DYNAMIC PRICING WITH NEURAL NETWORKS

Daniel Xukun Ma

Supervisor: Dr Quoc Thong Le Gia

School of Mathematics and Statistics

UNSW Sydney

November 2023

Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Science with Honours

Plagiarism statement

I declare that this thesis is my own work, except where acknowledged, and has not been submitted for academic credit elsewhere.

I acknowledge that the assessor of this thesis may, for the purpose of assessing it:

- Reproduce it and provide a copy to another member of the University; and/or,
- Communicate a copy of it to a plagiarism checking service (which may then retain a copy of it on its database for the purpose of future plagiarism checking).

I certify that I have read and understood the University Rules in respect of Student Academic Misconduct, and am aware of any potential plagiarism penalties which may apply.

By signing this declaration I am agreeing to the statements and conditions above.

Signed:  Date: 17/11/2023

Acknowledgements

By far the greatest thanks must go to my supervisor Quoc Thong Le Gia for the guidance, care, and support he provided me throughout this year.

Many thanks to the friends and people who I interacted with during this whole journey while working on this thesis, and for just being there throughout this year.

I am also extremely grateful to my parents for their support in what has been a tough year for me personally and professionally.

17 November 2023.

Abstract

Dynamic pricing is a widely adopted strategy that involves setting commodity prices in response to real-time market supply and demand dynamics. This thesis presents a novel approach to this practice, by employing neural networks to simulate dynamic pricing mechanisms. Through extensive experimentation, we establish that neural networks employing radial basis functions and long short-term memory outperform conventional techniques including linear regression. This finding underscores the potential for more accurate and responsive pricing strategies in dynamic market environments.

Contents

Chapter 1

Introduction

Dynamic pricing is a pricing strategy that has gained widespread popularity in recent times due to the increased availability of data and technology.

This strategy involves adjusting the price of a product or service in real-time based on various factors including market demand, consumer behaviour, demographics, and competitor pricing [19].

In dynamic pricing, the price is not fixed but changes over time, and as a result would offer the right price to the right customer at the right time depending on the supply and demand of the market at that time [16]. This allows businesses to achieve their main objective by using dynamic pricing and optimising revenue and profits.

Dynamic pricing can be implemented in various ways, such as using algorithms to analyse real-time data, monitoring competitors' prices, and adjusting prices based on the time of day or day of the week. It is used in different industries, including transportation, hospitality, e-commerce, and entertainment.

In this thesis, we focus on the use of neural networks (both feed-forward neural networks and recurrent neural networks), with the objective of predicting the price of a service at a given time based on supply and demand at that particular time. We will compare the performances of neural networks with linear regression, one of the simplest modelling techniques used for prediction.

To overcome the lack of a comprehensive set of data, we generate simulated realistic datasets based on the summary of real data from Uber apps from Schroder's paper [22].

The main theme of the thesis will be dynamic pricing with application to ride-sharing services, more specifically Uber. We will construct a model to predict the dynamic price of a service at any time of day.

In order to do that, we first need to generate datasets for the price of an Uber ride-share service over a period of 18 hours.

Once we have that complete, we will then consider using function approximation and surface plotting to ensure that we have data that account for the changes in price due to changes in demand, supply and/or time.

While dynamic pricing has shown to be an effective way to increase revenue, it can also lead to concerns about price discrimination and the ethical implications of using data and algorithms to set prices for products. It has also raised issues of fairness for consumers, as well as potential negative impacts on brand reputation.

Overall, dynamic pricing is a complex and constantly evolving field that requires a deep understanding of pricing strategies, data analysis, and consumer behaviour.

The original contributions of the thesis are listed below:

- A new methodology for generating 1-dimensional and 2-dimensional datasets for dynamic pricing simulations.
- Python code to create our Radial Basis Function Network 2-dimensional models, for both Uniform Grid Clustering and K-Means Clustering.
- Everything listed above is provided in this GitHub repository: https://github.com/DanielXMa/honours-dynamic_pricing

The structure of the thesis is as follows:

1. In Chapter 2, we review dynamic pricing from the economic theory point of view, and critically analyse real-life examples of dynamic pricing being used in various industries.

2. In Chapter 3, we provide a high-level description of neural networks and focus on recurrent neural networks and radial basis function networks.

3. In Chapter 4, we explore the dataset that we have sourced from [22] and describe the steps taken to generate a realistic dataset for our simulations. Additionally, we will develop our pricing model based on demand and supply, and outline the methodology undertaken for our simulations using neural networks.

4. In Chapter 5, we will analyse the results from our various models and compare their performances of predicting the dynamic price using various neural network models and classical modelling techniques for prediction.

5. Finally, in Chapter 6, we will make conclusions and discuss future work that could be explored beyond this thesis.

Chapter 2

Dynamic pricing from economics

2.1 Introduction

Dynamic pricing is a pricing strategy where businesses continually adapt the price of products or services by accounting for various external factors, including changes in demand and supply, and competitor pricing. This aims to maximise profit for a business by operating at multiple price points [16]. This contrasts with static pricing, where a business will have a single price point at which it operates. Dynamic pricing is a responsive and flexible strategy, while static pricing adheres to a single unchanging price point.

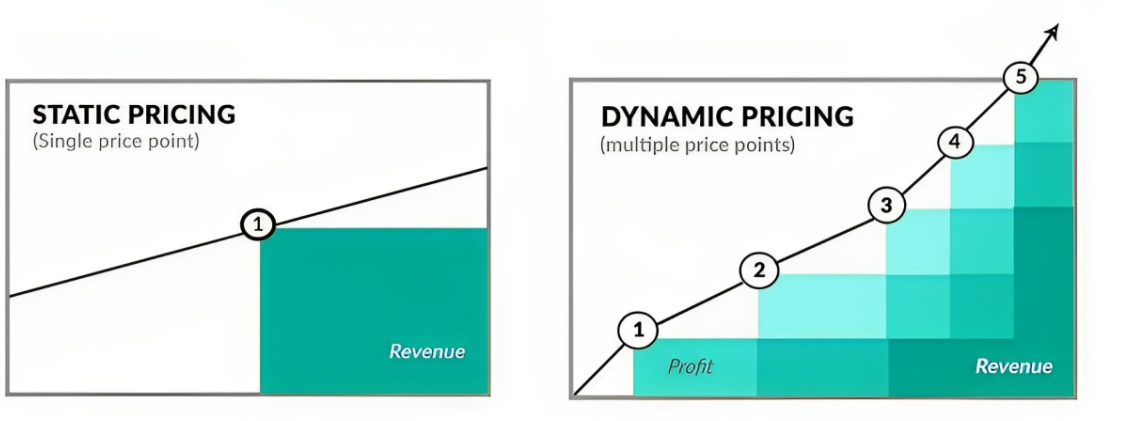


Figure 2.1: Static pricing vs. Dynamic pricing (image from [6])

This concept of dynamic pricing finds its roots in the fundamental principles of microeconomics, particularly the well-known law of supply and demand. As supply and demand levels shift within a given market, dynamic pricing ensures that the price

of a product or service adjusts accordingly [6]. It's a strategy that not only meets the ever-changing consumer landscape but also positions businesses to maximise their returns by harmonising their pricing strategies with the real-time variables of the market.

Dynamic pricing encompasses a diverse array of strategies, each with its own distinctive approach to stimulating demand and optimising the sale of products and services. These strategies are dynamic in the sense that they continuously adapt prices in response to various market conditions and consumer behavior [16]. Each of these strategies plays a distinct role in stimulating demand by aligning prices with the ever-changing conditions of the market and consumer behavior. They offer businesses the flexibility to optimise their pricing strategies and capture the maximum value from their offerings. We will explore some of the many strategies in the next section.

2.2 Types of dynamic pricing

2.2.1 Time-based Pricing

Time-based pricing is a strategic approach in which shifts in demand or supply are correlated with specific time intervals, resulting in fluctuations in the price of a product over a designated period. This dynamic pricing model acknowledges that market conditions can vary significantly based on factors such as seasons, holidays, or even daily routines. For instance, during peak hours or festive seasons, when demand escalates, prices may surge to capitalise on increased consumer interest [25]. Conversely, during off-peak times, prices may be reduced to stimulate sales and maintain market competitiveness.

Furthermore, businesses leverage this strategy not only to adapt to market dynamics but also to influence consumer behavior. By strategically adjusting prices in accordance with anticipated demand patterns, companies aim to entice customers into making impulsive purchasing decisions.

Time-based pricing offers a multitude of benefits, including the ability to maximise revenue by charging higher prices when demand is at its peak and encouraging

customers to engage in spontaneous purchases. It also aids in efficient inventory management, helping businesses align their stock with the ebb and flow of demand [14].

However, this strategy comes with challenges, such as the need to manage customer perceptions of frequent price changes and the potential for price wars when competitors employ similar tactics.

Overall, time-based pricing stands as a versatile tool for businesses seeking to navigate the dynamic ebb and flow of market demand, all while strategically influencing consumer purchasing behavior.

2.2.2 Segmented Pricing

Segmented pricing, also known as differential pricing, is a strategic pricing approach that involves offering different price points for the same product or service to cater to diverse audience segments, each of which perceives the value of the product differently [20]. This pricing strategy recognises that not all customers are the same, and their willingness to pay for a product or service can vary significantly based on various factors, which we'll explore more deeply below.

1. Customer Demographics: Segmented pricing takes into account the differences in customer characteristics such as age, gender, income level, education, and occupation. For example, a luxury fashion brand may offer a premium line at a higher price point to attract affluent customers while simultaneously offering more affordable options to cater to budget-conscious shoppers.

2. Socio-economic Standards: People from different socio-economic backgrounds have varying levels of disposable income. Segmented pricing adjusts prices to accommodate these disparities. An illustration of this could be a fitness club that provides different membership tiers, with varying prices and benefits, to attract both budget-conscious individuals and those willing to pay more for premium facilities.

3. Location: Geographic location plays a crucial role in segmented pricing. Businesses may alter their pricing based on regional or local factors. For instance,

an airline might charge different fares for the same flight route, depending on the origin and destination, as well as the time of booking. This reflects the principles of supply and demand in different markets.

Segmented pricing can have several advantages:

- Maximising Revenue: By offering different price points to various segments, a company can optimise its revenue by capturing value from customers willing to pay more without alienating those seeking a more budget-friendly option.
- Market Expansion: Segmented pricing enables businesses to tap into different market segments, widening their customer base and potentially increasing market share.
- Customer Satisfaction: Customers appreciate choices, so segmented pricing allows customers to select the product or service that best fits their needs and budget, increasing overall satisfaction.

However, implementing segmented pricing can be complex, requiring in-depth market research, pricing strategy development, and effective communication to prevent customer confusion or backlash.

In conclusion, segmented pricing is a pricing strategy that recognises and caters to the diversity of customer preferences, demographics, socio-economic factors, and regional disparities. By tailoring prices to these different segments, businesses can enhance their competitiveness, boost revenue, and satisfy a broader range of customers. The bottom line is if we do not segment effectively, then we leave money on the table [20].

2.2.3 Peak Pricing

Peak pricing is a dynamic pricing strategy that shares similarities with time-based pricing. In peak pricing, the cost of a product or service is adjusted to increase during specific months or seasons, focusing on a particular peak period rather than continually fluctuating prices [9]. This approach allows retailers and businesses to harness the shifts in supply and demand, capitalising on economic principles to maximise their profits. Here's a more detailed exploration of peak pricing:

1. Seasonal Variation: Peak pricing predominantly operates during specific seasons or months when there is a notable surge in demand for particular products or services. This is often linked to seasonal changes in consumer preferences, such as the demand for warm clothing during winter or beachwear in the summer. Retailers anticipate these fluctuations and adjust their prices accordingly.

2. Supply and Demand Dynamics: Peak pricing hinges on the fundamental principle of supply and demand. When demand for a product is high or the supply is limited, prices are raised to balance the market. Conversely, during periods of lower demand or surplus supply, prices may be reduced to stimulate purchases. This strategy ensures that businesses make the most of their resources and inventory.

3. Strategic Timing: Peak pricing is strategically timed to coincide with events or periods that drive heightened consumer activity [11]. Common examples include:

    - Holidays: Around major holidays, such as Christmas or Valentine's Day, demand for certain products like gifts, decorations, and travel services tends to spike. Retailers adjust their prices to capture the increased demand, even if it's just for a short duration.

    - Sporting Events: During major sporting events like the Super Bowl, the World Cup, or the Olympics, businesses in related industries, such as sports bars, merchandise vendors, and travel agencies, implement peak pricing strategies to make the most of the high demand for their products and services.

    - Music Events: Concerts, music festivals, and other live music events can draw large crowds, and businesses in the entertainment, hospitality, and transportation sectors may increase prices during these events due to the surge in demand.

Peak pricing offers various benefits for businesses:

- Revenue Maximisation: By adjusting prices to match the ebb and flow of demand, businesses can capitalise on the most lucrative moments and optimise their revenue.
- Resource Management: This strategy helps businesses efficiently manage their resources and inventory. During peak periods, they can allocate their resources to meet heightened demands and generate higher profits.
- Customer Experience: Peak pricing encourages customers to plan purchases strategically, potentially reducing overcrowding and improving the overall customer experience, as customers have an incentive to shop or use services during off-peak times when prices are lower.

However, it's crucial for businesses to implement peak pricing thoughtfully, as it can sometimes lead to customer dissatisfaction if customers perceive the strategy as price gouging. Effective communication and transparency in pricing changes are essential to maintain customer trust.

In summary, peak pricing is a dynamic pricing approach that takes advantage of fluctuations in demand during specific periods or events. By adjusting prices to align with consumer behavior, businesses can optimise their revenue and better manage their resources.

2.2.4 Penetration Pricing

Penetration pricing is a strategic pricing approach employed by businesses to launch a new product into the market. This strategy involves initially setting a price for the product that is lower than the eventual market price, especially in price-sensitive markets where new products face strong competition soon after introduction [24]. The primary objective of penetration pricing is to entice customers to try the new product, thereby capturing a significant market share. Here's a more detailed exploration of this pricing strategy:

1. Market Entry: When a new product is introduced to the market, it often faces competition from existing alternatives. Penetration pricing is a way for

businesses to gain a foothold in the market by offering an attractive initial price that can lure consumers away from established brands or products.

2. Price Advantage: The lower introductory price creates a competitive advantage, making the new product more appealing to price-conscious consumers. It encourages them to make the switch from their current options to the new offering. This approach can help the product quickly gain visibility and popularity.

3. Customer Acquisition: Penetration pricing is designed to rapidly acquire new customers. The idea is that, once people try the product and have a positive experience, they will continue to purchase it even as the price gradually increases. This strategy relies on building a loyal customer base during the initial stages of product launch.

4. Gradual Price Increase: Over time, as the product gains traction in the market and demand begins to increase, businesses gradually raise the price of the product. The goal is to eventually reach the target market price that the company deems appropriate for the product's value and positioning.

Penetration pricing offers several advantages for businesses [2]:

- Market Share Expansion: By offering a lower price, businesses can quickly gain a substantial portion of the market. This can be especially advantageous for new entrants or companies trying to challenge established competitors.

- Fast Product Adoption: Customers are more likely to experiment with a new product when it is competitively priced. Penetration pricing accelerates the adoption process and can create early brand loyalists.

- Competitive Advantage: By offering an attractive initial price, businesses can position themselves favorably in the market, potentially disrupting the competition and drawing attention to their product.

However, there are challenges associated with penetration pricing:

- Profit Margin: Initially, businesses may operate with thin profit margins due to the low introductory price. It's essential to balance the need to capture market share with the goal of profitability.
- Customer Expectations: Customers who have become accustomed to lower prices may be resistant to price increases in the future. Effective communication is crucial to managing customer expectations.

In conclusion, penetration pricing is a strategy used by businesses to launch new products successfully. By initially setting a lower price than the eventual market value, they can attract customers, capture a significant market share, and gradually increase prices as the product gains traction in the market. This approach requires a careful balance between acquiring customers and maintaining profitability.

2.2.5 Competitive Pricing

Competitive pricing, also known as competition-based pricing, is a pricing strategy that primarily relies on monitoring and responding to competitors' pricing rather than being driven by the traditional factors of supply and demand. In this approach, a business continuously evaluates the prices set by its competitors, and its pricing strategy is based on setting its prices slightly lower than those of its rivals. The primary objective is to attract more interest and demand from customers [26]. Here's a more comprehensive explanation of competitive pricing:

1. Competitor-Centric Approach: This pricing strartegy places competitors at the center of the pricing strategy. Rather than focusing on internal factors like production costs or desired profit margins, the business primarily looks externally at the prices set by competitors.
2. Price Benchmarking: To implement competitive pricing effectively, a business needs to closely monitor the pricing strategies of its competitors. This often involves regular market research and price tracking to stay updated on changes in the competitive landscape.
3. Price Undercutting: The core tactic of competitive pricing is to set prices slightly lower than those of key competitors. By doing so, the business aims

to capture the attention of price-sensitive customers and encourage them to choose their product or service over those of competitors.

4. Driving Interest and Demand: Lower prices relative to competitors can create a sense of value and attract customers seeking a good deal. As a result, this pricing strategy can stimulate increased interest and demand for the product or service.

5. Market Share and Customer Loyalty: Competitive pricing can be a strategic choice for businesses seeking to expand their market share or gain an advantage in a highly competitive industry. By offering lower prices and attracting more customers, a business can work on building customer loyalty and potentially cross-selling or upselling additional products or services.

Competitive pricing offers several advantages:

- Market Visibility: By being competitively priced, a business is more likely to be noticed by customers comparing options in the market.
- Market Penetration: This strategy can help new entrants gain a foothold in the market and challenge established competitors.
- Customer Acquisition: It is particularly effective at acquiring price-sensitive customers who are looking for the best deal.

However, there are potential challenges and drawbacks to competitive pricing:

- Profit Margins: Competitive pricing can put pressure on profit margins, as businesses may need to sacrifice higher profits to stay competitive.
- Price Wars: A competitive pricing strategy may trigger price wars with rivals, which can be detrimental to all parties involved.
- Sustainability: Businesses need to assess the long-term sustainability of competitive pricing and consider whether it aligns with their overall business goals.

In conclusion, competitive pricing is a pricing strategy that prioritises responding to competitors' prices rather than supply and demand dynamics. By undercutting competitors' prices, businesses aim to attract price-sensitive customers, stimulate

interest, and increase demand. This approach can be effective for businesses looking to gain market share and establish themselves in a competitive market, but it also requires careful monitoring of competitors and consideration of its impact on profitability.

2.3 Examples of dynamic pricing

2.3.1 In ride-sharing services

In this section, we summarise an example of dynamic pricing from Uber [22].
Their model for price p of the service is decomposed into two parts: a base cost and a surge cost. The base cost is a flat fee that is dependent on duration and distance, much like any taxicab fare. However, the surge cost is an additional cost that is determined by their surge pricing algorithm. This algorithm comes into play during specific times of the day, notably during commuting hours, due to fluctuations in supply and demand, by actively responding to imbalances in supply and demand — a classic example of time-based pricing.
The price surges serve a dual purpose. Firstly, it encourages customers to consider delaying their requests, especially during peak demand periods. By doing so, they can minimise or even completely avoid the surge pricing which in turn, helps alleviate the demand surge. This would then ensure a more even distribution of service requests throughout the day.

Secondly, the surge cost incentivises drivers to offer their services in high-demand areas or during peak hours. The higher earnings potential during these times motivates drivers to meet the increased demand, thus boosting the supply.

As a result, the pricing model not only dynamically responds to the fluctuations in supply and demand but also actively encourages behavior that helps balance these factors, leading to a more efficient and responsive service ecosystem.

2.3.2 In airline services

This section will look at another example of dynamic pricing, this time in airline services.

Fiig challenged the processes in which current product offerings beyond flights are provided for consumers, which as a result, negatively affected profitability in the airline industry. He proposed an integrated offer management system (OMS) [4] that, while not currently feasible but is only an idea, he believes will optimise revenue from flights and improve profitability.

Dynamic pricing is introduced as an emergent idea, that can actively adjust the price of flight products given customer and contextual information. Fiig suggested offers as a way to satisfy their objective of dynamic pricing: to "maximise contribution by dynamically pricing the product offer, considering both customer and contextual information."

In summary, dynamic pricing is still a concept in this paper and requires future research to assess the validity of their offer management system. Fiig also mentioned applying machine learning techniques for more complex pricing situations.

2.3.3 In e-commerce

Finally, we explore how dynamic pricing is used in e-commerce.
Dynamic pricing is a strategy that many enterprises have implemented to dynamically adjust prices. With the continuous development of artificial intelligence (AI) technology, increasingly more scholars have sought to use AI methods to solve dynamic pricing problems. Companies that use dynamic pricing to adjust the price of goods and services have started to use deep reinforcement learning as a way to maximise efficiency in doing so.

In Yin and Han's paper [27], they applied deep reinforcement learning technology to the field of dynamic pricing by building an intelligent dynamic pricing system, which trains the model based on previous information, to predict a price based on the demand and other factors.

The goal mentioned by Yin and Han [27] of using deep reinforcement learning in dynamic pricing is to maximise long-term benefits, and hence the technical means of deep reinforcement learning would be able to achieve the intelligent pricing of goods and services.

However, it was not clear as to what particular neural network models were specifically used for their prediction model. Their steps provide options as to what model framework to use, such as convolutional neural networks, or recurrent neural networks, but Yin and Han do not indicate their own procedure clearly as to how they created their model.

[1]

[1]This chapter was written with assistance from ChatGPT for some initial drafts. However, the final version has been edited completely by the author. This is according to UNSW policy on the use of generative AI tools.

Chapter 3

Neural networks

This chapter will introduce neural networks, starting with the simplest structure called perceptron to multi-layered and recurrent neural networks. The materials presented in this chapter can be found in [3, 17, 1, 7, 12, 10, 23].

3.1 Overview of neural networks

Artificial neural networks (henceforth, simply *neural networks*), as a computational technique, are a collection of neural units loosely modelling the way a biological brain solves problems with large clusters of neurons connected by axons. Nowadays, neural networks have been used to solve a wide variety of tasks, like computer vision, speech recognition and image recognition [13].

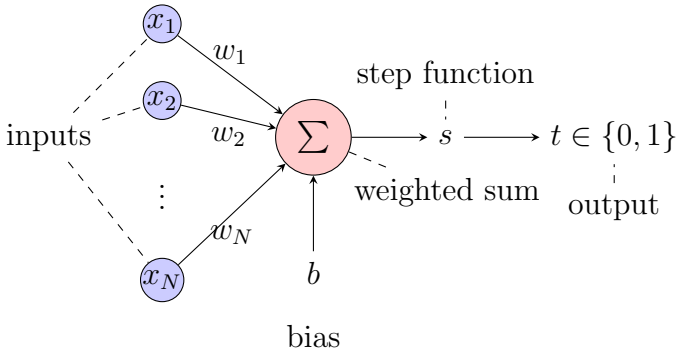
In the following, we start off by looking at a single neuron to understand how the process works.

3.1.1 The perceptron

Before we begin to explain what is a neural network, we should first look at a type of artificial neuron called a perceptron, developed by Rosenblatt [21] in 1958.

So how do perceptrons work? A perceptron will take in several binary inputs, say x_1, x_2, ..., x_N and will produce a single binary output. By binary, we mean that the value is either 1 or 0.

