TENSOR FLOW BASIC PRINCIPLE

* Define a graph of computations to perform and then Tensor Flow takes that graph and runs it efficiently using optimized C++ code
* It is possible to break up the graph into several chunks and run them in parallel across multiple CPU or GPUs
* Tensor Flow developed by GOOGLE BRAIN TEAM
* Tensor flow "auto diff" feature can automatically and efficient by compute the gradients for you
* Tensor Flow operations (OPS) can take any number of Input And produce any number of outputs. For example, the addition and multiplication ops each take two inputs & produce one output.

eg

```
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

* Constants and Variables take no input (source ops). The Input and outputs are multi dimensional array called tensors.

Feeding Data to the Training Algorithm:

* placeholder These nodes are special because they don't actually perform any Computation, they just output the data you tell them to output at runtime.

4D-Tensor

* [mini-batch Size, Height, Width, Channels]

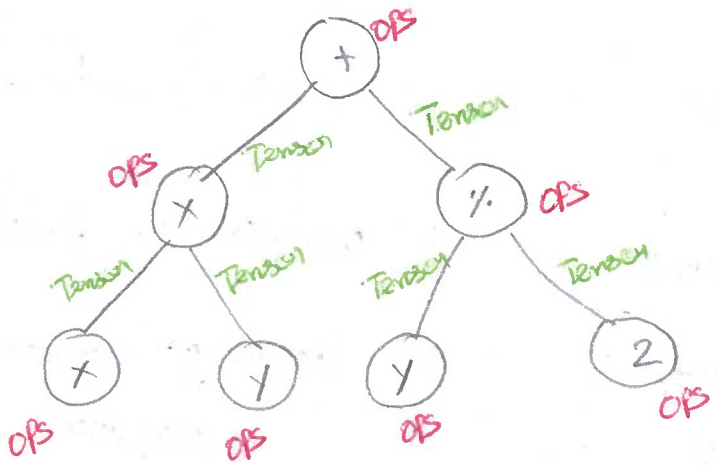
140 256 256 1

How to create a Tensor?

1) operations (ops)

2) Tensor

The operation is the one that results in the tensor



In the Computational graph

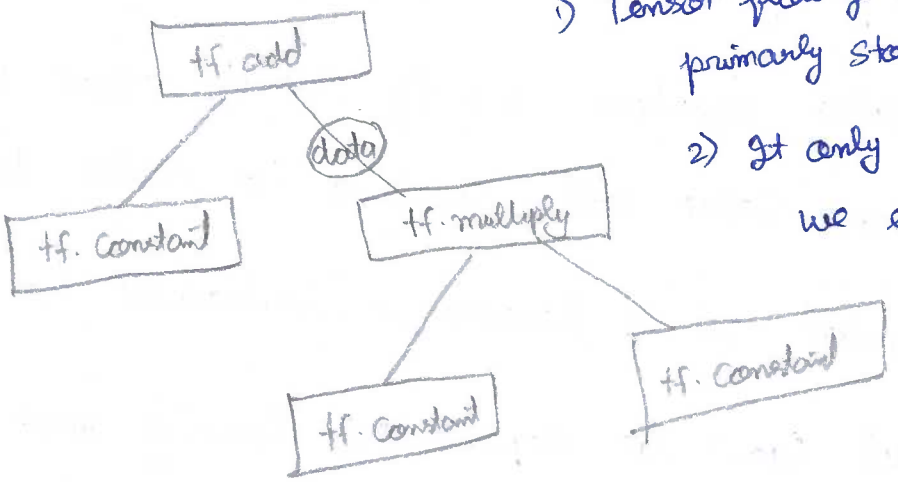
1) OPS are the nodes

2) Tensors are the edges

Hello = tf.Constant(name = 'op1', Value = 'Hello world')

Hello — Tensor

tf.Constant(...) — OPS



1) Tensor flow graph doesn't primarily store data

2) It only yield result wh
we execute sess.run
along with th
Input argume

Lazy Execution	Eager Execution
1) execution on Sess.run()	1) execution on function call
2) explicit session scope is required	2) defining Session is no longer required

0- tensor (SCALAR)

1- tensor (VECTOR)

2- tensor (MATRIX)

Random Tensors:

Many machine learning algorithms learn by performing updates to a set of tensors that hold weights. These updates equations usually satisfy that weights initialized at the same value will continue to evolve together. Thus, if the initial set of tensors is initialized to a constant value the model won't be capable of learning much.

tf. expand_dims → Convert a vector into a row vector or column vector

tf. Squeeze → Remove all dimension of size 1 from a tensor (Convert row or column vector into a flat vector)

TENSOR FLOW GRAPHS :-

Any Computation in TensorFlow is represented as an Instance of tf.Graph object. Such graph consist of a set of instances of tf.Tensor objects and tf.operation objects

What is tf.operation ?

A call to an operation like "tf. matmul" creates a "tf.operation" instance to make the need to perform the matrix multiplication operation

TENSOR FLOW VARIABLE

Stateful Computation → Learning algorithms are essentially rules for updating stored tensors.

tf.Variable() class provides a wrapper around tensors which allows for stateful computation

[NOTE] Variables have to be explicitly initialized

How to update the value of an existing variable ?

tf.assign()

GRAPH, SESSION and FETCHES

* Tensor Flow Involves two main phases

  1) CONSTRUCTING A GRAPH

  2) EXECUTING THE GRAPH

* A Session object is the part of the Tensor-Flow API that Communicates between Python objects and ~~at~~ data on our end, and the actual Computational System where memory is allocated for the objects we define, intermediate variables are stored and finally results are fetches for us.

OPTIMIZER :

