Artificial Neural Networks

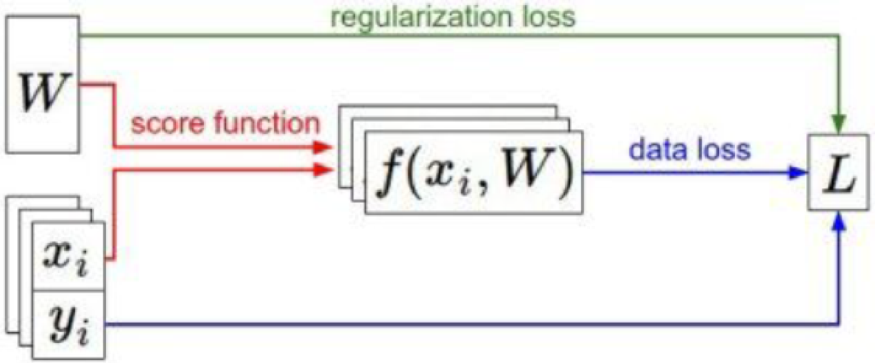
Recap

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \overset{\text{e.g.}}{=} Wx$
- We have a **loss function**:

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

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

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



Recap

Stochastic Gradient Descent (SGD)

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

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

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

The original linear classifier

- Linear score function $f = Wx$

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

Neural Network: 2 layers

- Linear score function $f = Wx$

- An example neural network would instead compute

$$f = W_2 * actv\_fn(W_1 x)$$

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

Neural Network: 2 layers

- Here, W_1 could be, for example, a $[100x3072]$ matrix transforming the image into a 100-dimensional intermediate vector.

- The function $actv\_fn$ is a non-linearity that is applied elementwise.

- There are several choices we could make for the non-linearity.

- Finally, the matrix W_2 would then be of size $[10x100]$, so that we again get 10 numbers out that we interpret as the class scores.

Neural Network: 2 layers

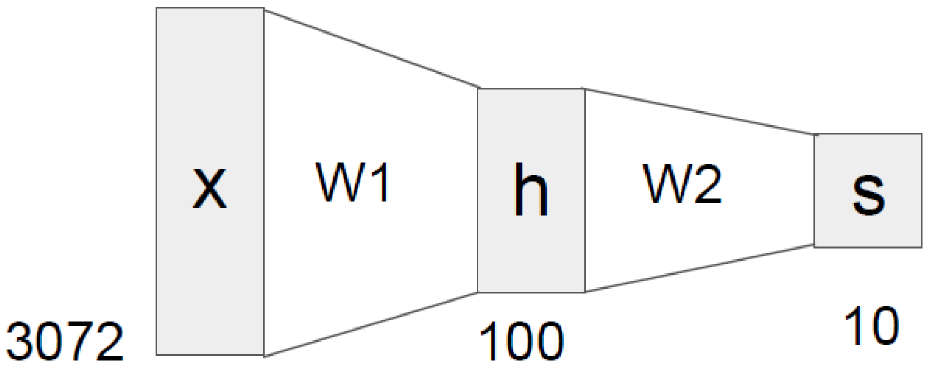
- Linear score function $f = Wx$
- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$

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

Neural Network: hierarchical computation

- Linear score function $f = Wx$

- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$



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

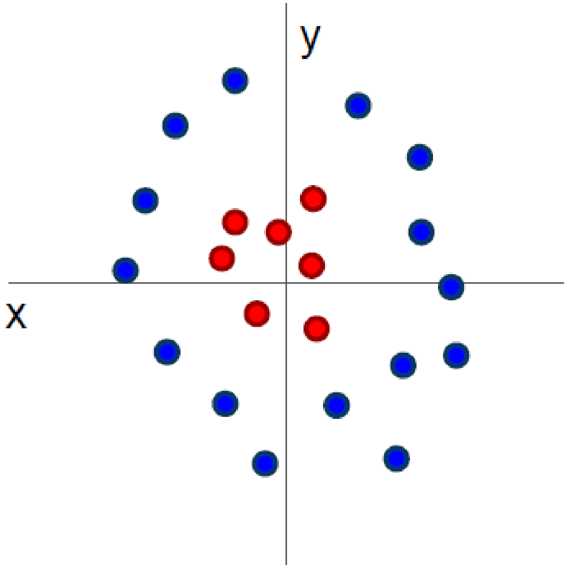
Neural Network: 2 layers

- Linear score function $f = Wx$

- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$

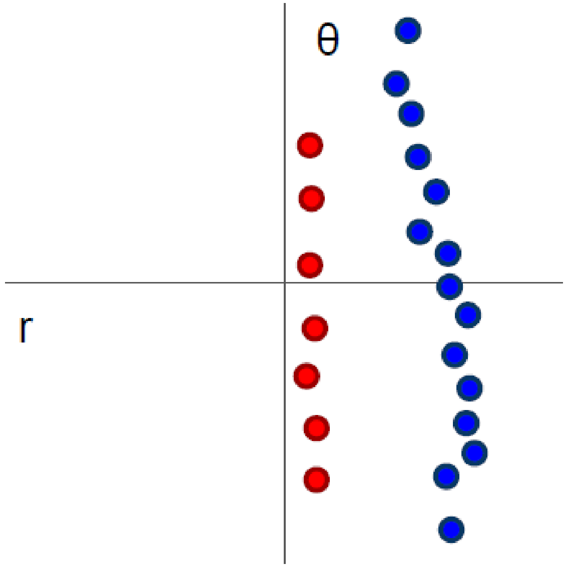
Activation function to introduce non-linearity

Why do we want non-linearity?



Cannot separate red and blue points with linear classifier

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



After applying feature transform, points can be separated by linear classifier

Neural Network: 2 layers

- Linear score function $f = Wx$
- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$

"Neural Network" is a very broad term;

these are more accurately called "fully-connected networks" or sometimes "multi-layer perceptrons" (MLP)

The original linear classifier

- Linear score function $f = Wx$
- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$
- 3-layer neural network $f = W_3 * actv\_fn(W_2 * actv\_fn(W_1 x))$

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

- …

Neural Network: 2 layers

- Linear score function $f = Wx$
- 2-layer neural network $f = W_2 * actv\_fn(W_1 x)$

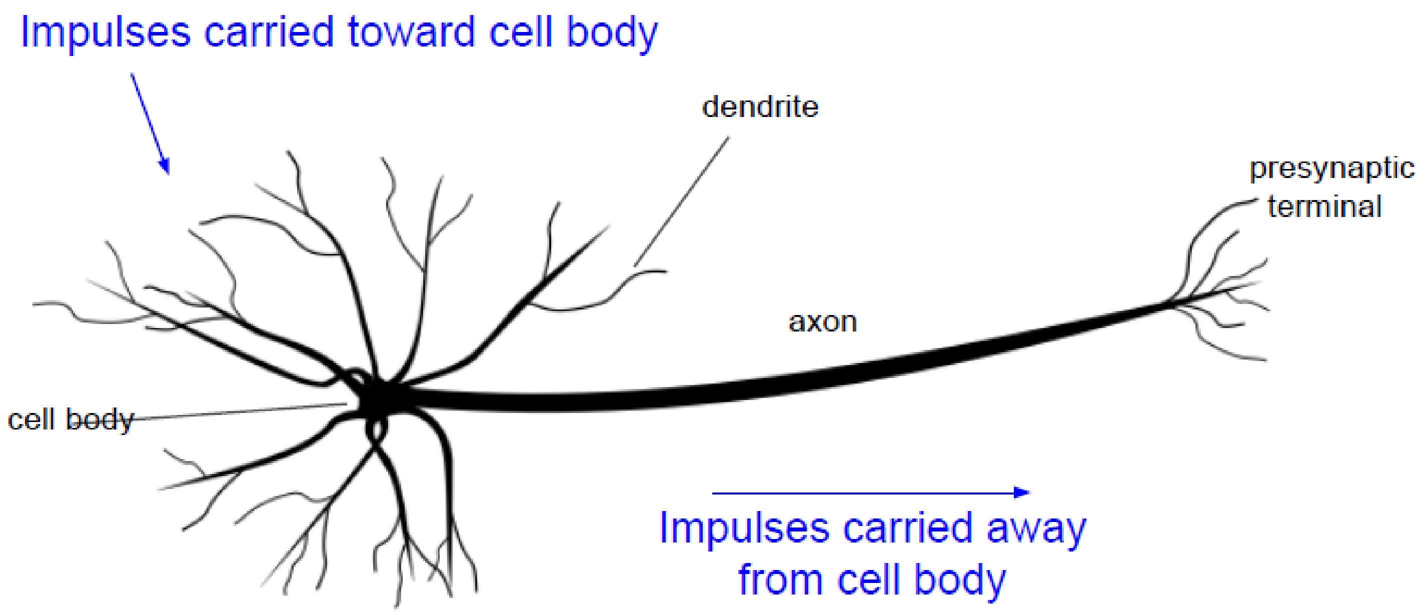
Q: What if we try to build a neural network without an activation function ?

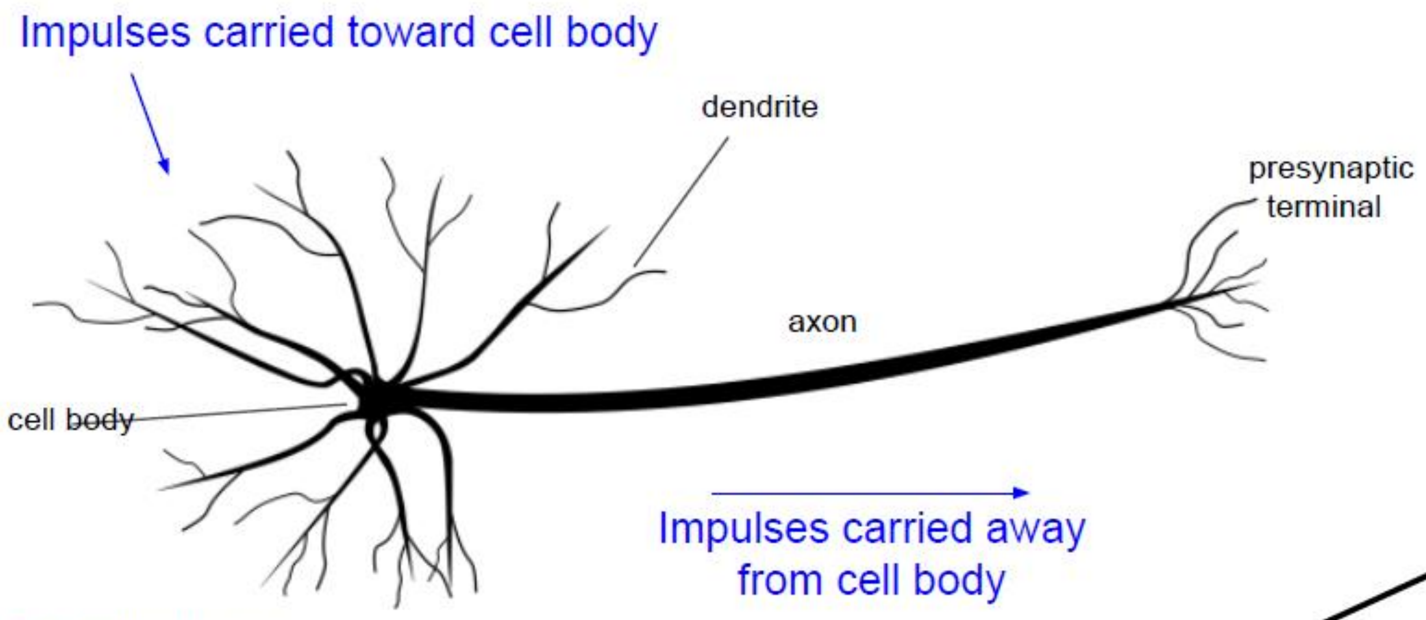
$$f = W_2 \, W_1 x$$

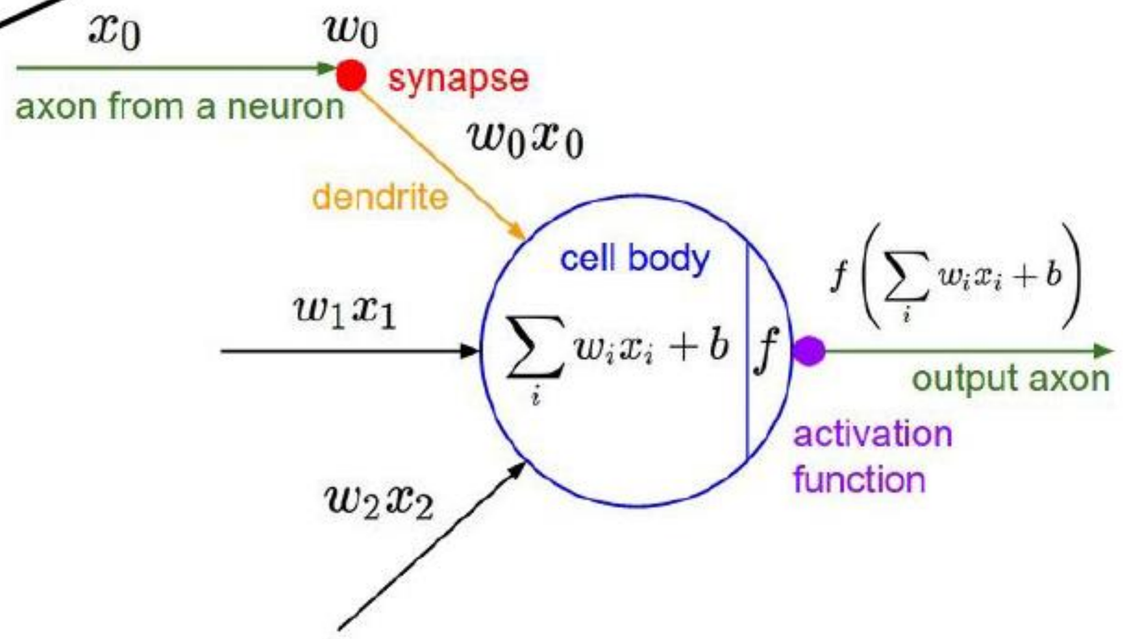
We end up with a linear classifier again!

