Introduction to

Deep Learning & Neural Networks

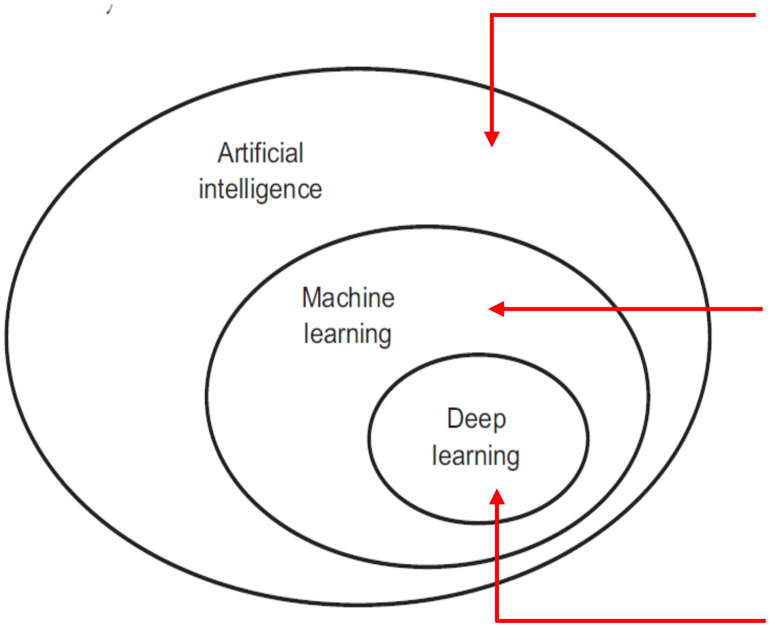
Kannan Singaravelu

23 May 2022

In this Lecture…

- Why do we need a different learning algorithm?
- Building blocks of neural networks
- Understanding activation functions
- Deep Learning for computer vision
- How do computers understand images?
- Underlying sequence modeling
- Problem of short term memory
- Brief overview of Generative Adversarial Networks
- Pros & cons of deep learning

Understanding AI



- Science & Engineering of making intelligent machines
- Ability to learn automatically without being explicitly programmed
- Layered or hierarchical representations and learning using Neural Networks

Why Deep Learning Now?

1952 Stochastic Gradient Descent (SGD)

1958 Perceptron

- Learnable Weights

.

.

.

1986 Backpropagation

- Multi-Layer Perceptron (MLP)

1995 Deep Convolutional Neural Network

- Digit Recognition

1997 Long Short-Term Memory

- Sequential Timeseries

.

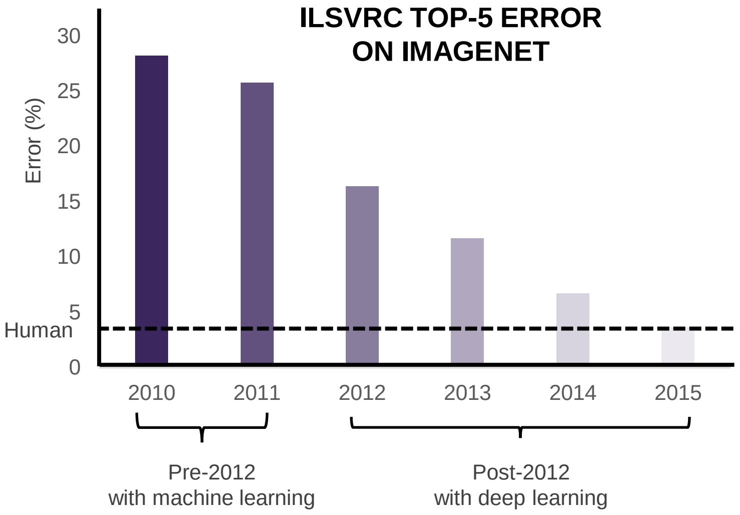
2012 Watershed Moment in NN History

- ImageNet

Big Data - Large datasets. Easier storage and collection

Hardware – Faster CPUs. Massively parallelizable chips : GPUs, and TPUs

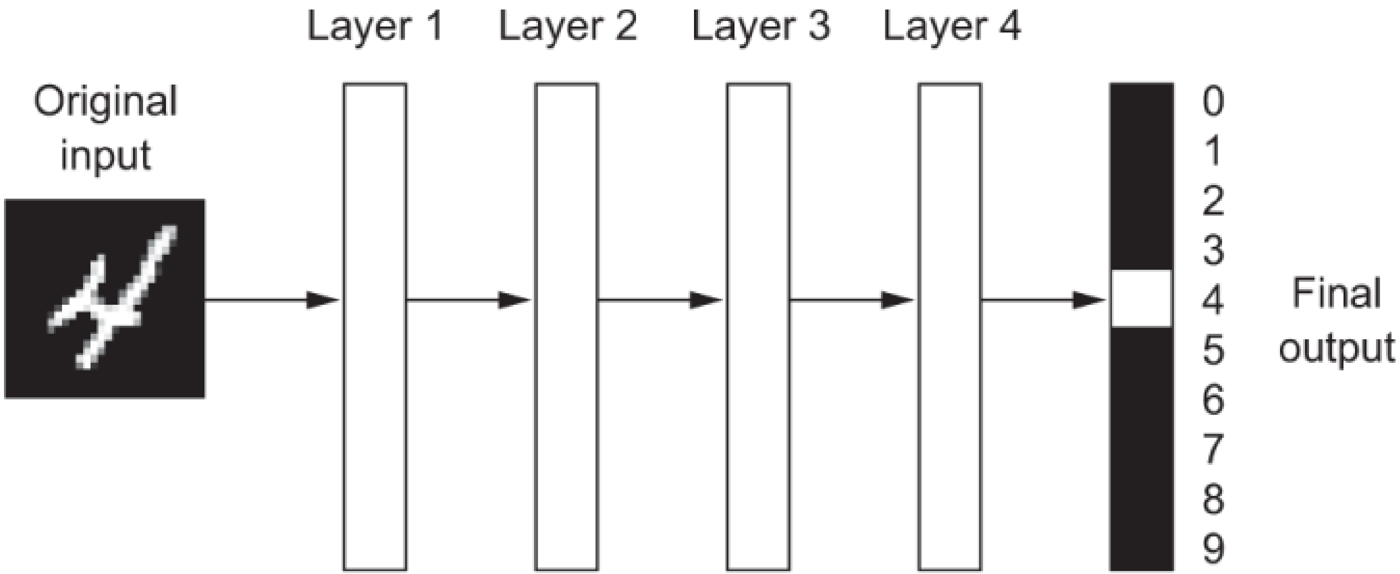
Algorithms & Software - Improved techniques. New Models & Toolsets. Democratization of Deep Learning



What is Deep Learning?

- Inspired from our understanding of human brains
- Layered or Hierarchical representations learning
- *“**deep**”* in deep learning stands for successive layers of representations
- Layered representations are learned via models called ***neural network***
- Neural networks are structured layers stacked on top of each other
- Mathematical framework for learning representations from data

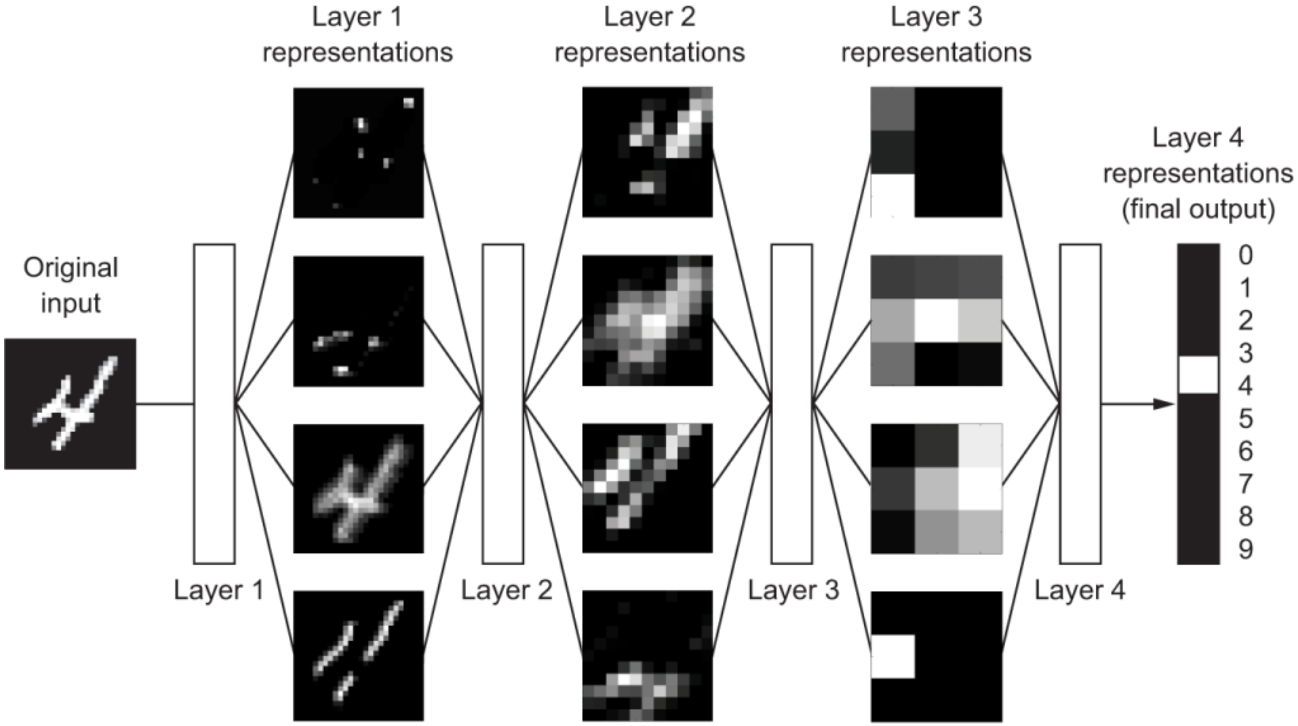
Layered Representations



Network of layers transforms image to digit

Image Source: Francois Chollet (2017), Deep Learning with Python

Layered Representations



Multistage way to learn data representations

Information-distillation operation with successive filters

Image Source: Francois Chollet (2017), Deep Learning with Python

How Deep Learning Works?

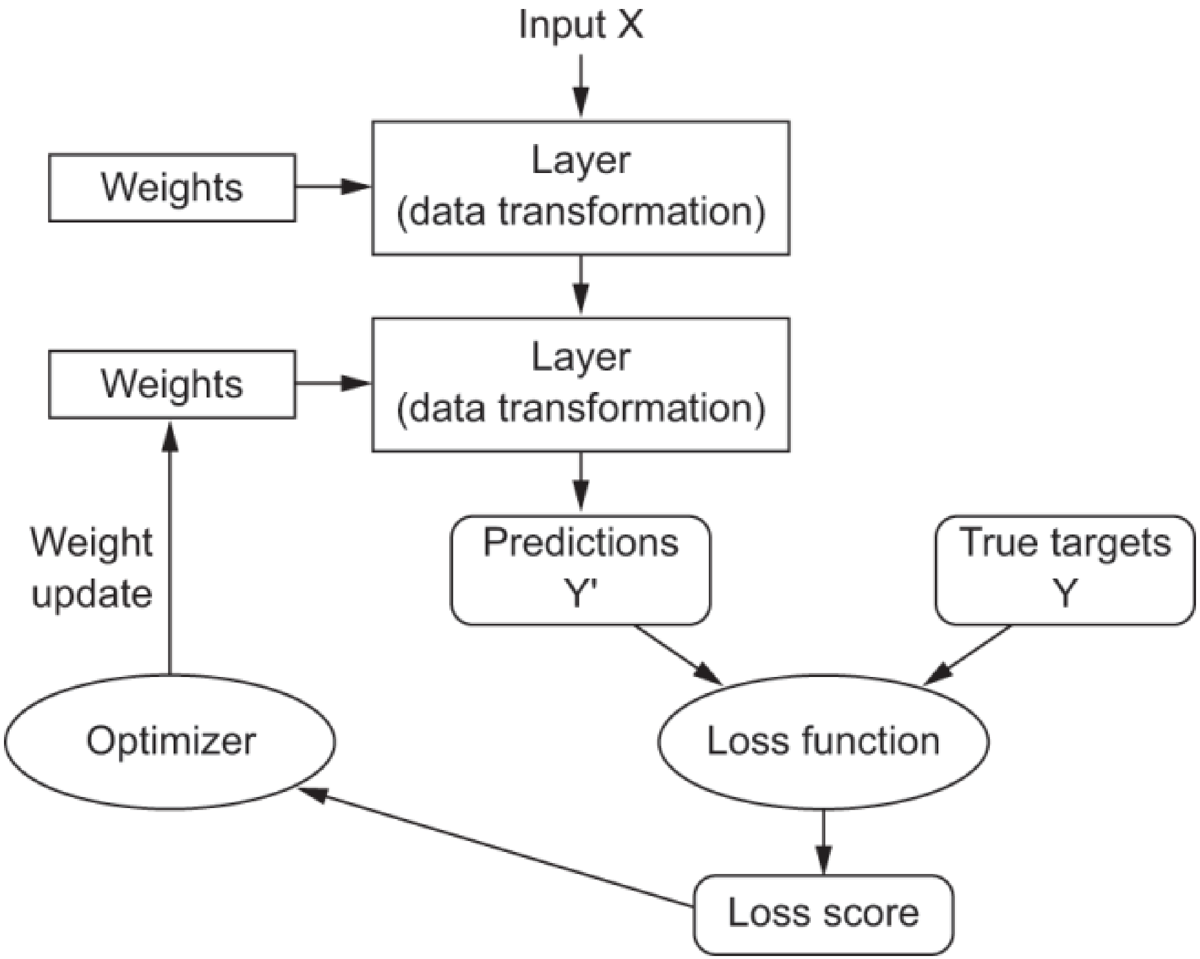
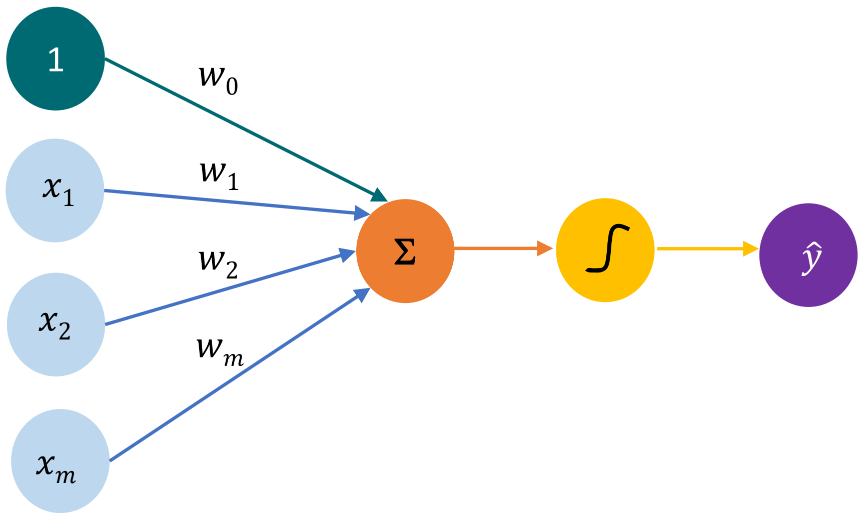


