Deep Learning ND

Notes by Pranjal Chaubey

https://www.linkedin.com/in/pranjall/

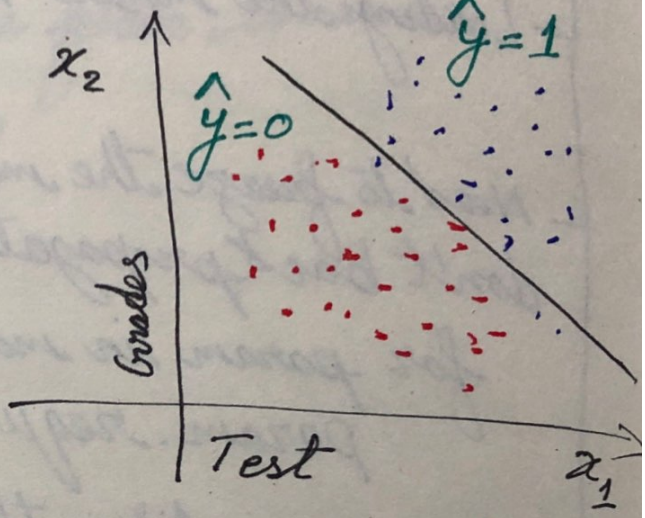
https://github.com/pranjalchaubey

UDACITY

19th June 2019

Introduction To Neural Networks

Trivially speaking, Neural Networks (NN) simply draw lines (or HYPERPLANES) to divide data into distinct classes.



Boundary Line:

$2x_1 + x_2 - 18 = 0$

↓ Score

$2 * Test + Grades - 18$

Prediction:

Score > 0 → Accept

Score < 0 → Reject

Generalized Form

$w_1 x_1 + w_2 x_2 + b = 0$

Vector Form

$Wx + b = 0$

Weights Inputs Biases

We are trying to predict the label y → $y = 0$ or 1

PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx+b \geq 0 \\ 0 \text{ if } Wx+b < 0 \end{cases}$$

If we get more data columns (Rank, Grades, Test), then we will simply be finding a DECISION BOUNDARY in an $n-1$ dimensional HYPERPLANE.

n-dimensional space:

$x_1, x_2, x_3 \ldots x_n$

Boundary:

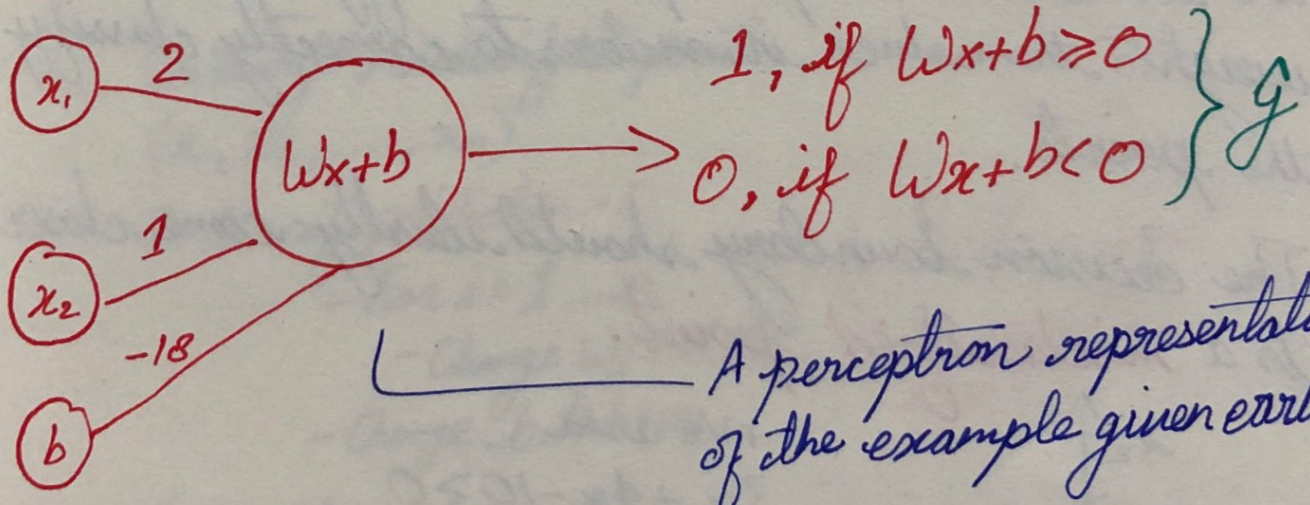
$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$

$Wx + b = 0$

Prediction

$$\hat{y} = \begin{cases} 1 \text{ if } Wx+b \geq 0 \\ 0 \text{ if } Wx+b < 0 \end{cases}$$

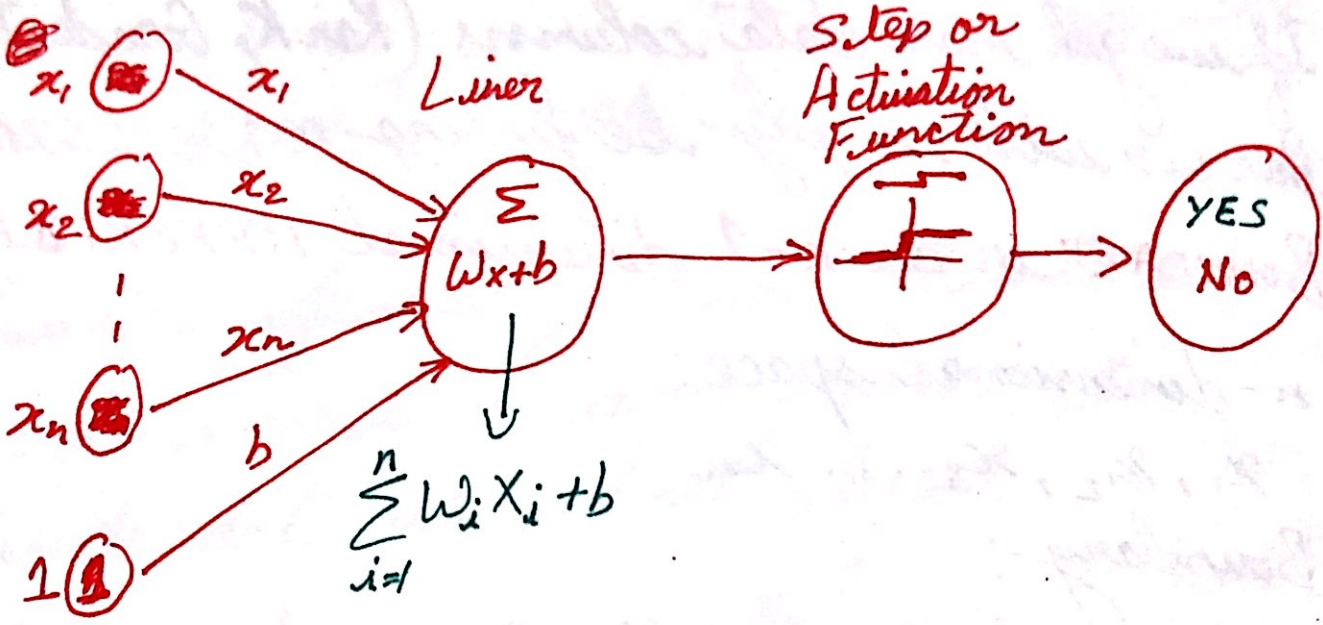
PERCEPTRON



A perceptron representation of the example given earlier.

A Perceptron is the basic building block of a NN.

