Coursework 2

Coursework 2 - Overview and Objectives

- **Overview:** Use a selection of the techniques covered in the course so far to train accurate multi-layer networks for MNIST classification
- **Objective:** Assess your ability to design, implement and run a set of experiments to answer specific research questions about the models and methods covered in MLP
- Choose three topics – one simpler, two more complex
  - Simpler topics include exploration of: early stopping; L1 vs L2 regularization; number of layers; hidden unit transfer functions; preprocessing of input data
  - More complex topics: data augmentation;. convoltional layers; skip connections / ResNets; Batch normalisation; ...

Coursework 2 - What to submit

- Submit a report (PDF), your notebook, and python code.
- Primarily assessed on the report – For each topic:
  - Clear statement of the research question investigated;
  - Clear description of methods and algorithms;
  - Motivation for each experiment completed;
  - Quantitative results including relevant graphs;
  - Discussion of your results and any conclusions you have drawn.
- Please
  - **Do** submit everything online using `submit`
  - **Don't** submit on paper to the ITO
  - **Don't** submit everything in your mlpractical directory
  - **Do start running the experiments for this coursework as early as possible** – Some of the experiments may take significant compute time

Can we design a network that takes account of the image structure?

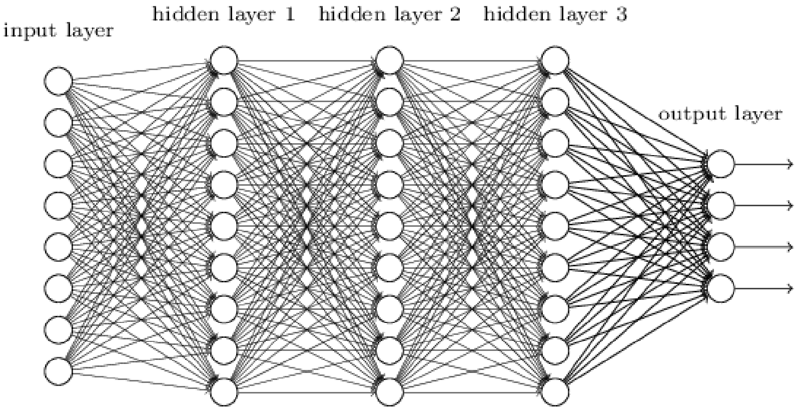
(And learns invariances...)

Convolutional Networks

Steve Renals

Machine Learning Practical — MLP Lecture 7
2 November 2016

Recap: Multi-layer network for MNIST



(image from: Michael Nielsen, *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/chap6.html)

How can we make this better?

On MNIST, we can get about 2% error (or even better) using these kind of networks, but

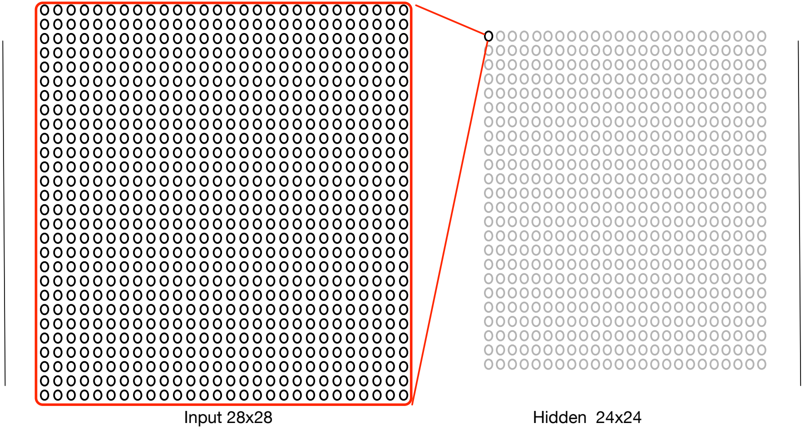
- They ignore the spatial (2-D) structure of the input images – unroll each 28x28 image into a 784-D vector
- Each hidden unit looks at the units in the layer below, so pixels that are spatially separate are treated the same way as pixels that are adjacent
- There is no obvious way for networks to learn the same features (e.g. edges) at different places in the input image