Image Source: Francois Chollet (2017), Deep Learning with Python

Building Blocks of Deep Learning

- Perceptron
- Forward Propagation
- Activation Functions
- Weight Initialization
- Backpropagation

Preceptron: The Forward Propogation

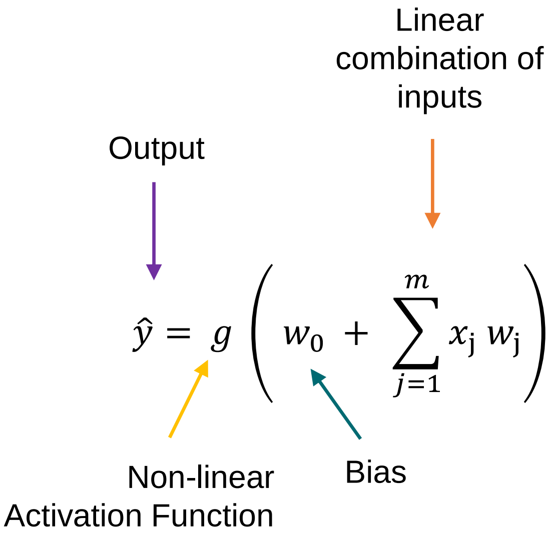


Inputs Weights

Sum

Activation Function

Output



Linear combination of inputs

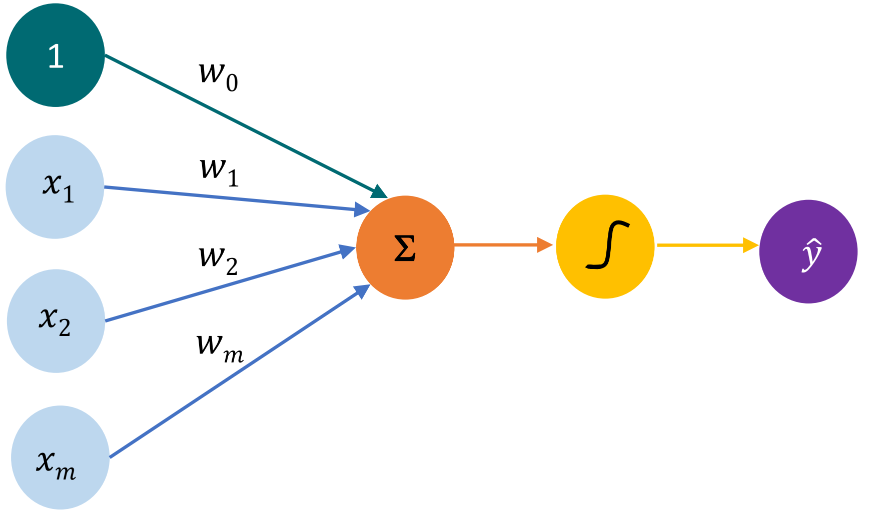
Output

$$\hat{y} = g\left(w_0 + \sum_{j=1}^{m} x_{\mathrm{j}}\, w_{\mathrm{j}}\right)$$

Non-linear Activation Function

Bias

Preceptron: The Forward Propogation



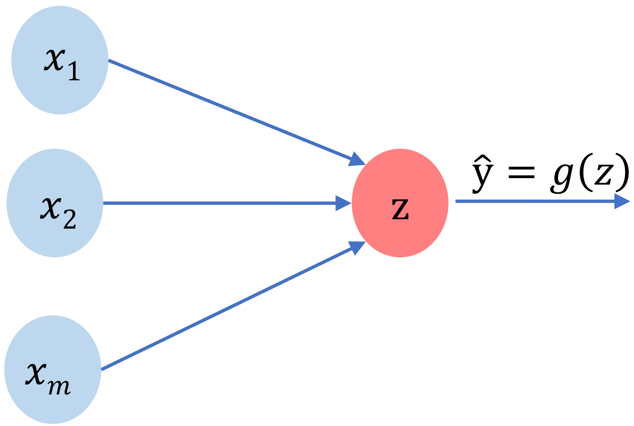
$$\hat{y} = g\left(w_0 + \sum_{j=1}^{m} x_j\, w_j\right)$$

$$\hat{y} = g\,(w_0 + \boldsymbol{X}^T\boldsymbol{W})$$

$$\text{where: } \boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \; and \; \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Inputs Weights Sum Activation Function Output

Preceptron: Simplified

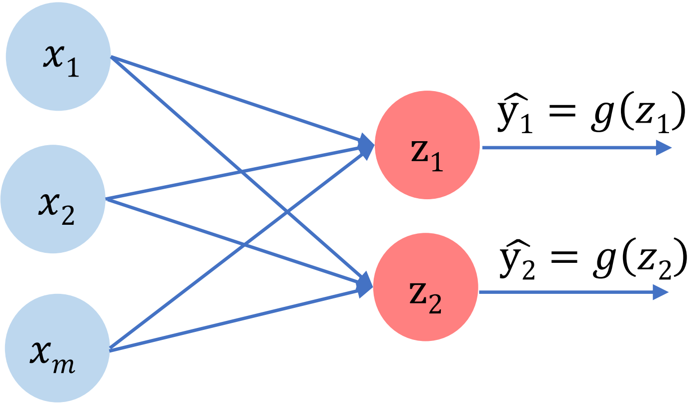


$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

$$\hat{y} = g(z) = a$$

Removing the bias in the visual representation for simplicity and ease of representation

Multi Output Preceptron

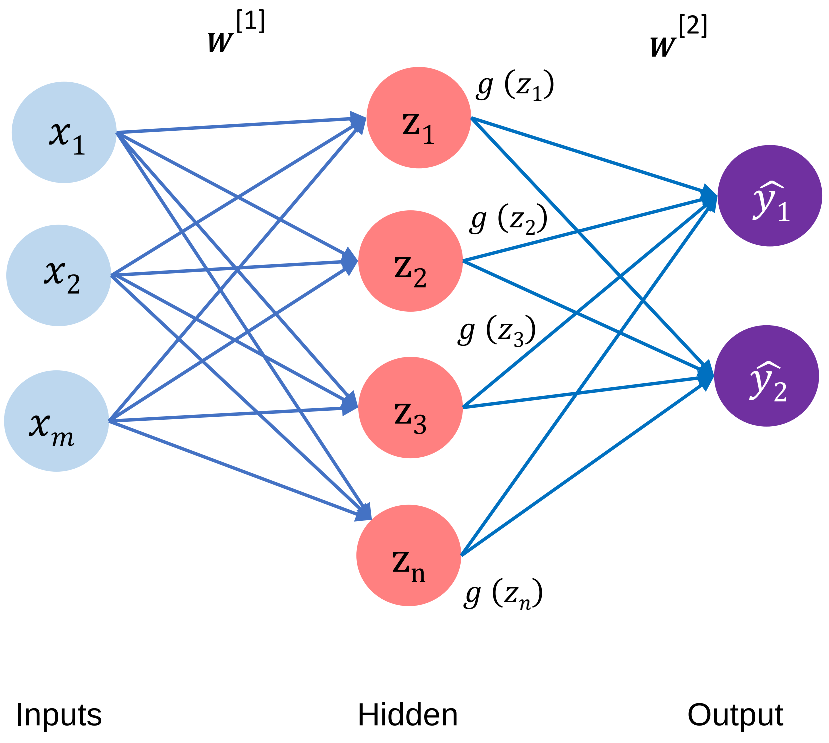


$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

$$\hat{y}_i = g(z_i) = a_i$$

MLPs are fully connected networks where all inputs are densely connected to all outputs

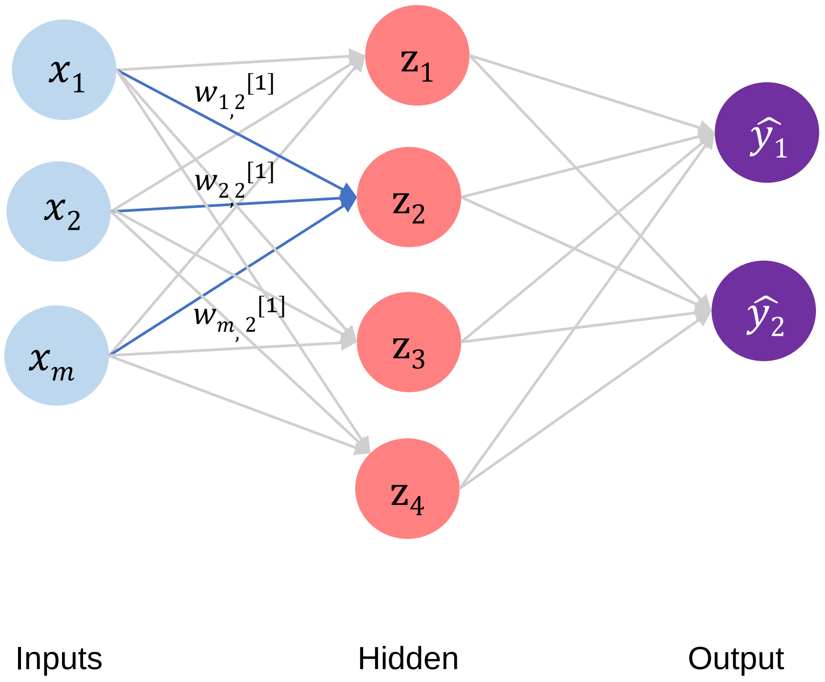
Single Hidden Layer Network



$$z_i = w_{0,i}^{[1]} + \sum_{j=1}^{m} x_j\, w_{j,i}^{[1]}$$

$$\hat{y}_i = g\left(w_{0,i}^{[2]} + \sum_{j=1}^{n} z_j\, w_{j,i}^{[2]}\right)$$

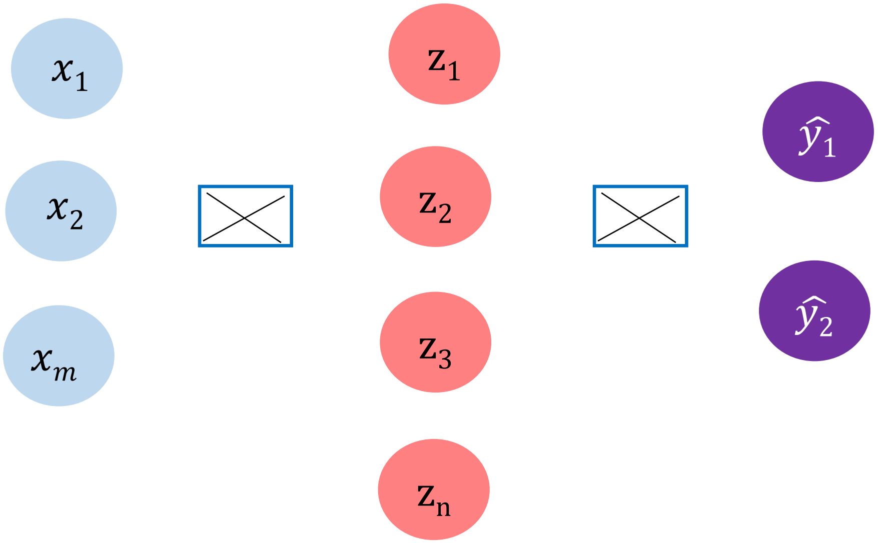
Single Hidden Layer Network



$$z_2 = w_{0,2}^{[1]} + \sum_{j=1}^{m} x_j \, w_{j,2}^{[1]}$$

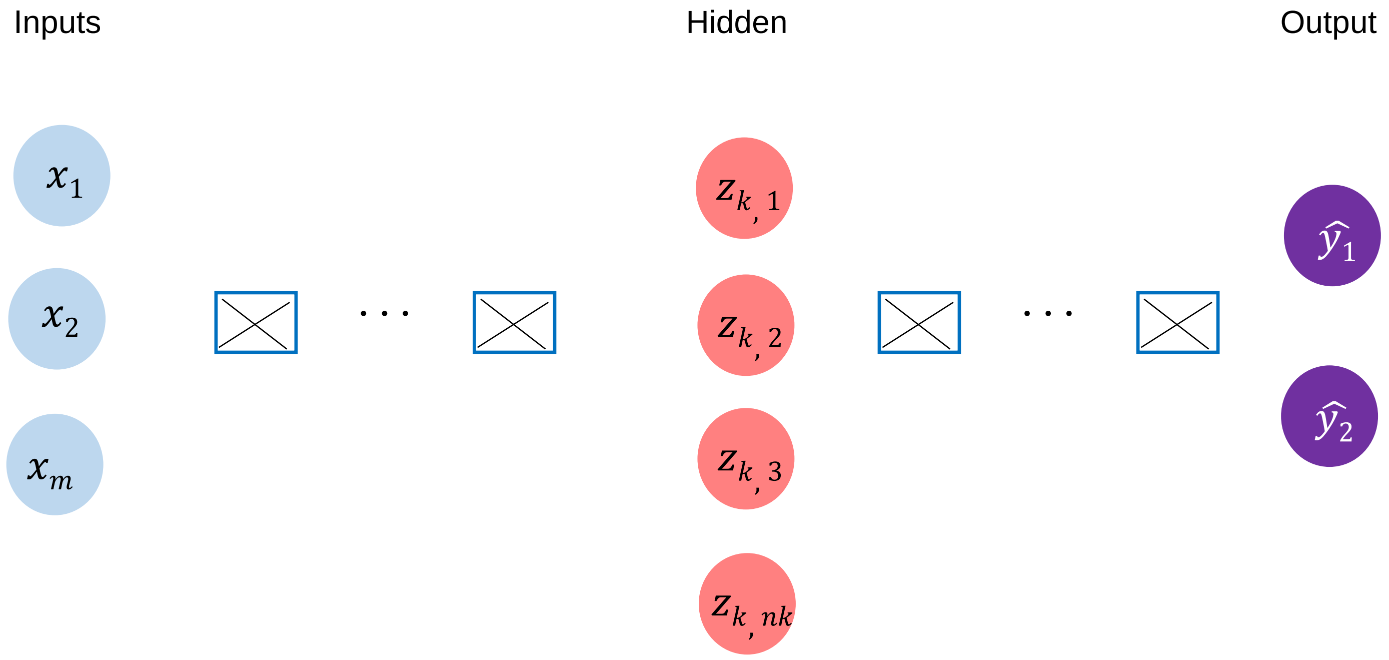
$$= w_{0,2}^{[1]} + x_1 w_{1,2}^{[1]} + x_2 w_{2,2}^{[1]} + x_m w_{m,2}^{[1]}$$

