COURSE : DEEP LEARNING
UNIT : I
TOPIC : INTRODUCTION TO DEEP LEARNING

DR.D.KAVITHA

UNIT - I
INTRODUCTION TO DEEP LEARNING

- Introduction
- Historical Trends in Deep Learning

INTRODUCTION

Problems/Tasks for Human Beings

Intellectually difficult
Ex Multiplying two 10-digit numbers
Playing Chess

Easy for computers
IBM's Deep Blue chess-playing system

Can be described by a list of formal mathematical rules.

Easy, feels automatic, solved intuitively
Ex Understanding / recognizing spoken words
Identifying people faces

Difficult for computers
Less than ability of an average human beings to recognize objects or speech.

Hard for people to describe formally.

Solutions for these intuitive problems....?

SOLUTIONS FOR THE INTUITIVE PROBLEMS

- Allow computers to learn from experience.
- Understand the world in terms of hierarchy of concepts.
- Learn complicated concepts by building them out of simpler ones.

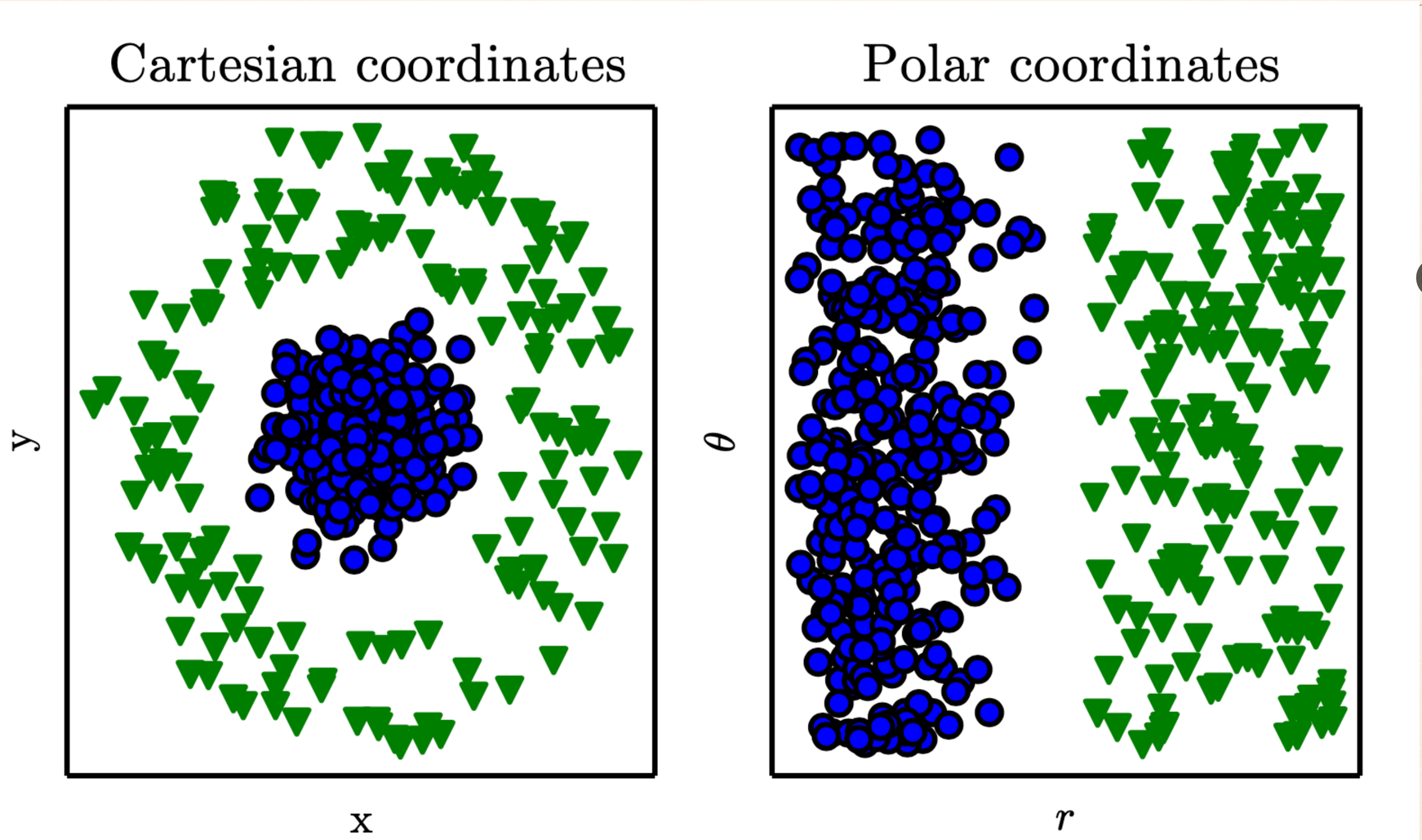
SOLUTION 1- KNOWLEDGE BASE

- AI projects hard code knowledge about the world in formal languages.
- Ex: Cyc is an inference engine and a database of statements in a language called CycL.
- Formal Rules to understand world.
  - Detected an inconsistency.
  - People do not have electrical parts.
    - Fred was holding an electric razor.
    - Failed to understand “FredWhileShaving”.
    - ***Is Fred still a person while shaving…?***

SOLUTION 2- MACHINE LEARNING

- AI systems needed the ability to acquire their own knowledge- extract patterns from raw data - Machine Learning -
    - Ex 1: Logistic Regression to identify cesarean delivery. If logistic regression is given MRI scan of the patient instead of doctor's formalized report, cannot make useful prediction.
    - Ex 2 : Naive Bayes to separate legitimate e-mails from spam e-mail.
    - Speaker Identification - size of speakers vocal tract - man, woman, child
- Performance depends on data representation
- Each piece of information represented in the data -feature
- Challenge - Extracting right set of features

REPRESENTATIONS MATTER



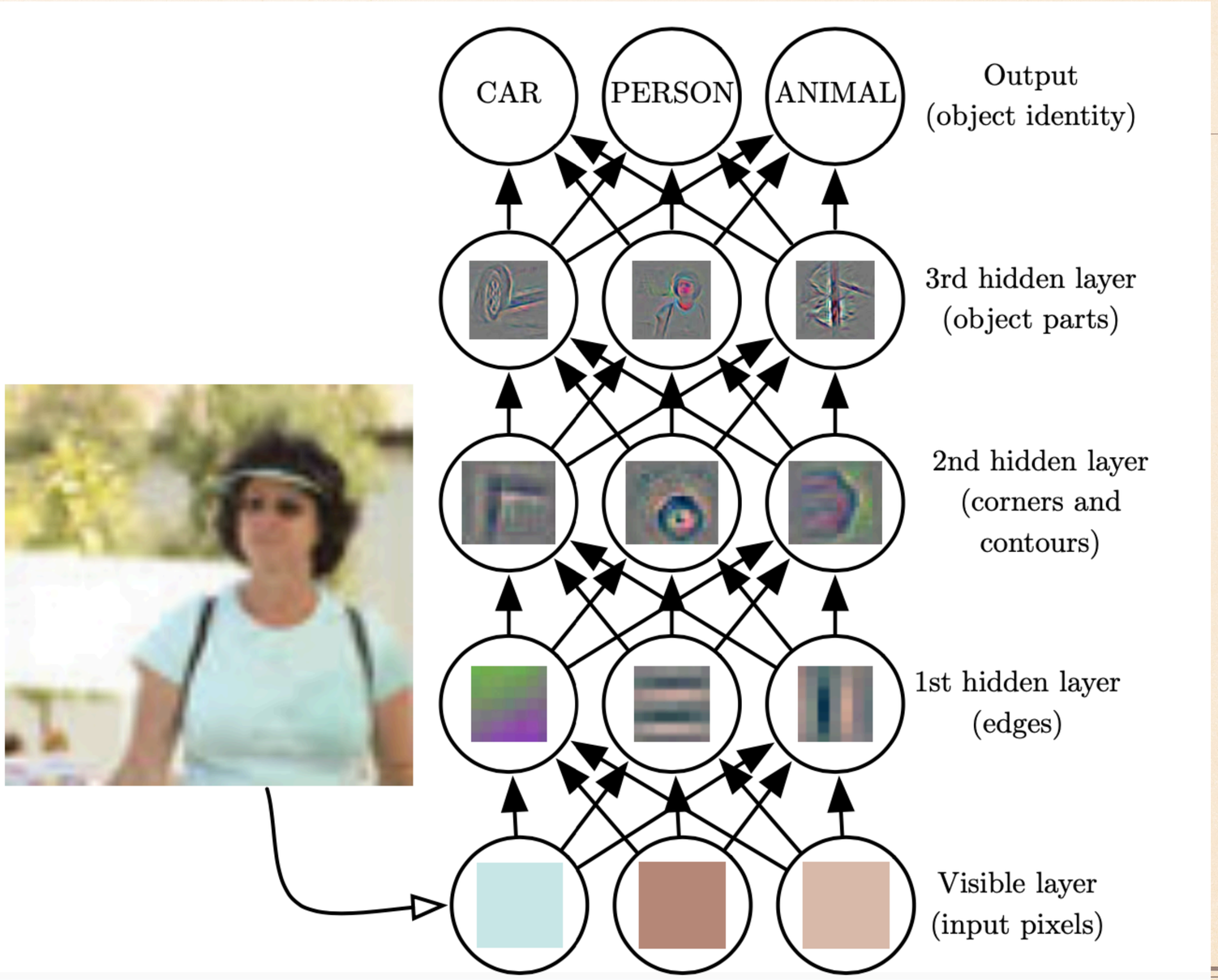
Cartesian Coordinates
or
Polar Coordinates

SOLUTION 3 - REPRESENTATION LEARNING

- Let AI systems allow ML algorithms loonily discover representation to output, but also representation itself.
- Learned representations - Better performance than hand designed representations
- Simple task - minutes , complex task- hours/months
- Ex : Autoencoder - Representation learning algorithm
    - Encoder function -Converts input data into different representation
    - Decoder Function- Converts new representationbackinto original format
- Separate factors of variation
    - Speech recording analysis - Speaker's age, sex, accent, words they are speaking
    - Analyzing car image - position of the car, color, angle and brightness of the sun
- Factors of variation affect every single piece of data.
    - Pixels in an image of a red car might appear black at night.

SOLUTION 4 - DEEP LEARNING

- Complex concepts built out of simpler concepts.
- Image of a person is represented by combining simpler concepts - edges, corners, contours, object parts.
- Ex : Multilayer perceptron

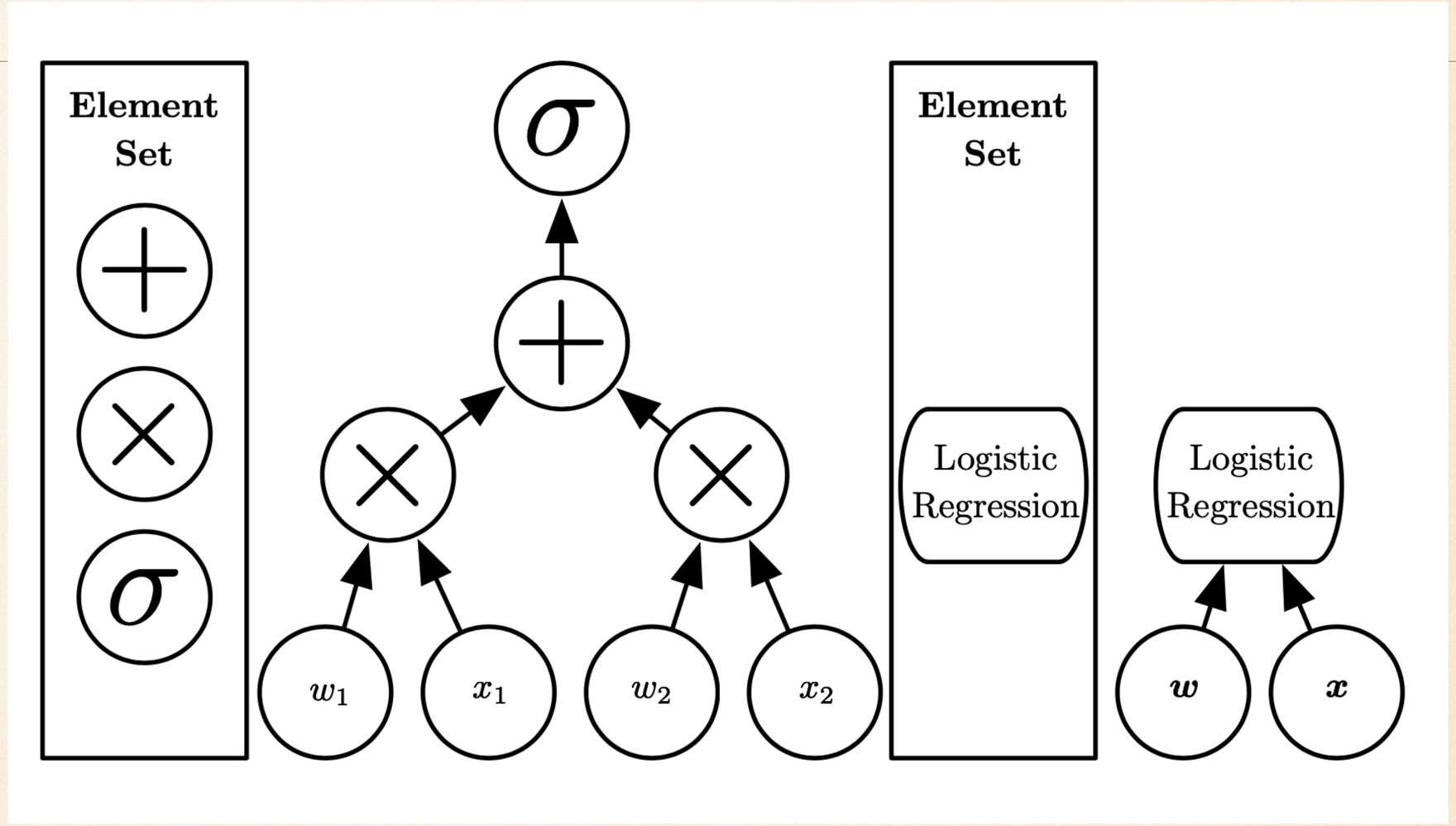


SOLUTION 4 - DEEP LEARNING

- Function mapping from a set of pixels to an object identity is very complicated.
- Complicated mapping is divided into simple mappings.
- Visible Layer - input - Contains the variables that are abletoobserve.
- Hidden Layers - Extracts abstract features from the image
- Model determines which concepts are useful for explaining relationships unobserved data.
- Networks with greater depth execute more instructions in sequence.

MEASURING THE DEPTH OF MODEL-COMPUTATIONAL GRAPHS

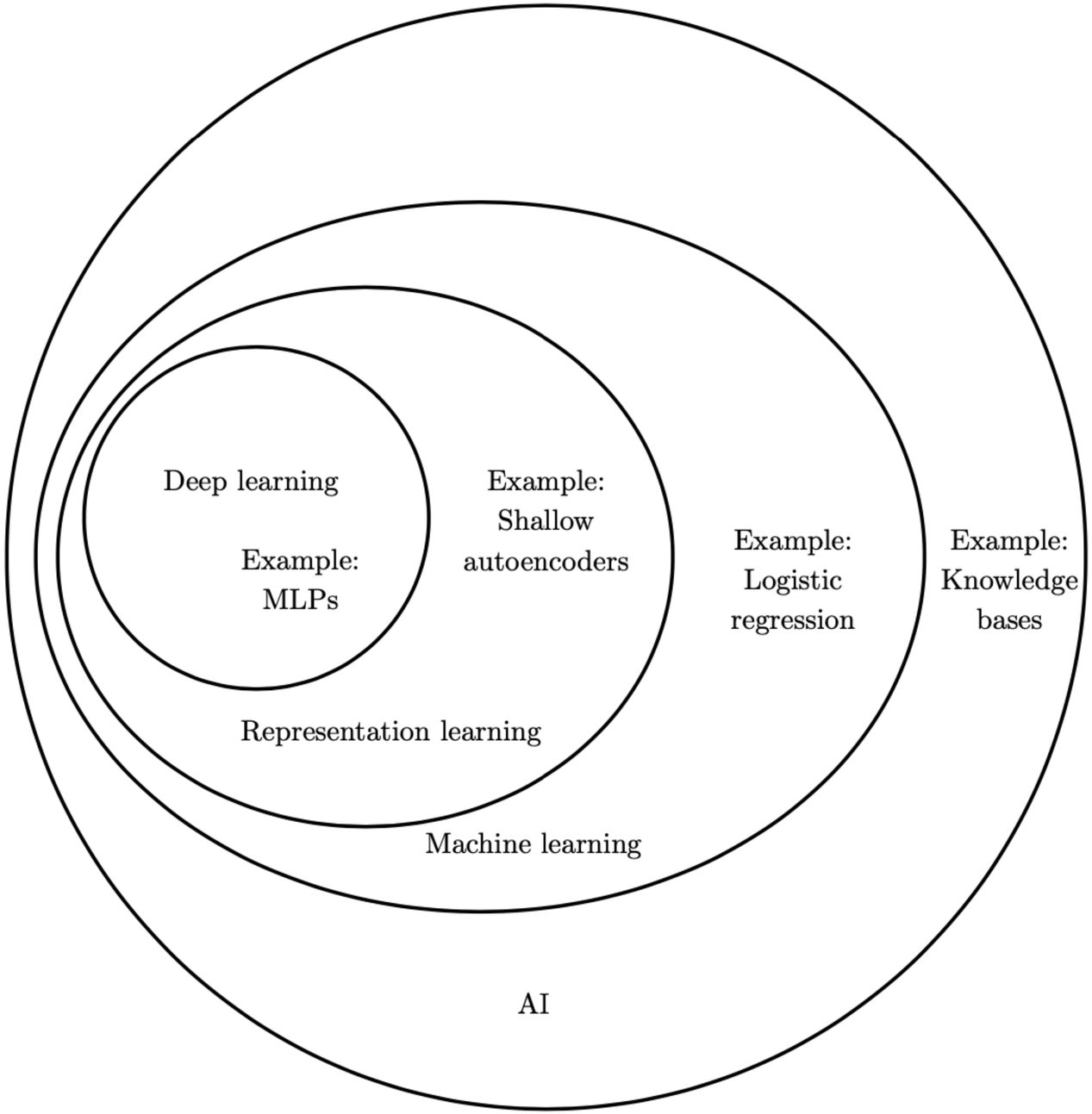
- No.of sequential instructions that must be executed to evaluate architecture. Length of the longest path from input to output.



MEASURING THE DEPTH OF MODEL-DEEP PROBABILISTIC MODELS

- Depth of the graph describing how concepts are related to one another.
- Ex :An AI system observing the image of a face with one eye in shadow.
  - Deep probabilistic models consists of 2 layers - a layer for eyes, layer for faces.
  - Computational graphs includes 2n layers - each concept times.

No single value for the correct depth of an architecture.



Machine Learning and AI

Housing data set

x_0	Size (ft^2) (x_1)	No. of bedrooms (x_2)	No. of floors (x_3)	Age of home (x_4)	Price ($)
1	2104	5	1	45	450
1	1416	3	2	40	232
1	1534	3	2	30	315
	⋮				

n = no. of features. Here $n = 4$.

$x^{(i)}$ = i/p of i^{th} training example

$x_j^{(i)}$ = value of feature j in i^{th} training example.

Here $x^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix}$

x_0 = bias/intercept term

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$

Hypothesis

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

(also written as $\hat{y}$) (Predicted value)

where $x_0^{(i)} = 1$

$$\hat{y} = h_\theta(x) = \theta^T x$$

$$= [\theta_0 \ \theta_1 \ \theta_2 \ \dots \ \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \theta^T x$$

Training a model means setting the parameters θ so that the model best fits the training set.

We need to measure how well the model fits.

So performance measures for Regression model are

RMSE, MSE.

MSE fn. for linear Regression Model

$$MSE(X, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2$$ cost fn.

To find θ, that minimizes cost fn., there is a closed form solution also called Normal Equation.

$$\hat{\theta} = (X^T X)^{-1} X^T y.$$

$\hat{\theta}$ = value of θ that minimises cost fn.

Computation complexity of inverting a matrix is about $O(n^{2.4})$ to $O(n^3)$. Increases with n.

However, it is linear to no. of instances.

Gradient Descent

optimization algorithm to tweak the parameter to minimize the cost fn.

Start with random initialization of θ.

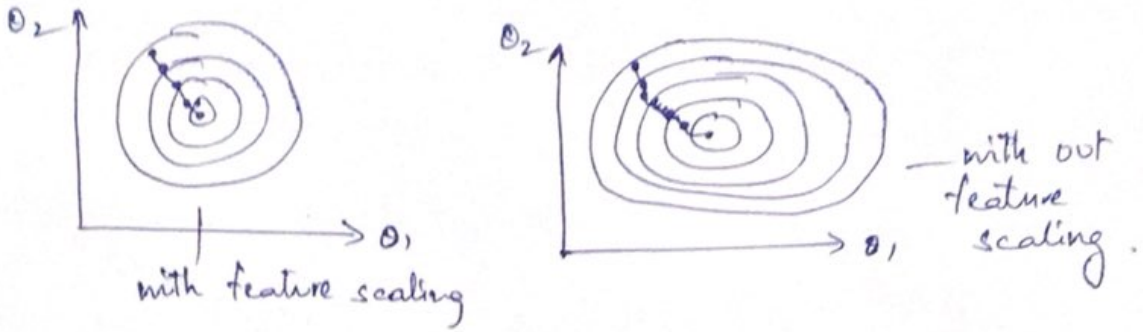
Take step by step to decrease cost fn (MSE) until algorithm converges to minimum.

learning rate α is hyperparameter.

α is too small $\Rightarrow$ many iterations to converge

α is too high $\Rightarrow$ algorithm diverges, with large values, fails to find good solution.

MSE cost fn. for a linear regression is convex, so no local minima, only one global minima.



Batch Gradient Descent

Compute the gradient of cost fn. w.r.t each model parameter θ_j

How much cost fn. will change by, changing θ_j just a little.

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right) x_j^{(i)}$$

Instead of computing them individually, use gradient vector. $\nabla_\theta MSE(\theta)$

$$\nabla_\theta MSE(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{bmatrix} = \frac{2}{m} X^T (X\theta - y)$$

Uses whole batch of training data at each and every step.

GD next step is

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta MSE(\theta)$$

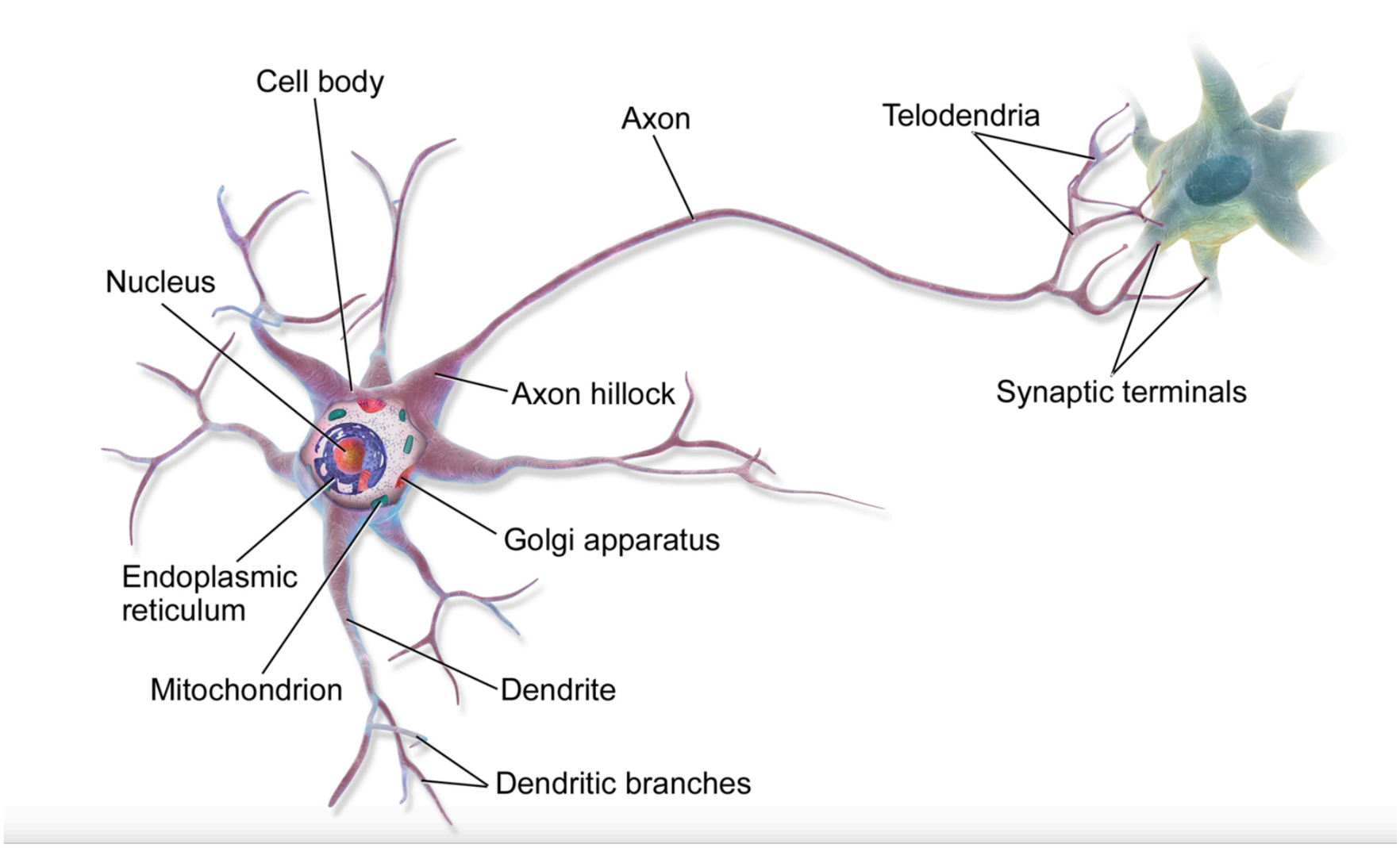
If u have gradient descent vector going uphill just go in opposite direction by subtracting $\nabla_\theta MSE(\theta)$ from θ.

Introduction to Artificial Neural Networks

UNIT - II

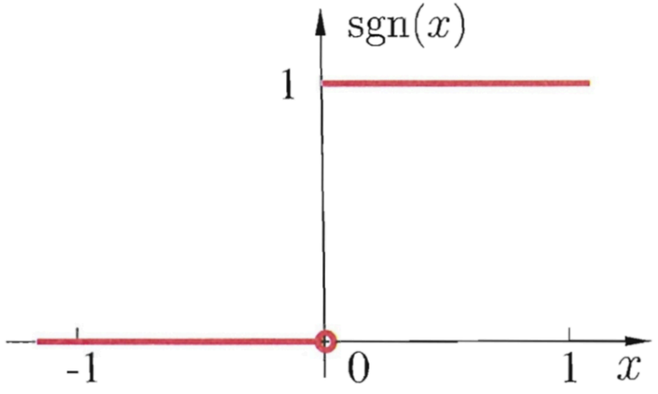
Kavitha

Biological Neuron

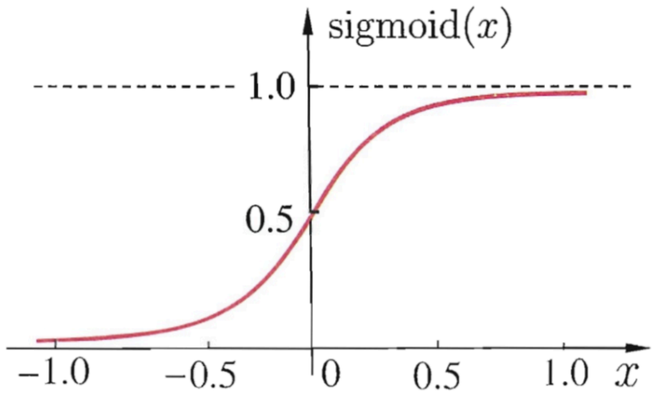


Artificial Neuron

- *Artificial neuron*, also called *linear threshold unit* (LTU), by McCulloch and Pitts, 1943: with one or more numeric inputs, it produces a weighted sum of them, applies an *activation function*, and outputs the result.
- Common activation functions: step function and sigmoid function.



$$\text{sgn}(x) = \begin{cases} 1, & x \geqslant 0; \\ 0, & x < 0. \end{cases}$$

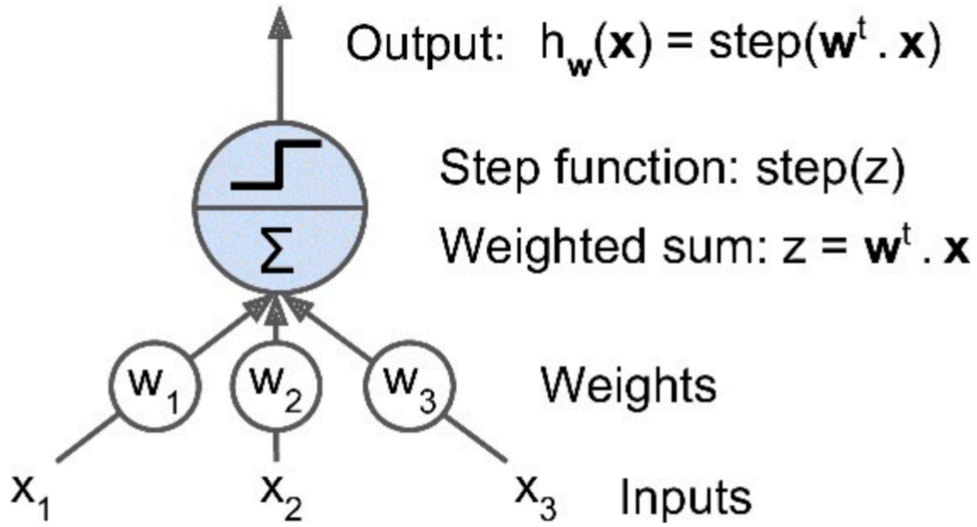


$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

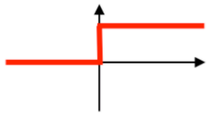
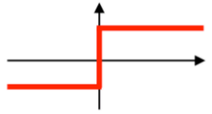
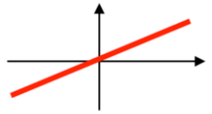
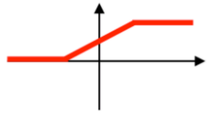
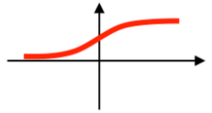
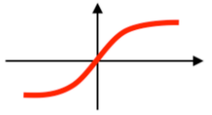
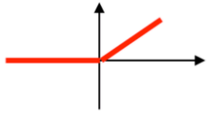
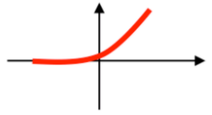
- Other activation functions are tanh and ReLU
- Perceptrons and Sigmoid Neurons

LTU with step function - Perceptron

- Below is an LTU with the activation function being the step function.

