M20AIE323
VINEET VERMA

Problem 1: Perceptron

Following training samples are given -

X_1	X_2	class
1	1	+1
-1	-1	-1
0	0.5	-1
0.1	0.5	-1
0.2	0.2	+1
0.9	0.5	+1

weight vector of initial decision boundary

$W^T X = 0$ as $W = [1, 1]$

Ques 1.1

In how many steps perception learning algorithm will converge.

~~Answer~~ :

Ques 1.2:

What will be the final decision boundary ?
Show step-wise update of weight vector using computation as well as hand written plot.

Answer :

Assuming data is centered around the origin for simplicity bias is absorbed into weight vector.

We can represent decision boundary as $\boxed{W^T x = 0}$ where W includes bias

if we consider bias seperately. convergence steps will change.

Initial weight vector $W = [1, 1]$, α (learning rate) $= 1$

$$f(y_{predicted}) = \begin{cases} 1 & y_{pred} > 0 \\ -1 & y_{pred} \leq 0 \end{cases}$$

(1) Using initial weight $[1, 1]$ for first datapoint $d_1(1, 1)$

$$y_{predicted_1} = x_1 w_1 + x_2 w_2$$
$$= (1)1 + (1)1 = 2 \Rightarrow f(y_{pred_1}) = 1 \rightarrow y_{prd_1}$$

Predicted label is correct and matches true class

no weight updation.

(2) $W[1,1]$ for datapoint $(-1, -1)$ $d_2(-1, -1)$

$$y_{prd_2} = x_1 w_1 + x_2 w_2$$
$$= (-1)(1) + (-1)(1) = -2 \Rightarrow f(y_{pred_2}) = -1 \; y_{prd_2}$$

Predicted label is correct and matches the true class

No weight updation

(3) $W[1,1]$ for datapoint $(0, 0.5)$ $d_3(0, 0.5)$

$$y_{prd_3} = x_1 w_1 + x_2 w_2$$
$$= (1)(0) + (1)(0.5) = 2.5 \Rightarrow f(y_{pred_3}) = +1 \; y_{prd_3}$$

Predicted label is not correct and doesnot match the true class

Hence weight update is required

$W_{new_1} = W_{old} + (\alpha)(Y_3 - Y_{pred_3})(X_{31}) \Rightarrow 1 + (-1-(+1))0$ (2)
$\Rightarrow 1$

$W_{new_2} = W_{old} + (\alpha)(Y_3 - Y_{pred_1})(X_{32}) \Rightarrow 1 + (-1-(+1))0.5$
$= 1 + (-2)(0.5)$
$= 0$

(4)

$W = [1, 0]$

Use $W = [1,0]$ to predict datapoint $(0.1, 0.5) \leftarrow d_4$

$Y_{pred_4} = x_1 w_1 + x_2 w_2$
$= (1)(0.1) + (0)(0.5) = 0.1$

$f(Y_{pred_4}) = (1)$

Predicted label is incorrect, hence weight update is required

$W_{new_1} = W_{old} + \alpha (Y_4 - Y_{pred_4}) X_{41} \Rightarrow 1 + 1(-1-1)0.1 = 0.8$

$W_{new_2} = W_{old} + \alpha (Y_4 - Y_{pred_4}) X_{42} \Rightarrow 0 + 1(-1-1)0.5 = -1$

(5)

$W = [0.8, -1]$ ← weight vector

Use W $[0.8, -1]$ to predict datapoint $(0.2, 0.2) \leftarrow d_5$

$Y_{pred_5} = x_1 w_1 + x_2 w_2$
$= (0.8)(0.2) + (-1)(0.2)$
$= -0.04$

$f(Y_{pred_5}) = (-1)$

Predicted label is incorrect, hence weight update is required

$W_{new_1} = 0.8 + (1-(-1))0.2 = 1.2$

$W_{new_2} = -1 + (1-(-1))0.2 = -0.6$

$[1.2, -0.6]$

(6)

