Tutorial Letter A2S/0/2023

Machine Learning

COS4852

Year module

School of Computing

IMPORTANT INFORMATION

This document contains a discussion on Assignment 2 for COS4852 for 2023.

BAR CODE

CONTENTS

Page

1 INTRODUCTION

This document contains a discussion on Assignment 2 for COS4852 for 2023.

2 Assignment 2

Question 1

Read Chapter 4 of Nilsson's book (which you downloaded for Assignment 1). Take special note of section 4.1 and its discussion of decision boundaries and their polarity and the concept of the neural network weight-space. The terms TLU and Perceptron refer to the same construct here (i.e. a neuron with weights and a threshold activation function). Similarly, the terms hyperplane, decision boundary, and decision surface refer to the same concept, which linearly divides a space into two sub-spaces. The space can have any number of dimensions. In a 2-dimensional space the decision boundary is a straight line. There is a direct mapping between the decision boundary(s) and the weights of the neural network.

Question 1(a)

Consider Case (a), as shown in Figure 1. The figure shows the instance space of a Perceptron (a single neuron) with its decision boundary. This is Case (a). Positive instances are marked as P_i and negative instance as N_i:

$$\begin{aligned}
P_1 &= (\text{-}4, 4)\\
P_2 &= (1, 4)\\
P_3 &= (2, 6)\\
P_4 &= (6, 2)\\
N_1 &= (\text{-}3, 1)\\
N_2 &= (1, \text{-}5)\\
N_3 &= (6, \text{-}4)\\
N_4 &= (4, \text{-}2)
\end{aligned}$$

Calculate the values of the weights of the Perceptron, using the position of the decision boundary. The cut-off points of the decision boundary on the axes are at $(6, 0)$ and $(0, 2)$.

Discussion on Question 1(a)

The decision boundary of a Perceptron with two inputs, x and y must satisfy the equation:

$$w_0 + w_1 x + w_2 y = 0$$

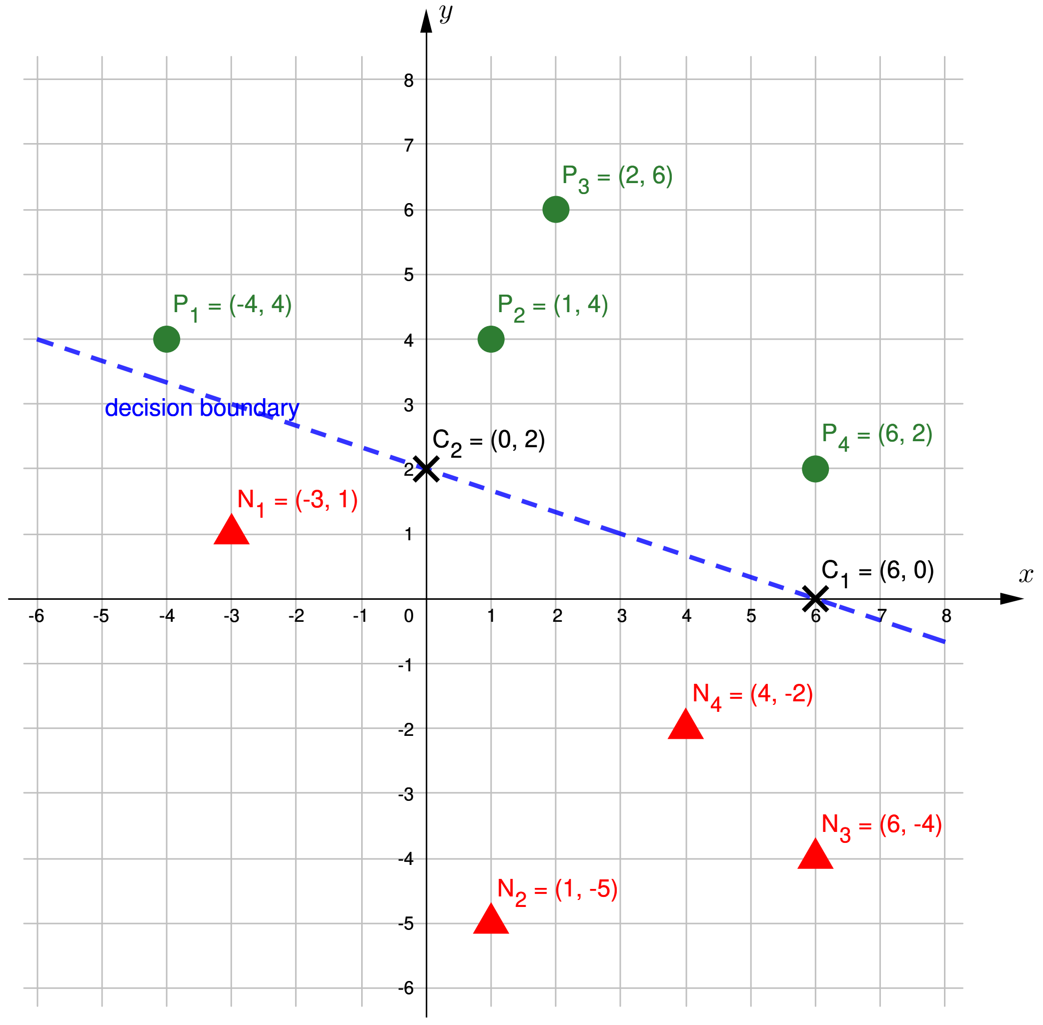


Figure 1: Case (a): instances and the decision boundary of a Perceptron.

Two known points on the line of the decision boundary are the intersection points with the axes, $C_1 = (6, 0)$ and $C_2 = (0, 2)$. Inserting these values into the equation for the Perceptron decision boundary gives:

$$w_0 + 6w_1 = 0$$

$$w_0 + 2w_2 = 0$$

Which gives:

$$w_0 = -6w_1$$

$$w_0 = -2w_2$$

Any values of w_0 , w_1 and w_2 that satisfy these two equations will suffice. Choose $w_0 = 6$. This gives $w_1 = -1$ and $w_2 = -3$. To summarise:

$$\begin{aligned} w_0 &= 6 \\ w_1 &= -1 \\ w_2 &= -3 \end{aligned}$$

The decision boundary has a polarity, as shown in Figure 2. In other words it has a positive and negative side, classifying all instances on one side as positive and those on the other side as negative. This is due to the threshold activation function used by the Perceptron.

There are an infinite set of values for w_i that will describe a line that corresponds to the decision boundary, but only half of these will classify the data correctly. To test whether the calculated weight values correctly classifies the data substitute the (x_1, x_2) values of any example into the equation of the decision boundary, using the calculated weights. Choose the positive instance $P_1 = (-4, -4)$:

$$\begin{aligned} & w_0 + w_1x + w_2y \\ = \; & 6 + (-1)(-4) + (-3)(4) \\ = \; & -2 \\ < \; & 0 \end{aligned}$$

The calculated weights classifies the positive instance at P_1 as negative. This means that the weight values are therefore incorrect. The reason for this is due to the polariry of the decision boundary or decision boundary.

We can get the exact same line by changing the sign of initial weights, to become:

$$\begin{aligned} w_0 &= -6 \\ w_1 &= 1 \\ w_2 &= 3 \end{aligned}$$

which gives another valid equation for the line, but classifies the data point correctly. Test P_1 again:

$$\begin{aligned} & w_0 + w_1x + w_2y \\ = \; & -6 + (1)(-4) + (3)(4) \\ = \; & 2 \\ > \; & 0 \end{aligned}$$

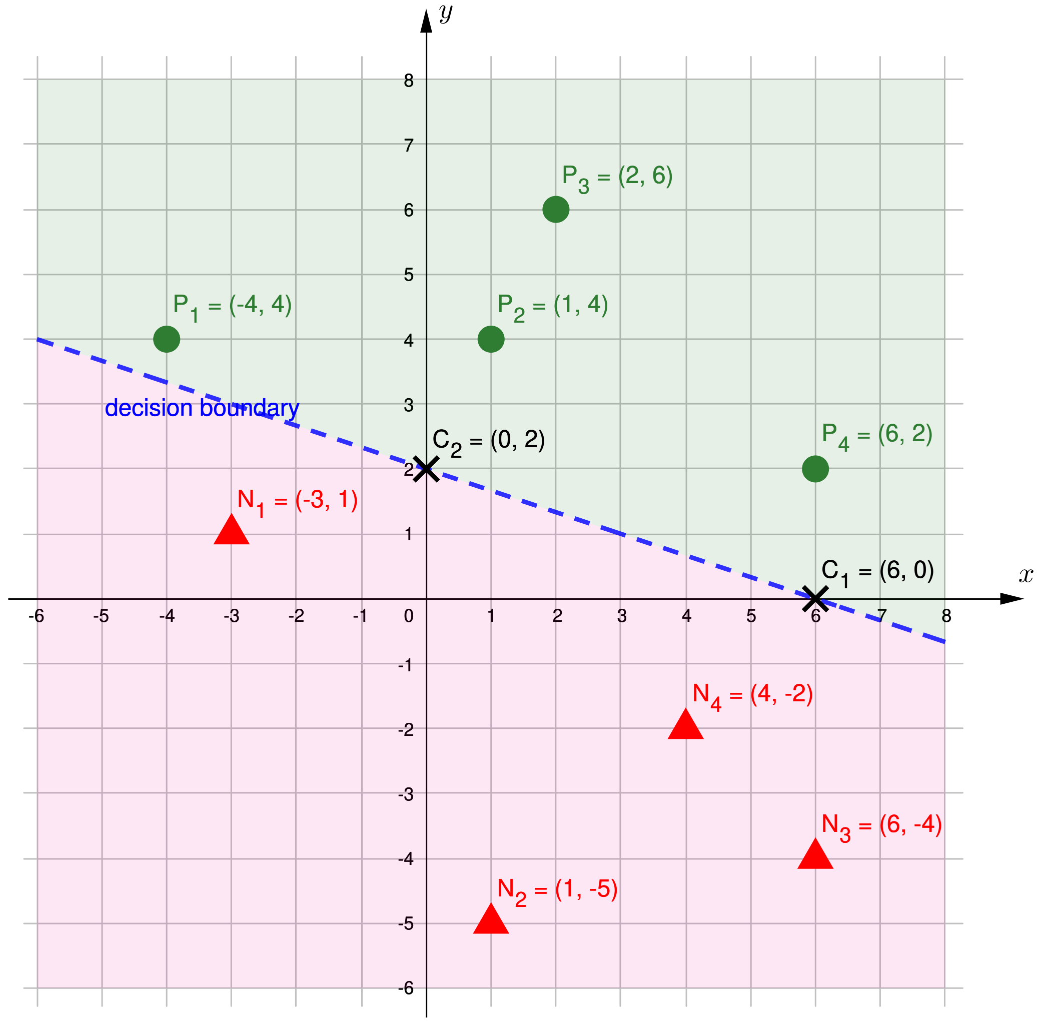


Figure 2: Case (a): polarity of the decision boundary of a Perceptron.

thereby correctly classifying the positive instance P_1 as positive. These weights are therefore correct. The resulting correct polarity is shown in Figure 2. All the instances should be checked to make sure that there is not some other calculation error, but is left as an exercise to the reader (you).

