Artificial Neural Network and Deep Learning
Lecture 5
Deep Learning & Convolution Neural Networks (CNN)

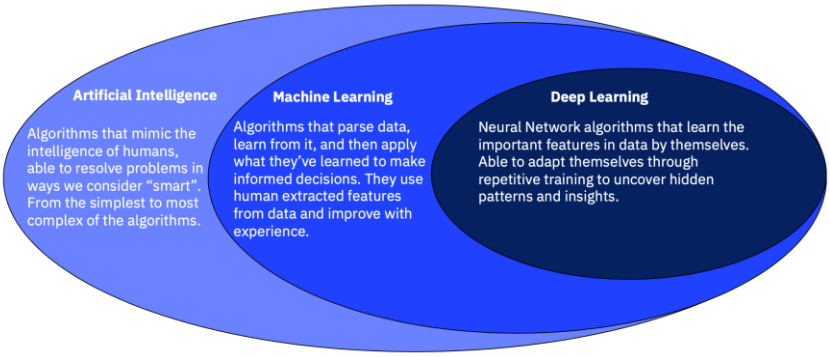
Agenda

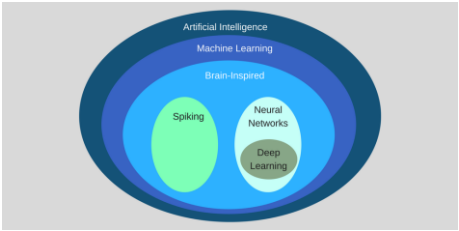
- Deep Learning
- Convolution Neural Networks (CNN)
  - **Convolution Layer**
  - **ReLU**
  - **Pooling Layers**
  - **Fully Connected layer & Classification**

- Training
- Dropout

- Neural Networks in Practice: Mini-batches
- Batch Norm layer

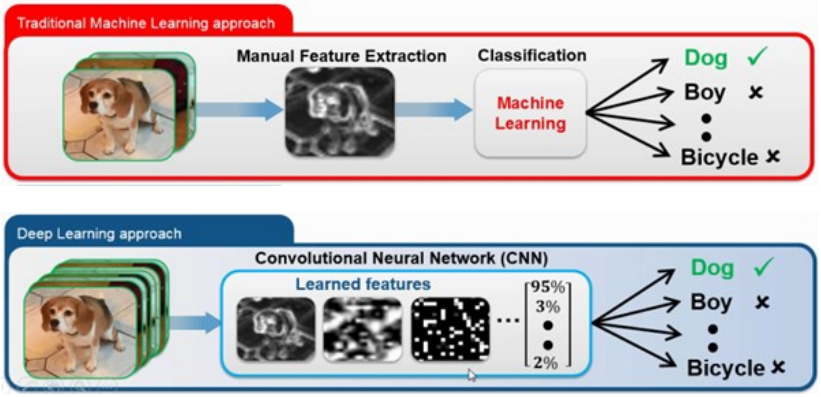
Deep Learning





Deep Learning

- Deep learning is a Neural Networks algorithms that can learn useful representations or features directly from images, text and sound.

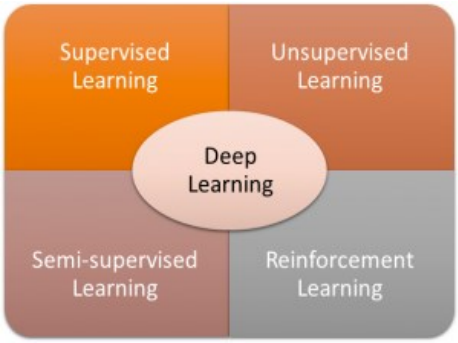


Deep Learning, cont.

Deep learning is a set of algorithms that learn to represent the data. The most popular ones.

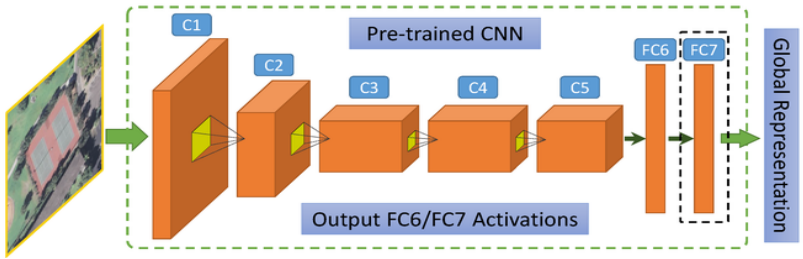
- Convolutional Neural Networks
- Deep Belief Networks
- Deep Auto-Encoders
- Recurrent Neural Networks (LSTM)

One of the promises of deep learning is that they will substitute hand-crafted feature extraction. The idea is that they will "learn" the best features needed to represent the given data.



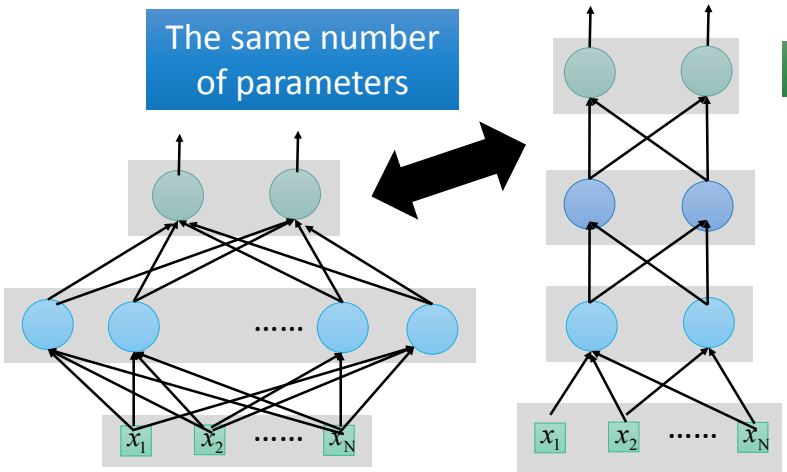
Layers and layers

- Deep learning models are formed by multiple layers.
- On the context of artificial neural networks the multi layer perceptron (MLP) with more than 2 hidden layers is already a Deep Model.
- As a rule of thumb deeper models will perform better than shallow models, the problem is that more deep you go more data, you will need to avoid over-fitting.



Fat + Short v.s. Thin + Tall

The same number of parameters



Shallow

Deep

Which one is better?

As a rule of thumb deeper models will perform better than shallow models, the problem is that more deep you go more data, you will need to avoid over-fitting.

Layer types

- Convolution layer
- Pooling Layer
- Dropout Layer
- Batch normalization layer
- Fully Connected layer
- Relu, Tanh, sigmoid layer
- Softmax, Cross Entropy, SVM, Euclidean (Loss layers)

Some guys from Deep Learning

Geoff Hinton
Google

Yann Lecun
Facebook

Yoshua Bengio
U. of Montreal

Andrew Ng
Baidu

Alex Krizhevsky
Google

Old vs New

Actually the only new thing is the usage of something that will learn how to represent the data (feature selection) automatically and based on the dataset given.

Is not about saying that SVM or decision trees are bad, actually some people use SVMs at the end of the deep neural network to do classification.

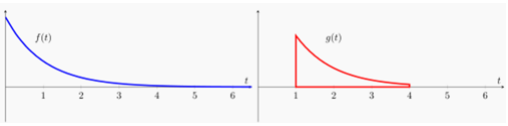
The only point is that the feature selection can be easily adapted to new data.

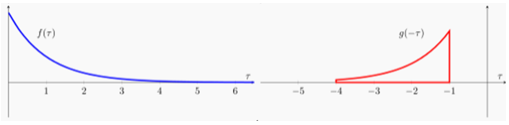
Convolution

Convolution

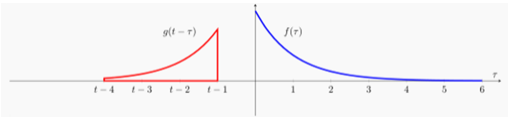
- Convolution is a mathematical operation that does the integral of the product of 2 functions(signals), with one of the signals flipped. For example bellow we convolve 2 signals f(t) and g(t).

