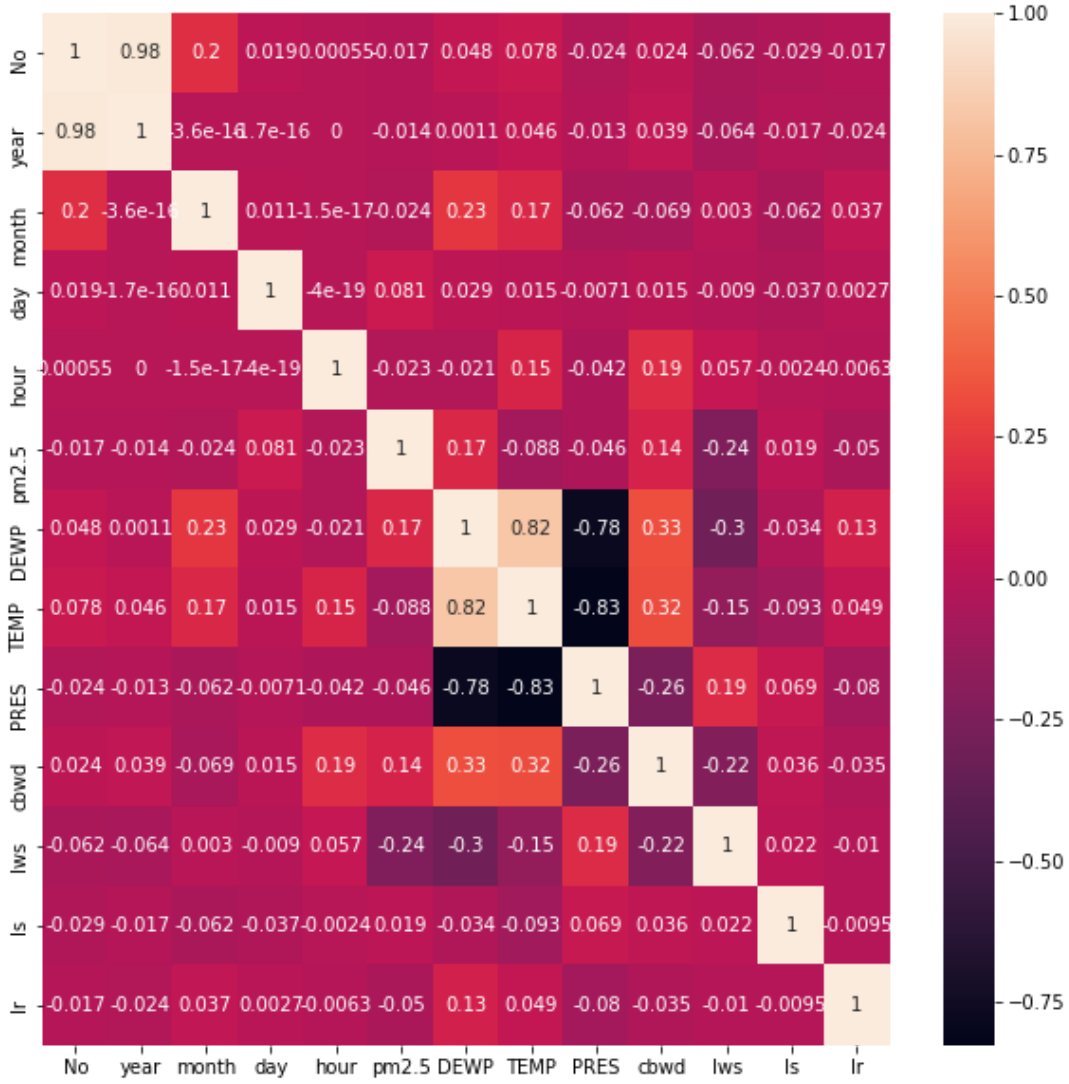
CSE343 Assignment-2, Monsoon '21, IIITD
Hardik Garg, 2019040

Q1

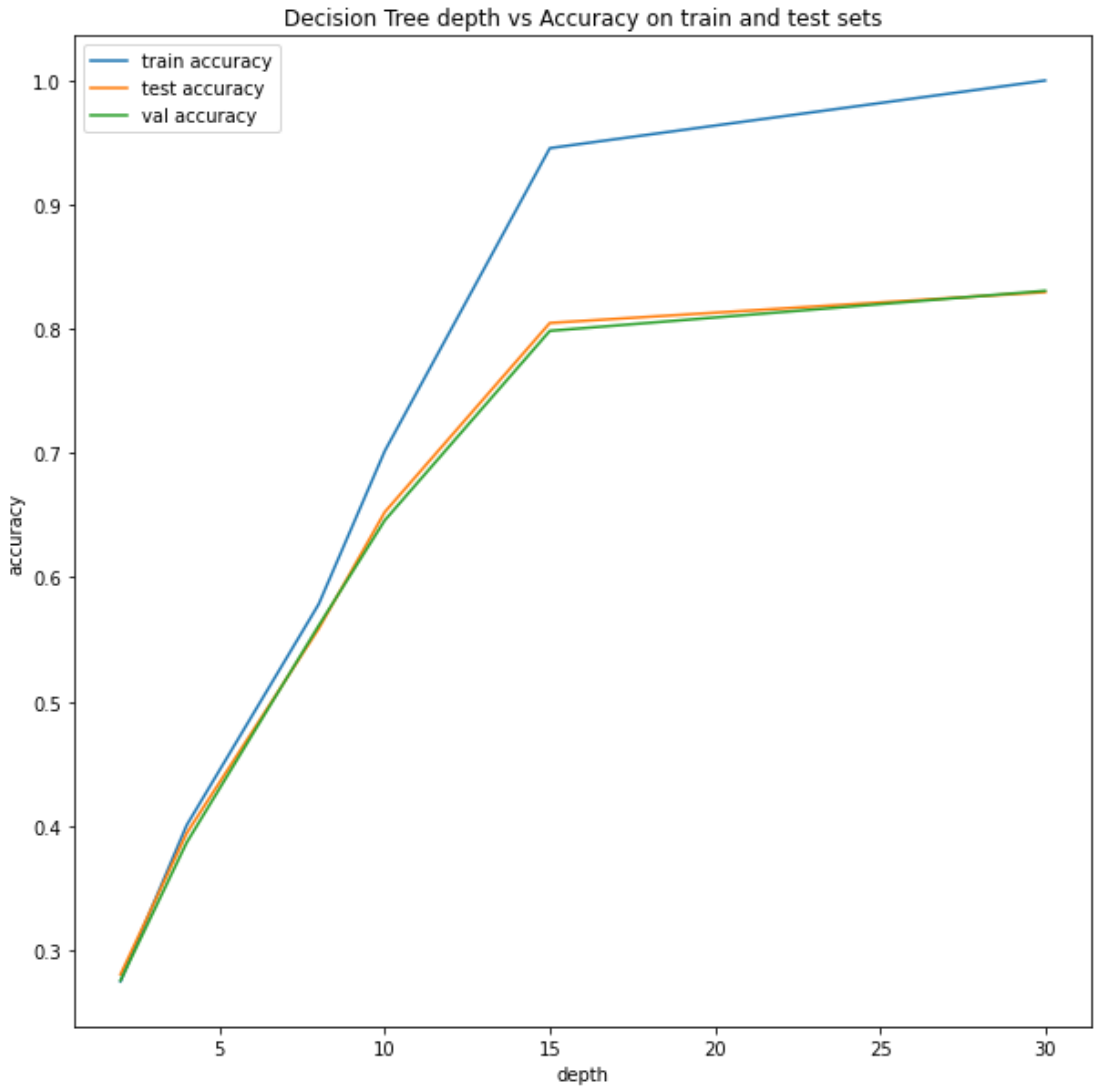
(a) Preprocessing - Only pm2.5 column had NaN values, which were handled by replacing all NaN values by the mean. The datatest was split into train (30677), test (6573) and validation (6573) sets. The values in cbwd column were converted from categorical strings to numeric values. The correlation matrix plotted for the data is -