- Takes inputs and bias, multiplies them, sums them up
- outputs the prediction $\hat{y}$ based on $Wx+b$

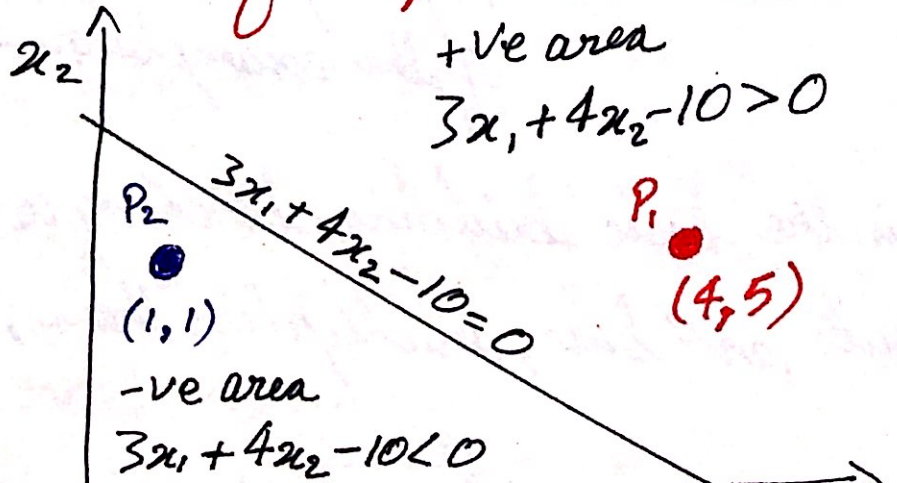


Perceptron can be made to behave like a logical operator.

- AND
- OR
- NOT
- XOR

PERCEPTRON TRICK

- We would like the perceptrons to figure out the weights themselves in order to correctly classify the points.
- The decision boundary should ideally come closer to a misclassified point.



To make the line come closer to the misclassified point P1

- Substract the coordinates from the weights and bias of the decision line

Use learning rate to slow down, else the line might move too fast and misclassify other points

To make the line come closer to the misclassified point P2

- Add the coordinates ~~from~~ to the weights and bias of the decision line

Multiply by the learning rate to slow things down

PERCEPTRON ALGORITHM

1. Start with Random Weights
   $w_1, w_2, \ldots w_n, b$
2. For every misclassified point
   $(x_1, x_2, \ldots, x_n)$
   2.1. if prediction = 0:
   - For $i = 1 \ldots n$
     - Change $w_i + \alpha x_i$
   - Change b to $b + \alpha$

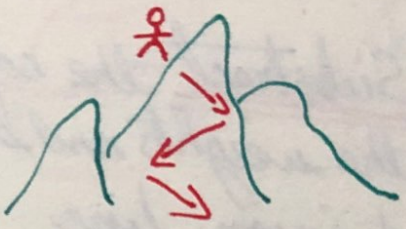
   Positive point in negative area

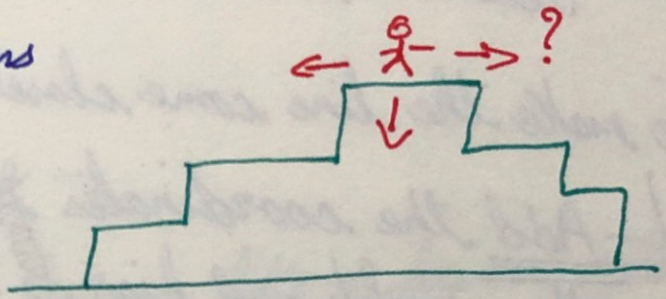
   2.2. if prediction = 1:
   - For $i = 1 \ldots n$
     - Change $w_i - \alpha x_i$
   - Change b to $b - \alpha$

   Negative point in positive area
3. Either iterate for a certain no. of times, or stop when desired accuracy is achieved.

Log-Loss Error Function

Gradient Descent
- GD is like coming down from a mountain.
- We take steps in directions where we go down the fastest.





But if we are standing on a flat aztec ~~mountain~~, no matter where we go, we will still be at the same level.
- Our error functions need to be CONTINUOUS as well as DIFFERENTIABLE, else GD won't work.

- Error = No. of points incorrectly classified
  - We update decision boundary's weights in such a way, such that the loss decreases.

Log-Loss Error = Penalize all the points
- Large penalty for a misclassified point
- We try and reduce the overall error magnitude to the minimum

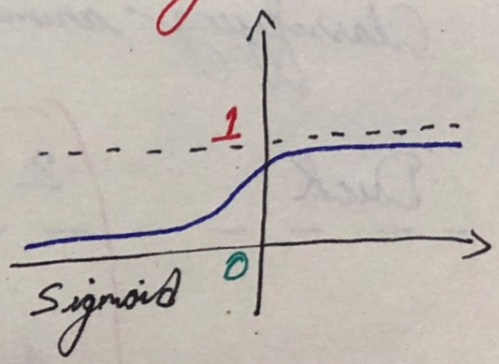
Discrete vs. Continuous Predictions

The Sigmoid Function

Discrete Algorithm — Yes No

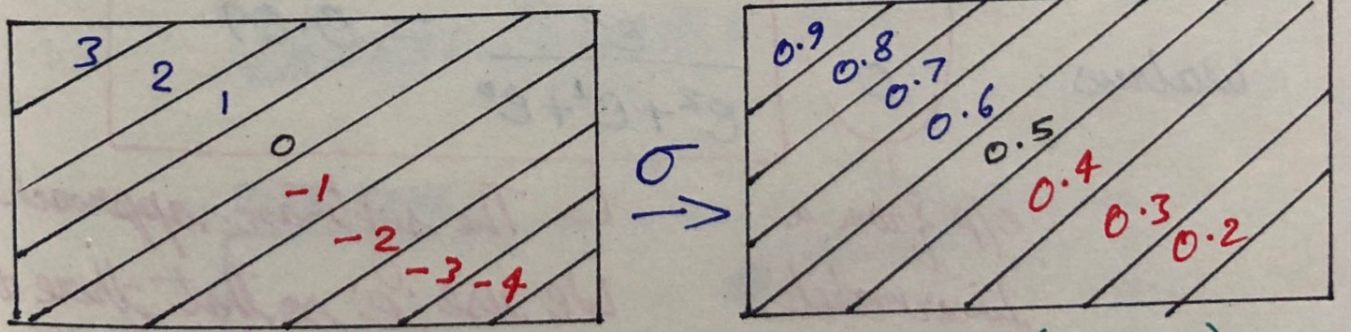
Continuous Predictions — 80% 70% 60% ... 30% 20% 10%

To move from discrete to continuous predictions, change the activation function from Step to Sigmoid.

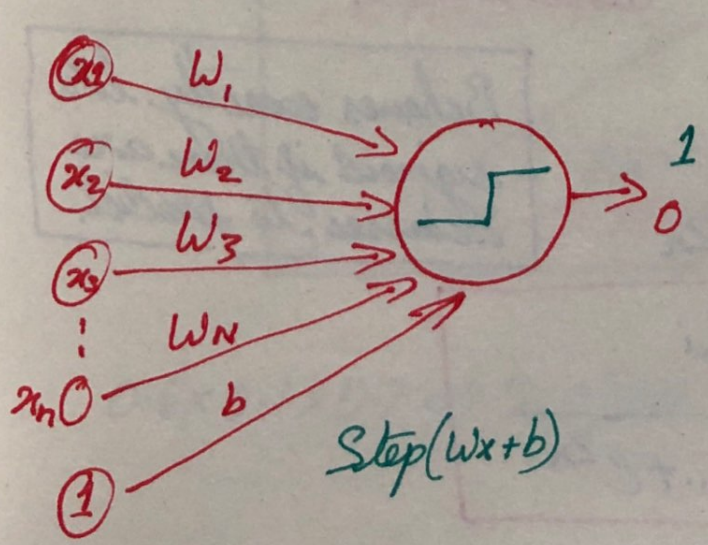


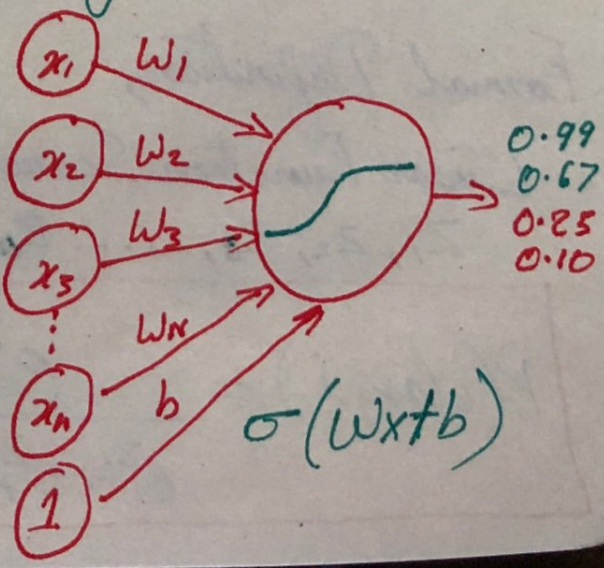
We require continuous predictions for our GD algorithm to work.