$W\ [1.2, -0.6]$ ← weight vector

Use $W\ [1.2, -0.6]$ to predict datapoint $(0.9, 0.5)$

$$Y_{pred_6} = x_1 w_1 + x_2 w_2$$

$$= (1.2)(0.9) + (0.5)(-0.6)$$

$$= 1.08 - 0.3$$

$$= 0.78$$

$$f(Y_{pred_6}) = \boxed{+1}$$

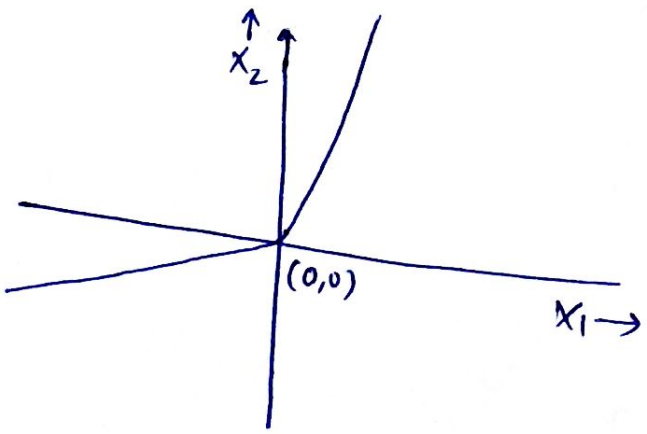
Predicted label is classified correctly and weight update is not required

1. Total Six → (6) steps are required to converge weight update happens (3) times in convergance

2. final decision boundary is

$$1.2\, x_1 - 0.6\, x_2 = 0$$

$$\boxed{x_1 = \tfrac{1}{2} x_2}$$



Problem 2:

Learning to implement Neural Network

- Simple Neural Network was created to perform Gurmukhi Handwritten Digit classification.

Three Approach were taken:

(1) Simple ANN was implemented without using any framework like keras, pytorch

Test accuracy achieved = 94.94%

(2) Simple ANN implemented using keras library

Test accuracy achieved = 93.82%

(3) simple ANN using keras and ImageDataGenerator lib

Test accuracy achieved = 83.70

Kindly refer the supporting attached python notebook (ipynb) file / Github link for implementation details.

Problem 03

Chart Image classification using CNN

Report

~~Task I~~ TASK - 01

Problem was solved in Jypter notebook on google colab using IIT I email id.

Dataset was downloaded and labels were read from .csv file.

Images were split into 80:20 ratio for training and testing purpose.

Task - 02

Two layered CNN model was created.

It consists of input, hidden and output layer

Optimizer used was Adam

Test accuracy obtained was ~~0.9~~ 98 %

Prediction of the model was also compared with Actual truth label for image.

model training and testing was also performed on multi-layered CNN.

Test accuracy obtained was 0.99 %

Kindly refer the attached notebook (.ipynb) file or Github link to refer implementation of models

Task-03

Fine tunning of ~~model~~ pre-trained model VGG16 was also performed on the training dataset.

Test accuracy achieved was **99.50%.**

Kindly refer the Jypter notebook (ipynb) file or Github ~~link~~ link for code implementation.

Problem 4: Gradient Descent and Backdrop

Ques. 4.1

What is difference between Stochastic Gradient Descent and Mini-Batch Gradient Descent?

Answer

The main difference between SGD and Mini-Batch GD is no of training sample used to compute the gradient at each iteration.

In SGD

- only one sample / trainig instance record is randomly choosen from given dataset to calculate gradient

In Mini Batch GD

a small batch of samples / trainig instance / record is randomly choosen from given trainig dataset to calculate the gradient.

In SGD

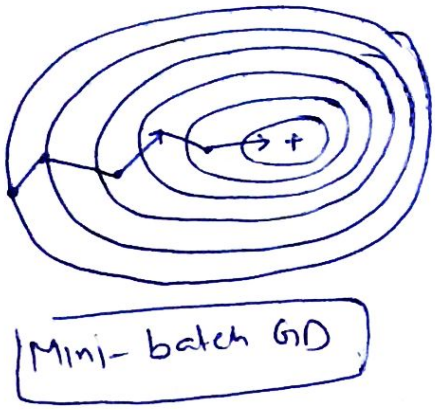
↳ convergence might be faster but gradient estimate might be noisy and not stable

In min-batch GD

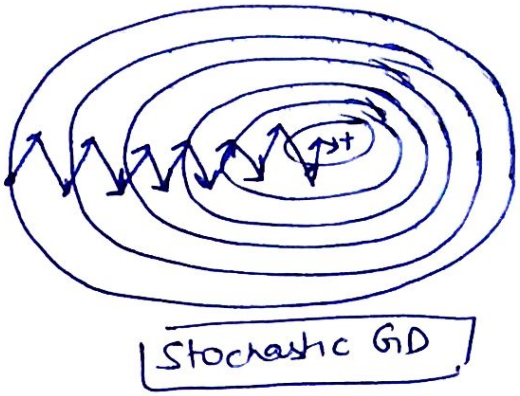
↳ convergence might take some time compared to SGD However gradient estimate will be less noisy and stable

Ques 4.1 cfd..