Multi Output Preceptron



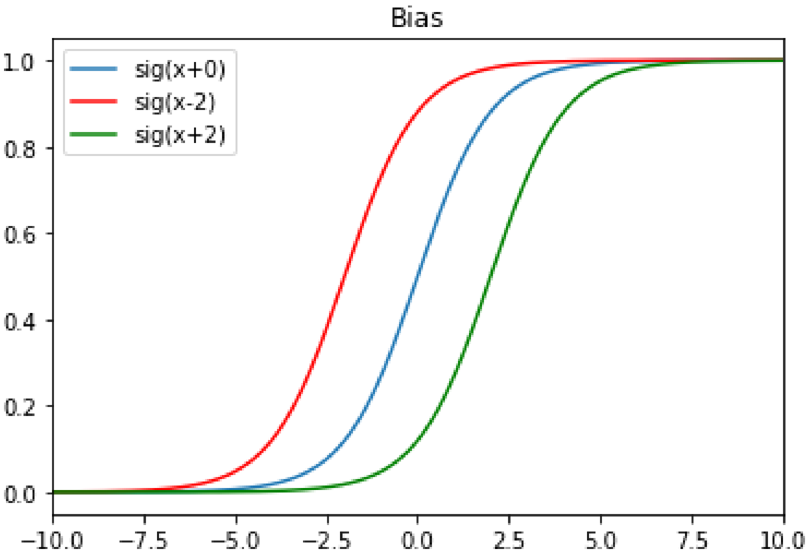
Inputs Hidden Output

Deep Neural Network



$$z_{k,i} = w_{0,i}^{[k]} + \sum_{j=1}^{nk_{-1}} g(z_{\mathrm{k}-1,j})\ w_{j,i}^{[k]}$$

Bias

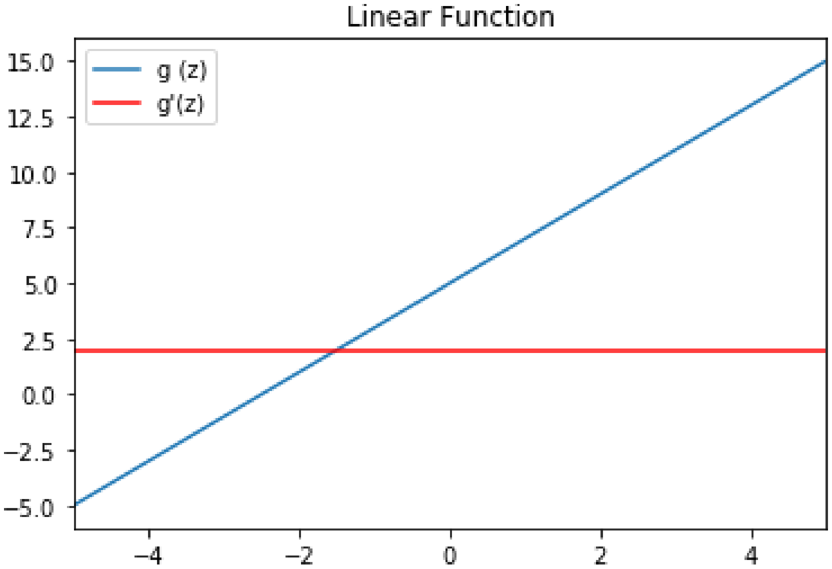


- Bias can be thought of as analogous to the role of a constant in a linear function.
- Bias value allows the activation function to be shifted to the left or right, to better fit the data.
- Influence the output values and doesn’t interact with the actual input data.

Activation Functions

- New take on learning representations from data
- Introduce non-linearity in the network
- Decide whether a neuron can contribute to the next layer
- Specify contribution threshold
- Should be computationally efficient

Why a Non-Linear Functions?

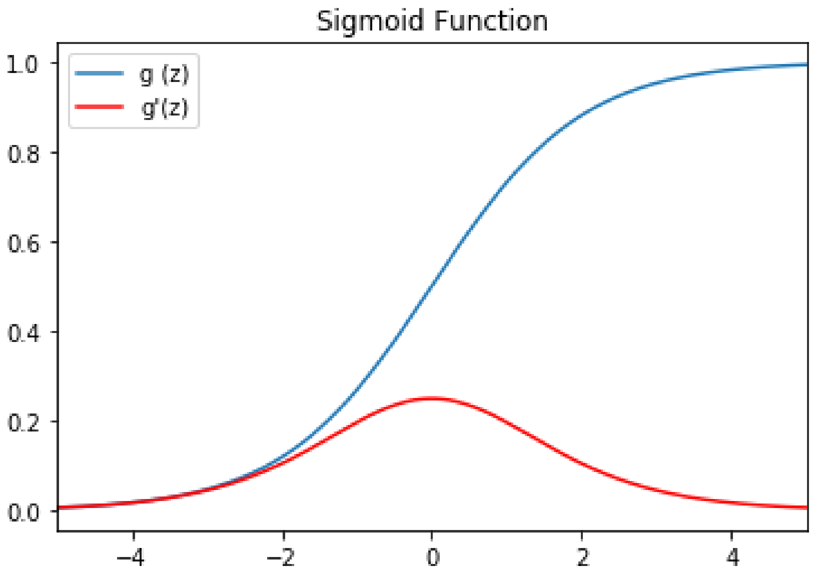


- Derivative is constant
- No gradient relationship with input data
- Unbounded output
- N-layer network ≈ single layer

$$g(z) = m(z) + b$$

$$g'(z) = m$$

Sigmoid Function

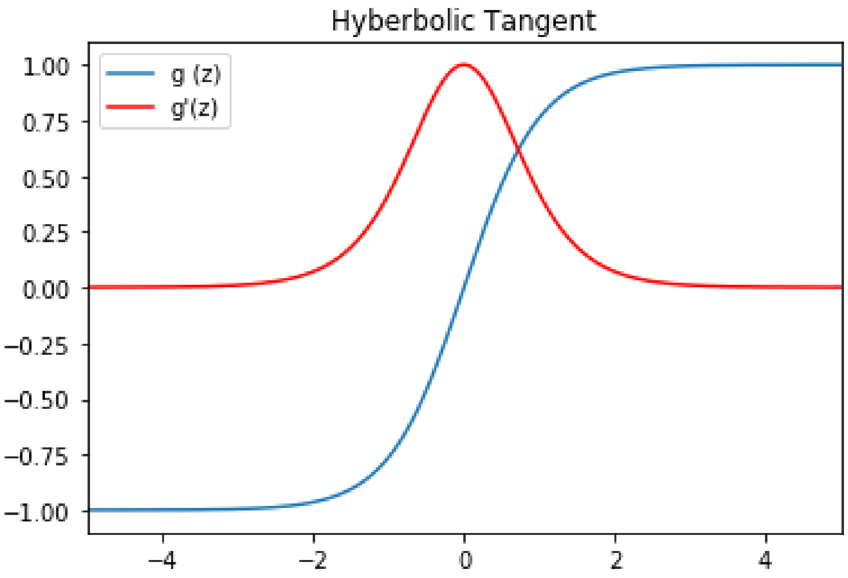


- Non-binary activations (analog outputs)
- Differentiable
- Bounded output
- Vanishing gradient problem

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

Tanh Function

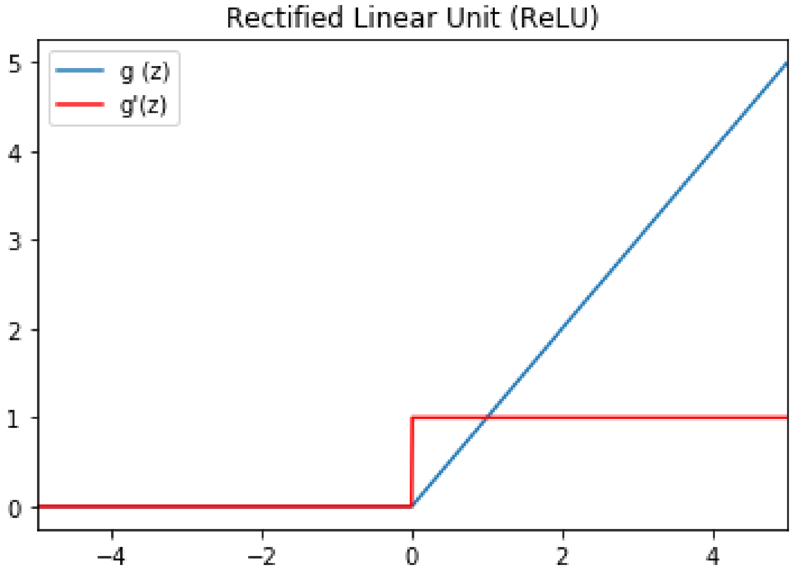


- Non-linear
- Derivative steeper than Sigmoid
- Bounded output
- Vanishing gradient problem

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^{-2}$$

ReLU Function

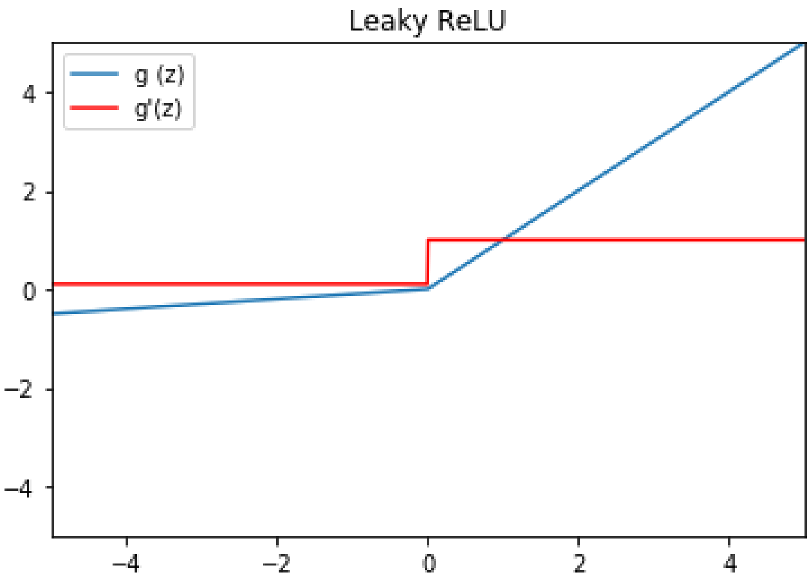


- Non-linear
- Unbounded output
- Sparse activations
- Dying ReLU problem

$$g(z) = max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & otherwise \end{cases}$$

Leaky ReLU Function



- Non-zero slope
- Less computationally expensive
- Parametric ReLU function with a = 0.01

$$g(z) = max(0.01z, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0.01, & otherwise \end{cases}$$

Which Activation Functions to Use?

- More than 20 activation functions including Hard Sigmoid, Softmax, ELU, PReLU, Maxout and Swish
- No single activation function that works in all cases
- Linear activation function can only be used in output layer for regression problem
- ReLU and their combinations preferred - used only in hidden layers
- Sigmoid, Softmax work better for classifier - preferred in output layers

Weight Initialization

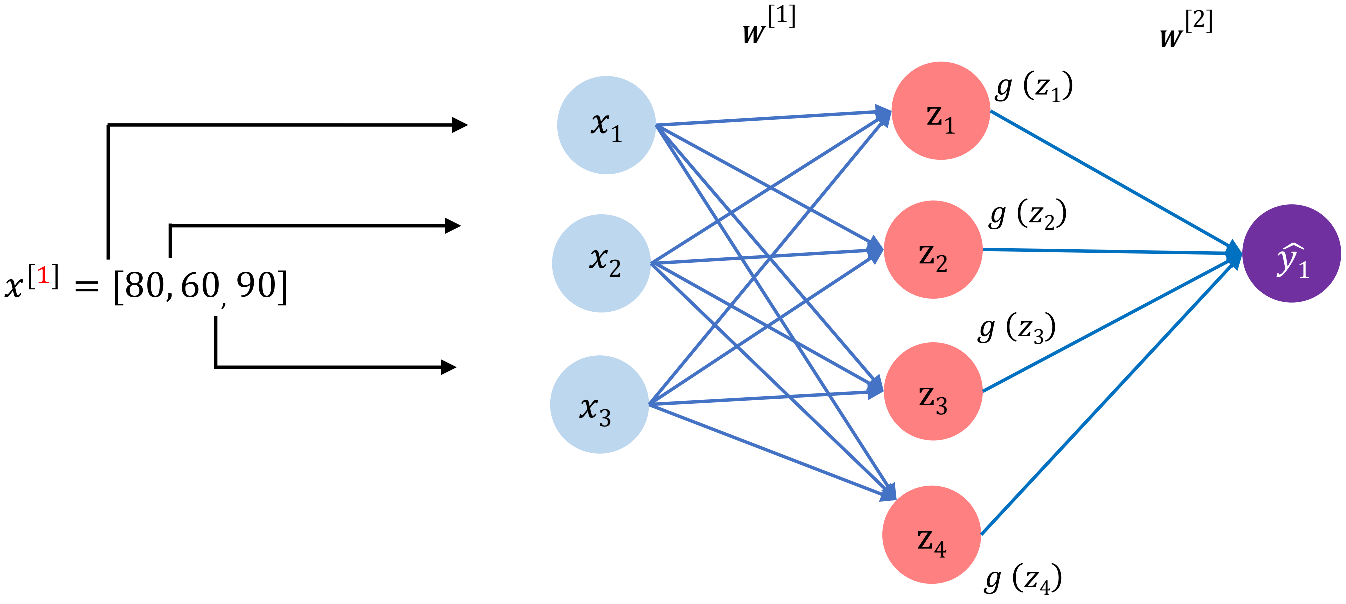
- Small, different and have good variance
- Uniform Distribution $\boldsymbol{W}_{\boldsymbol{j}, \boldsymbol{i}} \sim U\left[\frac{-1}{\sqrt{fan_{in}}}, \frac{1}{\sqrt{fan_{in}}}\right]$
- Xavier Glorot
  - Normal $\boldsymbol{W}_{\boldsymbol{j}, \boldsymbol{i}} \sim N(0, \sigma)$, where $\sigma = \sqrt{\frac{2}{(fan_{in}+fan_{out})}}$
  - Uniform $\boldsymbol{W}_{\boldsymbol{j}, \boldsymbol{i}} \sim U\left[\frac{-\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}\right]$
- Uniform and Xavier Glorot works well with sigmoid activation function
- Xavier Glorot works well with tanh activation function

Weight Initialization

- He Init
  - Normal $\boldsymbol{W_{j,i}} \sim N(0, \sigma)$, where $\sigma = \sqrt{\frac{2}{(fan_{in})}}$
  - Uniform $\boldsymbol{W_{j,i}} \sim U\left[-\sqrt{\frac{6}{(fan_{in})}}, \sqrt{\frac{6}{(fan_{in})}}\right]$
- Works well with ReLU activation function

Applying Neural Networks

Example : Will I pass the CQF program?