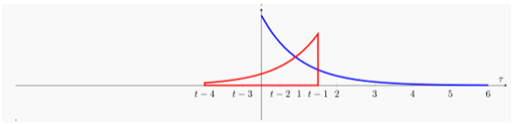
conv(a,b)==conv(b,a)

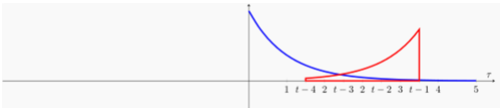




1- flip horizontally (180 degrees) the signal g.







2- slide the flipped g over f, multiplying and accumulating all it's values.

Application of convolutions

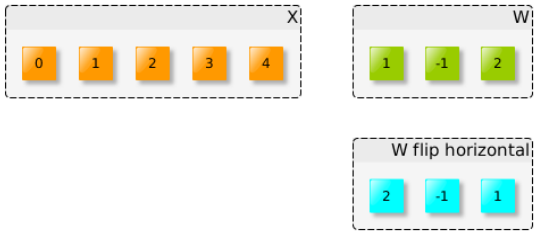
People use convolution on signal processing for the following use cases:

- Filter signals (1D audio, 2D image processing)
- Check how much a signal is correlated to another
- Find patterns in signals

Example

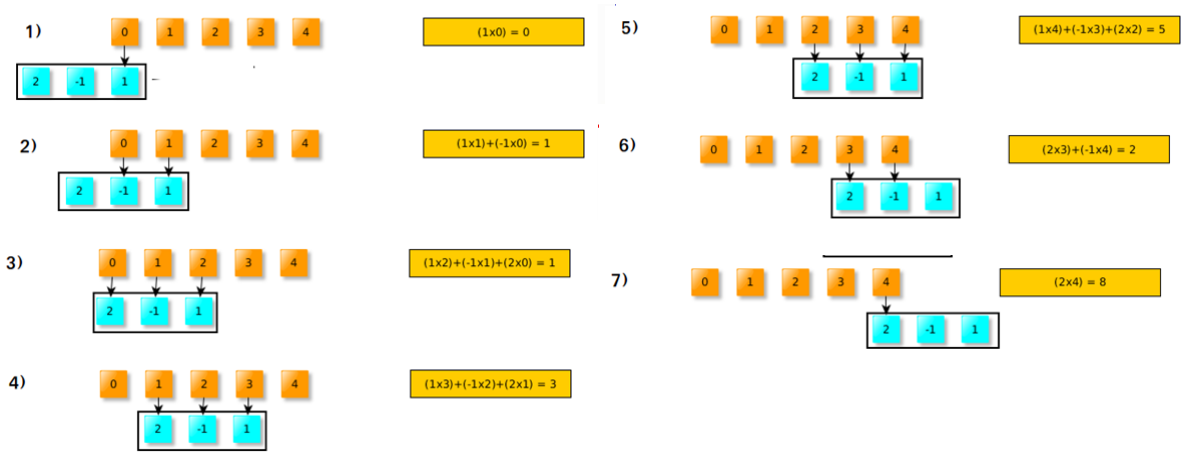
- convolve two signals x = (0,1,2,3,4) with w = (1,-1,2).

1- The first thing is to flip W horizontally (Or rotate to left 180 degrees).



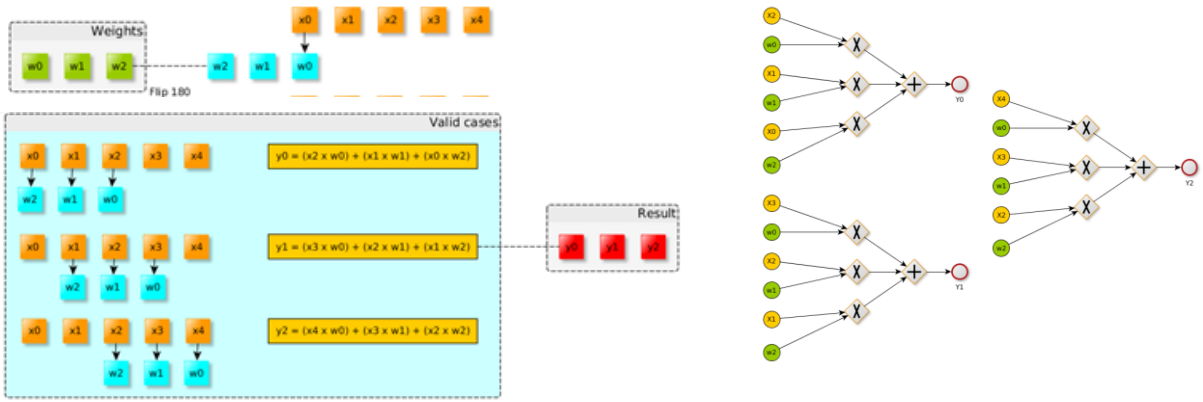
2- After that we need to slide the flipped W over the input X

2- After that we need to slide the flipped W over the input X.



- Observe that on steps 3,4,5 the flipped window is completely inside the input signal.
- The cases where the flipped window is not fully inside the input window(X), we can consider to be zero, or calculate what is possible to be calculated, e.g. on step 1 we multiply 1 by zero, and the rest is simply ignored.

Transforming convolution to computation graph



2D Convolution

- 2D convolutions are used as image filters.

2D Convolution

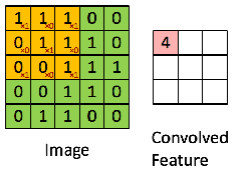
Consider 5 x 5 image

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

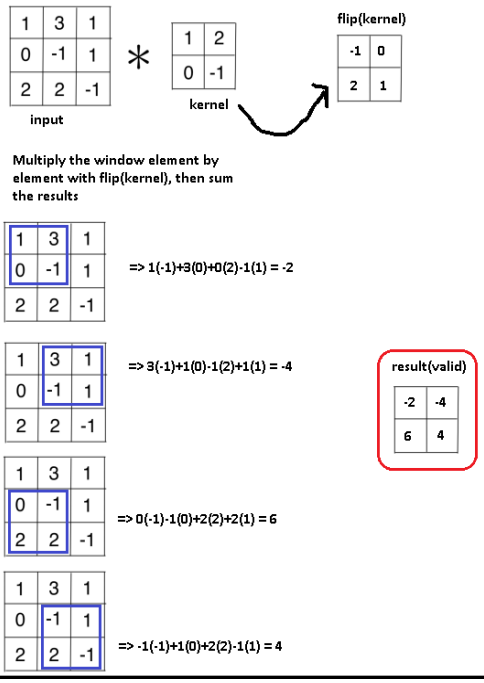
3 x 3 Filter

1	0	1
0	1	0
1	0	1

Output (Feature map)



Example



Stride

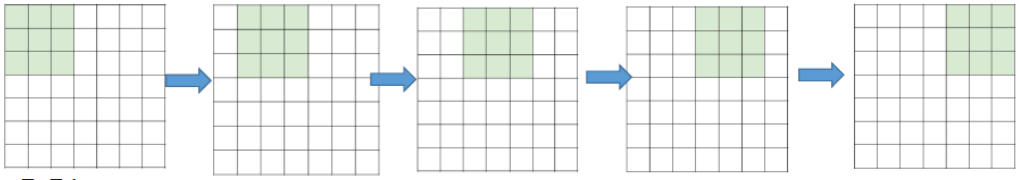
By default when we're doing convolution we move our window one pixel at a time (stride=1),

but some times in convolutional neural networks we want to move more than one pixel.

- For example, with kernels of size 2 we will use a stride of 2. Setting the stride and kernel size both to 2 will result in the output being exactly half the size of the input along both dimensions.

