7CCSMPNN: Pattern Recognition, Neural Networks, and Deep Learning

Deep Discriminative Neural Networks

Part a

Introduction to, and Motivations for, Deep NNs

This Week

- What is a Deep Neural Network?
- Why Build Deep Neural Networks?
- Difficulties Training Deep Neural Networks
  - solutions
- Convolutional Neural Networks (CNNs)
  - convolutional layers
  - pooling and fully-connected layers
- Issues with CNNs/deep networks

Contents

What is a Deep Neural Network?

A deep network is a neural network with >3 layers

- typically >>3 layers

A shallow network is a neural network with ≤3 layers

- e.g. the types of network described in the preceding 2 weeks

The term deep learning refers to the process of training a deep neural network

(note, above counts linear input layer in number of layers)

Why Build Deep Neural Networks?

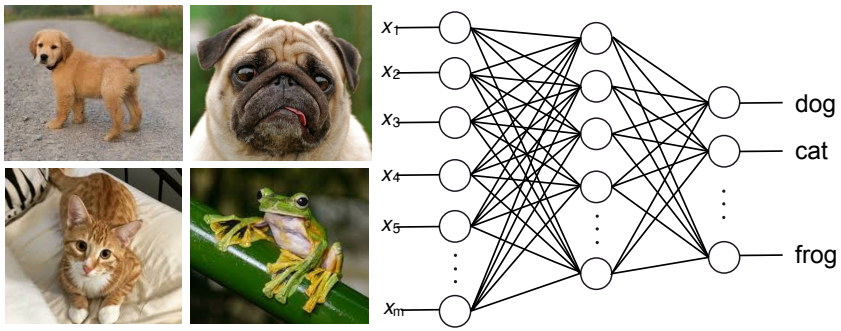
Three-layer Neural Networks able to approximate any continuous non-linear function arbitrarily well.

- Universal function approximators
- Can solve any pattern recognition task, in theory

So what is the point of building deeper networks?

Why Build Deep Neural Networks?

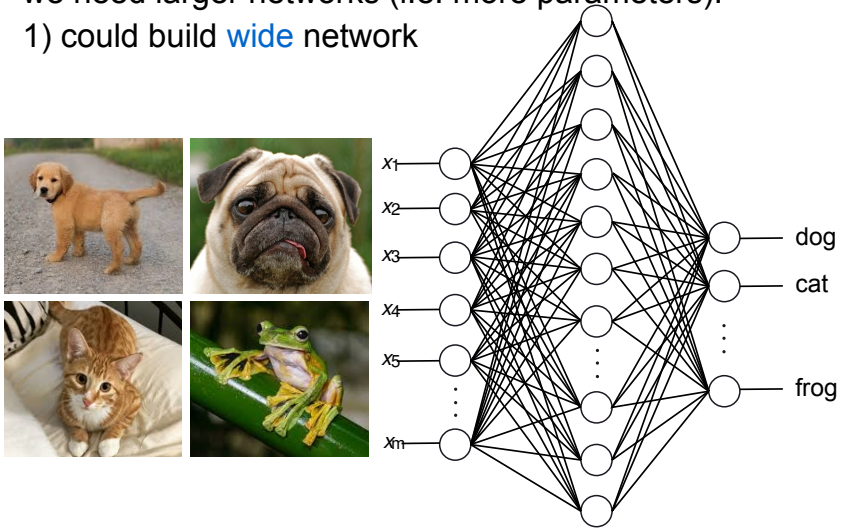
To solve complex tasks (i.e. to perform complex mappings) we need larger networks (i.e. with more parameters).



Why Build Deep Neural Networks?

To solve complex tasks (i.e. to perform complex mappings) we need larger networks (i.e. more parameters).

1) could build wide network

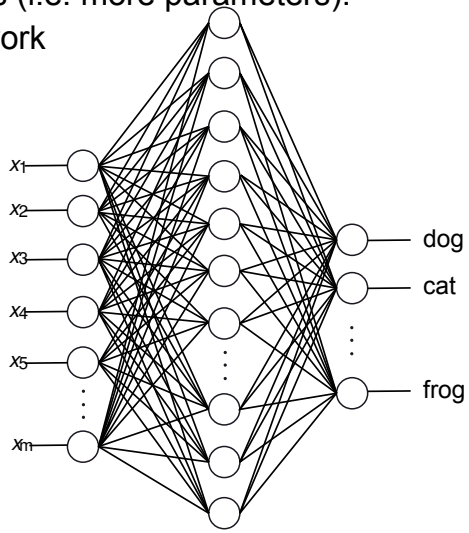


Why Build Deep Neural Networks?

To solve complex tasks (i.e. to perform complex mappings) we need larger networks (i.e. more parameters).

1) could build wide network

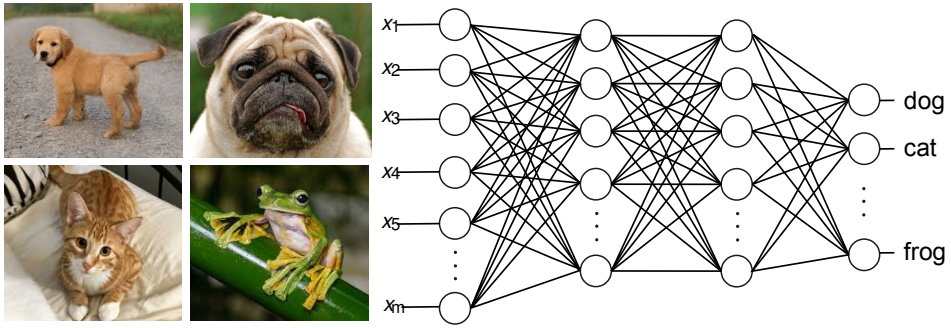
- large increase in parameters needed to achieve a certain level of performance on a task
- tends to memorise training inputs, fails to generalise



Why Build Deep Neural Networks?

To solve complex tasks (i.e. to perform complex mappings) we need larger networks (i.e. more parameters).

2) could build deep network

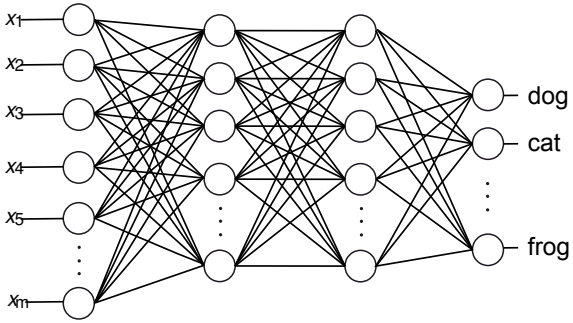


Why Build Deep Neural Networks?

To solve complex tasks (i.e. to perform complex mappings) we need larger networks (i.e. more parameters).

2) could build deep network

- smaller increase in parameters needed to achieve a certain level of performance on a task
- aims to capture the natural "hierarchy" of the task to improve generalisation



Why Build Deep Neural Networks?

Deep networks aim to provide a hierarchy of representations with increasing level of abstraction.

Natural for dealing with many tasks, e.g.:

Image recognition
- pixel → edge → texton/contour → part → object

Text recognition
- character → word → clause → sentence → story

Speech recognition
- sound → phone → phoneme → word → clause → sentence

What is a Deep Neural Network?

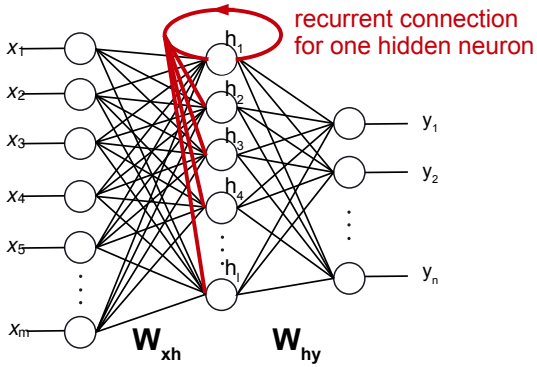
"Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones". [Yoshua Bengio]

Why Build Deep Neural Networks?

Need to be able to build deep neural networks also arises from recurrent networks

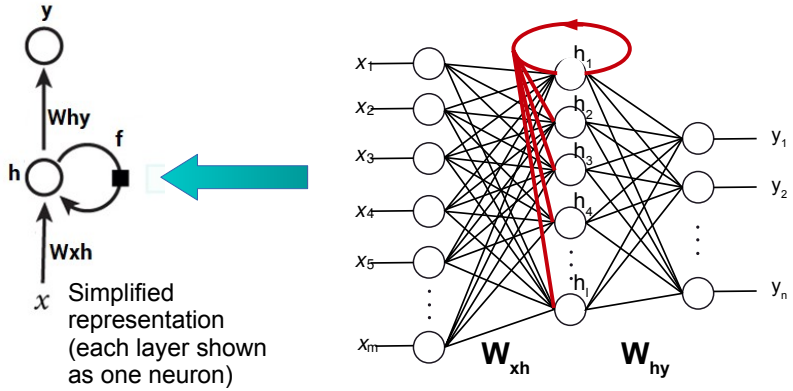
Recurrent networks process temporal information (i.e. where values of **x** change over time).

In a simple recurrent network, connections take the output of every hidden neuron and feed it in as addition input at the next iteration.



Why Build Deep Neural Networks?

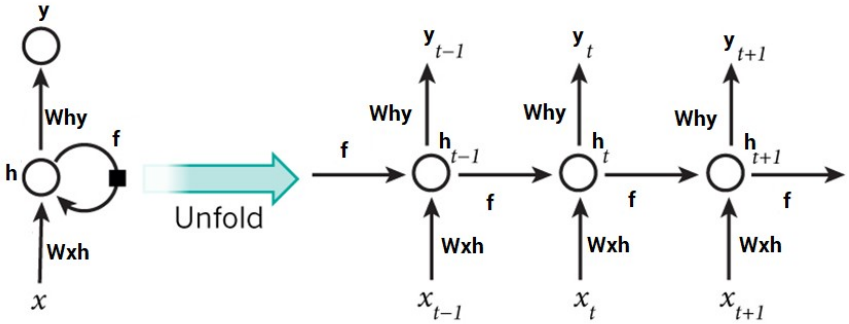
Need to be able to build deep neural networks also arises from recurrent networks



Why Build Deep Neural Networks?