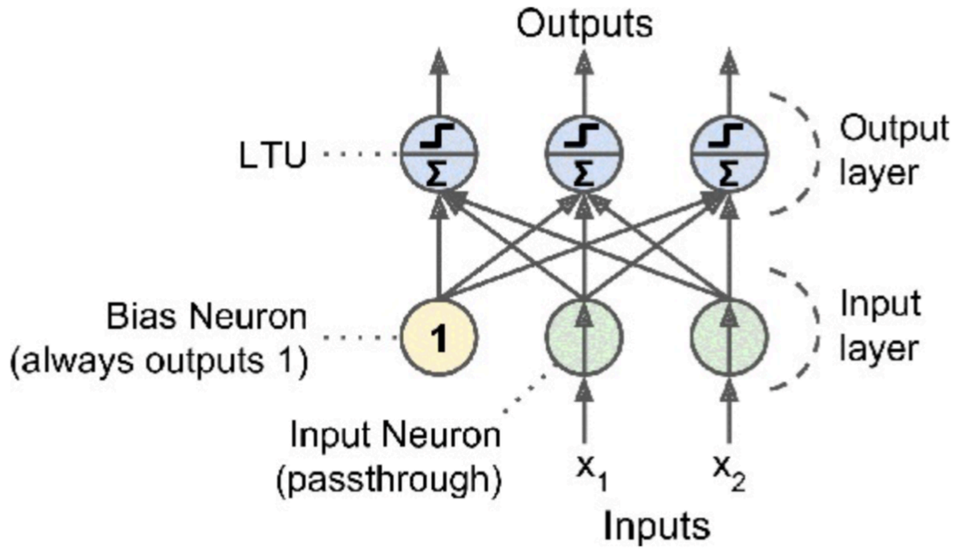


Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

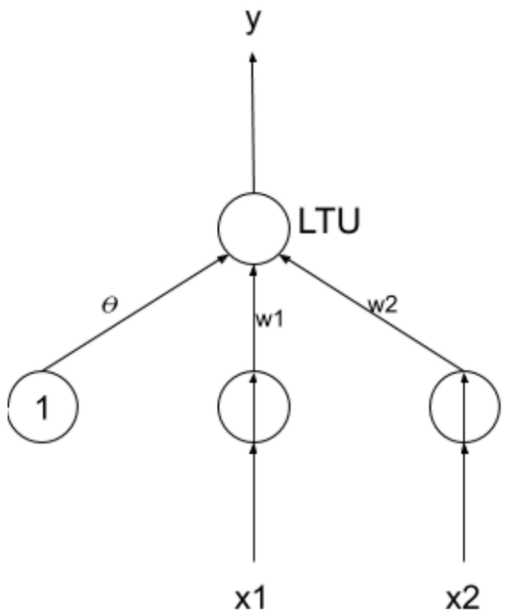
Perceptron

- A *perceptron*, by Rosenblatt in 1957, is composed of two layers of neurons: an input layer consisting of special passing through neurons and an output layer of LTU's.
- The bias neuron is added for the completeness of linearity.
- Rosenblatt proved that, if training examples are linearly separable, a perceptron always can be learned to correctly classify all training examples.



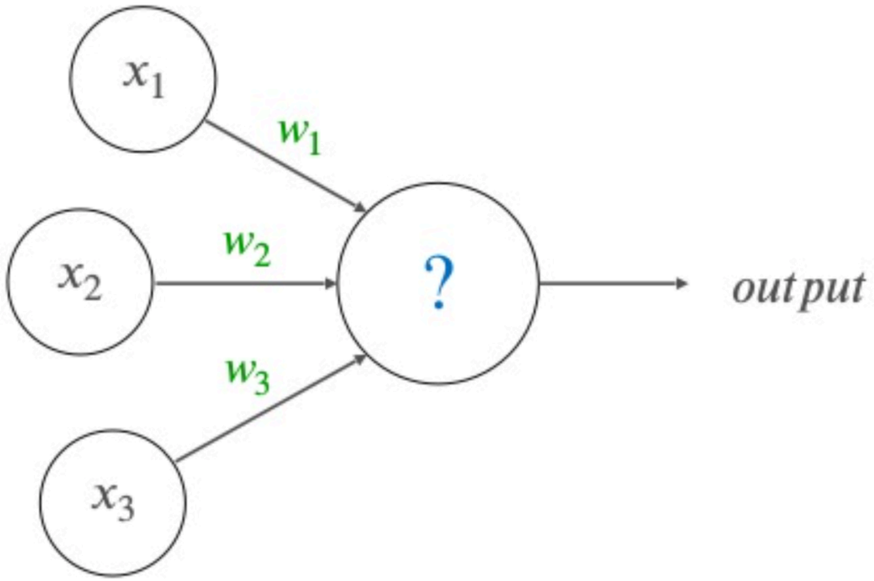
Perceptrons

- For instance, perceptrons can implement logical conjunction, disjunction and negation.
- For the following perceptron of one LTU with the step function as the activation function.
  - $x_1 \wedge x_2$: $w_1 = w_2 = 1, \theta = -2$
  - $x_1 \vee x_2$: $w_1 = w_2 = 1, \theta = -0.5$
  - $\neg x_1$: $w_1 = -0.6, w_2 = 0, \theta = -0.5$



Perceptrons

Simplified (binary) artificial neuron with weights

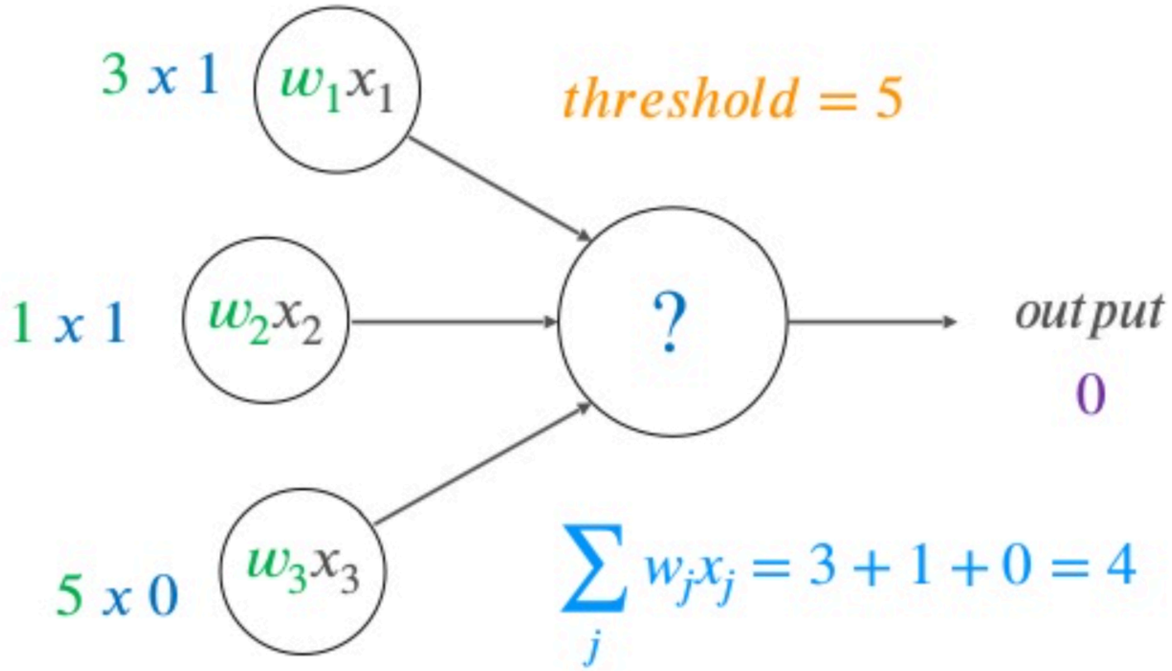


$$output = \begin{cases} 0, & \sum_{j=0}^{n} w_j x_j \leq threshold \\ 1, & \sum_{j=0}^{n} w_j x_j > threshold \end{cases}$$

Perceptrons

Simplified (binary) artificial neuron; *add weights*

Person : Ashok



Do I snowboard this weekend?

$x_1 = 1$ (*good weather*)

$w_1 = 3$

$x_2 = 1$ (*a lot of powder*)

$w_2 = 1$

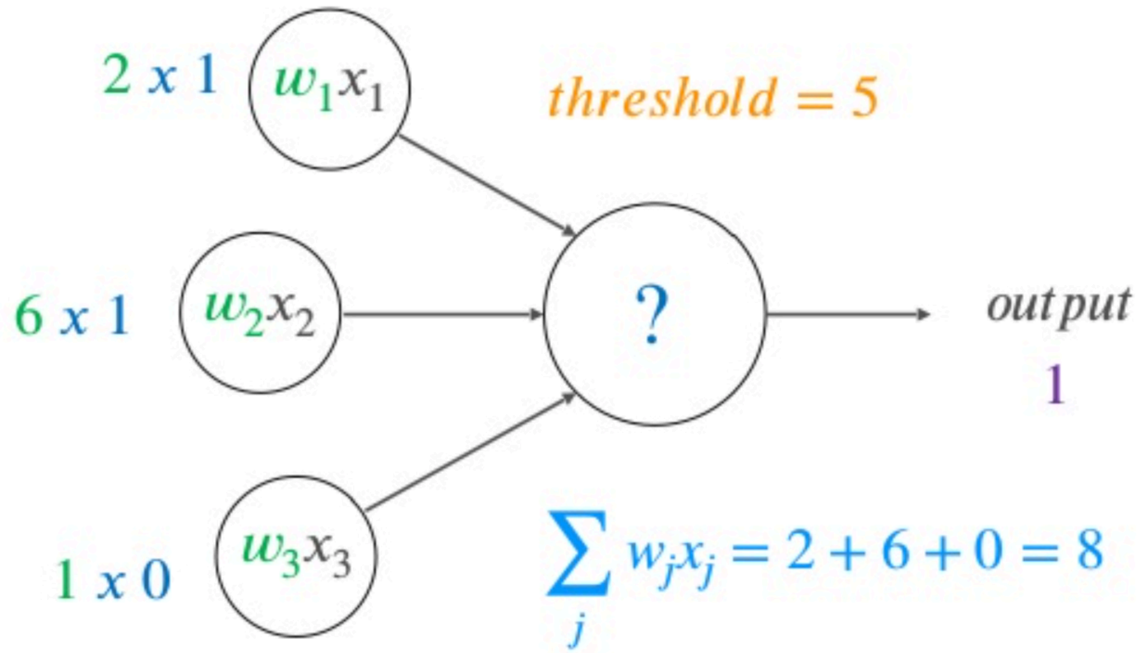
$x_3 = 0$ (*driving sucks*)

$w_3 = 5$

Perceptron

Simplified (binary) artificial neuron; *add weights*

Persona: Shredder



Do I snowboard this weekend?

$x_1 = 1$ (*good weather*)

$w_1 = 2$

$x_2 = 1$ (*a lot of powder*)

$w_2 = 6$

$x_3 = 0$ (*driving sucks*)

$w_3 = 1$

Introducing Bias

Perceptron needs to take into account the bias

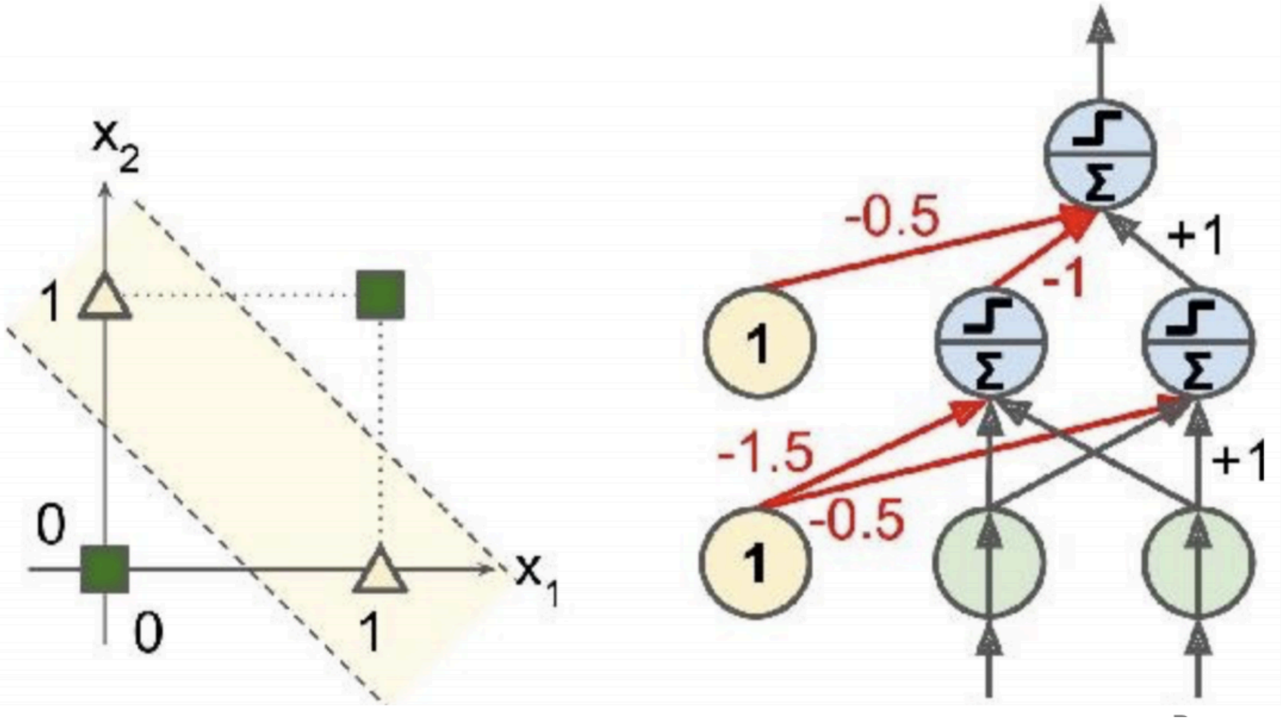
$$output = \begin{cases} 0, & wx + b \leq 0 \\ 1, & wx + b > 0 \end{cases}$$

where b is how easy it is to get the perceptron to fire

Shredder has strong positive bias to go to Whistler while Ashok is not as strong

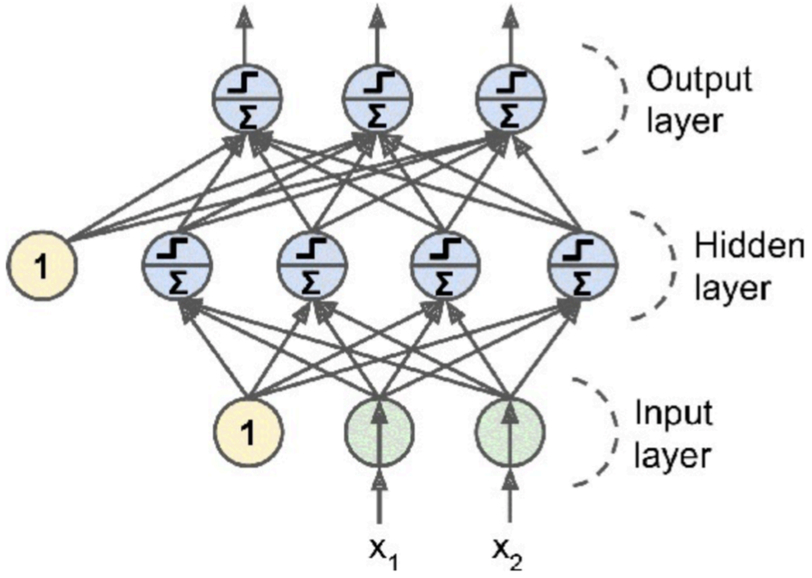
Multi Layer Perceptron

- However, perceptrons cannot solve some trivial non-linear separable problems, such as the Exclusive OR classification problem.
- This is shown by Minsky and Papert in 1969.
- Turned out stacking multiple perceptrons can solve any non-linear problems.



Multi Layer Perceptron

- A *Multi-Layer Perceptrons* (MLP) is composed of one passthrough input layer, one or more layers of LTU's, called *hidden layer*, and one final layer of LTUs, called *output layer*.
- Again, every layer except the output layer includes a bias neuron and is fully connected to the next layer.
- When an MLP has two or more hidden layers, it is called a *deep neural network* (DNN).

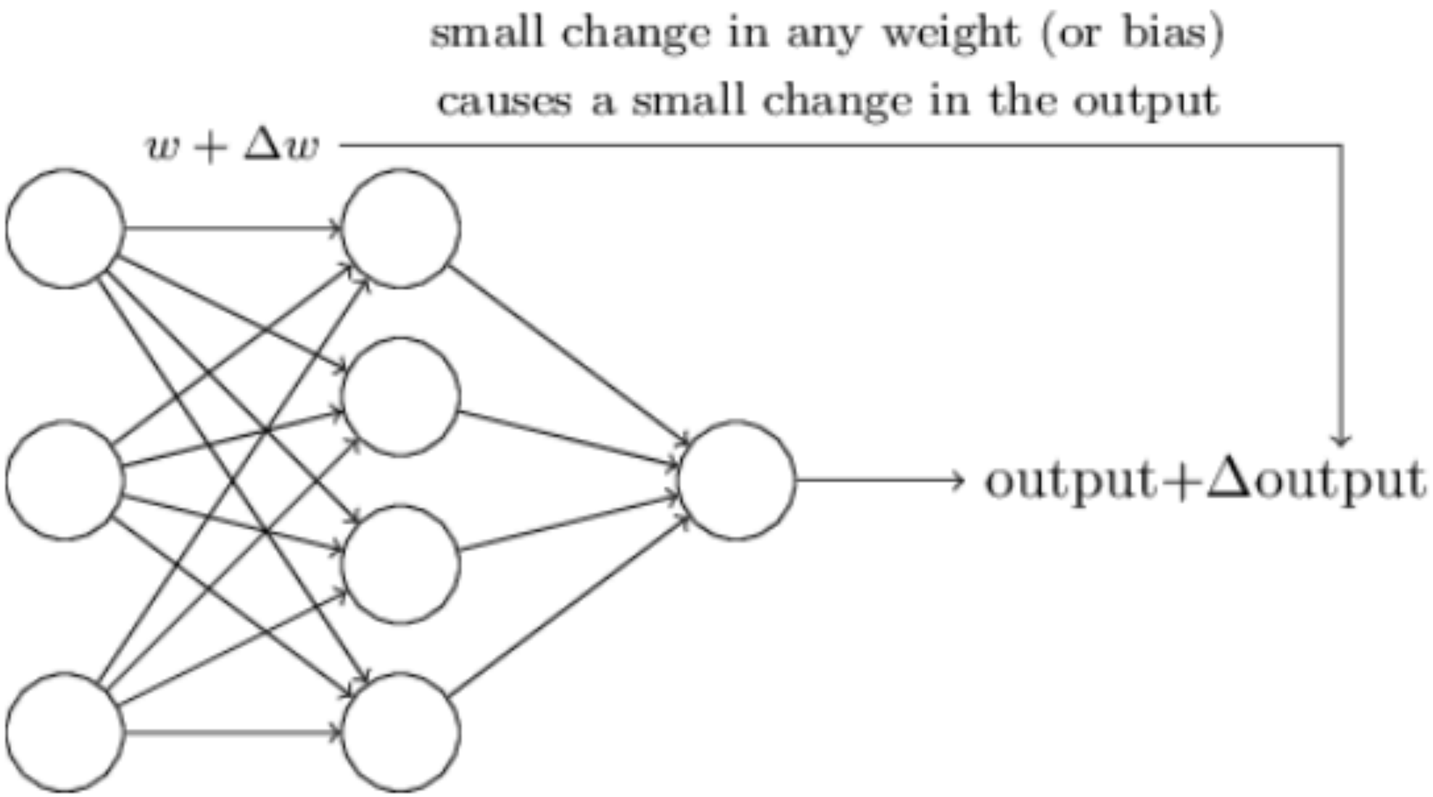


Multi Layer Perceptron

- Complex network perceptrons can make subtle decisions.
- Output = 0 if wx + b less than or equal to 0

  = 1 if wx + b is greater than 0

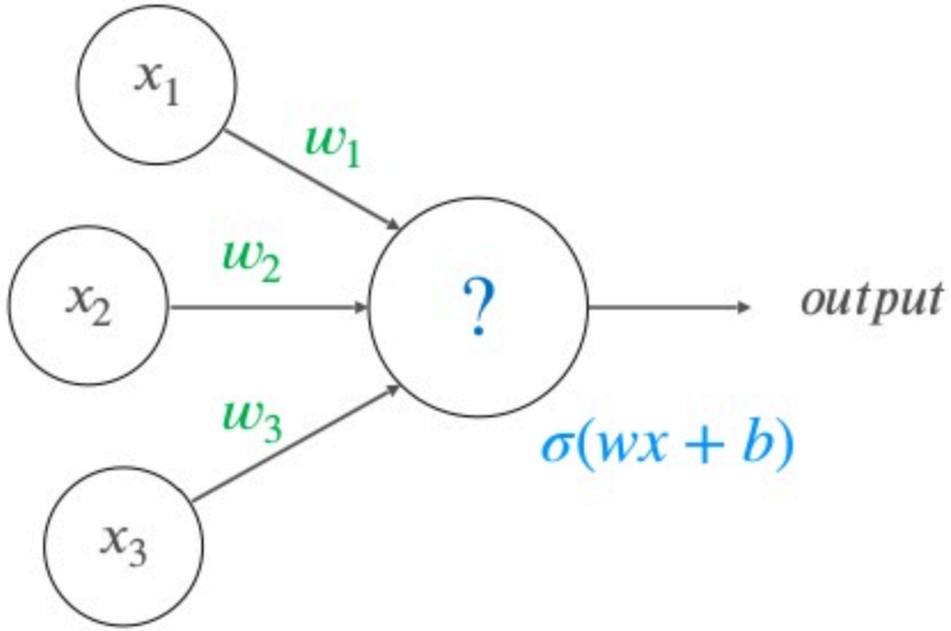
Sigmoid Neuron

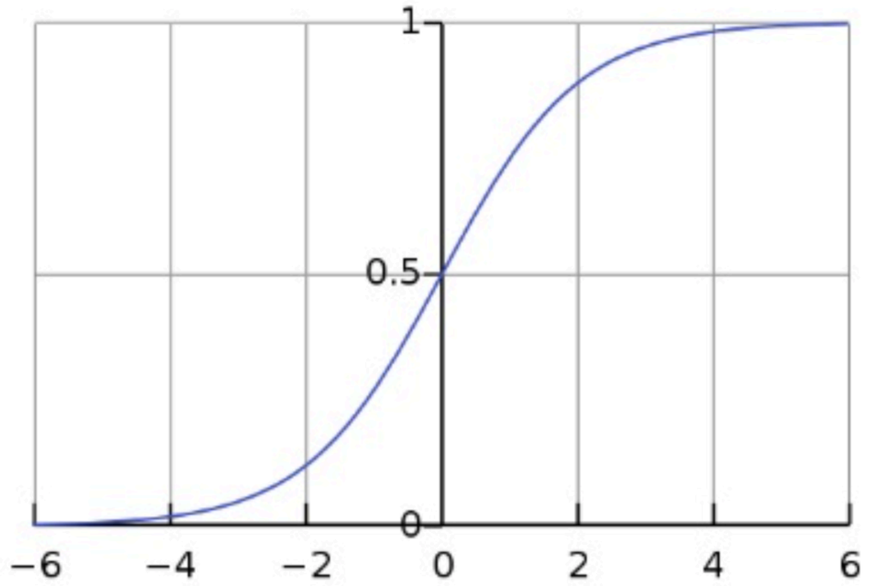


$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

Sigmoid Neuron

The more common artificial neuron



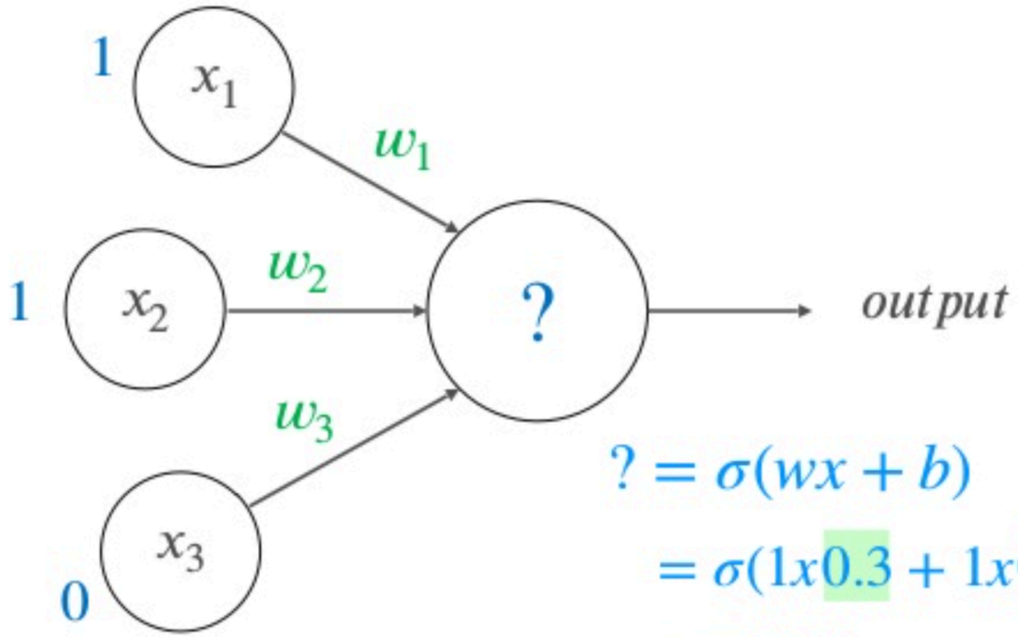


Instead of [0, 1], now (0…1)

Where output is defined by $\sigma(wx + b)$

Sigmoid Neuron

Persona: Shredder



$$? = \sigma(wx + b)$$
$$= \sigma(1x0.3 + 1x0.6 + 0x0.1)$$
$$= \sigma(0.9)$$
$$= 0.7109$$

Do I snowboard this weekend?

$x_1 = 1$ (*good weather*)

$w_1 = 0.3$

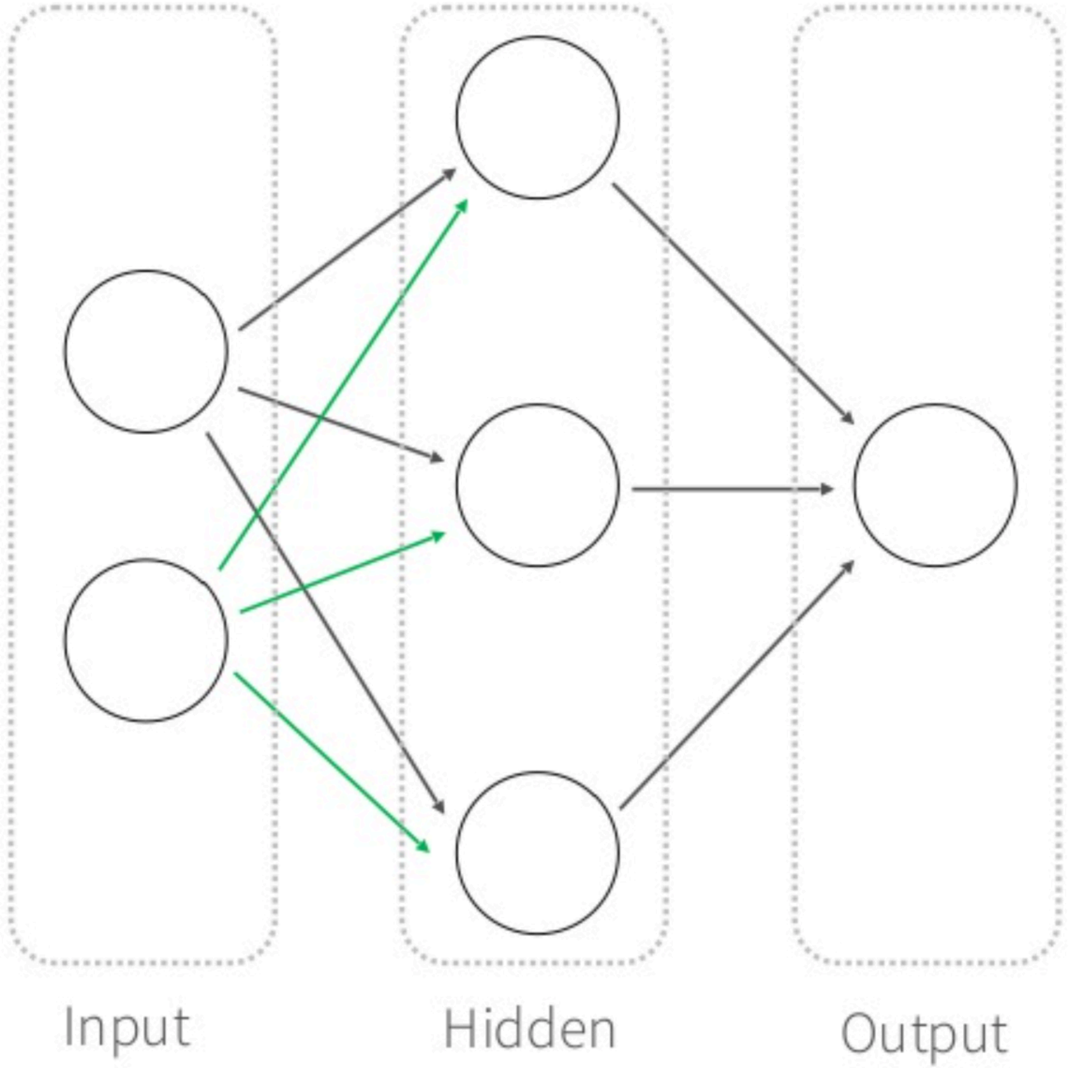
$x_2 = 1$ (*a lot of powder*)

$w_2 = 0.6$

$x_3 = 0$ (*driving sucks*)

$w_3 = 0.1$

Simplified Two Layer ANN



Do I snowboard this weekend?