Question 1(b)

Consider Case (b), as shown in Figure 3. The figure shows the instance space of another Perceptron.

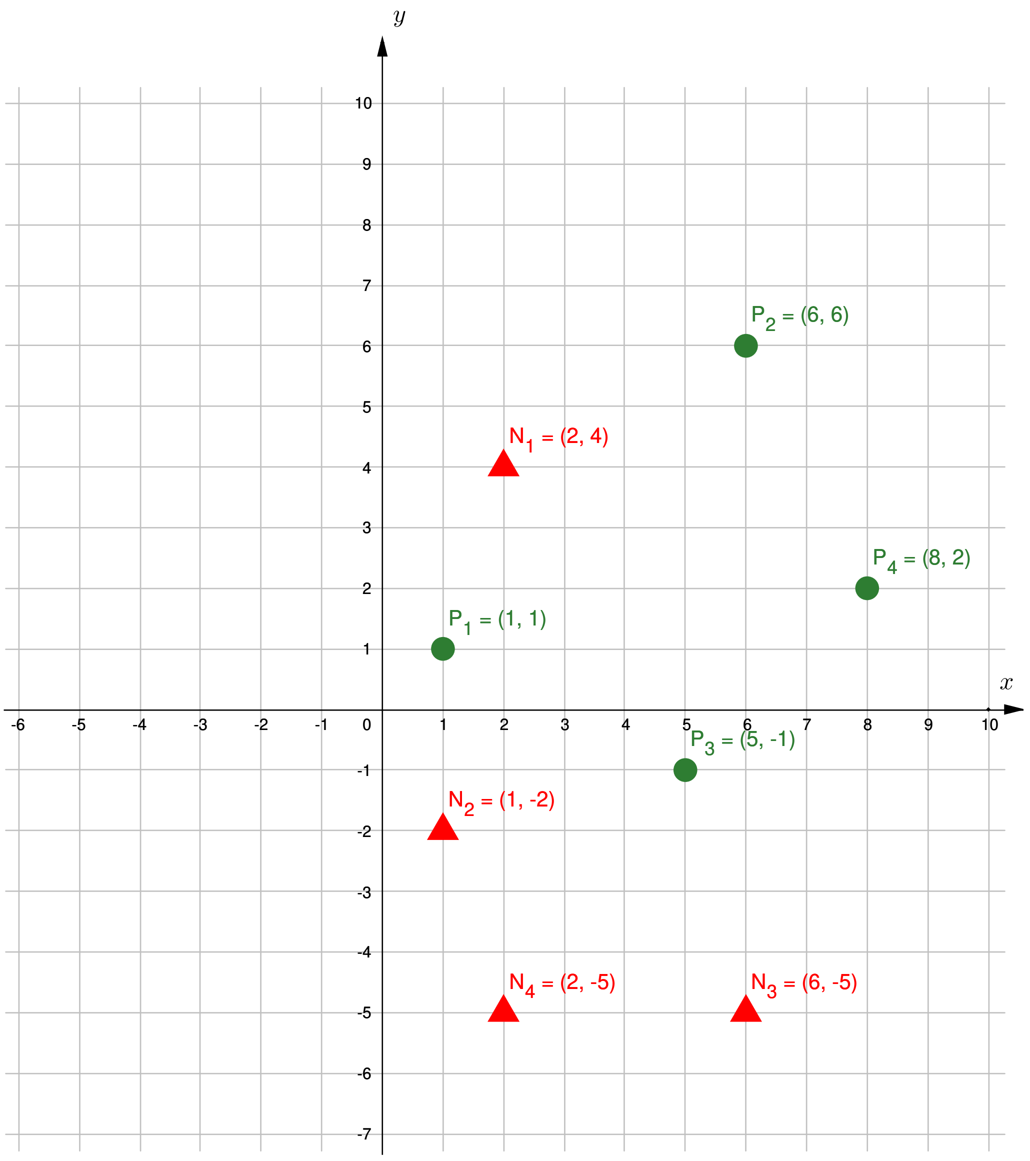


Figure 3: Case (b): Positive and Negative instance data for a Perceptron.

Calculate the weights for a single-unit Perceptron (i.e. one neuron) that will classify the given instances with the lowest error. Derive the decision boundary for this neuron. Draw a diagram showing the correct decision boundary. Discuss your solution.

What is different about this problem than the one in Question 1(a)? How would you go about finding a neural network that will classify this data 100% correctly? Discuss.

Discussion on Question 1(b)

The short answer is that the instances as given in Figure 3 are not linearly separable. In other words, there is no single line that will separate all the positive instances from all the negative instances. This, in turn, means that a single Perceptron cannot classify all the instances correctly.

It is possible to find a single line (per implication a single Perceptron) that *minimises* the classification error, but there will still be incorrect classifications. A possible decision boundary (using the shortest distance from the centroids of each set of the positive and negatives instance respectively, to the decision boundary) is given by the equation:

$$-7.56x - 5.44y + 7.34 = 0$$

as shown in Figure 4.

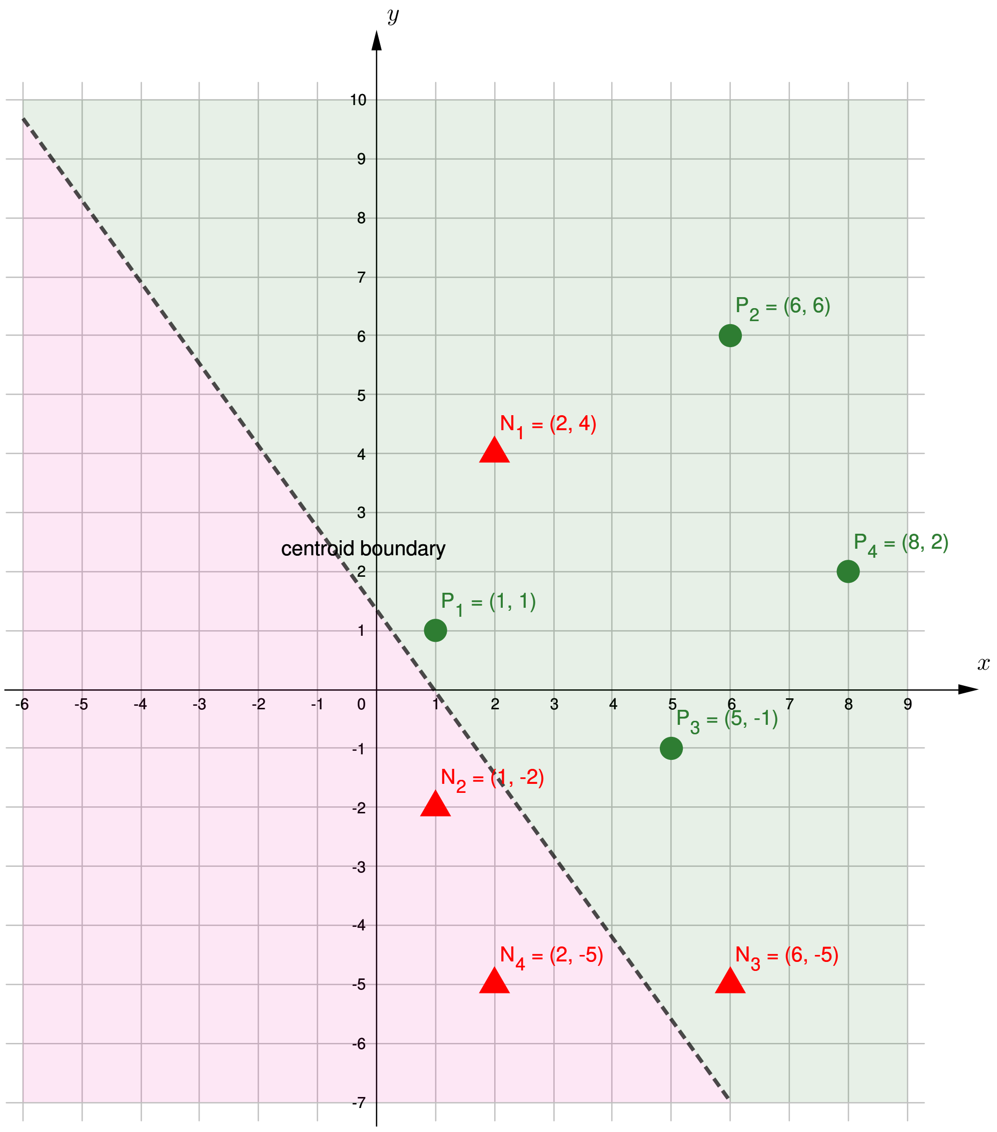


Figure 4: Case (b): A possible decision boundary.

Notice that with this decision boundary the negative instance N_1 is incorrectly classified as positive.

To get a correct classification, a multi-layer neural network is required, with at least two neurons in the hidden layer.

As an exercise, derive the equation for a valid decision boundary, using the least squares errors method.

As a further exercise, design a neural network, with 2 hidden layer neurons, and a single output, that will correctly classify this data set. Figure 5 shows what the decision boundaries of the 2 hidden layer neurons of such a network may look like. Use the cut-off points in this figure to calculate the weights.

HINT: Take a look at this Youtube video on the XOR problem – which is very similar to our problem here.