The accuracy achieved on test set is as follows -
gini accuracy on test set: 0.8171306861402708
entropy accuracy on test set: 0.834778639890461

Clearly, entropy performs better in this case by 2.15%

(b) Plot of training and testing accuracy with varying depth is as follows (criterion used is entropy)-



From the plot we conclude that the depth 30 is showing maximum accuracy for both train and test sets. The accuracy is as follows -

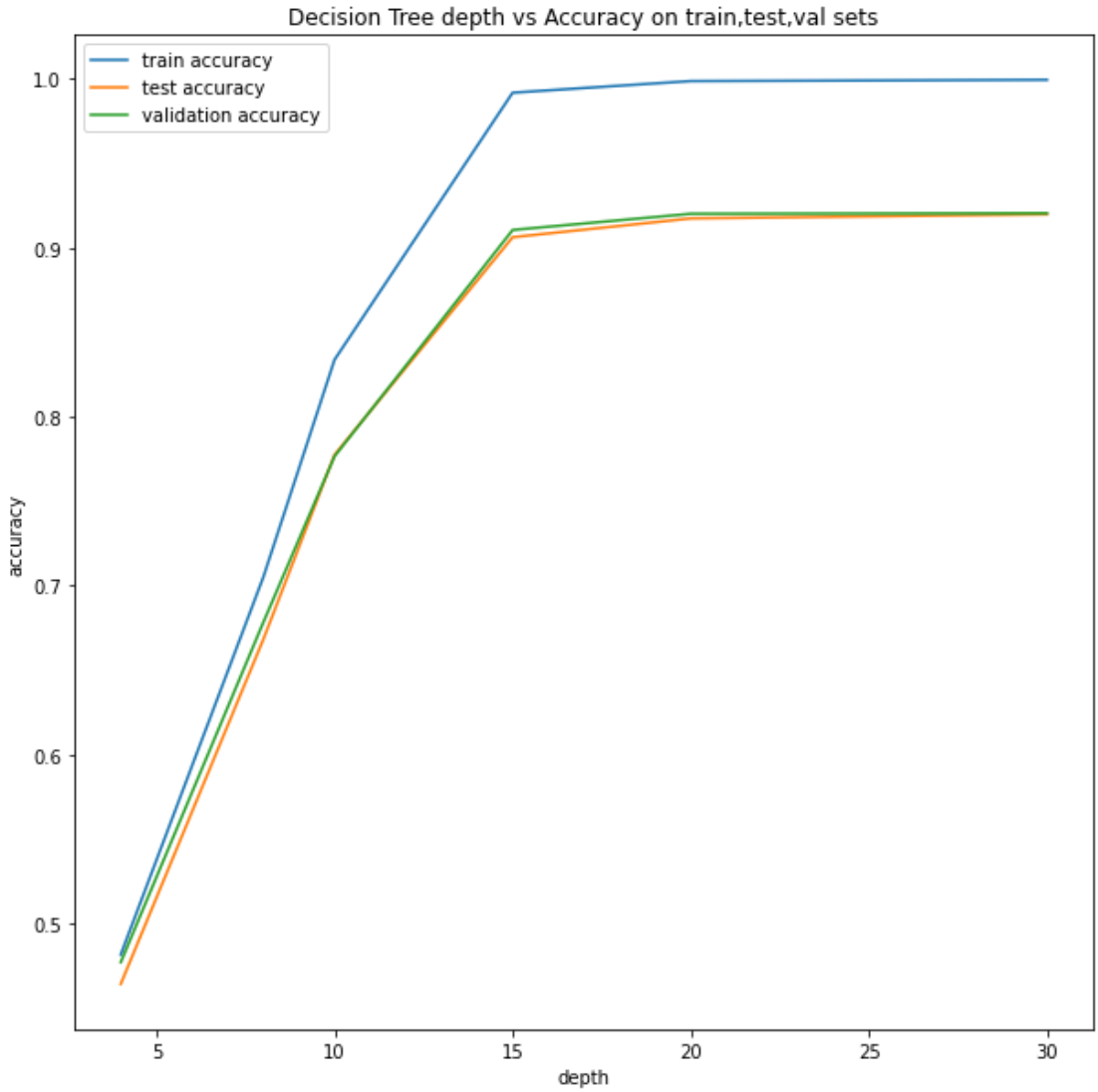
Train acc:
[0.27479870913061905, 0.4003650943703752, 0.5784137953515663, 0.7016331453531962, 0.9455618215601265, 1.0]
Test acc:
[0.2803894720827628, 0.3938840712003651, 0.5586490187129165, 0.6525178761600486, 0.8048075460216035, 0.8296059637912673]
Val acc:
[0.27547916032856706, 0.38637055065409187, 0.561149984788561, 0.6458777000304229, 0.7984484332217828, 0.8306966839062976]

(c) The random forest ensemble model is created with 100 decision stumps (max depth 3), trained on 50% data and the final decision is taken by majority vote. The accuracy achieved is -

train accuracy: 0.4330280014342993
test accuracy: 0.42431157766621025

This accuracy is very low as compared to the previous parts. The reason for this is the very low depth of the decision stump (3). The dataset has 10 features and a depth of 3 is not sufficient to consider all of them, hence the poor performance.