$x_1 \rightarrow$ *Apres Ski'er*

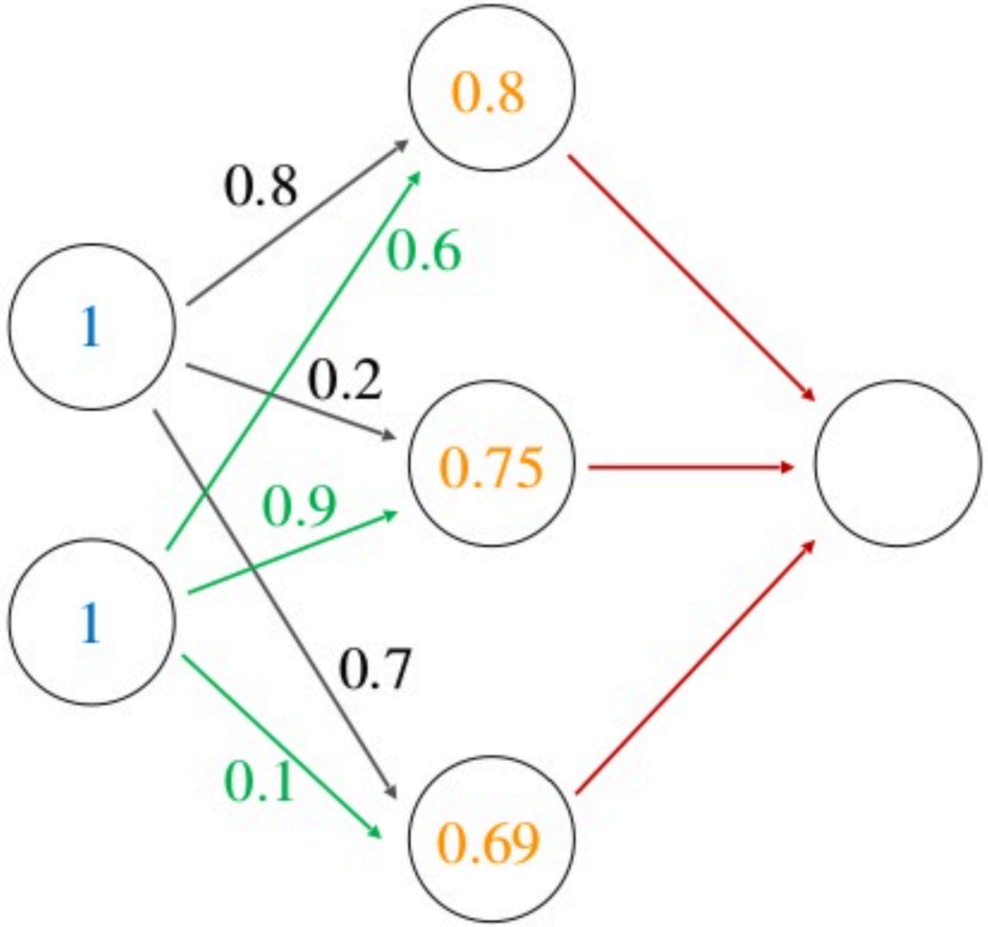
$x_2 \rightarrow$ *Shredder*

$h_1 \rightarrow$ *weather*

$h_2 \rightarrow$ *powder*

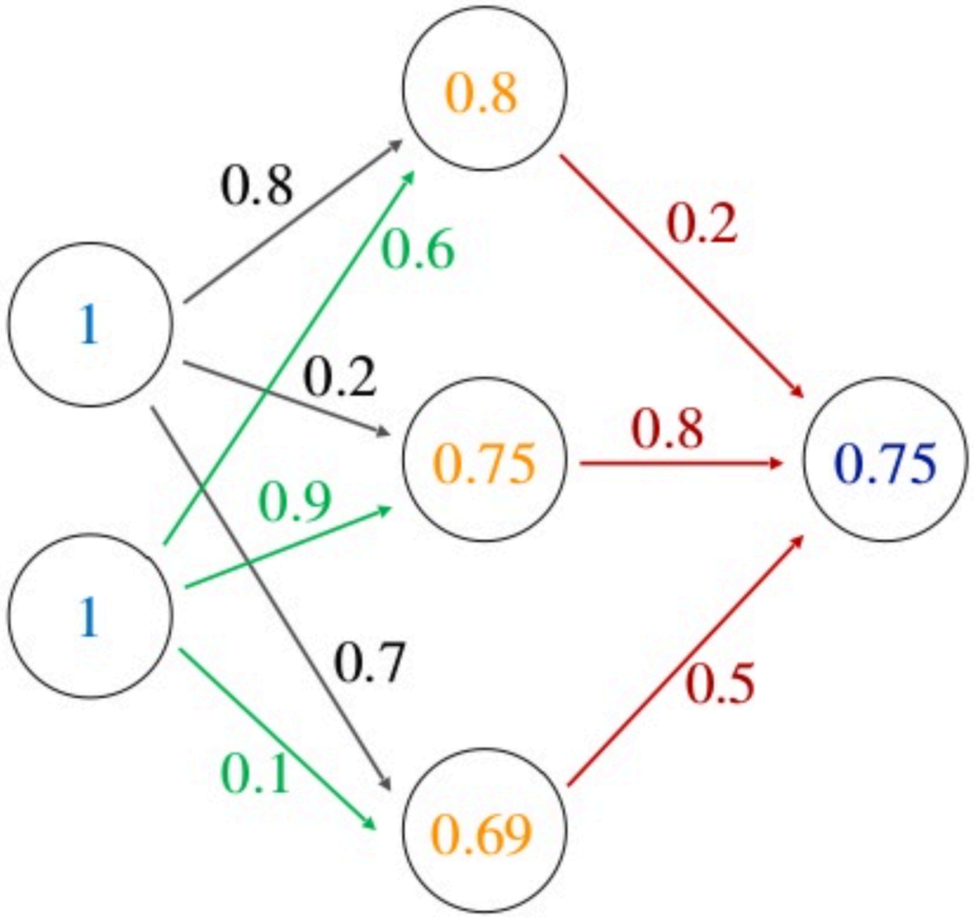
$h_3 \rightarrow$ *driving*

Simplified Two Layer ANN



$$h_1 = \sigma(1x0.8 + 1x0.6) = 0.80$$
$$h_2 = \sigma(1x0.2 + 1x0.9) = 0.75$$
$$h_3 = \sigma(1x0.7 + 1x0.1) = 0.69$$

Simplified Two Layer ANN

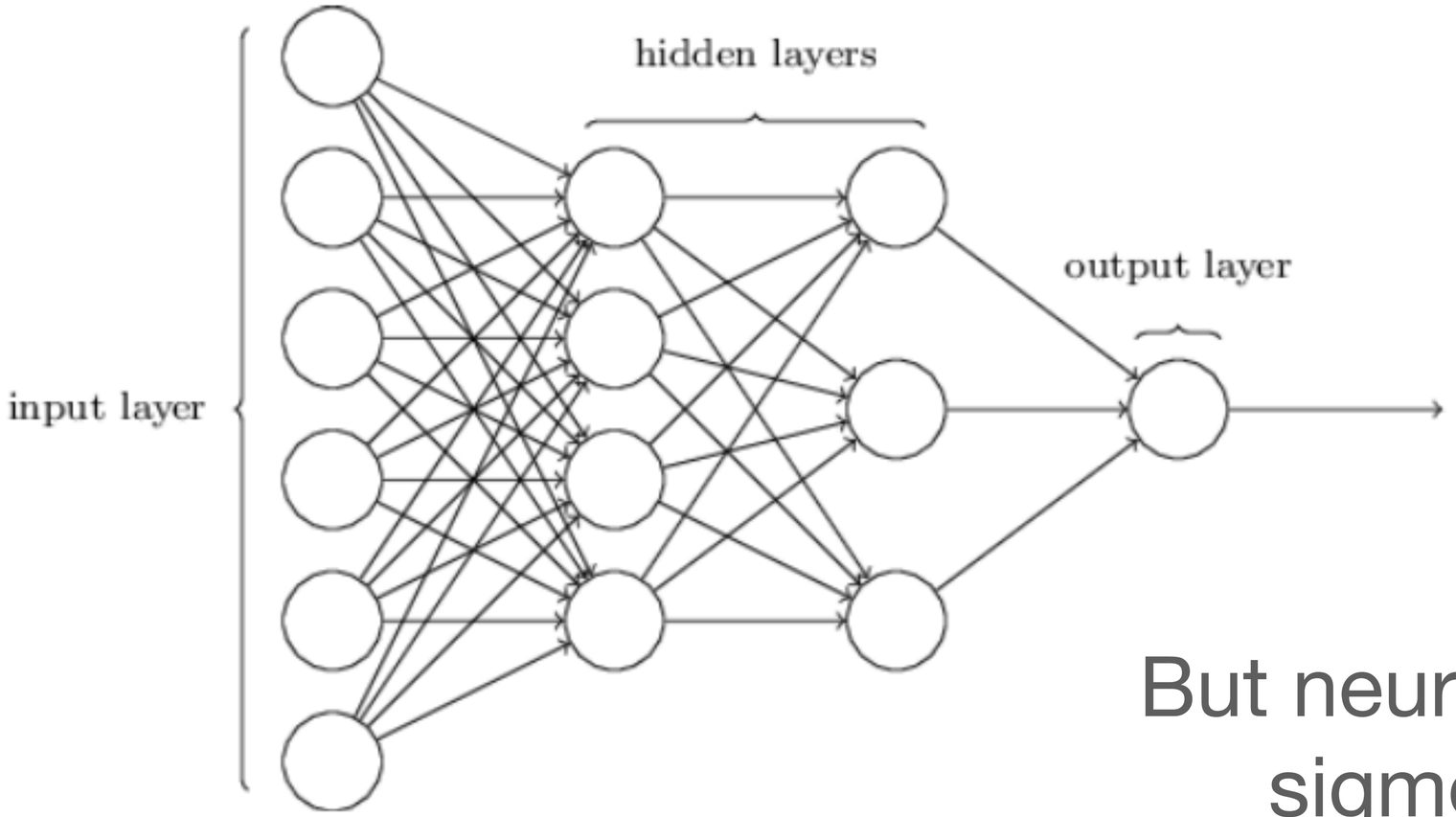


$$out = \sigma(0.2x0.8 + 0.8x0.75 + 0.5x0.69)$$

$$= \sigma(1.105)$$

$$= 0.75$$

Architecture of Artificial Neural Network - Multi Layer Perceptrons



But neurons in MLP are sigmoid neurons

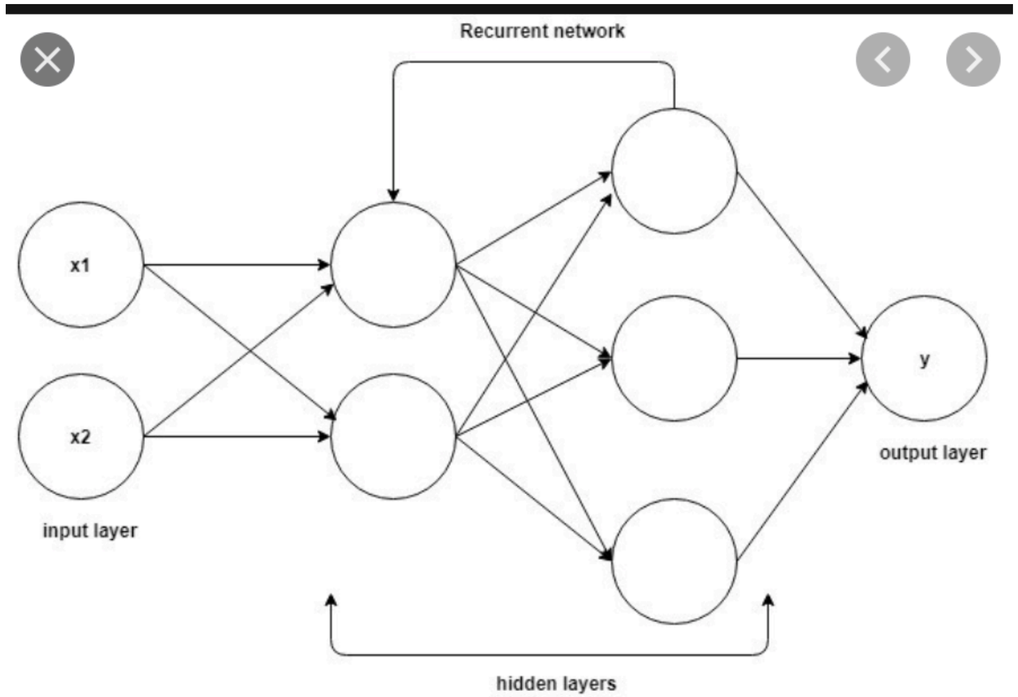
Architecture of Artificial Neural Network

Feed Forward Neural Network

- Output from one layer is fed as input to the next layer.
- Noloops in the network.

Recurrent Neural Networks

- Feedback loops are possible.



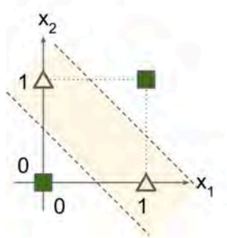
Learning Perceptrons

- Idea : Cells that fire together, wire together - Hebb's Rule/Hebbian Learning
- Perceptron Learning Rule

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\hat{y}_j - y_j\right)x_i$$

  - $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neu ron.
  - x_i is the i^{th} input value of the current training instance.
  - $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
  - y_j is the target output of the j^{th} output neuron for the current training instance.
  - η is the learning rate.

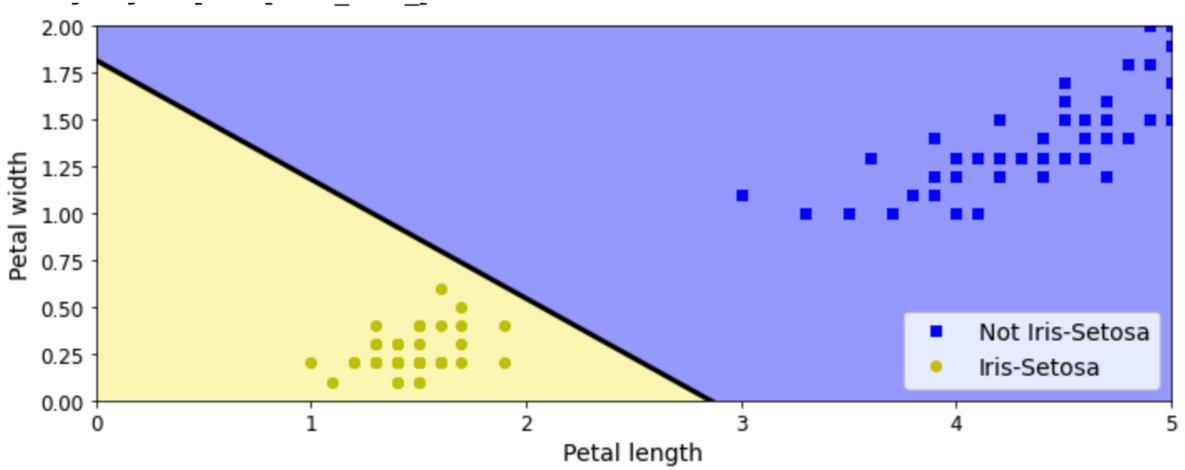
- Perceptron Convergence Theorem - Converge to solution if training instances are linearly separable.
- Not capable of of solving XOR classification problem.
- Do not output class probabilities, instead make predictions based on hard threshold.



Learning Perceptrons

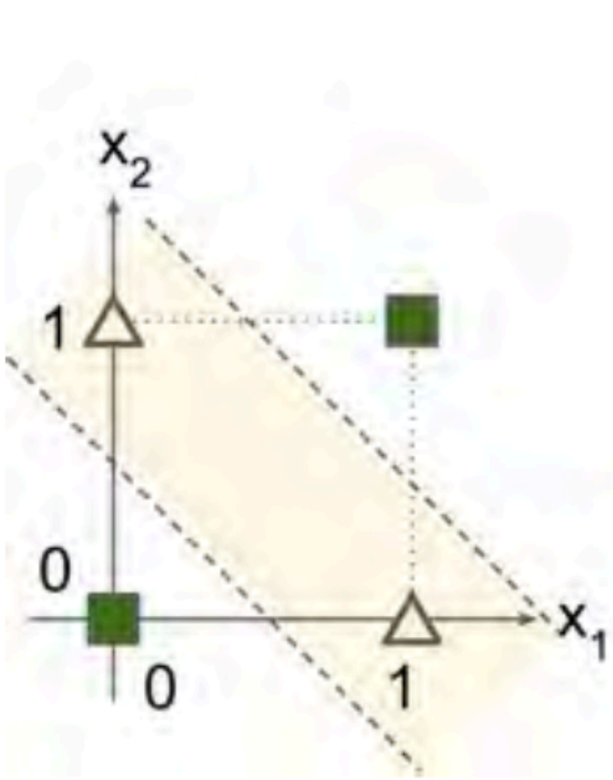
Iris data set - Implementation using Scikit Learn

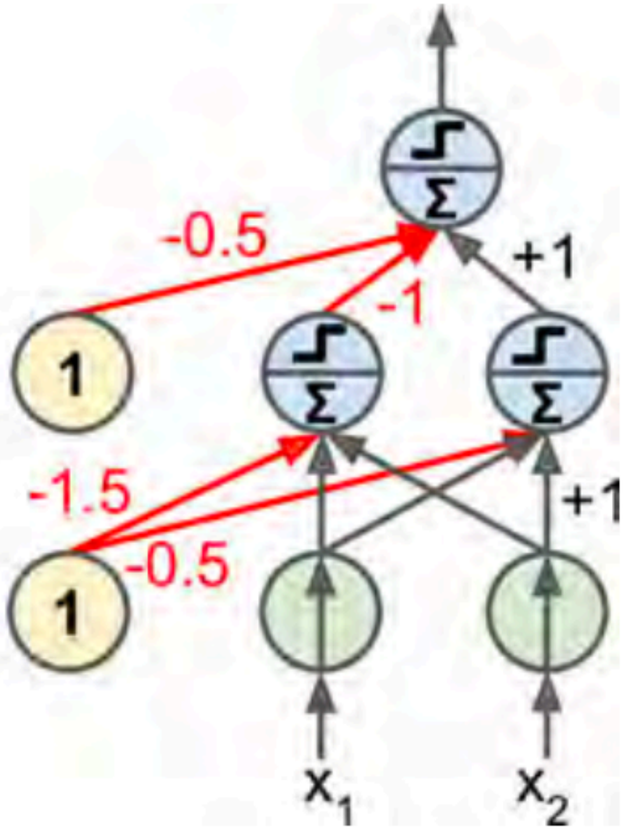
- No. of training instances : 150, No. of columns : 4
- Input - Petal Length, Petal Width
- Output(Target) - Iris Setosa? *Binary Classification Problem*
- Uses Perceptron class present in Scikit-Learn
- *Refer to Jupyter Notebook for implementation details.*



Multi Layer Perceptrons

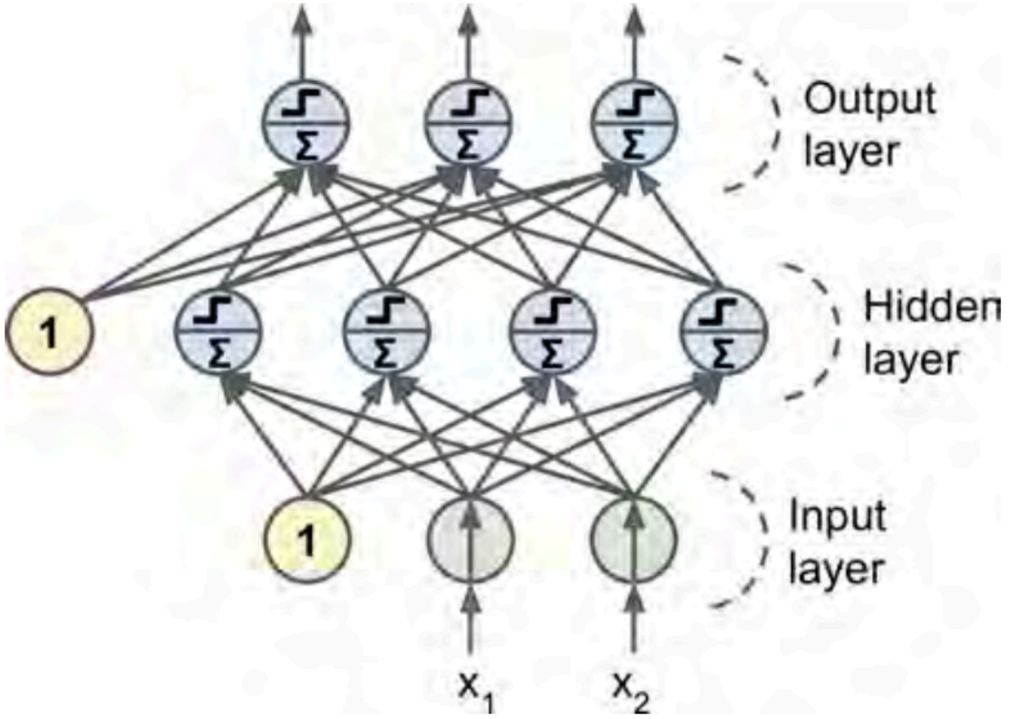
XOR classification problem can be solved by MLP





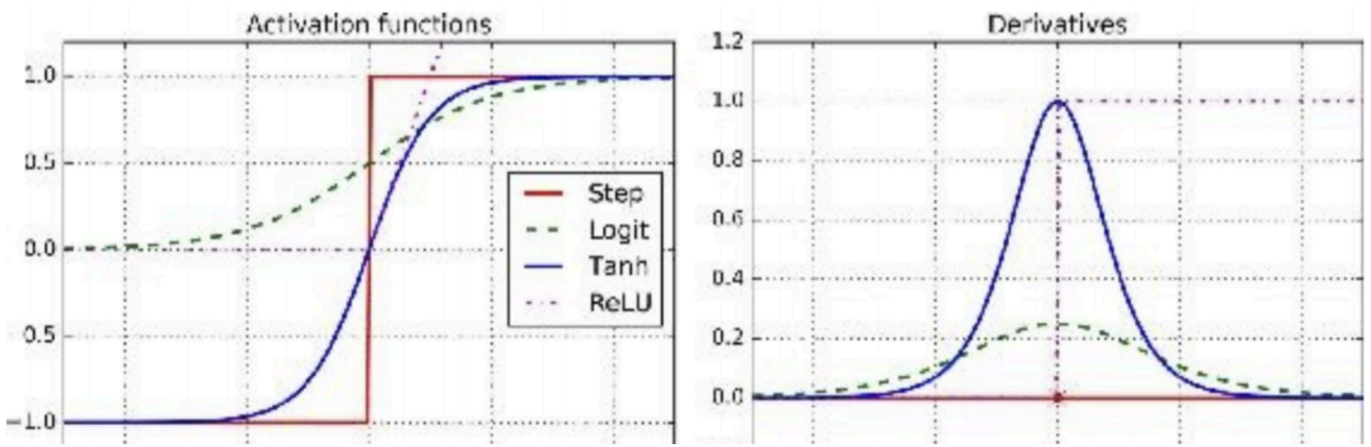
Multi Layer Perceptrons

- Input Layer- passthrough input neurons and bias neuron
- 1or more layers of LTUs - Hidden Layers. Includes bias neurons
- Output Layer - 1 final layer of LTUs. No bias neurons.
- Each layer fully connected to next layers.
- Deep Neural Network- 2 or more hidden layers.



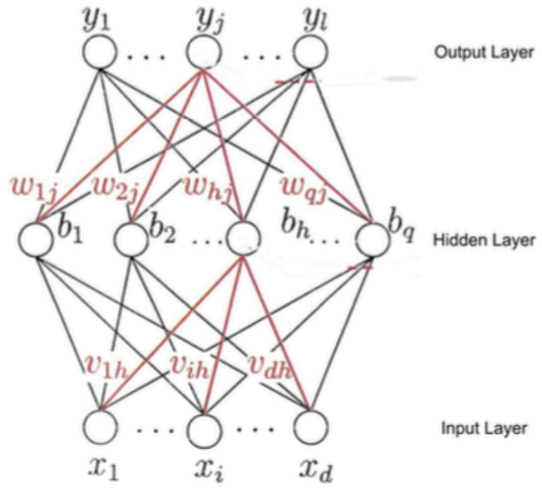
Learning Multi Layer Perceptrons : Error Back Propagation

- Error backpropagation so far is the most successful learning algorithm for training MLP's.
- *Learning Internal Representations by Error Propagation*, Rumelhart, Hinton and Williams, 1986.
- Idea: we start with a fix network, then update edge weights using $v \leftarrow v + \Delta v$, where Δv is a gradient descent of some error objective function.
- Before learning, let's take a look at a few possible activation functions and their derivatives.



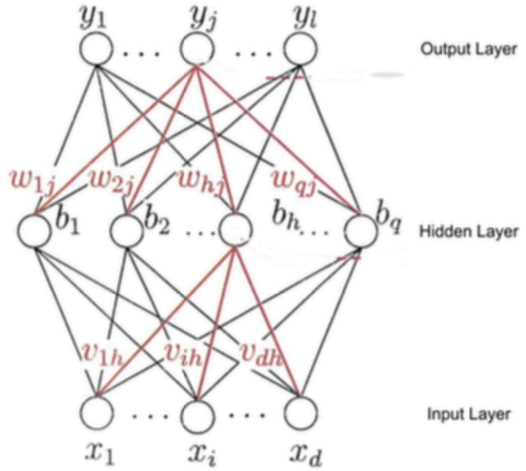
Learning Multi Layer Perceptrons : Error Back Propagation

- Now, I explain the error backpropagation algorithm for the following MLP of d passthrough input neurons, one hidden layer of q LTU's, and output layer of l LTU's.
- I will assume activation functions f at the $l + q$ LTU's all are the sigmoid function.
- Note the bias neuron is gone from the picture, as now the bias is embedded in the LTU's activation function $y_j = f(\sum_i w_i x_i - \theta_j)$.



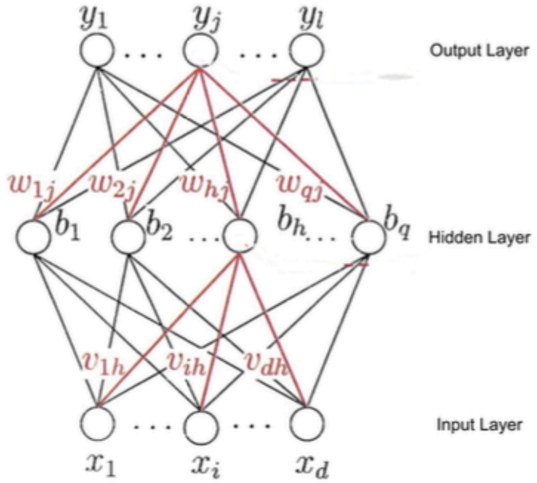
Learning Multi Layer Perceptrons : Error Back Propagation

- So our training set is $\{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_m}, \boldsymbol{y_m})\}$, where $\boldsymbol{x_i} \in \mathbb{R}^d$ and $\boldsymbol{y_i} \in \mathbb{R}^l$.
- We want to have an MLP like below to fit this data set; namely, to compute the $l \cdot q + d \cdot q$ weights and the $l + q$ biases.
- To predict on a new example $\boldsymbol{x}$, we feed it to the input neurons and collect the outputs. The predicted class could be the y_j with the highest score. If you want class probabilities, consider softmax.
- MLP can be used for regression too, when there only is one output neuron.



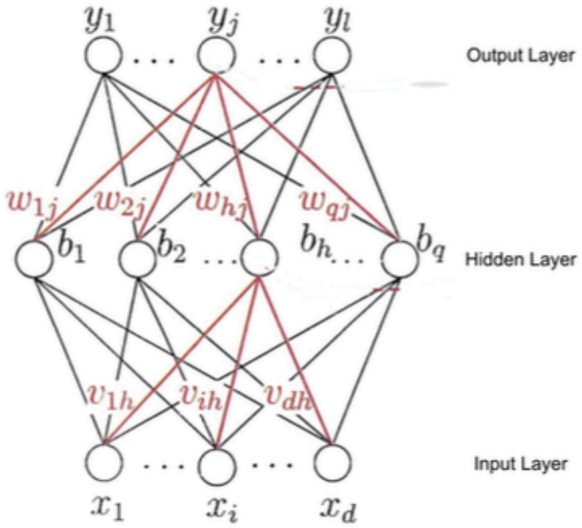
Learning Multi Layer Perceptrons : Error Back Propagation

- I use θ_j to mean the bias in the j-th neuron in the output layer, and γ_h the bias in the h-th neuron in the hidden layer.
- I use $\beta_j = \sum_{h=1}^{q} w_{hj} b_h$ to mean the input to the j-th neuron in the output layer, and $\alpha_h = \sum_{i=1}^{d} v_{ih} x_i$.
- Take a training example $(\boldsymbol{x}, \boldsymbol{y})$, I use $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_l)$ to mean the predicted output, where each $\hat{y}_j = f(\beta_j - \theta_j)$.



Learning Multi Layer Perceptrons : Error Back Propagation

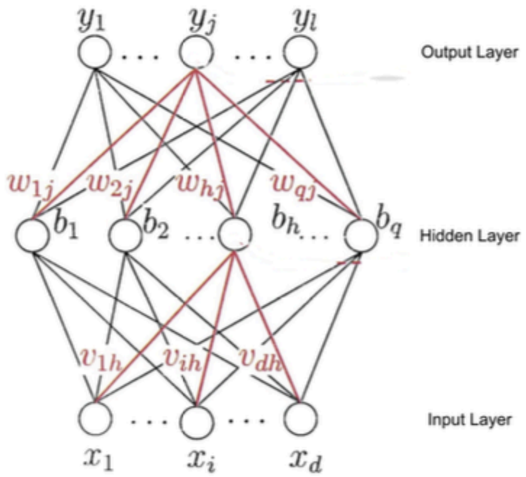
- Error function (objective function): $E = \frac{1}{2}\sum_{j=1}^{l}(\hat{y}_j - y_j)^2$.
- This error function is a composition function of many parameters and it is differentiable, so we can compute the gradient descents to be used to update the weights and biases.



