1 CONTENTS

Unit 4: Linear and Non-Linear Regression

Unit learning outcomes

After completing this unit you should be able to:

- use linear regression models in order to predict the value of a continuous label (regression)
- understand how we can establish objective function to optimise a linear model
- devise an exact solution for linear regression models through least squares
- devise an inexact solution for linear regression models through gradient descent
- understand how generalised linear models utilise basis functions in order to obtain the capabilities of modelling a non-linear function.
- further develop the generalised linear regression models into multi output generalised linear regression models
- devise a multi-layer neural network learning algorithm with one output, that is capable of automatically inferring the best basis for a regression problem

In the previous unit, we saw how we can apply decision trees to classify data with tree induction algorithms such as CART. When the data is purely numerical (all features are numerical not categorical or nominal) then DT might not be the best technique to choose. Similarly, although we have not showed it yet, DT can be used for regression, but by nature DT in its simplest form needs to utilise some form of discretisation for regression and will normally perform inferior to other techniques that are originated form numerical optimisation.

In this unit, we will cover models that allow us to perform regression and classification on numerical datasets. Even if the dataset is not fully numerical, we can convert those features that are categorical into numerical values without loss of accuracy or generality. Of course, this will depend on the feature's values and its nature. A common feature is gender where we can convert its values into {0, 1, 2} to represent male, female, and no specific gender, respectively. Those models also output numerical values. For regression, this would be exactly what we want since regression is concerned with predicting a specific value instead of a category. For classification, we can either take the output to mean probability of belonging to a specific class (and in this case we would need multi-output that represents each possible class) or we can convert it via other discretisation methods such as picking category based on a range of values, each specific range corresponds to a specific category.

2 LESSON: SIMPLE LINEAR REGRESSION MODELS

Lesson learning outcomes:

After completing this lesson you should be able to:

- use the straight line formula in order to predict the value of a continuous variable
- understand that the weights map into the coefficient of a straight line formula
- generalise linear regression models from a 1 dimensional input space into multiple dimensional input space
- devise a loss function for a linear regression model with multi input space
- exactly optimise a loss function via the least squares algorithm
- approximately optimise a loss function via the gradient descent algorithm
- understand how to perform batch learning and sequential learning, and the differences between them.

In this unit we talk about a simple yet effective group of data mining models, namely linear models. These are called **supervised learning techniques** since we train our models via a training set that has the required answers that we are after. Those answers are called labels. The idea is that later on after we train our model, if we present a fresh new case to it, the model will be able to guess what its answers. This brings us to two issues that we will deal with.

2.1 What is regression?

First the nature of the answer is specified by the nature of the mining task that we are trying to perform. In supervised learning we have largely two types of answers that we are looking for: categorical or numerical. If it is categorical, our task would be to guess the category (or the class) of the cases under examination. This type of data mining task is called classification. If the label is numerical, then we need to come up with a numerical prediction that is as close as possible to the **desired answer** stored in our dataset. This type of numerical prediction is called **regression**. After training, during prediction, our regression model attempts to guess the best answer of an unknown data point label.

Note: We call the answer that is stored in the dataset the **desired answer**, and the model answer is called the **predicted answer**. This applies for both classification and regression.

Remember: we agreed to call these cases **data points** because each is recognised by a set of features. The features themselves can be categorical or numerical. Since we can convert any categorical data into numerical as we pointed out in unit 1, we will assume that all the data that we are dealing with is numerical.

The second question to ask is how we can verify that the model is good enough for our task; in other words, how can we measure the performance of our model. Measuring the performance will be done by utilising the available answers that we have in our dataset. Think about it: if we do not know the correct answers, we have no way of telling whether the model answers are accurate or good enough. Therefore, we set aside part of our dataset with its available answers, for measuring the performance of our model. This is called partitioning and was covered in unit 2. So, in simple terms, similar to classification, in regression tasks, we split the dataset into two parts, one for training and one for measuring the performance. And when the data is not big enough or when we want to scrutinise the model performance, we can partition the dataset into three parts, training, validation and testing and we can further apply cross validation as we saw in unit 2 for the classification problems.

So, really the difference between classification and regression is the nature of the labels. Note that we can sometimes convert a regression problem into classification if we categorised the labels. Of course, this is not desired since we lose a lot of granularity of the labels and we only obtain very rough estimates of the answers, and usually it is not a good idea. The figure below shows a schematic illustration of the regression model, y here is a continuous value.

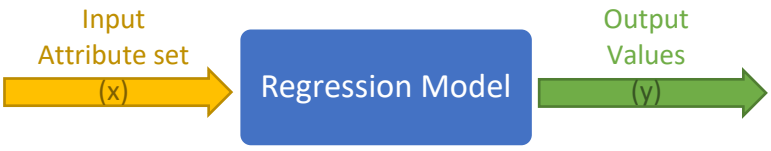

Figure (1.1) A Schematic Illustration of regression.

2.2 Example of Linear Regression Model

In its simplest form, linear models attempt to draw a straight line that fits a set of data point. Given the following dataset:

x	-10	-7.8	-5.6	-3.3	-1.1	1.1	3.3	5.6	7.8	10
y	-8.9	-4.2	0	3.1	6.6	11.9	17.3	19.8	24.8	29.8

We can fit a linear model that we show below on the left. On the right we show another larger dataset with its fitted model (the dataset can be viewed here).

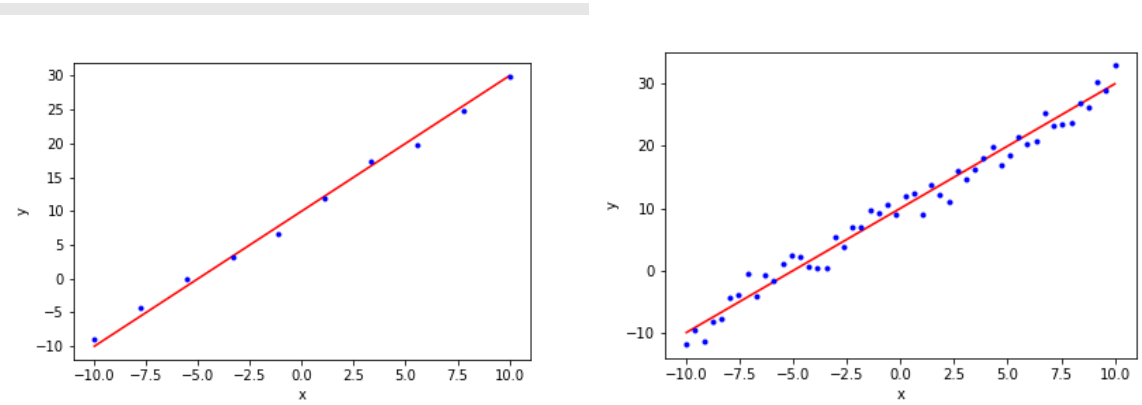

Figure (1.2): A linear regression model (red line) that fits the blue data points. Left: 10 data points, right: 50 data points.

Important Note: We denote matrices by a bold face capital letter, vectors by a bold face letter and variables by a normal face letter and components of a vector will be italic. So for example:

- x_n is a real value corresponds to data point n in one dimension space.
- while $\mathbf{x}_n$is a vector of multiple components that corresponds to data point n in D-dimensional space.
- t_n is a real value corresponds to a target n in one dimension space.
- $\boldsymbol{t}_n$ is a real value vector that corresponds to a multi-output target n in K dimensional space.
- x_i (and $x_{n,i}$) is components i of some vector $\mathbf{x}$ (or $\mathbf{x}_n$) depending on the context.
- $\mathbf{w}$ is a weight *vector* of D (or M) components in other words it is a vector of D rows and 1 column.
- while $\mathbf{W}$ is a weight *matrix* of $D \times K$ (or $M \times K$) components. In other words it has D rows and K columns.

2.3 Inference in Linear Regression Models

The idea is that later on when we want to know the y of a given x we can extrapolate or interpolate. So if we assume that the previous linear regression model has the formula $y = 2x + 10$ then given that $x = 2$ then we can immediately infer from our model that $y = 2x2 + 10 = 14$. Similarly, given any regression model we can ask the model to predict for us a value y given an input x. In the next sections we will see how we can actually build or train such linear regression models. We start by formulating the regression problem in a proper mathematical framework and we will see how we can train a linear regression model via a set of algorithms.

2.4 Problem Formulation

We assume that we have a training dataset that consists of:

1. N observations $\{\mathbf{x}_n\}$, where $n = 1, \dots, N$
2. A set of corresponding target values $\{t_n\}$.

We call $\mathbf{x}_n$a data point, an observation, a record or a case, interchangeably. Similarly, we call t_n the label, the target value or the answer, interchangeably.

Note: that $\mathbf{x}_n$ is a vector of D real values. We assume that t_n is a real value scalar. Also, note that in the general case $\boldsymbol{t}_n$ can be a vector of size K, we will come to that later.

Important note: All the techniques of partitioning the dataset into training, validation and testing sets, with cross validation sets if necessary, that we have discussed in unit 2 apply to all the techniques that we discuss in this unit. From now on any reference to a dataset in a training context assumes that we are talking about a training set.

2.5 The Aim of Constructing a Model

Given a new unseen observation $\mathbf{x}$, the goal is to train a model to predict the target value $y(\mathbf{x})$ for the given observation $\mathbf{x}$ so that $y(\mathbf{x})$ resemble or come as close as possible to the 'would be' real target value t. During training, both $y(\mathbf{x}_n)$ and t_n are available and their difference drives the learning journey of the model. After training, when the model is used in real settings, we do not know the target value t. The whole point of constructing the model is to predict such a value. So the generalisation and prediction capabilities of our model have to be specified from the available answers t_n.

2.6 Linear Regression as a Parametric Model

One of the simple types of parametric models is the linear regression model which is the topic of this lesson. It belongs to a wider group of models called parametric models. Parametric models have one important thing in common, which is that they all use a set of adjustable parameters (aka weights) which the model learning algorithm tweaks in order to reduce the loss function and make the model prediction as close to the desired target values as possible. Throughout this unit we denote the adjustable parameters or weights as w. In the next section we will see how the linear regression model can be expressed in terms of its weights.

2.7 Why we call it linear:

Before we start discussing the different forms of linear regression models. We need to be clear about the word linear. When we say *linear* we mean in terms of the weights and not necessarily in terms of features. This will be clear when we go through the different types of linear models for regression.

Ok, let us start with the simplest linear regression model.

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1$$

This is the simplest type of linear model. It defines a relationship between y and x_1 as a straight line. If you remember from high school that we define a straight line as $y = c + mx$ you can immediately realise that w_0 is the intercept of the straight line on the y axis (corresponding to c) and w_1 are just the slope of the straight line (corresponding to m).

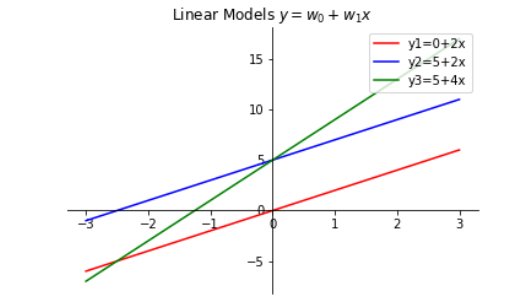

Figure (1.3): shows three different linear models with one variable x, each has its bias w_0 and slop w_1. Note that the red and blue has the same slope, while the green and the blue has the same bias.

2.8 LINEAR REGRESSION FOR MULTI-DIMENSIONAL INPUT SPACE

We can generalise the idea of a linear model from $D = 1$ dimensional space into $D \geq 1$ space input. We only need to take into account that we have more than 2 coefficients and these coefficients are called the weights in general, w_0 is still called the bias. Let us assume that we have a D input space like a set of numerical features of some entity like a house or a car etc. Each input $\mathbf{x}$ will be a vector and will take the form

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \\ \vdots \\ x_D \end{bmatrix}$$

Which also can be written as $\mathbf{x} = [x_1, x_2, \dots, x_D]^\top$, where $\top$ denotes the transpose of a matrix or a vector. A linear regression model performs the prediction of the input vector $\mathbf{x}$ by multiplying each component of the observation x_i by a weight w_i as follows:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D$$

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i$$

where $\sum_{i=1}^{D} w_i x_i$ means a simple multiplication of each x_i with w_i and summing all the multiplications. Learning takes place by adjusting these parameters to make the model produce the desired answers. In simple terms, linear means first order sum.

2.9 VECTOR MATRIX REPRESENTATION WITH DUMMY COMPONENT

It is often convenient to express machine learning tasks such as linear regression in terms of vectors and matrices. This is due to two main reasons. The first, we can apply linear algebra concepts and methods which is widely studied and available in all branches of science. Second, vectorised forms are often far more efficient to implement on a computer using scientific computing languages and packages (including Python numpy, MATLAB and FORTRAN) instead of using loops. So in any implementation task that you do in machine learning you should strive to make your solution depend on vectors and matrices instead of loops whenever possible. The process of changing from loops to vectors and matrices operations is called vectorisation.

Looking at the above linear model expressions, we can easily identify that they can be changed into

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D$$

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \mathbf{w}^\top \mathbf{x}$$

However, it is also more useful if we express the whole model using vectors. To do so, we can define a *dummy* feature $x_0 = 1$ for all vectors of the input space. In this case we can define our linear model as

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x} \tag{1}$$

Where $\mathbf{w}^\top = [w_0, w_1, w_2, \dots, w_D]$ and **our extended input space** vector $\mathbf{x}^\top = [x_0, x_1, x_2, \dots, x_D]$ and $x_0 = 1$.

$\mathbf{w}^\top = [w_0 \quad w_1 \quad \dots \quad w_D]$, $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \\ \vdots \\ x_D \end{bmatrix}$ and $\mathbf{w}^\top \mathbf{x} = \sum_{i=1}^{D} w_i x_i$.

So, we added the dummy feature x_0 to the input space. The operation $\mathbf{w}^\top \mathbf{x}$ gives us one value because $\mathbf{w}$ is a vector, later we will adjust this to get multi-output via $\mathbf{W}^\top \mathbf{x}$ where $\mathbf{W}$ is a matrix not a vector. Below we show a schematic representation of a linear regression model.

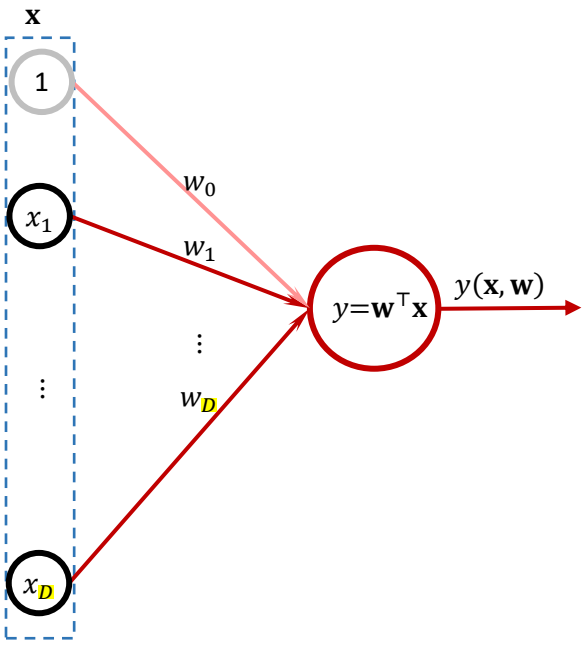

Figure (1.4): schematic representation of a linear regression model.

2.9.1 Vectorised Representation for the Entire Dataset

We can go further and vectorise the entirety of a dataset to obtain all the predictions at once via one operation:

First we define:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \\ \vdots \\ \mathbf{x}_N^\top \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1} & \cdots & x_{1,D} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \cdots & x_{n,D} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N,1} & \cdots & x_{N,D} \end{bmatrix} \quad \text{and} \quad \mathbf{t} = \begin{bmatrix} t_1 \\ \vdots \\ t_n \\ \vdots \\ t_N \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{bmatrix}$$

where we call $\mathbf{X}$ the *design matrix*, it has dimension of $N \times D$ and its nth row is $\mathbf{x}_n^\top$, while we call $\mathbf{t}$ the targets matrix. The design matrix visually resembles data in a table,and so it makes the implementation of the formula more convenient. Let us now re-formalise the problem using $\mathbf{X}$ and $\mathbf{t}$. The prediction of the linear model on the entirety of the dataset can be written as

$$\boldsymbol{y}(\mathbf{X}, \mathbf{w}) = \mathbf{X}\mathbf{w} \tag{2}$$

where $\boldsymbol{y}$ is a vector that comprises all the predications for all data points $\mathbf{x}_n$.

If we want to take the difference between each target and model prediction for the target then we can simply do:

$$\mathbf{t} - \mathbf{X}\mathbf{w}$$

This operation will produce a vector of size N each component of it is $t_n - \mathbf{w}^\top \mathbf{x}_n$.

Please watch the following video on the concepts of regression and see the following presentation for a summary of what we cover in this unit.

2.10 LOSS, OBJECTIVE, ERROR OR COST FUNCTION FOR REGRESSION

In this section we will develop the concept of a loss function. The discussion applies for any learning method that attempts to minimise a loss function and not only linear regression. Since we have numerical data, we want to come up with a function (called the loss function, it will become apparent later why we call as such) that is closely linked to the distance between the desired and the actual answers of our regression model. The idea here is that we want to lead the learning process via a minimisation of the loss function so that we minimise the difference between the desired and actual answers. So really we are talking about an aggregate metric that looks into each data point instead of looking at counting the correctly classified and incorrectly classified cases as we did in the confusion matrix. Later on when we will develop other better classifiers to deal with numerical classification we will actually also use the loss function to lead the learning process (by optimising it) and we are still going to use the confusion matrix to measure the performance of the model *after* learning has finished. So, the loss function is going to be used in this unit for both the regression and classification to lead the optimisation process (learning) in order to learn a best model fit. When we are talking about multi-component labels(a set of numerical answers instead of one). The loss will be defined on the basis of vector distances, this will become apparent later in this section.

2.10.1 Loss function for an individual point

The first thing that comes to mind when we try to measure the accuracy of our model's prediction is to take the difference between (also called the residual or the error) the prediction $y(x_n)$ and t_n. Let us denote $y_n = y(\mathbf{x}_n, \mathbf{w})$ where we will use either of y_n or $y(\mathbf{x}_n, \mathbf{w})$ interchangeably depending on what we are trying to emphasise. So, we can define our loss function $J(\mathrm{x}_n)$ which we denote for brevity J_n as:

$$J_n = t_n - y_n \tag{2}$$

2.10.2 Loss function for a Set of Points (Dataset)

So, given that we have plenty of data points in our dataset, it is natural that we would want our model to perform well on all of them. One problem with the above type of individual point loss function J_n is that it can be either negative or positive and for our prediction both of them are errors. However, the danger is that if we sum negative and positive values for multiple points they can cancel each other at least partially.

One way to make sure that negative and positive residuals do not cancel is to take the absolute of these residuals $|J(\mathrm{x}_n)| \geq 0$ and then to sum over all data points to get the sum of absolute errors (SAE) :$|J| = \sum_{n=1}^{N} |J_n|$ .The $\sum_{n=1}^{N}$ means to sum over all $n = 1, \dots, N$. So if we have $N = 3$ data points in our dataset, then

$$|J| = |t_1 - y_1| + |t_2 - y_2| + |t_3 - y_3|$$

One important issue that we will face with such loss function is that it is not differentiable at 0, making dealing with the derivatives for optimisation not straightforward. A better candidate in that sense is the squared error, giving rise to the sum of squared errors ($SSE = \sum_{n=1}^{N} J_n^2$) loss function that takes the form:

$$J^2 = \sum_{n=1}^{N} J_n^2$$

$$J^2 = \sum_{n=1}^{N} (t_n - y_n)^2$$

$$J^2(\boldsymbol{w}) = \sum_{n=1}^{N} \big(t_n - y(\boldsymbol{x}_n, \boldsymbol{w})\big)^2$$

Note that the 2 on top of J is to indicate that we are summing over the squares and not to indicate a direct squaring operation.

SSE has pros and cons. Its pros are its ease of derivation and positivity. One of its cons is that it exaggerates the residuals, so if a residual is -3, then its squared $(-3)^2$ becomes 9. Nevertheless, SSE is widely used and its advantages outweigh its disadvantages for many problems. Before we settle on it we need to make two tweaks to make later developments easy to express.

As we know finding the minimum for y^2 is the same as finding the minimum for $\frac{1}{2}y^2$ but the latter leads to a simpler derivative: $\frac{d}{dy}\left(\frac{1}{2}y^2\right) = y$, while $\frac{d}{dy}(y^2) = 2y$. Hence, we can use the following loss function (based on SSE) that simplifies taking derivatives:

$$J^2 = \frac{1}{2}\sum_{n=1}^{N}\left(t_n - y(\mathbf{x}_n, \mathbf{w})\right)^2$$

Furthermore, we can take the mean of squares to obtain the mean squared error (based on $MSE = \frac{1}{N}\sum_{n=1}^{N} J_n^2$) as a loss function

$$\bar{J}^2 = \frac{1}{2N}\sum_{n=1}^{N} J_n^2 \tag{3}$$

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}\left(t_n - y(\mathbf{x}_n, \mathbf{w})\right)^2 \tag{4}$$

This loss function has the desired properties of keeping the range of the loss function fixed regardless of the size of the dataset. When we want to differentiate between the loss for training set and for validation set we write the loss as

$$\bar{J}^2(\mathbf{w}, \mathbf{X}, \mathbf{t}) = \frac{1}{2N}\sum_{n=1}^{N}\left(t_n - y(\mathbf{x}_n, \mathbf{w})\right)^2$$

$$\bar{J}^2(\mathbf{w}, \mathbf{X}', \mathbf{t}') = \frac{1}{2N'}\sum_{n=1}^{N'}\left(t_n - y(\mathbf{x}_n, \mathbf{w})\right)^2$$

where N' is the size of the validation set. For the majority of the coverage here we will refer to the loss either as $\bar{J}^2$ or as $\bar{J}^2(\mathbf{w})$. In theory, we can use either J^2 or $\bar{J}^2$ in the techniques that we will cover. This is because scaling any function by a fixed constant such as $\frac{1}{2}$ or $\frac{1}{2N}$ will not change the shape of the function and so its stationary (optimal and saddle) points stay the same. For example, if we take the derivatives and we set them to 0 both will yield the same solution. However, $\bar{J}^2$ offers more numerical stability than J^2 and there are few cases (particularly when we add a regularisation term to facilitate a stochastic gradient decent algorithm) where starting from $\bar{J}^2$ will make the update term slightly more consistent with other updates.

Below we show an example of a linear model with its loss function, the learning algorithm mission will be to find the parameter settings that minimise the loss function for the given data, i.e. to find the bottom of the bowl shaped loss function. Linear models have a similar shaped loss function, but not all models have loss functions that look as nice and tidy as this example, in particular non-linear models might have very difficult terrain to navigate.

