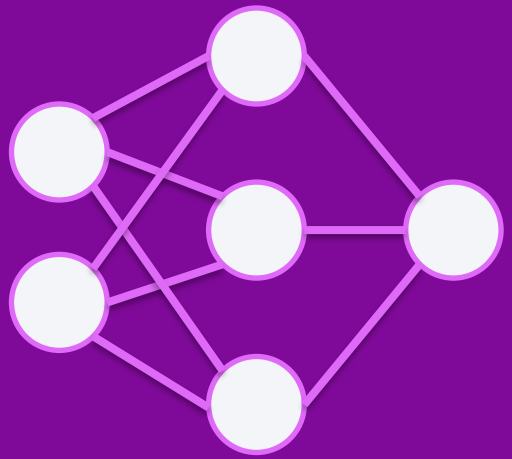
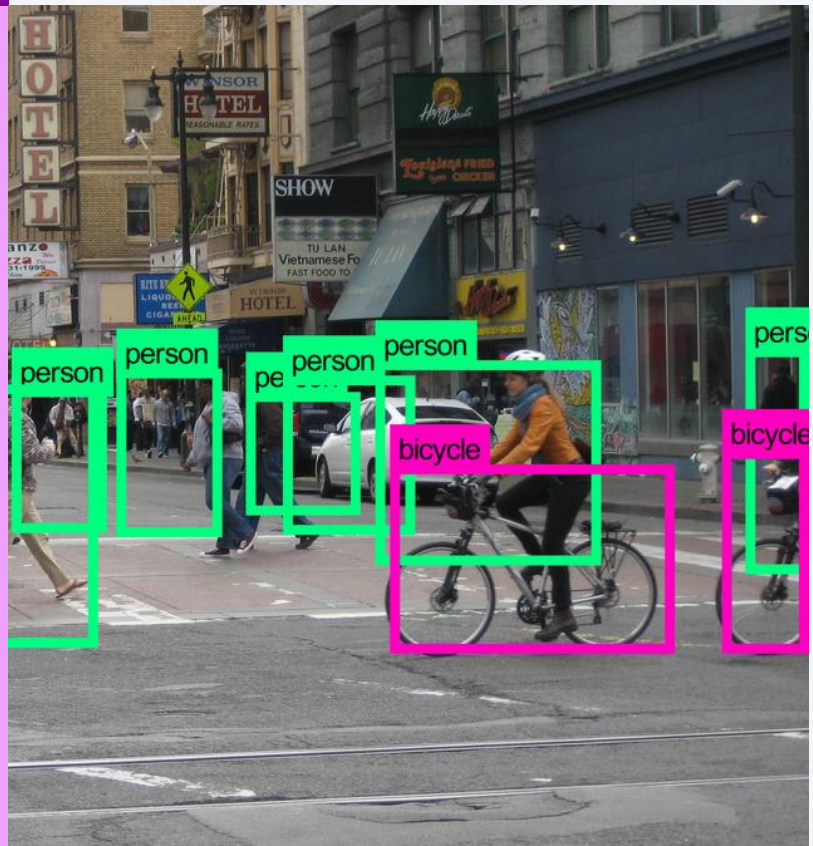
Deep Learning: Computer Vision



Arsham Gholamzadeh Khoee

Alireza Javaheri

LOOP Academy



What is computer vision?

Computer vision is a field of artificial intelligence that trains computers to interpret and understand the visual world. Using digital images from cameras and videos and deep learning models, machines can accurately identify and classify objects — and then react to what they “see.”

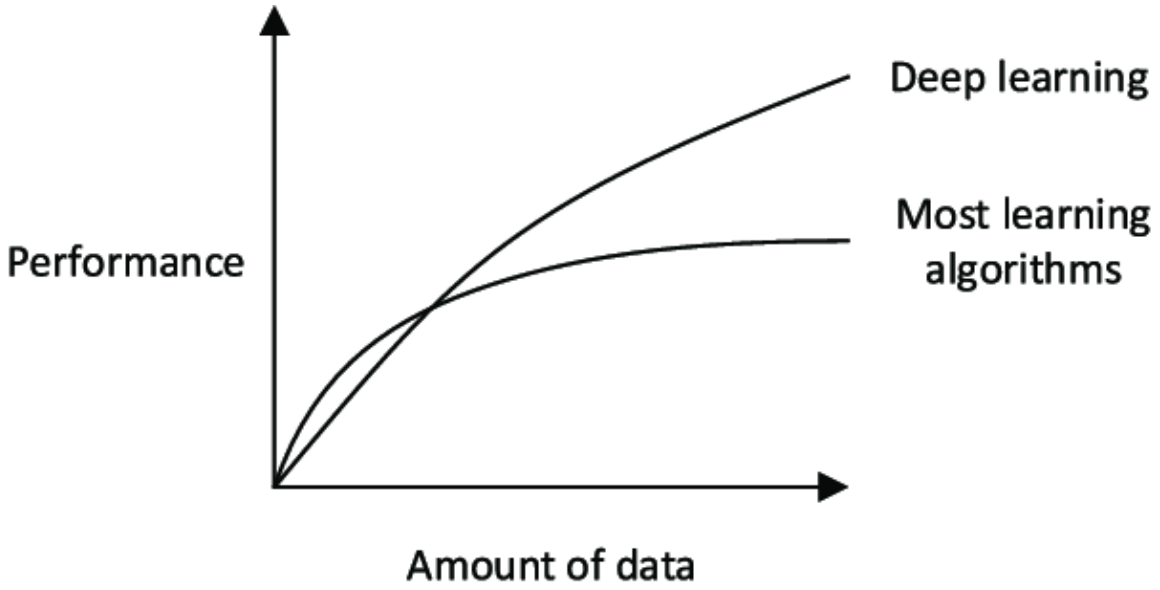
THE MAIN PROBLEM

Finding an algorithm to detect these numbers

Human learning system

Neural networks

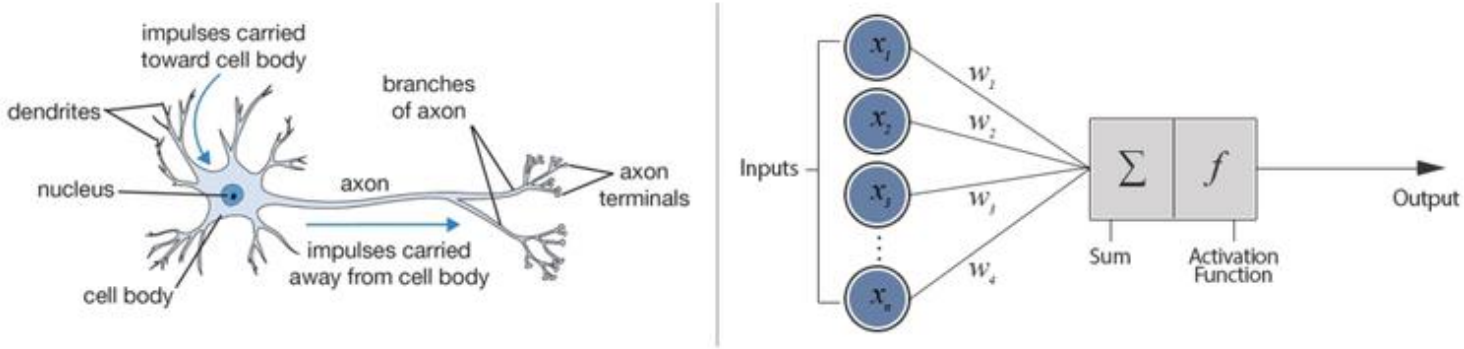
WHY DEEP LEARNING?



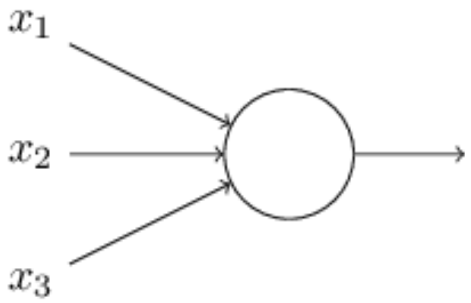
1. ARTIFICIAL NEURAL NETWORKS

Introducing McCulloch-Pitts Model

MCCULLOCH-PITTS MODEL



PERCEPTRON

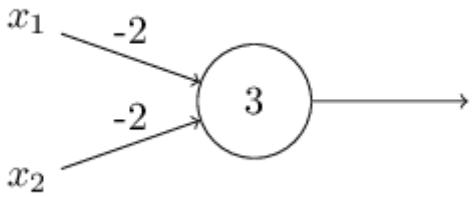


$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

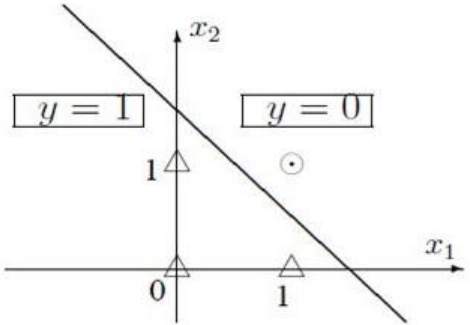
x_1, x_2, x_3 *are binary inputs*

Bias: $b \equiv -threshold$

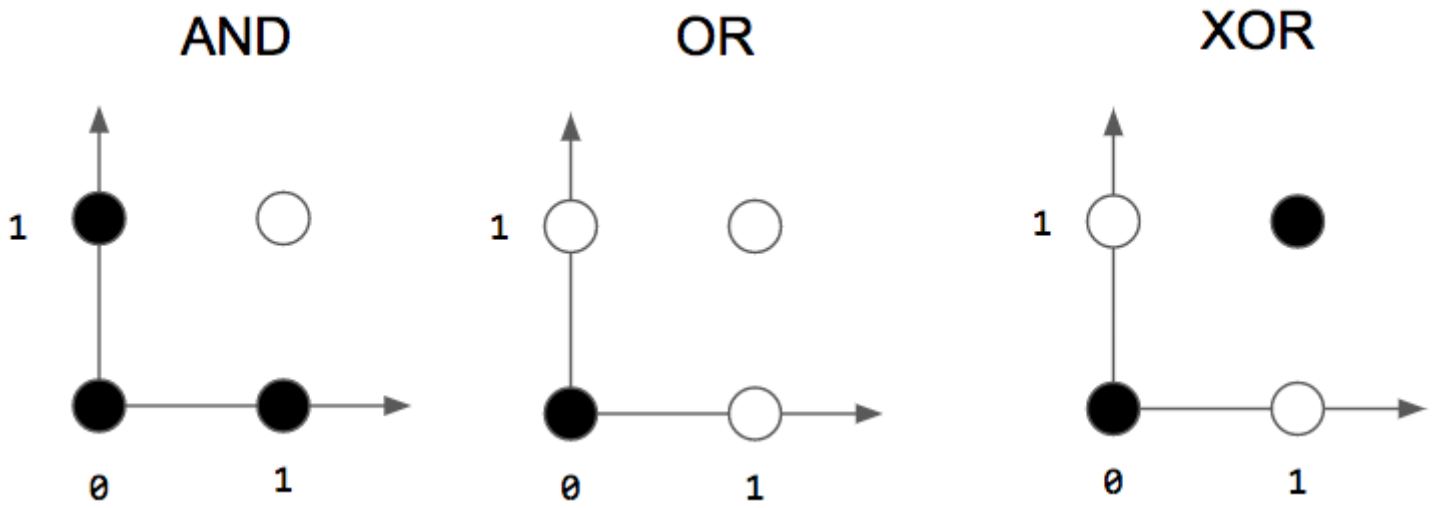
NAND



x_1	0	0	1	1
x_2	0	1	0	1
y	1	1	1	0

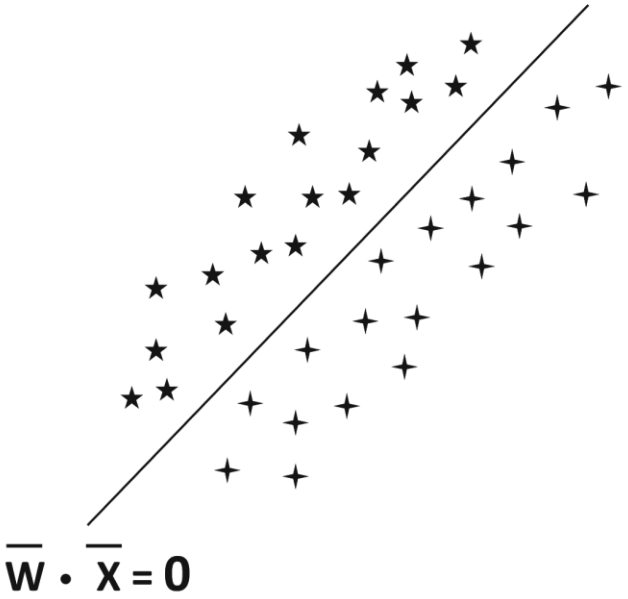


XOR



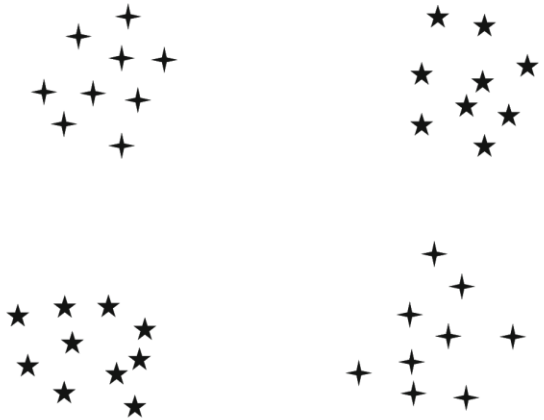
A single perceptron an only solve linearly separable problems

CLASSIFICATION



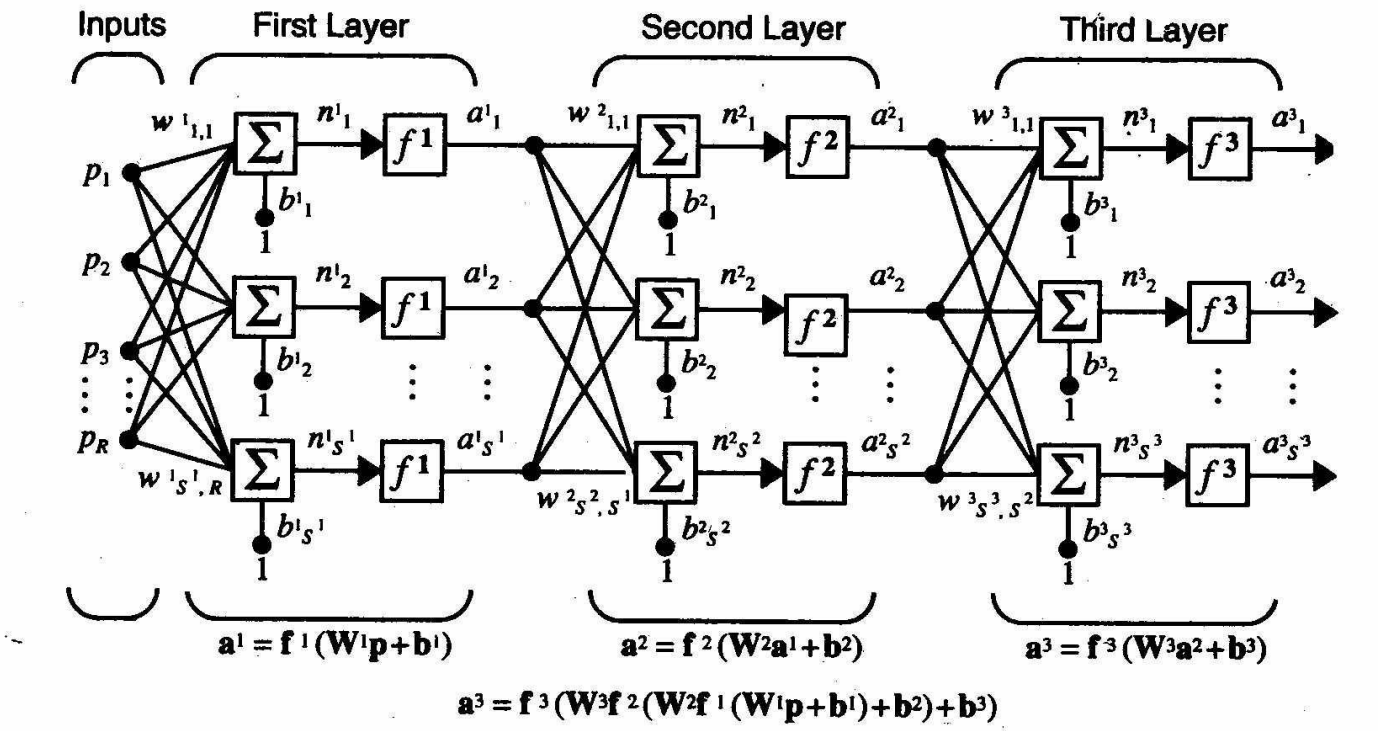
$\overline{W} \cdot \overline{X} = 0$

LINEARLY SEPARABLE



NOT LINEARLY SEPARABLE

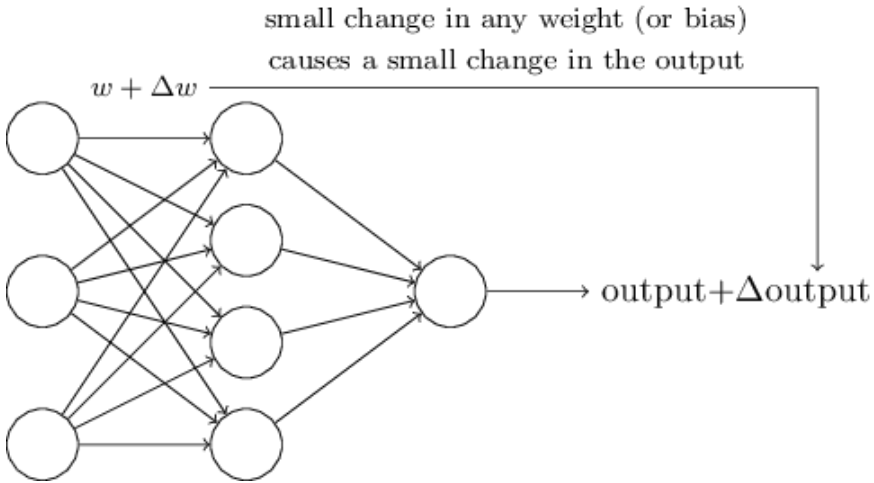
MULTILAYER PERCEPTRON (MLP)



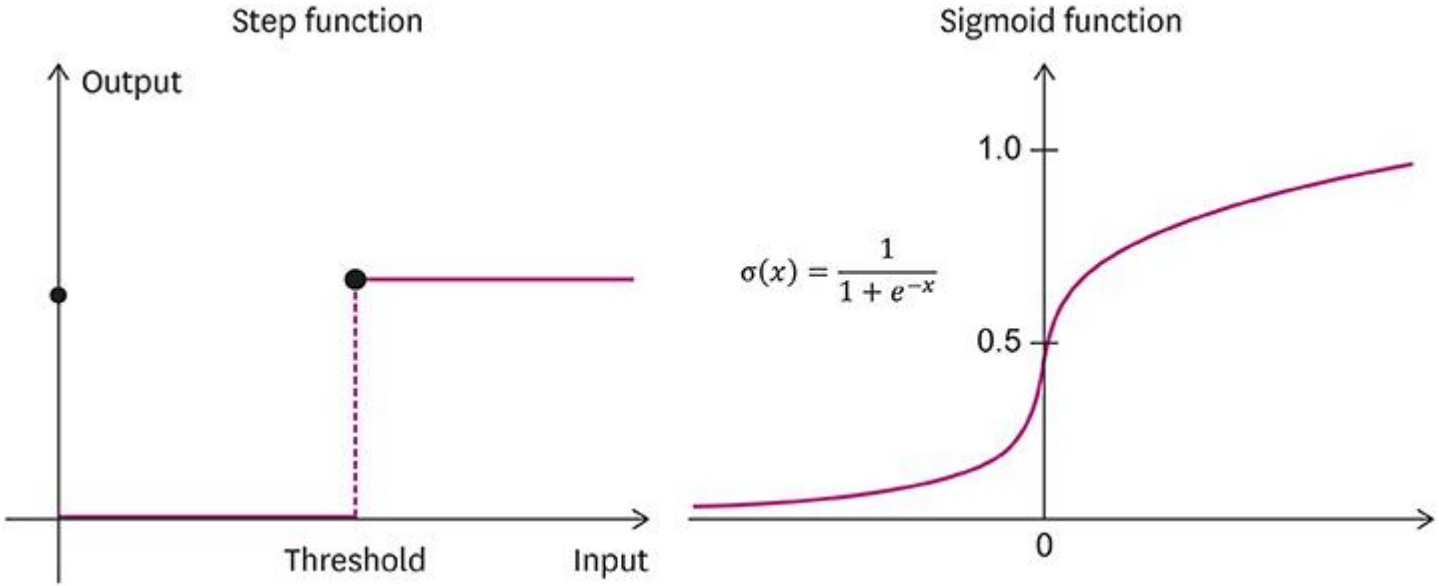
http://www.ccom.ucsd.edu/~cdeotte/programs/neuralnetwork.html

SIGMOID NEURONS

Multilayer perceptrons are often applied to supervised learning problems, they train on a set of input-output pairs and learn to model the correlation (or dependencies) between those inputs and outputs. Training involves adjusting the parameters, or the weights and biases, of the model in order to minimize error.



SIGMOID ACTIVATION FUNCTION



SMOOTHNESS OF SIGMOID

The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change $\Delta output$ in the output from the neuron.

