Chapter 7:
Convolutional Neural Networks (CNN)

Neural networks and deep learning

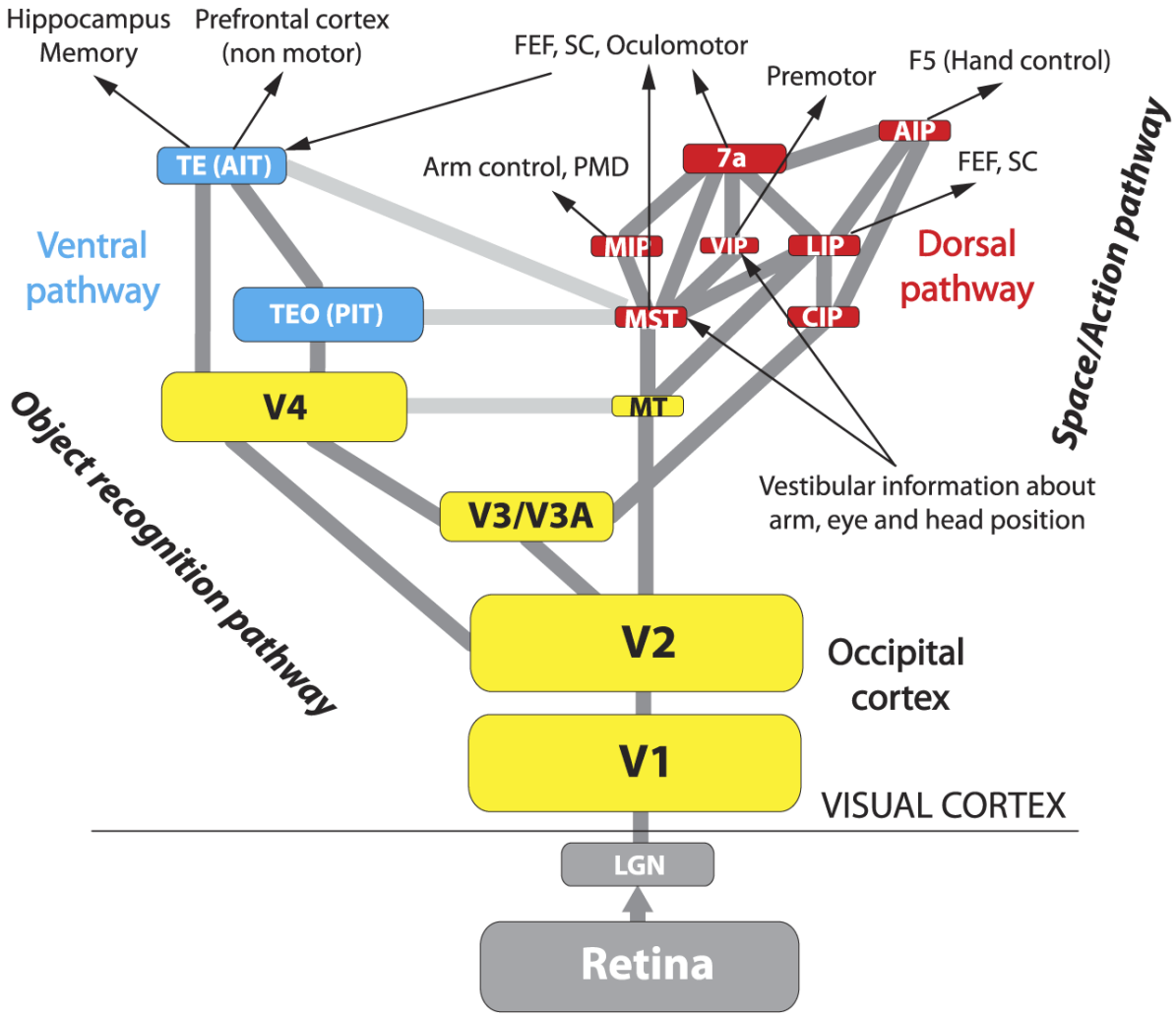
Motivation

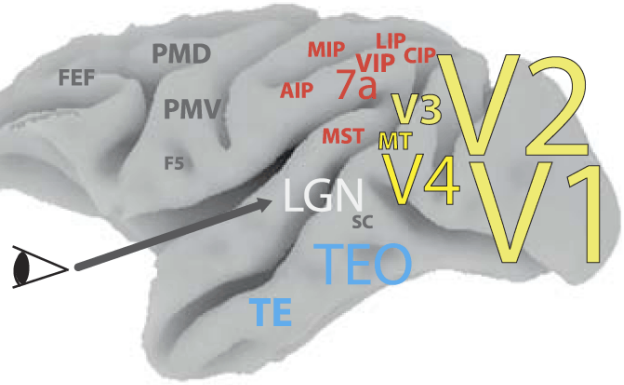
Convolutional neural networks (CNN) are biologically inspired variants of feedforward neural networks. The animal visual cortex being the most powerful visual processing system in existence and CNN emulates its behavior.

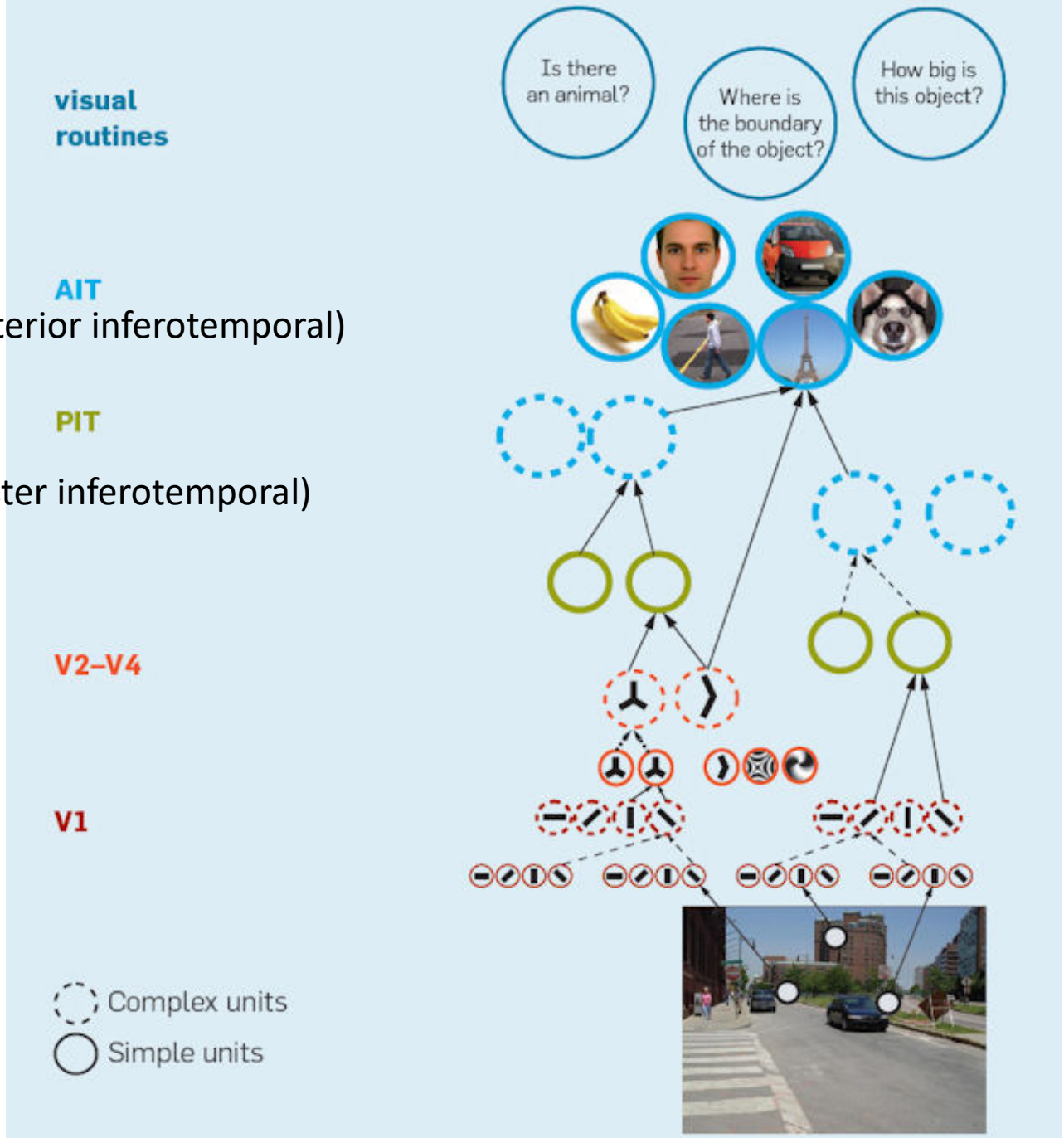
Visual cortex contains a complex arrangement of neurons. Each cell is responsive to small subregions of the visual field, known as *receptive fields*. The subregions are tiled over to cover the entire visual field. The neurons act as local filters over the input space and exploit the strong local correlations in natural images.

There are two types of cells (neurons) are identified: *simple cells* and *complex cells*. Simple cells respond maximally to specific edge-like patterns within their respective receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the patterns

Visual pathways of the brain






visual routines
Is there an animal?
Where is the boundary of the object?
How big is this object?
AIT
(anterior inferotemporal)
PIT
(poster inferotemporal)
V2–V4
V1
Complex units
Simple units

Locally connected networks

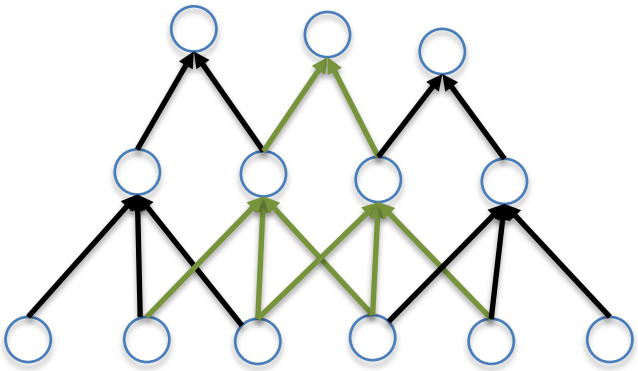
- Back propagation works for signals and images that are low in resolution or dimension. Deep convolution neural networks were developed to process more realistic signals and images that are of larger dimensions.

