UNIVERSITY OF WARWICK

Extracting Information from Facial Images Using Artificial Neural Networks

by

Ayman Boustati

A thesis submitted in partial fulfillment for the
degree of Masters of Mathematics, Operational Research, Statistics and
Economics

in the
Faculty of Science
Department of Statistics

May 2015

Declaration of Authorship

I, AYMAN BOUSTATI, declare that this dissertation titled, 'EXTRACTING INFORMATION FROM FACIAL IMAGES USING ARTIFICIAL NEURAL NETWORKS' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a taught degree at the University of Warwick.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: 04-05-2015

"Wisdom is not a product of schooling, but of the lifelong attempt to acquire it."

Albert Einstein

UNIVERSITY OF WARWICK

Abstract

Faculty of Science
Department of Statistics

Masters of Mathematics, Operational Research, Statistics and Economics

by Ayman Boustati

With the current explosion of "Big Data", there has been an increase in the usage of statistical techniques that are able to extract information from different types of data. These techniques are often labelled as *Statistical Learning Methods*, and have become ubiquitous, not only in scientific fields of study, but also in other disciples such as marketing, finance and economics [1]. One of the most widely used tools of statistical learning is artificial neural networks, a class of powerful machine learning tools that can cope with high dimensional and unstructured data such as facial images. This dissertation will focus on exploring the usage of artificial neural networks for extracting information from facial images. Hence, it will study five applications of artificial neural networks on facial images: *gender classification, age prediction, dimensionality-reduction, frontalisation and keypoints detection.* Moreover, it will provide some insights into the history and applications of machine learning and artificial neural networks, in addition to discussing some of the theories regarding these topics.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr Ben Graham for his support and guidance throughout the course of this project, and for sharing with me his vast experience in the fields of statistical learning and machine learning. I would also like to thank my tutor, Dr Dario Spano for his help and support throughout the progression of my degree. I would further like to extend my gratitude to all the professors and staff in the Department of Statistics for sharing their knowledge and making my degree such an enjoyable and rewarding experience. I am also grateful to my colleagues whom I have had a great pleasure in meeting and interacting with. Finally, I would like to thank my parents, Mamoun Boustati and Nazli Alouch, and my sister, Alma, for their support and encouragement during my years away from home.

Contents

List of Figures

List of Tables

To my parents:

Mamoun Boustati

and

Nazli Alouch

Chapter 1

Introduction

1.1 Statistical Learning

1.1.1 Overview

Statistical Learning encompasses tools and models that are designed to extract information from complex data. These tools are primarily used in problem settings where the objective is predictive rather than explanatory. Statistical learning is different from traditional statistical modelling, as it relies heavily on algorithmic modelling without making many assumptions on the generating mechanism of the data [2]. Hence, most statistical learning methods assume that the data was generated from a black-box and the aim is to find a function that can predict the response to this data from known observations. This means that the tools of statistical learning are primarily driven by the quality of their results; hence, they employ different validation criteria than the ones used in traditional statistical modelling .

1.1.2 History

According to James, G., *et al.* [1], the term ”statistical learning” is relatively new; however, some of the tools employed in statistical learning date back to as far as the beginning of the nineteenth century, when Legendre and Gauss came up with the *method of least squares*, introducing the early version of *linear regression.* Linear regression was incredibly useful for predicting continuous quantitative values; however, it fell short when the target values were qualitative, such as predicting whether it was going to rain or not the next day. Hence, this led Fisher to develop the *linear discriminant analysis* in 1936 and various authors like Pearl, R., *et. al.* [3] to propose *logistic regression* in

1940. In 1972, Nelder and Wedderburn introduced *generalised linear models* [4], a class of models that include linear regression and logistic regression as special cases. Most of the statistical learning techniques that were introduced before the 1980s were linear methods, because of the computational infeasibility of modelling non-linear relationships. However, the improvement in computing technology in the 1980s opened to door to exploring non-linear learning models. In 1984, Breiman, Friedman, Olshen and Stone introduced *classification and regression tree (CART)* models [5], and in 1986 Hastie and Tibshirani proposed *generalised additive models* extending generalised linear models to non-linear settings [6]. With the introduction of these new models and many others, statistical learning became an exciting new sub-field of statistics. Furthermore, the revival of computationally heavy statistical learning techniques, such as *artificial neural networks* in the late 1980s [7], and the introduction of new ones in the 1990s, such as *support vector machines* in 1992 [8], helped bridge the gap between statistics and computer science, introducing *machine learning* as a shared area of interest between the two fields. By the turn of the twenty-first century, statistical learning and its sister field, machine learning, became prominent academic fields, with many researchers from the top statistics and computer science departments around the world contributing in their development. With the huge advancements in computer technology and the rise of Big Data in the late 2000s and early 2010s, statistical learning and machine learning became widely used and crucially important in many industries and disciplines inside and outside of the scientific sphere.

1.2 Artificial Neural Networks

1.2.1 Overview

Artificial Neural Networks are a class of non-linear machine learning algorithms, that are inspired by the biological model of the neurons in the brain [9]. They are complex models that are capable of performing the two main supervised learning tasks, classification and regression. Artificial neural networks are capable of learning from different formats of data such as structured data, images, sounds, etc. Neural networks are particularly good at performing pattern recognition tasks; hence, most of their applications are in this area. One of the strengths of artificial neural networks, is that they do not make any assumptions about the input data, but try to learn the relationship between the inputs and outputs by example. This is a similar fashion to how the human brain learns. Hence, artificial neural networks are very useful in problem settings where the underlying relationship between the inputs and the outputs are unknown.

1.2.2 History

According to Bishop [9], the study of artificial neural networks, has been inspired by the study of its biological counterpart. In 1943, McCulloh and Pitts [10], proposed an "all-or-none" model for the activity of neurons, in which a single neuron takes in a weighted sum of inputs from other neurons, and it is then either activated by these inputs and it stays dormant. In 1949, Donald Hebb proposed a learning mechanism for neural networks, which can be summarised by his "neurons that fire together, wire together" principle [11]. The first artificial neural network was created by Minsky in 1951 while he was a student at Princeton University. This network was called the *Stochastic Neural Analog Reinforcement Calculator (SNARC)* and was based Hebb's theory. SNARC was a physical network created from 3000 vacuum tubes and constituted of 40 neurons [12]. In 1958, the first attempt to create an artificial neural network for pattern recognition was made by Rosenbalt at Cornell University. Rosenbalt tried to implement the techniques of neural networks for the purpose of character recognition, creating the *Mark I Perceptron* [13], a single layer network. However, despite being one of the pioneers of the study of artificial neural networks, Minsky, along with his colleague Papert proved that single layer networks like the Mark I Perceptron were very limited when it comes to pattern recognition [13] [14]. Hence, more complex networks were needed to perform this task. It was not until 1986, when the *back-propagation algorithm* was developed, that multi-layer networks were possible to implement. The general method of the back-propagation algorithm was first discovered in 1974 by Werbos [15]. However, in 1986, Rumelhart, Hinton and Williams [16] managed to independently rediscover the work of Werbos and were able to popularise a special case of his general method that became the version of the back-propagation algorithm that is used today. Nevertheless, in the late 1990s and early 2000s, interest in artificial neural networks began to fade in favour of other statistical learning algorithm, most notably, support vector machines. However, this trend changed in 2006 when *Deep Learning* came into prominence when Hinton came up with a working implementation of the idea of deep layered networks [17]; thus, sparking renewed interest in artificial neural networks.

