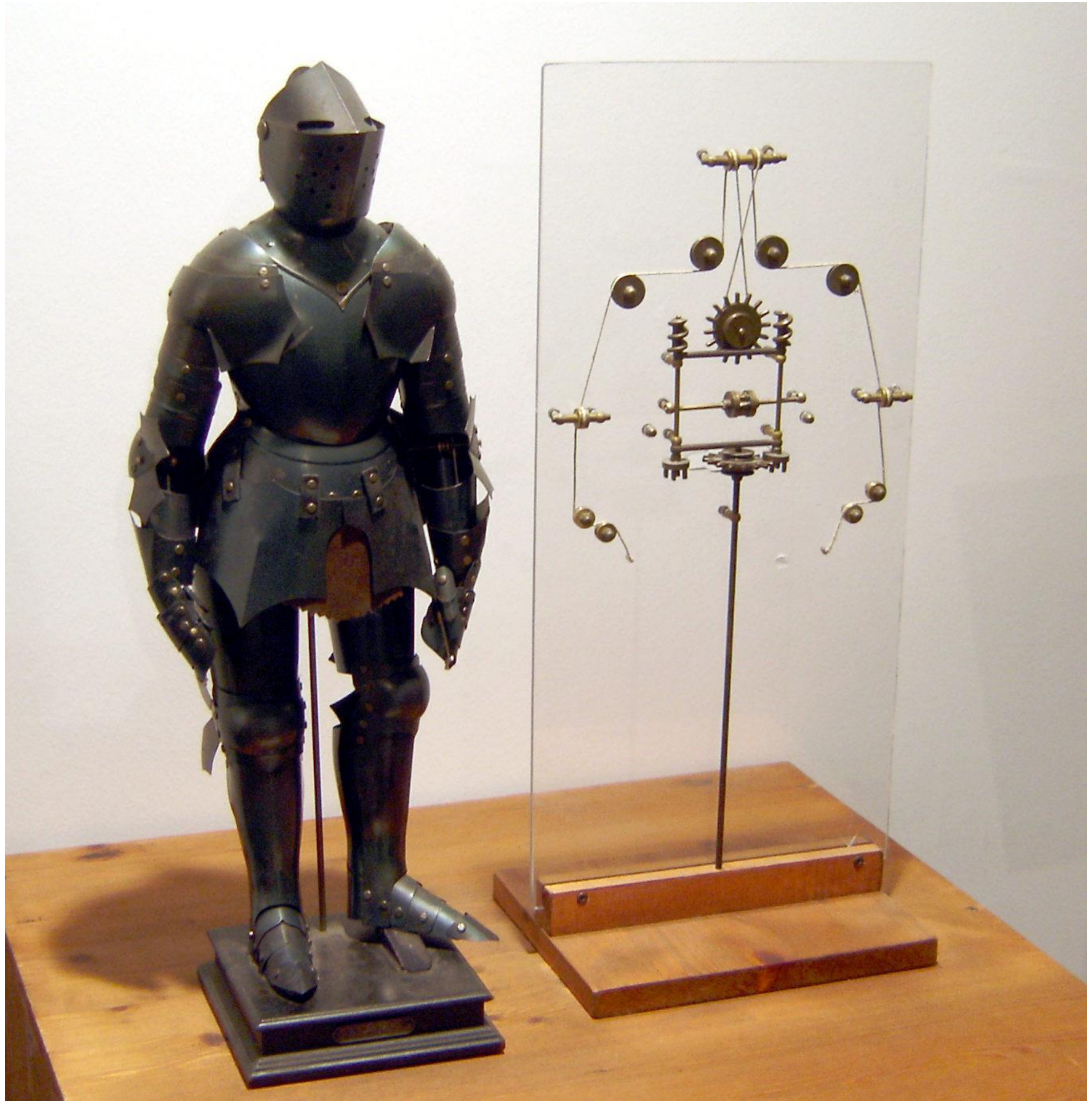
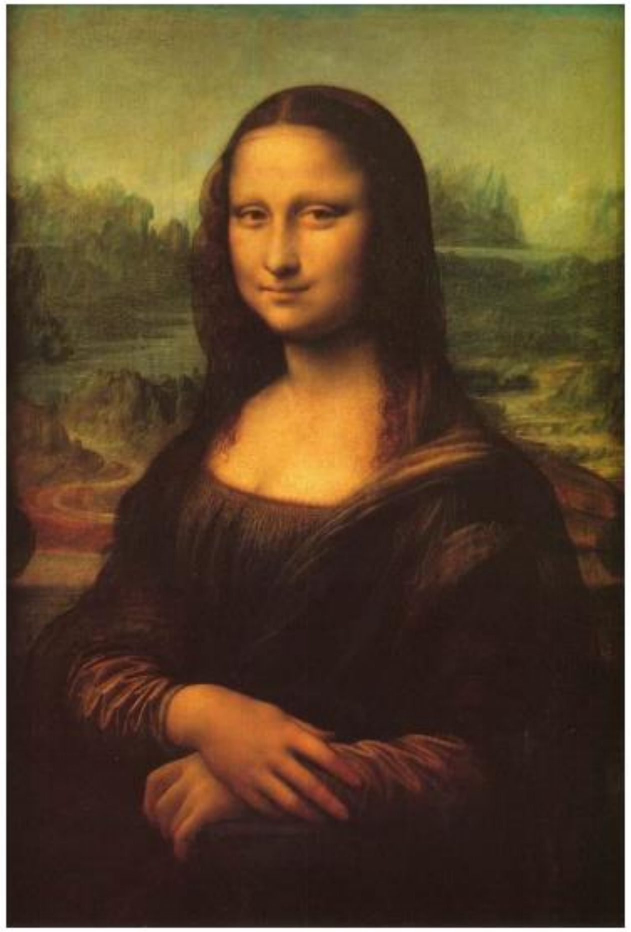




Leonardo's robot, was a humanoid automaton designed and possibly constructed by Leonardo da Vinci around the year 1495

Week 0 - Tutorial

Schedule

1. Introduction
2. Week-1 Tutorial Questions.

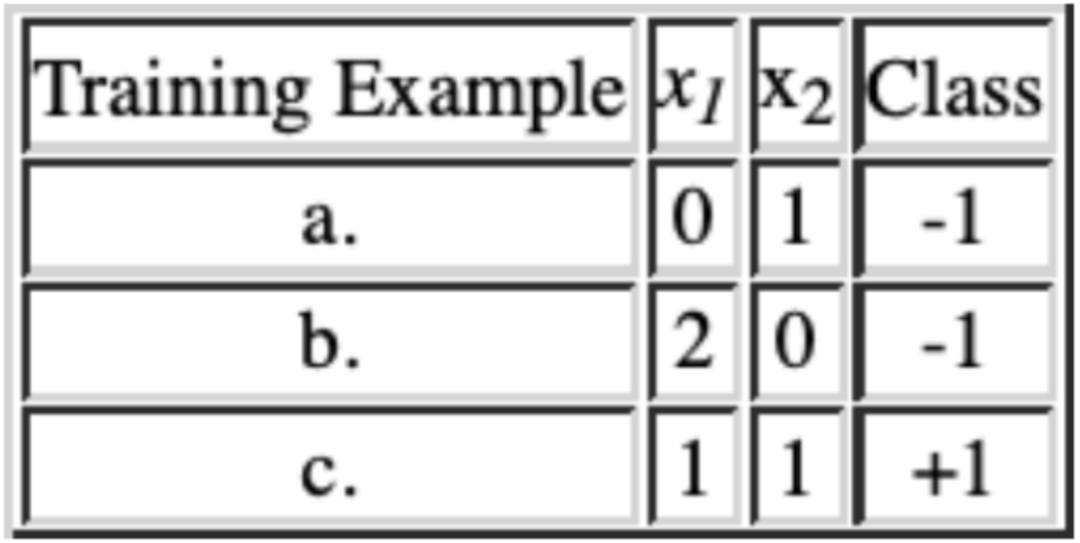
A Tour of the World's First Robot-Staffed Hotel

1. What AI technologies did they use in the video?
2. https://www.youtube.com/watch?v=C6bQHUlq664&ab_channel=WallStreetJournal

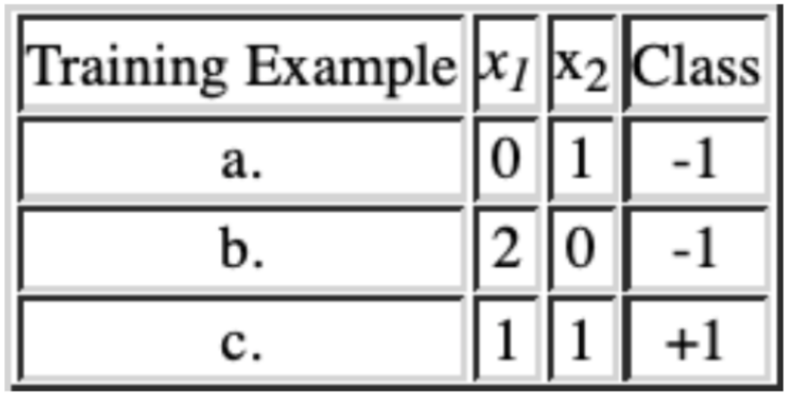
Week 2 - Tutorial

Q1: Activity 1

(1) Construct by hand a Perceptron which correctly classifies the following data; use your knowledge of plane geometry to choose appropriate values for the weights w0, w1 and w2.



Training Example	x_1	x_2	Class
a.	0	1	-1
b.	2	0	-1
c.	1	1	+1



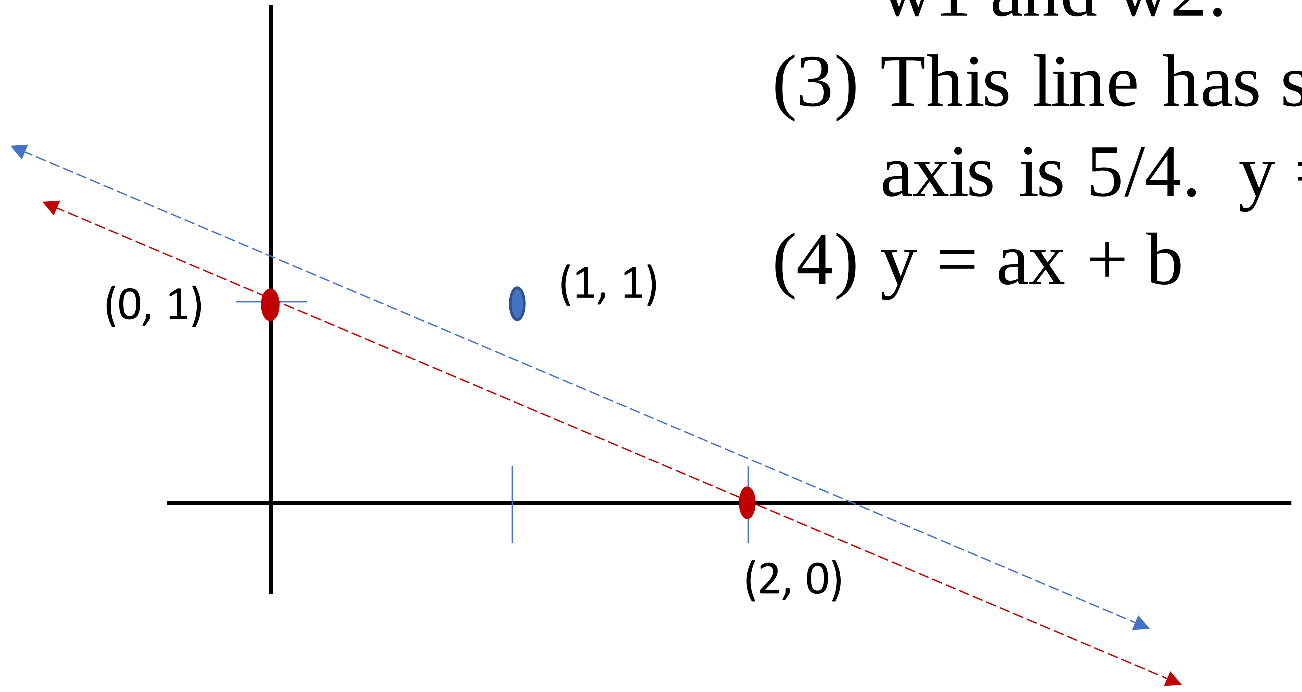
Training Example	x_1	x_2	Class
a.	0	1	-1
b.	2	0	-1
c.	1	1	+1

(1) The first step is to plot the data on a 2-D graph, and draw a line which separates the positive from the negative data points:

(2) Use your knowledge of plane geometry to choose appropriate values for the weights w0, w1 and w2.

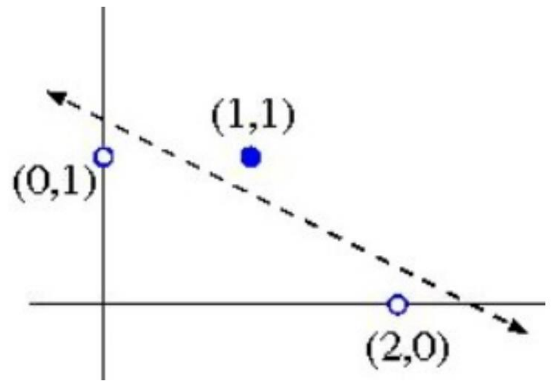
(3) This line has slope -1/2, the y-intersect to y-axis is 5/4. y = -1/2 x + 5/4 = 4y +2x -5 = 0

(4) y = ax + b



$$x_2 = \frac{5}{4} - \frac{x_1}{2}$$

i.e. $2x_1 + 4x_2 - 5 = 0$



Based on the two intersection points, we can derive the following line slope:

$$Slope = m = \frac{(y_2 - y_1)}{(x_2 - x_1)} = \frac{(0 - 1)}{(2 - 0)} = -0.5$$

$$Slope = m = \frac{(y_2 - y_1)}{(x_2 - x_1)} = \frac{(0 - 1)}{(2 - 0)} = -0.5$$

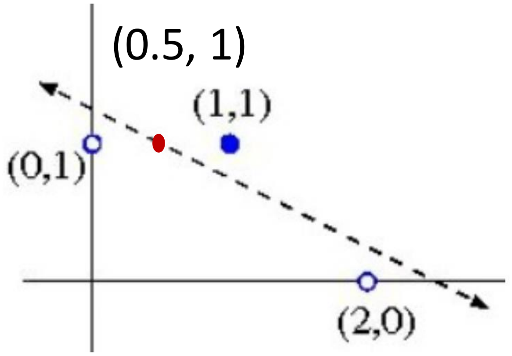
With the slope, the corresponding line equation can be partially calculated, utilising an identified point on the line, which will be (0.5, 1) for this example (the point between (0,1) and (1,1)):

$y = ax + b$

$\cong 1 = -0.5 * 0.5 + b$

$b = 5/4 \equiv 1.25$

$y = -0.5x + 1.25$



Since x2 is equivalent to y based on our dataset, we can deduce a quadratic equation using our line equation which will provide the weightings:

$x_2 = -0.5x_1 + 1.25$

$\equiv 0 = 0.5x_1 + x_2 - 1.25$

$\equiv 0 = 2x + 4y - 5$

This results in the following weights: w0 = -5, w1 = 2, w2 = 4

1) Line representation: ***y* = a*x* + b => x2 = ax1 + b**
where:
a is the slope or gradient of the line.
b is the y-intercept of the line.
x is the independent variable of the function y = f(x).

2) Distance from a point to a line:

$$\mathrm{distance}(ax + by + c = 0, (x_0, y_0)) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

3) a = -1/2 , y = -1/2x + b => 2y +x –b =0

4) $\frac{|2y+x-b|}{\sqrt{4+1}}$, put (2, 0) and (1, 1) into the distance formulation. => b = 5/4
5) y = -1/2x + 5/4 => 2x + 4y -5 =0 => w0 = -5, w1 = 2, w2 = 4

Q2 – Perception Learning Rule

Adjust the weights as each input is presented.

recall: $s = w_1 x_1 + w_2 x_2 + w_0$

if $g(s) = 0$ but should be 1,

$$\begin{aligned} w_k &\leftarrow w_k + \eta x_k \\ w_0 &\leftarrow w_0 + \eta \\ \text{so} \quad s &\leftarrow s + \eta\left(1 + \sum_k x_k^2\right) \end{aligned}$$

if $g(s) = 1$ but should be 0,

$$\begin{aligned} w_k &\leftarrow w_k - \eta x_k \\ w_0 &\leftarrow w_0 - \eta \\ \text{so} \quad s &\leftarrow s - \eta\left(1 + \sum_k x_k^2\right) \end{aligned}$$

otherwise, weights are unchanged. ($\eta > 0$ is called the **learning rate**)

Theorem: This will eventually learn to classify the data correctly, as long as they are **linearly separable**.

