NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

Master Thesis

Applications of Neural Networks in production management problems

Author:
Georgios VOUNTOURAKIS

Supervisor:
Prof. Apostolos BURNETAS

A thesis submitted in partial fulfillment of the requirements for the degree of
M.Sc in Statistics and Operational research
in the

Faculty of Science,
Department of Mathematics

Athens, February 2021

Η παρούσα Διπλωματική Εργασία
εκπονήθηκε στα πλαίσια των σπουδών
για την απόκτηση του

Μεταπτυχιακού Διπλώματος Ειδίκευσης

«Στατιστική και Επιχειρησιακή Έρευνα»

που απονέμει το

Τμήμα Μαθηματικών

του

Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών

Εγκρίθηκε την …… από Εξεταστική Επιτροπή

Αποτελούμενη από τους:

Ονοματεπώνυμο	Βαθμίδα	Υπογραφή
1. Απόστολος Μπουρνέτας (Επιβλέπων Καθηγητής)	Καθηγητής	. . . . .
2. Αθανασία Μάνου	Επίκουρη Καθηγήτρια	. . . . .
3. Αντώνιος Οικονόμου	Καθηγητής	. . . . .

Περίληψη

Τα προβλήματα οργάνωσης παραγωγής και διαχείρισης αποθεμάτων για πολλά προϊόντα και σταθμούς παραγωγής ανάγονται σε προβλήματα βελτιστοποίησης μεγάλης πολυπλοκότητας. Επίσης, σε πολλές περιπτώσεις το πρόβλημα πρέπει να λυθεί κάτω από συνθήκες ελλιπούς πληροφόρησης ως προς τις κατανομές πιθανότητας της ζήτησης, τις παραμέτρους κόστους κλπ.

Τα τεχνητά νευρωνικά δίκτυα είναι αρκετά ισχυρά υπολογιστικά μοντέλα. Στην εργασία θα γίνει ανασκόπηση των βασικών μοντέλων τεχνητών νευρωνικών δικτύων και των εφαρμογών τους σε προβλήματα βελτιστοποίησης, με έμφαση στο πρόβλημα του εφημεριδοπώλη. Επιπλέον, θα προταθούν νευρωνικά δίκτυα για αυτά τα προβλήματα και θα μελετηθεί η απόδοσή τους μέσω υπολογιστικών πειραμάτων

Τα νευρωνικά δίκτυα αποτελούνται από πολλούς νευρώνες, που είναι συνδεδεμένοι μεταξύ τους και στέλνουν σήματα ο ένας στον άλλον για να υλοποιήσουν ένα στόχο. Τα τεχνητά νευρωνικά δίκτυα δημιουργήθηκαν για να μιμηθούν τον τρόπο με τον οποίο λειτουργεί ο ανθρώπινος εγκέφαλο και να εκμεταλλευτούν τη δομή του και συνεπώς είναι πολύπλοκα μοντέλα με πολλές ικανότητες. Οι συνδέσεις μεταξύ των νευρώνων γίνονται είτε πιο δυνατές είτε πιο αδύναμες βάσει μίας διαδικασία μάθησης, έτσι ώστε να υλοποιούν καλύτερα τον επιθυμητό στόχο. Η διαδικασία μάθησης που θα επικεντρωθούμε ονομάζεται επιβλεπόμενη μάθηση όπου το νευρωνικό δίκτυο έχει ένα σύνολο δεδομένων εισόδου-εξόδου από το οποίο εκπαιδεύεται έτσι ώστε να μπορεί να γενικεύσει και να δίνει αποτέλεσμα μικρού σφάλματος όταν του δοθεί άγνωστη είσοδος.

Θα εστιάσουμε στις εφαρμογές των νευρωνικών δικτύων στο πρόβλημα του εφημεριδοπώλη και στις παραλλαγές του. Στο πρόβλημα του εφημεριδοπώλη, ο διαχειριστής πρέπει να αποφασίσει τη βέλτιστη ποσότητα παραγγελίας ενός ευπαθούς προϊόντος έτσι ώστε να ελαχιστοποιήσει το κόστος. Οι περιορισμοί του προβλήματος είναι ότι η πραγματική κατανομή πιθανότητας της ζήτησης είναι άγνωστη και κάθε προϊόν που δεν πωλήθηκε δεν μπορεί να αποθηκευτεί για μελλοντική χρήση. Μία παραλλαγή αυτού του προβλήματος είναι όταν ο διαχειριστής πρέπει να αποφασίσει την ποσότητα παραγγελίας για περισσότερα από ένα προϊόντα. Επιπλέον, μια ακόμα παραλλαγή είναι όταν ο διαχειριστής πρέπει να αποφασίσει την βέλτιστη τιμή για να πουλήσει το προϊόν καθώς και τη βέλτιστη ποσότητα της παραγγελίας με στόχο την μεγιστοποίηση των κερδών του.

Θα γίνει μία ανασκόπηση των νευρωνικών δικτύων που έχουν προταθεί στη βιβλιογραφία και στη συνέχεια θα προτείνουμε το δικό μας νευρωνικό δίκτυο για αυτά τα προβλήματα. Ειδικότερα, θα παρουσιάσουμε διάφορα νευρωνικά δίκτυα με διαφορετικό αριθμό νευρώνων, κανόνων μάθησης και συναρτήσεων

ενεργοποίησης.
Τέλος θα υλοποιήσουμε την απόδοσή τους μέσω υπολογιστικών πειραμάτων με αρκετές διαφορετικές παραμέτρους.

Πιο συγκεκριμένα η δομή της εργασίας είναι ως εξής:

Στο δεύτερο κεφάλαιο, θα παρουσιαστούν τα βασικά στοιχεία ενός νευρωνικού δικτύου, τις διαφορετικές αρχιτεκτονικές καθώς και διαφορετικές μέθοδοι εκπαίδευσης έτσι ώστε να αναλυθεί ο τρόπος με τον οποίο ένα νευρωνικό δίκτυο εκπαιδεύεται για να επιλύει ένα πρόβλημα.
Στο τρίτο κεφάλαιο, θα οριστεί το πρόβλημα του εφημεριδοπώλη καθώς και κάποιες παραλλαγές του που θα αναλυθούν στα επόμενα κεφάλαια.
Στο τέταρτο κεφάλαιο, θα γίνει ανασκόπηση των νευρωνικών δικτύων που προτείνονται στη βιβλιογραφία για την επίλυση των προβλημάτων που ορίσθηκαν στο δεύτερο κεφάλαιο
Στο πέμπτο κεφάλαιο, θα κατασκευάσουμε νευρωνικά δίκτυα που επιλύουν διαφορετικές παραλλαγές του προβλήματος του εφημεριδοπώλη και θα μελετηθεί η απόδοσή τους.

Acknowledgements

First of all, I would like to thank my thesis supervisor, Professor Apostolos Burnetas, for his guidance through each stage of the process, for all his help and advice, and for inspiring my interest in the field of artificial neural networks and optimization.

In addition, I would like to express my deepest appreciation to my examination committee, Professor Antonis Economou and Assistant Professor Athanasia Manou, for their insightful comments and suggestions.

Finally, I would like to express my gratitude to my family and friends for all their encouragement and support during my studies.

Contents

1 Introduction

The production management problems for many products are reduced to optimization problems of high complexity. Moreover, in many cases the problem has to be solved under partial information of the distribution of probability of demand, the parameters of the cost etc.

Artificial neural networks are a very powerful computational model. In this thesis we will present the basic models of Artificial Neural Networks and their applications to optimization problems with main focus on the newsvendor problem. Furthermore, we will propose our own neural network for these problems and evaluate their performance upon numerical experiments.

Neural Networks consist of many neurons that are connected with each other and transmit signals to one another in order to perform some task. Artificial Neural Networks were created in order to try to mimic the way the human brain functions and take advantage of the structure and so they are very complex models with many capabilities. The connections between the neurons become stronger or weaker according to some learning procedure in order to better perform a task. The learning technique we will focus on this thesis is called supervised learning, where the Neural Network is given some training data in order to adjust the parameters and be able to generalize so it can perform the task with a minimum error when faced with unseen data.

We will mainly focus on the applications of neural networks to the newsvendor problem along with its variations. In the newsvendor problem, the inventory manager wants to find the optimal order quantity of a perishable product in order to minimize his cost. The constrains of this problem is that the true distribution of the demand is unknown to the inventory manager and every unsold product can not be stored for later use. One variation of this problem is when the inventory manager has to make a decision for the order quantity more than one product. Moreover, another variation is when the inventory manager has to decide the optimal price to sell the product as well as the optimal order quantity in order to maximize the profits.

We will review some neural networks that have been proposed in the literature and then we will propose our own neural networks for these problems. We will present a variety of neural networks with different number of neurons, learning rules and cost functions.
Finally, we will evaluate their performance by numerical experiments with different parameters

More precisely, the structure of this thesis is as follows:

In the second chapter, we will present the basic components of a neural network, its different architectures, as well as the different training techniques in order to fully understand how a neural network is trained to perform a task. Finally, we will discuss some methods to overcome some common pitfalls of the neural networks.
In the third chapter, we will define the newsvendor problem along with some variations that we will focus on later chapters.
In the fourth chapter, we will review some proposed neural networks that focus on the problems we defined in the second chapter.
In the fifth chapter, we will create our own neural networks to solve the different variations of the newsvendor problem and we will test their performance.

2 Artificial Neural Networks

2.1 History

Artificial neural networks are inspired by the field of biology and more precisely by the way the characteristics of the brain function are performing a task. The neurons are the structural constituents of the brain and the human brain has approximately 10 billion neurons that are highly connected, with approximately 10^{14} connections per neuron.

In a high-level description, neurons are composed mainly by: the dendrites, the cell body and the axon. Electrical signals are carried by the dendrites into the cell body, that receives and processes the incoming signals. Afterwards, the signals are carried by the axon from the cell body to other neurons. The connection between two neurons, and specifically from the axon of a neuron to a dendrite of another neuron, is called synapse and it is responsible for allowing the interaction between two neurons.

The function of the neural network is determined by the neurons and the synapses between neurons. A part of the neural structure is defined at birth, but other parts are developed through learning where in this case synapses are strengthened while other synapses are weakened.

Therefore, the inspiration for artificial neural networks arises from the fact that the computations in the human brain are done in an entirely different way than in conventional computers. Despite this fact, artificial neural networks do not approach the complexity of the brain.

The first artificial neural network was constructed in 1943 in the work of Warren McCulloch and Walter Pitts. They proved that artificial neurons could compute any arithmetic or logical function. A few years later, in 1949, Hebb presented the first rule for self-organized learning. Furthermore, in 1958, the first practical application of neural networks was presented by Rosenblatt, who proposed perceptron network together with the first model for supervised learning. The above are considered as the most pioneering contributions in the field of artificial neural networks.

Artificial neural networks are very powerful computational models and have numerous applications. Some of them are function approximation, prediction, sequential decision making, classification, pattern recognition and clustering.