- Fully connected networks where neurons at one level are connected to the neurons in other layers are not feasible for signals of large resolutions and are computationally expensive in feedforward and backpropagation computations.

- One solution would be to restrict the number of connections from one layer to the other layer to a smaller subset. The idea is to connect only a contiguous (local) set of nodes to the next layer.

Sparse local connectivity

CNNs exploit spatially local correlations by enforcing local connectivity between neurons of adjacent layers.

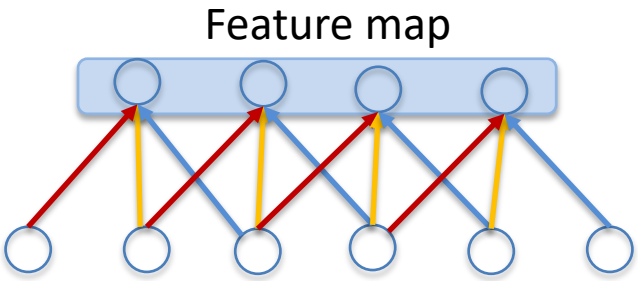


Note that the receptive fields of the neurons are limited because of local connectivity. The receptive fields of neurons at the upper layers combine the receptive fields of neurons at the lower layer.

For example, in the above figure the receptive field of the hidden-layer neurons is 3 neurons in the input layer and that of output-layer neurons is 4 input neurons.

Shared connections

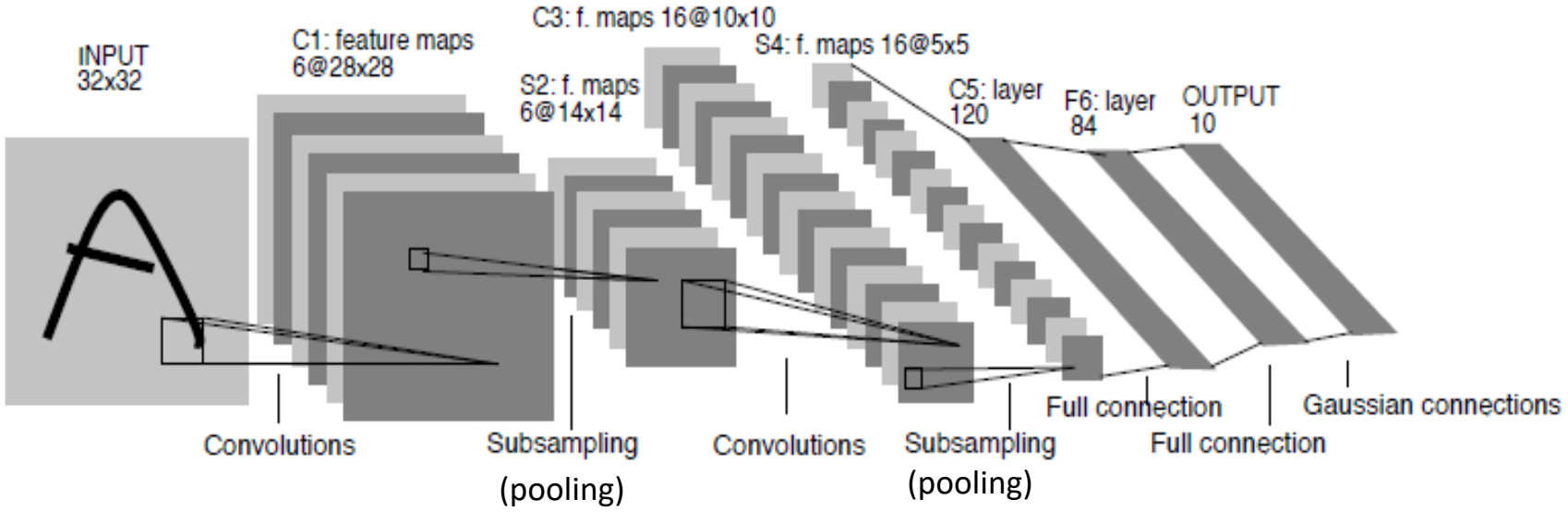
In CNN, each filter (weight vector) is replicated across the entire visual field. These replicated units share same parameterization (weight vector and bias) and form a *feature map*. Feature maps represent activations of the neurons at the upper layers, corresponding to individual filters.



In the above figure, the output neurons belong to the same feature map. Weights of the same color are shared – constrained to be identical. Gradient descent can still learn such share parameters.

Replicating weights in this ways allows for features to be detected regardless of the position in the visual field. Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters.

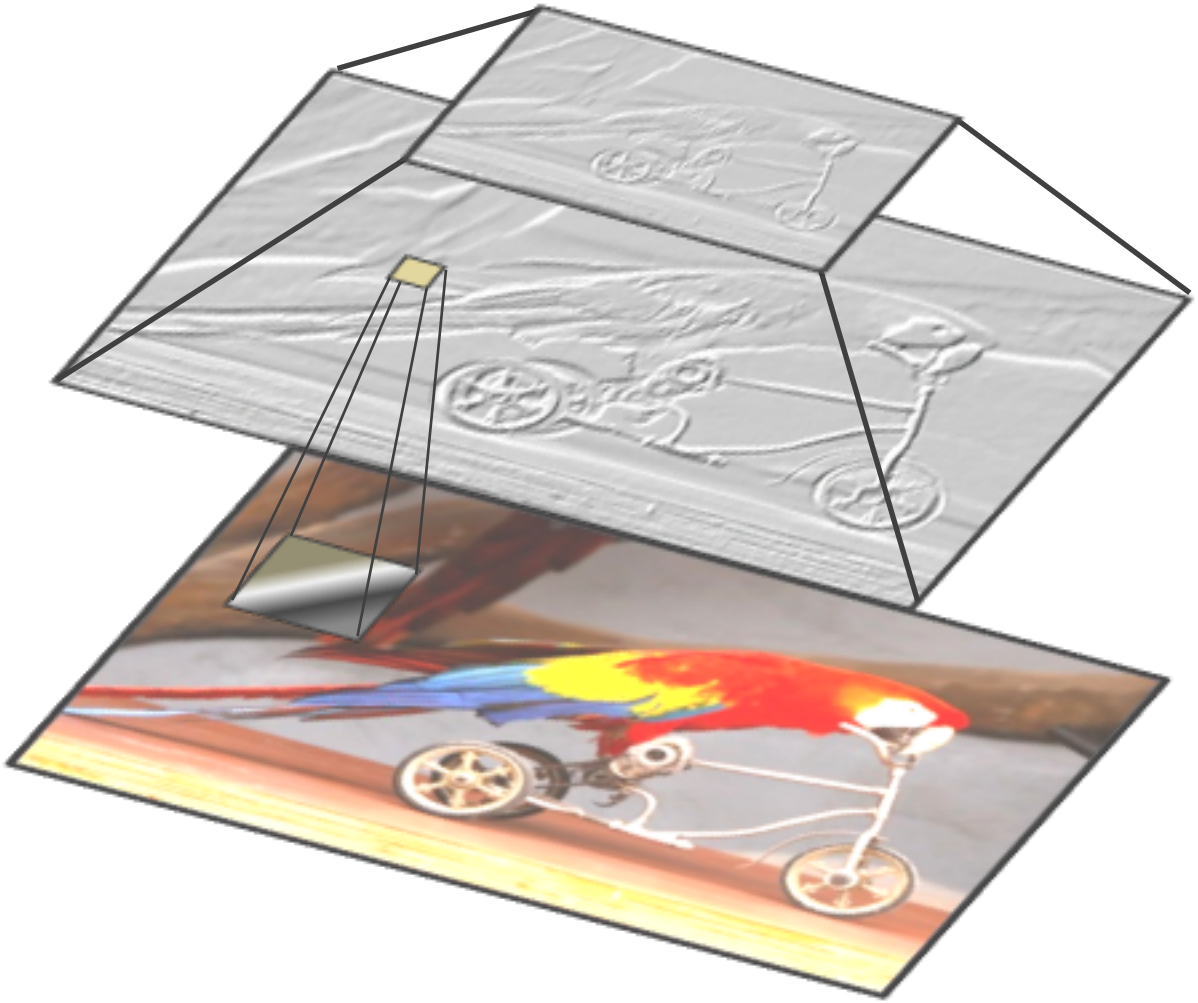
Convolutional networks: LeNet 5



LeCun et al., 1998

60,000 train and 10,000 test images from MNIST database

Basic Computations in CNN

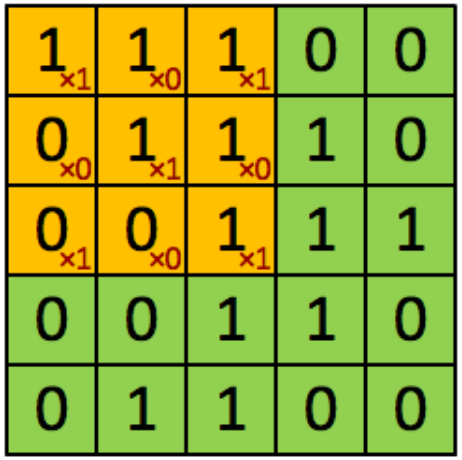


Pooling

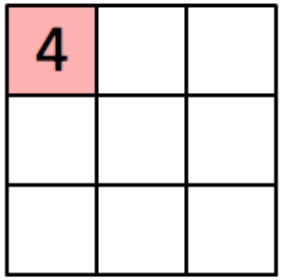
Convolution

Image

Convolution Layer



Image



Convolved Feature

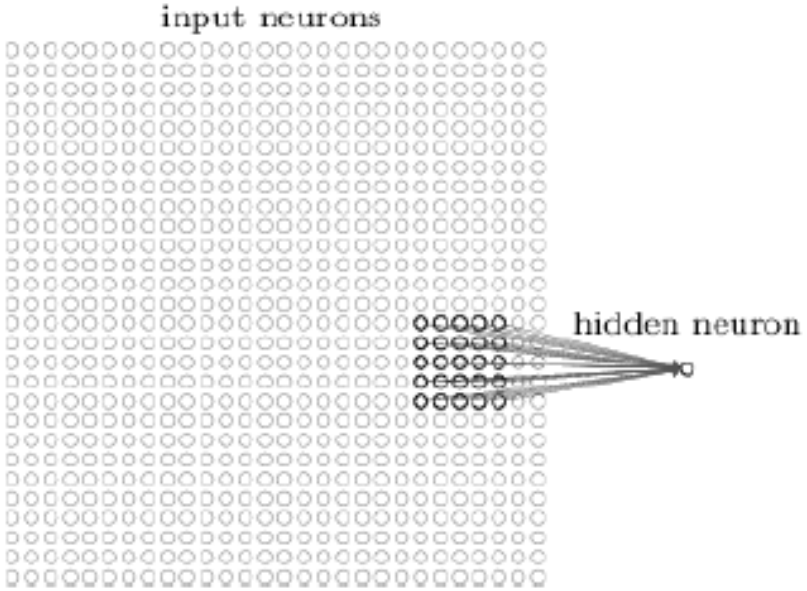
$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

Filter (weights or kernel)

Convolution Layer

- Natural images are stationary meaning that the features of the images are invariant at every location of the image. Filters are replicated over to cover and learn over the entire visual field.

