CHAPTER II

Digital Imagery

Digital image concepts

Digital images are electronic shots of a scene or scanned from documents. Each pixel is depicted by an intensity value, which is represented in binary code.

The density of pixels in an image is called ‘resolution’. With the same image size, the more pixels that we keep to describe an image, the more detailed the image.

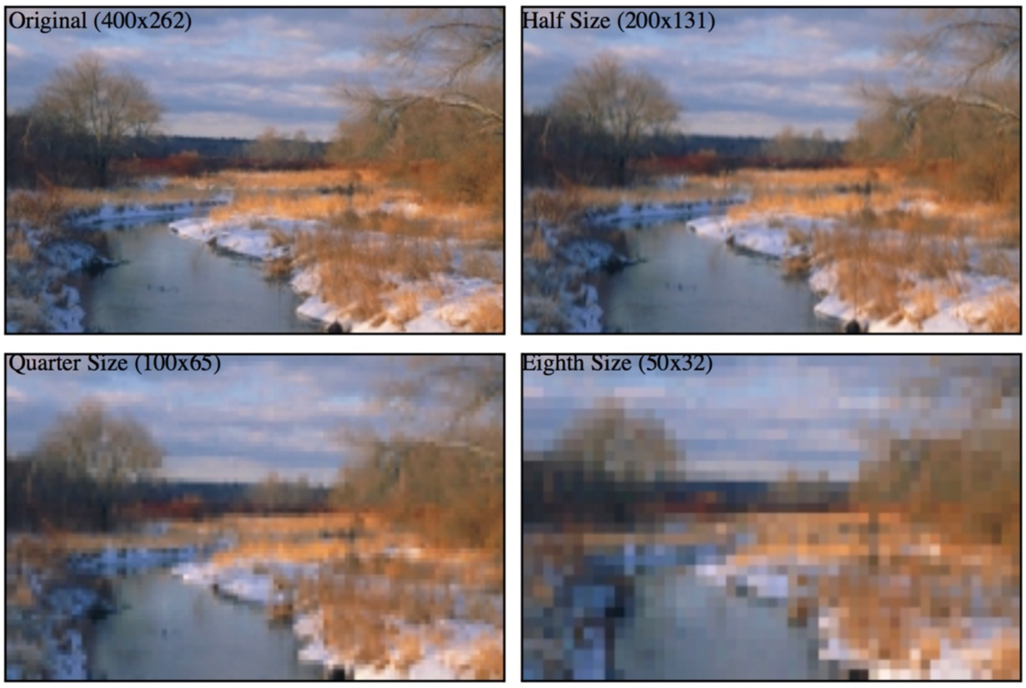


Figure 2.1.1 1 what happens as we reduce the resolution of an image while keeping its size the same. Image from Digital Image Basics by Jonathan Sachs

A monochrome image can be described in terms of a two-dimensional light intensity function f(x,y), where the amplitude of f(x,y) is the intensity (brightness) of the image at position (x,y). The intensity of a monochrome image lies in the range

$$L_{\min} < f(x,y) < L_{max} \quad (2.1.1.1)$$

where the interval $[L_{\min}, L_{max}]$ is called the grey scale. There are two common grey scale storage methods: 8-bit storage and 1-bit storage.

The most common storage method is 8-bit storage. It can represent up to 2^8colours for each pixel. The grey scale interval is [0,255], with 0 being black and 255 being white.

The less common storage method is 1-bit storage. There are two grey levels, with 0 being black and 1 being white.

Figure 2.1.1 2 256-level grey scale. Image from Digital Image Basics by Jonathan Sachs

A colored image can be represented using multi-channel color models. The most widely used model is the RGB. Colored images are made up of three primary spectrums: red, green and blue (RGB). These three primary spectrums together create a three-dimensional color space where red defining one axis, green defining the second, and blue defining the third. Every existing color is described as a mixture of red, green, and blue light and located somewhere within the color space.

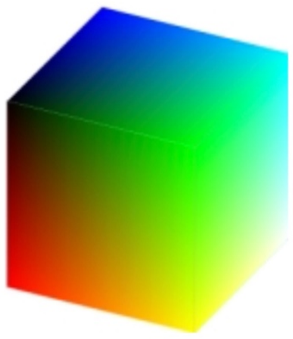


Figure 2.1.1 3 color space. Image from Digital Image Basics by Jonathan Sachs

Using RGB model, a colored image can be described using a three-channel intensity function

$$\mathrm{I}_{\mathrm{RGB}}(\mathrm{x},\mathrm{y}) = (\mathrm{F}_{\mathrm{R}}(\mathrm{x},\mathrm{y}), \mathrm{F}_{\mathrm{G}}(\mathrm{x},\mathrm{y}), \mathrm{F}_{\mathrm{B}}(\mathrm{x},\mathrm{y})) \quad (2.1.1.2)$$

where $\mathrm{F}_{\mathrm{R}}(\mathrm{x},\mathrm{y})$ is the intensity at position (x,y) in the red channel, $\mathrm{F}_{\mathrm{G}}(\mathrm{x},\mathrm{y})$ is the intensity at position (x,y) in the green channel, and $\mathrm{F}_{\mathrm{B}}(\mathrm{x},\mathrm{y})$ is the intensity at position (x,y) in the blue channel. Each channel usually uses an 8-bit storage which can describe up to 2^8 colours. Thus, computers commonly use a 24-bit storage to describe the intensity at position (x,y), which can describe up to 2^{24} colours.

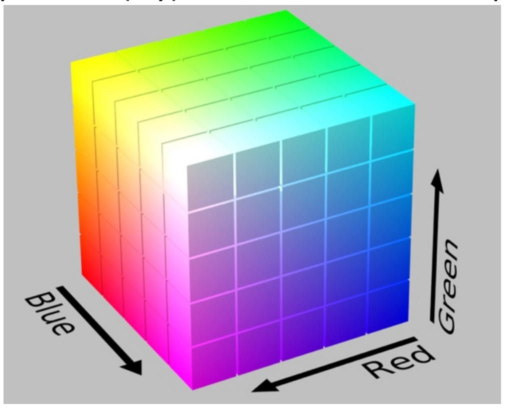


Figure 2.1.1 4 color intensity. Image from Wikipedia

Convolution

Convolution is an image transformation technique using neighborhood (or area-based) operators. The objective of image convolution is to process an input image in order to enhance some important features that is suitable for a specific application. It has two main approaches: spatial-domain approach and frequency-domain approach.

Spatial-domain and frequency-domain approach can both be described in terms of convolutional function:

$$g(x,y) = h(x,y)*f(x,y) \quad (2.1.2.1)$$

given f(x,y) is the output image, h(x,y) is the invariant operator and g(x,y) is the output image.

In spatial-domain, we look at the value of each pixel varies with respect to scene whereas in frequency-domain, we look at the rate at which the pixel values change in spatial-domain. Thus, in frequency-domain approach, we will have to convert the pixel values into frequency-domain before applying convolution, then convert the result back into spatial-domain. We will mainly discuss about spatial-domain procedure in this section.

The procedure for convolution in spatial-domain is as follows: A filter (sometimes can be referred to as a mask, a kernel or a window) centered at point (u,v) is flipped in both dimensions and then slit around the input image. Each time the filter is placed at a new position, every pixel of the input image contained within the filter is multiplied by the corresponding filter coefficient and then summed together. The result from each multiplication and summation declares the next pixel of the output image. The described procedure is repeated this until all values of the image has been calculated. It is mathematically described as:

$$G(i,j) = \Sigma\ \Sigma\ H(u,v)F(i-u,j-v) \quad (2.1.2.2)$$

where F is the output image, H is the filter and G is the output image.