The perceptron's output is then determined by whether the weighted sum $\sum_{j=1}^{N} w_j x_j$ is less than or greater than some threshold value $\theta \in \mathbb{R}$. Here the real numbers $w_1, w_2, \ldots, w_N$ are called the weights.



Let us introduce the perceptron's bias, $b := -\theta$. Using the bias instead of threshold, the perceptron output t can be written as the value of a step function s. If we let $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$, then

$$t = s(z) = \begin{cases} 0, & \text{if } z \leq 0, \\ 1, & \text{if } z > 0. \end{cases}$$

3.1.2 Sigmoid neurons

Sigmoid neurons (also known as *sigmoidal neurons*) are similar to perceptrons, but they are modified so that small changes in their weights and bias cause only a small change in their output. The inputs $x_1, x_2, \ldots, x_N$ to the sigmoid neuron are real-valued. Similarly to the perceptron, the sigmoid neuron has weights for each input, $w_1, w_2, \ldots, w_N$, and an overall bias, b. But the output is not just 0 or 1. Again, if we let $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$, then the output t is given by

$$t = \sigma(z) = \frac{1}{1 + e^{-z}}, \tag{3.1.1}$$

where $0 < \sigma(z) < 1$. The function σ is called the *sigmoid* or *sigmoidal function*. It is possible to use any similar function in place of the one given, for example tanh. Such functions are known as *activation functions*, as they determine whether or not the neuron is activated (the biological metaphor is whether the neuron fires or not). In the following work, we will use a general function σ.

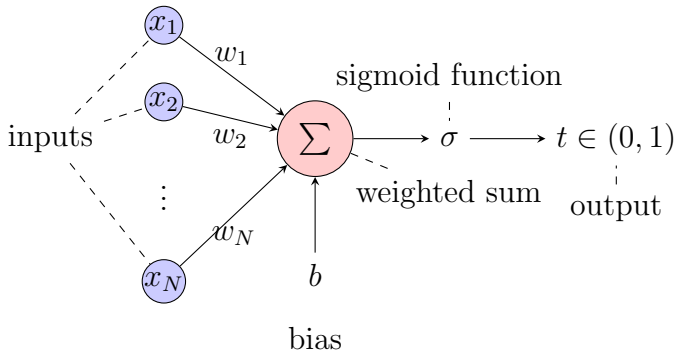


Table 3.1 includes the formulas of these 4 popular activation functions.

Activation function	Formula
Gaussian	$\Phi(z) = e^{-(z-c)^2/(2\alpha^2)}$
sigmoid	$\sigma(z) = \dfrac{1}{1+e^{-z}}$
tanh	$\tanh(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$

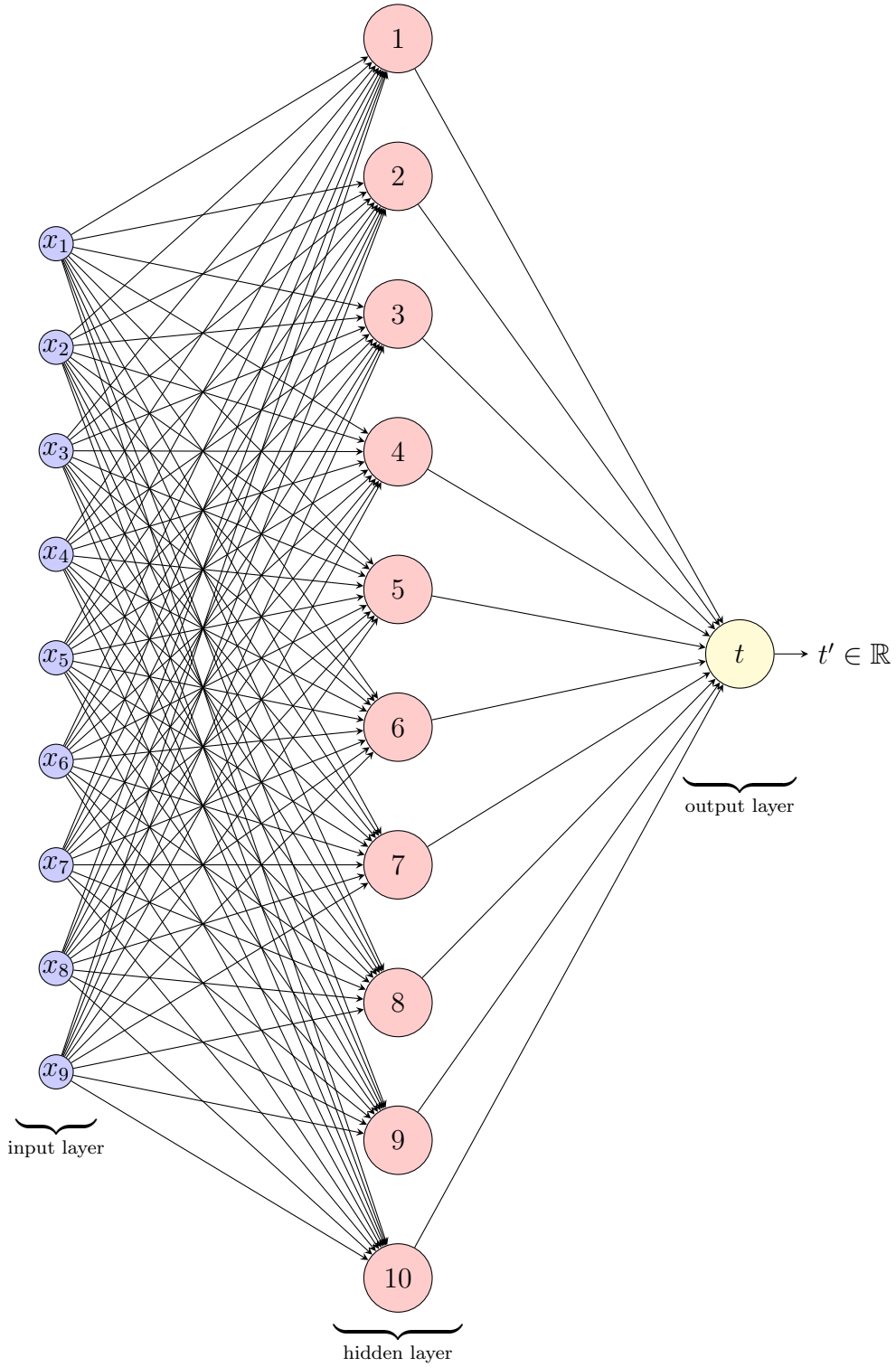
Table 3.1: Popular activation functions and their formulas.

3.1.3 Networks of neurons

A neural network, as defined in this thesis, is a collection of many sigmoid neurons, arranged in layers. It is more illustrative to examine Figure 3.1.

The leftmost layer in this network is called the *input layer*, and the neurons within the layer are called *input neurons*. This name may be slightly misleading, as these are simply the untransformed input values $x_1, x_2, \ldots, x_N$. The rightmost layer is called the *output layer* and contains the *output neurons*, or, as in this case, a single output neuron. This neuron computes a value $t \in \mathbb{R}$. The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs. The network below has a single hidden layer for simplicity, but it is possible to have as many hidden layers as desired. Neurons in this layer are unsurprisingly called *hidden neurons*.

Figure 3.1: Architecture of a neural network.



The architecture of the neural network can be summarised using graph theoretic terms. It is a directed, acyclic graph with subsequent layers forming complete bi-

partite graphs (if direction is ignored). So-called *sparse networks* can be formed by deleting some of the connections, or having connections bypass layers. In the first case, this is analogous to setting the respective weights to zero. In our work, we will consider the architecture in Figure 3.1.

It should be noted that multilayer perceptrons are also possible. Neural networks, as defined above, are sometimes known as multilayer perceptrons, despite the lack of perceptrons. In any case, the architecture of the network is similar.

3.2 Training a neural network

Training a neural network is the process of computing the weights and biases of the network based on a given training dataset D. Specifically, the training dataset is a set of N vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N \in \mathbb{R}^n$, where we will write $\boldsymbol{x}_i = \left(x_i^{(1)}, x_i^{(2)} \ldots, x_i^{(n)}\right)^\top$ for $i = 1, \ldots, N$, for which we know the desired outputs $y_1, y_2, \ldots, y_N \in \mathbb{R}$. So, it is in fact a set of ordered pairs,

$$D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}.$$

3.2.1 A cost function

Our network "learns" by adjusting its parameters such that some error is reduced. It follows naturally that we should define a cost function with respect to the weights and biases of the network. Let $\boldsymbol{w}, \boldsymbol{b}$ be vectors containing the weights and biases of the network. Then the cost function C may be defined as

$$C(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{2N} \sum_{i=1}^{N} |y_i - t_i|^2,$$

where $|\cdot|$ is the usual absolute value. Here t_i is the output of the network for data $\boldsymbol{x}_i$. Since $t_i = t_i(\boldsymbol{w}, \boldsymbol{b})$, C is clearly a function of $\boldsymbol{w}$ and $\boldsymbol{b}$. This is the well-known mean-squared error. Clearly, $C(\boldsymbol{w}, \boldsymbol{b})$ is non-negative. In general, any similar cost function may be used.

The goal of the training algorithm is to find the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ so that the value of the cost C is minimised. That is, $C(\boldsymbol{w}, \boldsymbol{b}) \approx 0$. This occurs when y_i is approximately equal to the output, t_i, for all training inputs, $\boldsymbol{x}_i$.

In short, this is achieved through using an optimisation algorithm. This could be the stochastic gradient descent in Algorithm 1, wherein derivatives of the cost function are computed via the backpropagation algorithm in Algorithm 2, or the Adam algorithm in Algorithm 3.

3.2.2 Learning with (stochastic) gradient descent

Given a general architecture with weights $\boldsymbol{w}$, biases $\boldsymbol{b}$ and a cost function C, we wish to find $\boldsymbol{w}$, $\boldsymbol{b}$ such that $C(\boldsymbol{w}, \boldsymbol{b}) = 0$. Unfortunately, it is in general intractable to find analytic solutions to $C(\boldsymbol{w}, \boldsymbol{b}) = 0$ and $\nabla C(\boldsymbol{w}, \boldsymbol{b}) = 0$. Furthermore, C is typically non-convex and contains many non-linear dependencies on the weights and biases. Hence, there may be many points in the weight space where ∇C is approximately zero. We thus rely on iterative procedures that will converge to a minimum of the cost function. In practice, we do not know whether the procedure has found a local or global minimum, or even a true minimum of the function. Nevertheless, it may be prudent to examine several different minima found and to take the best one.

To this end, we will consider the gradient descent algorithm, which we will later develop into the stochastic gradient descent algorithm. Gradient descent is an iterative optimisation algorithm to find a local minimum of a multivariate, differentiable function.

Consider some general multivariate, real-valued function $C(\boldsymbol{v})$, where

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_K)^T.$$

Assume that C is continuous and differentiable and is convex. We wish to find a global minimum of C.

Consider a small perturbation $\Delta\boldsymbol{v} = (\Delta v_1, \Delta v_2, \ldots, \Delta v_K)$. Then the change in C, given by $\Delta C = C(\boldsymbol{v} + \Delta\boldsymbol{v}) - C(\boldsymbol{v})$ can be computed approximately by a first-order

Taylor approximation:

$$\begin{aligned}\Delta C &= C(\boldsymbol{v}+\Delta\boldsymbol{v}) - C(\boldsymbol{v}) \\ &= C(\boldsymbol{v}) + \nabla C(\boldsymbol{v})\cdot\Delta\boldsymbol{v} + O((\Delta\boldsymbol{v})^2) - C(\boldsymbol{v}) \\ &= \nabla C(\boldsymbol{v})\cdot\Delta\boldsymbol{v} + O((\Delta\boldsymbol{v})^2) \\ &\approx \nabla C(\boldsymbol{v})\cdot\Delta\boldsymbol{v},\end{aligned} \tag{3.2.1}$$

where ∇C is the gradient of the function C defined by

$$\nabla C = \left(\frac{\partial C}{\partial v_1},\ldots,\frac{\partial C}{\partial v_K}\right)^T.$$

So if we choose

$$\Delta\boldsymbol{v} = -\eta\nabla C(\boldsymbol{v}),$$

where η is a small, positive parameter then it follows from Equation (3.2.1) that

$$\Delta C \approx -\eta\nabla C(\boldsymbol{v})\cdot\nabla C(\boldsymbol{v}) = -\eta\|\nabla C(\boldsymbol{v}\|^2 \leq 0,$$

which makes the function C decrease. That is $C(\boldsymbol{v}+\Delta\boldsymbol{v}) \leq C(\boldsymbol{v})$. The parameter η is known as the *learning rate*, and is normally set to 0.1. This value is simply a heuristic determined from practice.

The gradient descent algorithm is summarised in Algorithm 1. Note that $\leftarrow$ is the assignment operator and in fact, it is not necessary to set both a maximum number of iterations and a tolerance. It may be desirable to set only a maximum number of iterations (in the interest of time) or only a tolerance (in the interest of precision).

When applying the gradient descent algorithm to the training of a neural network, the weights $\boldsymbol{w} = \{w_k : k = 1,2,\ldots\}$ and the biases $\boldsymbol{b} = \{b_\ell : \ell = 1,2,\ldots\}$ are initialised randomly. Then the update rules for each component of the weights and

Algorithm 1 The gradient descent algorithm.

Set a maximum number of iterations K and a tolerance $\tau > 0$.

Fix $\eta > 0$.

Initialise $\boldsymbol{v}$ randomly.

Initialise $i \leftarrow 0$.

while $i < K$ and $C(\boldsymbol{v}) > \tau$ **do**

$\boldsymbol{v}' \leftarrow \boldsymbol{v} - \eta \nabla C(\boldsymbol{v})$

$\boldsymbol{v} \leftarrow \boldsymbol{v}'$

$i \leftarrow i + 1$

end while

the biases are given by

$$w_k' \leftarrow w_k - \eta \frac{\partial C}{\partial w_k}, \tag{3.2.2}$$

$$b_\ell' \leftarrow b_\ell - \eta \frac{\partial C}{\partial b_\ell}, \tag{3.2.3}$$

$$w_k \leftarrow w_k', \tag{3.2.4}$$

$$b_\ell \leftarrow b_\ell.' \tag{3.2.5}$$

By repeatedly applying the update rules, we decrease the cost function at each iteration step. We should note that this algorithm does not necessarily find a true minimum of the function C. Furthermore, whether the minimum is local or global is also unknown. Since $C(\boldsymbol{w}, \boldsymbol{b}) \geq 0$, if we find $\boldsymbol{w'}, \boldsymbol{b'}$ such that $C(\boldsymbol{w'}, \boldsymbol{b'}) = 0$ (at least to machine precision), then this point may be regarded as a global minimum. For practical purposes, finding exactly such a minimum may not be desirable. As long as the point minimises C to within some tolerance, the neural network will perform satisfactorily. Alternatively, we may wish to examine $|\Delta C|$. If the current value for $|\Delta C|$ is sufficiently small (that is, the function is not decreasing very much for subsequent iterations), we may decide to terminate the algorithm.

There are many additional challenges in applying the gradient descent rule. One major problem is when the number of training inputs is very large. When this happens, computing the gradient may be computationally expensive for large datasets.

This can be seen by observing that the cost function C is of the form

$$C(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{N} \sum_{i=1}^{N} C_{\boldsymbol{x}_i}.$$

That is, it is an average over costs $C_{\boldsymbol{x}_i} = |y_i - t_i|^2/2$ for individual training examples $\boldsymbol{x}_i$. Hence, as the number of examples grows, the time complexity of calculating the error grows.

In order to speed up the computation time, an algorithm called stochastic gradient descent will be used. A small number $M \geq 1$ of training inputs are randomly chosen, which we denote $X_1, X_2, \ldots, X_M$. If the sample size M is large enough, we expect the average value of ∇C_{X_j}, for $j = 1, 2, \ldots, M$, to be approximately equal to the average over all $\nabla C_{\boldsymbol{x}_i}$ by the central limit theorem. In other words,

$$\frac{1}{M} \sum_{j=1}^{M} \nabla C_{X_j} \approx \frac{1}{N} \sum_{i=1}^{N} \nabla C_{\boldsymbol{x}_i} = \nabla C(\boldsymbol{w}, \boldsymbol{b}). \tag{3.2.6}$$

We can rewrite Equation (3.2.6) as

$$\nabla C \approx \frac{1}{M} \sum_{j=1}^{M} \nabla C_{X_j}. \tag{3.2.7}$$

To apply the idea to training our network, suppose again that $\boldsymbol{w} = \{w_k : k = 1, 2, \ldots\}$ and $\boldsymbol{b} = \{b_\ell : \ell = 1, 2, \ldots\}$ contain the weights and the biases. The rules to update the weights and the biases at each iteration are given by

$$w_k' \leftarrow w_k - \frac{\eta}{M} \sum_{j=1}^{M} \frac{\partial C_{X_j}}{\partial w_k}, \tag{3.2.8}$$

$$b_\ell' \leftarrow b_\ell - \frac{\eta}{M} \sum_{j=1}^{M} \frac{\partial C_{X_j}}{\partial b_\ell}, \tag{3.2.9}$$

$$w_k \leftarrow w_k', \tag{3.2.10}$$

$$b_\ell \leftarrow b_\ell'. \tag{3.2.11}$$

3.2.3 The backpropagation equations

Although our previous work may give us some hope for an easy solution, in fact we have not mentioned how the gradient may be calculated. Typically, explicit calculation of the gradient is not feasible. Thus, a numerical method is used. In this subsection, we describe the standard *backward propagation of errors algorithm*, or simply the *backpropagation algorithm*. This algorithm was first described in the 1960s by Kelley in the 1960s, with subsequent developments to the modern standard technique. We will continue with the derivation using the chain-rule.

We begin with some notations. Let us denote w_{jk}^{ℓ} the weight for the connection from the k^{th} neuron in the $(\ell-1)^{\text{th}}$ layer to the j^{th} neuron in the $(\ell-1)^{\text{th}}$ layer to the j^{th} neuron in the ℓ^{th} layer. For example, in Figure 3.2, the weight w_{24}^{3} is on the connection from the 4^{th} neuron in the 2^{nd} layer to the 2^{nd} neuron in the 3^{rd} layer of a network. We use similar notations for the network's biases and activation functions. Let us denote by b_j^{ℓ} the bias of the j^{th} neuron in the ℓ^{th} layer, and denote by a_j^{ℓ} the *activation* of the j^{th} neuron in the ℓ^{th} layer. The activation a_j^{ℓ} of the j^{th} neuron in the ℓ^{th} layer is related to the activation in the $(\ell-1)^{\text{th}}$ layer by the following equation

$$a_j^{\ell} = \sigma\left(\sum_k w_{jk}^{\ell} a_k^{\ell-1} + b_j^{\ell}\right), \tag{3.2.12}$$

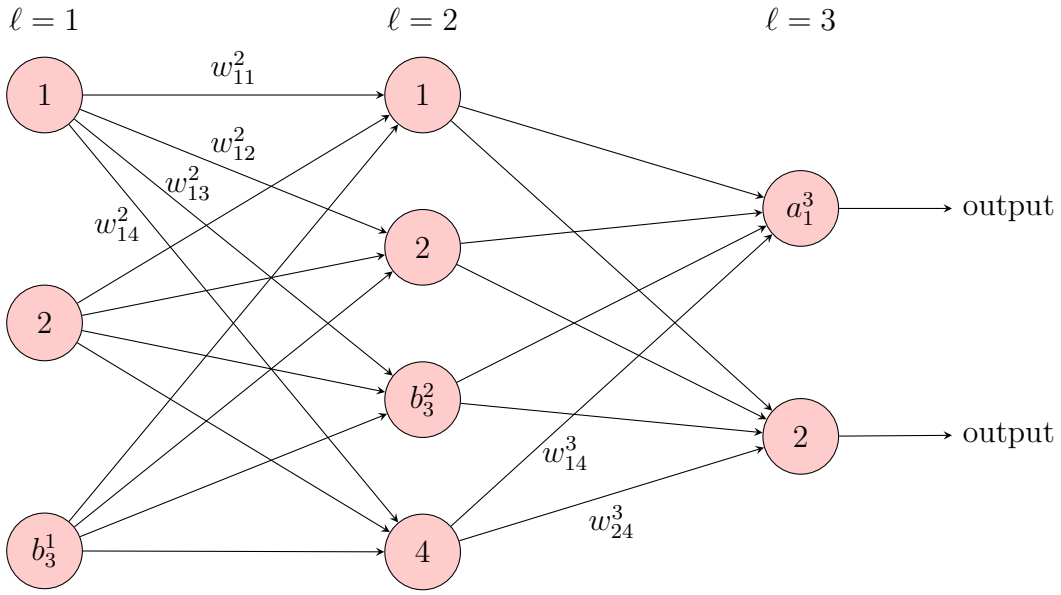
where the sum is over all neurons in the $(\ell-1)^{\text{th}}$ layer, indexed by k. An example of these notations is given again in Figure 3.2.

We now introduce the *weight matrix* $\boldsymbol{w}^{\ell}$ for the ℓ^{th} layer. The entries of the matrix $\boldsymbol{w}^{\ell}$ are just w_{jk}^{ℓ}, with the rows indexed by j and columns by k, as usual. Similarly, for each layer ℓ we define the *bias vector*, $\boldsymbol{b}^{\ell}$, with entries b_j^{ℓ}. Finally, the activation vector $\boldsymbol{a}^{\ell}$ just contains the entries a_j^{ℓ}. So 3.2.12 can be rewritten in the more compact form

$$\boldsymbol{a}^{\ell} = \sigma(\boldsymbol{z}^{\ell}), \quad \text{with} \quad \boldsymbol{z}^{\ell} = \boldsymbol{w}^{\ell}\boldsymbol{a}^{\ell-1} + \boldsymbol{b}^{\ell}, \tag{3.2.13}$$

where it is understood that when the sigmoid function σ is applied to a vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_K)^{\top}$ it will return the vector $(\sigma(v_1), \sigma(v_2), \ldots, \sigma(v_K))^{\top}$.

Figure 3.2: Notations for a neural network with three layers. For certain neurons, some weights are indicated, as are some biases and activations.



Before introducing the backpropagation algorithm, we introduce the four fundamental backpropagation equations to be used in the algorithm. Each equation will be treated in its own section, before being employed in Algorithm 2. Fix some $\boldsymbol{x}_i$ randomly chosen from the training dataset. Then the backpropagation equations are given by

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L), \tag{\textbf{BP1}}$$

$$\boldsymbol{\delta}^\ell = ((\boldsymbol{w}^{\ell+1})^\top \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\boldsymbol{z}^\ell), \tag{\textbf{BP2}}$$

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell, \tag{\textbf{BP3}}$$

$$\frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell. \tag{\textbf{BP4}}$$

The conscientious reader may be confused by the partial derivatives involving C. Firstly, we note that $\boldsymbol{t}_i = \boldsymbol{a}_i^L$, where $\boldsymbol{a}_i^L$ is the activation in the L^{th} layer for some input $\boldsymbol{x}_i$. Hence the term $\nabla_{\boldsymbol{a}^L} C$ should be interpreted with respect to some general $\boldsymbol{x}$. In fact, the stochastic gradient descent algorithm takes only an average over the subset of training data, as shown in Equations (3.2.6) and (3.2.7). Hence, for a fixed

$\boldsymbol{x}_i$ (given above), the quantity to be differentiated is then

$$C_{\boldsymbol{x}_i} = \frac{1}{2}|y_i - a_i^L|^2,$$

where $i \in \{1, 2, \ldots, N\}$. By calculating $\nabla_{\boldsymbol{a}^L} C_{\boldsymbol{x}_i}$ for a fixed value of $i \in \{1, 2, \ldots, N\}$ (that is, calculating $\nabla_{\boldsymbol{a}^L} C_{\boldsymbol{x}_i}$ one at a time), we will obtain the approximation given by (3.2.7). We will continue to suppress the subscript $\boldsymbol{x}_i$ for ease of notation. Note that we assume throughout the rest of this chapter that C and $C_{\boldsymbol{x}_i}$ are both differentiable.