- The convolution layer learns the features over small patches of the image, and the learned features (filters) are then convolved with the image to obtain feature activations (maps).

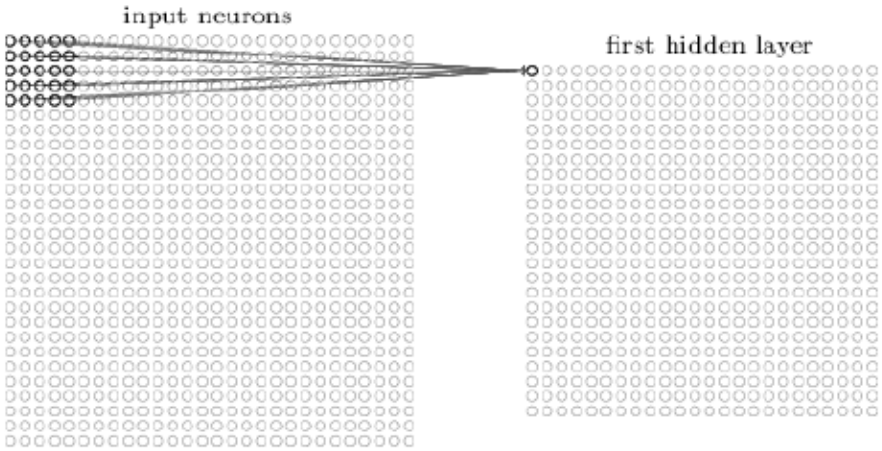


Output of the neuron:

$$y_p = f\left(\boldsymbol{x}_p^T \boldsymbol{w} + b\right)$$

where $\boldsymbol{x}_p$ is the local window of inputs at pth pixel location.

Convolution Layer



- Features or the weights of filters are learned using feature detectors or backpropagation algorithm. The features or weights learned are known as *filters* or *kernels*
- The activations are obtained by convolving the filters (weights) with the input activations. The output activation learned by a particular filter is known as a *feature map*
- The region of the input layer that is connected to a layer is referred to as the *receptive field*.

Convolution Layer

Synaptic input at location $p = (i, j)$ of the first hidden layer due to a kernel $\boldsymbol{w} = \{w(l,m)\}_{l.m=-L/2,-M/2}^{L/2,M/2}$ is given by

$$u(i,j) = \sum_{l=-L/2}^{L/2} \sum_{m=-M/2}^{M/2} x(i+l, j+m) w(l,m) + b$$

where $L \times M$ is the window size.

The output of the neuron at (i, j) of the convolution layer,

$$y(i,j) = f\big(u(i,j)\big)$$

Where *f* is the sigmoid activation function.

Convolutional Layer

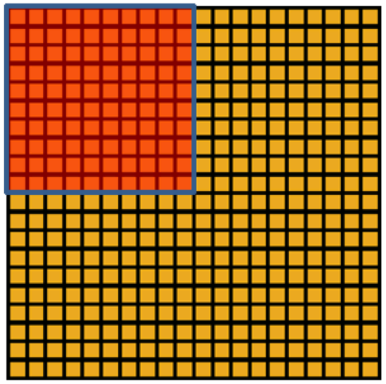
Note that one weight tensor $\boldsymbol{w}_k = \{ w_k(l,m)\}$ or kernel (filter) creates one feature map:

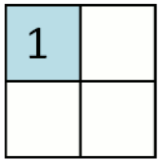
$$\boldsymbol{y}_k = \{y_k(i,j)\}.$$

If there are K weight vectors $(\boldsymbol{w}_k)_{k=1}^K$, the convolutional layer is formed by K feature maps:

$$\boldsymbol{y} = (\boldsymbol{y}_k)_{k=1}^K$$

Pooling Layer





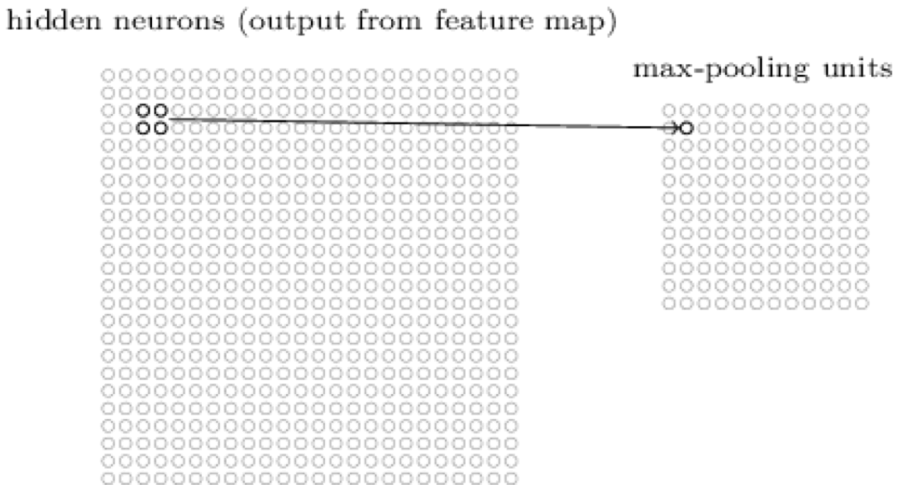
Convolved feature

Pooled feature

- To reduce the dimensions (that is to subsample the input space) of the convolution layer, the pooling of the activation is performed at the pooling layer. Either ‘max’ or ‘average’ pooling is used at the pooling layer. Pooling *down-sample* the input.

- The convolved features are divided into disjoint regions and pooled by taking either maximum or averaging. Pooled features are ‘translational invariant’.

Pooling Layer



Consider polling with non-overlapping windows $\{(l,m)\}_{l.m=-L/2,-M/2}^{L/2,M/2}$ of size $L \times M$.

The max pooling output is the maximum of the activation inside the pooling window. Polling of a feature map $\boldsymbol{y}$ at $p = (i,j)$ produced pooled feature:

$$z(i,j) = \max_{l,m}\{y(i+l, j+m)\}$$

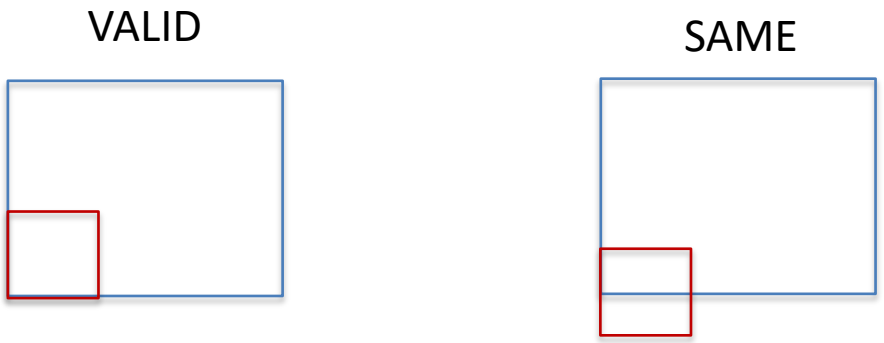
The mean pooling output is the mean of activations in the pooling window:

$$z(i,j) = \frac{1}{L \times M} \sum_{l} \sum_{m} y(i+l, j+m)$$

Size of the feature maps

There are two ways to handle the boundary:

1. VALID: apply the filter wherever it completely overlaps with the input.
2. SAME: apply the filter to make the output same size as input. The input is padded with zeros in obtaining the output.



If input image is $I \times J$, and kernel size is $L \times M$, then the size of the output layer:

VALID: $(I - L + 1) \times (J - M + 1)$

SAME: $I \times J$

In tensorflow, cases 'VALID' and 'SAME' are options for padding with convolution and pooling. Note that the input is scanned from top-left corner to right and down.