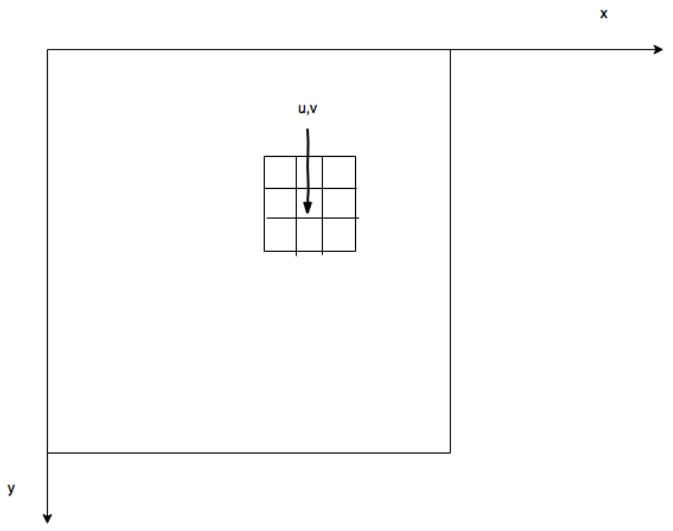


Figure 2.1.2. 1 a 3x3 neighborhood about point (u,v) in an image

Below we will give two simple examples to illustrate the application of convolutional filtering.

The first application is to highlight edges in an image. Some known edge enhancement filters are Prewitt operator, Sobel operator, Robinson compass masks, Krisch compass Masks and Laplacian operator. The decision on which filter to use depends on our desired results. Here we will describe the use of Sobel edge detection. A Sobel filter is used to calculate edges in both horizontal and vertical direction.

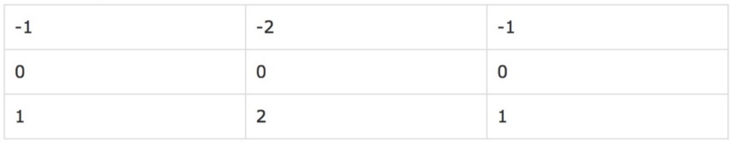
The vertical Mask of Sobel operator is as follows:

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

Figure 2.1.2. 2 the vertical Mask of Sobel operator. Image from tutorial point-Sober operator

The pixels at the corresponding position to the area declared by this filter of the input image are respectively multiply by -1,0,1,-2,0,2,-1,0 and 1. The results of these nine multiplications are then summed. This process is repeated until all values of the image has been calculated.

The horizontal Mask of Sobel operator is as follows:



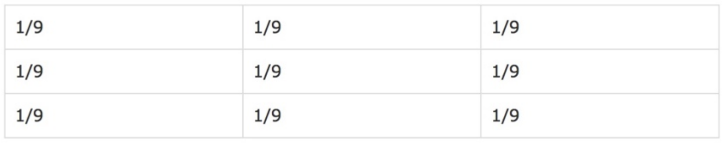
-1	-2	-1
0	0	0
1	2	1

Figure 2.1.2. 3 the horizontal Mask of Sobel operator. Image from tutorial point- Sober operator

Similarly, the pixels at the corresponding position to the area declared by this filter of the input image are respectively multiply by -1,-2,-1,0,0,0,1,2 and 1. The results of these nine multiplications are then summed. This process is repeated until all values of the image has been calculated.

This give more weight age to the pixel values around the edge region. It thus increases the edges intensities. As a result, the output image edges become enhanced comparatively to the original image.

The first application is to blur an image. There are three common type of filters that are used to perform blurring: Mean filter, weighted average filter and Gaussian filter. We are going to discuss about mean filter. In a mean filter, there are an odd number of filter elements, all of which are the same and can be summed to one. For example, a 3x3 can be declared as followed:



1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 2.1.2. 4 the 3x3 mean filter. Image from tutorial point- concept of blurring

The pixels at the corresponding position to the area declared by this filter of the input image are respectively multiply by 1/9. The results of these nine multiplications are then summed.

This operation can be used to discard false effects that may be present in a digital image as a result of poor sampling system or transmitting channel. This process is repeated until all values of the image has been calculated.

Machine learning

Core concepts of machine learning

Machine learning studies computers' abilities to automatically recognize patterns and generate intelligent conclusion from the given data. It involves two learning types: supervised and unsupervised.

Supervised learning (SVM) is equivalent to data classification. Computers learn patterns from the labeled examples then use them to make intelligent classification on the unknown data.

Unsupervised learning (USVM) is equivalent to clustering. Initially, there is no label associated with the data. Computers try to divide the dataset into clusters to discover classes within the data. USVM cannot narrate semantic meaning of the clusters in their learnt models.

Data classification consists of learning step and classification step. Computers build a classification model by studying the **training set** made up of pre-labeled tuples. Each tuple X is represented by vector of n-dimension X = (x_1, x_2,..., x_n) and a class label attribute x_c. The class label attribute has a discrete-valued that indicates which class tuple X tuple belongs.

In the classification phrase, the learned model is used for predicting the class label for given data. To avoid **overfitting** (i.e. Overfitting describes the problem when the model is tailored to fit the random noise in one specific sample rather than reflecting the overall population), a **testing set** is used. Each testing set is also made up of pre-labeled tuples and is independent of training sets.

The given dataset is divided into training and testing sets using one of these three following methods:

Hold out: The given data set is divided into two independent sets: training set and testing set. Two-thirds of the data are in the training set and one-third of the data is in the testing set.

k-fold cross-validation: The data set is divided into k subsets, and the holdout method is repeated k times. Each time, one of the k subsets is used as the test set and the other k-1 subsets are put together to form a training set. Then the average error across all k trials is computed. The advantage of this method is that it matters less how the data gets divided.

Bootstrap: A common bootstrap method is the .632 bootstrap. Given a set of d tuples. This dataset will be sample d times with replacement. Each time a tuple is selected, it is re-added into the data pool and likely to be selected again. The data that do not make it into the training set will eventually be added into the testing set. The training set and the testing set are not independent. In .632 bootstrap, 63.2% of the dataset will end up in the bootstrap sample, and the 36.8% that did not make it to the bootstrap sample will together form the test set.

Multilayer feed-forward neural networks (MFNN)

Overviews

MFNN is a classic machine learning algorithm that simulates the way human brain works. MFNN is very useful for studying large datasets due to its ability to tolerate noise data as well as to recognize patterns on which they have little knowledge. It is especially efficient for real world data, where we do not know much about the relationships between attributes and classes. Generally, MFNN requires a large amount of data in order to well perform.

Each MFNN consists of an input layer, one more hidden layers and an output layer. Layers are connected in acyclic graph. Each layer is made up of neurons. Neurons between two adjacent layers are pairwise connected, but neurons in one layer share no connection. No direct connection exits between input and output layer. Cycles are

not allowed.
Inputs are fed into the neurons making up the input layer. The outputs produced by this layer are weighted and passed simultaneously to the first hidden layer. This hidden layer outputs are again weighted and input to an another hidden layer and so on. It is arbitrary how many hidden layers there should be, but normally we only use one. The weighted outputs of the last hidden layer are input to the output layer, where the prediction for the given tuples will be produced.
Neurons in the input layer are 'input units'. Neurons in the hidden layers and the output layers are called 'neurodes' or sometimes referred to as 'output units'. The number of input units are not necessarily equal number of input units. There can be more or less number of hidden units than number of input or output units. Each output unit applies a nonlinear (activation) function to its input. The activation function will be described in section 2.2.3.4. The output is suggested as in the following function:

$$a_k^l = f(\sum_{j=1}^{m} w_{jk} a_j^{l-1} + b_k^l) \qquad (2.2.2\ 1)$$

Where f is the neural activation function, a_k^l is output of neuron k at layer l, a_j^{l-1} is output of neuron j at layer (l-1), w_{jk} is the weight of the connection between node j and k, b_k^l is the bias of node k.

