

# Mastering Deep Learning Within a Few Hours

Stanley Zheng, CSE, CUHK

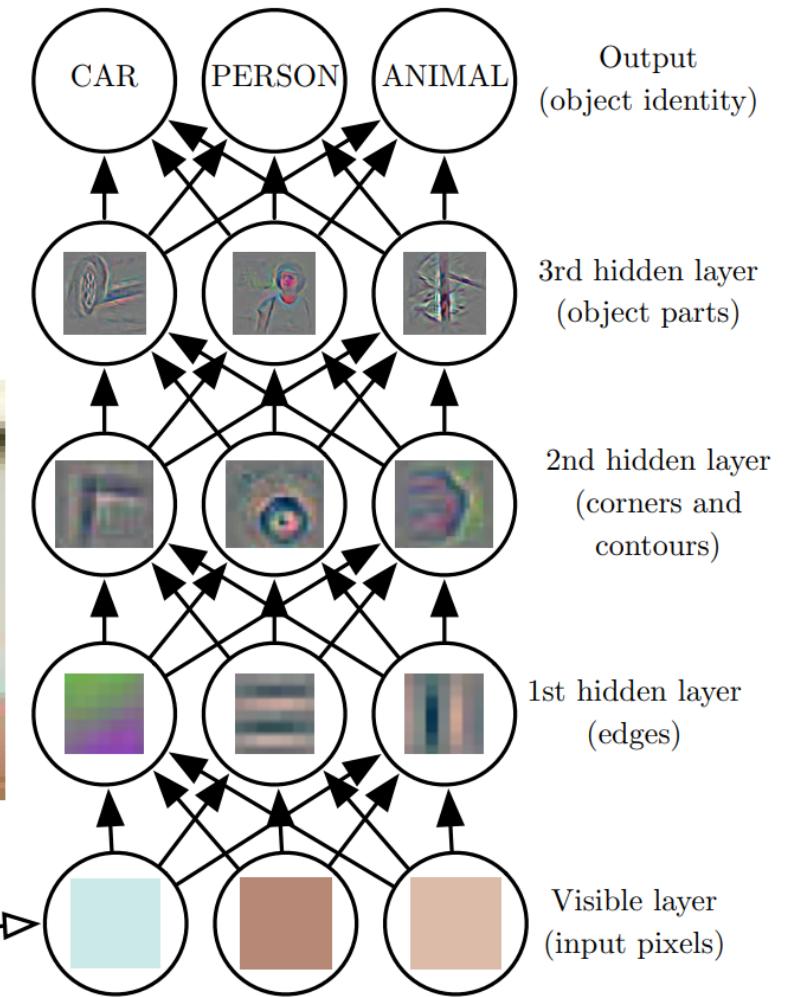
- Introduction
- Problem Formulation
- Implementation
- Experiments
- Advanced Topics