Size of the feature maps

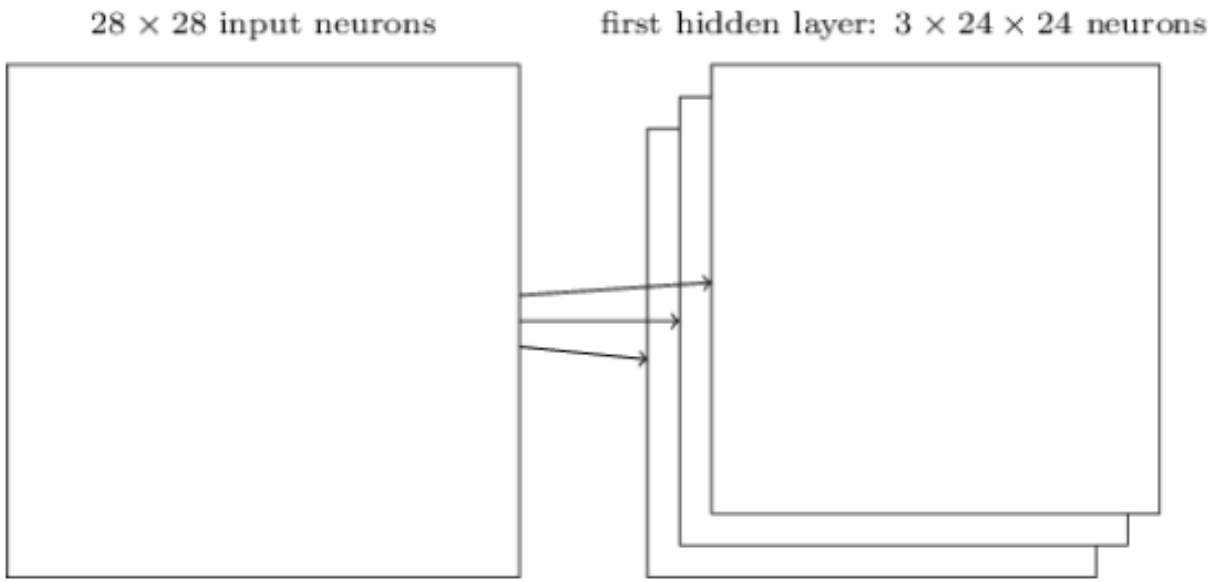
Stride is the factor with which the output is subsampled. That is, distance between adjacent centers of the kernel.

The convolution operation is intended to extract the features of the input space as finely as possible. The default strides for convolution is 1 unit.

Pooling is intended to subsample the convolution layer. The default stride for pooling is equal to the filter width.

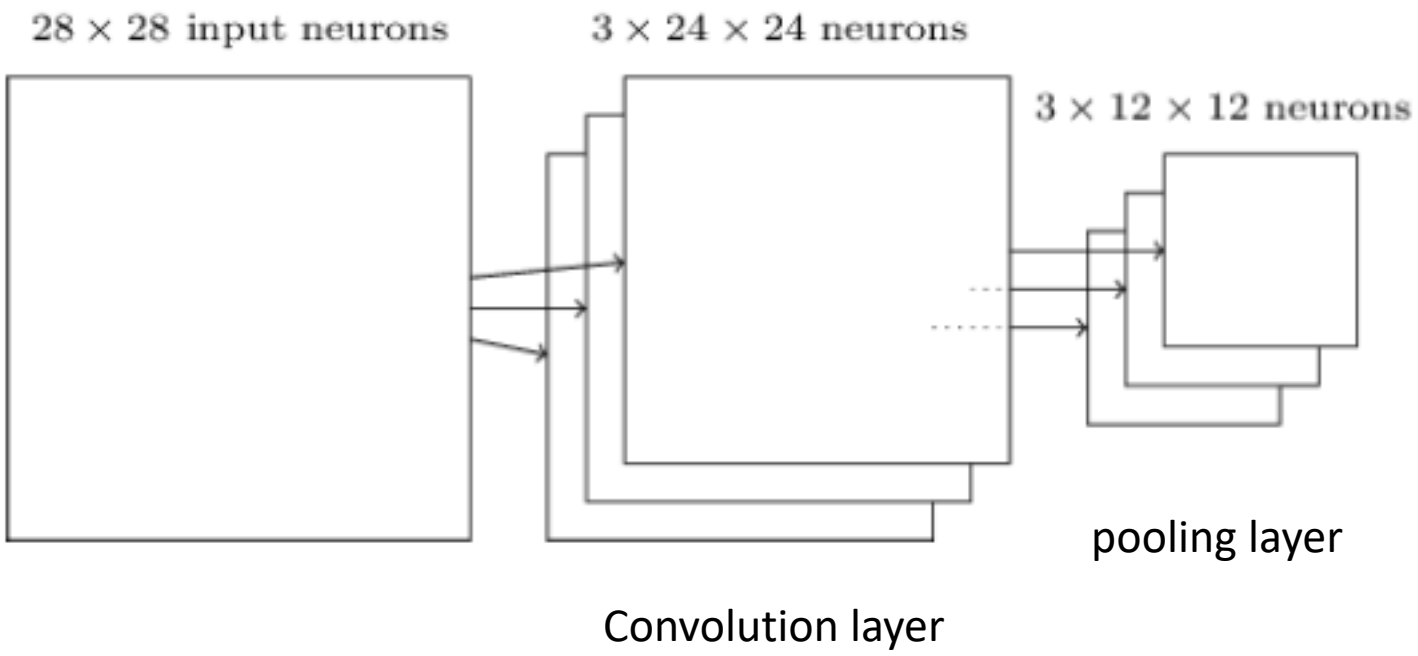
Convolution layer

Consider an input layer of 28x28 that uses ‘valid’ padding for convolution with stride = 1 and three features:



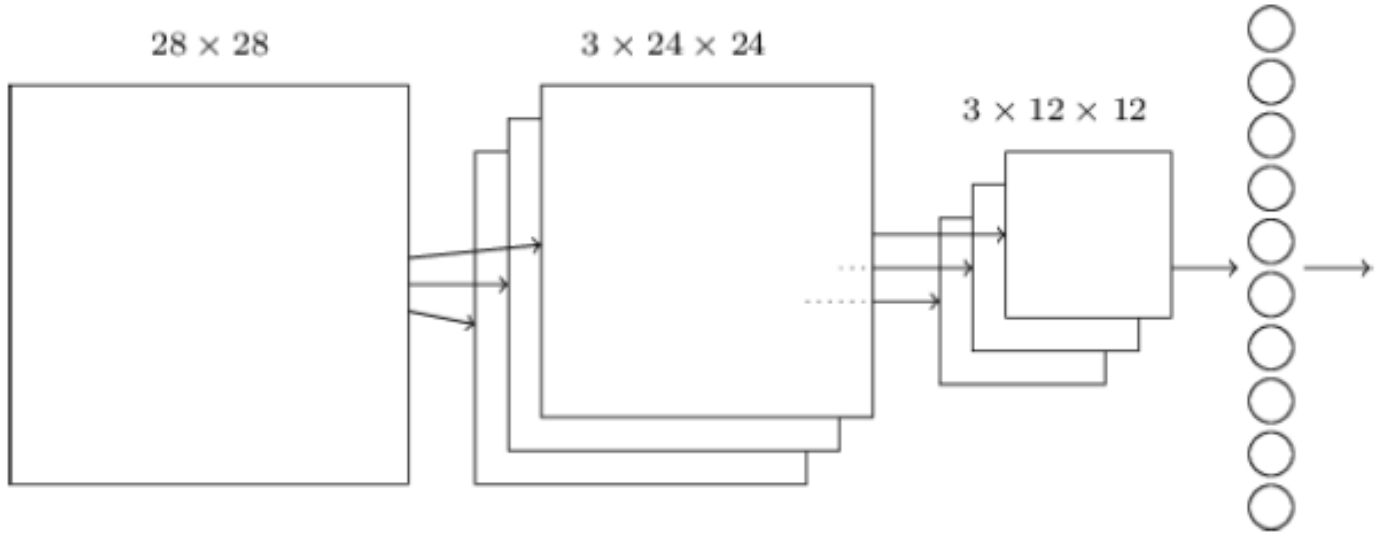
Pooling layer

If pooling window of size 2x2 is used with a stride of 2.



Fully Connected Layers

- The final layers of deep CNN consist of one or more fully connected layers. The output layer is usually a softmax regression layer for classification and a linear layer for regression.



Example 1

Given an input pattern $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$\boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ and bias = 0.05.

If convolution layer has a sigmoid activation function, find the outputs of the convolution layer if the padding is VALID at strides = 1.

If the pooling layer uses max pooling, has a pooling window size of 2x2, and strides = 2, find the activations at the pooling layer for VALID and SAME padding.

Example 1

$$\boldsymbol{I} = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ -1.0 & 0.2 & 0.0 & 0.3 & -0.2 \\ 0.7 & 0.1 & 0.2 & -0.6 & 0.3 \\ -0.4 & 0.0 & 0.2 & 0.3 & -0.3 \end{pmatrix}; \boldsymbol{w} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(i,j) = \sum_l \sum_m x(i+l, j+m) w(l,m) + b$$

For VALID padding:

$u(1,1) = 0.5\times0 - 0.1\times1 + 0.2\times1 + 0.8\times1 + 0.1\times0 - 0.5\times1 - 1.0\times1 + 0.2\times1 + 0.0\times0 + 0.05 = -0.35$

$u(1,2) = -0.1\times0 + 0.2\times1 + 0.3\times1 + 0.1\times1 - 0.5\times0 + 0.5\times1 + 0.2\times1 + 0.0\times1 + 0.3\times0 + 0.05 = 1.35$

$u(1,3) = 0.2\times0 + 0.3\times1 + 0.5\times1 - 0.5\times1 + 0.5\times0 + 0.1\times1 + 0.0\times1 + 0.3\times1 - 0.2\times0 + 0.05 = 0.75$

$u(2,1) = 0.8\times0 + 0.1\times1 - 0.5\times1 - 0.1\times1 + 0.2\times0 + 0.0\times1 + 0.7\times1 + 0.1\times1 + 0.2\times0 + 0.05 = -0.55$

Example 1

Synaptic input to the convolution layer:

$$\boldsymbol{U} = \begin{pmatrix} -0.35 & 1.35 & 0.75 \\ -0.55 & 0.85 & 0.05 \\ 0.75 & 0.05 & 1.15 \end{pmatrix}$$

Output of the convolution layer:

$$f(\boldsymbol{U}) = \frac{1}{1 + e^{-\boldsymbol{U}}} = \begin{pmatrix} 0.413 & 0.794 & 0.679 \\ 0.366 & 0.701 & 0.512 \\ 0.679 & 0.512 & 0.76 \end{pmatrix}$$

Output of the max pooling layer:

(0.794) for 'VALID' pooling

$\begin{pmatrix} 0.794 & 0.679 \\ 0.679 & 0.76 \end{pmatrix}$ for 'SAME' padding

Example 2

Inputs are digit images from MNIST database:
http://yann.lecun.com/exdb/mnist/

Input image size = 28x28

First convolution layer consists of three filters:

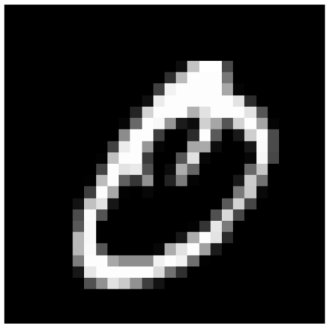
$$w_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad w_2 = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}, \quad w_3 = \begin{pmatrix} 3 & 4 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 3 \end{pmatrix}$$

Find the feature maps at the the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID'.
For pooling layer, use a window of size 2x2 and a stride if 2 and 'VALID' padding.

