CNN Primer

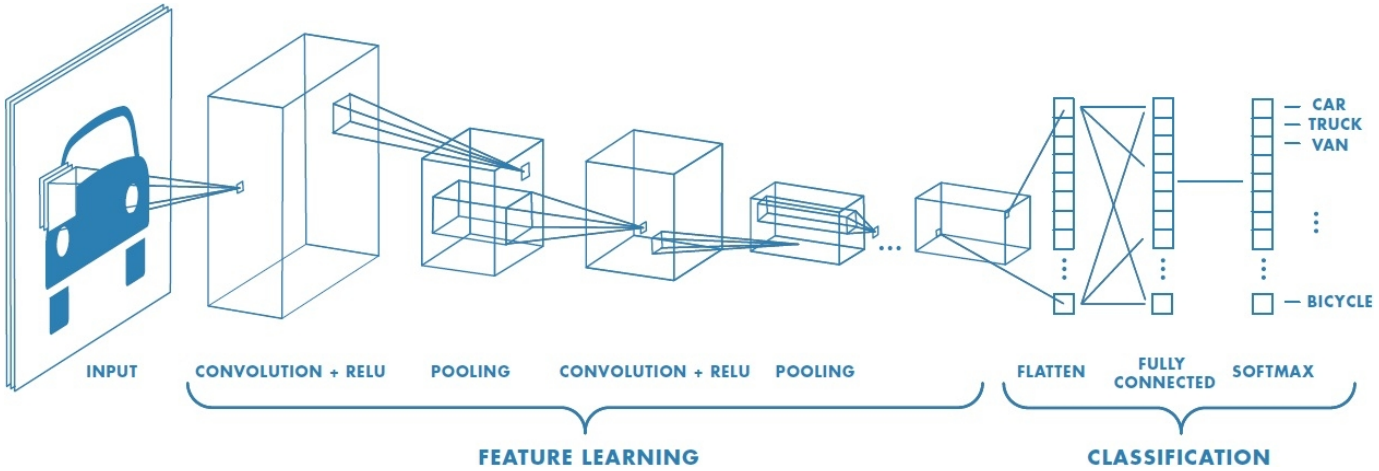
邓仰东
清华大学软件学院

Definitions

- “Parameter learning” or “training”
  - The calculation of weights of a model (either from closed-form solution, or numerical solution)
- “Inference” or “testing”
  - The process of applying an existing model with weights known onto solving a real-world problem
  - To decode a hidden state of data, e.g. to predict a label

CNN Definition

- A CNN is a DNN that is forced to extract different features at different hierarchical levels by imposing a convolution before the activation



提纲

1. CNN Components
2. Training by Backpropagation
3. Optimization
4. LeNet

Feature Detection

Feature

- How to describe an apple?

Feature

- **Method I:** Use size of picture


(640, 580)


(640, 580)


(640, 580)

- **Method II:** Use average RGB


(219, 156, 140)

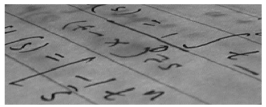

(243, 194, 113)

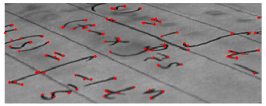

(216, 156, 155)

Feature

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}$$

Canny edge

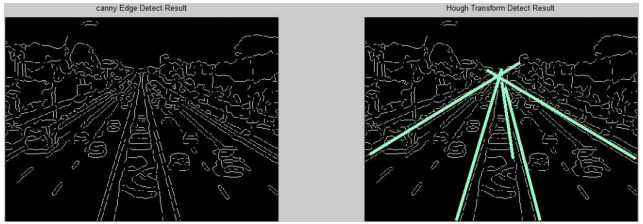




Interest point detection

Ridge detection

Scale-invariant feature transform, SIFT



Hough Transformation

Edge detection
Canny • Deriche • Differential • Sobel • Prewitt • Roberts cross

Corner detection
Harris operator • Shi and Tomasi • Level curve curvature • Hessian feature strength measures • SUSAN • FAST

Blob detection
Laplacian of Gaussian (LoG) • Difference of Gaussians (DoG) • Determinant of Hessian (DoH) • Maximally stable extremal regions • PCBR

Ridge detection

Hough transform
Hough transform • Generalized Hough transform

Structure tensor

Affine invariant feature detection
Affine shape adaptation • Harris affine • Hessian affine

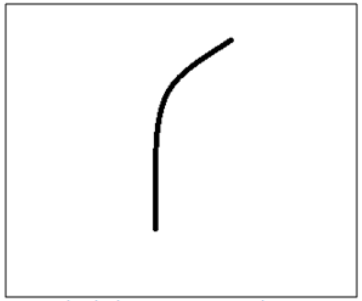
Feature description
SIFT • SURF • GLOH • HOG

Scale space
Scale-space axioms • Axiomatic theory of receptive fields • Implementation details • Pyramids

Feature Detection

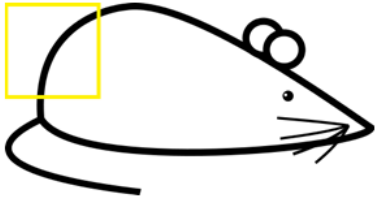
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Original image



Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

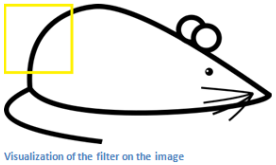
*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Feature Detection (Cont')



Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

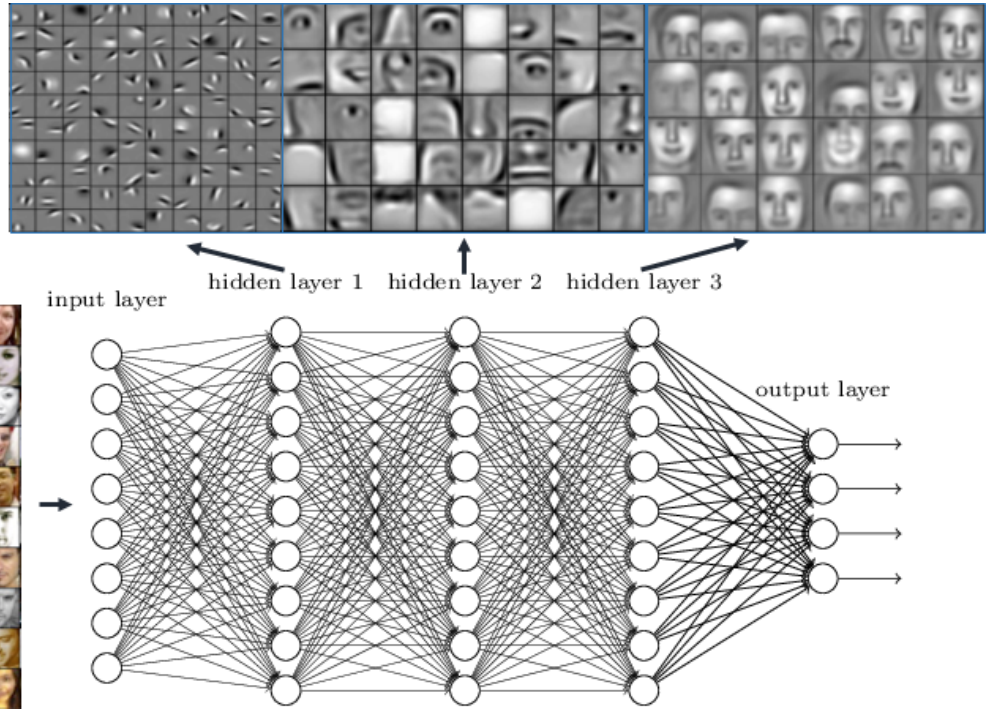
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

