ECS708: *Machine Learning* Lab Assignment 1 Part 2: REPORT

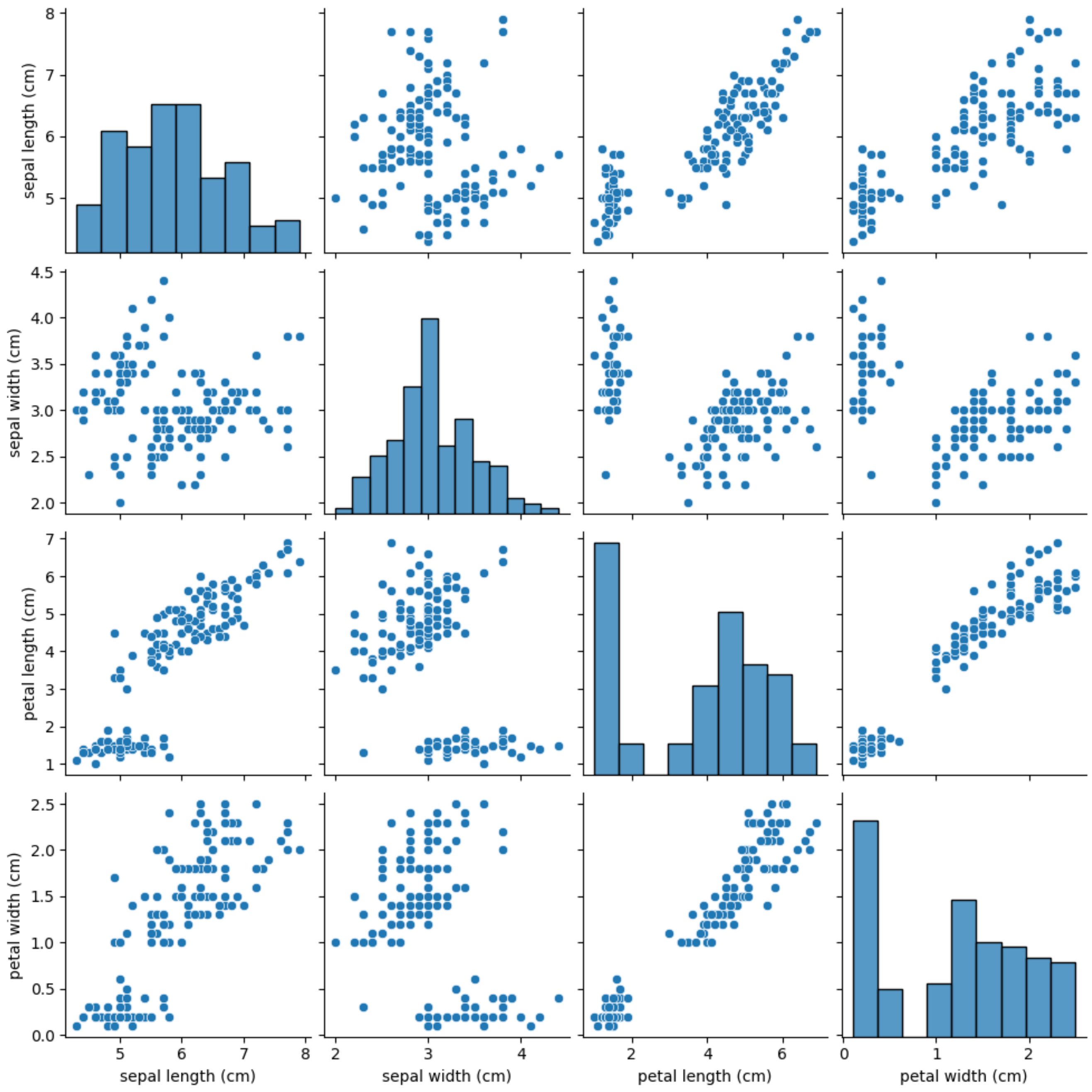
Assignment details

- **Assignment name**: Assignment 1 Part 2 (A & B)
- **Course name**: Machine learning
- **Course code**: ECS708U/ECS708P
- **Student name**: Pranav Narendra Gopalkrishna
- **Student number**: 231052045

PART A:
Linear classification & logistic regression

Introduction

The aim of this lab is to get familiar with **classification problems** and **logistic regression**. We will be using some code extracts that were implemented last week and build a logistic regression model. For this lab, we will be using the iris dataset. Below is a brief look at the data...



The scatterplots give a sense of the relationship or at least correlation between each pair of variables, whereas the histograms depict the respective variable's distribution. We can see that the relationships between the feature variables and the distribution of each variable is quite varied.

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4
6	4.6	3.4	1.4	0.3
7	5.0	3.4	1.5	0.2
8	4.4	2.9	1.4	0.2
9	4.9	3.1	1.5	0.1

To help detect and prevent overfitting, we will split the data into train and test sets. For consistency and to allow for meaningful comparison the same splits are maintained in the remainder of this part. For convenience, let M_i be the number of observations in dataset i and let N be the number of features. We have the following PyTorch tensors:

- `x_train` : An $M_{tr} \times N$ matrix where each column denotes the values of a particular feature & each row denotes a particular observation
- `x_test` : An $M_{te} \times N$ matrix with the same arrangement as `x_train`
- `y_train` : An $M_{tr} \times 1$ column vector of the target values corresponding to each observation of the training set
-
    - `y_train` : An $M_{te} \times 1$ column vector of the target values corresponding to each observation of the testing set

Also note that, for the same of improving our model's performance, i.e. for reducing the need to scale and shift weights and biases to match the scale and mean of the dataset, we normalise the features (which may have features with different scales and shifts to each other). So, `x_train` and `x_test` are normalised (both with respect to the mean and standard deviation of `x_train`). By inspecting the dataset we see that it contains four attributes:

- `sepal length`
- `sepal width`
- `petal length`
- `petal width`

We also observe that the target variable takes on three possible integer values which are not quantitative but nominal in nature:

- 0 means setosa
- 1 means versicolor

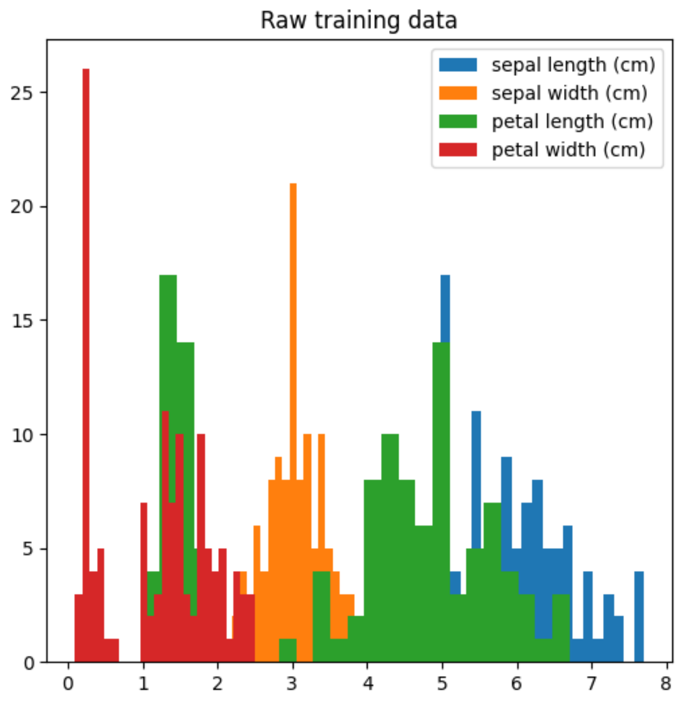
- 2 means virginica

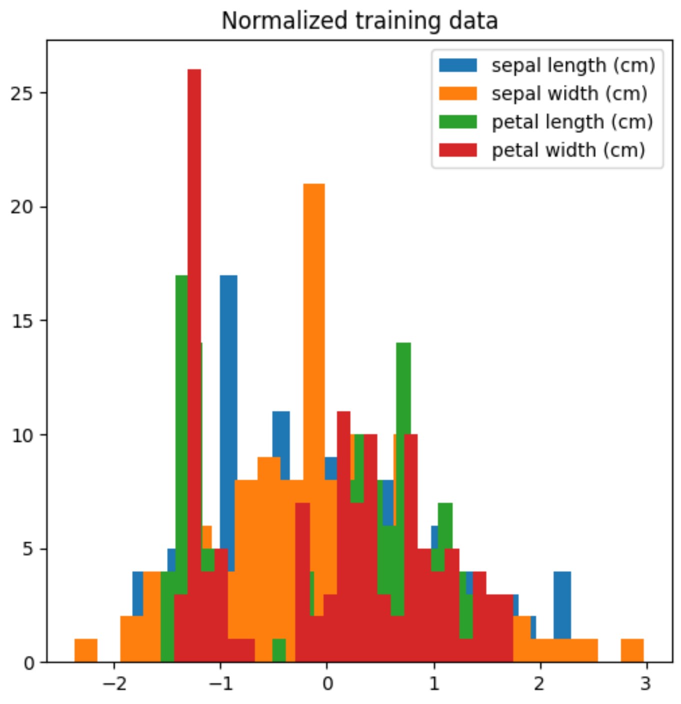
1. Single class

Question 1

We again notice that the attributes are on different scales. Use the normalisation method from last lab, to standardize the scales of each attribute on both sets. Plot the normalized and raw training sets; what do you observe? [2 marks]

Plotting normalized and raw (original) training data separately...



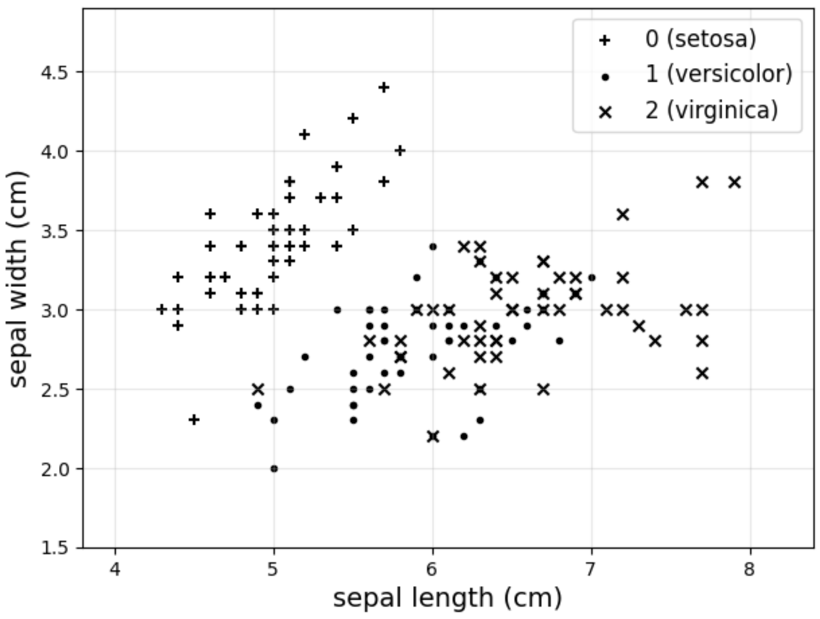


OBSERVATIONS:
Plotted above are the histograms of each feature drawn together to see how the values of the features are distributed with respect to each other, A clear difference between the unnormalised data and the normalised data is the distribution of the values. In the normalised version, the values are more packed together; indeed, they all revolve more closely around the mean 0. But we also note that normalisation does not change the way values are distributed within a feature; normalisation only brings all feature values to the same centre and scale (mean and variance).

Hence, seeing the need, we normalise the training & test feature data for future work...

By inspecting the dataset we see that it contains 4 attributes. (`sepal length` , `sepal width` , `petal length` , `petal width` , in centimeters). For simplicity we will focus on the first two.

Plotting categories with respect to sepal length and sepal width...



As there are multiple classes which are not all easily separable from each other, we will focus for now on class 0 (setosa). Henceforth, for the rest of this part of the assignment, we modify the `y_train` and `y_test` tensors so that each label is 1 if the class is setosa and 0 if otherwise.

Sigmoid function

With logistic regression the values we want to predict are now discrete classes, not continuous variables. In other words, logistic regression is for classification tasks. In the binary classification problem we have classes 0 and 1, ex. classifying email as spam or not spam based on words used in the email. The logistic/sigmoid function given by the formula below:

$h_\theta(x) = g(x\theta^T) = \frac{1}{1+e^{-\theta^T x}}$

where

- θ is the row vector of weights
- x is the row vector of inputs in a single observation
- $x\theta^T$ is essentially a linear combination, i.e. a weighted sum of the elements of x

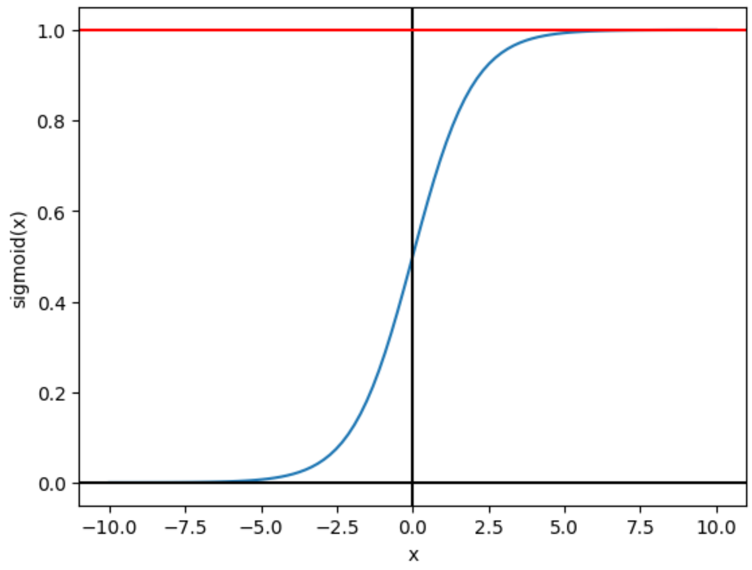
Questions 2 & 3

Q2. First implement the above function in `def sigmoid()` . [2 marks]

Q3. Then, using the implementation of `LinearRegression` from last week as guideline, create a custom pytorch layer for `LogisticRegression` [2 marks]

The implementation for these questions is available in the notebook. However, we shall look at some useful details for the following questions.

Sigmoid function visualisation



It can be considered as a probabilistic/continuous version of the step function, and using threshold values and the right weights, it can be used to perform binary classification, which is the basic goal of logistic regression.

Cost function for logistic regression

To evaluate the cost of the regression model (hence understand how the weights must be updated), we must define an appropriate cost function. The cost function we will use for logistic regression is the **cross entropy loss**, which is given by the form:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))]$$

Which when taking partial derivatives with respect to the vector of weights θ, we get:

$$\frac{\delta J}{\delta \theta} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Putting this into the gradient descent update equation gives:

$$\theta_j = \theta_j - \alpha\frac{\delta J}{\delta \theta} = \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

NOTE: To evaluate performance with respect to **cross entropy loss** does not require a division by m (i.e. number of observations), but such a division is helpful when comparing the costs of two datasets on the same variables but with different numbers of observations (ex. it is useful in comparing the training and testing errors), since it brings the cost values to the same scale.