Inspiration: The Brain

- Many machine learning methods inspired by biology, e.g., the (human) brain


Impulses carried toward cell body
dendrite
presynaptic
terminal
axon
cell body
Impulses carried away
from cell body


Impulses carried toward cell body
dendrite
presynaptic terminal
axon
cell body
Impulses carried away from cell body

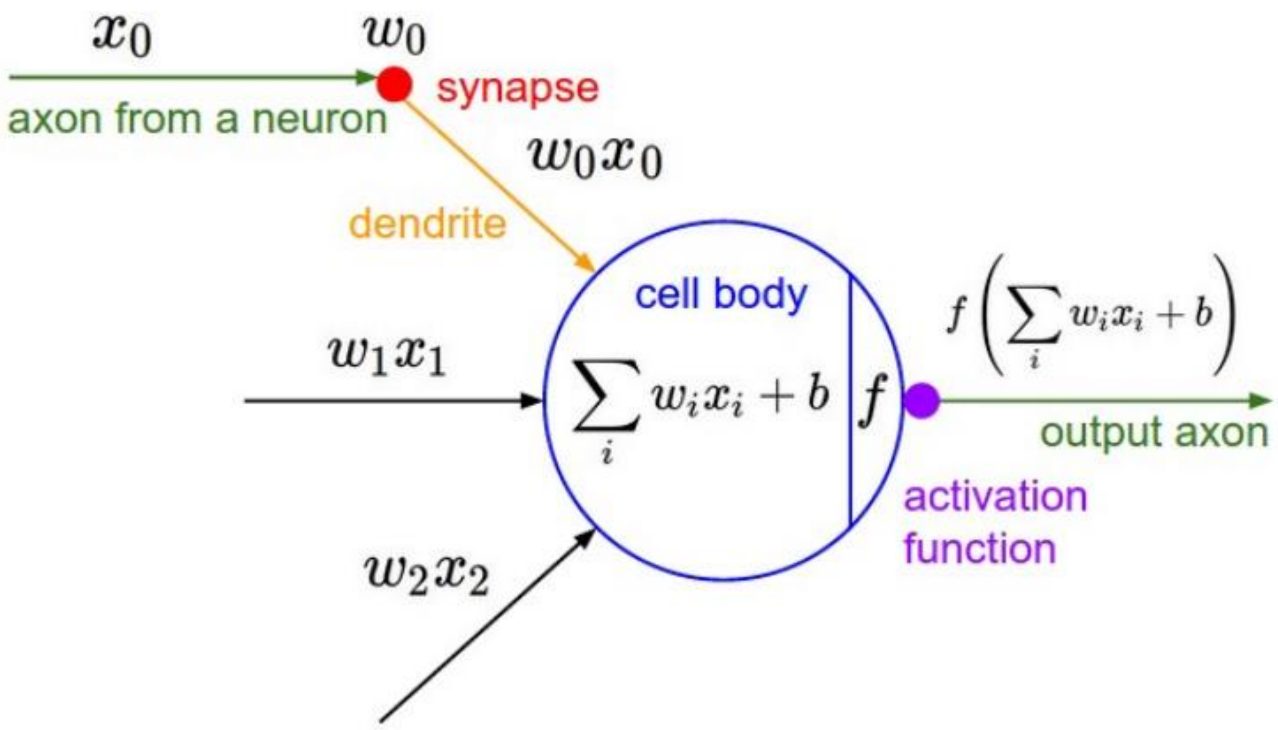

x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$f\left(\sum_i w_i x_i + b\right)$
$w_1 x_1$
$\sum_i w_i x_i + b$
f
output axon
activation function
$w_2 x_2$

Be very careful with your brain analogies!

- Biological Neurons:
  - Many different types
  - Dendrites can perform complex non-linear computations
  - Synapses are not a single weight but a complex non-linear dynamical system

Artificial neural network

- Artificial neurons are called units



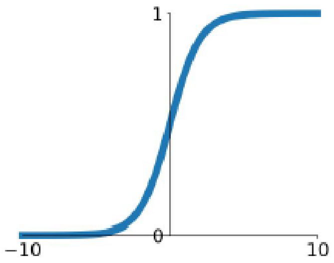
Activation function

- Activation functions are a crucial component of neural networks
- Activation functions introduce non-linearity into the model, enabling it to learn complex patterns and relationships in data.

Most commonly used activation functions

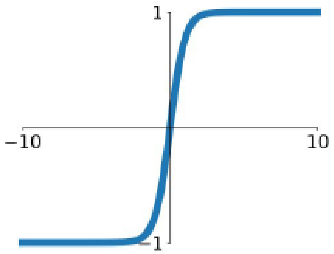
Sigmoid

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



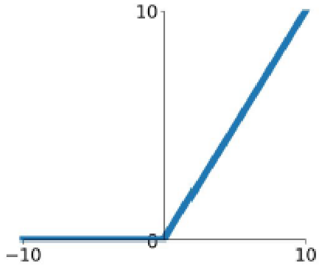
tanh

$\tanh(x)$



ReLU

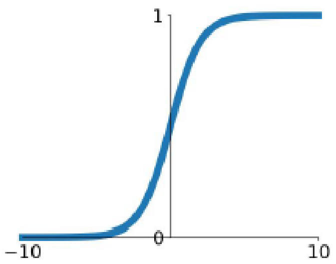
$\max(0, x)$



Most commonly used activation functions

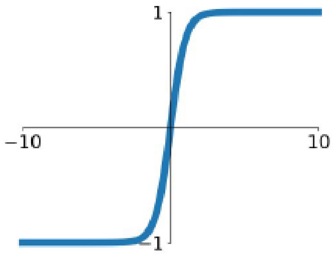
Sigmoid

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



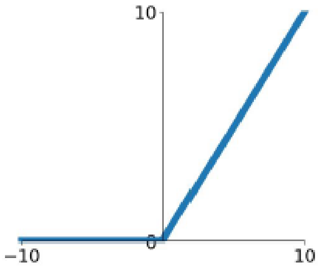
tanh

$\tanh(x)$



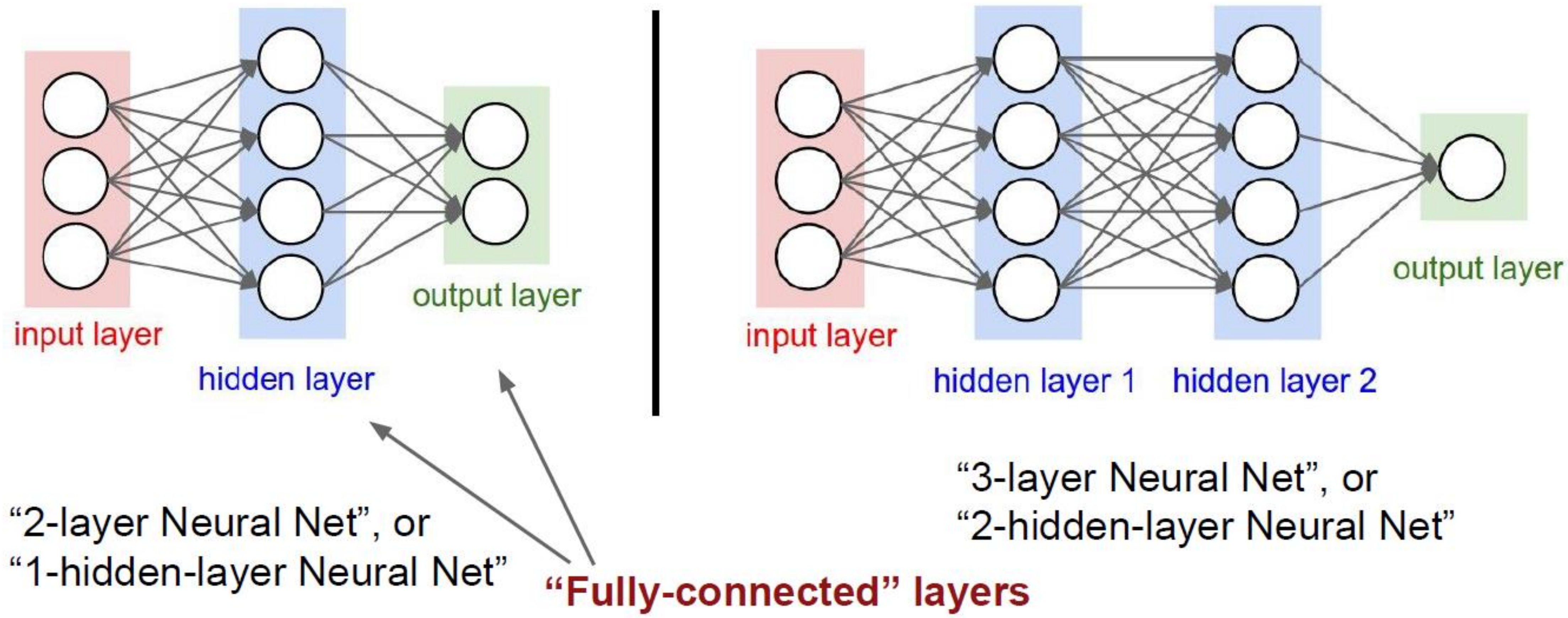
ReLU

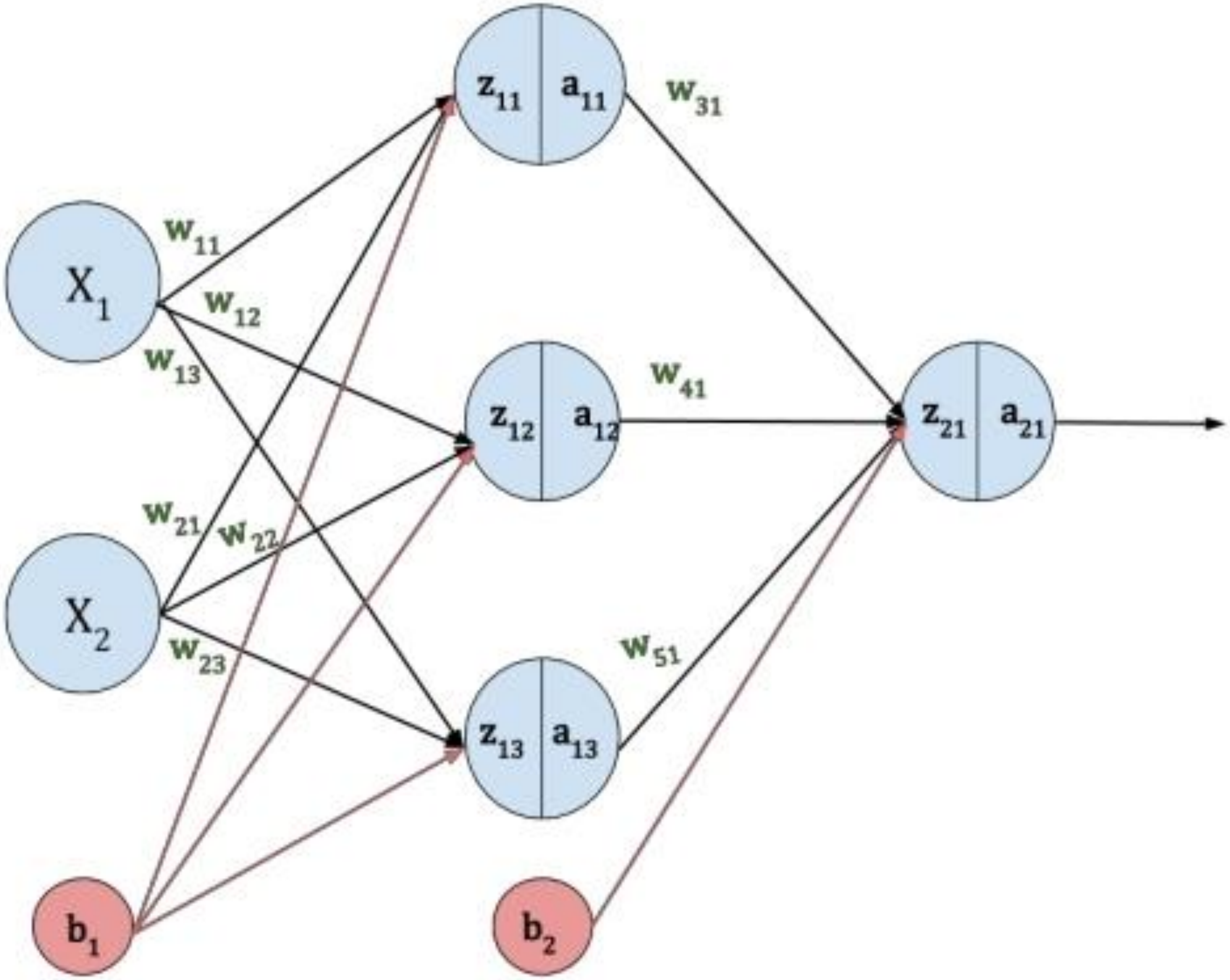
$\max(0, x)$



ReLU is a good default choice for most problems

Neural networks: Architectures





X1
X2
b1
b2
w11
w12
w13
w21
w22
w23
w31
w41
w51
z11 a11
z12 a12
z13 a13
z21 a21

Plugging in neural networks with loss functions

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x)$$ Nonlinear score function