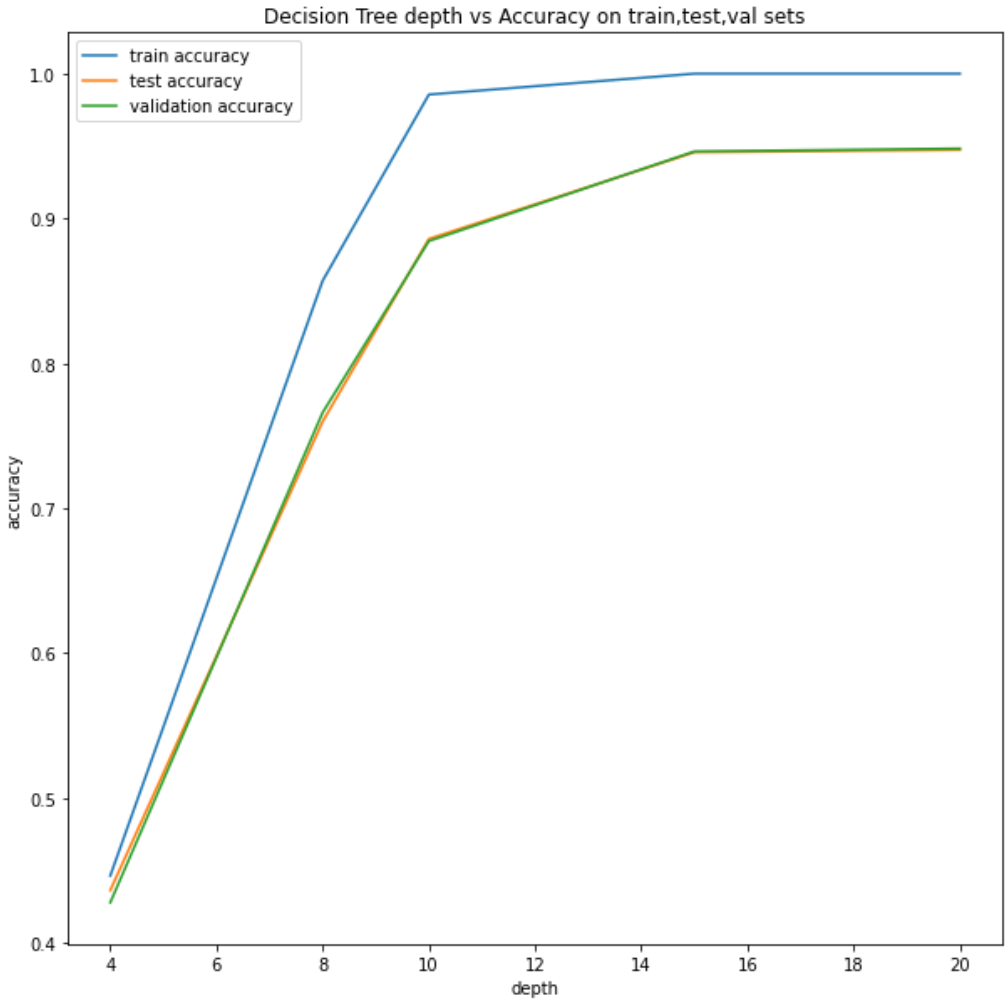
(d) The ensemble model was run with different depths. The number of trees was chosen to be 100 for uniformity with previous parts for fair comparison. The accuracy plot for train, test, val sets with depth is -



As we observe, the depth of 30 (the one from part (b)) gives the best accuracy for all the three sets - train, test, val. The accuracies are as follows -

Train acc:
[0.48185937347198227, 0.705023307363823, 0.8336864752094403, 0.9916223881083548, 0.9985005052645304, 0.9992176549206245]
Test acc:
[0.46447588620112584, 0.668188041989959, 0.777422790202343, 0.9061311425528678, 0.9172371824129013, 0.9196713829301689]
Val acc:
[0.4773349558868269, 0.6782780651049589, 0.7765439610587161, 0.9104046242774566, 0.9199878308487983, 0.9202920596288409]

(e) AdaBoost classifier was trained for values [4,8,10,15,20] with DecisionTree as the base classifier and number of estimators as 100, for fair comparison across parts. The plot obtained is as follows -



As we observe, we get the best results for depth=20. The exact accuracy values are as follows -

Train acc:
[0.4461322815138377, 0.8570590344557812, 0.9855918114548359, 1.0, 1.0]
Test acc:
[0.4360261676555606, 0.7597748364521527, 0.8858968507530808, 0.9456869009584664, 0.947360413814088]
Val acc:
[0.42759355034986307, 0.7660480681472467, 0.884393063583815, 0.9463036203224825, 0.9482811073927594]

AdaBoost gives better accuracy than RandomForest for higher depths (10,20) by approximately 3.12%

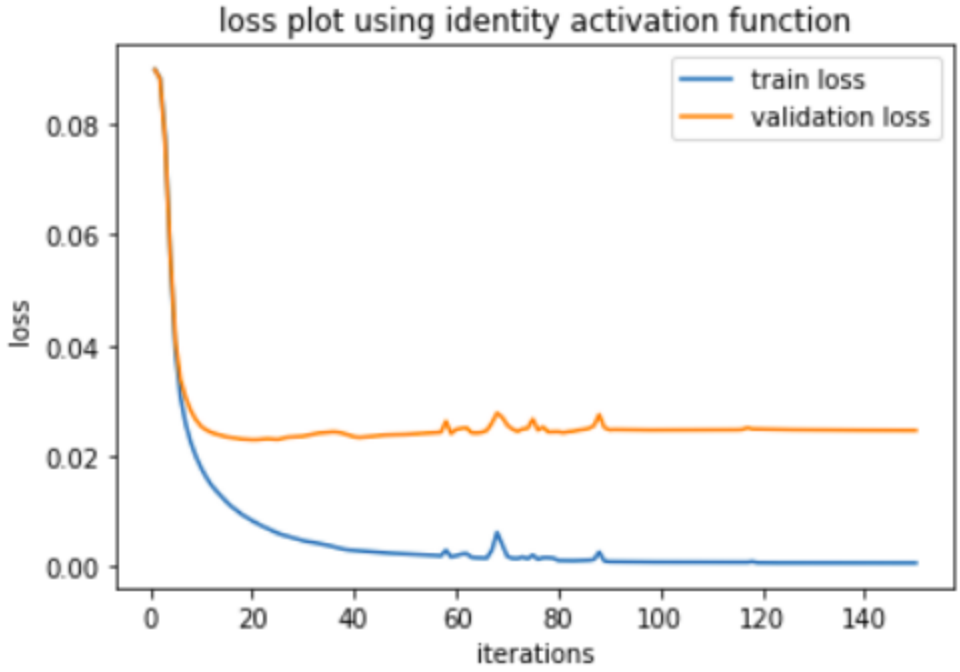
Q2 Overview - MyNeuralNetwork class is implemented with the functions specified in the question. Along with it, helper classes to represent different layers in the neural network are also implemented. The working of each function is documented in the code itself.