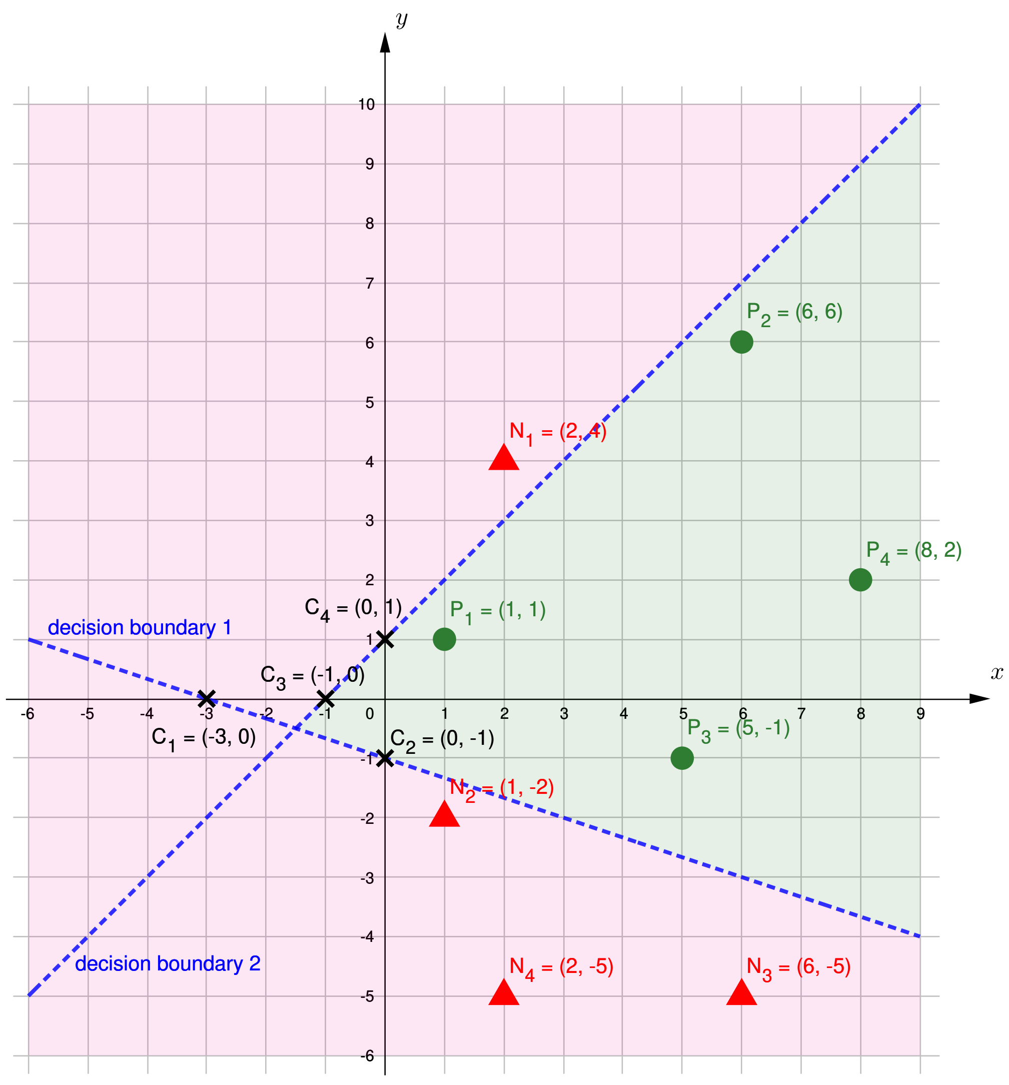


Figure 5: Case (b): Decision boundaries of two hidden layer neurons, using a multi-layer neural network.

Question 1(c)

Consider Case (c), as shown in Figure 6.

The figure shows the instance space of a Perceptron with its decision boundary. Calculate the values of the weights of the Perceptron, using the position of the decision boundary. The cut-off points of the decision boundary on the axes are at $C_1 = (6, 0, 0)$, $C_2 = (0, 9, 0)$ and $C_3 = (0, 0, 8)$.

Positive instances are marked as P_i and negative instance as N_i:

$$
\begin{aligned}
P_1 &= (3, 5, 3) \\
P_2 &= (6, 6, 0) \\
P_3 &= (5, 0, 7) \\
P_4 &= (9, 6, 3) \\
N_1 &= (4, 0, 2) \\
N_2 &= (1, 3, 3) \\
N_3 &= (\text{-}4, 5, 4) \\
N_4 &= (2, 2, 0)
\end{aligned}
$$

Discussion on Question 1(c)

The decision boundary of a Perceptron with three inputs, x, y and z must satisfy the equation:

$$w_0 + w_1 x + w_2 y + w_3 z = 0$$

Three known points on the surface of the decision boundary are its intersection points with the axes, $C_1 = (6, 0, 0)$, $C_2 = (0, 9, 0)$ and $C_3 = (0, 0, 8)$. Substituting these into the decision boundary equation produces the simultaneous equations:

$$w_0 + 6w_1 = 0$$

$$w_0 + 9w_2 = 0$$

$$w_0 + 8w_3 = 0$$

Which is the same as:

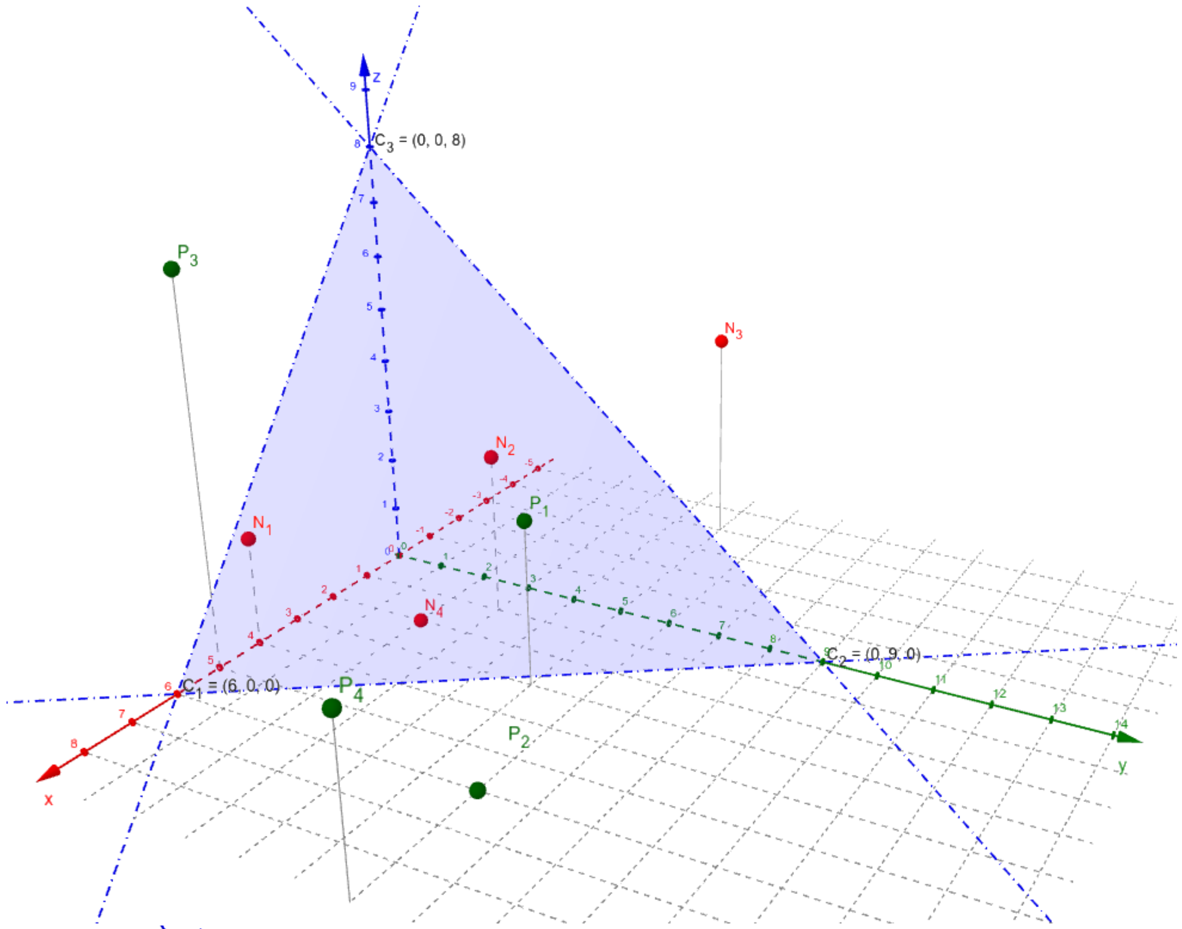
$$
\begin{aligned}
6w_1 &= \text{-}w_0 \\
9w_2 &= \text{-}w_0 \\
8w_3 &= \text{-}w_0
\end{aligned}
$$

Therefore:

$$6w_1 = 9w_2 = 8w_3$$

Taken apart, these become:

$$
\begin{aligned}
w_1 &= {}^{9}\!/_{6} \cdot w_2 = {}^{3}\!/_{2} \cdot w_2 \\
w_1 &= {}^{8}\!/_{6} \cdot w_3 = {}^{4}\!/_{3} \cdot w_3
\end{aligned}
$$



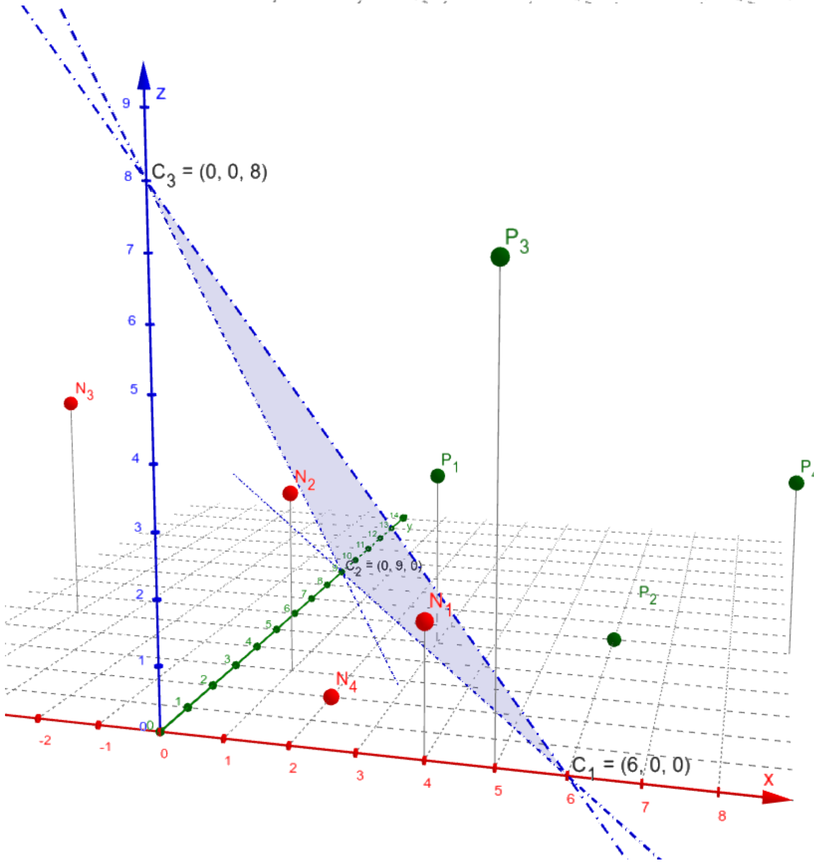


Figure 6: Case (c): instances and the decision boundary of a Perceptron (seen from different angles).

$$w_2 = {}^{8}\!/_{9} \cdot w_3$$

Any values of the weights, w_1, w_2 and w_3 that satisfy these equations simultaneously will produce a valid decision boundary. Arbitrarily choose $w_1 = -1$.