1.2.3 Applications

Artificial neural networks have been used in a wide variety of applications. From engineering to finance and urban planning to medicine, the use of artificial neural networks is becoming more and more omnipresent. One of the most popular applications of neural networks is pattern recognition. To this end, LeCun and Battou [18] successfully used a *convolutional neural network* to recognise generic objects, such as animals, cars, plane,

etc. in cluttered scenes with different poses and lighting conditions. Furthermore, convolutional neural networks have also been used in the classification of YouTube videos into a large number of categories by Karpathy, *et. al.* [19].
In engineering, artificial neural networks are used to aid in the optimisation of different processes. For instance, Google's Jim Gao [20] developed an artificial neural network that models the performance of Google's data centres to aid in optimising the energy efficiency of these data centres.
Neural networks are also used in medical applications, especially, as an aid in the diagnosis of cancer. For example, neural networks can be used to analyse the results of CT scans of the lungs to help in the early detection of lung cancer [21].
With the recent progress in Deep Learning, neural networks are rapidly becoming an integral part in many modern scientific and industrial applications.

1.3 Facial Images

1.3.1 Overview

The human face contains a great deal of information about its subject, such as their identity, gender, age and emotional state. In some cases, it can even reveal some information about the subject's personality. Humans have the innate ability to analyse the facial appearances of their fellow humans, and extract some of this information. This ability is a remarkable feature in human intelligence and extending it to machines can be a big first step towards creating a sentient Artificial Intelligence, among many other useful applications.

1.3.2 Potential Applications

The ability to study and extract information from faces can enhance many systems that are in use today. For instance, it can revolutionise the fields of robotics and artificial intelligence. There are already some implementations of facial recognition and emotion detection in some robotic prototypes [22] [23]; hence, adding other facial analysis abilities can help in enhancing these concepts, making them more lifelike. Age prediction and gender classification systems can help in the automation of some aspects of market research; for instance, retail companies can utilise systems that analyse facial photographs of their customers to gain insight into their customers' demographics. Facial detection and recognition can considerably enhance biometric and security systems. In fact, some security systems currently employ facial recognition methods to identify unwanted individuals [24].

1.4 Outline

The aim of this dissertation is to explore some of the applications of artificial neural networks with regard to facial image processing and information extraction. Hence, it will start by developing some background theory on machine learning and artificial neural networks in Chapter 2. This is followed by a study of five applications of artificial neural networks on facial images: gender classification in Chapter 3, age prediction in Chapter 4, dimensionality reduction in Chapter 5, facial frontalisation in Chapter 6 and keypoints detection in Chapter 7.

Chapter 2

Theory

2.1 Introduction

2.1.1 Machine Learning

Machine Learning is an interdisciplinary field combining statistical learning and computer science. The aim of machine learning is to create machines that are able to learn and generalise from data without explicit programming.

2.1.2 Types of Learning Problems

In machine learning, there are two types of problems:

- **Unsupervised Learning** is learning from unlabelled data. Unsupervised learning aims to discover a hidden structure or relationship within the data. *Clustering* and the *Hidden Markov Models* are examples of unsupervised learning problems.
- **Supervised Learning** is learning from labelled data. For a pair of vectors (x, y), supervised learning aims to produce a function $f_\theta(x)$ such that for some parameter vector θ, $f_\theta(x) = y$ [25]. In other words, in supervised learning, the objective is to find a function that links the observation x to the label y.

This dissertation is primarily focused on supervised learning; hence, the theory that is covered in this chapter will only involve supervised learning.

2.1.3 Types of Supervised Learning

There are two main types of supervised learning problems:

- **Classification:** the task in *classification* problems is to assign an observation x to one of n classes $\mathcal{C}_i$ for $i \in \{1, \dots, n\}$.
- **Regression:** *regression* problems assign a value equal y to an observation x, where y is a continuous variable.

2.2 Simple Linear Regression

2.2.1 Formulation

A trivial example of supervised learning is *Simple Linear Regression.* Recall, for a predictor variable X and response variable Y, assuming an approximate linear relationship between the two, it is possible to predict Y from X by setting:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

In this model, β_0 and β_1 are constants, called the *parameters* of the model, and ϵ is an unknown *error* term, which could be modelled as a normal random variable with mean 0 and variance σ^2 (i.e. $N(0, \sigma^2)$). Hence, from the formulation of the model it is easy to note that:

$$\mathbb{E}[Y|X] = \beta_0 + \beta_1 X \tag{2.1}$$

(2.1) could be used to approximate the value of Y from the value of X.

2.2.2 Fitting the Model

In practice, the model parameters β_0 and β_1 are unknown; consequently, they have to be estimated from observed data. Consider n pairs of observations from X and Y:

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$$

Good approximations for β_0 and β_1 (denoted as $\hat{\beta}_0$ and $\hat{\beta}_1$ respectively) are ones that can produce estimates $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ of y_i that are as close as possible to the observed value of y_i for $i \in \{1, 2, \dots, n\}$. Geometrically, this is the line with intercept $\hat{\beta}_0$ and slope $\hat{\beta}_1$ that runs as close as possible to all n points. To do this, it is essential to define a measure of closeness between y_i and $\hat{y}_i$. The most common choice for such measure in linear regression is $(y_i - \hat{y}_i)^2$ which is the square distance between the observed value y_i and the predicted value $\hat{y}_i$. This can also be presented as e_i^2, the square of the residual $e_i = y_i - \hat{y}_i = y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i$. Summing the residuals for all n the observed points, will yield an expression that quantifies the amount of the overall deviation of the line from all of the n points [1]. This expression is called the *Residual Sum of Squares (RSS).*

$$RSS = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2 \tag{2.2}$$

Notice that the RSS is nothing but the square of the $L2$-norm of the vector of the residuals. To find the best values of $\hat{\beta}_0$ and $\hat{\beta}_1$ that minimise the RSS (2.2), all is needed is to solve the following minimisation problem:

$$\underset{\hat{\beta}_0,\hat{\beta}_1}{\arg\min} \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2$$

The problem can be easily solved analytically, to find the optimal values of $\hat{\beta}_0$ and $\hat{\beta}_1$:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \frac{\sum_{i=1}^{n} x_i}{n})(y_i - \frac{\sum_{i=1}^{n} y_i}{n})}{\sum_{i=1}^{n}(x_i - \frac{\sum_{i=1}^{n} x_i}{n})^2}$$

$$\hat{\beta}_0 = \frac{\sum_{i=1}^{n} y_i}{n} - \hat{\beta}_1 \frac{\sum_{i=1}^{n} x_i}{n}$$