1.

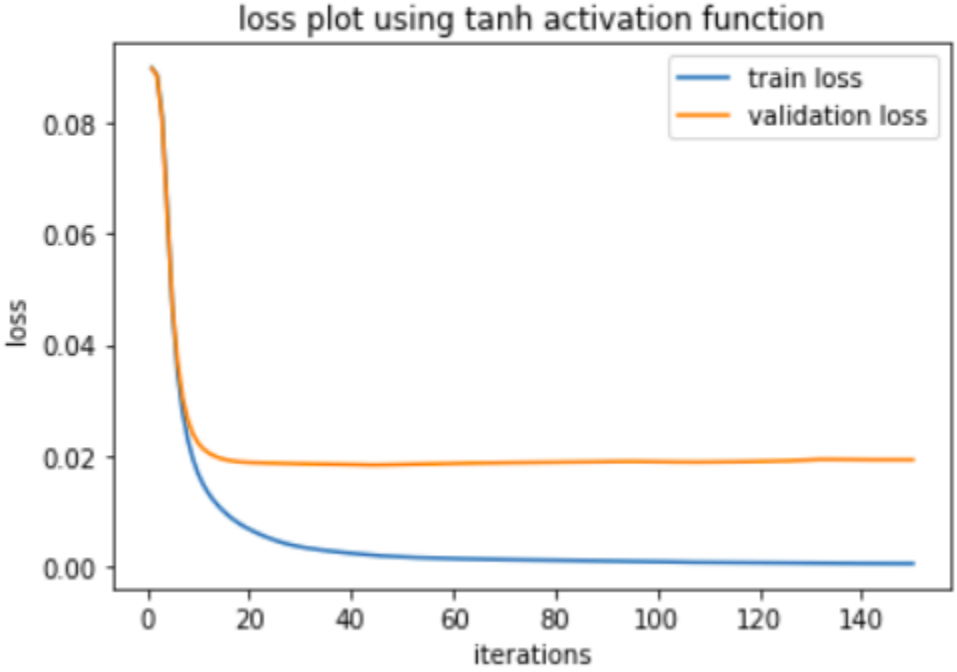
Activation Function	Test Accuracy	Validation Accuracy
ReLU	0.853	0.856
leaky ReLU	0.852	0.857
tanh	0.869	0.876
sigmoid	0.796	0.796
identity	0.851	0.854

2. The loss plots for training and validation data are as follows -

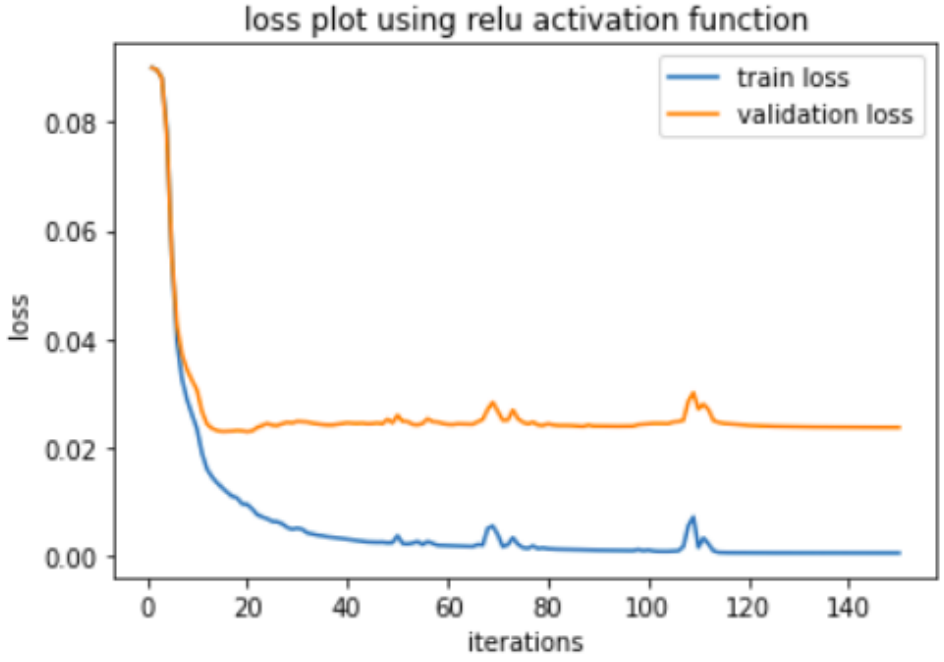
```
test acc using identity is: 0.8506428571428571
val acc using identity is: 0.854
```



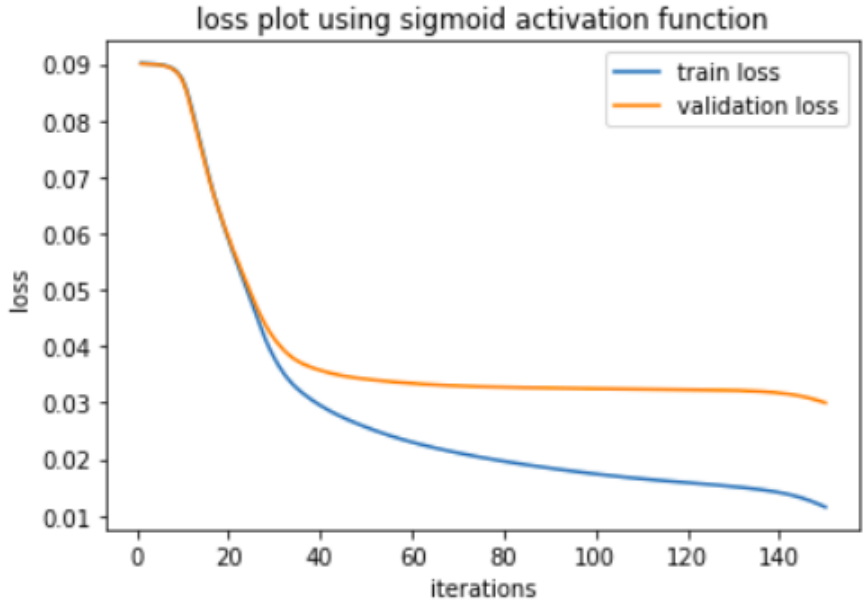

```
test acc using tanh is: 0.8695
val acc using tanh is: 0.8762857142857143
```



