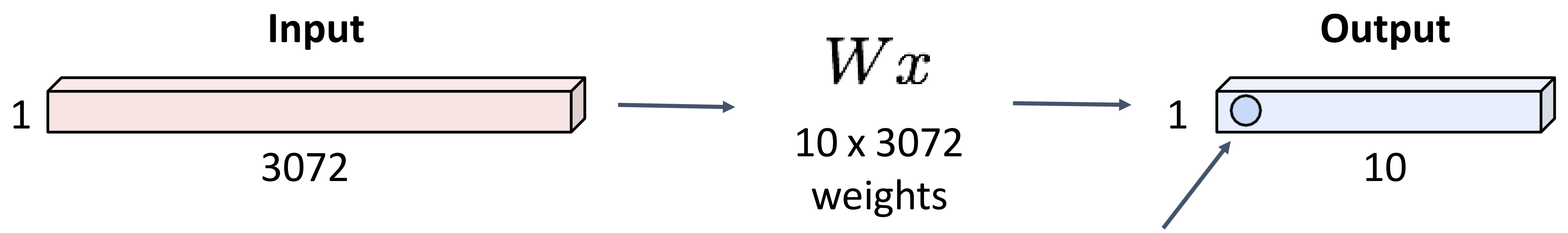
Fully-Connected Layer

32x32x3 image -> flatten to 3072 x 1



Fully-Connected Layer

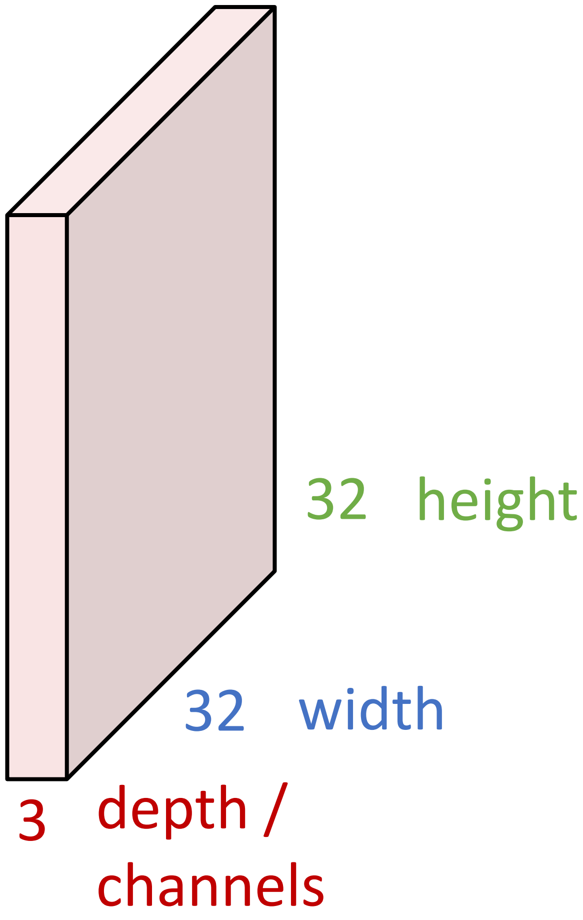
32x32x3 image -> flatten to 3072 x 1



1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

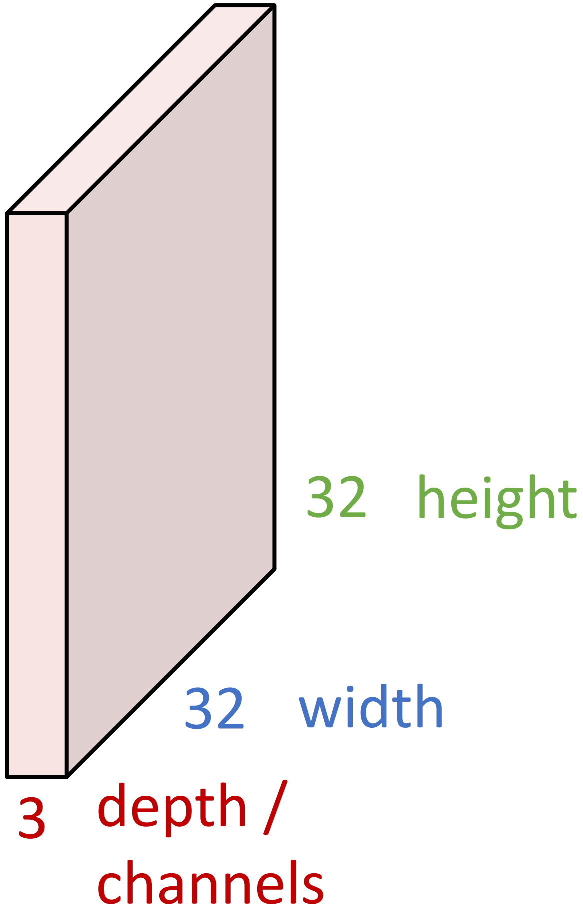
Convolution Layer

3x32x32 image: preserve structure

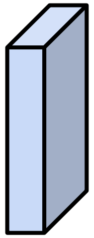


Convolution Layer

3x32x32 image



3x5x5 filter

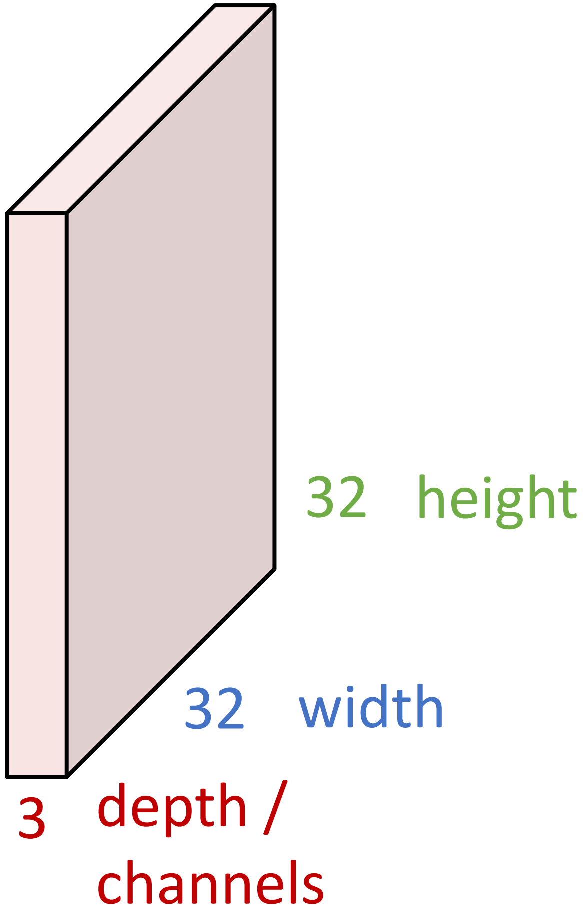