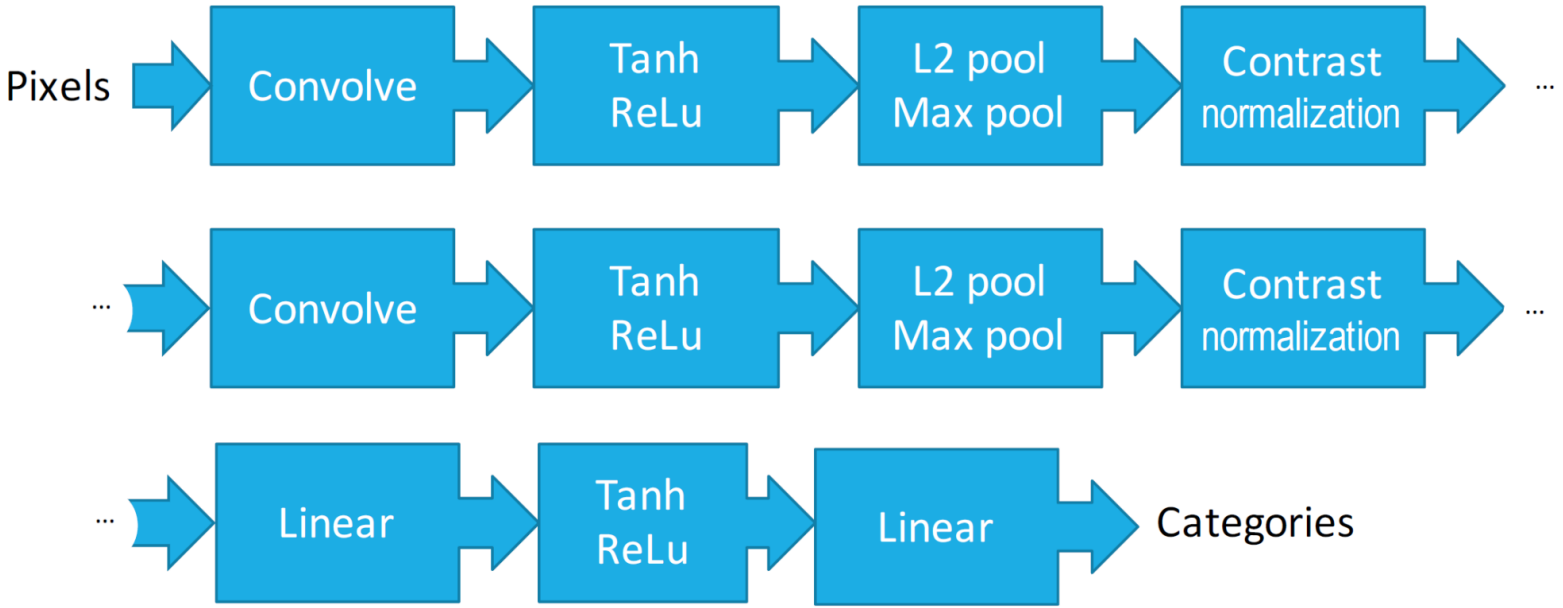
Features of CNN

Deep neural networks learn hierarchical feature representations



Convolution Neural Network Conponents

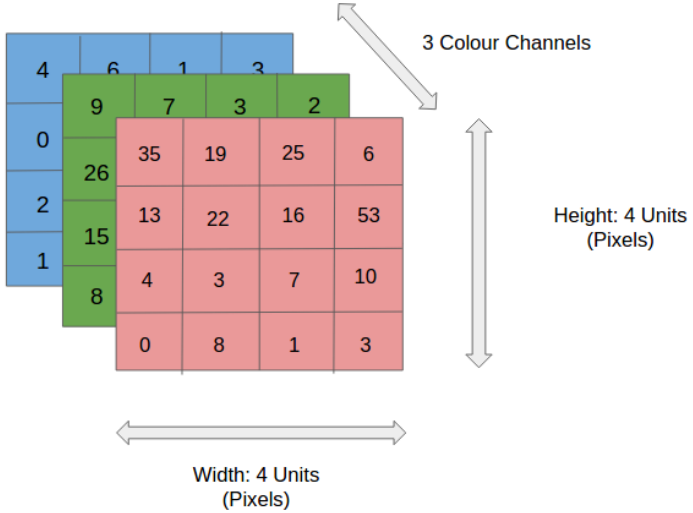
Pattern of CNN



Input

What We See

What Computers See

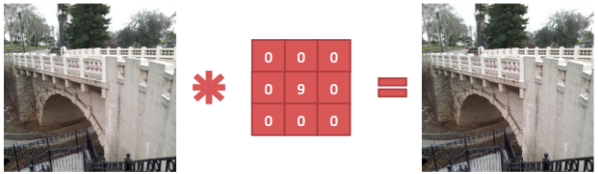


Convolution

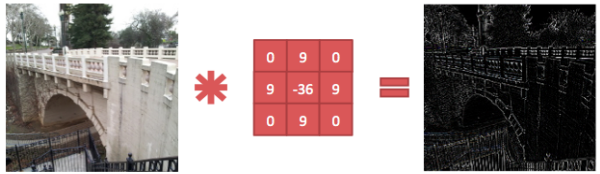
- Convolution is a mechanism to combine or “blend” two functions of time3 in a coherent manner
- One-dimensional $Eq.1: \quad y[n] = x[n] * h[n] = \sum_{k} x[k] * h[n-k] \;\; where \;\; k \in [-\infty, +\infty]$
- Two-dimensional $Eq.2: \quad (f * g)(x, y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u, v).g(x-u, y-v)$
- Though conventionally called as such, the operation performed on image inputs with CNNs is not strictly convolution, but rather a slightly modified variant called cross-correlation[10], in which one of the inputs is time-reversed:

$$Eq.3: \quad y[n] = x[n] * h[n] = \sum_{k} x[k] * h[n+k] \quad where \quad k \in [-\infty, +\infty]$$

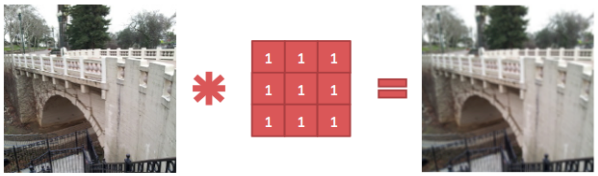
- An
- Loc



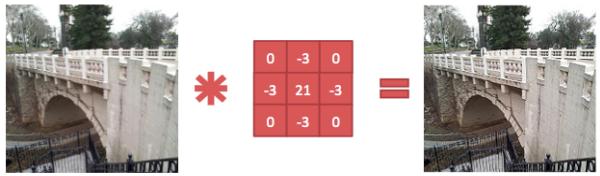
(a) Identity kernel



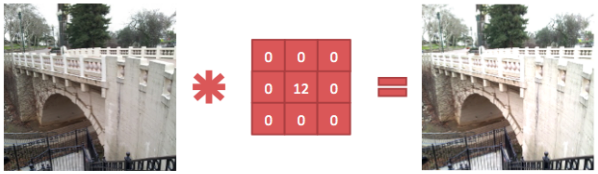
(b) Edge detection kernel



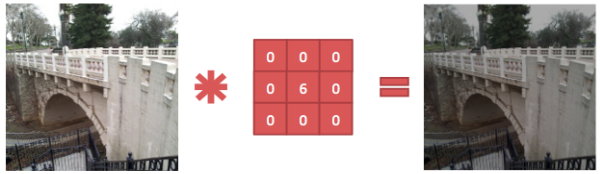
(c) Blur kernel



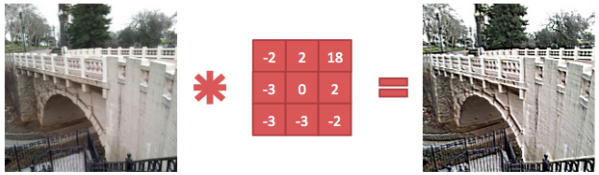
(d) Sharpen kernel



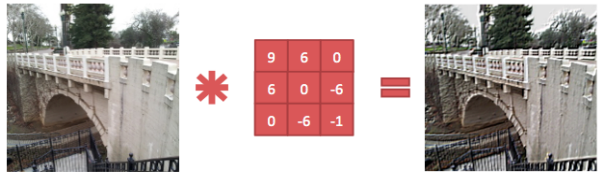
(e) Lighten kernel



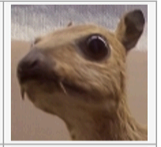
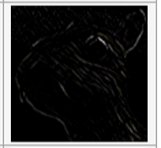
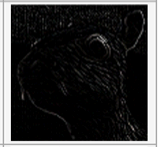
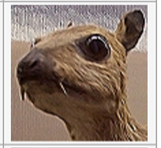
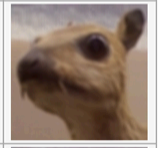
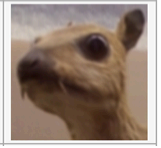
(f) Darken kernel