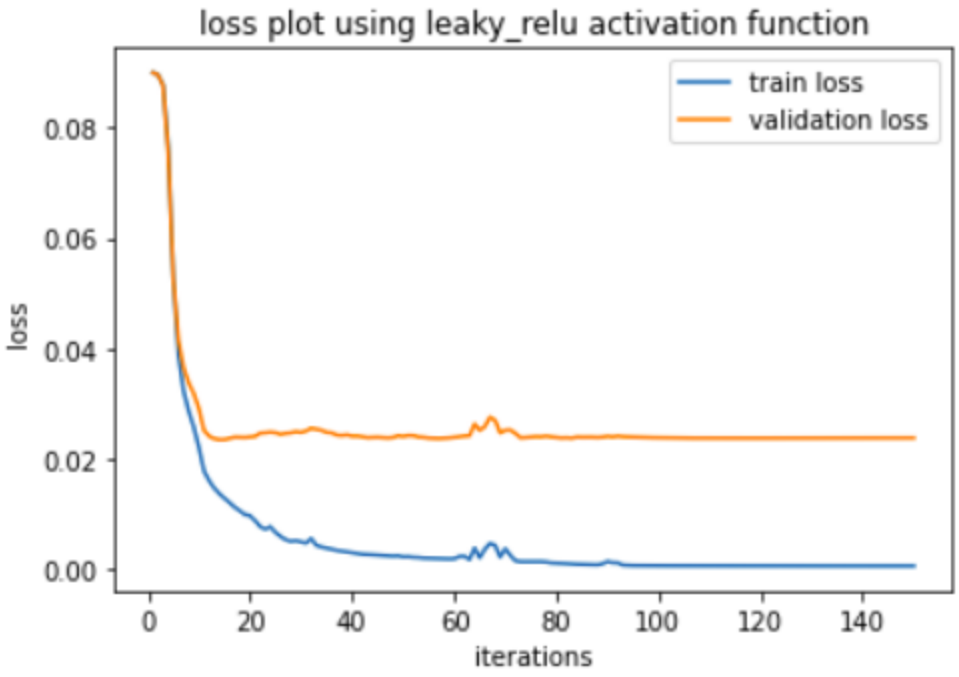
```
test acc using relu is: 0.8532857142857143
val acc using relu is: 0.8565714285714285
```




```
test acc using sigmoid is: 0.7967857142857143
val acc using sigmoid is: 0.7961428571428572
```



```
test acc using leaky_relu is: 0.8527142857142858
val acc using leaky_relu is: 0.8572857142857143
```



Leaky ReLU seems to give the best results out of all activation functions. The reason for this is that it has the features of ReLU (simple and linear function) and it doesn't have zero slope for the negative parts which ensures better convergence. Also, it is computationally inexpensive.

3. In every case, the activation function in the output layer should be **softmax** because we are performing a multiclass classification on MNIST dataset (10 classes) and softmax gives us the probability of each class towards the end, and the class having maximum probability is picked as the y_pred. Softmax is the generalised version of sigmoid which is used in binary classification. Other activation functions (such as ReLU, tanh etc) do not give us values between 0 and 1 and the values can also be negative which makes it difficult to make a prediction. Therefore, softmax is used in the final layer. Layers prior to it can have any activation function as per requirement.

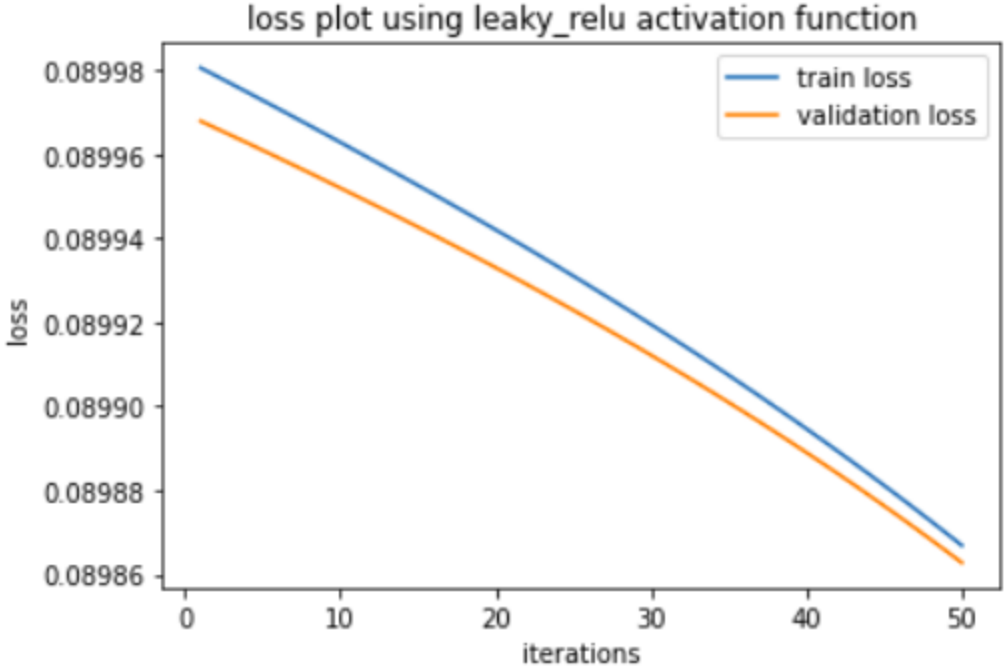
4. Inbuilt model is run for 150 epochs, learning rate 0.08. Table showing results of inbuilt models is as follows -

Activation Function	Test Accuracy	Validation Accuracy
tanh	0.877	0.876
relu	0.879	0.875
sigmoid	0.877	0.875
identity	0.857	0.859