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

$$R(W) = \sum_k W_k^2$$ Regularization

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$$ Total loss: data loss + regularization

If we can compute $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$ then we can learn W_1 and W_2

Derive $\nabla_W L$ on paper

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

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

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

Bad Idea

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2$$

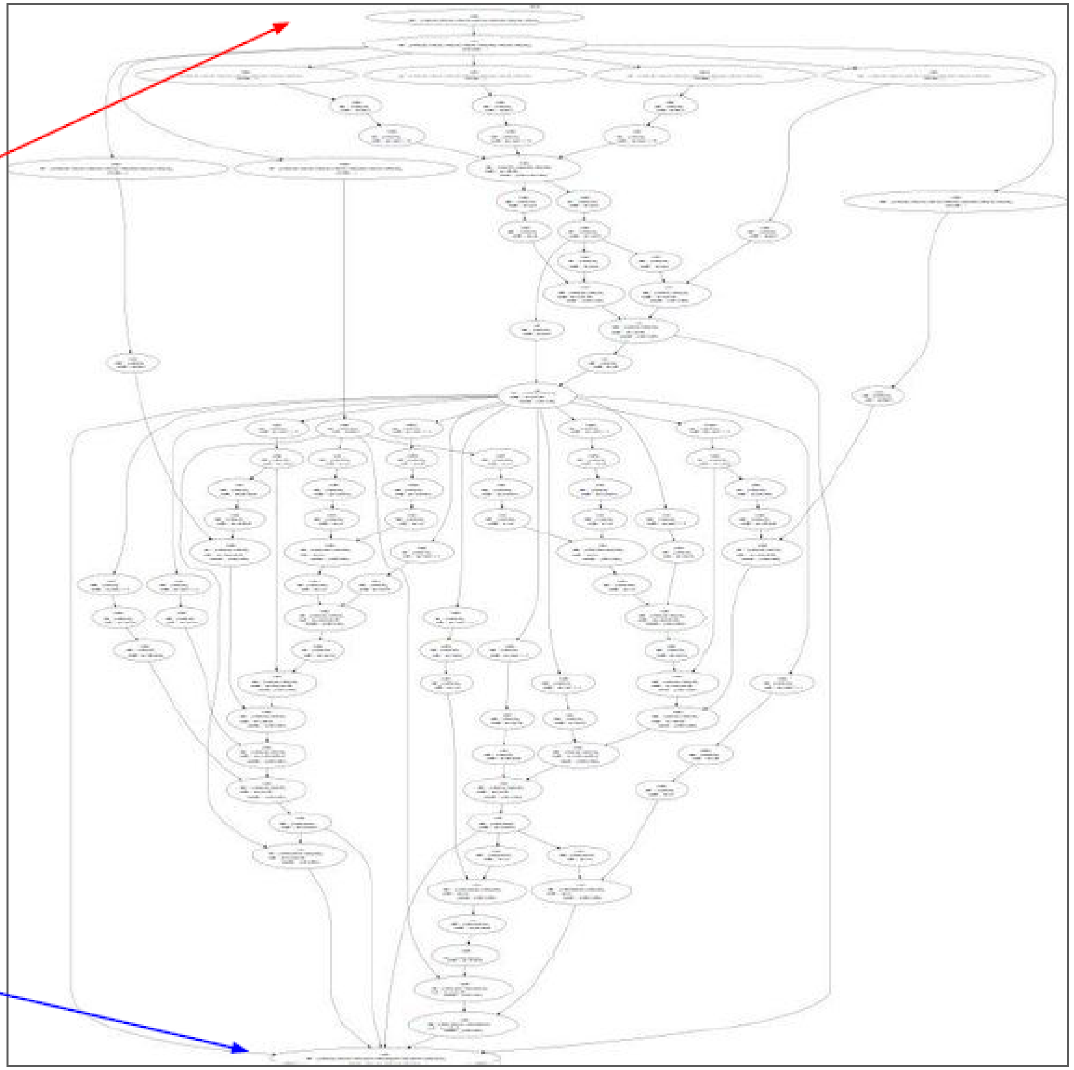
$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Really complex neural networks!!

input image

loss



Solution: Backpropagation

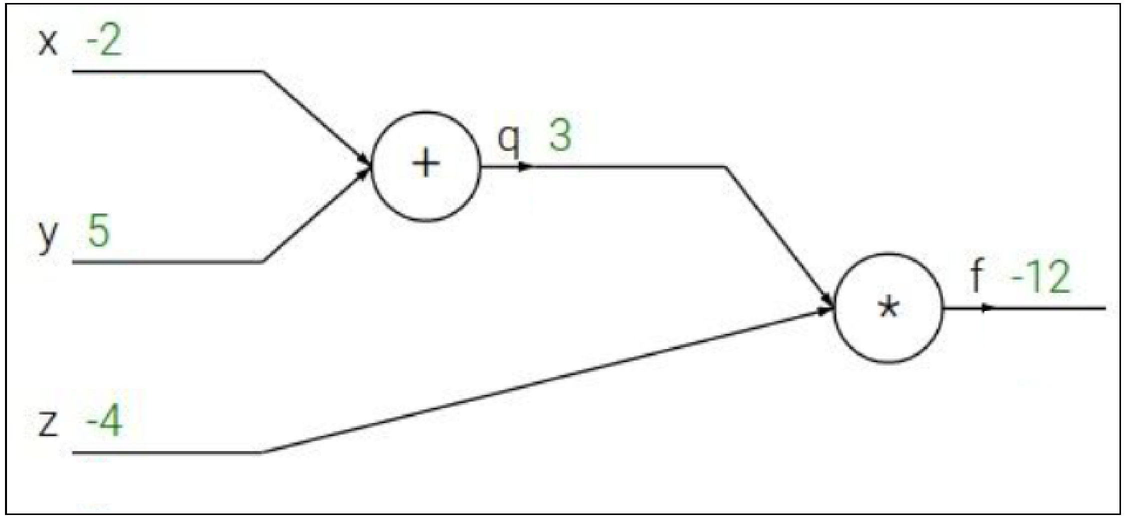
- An efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network.

- It's an optimization algorithm used to adjust the model's weights and biases during the training process, allowing the network to learn from data and improve its performance.

Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



Backpropagation: a simple example

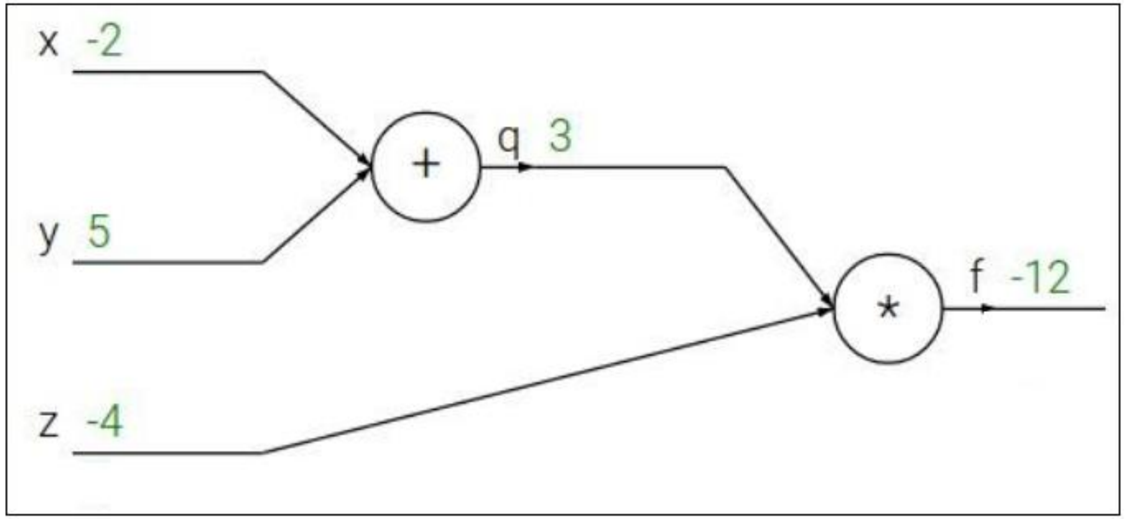
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

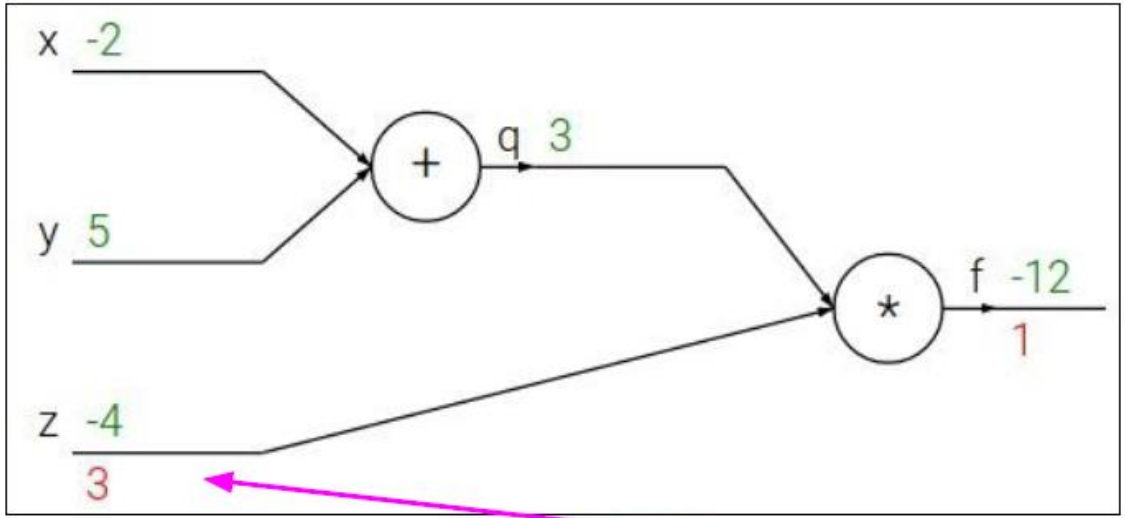
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

Backpropagation: a simple example

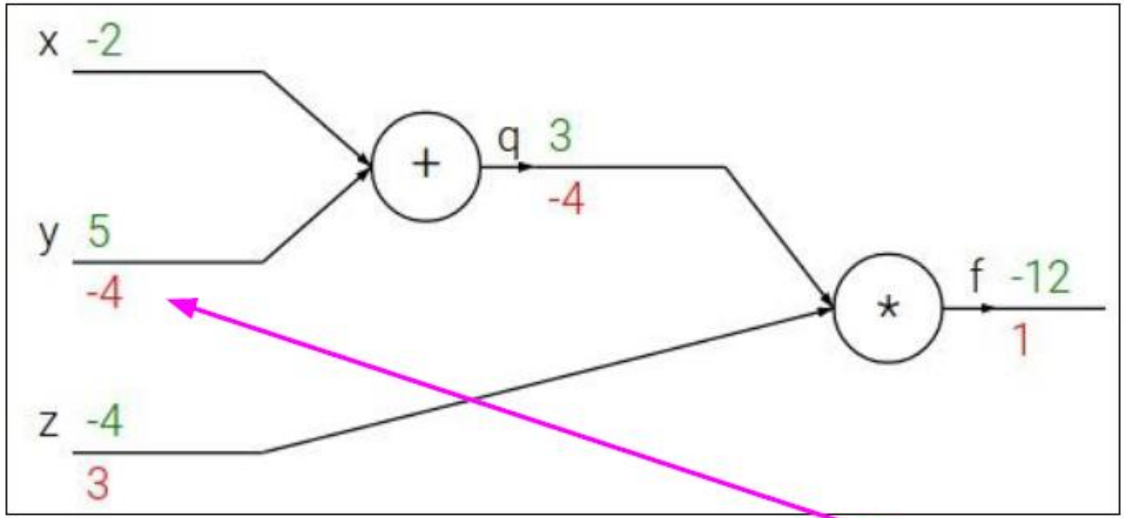
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$\boxed{q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1}$$

$$\boxed{f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q}$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\boxed{\frac{\partial f}{\partial y}}$$