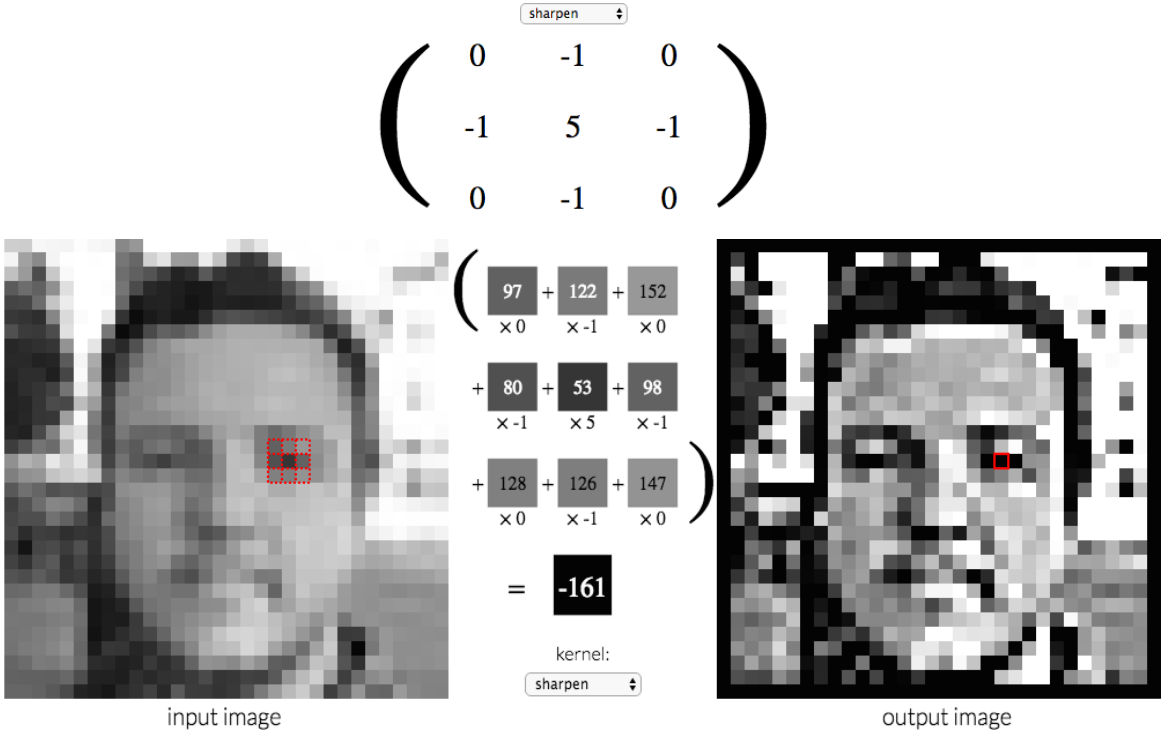
(g) Random kernel 1



(h) Random kernel 2

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Kernel Example： Sharpen

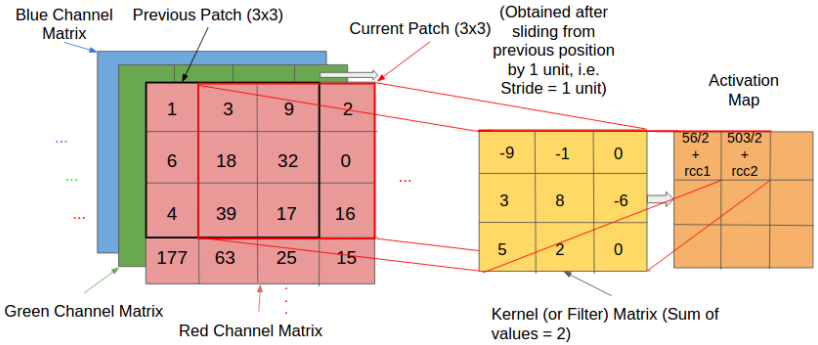


Filter

- Kernel under the context of CNN
- A kernel is a smaller-sized matrix in comparison to the input dimensions of the image, that consists of real valued entries
- The kernels are then convolved with the input volume to obtain so-called ‘activation maps’. Activation maps indicate ‘activated’ regions, i.e. regions where features specific to the kernel have been detected in the input. The real values of the kernel matrix change with each learning iteration over the training set, indicating that the network is learning to identify which regions are of significance for extracting features from the data

Kernel Operations Detailed

- Pixels are numbered from 1 in the example
- Values in the activation map are normalized to ensure the same intensity range between the input volume and the output volume. Hence, for normalization
- We divide the calculated value for the ‘red’ channel by 2 (the sum of values in the kernel matrix)
- We assume the same kernel matrix for all the three channels, but it is possible to have a separate kernel matrix for each colour channel

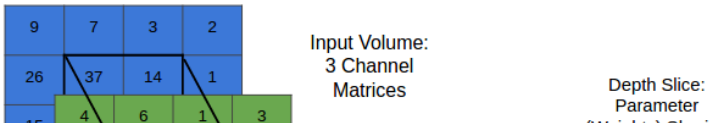


$$AM[1][2] = \textit{Red Channel Matrix Contribution(CMC)} + \textit{Green CMC} + \textit{Blue CMC)}$$
$$= [(3, 9, 2).(-9, -1, 0) + (18, 32, 0).(3, 8, -6) + (39, 17, 16).(5, 2, 0)]/2 \text{ (From-red-channel)} + \text{rcc2}$$
$$= [-27-9+0+54+256-0+195+34+0]/2 + \text{rcc2} = \mathbf{503/2 + rcc2}$$

where rcc2 refers to the contribution from the remainder channels.

Receptive Field

- It is impractical to connect all neurons with all possible regions of the input volume. It would lead to too many weights to train, and produce too high a computational complexity. Thus, instead of connecting each neuron to all possible pixels, we specify a 2 dimensional region called the ‘receptive field’ (say of size 5×5 units) extending to the entire depth of the input (5x5x3 for a 3 colour channel input), within which the encompassed pixels are fully connected to the neural network’s input layer. It’s over these small regions that the network layer cross-sections (each consisting of several neurons (called ‘depth columns’)) operate and produce the activation map
- A receptive field of a feature can be fully described by its center location and its size
- For example, for an input image of dimensions 28x28x3, if the receptive field is 5 x 5, then each neuron in the Conv. layer is connected to a region of 5x5x3 (the region always comprises the entire depth of the input, i.e. all the channel matrices) in the input volume

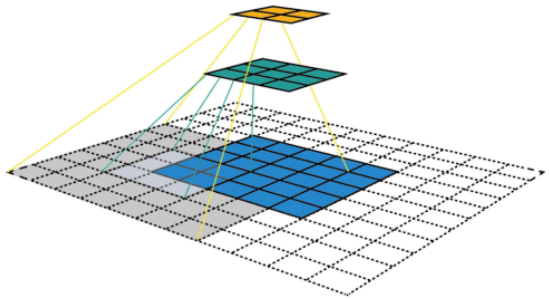


Receptive Field

- Perhaps one of the most important concepts in Convolutional Neural Networks
- The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by)
- A receptive field of a feature can be fully described by its center location and its size.

Receptive Field

- By applying a convolution C with kernel size k = 3x3, padding size p = 1x1, stride s = 2x2 on an input map 5x5, we will get an output feature map 3x3 (green map)
- Applying the same convolution on top of the 3x3 feature map, we will get a 2x2 feature map (orange map).
- The number of output features in each dimension can be calculated using the following formula

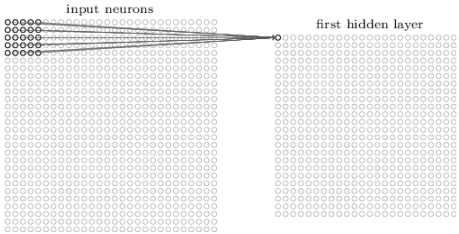


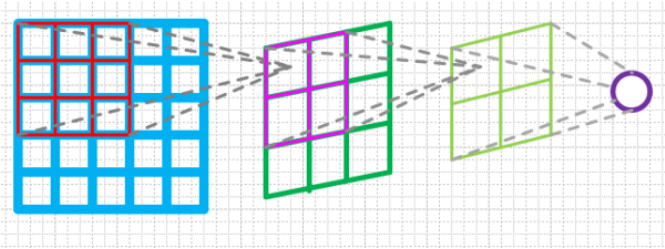
$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

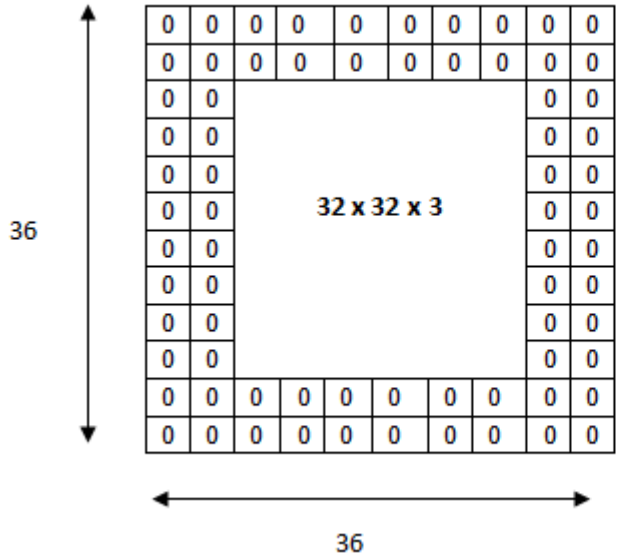
Receptive Field

- 局部感受视野(*local receptive field*)，表示一个隐藏层神经元在输入层的感受区域
  - 每一个隐藏层神经元 对应 5x5=25个权重参数w和一个基值参数b，实际上 我们规定 每一个隐藏层神经元的这些25个权值 w 和 b都共享。 也就是说隐藏层神经元共享权值
- 感受野的定义是： 卷积神经网络的每一层输出的特征图上的像素点在原图像上映射的区域大小