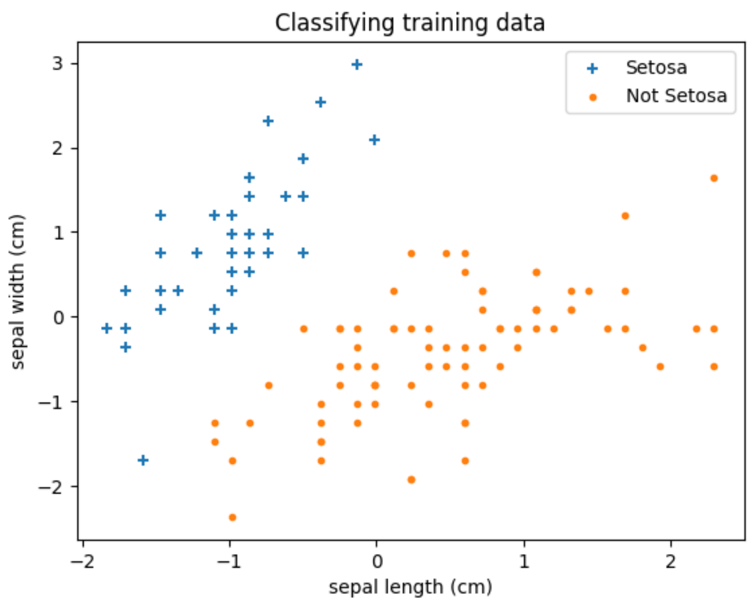
Question 5

Draw the decision boundary on the test set using the learned parameters. Is this decision boundary separating the classes? Does this match our expectations? [2 marks]

We shall approach the final answer in steps, starting with separating out classes on a plot for training data.

Plotting classes

Note that we are classifying flowers (as "Setosa" or "Not Setosa") with respect to sepal length & sepal width (the two predictor variables). Plotting scatterplot of classified points without decision boundary...



Plotting decision boundary

> KEY REFERENCE: https://medium.com/analytics-vidhya/decision-boundary-for-classifiers-an-introduction-cc67c6d3da0e

General approach

For convenience, let y denote the predicted target variable, let x_1 denote sepal length, and let x_2 denote sepal width (x_1 and x_2 are the predictor variables here). In the previous section's scatterplot, we see that the axes are the two predictor variables (x_1 is the horizontal axis, while x_2 is the vertical axis). This scatterplot is done within a finite grid.

In order visualise the decision boundary as predicted by our model, we must obtain the line going through the set of coordinates whose predicted value lies at or close to 0.5. This is because 0.5 is the decision value; any coordinate (p, q) with a y value higher than 0.5 is classified as 1 while every coordinate (p, q) with a y value lower than 0.5 is classified as 0.

To do the above, we can do the following:

- Obtain the y value for every possible coordinate (p, q) in a finite grid, where $p \in x_1, q \in x_2$
- Plot a contour line at $y = f((p, q)) = 0.5$ (where f is the model)

TERMINOLOGY: **Contour line**:
A contour line (also isoline, isopleth, or isarithm) of a function of two variables is a curve along which the function has a constant value, so that the curve joins points of equal value (source: https://en.wikipedia.org/wiki/Contour_line).

The above goals can be obtained using the following approach:

- Create a meshgrid for a range of values from the axes x_1 & x_2
- Use this meshgrid to create a tensor of predictor values (with bias)
- Apply the model tp the above tensor to obtain the predictions
- Using the predictions & the previous meshgrid, plot the contour

To do this, we need the following two functions...

1. `torch.meshgrid`

This function does the following:

- Takes an array of values x_1 of size n_1 from axis 1
- Takes an array of values x_2 of size n_2 from axis 2
- Returns a grid of all possible pairs (p, q) where $p \in x_1, q \in x_2$

Essentially, this function returns the Cartesian product of x_1 and x_2. This grid of combinations is given in the form of a tuple containing two arrays of row vectors. The nature of these arrays of row vectors depends on the indexing used (given by the argument `indexing`).

CASE 1: `indexing="ij"` **(default)**:

The first element of the tuple is the array of n_1 row vectors wherein row vector i is the array of n_2 copies of the ith element of x_1. The second element of the tuple is the array of n_1 copies of the array x_2.

CASE 1: `indexing="xy"` **(kind of the reverse of "ij")**:

The first element of the tuple is the array of n_2 copies of the array x_1. The second element of the tuple is the array of n_2 row vectors wherein row vector i is the array of n_1 copies of the ith element of x_2.

Both of these arrangements are such that matching value i from row j of the first element of the tuple to the value j from row j of the second element of the tuple gives you a pair (p, q) where $p \in x_1, q \in x_2$. Doing this for all $i \in [0, n_2]$ and all $j \in [0, n_1]$ gives you all the possible pairs of the values of x_1 and x_2.

This function can be used to generate a grid of coordinates given two arrays of values corresponding to different dimensions/axes.

> REFERENCE: https://pytorch.org/docs/stable/generated/torch.meshgrid.html

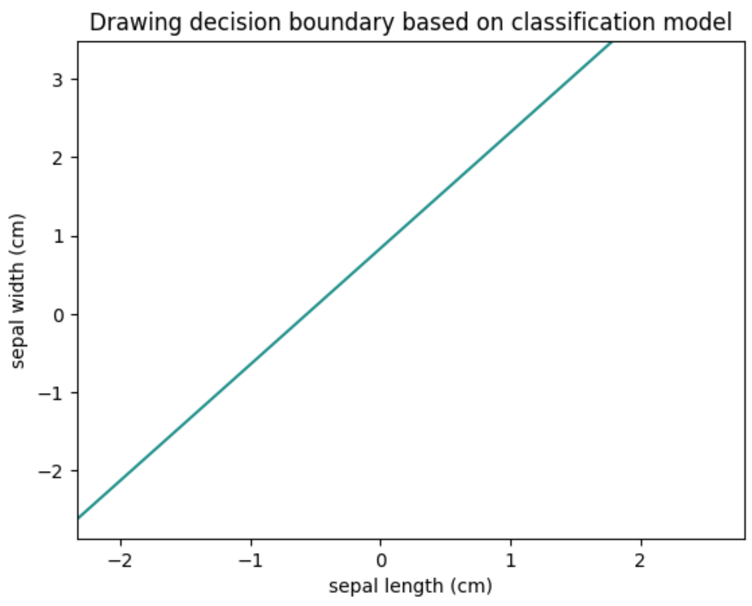
2. `matplotlib.pyplot.contour`

This function inputs:

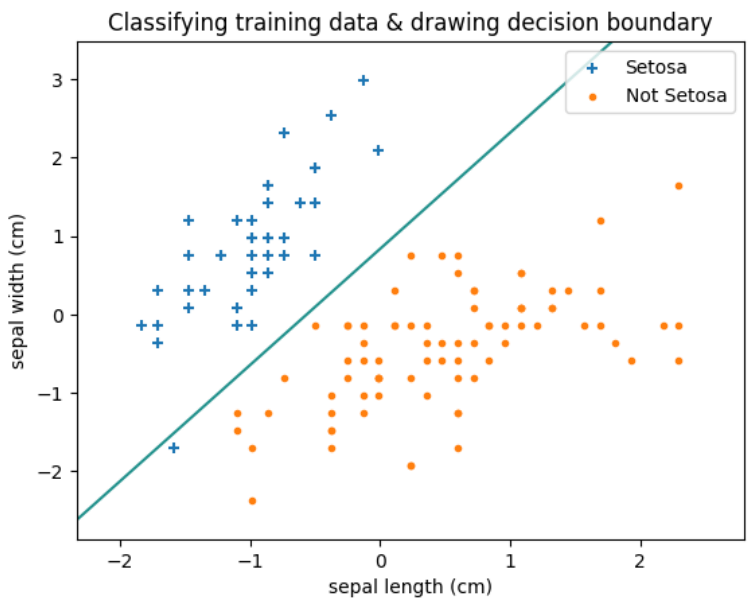
- Two lists of row vectors X & Y that together form a meshgrid
- The list of row vectors Z of functional values for each coordinate of the meshgrid