Convolutional networks

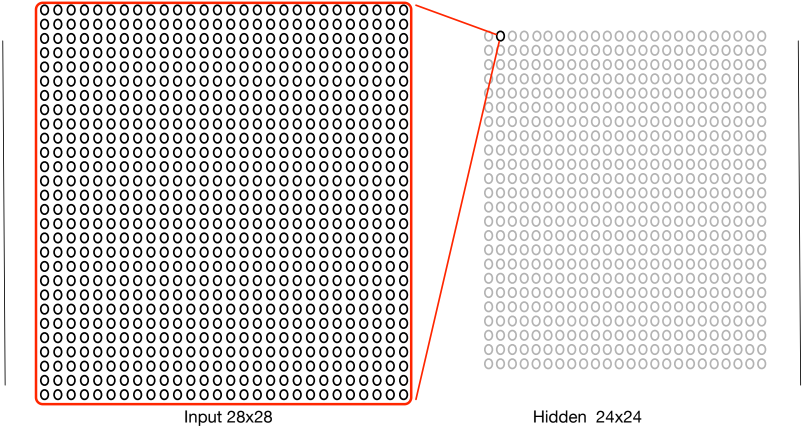
Convolutional networks address these issues through

- **Local receptive fields** in which hidden units are connected to local patches of the layer below,
- **Weight sharing** which enables the construction of feature maps,
- **Pooling** which condenses information from the previous layer.

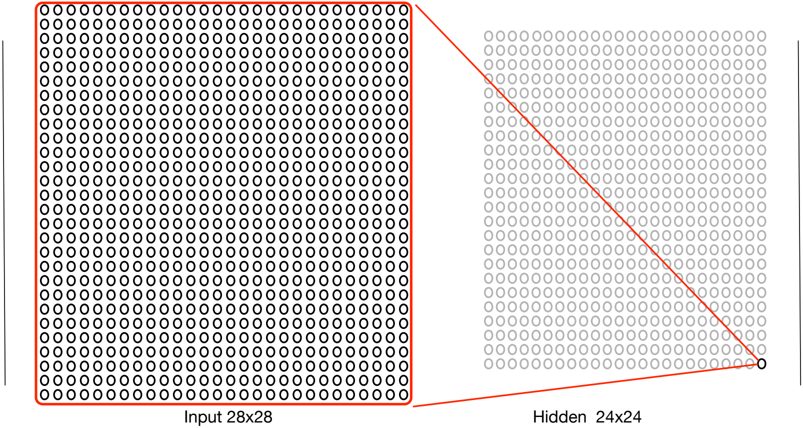
Fully connected hidden layer – 576 hidden units



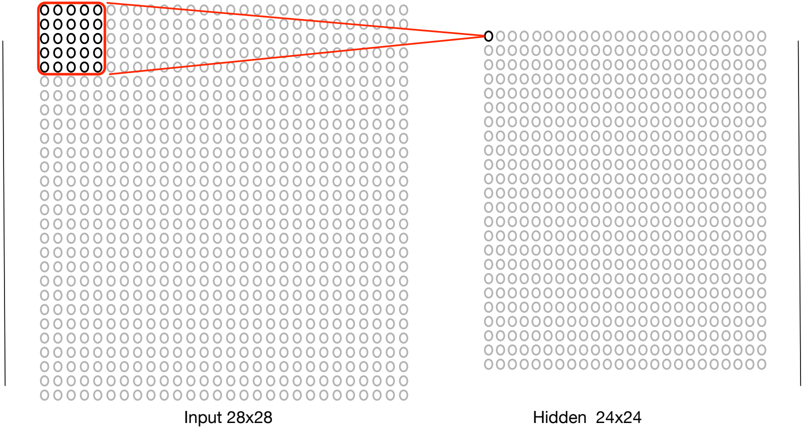
Fully connected hidden layer – 576 hidden units