Convolution and stride

Replicate this column of hidden neurons across space, with some **stride.**



7x7 input

assume 3x3 connectivity, stride 1

⇒ **5x5 output**

⇒ what about stride 2?

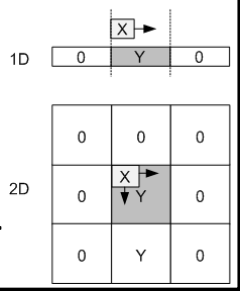
⇒ **3x3 output**

⇒ what about stride 3? Cannot.

Output size : (N-F) / stride +1

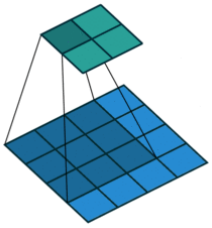
Input padding

- By default the convolution output will always have a result smaller than the input. To avoid this behaviour we need to use padding.
- In order to keep the convolution result size the same size as the input, and to avoid an effect called circular convolution, we pad the signal with zeros.
  Where you put the zeros depends on what you want to do,
  - i.e.: on the 1D case you can concatenate them on each end,
  - but on 2D it is normally placed all the way around the original signal.

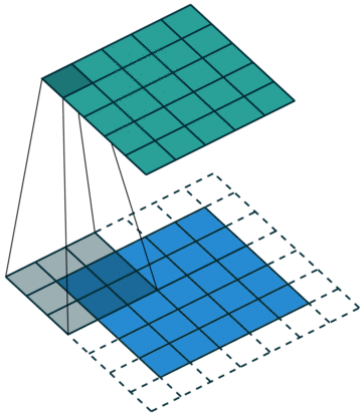


- **We have an input 5x5 convolved with a filter 3x3 (k=3).**

Convolution with no padding and stride of 1



Convolution with padding and stride of 1



Output size for 2D

- If we consider the padding and stride, input of spatial size [H, W] padded by P, with a square kernel of size F and using stride S, the output size of convolution is defined as:

$$outputSize_W = (W - F + 2P)/S + 1$$
$$outputSize_H = (H - F + 2P)/S + 1$$

- **Example**:

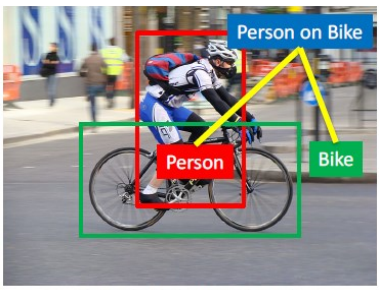
5x5 (WxH) input, with a conv layer with the following parameters Stride=1, Pad=1, F= (3x3 kernel).

The output size: ((5 - 3 + 2)/1)+ 1 = 5

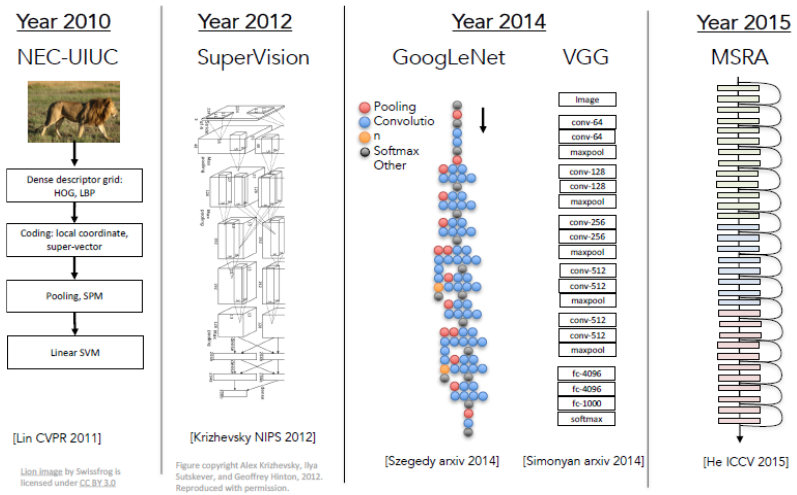
Convolution Neural Networks (CNN)

Convolution Neural Networks (CNN) have become an important tool for object recognition

- Object detection
- Action Classification
- Image Captioning (Description)

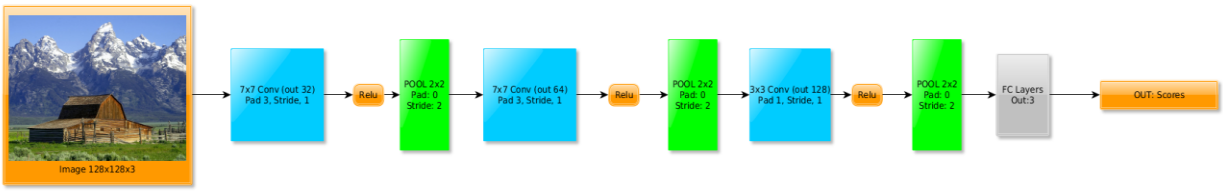


Large Scale Object Recognition Challenge



CNN

- **A CNN is composed of layers that filters(convolve) the inputs to get useful information.**
- **These have two kinds of layers: convolution layers and pooling layers.**
- **The convolution layer has a set of filters. Its output is a set of feature maps, each one obtained by convolving the image with a filter.**
- **CNN are better to work with images.**
- **Common architecture for CNN:**

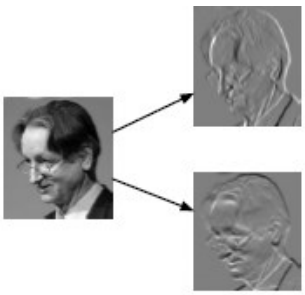


[CONV->ReLU->Pool->CONV->ReLU->Pool->FC->Softmax_loss(during train)]

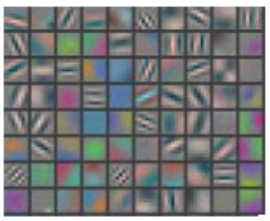
Main actor the convolution layer

- The most important operation on the convolutional neural network are the convolution layers.
- The filter will look for a particular thing on all the image, this means that it will look for a pattern in the whole image with just one filter.

Main actor the convolution layer



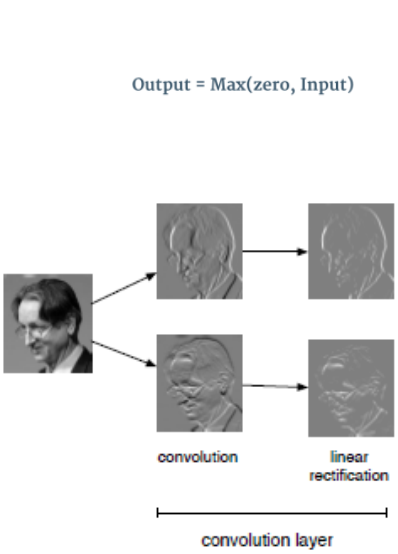
convolution

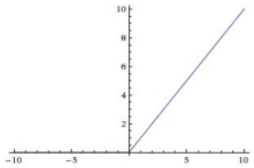


(Zeiler and Fergus, 2013, Visualizing and understanding convolutional networks)

ReLU

- It's common to apply a linear rectication nonlinearity: yi = max(zi ; 0)

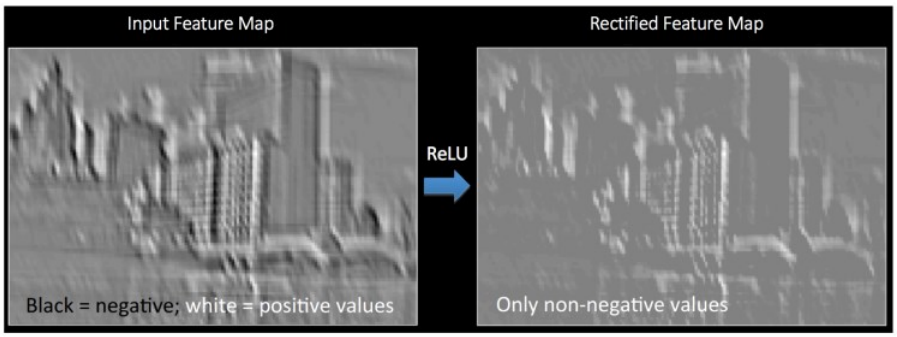




Why might we do this?