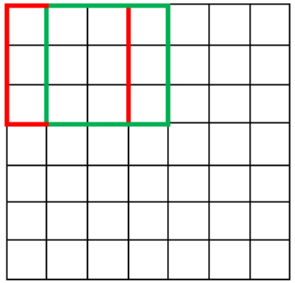
Zero-Padding



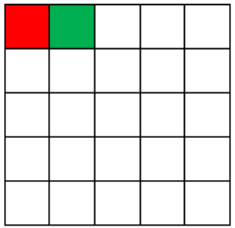
The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

Stride

7 x 7 Input Volume

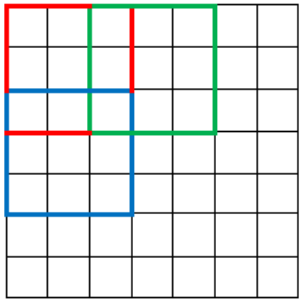


5 x 5 Output Volume

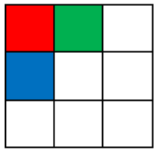


Stride = 1

7 x 7 Input Volume



3 x 3 Output Volume



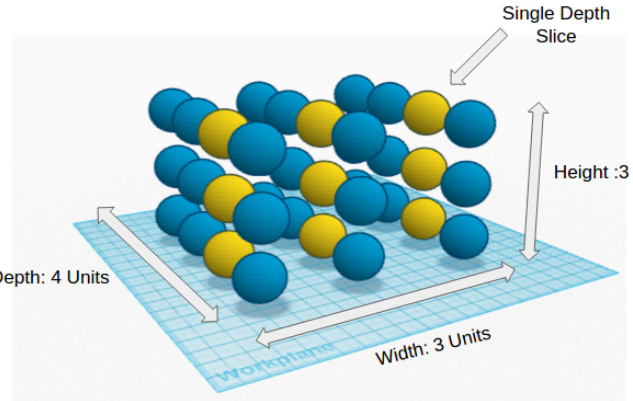
Stride = 2

Hyperparameters

- In CNNs, the properties pertaining to the structure of layers and neurons, such spatial arrangement and receptive field values, are called hyperparameters
- Hyperparameters uniquely specify layers
- The main CNN hyperparameters are receptive field (R), zero-padding (P), the input volume dimensions (Width x Height x Depth, or W x H x D) and stride length (S)

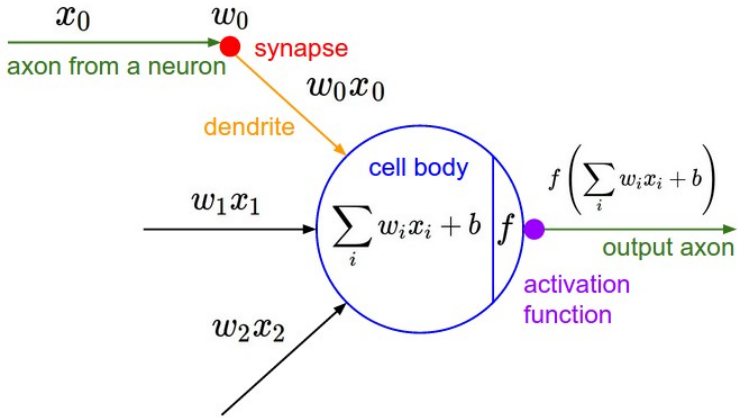
Convolutional Layer

- It performs the convolution operation over the input volume as specified in the previous section, and consists of a 3-dimensional arrangement of neurons (a stack of 2-dimensional layers of neurons, one for each channel depth)
- Shared Weights model that reduces the number of unique weights to train and consequently the matrix calculations to be performed per layer. In this model, each ‘depth slice’ or a single 2 onv architecture all share the same weights. that it doesn’t work well with images that en (such as face images), and in applicati s of the image to be detected in spatially diffe



Activation Function

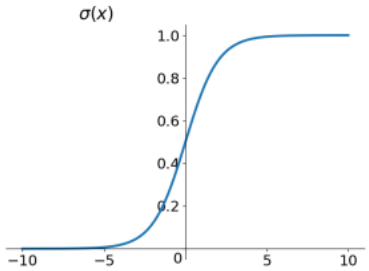
- 深度学习的基本原理是基于人工神经网络，信号从一个神经元进入，经过**非线性的** activation function，传入到下一层神经元；再经过该层神经元的activate，继续往下传递，如此循环往复，直到输出层。正是由于这些非线性函数的反复叠加，才使得神经网络有足够的capacity来抓取复杂的pattern

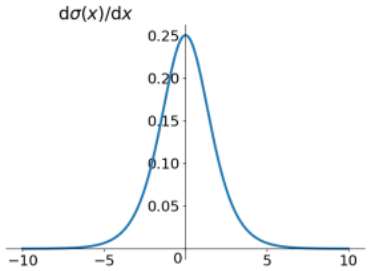


Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$
Bent identity		$f(x) = \frac{\sqrt{x^2+1} - 1}{2} + x$	$f'(x) = \frac{x}{2\sqrt{x^2+1}} + 1$
SoftExponential		$f(\alpha, x) = \begin{cases} -\frac{\log_e(1-\alpha(x+\alpha))}{\alpha} & \text{for } \alpha < 0 \\ x & \text{for } \alpha = 0 \\ \frac{e^{\alpha x}-1}{\alpha} + \alpha & \text{for } \alpha > 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \frac{1}{1-\alpha(\alpha+x)} & \text{for } \alpha < 0 \\ e^{\alpha x} & \text{for } \alpha \geq 0 \end{cases}$
Sinusoid		$f(x) = \sin(x)$	$f'(x) = \cos(x)$
Sinc		$f(x) = \begin{cases} 1 & \text{for } x = 0 \\ \frac{\sin(x)}{x} & \text{for } x \neq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x = 0 \\ \frac{\cos(x)}{x} - \frac{\sin(x)}{x^2} & \text{for } x \neq 0 \end{cases}$
Gaussian		$f(x) = e^{-x^2}$	$f'(x) = -2xe^{-x^2}$

Sigmoid函数

- Sigmoid函数是深度学习领域开始时使用频率最高的activation function
- 它是便于求导的平滑函数，其导数为 ， $\sigma(x)(1-\sigma(x))$
- 然而，Sigmoid有三大缺点：
  - 容易出现gradient vanishing
  - 函数输出并不是zero-centered
  - 幂运算相对来讲比较耗时

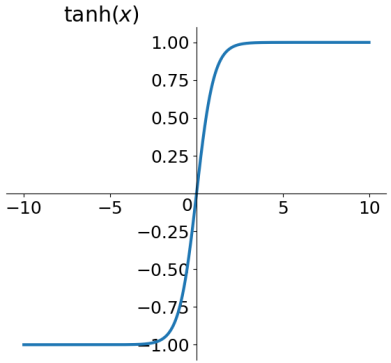


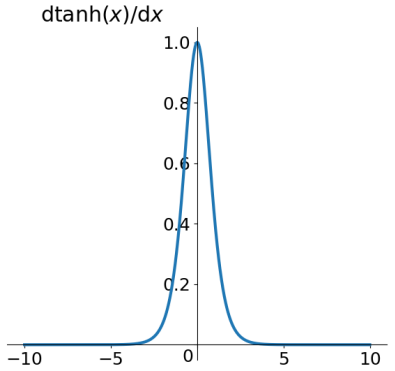


tanh函数

- tanh读作Hyperbolic Tangent，如上图所示，它解决了zero-centered的输出问题，然而，gradient vanishing的问题和幂运算的问题仍然存在