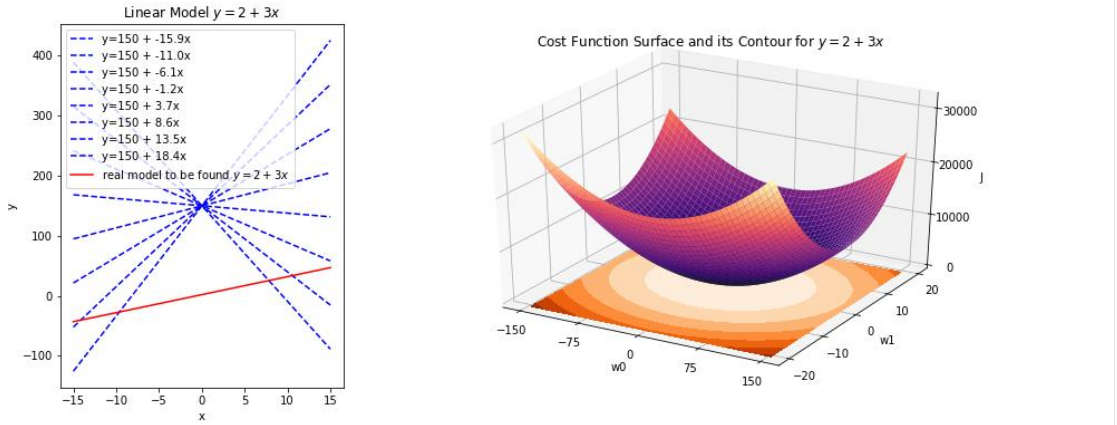

Figure (1.5): (left) Example of a linear model (right) the loss function of a different settings for w0 and w1 (shown in purple) and the loss function contours plot shown in orange. The task of learning is to reach the bottom of the loss function where the optimal settings of the weights values are. Contour plots project the surface above it and signifies the J by the darkness of the colour so the more orange the higher J is and more error we have. [The code to generate the figure is available here.]

2.10.3 Vectorised version of the Loss Function

The loss function can be vectorised and written as follows:

$$\bar{J^2} = \frac{1}{2N}\|\mathbf{t} - \boldsymbol{y}(\mathbf{X}, \mathbf{w})\|^2 \tag{5}$$

$$\bar{J^2} = \frac{1}{2N}\|\mathbf{t} - \mathbf{X}\mathbf{w}\|^2 \tag{6}$$

Where $\|.\|^2$ is the norm of a vector = sum of the squared of all of its components and $\mathbf{X}$ and $\mathbf{t}$ are the design matrix and target vector that were defined in the previous section.

Generalising SSE to Minkowski Loss

You might wonder why we do not use a smaller exponent $1 < a < 2$ for y^a instead of y^2? Although this might seem reasonable, since for example $3^{1.01} \approx 3$ and taking the derivative for y^a is straightforward. However, this has two issues. The first is related to positive residuals, which the derivation underestimates. For example, $\frac{d}{dy}(y^{1.1}) = 1.1y^{0.1} = 1.1y^{\frac{1}{10}} = 1.1\sqrt[10]{y}$,and if $y = 30$ then its derivative is ≈ 1.546 .The second and more serious issue is that real powers for negative residuals are not defined, try $(-3)^{1.01}$on the calculator. In fact, even for fractional exponent it might still not be defined if the denominator is even: try to calculate $(-3)^{\frac{2}{3}}$ and $(-3)^{\frac{2}{4}}$.

Note that $(-3)^{\frac{2}{4}} = \sqrt[2]{-3}$ is not defined in the real number set $\mathbb{R}$. Also, note that although we can write $(-3)^{\frac{2}{4}} = \sqrt[4]{(-3)^2} \approx 1.732$, for such operation to be well defined we should have $\sqrt[4]{(-3)^2}$ to be equal to $\left(\sqrt[4]{-3}\right)^2$ unfortunately, the latter is not defined (in $\mathbb{R}$).

Exercise: try the same procedure for $(-3)^{\frac{2}{3}}$, i.e. calculate $\sqrt[3]{(-3)^2}$ and $\left(\sqrt[3]{-3}\right)^2$ and see if they are equal. A credible solution then is to simply do the following $J = \sum_{n=1}^{N}|t_n - y(\mathbf{x}_n)|^{1.1}$ which would avoid the issues that arises when the residuals are negative.

A generalisation of the above function would be in the form of **Minkowski** loss defined as:

$$J^q = \sum_{n=1}^{N}|t_n - y(\mathbf{x}_n, \mathbf{w})|^q$$

Where q can take any value. When $q = 2$ we go back to the SSE loss.

More on SSE Format

SSE is written as $J^2(\mathbf{w}) = \sum_{n=1}^{N}\left(t_n - y_n\right)^2$

Which also can be written as $J^2(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - t_n\right)^2$

Both will produce the same results due to the square, ex. $(10-x)^2 = (x-10)^2$, and both have the same derivatives with respect to y_n.

The first form has a derivative $\frac{\partial J^2}{\partial y_n} = -\frac{2}{2}\left(t_n - y_n\right) = -\left(t_n - y_n\right)$

The second form has a derivative $\frac{\partial J^2}{\partial y_n} = \frac{2}{2}\left(y_n - t_n\right) = -\left(t_n - y_n\right)$

The first form is *more desirable* because when we move to a stochastic gradient descent setting the term $\left(t_n - y_n\right)$ in the bracket will appear in the update rule without the squares. And so using this from will help our memory to remember that in our treatments an update the target t_n always comes before the estimation y_n.

2.11 Optimising the Loss: Training Approaches for Parametric Models

To recap, in order to come up with a best settings for our model eq.(1)

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$$

we need to minimise the loss function eq.(4)

$$\overline{J^2} = \frac{1}{2N}\sum_{n=1}^{N}\left(t_n - y(\mathbf{x}_n, \mathbf{w})\right)^2$$

that involves the N points in our training set. To do so, we 1) take the derivative of the loss function $\overline{J^2}$ and 2) set it to 0 to obtain the best weights setting that makes our loss minimal (the point that lies on the bottom of the loss function). In other words we need to minimise $\overline{J^2}(\mathbf{w})$ with respect to $\mathbf{w}$. This is called optimising the loss function $\overline{J^2}$, so learning in this context is a form of optimisation (there are plenty of perspectives for learning that differs or complement this point of view, one of them is the probabilistic approach. We touch upon the probabilistic perspective in later sections).

When we want to take the derivative of a function with respect to a *vector* we take the *gradient of the function*. Using usual rules of derivations, we obtain the gradient of the loss function. Since we are optimising with respect to weights $\mathbf{w}$ we take the gradient with respect to $\mathbf{w}$. The gradient of the loss function $\overline{J^2}(\mathbf{w})$ is a *vector* that takes the partial derivative with respect to each component of $\mathbf{w}$ (the function itself outputs just one positive real-value; a *scalar*):

We then can solve to obtain a solution that minimises the loss which in turn makes our model perform the required calculations to produce the desired output t.

From this point we can adopt any of the following approaches to train our model, (they will be covered in subsequent sections but we outline them here):

1- Either solve the equation $\nabla\overline{J^2}(\mathbf{w}) = 0$ directly through the least squares method to obtain optimal weights $\mathbf{w}^*$.
2- Or take a numerical approach by starting from any initial weights and moving gradually towards the minimal weights $\mathbf{w}^*$ in each iteration. In each iteration the weights are changed proportional and

opposite to the gradient $\nabla\bar{J}^2(\mathbf{w})$. This approach is called gradient descent and in turn can be performed in any of the following ways:

a. Batch Gradient Descent: where accumulate the gradients of all the training set before taking one update that commit them all at once. This approach can be looked at as taking approach 1) update and dividing it into several iterations. Each iteration sweep through the whole training set and is called an epoch. This approach is impractical for large datasets due to its high demand on memory (i.e. its space complexity is high) even when we use vectorisation.
b. Sequential (also called online) approach that involves *one* data point gradient update at a time. This suits sequential data or data coming from a stream. If the training data is not a stream this approach digests the whole training set but one point at a time. Going through all the data points in the training set is called an epoch.
c. Minim-batch: a compromise between the above two extremes (a and b). Here,
    i. We partition the training set into several mini-batches. Each min-batch involves several data points and all mini-batches have equal size (there might be a smaller remainder final batch that will be treated similarly). For example, if we have 100 data points in our training set and we set the mini-batch size into 25 then we need 4 mini-batches to digest the whole dataset and each epoch will involve 4 mini-batches that constitute 4 iterations.
    ii. In each iteration we collect the updates of all the data pairs $(\mathbf{x}_n, t_n)$ inside the mini-batch b_i and we commit all of them at once.
    iii. In order to mitigate for the bias and variance that might occur due to a particular lucky (or unlucky) batch, we often shuffle the training set.

This approach has the disadvantage that the algorithm might get trapped in local minima. Therefore, to avoid dropping into local minima, we change the weights according to proportion of the gradient not all of it. The percentage of the error is called the learning rate.

Further, we decay the learning rate between one epoch and the other because after each epoch our weights become closer and closer to the weights that is optimal for the entire training set and not only one batch and to avoid overshooting the amount of changes required to end up in the local minimum we reduce the learning step in later epochs. Decaying the weights can be done in several ways. You will discuss these further in the machine learning module.

The above approaches can be applied on any numerical machine learning technique that is based on optimising a loss function and not only for linear regression. In fact unless the dataset is really large, it is excessive to utilise a mini-batch approach for linear regression since the model is too simple and the parameters are linear in the dimensionality of the dataset under consideration.

Please watch the following video on loss function optimisation.

2.12 Batch Learning: The Least Squares for Linear Regression Models

In this section we will *minimise* the mean sum of *squares* by solving the gradient equation directly. This is called the least squares and is a well-known basic method for regression. Understanding it will pave the way to understanding the basic ideas of learning in machine learning. We take the derivative of our loss function and set it to 0 to obtain the best setting that makes our loss minimal. We can actually either start from the non-vectorised or the vectorised from of the cost function. It is easier to use the latter for the least squares while it is easier to use the former for gradient methods.

Earlier we saw that the loss function can be written as norm as follows:

$$\bar{J}^2 = \frac{1}{2N}\|\mathbf{t} - \boldsymbol{y}(\mathbf{X}, \mathbf{w})\|^2$$

$$\bar{J^2} = \frac{1}{2N}\|\mathbf{t} - \mathbf{X}\mathbf{w}\|^2$$

By taking the gradient and setting it to 0 we get

$$\nabla\bar{J^2} = \frac{2}{2N}\mathbf{X}^\top(\mathbf{t} - \mathbf{X}\mathbf{w}) = 0$$

$$\mathbf{X}^\top\mathbf{X}\mathbf{w}^* = \mathbf{X}^\top\mathbf{t}$$

The above is called the normal equation of the least squares. By solving this equation we obtain the final solution that optimise the loss function in other words, the best weights that fit the data. The solution is given as

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{t} \quad (1.)$$

The matrix $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ is called the Moore-Penrose pseudo-inverse of the matrix $\mathbf{X}$. The name reflects the fact that this form is a generalisation of the concept of matrix inverse from square matrices (the common one) to a non-squared matrices. Nevertheless, the above closed form solution is better to be performed in different precedence than that of the Moore-Penrose pseudo-inverse, to impose a slightly better efficiency of calculations as follows:

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{t}) \quad (2.)$$

we have surrounded the operation $(\mathbf{X}^\top\mathbf{t})$ with brackets to impose its precedence. This is because calculating $(\mathbf{X}^\top\mathbf{X})^{-1}$ and then multiplying the result by $\mathbf{X}^\top\mathbf{t}$ is computationally cheaper than calculating $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ and then multiplying is by vector $\mathbf{t}$. We talk more about this in the next section.

The above gives us a closed form solution for $\mathbf{w}^*$. Closed form solutions are not always available for a machine learning or data mining task. Their existence facilitate more analysis and insights into the problem. Some problems might not have a closed form solution formula, however we can still estimate the solutions numerically. Sometimes also closed form solutions can be impractical for big datasets due to their high computational demands. An example is $(\mathbf{X}^\top\mathbf{X})^{-1}$ the inverse of the matrix $(\mathbf{X}^\top\mathbf{X})$. Finding the inverse of a matrix is an expensive operation and its complexity is $\mathcal{O}(D^3)$ and can be reduced to $\mathcal{O}(D^{2.376})$ which can be expensive for a very large D (to be prices it is $\mathcal{O}(\bar{D}^3)$ where $\bar{D} = D + 1$).

Below we show the Least Squares algorithm for regression, which returns the optimal solution for a linear model.

Algorithms 1: Least Squares for Linear Regression Model

Input:

Input set: design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is of size D

Labels set: vector $\mathbf{t} = [t_1, \dots, t_N]^\top$ each t_n is a scalar

Output: $\mathbf{w}^*$ optimum weights; a vector of size $D + 1$

LS_LRegress($\mathbf{X}, \mathbf{t}$):

$\mathbf{X} = [\mathbf{1}_N, \mathbf{X}]$ # add dummy feature to the design matrix

$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}(\mathbf{X}^\top\mathbf{t})$

Return $\mathbf{w}^*$

Please watch the following video on the least squares.

2.12.1 Complexity of the Least Squares

We refer to the computational costs (of how many primitive operations a process costs) as time complexity. Space complexity focuses on how much memory (computational space) a process needs. Here we are mainly talking about time complexity. For example, the complexity of multiplying a vector of size N with a row of size N costs $N \times N$ since a processor has to perform $N \times N$ multiplication operations. Estimating the time using number of operations provides a better reference in terms of time than actual time in seconds or milliseconds

since machines varies greatly in processing power. We largely study a set of operations costs in terms of the more costly operations and we refer to this using the big $\mathcal{O}$ notation which ignores the small pieces of the calculations and concentrate on the dominant operations that takes the longest. For example if an operation costs $2N^2 + N$ then its big $\mathcal{O}$ is $\mathcal{O}(N^2)$. We refer to the vector to vector complexity as $\mathcal{O}(D^2)$ if vectors are of size D or as $\mathcal{O}(N^2)$ if vectors are of size N. For a matrix of size $N \times D$ and a vector of size D the multiplication operation costs $\mathcal{O}(D^2 \times N)$.

We refer to the size of the extended input space, that comprises the original input space along the side with the dummy input $x_0 = 1$ as $\overline{D} = D + 1$. The matrix $(\mathbf{X}^\top\mathbf{X})^{-1}$ is the inverse of the matrix $\mathbf{X}^\top\mathbf{X}$ both of which is of size $\overline{D} \times \overline{D}$. Nevertheless, we will suffice by studying the complexity using D since the difference is minor and to promote simplicity.

$\mathbf{X}$ is a matrix of size $N \times D$, so the matrix ($\mathbf{X}^\top\mathbf{X}$), and its inverse, eliminate the dimension N related to the number of data points. Of course the dimension N stays in the solution due to $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ which is an $N \times \overline{D}$ matrix and is called the Moore-Penrose pseudo –inverse of the matrix $\mathbf{X}$. However, in general it's more efficient to do the calculation $\mathbf{X}^\top\mathbf{t}$ which results in a vector and multiply the results by the inverse $(\mathbf{X}^\top\mathbf{X})^{-1}$instead of calculating $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ and then multiply it by $\mathbf{t}$ since this avoids a matrix to matrix multiplication.

The full calculation $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{t}$ result in a vector of size $\overline{D}$. It can be performed in two ways:

1- Either $(\mathbf{X}^\top\mathbf{X})^{-1}$ and then multiplying by $\mathbf{X}^\top\mathbf{t}$
2- Or calculating $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ and then multiplying by $\mathbf{t}$

Both of them gives the same results but the first is more efficient than the second. The effect is specifically noticeable when D is large. To see why note that both of them calculate $(\mathbf{X}^\top\mathbf{X})^{-1}$ so initially we will not factor its complexity in the comparison to keep things simple.

1- The distinguished cost of the first method is:	2-The distinguished cost of the second method is:
- calculating $\mathbf{X}^\top\mathbf{t}$ costs $\mathcal{O}(D \times N)$ and results in a vector $\boldsymbol{z}$ of size D - multiplying$(\mathbf{X}^\top\mathbf{X})^{-1}$ by $\mathbf{X}^\top\mathbf{t}$ costs $\mathcal{O}(D^2)$ and results in a vector of size D - total complexity is $\mathcal{O}(D^2) + \mathcal{O}(D \times N)$.	- calculating $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ costs $\mathcal{O}(D^2 \times N)$ and results in a $D \times N$ matrix - multiplying the $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$by vector $\mathbf{t}$ costs $\mathcal{O}(D \times N)$ and results in a vector of size D - total complexity is $\mathcal{O}(D^2 \times N) + \mathcal{O}(D \times N)$.
If $N > D$ then the results $\mathcal{O}(D \times N)$	If $N > D$ then the results $\mathcal{O}(D^2 \times N)$

Hence, the first method is preferred over the second method.

In comparison to the cost of calculating $(\mathbf{X}^\top\mathbf{X})^{-1}$ is as follows

- $\mathbf{X}^\top\mathbf{X}$ costs $\mathcal{O}(D^2 \times N)$ and results in a matrix of size $D \times D$
- $(\mathbf{X}^\top\mathbf{X})^{-1}$ costs $\mathcal{O}(D^3)$ and results in a matrix $\mathbf{F}$ of size $D \times D$
- total cost is $\mathcal{O}(D^2 \times N) + \mathcal{O}(D^3)$ and if we have that $N > D$ then the final complexity is $\mathcal{O}(D^2 \times N)$.

So if we put all the operations together then the least squares costs $\mathcal{O}(D^2 \times N)$ which appears to cancel out the gain due to first method. Nevertheless, it is still a better practice to add the brackets to enforce some complexity saving whenever possible. In practice, you will normally use a solver to perform the lease squares. It is even available in Excel.

2.13 EXERCISE

Let us see how to implement and apply a simple least squares solution on synthetic 2d data, on the following Jupyter notebook.

2.14 EXERCISE

In the following Jupyter notebook, we see an example of the difference between the limitation imposed by the dataset and the limitation imposed by the nature of the model.

2.15 APPROXIMATE SOLUTIONS: GRADIENT DESCENT

For linear models we saw that we can find analytically a solution via the normal formula by setting the gradient to 0 and solve with respect to $\mathbf{w}$, such solutions are either not available when we deal with non-linear optimisation or is not desirable due to efficiency requirements. Even if an analytical close form solution is available, the complexity of finding the least squares is $\mathcal{O}(D^2 \times N)$ which is quite expensive when N is reasonably large.

In such cases, it is desirable to find an *approximate solution* for the problem (i.e. an approximation for $\mathbf{w}^*$) to come as close as possible to the minimum *without* necessarily finding the *exact solution*. Algorithms that try to achieve this are called approximation algorithms, you will study several of these in the Algorithms Module including greedy, local search and dynamic programming algorithms. In our case, we will utilise an important and pervasive approximation algorithm that is utilised throughout machine learning. It is not necessary the best approximation algorithm but it is the simplest to understand and to implement.

This optimisation algorithm is called the gradient descent or steepest descent. This technique aims at iteratively finding the minimum of a function (the loss function $\overline{J^2}$ in our case). The algorithm starts from any point on the surface of the loss function (i.e. by taking a random initial value for $\mathbf{w}$) and then it takes small steps in the direction of the minimum of the function by changing the weights gradually in each step. The direction of the point $\mathbf{w}^*$that minimise $\overline{J^2}$ from any point $\mathbf{w}^{(\tau)}$ is always opposite to the gradient of the function at this point $-\nabla\overline{J^2}(\mathbf{w}^{(\tau)})$. This is because the gradient of a function always points in a direction opposite to the minimum.

Here we are talking about a minimum, often complex loss functions have several minima so we will come back to this idea later when we move to the non-linear models towards the end of the unit. We are taking small steps towards the minimum because taking large steps lead to overshooting the minimum or oscillating around it. The size of the step (denoted as η) is called the learning rate because it represents how fast a model can learn the solution of the problem. Gradient descent is a numerical optimisation technique so it is an iterative technique that keep working though iterations until it reaches a good enough approximate solution. Reaching a minimum is called *convergence* (a well known concept in calculus).

Let us start by the basic gradient descent update which takes the following form:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta_\tau \nabla\overline{J^2}(\mathbf{w}) \tag{3.}$$

where $\mathbf{w}^{(\tau)}$ represents the weight vector at iteration τ and *this is not exponentiation*. η_τ is a learning step that can be varied between iterations to make the algorithms responsive to changes in the loss function terrain. The basic form of the GD is given below.

Algorithm 2: Approximation Algorithm: Gradient Descent.

Input:

Input set : $\mathbf{X} = \{\mathbf{x}_1, \dots \mathbf{x}_N\}$ | *Training set*

Labels set : $\mathbf{t} = \{t_1, \dots t_N\}$

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$

GD($\mathbf{X}, \mathbf{t}$):

Initialise $\mathbf{w}$

For $\tau = 1 : \tau max$

$\mathbf{w} = \mathbf{w} - \eta_\tau \nabla \overline{J^2}(\mathbf{w}, \mathbf{X}, \mathbf{t})$ # obtain the gradient of the loss for current $\mathbf{w}$ on the entire training set

Return the final solution $\mathbf{w}$

There are some optimisation techniques that give us how to vary the learning rate η_τ but we have not shown this here for simplicity. Also note that both η_τ and τmax should be inputs to the algorithm but we omit this to promote simplicity.

In the next section we will see how to apply the gradient descent algorithm on the linear regression model.

2.16 Batch Gradient Descent for Linear Regression Models

We will take the gradient of the cost function directly without using it vectorised form but later we develop a vectorised version. We have seen already in a previous section that the gradient of the linear regression loss function takes the form:

$$\overline{J^2} = \frac{1}{2N} \sum_{n=1}^{N} J_n^2$$

$$\nabla \overline{J^2}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} \nabla J_n^2(\mathbf{w})$$

$$\nabla \overline{J^2}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} 2 \nabla J_n(\mathbf{w}) J_n(\mathbf{w})$$

$J_n(\mathbf{w}) = (t_n - \mathbf{w}^\top \mathbf{x}_n)$ and $\nabla J_n(\mathbf{w}) = -\mathbf{x}_n$ hence

$$\nabla \overline{J^2}(\mathbf{w}) = -\frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n (t_n - \mathbf{w}^\top \mathbf{x}_n) \tag{5}$$

This is an important formula that we will refer back to often. To get a taste of what this gradient entails, we show below what is involved in it

$$\nabla \overline{J^2}(\mathbf{w}) = -\frac{1}{N} \left(\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_D \end{bmatrix}_1 J_1 + \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_D \end{bmatrix}_2 J_2 + \cdots + \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_D \end{bmatrix}_N J_N \right) = 0$$

Therefore, the gradient descent algorithm for linear regression model, which acts on the entire training set, takes the form:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \nabla \overline{J^2}(\mathbf{w})$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \frac{1}{N} \sum_{n=1}^{N} \eta \mathbf{x}_n (t_n - \mathbf{w}^\top \mathbf{x}_n) \tag{4.}$$

The number of iterations that we need to take in order to reach the minimum depends on η. The smaller η is the more iterations we need to take, however we need to strike a balance here because if η is too big then the algorithm might either oscillate or completely diverge (go away from the minimum). η is almost always less than 1, a reasonable value of $\eta = 0.01$ for linear regression. For other more complex models η may need to take much smaller values. Each sweep through the entire dataset is called an epoch and this is a hyper parameter that we need to set, often between 10 and 100.