The next thing we need to figure out is how to MINIMIZE the Loss function

optimizers update the set of weights Iteratively in a way that decreases the loss over time.

The Most Commonly used approach is GRADIENT DESCENT

* While Convergence to the GLOBAL MINIMUM is guaranteed for CONVEX FUNCTION, for NON-CONVEX problems they can get Stuck in the LOCAL MINIMA

Difference between CONVEX and NON-CONVEX OPTIMIZATION?

A Convex optimization Problem maintains the property of linear programming problem

There can be only one optimal solution, which is globally optimal or you might prove that there is no feasible solution to the Problem

A NON-CONVEX problem maintains the property of non-linear programming problem

They can have multiple locally optimal points and it can take a lot of time to identify whether the problem has no solution or if the solution is global.

It might lead you to DEAD END

tf.name_scope(): This is used to group together parts that are related to in passing the output, like defining the loss, and setting and creating the training object.

Input Size: [None, 28, 28, 1]

1) None: we have an unknown number of Images
2) each 28×28 pixel
3) 1 color channel (since these are gray scale Images)

TWO POPULAR EXAMPLE OF GAN IMPLEMENTATION:

1) Deep Convolutional GAN (DCGAN)
2) Conditional GAN (CGAN)

GAN can generate NEW celebrity faces that are not of real people by performing latent space interpolations

GANs are able to learn how to model the Input distributions by training two competing (and cooperating) networks referred to as GENERATOR and DISCRIMINATOR (Sometimes known as CRITIC).

GENERATOR: the role of the generator is to keep figuring out how to generate fake data or signals that can fool the discrimina

DISCRIMINATOR: Trained to distinguish between fake and real signals.

As the training progresses, the discriminator will no longer be able to see the differences between the synthetically generated data and the real ones.

From these, the discriminator can be DISCARDED and the generator can now be used to create new realistic signals that have never been observed before.

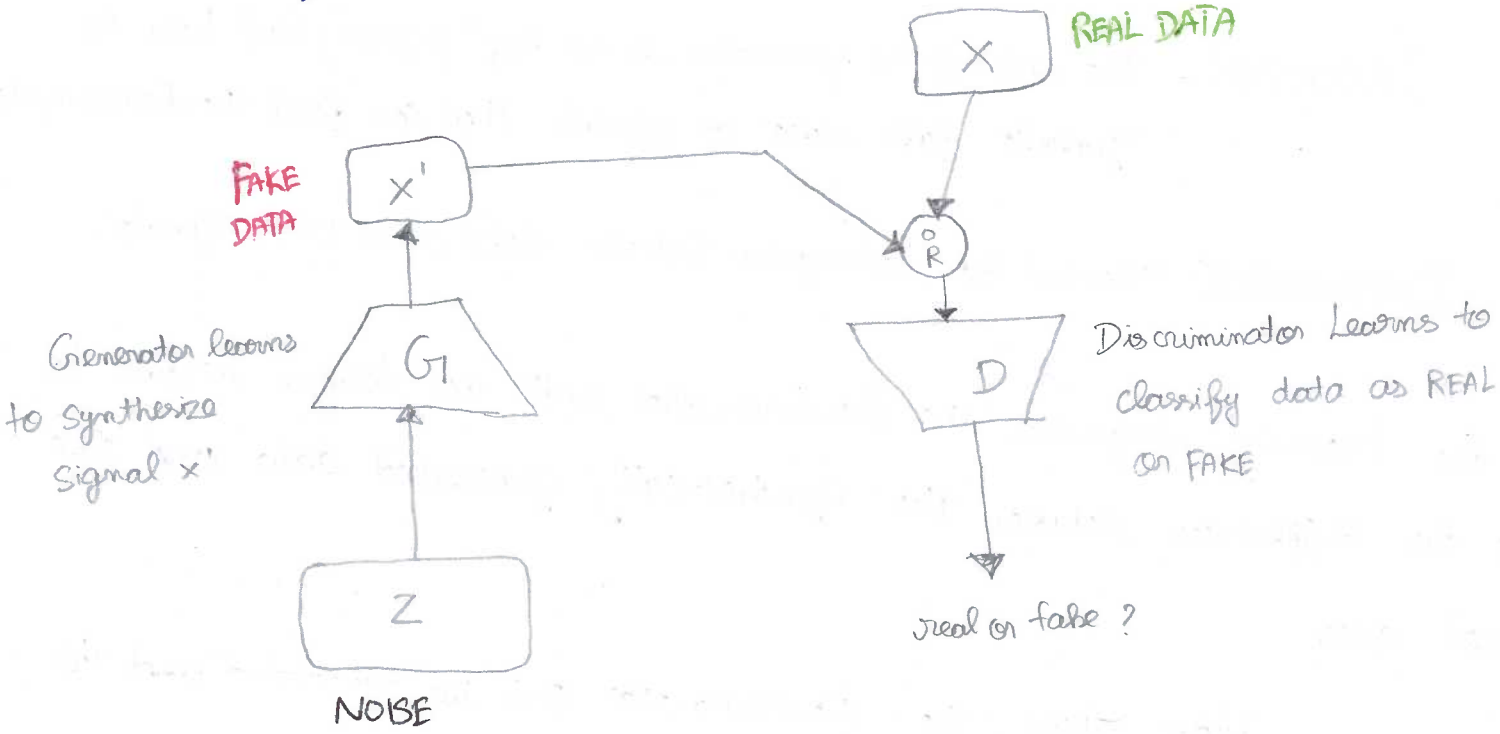
CHALLENGE IN GAN:

1) There must be a HEALTHY competition between the generator and the discriminator in order for both networks to be able to learn simultaneously

2) LOSS FUNCTION is computed from the output of the discriminator its parameter update is fast

S♦ ♥♦♦♦ ♦♦ ♦

1) When the discriminator converges faster, the generator no longer receives sufficient GRADIENT UPDATES for its parameters and fails to converge.



PRINCIPLES OF GANs

1) DISCRIMINATORS are taught how to determine if a data is REAL or FAKE

2) SAMPLES of Real and fake data are used to train the DISCRIMINATOR

3) From time to time the GENERATOR will attempt to pretend that it generates REAL data

4) Initially, the DISCRIMINATOR will not be fooled and will tell the GENERATOR why the data is FAKE.

5) Taking into consideration this feedback, the GENERATOR hones his skills again and attemp to produce new Fake data.

6) The DISCRIMINATOR again will be able to both spot the data as Fake and Justify why the data are fake.

7) As training progresses the GENERATOR masters his skills in making fake datas that are indistinguishable from real ones.

8) the GENERATOR can then infinitely print fake datas without getting caught by the DISCRIMINATOR as they are no longer indefinable as FAKE.

---