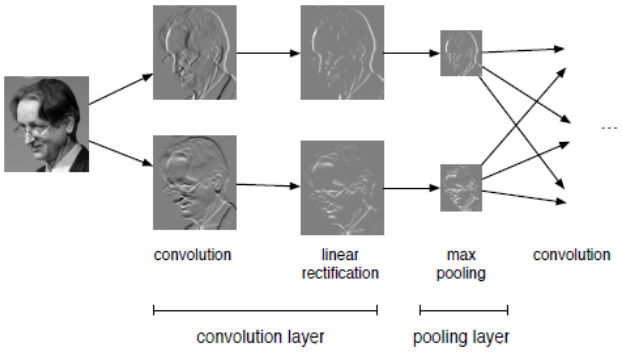
- Convolution is a linear operation.
- Therefore, we need a **non-linearity**, otherwise 2 convolution layers would be no more powerful than 1.
- ReLU has been used after every Convolution operation.

ReLU



- Other functions are also used to increase nonlinearity, for example the saturating hyperbolic tangent, sigmoid function .
- Compared to other functions the usage of **ReLU** is preferable, because it results in the neural network training several times faster.

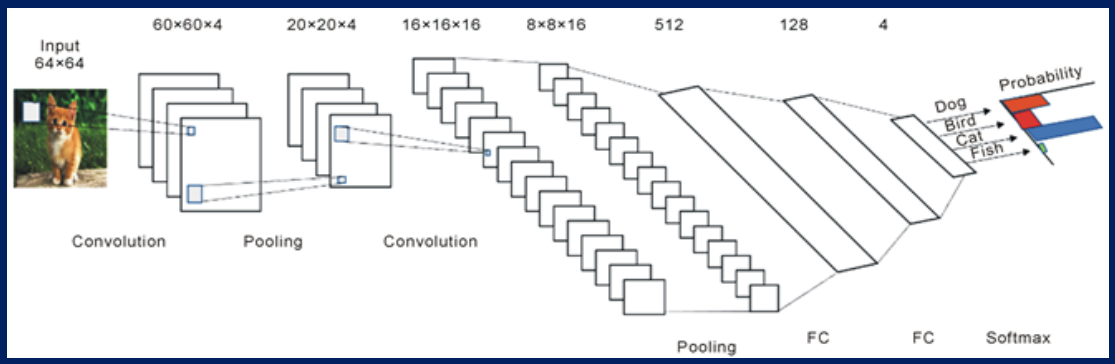
Pooling Layers



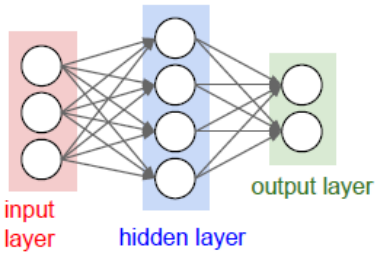
- The other type of layer is a pooling layer. These layers reduce the size of the representation and build in invariance to small transformations.

CNN

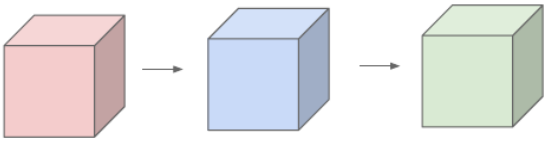
- A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolution layers, pooling layers, fully connected layers and normalization layer.



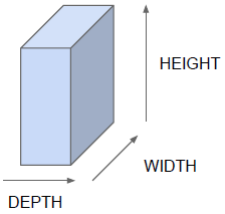
Before:



Now:



All Neural Net activations arranged in **3 dimensions:**

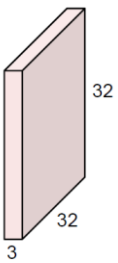


For example, a CIFAR-10 image is a 32x32x3 volume
32 width, 32 height, 3 depth (RGB channels)

Convolution Layer

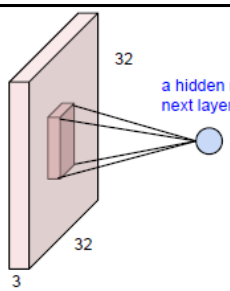
The filter depth must have the same depth as the input.

32x32x3 image



5x5x3 filter

Convolve the **filter** with the image i.e. "slide over the image spatially, computing dot products".



Connect neurons only to local receptive fields.

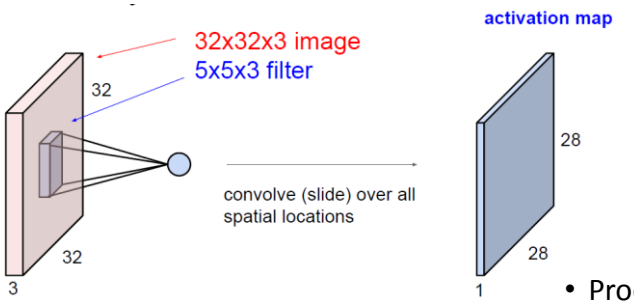
image: 32x32x3 volume

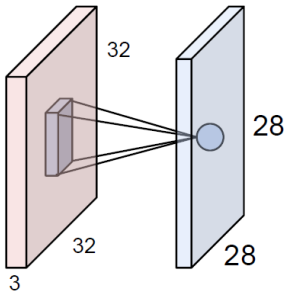
before: full connectivity: 32x32x3 weights

now: one neuron will connect to, e.g. 5x5x3 chunk and only have 5x5x3 weights.

1 number: The result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

Convolution Layer





Input volume: **32x32x3,**

filter 5x5x3, stride 1

Output size: **(N - F) / stride + 1**

= (32 - **5**) / 1 + 1

= 28

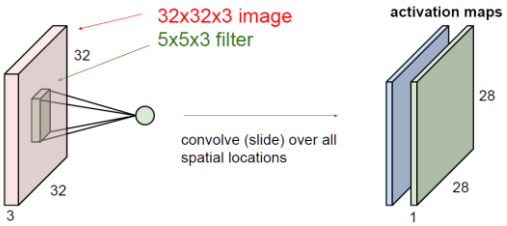
- Produces a new mapping of the image named an **activation map or feature map which** is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input.
2. All of them share parameters.

"5x5 filter" -> "5x5 receptive field for each neuron"

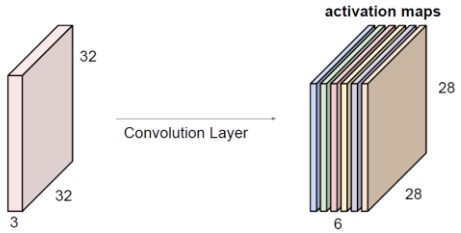
Convolution Layer

- Consider a second filter



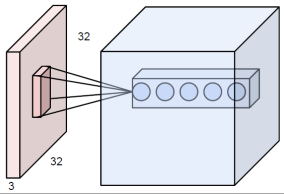
Each filter focuses on specific patterns in the image (e.g. vertical edges, horizontal edges, color, etc.) and produces a new mapping of the image named **activation map** or **feature map.**

- For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

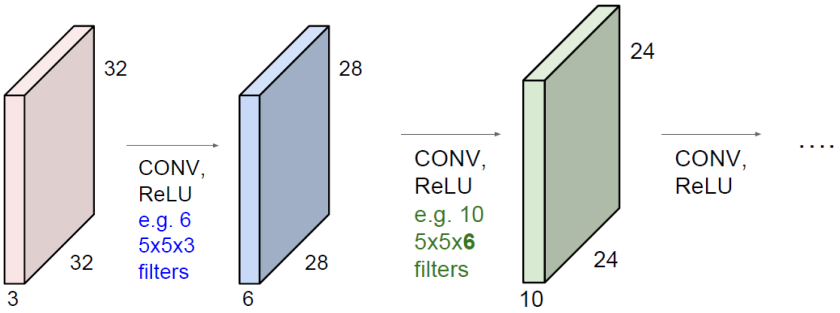


- E.g. with 5 filters, CONV layer consists of neurons arranged in a 3D grid (28x28x5).

