Convolutional Neural Nets: Key Concepts, ImageNet & AlexNet

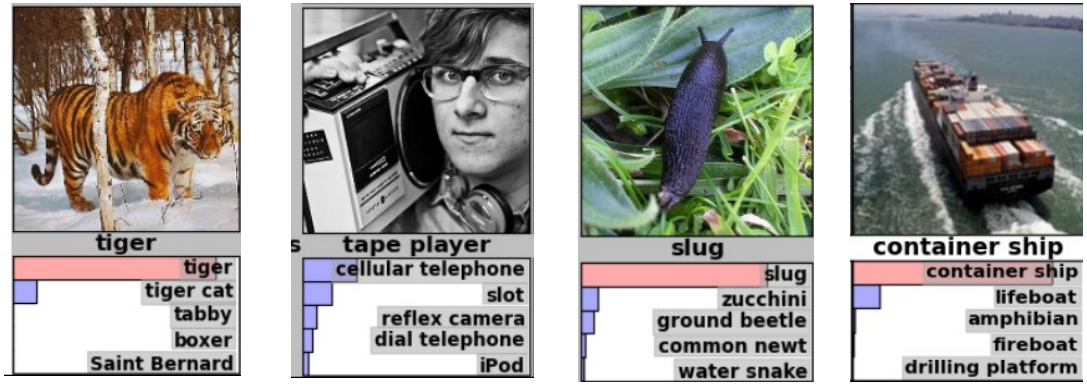
(Source: Taken many slides from Oliver Durr, Boris Ginzburg, Geoffrey Hinton)

CNN History

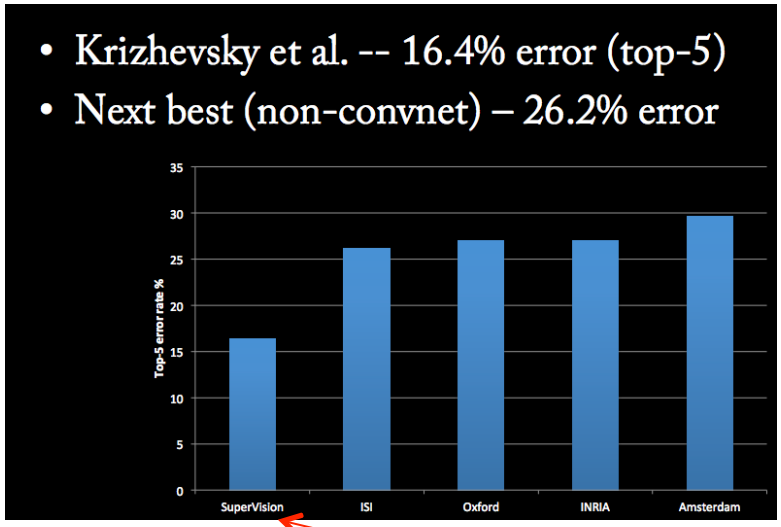
- 1980 Kunihiko Fukushima (Neocognitron)
- 1998 Le Cun (LeNet5, Backpropagation)
- Wining pattern recognition contest
  - 2011& 2014 MINST Handwritten Dataset
  - 201X Chinese Handwritten Character
  - 2011 German Traffic Signs
- ImageNet
  - Alex Net (2012) winning solution of ImageNet benchmarking

ImageNet 2012, 2013, 2014

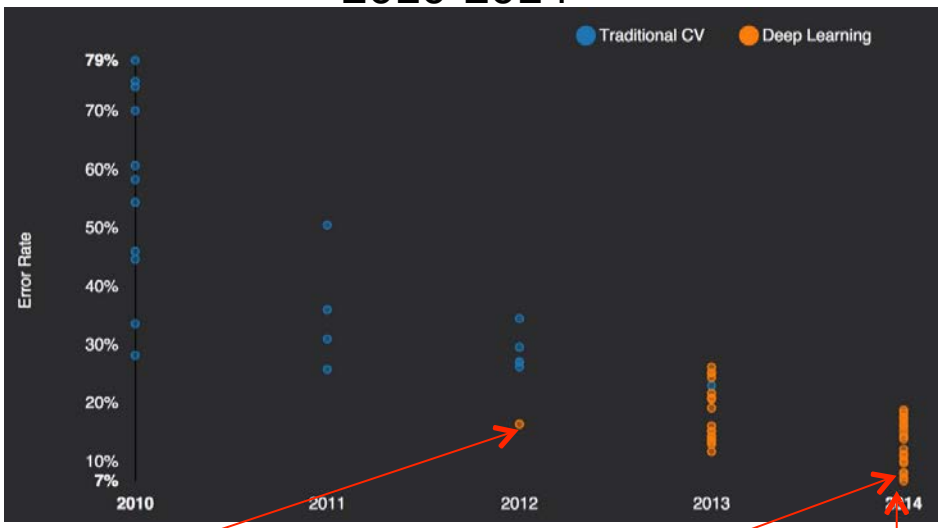
Examples of AlexNet results on ImageNet (1000 Classes, 1.2M images)



2012



2010-2014



AlexNet 7 layers

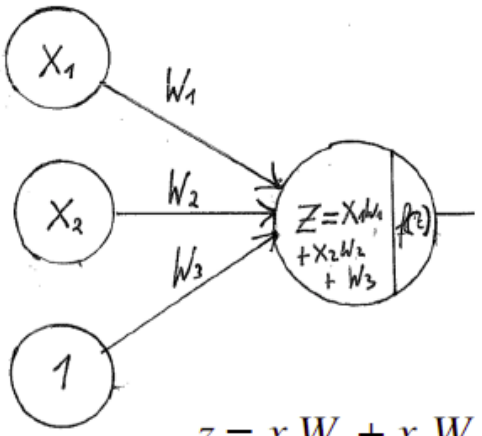
GoogLeNet 22 layers 6.7%

VGG net up to 19 layers

What is an Artificial Neural Network

An Artificial Neuron: The Basic Unit

Activation Function:

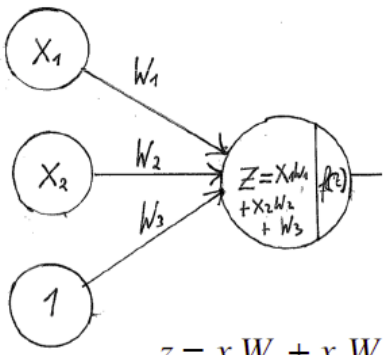


$$f(z) = \begin{cases} \dfrac{\exp(z)}{1+\exp(z)} \\ \max(0,z) \end{cases}$$

$$z = x_1 W_1 + x_2 W_2 + W_3 = \theta^T x$$

An Artificial Neuron: The Basic Unit

Activation Function:

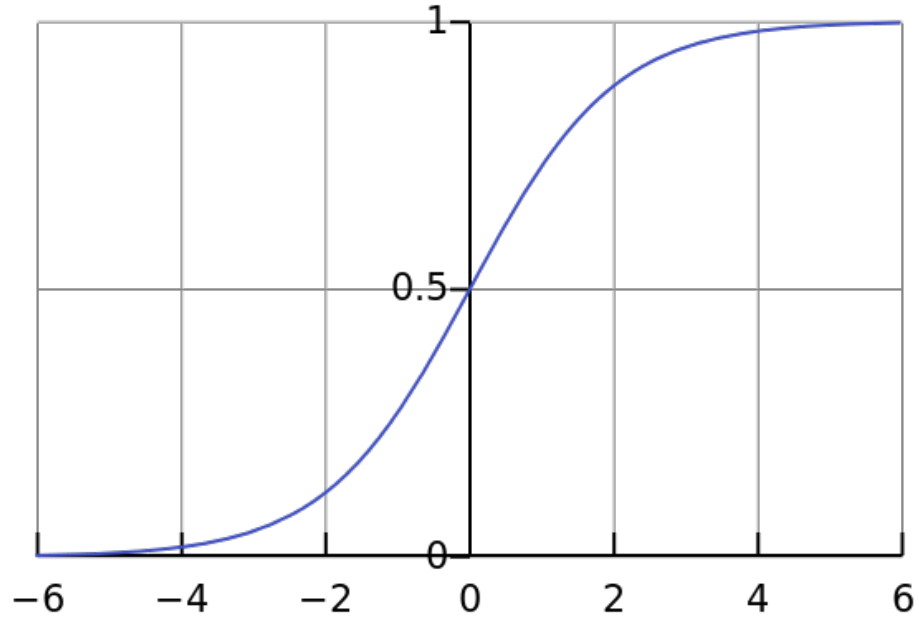


$$f(z) = \begin{cases} \dfrac{\exp(z)}{1+\exp(z)} \\ \max(0,z) \end{cases}$$

$$z = x_1 W_1 + x_2 W_2 + W_3 = \theta^T x$$

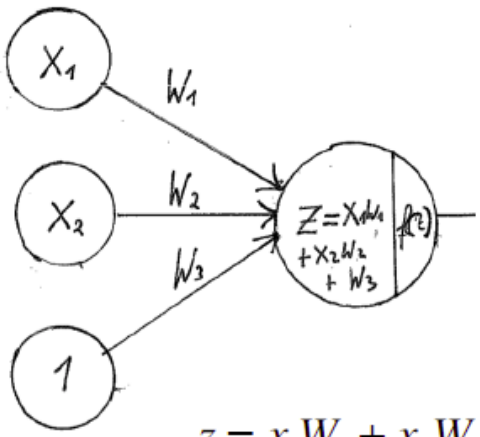
Logistic Regression (Sigmoid Function)