Of course, the shapes and ordering of each of the above must match, or else the contour line cannot be properly constructed. Another important (though optional) argument is `levels`, which specifies the number of contour lines to be drawn from the range, i.e. between the minimum and maximum values of Z. Given n levels, n contour lines are calculated for n functional values uniformly spread across the range of Z. Hence, `levels=1` plots the contour line for the functional value at the middle of the range. In our case, this value is 0.5 (since the target ranges from 0 to 1), which fits our purpose perfectly.

> REFERENCE: https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.contour.html



Plotting classes with decision boundary



OBSERVATIONS:
The above decision boundary almost completely (excluding one point) partitions the data (consisting of value pairs of sepal length and sepal width) according to the respective classes ("Setosa", "Not Setosa") to which they belong. The contour of the decision boundary is quite apparent even when looking at the training data points when they were only coloured by their classes. The clusters of different colours corresponding to different classes were quite apart from each other, fairly closely clumped within the cluster and well separated from the ohter cluster by some space. In other words, we see that the data here is linearly separable and each linear combination of inputs and weights does fall to either one side of a threshold value (0.5 in our case). Thus, the decision boundary is as expected.

2. Multiclass

So far, we have focused on a binary classification (is this iris setosa or not), however in this section we will address the problem as a multiclass classification. We will be using a 1 vs. all approach. We will also be using all four attributes for the classification.

RECAP: **How classes are represented in the original dataset**:
The target variable can take integer values that are nominal in nature, not representing quantities but particular labels (of flowers):

- 0 means setosa
- 1 means versicolor
- 2 means virginica

Obtaining the required representation for the target variable

First, some conceptual and implementation notes...

PROGRAMMING NOTE: **The `torch.nn.functional` module**:
This module contains numerous functions that can be used in convolutions.

About convolution

Convolution is a mathematical operation on two functions f_1 and f_2 that produces a third function $f_3 = (f_1 * f_2)$ that maps f_1 to its modification via f_2, i.e. it expresses how the shape of one is modified by the other. Note that the term convolution refers to both the result function f_3 and the process of computing it.

SIDE NOTE 1: *The term convolution comes from the latin con (with) + volutus (rolling).*

The convolution can be given as an integral of the product of two functions f_1 and f_2 wherein one function is reflected along the y-axis (i.e. its variable input's sign is reversed) and shifted along the x-axis, i.e. its variable input is added to a constant x (which is in turn the variable input of the convolution). In other terms:

$$f_3(x) = (f_1 * f_2)(x) = \int_{-\infty}^{\infty} f_1(t) f_2(x-t) dt$$

SIDE NOTE 2: *t in the integral is only an integration variable. x is the input of the convolution and is constant with respect to the integral.*
SIDE NOTE 3: *Convolution is a commutative operation, i.e.* $(f_1 * f_2) = (f_2 * f_1)$.

> REFERENCES:
>
> - https://www.ml-science.com/convolution
> - https://en.wikipedia.org/wiki/Convolution

Firstly, we need to process `y_train, y_test` so that each label is a vector rather than an integer, specifically a vector wherein all values except the index corresponding to the class are 0; the index corresponding to the class is assigned 1. This representation allows us to treat each class as the probability of being in a certain class. The labelled data's categorisation is obviously certain, so they hold either 1 (complete certainty of being in a class) and 0 (complete certainty of not being in a given class). This enables us to treat the set of multiple classes as a set of multiple binary classification problems that must be solved together to reach a final result. This representation is called "one-hot".

TERMINOLOGY NOTE: `one_hot` method from `torch.nn.functional` :
In digital circuits and machine learning, a one-hot is a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0) (source: https://en.wikipedia.org/wiki/One-hot). `torch.nn.functional.one_hot` is a method to convert the values of a categorical variable (whose n classes are represented as integers $0, 1...(n-1)$). Given that a tensor `y` is a tensor denoting such a categorical variable with n unique classes, and given that we give the optional argument `num_labels` as k, the `one_hot` function converts each value y_i of `y` into a row vector where the value at the y_ith index holds 1 and every other index holds 0. In this way, we convert each class into a one-hot.

Creating one-hot matrices from target vectors...

```
# NOTE: The following import was made: `from torch.nn import functional as F`
y_train_oneHot = F.one_hot(y_train.reshape(-1).long(), num_classes=3)
y_test_oneHot = F.one_hot(y_test.reshape(-1).long(), num_classes=3)
```

A brief look at the data representation...

```
SHAPES
Shape of y_train: torch.Size([120, 1])
Shape of y_train_oneHot: torch.Size([120, 3])
------------
UNIQUE VALUES
Unique values in y_train: tensor([0, 1, 2], dtype=torch.int32)
Unique values in y_train_oneHot: tensor([0, 1])
------------
CONTENTS
Printing some rows of y_train_oneHot:
tensor([[1, 0, 0],
        [1, 0, 0],
        [0, 1, 0]])
```

Question 6.2 (6.1. was implemnentation-based)

Using the 3 classifiers, predict the classes of the samples in the test set and show the predictions in a table. Do you observe anything interesting? [4 marks]

Code:

```
y = y_test.reshape(-1).numpy()
isSetosa = setosa.model(x_test).detach().reshape(-1).numpy()
isVersicolor = versicolor.model(x_test).detach().reshape(-1).numpy()
isVirginica = virginica.model(x_test).detach().reshape(-1).numpy()
"""
NOTE ON RESHAPING:
`.reshape()` was used to convert the variables from a column vector with multiple
rows of size 1 into a 1-D array (i.e. a row vector). This is done to pass the
data smoothly into the dataframe (where each column needs to be a 1-D array).
"""

# Obtaining data to display:
data = {}
data["class"] = y
data["isSetosa"] = isSetosa
data["isVersicolor"] = isVersicolor
data["isVirginica"] = isVirginica

# Obtaining softmax-based predicted classes:
indexWithMaxProbability = []
for i in range(len(isSetosa)):
  p = [isSetosa[i], isVersicolor[i], isVirginica[i]]
  indexWithMaxProbability.append(p.index(max(p)))

data["indexWithMaxProbability"] = indexWithMaxProbability

# Creating the dataframe:
pd.DataFrame(data=data)
```

	class	isSetosa	isVersicolor	isVirginica	indexWithMaxProbability
0	1	0.036232	0.739495	0.443046	1
1	0	0.999959	0.170995	0.109746	0
2	2	0.000003	0.839322	0.946888	2
3	1	0.038054	0.589881	0.706107	2
4	1	0.007607	0.766065	0.531834	1
5	0	0.999647	0.317734	0.083246	0
6	1	0.280010	0.549291	0.535536	1
7	2	0.001701	0.382830	0.971930	2
8	1	0.000426	0.915994	0.395699	1
9	1	0.076717	0.733305	0.373754	1
10	2	0.012826	0.347955	0.949967	2
11	0	0.999313	0.586312	0.021700	0
12	0	0.999912	0.304986	0.049055	0
13	0	0.999522	0.535209	0.025369	0
14	0	0.999985	0.129564	0.131110	0
15	1	0.153275	0.340494	0.860016	2
16	2	0.000809	0.464559	0.969108	2
17	1	0.040580	0.828239	0.251402	1
18	1	0.054235	0.669739	0.550342	1
19	2	0.000347	0.579525	0.956144	2
20	0	0.999681	0.422103	0.045644	0

	class	isSetosa	isVersicolor	isVirginica	indexWithMaxProbability
21	2	0.014712	0.475646	0.888799	2
22	0	0.999750	0.282217	0.100425	0
23	2	0.000476	0.604897	0.940778	2
24	2	0.017465	0.205937	0.977406	2
25	2	0.001025	0.431421	0.970404	2
26	2	0.000105	0.852253	0.773094	1
27	2	0.001237	0.346241	0.982149	2
28	0	0.998712	0.534586	0.040105	0
29	0	0.999337	0.502288	0.037067	0