Chain rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}}$$

Upstream gradient

Local gradient

Backpropagation: a simple example

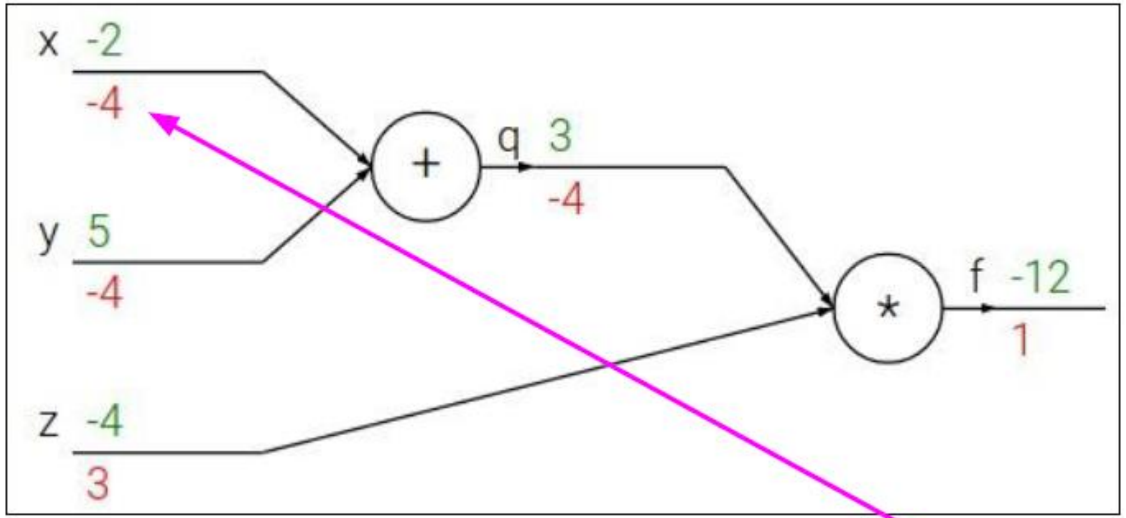
$$f(x,y,z) = (x+y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



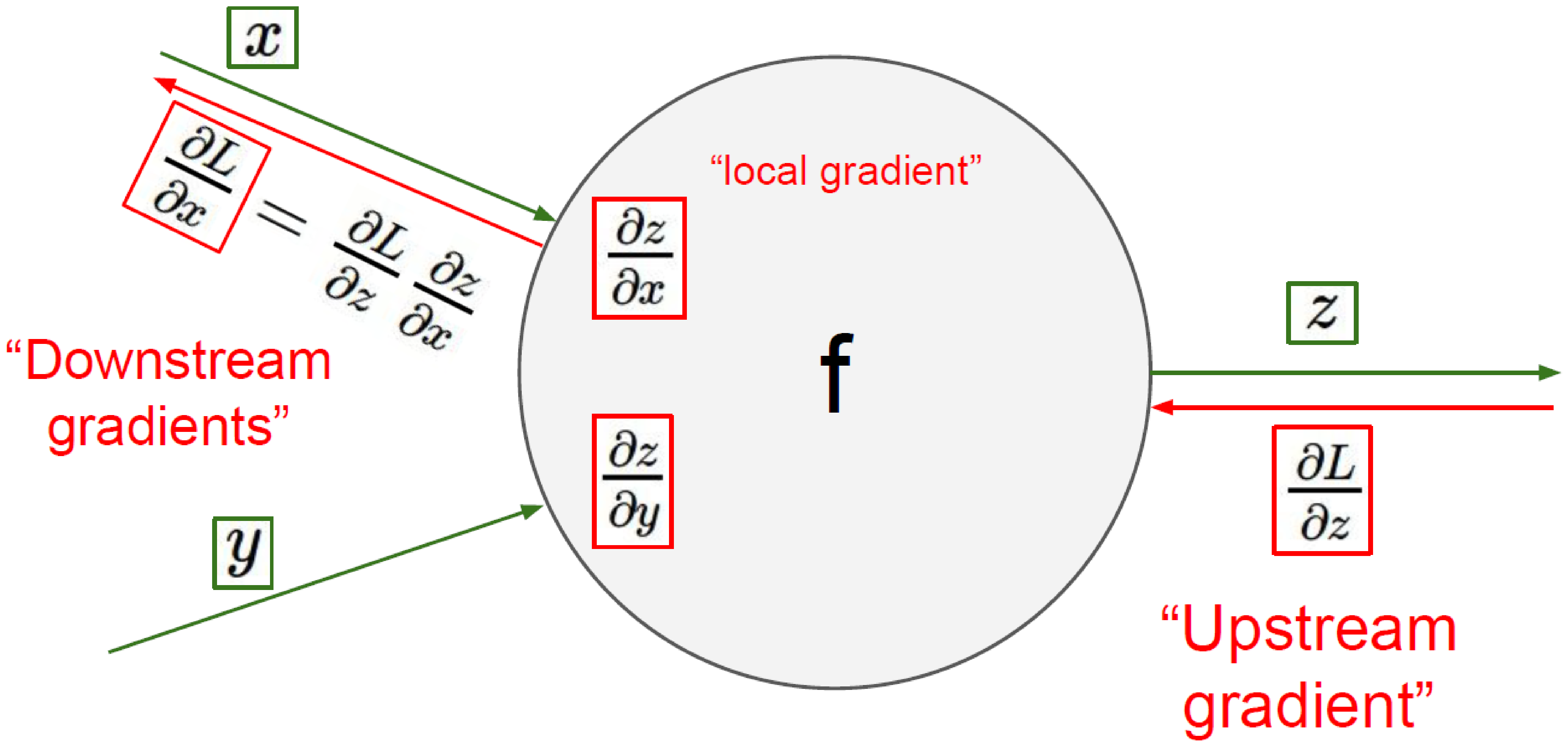
$$\frac{\partial f}{\partial x}$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Upstream gradient

Local gradient



x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
“Downstream gradients”
y
“local gradient”
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
f
z
$\frac{\partial L}{\partial z}$
“Upstream gradient”

So far …

- (Fully-connected) Neural Networks are stacks of linear functions and nonlinear activation functions; they have much more representational power than linear classifiers.

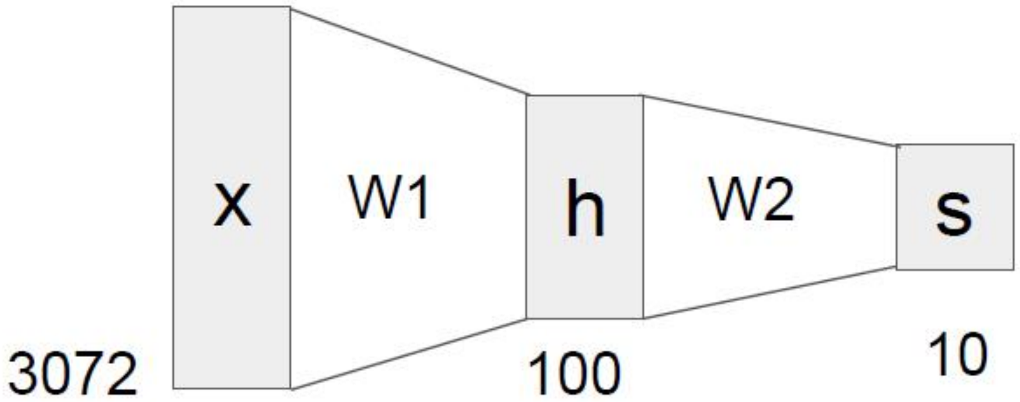
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates.

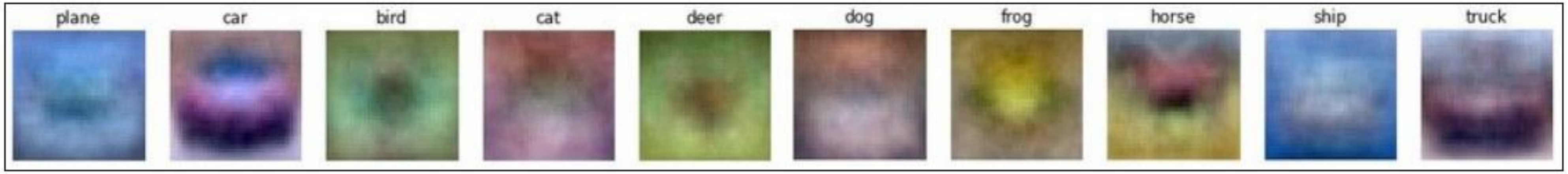
- **Forward** path: compute result of an operation and save any intermediates needed for gradient computation in memory.

- **Backward** path: apply the chain rule to compute the gradient of the loss function with respect to the inputs

```python
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers

# Load and preprocess the CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.cifar10.load_data()
X_train, X_test = X_train / 255.0, X_test / 255.0

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

print(X_train.shape)
print(y_train.shape)

# Create a simple feedforward neural network
model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape=(32, 32, 3)))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```python
model.summary()

# Train the model with a validation set
model.fit(X_train, y_train,
          batch_size=32,
          epochs=10,
          validation_data=(X_val, y_val))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy}")
```

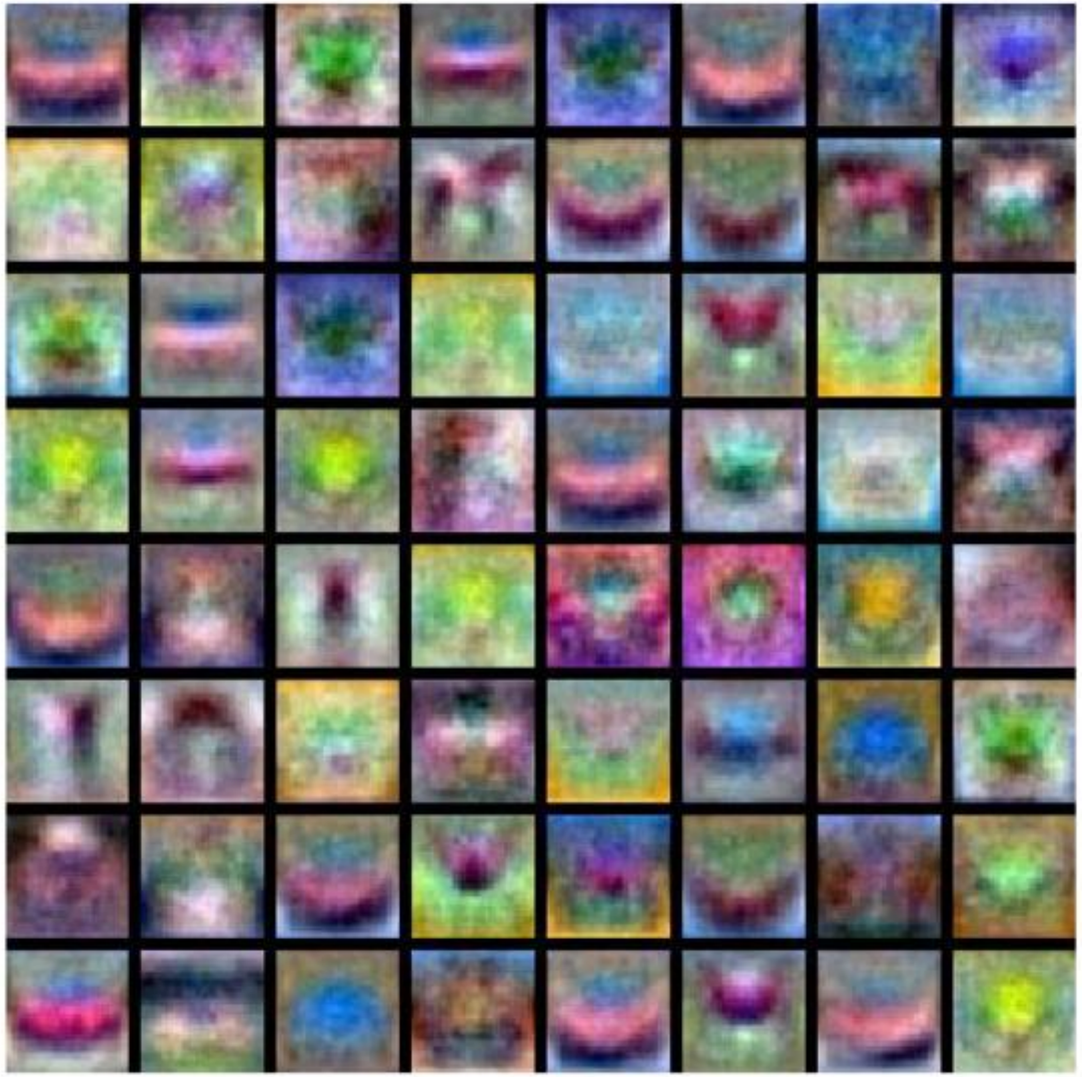




Learn 100 templates instead of 10.

