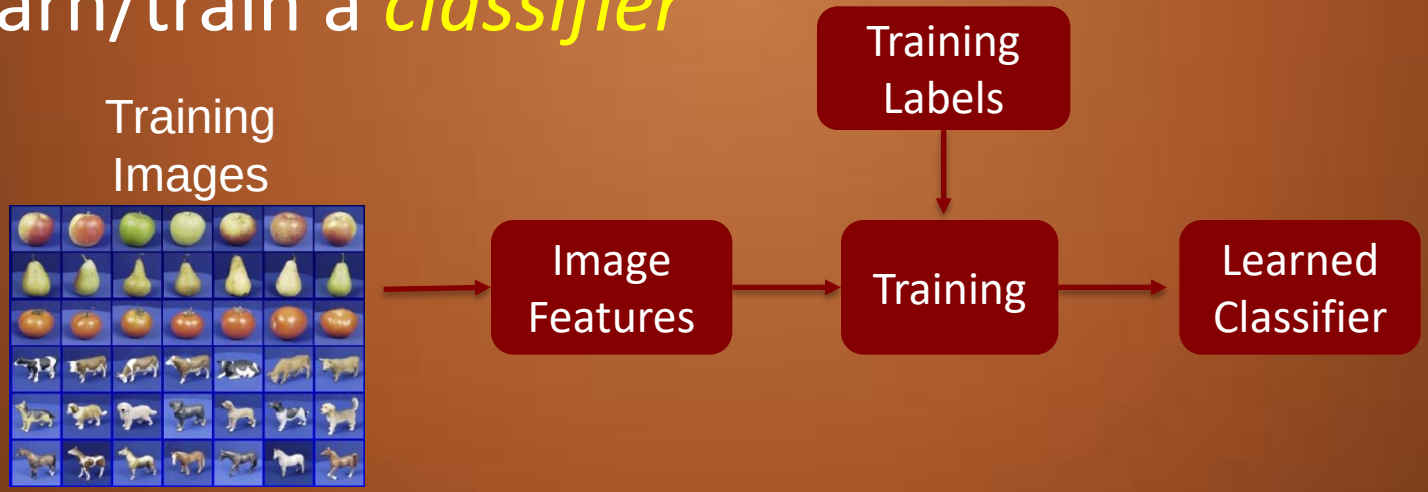
Image and Video Processing

Image Classification/Recognition

Recap: Basic Recognition framework

Train

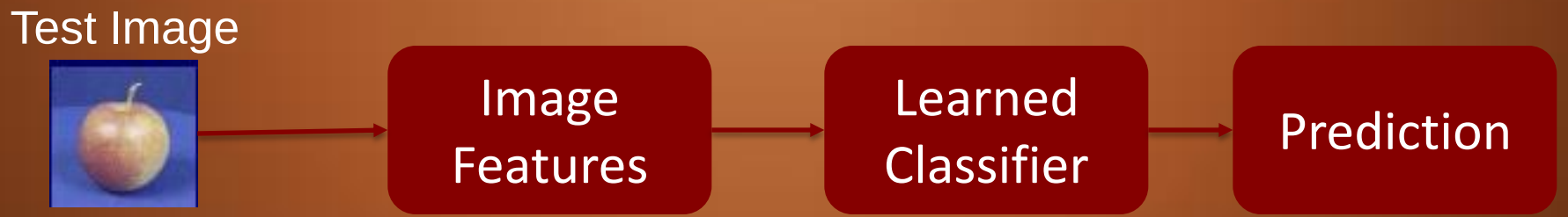
- Feature *representation - to describe training instances/objects (here images)*
- Learn/train a *classifier*



Recap: Basic Recognition framework

Test

- Generate candidates in new image
- *Score* the candidates



First classifier: Nearest Neighbor Classifier

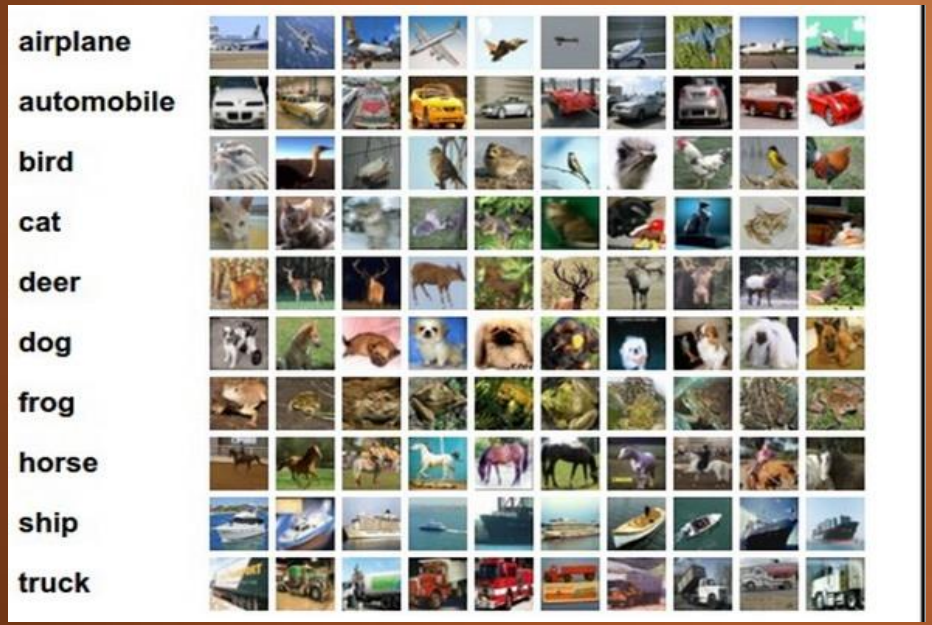
```
def train(train_images, train_labels):
  # build a model for images -> labels...
  return model

def predict(model, test_images):
  # predict test_labels using the model...
  return test_labels
```

Remember all training images and their labels

Predict the label of the most similar training image

Example dataset: CIFAR-10

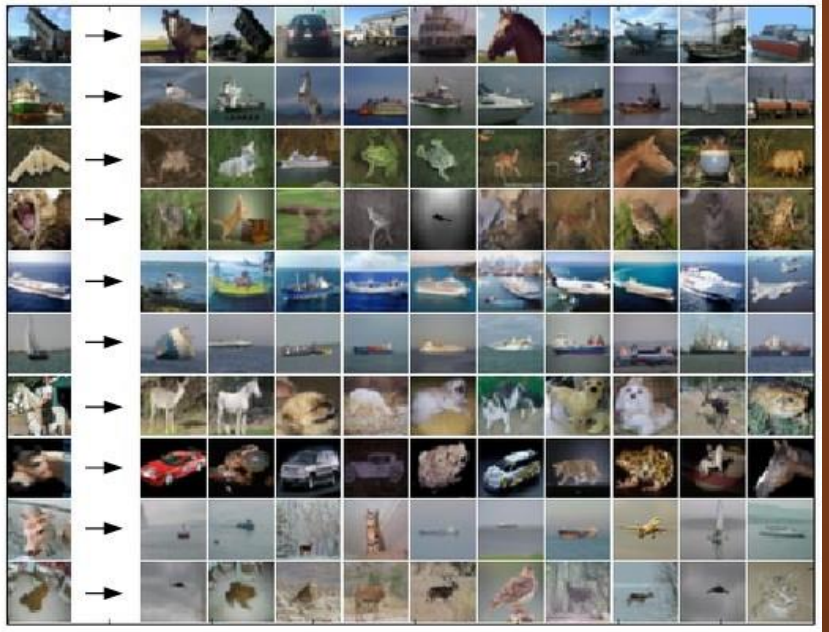
10 labels
50,000 training images, each image is tiny: 32x32
10,000 test images.



Example dataset: CIFAR-10

10 labels
50,000 training images
10,000 test images.

For every test image (first column), examples of nearest neighbors in rows



How do we compare the images?
What is the **distance metric**?

L1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 (Euclidean) distance

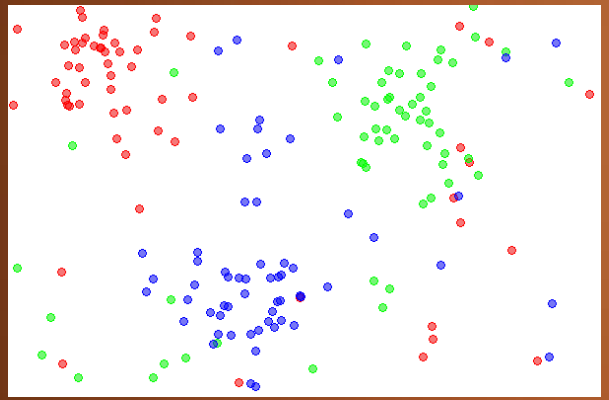
$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$

test image					training image					pixel-wise absolute value differences				
56	32	10	18	-	10	20	24	17	=	46	12	14	1	→ 456
90	23	128	133		8	10	89	100		82	13	39	33	
24	26	178	200		12	16	178	170		12	10	0	30	
2	0	255	220		4	32	233	112		2	32	22	108	

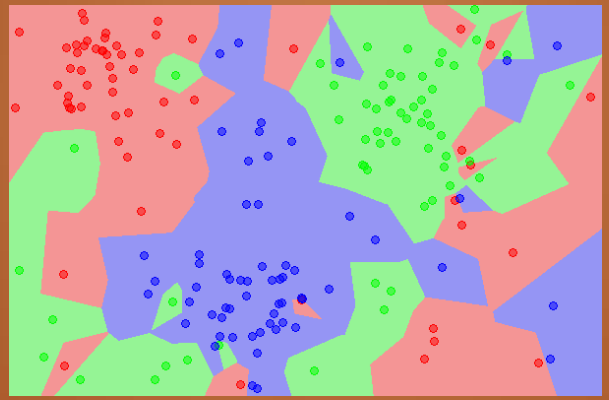
k-Nearest Neighbor

find the k nearest images, have them vote on the label

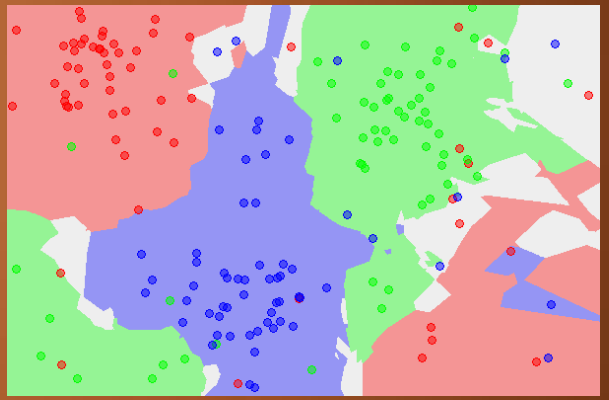
the data



NN classifier



5-NN classifier



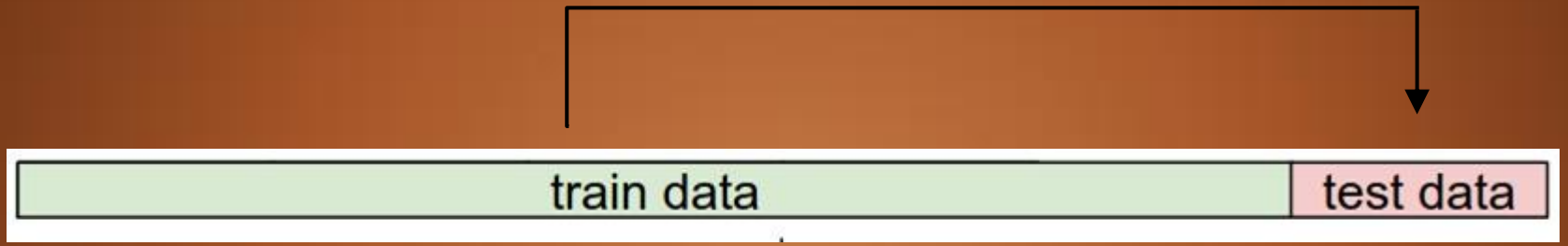
http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

Tuning of Hyperparameters

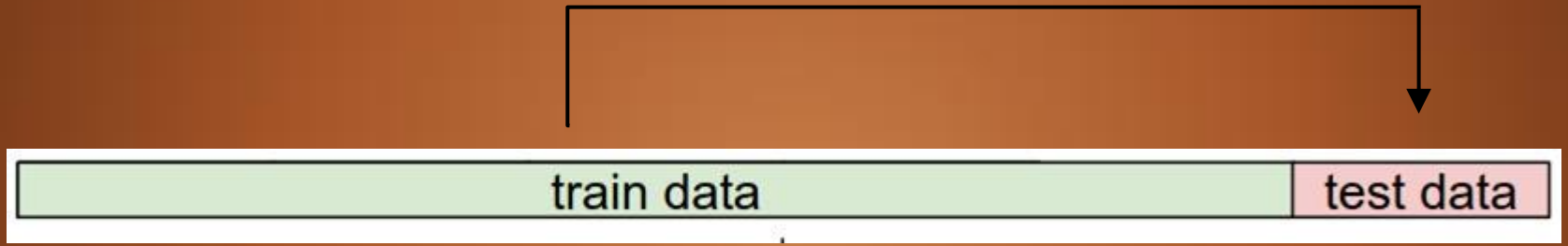
What is the best **distance** to use?
What is the best value of **k** to use?
i.e. how do we set the **hyperparameters**?

Very problem-dependent.
Must try them all out and see what works best.

Try out what hyperparameters work best on test set.



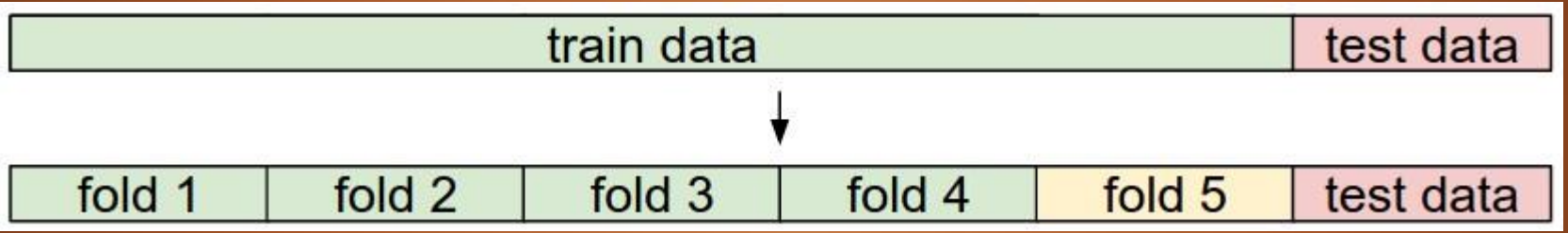
Trying out what hyperparameters work best on test set:



Very bad idea. The test set is a proxy for the generalization performance!

Use only **VERY SPARINGLY,** at the end.

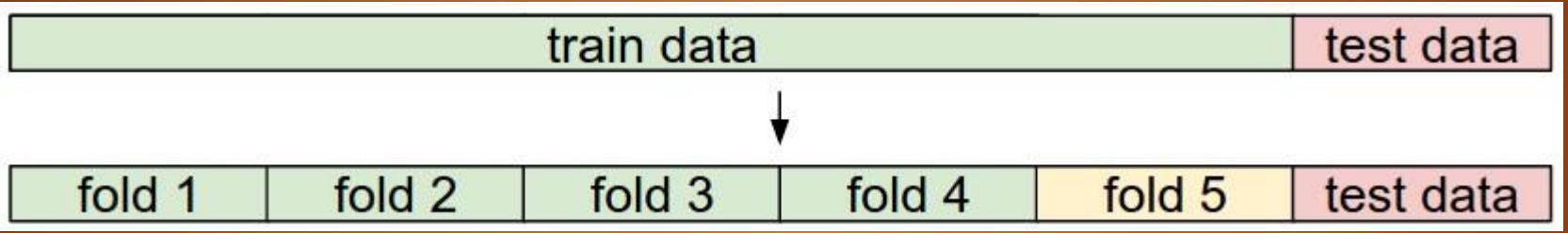
Validation Set:



Validation data

use to tune hyperparameters

Validation Set:



Cross-validation

cycle through the choice of which fold is the validation fold, average results.

Performance?

Q: how does the classification speed depend on the size of the training data?

Ans: **linearly :(Bad!**

Test time performance is usually much more important in practice.

Q: what is the accuracy on test data?

Ans: Can mess up badly!