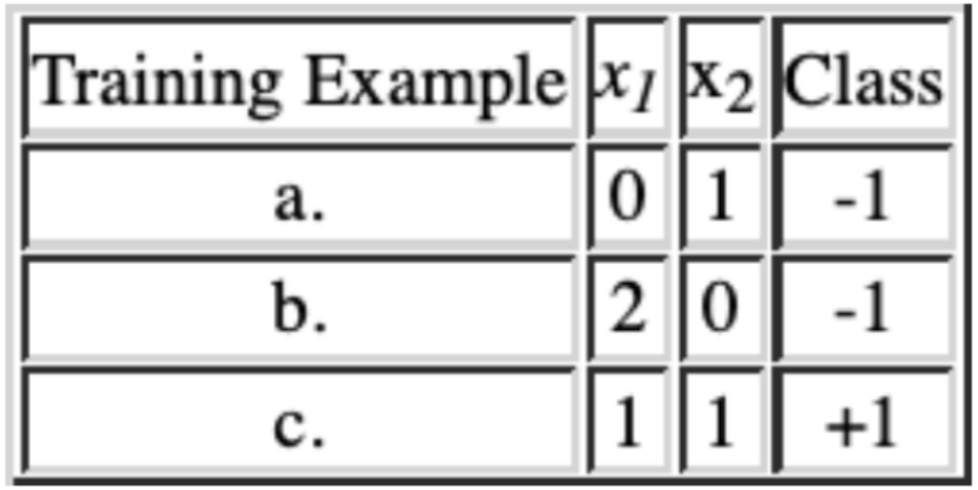
Q2 – Perception Learning Rule

(2) Demonstrate the Perceptron Learning Algorithm on the above data, using a learning rate of 1.0 and initial weight values of:

w_0 = -1.5

w_1 = 0

w_2 = 2



Training Example	x_1	x_2	Class
a.	0	1	-1
b.	2	0	-1
c.	1	1	+1

Iteration	w_0	w_1	w_2	Training Example	x_1	x_2	Class	$s=w_0+w_1x_1+w_2x_2$	Action
1	–1.5	0	2	a.	0	1	–	+0.5	Subtract
2	–2.5	0	1	b.	2	0	–	–2.5	None
3	–2.5	0	1	c.	1	1	+	–1.5	Add
4	–1.5	1	2	a.	0	1	–	+0.5	Subtract
5	–2.5	1	1	b.	2	0	–	–0.5	None
6	–2.5	1	1	c.	1	1	+	–0.5	Add
7	–1.5	2	2	a.	0	1	–	+0.5	Subtract
8	–2.5	2	1	b.	2	0	–	+1.5	Subtract
9	–3.5	0	1	c.	1	1	+	–2.5	Add
10	–2.5	1	2	a.	0	1	–	–0.5	None
11	–2.5	1	2	b.	2	0	–	–0.5	None
12	–2.5	1	2	c.	1	1	+	+0.5	None

Explain how each of the following could be constructed:

a) Perceptron to compute the OR function of m inputs

Set the bias weight to -½, all other weights to 1.

The OR function is almost always True. The only way it can be False is if all inputs are 0. Therefore, we set the bias to be slightly less than zero for this input.

b) Perceptron to compute the AND function of n inputs

Set the bias weight to $\left(\frac{1}{2} - n\right)$, *all other weights to 1.*

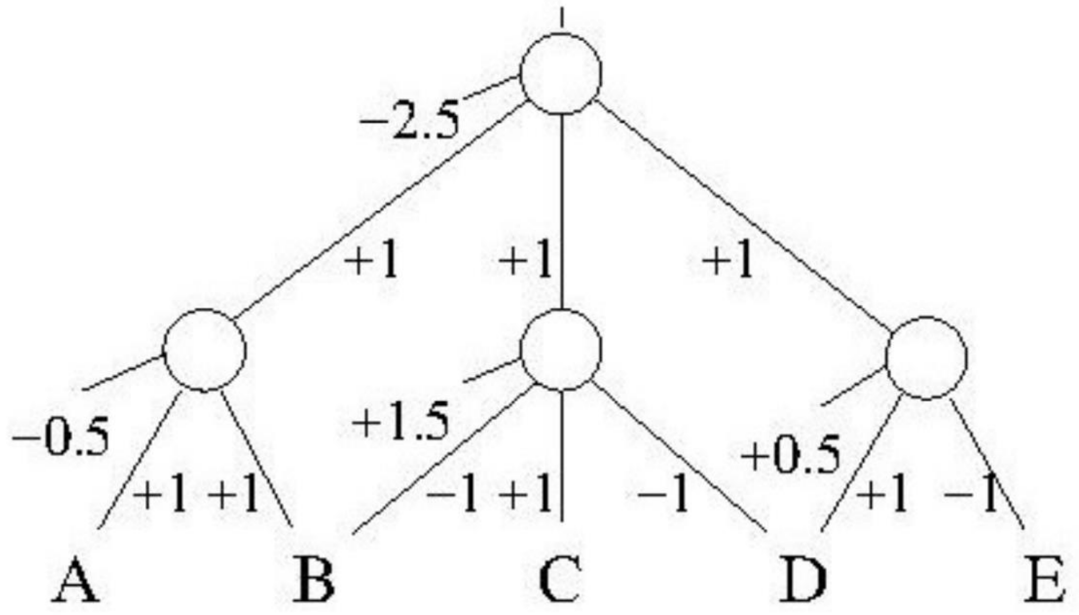
AND function is almost always False. The only way it can be True is if all inputs are 1. Therefore, we set the bias so that, when all inputs are 1, the combined sum is slightly greater than 0.

Q3

c) 2-Layer Neural Network to compute any (given) logical expression, assuming it is written in Conjunctive Normal Form.

Each hidden node should compute one disjunctive term in the expression. The weights should be -1 for items that are negated, +1 for others. The bias should be k - ½ where k is the number of items that are negated. The output node then computes the conjunction of all hidden nodes.

For example, here the network computes $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$



Explain how each of the following could be constructed:

a) Perceptron to compute the OR function of m inputs

Set the bias weight to -½, all other weights to 1.

The OR function is almost always True. The only way it can be False is if all inputs are 0. Therefore, we set the bias to be slightly less than zero for this input.

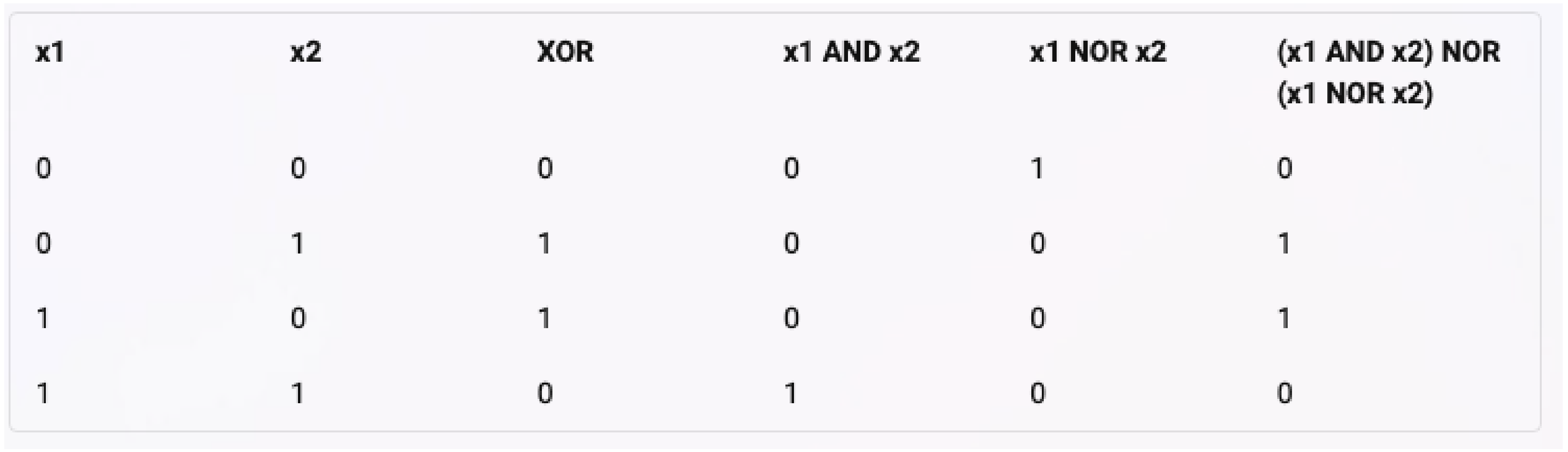
b) Perceptron to compute the AND function of n inputs

Set the bias weight to $\left(\frac{1}{2} - n\right)$*, all other weights to 1.*

AND function is almost always False. The only way it can be True is if all inputs are 1. Therefore, we set the bias so that, when all inputs are 1, the combined sum is slightly greater than 0.

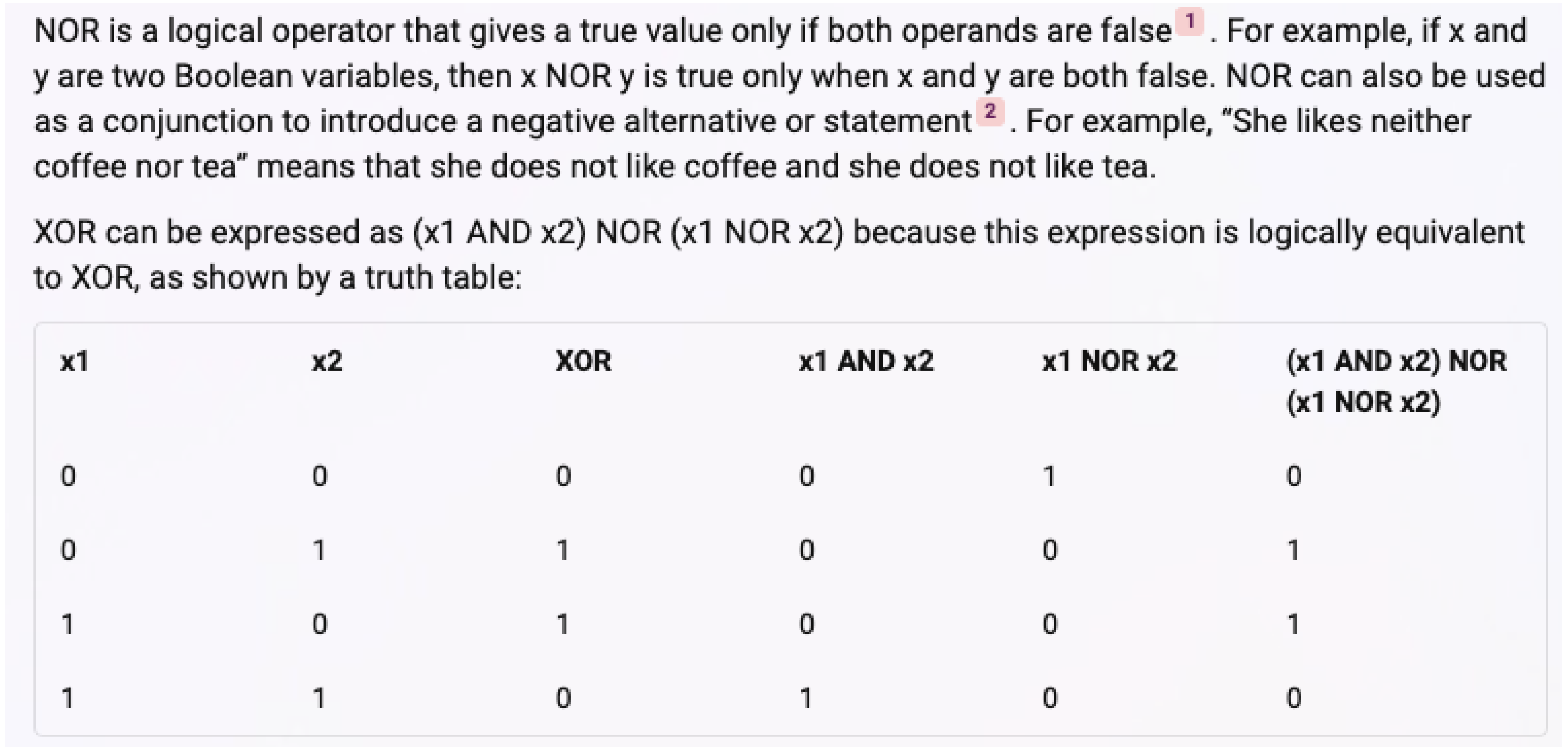
Q4 – XOR

XOR can be expressed as (x1 AND x2) NOR (x1 NOR x2) because this expression is logically equivalent to XOR. To see why, we can use a truth table to compare the outputs of XOR and the expression for different values of x1 and x2:



x1	x2	XOR	x1 AND x2	x1 NOR x2	(x1 AND x2) NOR (x1 NOR x2)
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	0	1	0	0

NOR: NOR is a logical operator that gives a true value only if both operands are false[1]. For example, if x and y are two Boolean variables, then x NOR y is true only when x and y are both false.
OR: is when both are x1, and x2 are false, then the result is false. NOR is opposite, when both x1, and x2 are false, then the x1 NOR x2 is True.



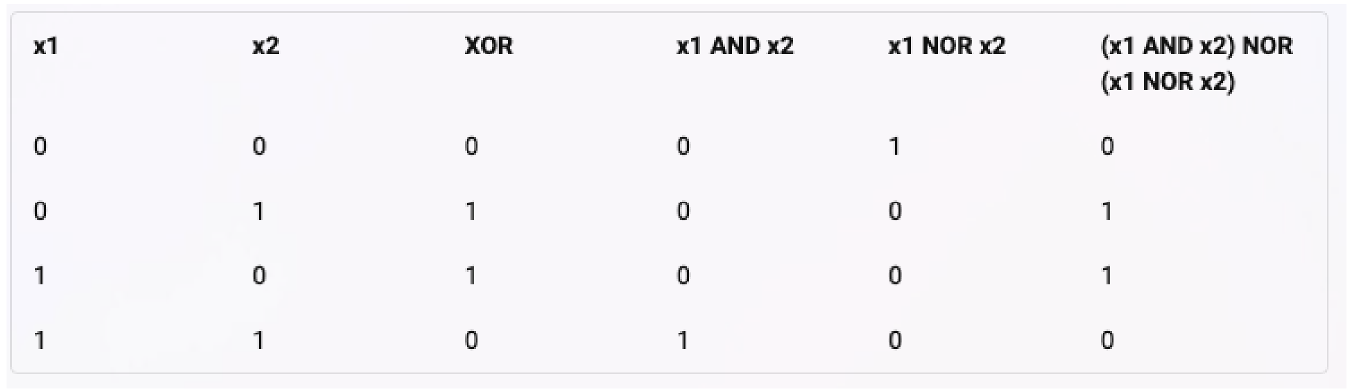
NOR is a logical operator that gives a true value only if both operands are false[1]. For example, if x and y are two Boolean variables, then x NOR y is true only when x and y are both false. NOR can also be used as a conjunction to introduce a negative alternative or statement[2]. For example, "She likes neither coffee nor tea" means that she does not like coffee and she does not like tea.