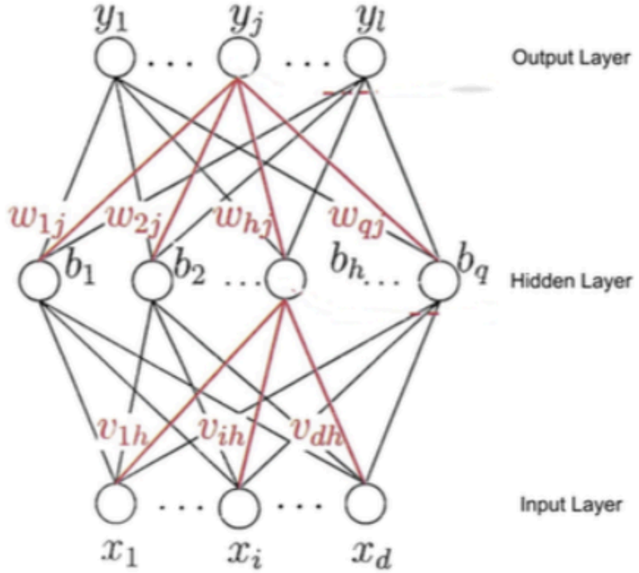
Error Back Propagation Algorithm

Given a training set $D = \{(\boldsymbol{x}, \boldsymbol{y})\}$, and learning rate η, we want to finalize the weights and biases in the MLP.

1. Initialize all the weights and biases in the MLP using random values from interval $[0, 1]$;
2. Repeat the following until some stopping criteron:
   1. For every $(\boldsymbol{x}, \boldsymbol{y}) \in D$, do
      1. Calculate $\hat{\boldsymbol{y}}$;
      2. Calculate gradient descents Δw_{hj} and $\Delta \theta_j$ for the neurons in the output layer;
      3. Calculate gradient descents Δv_{ih} and $\Delta \gamma_h$ for the neurons in the hidden layer;
      4. Update weights w_{hj} and v_{ih}, and biases θ_j and γ_h;

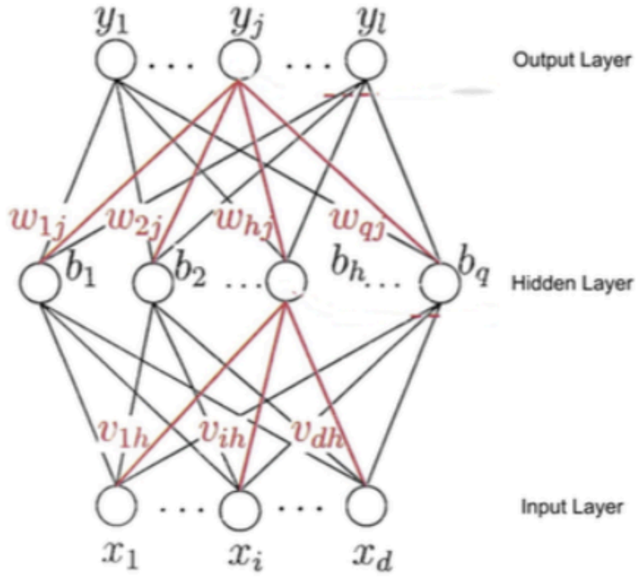


Error Back Propagation Algorithm



1. $\Delta w_{hj} = -\eta \frac{\partial E}{\partial w_{hj}}$
2. $\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}}$
3. We know $\frac{\partial \beta_j}{\partial w_{hj}} = b_h$, for we have $\beta_j = \sum_{h=1}^{q} w_{hj} b_h$
4. We know $\frac{\partial E}{\partial \hat{y}_j} = \hat{y}_j - y_j$, for we have $E = \frac{1}{2} \sum_{j=1}^{l} (\hat{y}_j - y_j)^2$
5. We also know $\frac{\partial \hat{y}_j}{\partial \beta_j} = f'(\beta_j - \theta_j) = \hat{y}_j(1 - \hat{y}_j)$, for we know f is sigmoid
6. Bullets 3, 4 and 5 together can solve Δw_{hj} in bullet 1.
7. Computing $\Delta \theta_j$ is similar.

Error Back Propagation Algorithm



1. $\Delta v_{ih} = -\eta \frac{\partial E}{\partial v_{ih}}$
2. $\frac{\partial E}{\partial v_{ih}} = \frac{\partial E}{\partial b_h} \cdot \frac{\partial b_h}{\partial \alpha_h} \cdot \frac{\partial \alpha_h}{\partial v_{ih}}$
3. (Long story short)
4. $\Delta v_{ih} = \eta x_i b_h (1 - b_h) \sum_{j=1}^{l} w_{hj} g_j$, where $g_j = (y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j)$
5. So we update Δw_{hj} and $\Delta \theta_j$ for the output layer first, then Δv_{ih} and $\Delta \gamma_h$ for the hidden layer.
6. This is why it is called *backpropagation*.

Training Multi Layer Perceptron

Using High Level API - TF.Learn

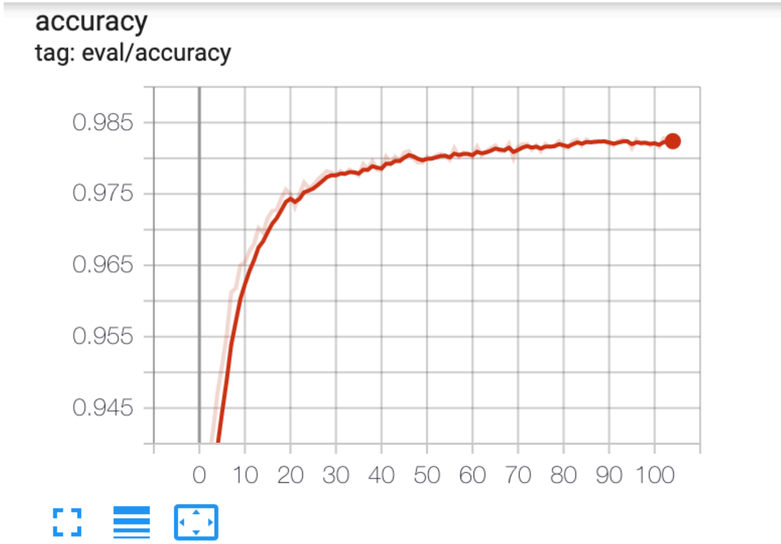
- MNIST data set - 28 X 28 pixel grey scale images of handwritten digits
- Input layer - 28X28=784 input neurons. Pixel intensity represented between 0 and 1. (0-white, 1-black)
- 2 Hidden layers - ReLU activation function
    - hidden layer 1 : 300 neurons, hidden layer 2 : 100 neurons
- Output layer - 10 neurons (10 digits - 0,1,….,8,9)
- Softmax output layer to output estimated class probabilities.
- Cost function : Cross entropy, Accuracy - 98.1%
- *Refer to Jupyter Notebook for implementation*

Training Deep Neural Network

Using TensorFlow's low level Python API

- MNIST data set - 28 X 28 pixel grey scale images of handwritten digits
- Input layer - 28X28=784 input neurons. Pixel intensity represented between 0 and 1. (0-white, 1-black)
- 2 Hidden layers - ReLU activation function
    - hidden layer 1 : 300 neurons, hidden layer 2 : 100 neurons
- Output layer - 10 neurons (10 digits - 0,1,….,8,9)
- Logits and Softmax activation functions
- Softmax output layer to output estimated class probabilities.
- Uses Mini Batch Gradient Descent
- Cost function : Cross entropy

Refer to Jupyter notebook for implementation-Uses dense() instead of neuron_layer()

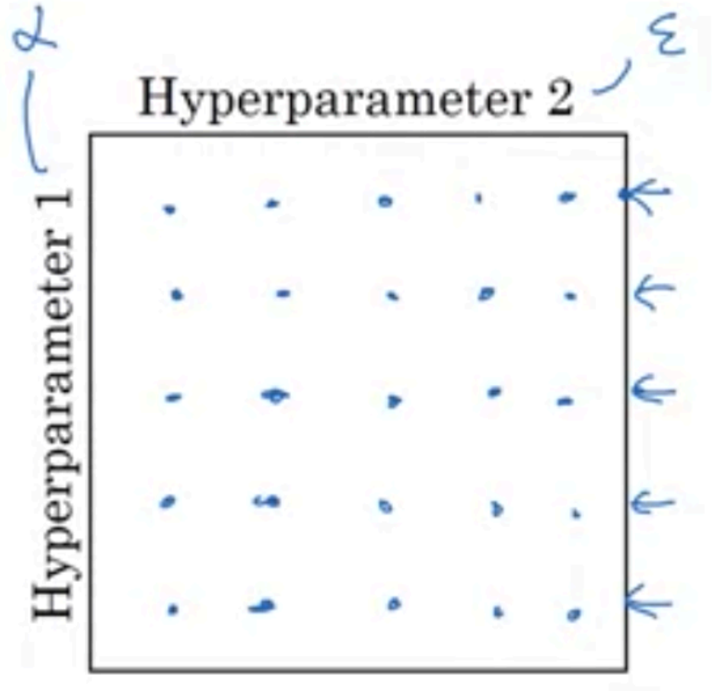


Fine Tuning Neural Network Hyperparameters

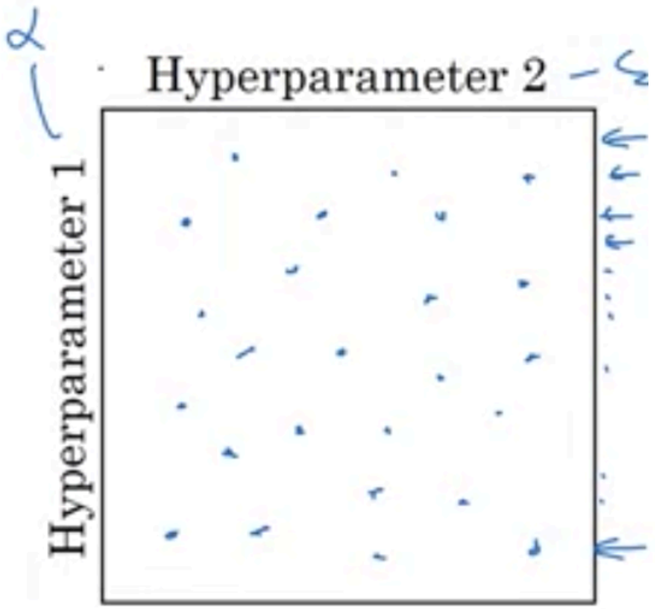
- Number of hidden layers
- Number of neurons per hidden layer
- Activation functions
- To find right set of Hyperparameters, use
    - Grid search with cross validation-tiny part of hyper parameter space unlimited amount of time
    - Randomized search
    - Oscar tool - implements complex algorithms

Right Setting of Hyperparameters

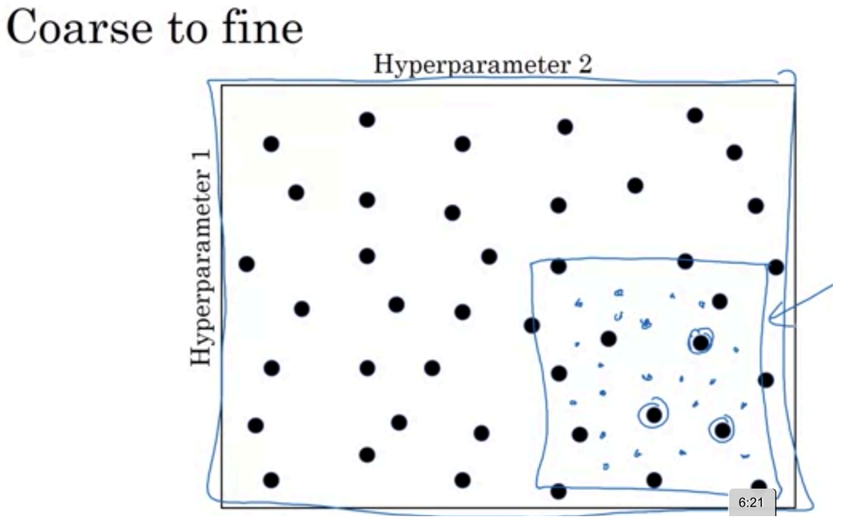
Grid Search



Randomised Search



Coarse to fine



Wednesday, April 28, 2021

Introduction to ANN

UNIT II

Fine-Tuning Neural Network Hyperparameters (OR) Improving deep Neural Networks

Various Hyperparameters

Learning Rate (η)
Optimizers
Momentum in Momentum optimizer (β)
Adam Optimizer (β1,β2, β3 ,ϵ) Best to update weights than using SGD
Number of neurons in hidden layer
Mini-batch size
Number of hidden layers
Learning Rate Decay
Regularization parameter (ƛ)
Number of epochs
Weight Initialization Logic

- **Number of Hidden layers**

  - MLP with a single hidden layer and enough neurons can model complex functions.
  - Deep networks with exponentially fewer neurons can model complex functions with high parameter efficiency than shallow nets.
  - Analogy : To draw a forest using drawing software, you can use copy/paste feature and create one leaf and copy, paste it to create a branch, then copy, paste branch to create a tree, then copy, paste tree to create a forest.

    Similarly in DNN, lower hidden layers model low level structures (ex. line segments of various shapes), intermediate hidden layers combine lower level structures to model intermediate-level structures(ex. squares, circles), highest hidden layers and

the output layer combine intermediate structures to model high level structures(ex. faces)

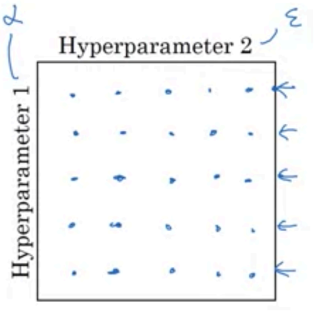
- Hierarchical architecture helps in fast convergence and also generalizes to new data sets. Ex. A model is already trained to recognize faces in pictures. Now to recognize hair styles, reuse the lower layers of the first network .Initialize the weights and biases of the lower layers in new network with the values of weights and biases of the lower layers of the first network.
- Start with 1 - 2 hidden layers and this works for many problems. For complex problems, add up the hidden layers until you get overfitting. Ex. Image classification and speech recognition tasks require dozens or hundreds of layers(may not be fully connected). But such networks are rarely trained from scratch.

- **Number of neurons per hidden layer**
  - Input Layer : Type of Input. Ex: 784 input neurons for MNIST hand written digit classification task
  - Output Layer :Type of output. Ex: 10 output neurons for MNIST hand written digit classification task
  - Hidden layers :Pick a model with more layers and more neurons than actually needed and then use early stopping to prevent from overfitting.
  - Analogy : Instead of wasting time to find the correct size stretch pants, use large stretch pants that will shrink down to the right size.
- **Activation Functions**
  - Type of activation function to use in each layer.
  - Hidden layers - ReLU activation function, faster to compute, does not saturate enlarge input values.
  - Output layer -
    - Softmax activation function - classification tasks

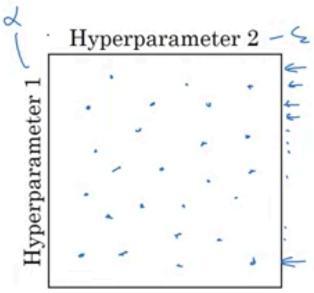
- No activation function - Regression tasks

Right setting of this hyper parameters can be done by using

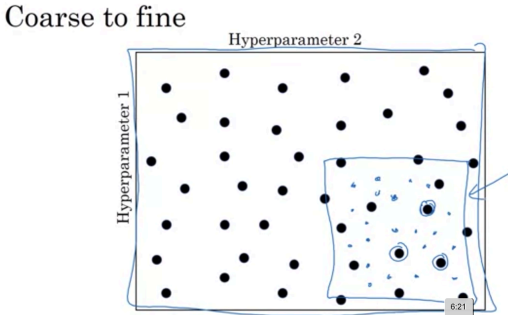
1. Grid Search with cross validation : This approach explores tiny part of hyperparameter space in reasonable amount of time.



2. Random sampling and adequate search



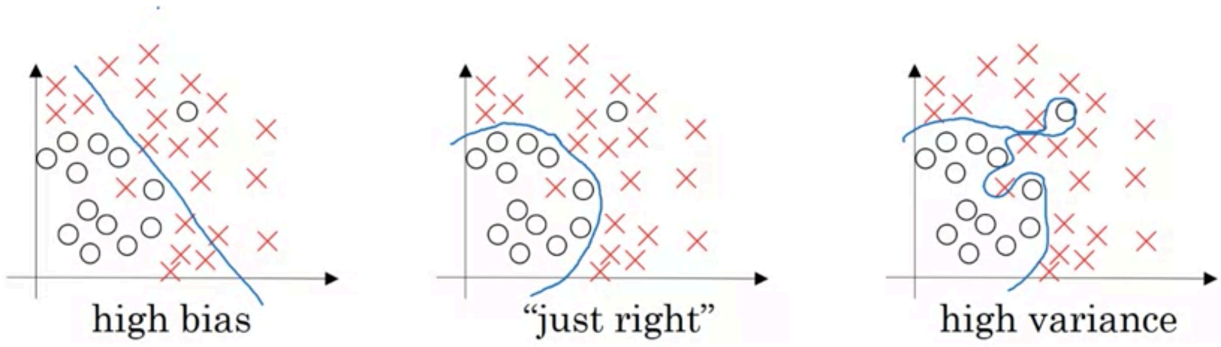
3. Implement coarse to fine search process



4. Oscar tool implements complex algorithms

Training Deep Neural Nets

Bias and Variance



High bias ====> Underfitting
High Variance =====> Overfitting

Training Set Error (%)	1	15	15	0.5
Train Dev Set (Validation Set) Error (%)	11	16	30	1.0
Remarks	**High variance**	**High Bias**	**High Bias and High variance**	**Low bias and Low Variance**
Solution strategies	More data	Bigger network		
	Regiularization	Train longer		
	Reduce number of features			

Bayes error is the theoretical lowest error possible on a task, there can be no lower error rate.

Human error is the empirical lowest error that a human can perform

Techniques to improve the way neural networks learn

1. Better initialization of weights
2. Regularization methods
    1. L1 regularization
    2. L2 regularization
    3. Data augmentation - artificial expansion training data
    4. Early stopping

3. Heuristics to choose good hyper parameters
4. Better cost function - cross entropy

Problems while training a Deep NeuralNets

1. Vanishing & Exploding Gradient Descent Problem
2. Extreme slow training in such a large network
3. Millions of parameters risk in overfitting the training set.

VANISHING/EXPLODING GRADIENT DESCENTS PROBLEM

Gradient is calculated using Backpropagation.

- Gradient of the loss with respect to weights

In a network of *n* hidden layers, *n* derivatives will be multiplied together. If the derivatives are large then the gradient will increase exponentially as we propagate down the model until they eventually explode, and this is what we call the problem of ***exploding gradient***. Alternatively, if the derivatives are small then the gradient will decrease exponentially as we propagate through the model until it eventually vanishes, and this is the ***vanishing gradient*** problem.

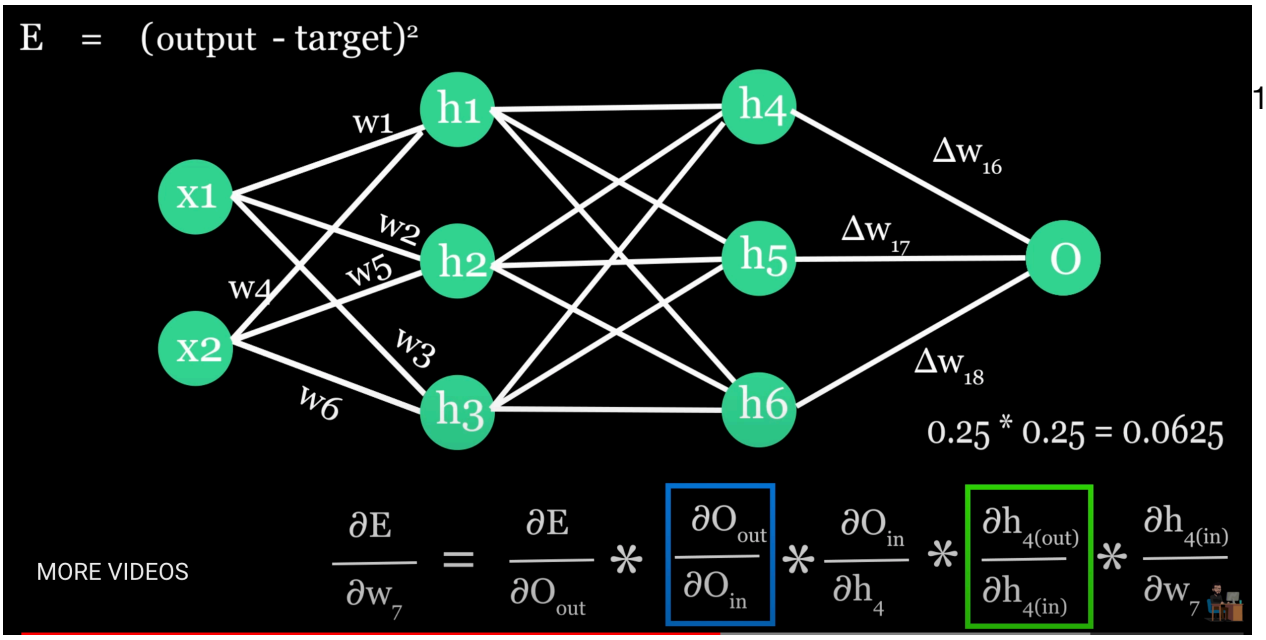
Because of this problem, deep neural networks were abandoned in early 2000s.

Vanishing Gradient Problem

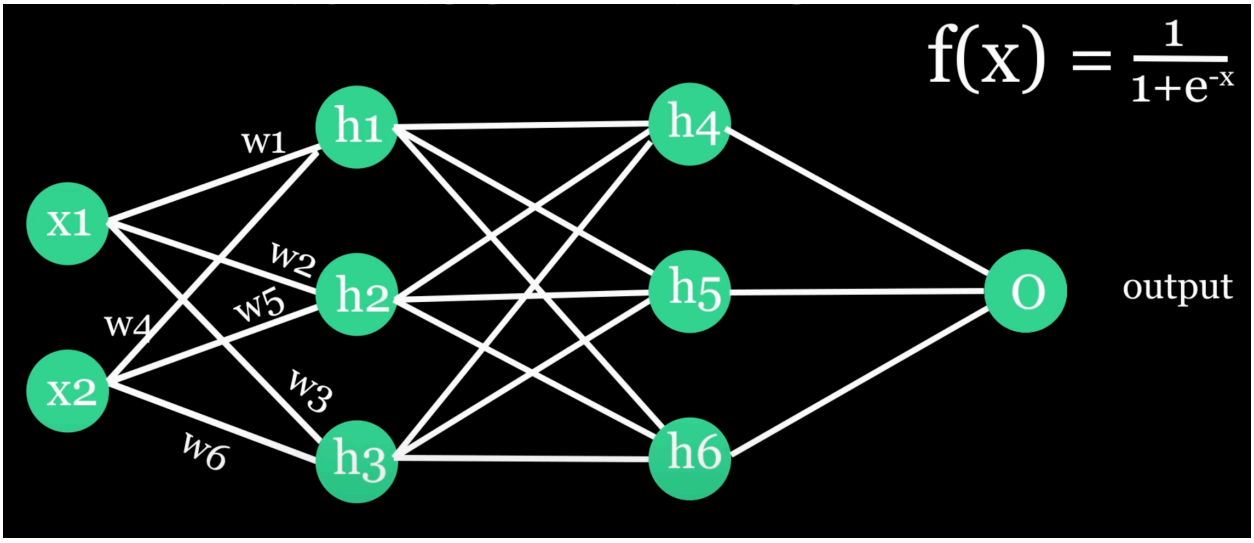
Gradients are calculated from back propagation - unstable gradients

Gradients are used to update the weights

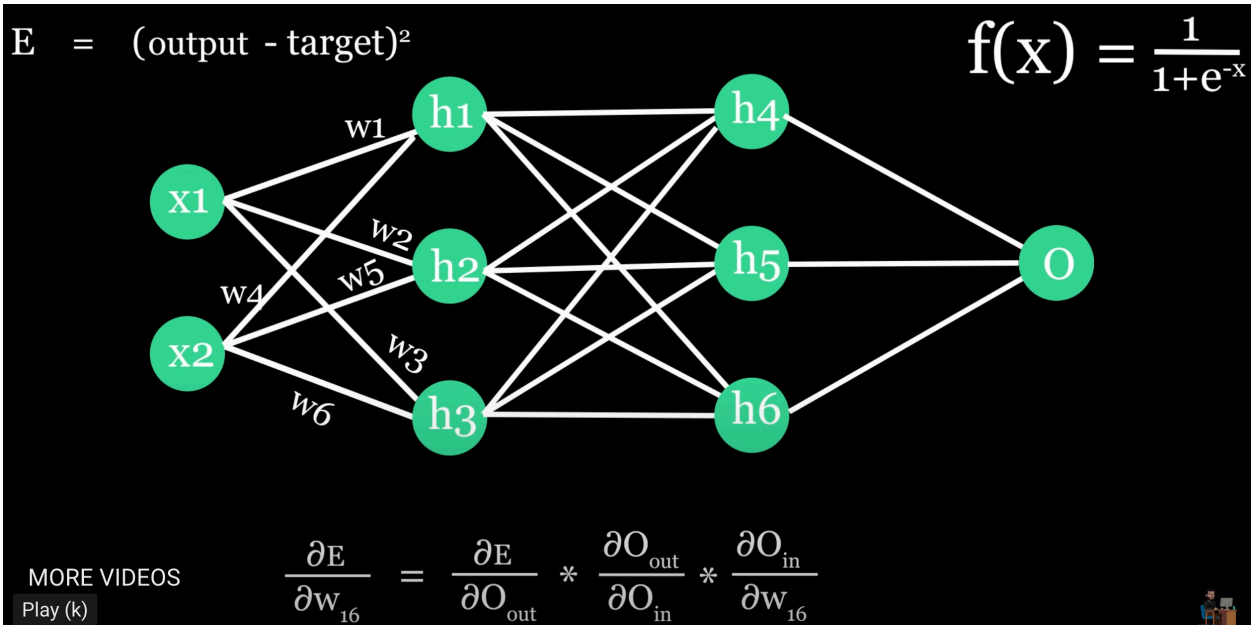
As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train. When *n* hidden layers use an activation like the sigmoid function, *n* small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers. Sometimes the gradient with respect to weights in earlier layers of the network becomes really small, like vanishingly small. Hence, vanishing

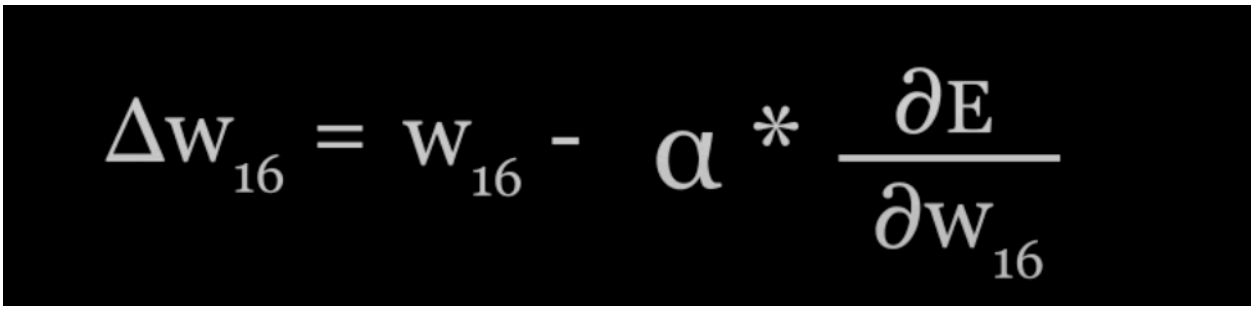


gradient.