We briefly introduce some notation to be used in the following sections. Firstly, we let $\odot$ define the element-wise product of vectors, also known as the *Hadamard* or *Schur* product. For example, let $\boldsymbol{x}$, $\boldsymbol{y}$ be vectors of the same length n. Then $\boldsymbol{x} \odot \boldsymbol{y} = (x_1 y_1, x_2 y_2, \ldots, x_n y_n)^T$. By $\boldsymbol{\delta}^\ell$, we denote a vector of values δ_j^ℓ. By δ_j^ℓ we denote the *error* of the j^{th} neuron in the ℓ^{th} layer of the network. The definition is then

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell},$$

where $z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$, similarly to 3.2.13. In other words, z_j^ℓ is the value to be computed by the j^{th} neuron in the ℓ^{th} layer, which is calculated by the neuron's weights and bias and the activations of the neurons in the previous layer. Then $\boldsymbol{\delta}^\ell$ is the vector of errors in the ℓ^{th} layer. By $\boldsymbol{z}^\ell$, we denote the vector of values z_j^ℓ. The definitions of these quantities will be reiterated as needed.

***BP1**: Error in the output layer*

Suppose the network has L layers and let $\boldsymbol{\delta}^L$ denote the error in the output layer. The vector $\boldsymbol{\delta}^L$ has components $\delta_1^L, \delta_2^L, \ldots$. Begin by recalling that for each component δ_j^L, we have

$$\delta_j^L = \frac{\partial C}{\partial z_j^L},$$

by definition. It follows from the chain-rule that

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \tag{3.2.14}$$

where the neurons are indexed by k. Recall that the activation of the k^{th} neuron in the output layer depends only on its input. That is, a_k^L is independent of z_j^L for all $k \neq j$. Hence,

$$\frac{\partial a_k^L}{\partial z_j^L} = 0, \text{ for } k \neq j.$$

Therefore, 3.2.14 is simply reduced to

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}.$$

Recall that $a_j^L = \sigma(z_j^L)$, which implies that $\dfrac{\partial a_j^L}{\partial z_j^L} = \dfrac{da_j^L}{dz_j^L} = \sigma'(z_j^L)$. Hence,

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$

It then follows that

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L),$$

where $\odot$ is the element-wise product of vectors. This is **BP1**, as required.

BP2*: Error backpropagation*

This equation gives a "backwards-recursive" definition of the errors of the ℓ^{th} layer $\boldsymbol{\delta}^\ell$, which depends on information from the $(\ell+1)^{\text{th}}$ layer.

As before, recall from the definition that

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}.$$

Then, again applying the chain-rule,

$$\begin{aligned}\delta_j^\ell &= \frac{\partial C}{\partial z_j^\ell} \\ &= \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \\ &= \sum_k \delta_k^{\ell+1} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \\ &= \sum_k \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \delta_k^{\ell+1}, \end{aligned} \tag{3.2.15}$$

where, again, the neurons are indexed by k and we have used the definition of $\delta_k^{\ell+1}$ to obtain the second-last line.

In order to evaluate the partial derivative $\dfrac{\partial z_k^{\ell+1}}{\partial z_j^\ell}$, recall that

$$z_k^{\ell+1} = \sum_j w_{kj}^{\ell+1} a_j^\ell + b_k^{\ell+1} = \sum_j w_{kj}^{\ell+1} \sigma(z_j^\ell) + b_k^{\ell+1}.$$

Then, for a fixed j,

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = w_{kj}^{\ell+1} \sigma'(z_j^\ell).$$

Thus, by substituting this into 3.2.15, we obtain

$$\delta_j^\ell = \sum_k w_{kj}^{\ell+1} \delta_k^{\ell+1} \sigma'(z_j^\ell).$$

Then $\boldsymbol{\delta}^\ell$ is a vector of these δ_j^ℓ. Recall that the weight matrix $\boldsymbol{w}^{\ell+1}$ defines the weights for the neurons in the $(\ell+1)^{\text{th}}$ layer. Then,

$$\boldsymbol{\delta}^\ell = ((\boldsymbol{w}^{\ell+1})^\top \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\boldsymbol{z}^\ell),$$

as required.

***BP3**: Rate of change of the cost with respect to any bias*

Again, we use the chain rule on the definition of δ_j^ℓ, which yields

$$\begin{aligned}\delta_j^\ell &= \frac{\partial C}{\partial z_j^\ell} \\ &= \sum_k \frac{\partial C}{\partial b_k^\ell}\frac{\partial b_k^\ell}{\partial z_j^\ell} \\ &= \frac{\partial C}{\partial b_j^\ell}\frac{\partial b_j^\ell}{\partial z_j^\ell}\end{aligned}$$

since $z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$ implies that $\dfrac{\partial b_k^\ell}{\partial z_j^\ell} = 0$ for all $k \neq j$. Furthermore, it also implies that $\dfrac{\partial b_j^\ell}{\partial z_j^\ell} = 1$. Hence, we obtain

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell,$$

as required. This relation is useful since (**BP1**),(**BP2**) tell us how to calculate δ_j^ℓ explicitly.

***BP4**: Rate of change of the cost with respect to any bias*

Observe that

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}^\ell} &= \frac{\partial C}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial w_{jk}^\ell} \\ &= \delta_j^\ell \frac{\partial z_j^\ell}{\partial w_{jk}^\ell}.\end{aligned}$$

From the definition of z_j^ℓ, observe that $\dfrac{\partial z_j^\ell}{\partial w_{jk}^\ell} = a_k^{\ell-1}$ for a fixed k. Hence,

$$\frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1}\delta_j^\ell,$$

as required.

3.2.4 The backpropagation algorithm

The equations developed in the previous subsection are summarised below:

$$\boldsymbol{\delta}^L = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L), \tag{BP1}$$

$$\boldsymbol{\delta}^\ell = ((\boldsymbol{w}^{\ell+1})^\top \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\boldsymbol{z}^\ell), \tag{BP2}$$

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell, \tag{BP3}$$

$$\frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell. \tag{BP4}$$

Note that the algorithm first requires the (untrained) network to process a training datapoint before errors can be found. This is the *feedforward* stage of the process. This feedforward process is repeated to obtain new values of C. The entire algorithm is given in Algorithm 2.

Algorithm 2 The backpropagation algorithm

Set the corresponding activations $\boldsymbol{a}^1$ for the input layer.

for $\ell = 2$ to L **do** (feedforward)

$\boldsymbol{z}^\ell \leftarrow \boldsymbol{w}^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell$,

$\boldsymbol{a}^\ell \leftarrow \sigma(\boldsymbol{z}^\ell)$.

end for

Calculate the output error $\boldsymbol{\delta}^L \leftarrow \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L)$ using **BP1**.

for $\ell = L - 1$ **down to** 2 **do** (backpropagate the error using **BP2**)

$\boldsymbol{\delta}^\ell \leftarrow ((\boldsymbol{w}^{\ell+1})^T \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\boldsymbol{z}^\ell)$.

end for

The gradient of the cost function is given by

$$\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell \quad \text{and} \quad \frac{\partial C}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell,$$

using (**BP3**),(**BP4**).

3.2.5 Adam algorithm

Adam (short for Adaptive Moment Estimation) is an optimisation algorithm used in machine learning and deep learning to update the parameters of a model.

We first present the algorithm as in Algorithm 3 and discuss some of its characteristics. It is a popular algorithm in large-scale applications of neural networks as it relies on first-order gradients to update parameters without involving matrix calculations, and avoids storing many parameter values.

The Adam algorithm uses a moving average of the gradients, denoted as m_k and $\widehat{m}_k$, and the second moments of the gradients, denoted v_k and $\widehat{v}_k$, to adaptively update the learning rate for each parameter. The stability of the updates across iterations are then controlled via the exponential decay parameters β_1 and β_2 for the estimated first and second gradient moments respectively. The hatted values for the moments are there to address the bias of the early values of m_k and v_k, which will be close to zero for small k. Moreover, the $\varepsilon > 0$ term is a sufficiently small constant used to prevent divisions by zero or related numerical problems.

The particular update used by Adam is invariant to diagonal rescaling of the gradient by a scalar. This is quite simple to see by multiplying g_k by some scalar and seeing how it feeds into the calculation of $m_k/\sqrt{v_k}$ (or the bias-corrected quantities). It is also mentioned how the effective size of the parameter update used by Adam, that is $\|\Delta_k\| = \eta\widehat{m}_k/\sqrt{\widehat{v}_k}$ (after setting $\varepsilon = 0$) is generally bounded by the step-size η in most cases. This can be useful to have more direct control over the updates involved in the training of a neural network and potentially provide ways to estimate the right step-size to complete an optimisation problem within a certain number of steps. However, in cases where prior information on the regions where the optimal neural network weights are unknown, the Adam algorithm can appear to place increased importance on the selection of a good learning rate η to achieve convergence. A potential remedy provided by the authors is that the updates performed by Adam provided some level of automatic annealing of η, where often, the first moment of the gradients becomes closer to 0 as Adam nears an optimum point and the second moment (or the variance) of the gradients increases in the same situation. Intuitively,

the factor $\widehat{m}_k/\sqrt{\widehat{v}_k}$ represents some normalised quantity that provides a direction for Adam to search for an optimal solution, with its magnitude unimportant due to it being bounded by η, and as the algorithm nears an optimum, the stochastic updates can cause the gradients g_k to change drastically over a small number of steps. As a result, the effective step-size $\|\Delta_k\|$ will tend to 0 when close to an optimal point, allowing Adam to perform some level of annealing in the process.

The convergence analysis of Adam is based on the regret measure, defined as the summation of the differences between the prediction $f(\boldsymbol{\theta}_k, \xi_k)$ from the best-fixed point prediction $f(\boldsymbol{\theta}^*, \xi_k)$ for all the previous steps in the sense

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}\in\Theta} \sum_{k=1}^{K} f(\boldsymbol{\theta}, \xi_k),$$

that is

$$R(K) := \sum_{k=1}^{K} [f(\boldsymbol{\theta}_k, \xi_k) - f(\boldsymbol{\theta}^*, \xi_k)]. \tag{3.2.16}$$

It is then shown that Adam has an $\mathcal{O}(\sqrt{K})$ regret bound (Theorem 4.1, pp. 4) and that the average regret $R(K)/K$ achieves a bound of $\mathcal{O}(1/\sqrt{K})$. This is slightly different to the results given for SGD with fixed step-sizes. The Adam algorithm has various advantages:

- Magnitude of parameter updates is invariant to size of gradient
- Step-size is approximately bounded by chosen learning rate
- Naturally performs step-size annealing
- Objective is not required to be stationary (lessens the need for batch normalisation).

In summary, the Adam algorithm is extremely popular as it incorporates most of the advantages of other descent algorithms, and often performs competitively with respect to the best of the other methods.

Algorithm 3 Adam (Adaptive Moment estimation)

Require: $\eta > 0$: learning rate hyperparameter

Require: $\beta_1,\ \beta_2 \in [0, 1)$: exponential decay rates

Require: $f(\theta)$: objective function with respect to learnable parameters θ

Require: θ_0 : initial parameter value

$m_0 \leftarrow 0$ ▷ initialise 1st moment vector

$v_0 \leftarrow 0$ ▷ initialise 2nd moment vector

$k \leftarrow 0$ ▷ initialise current time step

while θ_k not converged **do**

$k \leftarrow k + 1$

Generate random variable ξ_k.

$g_k \leftarrow \nabla_\theta f(\theta_{k-1}, \xi_k)$ ▷ get gradient vector of objective wrt θ

$m_k \leftarrow \beta_1 m_{k-1} + (1 - \beta_1) \cdot g_k$ ▷ update biased 1st moment estimate

$v_k \leftarrow \beta_2 v_{k-1} + (1 - \beta_2) \cdot g_k^2$ ▷ update biased 2nd moment estimate

$\widehat{m}_k \leftarrow m_k / (1 - \beta_1^k)$ ▷ compute unbiased 1st moment estimate

$\widehat{v}_k \leftarrow v_k / (1 - \beta_2^k)$ ▷ compute bias-correct 2nd moment estimate

$\theta_k \leftarrow \theta_{k-1} - \eta \widehat{m}_k / (\sqrt{\widehat{v}_k} + \varepsilon)$ ▷ update parameters

end while

return θ_k

3.3 Recurrent neural networks

Traditional neural networks have one major shortcoming, which is it is unable to use reasoning from previous values to inform later ones. This is known as the long-term dependency problem.

This type of architecture would not be suitable for modelling time series data, since we would like to use information from the past to predict the future. This motivates us to explore recurrent neural networks (RNNs), a class of networks with loops in them that allow information to persist and thus better predict the future, addressing this issue.

3.3.1 *Simple Recurrent Neural Networks*

Consider a chunk of a neural network, $\mathbf{A}$, which receives some input $\boldsymbol{x}_t \in \mathbb{R}^{d_1}$ and produces the output $\boldsymbol{h}_t \in \mathbb{R}^{d_2}$. It also loops the output back to itself, as shown in Figure 3.3 (left).

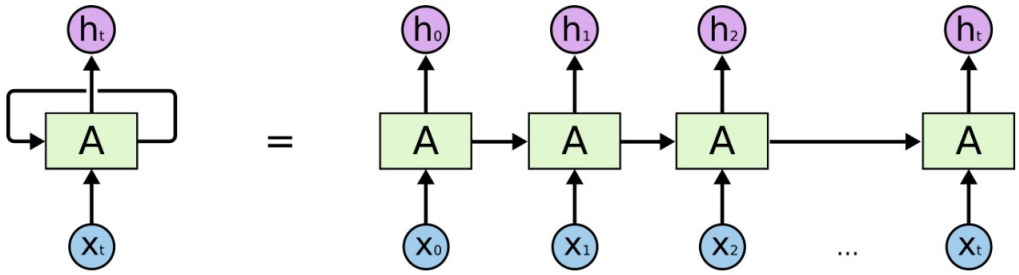


Figure 3.3: A recurrent neural network (left) unrolled through time (right). From [18].

To understand what is happening behind the scenes, we can unroll the network through time as shown in Figure 3.3. We see that at each time step t, the network receives two inputs: $\boldsymbol{x}_t$ and the output from the previous time step, $\boldsymbol{h}_{t-1}$. It is this second input that allows information to be passed on at every step and makes the network recurrent.

Since the network will receive two inputs at each time step, we also need two weight matrices: $\boldsymbol{V}^x \in \mathbb{R}^{d_2 \times d_1}$ for the inputs $\boldsymbol{x}_t$ and $\boldsymbol{V}^h \in \mathbb{R}^{d_2 \times d_2}$ for the previous output $\boldsymbol{h}_{t-1}$. Then at any time step t, the output $\boldsymbol{h}_t$ is given by

$$\boldsymbol{h}_t = \phi\left(\boldsymbol{V}^x \boldsymbol{x}_t + \boldsymbol{V}^h \boldsymbol{h}_{t-1} + \boldsymbol{v}\right) \tag{3.3.1}$$

where $\phi(\cdot)$ is the activation function and $\boldsymbol{v} \in \mathbb{R}^{d_2}$ is the bias vector. Note that for the first time step $t = 0$, the output from the previous time step does not exist, so we set it to be $\mathbf{0}$. As a result, (3.3.1) then simplifies to

$$\boldsymbol{h}_0 = \phi\left(\boldsymbol{V}^x \boldsymbol{x}_0 + \boldsymbol{v}\right)$$

at the first time step $t = 0$.

Although simple RNNs can capture information from the current and previous time step, they face issues with long-term dependencies - when the desired output depends on inputs near the start of the sequence. This results in gradients that may vanish or explode, where the gap between inputs and output gets larger, which means a simple RNN is unable to learn to connect the information. This brings us to look at the LSTM network - a special type of RNN capable of learning long-term dependencies.

3.3.2 Long Short-Term Memory Networks

The Long Short-Term Memory (LSTM) neural network was introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [10], designed to tackle the long-term dependency problem. We first give a high-level explanation of how it works, before diving deeper into the computations involved.

We have seen that in a simple RNN, the output at any time step t is calculated via equation (3.3.1), which is just the computation of a single neural network layer. However, the structure of an LSTM network is slightly more complicated as it consists of four neural network layers that interact with each other, as shown in Figure 3.4. An LSTM network contains a cell state, which can essentially be thought of as the

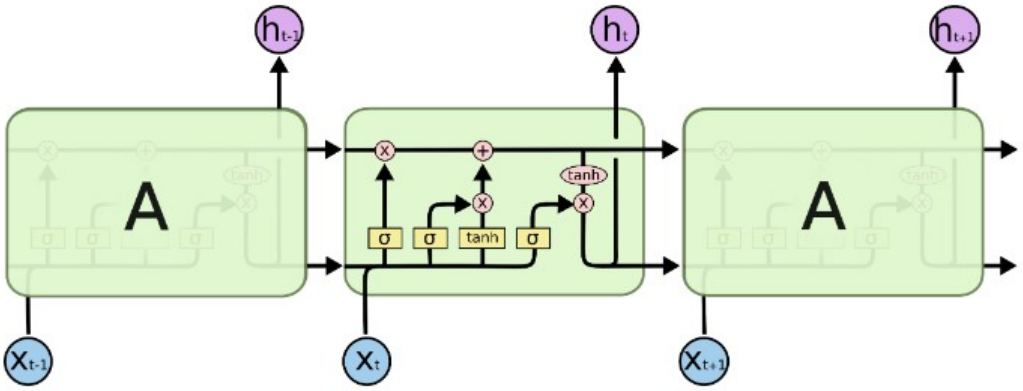


Figure 3.4: The structure of an LSTM. From [18].

long-term memory. It is shown as the horizontal line running through the top of Figure 3.4. Information can either be added or removed from the cell state via gates - a combination of a sigmoid layer and a pointwise multiplication operation. This

provides a short-term memory that can last thousands of time-steps, hence its name long short-term memory.

Let us continue with the same notation as with simple RNNs, where $\boldsymbol{x}_t \in \mathbb{R}^{d_1}$ is an input and $\boldsymbol{h}_t \in \mathbb{R}^{d_2}$ is the corresponding output. Define $\boldsymbol{c}_t \in \mathbb{R}^{d_2}$ to be the cell state. We now introduce some new notation to represent the weights and biases corresponding to the four neural network layers in an LSTM. A subscript f, i, c or o indicates if the weight matrix $\boldsymbol{V}$ or bias vector $\boldsymbol{v}$ is associated with the forget gate layer, input gate layer, a candidate cell layer, or an output gate layer respectively. A superscript x or h is used to indicate whether the quantity corresponds to the current input $\boldsymbol{x}_t$ or the previous output $\boldsymbol{h}_{t-1}$. For example, $\boldsymbol{V}_f^h$ denotes the weight matrix associated with the previous output $\boldsymbol{h}_{t-1}$ in the forget gate layer. We now go through the computations performed in each of these layers.

Forget Gate

As the LSTM network processes new information, some old information also needs to be removed from the cell state. This is done by the forget gate. The sigmoid layer outputs a number between 0 and 1 for each element in the cell state $\boldsymbol{c}_{t-1}$, with

$$\boldsymbol{f}_t = \sigma(\boldsymbol{V}_f^h \boldsymbol{h}_{t-1} + \boldsymbol{V}_f^x \boldsymbol{x}_t + \boldsymbol{v}_f).$$

The forget gate will then perform element-wise multiplication between $\boldsymbol{f}_t$ and $\boldsymbol{c}_t$. A value closer to 0 suggests the LSTM network should forget this element, whereas a value closer to 1 represents the LSTM network should keep this element.

Input Gate

The LSTM network now needs to decide what new information should be stored in the cell state. This process is performed in two steps. The first part involves the input gate in which the sigmoid layer outputs a number between 0 and 1 for each number in the cell state to decide which values to update. This is represented by the following equation

$$\imath_t = \sigma(\boldsymbol{V}_i^h \boldsymbol{h}_{t-1} + \boldsymbol{V}_i^x \boldsymbol{x}_t + \boldsymbol{v}_i).$$

The candidate cell layer with a general activation function ϕ then creates a vector of new candidate values $\tilde{\boldsymbol{c}}_t$ that can be added to the cell state, with

$$\tilde{\boldsymbol{c}}_t = \phi(\boldsymbol{V}_c^h \boldsymbol{h}_{t-1} + \boldsymbol{V}_c^x \boldsymbol{x}_t + \boldsymbol{v}_c).$$

Note that the tanh activation function is typically used by default. It is now time to update the old cell state as described in the previous steps, resulting in the new cell state

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \imath_t \odot \tilde{\boldsymbol{c}}_t$$

where $\odot$ is the Hadamard product.

Output Gate

The final step of an LSTM involves the output gate that decides which part of the cell state should be outputted. It is essentially a filtered version of the cell state, and can be computed by the following equations:

$$\begin{aligned} \boldsymbol{o}_t &= \sigma(\boldsymbol{V}_o^h \boldsymbol{h}_{t-1} + \boldsymbol{V}_o^x \boldsymbol{x}_t + \boldsymbol{v}_o) \\ \boldsymbol{h}_t &= \boldsymbol{o}_t \odot \phi(\boldsymbol{c}_t) \end{aligned}$$

where ϕ is the same activation function used in the candidate cell layer.

3.4 Radial Basis Function Neural Networks

A radial basis function (RBF) is a particular function whose value depends on the distance to a center $\boldsymbol{c}$ in the input space. Normally, Euclidean distance is the distance used. The most commonly used RBF is a Gaussian RBF, which has the same form as the kernel of the Gaussian probability density function (pdf) and it is defined as

$$\Phi(\boldsymbol{x}) = \exp\left(\frac{-1}{2s^2}\|\boldsymbol{x} - \boldsymbol{c}\|^2\right). \tag{3.4.1}$$

We note that this is the same as the Gaussian in Table 3.1, however, $\boldsymbol{x}$ and $\mathbf{c}$ could be vectors in $\mathbb{R}^n$.

Radial Basis Function (RBF) Neural Networks are a commonly used type of artificial neural network that tend to be used for function approximation problems. They are a special class for feed-forward neural networks and are distinguished from other neural networks because of their universal approximation and faster learning speed [1].

The architecture of an RBF network typically consists of three layers: an input layer, a single hidden layer, and an output layer. This can be shown in Figure 3.5.