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



$Wx+b$ $\quad \hat{y} = \sigma(Wx+b)$





SOFTMAX

So far, our algorithm is able to tell a Yes or No answer. We can tell if the students got admitted or not in terms of a probability number.

└ Softmax is used when we have more than two categories to classify → Dogs Cats Rabbits

Classifying animals,

	o/p from a linear model	
Duck	2	$\frac{e^2}{e^2+e^1+e^0} = 0.67$
Beaver	1	$\frac{e^1}{e^2+e^1+e^0} = 0.24$
Walrus	0	$\frac{e^0}{e^2+e^1+e^0} = 0.09$

└ The softmax approach. We use 'e' so that there are no negative numbers in the denominator.

Behaves exactly like sigmoid if there are 2 classes to predict.

Formal Definition,

Linear Function Scores

$z_1, z_2, z_3, \ldots z_n$

$$P(\text{class } i) = \frac{e^{z_i}}{e^{z_1}+e^{z_2}+\ldots+e^{z_n}}$$

MAXIMUM LIKELIHOOD

We calculate the probabilities of all points being blue.

$$P(\text{blue}) = \sigma(wx + b)$$

Then we calculate the probabilites of all points being red

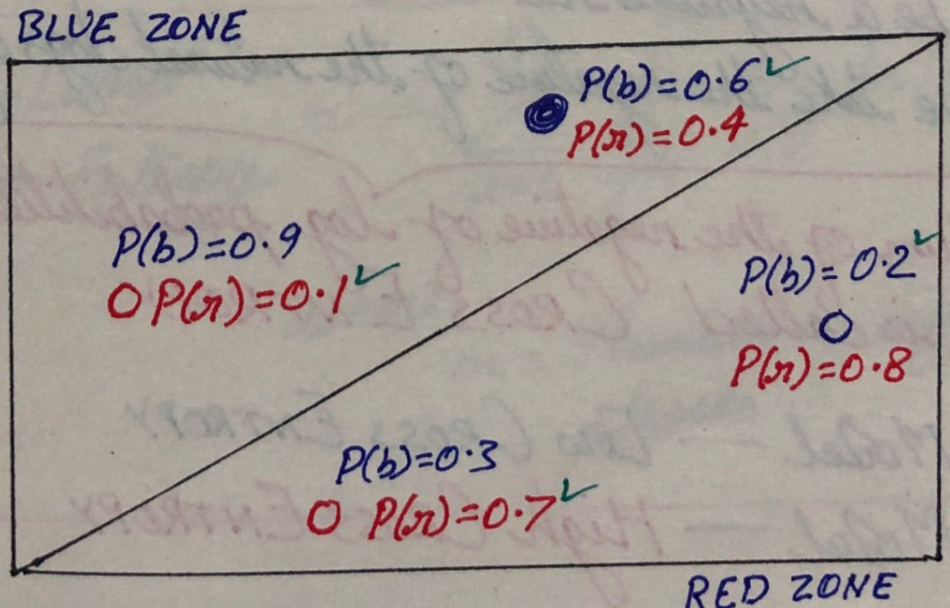
$$P(\text{red}) = 1 - P(\text{blue})$$

Use the ACTUAL PROBABILITIES of the points and MUTIPLY them.

Our job is to MAXIMIZE this product

⇓

Maximizing this probability product would result in a network with LOWEST POSSIBLE ERROR.



$0.6 \times 0.1 \times 0.7 \times 0.2 = P(\text{all})\ 0.0084$

Extremely small

MAXIMIZE this product

Maximizing Probabilities

On a real computer, we would like to stay away from 'products'.

- A product of thousands of data points will be something like '0.0000...'
  - This will underflow!
- We use SUMS instead.
  - LOGARITHMS! $\log_e(a*b) = \log_e(a) + \log_e(b)$
- Would maximizing the log sum would also result in decreasing the error function?
  - Cross-Entropy?

Cross-Entropy

Natural logarithms of probabilities (numbers less than 1) will be a negative number

- We take the negative of the natural logs of probabilities

Sum of the negative of log probabilities is called CROSS ENTROPY

Good Model — Low Cross Entropy

Bad Model — High Cross Entropy

✓

GOAL: MINIMIZE CROSS ENTROPY

Error can now be defined as the negative of the log probability at each point

- for correctly classified points, the error will be small
- for misclassified points, the error will be large
- Hence, minimize cross-entropy!

Events → (How likely the events happened based on those probabilities?) → Probabilities

- HIGHLY LIKELY → Low Cross Entropy

The concept of Cross Entropy given a certain set of events and their probabilities

$$\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i \ln(p_i) + (1-y_i)\ln(1-p_i)$$

$$\text{Multi-Class Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\ln(p_{ij})$$

where, m = Number of different classes

LOGISTIC REGRESSION

20th June 2019

Deriving the cross entropy (error) formula:

if the point is blue,

if $y = 1$

$P(\text{blue}) = \hat{y}$

$\text{Error} = -\ln(\hat{y})$

if the point is red,

if $y = 0$

$P(\text{red}) = 1 - P(\text{blue}) = 1 - \hat{y}$

$\text{Error} = -\ln(1-\hat{y})$

$\therefore \text{Error} = -y(\ln\hat{y}) - (1-y)(\ln(1-\hat{y}))$

for multi-class classification,

$$E = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij}\ln(\hat{y}_{ij})$$

$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\hat{y}_i) + (1-y_i)(\ln(1-\hat{y}_i))$$

Minimize this!

$$= -\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\sigma(Wx^{(i)}+b)) + (1-y_i)(\ln(1-\sigma(Wx^{(i)}+b)))$$

As a convention, the error is averaged out and divided by 'm'.

Gradient Descent, again,

We take the partial derivatives of the Error with respect to weights.

└ The vector sum of partial derivatives will point in the direction where the error increases the most

└ Therefore, we take the negative of the vector sum.

$\hat{y} = \sigma(Wx+b)$ — Bad $\hat{y}$

$= \sigma(W_1x_1 + W_2x_2 + \ldots + W_nx_n + b)$

$\nabla E = \left(\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial W_2}, \ldots, \frac{\partial E}{\partial W_n}, \frac{\partial E}{\partial b}\right)$

$\alpha = 0.1$ (Learning Rate)

update the weights and biases as following,

$W_i' \leftarrow W_i - \alpha \frac{\partial E}{\partial W_i}$

$b' \leftarrow b - \alpha \frac{\partial E}{\partial b}$

$\hat{y} = \sigma(W'x+b')$ — Better $\hat{y}$

The gradient ∇E actually comes out to be

$\nabla E = -(y-\hat{y})(x_1, x_2, \ldots x_n, 1)$

$\hookrightarrow W_i' \leftarrow W_i + \frac{1}{m} \cdot \alpha (y-\hat{y}) x_i$

$b' \leftarrow b + \frac{1}{m} \cdot \alpha (y-\hat{y})$

Gradient Descent Algorithm

1. Start with random weights:
   $w_1, \ldots, w_n, b$
2. For every point $(x_1, \ldots x_n)$:
   2.1. For $i = 1 \ldots n$
      2.1.1. Update $w_i' \leftarrow w_i - \alpha(\hat{y} - y)x_i$
      2.1.2. Update $b' \leftarrow b' - \alpha(\hat{y} - y)$
3. Repeat until the error is small.