* Since the Labeling process is automated, GANs are still considered part of the UNSUPERVISED LEARNING approach

* The OBJECTIVE of the DISCRIMINATOR is to learn from this supplied dataset on how to distinguish REAL and FAKE data. During this part of GAN training, only the DISCRIMINATOR parameters will be updated.

* Like a typical binary classifier, the DISCRIMINATOR is trained to predict on a orange of 0.0 to 1.0 in confidence values on how close a given input signal is to the True one.

* At regular Intervals, the GENERATOR will pretend that its output is a genuine data and will ask the GAN to label it as 1.0

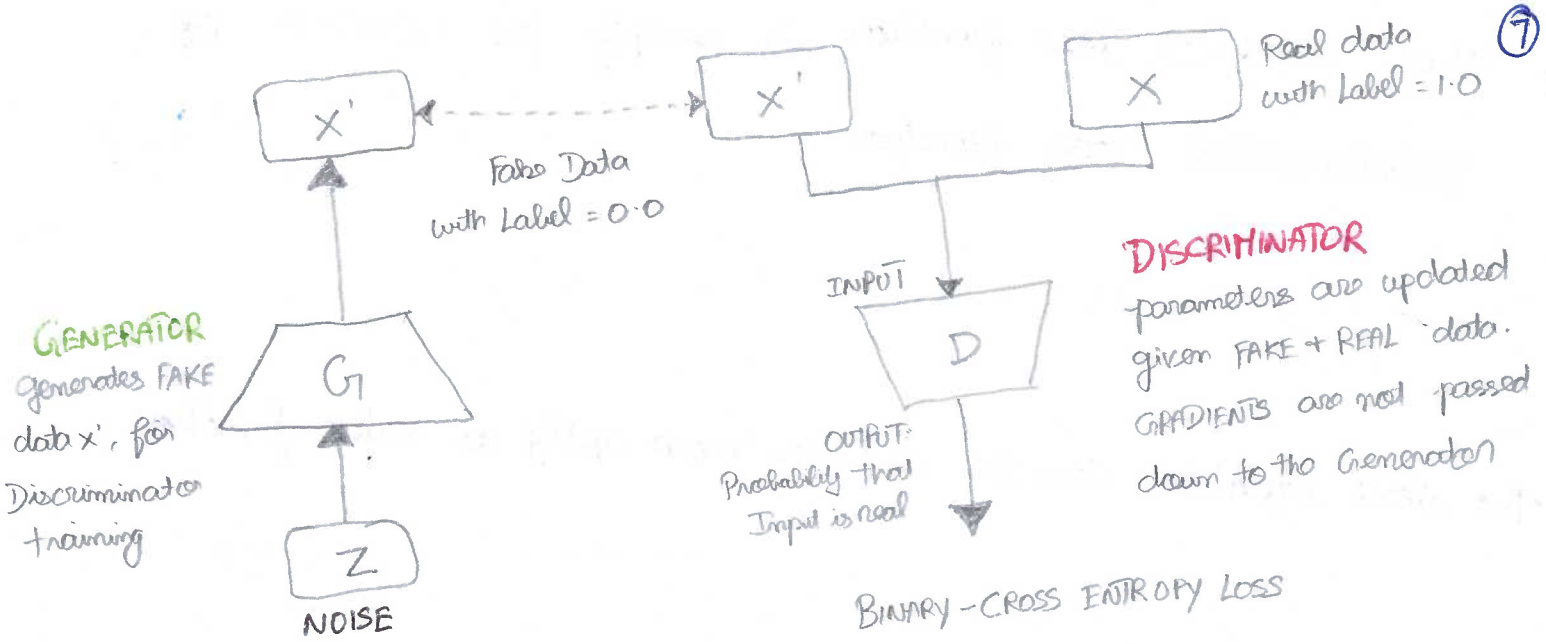
* The GENERATOR will use the gradients to update its parameters and improve its ability to synthesize fake signals.

* OVERALL, the whole process is akin to two networks COMPETING with one another while still CO-OPERATING at the same time.

* When the GAN training CONVERGES, the end result is a GENERATOR that can synthesize signals.

* the DISCRIMINATOR thinks these synthesized signal / data are real or with a label near 1.0, which means the DISCRIMINATOR can then be discarded.

* The GENERATOR part will be useful in producing meaningful outputs from arbitrary noise inputs.



As shown in the preceding figure, the DISCRIMINATOR can be trained by MINIMIZING the LOSS FUNCTION

$$L^{(D)}\left(\theta^{(G)}, \theta^{(D)}\right) = \underbrace{-E_{x \sim p_{data}} \overbrace{\log D(x)}^{\text{DISCRIMINATOR OUTPUT FOR REAL DATA "x"}} - E_z \log\left(1 - \overbrace{D(G(z))}^{\text{DISCRIMINATOR OUTPUT FOR GENERATED FAKE DATA G(z)}}\right)}_{\text{STANDARD BINARY CROSS-ENTROPY COST FUNCTION}}$$

* In order to minimize the loss function, the DISCRIMINATOR parameter $\theta^{(D)}$ will be updated through BACK-PROPAGATION by correctly identifying the genuine data $D(x)$ and synthetic data $1 - D(G(z))$
* Correctly identifying real data is equivalent to $D(x) \rightarrow 1.0$
* Correctly identifying false data is equivalent to $D(G(z)) \rightarrow 0.0$
* Z is the arbitrary encoding or noise vector that is used by GENERATOR to synthesize new signal

* the GENERATOR loss function is simply the NEGATIVE of DISCRIMINATOR loss function

$$L^{(G)}(\theta^{(G)}, \theta^{(D)}) = -L^{(D)}(\theta^{(G)}, \theta^{(D)})$$

the above equation can be re-written more aptly as value function

$$V^{(G)}(\theta^{(G)}, \theta^{(D)}) = -L^{(D)}(\theta^{(G)}, \theta^{(D)}) \quad \text{— (1)}$$

1) From the perspective of the GENERATOR, the above equation (1) should be MINIMIZED such that $D(G(z))$ is close to 1 (Discriminator is fooled into thinking that $G(z)$ is real)

2) From the perspective of the DISCRIMINATOR, the above equation (1) should be MAXIMIZED such that $D(x)$ is close to 1 and $D(G(z))$ is close to 0

3) Therefore, the generator training criterion can be written as a minimax problem

$$\theta^{(G)*} = \arg\min_{\theta^{(G)}} \min_{\theta^{(D)}} V^{(D)}(\theta^{(G)}, \theta^{(D)})$$

4) occasionally, we will try to "FOOL" the discriminator by pretending that the synthetic data is real with label 1.0

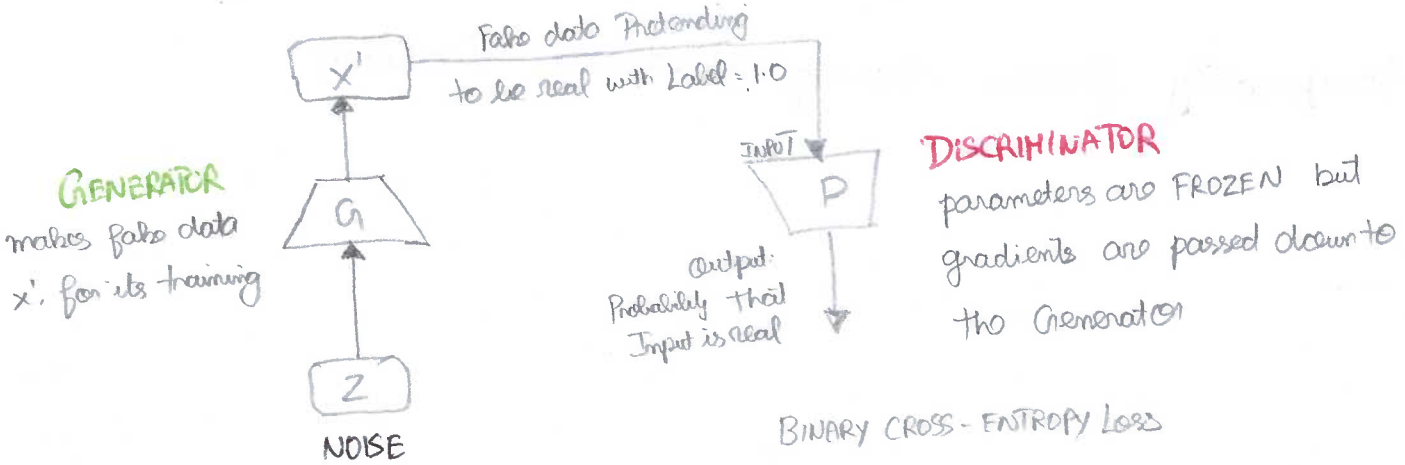
5) By MAXIMIZING with respect to $\theta^{(D)}$, the optimizer sends gradient updates to the DISCRIMINATOR parameters to consider this synthetic data as REAL.

6) At the same time by MINIMIZING with respect to $\theta^{(G)}$, the optimizer will train the GENERATOR parameters on how to trick the DISCRIMINATOR.