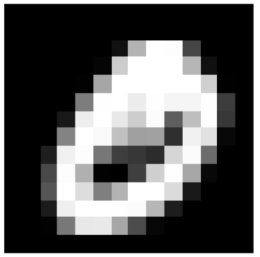
Size of convolution layer: 3x26x26
Size of pooling layer: 3x13x13

Example 2



Original Image
28x28

Convolution layer
3 x 26 x26



Mean-Pooling layer
3 x 13 x 13

```
x = [no_patterns, input_width, input_height, no_input_maps]
w = [ window_width, window_height, input_maps, output_maps]
```

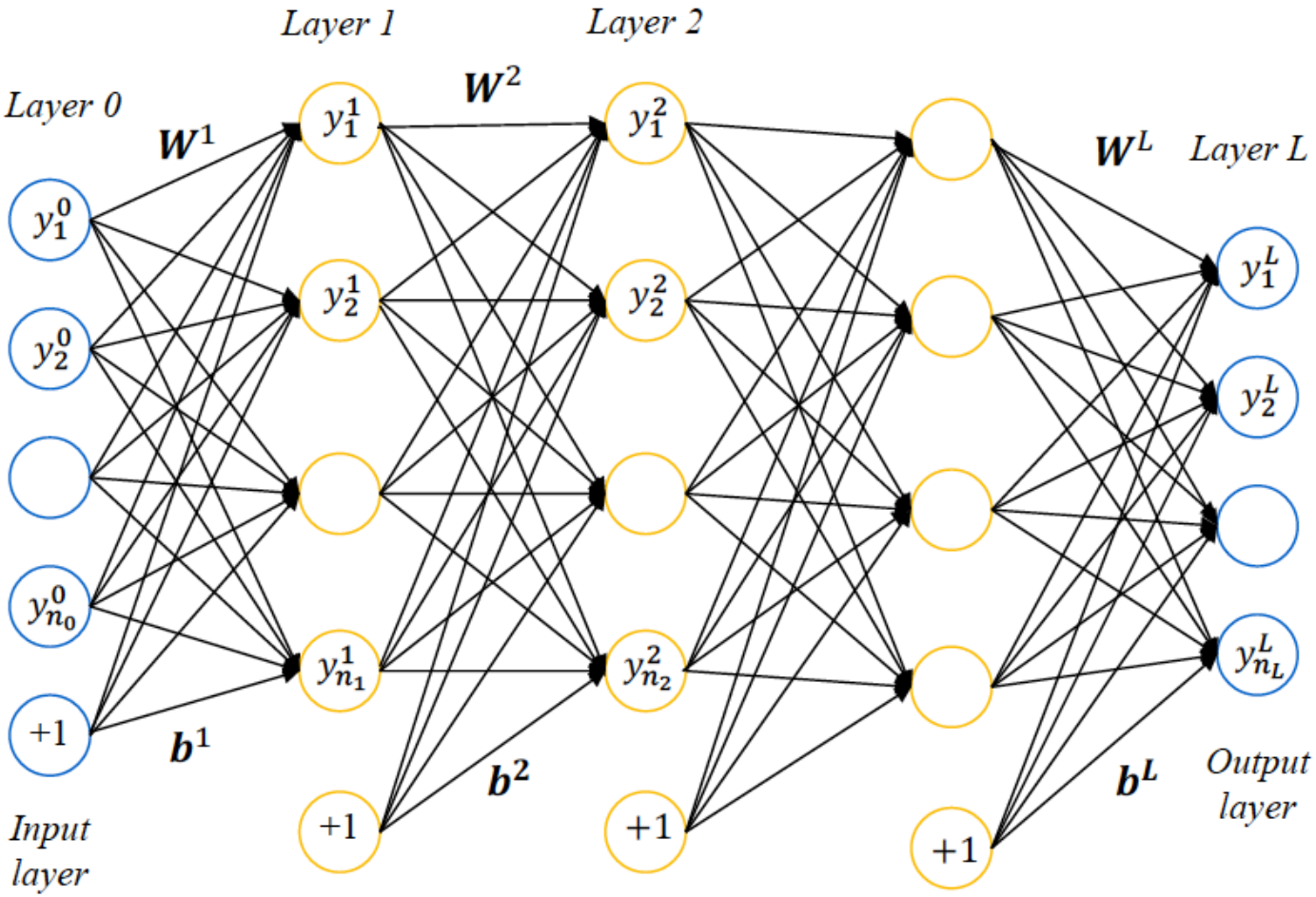
```
u1 = tf.nn.conv2d(x, w, strides = [1, 1, 1, 1], padding = 'VALID') + b
u2 = tf.nn.conv2d(x, w, strides = [1, 2, 2, 1], padding = 'SAME') + b
```

```
y1 = tf.nn.sigmoid(u1)
```

```
u1_, y1_ = sess.run([u1, y1], {x: I.reshape([1, 6, 6, 1])})
```

```
o1 = tf.nn.max_pool(y1, ksize=[1, 2, 2, 1], strides = [1, 2, 2, 1], padding = 'VALID')
```

DNN


Layer 0
Layer 1
Layer 2
Layer L
W^1
W^2
W^L
b^1
b^2
b^L
y_1^0
y_2^0
$y_{n_0}^0$
+1
y_1^1
y_2^1
$y_{n_1}^1$
+1
y_1^2
y_2^2
$y_{n_2}^2$
+1
y_1^L
y_2^L
$y_{n_L}^L$
Input layer
Output layer

Backpropagation for DNN

If $l = L$:

$$\nabla_{\boldsymbol{U}^l} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) \\ -\left(\boldsymbol{K} - f^L(\boldsymbol{U}^L)\right) \end{cases}$$

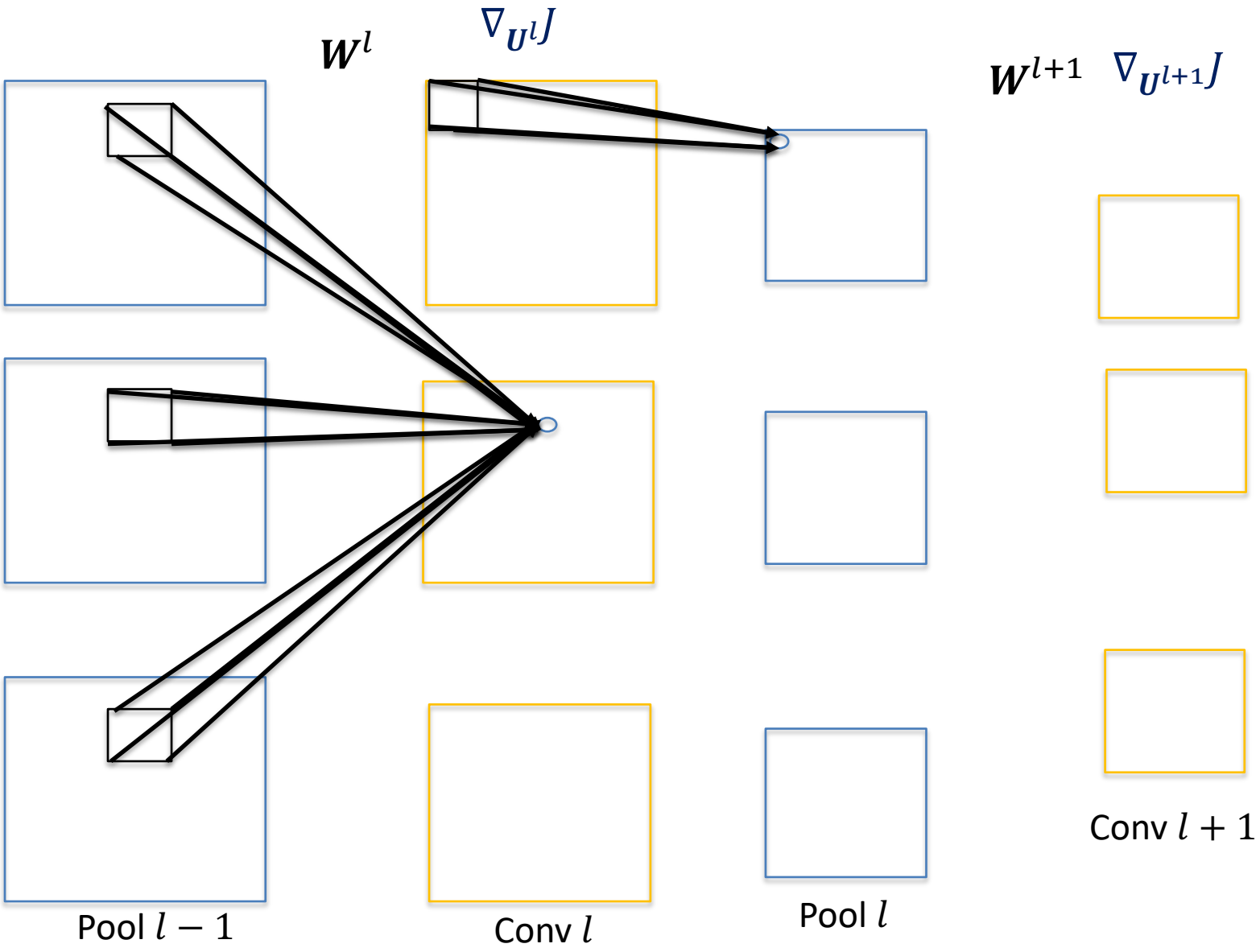
Else:

$$\nabla_{\boldsymbol{U}^l} J = \left(\nabla_{\boldsymbol{U}^{l+1}} J\right) \boldsymbol{W}^{l+1^T} \cdot f^{l'}\left(\boldsymbol{U}^l\right)$$

$$\nabla_{\boldsymbol{W}^l} J = \boldsymbol{H}^{l-1^T}\left(\nabla_{\boldsymbol{U}^l} J\right)$$

$$\nabla_{\boldsymbol{b}^l} J = \left(\nabla_{\boldsymbol{U}^l} J\right)^T \mathbf{1}_P$$

Backpropagation in CNN



Weights in CNN

Adaptable weights from previous pooling layer to a convolutional layer is a tensor of rank 4:

$$w(i, j, k_{in}, k_{out})$$

where k_{out}: output-layer feature map index

k_{in}: input-layer feature map index

index of the neuron location (i, j) of window of input layer

Backpropagation of deep CNN

During forward propagation, activations are downsampled at the pooling layer. So, during error backward propagation, the error terms need to be upsampled at the pooling layer.