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

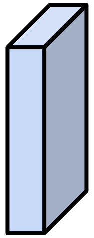
Convolution Layer

Filters always extend the full depth of the input volume

3x32x32 image

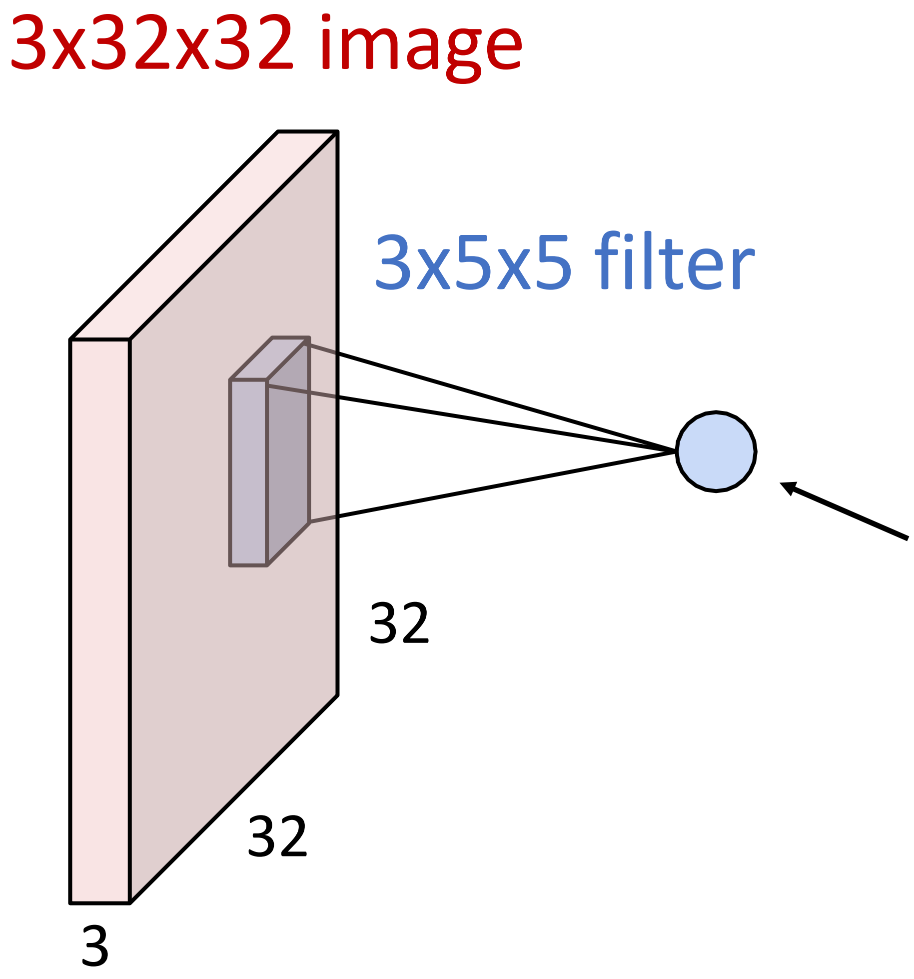


3x5x5 filter



Convolve the filter with the image i.e. "slide over the image spatially, computing dot products"

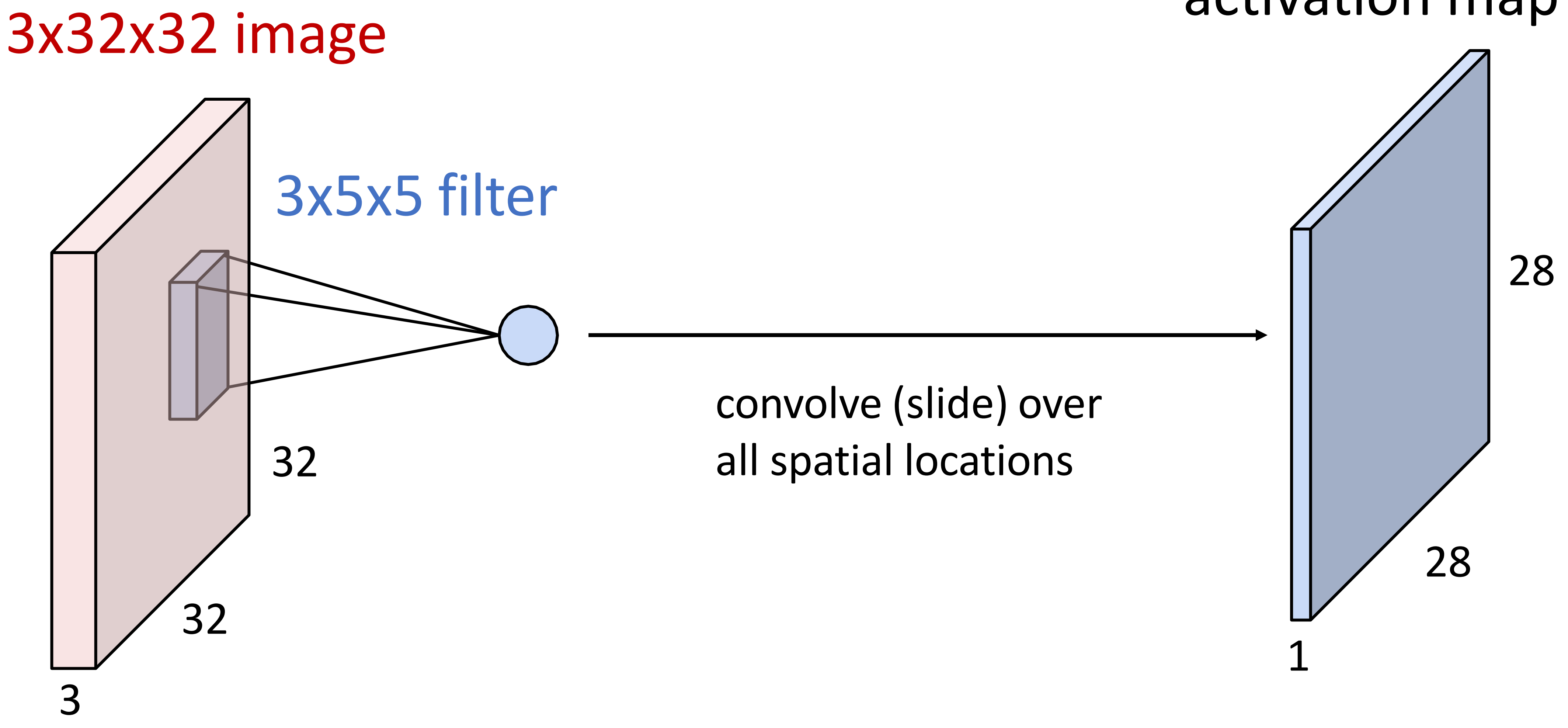
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image
(i.e. 3*5*5 = 75-dimensional dot product + bias)