OBSERVATIONS:

Above, we see that the class with the highest probability based on a binary 1 vs many classification (i.e., for a given class C, we reduce the set of possible classes for the data to C or *not* C) generally coincides with the actual (non-binary) class of the data. Here, we have indexed the classes with respect to their integer labels; her we see that the index with the highest probability generally coincides with the actual class label mapped to the data.

We also get some indication that this method of multi-classification may not be the completely reliable, as we see some correct predictions being correct by a small margin (i.e. a small difference in the probabilities for those classes)...

```
18 | 1  0.053554 |       0.671647          | 0.577937      | 1
```

... or wrong classification by a large margin...

```
15      | 1      0.147984         | 0.340691       | 0.863798       | 2
```

Question 7

Calculate the accuracy of the classifier on the test set, by comparing the predicted values against the ground truth. Use a softmax for the classifier outputs. [1 mark]

About softmax & accuracy for multiclass classifier

Softmax

Softmax is a mathematical function that converts a vector of numbers into a vector of values between 0 and 1 (which can be interpreted as probabilities in certain contexts), where the value for each number is proportional to the relative scale of that number in the vector. In other words, we obtain the value of each number x as a ratio of $f(x)$ with respect to all the other values of f for each number of the vector; f is chosen based on the following requirements:

- It should be positively valued (so it is easier to map to $[0, 1]$)
- $x \propto f(x)$ (i.e. proportionality should be maintained)

- $\frac{x_1}{x_2} \propto \frac{f(x_1)}{f(x_2)}$ (i.e. relative proportionality should be maintained)

The function $f(x) = e^x$ fulfills these conditions. Thus, the softmax function is defined as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

NOTE: $\sigma(\vec{z})_i$ is the softmax value for the ith element of vector $\vec{z}$, and K is the number of elements in $\vec{z}$.

In the case of classification, our vector is xw^T (*we are matching the dimensionalities of our current implementation*), where x is a row vector denoting an observation consisting of the values of N features along with a bias term 1, and w is the matrix consisting of K rows of weight vectors (each vector consisting of N weights along with a bias), each for a certain model trained with respect to a certain class; each of these weight vectors are based on a respective model mapping input vectors x to an output value that indicates how strongly x belongs to the respective class. If there are a set of classes $y = 1, 2...K$, the softmax function with respect to a certain class c maps each element of xw^T to the probability of the object (to which the observation x applies) being in the class c. The goal of training is to adjust the weights such that this classification becomes accurate. Clearly, this is a binary classification, as it only concerns with membership to a single class; even if y can take many values, softmax only lets us check membership for one class at a time. Thus, the softmax value of an object to which a certain observation x refers, with a certain class c to which this object may or may not belong, is given by:

$$P(y = c|x, w) = \frac{e^{xw_c^T}}{\sum_{k=1}^{K} e^{xw_k^T}}$$

NOTE: w_k is row vector of weights belonging to the classifier model for class k, and K is the total number of classes. w is the matrix where each row vector is the vector of weights for the a different class' classifier model (altogether covering all classes).

SIDE NOTE: *The softmax function reduces to the sigmoid function (used in logistic regression) when* $K = 2$.

> REFERENCES:
>
> - https://machinelearningmastery.com/softmax-activation-function-with-python/
> - https://www.turing.com/kb/softmax-multiclass-neural-networks

Accuracy function

Let M be the total number of observations (involving N features), let K be the total number of classes we are considering, and let y be a variable to denote the class (it can take one of K values). Hence, we define the following matrices:

- $X_{M \times N}$: Each row vector is a single observation

- $Y_{M \times K}$ Each row vector v_i is the vector of indicators equalling either 0 or 1 for the corresponding observation;
    - $0 \implies$ "not in class v_i"
    - $1 \implies$ "is in class v_i"
    - Hence, if $y = j$ for observation i, $Y_{ij} = 1$, else 0
- $W_{K \times N}$ is the matrix where each row vector is the vector of weights for the corresponding class' classifier model

In the one vs. many multiclass classifier, we performed a simple binary classification for each class, obtaining the weights above and thus obtaining the probability of belonging to each class (given by the softmax function). We shall use this to obtain the accuracy. Initialising $a = 0$, for each observation i, do the following...

- Get the index $j \in 1, 2 ... K$ for which he softmax value is maximum; j is thus the prediction
- Only if $Y_{ij} = 1$, increment a by 1

After finishing the above process, divide a by the total number of observations, which equals the total number of predictions. This is the accuracy. In general:

$$Accuracy = \frac{nCorrectObservations}{nTotalObservations}$$

Implementing the above accuracy function...

```python
classList = (setosa, versicolor, virginica) # Global variable

# Helper function to get the softmax value for a given observation & class:
def softmax(x, classIndex, classList=classList):
  """
  We expect x to be a row of predictor variable values (with bias term).
  classIndex indicates which class to calculate softmax for.
  classList is the whole list of possible classes.
  """
  V = []
  for C in classList:
    w = C.model.state_dict()['0.weight'] # Obtains the weights of the model
    V.append(torch.exp(x @ torch.transpose(w, 0, -1)))
  return V[classIndex]/sum(V)

#===============================================

# The main accuracy function:
def get_accuracy(x, y, K=3):
  a = 0
  for i in range(x.shape[0]):
    pMax, jMax = 0, 0
    for j in range(K):
      p_ij = softmax(x[i], j)
      if p_ij > pMax: pMax, jMax = p_ij, j
    if y[i, jMax] == 1: a += 1
  return a/y.shape[0]
```

```
# Testing the softmax function:
tr = softmax(x_train[0], 0), softmax(x_train[0], 1), softmax(x_train[0], 2)
te = softmax(x_test[0], 0), softmax(x_test[0], 1), softmax(x_test[0], 2)
print(f"VALUES={tr}\nSUM={sum(tr)}")
print(f"\nVALUES={te}\nSUM={sum(te)}")
```

```
VALUES=(tensor([1.0000]), tensor([2.2614e-06]), tensor([9.2007e-07]))
SUM=tensor([1.0000])

VALUES=(tensor([0.0102]), tensor([0.7731]), tensor([0.2166]))
SUM=tensor([1.0000])
```