XOR can be expressed as (x1 AND x2) NOR (x1 NOR x2) because this expression is logically equivalent to XOR, as shown by a truth table:

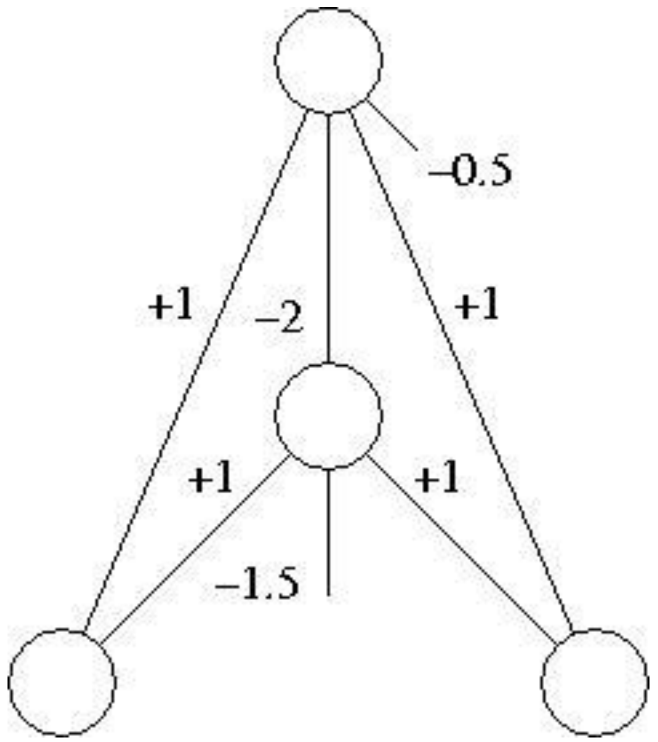
x1	**x2**	**XOR**	**x1 AND x2**	**x1 NOR x2**	**(x1 AND x2) NOR (x1 NOR x2)**
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	0	1	0	0

Q4 – XOR

XOR can be expressed as (x1 AND x2) NOR (x1 NOR x2) because this expression is logically equivalent to XOR. To see why, we can use a truth table to compare the outputs of XOR and the expression for different values of x1 and x2:



x1	x2	XOR	x1 AND x2	x1 NOR x2	(x1 AND x2) NOR (x1 NOR x2)
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	0	1	0	0

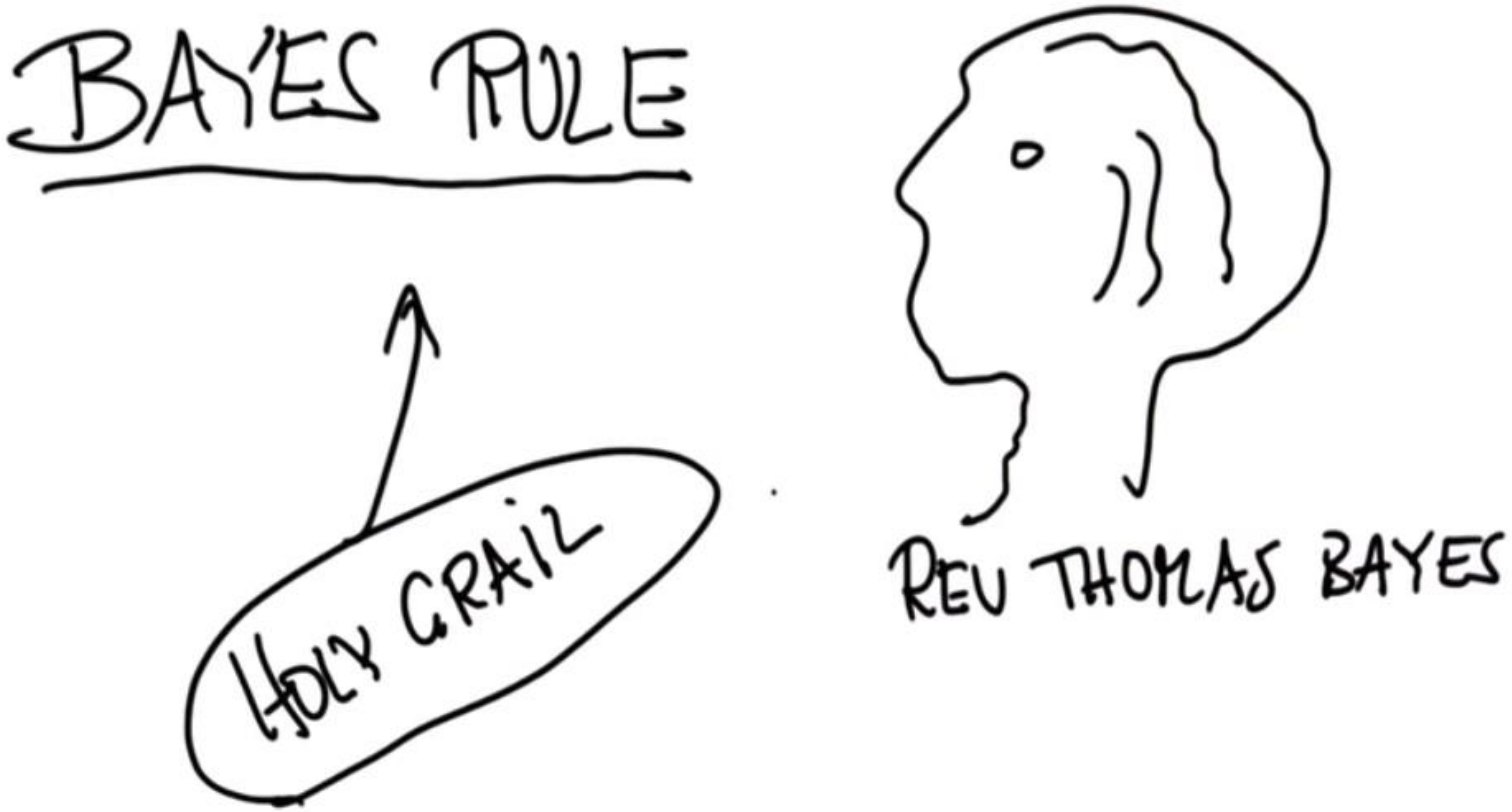


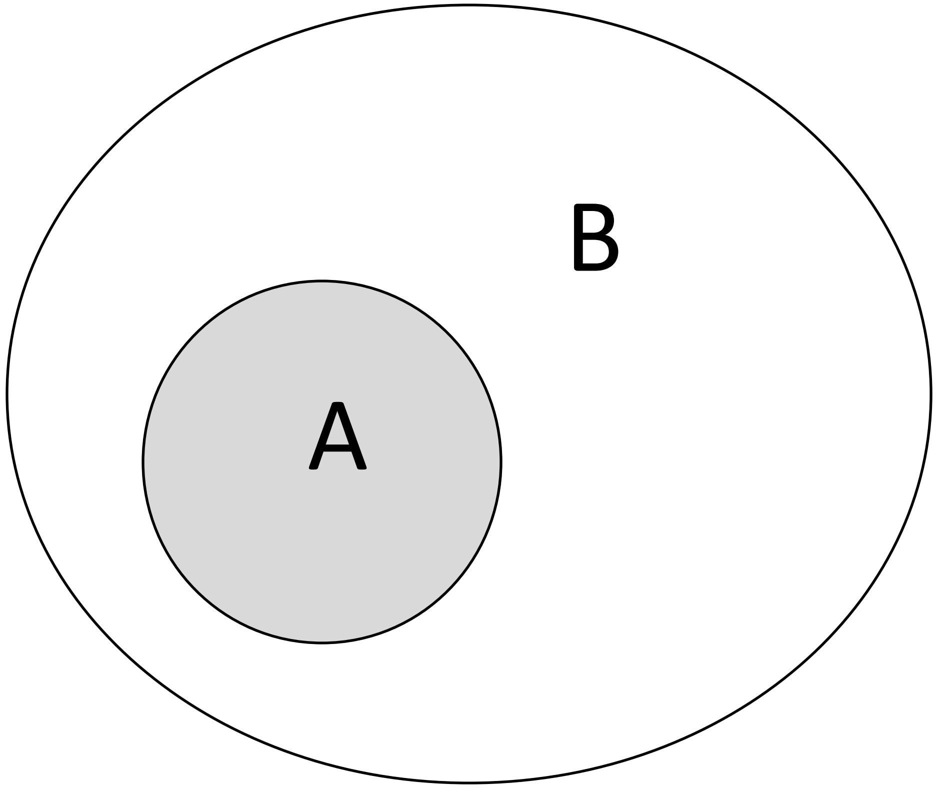
The hidden layer is (x1 AND x2), when both x1 and x2 are True (1), the output is true, otherwise the output is false (minus value).
E.g. if x1 and x2 both are true, both are 1, then x1 XOR x2 shall be 0.
The hidden layer's output is true, adding activation, the output is 1, then the final value is x1 x 1 + x2 x1 + (-2 x1) -0.5 = -0.5, slightly less than 0, is False.
E.g.2 if x1 is True, x2 is False, then the output of (x1 AND x2) is False, activation result is 0, then the final result is x1 x 1 + x2 x0 + (-2x0) + -0.5 = 0.5, is True, which is the output of XOR.

```
# Import libraries
import torch
import torch.nn as nn

# Define XOR inputs and outputs
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float)

# Define the network architecture
# Input layer: 2 neurons
# Hidden layer: 1 neuron
# Output layer: 1 neuron
# Shortcut connections: from input layer to output layer
class XORNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(2, 1) # hidden layer
        self.output = nn.Linear(3, 1, bias=False) # output layer

    def forward(self, x):
        h = torch.sigmoid(self.hidden(x)) # hidden layer activation
        xh = torch.cat((x, h), dim=1) # concatenate input and hidden layer
        y = torch.sigmoid(self.output(xh)) # output layer activation
        return y
```

Week 3 - Tutorial

Reasoning with Uncertainty





Definition: $P(A|B) = P(A \wedge B)/P(B)$ provided $P(B) > 0$

Product Rule: $P(A \wedge B) = P(A|B).P(B)$

$= P(B \wedge A) = P(B|A).P(A)$

A Bayesian network, Bayes network, belief network, decision network, Bayes(ian) model or probabilistic directed acyclic graphical model is a probabilistic graphical model (a type of statistical model) that represents a set of variables and their conditional dependencies via a directed acyclic graph(DAG).

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Definition Proof

1. $P(A \wedge B) = P(A|B).P(B)$
   $P(B \wedge A) = P(B|A).P(A)$
   Now $P(A \wedge B) = P(B \wedge A)$ [why exactly?]
   Therefore $P(A|B).P(B) = P(B|A).P(A)$
   Rearranging gives Bayes' Rule $P(A|B) = \frac{P(B|A).P(A)}{P(B)}$ if $P(B) > 0$

Q1:Bayes’Rule

(1) Bayes’ Rule: One bag contains 2 red balls and 3 white balls. Another bag contains 3 red balls and 2 green balls. One of these bags is chosen at random, and two balls are drawn randomly from that bag, without replacement. Both of the balls turn out to be red. What is the probability that the first bag is the one that was chosen?

Q1:Bayes'Rule

(1) Bayes' Rule: One bag contains 2 red balls and 3 white balls. Another bag contains 3 red balls and 2 green balls. One of these bags is chosen at random, and two balls are drawn randomly from that bag, without replacement. Both of the balls turn out to be red.

What is the probability that the first bag is the one that was chosen?

Let B = first bag is chosen, R = both balls are red. Then

P (R | B) = (2/5)*(1/4) = 1/10

P (R | ¬B) = (3/5)*(2/4) = 3/10

P (R) = (1/2)*(1/10) + (1/2)*(3/10) = 1/5

P (B | R) = P(R|B)*P(B) / P(R) = (1/10)*(1/2)/(1/5) = 1/4

Information Entropy

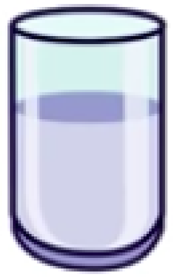
The concept of "information entropy" (using the concept of entropy in thermodynamics) is proposed by Shannon to solve the problem of information measurement in 1948.

Higher entropy means more randomness.

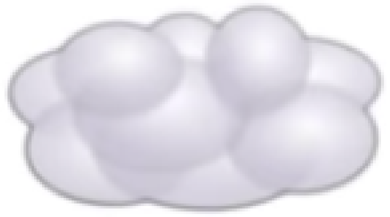
Assuming that the top 32 of the World Cup finals has been produced, what is the information entropy on the random variable X: X= "Who is the World Cup champion in the top 32 of the 2028 World Cup in Russia?”

Entropy

Low



Medium



High

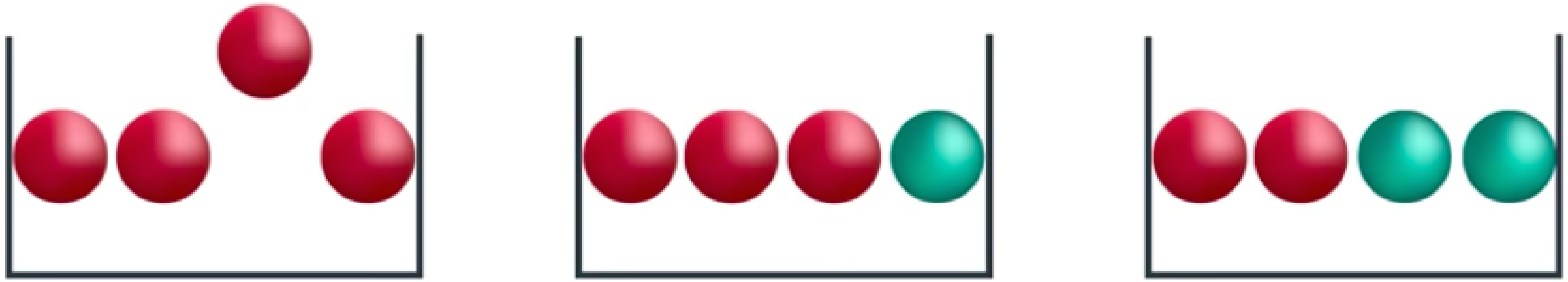
Entropy

- Entropy is a measure of "randomness" (lack of uniformity)
  - Related to prior distribution of some random variable
  - Higher entropy means more randomness
  - "Information" (about distribution) reduces entropy
- Idea: Split based on information gain
  - Loss of entropy based on "communicating" value of attribute
  - Related to Shannon's information theory
  - Measure information gain in bits

Definition. If the prior probabilities of n attribute values are $p_1, \cdots, p_n$, then the entropy of the distribution is

$$H(\langle p_1, \cdots, p_n \rangle) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

What ball will we pick?



Entropy

	P(red)	P(blue)	P(winning)	$-\log_2(P(\text{winning}))$	Entropy
🔴🔴🔴🔴	1	0	$1 \times 1 \times 1 \times 1 = 1$	0+0+0+0	0
🔴🔴🔴🟢	0.75	0.25	$0.75 \times 0.75 \times 0.75 \times 0.25 = 0.105$	0.415+0.415+0.415+2	0.81
🔴🔴🟢🟢	0.5	0.5	$0.5 \times 0.5 \times 0.5 \times 0.5 = 0.0625$	1+1+1+1	1

Entropy and Huffmann Coding

Example 2: $H(\langle 0.5, 0.25, 0.25 \rangle) = 1.5$ bits

To encode a message consisting of the letters `A`, `B` and `C`, where `B` and `C` occur equally often but `A` occurs twice as often as the other two letters, assign `A=0`, `B=10`, `C=11`

The average number of bits needed to encode each letter is 1.5

If the letters occur in some other proportion, they need to be "blocked" in some order to encode them efficiently

The average number of bits required by the most efficient coding scheme is given by the entropy

$$H(\langle p_1, \cdots, p_n \rangle) = \Sigma_{i=1}^{n} -p_i \log_2 p_i$$

Q2:Entropy and KL-Divergence for Discrete Distributions

Consider these two probability distributions on the same space $\Omega = \{A, B, C, D\}$

$p = \langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8} \rangle$
$q = \langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2} \rangle$

Q1: Construct a Huffmann tree for each distribution p and q

Q2:Entropy and KL-Divergence for Discrete Distributions

Huffman tree building

A simple algorithm (buildHuff):

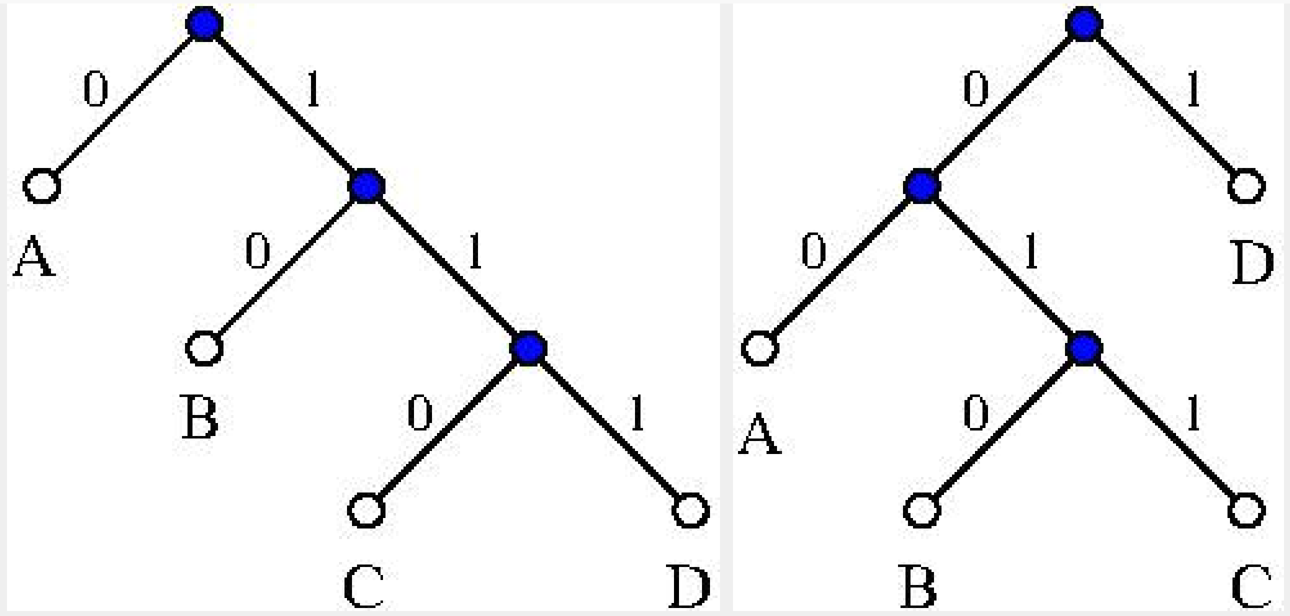
1. Prepare a collection of n initial Huffman trees, each of which is a single leaf node. Put the n trees onto a priority queue organized by weight (frequency).
2. Remove the first two trees (the ones with lowest weight). Join these two trees to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two children trees.
3. Put this new tree into the priority queue.
4. Repeat steps 2-3 until all of the partial Huffman trees have been combined into one.