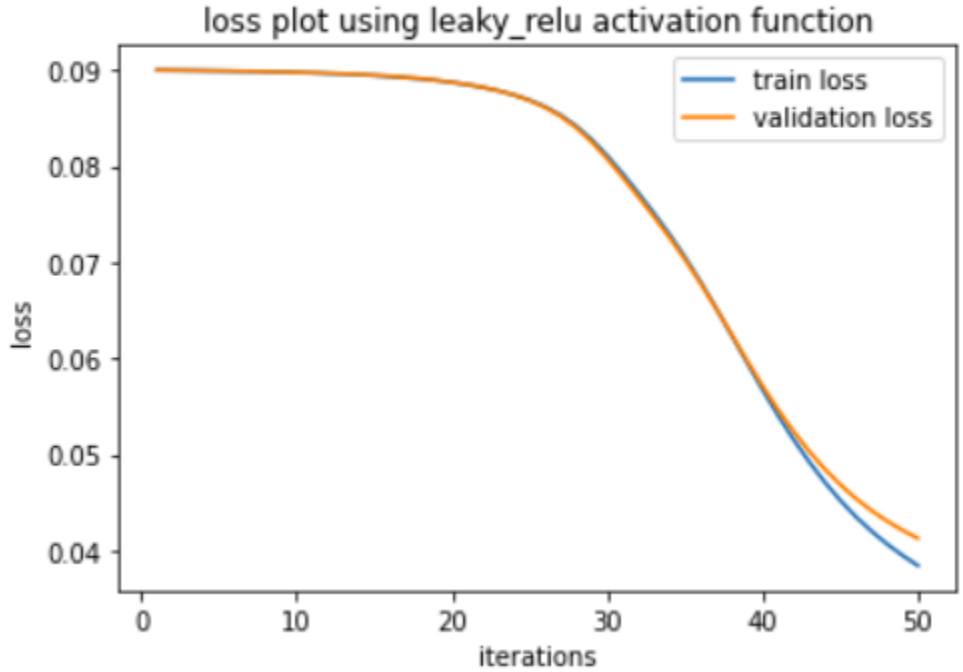
   The inbuilt model performs slightly better than the self implemented one. This is because of the following reasons -
   1. Inbuilt model uses the “adam” optimizer by default for updating weights whereas self implemented model uses “SGD”
   2. Inbuilt model has small optimizations - it stops when the change in weights is too small
   3. Input format is slightly different in both the models

5. Epochs=50, activation=leaky ReLU. The plots at different learning rates are as follows (lr = 0.001, 0.01, 0.1 and 1 => in this order)

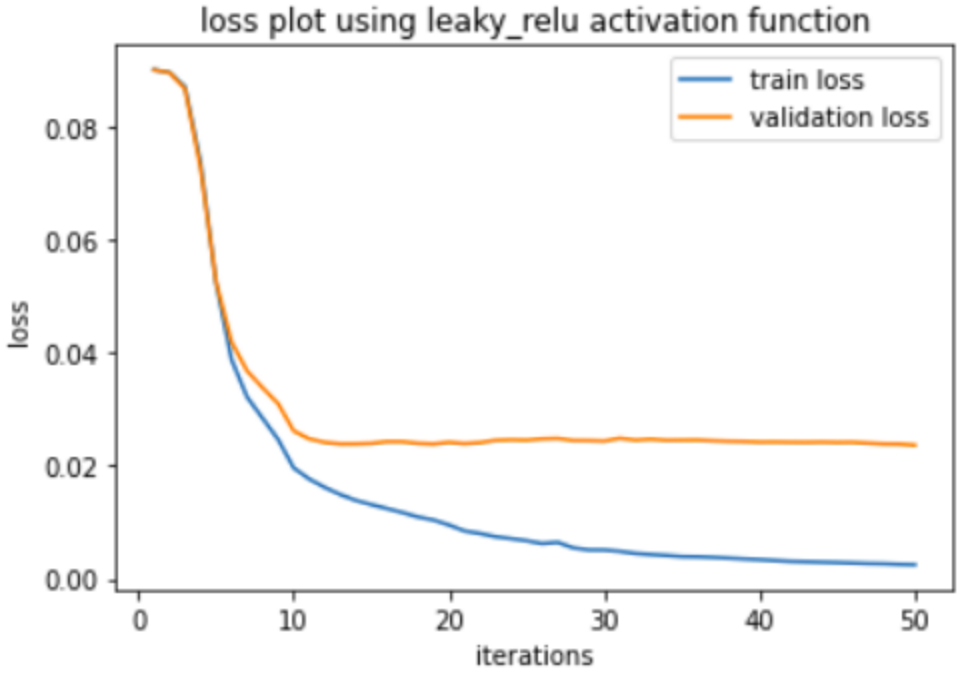
```
test acc using leaky_relu is: 0.221571428571428 56
val acc using leaky_relu is: 0.23314285714285715
```



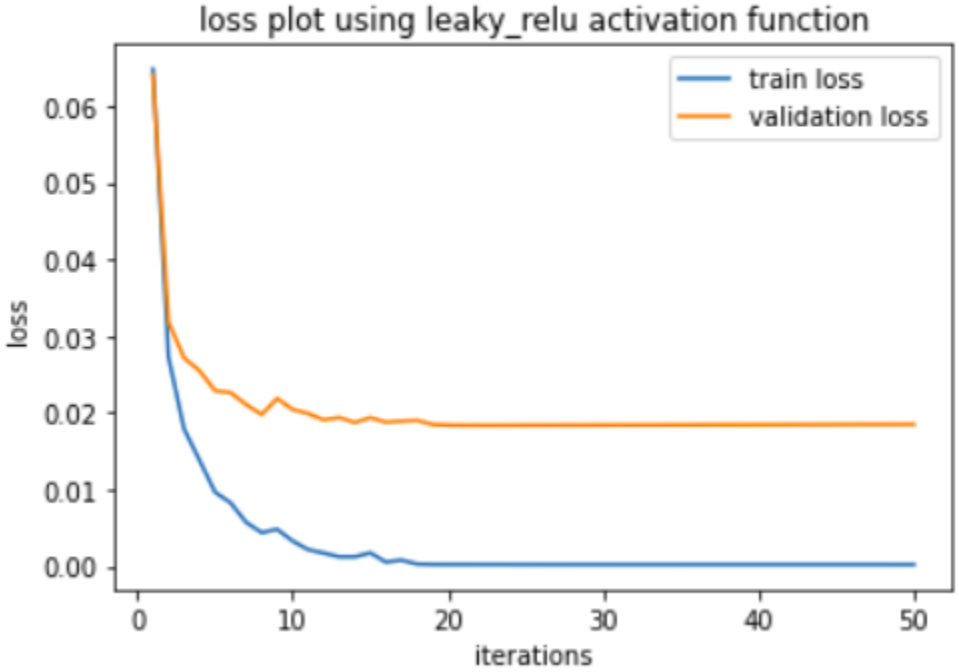

```
test acc using leaky_relu is: 0.7285
val acc using leaky_relu is: 0.7242857142857143
```



```
test acc using leaky_relu is: 0.8477857142857143
val acc using leaky_relu is: 0.848
```