Need to be able to build deep neural networks also arises from recurrent networks

Training a recurrent network is achieved by unfolding the network and using backpropagation on the unfolded network



There is one layer in the unfolded network for each time-step, so recurrent networks for processing long sequences are equivalent to very deep networks

Part b

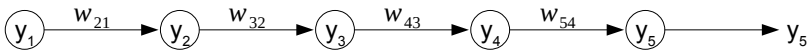
Difficulties Training Deep NNs

Contents

Difficulties Training Deep Neural Networks

Vanishing gradient problem

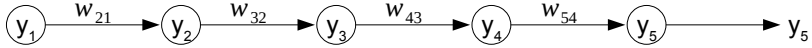
Consider a deep network with one neuron per layer:



Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



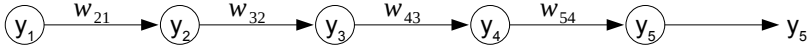
Applying backpropagation:

$$\Delta w_{54} = \eta (t - y_5)\, \varphi'(w_{54}\, y_4)\, y_4$$

Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



Applying backpropagation:

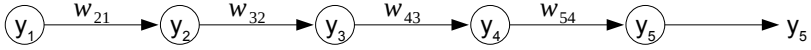
$$\Delta w_{54} = \eta (t - y_5)\, \varphi'(w_{54} y_4)\, y_4$$

$$\Delta w_{43} = \eta (t - y_5)\, \varphi'(w_{54} y_4)\, w_{54}\, \varphi'(w_{43} y_3)\, y_3$$

Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



Applying backpropagation:

$$\Delta w_{54} = \eta (t - y_5)\, \varphi'(w_{54} y_4)\, y_4$$

$$\Delta w_{43} = \eta (t - y_5)\, \varphi'(w_{54} y_4)\, w_{54}\, \varphi'(w_{43} y_3)\, y_3$$

$$\Delta w_{32} = \eta (t - y_5)\, \underbrace{\varphi'(w_{54} y_4)\, w_{54}}\, \underbrace{\varphi'(w_{43} y_3)\, w_{43}}\, \varphi'(w_{32} y_2)\, y_2$$

Each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji} x_i)\, w_{ji}$

Difficulties Training Deep Neural Networks

Vanishing gradient problem

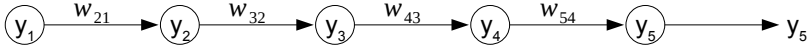
Consider a deep network with one neuron per layer:

Each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji} x_i)\, w_{ji}$

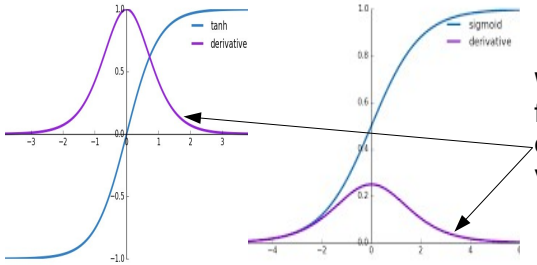
Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



Each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji}x_i)w_{ji}$

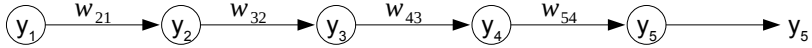


When using a standard activation function, like tanh or sigmoid, derivative is small for most values of ***wx***

Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



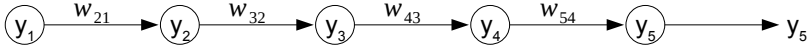
Each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji}x_i)w_{ji}$

When using a standard approach to weight initialisation, such as choosing weights from a distribution with a mean of zero, then this term may also be small

Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



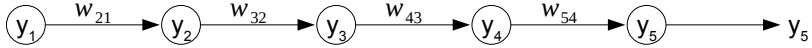
$w_{21} \approx 0.004 \quad w_{32} \approx 0.016 \quad w_{43} \approx 0.063 \quad \Delta w_{54} \approx 0.25$

If each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji}x_i)w_{ji} \ll 1$

Difficulties Training Deep Neural Networks

Vanishing gradient problem

Consider a deep network with one neuron per layer:



$\Delta w_{21} \approx 0.004 \quad \Delta w_{32} \approx 0.016 \quad \Delta w_{43} \approx 0.063 \quad \Delta w_{54} \approx 0.25$

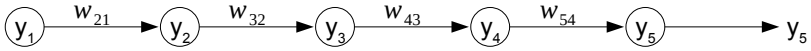
If not careful, the gradient tends to get smaller as we move backward through the hidden layers. This means:

- neurons in the earlier layers learn much more slowly than neurons in later layers,
- early layers contribute nothing to solving the task (keep initial random weights), and hence:
  - making network deeper does not improve performance

Difficulties Training Deep Neural Networks

Exploding gradient problem

Consider a deep network with one neuron per layer:



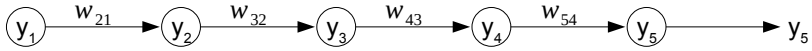
Each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji}x_i)w_{ji}$

Note if weights are initialised to, or learn, large values, then $\varphi'(w_{ji}x_i)w_{ji}>1$

Difficulties Training Deep Neural Networks

Exploding gradient problem

Consider a deep network with one neuron per layer:



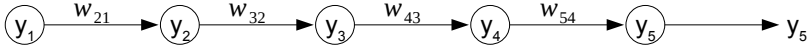
$\Delta w_{21}\approx 31.25$ $\Delta w_{32}\approx 6.25$ $\Delta w_{43}\approx 1.25$ $\Delta w_{54}\approx 0.25$

If each time the error is propagated further backwards it is multiplied by a factor of the form $\varphi'(w_{ji}x_i)w_{ji}>1$

Difficulties Training Deep Neural Networks

Exploding gradient problem

Consider a deep network with one neuron per layer:



$\Delta w_{21}\approx 31.25$ $\Delta w_{32}\approx 6.25$ $\Delta w_{43}\approx 1.25$ $\Delta w_{54}\approx 0.25$

If not careful, the gradient tends to get larger as we move backward through the hidden layers. This means:

- neurons in the earlier layers make, large, often random changes in their weights,
- later layers can not learn due to constantly changing output of earlier layers, and hence:
  - making network deeper makes performance worse

Difficulties Training Deep Neural Networks

Backpropagation is inherently unstable (gradients can easily vanish or explode).

To train deep networks it is necessary to mitigate this issue, by using:

- activation functions with non-vanishing derivatives
- better ways to initialise weights
- adaptive variations on standard backpropagation
- batch normalisation
- skip connections

Another issue is that deep networks have lots of parameters, it is therefore necessary to have:

- very large labelled datasets
- large computational resources

Part c

Overcoming the Difficulties with Training Deep NNs

Contents

Difficulties Training Deep Neural Networks

Backpropagation is inherently unstable (gradients can easily vanish or explode).

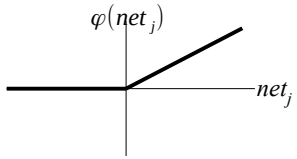
To train deep networks it is necessary to mitigate this issue, by using:

- activation functions with non-vanishing derivatives
- better ways to initialise weights
- adaptive variations on standard backpropagation
- batch normalisation
- skip connections

Activation Functions with Non-Vanishing Derivatives

Rectified Linear Unit (ReLU)

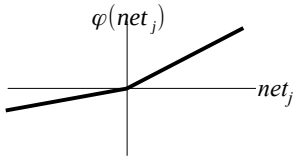
$$\varphi(net_j) = \begin{matrix} net_j \; if \; net_j \geq 0 \\ 0 \; if \; net_j < 0 \end{matrix}$$



Leaky Rectified Linear Unit (LReLU)

$$\varphi(net_j) = \begin{matrix} net_j \; if \; net_j \geq 0 \\ a \times net_j \; if \; net_j < 0 \end{matrix}$$

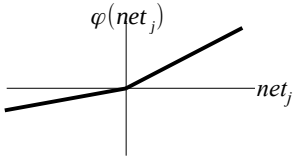
where a is fixed and same for all neurons



Parametric Rectified Linear Unit (PReLU)

$$\varphi(net_j) = \begin{matrix} net_j \; if \; net_j \geq 0 \\ a_j \times net_j \; if \; net_j < 0 \end{matrix}$$

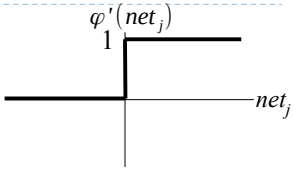
where a_j is learnt for each neuron



Activation Functions with Non-Vanishing Derivatives

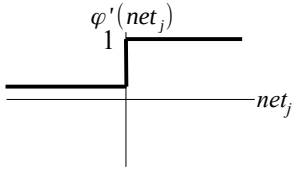
Rectified Linear Unit (ReLU)

$$\varphi'(net_j) = \begin{matrix} 1 \, if \, net_j \geq 0 \\ 0 \, if \, net_j < 0 \end{matrix}$$



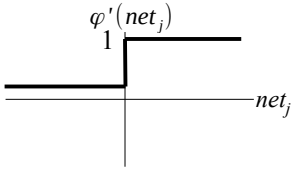
Leaky Rectified Linear Unit (LReLU)

$$\varphi'(net_j) = \begin{matrix} 1 \, if \, net_j \geq 0 \\ a \, if \, net_j < 0 \end{matrix}$$



Parametric Rectified Linear Unit (PReLU)

$$\varphi'(net_j) = \begin{matrix} 1 \, if \, net_j \geq 0 \\ a_j \, if \, net_j < 0 \end{matrix}$$



Better Ways to Initialise Weights

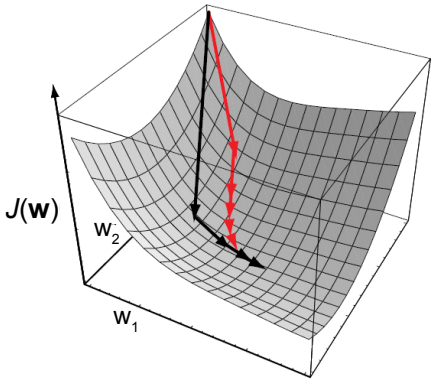
For weights connecting *m* inputs to *n* outputs:

Name of method	Activation function	Choose weights from	
		Uniform distribution with range:	Normal distribution with mean=0 and standard deviation:
Xavier / Glorot	sigmoid / tanh	$\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$	$\sqrt{\frac{2}{m+n}}$
Kaiming / He	ReLU / LReLU / PReLU	$\left(-\sqrt{\frac{6}{m}}, \sqrt{\frac{6}{m}}\right)$	$\sqrt{\frac{2}{m}}$

Adaptive Versions of Backpropagation

Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large.

Recall, backpropagation is performing gradient descent to find parameters that minimise a cost function (e.g. the number of misclassified samples)

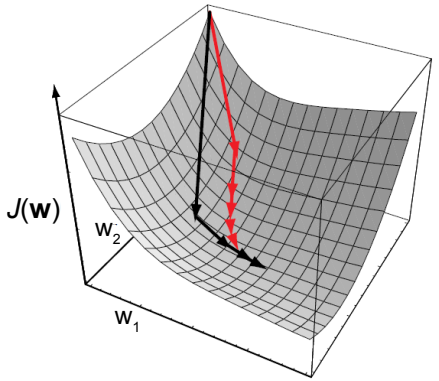


Adaptive Versions of Backpropagation

Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large.