Distance metrics on level of whole images can be very unintuitive

Parametric approach

image parameters

f(**x**, **W**)

10 numbers, indicating class scores

[32x32x3]
array of numbers 0...1
(3072 numbers total)

Parametric approach: **Linear classifier**

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

10 numbers, indicating class scores

[32x32x3]
array of numbers 0...1

Parametric approach: **Linear classifier**

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

3072x1

10x1

10x3072

[32x32x3]
array of numbers 0...1

10 numbers, indicating class scores

parameters, or “weights”

Parametric approach: **Linear classifier**

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

3072x1

(+b) 10x1

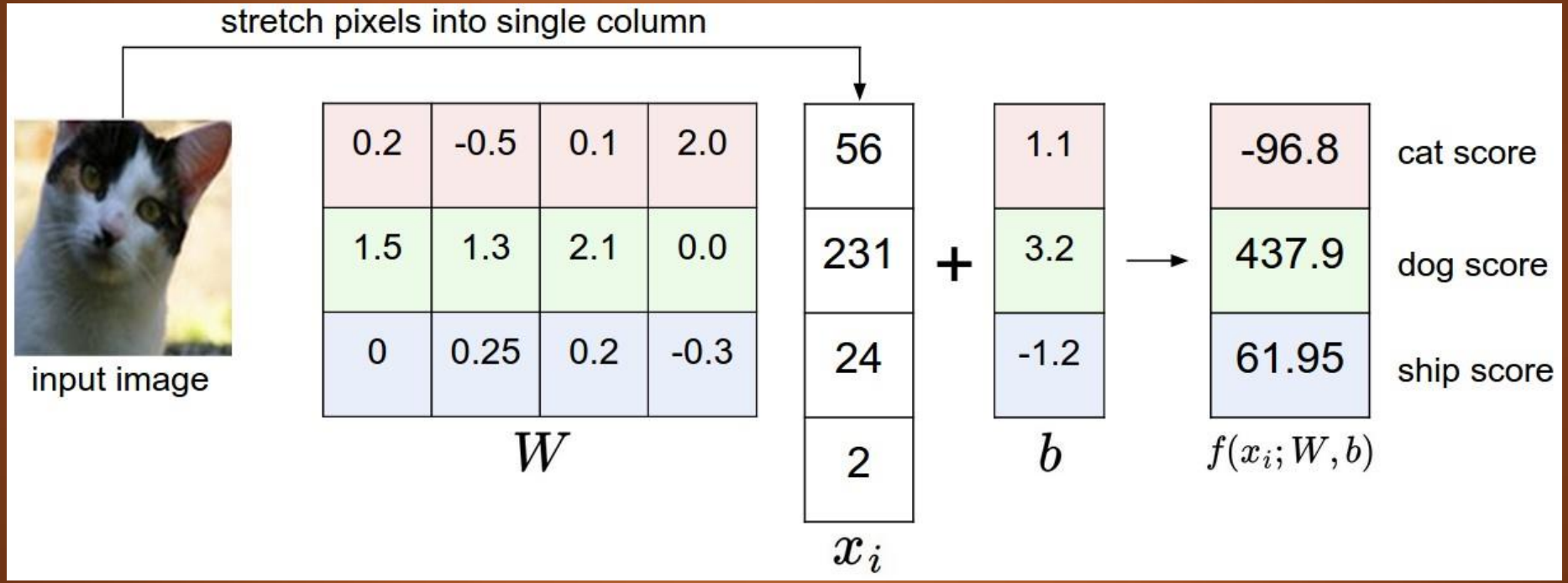
10x1 10x3072

[32x32x3]
array of numbers 0...1

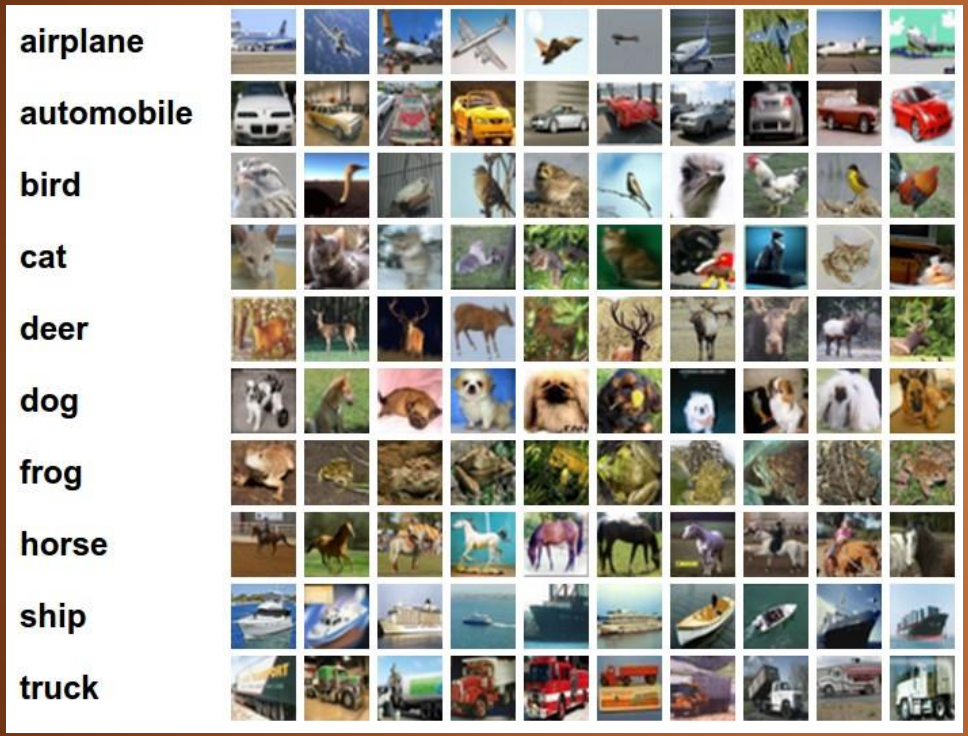
10 numbers, indicating class scores

parameters, or “weights”

Example image with 4 pixels, and 3 classes (cat/dog/ship)



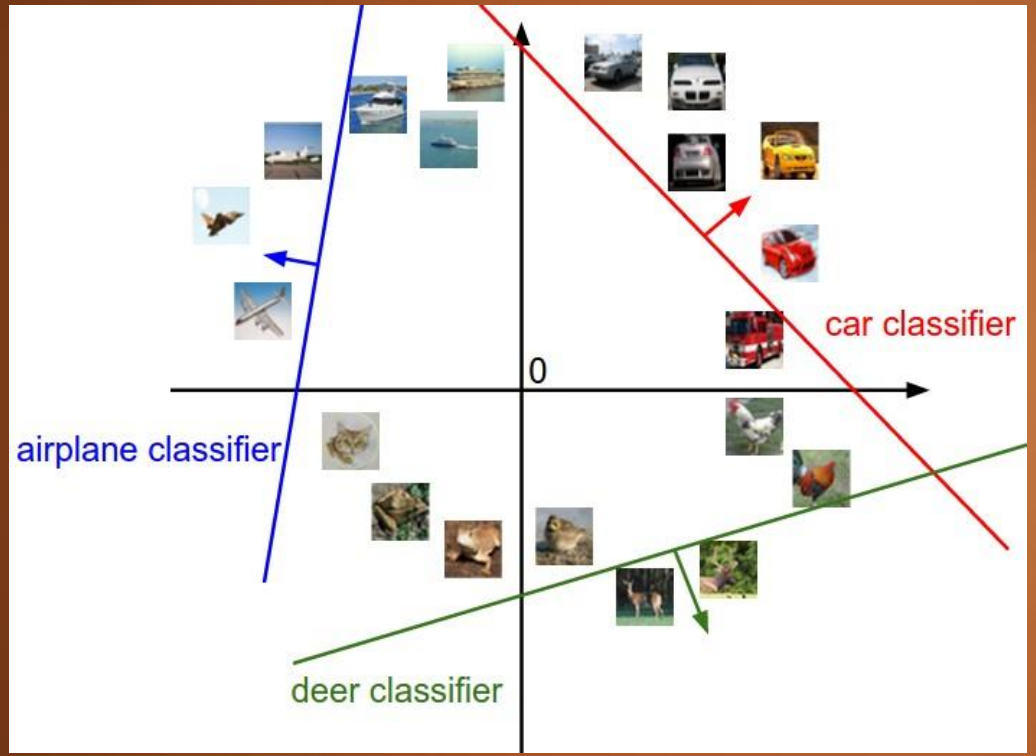
Interpreting a Linear Classifier



$$f(x_i, W, b) = Wx_i + b$$

Q: what does the linear classifier do, in English?

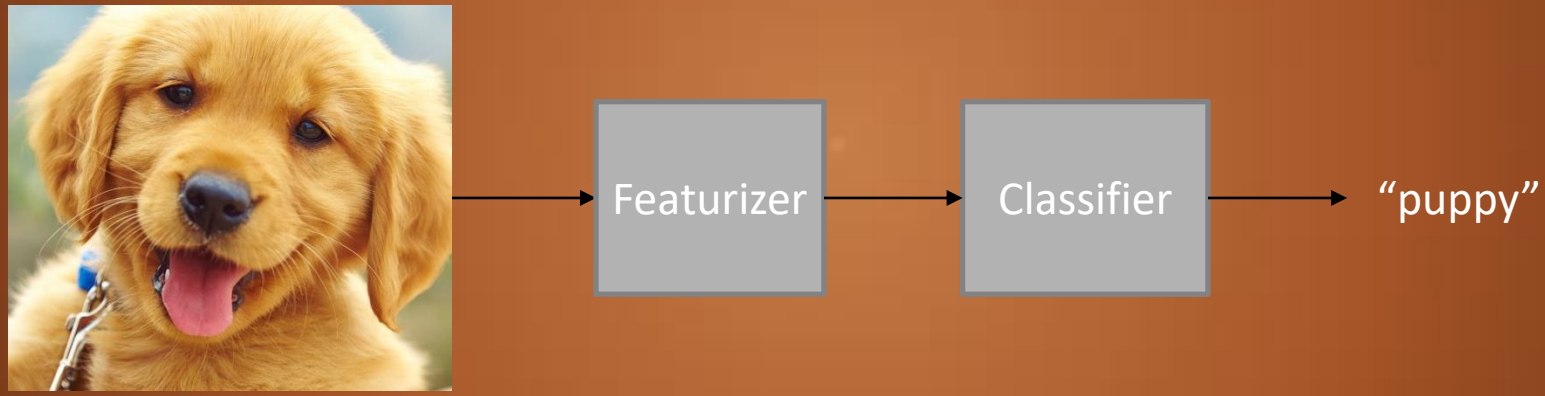
Interpreting a Linear Classifier



$$f(x_i, W, b) = Wx_i + b$$

what would be a very hard set of classes for a linear classifier to distinguish?

Recall Recognition framework…



Extract features to represent images and train classifier

Feature Extractor :

The weights can be seen as template/filter which when convolved gives higher score/probability for that input image which matches the template.

*

32x32 “Airplane Filter”

=

Feature Extractor

*

32x32 “Airplane Filter”

=

“probability” of the image being an airplane

Feature Extractor

*

32x32 “Airplane Filter”

=

“probability” of the image being an airplane

Feature Extractor

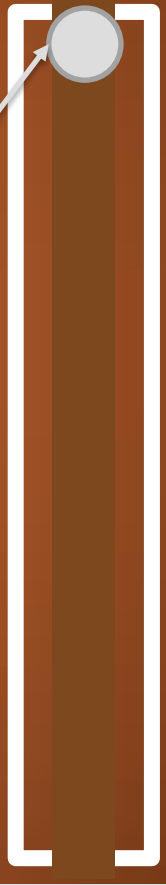
This is not really a probability but a score, because it can be less than 0 and greater than 1

*

32x32 “Airplane Filter”

=

“probability” of the image being an airplane



Feature Extractor

*

32x32
“Automobile
Filter”

=

“probability” of
the image being an
automobile

Feature Extractor

*

32x32 “Bird Filter”

=

“probability” of the image being a bird

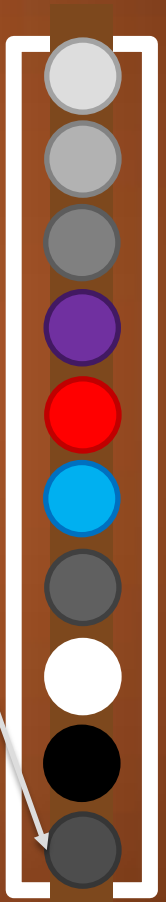
Feature Extractor

*

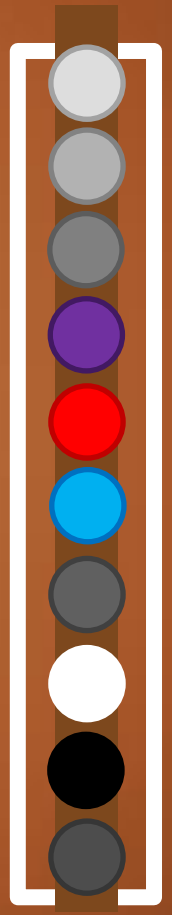
32x32 “Truck Filter”