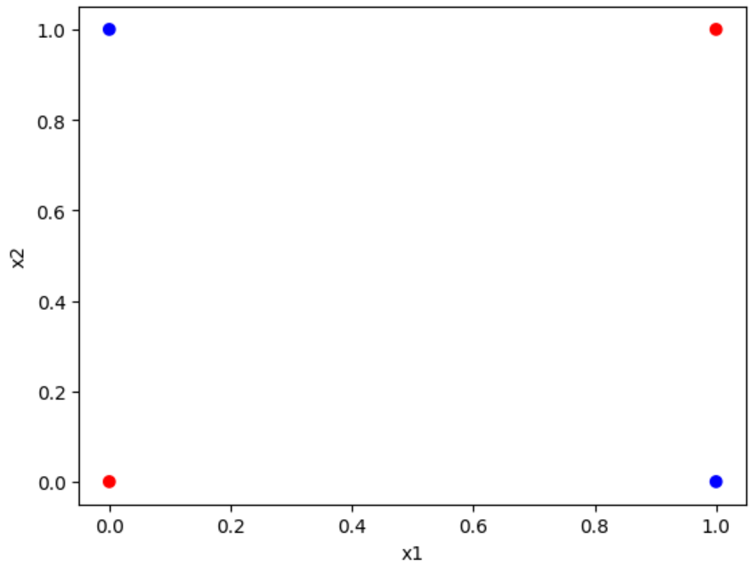
```
# Getting the training & testing accuracy:
print(f"""ACCURACY
Training accuracy: {get_accuracy(x_train, y_train_oneHot):.3f},
Testing accuracy: {get_accuracy(x_test, y_test_oneHot):.3f}""")
```

```
ACCURACY
Training accuracy: 0.850,
Testing accuracy: 0.900
```

3. The XOR problem

Question 8

Looking at the datapoints below, can we draw a decision boundary using Logistic Regression? Why? What are the specific issues of logistic regression with regards to XOR? [2 marks]



OBSERVATION:
The data above is not linearly separable, i.e. there is no straight line that can suitably divide the points above with respect to their classes. Logistic regression is a linear classifier, which means it partitions the data into one of two classes by checking on which side of a single threshold value the linear combination of inputs and weights lie. Thus, logistic regression by itself is ill-equiped to deal with non-linear class boundaries such as in XOR.

PART B: *Neural networks*

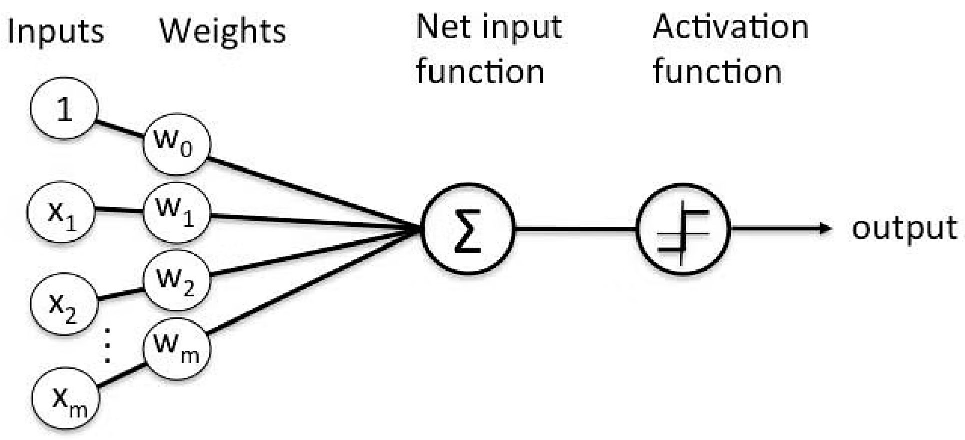
Introduction

The aim of this lab is to get familiar with **Neural Networks**. We will be using some code extracts that were implemented on the week 4 Classification I lab and build a Neural Network. For this lab, we will again be using the iris dataset.

0.1 Intro to Neural Networks

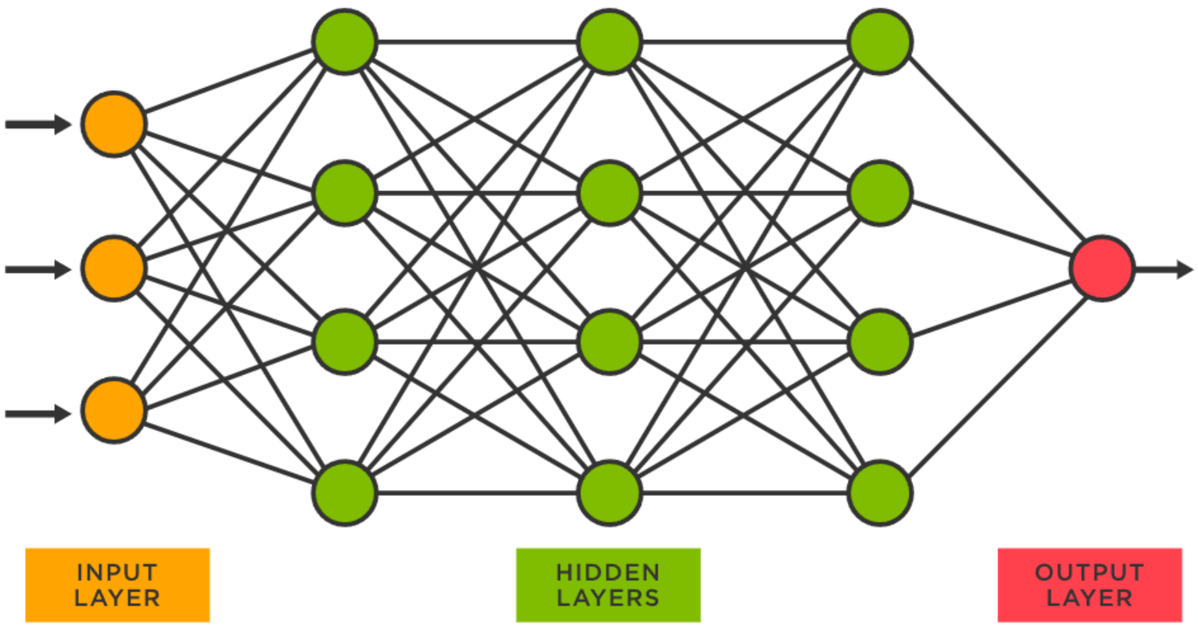
As covered in the lecture notes, Neural Networks (NN) are inspired by biological brains. Each "neuron" does a very simple calculation, however collectively they can do powerful computations. A simple model neuron is called a Perceptron and is comprised of three components:

1. The weights
2. The input function
3. The activation function



We can re-imagine the logistic regression unit as a neuron (function) that multiplies the input by the parameters (weights) and squashes the resulting sum through the sigmoid.

A Feed Forward NN will be a connected set of logistic regression units, arranged in layers. Each unit's output is a non-linear function (e.g., sigmoid, step function) of a linear combination of its inputs.



We will use the sigmoid as an activation function. Add the sigmoid function and `LogisticRegression` class from week 4 lab below. Change the parameter initialization in `LogisticRegression`, so that a random set of initial weights is used.