# Introduction

Artificial v.s. Intelligence

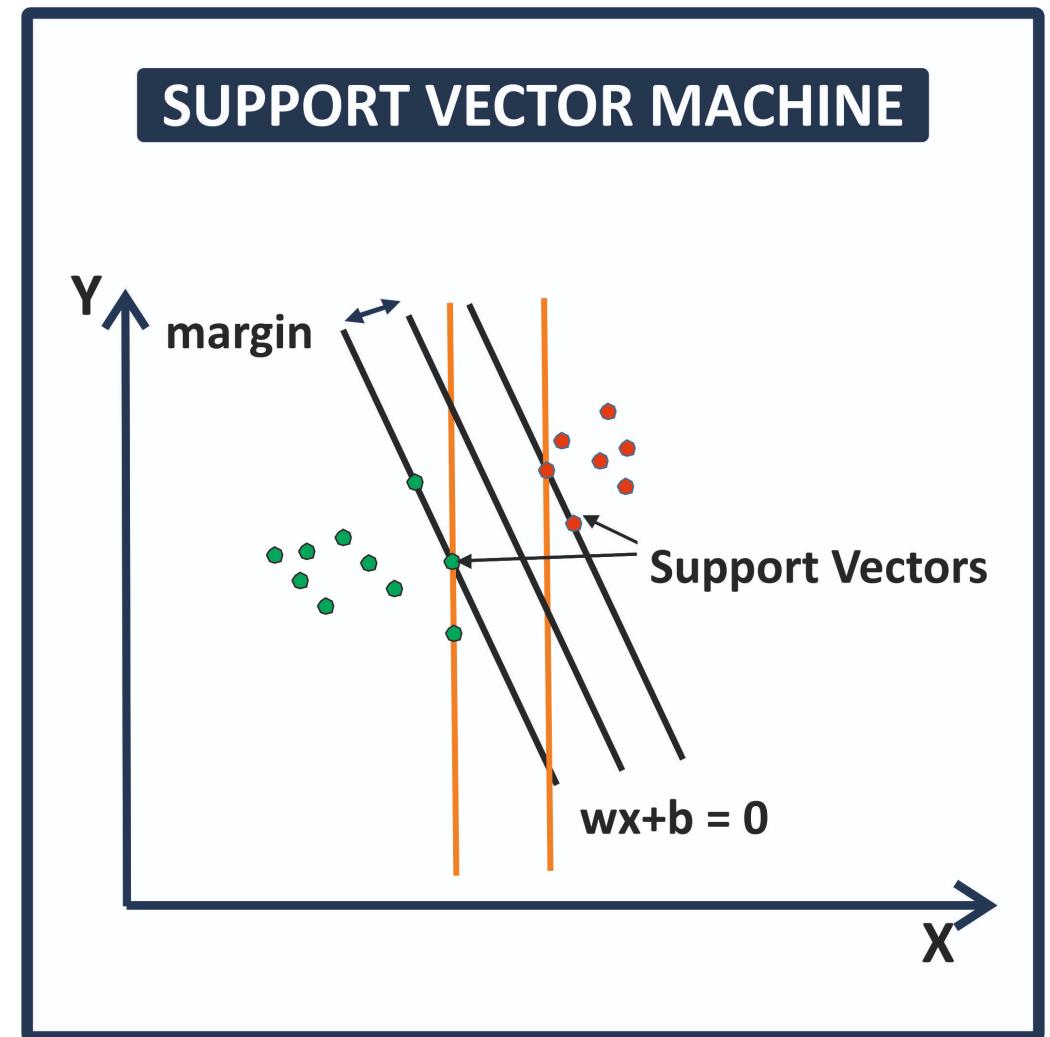
# Intelligence

- Human Perception



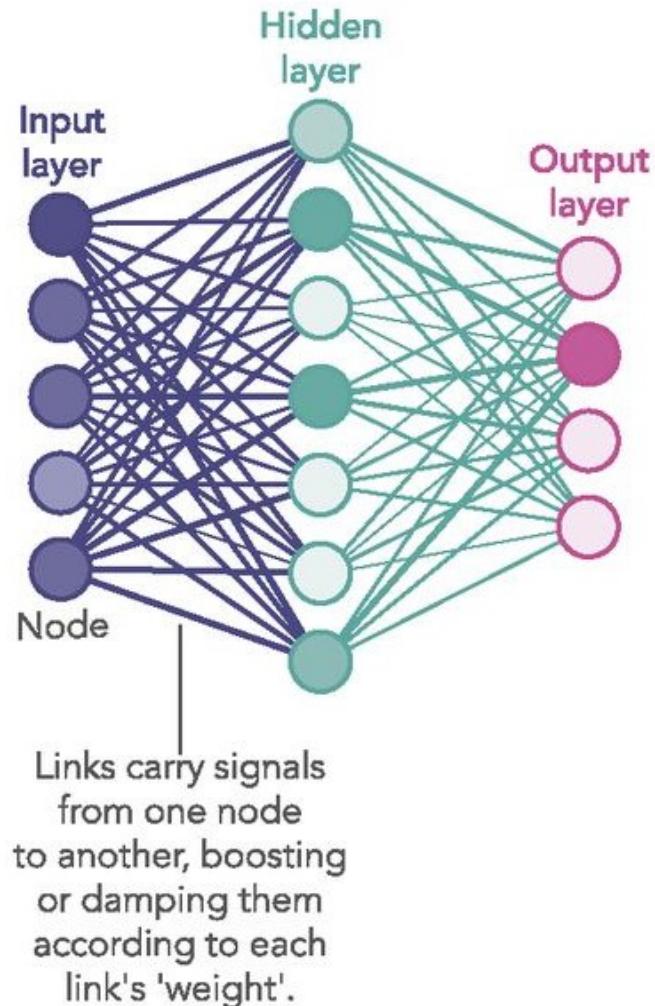
# Artificial

- Machine Learning

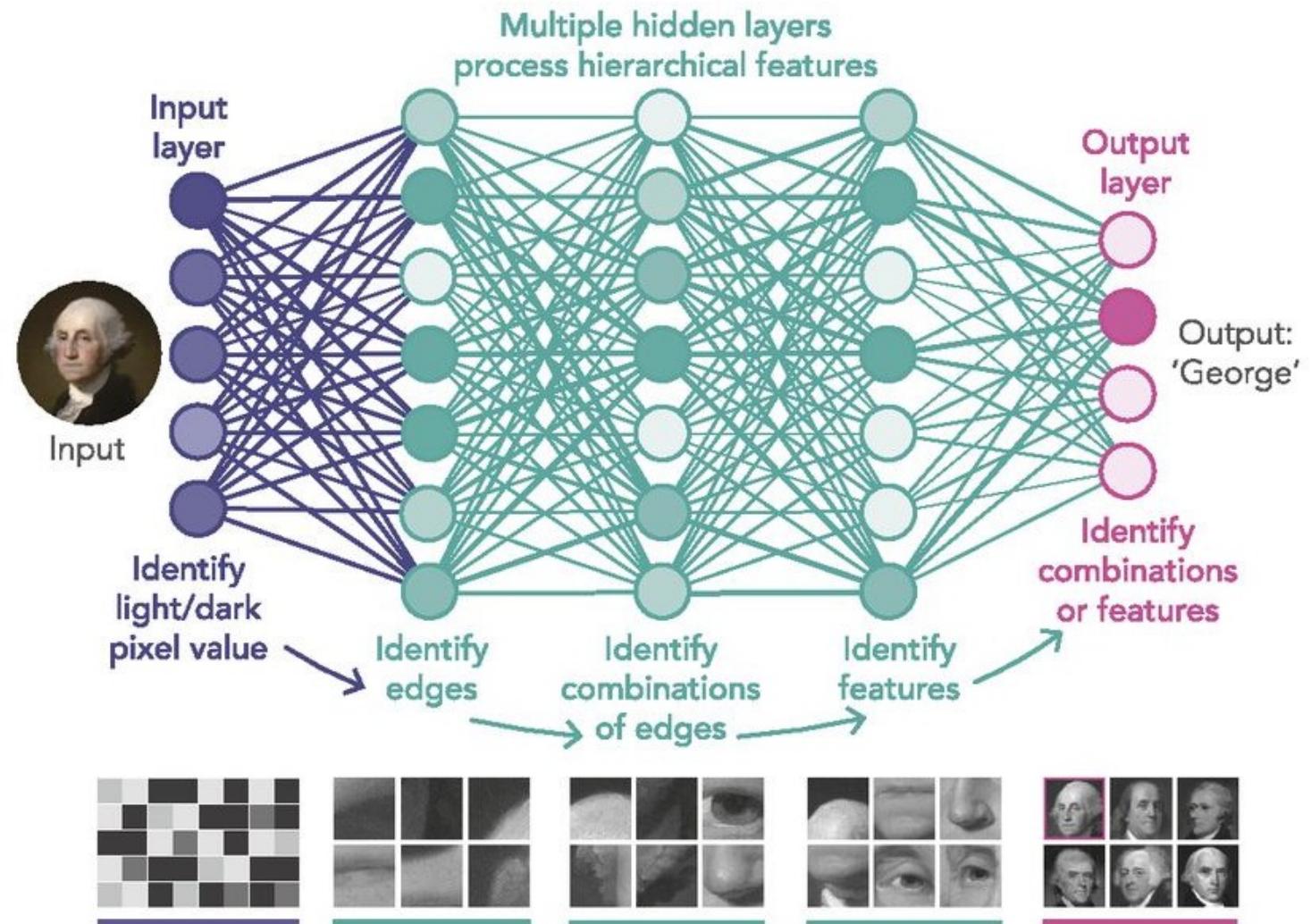


# Deep Learning

1980S-ERA NEURAL NETWORK

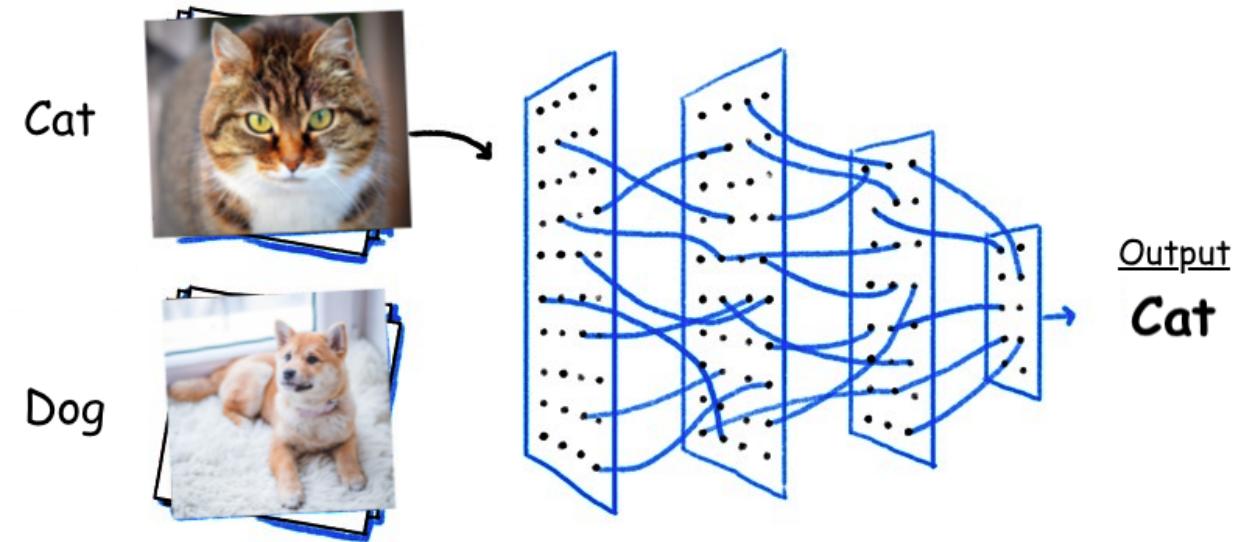


DEEP LEARNING NEURAL NETWORK



# What Can Deep Learning Do?

- Classification
- What is this?



## What Can Deep Learning Do?

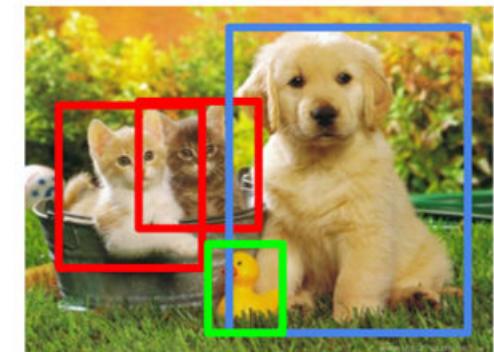
- Objective Detection
- Where is it?