2.2.3 Simple Linear Regression in a Machine Learning Setting

In a machine learning setting, the relationship $Y = \beta_0 + \beta_1 X$ is called a *hypothesis* and its associated function:

$$h_{\beta_0,\beta_1}(x) = \beta_0 + \beta_1 x$$

is called the *hypothesis function*, as it represents the hypothesis that X and Y are linearly related. Furthermore, the square distance is an example of a *cost function*, a function that quantifies the cost of the deviation from the hypothesis for each point. Hence, the RSS (2.2) represents the total cost of the deviation from the hypothesis for all n points. For simple linear, regression the cost function is:

$$c(\beta_0, \beta_1) = (y - h_{\beta_0,\beta_1}(x))^2$$

And the total cost is:

$$C(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - h_{\beta_0,\beta_1}(x_i))^2$$

Accordingly, the minimisation problem becomes:

$$\underset{\beta_0,\beta_1}{\arg\min}\; C(\beta_0, \beta_1)$$

with $\hat{\beta}_0$ and $\hat{\beta}_1$ as the solution.

2.3 Multivariate Linear Regression

2.3.1 Formulation

The idea of simple linear regression can be easily extended to incorporate more than one predictor variables. Given p predictors $X_1, X_2, \dots X_p$ and a single response variable Y. The *Multivariate Linear Regression* model takes the following form:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

Like in the simple linear regression model, $\epsilon \sim N(0, \sigma^2)$. Consequently,

$$\mathbb{E}[Y|X_1, X_2, \dots, X_p] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p \tag{2.3}$$

2.3.2 Multivariate Linear Regression in a Machine Learning Setting

For the multivariate linear regression model, the hypothesis is $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$, yielding a hypothesis function:

$$h_{\boldsymbol{\beta}}(\mathbf{x}) = \beta_0 + \beta_1 x_2 + \beta_1 x_2 + \cdots + \beta_p x_p$$

A cost function:

$$c(\boldsymbol{\beta}) = \big(y - \beta_0 - \sum_{j=1}^{p} \beta_j x_j\big)^2$$

And a total cost of:

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \big(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij}\big)^2$$

The cost function for multivariate linear regression is multivariate. Hence, attempting to solve for $\hat{\beta}_0, \hat{\beta}_1, \dots, \hat{\beta}_p$ analytically requires a great deal of mathematical prowess and is ultimately very impractical if p is large. Fortunately, there are other methods to find the solution to $\underset{\boldsymbol{\beta}}{\arg\min}\; C(\boldsymbol{\beta})$. One method involves converting the cost function to matrix form and using linear algebra to solve the minimisation problem. However, this method is not going to be discussed here as it is beyond the scope of the dissertation. Another method is to solve the minimisation problem numerically by applying an algorithm called *Gradient Descent*, which will be introduced in the next section (Section 2.4). Although computationally intensive, gradient descent is very efficient and produces very accurate solutions.

2.4 The Gradient Descent Algorithm

2.4.1 The Minimisation Problem

Consider a smooth and convex f:

$$f : \mathbb{R}^n \to \mathbb{R}$$

Finding the minimum of this function analytically requires solving a system of n equations that are possibly non-linear. This task is straight forward for small n; however, it gets increasingly difficult and inefficient as the number of variables n increases. A more efficient way to minimise f is to attempt to find the minimum numerically using an iterative algorithm called the gradient descent algorithm.

2.4.2 Notation

Before describing the gradient descent algorithm, it is beneficial to define some notation for the convenience of the reader:

- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is the vector of n variables; hence, $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$.
- $\nabla f(\mathbf{x}) = (\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \ldots, \frac{\delta f}{\delta x_n})$ is the gradient of the function f.
- $\tau \in \{0, 1, 2 \ldots\}$ is the current iteration number, starting from 0 in the initialisation phase and moving on to $1, 2, \ldots$
- $\mathbf{x}^{(\tau)}$ is the value of $\mathbf{x}$ in the τ^{th} iteration.
- α is the step size or the learning rate parameter.

2.4.3 The Algorithm

The idea of gradient descent is simple, starting from an initial point, every iteration the algorithm moves a short distance in the direction of the greatest rate of decrease of the function f [9]. This is represented algorithmically as follows:

1. Initialise $\mathbf{x}^0$.
2. For $\tau = 1, 2, \ldots$ do:

$$\mathbf{x}^{(\tau)} = \mathbf{x}^{(\tau-1)} - \alpha \nabla f(\mathbf{x}^{(\tau-1)}) \tag{2.4}$$

$$\text{Calculate} \quad f(\mathbf{x}^{(\tau)})$$

3. Stop when a certain criteria is met.

Hence, after the initialisation, in each iteration $\mathbf{x}$ is updated by moving a short distance scaled by α in the direction of the negative of the gradient, i.e. the direction of the steepest descent.

2.4.4 Stopping Criteria

The stopping criteria of the algorithm varies according to the nature of the problem and the accuracy of the results that is required. One natural choice for the stopping criteria is to stop when $||\nabla f(\mathbf{x}^{(\tau)})||_2 < \epsilon$, where ϵ is some threshold, typically a very small number. $||.||_2$ is the $L2$-norm. This stopping criteria refers to when the rate of the greatest decrease becomes very small causing only very small change or none whatsoever to the value of $f(\mathbf{x}^{(\tau)})$ in consequent iterations. Another possible choice for the stopping criteria is $||f(\mathbf{x}^{(\tau)}) - f(\mathbf{x}^{(\tau-1)})||_2 < \epsilon$. This means that the algorithm stops when the size of the change of $f(\mathbf{x}^{(\tau)})$ between consequent iterations is less that a specified threshold ϵ. It is even possible to not specify any stopping criteria at all, and keep the algorithm running as long as there is patience for it to run.

2.4.5 Convergence

The two necessary conditions for the convergence of the gradient descent algorithm to the minimum (or at least a local minimum), is for the function f to be convex and smooth, i.e. continuously differentiable (up to at least the second derivative). The convergence of gradient descent follows from the following theorem:
Theorem:
Let $f(\mathbf{x})$ be differentiable in $\mathbb{R}^n$, and let the gradient of $f(\mathbf{x})$ satisfy the Lipschitz condition:

$$||\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})||_2 \leq L||\mathbf{x} - \mathbf{y}||_2 \tag{2.5}$$

Also, let $f(\mathbf{x})$ be bounded below:

$$f(\mathbf{x}) \geq f^* > -\infty \tag{2.6}$$

And let α satisfy the condition:

$$0 < \alpha < \frac{2}{L} \tag{2.7}$$

Then, in (2.4), the gradient tends to zero:

$$\lim_{\tau \to \infty} \nabla f(\mathbf{x}^{(\tau)}) = 0$$

And the function monotonically decreases:

$$f(\mathbf{x}^{(\tau)}) \leq f(\mathbf{x}^{(\tau-1)})$$

The proof of this theorem can be found in Appendix A.
Although the theorem above suggests that the smoothness and convexity of the objective function are important conditions for the convergence of the gradient descent algorithm, in practice, the algorithm may converge even if these conditions are violated. However, in this case an absolute minimum is not guaranteed, but a ”very good” local minimum is attainable.

