High-level vision

Adrian F. Clark <alien@essex.ac.uk>
CSEE, University of Essex

What is *high-level* vision?

It might better to call this topic **CV meets ML**

Machine learning (ML) has been used in vision since its early days

The early ML researchers of the 1950s thought solving vision would take only a few years to do — how wrong they were!

Rather than going into a small number of techniques in detail, this will be a high-level overview of quite a wide range of them — given the title of the lecture, this somehow seems fitting

Supervised and unsupervised

There are two generic approaches to machine learning:

- **unsupervised**, such as k-means, in which the properties of the data are explored
- **supervised**, such as NN, GA, SVM *etc*, which require ground truthed training data to learn from

Supervised leaning normally achieves better results than unsupervised but take longer to learn — and you have to prepare ground-truthed training data

Unsupervised techniques are arguably most effective when exploring the properties of data

Unsupervised learning with k-means

Imagine that you have N data points which you know belong in k classes

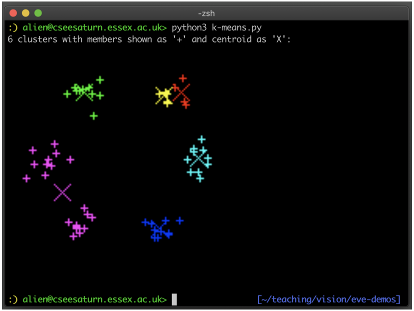
k-means involves two phases:

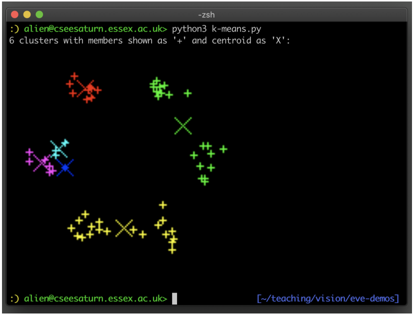
- assign data points to clusters based on distance from the cluster centroid
- calculate cluster centroids based on cluster membership

with the phases being iterated through until there are no changes (or say 100 times)

This doesn't necessarily work well, so it is normal to run it many times and take the best result


-zsh
:) alien@cseesaturn.essex.ac.uk> python3 k-means.py
6 clusters with members shown as '+' and centroid as 'X':
:) alien@cseesaturn.essex.ac.uk>
[~/teaching/vision/eve-demos]

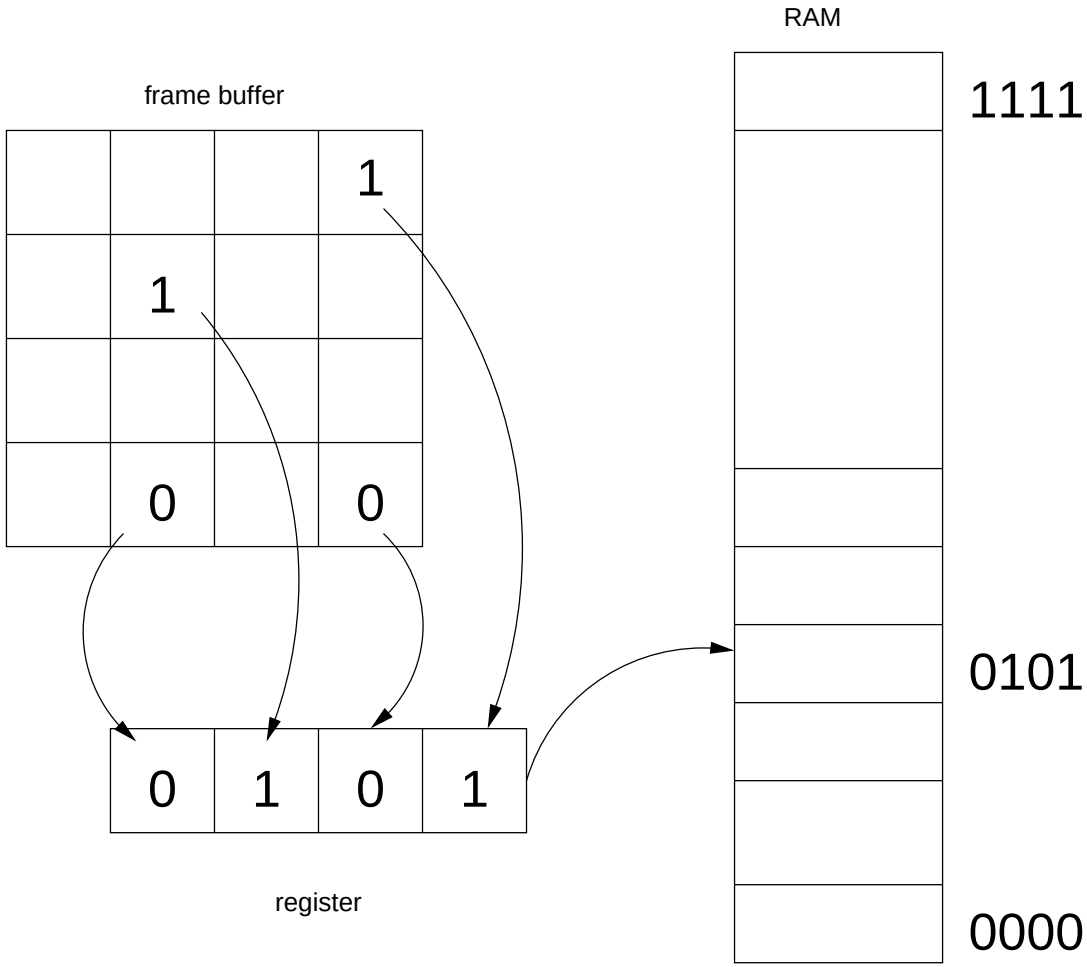

-zsh
:) alien@cseesaturn.essex.ac.uk> python3 k-means.py
6 clusters with members shown as '+' and centroid as 'X':
:) alien@cseesaturn.essex.ac.uk>
[~/teaching/vision/eve-demos]


-zsh
:) alien@cseesaturn.essex.ac.uk> python3 k-means.py
6 clusters with members shown as '+' and centroid as 'X':
:) alien@cseesaturn.essex.ac.uk>
[~/teaching/vision/eve-demos]


-zsh
:) alien@cseesaturn.essex.ac.uk> python3 k-means.py
6 clusters with members shown as '+' and centroid as 'X':
:) alien@cseesaturn.essex.ac.uk>
[~/teaching/vision/eve-demos]

Supervised learning with WISARD

Perhaps the simplest conceivable way of using image data to learn, easy to construct in hardware



How WISARD works

Values taken from random locations of a binarized image are concatenated to form an address (index into an array)

When training, the value at the address is set

When running, the value at the address is added to a *score*

Higher scores indicate a better match

How effective is WISARD?

On MNIST (10 classes, 60,000 training images, 10,000 test images), it achieves an accuracy of about 0.1 — about the same as random

With only 100 training images/class and 100 test images, it achieves an accuracy of 0.54 — which is moderate

Our conclusion is that WISARD's ability to store different patterns has been overwhelmed by the size of MNIST

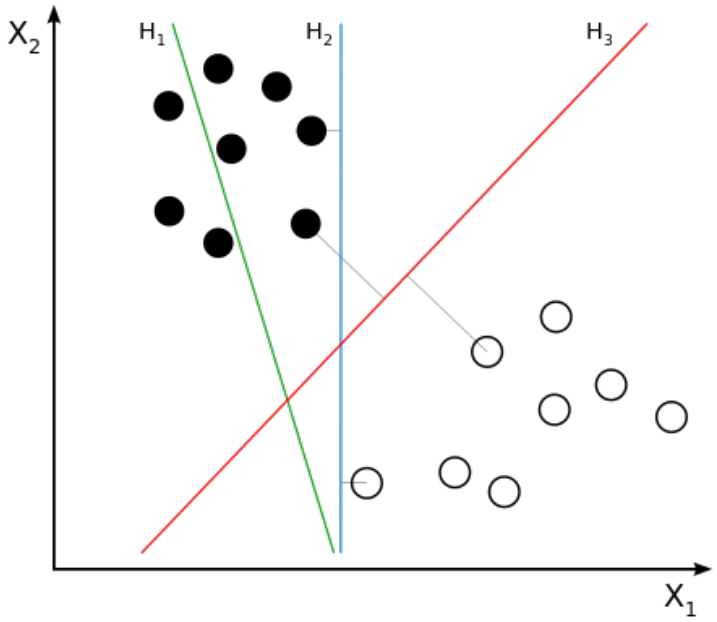
If we increased the size of WISARD's register and looked at more places in the image, it would likely do better

Support Vector Machine (SVM)

This was the standard ML technique in vision up to about 2012

The idea is to calculate the best-separating line (or hyperplane in > 2D) to the data

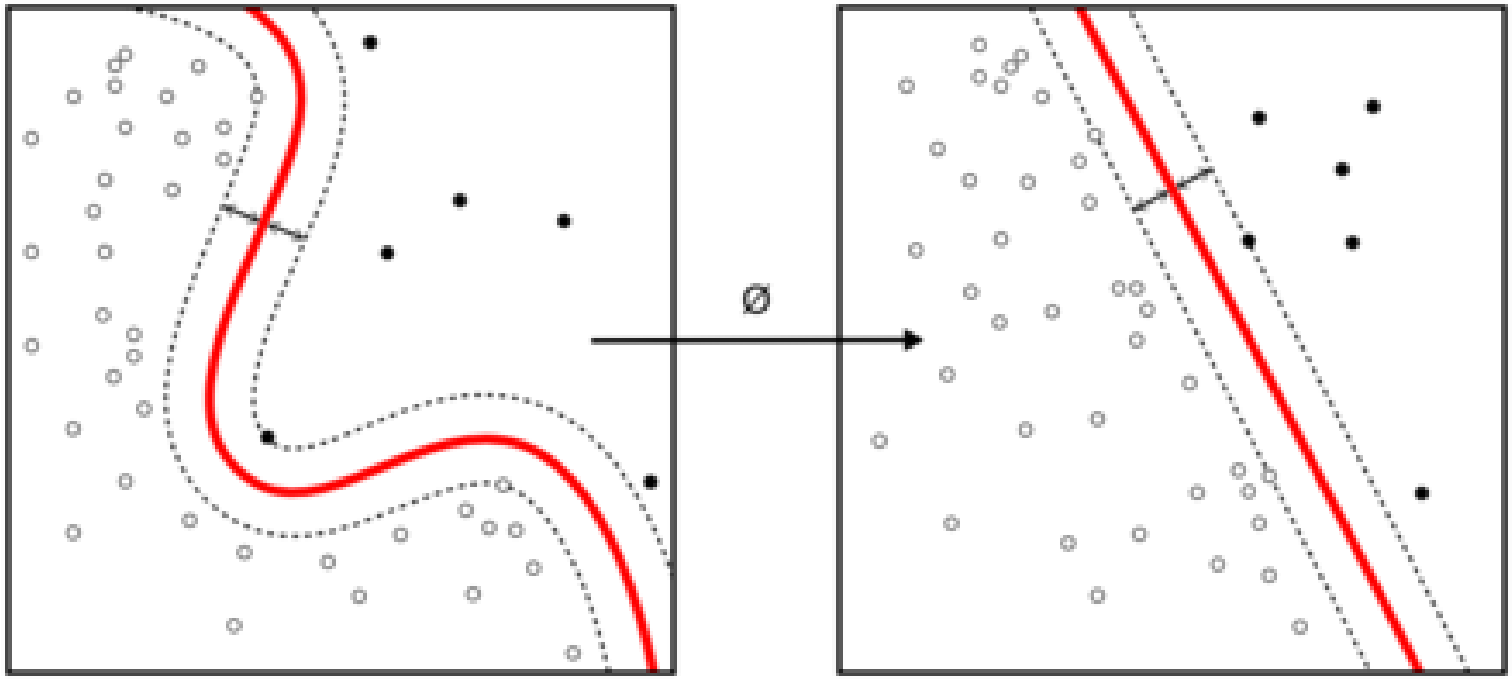
It is quite fast to run (no random numbers involved) and extends to many clusters and dimensions



Non-planar boundaries and SVM

My experience is that SVM needs to be tuned to perform well

SVM boundaries must be planar, so if this is not the case the data must be 'warped' before SVM is applied and 'unwarped' afterwards



Genetic Algorithms

This optimizes a set of numbers using a scheme that mimics sexual reproduction

Sets of numbers are combined into *chromosomes* and a population of them is generated randomly — say the six or so values (origin, rotation, translation) needed to align image patches

The effectiveness of each cluster on the problem is assessed using a *cost function*

Values are swapped between the best-performing individuals and sometimes mutated

I have used a real-valued GA on a number of problems and found it pretty effective, though quite slow

Genetic programming

This takes the ideas of the GA and applies it to program code, yielding programs such as

```
(if (> r 0)
    (if (> g 0) "yellow" "red")
    (if (> g 0) "green" "black")
)
```

which ascribes names to red, green, yellow and black pixels

[GP code is conventionally presented as Lisp]

This initially seems a crazy idea but works surprisingly well

GP is one of the few **explainable AI** techniques around

GP and vision

Much of the power of GP comes from the ability to define which operators are able to be used on a problem

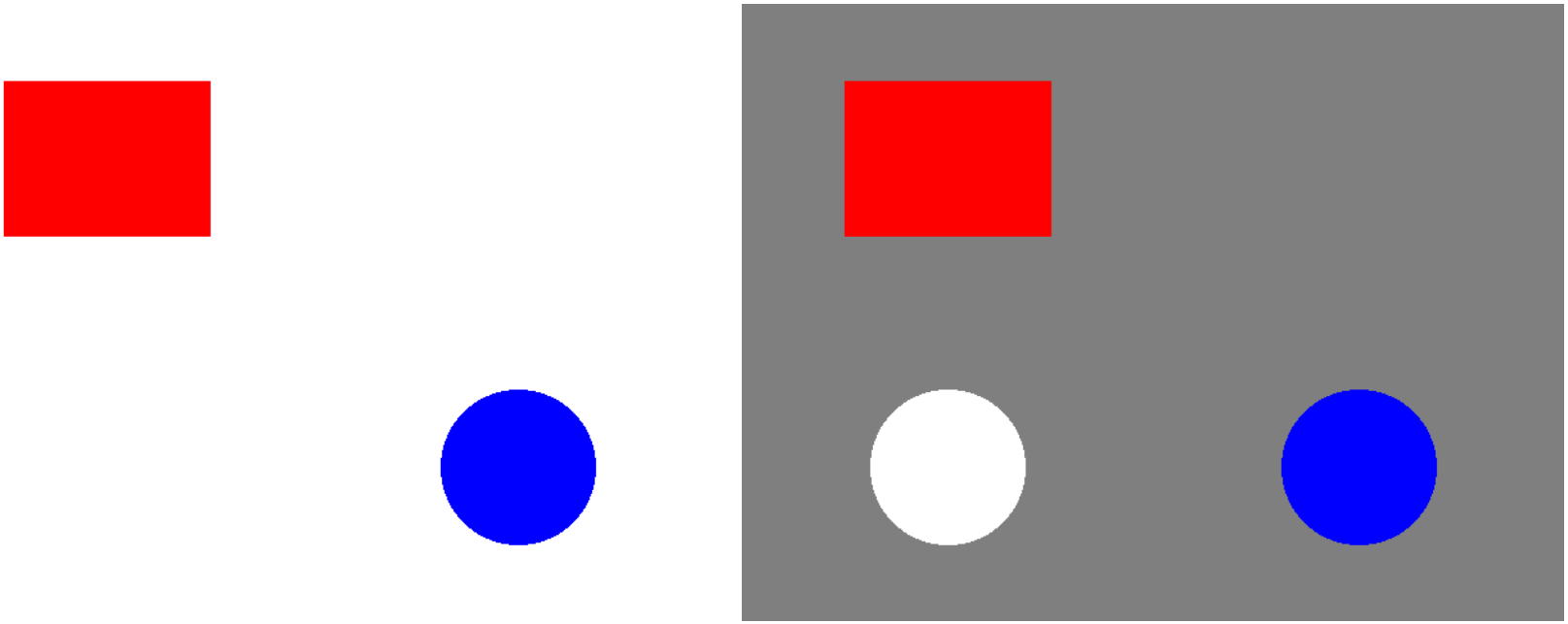
If we define operators similar to those you are using in your lab sessions, we can evolve programs that segment and classify images

Using the same set of operators and a cost function which simply measures the number of correctly-classified pixels, we have come up with a package, *ELVS*, which can solve a number of vision problems **entirely unchanged**

We believe it is the only system in the world that can do this

ELVS in action

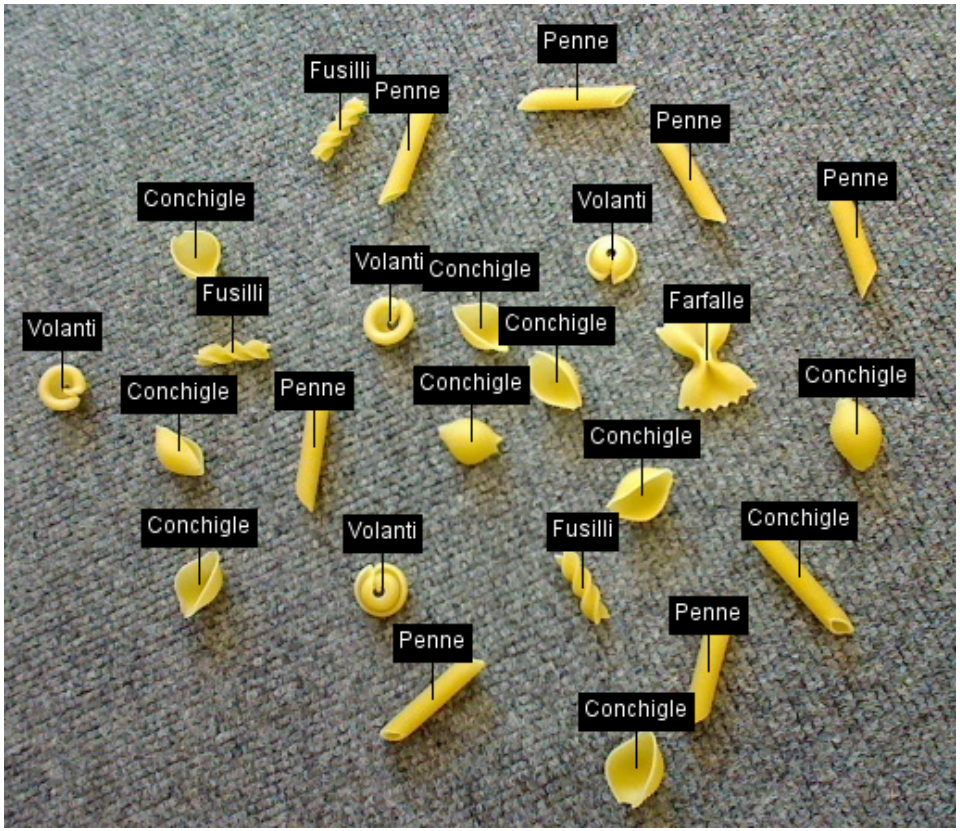
Given the training image and ground truth

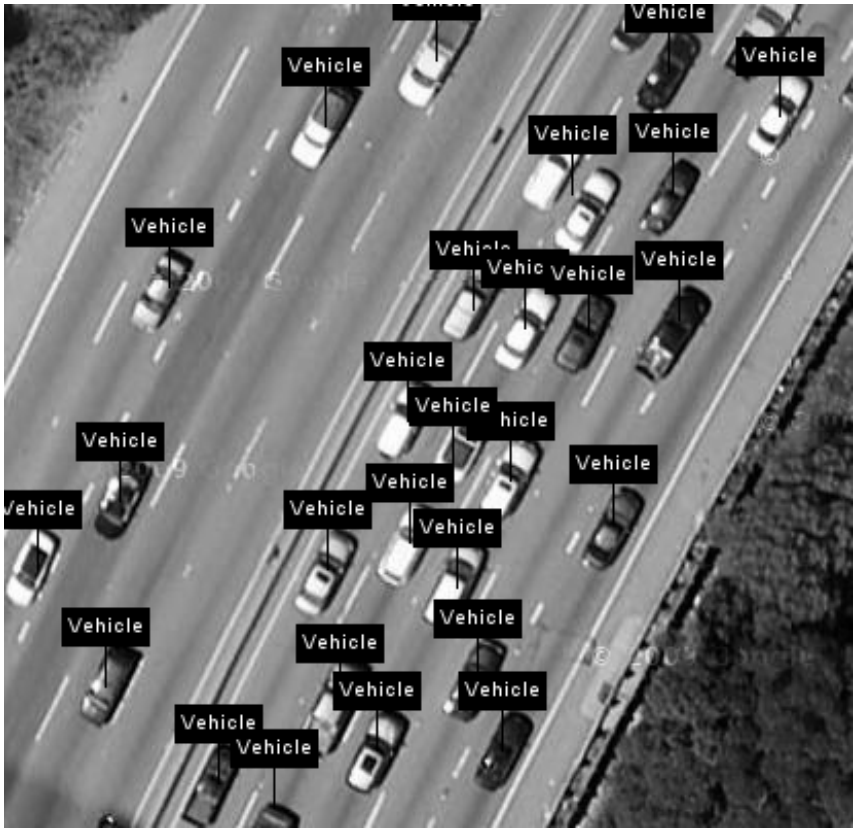


ELVS can evolve segmenters and classifiers such as:

```
(Otsu (grey input))                          # segment
(f< 1f (occupancy input))                    # classify as circle
(f> (height input) (width input))            # classify as rectangle
```

. . . and also. . .

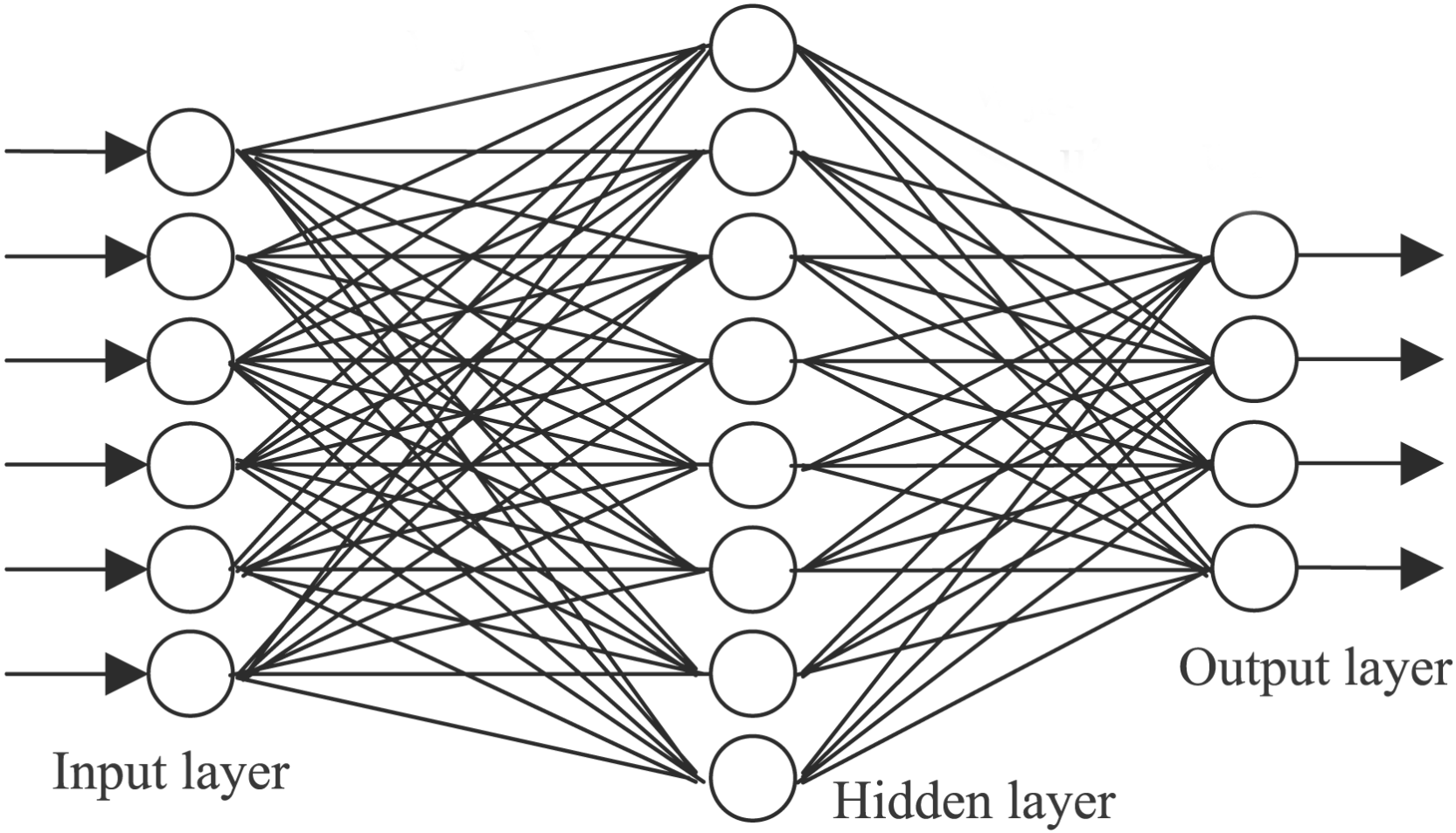

Penne
Fusilli
Penne
Penne
Penne
Conchigle
Volanti
Volanti
Conchigle
Fusilli
Farfalle
Volanti
Conchigle
Conchigle
Conchigle
Conchigle
Penne
Conchigle
Conchigle
Conchigle
Volanti
Fusilli
Conchigle
Penne
Penne
Conchigle


Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle
Vehicle

. . . and also

Multi-Layer Perceptron

The 'traditional' neural network architecture is the MLP



MLP neurons

Each neuron simple weights its inputs x_i by values w_i, adds the results together and passes them through an *activation function* f:

$$y = f\left(\sum_i w_i x_i\right)$$

The weights w_i are learned using *back-propagation*

Training data

If the amount of training data is small, the MLP tends to learn each input pattern separately and doesn't *generalize* ($\Rightarrow$ classify similar patterns correctly), which is usually what we want

So the amount of training data needs to be large (say 10^4 per class)

If we have only a small training set, we use *data augmentation* (adding noise, rotating, translating *etc* inputs) to — hopefully — produce a trained network able to generalize

Convolutional neural networks

Devised by Yann LeCun to read zip-codes on handwritten envelopes in the USA

The first CNN of note was AlexNet, which won an image recognition competition by a large margin

They have several different types of layers, the most important of which is a *convolution* layer whose coefficients are learned

CNNs are currently regarded as the state of the art in vision

CNN layer types

INPUT layer — the raw data (RGB or HSV) are presented here

CONV layer — this performs a convolution on a region of the image

RELU layer — this applies the activation function of an MLP

POOL layer — this down-samples or averages several pixels into one

FC layer — just as in an MLP

Putting these together in different ways results in different networks

CNN architectures of note

GoogLeNet

VVGnet

ResNet

See the lecture notes for diagrams of these architectures — and note that the number of layers keeps on growing!

These deep networks can take weeks to train on commodity hardware, though software such as TensorFlow and Caffe can exploit the SIMD processors on some graphics cards to speed this up

Transfer learning

It is becoming normal to take an existing network, pre-trained on (say) the ImageNet database, then 'tap off' layers that look useful

Use these as inputs to a MLP or other supervised learner

Present your training data to the pre-trained network, use its outputs to help train your own learner on your training data

You can argue that this is using the CNN as a feature detector

My experience is that this works fairly well

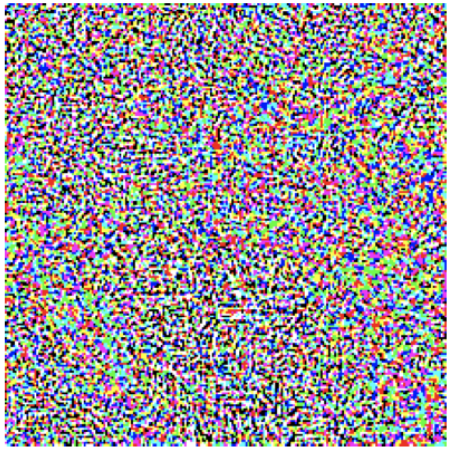
. . . but. . .

Visually identical inputs can yield high-confidence but different classifications — a real problem

$\boldsymbol{x}$

"panda"
57.7% confidence

$+ .007 \times$



$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
8.2% confidence

$=$

$\boldsymbol{x} + \epsilon \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
99.3 % confidence

Generative Adversarial Networks (GANs)

Uses two networks:

- a *generator* produces data
- a *discriminator* attempts to classify the data

Both are trained concurrently, with the generator trying to fool the discriminator and the discriminator trying to spot the generator's fakes

This is pretty much the state of the art, though in reality the idea is not a new one: it is *co-evolution* in GP, for example

Remember my comment in an earlier lecture about the importance of teaching a system to discard invalid images when training

Shortcomings of the current deep learning approach

The impressive results you read about are often the result of extensive manual tweaking of hyper-parameters

There is no design methodology

We don't know how they work — such "black boxes" are unacceptable in some domains

My opinion is that, in terms of pushing computer vision forward as a discipline, deep networks are **excellent engineering but poor science** because they yield solutions but don't give insights into their workings or how you can apply them to other problems

As GP lets a human review how evolved programs work, the approach gives much more insight into its solutions