In DNN, the error $\nabla_{\boldsymbol{U}^{l+1}} J$ is propagated to layer l as

$$\nabla_{\boldsymbol{U}^{l}} J = \left(\nabla_{\boldsymbol{U}^{l+1}} J\right) \boldsymbol{W}^{l+1^{T}} \cdot f^{l'}\left(\boldsymbol{U}^{l}\right)$$

For deep CNN, the transfer of error terms involves up-sampling of errors at the pooling Layer:

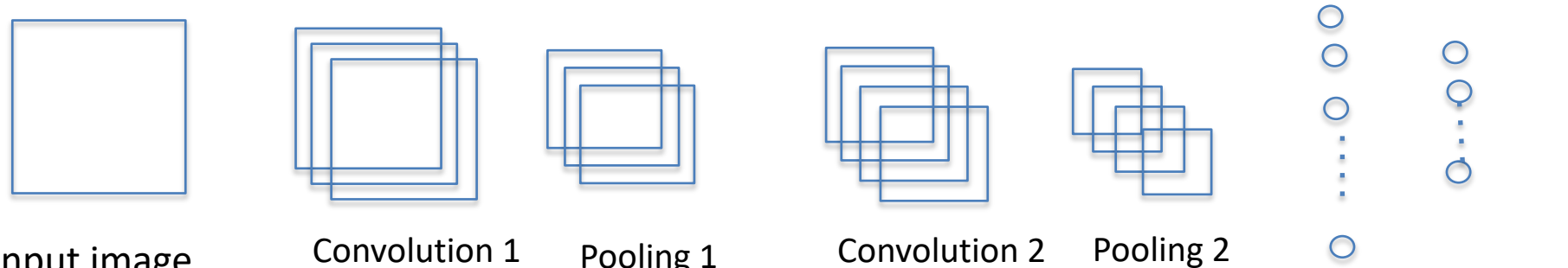
$$\nabla_{\boldsymbol{U}^{l}} J = upsample\left(\left(\nabla_{\boldsymbol{U}^{l+1}} J\right) \boldsymbol{W}^{l+1^{T}}\right) \cdot f'\left(\boldsymbol{U}^{l}\right)$$

Example 3: MNIST digit recognition

MNIST database: 28x28 =784 inputs
Training set = 12000 images
Testing set = 2000 images