7) However, in practice, the DISCRIMINATOR is confident in its prediction in classifying the synthetic data as FAKE and will not update its parameters.

8) Furthermore, the gradient updates are small and have diminished significantly as they propagate to the GENERATOR layers

9) As a result, the GENERATOR fails to converge.



10) the solution is to reformulate the loss function of the GENERATOR

$$L^{(G)}(\theta^{(G)}, \theta^{(D)}) = -E_z \log D(G(z))$$

11) the above Loss FUNCTION simply MAXIMIZES the chance of the DISCRIMINATOR into believing that the Synthetic data is real by training the GENERATOR.

12) the new formulation is no longer Zero-SUM and is purely heuristics-driven.

13) In figure (previous page), the GENERATOR parameters are only updated when the whole adversarial network is trained. this is because the gradients are passed down from the discriminator to the generator.

14) However, in practice the DISCRIMINATOR weights are only temporarily frozen during adversarial training.

GAN TRAINING ALGORITHM :

for number of training iterations do

  for K steps do

    • Sample minibatch of "m" noise samples $\{z^{(1)}, \dots z^{(m)}\}$ ← Label 0.0

    • Sample minibatch of "m" real examples $\{x^{(1)}, \dots x^{(m)}\}$ ← Label 1.0

      $P_{data}(x)$ { for a mini-batch, first the discriminator is trained on REAL and FAKE data }

    • update the DISCRIMINATOR by ascending its stochastic gradient

DISCRIMINATOR NETWORK

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log\left(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))\right) \right]$$

  end for { new batch of FAKE Images }

    • Sample minibatch of "m" noise samples $\{z^{(1)}, \dots z^{(m)}\}$ ← Label 1.0

    • update the GENERATOR by ascending its stochastic gradient

ADVERSARIAL NETWORK

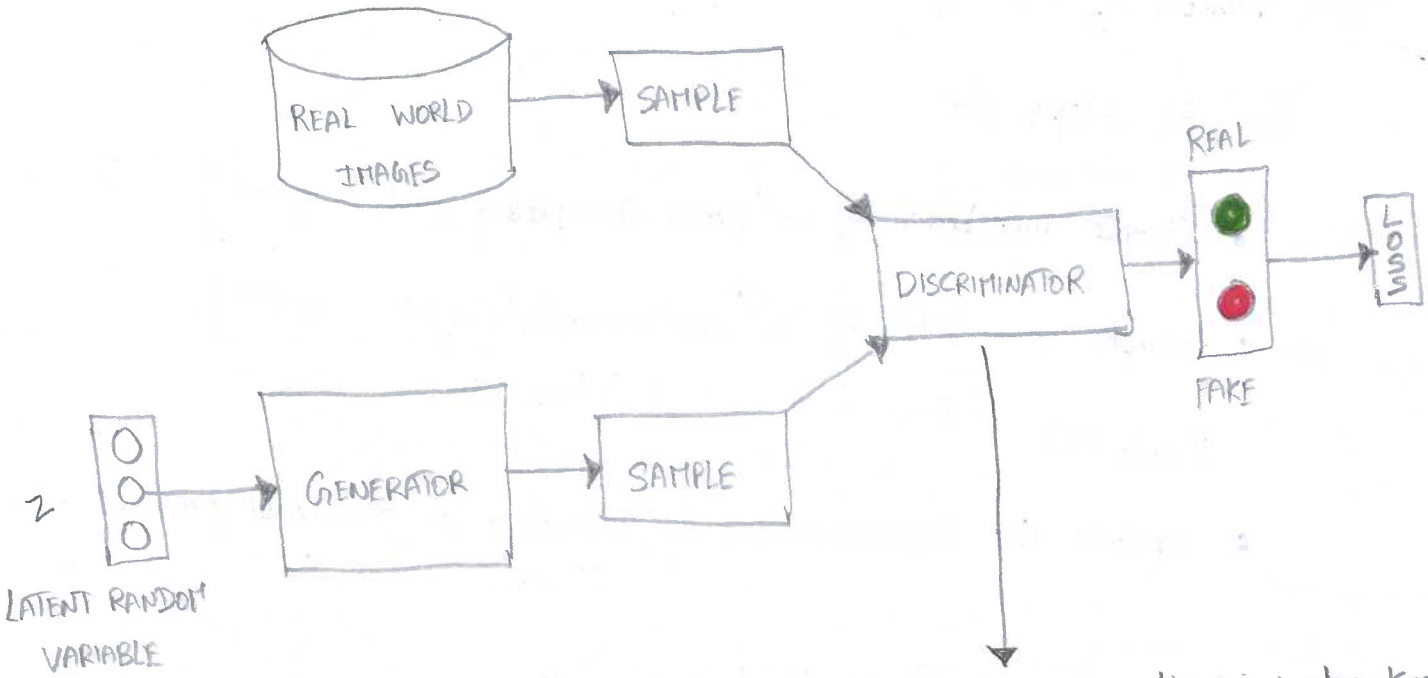
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D_{\theta_d}(G_{\theta_g}(z^{(i)}))\right)$$

→ Here "DISCRIMINATOR" weights are Frozen

  end for

{ K → Some people train the discriminator twice and train the generator once }

ADVERSARIAL LEARNING



(How can the discriminator know whether the data is REAL or FAKE ? you have to first train the DISCRIMINATOR)

1) the Job of the "GENERATOR" is to make Images, such that the "DISCRIMINATOR" cannot recognize it as FAKE

2) "DISCRIMINATOR" train itself to be better at detecting FAKE Images

3) SO "GENERATOR" and "DISCRIMINATOR" competes with each other

4) you train both of them at the same time

min max V(D, G)
G D
G(z)
max V(D, G)
D
min V(D, G)
G
G(z)
max V(D, G)
D
min V(D, G)
G
G(z)
max V(D, G)
D
G(z)

BATCH NORMALIZATION :- Batch Normalization is used to stabilize learning by normalizing the input to each layer to have zero mean and unit variance

Leakey ReLU :- Unlike ReLU instead of zeroing out all outputs when the input is less than zero, Leaky ReLU generates a small gradient equal to alpha x Input

What is Conv2DTranspose ?

We can Imagine transposed CNN (Conv2DTranspose) as the reversed process of CNN. In a simple example, If a CNN converts an IMAGE to FEATURE MAPS, a transposed CNN will produce an IMAGE given a FEATURE MAPS. Transposed CNN are used in decoders and also on generators.

ADVERSARIAL MODEL :- After building the GENERATOR and DISCRIMINATOR model, the ADVERSARIAL model is made by concatenating the generator and discriminator networks

DCGAN (Deep convolutional Generative Adversarial Networks) :-