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



An Artificial Neuron: The Basic Unit

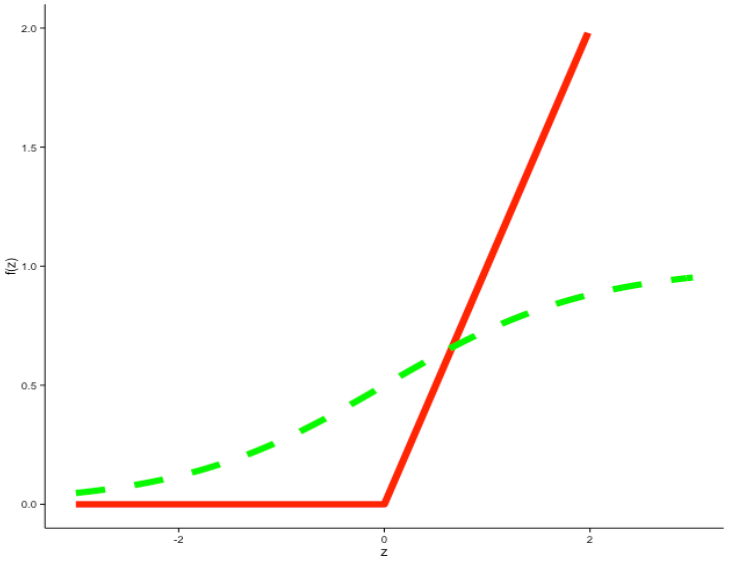
Activation Function:



$$f(z) = \begin{cases} \dfrac{\exp(z)}{1+\exp(z)} \\ \max(0,z) \end{cases}$$

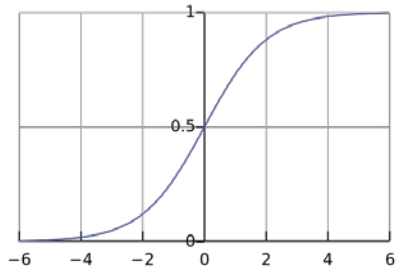
$$z = x_1 W_1 + x_2 W_2 + W_3 = \theta^T x$$

Activation Function Nonlinearity *f(z)*



Logistic Regression (Sigmoid Function)

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



Motivation:

Red: ReLU (Rectified Linear Unit)

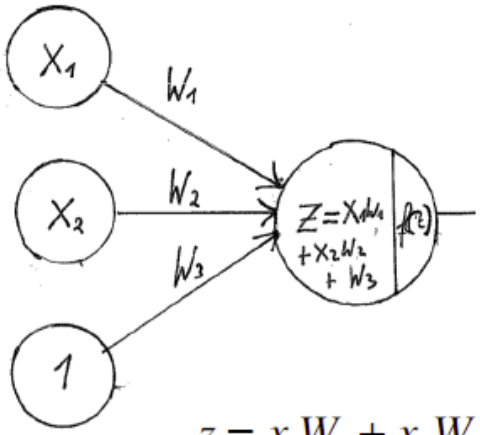
6 times faster convergence

Green: Logistic Regression

The gradient of Sigmoid close to zero (flat), slow convergence

An Artificial Neuron: The Basic Unit

Activation Function:

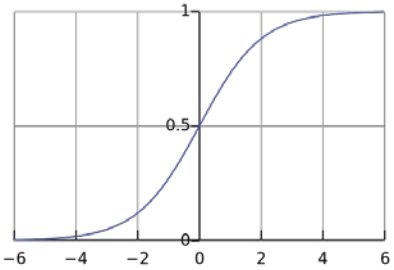


$$f(z) = \begin{cases} \dfrac{\exp(z)}{1+\exp(z)} \\ \max(0,z) \end{cases}$$

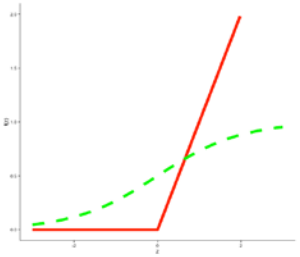
$$z = x_1 W_1 + x_2 W_2 + W_3 = \theta^T x$$

Logistic Regression (Sigmoid Function)

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



Activation Function Nonlinearity *f(z)*



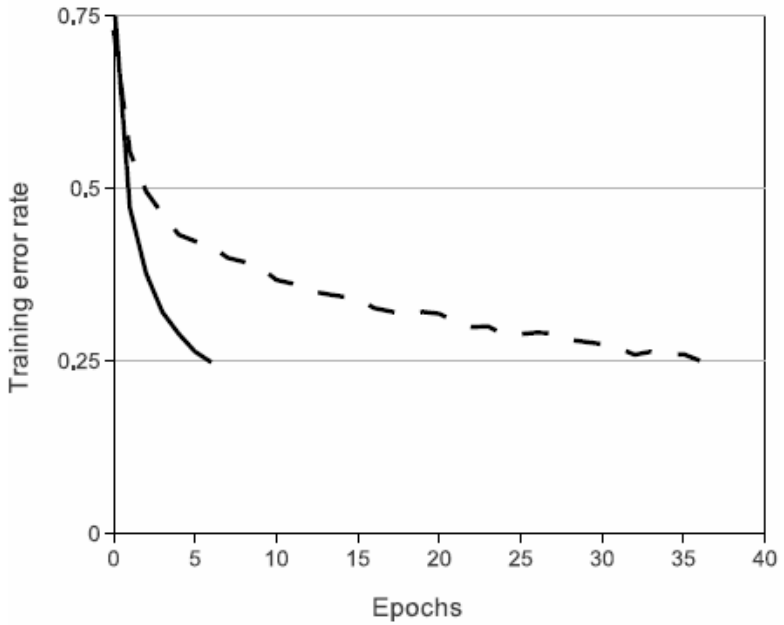
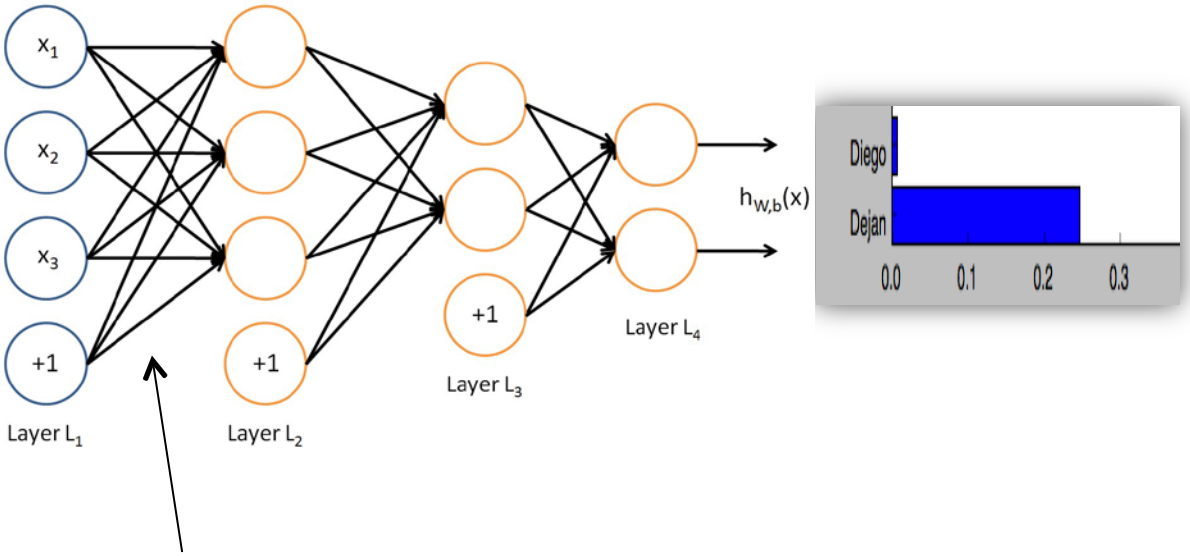