Q2:Entropy and KL-Divergence for Discrete Distributions

Q1: Construct a Huffmann tree for each distribution p and q

$p = \langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8} \rangle$

$q = \langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2} \rangle$



Note: the answer is not unique; this is one possible tree in each case.

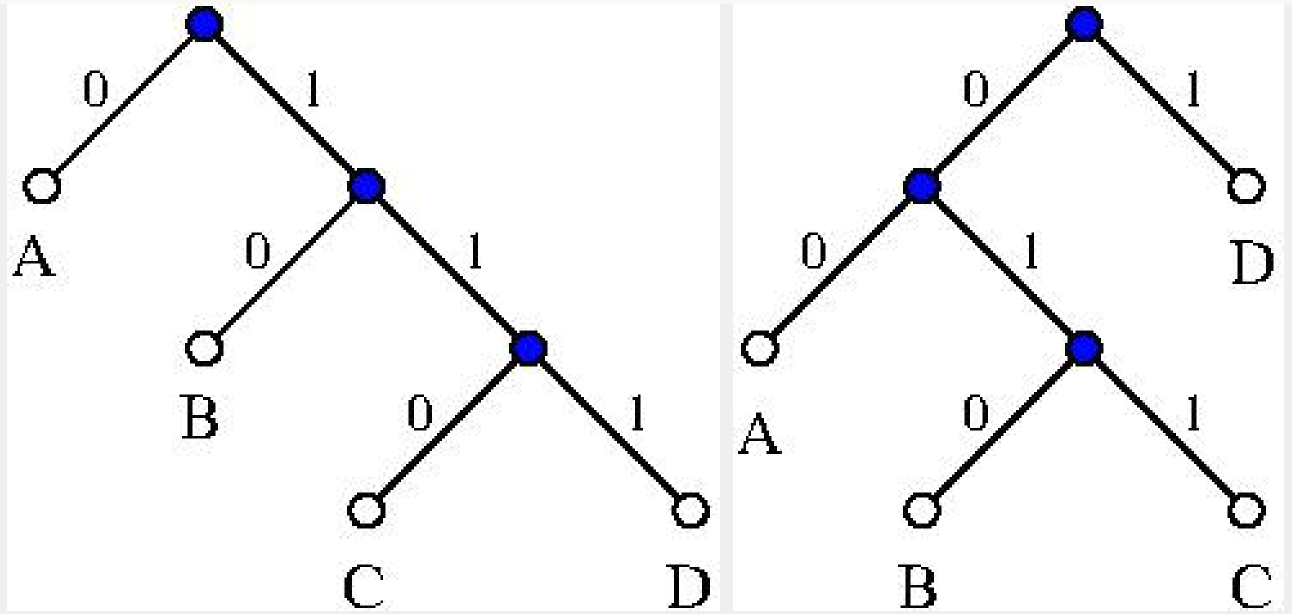
For distribution p, the process would look like this:

1.Start with the four trees: [1/2], [1/4], [1/8], [1/8].

2.Merge the two [1/8] trees into one tree: [1/2], [1/4], [1/4].

3.Merge the two [1/4] trees into one tree: [1/2], [1/2].

4.Merge the two [1/2] trees into one tree: [1]. This is the Huffman tree for distribution p.



Note: the answer is not unique; this is one possible tree in each case.

q:

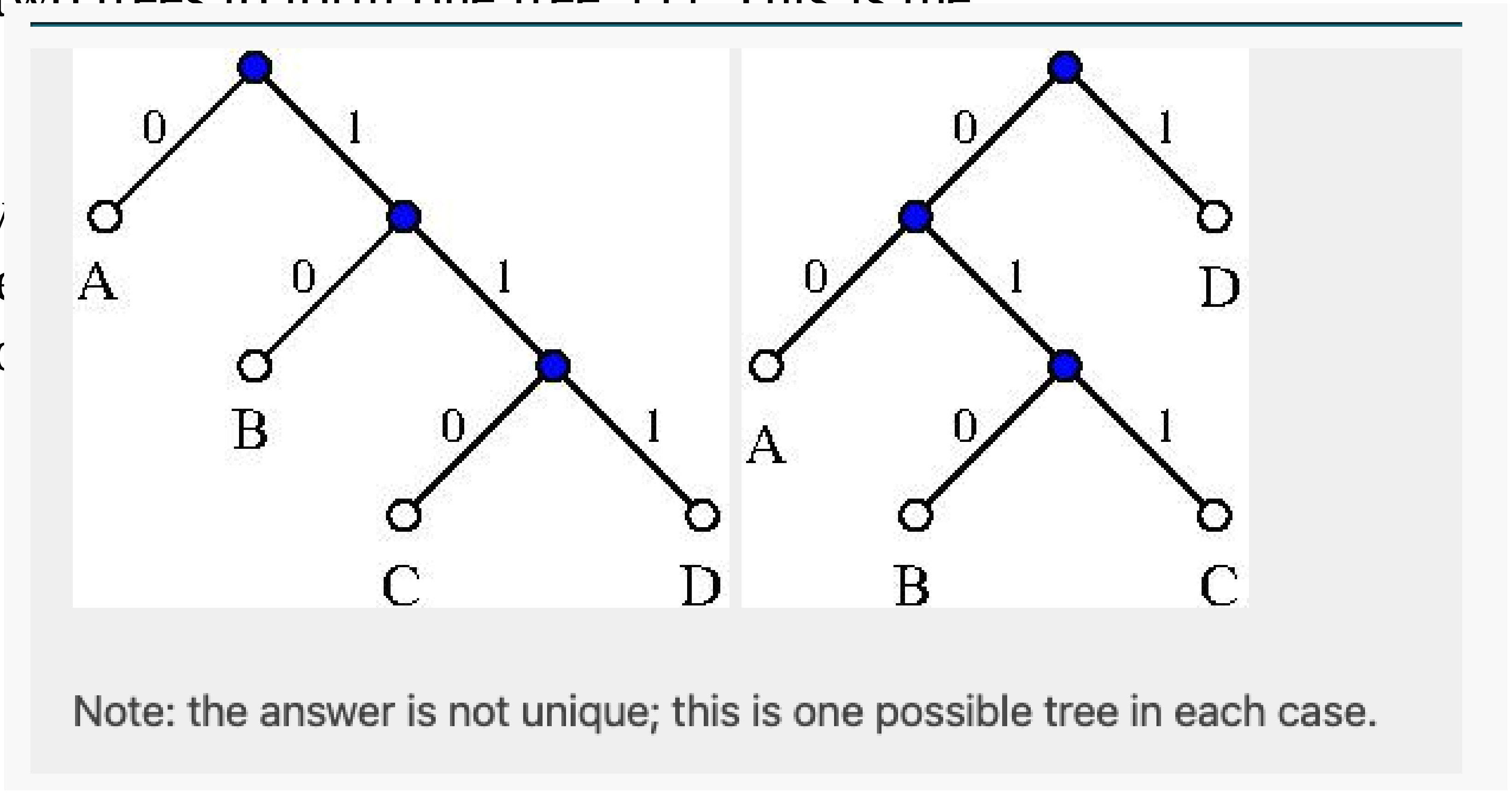
1.Start with the four trees: [1/4], [1/8], [1/8], [1/2].

2.The two smallest probabilities are both 1/8. Merge these two trees into a single tree: [1/4], [1/4], [1/2].

3.Now, merge the two trees with a probability of 1/4. You'll have two trees remaining: [1/2], [1/2].

4.Finally, merge the remaining two trees to form one tree: [1]. This is the Huffman tree for distribution q.

In the Huffman tree for q, the sy
have the shortest code, while the
and 1/8) will have the longest co



Note: the answer is not unique; this is one possible tree in each case.

Entropy and Huffmann Coding

Example 2: $H(\langle 0.5, 0.25, 0.25 \rangle) = 1.5$ bits

To encode a message consisting of the letters `A`, `B` and `C`, where `B` and `C` occur equally often but `A` occurs twice as often as the other two letters, assign `A=0`, `B=10`, `C=11`

The average number of bits needed to encode each letter is 1.5

If the letters occur in some other proportion, they need to be "blocked" in some order to encode them efficiently

The average number of bits required by the most efficient coding scheme is given by the entropy

$$H(\langle p_1, \cdots, p_n \rangle) = \Sigma_{i=1}^{n} -p_i \log_2 p_i$$

Q2. Compute the entropy H(p)

$p = \langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8} \rangle$
$q = \langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2} \rangle$

$$H(\langle p_1, \cdots, p_n \rangle) = \sum_{i=1}^{n} -p_i \log_2 p_i$$

$$H(p) = H(q)$$
$$= \tfrac{1}{2}(-\log \tfrac{1}{2}) + \tfrac{1}{4}(-\log \tfrac{1}{4}) + \tfrac{1}{8}(-\log \tfrac{1}{8}) + \tfrac{1}{8}(-\log \tfrac{1}{8})$$
$$= \tfrac{1}{2}(1) + \tfrac{1}{4}(2) + \tfrac{1}{8}(3) + \tfrac{1}{8}(3) = 1.75$$

Q2.c Compute the KL-Divergence in each direction DKL(p || q) and DKL(q || p)

1. KL divergence(Relative entropy/Information gain): Kullback-Leibler (KL) divergence is a measure of how different two probability distributions are. A 'true' probability distribution, P, and an approximate distribution, Q.
2. The KL divergence of Q from P, denoted DKL(P||Q), can be thought of as the "distance" from P to Q.
3. DKL(p||q): This measures the amount of information loss when q is used to approximate p. In other words, it tells us how much information we would lose if we used distribution q to encode/replace events from distribution p.

Q2.c KL-Divergence

KL divergence(Relative entropy/Information gain):

$$\begin{aligned}
D_{KL}(p||q) &= H(p,q) - H(p) \\
&= -\sum p(x)\log q(x) + \sum p(x)\log p(x) \\
&= -\sum p(x)\log\frac{q(x)}{p(x)} \\
&= \sum p(x)\log\frac{p(x)}{q(x)}
\end{aligned}$$

Q2.c Compute the KL-Divergence in each direction DKL(p ‖ q) and DKL(q ‖ p)

$p = \langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8} \rangle$
$q = \langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2} \rangle$

$$D_{KL}(p||q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

DKL(p ‖ q) = ½(log(*p(x)*)–log(*q(x)*)) + ¼ (log(*p(x)*)–log(*q(x)*))
+ 1/8 (log(*p(x)*)–log(*q(x)*)) + 1/8 (log(*p(x)*)–log(*q(x)*))
= ½(2–1) + ¼(3–2) + ⅛(3–3) + ⅛(1–3) = 0.5

DKL(q ‖ p) = ¼(1–2) + ⅛(2–3) + ⅛(3–3) + ½(3–1) = 0.625

Q2.c Compute the KL-Divergence in each direction DKL(p ∥ q) and DKL(q ∥ p)

$p = \langle \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{8}, \tfrac{1}{8} \rangle$
$q = \langle \tfrac{1}{4}, \tfrac{1}{8}, \tfrac{1}{8}, \tfrac{1}{2} \rangle$

DKL(q ∥ p) = Σ q(x) * log2(q(x)/p(x))

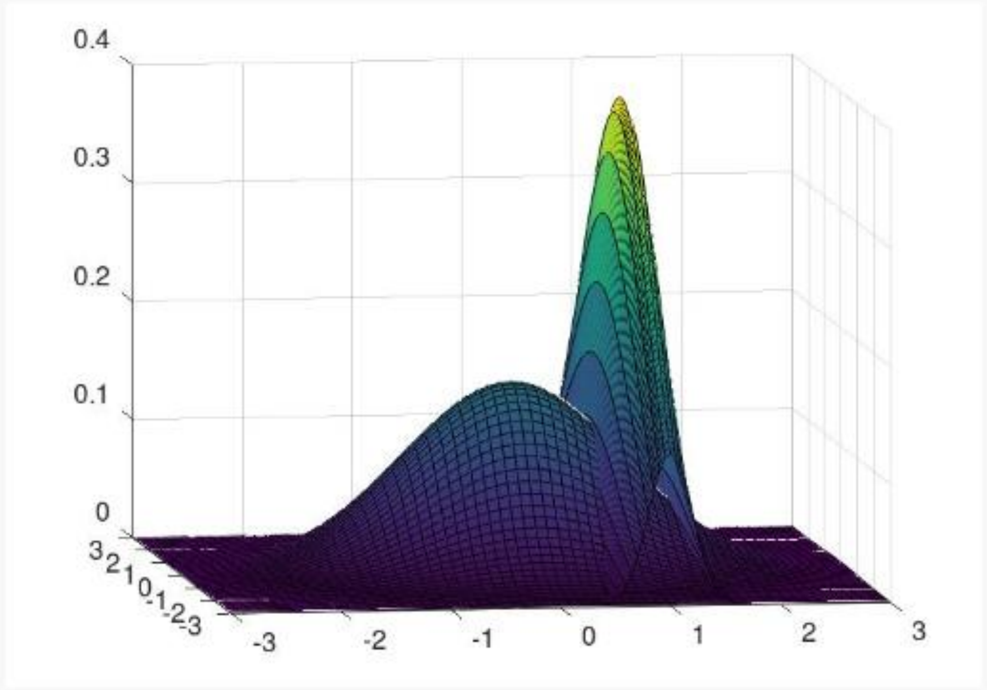
DKL(q ∥ p) = $\tfrac{1}{4}(1–2) + \tfrac{1}{8}(2–3) + \tfrac{1}{8}(3–3) + \tfrac{1}{2}(3–1) = 0.625$

Q2c. Which one is larger? Why?

DKL(q ‖ p) is larger, mainly because the frequency of D has increased from ⅛ to ½, so it incurs a cost of 3–1=2 additional bits every time it occurs (which is often).

DKL(p ‖ q) = ½(2–1) + ¼(3–2) + ⅛(3–3) + ⅛(1–3) = 0.5
DKL(q ‖ p) = ¼(1–2) + ⅛(2–3) + ⅛(3–3) + ½(3–1) = 0.625

3. Entropy, KL-Divergence and W2 Distance for Bivariate Gaussians

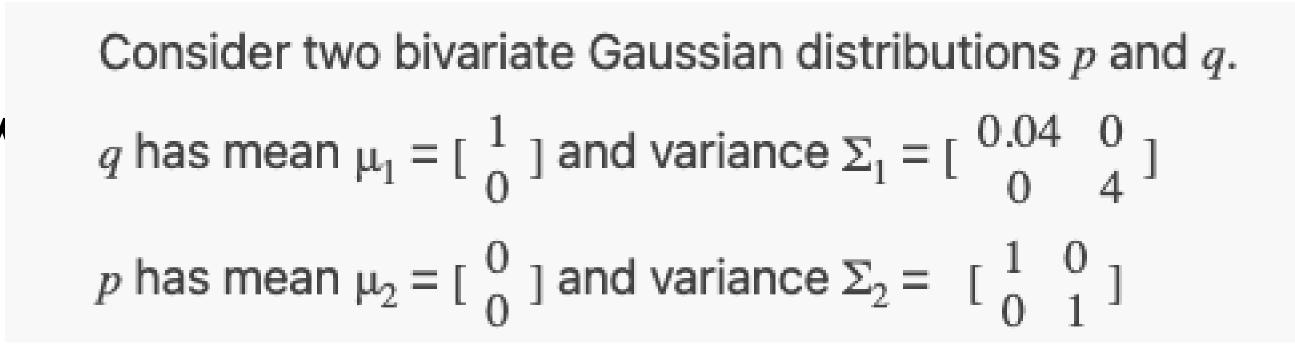


Consider two bivariate Gaussian distributions p and q.

q has mean $\mu_1 = [\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}]$ and variance $\Sigma_1 = [\begin{smallmatrix} 0.04 & 0 \\ 0 & 4 \end{smallmatrix}]$

p has mean $\mu_2 = [\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}]$ and variance $\Sigma_2 = [\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}]$

3. Entropy, KL-Divergence and W
Gaussians



Consider two bivariate Gaussian distributions p and q.

q has mean $\mu_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and variance $\Sigma_1 = \begin{bmatrix} 0.04 & 0 \\ 0 & 4 \end{bmatrix}$

p has mean $\mu_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and variance $\Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

a. Compute the Entropy H(p) and H(q). Which one is larger? Why?