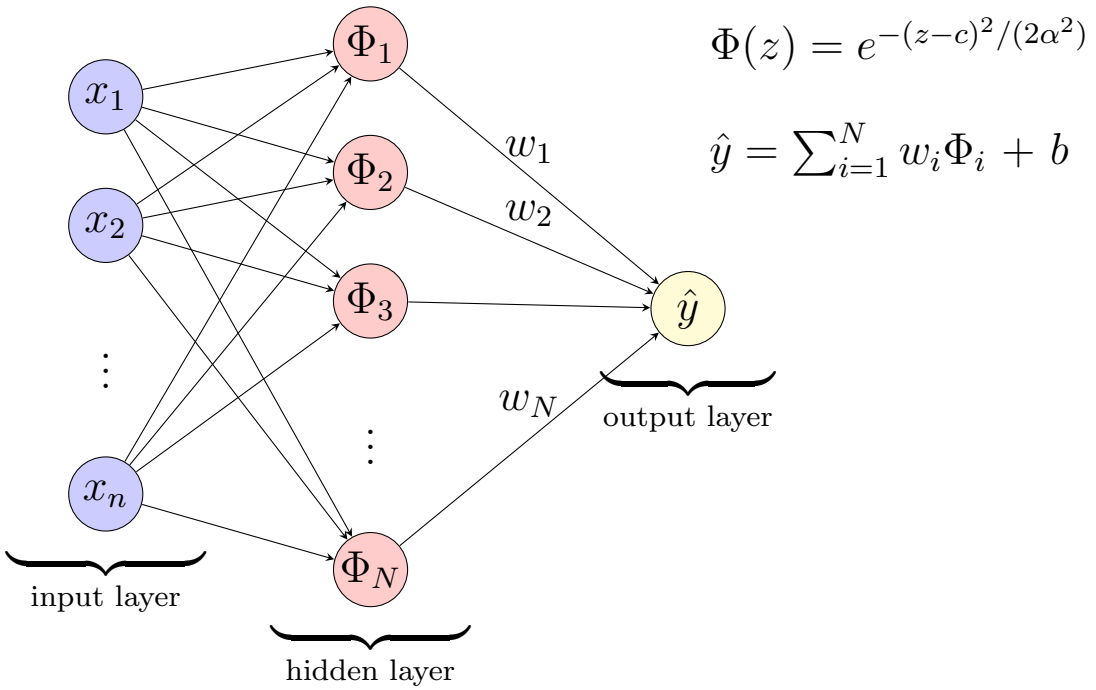


Figure 3.5: Diagram of RBF Network

The input layer simply transmits from the input features to the hidden layers. Therefore, the number of input units is exactly equal to the dimensionality of the data. As in the case of feed-forward networks, no computation is performed in the input layers, but in the input layer, the data gets grouped into clusters using a clustering algorithm. The input nodes are fully connected to the hidden layer nodes and carry the clusters forward to the hidden layer.

The hidden layer contains a variable number of neurons (the ideal number determined by the training process). Each neuron contains the activation function Φ. What makes RBF networks unique to other neural networks is that the activation

function is a radial basis function, so the neuron has Φ centered on a point. The number of dimensions coincides with the number of predictor variables.

When an $\boldsymbol{x}$ vector of input values is fed from the input layer, a hidden neuron calculates the Euclidean distance between the test case and the neuron's center point. It then applies the kernel function using the spread values. The resulting value gets fed into the output layer.

The output layer uses linear classification or regression modeling with respect to the inputs from the hidden layer. The connections from the hidden to the output layer have weights attached to them. The value obtained from each node in the hidden layer is then multiplied by a weight related to the neuron and passed to the output. The computations in the output layer are performed in the same way as in a standard feed-forward network, where the weighted values and bias are added up, and the sum is presented as the network's output, given by $\hat{y}$ in Figure 3.5.

RBF networks have their advantages, namely they have an easy design, only one hidden layer, strong tolerance to input noise, and a straightforward interpretation of the meaning or function of each node in the hidden layer.

3.5 K-means clustering

The K-means clustering algorithm [15, 5, 8] is one of the most widely used data exploration techniques, employing an error-correcting method to adjust its clusters through a simple but effective iterated algorithm. The following K-means clustering algorithm, originally proposed by Lloyd [15], details a simple but fast procedure that can be shown to provide a local optimum to the optimisation problem above.

3.5.1 The K-means algorithm

- Step 1: Choose initial cluster centers $\mathcal{C} := \{\boldsymbol{c}_k, k = 1, 2, \ldots, K\}$ arbitrarily.
- Step 2: For each $k = 1, 2, \ldots, K$, set the cluster C_k to be the points in D that are closer to the center $\boldsymbol{c}_k$ than they are to any other center $\boldsymbol{c}_j$, for all $j \neq k$.
- Step 3: Set new cluster centers $\boldsymbol{c}_k, k = 1, 2, \ldots, K$, such that the center c_k is the center of mass of all the data points in C_k, i.e. $\boldsymbol{c}_k = \frac{1}{|C_k|} \sum_{\boldsymbol{x} \in C_k} \boldsymbol{x}$.

- Step 4: Repeat steps 2 and 3 until convergence, where the cluster centers do not change.

Algorithm 4 K-mean algorithm

Choose initial cluster centers $\mathcal{C} = \{\boldsymbol{c_k}, k = 1, 2, \ldots, K\}$

while not converge **do**

for $k = 1, 2, \ldots, K$ **do**

$C_k = \{\boldsymbol{x} \in D : \|\boldsymbol{x} - \boldsymbol{c}_k\| \leq \|\boldsymbol{x} - \boldsymbol{c}_j\|, \quad \forall j \neq k\}$.

end for

for $k = 1, 2, \ldots, K$ **do**

$\boldsymbol{c}_k = \frac{1}{|C_k|} \sum_{\boldsymbol{x} \in C_k} \boldsymbol{x}$.

end for

end while

In practice, the initial centers chosen for Step 1 are points randomly chosen from our data set D.

For Step 2, if a point is equidistant from two cluster centers, ties should be allocated in a consistent manner. Noting the equality:

$$\frac{1}{|C_k|} \sum_{i,i' \in C_k} \|\boldsymbol{x}_i - \boldsymbol{x}_{i'}\|^2 = 2 \sum_{i \in C_k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_i\|^2, \tag{3.5.1}$$

where $\boldsymbol{\mu}_i$ is the mean vector for the ith cluster, it is clear that the algorithm is guaranteed to decrease the value of the objective function, since by Step 2 of the algorithm we have that the only changes to the clustering indices are those that decrease an observation's distance to a cluster centroid. However, this algorithm is only guaranteed to find a local optimum rather than a global one, depending on the initial assignment of clusters and so in most applications of K-means clustering, the algorithm is run multiple times with different initial configurations, and the solution chosen is given by the realisation that produces the lowest objective value for our optimisation problem.

Chapter 4

Simulation of dynamic pricing on Uber data set

In this chapter, we develop our simulation of dynamic pricing on a dataset of Uber, an example of dynamic pricing in ride-sharing services. In Section 4.1 we will describe the data set and explore the attributes that will be utilised to generate our pricing model. In Section 4.2, we explore the pricing model provided by Schroder [22]. In Section 4.3, we utilise the pricing model to generate our datasets, based on various assumptions, and we provide visualisations of our datasets. Finally, in Sections 4.4 and 4.5, we outline the procedures that will be taken for our LSTM and RBF networks for our simulations of dynamic pricing.

4.1 Description of the dataset

The dataset from [22] was a summary of aggregated statistics of over 28 million ride-hailing services of 137 routes all around the world between the dates of May 31st 2019 to June 25th 2019. Of these 137 routes, they can be further categorised based on the origin location of the route: 63 were from airports, 23 from convention centres, 12 from train stations, and 39 from the city.

Based on the data set, we will use the timescale analysis parameters from Table 4.4 to generate our pricing model, which will be further explored in Section 4.2.

In the subsequent tables, we give a more detailed description of each attribute of the data set.

Tables 4.1 - 4.4 show the different variables and attributes that the dataset contains about the 137 routes. Table 4.1 outlines very general geographical descriptions of the routes.

Heading	Description	Example
Label	Unique label given to route	ATL
City	City where the route is located	Stockholm
Region	Region where the route is located	Europe

Table 4.1: Geographical Attributes

Table 4.2 contains values that are associated with the Origin, the starting location, or the Destination, the end location of a route.

Heading	Description	Example
Pol [Type]	Type of location	Airport
Name	See Note 1	ATL
Latitude	Latitude of location (North is positive)	33.64086
Longitude	Longitude of location (East is positive)	-84.41694

Table 4.2: Location Attributes

Note 1: If there are multiple routes from the same city, then there is a more specific description of the location. Otherwise, the Name is the same as the Label from Table 4.1.

Table 4.3 contains parameters about each particular route over the course of the period when the results were recorded.

Table 4.4 are parameters used for timescale analysis. These values are important in generating Δp which will be further detailed in Section 4.3.

Heading	Description	Data Type
Start [local time]	Start time of recording results	Datetime
Stop [local time]	End time of recording results	Datetime
Rate [Requests per Minute]	No. of requests per minute (on average)	Numeric
Standard	"Standard" service type	Numeric
Max Standard Surge [base price]	Surge for "Standard" service	Numeric
Premium	"Premium" service type	Numeric
Max Premium Surge [base price]	Surge for "Premium" service	Numeric

Table 4.3: Parameters

Heading	Description	Data Type
sigma_base	Standard deviation of p_{base}	Numeric
sigma_surge	Standard deviation of p_{surge}	Numeric
weight_base	Weighting of p_{base}	Numeric
weight_surge	Weighting of p_{surge}	Numeric
weight_zero	Weighting of $\delta(\Delta p)$	Numeric

Table 4.4: Timescale Analysis Parameters

4.2 The pricing model

The price that is calculated will be a function of three variables: demand D, supply S, and time t. Specifically, S represents the number of drivers available, and D is the number of customers requesting a ride-share service at a particular time t.

The pricing model provided in [22] decomposes the price of the service, denoted by p into two parts, a base cost p_{base} and a surge fee p_{surge} such that

$$p_{total} = p_{base} + p_{surge}(D, S) \tag{4.2.1}$$

where p_{surge} is a function of demand (D) and supply (S). The base cost p_{base} also has its own formula of

$$p_{base} = p_0 + p_t \Delta t + p_\ell \Delta \ell \tag{4.2.2}$$

where p_0 are any one-off fees, p_t is the trip fee proportional to the duration Δt and p_ℓ is the trip fee proportional to the distance Δl of the trip. The base cost

will vary depending on the time of day. In this situation, it is assumed that the distance of the trip is constant. However, the duration of the trip will affect the base cost. For example, the duration of the trip may increase during peak hours due to factors such as traffic congestion, which increases the base cost and decreases during off-peak hours.

Additionally, we have another model where the price of a ride-sharing service can be expressed as a function of demand D and supply S which can be represented as a piece-wise linear function.

$$p'(S,D) = \begin{cases} p_{base}, \text{ if } S \geq D \\ p_{base} + p_{surge}^{max} \frac{D-S}{D}, \text{ otherwise.} \end{cases} \tag{4.2.3}$$

For this model, S and D are discretised, such that S is the number of drivers available, and D is the number of customers requesting for a ride-share service at a particular time point.

The next goal is to generate the sets of p_{base} and p_{surge}. To do that, we need to make some assumptions.

From [22] , we are given that

$$\Delta p_n = \frac{\text{total fare}(t_n)}{\text{base cost}(t_n)} - \frac{\text{total fare}(t_{n-1})}{\text{base cost}(t_{n-1})} = \frac{\text{surge cost}(t_n)}{\text{base cost}(t_n)} - \frac{\text{surge cost}(t_{n-1})}{\text{base cost}(t_{n-1})} \tag{4.2.4}$$

We can generate Δp as a sequence of random numbers $\{\Delta p_1, \Delta p_2, \ldots, \Delta p_N\}$, with assumptions on the maximum and minimum values such that $\Delta p^2 < 1 \times 10^{-14}$.

The Δp is a random variable that follows the following probability density function

$$f(t) = w_0\delta(t) + w_{base}\frac{1}{\sqrt{2\pi\sigma_{base}^2}}e^{-\frac{t^2}{2\sigma_{base}^2}} + w_{surge}\frac{1}{\sqrt{2\pi\sigma_{surge}^2}}e^{-\frac{t^2}{2\sigma_{surge}^2}}, \tag{4.2.5}$$

where w_{surge} is the surge contribution, σ_{surge} is the normalised surge strength, and $\delta(t)$ is a Dirac delta distribution.

This means that the probability density function 4.2.5 satisfies

$$\mathbb{P}(X \leq \Delta p \leq Y) = \int_X^Y f(t)dt$$

Next, assume that we have $\{t_0, \ldots, t_N\}$. We also assume that total fare(t_0) and base cost(t_0) are given. This would also mean that surge cost(t_0) can be calculated as

$$\text{total fare}(t_i) = \text{base cost}(t_i) + \text{surge cost}(t_i)$$

for $0 \leq i \leq N$, where N is the number of minutes. Additionally, that would mean

$$\frac{\text{total fare}(t_1)}{\text{base cost}(t_1)} = \frac{\text{total fare}(t_0)}{\text{base cost}(t_0)} + \Delta p_1$$

and

$$\frac{\text{surge cost}(t_1)}{\text{base cost}(t_1)} = \frac{\text{surge cost}(t_0)}{\text{base cost}(t_0)} + \Delta p_1$$

Suppose we have a sequence $\{p_1, \ldots, p_N\}$, we then use the formula (4.2.3) to generate the data set

$$\mathcal{D} = \{p(t_i, s_j, d_k)\} \tag{4.2.6}$$

where $t_i = \{t_0, t_1, t_2, \ldots t_N\}$, $s_j = \{0, 1, 2, \ldots, S_{\max}\}$ and $d_k = \{0, 1, 2, \ldots, D_{\max}\}$. where $S_{\max}$ and $D_{\max}$ are some positive constants to be specified later.

Then the data $\mathcal{D}$ can be used to train RBF-net and LSTM.

Once we have this data, there are various combinations of experiments that we can do, based on the variables

Instead of a simple piecewise model to determine the price of the ride-share service, we will use three different modelling techniques to predict the fare, given values of S and D: a Radial Basis Function (RBF) neural network model, a Long Short Term Memory (LSTM) neural network model and linear regression, which is a more classical method used in prediction modelling.

4.3 Creating the model

Because we did not have access to the actual dataset, what was done instead was to reverse the process and generate a realistic dataset based on the aggregated statistics that we were able to source from Schroder's paper [22]. Other options that we had to get access to a dataset included contacting Schroder for the original dataset, or creating an Application Program Interface (API), but generating our own dataset seemed most feasible.

Assumptions that were made about p_{base} include:

- p_0, p_ℓ, $\Delta\ell$ and p_t are constant, and;
- Δt is the only value that varies.

To create the overall general shape of the graph for p_{base}, two skewed graphs of normal distribution were added, based on assumptions of peak hour periods. A bit of randomness was also included to make our dataset more realistic. An example of a graph of p_{base} is shown in Figure 4.1.

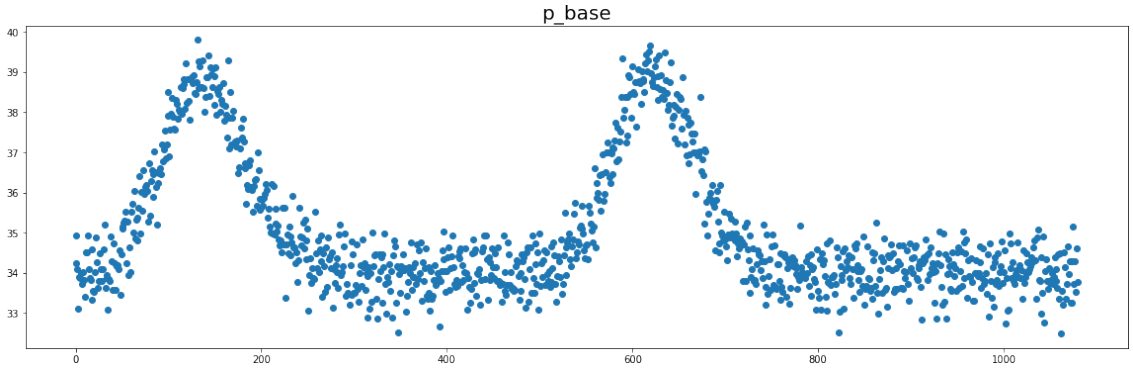


Figure 4.1: p_{base}, or base cost, as defined in Equation (4.2.2)

Once our p_{base} is generated, we can now use this dataset to assist with generating the surge cost p_{surge}.

One of the routes from the aggregated dataset was arbitrarily picked, and the variables σ_{base}, σ_{surge}, w_{base}, w_{surge} and w_{zero} were used to generate Δp by 4.2.5.

Adding the base and surge costs together, we get our p_{total} dataset which we will call $\mathcal{D}_1$. For $\mathcal{D}_1$, we have fixed values of S and D, and it is just a time series. It is defined as:

$$\mathcal{D}_1 := \{p(t_0), \ldots, p(t_N)\} = \{p(t_i, S, D) : i = 0, \ldots, N\} \tag{4.3.1}$$

and shown in Figure 4.5.

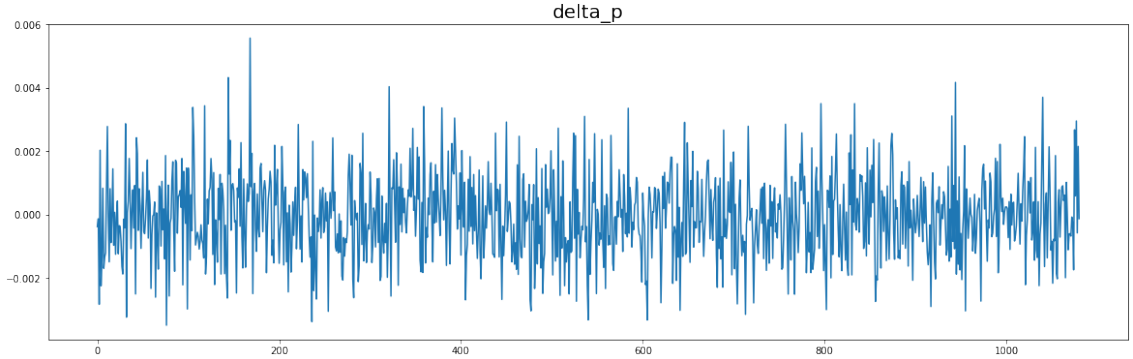


Figure 4.2: Δ_p, as defined in Equation (4.2.4)

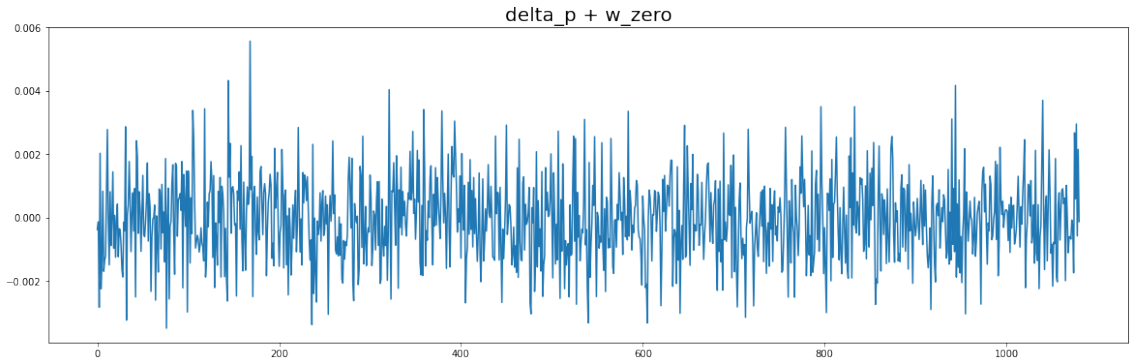


Figure 4.3: $\Delta_p + w_0$, as defined in Equation (4.2.5)

Now the aim is to determine the price based on (5.2.1), in the form of 2D surface plots. For all of these plots, the y axis represented $p'(t, S, D)$ where one of these variables are fixed.

In our first plot, we wanted to fix the price at a particular time point in the graph t' and see the changes in price when supply and demand were variable. This dataset

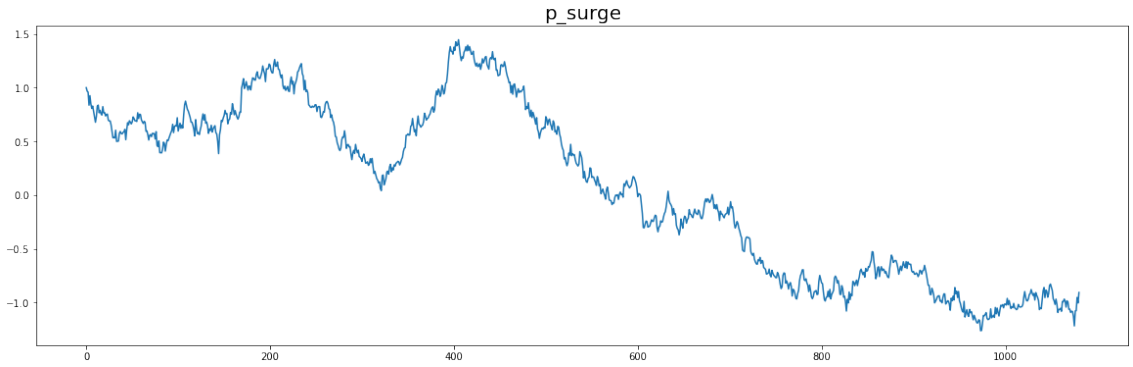


Figure 4.4: p_{surge}, or surge cost

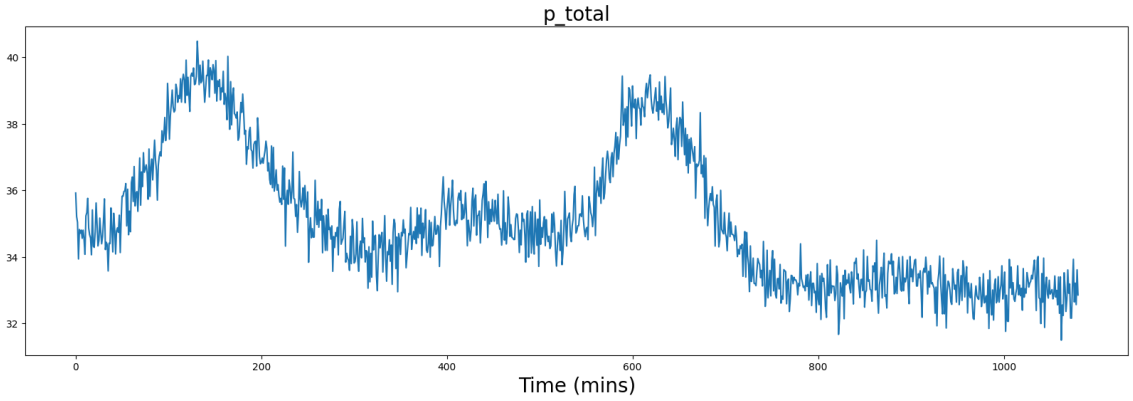


Figure 4.5: p_{total} defined in (4.2.1), also defined as $\mathcal{D}_1$ from Equation (4.3.1)

will be one of the datasets we will use for our 2-dimensional simulations. We will call it $\mathcal{D}_t$ and it is defined by

$$\mathcal{D}_t := \{p(t, S_j, D_k) : j = 0, \ldots, S_{\max}, k = 0, \ldots, D_{\max}\}. \tag{4.3.2}$$

and is visualised in Figure 4.6.