```
# build GENERATOR model

# build DISCRIMINATOR model

# build ADVERSARIAL model

# ADVERSARIAL = GENERATOR + DISCRIMINATOR
```

```
adversarial = Model (inputs, discriminator (generator (inputs)),
                     name = model_name)
```

Adversarial network training :-

Adversarial training is nothing but GENERATOR and DISCRIMINATOR training, with discriminator weights FROZEN

* The generator is trained via an adversarial network
* Alternatively train DISCRIMINATOR and ADVERSARIAL networks by batch. Discriminator is trained first with properly real and fake Images. Adversarial is trained next with fake Images pretending to be real.

$$\min_G \max_D V(D, G)$$

$$V(D,G) = E_{x \sim P_{data}(x)}\left[\log D(x)\right] + E_{z \sim P_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

G = Generator

D = Discriminator

$P_{data}(x)$ = distribution of real data

$P(z)$ = distribution of generator

x = Sample from $P_{data}(x)$

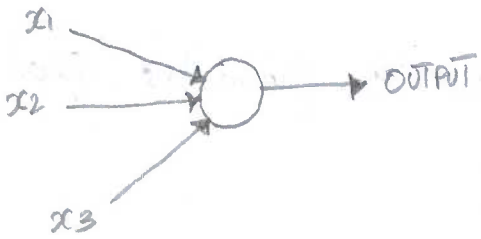
z = Sample from $P(z)$

$D(x)$ = Discriminator network

$G(z)$ = Generator network

How Perceptrons Work?

A perceptron takes several binary inputs $x_1, x_2, \ldots$ and produces a single binary output.

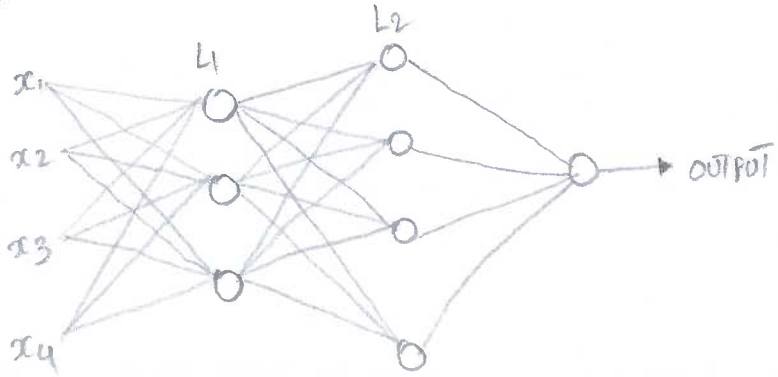


What are weights?

Weights $w_1, w_2 \ldots w_{3c}$ are real numbers expressing the ~~respective~~ IMPORTANCE of the respective inputs to the output.

The neuron's output 0 or 1 is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some Threshold Value

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



In this network, the first column of perceptrons - first layer of perceptrons is making 3 very simple decision by weighing the Input evidence.

The Perceptrons in the 2nd Layer is making a decision by weighing up the results from the first layer of decision - making.

In this way, the perceptrons in the 2nd Layer can make a decision at a more complex and abstract level thatn perceptrons in the first layer.

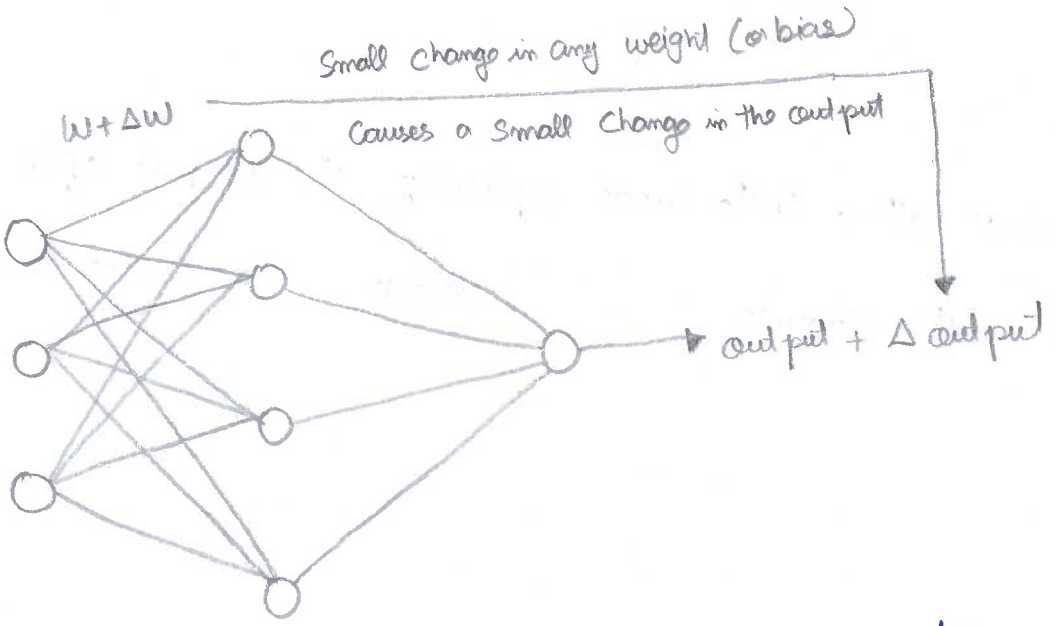
In this way, a many-layer network of perceptrons can engage in Sophisticated decision making

What are bias?