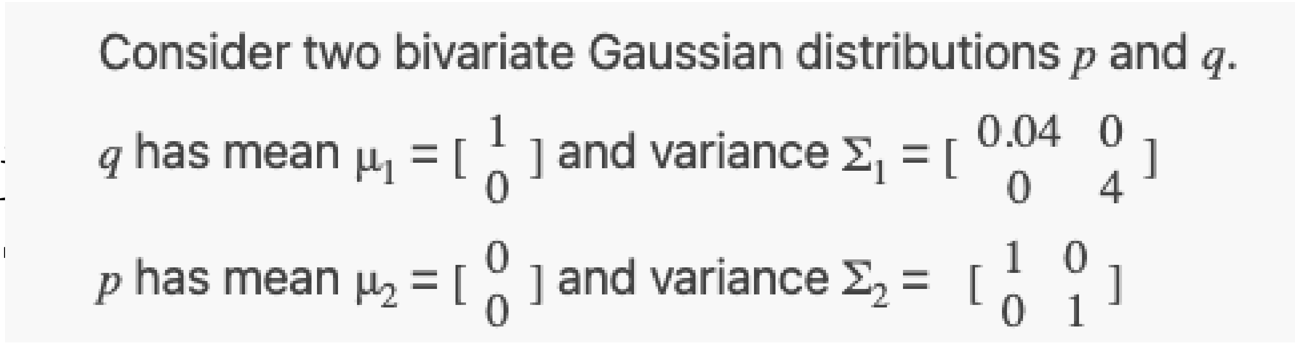
$H(p) = \frac{1}{2}\log|\Sigma 2| + 1 + \log(2\pi) = 1 + \log(2\pi)$

$H(q) = \frac{1}{2}\log|\Sigma 1| + 1 + \log(2\pi) = \log(0.4) + 1 + \log(2\pi)$

When compared to p, q has been compressed by a factor of 5 in one direction but stretched by a factor of 2 in the other, resulting an overall compression factor of 0.4. Consequently, H(q) is smaller than H(p) because the probability distribution is more concentrated and therefore less "uncertain".

H(p) is larger than H(q) by $-\log(0.4) = \log(2.5)$

4. Compute the KL-Divergence DKL(p || q). Which one is larger


Consider two bivariate Gaussian distributions p and q.

q has mean $\mu_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and variance $\Sigma_1 = \begin{bmatrix} 0.04 & 0 \\ 0 & 4 \end{bmatrix}$

p has mean $\mu_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and variance $\Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

DKL(q || p) $= ½[||\mu 1||2 + \mathrm{Trace}(\Sigma 1) – \log|\Sigma 1| – d] / 2$

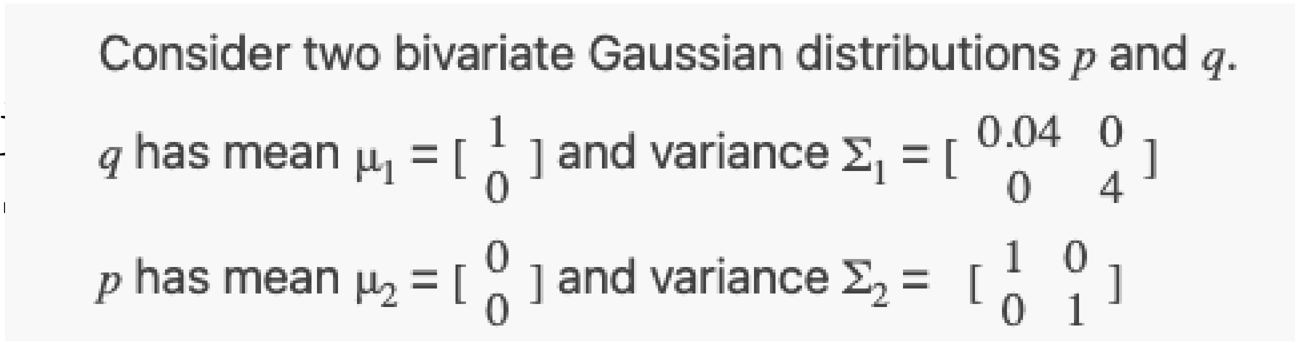
$= [1 + 4.04 – \log(0.16) – 2]/2 = 1.52 – \log(0.4)$

DKL(p || q) $= [(\mu 1-\mu 2)T\Sigma 1\text{-}1(\mu 1-\mu 2) + \mathrm{Trace}(\Sigma 1\text{-}1\Sigma 2) + \log|\Sigma 1| - \log|\Sigma 2| - d] / 2$

$= [25 + 25.25 + \log(0.16) – \log(1) – 2] / 2 = 24.125 + \log(0.4)$

The second one is much larger, because there are places where p is large but q is very close to zero.

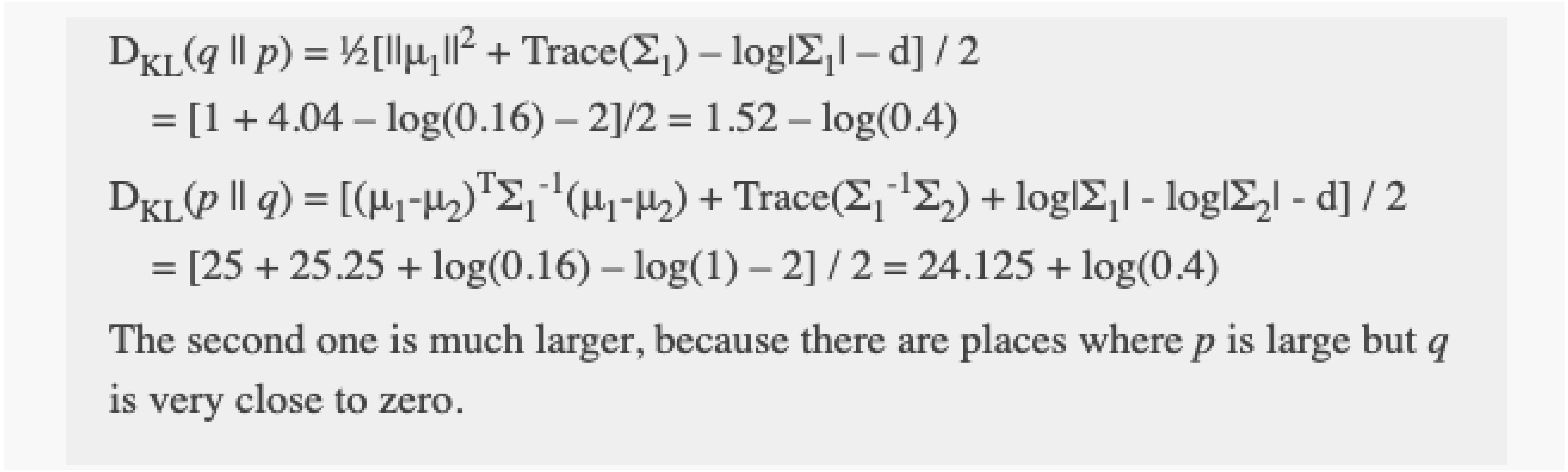
4. Compute the KL-Divergence DKL(p || q). Which one is larger



Consider two bivariate Gaussian distributions p and q.

q has mean $\mu_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and variance $\Sigma_1 = \begin{bmatrix} 0.04 & 0 \\ 0 & 4 \end{bmatrix}$

p has mean $\mu_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and variance $\Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$



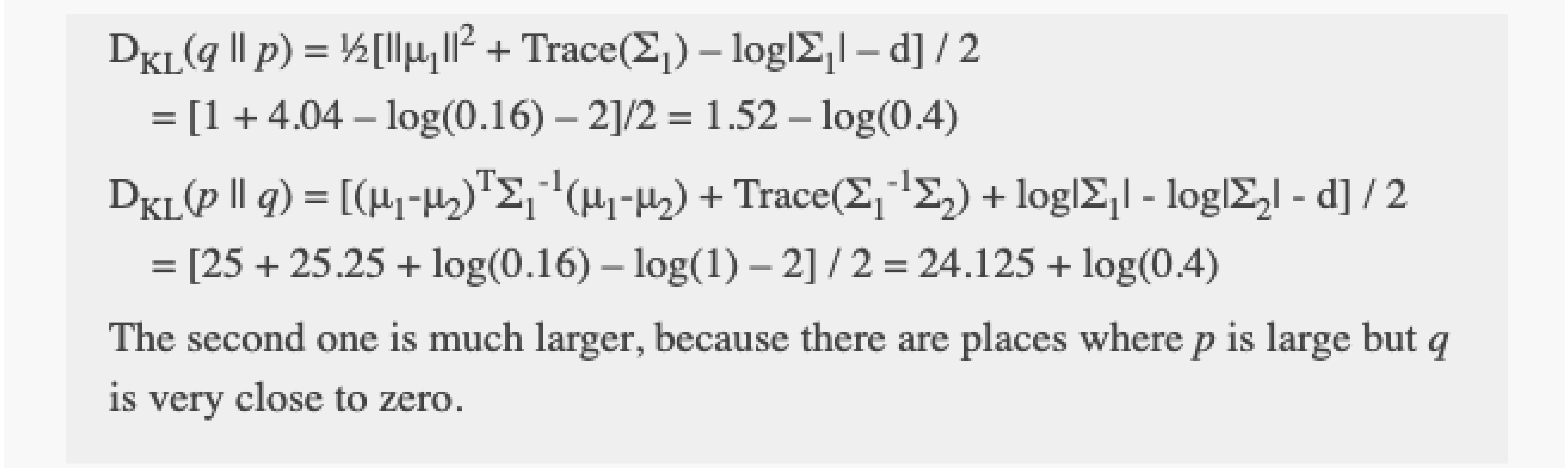
$$D_{KL}(q \,||\, p) = ½[||\mu_1||^2 + \mathrm{Trace}(\Sigma_1) - \log|\Sigma_1| - d] / 2$$

$$= [1 + 4.04 - \log(0.16) - 2]/2 = 1.52 - \log(0.4)$$

$$D_{KL}(p \,||\, q) = [(\mu_1-\mu_2)^T\Sigma_1^{-1}(\mu_1-\mu_2) + \mathrm{Trace}(\Sigma_1^{-1}\Sigma_2) + \log|\Sigma_1| - \log|\Sigma_2| - d] / 2$$

$$= [25 + 25.25 + \log(0.16) - \log(1) - 2] / 2 = 24.125 + \log(0.4)$$

The second one is much larger, because there are places where p is large but q is very close to zero.



$$D_{KL}(q \,\|\, p) = \tfrac{1}{2}[\|\mu_1\|^2 + \mathrm{Trace}(\Sigma_1) - \log|\Sigma_1| - d] / 2$$

$$= [1 + 4.04 - \log(0.16) - 2]/2 = 1.52 - \log(0.4)$$

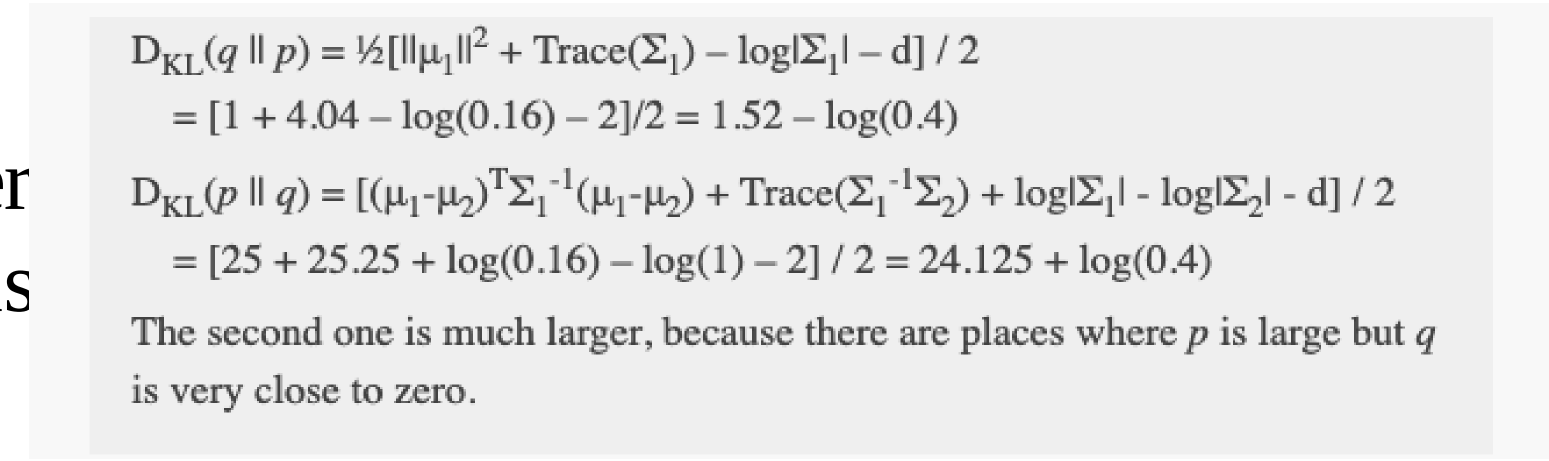
$$D_{KL}(p \,\|\, q) = [(\mu_1-\mu_2)^T\Sigma_1^{-1}(\mu_1-\mu_2) + \mathrm{Trace}(\Sigma_1^{-1}\Sigma_2) + \log|\Sigma_1| - \log|\Sigma_2| - d] / 2$$

$$= [25 + 25.25 + \log(0.16) - \log(1) - 2] / 2 = 24.125 + \log(0.4)$$

The second one is much larger, because there are places where *p* is large but *q* is very close to zero.

- ½ is a scaling factor that comes from the mathematical derivation of the KL divergence for Gaussian distributions.
- $||\mu1||$^2 is the squared length (also known as the squared norm) of the mean vector μ1 of distribution q.
- Trace(Σ1) is the trace of the covariance matrix Σ1 of distribution q. The trace of a matrix is the sum of its diagonal elements.
- log|Σ1| is the natural logarithm of the determinant of the covariance matrix Σ1. The determinant is a special number that can be calculated from a matrix.
- d represents the dimensionality of the space. Since we're dealing with bivariate Gaussian distributions, the dimensionality d is 2.

4. Compute the KL-Diver DKL(p || q). Which one is



$D_{KL}(q \,\|\, p) = ½[\|\mu_1\|^2 + \mathrm{Trace}(\Sigma_1) - \log|\Sigma_1| - d] / 2$

$= [1 + 4.04 - \log(0.16) - 2]/2 = 1.52 - \log(0.4)$

$D_{KL}(p \,\|\, q) = [(\mu_1-\mu_2)^T\Sigma_1^{-1}(\mu_1-\mu_2) + \mathrm{Trace}(\Sigma_1^{-1}\Sigma_2) + \log|\Sigma_1| - \log|\Sigma_2| - d] / 2$

$= [25 + 25.25 + \log(0.16) - \log(1) - 2] / 2 = 24.125 + \log(0.4)$

The second one is much larger, because there are places where *p* is large but *q* is very close to zero.