2.5 Logistic Regression

2.5.1 Motivation

Linear regression is very useful in predicting a continuous response. However, it is very limited when it comes to classification problems, as its output is not restricted to the interval $(0, 1)$, even if the observed values of the response are binary. This means that its output cannot be interpreted as a probability. An intuitive way to circumvent this, is to map the output of linear regression to the interval $(0, 1)$ [26]. This gives rise to Logistic Regression.

2.5.2 The Sigmoid Function

Consider the function g:

$$g : \mathbb{R} \to (0, 1)$$

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

This function is called the logistic sigmoid function, and has the property that its output always lies in the interval $(0, 1)$. Figure 2.1 shows the plot of the sigmoid function.

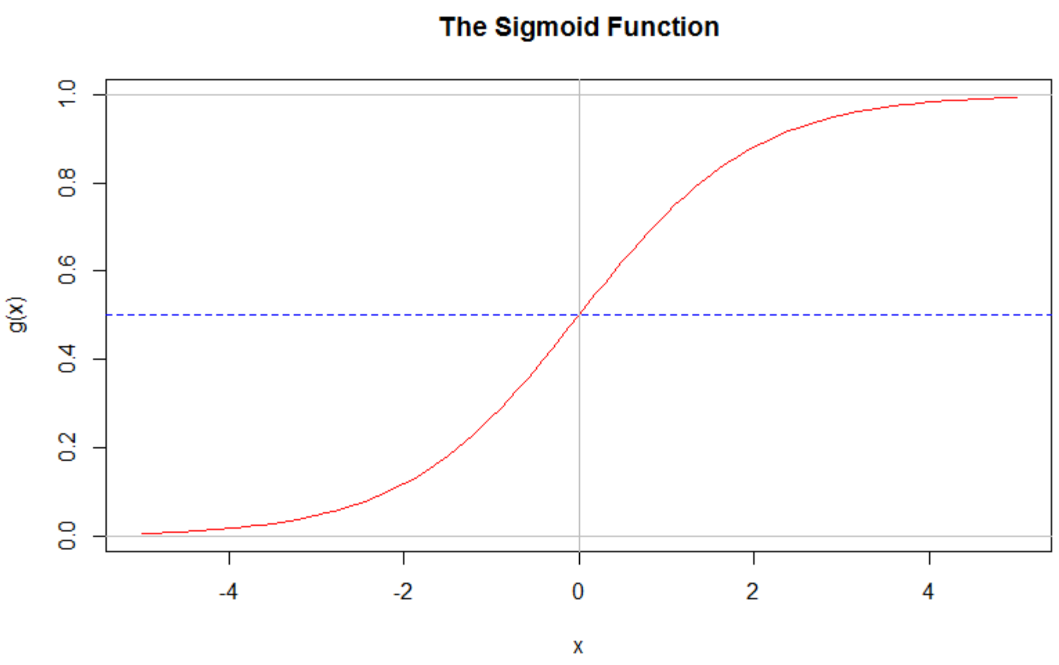


FIGURE 2.1: A graph of the sigmoid function.

2.5.3 Formulation

The sigmoid function can be used to map the output of linear regression to the interval $(0, 1)$, where one can perform two-class classification by specifying a probability threshold above which the subject will be allocated to a certain class, and below which the subject will be allocated to the other. Hence, for logistic regression with a feature vector $\mathbf{X} = (X_1, X_2, \dots, X_p)$, the hypothesis is [1]:

$$p(\mathbf{X}) = \mathbb{P}(Y = 1|\mathbf{X}) = g(\beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p) \tag{2.8}$$

In other words, the probability that the observation belongs to class $Y = 1$ is the sigmoid of the output of the linear regression on the feature vector $\mathbf{X}$.

2.5.4 The Machine Learning Formulation

The hypothesis (2.8) yields the following hypothesis function:

$$h_{\boldsymbol{\beta}}(\mathbf{x}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)}} \tag{2.9}$$

Note that, due to the nature of classification problems, the square distance between the hypothesis function $h_{\boldsymbol{\beta}}(\mathbf{x})$ and the label y cannot be used as a as cost function, because this will yield a non-convex function. An more suitable alternative to use in classification problems, is to score the cost of misclassification for each known observation. One can do this using the following function:

$$m : (0,1) \times \{0,1\} \to \mathbb{R}$$

$$m(h_{\boldsymbol{\beta}}(\mathbf{x}), y) = \begin{cases} -\log(h_{\boldsymbol{\beta}}(\mathbf{x})) & \text{if } y = 1 \\ -\log(1 - h_{\boldsymbol{\beta}}(\mathbf{x})) & \text{if } y = 0 \end{cases}$$

Another formulation of m is:

$$m(h_{\boldsymbol{\beta}}(\mathbf{x}), y) = -y\log(h_{\boldsymbol{\beta}}(\mathbf{x})) - (1-y)\log(1 - h_{\boldsymbol{\beta}}(\mathbf{x}))$$

This function is often referred to as the cross-entropy function. Hence, $m(h_{\boldsymbol{\beta}}(\mathbf{x}), y)$ can be used as a cost function.
Having defined a convex cost function, it is now possible to derive the total cost:

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} m(h_{\boldsymbol{\beta}}(\mathbf{x}), y)$$

To find the optimal parameter vector $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_p)$ for logistic regression, it is possible to solve the minimisation problem $\arg\min_{\boldsymbol{\beta}} C(\boldsymbol{\beta})$ using gradient descent (2.4).

2.6 Artificial Neural Networks

2.6.1 Motivation

Artificial Neural Networks are a family of machine learning models that can learn from high dimensional data. Artificial neural networks attempt to mimic the behaviour of the neurons in the brain. They are complex computational models that assume a non-linear relationship between their inputs and their outputs. A neural network consists of an input layer, zero or more hidden layers and an output layer. Each layer applies a function called an activation function, on a weighted subset of the neurons from another layer. If there are no feedback loops between the layers of the network, the network is called an *Feed-Forward Network*. However, if some layers in the network take inputs from consequent layers in the same network, the network is then called a *Recurrent Neural Network*. Note that this dissertation is not concerned with recurrent neural networks; hence, the theory presented in this chapter will only consider feed-forward networks. A network is said to be *fully-connected*, if all the neurons in each layer are connected to all the neurons in the next layer.