This gives:

$$-1 = {}^{3}\!/_{2} \cdot w_2$$

$$-1 = {}^{4}\!/_{3} \cdot w_3$$

and results in:

$$w_2 = -{}^{2}\!/_{3}$$

$$w_3 = -{}^{3}\!/_{4}$$

Solve w_0:

$$w_0 + 6 \cdot (-1) = 0$$

$$w_0 + 9 \cdot (-{}^{2}\!/_{3}) = 0$$

$$w_0 + 8 \cdot (-{}^{3}\!/_{4}) = 0$$

Therefore (all three equations agree):

$$w_0 = 6$$

To summarise:

$$\begin{array}{rcl} w_0 & = & 6 \\ w_1 & = & -1 \\ w_2 & = & -{}^{2}\!/_{3} \\ w_3 & = & -{}^{3}\!/_{4} \end{array}$$

Since the decision boundary has polarity the weight values should be tested to determine if the Perceptron with these weight values classifies the data correctly. Pick the positive instances $P_1 = (3, 5, 3)$ for the first test:

$$\begin{array}{rl} & w_0 + w_1 x + w_2 y + w_3 z \\ = & 6 + (-1)(3) + (-{}^{2}\!/_{3})(5) + (-{}^{3}\!/_{4})(3) \\ = & 6 - 3 - {}^{10}\!/_{3} - {}^{9}\!/_{4} \\ = & {}^{72}\!/_{12} - {}^{36}\!/_{12} - {}^{40}\!/_{12} - {}^{27}\!/_{12} \\ = & -{}^{31}\!/_{12} \\ < & 0 \end{array}$$

This means that the current weights incorrectly classifies the positive instance P_1 as negative. The weight values need to be inverted to correct the polarity of the decision boundary to produce correct classification of this particular instance.

$$\begin{aligned} w_0 &= -6 \\ w_1 &= 1 \\ w_2 &= {}^2\!/_3 \\ w_3 &= {}^3\!/_4 \end{aligned}$$

Testing P_1 now gives:

$$\begin{aligned} & w_0 + w_1x + w_2y + w_3z \\ = \; & -6 + (1)(3) + ({}^2\!/_3)(5) + ({}^3\!/_4)(3) \\ = \; & -6 + 3 + {}^{10}\!/_3 + {}^9\!/_4 \\ = \; & {}^{-72}\!/_{12} + {}^{36}\!/_{12} + {}^{40}\!/_{12} + {}^{27}\!/_{12} \\ = \; & {}^{31}\!/_{12} \\ > \; & 0 \end{aligned}$$

The positive instance is therefore correctly classified by the Perceptron. This calculation was done with a single instance, but should be repeated with all the instances. If all of them are correctly classified the decision boundary is correct. If any one of the instances are still incorrectly classified, it may mean that a single neuron network cannot classify the data, or that there was an error in your calculations.

Question 1(d)

Draw a diagram to illustrate the *complete* structure of the Perceptron in Question 1(c) (the instance space shown in Figure 6). Show on the diagram all the inputs, outputs, variables, functions, as well as the weight values you calculated in Question 1(c). Use the correct terminology and mathematical notation.

Discussion on Question 1(d)

Figure 7 shows the diagram of a Perceptron with 3 inputs and the weights as in Question 1(c). The

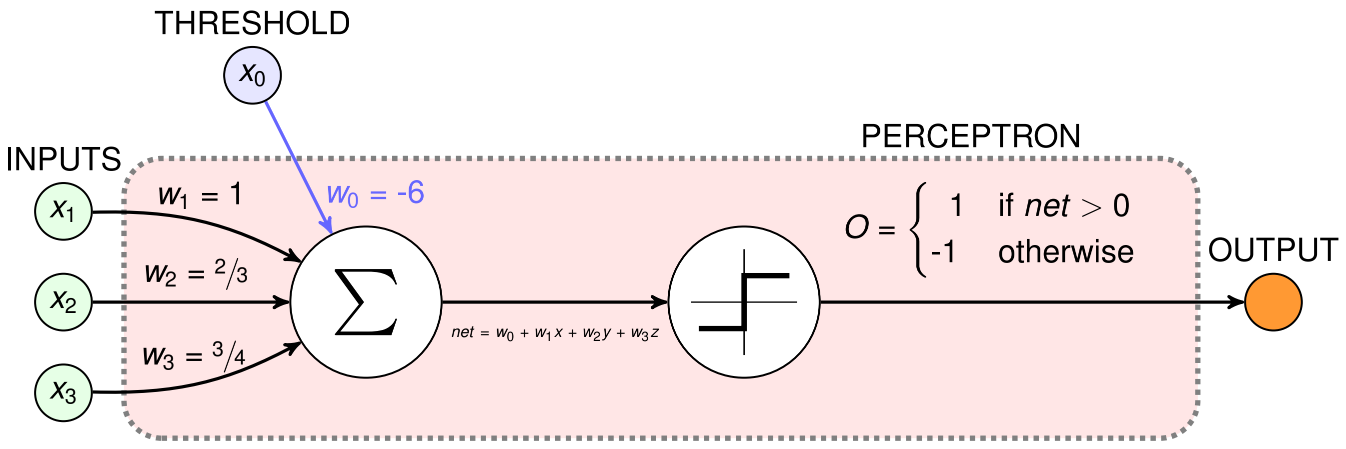

THRESHOLD
x_0
INPUTS
x_1
x_2
x_3
$w_1 = 1$
$w_2 = {}^2/_3$
$w_3 = {}^3/_4$
$w_0 = -6$
PERCEPTRON
$net = w_0 + w_1x + w_2y + w_3z$
$O = \begin{cases} 1 & \text{if } net > 0 \\ -1 & \text{otherwise} \end{cases}$
OUTPUT

Figure 7: A Perceptron with 3 inputs.

dotted box indicates the ‘boundaries’ of the Perceptron. The other nodes in the diagram are the input and output nodes and are external the Perceptron. The input and output nodes may themselves be Perceptron nodes, sending their output to, or receiving input from this Perceptron. Such an arrangement will produce a network of Perceptrons, called a Neural Network. The threshold (or bias) node x_0 always has a value of 1. A non-zero threshold (bias) weight w_0 moves the decision boundary away from the origin in instance space. If there was no threshold node, or if the bias or threshold weight was zero, the decision boundary would always touch the origin, severely limiting the classification ability of the Perceptron.

Consider again the Perceptron in Figure 1 and the effect of a decision boundary that always has to touch the origin. Even though the instance space is linearly separable, a Perceptron with a zero bias weight will not be able to classify the instances correctly.

Mark out of 100.
40 or less for clear indication that student does not understand the topic or evidence of plagiarism
50 for a fair understanding
60-70 for understanding and clear well defined examples
80+ for exceptional detail

Question 2

Design neural networks, using one or more Perceptron neurons, for the Boolean functions f_1, f_2, and f_3, as detailed in Questions 2(a) to (c).

Let your Perceptron use the threshold activation function (shown in Figure 8):

$$O(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

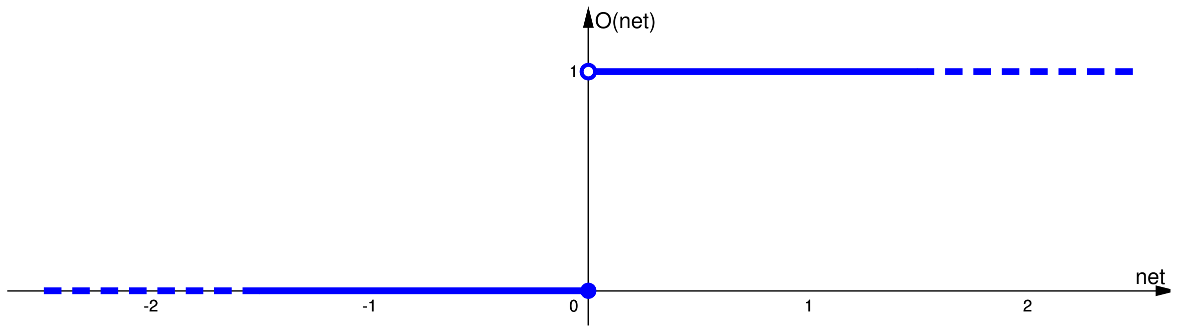


Figure 8: Threshold activation function.

For each sub-question, draw diagrams to show the decision boundaries that your network uses, and the structure of the neural network showing the weights that will correctly classify the function. Show the arguments for your choices, all your assumptions, definitions, and calculations. Prove that the weights you use will correctly classify the function.

Note that this question does **not** ask you to train a neural network using one of the neural network algorithms. The purpose here is similar to the question on the relationship between decision trees and Boolean expressions, in the previous assignment. It is possible to directly map a specific Boolean expression to a network of Perceptrons. Rojas has a very good explanation of the concepts needed in Chapter 2 (and some aspects on XOR in Chapter 6).

Question 2(a)

$f_1(x, y)$, as shown in Table 1

x	y	$f_1(x, y)$
-2	-2	0
-2	+2	1
+2	-2	0
+2	+2	0

Table 1: Function $f_1(x, y)$.

Discussion on Question 2(a)

A single Perceptron can represent any one of the primitive Boolean functions AND, OR, NAND and NOR. A single Perceptron can also represent the negation of any of these Boolean primitives by changing the signs of the weights associated with that particular input.

Plot the points of Table 1 in the input space of f_1. Figure 9 shows f_1 with a valid decision boundary. The decision boundary for a two-input Perceptron is defined by the function, $w_0 + w_1 x + w_2 y = 0$.

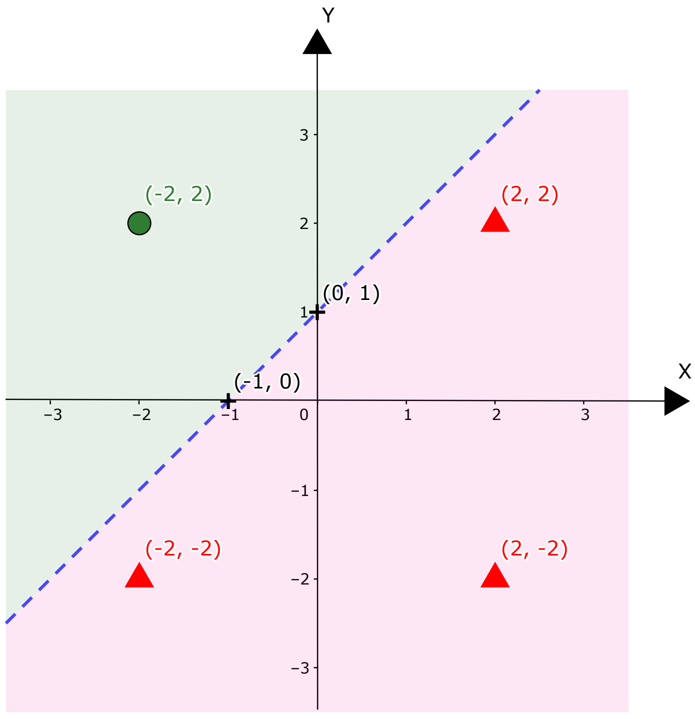


Figure 9: $f_1(x, y)$ with a valid decision boundary.

The task is now to find weight values that will match the decision boundary in Figure 9. Solving for w_i, using the decision boundary as in Figure 9, and testing the weight polarity (changing signs if need be) gives:

$$\begin{aligned} w_0 &= \text{-1} \\ w_1 &= \text{-1} \\ w_2 &= 1 \end{aligned}$$

Calculate the Perceptron output for each of the possible inputs, to get the results in Table 2, which show that a Perceptron network with a single neurons, using these weights correctly classify f_1.

The structure of the Perceptron using these weights are shown in Figure 10.

x	y	f_1	net = -1 $-$ $x + y$	O(net)
-2	-2	-1	$-1 + 2 - 2 = -1$	0
-2	+2	+1	$-1 + 2 + 2 = +4$	1
+2	-2	-1	$-1 - 2 - 2 = -5$	0
+2	+2	-1	$-1 - 2 + 2 = -1$	0

Table 2: Calculated outputs for the function $f_1(x, y)$ using the derived weights.