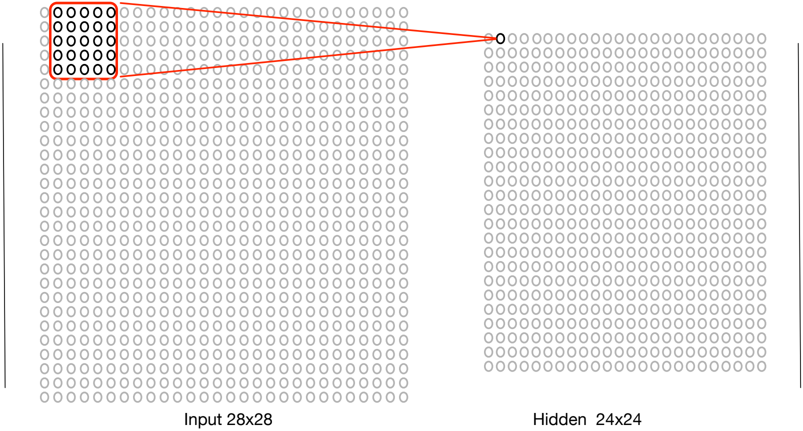
Fully connected hidden layer – 576 hidden units



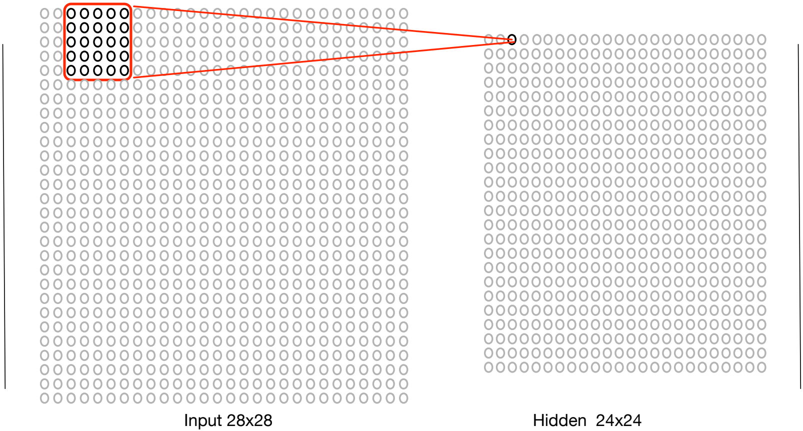
Local receptive fields – 24x24 hidden units



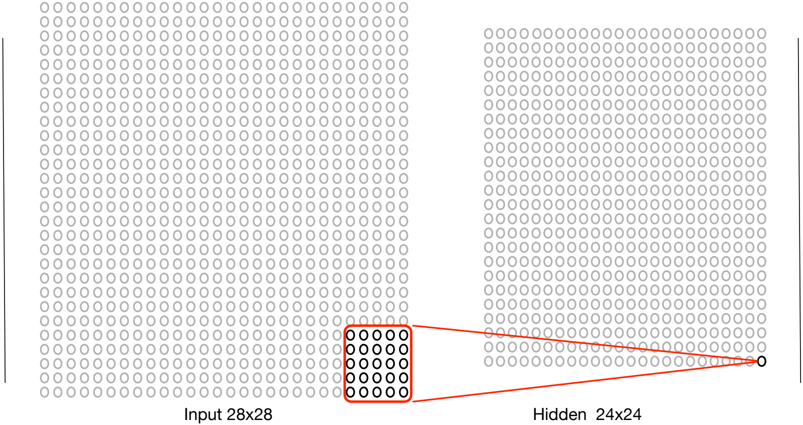
Local receptive fields – 24x24 hidden units