Derivative of sigmoid ranges from 0 to 0.25





Watch the following video

https://www.bing.com/videos/search?q=vanishing+gradient+problem&&view=detail&mid=0BCF23AEFC54B8268DE40BCF23AEFC54B8268DE4&&FORM=VRDGAR&ru=%2Fvideos%2Fsearch%3Fq%3Dvanishing%2Bgradient%2Bproblem%26FORM%3DHDRSC3

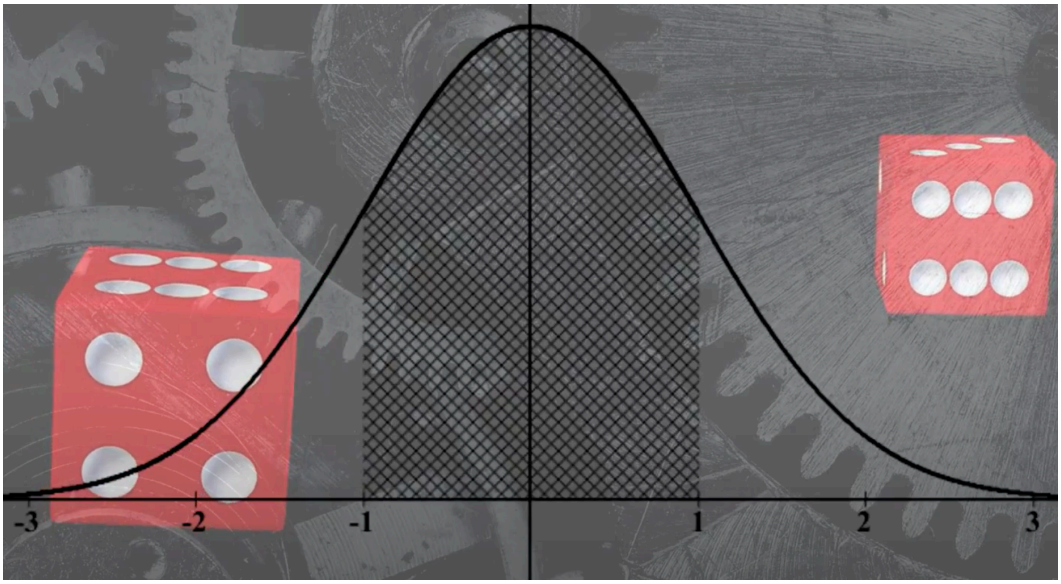
Solutions:

1. Weight Initialization- Xavier and He Initialization
2. Nonsaturating activating functions
3. Batch Normalization
4. Gradient Clipping

1. Weight Initialization

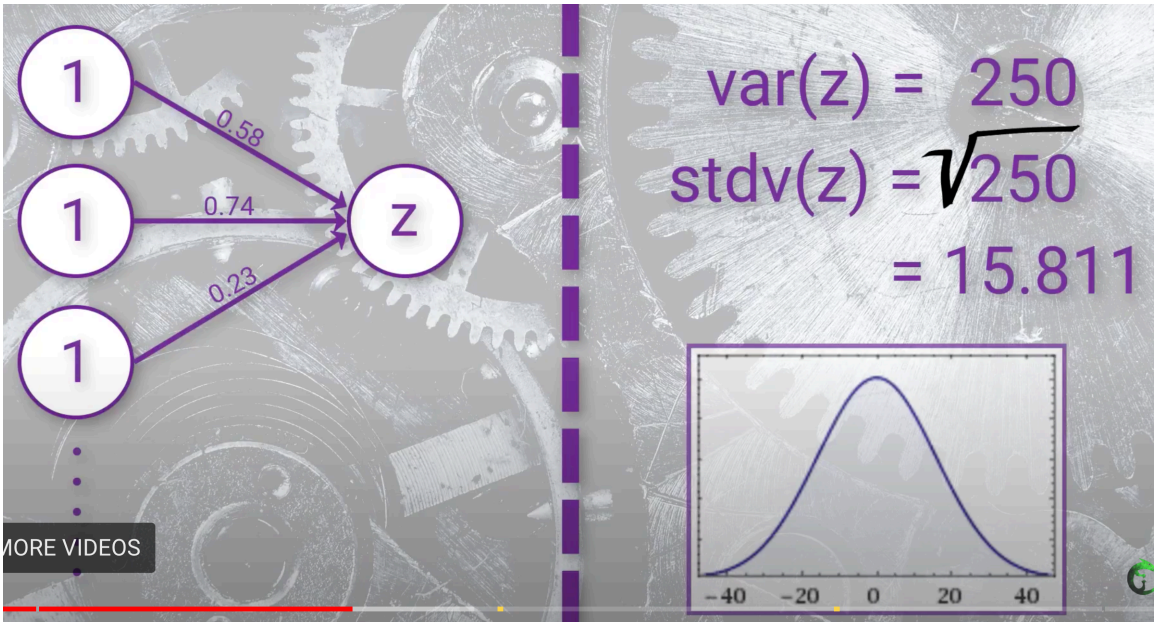
https://deeplizard.com/learn/video/8krd5qKVw-Q

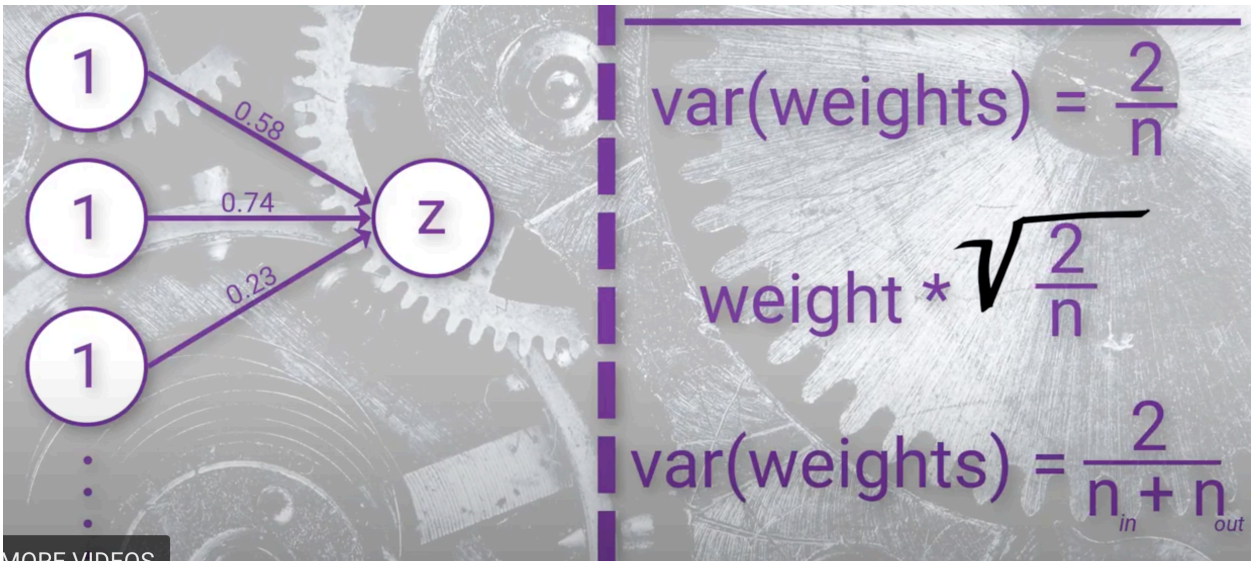
Random distribution of weights. This random values have a normal distribution with mean '0' and standard deviation '1'.



Impact of this random initialization:

Let there are 250 input nodes with values assigned as '1'. Weights are randomly initialized.





As number of nodes increase, std deviation takes significantly larger values compared to 1.Variance for each of our random numbers is 1, so the variance of z is the sum of these 250 numbers, is 250. Std deviation of z is 15.811.

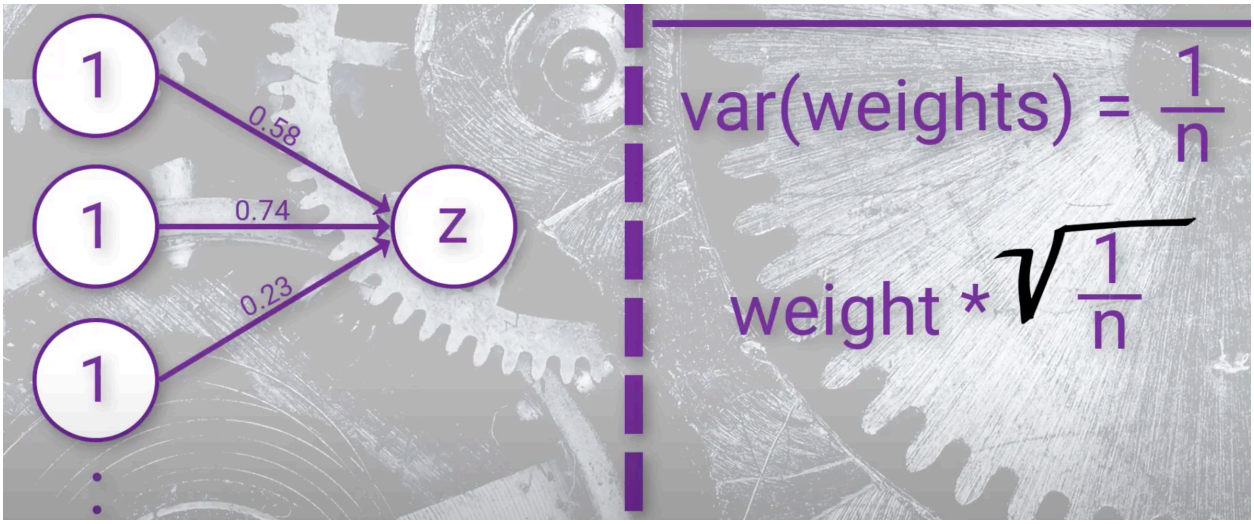
With this larger standard deviation, the value of z is significantly larger or smaller than 1.

. When value of z is passed to the activation function, sigmoid, for example, then most positive inputs, that are significantly larger than 1 will be mapped to the value 1 and most negative inputs will be mapped to 0. So when SGD updates weights to influence the activation output,it will barely make smaller change in the output of activation function.

Thus, the network's ability to learn becomes hindered, and training is stuck running in this slow and inefficient state.

Xavier / Glorot Initialization

Shrink the variance of the weights, and this will shrink the variance of the weighted sum.



For RELU activation function,

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

He Initialization

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

By default fully_connected() function uses Xavier initialization with uniform distribution. This can be changed to He initialization by using variance_scaling_initializer().

2. Nonsaturating Activation functions

ReLU(z)=max(0,z)

ReLU activation function is fast to compute and does not saturate for positive values. But it suffers from the problem of dying ReLUs during training.

When the weights are updated and the weighted sum of the neurons inputs is negative, the neuron outputs a '0'. In such a case, neurons may not comeback to life as the gradient of the ReLU function is '0'.

To solve the problem of dying ReLUs, Leaky ReLU is introduced and is defined as

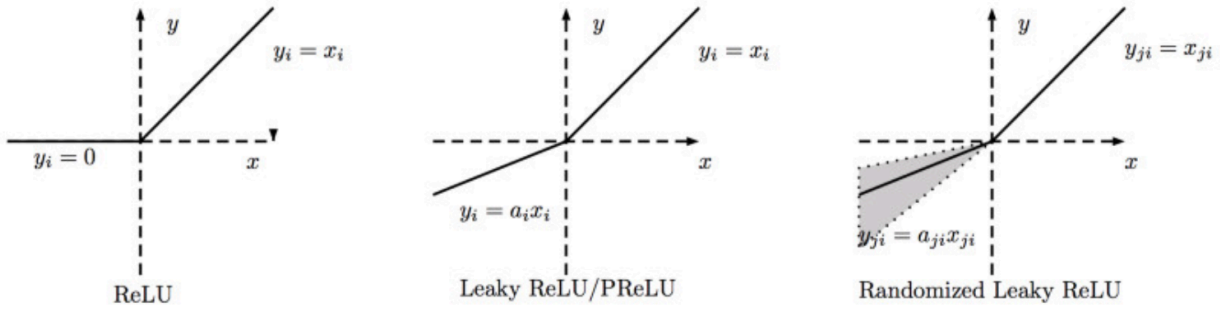
LeakyReLU(z) = max(αz , z) where hyper parameter α = slope of the function when z<0 (how much the function leaks).

α = 0.01 - small leak . Leaky ReLUs never die.

α = 0.2 - huge leak - results in better performance than small leak

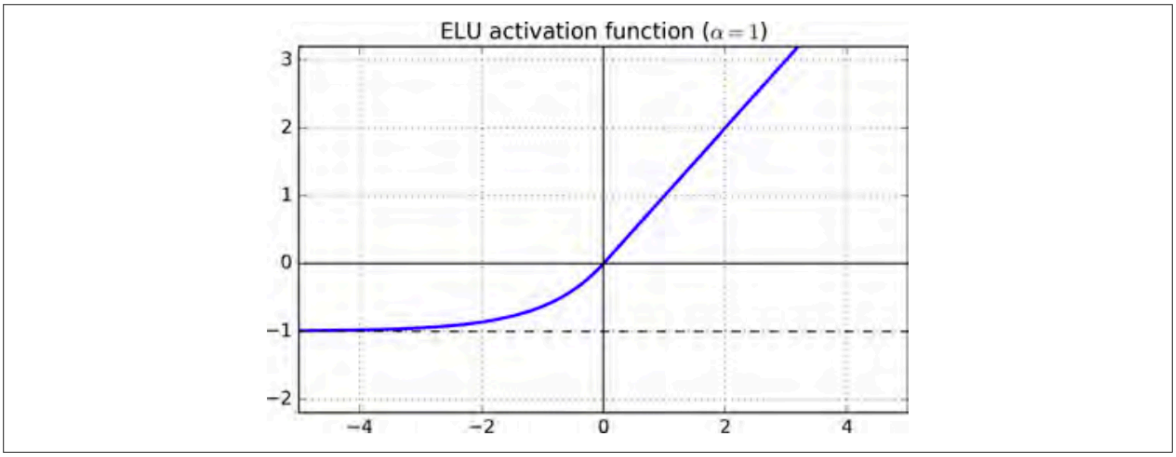
Randomized Leaky ReLU - α is random value in given range - acts as a regularizer.

Parametric Leaky ReLU - α learns during training- performs well on large image datasets, but overfits on smaller datasets.



Exponential Linear Unit, proposed by Djork Arne Clevert outperformed all the ReLU variants and reduced the training time .

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$



Differences of ELU with ReLU

1. ELU takes negative values when z<0.
2. It has non zero gradient when z<0.
3. Function is smooth, even at z=0.
4. It is slower to compute ,but compensated due to fast convergence rate.

Set activation_fn=tf.nn.elu in fully_connected() method.

3. Batch Normalization

https://deeplizard.com/learn/video/dXB-KQYkzNU

Regular Normalization

$$z = \frac{x - mean}{std}$$

A typical normalization process consists of scaling numerical data on a scale from zero to one, and a typical standardization process consists of subtracting the mean of the dataset from each data point, and then dividing that difference by the data set's

standard deviation.This forces the standardized data to take on a mean of zero and a standard deviation of one.

The larger data points in the non-normalized data sets can cause instability in neural networks because large inputs cascade down through the layers in the network, causing imbalanced gradients, and which results in exploding gradient problem. Also non normalized data decreases the training speed.

If one weight is larger than other weights, this causes the corresponding neuron to output large values and this cascading causes instability.

In Batch normalization, after normalizing the output from the activation function, batch norm multiplies this normalized output by some arbitrary parameter and then adds another arbitrary parameter to this resulting product.

Batch Normalization process

Step	Expression	Description
1	$z = \frac{x - mean}{std}$	Normalize output x from activation function.
2	$z * g$	Multiply normalized output z by arbitrary parameter g.
3	$(z * g) + b$	Add arbitrary parameter b to resulting product $(z * g)$.

b, g, mean, std are hyper parameters that are learned for each batch normalized layer.

Advantages:

1. Reduces vanishing gradient problem.
2. Less sensitive to weight initialization.
3. Larger learning rates can be used, speeding up the learning process.
4. Improves accuracy and also acts like a regularizer.

But this process adds complexity to the model and makes slower predictions.

Refer to Jupiter notebook for implementation.

4. Gradient Clipping

Gradient clipping solves exploding gradients problem.

Idea : If the gradient gets too large, we rescale it to keep it small.

If $\|\mathbf{g}\| \geq c$, then

$\mathbf{g} \mapsto c \cdot \mathbf{g}/\|\mathbf{g}\|$

where c is a threshold (hyperparameter), $\mathbf{g}$ is the gradient, and $\|\mathbf{g}\|$ is the norm of $\mathbf{g}$.

Since $\mathbf{g}/\|\mathbf{g}\|$ is a unit vector, after rescaling the new $\mathbf{g}$ will have norm c. If $\|\mathbf{g}\| < c$, then

no rescaling.

TensorFlow Implementation

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

REUSING PRETRAINED NETWORKS

To speed up training the neural networks, pertained models can be used during implementation in the following ways

1. Reusing Pretrained Layers
2. Unsupervised Pretraining
3. Pretraining on an auxiliary task

1. Reusing Pretrained Layers

Not a good idea to build a DNN from scratch.

Find an existing DNN which does a similar task and then reuse the lower layers of the

network — transfer learning.

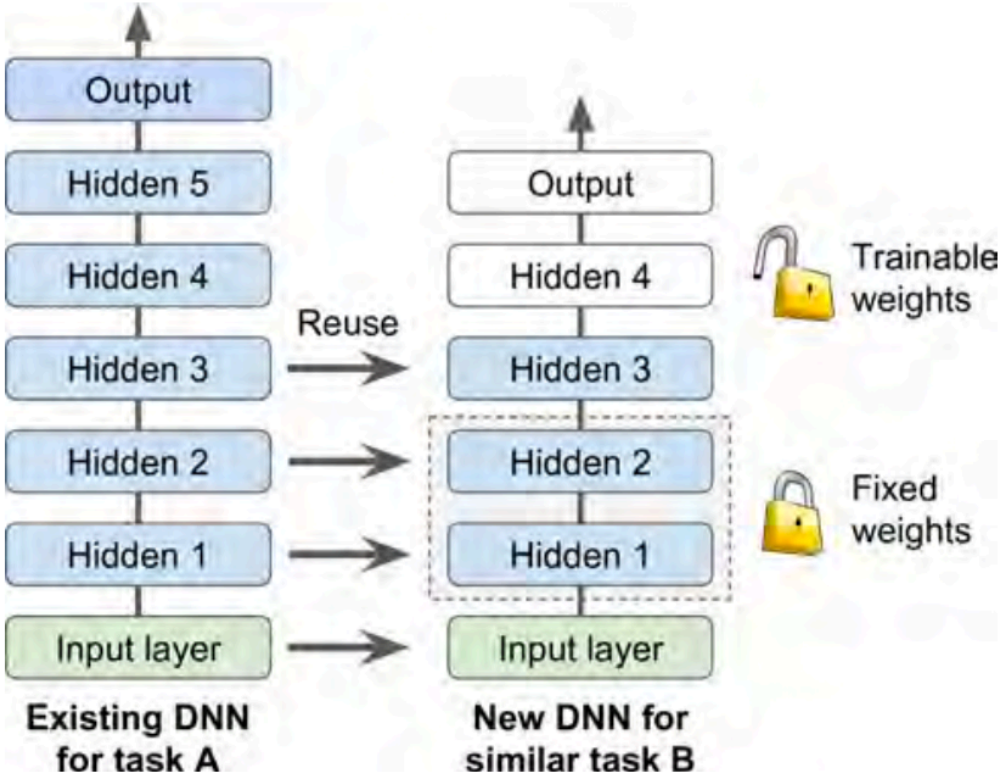
Idea : To build a DNN for taskB, use the lower layers of an existing DNN for a similar task A. This is called transfer learning.

Ex : Task A - A DNN is already trained to classify pictures into 100 different categories.

Task B - To train a DNN to classify specific types of vehicles.

So reuse parts of first network(TaskA)

Reusing Pretrained layers



Transfer learning performs better when the inputs have similar low level features.

Upper hidden layers of the original model are less likely to be useful than the lower layers.

Find the right number flayers to reuse.

Freeze all the reused hidden layers first and see that Gradient descent does not modify the weights in these layers. Check the performance.

Try unfreezing 1 or 2 top hidden layers and let back propagation tweak them and check the performance.

Depending on the amount of trains data, try dropping the top hidden layers and freezing the layers and check the performance. Iterate this process till performance improves.

Refer to Jupiter notebook for implementation

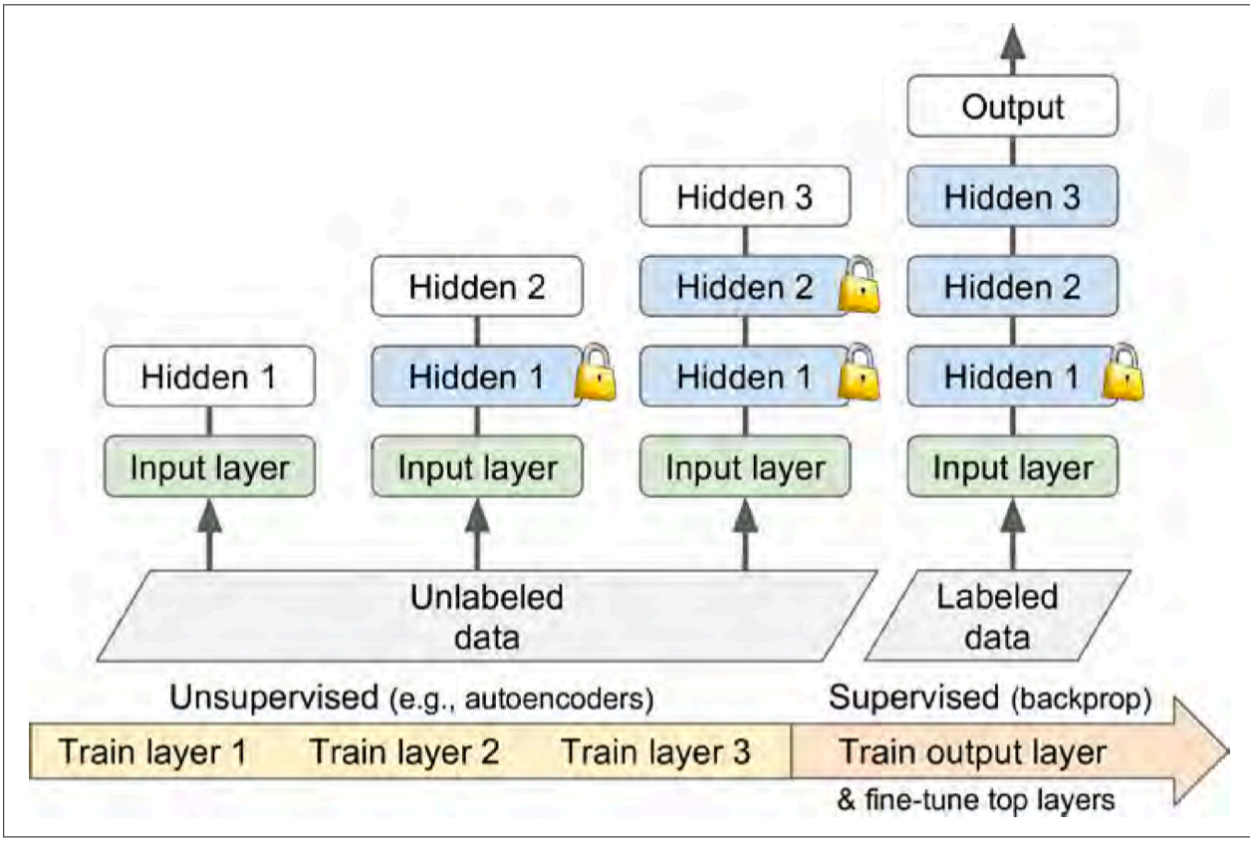
2. Unsupervised pretraining

If a complex task consists of much unlabeled training data, then perform unsupervised pretraining, layer by layer, starting with the lowest layer using unsupervised feature detector algorithm like Restricted Boltzmann Machines(RBM) or auto encoders.

Each layer is trained on the output of previously trained layers. Once all layers are trained, fine tune the network using supervised learning i.e Backpropagation.

Unsupervised pertaining with Autoencoders or GANs(Generative Adversial Networks) is preferable for a complex task with plenty of unlabeled training data and when no similar

model is available to reuse.



3.Pretraining on an Auxiliary Task

Train first neural network on an auxiliary task for which you can obtain labeled training data. Reuse the lower layers of this network for the actual task. The first network will learn feature detectors that are likely to be reusable by second network.

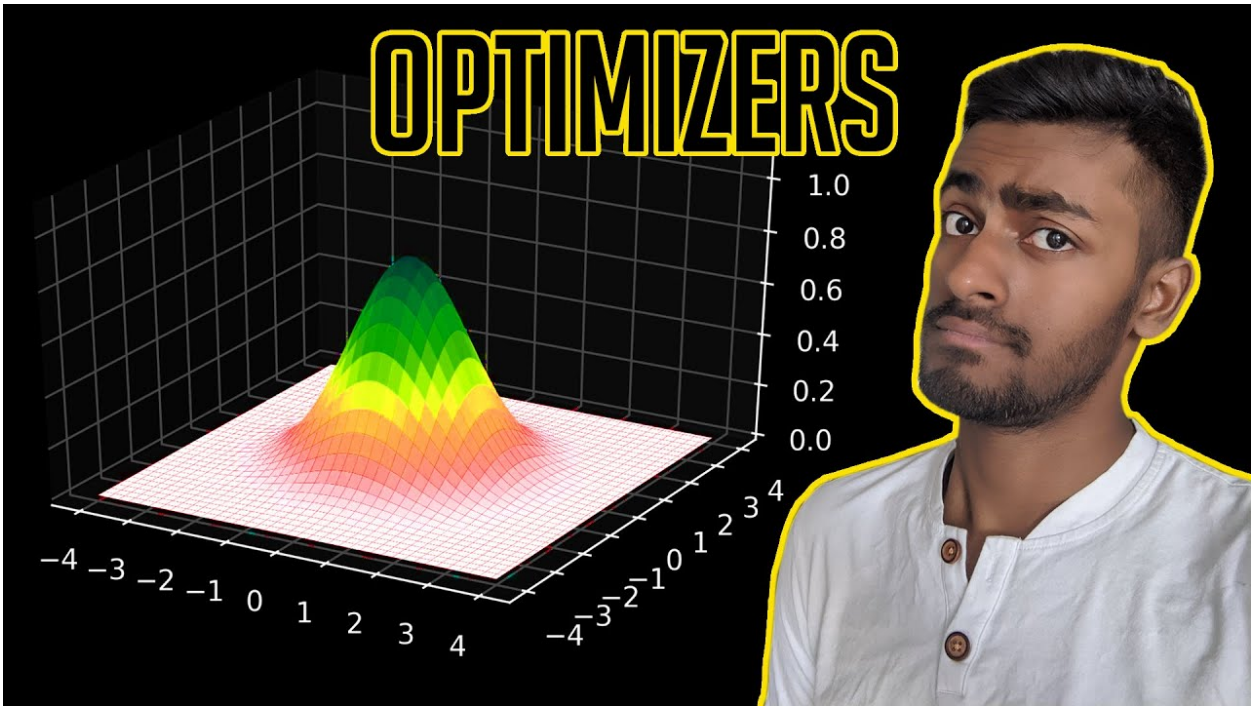
Ex To build a system to recognize faces and if you have smaller labeled data, then collect random pics from internet and detect whether two different pictures feature the same person.

For *natural language processing* (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what

the missing words are (e.g., it should predict that the missing word in the sentence "What ____ you saying?" is probably "are" or "were").

FASTER OPTIMIZERS

Watch the following videos



https://www.coursera.org/lecture/deep-learning-reinforcement-learning/optimizers-TuZPu

Faster Optimizers improve the training speed of large deep neural networks.

Popular ones are

- Momentum Optimization
- Nesterov Accelerated Gradient
- Ada Grad
- RMSProp

- Adam Optimization

1.Momentum Optimization

Idea : Imagine a bowling ball rolling down a gentle slope on a smooth surface:It will start slowly, but will quickly pickup momentum until it reaches terminal velocity.

Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

Gradient is used as an algorithm.

β hyper parameter called momentum is set between 0(high friction) and 1(no friction).Momentum is set to 0.9.If β is 0.9, terminal velocity is ten times the gradient times of the learning rate. Momentum optimization is 10 times that of gradient Descent.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum.

In Keras,

optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)

2. Nesterov Accelerated Gradient

Proposed by Yurii Nesterov in1983.

Faster than momentum optimization.

Idea : Measure the gradient of the cost function not at the local position θ,but slightly ahead in the direction of momentum. Control overshooting by looking ahead at θ+βm

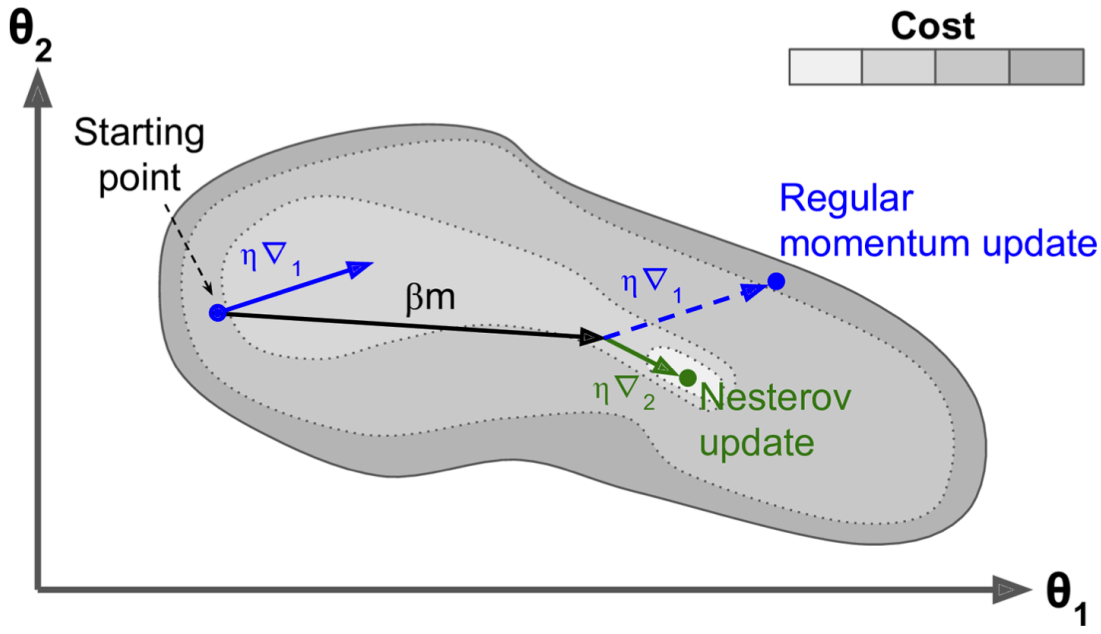


Figure 11-6. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta - \mathbf{m}$

In Keras,

optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)

3. AdaGrad

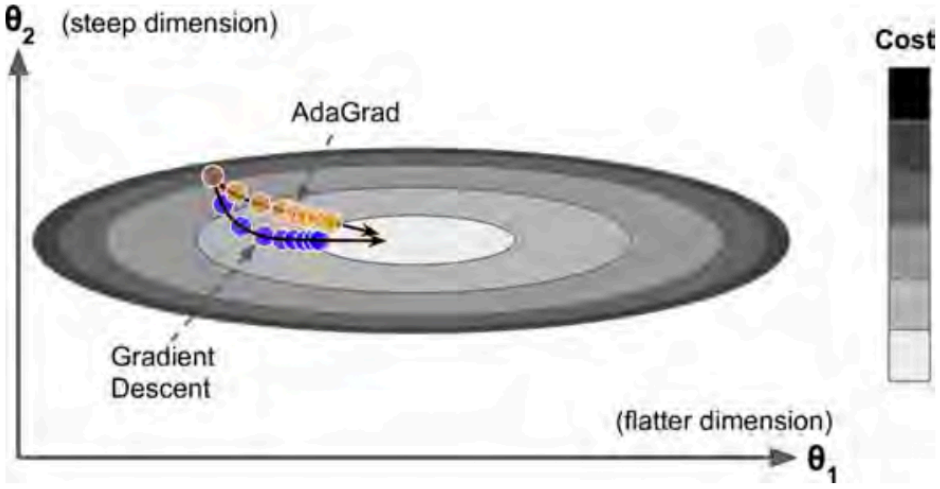
Idea : Scales down the gradient vector along the steepest dimension. Scale the update for each weight separately.

1. Update frequently-updated weights less.
2. Keep running sum of previous updates.

3. Divide new updates by factor of previous sum.

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

ϵ is a soothing term to avoid division by zero.



AdaGrad algorithm inefficient to train neural networks for simpler tasks like Linear Regression. But for quadratic problems,algorithm ends up stopping too early before reaching global optimum.

4. RMSProp

This optimizer accumulates the gradients from the most recent iterations rather than gradients since the beginning of training.

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1 - \beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

It uses exponential decay in the first step. The decay rate, β, hyperparameter is set to 0.9, which works well.

In Keras,

optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)

This optimizer works better than Momentum optimizer, Nesterov Accelerated Gradient, AdaGrad.

5. Adam Optimization

Adam (Adaptive moment estimation) combines Momentum Optimization and RMSProp. This keeps track of exponentially decaying average of past gradients like Momentum optimization and average of past squared gradients like RMSProp.

Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1)\nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - {\beta_1}^T}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - {\beta_2}^T}$
5. $\theta \leftarrow \theta - \eta\, \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

Momentum decay hyperparameter β1 is set to0.9.

Scaling decay hyperparameter β2 is set to0.999

ϵ is set to 10^{-8}

η=0.001

In Keras,

optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)

Sparse Models

All these optimization algorithms produce dense models(most parameters are non zero). To build a fast model at runtime, construct a sparse model.