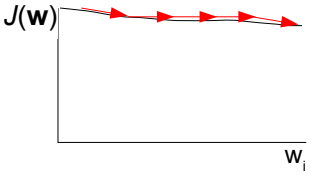
Variation in the magnitude of the gradient may occur between:

- different layers (due to vanishing and exploding gradients)
- different parts of the cost function for a single neuron
- different directions for a multi-dimensional function

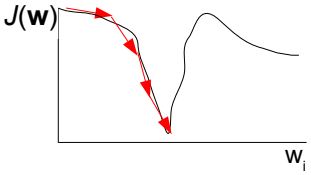


Adaptive Versions of Backpropagation

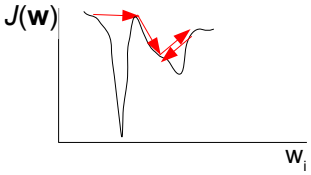
Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large.



gradient too low: many iterations for little gain in performance



gradient just right: optimal parameters found with few iterations

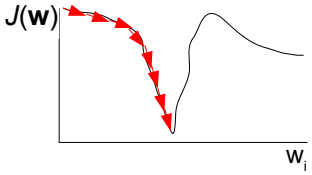


gradient too large: fails to find optimal parameters, oscillates

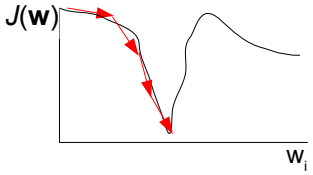
Adaptive Versions of Backpropagation

Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large.

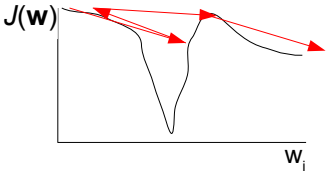
An analogous issue occurs when trying to select an appropriate learning rate:



rate too low: many iterations for little gain in performance



rate just right: optimal parameters found with few iterations

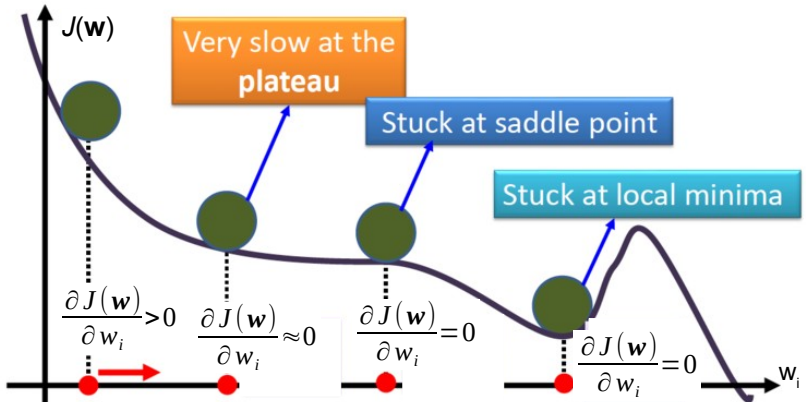


rate too large: fails to find optimal parameters, oscillates

Adaptive Versions of Backpropagation

Backpropagation struggles to deal with gradients in the cost function $J(\mathbf{w})$ that are too small, or too large.

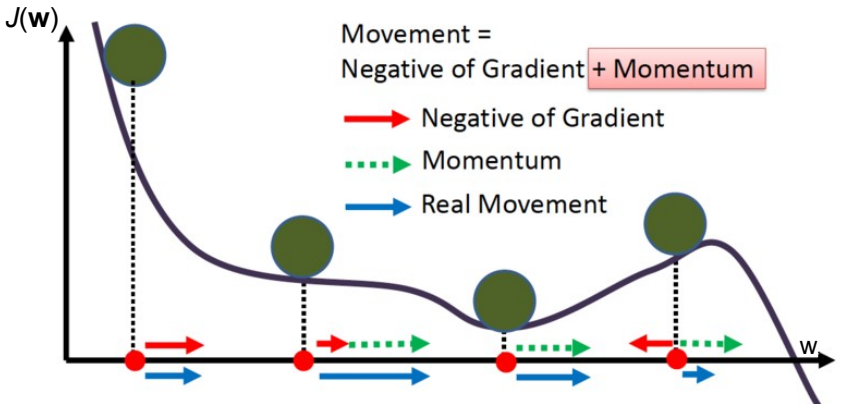
Contrary to previous illustrations, step size is proportional to the gradient, this make problem even worse:



Adaptive Versions of Backpropagation

momentum: adds moving average of previous gradient to current gradient

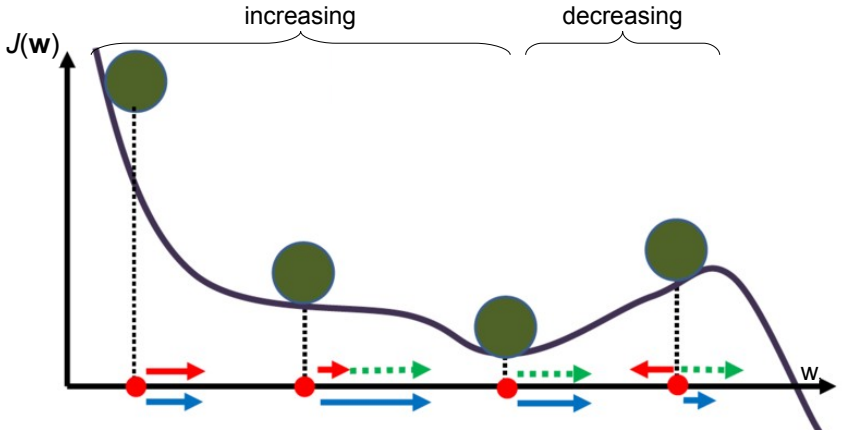
- Increases step size when weight changes are consistently in same direction (helps with plateaus and local minima)



Adaptive Versions of Backpropagation

adaptive learning rate: vary the learning rate (for individual parameters) during training

- increasing learning rate if cost is decreasing
- decreasing learning rate if cost is increasing



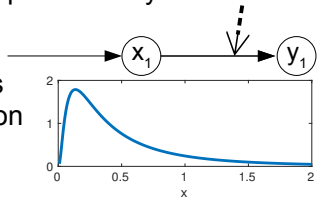
Batch Normalisation

Learning in one layer of a network will change the distribution of inputs received by the subsequent layer of the network, e.g.:

- at some time during training

weight of 2nd neuron will have been trained to be appropriate for outputs produced by 1st neuron

distribution of outputs produced by 1st neuron with current weights



Adaptive Versions of Backpropagation

adaptive learning rate: vary the learning rate (for individual parameters) during training

- increasing learning rate if cost is decreasing
- decreasing learning rate if cost is increasing

Backpropagation algorithms with adaptive learning:

- AdaGrad
- RMSprop

Backpropagation algorithms with adaptive learning and momentum:

- ADAM
- Nadam

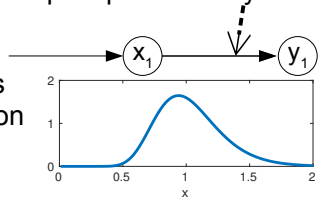
Batch Normalisation

Learning in one layer of a network will change the distribution of inputs received by the subsequent layer of the network, e.g.:

- at some later time during training

weight of 2nd neuron will have to be re-trained to be appropriate for new outputs produced by 1st neuron

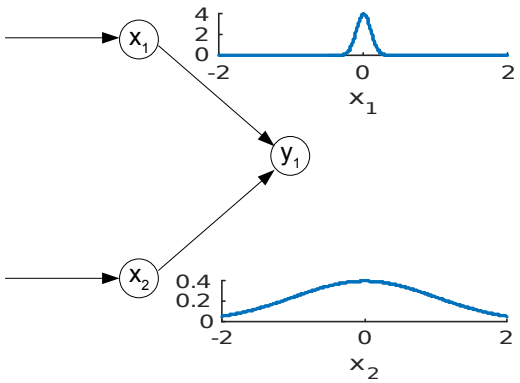
distribution of outputs produced by 1st neuron with updated weights



Consequently, learning is slow as later layers are always having to compensate for changes made to earlier layers.

Batch Normalisation

Different inputs to one neuron can have very different scales, e.g.:



Inputs with smaller scales will tend to have less influence than ones with large scale (even if the smaller scale inputs are more discriminatory).

Batch Normalisation

Batch normalisation attempts to solve both these issues by scaling the output of each individual neuron so that it has a mean close to 0 and a standard deviation close to 1.

$$BN(x)=\beta+\gamma\frac{x-E(x)}{\sqrt{Var(x)+\epsilon}}$$

- β and γ are parameters learnt by backpropagation
- ϵ is a constant used to prevent division-by-zero errors
- $E(x)$ is the mean of x
- $Var(x)$ is the variance (the squared standard-deviation) of x

$E(x)$ and $Var(x)$ can be calculated using the values of x in the current batch or using all the training data presented so far

Batch Normalisation

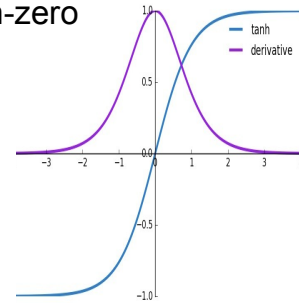
Batch normalisation can be applied

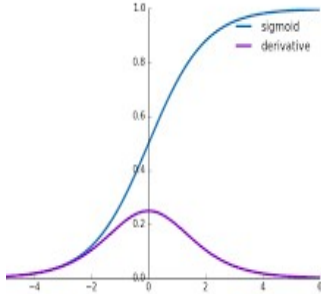
- before $y_j=\varphi(BN(w_{ji}x_i))$
- or after $y_j=BN(\varphi(w_{ji}x_i))$

the activation function

Batch Normalisation

- Enables saturating activation functions to be used as it limits activations to the range where the gradients are non-zero

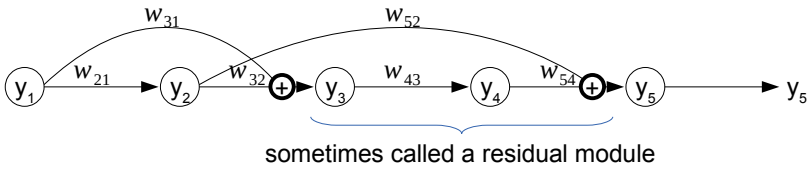




- Make weight initialisation less critical
- generally stabilises learning (gradients less likely to vanish or explode).

Skip Connections

Connections that skip one or more layers of the network:



Skip connections let gradients by-pass parts of the network where the gradient has vanished

- Network effectively becomes shallower, but this may be temporary

Part d

Convolutional NNs: Convolutional Layers

Contents

- What is a Deep Neural Network?
- Why Build Deep Neural Networks?
- Difficulties Training Deep Neural Networks
  - solutions
- Convolutional Neural Networks (CNNs)
  - convolutional layers
  - pooling and fully-connected layers
- Issues with CNNs/deep networks

Convolutional Neural Networks (CNNs)

The most popular type of deep neural network.

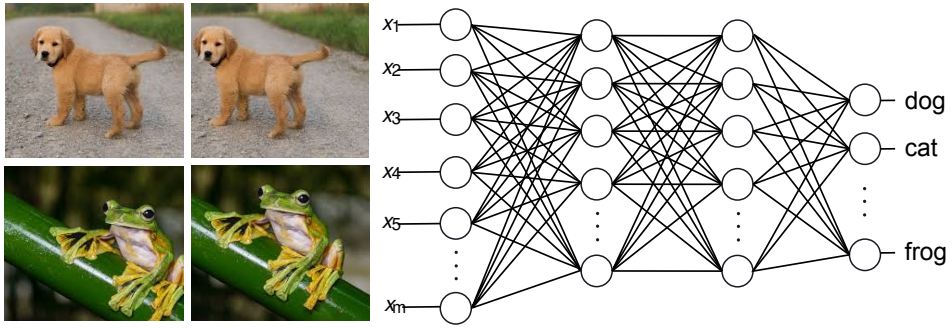
A CNN is:

- Any neural network in which at least one layer has an transfer function implemented using convolution/cross-correlation.
- (as opposed to vector multiplication).