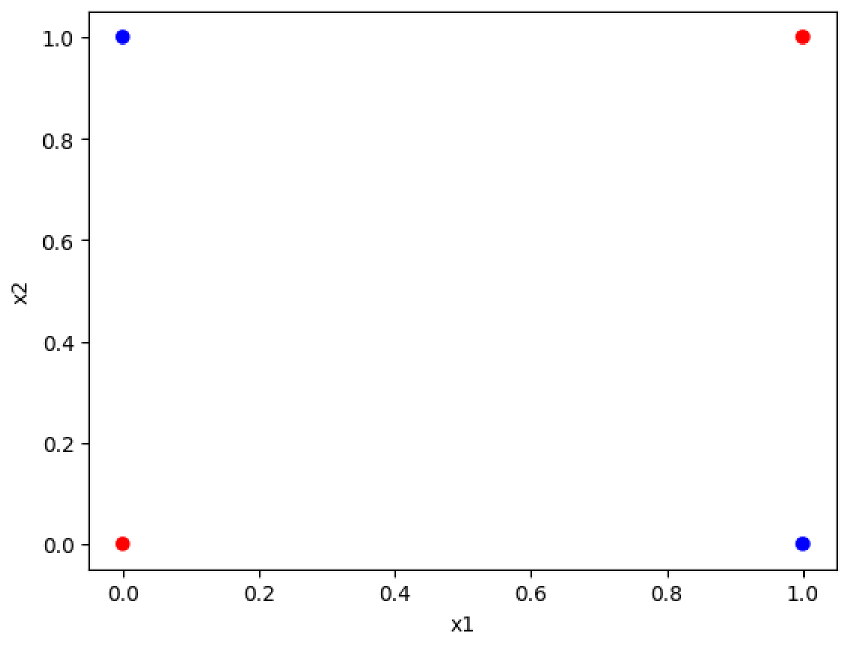
Question 1

Why is it important to use a random set of initial weights rather than initializing all weights as zero in a Neural Network? [2 marks]

The answer lies in the way the weights are updated. The partial derivative of the final error/loss function value (between the output of the network and the expted output) is backpropagated and calculated not with respect to every individual weight but with respect to every individual *neuron*'s input (which is a linear combination of all the weights and inputs to the neuron). Weights in turn are updated based on the activation function (*the function of the linear combination of weights and inputs to the neuron*) and the neuron's respective partial derivative value. Hence, if the weights of a neuron are all equal, they will change in the same direction at the same rate, becoming no more than copies of each other and thus losing the ability to simulate complex multiparameter functions within their limited scope. Hence, all the weights of the network being set to zero will render the whole network useless for estimating complex multiparameter functions.

1. The XOR problem

Let's revisit the XOR problem...

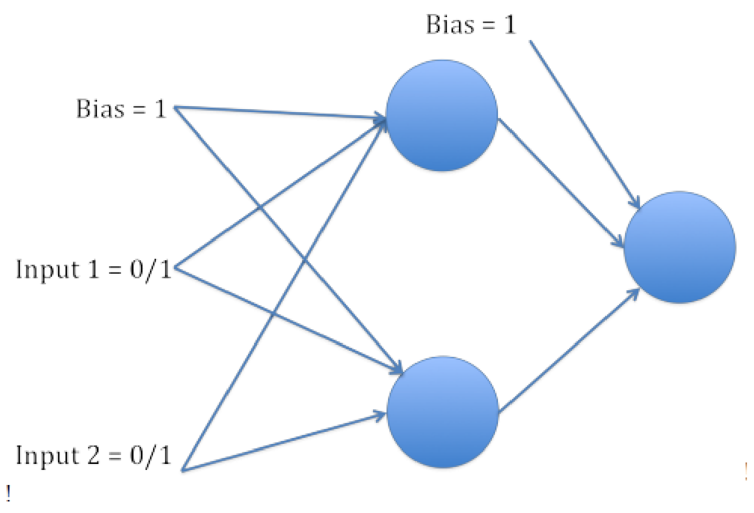


How does a NN solve the XOR problem? [1 marks]

Mathematically, a neural network is a sequence of linear combinations and one or more non-linear transformations on a set of inputs (input vectors) that can in theory model any function, no matter how complex. This is useful in the case of a classification case such as XOR, which is not linearly separable and thus not amenable to standalone linear classifiers. In the case of the XOR problem, a NN two or more hidden neurons (*each with its own weighted input sum and transformed output*) can be tuned to estimate a complex, non-linear function that can help classify XOR inputs, *provided that at least some activation functions used, i.e. some of the functions that transform the weighted inout are non-linear functions*.

NOTE: *A NN with one hidden neuron is essentially a single function on a linear combination of a single set of weights; not much better than a usual classifier*.

We will implement back-propagation on a feed forward network to solve the XOR problem. The nework will have two inputs, two hidden neurons and one output neuron. The architecture is visualised as follows:



Test output of the implemented neural network (implementation not presented here):

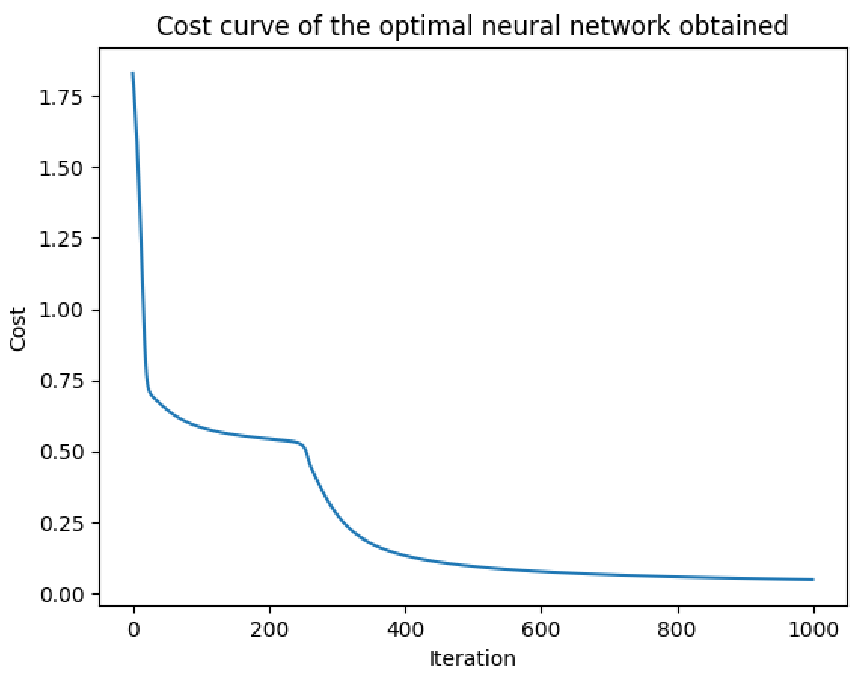
```
Prediction: 0.6679260730743408
------------
Model:
NeuralNetwork(
  (hidden_layer): ModuleList(
    (0-1): 2 x LogisticRegression()
  )
  (output_layer): ModuleList(
    (0): LogisticRegression()
  )
)
------------
Named parameters:
hidden_layer.0.weight Parameter containing:
tensor([[-0.0225,  1.6093, -2.4691]])
hidden_layer.1.weight Parameter containing:
tensor([[-2.2078, -1.1555,  0.8045]])
output_layer.0.weight Parameter containing:
tensor([[-0.0594,  2.3787, -0.2662]])
------------
Hidden layer weight matrix:
tensor([[[-0.0225,  1.6093, -2.4691]],

        [[-2.2078, -1.1555,  0.8045]]])
```

IMPLEMENTATION/CONCEPTUAL NOTE:

To optimise my model, I have used a loop to reinitialise the neural network model and restart the optimisation n times and return the best model out of n. The reason for this is that though random weights are better than constant weights initially, it is not guaranteed that the assigned random weights will be overall able to converge to the desired goal. Evidently, the initial weights of the neural network have a significant impact on the accuracy of the output (as well as on the convergence of error or loss). No matter the number of steps used (within 1000-10000 steps), in some initialisations, the network never converges to the desired outputs, instead getting stuck in a local optimum. This is a known problem in neural networks, as the complex functions they embody as well as their associated loss/cost functions tend to have various local optima.

After n iterations, we pick the best...



```
RESULTS:
y:0.0, prediction:0.041922565549612045
y:1.0, prediction:0.951196551322937
y:1.0, prediction:0.9509459733963013
y:0.0, prediction:0.04670383781194687

Final error obtained: 0.0478825569152832
```

The error function used to visualise the costs per iteration is the BCE, i.e. binary cross entropy function.