2.2 Architecture

Artificial neural networks are a computational model which is composed by a set of connected nodes called artificial neurons, that we will simply call neurons. Each neuron is a computational entity and each connection between two neurons has a weight w. A neuron can have multiple input and output connections. The input of each neuron is the output of its input connection scaled by the weight w of the corresponding connection. A neuron has also an activation function, which determines the output of the neuron. The value of the output is the value of the activation function on the sum of a constant term with the linear combination of the neuron's inputs with the corresponding weights of each input connection.

An example of a simple artificial neural network is shown in Figure 1.

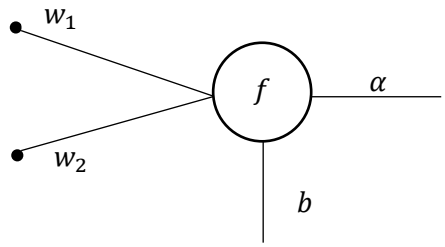


Figure 1

A computation of the neural network is as follows: each input node x_1, x_2 is multiplied with its weight w_1, w_2 respectively and then they are summed together with an extra term b, which is called bias. The result forms the net input z. Finally, the net input goes into an activation function f, which produces the neuron output α.

More precisely the mathematical form of the activation function is:

$$\alpha = f(w^T x + b)$$

Where x is the vector $[x_1, x_2]^T$ of the input, w is the vector $[w_1, w_2]^T$ of the weights of the connections and b, α are scalars.

Some questions that arise are what are the weights, the bias and the activation function.

The bias b can be viewed as another weight of a neuron with constant input $x_0 = 1$. The weights and the biases are the parameters of the neural network that will be adjusted by some learning rule to meet a desired goal. The activation function is chosen by the designer of the network and can be any function of his choice. The activation function and the learning rules are chosen so that the output reaches the desired goal.

The choice of the activation function depends on the problem the network is trying to solve.
For example, if the output is required to be 0 or 1, one common activation function that is used is shown in Figure 2:

$$f(z) = \begin{cases} 1, & if\ z \geq 0 \\ 0, & if\ z < 0 \end{cases},$$

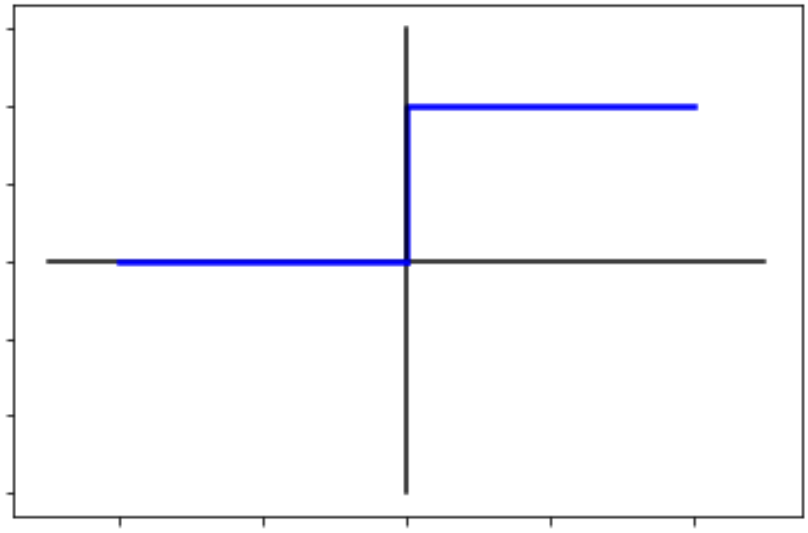


Figure 2

where $\alpha = f(z) = f(w^T x + b)$. It represents the operation that if the net input is $z \geq 0$ then the output neuron is 1, otherwise it is 0.
We can rewrite this as $w^T x + b > 0 \Rightarrow w^T x > -b$.
This implies that if the linear combination of the input vector with the weight vector is greater than some threshold b, then the output of the neural network is 1, otherwise it is 0.
In this example where we have 2 inputs x_1, x_2 , it is easy to make a visualization in a 2-dimentional plane to understand it better as shown in Figure 3, the $w^T x$ represents the decision boundary and everything that is in one side is 1 and everything on the other side is 0.

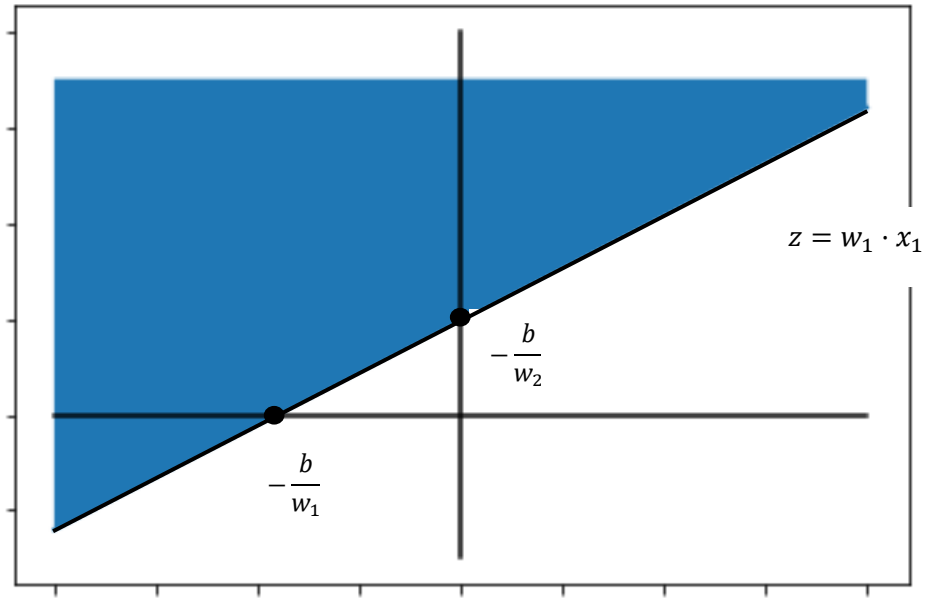
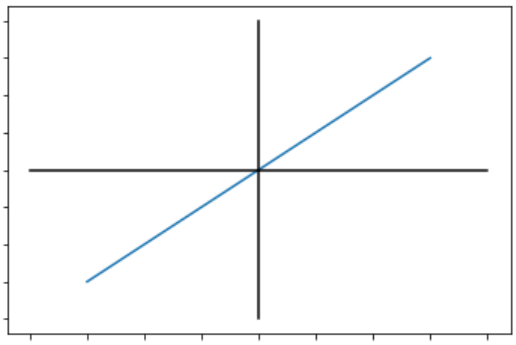
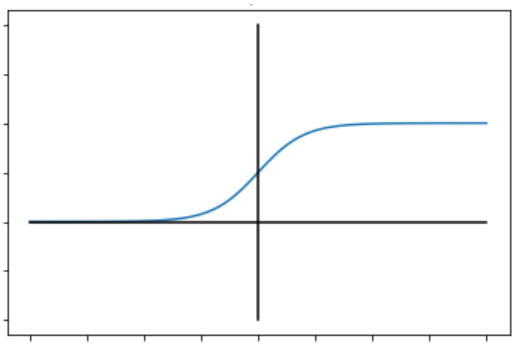


Figure 3

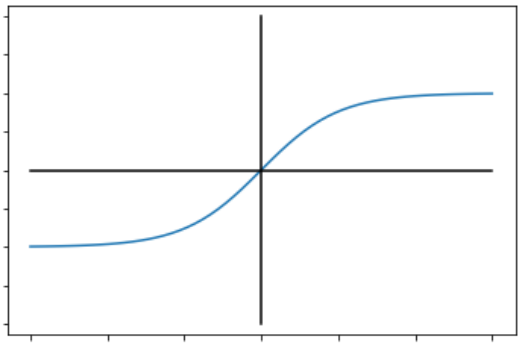
Some other activation functions that are commonly use are shown in Figure 4:



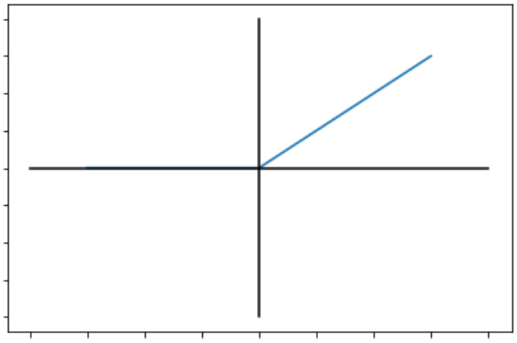
Linear: $f(z) = z$



Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$



Tanh: $f(z) = \frac{e^z - e^z}{e^z + e^{-z}}$



$ReLU$: $f(z) = \max\{0, z\}$

Figure 4

2.3 Multilayer neural network

An artificial neural network can have multiple neurons divided into layers. In each layer all neurons have the same inputs scaled with different weights. Each neuron of a layer has input connections, a bias, an activation function and an output. The outputs of a layer of neurons form the output vector of the layer.

In a multilayer neural network, each layer of neurons is fully connected with its adjacent layer. More precisely, every node of one layer of the network is connected to every node in the adjacent forward layer and the inputs of a layer are the outputs of the previous layer. The first layer of the neural network is called input layer and the last layer is called the output layer. All the intermediate layers are called hidden layers.

We denote by S^{ℓ} the number of neurons in layer ℓ and by S^0 the number of inputs in the input layer. We will use superscript to denote the layer we are referring to. For the layer ℓ of the neural network we will denote the weight matrix by W^{ℓ}, the bias vector by b^{ℓ}, the net input by z^{ℓ}, the activation function by f^{ℓ} and the output of the layer by α^{ℓ}.

The weight matrix of the ℓ-th layer is:

$$W^{\ell} = \begin{bmatrix} w_{1,1}^{\ell} & w_{1,2}^{\ell} & \cdots & w_{1,S^{\ell-1}}^{\ell} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S^{\ell},1}^{\ell} & w_{S^{\ell},2}^{\ell} & \cdots & w_{S^{\ell},S^{\ell-1}}^{\ell} \end{bmatrix}$$

Where the weight $w_{i,j}^{\ell}$ corresponds to the connection of the j-th neuron of the $\ell - 1$ layer to the i-th neuron in the ℓ-th layer.

A representation of a fully connected neural network with 3 hidden layers is shown in Figure 5.