1. Get rid of the tiny weights by setting them to zero.
2. Apply strong l1 regularization during training, as it pushes the optimizer to zero out as many weights as possible.
3. Apply dual averaging or Follow the Regularised Leader along with with l1 regularization to generate sparse models.

Learning Rate Scheduling

If the learning rate is set too high, training may diverge. If it is too low, training may eventually to converge to the optimum but takes a long time. If set slightly too high, it progresses quickly at first, but sendup dancing around the optimum, never settling down. Ideal learning rate will learn quickly and converge to good solution.

Instead of having constant learning rate, start with high learning rate and reduce it once it stops making fast progress. This way , you can a get a good solution faster than constant learning rate.

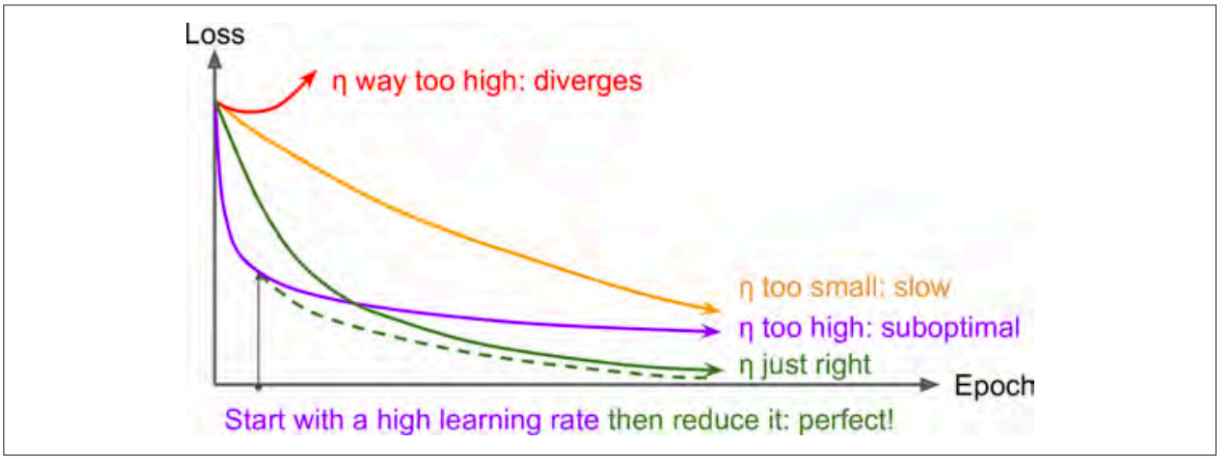


Figure 11-8. Learning curves for various learning rates η

Following are different learning schedules.

Predetermined piecewise constant learning rate
: For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling
: Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
: Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\, 10^{-t/r}$. This works great, but it requires tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

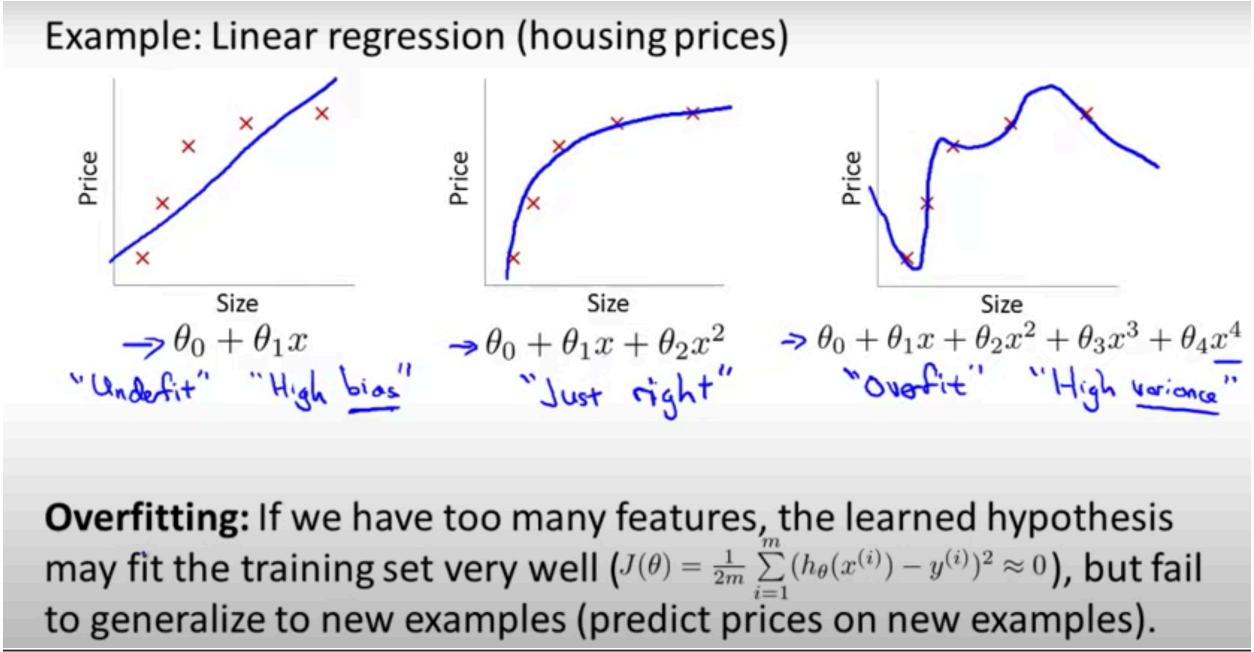
Power scheduling
: Set the learning rate to $\eta(t) = \eta_0\, (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

In Keras,

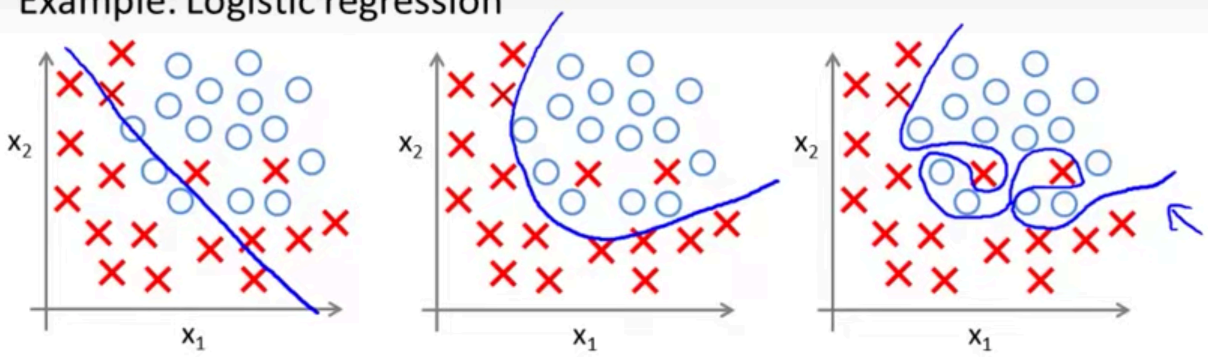
optimizer = keras.optimizers.SGD(lr=0.001, decay=1e-4)

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms, using exponential decay or performance scheduling can considerably speed up convergence.

AVOIDING OVERFITTING THROUGH REGULARIZATION



Example: Logistic regression



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

(g = sigmoid function)

"Underfit"

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$

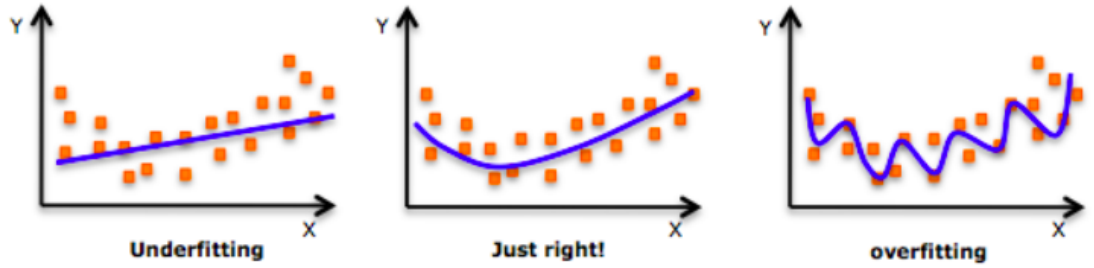
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots)$

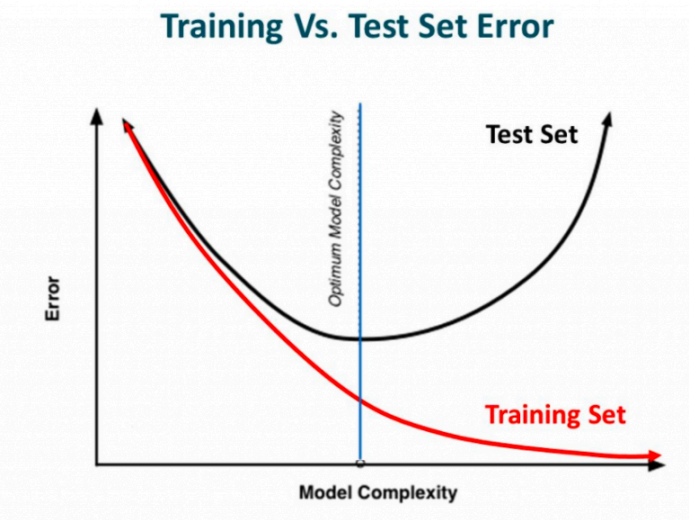
"Overfit"

Addressing overfitting:

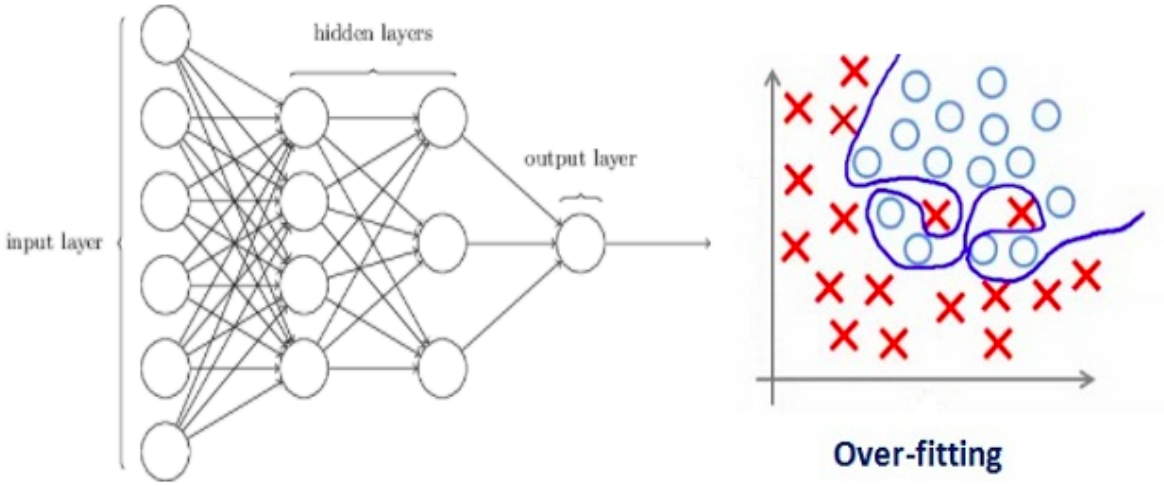
Options:

1. Reduce number of features.
   - — Manually select which features to keep.
   - — Model selection algorithm (later in course).
2. Regularization.
   - — Keep all the features, but reduce magnitude/values of parameters θ_j.
   - — Works well when we have a lot of features, each of which contributes a bit to predicting y.

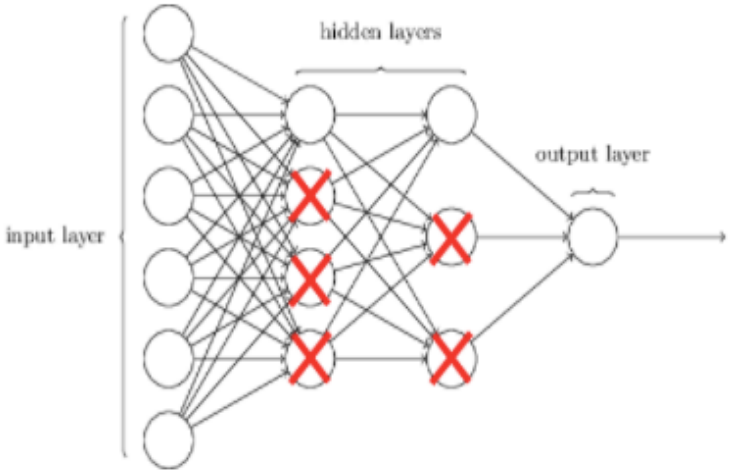


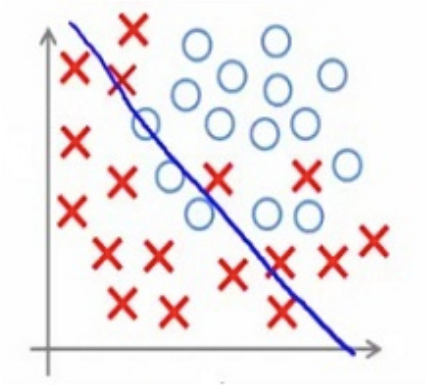


Consider a NN which is overfitting on the training data.

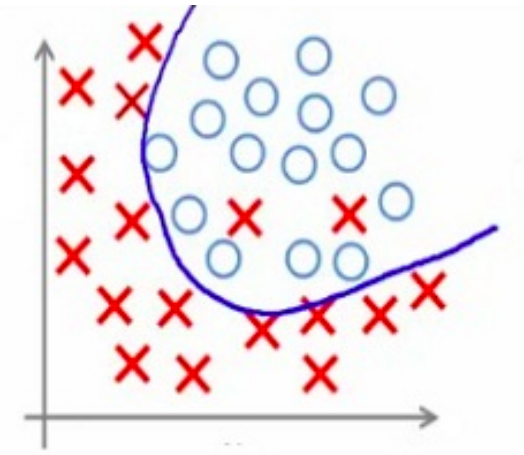


If the regularization coefficient is too high, then some of the weight matrices are nearly equal to zero.





Under-fitting



Appropriate-fitting

1. Early Stopping

A problem with training neural networks is in the choice of the number of training epochs to use.

Too many epochs can lead to overfitting of the training dataset, while too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

This works well in practice, but it performs better when combined with other

regularization techniques.

2. L1 and L2 Regularization

Cost function = Loss (say, binary cross entropy) + Regularization term

A regression model that uses L1 regularization technique is called ***Lasso Regression*** and model which uses L2 is called ***Ridge Regression***.

The key difference between these two is the penalty term.

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

L1 Regularization

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

L2 Regularization

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

For simple linear models, use l_2 regularization to constrain a neural network's connection weights.

For a sparse model , use l_1 regularization(with many weights equal to 0).

Here is how to apply l_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",

kernel_regularizer=keras.regularizers.l2(0.01))
```

similarly for l1 regularizer, use keras.regularizers.l1() and for

both l1 and l2 regularizer, use eras.regularizers.l1_l2().

As same regularizer, same activation function, same initialization strategy is used in all

the respective layers, above code appears repetitively.

So to avoid this, use functors.partial() function

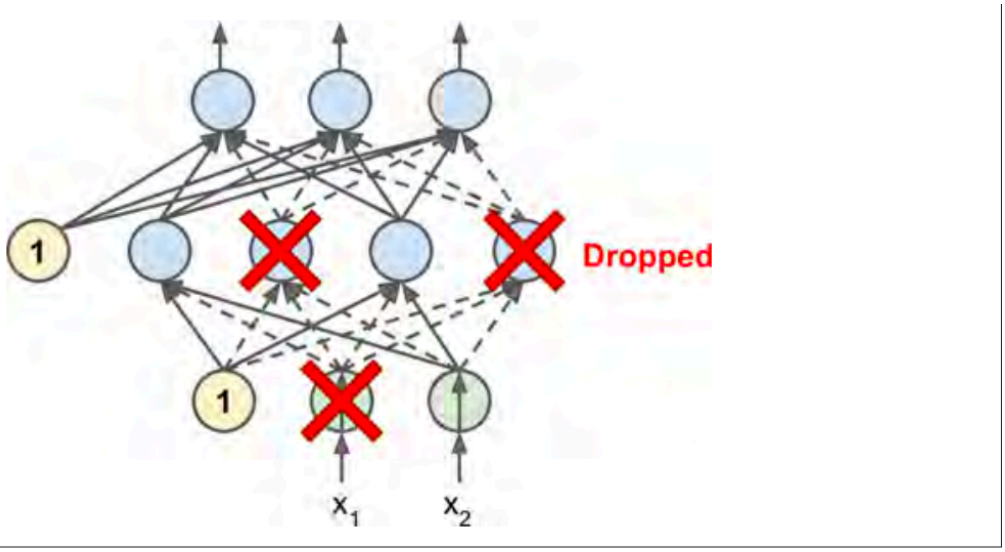
```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",

kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

3. Dropout

This is a popular regularization technique which boosts accuracy by 1-2%.

At every training step, each neuron (including input neurons but excluding output neurons) has a probability p of being temporarily dropped out at this training step and may be active in the next step. This hyper parameter p is called dropout rate and is set closer to 20-30% in RNNs and 40-50% in CNNs.

After training neurons aren't dropped any more.



The following code implements dropout regularization for 3-layer neural network.

If the models overfitting, increase dropout rate.

If the model is underfitting, decrease dropout rate.

Dropout may significantly slowdown the convergence but it will give better results when tuned properly.

Monte Carlo Dropout

4. Max-Norm Regularization

Max norm regularization constrains clipping each neuron's weight vector after each training step to ensure that its norm never exceeds some threshold r.

> *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.
>
> We typically implement this constraint by computing $\|\mathbf{w}\|_2$ after each training step and clipping $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2}$).

Reducing r, increases the amount of regularization and reduces overfitting.

It alleviates vanishing/exploding gradients problem.

TensorFlow does not have off the shelf max-norm regularizer.

```
threshold = 1.0
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

You would then apply this operation after each training step, like so:

```
with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
```

You may wonder how to get access to the `weights` variable of each layer. For this you can simply use a variable scope like this:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")

with tf.variable_scope("hidden1", reuse=True):
    weights1 = tf.get_variable("weights")
```

Alternatively, you can use the root variable scope:

```
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
```

```
[...]

with tf.variable_scope("", default_name="", reuse=True):  # root scope
    weights1 = tf.get_variable("hidden1/weights")
    weights2 = tf.get_variable("hidden2/weights")
```

If you don't know what the name of a variable is, you can either use TensorBoard to find out or simply use the `global_variables()` function and print out all the variable names:

```
for variable in tf.global_variables():
    print(variable.name)
```

Although the preceding solution should work fine, it is a bit messy. A cleaner solution is to create a `max_norm_regularizer()` function and use it just like the earlier `l1_regularizer()` function:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None  # there is no regularization loss term
    return max_norm
```

This function returns a parametrized `max_norm()` function that you can use like any other regularizer:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          weights_regularizer=max_norm_reg)
```