Figure 1: A four-layer convolutional neural network with ReLUs (**solid line**) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

(Krizhevsky et al 2012)

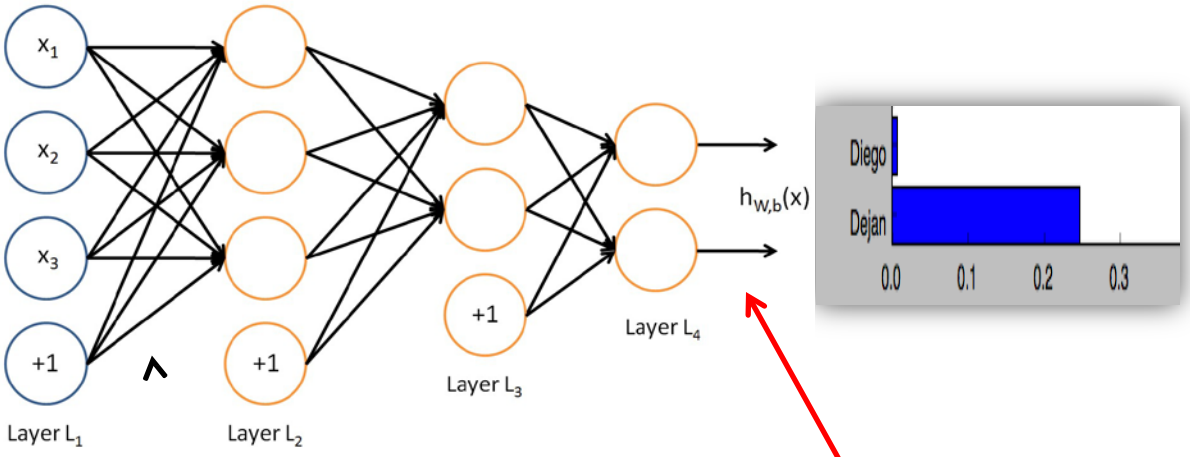
An Artificial Neural Network



Contains many weights $W^{(l)}_{ij}$

A complex function of the many weights $\theta = W^{l}_{ij}$ and the input images, the output predicting the probability of a class label

Model output – the Softmax function



Output (Softmax)

$$f(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{N} e^{z_i}}$$

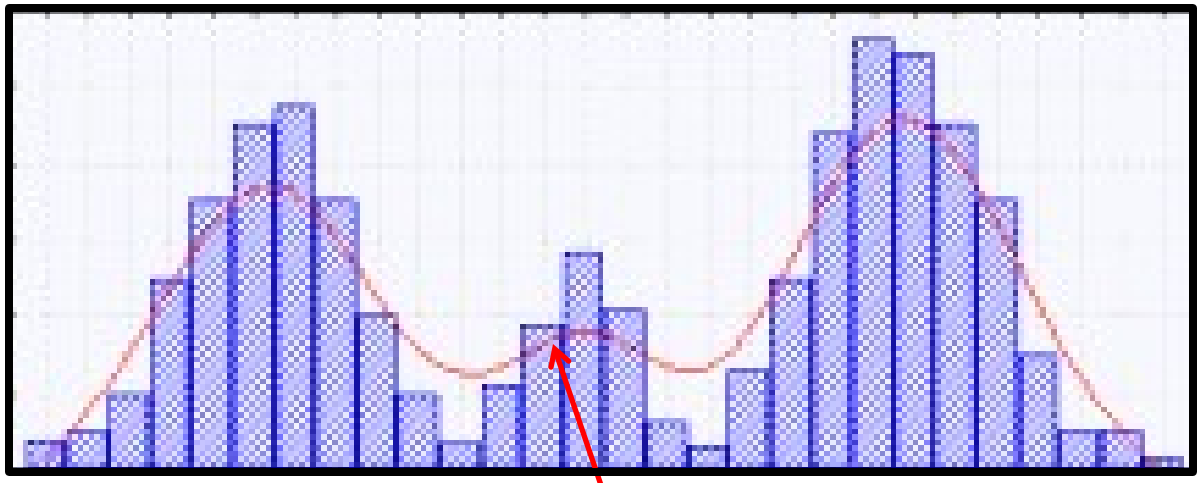
Propability of a class given the input vector **z**

Softmax function (normalized exponential) – multinomial logistic regression:

$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

where *y* are class labels and **x** are input vectors (not the images)

Model output – the Softmax function



Output (Softmax)

$$f(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{N} e^{z_i}}$$

Propability of a class given the input vector **z**

Softmax function (normalized exponential) – multinomial logistic regression:

Sum to 1

$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

where *y* are class labels and **x** are input vectors (not the images)

Model training – the loss function

- Use training data j=1,...,N_{train} to optimize loss function *J* , sensitive to error
- Use a subset n (minibatch) of the training data for optimization, and random repeats (dropout & data augmentation)

$$-nJ(\theta) = \sum_{i=1}^{n\ \text{(Mini Batch Size)}} \text{Cost (loss) of Training example } X_i$$

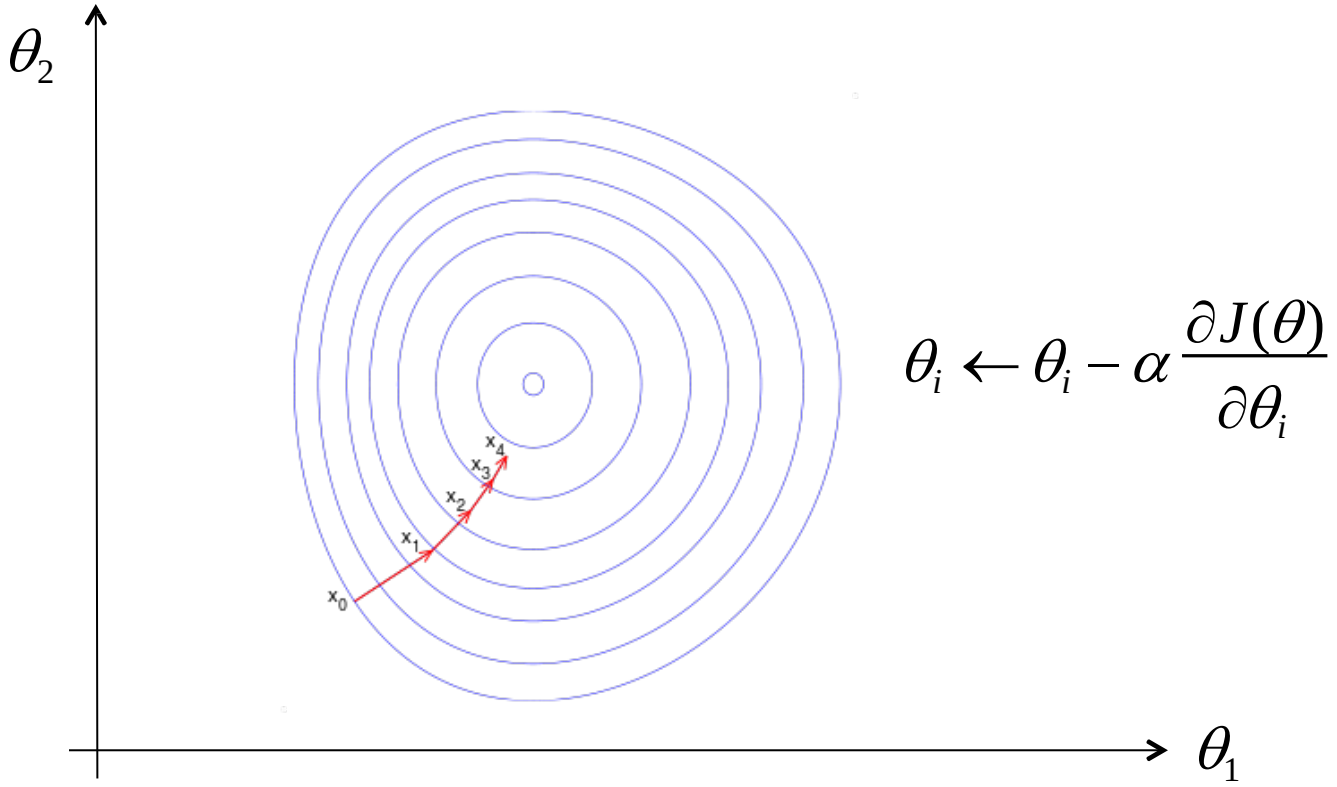
- **How to learn a model: Optimise loss function** – minimise the negative log likelihood, by either maximum likelihood estimation (MLE), or maximum a posteriori probability (MAP) given (assuming) uniform prior