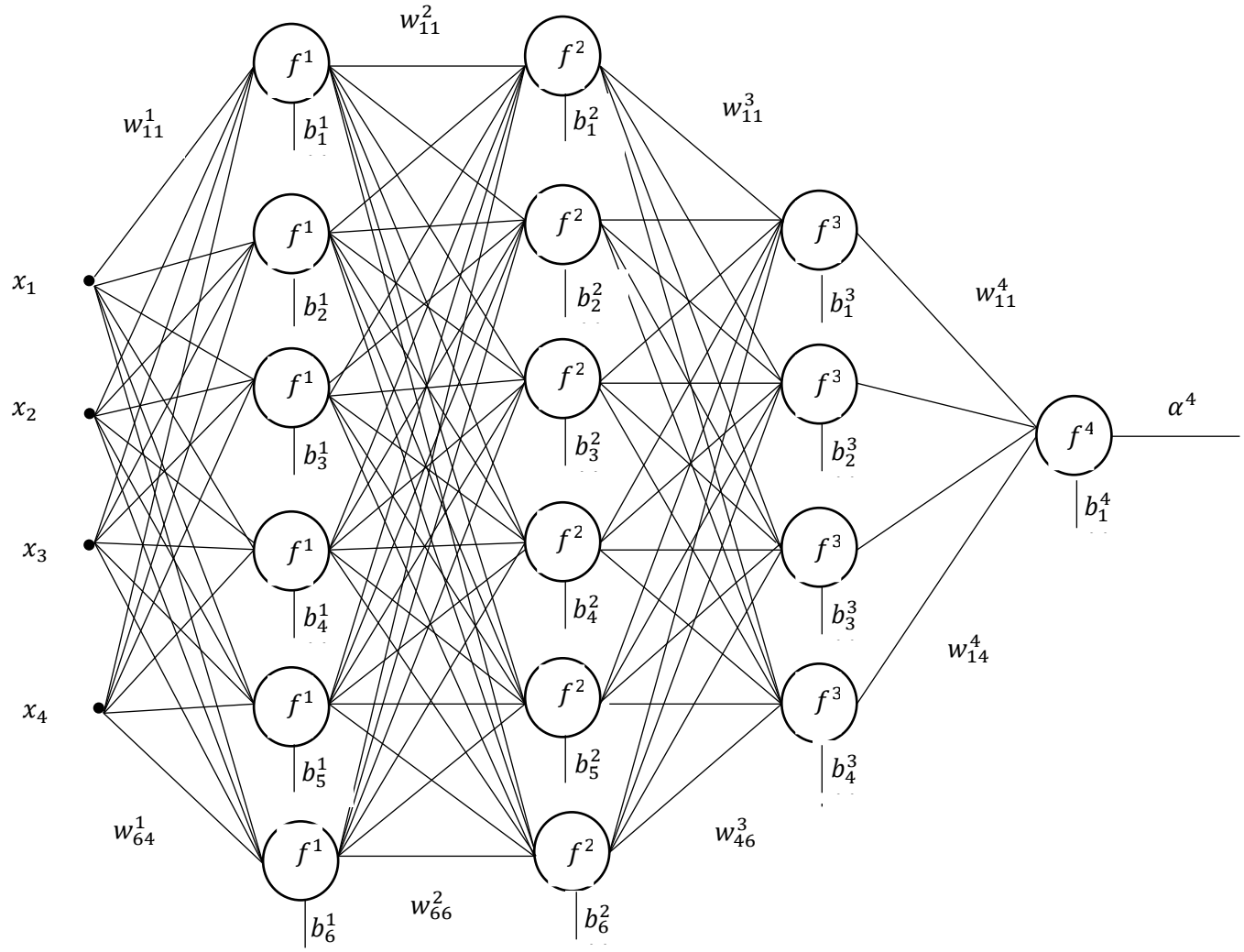


Figure 5

The neural network of Figure 5 has 4 neurons in the input layer, 6 neurons in the first hidden layer, 6 in the second hidden layer, 4 in the third hidden layer and 1 neuron in the output layer.

Here, in the first layer we have the vector $\alpha^1 = f(W^1 x + b^1)$, where $x = [x_1, x_2, x_3, x_4]^T$, $b^1 = [b_1^1, b_2^1, \dots, b_6^1]^T$ and

$$W^1 = \begin{bmatrix} w_{1,1}^1 & w_{1,2}^1 & \dots & w_{1,4}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{6,1}^1 & w_{6,2}^1 & \dots & w_{6,4}^1 \end{bmatrix},$$

in the second layer we have $\alpha^2 = f(W^2 \alpha^1 + b^2)$, where $b^2 = [b_1^2, b_2^2, \dots, b_6^2]^T$ and

$$W^2 = \begin{bmatrix} w_{1,1}^2 & w_{1,2}^2 & \dots & w_{1,6}^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_{6,1}^2 & w_{6,2}^2 & \dots & w_{6,6}^2 \end{bmatrix},$$

in the third layer we have $\alpha^3 = f(W^3\alpha^2 + b^3)$, where $b^3 = [b_1^3, b_2^3, \dots, b_4^3]^T$ and

$$W^3 = \begin{bmatrix} w_{1,1}^3 & w_{1,2}^3 & \dots & w_{1,6}^3 \\ \vdots & \vdots & \ddots & \vdots \\ w_{4,1}^3 & w_{4,2}^3 & \dots & w_{4,6}^3 \end{bmatrix},$$

and in the output layer we have $\alpha^4 = f(W^4\alpha^3 + b^4)$, where $b^4 = b_1^4$ and

$$W^4 = \left[w_{1,1}^4, w_{1,2}^4, \dots, w_{1,4}^4\right].$$

We can see that the network is much more complicated and the number of our parameters, weights and biases are now 24+6+36+6+24+4+4+1 = 105

It is common to use the same activation function for all the hidden layers but it is not necessary. The purpose of the activation function in the hidden layers is to make our network more complex and the advantage is that it has higher flexibility and can approximate a wider class of functions. That's why it is common for the activation function of the hidden layers to be non-linear.

In the case where the activation function is linear, we can rewrite the output of the neural network as:

$$\begin{aligned} \alpha^4 &= f(W^4\alpha^3 + b^4) \\ &= f(W^4\, f(W^3\alpha^2 + b^3)\, + b^4) \\ &= f(W^4\, f(W^3 f(W^2\alpha^1 + b^2) + b^3) + b^4) \\ &= f(W^4\, f(W^3 f(W^2 f(W^1\, x + b^1) + b^2) + b^3) + b^4) \\ &= f(W'x + b') \end{aligned}$$

for some W' and b', since f is a linear function.

This means that if the activation function of the hidden layers is linear, then the additional complexity of the multilayer neural network does not offer any advantage because all the hidden layers can be omitted and the neural network becomes shallow with only the input layer and the output layer with a matrix W' and bias b'.

Example

Assume that we have some fruits and vegetables and we want to classify them into apples, oranges bananas and carrots. Each fruit is been represented by 3 features: shape, texture, weight. These features are binary numbers, the shape is 1 if it is round and 0 if it is not, the texture is 1 if the surface is more smooth and 0 if it is not and the weight is 1 if the weight is more than 0.2 kilograms and 0 if it is not. A prototype apple will be represented by p_1, a prototype orange will be represented by p_2, a prototype banana will be represented by p_3, a prototype carrot will be represented by p_4, where p_1, p_2, p_3, p_4 are:

$$p_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \qquad p_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \qquad p_3 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \qquad p_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We will use the following neural network to decide which kind of fruit is represented by these features. The three features are going to be the input of the neural network. The neural network will have 2 output neurons representing the 4 fruits.

The activation function will be:

$$f(z) = \begin{cases} 1, & if\ z \geq 0 \\ 0, & otherwise \end{cases}$$

In order to determine the weight matrix and the biases of our network we need to determine the decision boundaries of each neuron such that they separate apples, oranges bananas and carrots into 4 categories. The goal is to have the decision boundary of the first neuron to separate the vectors p_1 and p_2 from p_3 and p_4 and the decision boundary of the second neuron to separate p_1 and p_3 from p_2 and p_4.

If the weight matrix is

$$W^1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

and the biases are

$$b^1 = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix},$$

then for every input $p_i,\ i = 1, \dots, 4$ the output of the neural network will be:

$$f(W^1 \cdot p_1 + b^1) = f\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \ (apple)$$

$$f(W^1 \cdot p_2 + b^1) = f\left(\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}1\\0\\1\end{bmatrix} + \begin{bmatrix}-0.5\\-0.5\end{bmatrix}\right) = \begin{bmatrix}1\\0\end{bmatrix} \ (orange)$$

$$f(W^1 \cdot p_3 + b^1) = f\left(\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}0\\1\\1\end{bmatrix} + \begin{bmatrix}-0.5\\-0.5\end{bmatrix}\right) = \begin{bmatrix}0\\1\end{bmatrix} \ (banana)$$

$$f(W^1 \cdot p_4 + b^1) = f\left(\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}0\\0\\0\end{bmatrix} + \begin{bmatrix}-0.5\\-0.5\end{bmatrix}\right) = \begin{bmatrix}0\\0\end{bmatrix} \ (carrot)$$

We can see that it categorizes the inputs as follows: if the output is $\begin{bmatrix}1\\1\end{bmatrix}$ then it is categorized as apple, if the output is $\begin{bmatrix}1\\0\end{bmatrix}$ then it is categorized as orange, if the output is $\begin{bmatrix}0\\1\end{bmatrix}$ then it is categorized as banana, if the output is $\begin{bmatrix}0\\0\end{bmatrix}$ then it is categorized as carrot.

Therefore, the neural network classifies perfectly the prototype apples, oranges, bananas and carrots.

In the case that the input of the neural network is not one of the prototype descriptions given in p_1, p_2, p_3, p_4 then the input will be classified in the category of the prototype that is closer to it in Euclidean distance.

2.4 Learning Techniques

Leaning is a very important procedure of a neural network since it is responsible for adjusting the parameters of the network (the weights and the biases) so that for every input we get the desired output. There are many learning techniques. The main categories of learning techniques are: supervised learning, unsupervised learning and reinforcement learning.

In **Supervised learning,** we are given a set of training examples

$$(X_1, Y_1), (X_2, Y_2), \dots, (X_n, Y_n)$$

where X_i is the input of the network and Y_i is the desired output of the particular input.

The procedure is to adjust the parameters of the network through some learning rule in order to get a network output as close as possible to the correct output. The goal of this procedure is to create a neural network that will be trained by a set

of given examples and will give us the correct output when the input of the network is an unseen data set.
This can be achieved by defining a cost function, which measures how close is the network's output to the correct output, and try to minimize it.

A commonly used cost function is:

$$C(w,b) = \frac{1}{2n}\sum_{i}\left\|Y_i - a^{i,L}\right\|,$$

where $a^L$is the output of the neural network on input X_i and Y_i is the desired output of the network given the input is X_i.

Another commonly used cost function is the quadratic cost function:

$$C(w,b) = \frac{1}{2n}\sum_{i}\left\|Y_i - a^{i,L}\right\|^2.$$

The cost function can be viewed as another parameter that the designer of the network can choose.

In **Unsupervised learning,** we are given a set of data whose output is unknown. The goal of the network is to search for input patterns and classify them correctly.

In **Reinforcement learning,** an agent takes actions in an environment and depending on the current state (the network's input) and action (the network's output), he receives a reward. The goal for the agent is to learn an optimal policy that maximizes the expected long-term rewards.