There will be 5 different neurons <u>all looking at the same region in the input volume</u>.



Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



- Output size:

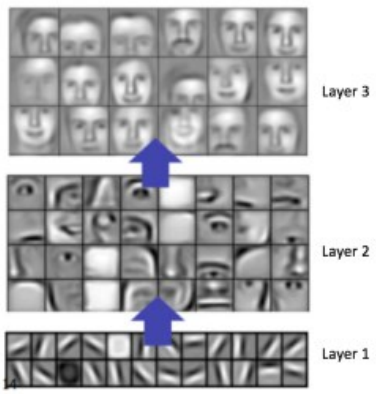
(32 - 5) / 1 + 1 = 28

- Output size:

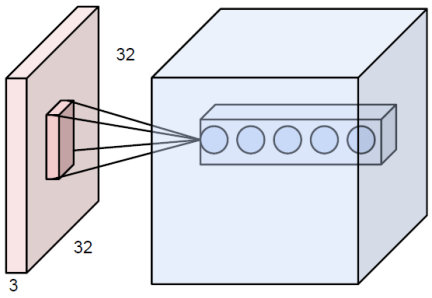
(28 - 5) / 1 + 1 = 24

Higher-level features

- In general, the more convolution steps we have, the more complicated features our network will be able to learn to recognize.
- In Image Classification, a ConvNet may learn to detect edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to detect higher-level features, such as facial shapes in higher layers.



Example 1



Input volume: **32x32x3**
Receptive fields FxF: **5x5**, **stride 1**
Number of filters: **10**

- Output size: **(N - F) / stride + 1**

Output volume size: (32 - **5**) / 1 + 1 = 28, so: **28x28x10**

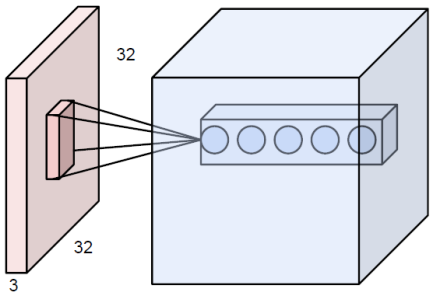
- Number of parameters in this layer?

each filter has **5*5*3** + 1 = **76** params (+1 for bias)
⇒**76*10** = **760**

Now: CNN (parameter sharing)

Before:
⇒Number of weights in such layer: (32*32*3)*10*76 = 30720 *76 ~= **3 million :**

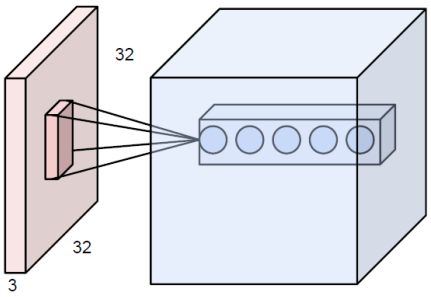
Example 1, cont.



Input volume: **32x32x3**
Receptive fields FxF: **5x5**, **stride 2**
Number of filters: **10**

Output volume: ? Cannot: (32-5)/2 + 1 = 14.5 :\

Example 1, cont.



Input volume: **32x32x3**
Receptive fields FxF: **5x5**, **stride 3**
Number of filters: **10**

- Output volume size: ? (32 - **5**) / 3 + 1 = 10, so: **10x10x10**

- Number of parameters in this layer?

each filter has **5*5*3** + 1 = **76** params (+1 for bias)
⇒**76*10** = **760**
(unchanged)

Example 2

Assume input **32x32x3 image**

If we had **30** filters with receptive fields **5x5**, applied **stride 1** and **pad 2:**

=> output volume: [32x32x**30**] (32*32***30** = 30720 neurons)

Each neuron has 5*5*3 +1 (=76) weights

=> Number of weights in the layer: (**30** * 75) + 30 = **2280 (+30 Biases, one for each neuron)**.

Output volume size

Input volume of size [W1 x H1 x D1]

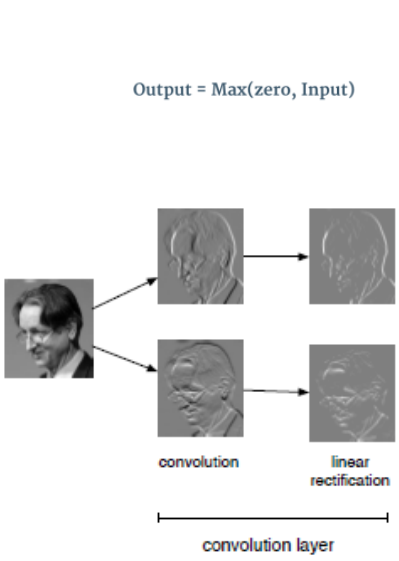
using K filters with receptive fields F x F and applying them at strides of S gives

Output volume size: [W2, H2, D2]

$$W2 = (W1-F)/S+1 \quad H2 = (H1-F)/S+1 \quad D2 = K$$

$F*F*D_1$ weights per filter, for a total of $(F*F*D_1*K)$ weights and **K** biases.

- It's common to apply a linear rectication nonlinearity: yi = max(zi ; 0)



Why might we do this?

- Convolution is a linear operation.
- Therefore, we need a **non-linearity**, otherwise 2 convolution layers would be no more powerful than 1.
- ReLU has been used after every Convolution operation.

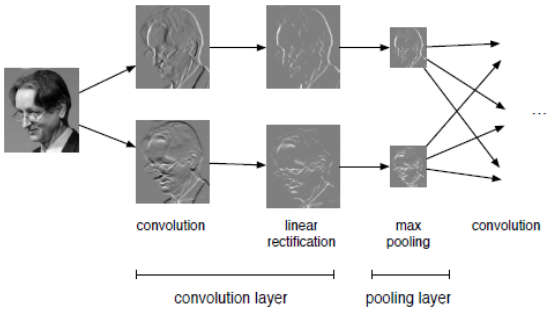
What are the advantages of ReLU over sigmoid function in deep neural networks?

ReLU is h(a)=max(0,a) where a=Wx+b.

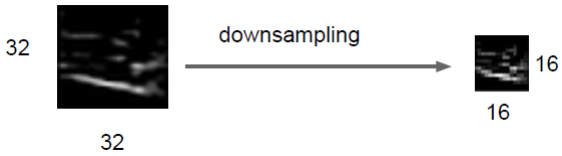
1- ReLU More computationally efficient to compute than Sigmoid functions since ReLU just needs to pick max(0,a) and not perform expensive exponential operations as in Sigmoids.
2- In practice, networks with **ReLU** tend to show **better** convergence performance **than sigmoid**.
3- Sigmoid: tend to vanish gradient (cause there is a mechanism to reduce the gradient as “u" increases, where “u" is the input of a sigmoid function).
Gradient of Sigmoid: $S'(u)=S(u)(1-S(u))$. When "a" grows to infinite large, $S'(u)=S(u)(1-S(u))=1\times(1-1)=0$.
Relu : not vanishing gradient (reduced likelihood of vanishing gradient).

- **Advantage:**
  - Sigmoid: not blowing up activation.
  - Relu : not vanishing gradient
  - Relu : More computationally efficient to compute than Sigmoid functions since Relu just needs to pick max(0, *x*) and not perform expensive exponential operations as in Sigmoids
  - Relu : In practice, networks with Relu tend to show better convergence performance than sigmoid. (Krizhevsky et al.)
- **Disadvantage:**
  - Sigmoid: tend to vanish gradient.
  - Relu : tend to blow up activation (there is no mechanism to constrain the output of the neuron, as “a" itself is the output)
  - Relu : Dying Relu problem - if too many activations get below zero then most of the units (neurons) in network with Relu will simply output zero, in other words, die and thereby prohibiting learning.(This can be handled, to some extent, by using Leaky-Relu instead.)