Input pixel values were normalized:
mean = 0.0, s.d. = 1.0

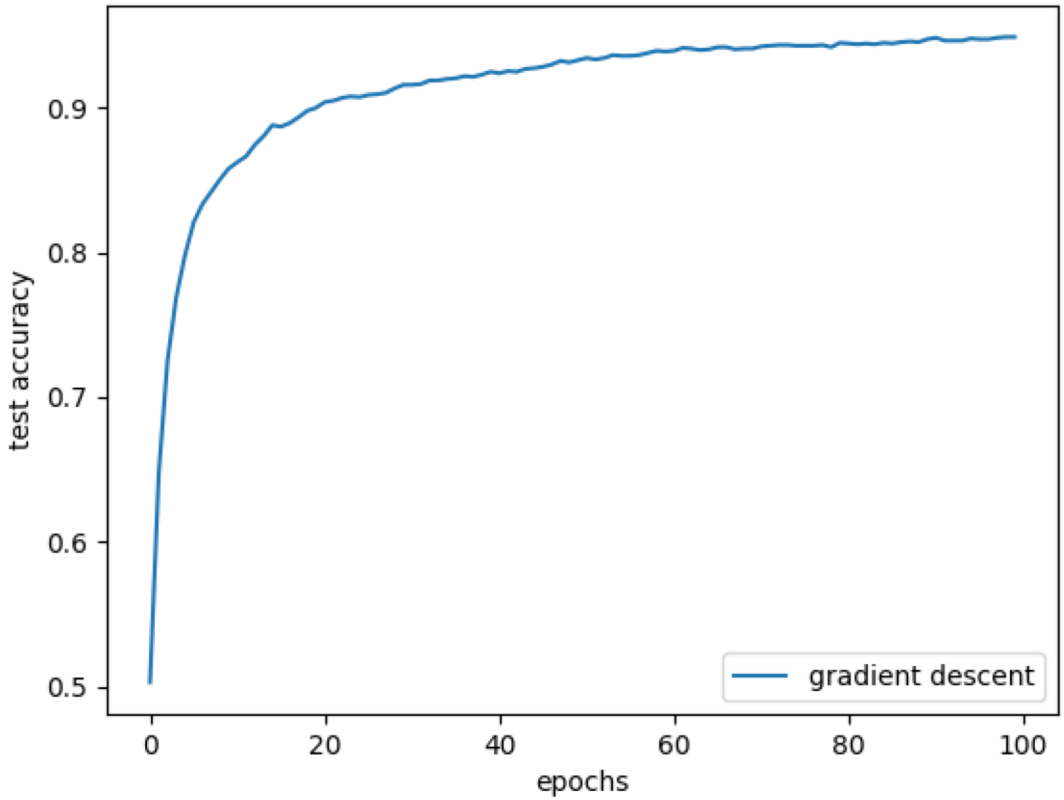
Example 3: Architecture of CNN



ReLU neurons

Gradient descent optimizer with batch-size = 128

Leaning parameter = 10^{-3}

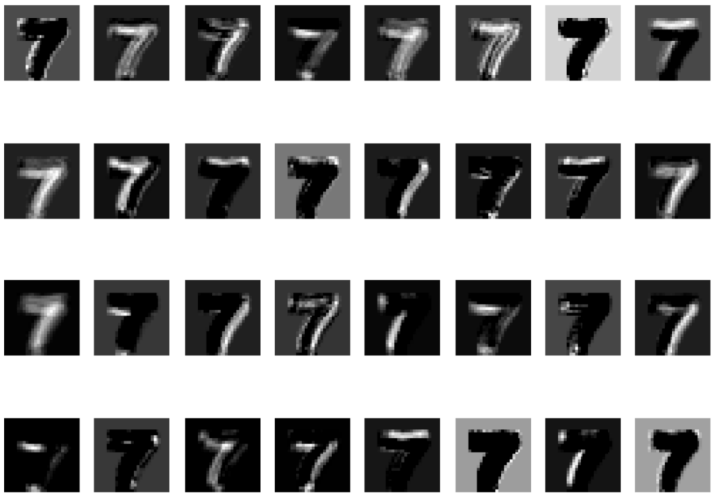

0.9
0.8
0.7
0.6
0.5
test accuracy
0
20
40
60
80
100
epochs
gradient descent

Weights learned at convolution layer 1

32 x 5 x 5

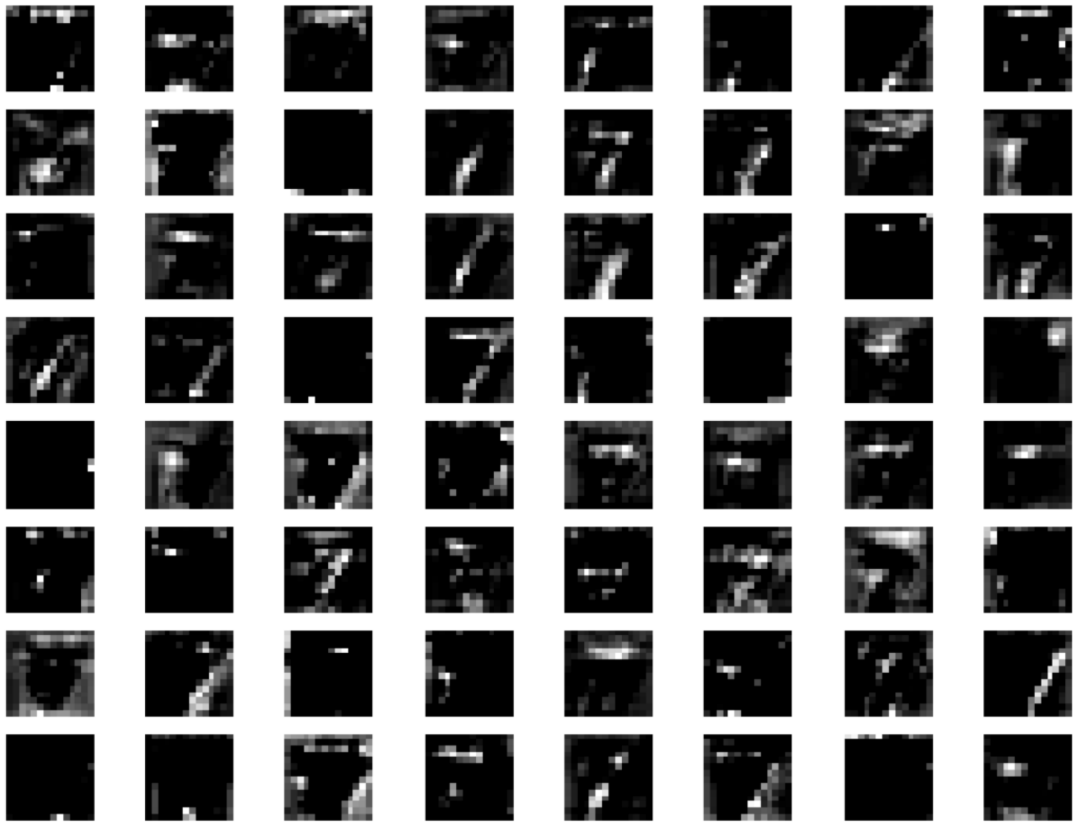
Input image
28x28

Feature maps
at conv 1
32x28x28

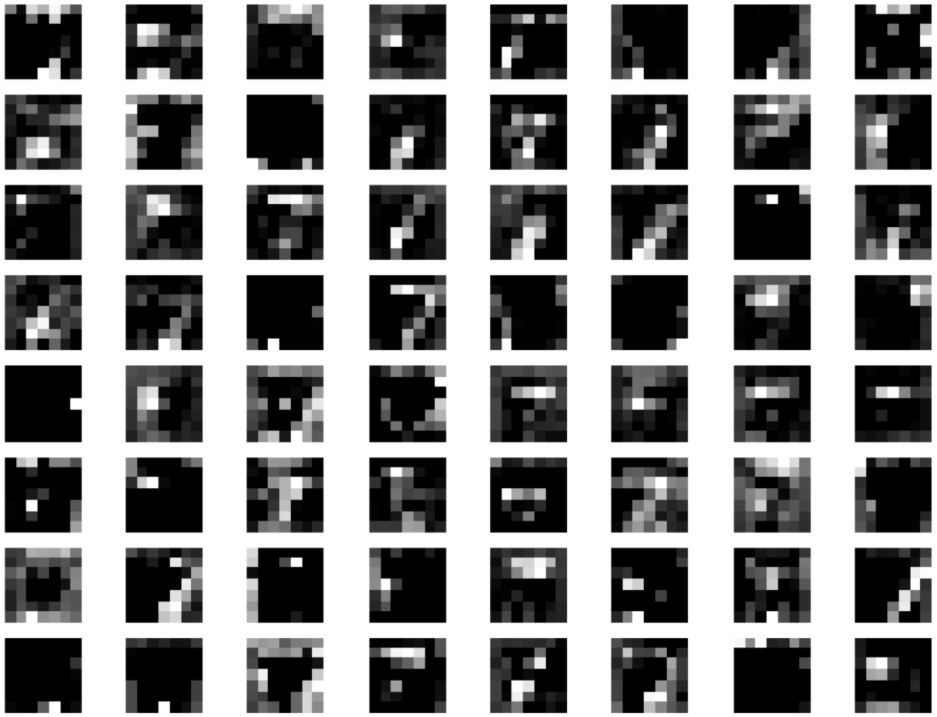


Feature maps
at pool 1
32x14x14

Feature maps at second convolution layer
64x14x14

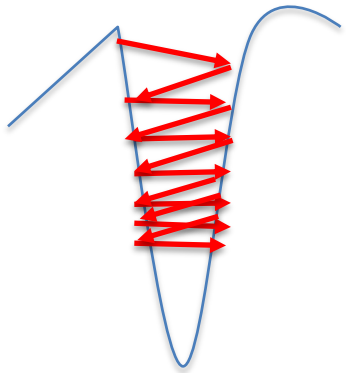


Feature maps at second pooling layer
64x7x7



GD with Momentum

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.



When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture.

Momentum is one method of speeding the convergence along a narrow ravine.

GD with Momentum

Momentum update is given by:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector $\boldsymbol{W}$.

The momentum parameter $\gamma \in [0,1]$ indicates how many iterations the previous gradients are incorporated into the current update. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

GD with Nesterov Momentum

With Nesterov momentum, the gradient is evaluated after the current velocity is applied:

$$\widetilde{\boldsymbol{W}} \leftarrow \boldsymbol{W} + \gamma \boldsymbol{V}$$
$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \nabla_{\widetilde{\boldsymbol{W}}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

Nesterov gradient improves the convergence of batch gradient.

Nesterov momentum attempts to add a correction factor to the standard momentum algorithm. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated.

GD with adaptive learning rates

Often, stochastic gradient descent employs a few training samples or a mini batch of samples. This reduces variance of the individual patterns and achieves stable convergence but at the expense of the true minimum.

The learning rate in the online (stochastic) version is much less than the learning rate in a batch version. When on-line or mini-batch version of gradient descent is used, because of the variance in individual patterns, it is not easy to determine the value of the learning parameter. One approach is to allow learning rate to adapt as learning proceeds.

Algorithms with adaptive learning rates:

- Anealing
- AdaGrad tf.train.AdagradOptimizer()
- RMSprop tf.train.RMSPropOptimizer()
- Adam tf.train.AdamOptimizer()

Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to start with a large leaning factor and then gradually reducing it.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions. Learning trajectory of a neural network minimizing non-convex cost function passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + (\nabla_{\boldsymbol{W}} J)^2$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

In other words, learning rate:

$$\tilde{\alpha} = \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}}$$

α and ε are two parameters.

RMSProp Algorithm

RMSProp improves upon AdaGrad algorithms uses and exponentially decaying average to decay from the extreme past gradient. RMSProp uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\varepsilon + \boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

The parameter ρ controls the length of the moving average of gradients.

RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

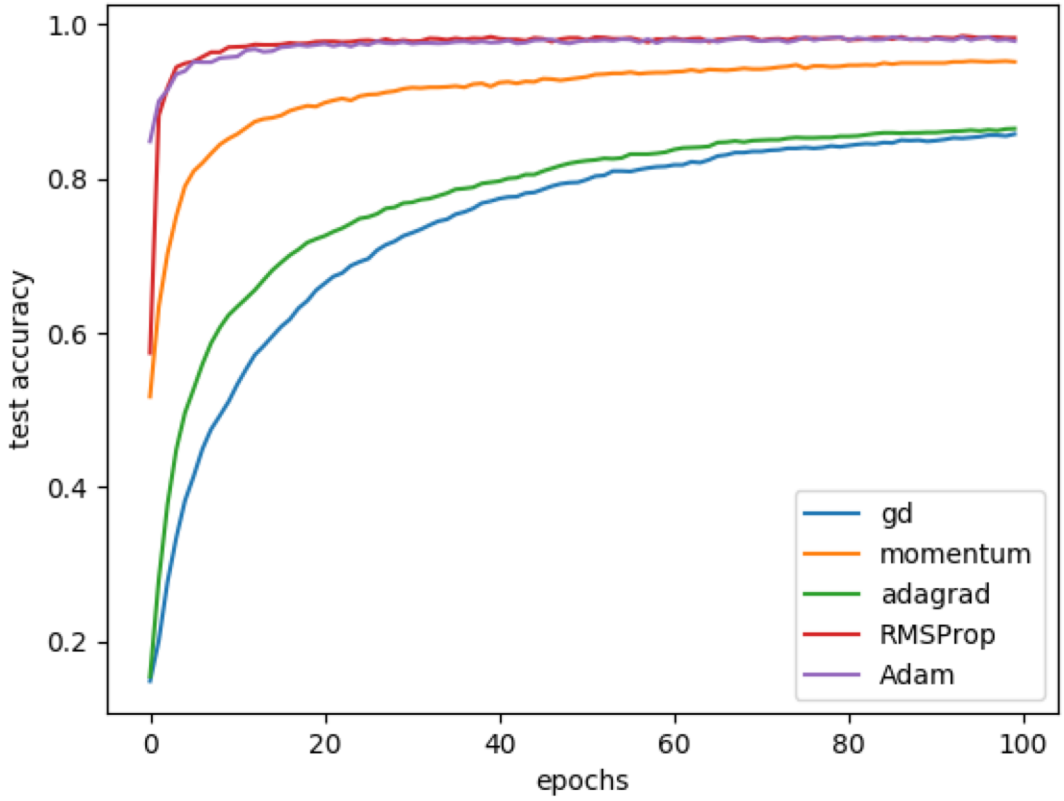
Adams optimizer combines RMSProp and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1}$$
$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

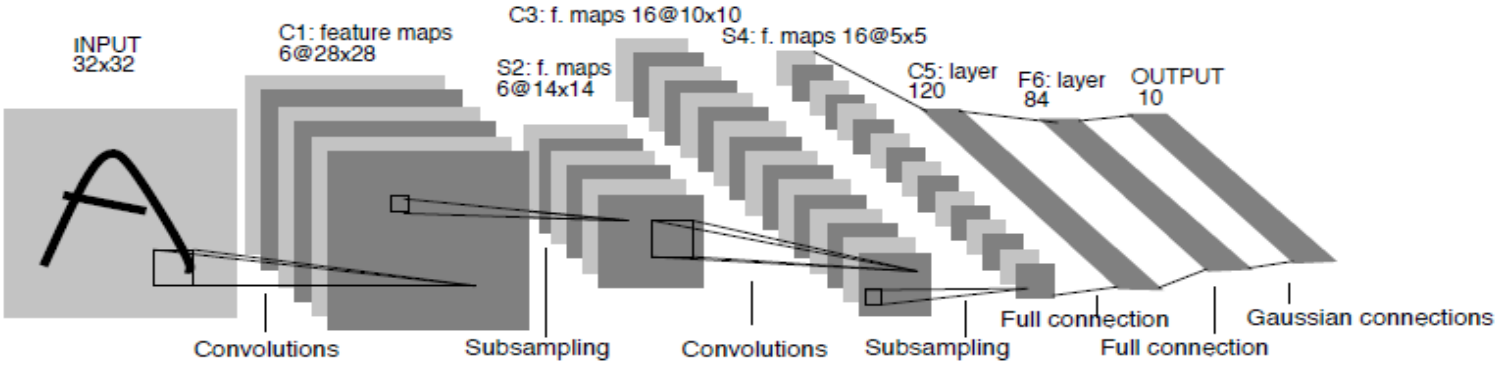
Note that $\boldsymbol{s}$ adds the momentum and $\boldsymbol{r}$ contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, and $\varepsilon = 10^{-8}$.