$$w^T x + b$$

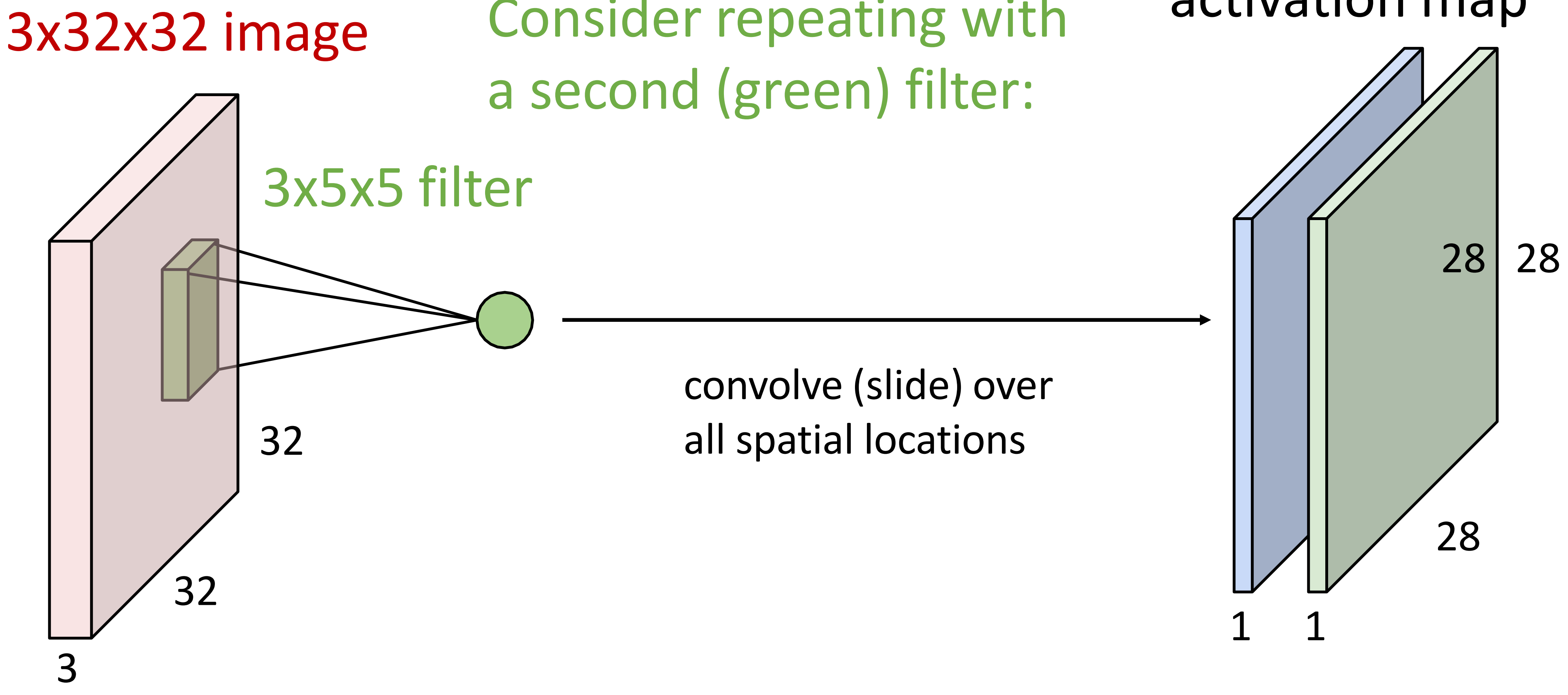
Convolution Layer



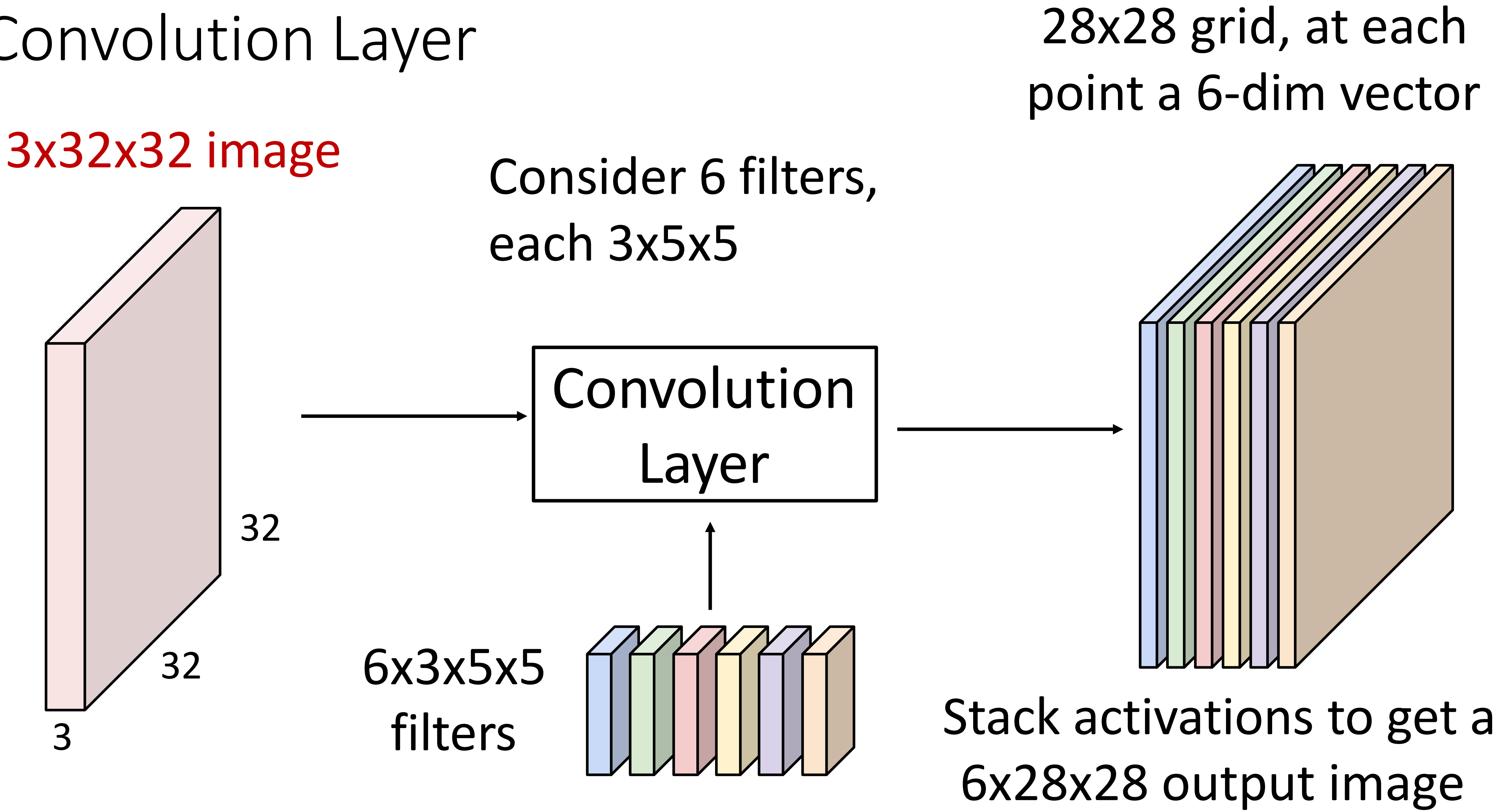
Convolution Layer

two 1x28x28 activation map

Consider repeating with a second (green) filter:

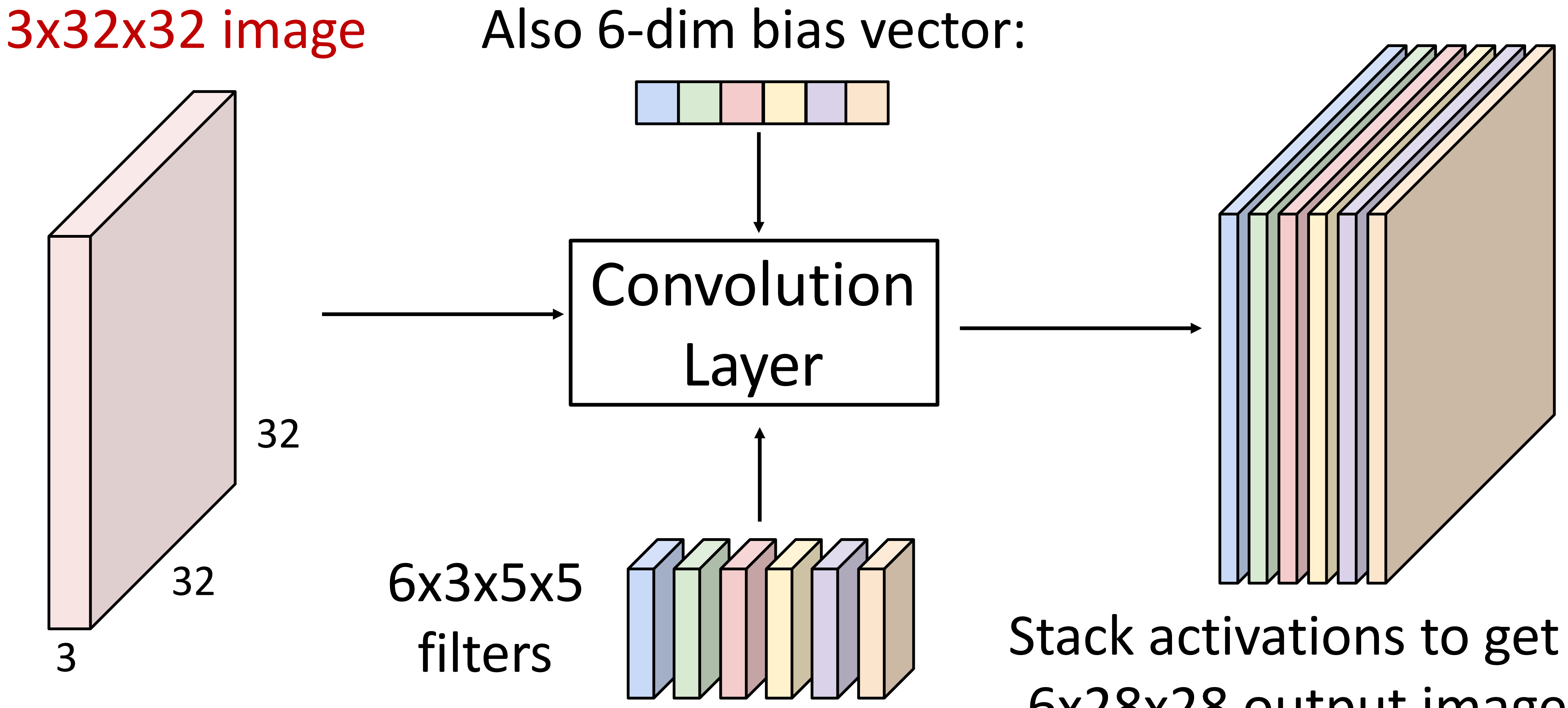


Convolution Layer



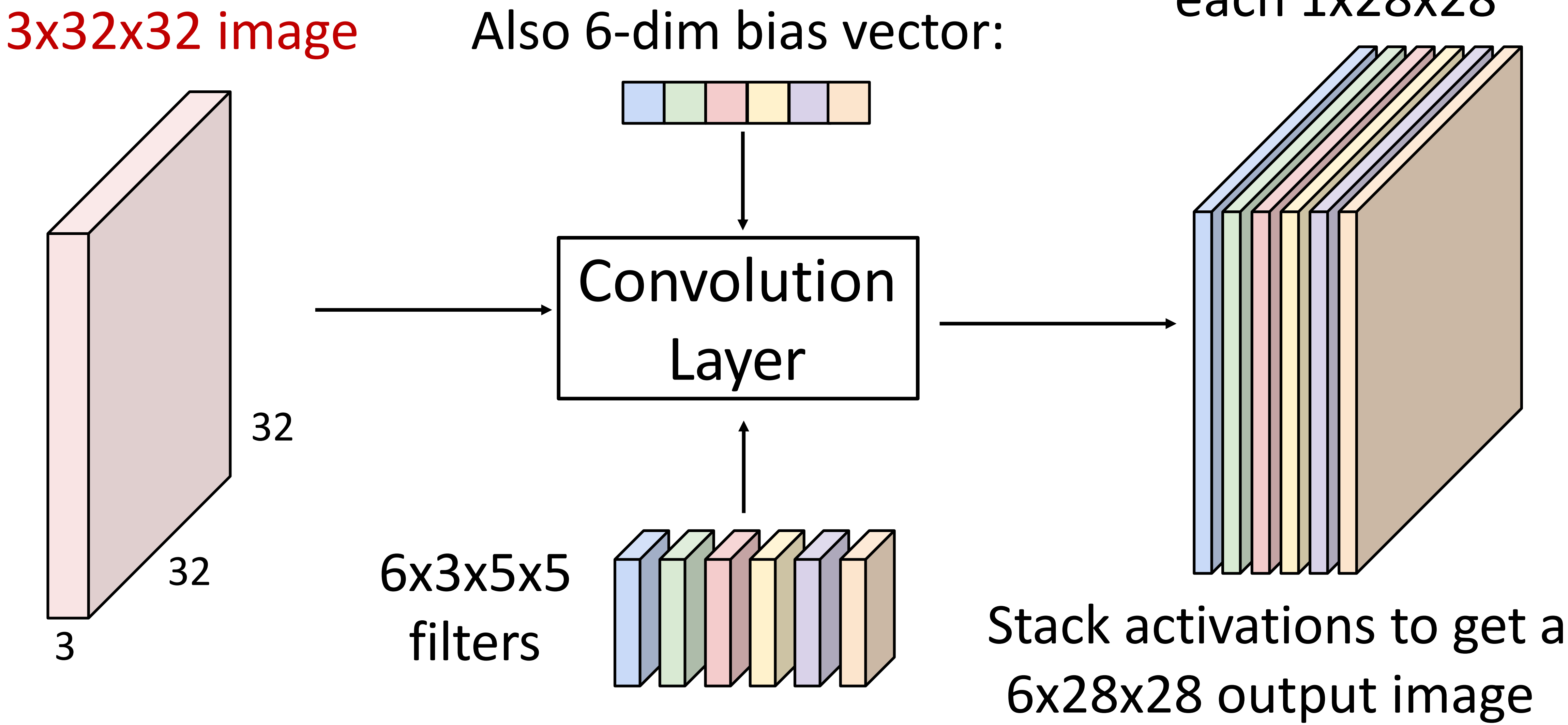
Convolution Layer

28x28 grid, at each point a 6-dim vector