Classification



CAT

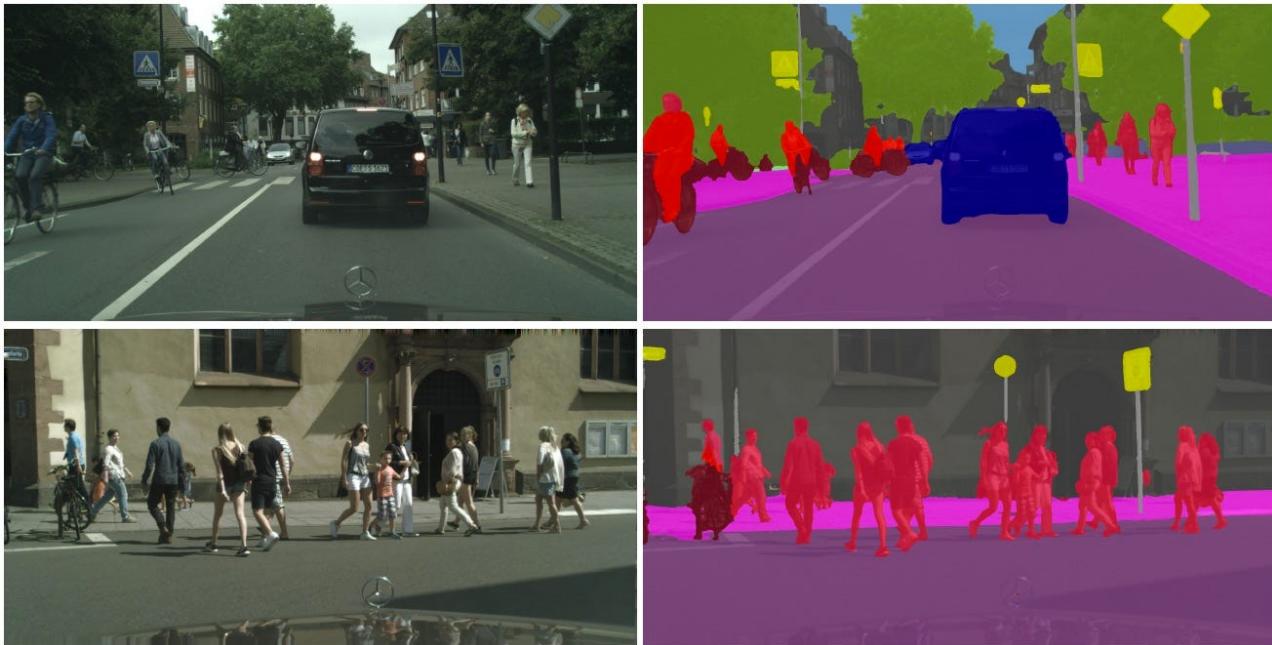
Object Detection



CAT, DOG, DUCK

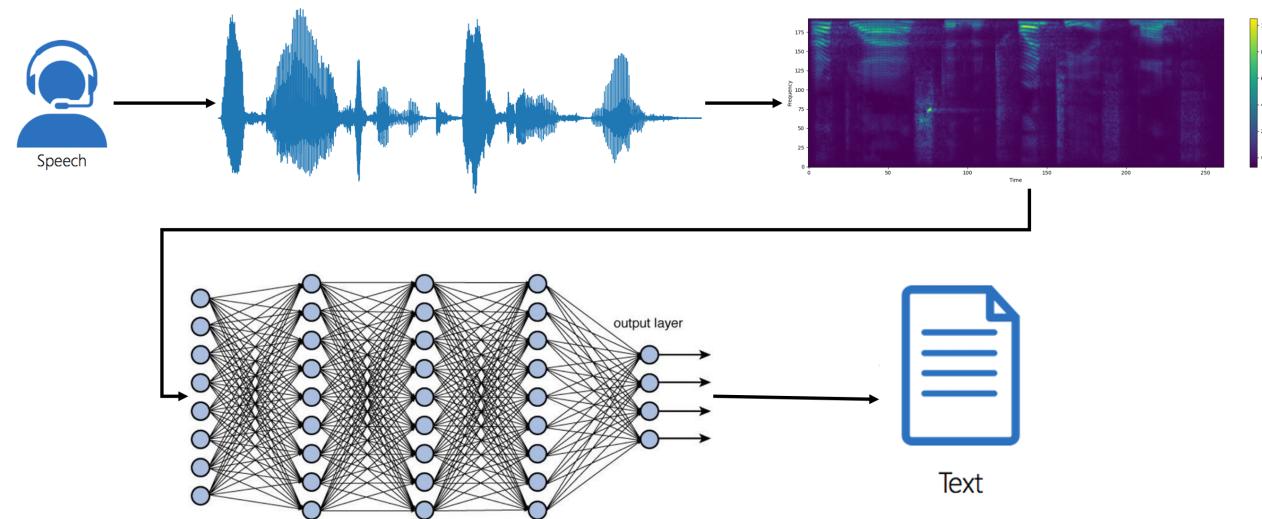
# What Can Deep Learning Do?

- Semantic Segmentation
- Classify every pixel



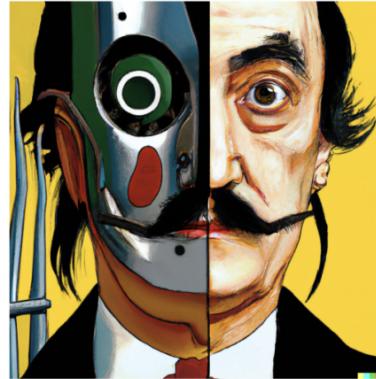
# What Can Deep Learning Do?

- Speech Recognition
- Voice to text



# What Can Deep Learning Do?

- AIGC: AI-Generated Content
- Text-to-image/image-to-image



vibrant portrait painting of Salvador Dalí with a robotic half face



a shiba inu wearing a beret and black turtleneck



a close up of a handpalm with leaves growing from it



an espresso machine that makes coffee from human souls, artstation



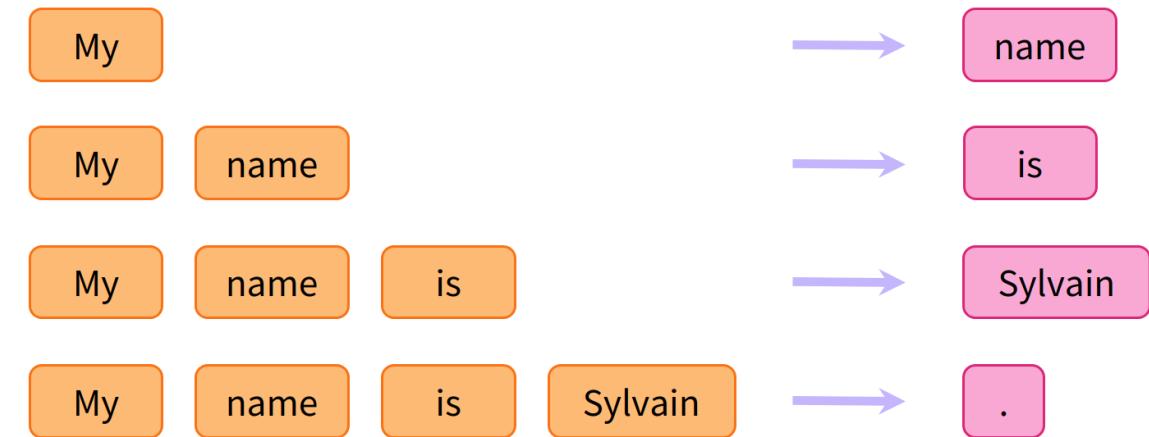
panda mad scientist mixing sparkling chemicals, artstation



a corgi's head depicted as an explosion of a nebula

## What Can Deep Learning Do?

- GPT: Generative Pre-trained Transformer
- Autoregressive model



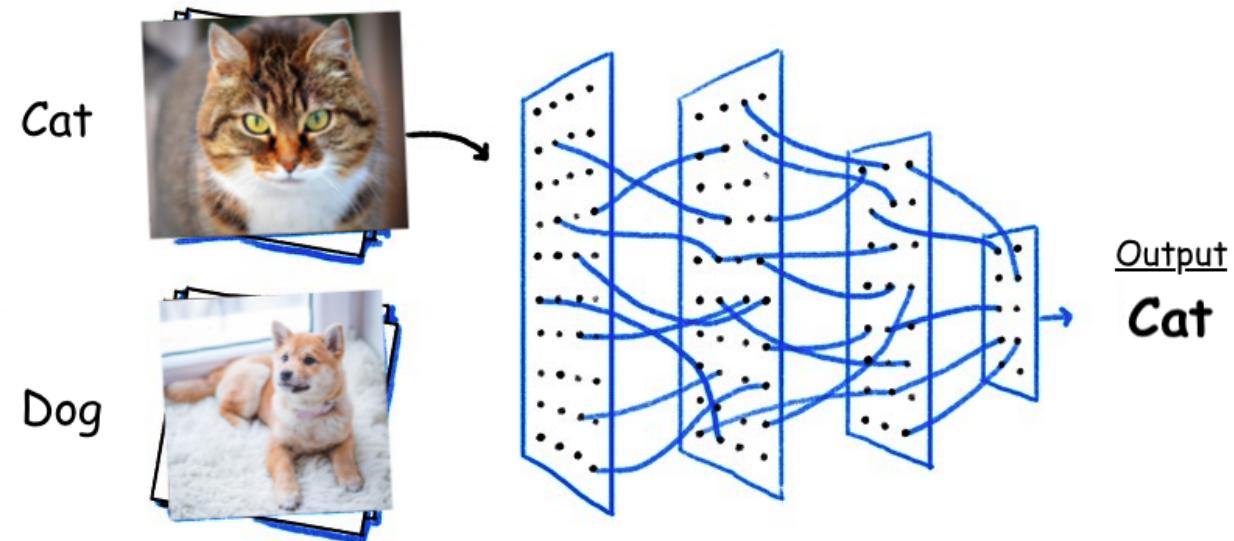
# Problem Formulation

$$\bar{\mathbf{y}} = f(\mathbf{W}, \mathbf{x}) = f_M(W_M, f_{M-1}(W_{M-1}, \dots, f_1(W_1, \mathbf{x})))$$

$$\mathbf{W} = \arg \min_{\mathbf{W}} \sum_{i=1}^N L(\bar{\mathbf{y}}, \mathbf{y})$$