After that, we wanted to see what would happen if either demand or supply were fixed. So we fixed the demand for an arbitrary value D' between 0 and the maximum demand D_{max} (which in our case was assumed to be 150). This gives us a surface where supply and time are variable and we get the surface plot shown in Figure 4.7.

Finally, fixing supply for a particular value S', varying demand and time, we achieved the surface plot shown in Figure 4.8.

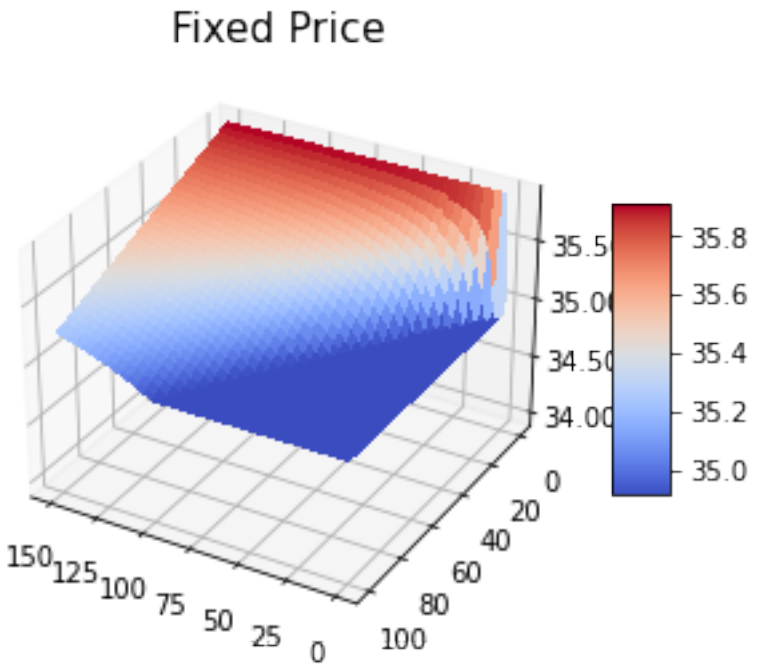


Figure 4.6: Surface plot $p(t', S, D)$, also defined as $\mathcal{D}_t$, from Equation (4.3.2)

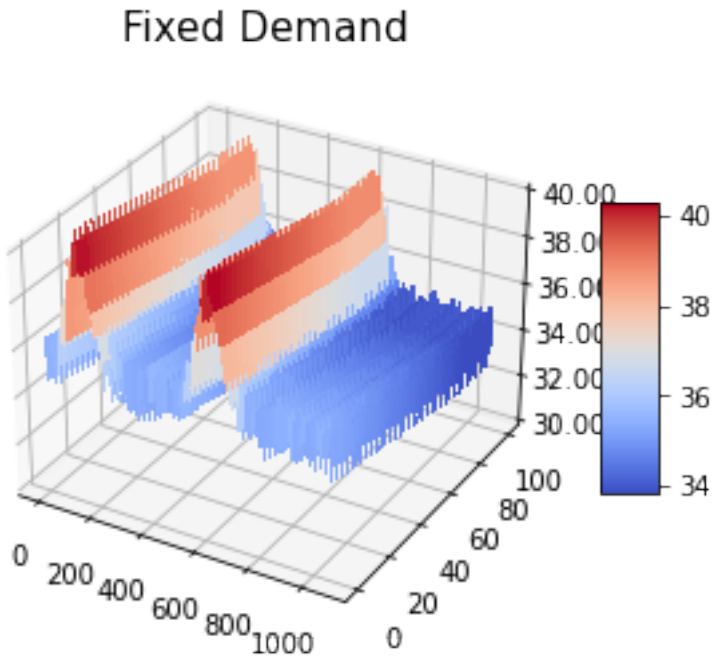


Figure 4.7: Surface plot $p(t, S, D')$

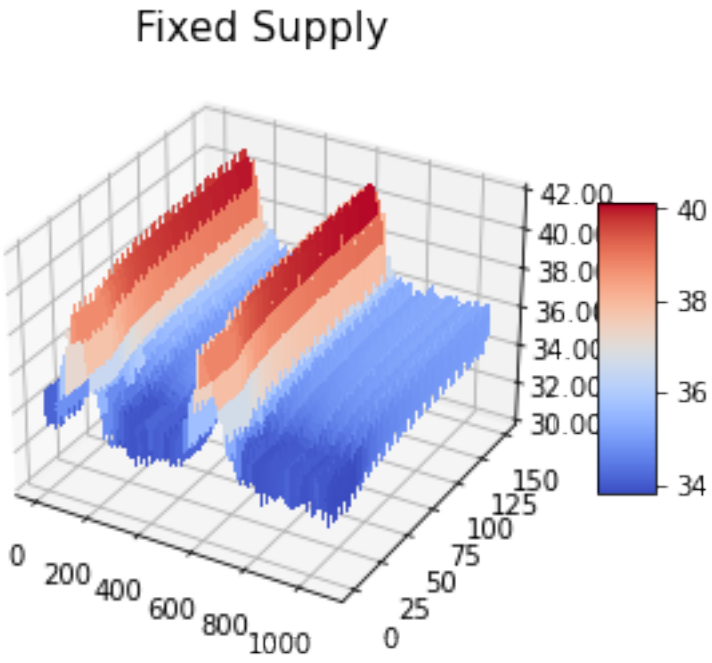


Figure 4.8: Surface plot $p(t, S', D)$

4.4 Setting up LSTM

Here, we will describe how to set up the LSTM model (described in Section 3.3) for our simulation. For the LSTM experiments, the procedure for our experiments is as follows:

1. We select some starting values for batch size, lookback, and epochs. These parameters affect how many values at a time are chosen from the training set for training in our LSTM model, the number of values that the model looks back for each t_i predicting t_{i+1} in our prediction and test sets, and the number of iterations for each repetition such that the loss function descends close to 0.

2. Arbitrarily select values in the dataset to form our test set.

3. Sort the values of our test set based on their index in the original dataset.

4. After that, we choose the value M as the number of repetitions of our experiment.

5. Similarly to the test set, we choose a portion of the remaining values to become our training set, which is then sorted based on their indices in the original dataset.

6. Update the weight

$$\boldsymbol{w}$$

and bias b after each epoch in the dataset. This will train our model and from there we can determine our predicted value $\hat{t}_{i+1}$.

7. Once we have our predicted value, we are then able to compute our accuracy metrics: RMSE and relative RMSE.

Now we repeat the procedures and adjust parameters one at a time to determine optimal parameters for the LSTM model.

There was also the potential of doing a straight split in the dataset (e.g. first 80% of the data becomes the training set, and the remaining 20% becomes the test set).

However, we chose to avoid this method of splitting the data into training and test sets as we could run into the risk of selection bias.

4.5 Setting up RBF

We also wanted to compare the performance of LSTM with another neural network, which is the RBF network.

Our procedure for the experiment of our RBF network is as follows:

1. Choose an arbitrary number of bases/kernels k, depending on the shape of the graph and number of epochs in the model. This will determine the number of kernels for K-means clustering.
2. Arbitrarily select values in the dataset to form our test set.
3. Sort the values of our test set based on their index in the original dataset.
4. After that, we choose the value M as the number of repetitions of our experiment.
5. Similarly to the test set, we choose a portion of the remaining values to become our training set, which are then sorted based on the index in the original dataset.
6. Update the weight $\boldsymbol{w}$ and bias b after each epoch in the dataset. This will train the RBF network.
7. Predict the values on our testing set and compute accuracy metrics.

Similarly, we repeat these experiments while adjusting for parameters k and epochs to find the optimal parameters.

Once we get the accuracy metrics, we can then compare the performance of both LSTM and RBF Network in their predictive abilities.

Chapter 5

Results

5.1 Introduction

In this chapter, we will carry out experiments using 1-d and 2-d datasets generated in Sections 4.2 and 4.3 and compare and critically analyse the results from RBF nets vs. LSTM. In particular, we will:

- Outline the general setting of the experiments in 5.2.
- Report the numerical results using RBF net experiments in 5.3
- Report the results from the LSTM experiments in 5.4 and compare them with results from RBF-net.
- Detail the results from the RBF-net 2-dimensional experiments and compare the two different clustering methods first idealised in 5.2.2 and later described in 5.6.
- Compare using our classical method of linear regression with neural networks for both 1-dimensional and 2-dimensional experiments.

5.2 Setting up the experiments

5.2.1 The 1-dimensional data set

For fixed values of supply S and demand D, when time varies, the price (see (4.2.6)) is just a time series

$$\{p(t_0), \ldots, p(t_N)\} = \{p(t_i, S, D) : i = 1, \ldots, N\}.$$

We first choose a random set of indices $\mathcal{U}$ from $\{0, \ldots, N\}$ to form a fixed test set with $|\mathcal{U}| = (N+1)/10$. This results in $(N+1) - |\mathcal{U}|$ indices remaining. We then choose a random set of indices $\mathcal{T}$ from the remaining $N - |\mathcal{U}|$ indices to form our training set. In our experiments in this section $N = 1080$, $|\mathcal{U}| = 108$. The training set is

$$\{p(t_k) : k \in \mathcal{T}\}.$$

Let the cardinality of the training set $\mathcal{T}$ be $|\mathcal{T}|$. We will vary the ratio

$$|\mathcal{T}|/((N+1) - |\mathcal{U}|)$$

in our experiments, taking values of $0.9; 0.8; 0.7$. Correspondingly, $|\mathcal{T}| = 875; 778; 583$ respectively.

The root mean square error (RMSE) of each experiment is defined as follows:

$$RMSE = \sqrt{\frac{\sum_{i \in \mathcal{U}} (\hat{p}(t_i) - p(t_i))^2}{|\mathcal{U}|}}.$$

Relative RMSE is defined by

$$\text{rel RMSE} = \sqrt{\frac{\sum_{i \in \mathcal{U}} (\hat{p}(t_i) - p(t_i))^2}{\sum_{i \in \mathcal{U}} |p(t_i)|^2}}$$

We repeat this experiment M times randomly selecting indices every time to reduce bias.

We experimented with different values of M, specifically $M = 20$, and $M = 50$ to get the values of RMSE each time. From there, we determine the average root mean square errors

$$RMSE = \frac{1}{M} \sum_{k=1}^{M} \text{RMSE}_k.$$

5.2.2 *The 2-dimensional data set*

When time is fixed, we can vary supply and demand quantities to get a 2-dimensional data set, of the form

$$\{p(t, S_j, D_k) : j = 0, \ldots, S_{\max}, k = 0, \ldots, D_{\max}\},$$

which we labelled $\mathcal{D}_t$, from Section 4.3. For our experiments, we set $S_{\max} = 100$ and $D_{\max} = 150$. Similarly to our method for the 1-dimensional data set, we first choose a random set of indices $\mathcal{U}$ from $\{0, \ldots, S_{\max}\} \times \{0, \ldots, D_{\max}\}$ to form a fixed test set with $|\mathcal{U}| = L/10$, where $L = (S_{\max}+1)(D_{\max}+1)) = (100+1) \times (150+1) = 15,251$. This results in $N - |\mathcal{U}|$ indices remaining. We then choose a random set of indices $\mathcal{T}$ from the remaining $N - |\mathcal{U}|$ indices to form our training set. So $|\mathcal{U}| = 1,525$. Consequently, the training set is

$$\{p(S_j, D_k) : (j, k) \in \mathcal{T}\}.$$

Let the cardinality of the training set $\mathcal{T}$ be $|\mathcal{T}|$. We will vary the ratio

$$|\mathcal{T}|/(L - |\mathcal{U}|)$$

in our experiments, taking values of $0.9; 0.8; 0.7$. For this dataset,

$$|\mathcal{T}| = 12353; 10981; 9608$$

respectively.

For our 2-D data set, we experimented with using two different clustering methods for our RBF-net:

1. K-means clustering, using the K-means algorithm, see Section 3.5, Algorithm 4.

2. Uniform Grid clustering. In this approach, we define the centres (x_i, y_j) of each cluster as a point on a uniform grid of the following form

$$(x_i, y_j) = ((i + 1/)2h, (j + 1/2)h), i = 1, \ldots, I; j = 1, \ldots, J,$$

where $h > 0$ is a given step-size and I and J are some given positive integers.

These methods will be further explained in detail in Section 5.6.

The root mean square error (RMSE) of each experiment is defined as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_{(j,k)\in\mathcal{U}}(\hat{p}(D_j, S_k) - p(D_j, S_k))^2}{|\mathcal{U}|}}.$$

Relative RMSE is defined by

$$\text{rel RMSE} = \sqrt{\frac{\sum_{(j,k)\in\mathcal{U}}(\hat{p}((D_j, S_k)) - p(D_j, S_k))^2}{\sum_{i\in\mathcal{U}}|p(D_j, S_k)|^2}}.$$

We repeat this experiment M times randomly selecting indices every time to reduce bias.

We experimented with different values $M = 5$ and $M = 10$ to get the values of RMSE each time. The reason why we only did so few repetitions was because our code to calculate RMSE was time-consuming and choosing these specific values of M ensured that we would obtain accurate results in a reasonable amount of time. From there, we then determine the average root mean square errors

$$RMSE = \frac{1}{M}\sum_{k=1}^{M}\text{RMSE}_k.$$

5.3 Radial Basis Function (RBF) Network

For our 1-dimensional simulations, we follow the procedures outlined in Section 4.5 and Section 5.2.1 for our experiment, train the RBF-Net on the training set, and predict the values $\hat{p}(t_i)$ for $i \in \mathcal{U}$.

Figure 5.1 gives us a visualisation of the RBF-Net with $k = 20$ clusters using k-means clustering.

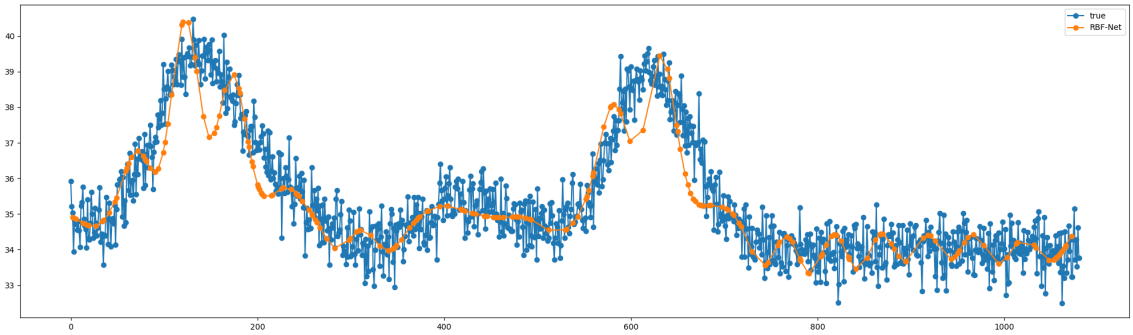


Figure 5.1: RBF-Net using 20 clusters example

In Figure 5.1, the blue are data points, and the orange curve is the prediction produced by RBF-net with 20 clusters using our price data set of 1081 points. The horizontal axis is the time (in minutes), where 0 corresponds to 6 a.m. local time and ranges to 12 a.m. local time, while the vertical axis is the price of the service in USD.

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1) - \lvert\mathcal{U}\rvert)$		
k (No. of clusters)	70%	80%	90%
10	1.089	1.081	1.069
20	0.900	0.865	0.853
50	0.911	0.860	0.855

Table 5.1: RMSE with $M = 20$ repetitions

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1) - \lvert\mathcal{U}\rvert)$		
k (No. of clusters)	70%	80%	90%
10	0.03045	0.03023	0.02987
20	0.02493	0.02492	0.02417
50	0.02547	0.02405	0.02390

Table 5.2: Relative RMSE with $M = 20$ repetitions

Notice for smaller values of M, the RMSE does increase as the training ratio increases, but we notice that for higher values, (i.e. $M \geq 20$), the RMSE does decrease, as expected.

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
k (No. of clusters)	70%	80%	90%
10	1.153	1.064	1.043
20	0.873	0.870	0.869
50	0.891	0.886	0.857

Table 5.3: RMSE with $M = 50$ repetitions

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
k (No. of clusters)	70%	80%	90%
10	0.03223	0.02974	0.02915
20	0.02449	0.02425	0.02400
50	0.02491	0.02476	0.02397

Table 5.4: Relative RMSE with $M = 50$ repetitions

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.882	0.872	0.850
500	0.873	0.857	0.860
1000	0.892	0.867	0.856
2000	0.874	0.864	0.858

Table 5.5: RMSE with $M = 20$ repetitions and $k = 20$ clusters)

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.02467	0.02437	0.02377
500	0.02440	0.02395	0.02403
1000	0.02496	0.02425	0.02393
2000	0.02443	0.02415	0.02398

Table 5.6: Relative RMSE with $M = 20$ repetitions and $k = 20$ clusters

We also carried out experiments with different numbers of clusters in the configuration of the RBF net. We experimented with $k = 10, 20$ and 50 clusters in our training model, we obtained these results provided in Tables 5.1 - 5.4. Additionally, we wanted to see

As can be seen in the Tables, we see that when the number of cluster increases, then errors decreases

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.863	0.882	0.860
500	0.870	0.869	0.871
1000	0.881	0.870	0.862
2000	0.894	0.858	0.856

Table 5.7: RMSE with $M = 50$ repetitions and $k = 20$ clusters)

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1)-\lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.02414	0.02468	0.02405
500	0.02432	0.02430	0.02435
1000	0.02464	0.02432	0.02411
2000	0.02500	0.02398	0.02393

Table 5.8: Relative RMSE with $M = 50$ repetitions and $k = 20$ clusters

We repeated the same experiment but we changed the number of clusters from 20 to 10 to observe the effect of our metrics as a result of varying the number of clusters. We attained the following results:

5.4 Long Short-Term Memory (LSTM) Network

For the one-dimensional case using Long Short-Term Memory neural network (LSTM), we repeated the same method to the RBF Network and determined the training and test sets such that the ratio of $|\mathcal{T}|/|\mathcal{U}|$, varies between values 0.9;0.8;0.7. These values are fixed for all repetitions M. See Section 4.4 for the full methodology.

To reduce the risk of bias, the values of the dataset were randomly selected. The indices of the training and test sets are then ordered by their index to ensure that the values maintained the shape of the original dataset.

In all cases, we used the Adam optimiser, see Section 3.2.5.

For Tables 5.9 - 5.12, the lookback was kept at 1 for simplicity, as the first objective was to determine what value for batch size optimises our accuracy metrics.

	Training Ratio $= \|\mathcal{T}\|/((N+1) - \|\mathcal{U}\|)$		
batch size	70%	80%	90%
1	0.836	0.791	0.779
5	0.798	0.787	0.784
10	0.811	0.796	0.791
20	0.845	0.802	0.789
50	0.964	0.794	0.790

Table 5.9: RMSE for M (No. of repetitions) = 20 and lookback = 1.

	Training Ratio $= \|\mathcal{T}\|/((N+1) - \|\mathcal{U}\|)$		
batch size	70%	80%	90%
1	0.02336	0.02212	0.02177
5	0.02231	0.02200	0.02190
10	0.02269	0.02225	0.02210
20	0.02362	0.02241	0.02205
50	0.02712	0.02210	0.02202

Table 5.10: Rel RMSE for M (No. of repetitions) = 20 and lookback = 1.

	Training Ratio $= \|\mathcal{T}\|/((N+1) - \|\mathcal{U}\|)$		
batch size	70%	80%	90%
1	0.805	0.772	0.776
5	0.797	0.776	0.769
10	0.803	0.781	0.767
20	0.818	0.789	0.776
50	0.858	0.792	0.786

Table 5.11: RMSE for M (No. of repetitions) = 50 and lookback = 1.

By making M and lookback fixed, what we observe from tables 5.9 - 5.12 is that improvement in error is consistent when we increase the training ratio. Additionally, and more importantly, the error is smaller when batch size is minimised. More specifically, RMSE was minimised with a batch size of 5.

Now we want to observe the changes to our metrics when we adjust the lookback value.

The next set of tables (Tables 5.13 - 5.16) aims to analyse the effect of varying lookback while keeping batch size fixed. In our experiments, we will use lookback

	Training Ratio = $\|\mathcal{T}\|/((N+1)-\|\mathcal{U}\|)$		
batch size	70%	80%	90%
1	0.02252	0.02158	0.02170
5	0.02229	0.02170	0.02149
10	0.02245	0.02183	0.02144
20	0.02288	0.02204	0.02169
50	0.02396	0.02213	0.02184

Table 5.12: Relative RMSE for M (No. of repetitions) = 50 and lookback = 1.

values of 1, 5, 10, and 20, to see whether we can retrieve any insights from the data. As a result of tables 5.9 - 5.12, we kept the batch size at 5.

	Training Ratio = $\|\mathcal{T}\|/((N+1)-\|\mathcal{U}\|)$		
lookback	70%	80%	90%
1	0.813	0.782	0.784
5	0.792	0.792	0.792
10	0.837	0.896	0.917
20	0.941	0.983	1.072

Table 5.13: RMSE for M (No. of repetitions) = 20 and batch size = 5.

	Training Ratio = $\|\mathcal{T}\|/((N+1)-\|\mathcal{U}\|)$		
lookback	70%	80%	90%
1	0.02272	0.02189	0.02193
5	0.02209	0.02210	0.02212
10	0.02335	0.02500	0.02558
20	0.02646	0.02764	0.03011

Table 5.14: Relative RMSE for M (No. of repetitions) = 20 and batch size = 5.

	Training Ratio = $\|\mathcal{T}\|/((N+1)-\|\mathcal{U}\|)$		
lookback	70%	80%	90%
1	0.795	0.774	0.767
5	0.788	0.788	0.787
10	0.828	0.856	0.855
20	1.003	1.027	1.007

Table 5.15: RMSE for M (No. of repetitions) = 50 and batch size = 5.

	Training Ratio = $\|\mathcal{T}\|/((N+1)-\|\mathcal{U}\|)$		
lookback	70%	80%	90%
1	0.02222	0.02164	0.02143
5	0.02200	0.02201	0.02198
10	0.02311	0.02388	0.02387
20	0.02818	0.02887	0.02829

Table 5.16: Relative RMSE for M (No. of repetitions) = 50 and batch size = 5.

The first insight that we derive from Tables 5.13 to 5.16 is that there isn't a linear relationship between the lookback value and the error values, as the errors for lookback equal to 1 performed much worse than when lookback was equal to 5. This would make sense, as LSTM models have an architecture such that they are able to look back to previous values to predict future values without any problems. Moreover, a lookback of 1 would render the model ineffective, as it would produce similar prediction results to other recurrent neural network models.

We can conclude from our empirical results that when varying lookback, the LSTM model performed best when the value of lookback was equal to 5.

From these preliminary experiments and looking at our accuracy metrics from tables 5.9 - 5.16, we can conclude that a smaller batch size of 5 and a lookback value of 5 optimises the model.

Now, we want to look at what happens when we adjust the number of epochs in our model. For this experiment, we will use a batch size of 5, a lookback value of 5, and set M (No. of repetitions) to 50, as in previous models, a higher number of repetitions did tend to reduce the error.