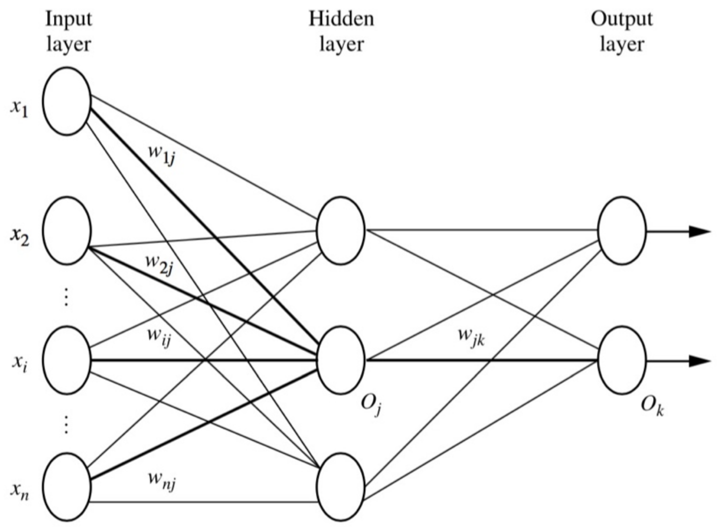


Figure 2.2.2 1 fully connected layers. Photo from Data Mining Concepts and Techniques, 3rd edition, Jiawei Han & Micheline Kamber

Back propagation

Back propagation is the learning algorithm of neural networks. Total error is the summation of error at each output unit, which can be calculated using the squared error function:

$$E = \sum_N \frac{1}{2}(y_{o_N}^o - a_{o_N}^o)^2 \qquad (2.2.2.2)$$

Where $a_{o_N}^o = \frac{1}{1+e^{-z_{o_N}^o}}$

The error rate at each output unit can thus be calculated from this total error:

$$\delta^o_{o_N} = \frac{\partial E}{\partial z^o_{o_N}} = \frac{\partial E}{\partial a^o_{o_N}} * \frac{\partial a^o_{o_N}}{\partial z^o_{o_N}} = -(y^o_{o_N} - a^o_{o_N})a^o_{o_N}(1 - a^o_{o_N}) \qquad (2.2.2.3)$$

This error is then propagated back to the networks:

$$\delta^l_{z_j} = \frac{\partial E}{\partial a^l_j} * \frac{\partial a^l_j}{\partial z^l_j} = \frac{\partial E}{\partial z^{l+1}_i} * \frac{\partial z^{l+1}_i}{\partial a^l_j} * \frac{\partial a^l_j}{\partial z^l_j} = \left(\sum\nolimits^l_{0\le i'\le m} \delta^{l+1}_{z_{i'}} * w^{l+1}_{ji'}\right) * \frac{\partial a^l_j}{\partial z^l_j} \qquad (2.2.2.4)$$

With learning rate η, this error will affect the corresponding weight by an amount of:

$$w^l_{ij} = w^l_{ij} \pm \eta \Delta w^l_{ij} \qquad (2.2.2.5)$$

where

$$\Delta w^l_{ij} = \frac{\partial E}{\partial w^l_{ij}} = \frac{\partial E}{\partial a^o_j} * \frac{\partial a^l_j}{\partial z^o_j} * \frac{\partial z^l_j}{\partial w^l_{ij}} = \delta^l_{z_j} a^{l-1}_i \qquad (2.2.2.6)$$

Convolutional neural networks

Overviews

CNN is a “state-of-the-art technique for image recognition”[5]. It is a multilayer neural networks, consists of one or more convolutional layers, followed by subsampling layers (sometimes called pooling layers), and one or more fully connected layers.

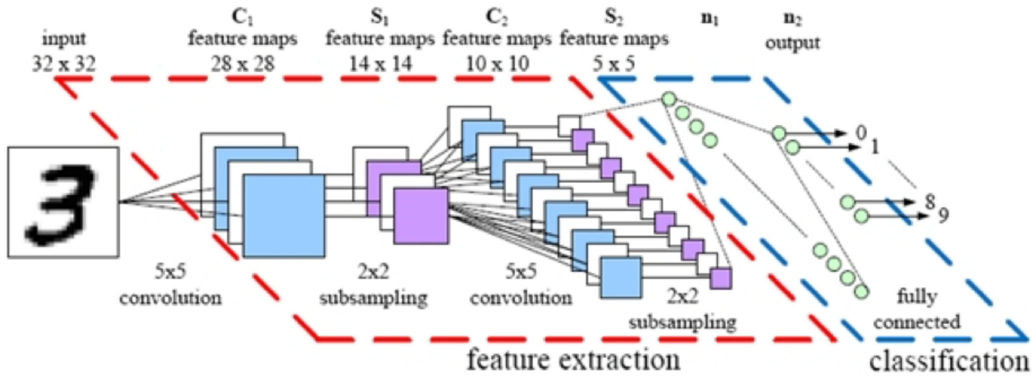


Figure2.2.3 1 a convolutional neural networks. Image by Eindhoven University of Technology – PARsE

Convolutional layer

Four parameters are required in convolutional layer: the number of layer K, receptive field size F, the stride S and the amount of zero padding P. This layer accepts input of volume size $Width_{in}Height_{in}Dimension_{in}$ and produces an output of volume size $[(Width_{in} - F + 2P)/S + 1][(Height_{in} - F + 2P)/S + 1][K]$.

Local receptive field

Think of the input image as a square of n x n neurons. The value of each neuron is the corresponding pixel intensity of the input image. We will map a localized region of the input neurons to a neuron in the hidden layer. These localized regions of input neuron are called local receptive fields.

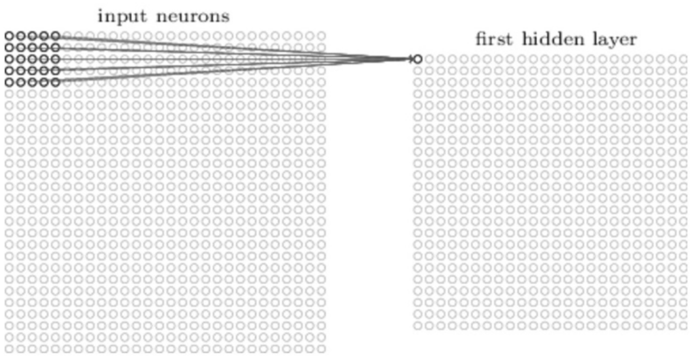


Figure 2.2.3 2 local receptive fields. Image from Deep learning - draft book in preparation, by Yoshua Bengio, Ian Goodfellow, and Aaron Courville

Filter (Kernel)

Each mapping from input layer to a hidden neuron at the hidden layer next to it learns a weight $w_{a,b}^{l}$ and each corresponding hidden neuron leans a bias $b_{x,y}^{l}$. This weight and bias together make up a kernel (sometimes called a filter). The kernel value is shared among all neurons in the same hidden layer. The output generated by the x^{th}, y^{th} hidden neuron is a convolutional function between an input neuron at position $(x-a, y-b)^{th}$'s value and its weight:

$$a_{x,y}^{l} = \sigma\left(\sum_a \sum_b w_{a,b}^{l}\, a_{x-a,y-b}^{l-1} + b_{x,y}^{l}\right) \quad (2.2.3.1)$$

Where σ is neural activation function, $a_{x,y}^{l}$is the real output value at position (x,y), $a_{x-a,y-b}^{l-1}$ is the input pixel value at position (x-a, y-b), $b_{x,y}^{l}$ is the shared value for the bias, $w_{a,b}^{l}$ the weight of the connection at the a^{th} row and the b^{th} column of the receptive field, where the weight is pre-inversed (as discussed in **Convolution** section).

Feature map and stride

The combination of outputs generated by a hidden layer is called a feature map. To calculate the value of the next pixel in the feature map, will move our kernel k pixel along both width and height consequently (in which case we would say a stride length of k is being used) and re-apply the convolution function discussed above, until we run out of input neuron.