Gradient Descent

The learning technique that we will focus on in this thesis is supervised learning. As we mentioned, the closer the network output is to the correct output the smaller the cost function gets. So, it is normal to have a learning rule that tries to minimize the cost function with respect to the weights and biases parameters. A common optimization method used for the minimization is gradient descent

Gradient descent is an iterative algorithm based on the gradient of the function we want to minimize.

The derivative of a function $f(x)$: $\mathbb{R} \to \mathbb{R}$ at a point x_t, gives the slope of the function at the point x and consequently it gives us a direction of how the function is going to change if we make small change to x_t. The gradient of a function

$f(x)$: $\mathbb{R}^n \to \mathbb{R}$ at a point x_t is a vector and contains all the partial derivatives of this function at the point x_t.

The idea of gradient descent is that if we take a small step towards the opposite direction of the function's gradient to the point x_{t+1} then the value of the function should be decreased: $f(x_{t+1}) < f(x_t)$.

In our neural network the parameters that we want to adjust are the weights and biases and the function we want to minimize is the cost function.

Therefore, if we can compute the gradient of the cost function with respect to the weights and biases then starting with some initial values, we could have a guidance on whether we should increase or decrease them to minimize the cost function.

Thus, the learning rule that will minimize the cost function is:

$$w'_{ij} = w_{ij} - \alpha \cdot \frac{\partial C}{\partial w_{ij}}$$

$$b'_j = b_j - \alpha \cdot \frac{\partial C}{\partial b_j}$$

Where α is called the learning rule and we can think of it as the step size that we are going to move in the opposite direction of the gradient.

There are some challenges when applying the gradient descent rule. One of them is the value that we have to choose for the step size α, which in the neural network terminology is referred to as the learning rule. The gradient only gives the slope of the function near the value x_t. If the learning rule α is large, then the new point x_{t+1} will be far away from the point x_t and the calculation of the gradient does not guarantee that the final move will still be towards the direction of the true minimum. On the other hand, if the learning rule is small then the procedure can take a very time until we reach a desired point. That is why we have to be careful when deciding the learning rule. It is common practice to train the model many times with different values of the learning rule α until we find a suitable learning rate. Also, it is common to change the learning rate as the training evolves.

Another challenge is the complexity of computing the gradient of the cost function with respect to all the weights and biases for every different training example.

The cost function is

$$C(w,b) = \frac{1}{2n} \sum_i \left\| Y_i - a^{i,L} \right\|^2,$$

which can be rewritten as:

$$C(w,b) = \frac{1}{n}\sum_i C_i,$$

where $C_i = \frac{\|Y_i - a^L\|^2}{2}$ is the cost for one individual training example.

In practice to compute the gradient ∇C we compute ∇C_i for every training example and then take the average to find $\nabla C = \sum_i \nabla C_i$.

In many cases the number of training examples is very large and it is inefficient to compute the gradient for every training example in each iteration of the gradient algorithm. The method of **stochastic gradient descent** can be used to speed up the process without giving up the accuracy of the algorithm. The stochastic gradient descent converges almost surely to a local minimum, when the step size decreases with an appropriate rate, and is subject to relatively mild assumptions [1].

In stochastic gradient descend, we estimate the gradient ∇C by choosing a small random sample of our training examples, called mini-batch, compute the ∇C_i only for this sample and then average them. In this way, we have a good estimate of the true gradient ∇C, which helps significantly to speed up the learning process.

In mathematical terms this is expressed as:

$$w'_{kj} = w_{kj} - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial C_i}{\partial w_{kj}}$$

$$b'_j = b_j - \frac{\alpha}{m}\sum_{i=1}^{m}\frac{\partial C_i}{\partial b_j} \quad .$$

One other challenge when applying the gradient descent or the stochastic gradient descent is the computation of the gradient itself for every training example. Because the number of weights and biases is often quite large, a naive approach to calculate the gradient with respect to all these weights is impractical.

For this reason, we use a very efficient algorithm called **backpropagation** that can efficiently compute the gradient of the cost function.

Despite that the computation of the gradient of function is theoretically easy to compute, its computational evaluation is usually inefficient. The backpropagation algorithm solves this problem, by introducing an efficient procedure to achieve it.

Before we dive into the backpropagation algorithm, we define the quantity δ_j^ℓ, which we will call the error in the j-th neuron of the ℓ-th layer:

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

where C is the cost function, z is the net input $z = w^T\alpha^{\ell-1} + b,$ and $\alpha^{\ell-1}$ is the output of the neurons of the $\ell - 1$ layer.

Recall that $\alpha^\ell = f(z^\ell) = f(w^T\alpha^{\ell-1} + b)$.

Backpropagation gives us a way to compute the error δ_j^ℓ and then use it to compute the values $\frac{\partial C}{\partial w_{kj}^\ell}$ and $\frac{\partial C}{\partial b_j^\ell}$ that we are interested in.

The most common method used in backpropagation is the chain rule method from multivariable calculus.

The output error is: $\delta_j^L = \frac{\partial C}{\partial z_j^L}$

Applying the chain rule we get:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial \alpha_k^L} \cdot \frac{\partial \alpha_k^L}{\partial z_j^L}$$

However, we have that:

$$\alpha_k^L = f(z_k^L),$$

therefore,

$$\delta_j^L = \frac{\partial C}{\partial \alpha_j^L} \cdot f'(z_j^L).$$

If we use the cost function $C = \frac{\|Y - a^L\|^2}{2},$ then $\frac{\partial C}{\partial \alpha_j^L} = |Y - a_j^L|$, and

$$\delta_j^L = |Y - a_j^L| \cdot f'(z_j^L).$$

For an arbitrary ℓ:

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} = \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \cdot \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} \cdot \delta_k^{\ell+1}$$

However, we have that

$$z_k^{\ell+1} = \sum_j w_{kj}^{\ell+1} \cdot \alpha_j^\ell + b_k^{\ell+1} = \sum_j w_{kj}^{\ell+1} f(z_j^\ell) + b_k^{\ell+1}.$$

Hence,

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

We thus obtain:

$$\delta_j^\ell = \sum_k w_{kj}^{\ell+1}\, \delta_k^{\ell+1} f'\left(z_j^\ell\right).$$

Now that we have computed the error δ_j^ℓ for every layer and neuron, we can use it to compute the desired quantities $\frac{\partial C}{\partial w_{kj}^\ell}, \frac{\partial C}{\partial b_j^\ell}$.

We know that:

$$z_j^\ell = \sum_k w_{jk}^\ell\, \alpha_k^{\ell-1} + b_j^\ell$$

Hence,

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

and

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

Summing up, the four fundamental equations for backpropagation are:

$$\delta^L = \nabla_\alpha C \odot f'(z^L)$$

$$\delta^\ell = \left(\left(w^{\ell+1}\right)^T \cdot \delta^{\ell+1}\right) \odot f'\left(z^\ell\right)$$

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

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

where the $\odot$ is the Hadamard matrix product, i.e, the product is taken elementwise.

Now that we have written the equations in the above form, we can easily see how the backpropagation algorithm will work.

Given a training example $(X_i,\ Y_i)$,
Step 1: We perform a feedforward pass, meaning that the input X_i will pass through the neural network to give an output α^L.

Step 2: We compute the output error: δ_j^L

Step 3: We backpropagate the error and compute the quantities δ_j^ℓ for all ℓ and j.

Step 4: The gradient of the cost function is given simply by computing: $\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b}$

This algorithm is very efficient because it only has to pass through the network twice, one for its feedforward computations and once to backpropagate the errors.

Summing up, to train our neural network we will use stochastic descent with the help of the backpropagation algorithm to compute the gradients.

Stochastic gradient descent and backpropagation are very commonly used algorithms in practice, despite that they are known for many years. Of course, there are also some other algorithms used for training the parameters of the neural network. Some of them are variations of the stochastic gradient descent and some of them have different computation complexity. One variation of the stochastic gradient descent that we will also use in our analysis is called Adam.

Adam Algorithm

Adaptive Moment Estimate (Adam) algorithm [6] is a little more complicated than the one-line stochastic gradient descent but empirically we find that it converges more quickly than stochastic gradient descent.

In each iteration in order to adjust the parameters it uses both the average of the first moment estimation (mean) and the average of the second moment estimation of the gradient.

In each iteration t, it computes the quantities m_t, and v_t respectively:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t \odot g_t \,,$$

where g_t is the gradient of the cost function C with respect to the weights and biases, $\beta_1, \beta_2 \in [0,1)$ are hyper-parameters.
Initially, m_0 and v_0 are vectors of zeros.

The quantities m_t, v_t calculated above are biased towards zero, since their initial values is a vector of zeros. Therefore, we will use the biased-corrected estimates $\widehat{m_t}$ and $\widehat{v_t}$:

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

The last step is to update the parameters w, b :

$$w_t = w_{t-1} - \alpha \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$$

$$b_t = b_{t-1} - \alpha \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon},$$

where α is the learning rate, ϵ is a hyperparameter.

2.5 Improving the performance of Neural Networks

Up until now, we have discussed the main components of a neural network as well as some learning techniques to train the network.

In this chapter we consider again the main components of the neural network in order to understand better why we use each one of them and mention some ways to increase performance.

First of all, the most important element in the process of training a neural network is the training data. When training a neural network, we must "feed" the neural network with the training data that we have available until is fully trained and can perform optimally with new data points. The first problem that arises is the fact that we don't know exactly when the neural network is fully trained and how well it can perform to unseen data. Also, there is the problem of **overfitting**. If we train the network too much, then it starts to learn the training data exclusively and becomes incapable of generalizing to new input data.

One way to tackle this problem is to estimate how well our network is being trained to unseen data. To do this, before we start the training process, we can divide the

training data to two different data sets. We use the first one to train the neural network and the second one to test its performance.

Although this is a good method, it has a significant drawback. If we change the hyperparameters many times based on the performance of the test data, then there is a chance that we will **overfit** the neural network to the test data as well. To overcome this problem, we can divide the data into three categories: the training data, the validation data and the test data.

We use the training data to train the model and the validation data to tune the parameters for better performance.

Lastly, only when we estimate that the neural network is sufficiently trained, we test it with the test data to measure its actual performance.

Some other methods that are widely used and have been seen in practice to help with the overfitting problem, are called regularization techniques. One of the most used technique is known as weight decay or L2 regularization [9].

The idea is to add an extra term to the cost function that will penalize the weights by some factor λ. λ is also a hyperparameter, that we must decide its value before the training begins. The idea of penalizing the weights is that we don't want to have large values of weights unless they make a significant reduction to the cost function.

This comes from the idea that when some connections have large values of weights, then these connections have more influence to the neural network than others. Hence, the behavior of the neural network may change significantly if we make some small changes to the input data.

We note that by changing the cost function to the new form:

$$C(w,b) = C_i + \frac{\lambda}{2n} \cdot \sum_w w^2$$

we also make a small modification to the gradient descent method.