1) You can think of bias as a measure of how easy it is to get the perceptron to output a 1 or, the bias is a measure of how easy it is to get the perceptron to fire

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

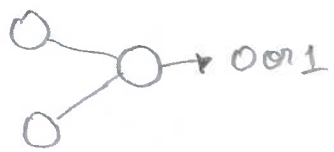
* It turns out that we can device LEARNING ALGORITHM which can automatically tune the weights and biases of a network of artificial neurons



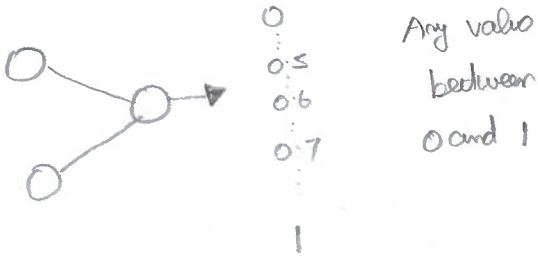
* If it we true that a small change in weight or bias causes only a small change in the output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

What is the difference between Perceptron & Sigmoid neuron?

Perceptron output



Sigmoid neuron output



$\sigma(w \cdot x + b)$ where σ is called SIGMOID FUNCTION

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

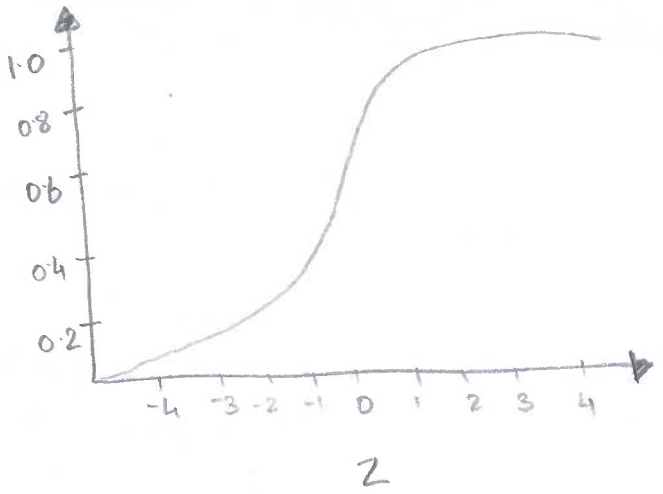
To put it all a little more explicitly, the output of a Sigmoid neuron with Inputs $x_1, x_2 \cdots$, weights $w_1, w_2 \cdots$ and bias b is

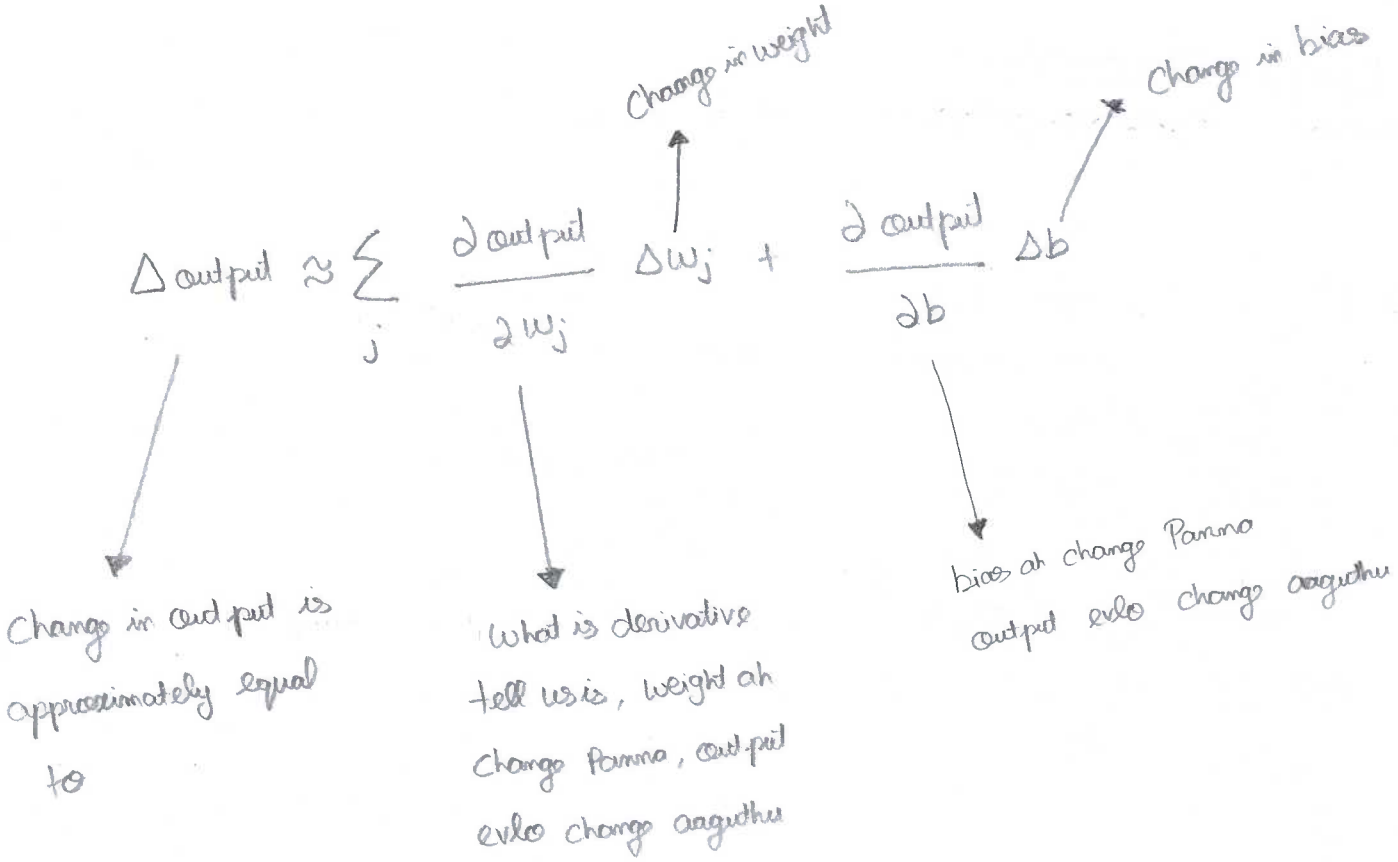
$$\frac{1}{1 + \exp\left(-\sum w_j x_j - b\right)}$$

If
$z = w \cdot x + b$ is a Large Positive number $\approx$ $\sigma(z) \approx 1$

If
$z = w \cdot x + b$ is a very negative number $\approx$ $\sigma(z) \approx 0$

SIGMOID FUNCTION

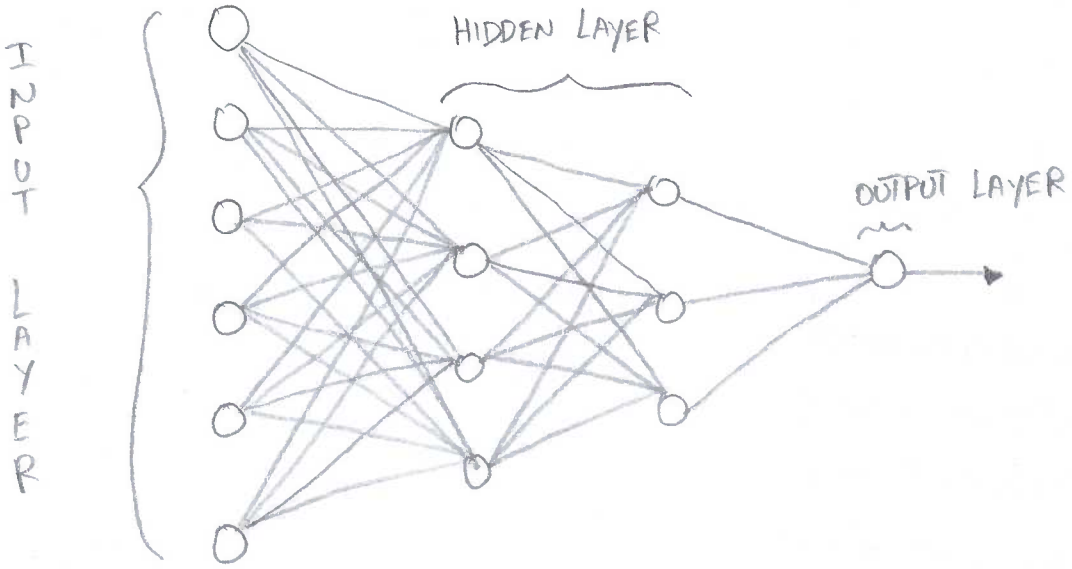




How should we interpret the output from a SIGMOID NEURON ?

One big difference between the Perceptron and Sigmoid neurons is the Sigmoid neurons don't just output 0 or 1. They can have as output any real number between 0 and 1, so value such as 0.173, 0.689, 0.04, 0.16.. are legitimate outputs. This can be useful, if we want to use the output value to represent the Intensity of the pixel in the Image to a neural network.

MULTI-LAYER PERCEPTRONS



1) Multiple Layer networks are called MULTI LAYER PERCEPTRONS
   They actually SIGMOID NEURONS

What is Feed Forward network ?

The output from one Layer is used as Input to the next Layer . Such networks are called "FEED FORWARD NETWORKS"

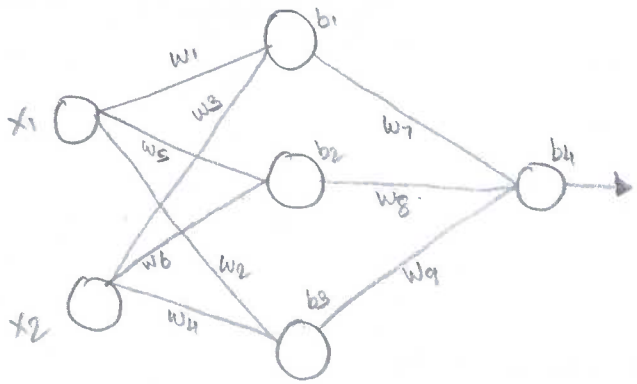
* this means there are no loops in the network
* Information is always Fed forward

NEURAL NETWORK CODE

The centerpiece is a NETWORK CLASS which we use to represent a neural network