Algorithm 3: Approximation Algorithm: Batch Gradient Descent (GD) for Linear Regression Model, without vectorisation.

Input:

Input set : $\mathbf{X} = \{\mathbf{x}_1, \dots \mathbf{x}_N\}$ each $\mathbf{x}_n$ is a vector of size D | *Training set*

Labels set : $\mathbf{t} = \{t_1, \dots t_N\}$ each t_n is a scalar

η: The learning rate

$epcs$: Max number of epochs

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$; a vector of size $D + 1$

GD_LRegress$(\mathbf{X}, \mathbf{t}, \eta, epcs)$:

Initialise $\mathbf{w}$

For epoch $= 1: epcs$

$\mathbf{w}' = \mathbf{0}_{D+1}$

For $n = 1: N$

$\mathbf{x}_n = [1, \mathbf{x}_n^\top]^\top$ # add a dummy attribute for each $\mathbf{x}_n$

$\boldsymbol{w}' = \boldsymbol{w}' + \eta \boldsymbol{x}_n (t_n - \boldsymbol{w}^\top \boldsymbol{x}_n)$ # accumulated the changes without committing them

$\mathbf{w} = \mathbf{w} + \frac{1}{N}\mathbf{w}'$ # now commit the changes

Return the final solution $\mathbf{w}$

Note how we accumulate the changes inside a temporary vector $\mathbf{w}'$ (this is just a vector of size $D + 1$) in an epoch and we commit at the end of the epoch. This is why it is called a batch gradient descent algorithm; we are waiting till the end of iterating through full batch of the dataset and then we change $\mathbf{w}$, i.e. *we do not change* $\mathbf{w}$ *during the epoch*.

This form of batch gradient descent does not take advantage of vectorisation and is slow. For large dataset vectorising the implementation is impractical. However, later on when we talk about mini-batch stochastic gradient descent we will see a way to make a good compromise that will allow us to utilise vectorisation. See this paper.

Please watch the following video on gradient descent.

2.17 Sequential Learning: Stochastic Gradient Descent for Linear Regression Models

Batch learning algorithm such as LS Regression or Batch Stochastic Gradient Descent take into account the entirety (the whole batch) of the dataset at once. No intermediate learning occurs. Another way to minimise the loss function is to gradually change the weights towards minimising the loss function instead of going all the way according to the sum of the errors. This is called sequential learning. There are several advantages for this approach. The most obvious advantage is that it allows for a stream of data to be fed into a system and the system can learn live as the data arrives from the stream. The main advantage is that learning can occur immediately for any fed sample and we do not need to wait to see the entirety of the dataset to learn a model.

Here we need to understand the concept of a learning rate or learning steps denoted as η. This hyper parameter specifies how much of the individual step error we want to take into account. In simple linear models this will

not make a difference and in fact if assumed that the loss function is convex i.e. it has a global optimum then we can go all the way and adopt the entirety of each step error $(\mathbf{w}^\top \mathbf{x}_n - t_n)$ offline without changing the weights in each step. However, when the concavity of the loss function (existence of global optimum) is not guaranteed and when the loss function has several local optima some of which are really slight valleys (or when it is infested with local optima) then adopting the full error $t_n - \boldsymbol{y}(\mathbf{x}_n)$ is not a good idea. This is because it will force the model to fall into the nearest local minimum and consequent updates are spent on moving out or into local minima. Bearing in mind that the data is noisy anyway, we would want to utilise the learning step for our benefit to reduce the effect of the noise and help avoid the problem of overfitting. Essentially we replace the loss function J^2 by J_n^2, so after a data point becomes available, we update according to:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \frac{1}{2} \nabla J_n^2$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta(-\mathbf{x}_n)(t_n - \mathbf{w}^{(\tau)^\top} \mathbf{x}_n)$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \eta \mathbf{x}_n (t_n - \mathbf{w}^{(\tau)^\top} \mathbf{x}_n) \qquad (5.)$$

Where τ rpersents the iteration (or the time step) and η is the learning rate parameter which should be carefully chosen so that it does not lead to divergence or oscillation of the algorithms. This is because effectively we are accounting for only a very small part of the gradient and we need to leave room for gradients of other data points to take effect in later iterations. Note that we added a (-) to go against the gradient direction which will make the changes go in the direction that will minimise the error. A reasonable choice of η is to make it proportional to the number of expected data points.

Stochastic Gradient Descent (SGD) suffers from several issues, mainly its high variance and its short sighted look into the loss function terrain by considering just one or few data points. The gradient as a vector, has two essential properties that will direct our search for the minimum of the loss. The first is its direction and the second its magnitude. In general, variants of SGD optimisation use the gradient to specify the *direction of changes* in the weight vector, they vary in the way they estimate how much of the magnitude of the gradient to be considered (the *amount of change*). Other optimisation techniques use different direction than that of the gradient altogether, ex. the conjugate gradient of the loss, these however lie outside the scope of our coverage. Vanilla SGD just uses the learning rate η to uniformly take a proportion of the gradient magnitude not all of it. But this makes it difficult to calibrate the learning rate because it has to fit all the different terrains of the loss function, so we normally end up reducing η and taking lots of steps to converge to the minimum (of course there local and global minimum, but let us not differentiate for a moment).

There are a lot variations for the stochastic gradient descent. Mainly they are concerned with tailored learning rate that is responsive to the geometrical aspects of the loss function. (a) For example we can employ the idea of momentum which changes the learning rate according to a linear combination of the gradient at the current step with the previous update: called Momentum. (b) We can also employ the idea of averaging the weights from all past steps which do not play around with the learning rate, it replaces it by averaging of the weights themselves: called Averaging. (c) We can adapt the learning rate for each individual weight. Once way to do that is by dividing keeping a running average for each weight and then we divide the learning rate of each individual weight by the weight running average: called RMSProp.(d) we can also make the learning rate changes for each parameter according to how often the parameter is updated. This particularly applicable for when we have many zeros in our inputs for several attributes quite often- technical term is *sparsity*: this approach called Adagrad. (f) Another idea is to keep a running average of each individual weight but involve both the gradient and the square of the gradient (second moment) of the weight as well as the idea of sparsity in Adagrad: this approach is called adaptive moment-Adam optimisation and is very popular in deep learning. And plenty more, see this paper for more details. Standard optimisation libraries allow you to choose what type of optimisation you want to use without having to implement it from scratch.

We will refer to all of these strategies by using a normalisation vector $\overline{\boldsymbol{N}}$ that can represent any of the above strategies. To cover the per-weight learning rate adaptation methods, such as the Adagrad and Adam, we need component-wise multiplication (denoted as $\circ$). We can write $\frac{1}{\overline{\boldsymbol{N}}} \circ \mathbf{x}_n$ to express that we are adjusting the weights components differently, this is a crude way of describing these optimisation but promote simplicity.

For linear regression we can set η to relatively high value such as 0.3 to take into account a good chunk of the errors since we know that the loss function is convex. The loss function is convex since we are taking the squares of weights with no activation function (we will talk more about activation function later in numerical classification). Still, we might want to use a reduced learning rate to cancel some of the noise of the data. Recall that any data will always have some noise in it and reducing the learning rate helps in reducing the risk of model overfitting and helps in reducing the effect of the noise. This is especially relevant when we talk about data streaming where we do not want to take into account all the error of the current input so as not undo completely some previous learning. Also this brings us to the idea of input normalisation which should be used if possible for input coming from data streams.

The idea of a learning step is pervasive in machine learning and can be powerful in tackling some of the overfitting issues that arise when dealing with regression. For example, we can anneal (gradually reduce) the learning step in each step or every b steps in order to hinge towards a global optima when the loss function is not convex.

Another important reason to use SG Regression is that it is often faster to converge in practice than LS when we deal with more complex techniques and is more efficient to implement when the size of the dataset or its dimensionality is intractable. This is only for extremely large dataset but something worth putting in mind for future reference.

We can also apply SGD regression on a static dataset, we get a similar results to the LS regression. However, you should be bear in mind that there are quite subtle differences between the two algorithms (the LS Regression and SG Regression). Let us see first SGD Regression on a dataset below.

Algorithm 4: Sequential Learning: Stochastic Gradient Descent for Linear Regression Model.

Input:

Input set : $\mathbf{X} = \{\mathbf{x}_1, \dots \mathbf{x}_N\}$ each $\mathbf{x}_n$ is a vector of size D | *Training set*
Labels set : $\mathbf{t} = \{t_1, \dots t_N\}$ each t_n is a scalar

η: The learning rate

$epcs$: Max number of epochs

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$; a vector of size $D + 1$

SGD_LRegress1($\mathbf{X}, \mathbf{t}, \eta, epcs$):

Initialise $\mathbf{w}$

For epoch $= 1: epcs$

For $n = 1: N$

$\mathbf{x}_n = [1, \mathbf{x}_n^\top]^\top$ # add a dummy attribute for each $\mathbf{x}_n$

$\boldsymbol{w} = \boldsymbol{w} + \eta \boldsymbol{x}_n (t_n - \boldsymbol{w}^\top \boldsymbol{x}_n)$ # commit the changes in every step

Return the final solution $\mathbf{w}$

Comparing Algorithm 3 and Algorithm 4. It becomes clear that in Algorithm 4 the weights fixed *during* learning. In contrast Algorithm 3 accumulates all the changes of the weights and applies them all at once.

2.18 MINI-BATCH LEARNING: MINI-BATCH STOCHASTIC GRADIENT DESCENT FOR LINEAR REGRESSION MODELS

Mini-batch SGD algorithm can be used to reach a compromise between sequential and batch gradient methods. To achieve this: the weights changes can be accumulated on-the-sides not for the entire training set but for a limited number of steps b and be committed every b steps. On the extremes when we set $b = N$ (wait until all the data finishes) we get an algorithm that is equivalent to the batch gradient descent, while when we set $b = 1$ we get the stochastic gradient descent. So b parametrise a middle ground approach for both cases. Below we show the algorithm.

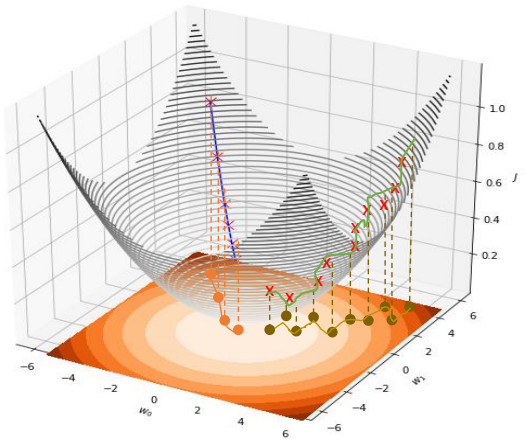

Figure (1.6): SGD algorithm behaviour: SGD takes gradual steps towards the minimum of the loss function by following the gradient of the loss. The line shows an example of the paths of a batch gradient descent (blue on the loss surface function and its projection is orange on the loss contour) and stochastic gradient descent algorithms (green on the loss surface function and brown on the loss contours).

Algorithm 5: Mini-Batch Stochastic Gradient Descent Updates for Linear Regression Model: Vanilla implementation that can be sped up- see next algorithm.

Input:

Input set : $\mathbf{X} = \{\mathbf{x}_1, \dots \mathbf{x}_N\}$ each $\mathbf{x}_n$ is a vector of size D | *Training set*

Labels set : $\mathbf{t} = \{t_1, \dots t_N\}$ each t_n is a scalar

η: learning rate

b: mini-batch size (specifies how frequently we want to update the weights $\mathbf{w}$).

$epcs$:Max number of epochs

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$; a vector of size $D + 1$

SGD_LRegress2($\mathbf{X}, \mathbf{t}, b, \eta$):

Initialise $\mathbf{w}$ and set $\mathbf{w}' = \mathbf{0}$

For epoch $= 1: epcs$

For $n = 1: N$

$\mathbf{x_n} = [\mathbf{1}, \mathbf{x}_n^\top]^\top$ # add a dummy feature for each $\mathbf{x}_n$

$\mathbf{w}' = \mathbf{w}' + \eta \mathbf{x}_n (t_n - \mathbf{w}^\top \mathbf{x}_n)$

If $n\%b == 0$ # there is a better condition see the discussion below

$\mathbf{w} = \mathbf{w} + \frac{1}{b}\mathbf{w}'$

$\mathbf{w}' = \mathbf{0}$

Return the final solution $\mathbf{w}$

The above algorithm utilises the modulus function $n\%b$ which will give us the remainder of a division of the data point count n by b the batch size. When this function $n\%b == 0$ it means that b number of steps has elapsed. For example if $N = 90$ and we set $b = 10$ then the weights $\mathbf{w}$ (not $\mathbf{w}'$) will be updated every 10 steps and the updates will be executed 9 times. Note that we accumulate all the changes inside each 10 steps in $\mathbf{w}'$ and we commit them at the 10th step.

If we have $N = 94$ and we set $b = 10$ then the last 4 data points will be left if we just use the condition $\%b == 0$. Therefore we can adjust as follows:

If $n\%b == 0 \ or \ n == N$:
  If $n\%b \neq 0$ then $b' = n\%b$ # accommodate the last few points that do not fit a batch
  Else $b' = b$
  $\mathbf{w} = \mathbf{w} + \frac{1}{b}\mathbf{w}'$
  $\mathbf{w}' = \mathbf{0}$

the condition $n\%b == 0 \ or \ n == N$ is used to accommodate the last few points that cannot form a full batch. For the same reason we use $b' = n\%b$ instead of b, which will yield b when $n\%b == 0$ and the remainder of the batch (4 in our example) otherwise when n==N. we have not include this snippet to keep the algorithm simple

Also we use the weights $\mathbf{w}$ in our output estimation $\mathbf{w}^\top \mathbf{x}_n$ in the update rule $\mathbf{w}' = \mathbf{w}' + \frac{1}{N}\eta(t_n - \mathbf{w}^\top \mathbf{x}_n)\mathbf{x}_n$ this is deliberate to guarantee stability. Mini-batch stochastic gradient descent reduce the variance caused by stochastic gradient descent and hence help stabilise the learning process.

2.18.1 Faster Mini-Batch SGD

The above algorithm is a vanilla algorithm of an SGD mini-batch that can be sped up. The algorithm slows down the process by processing the data points individually in each batch. There is a better approach, can you guess it, take a minute or two to think about it…Ok, it is vectorisation. The idea is as follows. We assume that the size of the batch is b and for simplicity that N is divisible by b the number of mini-batches as $q=N/\ b$ then. Now we can adopt one of two strategies:

1- Define a partition of the training set $\{\mathbf{X}, \mathbf{t}\}$ into a set of q mini-batches as follows:

$$\mathbf{X} = \begin{bmatrix} \overbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,D} \\ x_{2,1} & x_{2,2} & \dots & x_{2,D} \\ \vdots & \vdots & \vdots & \vdots \\ x_{b,1} & x_{b,2} & \dots & x_{b,D} \end{bmatrix}}^{\mathbf{X}_1} \\ \overbrace{\begin{bmatrix} \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ x_{2b,1} & x_{2b,2} & \dots & x_{2b,D} \end{bmatrix}}^{\mathbf{X}_2} \\ \vdots \\ \overbrace{\begin{bmatrix} \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ x_{qb,1} & x_{qb,2} & \dots & x_{qb,D} \end{bmatrix}}^{\mathbf{X}_q} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \vdots \\ \mathbf{X}_q \end{bmatrix} = \left[\mathbf{X}_1^\top, \mathbf{X}_2^\top, \dots, \mathbf{X}_q^\top\right]^\top, \quad \mathbf{t} = \begin{bmatrix} \overbrace{\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_b \end{bmatrix}}^{\mathbf{t}_1} \\ \overbrace{\begin{bmatrix} \dots \\ \dots \\ t_{2b} \end{bmatrix}}^{\mathbf{t}_2} \\ \vdots \\ \overbrace{\begin{bmatrix} \dots \\ \dots \\ t_{qb} \end{bmatrix}}^{\mathbf{t}_q} \end{bmatrix} = \begin{bmatrix} \mathbf{t}_1 \\ \mathbf{t}_2 \\ \vdots \\ \mathbf{t}_q \end{bmatrix} = \left[\mathbf{t}_1^\top, \mathbf{t}_2^\top, \dots, \mathbf{t}_q^\top\right]^\top$$

where $\mathbf{X}_\tau \quad \tau = 1{:}\, q$ is a matrix of size $b \times D$ and $\mathbf{t}_\tau \quad \tau = 1{:}\, q$ is a vector of size $b \times 1$. *We refer to both as a mini-batch of size* b.
so now we sweep through all the mini-batches one after the other in each *iteration* to cover the whole training set, we call this an *epoch*. After each epoch we need to *shuffle* the dataset (or equivalently shuffle the membership assignment in the mini-batches which is what we always do in the implementation). Note that our weights estimation are expected to improve from one batch to another. Moreover, the weights error (cost function) is expected to improve from one epoch to another since we employ normally a learning rate<1. This strategy guarantees stability and efficiency at the same time. We will refer to this strategy as shuffling and partitioning strategy. Note that in this strategy each data point must appear once in one of the min-batches in each epoch.

2- The second strategy is just to draw a random mini-batch $\mathbf{X}_\tau$ and $\mathbf{t}_\tau$ of size b without partitioning which is even more efficient than the first strategy. This is drawing with replacement so the same data point can appear in multiple mini-batches inside the same epoch or may not appear at all in any mini-batch (but the same data point does not appear more than once in the same mini-batch). In this strategy we can decouple the number of mini-batches from the size of the training set, so $q \geq N/\, b$ but it can still provide a general guide on the number of iterations (or mini-batches) the algorithm will go through. This strategy is faster in implementation but it may lead to less stability than the first strategy, so it is more preferred in large scale learning. It all depends on selecting the trade-off that suits the application.

Both strategies are amenable for parallelisation, where we feed parts of the process to a different processor. Clearly, we can feed parts of a batch to different core processor but we cannot give different batches to different processor because the idea of the mini-batch is to update the weights directly after each batch and then use the new weights in the next batch. If we give away on this idea then we can distribute different batches on different processors and collect and aggregate the changes afterwards. Such a process will take us back to batch gradient descent. So the implementation is similar to a min-batch but the effect is a batch gradient descent.

Both strategies are equivalent in the extremes: when b=1 or when b=N. when b=1 the SGD mini-batch algorithm turn into an SGD algorithm. When b=N the SGD mini-batch algorithm becomes a batch SGD algorithm which in turn approximates the least squares but without the overhead of matrix inversion.

Below we show the final mini-batch algorithm. We have left which strategy to adopt open in the algorithm and we have stated it as 'Select a mini-batch $\mathbf{X}_\tau, \mathbf{t}_\tau$ of size b from $\mathbf{X}, \mathbf{t}$: $q \geq N/\, b$ (randomly or by shuffling and partitioning)'. So, this algorithm is actually two different algorithms depending on whether we choose to shuffle and partition the training set or just simply draw a random mini-batch (with replacement).
We normally decay the learning rate in order to prevent zigzagging around the minimum of the cost function when we start by a high learning rate or to fine tune our final weights.

Algorithm 6: Mini-Batch Stochastic Gradient Descent Updates for Linear Regression Model: with vectorisation.

Input:

Input set: design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is a vector of size D | *Training set*

Labels set: vector $\mathbf{t} = [t_1, \dots, t_N]^\top$ each t_n is a scalar

b: The mini-batch size (specifies how frequently we want to update the weights $\mathbf{w}$).

η_0: Initial learning rate

$epcs$: Number of epochs

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$; a vector of size $D+1$

SGD_LRegress$(\mathbf{X}, \mathbf{t}, b, \eta_0, epcs)$:

Initialise $\mathbf{w}$ and $\eta = \eta_0$

For epoch $= 1{:}\, epcs$

For iteration $\tau = 1{:}\, q$ # $q \geq N/b$

Select a mini-batch $\mathbf{X}_\tau, \mathbf{t}_\tau$ of size b from $\mathbf{X}, \mathbf{t}$ # by sampling or by shuffling & partitioning

$\mathbf{X}_\tau = [\mathbf{1}_b, \mathbf{X}_\tau]$ # add dummy feature to the mini-batch

$\mathbf{w} = \mathbf{w} + \eta \frac{1}{b} \mathbf{X}_\tau^\top (\mathbf{t}_\tau - \mathbf{X}_\tau \mathbf{w})$

Decay η # if necessary: ex. η=0.9×η

Return the final solution $\mathbf{w}$

Stochastic gradient descent, especially the minim-batch version is an excellent tool to tackle large-scale learning and has received recently a considerable attention. Large scale learning is learning from a large dataset with a huge amount of instances available. In such a case, we cannot expect to train on the whole dataset because it is way beyond a single machine capabilities or even the capabilities of a medium cluster of machines. Stochastic gradient descent is an excellent tool because it allows us to simply tackle as much as we can digest in our available hardware. Of course there are several augmentations to the simple (vanilla) SGD in terms of cleverer selection of the data to be processed which goes beyond the basic form presented here. At the same time, parallelisation techniques can be employed to promote efficient parallelised implementation of amortised complexity of $\mathcal{O}(\log_r N)$ where N is the dataset size or the size of the processed data that can be pulled from a data lake or a data centre and r is the number of parallel processors available to the algorithm. You can have a look at this paper for more details.

It should be stressed here also that SGD is sensitive to feature scaling and it is recommend to rescale all features in the dataset, as we have discussed in unit. This is to avoid one feature overwhelming other features by its high values, especially when the attribute value do not vary that much. The simplest way is to rescale by dividing by the max value of the feature, or to subtract the minimum and divide by the difference between the minimum and the maximum of the feature. We get the min and max either by looking into the dataset or by understanding the domain of the feature. Consulting domain experts who work with the data is also useful to understand the nature of the dataset that we are tackling.

The above algorithm can be easily adapted when we are dealing with a data stream, all what we need to do is to accumulate $\mathbf{X}_\tau$ as the data arrives until it is of the required size b, and we can even vary the size b itself between different iterations. Note that when $b = N$ the algorithm goes back to a vectorised batch stochastic gradient descent which can be applied when the dataset size permits, the resultant weights are still an approximation even if it might be very close to the optimum solution $\mathbf{w}^*$.

To summarise, we emphasise here, contrary to what one might expect, stochastic and mini-batch stochastic gradient descent converge faster than batch gradient descent in practice. This is due to several reasons. One reason is that both stochastic gradient algorithms infuse noise in the update which is quite useful to escape local minima. Another reason is that by nature stochastic algorithms are faster to execute and they execute several

updates per clock time in comparison with batch gradient which keeps accumulating the gradients on the side until it sweeps through the whole training set. Assuming that the training set is finite but large, then the roughness of stochastic updates outperforms the more exactness of batch gradient. A third reason is that all gradient descents, even the batch one, do not point exactly to the global minimum, instead they roughly point to a direction that will lead us to the minimum. Therefore, it does not make sense to spend a lot of computational power (as in the batch GD) to try to improve the gradient by considering more and more points until we consume the whole training set. Because even then the gradient is not necessarily pointing to the exact direction of the minimum, especially for complex loss function. Although linear regression loss function is quadratic and has a global minimum, nevertheless these issues can still be seen and you can examine them in the next exercise.