The only change is in the partial derivative with respect to the weight:

$$\frac{\partial C}{\partial w} = \frac{\partial C_i}{\partial w} + \frac{\lambda w}{n}$$

Therefore, the gradient descent rule becomes:

$$w' = w - \alpha \cdot \frac{\partial C_i}{\partial w} - w \cdot \frac{\alpha\lambda}{n},$$

where $\frac{\partial C_i}{\partial w}$ is the same as the one we found from backpropagation.

We next focus on the hyper-parameters of the neural network.

First of all, we must specify the parameters of the network. The network has two kind of parameters that we want to find their optimal value. In the first category, there are the weights and biases which, as we have mentioned, will be changing through some learning rule until they reach their optimal values. In the second category, there are all the other parameters that of the neural network. These are for example the number of layers, the size of each layer, the learning rate α of the stochastic gradient descent algorithm, the activation functions of every neuron, etc. These parameters are called hyper-parameters.

The hyper-parameters have to be assigned before the neural network begins its training and they are usually being decided by the designer of the neural network through trial and error.

Luckily there are some guidelines to tune these hyper-parameters but they are only guidelines and we will always have to test different kind of combinations of these hyper-parameters until we find their optimal value.

Unfortunately, there can't be a universal neural network that can solve every problem and so we have to tune our neural network every time we face a different problem. This is known as the "No free lunch theorem" [13].

On the other hand, in [5] it has been proven that given enough neurons, a neural network can approximate any function.

Lastly, as we have seen, in order for the Stochastic Gradient Descent or Adam algorithm to work, we have to assign the weights and biases to some initial values before the neural network starts.

The initial values of the weights and biases is very important because for different initial values, given that everything else stays the same, we might end up to different local minimums.

Unfortunately, there is no general method that we can use to find the global minimum with ease. One thing that we can do is to train our neural network with random initialization to the weights and biases multiple times and keep the one where it performs best. Of course, as we mentioned before, we want the weights and biases to have relatively small values, so it is better if we initialize them as such. A common method is to initialize them with a Normal distribution with mean 0 and variance 1.

Several alternative techniques have been proposed to improve the performance of the neural network. Many of these are only empirically proven to improve performance.

3 Newsvendor Problem

A classical optimization problem in operational research is inventory optimization. In every period the newsvendor manager has to make a decision of how much quantity of a product he will order. His objective is to maximize his profits in a finite or infinite horizon.

However, there are some cases that the corresponding products are perishable. This means that they lose their value at the end of each period, so it is impractical to store them. One example of this kind of products is newspapers. Each day's newspapers are worthless for the next day, so the newsvendor has to make a new decision every day knowing the unsold newspapers of each day will become worthless. Influenced by the newsvendor decision this class of problems are called newsvendor problems. This is the type of problems we will try to analyze with some extensions.

The problem was first introduced by Francis Edgeworth in 1888, where he used the central limit theorem to determine the optimal decision.

Problem Definition

In the newsvendor problem the retailer makes an order at the beginning of the period and sells them during that period. The actual demand of the product is unknown to the retailer. It is considered to be stochastic and it is represented by a random variable X with density function $f(x)$ and cumulative distribution $F(x) = P(X \leq x)$. To model the objective function of the retailer we will assume that we have two kinds of costs at the end of each period depending on the order quantity and the actual demand. If at the end of a period we have unsold products then each unsold product has a holding cost. On the other hand, if the retailer runs out of products in the middle of the period then he is charged a shortage cost per unit for the potential profit of the unsatisfied demand.

The objective of the newsvendor is to find the optimal quantity Q that minimizes the cost function $C(Q)$ where:

$$C(Q) = E_D\left[c_p(D-Q)^+ + c_h(Q-D)^+\right] =$$

$$= c_p \int_0^{\infty} \max(0, x - Q)\, f(x)dx + c_h \int_0^{\infty} \max(0, Q - x)\, f(x)dx$$

$$= c_p \int_Q^{\infty} (x - Q)\, f(x)\ dx + c_h \int_0^{Q} (\mathrm{Q} - x)\, f(x)dx,$$

where c_p: is the shortage cost,
c_h: is the holding cost,
D is the actual demand and
$(D - Q)^+ = \max(0, D - Q)$.

If the distribution of the demand is known, then the optimal solution of $C(Q)$ is:

$$Q^* = F^{-1}\left(\frac{c_p}{c_p + c_h}\right)$$

However, in real world problems the actual distribution of the demand is rarely known and this is our main interest.

It is worth noting that the distribution of the demand may be independent from any parameters or as it is usually the case, it can depend on some external parameters such as weather conditions, the day of the week, the store location etc. In every period the newsvendor knows these external parameters and must decide his order quantity based on them.

A variation of the above problem is if we take into account the price that the retailer sells the products as well as the cost of the ordered quantity. Then the problem becomes a maximization problem where the objective of the retailer is to find the optimal order quantity such that his profits are maximized.

The expected profit is:

$$\Pi(Q) = E_D[p \cdot min(Q, D) - w \cdot\ Q]\,,$$

where p is the selling price and
w is the cost the retailer is buying the products.

From here we can derive a more realistic approach of the problem where the actual distribution of the demand depends on the selling price of the product. One example of this dependence is if $D \sim N(f(p),\ \sigma^2)$, i.e, the actual distribution of the demand

is normal with constant variance σ^2 , but with mean equal to some function of the price, e.g. $\mu(p) = max(A - dp, 0)$, where A and d are some constant unknown parameters

This extends to the problem, where the objective of the newsvendor is not only to find the optimal order quantity but also to decide the optimal price of which he will sell his products along with an order quantity.

Thus, the expected profit becomes:

$$\Pi(p, Q) = E_D[p \cdot min(Q, D) - w \cdot Q] .$$

To find the optimal solution to this problem we will consider the approach followed in [11] and [12], with $D \sim N(y(p),\ \sigma^2)$. We rewrite $D = y(p) + \varepsilon$ where ε $\sim N(0,\ \sigma^2)$ and Q as $Q = y(p) + z$.
They find the solution in 2 stages. They first maximize the order quantity as a function of the price and then maximize the price based on the optimal order quantity function.

The order quantity is given as

$$z^* = F^{-1}\left(\frac{p - w}{p}\right).$$

To find the optimal price we must solve the equation:

$$\frac{\partial \Pi(p)}{\partial p} = 0$$

where

$$\frac{\partial \Pi(p)}{\partial p} = y(p) + p \cdot y'(p) + \int_{-\infty}^{z^*} u \cdot f(u)du + \int_{z^*}^{\infty} z^* f(u)du - w \cdot y'(p)$$

$$= y(p) + p \cdot y'(p) + \int_{-\infty}^{z^*} u \cdot f(u)du + z^*(1 - F(z^*)) - w \cdot y'(p)$$

Where

$$\int_{-\infty}^{z^*} u \cdot f(u)du = \int_{-\infty}^{z^*} u \cdot \frac{1}{\sigma\sqrt{2}} \cdot exp\left(-\frac{u^2}{2\sigma^2}\right) du = -\frac{2\sigma}{\sqrt{2}} \cdot \exp\left(-\frac{z^{*2}}{2\sigma^2}\right)$$

Hence,

$$\frac{\partial \Pi(p)}{\partial p} = \ y(p) + p \cdot y'(p) - \frac{2\sigma}{\sqrt{2}} \cdot \exp\left(-\frac{z^{*2}}{2\sigma^2}\right) + z^*(1 - F(z^*)) - \ w \cdot y'(p)$$

We find the optimal values by solving these equations.

In the next chapter, we present some representative papers that use neural network models for the newsvendor problem, and in Chapter 5, we will also develop a neural network algorithm for the above joint price-quantity optimization problem.
The analysis is inspired by [10], which has already analyzed the first problem of optimal order quantity.

4 Neural Networks for the newsvendor problem

In this chapter we will analyze some approaches to the newsvendor problem and inventory optimization in general. Our main focus will be the two main neural networks that have been proposed by [10] and [14].

In the paper [10] the authors propose a neural network to solve variation of the first problem that we discussed in the previous chapter.

Instead of trying to find the optimal order quantity for one single product, they assume that the retailer has m products and he needs to find the optimal order quantity for each one of them. Also, for each day, they use p different features that affect the demand distribution.

In mathematical terms, given n training data

$$\{ (x_i^1, d_i^1) , \dots , (x_i^m, d_i^m) \}_{i=1}^n ,$$

where $x_i^j \in \mathbb{R}^p$ and $d_i^j \in \mathbb{R}$, for $i = 1, \dots , n$, $j = 1, \dots , m$ and
x_i^j represents the features from the i-th data point and the j-th product,
d_i^j is the actual demand from the i-th data point and the j-th product given the x_i^j features.
Hence the cost function that they minimize is:

$$C(w,b) = \sum_{i=1}^{n} (\frac{1}{n} (\sum_{j=1}^{m} c_h \left(a_i^{j,L} - d_i^j\right)^+ + c_p \left(d_i^j - a_i^{j,L}\right)^+)),$$

where $a_i^{j,L}$ is the output of the neural network on input x_i^j
and d_i^j is the desired output of the network given the input is x_i^j

In the paper they also use a quadratic loss function:

$$C(w,b) = \sum_{i=1}^{n} \left(\frac{1}{n} \left(\sum_{j=1}^{m} \left(c_h \left(a_i^{j,L} - d_i^j\right)^+ + c_p \left(d_i^j - a_i^{j,L}\right)^+ \right)^2 \right) \right).$$

They develop a neural network with the stochastic gradient descent learning rule and the backpropagation to compute the gradient of the cost functions.

It is worth noting that in their analysis they use neural networks with both 2 and 3 hidden layers with a number of neurons in each layer being selected at random based on the number of neurons in the previous layer. More precisely, denoting nn_k the number of neurons in the layer k, for the network with two layers they choose:

$$nn_2 \in [0.5 \cdot nn_1, \quad 3 \cdot nn_1]$$

$$nn_3 \in [0.5 \cdot nn_2, \quad nn_2]$$

$$nn_4 = 1\,.$$

They also use a regularization parameter that is drawn uniformly from $[10^{-2},\ 10^{-3}]$. Lastly, they use the sigmoid function for the hidden layers.

The authors show that the neural network with the quadratic cost function performs better than the neural network with the simple cost function, but both networks perform better than some other state-of-the-art approaches that solve the same problem.

In [14] the authors consider the same problem using a slight variation of the neural network in [10]

They suggest the same neural network as [10] but they also add another layer at the end of the neural network with the ReLU activation function.

They also use a single input neuron in the last hidden layer that represents the actual demand of the products.

The weights are fixed from the last hidden layer to the output as:

$$\begin{bmatrix} -c_p & c_p \\ c_h & -c_h \end{bmatrix}$$

A visual representation of the network is given in Figure 6.