```
Class Network (object):

    def __init__ (Self, Sizes):

        Self. num_layers = len (Sizes)

        Self. Sizes = Sizes

        Self. biases = [np. random. randn (y, 1) for y in Sizes[1:]]

        Self. weights = [np. random. randn (y, x)
                         for x, y in zip (Sizes[:-1], Sizes[1:])]
```

* In Python 3 Zip wraps two or more iterators with a Lazy generator.
* the Zip generator yields tuples containing the next value from each Iterator



net = Network([2, 3, 1])

- Input
- no of neurons in 1st Layer
- output

Sizes = [2, 3, 1]

GENERATE GAUSSIAN DISTRIBUTION WITH MEAN "0" and SD "1"

Self.bias = [np.random.randn(y, 1) for y in sizes[1:]]

sizes[1:] → [3, 1]

Because we don't need bias for the Input Layer

Self.bias = [np.array([[-0.8167], [-0.91592], [0.21430]]) (3,1),
np.array([[-0.51318]]) (1,1)]

Returns List of numpy array

Self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

sizes[:-1] → [2, 3]
sizes[1:] → [3, 1]

Returns List of numpy array

self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

[2, 3] → x_1 x_2 ; [3, 1] → y_1 y_2

↓

Returns List of Numpy array

self.weights = [np.array(([[-0.90317, -0.79685], [-1.0399, -1.13921], [0.7169, 0.3896]])), np.array(([[-0.54459, -1.49075, -1.05287]]))]

np.array(y_1, x_1) (3, 2) → weights connecting Input layer & the 1st Layer

np.array(x_2, y_2) (1, 3) → weights connecting 1st Layer & output Layer

SIGMOID FUNCTION CODE

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

* we then add a Feed Forward method to the Network class, which given an input "a" for the network, returns the corresponding output

```
def feedforward(self, a):

    for b, w in zip(self.biases, self.weights):

        a = Sigmoid(np.dot(w, a) + b)

    return a
```

Vector of activations of a layer

a is a (n, 1) numpy ndarray

where "n" is the number of

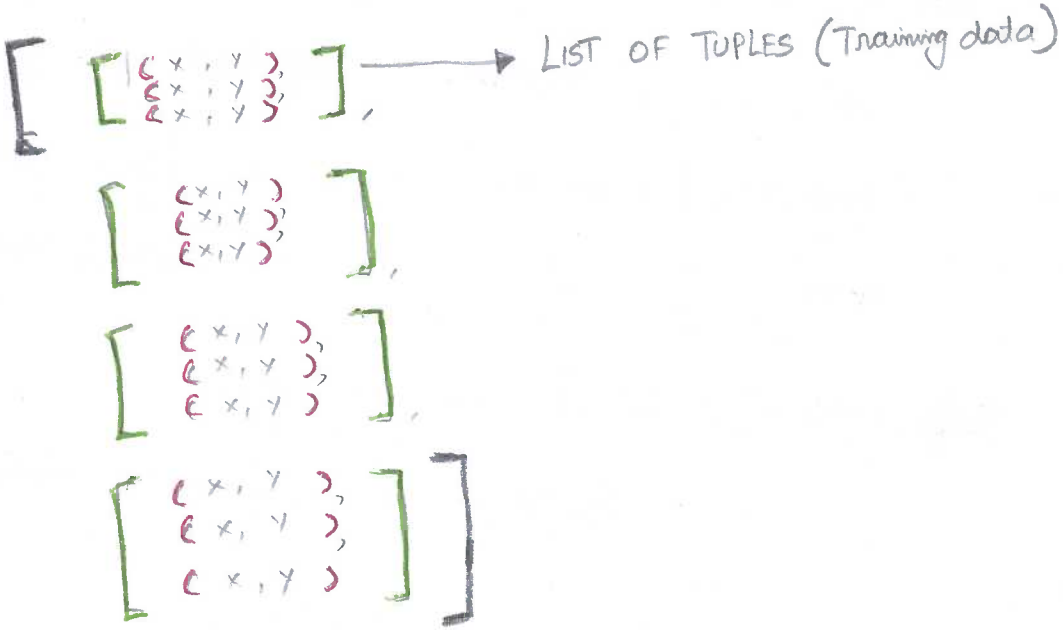
Input to the layer

* of course, the main thing we want our Network object to do is to learn. we will Implement SGD

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):

    if test_data: n_test = len(test_data)

    n = len(training_data)

    for J in range(0, epochs):

        random.shuffle(training_data)
```

```
mini_batches = [ training_data[k : k + mini_batch_size]
                 for k in range(0, n, mini_batch_size) ]
```

Returns List of List




```
for mini_batch in mini_batches:
    self.update_mini_batch(mini_batch, eta)
```