=

“probability” of the image being a truck

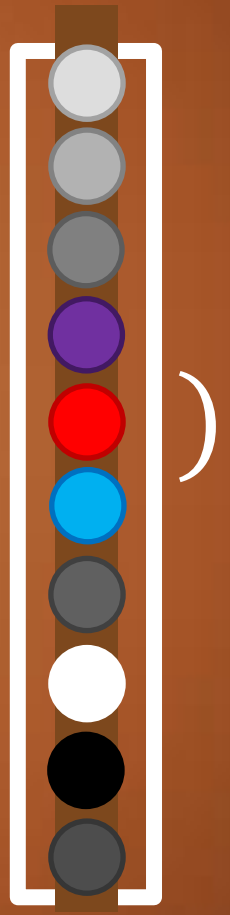


Classifier

$$c_{pred} = \arg\max($$  $$)$$

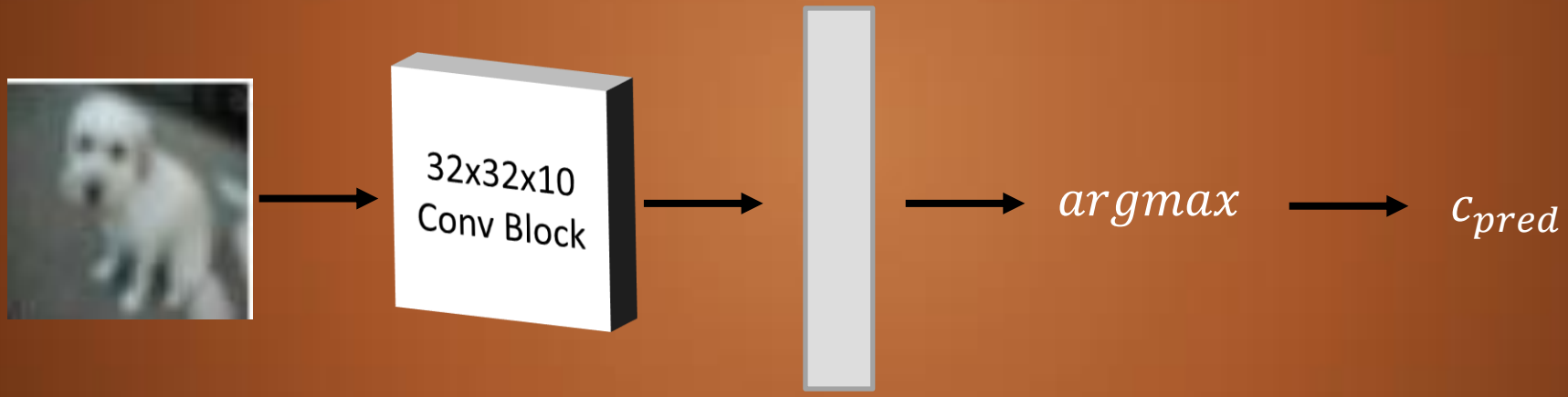
Classifier

$$c_{pred} = \arg\max(\quad)$$

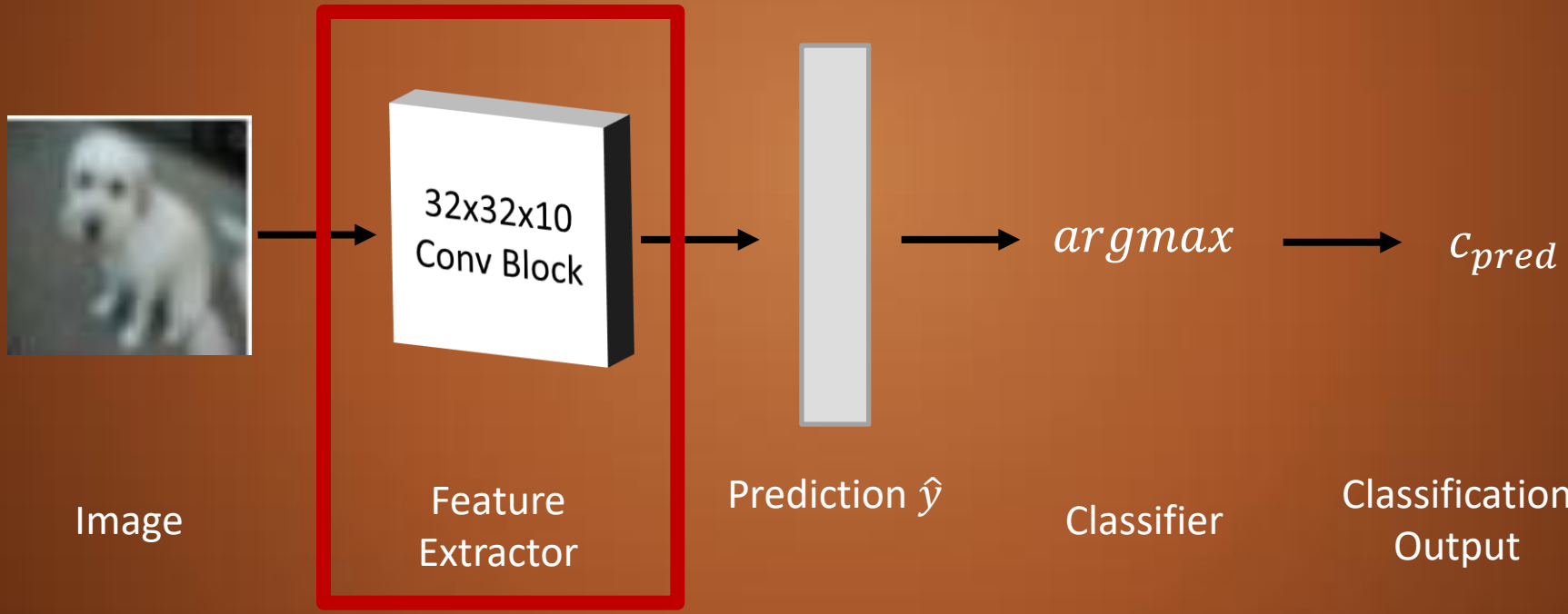


We predict the class that has the highest probability/score!

The Pipeline



The Pipeline



Feature Extractor

$$Wx = \hat{y}$$

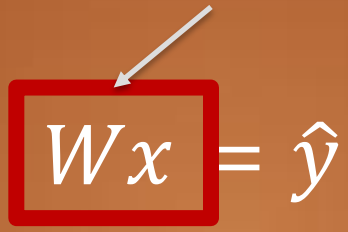
W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”

Feature Extractor

This simple computation can be seen as *fully-connected NN layer*!

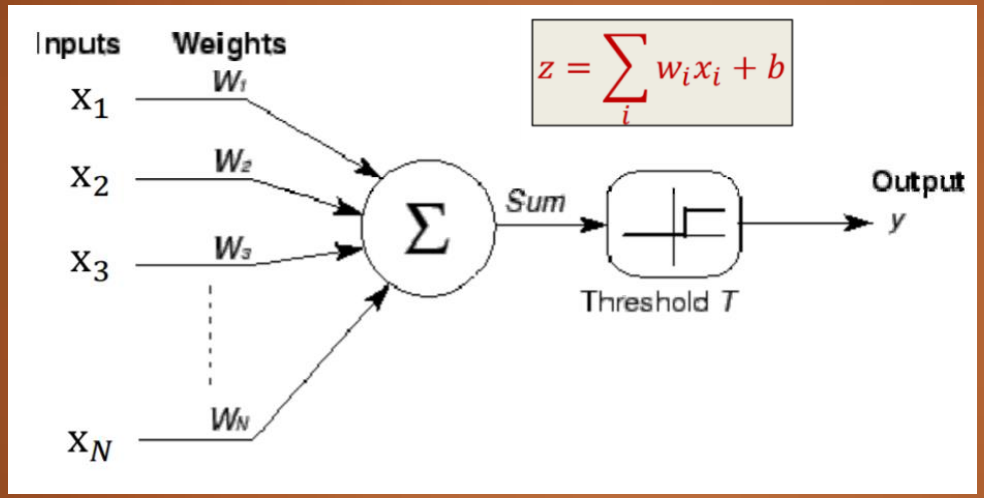


W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

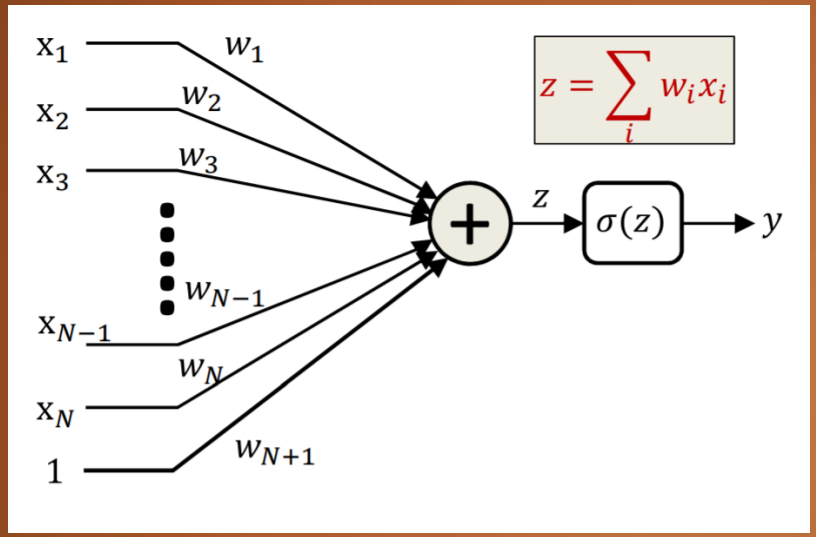
$\hat{y}$: the (10x1) vector of class "probabilities"

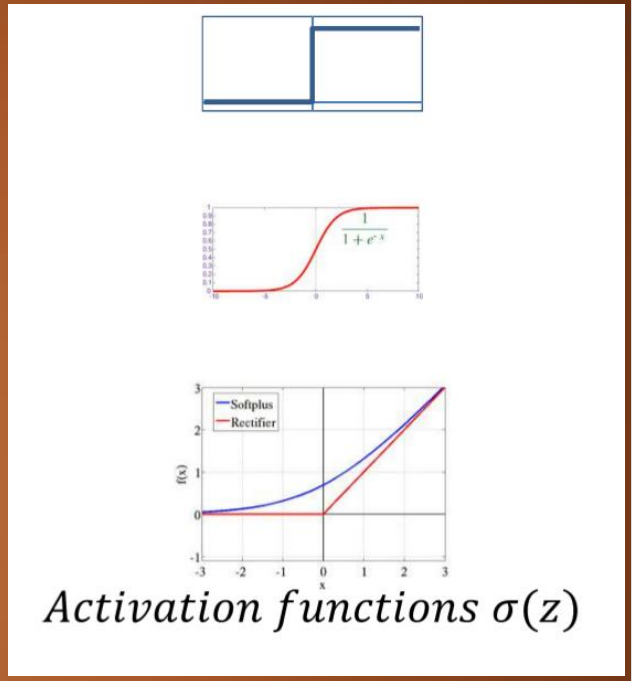
Aside: Original Perceptron



- Unit comprises a set of weights and a threshold/bias
- The bias can also be viewed as the weight of another input component that is always set to 1

Aside: Units in the network

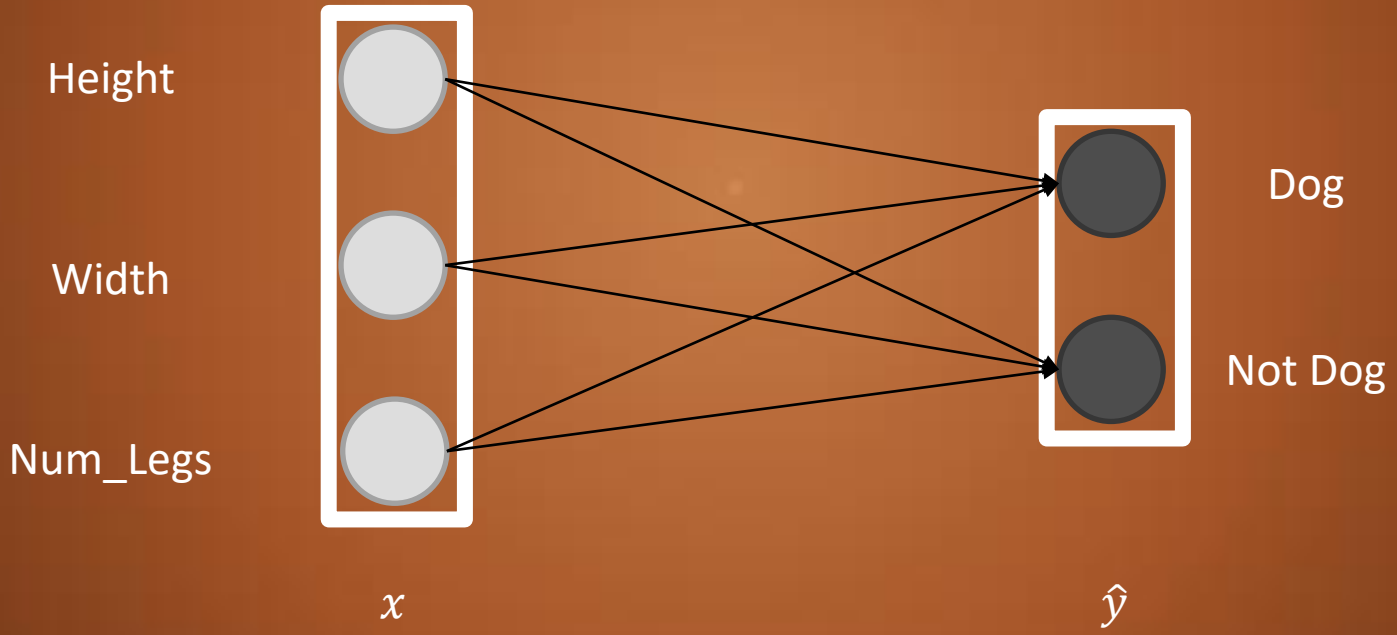




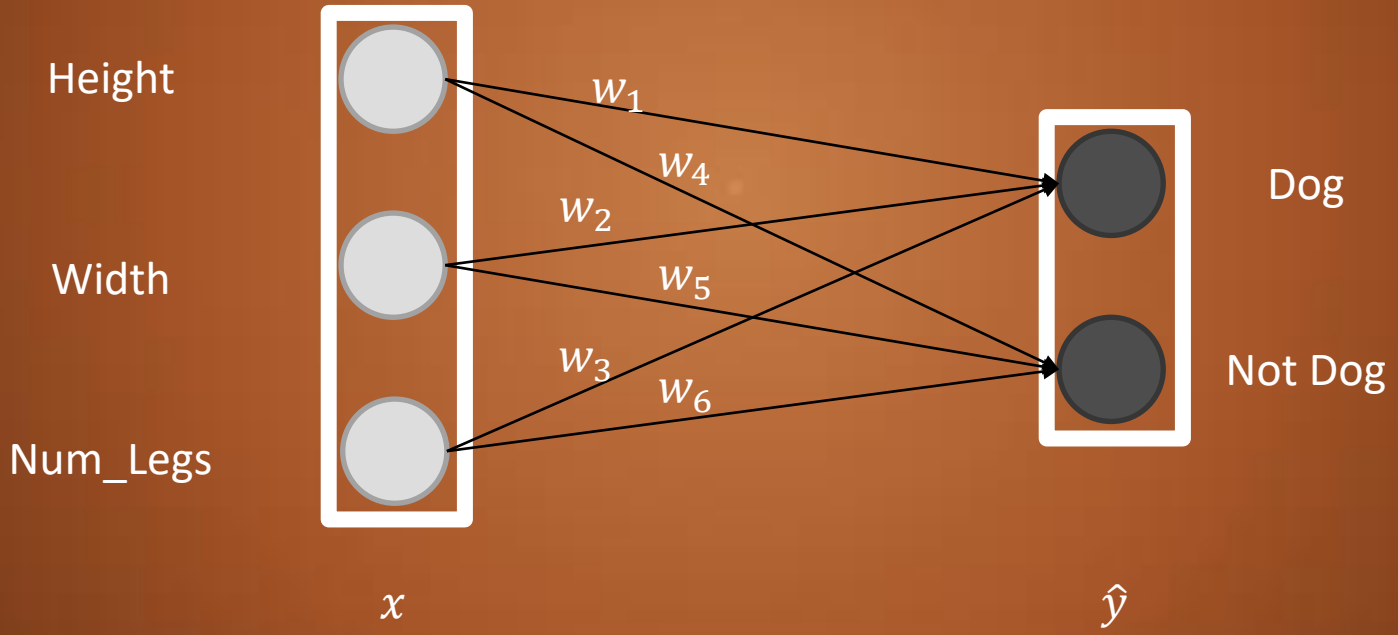
Activation functions are not necessarily threshold functions

- With these output is real valued between 0 and 1
- Introduces non-linearity
- Small change in any weight cause small change in output

Aside: Fully-Connected Neural Networks



Aside: Fully-Connected Neural Networks



Aside: Fully-Connected Neural Networks

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \cdot$$  $=$ 

W x $\hat{y}$

$$Wx = \hat{y}$$

New Feature Extractor

$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”

New Feature Extractor

$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”?

Class Probability Vector

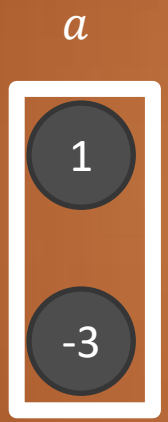
Must have values between 0 and 1

Must sum to 1

There's no guarantee either requirement is satisfied!