Share templates between classes

MLP: Bank of whole-image templates



Some of the hyperparameters that can be changed

- Number of hidden layers and neurons per layer
- Activation functions (e.g., 'relu', 'tanh', 'sigmoid')
- Optimizer (e.g., 'adam', 'sgd')
- Learning rate
- Batch size
- Number of training epochs
- Loss function (e.g., 'sparse_categorical_crossentropy', 'categorical_crossentropy')
- Regularization techniques (e.g., dropout, L2 regularization)

Convolutional Neural Networks

Convolutional Neural Networks

- Convolutional Neural Networks are very similar to ordinary Neural Networks.
- They are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.
- The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.
- And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what changes?

- ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture.

- These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

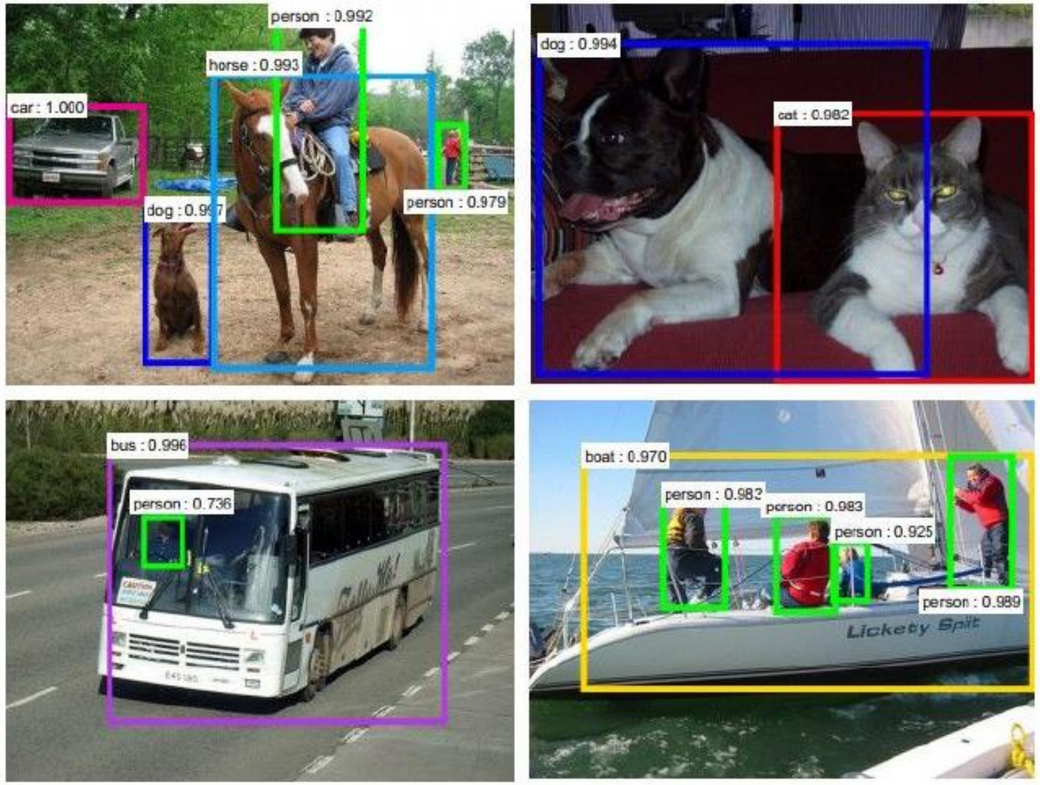
Convnets are everywhere

Classification

Retrieval

Convnets are everywhere

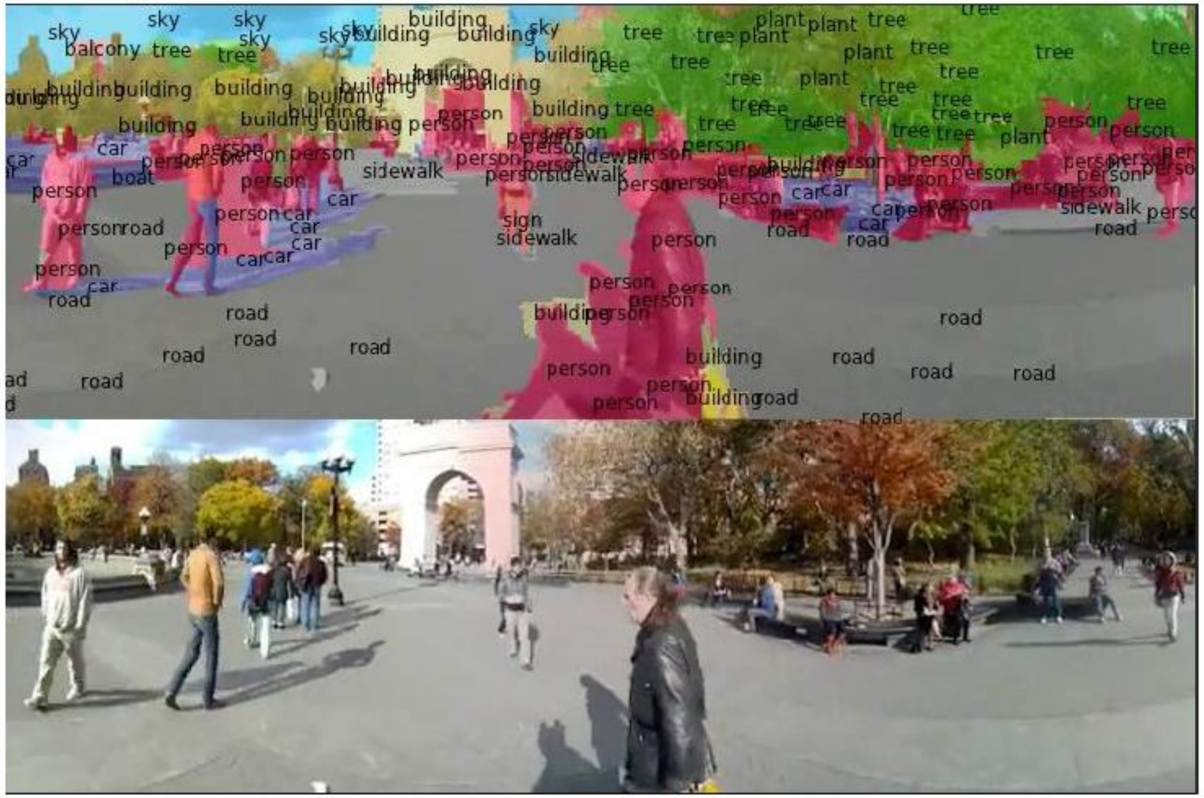
Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girschick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



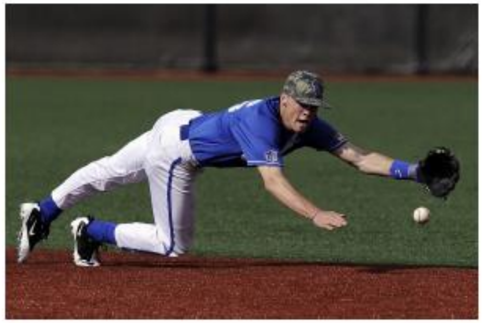
Figures copyright Clement Farabet, 2012.
Reproduced with permission.

[Farabet et al., 2012]

Convnets are everywhere

No errors | Minor errors | Somewhat related

A white teddy bear sitting in the grass



A man in a baseball uniform throwing a ball

A woman is holding a cat in her hand

A man riding a wave on top of a surfboard

A cat sitting on a suitcase on the floor

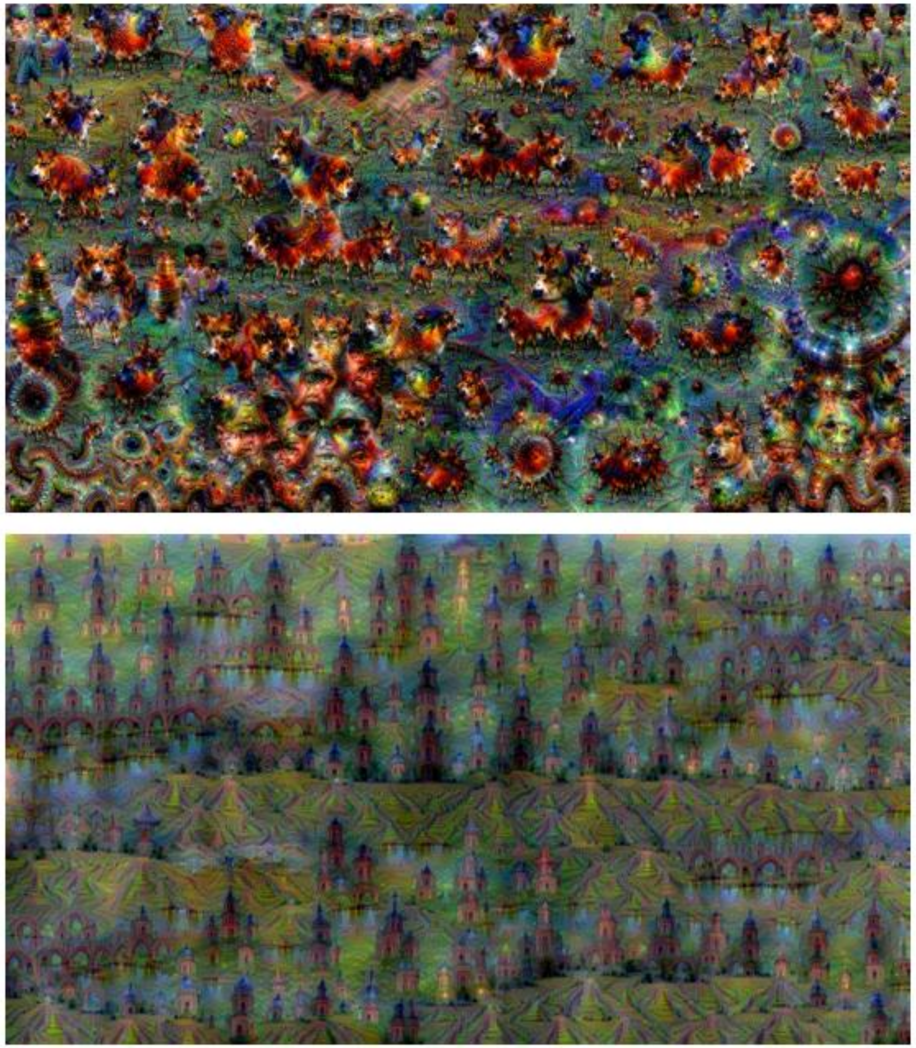
A woman standing on a beach holding a surfboard

Image Captioning

[Vinyals et al., 2015]
[Karpathy and Fei-Fei, 2015]

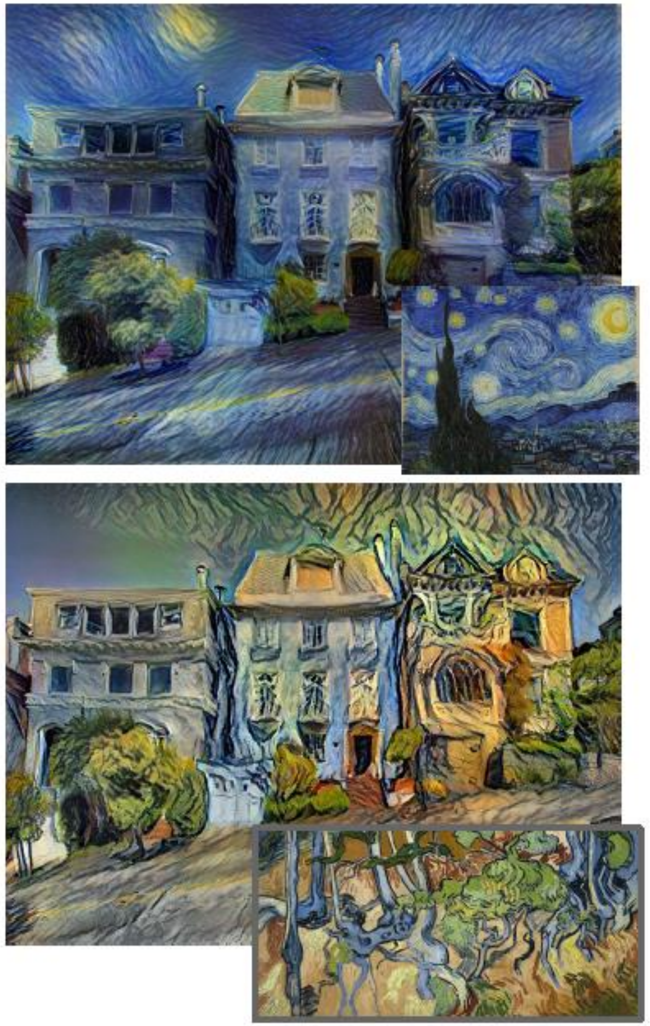
Captions generated by Justin Johnson using Neuraltalk2

Convnets are everywhere



Figures copyright Justin Johnson, 2015. Reproduced with permission. Generated using the Inceptionism approach from a blog post by Google Research.