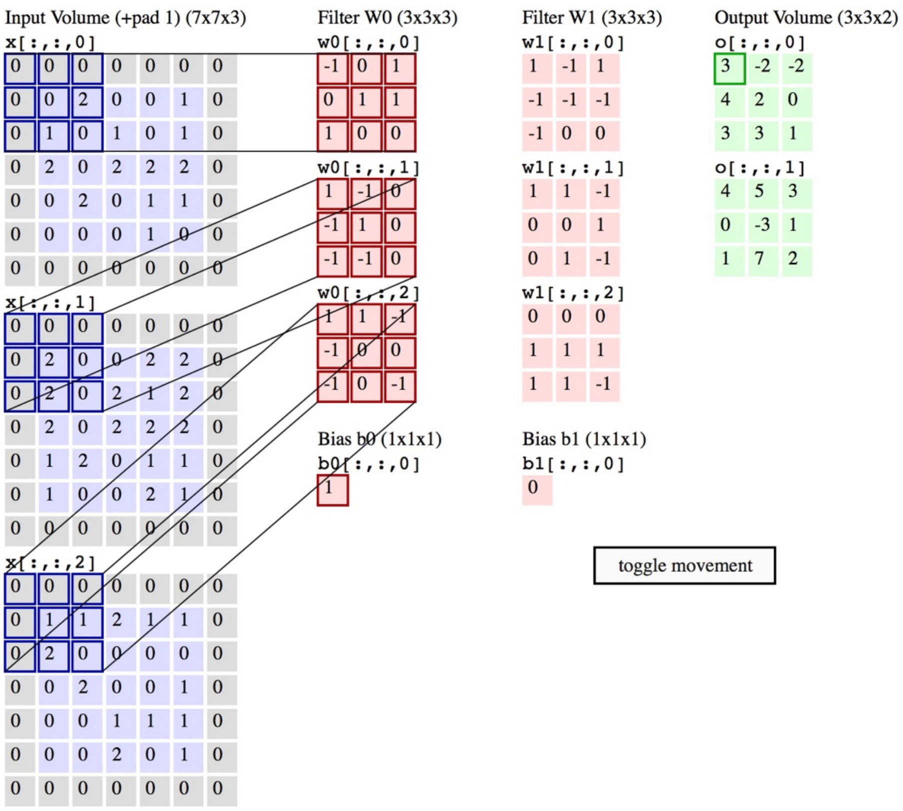


Figure 2.2.3 3 convolutional process - An input image of size 7x7x3 is filtered by 2 3×3x3 convolutional kernels with stride 2 which create 2 feature maps. Image by Stanford University – Convolutional neural networks for visual recognition.

Each feature map can detect one feature at different locations of the same input image. To detect many features from one input image, we need several features maps. To generate several feature maps, we will need several kernels. A complete convolutional layer consists of several different features maps.

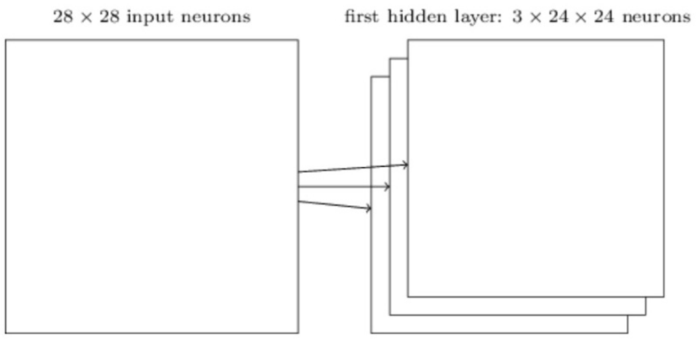


Figure 2.2.3 4 the first hidden layer consists of several features maps. Image from Deep learning - draft book in preparation, by Yoshua Bengio, Ian Goodfellow, and Aaron Courville

The use of zero-padding

Zero padding is a simple method of padding the borders of the input image in order to control the dimensionality of the output image. Specifically, the relationship among padding size P, input height/length W, output height/length O and filter size K can be described as follow:

$$O = \frac{W - K + 2P}{2} + 1 \qquad (2.2.3.2)$$

ReLU non-linearity

A standard way to model a neuron output is with tanh function:

$$f(x) = (1 + e^{-x})^{-1} \quad (2.2.3.3)$$

Rectified Linear Unit is a modern way to represent an output f as a function of input x. It computes the function:

$$f(x) = \max(0,x) \qquad (2.2.3.4)$$

$$\text{Where } x = wx + b \qquad (2.2.3.5)$$

which means the output is 0 when the input is less than 0 and the output is a linear with slope 1 when x the input is greater than 0.

ReLU has two advantages over tanh function. Deep learning neural networks with ReLU was found to train significantly faster than networks with tanh unit. This is because saturating nonlinearities are generally slower than non-saturating nonlinearities in terms of training time with gradient descent. Secondly, while tanh function involves expensive operations, ReLU can be implemented easily by thresholding a matrix of activations at zero. A convolutional layer is often followed by ReLU.

Pooling layer

Pooling layer requires two parameters: filter size F and stride S. It accepts input with volume size $Width_{in}Height_{in}Dimension_{in}$ and produces an output of volume size $[(Width_{in} - F)/S + 1][\,(Height_{in} - F)/S + 1][Dimension_{in}]$.

A convolutional layer is usually followed by a pooling layer. Pooling layer's function is to simplify every feature map in the previous layer at every depth slide spatially. Thus it reduces the amount of parameters, computation in the network and therefore overfitting. A successful pooling layer should be able to preserve critical information while being "invariant to troublesome deformations" [9].

Pooling is done via a summary statistic over a region of neurons in the preceding layer. The summary statistic in l_i is defined by the l_i norm of inputs in the pools. If node $a^{l-1}_{x+p,y+q}$ (where $0 \leq p, q \leq k$) from the preceding layer are in the pool, the output of the pooling process for each pixel at position (x,y) can be mathematically represented as:

$$a^l_{x,y} = (\textstyle\sum_{0\leq p,q\leq k} |a^{l-1}_{x+p,y+q}|^i)^{1/i} \quad (2.2.3.6)$$

Two widely used pooling techniques can be derived from function (2.2.3.6). The first one is **max pooling**. This is when p → ∞

$$a^l_{x,y} = max_{0\leq p,q\leq k}(a^{l-1}_{x+p,y+q}) \qquad (2.2.3.7)$$

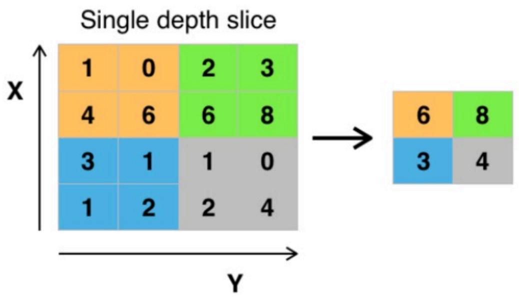


Figure 2.2.3 5 pooling layer – An input single depth slice of size 4 x 4 is filtered by a 2x2 kernel with stride 2 which results in a pooling feature map of size 2 x 2. Image by Stanford University – Convolutional neural networks for visual recognition.

When p = 2, we have $\boldsymbol{l_2}$ **pooling**:

$$a_{x,y}^{l} = (\textstyle\sum_{0\le p,q\le k} |a_{x+p,y+q}^{l-1}|^2)^{1/2} \quad (2.2.3.8)$$

A filter size of 2x2 and a stride of length 2 are often applied with these two pooling techniques. **Average pooling** (when P = 1) was often used historically but has recently fallen out of favor because it gives less competitive results than the other two.

An another known pooling technique is called **subsampling**. The output of subsampling function is given by:

$$a_{x,y}^{l} = \tanh\,(\beta \frac{\sum_{0\le p,q\le k} a_{x+p,y+q}^{l-1}}{kxk} + b_{x,y}^{l}) \quad (2.2.3.9)$$

Where β is a trainable scalar, b is a bias, kxk is the size of the receptive field. According to some research [4,9], there is no best pooling technique. We may need to try applying several pooling techniques to our problem to see which one yields the best result.

Overlap pooling

A. Krizhevsky et al. has taken a step further to the traditional pooling [1]. Suppose a pooling layer consists of a grid of pooling units, each summarizes a neighborhood of size z x z and stride s. Traditionally, we set z = s, where we obtained non overlap pooling. Now we will set s < z to obtain overlap pooling. A. Krizhevsky et al. has pointed out this is a more effective pooling technique as it reduces the top-1 error rate by 0.4% and top-5 error rate by 0.3%.

Exploiting viewpoints in pooling

This approach was introduced by Sander et al. in 2014. After preprocessing and data augmentation, viewpoints are extracted by the combination of flipping, rotating, cropping to the input image. Notice that the combination of those operations is necessary because it reduces redundancy. All viewpoints are presented to the same convolutional architecture. The resulting feature maps from each viewpoint are first concatenated, then processed by a set of fully connected layer to obtain predictions. The benefit of exploiting viewpoints in pooling is that it allows the network to “look at” the image at different angles.