Dangerously similar to the Perceptron Learning Algorithm

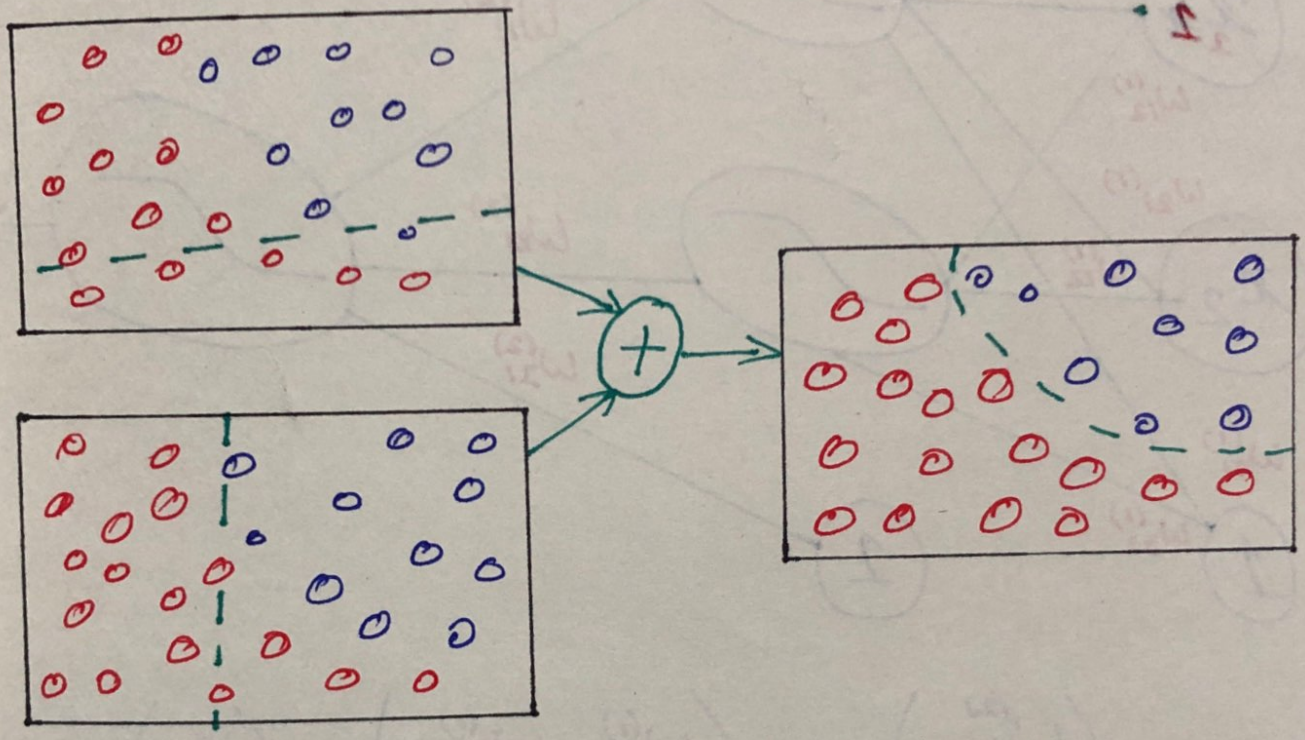
Logistic Regression is a classification technique. The optimal parameters for Logistic Regression are found by minimizing the loss function. Gradient Descent is the most common algorithm to reduce a loss function.

In perceptron learning algorithm, we change weights only if a point is misclassified, while in GD algorithm we are always modifying the weights till we get the desired accuracy.

Neural Network Architecture

We can combine two or more perceptrons to result in non-linearly separated boundaries.

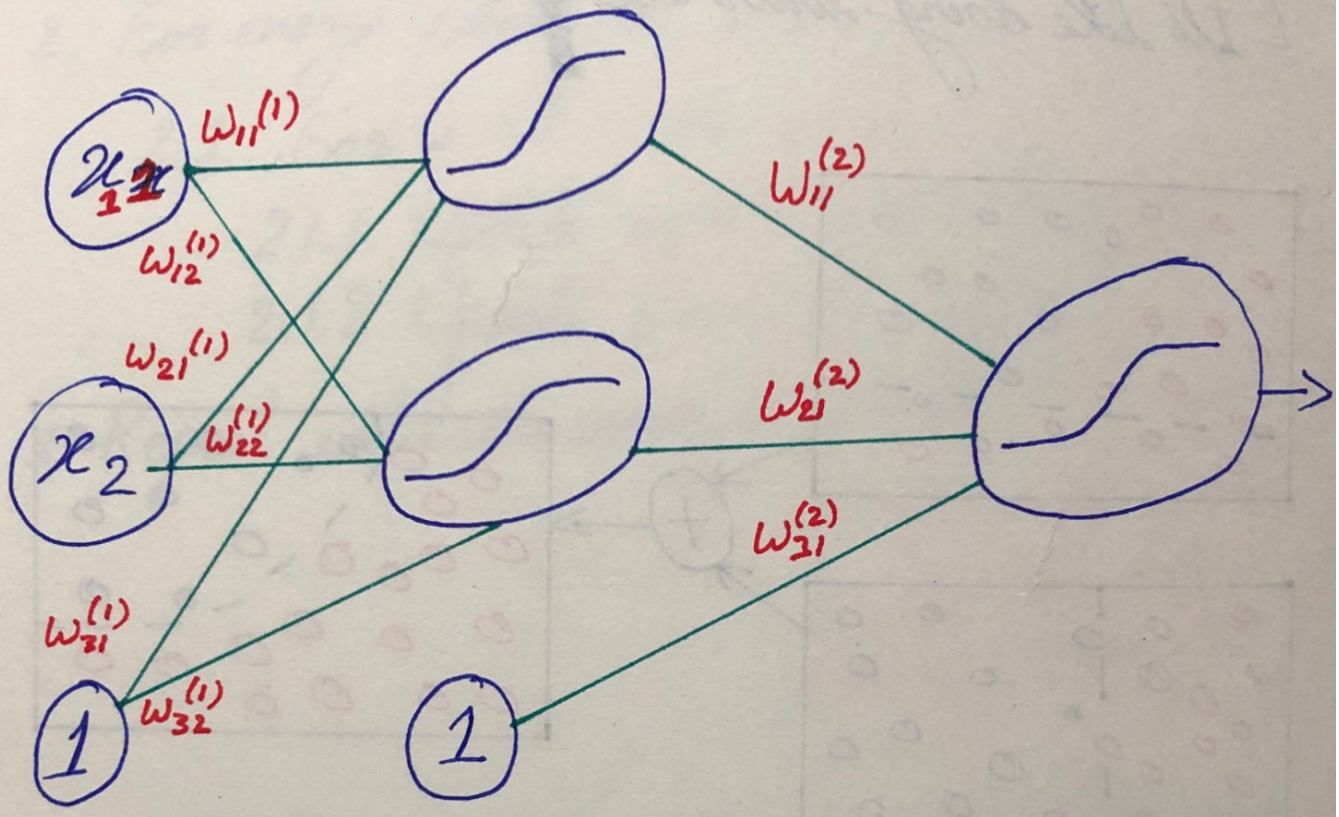
↳ Its like doing arithmetic on linear models.



Multiple layers of perceptrons can be combined into a deep learning network for even more non-linear output. And even multiple outputs in case of multi-class classification.

FEEDFORWARD

Feedforward is the process through which NNs make the prediction, $\hat{y}$.



$$\hat{y} = \sigma \begin{pmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \\ w_{31}^{(2)} \end{pmatrix} \sigma \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}$$

$$\hat{y} = \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)}(x)$$

BACKPRORAGATION

1. Do a feedforward operation
2. Compare $\hat{y}$ with y
3. Calculate error
4. Feedforward backwards and SPREAD the error to each of the weights (backpropagation)
5. Update the weights to get a better model.
6. Loop till the desired accuracy is achieves

Training a Neural Network using Backpropagation

Through backpropagation we try to find how much each of the weights are contributing to the error (partial derivative gives the rate of change of error with respect to a particular weight), then we try to adjust the weights in such a fashion so that the error gets minimized.