This variation has the property that the output layer is the cost function itself that is minimized, since:

$$C\big(f(x_i, q)\big) = \begin{cases} c_h(a_i^L - d_i)^+, & if\ a_i^L \geq\ d_i \\ c_p(d_i - a_i^L)^+ & if\ d_i \geq\ a_i^L \end{cases}\ .$$

The authors argue that the proposed quadratic cost function of [10] brings worse results than the simple cost function.

Other papers in the literature that tackle these problems or other more general problems for supply chain management, try to estimate the demand using neural networks. One drawback in this approach is that they don't take into account the costs c_p and c_h for the minimization problem. Such papers are [2] and [7].

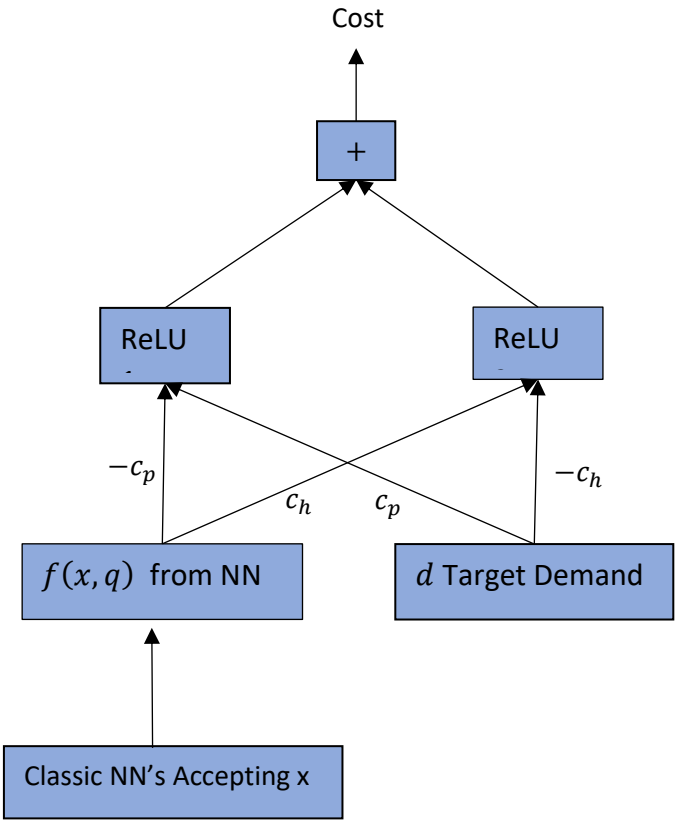

Cost
+
ReLU
ReLU
$-c_p$
c_h
c_p
$-c_h$
$f(x,q)$ from NN
d Target Demand
Classic NN's Accepting x

Figure 6

5 Applications

In this chapter we develop a neural network model for the problem of order quantity optimization as well as for the problem of price-order quantity optimization in a newsvendor inventory environment.

5.1 Order Quantity Optimization

We will first analyze our approach to finding the optimal order quantity under unknown demand distribution. Recall that in the problem the objective in the neural network is to find the optimal order quantity y in order to minimize the empirical cost function:

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n}\left(\; c_h\left(a_i'^{L} - d_i\right)^+ + c_p(d_i - a_i^L)^+ \; \right) = \frac{1}{n}\sum_{i=1}^{n} C_i(w,b) \qquad (1)$$

$$C_i(w,b) = \begin{cases} c_h(a_i^L - d_i), & if\ a_i^L \geq \ d_i \\ c_p(d_i - a_i^L) & if\ d_i \geq \ a_i^L \end{cases} \qquad (1.b)$$

where c_p is the shortage cost,
c_h is the holding cost,
d_i is the actual demand,
a_i^L is the output of the neural network on input x_i,
$\left(a_i'^{L} - d_i\right)^+ = \max\left(0, a_i'^{L} - d_i\right)$ and
w, b are the matrices of weight and bias vectors in the neural network.

We will also consider a variation with the quadratic cost function:

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n}\left(c_h(a_i^L - d_i)^+ + cp(d_i - a_i^L)^+\right)^2 = \frac{1}{n}\sum_{i=1}^{n} C_i(w,b), \qquad (2)$$

where $C_i(w,b) = \begin{cases} \left(c_h(a_i^L - d_i)\right)^2, & if\ a_i^L \geq \ d_i \\ \left(c_p(d_i - a_i^L)\right)^2 & if\ d_i \geq \ a_i^L \end{cases} \qquad (2.b)$

Our approach is based on the model in [10] with some variations.
First of all, we consider a simpler version of the problem where there are no feature values. This is done mainly for practical reasons and secondly because we will analyze an application with the price as a feature in our next application.

For simplicity we are going to find the optimal order quantity for only one product and we will not consider the problem with multiple products. The analysis can easily be generalized to the multi-product and the multi-featured problem.

In this problem we are given n training examples:

$$\{(X_1, D_1), \dots, (X_n, D_n)\},$$

where D_i is the actual demand generated by some distribution and
$X_1 = X_2 = \cdots = X_n = r$ where r is an arbitrary number between $[0,1]$

We use this convention because in this problem we don't include any features. This is equivalent as having the same feature in every training example.

In our analysis, we will use simulated data from known distributions and test different kinds of neural networks to measure their performance. We will first generate some data that come from the normal distribution with mean μ and variance σ^2 and test our neural network for different values of μ, σ^2, c_p, c_h.

We will use the same tuning of neural network for all the different values of μ, σ^2, c_p, c_h to test its performance. We will also be testing it for the two different kinds of cost functions.

It is worth noting that because these data are being generated by a known distribution, we can compare the performance of our neural networks with the theoretical optimal quantity $F^{-1}\left(\frac{c_p}{c_p+c_h}\right)$.

We will then generate some data using the exponential distribution and use a different neural network to test these data for different values of the mean $\frac{1}{\lambda}, c_p, c_h$. We will also be testing it for the two cost functions.

Training data generated by the normal distribution

In this framework, after a lot of training and tuning of the hyperparameters of our network we propose a neural network with 2 hidden layers. The first (input) layer of the network will have only 1 neuron, the first hidden layer will have 32 neurons, the second hidden layer will have 16 neurons and the output layer will have 1 neuron.

We found that a suitable activation function is the LeakyReLU [8] and we will use it for all of the neurons. LeakyReLU is a variation of the ReLU function that we stated in the beginning of this thesis.

The function for LeakyReLU is $f(x) = \begin{cases} z & , \quad if\ z \geq 0 \\ 0.001z & , \quad if\ z < 0 \end{cases}$

In order to train the weights and the biases, we will apply the Adam algorithm for all the neural networks in the first application. In the next application we will use the Stochastic Gradient Descent algorithm.
The parameters of the Adam used except for the learning rate, are the same for all the neural networks:

$b_1 = 0.9,\ b_2 = 0.999,\ e = 10^{-8}.$

The only value that will vary from one neural network to an other is the learning rate.
As we have stated, choosing the learning rate is a very important decision and it can influence the time that the neural network will converge. We found that starting with a relatively large value of learning rate and decreasing it as the number of iterations of the algorithm is increasing, helps the neural network converge faster and not oscillate around a solution.

When using the cost function (1), we will use the learning rates: 0.01 , 0.001, 0.0001 that will be changing when the number of iterations reaches 0, 100 and 250 respectively.
As for the second neural network we will use the learning rates: 0,1, 0.01, 0.001 , 0.0001 that will be changing when the number of iterations reaches 0, 6, 100 and 250 respectively.

We must also discuss the number of epochs that the neural network will be trained, i.e, the number of iterations the Adam algorithm will perform until it stops training the weights and biases. We found that a number of 500 iterations is enough to train this model.

We will use a mini-batch of length 10. The mini-batch is the size of the training data to estimate the full gradient.

Lastly, we should note that because in all our neural networks we start by initializing the weights and biases to some random numbers, we will only show the results of the neural network that had the best performance in a best-of-three runs.

Training data generated by the exponential distribution

In this framework, we propose a neural network with 3 hidden layers. The first (input) layer of the network will have only 1 neuron, the first hidden layer will have 16 neurons, the second hidden layer will have also 16 neurons, the third hidden layer will have 8 neurons and the output layer will have 1 neuron.

We will also use the LeakyReLU function for all of the neurons

As mentioned above, we will use the Adam algorithm to train the neural network for this application with parameters:

$b_1 = 0.9,\ b_2 = 0.999,\ e = 10^{-8}.$

As for the learning rate, when using the cost function (1), we will use the learning rates: 0.01 , 0.001, 0.0001 that will be changing when the number of iterations reaches 0, 100 and 250 respectively.
As for the second neural network we will use the learning rates: 0,1, 0.01, 0.001 , 0.0001 that will be changing when the number of iterations reaches 0, 6, 100 and 250 respectively.

Lastly, the number of epochs will again be 500 and the mini-batch size 10.

<u>Numerical Experiments</u>

We next generate simulated data to test our neural network and observe its performance under different distributions of the demand and different values of the parameters c_p, c_h

The first data set is very small. We only consider 5 data points for the first data set. The reason is so that we can see the performance of our neural network when faced with small data.

The results are shown in Table 1.

- NN1 represents the neural network that was trained with respect to the cost function (1),
- NN2 represents the neural network that was trained with respect to the Quadratic cost function (2), and
- Optimal represents the Optimal Theoretical value that we found by solving

$$Q^* = F^{-1}\left(\frac{c_p}{c_p + c_h}\right)$$

We only use different kind of (c_p, c_h) values that it holds $c_p \geq c_h$ because this is almost always true for real applications.

Table 1

Data = 5				
Distribution	(c_p, c_h)	Method of Estimation	Proposed Ordered Demand	Cost
N(50,1)	(3,3)	NN1	49.63	2.18
		NN2	49.74	2.11
		Optimal	50.0	1.96
	(6,3)	NN1	50.06	3.24
		NN2	49.94	3.10
		Optimal	50.43	3.68
	(11,3)	NN1	49.71	2.85
		NN2	50.03	2.91
		Optimal	50.79	2.00
	(20,3)	NN1	51.01	5.39
		NN2	50.76	6.92
		Optimal	51.12	4.69
N(50,6)	(3,3)	NN1	49.75	18.30
		NN2	49.64	18.36
		Optimal	50.0	1814
	(6,3)	NN1	45.64	62.61
		NN2	45.75	61.92
		Optimal	52.58	24.05
	(11,3)	NN1	52.24	29.56
		NN2	53.26	26.90
		Optimal	54.75	24.44
	(20,3)	NN1	59.47	39.79
		NN2	58.57	37.09
		Optimal	56.75	31.61
N(50,20)	(3,3)	NN1	33.91	62.09
		NN2	40.98	61.68
		Optimal	50.0	67,096
	(6,3)	NN1	90.60	137.24
		NN2	88.47	134.68
		Optimal	58.61	98.85
	(11,3)	NN1	77.37	62.61
		NN2	78.03	62.74
		Optimal	65.83	60.30
	(20,3)	NN1	75.61	85.96
		NN2	73.20	78.74
		Optimal	72.49	79.61
Exp(1/40)	(3,3)	NN1	41.07	146.99

		NN2	64.78	189.66
		Optimal	27.73	130.87
	(6,3)	NN1	75.38	150.88
		NN2	72.56	142.41
		Optimal	43.94	85.71
	(11,3)	NN1	40.89	122.36
		NN2	39.37	126.32
		Optimal	61.62	117.49
	(20,3)	NN1	91.80	209.24
		NN2	86.87	194.45
		Optimal	81.48	178.27

As we can see from Table 8, even though the data sets are very small, both neural networks seem to have very good performance, especially when the variance is small. We can see that in many cases, the order quantity that the neural networks suggest is very close to the theoretical optimal.
In some cases we can even see that the costs of some neural networks is smaller than the one from the theoretical optimal. This is because the data sets are very small and in some cases the variance is very large.