Mini batch GD is more compatable with Hardware due to its advantage of parallel process.



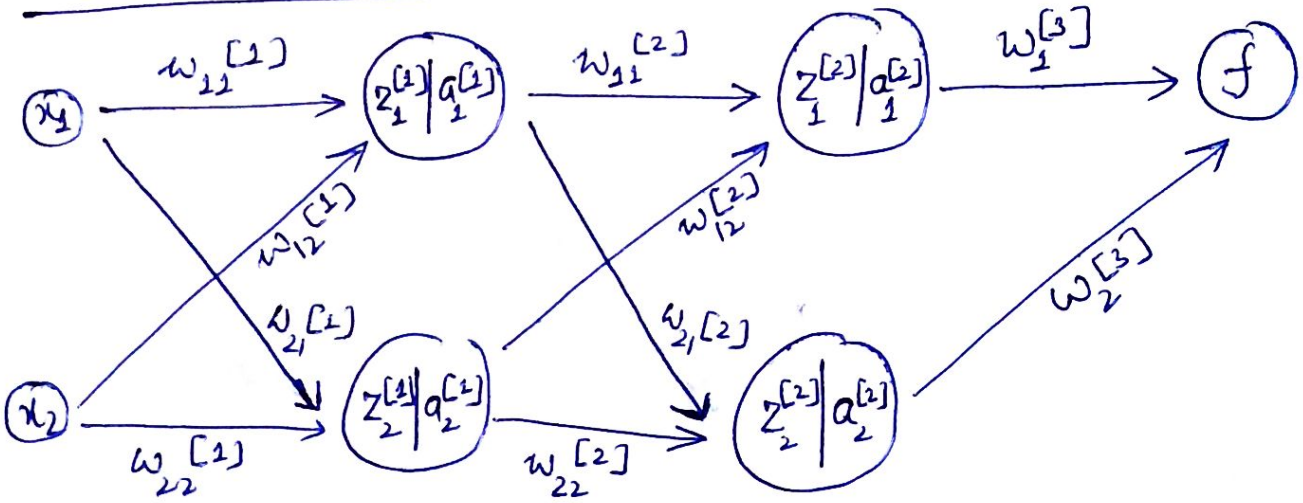
Mini- batch GD



Stochastic GD

Ques 4.2

Consider a 3-layer network shown in figure



$$Z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \end{bmatrix} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad A_1^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \end{bmatrix} = \begin{bmatrix} \sigma(z_1^{[1]}) \\ \sigma(z_2^{[1]}) \end{bmatrix}$$

$$Z^{[2]} = \begin{bmatrix} z_1^{[2]} \\ z_2^{[2]} \end{bmatrix} = \begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} \\ w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix} \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \end{bmatrix} \qquad A^{[2]}_2 \begin{bmatrix} a_1^{[2]} \\ a_2^{[2]} \end{bmatrix} = \begin{bmatrix} \sigma(z_1^{[2]}) \\ \sigma(z_2^{[2]}) \end{bmatrix}$$

Ques 4.2 ctd..

2. Consider a 3 layer network shown

Given $f = w_1^{[3]} a_1^{[2]} + w_2^{[3]} a_2^{[2]}$. Compute the following derivative

$$\frac{df}{dz_1^{[2]}}, \quad \frac{df}{dz_2^{[2]}}, \quad \frac{df}{dz^{[i]}}, \quad \frac{df}{dw_{11}}.$$

Question 5.1

What's the risk of tuning hyperparameter using a test dataset?

Answer -

If hyperparameter are tuned on the test dataset, model is exposed to test dataset or you can say model is also trained on test dataset which will make model to perform very good on the test dataset. Model will not learn the generalized pattern and model will perform poorly in the real world data or 'unseen' data.