3rd October 2019

Implementing Gradient Descent

Gradient Descent with Squared Errors

- Make predictions as close to the Real Values.
  - Metric for measurement of incorrectness — ERROR

Error — Sum of Squared Errors (SSE)

$$E = \frac{1}{2} \sum_{\mu} \sum_{j} \left[y_j^{\mu} - \hat{y}_j^{\mu} \right]^2$$

y - True Value
$\hat{y}$ - Predicted Value
j - Output units of the network
μ - Data Point

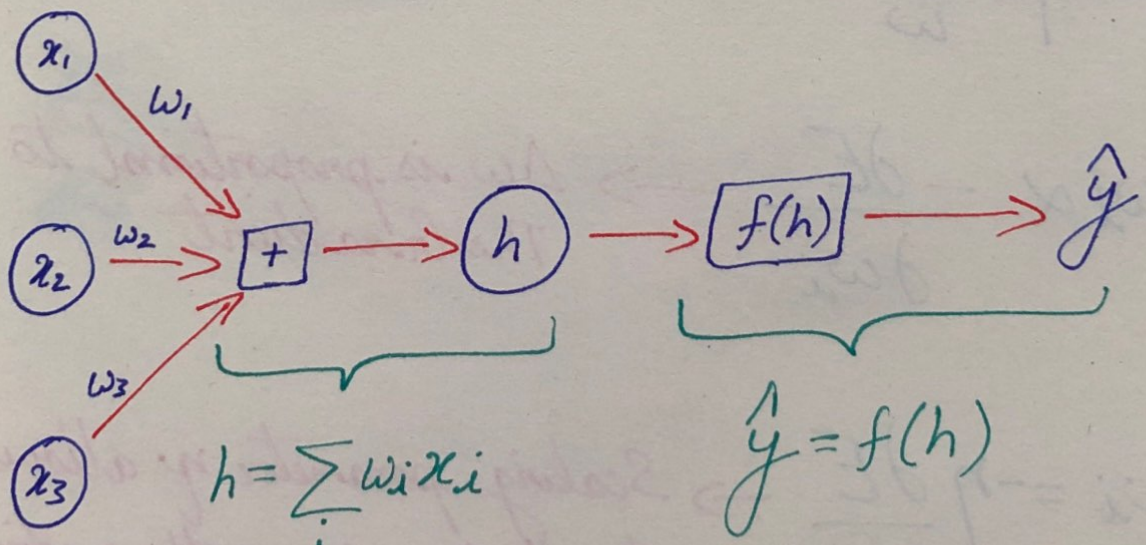
- Square of errors is always positive
- Larger errors are penalized more

Neural Network output depends on weights

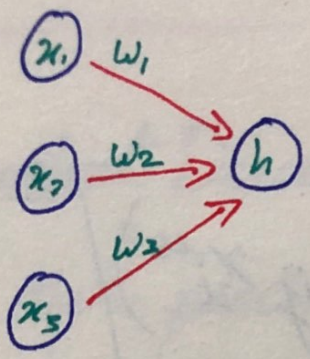
$$\therefore \hat{y}_j^\mu = f\left(\sum_i w_{ij} x_i^\mu\right)$$

$$E = \frac{1}{2} \sum_\mu \sum_j \left[y_j^\mu - f\left(\sum_i w_{ij} x_i^\mu\right) \right]^2$$

Adjust the weights
- Reduce the error
- Improve accuracy
- 'LEARN'



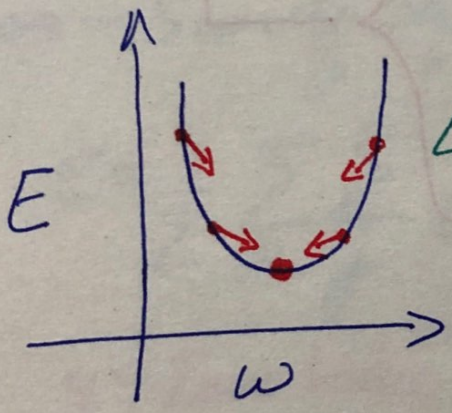
Gradient Descent : The Math



$$E = \frac{1}{2}(y - \hat{y})^2$$

$$= \frac{1}{2}\left(y - f\left(\sum_i w_i x_i\right)\right)^2$$

Convex Optimization Problem { Error is a function of the weights



$\Delta w = -\text{gradient}$

$w_i = w_i + \Delta w_i$

$$\Delta w_i \propto \frac{-dE}{dw_i}$$ → Δw is proportional to The Gradient

$$\Delta w_i = -\eta \frac{dE}{dw_i}$$ → Scaling parameter η allows to set the size of the gradient descent step

LEARNING RATE

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} (y - \hat{y})^2$$

$$= \frac{\partial}{\partial w_i} \frac{1}{2} (y - \hat{y}(w_i))^2$$

$\hat{y}$ is a function of w_i

CHAIN RULE } — We will use chain rule to calculate the derivative

$$\frac{\partial}{\partial z} p(q(z)) = \frac{\partial p}{\partial q} \frac{\partial q}{\partial z}$$

$q = (y - \hat{y}(w_i))$ = Error

$p = \frac{1}{2} q(w_i)^2$ = Square of the error

~~$\frac{\partial E}{\partial w_i}$~~ $= \frac{\partial p}{\partial q} = \frac{1}{2} q(w_i)^2$

$= 2 \times \frac{1}{2} \times q(w_i)$

$= q(w_i)$

$$\therefore \frac{\partial E}{\partial w_i} = (y - \hat{y}) \frac{\partial}{w_i} (y - \hat{y})$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2}(y-\hat{y})^2$$

$$= \frac{\partial}{\partial w_i} \frac{1}{2}(y-\hat{y})^2$$

$$= (y-\hat{y}) \frac{\partial}{\partial w_i}(y-\hat{y})$$

$$= -(y-\hat{y}) \frac{\partial \hat{y}}{\partial w_i}$$

$\hat{y} = f(h)$ — Activation Function

$\quad \llcorner\ h = \sum_i w_i x_i$

$$\frac{\partial E}{\partial w_i} = -(y-\hat{y}) \frac{\partial \hat{y}}{\partial w_i}$$

$$= -(y-\hat{y}) f'(h) \underbrace{\frac{\partial}{\partial w_i} \sum w_i x_i}$$

$$= \frac{\partial}{\partial w_1} [w_1 x_1 + w_2 x_2 + \dots + w_n x_n]$$

$$= x_1 + 0 + 0 \dots \dots$$

$$\therefore \frac{\partial}{\partial w_i} \sum w_i x_i = x_i$$

$$\frac{\partial E}{\partial w_i} = -(y-\hat{y})\, f'(h)\, x_i$$

Error — Derivative of the Activation Function — Input Value

$$\Delta w_i = \eta\,(y-\hat{y})\, f'(h)\, x_i$$

$$\text{Error Term} = \delta = (y-\hat{y})\, f'(h)$$

$$w_i = w_i + \eta\,\delta\, x_i \quad \text{— } \Delta w_i$$

From the MATHS above, we can see that it is not possible to ignore/substitute $f'(h)$ out of the equations. That is why it is MANDATORY for the activation functions to be DIFFERENTIABLE!

Mean Square Error (MSE)

- $\sum_i w_i$ with lots of data can lead to large updates
  - Gradient Descent might Diverge

$$E = \frac{1}{2m} \sum_{\mu} (y^{\mu} - \hat{y}^{\mu})^2$$

Algorithm:

- $\Delta w_i = 0$
- For each record in training data: ↑ h
  - make a forward pass, $\hat{y} = f(\sum_i w_i x_i)$
  - calculate error, $\delta = (y - \hat{y}) \times f'(\sum_i w_i x_i)$
  - $\Delta w_i = \Delta w_i + \delta x_i$
- Update the weights
  $\Delta w_i = \Delta w_i + \delta x_i$
  $\hookrightarrow w_i = w_i + \eta \frac{\Delta w_i}{m}$
  - η: Learning Rate
  - m: no. of records since we are averaging
- Repeat for e epochs