Please watch the following video on stochastic and mini-batch gradient descent for linear regression.

2.19 Evaluating a Regression Model via R2 and the RMSE

Besides using the loss mean of squared errors to drive the learning process, we can utilise it to measure the performance of the learned mode. The problem with this approach is that the squares give inflated estimation of the error. To deal with this, we can use the root mean squared error (RMSE) instead to measure the model performance. Please distinguish between performance measure and the loss function, although they are related we often use easy to differentiate continuous function as the loss, while for regression performance we can use any whole-measure to give a point evaluation for the model. RMSE is given as:

$$RMSE = (MSE)^{0.5}$$

Another possibility, when we want to be able to compare the performance of the model in different datasets, (or between different models and dataset) is to use the coefficient of determination of R^2 metric. R^2 is defined as

$$R^2 = 1 - \frac{\sum_{n=1}^{N}(t_n - y_n)^2}{\sum_{n=1}^{N}(t_n - \bar{t})^2}$$

where $\bar{t}$ is the mean of the target. The numerator is referred to as SSR or SSE and is our usual sum of squared errors, while denominator, denoted as SST (sum of squared total), is the sum of squared deviation from the mean (which is the variance of the target times the size of the data under consideration). It represents the sum of squared errors for a base model, where the base model uses the mean of the targets as the prediction for any data points.

R^2 gives an indication of the extent to which our model is better than just using the mean of the target to predict all data points' values. The closer the value is to 1, the better the model is. If the metric is less than 0 then that indicates that it is worse than guessing as per the mean.

2.20 Exercise

See the following Jupyter notebook to compare the performance of vanilla and vectorised SGD, the code is written in pure python code using numpy.

2.21 Comprehensive Example

Please watch the following videos for a comprehensive example that covers the different concepts of this lesson.
Comprehensive Example video 1
Comprehensive example video 2
Comprehensive example video 3

2.22 SUMMARY

In this lesson you have learnt about simple linear regression models, and their coefficient that maps into weights. You have understood the generalisation from one dimension of input into multi-dimensional input space. You have seen how to utilise loss function effectively in order to solve the linear regression problem exactly via least squares, and approximately via the gradient descent. In addition you have learnt how to address sequential learning problems by utilising the ideas of mini-batch learning.

3 LINEAR REGRESSION WITH LINEAR AND NON-LINEAR BASIS FUNCTIONS

Learning outcomes:

After completing this lesson you should be able to:

- generalise linear regression models with basis
- understand different types of basis functions that can be utilised in the generalised linear regression models
- understand the limits of generalised linear models in terms of their ability to model non-linear functions
- perform batch and sequential learning in generalised linear models
- understand overfitting and underfitting in generalised regression models
- understand generalised multi output linear regression models

In the previous lesson we saw how to minimise a loss function of a linear model of the form:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D$$

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{d=1}^{D} w_d x_d$$

But what about if wanted to process the data $\mathbf{x}$ before we try to learn a model? This is called input space mapping, i.e. we would like to map the input space $\mathbf{X}$ to some other space $\mathbf{\Phi}$. We do that when we perform some pre-processing on the input space but also when we want to increase or decrease the dimensionality of the input space. There are numerous advantages of moving from once space to the other in data mining; it all amounts to simplifying or reducing the complexity of the data or it processing.

Learning takes place by adjusting these parameters to make the model produce the desired answers. In simple terms, linear means first order sum. So the above is linear because the model constitutes a linear function in the weights. For example, the following is **not a linear model**: $y(\mathbf{x}, \mathbf{w}) = w_0^2 + w_1^2 x_1 + w_2^2 x_2 + \cdots + w_D^2 x_D$, because we are multiplying each components of x_i by the square of the weight$(w_n)^2$. Similarly, $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1^3 x_1 + w_2 x_2$ is not a linear model. Polynomials of order higher than 1, quadratic, cubic as well as exponential e^x, logarithmic, sin, cos are all **nonlinear** functions).

We still call w_0 the bias or the intercept. Note that given an input space with one input x_1 our linear model can be defined as $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1$

However, the following are all **linear models**:

- $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1^2 + w_2 x_2^2 + \cdots + w_D x_D^2$
- $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1^1 + w_2 x_2^2 + \cdots + w_D x_D^D$
- $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 e^{x_1} + w_2 e^{x_2} + \cdots + w_D e^{x_D}$
- $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 e^{x_1} + w_2 e^{x_2} + w_3 e^{x_1 + x_2} + w_3 e^{x_1 - x_2}$

Remember our note about being linear with respect to the weights. Since these expressions are linear combination of the weights, the linearity of the models holds for all the above examples. Note that in the last example we have 5 weights although the input space has a 2 features x_1 and x_2.

More generally, we can define a linear model using a set of M functions called basis. Each maps an input vector $\mathbf{x} = (x_1, x_2, \dots, x_D)^\top$ into a real value $\phi_m(\mathbf{x})$ i.e. ϕ_m: $\mathbb{R}^{\mathrm{D}} \longrightarrow \mathbb{R}$. Then, we can use each set of basis functions to map the input space into a new input space, and our linear regression model can be expressed as:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1\phi_1(\mathbf{x}) + w_2\phi_2(\mathbf{x}) + \cdots + w_M\phi_M(\mathbf{x})$$

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{m=1}^{M} w_m\phi_m(\mathbf{x})$$

Note that the sum has M elements according to the number of basis functions that we use and we have also a set of M weights (including w_0) instead of the D weights that we had earlier when we were dealing directly with the components of the input space. Note that the linear models without basis becomes a special case of linear models with basis $\phi_m(\mathbf{x}) = x_m$ where $M = D$. So, we will be dealing with the more general definition of the linear model from now on. Note that in effect, the set of basis functions defines a set of components of a vector $\boldsymbol{\phi}(\mathbf{x}) = \big(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \dots, \phi_M(\mathbf{x})\big)^\top$, which in turn means that $\boldsymbol{\phi}$: $\mathbb{R}^{\mathrm{D}} \longrightarrow \mathbb{R}^{\mathrm{M}}$. For brevity, we write $\boldsymbol{\phi} = (\phi_1, \phi_2, \dots, \phi_M)^\top$ when we are not concerned in explicitly stating $\mathbf{x}$.

Moving from input space to feature space has several desired advantages. The main one is that while moving to a high dimensionality may entail some extra processing, it can simplify the model that is needed in order to fit the data. Specifically, in moving to higher dimensionality feature space we hope to map a non-linear relationship between the input and the label into a linear relationship between the features and the label. This trick allows us to employ simpler models and promote speed and efficiency. Even when the relationship does not become linear, moving to a feature space can make it simpler.

3.1 Vector Matrix Representation with Dummy Component

Similar to what we did earlier for the input space we can also vectorise the features space. Looking at the above linear model expressions, we can easily identify that they can be changed into:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1\phi_1(\mathbf{x}) + w_2\phi_2(\mathbf{x}) + \cdots + w_M\phi_M(\mathbf{x})$$

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \mathbf{w}^\top\boldsymbol{\phi} \tag{6.}$$

However, it is also more useful if we express the whole model using vectors. To do so, we can define a **dummy** feature $\phi_0 = 1$ for all vectors of the input space. In this case we can define our linear model as:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top\boldsymbol{\phi} \tag{7.}$$

Where $\mathbf{w} = (w_0, w_1, w_2, \dots, w_M)$ and $\boldsymbol{\phi} = (\phi_0, \phi_1, \phi_2, \dots, \phi_M)^{\mathrm{T}}$ and $\phi_0 = 1$.

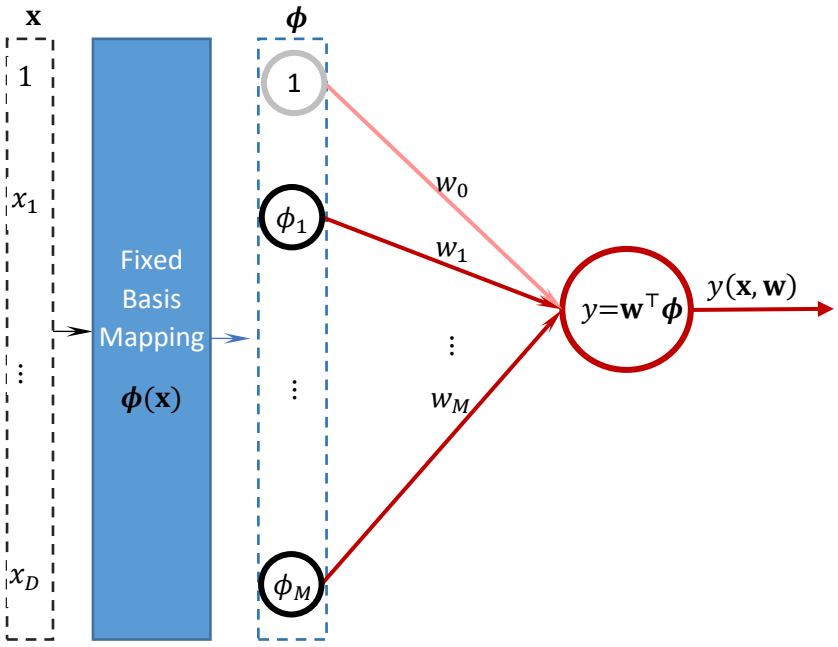

Figure (2.1): schematic representation of a linear regression model with basis.

By fixed basis we mean that the basis function does not change. However, this does not mean that it is constant, it means the basis function itself does not change from one function to another-in terms of type and parameters.. So we can use a fixed basis to map the input space into a new features space and then use the features to learn a linear model. The resultant model is still linear in the feature space.

3.2 BASIS FUNCTIONS

3.2.1 Polynomial basis functions

As we said earlier, we can use any basis function in our linear model as long as the weights are all linear. If the basis function is polynomial our linear model can take lots of different forms. For example: $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1^2 + w_2 x_2^2 + \cdots + w_D x_D^2$ is a linear model with quadratic basis. While $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1^3 + w_2 x_1^5$ is linear model with polynomial of power 5 of x_1. Our first simple linear models $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D$ become a special case of this form with polynomial of degree 1 since it can be written as $y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1^1 + w_2 x_2^1 + \cdots + w_D x_D^1 = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D$.

The main advantage of using such basis is its ability to represent the more complex shaped relationship between the input and the output. Sometimes this is exactly what we want, see Figure 3.2.

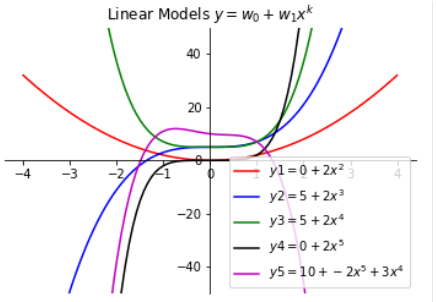

Figure (2.2): shows a set of different polynomial basis for 4 linear models with single input variable x.

Although these are more capable of capturing more nonlinear function shape (remember our model is still linear in the weights) nevertheless such basis have limitations. The most important limitation of polynomial basis is that they are global functions. Meaning that changes in one part infiltrate to affect other parts. Note that even polynomial has similar shapes that span the first and second quarters of the real plane (y is positive) and odd polynomial have similar shapes that span the first and third quarter of the real plane (y is can be positive and negative).

3.2.2 Gaussian basis functions aka radial basis functions

Another important basis that we will utilise is the Gaussian basis functions also known as radial basis functions (RBF). As you know, Gaussian distribution is also known as the normal distribution. This basis is inspired by the Gaussian but it is not exactly the same, more on that in a moment. This basis takes the following form:

$$\phi_j(x) = e^{-\frac{1}{2\sigma^2}(x-\mu_j)^2} \tag{8.}$$

where μ_j specifies the centre of the basis and σ specifies the spread of the basis. Note that this is not the full Gaussian distribution function, the normalisation factor $\frac{1}{\sqrt{2\pi\sigma^2}}$ is missing, so it does not necessarily have a probabilistic meaning because the basis is going to be scaled by a weight anyway inside the model. As a reminder, a univariate Gaussian probability density denoted as $\mathcal{N}(x|\mu,\sigma^2)$ with mean μ and variance σ^2 is given as:

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{9.}$$

Note however, that the max value that the RBF basis can take is 1, contrary to the Gaussian distribution which has its sum over all x's is 1. See Figure 3.3.

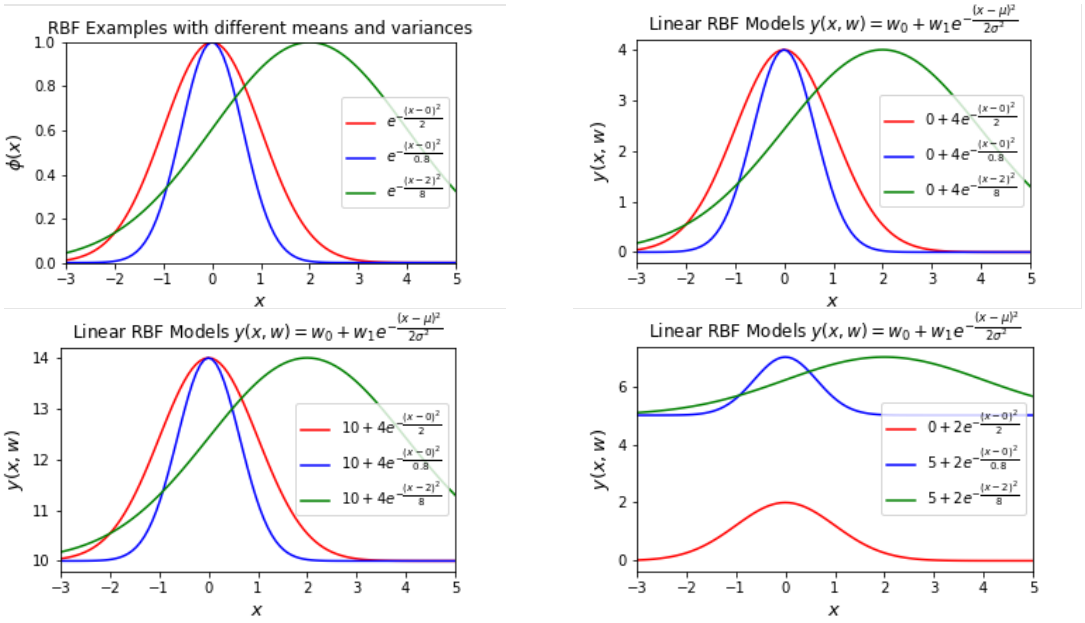

Figure (2.3): (top-left) shows three examples of RBF basis with means of 0,0,2 and variances of 1, 0.4, 4 respectively. (top-right) same RBF when used in linear models with same weights of 0 and 4 respectively. (bottom-left) same RBF when used in linear models with same weights of 10 and 4 respectively. (bottom-right) same RBF when used in linear models with different weights of (0,2), (5,2) and (5,2) respectively.

We can see from figure 3.3 that the effect of weights is as follows: the bias w_0 shifts the entire model up or down while the w_1 scale the entire model to a range of [0,w_1]. The effect of the mean and the variance is as usual; the mean specifies where the model peaks and the variance specifies how narrow or wide the model.

Multidimensional input space we use multivariate radial basis functions takes the form:

$$\phi_j(\mathbf{x}_n) = e^{-\frac{1}{2}(\mathbf{x}_n-\boldsymbol{\mu_j})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}_n-\boldsymbol{\mu}_j)} \quad (10.)$$

where $\boldsymbol{\mu_j}$ has the dimension D of the input space however, note that the number of those basis $M-1$ (it is M if including the dummy features) specifies the size of the feature space which would be the input for the linear model. $\boldsymbol{\Sigma^{-1}}$ is the inverse of the covariance matrix. The covariance matrix $\boldsymbol{\Sigma}$ is an $(M-1)\times(M-1)$ squared, symmetrical, positive and semi-definite matrix.

The above define a set of multinomial Gaussians (without a normalisation factor), each is defined by a different means $\boldsymbol{\mu}_j$ vectors and all share the same covariance matrix $\boldsymbol{\Sigma}$. The means $\boldsymbol{\mu}_j$ need to be specified sensibly, however this is not trivial. Sometimes, we can do that by exploiting some domain knowledge and a crude analysis of the data (see the next notebook). We will see more proper and better ways of specifying those centres in the Machine Learning(ML) module. One way is to cluster the data and choose the centroids of the clusters. As we saw earlier in unit 3, clustering provides a good insight into the nature of the dataset and can be utilised here to decide on the radial basis centres. This might still need tuning and sometimes it is not straight forward to know which distance metric to use for the clustering process. Another way is via Gaussian processes. The topic of how we fit a Gaussian or in a parametric or nonparametric model is an important and significant one in machine learning, and will be left for the abovementioned ML module. Note also that we assumed that the covariance matrix is the same for all the basis but this need not be the case. We can allow each feature to take on a different Gaussian basis with its own different covariance matrix.

> Note that on the diagonal of $\boldsymbol{\Sigma}$ in the usual Gaussian distribution we have $i=j$, $cov(x_i^2) = E[(x_i-\mu_i)^2] = \sigma_i^2$, where σ_i is the variance of x_i. Note that $|cov(x_i x_j)| \leq |\sigma_i \sigma_j|$ in fact $cov(x_i x_j) = \frac{1}{2}var(x_i+x_j) - \sigma_i + \sigma_j$. Note also that although the covariance can be calculated in the following two equivalent ways, we prefer the first because the second is computationally susceptible to an issue called catastrophic cancellation:
>
> $$cov(x_i x_j) = E\big[(x_i-\mu_i)(x_j-\mu_j)\big]$$
>
> $$cov(x_i x_j) = E(x_i x_j) - \mu_i \mu_j$$
>
> Note that the normal distribution maps a vector $\mathbf{x}$ to a real number since $\underbrace{\underbrace{(\mathbf{x}-\boldsymbol{\mu})^\top}_{vector}\underbrace{\boldsymbol{\Sigma}(\mathbf{x}-\boldsymbol{\mu})}_{vector}}_{scalar}$ i.e. $\mathcal{N}: \mathbb{R}^M \mapsto \mathbb{R}$.

3.2.3 Sigmoidal basis functions

Another exponential basis is the sigmoidal basis. This type of basis functions takes the form:

$$g(\alpha) = \frac{1}{1+e^{-\alpha}} \quad (11.)$$

If we take the derivative of the sigmoid we get (you can have a look at the Sigmoid Derivative box for details)

$$\frac{dg}{d\alpha} = g(1-g) \quad (12.)$$

Often we define: $\alpha_j = \frac{1}{\sigma}(x-\mu_j)$ where μ_j specifies the centre of the basis and σ specifies the spread of the basis:

$$\phi_j(x) = g(\alpha_j) = \frac{1}{1+e^{-\frac{1}{\sigma}(x-\mu_j)}} \quad (13.)$$

Note that the term $\alpha_j = \frac{(x-\mu_j)}{\sigma}$ appears without squaring (and the $\frac{1}{2}$) in contrast to the RBF basis which takes the form $e^{-\frac{1}{2\sigma^2}(x-\mu_j)^2}$.

Below we show some examples of the behaviour of the sigmoid for 1d input space.

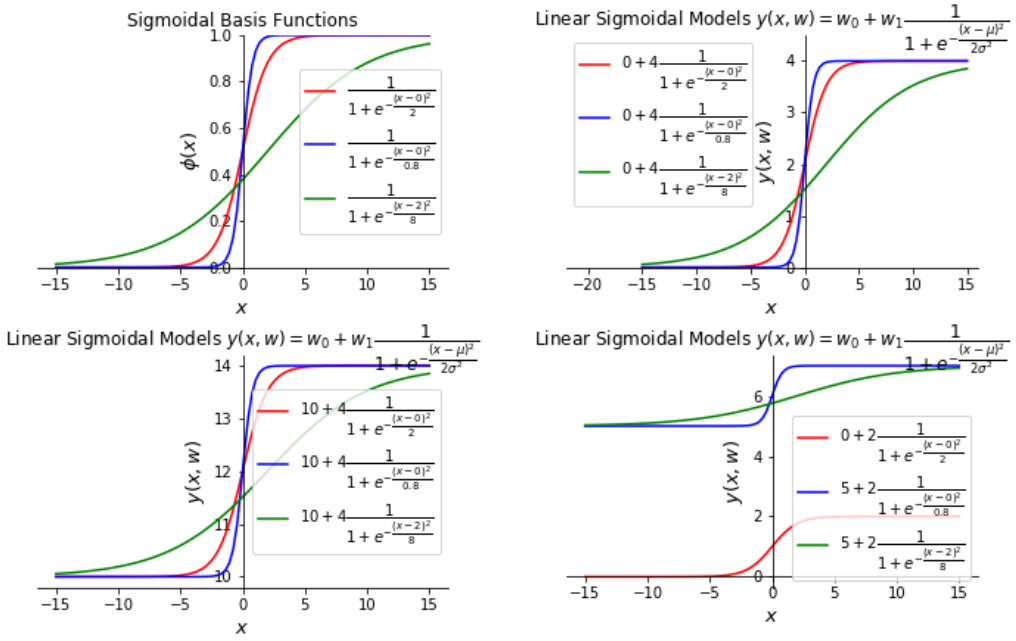

Figure (2.4): (top-left) shows three examples of sigmoidal basis with means of 0,0,2 and variances of 1, 0.4, 4 respectively. (top-right) same sigmoidal basis when used in linear models with same weights of 0 and 4 respectively. (bottom-left) same sigmoidal basis when used in linear models with same weights of 10 and 4 respectively. (bottom-right) same sigmoidal basis when used in linear models with different weights of (0,2), (5,2) and (5,2) respectively.

Sigmoid Derivative

The sigmoid can be written as $g(\alpha)=\frac{1}{1+e^{-\alpha}}=\frac{e^{\alpha}}{e^{\alpha}+1}$

If we take the derivative of the sigmoid we get that:

$$\frac{dg}{d\alpha}=\frac{e^{\alpha}(1+e^{\alpha})-e^{\alpha}e^{\alpha}}{(1+e^{\alpha})^2}=\frac{e^{\alpha}(1+e^{\alpha}-e^{\alpha})}{(1+e^{\alpha})^2}=\frac{e^{\alpha}}{(1+e^{\alpha})^2}$$

However:

$$g(\alpha)\big(1-g(\alpha)\big)=\frac{e^{\alpha}}{1+e^{\alpha}}\cdot\frac{1}{1+e^{\alpha}}=\frac{e^{\alpha}}{(1+e^{\alpha})^2}$$

And so we have:

$$\frac{dg}{d\alpha}=g(\alpha)\big(1-g(\alpha)\big)$$

For a multidimensional input space where we have $\mathbf{x}_n$ as a vector of dimension D, we can utilise the square root of the exponent of a multivariate Gaussian to calculate α_j as follows (known as Mahalanobis distance):

$$\alpha_j=\Big((\mathbf{x}_n-\boldsymbol{\mu}_j)^{\top}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n-\boldsymbol{\mu}_j)\Big)^{\frac{1}{2}} \tag{14.}$$

$$\phi_j(\mathbf{x}_n)=g(\alpha_j)=\frac{1}{1+e^{-\alpha_j}} \tag{15.}$$

to generate a set of different basis $j=1,\dots,M$, where $\boldsymbol{\mu}_j$ are vectors of M means each of size D. All the basis may share the same covariance $\boldsymbol{\Sigma}$ or have a different covariance matrices $\boldsymbol{\Sigma}_j$. We showed the former above because it is more common to have one covariance although this normally requires that the input data is normalised first.