2.6.2 Logistic Regression Revisited

A good start to motivate the theory of neural networks, is to consider logistic regression. A logistic regression model of p features can be represented by the graph in Figure 2.2. The neural network representing logistic regression in the figure, takes $x_1, x_2, \ldots, x_p$ as inputs, in addition to the $+1$ term, called the *bias* term, which corresponds to the intercept term in logistic regression. These inputs form the first layer of the network, the *Input Layer*. The weights on the directed arrows leading to the *Output Layer* are the coefficients of logistic regression, which act as the parameters of the model. Hence, the inputs in the first layer are weighted then fed into the output layer, which applies a sigmoid activation function $g(z)$ on the sum of the weighted inputs. Hence, the hypothesis function for this neural network is exactly the same as the hypothesis function for logistic regression (2.9), introduced in Section 2.5.4. Furthermore, the cost function for this neural network is the cross-entropy function, again the same as the cost function for standard logistic regression. Hence, this leads to a total cost of:

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} m(h_{\boldsymbol{\beta}}(\mathbf{x}), y)$$

Where m is the cross-entropy function.

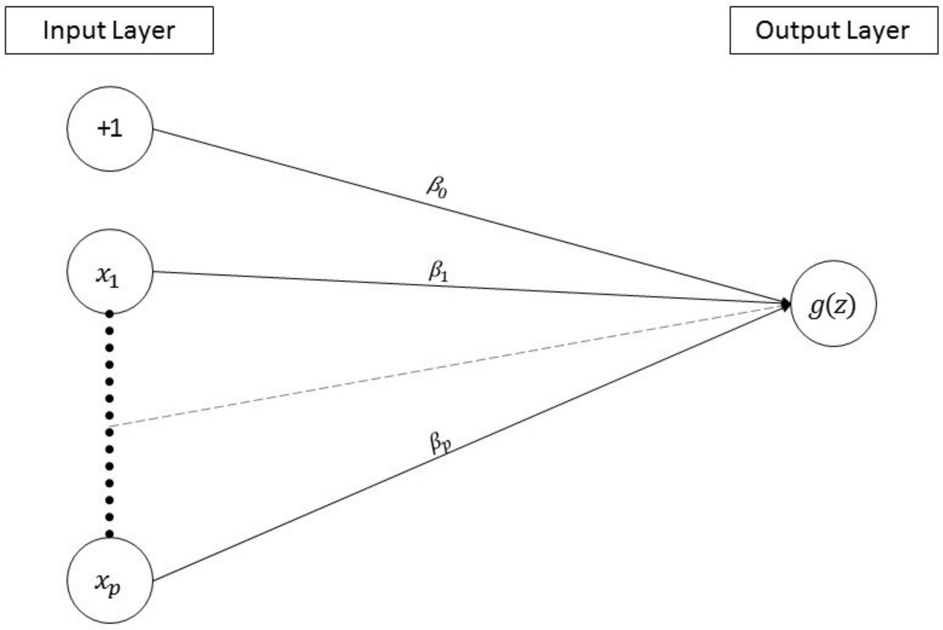


FIGURE 2.2: A neural network representation of logistic regression.

2.6.3 Notation

Before attempting to derive any results about artificial neural networks, it is beneficial to introduce some notation. Consider a neural network of $L+1$ layers each containing N_l neurons for $l \in \{0, 1, \dots, L\}$. This network takes p features as inputs; hence, $N_0 = p$ corresponding to the p inputs. The first hidden layer of this network contains N_1 neurons each taking as input the p neurons from the input layer, in addition to an extra bias unit corresponding to the intercept. In general, any layer l consists of N_l neurons, each taking $N_{l-1}+1$ inputs corresponding to the neurons in the previous layer plus a bias unit. The output layer of the network produces N_L outputs. For this neural network, the following are defined:

- $W_{ij}^{(l)}$ is the weight associated with the connection going from the i^{th} neuron in the layer $l-1$ to the j^{th} neuron in the layer l, for $i \in \{1, 2, \dots, N_{l-1}\}$, $j \in \{1, 2, \dots, N_l\}$ and $l \in \{1, 2, \dots, L\}$.
- $b_j^{(l)}$ is the weight associated with the bias unit going to the j^{th} neuron in the layer l, for $j \in \{1, 2, \dots, N_l\}$ and $l \in \{1, 2, \dots, L\}$.
- $f^{(l)}(z)$ is the activation function in the neurons of the layer l for $l \in \{1, 2, \dots, L\}$. Note that, the hypothesis function of a neural network is $h(z) = f^{(L)}(z)$.
- $z_j^{(l)} = \sum_{i=1}^{N_{l-1}} W_{ij}^{(l)} a_i^{(l)} + b_j^{(l)}$ is the weighted sum of the output of the neurons from layer $l-1$ plus the bias term, all going to the j^{th} neuron in the layer l, for $j \in \{1, 2, \dots, N_l\}$ and $l \in \{1, 2, \dots, L\}$. Note that, for the first hidden layer $z_j^{(1)} = \sum_{i=1}^{p} W_{ij}^{(1)} x_i + b_j^{(1)}$, where $x_1, x_2, \dots, x_p$ are the p input features.
- $a_j^{(l)} = f^{(l)}(z_j^{(l)})$ is the activation of $z_j^{(l)}$, i.e. the output of the j^{th} neuron in the layer l, for $j \in \{1, 2, \dots, N_l\}$ and $l \in \{1, 2, \dots, L\}$.

2.6.4 Forward-Propagation

From the definitions in Section 2.6.3, one can construct a set of equations that describes a general neural network as follows:

$$
\begin{aligned}
z_j^{(1)} &= \sum_{i=1}^{p} W_{ij}^{(1)} x_i + b_j^{(1)} && \text{for } j \in \{1, 2, \ldots, N_1\} \\
a_j^{(1)} &= f^{(1)}(z_j^{(1)}) && \text{for } j \in \{1, 2, \ldots, N_1\} \\
z_j^{(2)} &= \sum_{i=1}^{N_1} W_{ij}^{(2)} a_i^{(1)} + b_j^{(2)} && \text{for } j \in \{1, 2, \ldots, N_2\} \\
a_j^{(2)} &= f^{(2)}(z_j^{(2)}) && \text{for } j \in \{1, 2, \ldots, N_2\} \\
&\vdots \\
z_j^{(L)} &= \sum_{i=1}^{N_{L-1}} W_{ij}^{(L)} a_i^{(L-1)} + b_j^{(L)} && \text{for } j \in \{1, 2, \ldots, N_L\} \\
a_j^{(L)} &= h_j(z_j^{(L)}) = f^{(L)}(z_j^{(L)}) && \text{for } j \in \{1, 2, \ldots, N_L\}
\end{aligned}
$$

The equations above describe what is called the *forward propagation* process of the neural network. These equations define the relationship between the input features $x_1, x_2, \ldots, x_p$ and the outputs of the neural network $h_1, h_2, \ldots, h_{N_L}$. Figure 2.3, is an illustration of the structure of the forward propagation of a feed-forward network.