4th October 2019

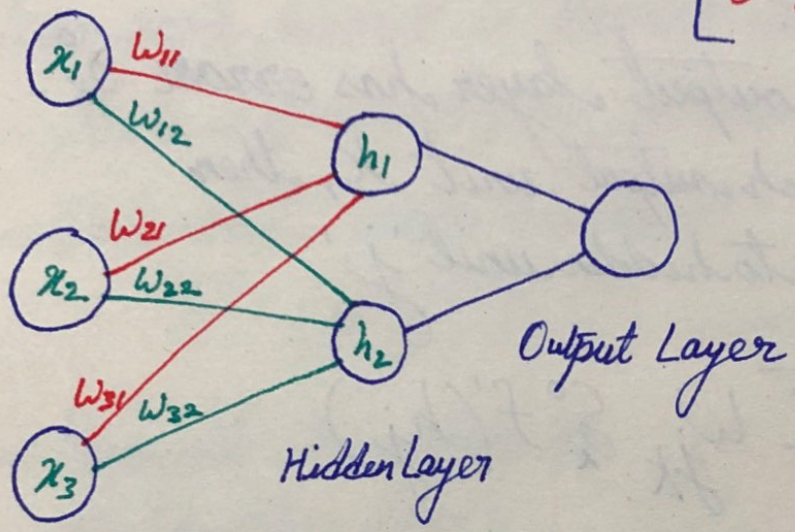
Multilayer Perceptrons

w_{ij} — i: Input Units, j: Hidden Units, k: Output Units

Hidden Units →

Input Units ↓

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$



$$h_j = \sum_i w_{ij} x_i$$

example, for 'h_1'

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

$$h_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} \Rightarrow \text{Dot Product}$$

Inputs are mostly in the form of row vectors.

BACKPROPAGATION

- extension of Gradient Descent
- uses chain rule to find the error with respect to the weights
  - Nodes with bigger weights contribute more towards the error and vice-versa.

for example, if output layer has errors δ_k^o attributed to each output unit k, then error attributes to hidden unit 'j'

$$\delta_j^h = \sum w_{jk} \delta_k^o f'(h_j)$$

Gradient Descent Step,

$$\Delta w_{ij} = \eta \delta_j^h x_i$$

Implementing BACKPROPAGATION

$$\delta_k = (y_k - \hat{y}_k) f'(a_k)$$ } Output Layer ~~for the~~ Error Term

$$\delta_j = \sum [w_{jk} \delta_k] f'(h_j)$$ } Error Term for the hidden layer

Algorithm for Backpropagation

- Set the ~~weights~~ steps for each layer to zero:
  - Input to hidden weights ~~$\Delta w_{ij} = 0$~~ $\Delta w_{ij} = 0$
  - Hidden to output weights $\Delta W_j = 0$
- For each record in training data
  - Make a forward pass calculating $\hat{y}$
  - Calculate error gradient

$$\delta^o = (y - \hat{y}) f'(z) \longrightarrow z = \sum_j W_j a_j$$

Input to the output unit

  - Propagate the error to the hidden layer

$$\delta_j^h = \delta^o W_j f'(h_j)$$

  - Update the weight steps

$$\Delta W_j = \Delta W_j + \delta^o a_j$$

$$\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$$

- Update the weights

$$W_j = W_j + \eta \Delta W_j / m$$

$$w_{ij} = w_{ij} + \eta \Delta w_{ij} / m$$

m = number of records/rows

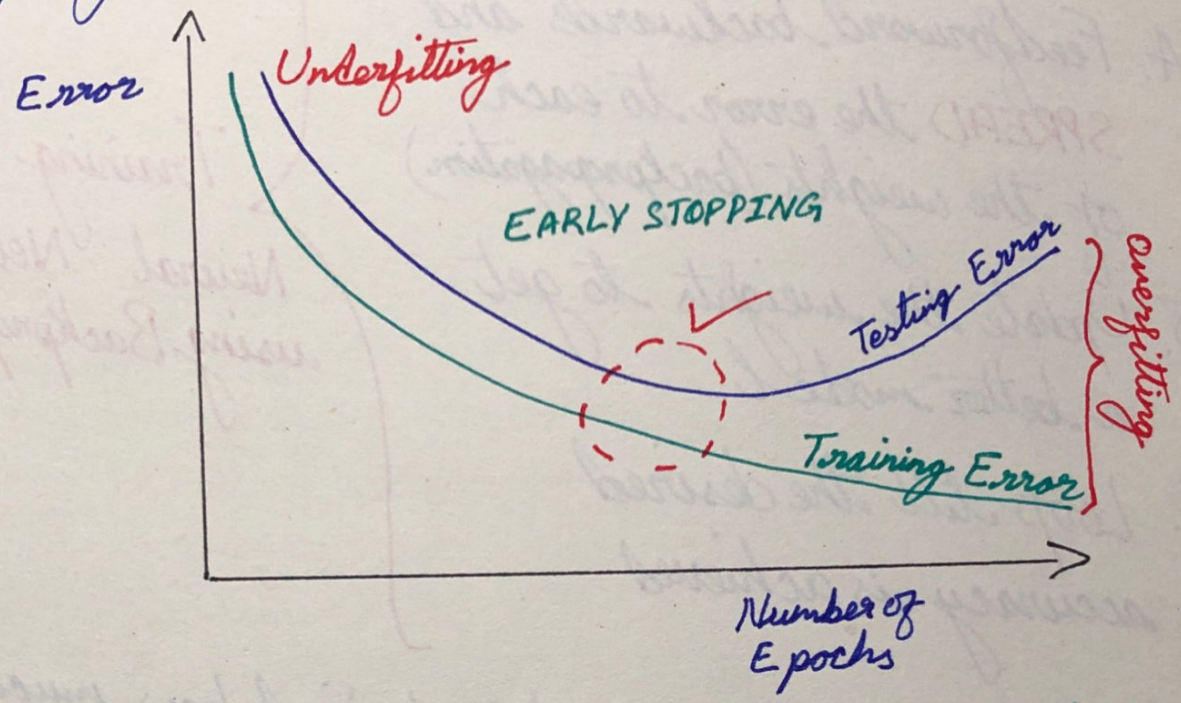
- Repeat for 'e' epochs

Training Neural Networks

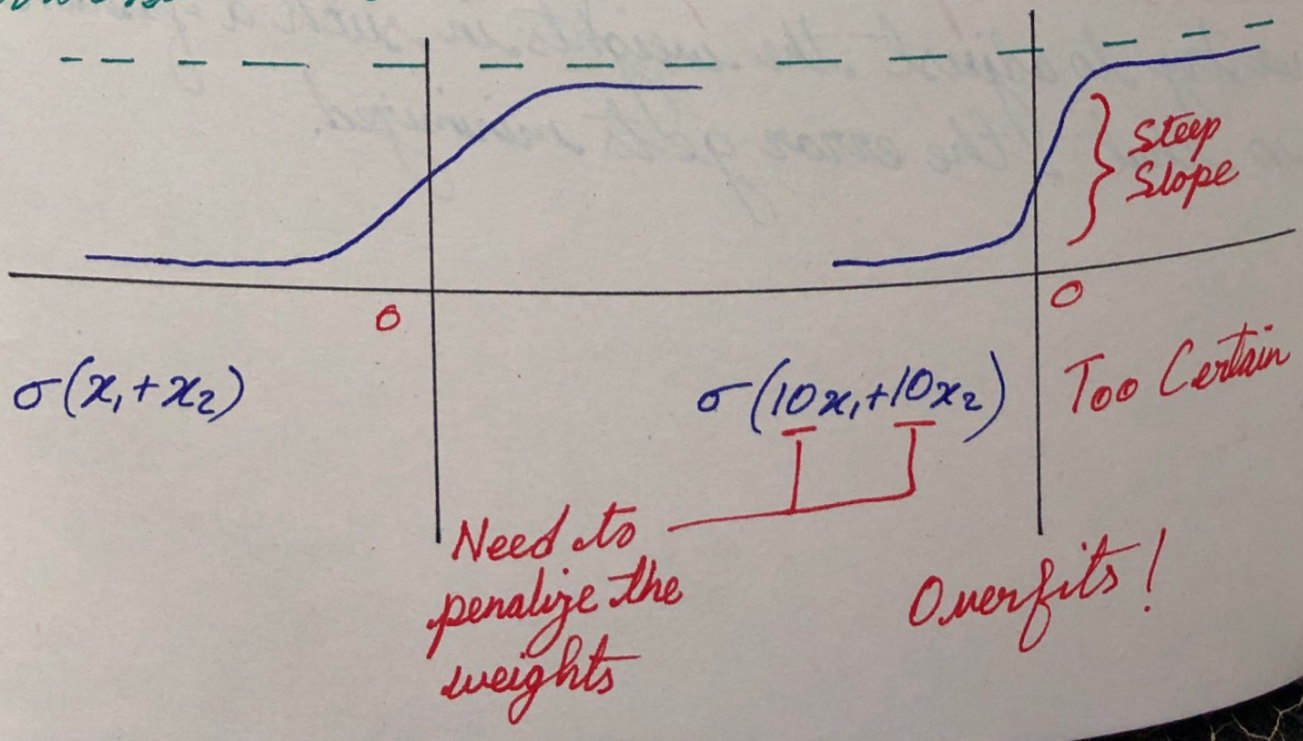
21st June 2019

Underfitting – Error due to bias

Overfitting – Error due to variance



Early Stopping – We keep running the training loop till the training and testing errors keep decreasing. As soon as testing error starts to increase we STOP.