3.2.4 The *tanh* Basis Functions

A basis that is closely related to the sigmoid (both belongs to the family of exponential distributions) is the tanh. The tanh basis function takes the form:

$$h(\alpha) = \frac{e^{2\alpha}-1}{e^{2\alpha}+1} \tag{16.}$$

The derivative of the tanh is given as:

$$\frac{dh}{d\alpha} = 1 - h^2 \tag{17.}$$

It is important to note that both the tanh and the sigmoid have the following relationship:

$$\mathrm{h}(\alpha) = 2\, g(2\alpha) - 1 \tag{18.}$$

This relationship suggests that using either in a linear model is equivalent. However, we need to be mindful that the derivatives of these functions behave differently and so when they are used in other contexts (as activation functions that acts on the weights, ex. in a non-linear model) they result in different behaviours and the non-linear models that use them also differ.

Both the tanh and sigmoid are members of the family of sigmoidal functions.

3.3 Exercise

Please run the following Jupyter notebook and experiment with different weights values to develop an intuition of the effect of weights changes of a linear basis model for a single variable.

3.4 Batch Learning: The Least Squares for Linear Regression Models with Fixed Basis

In this section we will derive how to train a linear regression model that uses some basis. In general, we will use the same formulation that we used earlier for training a regression model without basis. So, we have the same loss function as before but the model is expressed in terms of basis instead of the input features. We would need to adjust our estimations $y(\mathbf{x})$ by adjusting the parameters $\mathbf{w}$ to give us the desired answers t. Our model is written as:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) \tag{19.}$$

where $\mathbf{w} = (w_0, w_1, w_2, \dots, w_M)$ and $\boldsymbol{\phi} = (\phi_0, \phi_1, \phi_2, \dots, \phi_M)^\top$ and $\phi_0 = 1$.

For a data point $\mathbf{x}_n$ we have

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_n) \tag{20.}$$

For brevity we will refer to $\boldsymbol{\phi}(\mathbf{x}_n)$ as $\boldsymbol{\phi}_n$ so the above is expressed as:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}_n \tag{21.}$$

And the loss function can still be written as

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}\big(t_n - y(\mathbf{x}_n, \mathbf{w})\big)^2$$

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(t_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 \tag{22.}$$

We define the **design matrix** $\mathbf{\Phi}$, which has a dimension of $N \times M$ and whose n[th] row is $\boldsymbol{\phi}_n^\top$ and $\mathbf{t}$ is the target matrix, as follows:

$$\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}_1^\top \\ \vdots \\ \boldsymbol{\phi}_n^\top \\ \vdots \\ \boldsymbol{\phi}_N^\top \end{bmatrix} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\top \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^\top \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_N)^\top \end{bmatrix} = \begin{bmatrix} 1 & \phi_1(\mathbf{x}_1) & \cdots & \phi_M(\mathbf{x}_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(\mathbf{x}_n) & \cdots & \phi_M(\mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(\mathbf{x}_N) & \cdots & \phi_M(\mathbf{x}_N) \end{bmatrix} \qquad \mathbf{t} = \begin{bmatrix} t_1 \\ \vdots \\ t_n \\ \vdots \\ t_N \end{bmatrix}$$ and of course $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \\ \vdots \\ w_D \end{bmatrix}$

Similar to what we did in the previous section for the simple linear models we can arrive to the normal equations for the feature space by replacing $\mathbf{X}$ with $\mathbf{\Phi}$. All predictions of the model can be expressed as:

$$\mathbf{y}(\mathbf{X}, \mathbf{t}) = \mathbf{\Phi}\mathbf{w}$$

The loss function can be written as a vector norm $\|.\|^2$ as follows:

$$\overline{J^2} = \frac{1}{2N} \|\mathbf{t} - \mathbf{\Phi}\mathbf{w}\|^2$$

By taking the gradient and setting it to 0 we get:

$$\nabla \overline{J^2} = \frac{2}{2N} \mathbf{\Phi}^\top (\mathbf{t} - \mathbf{\Phi}\mathbf{w}) = 0$$

$$\mathbf{\Phi}^\top \mathbf{\Phi} \mathbf{w}^* = \mathbf{\Phi}^\top \mathbf{t} \tag{23.}$$

we multiply both sides by the inverse of $\mathbf{\Phi}^\top \mathbf{\Phi}$ to get

$$\mathbf{w}^* = (\mathbf{\Phi}^\top \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{t} \tag{24.}$$

The above gives us a closed form solution for $\mathbf{w}^*$ and is known as the normal equations. Closed form solutions are not always available for a machine learning or data mining tasks. Their existence facilitates more analysis and insights into the problem. Some problems might not have a closed form formula of their solution however we can still estimate the solutions numerically. Sometimes also closed form solution can be impractical for big datasets due to their computational demanding nature.

The algorithm that returns the optimal solution for a linear model is given as follows:

Algorithm 1': Least Squares for Linear Regression Model with Gaussian Basis

Input:

Input set: design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \ldots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is of size D

Labels set: vector $\mathbf{t} = [t_1, \ldots, t_N]^\top$ each t_n is a scalar

$\boldsymbol{\mu}_j$: M Basis centres, each is a vector of size D

$\mathbf{\Sigma}$: Covariance matrix of size D×D

Output: $\mathbf{w}^*$ optimum weights; a vector of size $M + 1$

LS_LRegressBasis($\mathbf{X}, \mathbf{t}, \boldsymbol{\mu}, \mathbf{\Sigma}$):

Map the data $\mathbf{X}$ into design matrix $\mathbf{\Phi}$ via the Gaussian basis $\phi_j(\mathbf{x}_n) = e^{-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_j)^\top \mathbf{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_j)}$

$\mathbf{\Phi} = [\mathbf{1}_N, \mathbf{\Phi}]$ # add dummy feature to the design matrix

$\mathbf{w}^* = (\mathbf{\Phi}^\top \mathbf{\Phi})^{-1} (\mathbf{\Phi}^\top \mathbf{t})$

Return the solution $\mathbf{w}^*$

Note that the basis is fixed and not changed during learning. This a key difference between linear and non-linear models which has adaptive basis.

3.4.1 Complexity discussion

The same discussion that we had earlier applies again of course for the case of feature space but this time the complexity is in tern of M. The Least Squares on linear regression with basis has the same complexity as in the simple regression without basis. The only difference is that it will be related to M the feature space dimension instead of D the input space dimension.

3.5 Batch Learning, Sequential or Mini-Batch Stochastic Learning:

The least squares solution is called batch solution since they dictate processing the entire dataset at once (see Algorithm1) specifically in terms of the Design matrix $\mathbf{\Phi}$**.** This restricts the applicability of the algorithms and makes it difficult and computational costly to apply them on a big dataset. This is when the stochastic (or sequential) gradient decent comes to the rescue. Essential all the three algorithms that we have considered for linear models on input X also apply for the feature space. Therefore for brevity we will only show the vectorised stochastic gradient descent. We consider one (or more mini-batch) data point at a time and we update the weights accordingly without waiting until all the other updates becomes available. This is particularly useful when the data is fed from a stream or when we are tackling large-scale learning.

Algorithm 6': Mini-Batch Stochastic Gradient Descent Updates for Linear Regression Model: with vectorisation and basis functions.

Input:

Input set: design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is a vector of size D
Labels set: vector $\mathbf{t} = [t_1, \dots, t_N]^\top$ each t_n is a scalar
Training set

b: The mini-batch size (specifies how frequently we want to update the weights $\mathbf{w}$).
η_0: Initial learning rate
$epcs$: Number of epochs
$\boldsymbol{\mu}_j$: M Basis centres, each is a vector of size D
$\mathbf{\Sigma}$: Covariance matrix of size D×D

Output: $\mathbf{w}$ an approximation for optimum weights $\mathbf{w}^*$; a vector of size $M+1$

SGD_LRegressBasis$(\mathbf{X}, \mathbf{t}, b, \eta_0, epcs)$:

Initialise $\mathbf{w}$ and $\eta = \eta_0$
For epoch $= 1{:}\, epcs$
  For iteration $\tau = 1{:}\, q$ # $q \geq N/b$
    Select a mini-batch $\mathbf{X}_\tau, \mathbf{t}_\tau$ of size b from $\mathbf{X}, \mathbf{t}$ # by sampling or by shuffling & partitioning
    Map $\mathbf{X}_\tau$ to $\mathbf{\Phi}_\tau$ # $\phi_j(\mathbf{x}_n) = e^{-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_j)^\top \mathbf{\Sigma}(\mathbf{x}_n - \boldsymbol{\mu}_j)}$ $j = 1, \dots, M$
    $\mathbf{\Phi}_\tau = [\mathbf{1}_b, \mathbf{\Phi}_\tau]$ # add dummy feature to the mini-batch
    $\mathbf{w} = \mathbf{w} + \eta \frac{1}{b} \mathbf{\Phi}_\tau^\top (\mathbf{t}_\tau - \mathbf{\Phi}_\tau \mathbf{w})$
  Decay η # if necessary: ex. η=0.9×η
Return the final solution $\mathbf{w}$

3.6 Exercise

See the following Jupyter notebook for an example of implementing a stochastic gradient descent with Gaussian Basis.

3.7 Overfitting of Regression Models

Similar to classification models, regression models can suffer from overfitting. Linear regression models are relatively less exposed to this phenomenon because it is a simple regression model. The more complex the model is the more exposed to overfitting. So more complex regression model tends to be more exposed to overfitting. There is a famous principle called Occam's razor which roughly speaking states that we should prefer less complex model if they give us the capability that we want for our model and only use more complex models if necessary. Let us see an example of how overfitting and underfitting behaves in the context of regression model.

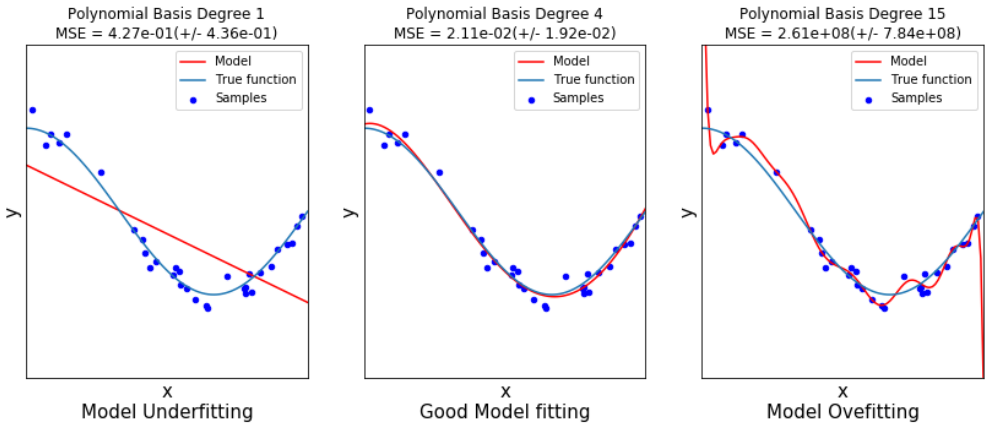

Figure (2.5): The effect behaviour of overfitting and underfitting on a linear model with a polynomial basis. The fitted function that we sampled the data from is a sin wave function.

As we can see the data (blues dots as usual) is non-linear (we cannot represent it with a straight line). We can see in the above figure that when the model is not sufficiently capable (such when we use no basis, which is considered a linear model with degree 1 polynomial basis) we get underfitting because the straight line underfits or is incapable of coming close enough to the actual dataset. When the model is too complex such as when we use a polynomial of degree 15 we get overfitting, because the model is trying to fit every single data point as perfectly as possible, the results is not great and the generalisation of both cases of overfitting and underfitting is poor. On the other hand when the model complexity is just right as in the middle with a polynomial of degree 4 we get an excellent approximation and the model generalisation ability is maximum.

3.8 Least Squares for Regression with Regularisation

Regularising the weights helps suppress the weights from changing or growing too much. This often helps prevent the problem of overfitting. To add regularisation to our linear regression model we start by adjusting the loss function. We simply add a term that discourage the weights from growing. This can be done in few ways one of them is to add a magnitude $\|\mathbf{w}\|$ of the weights inside the loss function to try to minimise it along with the error.

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}\left(\underbrace{\|\mathbf{t} - \mathbf{\Phi w}\|^2}_{term1} \quad + \quad \underbrace{\lambda\|\mathbf{w}\|^2}_{term\ 2}\right) \tag{25.}$$

If these two terms seem to you to work against each other, you are right, it is the case, and this is precisely what we want. We want them to balance each other so that we do not end up fitting the data too much by changing the weights too much or growing the weights too large but at the same time we still want to reduce the differences between the predicted values and the actual values. As usual we square the magnitude of the weights vector to keep the terms nicely differentiable and we will multiply by a coefficient λ to be able to place more or less emphasis on one of the terms. Also for differentiability and because these two terms are squared we multiply by $\frac{1}{2}$

to keep the resultant formula tidy and since it is not going to affect the direction of the changes only the magnitude. Note that $\|\mathbf{w}\|^2 = \mathbf{w}^\top\mathbf{w}$ which is useful for differentiation as well.
By taking the gradient and setting it to 0 we get

$$\nabla\bar{J}^2(\mathbf{w}) = \frac{2}{2N}(\mathbf{\Phi}^\top(\mathbf{t} - \mathbf{\Phi}\mathbf{w}) + \lambda\mathbf{w}) = 0$$

$$\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{w}^* + \lambda\mathbf{w}^* = \mathbf{\Phi}^\top\mathbf{t} \tag{26.}$$

$$\mathbf{w}^* = (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^\top\mathbf{t} \tag{27.}$$

If the matrix $(\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda\mathbf{I})$ is invertible then the solution exists. The resultant algorithm is very similar to Algorithm 3, the only difference is that we change the calculation $\mathbf{w}^*$ so that the invers involve $\lambda\mathbf{I}$, we will not include it here for brevity.

This technique is also called Ridge regression. See figure below for how linear regression with polynomial basis of degree 15 overfitting problem can be brought under control using regularisation. The problem is contrived but because in the first place we should not be using that high degree polynomial in the first place but it demonstrates the effect of regularisation. Note that the value of λ has a major effect of whether the model underfit or makes a good fit, this type of hyper parameters requires tuning which is a trial and error process and a time consuming exercise. There are some ways to automate the process a bit. For example one use grid search method to try out different values for λ and choose the optimal one as we saw previously in Unit2.

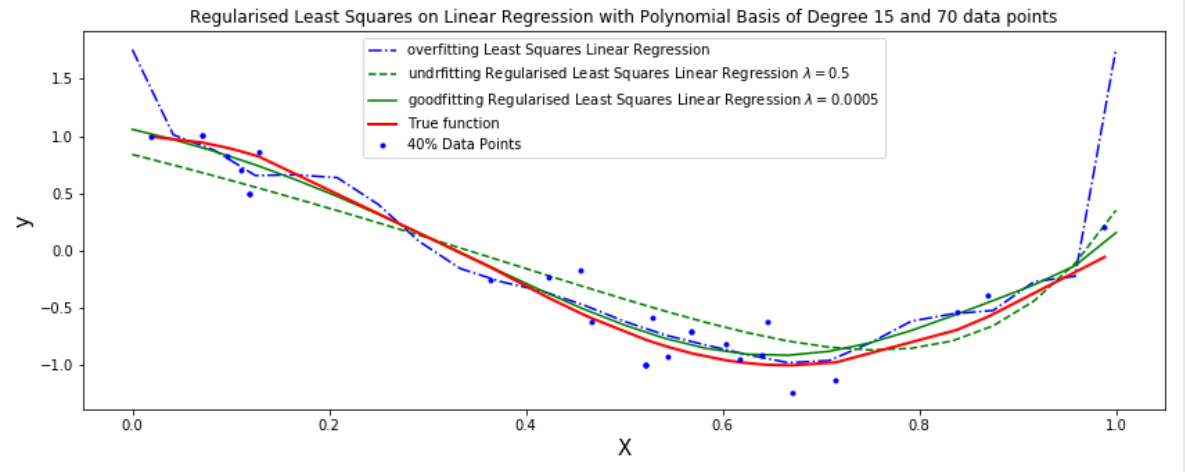

Figure (2.6): The effect of the regularisation constant λ on overfitting and underfitting.

Note in the below figure how both the problems of underfitting of a regularised linear regression models and the overfitting of a linear regression model were greatly reduced when we increased the data from 70 to 500. Both figures show 40% only of the actual data. This illustrate an important aspect of modelling which is that the models are going to be much more resilient with more data and less resilient and more sensitive to overfitting and underfitting and outliers with less data. We will talk about outliers in later units.

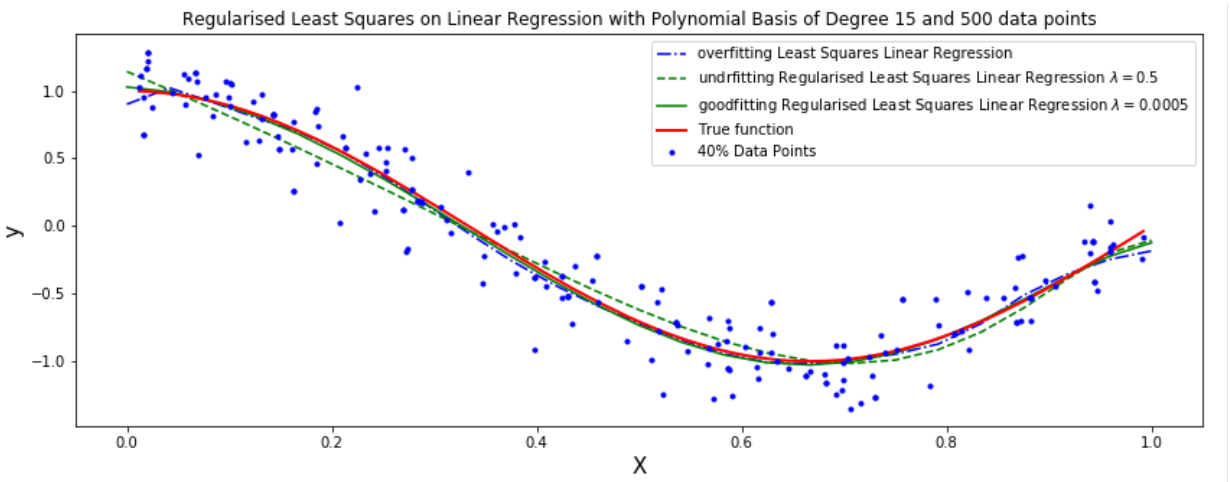

Figure (2.7): overfitting is reduced by increasing the number of data points considered from 70 to 500.

3.9 Exercise

Amend the previous exercise (which implements the Least Squares with Basis) to utilise regularisation, using numpy only. Also you can try to implement a grid search technique to select a best value for λ, make sure to split the data properly and add cross validation if necessary.

3.10 Exercise

Regularisation can be combined easily with other techniques in sklearn, you may have a look at the following notebook for an example of how to pipeline different techniques in sklearn.

3.11 Regularised Mini-Batch Stochastic Gradient Descent Updates for Linear Regression Model

Similar to what we have done before, there is a regularised version of the minim-batch stochastic gradient descent that we show below:

The regularised loss function for one-output regression problem can be written as

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}(\|\mathbf{t} - \mathbf{\Phi}\mathbf{w}\|^2 + \lambda\|\mathbf{w}\|^2)$$

Therefore since we have that $\bar{J}^2(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N} J_n^2$ then $J_n^2 = (t_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 + \lambda\|\mathbf{w}\|^2$

This in turn allows us to take the derivative with respect to one data point:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\frac{1}{2N}\nabla J_n^2$$

$$\nabla J_n^2 = -2\boldsymbol{\phi}_n(t_n - \mathbf{w}^\top \boldsymbol{\phi}_n) + \lambda\mathbf{w}$$

And when we do not know N in advance we can just select a step:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\frac{1}{N}\left[-\boldsymbol{\phi}_n\left(t_n - \mathbf{w}^{(\tau)^\top}\boldsymbol{\phi}_n\right) + \lambda\mathbf{w}^{(\tau)}\right]$$

$$\mathbf{w}^{(\tau+1)} = \left(1 - \frac{1}{N}\eta\lambda\right)\mathbf{w}^{(\tau)} + \eta\frac{1}{N}\boldsymbol{\phi}_n(t_n - \mathbf{w}^{(\tau)^\top}\boldsymbol{\phi}_n) \quad (28.)$$

The resultant algorithm is similar to Algorithm 4' and is not shown for brevity. New results on the regularisation can be found in this recent paper and its associated video and you can read this paper to discuss large scale SGD.

3.12 General Case: Linear Models with Multiple Outputs and Fixed Basis

In this section we extend the ideas of a one output linear regression model that we have dealt with so far into a multi-output linear regression model. When we have multiple output for each input, i.e. each output is a vector of K values: $\boldsymbol{t}_n = [t_{n,1}, t_{n,2}, \dots, t_{n,K}]$. The linear model with multiple output can be expressed as

$$\boldsymbol{y}(\mathbf{x}, \mathbf{W}) = \mathbf{W}^\top \boldsymbol{\phi}(\mathbf{x}) \tag{29.}$$

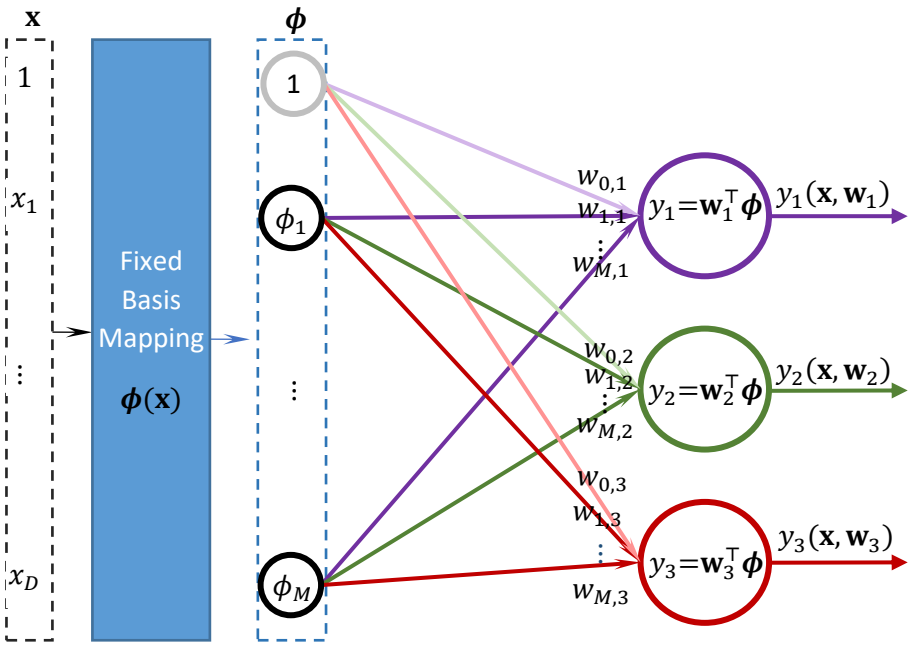

Figure (2.8): Schematic representation of a **linear regression model** with multiple outputs and fixed basis functions.