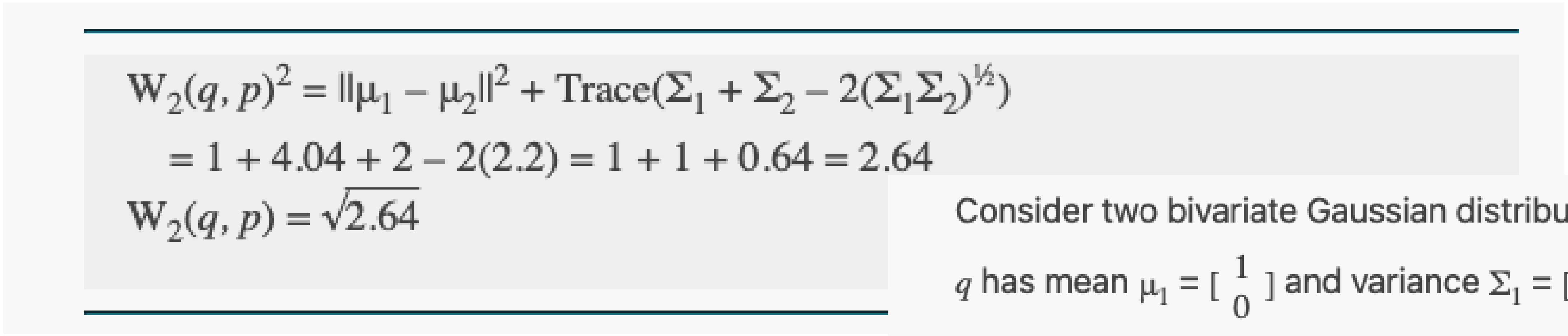
1. When computing KL divergence, we are concerned with the relative entropy (information loss) between two distributions. When we compute DKL(q||p), we are trying to understand how much information there would be if we approximated q by p (that is, assuming our data was generated by p when it was actually generated by q) loss. In this case, we are more concerned with the properties of q than with the difference between p and q.
2. The same logic applies to the mean μ. We are not directly concerned with the difference between μ1 and μ2, but with how to minimize the information loss when approximating q with p. Therefore, μ1 is not directly reflected in this formula. However, the difference between μ1 and μ2 becomes important when we think in the other direction, namely DKL(p||q), since then we are concerned with how to approximate p with q as best as possible .

c. Compute the Wasserstein Distance W2(q, p)

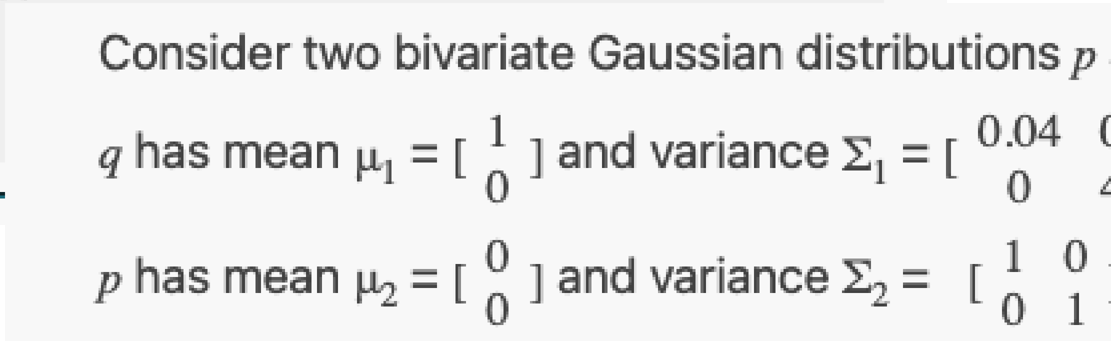
$$W_2(q, p)^2 = \|\mu_1 - \mu_2\|^2 + \mathrm{Trace}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1\Sigma_2)^{½})$$

$$= 1 + 4.04 + 2 - 2(2.2) = 1 + 1 + 0.64 = 2.64$$

$$W_2(q, p) = \sqrt{2.64}$$



$$W_2(q, p)^2 = \|\mu_1 - \mu_2\|^2 + \text{Trace}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1\Sigma_2)^{½})$$
$$= 1 + 4.04 + 2 - 2(2.2) = 1 + 1 + 0.64 = 2.64$$
$$W_2(q, p) = \sqrt{2.64}$$



Consider two bivariate Gaussian distributions p

q has mean $\mu_1 = [\begin{smallmatrix}1 \\ 0\end{smallmatrix}]$ and variance $\Sigma_1 = [\begin{smallmatrix}0.04 \\ 0\end{smallmatrix}$

p has mean $\mu_2 = [\begin{smallmatrix}0 \\ 0\end{smallmatrix}]$ and variance $\Sigma_2 = [\begin{smallmatrix}1 & 0 \\ 0 & 1\end{smallmatrix}$

1. μ1 and μ2 are the mean vectors of the two distributions.
2. ||μ1 – μ2||^2 is the square of the Euclidean distance between these two mean vectors, resulting in 1.
3. Σ1 and Σ2 are the covariance matrices of the two distributions. In this formula, Trace(Σ1 + Σ2 – 2(Σ1Σ2)^½) calculates the sum of the traces (Trace) of the two covariance matrices, minus twice the square root of the product of the two covariance matrices. The result is 1 + 0.64.
4. Adding the above two parts together gives W2(q, p)^2 = 2.64.
5. Finally, what we want is W2(q, p), the square root of the distance, so W2(q, p) = √2.64.

4. **Simple Gradient Descent by Hand**

Consider the simplest possible machine learning task:

> Solve $f(x) = wx$ such that $f(1) = 1$, i.e. $f(x) = t$, for $x = 1$, $t = 1$.

We can solve this by gradient descent using the loss function $E = ½(wx - t)^2$, with learning rate $\eta = 0.5$ and initial value $w = 0$.

a. Perform the first epoch of training by completing this table:

$w = 0$

$x = 1$

$f(x) = ?$

$E = ?$

$\partial E/\partial w = ?$

$w \leftarrow w - ? = ?$

$w = 0$

$x = 1$

$f(x) = wx = 0$

$E = ½(wx - 1)^2 = 0.5$

$\partial E/\partial w = (w.1 - 1) = -1$

$w \leftarrow w - 0.5(-1) = 0.5$

---

b. Repeat these calculations for the second epoch.

$w = 0.5$

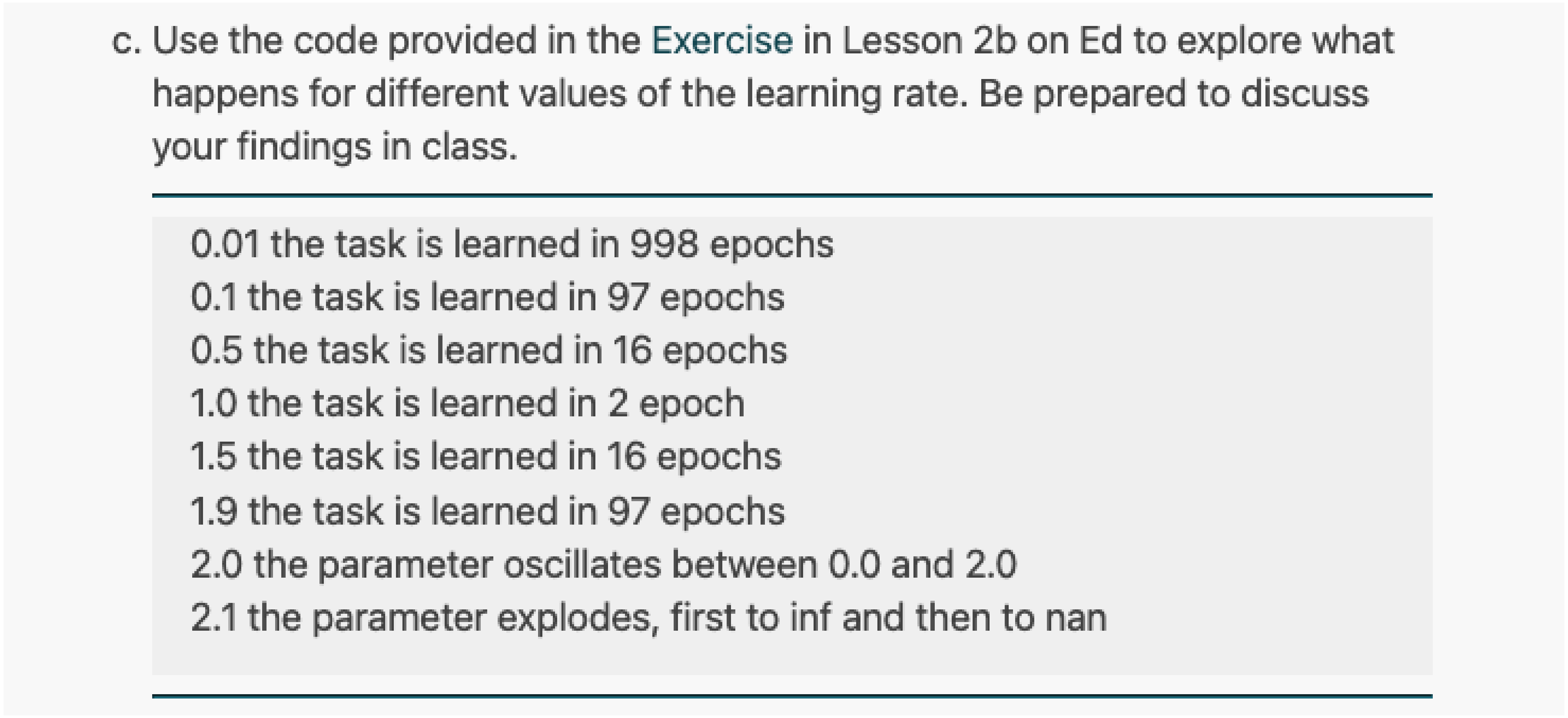
$x = 1$

$f(x) = wx = 0.5$

$E = ½(wx - 1)^2 = 0.125$

$\partial E/\partial w = (w.1 - 1) = -0.5$

$w \leftarrow w - 0.5(-0.5) = 0.75$



c. Use the code provided in the Exercise in Lesson 2b on Ed to explore what happens for different values of the learning rate. Be prepared to discuss your findings in class.

0.01 the task is learned in 998 epochs
0.1 the task is learned in 97 epochs
0.5 the task is learned in 16 epochs
1.0 the task is learned in 2 epoch
1.5 the task is learned in 16 epochs
1.9 the task is learned in 97 epochs
2.0 the parameter oscillates between 0.0 and 2.0
2.1 the parameter explodes, first to inf and then to nan

The behavior of the gradient descent algorithm under different learning rates:

1. Small learning rates (like 0.01) make the algorithm converge slowly. This means that it takes many epochs (998 in this case) to reach the optimal value. This is because the updates to the parameter `w` are very small at each step.
2. As the learning rate increases, the algorithm converges more quickly. This is because larger updates are made to the parameter `w` at each step. For a learning rate of 0.5, the algorithm converges in just 16 epochs, and for a learning rate of 1.0, it converges in only 2 epochs.
3. However, if the learning rate is too high, the algorithm may fail to converge. This is evident for learning rates of 1.5 and 1.9, where it takes many epochs to converge, and for a learning rate of 2.0, where the parameter `w` simply oscillates between 0.0 and 2.0 without ever settling on the optimal value.
4. Learning rates above a certain threshold (like 2.1 in this case) can even cause the algorithm to "explode", where the parameter `w` rapidly grows to infinity and then becomes undefined (nan, or "not a number").

In general, choosing an appropriate learning rate: too small, and the algorithm may be slow and inefficient. Too large, and it may fail to converge, or even diverge entirely.

Week 4 - Tutorial

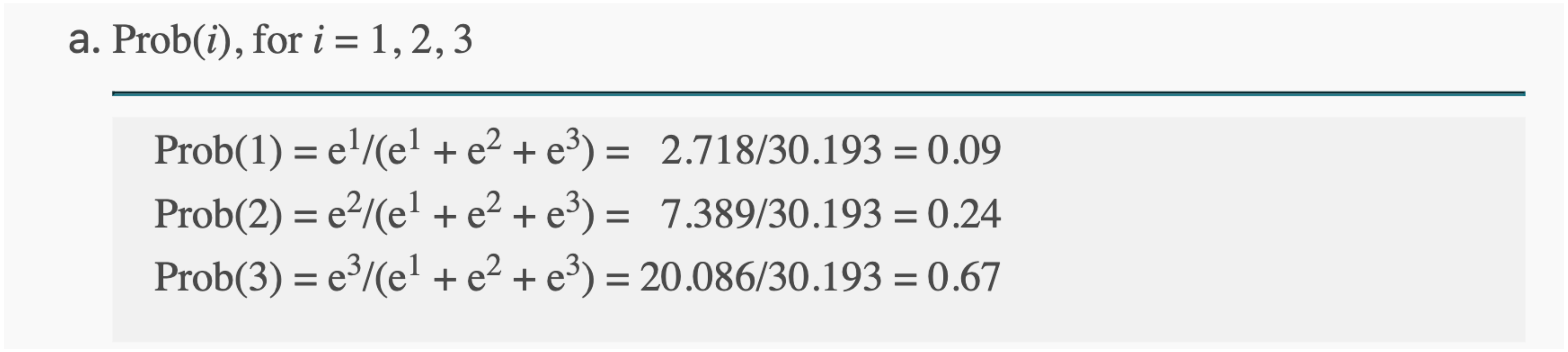
Q1. Softmax

Prob(i) = $\exp(z_i) / \Sigma_j \exp(z_j)$

Consider a neural network being trained on a classification task with three classes 1, 2, 3. When the network is presented with a particular input, the output values are:

$z_1 = 1.0$, $z_2 = 2.0$, $z_3 = 3.0$

Suppose the correct class for this input is Class 2. Compute the following, to two decimal places:



a. Prob(i), for $i = 1, 2, 3$

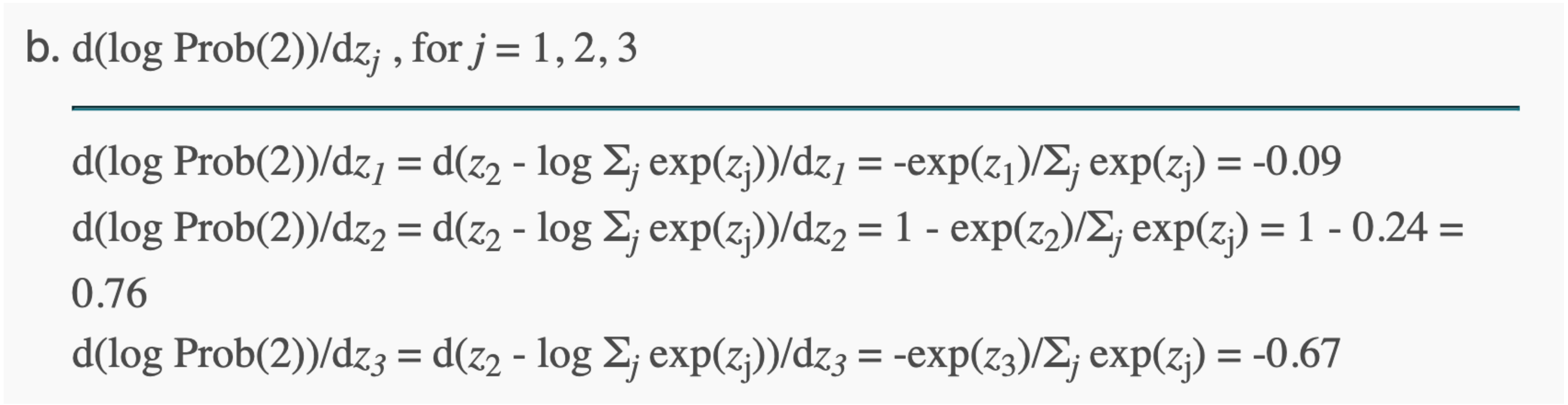
Prob(1) = $e^1/(e^1 + e^2 + e^3) = 2.718/30.193 = 0.09$

Prob(2) = $e^2/(e^1 + e^2 + e^3) = 7.389/30.193 = 0.24$

Prob(3) = $e^3/(e^1 + e^2 + e^3) = 20.086/30.193 = 0.67$

Q2. Partial derivative: d(log Prob(2))/dz_j , for $j = 1, 2, 3$

d(log Prob(2))/dzj represents the partial derivative of the natural logarithm of the probability of class 2 with respect to the score zj of class j.



b. d(log Prob(2))/dz_j , for $j = 1, 2, 3$

d(log Prob(2))/dz_1 = d(z_2 - log Σ_j exp(z_j))/dz_1 = -exp(z_1)/Σ_j exp(z_j) = -0.09

d(log Prob(2))/dz_2 = d(z_2 - log Σ_j exp(z_j))/dz_2 = 1 - exp(z_2)/Σ_j exp(z_j) = 1 - 0.24 = 0.76

d(log Prob(2))/dz_3 = d(z_2 - log Σ_j exp(z_j))/dz_3 = -exp(z_3)/Σ_j exp(z_j) = -0.67

Q2. Partial derivative: $d(\log \text{Prob}(2))/dz_j$, for $j = 1, 2, 3$

(1). $d(\log \text{Prob}(2))/dz_1$
$= d(z_2 - \log \Sigma_j \exp(z_j))/dz_1$
$= -\exp(z_1)/\Sigma_j \exp(z_j)$
$= -0.09$

$d(\log \text{Prob}(2))/dz_2 = d(z_2 - \log \Sigma_j \exp(z_j))/dz_2 = 1 - \exp(z_2)/\Sigma_j \exp(z_j) = 1 - 0.24 = 0.76$
$d(\log \text{Prob}(2))/dz_3 = d(z_2 - \log \Sigma_j \exp(z_j))/dz_3 = -\exp(z_3)/\Sigma_j \exp(z_j) = -0.67$

Let's break this down:
$\text{Prob}(i) = \exp(z_i) / \Sigma_j \exp(z_j)$, j = 1, 2, 3
$\text{Prob}(2) = e^{z2}/(e^{z1} + e^{z2} + e^{z3})$, then: natural log of both sides:
$\log(\text{Prob}(2)) = z2 - \log(e^{z1} + e^{z2} + e^{z3})$

Then, partial derivative:
$d(\log \text{Prob}(2))/dz1 = d(\ z2 - \log(e^{z1} + e^{z2} + e^{z3})\ ) /dz1 =$
$dz2 /dz1 – d(\log(e^{z1} + e^{z2} + e^{z3})\ )/dz1 = 0 - d(\log(e^{z1} + e^{z2} + e^{z3})\ )/dz1$

The derivative of z2 with respect to z1 is zero, since z2 does not involve z1.