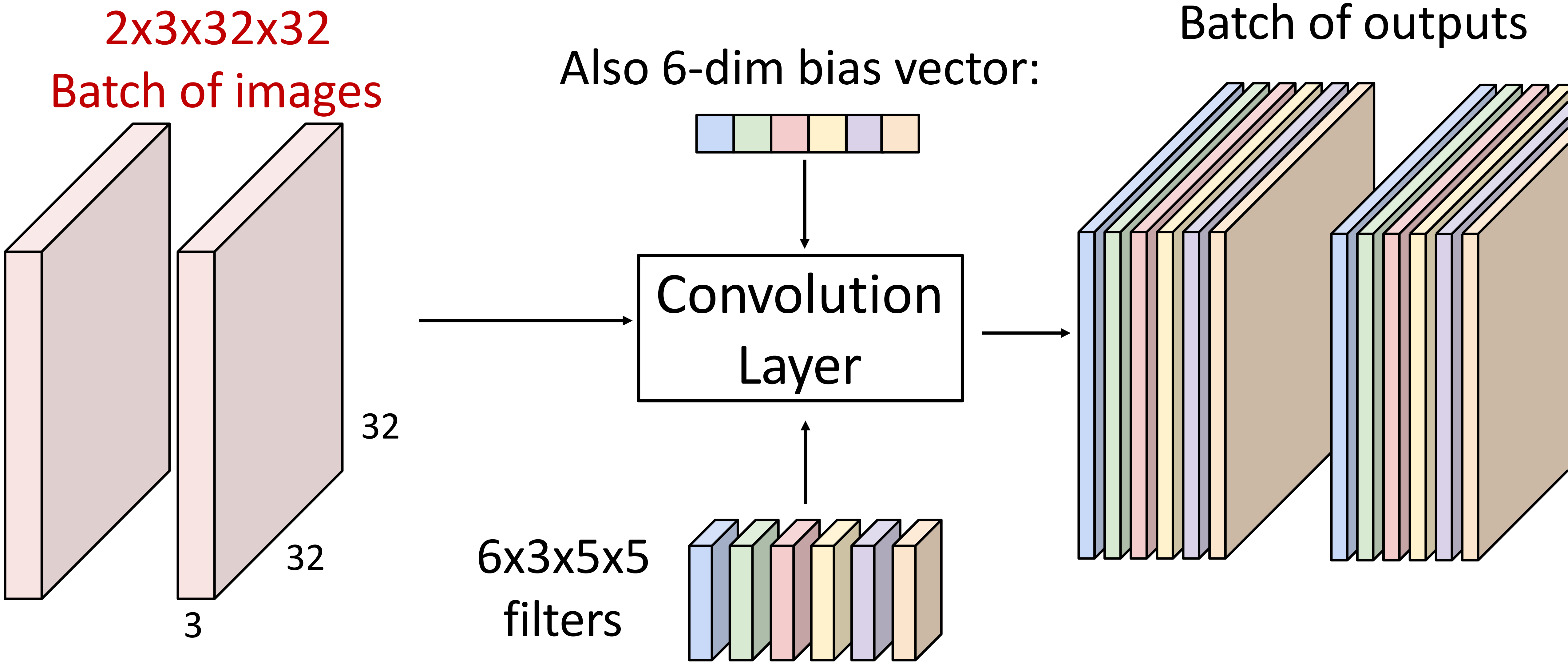


Stack activations to get a 6x28x28 output image

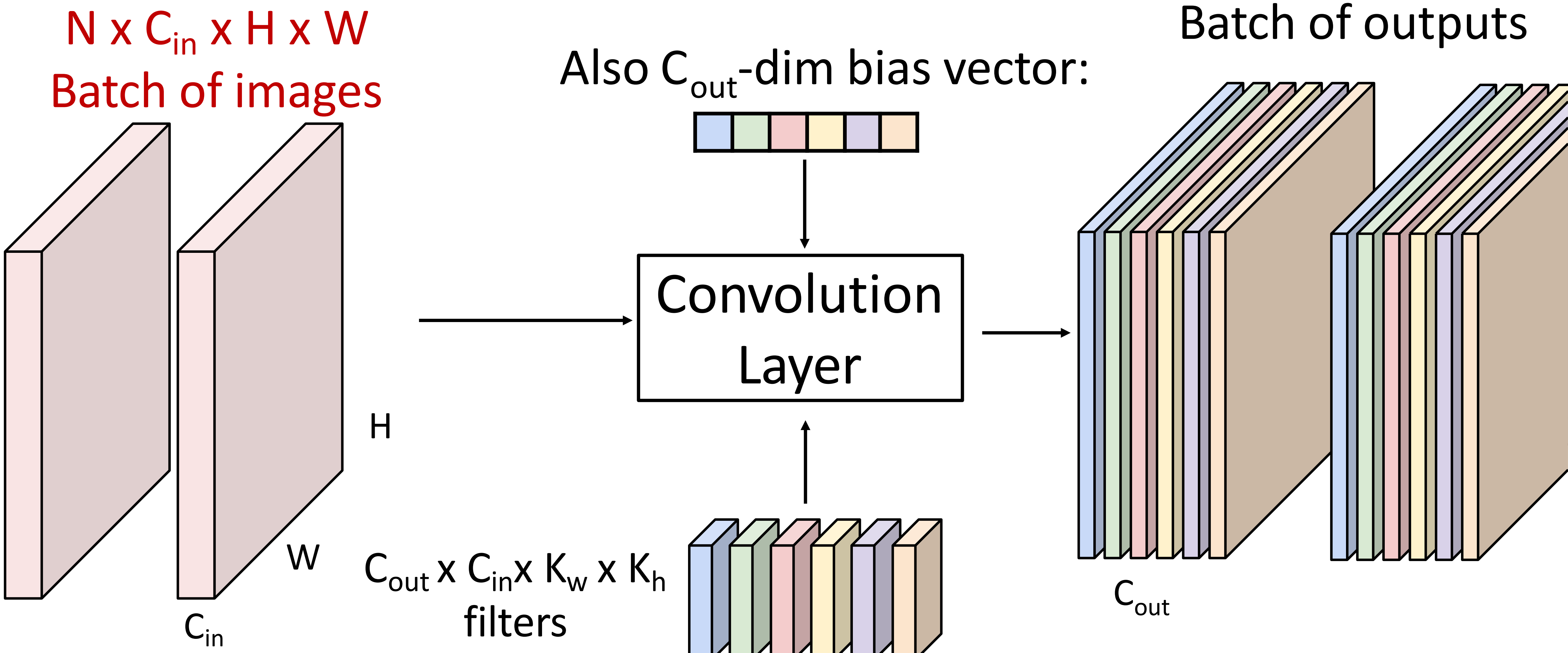
Convolution Layer