Pooling Layers

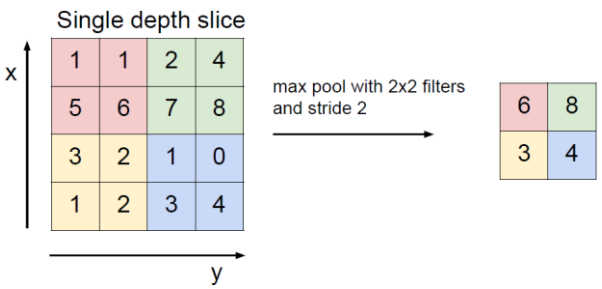


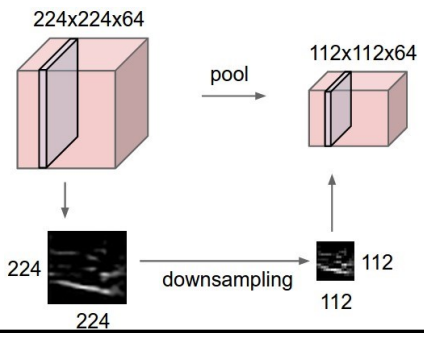
- A pooling layer **is** another building block of a CNN.
- These layers reduce the spatial dimensionality of each feature map (but not depth) (reduce the amount of parameters and computation in the network) and build in invariance to small transformations.



MAX Pooling

- Pooling retains the most important information.
- Spatial Pooling can be of different types: Max, Average, Sum, *L*2 norm, Weighted average based on the distance from the central pixel, etc.
- The most common type of pooling is the max-pooling layer, which slides a window, like a normal convolution, and get the biggest value on the window as the output.





Pooling layer

- Pooling operation is applied separately to each feature map.

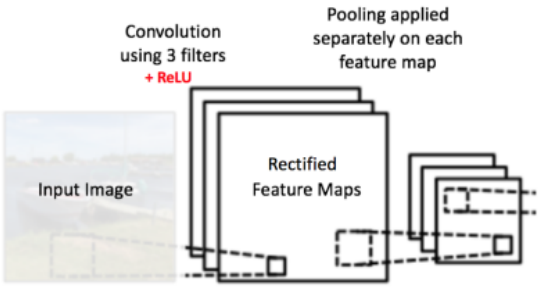
Input volume of size [W1 x H1 x D1]

Pooling unit receptive fields **F x F** and applying them at strides of **S** gives:

Output volume: [W2, H2, D1]

W2 = (W1-F)/S+1

H2 = (H1-F)/S+1

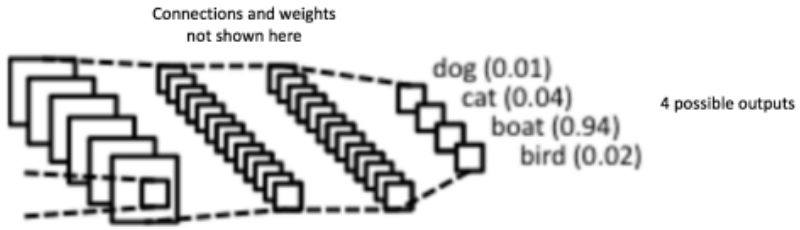


Advantage of Pooling layer

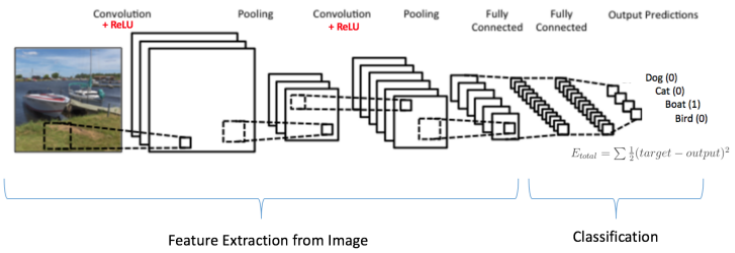
- Makes the input representations (feature dimension) smaller and more manageable.
- Reduces the number of parameters and computations in the network, therefore, controlling **Overfitting.**
- Makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).

Fully Connected layer & Classification

- The Fully Connected layer is a traditional **Multilayer Perceptron MLP** that uses a **Softmax** activation function in the output layer.
- The output from the convolutional and pooling layers represent **high-level features** of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into **various classes** based on the training dataset.
- Apart from classification, adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.



Training



- **Step1:** We initialize all filters and parameters, weights with random values.
- **Step2: Compute** convolution, ReLU and pooling operations along with forward propagation in the fully connected layer and finds the output probabilities for each class.
- **Step3:** Calculate the total error at the output layer (summation over all 4 classes) **Total Error = ∑ ½ (target probability – output probability)²**
- **Step4:** Use Backpropagation to calculate gradients, update the weights.
- **Step5:** Repeat step2 to step4 with all images in the training set.

Training

- Note:

Parameters like

number of filters,

filter sizes,

architecture of the network etc.

have all been **fixed** before Step 1 and do not change during training process.

only the

values of the filter matrix and connection weights

get updated.

Test

- When a new (unseen) image is input into the ConvNet, the network would go through the **forward propagation step and output a probability for each class** (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples).
- If our training set is **large enough**, the network will (hopefully) **generalize well** to new images and **classify them into correct categories**.

Typical ConvNets look like:

[CONV-RELU-POOL]xN,[FC-RELU]xM,FC,SOFTMAX

or

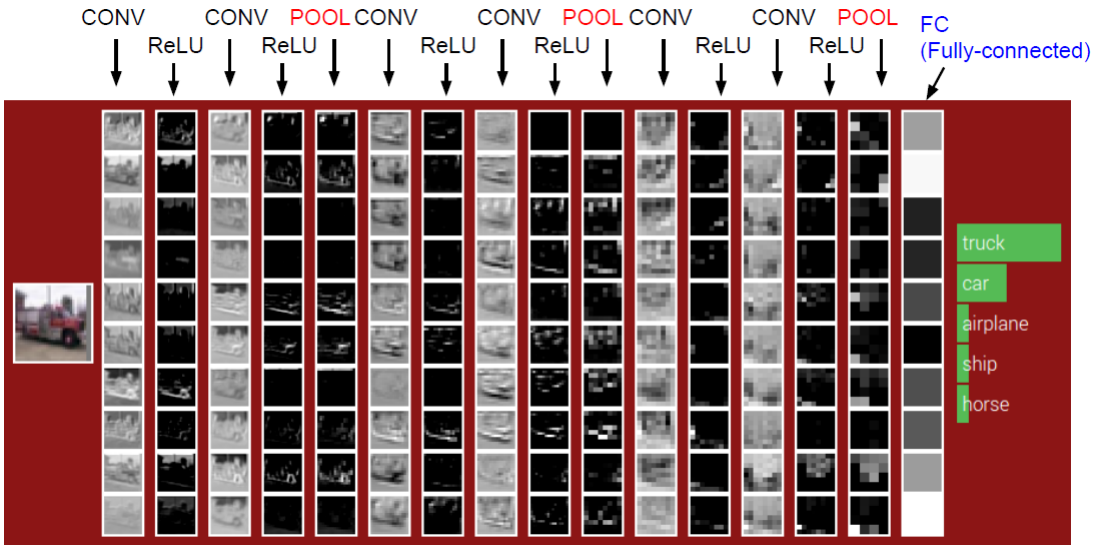
[CONV-RELU-CONV-RELU-POOL]xN,[FC-RELU]xM,FC,SOFTMAX

N >= 0, M >=0

Note:

(last FC layer should not have RELU - these are the class scores)