Example 4: MNIST recognition with CNN with different learning algorithms



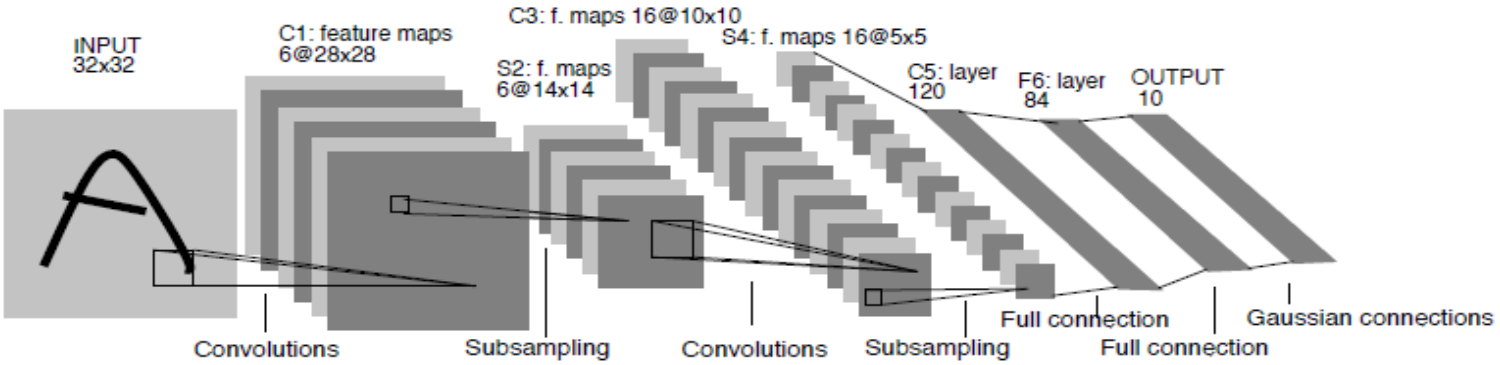
LeNet 5, LeCun et al. 1988



Input size 32x32 images
C: convolution layer, S: subsampling layer, F: fully-connected layer
Pixel values were normalized to mean = 0 and s.d. = 1.0

C1: convolution layer of 6 feature maps of size 28x28
each neuron in C1 has a 5x5 receptive field in the input layer
(5x5+1)x6 = 156 parameters to learn
connections = 28x28x(5x5+1)x6=122304

S2: subsampling layer of 6 feature maps of size 14x14
2x2 non-overlapping receptive fields in C1
two trainable parameters: scale and bias; 12 parameters to learn
Connections = 14x14x(2x2+1)x6 = 5880



C3: convolution layer of 16 feature maps of size 10x10
each neuron connects to several 5x5 receptive fields of same location
1516 parameters to learn
151600 connections

S4: Subsampling layer of 16 feature maps of size 5x5
Each neuron connects to 2x2 receptive field at C3
16x2 parameters to learn
5x5x(2x2+1)x16 = 2000 connections

C5: Convolution layer of 120 feature maps of 1x1 size
Each neuron of C5 is connected to 16 5x5 receptive fields in S4
48120 trainable parameters and connections (fully-connected)

F5: 84 fully-connected neurons
Output layer: 10 RBF neurons

AlexNet, Krizhevsky et al. 2012

Images from ImageNet:

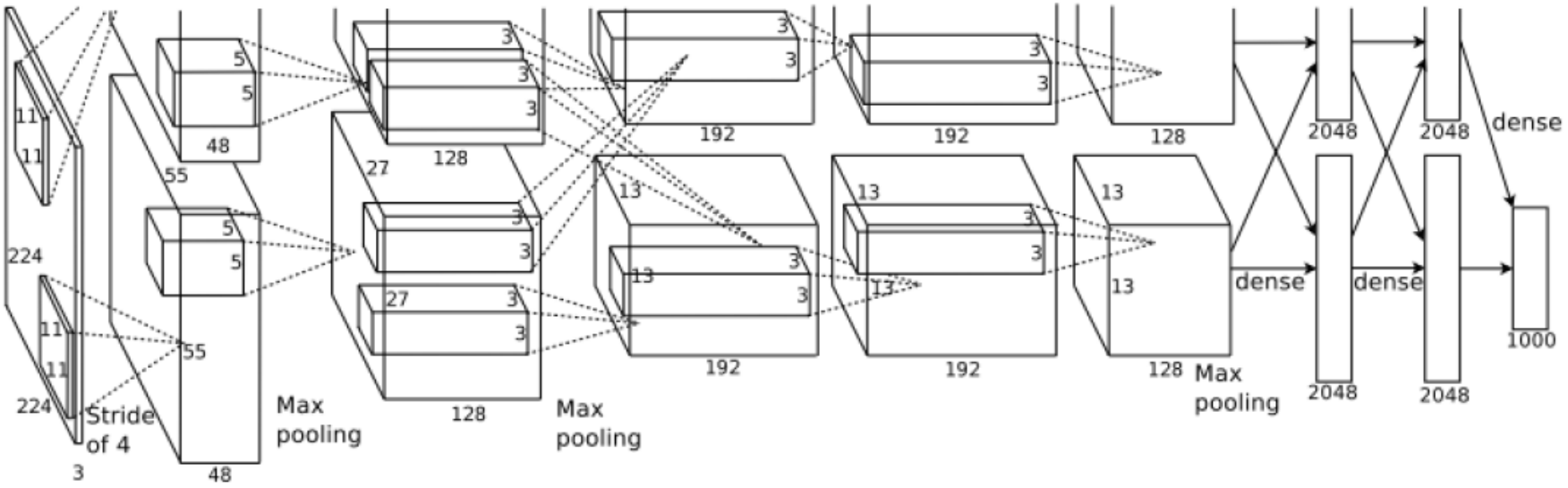
http://www.image-net.org/index

15 million images and 22K categories
Images collected from web, manual labeling

Alexnet wase tested on 1.2K training images, 1000 categories,
50,000 validation images, 150,000 test images

RBG images: each image consists of three color image channels
Input images were scaled to 224x224 size.

AlexNet architecture



Input layer: size 224x224x3

First convolution layer: 96 kernels of size 11x11x3, with a stride of 4 pixels (i.e., the distance between two receptive field centers of two neurons)

Second pooling layer: Max pooling layer of 5x5 receptive field

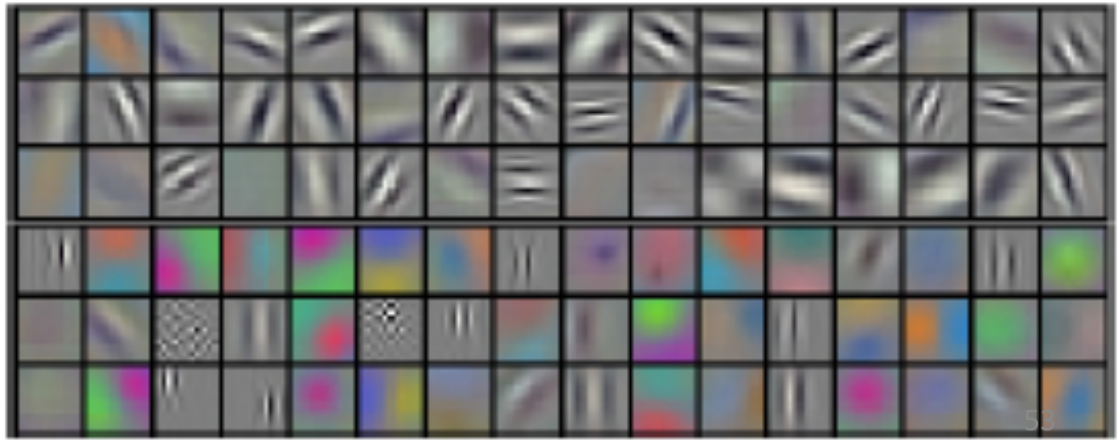
AlexNet architecture

Trained with stochastic gradient of batch size 128
On two NVIDIA GTX 580 3GB GPU
Trained for about a week

650,000 neurons
60 million parameters
630 million connections
5 convolution and pooling layers and 3 fully connected layers

Top 1 test error: 37.5%
Top 5 test error: 17.0%

96 kernels learned by first convolution layer; 48 kernels were learned by each GPU

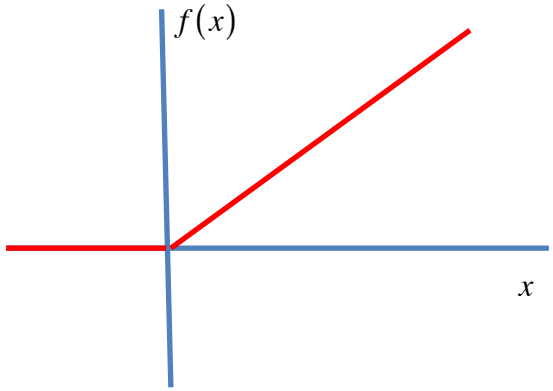


AlexNet features

Rectified Linear Unit (ReLU):
Activation function of neurons is given by

$$f(x) = \max(0, x)$$

About 6 time faster compared to sigmoidal.



Data augmentation:
Enlarge the data set artificially by applying label preserving transformations: image translations, image reflections, changing RGB intensities

Dropouts:
Output of the neurons were set to zero with probability 0.5; neurons which were dropped out did not participate in the forward activation propagation and backward error propagation.
Dropouts allowed more robust features to learn; approximately doubles the number of iterations

AlexNet training

Learning equations:

$$\boldsymbol{V} \leftarrow \gamma \boldsymbol{V} - \alpha \left(\frac{\partial J}{\partial \boldsymbol{W}} + \beta \boldsymbol{W} \right)$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{V}$$

Learning parameter α = 0.1 (initially)
Momentum parameter ϒ = 0.9
Decay parameter β=0.0005 (also called L_2 regularization factor)
Batch size = 128