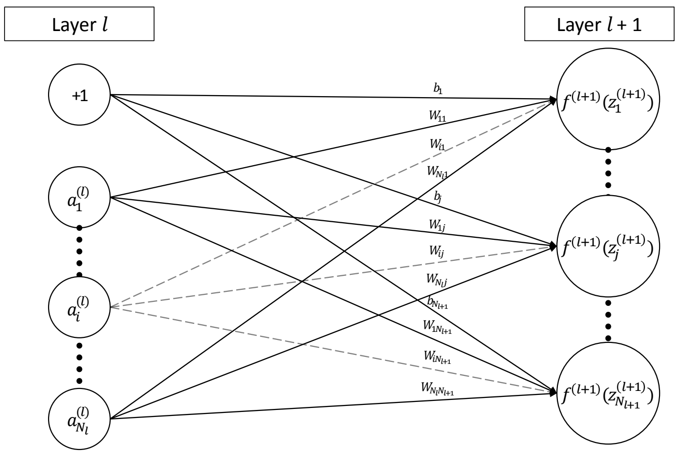


FIGURE 2.3: An illustration of two layers of a fully-connected network.

2.6.5 Another Formulation

It is possible to re-formulate the forward propagation equations using linear algebra to simplify the notation and to make the equations easier for manipulation. Hence, the following are defined:

- $\mathbf{W}^{(l)}$ is an $N_l \times N_{l-1}$ matrix representing the weights associated with all connections between layer $l-1$ and layer l.
- $\mathbf{b}^{(l)}$ is an $N_l \times 1$ column vector representing all the weights associated with the bias unit connecting to layer l.
- $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$ is an $N_l \times 1$ column vector representing the weighted sum of the activated neurons from the layer $l-1$ plus the bias term.
- $\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)})$ is an $N_l \times 1$ column vector representing the activation of the neurons in the layer l.

Consequently, it is possible to rewrite the forward propagation equations for the neural network as follows:

$$\begin{aligned}
\mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{a}^{(1)} &= f^{(1)}(\mathbf{z}^{(1)}) \\
\mathbf{z}^{(2)} &= \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \\
\mathbf{a}^{(2)} &= f^{(2)}(\mathbf{z}^{(2)}) \\
&\vdots \\
\mathbf{z}^{(L)} &= \mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)} \\
\mathbf{a}^{(L)} &= h_{\mathbf{W},\mathbf{b}}(\mathbf{z}^{(L)}) = f^{(L)}(\mathbf{z}^{(L)}) \qquad (2.10)
\end{aligned}$$

The equation (2.10) defines the hypothesis function for a neural network $h_{\mathbf{W},\mathbf{b}}(\mathbf{z}^{(L)}) = f^{(L)}(\mathbf{z}^{(L)})$, where $\mathbf{W}$ and $\mathbf{b}$ represent the parameters $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(L)}$ and $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \dots, \mathbf{b}^{(L)}$. From the hypothesis function, one can define a total cost for the neural network:

$$C(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^{n} c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i) \qquad (2.11)$$

Where $c(.)$ is a cost function, and $\mathbf{y}_i$ is the label vector for the $i^t h$ training example for $i \in \{1, 2, \dots, n\}$, where n is the number of training samples. To find the optimal choice for the parameters $\mathbf{W}$ and $\mathbf{b}$ the following minimisation problem must be solved:

$$\underset{\mathbf{W},\mathbf{b}}{\arg\min}\, C(\mathbf{W}, \mathbf{b})$$

This problem can be solved using gradient descent as before; however, due to the complexity of the cost function, it is not easy to obtain its partial derivatives for the gradient descent task. Fortunately, there exists an algorithm called the *Backward Propagation of Errors* that exploits the chain rule, in order to simplify the task of obtaining the derivatives.

2.6.6 Backward Propagation of Errors

The backward propagation of errors algorithm makes use of the chain rule to derive the partial derivatives of the cost function with respect to all the parameters. Let,

$$\boldsymbol{\delta}^{(l)} := \frac{\delta C}{\delta \mathbf{z}^{(l)}}$$

Hence,

$$\begin{aligned}
\boldsymbol{\delta}^{(l-1)} &= \frac{\delta C}{\delta \mathbf{z}^{(l-1)}} \\
&= \frac{\delta C}{\delta \mathbf{z}^{(l)}} \frac{\delta \mathbf{z}^{(l)}}{\delta \mathbf{z}^{(l-1)}} \qquad \text{by the chain rule} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})}{\delta \mathbf{z}^{(l-1)}} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta(\mathbf{W}^{(l)} f^{(l-1)}(\mathbf{z}^{(l-1)}) + \mathbf{b}^{(l)})}{\delta \mathbf{z}^{(l-1)}} \\
&= (\mathbf{W}^{(l)^T} \boldsymbol{\delta}^{(l)}) \circ f'^{(l-1)}(\mathbf{z}^{(l-1)})
\end{aligned} \tag{2.12}$$

Hence, the value of $\boldsymbol{\delta}^{(l-1)}$ can be calculated from the $\boldsymbol{\delta}^{(l)}$, for $l \in \{1, 2, \dots, L\}$. This means that by obtaining the value of $\boldsymbol{\delta}^{(L)}$, all other values $\boldsymbol{\delta}^{(l)}$ can be obtained.

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \frac{\delta C}{\delta \mathbf{z}^{(L)}} \\
&= \frac{\delta C}{\delta \mathbf{a}^{(L)}} \frac{\delta \mathbf{a}^{(L)}}{\delta \mathbf{z}^{(L)}} \\
&= \nabla_a C \circ f'^{(L)}(\mathbf{z}^{(L)})
\end{aligned} \tag{2.13}$$

Working backwards from the total cost function:

$$\begin{aligned}
\frac{\delta C}{\delta \mathbf{W}^{(l)}} &= \frac{\delta C}{\delta \mathbf{a}^{(l)}} \frac{\delta \mathbf{a}^{(l)}}{\delta \mathbf{W}^{(l)}} \\
&= \frac{\delta C}{\delta \mathbf{a}^{(l)}} \frac{\delta \mathbf{a}^{(l)}}{\delta \mathbf{z}^{(l)}} \frac{\delta \mathbf{z}^{(l)}}{\delta \mathbf{W}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta \mathbf{z}^{(l)}}{\delta \mathbf{W}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})}{\delta \mathbf{W}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \mathbf{a}^{(l-1)^T} \qquad (2.14)
\end{aligned}$$

Similarly,

$$\begin{aligned}
\frac{\delta C}{\delta \mathbf{b}^{(l)}} &= \frac{\delta C}{\delta \mathbf{a}^{(l)}} \frac{\delta \mathbf{a}^{(l)}}{\delta \mathbf{b}^{(l)}} \\
&= \frac{\delta C}{\delta \mathbf{a}^{(l)}} \frac{\delta \mathbf{a}^{(l)}}{\delta \mathbf{z}^{(l)}} \frac{\delta \mathbf{z}^{(l)}}{\delta \mathbf{b}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta \mathbf{z}^{(l)}}{\delta \mathbf{b}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \frac{\delta(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})}{\delta \mathbf{b}^{(l)}} \\
&= \boldsymbol{\delta}^{(l)} \qquad (2.15)
\end{aligned}$$

Hence, the back-propagation algorithm, specified by equations (2.12), (2.13), (2.14) and (2.15), provides a simple algorithm for computing the partial derivatives of the total cost, allowing the network to be trained by gradient descent.