Note that max-norm regularization does not require adding a regularization loss term to your overall loss function, so the `max_norm()` function returns `None`. But you still need to be able to run the `clip_weights` operation after each training step, so you need to be able to get a handle on it. This is why the `max_norm()` function adds the `clip_weights` node to a collection of max-norm clipping operations. You need to fetch these clipping operations and run them after each training step:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    [...]
    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)
```

5. Data Augmentation

Data Augmentation consists of generating new realistic training instances from existing ones, artificially boosting the size of training set. This reduces overfitting.

Ex , if you want to classify pictures of mushrooms, you may slightly shift, rotate, resize every picture in the training set by various amounts and add resulting pictures to the training set. Thenthe resulting model becomes more tolerant to the size, orientation, position of mushrooms in the pictures.

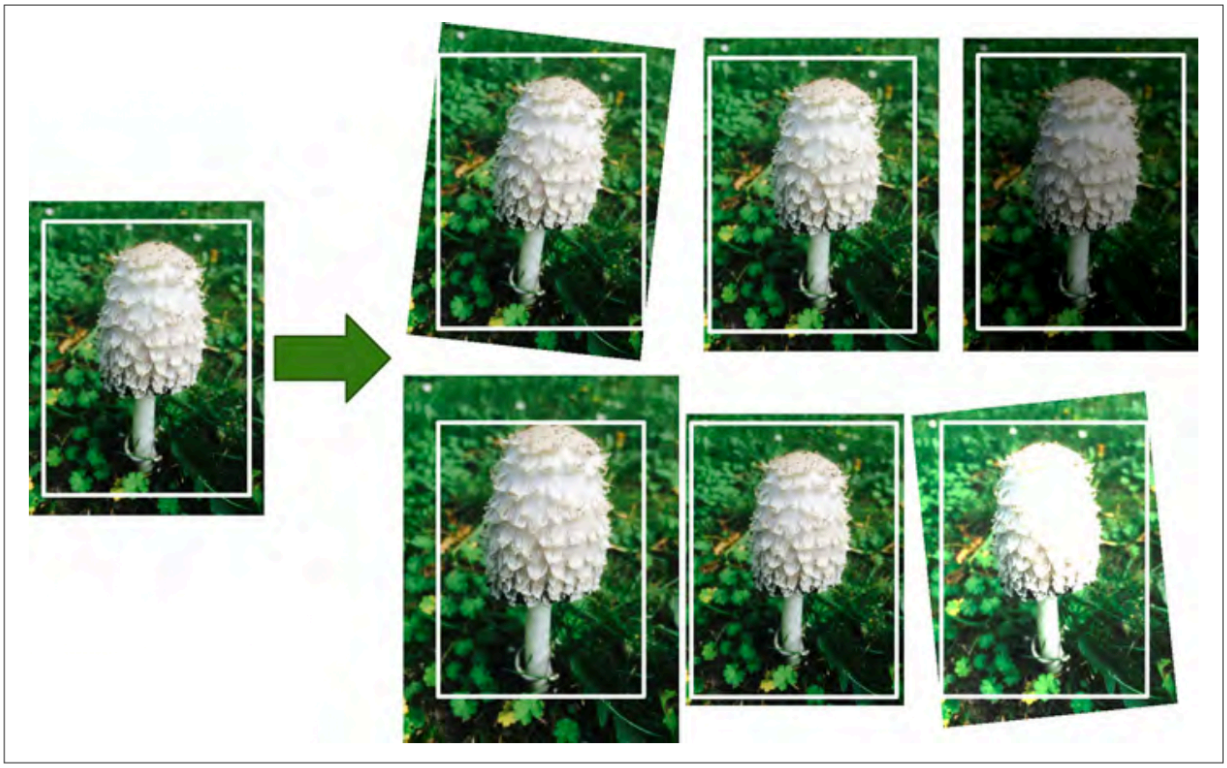


Figure 11-10. Generating new training instances from existing ones

It is preferable to generate new training instances on the fly during training so as to save storage space and network bandwidth.

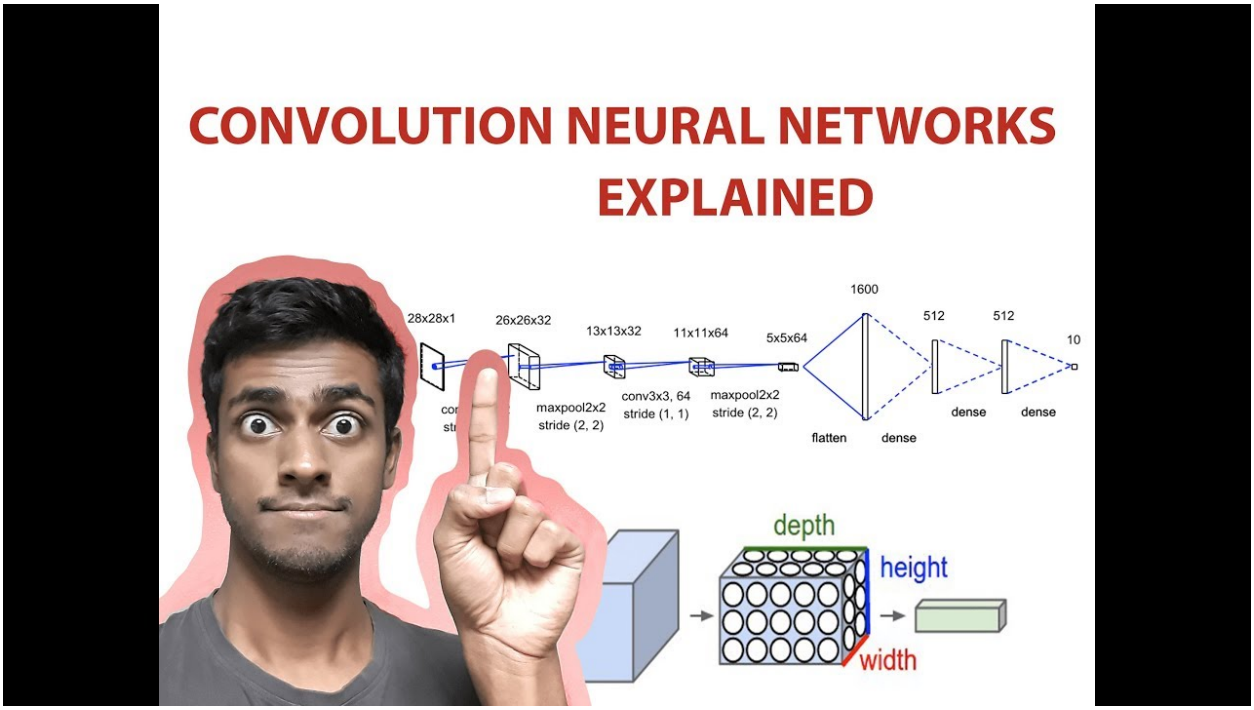
TensorFlow offers several image manipulation operations like transposing, rotating, resizing, flipping, cropping, adjusting brightness, contrast, saturation, hue.

Steps involved in training a DNN

1. Randomly seed weights
2. Fetch a batch data
3. Forward pass
4. Compute Cost
5. Backward pass
6. Update weights

Convolutional Neural Networks

Watch the following video



Convolutional Neural Network(CNN)

Also called Convolutional Networks, are a special kind of neural networks, that has a known grid like topology.

Ex: 1. Timeseries data —1 D grid with samples at regular time intervals

2. Image data - 2D grid of pixels

This CNN employs a mathematical, specialized kind of linear operation called convolution. CNNs are neural networks that use convolution operation in place of general matrix multiplication in at least one of their layers. CNNs emerged from the study of humans visual cortex. CNNs are successful in visual perception, voice recognition, Natural Language Processing tasks, etc.

CNNs use three basic ideas.

1. Local receptive field
2. Shared weights
3. Pooling

Architecture of Visual Cortex

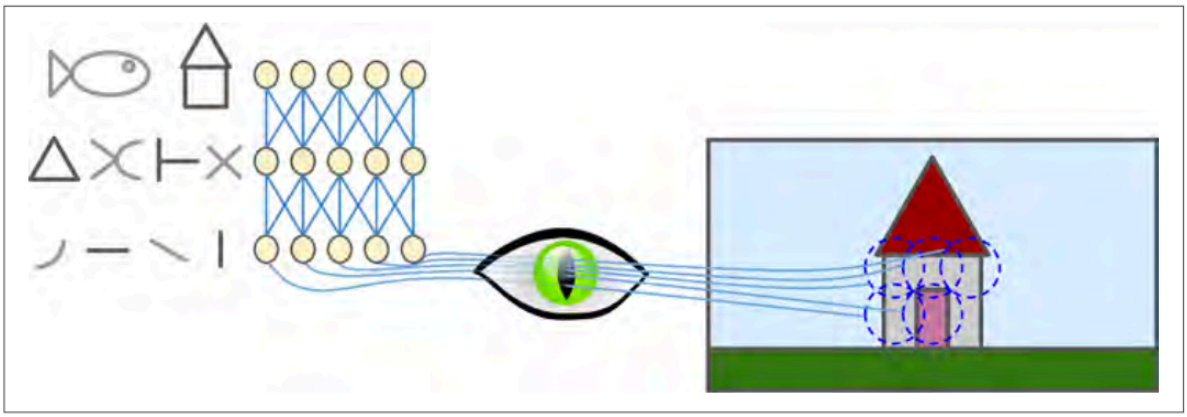
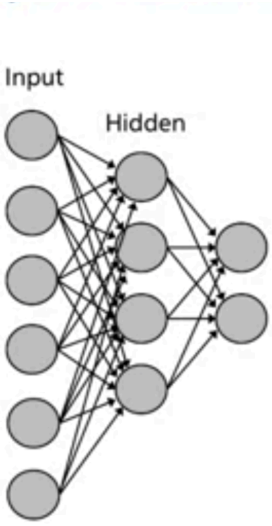


Figure 13-1. Local receptive fields in the visual cortex

Many neurons in the visual cortex have a small local receptive field. Local receptive fields of five neurons are indicated by five dashed circles. Receptive fields of different neurons may overlap and together they tile the whole visual field. Some neurons react to images of only horizontal lines, while others to vertical lines or other orientations.Some neurons may have larger receptive fields and react to more complex patterns that are combinations of lower level patterns. So higher level neurons are based on the outputs of the neighboring lower level neurons.

Why Convolution?



MNIST dataset: 28 x28 pixels (784 pixels)
First layer weights: 784x128 ~ 100K parameters

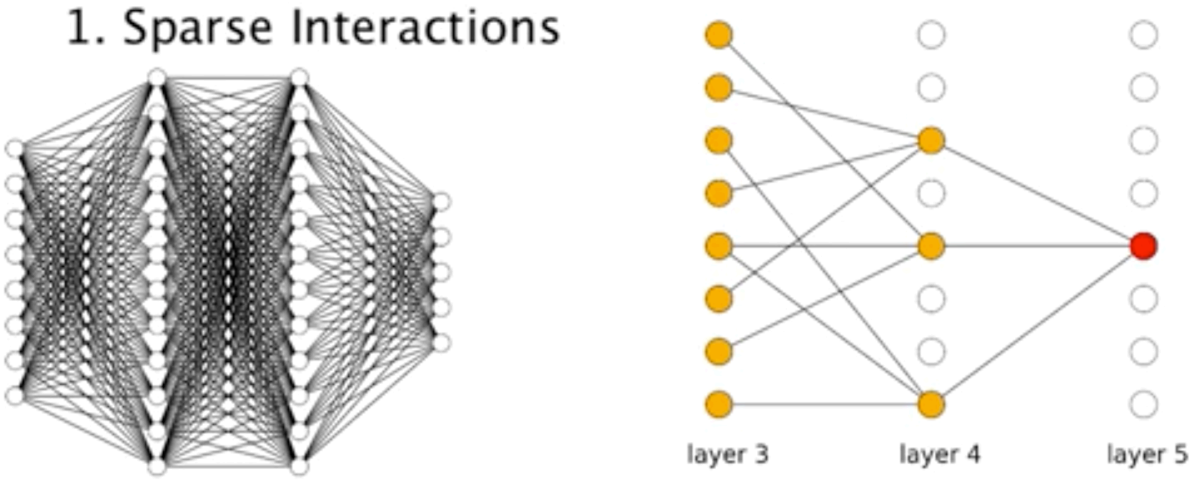
Typical Image: 256 x 256 (56,000 pixels)
First layer weights: 8M parameters !

Too many parameters, unscalable to real images

Layers in CNN

1. Convolutional Layer
2. ActivationLayer
3. Pooling Layer
4. Fully Connected Layer

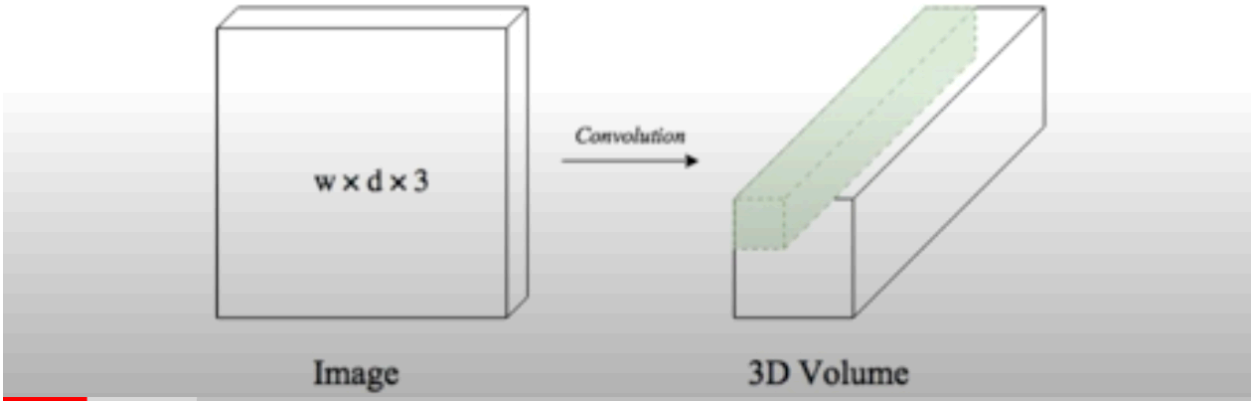
1. Convolutional Layer



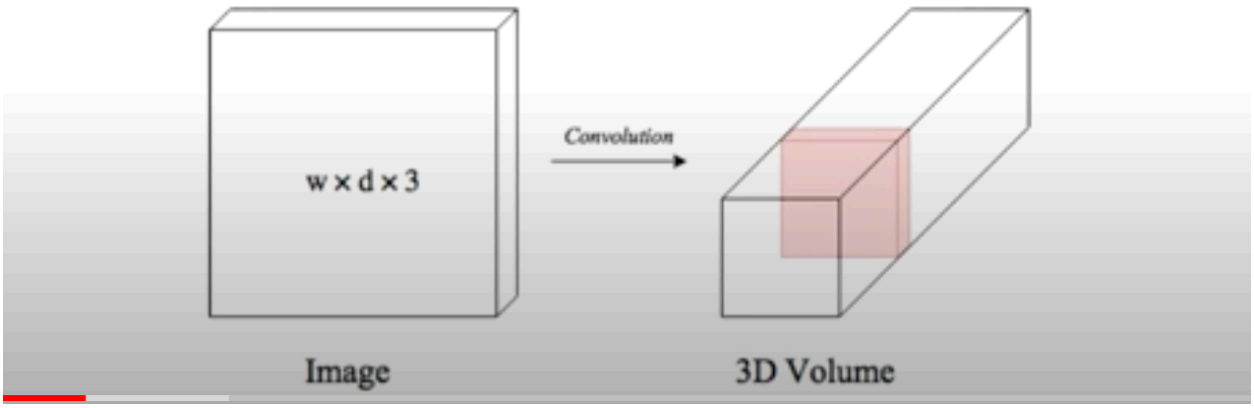
For the hand written digit classification on MNIST dataset,

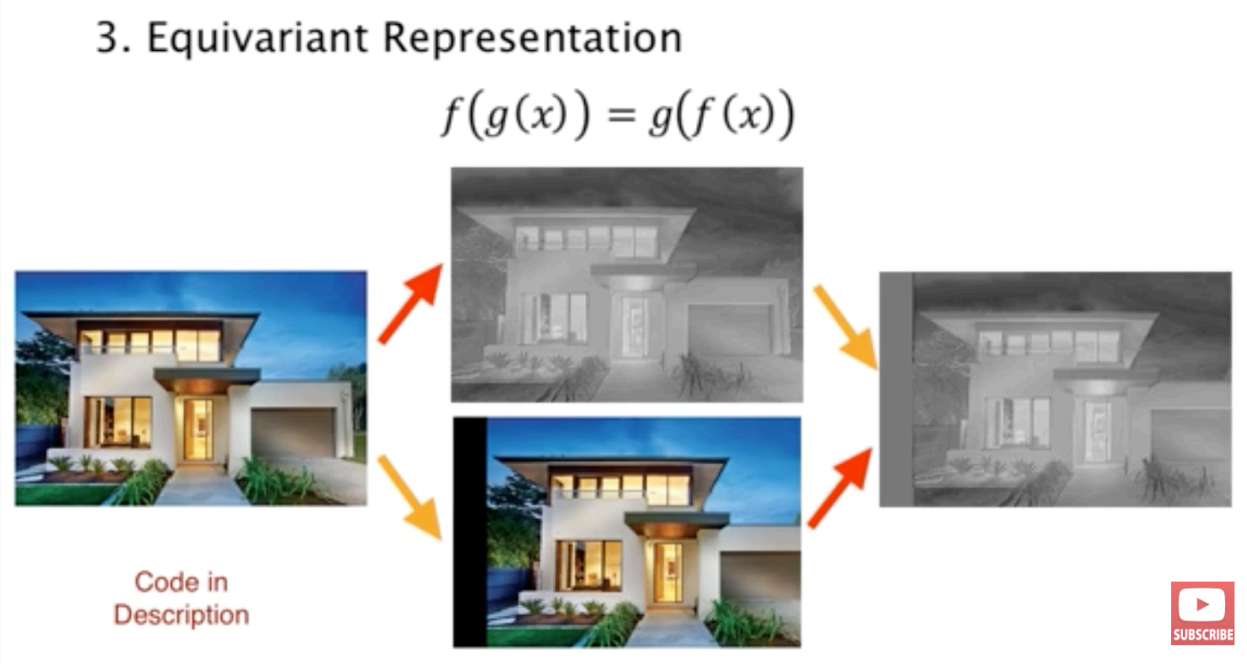
in a fully connected network ,input layer is represented as 28x28=784 vertical line of input neurons where as in Convolutional net, it is represented as 28x28 square of neurons, whose values correspond to pixels intensities. Each neuron in the first hidden layer will be connected to only a small region of the input neurons. Ex a 5x5 region corresponding to 25 pixels. Region in the input image is called the local receptive field for the hidden neurons.

2. Parameter Sharing



2. Parameter Sharing





Convolutional layers are the major building blocks used in convolutional neural networks.

Convolution Operation

Convolution is an operation on two functions of real valued arguments.

Ex Suppose we are tracking the location of a spaceship with laser sensor. x(t) is the output of the laser sensor which specifies the position of spaceship at time t. Suppose the laser sensor is noisy. To get less noisy estimate space ship position , we average .together several measurements. Let w(a) be weighing function with a as the age

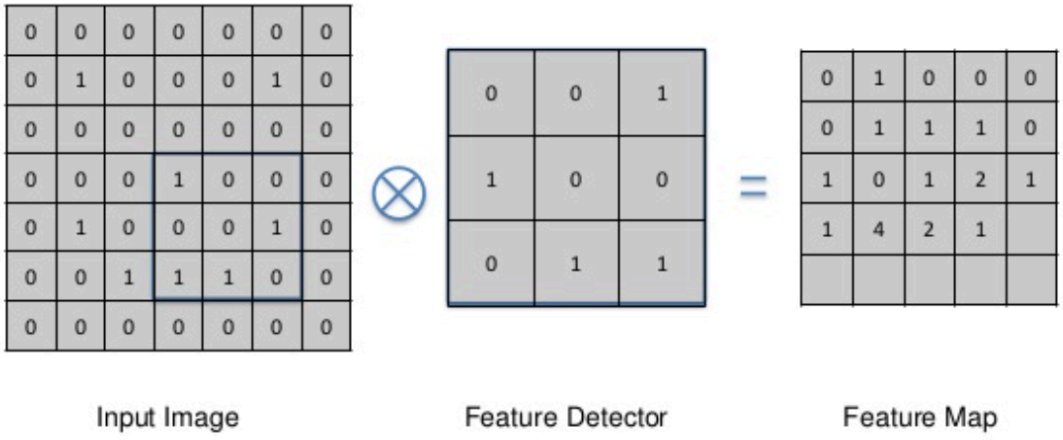
of measurement.

$$s(t) = \int x(a)w(t-a)da$$

This operation is called **convolution**. The convolution operation is denoted with an asterisk:

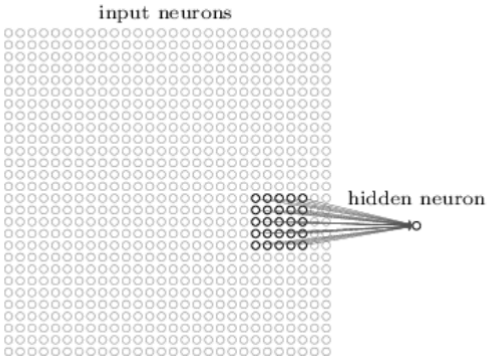
$$s(t) = (x * w)(t)$$

The first argument, x is called input and the second argument, w is called the kernel filter. The output is called feature map

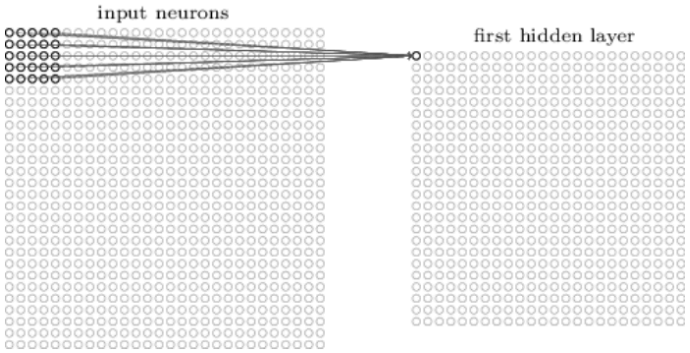


A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected

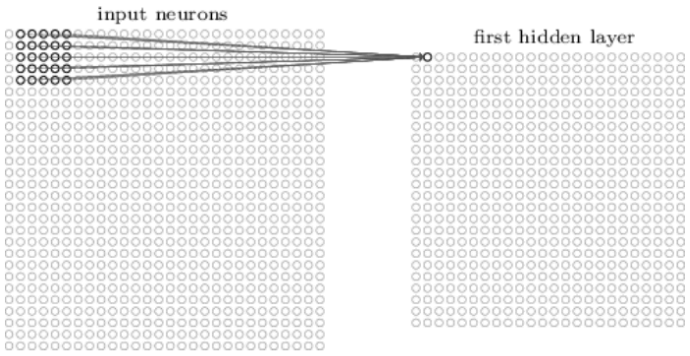
feature in an input, such as an image.



Each connection learns a weight and the hidden neuron learns overall bias. Each hidden neuron is trying to analyze its own local receptive field.



Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:



Slide the local receptive field across the entire image.

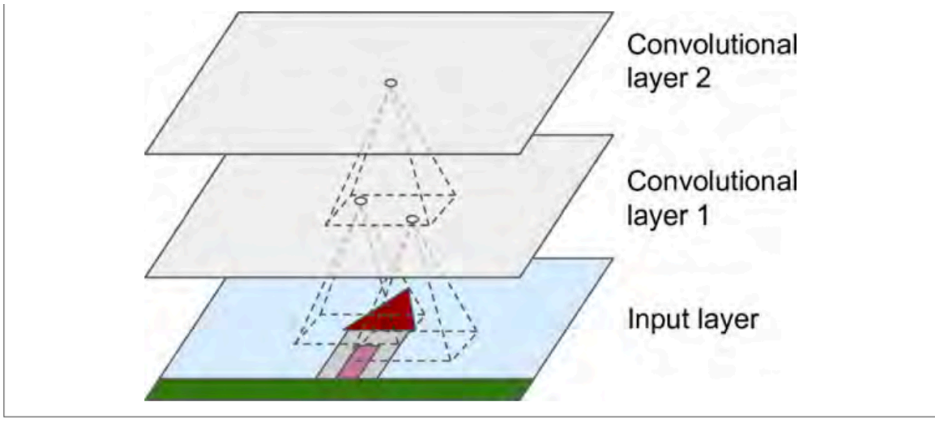


Figure 13-2. CNN layers with rectangular local receptive fields

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 13-3). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

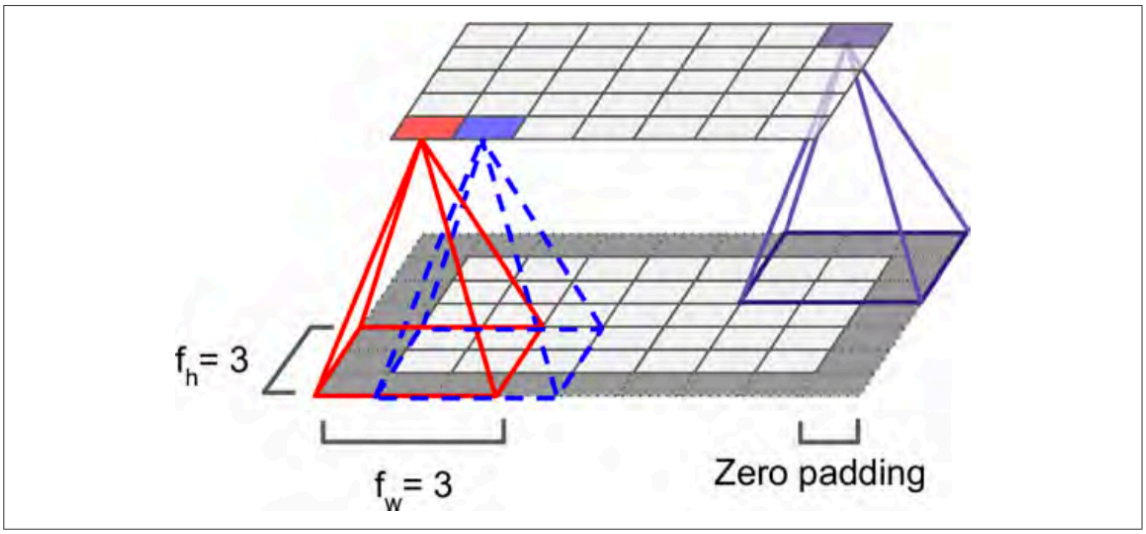
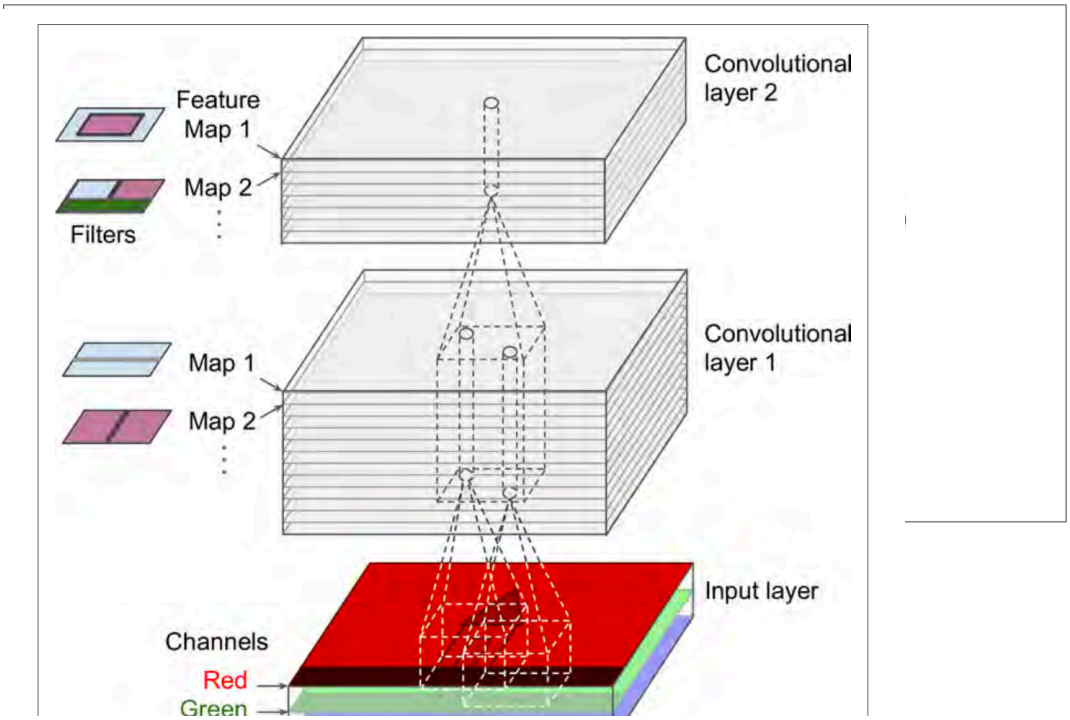


Figure 13-3. Connections between layers and zero padding

The distance between two consecutive receptive fields is called the stride.

neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

S



Equation 13-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

If the hidden neuron has a bias and a 5x5 weights connected to its local receptive field, then the output of j, k-th hidden neuron is

$$\sigma\left(b+\sum_{l=0}^{4}\sum_{m=0}^{4}w_{l,m}a_{j+l,k+m}\right).$$

where σ is the neural activation function, b is the shared value for the bias and

$a_{x,y}$ is the input activation at position x, y.

All the neurons in the first hidden layer detect exactly the same feature just at different locations in the input image. Weights and bias of a hidden neuron such that they pickup vertical edge. Map from the input layer to the hidden layer is called feature map. Weights and bias defining the feature map is called shared weights and biases. Shared weights and biases is called filter or kernel. Big advantage with sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network.

Convolutional layer consists of several different feature maps.

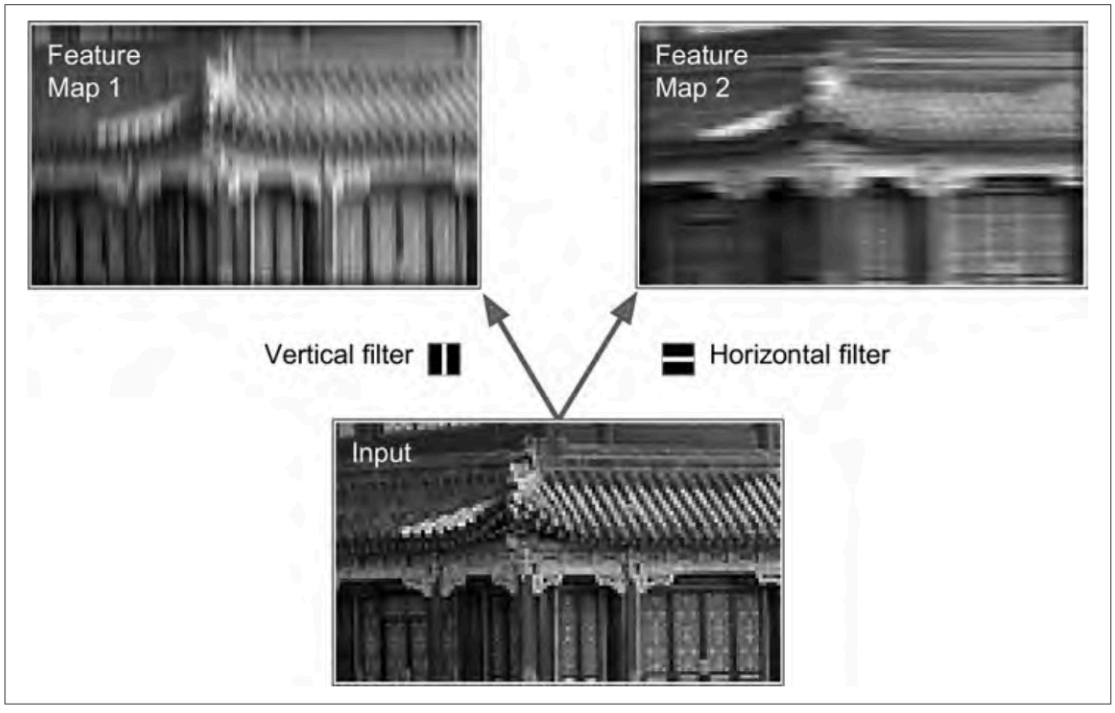


Figure 13-5. Applying two different filters to get two feature maps

Stacking multiple feature maps

With in one feature map, all neurons share same weights and bias term, but different feature maps may share different parameters.

TensorFlow Implementation

```
import numpy as np
from sklearn.datasets import load_sample_images

# Load sample images
dataset = np.array(load_sample_images().images, dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Create 2 filters
filters_test = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters_test[:, 3, :, 0] = 1  # vertical line
filters_test[3, :, :, 1] = 1  # horizontal line

# Create a graph with input X plus a convolutional layer applying the 2 filters
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1])  # plot 1st image's 2nd feature map
plt.show()
```

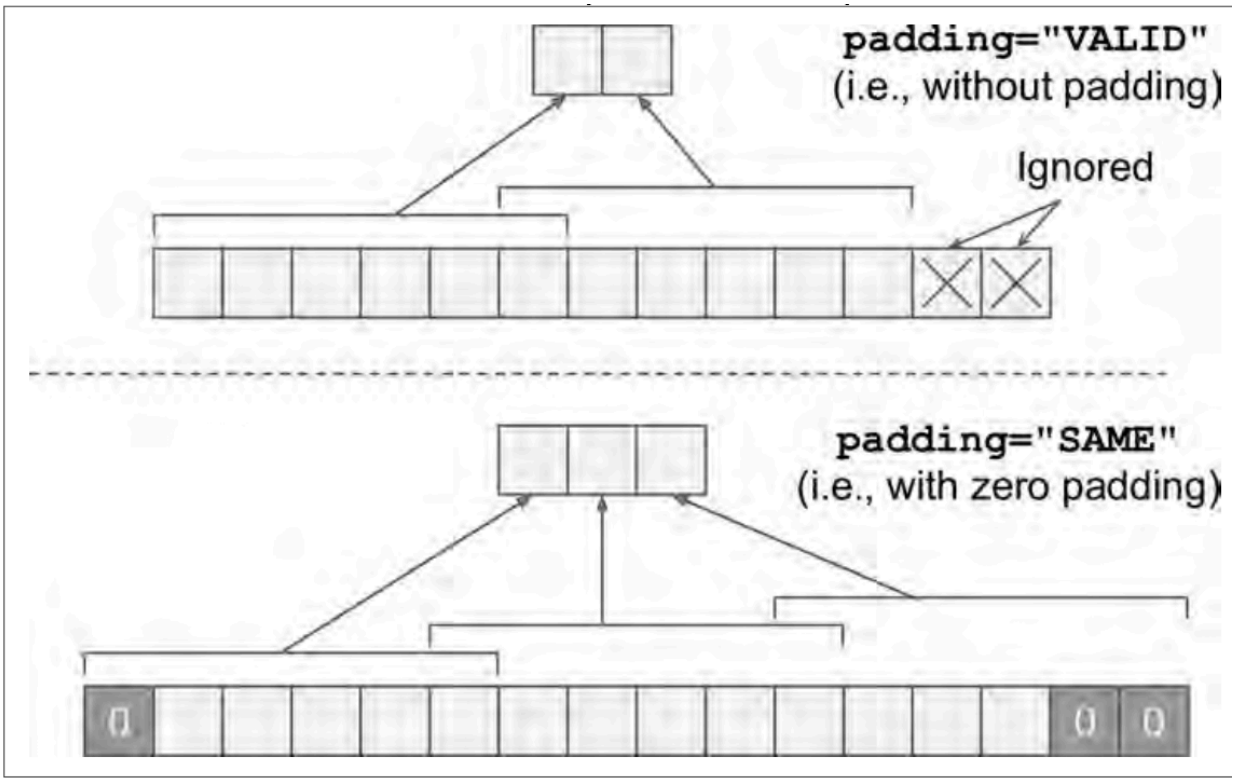


Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

Memory Requirements

For example, consider a convolutional layer with 5 × 5 filters, outputting 200 feature maps of size 150 × 100, with stride 1 and SAME padding. If the input is a 150 × 100 RGB image (three channels), then the number of parameters is (5 × 5 × 3 + 1) × 200 = 15,200 (the +1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.[7] However, each of the 200 feature maps contains 150 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5 × 5 × 3 = 75 inputs: that's a total of 225 million float multiplications. Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy 200 × 150 × 100 × 32 = 96 million bits (about 11.4 MB) of RAM.[8] And that's just for one instance! If a training batch contains 100 instances, then this layer will use up over 1 GB of RAM!

2. Activation Layer

Only non linear activation functions are used for learning.

A1*(A2*X) = (A1*A2)*X = A*X

reLU/Leaky reLU can be used as activation function.

Leaky reLU solves dying reLU problem.

reLU(MaxPool(Conv(M))) = MaxPool(reLU(Conv(M)))

3. Pooling Layer

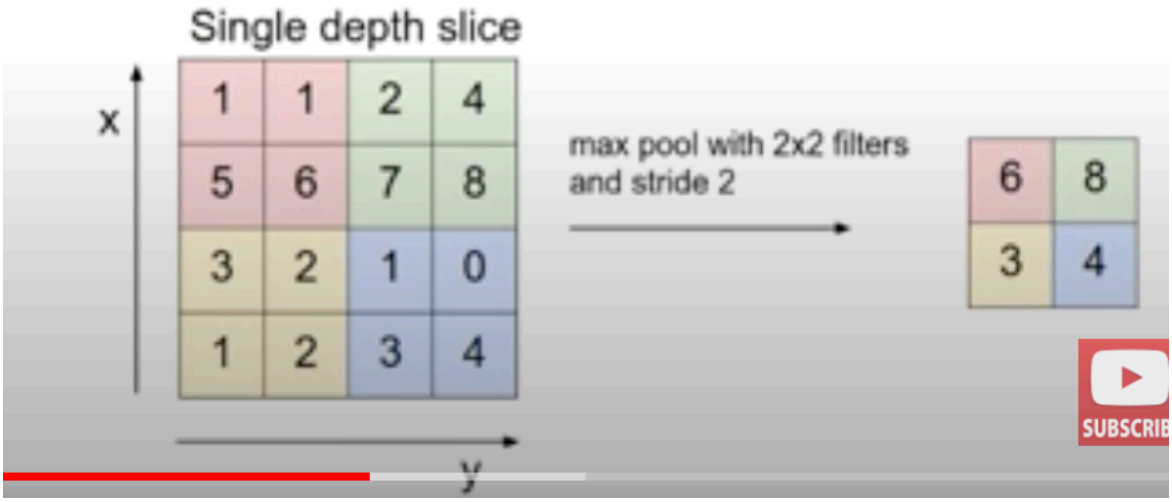
Down sampling features to reduce computational load, memory usage and number of parameters.

Two hyper parameters

1. Dimension of spatial extent
2. Stride

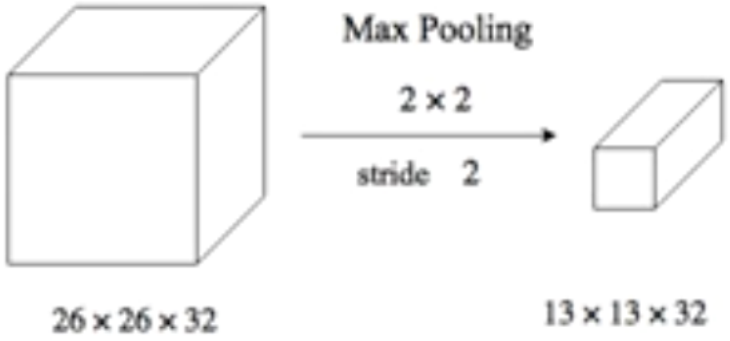
Pooling layer has no weights. But it uses an aggregation function of max or mean.

A common pooling layer uses 2x2 maxpool filter with stride 2.



Depth remains the same after pooling layer. Pooling reduces the chance of overfitting as there are less parameters.

For the MNIST dataset, after the convolutional layer



Number of features are reduced to 25% of the original number.

TensorFlow implementation of pooling layer with a 2x2 kernel, stride of 2 and no padding.

```
[...] # load the image dataset, just like above

# Create a graph with input X plus a max pooling layer
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

ksize argument represents batch size , height, width, channels.

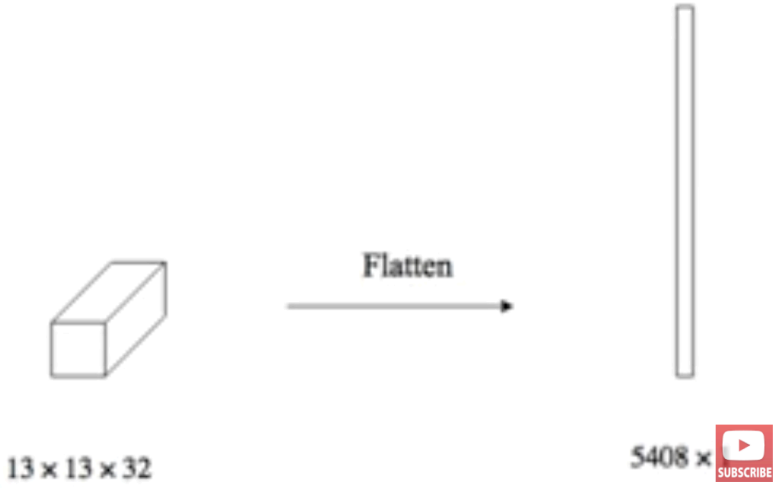
4. Fully Connected Layer

This layer is used to learn non-linear combination of features while convolutional layer provides meaningful, low-dimensional, invariant feature space.