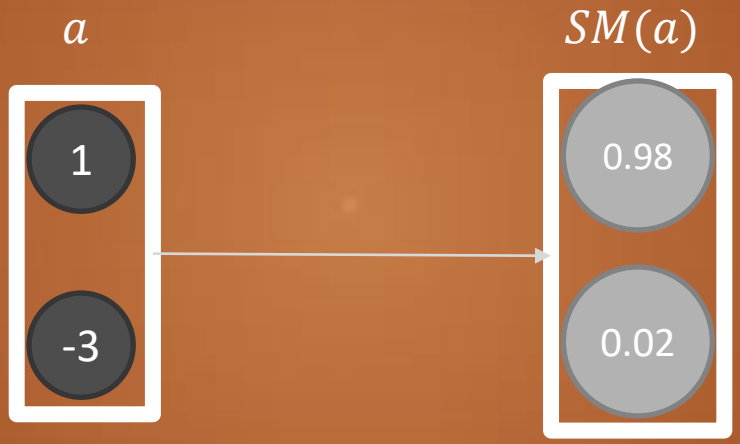
$$\hat{y} = Wx$$

Softmax: An activation Function



Softmax: $a(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Softmax Function



Softmax: $a(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Class Probability Vector

Must have values between 0 and 1

Must sum to 1

$$\hat{y} = SM(Wx)$$

System so far…

Feature extractor:

$$\hat{y} = SM(Wx)$$

Classifier:

$$c_{pred} = \arg\max(\hat{y})$$

Learning the parameters (W)

Let's compare our prediction with the real answer! For each image, we have the label y which tells us the true class:

X

y

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Dog class index

Key Insight:

We want:

$$\arg\max(\hat{y}) = \arg\max(y)$$

Key Insight:

We want:

$$\arg\max(\hat{y}) = \arg\max(y)$$

Which we can accomplish by:

$$W^* = \arg\min_W \left(-\sum_{x,y} \log(p_c) \right)$$

Key Insight:

We want:

$$\arg\max(\hat{y}) = \arg\max(y)$$

Which we can accomplish by:

$$W^* = \arg\min_W \left(-\sum_{x,y} \log(p_c) \right)$$

Where p_c is the probability of the true class in y

Loss function: log likelihood

Our loss function represents *how bad we are currently doing*:

$$L = -\log(p_c)$$

Loss function: log likelihood

Our loss function represents *how bad we are currently doing*:

$$L = -\log(p_c)$$

Examples:

$p_c = 0 \rightarrow L = -\log(0) = \infty$

$p_c = 0.1 \rightarrow L = -\log(0.1) = 2.3$

$p_c = 0.9 \rightarrow L = -\log(0.9) = 0.1$

$p_c = 1 \rightarrow L = -\log(1) = 0$

Loss function: log likelihood

Our loss function represents *how bad we are currently doing*:

$$L = -\log(p_c)$$

Examples:

$p_c = 0 \rightarrow L = -\log(0) = \infty$

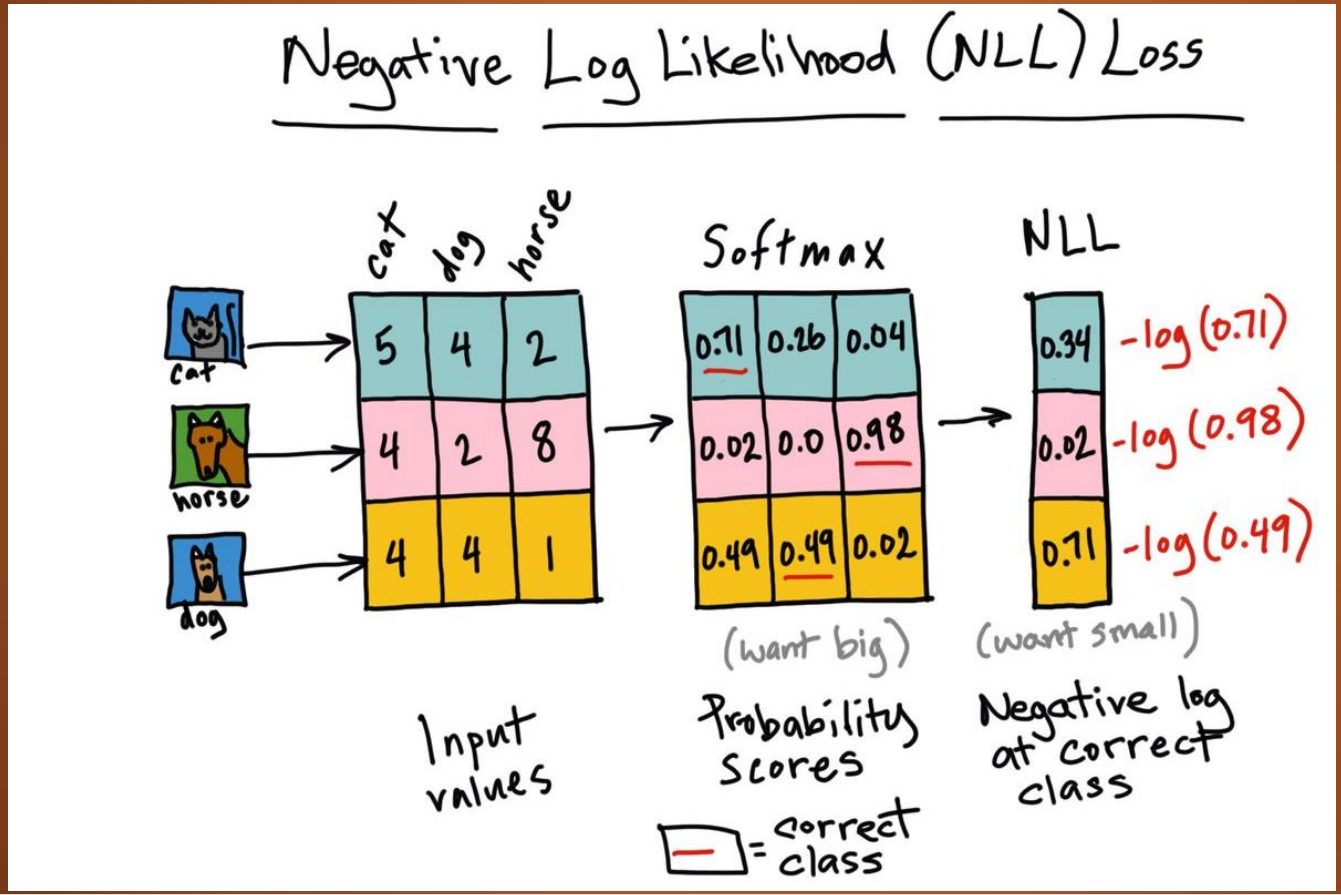
$p_c = 0.1 \rightarrow L = -\log(0.1) = 2.3$

$p_c = 0.9 \rightarrow L = -\log(0.9) = 0.1$

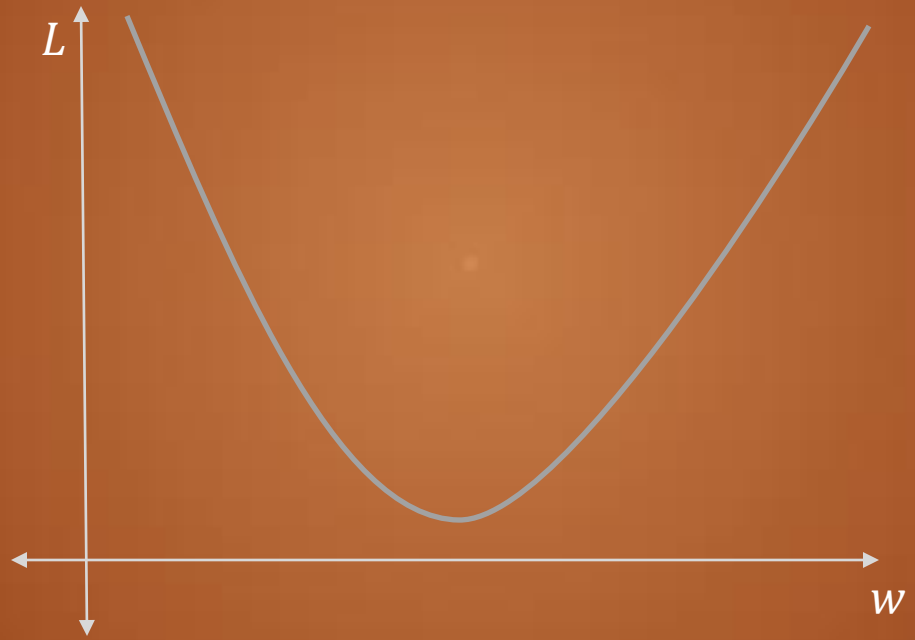
$p_c = 1 \rightarrow L = -\log(1) = 0$

The larger the loss, the worse our prediction. We want to minimize L!

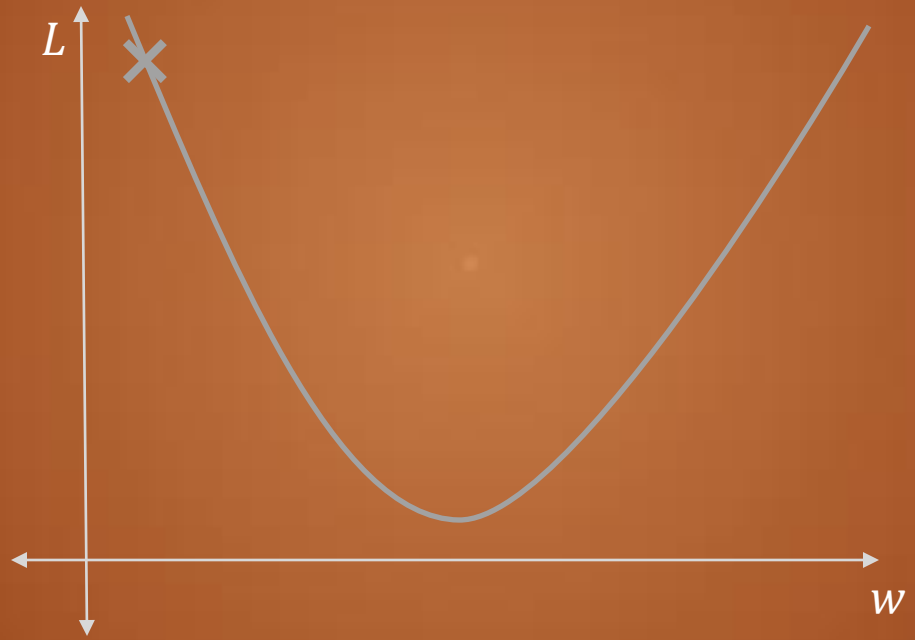
Example


Negative Log Likelihood (NLL) Loss
cat
dog
horse
5 4 2
4 2 8
4 4 1
Softmax
0.71 0.26 0.04
0.02 0.0 0.98
0.49 0.49 0.02
NLL
0.34 -log(0.71)
0.02 -log(0.98)
0.71 -log(0.49)
(want big)
(want small)
Input values
Probability scores
Negative log at correct class
= correct class