CIFAR-10 example



CIFAR-10 example

input: [32x32x3]

CONV with 10 3x3 filters, stride 1, pad 1:

gives: [32x32x10]

new parameters: (3*3*3)*10 + 10 = 280

RELU

CONV with 10 3x3 filters, stride 1, pad 1:

gives: [32x32x10]

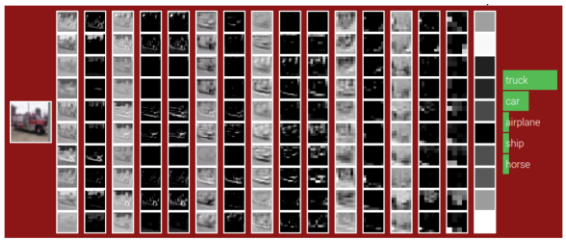
new parameters: (3*3*10)*10 + 10 = 910

RELU

POOL with 2x2 filters, stride 2:

gives: [16x16x10]

parameters: 0



CONV with 10 3x3 filters, stride 1:

gives: [16x16x10]

new parameters: (3*3*10)*10 + 10 = 910

RELU

CONV with 10 3x3 filters, stride 1:

gives: [16x16x10]

new parameters: (3*3*10)*10 + 10 = 910

RELU

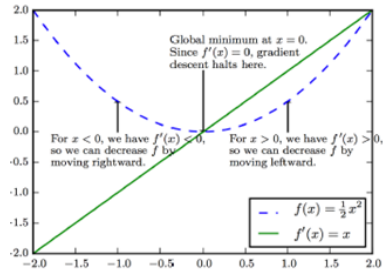
POOL with 2x2 filters, stride 2:

gives: [8x8x10]

parameters: 0

Cost function

- Error function (cost function) is minimized by moving from current solution in direction of the negative of gradient.
- Cost function often decompose as a sum per sample loss function.
- As training set size grows to billions, time taken for single gradient step becomes prohibitively long.

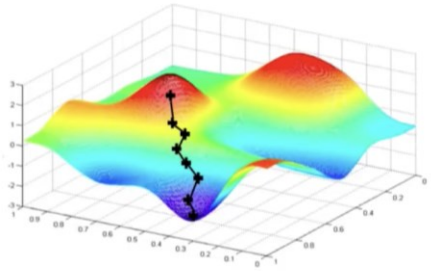


Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Can be very **computationally intensive** to compute!

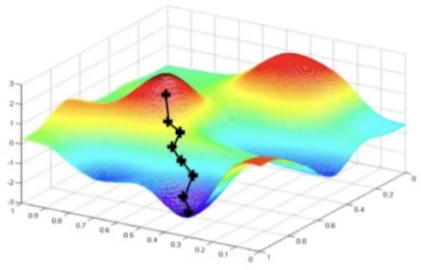


Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

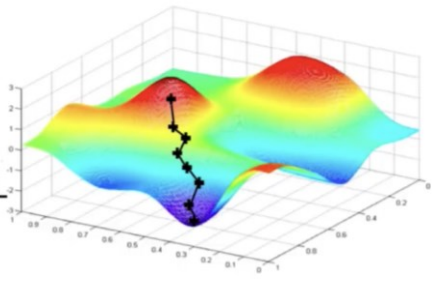
Easy to compute but **very noisy** (stochastic)!



Stochastic Gradient Descent

Algorithm

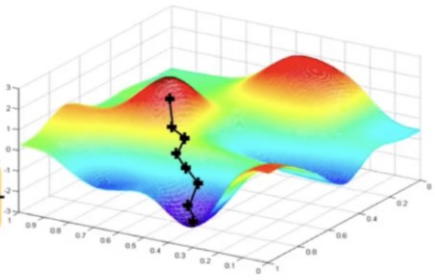
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Mini-batches while training

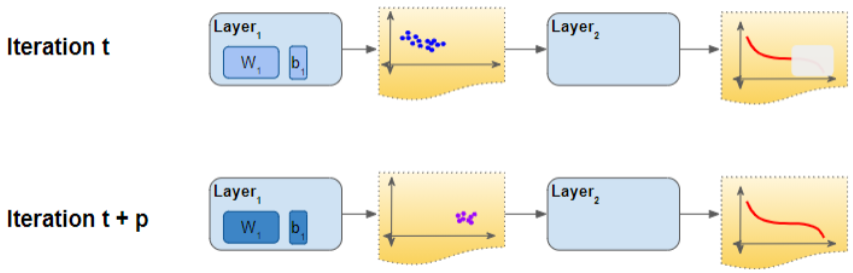
- Mini-batch: Only use a small portion of the training set to compute the gradient.
- Common mini-batch sizes are ~ 100 examples.
- More accurate estimation of gradient.
  - Smoother convergence.
  - Allows for large learning rates.
- Mini-batches lead to fast training
  - Can parallelize computation + achieve significant increases on GPU's.

Batch Norm layer - Motivation

The range of values of raw training data often varies widely.

In general, Gradient descent converges much faster with feature scaling than without it.

Internal covariate shift



- That is also the input for layer 'k'. In other words, that layer receives input data that has a different distribution than before.
- It is now forced to learn to fit to this new input.
- As we can see, each layer ends up trying to learn from a constantly shifting input, thus taking longer to converge and slowing down the training.

Common normalizations

Two methods are usually used for rescaling or normalizing data:

1. Scaling data all numeric variables to the range [0,1]. One possible formula is given below:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

2. To have zero mean and unit variance:

$$x_{new} = \frac{x - \mu}{\sigma}$$

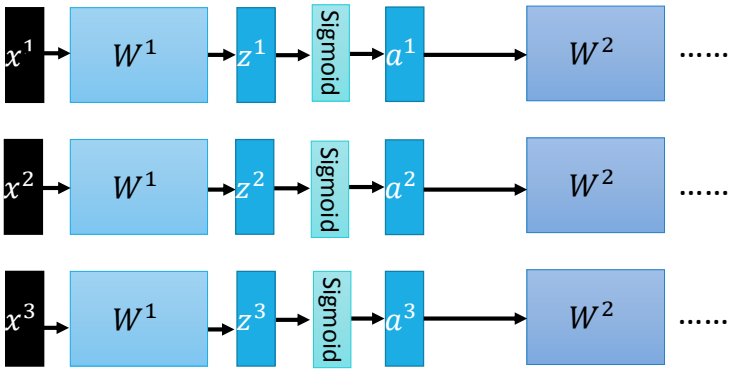
- In the NN community this is call *Whitening*

Proposed Solution: Batch Normalization (BN)

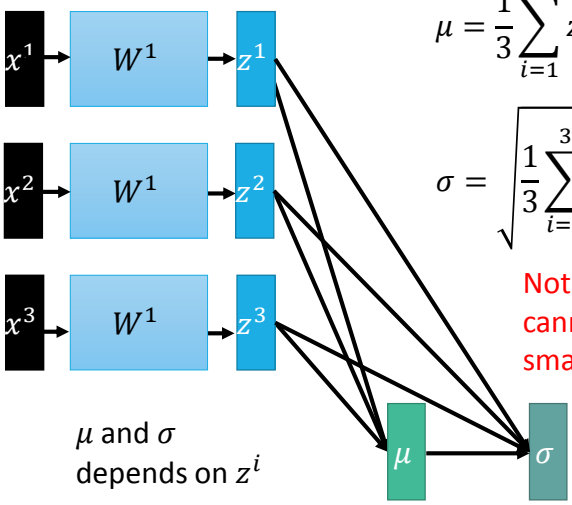
- Batch Normalization (BN) is a normalization method/layer for neural networks.
- Batch Norm is a normalization technique done between the layers of a Neural Network instead of in the raw data.
- Moreover, the Batch Norm helps to **stabilize these shifting distributions** from one iteration to the next, and thus speeds up training.
- Batch Normalization – Is a process normalize each scalar feature independently, by making
  - it have the mean of zero and the variance of 1
  - and then scale and shift the normalized value *for each training mini-batch*

thus reducing internal covariate shift fixing the distribution of the layer inputs x as the training progresses.