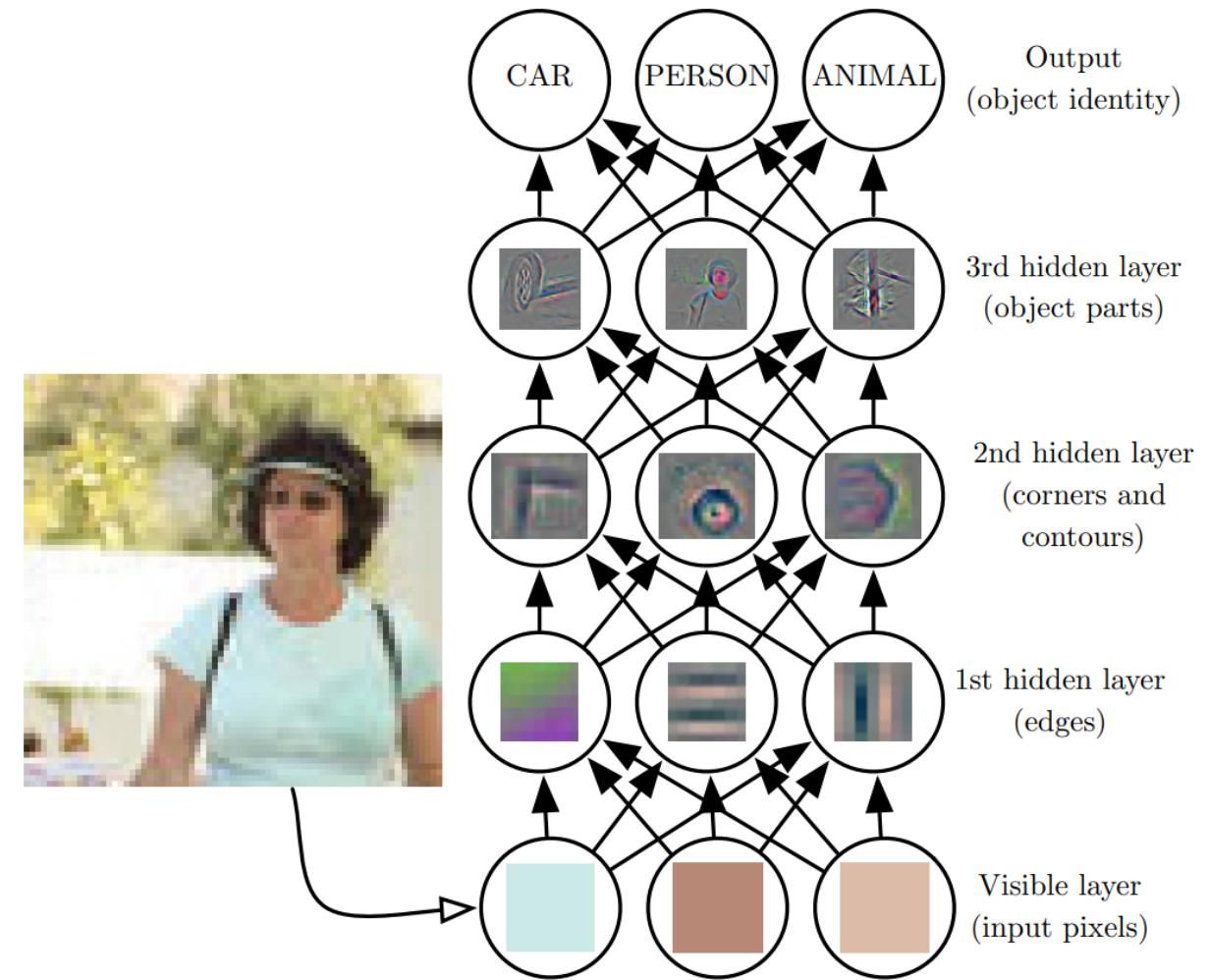
## Revisiting Classification

- Input:  $\mathbf{x}$  is the image (RGB)
- Output: cat:  $\mathbf{y} = [1, 0]$ ; dog:  $\mathbf{y} = [0, 1]$
- $\bar{\mathbf{y}} = f(\mathbf{W}, \mathbf{x})$



## Imitating Human Perception

- Input:  $\mathbf{x}$  is the image (RGB)
- 1st hidden layer:  
$$\mathbf{x}_1 = \mathbf{f}_1(\mathbf{x}) = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$
- 2nd hidden layer:  
$$\mathbf{x}_2 = \mathbf{f}_2(\mathbf{x}) = \sigma(\mathbf{W}_2 \mathbf{x}_1 + \mathbf{b}_2)$$
- ....



# Matrix Multiplication

- $W_1x$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} (1)(7)+(2)(8)+(3)(9) \\ (4)(7)+(5)(8)+(6)(9) \end{bmatrix} = \begin{bmatrix} 7+16+27 \\ 28+40+54 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

**2 x 3                    3 x 1**  
columns on 1st = rows on 2nd  
**2 x 1                    2 x 1                    2 x 1**

The number of rows in the 1st matrix and the number of columns in the 2nd matrix, make the dimensions of the final matrix