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b,$$

COST FUNCTION

Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

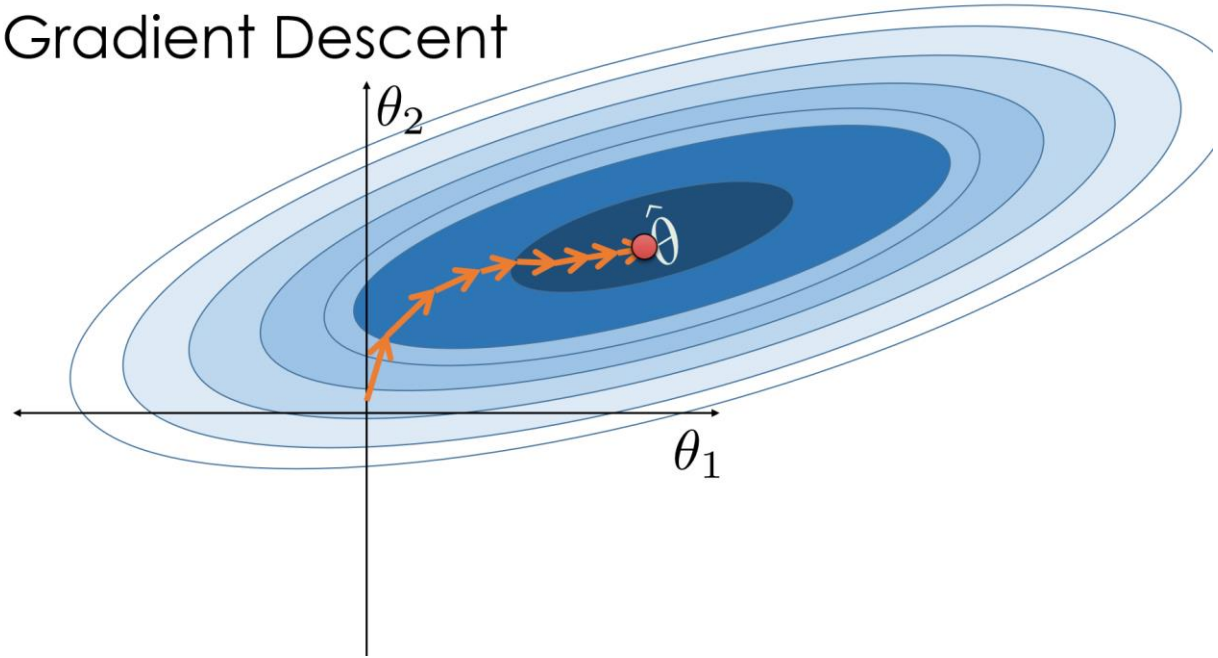
Cost Function:

$$\text{(MSE)} \quad J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

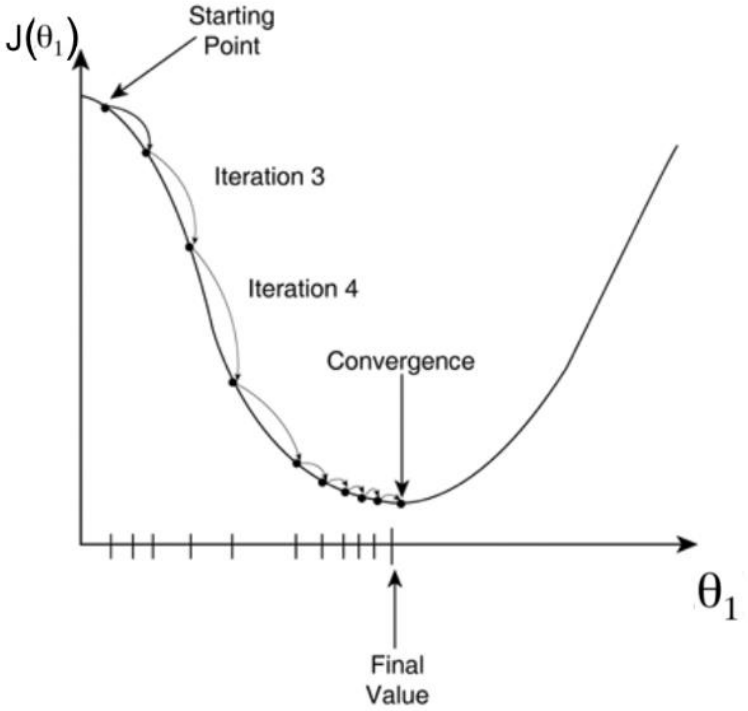
Goal:

$$\underset{\theta_0, \theta_1}{minimize} J(\theta_0, \theta_1)$$

HOW TO ACHIEVE THE GOAL?



GRADIENT DESCENT ALGORITHM



Cost Function – "One Half Mean Squared Error":

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Objective:

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

Derivatives:

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\cdot x^{(i)}$$

DERIVATION OF MSE

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0,\theta_1) = \frac{\partial}{\partial\theta_0}\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}2\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$= \frac{2}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

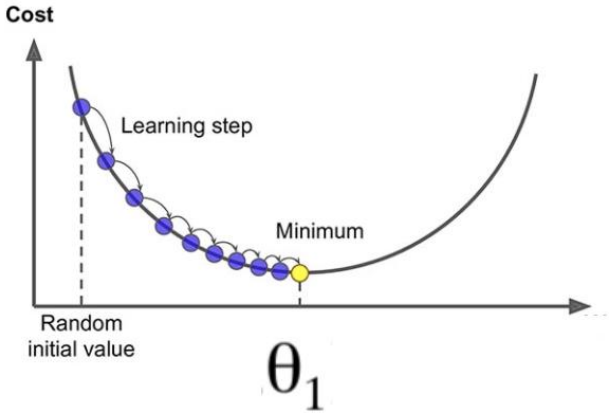
LEARNING RATE

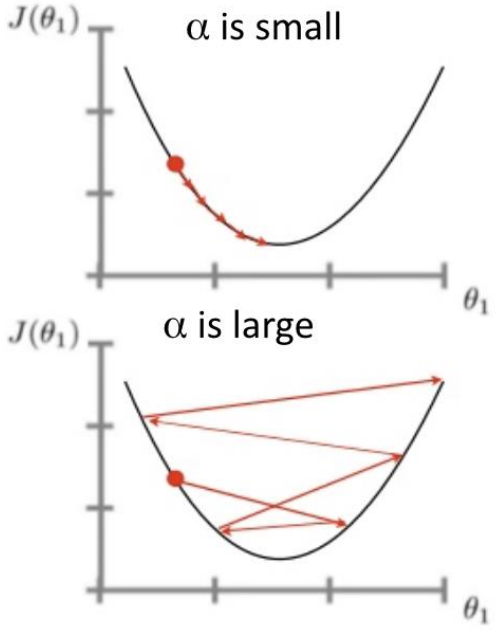
repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}





LOOP Academy

THREE VARIANTS OF GRADIENT DESCENT

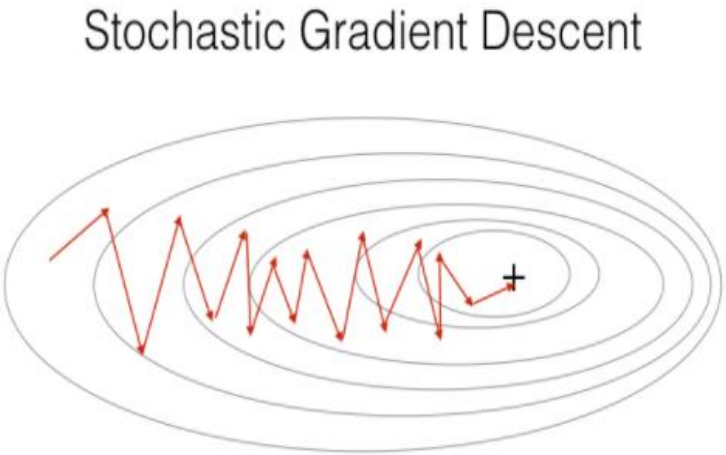
- **Batch gradient descent (BGD):** calculate the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.
- **Stochastic gradient descent (SGD):** calculate the error and updates the model for each example in the training dataset.
- **Mini-Batch gradient descent:** split the training dataset into small batches that are used to calculate model error and updated model coefficients. (the most common implementation of gradient descent used in the field of deep learning) **Mini-Batch gradient descent can find a balance between the robustness of SGD and the efficiency of BGD.**

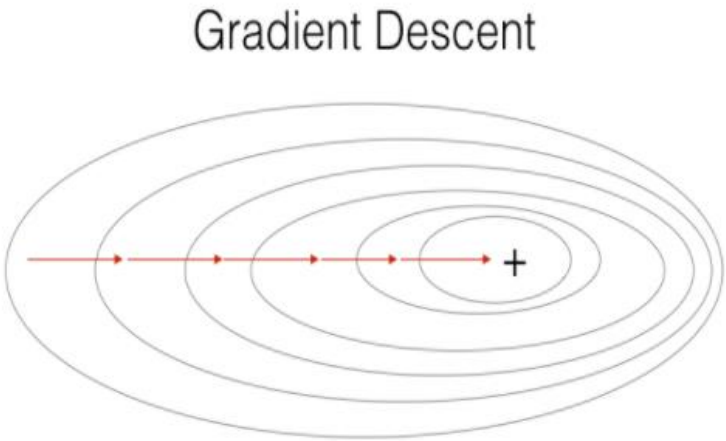
STOCHASTIC GRADIENT DESCENT

However, there is a disadvantage of applying a typical Gradient Descent optimization technique in our dataset. It becomes computationally very expensive to perform because we have to use all of the one million samples for completing one iteration, and it has to be done for every iteration until the minimum point is reached. This problem can be solved by Stochastic Gradient Descent.

The word ‘stochastic’ means a system or a process that is linked with a random probability. Stochastic gradient descent uses this idea to speed up the process of performing gradient descent. Hence, unlike the typical Gradient Descent optimization, instead of using the whole data set for each iteration, we are able to use the cost gradient of only 1 example at each iteration

SGD VS. GRADIENT DESCENT





ADVANTAGES OF GRADIENT DESCENT

- Less oscillations and noisy steps taken towards the global minima of the loss function due to updating the parameters by computing the average of all the training samples rather than the value of a single sample
- It can benefit from the vectorization which increases the speed of processing all training samples together
- It produces a more stable gradient descent convergence and stable error gradient than stochastic gradient descent
- It is computationally efficient as all computer resources are not being used to process a single sample rather are being used for all training samples

DISADVANTAGES OF GRADIENT DESCENT

- Sometimes a stable error gradient can lead to a local minima and unlike stochastic gradient descent no noisy steps are there to help get out of the local minima
- The entire training set can be too large to process in the memory due to which additional memory might be needed
- Depending on computer resources it can take too long for processing all the training samples as a batch

ADVANTAGES OF STOCHASTIC GRADIENT DESCENT

- It is easier to fit into memory due to a single training sample being processed by the network
- It is computationally fast as only one sample is processed at a time
- For larger datasets it can converge faster as it causes updates to the parameters more frequently
- Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums of the loss function (in case the computed position turns out to be the local minimum)

DISADVANTAGES OF STOCHASTIC GRADIENT DESCENT

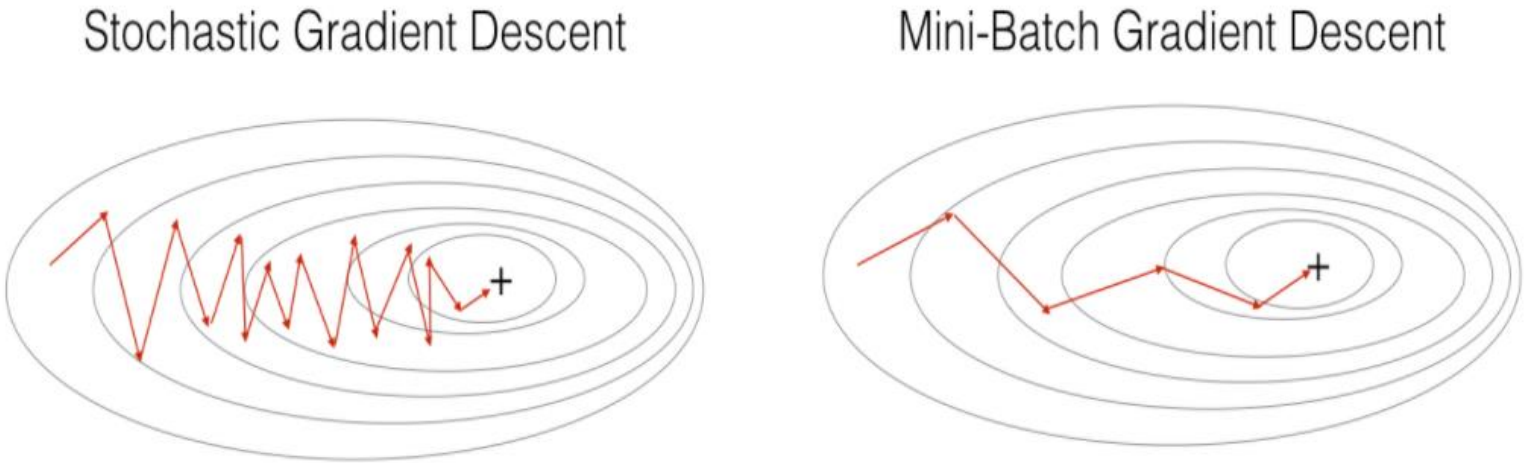
- Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions.
- Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function
- Frequent updates are computationally expensive due to using all resources for processing one training sample at a time
- It loses the advantage of vectorized operations as it deals with only a single example at a time

MINI-BATCH GRADIENT DESCENT

This is a mixture of both stochastic and batch gradient descent. The training set is divided into multiple groups called batches. Each batch has a number of training samples in it. At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network. For example, say the training set has 128 training examples which is divided into 4 batches with each batch containing 32 training examples. This means that the neural network iterate 4 times per an epoch.

This ensures the following advantages of both stochastic and batch gradient descent are used due to which Mini Batch Gradient Descent is most commonly used in practice.

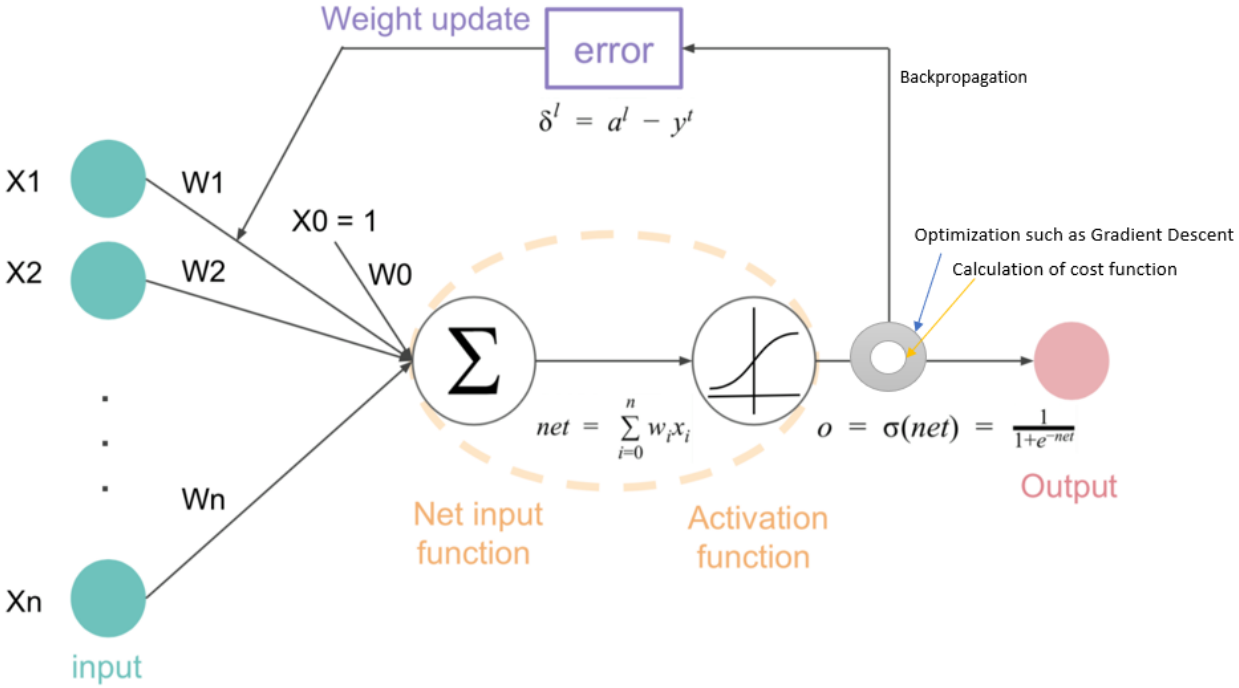
SGD VS. MINI-BATCH GRADIENT DESCENT



WHY MINI-BATCH GRADIENT DESCENT?

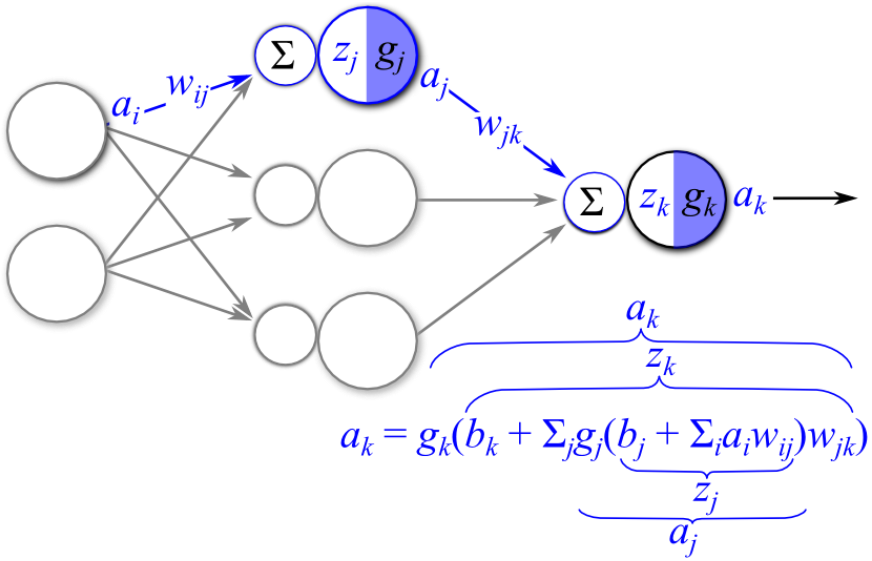
- Easily fits in the memory
- It is computationally efficient
- Benefit from vectorization
- If stuck in local minimums, some noisy steps can lead the way out of them
- Average of the training samples produces stable error gradients and convergence

BACKPROPAGATION



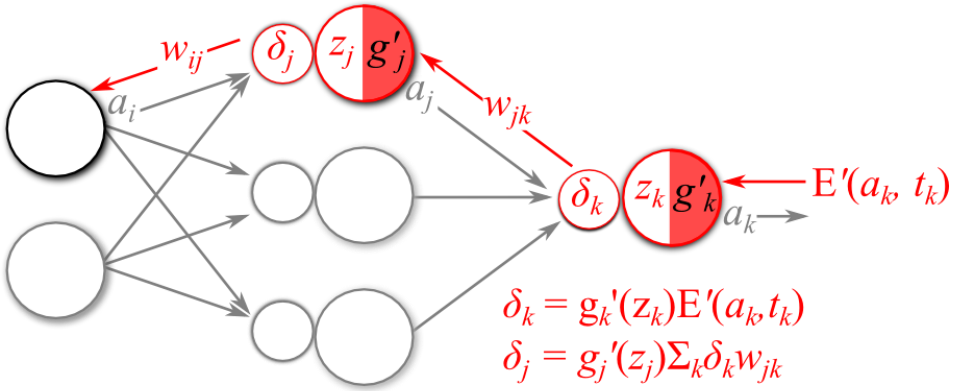
The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters (weights and biases).

I- FORWARD PROPAGATE INPUT SIGNAL



Note that in the Figure a_k could be considered network output (for a network with one hidden layer) or the output of a hidden layer that projects the remainder of the network (in the case of a network with more than one hidden layer). For this discussion, however, we assume that the index k is associated with the output layer of the network, and thus each of the network outputs is designated by a_k. Also note that when implementing this forward-propagation step, we should keep track of the feed-forward pre-activations z_l and activations a_l for all layers l, as these will be used for calculating backpropagated errors and error function gradients.

II- BACKWARD PROPAGATE ERROR SIGNALS



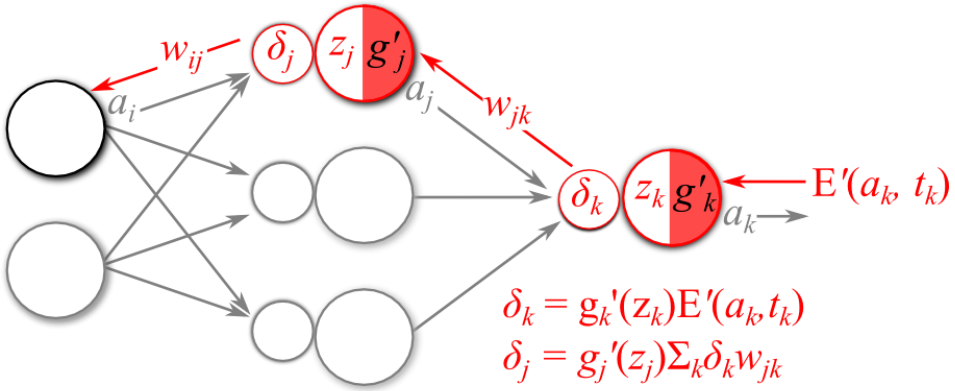
Step II of the algorithm is to calculate the network output error and backpropagate it toward the input. Let's again that we are using the sum of squared differences error function:

$$E = \frac{1}{2}\sum_{k\in K}(a_k - t_k)^2$$

where we sum over the values of all k output units (one in this example). We can now define an "error signal" δ_k at the output node that will be backpropagated toward the input. The error signal is calculated as follows:

$$\begin{aligned}\delta_k &= g'_k(z_k)E'(a_k, t_k)\\ &= g'_k(z_k)(a_k - t_k)\end{aligned}$$

II- BACKWARD PROPAGATE ERROR SIGNALS

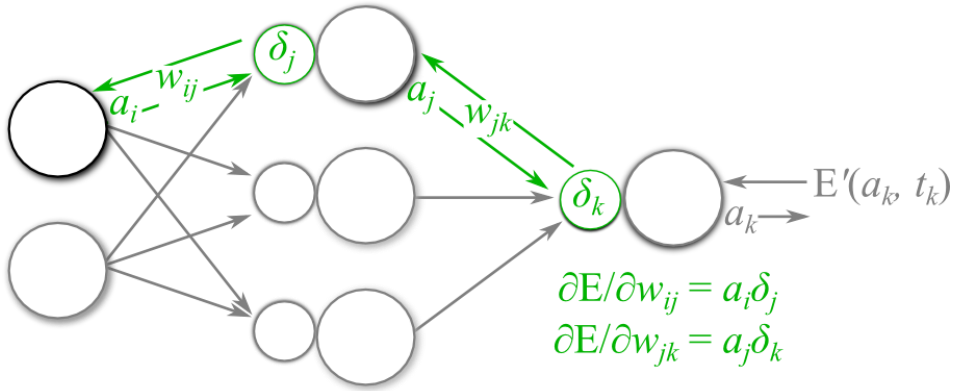


Thus the error signal essentially weights the gradient of the error function by the gradient of the output activation function (notice there is a z_k term is used in this calculation, which is why we keep it around during the forward-propagation step). We can continue backpropagating the error signal toward the input by passing δ_k through the output layer weights w_{jk}, summing over all output nodes, and passing the result through the gradient of the activation function at the hidden layer $g'_j(z_j)$. Performing these operations results in the back-propagated error signal for the hidden layer, δ_j:

$$\delta_j = g'_j(z_j) \sum_k \delta_k w_{jk}$$

For networks that have more than one hidden layer, this error backpropagation procedure can continue for layers $j-1, j-2, \ldots$, etc.

III- CALCULATE PARAMETER GRADIENTS

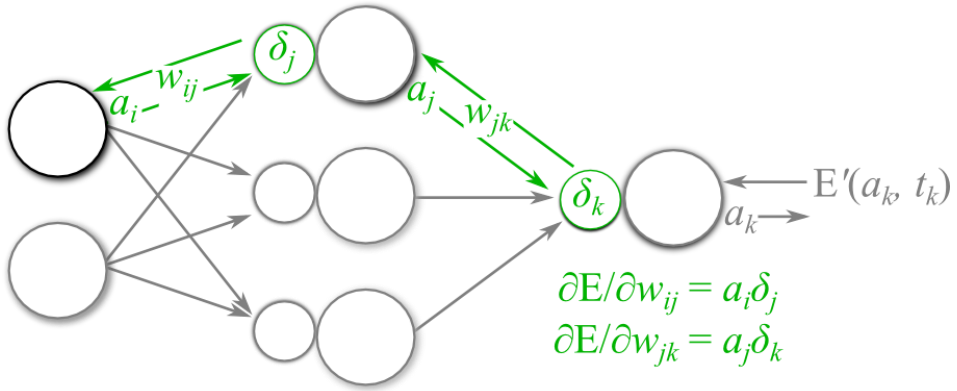


Step III of the backpropagation algorithm is to calculate the gradients of the error function with respect to the model parameters at each layer l using the forward signals a_{l-1}, and the backward error signals δ_l . If one considers the model weights $w_{l-1,l}$ at a layer l as linking the forward signal a_{l-1} to the error signal δ_l, then the gradient of the error function with respect to those weights is:

$$\frac{\partial E}{\partial w_{l-1,l}} = a_{l-1}\delta_l$$

Thus the gradient of the error function with respect to the model weight at each layer can be efficiently calculated by simply keeping track of the forward-propagated activations feeding into that layer from below, and weighting those activations by the backward-propagated error signals feeding into that layer from above!