$x^{[1]} = [80, 60, 90]$

Predicted 0.3

Actual 1

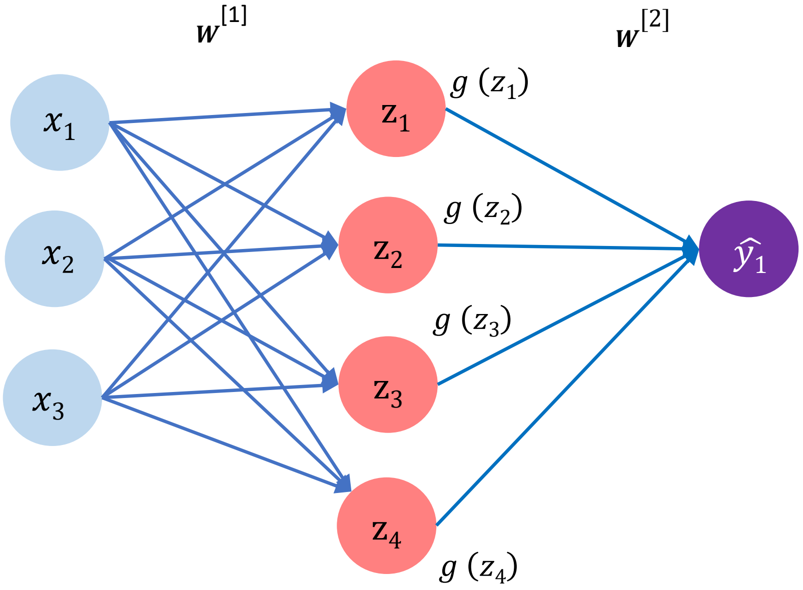
$$L\left(f\left(x^{[i]}; \boldsymbol{W}\right), y^{[i]}\right)$$

$x_1 =$ Exam 1 marks | $x_2 =$ Exam 2 marks | $x_3 =$ Exam 3 marks

Quantifying Loss

Empirical loss measures the total loss over the entire dataset

$$\boldsymbol{X} = \begin{bmatrix} 80 & 60 & 90 \\ 70 & 50 & 80 \\ 100 & 80 & 90 \\ \cdots & \cdots & \cdots \end{bmatrix}$$



$$\begin{array}{cc} f(x) & y \\ \begin{bmatrix} 0.3 \\ 0.8 \\ 0.6 \end{bmatrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

Quantifying Loss

- Loss of our neural network measures the cost incurred from incorrect predictions.
- The loss or objective function is the quantity that will be minimized during training.
- **Binary Cross Entropy -** used with models that output a probability between 0 and 1

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} y^{[i]}\, log\big(f(x^{[i]};\boldsymbol{W})\big) + (1 - y^{[i]})\, log\big(1 - f(x^{[i]};\boldsymbol{W})\big)$$

- **Mean Squared Error -** used with regression models that output continuous values

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\big(y^{[i]} - f(x^{[i]};\boldsymbol{W})\big)^{2}$$

Optimization of Loss

- Training the neural networks essentially mean finding the network weights that achieve the lowest loss.

$$\boldsymbol{W}^{*} = \underset{W}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} L\left(f(x^{[i]}; \boldsymbol{W}), y^{[i]}\right)$$

- Optimizer to determine how the network will be updated based on the loss function by implementing a specific variant of stochastic gradient descent (SGD)

Backpropagation

- Backpropagation is used to compute gradients.
- Algorithm typically has the following steps
  - Initialize random weights $\sim \; N(0, \sigma^2)$
  - Loop until convergence
  - Compute gradient $\frac{\partial J(W)}{\partial W}$
  - Update weights $W_{new} \leftarrow W_{old} - \eta \frac{\partial J(W)}{\partial W}$
  - Return weights

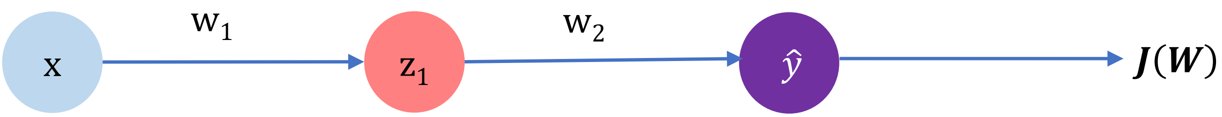
Vanishing or Exploding Gradient

- Vanishing gradient problem occurs when $0 < w < 1$
- Exploding gradient problem occurs when $w > 1$
- For a layer to experience this problem, there must be more weights that satisfy the condition for either vanishing or exploding gradients.

Gradient Clipping / Norm

- Basic idea is to set up a rule for avoiding vanishing or exploding gradients.
- Clip the derivatives of the loss function to a given threshold value if a gradient value is less than a negative threshold or more than the positive threshold.
  - Specify a threshold value; e.g. 0.5.
  - If the gradient value exceeds 0.5 or − 0.5 , then it will be either scaled back by the gradient norm or clipped back to the threshold value.
- Change the derivatives of the loss function to have a given vector norm when the L2 vector norm (sum of the squared values) of the gradient vector exceeds a threshold value.
  - If the vector norm for a gradient exceeds 1.0, then the values in the vector will be rescaled so that the norm of the vector equals 1.0

Backpropagation : Computing Gradients



$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Setting the Learning Rate

- There can be multiple local extremum
- Loss functions can be difficult to optimize
- Small learning rate converges slowly and gets stuck in false local minima
- Large learning rate overshoot and become unstable and diverge
- Selecting adaptive learning rates will address these issues
- Adaptive algorithm used in optimization : Adam, Adadelta, Adagrad, RMSProp

Mini-batches

- Gradient descent algorithms are computationally expensive
- One idea is to compute gradient using single data point → $\frac{\partial J_i(W)}{\partial W}$
- Single data point (SGD) computation can be very noisy
- Computing gradient by taking batch of points is a good practice

$$\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(W)}{\partial W}$$

- The true gradient is then the average of the gradient from each of those batches

Mini-batches

- Mini-batch ensure more accurate estimation of gradient and lead to fast training
- Batch Size = Size of Training Set → Batch Gradient Descent
- Batch Size = 1 → Stochastic Gradient Descent
- 1 < Batch Size < Size of Training Set → Mini-batch Gradient Descent

Problem of Overfitting

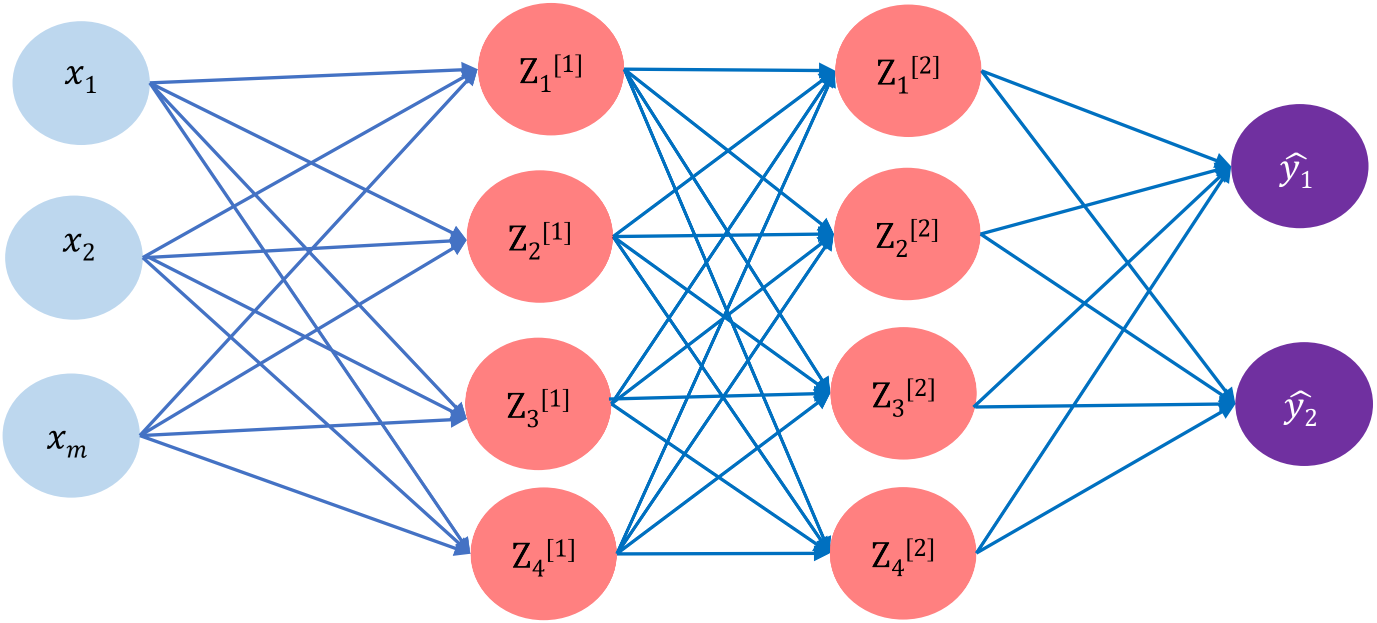
- Underfitting is the model that doesn't not have the capacity to fully learn from the data
- Overfitting is too complex and does not generalize well with the data as it starts to memorize the training data
- The process of fighting overfitting is called **Regularization**

Problem of Overfitting

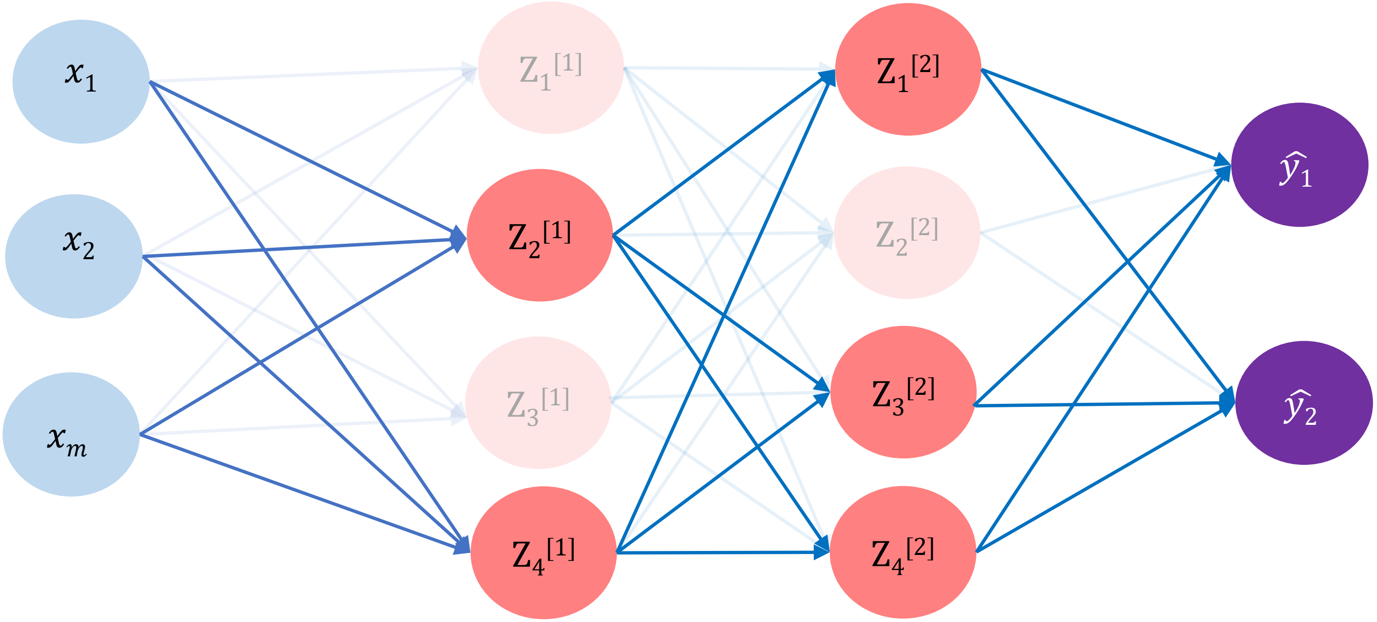
- **Regularization I: Dropout**
  - Randomly set some activations to zero
  - Typically drop 50% of activations in layer
  - Forces network to not rely on any one node

- **Regularization II: Early Stopping**
  - Stop training before there is a possibility of over-fitting