Motivated by desire to recognise patterns with tolerance to location.

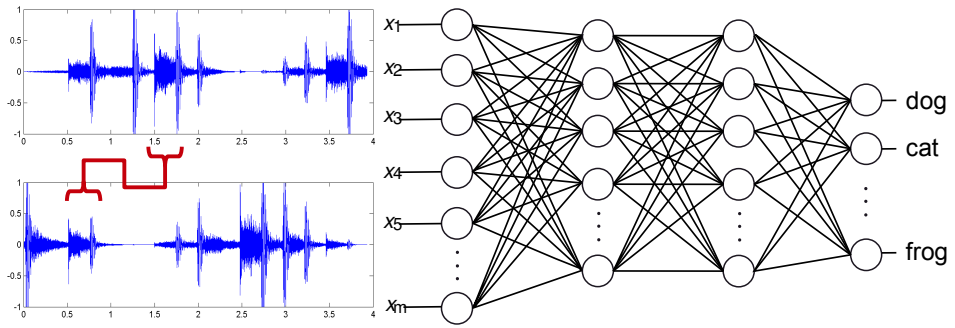
Convolutional Neural Networks (CNNs)

Motivated by desire to recognise patterns with tolerance to location, e.g. spatial location:



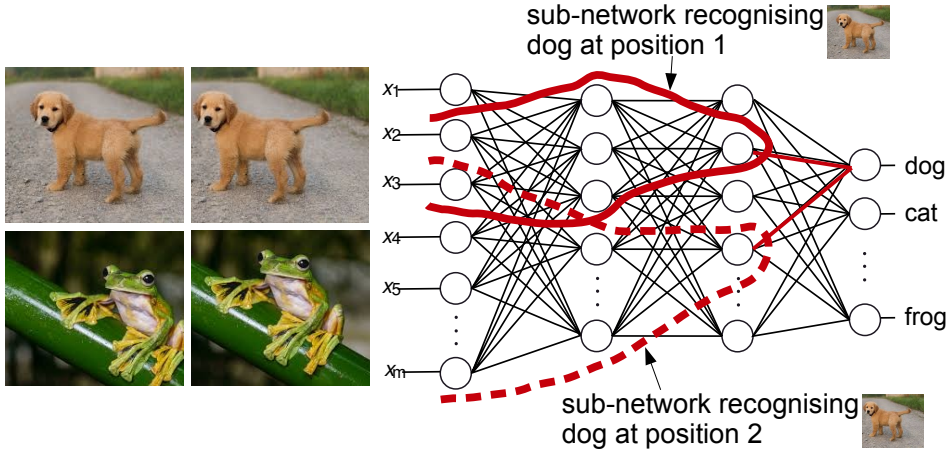
Convolutional Neural Networks (CNNs)

Motivated by desire to recognise patterns with tolerance to location, e.g. temporal location:



Convolutional Neural Networks (CNNs)

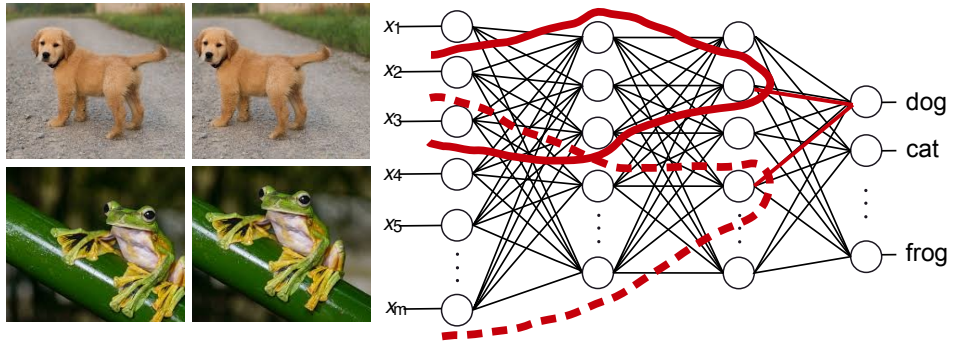
Tolerance to location could be achieved by learning to recognise the pattern at each location independently.



Convolutional Neural Networks (CNNs)

Tolerance to location could be achieved by learning to recognise the pattern at each location independently.

Computationally more efficient to share the weights between the sub-networks (i.e. to have multiple copies of the same sub-network processing different locations).



CNNs: Convolution Layers

Weight sharing is achieved by using cross-correlation as the transfer function.

A neuron's weights are defined as an array, e.g.: called a "mask", or "filter", or "kernel"

-1	1
0	-2

The values in this array, the weights, will be learnt using backpropagation.

The input is also an array, e.g.: an image or the output of the preceding layer of the network

-1	1	0.5	1
0	-2	-1	0
0.5	-1	-1	1
-1	1	0	-2

(note: can have 1D input and use 1D kernels.)

CNNs: Convolution Layers

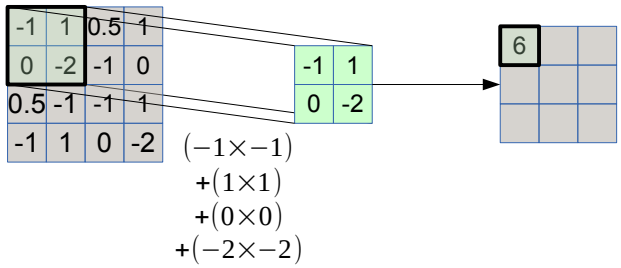
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

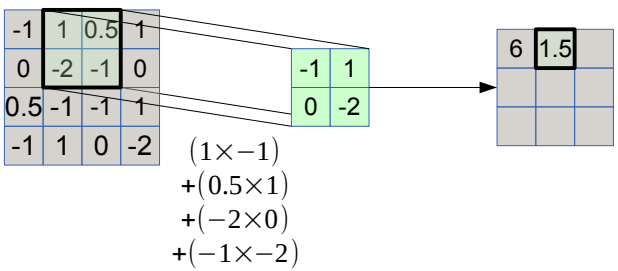
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

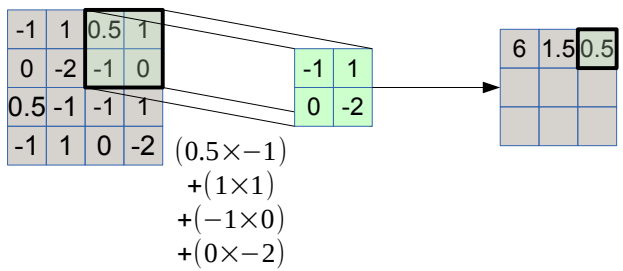
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

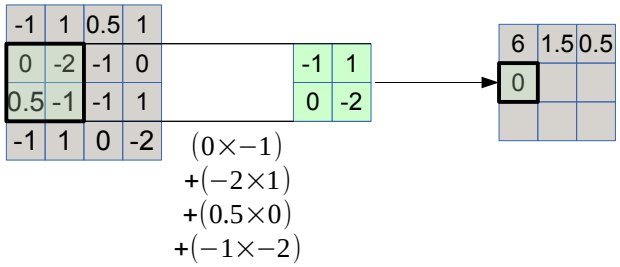
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

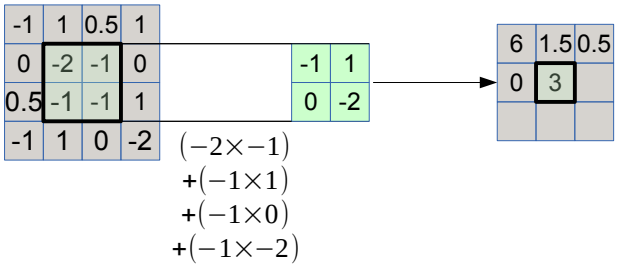
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

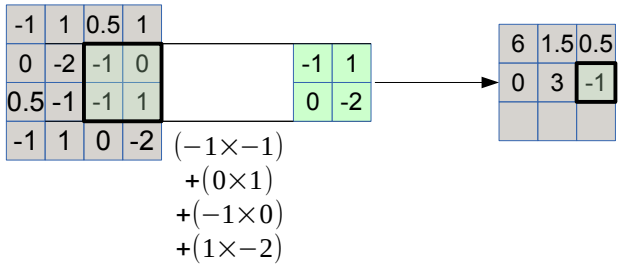
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

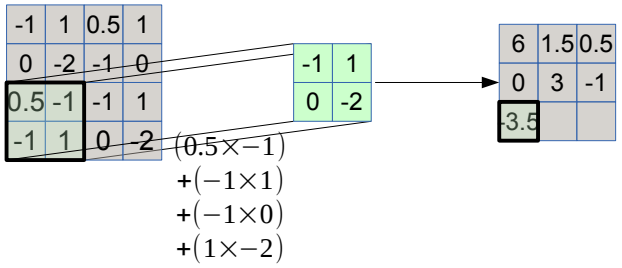
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

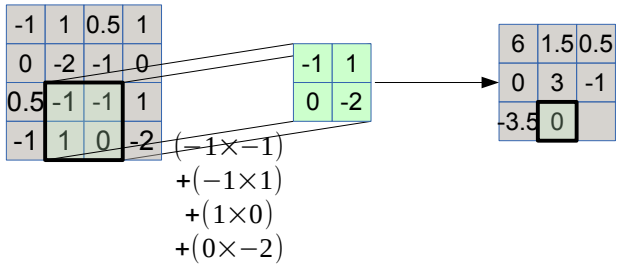
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



CNNs: Convolution Layers

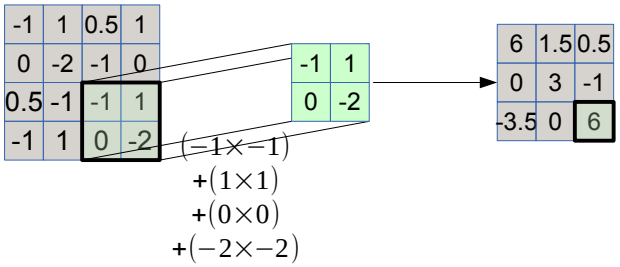
Weight sharing is achieved by using cross-correlation as the transfer function.

Cross-correlation is implemented as follows.