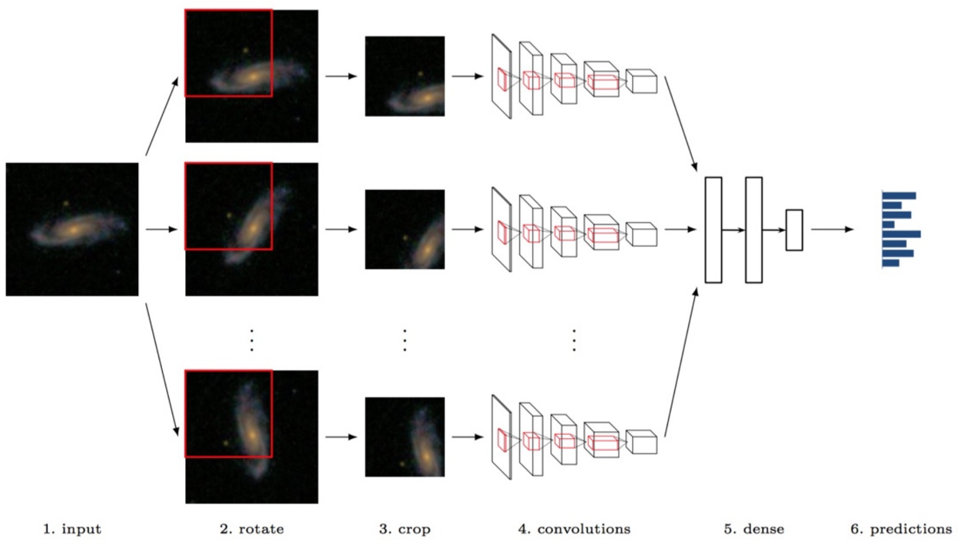


Figure 2.2.3 6 overview viewpoints exploitation in pooling. Different viewpoints are extracted from the original image (2,3). Each viewpoint is fed to a separated convolutional architecture (4). Results generated by each convolutional architecture are concatenated and feed into dense layers (5) to obtain prediction (6). Image from Sander et al.

Fully connected layer

Flatten output from the last pooling layer is proceeded by a set of fully connected layers. Fully connected layers are originated from multi-layers feed forward neural networks. More details on MFNN has been discussed in section 2.2.2.

Softmax regression layer

Softmax regression is a generalized logistic regression model which is used to turn a single neuron into a linear classifier so that the neuron can handle multiple classes. For each input a_i^{l-1}, the real output a_j^l can be computed. This real output a_j^l is then used to interpret in which of the K classes does the input a_i^{l-1} belongs

$$a_j^l = P\left(y_i^{l-1} = k \middle| a_i^{l-1}; w\right) = \frac{e^{(w_{ij}a_i^{l-1}+b_j^l)}}{\sum_{k=1}^{K} e^{(w_{ik}a_i^{l-1}+b_j^l)}} = \frac{z_j^l}{\sum_{k=1}^{K} e^{(w_{ik}a_i^{l-1}+b_k^l)}} \qquad (2.2.3.10)$$

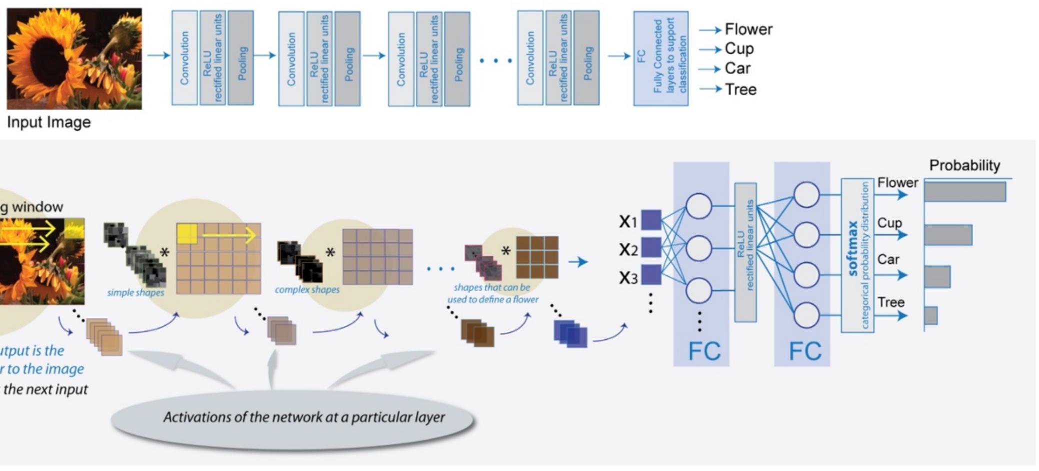
where y_i^{l-1} represents the predicted class k for input a_i^{l-1}, K is the number of classes at output layer, $(w_{ij}a_i^{l-1} + b_j^l)$ is the output produced by activated equation given the input $a_i^{l-1}$and and the weight of its connection to class k, $(w_{ik}a_i^{l-1} + b_j^l)$ is the output produced by activated equation given the input a_i^{l-1} and the weight of its connection to each output class j, where j = 1,…,K

This function can map an input to an output in a range between 0 and 1, and all calculated probabilities of different classes must sum to 1.

Back propagation

Back propagation is a process to adjust the learnt weight and bias by comparing the network's prediction with the tuple's known target value. The aim is to minimize the error between the network's prediction and the known target value. The target value can either be a known class label or a continuous value. The weight and bias modification process is done in a backward direction, from softmax layer through fully connected, pooling and ReLU layers to the convolutional layer. The forward and backward process is repeated until all the weights in the network converges. The steps are as follows:

1. Initialize all weights and bias in the network with some small random values and choose a learning rate.
2. Propagate the inputs forward the networks using our initialized weights and bias until we reach the output layer.
3. Calculate error rate and consequently update the weight at each layer.
4. Repeat the process until when one of the following conditions is reached:
    - All Δw in the previous epoch are bellow some pre-specified threshold
    - A pre-specified number of epochs has reached
    - The percentage of misclassified tuples in the previous run is lower than some predefined threshold



Input Image
Convolution
ReLU rectified linear units
Pooling
FC Fully Connected layers to support classification
Flower
Cup
Car
Tree
Sliding window
Filters
light and dark
simple shapes
complex shapes
shapes that can be used to define a flower
Every feature map output is the result of applying a filter to the image
The new feature map is the next input
Activations of the network at a particular layer
X1
X2
X3
FC
FC
softmax
categorical probability distribution
Probability

Figure 2.2.3 7 layers of convolutional neural networks. Image by Mathworks, Convolutional neural networks documentation

Softmax layer

The total error E_{total} (will be refer to later on as E) is the sum of errors for each output neuron and can be calculated using log-loss (sometimes called cross entropy) function:

$$E_{total} = \frac{-1}{N}\left(\sum_{k=1}^{N} y_k(\log a_k) + (1 - y_k)\log(1 - a_k)\right) \quad (2.2.3.11)$$

where N is the number of output neurons, y_k is the expected output of neuron k and z_k is the real output of neuron k.

We will look at how the networks adjust its weight using E_{total} by looking at E_{total} effect on each layer.

Fully connected layer (Neural networks)

The real output a_j^l at fully connected layer can be expressed in terms of net output z_j^l by function () in section

$$a_j^l = \frac{e^{(w_{ij}a_i^{l-1}+b_j^l)}}{\sum_{k=1}^{K} e^{(w_{ik}a_i^{l-1}+b_j^l)}} = \frac{z_j^l}{\sum_{k=1}^{K} e^{(w_{ik}a_i^{l-1}+b_k^l)}} \quad (2.2.3.12)$$

Therefore, we can calculate $\frac{\partial a_j^l}{\partial z_j^l} = \frac{\sum_{k=1}^{K} e^{(w_{ij}a_i^{l-1}+b_j^l)} - {z_j^l}^2}{{e^{(w_{ij}a_i^{l-1}+b_j^l)}}^2}$ (2.2.3.13)