Another observation is that in most cases, the neural network with the quadratic cost function performs better that the first one.

Next, we consider a data set with 1000 points and test their performance
The result can be seen in Table 2

Table 2

Data = 1000				
Distribution	(c_p, c_h)	Method of Estimation	Proposed Ordered Demand	Cost
N(50,1)	(3,3)	NN1	50.02	1.16
		NN2	50.02	1.16
		Optimal	50.00	1.16
	(6,3)	NN1	50.44	1.62
		NN2	50.56	1.65
		Optimal	50.43	1.62
	(11,3)	NN1	50.76	2.15
		NN2	50.98	2.16
		Optimal	50.79	2.14
	(20,3)	NN1	51.21	2.33
		NN2	53.02	4.53
		Optimal	51.12	2.32

N(50,6)	(3,3)	NN1	49.99	7.27
		NN2	50.22	7.28
		Optimal	50.0	7.27
	(6,3)	NN1	52.37	10.14
		NN2	53.31	10.15
		Optimal	52.58	10.12
	(11,3)	NN1	54.29	12.67
		NN2	55.86	12.92
		Optimal	54.75	12.66
	(20,3)	NN1	57.50	14.27
		NN2	59.15	15.34
		Optimal	56.75	14.18
N(50,20)	(3,3)	NN1	49.75	24.51
		NN2	49.66	24.51
		Optimal	50.0	24.52
	(6,3)	NN1	58.63	35.16
		NN2	60.79	35.24
		Optimal	58.61	35.16
	(11,3)	NN1	64.27	40.01
		NN2	69.18	40.98
		Optimal	65.83	40.08
	(20,3)	NN1	81.82	247.33
		NN2	117.75	287.56
		Optimal	72.49	249.76
Exp(1/40)	(3,3)	NN1	29.16	88.07
		NN2	39.89	92.59
		Optimal	27.73	88.06
	(6,3)	NN1	41.01	130.93
		NN2	63.38	145.99
		Optimal	43.94	131.06
	(11,3)	NN1	63.35	172.13
		NN2	88.98	198.07
		Optimal	61.62	171.55
	(20,3)	NN1	81.58	249.33
		NN2	118.59	287.75
		Optimal	81.48	249.30

As we can see, the performance of our neural networks is even better than the previous examples. This is expected since they are now trained with more data points and their accuracy is improved.

One interesting thing that we observe is that in most cases, the neural network with the quadratic cost function now performs worse that the first one. This is consistent with the discussion in [14]

5.2 Price and Order Quantity Optimization

In this chapter we analyze our approach to the second proposed problem of the newsvendor.

To recall the problem, the objective is to find both optimal price value and the order quantity to maximize the expected profits:

$$\Pi(p, Q) = E_D[p \cdot min(Q, D) - w \cdot Q]$$

Where: p is the selling price,
Q is the order quantity,
D is the actual demand, and
w is the cost per unit that the retailer procures the products from the supplier

In this problem we are given n training examples $((X_1, D_i), \dots, (X_n, D_n))$

Where X_i is the feature price and
D_i is the actual demand and its distribution is now depending on the corresponding price.

We are going to tackle this problem in two stages, where in every stage we will build a different neural network.

In the first stage we will create a neural network that will perform a function approximation between the price and actual demand. In other words, our neural network will receive the price as input and its output would be the estimated demand of the product for that price.

In the next stage, we will create a neural network to solve the optimization problem. More precisely, with the help of the neural network from the first stage, this neural network's output will be the optimal price in order to maximize the profit function. Based on that proposed price, we will again use our first neural network to determine the optimal order quantity.

We will first analyze the first neural network and then we will analyze the second.

First Stage

In this stage, we want to approximate the relationship between the price and the actual demand of the product.
The actual demand can depend on the price in many different ways.

In our application, the actual demand will be a random variable following a Normal distribution with mean $\mu(p)$ and a constant variance σ^2

That is, the mean of the distribution of the actual demand will depend on the given price.

We will examine three different functions of $\mu(p)$:

1) $\mu(p) = (A_1 - B_1 p)^+$

2) $\mu(p) = \frac{A_2}{p+B_2}$

3) $\mu(p) = A_3\, e^{-B_3 p}$

where A_1, A_2, A_3 and B_1, B_2, B_3 are some constants.

We can see that the first function of $\mu(p)$ is linear whereas the other two are non-linear.
This means that the neural network for the first function will be simpler than the other two.

More specifically, for the **first** function, we propose a neural network with only one hidden layer. The first (input) layer of the network will have only 1 neuron, the hidden layer will have 8 neurons and the output layer will have 1 neuron.

We found that a suitable activation function is the LeakyReLU for the neurons in the hidden layer and the output neuron.

As for the cost function, we will use the quadratic cost function:

$$C(w,b) = \frac{1}{2n}\sum_i \left\| D_i - a^{i,L} \right\|^2$$

In order to train the weights and the biases, we will apply the Stochastic Gradient Descent algorithm for all the neural networks in this application.

As for the learning rate, we will use the learning rates: 0.8, 0.3, 0.09, 0.06 that will be changing when the number of iterations reaches 0, 300, 500 and 800 respectively. The number of epochs will be 1000 and the mini-batch size 10.

Lastly, we should note that we are applying the same best-of-three rule as in the first application.

For the **second** function, we propose a neural network with three hidden layers. The first (input) layer of the network will have only 1 neuron, the first hidden layer will have 64 neurons, the second hidden layer will have 32 neurons, the third hidden layer will have 16 neurons and the output layer will have 1 neuron.

We will also use the LeakyReLU function for all of the neurons in the hidden layers and the Sigmoid function for the output neuron.

The learning rates that we will use for the Stochastic Gradient Descent algorithm are: 0.15, 0.1, 0.09 that will be changing when the number of iterations reaches 0, 4 and 600

The number of epochs will be 1500 and the mini-batch size 10.

For the **third** function, we propose a neural network with two hidden layers. The first (input) layer of the network will have only 1 neuron, the first hidden layer will have 64 neurons, the second hidden layer will have 32 neurons and the output layer will have 1 neuron.

We will also use the LeakyReLU function for all of the neurons in the hidden layers and the Sigmoid function for the output neuron.
In order to use the Sigmoid function in the output neuron, we first rescaled all of our data points to lie in the interval [0,1]

The learning rates that we will use for the Stochastic Gradient Descent algorithm are: 0.65, 0.6, 0.28, 0.09 that will be changing when the number of iterations reaches 0, 4, 300 and 600

The number of epochs will again be 1000 and the mini-batch size 10.

Numerical Experiments

We next generate simulated data to test our three neural network and observe their performance.

The functions of $\mu(p)$ that we use are:

1) $\mu(p) = (1000 - 10p)^+$

2) $\mu(p) = \frac{10000}{p+100}$

3) $\mu(p) = 100e^{-\frac{p}{50}}$

The results are shown in Figures 7, 8 and 9.

In each Figure the first plot is the function generated by our neural networks,
the second plot is the actual function of $\mu(p)$
and the third plot is the training data points D_i that our neural network was trained.
In all three cases we generated the actual demand D_i from a Normal Distribution
with mean $\mu(p)$ and $\sigma^2 = 2$.

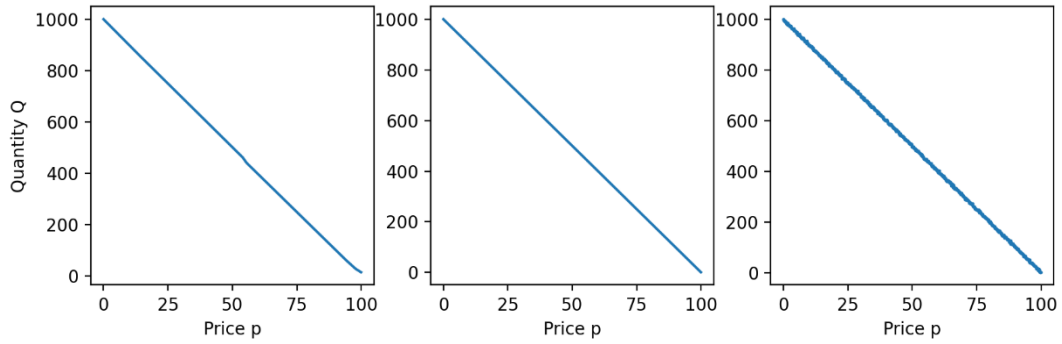


Figure 7

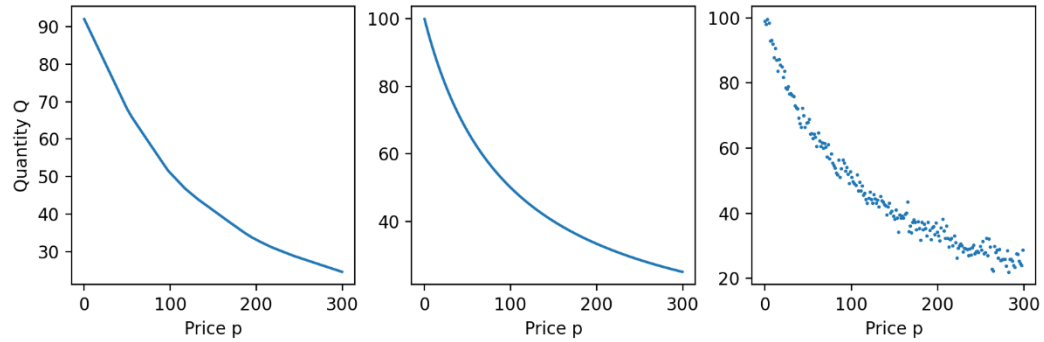


Figure 8

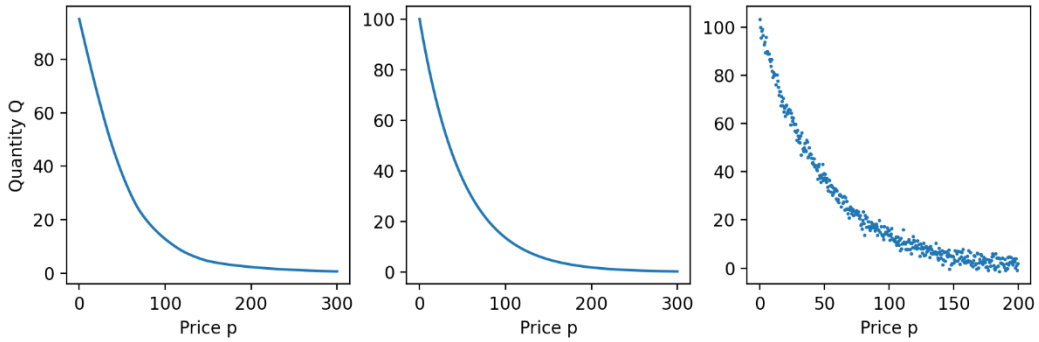


Figure 9

To test the performance of our neural networks, we generated a new data set for each case and we evaluated the difference between our neural networks and the data points. The mean of this cost for each case is:

1) $cost1 = 1.98$
2) $cost2 = 2.14$
3) $cost3 = 1.78$

Second Stage

Our goal is to find both optimal price value and the order quantity Q in order to maximize our expected profits.