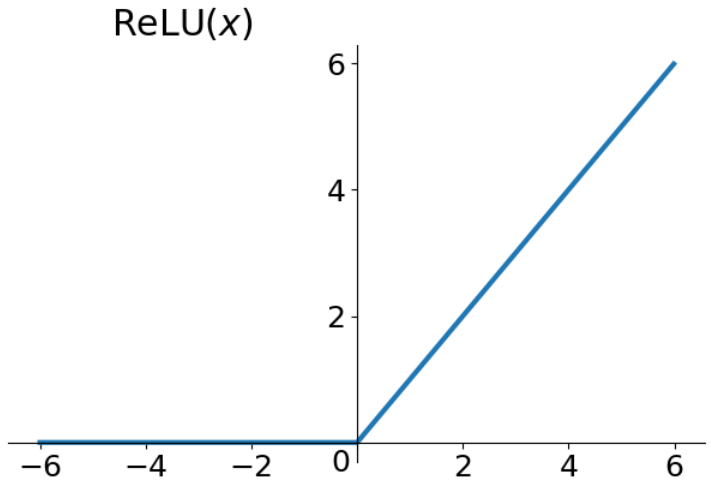
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

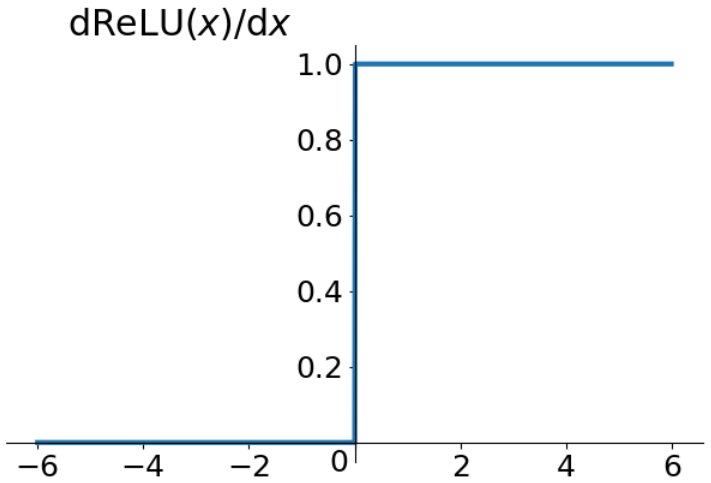




ReLU函数

- ReLU函数其实就是一个取最大值函数，注意这并不是全区间可导的，但是我们可以取sub-gradient，如上图所示。ReLU虽然简单，但却是近几年的重要成果，有以下几大优点：
- 解
- 计
- 收
- Re
- De
新
(2)
可
rat
- 尽
推





ReLu (Rectified Linear Unit) Layer

- The most commonly deployed activation function for the outputs of the CNN neurons

$$Eq.3 : max(0, x)$$

- Unfortunately, the ReLu function is not differentiable at the origin, which makes it hard to use with backpropagation training. Instead, a smoothed version called the Softplus function is used in practice:

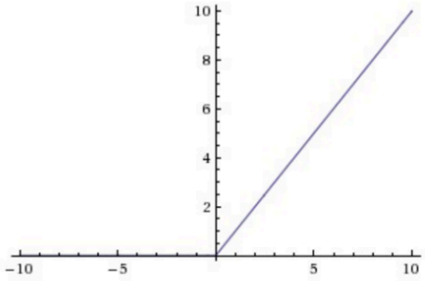
$$Eq.4 : f(x) = ln(1 + e^x)$$

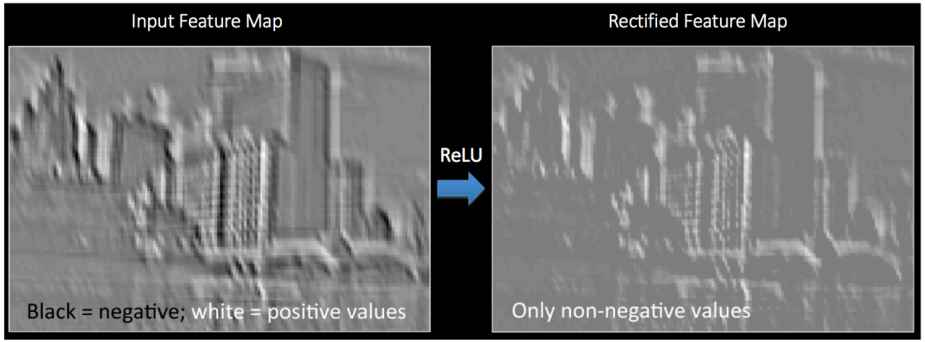
- The derivative of the softplus function is the sigmoid function, as mentioned in a prior blog post

$$Eq.5 : f'(x) = \frac{\mathrm{d}(ln(1 + e^x))}{\mathrm{d}x} = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

- ReLU 是一个元素级别的操作（应用到各个像素），并将特征图中的所有小于 0 的像素值设置为零。ReLU 的目的是在 ConvNet 中引入非线性，因为在大部分的我们希望 ConvNet 学习的实际数据是非线性的（卷积是一个线性操作——元素级别的矩阵相乘和相加，所以我们需要通过使用非线性函数 ReLU 来引入非线性

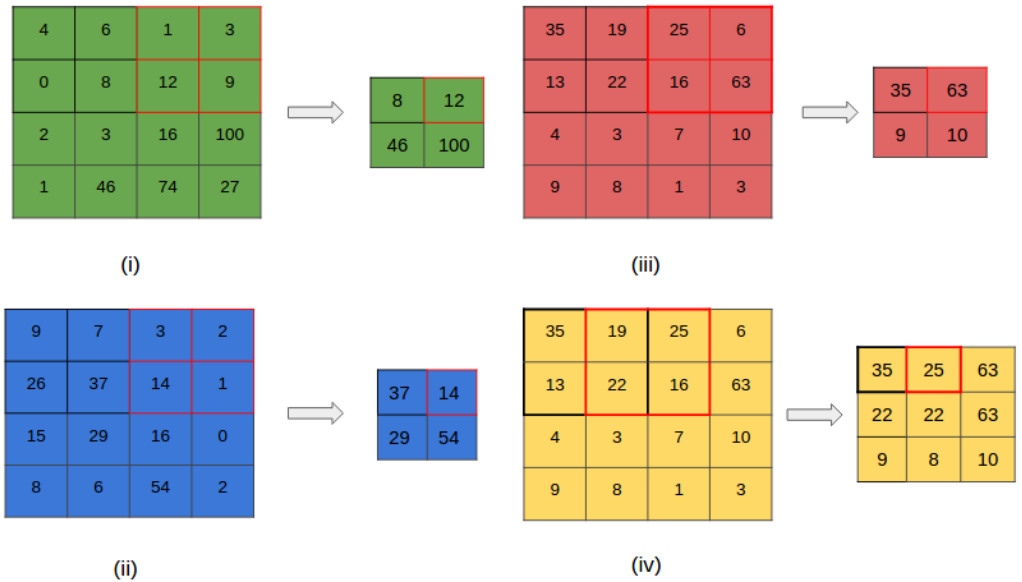
Output = Max(zero, Input)





The Pooling Layer (Sub-Sampling)

- The pooling layer is usually placed after the Convolutional layer. Its primary utility lies in reducing the spatial dimensions (Width x Height) of the Input Volume affect the depth dir
- The operation perf as the reduction of such a loss is bene
  - the decrease in si layers of the netw
  - it work against ove

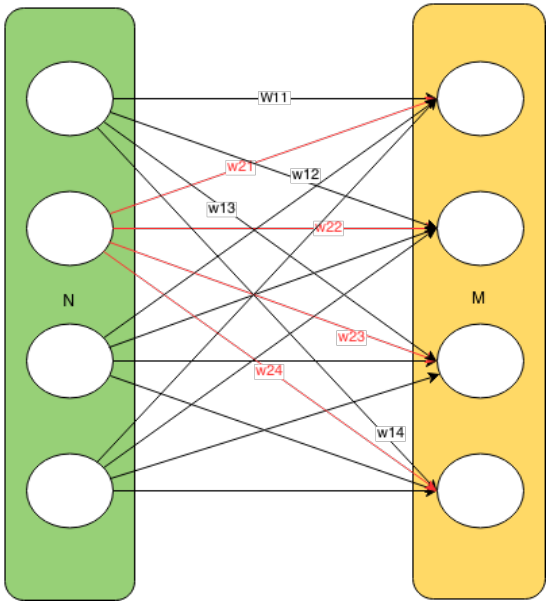


ling',
ever,

ing

Fully Connected Network

- Fully-connected layers are typically used in the last stages of the CNN to connect to the output layer and construct the desired number of outputs.



CNN Design Principles

- Given the aforementioned building blocks, the last detail before implementing a CNN is to specify its design end to end, and to decide on the layer dimensions of the Convolutional layers
- A quick and dirty empirical formula[15] for calculating the spatial dimensions of the Convolutional Layer as a function of the input volume size and the hyperparameters we discussed before can be written as follows: For each (ith) dimension of the input volume, pick:

$$W_{Out}(i) = 1 + \frac{W_{In}(i) - R + 2P}{S}$$

- Where is the (ith) input dimension, R is the receptive field value, P is the padding value, and S is the value of the stride. Note that the formula does not rely on the depth of the input.