Original image is CC0 public domain
Starry Night and Tree Roots by Van Gogh are in the public domain
Bokeh image is in the public domain
Stylized images copyright Justin Johnson, 2017;
reproduced with permission



Gatys et al, "Image Style Transfer using Convolutional Neural Networks", CVPR 2016
Gatys et al, "Controlling Perceptual Factors in Neural Style Transfer", CVPR 2017

Layers used to build convnets

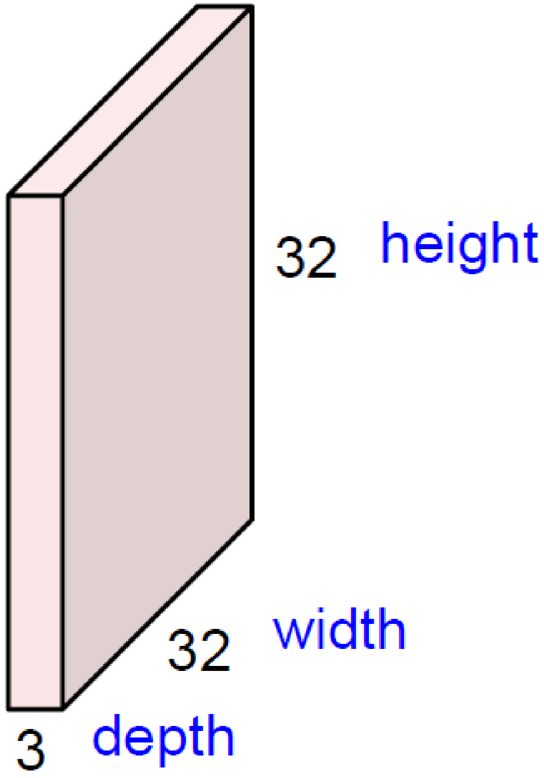
- a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function.

- We use three main types of layers to build ConvNet architectures:
  - Convolutional Layer
  - Pooling Layer
  - Fully-Connected Layer

- We will stack these layers to form a full ConvNet architecture.

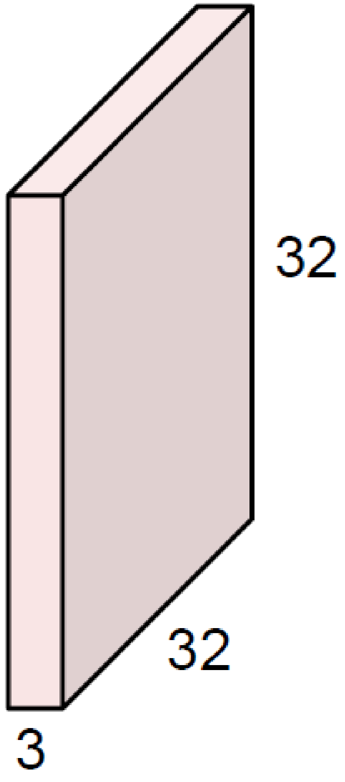
Convolution Layer

32x32x3 image -> preserve spatial structure

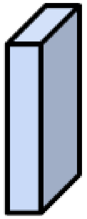


Convolution Layer

32x32x3 image



5x5x3 filter

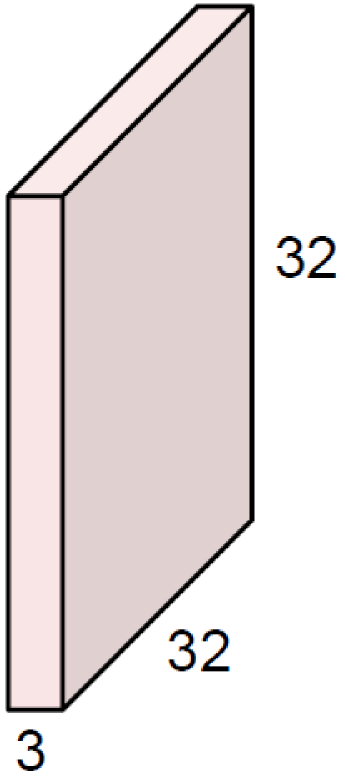


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

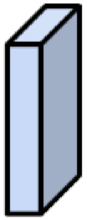
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

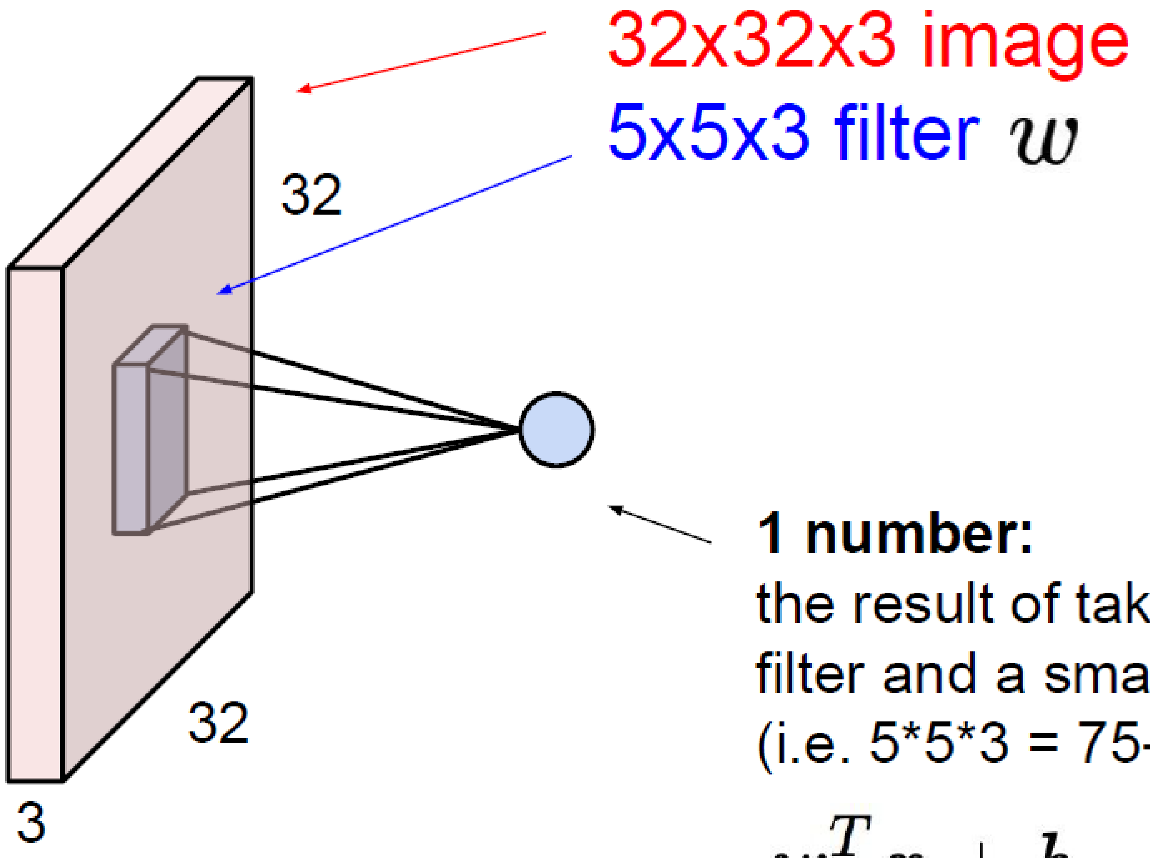


5x5x3 filter



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

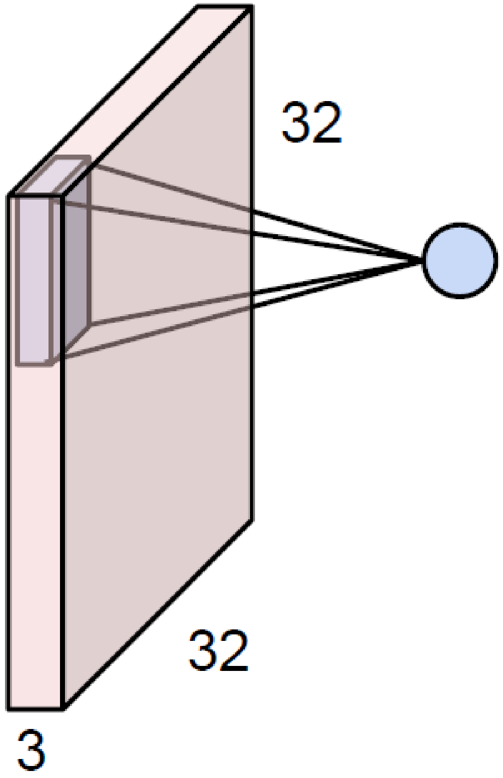
Convolution Layer



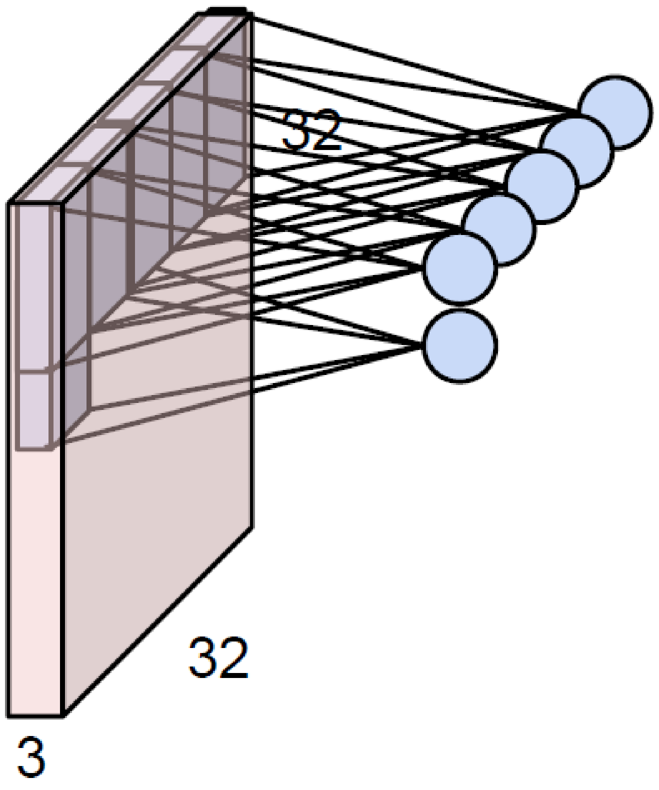
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