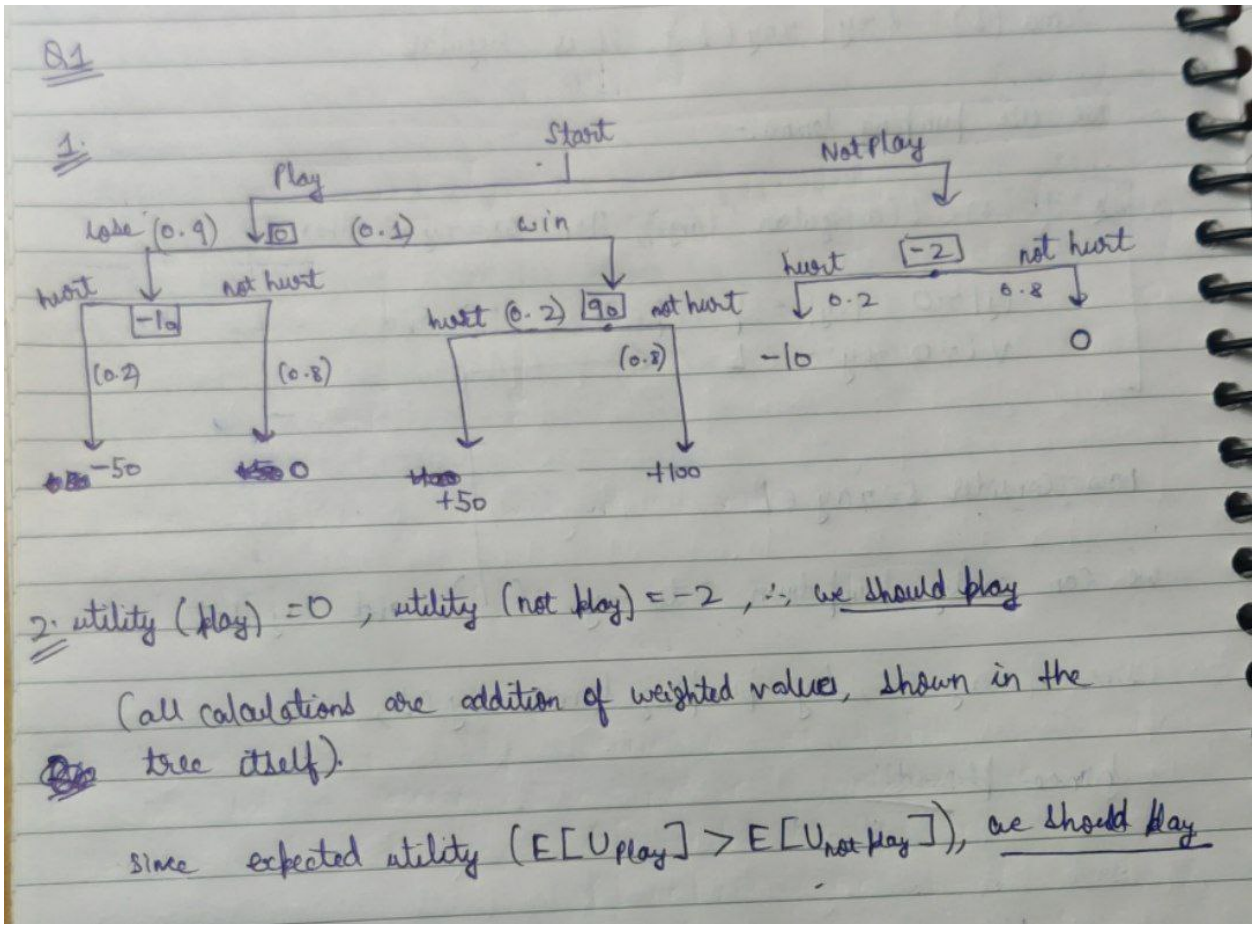



```
test acc using leaky_relu is: 0.8835
val acc using leaky_relu is: 0.8885714285714286
```

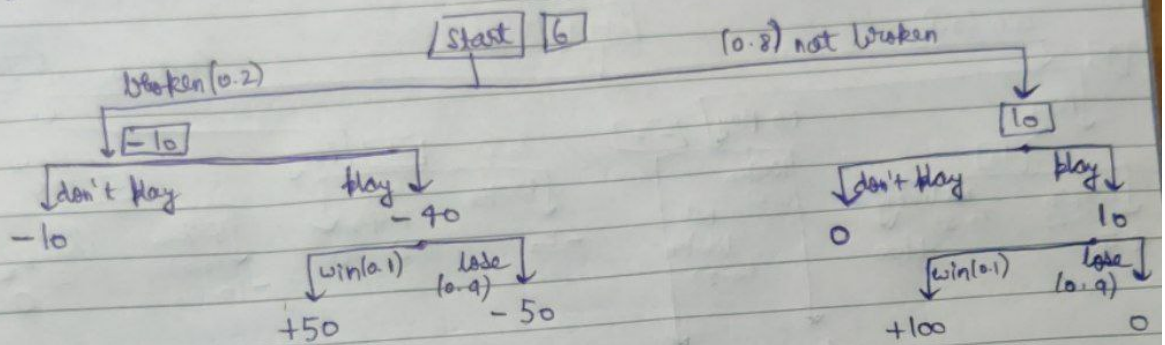


We observe that learning rates 0.1 and 1 give the best accuracy. Out of these, we use **0.1 as the learning rate** to prevent overshoot after a large number of epochs for learning rate 1. Other learning rates are too small as we can infer from the plots also, that the loss increases very slowly for them.