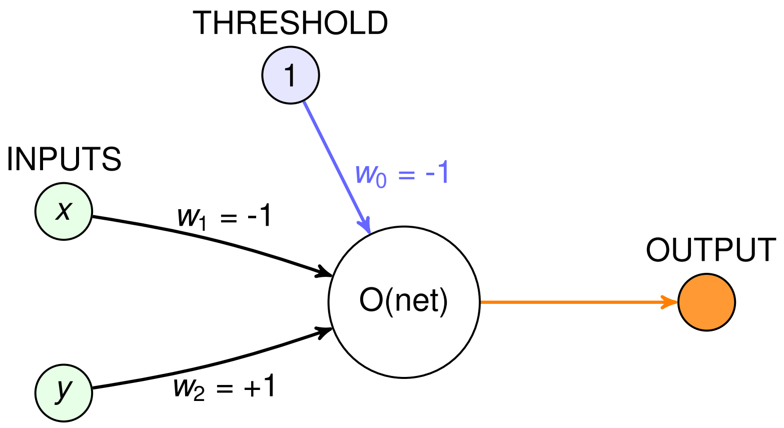


Figure 10: A Perceptron with two inputs for $f_1(x, y)$.

Question 2(b)

$f_2(x, y)$, as shown in Table 3

x	y	$f_2(x, y)$
-2	-2	0
-2	+2	1
+2	-2	1
+2	+2	0

Table 3: Function $f_2(x, y)$.

Discussion on Question 2(b)

A single Perceptron cannot correctly classify f_2, since it is not linearly separable. In other words there is not a single decision boundary in the input space of the Perceptron that will separate the input examples correctly. This can be better seen in Figure 11.

In order to use Perceptrons to perform the classification required for f_2, a network of Perceptrons in two layers are needed, where the first layer performs sub-classifications matching each of the blue lines in Figure 11. The output of this Perceptron is then fed to another Perceptron in a second layer. The first layer of Perceptrons therefore re-maps the input points into linearly separable values. Effectively $f_2(x, y)$ is re-written as a Boolean function of two sub-functions,

$$f_2(x, y) = g_3(x, y) = g_1(x, y) \wedge g_2(x, y)$$

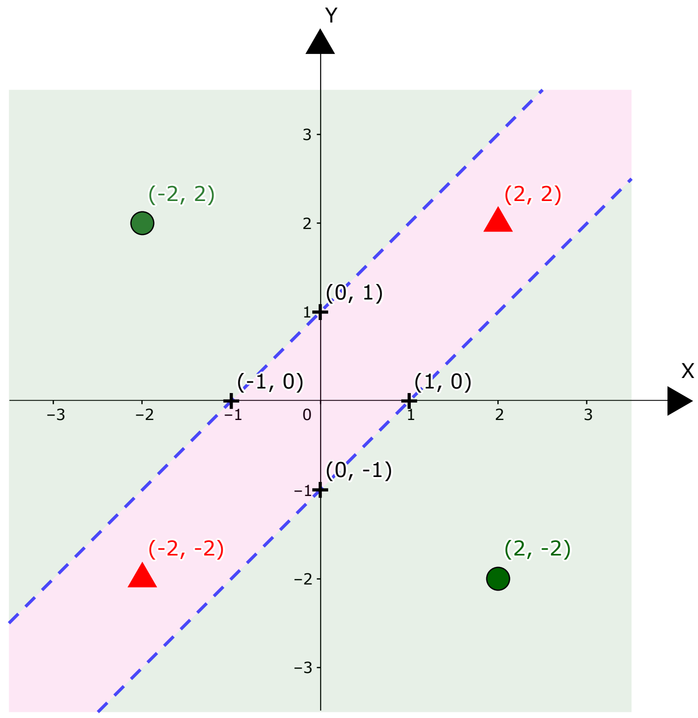


Figure 11: $f_2(x, y)$ needs two decision boundaries to classify correctly.

x	y	f_2	$g_1(x, y)$	$g_2(x, y)$	$g_3(g_1, g_2)$
-2	-2	-1	-1	-1	0
-2	+2	+1	+1	-1	1
+2	-2	+1	-1	+1	1
+2	+2	-1	-1	-1	0

Table 4: Re-map f_2 as sub-functions $g_1(x, y)$, $g_2(x, y)$ and $g_3(g_1, g_2)$.

as shown in Table 4.

Sub-functions g_1, g_2, and g_3 (as shown in Figure 12) are linearly separable. They can therefore be implemented as Perceptrons using a threshold function.

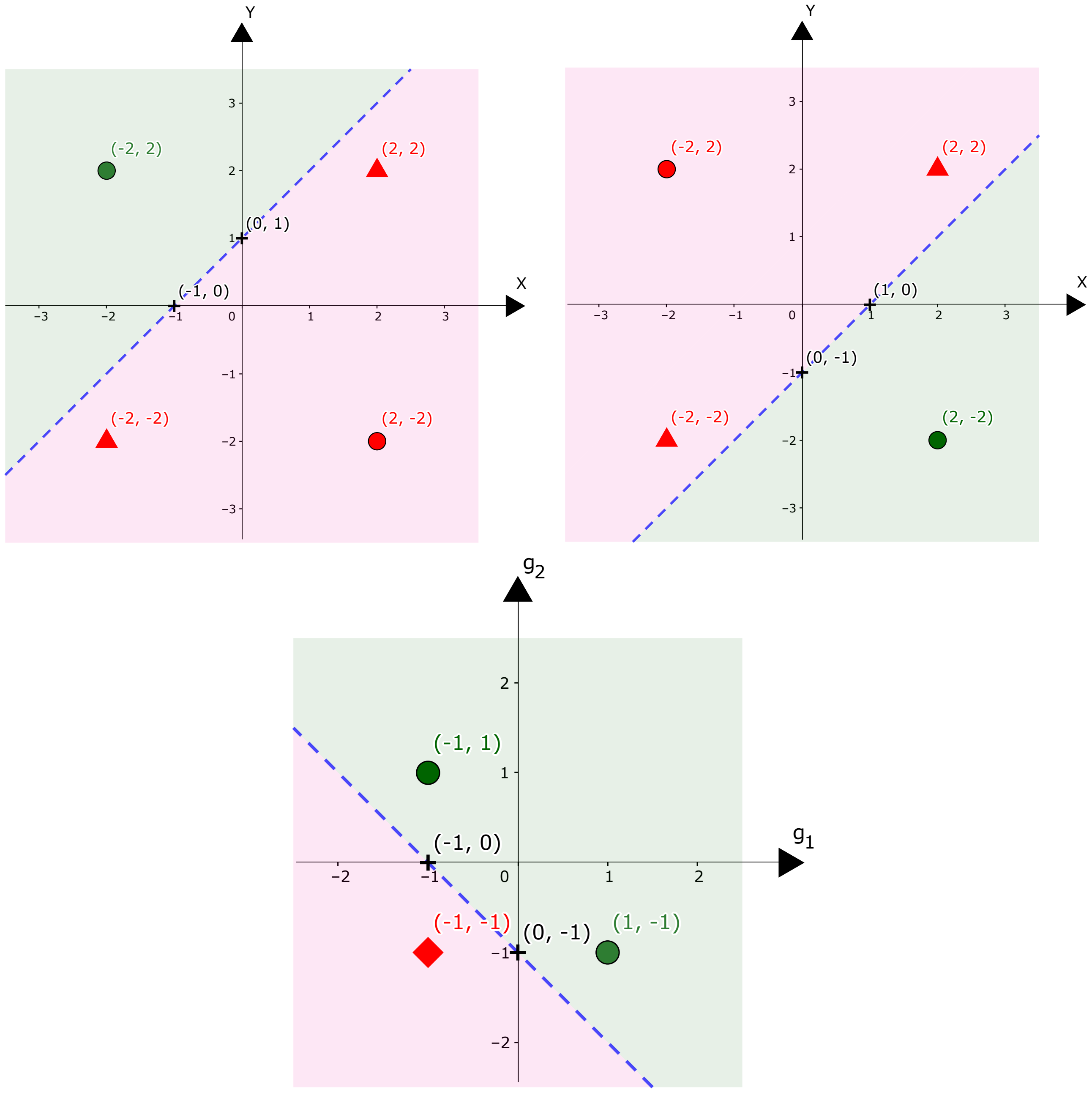


Figure 12: Functions $g_1(x, y)$, $g_2(x, y)$, and $g_3(g_1, g_2)$ with decision boundaries that will classify each of the sub-problems correctly.

The weights for these Perceptrons are calculated, polarity corrected, and tested for correctness, in the same way as before, and results in:

$$\begin{aligned}
w_{g_1 0} &= \text{-1}\\
w_{g_1 x} &= \text{-1}\\
w_{g_1 y} &= 1\\
w_{g_2 0} &= \text{-1}\\
w_{g_2 x} &= 1\\
w_{g_2 y} &= \text{-1}\\
w_{g_3 0} &= 1\\
w_{g_3 g_1} &= 1\\
w_{g_3 g_2} &= 1
\end{aligned}$$

The Perceptron network for $f_2(x, y)$ is shown in Figure 13. Note the three neurons calculating the three sub-functions g_1, g_2, and g_3.

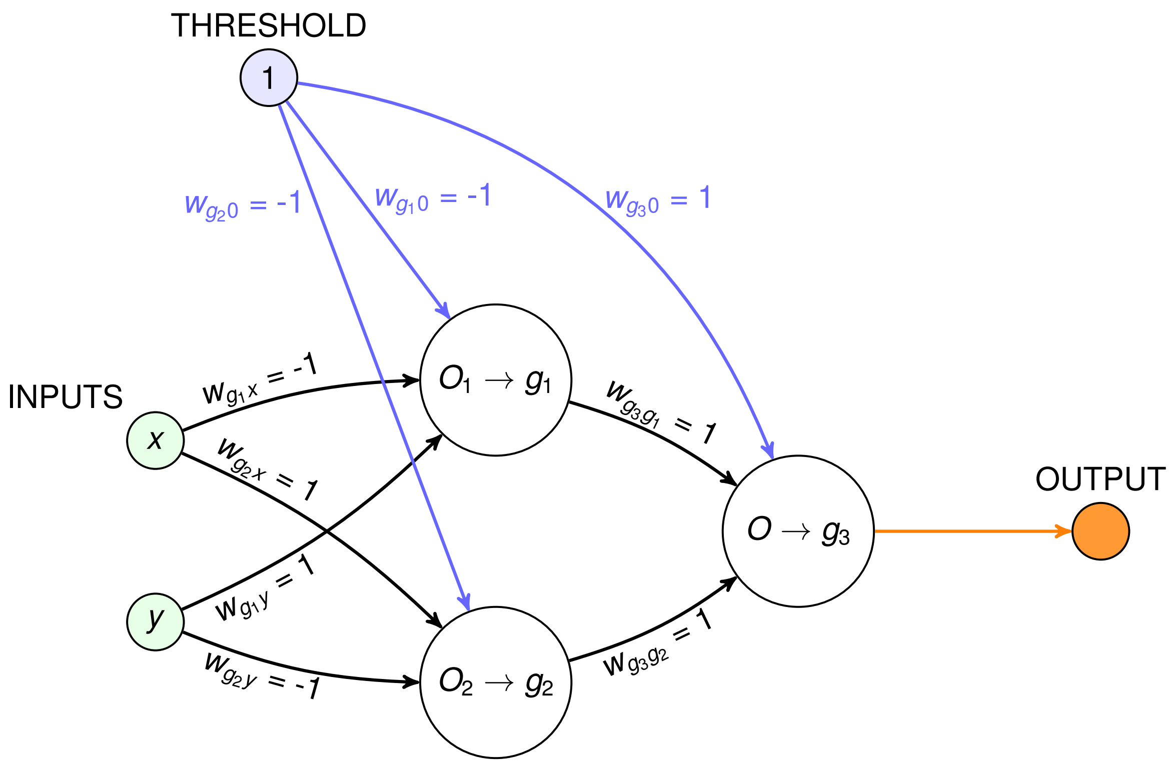


Figure 13: The Perceptron network for $f_2(x, y)$.