III- CALCULATE PARAMETER GRADIENTS



What about the bias parameters? It turns out that the same gradient rule used for the weight weights applies, except that “feed-forward activations” for biases are always +1 (in part I). Thus the bias gradients for layer l are simply:

$$\frac{\partial E}{\partial b_l} = (1)\delta_l = \delta_l$$

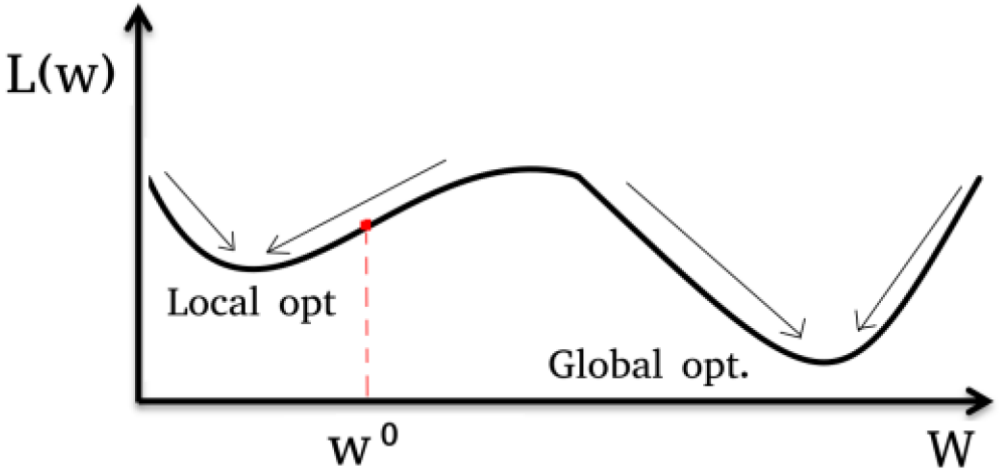
IV- UPDATE PARAETERS

$$w_{ij} = w_{ij} - \eta(\partial E/\partial w_{ij})$$
$$w_{jk} = w_{jk} - \eta(\partial E/\partial w_{jk})$$
for learning rate η

The fourth and final step of the backpropagation algorithm is to update the model parameters based on the gradients calculated in Step III. Note that the gradients point in the direction in parameter space that will increase the value of the error function. Thus when updating the model parameters we should choose to go in the opposite direction. How far do we travel in that direction? That is generally determined by a user-defined step size (aka learning rate) parameter, η. Thus given the parameter gradients and the step size, the weights and biases for a given layer are updated accordingly:

$$w_{l-1,l} \leftarrow w_{l-1,l} - \eta \frac{\partial E}{\partial w_{l-1,l}}$$
$$b_l \leftarrow b_l - \eta \frac{\partial E}{\partial b_l}$$

INITIAL WEIGHTS MATTER

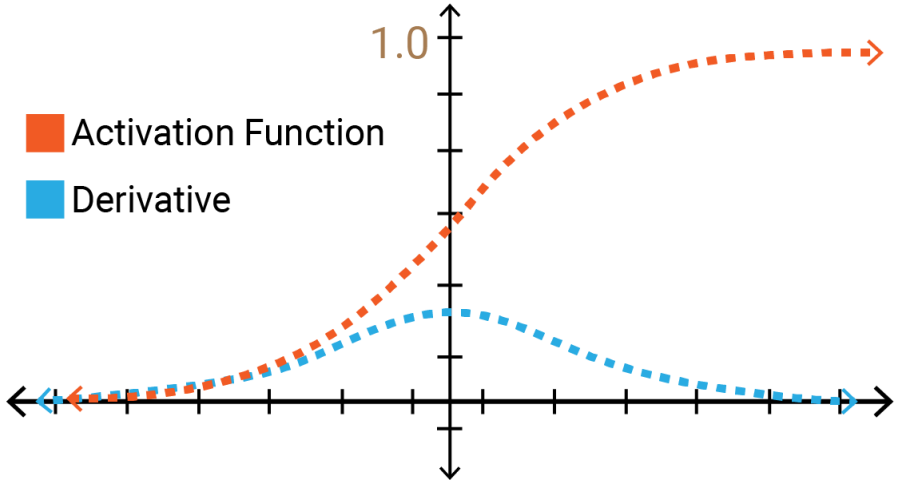


SYMMETRY BREAKING

- **If all weights start with equal values and if the solution requires that unequal weights be developed, the system can never learn.**
- This is because error is propagated back through the weights in proportion to the values of the weights. This means that all hidden units connected directly to the output units will get identical error signals, and, since the weight changes depend on the error signals, the weights from those units to the output units must always be the same. The system is starting out at a kind of unstable equilibrium point that keeps the weights equal, but it is higher than some neighboring points on the error surface, and once it moves away to one of these points, it will never return. We counteract this problem by starting the system with small random weights. Under these conditions symmetry problems of this kind do not arise.

EXPLODING AND VANISHING GRADIENT PROBLEM

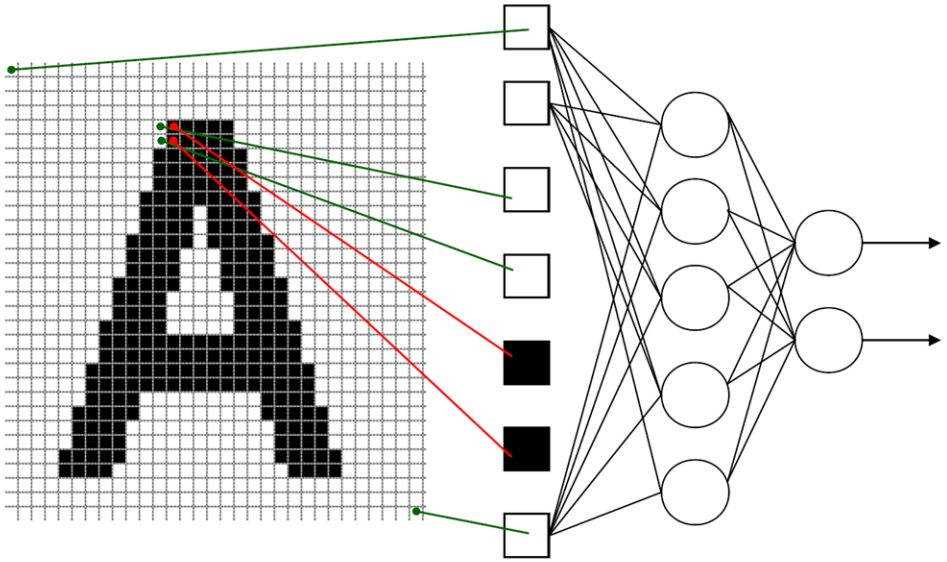
Consider the graph of sigmoid function and it's derivative. Observe that for very large values for sigmoid function the derivative takes a very low value. If the neural network has many hidden layers, the gradients in the earlier layers will become very low as we multiply the derivatives of each layer. As a result, learning in the earlier layers becomes very slow.



2.
CONVOLUTIONAL NEURAL NETWORKS

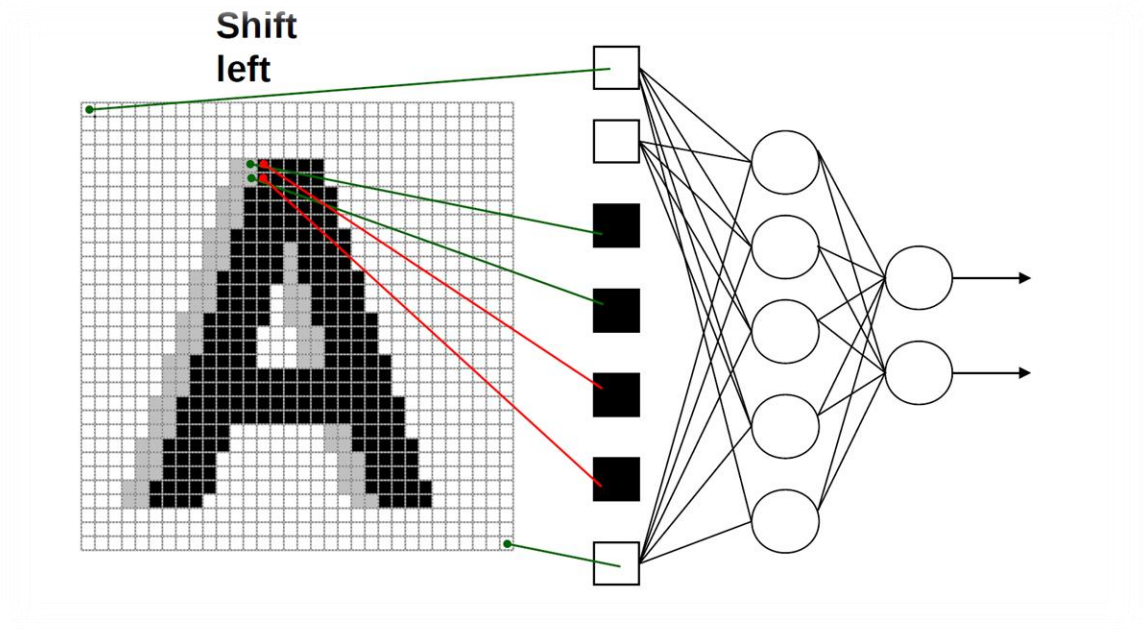
Introducing CNNs to analyzing visual imagery.

DRAWBACKS OF MLP



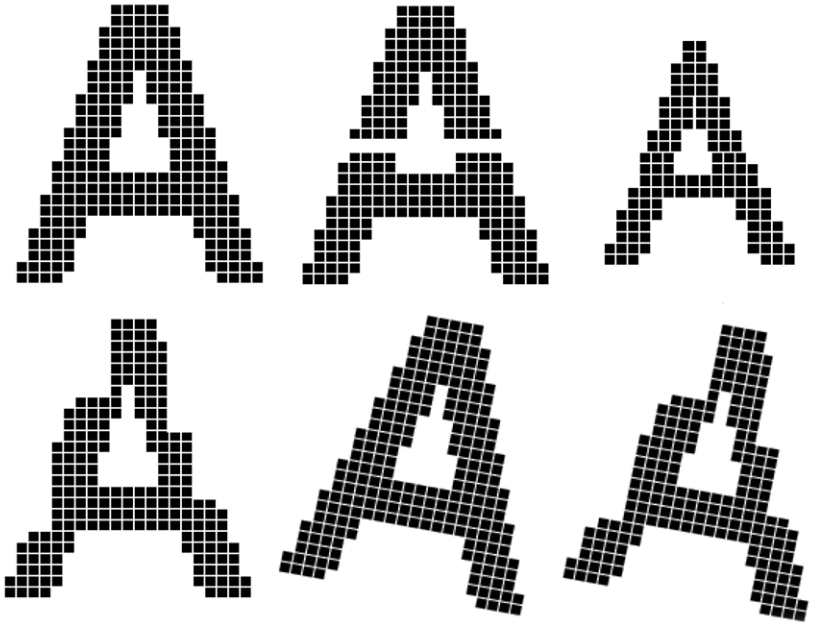
Little or no invariance to shifting, scaling, and other forms of distortion

DRAWBACKS OF MLP



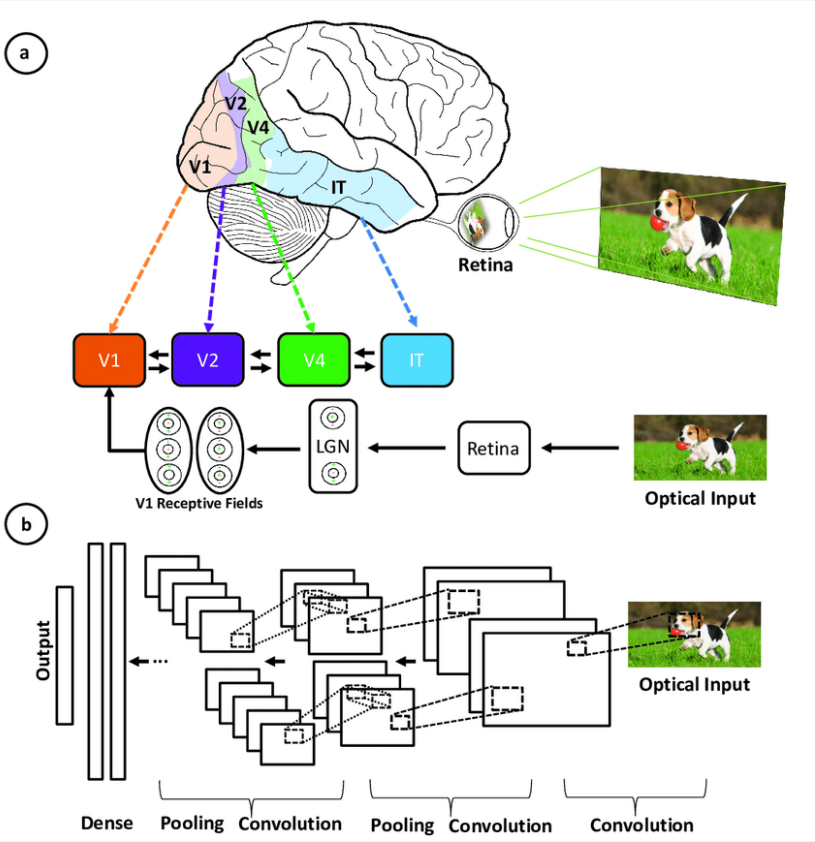
Little or no invariance to shifting, scaling, and other forms of distortion

DRAWBACKS OF MLP



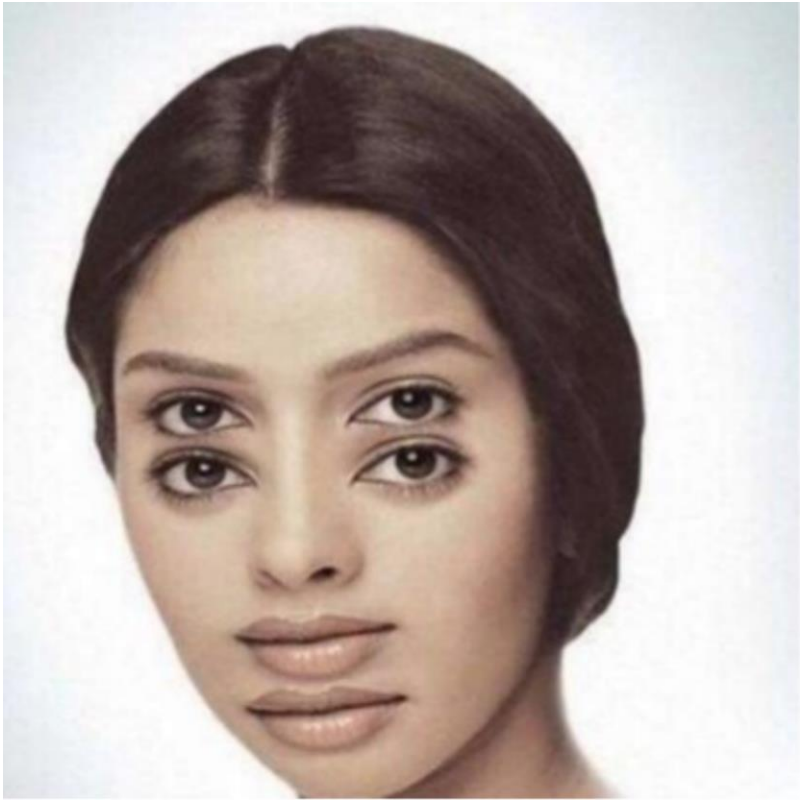
Little or no invariance to shifting, scaling, and other forms of distortion

HUMAN VISUAL SYSTEM MODEL VS. CNN



- The first layer is responsible for detecting lines, edges, changes in brightness, and other simple features.
- The information is then passed onto the next layer, which combines simple features to build detectors that can identify simple shapes.
- The process continues in the next layer and the next, becoming more and more abstract with every layer. The deeper layer will be able to extract high-order features such as shapes or specific objects.
- The last layers of the network will integrate all of those complex features and produce classification predictions.
- The predicted value will be compared to the correct output, where those that are wrongly classified will cause a large error gap and will cause the learning process to backpropagate to make changes to the parameter in order to give out a more accurate outcome.
- The network goes back and forth, correcting itself until the satisfying output is achieved (where the error is minimised).

IS YOUR VISUAL SYSTEM TRAINED?