By fixed basis we mean that the set of basis functions do not change. So if we use a Gaussian basis for example the mean and the variance are fixed, similarly if we use any other basis their parameters do not change. This means that we can use a *separate* model that first learns a basis representations (in a separate pre-processing stage) that suits our problem and then we stop the basis learning to make the basis model fixed and then use this fixed basis model to map the input space into our features space and then use the features to learn a multi-output linear model. The resultant model is still linear in the feature space.

Note that we now use a bold face letter $\boldsymbol{t}_n$ to express the fact that we have a vector of multi-output target values. Similarly, we use a bold face $\boldsymbol{y}$ to denote that the model outputs a vector of multi-output values. In addition, we have used a bold capital $\mathbf{W}$ to signify that we are dealing with a $(M+1) \times K$ matrix (including the biases) instead of a vector of weights. We need in this case a matrix of weights (instead of a vector of weights) since each output component y_i will require its own weights vector. We can combine all the weight vectors in a matrix of weights and we use the same matrix multiplication mechanism that we used before. We can easily adjust our least squares algorithm to accommodate for such a scenario. First we write the model prediction in terms of the weights, then in the next few sections we show the least squares. Staring with the output matrix which we denote now as $\mathbf{T}$ is defined as

$$\boldsymbol{\Phi} = \begin{bmatrix} \boldsymbol{\phi}_1^\top \\ \vdots \\ \boldsymbol{\phi}_n^\top \\ \vdots \\ \boldsymbol{\phi}_N^\top \end{bmatrix} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1)^\top \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_n)^\top \\ \vdots \\ \boldsymbol{\phi}(\mathbf{x}_N)^\top \end{bmatrix} = \begin{bmatrix} 1 & \phi_1(\mathbf{x}_1) & \cdots & \phi_M(\mathbf{x}_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(\mathbf{x}_n) & \cdots & \phi_M(\mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(\mathbf{x}_N) & \cdots & \phi_M(\mathbf{x}_N) \end{bmatrix} \mathbf{T} = \begin{bmatrix} \boldsymbol{t}_1^\top \\ \vdots \\ \boldsymbol{t}_n^\top \\ \vdots \\ \boldsymbol{t}_N^\top \end{bmatrix} = \begin{bmatrix} t_{1,1}, t_{1,2} \dots t_{1,K} \\ \vdots \\ t_{n,1}, t_{n,2} \dots t_{n,K} \\ \vdots \\ t_{N,1} t_{N,2} \dots t_{N,K} \end{bmatrix}$$

$$\mathbf{W}^\top = \begin{bmatrix} \mathbf{w}_1^\top \\ \vdots \\ \mathbf{w}_k^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} = \begin{bmatrix} w_{0,1}, w_{1,1} \dots w_{M,1} \\ \vdots \\ w_{0,k}, w_{1,k} \dots w_{M,k} \\ \vdots \\ w_{0,K}, w_{1,K} \dots w_{M,K} \end{bmatrix}$$

The model now can be expressed in a full vectorised form as:

$$\mathbf{Y}(\mathbf{X}, \mathbf{W}) = \boldsymbol{\Phi}\mathbf{W} \tag{30.}$$

3.13 Batch Learning: The Least Squares for Multi-Outputs Linear Regression Models with Basis

The loss function for multiple output linear models is defined as

$$\bar{J}^2(\mathbf{W}) = \frac{1}{2N}\sum_{n=1}^{N} \|\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n\|^2 \tag{31.}$$

Note here that we are using the norm $\|.\|^2$ since we have a vector of target values. Each operation $\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n$ produces a vector of errors of size K that corresponds to K different outputs. We take the gradient as usual and set it to 0:

$$\nabla \bar{J}^2(\mathbf{W}) = -\frac{2}{2N}\sum_{n=1}^{N} \boldsymbol{\phi}_n(\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n)^\top = 0$$

$$\underbrace{\left(\sum_{n=1}^{N} \boldsymbol{\phi}_n \boldsymbol{\phi}_n^\top\right)}_{\boldsymbol{\Phi}^\top\boldsymbol{\Phi}} \mathbf{W}^* = \underbrace{\sum_{n=1}^{N} \boldsymbol{\phi}_n \boldsymbol{t}_n^\top}_{\boldsymbol{\Phi}^\top\mathbf{T}}$$

where $\boldsymbol{t}_n^\top = [t_{n,1} \quad t_{n,2} \quad \dots \quad t_{n,K}]$. Each $\boldsymbol{\phi}_n \boldsymbol{t}_n^\top$ produce a matrix of size $M \times D$ .We can summarise the operation $\sum_{n=1}^{N} \boldsymbol{\phi}_n \boldsymbol{\phi}_n^\top = \boldsymbol{\Phi}^\top\boldsymbol{\Phi}$ to obtain the normal equation for multiple outputs as follows:

$$\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\mathbf{W}^* = \boldsymbol{\Phi}^\top\mathbf{T}$$

The same result can be obtained by expressing the loss function in a full vector notation without using the sum as we did in an earlier section. This time we need to be careful since we are dealing with multiple outputs that results in matrices $\mathbf{T}$ and $\mathbf{Y}(\mathbf{X}, \mathbf{W})$ instead of vectors $\mathbf{t}$ and $\mathbf{y}(\mathbf{X}, \mathbf{w})$. To do so we will use matrices norms defined as $\|\mathbf{A}\|_F^2 = \sum_{i=1}^{K}\sum_{j=1}^{K} a_{i,j}^2$. It is really nothing but squaring all the elements of a matrix and summing them up all together. This is called *Frobenius norm or Euclidian norm for matrices* (similar to Euclidian vector norm that we have been using so far). For simplicity of presentation we will just denoted as we denote a usual vector norm, we can easily tell the difference due to using capital letters for matrices. Now the loss function can be expressed as

$$\bar{J}^2(\mathbf{W}) = \frac{1}{2N}\|\mathbf{T} - \mathbf{Y}(\mathbf{X}, \mathbf{W})\|^2 \tag{32.}$$

$$\bar{J}^2(\mathbf{W}) = \frac{1}{2N}\|\mathbf{T} - \boldsymbol{\Phi}\mathbf{W}\|^2 \tag{33.}$$

We simply can obtain the gradient as

$$\nabla \bar{J}^2(\mathbf{W}) = \frac{1}{2N}\boldsymbol{\Phi}^\top(\mathbf{T} - \boldsymbol{\Phi}\mathbf{W})$$

We set the gradient as usual to 0 and solve in order to obtain optimal solution $\mathbf{W}^*$:

$$\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\mathbf{W}^* = \boldsymbol{\Phi}^\top\mathbf{T}$$

Which is the same normal equation as above.

To see how the matrices are interacting with each other in terms of the dimensions we can add the dimension of each matrix so see how the intermediate dimensions are cancelled out during multiplication to end up with matrix of size (M,K) on both sides of the equation. $\boldsymbol{\Phi}_{(M,N)}^\top \boldsymbol{\Phi}_{(N,M)} \mathbf{W}_{(M,K)}^* = \boldsymbol{\Phi}_{(M,N)}^\top \mathbf{T}_{(N,K)}$.

Ok, now we multiply by the inverse of $\boldsymbol{\Phi}^\top\boldsymbol{\Phi}$ to get

$$\boldsymbol{\Phi}^\top \boldsymbol{\Phi} \mathbf{W}^* = \boldsymbol{\Phi}^\top \mathbf{T}$$
$$\mathbf{W}^* = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \mathbf{T} \quad (34.)$$

Again the algorithm is similar to the least squares shown in Algorithm 1, but we are dealing with a matrix of weights $\mathbf{W}$ instead of a vector of weights $\mathbf{w}$. we will show you a one later once we develop the concept of regularisation for this general multi-output case.

3.14 Sequential Learning: Multi-Output Stochastic Gradient Descent for Linear Regression Models with Basis

The stochastic gradient descent algorithm for multi-output regression can be written similar to one-output by adjusting the algorithm to deal with weights matrix instead of a weight vector. Essentially we replace the loss function $\bar{J}$ by J_n^2, so after a data point becomes available, we update according to:

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} - \eta \frac{1}{2N} \nabla J_n^2$$
$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} - \eta \frac{1}{N} (-\boldsymbol{\phi}_n) \left(\boldsymbol{t}_n - \mathbf{W}^{(\tau)^\top} \boldsymbol{\phi}_n \right)^\top$$
$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} + \eta \frac{1}{N} \boldsymbol{\phi}_n \left(\boldsymbol{t}_n^\top - \boldsymbol{\phi}_n^\top \mathbf{W}^{(\tau)} \right) \quad (35.)$$

Where τ rpersents the iteration (or the time step) and η is the learning rate parameter which should be carefully chosen so that it does not lead to divergence or oscillation of the algorithms. The resultant algorithm is similar to Algorithm 2 but we are dealing with a matrix of weights $\mathbf{W}$ instead of a vector of weights $\mathbf{w}$.

3.15 Preventing Overfitting the Data and Overshooting the Loss Minimum

In this section we tackle overfitting for stochastic gradient decent algorithms. We provide several mechanism to prevent overfitting. The first goes to the level of the loss function itself via regularisation similar to what we have covered earlier. The second is via a combination of sweeps through the dataset (epochs) as well as weight decay and early stopping. At the same time these techniques are suitable to prevent overshooting the global minimum of the loss function (if there is a one). Remember in SGD we are going in steps towards the minimum of the loss function. On the way, our algorithm might overshoot the minimum and keep fluctuating around it. This is often due to a high learning rate. Overshooting the global minimum and ending up in a local minimum is another problem that we often face with more complex models such as neural networks. In fact overcoming local minima and the fact that a neural network loss function is infested with these local minima are among the main motivation for the next subsection.

3.15.1 Regularised Multi-output Least Squares for Linear Regression Model with Basis

For the least squares we can regularise it by using the vectorised form of the loss function on the whole training set as follows:

$$\bar{J^2}(\mathbf{W}) = \frac{1}{2N} (\|\mathbf{T} - \mathbf{Y}(\mathbf{X}, \mathbf{W})\|^2 + \lambda \|\mathbf{W}\|^2)$$
$$\bar{J^2}(\mathbf{W}) = \frac{1}{2N} (\|\mathbf{T} - \boldsymbol{\Phi}\mathbf{W}\|^2 + \lambda \|\mathbf{W}\|^2) \quad (36.)$$

We simply can obtain the gradient as

$$\nabla \bar{J^2}(\mathbf{W}) = \frac{1}{N} (\boldsymbol{\Phi}^\top (\mathbf{T} - \boldsymbol{\Phi}\mathbf{W}) + \lambda \mathbf{W})$$

We set the gradient as usual to 0 and solve in order to obtain optimal solution $\mathbf{W}^*$:

$$(\boldsymbol{\Phi}^\top \boldsymbol{\Phi} + \lambda \mathbf{I}) \mathbf{W}^* = \boldsymbol{\Phi}^\top \mathbf{T}$$

Which is the regularised normal equation as above for multi-output linear regression.

Algorithm 1'': Least Squares for Multi-output Linear Regression Model with Gaussian Basis

Input:

Dataset as a design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is of size D

Corresponding labels matrix $\mathbf{T} = [\boldsymbol{t}_1^\top, \dots, \boldsymbol{t}_N^\top]^\top$ each $\boldsymbol{t}_n$ **is** of size K

$\boldsymbol{\mu}_j$: M Basis centres, each is a vector of size D

$\boldsymbol{\Sigma}$: Covariance matrix of size D×D

Output:$\mathbf{W}^*$ optimum weights; a matrix of size $(M+1) \times K$

RLS_LRegressBasis($\mathbf{X}, \mathbf{T}, \boldsymbol{\mu}, \boldsymbol{\Sigma}$): # B for basis, K Outputs

Map the data $\mathbf{X}$ into design matrix $\boldsymbol{\Phi}$ via the Gaussian basis $\phi_j(\mathbf{x}_n) = e^{-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_j)}$

$\boldsymbol{\Phi} = [\mathbf{1_N}, \boldsymbol{\Phi}]$ # add dummy feature to the design matrix

$\mathbf{W}^* = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi} + \lambda \mathbf{I})^{-1} (\boldsymbol{\Phi}^\top \mathbf{T})$

Return $\mathbf{W}^*$

3.15.2 Regularised Multi-output Stochastic Gradient Descent for Linear Regression Model with Basis

Similarly the above can be done on a multi-output regression.

$$\bar{J^2}(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^{N} [\|\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n\|^2 + \lambda \|\mathbf{W}\|_F^2]$$

where $\|\mathbf{W}\|^2$ is the Euclidean norm of the weights matrix, which is just the sum of the squares of all of the elements of $\mathbf{W}$. More formally, this is called Frobenius norm and is defined as $\|\mathbf{W}\|_F^2 = \sum_{i=1}^{K} \sum_{j=1}^{K} w_{i,j}^2$.

$$\bar{J^2}(\mathbf{W}) = \frac{1}{2N} \sum_{n=1}^{N} J_n^2 \text{ where } J_n^2 = \|\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n\|^2 + \lambda \|\mathbf{W}\|^2$$

This in turn allows us to take the derivative with respect to one data point:

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} - \eta \frac{1}{2N} \nabla J_n^2$$

$$\nabla J_n^2 = -\boldsymbol{\phi}_n (\boldsymbol{t}_n - \mathbf{W}^\top \boldsymbol{\phi}_n)^\top + \lambda \mathbf{W}$$

when we do not know N in advance we just can suffice by a smaller learning rate η :

$$\mathbf{W}^{(\tau+1)} = \mathbf{W}^{(\tau)} - \eta \frac{1}{N} \left[-\boldsymbol{\phi}_n \left(\boldsymbol{t}_n^\top - \boldsymbol{\phi}_n^\top \mathbf{W}^{(\tau)} \right) + \lambda \mathbf{W}^{(\tau)} \right]$$

$$\mathbf{W}^{(\tau+1)} = \left(1 - \frac{1}{N} \eta \lambda \right) \mathbf{W}^{(\tau)} + \eta \frac{1}{N} \boldsymbol{\phi}_n \left(\boldsymbol{t}_n^\top - \boldsymbol{\phi}_n^\top \mathbf{W}^{(\tau)} \right) \quad (37.)$$

Before we state the regularised mini-batch SGD we would like to add few more techniques to our arsenal against overfitting.

3.15.3 Early Stopping and Learning Rate Decay

One of main tools to prevent overfitting is regularisation as we saw earlier.

Going through several epochs can be combined with learning rate decay stabilise and prevent overshooting the minimum of the loss function. Overshooting the global minimum and ending up in a local minimum is a problem that we often face with more complex models such as neural networks. In fact overcoming local minima and the fact that a neural network loss function is infested with these local minima is the main motivation of going through several epochs and to shuffle the data along the way in order to make sure that we head toward the global minima from several directions. Also SGD with just one point update helps us to overcome some of the local

minima on the way of the global minimum due to its high variance. When we move to a mini-batch SGD setting we hope to keep this ability but to further stabilise the process. This randomisation helps the convergence of the solution to a global minimum. In the case of the simple linear model this is not a problem since we have a convex loss function that has just one global minimum.

Actually, having epochs with learning decay and early stopping are all a little excessive for linear models and are only justified for special cases when the dataset is extremely large, nevertheless for the sake of completeness of coverage and to gain familiarity with these central concepts in modern data science and machine learning, we show them here.

On the other hand, since going through several epochs can lead in some cases to a convergence *earlier* than we might expect we can add also the concept of early stopping by checking if the loss value has increased beyond a specific threshold.

If you remember in unit2 we have spoken about discovering overfitting via comparison of accuracy/error on training set and a validation (or testing) set. Overfitting starts to occur when the performance of the training keep increasing while the performance on a validation set starts to decrease (i.e. they forked). The same concept applies her on regression but we use a different metric such as the MSSE. So to detect overfitting we just have to look at the performance of a hold out set while we are training the model. Once the loss of the model on the validation set starts to increase we can just stop training. This is called early stopping and it can be done in several ways we cover the basic idea only. We need to realise two things here.

The first is that the performance of the hold out set tends to fluctuate a bit so we cannot just stop learning immediately once the generalisation error of the validation set starts to increase. We need to be patient and allow for a leeway for the error to fluctuate a bit. We can do that via a hyper parameter ε where we stop training only when the current validation error exceeds the past validation error beyond ε.

The second thing is that we need a way to track back our best weights before the latest update that led to the increase of the generalisation error. So we need a mechanism to store the best weight and only update them if we are sure that the latest update result in no increase in the generalisation error. We will do this on the level of the epochs not on the level of the mini-batches because individual mini-batch may not necessarily reflect the whole picture of the error. So we will often set an enough high number of epochs and we employ early stopping to stop when we reach the critical point of having maximally trained the model and it reached its peak performance and just before it starts to overfit the data and capture the noise along the patterns in the data.

One thing we would like to point out is that early stopping has been shown to be equivalent to regularisation so both have similar effect of clipping the weights and preventing them from overly changing to accommodate noise as well as the pattern in the data.

The final regularised mini-batch SGD algorithm that is fortified against overfitting is shown below.

Algorithm 6'': Regularised Mini-Batch Stochastic Gradient Descent Updates for Linear Regression Model with Vectorisation

Input:

Input set as a design matrix $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_N^\top]^\top$ each $\mathbf{x}_n$ is of size D
Labels set as a matrix $\mathbf{T} = [\boldsymbol{t}_1^\top, \dots, \boldsymbol{t}_N^\top]^\top$ each $\boldsymbol{t}_n$ is of size K
$\mathbf{X}', \mathbf{T}'$ holdout validation set that have similar structure to the above
$\boldsymbol{\mu}_j$: M Basis centres, each is a vector of size D
$\boldsymbol{\Sigma}$: Covariance matrix of size D×D
η_0: initial learning rate
b: mini-batch size (specifies how frequent we want to update the weights $\mathbf{W}$
λ: regularisation parameter
ep: max number of epochs
ε: early stopping threshold

Output: $\mathbf{W}$ an approximation for optimum weights $\mathbf{W}^*$; a matrix of size $(M+1) \times K$

SGD_RegressBasisK($\mathbf{X}, \mathbf{T}, \mathbf{X}', \mathbf{T}', \eta_0, b, \lambda, ep, \varepsilon$):

Initialise $\mathbf{W}$, $\mathbf{W}' = \mathbf{W}, \eta = \eta_0$ and $\bar{J}_0 = \infty$
For epoch $= 1: ep$ # hyper parameter: max number of epochs
  For iteration $\tau = 1: q$ # $q \geq N/b$
    Select a mini-batch $\mathbf{X}_\tau, \mathbf{T}_\tau$ of size b from $\mathbf{X}, \mathbf{T}$ # randomly or by shuffling & partitioning
    Map $\mathbf{X}_\tau$ to $\boldsymbol{\Phi}_\tau$: $\boldsymbol{\phi}(\mathbf{x}_n) = (1, \boldsymbol{\phi}_1, \dots, \boldsymbol{\phi}_{M-1})$ # $\phi_j(\mathbf{x}_n) = \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_j, \boldsymbol{\Sigma})$ or other basis
    $\mathbf{W}' = \left(1 - \frac{1}{b}\eta\lambda\right)\mathbf{W}' + \frac{1}{b}\eta\boldsymbol{\Phi}_\tau^\top(\mathbf{T}_\tau - \boldsymbol{\Phi}_\tau\mathbf{W}')$ # update with regularisation
  Decay η # if necessary
  $\bar{J}_{ep} = \frac{1}{2N}\|\mathbf{T}' - \boldsymbol{\Phi}'\mathbf{W}'\|^2$ # calculate the loss or other metric on the validation set
  If $\bar{J}_{ep} > \bar{J}_{ep-1} + \varepsilon$: break # simple early stopping or other more sophisticate cond.
  Else $\mathbf{W} = \mathbf{W}'$
Return the final solution $\mathbf{W}$

Note that we did not have an algorithm for the least squares because we cannot do it for multi-layer non-linear neural network. In all of the algorithms for mini-batch we have given them number 6 (6, 6', 6'') the ' signifies the stage of the algorithm, where they cover: linear, linear with basis, linear with basis and multi-outputs, respectively.

See the following paper for an idea why we might want to use multi-output models even if our target is a single value.

Note that arXiv papers are not necessarily peer-reviewed, so we need to be mindful not to take the findings for granted (even with peer-reviewed papers we might occasionally find some flaws). Researchers often publish in the arXiv as a first step because it is a faster means to get feedback on their research, and often they would submit to a reputable journal later. The journal peer-review publications process takes long time, sometimes more than a couple of years. Since AI is moving extra fast, this warrants the use of arXiv but with caution.

3.16 Exercise

See the following notebook to see how we can implement a multi-output linear regression mode using numpy.

3.17 Multi-Layer Multi-Output Linear Regression Models

We will prove that the above model is actually equivalent to a multi-output linear regression model. First, we denote the weight matrix that links the input features with the first set of linear models (called the hidden layer) as $\dot{\mathbf{W}}$ and it is of dimension $3 \times (M+1)$. Next, we denote the weight matrix that links the hidden linear models with the output models (called output layer) as $\ddot{\mathbf{W}}$ and it is of dimension $2 \times (3+1)$.

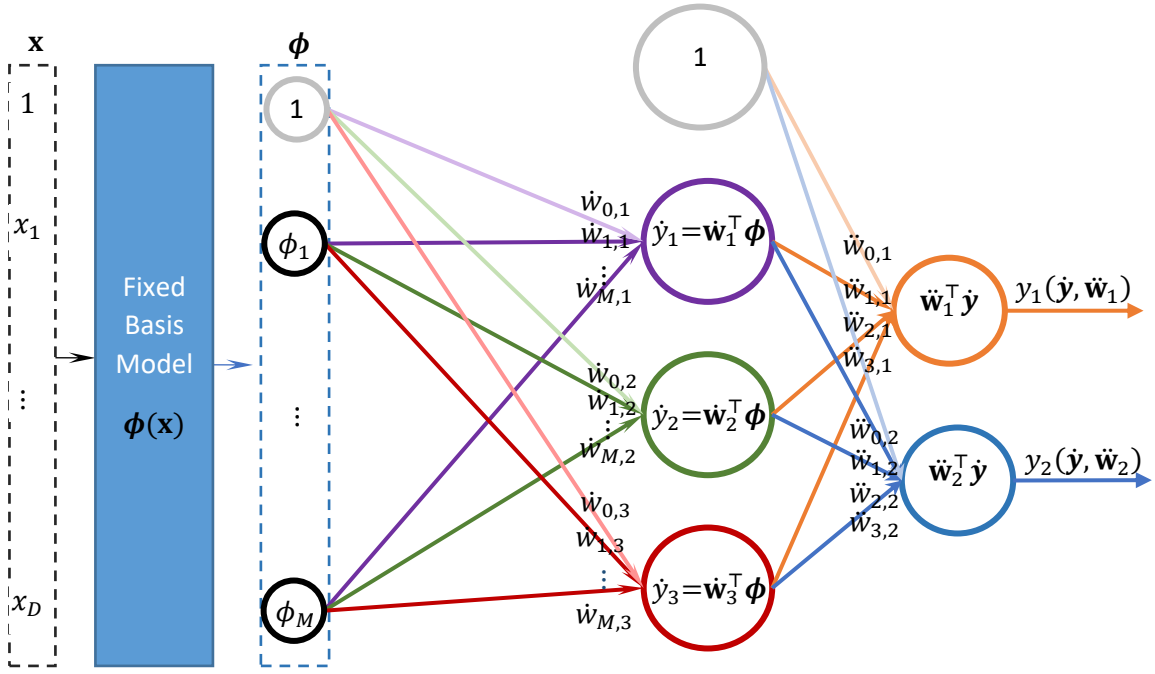

Figure (2.9): Schematic representation of a multiple outputs multi-layers **linear regression model** with fixed basis. We prove that this model is actually equivalent to multiple outputs one-layers linear regression model with fixed basis, shown below.