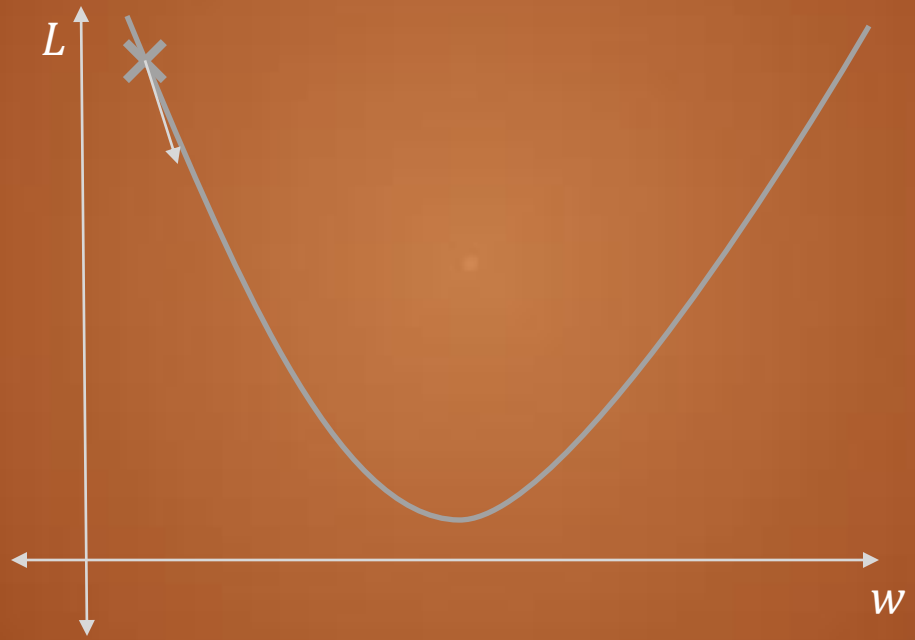
Minimizing Loss



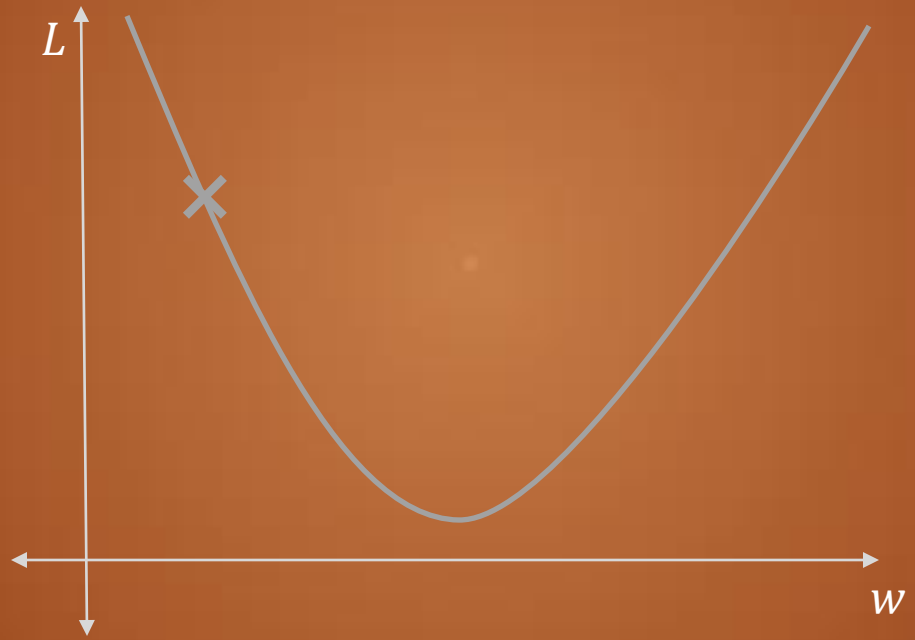
Minimizing Loss



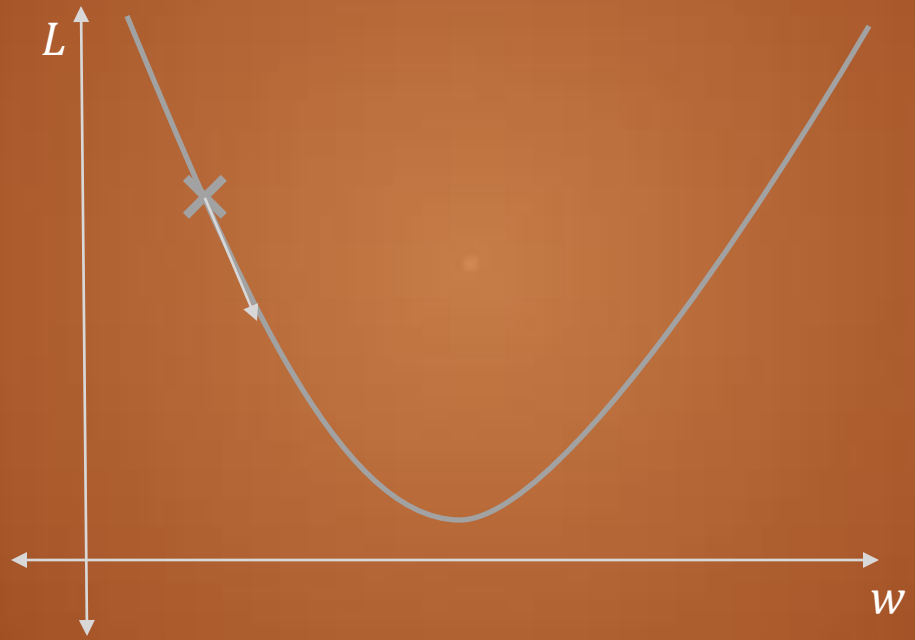
Minimizing Loss



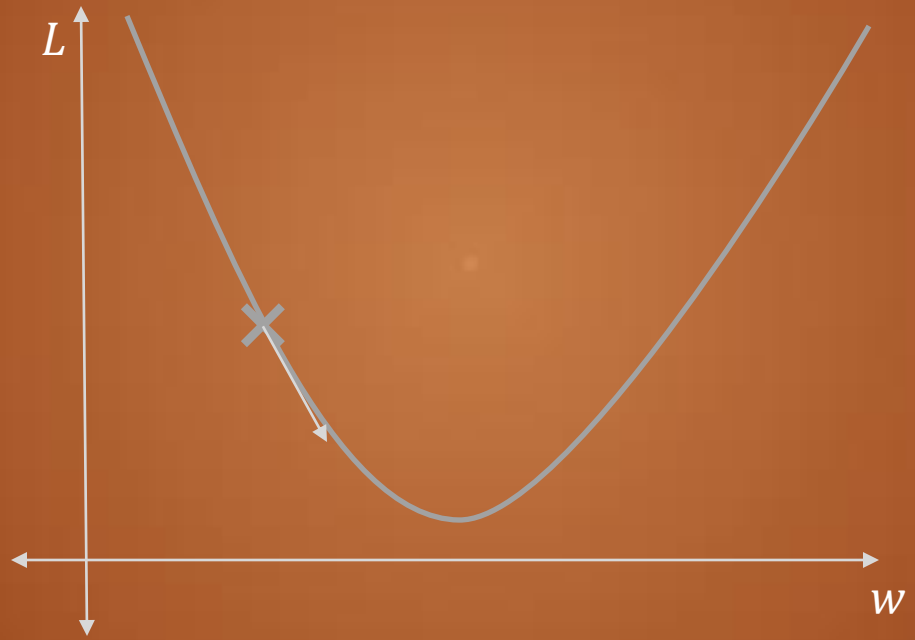
Minimizing Loss



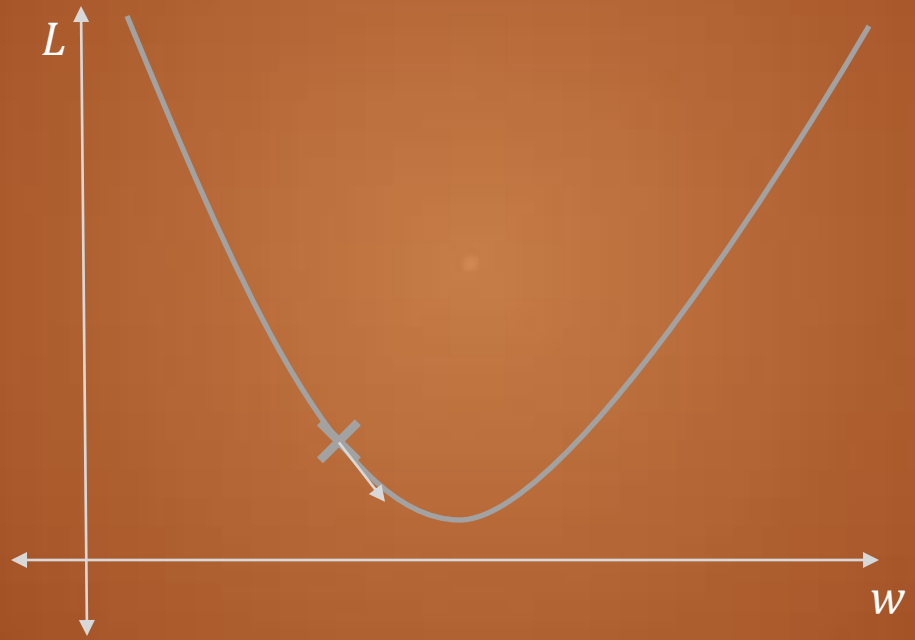
Minimizing Loss



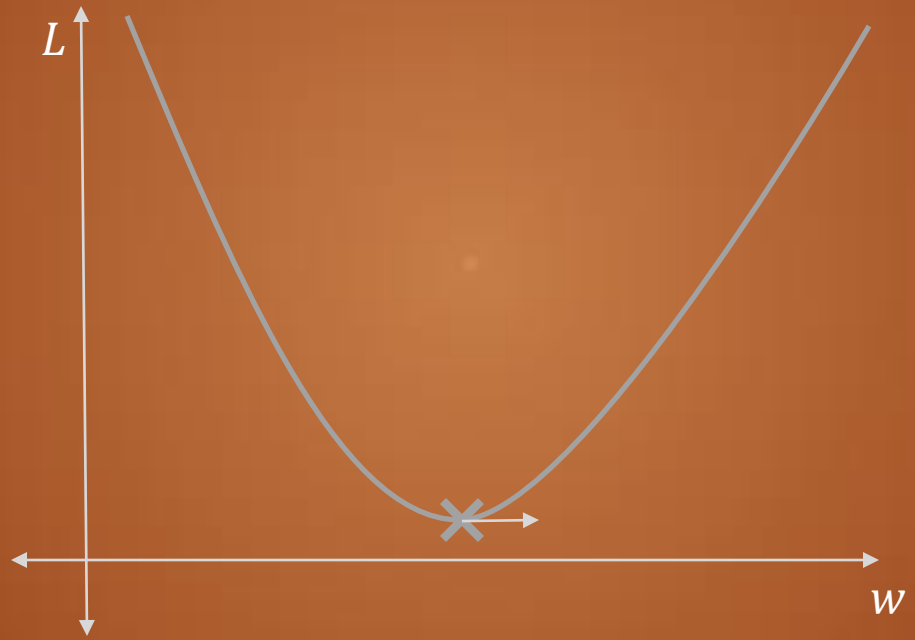
Minimizing Loss



Minimizing Loss



Minimizing Loss



Minimizing Loss

GRADIENT DESCENT

C here is the Loss/cost in the predicted output

- $\begin{bmatrix} \mathbf{w} + \Delta\mathbf{w} \\ b + \Delta b \end{bmatrix} \rightsquigarrow C + \Delta C.$
- $\Delta C \approx \nabla C \cdot \begin{bmatrix} \Delta\mathbf{w} \\ \Delta b \end{bmatrix}.$
- We want: $\Delta C < 0.$
- Update: $\begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} - \eta \nabla C.$
- $\eta > 0$ is called the *learning rate*.

STOCHASTIC GRADIENT DESCENT

1. Pick a mini-batch size $m \ll n$.
2. Randomly (without replacement) choose a mini-batch $\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_m}$.
3. Update all weights and biases:

$$\begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix} - \frac{\eta}{m} \sum_{j=1}^{m} \nabla C_{i_j}.$$

4. Repeat 2-3.
5. An *epoch* is completed, when every input $\mathbf{x}_i$ has been used.

Pseudocode for weight updation

for i in {0,…,num_epochs}:
  while (data set has atleast m input samples)
    Randomly pick a mini batch of size m
    for x, y in mini batch:

$$\hat{y} = SM(Wx)$$

$$\mathrm{L} = LL(\hat{y}, y)$$

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

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

Where SM stands for Softmax function and
LL for Log likelihood function

Getting the Gradient

$$z = Wx$$

$$L = LL(SM(z), y)$$

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

Getting the Gradient

$$z = Wx$$

$$L = LL(SM(z), y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}(x)$$

Getting the Gradient

$$z = Wx$$

$$L = LL(SM(z), y)$$

$$\frac{dL}{dW} = (SM(z) - y)(x)$$

*It can be shown with little bit of maths that derivative of log likelihood with respect to the softmax layer gives this simple and **desirable (?)** expression at the end.

- Suppose the probabilities we computed were p = [0.2, 0.3, 0.5], and that the correct class was the middle one. According to this derivation the gradient on the scores would be [0.2, -0.7, 0.5] .
- The gradient of -0.7 is telling us that increasing the correct class score would lead to a decrease of the loss. Also greater the error, faster the learning

Getting the Gradient

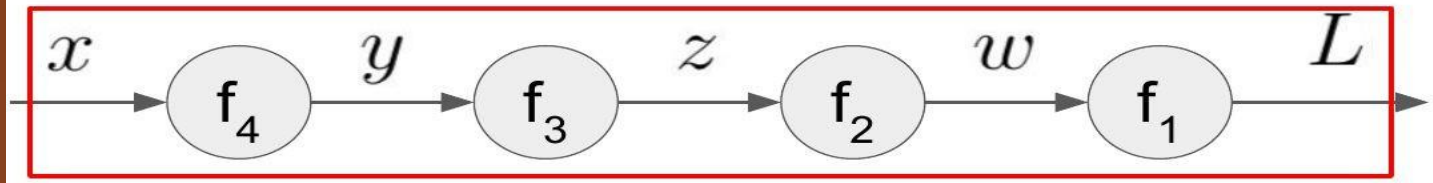
$$z = Wx$$
$$L = LL(SM(z), y)$$
$$\frac{dL}{dW} = (SM(z) - y)(x)$$

When there are more (hidden) layers in the network?

BACKPROPAGATION!

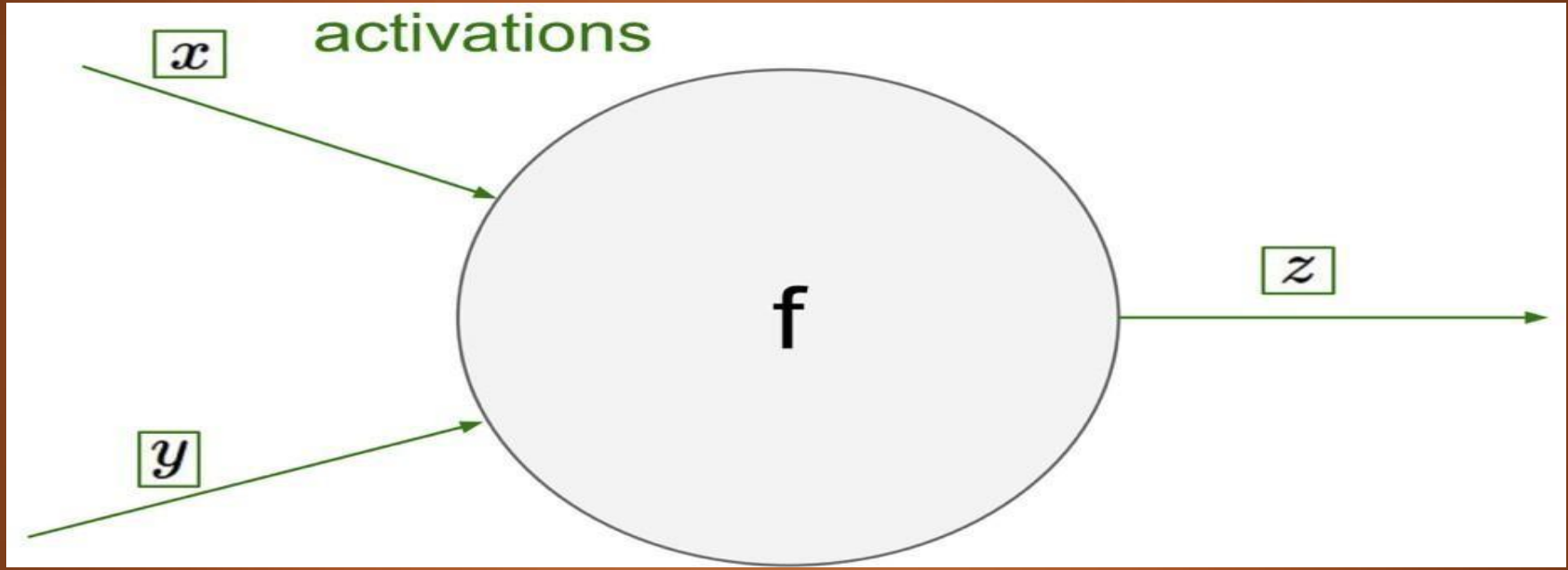
Chain Rule: Backpropagation

$$L(x) = f_1 \circ f_2 \circ f_3 \circ f_4\,(x)$$

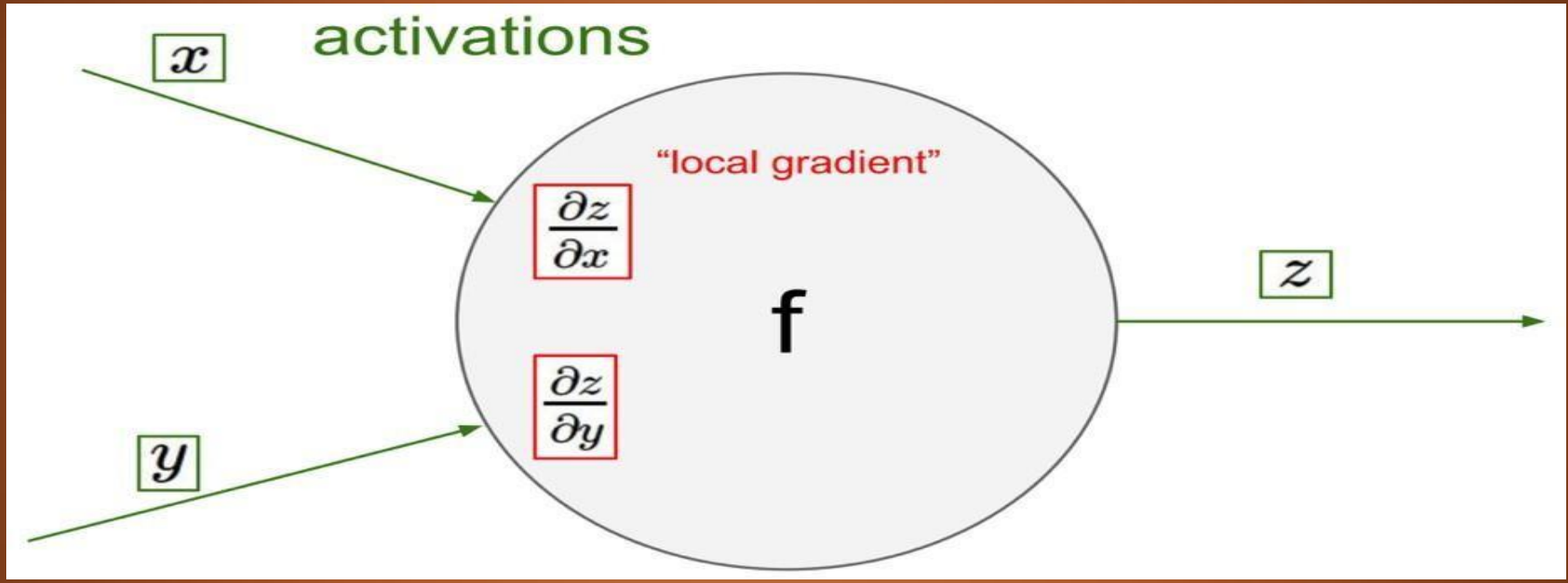


$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial w}\frac{\partial w}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