AN IMAGE AS A MATRIX

What We See

```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

What Computers See

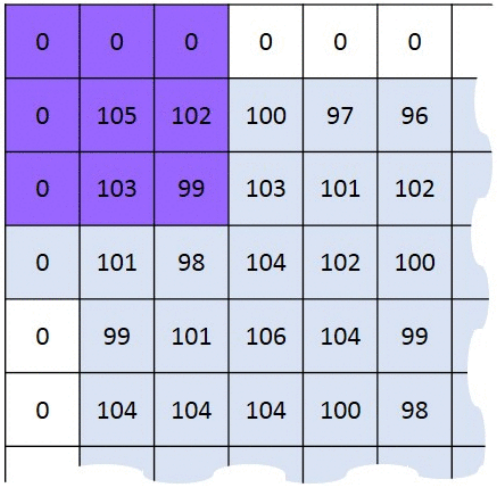
CONVOLUTION OPERATOR

Convolution is a mathematical way of combining two signals to form a third signal.

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m].$$

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau.$$

CONVOLVING A FILTER ON THE INPUT IMAGE

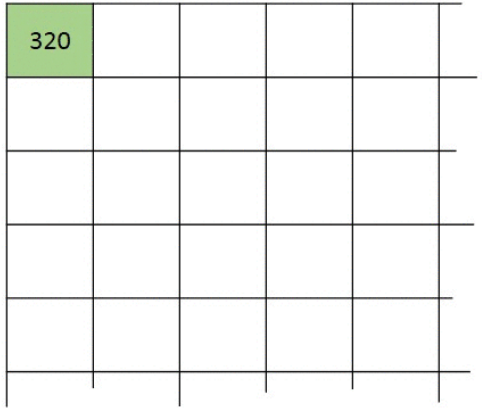


0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Image Matrix

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0



320					

Output Matrix

$$0*0+0*-1+0*0$$
$$+0*-1+105*5+102*-1$$
$$+0*0+103*-1+99*0 \quad = 320$$

Convolution with horizontal and vertical strides = 1

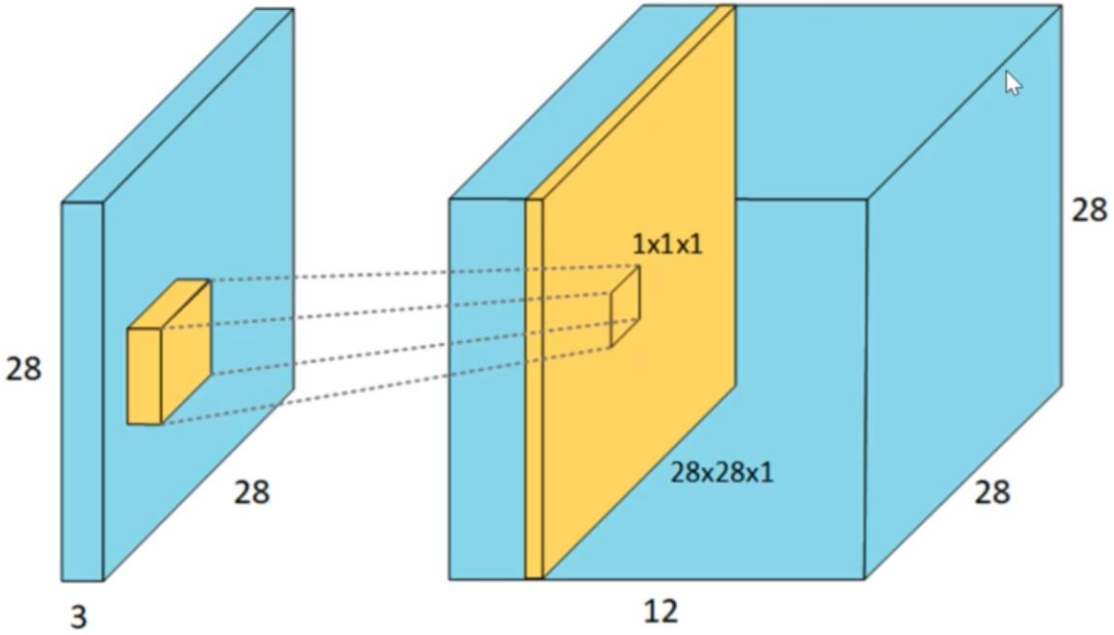
UNDERSTANDING THE CONVOLUTION FILTER

$*$

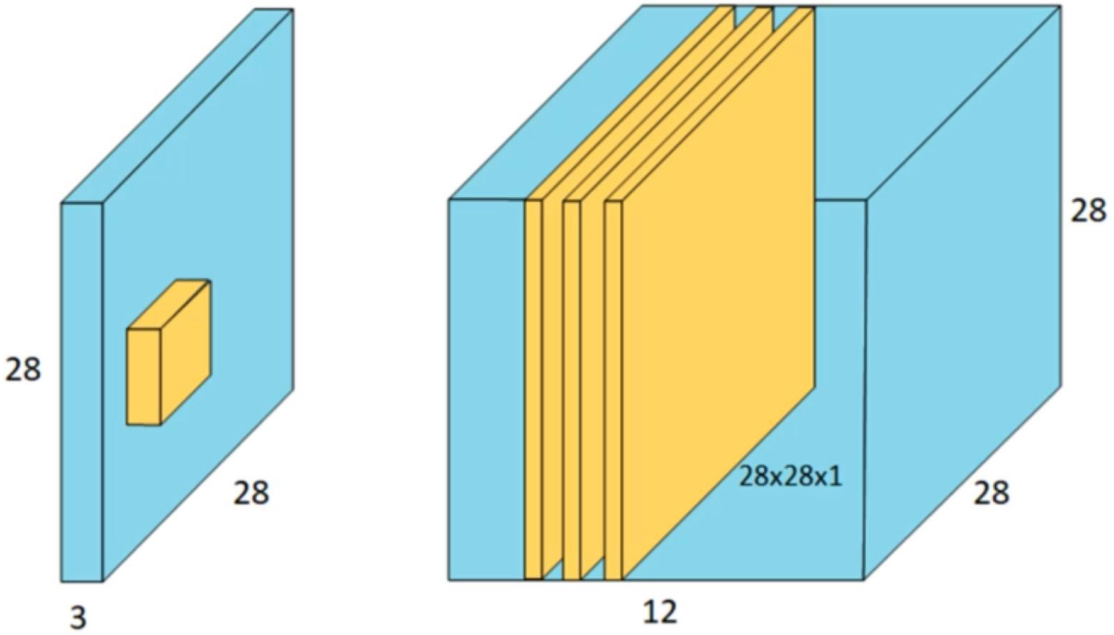
1	0	-1
2	0	-2
1	0	-1

For better illustration you can visit: https://setosa.io/ev/image-kernels/

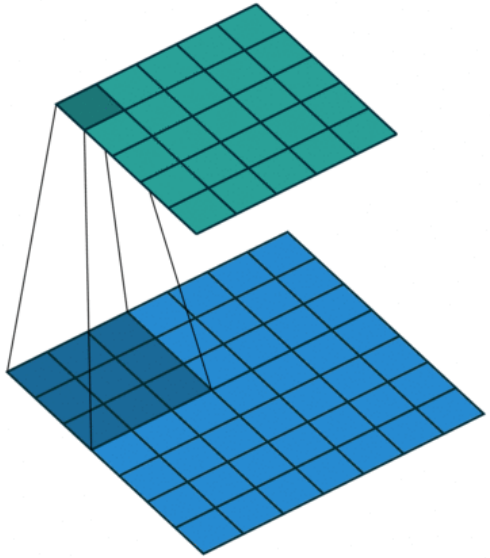
MULTIPLE KERNELS



MULTIPLE KERNELS

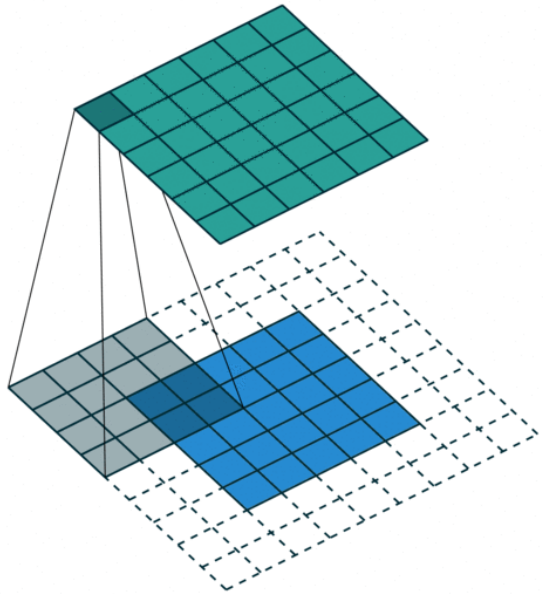


STRIDES



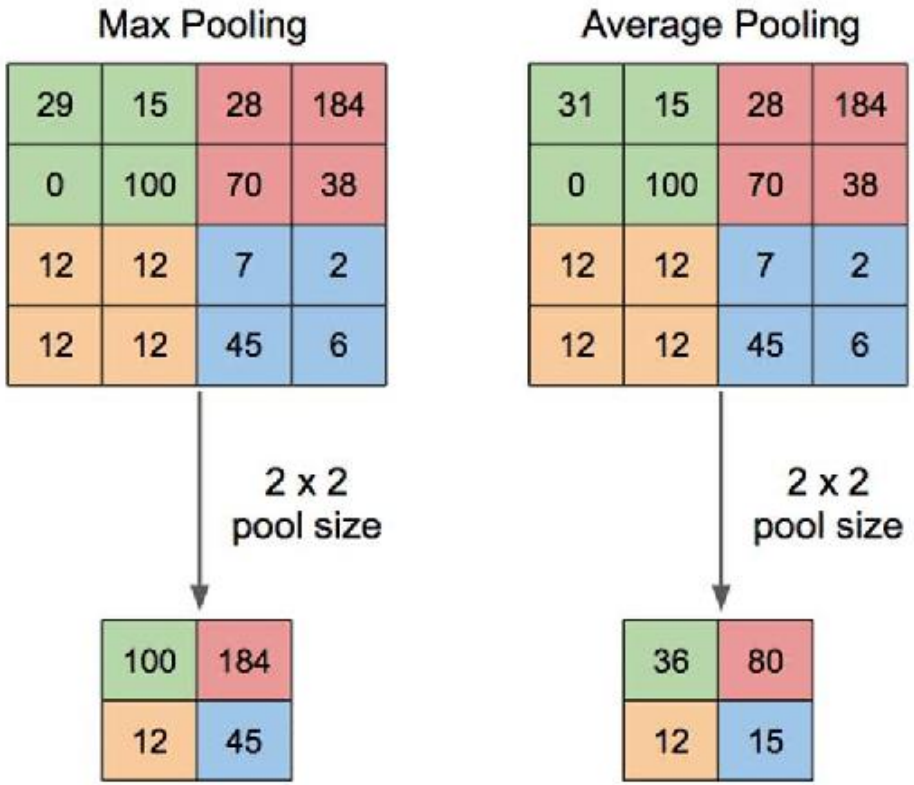
- Strides: increment step size for the convolution operator
- Reduces the size of the output map

PADDING



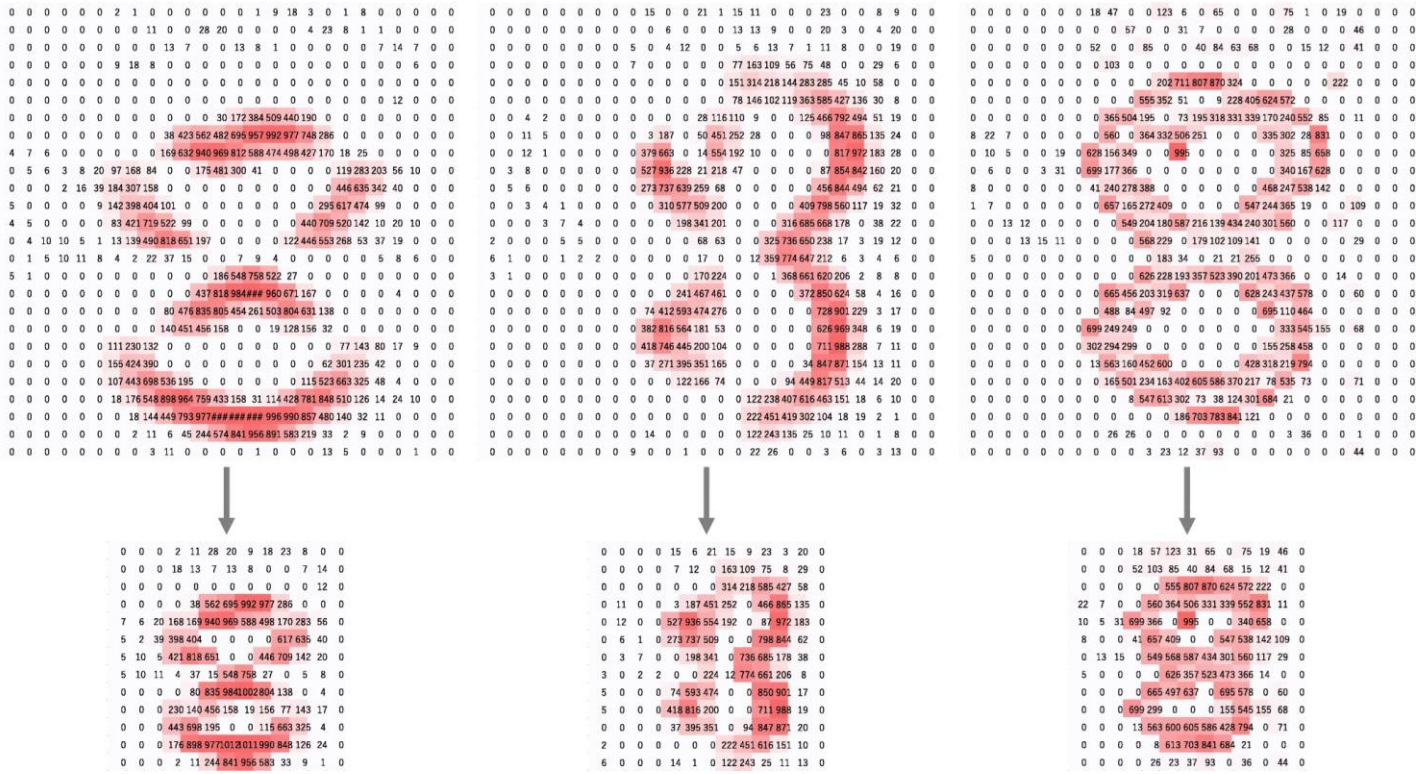
- Padding: artificially fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields

POOLING LAYERS

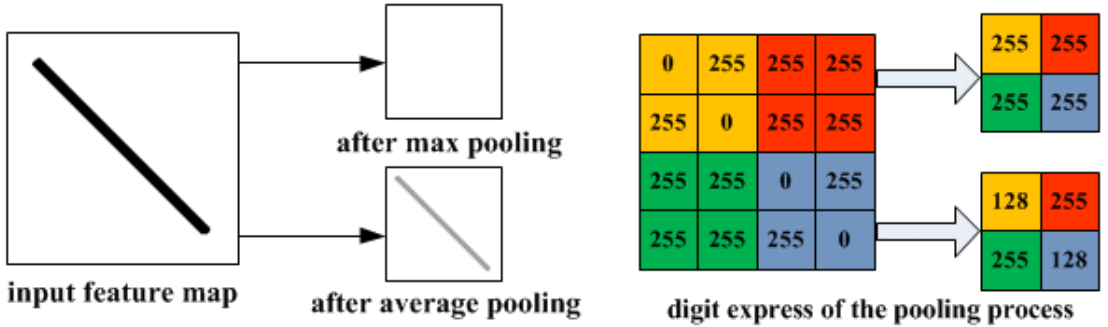


- Convolutional layers provide activation maps.
- Pooling layer applies non-linear downsampling on activation maps.
- Pooling is aggressive (discard info); the trend is to use smaller filter size and abandon pooling

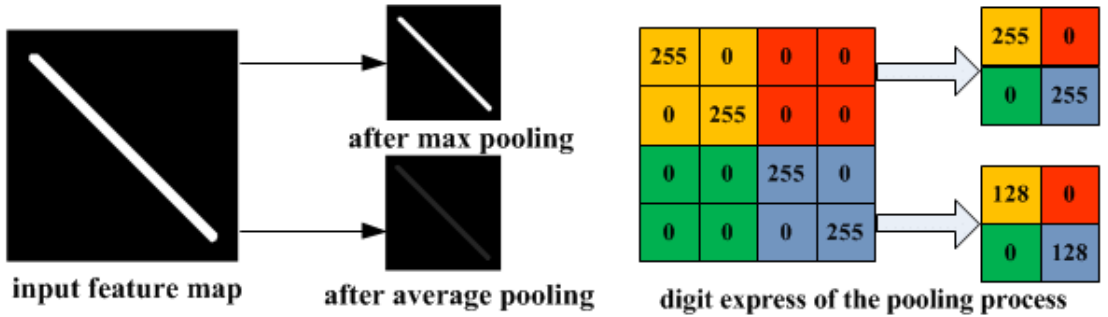
DOWNSAMPLING



MAX POOLING VS. AVERAGE POOLING



(a) Illustration of max pooling drawback



(b) Illustration of average pooling drawback

LOSS LAYER

Binary case $-y \log \hat{y} - (1-y) \log(1-\hat{y})$

General case $-\sum_i y_i \log(\hat{y}_i)$

- Cross-Entropy loss (works well for classification, e.g., image classification)
- Mean Squared Error (works well for regression task, e.g., Behavioral Cloning)

CROSS-ENTROPY VS. QUADRATIC COST (MSE)

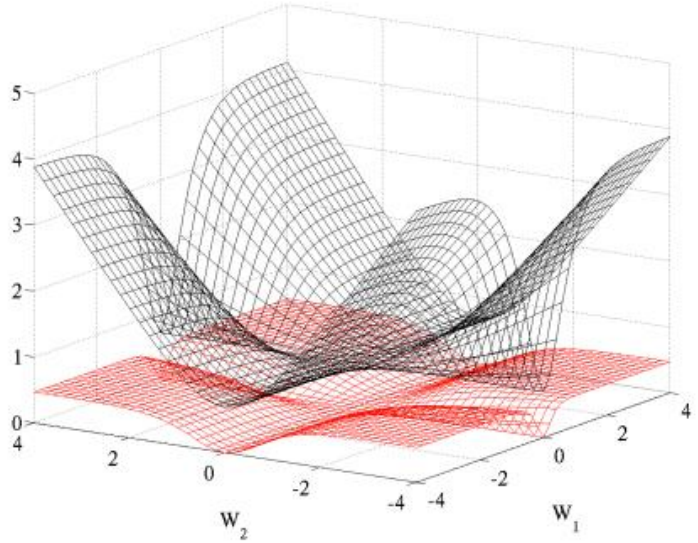
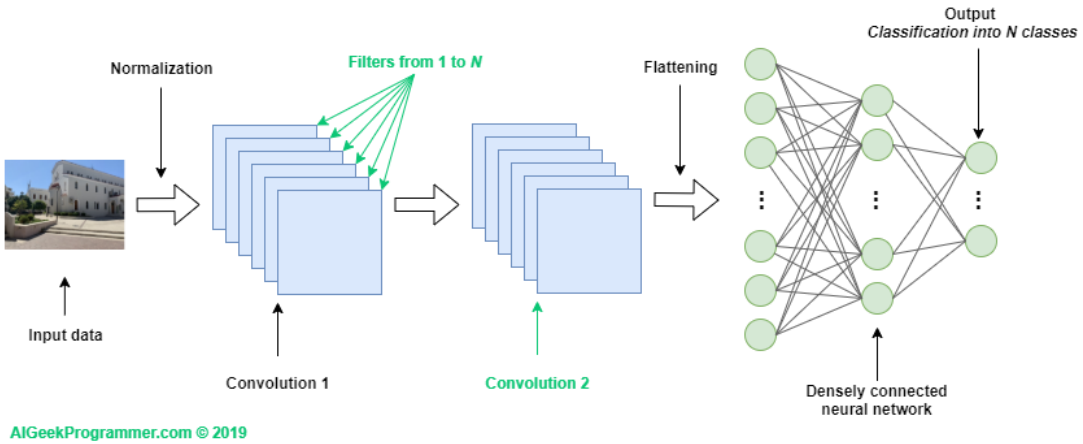


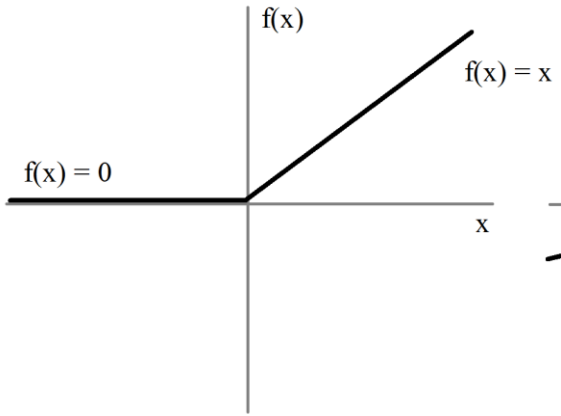
Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.*

FULLY CONNECTED LAYER

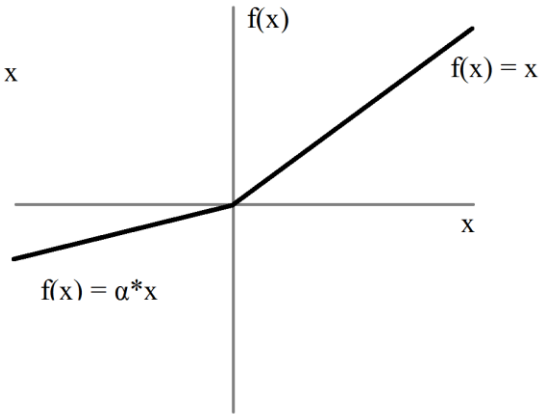


- Regular neural network
- Can view as the final learning phase, which maps extracted visual features to desired outputs
- Usually adaptive to classification/encoding tasks
- Common output is a vector, which is then passed through softmax to represent confidence of classification
- The outputs can also be used as “bottleneck”

ACTIVATION FUNCTIONS



ReLU



Leaky ReLU

Other types:
Randomized Leaky ReLU, Parameterized ReLU
Exponential Linear Units (ELU), Scaled Exponential Linear Units
Tanh, hardtanh, softtanh, softsign, softmax, softplus...

- Used to increase non-linearity of the network without affecting receptive fields of conv layers
- Prefer ReLU, results in faster training
- LeakyReLU addresses the vanishing gradient problem

SOFTMAX

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

Given sample vector input **x** and weight vectors $\{\mathbf{w}_i\}$, the predicted probability of y = j

- A special kind of activation layer, usually at the end of fully-connected layer outputs
- Can be viewed as a fancy normalizer (a.k.a. Normalized exponential function)
- Produce a discrete probability distribution vector
- Very convenient when combined with cross-entropy loss

LOOP Academy

WEIGHT INITIALIZATION: REVISITED

Why shouldn't you initialize the weights with zeroes or randomly (without knowing the distribution):

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it's too tiny to be useful.
- If the weights in a network start too large, then the signal grows as it passes through each layer until it's too massive to be useful.

XAVIER INITIALIZATION

Xavier Initialization initializes the weights in your network by drawing them from a distribution with zero mean and a specific variance,

$$Var(w_i) = \frac{1}{fan\_in}$$

where fan_in is the number of incoming neurons.

It draws samples from a truncated normal distribution centered on 0 with stddev = sqrt(1 / fan_in) where fan_in is the number of input units in the weight tensor.

Generally used with tanh activation.

Also generally,

$$Var(w_i) = \frac{1}{fan\_in + fan\_out}$$

is used where fan_out is the number of neurons the result is fed to.

HE INITIALIZATION

This method of initializing became famous through a paper submitted in 2015 by He-et-al, and is similar to Xavier initialization, with the factor multiplied by two. In this method, the weights are initialized keeping in mind the size of the previous layer which helps in attaining a global minimum of the cost function faster and more efficiently. The weights are still random but differ in range depending on the size of the previous layer of neurons. This provides a controlled initialization hence the faster and more efficient gradient descent.

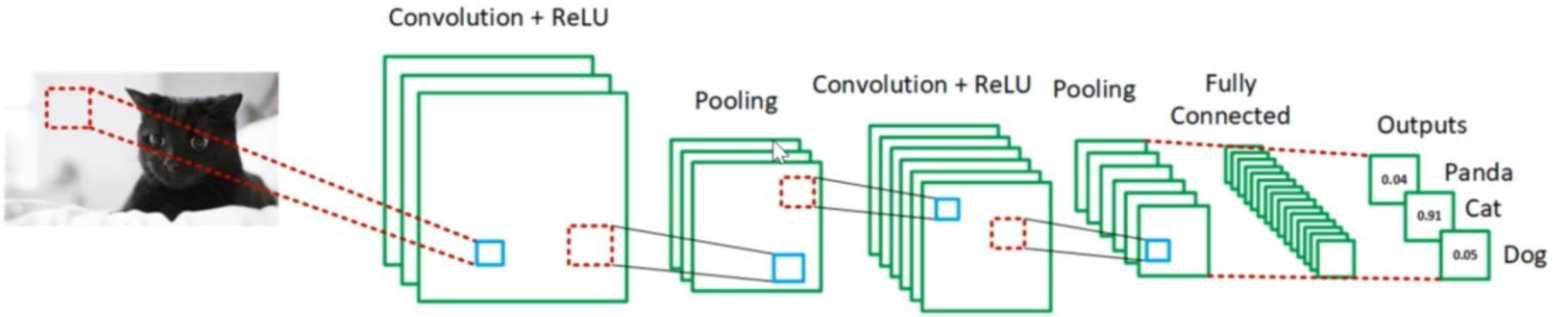
if ReLU activation:

$$Y = w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

$$Var(w_i) = \frac{2}{fan\_in}$$

It draws samples from a truncated normal distribution centered on 0 with stddev = sqrt(2 / fan_in) where fan_in is the number of input units in the weight tensor.

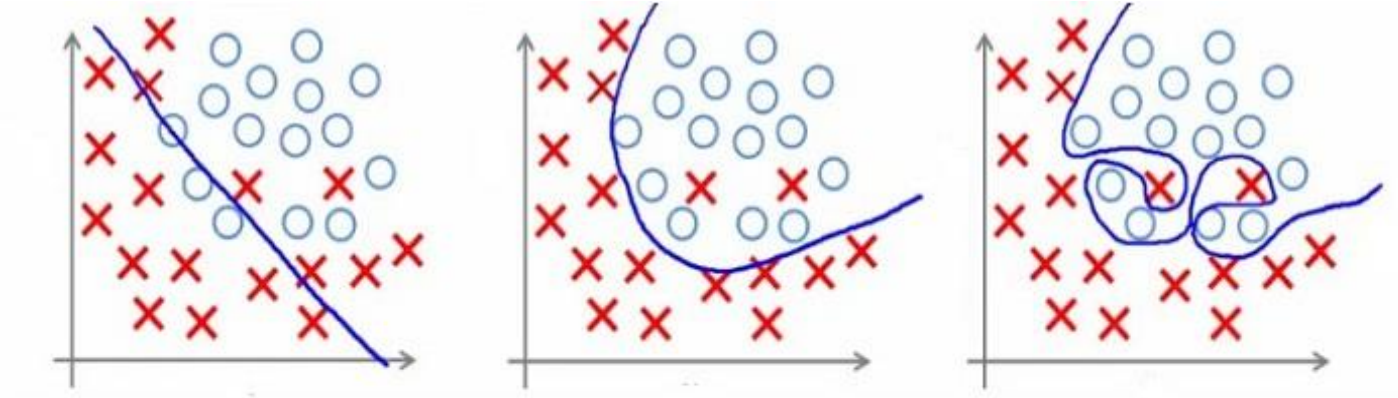
A SIMPLE CNN ARCHITECHTURE



3.
REGULARIZATION IN NEURAL NETWORKS

Introducing some techniques to avoid overfitting problem.

OVERFITTING



Under-fitting

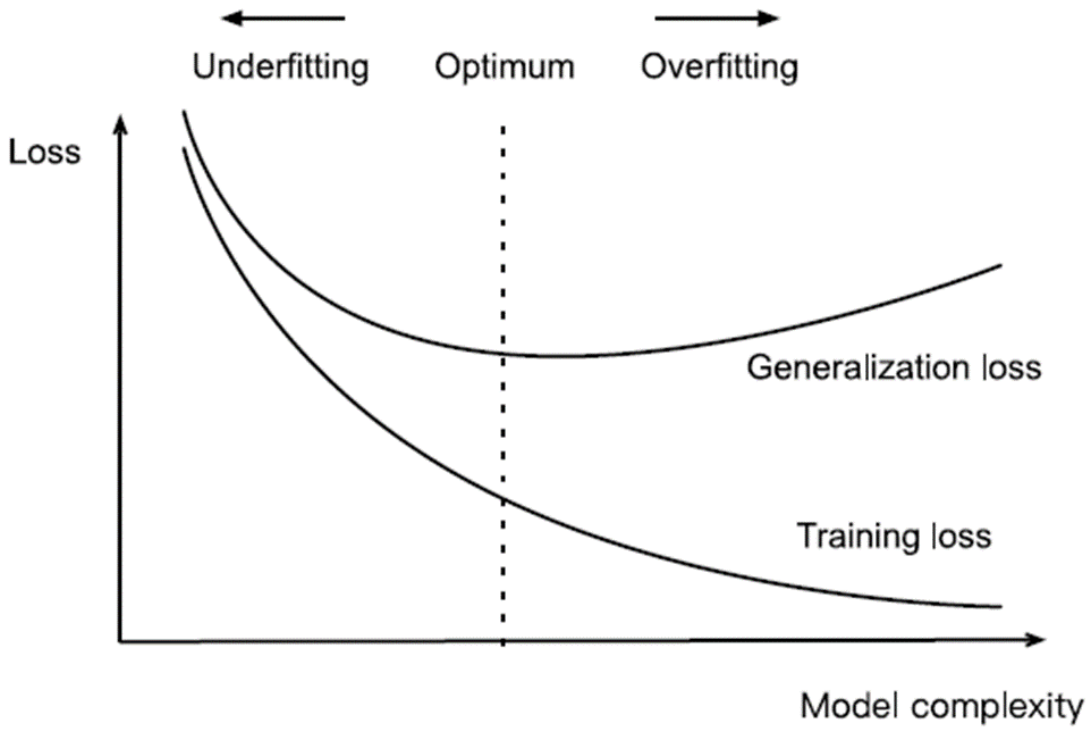
(too simple to explain the variance)

Appropriate-fitting

Over-fitting

(forcefitting -- too good to be true)

LOSS PLOT


Underfitting
Optimum
Overfitting
Loss
Generalization loss
Training loss
Model complexity

REGULARIZATION

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

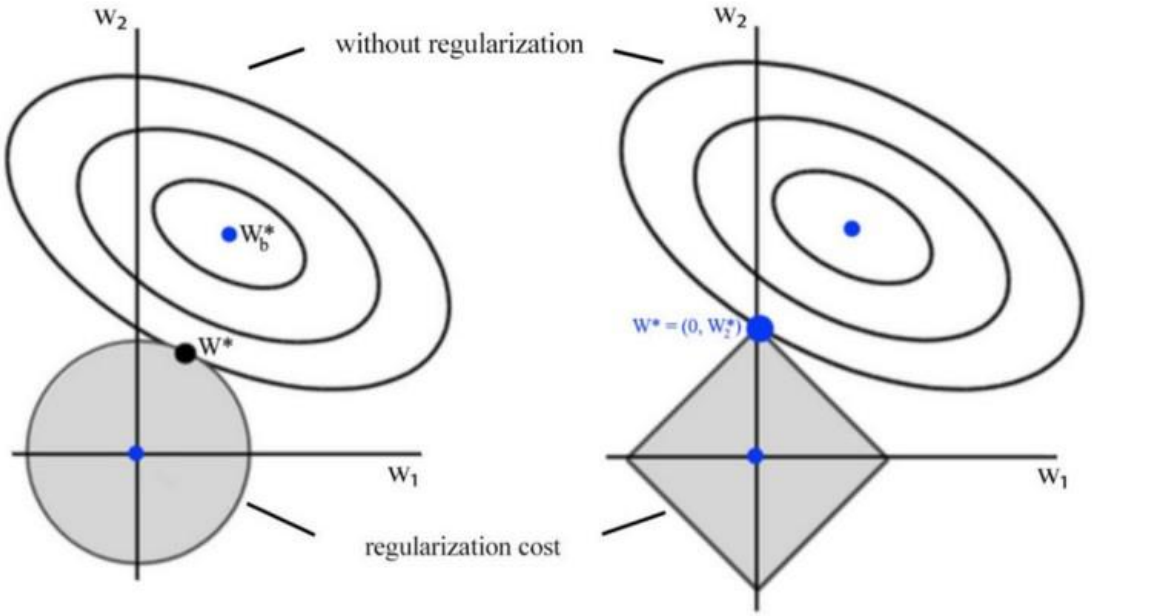
L2 Regularization

$$\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2$$

Loss function

Regularization Term

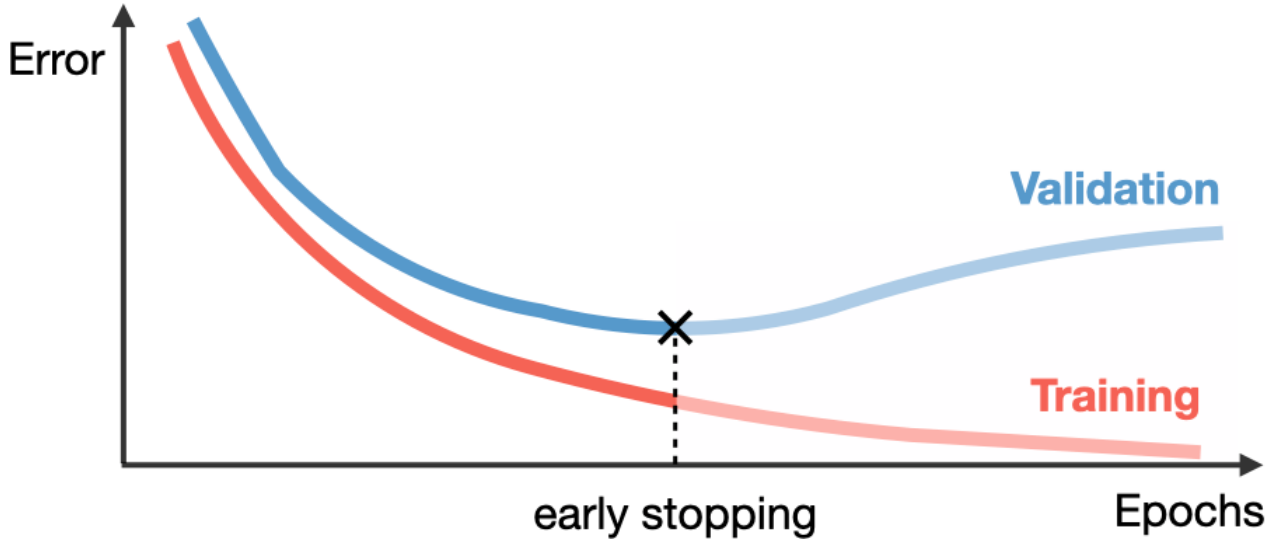
L1 VS. L2



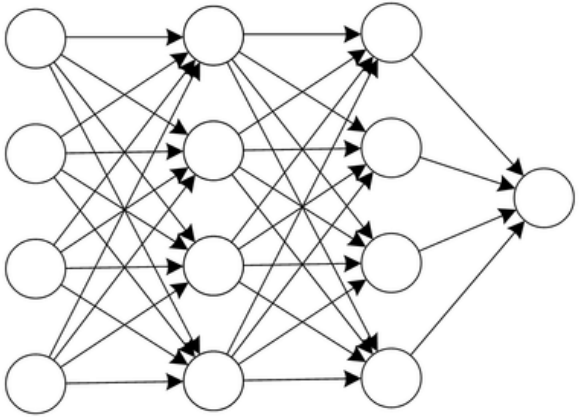
L2 regularization promotes small parameters

L1 regularization promotes sparse parameters

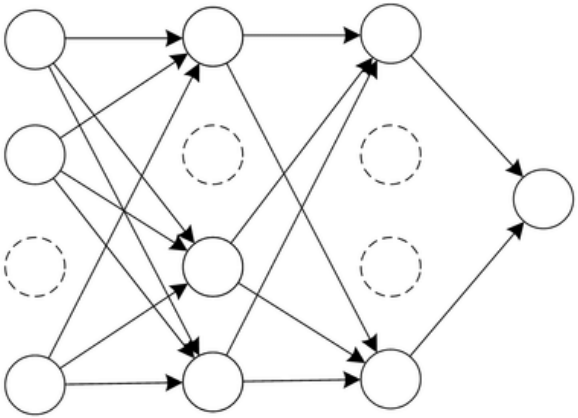
EARLY STOPPING



DROPOUT