Ques 5.2

Give two strategies for ~~the~~ addressing the overfitting problem in neural network.

Answers -

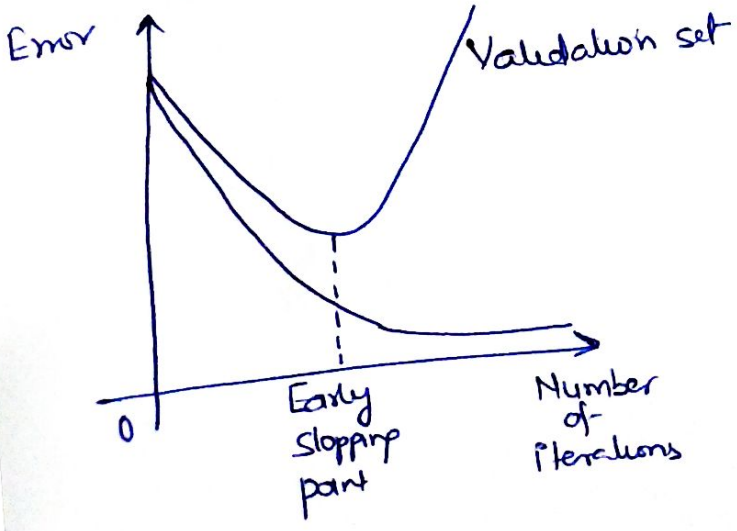
Here are the two strategy for addressing the overfitting prob:

1. Early Stopping - idea is to stop the training before a model starts to overfit. Early stopping is an optimization technique which is used to ~~pre~~ prevent model from overfitting. There are few ways using which we can introduce early stopping in model training

→ Monitor the loss function and stop model training when loss function is significantly low. ~~and model performance~~

→ Monitor the validation error

Keep training the model till the time validation error is decreasing. There will be a point during training when validation error will again begin to rise. Stop training the model at this time



2. Dropout

Dropout is one of the regularization technique used frequently to prevent model from overfitting.

During each iteration of model training, weights of some of the neurons in the network are set to zero. means those neurons don't participate in the model training for that iteration. Percentage of dropout of neurons for each iteration/epoch is set quite less between 1% to 5%..

Dropping random neurons during each iteration, prevents network from depending on single neuron or layer of neuron.

This introduces more generalization in network

Ques 5.3

How do you decide input layer and output layer size for solving a particular problem using a neural network ?

Answer:

Size of input layer and output layer in a neural network depends on the dataset on which model training is going to happen

1. Input layer → No. of neurons in input layer should exactly match the size of input data.
e.g. image size is 128 x 128. Then no. of neurons in input layer will be 16,384 (128 x 128).
if input data is time-series data of size 100 then input layer will be 100

2. Output layer

→ if Neural network is a regressor, output layer will have single node or it depends on size of target variable

→ if Neural Network is a classifier then it has neurons equivalent to no. of classes present in the dataset.