- Learning objective: Optimal weights from many **iterations (epochs)** using gradient descent (controlled by ***α* learning rate**)

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

- Back-propagation (the chain rule) is used to calculate the gradient

Gradient Descent (in a parameter space)



$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J(\theta)}{\partial \theta_i}$$

Weight parameter space: showing θ_1,θ_2 (just two weights from millions)

- Gradient descent (first order)
- Newton Taylor expansion 2nd order using Hessian

Gradient Descent: Momentum

- Problem with (Stochastic) Gradient Descent are Valleys – local minimum. The loss function bounces up & down the walls, not descenting the slope.
  - Nesterov Accelerated Gradient (NAG) – Added **momentum** μV_t for faster descent

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$
$$W_{t+1} = W_t + V_{t+1}$$

(SGD)

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$
$$W_{t+1} = W_t + V_{t+1}$$

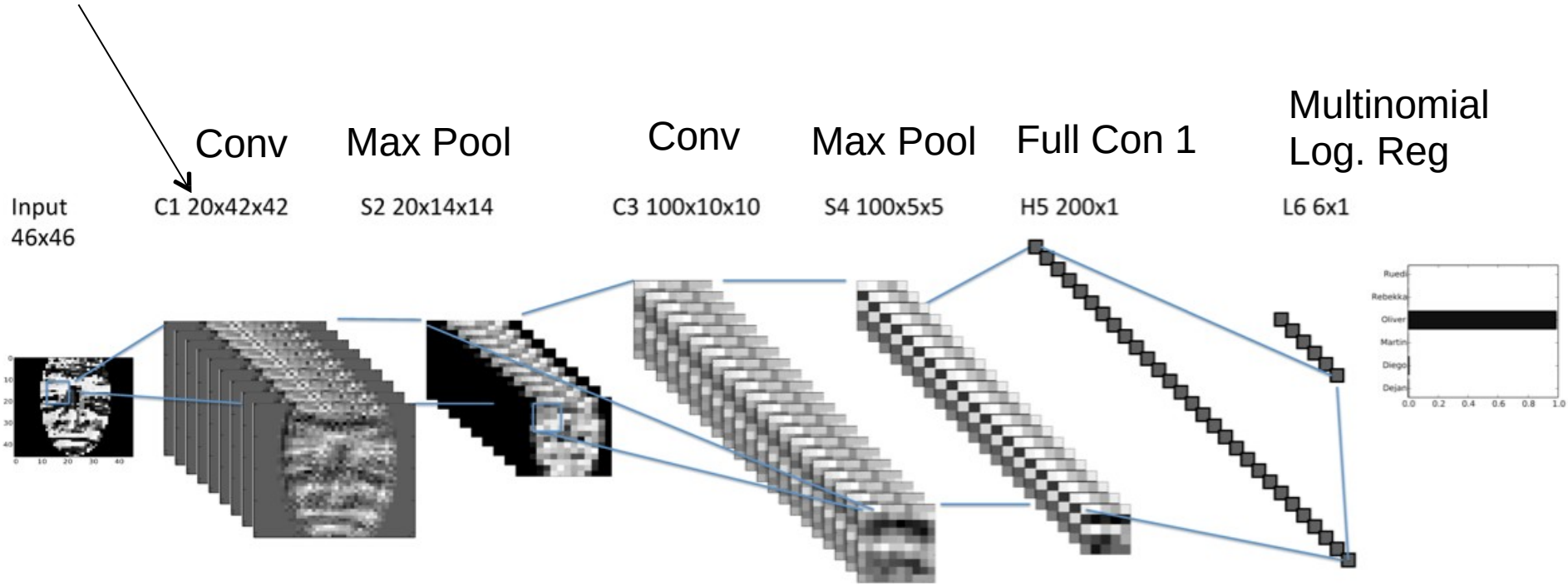
(NAG)

- Set mu=0, NAG becomes SGD (Stochastic Gradient Descent)

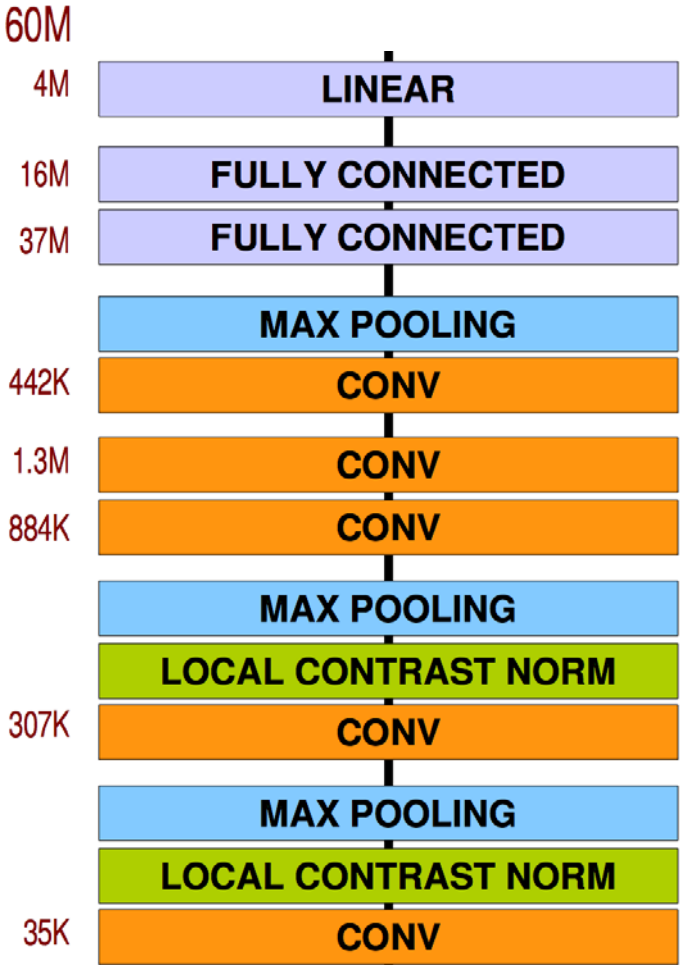
- Caffe Tutorial:
  - http://caffe.berkeleyvision.org/tutorial/solver.html
- Visualization:
  - http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect- facial-keypoints-tutorial/

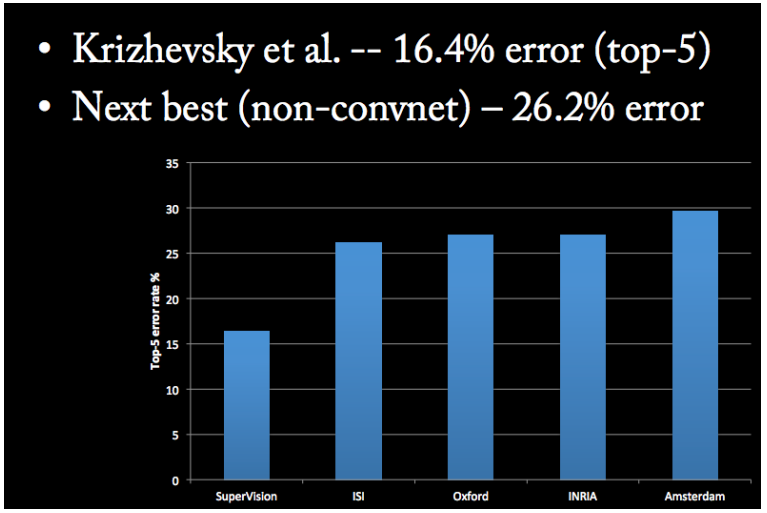
An early CNN (LeNet5, 1998)