CNN Design Principles

1. Let the dimensions of the input volume be 288x288x3, the stride value be 2 (both along horizontal and vertical directions)
2. Now, since WIn=288 and S = 2, (2.P – R) must be an even integer for the calculated value to be an integer. If we set the padding to 0 and R = 4, we get WOut=(288-4+2.0)/2+1 =284/2 + 1 = 143. As the spatial dimensions are symmetrical (same value for width and height), the output dimensions are going to be: 143 x 143 x K, where K is the depth of the layer. K can be set to any value, with increasing values for every Conv. layer added. For larger networks values of 512 are common
3. The output volume from a Conv. layer either has the same dimensions as that of the Conv. layer (143x143x2 for the example considered above), or the same as that of the input volume (288x288x3 for the example above).

CNN Design Principles

- The generic arrangement of layers can thus be summarized as follows

$$
\begin{aligned}
& Input && \rightarrow \\
& [([Conv \rightarrow ReLu] \times N) \rightarrow Pool?] \times M && \rightarrow \\
& [FullyConnected \rightarrow ReLu] \times K && \rightarrow \ FullyConnected
\end{aligned}
$$

- Where N usually takes values between 0 and 3, M >= 0 and K$\in$[0,3).

提纲

1. CNN Components
2. Training by Backpropagation
3. Optimization
4. LeNet

Training Problem

- Problems:
  - How do the filters in the first conv layer know to look for edges and curves?
  - How does the fully connected layer know what activation maps to look at?
  - How do the filters in each layer know what values to have?
- The way the computer is able to adjust its filter values (or weights) is through a training process called backpropagation.

Major Components of Training

- During the **forward pass**, you take a training image which as we remember is a 32 x 32 x 3 array of numbers and pass it through the whole network.
  - The network, with its current weights, isn't able to look for those low level features or thus isn't able to make any reasonable conclusion about what the classification might be.
- This goes to the **loss function** part of backpropagation. Remember that what we are using right now is training data. This data has both an image and a label. Let's say for example that the first training image inputted was a 3. The label for the image would be [0 0 0 1 0 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is ½ times (actual - predicted) squared.

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

Major Components of Training (Cont')

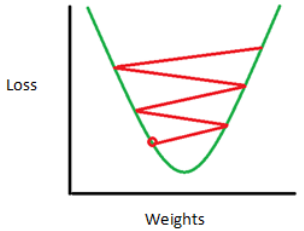
- This is the mathematical equivalent of a dL/dW where W are the weights at a particular layer. Now, what we want to do is perform a **backward pass** through the network, which is determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases.
- Once we compute this derivative, we then go to the last step which is the **weight update**. This is where we take all the weights of the filters and update them so that they change in the opposite direction of the gradient.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

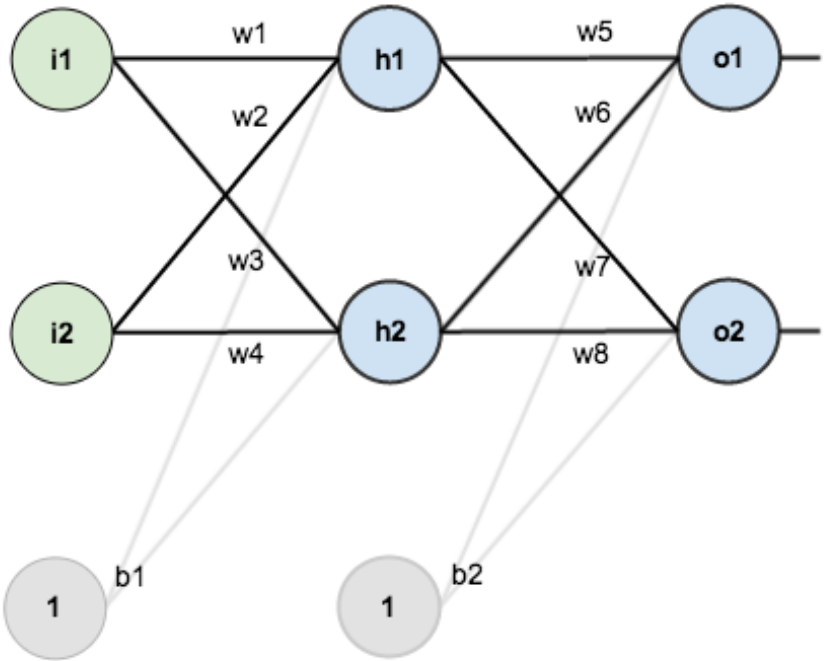
Major Components of Training (Cont')

- The learning rate is a parameter that is chosen by the programmer. A high learning rate means that bigger steps are taken in the weight updates and thus, it may take less time for the model to converge on an optimal set of weights. However, a learning rate that is too high could result in jumps that are too large and not precise enough to reach the optimal point.



Consequence of a high learning rate where the jumps are too large and we are not able to minimize the loss.

Backpropagation

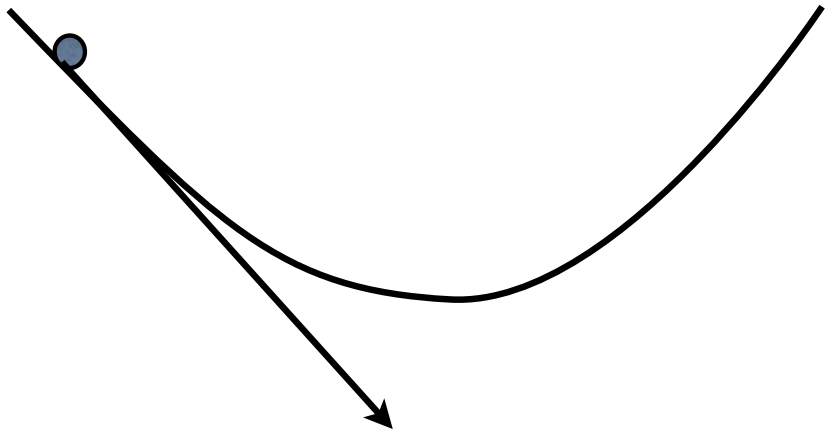


Learning Algorithm

- While not done
  - Pick a random training case (x, y)
  - Run neural network on input x
  - **Modify** connection weights to make output closer to y

How to Modify?

- Follow the gradient of the error w.r.t. the connections



Gradient points in direction of improvement

Gradient Descent Optimization Algorithm

- Increasingly popular, but often used as black-box optimizers
- by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent

Gradient Descent Optimization Algorithm

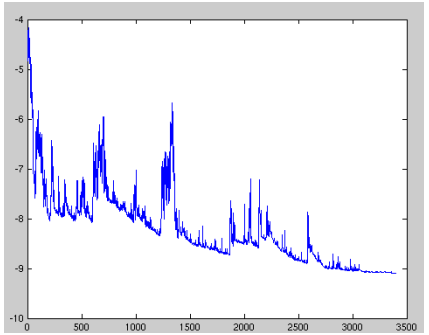
- Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in Rd$ by updating the parameters in the opposite direction of the gradient of the
- objective function $\nabla\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum.

Batch Gradient Descent

- Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:
  - $\theta = \theta - \eta \cdot \nabla\theta J(\theta)$
- As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory.
- Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update
- Batch gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

Stochastic Gradient Descent

- Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:
  - $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$
- While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.