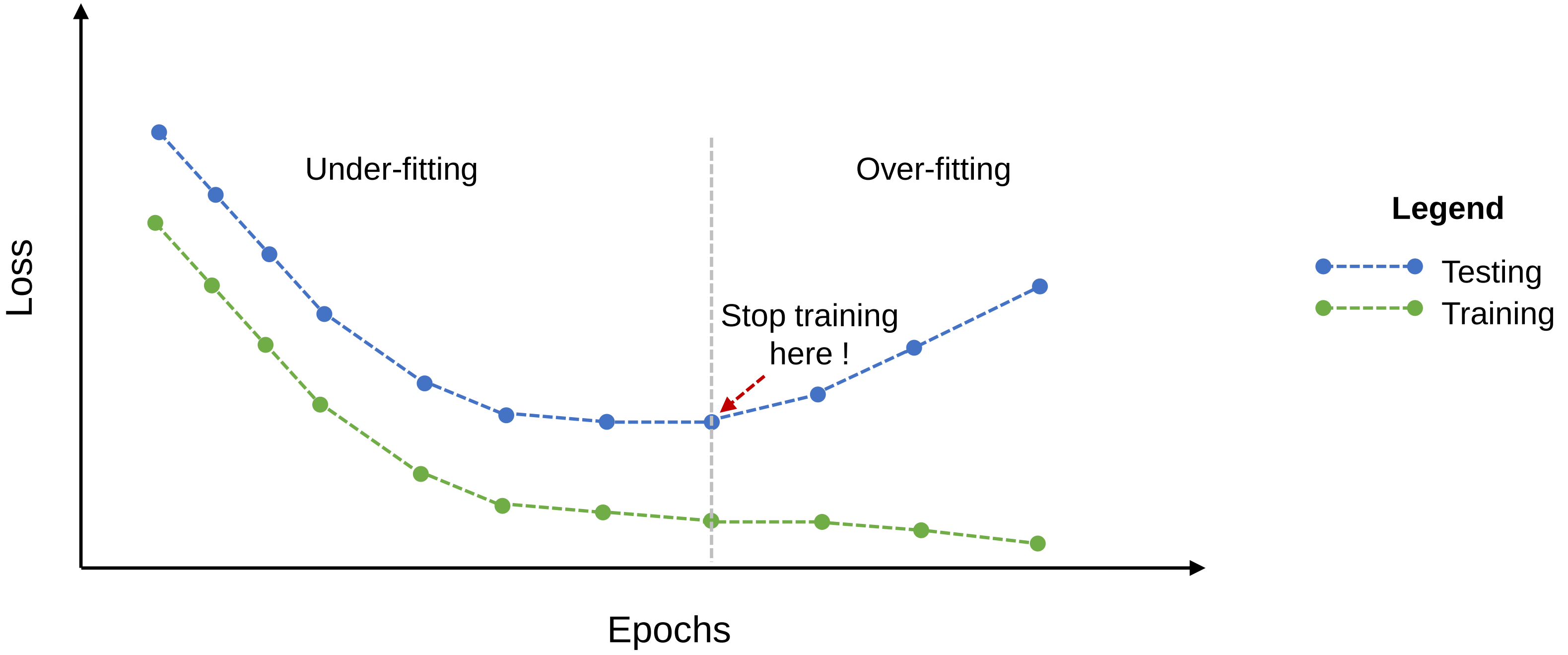
Regularization I : Dropout



Regularization I : Dropout



Regularization II : Early Stopping



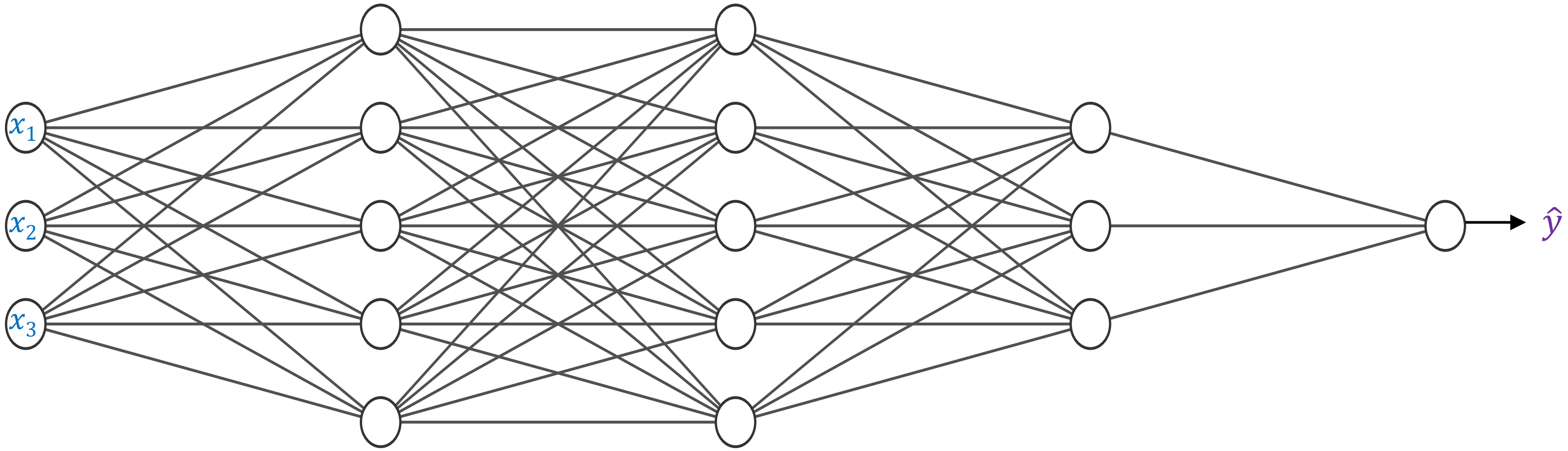
Neural Network Representation

- Shallow vs Deep
- Logistic regression are the simplest form of neural network which is a shallow model
- A multi hidden network of layers that are highly interconnected are an example of deep model
- A neural network with 1-hidden layer is a 2 layer neural network

Neural Network Representation

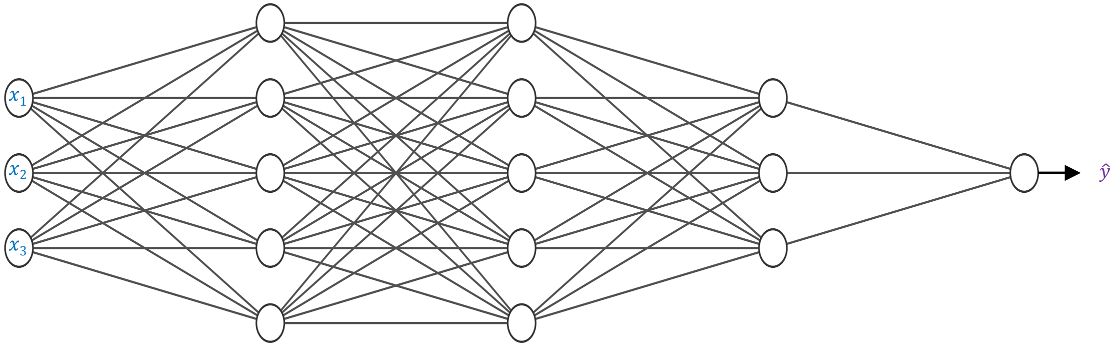
$L = Number\ of\ layers$

$n^{[l]} = Number\ of\ neurons\ in\ layer\ l$



$n^{[0]} = nx = 3$ $n^{[1]} = 5$ $n^{[2]} = 5$ $n^{[3]} = 3$ $n^{[4]} = n^{[L]} = 1$

Neural Network Dimensions



$$w^{[l]} = \left(n^{[l]}, n^{[l-1]} \right)$$
$$b^{[l]} = (n^{[l]}, 1)$$

$$z^{[1]} = w^{[1]} x + b^{[1]}$$

$$z^{[1]} = (5,1) \quad = \quad (5,3)\,(3,1) + b^{[1]}$$

$$z^{[1]} = \left(n^{[1]}, 1\right) \;=\; (n^{[1]}, 1)\;(n^{[0]}, 1)$$

$$a^{[1]} = g\left(z^{[1]}\right)$$

$$w^{[1]} = (n^{[1]}, n^{[0]})$$

$$w^{[2]} = \left(n^{[2]}, n^{[1]} \right) = (5,5)$$

$$z^{[2]} \;=\; w^{[2]}\, a^{[1]} + b^{[2]}$$

$$z^{[2]} \;=\; (5,1) = \; (5,5)\;(5,1) + b^{[2]}$$

$$a^{[2]} \;=\; g\left(z^{[2]}\right)$$

Neural Network Hyperparameters

- Some of the most common hyperparameters that can be optimized for better results
  - Number of hidden layers
  - Number of neurons
  - Choice of activation
  - Number of epochs
  - Learning rate
  - Mini-batch size
  - Regularization parameters

Deep Learning for Computer Vision

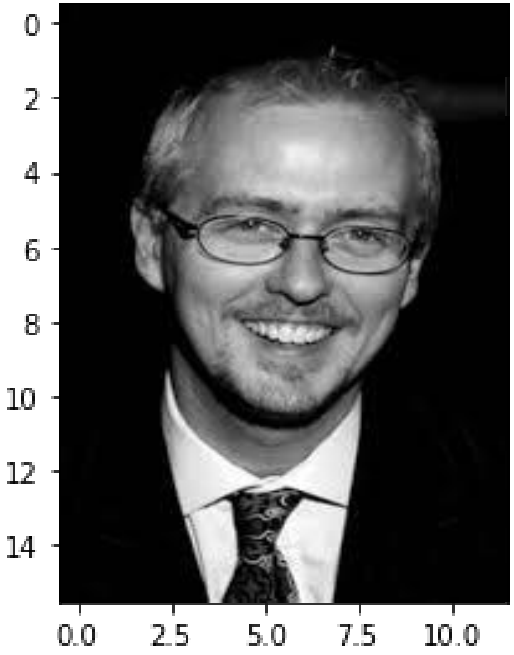
Convolutional Neural Network

Deep Learning for Computer Vision

- Computer Vision is one of the rapidly advancing fields
- Field of having a computer understand and label what is present in an image
- Giving machines a sense of vision
- What computers “see”?
- How does it process an image or video?
- Why Not a Fully connected Neural Network?

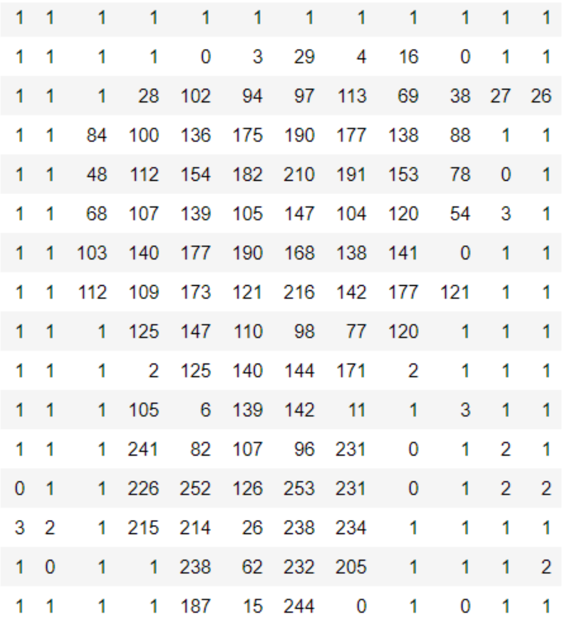
Images are Numbers

Input Image



An image is just a matrix of numbers [0, 255]

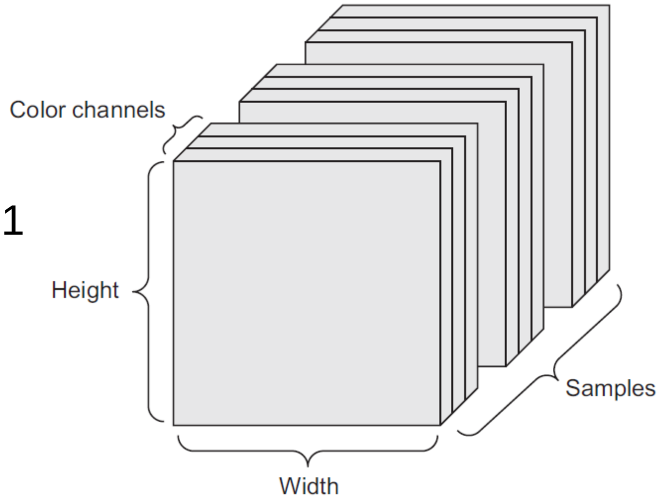
What the computer sees



1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	3	29	4	16	0	1	1
1	1	1	28	102	94	97	113	69	38	27	26
1	1	84	100	136	175	190	177	138	88	1	1
1	1	48	112	154	182	210	191	153	78	0	1
1	1	68	107	139	105	147	104	120	54	3	1
1	1	103	140	177	190	168	138	141	0	1	1
1	1	112	109	173	121	216	142	177	121	1	1
1	1	1	125	147	110	98	77	120	1	1	1
1	1	1	2	125	140	144	171	2	1	1	1
1	1	1	105	6	139	142	11	1	3	1	1
1	1	1	241	82	107	96	231	0	1	2	1
0	1	1	226	252	126	253	231	0	1	2	2
3	2	1	215	214	26	238	234	1	1	1	1
1	0	1	1	238	62	232	205	1	1	1	2
1	1	1	1	187	15	244	0	1	0	1	1

Image Representation in CV

- Three Dimension: height, width, and color channels
- Made up of pixels
- 28 x 28 greyscale image is represented as 28 x 28 x 1
- 28 x 28 color image is represented as 28 x 28 x 3
- Images are 4D tensors while video data are 5D

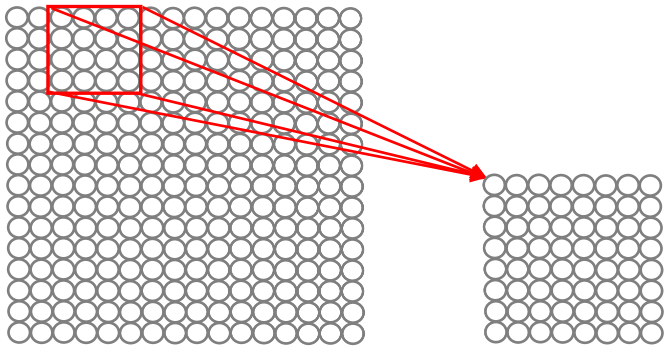


Adapted from Francois Chollet (2017), Deep Learning with Python

Learning Visual Features

- In fully connected or dense neural networks, each hidden layer is densely connected to its previous layer - every input is connected to every output in that layer
- In a densely connected network, the 2D input (spatial structure) is collapsed down into 1D vector which is fed into the dense network. Every pixel in that 1D vector will be feed into the next layer and in the process, we lose all of the very useful spatial structure of the image
- Deep learning on large images on a fully connected layers aren’t feasible
- Spatial structures are super important in image data and we need to preserve this

Feature Extraction with Convolution



- Filter size: 4 x 4
- 16 different weights
- Continue this filter to 4 x 4 patches in input
- Shift 2 pixels for next patch

- The ‘patch’ method is called **convolution**

- Edge detection to connect patch in input layers to a single neuron in subsequent layer
- Slide through the window to define connections and apply set of weights (weighted sum) to extract local features
- Use multiple filters – weights – to extract different features
- Spatially share the parameters of each filter to extract maximum spatial features

Feature Extraction with Convolution

- Process of adding each element of the image to its local neighbors, weighted by the filter
- One of the most import operations in signal and image processing
- Filter is a matrix of values whose size and values determine the transformation effect

Convolution Operation

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

3 x 3 Filter

3 x 3 Feature Map

5 x 5 Image

- Apply the 3 x 3 filter over the input image
- Perform element-wise multiplication
- Add the outputs

Vertical Edge Detection

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 x 6 Image

*

1	0	-1
1	0	-1
1	0	-1

3 x 3 Filter

=

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

4 x 4 Feature Map

- In vertical edge deduction, a vertical edge is a 3 x 3 region (in the above example), where there are bright pixels on the left and dark pixels on the right

Horizontal Edge Detection

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 30 & 10 & -10 & -30 \\ 30 & 10 & -10 & -30 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

6 x 6 Image

3 x 3 Filter

4 x 4 Feature Map

- In horizontal edge deduction, a horizontal edge is a 3 x 3 region (in the above example), where the pixels are relatively bright on top and dark in the bottom

Other Common Filters

1	0	-1
1	0	-1
1	0	-1

1	1	1
0	0	0
-1	-1	-1

Prewitt

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

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

Sobol

3	0	-3
10	0	-10
3	0	-3

3	10	3
0	0	0
-3	-10	-3

Scharr

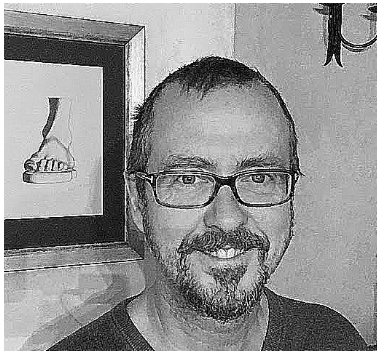
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Parametric Filter

(using back propagation)