For each location in the input array in turn:

1. Multiply each weight by the corresponding value in the input
2. Sum these products and write answer in the corresponding location in the output array

e.g.:



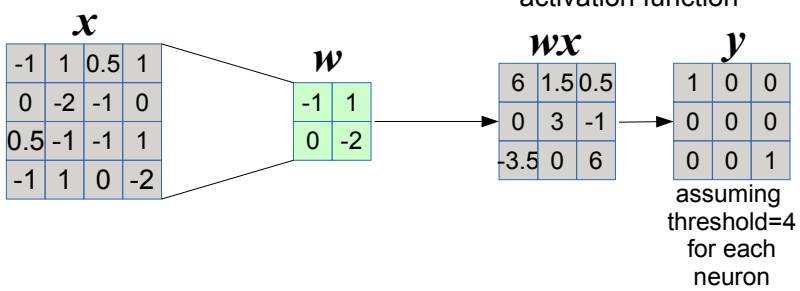
CNNs: Convolutional Layers

Having used cross-correlation to implement the transfer function, the final output can be calculated by applying an activation function to each output.

e.g. to implement a layer of linear threshold units:

$$y_j = \underbrace{H\left(\sum_i w_{ji} x_i - \theta_j\right)}_{\text{activation function}}$$

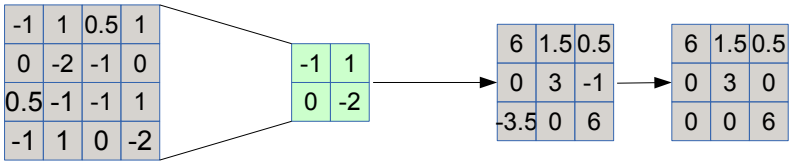
transfer function



CNNs: Convolutional Layers

Having used cross-correlation to implement the transfer function, the final output can be calculated by applying an activation function to each output.

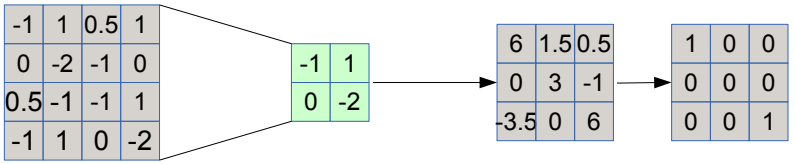
e.g. using ReLU as the activation function:



CNNs: Convolution Layers

Note, the output is also an array of numbers.

- sometime called a “feature map”

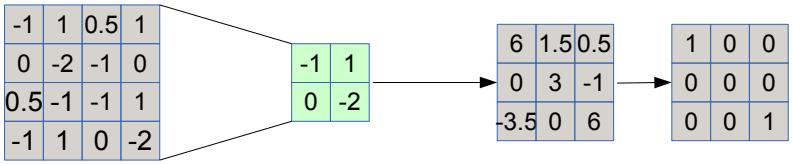


CNNs: Convolution Layers

Note, the output is strongest where the location in the input matches the weights in the kernel.

Note, the output is the same at the two locations in the input with the same values.

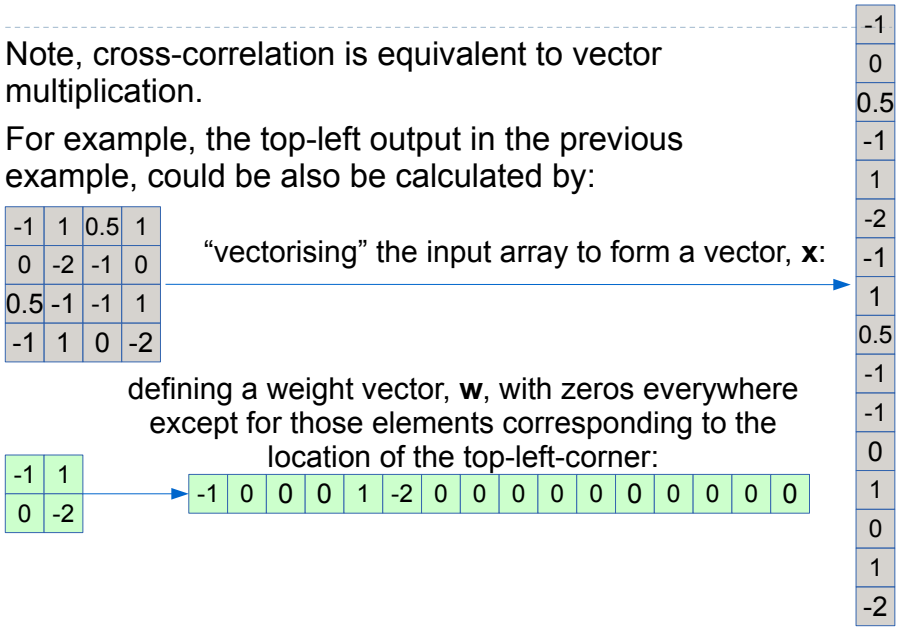
Provides tolerance to location: the same pattern in different locations produces the same output.



CNNs: Convolutional Layers

Note, cross-correlation is equivalent to vector multiplication.

For example, the top-left output in the previous example, could be also be calculated by:



CNNs: Convolutional Layers

Note, cross-correlation is equivalent to vector multiplication.

Calculating the output of the transfer function in the normal way would give:

$$\sum_i w_{ji} x_i = \boldsymbol{wx}$$

-1 0 0 0 1 -2 0 0 0 0 0 0 0 0 0 0 x

-1 0 0.5 -1 1 -2 -1 1 0.5 -1 -1 0 1 0 1 -2

=6

The other outputs in the previous example, could be also be calculated using appropriate weight vectors.

However, it is easier to use one kernel to produce all the outputs using cross-correlation.

CNNs: Convolutional Layers

Note, despite being called “convolution” layers (and the resulting networks being called “convolutional” neural networks, the operation performed by the convolutional layers is cross-correlation (or linear filtering) not convolution.

cross-correlation:

$$Y(i,j) = X \star H = \sum_{k,l} X(i+k, j+l) H(k,l)$$

-1	1
0	-2

convolution:

$$Y(i,j) = X * H = \sum_{k,l} X(i+k, j+l) H(-k,-l)$$

convolution same as cross-correlation with a rotated mask:

-2	0
1	-1

CNNs: Convolutional Layers

The mask defines the weights of a neuron.

A neuron only receives input from a (small) region of the input array

- called its receptive field (RF)

Different masks can have different RFs sizes, e.g.:

2x2 mask

-1	1
0	-2

3x3 mask

-1	1	0
0	-2	-2
-1	1	0.1

1x5 mask

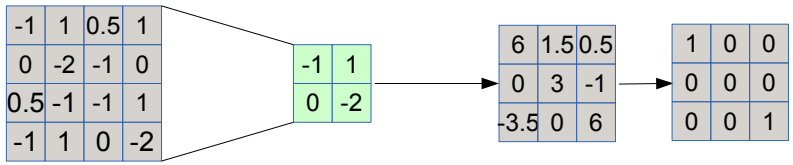
1	0	-2	-2	-1

(for 1D signal)

CNNs: Convolutional Layers

Cross-correlation generates “copies” of the same neuron at each location

- the output is the response of multiple, identical, neurons with the same weights but different RF locations



CNNs: Convolutional Layers

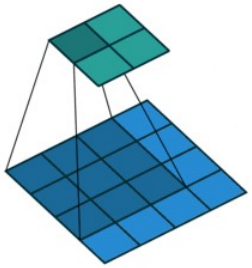
There are a number of variations on standard cross-correlation

Padding defines number of 0 valued pixels appended to each edge of the input. Increasing padding increases size of output.

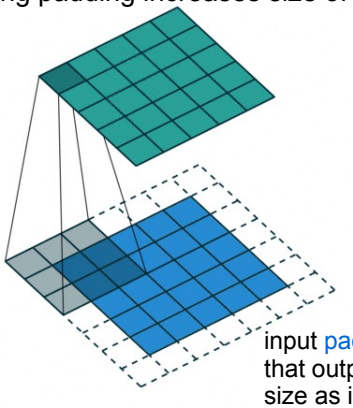
standard

output

input



no padding of input: output smaller than input (called “valid” padding)



input padded so that output is same size as input (called “same” padding)

CNNs: Convolutional Layers

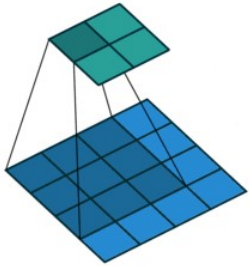
There are a number of variations on standard cross-correlation

Stride defines number of pixels moved between calculations of the output. Increasing stride reduces size of output.

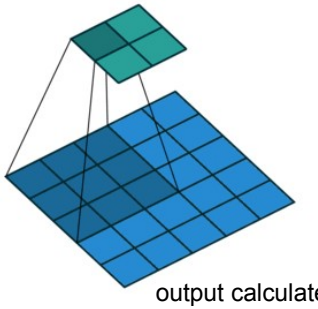
standard

output

input



output calculated at every location (stride=1). “Non-strided”



output calculated every other location (stride=2). Larger strides also possible.

CNNs: Convolutional Layers

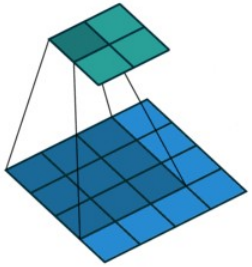
There are a number of variations on standard cross-correlation

Dilation defines the spacing between the pixels to which the mask is applied. Increasing dilation increases RF size.

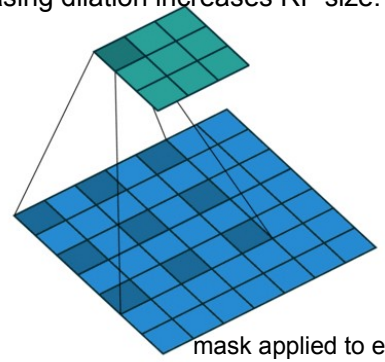
standard

output

input



mask applied to adjacent pixels in the input array (dilation=1)

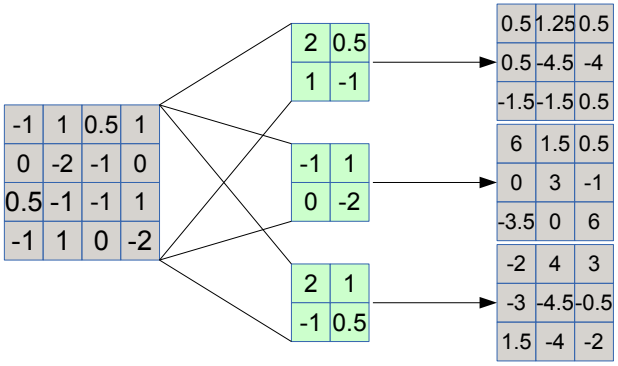


mask applied to every other pixel (dilation=2). Larger dilations also possible.

CNNs: Convolutional Layers

Multiple masks can be used to detect multiple patterns

This will result in multiple output arrays:

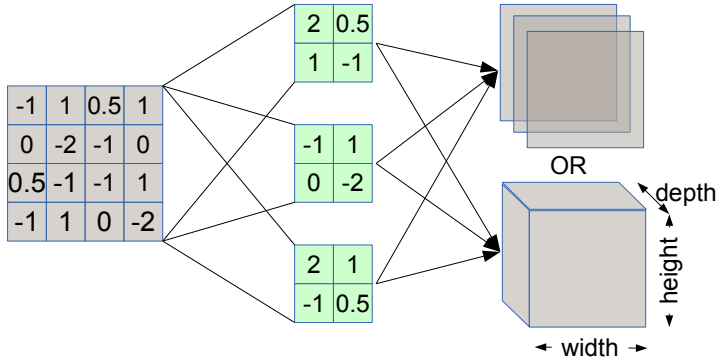


CNNs: Convolutional Layers

Multiple masks can be used to detect multiple patterns

This will result in multiple output arrays.