Tables 5.17 & 5.18 show the results from these experiments.

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1) - \lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.7913	0.7914	0.8027
500	0.7882	0.7960	0.8119
1000	0.7863	0.7880	0.7948
2000	0.7971	0.7961	0.7932

Table 5.17: RMSE with $M = 50$, batch size $= 5$ and lookback $= 5$

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1) - \lvert\mathcal{U}\rvert)$		
E (No. of epochs)	70%	80%	90%
200	0.022088	0.022090	0.022407
500	0.022000	0.022220	0.022663
1000	0.021947	0.021995	0.022184
2000	0.022250	0.022221	0.022140

Table 5.18: Relative RMSE with $M = 50$, batch size $= 5$ and lookback $= 5$

It is noted that if we change the number of epochs, we do observe marginal improvement in the error values, shown in Tables 5.17 & 5.18.

Additionally, we can clearly see that our model performs best when we set the number of epochs E to 1000. Something interesting to note as well is that the model seemed to perform worse as the training ratio increased. But overall, there isn't a cause for concern to adjust the training ratio as the difference in errors is marginal and is significant to 1×10^{-4}.

Therefore, from our experiments, the optimal configuration parameters that minimises RMSE best for our LSTM model are batch size $= 5$, lookback $= 5$ and epochs $= 1000$.

5.5 Linear Regression (1D)

We also wanted to compare using neural networks with linear regression, a more classical method that is used for predicting future values. We used the same methodology outlined in Section 5.2.1. The results are as shown in Tables 5.19 and 5.20.

	Training ratio $= \lvert\mathcal{T}\rvert/(N+1) - \lvert\mathcal{U}\rvert)$		
M (No. of repetitions)	70%	80%	90%
20	1.5421	1.5429	1.5426
50	1.5435	1.5430	1.5427

Table 5.19: RMSE (Linear Regression)

	Training ratio $= \lvert\mathcal{T}\rvert/((N+1) - \lvert\mathcal{U}\rvert)$		
M (No. of repetitions)	70%	80%	90%
20	0.043114	0.043135	0.043127
50	0.043152	0.043139	0.043129

Table 5.20: Relative RMSE (Linear Regression)

Tables 5.19 and 5.20 show that using linear regression is not suitable for modelling the data we have generated. This is further supported by Figure 5.2, which clearly shows how poorly linear regression performs on our model.

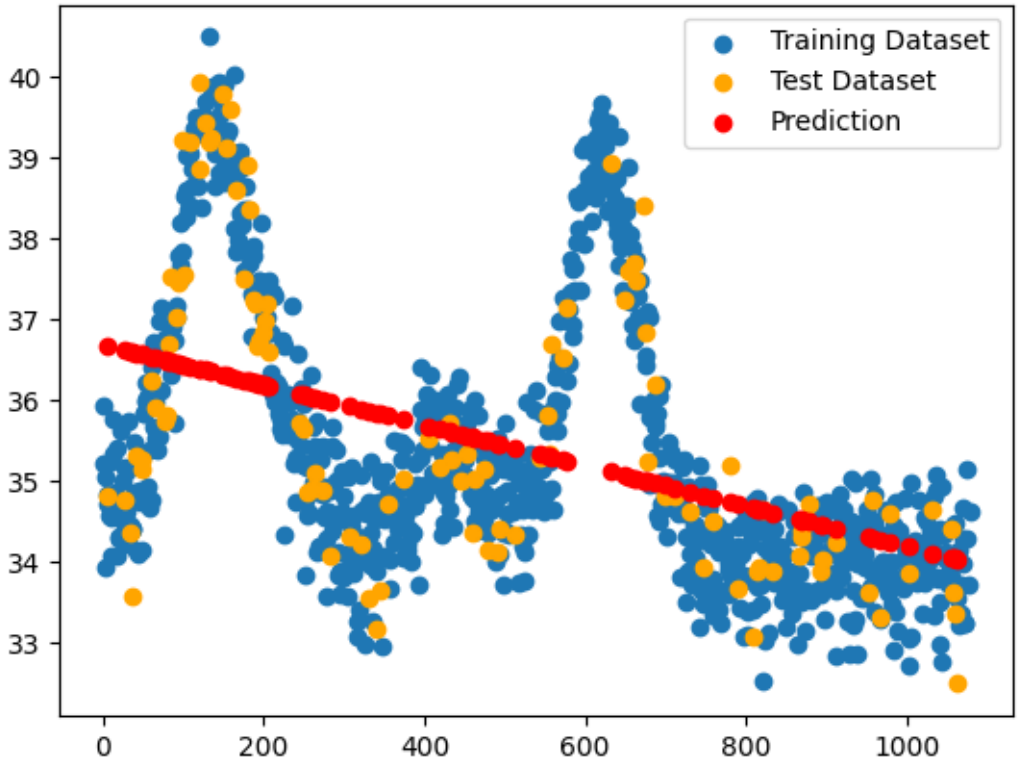


Figure 5.2: Linear Regression (Training Ratio = 90%)

Looking at all of our results for the 1-dimensional experiments, it is clear that using neural networks provided a much more accurate model in predicting the price.

5.6 Radial Basis Function Network (2D)

For our first set of experiments for our 2-dimensional data, we adjusted the K-means algorithm to accommodate for multi-dimensional cases (i.e. $\boldsymbol{x} = (x_1, x_2)$). We followed a similar method to our RBF 1-dimensional case to determine the values that would become our test and training sets. We then trained the model and tested it, compared our prediction to our testing set to derive the RMSE and relative RMSE, which are given in 5.2.2. The results from using K-means clustering are shown in Tables 5.21 - 5.24.

The next set of experiments was to test out how the error would change if we were to choose the positions of the cluster centres manually, almost like a uniform grid. These cluster centres will be positioned such that they are evenly spaced apart on both axes of our 2D plane. We also take into account positioning the initial centres such that they are not on the edge of the plane. However, it is possible for cluster centres to be on the edges at $S_{\max}$ and $D_{\max}$.

For all of our experiments, $S_{\max} = 100$ and $D_{\max} = 150$.

In the next experiments, we choose our cluster centers to be

$$\begin{aligned}(x_i, y_j) &= (ih + h/2, jh + h/2) \\ &\text{with } i = 0, 1, 2, 3, S_{\max}/h; j = 0, 1, 2, 3 \ldots D_{\max}/h.\end{aligned} \tag{5.6.1}$$

We experimented for $h = 10$ and $h = 20$. For the $h = 10$, this would give us 150 evenly spaced cluster centres (15 across each row, 10 in each column). In our experiments for $h = 20$, we calculated that we would get 40 clusters. However, in this situation, the cluster centres are not quite spaced apart evenly as some of the cluster centres end up on the edges.

We also wanted to test where we can adjust the factor value such that the h factor value would be different across the two axes of our 2D plane. The equation could be adjusted such that the h factor value is different for i and j, and we call them h_i and h_j respectively.

$$(x_i, y_j) = (ih_i + h_i/2, jh_j + h_j/2) \text{ with}$$
$$i = 0, 1, 2, 3, S_{\max}/h_i; j = 0, 1, 2, 3 \ldots D_{\max}/h_j. \tag{5.6.2}$$

And so simply, Equation (5.6) is a special case of Equation 5.6.2 where $h_i = h_j$.

Adjusting the factor value across both axes of our plane allows us to adjust the parameters and make results more comparable under the same number of clusters with the k-means clustering method. The first experiment we will perform will be such that $h_i = 10, h_j = 15$, which through 5.6.2 would give $k = 100$ clusters, and $h_i = 20, h_j = 15$, which similarly would be equivalent to $k = 50$ clusters. The reason why these values were selected such that it would be feasible to compare the two different clustering methods and see how they would both perform given the same data.

The results of all these experiments using the uniform grid clustering method are provided in Tables 5.21 to 5.24. We also included visualisations of the prediction surfaces and the absolute error which are given in Figures 5.12 to 5.19.

	Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
k (No. of clusters)	50%	60%	70%	80%	90%
20	0.1405	0.1431	0.1748	0.1903	0.1925
50	0.1235	0.1088	0.1194	0.1203	0.1187
100	0.1905	0.1649	0.1929	0.1904	0.2217
150	0.3198	0.3119	0.3143	0.2692	0.2641

Table 5.21: RMSE with $M = 5$ repetitions, K-Means Clustering

	Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
k (No. of clusters)	50%	60%	70%	80%	90%
20	0.00398	0.00406	0.00496	0.00540	0.00546
50	0.00350	0.00308	0.00338	0.00341	0.00336
100	0.00540	0.00467	0.00547	0.00540	0.00628
150	0.00906	0.00884	0.00891	0.00763	0.00749

Table 5.22: Relative RMSE with $M = 5$ repetitions, K-Means Clustering

It is observed that the error is the best when the training ratio is actually around $60 - 70\%$. While there is no definite reason for why this contradicts our hypotheses

	Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
k (No. of clusters)	50%	60%	70%	80%	90%
20	0.1400	0.1493	0.1650	0.1878	0.1825
50	0.1068	0.1057	0.1087	0.1109	0.1145
100	0.2052	0.1856	0.1730	0.1926	0.1853
150	0.3265	0.3151	0.2784	0.2816	0.2618

Table 5.23: RMSE with $M = 10$ repetitions, K-Means Clustering

	Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
k (No. of clusters)	50%	60%	70%	80%	90%
20	0.00397	0.00423	0.00468	0.00532	0.00517
50	0.00303	0.00300	0.00308	0.00315	0.00324
100	0.00582	0.00526	0.00490	0.00546	0.00525
150	0.00926	0.00893	0.00789	0.00798	0.00742

Table 5.24: Relative RMSE with $M = 10$ repetitions, K-Means Clustering

of the error reducing as the training ratio increases, potential reasons could include overfitting the data into the model, but we are not certain and would need some more investigation.

			Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
h_i	h_j	Equivalent no. of clusters k	50%	60%	70%	80%	90%
10	10	150	0.1214	0.1096	0.1125	0.1039	0.0974
20	20	40	0.1341	0.1254	0.1282	0.1228	0.1096
10	15	100	0.0902	0.0898	0.0788	0.0823	0.0714
20	15	50	0.1300	0.1236	0.1310	0.1126	0.1051

Table 5.25: RMSE with $M = 5$ repetitions, Uniform Grid Clustering

Figures 5.4 to 5.7 show the prediction of our planes for different values of k, and Figures 5.8 to 5.11 are the absolute error derived from K-means clustering. Across all four figures showing the absolute error, it is clear that the position where $S = D = 0$ is where the error is at its largest. Additionally, we also observe that the region with the largest errors across all of these absolute error planes seems to follow a line where $S = D$, i.e. where supply and demand are equal. This corresponds with the non-differentiable region in our dataset $\mathcal{D}_t$, which would make sense as it is difficult

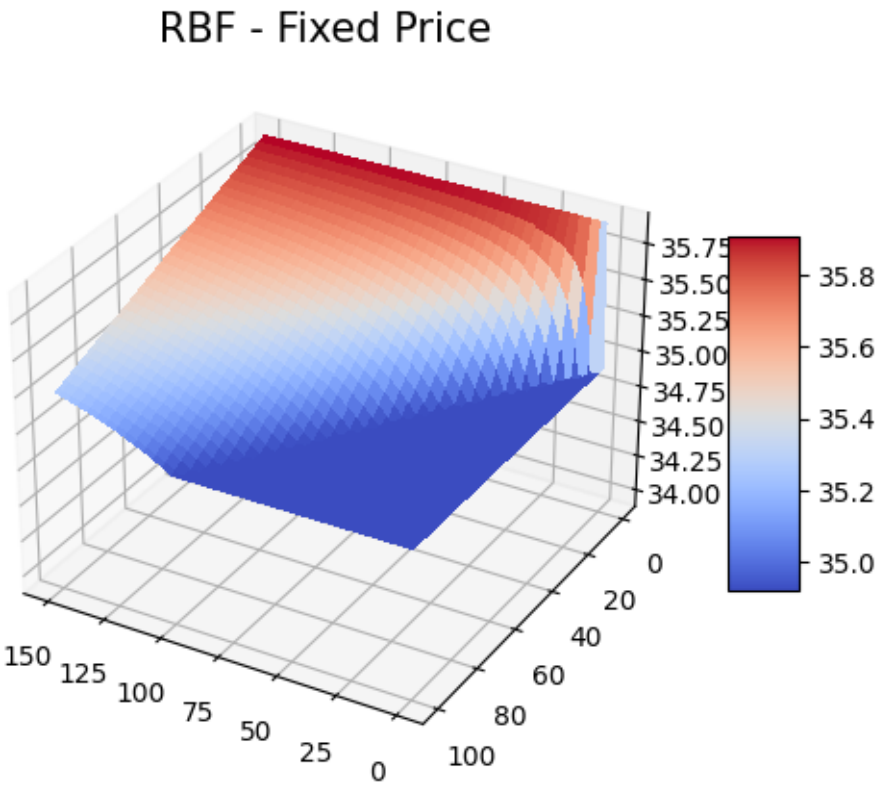


Figure 5.3: Original 2D Dataset

			Training ratio = $\lvert\mathcal{T}\rvert/(L-\lvert\mathcal{U}\rvert)$				
h_i	h_j	Equivalent no. of clusters k	50%	60%	70%	80%	90%
10	10	150	0.00344	0.00311	0.00319	0.00295	0.00276
20	20	40	0.00380	0.00356	0.00363	0.00348	0.00311
10	15	100	0.00256	0.00255	0.00223	0.00233	0.00203
20	15	50	0.00369	0.00350	0.00371	0.00319	0.00298

Table 5.26: Relative RMSE with $M = 5$ repetitions, Uniform Grid Clustering

for a Gaussian function to correctly adjust and approximate a piece-wise function, especially at points or regions where a curve or surface is non-differentiable.

Figure 5.9 also supports our argument that our RBF network performed best at $k = 50$.

For Figures 5.12 to 5.19, we notice that utilising uniform grid clustering also has the largest errors at the corners of the surface (where $S = D = 0$). Additionally, when comparing uniform grid clustering to K-means clustering, we observe that

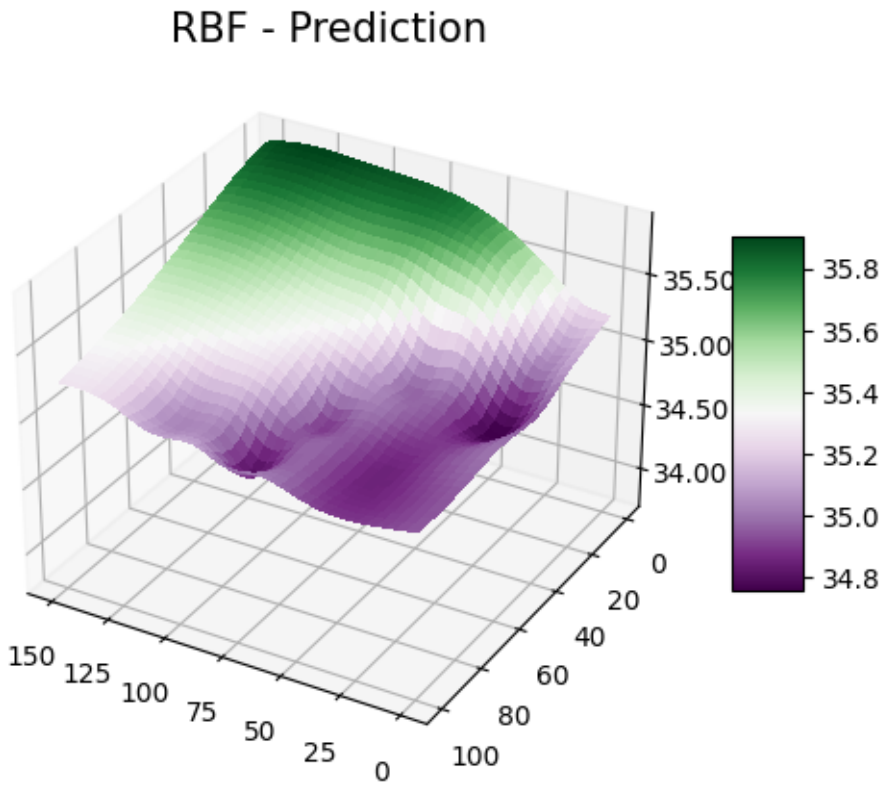


Figure 5.4: RBF - Prediction (K-Means Clustering, $k = 20$ clusters)

			Training ratio $= \lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
h_i	h_j	Equivalent no. of clusters k	50%	60%	70%	80%	90%
10	10	150	0.1259	0.1103	0.1110	0.1062	0.1102
20	20	40	0.1375	0.1262	0.1290	0.1140	0.1152
10	15	100	0.1011	0.0938	0.0828	0.0755	0.0801
20	15	50	0.1370	0.1216	0.1261	0.1216	0.1255

Table 5.27: RMSE with $M = 10$ repetitions, Uniform Grid Clustering

uniform grid clustering seems to spread the error across the whole surface. This contrasts with the results in K-means clustering, as we don't discover any region with larger errors like the absolute error plots from using K-means clustering. This may explain why uniform grid clustering is a much better clustering algorithm than K-means clustering when comparing the accuracy metrics.

Comparing Clustering Methods

We wanted to compare both cluster methods when they have the same number of clusters on the surface.

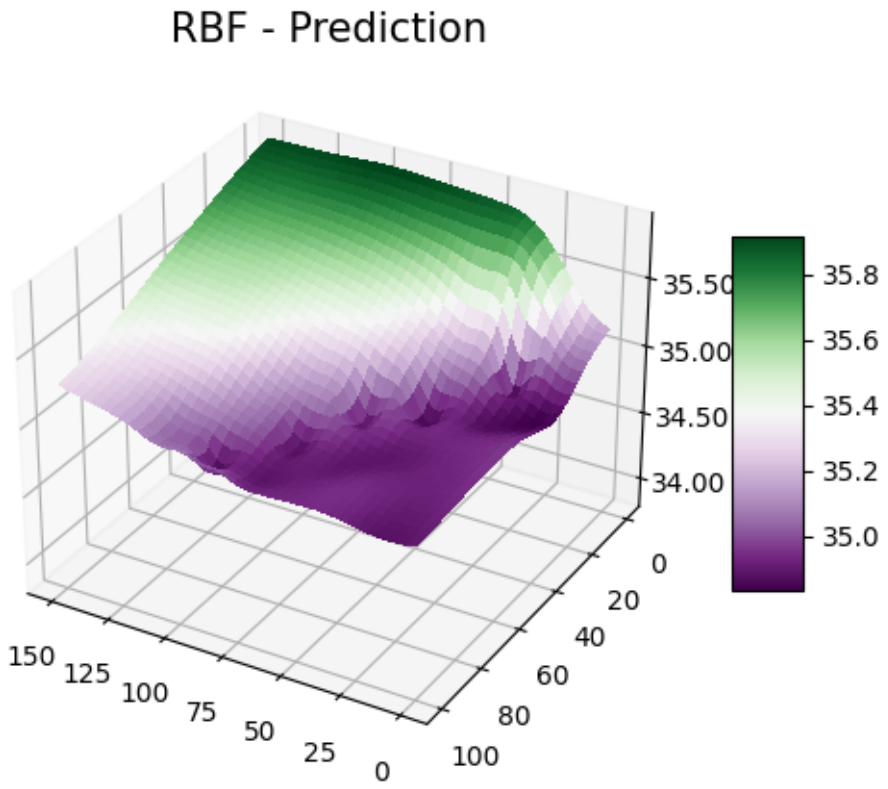


Figure 5.5: RBF - Prediction (K-Means Clustering, $k = 50$ clusters)

			Training ratio = $\lvert\mathcal{T}\rvert/(L - \lvert\mathcal{U}\rvert)$				
h_i	h_j	Equivalent no. of clusters k	50%	60%	70%	80%	90%
10	10	150	0.00357	0.00313	0.00315	0.00301	0.00312
20	20	40	0.00390	0.00358	0.00366	0.00323	0.00327
10	15	100	0.00287	0.00266	0.00235	0.00214	0.00227
20	15	50	0.00388	0.00345	0.00357	0.00345	0.00356

Table 5.28: Relative RMSE with $M = 10$ repetitions, Uniform Grid Clustering

Comparing both models with different clustering algorithms, this new uniform grid clustering method seemed to perform really well. In undertaking previous experiments, we observed that our models performed best with a training ratio of 80%. But the objective of choosing the cluster centres ourselves was to determine whether it performed better than K-means clustering. But now, we want to see how both of these clustering techniques would work in the same code, which would ensure reliability in the model.

So, we fixed the training ratio to be 80% and predetermined the values for the

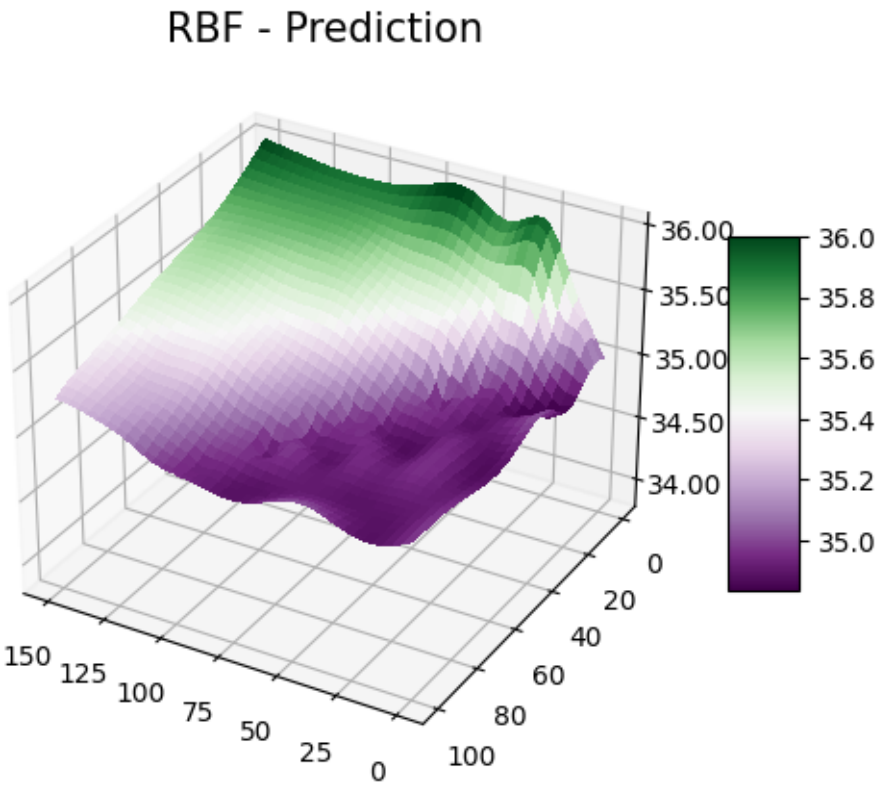


Figure 5.6: RBF - Prediction (K-Means Clustering, $k = 100$ clusters)

weights to better compare the prediction results using the two different clustering methods. So for K-means, we had $k = 50$, which performed the best, and the equivalent factor values of the uniform grid clusters. We also compared for $k = 100$, and the equivalent factor values were $h_i = 10, h_j = 15$. The results are shown in Tables 5.29 & 5.30.