Question 2(c)

$f_3(x, y)$, as shown in Table 5

x	y	$f_3(x, y)$
-1	-1	1
-1	+1	0
+1	-1	1
+1	+1	1

Table 5: Function $f_3(x, y)$.

Discussion on Question 2(c)

A quick comparison of Tables 1 and 5 show that f_3 is a scaled inverse of f_3. By plotting the points in (x, y)-space (shown in Figure 14) it is obvious that the decision boundary used for f_1 also separates the positive and negative instances.

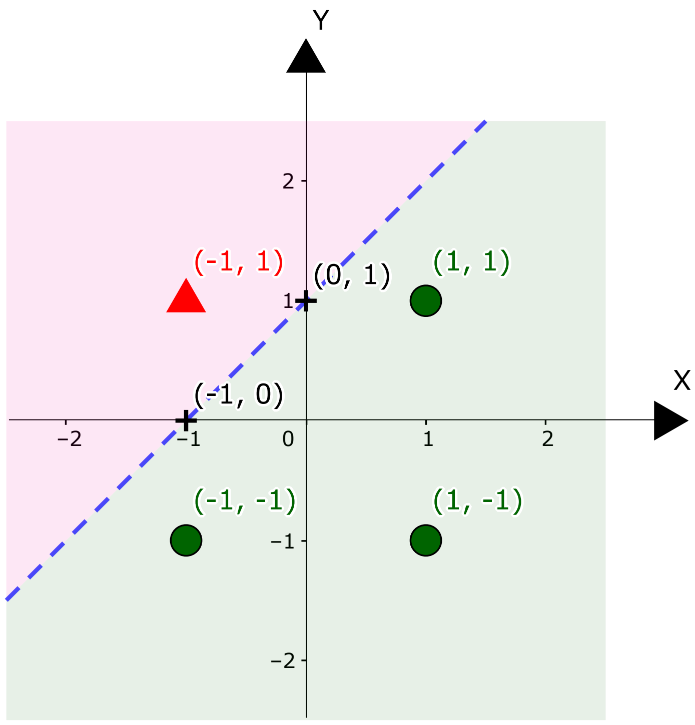


Figure 14: $f_3(x, y)$ needs two decision boundaries to classify correctly.

The difference is that the polarity of the decision boundary is reversed. This means the weights used for the Perceptron for f_1 can be inverted and used for f_3.

$$\begin{aligned} w_0 &= 1 \\ w_1 &= 1 \\ w_2 &= -1 \end{aligned}$$

x	y	f_1	net = $1 + x - y$	O(net)
-1	-1	+1	1 + -1 + 1 = 1	1
-1	+1	-1	1 + -1 − 1 = -1	0
+1	-1	+1	1 + 1 + 1 = 3	1
+1	+1	+1	1 + 1 − 1 = 1	1

Table 6: Calculated outputs for the function $f_3(x, y)$ using the inverted weights from f_1.

Test this observation by calculating the network outputs for the inverted weights. The results in Table6 show that this observation was correct.

The resulting Perceptron is shown in Figure 15.

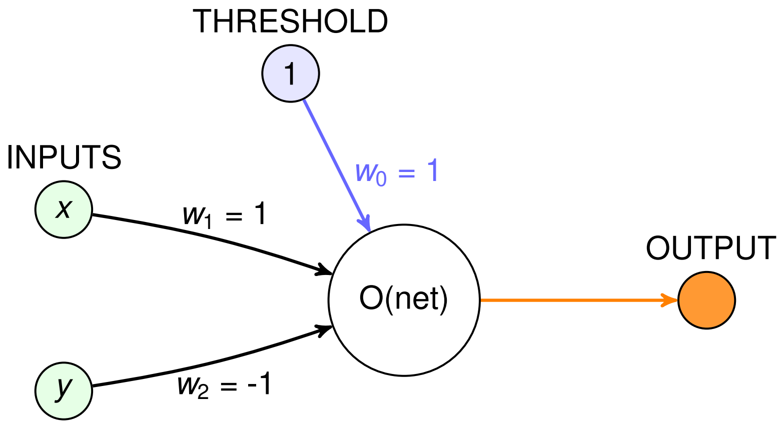


Figure 15: A Perceptron with two inputs for $f_3(x, y)$.

Mark out of 100.
40 or less for clear indication that student does not understand the topic or evidence of plagiarism
50 for a fair understanding
60-70 for understanding and clear well defined examples
80+ for exceptional detail

Question 3

Question 3(a)

Find the original 1986 article by Rumelhart, Hinton, and Williams that introduced the BACKPROPAGATION algorithm, and read it. Give the URL where you found it.

Discussion on Question 3(a)

The original article (letter) was published as an article in *Nature*:

Authored By:	D. E. Rumelhart, G. E. Hinton and R. J. Williams
Paper Title:	Learning Representations By Back-Propagating Errors
In:	Nature, Vol. 323
Date:	1986
Pages:	533-536

A more comprehensive version was later published in a book.

Some URLs where it can be found:

```
https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf
https://www.academia.edu/2520405/Learning_representations_by_back-propagati
http://www.cs.utoronto.ca/~hinton/absps/naturebp.pdf
```

Question 3(b)

Go to the `brilliant.org` website on Artificial Neural Networks and Feedforward Networks (links below). Study the material presented there. Then go to the `brilliant.org` website on the BACKPROPAGATION algorithm (link below) on how to train a feedforward neural network. Study the material presented there.

```
https://brilliant.org/wiki/artificial-neural-network/
https://brilliant.org/wiki/feedforward-neural-networks/
https://brilliant.org/wiki/backpropagation/
```

Study the Python code at the bottom of the `brilliant.org` site on BACKPROPAGATION, to see one way of implementing the training phase of the BACKPROPAGATION algorithm, in Python. This particular network consists of three inputs units, one layer of three hidden units, and a single output units, that learns function f_4 in Table 7.

Copy and execute the code in a Python 3 interpreter or a Jupyter notebook. Copy the output of your program into your answer. Explain what the output says about the training of the network. Explain the output values in terms of the training data.

Discussion on Question 3(b)

The code below is modified from the `brilliant.org` website:

x_1	x_2	x_3	$f_4(x_1, x_2, x_3)$
0	0	1	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 7: $f_4(x_1, x_2, x_3)$

```
#
# Modified from code at https://brilliant.org/wiki/backpropagation/
#

import numpy as np

# define the sigmoid function
def sigmoid(x, derivative=False):

    if (derivative == True):
        return x * (1 - x)
    else:
        return 1 / (1 + np.exp(-x))

# choose a random seed for reproducible results
np.random.seed(1)

# learning rate
alpha = .1

# number of nodes in the hidden layer
num_hidden = 3

# inputs
X = np.array([
    [0, 0, 1],
    [0, 1, 1],
    [1, 0, 0],
    [1, 1, 0],
    [1, 0, 1],
    [1, 1, 1],
])

# outputs
# x.T is the transpose of x, making this a column vector
y = np.array([[0, 1, 0, 1, 1, 0]]).T

# initialize weights randomly with mean 0 and range [-1, 1]
# the +1 in the 1st dimension of the weight matrices is for the bias weight
hidden_weights = 2*np.random.random((X.shape[1] + 1, num_hidden)) - 1
output_weights = 2*np.random.random((num_hidden + 1, y.shape[1])) - 1

# number of iterations of gradient descent
num_iterations = 10000

# for each iteration of gradient descent
for i in range(num_iterations):

    # forward phase
    # np.hstack((np.ones(...), X) adds a fixed input of 1 for the bias weight
    input_layer_outputs = np.hstack((np.ones((X.shape[0], 1)), X))
    hidden_layer_outputs = np.hstack((np.ones((X.shape[0], 1)),
      sigmoid(np.dot(input_layer_outputs,
      hidden_weights))))
    output_layer_outputs = np.dot(hidden_layer_outputs,
       output_weights)

    # backward phase
    # output layer error term
    output_error = output_layer_outputs - y
    # hidden layer error term
    # [:, 1:] removes the bias term from the backpropagation
    hidden_error = hidden_layer_outputs[:, 1:] *
     (1 - hidden_layer_outputs[:, 1:]) *
     np.dot(output_error, output_weights.T[:, 1:])

    # partial derivatives
    hidden_pd = input_layer_outputs[:, :, np.newaxis] *
     hidden_error[: , np.newaxis, :]
    output_pd = hidden_layer_outputs[:, :, np.newaxis] *
     output_error[:, np.newaxis, :]

    # average for total gradients
    total_hidden_gradient = np.average(hidden_pd, axis=0)
    total_output_gradient = np.average(output_pd, axis=0)

    # update weights
    hidden_weights += - alpha * total_hidden_gradient
    output_weights += - alpha * total_output_gradient

# print the final outputs of the neural network on the inputs X
print("Output After Training: \n{}".format(output_layer_outputs))
```

The initial output of the code gives:

```
Output After Training:
[[2.11135662e-04]
 [9.99525588e-01]
 [1.66889680e-04]
 [9.99576185e-01]
 [9.99362960e-01]
 [1.30185107e-03]]
```

These six values are the output values of the network that correspond to the six input patterns, as in Table 8.

x_1	x_2	x_3	$f_4(x_1, x_2, x_3)$	network output	rounded
0	0	1	0	0.000111357	0.00
0	1	1	1	0.999525588	1.00
1	0	0	0	0.000166890	0.00
1	0	1	1	0.999576185	1.00
1	1	0	1	0.999362960	1.00
1	1	1	0	0.001301851	0.00

Table 8: $f_4(x_1, x_2, x_3)$ and results from the original Python code

The table also shows what happens when the network outputs are rounded (to the nearest hundredth in this case). The outputs effectively become 0 and 1 and map the values of f_4 exactly. This shows that the network learned to map the input data correctly onto the output data. The reason we can do this has to do with the asymptotic nature of the sigmoid activation function. As the absolute input to the sigmoid function increases the closer its output gets to either 0 or 1. Refer back to your study of the `brilliant.org` website on Artificial Neural Networks.