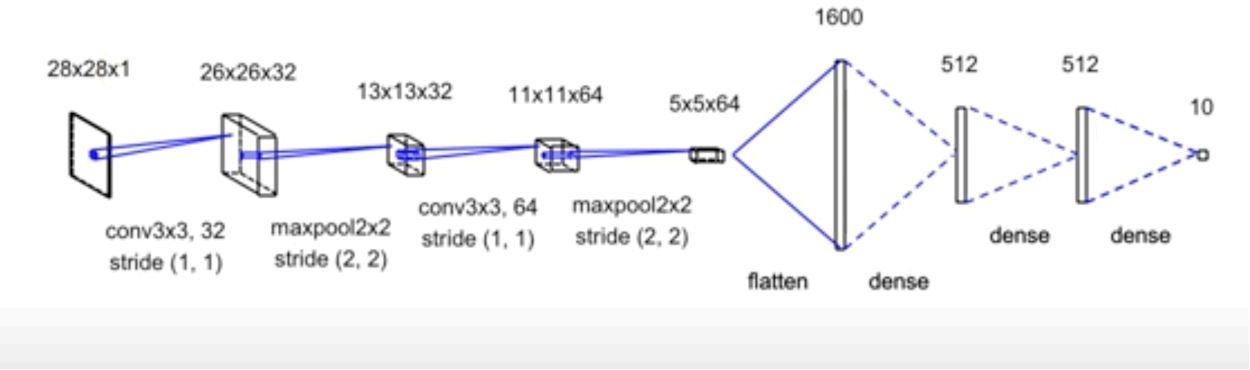
Pooling output is a 3D feature map

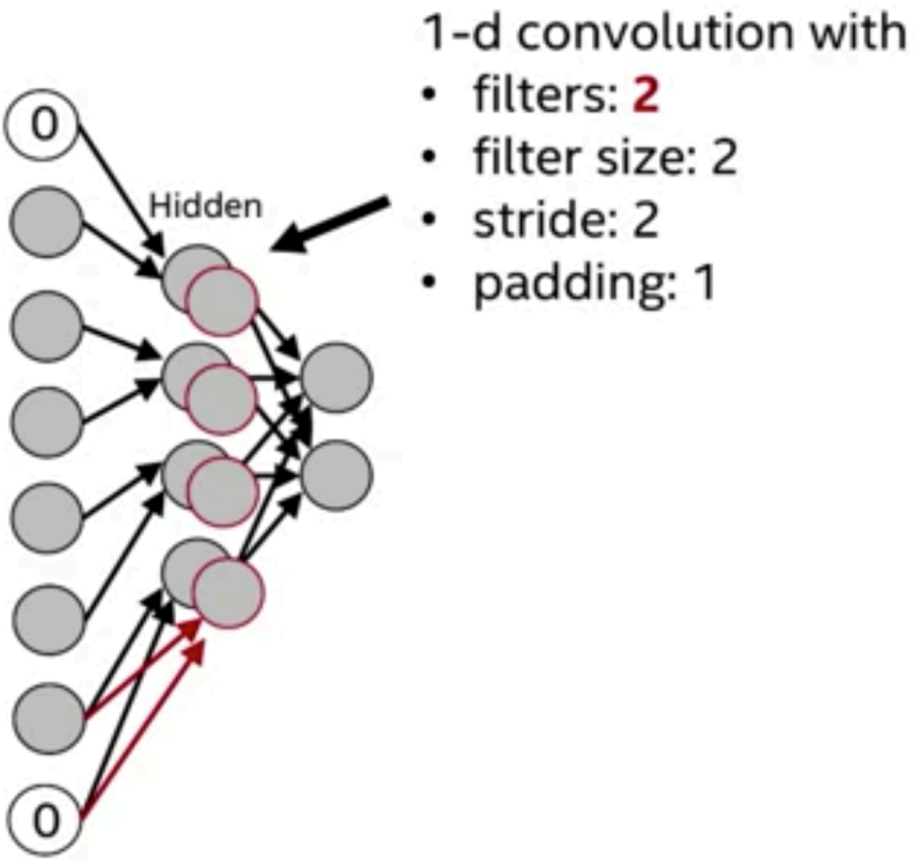
FC input is a 1D feature vector.

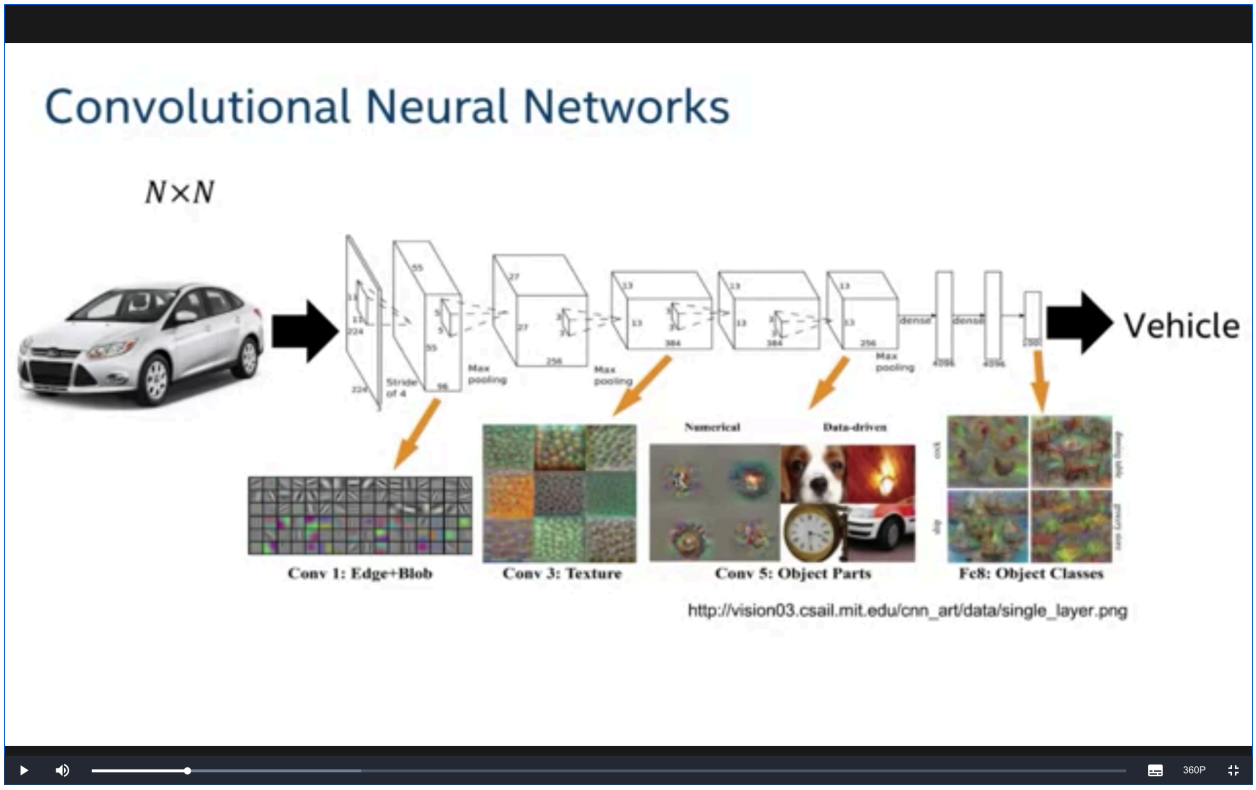
This conversion is done through flattening.

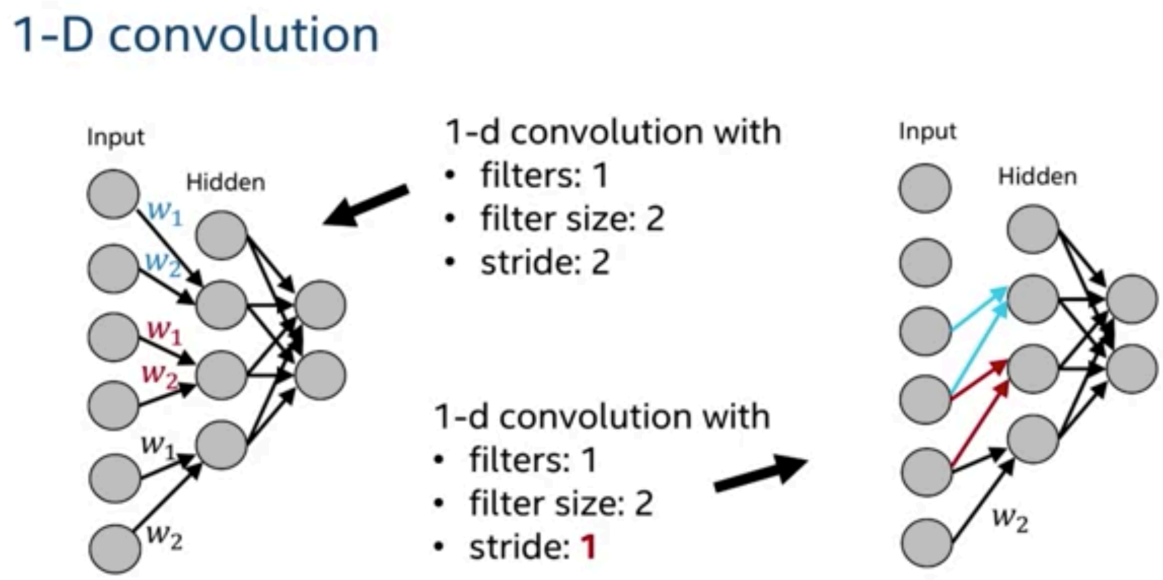


CNN for MNIST dataset to recognize hand written digits






Convolutional Neural Networks
N×N
Vehicle
Conv 1: Edge+Blob
Conv 3: Texture
Numerical
Data-driven
Conv 5: Object Parts
Fc8: Object Classes
http://vision03.csail.mit.edu/cnn_art/data/single_layer.png



https://www.coursera.org/lecture/intro-practical-deep-learning/convolutional-neural-networks-iOxAv

CNN Architectures

Typical CNN architectures stack a few convolutional layers followed by a pooling layer, again convolution layers and then pooling layer and so on. Image gets smaller and smaller as it progresses through the network but it gets deeper and deeper with more feature maps.Top of the stack is a regular feed forward neural network in which the output layer is usually a softmax layer that outputs class probabilities.

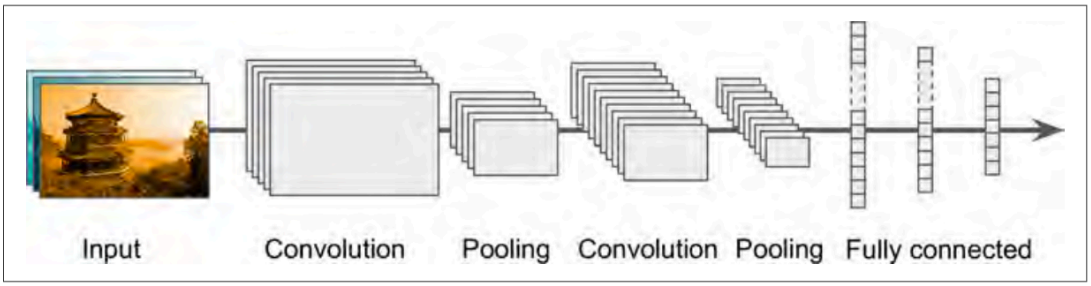


Figure 13-9. Typical CNN architecture

1. LeNet-5
2. AlexNet
3.GoogLeNet

4. ResNet

LeNet-5

Created by Yann LeCun in 1998.

Used for hand written digit recognition(MNIST dataset)

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg Pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg Pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

- MNIST images are 28×28 pixels, but they are zero-padded to 32×32 pixels and normalized before being fed to the network. The rest of the network does not use any padding, which is why the size keeps shrinking as the image progresses through the network.
- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.

  - Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps). See table 1 in the original paper for details.
  - The output layer is a bit special: instead of computing the dot product of the inputs and the weight vector, each neuron outputs the square of the Euclidian distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross entropy cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and thus converging faster.

AlexNet

AlexNet CNN architecture won 2012 ILSVRC challenge

ILSVRC-ImageNet Large Scale Visual Recognition Challenge

ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. Achieved 17% top-5 error rate.

Developed Alex Krizhevsky.

It was the first to stack convolutional layers directly on one another instead of pooling layer on top of each convolutional layer.

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

Two regularization techniques are used to reduce overfitting.

1. Dropout with 50% dropout rate to the output of layers F8 and F9
2. Data Augmentation

It used Local Response Normalization.

Equation 13-2. Local response normalization

$$b_i = a_i\left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2\right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

- b_i is the normalized output of the neuron located in feature map i, at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k, α, β, and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

GoogLeNet

Developed by Christian Szegedy.

ILSVRC 2014 classification winner

6.7% top5 error

Efficient Inception Module

12x less parameters thanAlexNet

Architecture of an Inception Module

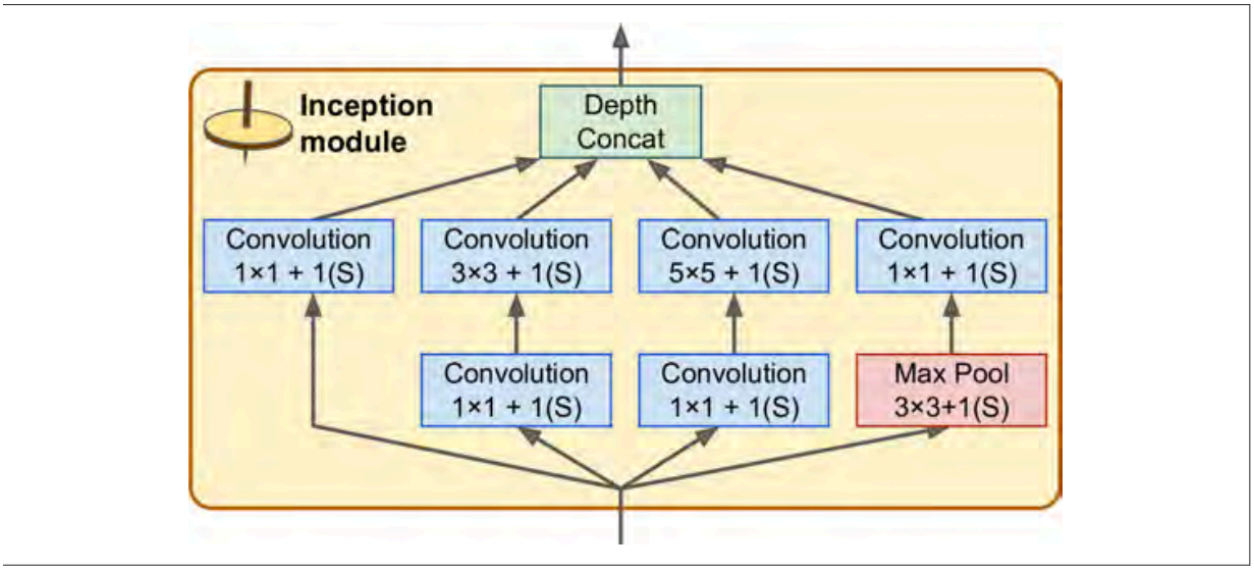


Figure 13-10. Inception module

Notation 3x3 + 1(S) means that the layer uses 3x3 filter and a stride of 1.with the SAME padding.

Different kernel sizes (1x1,3x3,5x5) are used in second convolutional layer allows to capture patterns at different scales.

As all the convolutional and maxpool layers uses a stride of 1, the outputs can be concatenated along the depth dimension in the final depth concat layer.

Can be implemented using concat() functioning TensorFlow.

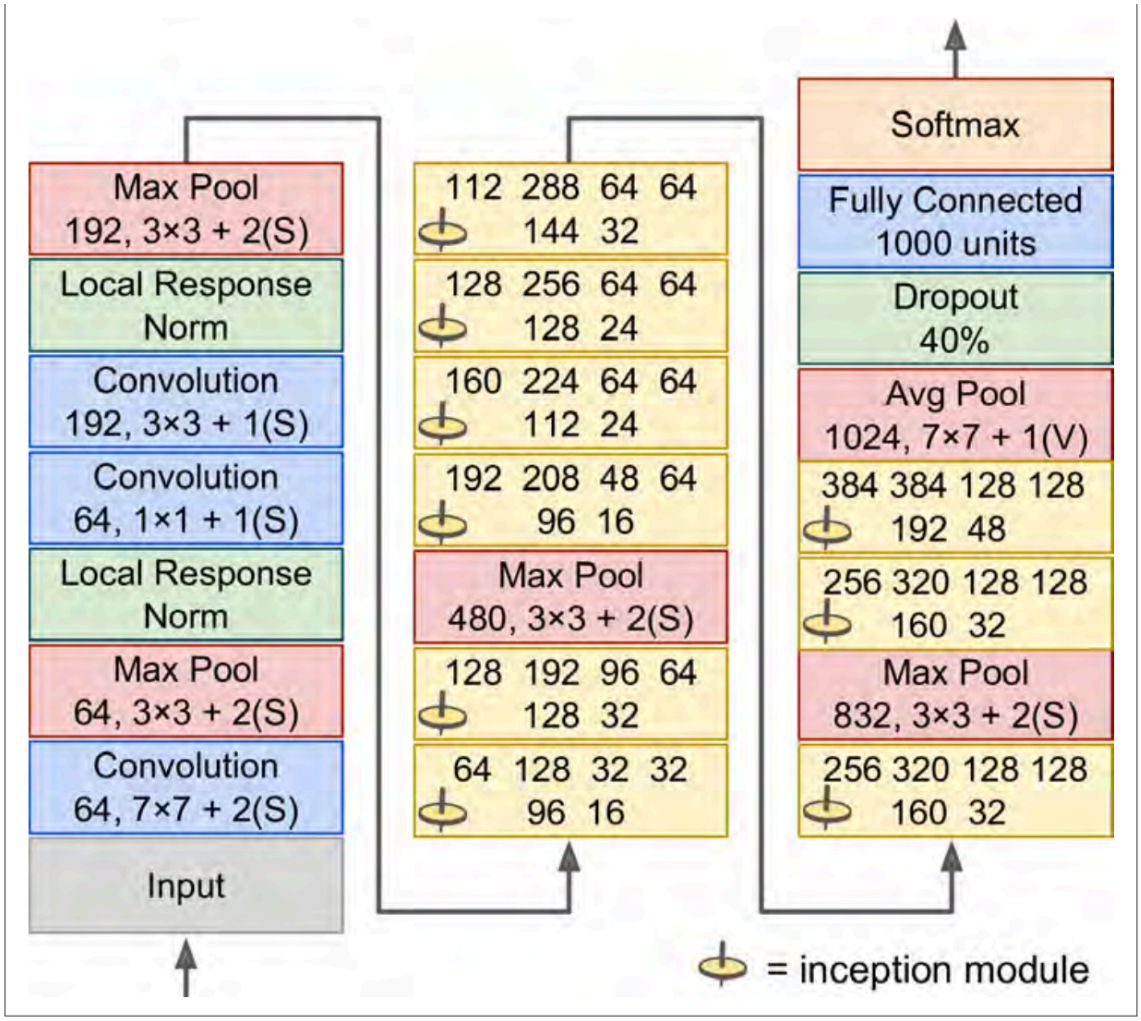
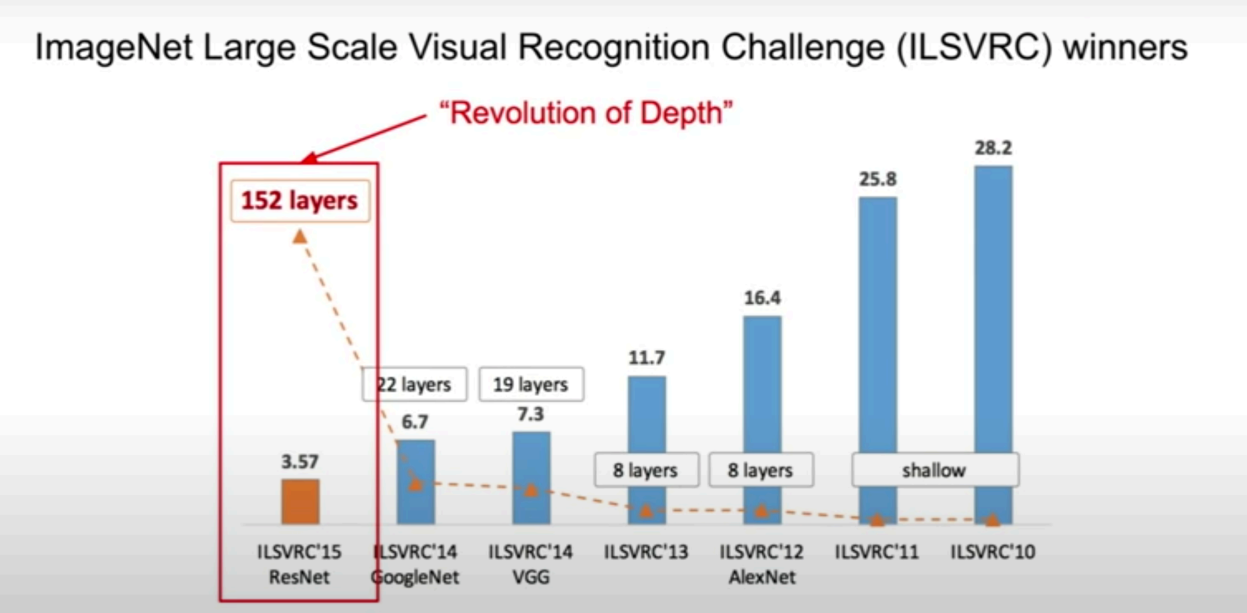


Figure 13-11. GoogLeNet architecture

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a *bottleneck layer*. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next a max pooling layer reduces the image height and width by 2, again to speed up computations.

- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.
- Next, the average pooling layer uses a kernel the size of the feature maps with VALID padding, outputting 1×1 feature maps: this surprising strategy is called *global average pooling*. It effectively forces the previous layers to produce feature maps that are actually confidence maps for each target class (since other kinds of features would be destroyed by the averaging step). This makes it unnecessary to have several fully connected layers at the top of the CNN (like in AlexNet), considerably reducing the number of parameters in the network and limiting the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with a softmax activation function to output estimated class probabilities.

ResNet



Very deep network using residual connections

152 layer model for ImageNet

ILSVRC 2015 challenge winner

3.57% top 5 error

Introduces skip connections and performs heavy batch normalization.

Idea : Deeper network may be made from a shallow network by copying weights from shallow network and setting other layers in the deeper network to be identity mapping.

This formulation indicates that deeper model should not produce higher training error than the shallow counterpart.

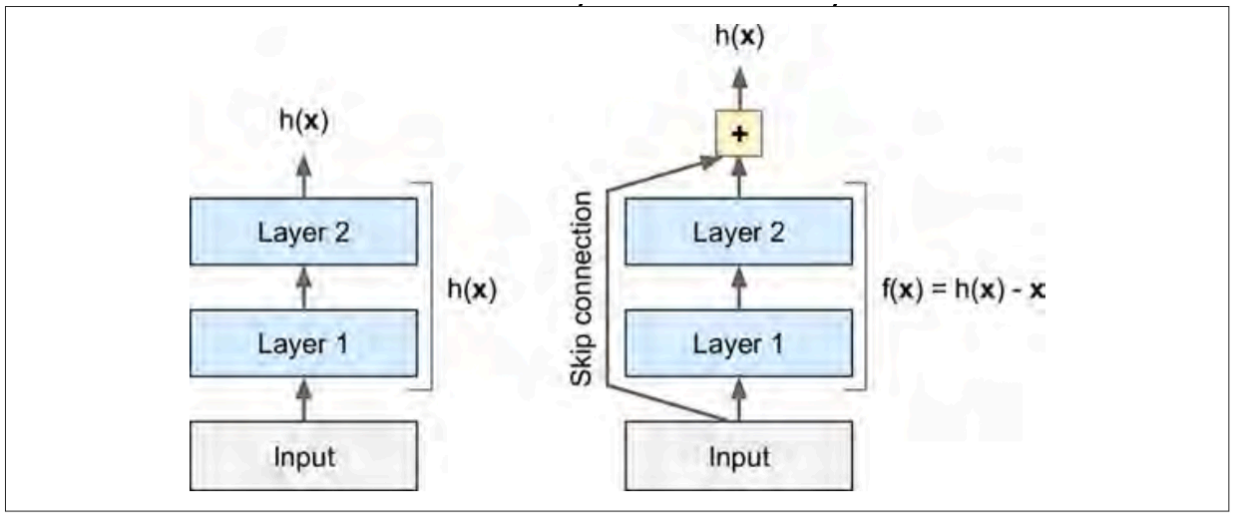


Figure 13-12. Residual learning

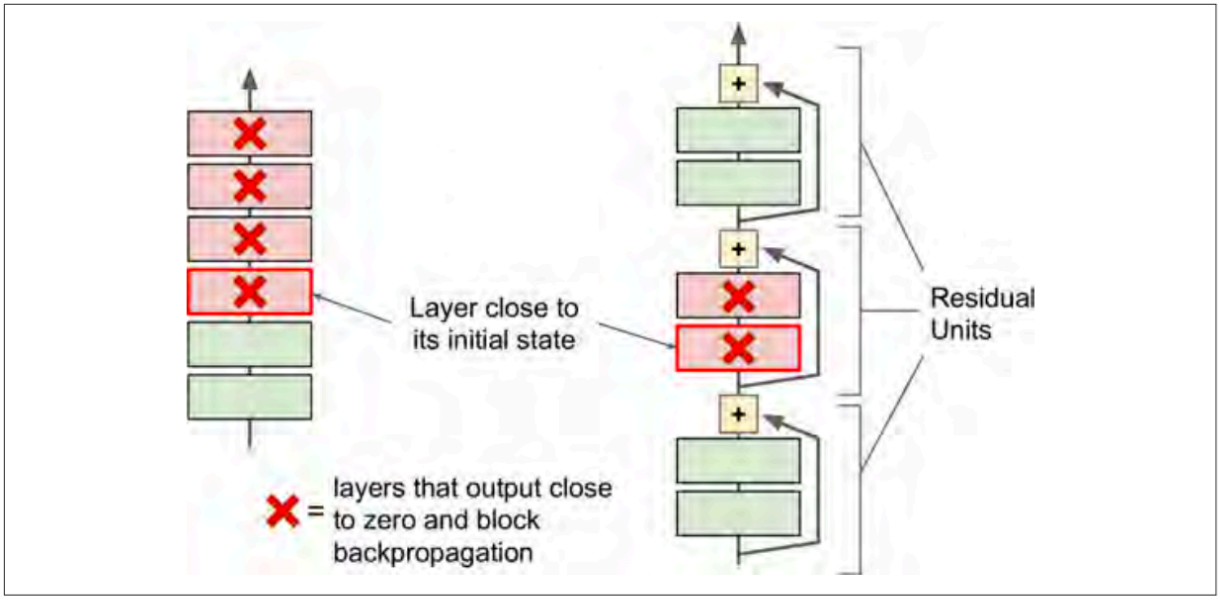


Figure 13-13. Regular deep neural network (left) and deep residual network (right)

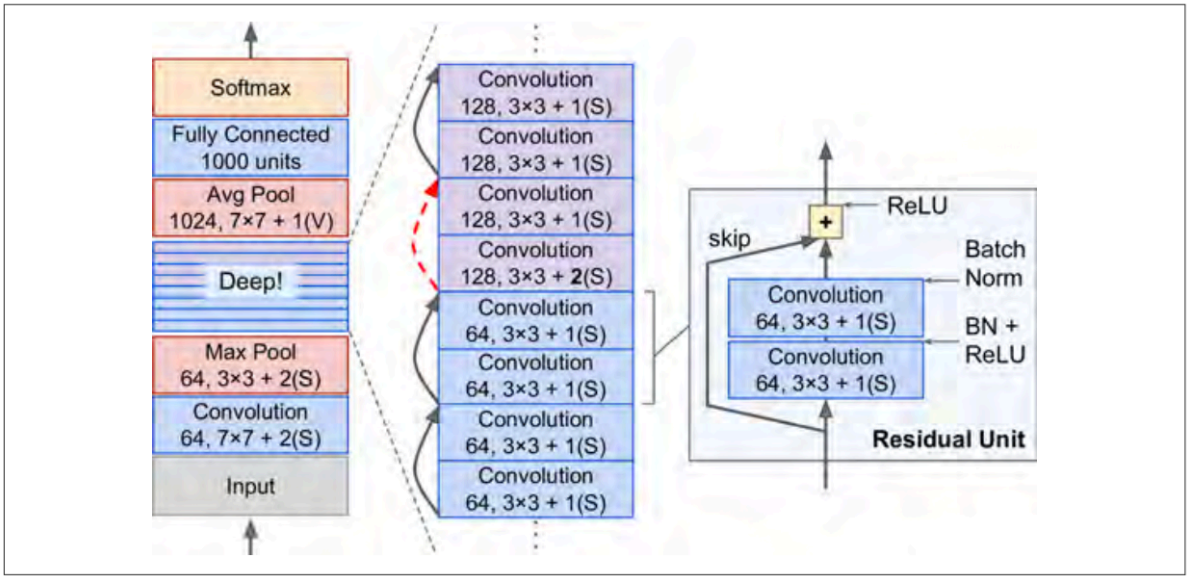


Figure 13-14. ResNet architecture

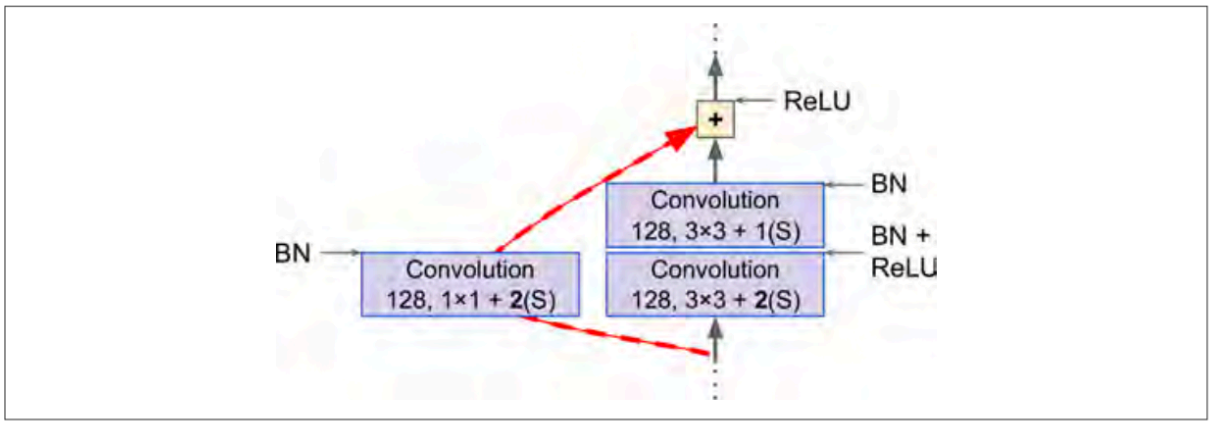


Figure 13-15. Skip connection when changing feature map size and depth

https://towardsdatascience.com/review-hikvision-1st-runner-up-in-ilsvrc-2016-object-detection-1f0a42cda767