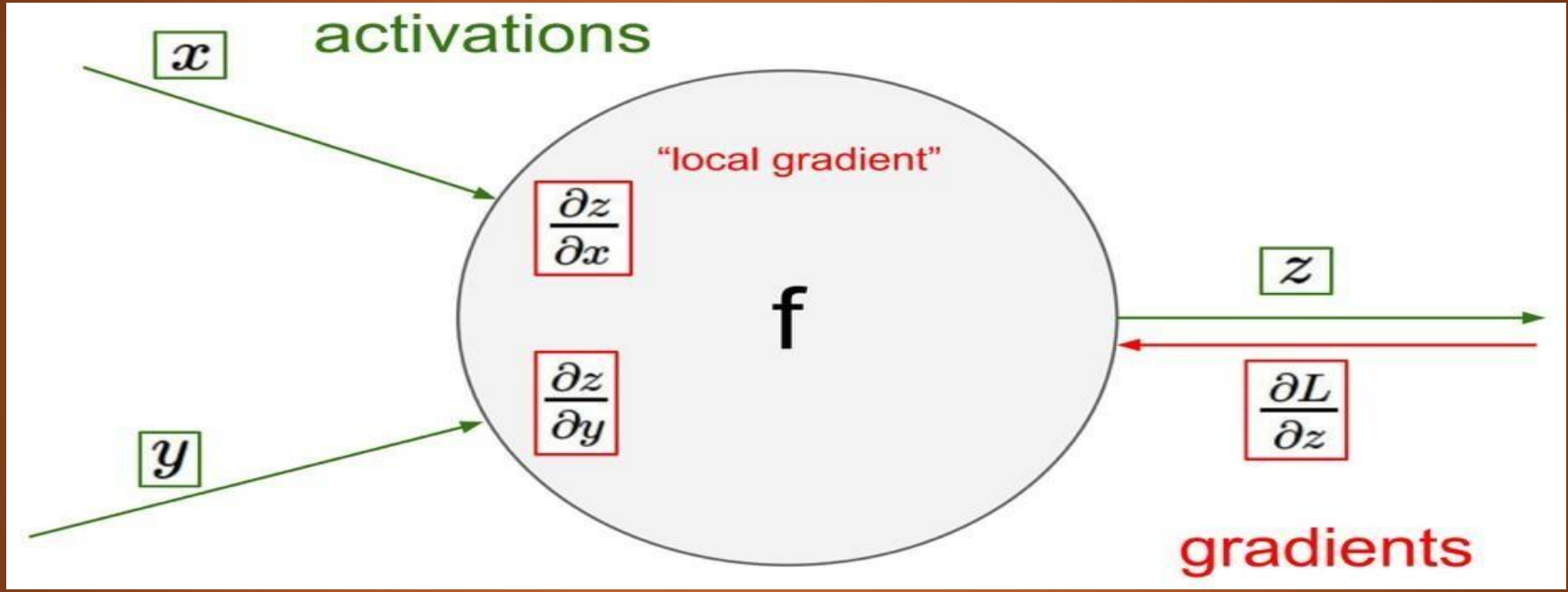
Backpropagation



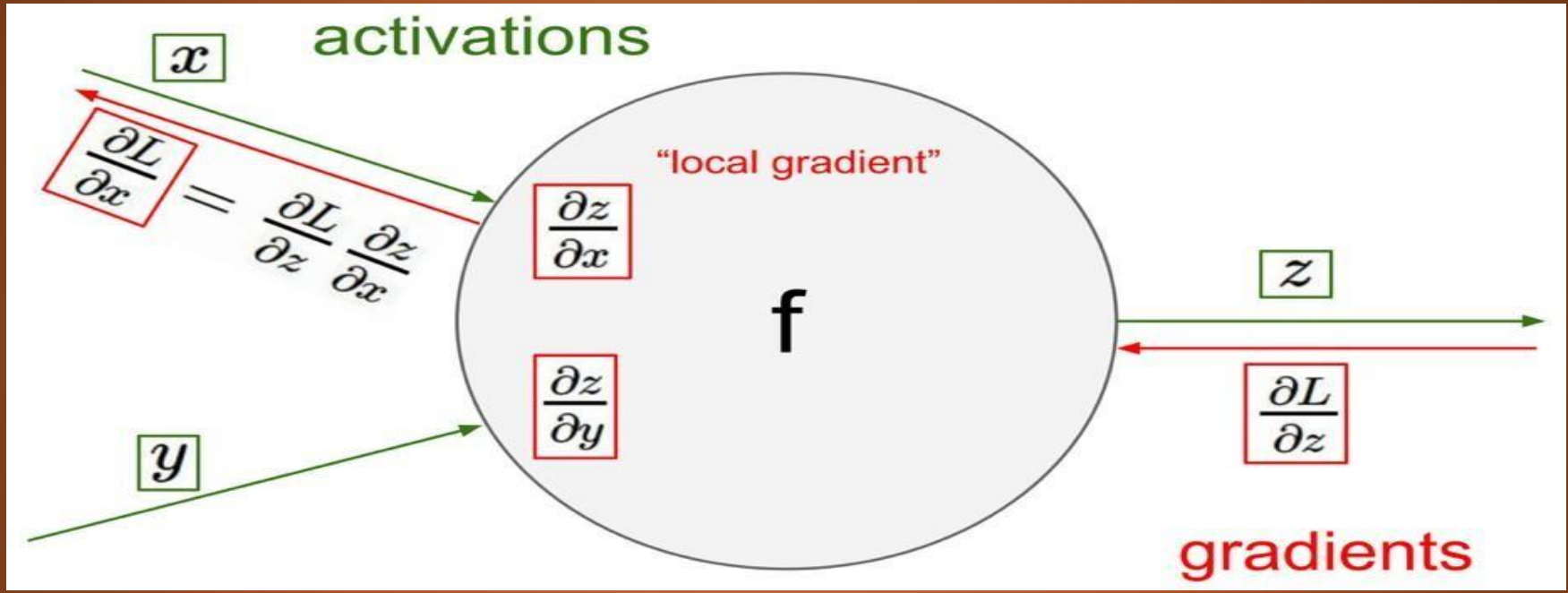
Backpropagation



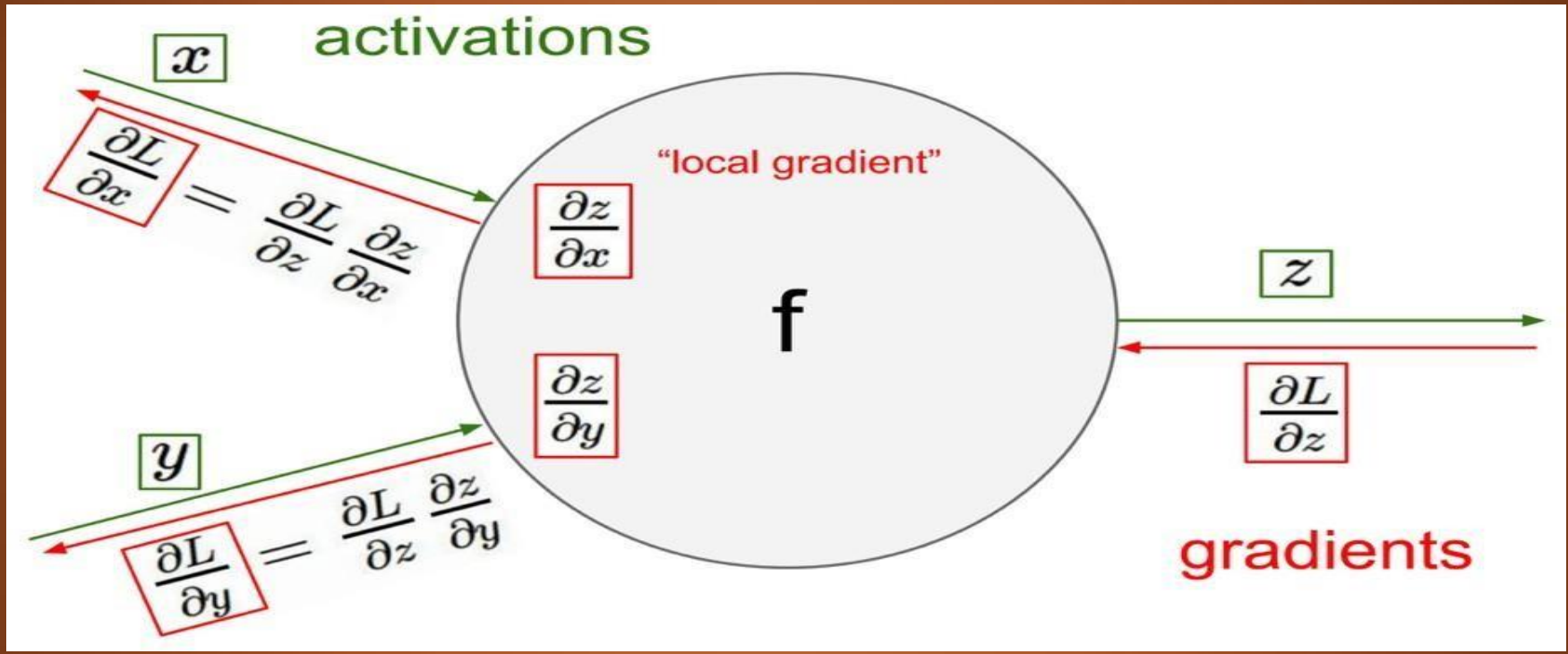
Backpropagation



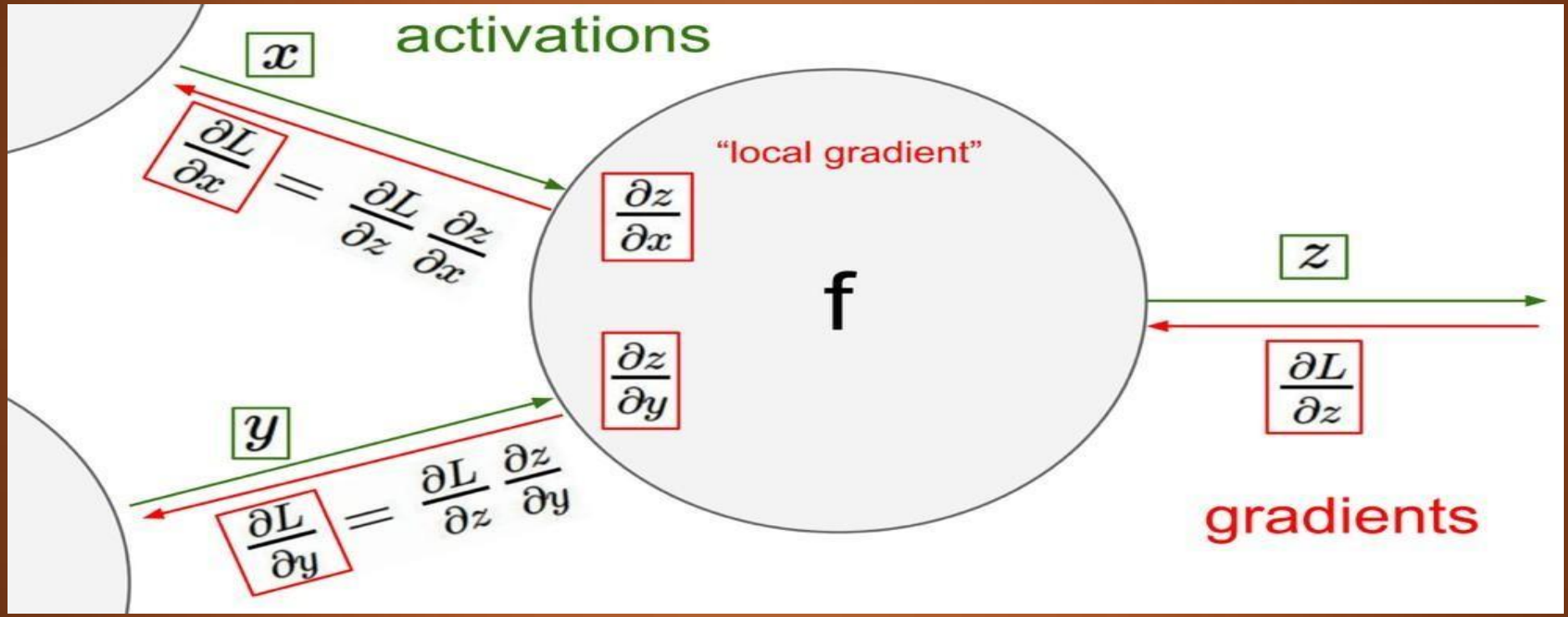
Backpropagation



Backpropagation

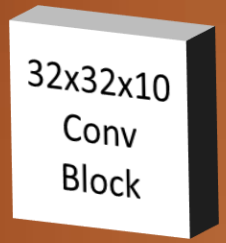


Backpropagation



Simple NN-based Classification

Input Image



Feature Extractor

$\hat{y} = SM(Wx)$

$argmax$

Classifier

c_{pred}

Classification Output

y

Input Label

$Log\ likelihood$

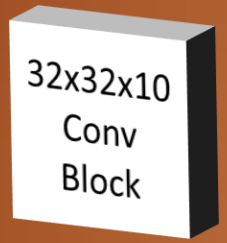
Cost Function

L

Loss Value

Simple NN-based Classification

Input Image



Feature Extractor

$\hat{y} = SM(Wx)$

$argmax$

Classifier

c_{pred}

Classification Output

y

Input Label

$Log\ likelihood$

Cost Function

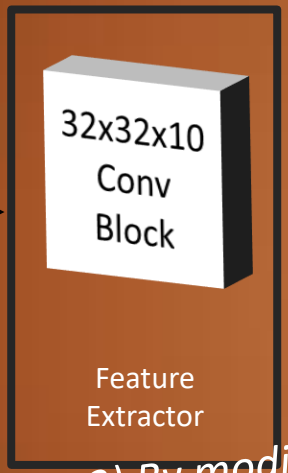
1) Minimize this...

L

Loss Value

Simple NN-based Classification

Input Image



Feature Extractor

$\hat{y} = SM(Wx)$

$argmax$

Classifier

c_{pred}

Classification Output

2) By modifying this…

1) Minimize this…

y

Input Label

$Log\ likelihood$

Cost Function

L

Loss Value

Simple NN-based Classification

Input Image

32x32x10 Conv Block

Feature Extractor

W

Wx

Classifier

c_{pred}

Classification Output

BUT DOES IT WORK?

1) Minimize this...

y

Input Label

Log likelihood

Cost Function

L

Loss Value

*3) Using **gradient descent & backprop!***

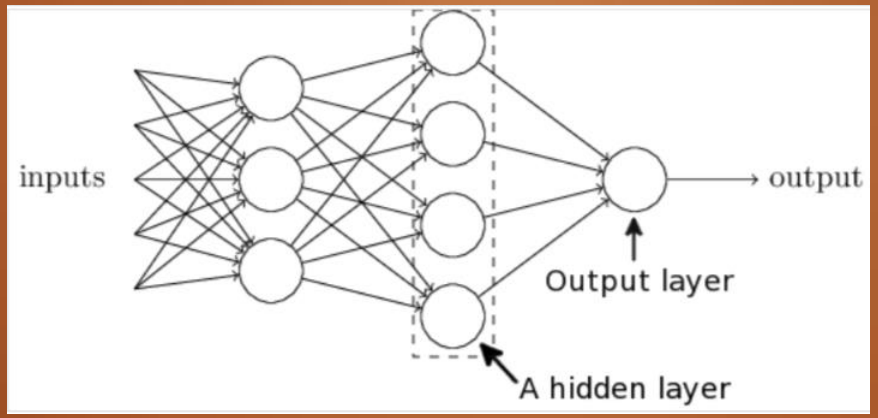
System’s Performance

- Moderate accuracy which could be improved by tuning the hyperparameters & other optimizations
- What about the filters? What do they look like?

Improving NN's performance

There are a lot of techniques, tricks, and best practices (some based on theory, some on empirical trial and error)

1. Add intermediate/hidden layers:



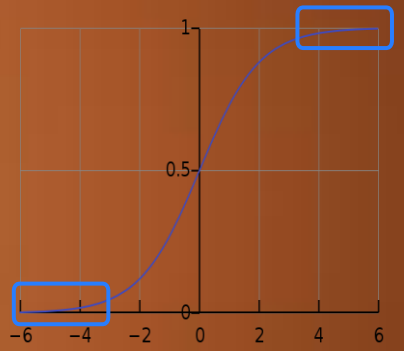
Hidden layer: Making a decision at a more abstract level by weighing up the results of the first layer.

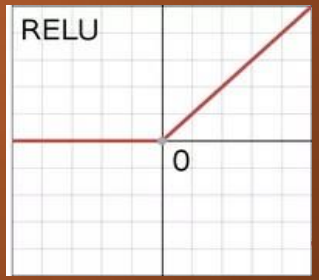
Improving NN's performance

There are a lot of techniques, tricks, and best practices (some based on theory, some on empirical trial and error)

2. Activation functions:

- Sigmoid function not used as much because values too close to 0 or 1 result in gradients too close to 0 stopping backpropagation.
- ReLU is one of the more popular ones because its simple to compute and robust
- Softmax is used at the last/output layer

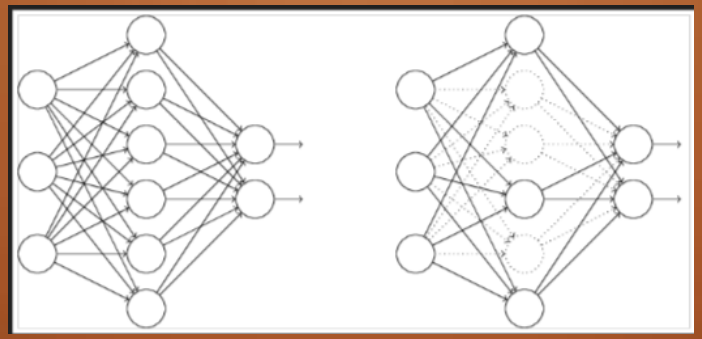




Improving NN's performance

3. REGULARIZATION to reduce over-fitting to the training data.

- L2: minimize $C + \frac{\lambda}{2n} \sum_w w^2.$
- L1: minimize $C + \frac{\lambda}{n} \sum_w |w|.$
- Dropout: Within every iteration of backprop, randomly and temporarily delete some neurons.