Batch



Batch normalization



$$\mu = \frac{1}{3}\sum_{i=1}^{3} z^i$$

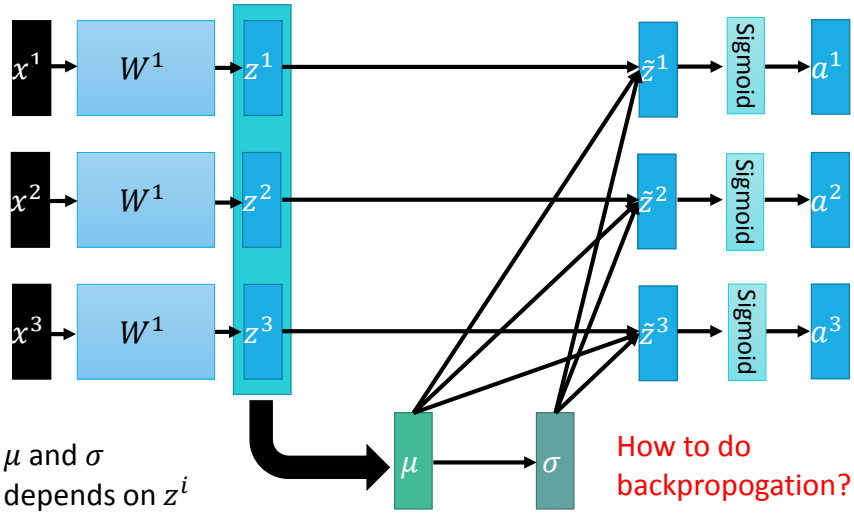
$$\sigma = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(z^i - \mu)^2}$$

Note: Batch normalization cannot be applied on small batch.

μ and σ depends on z^i

Batch normalization

$$\tilde{z}^i = \frac{z^i - \mu}{\sigma}$$



μ and σ depends on z^i

How to do backpropogation?

Batch normalization

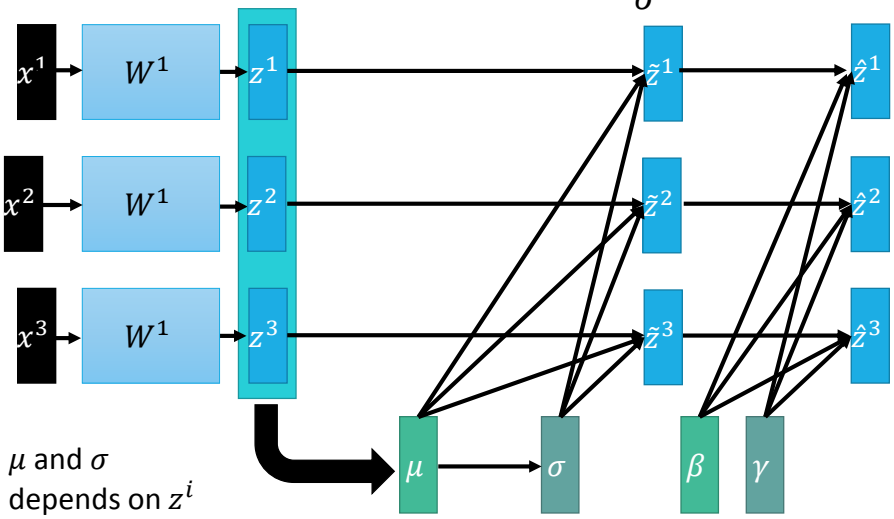
- It is done **along mini-batches** instead of the full data set. It serves to **speed up training** and **use higher learning rates**, making learning easier.
- Normally, **large learning rates** may increase the scale of layer **parameters**, which then amplify the **gradient during** backpropagation and lead to the model **explosion**.
- However, with Batch Normalization, backpropagation through a layer is unaffected by the scale of its parameters.
- The output of the batch norm layer, has γ *and* β parameters. Those parameters will be learned to best represent your activations. Those parameters allows a learnable (scale and shift) factor.

$$\hat{z}^i = \gamma \odot \tilde{z}^i + \beta$$

- They shift the mean and standard deviation, respectively

Batch normalization

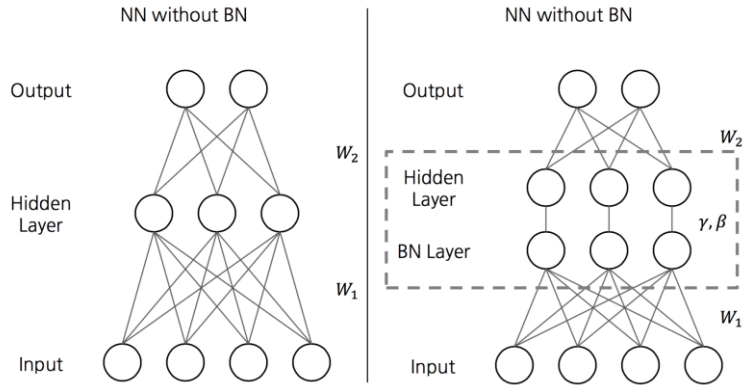
$$\tilde{z}^i = \frac{z^i - \mu}{\sigma} \qquad \hat{z}^i = \gamma \odot \tilde{z}^i + \beta$$



The proposed solution: To add an extra layer

we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value:

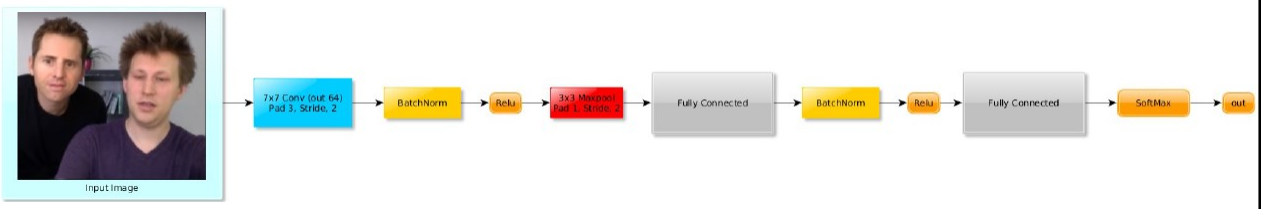
$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}.$$



A new layer is added so the gradient can "see" the normalization and make adjustments if needed.

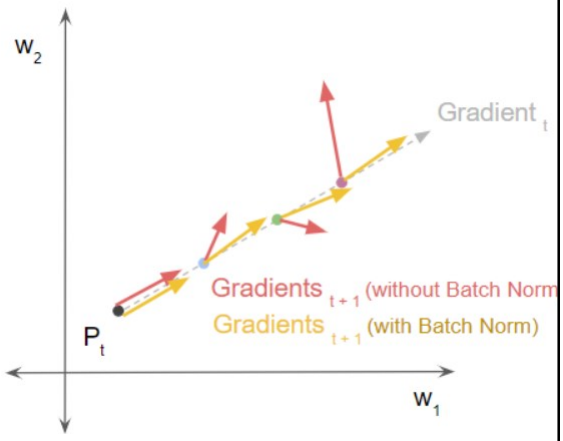
Where to use the Batch-Norm layer in CNN

- The batch norm layer is used after linear layers (ie: FC, conv), and before the non-linear layers (Relu).
- There is actually 2 batch norm implementations one for FC layer and the other for conv layers.



Batch normalization: Other benefits in practice

- BN reduces training times (Because of less Covariate Shift, less exploding/vanishing gradients.) , and make very deep net trainable.
- BN reduces demand for regularization (for generalization), e.g. dropout or L2 norm.
- BN enables training with saturating nonlinearities in deep networks, e.g. sigmoid. (Because the normalization prevents them from getting stuck in saturating ranges, e.g. very high/low values for sigmoid.)



Thanks