Local receptive fields – 24x24 hidden units



Local receptive fields – 24x24 hidden units



Local receptive fields

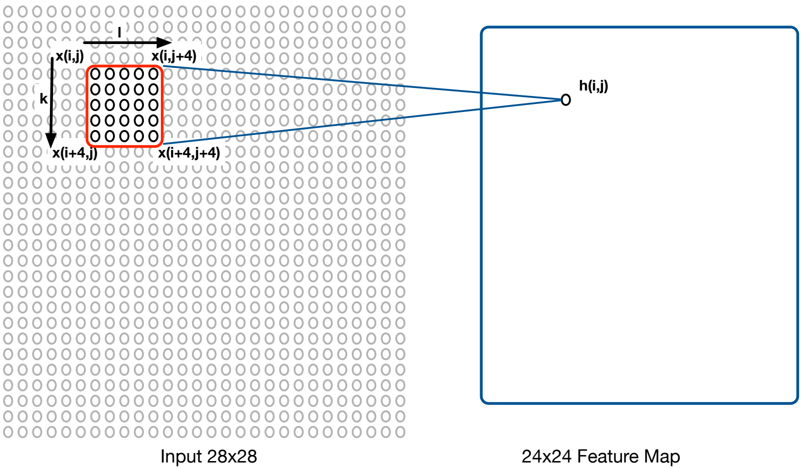
- Each hidden unit is connected to a small $(m \times m)$ region of the input space – the *local receptive field*
- If we have a $d \times d$ input space, then we have $(d - m + 1) \times (d - m + 1)$ hidden unit space
- Each hidden unit extracts a feature from "its" region of input space
- Here the receptive field "stride length" is 1, it could be larger

Shared weights

- Constrain each hidden unit $h_{i,j}$ to extract the same feature by sharing weights across the receptive fields
- For hidden unit $h_{i,j}$

$$h_{i,j} = \text{sigmoid}(\sum_{k=0}^{m-1} \sum_{\ell=0}^{m-1} w_{k,\ell} x_{i+k,j+\ell} + b)$$

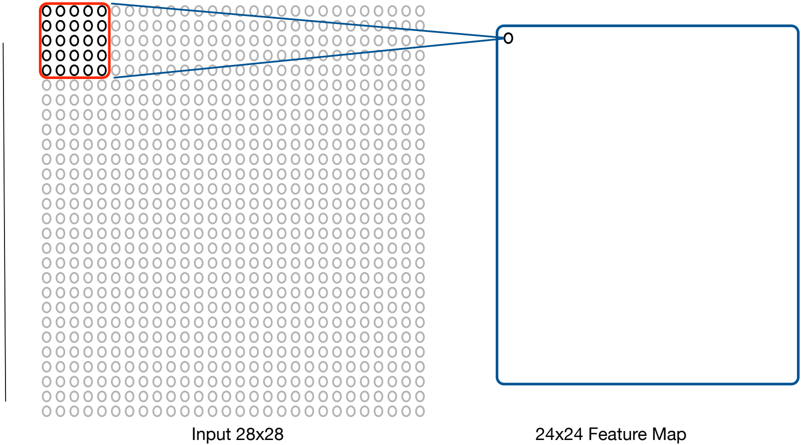
  where $w_{k,\ell}$ are elements of the shared $m \times m$ weight matrix $\mathbf{w}$, b is the shared bias, and $x_{i+k,j+\ell}$ is the input at $i+k, j+\ell$
- We use k and l to index into the receptive field, whose top left corner is at $x_{i,j}$


l
x(i,j)
x(i,j+4)
k
x(i+4,j)
x(i+4,j+4)
h(i,j)
Input 28x28
24x24 Feature Map

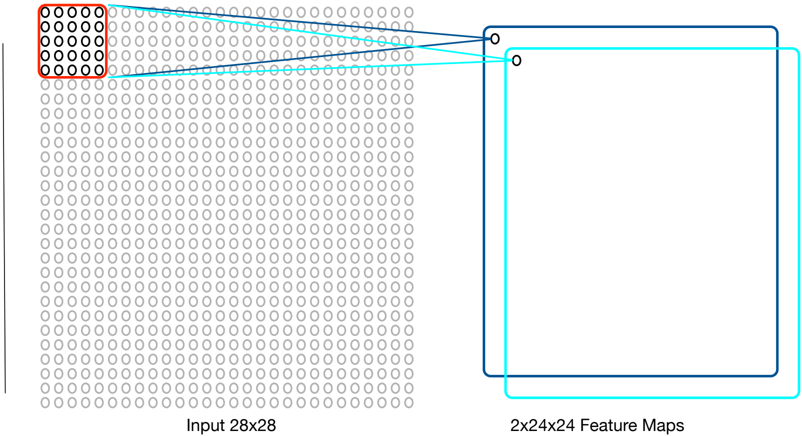
Feature Maps

- Local receptive fields with shared weights result in a **feature map**
  - a map showing where the feature corresponding to the shared weight matrix (**kernel**) occurs in the image
- Feature map encodes **translation invariance**
  - extract the same features irrespective of where an image is located in the input
- **Multiple feature maps**
  - a hidden layer can consist of F different feature maps – in this case $F \times 24 * 24$ units in total