Improving NN’s performance

4. INCREASE TRAINING DATA

- Expensive and effortful to create larger dataset
- Data augmentation helps: slightly altering (rotating, translating, skewing etc.) to increase the number of samples

Drawbacks of traditional NN based classifiers

- Effective training was limited to a small number of layers and nodes per layer due to huge parameters and problems like vanishing gradient etc.
- Their architecture does not take into account spatial structure of the images.
- Did not scale well to larger images.
- Training could end up focusing on local minima (unreliable)

Convolutional Neural Networks

CNNs depart from regular ANNs in following ways:

1. Adapted to preserve/take advantage of spatial structure in images

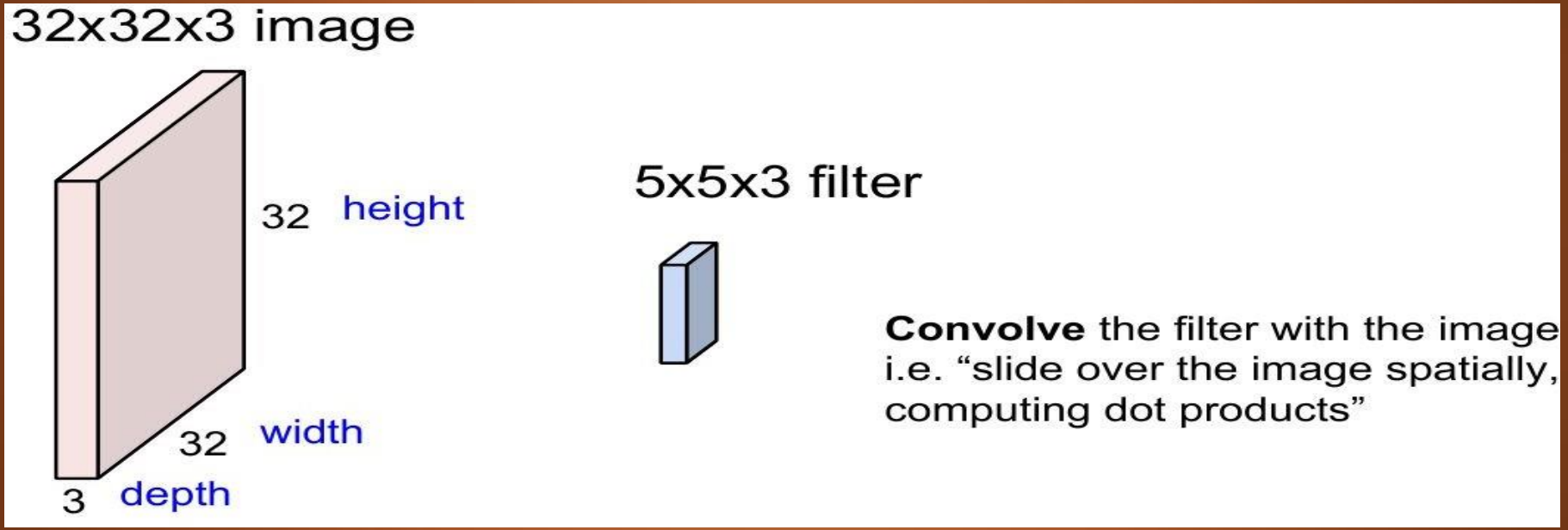
Input to Convolutional Neural Networks

Convolutional Neural Networks

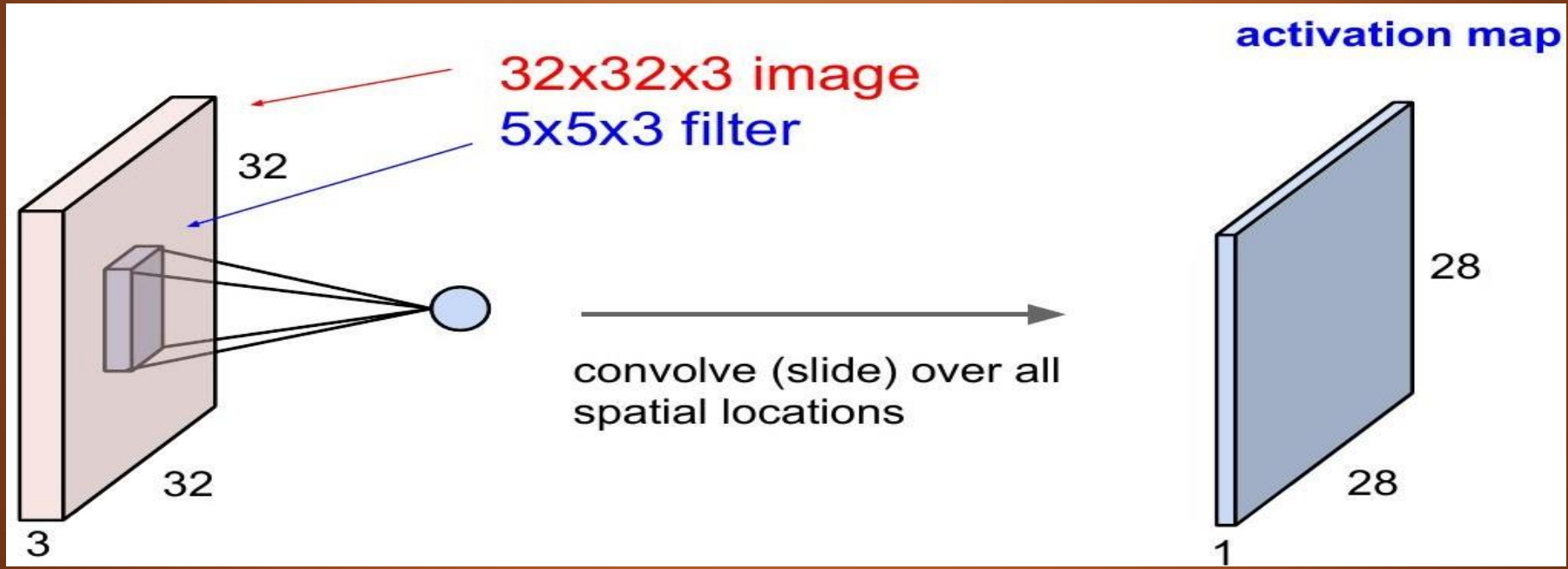
CNNs depart from regular ANNs in following ways:

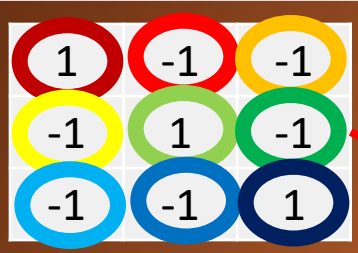
1. Adapted to preserve/take advantage of spatial structure in images
2. Convolution layer neurons have local connectivity (with overlapping receptive/ input fields) and share same weight parameters across the whole layer.

Convolution Layer



Convolution Layer





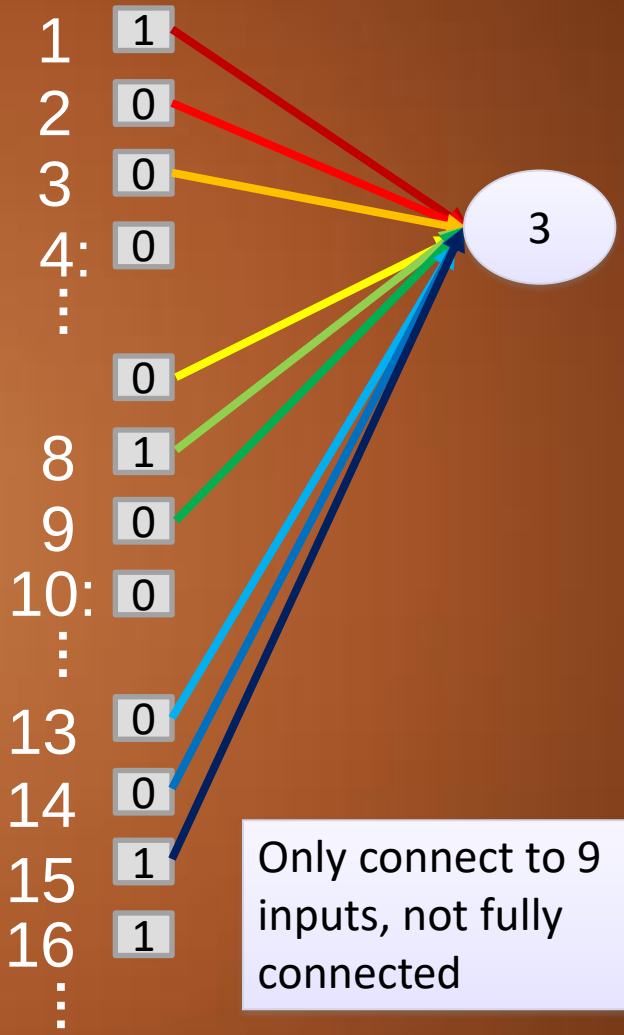
Filter 1

1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	
0	1	0	0	1	0
0	0	1	0	1	0

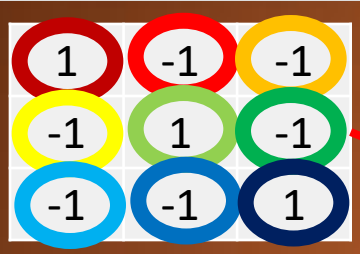
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

6 x 6 image

fewer parameters!



Only connect to 9 inputs, not fully connected



Filter 1

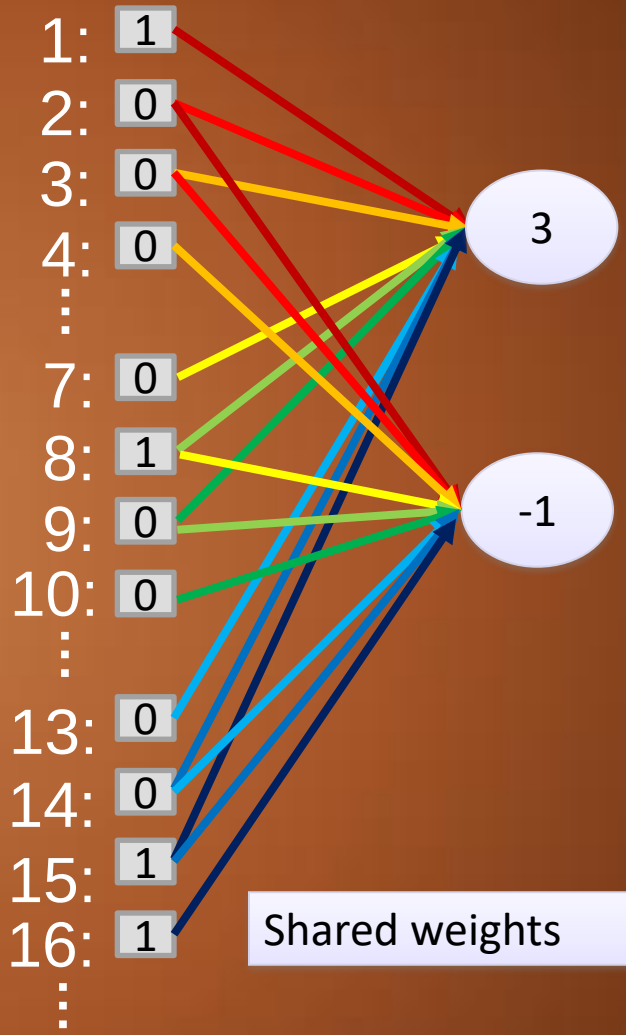
1	0	0	0	0	
0	1	0	0	1	
0	0	1	1	0	
1	0	0	0	1	
0	1	0	0	1	0
0	0	1	0	1	0

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

6 x 6 image

Fewer parameters

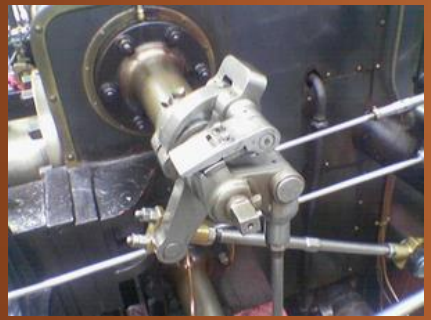
Even fewer parameters



Shared weights

Why Convolution?

Allow us to find **interesting insights/features** from images!



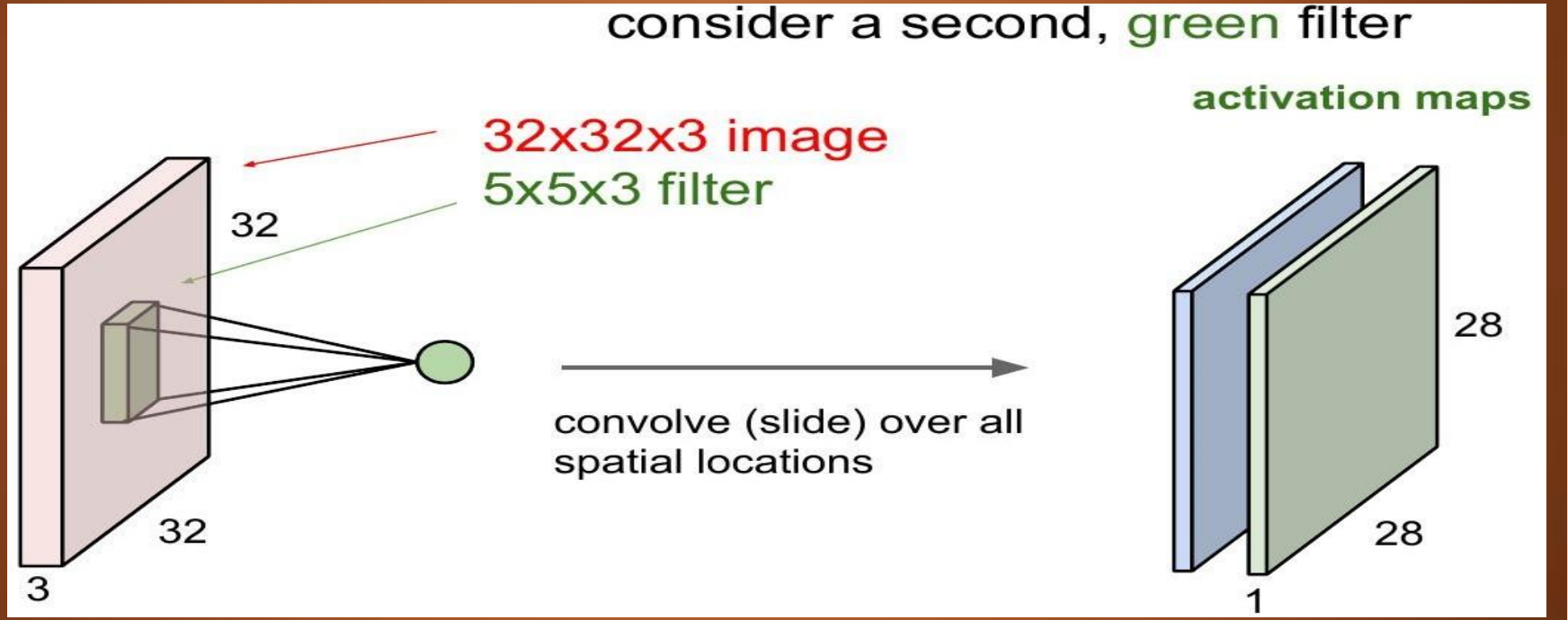
*

0	-½	0
0	0	0
0	½	0

=

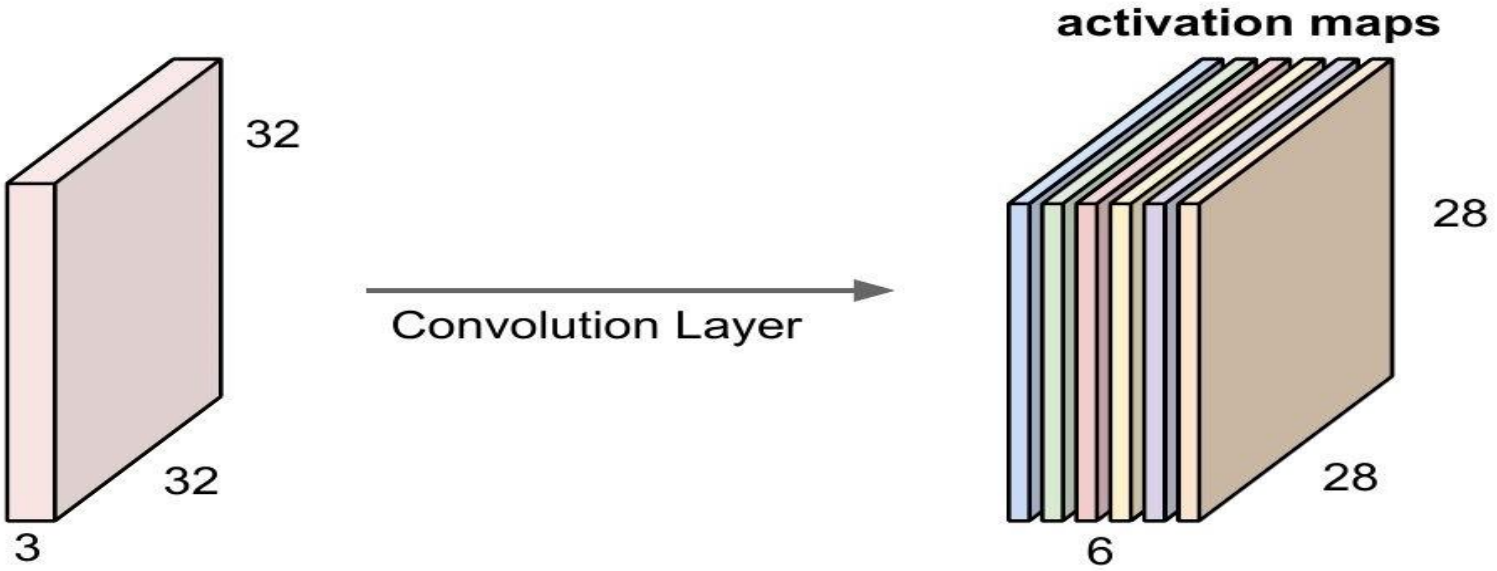
More Convolutions = More Insights!