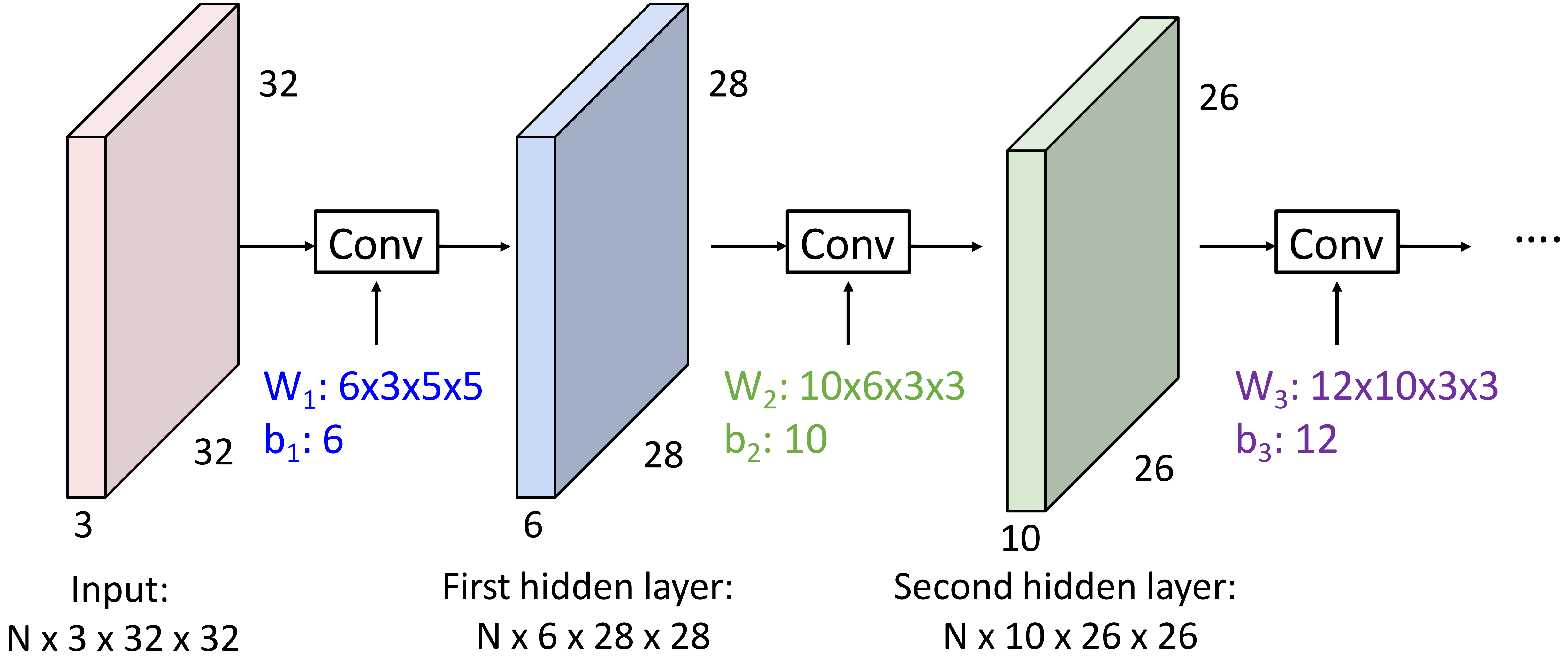
Convolution Layer



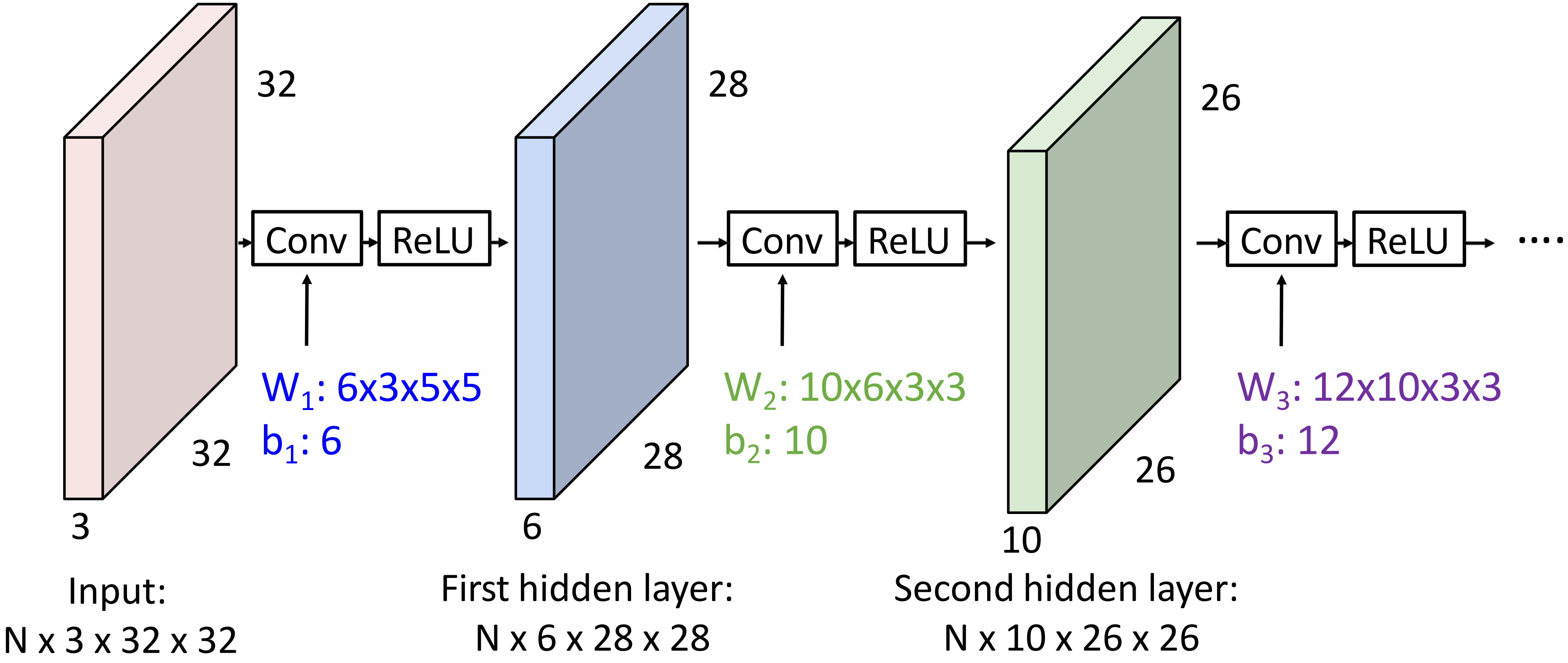
Convolution Layer



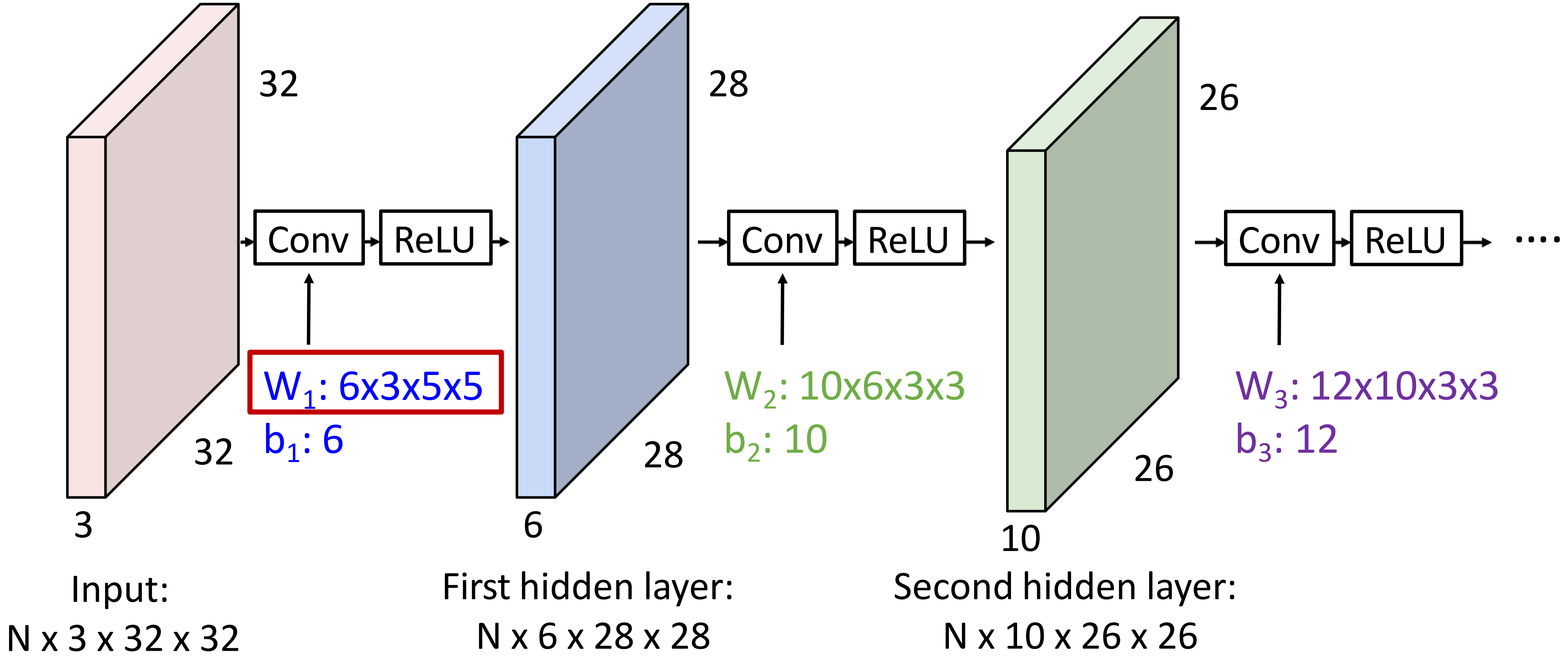
Stacking Convolutions