$- d(\log(e^{z1} + e^{z2} + e^{z3})\ )/dz1 = - d(\log(u))\ / dz1$, $u = e^{z1} + e^{z2} + e^{z3}$

By the chain rule of differentiation: **$(d\ [f(g(x))]\ )/ dx = f'(g(x)) * g'(x)$**, Therefore,

$= - d(\log(u))/d(u) * d(u) /dz1$
The derivative of log(u) with respect to u is -1/u, the derivative of u with respect to z1, is $d(u)/dz1 = d\ (e^{z1} + e^{z2} + e^{z3})\ /d\ z1 =\ e^{z1}$,

Then:
The derivative of -log(ez1 + ez2 + ez3) with respect to z1 is $-1/u * e^{z1} = - e^{z1} / (e^{z1} + e^{z2} + e^{z3})$.

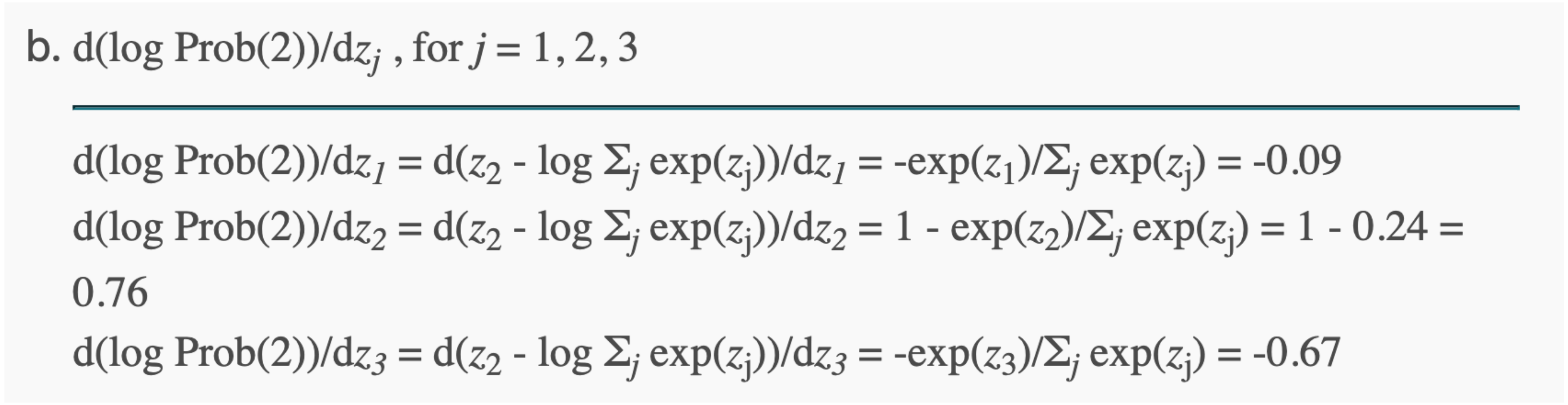
Q2. Partial derivative: d(log Prob(2))/dz_j , for $j = 1, 2, 3$

(1). d(log Prob(2))/dz_1
= d(z_2 - log Σ_j exp(z_j))/dz_1
= -exp(z_1)/Σ_j exp(z_j)
= -0.09

(2) d(log Prob(2))/dz2
= d(z2 - log Σj exp(zj))/dz2
= 1 - exp(z2)/Σj exp(zj) = 1 - 0.24 = 0.76

(3) d(log Prob(2))/dz3
= d(z2 - log Σj exp(zj))/dz3
= -exp(z3)/Σj exp(zj) = -0.67

2. d(log Prob(2))/dz_j , for $j = 1, 2, 3$,



b. d(log Prob(2))/dz_j , for $j = 1, 2, 3$

d(log Prob(2))/dz_1 = d(z_2 - log Σ_j exp(z_j))/dz_1 = -exp(z_1)/Σ_j exp(z_j) = -0.09

d(log Prob(2))/dz_2 = d(z_2 - log Σ_j exp(z_j))/dz_2 = 1 - exp(z_2)/Σ_j exp(z_j) = 1 - 0.24 = 0.76

d(log Prob(2))/dz_3 = d(z_2 - log Σ_j exp(z_j))/dz_3 = -exp(z_3)/Σ_j exp(z_j) = -0.67

Note how in each case, the partial derivative is proportional to the difference between the estimated probability and its target value (1 for the correct class; 0 for the incorrect classes).

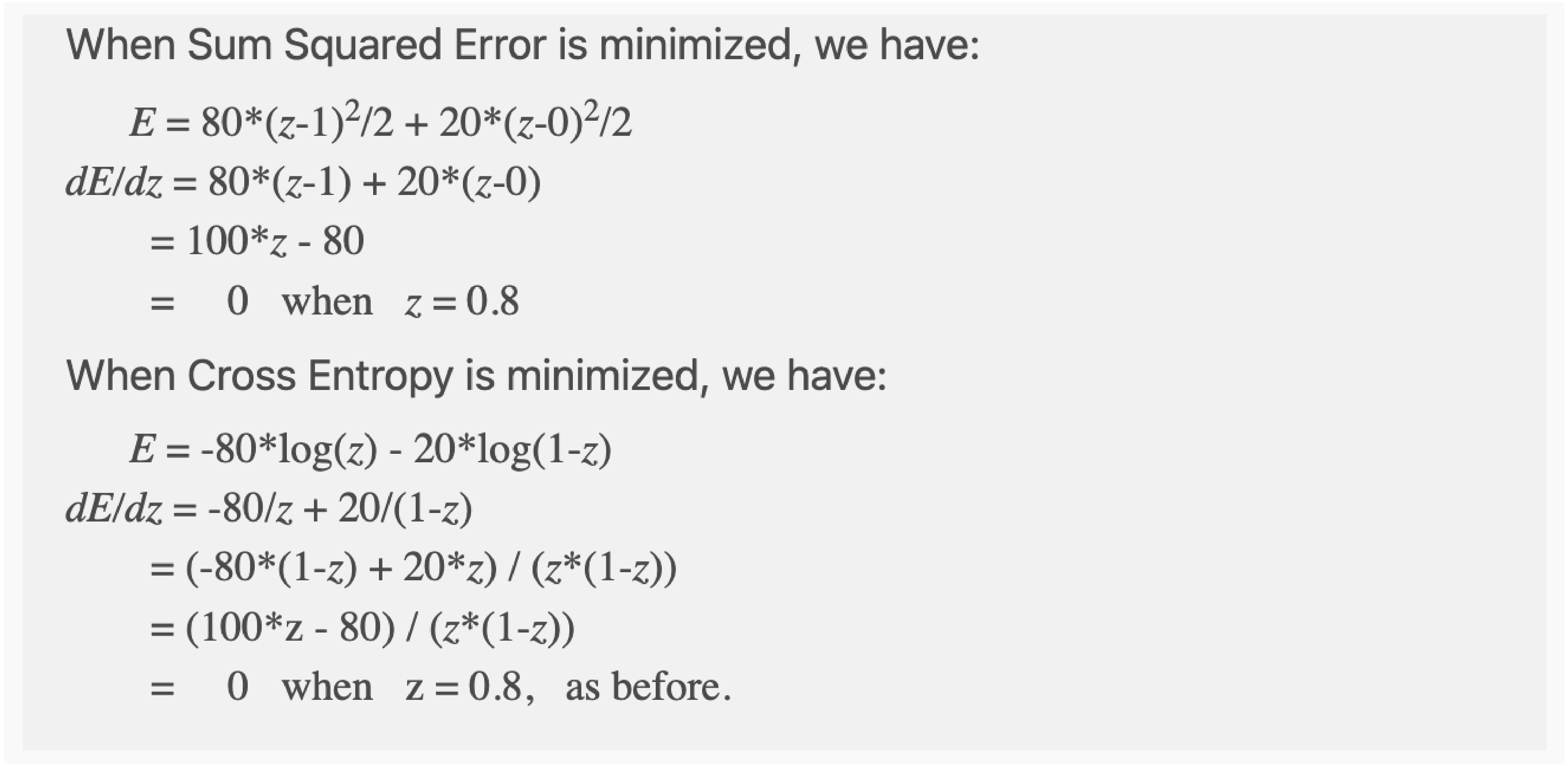
Q2. Identical Inputs

Consider a degenerate case where the training set consists of just a single input, repeated 100 times. In 80 of the 100 cases, the target output value is 1; in the other 20, it is 0. What will a back-propagation neural network predict for this example, assuming that it has been trained and reaches a global optimum? If the loss function is changed from Sum Squared Error to Cross Entropy, does it give the same result? (Hint: to find the global optimum, differentiate the loss function and set to zero.)

Q2. Identical Inputs

Consider a degenerate case where the training set consists of just a single input, repeated 100 times. In 80 of the 100 cases, the target output value is 1; in the other 20, it is 0. What will a back-propagation neural network predict for this example, assuming that it has been trained and reaches a global optimum? If the loss function is changed from Sum Squared Error to Cross Entropy, does it give the same result? (Hint: to find the global optimum, differentiate the loss function and set to zero.)

Q2. Identical Inputs



When Sum Squared Error is minimized, we have:

$$E = 80*(z-1)^2/2 + 20*(z-0)^2/2$$
$$dE/dz = 80*(z-1) + 20*(z-0)$$
$$= 100*z - 80$$
$$= \quad 0 \quad \text{when} \quad z = 0.8$$

When Cross Entropy is minimized, we have:

$$E = -80*\log(z) - 20*\log(1-z)$$
$$dE/dz = -80/z + 20/(1-z)$$
$$= (-80*(1-z) + 20*z) \, / \, (z*(1-z))$$
$$= (100*\mathrm{z} - 80) \, / \, (z*(1-z))$$
$$= \quad 0 \quad \text{when} \quad \mathrm{z} = 0.8, \quad \text{as before.}$$

Q2. Sum Squared Error

Loss Functions

In Week 1 we introduced the **sum squared error** (**SSE**) loss function, which is suitable for function approximation tasks.

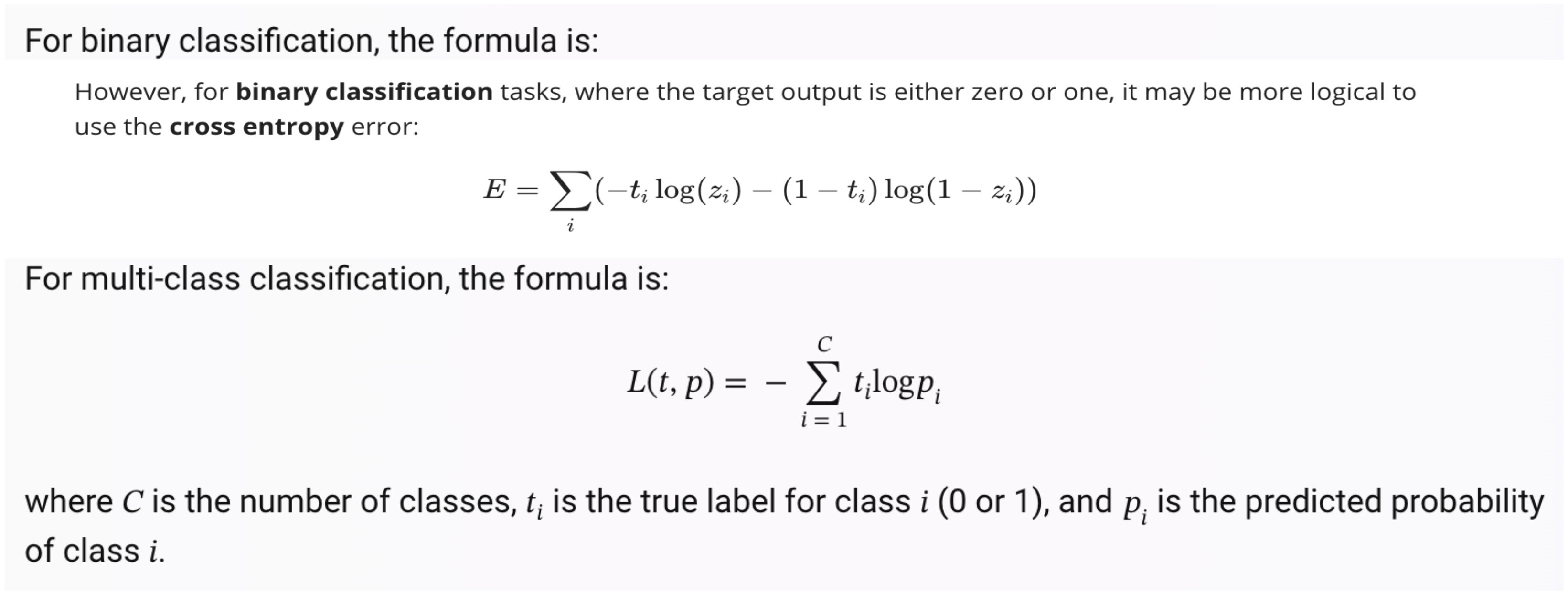
$$E = \frac{1}{2}\sum_i (t_i - z_i)^2$$

When Sum Squared Error is minimized, we have:

$$E = 80*(z-1)^2/2 + 20*(z-0)^2/2$$

$$\begin{aligned} dE/dz &= 80*(z-1) + 20*(z-0) \\ &= 100*z - 80 \\ &= 0 \quad \text{when} \quad z = 0.8 \end{aligned}$$

Q2. Cross-Entropy Loss



For binary classification, the formula is:

However, for **binary classification** tasks, where the target output is either zero or one, it may be more logical to use the **cross entropy** error:

$$E = \sum_i (-t_i \log(z_i) - (1 - t_i) \log(1 - z_i))$$

For multi-class classification, the formula is:

$$L(t, p) = -\sum_{i=1}^{C} t_i \log p_i$$

where C is the number of classes, t_i is the true label for class i (0 or 1), and p_i is the predicted probability of class i.

When Cross Entropy is minimized, we have:

E = -80*log(z) - 20*log(1-z)

dE/dz = -80/z + 20/(1-z)

= (-80*(1-z) + 20*z) / (z*(1-z))

= (100*z - 80) / (z*(1-z))

= 0 when z = 0.8, as before.