Ques 5.4

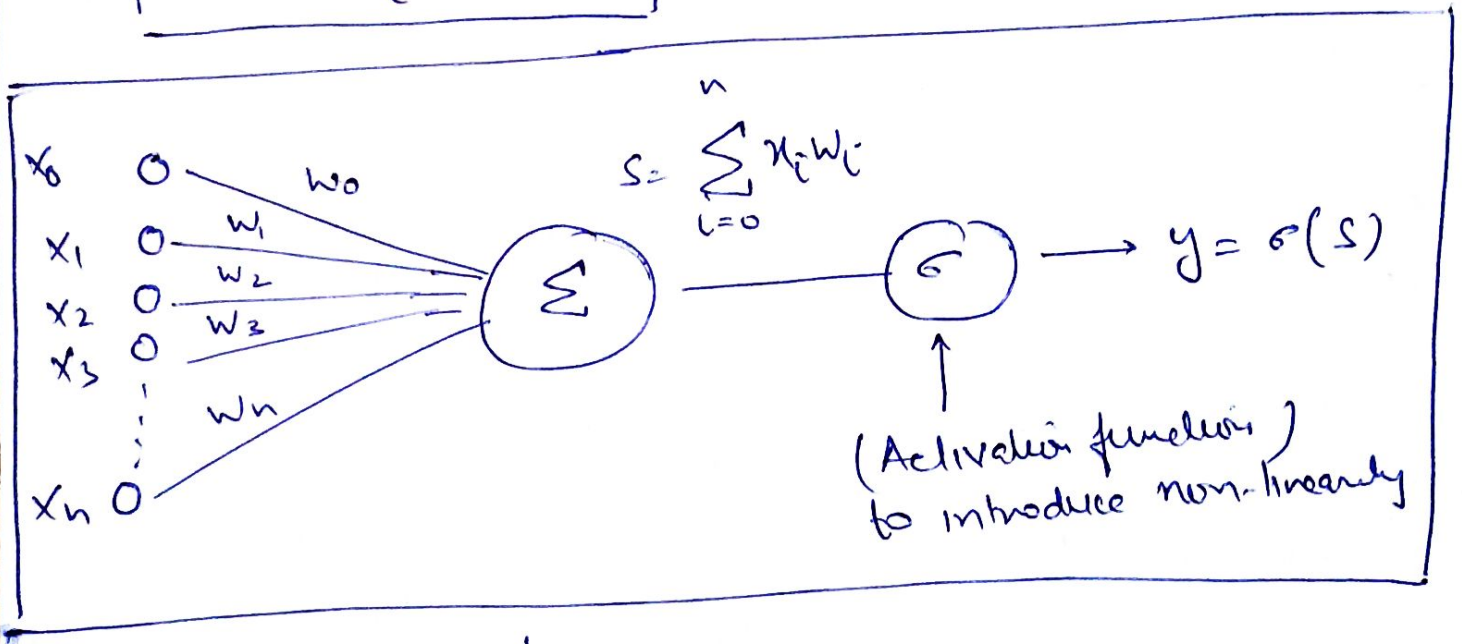
Explain the Sigmoid activation function.

Answer.

Sigmoid activation function is nothing but a mathematical function which is applied to the output of each neuron before output is fed as input to the next layer. The purpose of sigmoid activation function is to help network complex non-linear relationship between Input variables and output

mathematical formula

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



$X_i \rightarrow$ input parameters

$W_i \rightarrow$ weights to the parameters

Sigmoid activation function is good for binary classification

Question 5.5

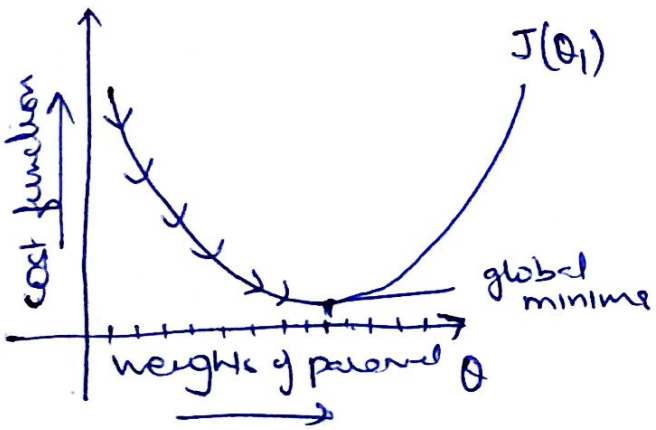
In a neural network what is the learning rate? Will the network converge faster, if I have a very large learning rate.

Answer

In neural network learning rate is a hyper-parameter which monitors the pace at which ml model / algorithm updates or learn the value of parameters. Basically learning rate adjusts the weight of our algorithm w.r.t loss gradient. Learning rate is also called ('α')

Here is the relationship

updated weight of parameter	=	current weight of parameter	−	learning rate	*	gradient

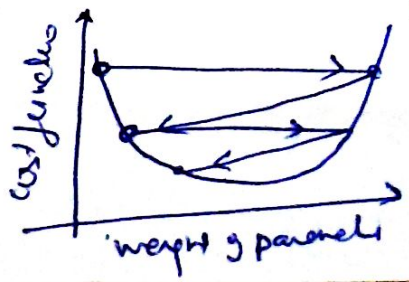


learning rate

$$\theta_1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

if 'α' is small, gradient descent will eventually achieve minima but time taking.

if we have very large learning rate, gradient descent may overshoot / miss the minima. It might fail to converge or even diverge



← (if α is too large)