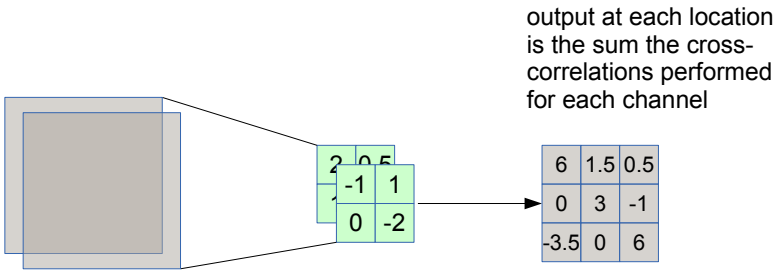
Such a multi-channel output is often represented as a stack of squares or as a cuboid:



CNNs: Convolutional Layers

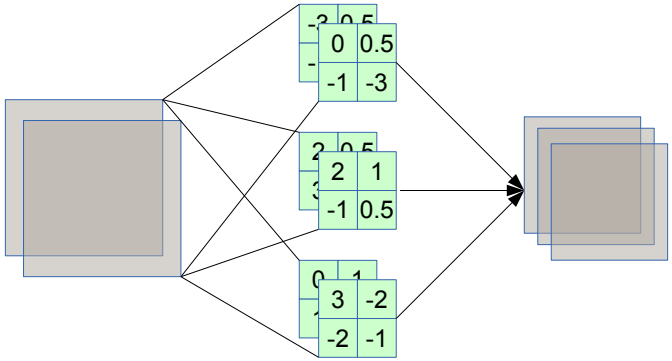
A neuron can have multi-channel input (e.g. from different colour channels of an image, or multichannel output from a preceding convolutional layer).

In this case the mask has multiple channels:



CNNs: Convolutional Layers

Typically, the input, output and (the multiple) masks are all multi-channel:

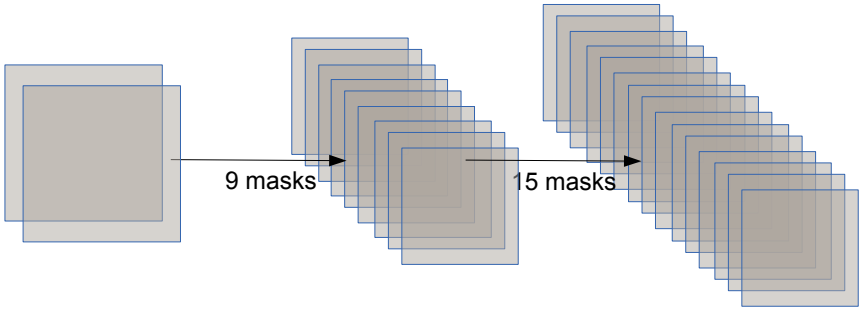


CNNs: Convolutional Layers

It may be necessary to use many masks to learn to recognise many different sub-patterns in the data.

However, this will result in many channels in the input to the next layer.

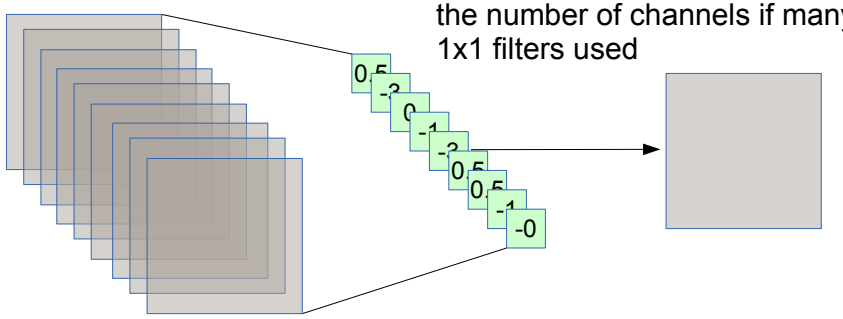
1x1 convolution can be used to reduce the number of channels, and reduce the computational burden of the next layer.



CNNs: Convolutional Layers

1x1 convolution

- contains a single weight in each channel
- always applied with stride=1, and no padding
- provides a weighted sum of the input channels at each location
- multiple 1x1 filters can be used to produce differently weighed sums of the input channels
- can also be used to increase the number of channels if many 1x1 filters used



Part e

Convolutional NNs: Pooling and Fully-Connected Layers

Contents

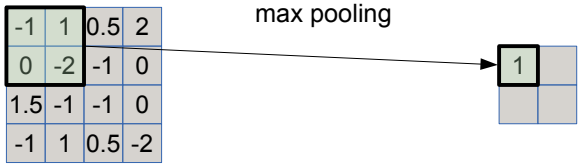
CNNs: Pooling Layers

CNNs also contain pooling layers to increase tolerance to:

- the location of patterns
- the configuration of sub-patterns

Pooling layers replace values within a region of the input array with a single value calculated, typically, as the:

- mean (“average pooling”)
- maximum (“max pooling”)



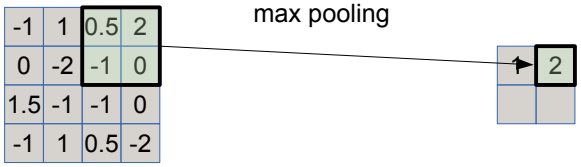
CNNs: Pooling Layers

CNNs also contain pooling layers to increase tolerance to:

- the location of patterns
- the configuration of sub-patterns

Pooling layers replace values within a region of the input array with a single value calculated, typically, as the:

- mean (“average pooling”)
- maximum (“max pooling”)



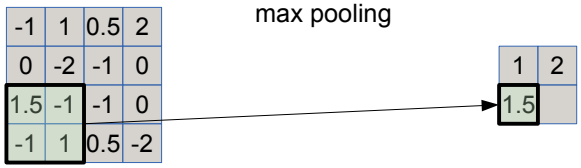
CNNs: Pooling Layers

CNNs also contain pooling layers to increase tolerance to:

- the location of patterns
- the configuration of sub-patterns

Pooling layers replace values within a region of the input array with a single value calculated, typically, as the:

- mean (“average pooling”)
- maximum (“max pooling”)



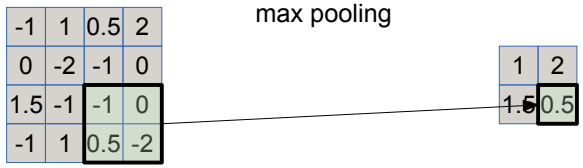
CNNs: Pooling Layers

CNNs also contain pooling layers to increase tolerance to:

- the location of patterns
- the configuration of sub-patterns

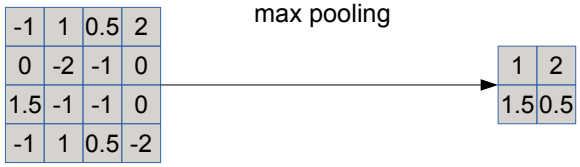
Pooling layers replace values within a region of the input array with a single value calculated, typically, as the:

- mean (“average pooling”)
- maximum (“max pooling”)



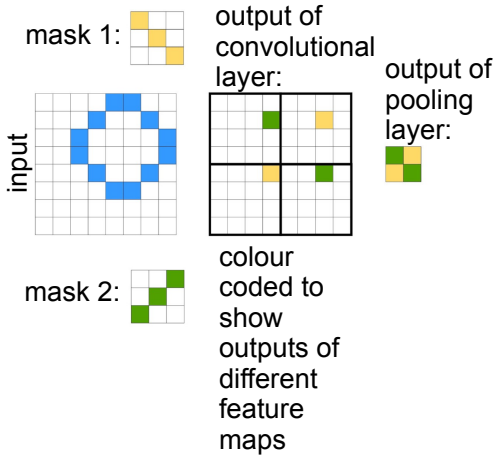
CNNs: Pooling Layers

- Pooling typically has a stride>1 to decrease the spatial size of the layer
- Different sized pooling regions (RFs) can be used
  - if same size as the input array, the output is a scalar value: this is called “global pooling”
- The pooling operation is user specified, not learnt
- Pooling also called “sub-sampling” or “down-sampling”



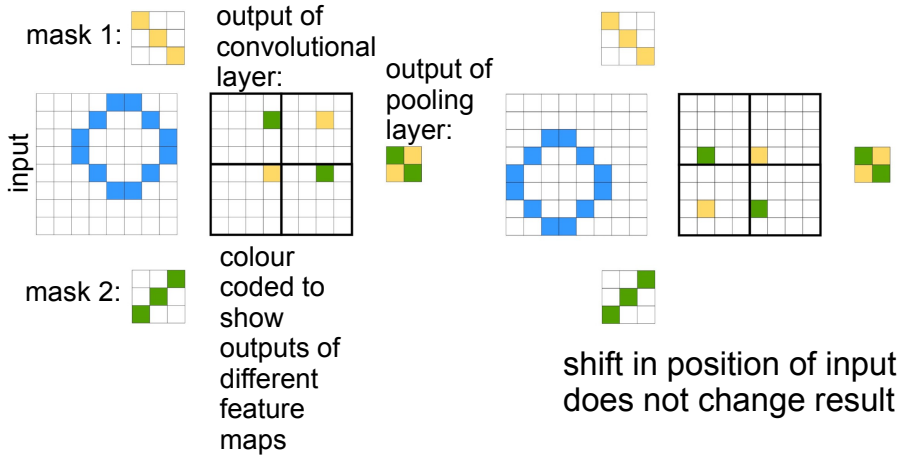
CNNs: Pooling Layers

Pooling increases tolerance to location:



CNNs: Pooling Layers

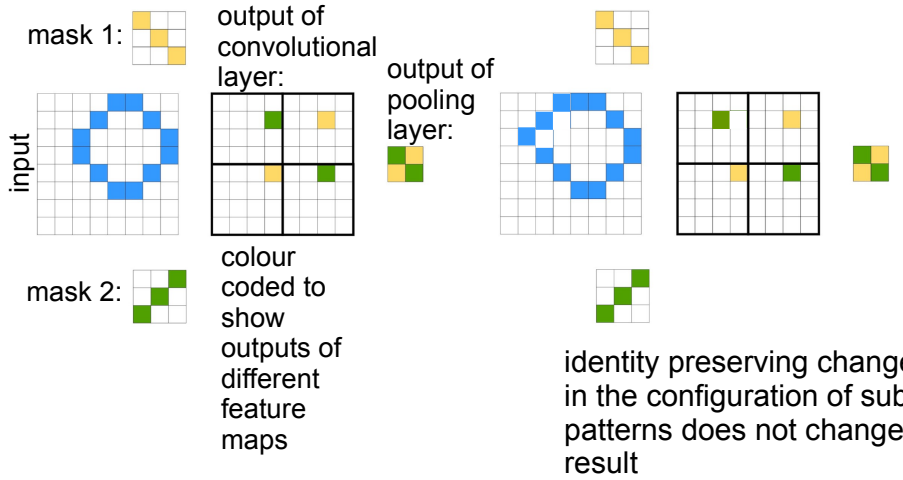
Pooling increases tolerance to location:



shift in position of input does not change result

CNNs: Pooling Layers

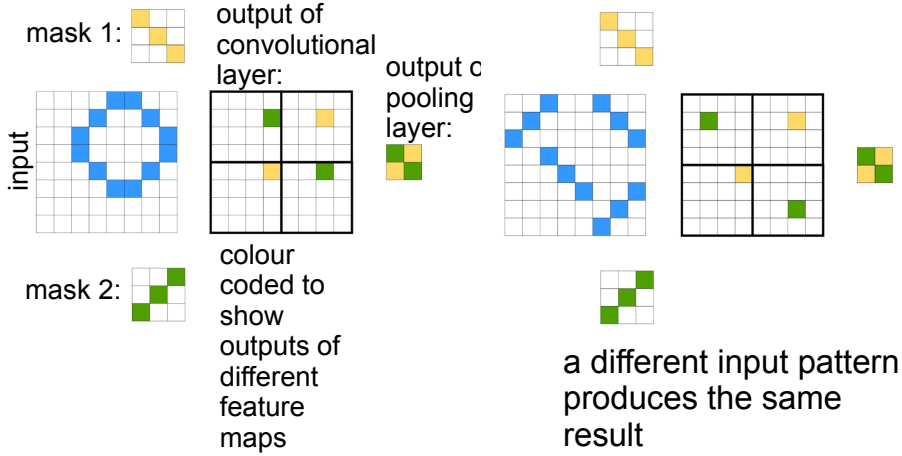
Pooling increases tolerance to configuration:



identity preserving changes in the configuration of sub-patterns does not change result

CNNs: Pooling Layers

Pooling decreases discrimination:

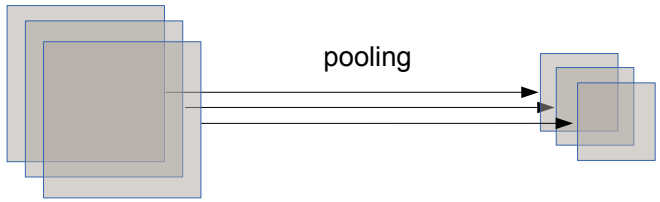


a different input pattern produces the same result

CNNs: Pooling Layers

Typically, pooling is applied separately to each input array

- so the number of output channels is equal to the number of input channels

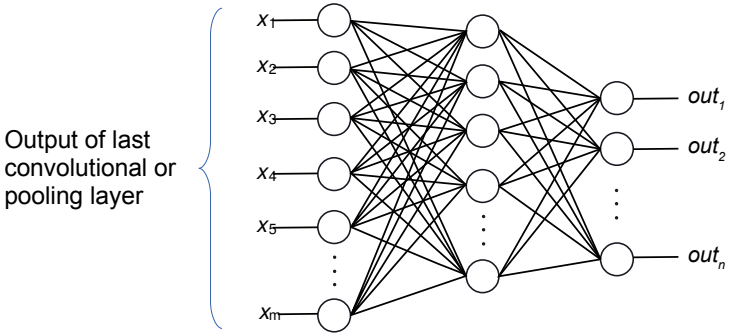


However, it is possible to perform cross-channel pooling to reduce the resulting number of channels

CNNs: Fully Connected Layers

Typically, particularly when applied to classification tasks, the final layers of a CNN are fully connected layers:

- they work exactly the same as a traditional, feedforward, neural network, e.g. a multilayer perceptron
- Classification is then performed using the final layer of the fully connected layer



CNNs: Fully Connected Layers

To create the input to the 1st fully connected layer it is necessary to "flatten" the output of the last convolutional or pooling layer

- this is the same as vectorisation:

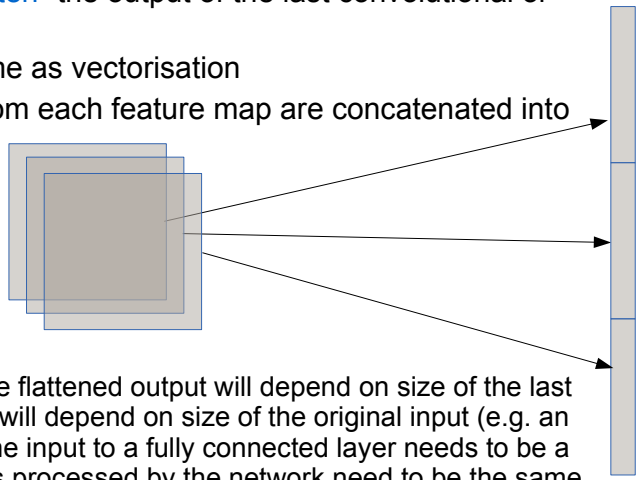
-1	1	0.5	1
0	-2	-1	0
0.5	-1	-1	1
-1	1	0	-2

-1
0
0.5
-1
1
-2
-1
1
0.5
-1
-1
0
1
0
1
-2

CNNs: Fully Connected Layers

To create the input to the 1st fully connected layer it is necessary to "flatten" the output of the last convolutional or pooling layer

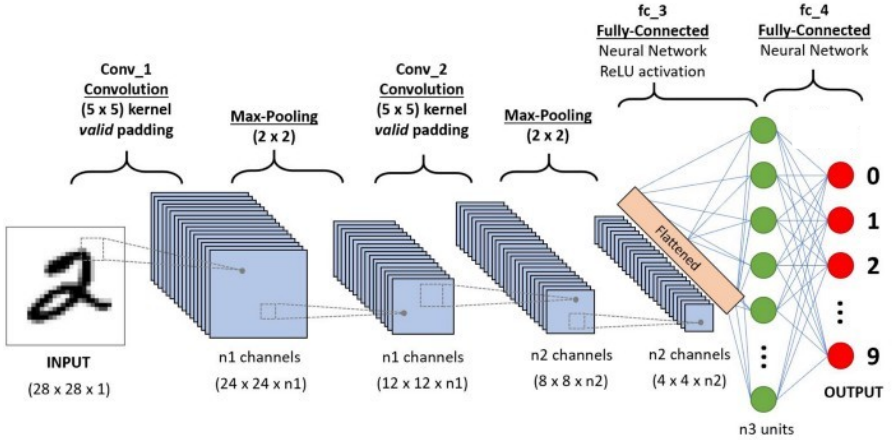
- this is the same as vectorisation
- the vectors from each feature map are concatenated into one vector:



Note, the size of the flattened output will depend on size of the last feature map which will depend on size of the original input (e.g. an image). Because the input to a fully connected layer needs to be a fixed size, all inputs processed by the network need to be the same size.

Convolutional Neural Networks

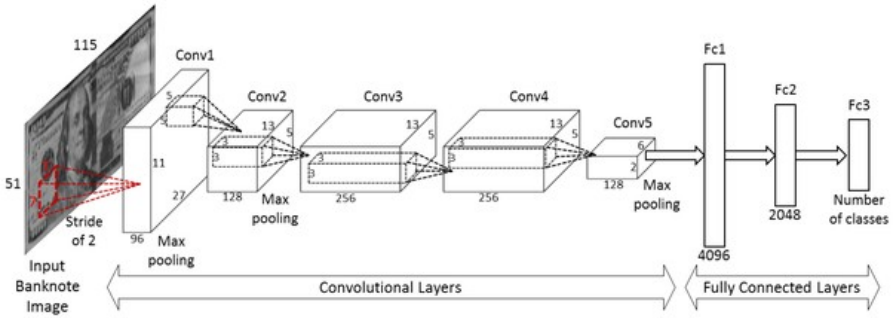
The final network may contain multiple convolutional, pooling and fully connected layers connected sequentially, e.g.:



Convolutional Neural Networks

The final network may contain multiple convolutional, pooling and fully connected layers connected sequentially, e.g.:

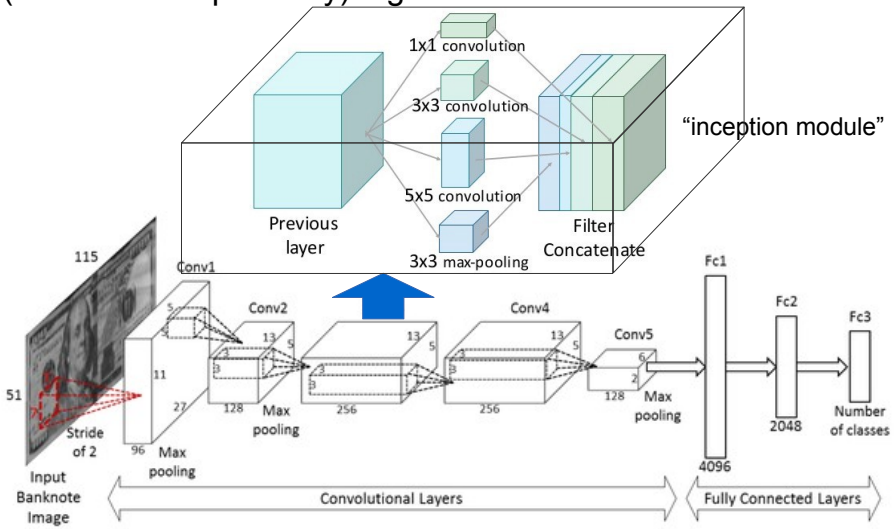
an alternative format for drawing a similar network



note, cuboids are rotated, pooling layers are not explicitly shown, masks are illustrated by smaller boxes within cuboids

Convolutional Neural Networks

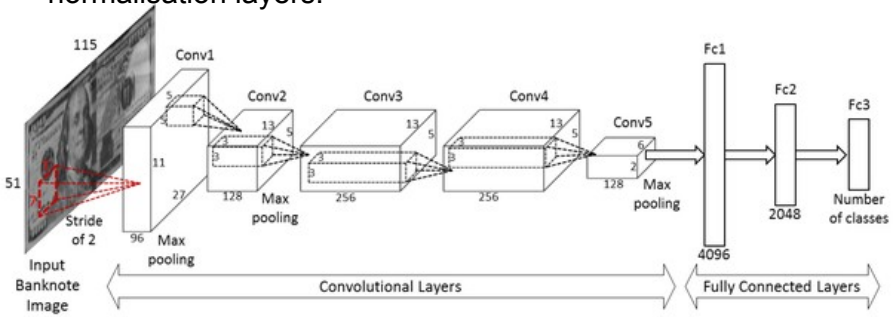
It is also possible for operations to be performed in parallel (as well as sequentially) e.g.:



Convolutional Neural Networks

Training

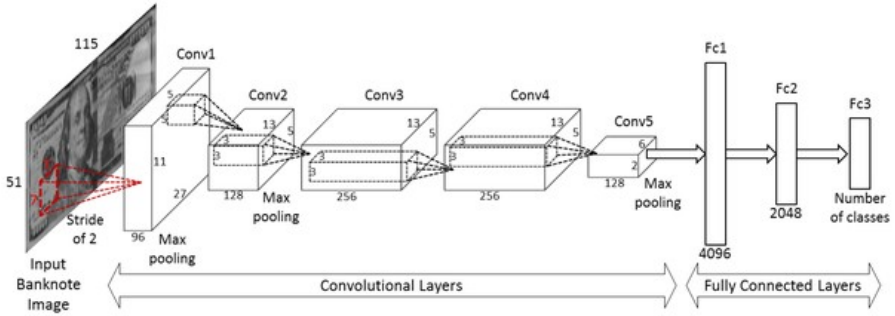
- Input presented and activation propagates through the network to the final layer.
- Output at final layer compared to expected output (provided by label associated with the input) to calculate error.
- Error backpropagated to adjust weights (in conv and fc layers), thresholds (in activation functions), parameters of normalisation layers.