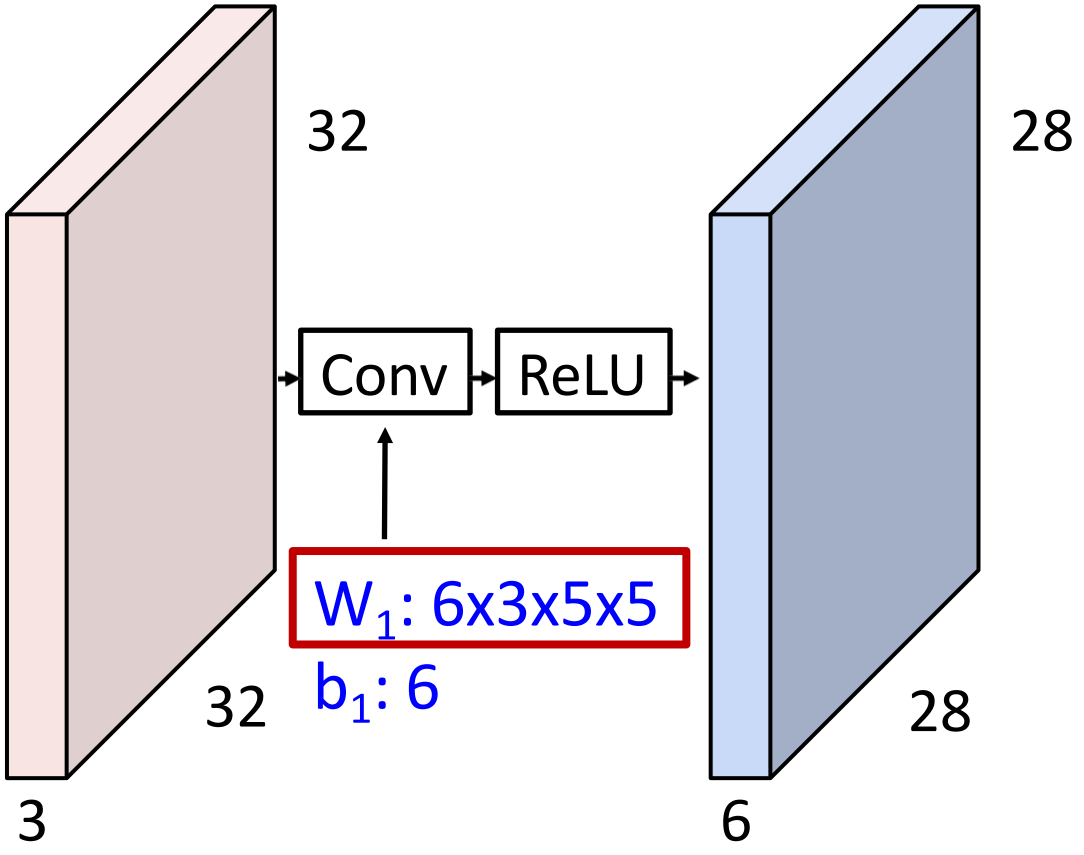
Stacking Convolutions: Add Non-linearity



What do convolutional filters learn?



What do convolutional filters learn?