2. Iris Dataset

We will now use pytorch built-in methods to create an MLP classifier for the iris dataset.

Description of the data set `iris`

Iris plants dataset

Data Set Characteristics:

- Number of Instances: 150 (50 in each of three classes)
- Number of Attributes: 4 numeric, predictive attributes and the class
- Attribute Information:

    - sepal length in cm
    - sepal width in cm
    - petal length in cm
    - petal width in cm
    - class:
        - Iris-Setosa
        - Iris-Versicolour
        - Iris-Virginica

Summary Statistics:

	Min	Max	Mean	SD	Class correlation
sepal length	**4.3**	**7.9**	**5.84**	**0.83**	**0.7826**
sepal width	**2.0**	**4.4**	**3.05**	**0.43**	**-0.4194**
petal length	**1.0**	**6.9**	**3.76**	**1.76**	**0.9490**
petal width	**0.1**	**2.5**	**1.20**	**0.76**	**0.9565**

- Class Distribution: 33.3% for each of 3 classes ($\implies$ data points are uniformly divided into classes)
- Creator: R.A. Fisher
- Donor: Michael Marshall
- Date: July, 1988

SIDE NOTE: *Class correlation is the degree of correlation with respect to the given feature within each class.*

As before, we split the data to train and test sets (make sure the same random seed is used as previously) and normalize using the method from part A. Hence, again, we get the following tensors:

- `x_train` : An $M_{tr} \times N$ matrix where each column denotes the normalised values of a particular feature & each row denotes a particular observation
- `x_test` : An $M_{te} \times N$ matrix with the same arrangement as `x_train` (also normalised with respect to the mean & standard deviation of `x_train`)
- `y_train` : An $M_{tr} \times 1$ column vector of the target values corresponding to each observation of the training set
-
    - `y_train` : An $M_{te} \times 1$ column vector of the target values corresponding to each observation of the testing set

As a reminder, the features are...

- `sepal length`
- `sepal width`
- `petal length`
- `petal width`

... and the target values are...

- 0 means setosa
- 1 means versicolor
- 2 means virginica

Creating a multilayer perceptron (MLP)

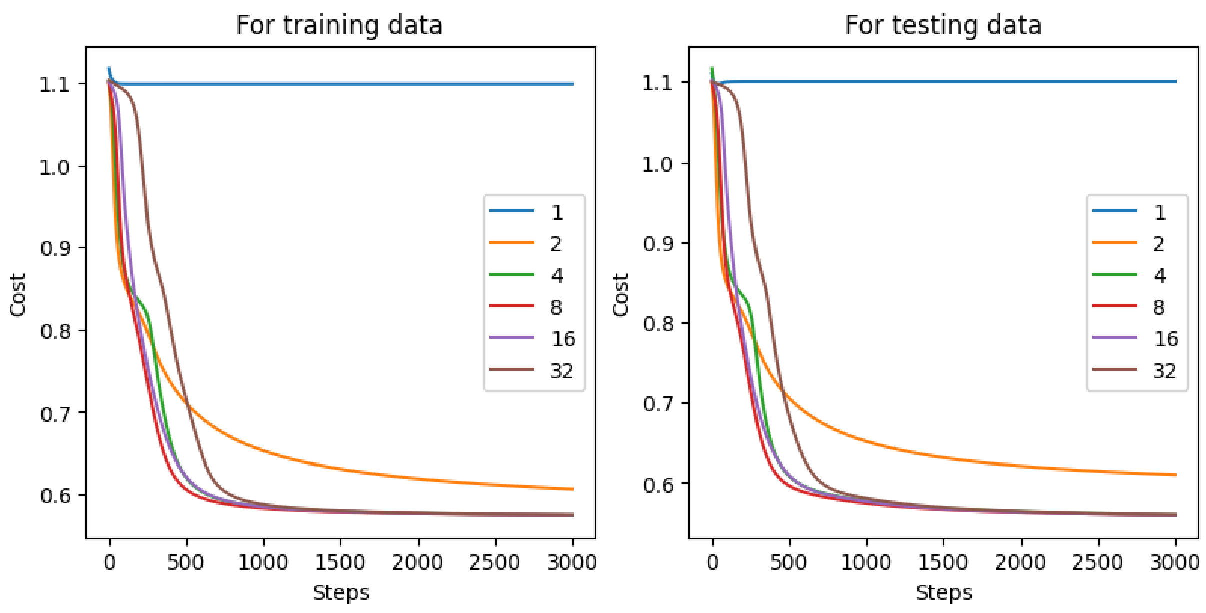
A MLP is a simple neural network consisting of a set of two or more sequentially interconnected layers of neurons, i.e. it is a network consisting of multiple perceptrons, hence the name.

Though a MLP generally has more than two hidden layers, based on the constraints of this assignment, we work with only two hidden layers. The implemented neural network has two sequential layers:

- **Linear layer 1** with *softmax* as its activation function
- **Linear layer 2** taking the previous activation function's output vector as its input, with *softmax* as its activation function (for the final output)

Softmax is used as it is an efficient way to assign probabilities of classification to each input vectors. The linear layer, via matrix multiplication, simulates a hidden layer of neurons (this is discussed further in the implementation notes in the associated Jupyter Notebook).

Below are the outcomes of the training (and simultaneous testing) of the created neural network. The numbers 1, 2, 4, 8, 16 and 32 in the graphs are the number of hidden nodes the respective model has (corresponding to that model's cost curve).



```
Minimum training cost obtained for each number of hidden nodes:
1: 1.0984039306640625
2: 0.6064962148666382
4: 0.5748772025108337
8: 0.5747455358505249
16: 0.57480388879776
32: 0.5750030875205994

Minimum testing cost obtained for each number of hidden nodes:
1: 1.1004875898361206
2: 0.6097186803817749
4: 0.5598499178886414
8: 0.5594740509986877
16: 0.5594872236251831
32: 0.559897780418396
```

Having a look at the accuracy of the predictions (i.e. multiclass classification) of the above models (using the accuracy function from part A)...

```
ACCURACY
Given hidden nodes...
1; training:0.342, testing:0.300
2; training:0.967, testing:0.967
4; training:0.983, testing:1.000
8; training:0.983, testing:1.000
16; training:0.983, testing:1.000
32; training:0.983, testing:1.000
```

OBSERVATIONS: **One vs. many logistic regression & the above neural network**:

For the same problem of classifying data points as one of three flowers (setosa, versicolor and virginica) based on the attributes (*sepal length, sepal width, petal length, petal width*) was approached in two ways:

1. The above neural network (NN)

2. The previous one vs. many logistic regression method

Overfitting (or lack thereof)...

The previous one vs. many logistic regression model had the following results:

- Training accuracy: 0.850,
- Testing accuracy: 0.9

The levels of accuracy of the training and testing data being quite similar indicate a lack of overfitting in the model; hence, we can consider the model to be well generalised. We observe the same phenomenon with the neural networks, wherein the training and testing loss function values are similar to each other, no matter the number of hidden nodes. Hence, we can conclude that the neural network in this scenario does not overfit to the training data.

Accuracy of NN...

1.

With respect to number of hidden nodes in the hidden layer...

At one hidden node, the accuracy is abysmal, but rises rapidly with the second hidden node and stays more or less the same (above 0.95) for all subsequent versions.

2.

With respect to one vs. many multiclass classifier...

The view of the accuracy of the neural network models given above show that with enough hidden nodes, the neural network tends to be more accurate than the one vs. many mutliclass classifier.