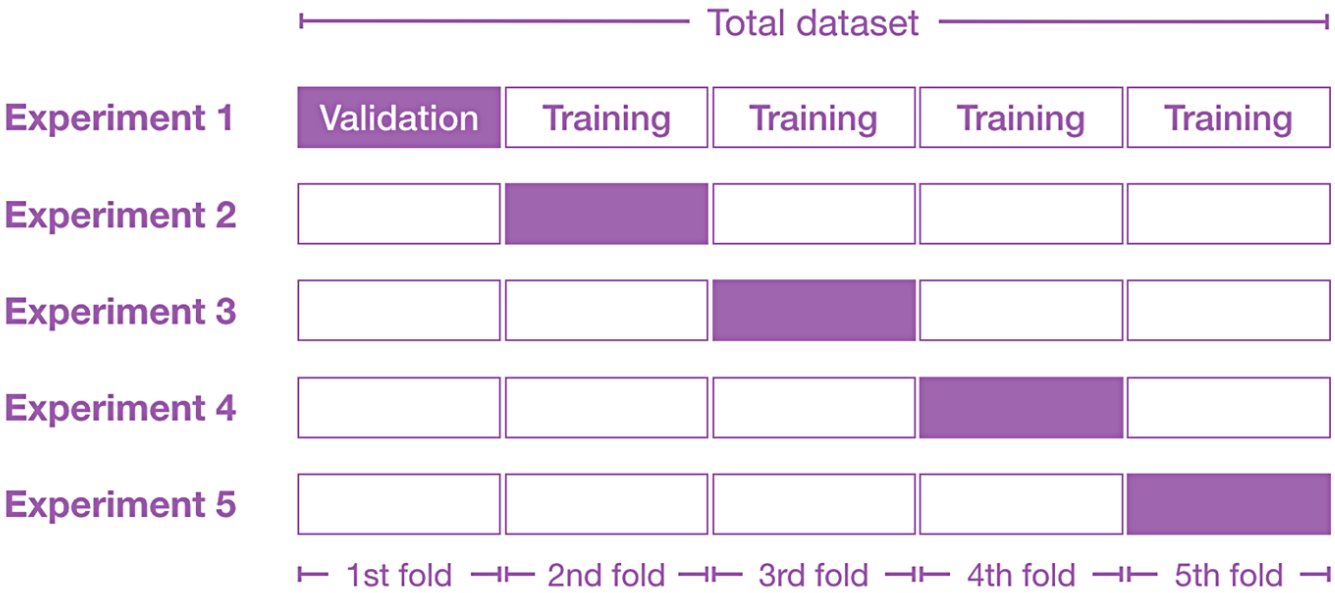
(a) Standard Neural Network



(b) Network after Dropout

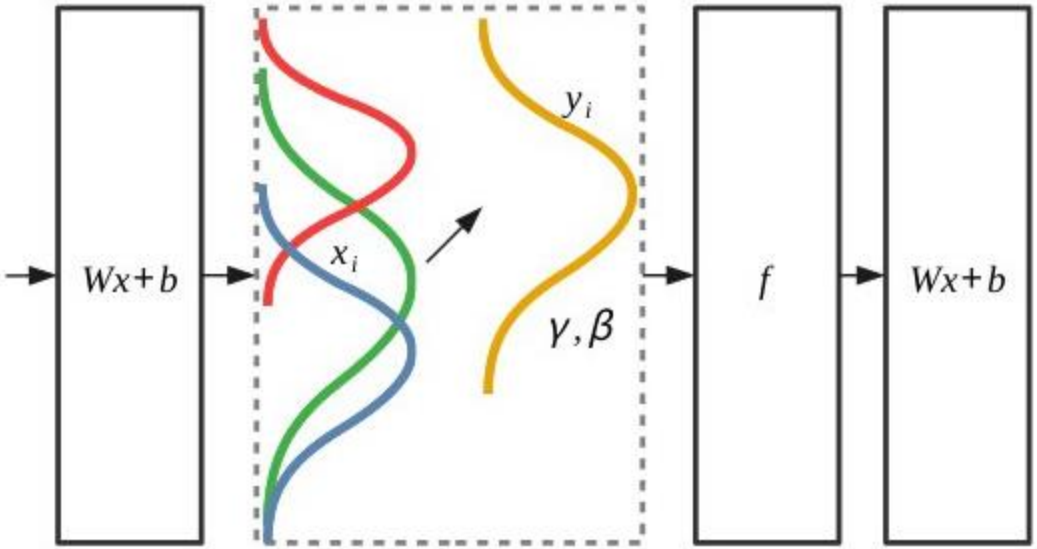
- During the training phase, randomly ignore some of the hidden layer neurons by the probability of p
- Reduce the complexity of the network

K-FOLD CROSS VALIDATION



- For example: k = 5

BATCH-NORMALIZATION



- Ensure the output statistics of a layer are fixed.

BATCH-NORMALIZATION

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

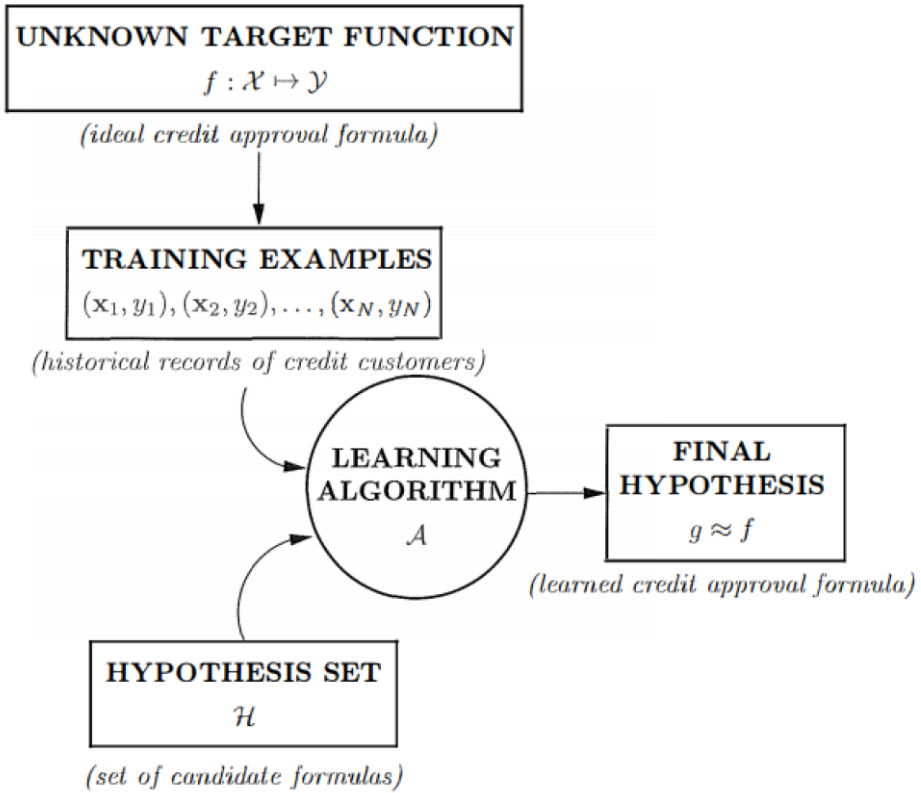
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Makes networks robust to bad initialization of the weights
- Usually inserted right before the activation layers
- Reduce covariance shift by normalizing and scaling the inputs
- The scale and shift parameters are trainable to avoid losing stability of the network

4.
CLASSIFICATION PROBLEM

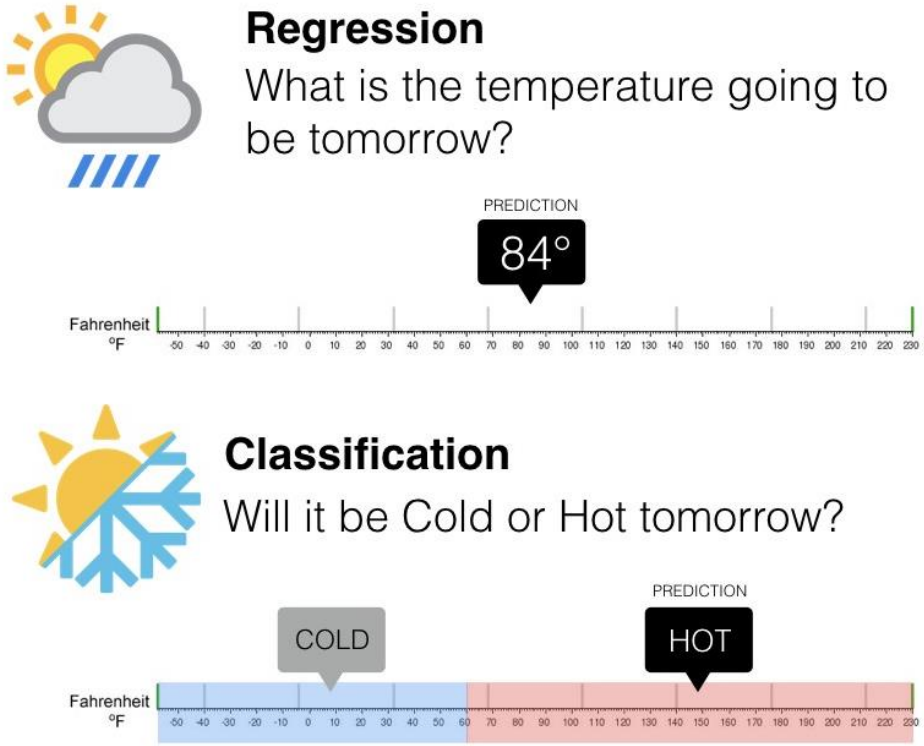
Studying the detail of the supervised learning.

SUPERVISED LEARNING



- Supervised learning, also known as supervised machine learning, is a subcategory of machine learning and artificial intelligence. It is defined by its use of labeled datasets to train algorithms that to classify data or predict outcomes accurately.
- Supervised learning can be separated into two types of problems: classification and regression

CLASSIFICATION VS. REGRESSION



- The main difference between them is that the output variable in regression is numerical (or continuous) while that for classification is categorical (or discrete).
- E.g. a regression model can be used to predict temperature for the next day, we can use a classification algorithm to determine whether it will be cold or hot according to the given temperature values.

CLASSIFICATION IN DETAIL

	Predicted class		
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

- True positive and true negatives are the observations that are correctly predicted and therefore shown in green. We want to minimize false positives and false negatives so they are shown in red color.

ACCURACY

- Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. One may think that, if we have high accuracy then our model is best. Yes, accuracy is a great measure but only when you have symmetric datasets where values of false positive and false negatives are almost same. Therefore, you have to look at other parameters to evaluate the performance of your model.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

PRECISION

- Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. The question that this metric answer is of all passengers that labeled as survived, how many actually survived? High precision relates to the low false positive rate

$$Precision = \frac{TP}{TP + FP}$$

RECALL

- Recall (Sensitivity) is the ratio of correctly predicted positive observations to the all observations in actual class - yes. The question recall answers is: Of all the passengers that truly survived, how many did we label?

$$Recall = \frac{TP}{TP + FN}$$

F1-SCORE

- F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall.

$$F1\ Score\ = \frac{2 \times (Recall\ \times\ Precision)}{(Recall\ +\ Precision)}$$

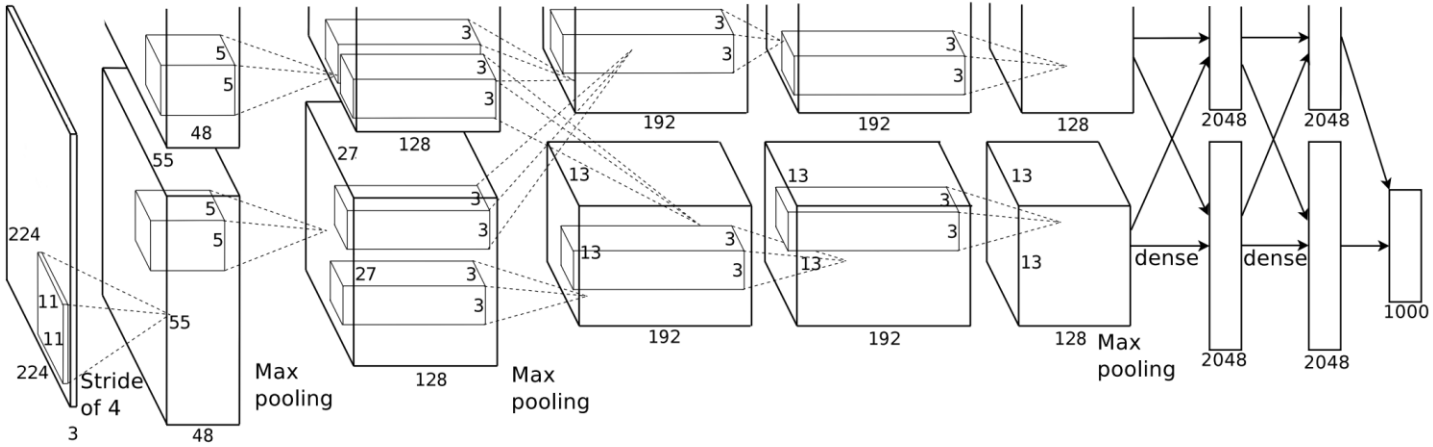
5. PARAMETER OPTIMIZATION

Studying the detail of the supervised learning.

6.
ADVANCED ARCHITECRURES

Introducing other CNN architectures and their history.

ALEXNET

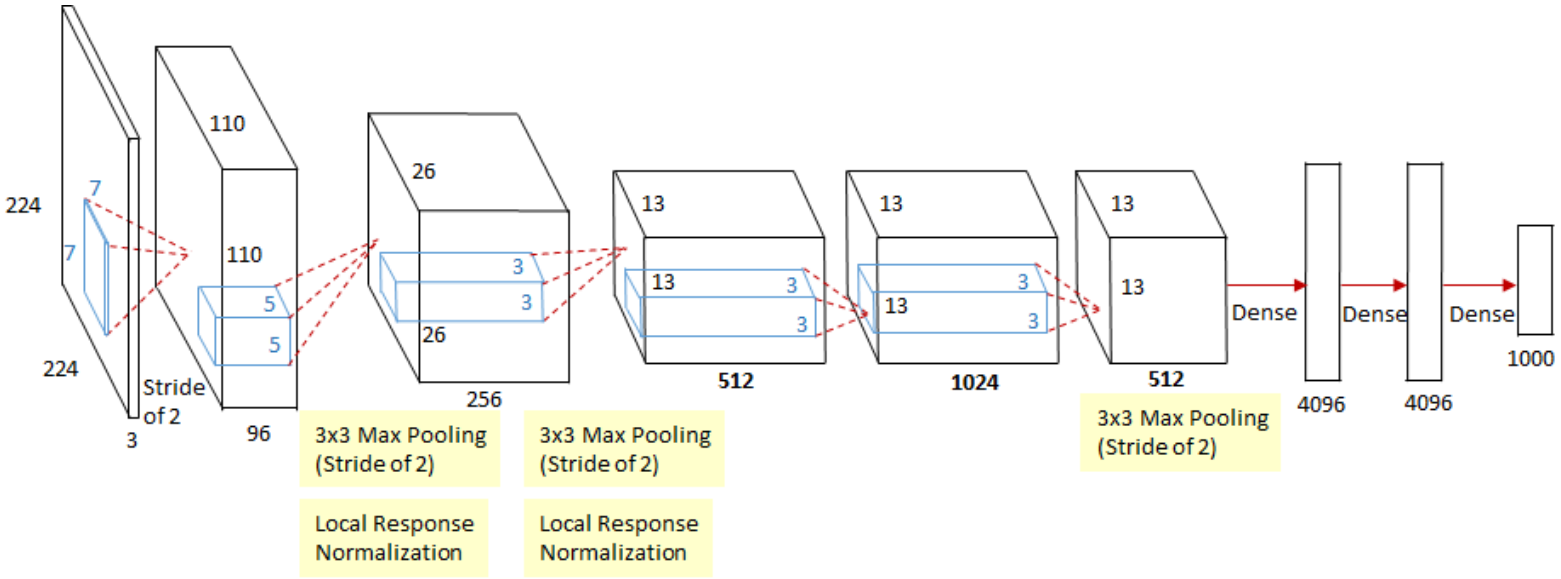


- The network was made up of 5 conv layers, max-pooling layers, dropout layers, and 3 fully connected layers.
- The network they designed was trained on ImageNet dataset and was used for classification with 1000 possible categories.

ALEXNET

- Trained the network on ImageNet data, which contained over 15 million annotated images from a total of over 22,000 categories.
- Used ReLU for the nonlinearity functions (Found to decrease training time as ReLUs are several times faster than the conventional tanh function).
- Used data augmentation techniques that consisted of image translations, horizontal reflections, and patch extractions.
- Implemented dropout layers in order to combat the problem of overfitting to the training data.
- Trained the model using batch stochastic gradient descent, with specific values for momentum and weight decay.
- Trained on two GTX 580 GPUs for five to six days.
- Achieved a top 5 test error rate of 15.4%. The next best entry achieved an error of 26.2%, which was an astounding improvement that pretty much shocked the computer vision community.

ZFNET



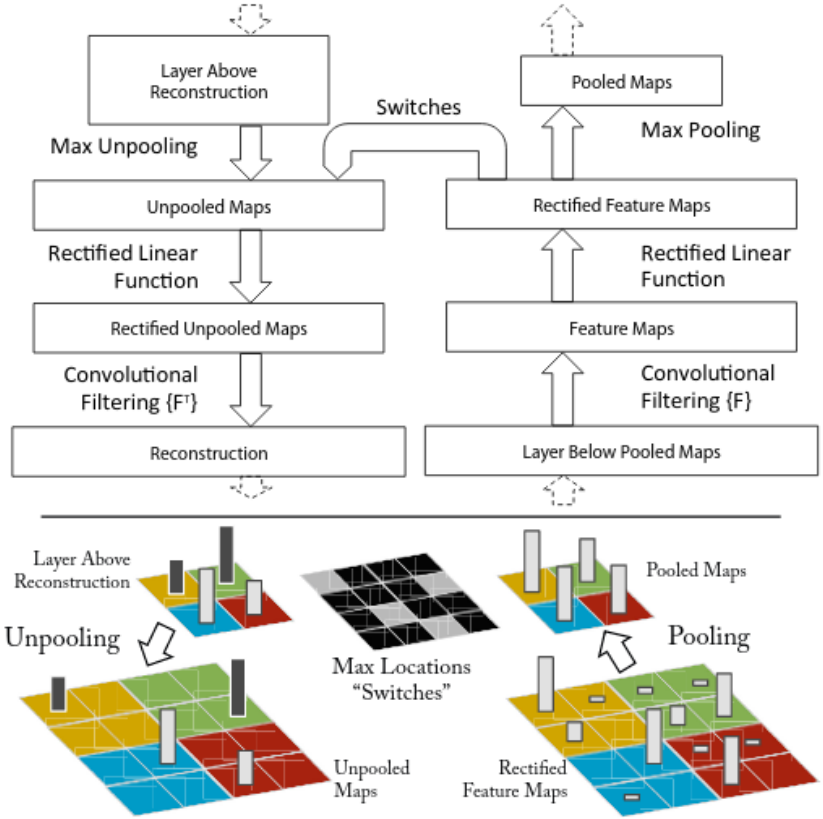
- This model achieved an 11.2% error rate

LOOP Academy

ZFNET

- Very similar architecture to AlexNet.
- AlexNet trained on 15 million images, while ZF Net trained on only 1.3 million images.
- Instead of using 11x11 sized filters in the first layer (same as AlexNet), ZF Net used filters of size 7x7 and a decreased stride value. The reasoning behind this modification is that a smaller filter size in the first conv layer helps retain a lot of original pixel information in the input volume. A filtering of size 11x11 proved to be skipping a lot of relevant information, especially as this is the first conv layer.
- As the network grows, we also see a rise in the number of filters used.
- Used ReLUs for their activation functions, cross-entropy loss for the error function, and trained using batch stochastic gradient descent.
- Trained on a GTX 580 GPU for twelve days.
- Developed a visualization technique named Deconvolutional Network, which helps to examine different feature activations and their relation to the input space.

DECONVNET



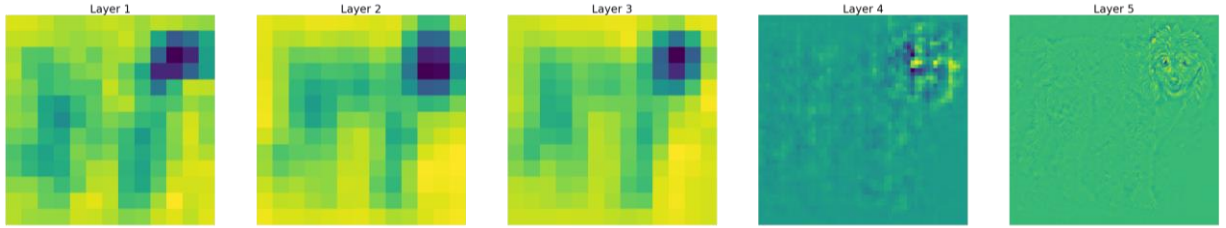
- By visualizing each layer, we can get more insight about what the model is learning and thus, make some adjustments to make it more optimize.
- That's how ZFNet was created, an AlexNet fine-tuned version based on visualization results.

DECONVNET

Class: Bedlington terrier, probability: 0.2510

Top-5 Accuracy:

- Bedlington terrier, probability: 0.2510
- miniature poodle, probability: 0.2062
- toy poodle, probability: 0.1112
- komondor, probability: 0.0714
- standard poodle, probability: 0.0656

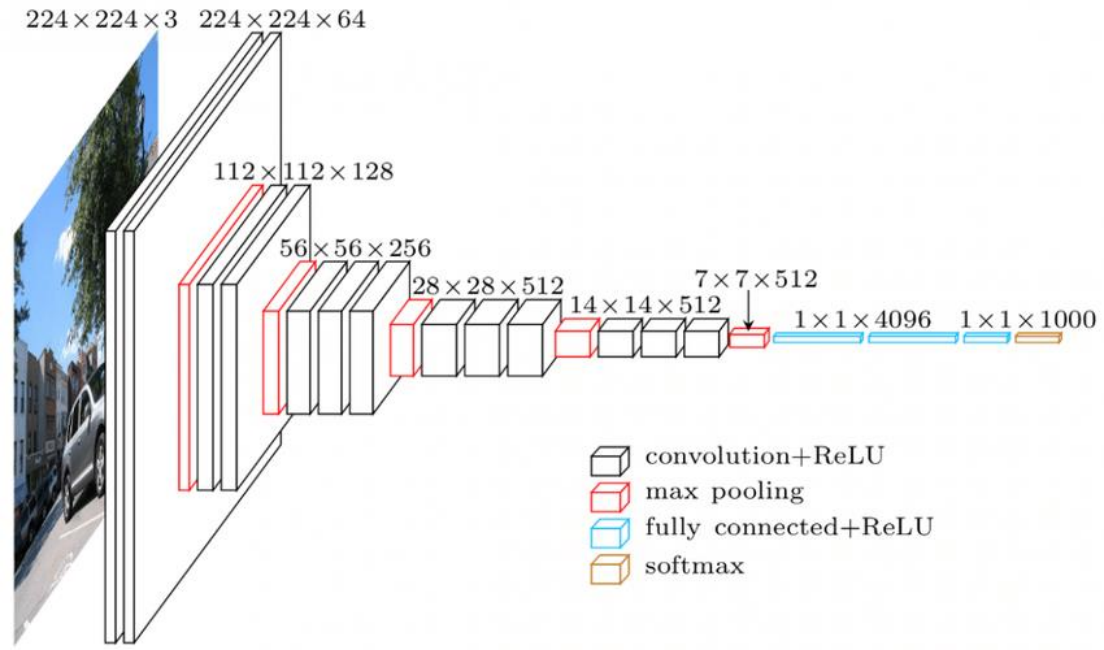


Check the following website for the implementation:
https://hackmd.io/@bouteille/ByaTE80BI

DECONVNET