Batch & Stochastic Gradient Descent

- Batch Gradient Descent
  - for i in range(nb_epochs)
    - params_grad = evaluate_gradient(loss_function, data, params)
    - params = params - learning_rate * params_grad
- Stochastic Gradient Descent
  - for i in range(nb_epochs): np.random.shuffle(data)
    - for example in data
    - params_grad = evaluate_gradient(loss_function , example , params)
    - params = params - learning_rate * params_grad

Mini-batch Gradient Descent

- Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:
  - $\theta = \theta - \eta \cdot \nabla\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
- This way, it a) reduces the variance of the parameter updates, which can lead to more stable conver- gence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

Mini-Batch Gradient Descent

- for i in range(nb_epochs)
  - np.random.shuffle(data)
  - for batch in get_batches(data, batch_size=50)
    - params_grad = evaluate_gradient(loss_function, batch, params)
    - params = params - learning_rate * params_grad

Challenge

- Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:
- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Learning rate schedules [18] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics [4].
- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al. [5] argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

提纲

1. CNN Components

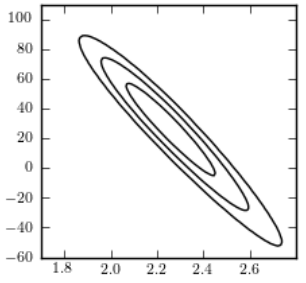
2. Training by Backpropagation

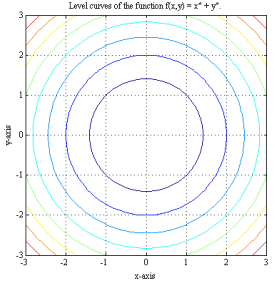
3. Optimization

4. LeNet

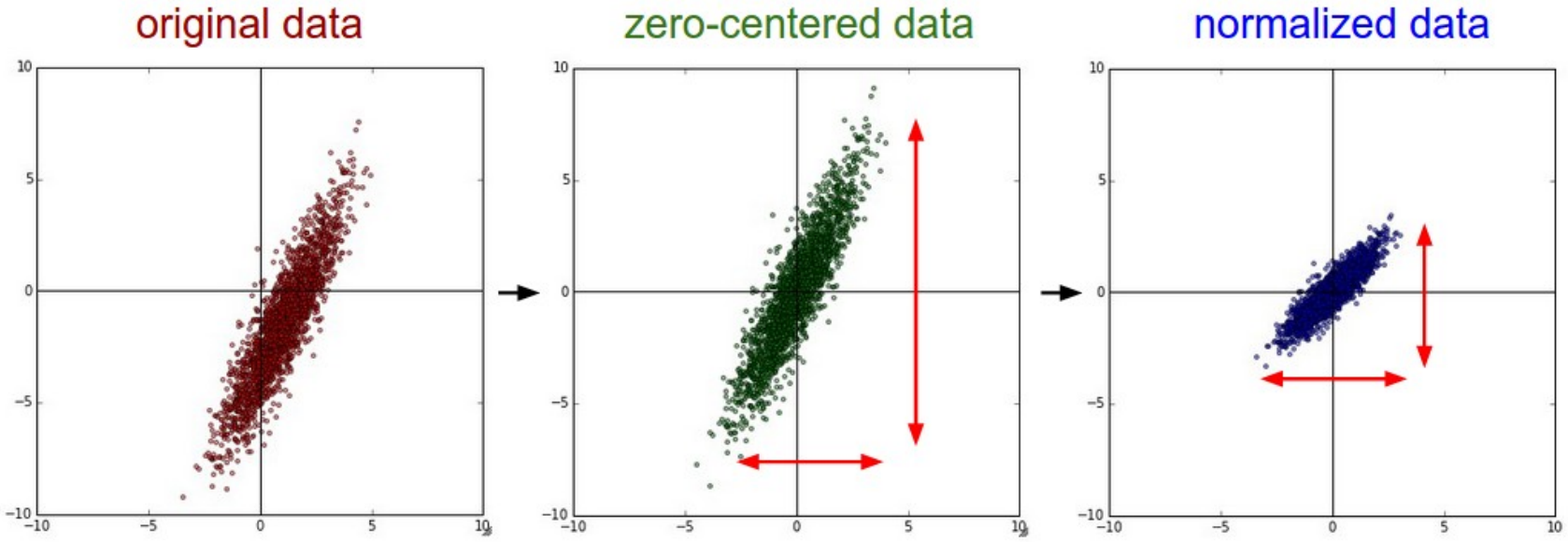
Data Preprocessing

- 零均值化（Mean subtraction）
  - 为什么要零均值化？
  - 人们对图像信息的摄取通常不是来自于像素色值的高低，而是来自于像素之间的相对色差。零均值化并没有消除像素之间的相对差异（交流信息），仅仅是去掉了直流信息的影响。
  - 数据有过大的均值也可能导致参数的梯度过大。
  - 如果有后续的处理，可能要求数据零均值，比如PCA
- 归一化（Normalization）
  - 归一化是为了让不同纬度的数据具有相同的分布规模。
  - 假如二维数据数据（x1,x2）两个维度都服从均值为零的正态分布，但是x1方差为100，x2方差为1。可以想像对（x1,x2）进行随机采样并在而为坐标系中标记后的图像，应该是一个非常狭长的椭圆形





Data Preprocessing



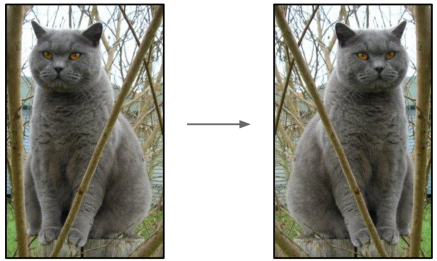

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

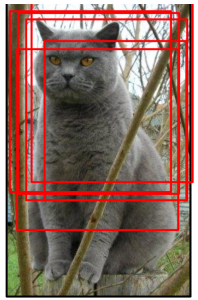
Data Augmentation

- Some popular augmentations people use are gray scales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more.
- By applying just a couple of these transformations to your training data, you can easily double or triple the number of training examples.

Data Augmentation



Flip horizontally



Random crops/scales

Color jittering

Random mix/combinations of :
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data Augmentation

- Fancy techniques: Altering the intensities of the RGB channels.
  - Compute PCA on all [R,G, B] points values in the training data;
  - Sample some color offset along the principal components at each forward pass
  - Add the offset to all pixels in a training image
- This scheme reduces the top-1 error rate by over 1%

Initialization

- Set all the initial weights to zero?
  - which you expect to be the “best guess” in expectation
- But, this turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during back-propagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

Initialization

- Set all the initial weights to zero?
  - NO!
- With proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative
- We still want the weights to be very close to 0
  - weights ~ 0.001 × N (0, 1) (Uniform distribution is also OK)

Initialization

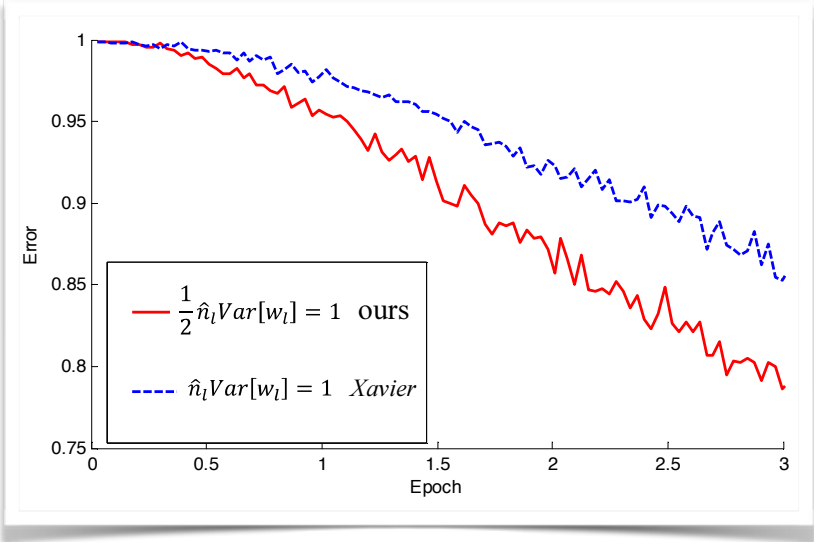
- The distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs

```
w = np.random.randn(n) / sqrt(n)
```

- This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence

Initialization

- They reached the conclusion that the variance of neurons in the network should be **2.0/n**



[K. He, et al., ICCV'15]

Hyperparameters

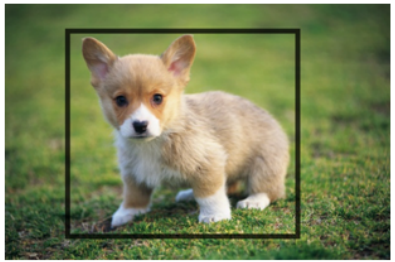
Dropout

- dropout layers have a very specific function in neural networks. In the last section, we discussed the problem of overfitting, where after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples. The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that I mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

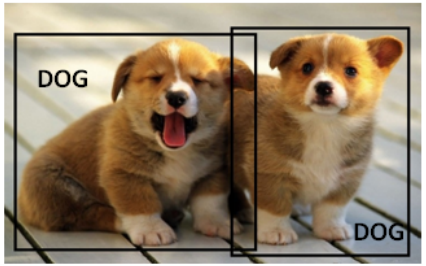
Network in Network Layers

- Classification, Localization, Detection, Segmentation

Object Classification is the task of identifying that picture is a dog



Object Localization involves the class label as well as a bounding box to show where the object is located.



Object Detection involves localization of multiple objects (doesn't have to be the same class).

Object Segmentation involves the class label as well as an outline of the object in interest.