20 Kernels a 5x5 weights to go from one to the next



A recent CNN (AlexNet, 2012) – A Game Changer



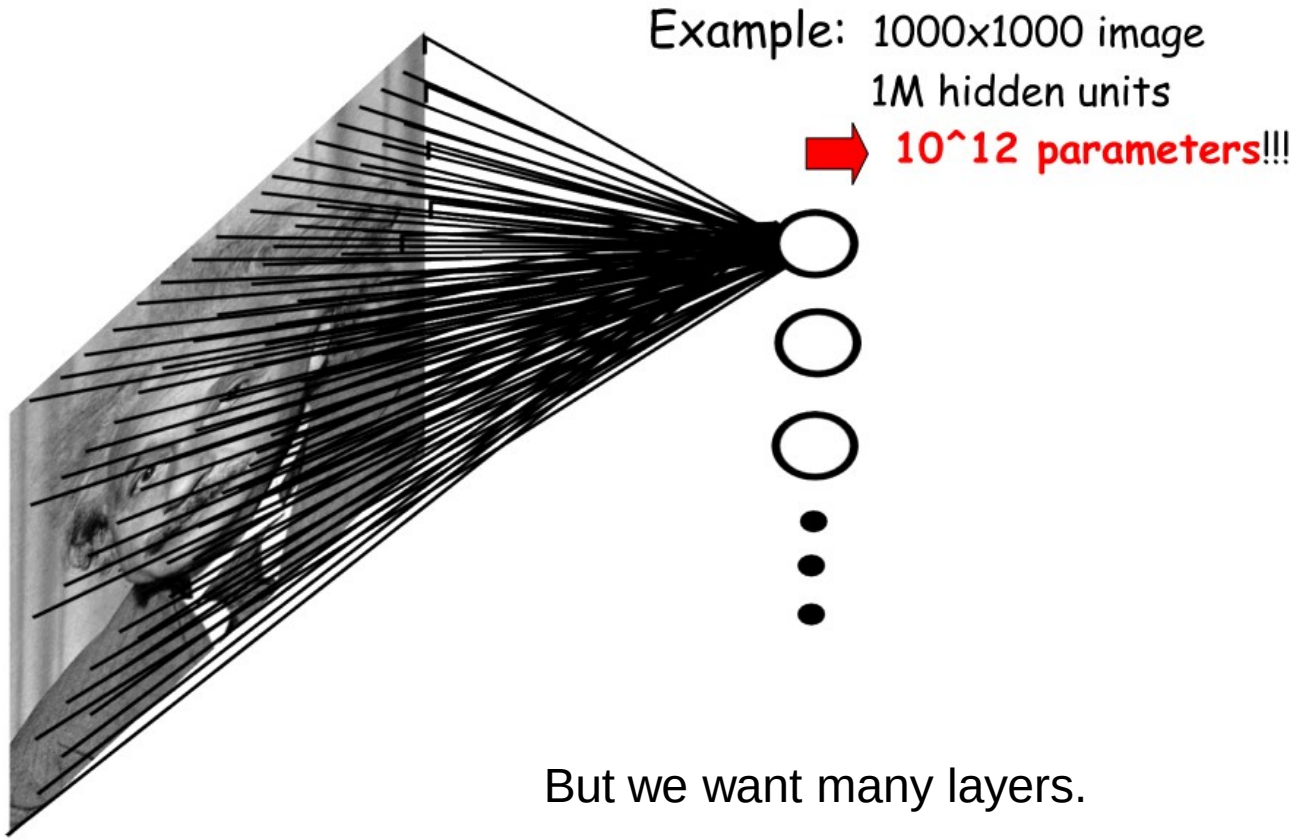


Large Scale Visual Recognition Challenge (ILSVRC-2012)
ImageNet Competition 26.2% error ↠ 16.5%, significant.

Key:

- Dropout
- ReLU instead of sigmoid (logistic regression)
- Two GPUs implementation (less learning time)
- Local Response Normalization

Too many weights (parameters)

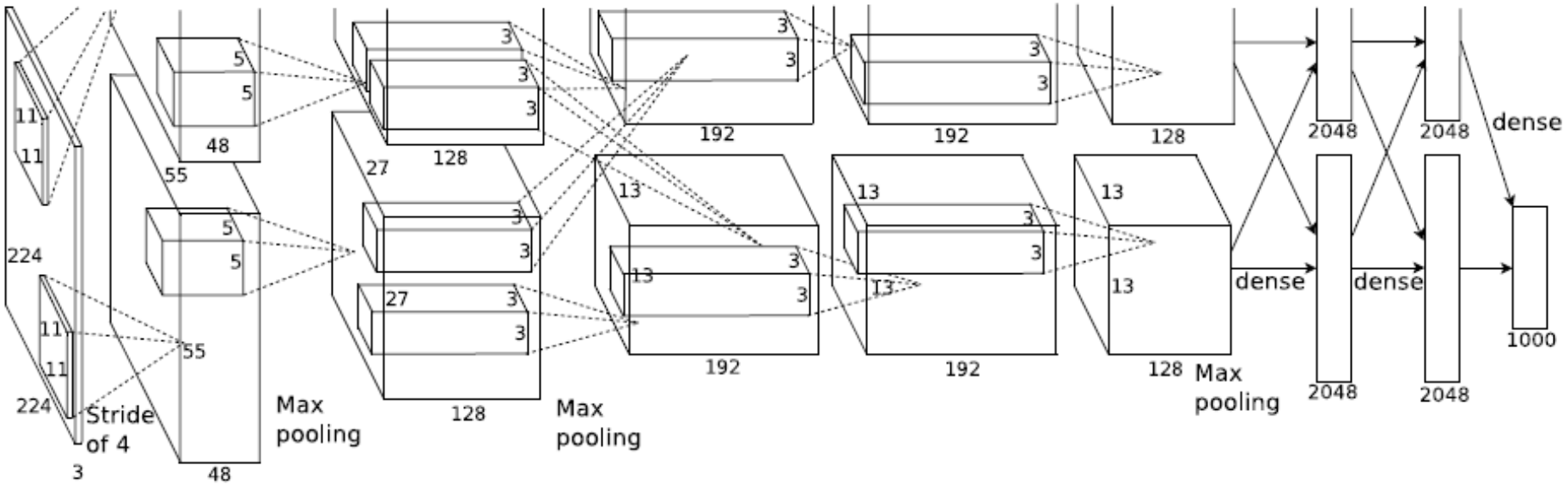


But we want many layers.

Remedy:

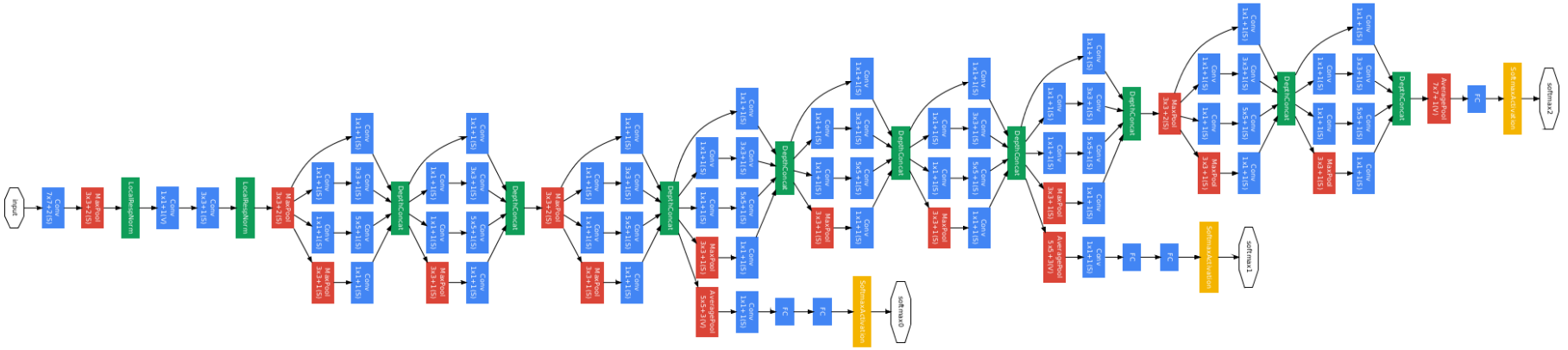
- Weight sharing ↠ Convolution
- Sparse connectivity ↠ Pooling