- Filters at layer 1 are a mix of extremely high and low frequency information, with little coverage of the mid frequencies. Without the mid frequencies, there is a chain effect that deep features can only learn from extremely high and low frequency information.
- Layer 2 shows aliasing artifacts caused by the large stride 4 used in the 1st layer convolutions. Aliasing occurs when sampling frequency is too low.
- Layer 3 starts to learn some general patterns, such as mesh patterns, and text pattern.
- Layer 4 shows significant variation, and is more class-specific, such as dogs' faces and birds' legs.
- Layer 5 shows entire objects with significant pose variation, such as keyboards and dogs.

VGG



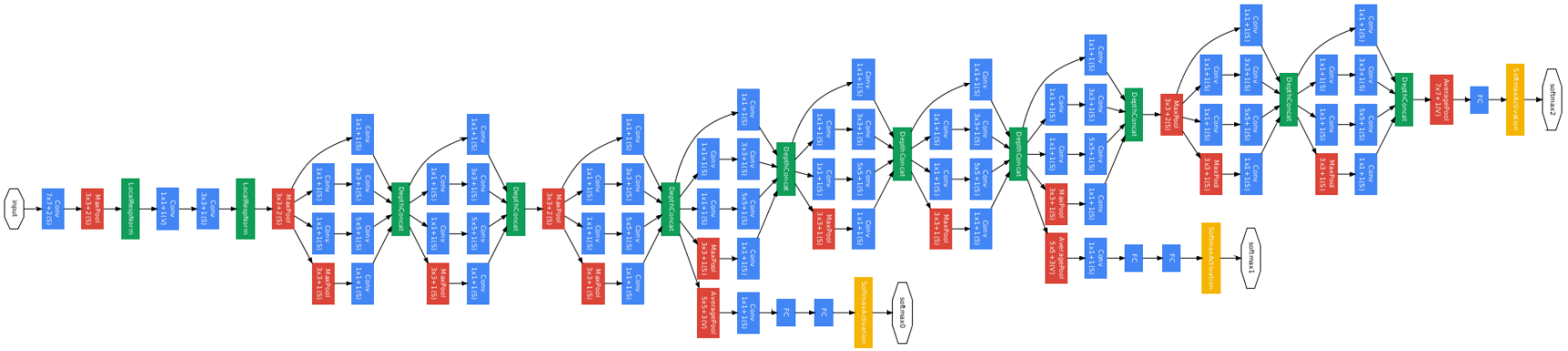
- VGG addresses another very important aspect of CNNs: depth !
- That's what a model created in 2014 best utilized with its 7.3% error rate.

VGG

VGG16	VGG19
conv 3x3, 64	conv 3x3, 64
conv 3x3, 64	conv 3x3, 64
max pool	
conv 3x3, 128	conv 3x3, 128
conv 3x3, 128	conv 3x3, 128
max pool	
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
conv 3x3, 256	conv 3x3, 256
	conv 3x3, 256
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
conv 3x3, 512	conv 3x3, 512
	conv 3x3, 512
max pool	
fc-4096	
fc-4096	
fc-1000	
softmax	

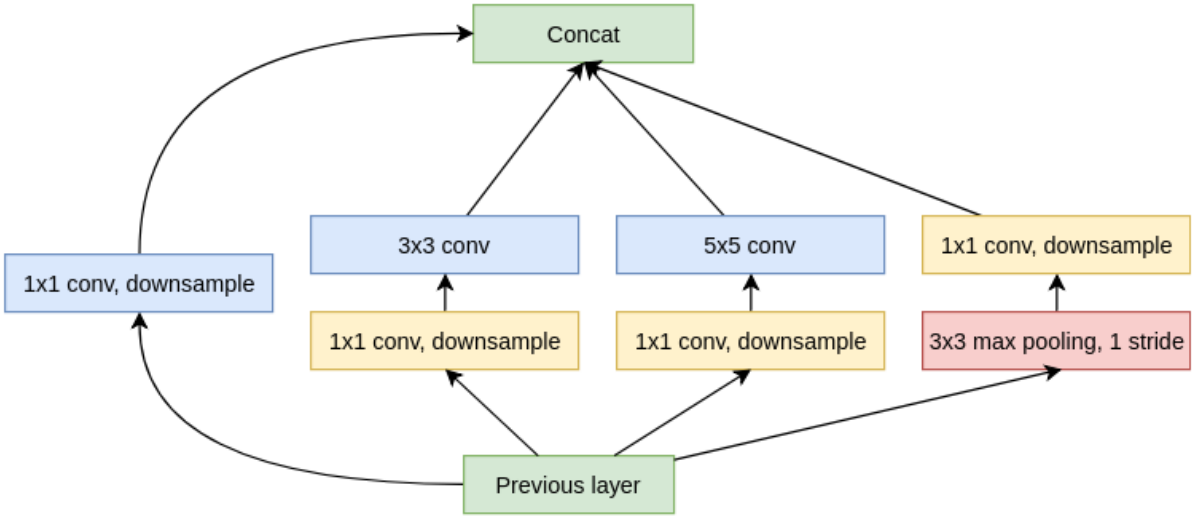
- The use of only 3x3 sized filters is quite different from AlexNet's 11x11 filters in the first layer and ZF Net's 7x7 filters. The authors' reasoning is that the combination of two 3x3 conv layers has an effective receptive field of 5x5. This in turn simulates a larger filter while keeping the benefits of smaller filter sizes. One of the benefits is a decrease in the number of parameters. Also, with two conv layers, we're able to use two ReLU layers instead of one.
- 3 conv layers back to back have an effective receptive field of 7x7.
- As the spatial size of the input volumes at each layer decrease (result of the conv and pool layers), the depth of the volumes increase due to the increased number of filters as you go down the network.
- Interesting to notice that the number of filters doubles after each maxpool layer. This reinforces the idea of shrinking spatial dimensions, but growing depth.
- Worked well on both image classification and localization tasks. The authors used a form of localization as regression (see page 10 of the paper for all details).
- Used scale jittering as one data augmentation technique during training.

GOOGLENET (INCEPTION V1)



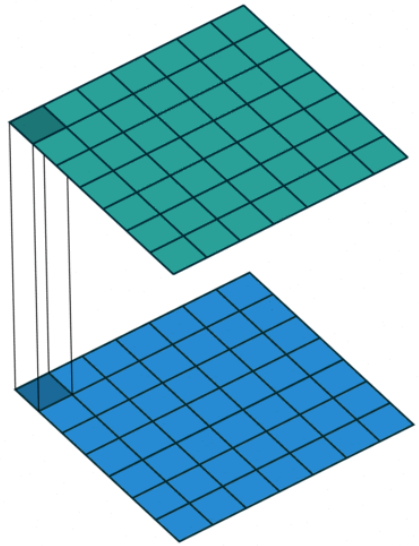
- GoogleNet is a 22 layer CNN and was the winner of ILSVRC 2014 with a top 5 error rate of 6.7%.
- Used 9 Inception modules in the whole architecture, with over 100 layers in total! Now that is deep…
- No use of fully connected layers! They use an average pool instead, to go from a 7x7x1024 volume to a 1x1x1024 volume. This saves a huge number of parameters.
- Uses 12x fewer parameters than AlexNet.

INCEPTION V1 BLOCK



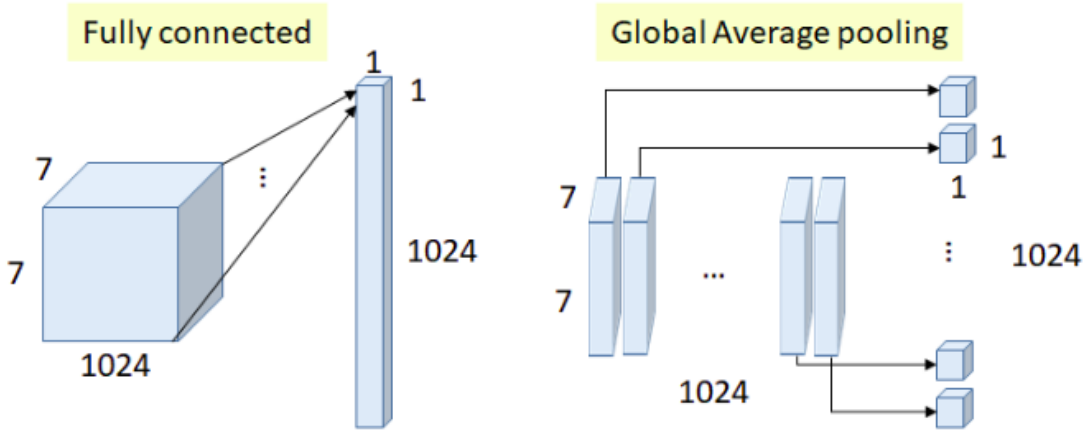
- When image's coming in, different sizes of convolutions as well as max pooling are tried. Then different kinds of features are extracted.
- After that, all feature maps at different paths are concatenated together as the input of the next module.
- 1×1 convolution is inserted into the inception module for dimension reduction!

1x1 CONOLUTIONAL LAYER (NETWORK IN NETWORK)



- We use 1x1 filter to down sample the depth or number of feature maps.
- Without the 1×1 convolution as above, we can imagine how large the number of operation is!
- This simple 1×1 filter provides a way to usefully summarize the input feature maps
- A convolutional layer with a 1×1 filter can, therefore, be used at any point in a convolutional neural network to control the number of feature maps.
- It is a linear weighting or projection of the input. Further, a nonlinearity is used as with other convolutional layers, allowing the projection to perform non-trivial computation on the input feature maps.

GLOBAL AVERAGE POOLING

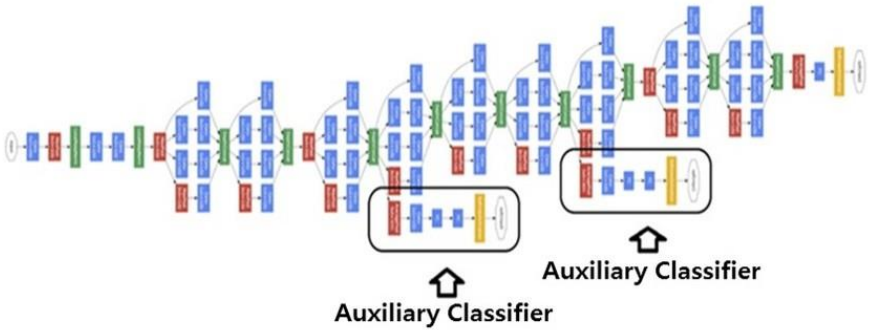


Fully Connected Layer VS Global Average Pooling

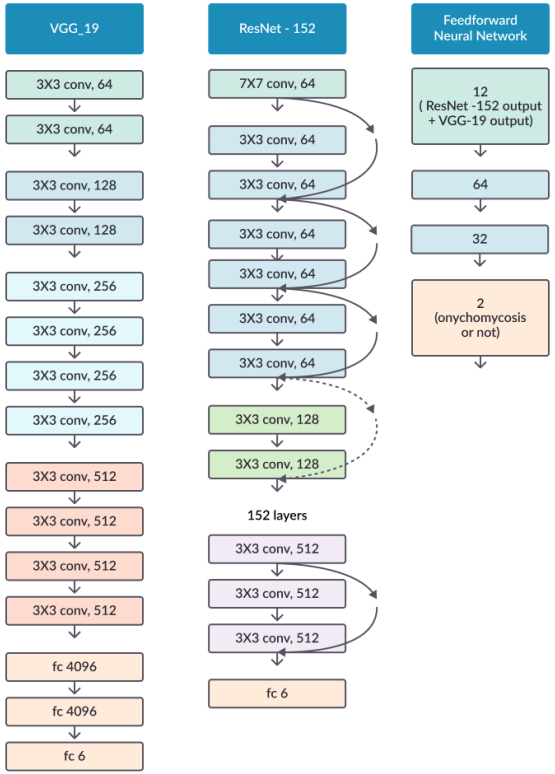
- Previously, **fully connected (FC) layers** are used at the end of network, such as in AlexNet. All inputs are connected to each output. (**Number of weights (connections) = 7×7×1024×1024 = 51.3M)**
- **In GoogleNet, global average pooling is used nearly at the end of network by averaging each feature map from 7×7 to 1×1, as in the figure above. (Number of weights = 0)**

AUXILIARY CLASSIFIERS

- There exists two additional classification output (with the same groundtruth labels) at various intermediate layers. During training, the total value of the loss is a weighted sum of the auxiliary losses and the real loss.
- It can be used for combating gradient vanishing problem, also providing regularization.



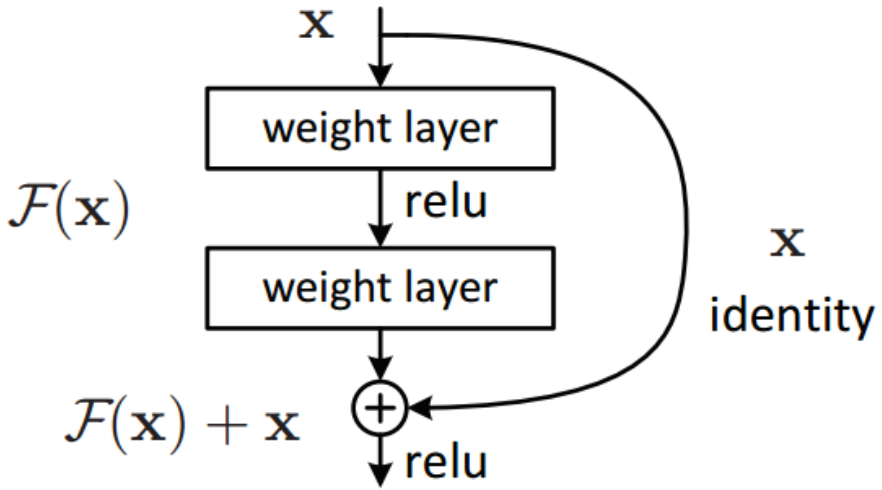
RESNET



Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper. While AlexNet had only 5 convolutional layers, the VGG network and GoogleNet (Inception v1) had 19 and 22 layers respectively.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

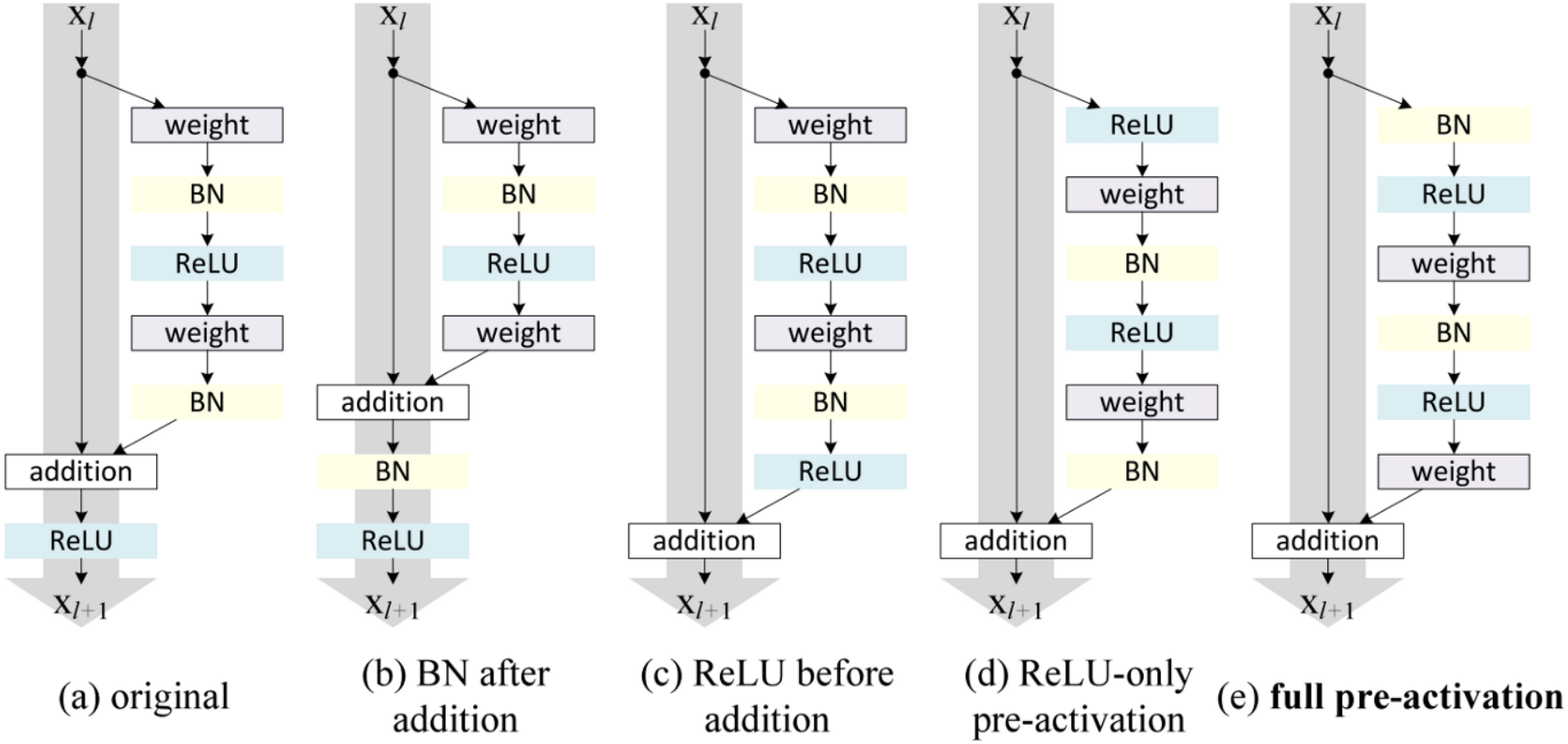
RESIDUAL BLOCK



H(x) = F(x) + x is the desired mapping

It is easier for F(x) to just learn the residuals H(x) -x

VARIANTS OF RESIDUAL BLOKS



THE FAMILY OF THE MOST POPULAR RESIDUAL NETWORKS

output size	18-layer	34-layer	50-layer	101-layer	152-layer
112x112	7x7 conv, stride 2				
56x56	3x3 max pool, stride 2				
	3x3, 64 3x3, 64 x2	3x3, 64 3x3, 64 x3	1x1, 64 3x3, 64 1x1, 256 x3	1x1, 64 3x3, 64 1x1, 256 x3	1x1, 64 3x3, 64 1x1, 256 x3
28x28	3x3, 128 3x3, 128 x2	3x3, 128 3x3, 128 x4	1x1, 128 3x3, 128 1x1, 512 x4	1x1, 128 3x3, 128 1x1, 512 x4	1x1, 128 3x3, 128 1x1, 512 x8
14x14	3x3, 256 3x3, 256 x2	3x3, 256 3x3, 256 x6	1x1, 256 3x3, 256 1x1, 1024 x6	1x1, 256 3x3, 256 1x1, 1024 x23	1x1, 256 3x3, 256 1x1, 1024 x36
7x7	3x3, 512 3x3, 512 x2	3x3, 512 3x3, 512 x3	1x1, 512 3x3, 512 1x1, 2048 x3	1x1, 512 3x3, 512 1x1, 2048 x3	1x1, 512 3x3, 512 1x1, 2048 x3
1x1	average pool, 1000-d fc, softmax				

LOOP Academy

BOTTLENECK RESIDUAL BLOCK

In the paper, He et al. use bottleneck architecture for each the residual block. It means that the residual block consists of 3 layers in this order: 1x1 convolution - 3x3 convolution - 1x1 convolution. The first and the last convolution is the bottleneck. It mostly just for practical consideration, as the first 1x1 convolution is being used to reduce the dimensionality, and the last 1x1 convolution is to restore it. So, the same network is now become 50 layers.