Convolution Layer



Convolution Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

Convolutional Neural Networks

CNNs depart from regular ANNs in following ways:

1. Adapted to preserve/take advantage of spatial structure in images
2. Convolution layer neurons have local connectivity (with overlapping receptive/ input fields) and share same weight parameters across the whole layer.
3. Stacking of multiple convolutional layers to extract features at different abstraction level

Recall Hubel and Weisel…

The thing has edges…

The edges can be grouped into triangles and ovals…

The triangles are ears, the oval is a body…

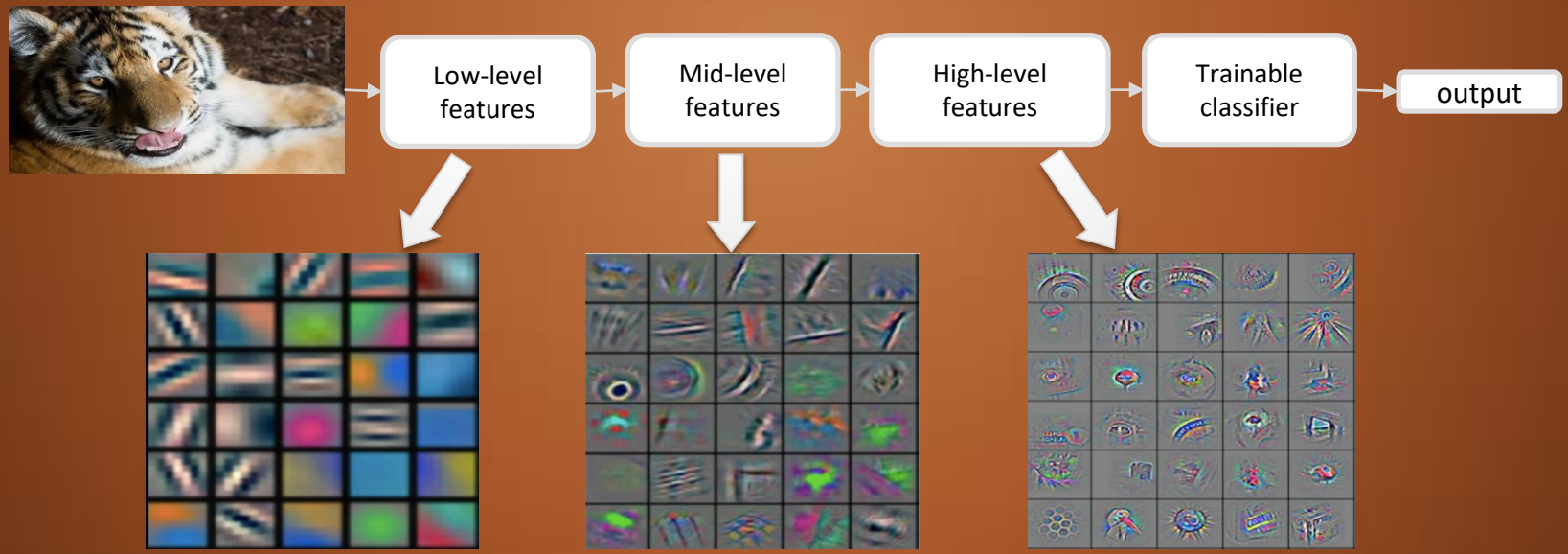
It's a mouse toy!

Learning Hierarchical Representations

Hierarchy of representations with increasing level of abstraction

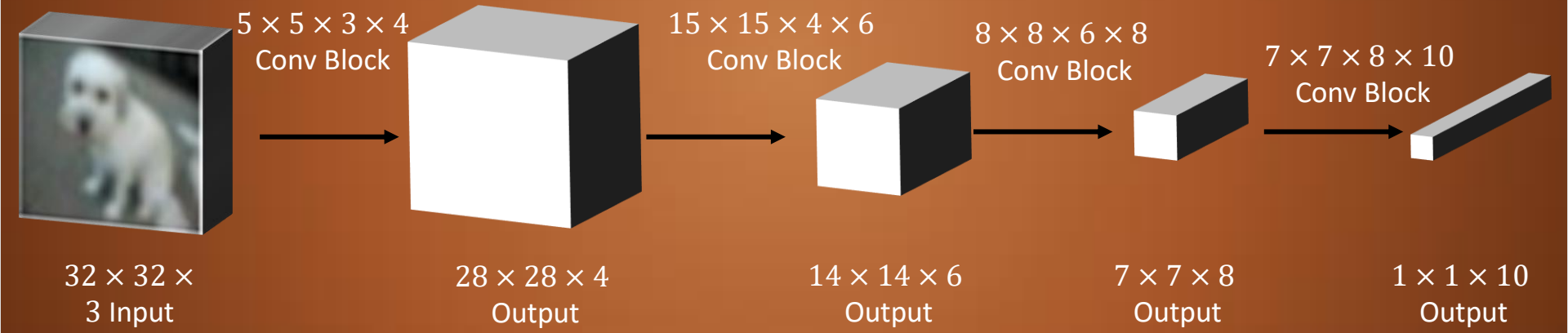
Image recognition

Pixel → edge → texton → motif → part → object



Feature visualization of convolutional net trained on ImageNet (Zeiler and Fergus, 2013)

Stacking Convolutions



Convolutional Neural Networks

CNNs depart from regular ANNs in following ways:

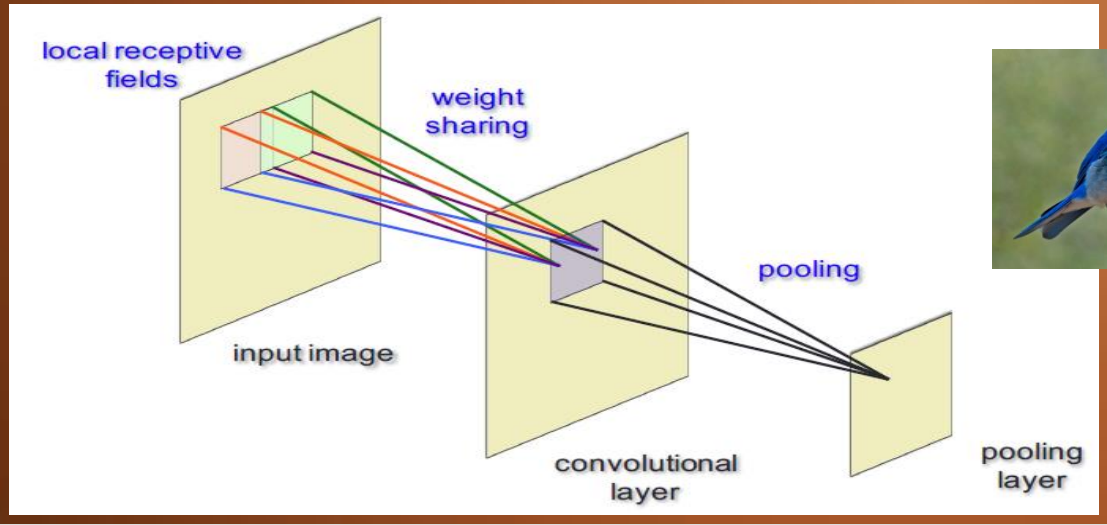
1. Adapted to preserve/take advantage of spatial structure in images
2. Convolution layer neurons have local connectivity (with overlapping receptive/ input fields) and share same weight parameters across the whole layer.
3. Stacking of multiple convolutional layers to extract features at different abstraction level
4. Intersperse convolution layers with subsampling or “pooling” layers.

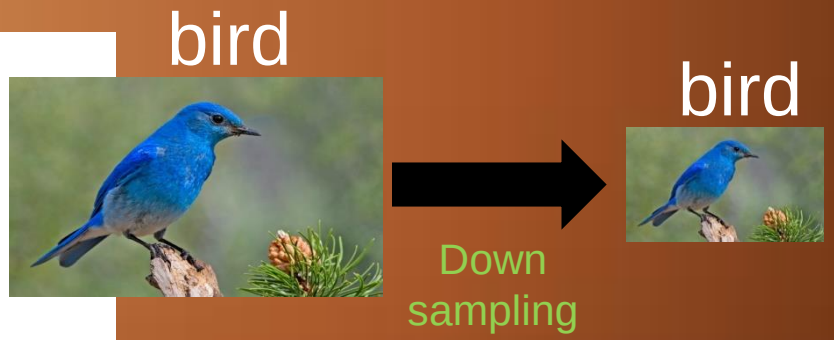
Pooling

- Why? Further reduces complexity - fewer parameters to characterize the image
- Common pooling operations:

  Max pooling: reports the maximum output within a rectangular neighborhood.

  Average pooling: reports the average output of a rectangular neighborhood (possibly weighted by the distance from the central pixel).



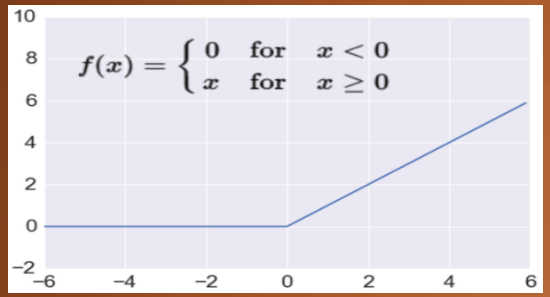


Convolutional Neural Networks

CNNs depart from regular ANNs in following ways:

1. Adapted to preserve/take advantage of spatial structure in images
2. Convolution layer neurons have local connectivity (with overlapping receptive/ input fields) and share same weight parameters across the whole layer.
3. Stacking of multiple convolutional layers to extract features at different abstraction level
4. Intersperse convolution layers with subsampling or “pooling” layers.
5. Use of rectified linear unit (ReLU) as activation function

Activation: ReLU



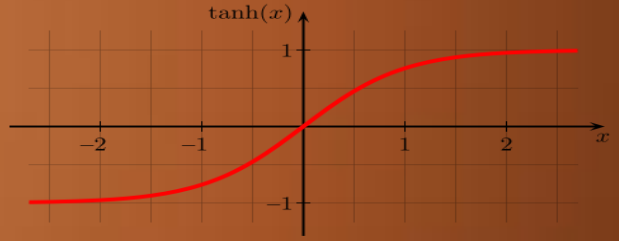
$$f(x) = \max(0, x)$$

Takes a real-valued number and thresholds it at zero

$$R^n \rightarrow R^n_+$$

Most Deep Networks use ReLU nowadays:

- Trains much **faster**
  - accelerates the convergence of SGD
  - due to linear, non-saturating form
- Less expensive operations
  - compared to sigmoid/tanh (exponentials etc.)
  - implemented by simply thresholding a matrix at zero



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

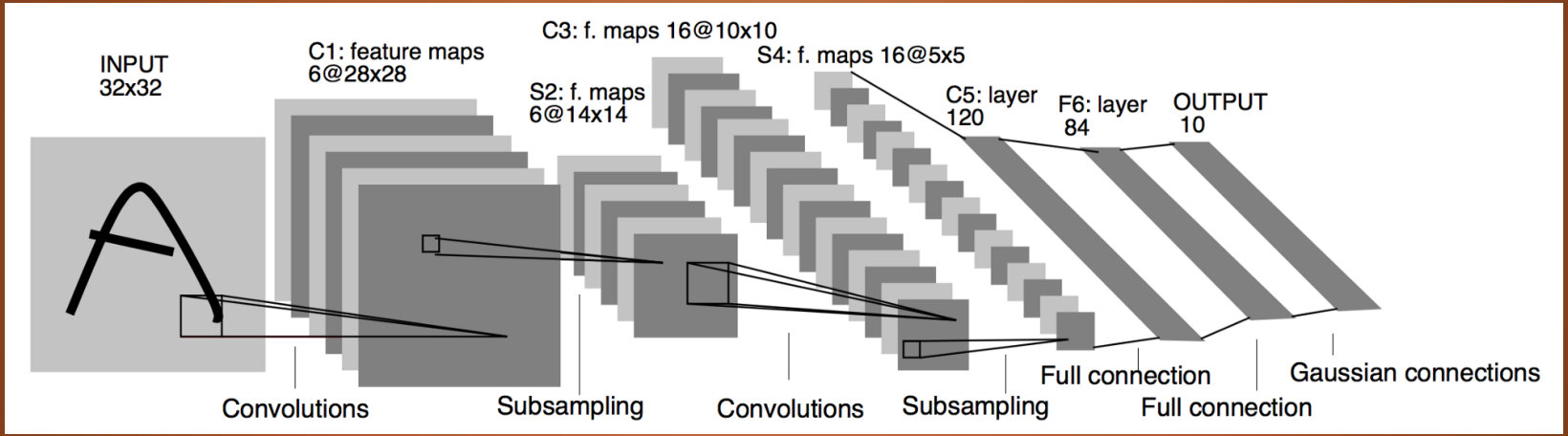
Convolutional Neural Networks

CNNs depart from regular ANNs in following ways:

1. Adapted to preserve/take advantage of spatial structure in images
2. Convolution layer neurons have local connectivity (with overlapping receptive/ input fields) and share same weight parameters across the whole layer.
3. Stacking of multiple convolutional layers to extract features at different abstraction level
4. Intersperse convolution layers with subsampling or “pooling” layers.
5. Use of rectified linear unit (ReLU) as activation function

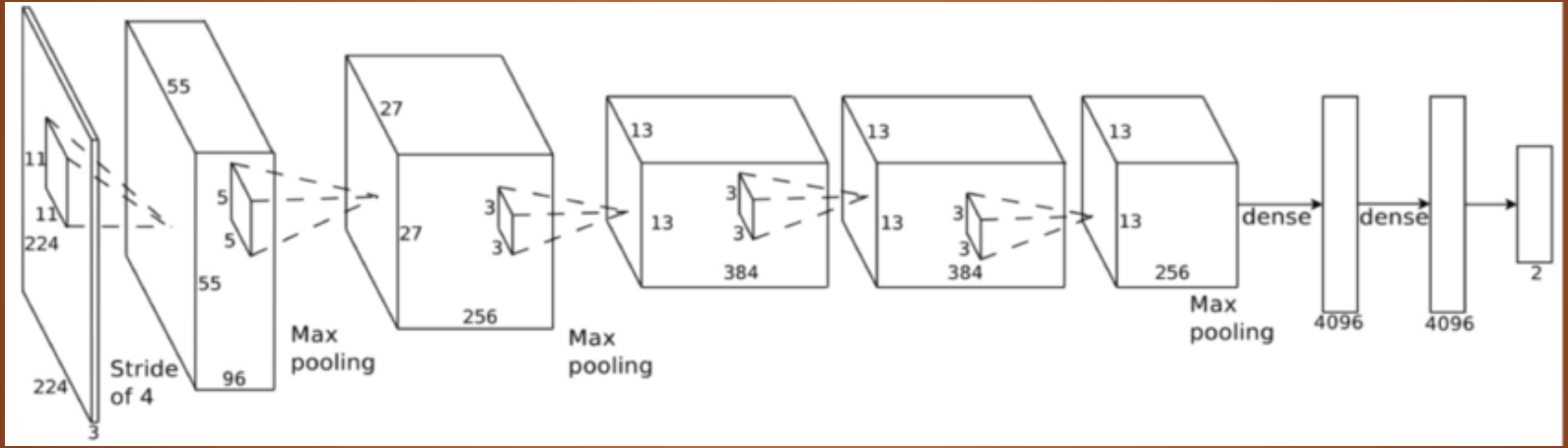
Often ends in fully-connected layer with softmax fn. as the “classifier”

History of ConvNets



LeNet – 1998. Built at NYU in yann lecunn's group.
Offer's near human performance > 99% accuracy on MNIST Dataset

History of ConvNets



AlexNet – 2012. Simplified ImageNet (1000 fine-grained classes)
16.4% (top5) test error rate (down from 26% test error rate) - big jump which attracted attention of people.