Q3. Hidden Unit Dynamics

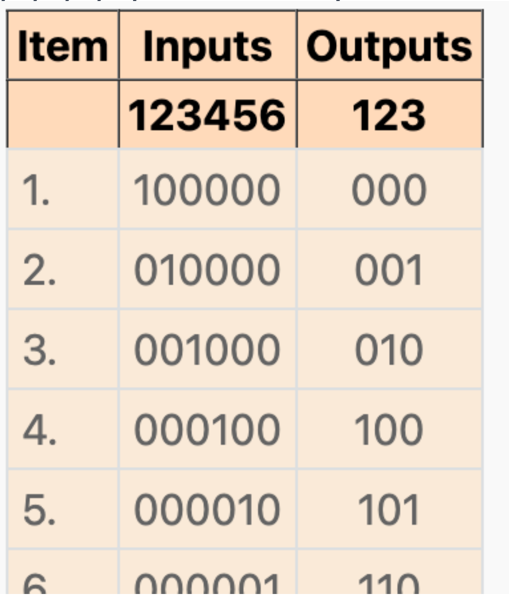
Consider a fully connected feedforward neural network with 6 inputs, 2 hidden units and 3 outputs, using tanh activation at the hidden units and sigmoid at the outputs. Suppose this network is trained on the following data, and that the training is successful.

Draw a diagram showing:

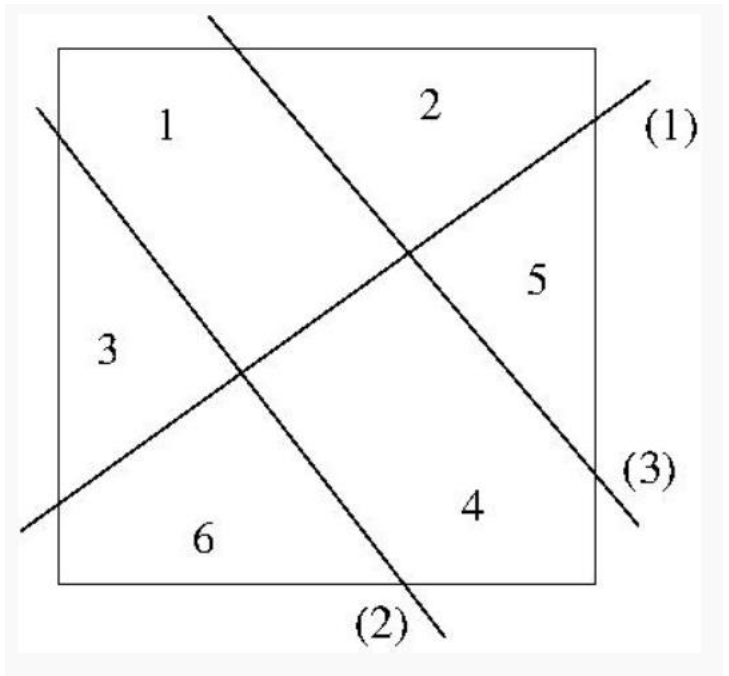
a.for each input, a point in hidden unit space corresponding to that input, and

b.for each output, a line dividing the hidden unit space into regions for which the value of that output is greater/less than one half.

c., the input is [1,0,0,0,0,0] and the expected output is [0,0,0].



Item	Inputs	Outputs
	123456	123
1.	100000	000
2.	010000	001
3.	001000	010
4.	000100	100
5.	000010	101
6.	000001	110



Q3. Input points in hidden space, output decision boundaries

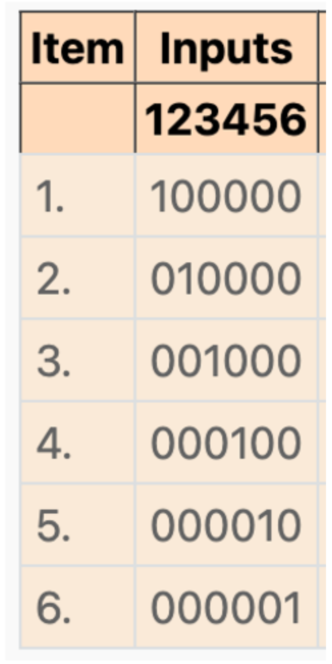
First, let's analyze the inputs and outputs. From the dataset, we can see that each input is a 6-dimensional vector, where only one element is 1 and the rest are 0. The output is a 3-dimensional vector, where the value of each element is based on the position of the input. Assuming the network has been successfully trained on this dataset, our goal is to find the point in the hidden unit space corresponding to each input and the decision boundary in the hidden unit space corresponding to each output.

1.Points of each input in the hidden unit space:
For each input, we pass it to the hidden layer of the network. After the application of weights and the tanh activation function, we get a two-dimensional output, which is the point in the hidden unit space. For example, the input "100000" might be mapped to point A in the hidden space, and the input "010000" might be mapped to point B. The exact coordinates depend on the weights and bias parameters of the network.

2. Decision boundaries of each output in the hidden unit space:
The output layer is obtained by processing the output of the hidden layer through weights and a sigmoid activation function. The output range of the sigmoid function is (0,1), and we want to find a decision boundary such that the sigmoid value corresponding to the hidden unit output on one side of this line is greater than 0.5, and on the other side it's less than 0.5. This line divides the hidden unit space into two parts, corresponding to the "activation" and "non-activation" of the network for a specific output.

Input: Points of each input in the hidden unit space

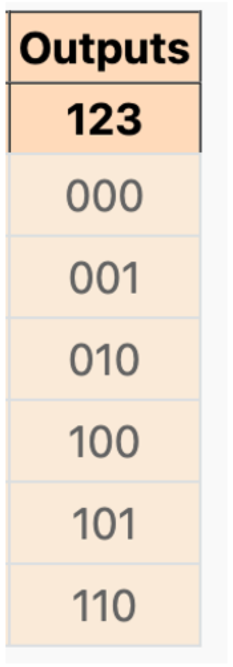
Our model has 6 inputs, 2 hidden neurons, and 3 outputs. We use tanh as the activation function for the hidden layer and sigmoid as the activation function for the output layer. When calculating the output of the neural network's hidden layer, what we do is take the dot product of the input and the weights (i.e., multiply the corresponding elements and then add them up), and then add the bias.
In this network, we first define the weight matrix and bias from the input layer to the hidden layer. Suppose our weight matrix is as follows:



Item	Inputs
	123456
1.	100000
2.	010000
3.	001000
4.	000100
5.	000010
6.	000001

```
W1 = [[0.1, 0.2],
      [0.3, 0.4],
      [0.5, 0.6],
      [0.7, 0.8],
      [0.9, 1.0],
      [1.1, 1.2]]

b1 = [0.1, 0.1]
```

Hidden layer: 2 neurons



Outputs
123
000
001
010
100
101
110

For a given input, we take the dot product of it with the weight matrix (i.e., multiply the corresponding elements and then add them up), then add the bias, and finally pass the result through the activation function tanh to get the result. The two-dimensional vector obtained is the coordinates of the input in the hidden layer space. For example, our example: the two-dimensional coordinates of Item1 in the hidden layer are: 100000 x W1 + b1 = (0.2, 0.3), and then [tanh(0.2), tanh(0.3)], we get the two-dimensional coordinates of the first example in the hidden layer.

Input: 6 examples, each example, 6 dimension

Output: 6 examples, each example, 3 classes output. Class 1 for example 1’s class is 0, for class2 is 0, and class 3 is also 0 class.

Input: Points of each input in the hidden unit space

The above figures are a set of example coordinates selected to illustrate this mapping process.

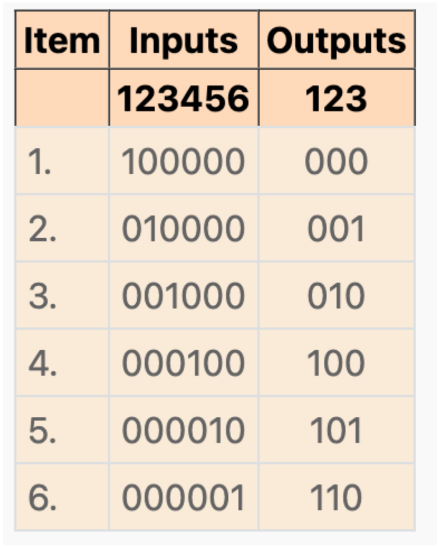
In real case, the weights and biases of our neural network are learned through the training process, which involves using an optimization algorithm (such as stochastic gradient descent) to minimize the difference between the predicted output and the actual output. After the learning is completed, the neural network can effectively map the input to the hidden space and generate predictions at the output layer.

A function to converto Tanh activation

```
import numpy as np

pre_activation_output = [0.2, 0.3]
hidden_layer_output = np.tanh(pre_activation_output)
```

Output: Decision boundaries of each output in the hidden unit space

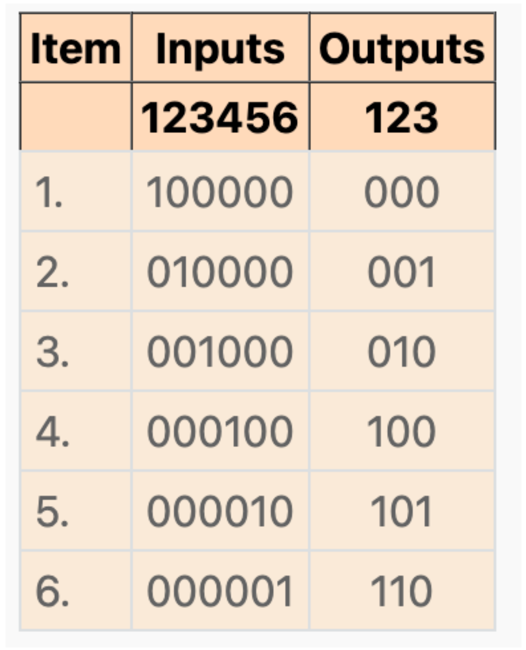
In the output layer of the neural network, we take the dot product of the output of the hidden layer with the weights, add the bias, and then pass it to the sigmoid activation function. Suppose our network only has one output (actual we have 3 output, 3 classes), then the output of the sigmoid is a value between 0 and 1, which can be interpreted as the probability of belonging to a certain class. We interpret the output of the sigmoid as greater than 0.5 as the network predicting that the input belongs to a certain class, and less than 0.5 as the network predicting that the input does not belong to this class. Therefore, we can find a line (in two-dimensional space) or a hyperplane (in higher dimensional spaces), so that all points on one side of this line or plane, after being processed by the neural network, have an output greater than 0.5, and the output of all points on the other side is less than 0.5. This line or plane is the decision boundary.



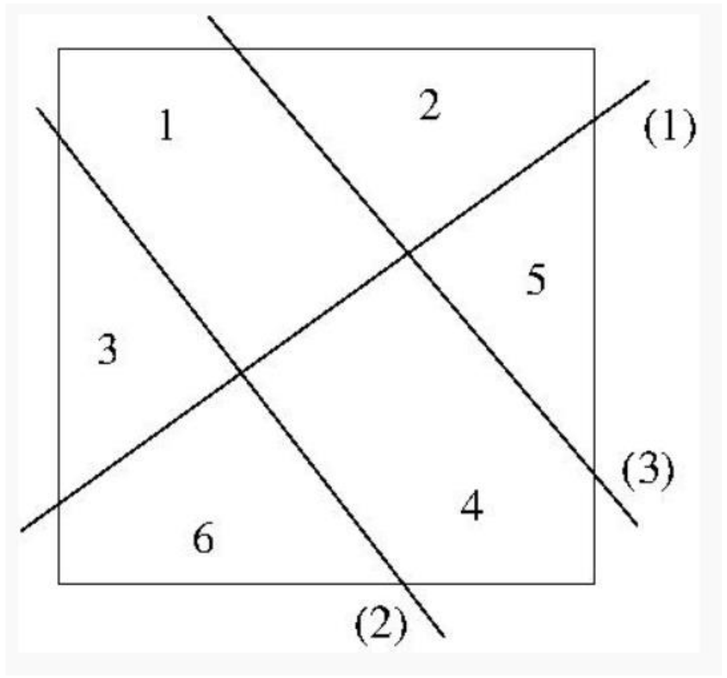
Item	Inputs	Outputs
	123456	123
1.	100000	000
2.	010000	001
3.	001000	010
4.	000100	100
5.	000010	101
6.	000001	110

The above output is the result obtained after the neural network has been successfully trained given in the question description. We have three classes in our output, where "1" indicates that the prediction probability of the corresponding class is greater than 0.5, and "0" indicates that the prediction probability of the corresponding class is less than 0.5. For example, for the input "010000", its output is "001", which indicates that the prediction probability of the third class is greater than 0.5, while the prediction probabilities of the first two classes are less than 0.5.

Output: Decision boundaries of each output



Item	Inputs	Outputs
	123456	123
1.	100000	000
2.	010000	001
3.	001000	010
4.	000100	100
5.	000010	101
6.	000001	110



A total of three lines are drawn in the figure, (1), (2), (3), because there are three classes in the output. Looking at the first column of the output, there are three points where the prediction result is 0 and the other three points where the prediction result is 1. Therefore, looking at the decision boundary (1), points 1, 2, 3 are above the decision boundary (1), and the other three points, 4, 5, 6, are below the decision boundary (1). Similarly, looking at the second column, items 3 and 6 belong to category 1, and the other four items belong to category 0. Looking at the decision boundary (2), points 3, 6 are below the decision boundary (2), less than 0.5 belong to one category, and items 1, 2, 5, 4, are above the decision boundary (2), greater than 0.5 belong to one category.

Q4. Linear Transfer Functions

Suppose you had a neural network with linear transfer functions. That is, for each unit the activation is some constant c times the weighted sum of the inputs.
a.Assume that the network has one hidden layer. We can write the weights from the input to the hidden layer as a matrix WHI, the weights from the hidden to output layer as WOH, and the bias at the hidden and output layer as vectors bH and bO.

Using matrix notation, write down equations for the value O of the units in the output layer as a function of these weights and biases, and the input I. Show that, for any given assignment of values to these weights and biases, there is a simpler network with no hidden layer that computes the same function.

Using vector and matrix multiplication, the hidden activations can be written as

$$\mathbf{H} = c * (\mathbf{b}^{H} + \mathbf{W}^{HI} * \mathbf{I})$$

The output activations can be written as

$$\begin{aligned}\mathbf{O} &= c * [\mathbf{b}^{O} + \mathbf{W}^{OH} * \mathbf{H}] \\ &= c * [\mathbf{b}^{O} + \mathbf{W}^{OH} * c * (\mathbf{b}^{H} + \mathbf{W}^{HI} * \mathbf{I})] \\ &= c * [(\mathbf{b}^{O} + \mathbf{W}^{OH} * c * \mathbf{b}^{H}) + (\mathbf{W}^{OH} * c * \mathbf{W}^{HI}) * \mathbf{I}]\end{aligned}$$

Due to the associativity of matrix multiplication, this can be written as

$$\mathbf{O} = c * (\mathbf{b}^{OI} + \mathbf{W}^{OI} * \mathbf{I})$$

where

$$\mathbf{b}^{OI} = \mathbf{b}^{O} + \mathbf{W}^{OH} * c * \mathbf{b}^{H}$$
$$\mathbf{W}^{OI} = \mathbf{W}^{OH} * c * \mathbf{W}^{HI}$$

Therefore, the same function can be computed by a simpler network, with no hidden layer, using the weights $\mathbf{W}^{OI}$ and bias $\mathbf{b}^{OI}$.

假设该网络有一个隐藏层。我们可以将输入到隐藏层的权重写为矩阵WHI，将隐藏层到输出层的权重写为WOH，将隐藏层和输出层的偏置写为向量bH和bO。使用矩阵表示法，根据这些权重和偏置以及输入I，写出输出层单元值O的方程式。证明，对于这些权重和偏置的任何给定赋值，都存在一个没有隐藏层的更简单的网络可以计算相同的函数。

使用向量和矩阵乘法，隐藏激活可以写为
$H = c * (bH + WHI * I)$

输出激活可以写为
$O = c * [bO + WOH * H]$
$= c * [bO + WOH * c * (bH + WHI * I)]$
$= c * [(bO + WOH * c * bH) + (WOH * c * WHI) * I]$

由于矩阵乘法的结合性，这可以写为
$O = c * (bOI + WOI * I)$

其中
$bOI = bO + WOH * c * bH$
$WOI = WOH * c * WHI$

因此，相同的函数可以由一个更简单的网络计算，该网络没有隐藏层，使用权重WOI和偏置bOI。

b. Repeat the calculation in part (a), this time for a network with any number of hidden layers. What can you say about the usefulness of linear transfer functions?

By removing the layers one at a time as above, a simpler network with no hidden layer can be constructed which computes exactly the same function as the original multi-layer network. In other words, with linear activation functions, you don't get any benefit from having more than one layer.

Email:
JINGYING.GAO@UNSW.EUD.AU

Assignment Question Email:
cs3411@cse.unsw.edu.au