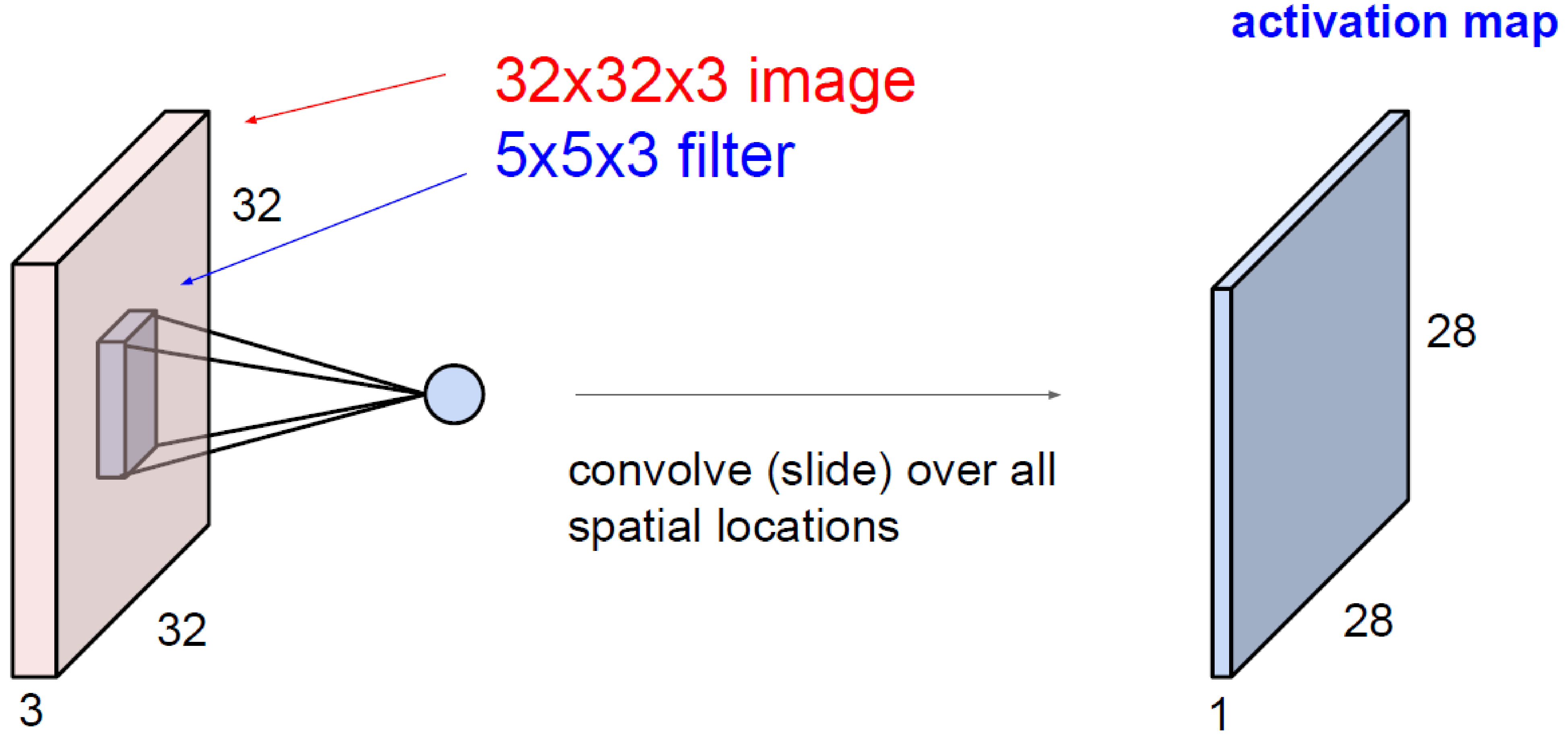
Convolution Layer



Convolution Layer

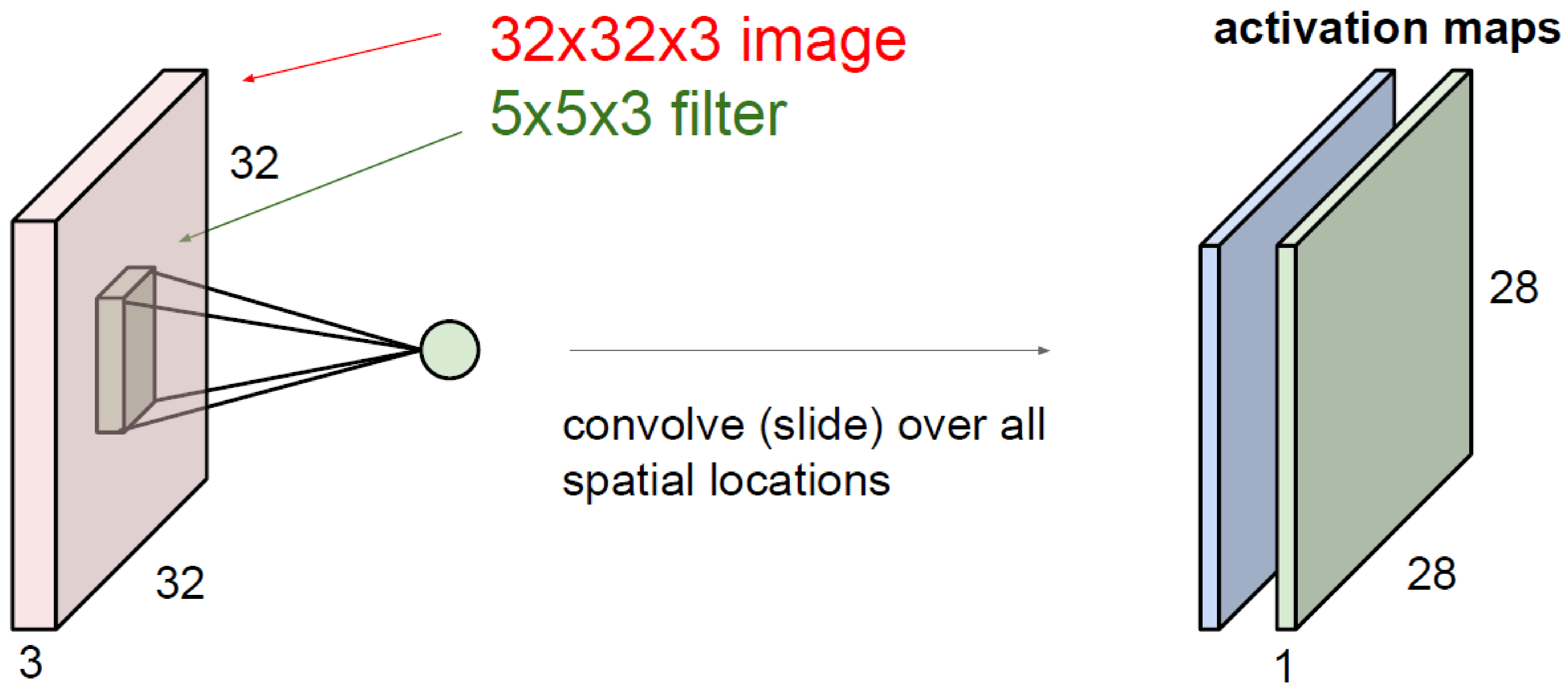


Convolution Layer

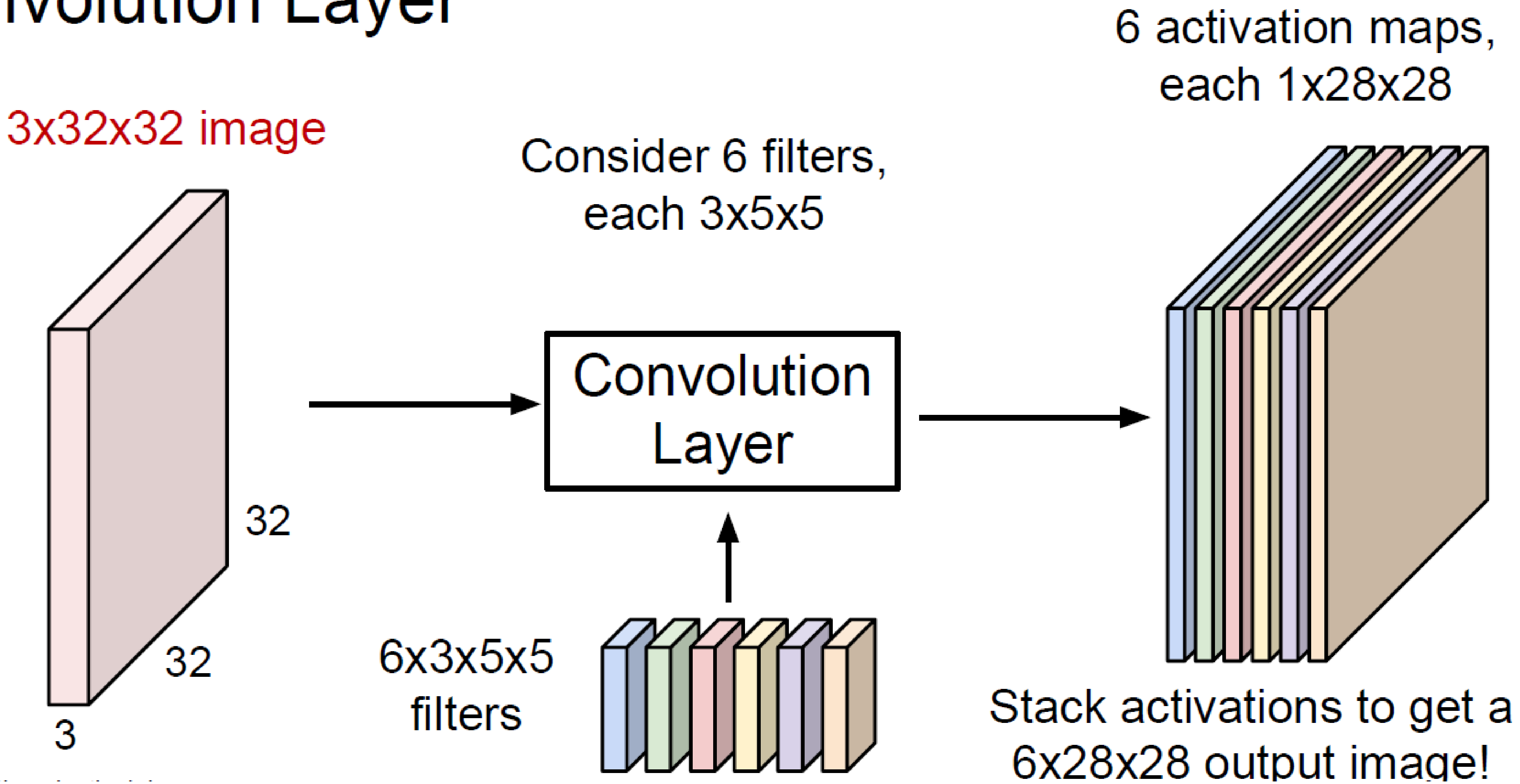


Convolution Layer

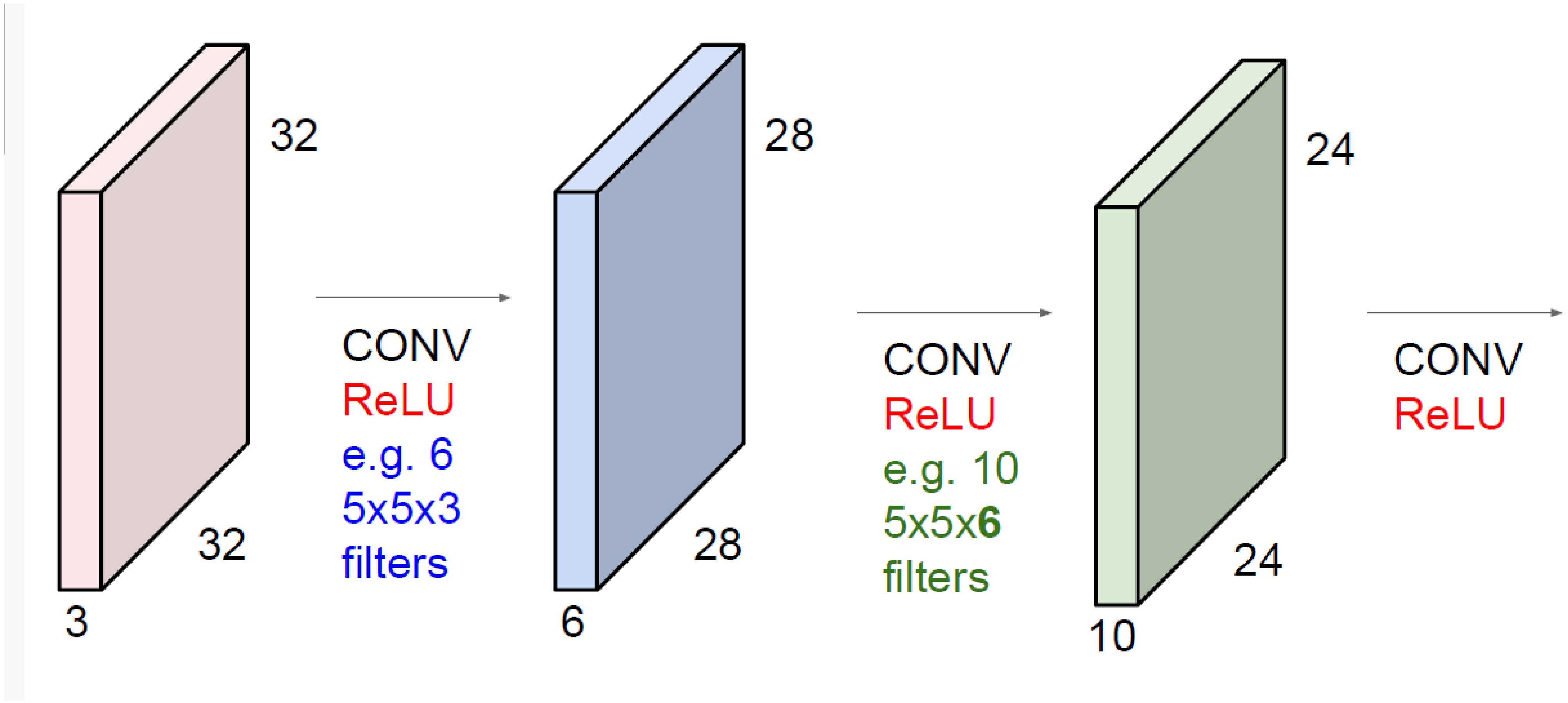
consider a second, green filter



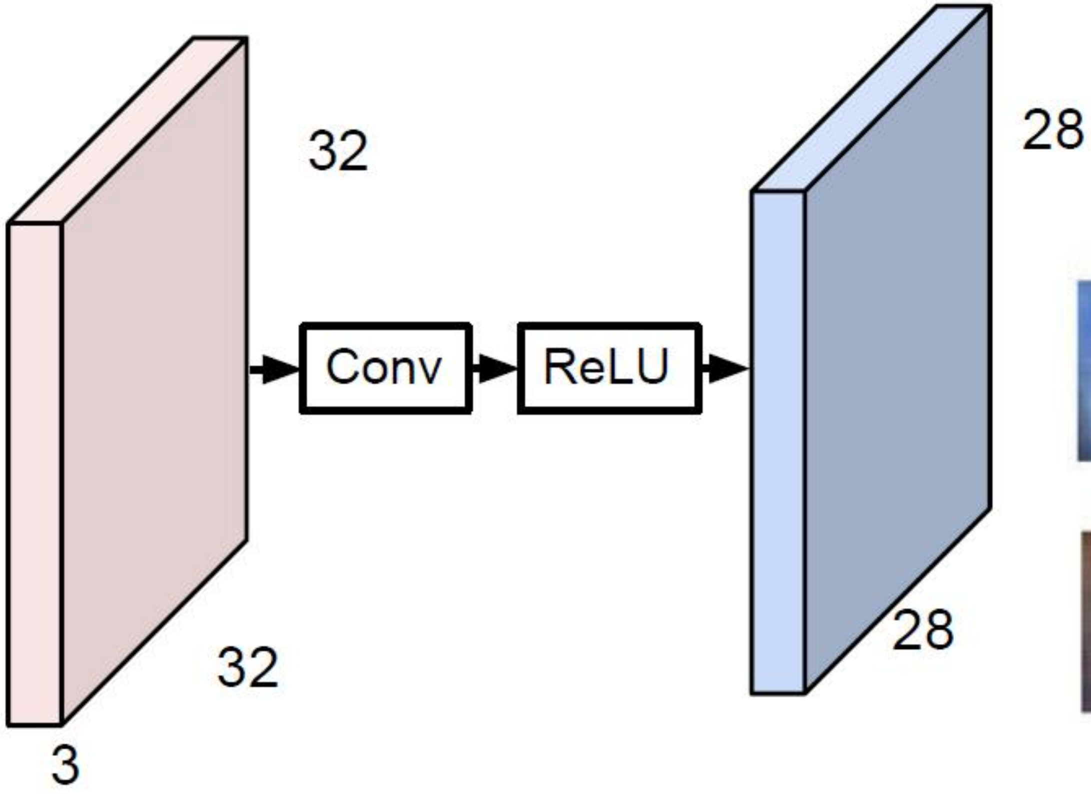
Convolution Layer



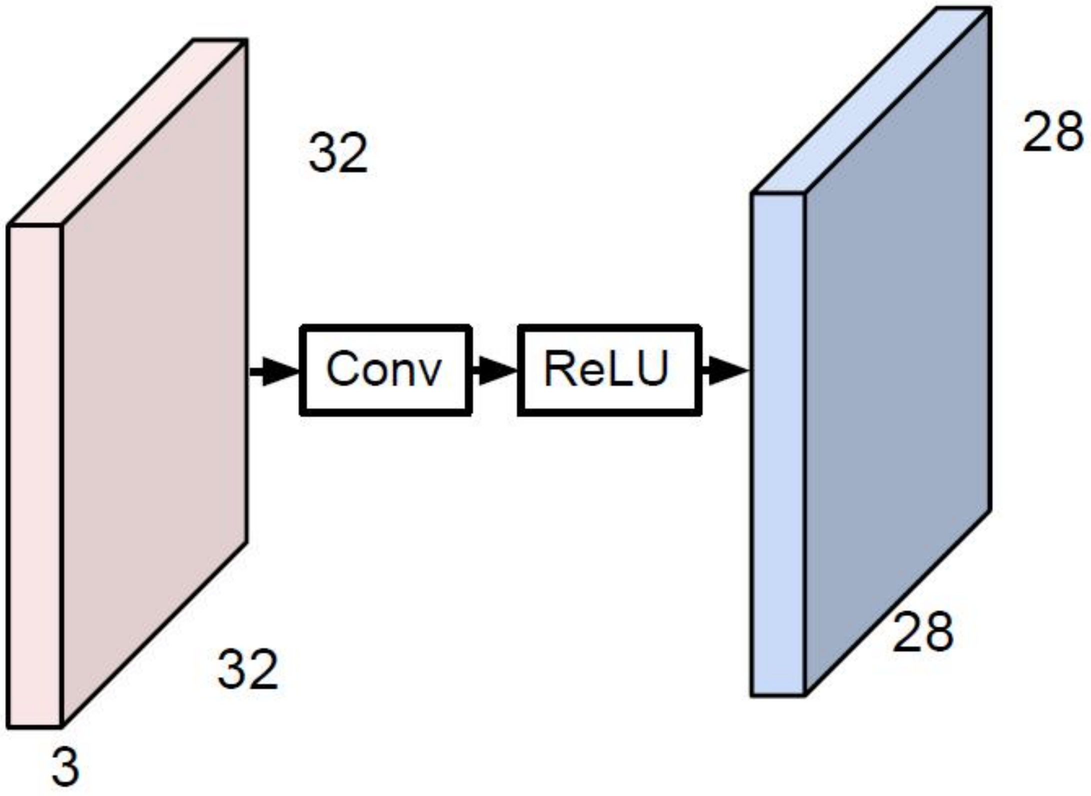
ConvNet is a sequence of Convolution Layers, with activation functions