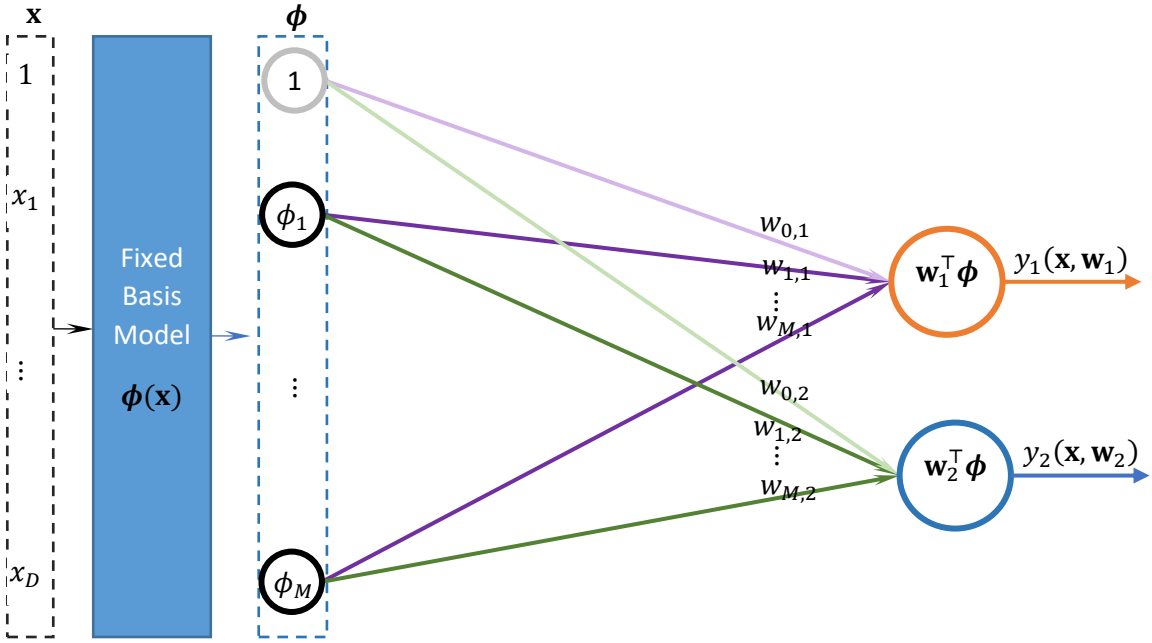

Figure (2.10): Schematic representation of a multiple outputs linear regression model with fixed basis that is equivalent to the multi-outputs multi-layers linear regression model with fixed basis, shown above.

Both layers can be seen as multi-output linear regression model. So let us see how they interact with each other: the hidden layer output is given as

$$\dot{\boldsymbol{y}} = \dot{\mathbf{W}}^{\top}\boldsymbol{\phi}(\mathbf{x})$$

We will denote $\boldsymbol{\phi}(\mathbf{x})$ as $\boldsymbol{\phi}$ and $\boldsymbol{\phi}_n(\mathbf{x})$ as $\boldsymbol{\phi}_n$ hence

$$\dot{\boldsymbol{y}} = \dot{\mathbf{W}}^{\top}\boldsymbol{\phi}$$

Where $\boldsymbol{\phi}(\mathbf{x})$ is a feature vector that includes a dummy feature $\phi_0 = 1$. The output of the output layer is:

$$\boldsymbol{y} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}1\\ \dot{\boldsymbol{y}}\end{bmatrix}$$

It is clear that the weight matrices has incompatible dimension and in particular we need $\dot{\mathbf{W}}^{\top}$ to be of size $(M+1)\times(3+1)$ to be able to multiply it by $\ddot{\mathbf{W}}^{\top}$ which is of size $(3+1)\times 2$ so that the multiplication cancels out the dimension $(3+1)$ to get an overall matrix of size $(M+1)\times 2$.

Let us define a vector of a 1 followed by M 0s as $\mathbf{1_{0_M}} = [1, \underbrace{0, \dots, 0}_{M}]$. So now we append this vector into the weight matrix $\dot{\mathbf{W}}^{\top}$ to a get the matrix $\begin{bmatrix}\mathbf{1_{0_M}}\\ \dot{\mathbf{W}}^{\top}\end{bmatrix}$ and now we can multiply. By substituting $\dot{\boldsymbol{y}}$ in the above we get

$$\boldsymbol{y} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}1\\ \dot{\mathbf{W}}^{\top}\boldsymbol{\phi}\end{bmatrix}$$

We can pull $\boldsymbol{\phi}$ out by padding M 0s to the 1 as follows:

$$\boldsymbol{y} = \begin{bmatrix}y_1\\ \vdots\\ y_K\end{bmatrix} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}1 & 0 & \cdots & 0\\ & & \dot{\mathbf{W}}^{\top} & \end{bmatrix}\begin{bmatrix}1\\ \phi_1\\ \vdots\\ \\ \phi_M\end{bmatrix}$$

The last multiplication in turn can be succinctly written as:

$$\boldsymbol{y} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}\mathbf{1_{0_M}}\\ \dot{\mathbf{W}}^{\top}\end{bmatrix}\boldsymbol{\phi}$$

This setting will allows us not only to match the dimensions, but more importantly to obtain the output of the first layer with a dummy feature of $\dot{y}_0 = 1$ due to the $\mathbf{1_{0_M}}\boldsymbol{\phi} = 1$ multiplication which yields 1.

Now we define $\mathbf{W}^{\top} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}\mathbf{1_{0_M}}\\ \dot{\mathbf{W}}^{\top}\end{bmatrix}$ and substituting we get

$$\boldsymbol{y} = \mathbf{W}^{\top}\boldsymbol{\phi}$$

And so we have written the two-layer model as a one layer linear regression model. This gives the following insight: we do not need multi-layer linear model as it can be expressed as a one layer model. However, we might still want to use two or more layers to reduce the number of parameters that we are dealing with. For example if we have an input space of 100 attributes and 10 outputs then we would need to deal with 1000 parameters, while if we introduced a 5 neurons hidden layer in the middle then we would need to deal with $100\times 50 + 5\times 10 = 550$ parameters. So in this case by using two layers instead of one we reduced the number of parameters of the model which is desirable.

On the other hand, multi-layer model makes more sense when we use a *non-liner activation function* in the hidden layer as follows.

$$\dot{\boldsymbol{y}} = f(\dot{\mathbf{W}}^{\top}\boldsymbol{\phi}) \qquad \boldsymbol{y} = \ddot{\mathbf{W}}^{\top}\begin{bmatrix}1\\ \dot{\boldsymbol{y}}\end{bmatrix}$$

In this case it makes sense to use multi-layer model as the model becomes a non-linear regression model, or a neural network which has more expressiveness power to represent more complex and non-linear relationship than a linear model. In particular, please do not take this section from an impression that multiple hidden layers are always redundant, it is not always the case at all; it may be the case when those hidden layers are *linear hidden layers* that do not serve a purpose of increasing the efficiency of the training.

Specifically when we are trying to infuse multiple layer of abstraction for an application, adding *non-linear hidden layers* forms an excellent tool for us to do so. In fact, deep neural network uses many non-linear hidden layers and they are very successful in a wide range of applications that is ever increasing. What we are trying to do here is to develop your intuition into when adding layers make sense. This explains why we only have few fully connected linear output layers in deep learning architectures, but we have plenty of non-linear hidden layers. We normally have one (or two) fully connected output layers, but normally not more (the reason we have two is to reduce processing at the last couple of layers if we ended up with high number of features produced by the hidden layers).

3.18 SUMMARY

In this lesson we have covered different bases for the generalised linear models with regularisation. We have seen how a multi output regression model maps effectively into a model weights matrix, and we have seen how to move from a batch learning approach to a sequential learning approach. We have also covered ways to overcome the overfitting problem by utilising early stopping and learning rate decay. We have covered the basics of regularised multi output linear and non-linear regression models.

4 NON-LINEAR REGRESSION VIA NEURAL NETWORKS

Learning outcomes:

After completing this lesson you should be able to:

- build the architecture of a two layer or multilayer neural network
- address the regression problem in full using feature extraction process that is built into the neural network architecture, negating the need from the designer to select fixed basis functions
- understand the role of activation function in the context of a neural network
- devise a suitable loss function for a neural network
- explain the process of building a suitable learning algorithm for a multi-layer neural network that depends on the gradient descent and the error propagated through the network's layers

In the previous lesson we have covered a linear regression model with multiple outputs. In this lesson we extend this idea into models that have multiple layers with different activation function. The layers should be defined in a way that do not reduce them into one layers in order to justify the added complexity of the new layers. This is not a strict guideline but it undesirable to have redundant layers that can be otherwise replaced by fewer layers. The reducibility of the layers is tightly connected to the form of the activation function.

An activation function is a function that we pass the output though in order transforms the output into a form that more useful for our model. The activation function can be linear or non-linear. In fact so far we can say that we have been implicitly using an identity activation function (i.e.) the output stays as is. The non-linearity of the activation function is a powerful tool that can transform an input into an output that has gone through considerable processing. The result of such a model with multiple layers and non-linear and linear activation function for each layer is called a neural network. Let us have a look at the following architecture

4.1 NON-LINEAR ONE-OUTPUT REGRESSION USING MULTI-LAYER MODELS

We start with the latest architecture that we have developed in the previous sections and amend it to suit our needs. Please bear in mind that we are building the simplest feedforward neural network here, but there are much more complex networks architecture that you will see later in the machine learning and deep learning modules. We will adopt an approach where we will now develop an architecture that will allow us to *adapt* the basis function that we have assumed previously as being fixed. The number of the features are usually fixed in neural networks because it corresponds with number of neurons in a layers. Other techniques such as support vector machine allow for the flexibility of the adapting the number of feature basis according to the dataset but on the expense of less efficiency during the prediction. In neural networks the adaptation takes place in changing the expressing powers of the basis function by changing the weights of the hidden layer. Below we show how we move from a fixed basis model architecture into flexible adapted basis model architecture.

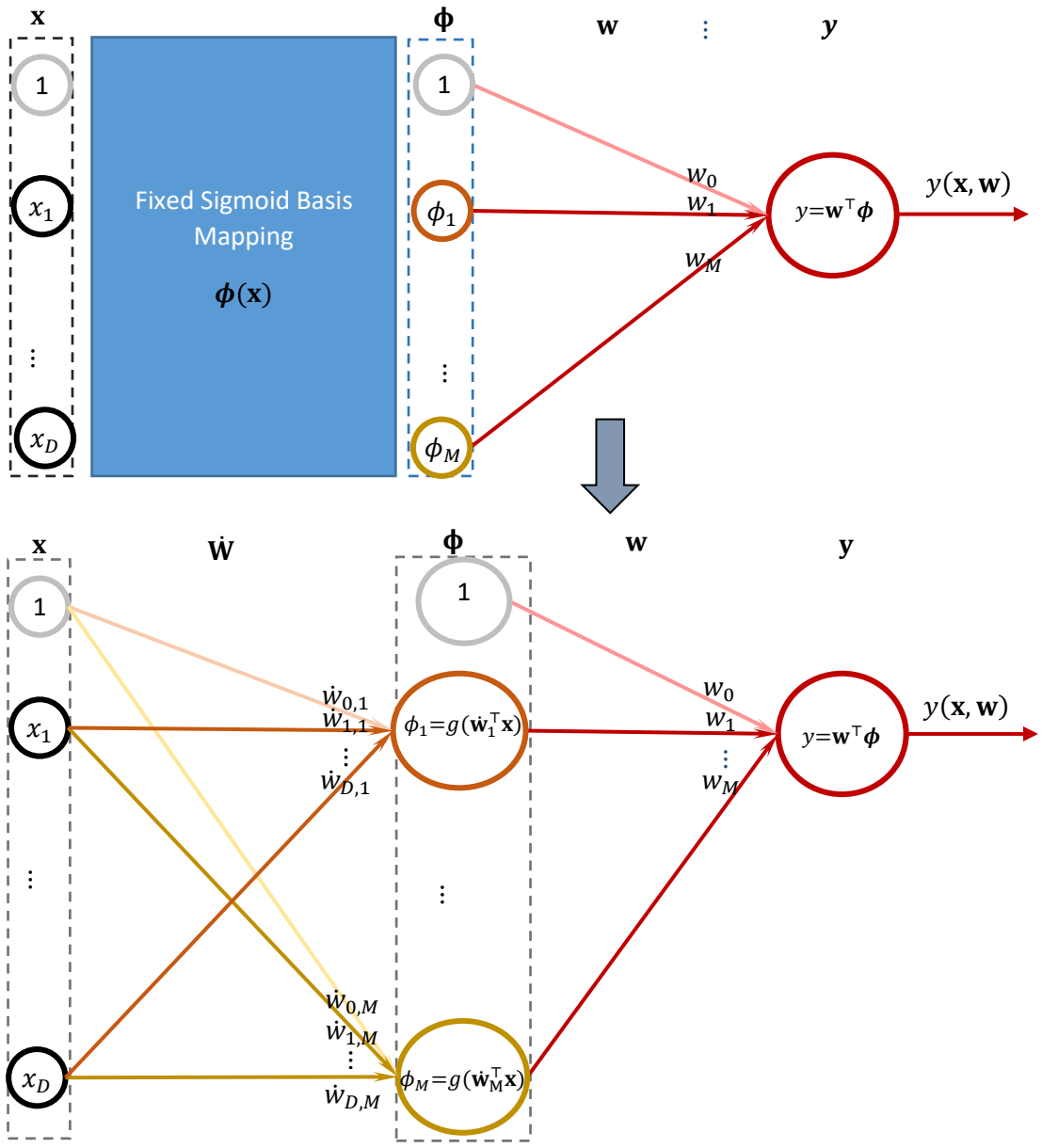

Figure (3.1): (top) Schematic representation of a multiple outputs multi-layers **linear regression model** with fixed basis. (bottom) **non-linear** Multi-layer **Neural Network** model with *adaptive basis*.

First, we denote the weight matrix that links the input features with the first set of linear models (called the hidden layer) as $\dot{\mathbf{W}}$ and it is of dimension $M \times (D+1)$ where D is the input space dimension and M is the number of features that we would like to obtain from the hidden layer. Next, we denote the weight vector that links the hidden linear models with the output models (called output layer) as $\mathbf{w}$ and it is of dimension $(M+1)$. Note that this architecture cannot be reduced into one layer output as we did earlier due to the presence of the activation function. Which support what we have mentioned earlier that this the simplest ANN for regression. So what are the advantages of such an architecture over the one layer architecture you might ask? The answer is that it allows us to capture more complex relationship between the input and the output automatically without the need to come up with a suitable basis functions.

Both layers can be seen as linear regression model that have been cascaded together. The question remains to study how they interact with each other. The hidden layer output is given as

$$\boldsymbol{y} = \mathbf{w}^{\top}\boldsymbol{\phi}$$

Where $\boldsymbol{\phi}(\mathbf{x})$ is denoted as $\boldsymbol{\phi}$ and $\boldsymbol{\phi}$ is a feature vector that includes a dummy feature $\phi_0 = 1$. The output of the input layer is given as:

$$\boldsymbol{\phi} = f(\dot{\mathbf{W}}^{\top}\mathbf{x})$$

where $\mathbf{x}$ is a vector that dummy attribute $x_0 = 1$ and f is an activation function that can be non-linear such as the sigmoid and the tanh or a linear one such as a step function. An important aspect of the activation function is that it should be differentiable in order for the optimisation to be possible and for learning to take place.

So if we decided to choose a sigmoid activation function then the full network can be expressed as

$$\boldsymbol{y} = \mathbf{w}^{\top}\boldsymbol{\phi} \qquad \boldsymbol{\phi} = g(\dot{\mathbf{W}}^{\top}\mathbf{x})$$

This is called a forward propagation which will give us a multi-output prediction for an input $\mathbf{x}$ (both $\mathbf{x}$ and $\boldsymbol{\phi}$ has a dummy component).

Note that if we put the non-linear activation function on the output units and the linear on the hidden layer then we end up reducing the whole network into a one layer network with non-linear activation function so we effectively lose the hidden layer. More formally, let us ignore the biases for a moment, we would have

$\boldsymbol{y} = \mathrm{g}(\mathbf{w}^{\top}\dot{\mathbf{W}}^{\top}\mathbf{x})$, now if we define $\ddot{\mathbf{W}}^{\top} = \mathbf{w}^{\top}\dot{\mathbf{W}}^{\top}$ then we can reduce the model into $\boldsymbol{y} = \mathrm{g}(\ddot{\mathbf{W}}^{\top}\mathbf{x})$ which is equivalent to a one layer non-linear model.

4.2 THE ACTIVATION FUNCTION

Note that for the activation function, we have $\mathbf{w}^{\top}\mathbf{x}$ on the horizontal axis and y on the vertical axis, so please do not mix between x_2 and y they are two different things; x_2 is an attribute and it participates in forming the depicted decision boundaries, while y is a label. More explicitly, in regression the straight line equations in 2D represented the relationship between a *one* attribute x and the label y. On the other hand, the straight line here represents the relationship between attributes x_1 and x_2 and is used to separate the classes using a step activation function.

If we want to confine the values to a]0,1[interval while allowing the activation function to take values in between to reflect the strength of the belief, or the probability, that a data point $\mathbf{x}_n$ belongs (or not) to the positive class then we can use the logistic function shown below.

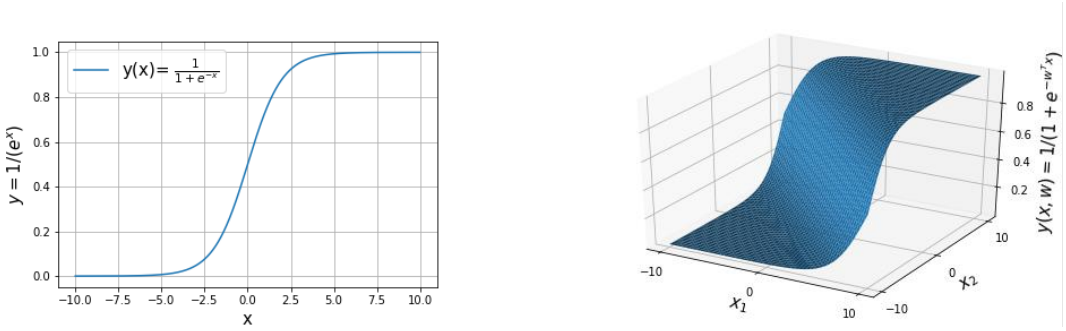

Figure (3.2): Logistic function: (left) logistic function in 2d space with one attribute x. where we can see that the logistic has an inflection point at x=0 where its curvature changes from concave-upward to concave-downward, at this point the logistic value is y=0.5. (right): logistic surface in 3d space with two attributes $x_1$and x_2, where w=[0.6,0.6]. The surface has an inflection surface at x=0.

This activation function is used to be the most common activation function for hidden layers in neural networks for treating non-linear models. It is still an important one that create synergy with a different loss function called cross entropy for classification as we shall see later in the next unit. We will use it when we move from one layer model (including multi-output one) to multi-layer models we need to adjust our cost function.

In the following two sections, we show how to create a multi-layer neural network with one output. This will be a useful step towards generalising into multi-output architecture in the next unit.

4.3 LOSS FUNCTION

Earlier we saw that for regression the mean squared errors is a useful loss function. With multi-outputs one-layer model we did not need to change the cost function because each output acts independently and has its own loss function. In that case we did not need to tie up these loss functions together because each weight vector lives by its own and do not affect other weight vectors. So for example, one of the weights vectors that corresponds to an output can be frozen while we train the other outputs weights and we still get the same results for the rest of weights when we allow the frozen weights to participate in the training process with the rest of the weights team. The idea that we are trying to convey here is that the different outputs weights vectors are independent.

One the other hand, when we cascade one layer with one output after a multi-layer, similar to the architecture in the figure below, things change dramatically. The weights of the hidden layer $\dot{\mathbf{W}}$ will affect the performance of the consequent layer (output layer) $\mathbf{w}$ and even though we can still treat the output layer weights $\mathbf{w}$ independently the performance of the hidden layer $\dot{\mathbf{W}}$ will directly affect the performance of the whole model. Hence, we need now to coordinate the training of all the outputs together because otherwise changing the hidden layer in isolation according to one of the outputs will have inadvertent implications on the performance of the other outputs (since they are also linked to the hidden layer). For example, if the hidden layer is trained with one of the output is frozen (or by ignoring the performance of one of the outputs), then the resultant hidden weights will be good for all outputs except for the one that we have not incorporated in our training, and hence the performance of the network may become biased against this particular frozen output.

It is true that we can try to make the output weights compensate for the lack of training but this is not guaranteed. We can start by a random hidden layer and train only the output layer (which is one of the main ideas of models called extreme machines). In this case, we would need a large enough hidden layer to give us enough variety and diversity to encode input for the output layer. There are lots of debates around whether we need extreme machines and whether its ideas are novel, in any case it is outside the scope of our coverage.

4.4 OUTPUT LAYER UPDATE FOR ONE-OUTPUT NEURAL NETWORKS

We can now express the loss function for the below architecture as follows

$$\bar{J}^2(\mathbf{w}) = \frac{1}{2N}(\|\mathbf{t} - \mathbf{\Phi w}\|^2)$$

To deal with the derivative with respect to $\mathbf{w}$ we will just express the loss function as a function of the output weights only. Taking the derivatives we get:

$$\nabla\bar{J}^2(\mathbf{w}) = \frac{1}{N}\mathbf{\Phi}^\top(\mathbf{t} - \mathbf{\Phi w})$$

So for the output layer, the update takes the form of the usual linear model since there is no activation function applied on this layer.