Transfer Learning

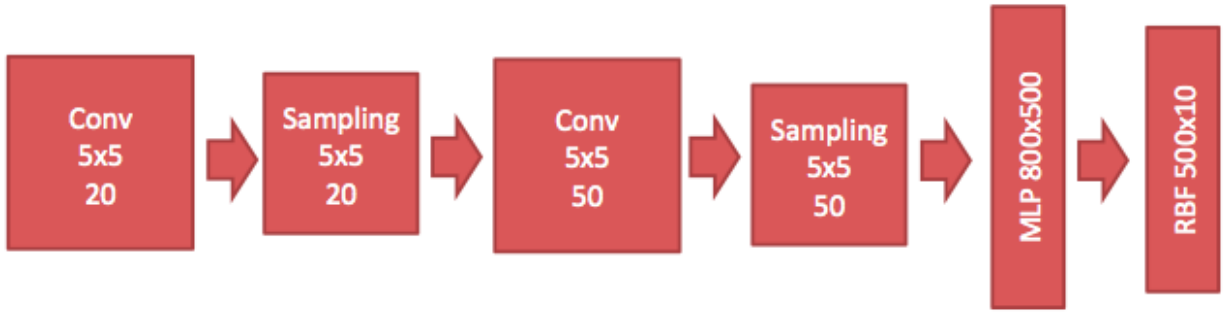
- Now, a common misconception in the DL community is that without a Google-esque amount of data, you can't possibly hope to create effective deep learning models. While data is a critical part of creating the network, the idea of transfer learning has helped to lessen the data demands. Transfer learning is the process of taking a pre-trained model (the weights and parameters of a network that has been trained on a large dataset by somebody else) and "fine-tuning" the model with your own dataset. The idea is that this pre-trained model will act as a feature extractor. You will remove the last layer of the network and replace it with your own classifier (depending on what your problem space is). You then freeze the weights of all the other layers and train the network normally (Freezing the layers means not changing the weights during gradient descent/optimization).
- Let's investigate why this works. Let's say the pre-trained model that we're talking about was trained on ImageNet (For those that aren't familiar, ImageNet is a dataset that contains 14 million images with over 1,000 classes). When we think about the lower layers of the network, we know that they will detect features like edges and curves. Now, unless you have a very unique problem space and dataset, your network is going to need to detect curves and edges as well. Rather than training the whole network through a random initialization of weights, we can use the weights of the pre-trained model (and freeze them) and focus on the more important layers (ones that are higher up) for training. If your dataset is quite different than something like ImageNet, then you'd want to train more of your layers and freeze only a couple of the low layers.

提纲

1. CNN Components
2. Training by Backpropagation
3. Optimization
4. LeNet

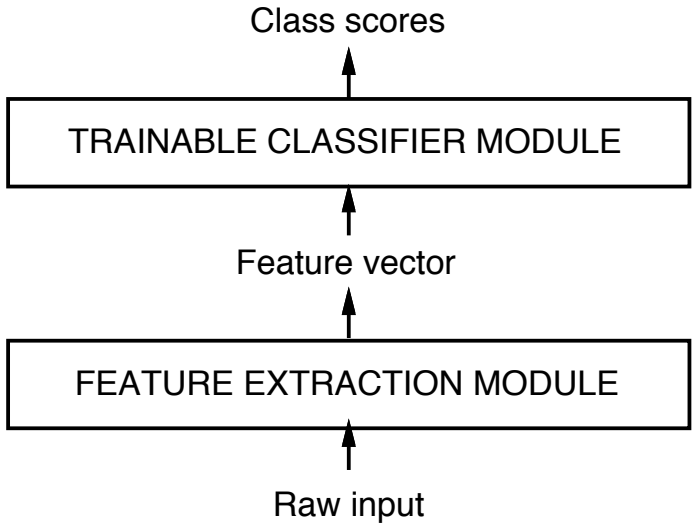
LeNet

- LeNet is known as its ability to classify digits and can handle a variety of different problems of digits including variances in position and scale, rotation and squeezing of digits, and even different stroke width of the digit. Meanwhile, with the introduction of LeNet, LeCun et al. (1998b) also introduces the MNIST database, which later becomes the standard benchmark in digit recognition field.



Design Principle

- The first module, called the feature extractor, transforms the input patterns so that they can be represented by low-dimensional vectors or short strings of symbols that: 1) can be easily matched or compared and 2) are relatively invariant with respect to transformations and distortions of the input pat- terns that do not change their nature.
  - The feature extractor contains most of the prior knowledge and is rather specific to the task.
- The classifier, on the other hand, is often general purpose and trainable



AT&T
LeNet 5
RESEARCH
answer: 0
0
1 0 3

Build A CNN with TensorFlow

- 首先是数据预处理和model的设置。然后添加第一个卷积层，滤波器数量为32，大小是5*5，Padding方法是same即不改变数据的长度和宽带。因为是第一层所以需要说明输入数据的shape ，激励选择 relu 函数
- 第一层 pooling（池化，下采样），分辨率长宽各降低一半，输出数据shape为（32，14，14）

```
model.add(Convolution2D(
    batch_input_shape=(64, 1, 28, 28),
    filters=32,
    kernel_size=5,
    strides=1,
    padding='same',       # Padding method
    data_format='channels_first',
))
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(
    pool_size=2,
    strides=2,
    padding='same',     # Padding method
    data_format='channels_first',
))
```

Build A CNN with TensorFlow

- 再添加第二卷积层和池化层

```
model.add(Convolution2D(64, 5, strides=1, padding='same', data_format='channels_first'))
model.add(Activation('relu'))
model.add(MaxPooling2D(2, 2, 'same', data_format='channels_first'))
```

- 经过以上处理之后数据shape为（64，7，7），需要将数据抹平成一维，再添加全连接层1

```
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('relu'))
```

- 添加全连接层2（即输出层）

```
model.add(Dense(10))
model.add(Activation('softmax'))
```

Build A CNN with TensorFlow

- 设置adam优化方法，loss函数, metrics方法来观察输出结果

```
model.compile(optimizer=adam,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- 开始训练模型

```
model.fit(X_train, y_train, epoch=1, batch_size=32,)
```

- 输出test的loss和accuracy结果

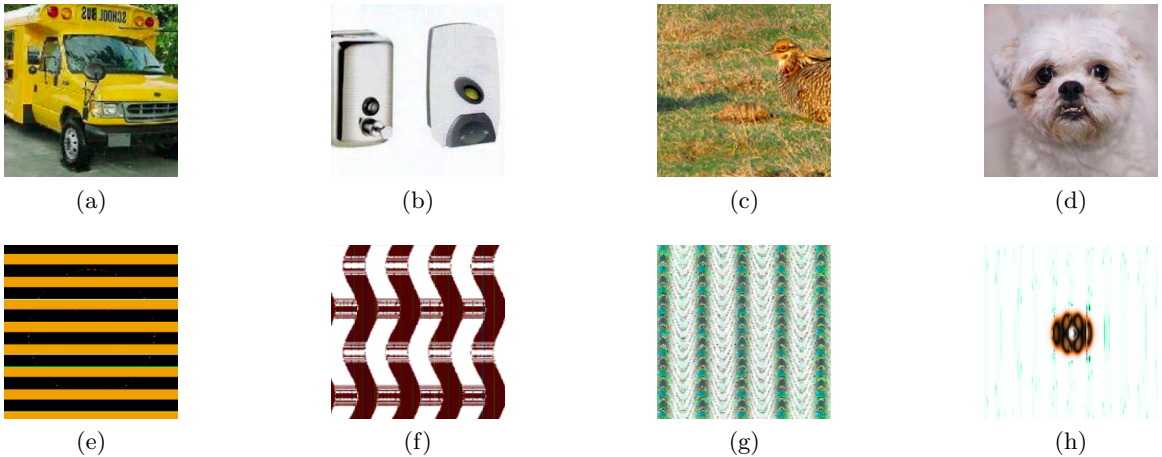
```
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.pkl.gz
15286272/15296311 [============================>.] - ETA: 0sTraining ------------
Epoch 1/1
60000/60000 [==============================] - 485s - loss: 0.2621 - acc: 0.9447

Testing ------------
10000/10000 [==============================] - 24s

test loss:  0.0894704091235

test accuracy:  0.9733
```

FUTURE



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 20: Illustrations of some mistakes of neural networks. (a)-(d) (from (Szegedy et al., 2013)) are adversarial images that are generated based on original images. The differences between these and the original ones are un-observable by naked eye, but the neural network can successfully classify original ones but fail adversarial ones. (e)-(h) (from (Nguyen et al., 2015)) are patterns that are generated. A neural network classify them into (e) school bus, (f) guitar, (g) peacock and (h) Pekinese respectively.