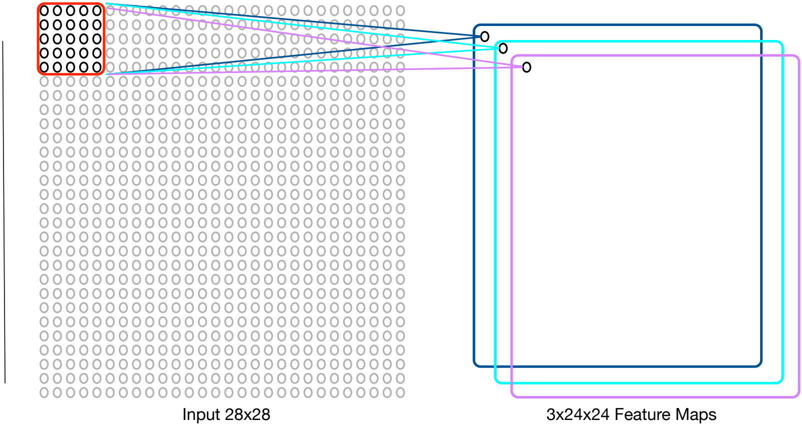
Feature Maps



Feature Maps



Feature Maps



Weights and Connections

Consider an MNIST hidden layer with feature maps using a 5x5 kernels (resulting in 24x24 feature maps):

- Number of connections per feature map:
  $24 \times 24 \times 5 \times 5 = 14,400$ connections
  $24 \times 24 = 576$ biases
- But since weights are shared within a feature map, we have
  $5 \times 5 = 25$ weights
  1 bias

Consider the case where we have 40 feature maps. We will have

- 1,000 (25×40) weights (+ 40 biases)
- but 576,000 (+ 23,040) connections

In comparison a 100 hidden unit MLP from the first coursework has $784 \times 100 + 100 = 78,500$ input-hidden weights

Learning image kernels

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

https://en.wikipedia.org/wiki/Kernel_(image_processing)

- Image kernels have been designed and used for feature extraction in image processing (e.g. edge detection)
- However, we can learn multiple kernel functions (feature maps) by optimising the network cost function
- Automating feature engineering

Convolutional Layer

- This type of feature map is often called a **Convolutional layer**
- We can write the feature map hidden unit equation:

$$h_{i,j} = \text{sigmoid}(\sum_{k=1}^{m}\sum_{\ell=1}^{m} w_{k,\ell} x_{i+k,j+\ell} + b)$$
$$h = \text{sigmoid}(\mathbf{w} \otimes \mathbf{x} + b)$$

  $\otimes$ is a cross-correlation and is closely related to a *convolution*
- In signal processing a 2D convolution is written as

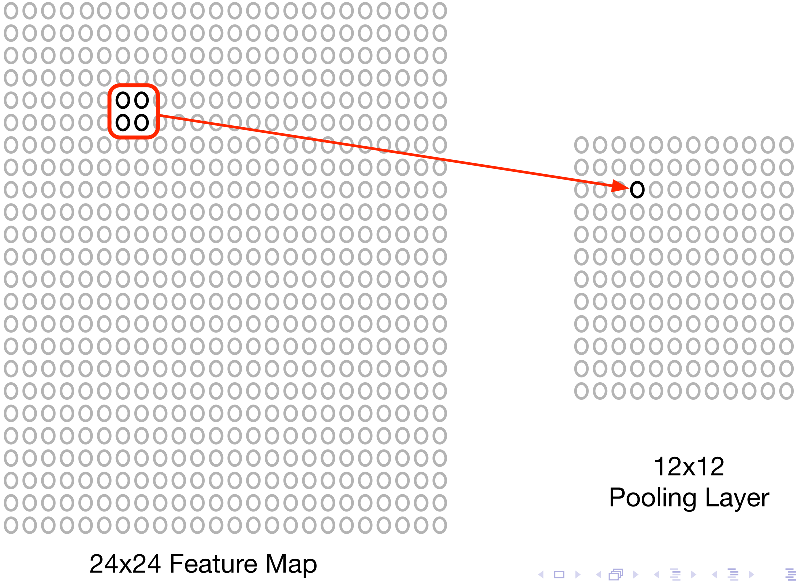
$$H_{i,j} = \text{sigmoid}(\sum_{k=1}^{m}\sum_{\ell=1}^{m} v_{k,\ell} x_{i-k,j-\ell} + b)$$
$$H = \text{sigmoid}(\mathbf{v} * \mathbf{x} + b)$$