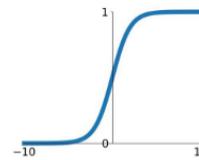
## Activation Function

- $f_1(\mathbf{x}) = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$

## Activation Functions

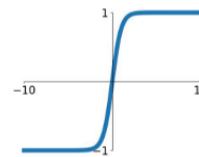
### Sigmoid

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



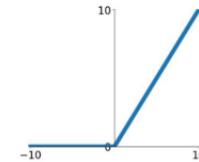
### tanh

$$\tanh(x)$$



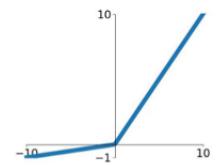
### ReLU

$$\max(0, x)$$



### Leaky ReLU

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

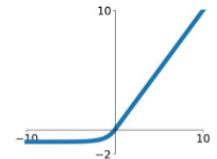


### Maxout

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

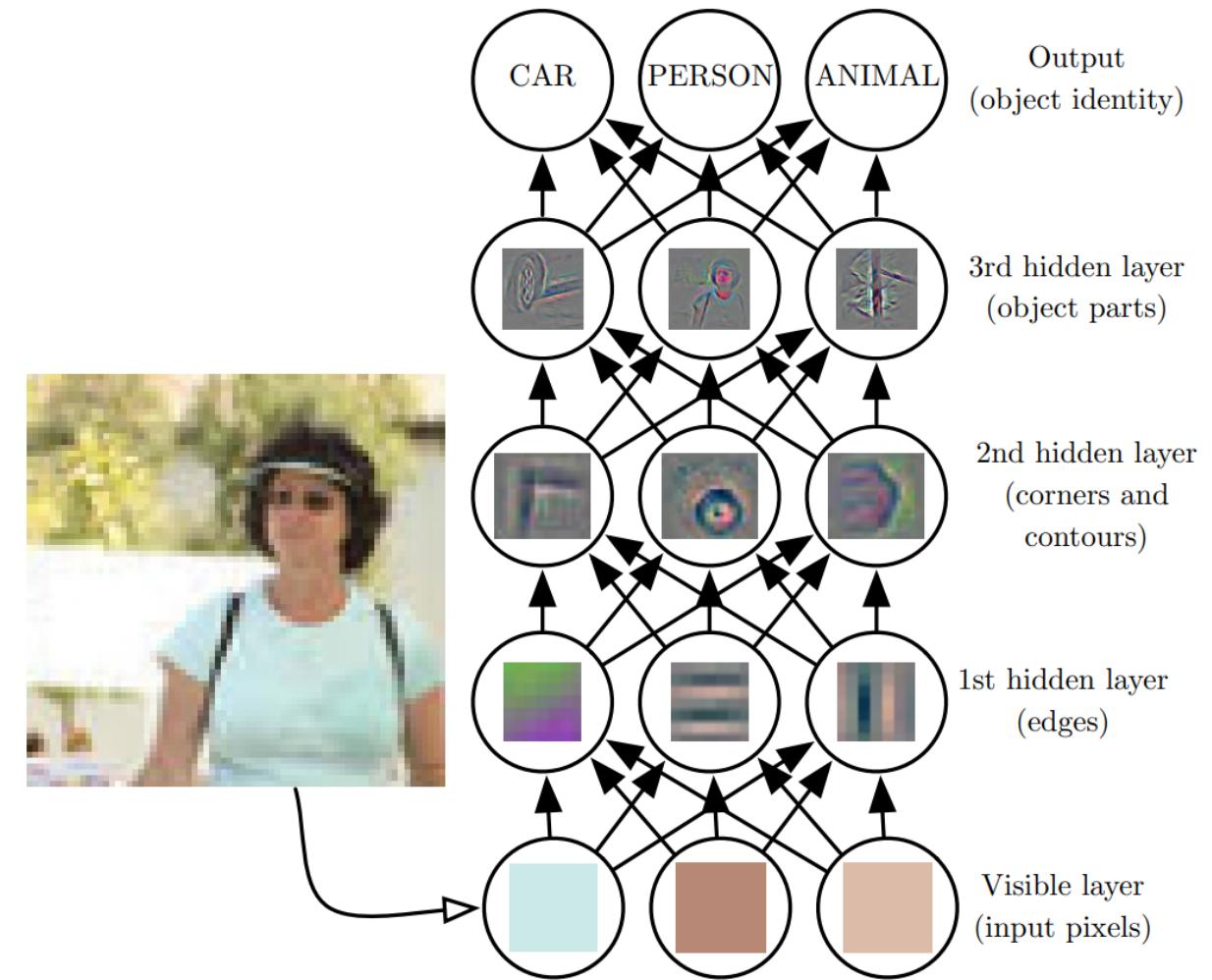
### ELU

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



## Imitating Human Perception

- Input:  $\mathbf{x}$  is the image (RGB)
- 1st hidden layer:  
$$\mathbf{x}_1 = \mathbf{f}_1(\mathbf{x}) = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$
- 2nd hidden layer:  
$$\mathbf{x}_2 = \mathbf{f}_2(\mathbf{x}) = \sigma(\mathbf{W}_2 \mathbf{x}_1 + \mathbf{b}_2)$$
- ....



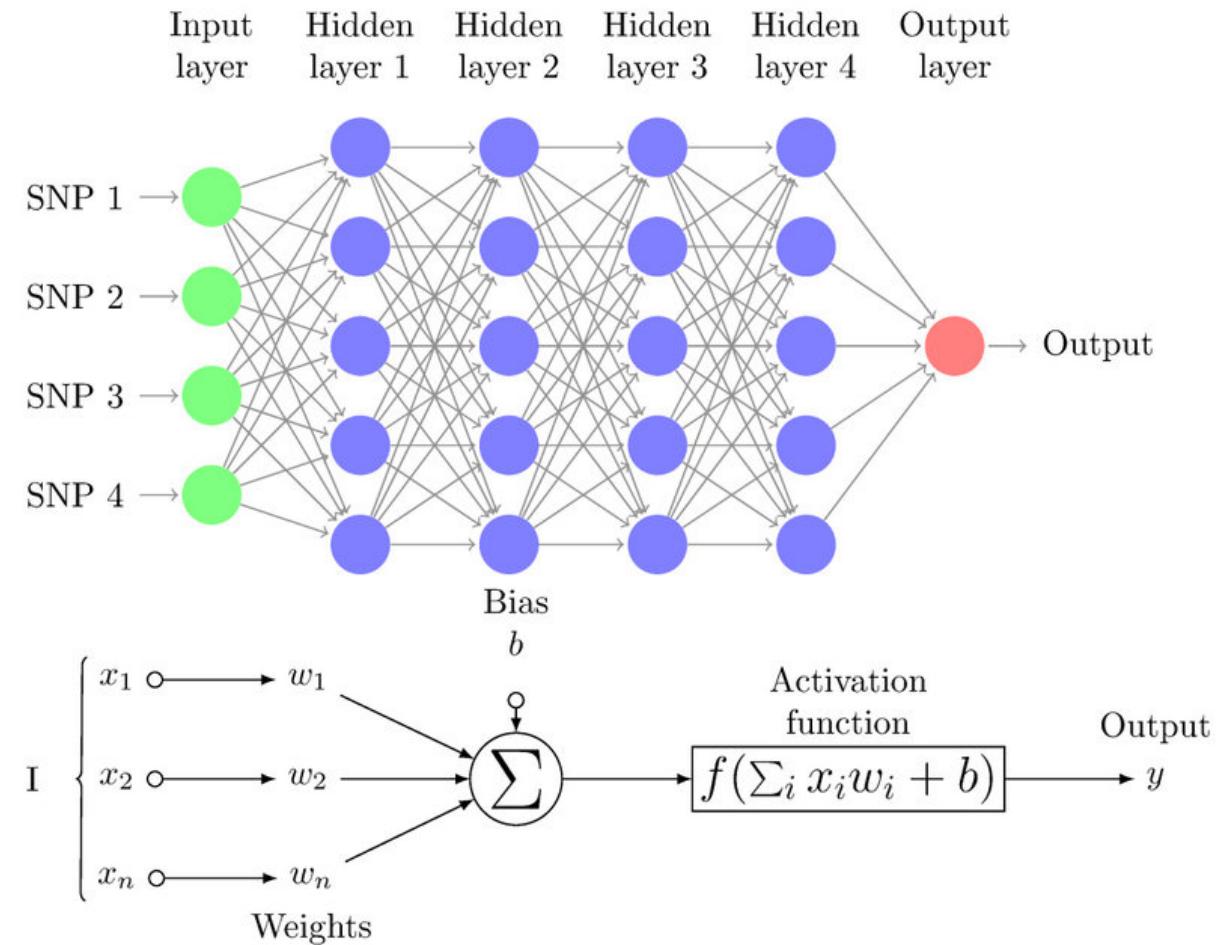
## Output Function

- $f_M(\mathbf{x}) = \text{softmax}(\mathbf{W}_M \mathbf{x}_{M-1} + \mathbf{b}_l)$
- $\text{softmax}(\mathbf{x}_M)_i = \frac{x_{M,i}}{\sum_j x_{M,j}}$
- Regarded as probabilities

$$\begin{bmatrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \\ \frac{e^{x_4}}{e^{x_1}+e^{x_2}+e^{x_3}+e^{x_4}} \end{bmatrix} = \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ e^{x_3} \\ e^{x_4} \end{bmatrix} / (e^{x_1} + e^{x_2} + e^{x_3} + e^{x_4}) \implies e^x / \text{sum}(e^x)$$

## Multi-layer Perceptrons (MLP)

- $f(\mathbf{x}) = f_M \circ f_{M-1} \circ \dots \circ f_1(\mathbf{x})$
- $\mathbf{W}_1, \mathbf{b}_1, \dots, \mathbf{W}_M, \mathbf{b}_M$  are trainable



## Loss Function

- $L(\bar{y}, y)$ , indicates the quality
- Smaller is better ↓
- Cross entropy is for classification
- e.g.  $p = [1, 0]; q = [0.8, 0.2]$
- $CE(p, q) = -\log(0.8)$

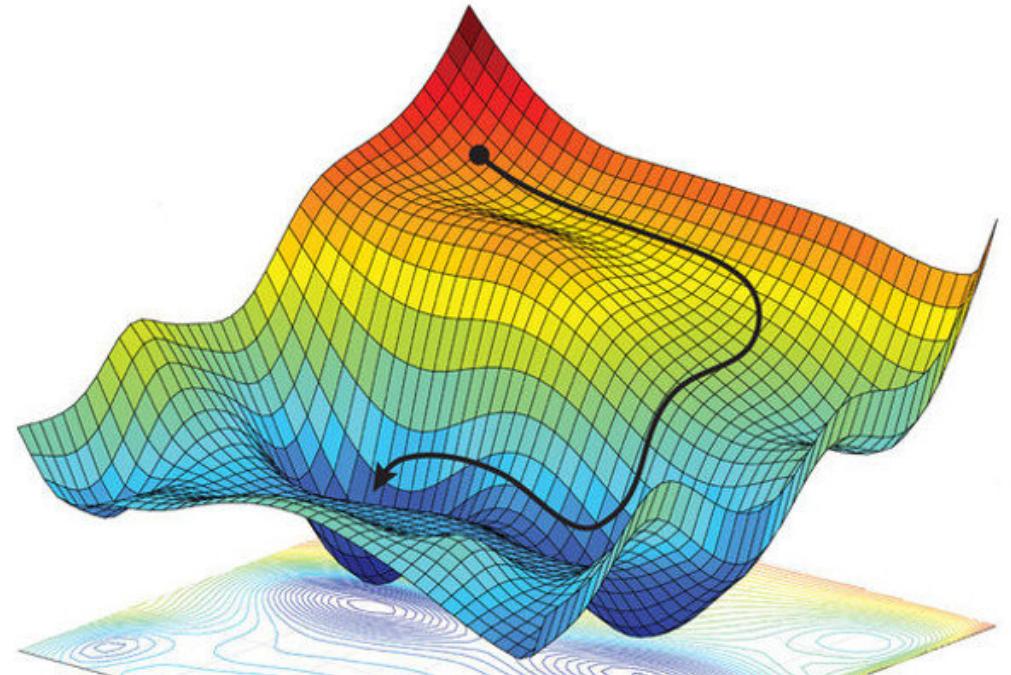
$$H(p, q) = - \sum_{x \in \text{classes}} p(x) \log q(x)$$