Each neuron at the output layer has a net value defined by the following function

$$z_k^l = \sum_{j=1}^{m} w_{jk} a_j^{l-1} + b_k^l \quad (2.2.3.14)$$

where z_k^l is the net output of neuron k at layer l, a_j^{l-1} is the real output of neuron j at layer (l-1) where j = {1,2,3,…,m}, w_{jk} is the weight of the connection between neuron j and k, b_k^l is the bias value of neuron k.
The amount of error that we want to adjust at each neuron is equal to:

$$\delta_{o_N}^o = \frac{\partial E}{\partial z_{o_N}^o} = \frac{\partial E}{\partial a_{o_N}^o} * \frac{\partial a_{o_N}^o}{\partial z_{o_N}^o} \quad \text{(as a result of chain rule)} \quad (2.2.3.15)$$

where N is the number of output neurons, o stands for output layer, $\delta_{o_N}^o$ is the error rate at neuron N in the output layer, $z_{o_N}^o$ is the net output of neuron N in the input layer, , $a_{o_N}^o$ is the real output of neuron N in the input layer.
We will propagate this error forward. Accordingly, this rate will affect the weight at neuron N by an amount of

$$\Delta w_{jo_N}^o = \frac{\partial E}{\partial w_{jo_N}^o} = \frac{\partial E}{\partial a_{o_N}^o} * \frac{\partial a_{o_N}^o}{\partial z_{o_N}^o} * \frac{\partial z_{o_N}^o}{\partial w_{jo_N}^o} \quad \text{(as a result of chain rule)} \quad (2.2.3.16)$$

where $\Delta w_{jo_N}^o$ is the amount of weight to adjust between neuron j and neuron o_N, $a_{o_N}^o$ is the real output of neuron o_N at output layer, $z_{o_N}^o$ is the net output of neuron o_N at output layer, $w_{jo_N}^o$ is the current weight between neuron j and neuron o_N.
From function () and (), function () can be rewritten as:

$$\Delta w_{jo_N}^o = \frac{\partial E}{\partial w_{jo_N}^o} = \delta_{o_N}^o a_j^{h1} \quad (2.2.3.17)$$

where $\Delta w_{jo_N}^o$ is the amount of weight to adjust between neuron j and neuron o_N, $w_{jo_N}^o$ is the current weight between neuron j and neuron o_N, $\delta_{o_N}^o$ is is the error rate at neuron N in the output layer, a_j^{h1} is the real output of neuron a_j at hidden layer h1 where h1 precedes o.
To decrease the error, we then subtract this value from the current weight

$$w_{jo_N}^o = w_{jo_N}^o \pm \eta * \Delta w_{jo_N}^o \quad (2.2.3.18)$$

where η denotes the learning rate, which is normally set to 0.5

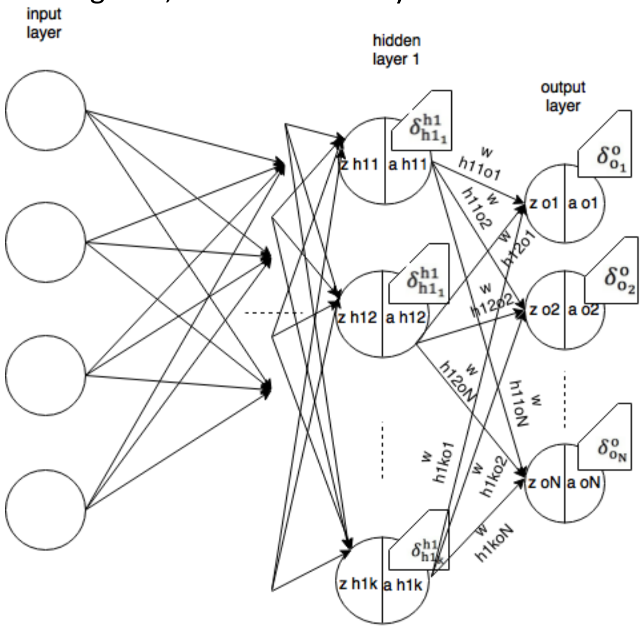


Figure 2.2.3 8 fully connected layer

For every neuron at each hidden and input layer of the neural networks, we can calculate δ and Δw by applying the same technique. Thus our calculation can be generalized as follows:

$$\delta_{z_j}^{l} = \frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial E}{\partial z_i^{l+1}} * \frac{\partial z_i^{l+1}}{\partial a_j^l} * \frac{\partial a_j^l}{\partial z_j^l} = \left(\sum_{0\le i'\le m}^{l} \delta_{z_{i'}}^{l+1} * w_{ji'}^{l+1}\right) * \frac{\partial a_j^l}{\partial z_j^l} \quad (2.2.3.19)$$

where $\delta_{z_j}^{l}$ is the error rate at neuron j in layer l, z_j^l is the net output of neuron j in layer l, a_j^l is the real output of neuron j in layer l , z_i^{l+1} represents the total net output of all neurons in layer (l+1) that are pairwise connected with neuron j, m is the total number of neurons in layer (l+1), $\delta_{z_{i'}}^{l+1}$ is the error rate at neuron i' in layer (l+1) and $w_{ji'}^{l+1}$ is the weight between neuron j and neuron i' where i' = {0,1,…,m}.
We will propagate this error forward

$$\Delta w_{ij}^{l} = \delta_{z_j}^{l} a_i^{l-1} \qquad (2.2.3.19)$$

where Δw_{ij}^{l} is the amount of weight to adjust between neuron i and neuron j, δ_j^l is the error rate at neuron j in layer l, a_i^{l-1} is the real output of neuron i in layer (l-1) .

ReLU layer

The output a_j^l for each neuron can be written as

$$a_j^l = max(0, a_i^{l-1}) \quad (2.2.3.20)$$

where a_i^{l-1} is the real output value of neuron i of ReLU layer (l-1). Thus for each output at layer ReLU layer, we want to adjust an error amount equivalent to

$$\delta_{a_i}^{l-1} = \frac{\partial E}{\partial a_i^l} = \begin{cases} = 0, if\ (a_i^{l-1} \le 0) \\ = \frac{\partial E}{\partial a_j^l} * \frac{\partial a_j^l}{\partial a_i^{l-1}} = \frac{\partial E}{\partial a_j^l} = \delta_{a_i}^{l}\ otherwise \\ (since\ a_j^l = a_i^{l-1}\ in\ this\ case) \end{cases} \quad (2.2.3.21)$$

We need not to adjust weight at this layer because it has no weight.

Max pooling layer

The output a_j^l for each neuron can be represented by

$$a_{x,y}^{l} = max_{0\le p,q\le k}(a_{x+p,y+q}^{l-1}) \qquad (2.2.3.22)$$

where $a_{x,y}^{l}$ is the the real output value of neuron at position (x,y) of max pooling layer l, $a_{x+p,y+q}^{l-1}$ is the real output value of neuron at position (x+p,y+q) of convolutional layer (l-1) where (p,q) is the position in a kernel of size kxk.
For each output pixel at max pooling layer, we want to adjust an amount of

$$\delta_{a_{x+p,y+q}}^{l-1} = \frac{\partial E}{\partial a_{x+p,y+q}^{l-1}} = \begin{cases} = 0, if\ (a_{xy}^{l} \ne a_{x+p,y+q}^{l-1}) \\ = \frac{\partial E}{\partial a_{xy}^{l}} = \delta_{a_{x,y}}^{l}\ otherwise \end{cases} \quad (2.2.3.23)$$

We need not to adjust weight at this layer because it also has no weight.