	K-Means Clustering	Uniform Grid Clustering	
Accuracy Metric	k	h_i	h_j
	50	20	15
RMSE	0.1080	0.1125	
Relative RMSE	0.00306	0.00318	

Table 5.29: RMSE for Training Ratio = 80%

From Tables 5.29 & 5.30 as well as Tables 5.21 to 5.28, we get the following conclusions:

- When using K-means clustering at its optimal value (from our experiments, $k = 50$), it does outperform the uniform grid clustering.

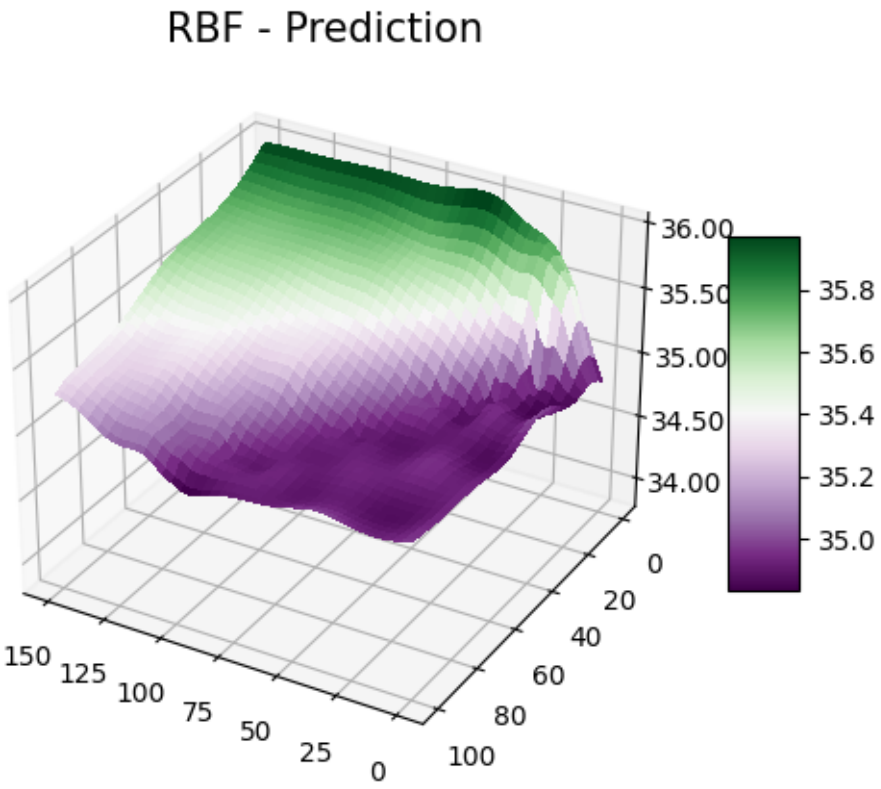


Figure 5.7: RBF - Prediction (K-Means Clustering, $k = 150$ clusters)

	K-Means Clustering	Uniform Grid Clustering	
Accuracy Metric	k	h_i	h_j
	100	10	15
RMSE	0.2062	0.0777	
Relative RMSE	0.005847	0.002204	

Table 5.30: RMSE for Training ratio = 80%

- However, overall, the prediction errors using the uniform grid clustering method varied less compared to the k-means clustering.
- Hence, we conclude that uniform grid clustering is a more optimal clustering method for RBF-Net using our 2-dimensional dataset $\mathcal{D}_t$.

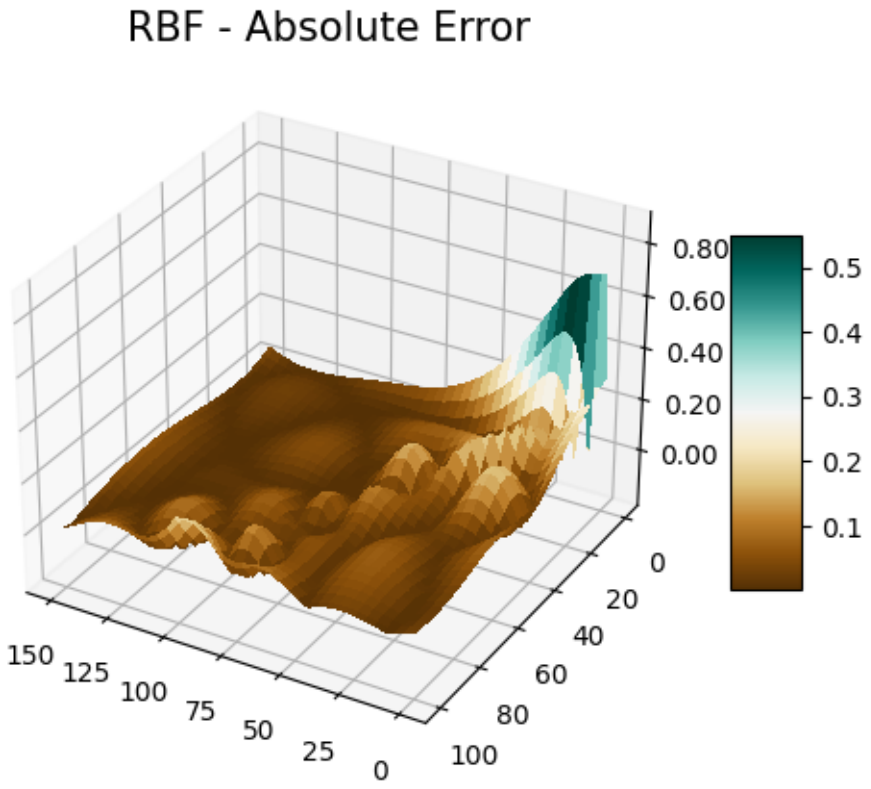


Figure 5.8: RBF - Absolute Error (K-Means Clustering, $k = 20$ clusters)

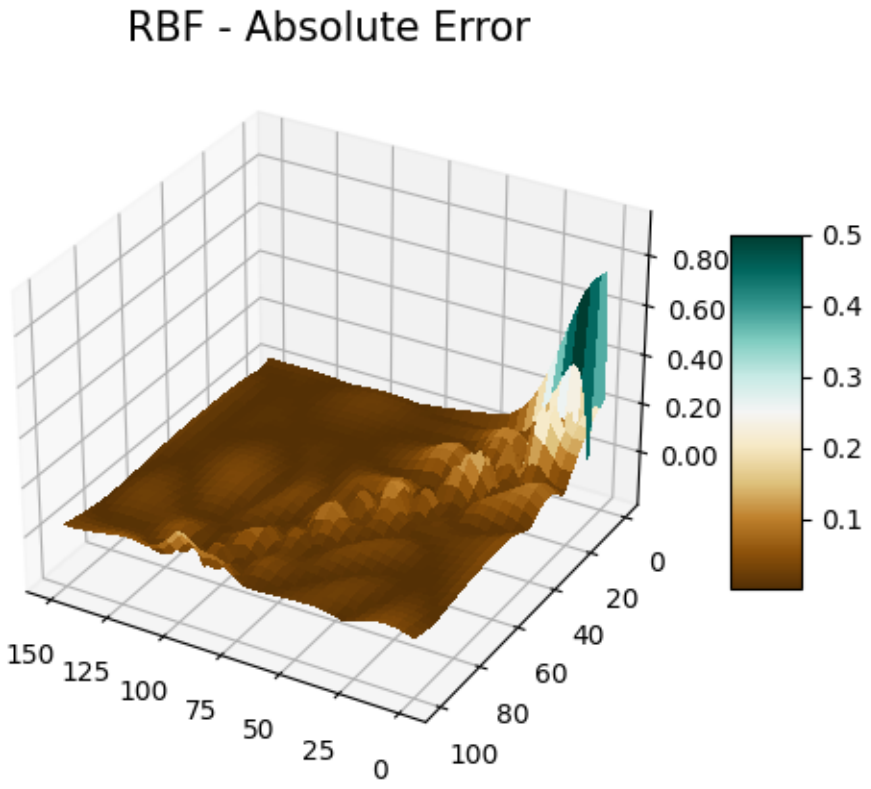


Figure 5.9: RBF - Absolute Error (K-Means Clustering, $k = 50$ clusters)

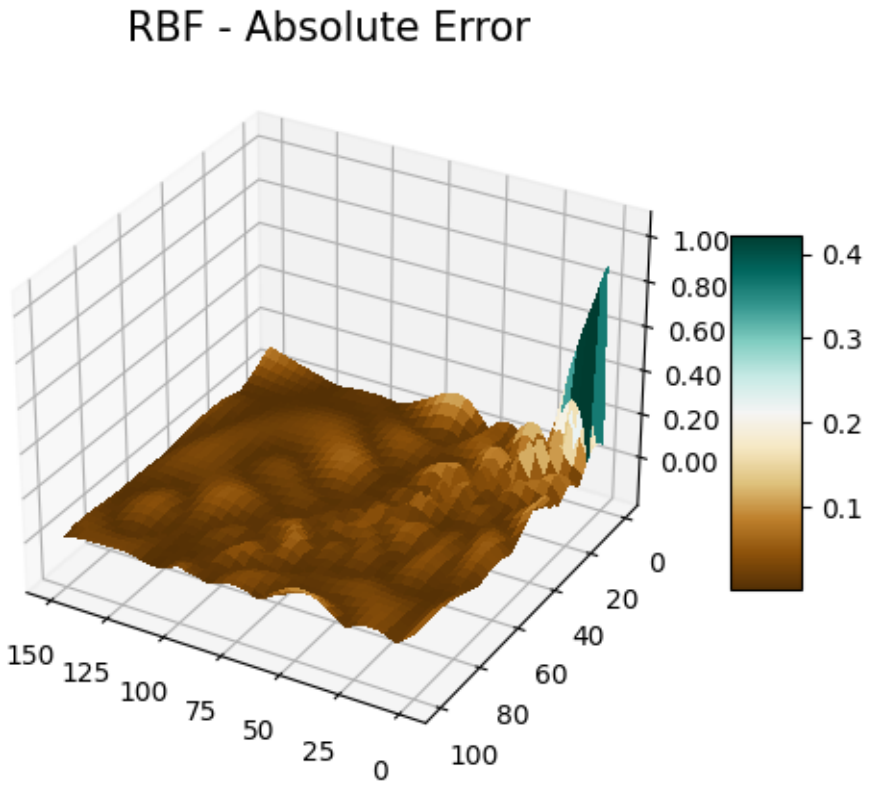


Figure 5.10: RBF - Absolute Error (K-Means Clustering, $k = 100$ clusters)

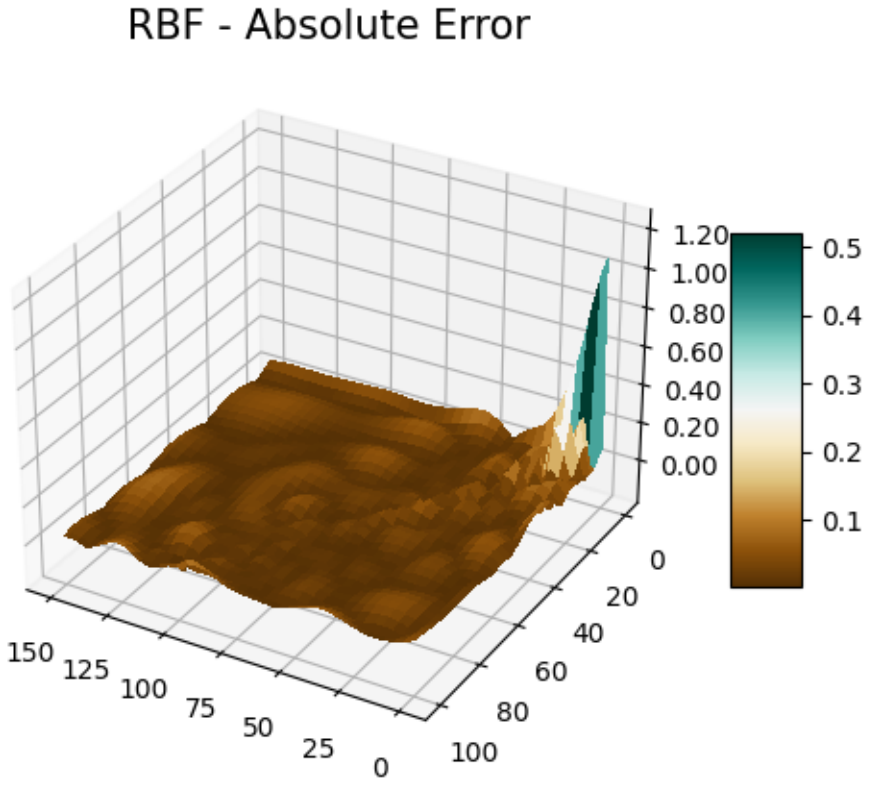


Figure 5.11: RBF - Absolute Error (K-Means Clustering, $k = 150$ clusters)

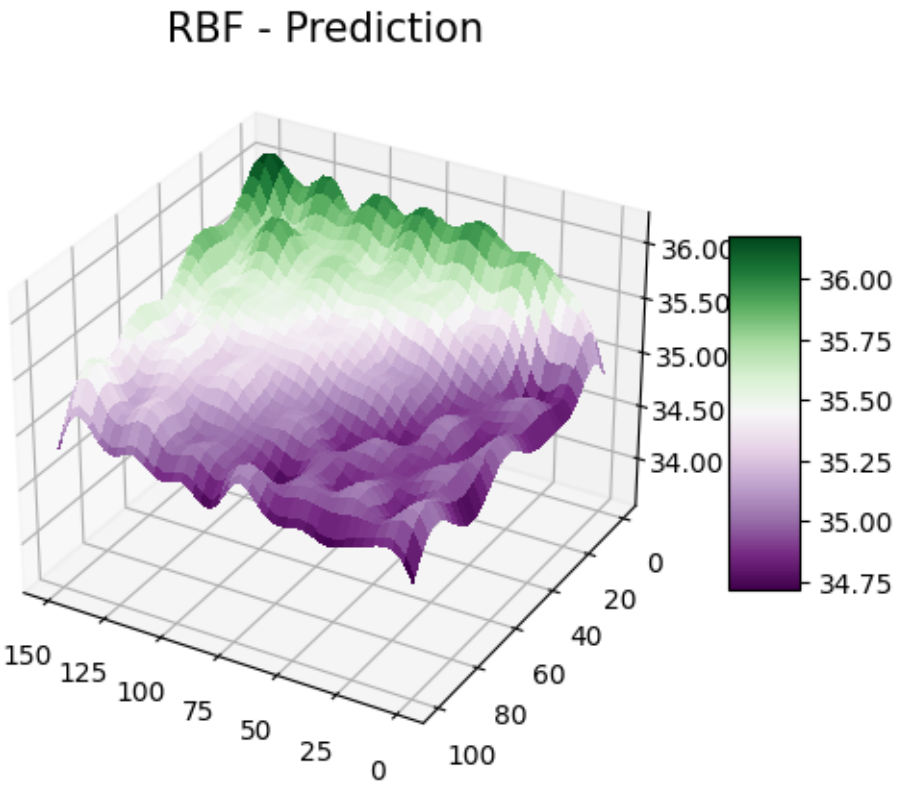


Figure 5.12: RBF - Prediction (Uniform Grid Clustering, $h_i = h_j = 10$, equivalent to $k = 150$)

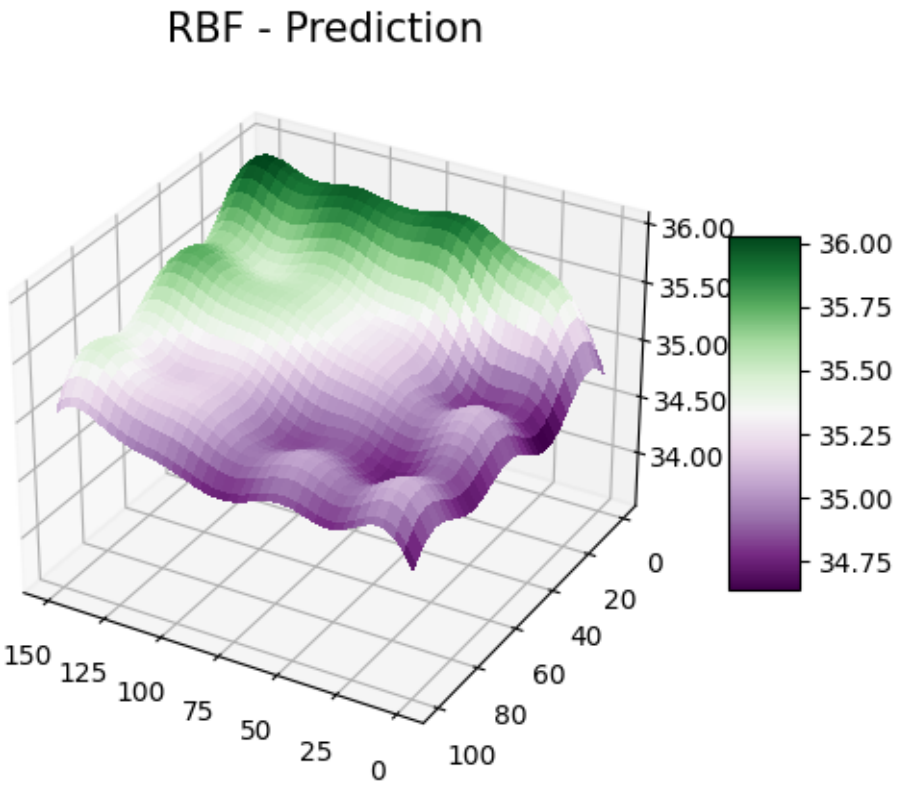


Figure 5.13: RBF - Prediction (Uniform Grid Clustering, $h_i = h_j = 20$, equivalent to $k = 40$)

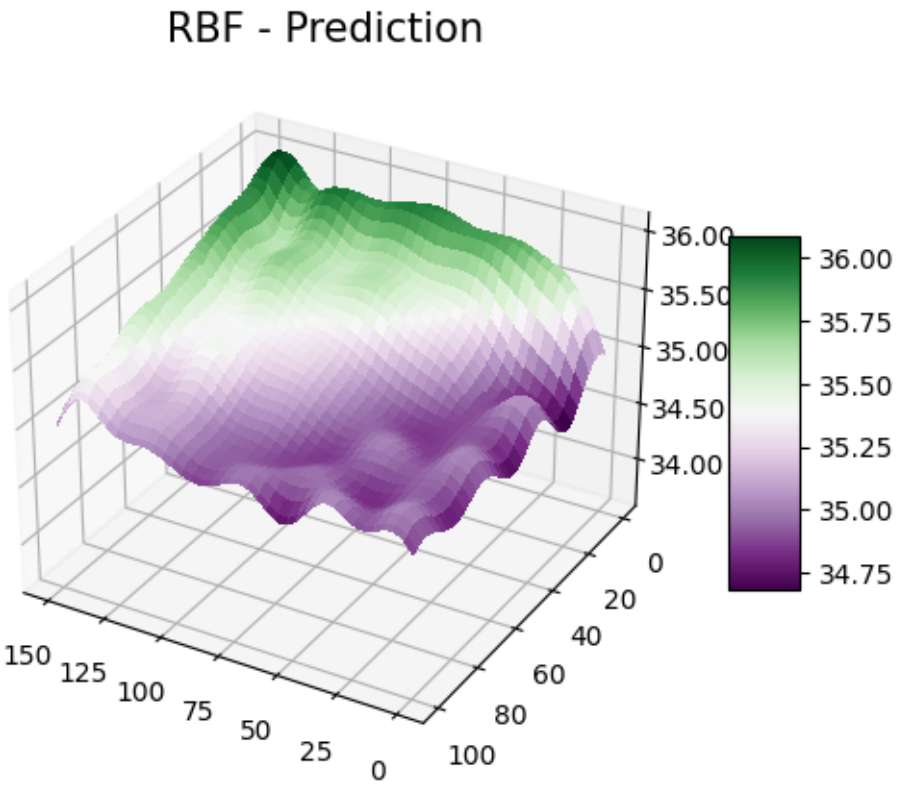


Figure 5.14: RBF - Prediction (Uniform Grid Clustering, $h_i = 10, h_j = 15$, equivalent to $k = 100$)

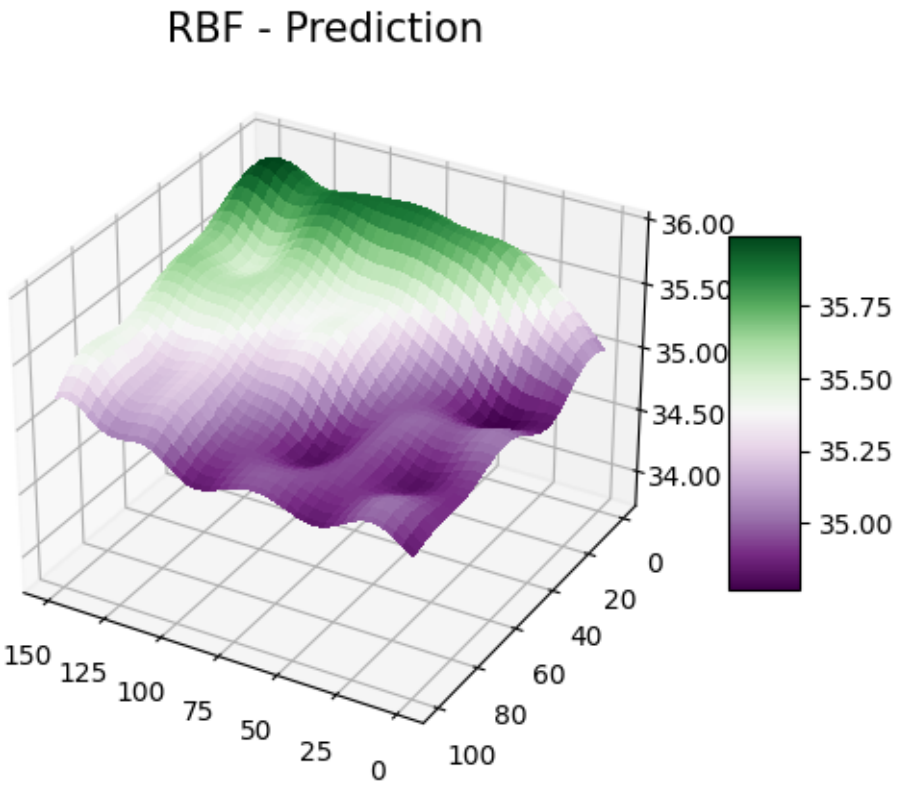


Figure 5.15: RBF - Prediction (Uniform Grid Clustering, $h_i = 20, h_j = 15$, equivalent to $k = 50$)