True probability distribution  
(one-hot)

Your model's predicted  
probability distribution

## Gradient Descent

- Iterative algorithm
- $\mathbf{W}^{(t)} = \mathbf{W}^{(t-1)} - \gamma \frac{\partial L}{\partial \mathbf{W}^{(t)}}$
- $\gamma$  is the learning rate
- Chain rule  $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$

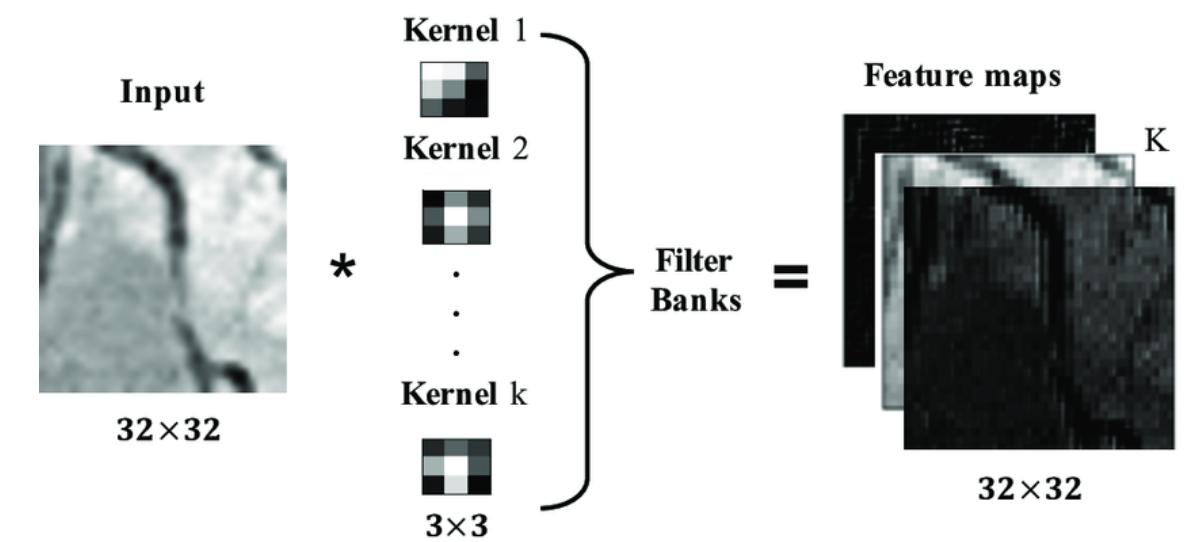


# Implementation

- Convolutional Neural Networks
- Training and Testing
- PyTorch

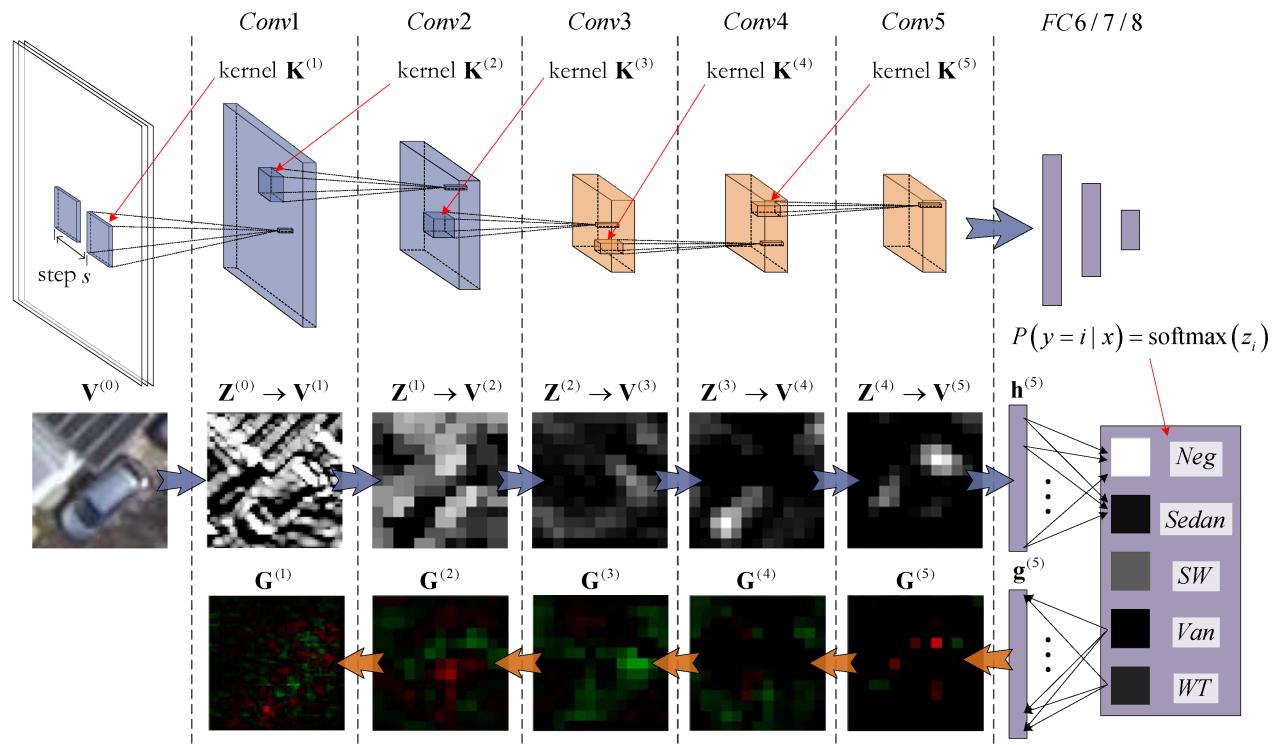
# Convolution

- From signal processing
- For feature extraction



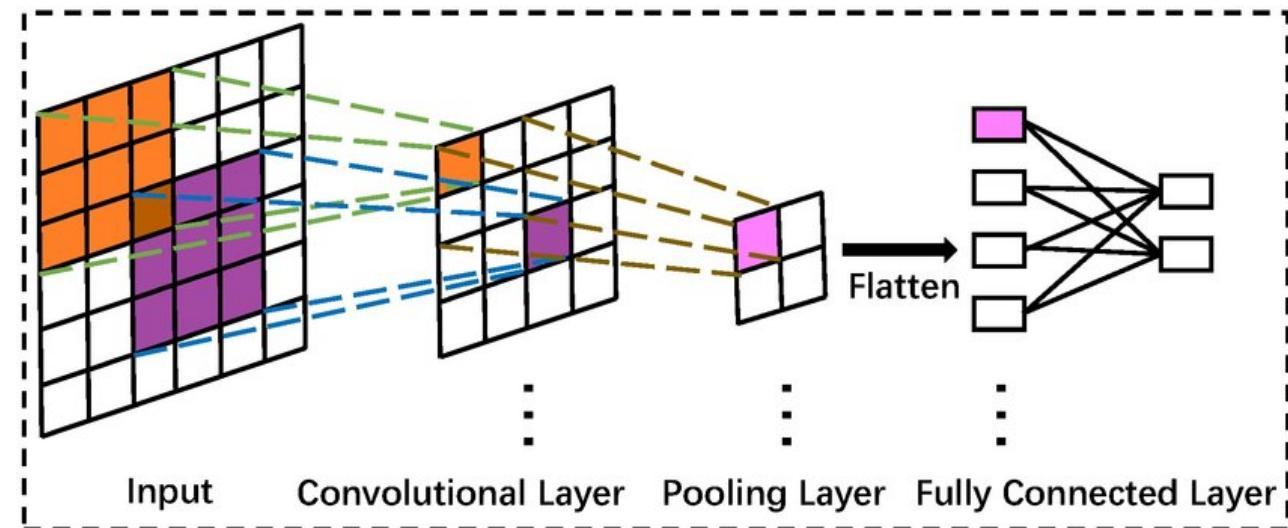
# Convolutional Layers

- Convolutional layer
- Preserve 2D structure
- Local perception
- Less weights



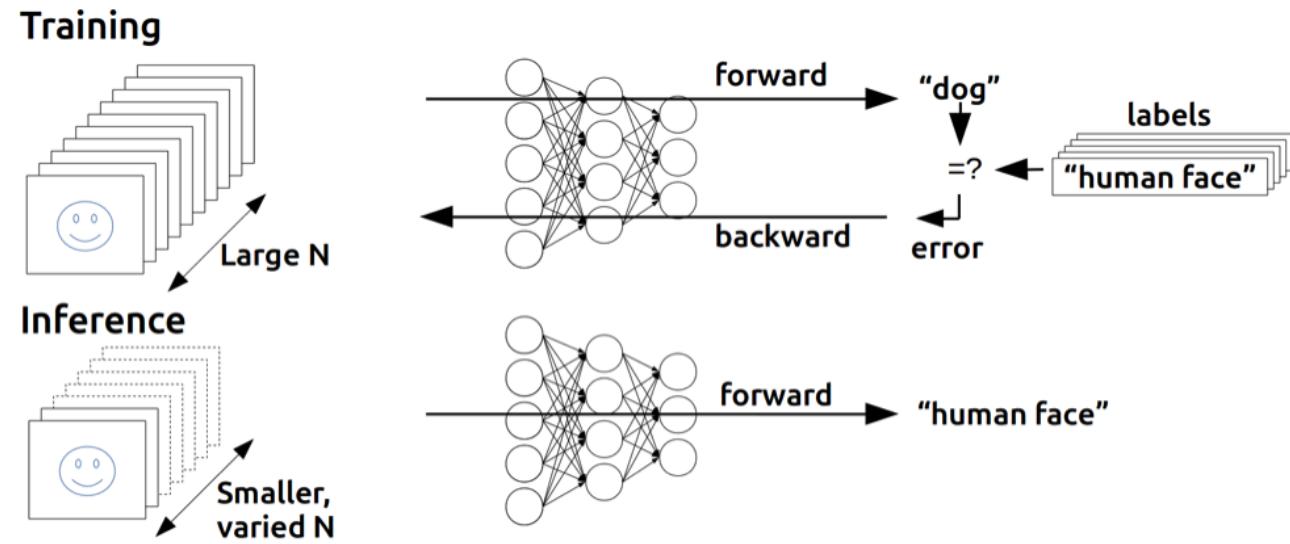
# Convolutional Neural Networks

- Convolutional layer
- Max/avg. pooling layer
- Fully connected layer (MLP)



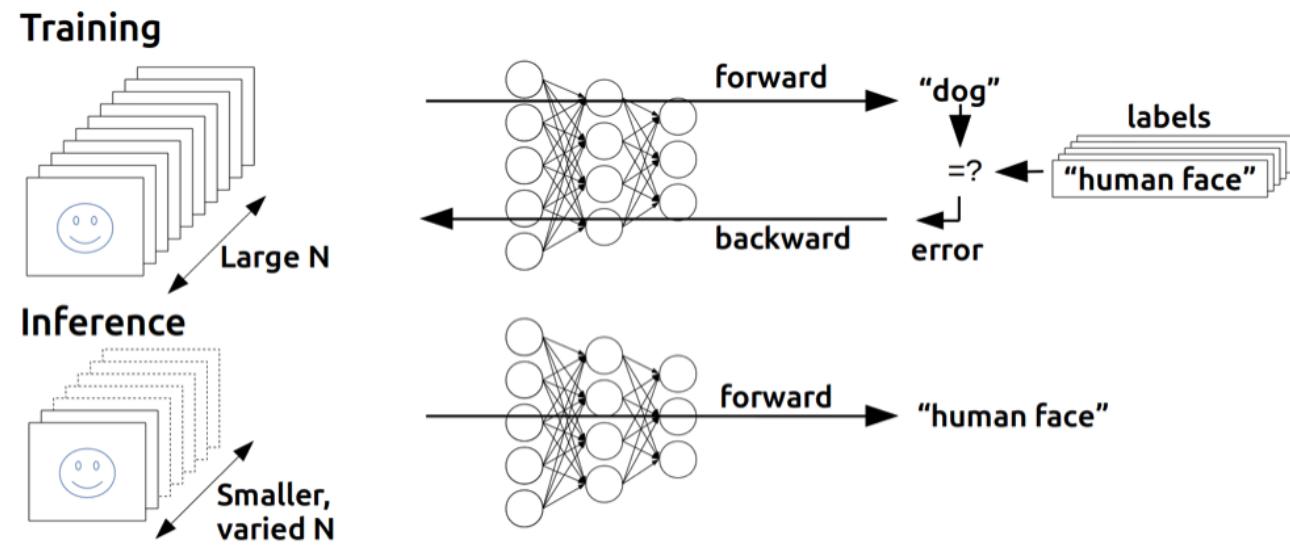
# Training Neural Networks

- Sample a batch of data
- Forward: compute the loss
- backward: update the weights
- $\mathbf{W}^{(t)} = \mathbf{W}^{(t-1)} - \gamma \frac{\partial L}{\partial \mathbf{W}^{(t)}}$



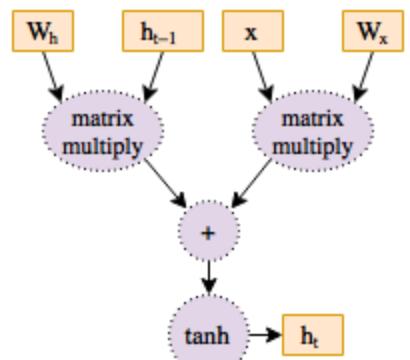
## Testing Neural Networks

- Sample a batch of data
- Compute the loss and accuracy

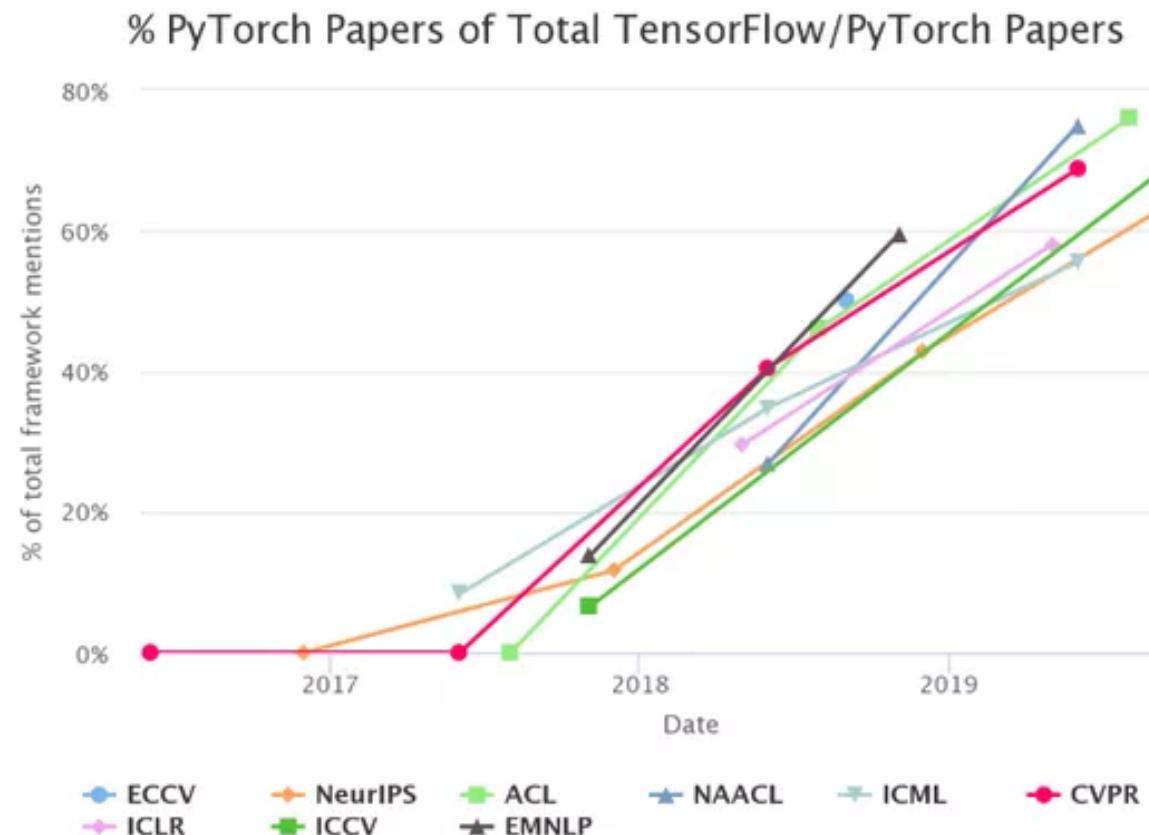


# PyTorch

- PyTorch is famous
- Computation graph



$$h_t = \tanh(W_h h_{t-1}^\top + W_x x^\top)$$



## Import Libraries

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
```

- "torch.nn" for neural network layers (Conv2D, Linear, ... ...)
- "torch.nn.functional" for basic functions (sigmoid, max\_pool2d, ... ...)
- "torch.optim" for optimizers (gradient descent)
- "torchvision" for data preparation

## Load Data