→ UPDATES NETWORK WEIGHTS AND BIASES

```
if test_data:
    Print "Epoch {0}: {1} / {2}".format(j, self.evaluate(test_data), n_test)

else:
    Print "Epoch {0} Complete".format(j)
```

```
def update_mini_batch (self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

(x)

* update_mini_batch works simply by computing the GRADIENTS for every training example in the mini_batch and the updating

self.weights

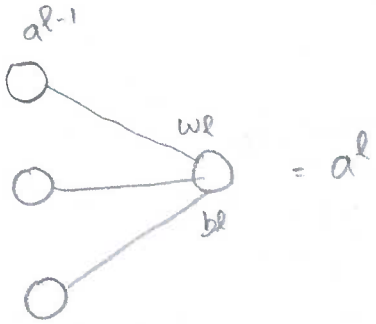
self.biases

appropriately

BACK PROPAGATION

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

* This expression gives us a much more global way of thinking about how the activation in one layer relate to activations in the previous layer.



* The GOAL of back propagation is to compute the partial derivatives of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to any weight "w" or bias "b" in the network.

* Back propagation is about understanding how changing the weights and bias in a network changes the cost function

* Chain rule is used to compute the gradient

Why is learning slow?

Our neurons learns by Changing the weight and the bias at a rate determined by the partial derivatives of the cost function $\partial C/\partial w$ and $\partial C/\partial b$. So Saying "Learning is slow" is really the same as saying that those partial derivatives are small

CROSS-ENTROPY COST FUNCTION :-

$$a = \sigma(z)$$

$$C = \frac{-1}{n} \sum_x \left[\underbrace{y}_{\text{GROUND}} \ln \underbrace{a}_{\text{PREDICT}} + \underbrace{(1-y)}_{\text{GROUND}} \ln \underbrace{(1-a)}_{\text{Predict}} \right] \quad \text{---} (1)$$

n = total number of items of Training data

Properties

* The cost function is NON-NEGATIVE ie $(C > 0)$
* Both Logarithm are of numbers in the range 0 to 1
* there is a minus sign out the front of the Sum

* If the neuron's actual output is close to the desired output for all training inputs "x", then the cross-entropy will be close to "zero"

Example:

Suppose

$y = 0$ and $a \approx 0$ for some Input "x"

→ This is the case when the neuron is doing a good Job on that input.

→ First term in equation (1) Vanishes

How cross-entropy differs from quadratic cost ?

Cross-entropy avoids the Problem of

LEARNING SLOWING DOWN

* the CROSS-ENTROPY is POSITIVE, and tends toward Zero as the neuron gets better at computing the desired output "y" for all training inputs "x"

Max-Pooling Intuition:

* We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the Image.

It then throws away the exact positional information, L2 Pooling is a way of CONDENSING information from the CONVOLUTIONAL layer.

HOW NETWORK PARAMETERS ARE UPDATED:

* The parameters of the network, filters in the Convolutional layer, weight-matrices in the fully-connected layer and biases are TRAINED by BACKPROPAGATING the DERIVATIVE of the Loss with respect to the parameters throughout the network and updating the parameters via STOCHASTIC GRADIENT DESCENT

* Understanding the operation of a CONVNET requires interpreting the feature activity in Intermediate layers.

* The CONVNET uses RELU non-Linearities, which rectify the feature maps, thus ensuring the feature maps are always POSITIVE

FEATURE VISUALIZATION :

* The visualizations reveal that a set of particular feature maps focuses on the "GRASS" in the background, not the foreground objects.

What causes high activations in a feature map ?

* Layer 2 responds to corners and edges
* Layer 3 has more complex invariances like Mesh patterns
* Layer 4 captures class-specific like dog's face, bird's leg
* Layer 5 shows entire object with significant pose variation

ARCHITECTURE SELECTION :

* While visualization of a trained model give insight into its operation, it can also assist with selecting good architectures in the first place.

* The first and 2nd layer of Krizhevsky architecture, the 1st Layer filters are a mix of extremely HIGH and LOW frequency information with the little coverage of MID frequencies

Does the Model learns the location of the object?

With "IMAGE CLASSIFICATION" approaches, a natural question is If the MODEL is truly identifying the LOCATION of the object in the IMAGE or Just using the Surrounding context.

This can be clarified by systematically OCCLUDING different portions of the Input Image with a GREY SQUARE and monitoring the output of the CLASSIFIER.

The example clearly Shows the MODEL is LOCALIZING the objects with the SCENE, as the probability of the correct class drops significantly when the object is OCCLUDED.

WHY CONVOLUTION?

The first thing to have in mind is that an Input Volume is normally Convolved with several different KERNELS

Why?

Because, we expect each different Kernel to extract different features from the Input Image.

CONVNETS are Inherently translation Invariant.

QuickNAT:

ABSTRACT

* Pre-train on auxiliary labels created from existing segmentation software
* Pre-trained model is fine TUNED on MANUAL LABELS to rectify errors in auxiliary labels.

INTRODUCTION

* In order to access measurements like VOLUME, THICKNESS or SHAPE OF A STRUCTURE, the neuroanatomy needs to be SEGMENTED
* Existing ATLAS-BASED methods require hours of processing time for each Scan and may result in SUB-OPTIMAL Solutions.
* the Limited availability of training data with Manual annotations present the main challenge in extending F-CNN models to brain Segmentation

1) AUXILIARY LABELS – Freesurfer generated Segmentation
2) MANUAL LABELS – Expert clinician Segmentation

* Pre-training provides a good prior initialization of the network

METHODS

1) Given an Input MRI brain Scan I, we want to Infer its segmentation map S, which indicates 27 CORTICAL and SUB-CORTICAL Structures.

ARCHITECTURE:

1) QuickNAT has an ENCODER/DECODER like 2D F-CNN architecture with 4 encoders and 4 decoders seperated by a BOTTLE NECK Layer.

2) Final Layer is a classifier block with "SOFTMAX"

3) The architecture includes SKIP CONNECTIONS between all encoder and decoder blocks of the same spatial resolution

4) These skip connections not only provide ENCODER FEATURE INFORMATION to the DECODER, but also provide a path of gradient flow from the shallow layers to deep layer, improving training

WHAT ARE SKIP-CONNECTIONS ?

SKIP CONNECTIONS add encoder features to the decoders for aiding segmentation and thus allow gradients to flow from deeper to shallower regions of the network.

CLASSIFIER BLOCK ?

The output feature map from the last decoder block is passed to the classifier block, which is basically a Convolutional layer with 1×1 Kernel Size that maps the input to a N channel feature map, where "N" is the number of classes (28 in our case). This is followed by a SOFTMAX LAYER to map the activations to PROBABILITIES

* In Probability theory, the output of the SOFTMAX FUNCTION can be used to represent a categorical distribution - that is, a probability distribution over "K" different possible outcomes.

NON-LINEAR FUNCTION

* The main terminologies needed to understand for non-linear functions are

Derivative/ Differential } Change in y-axis w.r.t Change in x-axis. It is also known as slope

Monotonic function } A function which is either non-increasing or non-decreasing

SIGMOID or LOGISTIC ACTIVATION FUNCTION

1) The Sigmoid function curve looks like an S-shape

2) The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to "predict the probability" as an output.

3) The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points

4) The function is **monotonic** but function's derivative is not.

BOTTLE NECK LAYER:

1) This Layer forces the ENCODER to Compress the Input Image and create some more high level representation

2) The DECODER will also have to learn how to decompress the high level representation into something that looks like an Input Image.

3) We have to Just Switch what "output target is" from trying to recreate the Input Image into trying to create the SEGMENTATION maps.

FULLY CONVOLUTIONAL NETWORK - (U-Net)

1) Domain specific Image FEATURES are learnt using SGD

2) Learnt kernels are shared across all pixels

3) Image CONVOLUTION operation exploit the structural Information in medical Images

4) Coarse feature-maps capture CONTEXTUAL Information and highlight the category and location of foreground objects

5) Feature - maps extracted at multiple scales are later MERGED Through SKIP CONNECTIONS TO COMBINE Coarse and fine - level dense predictions.

6) Convolutional layer learns filters Capturing local SPATIAL pattern along all the Input channels and generates feature maps Jointly encoding the SPATIAL and CHANNEL information.

SPATIAL → Means anything related to SPACE

CHANNEL → 3 for RGB Images

1 for GRAY Scale Images

SPATIAL INVARIANCE → Most of us can recognize specific objects under Variety of conditions because we learn abstractions. These abstractions are thus Invariant to

1) SIZE
2) CONTRAST
3) ROTATION
4) ORIENTATION
5) LOCATION

CONVNET

Successfully Captures the SPATIAL and TEMPORAL Information

SPATIAL	TEMPORAL
1) ~~When you have a series of Images~~ When you are analyzing Image. It Includes Co-ordinates, Intensity, gradient, resolution, Size, Contrast. Rotation, orientation & Location	1) When you have a series of Images taken at different time. Correlations between the Images are often used to monitor the dynamic change of the object

What is SE all about ?

Attention Gates Network (SE Blocks) explores the "SPATIAL" and "CHANNEL-WISE" patterns independently. Modelling the Interdependencies between the Channels of feature maps.

1) Spatial dependency is a process in which the Variables like gradient, Intensity, Size, Contrast, orientation + location may be dependent