AlexNet design

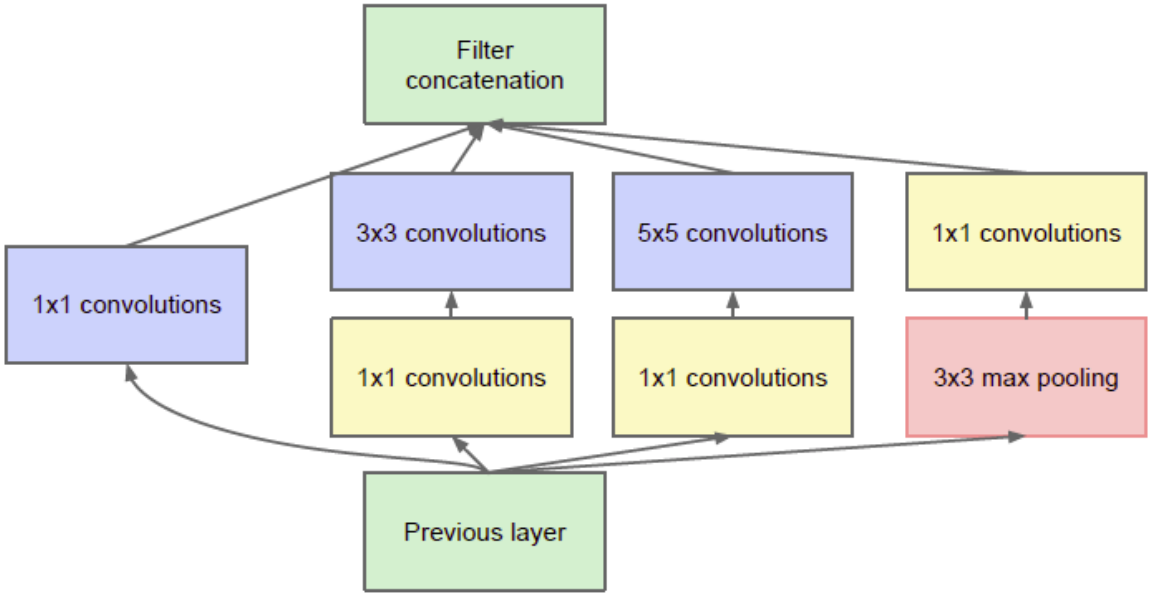


- 7 layers (not counting input image & class-label layer), 5 conv layers, 2 FC layers
- Input to 1st conv layer is 224 x 224 x 3 (x, y size & three colour channels); sampled (filtered) by 96 kernels for 3D (11x11x3) convolutions with a stride-4 (4 pixels distance shift between the centre of each 11x11x3 convolution kernel)
- Input to 2nd conv layer is the output from 1st after maxpooling: Filtering by 256 kernels 3D convolution with a feature map size 55x55x48 (48 channels of features), kernel size 5x5x48
- Input to 3rd conv layer is output of 2nd after maxpooling: 384 kernels with feature map size 27x27x128 and the kernel size of 3x3x128; 4th identical to 3rd; 5th with kernel size 3x3x192
- 6th and 7th FC layers have a single vector of 4096 dimensions feeding into a 1000 units output class label layer (1000 dimensions) by softmax (multinomial logistic regression)
- The last (8th) layer represents 1000 object class labels, with a softmax function computing a probability distribution (sum to 1) for class prediction

Going Deeper (GoogLeNet, 2014)



The inception module (convolutions and maxpooling)



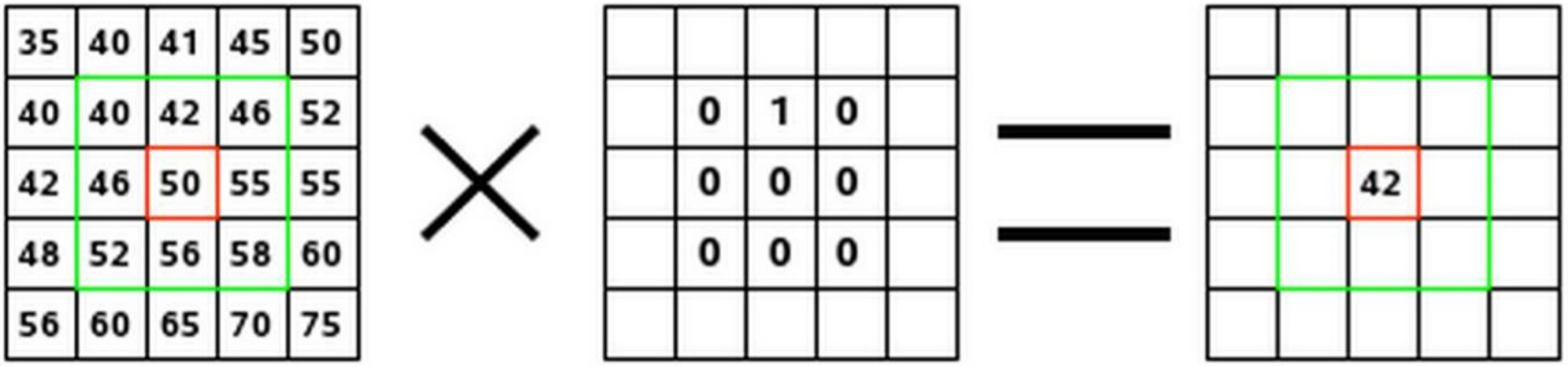
22 parameter layers (27 including pooling layers) vs. AlexNet 8 parameter layers

Pro: Fewer parameters
Con: Harder to train

CNN Key Concepts

- Convolution
- Pooling
- ReLU
- Dropout
- Data augmentation
- Normalisation
- Deeper the better, and many epochs

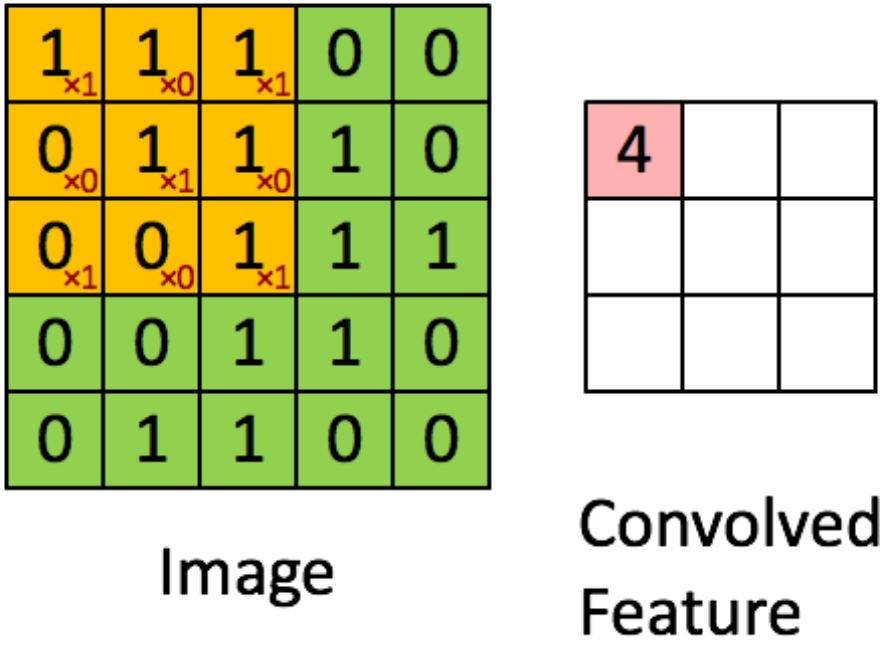
Convolution



The 9 weights W_{ij} are called a Kernel.