Convolutional layer

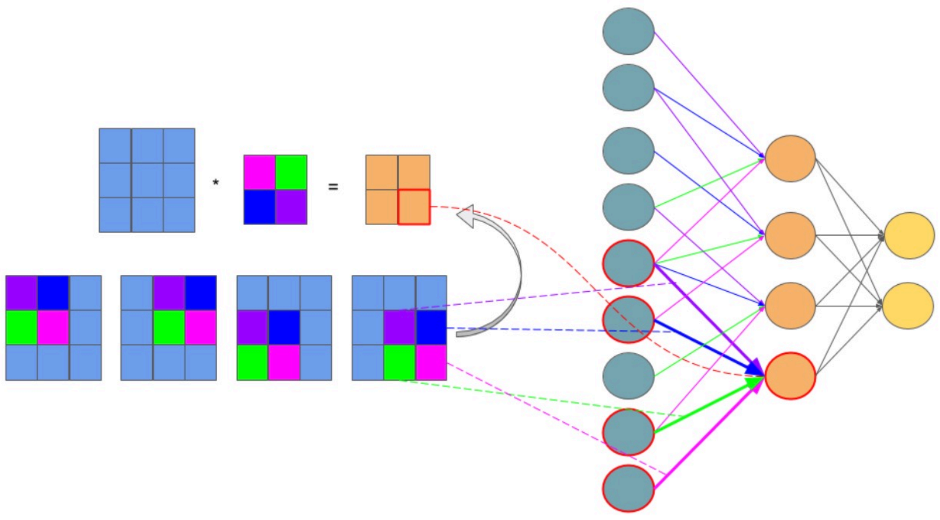


Figure 2.2.3 9 feedforward in CNN is identical with convolution operation.
Image by Grzegorzgwardys

The net output $z^{l+1}_{x,y}$ of each neuron at convolutional layer can be represented by

$$z^{l}_{x,y} = \sum_a \sum_b w^{l}_{a,b}\, a^{l-1}_{x-a,y-b} + b^{l}_{x,y} \quad (2.2.3.24)$$

as discussed in section

The amount of error that we want to adjust at each neuron is equal to:

$$\delta^{l}_{z_{x,y}} = \frac{\partial E}{\partial z^{l}_{x,y}} = \frac{\partial E}{\partial a^{l}_{x,y}} * \frac{\partial a^{l}_{x,y}}{\partial z^{l}_{x,y}} = \frac{\partial E}{\partial z^{l+1}_{i,j}} * \frac{\partial z^{l+1}_{i,j}}{\partial a^{l}_{x,y}} * \frac{\partial a^{l}_{x,y}}{\partial z^{l}_{x,y}} \qquad (2.2.3.25)$$

(as discussed in fully connected layer)

function (2.2.3.25) is thus equivalent to

$$\delta^{l}_{z_{x,y}} = \sum_{i'} \sum_{j'} \left(\frac{\partial E}{\partial z^{l+1}_{i',j'}} * \frac{\partial z^{l+1}_{i',j'}}{\partial z^{l}_{i',j'}} * \frac{\partial a^{l}_{i',j'}}{\partial z^{l}_{i',j'}} \right) = \sum_{i'} \sum_{j'} \left(\delta^{l+1}_{i',j'} * w^{l+1}_{(x,y),(i',j')} * \frac{\partial a^{l}_{i',j'}}{\partial z^{l}_{i',j'}} \right) \quad (2.2.3.26)$$

where $z^{l}_{x,y}$ is net output of pixel at position (x,y) at layer l, $a^{l}_{x,y}$ is real output of pixel at position (x,y) at layer l, $z^{l+1}_{i,j}$ represents the total net output of all pixels at layer (l+1) that are pairwise connected with pixel at position (x,y), $\delta^{l}_{z_{x,y}}$ is the error rate of pixel at position (x,y) at layer l, $\delta^{l+1}_{z_{i',j'}}$ is the error rate of pixel at position (x,y) at layer (l+1), $w^{l+1}_{i',j'}$ is the weight between (i',j') and (x,y), $z^{l}_{i',j'}$ is the net output of pixel (i',j'), $a^{l}_{i',j'}$ is the real output of pixel (i',j').

This error is contributed into the current weight as follows:

$$\frac{\partial E}{\partial w^{l}_{(a,b),(x,y)}} = \sum_x \sum_y \frac{\partial E}{\partial a^{l}_{x,y}} * \frac{\partial a^{l}_{x,y}}{\partial z^{l}_{x,y}} * \frac{\partial z^{l}_{x,y}}{\partial w^{l}_{(a,b),(x,y)}} = \sum_x \sum_y \delta^{l}_{z_{x,y}} * \sigma(z^{l-1}_{x-a,y-b}) \qquad (2.2.3.27)$$

where $w^{l}_{(a,b),(x,y)}$ is the weight between pixel (a,b) and pixel (x,y), $z^{l}_{x,y}$ is net output of pixel (x,y) at layer l, $a^{l}_{x,y}$ is real output of pixel (x,y), $\delta^{l}_{z_{x,y}}$ is the error rate of pixel (x,y) at layer l.

Data Augmentation

Artificially enlarge the dataset with label-preserving transformations is the simplest and the most known strategy to reduce overfitting on image data. To each randomly chose sample image, we will apply n random transformations. Each of these random transformations is a combination of several elementary forms transformation, which we will describe shortly. The benefit of using little computation is that we will not have to store the pre-processed image on the disk. Image transformation contains two major approaches: point operators (sometimes called 1-to-1 pixel transforms) and neighborhood (or area-based) operators. In point operators, each output pixel value is strictly a function of the corresponding input pixel value. Brightness and contrast adjustments are two examples of such transformation. Techniques like convolution, which we have discussed in part 2.1.2, are not 1-to-1 transform.

Brightness adjustment

We add or subtract a constant amount of light to all input pixel in order to change an image brightness, as suggested in the following function:

$$g(i,j)= f(i,j)+\beta \qquad (2.2.3.28)$$

Where f(i,j) is the pixel located in the *i-th* row and *j-th* column of the input image, g(i,j) is the pixel located in the *i-th* row and *j-th* column of the output image and β is a bias parameter which is used to control the image brightness.

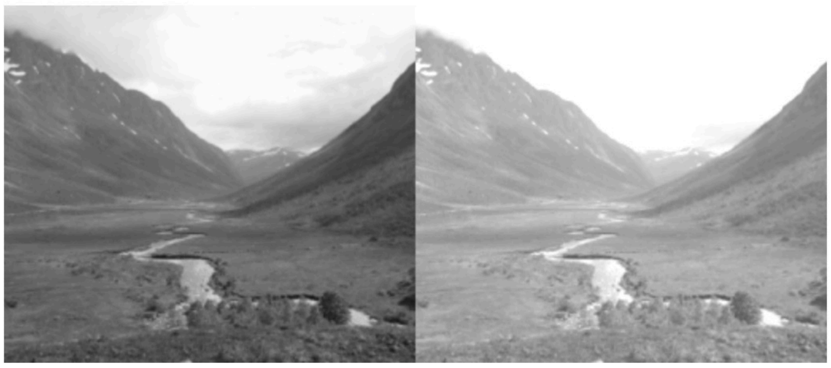


Figure 2.2.3 10 brightness adjustment. Image from pippin – point operations tutorial.

Contrast adjustment

Contrast is the different between the maximum and the minimum pixel intensity of an image. To change the contrast of an image, we change the range of the luminance value presented in the input pixels. It is mathematically suggested by the following function:

$$g(i,j)= \alpha f(i,j) \quad (2.2.3.29)$$

Where f(i,j) is the pixel located in the *i-th* row and *j-th* column of the input image, g(i,j) is the pixel located in the *i-th* row and *j-th* column of the output image and α is a weight parameter which is used to control the image contrast.

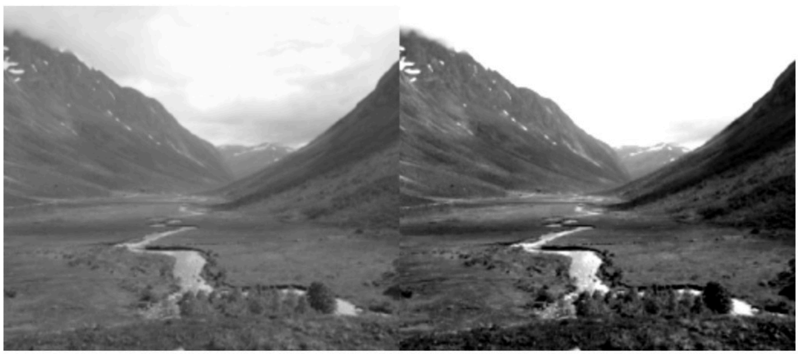


Figure 2.2.3 11 contrast adjustment. Image from pippin – point operations tutorial.

We can combine the brightness and contrast control in a single operation. It is mathematically represented as:

$$g(i,j)= \alpha f(i,j) + \beta \qquad (2.2.3.30)$$

Scaling

Scaling can be done by multiply the scale of the patch by a factor x.

Artificially enlarge the dataset using the combination of these three forms of image translation in image pre-processing has two extra benefits. Firstly, it allows the network to “look at” the seed image at different perspectives. Secondly, it allows the CNN model to work properly, because CNN often require to be fed with a considerably large dataset. Notice that data augmentation is different from image pre-processing.