Question 3(c)

Use the data from function f_4, and perform the following three experiments, by modifying the Python code as indicated:

(1) Run the code as-is, with the three (3) hidden neurons.

(2) Change the code to modify the network structure to use only two (2) hidden neurons instead of three (3).

(3) Change the code again to modify the network structure to use a single (1) hidden neuron.

Change or remove the random seed value to allow random initialisation of weight values. Play with the learning rate `alpha`, and the number of training iteration, `num_iterations` to achieve the best network results. Modify the code to show the hidden layer weights. Show listings of your modifications to the code. Show the outputs from the code. Discuss the results of these three experiments, in terms of:

(i) Whether the training was successful or not.

(ii) What the output results mean.

(iii) Compare the results of the three experiments, and explain why some were successful and some not.

(iv) Use the weight values from the program output and draw a diagram of the input space and the hidden layer decision boundaries to support your argument.

Discussion on Question 3(c)

Start by drawing the input space, as in Figure 16. This shows that it should be possible to define two decision boundaries that will separate the input space into three distinct areas, with each area containing instances of only one class. This in turn will make it possible for the output neuron to correctly classify any instance in each area into the correct class.

Does this observation correspond with the results of the experiments?

Figure 16 shows the input space for the three experiments.

Question 3(c)(1)

A network with three hidden neurons should be successful. Experiment (1) is simply a repeat of Question 3(b, where the training was successful, and confirms the hypothesis.

Question 3(c)(2)

The second experiment changes the following code snippet to use 2 hidden layers instead of 3:

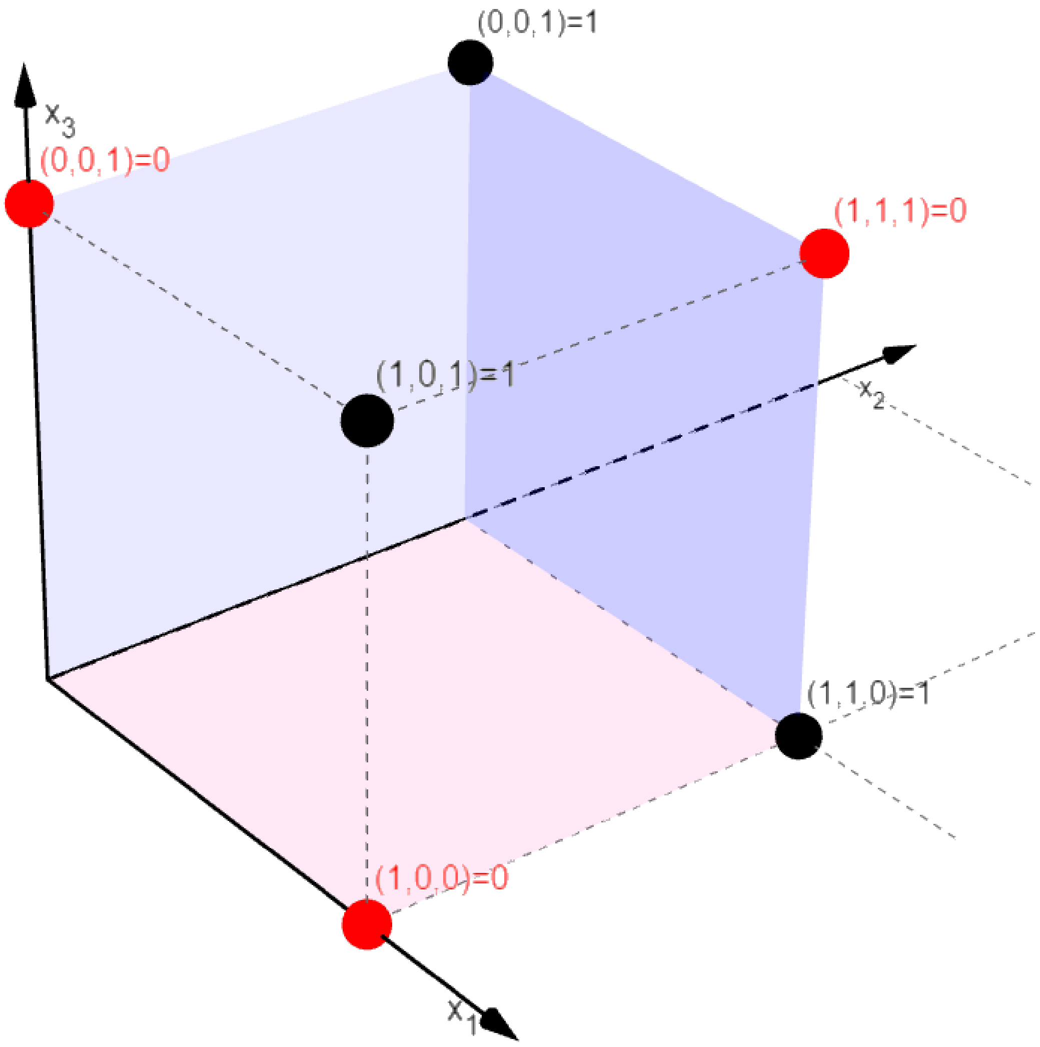


Figure 16: The input space for the three experiments.

```
# number of nodes in the hidden layer
num_hidden = 2
```

A network with two hidden neurons should struggle more to be trained successfully, as the option space for correct weight values are much smaller, and therefore more difficult to find. This should show in how many attempts it takes to train the network successfully. One set of weights that are successful for a network with two hidden neurons, are:

$$
\begin{aligned}
w_{10} &= -0.66 \\
w_{11} &= 5.25 \\
w_{12} &= -7.04 \\
w_{13} &= 5.10 \\
w_{20} &= 8.23 \\
w_{21} &= -5.44 \\
w_{22} &= 6.70 \\
w_{23} &= -5.39
\end{aligned}
$$

Using the familiar equations for a plane in 3D-space,

$$
\begin{aligned}
w_{10} + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 &= 0 \\
w_{20} + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 &= 0
\end{aligned}
$$

we can draw the decision boundaries in the input space for experiment (2), as in Figure 17. This shows that the decision boundaries for the two hidden layer neurons were able to re-map the input space into three distinct regions, each with a distinct set of output values from the hidden layer neurons, as shown in Table 9. These unique sets of hidden output values match the expected class,

x_1	x_2	x_3	class	h_1	h_2	rounded	region
0	0	1	0	0.98840949	0.94485492	11	2
0	1	1	1	0.06978661	0.99992827	01	3
1	0	0	0	0.98996648	0.94180994	11	2
1	0	1	1	0.99993854	0.06870401	10	1
1	1	0	1	0.07986793	0.99992406	01	3
1	1	1	0	0.93469969	0.98361150	11	2

Table 9: The hidden layer outputs for the 2-hidden-layer network.

enabling the output layer to correctly classify the input data.

Question 3(c)(3)

The third experiment changes the following code snippet to use 1 hidden layer instead of 3:

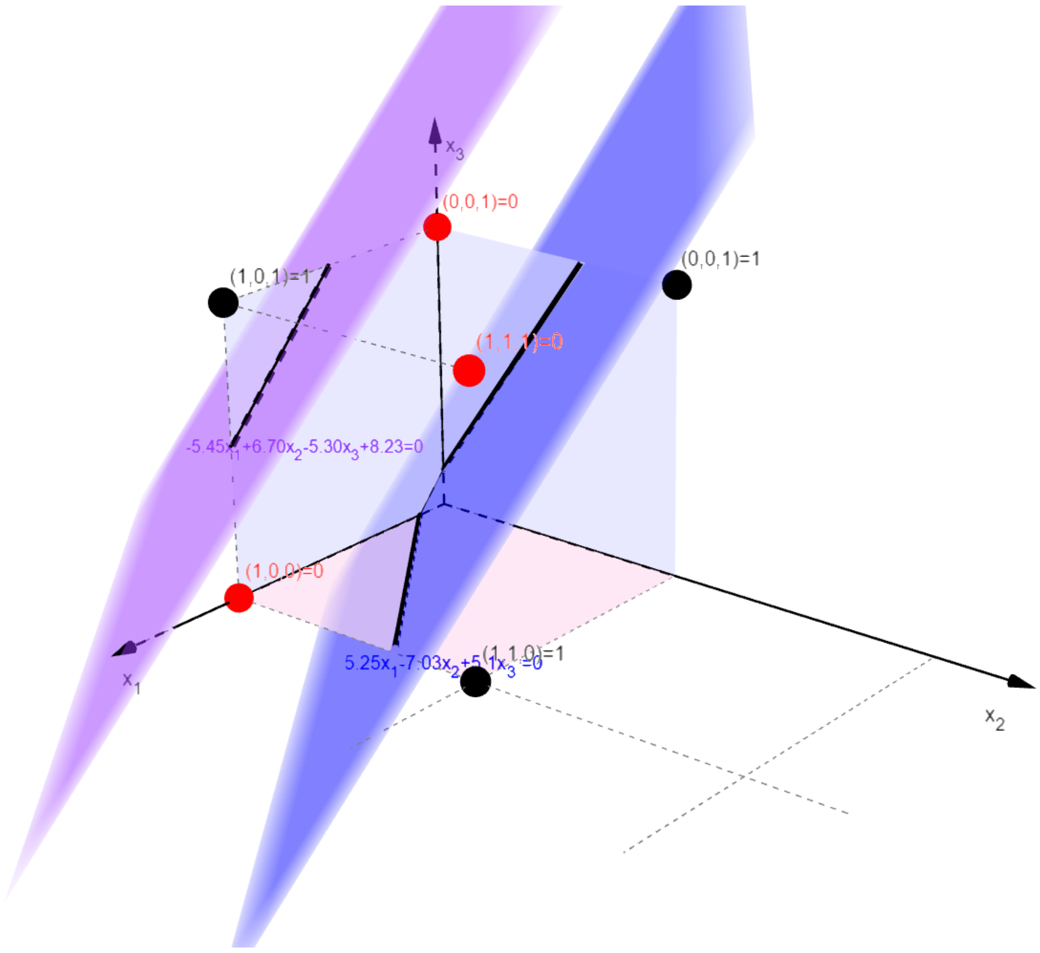


Figure 17: The input space experiment (2) with two decision boundaries.

```
# number of nodes in the hidden layer
num_hidden = 1
```

Given what we know about the input space, we do not expect this network to be able to be successfully trained. After some experimentation with the training rate and number of training cycles, one set of weights for the hidden neuron is:

$$\begin{aligned} w_{10} &= -10.05 \\ w_{11} &= 8.63 \\ w_{12} &= 9.89 \\ w_{13} &= 8.68 \end{aligned}$$

Again, using the familiar equations for a plane in 3D-space,

$$w_{10} + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 = 0$$

we can draw the decision boundary in the input space for experiment (3), as in Figure 18. This shows that the single decision boundary for the one hidden layer network was unable to re-map the input space into regions that each produce neuron outputs of one class only. The input values $(1, 1, 1)$ is incorrectly placed in a region of class 1. The output values of the hidden neuron is shown in Table 10.

x_1	x_2	x_3	class	h_1	rounded	region
0	0	1	0	0.20218074	0	1
0	1	1	1	0.99980072	1	2
1	0	0	0	0.19405193	0	1
1	0	1	1	0.99929478	1	2
1	1	0	1	0.99979026	1	2
1	1	1	0	0.99999996	1	2

Table 10: The hidden layer outputs for the 1-hidden-layer network.