The weights are not fixed but learned

Convolution as Kernel Sliding Window – Stride size



The *same* weights are "slid" over the image at a fixed displacement distance – stride size

Effect of convolution kernel sliding

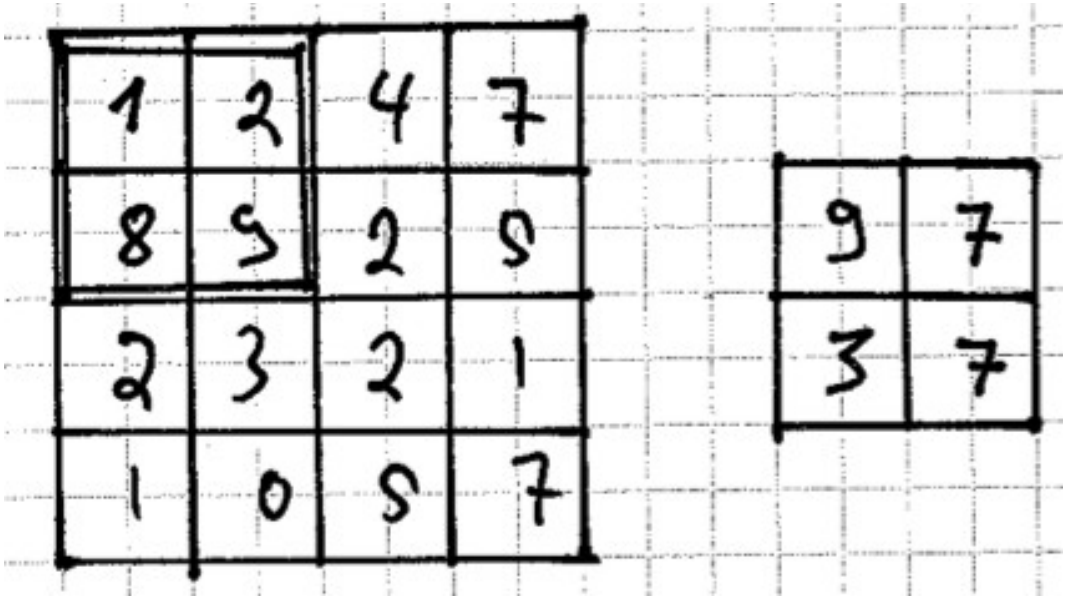
Edge enhance
Filter

	0	0	0	
	-1	1	0	
	0	0	0	

The weights are not fixed but learned

Max-Pooling

1. Reduce number of weights / resolution
2. More robust to mis-alignment
3. At the cost of losing information



Subsampling, e.g. 2x2 adjacent pixels in one

Also sliding window (stride)

Hinton (2014 AMA): "*The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster*"!

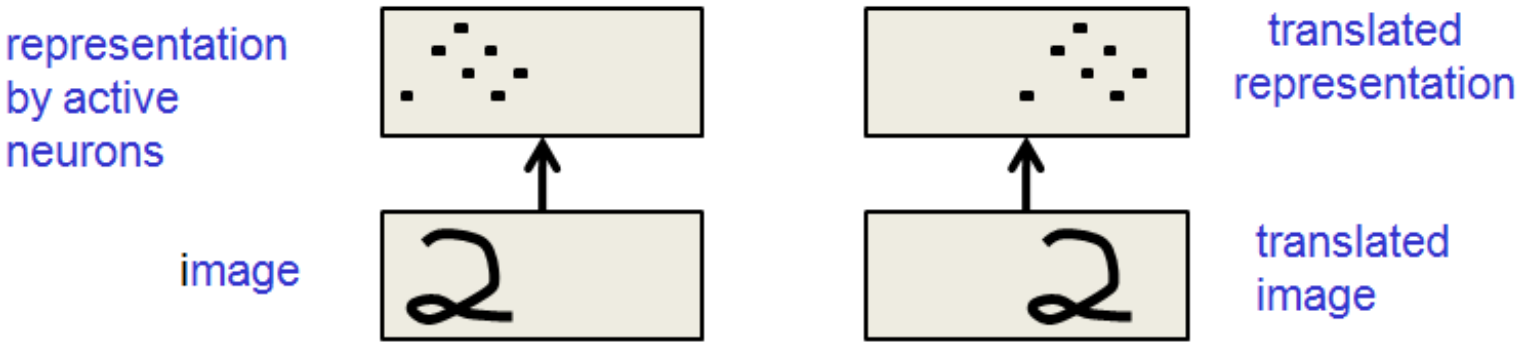
https://www.reddit.com/r/machinelearning/comments/2lmo0l/ama_geoffrey_hinton

Pooling the outputs of replicated feature detectors

1. Get a small amount of translational invariance at each level by averaging four neighbouring replicated detectors to give a single output to the next level.
    - This reduces the number of inputs to the next layer of feature extraction, thus allowing us to have many more different feature maps.
    - Taking the maximum of the four works slightly better.
2. Problem: After several levels of pooling, we have lost information about the precise positions of things.
    - This makes it impossible to use the precise spatial relationships between high-level parts for recognition.
    - Pooling “works” due to overlapping but this is not optimal.

What replicating the feature detectors achieve?

- Equivariant activities: Replicated features do not make the neural activities invariant to translation. The activities are equivariant.

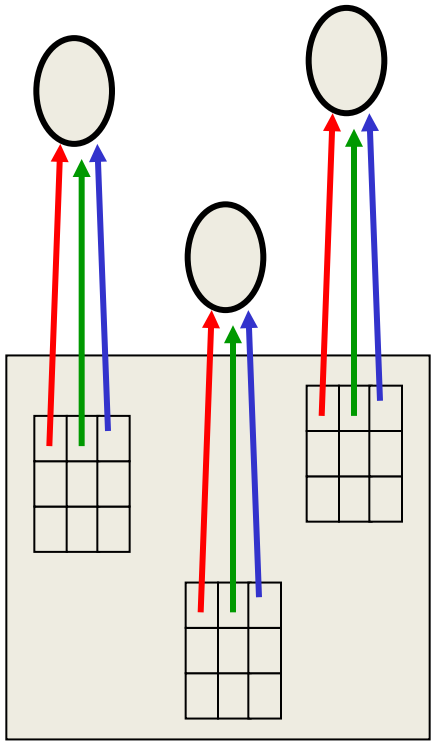


- Invariant knowledge: If a feature is useful in some locations during training, detectors for that feature will be available in all locations during testing.

Why Deep – Replicated features

- Use many different copies of the same feature detector with different positions.
  - Could also replicate across scale and orientation (tricky and expensive)
  - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
  - Allows each patch of image to be represented in several ways.

The red connections all have the same weight.



Encoding Priors: Architecture vs. Data

1. Encode human prior knowledge about the task into the network by designing appropriate:
    - Connectivity.
    - Weight constraints.
    - Neuron activation functions

- Encoding priors less intrusive than hand-designing the features.
    - But it still prejudices the network towards the particular way of solving the problem as we want (task driven).

2. Alternatively, use prior knowledge to create more training data.
    - This may require a lot of work (Hofman&Tresp, 1993)
    - It may make learning to take much longer.

- The optimization process discovers optimal ways of using the network that we did not know (so cannot design).
    - And we may never fully understand how it does it.

Avoid overfitting by random repeats

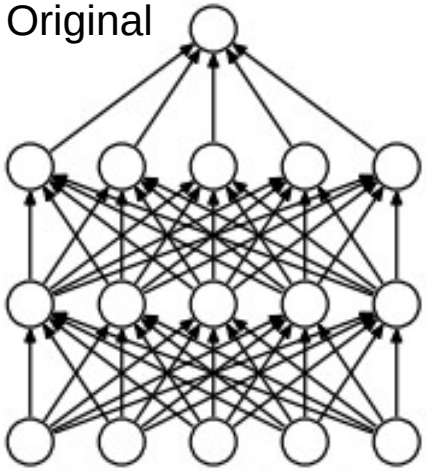
1. Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images (data augmentation)
    - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.

2. Use "dropout" to regularize the weights in the globally connected layers (which contain most of the parameters).
    - Dropout – half of the hidden units in a layer are randomly "removed" (weights frozen) for each training example.
    - This stops hidden units from relying too much on other hidden units nearby.

Avoid Overfitting – Dropout

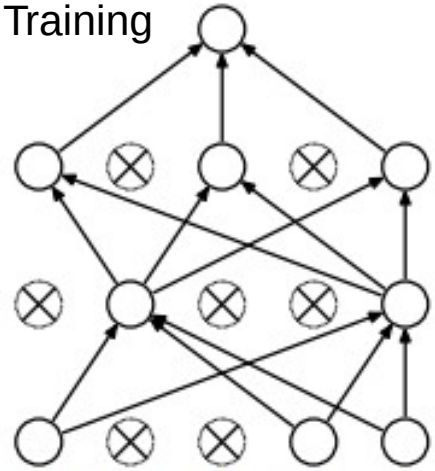
Model overfitting: Small training size to learn large parameter space results in poor model generalisation

Original



(a) Standard Neural Net

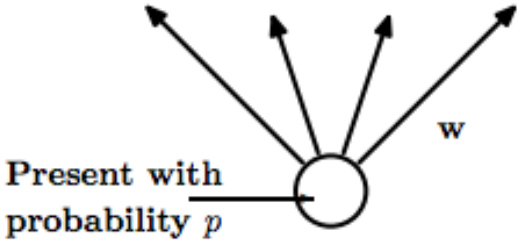
Training



(b) After applying dropout.