Convolutional Neural Networks

Usage

- Input presented and activation propagates through the network to the final layer.
- Output at final layer determines predicted class of input.



Part f

Limitations of Deep NNs

Contents

Issues with Deep NNs

Deep NNs/CNNs have lots of tunable parameters:

- Typically 4 million to >140 million for classifying colour images
- There is a trend to increasing depth and complexity of deep NNs/CNNs

Consequently:

1. Need lots of training data
2. Training is computationally intensive
3. There is a danger of over-fitting the training data

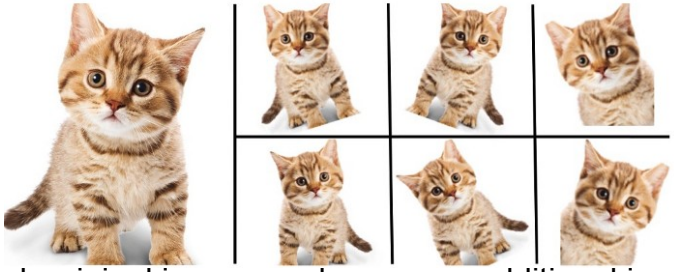
Issues with Deep NNs: Volume of Training Data

To address issue:

1. Need lots of training data

Can use data augmentation

e.g. for images apply class preserving transformations (shifts, rotations, changes to scale, flips, shears, crops, etc.)



for each original image produce many additional images

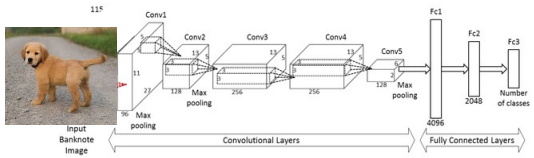
Issues with Deep NNs: Volume of Training Data

To address issue:

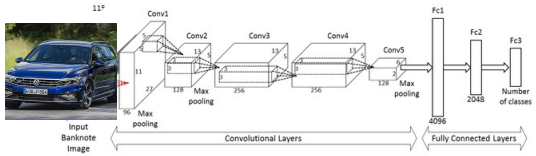
1. Need lots of training data
2. Training is computationally intensive

Can use transfer learning

train network (with initially random weights) on related task with lots of available data



train network (with pre-trained weights) on task with little data



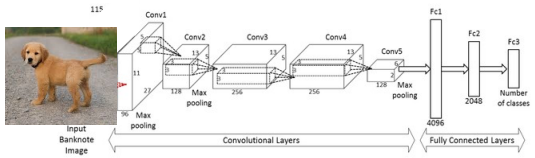
Issues with Deep NNs: Volume of Training Data

To address issue:

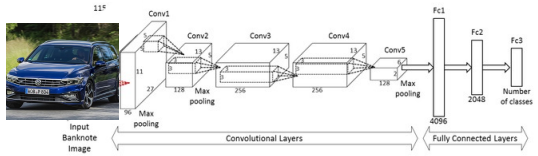
1. Need lots of training data
2. Training is computationally intensive

Can use transfer learning

copy pre-trained network from internet



train network (with pre-trained weights) on task for fewer epochs



Issues with Deep NNs: Overfitting

To address issue:

3. There is a danger of over-fitting the training data

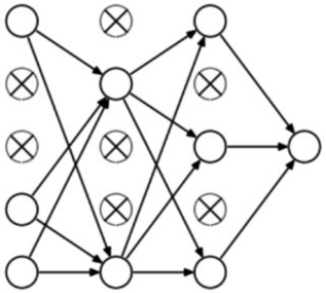
Can use regularization

a modification to the learning algorithm intended to improve generalisation

One of the most effective methods of regularization is dropout

- During training:

  at each iteration, randomly select a fraction of neurons and set their activity to be zero.



Issues with Deep NNs: Overfitting

To address issue:

3. There is a danger of over-fitting the training data

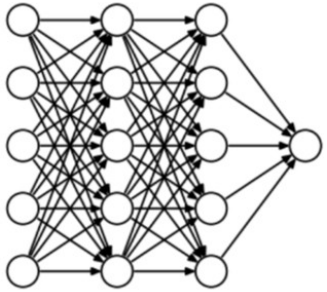
Can use regularization

a modification to the learning algorithm intended to improve generalisation

One of the most effective methods of regularization is dropout

- During usage:

  all neurons behave normally



Issues with Deep NNs: Overfitting

To address issue:

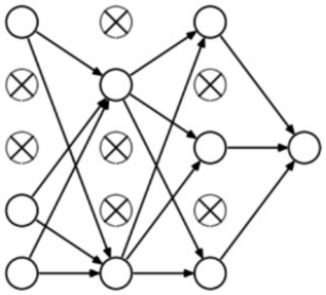
3. There is a danger of over-fitting the training data

Can use regularization

a modification to the learning algorithm intended to improve generalisation

One of the most effective methods of regularization is dropout

- dropout forces random sub-networks to correctly classify each sample, preventing recognition of individual samples being reliant on individual neurons.



Issues with Deep NNs: Overfitting

Even if all precautions are taken against over-fitting, the network may still learn to exploit unintended characteristics of the data, e.g.:

If a network is trained to distinguish object A from object B and all training images of object A are taken under, e.g.:

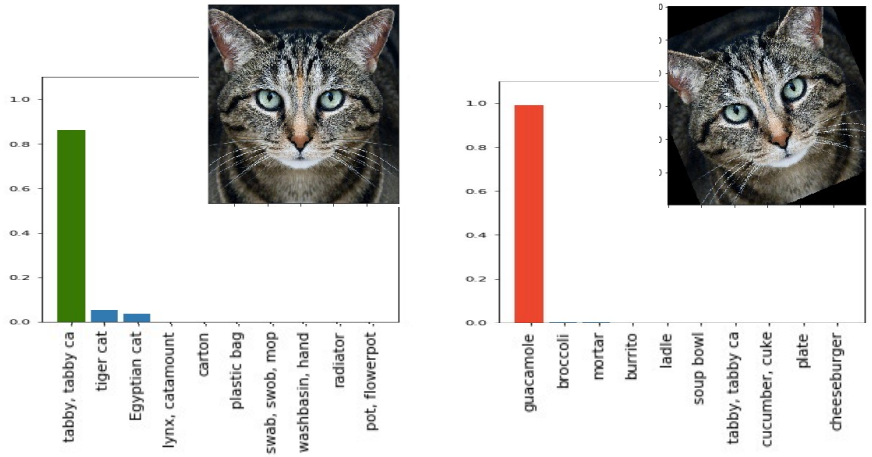
- certain lighting conditions
- with smudge on lens
- in a certain context
- with a camera with a dead pixel

then the neural network may learn to discriminate these classes using these irrelevant characteristics, and learn nothing about what the objects look like.

Issues with Deep NNs: Failure to Generalise

Deep NNs can produce excellent results on testing dataset, but still fail to generalise.

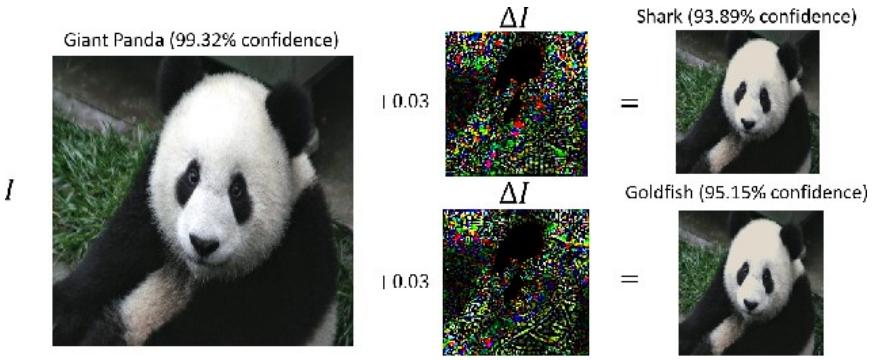
e.g. rotation leads to very different categorisations:



Issues with Deep NNs: Failure to Generalise

Deep NNs can produce excellent results on testing dataset, but still fail to generalise.

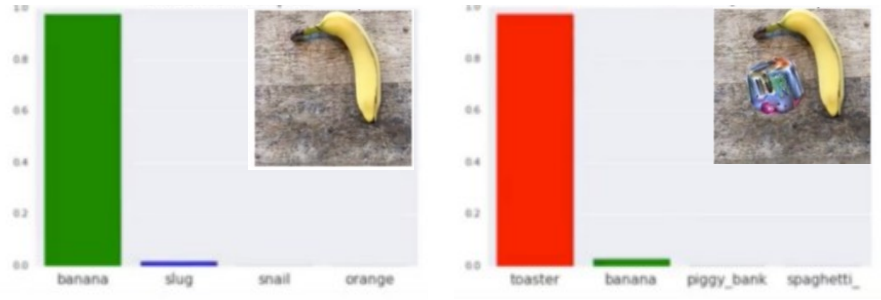
This is a particular problem as it means it is easy for someone to manipulate an image (into an adversarial example) so that it is wrongly classified, e.g. by making imperceivable changes to the pixel values:



Issues with Deep NNs: Failure to Generalise

Deep NNs can produce excellent results on testing dataset, but still fail to generalise.

This is a particular problem as it means it is easy for someone to manipulate an image (into an adversarial example) so that it is wrongly classified, e.g. by changing the context:



Summary

Deep Learning defines a set of “building-blocks” or “ingredients”:

- convolutional layers
- pooling layers
- fully-connected layers
- activation functions
- dropout
- batch normalisation

which can be combined together in many different ways to define many different neural network architectures.

Issues with Deep NNs: Failure to Generalise

Deep NNs can produce excellent results on testing dataset, but still fail to generalise.

This is a particular problem as it means it is easy for someone to manipulate the real world so that images are wrongly classified, e.g.:

Summary

Each building-block often has multiple hyper-parameters:

- number of masks, size of masks, padding, stride, dilation
- pooling type (max, mean), region size, stride
- number of layers and neurons
- ReLU, PReLU, tanh, sigmoid, etc.
- weight initialisation method
- regularisation method, dropout fraction
- data augmentation method
- learning algorithm (AdaGrad, RMSprop, ADAM, etc.)
- learning rate, parameters controlling momentum and adaption of learning rate

There are a huge number of possibilities.

Summary

These possibilities are normally explored by trial-and-error in order to create a network for a particular application.

There are excellent tools to help with the process of building and testing networks, e.g.:

- Torch / PyTorch (Facebook)
- TensorFlow (google)
- cognitive toolkit / CNTK (Microsoft)
- deep learning toolbox (MATLAB)
- caffe
- MXNet
- pyBrain
- KERAS (see instructions on KEATS)