REGULARIZATION

Big weights tend to overfit the model on the data.
↳ Penalize them!

L1 Regularization or LASSO Regression

LASSO - Least Absolute Shrinkage and Selection Operator

$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\hat{y}) + (1-y_i)\ln(1-\hat{y}) + \lambda(|w_1| + |w_2| + \ldots + |w_n|)$$

$$\hookrightarrow \lambda \sum_{j=1}^{n} |w_j|$$

Lasso shrinks the less important feature's coefficient to ZERO, removing them altogether. Useful in Feature Selection when there is a large number of features.

L2 Regularization or RIDGE Regression

Penalizes the squared magnitude of the coefficients
↳ Better in Training ~~Test~~ Models

$$\text{Error Function} = -\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\hat{y}) + (1-y_i)\ln(1-\hat{y}_i) + \lambda(w_1^2 + w_2^2 + \ldots + w_n^2)$$

$$\hookrightarrow \lambda \sum_{j=1}^{n} w_j^2$$

DROPOUT

We randomly drop some of the nodes during training and let the other nodes train
- Required since a few nodes start to dominate training sometimes
  - Rest of the nodes don't get to participate in training
    - Results in a weak network that generalizes poorly on the testing data.
- Results in a very robust network.

VANISHING GRADIENTS

The derivative of the sigmoid function on the far edges is a very small number.
- In backpropagation step, we multiply all the gradients all the way back to the input weights.
  - Resulting number will be very small
    - the problem of VANISHING GRADIENT
- Alternate Activation Functions

Hyperbolic Tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- $-1 < x < 1$

Rectified Linear Unit (ReLU)

$$\text{relu}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Deep Learning with PyTorch

5th June 2019

Best method for ~~matrix~~ tensor multiplication in PyTorch

`torch.mm(tensor1, tensor2)`

`torch.matmul()` — Broadcasts tensors, hence could give results even when the dimensions don't match. Not Recommended!

Methods to manipulate the shape of ~~tensor~~ tensor

`weights.reshape(a,b)` — 'Sometimes' copies the whole tensor to a new memory — NOT EFFICIENT!

`weights.resize_(a,b)` — If the new shape doesn't match, it will add/subtract new elements in the tensor without giving any error. — NOT RECOMMENDED!

`weights.view(a,b)` — Does what is asked, in the way it is expected. — HIGHLY RECOMMENDED

`.view(-1)` — Flattens a tensor

`images.view(images.shape[0], -1)` — Flattens the 'image' tensor in this case

`images.shape[0]` — Size of the Batch

Loading the training data through DataLoader

```
trainloader = torch.utils.data.DataLoader
              (trainset, batch_size=64, shuffle=True)
```

Training Dataset

When iterating through 'trainloader', each iteration will contain a batch of 64 images.

Randomly pick images in a batch for training

Converting 'trainloader' in an iterable,

```
dataiter = iter(trainloader)
images, labels = dataiter.next()
```

Sigmoid Activation Function

```
def activation(x):
    return 1/(1+torch.exp(-x))
```

Softmax Function

```
def softmax(x)
    return torch.exp(x)/torch.sum(torch.exp(x),
                                  dim=1).view(-1, 1)
```

$$Softmax = \sigma(x_i) = \frac{e^{x_i}}{\sum_k^K e^{x_k}}$$

Building Networks with PyTorch nn Module

```
from torch import nn

class Network(nn.Module):
    def __init__(self):
        super().__init__()

        # Inputs to hidden layer linear transformation
        self.hidden = nn.Linear(784, 256)
        # Output layer
        self.output = nn.Linear(256, 10)

        # Define sigmoid activation & softmax output
        self.sigmoid = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        # Pass the input tensor through each operation
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.softmax(x)

        return x
```

Registers the 'network' with nn module

This section can be removed using nn.functional

```
import torch.nn.functional as F
class Network(nn.module):
    def __init__(self):
        super().__init__()
        #Inputs to hidden layer linear transformations
        self.hidden = nn.Linear(784, 256)
        #Output Layer
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        #Hidden Layer with sigmoid activation
        x = F.sigmoid(self.hidden(x))
        #Output Layer with Softmax
        x = F.softmax(self.output(x), dim=1)
        return x
```

x = x.view(x.shape[0], -1)

Adding this will result in the tensor automatically getting flattened out by the network

A more succint version of the previous network

```
# Initialize the 'model' by creating the network object
model = Network()
```

Using nn.Sequential

```
# Hyperparameters for our network
input_size = 784
hidden_sizes = [128, 64]
output_size = 10

from collections import OrderedDict
model = nn.Sequential(OrderedDict([
                      ('fc1', nn.Linear(input_size, hidden_sizes[0])),
                      ('relu1', nn.ReLU()),
                      ('fc2', nn.Linear(hidden_sizes[0], hidden_sizes[1])),
                      ('relu2', nn.ReLU()),
                      ('output', nn.Linear(hidden_sizes[1], output_size)),
                      ('softmax', nn.Softmax(dim=1))]))
```

Layer names have to be UNIQUE

Ultra fast method to define a network

```
model.fc1.weight
model.fc1.bias
```
} Accessing various model layers

```
model.fc1.bias.data.fill_(0)
```
} Setting all biases to 0

```
model.fc1.weight.data.normal_(std=0.01)
```
↳ Setting weights to a sample from random normal distribution with a Standard Deviation of 0.01

Training a Neural Network

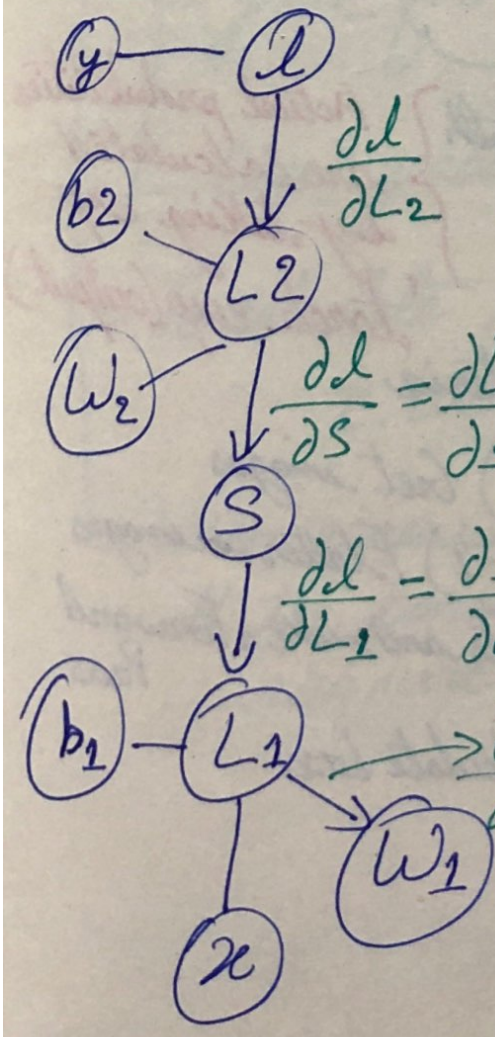
7th June 2019

We need a loss function to determine the errors our neural network is making, in case of ~~Regression~~ regression and binary ~~classification~~ classification problems, often **Mean Squared Error** is used.