2.7 Convolutional Neural Networks

Convolutional Neural Networks are a type of feed-forward neural networks that are designed to enable learning from multi-dimensional data, such as images, with minimal preprocessing. They are able to learn directly from the raw data while preserving its structure throughout the layers of the network. A convolutional neural network usually consists of one or more convolutional layers with optional sub-sampling layers in between. The convolutional layers are often followed by fully-connected layers, but this is not mandatory. Hence, a convolutional neural network can be entirely constructed by convolutional layers only.

2.7.1 Convolutional Layers

In a convolutional neural network, a convolutional layer is represented by a 3-dimensional array of neurons. Each neuron in the convolutional layer receives input from a set of neurons in a "small neighbourhood" in the previous layer [27]. In a convolutional layer, a 2-dimensional plane of neurons constitute a *feature map*; hence, a convolutional layer of size $k \times m \times n$, has k feature maps, each of size $m \times n$. The "small neighbourhood" that the convolutional layer receives neurons from as inputs, is defined by a *filter*. A filter is a 2-dimensional array of a fixed size $r \times s$ that is convolved with the feature maps in the previous layer to produce the inputs of the convolutional layer. In other words, a filter identifies patches of neurons of size $r \times s$ in an input feature map that are weighted and fed into the convolutional layer to contribute to a single neuron in its output feature map. The weights associated with the patches are exactly the same for all other patches the same input feature map; however, these weights can be different for other input feature maps. The weights can also be different for different output feature maps.

2.7.2 Convolution

To understand convolutional layers more clearly, it is important to recall the definition of a convolution for discrete 2-dimensional functions. A convolution on two function f and g, denoted by $h = f * g$ is defined as follows:

$$h(i,j) = (f * g)(i,j) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v)g(i-u, j-v) \tag{2.16}$$

Hence, the feature maps in a convolutional layer can be specified by the convolution of filters with the feature maps of the previous layer. Consider the k^{th} feature map, $\mathbf{a}_k^{(l)}$, of size $m \times n$ in layer l. This feature map can be represented as a function of the indices of its neurons:

$$a_k^{(l)}(i,j) = a_{kij}^{(l)} \qquad \text{for} \quad i \in \{1, 2, \ldots, m\}, \quad j \in \{1, 2, \ldots, n\}$$

That is the neuron in position (i,j) in feature map k of layer l. Similarly a filter $\mathbf{W}_k^{(l)}$ of size $r \times s$, associated with the k^{th} feature map in layer l, can be represented as a function of the indices of the weights:

$$W_k^{(l)}(i,j) = W_{kij}^{(l)} \qquad \text{for} \quad i \in \{1, 2, \ldots, r\}, \quad j \in \{1, 2, \ldots, s\}$$

That is the weight in position (i, j) in the filter associated with the k^{th} feature map in layer l. It is possible to define the convolution of the feature map with the filter as follows:

$$\begin{aligned} h(i,j) &= (a * W)_k^{(l)}(i,j) \\ &= \sum_{u=1}^{r}\sum_{v=1}^{s} a_k^{(l)}(i+u-1, j+v-1) W_k^{(l)}(i,j) \\ &\text{for} \quad i \in \{1, 2, \dots, n+1-r\}, \quad j \in \{1, 2, \dots, m+1-s\} \end{aligned} \tag{2.17}$$

Notice that the expression in (2.17) is slightly different from (2.16). The new expression is sometimes referred to as the *cross-correlation*, which is a type of mathematical operations similar to the convolution operation. Hence, for the sake of simplicity, it will be referred to as a convolution throughout the rest of this dissertation.

2.7.3 Froward Propagation of Convolutional Layers

Having defined, the convolution between a filter and a feature map, it is now possible to specify the forward propagation equations of a convolutional layer. However, it is first helpful to define some notations:

- K_l is the number of feature maps in layer l.
- $a_k^{(l)}(i, j)$ is a function representing the k^{th} feature map in layer l.
- $M_l \times N_l$ is the size of the feature maps in layer l.
- $W_{kk'}^{(l)}(i, j)$ is the filter associated with the convolution with k^{th} feature map in layer l, going to the k'^{th} feature map in layer $l + 1$. The filter specifies the weights.
- $R_l \times S_l$, is the size of the filters associated with the layer l.

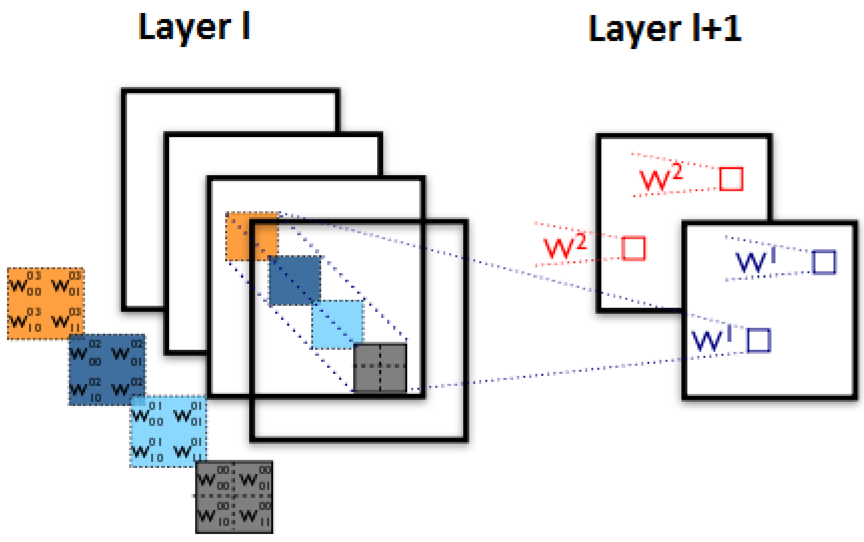
Hence, a convolutional layer is specified by the following forward propagation equations:

$$\begin{aligned} z_{k'}^{(l+1)}(i,j) &= \sum_{k=1}^{K_l} (a * W_{k'})_k^{(l)} + b_{k'} \\ a_{k'}^{(l+1)}(i,j) &= f^{(l+1)}(z_{k'}^{(l+1)}) \\ &\text{for} \quad k' \in \{1, 2, \dots, K_{l+1}\}, \\ &\qquad i \in \{1, 2, \dots, M_l + 1 - R_l = M_{l+1}\}, \\ &\qquad j \in \{1, 2, \dots, N_l + 1 - S_l = N_{l+1}\} \end{aligned}$$

The partial derivatives of the total cost with respect to the parameters of the convolutional layers, can be easily calculated by applying the backward propagation of errors algorithm.