In the first stage we created a neural network that approximates the function between the price and actual demand. That means that given a price p, we have an estimation about the optimal quantity that we want to order.

Hence, our problem now becomes finding only the optimal price value that maximizes the expected profits, because given that price, we can use our neural network from the first stage and find the optimal order quantity.

This neural network is similar to the neural network from the first application with

some necessary variations.
One variation in this neural network is in the training data set. We need a set of the actual demands in order to calculate the profit function and be able to maximize it. The problem in this application is that the actual demand depends on the price that we set every time. This means that the actual demand depends on our neural network's output and we can't know it in advance. To solve this problem, we must start with a random initial input and at the end of each iteration that our neural network proposes a price, we should generate the actual demand based on that particular price. With the help of the first neural network we can also produce the proposed demand for that particular price and then we proceed to calculate the profit cost and its derivative in order to use the Stochastic Gradient Descent algorithm and train the weights and biases to maximize the profit function.

The profit function that we want our neural network to maximize is :

$$\Pi(w,b) =\cdot a^L \cdot min(NN3(a^L),\ D(a^L)) - w \cdot\ NN3(a^L)\ \Rightarrow$$

$$\Pi(w,b) = \begin{cases} a^L \cdot\ \mathrm{D}(a^L) - w \cdot\ NN3(a^L), & if\ NN3(a^L) \geq\ \mathrm{D}(a^L) \\ a^L\ \cdot NN3(a^L)\ - w \cdot\ NN3(a^L) & if\ \mathrm{D}(a^L) \geq\ NN3(a^L) \end{cases}$$

$a^L$is the output of our neural network, which is the proposed selling price
$NN3(a^L)$ is the output of the neural network from the first stage given the price a^L, which is the proposed order quantity
D(a^L) is the actual demand given the price a^L

We must also compute the partial derivative of the profit function with respect to the output a^L because we will use it in the process of training our neural network:

For $NN3(a^L) \geq\ \mathrm{D}(a^L)$ we have that

$$\frac{\partial \Pi(a^L)}{\partial a^L} =\ \mathrm{D}(a^L) -\ w \cdot \frac{\partial NN3(a^L)}{\partial a^L}$$

The quantity $\frac{\partial NN3(a^L)}{\partial a^L}$ is the partial derivative of the neural network from the first stage with respect to its input price a^L.
To find the quantity $\frac{\partial NN3(a^L)}{\partial a^L}$, we must recall the four fundamental equations of the backpropagation algorithm:

$$\delta^L = \nabla_\alpha\ C\ \odot f'(z^L)$$

$$\delta^{\ell} = \left(\left(w^{\ell+1}\right)^{T} \cdot \delta^{\ell+1}\right) \odot f'\left(z^{\ell}\right)$$

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

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

We will use the same equations but instead of computing the quantities $\delta_j^{\ell} = \frac{\partial C}{\partial z_j^{\ell}}$, we will compute the quantities : $\delta_j^{\ell} = \frac{\partial NN3(a^L)}{\partial z_j^{\ell}}$

And instead of computing the quantities $\frac{\partial C}{\partial b_j^{\ell}} = \delta_j^{\ell}$, $\frac{\partial C}{\partial w_{jk}^{\ell}} = \alpha_k^{\ell-1}\, \delta_j^{\ell}$

we will compute $\frac{\partial NN3(a^L)}{\partial b_j^{\ell}} = \delta_j^{\ell}$, $\frac{\partial NN3(a^L)}{\partial w_{jk}^{\ell}} = \alpha_k^{\ell-1}\, \delta_j^{\ell}$

Continuing with the same way as the backpropagation algorithm, in the last step, we compute the quantities $\frac{\partial C}{\partial b_j^{1}} = \delta_j^{1}$, $\frac{\partial C}{\partial w_{jk}^{1}} = \alpha_k^{0}\, \delta_j^{1}$,

Where α_k^0 is actually the input feature, price, that we give to our neural network.

The algorithm of backpropagation stops here.

But if we continue for one more step, we compute the desired quantity:

$\frac{\partial NN3(a^L)}{\partial a^L} = \delta^0 = ((w^1)^T \cdot \delta^1) \odot f'(z^0)$, where $f'(z^0)$ is the derivative of the "activation" function of our input. In neural networks, we don't have an activation function in the input layer so we can just assume that the "activation" function is the linear function: $f(x) = x$

Hence, $\frac{\partial NN3(a^L)}{\partial a^L} = ((w^1)^T \cdot \delta^1)$

This means that we can compute the partial derivative of the profit function for $NN3(a^L) \geq \mathrm{D}(a^L)$:

$$\frac{\partial \Pi(a^L)}{\partial a^L} = \mathrm{D}(a^L) - w \cdot \frac{\partial NN3(a^L)}{\partial a^L}$$

Similarly, for $\mathrm{D}(a^L) \geq NN3(a^L)$:

$$\frac{\partial \Pi(a^L)}{\partial a^L} = NN3(a^L) + a^L \cdot \frac{\partial NN3(a^L)}{\partial a^L} - w \cdot \frac{\partial NN3(a^L)}{\partial a^L}$$

We are now ready to introduce the neural network and examine its performance.

We will use the same neural network for all the three cases in the first stage.

We will use a simple neural network with 1 hidden layer. The first (input) layer of our network will have only 1 neuron, the hidden layer will have 4 neurons and the output layer will have 1 neuron.

We will also use the LeakyReLU function for all of the neurons in the hidden layers and the Sigmoid for the output neuron.

In this problem we want to maximize the profit function, so we will use a variation of the Stochastic Gradient Descent algorithm called Stochastic Gradient Ascent. It is almost the same as the Stochastic Gradient Descent and its equations for training the weights and biases become:

$$w'_{ij} = w_{ij} + \alpha \cdot \frac{\partial C}{\partial w_{ij}}$$

$$b'_j = b_j + \alpha \cdot \frac{\partial C}{\partial b_j} \quad .$$

The learning rates that we will use are: 0.5, 0.1, 0.05, 0.001 that they will be changing when the number of iterations reach 0, 75, 200 and 500

The number of epochs will be 1000 and the mini-batch size 1.

The results can be seen in the table 3.

Table 3

$\mu(p)$		Price	Demand	Total Profits
$\mu(p) = (1000 - 10p)^+$	NN	64	356.91	15702.07
	Optimal	62	380.92	15913.05
$\mu(p) = \frac{10000}{p + 100}$	NN	p>>	Q <<	
	Optimal	p>>	Q<<	
$\mu(p) = 100e^{-\frac{p}{50}}$	NN	89	16.03	743.79
	Optimal	95	15.36	750.00

We observe that in the first and third case, the neural networks give a very good estimation of the optimal order quantity and the optimal price.

In the second case, we can see that we don't have an exact value in the values of the price and the quantity.
This is so because in the search of the optimal values, we found that the theoretical profit function keeps increasing as the price increases.
And although our neural network was trained by training data that had a maximum price of 300, we found out that even for prices larger than 10^{10}, the theoretical profit function was still increasing.

6 Conclusion

In this thesis we presented the basic models of artificial neural networks, their architecture, the learning techniques and their applications in optimization problems. We then emphasized on the applications of neural networks in the newsvendor problem and variations.

We then created and trained neural networks to solve different variations of the newsvendor problem. In the first one the goal was to find the optimal order quantity in order to minimize the cost function and in the second one the goal is to find the optimal price value and the order quantity to maximize the expected profits. The numerical results for the two applications were very close to the theoretical optimal solutions and in the first application, when the training data set was larger, then the output of the neural network showed even better accuracy.

One possible extension to be considered is to modify the problem as an online optimization problem where we are given the training data one at a time. In our analysis, we were given n training data points to train our neural network. This approach is helpful when someone opens a new business and has no information about the actual demand of the products.

Another possible extension to be considered is when the actual demand is not known unless the order quantity is more than the true demand, i.e., if our proposed order quantity is less than the actual demand, then the only information that we get is that all the products were sold, and we get no information about how many products we could have potentially sold.

7 References

[1] L. Bottou, "Online Algorithms and Stochastic Approximations," in *Online Learning and Neural Networks*, D. Saad, Ed., Cambridge, UK, Cambridge University Press, 1998.

[2] R. Carbonneau, K. Laframboise and R. Vahidov, "Application of machine learning techniques for supply chain demand forecasting," *European Journal of Operational Research,* vol. 184, no. 3, pp. 1140-1154, 2008.

[3] H. B. Demuth, M. H. Beale, O. De Jess and M. T. Hagan, Neural Network Design, 2 ed., Stillwater, OK, USA: Martin Hagan, 2014.

[4] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[5] K. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks,* vol. 2, no. 5, pp. 459-366, 1989.

[6] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization,* arXiv:1412.6980.

[7] A. Kochak and S. Sharma, "Demand Forecasting Using Neural Network for supply chain management," *International Journal of Mechanical Engineering and Robotics Research,* vol. 4, no. 1, pp. 96-104, 2015.

[8] A. L. Maas, A. Y. Hannun and A. Y. Ng, "Rectifier Nonlinearities Improve Neural NetworkAcoustic Models," in *ICML*, 2013.

[9] M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.

[10] A. Oroojlooy jadid, S. Lawrence and M. Takáč, "Applying Deep Learning to the Newsvendor Problem," *IISE Transactions,* vol. 52, 2017.

[11] N. C. Petruzzi and M. Dada, "Pricing and the Newsvendor Problem: A Review with Extensions," *Operations Research,* vol. 47, no. 2, pp. 183-194, 1999.

[12] T. M. Whitin , "Inventory Control and Price Theory," *Management Science,* vol. 2, no. 1, pp. 61-68, 1955.

[13] D. H. Wolpert and W. G. Macreeady, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation,* vol. 1, no. 2, pp. 67-82, 1997.

[14] Y. Zhang and G. Junbin, *Assesing the performance of Deep Learning Algorithms for Newsvendor Problem,* arXiv:1706.02899, 2017.