$$l = \frac{1}{2n} \sum_{i}^{n} (y_i - \hat{y}_i)^2$$

- n: Number of examples
- y: True Labels
- $\hat{y}$: Predicted Labels

Backpropagation



$\frac{\partial l}{\partial L_2}$

We are effectively trying to figure out how 'l' is changing with respect to a change in 'W_1'. Once we know that, we can minimize 'l' by updating weights,

$$W_1' = W_1 - \alpha \frac{\partial l}{\partial W_1}$$

α: Learning Rate

$$\frac{\partial l}{\partial S} = \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

$$\frac{\partial l}{\partial L_1} = \frac{\partial S}{\partial L_1} \frac{\partial l}{\partial S} = \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

$$\frac{\partial l}{\partial W_1} = \frac{\partial L_1}{\partial W_1} \frac{\partial l}{\partial L_1}$$

$$= \frac{\partial L_1}{\partial W_1} \frac{\partial S}{\partial L_1} \frac{\partial L_2}{\partial S} \frac{\partial l}{\partial L_2}$$

Gradient

We typically use the nn.CrossEntropyLoss in PyTorch in classification problems.

Conventionally, the loss is assigned to a variable called criterion.

CrossEntropyLoss requires LOGITS as its inputs instead of the probabilities from the Softmax.

└ Logits are nothing but the INPUTS to the Softmax function.

CrossEntropyLoss is a combination of

nn.NLLLoss
|
Negative Log Likelihood Loss

nn.LogSoftmax
Used along with LogSoftmax

Actual probabilities are calculated by taking exp. torch.exp(output)

```
criterion = nn.NLLLoss()                    Loss criteria
images, labels = next(iter(trainloader))    Get images
images = images.view(images.shape[0], -1)   Flatten the images
logits = model(images)                      Get the logits and make a Forward Pass
loss = criterion(logits, labels)            Calculate Loss
print(loss)
```

AUTOGRAD

Autograd automatically calculates the gradients.

- Set requires_grad = True for a tensor
- Can be turned off for a block of code

```
x = torch.zeroes(1, requires_grad = True)
with torch.no_grad() — Turns off autograd
    y = x*2
```

- Gradients can be globally turned on and off using
  `torch.set_grad_enabled(True/False)`
- Keeps track of all the operations performed on a tensor
  - grad_fn can be used to check the function that generated this variable
  - Check gradients for a tensor `tensor.grad`
- To calculate the gradient of z w.r.t. x

```
z.backward()
```

- When a new network is created in PyTorch, all the parameters are initialized with requires_grad = True

8th June 2019

OPTIMIZER

Once we get the gradients from Autograd
- We need to update the weights
  - performed by an optimizer

Optimizers in PyTorch live inside the optim package
- from torch import optim

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

- SGD: Stochastic Gradient Descent
- model.parameters(): Model parameters to optimize
- lr: Learning Rate

General Steps to train a network in PyTorch
- Zero out the gradients since they keep getting accumulated with each batch
  `optimizer.zero_grad()`
- Make a Forward Pass through the network
- Use the network output to calculate the loss
- Perform backward pass using `loss.backward()` to calculate the gradients
- Take a STEP with the optimizer to update the weights
- One pass through the entire dataset is called an EPOCH

```
from torch import optim
optimizer = optim.SGD(model.parameters(), lr=0.01)

images, labels = next(iter(trainloader))
images.resize_(64, 784)
optimizer.zero_grad()
output = model.forward(images)
loss = criterion(output, labels)
loss.backward()
optimizer.step()
```

```
epochs = 5
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
        optimizer.zero_grad()
        output = model.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    else:
        print(f"Training loss {running_loss/
                               len(trainloader)}")
```

TRAINING LOOP

10th June 2019

VALIDATION

After training the model on 'training data', we need to 'validate' it on 'testing data' in order to check its accuracy.

Training data is loaded in trainloader.
Testing data is loaded in testloader.

Finding the model accuracy on a test batch is simple,

```
#Get the class probabilities
ps = torch.exp(model.forward(images))
#Use ps.topk(n) method that gives the highest
  'n' classes predicted for an input
# ps.topk(1) gives the highest class with the highest
  probability
top_p, top_class = ps.topk(1, dim = 1)
#Get the correctly predicted labels as 1
equals = top_class == labels.view(*top_class)
#Get the accuracy
accuracy = torch.mean(equals.type(torch.FloatTensor))
```

MARK

equals → ByteTensor

*top_class → To ensure correct dimensions of tensors

FloatTensor → Convert to float to calculate mean

```
for e in range(epochs):
    for images, labels in trainloader:
        .
        .
        .
        .
        .
    else:
        test_loss = 0
        accuracy = 0
        with torch.no_grad():
        for images, labels in testloader:
            log_ps = model.forward(images)
            test_loss += criterion(log_ps, labels)

            ps = torch.exp(log_ps)
            top_p, top_class = ps.topk(1, dim=1)
            equals = top_class == labels.view(*top_class)
            accuracy += torch.mean(equals.type(torch.
                                   FloatTensor))

        print(f'Test loss: {test_loss/len(testloader)}')
        print(f'Accuracy: {accuracy/len(testloader)}')
```

Training loop

Turn off gradients to speed things up

VALIDATION Loop

OVERFITTING

As the network tends to learn the training data better and better, it tends to OVERFIT.

- As a result, its accuracy on the VALIDATION/TESTING DATA goes DOWN!
- To avoid overfitting
  - Early Stopping — Save different models and choose the one with lowest validation loss.
  - Dropout — Randomly drop neural units to make the network more robust.

```
self.dropout = nn.Dropout(p=0.2)
                          Drop Probability

x = self.dropout(F.relu(self.fc1(x)))
```

- During testing/validation we would want to use the entire model without dropout.
  - model.eval()
    - Back to training mode
      - model.train()

Saving and Loading Models

The model parameters for PyTorch networks are stored in a model's state_dict.

- `print(model.state_dict().keys())`
- Saving the models is easy
  - `torch.save(model.state_dict(), 'filename.pth')`

Extension for PyTorch Models

- Loading the state_dict
  - `state_dict = torch.load('filename.pth')`
  - Attaching the state_dict to a new model
    - `model.load_state_dict(state_dict)`

The new model needs to have exactly the same number of layers and the number of neurons per layer as the original ~~net~~ model. Otherwise, PyTorch will throw an error.

Loading Image Data

Simplest way to load image data in PyTorch is through datasets.ImageFolder from torchvision

- `dataset = datasets.ImageFolder('path/to/data', transform=transform)`
- The system expects the folder to have a particular structure,

```
root/dog/d1.png
root/dog/d2.png
root/cat/c1.png
root/cat/c2.png
   ⋮
```

- Images are also usually transformed when they are loaded
  - `transform = transforms.Compose([transforms.Resize(255), transforms.CenterCrop(224), transforms.ToTensor()])`

Transforms Pipeline

- We generally introduce randomness into the data used for training the network to make it more robust
  - `.RandomRotation(30)`
  - `.RandomResizedCrop(224)`
  - `.RandomHorizontalFlip()`
  - `.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])`
    - Keeps the network weights near 0
    - Makes backpropagation more stable

TRANSFER LEARNING

We use pre-trained models to solve hard Computer Vision problems

- from torchvision import datasets, transforms, models
  - imports pre-trained models
- Loading the models `model = models.densenet121(pretrained=True)`
- Need to freeze the model parameters so that we don't backpropagate through them

```
for param in model.parameters():
    param.requires_grad = False
```

- We only modify the final classifier layer according to our applications

```
classifier = nn.Sequential(
    ⋮
)
```

New layers that we will attach as the final layers in the pretrained network

`model.classifier = classifier` — Attach the new layers to the pretrained model

- Moving tensors back and forth from GPU to CPU

```
model.cuda()      model.cpu()
images.cuda()     images.cpu()
print(torch.cuda.is_available())
model.to('cuda')
model.to('cpu')
```

Check if cuda is available