What do convolutional filters learn?

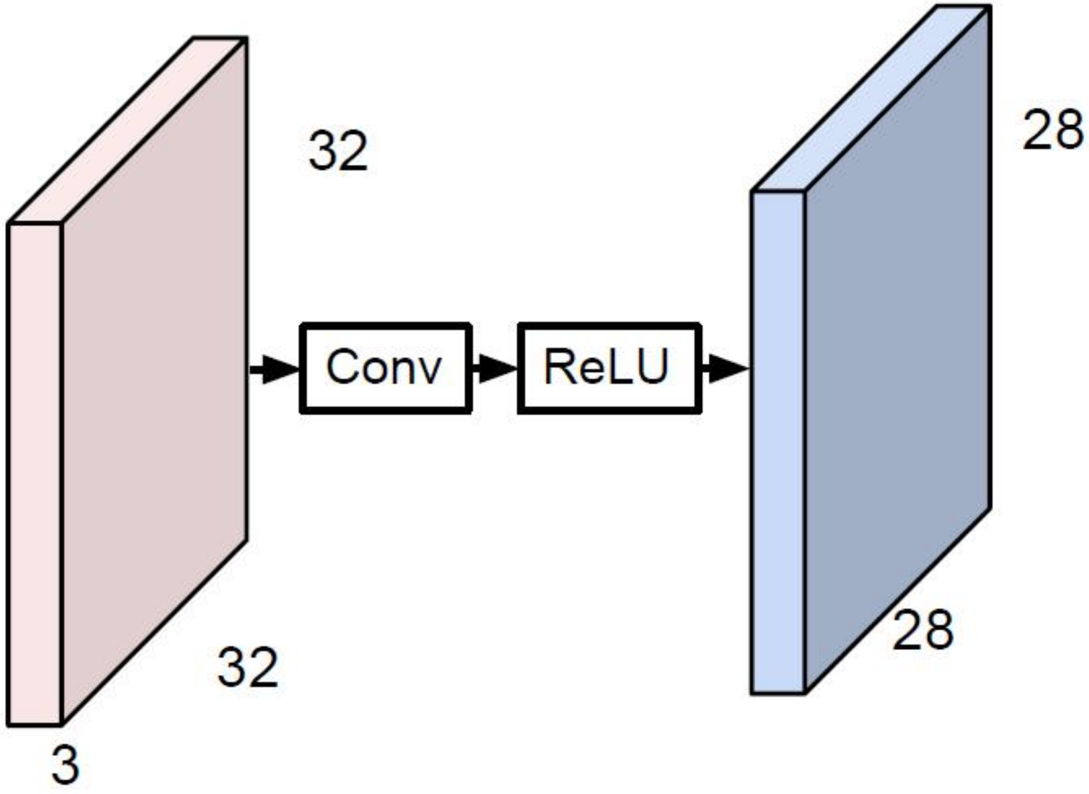


What do convolutional filters learn?

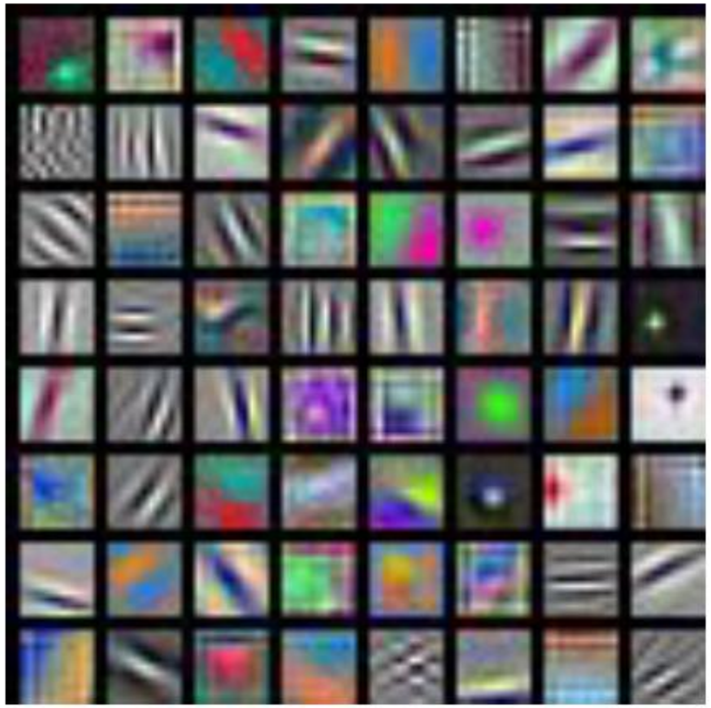


MLP: Bank of whole-image templates

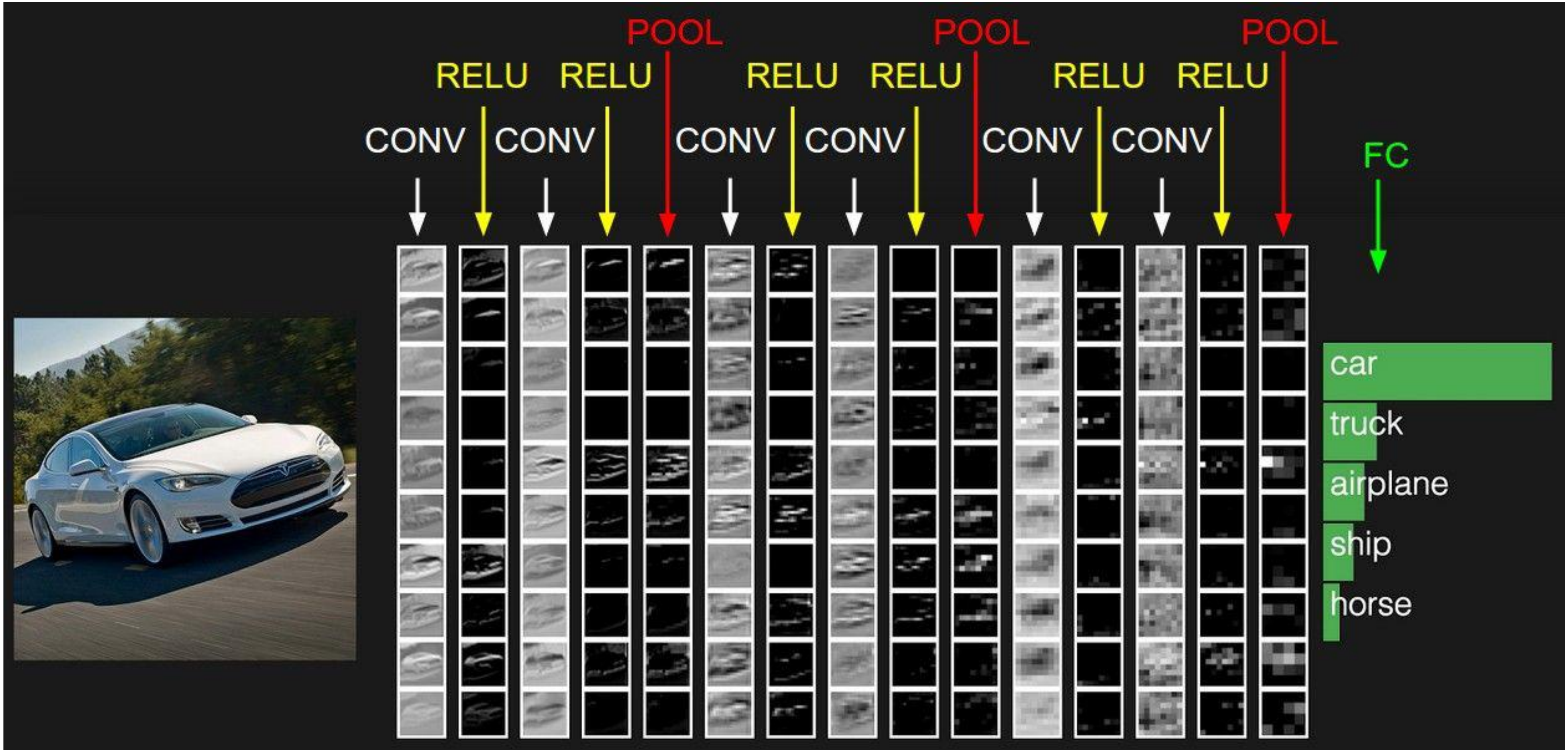
What do convolutional filters learn?



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



AlexNet: 64 filters, each 3x11x11


POOL
RELU
RELU
POOL
RELU
RELU
POOL
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse