Never Stand Still

Neural Learning

COMP9417 Machine Learning & Data Mining

Aims

This lecture will develop your understanding of Neural Network Learning & will extend that to Deep Learning

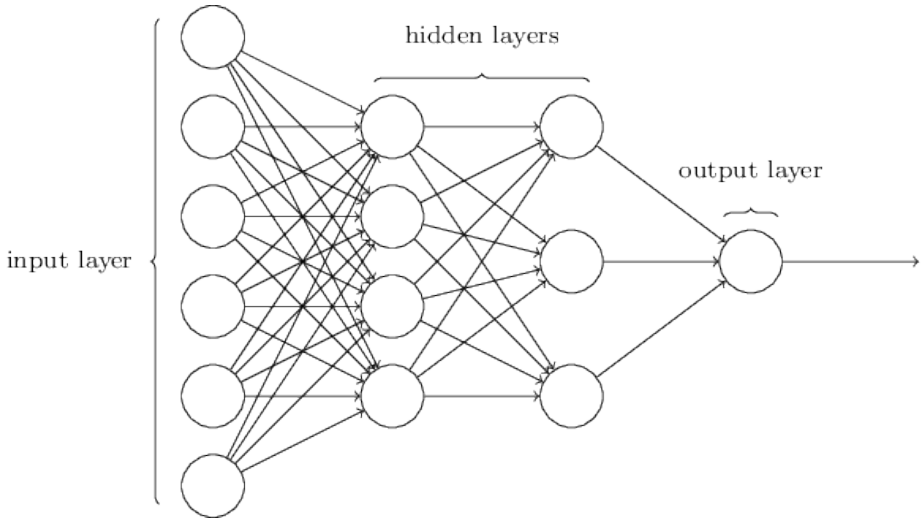
- describe Perceptrons and how to train them
- relate neural learning to optimization in machine learning
- outline the problem of neural learning
- derive the Gradient Descent for linear models
- describe the problem of non-linear models with neural networks
- outline the method of back-propagation training of a multi-layer
- understand Convolutional Neural Networks (CNNs)
- understand the main difference between CNN and regular NN
- know the basics of training CNN

Artificial Neural Networks

Artificial Neural Networks are inspired by human nervous system

NNs are composed of a large number of interconnected processing elements known as neurons

They use supervised error correcting rules with back-propagation to learn a specific task



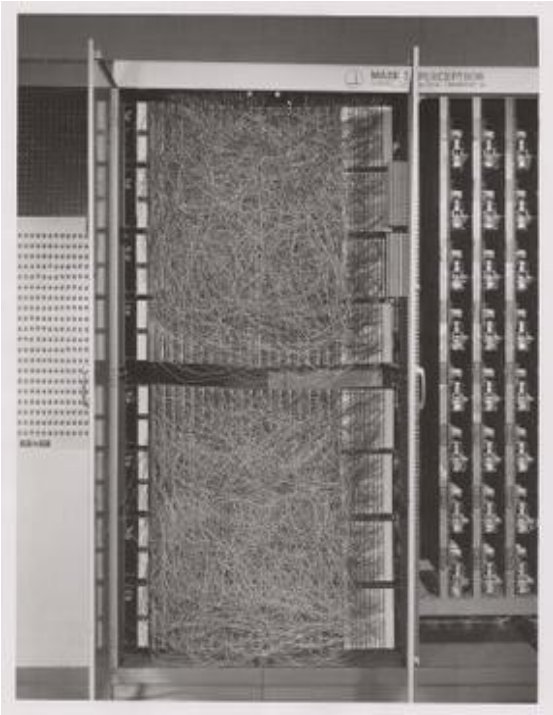
http://statsmaths.github.io/stat665/lectures/lec12/lecture12.pdf

Perceptron

A linear classifier that can achieve perfect separation on linearly separable data is the *perceptron* - a simplified neuron, originally proposed as a simple *neural network* by F. Rosenblatt in the late 1950s.

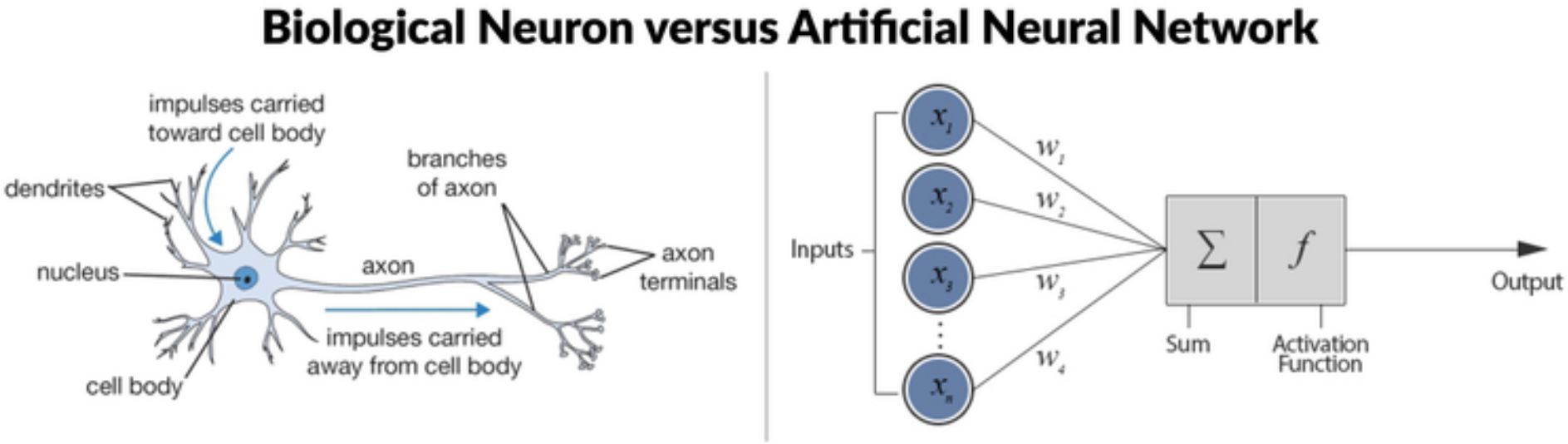
Perceptron

Originally implemented in software (based on the McCulloch-Pitts neuron from the 1940s), then in hardware as a 20x20 visual sensor array with potentiometers for adaptive weights.



Source http://en.wikipedia.org/w/index.php?curid=47541432

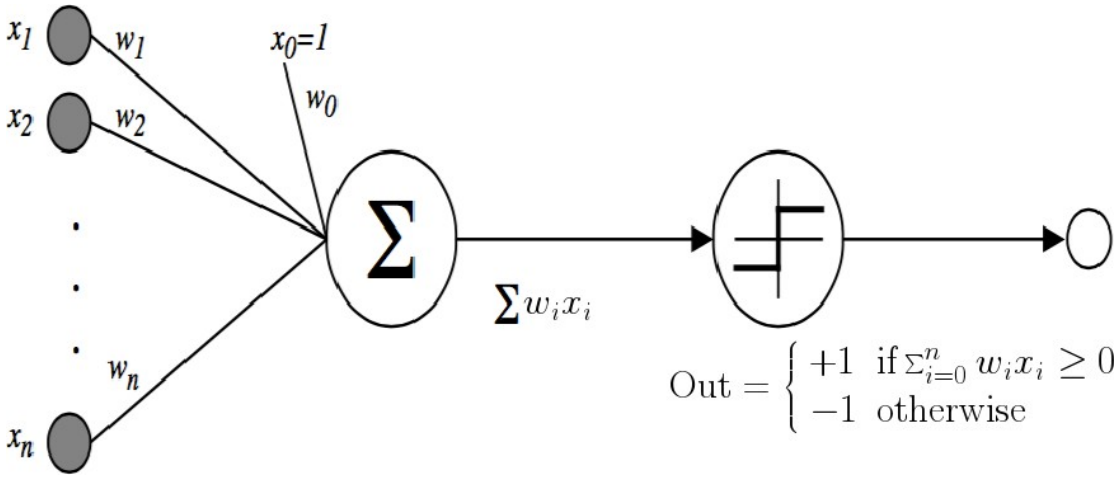
Perceptron



Each neuron has multiple dendrites and a single axon. The neuron receives its inputs from its dendrites and transmits its output through its axon. Both inputs and outputs take the form of electrical impulses. The neuron sums up its inputs, and if the total electrical impulse strength exceeds the neuron's firing threshold, the neuron fires off a new impulse along its single axon. The axon, in turn, distributes the signal along its branching synapses which collectively reach thousands of neighboring neurons.

https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b

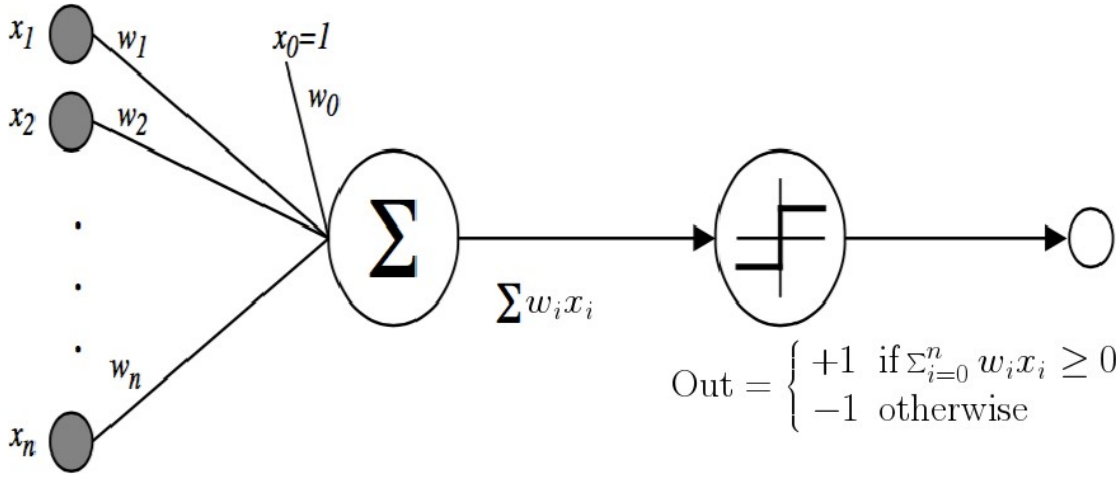
Perceptron



Output *o* is thresholded sum of products of inputs and their weights:

$$o(x_1, \ldots, x_n) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

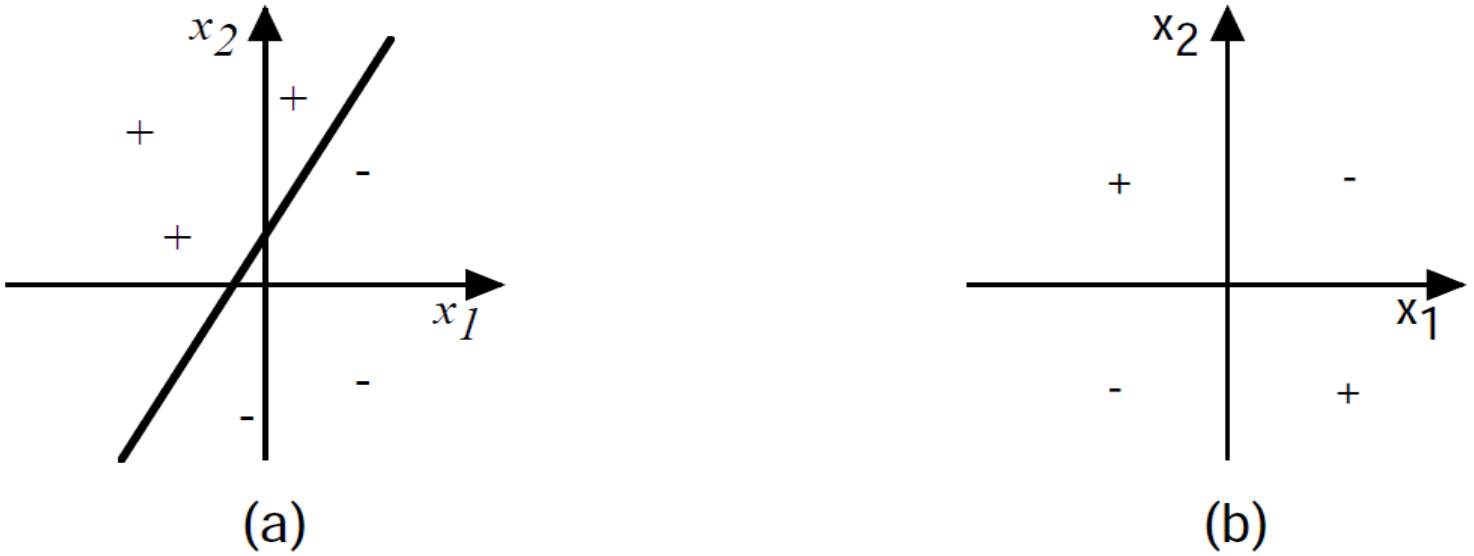
Perceptron



Or in vector notation:

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Decision Surface of a Perceptron



- Perceptron is able to represent some useful functions which are linearly separable (a)
- But functions that are not linearly separable are not representable (e.g. (b) XOR)

Perceptron Learning

Key idea:

Learning is "finding a good set of weights"

Perceptron learning is simply an iterative weight-update scheme:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \Delta\mathbf{w}_i$$

where the weight update $\Delta\mathbf{w}_i$ depends only on *misclassified* examples and is modulated by a "smoothing" parameter η typically referred to as the "learning rate".

Perceptron Learning

The perceptron iterates over the training set, updating the weight vector every time it encounters an incorrectly classified example.

- Let x_i be a misclassified positive example, then we have $y_i = +1$ and $w.\mathrm{x}_i < 0$. We therefore want to find w' such that $w'.\mathrm{x} > \mathrm{w}\,.\,\mathrm{x}$, which moves the decision boundary towards and hopefully past x_i.

- This can be achieved by calculating the new weight vector as $w' = w + \eta \mathrm{x}_i$, where $0 < \eta \leq 1$ is the *learning rate* (again, assume set to 1). We then have $w'.\mathrm{x}_i = w\,.\,\mathrm{x}_i + \eta \mathrm{x}_i.\mathrm{x}_i > w\,.\,\mathrm{x}_i$ ⬚ as required.

- Similarly, if x_j is a misclassified negative example, then we have $y_j = -1$ and $w.\mathrm{x}_j > 0$. In this case we calculate the new weight vector as $w' = w - \eta \mathrm{x}_j$, and thus $w'.\mathrm{x}_j = w.\mathrm{x}_j - \eta \mathrm{x}_j.\mathrm{x}_j < w.\mathrm{x}_j$.

Perceptron Learning

- The two cases can be combined in a single update rule:

$$w' = w + \eta y_i \mathrm{x}_i$$

- Here y_i acts to change the sign of the update, corresponding to whether a positive or negative example was misclassified
- This is the basis of the *perceptron training algorithm* for linear classification
- The algorithm just iterates over the training examples applying the weight update rule until all the examples are correctly classified
- If there is a linear model that separates the positive from the negative examples, i.e., the data is linearly separable, it can be shown that the perceptron training algorithm will converge in a finite number of steps.

Training Perceptron

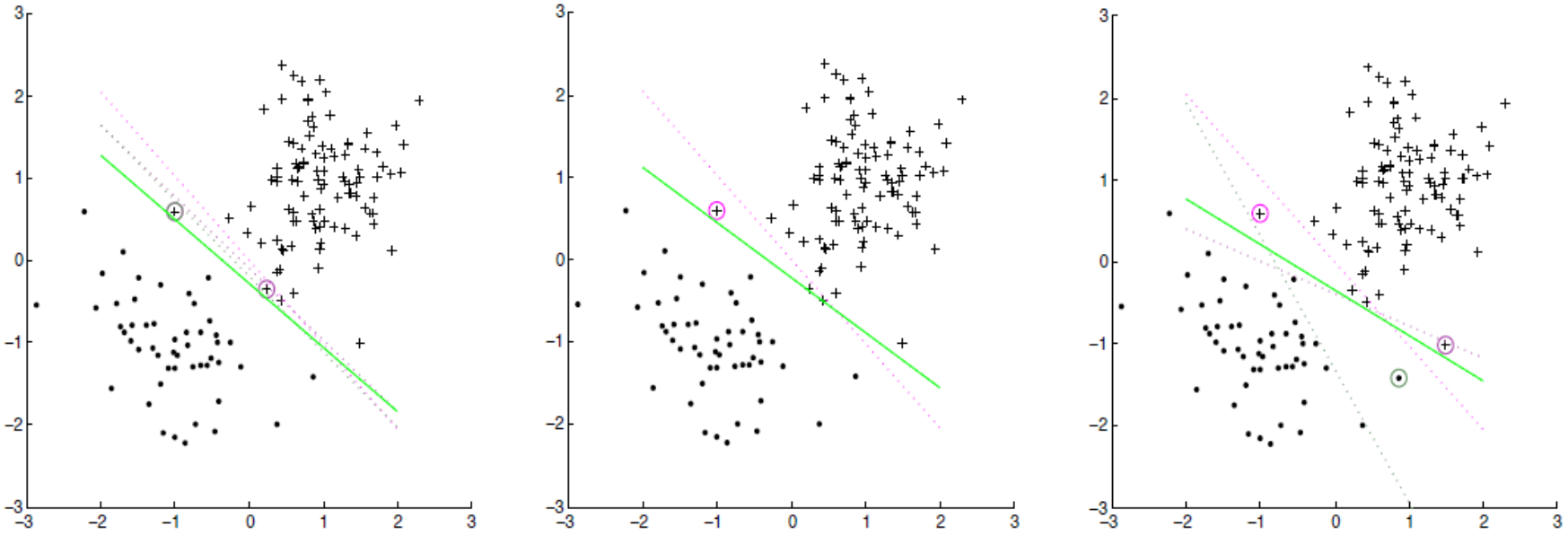
Algorithm Perceptron(D, η) // perceptron training for linear classification

Input: labelled training data D in homogeneous coordinates; learning rate η.

Output: weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

```
1  w ← 0 // Other initialisations of the weight vector are possible
2  converged ← false
3  while converged = false do
4      converged ← true
5      for i = 1 to |D| do
6          if y_i w · x_i ≤ 0 then          // i.e., ŷ_i ≠ y_i
7              w ← w + η y_i x_i
8              converged ← false  // We changed w so haven't converged yet
9          end
10     end
11 end
```

Perceptron Learning Rate



(left) A perceptron trained with a small learning rate ($\eta = 0.2$). The circled examples are the ones that trigger the weight update.
(middle) Increasing the learning rate to $\eta = 0.5$ leads in this case to a rapid convergence.
(right) Increasing the learning rate further to $\eta = 1$ may lead to too aggressive weight updating, which harms convergence.

Perceptron Convergence

Perceptron training will converge (under some mild assumptions) for linearly separable classification problems

A labelled data set is linearly separable if there is a linear decision boundary that separates the classes

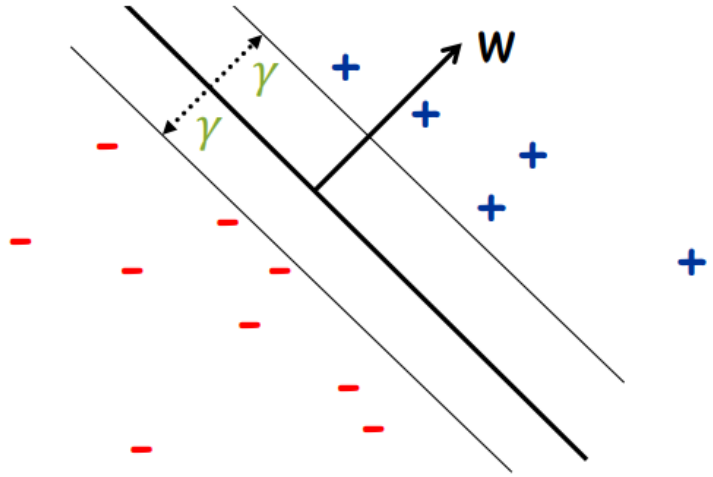
Perceptron Convergence

Dataset $D = \{(\mathrm{x}_1, y_1), \ldots, (\mathrm{x}_m, y_m)\}$

At least one example in D is labelled +1, and one is labelled -1.

A weight vector w^* exists s.t. $\|w^*\|_2 = 1$ and $\forall i \; y_i w^*.\mathrm{x}_i \geq \gamma$

γ is typically referred to as the "margin"

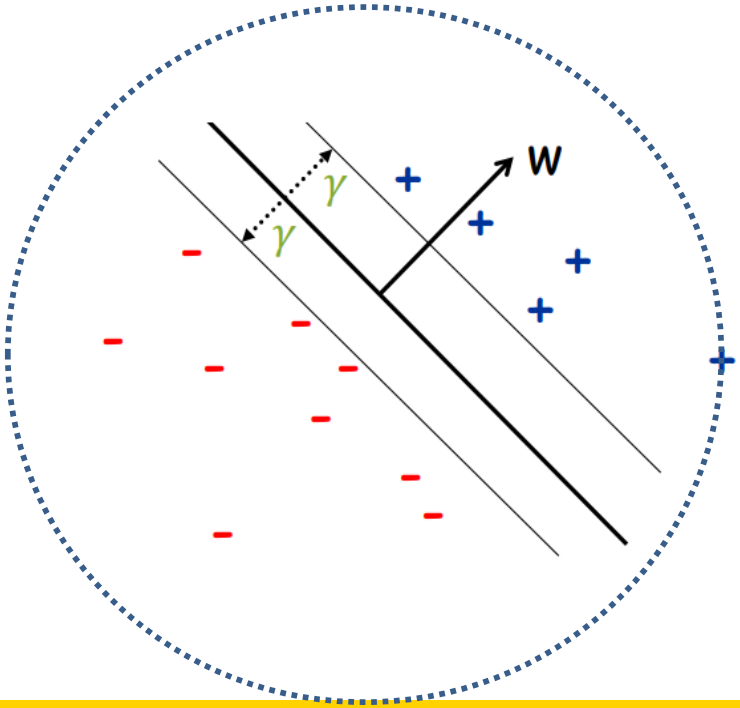


Perceptron Convergence

Perceptron Convergence Theorem (Novikoff, 1962)

$$R = max_i \ \|\mathrm{x}_i\|_2$$

The number of mistakes made by the perceptron is at most $\left(\frac{R}{\gamma}\right)^2$



Decision Surface of a Perceptron

- Unfortunately, as a linear classifier perceptrons are limited in expressive power
- So some functions not representable, e.g., not linearly separable
- For non-linearly separable data we'll need something else
- However, with a relatively minor modification many perceptrons can be combined together to form one model
  - *multilayer perceptrons*, the classic "neural network"

Optimisation

Studied in many fields such as engineering, science, economics, . . .

A general optimisation algorithm: [1]

1) start with initial point $\mathrm{x} = \mathrm{x}_0$
2) select a search direction p, usually to decrease $f(\mathrm{x})$
3) select a step length η
4) Set $s = \eta\, \mathrm{p}$
5) Set $\mathrm{x} = \mathrm{x} + \mathrm{s}$
6) go to step 2, unless convergence criteria are met

For example, could minimise a real-valued function f

Note: convergence criteria will be problem-specific.

[1] B. Ripley (1996) "Pattern Recognition and Neural Networks", CUP.

Optimisation

Usually, we would like the optimisation algorithm to quickly reach an answer that is close to being the right one.

- typically, we need to minimise a function
  - e.g., error or loss
  - optimisation is known as *gradient descent* or *steepest descent*
- sometimes, we need to maximise a function
  - e.g., probability or likelihood
  - optimisation is known as *gradient ascent* or *steepest ascent*

Gradient Descent

To understand, consider the simple linear unit, where

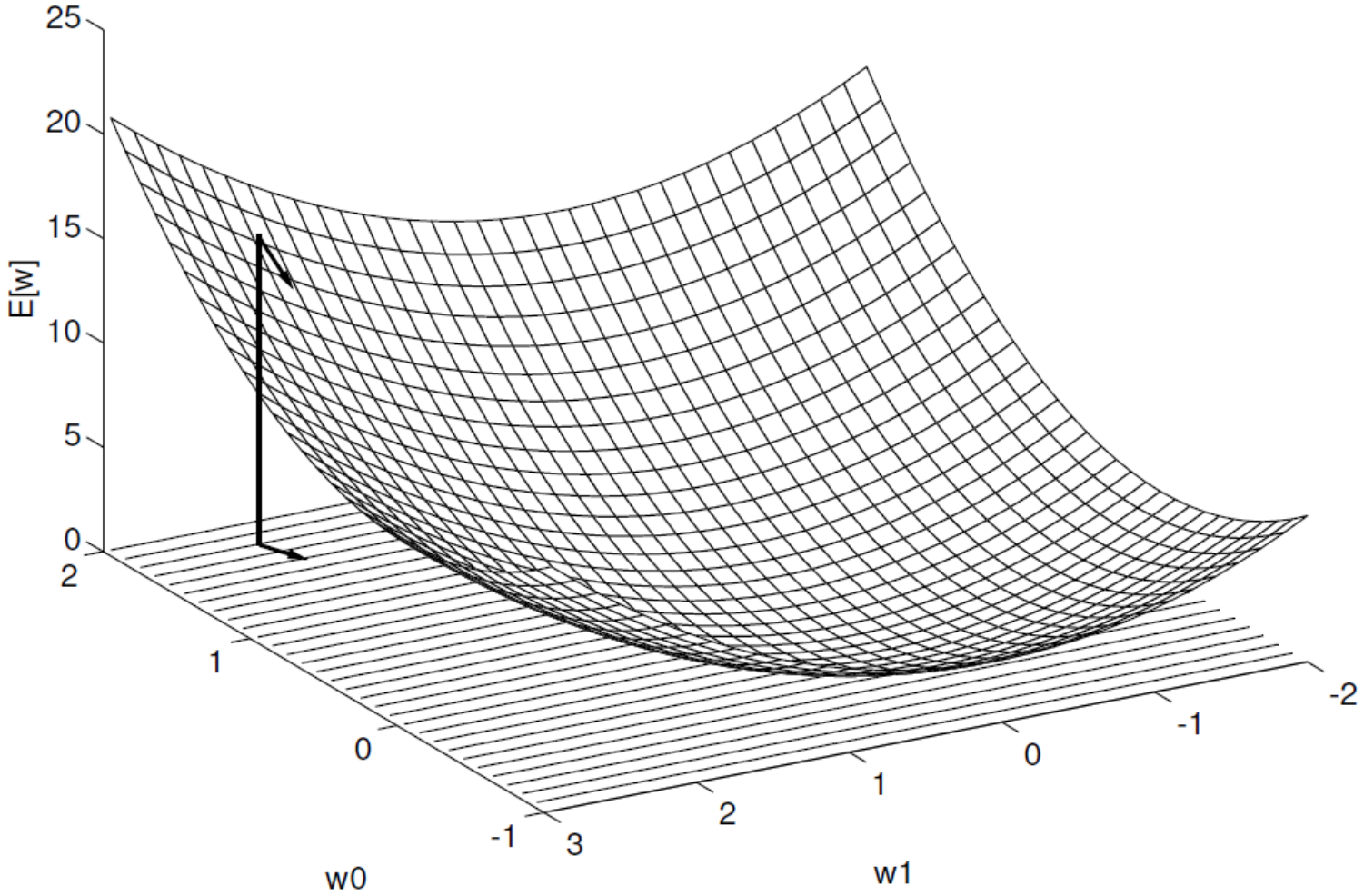
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Let's learn w_i that minimise the squared error

$$E[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is the set of training samples

Gradient Descent



Gradient Descent

Gradient:

$$\nabla E[\mathbf{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Gradient vector gives direction of *steepest increase* in error E

Negative of the gradient, i.e., *steepest decrease*, is what we want

Training rule: $$\Delta \mathbf{w} = -\eta \nabla E[\mathbf{w}]$$

i.e., $$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \mathbf{w}\cdot\mathbf{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}
$$

Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair $\langle \mathbf{x}, t \rangle$*, where* $\mathbf{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

Initialize each w_i to some small random value

Until the termination condition is met, Do

  Initialize each Δw_i to zero

  For each $\langle \mathbf{x}, t \rangle$ in $training\_examples$, Do

    Input the instance $\mathbf{x}$ to the unit and compute the output o

    For each linear unit weight w_i

      $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$

  For each linear unit weight w_i

    $w_i \leftarrow w_i + \Delta w_i$

Perceptron vs. Linear Unit

Perceptron training rule guaranteed to succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by H

Stochastic (Incremental) Gradient Descent

Batch mode Gradient Descent:

Do until satisfied

- Compute the gradient $\nabla E_D[\mathbf{w}]$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

$$E_D[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Stochastic (incremental) mode Gradient Descent:

Do until satisfied

- For each training example d in D
  - Compute the gradient $\nabla E_d[\mathbf{w}]$
  - $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

$$E_d[\mathbf{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) can approximate Batch Gradient Descent arbitrarily closely, if η made small enough
- Very useful for training large networks, or online learning from data streams
- Stochastic implies examples should be selected at random

Multilayer Networks

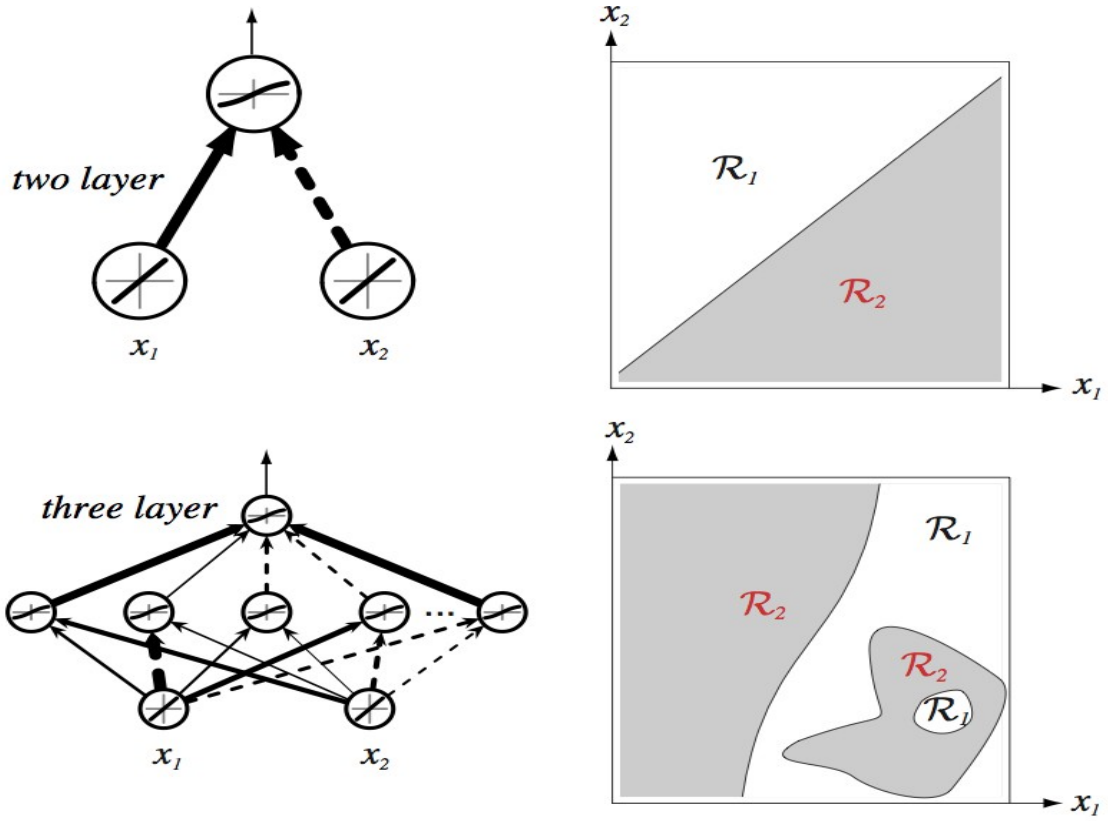
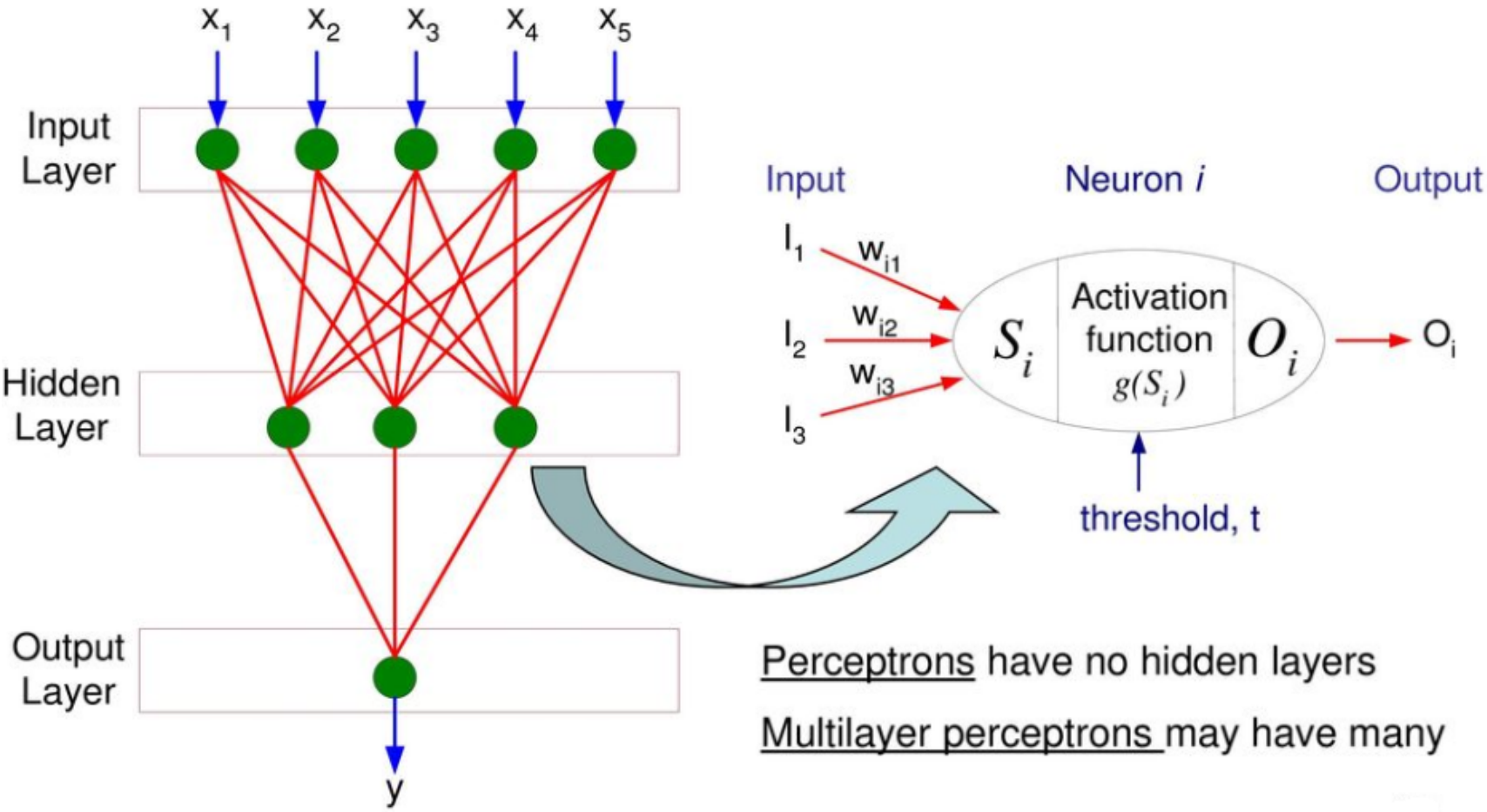


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Multilayer Networks

- Multi-layer networks can represent arbitrary functions, but an effective learning algorithm for such networks was thought to be difficult
- A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next (with activation feeding forward)
- The weights determine the function computed.
- Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer

General Structure of ANN



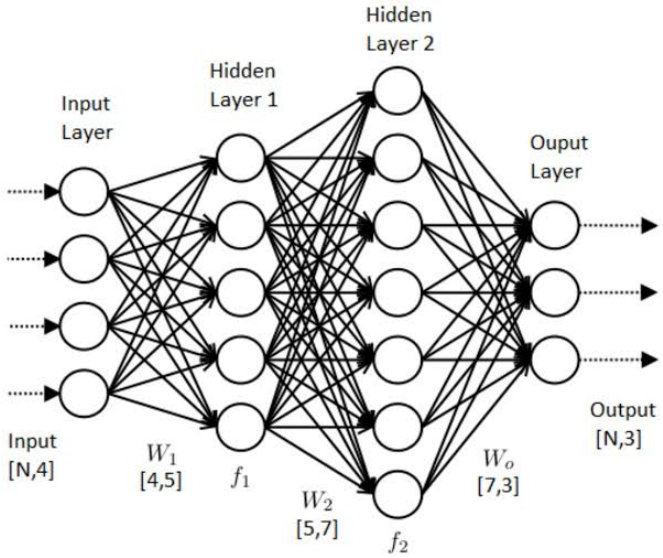
Perceptrons have no hidden layers

Multilayer perceptrons may have many

General Structure of ANN

Properties of Artificial Neural Networks (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically



Artificial Neural Network (Source: VIASAT)

When to Consider Neural Networks

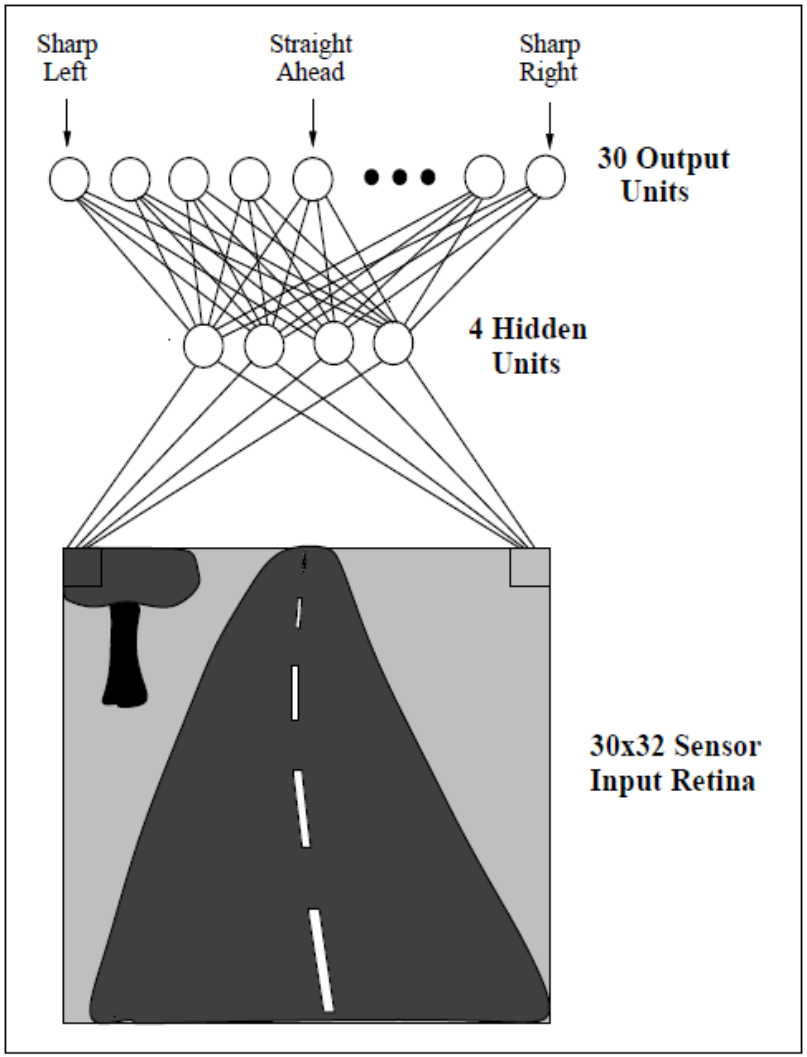
- Input is high-dimensional discrete or real-valued (e.g., raw sensor input)
- Output can be discrete or real-valued
- Output can be a vector of values
- Possibly noisy data
- Form of target function is unknown
- Interpretability of result is not important

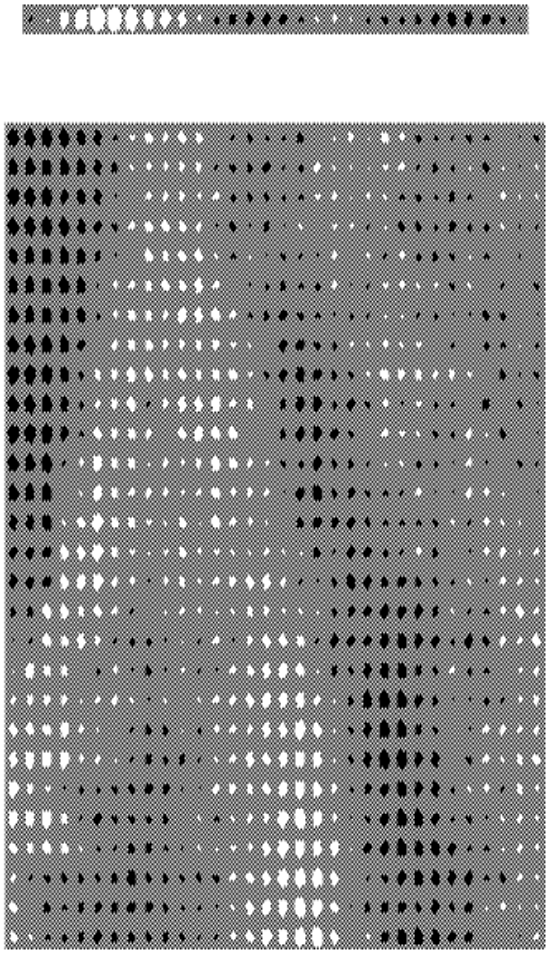
Examples:

- Speech recognition
- Image classification
- many others . . .

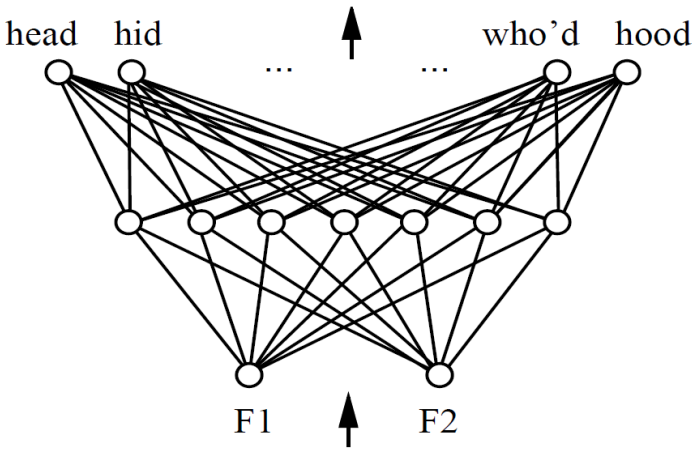
ALVINN drives 70 mph on highways

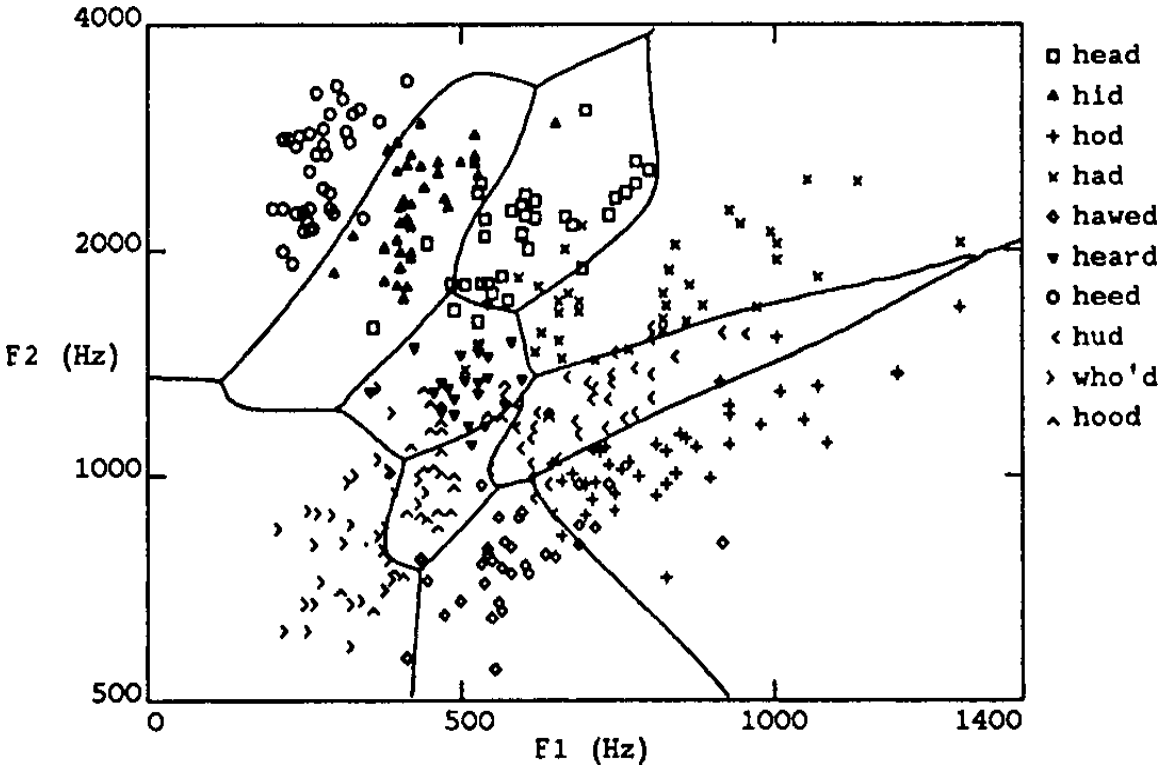
ALVINN


Sharp
Left
Straight
Ahead
Sharp
Right
30 Output
Units
4 Hidden
Units
30x32 Sensor
Input Retina



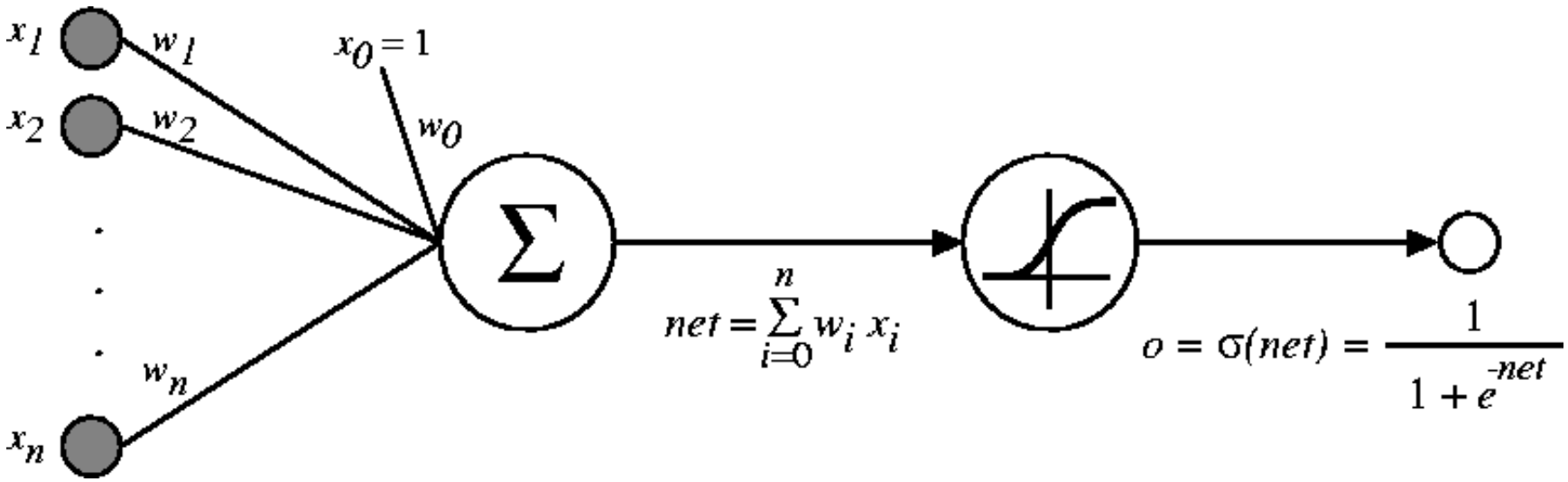
MLP Speech Recognition





Decision Boundaries

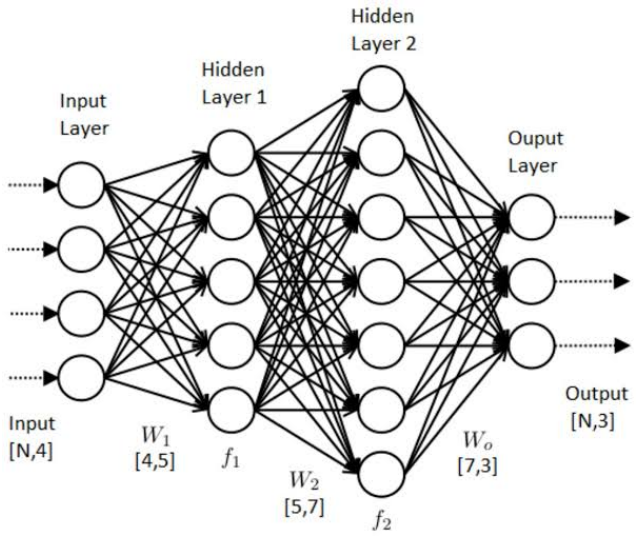
Sigmoid Unit



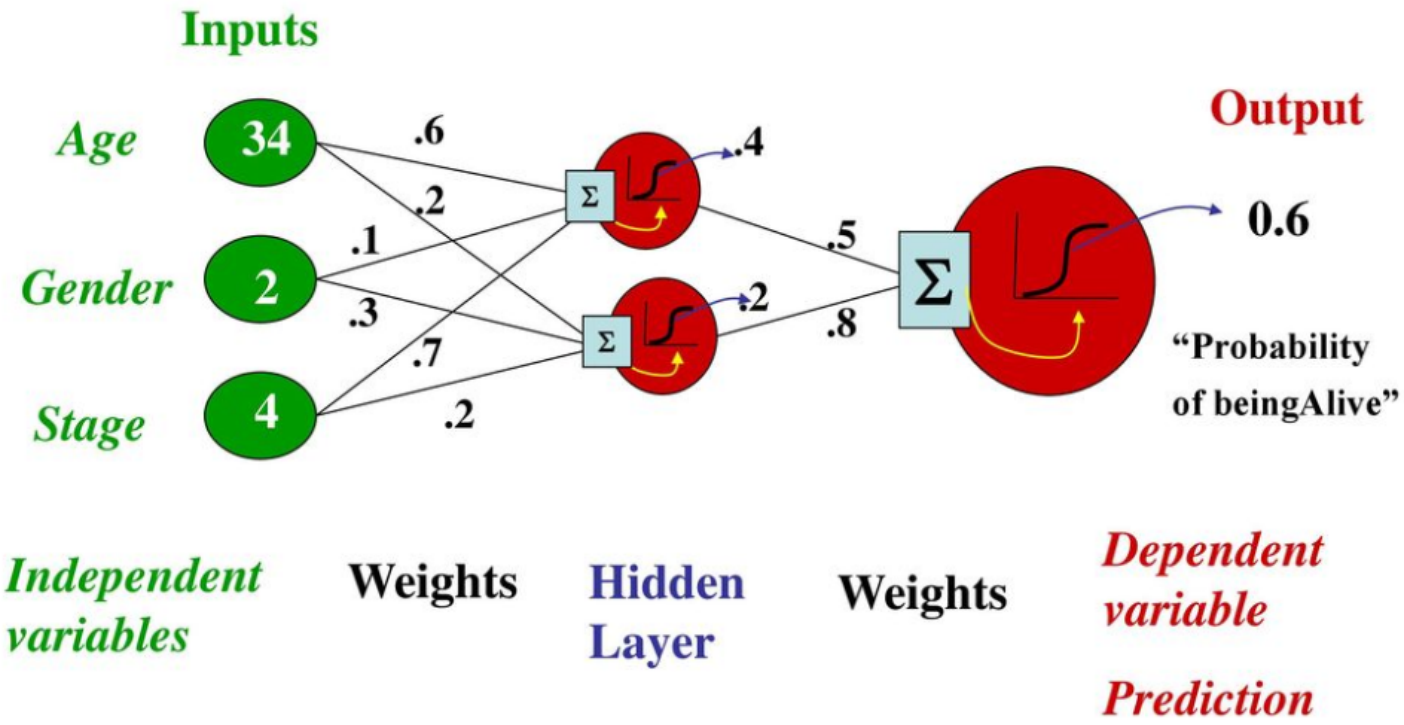
Sigmoid Unit

- Same as a perceptron except that the step function has been replaced by a nonlinear sigmoid function.

- Nonlinearity makes it easy for the model to generalise or adapt with variety of data and to differentiate between the output.



Sigmoid Unit



Sigmoid Unit

Why use the sigmoid function $\sigma(x)$?

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Note: in practice, particularly for deep networks, sigmoid functions are less common than other non-linear activation functions that are easier to train, but Sigmoids are mathematically convenient.

Error Gradient of Sigmoid Unit

Start by assuming we want to minimise squared error ($\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$) over a set of training examples D.

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\ \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d} 2(t_d - o_d)\ \frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d}(t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_{d}(t_d - o_d)\ \frac{\partial o_d}{\partial net_d}\ \frac{\partial net_d}{\partial w_i}
\end{aligned}
$$

Error Gradient of Sigmoid Unit

We know:

$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\mathbf{w} \cdot \mathbf{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d \in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied, Do

For each training example, Do

Input the training example to the network and compute the network outputs

For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

For each hidden unit h

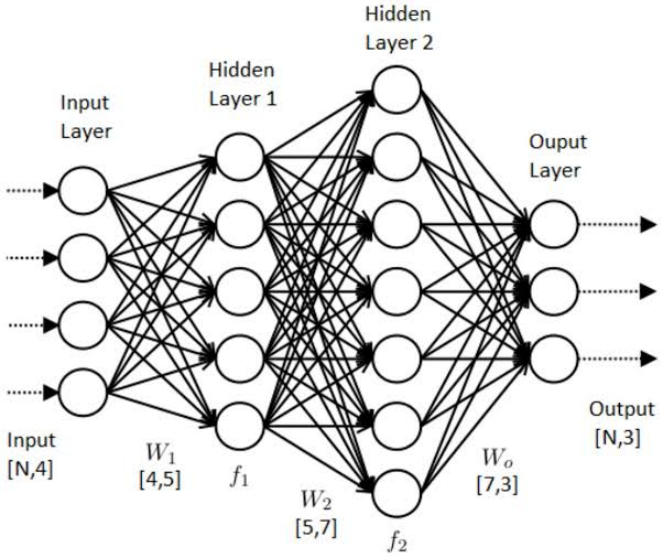
$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$



More on Backpropagation

A solution for learning highly complex models . . .

- Gradient descent over entire network weight vector
- Easily generalised to arbitrary directed graphs
- Can learn probabilistic models by maximising likelihood

Minimises error over all training examples

- Training can take thousands of iterations → slow!
- Using network after training is very fast

More on Backpropagation

Will converge to a local, not necessarily global, error minimum

- Might exist many such local minima
- In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- Stochastic gradient descent using "mini-batches"

Nature of convergence

- Initialise weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions become possible as training progresses

More on Backpropagation

Models can be very complex

- Will network generalise well to subsequent examples?
  - may *underfit* by stopping too soon
  - may *overfit* . . .

Many ways to regularise network, making it less likely to overfit

- Add term to error that increases with magnitude of weight vector

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Other ways to penalise large weights, e.g., weight decay
- Using "tied" or shared set of weights, e.g., by setting all weights to their mean after computing the weight updates
- Many other ways . . .

Expressive Capabilities of ANNs

Boolean functions:

- Every Boolean function can be represented by a network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by a network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988]

Being able to approximate any function is one thing, being able to *learn* it is another . . .

“Goodness of fit” in ANNs

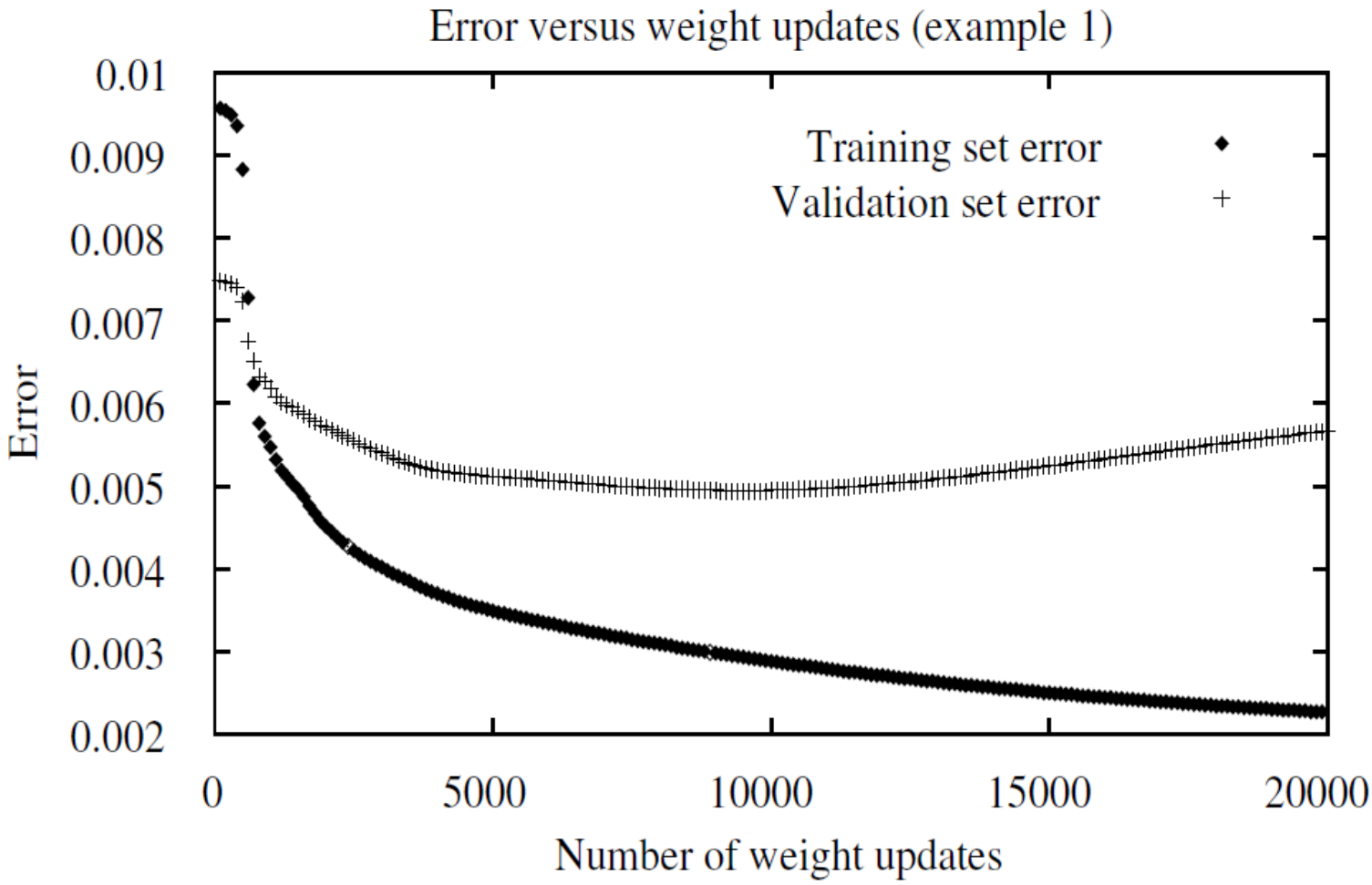
Can neural networks overfit/underfit ?

Next two slides: plots of “learning curves” for error as the network learns (shown by number of weight updates) on two different robot perception tasks.

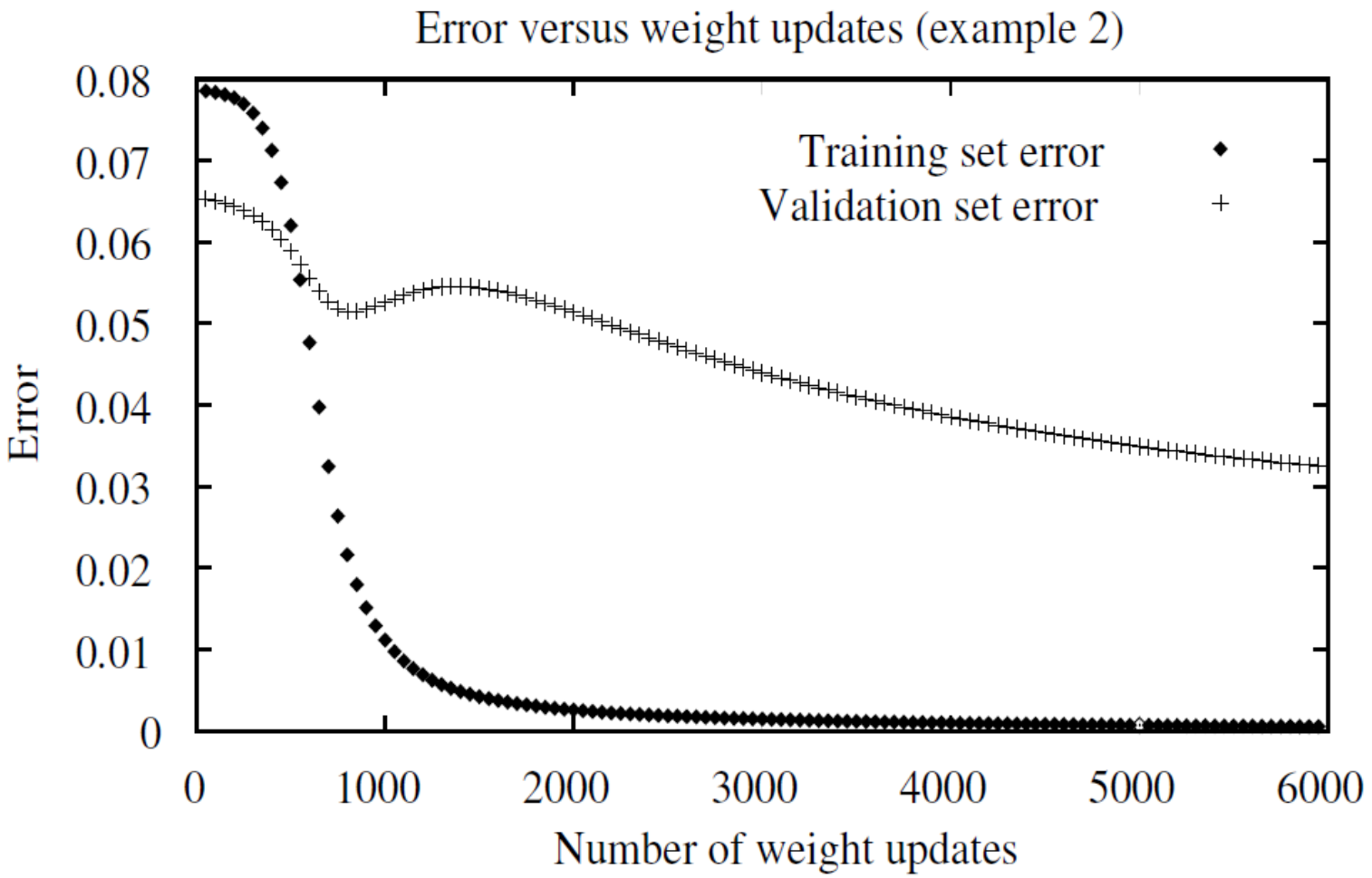
Note the difference between training set and off-training set (validation set) error on both tasks !

Note also that on second task validation set error continues to decrease after an initial increase — any regularisation (network simplification, or weight reduction) strategies need to avoid early stopping (underfitting).

Overfitting in ANNs



Underfitting in ANNs



Neural Networks for Classification

Sigmoid unit computes output $o(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x})$

Output ranges from 0 to 1

Example: binary classification

$$o(\mathbf{x}) = \begin{cases} \text{predict class } 1 & \text{if } o(\mathbf{x}) \geq 0.5 \\ \text{predict class } 0 & \text{otherwise.} \end{cases}$$

Questions:

- what error (loss) function should be used ?
- how can we train such a classifier ?

Neural Networks for Classification

Minimizing square error (as before) does not work so well for classification

If we take the output $o(\mathbf{x})$ as the *probability* of the class of **x** being 1, the preferred loss function is the *cross-entropy*

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

where:
$t_d \in \{0, 1\}$ is the class label for training example d, and o_d is the output of the sigmoid unit, interpreted as the probability of the class of training example d being 1.

To train sigmoid units for classification using this setup, one can use *gradient descent* and backpropagation algorithm – this will yield the *maximum likelihood* solution.

Application: Face Pose Recognition

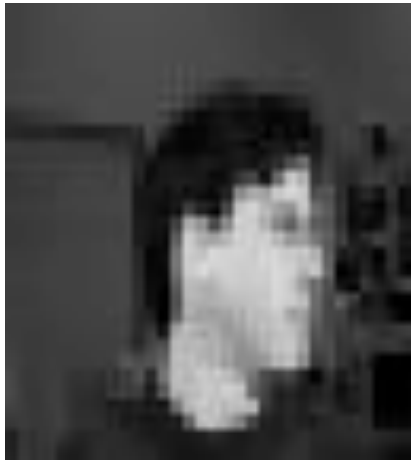
Dataset: 624 images of faces of 20 different people

- image size 120x128 pixels
- grey-scale, 0-255 intensity value range
- different poses
- different expressions
- wearing sunglasses or not

Raw images compressed to 30x32 pixels

MLP structure: 960 inputs $\times$ 3 hidden nodes $\times$ 4 output nodes

Application: Face Pose Recognition


left


straight

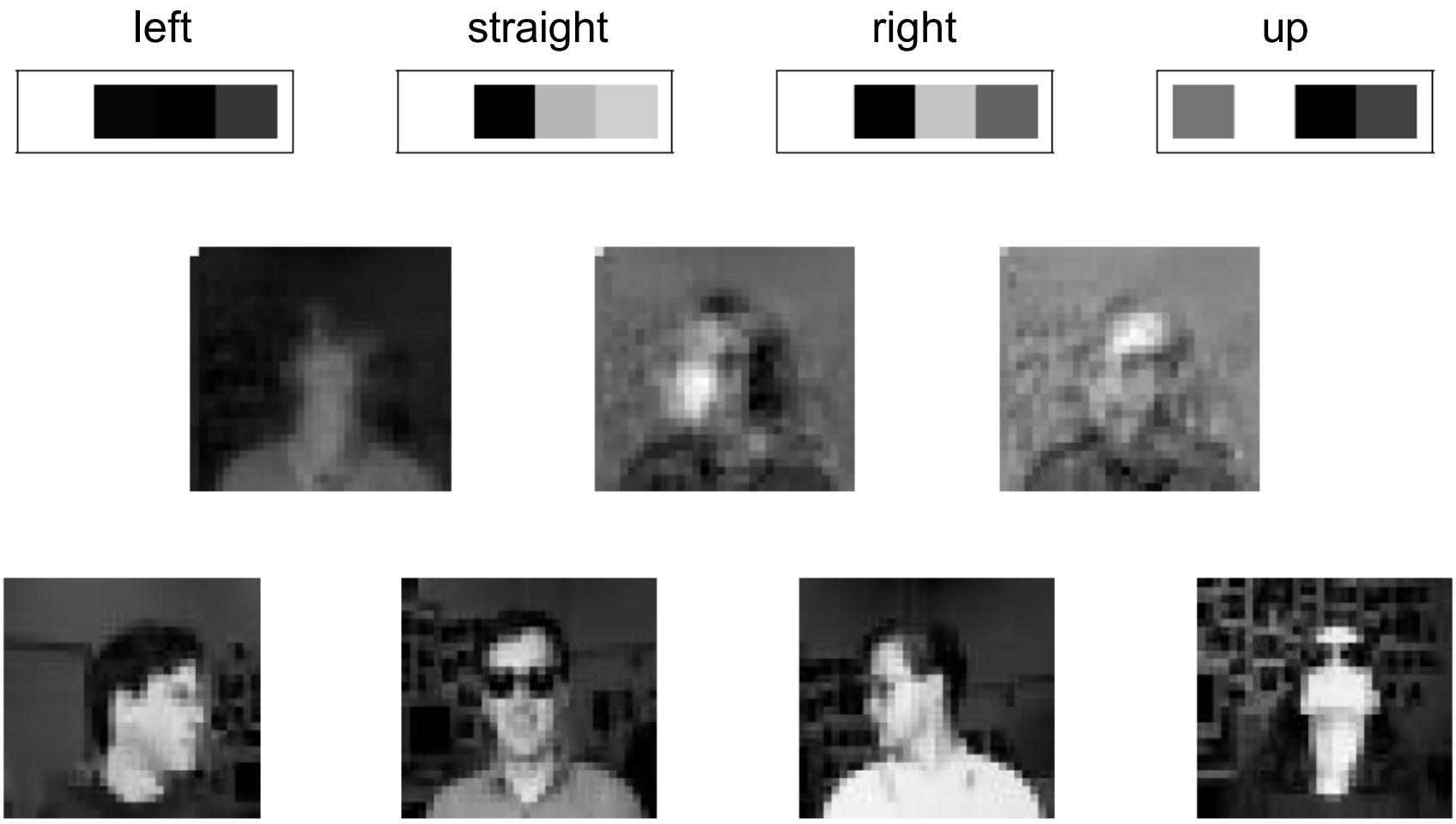

right


up

Four pose classes: looking left, straight ahead, right or upwards
Use a 1-of-*n* encoding: more parameters; can give confidence of prediction
Selected single hidden layer with 3 nodes by experimentation

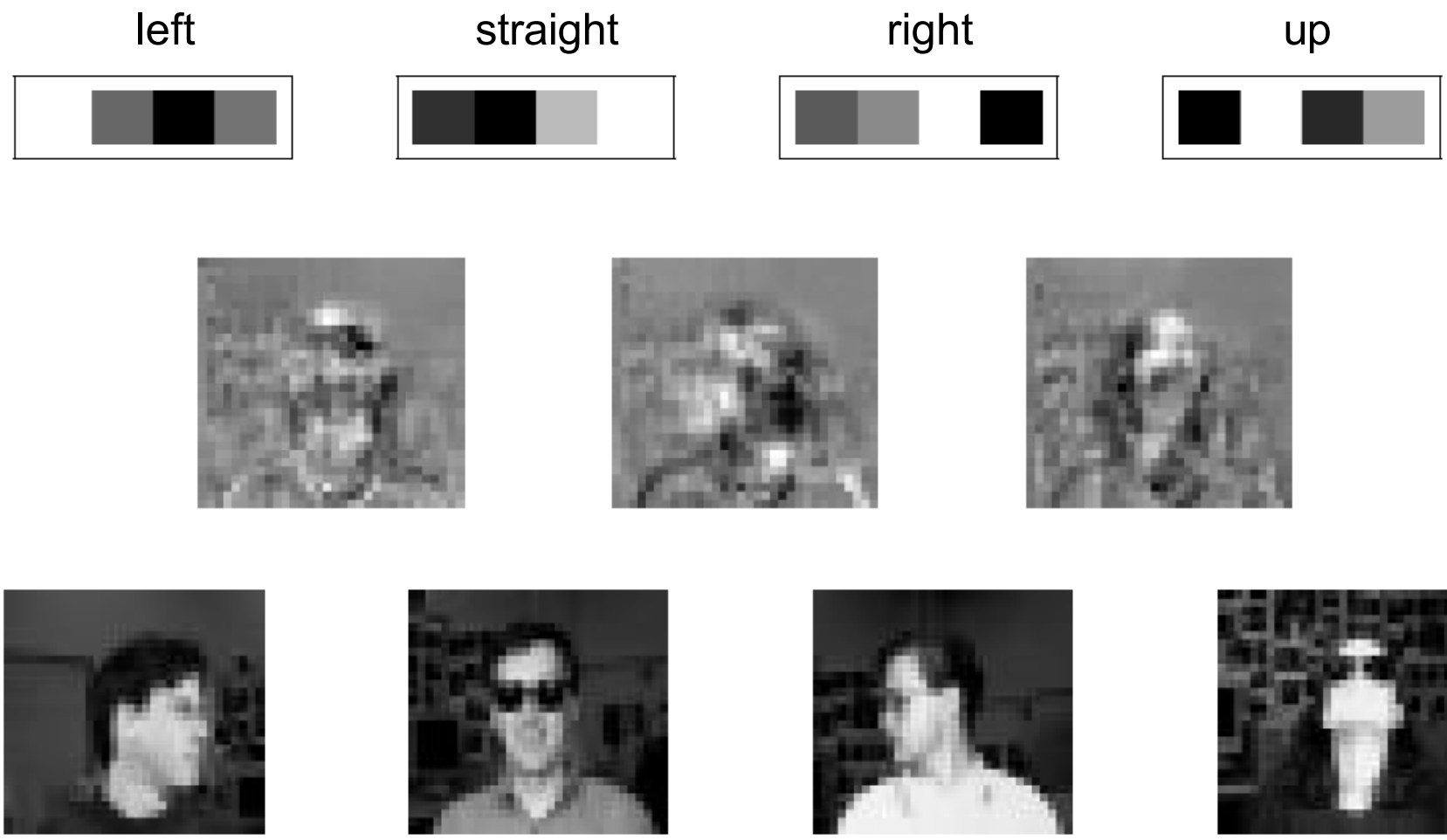
Application: Face Pose Recognition

After 1 epoch



Application: Face Pose Recognition

After 100 epochs



Application: Face Recognition

Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks.

Leftmost block corresponds to the bias (threshold) weight

Weights from each of 30x32 image pixels into each hidden unit are plotted in position of corresponding image pixel.

Classification accuracy: 90% on test set (default: 25%)

Question: what has the network learned ?

For code, data, etc. see http://www.cs.cmu.edu/~tom/faces.html

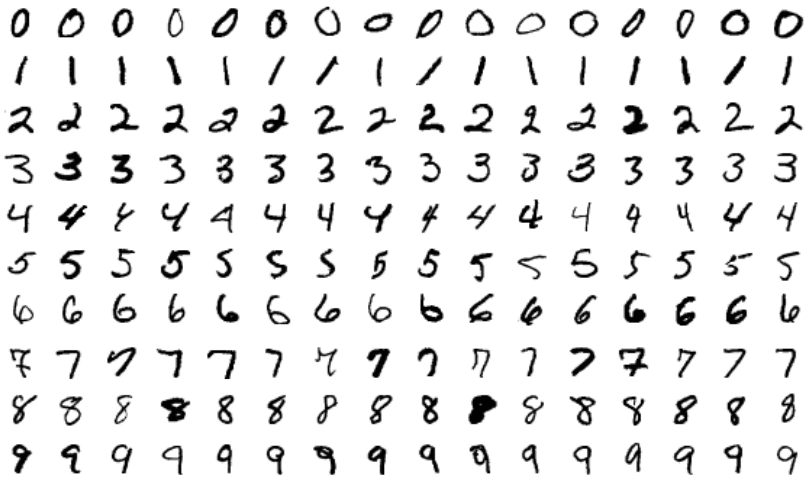
Deep Learning:
Convolutional Neural Networks

A Bit of History

- Earliest studies about visual mechanics of animals emphasised the importance of edge detection for solving Computer Vision problems
  - Early processing in cat visual cortex looks like it is performing convolutions that are looking for oriented edges and blobs
  - Certain cells are looking for edges with a particular orientation at a particular spatial location in the visual field
  - This inspired convolutional neural networks but was limited by the lack of computational power
- Hinton et al. reinvigorated research into deep learning and proposed a greedy layer wise training technique
  - In 2012, Alex Krizhevesky et al. won ImageNet challenge and proposed the well recognised AlexNet Convolutional Neural Network

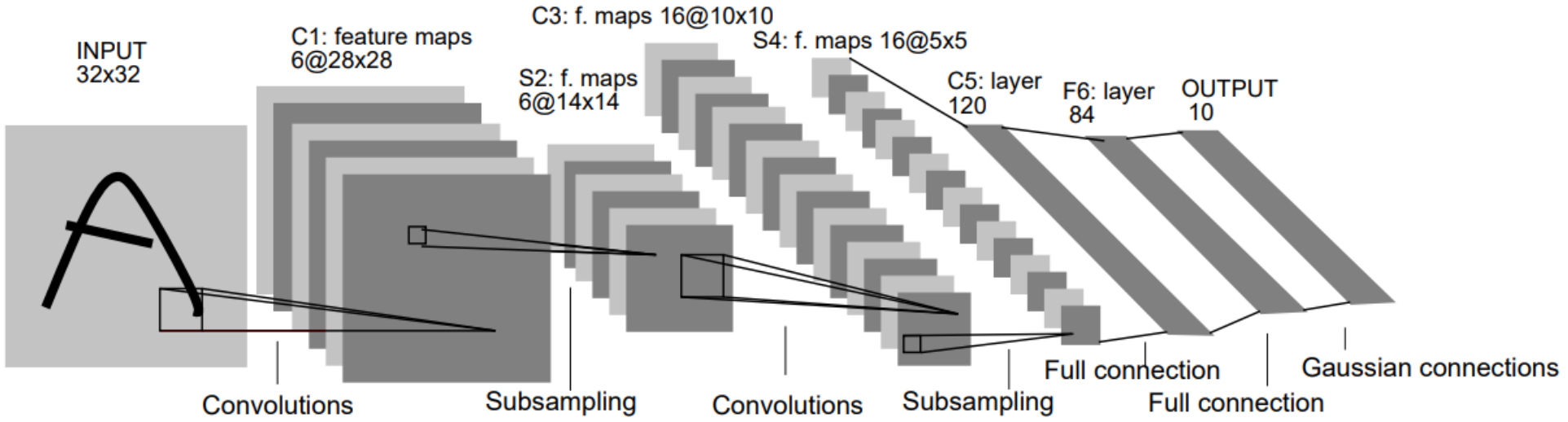
A Bit of History

- LeNet-5
  - The very first CNN
  - Handwritten digit recognition



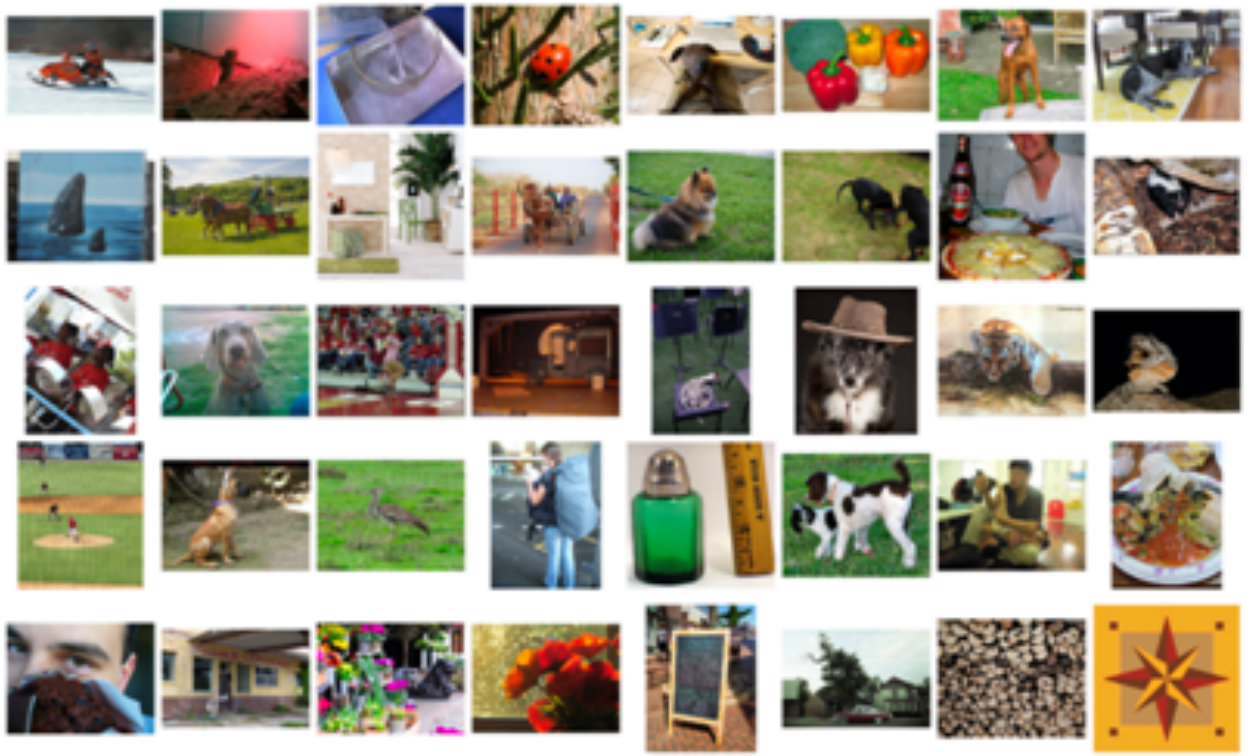
A Bit of History

- LeNet-5
  - The very first CNN
  - Handwritten digit recognition



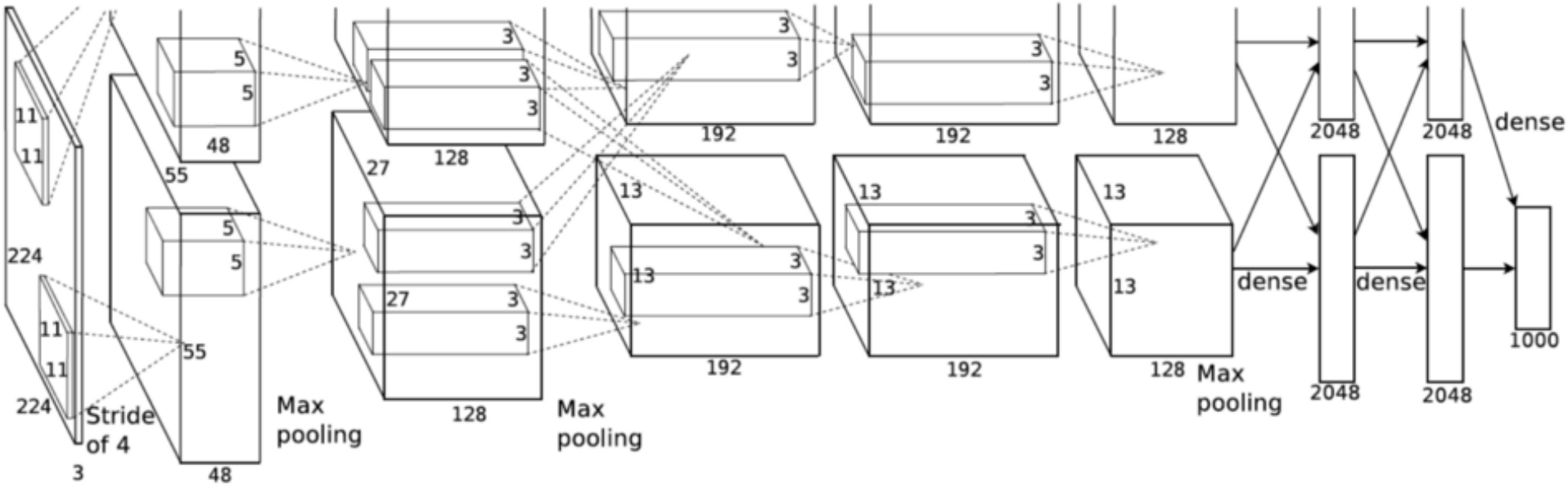
A Bit of History

- AlexNet (2012)
  - ImageNet classification
  - Images showing 1000 object categories



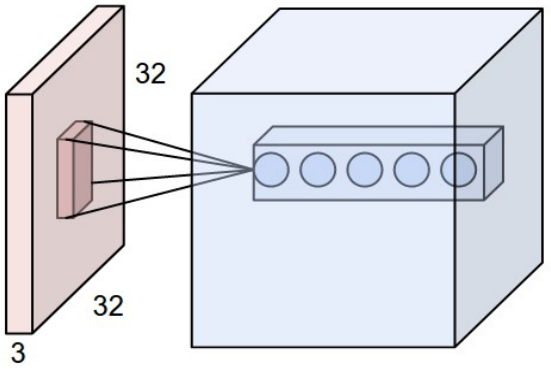
A Bit of History

- AlexNet (2012)
  - ImageNet classification
  - Images showing 1000 object categories



Deep Learning

- Deep learning is a collection of artificial neural network techniques that are widely used at present
- Predominantly, deep learning techniques rely on large amounts of data and deeper learning architectures
- Some well-known paradigms:
  - **Convolutional Neural Networks (CNNs)**
  - Recurrent Neural Networks
  - Auto-encoders
  - Restricted Boltzmann Machines



CNNs

- CNNs are very similar to regular Neural Networks
  - Made up of neurons with learnable weights
- CNN architecture assumes that inputs are images
  - So that we have local features
  - Which allows us to
    - encode certain properties in the architecture that makes the forward pass more efficient and
    - significantly reduces the number of parameters needed for the network

Why CNNs?

The problem with regular NNs is that they do not scale well with dimensions (i.e. larger images)

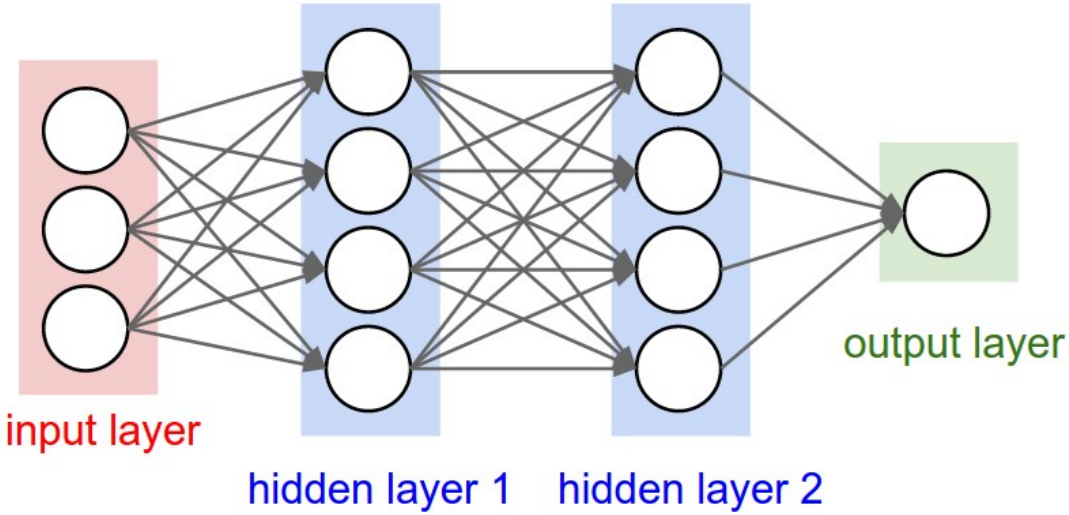
- Eg: 32x32 image with 3 channels (RGB) – a neuron in first hidden layer would have 32x32x3 = 3,072 weights : manageable.
- Eg: 200x200 image with 3 channels – a neuron in first hidden layer would have 200x200x3 = 120,000 weights and we need at least several of these neurons which makes the weights explode.

What is different?

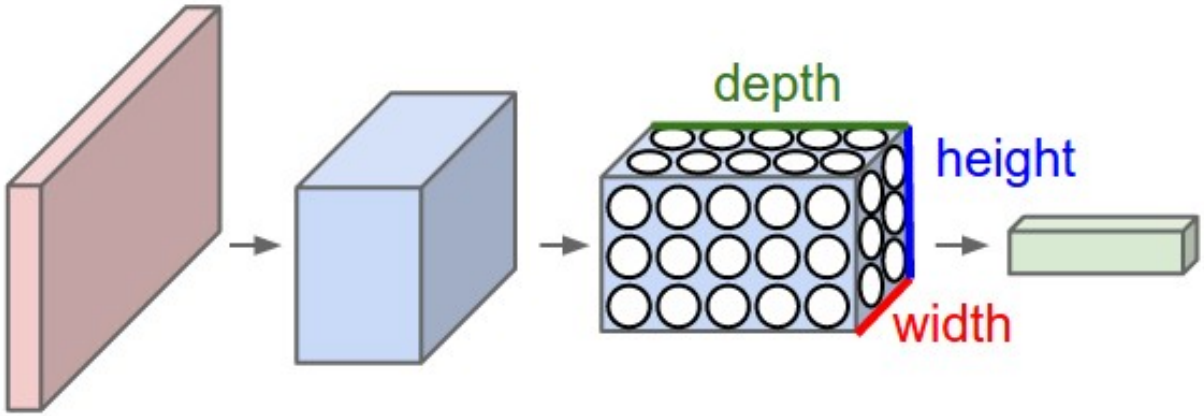
- In contrast, CNNs consider 3-D volumes of neurons and propose a parameter sharing scheme that minimises the number of parameters required by the network.
- CNN neurons are arranged in 3 dimensions: Width, Height and Depth.
- Neurons in a layer are only connected to a small region of the layer before it (hence not fully connected)

What is different?

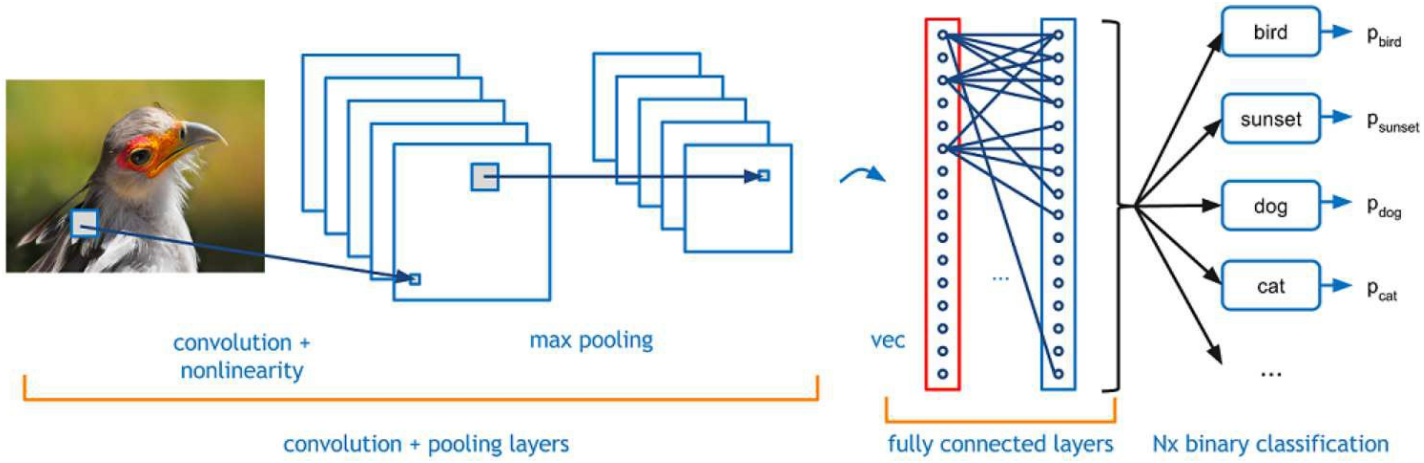
NN



CNN



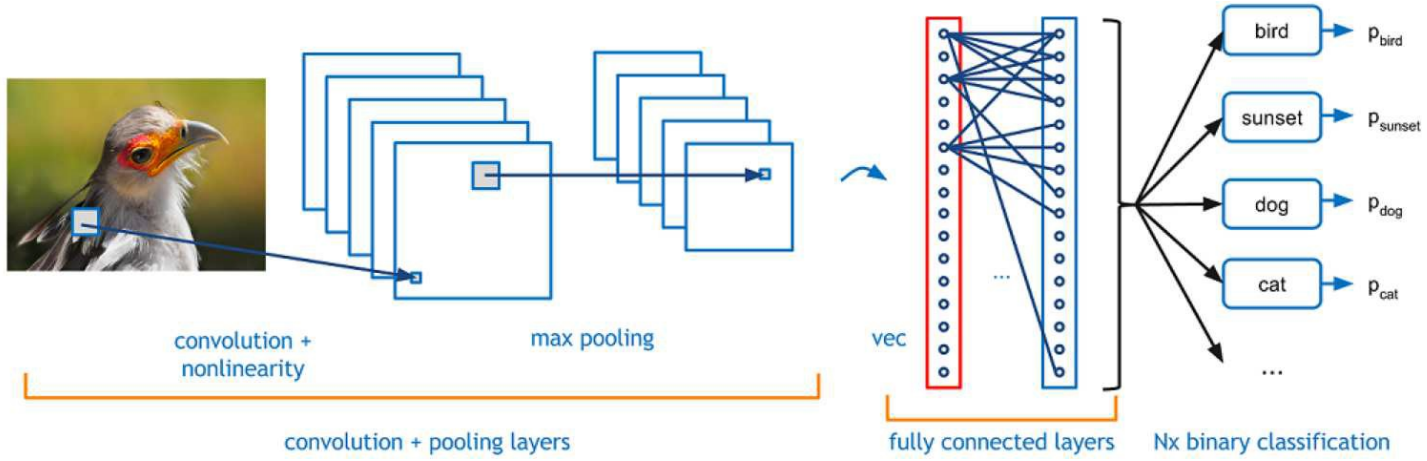
CNN architecture



Main layers:

- Convolutional
- Pooling
- ReLU
- Fully-connected
- Drop-out
- Output layers

Convolutional Layer

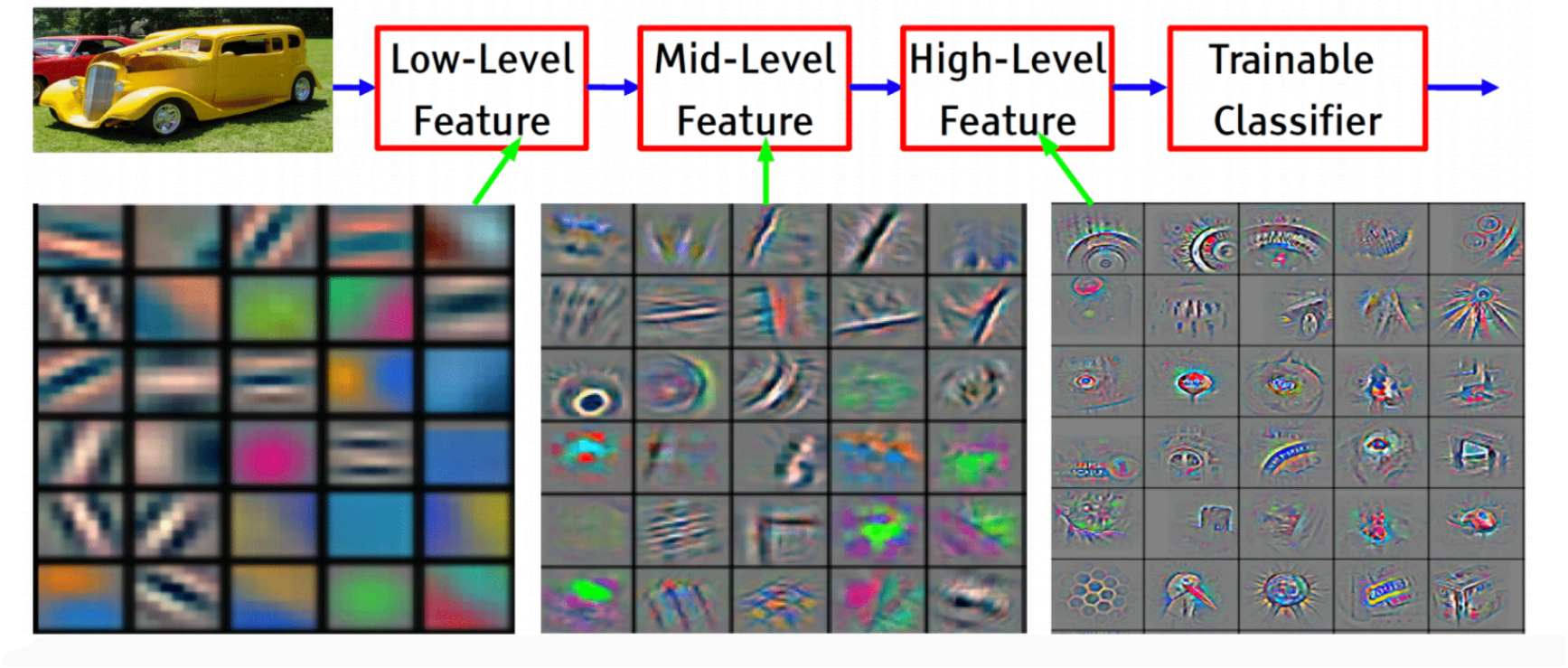


Suppose we want to classify an image as a bird, sunset, dog, cat, etc.

If we can identify features such as feather, eye, or beak which provide useful information in one part of the image, then those features are likely to also be relevant in another part of the image.

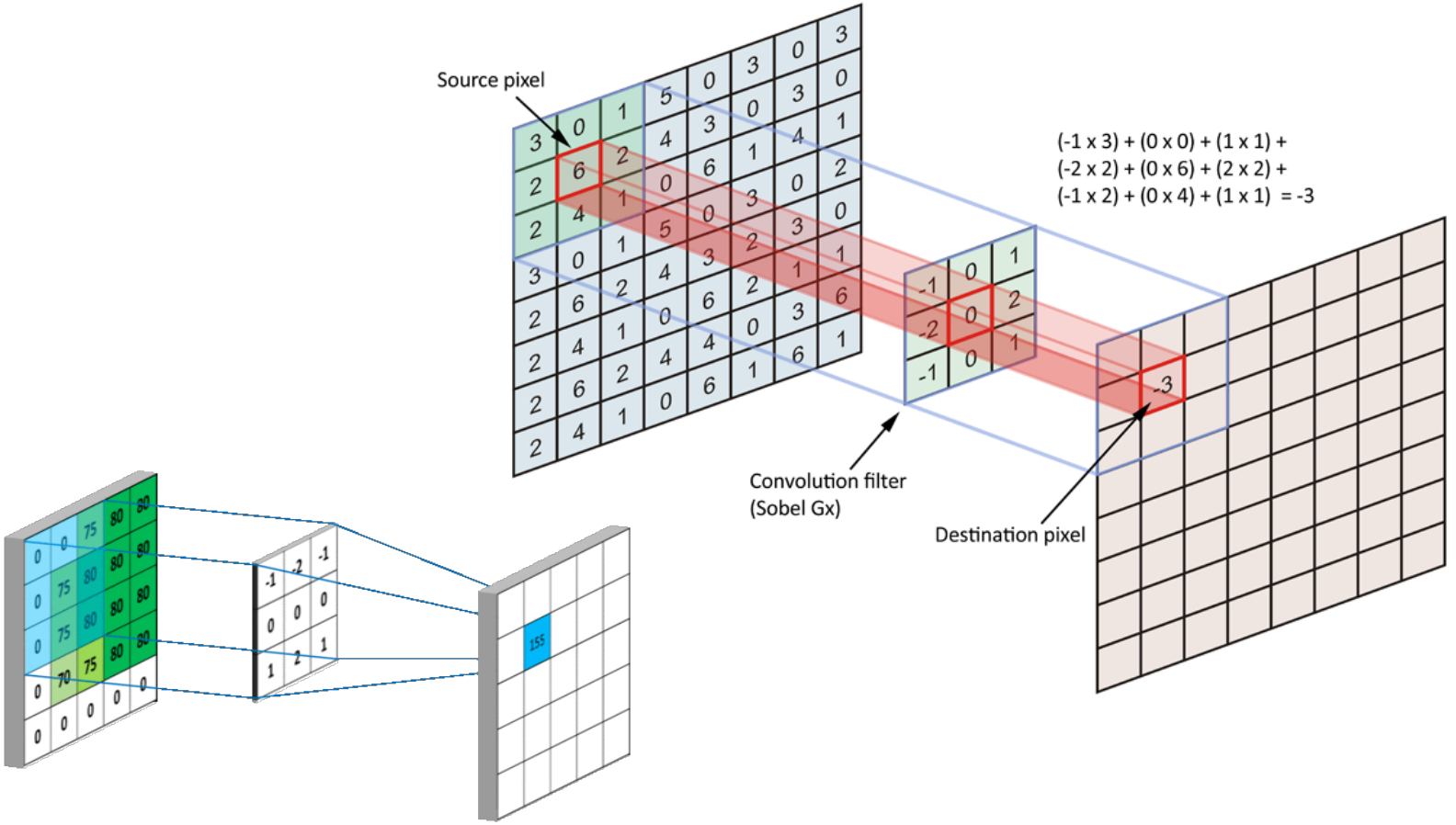
We can exploit this regularity by using a convolution layer which applies the same weights to different parts of the image.

Convolutional Layer

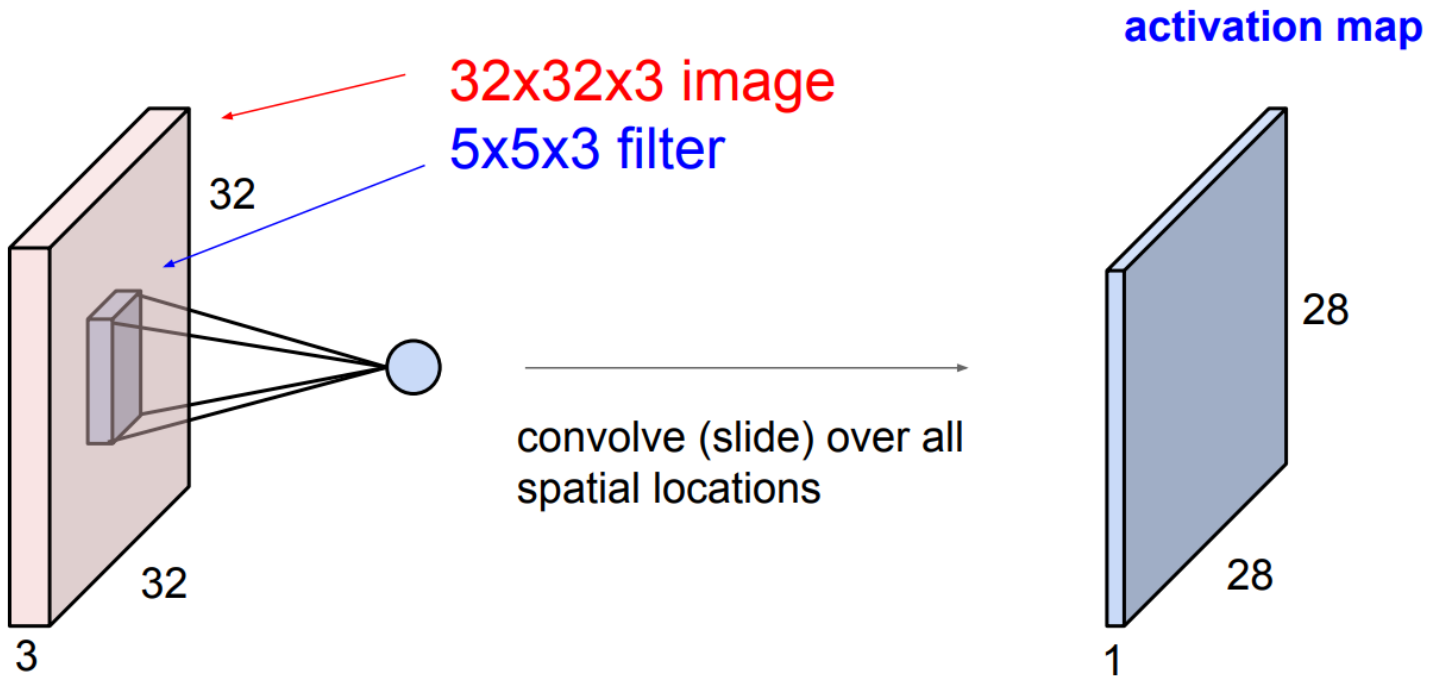


Convolutional Layer

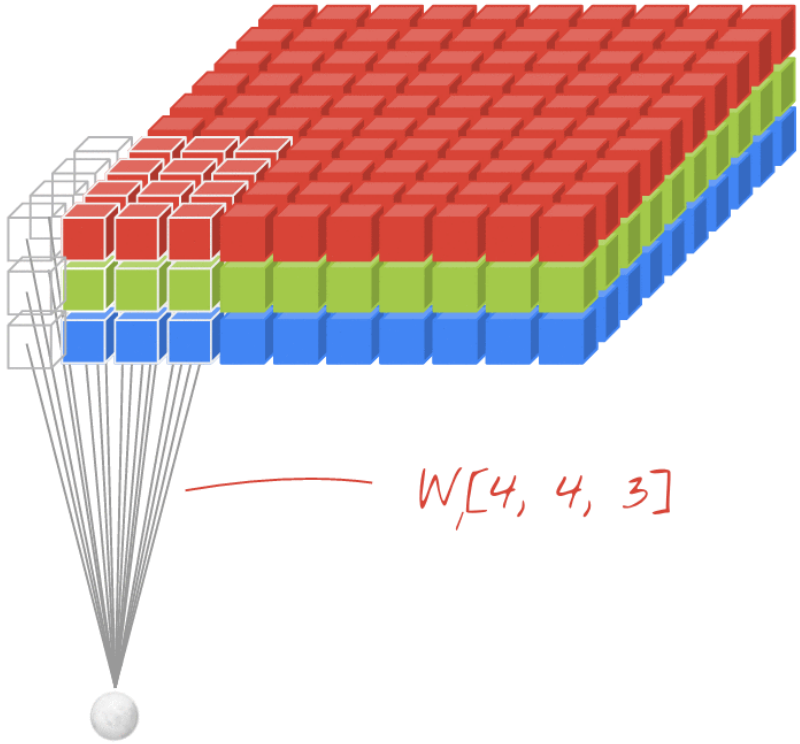
Convolution



Convolutional Layer



Convolutional Layer



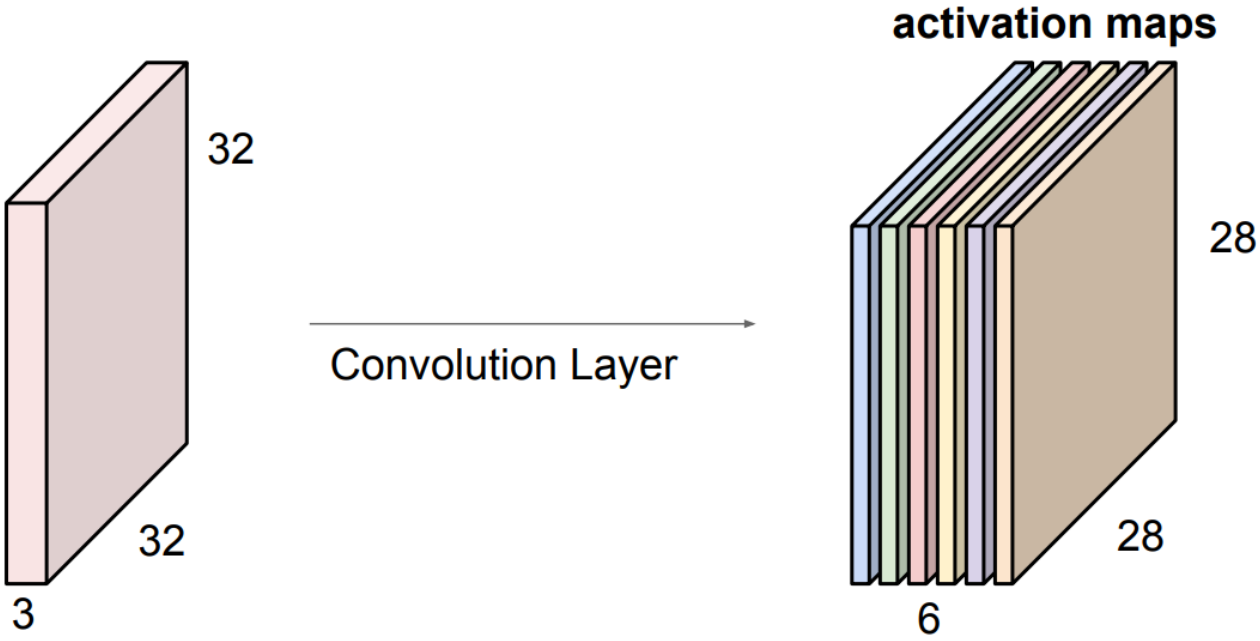
Original link: https://i.stack.imgur.com/nOLCe.gif

Convolutional Layer

- The output of the Conv layer can be interpreted as holding neurons arranged in a 3D volume.
- The Conv layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume.
- During the forward pass, each filter is slid (convolved) across the width and height of the input volume, producing a 2-dimensional activation map of that filter.
- Network will learn filters (via backpropagation) that activate when they see some specific type of feature at some spatial position in the input.

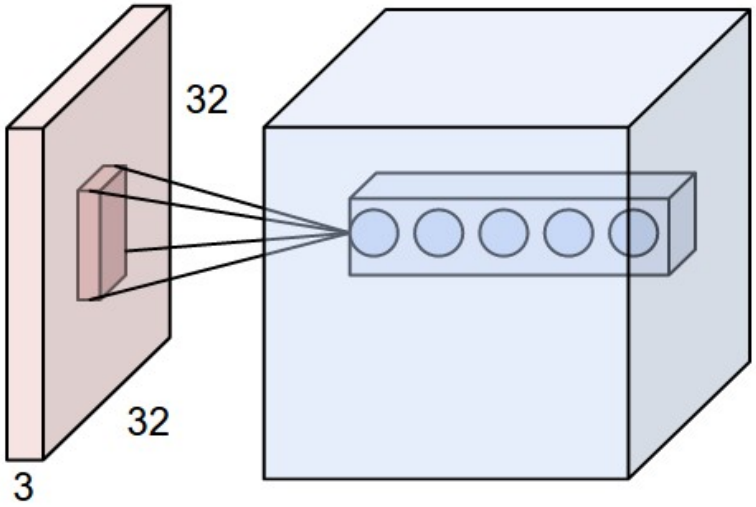
Convolutional Layer

- Stacking these activation maps for all filters along the depth dimension forms the full output volume
- E.g., with 6 filters, we get 6 activation maps



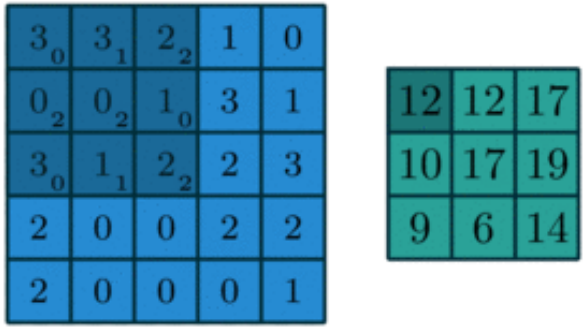
Convolutional Layer

- Three hyperparameters control the size of the output volume: the ***depth, stride*** and ***zero-padding***
  - ***Depth*** controls the number of neurons in the Conv layer that connect to the same region of the input volume



Convolutional Layer

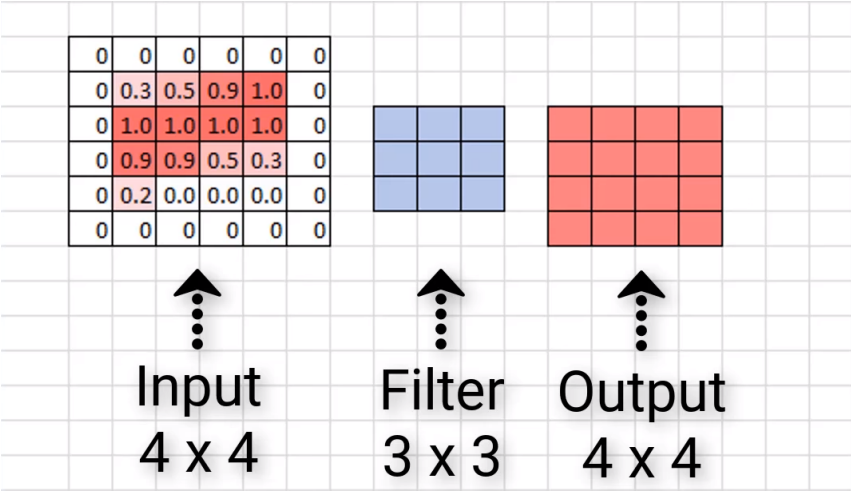
- Three hyperparameters control the size of the output volume: the ***depth, stride*** and ***zero-padding***
  - ***Stride*** is the distance that the filter is moved by in spatial dimensions



http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html

Convolutional Layer

- Three hyperparameters control the size of the output volume: the ***depth, stride*** and ***zero-padding***
  - ***Zero-padding*** is padding of the input with zeros spatially on the border of the input volume



https://deeplizard.com/learn/video/qSTv_m-KFk0

Convolutional Layer

- We can compute the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border:

$$(W-F+2P)/S+1$$

- If this number is not an integer, then the strides are set incorrectly and the neurons cannot be tiled so that they "fit" across the input volume neatly, in a symmetric way.

Example: AlexNet

For example, in the first convolutional layer of AlexNet,
$W = 227$, $F = 11$, $P = 0$, $S = 4$.

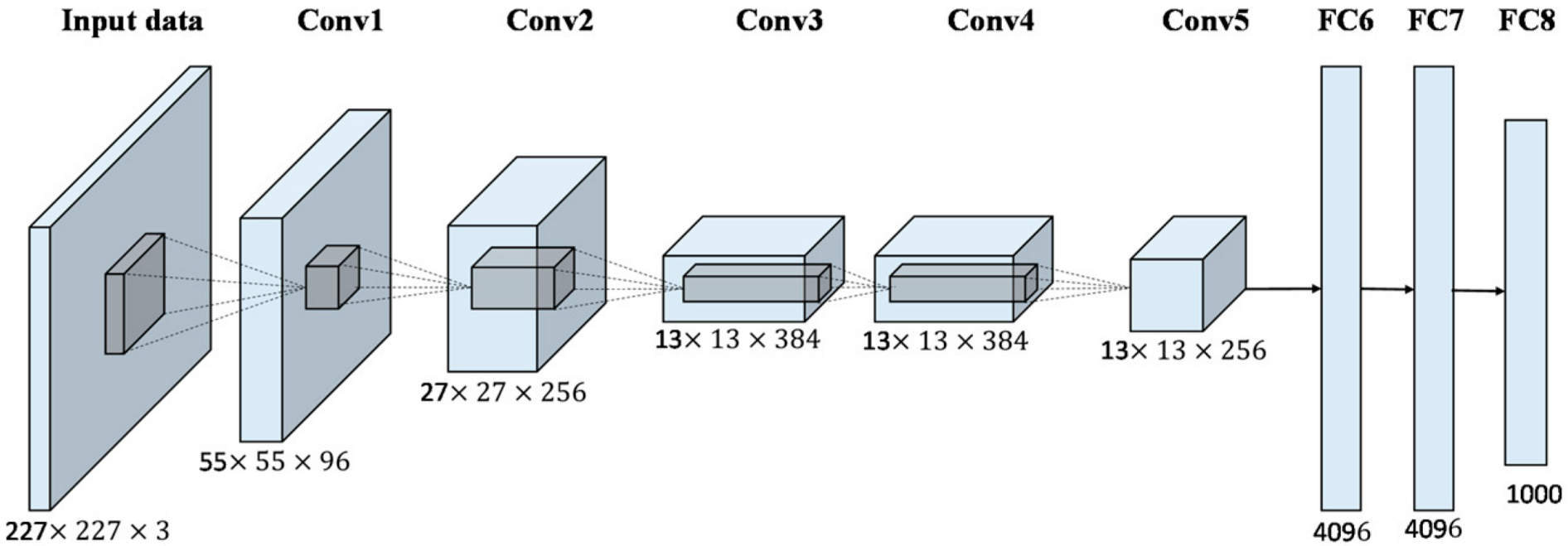
The width of the output is

$$(W-F+2P)/S+1 = (227-11+0)/4 + 1 = 55$$

There are 96 filters in this layer, the output volume of this layer is thus

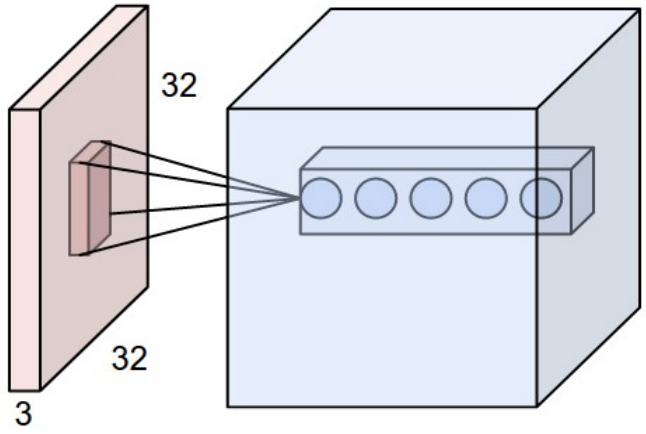
$$55 \times 55 \times 96$$

Example: AlexNet

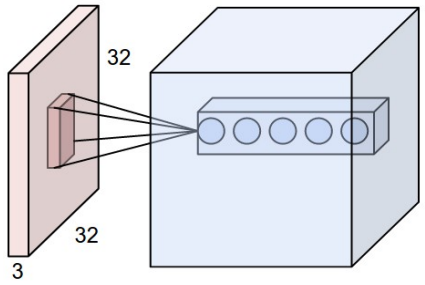


Convolutional Layer

- Main property – ***local connectivity***:
  - Each neuron only connects to a local region of the input volume.
  - The spatial extent of this connectivity is a hyperparameter called ***receptive field*** of the neuron.
  - The extent of the connectivity along the depth axis is always equal to the depth of the input volume.



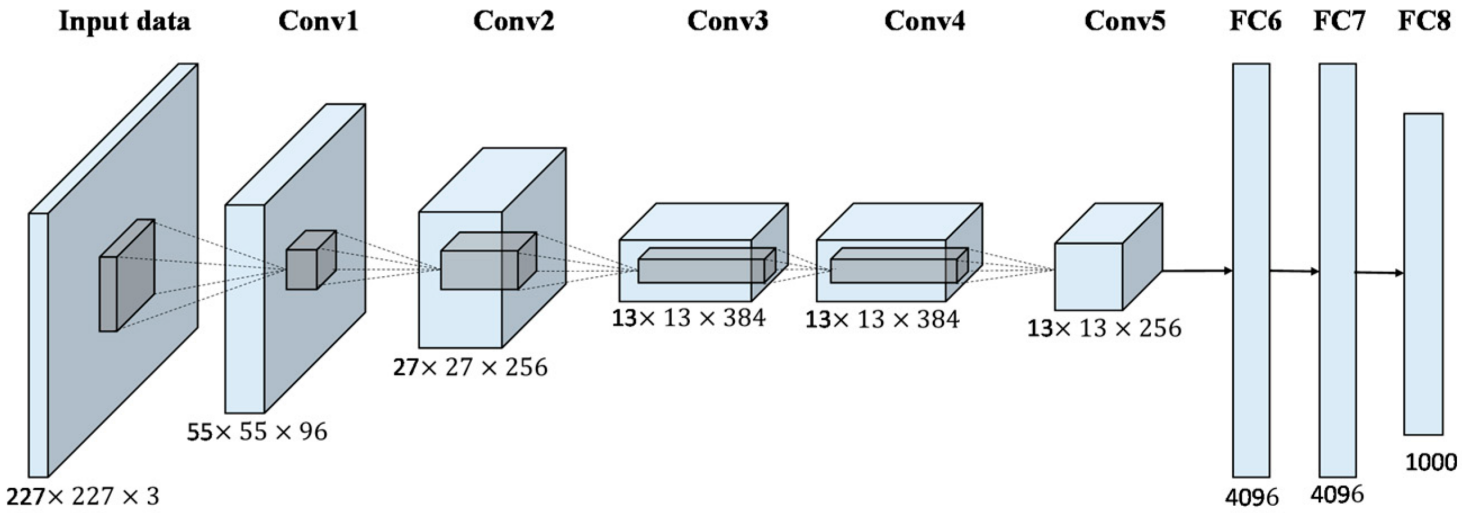
Convolutional Layer



- Examples:
  - Eg1: Suppose that the input volume has size [32x32x3]. If the receptive field is of size 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights. Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.
  - Eg2: Suppose an input volume had size [16x16x20], i.e. . Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

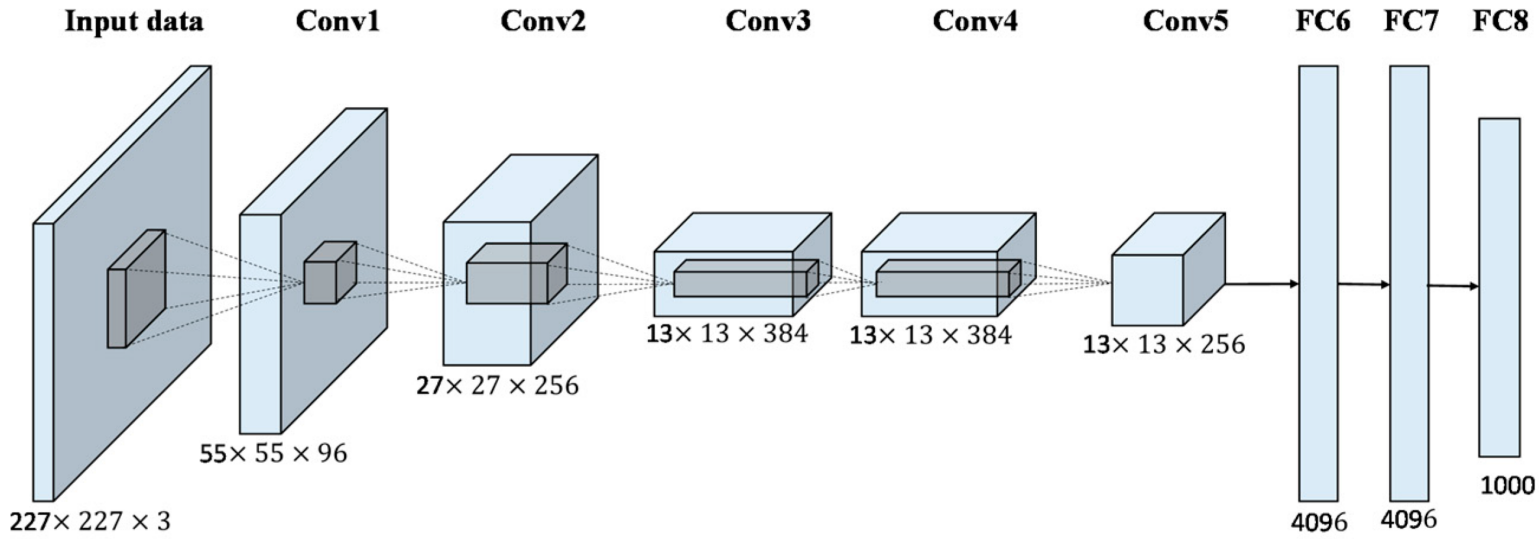
Convolutional Layer

- Main property – ***parameter sharing***:
  - Parameter sharing scheme used in Convolutional Layers to control the number of parameters
    - In other words, denoting a single 2-D slice as a depth slice (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and bias
    - This is exactly what we do with spatial filters for signals/images!



Convolutional Layer

- Main property – ***parameter sharing***:
  - Motivation of parameter sharing
    - If one patch feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2).



Convolutional Layer

- Example:
  - In AlexNet, without parameter sharing, there are 55*55*96 = 290,400 neurons in the first Conv Layer, and each has 11*11*3 = 363 weights and 1 bias.
  - Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the ConvNet alone. Clearly, this number is very high.
  - With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique sets of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases).
  - Alternatively, it can be viewed as all 55*55 neurons in each depth slice will now be using the same parameters.

Example: AlexNet

For example, in the first convolutional layer of AlexNet,
$W = 227$, $F = 11$, $P = 0$, $S = 4$.

The width of the output is

$$(W-F+2P)/S+1 = (227-11+0)/4 + 1 = 55$$

There are 96 filters in this layer. Compute the number of:

weights per neuron?
neurons?
connections?
independent parameters?

Example: AlexNet

For example, in the first convolutional layer of AlexNet,
$W = 227$, $F = 11$, $P = 0$, $S = 4$.

The width of the output is

$$(W-F+2P)/S+1 = (227-11+0)/4 + 1 = 55$$

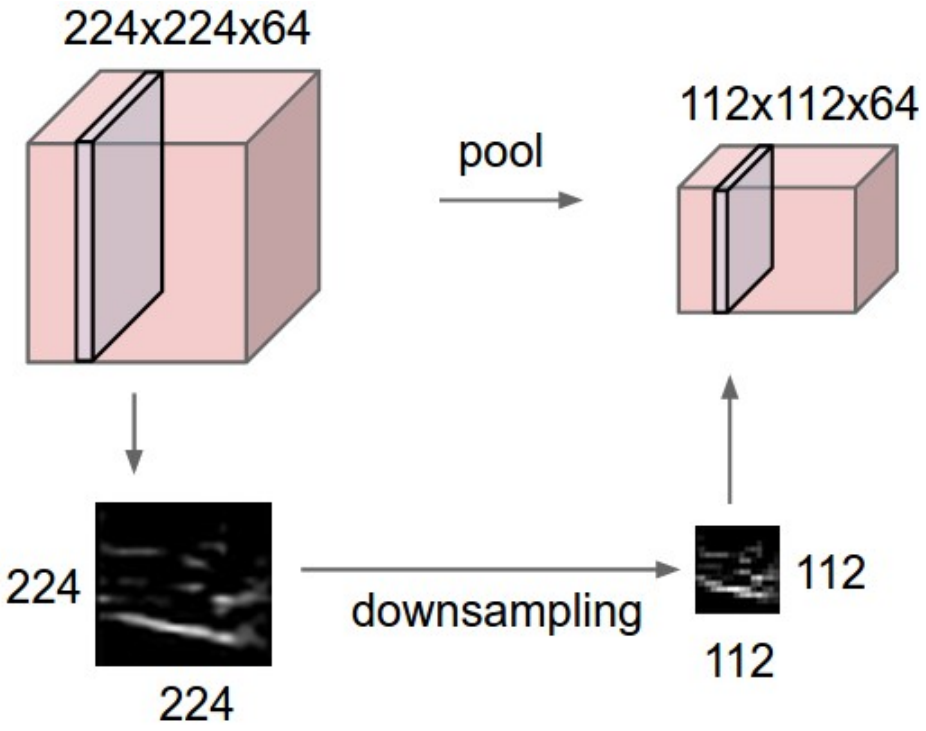
There are 96 filters in this layer. Compute the number of:

weights per neuron?	$1 + 11 \times 11 \times 3$	=	364
neurons?	$55 \times 55 \times 96$	=	290,400
connections?	$55 \times 55 \times 96 \times 364$	=	105,705,600
independent parameters?	96×364	=	34,944

Pooling Layer

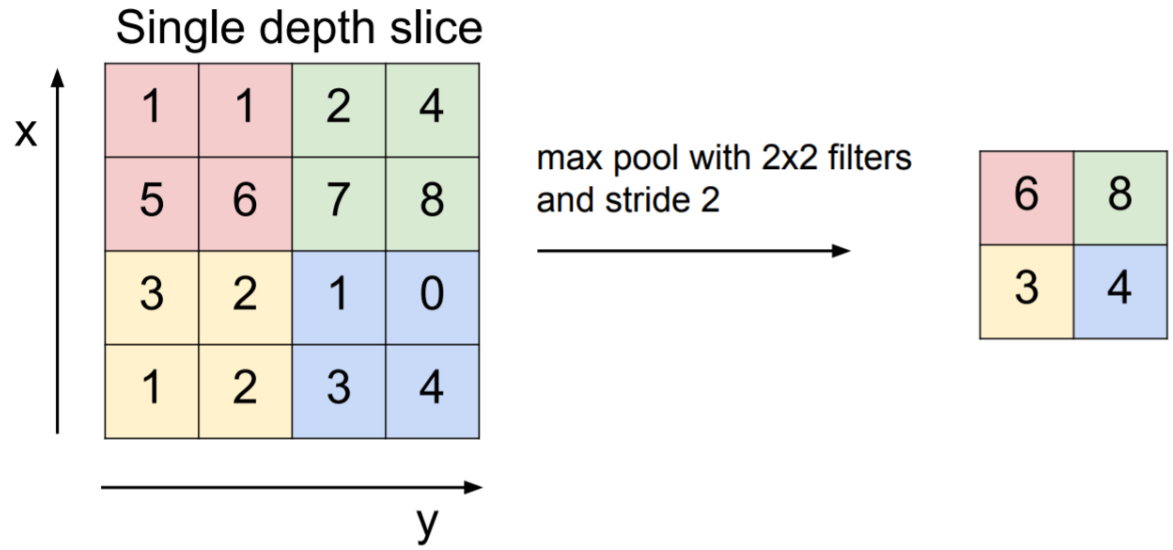
- The function of pooling layer
  - to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and
  - hence to also control overfitting
- The Pooling Layer operates
  - independently on every depth slice of the input and resizes it spatially, typically using the MAX operation (i.e.: max pooling)
  - The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations

Pooling Layer



Pooling Layer

Max pooling



Pooling Layer

- If the previous layer is $J \times K$, and max pooling is applied with width F and stride S, the size of the output will be

$$(1 + (J - F)/s) \times (1 + (K - F)/s)$$

- If max pooling with width 3 and stride 2 is applied to the feature map of size 55 ×55 in the first convolutional layer of AlexNet, what is the output size after pooling?

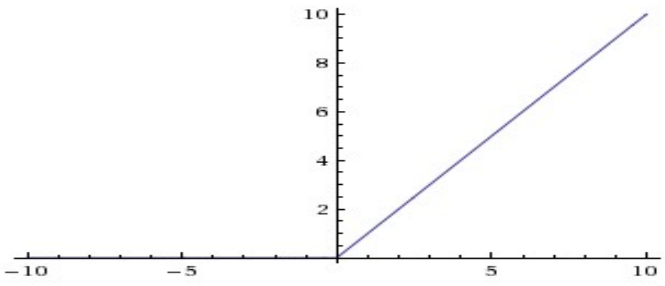
Answer: $1 + (55 - 3)/2 = 27$.

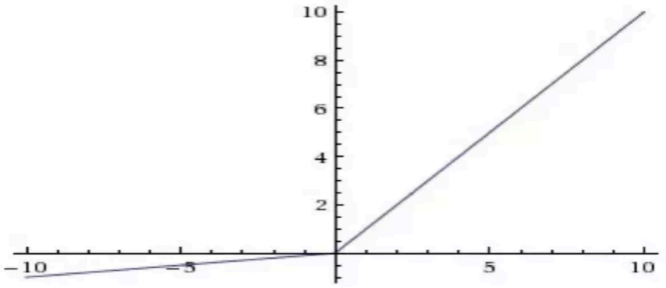
- How many independent parameters does this add to the model?

  Answer: None! (no weights to be learned, just computing max)

ReLU Layer

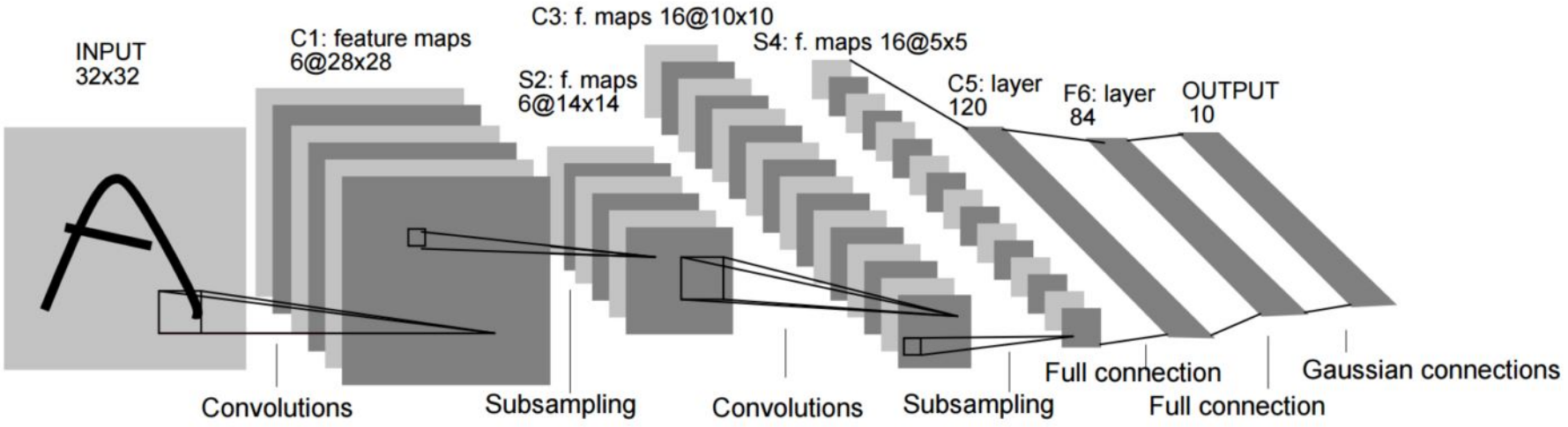
- Although ReLU (**Re**ctified **L**inear **U**nit) is considered as a layer, it is really an activation function:
- $$f(x) = max(0, x)$$
- This is favoured in deep learning as opposed to the traditional activation functions like Sigmoid or Tanh
  - To accelerate the convergence of stochastic gradient descent
  - Be computationally inexpensive compared to traditional ones
- However, ReLu units can be fragile during training and 'die'. Leaky ReLUs were proposed to handle this problem.





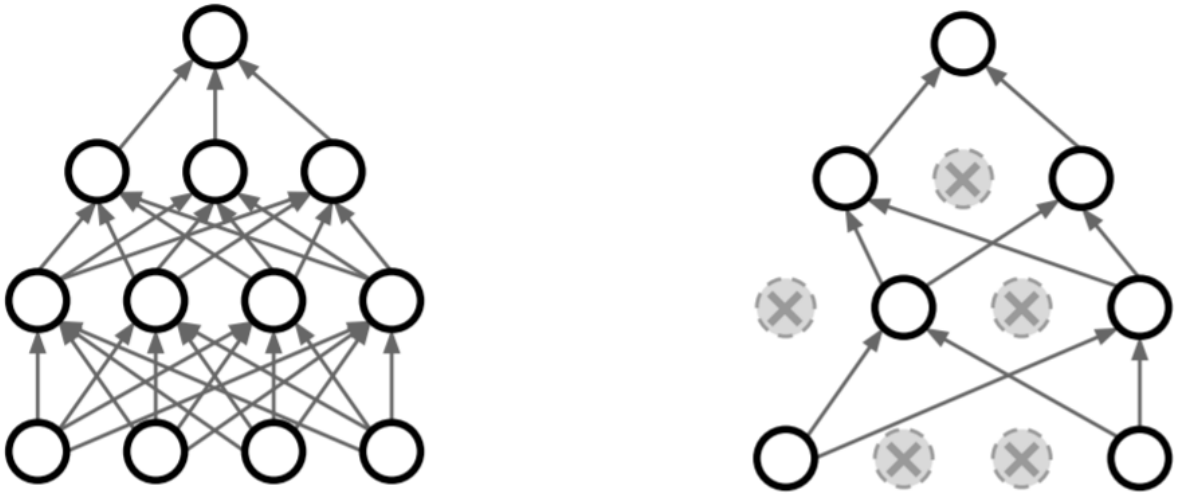
Fully-connected Layer

- Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.



Dropout Layer

- Problem with overfitting – model performs well on training data but generalises poorly to testing data
- Dropout is a simple and effective method to reduce overfitting
- In each forward pass, randomly set some neurons to zero
- Probability of dropping is a hyperparameter, such as 0.5

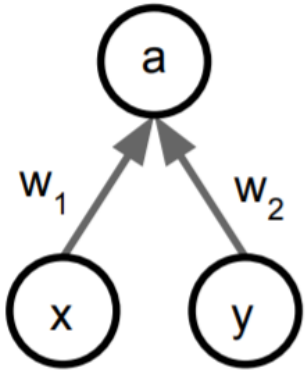


Dropout Layer

- Makes the training process noisy
- Forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs
- Prevents co-adaptation of features and simulates a sparse activation
- Analogous to training a large ensemble of models but with much higher efficiency

Dropout Layer

- During test time, direct application would make the output random
- A simple approach: multiply the activation by dropout probability (e.g. 0.5)



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

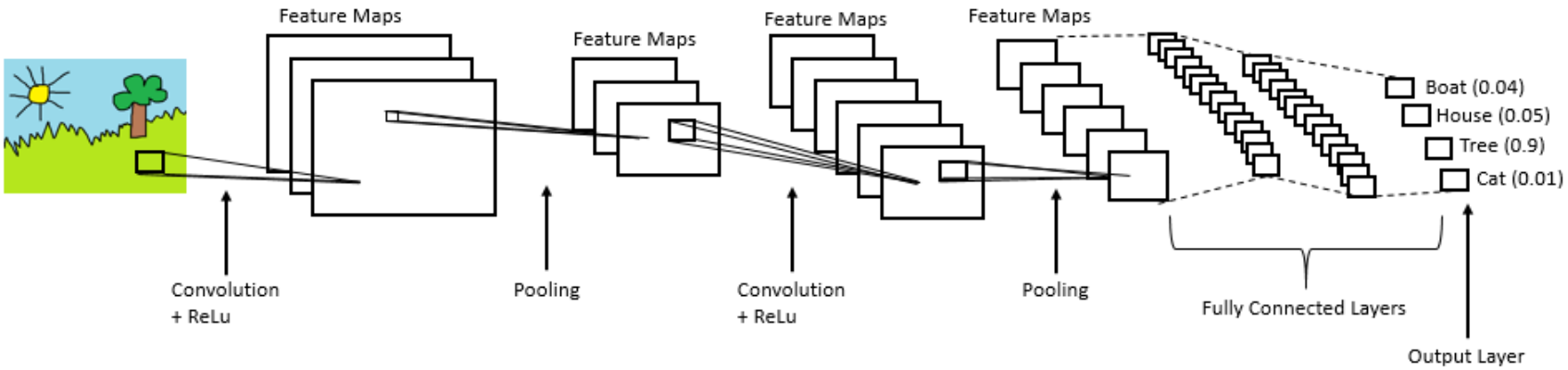
During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Output Layer

- The output layer produces the probability of each class given the input image
- This is the last layer containing the same number of neurons as the number of classes in the dataset
- The output of this layer passes through a Softmax activation function to normalize the outputs to a sum of one:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$



Loss Function

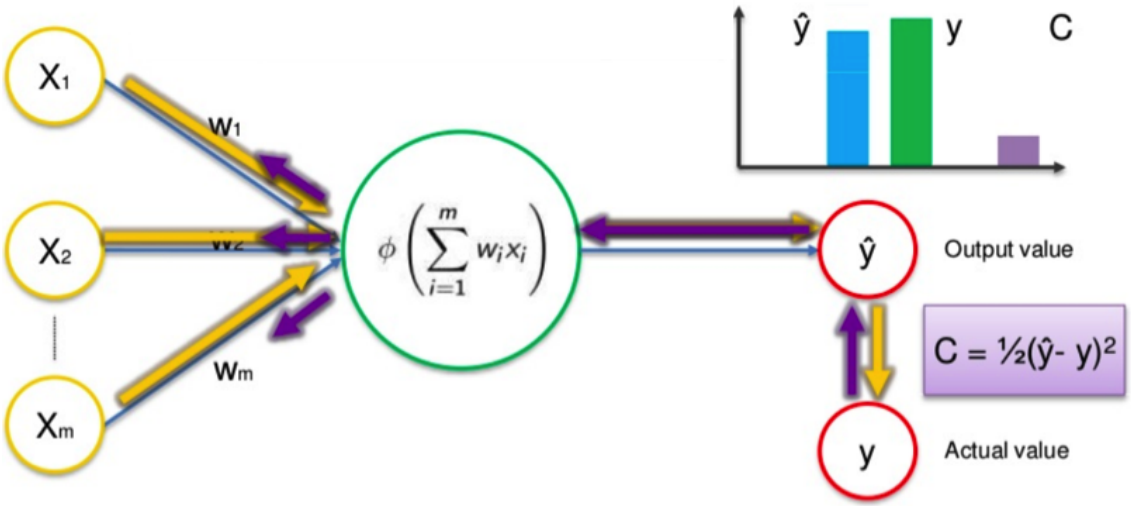
- A loss function is used to compute the model's prediction accuracy from the outputs
  - Most commonly used: categorical cross-entropy loss function

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

- The training objective is to minimise this loss
- The loss guides the backpropagation process to train the CNN model
- Stochastic gradient descent and the Adam optimiser are commonly used algorithms for optimisation

Training

- Backpropagation in general:

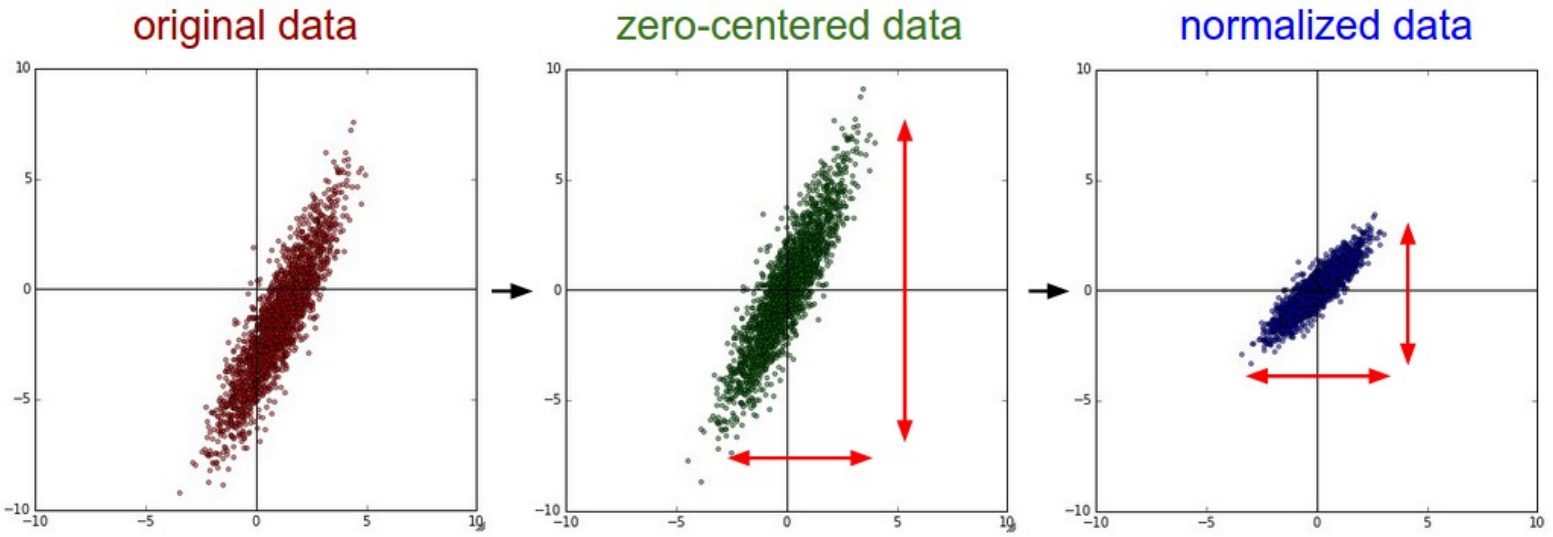


https://www.superdatascience.com/blogs/artificial-neural-networks-backpropagation

1. Initialise the network.
2. Input the first observation.
3. Forward-propagation. From left to right the neurons are activated and the output value is produced.
4. Calculate the error in the outputs (loss function).
5. From right to left the generated error is back-propagated and accumulate the weight updates (partial derivatives).
6. Repeat steps 2-5 and adjust the weights after a batch of observations.
7. When the whole training set passes through the network, that makes an epoch. Redo more epochs.

Pre-processing

Pre-processing: image scaling, zero mean, and normalisation

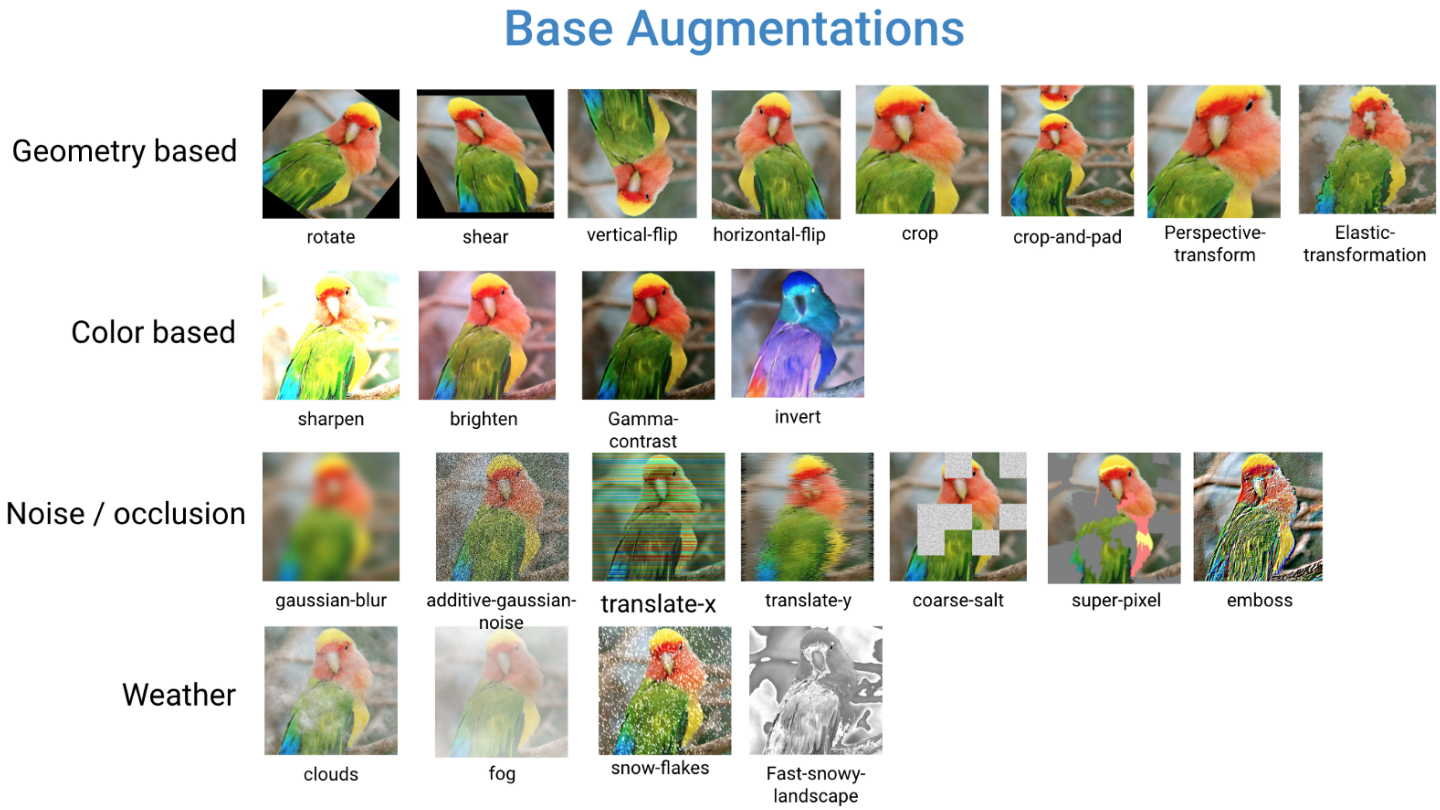


Data Augmentation

- Essential for increasing the dataset size and avoiding over-fitting
- More data augmentation often leads to better performance but also longer training time
- Commonly used techniques include:
  - Horizontal / vertical flipping
  - Random cropping and scaling
  - Rotation
  - Gaussian filtering
- During testing, average the results from multiple augmented input images

Data Augmentation

Need evaluation => not all techniques are useful



https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366

Initialisation

- Weight initialisation
  - Cannot be all 0's => Need to ensure diversity in the filter weights
  - Use small random numbers => might aggravate the diminishing gradients problem
    - With calibration
    - Sparse initialisation
    - More advanced techniques
  - Use ImageNet pretrained models => not always possible

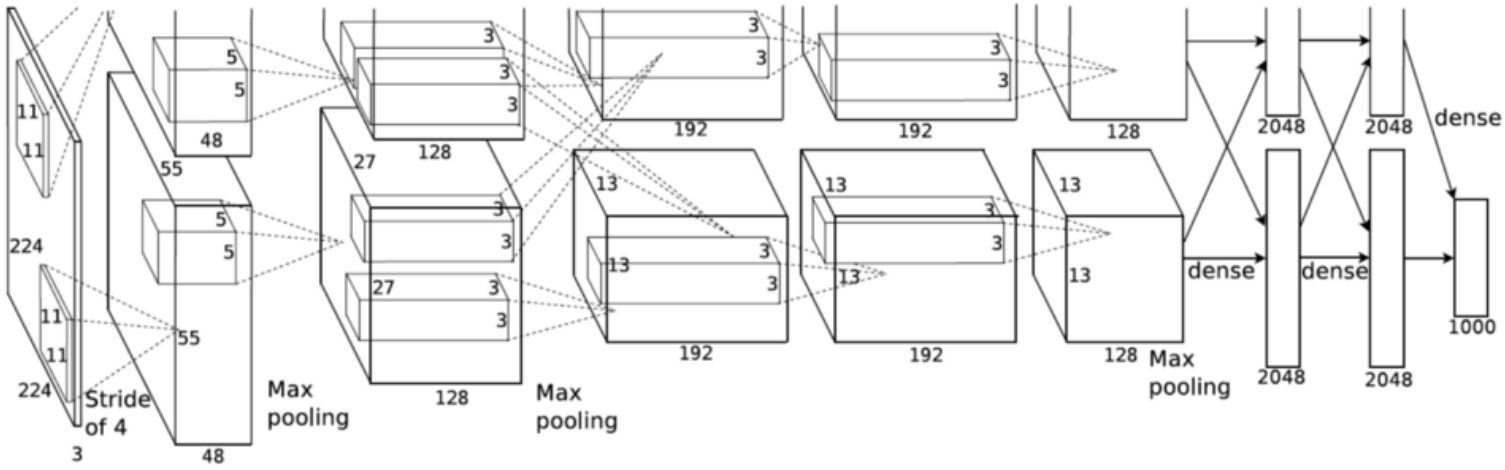
Balancing Data

- Balanced training data
  - Important to have similar numbers of training images for different classes, so the optimisation would not be biased by one class
  - Use random sampling to achieve this effect during each epoch of training
  - Assign different weights in the loss function

$$\alpha_c = median\_freq/freq(c) \qquad H(y, \hat{y}) = \sum_i \propto_i \; y_i \log \hat{y}_i$$

Testing

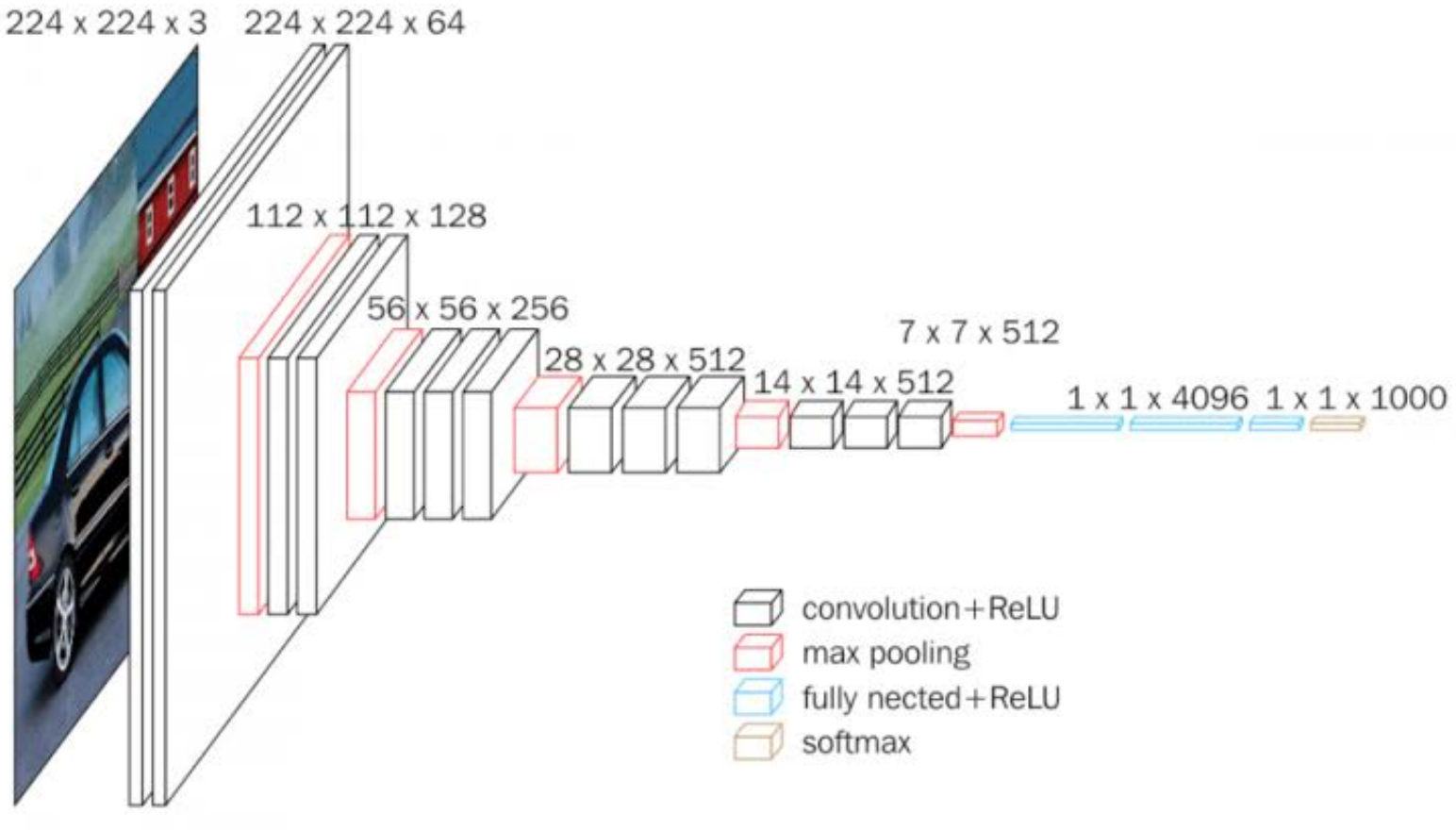
- Forward passes of the network throughout the layers give the prediction output of the input data



Transfer Learning

- CNN models trained on ImageNet can be applied to other types of images
- It is possible to finetune only the last FC layers to better fit the model to the specific set of images
- Especially useful for small datasets

VGGNet


224 x 224 x 3
224 x 224 x 64
112 x 112 x 128
56 x 56 x 256
28 x 28 x 512
14 x 14 x 512
7 x 7 x 512
1 x 1 x 4096
1 x 1 x 1000
convolution+ReLU
max pooling
fully nected+ReLU
softmax

VGGNet

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

Well-known Models

- Object Recognition
  - AlexNet (2012)
  - GoogLeNet
  - VGGNet
  - ResNet
  - Inception v3/v4
  - DenseNets is the current state-of-the-art
- Semantic Segmentation
  - Multi-scale CNN (2012)
  - FCN
  - U-net / V-net
  - U-net / V-net with skip/dense connections
  - Many other variations

Summary

Artificial Neural Networks

- Complex function fitting. Generalise core techniques from machine learning and statistics based on linear models for regression and classification.
- Learning is typically stochastic gradient descent. Networks are too complex to fit otherwise.
- A brief introduction of CNN – the most commonly used model of deep learning
- Widely used in computer vision studies
- In-depth knowledge in COMP9444

Acknowledgement

Material derived from slides for the book
"Elements of Statistical Learning (2nd Ed.)" by T. Hastie,
R. Tibshirani & J. Friedman. Springer (2009)
http://statweb.stanford.edu/~tibs/ElemStatLearn/

Material derived from slides for the book
"Machine Learning: A Probabilistic Perspective" by P. Murphy MIT Press (2012)
http://www.cs.ubc.ca/~murphyk/MLbook

Material derived from slides for the book "Machine Learning" by P. Flach Cambridge University Press (2012) http://cs.bris.ac.uk/~flach/mlbook

Material derived from slides for the book
"Bayesian Reasoning and Machine Learning" by D. Barber Cambridge University Press (2012)
http://www.cs.ucl.ac.uk/staff/d.barber/brml

Material derived from slides for the book "Machine Learning" by T. Mitchell McGraw-Hill (1997)
http://www-2.cs.cmu.edu/~tom/mlbook.html

Material derived from slides for the course "Machine Learning" by A. Srinivasan BITS Pilani, Goa, India (2016)

Slides from CISC 4631/6930 Data Mining https://slideplayer.com/slide/13508539/