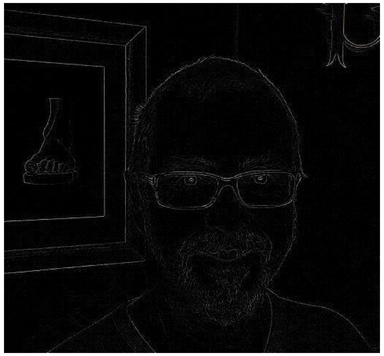
Producing Feature Maps

Original



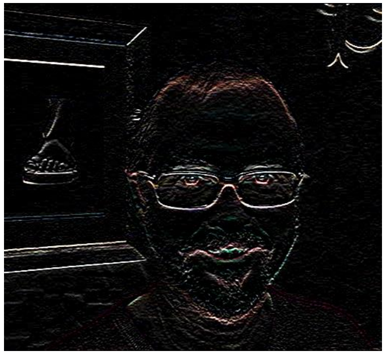
Sharpen

-1	-1	-1
-1	9	-1
-1	-1	-1



Edge Detect

0	1	0
-1	-4	1
0	1	0



‘Strong’ Edge Detect

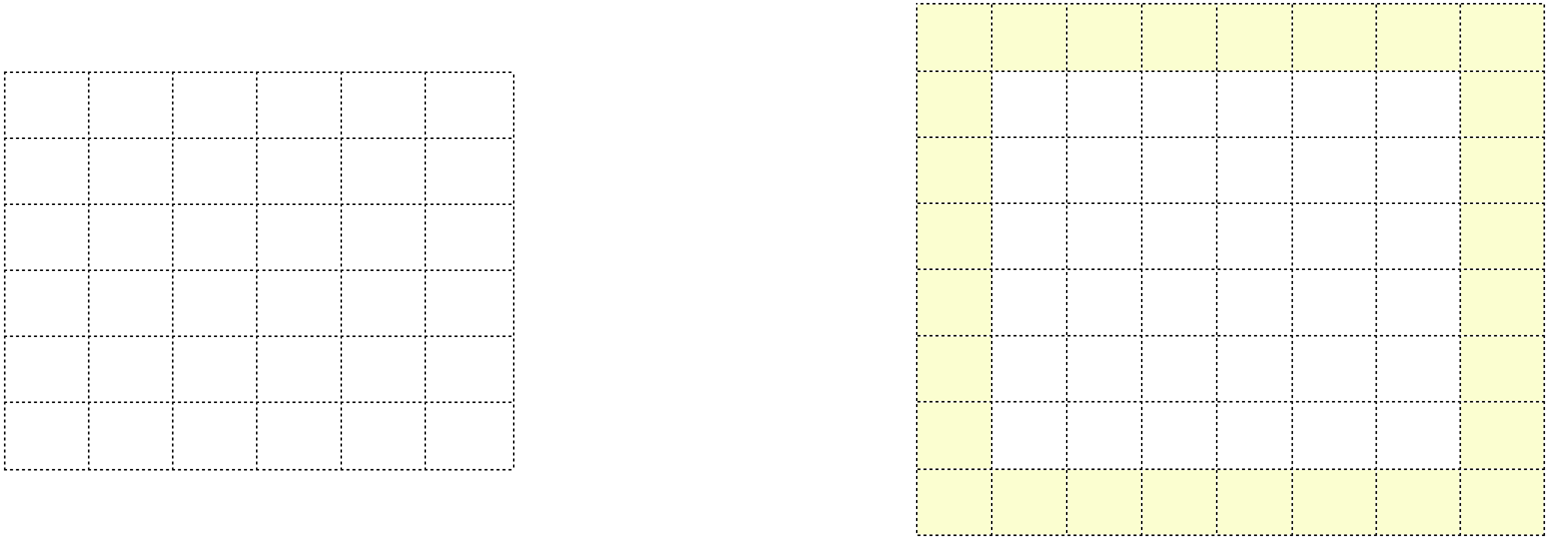
-1	-2	-1
0	0	0
1	2	1

Note: If the feature map contains negative values (black portion), one can convert negative values to non-negative values by applying ReLU activation functions, thus converting the black portions into grey.

Padding

- A ***n x n*** image with ***f x f*** filter will produce an output of ***[n-f+1] x [n-f+1]***
- On every convolutional operation (edge deduction), the image shrinks and we end up with very small image
- Information on the edges are used much less as compared to other parts of the image and we miss vital spatial information
- Padding before applying convolution operation help address this issues by adding one border of one pixel around the borders

Padding



Padding

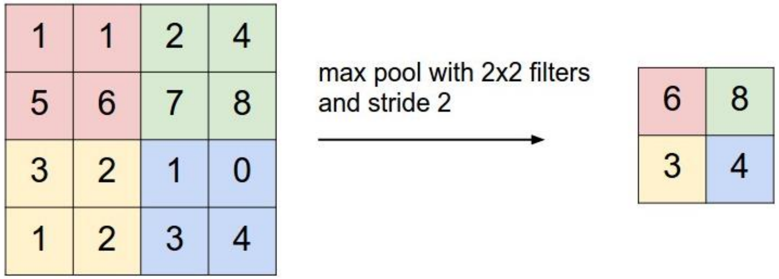
- A 6 x 6 image will then become 8 x 8 resulting into a 6 x 6 output, thus preserving the original input size
- By convention, we pad with zeros with one pixel as the padded amount (p = 1)
- The new output is of dimension ***[n+2p-f+1] x [n+2p-f+1]***
- Two common choices on how much to pad
  - Valid convolutions : no padding
  - Same convolutions : output size is same as the input size; $p = \frac{f-1}{2}$
- By convention, ***f*** is almost always an odd number

Strided Convolution

- Another piece of basic building block of convolutions
- Convolve with a stride of two (s=2); hop over two steps
- The new output is of dimension is $\left[\frac{n+2p-f}{s}+1\right] x \left[\frac{n+2P-f}{S}+1\right]$
- Round down the dimension if not an integer
- Filter must lie entirely within the image (or image plus the padded region)

Pooling

- Pooling down sample the image data extracted by the convolutional layers
- Reduces the dimensionality of the feature map in order to decrease the processing time
- Max Pooling extracts maximum value of the sub-regions of the feature map



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4

- For 3D inputs, the computation is done independently on each of the channels
- Average Pooling is used sometimes for very deep neural networks to collapse the representation

Convolutional Neural Networks

- Architecture designed for image classifications tasks
- Three parts to a CNN
  - **Convolution** : apply filters to generate feature maps by extracting features in the image or in the previous layers (generically)
  - **Non linearity** : apply non linear activation function – ReLU
  - **Pooling** : down sampling the spatial representation of the image to reduce dimensionality and to preserve spatial invariance
- Some classic CNN architectures are
  - LeNet, AlexNet, VGGNet, ResNet, GoogLeNet, XceptionNet, Fast R-CNN, U-Net, EfficientNet

Convolutional Neural Networks

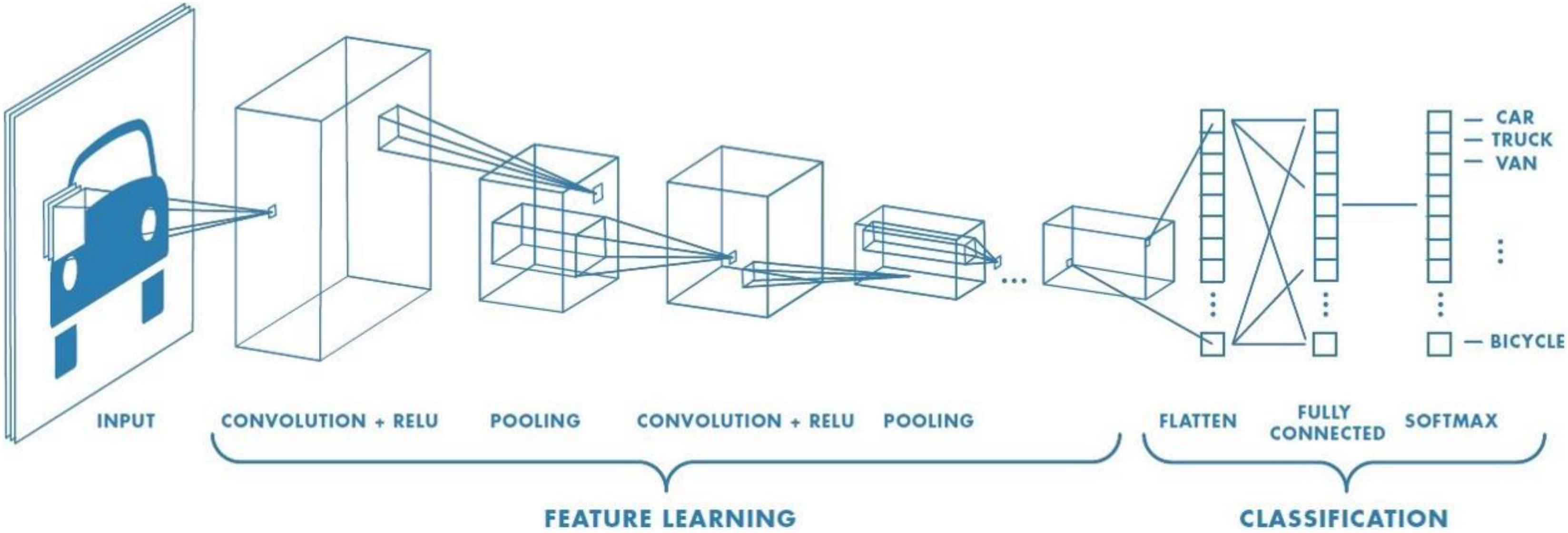


Image Source: mathworks.com

CNN : Key Takeaways

- Explicitly assume inputs are images
- Architecture designed for image classifications tasks
- Three parts to a CNN **Convolution, Non linearity, Pooling**
- Three hyperparameters - **Depth, Padding, Stride -** decides the output dimension
- Output dimension is given by $\left[\frac{n+2p-f}{s}+1\right] x \left[\frac{n+2P-f}{S}+1\right]$

Convolutions in Financial Time Series

- Convolutions are an unique type of Neural Networks which look data as a grid
- Applying convolutions to sequence data is an evolving idea
- Converts a non-image data into synthetic images
- Ensembe with traditional sequence models like LSTM to boost the model score
- A CNN-LSTM architecture uses CNN layers for features extraction combined with LSTM to support sequence prediction
  - Conv1D for univariate or multivariate time series
  - Conv@D if we have time series of images as input
  - CNN-LSTM ≠ ConvLSTM

Code Walkthrough

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))

model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu’))

model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu’))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