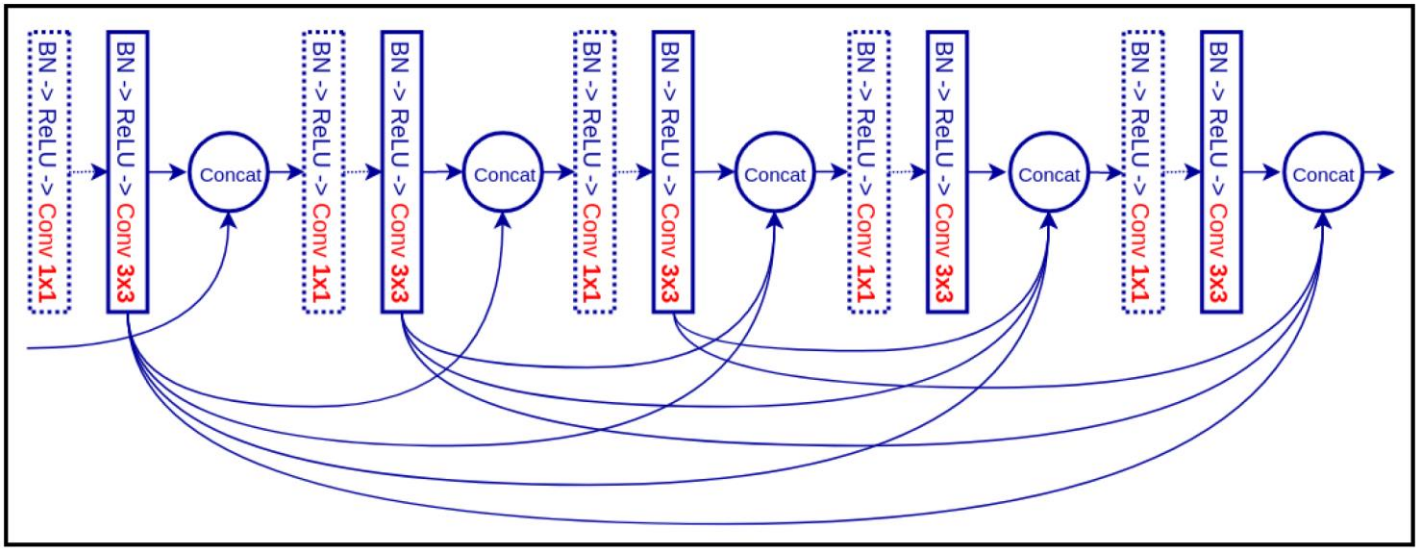
LOOP Academy

POINTS OF RESNET

- **“Ultra-deep” – Yann LeCun.**
- **More layers is better** but because of the **vanishing gradient problem**, model weights of the first layers can not be updated correctly through the backpropagation of the error gradient (**the chain rule multiplies error gradient values lower than one and then, when the gradient error comes to the first layers, its value goes to zero**). But the ResNet preserves the gradient because of using the identity matrix “*what if we were to backpropagate through the identity function? Then the gradient would simply be multiplied by 1 and nothing would happen to it!*”.
- The identity matrix transmits forward the input data that avoids the loose of information (the data vanishing problem).
- Trained on an 8 GPU machine for two to three weeks.
- 3.6% error rate on ImageNet.

DENSENET

DenseNet stands for Densely-Connected Convolutional Networks. It tries to alleviate the vanishing gradient problem and improve feature propagation, while reducing the number of network parameters. We've already seen how ResNets introduce residual blocks with skip connections to solve this. DenseNets take some inspiration from this idea and introduce dense blocks. A dense block consists of sequential convolutional layers, where any layer has a direct connection to all subsequent layers:



DENSENET ARCHITECTURE FOR IMAGENET

Layers	Output Size	DenseNet-121($k = 32$)	DenseNet-169($k = 32$)	DenseNet-201($k = 32$)	DenseNet-161($k = 48$)
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 36$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

DENSENET

7. TRANSFER LEARNING

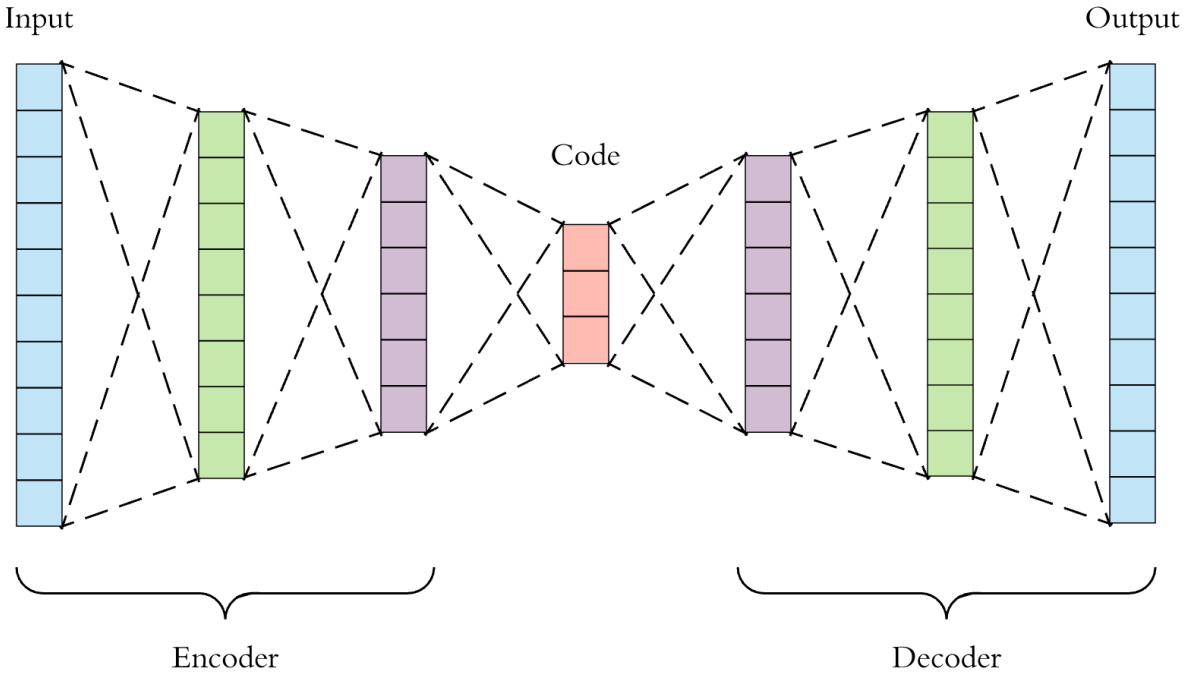
Advanced CNNs for Object Detection.

8. AUTOENCODERS

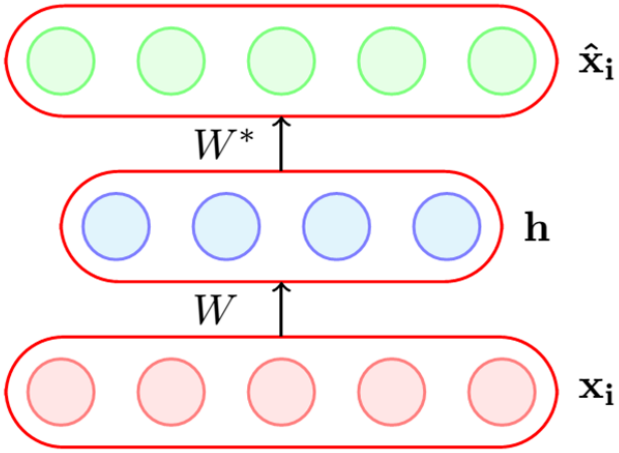
Using autoencoders for unsupervised learning in neural networks.

COMPONENTS OF AUTOENCODERS

An autoencoder is an unsupervised learning approach that applies backpropagation, setting the target values to be equal to the inputs.



THEORY OF AUTOENCODERS



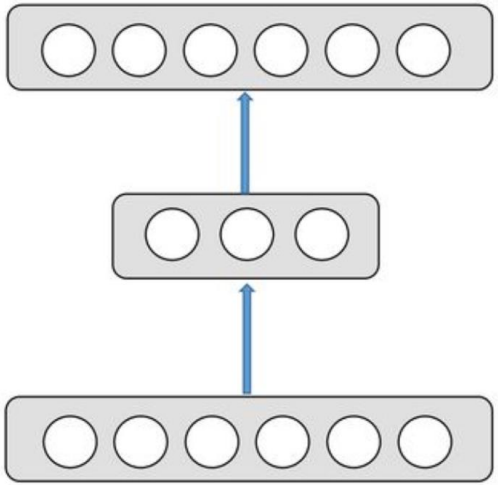
$$\mathbf{h} = g(W\mathbf{x_i} + \mathbf{b})$$
$$\hat{\mathbf{x}}_\mathbf{i} = f(W^*\mathbf{h} + \mathbf{c})$$

- An autoencoder is a special type of feed-forward neural network which does the following:
  - Encodes its input x_i into a hidden representation h.
  - Decodes the input again from this hidden representation.
  - The model is trained to minimize a certain loss function which will ensure that $\hat{x}_i$ is close to x_i.

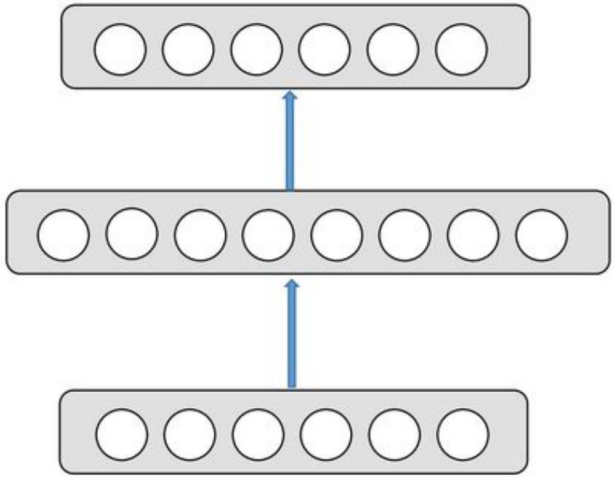
$$\min_{W,W^*,c,b} \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(\hat{x}_{ij} - x_{ij}\right)^2$$

UNDERCOMPLETE VS. OVERCOMPLETE

We distinguish between two types of autoencoder structures:

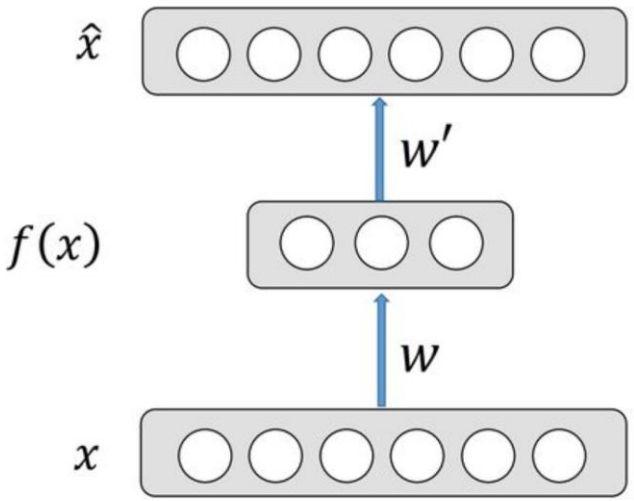


Undercomplete



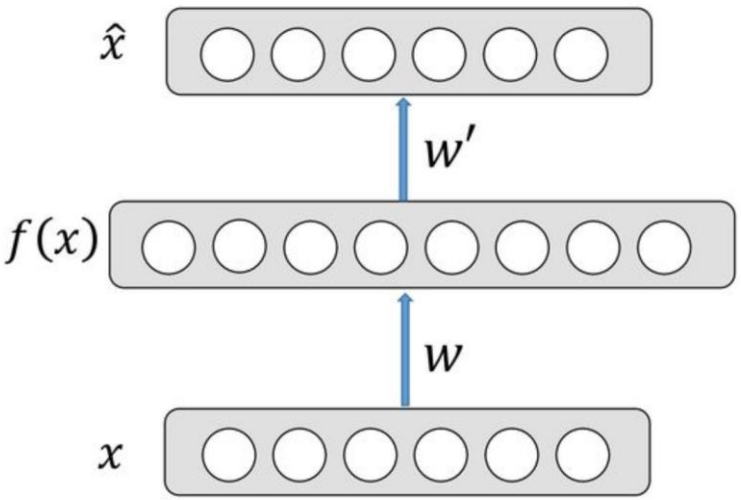
Overcomplete

UNDERCOMPLETE AUTOENCODERS



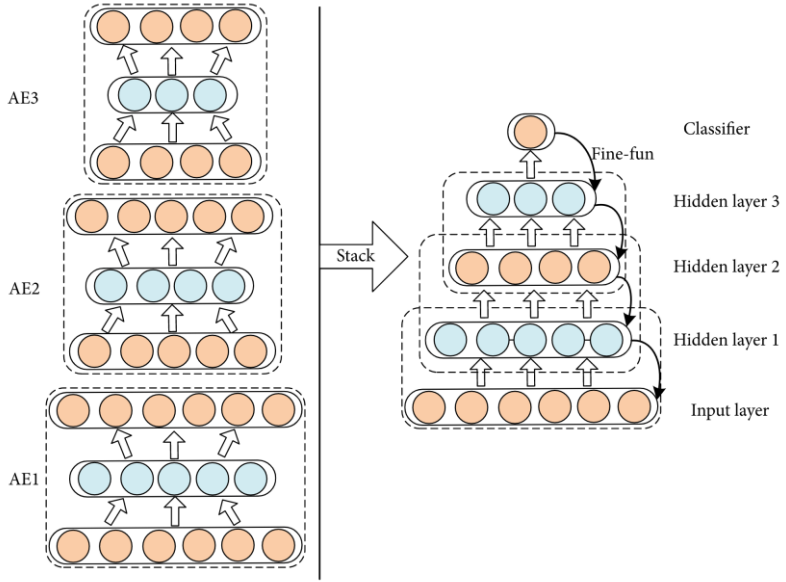
- Hidden layer is **undercomplete** if smaller than the input layer
  - Compresses the input.
  - Compresses well only for the training distribution.
- Hidden nodes will be
  - Good features for the training distribution.
  - Bad for other types of input.

OVERCOMPLETE AUTOENCODERS



- Hidden layer is **overcomplete** if greater that the input layer
  - No compression in hidden layer.
  - Each hidden unit could copy a different input component.
- No guarantee that hidden units will extract meaningful structure.
- Adding dimensions is good for training a linear classifiers (XOR case example).
- A higher dimension code helps model a more complex distribution

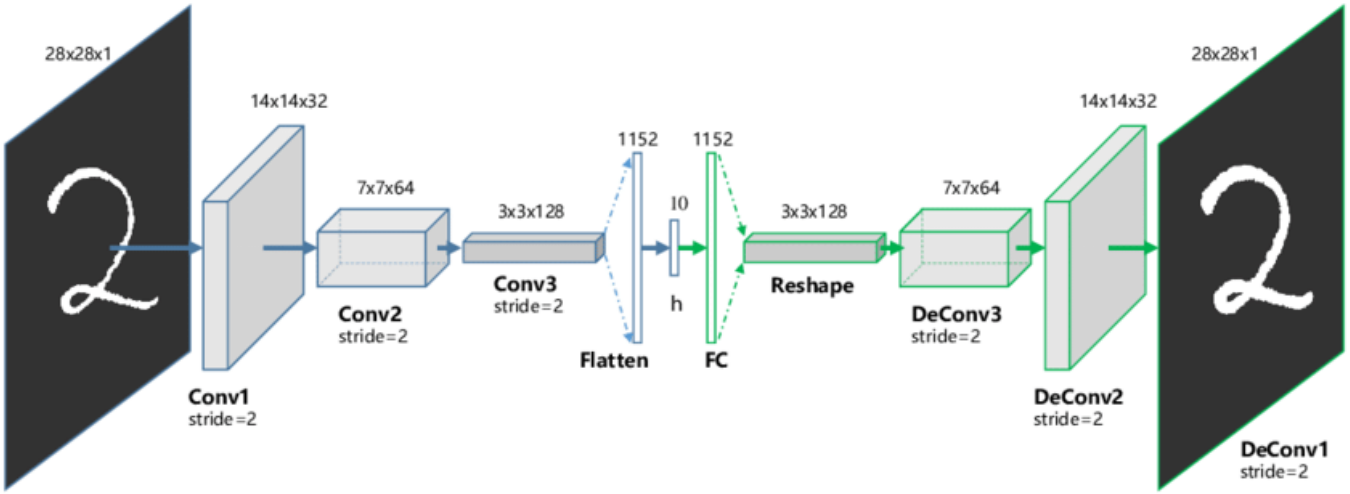
STACKED AUTOENCODER



- It can be difficult to obtain **convergence** for deep auto encoders, especially since network attempts to train all layers at once without imposing symmetry conditions on the network topology (arbitrary configuration of layers is allowed).
- So train a deep autoencoder layer by layer

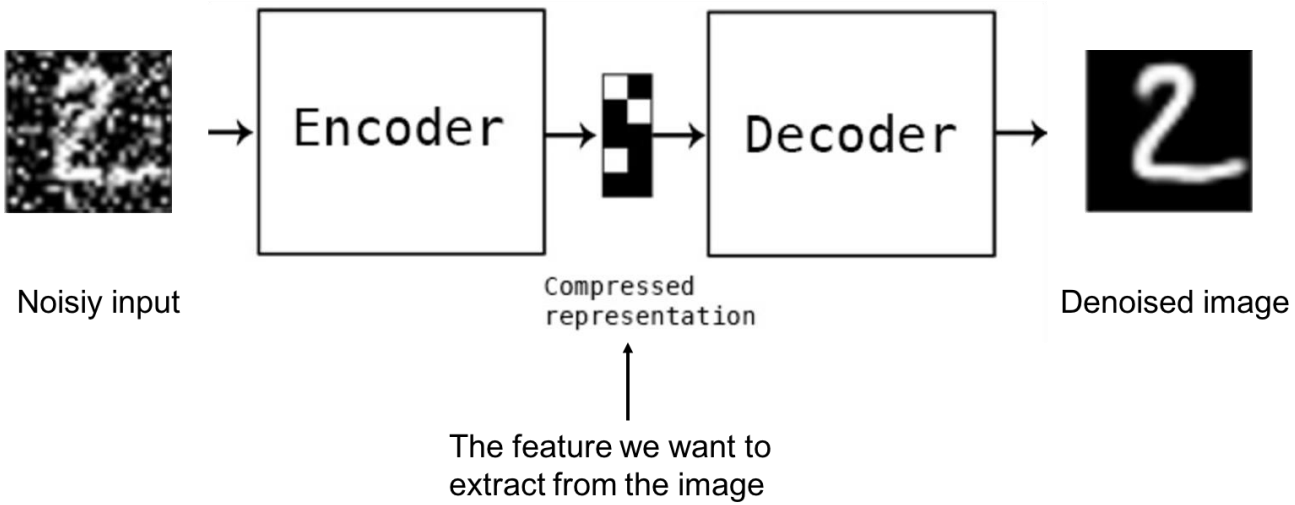
CONVOLUTIONAL AUTOENCODERS

Convolutional autoencoders use the convolution operator to learn to encode the inputs in a set of simple signals and then try to reconstruct the input from them.



NOISE REMOVAL

The input seen by the autoencoder is not the raw input but a stochastically corrupted version. A denoising autoencoder is thus trained to reconstruct the original input from the noisy version.

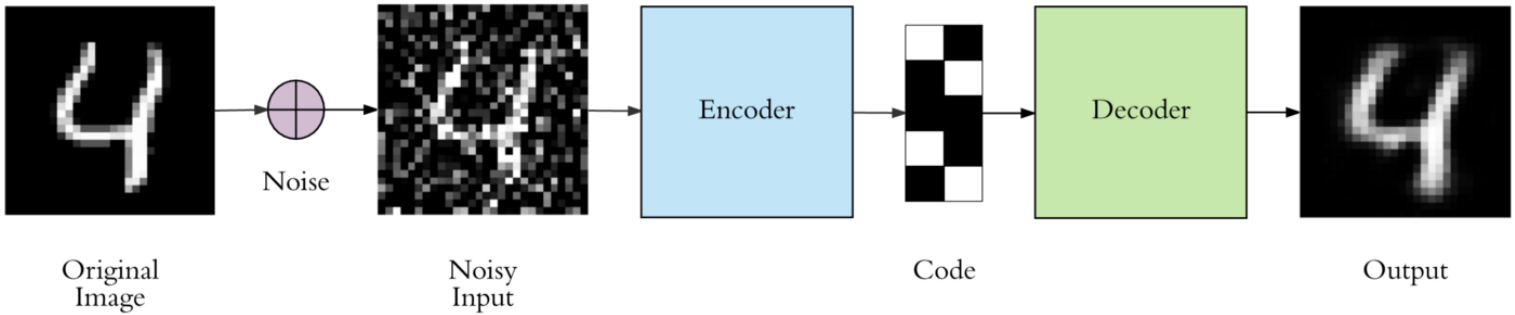


Convolutional Autoencoders for Image Noise Reduction | by Dr. Dataman | Towards Data Science

DENOISING AUTOENCODER (I)

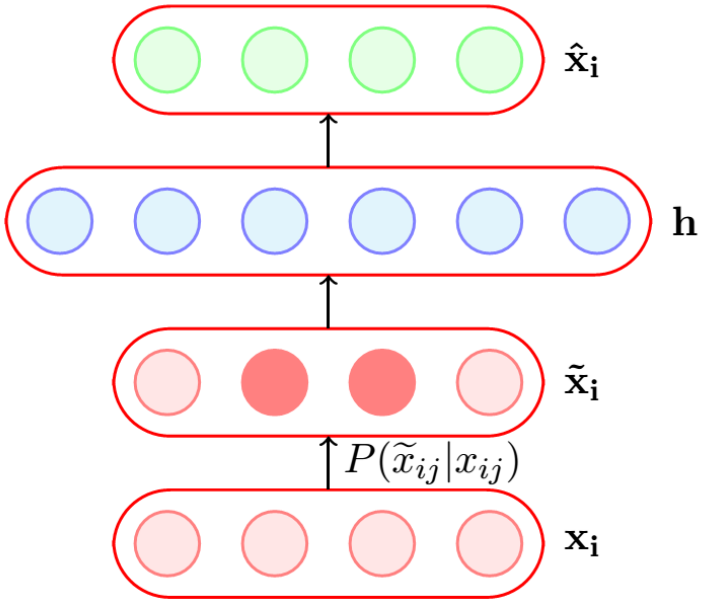
- In Vincent et al. (2010), "*a **good representation** is one that can be obtained **robustly** from a **corrupted input** and that will be useful for **recovering** the corresponding **clean input**.*"
  - The higher level representations are relatively stable and robust to input corruption.

DENOISING AUTOENCODER (II)



Denoising autoencoder adds some noise to the original input image creating a corrupted version of the input image. Adding noise helps to make the autoencoder robust to noise in the input image. The corrupt images are given to the autoencoder as input and then the autoencoder needs to reconstruct the original undistorted image. It forces the autoencoder to learn features from the input data instead of memorizing it.

DENOISING AUTOENCODER (III)



- A denoising encoder simply corrupts the input data using a probabilistic process ($P(\widetilde{x}_{ij}|x_{ij})$) before feeding it to the network
- A simple $P(\widetilde{x}_{ij}|x_{ij})$ used in practice is the following

$$P(\widetilde{x}_{ij} = 0|x_{ij}) = q$$
$$P(\widetilde{x}_{ij} = x_{ij}|x_{ij}) = 1 - q$$

- In other words, with probability q the input is flipped to 0 and with probability $(1 - q)$ it is retained as it is

Another way of corrupting the inputs is to add a Gaussian noise to the input

$$\widetilde{x}_{ij} = x_{ij} + \mathcal{N}(0, 1)$$

DENOISING AUTOENCODER (IV)

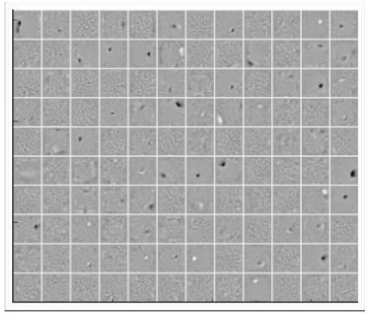


Figure: Vanilla AE (No noise)

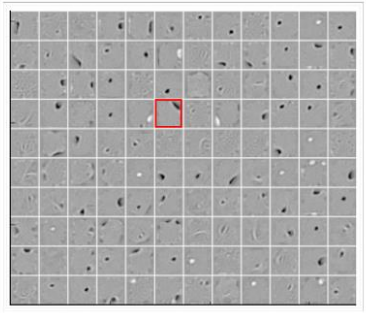


Figure: 25% Denoising AE (q=0.25)