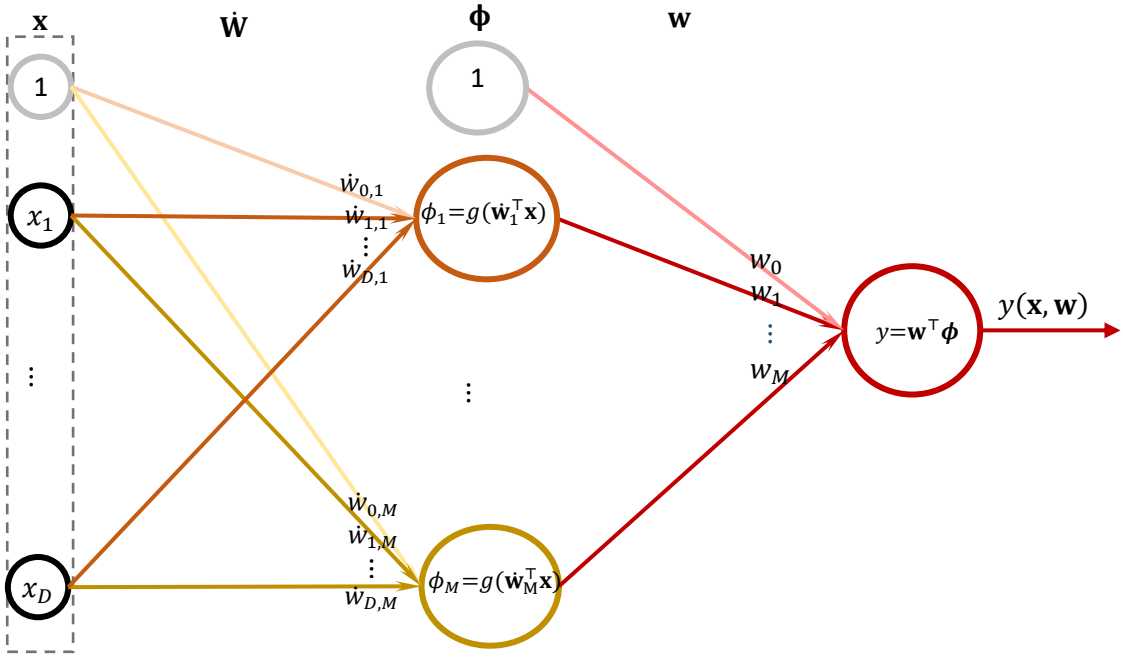

Figure (3.3): schematic representation of the multi-layer perceptron as a non-linear models with one output.

4.5 Hidden Layer Update for One Output Neural Network (Backpropagation)

Ignoring the bias for the time being, let us see how the hidden layer weights can be updated. As usual we need to take the derivative of the loss function and assign to 0. The loss function can be written as a function of the hidden layers weights $\dot{\mathbf{W}}$ as follows

$$\bar{J}^2(\dot{\mathbf{W}}) = \frac{1}{2N}\left\|\mathbf{t} - g(\mathbf{X}\dot{\mathbf{W}})\mathbf{w}\right\|^2$$

This time however, we are taking the derivative of the *output errors* (the loss) with respect to a *hidden layer* weights (we deal with the output weights as if they are fixed). This creates some extra complexity that we have to deal with it. In particular, if we look at the loss function we can realise that the hidden weights are tucked inside an activation function that produces the set of features that the model is learning. To be able to reach it we need to use the chain rule of derivations. As a reminder, the chain rule is used when we have a function of a function. In this case the loss function is a function of the features who are in turn functions of the hidden weights.

The update rule for the hidden layer can be deduced by realising that $g(\mathbf{X}\dot{\mathbf{W}}) = \mathbf{\Phi}$ and its gradient is $\nabla \boldsymbol{g} = \dot{\mathbf{\Phi}} = \mathbf{\Phi} \circ (\mathbf{1} - \mathbf{\Phi})$, where $\circ$ is element-wise matrix multiplication. The gradients $\nabla_{\dot{\mathbf{W}}}(\mathbf{X}\dot{\mathbf{W}}) = \mathbf{X}^\top$ and

$\nabla_{\mathbf{\Phi}}\mathbf{\Phi w} = \mathbf{w}^{\top}$. All of these elements are stitched together to form a backpropagated update for the hidden layer (via the chain rule of derivation).

There is an extra complexity associated with propagating the error back into previous layers. The deeper the error goes back, the more analytical overhead we have. Deep Learning (DL) have overcome these issues via few techniques and tricks. One of the important techniques employed by DL is automatic differentiation (AD). In AD the gradients of the loss with respect to early layers weights are calculated via built-in packages (algorithms) instead of inferring them analytically. It exploits the fact that we use a sequence of elementary operations in the forward pass of the output function. It then applies the chain rule hopping from one operation to the other in a backward manner. This is quite powerful and important tool to be able to update deeper networks. In addition, in order to overcome some of the difficulties of backpropagating the error, DL trains each layer separately and freezes the rest of the layers to be updated one layer at a time. There is also the issue of vanishing gradients for those deeper layers (the ones that are near the input and furthest from the output) that we overcome via adopting simpler non-linear activation functions, such as the ReLUs, and other methods such as batch normalisation. You will cover backpropagation on neural networks in depth in machine learning and deep learning modules.

4.6 Exercise

See the following Jupyter notebook for an example of non-linear regression and neural networks using sklearn.

4.7 Preventing Overfitting for Neural Networks

Please refer back to the previous section and to algorithm 5''.

4.8 Summary

In this section we have covered the basics of neural networks and we took the liberty to simplify its coverage. You will study this topic extensively in machine learning and deep learning. An important aspect of neural networks is that they allow us to represent the arbitrary relationship between input and output, linear and non-linear. Therefore, they are called universal approximator. In the next unit we will take advantage of the knowledge that you gained in dealing with regression to extend it to numerical classification.

See the following presentation for a summary of what we covered in this unit.

5 Learning from A Probabilistic Perspective

Learning outcomes:

After completing this lesson you should be able to:

- understand the optimisation process from a Bayesian framework
- use maximum likelihood method to optimise the loss function
- motivate the regularised least squares using posterior maximisation

In this section we will discuss the link between minimising a loss function and probability theory, to see how learning can take a probabilistic perspective. We will also establish links with a particular probabilistic framework namely the Bayesian framework for learning. This lesson and its subsections can be safely skipped, without consequences on other sections or future sections in the module. Similar material will be also covered in some form in the Machine Learning module.

5.1 MINIMISING LEAST SQUARES LOSS VIA MAXIMISING THE LIKELIHOOD

In conventional probability, that you have studied in high school mostly, we *assume* that the data has a specific distribution like Gaussian or Beta etc. and we use this *assumption* to calculate the probability of an even. But what if we want to do the other way round? What if we want to evaluate our *assumption*? What if we want to evaluate the likelihood that the given data has actually the assumed distribution? We will bootstrap and use the same concepts of probability to come up with a measure that quantify this concept. We call this quantity the *likelihood*.

5.1.1 What is Likelihood?

So, in likelihood we apply the concept of probability into model parameters (hyper parameters and adjustable parameters-weights). In likelihood we vary the hyper parameters of the model (mainly the mean and the variance) and we try to calculate the likelihood that the data comes from the model i.e. the *probability* that the model under consideration is the *correct* model *given* the data. So, we vary the distribution (using its mean and variance) and we measure, using the likelihood estimate, how likely it is that the data has actually come from the distribution. If we consider a continuum of values for the model the likelihood defines a function over these values. The likelihood function for a distribution parameter (mean or variance) might be itself distributed according to some known distribution. An example would be the central limit theorem itself which states that the mean of a data samples of a variable tends to a normal (Gaussian) distribution regardless of the actual distribution of the variable which might be completely different. Let us try this ourselves here.

Likelihood: $L(model \mid data)$ read as: the likelihood of model *given* the data
Probability: $P(data \mid model)$ read as: the probability of data *given* the model

Nevertheless, we calculate the likelihood via probability.

5.2 MAXIMISING THE LIKELIHOOD WITH IDENTICAL MEAN AND VARIANCE

For example, given a *univariate* Gaussian distribution $\mathcal{N}(x|\mu,\sigma^2)$ and a set of data points $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ of size N, the likelihood of this Gaussian model generating *all* the data is the probability that the *all* the given data has come from this distribution (distributed according to the Gaussian). Since the probability of a set of *independent* events taking place together equals to the multiplication of their individual probabilities and given that the dataset $\mathbf{X}$ is *independent and identically distributed* (i.i.d) - identically distributed means all of the data is drawn from the same distribution whether we know the distribution or we try to estimate it. Then, the likelihood of the model given the data is given as the probability $p(\mathbf{X}|\mu,\sigma^2)$ and is calculated as

$$p(\mathbf{X}|\mu,\sigma^2) = \prod_{n=1}^{N} \mathcal{N}(x_n|\mu,\sigma^2) \tag{38.}$$

Note that the variables here are μ, σ^2 since we already have the data points. Our mission would be then to come up with a model that maximise the above likelihood function.

5.3 MAXIMISING THE LIKELIHOOD WITH DIFFERENT MEANS AND IDENTICAL VARIANCE

In a previous section we showed that $y(\mathbf{x},\mathbf{w}) = E_t(t|\mathbf{x})$ and we have stated that this will be our main probability that we will try to estimate in order to minimise the expected loss is $p(t|\mathbf{x})$.

Often we are given a data set pairs $(\mathbf{x}_n, t_n)$ each value t_n is assumed to have a standard univariate Gaussian noise attached to it (t is a variable not a vector hence we are dealing with a univariate Gaussian)

$$t = y(\mathbf{x},\mathbf{w}) + \epsilon \tag{39.}$$

where ϵ is a zeros mean Gaussian random variable $\mathcal{N}(\epsilon|0,\beta^{-1})$ and $\beta = \frac{1}{\sigma^2}$. Therefore, the probability of a label t is given as:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) \tag{40.}$$

which state that the mean for t is $y(\mathbf{x}, \boldsymbol{w})$ and the variance is β^{-1}. This assumption is illustrated in the following figure:

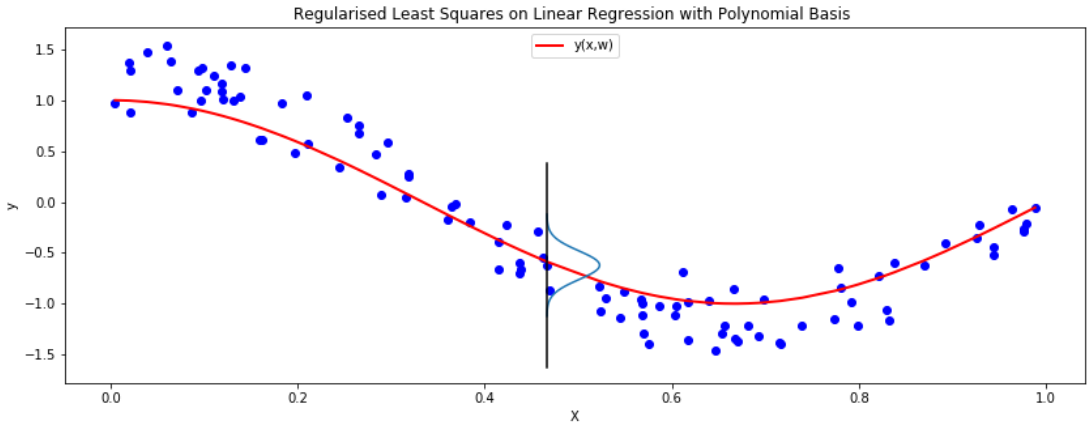

Figure (4.1): Linear model fitting with Gaussian Noise.

Given that we have a dataset $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ and corresponding $\boldsymbol{t} = \{t_1, t_1, \dots, t_N\}$ of size N, the likelihood of the above Gaussian model $\mathcal{N}(t|y(\mathbf{x}, \boldsymbol{w}), \beta^{-1})$ generating *all* the $t_n$is the probability that the *all* the given data has come from this distribution $\mathcal{N}(t|y(\mathbf{x}, \boldsymbol{w}), \beta^{-1})$. We make the assumption that t_n is *independent and identically distributed* (i.i.d). Then, the likelihood of the model given the data is given as the probability $p(\mathbf{t}|\mu, \sigma^2)$ and is calculated as

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}) \tag{41.}$$

5.4 Least Squares via Likelihood Maximisation (ML)

Since in supervised learning settings we will not seek to model the distribution of the input variables, and we have already eliminated the need for this distribution in the previously aforementioned section we will omit the input variables $\mathbf{X}$ to simplify our notation but it should be noted that it is implicitly assumed. And further noting that we have a linear model $y(\mathbf{x}_n, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_n)$

$$p(\mathbf{t}|\mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}\left(t_n|\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}\right) \tag{42.}$$

Recall that $\log(\prod_{n=1}^{N} g_n) = \log(g_1 g_2 \dots g_N) = \log(g_1) + \log(g_2) + \cdots \log(g_N) = \sum_{n=1}^{N} \log(g_n)$ and given that the logarithm is a monotonic function in the input, which means if we maximise $\log(g_1 g_2 \dots g_N)$ we would have maximised the product $g_1 g_2 \dots g_N$. We will seek to maximise $\log\big(p(\boldsymbol{t}|\boldsymbol{w}, \beta)\big)$ to maximise $p(\boldsymbol{t}|\mathbf{w}, \beta)$, therefore:

$$\log\big(p(\boldsymbol{t}|\mathbf{w}, \beta)\big) = \sum_{n=1}^{N} \log\big(\mathcal{N}(t_n|y(\boldsymbol{x}_n, \mathbf{w}), \beta^{-1})\big) \tag{43.}$$

Given that the Gaussian $\mathcal{N}(t_n|y(\boldsymbol{x}_n, \mathbf{w}), \beta^{-1}) = \frac{1}{\sqrt{2\pi}} \beta e^{-\frac{1}{2}\beta(t - \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_n))^2}$:

$$\log\big(p(\boldsymbol{t}|\mathbf{w}, \beta)\big) = \frac{N}{2}\log(\beta) - \frac{N}{2}\log(2\pi) - \beta \underbrace{\frac{1}{2}\sum_{n=1}^{N}(t_n - \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}_n))^2}_{J(\boldsymbol{w})} \tag{44.}$$

where $J(\mathbf{w})$ is the sum of squared error that we have seen in earlier section and where $\mathbf{w} = (w_0, w_1, w_2, \dots, w_{M-1})$ and $\boldsymbol{\phi} = (\phi_0, \phi_1, \phi_2, \dots, \phi_{M-1})^\top$ and $\phi_0 = 1$.

Now to maximise the likelihood we take the derivative for the $\log\big(p(\boldsymbol{t}|\boldsymbol{X},\boldsymbol{w},\beta)\big)$ with respect to $\mathbf{w}$. Note that the first two terms $\frac{N}{2}\log(\beta)-\frac{N}{2}\log(2\pi)$ are not related to $\mathbf{w}$ and their derivatives with respect to $\mathbf{w}$ are 0. Hence:

$$\nabla\log\big(p(\boldsymbol{t}|\mathbf{w},\beta)\big)=\beta\sum_{n=1}^{N}\big(t_n-\mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n)\big)\boldsymbol{\phi}^{\mathrm{T}}(\mathbf{x}_n) \tag{45.}$$

So now we realise that *minimising* the sum of squared errors $J(\mathbf{w})$ is equivalent to maximising the likelihood function. Now we set the derivative to 0 to obtain the $\mathbf{w}^*$ that minimise the loss function: $\beta\sum_{n=1}^{N}\big(t_n-\mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n)\big)\boldsymbol{\phi}^{\top}(\mathbf{x}_n)=0$ noting that β cannot be 0 because it is the inverse of the variance.

Therefore, the final solution that optimise likelihood is identical to the least squares for linear regression with basis and is given as

$$\mathbf{w}^*=\big(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\big)^{-1}\boldsymbol{\Phi}^{\top}\mathbf{t} \tag{46.}$$

Similarly, we can minimise (26) with respect to β to obtain optimal value $\hat{\beta}$ that maximise the likelihood function

$$\hat{\beta}^{-1}=\frac{1}{N}\sum_{n=1}^{N}(t_n-\mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n))^2 \tag{47.}$$

5.5 Least Squares with Regularisation via Posterior Maximisation (MAP)

In this section we show how to infer a linear model learning with regularisation. Regularising the weights helps suppress the weights from changing or growing too much. This often helps prevent the problem of overfitting. First, let us see how we can deal with the linear model learning as a full Bayesian learning problem. The assumption is that we will be given a data point x and a dataset $\mathbf{X}$ and labels $\mathbf{t}$. So far we have used a set of weights to estimate the output. We want to come up with an estimation that is valid for distribution over the weights and we want to marginalise the weights to eliminate their effect on the final output. In particular, we want to estimate $p(t|x,\mathbf{X},\mathbf{t})$ which is not conditional on the weights $\mathbf{w}$.

$$p(t|x,\mathbf{X},\mathbf{t},\beta)=\int p(t|x,\mathbf{w},\beta)p(\mathbf{w}|\mathbf{X},\mathbf{t},\beta)\mathrm{d}\boldsymbol{w} \tag{48.}$$

Previously, we showed how come up with $\mathbf{w}$ that maximise the likelihood. Now, we want to calculate the probability $p(t|x,\mathbf{X},\mathbf{t})$ to account for a full Bayesian treatment for a linear model. Note that from previous treatment we already assume that $p(t|\mathbf{x},\mathbf{w},\beta)=\mathcal{N}\big(\boldsymbol{t}\big|\mathbf{w}^{*\top}\boldsymbol{\phi}(\mathbf{x}_{\boldsymbol{n}}),\boldsymbol{\beta}^{-\mathbf{1}}\big)$ (note that this is different than but related to the likelihood $p(\mathbf{t}|\mathbf{X},\mathbf{w},\beta)$ that we maximised in the previous section) therefore, according to Bayes theorem:

$$p(\boldsymbol{w}|\mathbf{X},\mathbf{t},\beta)\propto p(\mathbf{t}|\mathbf{X},\mathbf{w},\beta)p(\mathbf{w}|\alpha) \tag{49.}$$

where $\propto$ means proportional to, $\propto$ will turn into equality once we normalise $p(\mathbf{t}|\mathbf{X},\mathbf{w},\beta)p(\mathbf{w}|\alpha)$.

The first term in (33) is $p(\mathbf{t}|\mathbf{X},\mathbf{w},\beta)$ is the likelihood function for our model and has been already showed to be Gaussian where we have shown how to come up with $\mathbf{w}^*$ that minimises this likelihood and we showed that it is a Gaussian.

The second term in (33) is $p(\mathbf{w}|\alpha)$ and is called the prior of the weights. It represents the prior assumptions about the weights that we can incorporate in our Bayesian treatment. We will assume that $p(\boldsymbol{w}|\alpha)$ is an isotropic Gaussian (which is the simplest form of a multivariate Gaussian). Therefore the posterior distribution $p(\boldsymbol{w}|\mathbf{X},\mathbf{t},\beta)$ is also Gaussian. In particular we will assume that the prior of the weights has a 0 means vector and has α^{-1} variance i.e.

$$p(\mathbf{w}|\alpha)=\mathcal{N}(\mathbf{w}|\mathbf{0},\alpha^{-1}\boldsymbol{I})=\Big(\frac{\alpha}{2\pi}\Big)^{\frac{M+1}{2}}e^{-\frac{\alpha}{2}\|\mathbf{w}\|^2} \tag{50.}$$

We can then maximise $p(\mathbf{w}|\mathbf{X},\mathbf{t},\beta)$ by finding the most probable value $\boldsymbol{w}$ for it given the data. This is called *posterior maximisation*.

$$p(\boldsymbol{w}|\mathbf{X},\mathbf{t},\beta) \propto \mathcal{N}(\mathbf{t}|\mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n),\beta^{-1})\mathcal{N}(\mathbf{w}|\mathbf{0},\alpha^{-1}\boldsymbol{I})$$

By taking the derivative for the negative logarithm and noting that $\|\boldsymbol{w}\|^2 = \boldsymbol{w}^{\top}\boldsymbol{w}$ we have

$$p(\mathbf{w}|\mathbf{X},\mathbf{t},\beta) \propto \left(\frac{\alpha}{2\pi}\right)^{M+1} e^{-\frac{\beta}{2}\sum_{n=1}^{N}(t_n-\mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n))^2-\frac{\alpha}{2}\boldsymbol{w}^{\top}\boldsymbol{w}}$$

We take the log and drop $\mathbf{X}, \beta$ to simplify the notation:

$$\log\big(p(\mathbf{w}|\boldsymbol{t})\big) = -\frac{\beta}{2}\sum_{n=1}^{N}(t_n - \mathbf{w}^{\top}\boldsymbol{\phi}(\mathbf{x}_n))^2 - \frac{\alpha}{2}\mathbf{w}^{\top}\mathbf{w} + const$$

Hence, maximising the posterior distribution with respect w corresponds to minimising the regularised sum of squared error. By taking the derivative with respect to w and setting it to 0 we have the regularised least squares algorithm as: $\lambda = \frac{\alpha}{\beta}$

$$\nabla \log\big(p(\mathbf{w}|\boldsymbol{t})\big) = \beta\sum_{n=1}^{N}\boldsymbol{\phi}(\mathbf{x}_n)(t_n - \boldsymbol{\phi}^{\top}(\mathbf{x}_n)\mathbf{w}) + \alpha\mathbf{w}$$

$$\left(\sum_{n=1}^{N}\boldsymbol{\phi}(\mathbf{x}_n)\boldsymbol{\phi}^{\top}(\mathbf{x}_n)\right)\mathbf{w}^* = \sum_{n=1}^{N}t_n\boldsymbol{\phi}^{\top}(\mathbf{x}_n) - \lambda\mathbf{w}^*$$

$$\boldsymbol{\Phi}^{\top}\boldsymbol{\Phi}\mathbf{w}^* + \lambda\mathbf{w}^* = \boldsymbol{\Phi}^{\top}\mathbf{t}$$

$$\mathbf{w}^* = (\boldsymbol{\Phi}^{\top}\boldsymbol{\Phi} + \lambda\boldsymbol{I})^{-1}\boldsymbol{\Phi}^{\top}\mathbf{t}$$

If the matrix $(\boldsymbol{\Phi}^{\top}\boldsymbol{\Phi} + \lambda\mathbf{I})$ is invertible then the solution exists.

6 UNIT SUMMARY

In this unit we have covered linear models, generalised linear models and extended their ideas into neural networks for regression. Neural networks are universal approximator, meaning they are capable of modelling any arbitrary relationship between a set of inputs and outputs. In the next unit we will take advantage of the knowledge that you gained in dealing with regression to extend it to numerical classification.

7 DISCUSSION

We would like to discuss the issues of optimising gradient descent in the context of neural networks. You would need to read the following paper and discuss the main take away messages and useful practical suggestion for training a non-linear deep models. Do you think the mentioned tips also apply for the case of shallow network (the one we covered here) which one does and which one does not apply in your opinion?

8 DATASETS

See the following datasets for different types of regression here

9 REFERENCES

Christopher M. Bishop. 2006. Pattern Recognition and Machine Learning (Information Science and Statistics), chapters 3 and 5, Springer-Verlag, Berlin, Heidelberg. Online version can be found here.