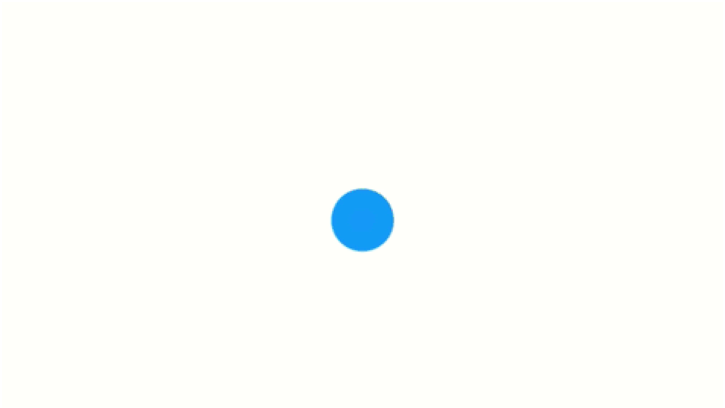
Code Walkthrough – Model Summary

```
Layer (type)                     Output Shape            Param #
=================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)      320
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)                (None, 11, 11, 64)      18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)                (None, 3, 3, 64)        36928
_________________________________________________________________
flatten_1 (Flatten)              (None, 576)             0
_________________________________________________________________
dense_1 (Dense)                  (None, 64)              36928
_________________________________________________________________
dense_2 (Dense)                  (None, 10)              650
=================================================================
Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```

Deep Sequence Modeling

Long Short Term Memory Network

Deep Sequence Modeling



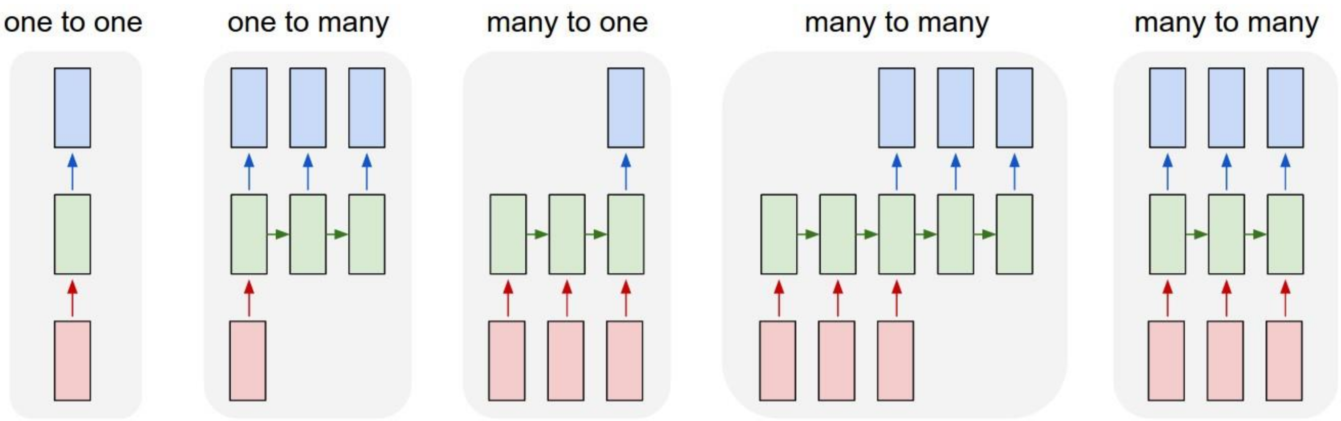
Deep Sequence Modeling

- Applying neural networks to problems involving sequential processing of data
- Sequence data comes in many forms : text, audio, video and financial time series
- Modeling to predict the next sequence of events (word, sound, time series)
- Effective for financial time series prediction
- Handle different types of network architecture
    - variable length sequence
    - track long-term dependencies
    - preserve information about the order and share parameters across the sequence

Recurrent Neural Network

- Generalization of feedforward neural network that has an internal memory
- Good at modeling sequence data
- RNN uses sequential memory for prediction
- Sequential memory is a mechanism used to identify the sequence patterns
- RNN are faster, uses less computational resources as there are less tensor operations

Recurrent for Sequence Modeling



[1] Feed Forward

[2] Sequence Output

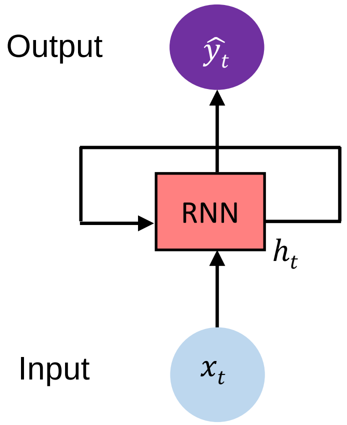
[3] Sequence Input

[4] Sequence Input & Output

[5] Synced sequence Input & Output

Adapted from Andrej Karpathy (2015), The Unreasonable Effectiveness of Recurrent Neural Networks

Short-term Memory



the clouds are dark, its about to …..

The sky is …

This is the 15th day of wildires in the bay area. There is smoke everywhere, it is showing ash and the sky is …

Problem of Short-term Memory

- Suffer from short-term memory
- Vanishing gradient due to the nature of back propagation algorithm
- Doesn’t learn long-range dependencies across time step

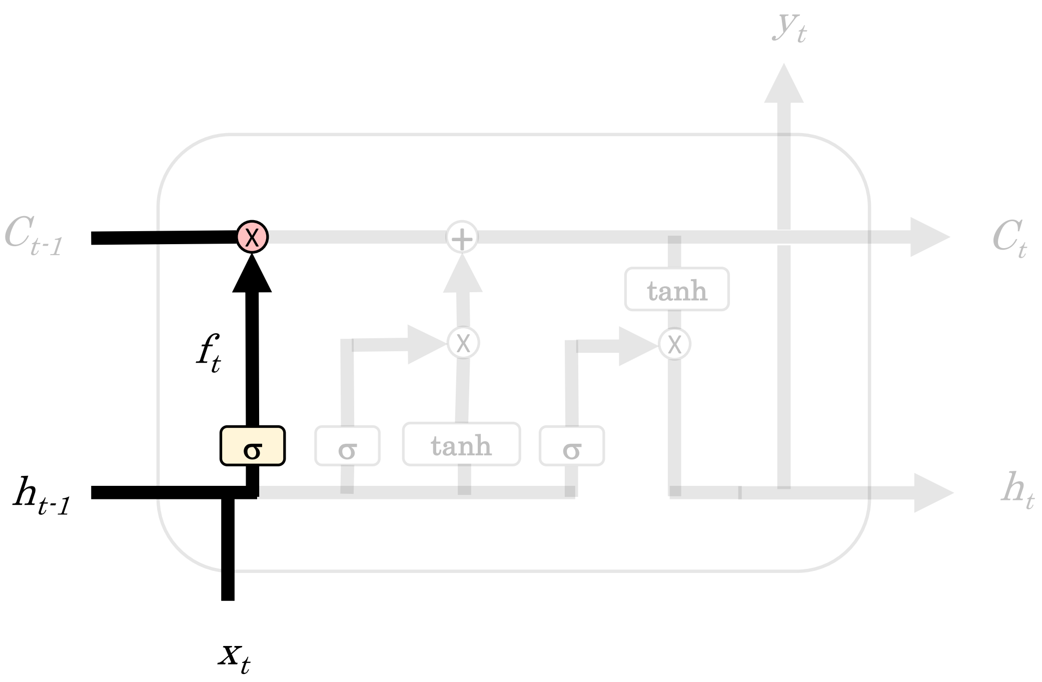
Long Short Term Memory Network

- LSTM algorithm is fundamental to deep learning for timeseries
- Special kind of RNN, explicitly designed to avoid the long-term dependency problem
- Widely used for sequence prediction problems and proved to be extremely effective
- LSTMs have four interacting layers

Long Short Term Memory Network

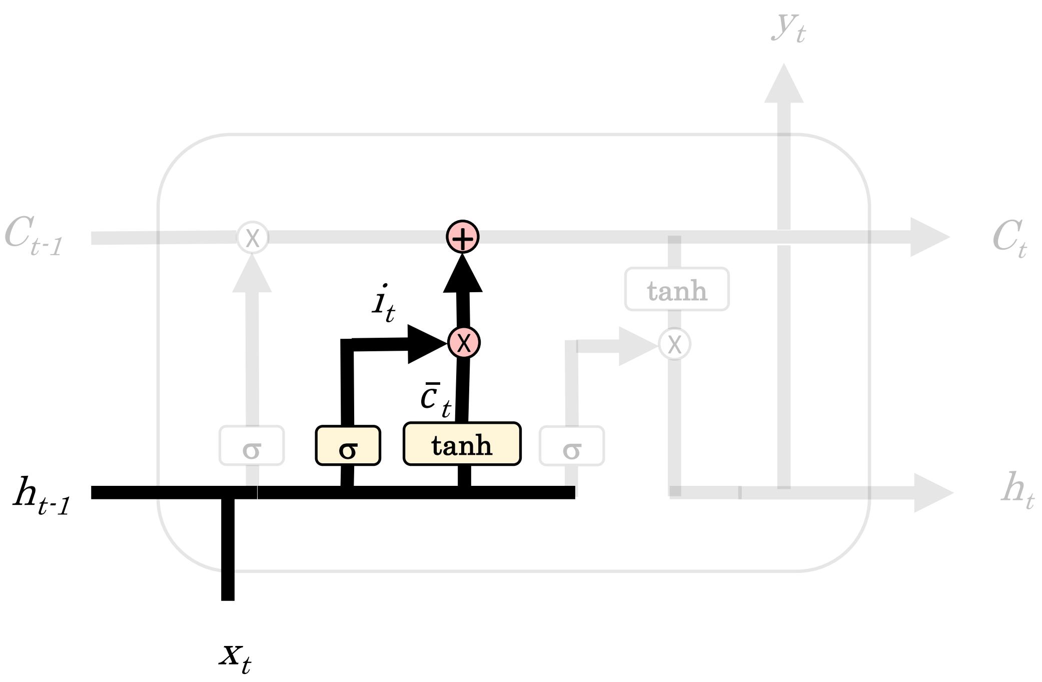
- Maintain a separate cell state from what is outputted
- Use gates to regulate the flow of information
  - Forget Gate
  - Input Gate
  - Update (Cell) State
  - Output Gate
- Uninterrupted gradient flow for backpropagation

LSTM : Forget Gate



$$f_t = \sigma\ (W_f\,.\,[h_{t-1,}\,x_t] + bf)$$

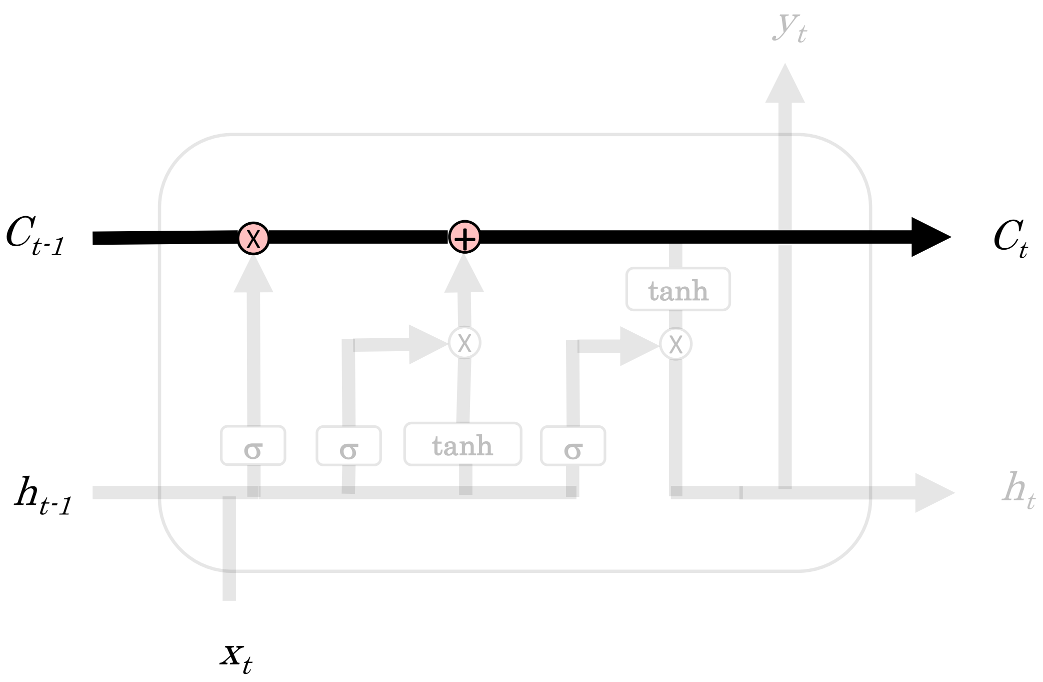
LSTM : Input Gate



$$i_t = \sigma\ (W_{i\,.}\ [h_{t-1,} x_t] + bi)$$

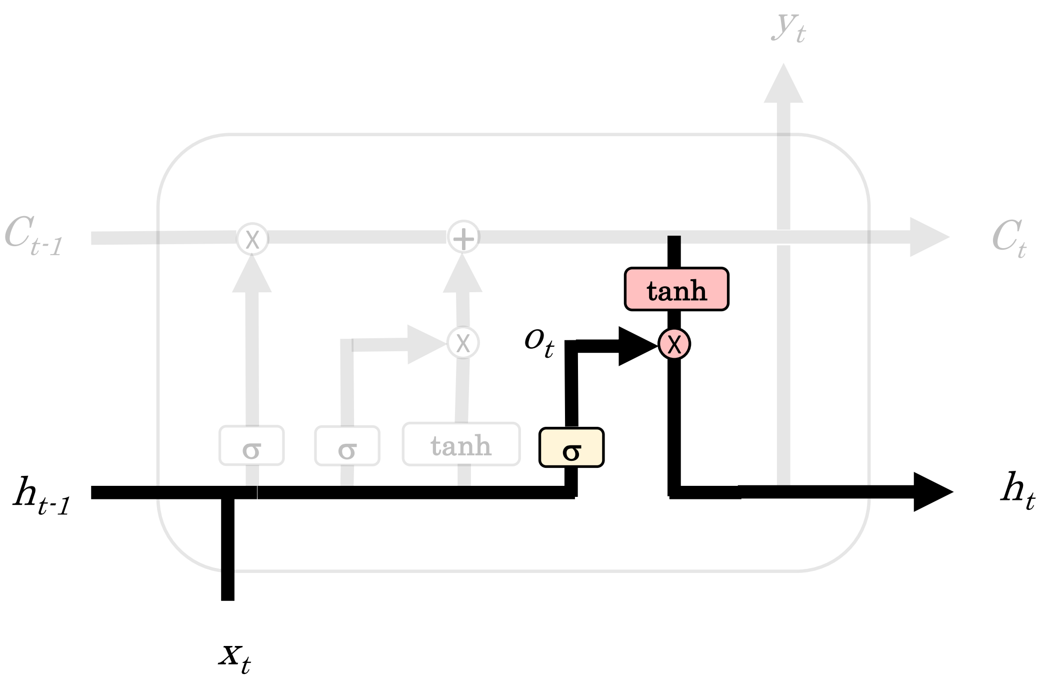
$$\bar{c}_t = tanh\ (W_{c\,.}\ [h_{t-1,} x_t] + b_c)$$

LSTM : Update Cell State



$$C_t = f_t * Ct_{-1} + it * \bar{c}_t$$

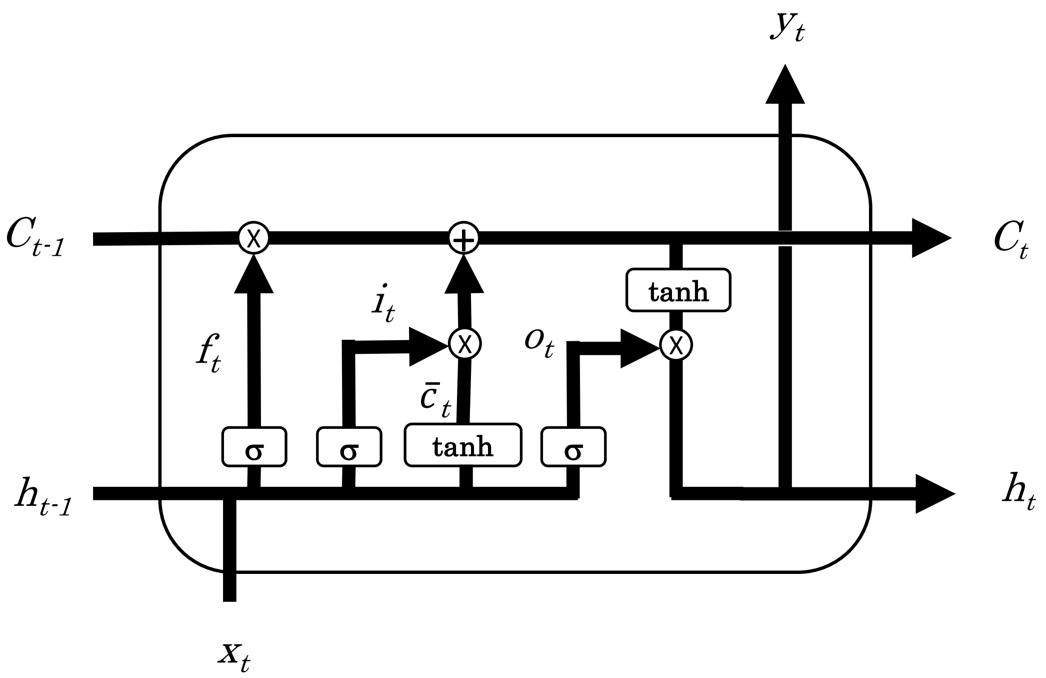
LSTM : Output Gate



$$o_t = \sigma\ (W_{o\ .}\ [h_{t-1,} x_t] + bo)$$

$$h_t = o_t * tanh\ (C_t)$$

LSTM Network



$$f_t = \sigma\,(W_f\,.\,[h_{t-1,}x_t] + bf)$$

$$i_t = \sigma\,(W_i\,.\,[h_{t-1,}x_t] + bi)$$

$$\bar{c}_t = tanh\,(W_c\,.\,[h_{t-1,}x_t] + b_c)$$

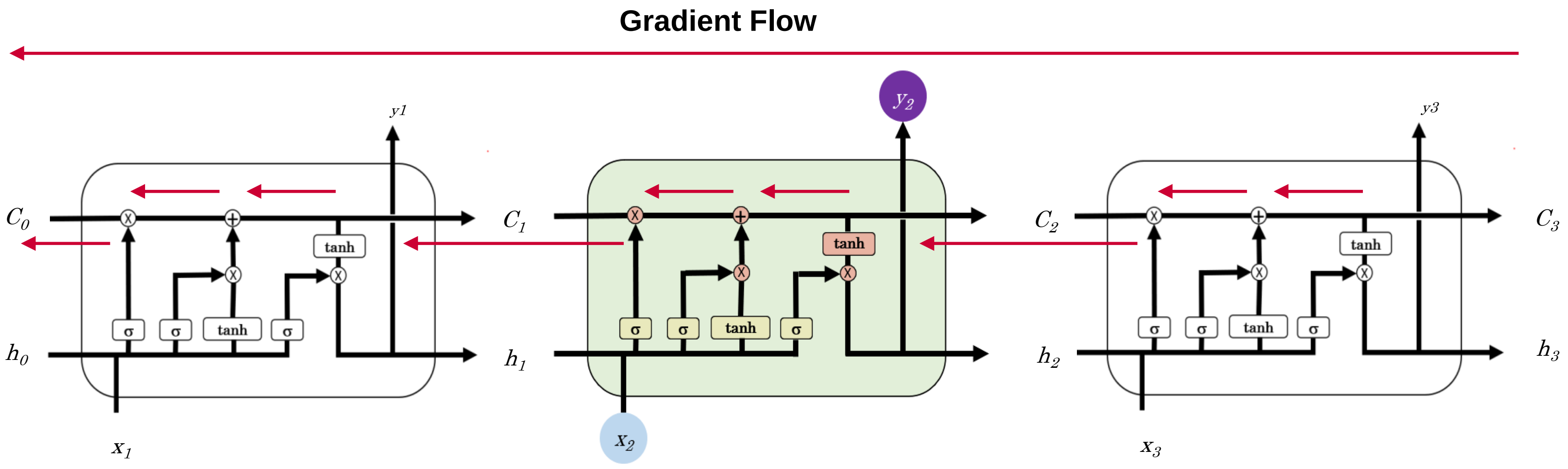
$$C_t = f_t\,*Ct_{-1} + \;it\;*\bar{c}_t$$

$$o_t = \sigma\,(W_o\,.\,[h_{t-1,}x_t] + bo)$$

$$h_t = o_t * tanh\,(C_t)$$

LSTM Gradient Flow

- LSTM network is comprised of different memory blocks called cells or units.



Code Walkthrough