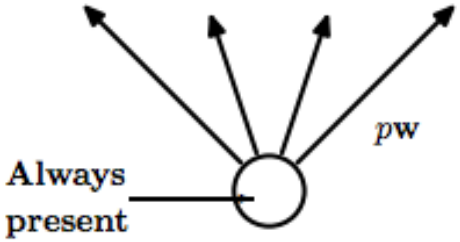
At each mini-batch remove random nodes “dropout”

Training time



(a) At training time

Test Time



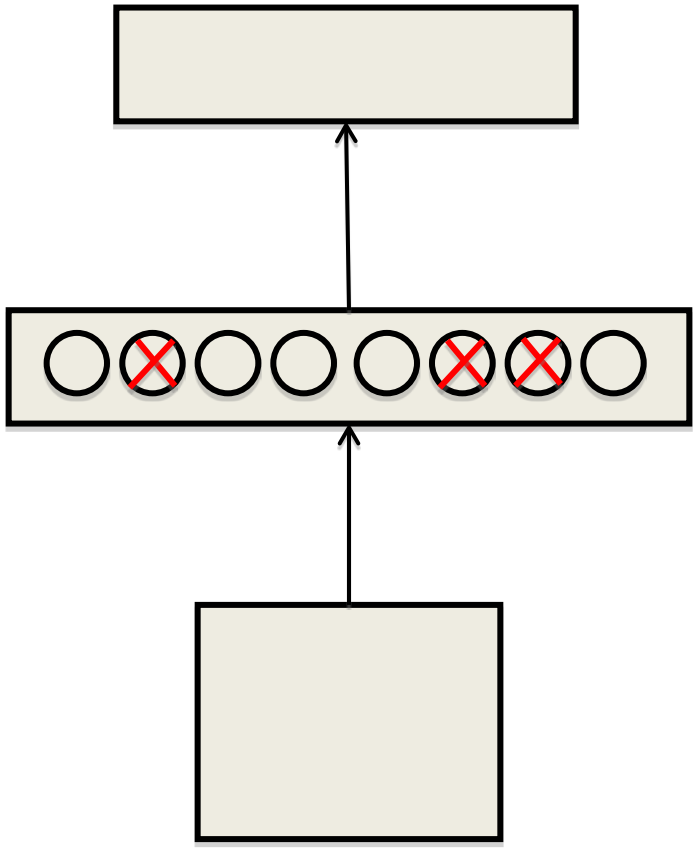
(b) At test time

Insight: Averaging over many different configuration (exact in case of linear). Typically 10% performance increase

Srivastava et al., Journal of Machine Learning Research 15 (2014) 1929-1958

Dropout: Average Many Network Models

- Consider a neural net with one hidden layer.
- Each time we present a training example, we randomly omit each hidden unit with probability 0.5.
- Randomly sampling from a total of 2^U different “thinned” architectures (U=number of total units in a network)
    - All architectures share weights (training time: “omit” implies reduced weight, e.g. 0.5, rather than removes a unit; test time: use all the weights of different models – model averaging)



Srivastava et al., Journal of Machine Learning Research 15 (2014) 1929-1958

Two Ways to Average Models (test time)

1. MIXTURE (arithmetic mean): Combine models by averaging *n* models output probabilities

Model A:	.3	.2	.5
Model B:	.1	.8	.1
Combined	.2	.5	.3

2. PRODUCT (geometric mean): Combine models by taking the *n*-th root of the product of *n* models output probabilities

Model A:	.3	.2	.5
Model B:	.1	.8	.1
Combined	$\sqrt{.03}$	$\sqrt{.16}$	$\sqrt{.05}$

3. Use all the hidden units but halve their outgoing weights (each hidden layer)

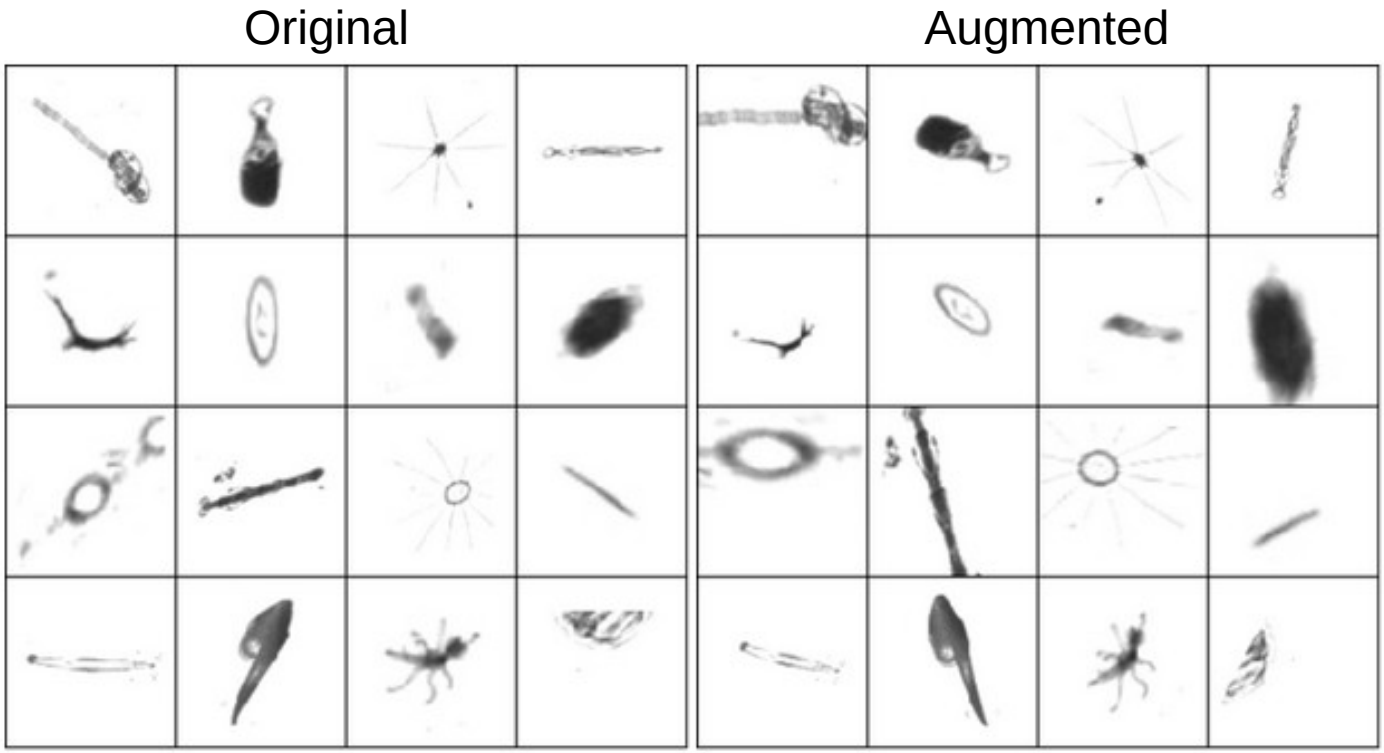
    - This computes the geometric mean of the predictions of all 2^U possible “thinner” models from U number of units in a network

Dropout Average of Many Hidden Layers (test time)

- Use dropout of 0.5 in every hidden layer.
- At test time, use the “mean net” that has all the outgoing weights halved.
  - This is not exactly the same as averaging all the separate dropped out models, but a good approximation, and fast.
- Alternatively, run the stochastic model several times on the same input.
  - This gives an idea of the uncertainty in the answer.
- Good news: Dropout helps overfitting signiticantly (10% improvement on error rate)
- Bad news: Takes much longer to train a model
- Insight: If a model is not overfitting given the available data, design a “bigger model” (deeper) and deploy dropout and data augmentation to overcome overfitting.

Avoid Overfitting: Data Augmentation

- Expand "new training" data through "label preserving transformation"
- Introduce simulated data variance under translation, rotation, scaling



Pre-processed images (left) and augmented versions of the same images (right).

Taken from the winning solution of the plankton challenge. http://benanne.github.io/2015/03/17/plankton.html