A network with a single hidden neuron is unable to learn the required classification, because it cannot separate the input space sufficiently.

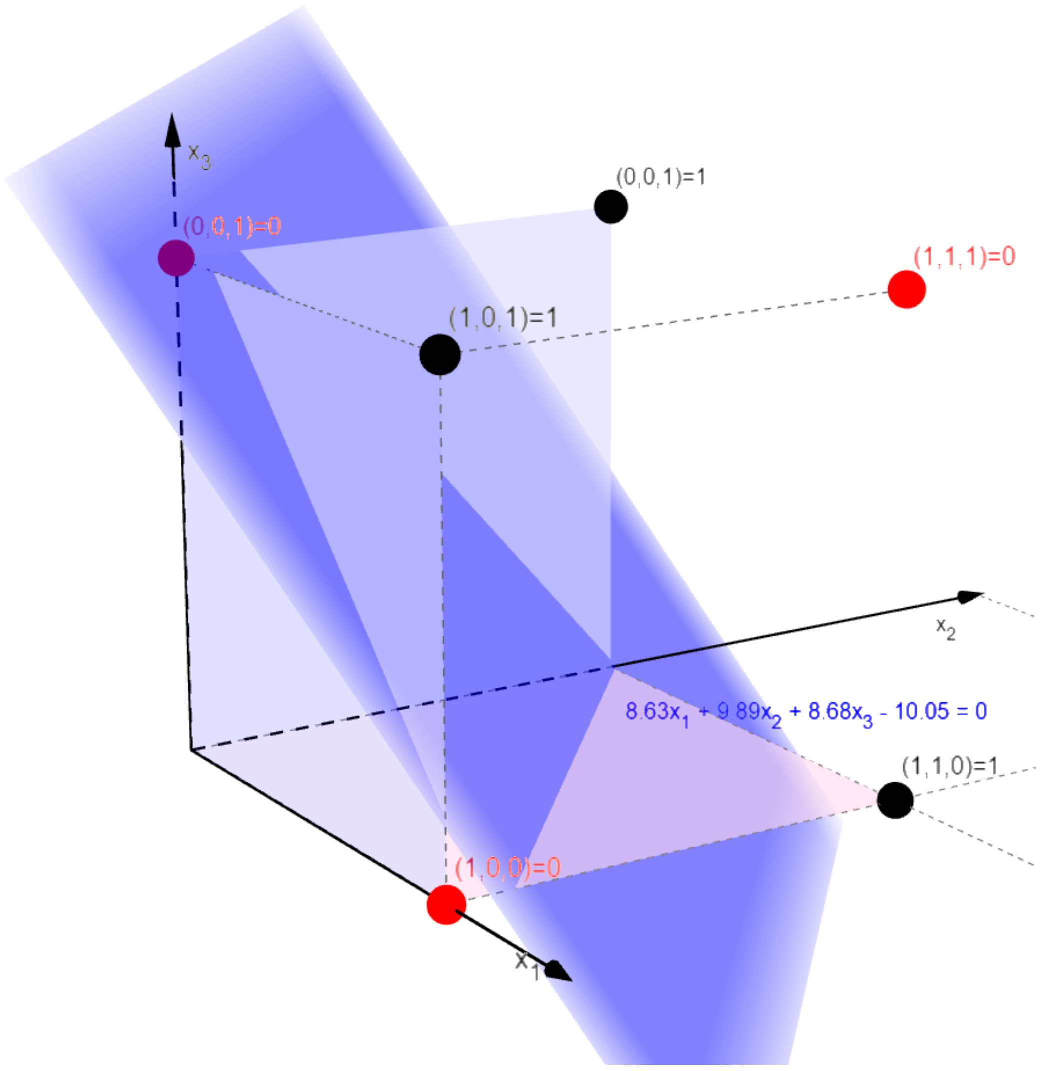


Figure 18: The input space for experiment (3) with the single decision boundary.

Question 3(d)

Discuss the different activation functions that can be used in feedforward neural networks, how they work, why they allow training to happen in various algorithms, and their pros and cons.

Discussion on Question 3(d)

Here is an excellent discussion on the topic on the nature of activation functions.

`https://missinglink.ai/guides/neural-network-concepts/7-types-neural-networ`

Another good website:

`https://missinglink.ai/guides/neural-network-concepts/7-types-neural-networ`

The simplest function to use in a neural network is the linear function:

$$f_{linear}(\text{net}) = \text{net}$$

A multi-layer neural network that consist of only linear activation functions are used, the complete network could be re-mapped to a single-layer network with linear activation functions. However, of the hidden layer uses non-linear activation functions (such as the sigmoid function) the network could be used to perform non-linear regression.

The most common activation function is the well-known sigmoid function:

$$f_{\text{sigmoid}}(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$$

This function is non-linear, has a positive derivative, and is asymptotic. Neural networks with activation functions with these properties could be trained using the BACKPROPAGATION algorithm, and learn to closely approximate any function. However, the sigmoid function can have negative output values, and can therefore map to output values close to zero, which in turn create very small weight update values, therefore causing the training to proceed extremely slowly.

This problem can be resolved by using the *tanh* activation function:

$$f_{\text{tanh}}(\text{net}) = 1 - \tanh^2(\text{net})$$

It has the same properties as the sigmoid activation function, but always has positive output values, which avoids the trap of very small weight update values.

Question 3(e)

Find and read the following two articles, by Cybenko and Hornik:

Cybenko, G. (1989) “Approximations by superpositions of sigmoidal functions”, *Mathematics of Control, Signals, and Systems*, 2(4), 303-314. doi:10.1007/BF02551274

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.441.7873&rep=rep1&type=pdf`

Kurt Hornik (1991) “Approximation capabilities of multilayer feedforward networks”, *Neural Networks*, 4(2), 251-257. doi:10.1016/8093-6080(91)90009-T

`http://cognitivemedium.com/magic_paper/assets/Hornik.pdf`

Discuss what kind of functions can be represented by feed-forward neural networks, and what sort of network structures are needed.

Discussion on Question 3(e)

The short answer is that a feedforward neural network that uses sigmoid activation functions can closely approximate any continuous function. The above articles go into great depth to provide the proof of this.

A detailed visual discussion on why this is the case is in Chapter 4 of Michael Nielsen’s *online* book on Neural Networks and Deep Learning (`http://neuralnetworksanddeeplearning.com/chap4.html`). The Wikipedia page on the Universal Approximation Theorem is also informative (`https://en.wikipedia.org/wiki/Universal_approximation_theorem`).

Question 3(f)

Discuss the problem of local minima in neural network training, and techniques to ensure optimal convergence.

Discussion on Question 3(f)

A particular classification problem that a BACKPROPAGATION neural network is trained on, may have (usually) an infinite number of solutions, many of which are not the optimal solution, and some of these could be really bad solutions. BACKPROPAGATION is essentially of finding a solution using gradient descent, which means that it could find a local minimum at some point in the training process and that the subsequent weight changes are not large enough to escape the local minimum. This local minimum may not the optimal solution and may even be a particularly bad solution.

There are a number of techniques to try and avoid local minima, or to escape those that are encountered. Stochastic learning vs. batch learning is one aspect that is often looked at (read up on these modes of BACKPROPAGATION learning), where stochastic learning reduces the chance of getting stuck in a local minimum. One solution is to add a momentum parameter, which uses the previous weight update to keep the direction of change fairly stable. Other approaches uses techniques to add noise to weight changes to ‘shake’ the solution out of a local minimum, which is referred to as simulated annealing.

A related problem is that of over-fitting, where the BACKPROPAGATION network tends to fit the solution function to the training very exactly, with the result that a new instance of the same class can be classified incorrectly. The network fails to generalise well. There are two main approached to countering this. Cross-validation, by separating the data into a training set and a smaller test set. is one. The total network error is calculated using the test data set, and the training stopped when this error is sufficiently low, but starts increasing again, while the total error using the training set is still decreasing. The second technique is using weight decay, by slowly changing the rate at which weights are updated, as training progresses. This will keep weight values small, and also fits well with the annealing technique to avoid local minima.

Question 3(g)

Read the Wikipedia page on Autoencoders:

`https://en.wikipedia.org/wiki/Autoencoder`

Feedforward neural networks typically consist of 3 layers, with the hidden (middle) layer effectively being able to learn hidden representations in the data. In this context, search for the 8×3×8 neural network structure, also called the identity function (give the URL for your source). Use the 8×3×8 neural network structure to discuss how feed-forward networks learn hidden representations in the context of autoencoders.

Write your own code (in any programming language) that will replicate the 8×3×8 experiment. Report on your experience with this.

Discussion on Question 3(g)

An interesting aspect of feedforward neural network, using the BACKPROPAGATION training algorithm, is that only the output layer weights are directly constrained by the training data. The hidden layer can adapt its weights in any way that find the solution, and often the hidden layer outputs reveal that the network learned an intermediate set of representations, that show properties of the input space that may not be immediately apparent.

The 8×3×8 encoder, or identity function mapping, is a good example to illustrate this.

Consider a neural network consisting of 8 inputs, 3 hidden layer neurons, and 8 output neurons that is trained to learn the function in Table 11.

inputs		outputs
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Table 11: The identity function.

Since there are only 3 hidden layer neurons the network must re-map the 8 inputs to a set of 3 hidden layer neurons, which are then mapped by the output layer onto the desired outputs. This means that the hidden layer must learn a new encoding of the the inputs. A Python program similar to the one used in Question 3(c) was trained to learn the identity function. Table 12 show the output from the three hidden layer neurons of this network. The table shows that some output values are

inputs	hidden outputs			hidden re-mapped
10000000	0.56193951	0.99782185	0.01584416	110
01000000	0.00119330	0.53053587	0.00386037	010
00100000	0.00192434	0.00288832	0.90070578	001
00010000	0.00214312	0.99275195	0.99651131	011
00001000	0.99337806	0.00221516	0.99665411	101
00000100	0.99291924	0.39265810	0.00357696	100
00000010	0.38169448	0.00176780	0.00720031	000
00000001	0.99772615	0.99675162	0.98769268	111

Table 12: The identity function as learned by a BACKPROPAGATION network written in Python.

far from the asymptotic values of the sigmoid function.

Even though it seems obvious that since there exist a 3-bit binary representation for the 8 input patterns, it is not all that easy to get a neural network to learn this. The main reason for this is that the 3-bit representation is one of the most compact forms of representing the first 8 numbers. Autoencoders used on real data is often built with multiple (deep) layers, and with some redundancy built in. These also require a lot of experimentation with the number of layers, number of neurons per layer, and the choice of activation functions in each layer. It helps if the network designer (data scientist) has a good feel for the structure of the data.

Mark out of 100.
40 or less for clear indication that student does not understand the topic or evidence of plagiarism
50 for a fair understanding
60-70 for understanding and clear well defined examples
80+ for exceptional detail