```
model = Sequential()

#Add first layer
model.add(LSTM(units=256, input_shape = (60,1), return_sequences=True))
model.add(Dropout(0.4))

#Add second layer
model.add(LSTM(units=256, return_sequences=False))
model.add(Dropout(0.4))

#Add a Dense layer
model.add(Dense(64,  activation = 'relu'))

#Add the output layer - output layer
model.add(Dense(1))
```

Code Walkthrough – Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_1 (LSTM)                (None, 60, 256)           264192    
_________________________________________________________________
dropout_1 (Dropout)          (None, 60, 256)           0         
_________________________________________________________________
lstm_2 (LSTM)                (None, 256)               525312    
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 64)                16448     
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65        
=================================================================
Total params: 806,017
Trainable params: 806,017
Non-trainable params: 0
```

Generative Adversarial Networks

Generator – Discriminator Network

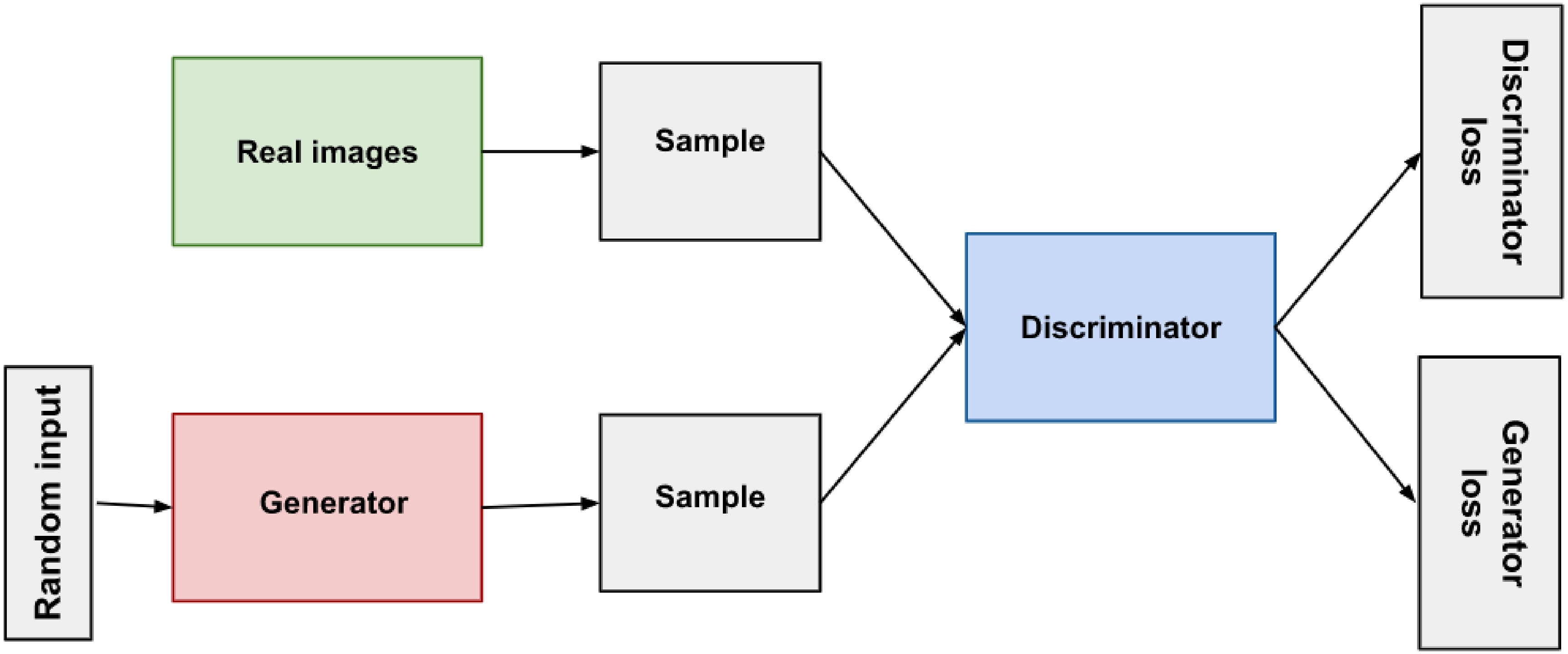
Generative Adversarial Networks

- Generative models
- Neural network that mimic a given distribution of the data
- Generate content such as images, text similar to what a human can produce
- Objective is to find the hidden laten meaning
- Foundational level insights of explanatory factors behind the data
- Impressive results on image and video generation such as style transfer using CycleGAN, and human face generation using StyleGAN
- Different types of GAN : Vanilla, CGAN, DCGAN, LAPGAN, SRGAN

Generative Adversarial Networks

- Consist of two neural networks
    - Generator : trained to generate new data from the problem domain
    - Discriminator : trained to distinguish fake data from real data
- Most applications in NNs are implemented using **discriminative models**
- GANs are part of a different class of models known as **generative models**
- Discriminative models learn the conditional probability $P(y|x)$
- Generative models capture the join probability $P(x,y)$ or $P(x)$, if there are not labels
- Unlike discriminative models, generative models are used for both supervised and unsupervised learning

GAN Structure

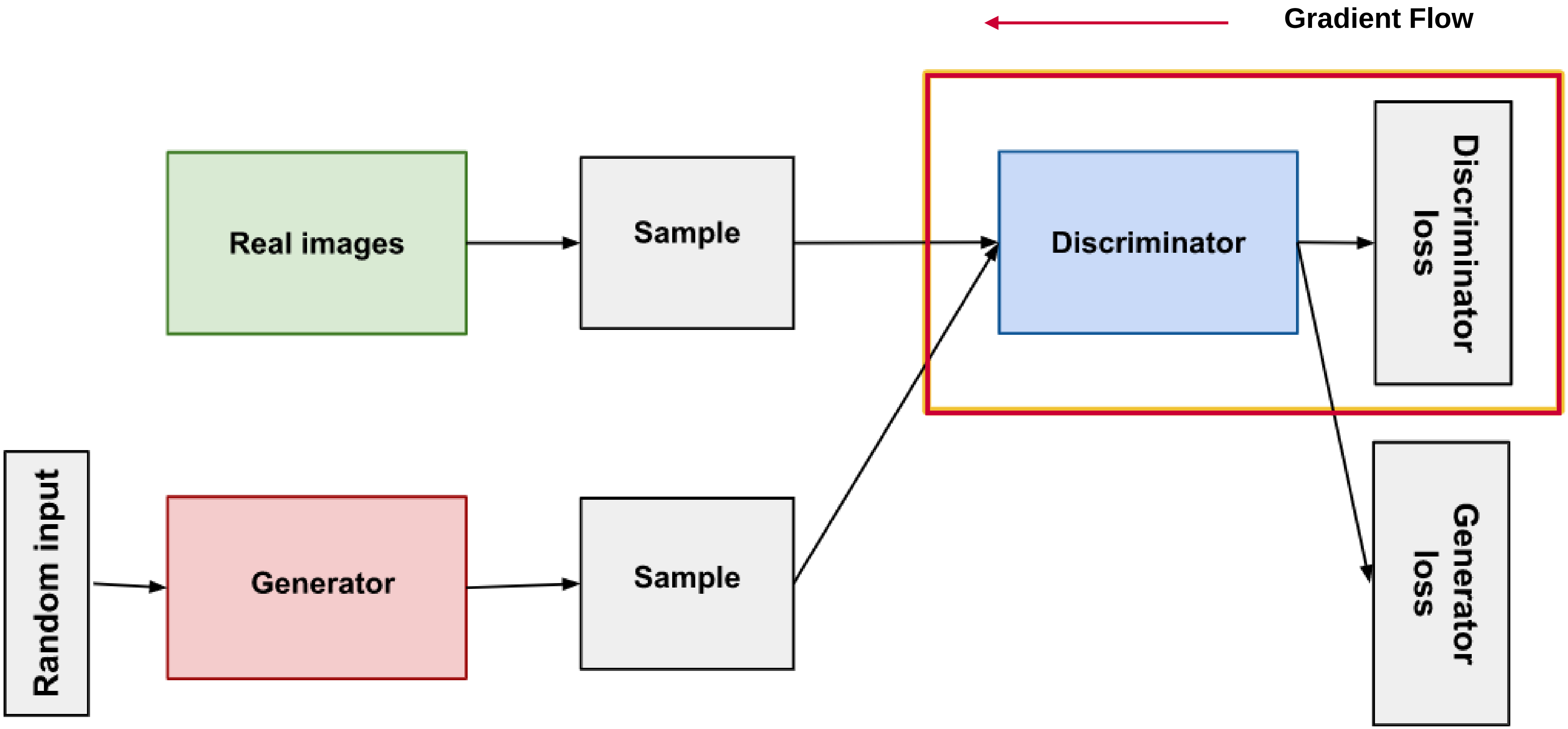


Adapted from Google Developers, Generative Adversarial Networks

Five Steps to GAN

- Define GAN architecture (based on the application)
- Train discriminator to distinguish real vs fake data
- Train the generator to fake data that can fool the discriminator
- Continue training both discriminator and generator for multiple epochs
- Save the generator model to create new, realistically fake data

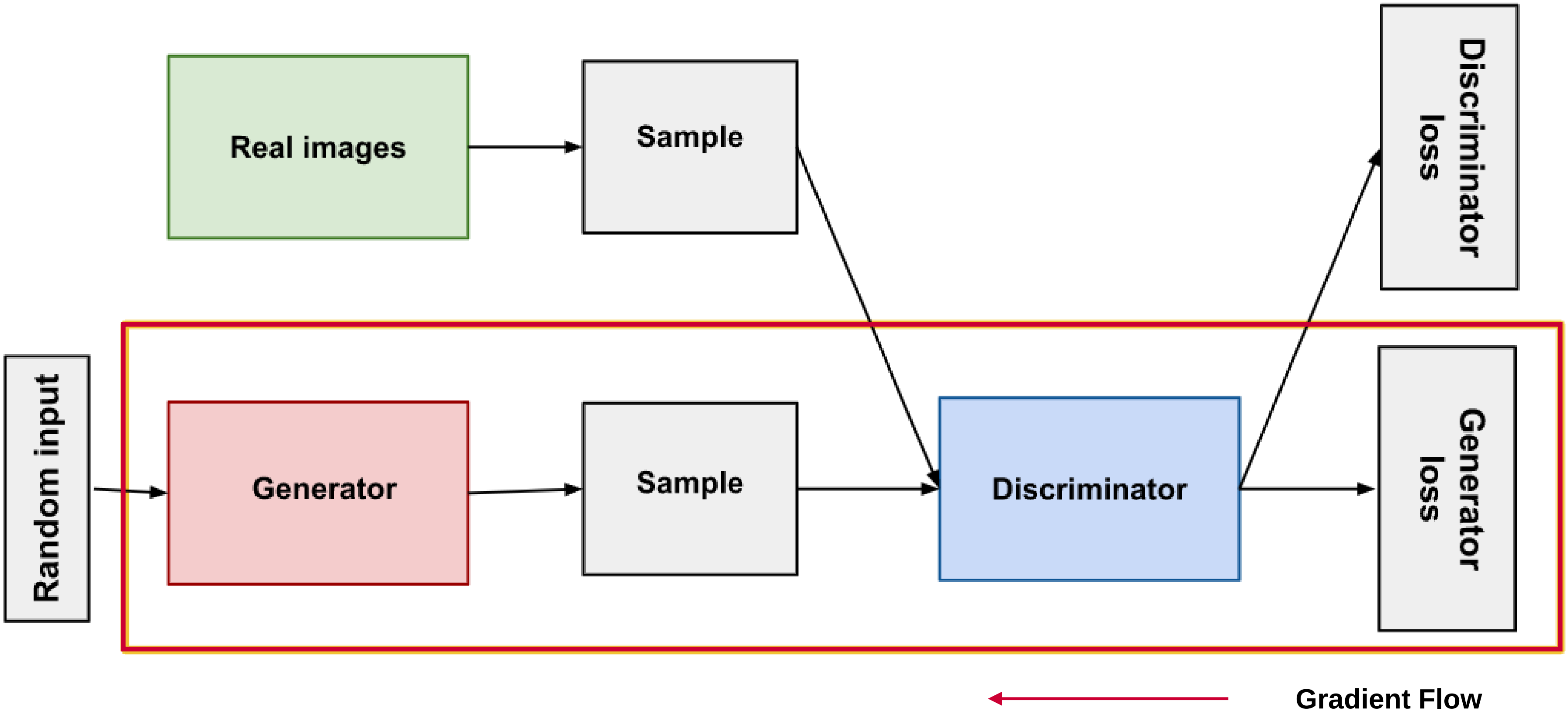
GAN : Discriminator



Note: Hold the generator values constant when training the discriminator and discriminator values constant when training the generator. Each of these should be trained against static adversary.

Adapted from Google Developers, Generative Adversarial Networks

GAN : Generator



Note: Hold the generator values constant when training the discriminator and discriminator values constant when training the generator. Each of these should be trained against static adversary.

Adapted from Google Developers, Generative Adversarial Networks

GAN Loss Functions

- Try to replicate a probability distribution
- Loss functions measure distance between the distribution by the GAN and the distribution of the real data
- Two common loss functions are Minimax loss and Wasserstein loss
- **Minimax** : Generator minimize and discriminator maximize the following function

$$E_x[log(D(x))] + Ez[log(1 - D(G(z)))]$$

- **Wasserstein** : depends on WGAN where discriminator does not classify instances
    - Critic Loss $D(x) - D(G(z))$
    - Generator Loss $D(G(z))$

Pros & Cons of Deep Learning

- Deep Learning algorithms are versatile and scalable
- Preferred for high dimensionality, complex and sequential problems
- Learns from data Incrementally, layer-by-layer and jointly
- Automating feature engineering is the key highlight of Deep Learning
- Most Machine Learning algorithms used in industry aren’t Deep Learning algorithms
- Deep Learning isn’t always the right tool as there may not be enough data available for deep learning to be applicable and/or can better be solved by a different algorithms

Limitations of Neural Networks

- Data Hungry
- Computationally intensive to train and deploy
- Subject to algorithmic bias
- Fooled by adversarial examples
- Requires expert knowledge to design and tune architectures
- Hype and Promise of AI : The AI Winters

References

- Chigozie, Winifred, Anthony, and Stephen (2018), Activation Functions: Comparison of Trends in Practice and Research for Deep Learning
- Francois Chollet (2017), Deep Learning with Python
- Andrej Karpathy (2015), The Unreasonable Effectiveness of Recurrent Neural Networks
- Christopher Olah (2015), Understanding LSTM Networks
- Michale Phi (2018), Illustrated Guide to LSTM’s and GRU’s: A step by step explanation
- Standford University, Massachusetts Institute of Technology, Technische Universität München, Notes on Artificial Intelligence
- TensorFlow, Keras, API Documentation
- Google Developers, Generative Adversarial Networks

Note: Some of the materials from above resources are adopted for these notes under CC-BY-SA 4.0. For details interpretation on the subject, refer to the above resources.