- If we "flip" (reflect horizontally and vertically) **w** (cross-correlation) then we obtain **v** (convolution)

Convolution vs Cross-correlation

- Cross-correlation is often referred to as convolution in deep learning....
- This is not problematic since the specific properties of convolution but not of cross-correlation (commutativity and associativity) are rarely (if ever) required for deep learning
- In machine learning the network learns the kernel appropriate to its orientation – so if convolution is implemented with a flipped kernel, it will learn that it is a flipped implementation
- So it is OK to use an efficient (flipped) implementation of convolution for convolutional layers

Pooling (subsampling)



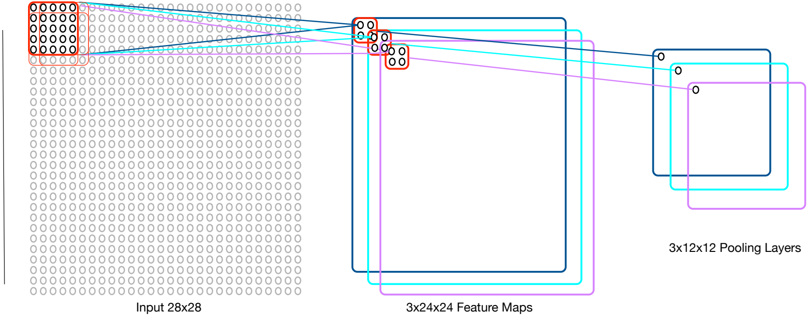
Pooling

- Pooling or subsampling takes a feature map and reduces it in size – e.g. by transforming a set of 2x2 regions to a single unit
- Pooling functions
  - Max-pooling – takes the maximum value of the units in the region (c.f. maxout)
  - L_p-pooling – take the L_p norm of the units in the region:

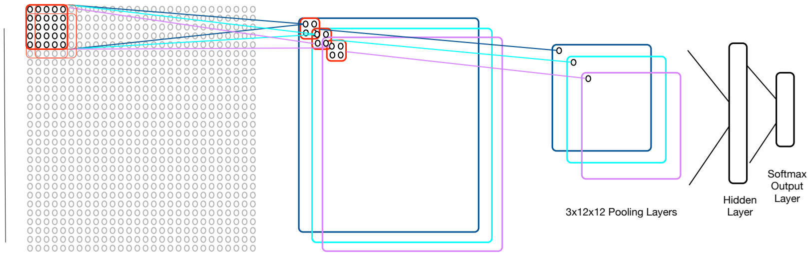
$$h' = \left(\sum_{i \in \text{region}} h_i^p \right)^{1/p}$$

  - Average- / Sum-pooling – takes the average / sum value of the pool
- Information reduction – pooling removes precise location information for a feature
- Apply pooling to each feature map separately

Putting it together – convolutional+pooling layer



ConvNet – Convolutional Network



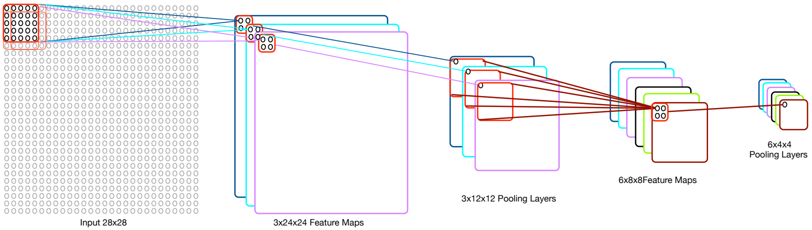
Simple ConvNet:

- Convolutional layer with max-pooling
- Final fully connected hidden layer (no sharing weight)
- Softmax output layer
- With 20 feature maps and a final hidden layer of 100 hidden unit:
  $20 \times (5 \times 5 + 1) + 20 \times 12 \times 12 \times 100 + 100 + 100 \times 10 + 10 = 289,630$ weights

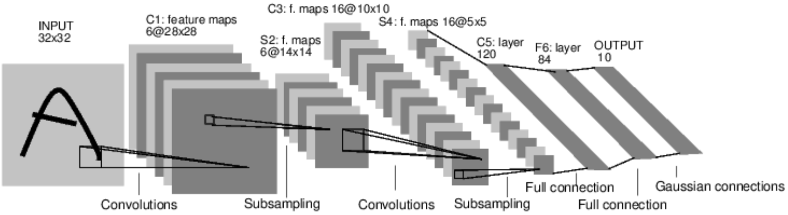
Multiple input images

- If we have a colour image, each pixel is defined by 3 RGB values – so our input is in fact 3 images (one R, one G, and one B)
- If we want stack convolutional layers, then the second layer needs to take input from all the feature maps in the first layer
- **Local receptive fields across multiple input images**
- In a second convolutional layer (C2) on top of 20 12×12 feature maps, each unit will look at $20 \times 5 \times 5$ input units(combining 20 receptive fields each in the same spatial location)
- Typically do not tie weights across feature maps, so each unit in C2 has $20 \times 5 \times 5 = 500$ weights, plus a bias. (Assuming a 5×5 kernel size)

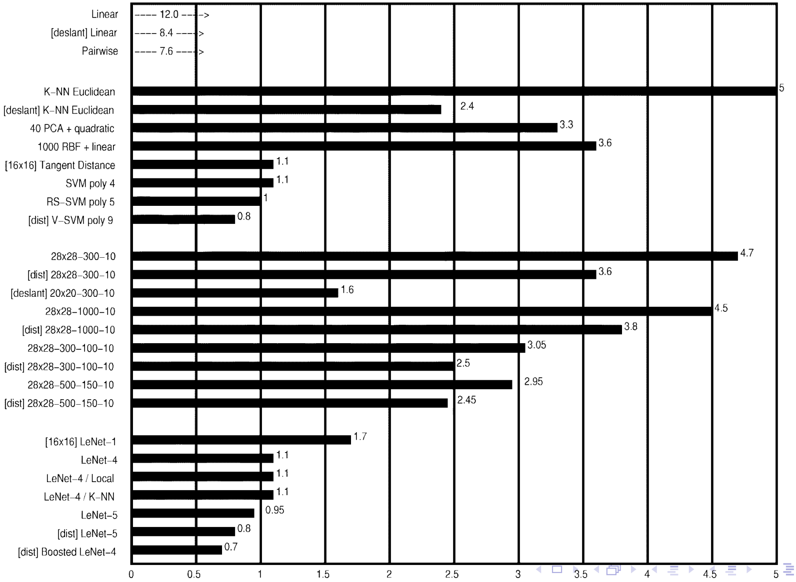
Stacking convolutional layers



Example: LeNet5 (LeCun et al, 1997)



MNIST Results (1997)



Training Convolutional Networks

- Train convolutional networks with a straightforward but careful application of backprop / SGD
- Exercise: prior to the next lecture, write down the gradients for the weights and biases of the feature maps in a convolutional network. Remember to take account of weight sharing.
- Next lecture: implementing convolutional networks: how to deal with local receptive fields and tied weights, computing the required gradients...

Summary

- Convolutional networks include local receptive fields, weight sharing, and pooling leading to:
  - Modelling the spatial structure
  - Translation invariance
  - Local feature detection
- Reading:

  Michael Nielsen, *Neural Networks and Deep Learning* (ch 6)
  http://neuralnetworksanddeeplearning.com/chap6.html
  Yann LeCun et al, "Gradient-Based Learning Applied to Document Recognition", *Proc IEEE*, 1998.
  http://dx.doi.org/10.1109/5.726791
  Ian Goodfellow, Yoshua Bengio & Aaron Courville, *Deep Learning* (ch 9)
  http://www.deeplearningbook.org/contents/convnets.html