```
batch_size_train = 100 # Size of a batch of data for training
batch_size_test = 100 # Size of a batch of data for testing
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('data/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize((0.1307,), (0.3081,)))
                               ])),
    batch_size=batch_size_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('data/', train=False, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize((0.1307,), (0.3081,)))
                               ])),
    batch_size=batch_size_test, shuffle=True)
```

# MNIST Handwritten Digit Dataset

label = 5



label = 0



label = 4



label = 1



label = 9



label = 2



label = 1



label = 3



label = 1



label = 4



label = 3



label = 5



label = 3



label = 6



label = 1



label = 7



label = 2



label = 8



label = 6



label = 9



## Define the Model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

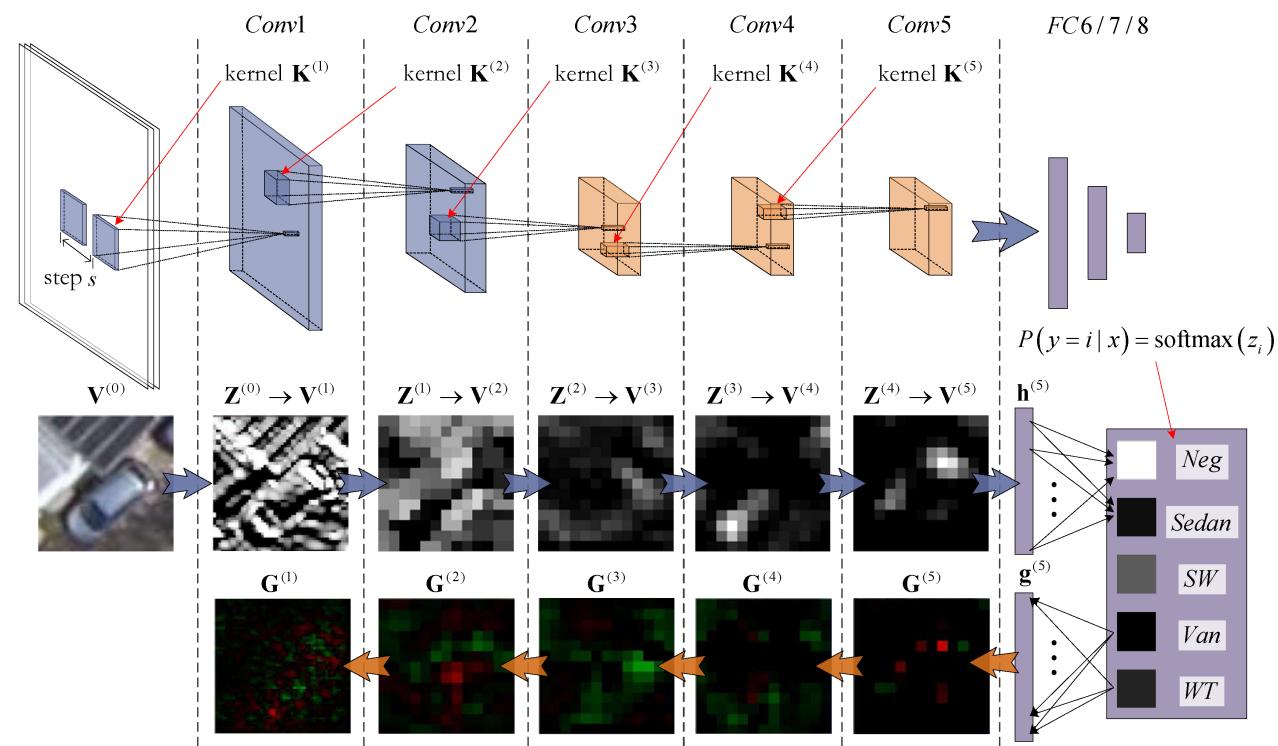
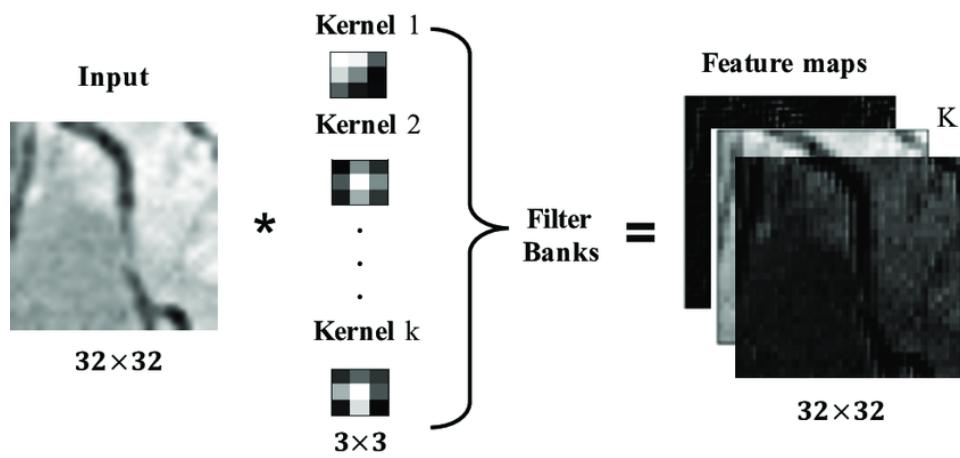
# Define the Model

- Define the layers
- "self.conv1": convolution layer, input channel 1, output channels 10, kernel size 5
- "self.conv2": convolution layer, input channel 10, output channels 20, kernel size 5

```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__()  
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)  
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)  
        self.fc1 = nn.Linear(320, 50)  
        self.fc2 = nn.Linear(50, 10)
```

# Convolution Layer

- Channels and kernel size



## Define the Model

- "self.fc1": fully connected layer, input size 320, output size 50
- "self.fc2": fully connected layer, input size 50, output size 10
- How to get the input size of self.fc1 ?

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
```

## Define the Forward Pass

- conv1 → pooling → ReLU → conv2 → pooling → ReLU → fc1 → ReLU → fc2

```
class Net(nn.Module):
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320) # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
    return x
```

## Instantiate the Model

- "network" is the model
- "optimizer" is for gradient descent
- Gradients are derived automatically

```
learning_rate = 0.01
# Instantiate the model
network = Net()
# Instantiate the optimizer
optimizer = optim.SGD(network.parameters(), lr=learning_rate)
```

## Training and Testing

- Epoch: a pass of training of **the dataset**
- Step: an iteration of gradient descent on **a batch of data**

```
for epoch in range(n_epochs):
    # Training
    for step, (data, target) in enumerate(train_loader):
        # A training step
    # Testing
    for step, (data, target) in enumerate(test_loader):
        # A testing step
```

## A Training Step

- "output" is the inferred results
- "loss" is the loss value
- "loss.backward()" computes the gradients
- "optimizer.step()" do gradient descent

```
# Inference
output = network(data)
# Compute the loss
loss = F.cross_entropy(output, target)
# Gradient descent
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

## A Testing Step

- "F.cross\_entropy" computes the loss
- "pred" is the prediction (the class with the maximum probability)
- "(pred == target).sum()" computes the accuracy

```
# Inference
output = network(data)
# Compute the loss
test_loss += F.cross_entropy(output, target).item()
# Get the prediction
pred = output.max(dim=1)[1]
# Count correct predictions
correct += (pred == target).sum()
```

# Experiments

- See `classification.ipynb`