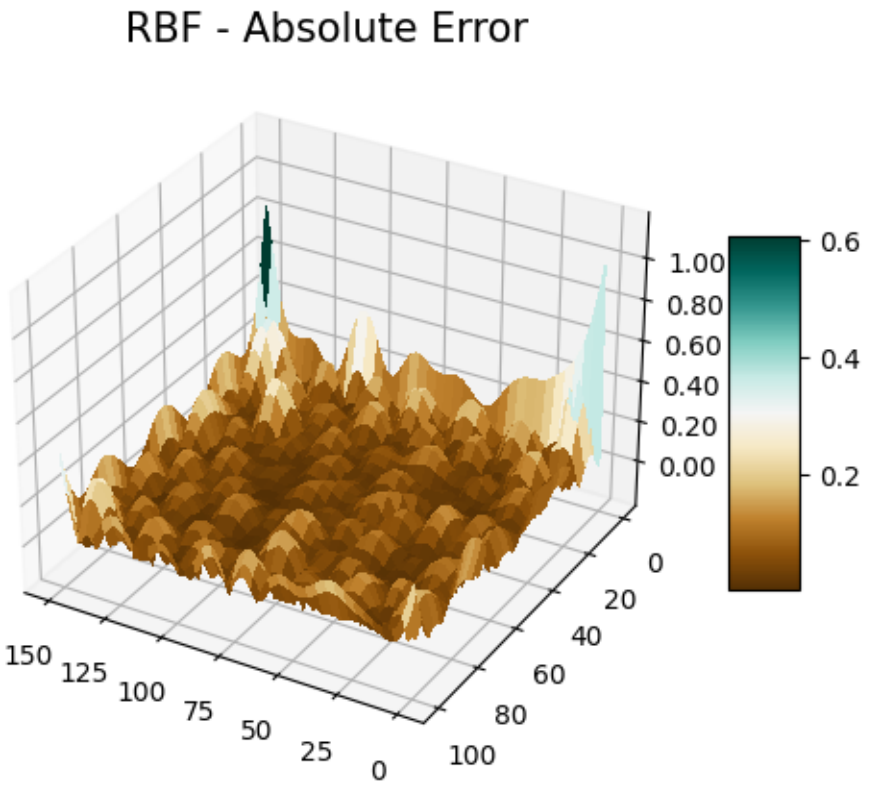


Figure 5.16: RBF - Absolute Error (Uniform Grid Clustering, $h_i = 10, h_j = 10$, equivalent to $k = 150$)

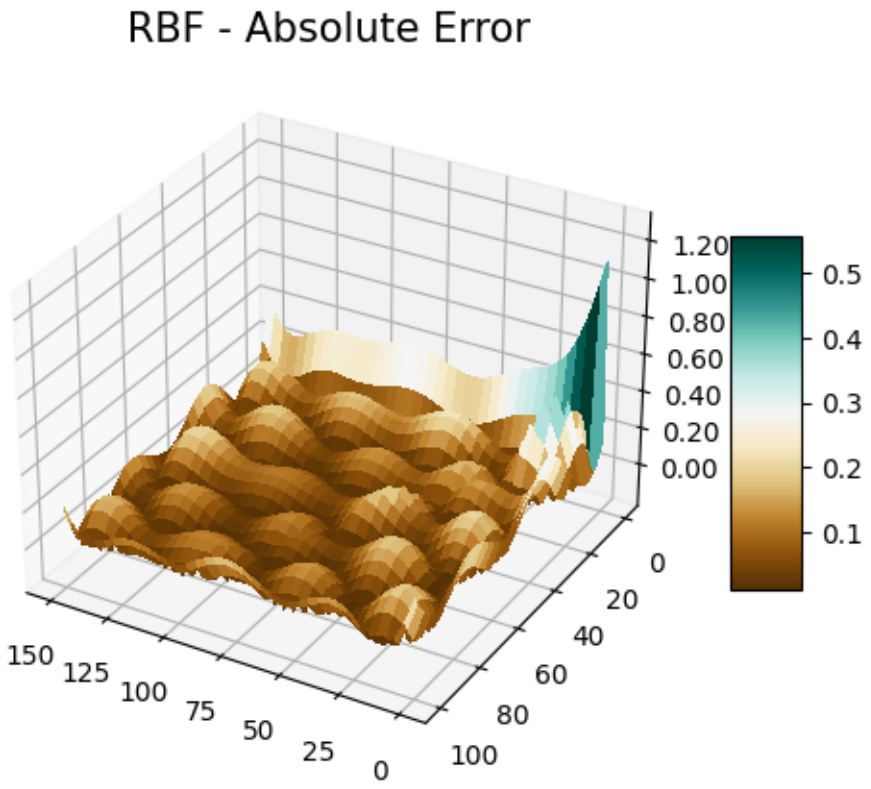


Figure 5.17: RBF - Absolute Error (Uniform Grid Clustering, $h_i = h_j = 20$, equivalent to $k = 40$)

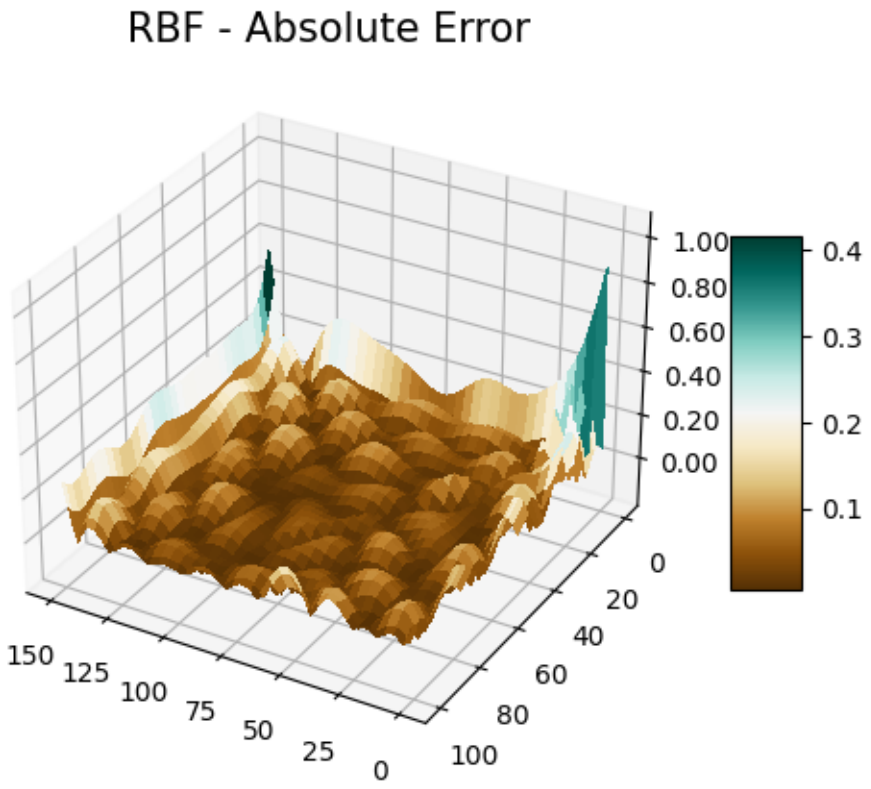


Figure 5.18: RBF - Absolute Error (Uniform Grid Clustering, $h_i = 10, h_j = 15$, equivalent to $k = 100$)

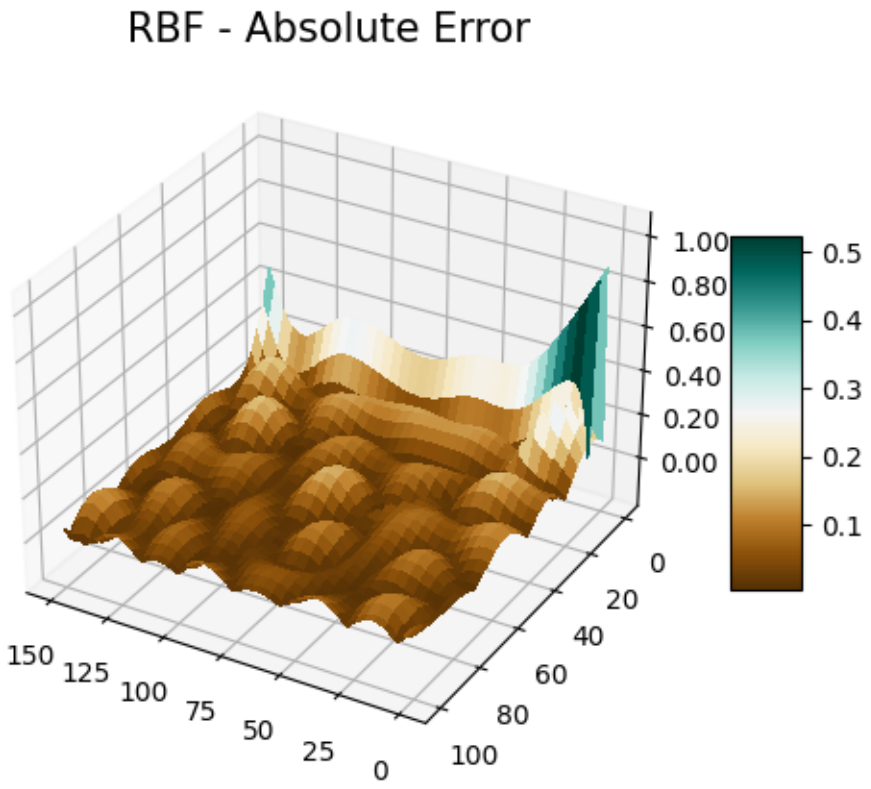


Figure 5.19: RBF - Absolute Error (Uniform Grid Clustering, $h_i = 20, h_j = 15$, equivalent to $k = 50$)

5.7 Linear Regression (2D)

Similarly, we wanted to compare RBF networks with linear regression in the 2-dimensional case. We followed the same methodologies as outlined in Section 5.2.2 and we get the results shown in Tables 5.31 & 5.32. We also have visualisations of the prediction and absolute error, which are given in Figures 5.20 & 5.21.

	Training Ratio = $\lvert\mathcal{T}\rvert/(L-\lvert\mathcal{U}\rvert)$				
M (No. of repetitions)	50%	60%	70%	80%	90%
5	0.1384102	0.1384063	0.1384103	0.1384051	0.1384047
10	0.1384114	0.1384115	0.1384126	0.1384059	0.1384048

Table 5.31: RMSE - Linear Regression (2-dimensional)

	Training Ratio = $\lvert\mathcal{T}\rvert/(L-\lvert\mathcal{U}\rvert)$				
M (No. of repetitions)	50%	60%	70%	80%	90%
5	0.003923497	0.003923387	0.003923502	0.003923354	0.003923341
10	0.003923533	0.003923525	0.003923568	0.003923375	0.003923344

Table 5.32: Relative RMSE - Linear Regression (2-dimensional)

When comparing results from Tables 5.31 & 5.32 with RBF results in Tables 5.21 to 5.28, we can clearly conclude that linear regression did not perform as well as either RBF-Net models using the $\mathcal{D}_t$ dataset.

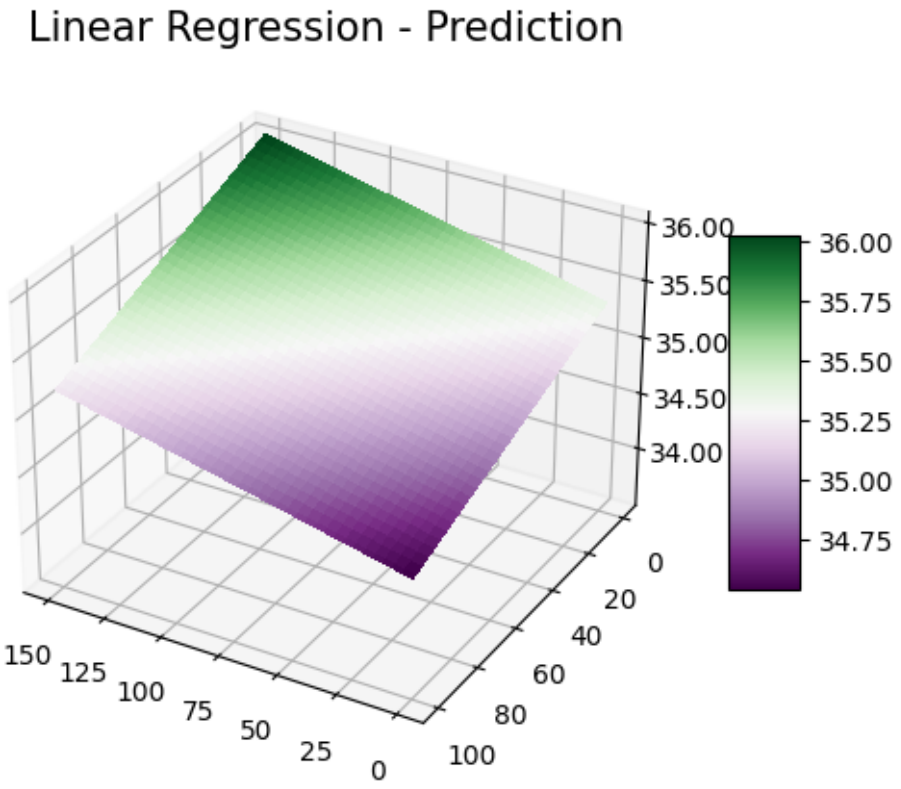


Figure 5.20: Linear Regression - Prediction

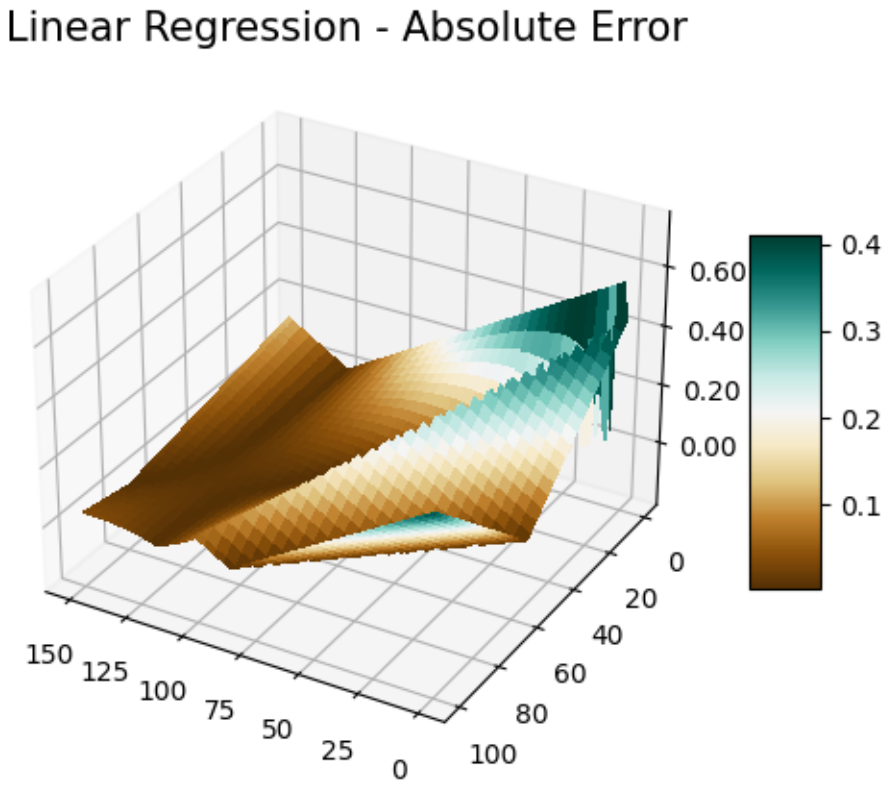


Figure 5.21: Linear Regression - Absolute Error

5.8 Main conclusions

From our simulations, we obtained the following conclusions:

- The optimal values of our parameters for RBF-Net 1D were $k = 20$ clusters and 500 epochs.
- The parameters that optimised accuracy metrics for LSTM were batch size = 5, lookback = 5, and epochs = 1000.
- Comparing both models for 1-dimensional simulations, LSTM with Adam optimiser performed better than RBF-Net with K-means clustering.
- Linear regression for 1-dimension did not perform as well as neural networks in predicting the price of a ride-share service at time t'.
- In comparing both clustering methods used for RBF-Net 2D, while K-means clustering does perform well for specific values (e.g. when $k = 50$ clusters), overall, Uniform Grid clustering performed better than K-means clustering when using the $\mathcal{D}_t$ dataset.
- For 2D, using neural networks produced vastly better results than linear regression.
- Overall, neural networks outperformed linear regression for both 1D and 2D cases in predicting the price of a ride-sharing service at any time t'.

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis, we have explored the predictive ability that neural networks have in comparison to classical modelling techniques such as linear regression in modelling a dynamic pricing model.

We have shown that with optimal parameters for Radial Basis Function and Long Short-Term Memory neural network models, the Long Short-Term Memory Network with Adam optimiser performed better than the Radial Basis Function Network with K-means clustering for 1-dimensional data. Additionally, both neural network models outperformed linear regression in predicting the price of the ride-share service.

For our 2-dimensional simulations, using two Radial Basis Function Networks with different clustering algorithms, uniform grid clustering provided a better result in minimising our accuracy metrics in comparison to K-means clustering. Linear regression for 2D also didn't fare well, producing worse results than neural networks once again.

With our best models, we were able to predict the price of an Uber ride-sharing service with a root-mean-square error of less than 1, so our models have achieved very accurate predictions.

Hence, we can conclude that a dynamic pricing model can get more accurate predictions when using neural networks compared to linear regression, for both 1-dimensional and 2-dimensional simulations.

6.2 Future work

In terms of future work, there are various directions that can be taken to further improve our simulations.

- If possible we could try to use the 2D plane and run it through LSTM and compare its performance to the RBF network for 2-dimensional data.
- We also created 2D planes when we fixed supply, and fixed demand separately. See Figures 4.8 and 4.7.
- If running speed allows it, we can extend from 2-dimensional simulations and attempt to run a 3D dataset through our RBF Network, as our modified model is able to accommodate multi-dimensional data as well.
- While we only tested for particular values over a huge range for batch size, lookback, and epochs, we can choose to be more accurate by testing all values in smaller ranges and seeing what combination of parameters best optimises the LSTM model. For the RBF Network in 1D, we can do something similar to find even better parameters for k (No. of clusters) and epochs.
- For our 2-dimensional simulations, we can do similar investigations for the RBF network. We can adjust the number of clusters or the values of h_i and h_j and determine whether the models perform better as a result of these modifications.
- An example of adjusting the number of clusters k is that we could potentially try to look at extreme boundary cases and see how these cases could affect our prediction, for example, when $k = 1$ or when $k = |\mathcal{T}|$ and see whether these results support our findings from preliminary empirical results.

- Most importantly, if the opportunity allows, having access to actual datasets, rather than a dataset that has been generated ourselves, would give us fewer assumptions to work with when training our data into the model.

Appendix A

Original Contribution

To access the Python code used to obtain the results and figures in this thesis, it is provided in the following link for the GitHub repository:
https://github.com/DanielXMa/honours-dynamic_pricing

References

[1] Aggarwal, C.C. (2018). Neural Networks and Deep Learning: A Textbook. Cham: Springer International Publishing.

[2] Azad Fouad Arif, S., Subrahmanyam, S. (2022). 'Penetration Pricing Strategy and Customer Retention – An Analysis', The Journal of Positive Psychology 6(5) 7058-7072

[3] Bishop, C. (2006). Pattern Recognition and Machine Learning. New York: Springer

[4] Fiig, T., Le Guen, R. & Gauchet, M. (2018). "Dynamic pricing of airline offers". J Revenue Pricing Manag 17, 381–393, https://doi.org/10.1057/s41272-018-0147-z

[5] Forgy, E. W. (1965). "Cluster Analysis of Multivariate Data: Efficiency versus Interpretability of Classifications". Biometrics. 21 (3), 768-769.

[6] Fuchs, J. (2022). *Dynamic Pricing: The Complete Guide*. [online] blog.hubspot.com. Available at: https://blog.hubspot.com/sales/dynamic-pricing.

[7] Goodfellow, I., Bengio, Y. and Courville, A. (2016). Deep Learning. [online] Deeplearningbook.org. Available at: https://www.deeplearningbook.org/.

[8] Hastie, T., Tibshirani, R. and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer, 2nd ed.

[9] Hayes, A. (2022). *Peak Pricing: Definition, How It Works, Examples.* [online] Investopedia. Available at: https://www.investopedia.com/terms/p/peak-pricing.asp

[10] Hochreiter, S., Schmidhuber, J., "Long short-term memory," *Neural computation*, 9(8), pp. 1735-180, 1997.

[11] Jeffrey, K. *Dynamic Pricing Strategy: Increase vs. Decrease.* [online] Available at: https://prenohq.com/blog/dynamic-pricing-strategy-increase-vs-decrease/

[12] Kingma, D. & Ba, J. 2014, 'Adam: A method for stochastic optimization', *Computer Science*, DOI: 10.48550/ARXIV.1412.6980

[13] Kumar, K., Thakur, G. S. M. (2012). "Advanced Applications of Neural Networks and Artificial Intelligence: A Review". In: International Journal of Information Technology and Computer Science 4(6), pp. 57–68.

[14] Lin, C., Shang, K. and Sun, P. (2023). 'Wait Time–Based Pricing for Queues with Customer-Chosen Service Times' 69(4) Management Science 2127

[15] Lloyd, P.S. (1957). "Least Square Quantization in PCM". Bell Telephone Laboratories Paper. Published in journal much later: Lloyd, Stuart P. (1982). "Least squares quantization in PCM". IEEE Transactions on Information Theory. 28(2): 129–137.

[16] Nagle, T.T. and Müller, G. (2018). *The Strategy and Tactics of Pricing: A Guide to Growing More Profitably.* New York; London: Routledge.

[17] Nielsen, M. A. *Neural networks and deep learning*, vol. 25. San Francisco, CA, USA: Determination press, 2015.

[18] C. Olah. (2015). *Understanding LSTM Networks* [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[19] Özer, Ö. (2014). *The Oxford Handbook of Pricing Management*. Oxford: Oxford University Press.

[20] Punwasee, A. (2022). *Price Segmentation: A Crucial Tool For Improving Pricing Strategy*. [online] Available at: https://revenueml.com/2022/04/price-segmentation-a-crucial-tool-for-improving-pricing-strategy.

[21] Rosenblatt, F. (1958). "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological review*, 65(6), pp. 386-408,

[22] Schröder, M., Storch, DM., Marszal, P. et al. (2020). *Anomalous supply shortages from dynamic pricing in on-demand mobility*. Nat Commun 11, 4831

[23] Sharif Ahmadian, A. (2016). *Chapter 7 - Numerical Modeling and Simulation*. [online] ScienceDirect. Available at: https://www.sciencedirect.com/science/article/abs/pii/B9780128024133000079

[24] Spann, M., Fischer, M. & Tellis, G.J. (2015). 'Skimming or Penetration? Strategic Dynamic Pricing for New Products', *Marketing Science* (Providence, R.I.), 34(2), pp. 235–249.

[25] Tang, C.S., Onesun S. Y. and Zhan, D. (2023). When should grocery stores adopt time-based pricing? Impact of competition and negative congestion externality. *Production and Operations Management*. doi:https://doi.org/10.1111/poms.14010.

[26] Tanir, B. (2023). *Council Post: Why And How You Should Perform A Competitive Pricing Analysis*. [online] Forbes. Available at: https://www.forbes.com/sites/forbestechcouncil/2023/10/10/why-and-how-you-should-perform-a-competitive-pricing-analysis/?sh=15bf0455655b

[27] Yin, C. and Han, J. (2021). Dynamic Pricing Model of E-Commerce Platforms Based on Deep Reinforcement Learning. *Computer Modeling in Engineering & Sciences*, 127(1), pp.291–307. doi:https://doi.org/10.32604/cmes.2021.014347.