Q3 Theory Questions



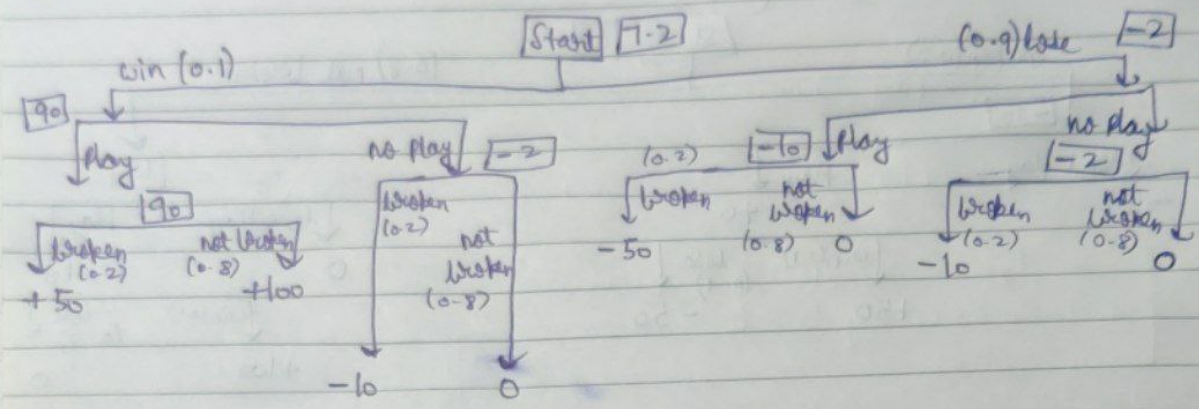
3. DT given perfect information about leg:



$\text{req'd val} = E[U_{\text{leg info}}] - E[U_{\text{no info}}] = 6 - 0 = \boxed{6}$

(assuming he plays if leg not broken and vice versa)

4 DT given perfect info about winning:-



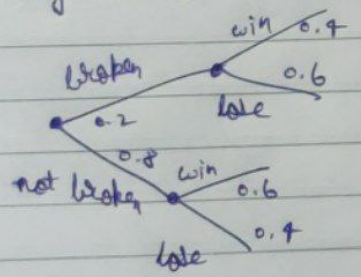
assuming he plays if he wins and vice versa,

reqd val = $E[U_{win\,info}] - E[U_{no\,info}] = 7.2 - 0 =$ 7.2

5. Currently, the probability of winning is same whether leg is broken or not. But, if winning has to depend on leg condition we can use conditional probability ⇒ P[win | broken leg] and P[win | not broken leg] will be different.

⇒ In tree diagram, always ensure that winning branch comes after the leg branch.

eg.

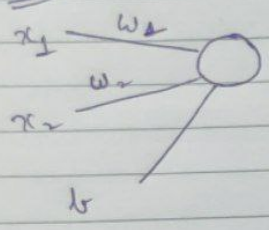


(just an example)

A2. $f: \{0,1\}^d \rightarrow \{0,1\}$

Since all inputs are boolean and output is also boolean, they can be represented as a combination of logic gates. All logic gates can be represented using a single layer neural network. Eg:-

AND

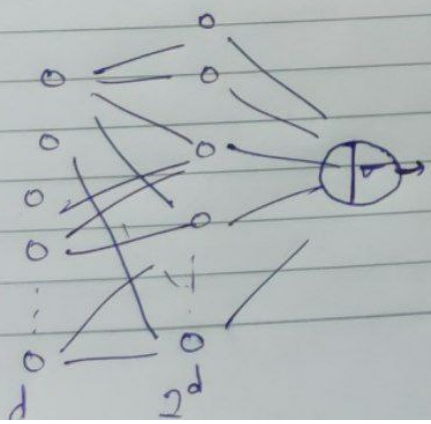


Let $w_1 = 10, w_2 = 10$ and $b = -15$

x_1	x_2	$w_1x_1 + w_2x_2 + b$	$x_1 \cdot x_2$
0	0	-15	0
0	1	-5	0
1	0	-5	0
1	1	5	1

⇒ We can take sigmoid of $w_1x_1 + w_2x_2 + b$ and round it off to 0 or 1. Similarly, other gates can be simulated.

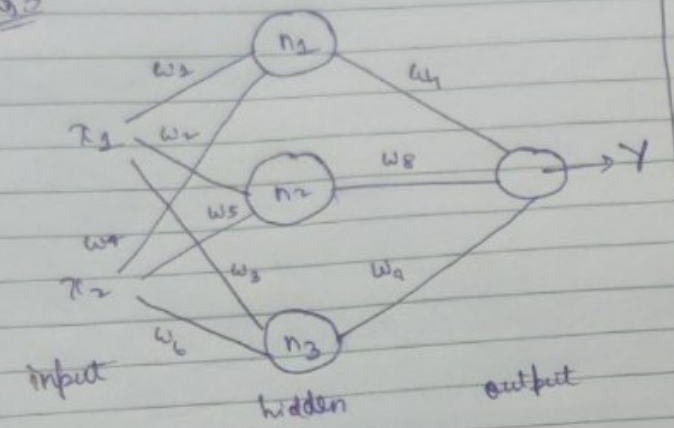
∴, we have 1 hidden layer of 2^d neurons to represent all possible inputs and the final boolean fn. is represented through logic gate neural network, followed by sigmoid activation function ⇒ $\frac{1}{1+e^{-z}}$



eg. $x_1 \text{ xor } x_2 = x_1x_2' + x_1'x_2$

↑ hidden layer

Q.3



$n_1 = (z_{11} | a_{11})$

$n_2 = (z_{12} | a_{12})$

$n_3 = (z_{13} | a_{13})$

$Z_1 = [z_{11}\ z_{12}\ z_{13}]$

$A_1 = [a_{11}\ a_{12}\ a_{13}]$

$Y = \boxed{Z_2 | A_2}$

$$W_1 = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix}$$ (b/w input and hidden)

(all w_i's $\in \mathbb{R}$)

$$W_2 = \begin{bmatrix} w_7 \\ w_8 \\ w_9 \end{bmatrix}$$ (b/w hidden and output).

$X = [x_1\ x_2]$ $(x_i \in \mathbb{R})$

$Z_1 = XW_1$

$A_1 = \text{activation}(Z_1)$ → activation = ReLU = $\max(0, x)$ = ReLU(x)

$Z_2 = A_1 W_2$

$A_2 = \text{Sigmoid}(Z_2)$ → $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$

and $Y = \underset{y}{\text{argmax}}\ (P(Y=y|X))$