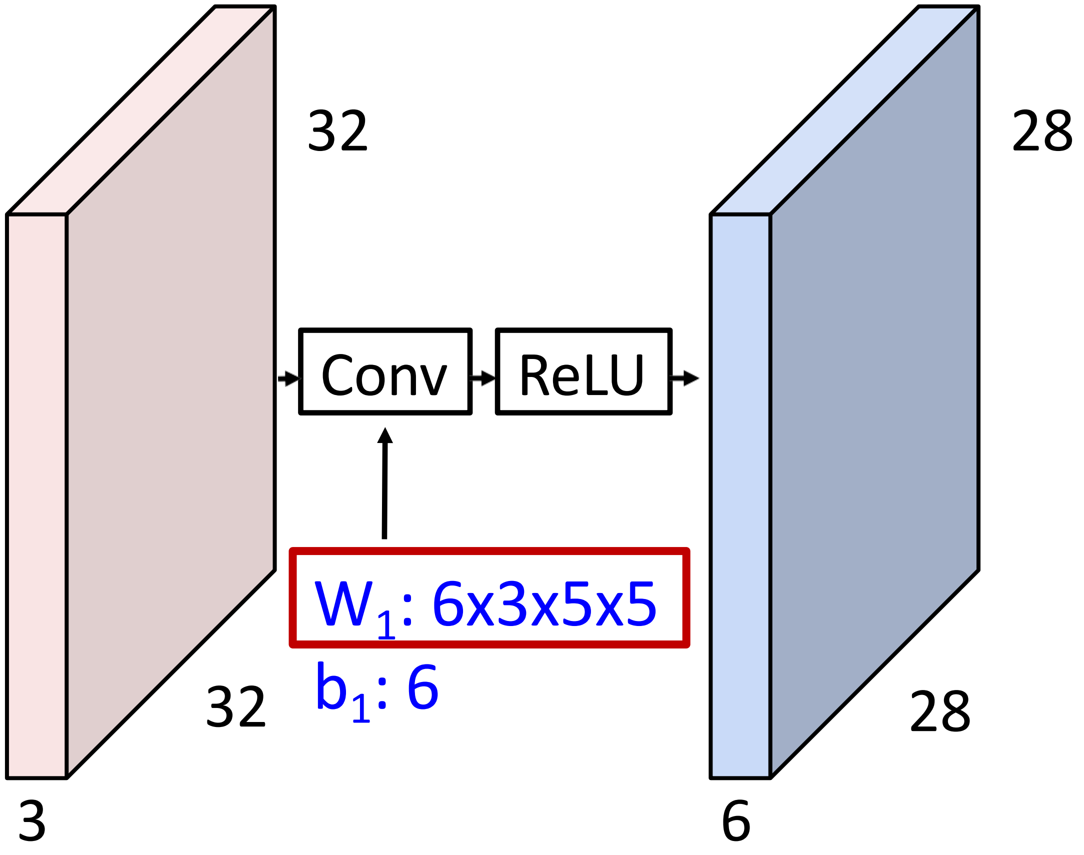


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

Linear classifier: One template per class

What do convolutional filters learn?

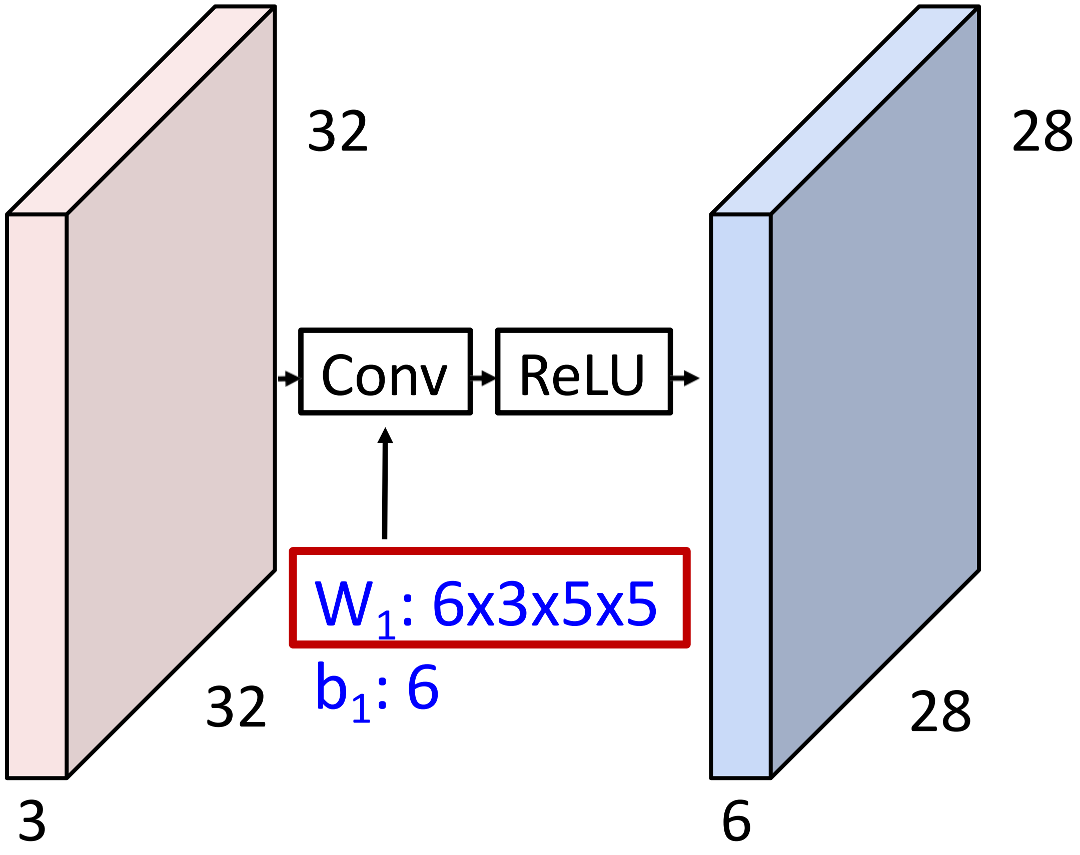


Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

MLP: Bank of whole-image templates

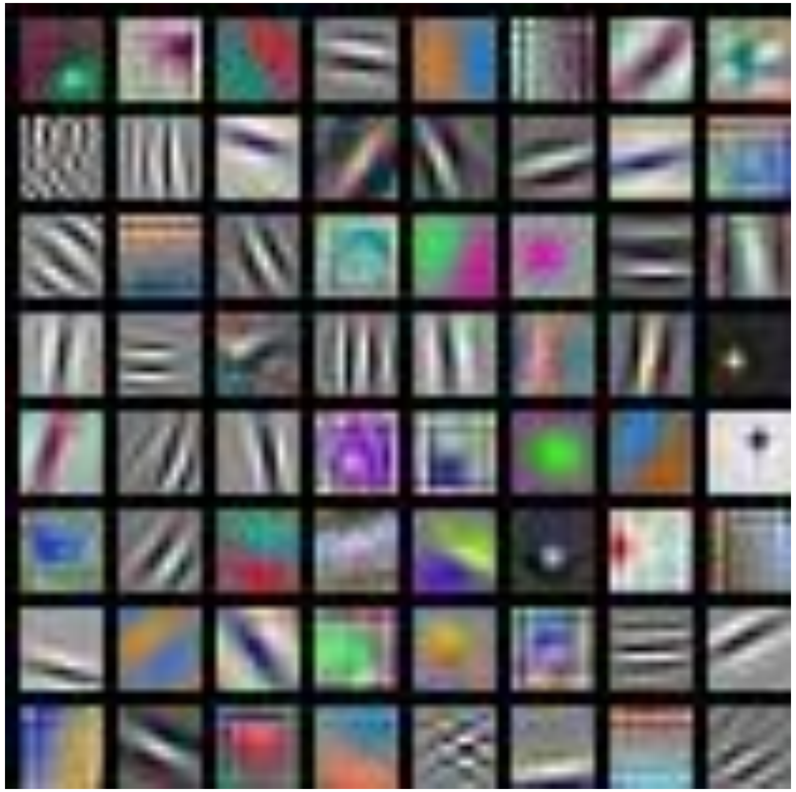
What do convolutional filters learn?



Input:
N x 3 x 32 x 32

First hidden layer:
N x 6 x 28 x 28

First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)



AlexNet: 64 filters, each 3x11x11