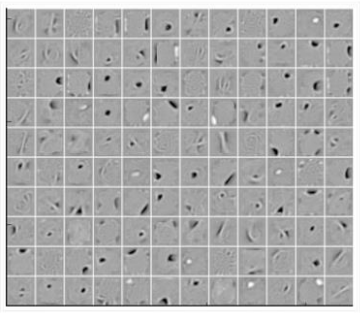


Figure: 50% Denoising AE (q=0.5)

- The vanilla AE does not learn many meaningful patterns
- The hidden neurons of the denoising AEs seem to act like pen-stroke detectors (for example, in the highlighted neuron the black region is a stroke that you would expect in a '0' or a '2' or a '3' or a '8' or a '9')
- As the noise increases the filters become more wide because the neuron has to rely on more adjacent pixels to feel confident about a stroke

DENOISING AUTOENCODER (V)

The loss function of Denoising autoencoder:

$$\min_{\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}_1,\mathbf{b}_2} \sum_{\ell} \|\mathbf{x}^{(\ell)} - \hat{\mathbf{x}}^{(\ell)}\|_2^2 + \lambda\left(\|\mathbf{W}_1\|_F^2 + \|\mathbf{W}_2\|_F^2\right)$$

where

$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}_1\tilde{\mathbf{x}}^{(\ell)} + \mathbf{b}_1)$$
$$\hat{\mathbf{x}}^{(\ell)} = \sigma(\mathbf{W}_2\mathbf{h}^{(\ell)} + \mathbf{b}_2)$$

Like deep Autoencoder, we can stack multiple denoising autoencoders layer-wisely to form a Stacked Denoising Autoencoder.

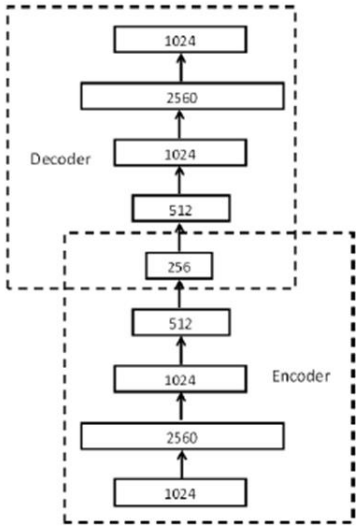


IMAGE COLORING

Autoencoders are used for converting any black and white picture into a colored image. Depending on what is in the picture, it is possible to tell what the color should be.

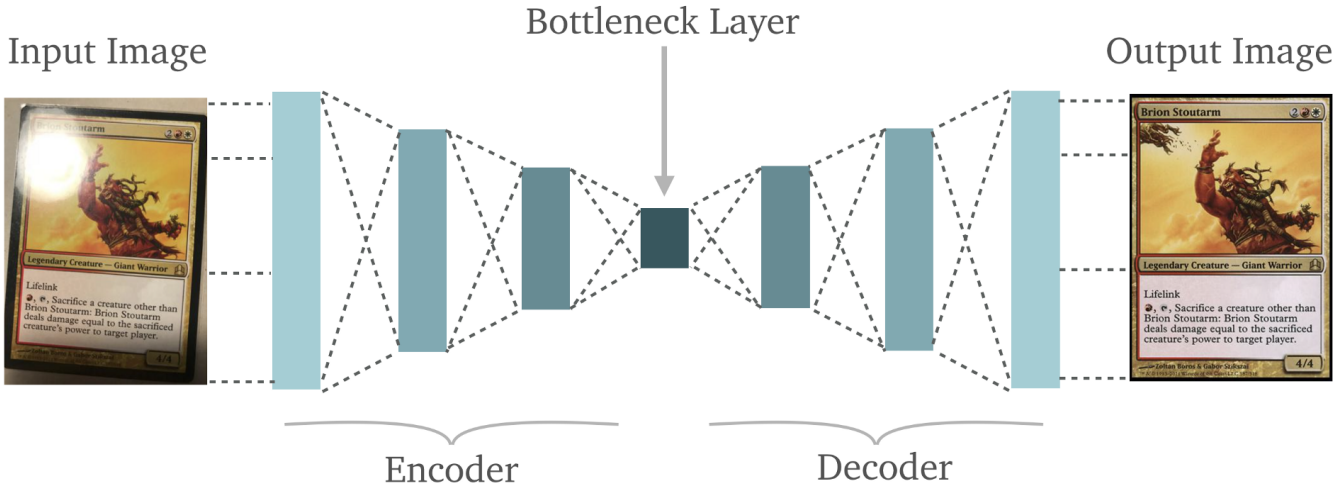
Auto Colorization of Black and White Images using Machine Learning “Auto-encoders” technique | by Mahmoud El-Jiddawi | Becoming Human: Artificial Intelligence Magazine

WATERMARK REMOVAL

It is also used for removing watermarks from images or to remove any object while filming a video or a movie.

FEATURE VARIATION

It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.

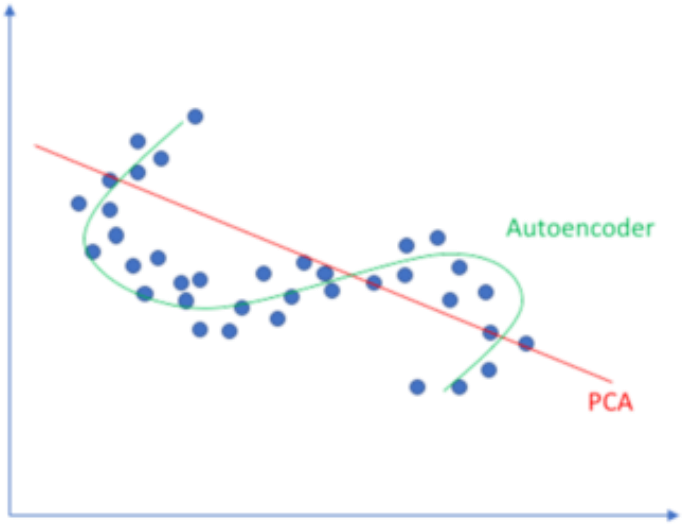


DIMENSIONALITY REDUCTION

The reconstructed image is the same as our input but with reduced dimensions. It helps in providing the similar image with a reduced pixel value.

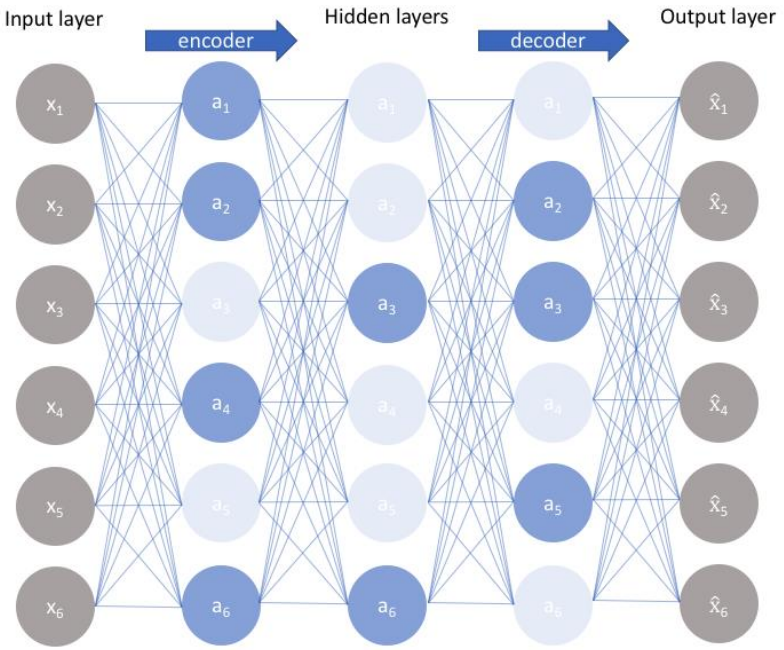
WHY AUTOENCODERS?

LINEAR VS. NONLINEAR DIMENSIONALITY REDUCTION



- An autoencoder can learn non-linear transformations with a non-linear activation function and multiple layers.
- It doesn't have to learn dense layers. It can use convolutional layers to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an autoencoder rather than learn one huge transformation with PCA.
- An autoencoder provides a representation of each layer as the output.
- It can make use of pre-trained layers from another model to apply transfer learning to enhance the encoder/decoder.
- So autoencoders are preferred over PCA!

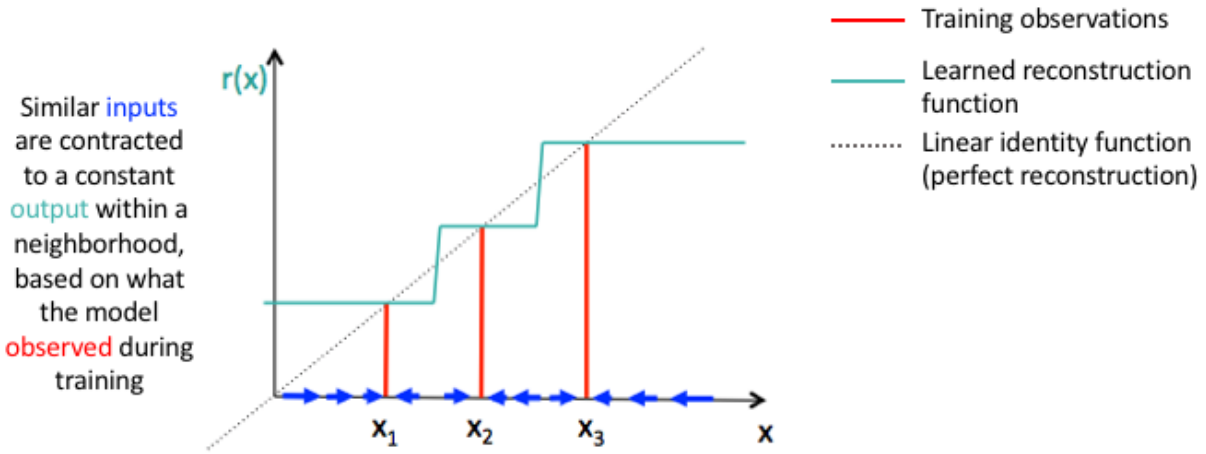
SPARSE AUTOENCODERS



- Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Instead, we'll construct our loss function such that we penalize activations within a layer.
- Detail: [Unsupervised Feature Learning and Deep Learning Tutorial (stanford.edu)]

CONTRACTIVE AUTOENCODER

A contractive autoencoder is an unsupervised deep learning technique that helps a neural network encode unlabeled training data. This is accomplished by constructing a loss term which penalizes large derivatives of our hidden layer activations with respect to the input training examples, essentially penalizing instances where a small change in the input leads to a large change in the encoding space.



AN EXPLICIT TERM IN THE LOSS OF CONTRACTIVE AUTOENCODER

An interesting approach to regularizing autoencoders is given by the assumption that for very similar inputs, the outputs will also be similar. We can enforce this assumption by requiring that the derivative of the hidden layer activations is small with respect to the input. This will make sure that small variations of the input will be mapped to small variations in the hidden layer. The name contractive autoencoder comes from the fact that we are trying to contract a small cluster of inputs to a small cluster of hidden representations.

Specifically, we include a term in the loss function which penalizes the Frobenius norm (matrix L2-norm) of the Jacobian of the hidden activations w.r.t. the inputs:

$$\|J_f(x)\|_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2$$

Hereby, h_j denote the hidden activations, x_i the inputs and $||*||_F$ is the Frobenius norm.

NEW LOSS FUNCTION IN CONTRACTIVE AUTOENCODER

New loss function:

$$\underbrace{l(f(\mathbf{x}^{(t)}))}_{\text{autoencoder reconstruction}} + \lambda \underbrace{||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2}_{\text{Jacobian of encoder}}$$

where, for binary observations:

$$l(f(\mathbf{x}^{(t)})) = -\sum_k \left(x_k^{(t)} \log(\hat{x}_k^{(t)}) + (1 - x_k^{(t)}) \log(1 - \hat{x}_k^{(t)}) \right)$$

encoder keeps good information

$$||\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})||_F^2 = \sum_j \sum_k \left(\frac{\partial h(\mathbf{x}^{(t)})_j}{\partial x_k^{(t)}} \right)^2$$

encoder throws away all information

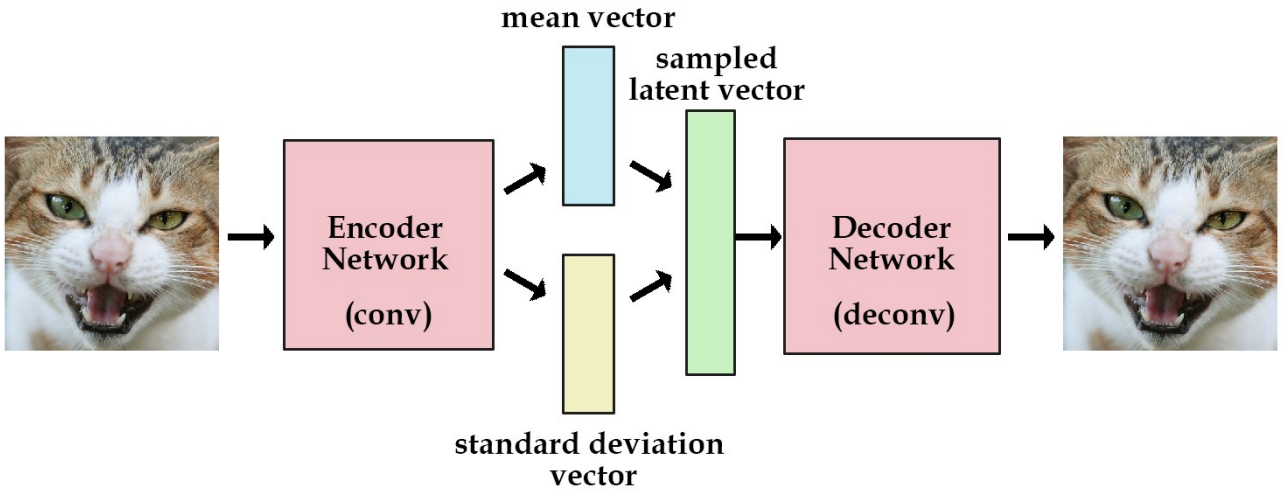
encoder keeps **only** good information

ANOMALY DETECTION

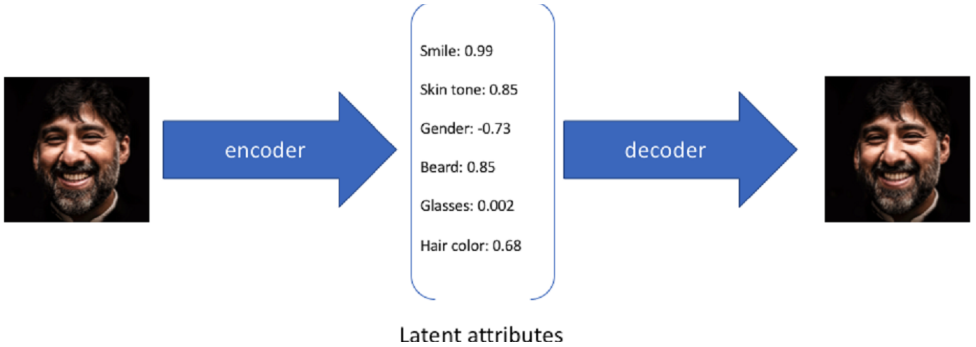
- if enough training data resembling some underlying pattern is provided, the network will train itself to easily learn the
identity when confronted with that pattern.
- if an anomalous test point does not match the learned pattern, **the auto encoder will likely have a high error rate in reconstructing this data, indicating anomalous data.**
- The dataset of ECG -heartbeats and the goal is to determine which heartbeats are outliers.

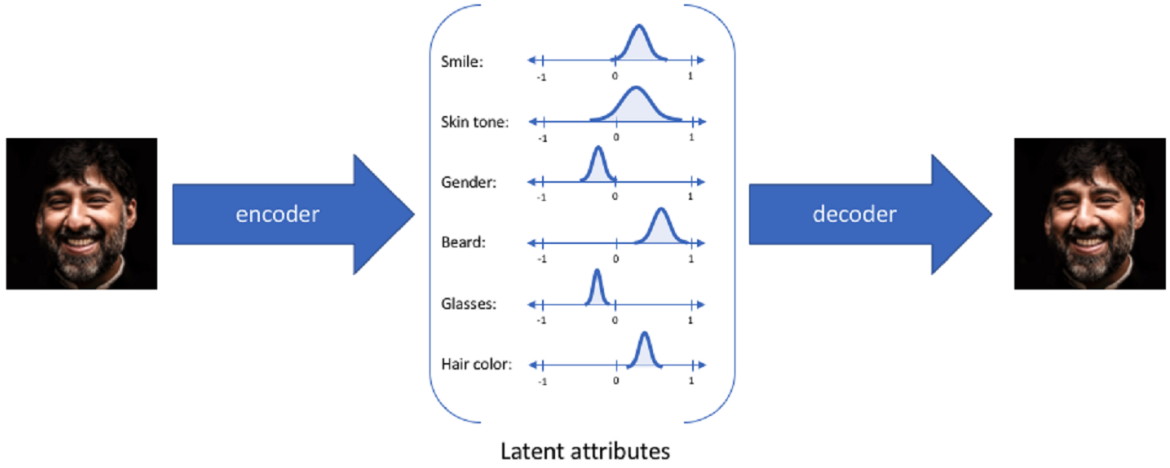
VARIATIONAL AUTOENCODER (I)

The crucial difference between variational autoencoders and other types of autoencoders is that VAEs view the hidden representation as a latent variable with its own prior distribution. This gives them a proper Bayesian interpretation. Variational autoencoders are generative models with properly defined prior and posterior data distributions.

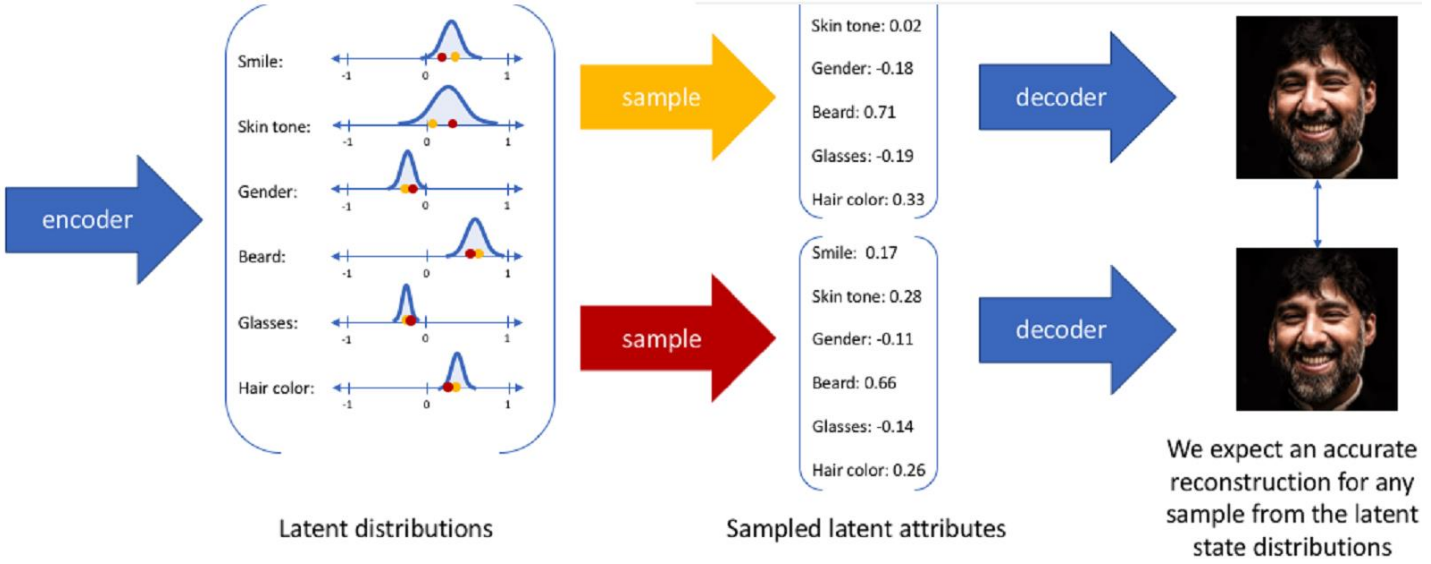


VARIATIONAL AUTOENCODER (II)





VARIATIONAL AUTOENCODER (III)



VARIATIONAL AUTOENCODER (IV)

In variational autoencoder, the encoder outputs two vectors instead of one, one for the **mean** and another for the **standard deviation** for describing the latent state attributes. These vectors are combined to obtain an encoding sample passed to the decoder for the reconstruction purpose.