Data post-processing

Combination of models is a very efficient way to reduce testing errors. At the same time, it significantly increases the number of networks parameters and thus computers processing speed. Dropout is a very efficient method for model combination. Dropout sets the output of each hidden layer to zero with the probability of 0.5. Propagation and back propagation processes will not consider the ‘dropout neurons’ into their calculations. Dropout reduces the existence of co-adaptations neurons: A neuron is forced to learn more ruggedized features to make predictions because it cannot rely on the existence of other neurons.

Model evaluation

Confusion matrix is often used to measure the performance of our classifier. It is represented a table of size jxj, where j is the number of output classes. A table entry a_{ij} is in row i^{th} and column j^{th}, indicates the number of tuples that are actually in class i but was classified by the classifier to be in class j.

For calculating convention, for every output class j, we will categorize the the tuples classified by the classifier into four groups, as suggested by Jiawei et al. [8]

True positive (TP): For each given class j, tuples that are pre-labeled as class j and correctly recognized by the classifier to be in class j are true positive. TP represents the number of true positive tuples.

True negative (TN): For each given class j, tuples that are pre-labeled as class i and correctly recognized by the classifier to be in class i are true negative. TN represents the number of true negative tuples.

False positive (FP): For each given class j, tuples that are pre-labeled as class i and mistakenly recognized by the classifier to be in class j are false positive. FP represents the number of false positive tuples.

False negative (FN): For each given class j, tuples that are pre-labeled as class i and mistakenly recognized by the classifier to be in any class other than j are false positive. FP represents the number of false positive tuples.

Table 2.2.3 1 and 2.2.3 2 shows two examples of TP, TN, FP, FN for class 0 and class 1 respectively.

	Predicted class

Actual class	0	1	2	…	j
0	**TP**	FN	FN	FN	FN
1	FP	TN	FN	FN	FN
2	FP	FN	TN	FN	FN
…	FP	FN	FN	TN	FN
j	FP	FN	FN	FN	TN

Table 2.2.3 1 Confusion matrix for class 0

			Predicted class		
Actual class	0	1	2	…	j
0	TN	FP	FN	FN	FN
1	FN	**TP**	FN	FN	FN
2	FN	FP	TN	FN	FN
…	FN	FP	FN	TN	FN
j	FN	FP	FN	FN	TN

Table 2.2.3 2 Confusion matrix for class 1

By dividing classified tuples into for groups, we can now evaluate our classifiers. The classifier accuracy or recognition rate on the given test set is calculated as follow:

$$accuracy = \frac{TP+TN}{P+N} \quad (2.2.3.31)$$

where P = TP+FP (2.2.3.32) and N = TN+FN (2.2.3.33).
Similarly, the classifier error rate or misclassification rate is:

$$\text{Error rate} = 1 - accuracy = \frac{FP+FN}{P+N} \quad (2.2.3.34)$$

GPU Concepts

The enormous amount of input data of deep learning may considerably reduce computers' processing speed. This problem can be overcome by spreading and training the networks across processing units. There are two types of processing units: central processing unit (CPU) and graphics processing unit (GPUs).
In compare with CPU, GPUs are far more powerful and efficient in parallel computing. They can be used to train far larger training sets in considerably less time. Current GPUs allow cross-GPU parallelization to be read and write into one another's memory directly. Furthermore, GPUs support networks to be trained in the cloud, which requires less power and infrastructure. Even though each GPU has limited memory, which may restrict the size of networks to be trained on one, it can

still distribute the networks size cross one another.
Recent deep learning toolkits are mostly develop based on CUDA library. This library only supports NVIDIA GPU card with compute capability >= 3.0.

CNN Software Frameworks

TensorFlow

TensorFlow (TF) is a relatively potential symbolic framework created by Google. It is developed based on CUDA library that supports parallelism through GPU mode. TF has an ability to produce a series of matrix operations (computational graph) on which automatic differentiation can be performed. With automatic differentiation, one does not have to hand-code a new variation of back propagation each time a new arrangement of the CNN is made. Its compile time is also rather fast compared to other deep learning toolkits such as Theano. Furthermore, TF is written using Python API over C/C++ and therefore supports many powerful operations. However, it at the same time has some shortages. First, the use of pure Python slows down the computational graph speed. Second, there are currently not many pre-trained models associated with it.

Caffe

Caffe is known non-symbolic deep learning framework developed by the BVLC. It is specialized in image classification problems. Caffe also uses Python API that runs on top of C/C++. It therefore also has an ability to support many powerful operations and implement a CNN fast. It allows users to finetune existing networks: One might not need to write any code to train their models! Hovever, Caffe only partially supports parallelism through its multi-GPU mode. The other problem with Caffe is that one needs to write C++ / CUDA for implementing new GPU layers.

Related research

CNN has proved to be very efficient in object recognition. A. Krizhesy et al. (2012) successfully modeled a deep CNN “to classify the 1.2 million high-resolution images in the ImageNet […] into the 1000 different classes” with “top-1 and top-5 error rates of 37.5% and 17.0%”. The networks contained five convolutional layers, three max pooling layers, three fully connected layers and a 1000-way softmax. Data argumentation, overlap pooling and dropout were used to reduce data overfitting. To improve the processing speed, the networks were split into different parts, which were trained on multiple GPUs [1]. Hokuto et al. (2014) developed a food detection and recognition deep CNN trained from 20,000 samples of food items. The networks consisted of two convolutional layers, one ReLU layer and was able to detect up to 93.8% and recognize up to 72.39% of testing food items [5]. Jonathan et al. (2014) exploited deep CNN in fine-grained object recognition. The objective was to identify different car models. The networks were built by adapting the model from A. Krizhesy et al. with little variation. A deep CNN model consists of two convolutional layers, three fully connected layers and a softmax loss were used to extract useful features from the seed image. The seed image then is used to retrieved its nearest neighbors (those that has the same pose with it). Only parts with highest energy detected from this seed image and its neighbors are chosen because they are likely to be important when describing the seed image. Learnt features obtained earlier from CNN are then pooled in the regions of each selected part. Those regions are said to describe critical parts for the class where seed image belongs. This networks were able to categorize testing data with accuracy up to 73.9%. However, the described technique is only useful for recognizing pictures with the same pose [4]. Sander et al. (2015) built a deep CNN to measure approximately 900,000 galaxy morphology by exploiting viewpoints in pooling. The model was claimed to "reproduce their consensus with near-perfect accuracy (> 99%) for most questions" in Galaxy Challenge Contest 2015. Techniques discussed in A. Krizhesy et al. were also applied to reduce data overfitting [2].

Reference

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25 (NIPS'2012), 2012
[2] Sander Dieleman, Kyle W. Willett and Joni Dambre.Rotation-invariant convolutional neural networks for galaxy morphology prediction. Mon. Not. R. Astron. Soc. 000, 1–20 (2014).
[3] Aäron van den Oord, Ira Korshunova, Jeroen Burms, Jonas Degrave, Lionel Pigou, Pieter Buteneers. Classifying plankton with deep neural networks. First prize of The National Data Science Bowl competition
[4] Jonathan Krause, Timnit Gebru, Jia Deng, Li-Jia Li, Li Fei-Fei. ICPR, 2014, supported by an ONR MURI grant and the Yahoo! FREP program. Learning Features and Parts for Fine-Grained Recognition.
[5] Hokuto Kagaya, Kiyoharu Aizawa, Makoto Ogawa. MM'14, November 3–7, 2014. Food Detection and Recognition Using Convolutional Neural Network
[6] Evgeny A. Smirnov*, Denis M. Timoshenko, Serge N. Andrianov. Comparison of Regularization Methods for ImageNet Classification with Deep Convolutional Neural Networks. 2013 2nd AASRI Conference on Computational Intelligence and Bioinformatics
[7] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. Draft book in preparation. Deep learning
[8] Jiawei Han & Micheline Kamber. Data Mining Concepts and Techniques, 3rd edition
[9] Jonathan Sachs. 1996-1999 Digital Light & Color. Digital Image Basics
[10] Rafael C. Gonzalez, Paul Wintz. Digital Image Processing, 2nd edition