2.7.4 Illustration

Consider the illustration in Figure 2.4. The figure shows two convolutional layers l and $l+1$ consisting of 4 and 2 feature maps respectively. The neurons in each feature map of layer $l+1$, are computed from the sum of the convolutions of 4 2×2 filters, containing the weights, with the 4 feature maps of layer l. Hence, all the neurons in a single feature map in layer $l+1$ share the same set of weights; however, these weights are different from those of the neurons in the other feature map. Hence, for each feature map in layer $l+1$, there is a set of 16 distinct weight parameters, corresponding to 4 filters of size of 2×2 and 4 input feature maps. There are 32 distinct weight parameters in total in layer $l+1$.



Layer l
Layer l+1
W^2
W^2
W^1
W^1
W^{03}_{00} W^{03}_{01} W^{03}_{10} W^{03}_{11}
W^{02}_{00} W^{02}_{01} W^{02}_{10} W^{02}_{11}
W^{01}_{00} W^{01}_{01} W^{01}_{10} W^{01}_{11}
W^{00}_{00} W^{00}_{01} W^{00}_{10} W^{00}_{11}

FIGURE 2.4: An illustration of a convolutional layer [28] (edited).

2.7.5 Sub-Sampling Layers

Sub-sampling layers usually follow convolutional layers. Sub-sampling layers aim to reduce the size of the input maps from a convolutional layer. The reason behind this, is that convolutional layers produce a very large number of features, hence, computing consequent layers from these features can be very computationally expensive. Moreover, the large number of features produced by multiple convolutional layers can cause the network to over-fit (see Section 2.8.2). Consequently, applying a function that down samples the feature maps from convolutional layers, can help alleviate these problems. A sub-sampling layer, performs a sampling operation on a patch of neurons in the input feature maps, this patch identifies a *pool* of neurons to be sub-sampled. Hence, a sub-sampling layer can be specified by the following equation:

$$\begin{aligned} a_k^{(l+1)}(i,j) &= \underset{u,v \in p_{ij}}{\text{down}}(a_k^{(l)}(u,v)) \\ \text{for} \quad & p_{ij} \in P_k^{(l)}, \\ & k \in \{1,2,\ldots,K_l = K_{(l+1)}\} \\ & i \in \{1,2,\ldots,M_{l+1} = \lfloor \frac{M_l}{s} \rfloor\}, \\ & j \in \{1,2,\ldots,N_{l+1} = \lfloor \frac{N_l}{s} \rfloor\} \end{aligned}$$

Where down(.) is some sub-sampling function, such as *max, min, mean*, etc., and P_k^l is the set of all pools of size $s \times s$ in feature map k in layer l. Note that, sub-sampling layers do not contain any parameters.

2.8 Practical Considerations

2.8.1 Cross-Validation

A very common practice in machine learning is to split the available dataset into two parts, a training dataset and a testing (or validation) dataset. This is done so that the model can be tested on a different set of data than the one used to fit it. Hence, the training set is used to fit the model and the testing set is used to check its performance. The splitting is usually done after randomising the original dataset. Typical splitting ratios for most practical applications are around 70:30 to 80:20 training samples to testing samples.

2.8.2 Over-Fitting and Regularisation

Over-fitting occurs when the machine learning model learns to fit the training data very well, but fails to generalise on different sets of data. This happens because the model learns to describes the noise in the training data and fails to capture the underlying relationship between the input features and their labels [1]. Over-fitting usually occurs when the model is too complex, i.e. the number of parameters is large in comparison to the number of training examples. To alleviate over-fitting, one can follow one or both of the following suggestions:

- Try to reduce the complexity of the model by either using more training examples, or removing some of the features in the available data thus reducing the number of parameters.

- Attempt to penalise the size of the parameters by adding *regularisation*. Regularisation can be added to a machine learning model by incorporating the penalty in the total cost. Consider as an example the total cost function for an artificial neural network:
$$C(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^{n} c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i)$$
An easy way to penalise the size of $\mathbf{W}$ and $\mathbf{b}$, is simply to add a multiple of them to the total cost:
$$C_\lambda(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^{n} c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i) + \lambda \cdot \mathrm{sum}(\mathbf{W}) + \lambda \cdot \mathrm{sum}(\mathbf{b})$$
Where λ is a regularisation parameter that refers to the size of the penalty on the parameters $\mathbf{W}$ and $\mathbf{b}$, and 'sum' is a function referring to the element-wise summation of all the elements of $\mathbf{W}$ and $\mathbf{b}$. Since fitting the model requires the minimisation of $C_\lambda(\mathbf{W}, \mathbf{b})$, adding these penalty terms encourages the parameters to be as small as possible.

2.8.3 The Learning Rate

The learning rate α in the gradient descent algorithm is usually chosen manually. Hence, it is beneficial to introduce some guidelines to help in this choice. Inequality (A.3 in Appendix A) gives an upper-bound on the size of the learning rate, below which the gradient descent algorithm is guaranteed to converge for smooth convex functions. Consequently, exceeding this bound might cause the algorithm to diverge. On the other hand, if the learning rate is very small, then the algorithm might take very long to converge to the minimum. Figures 2.5 and 2.6 illustrates these two problems. The choice of the learning rate must attempt to avoid these two pitfalls.
Note that, in the gradient descent algorithm, the closer the value of the update is to the minimum, the smaller the learning rate must be in the next iteration in order to improve the result. Hence, it is a common practice to decrease the learning rate, once the algorithm reaches a value near the minimum, in order to improve the result. The is often referred to as *fine tuning*. Alternatively, one can set the learning rate to be a decreasing function of the number of iterations. Hence, when the algorithm reaches a value near the minimum, the learning rate will have already become small enough to perform the fine tuning.

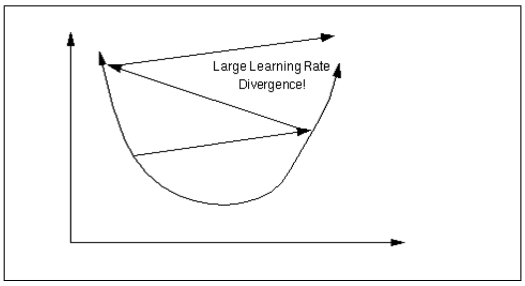


FIGURE 2.5: Large learning rate causes divergence [29].

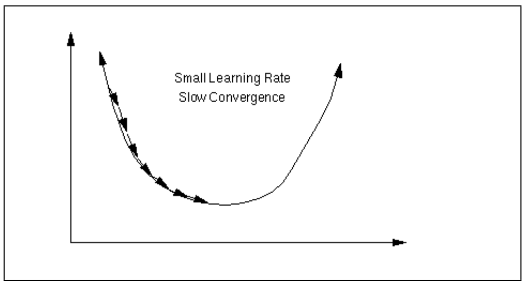


FIGURE 2.6: Small learning rate causes slow convergence [29].

2.8.4 Mini-Batch Gradient Descent

Fitting a machine learning model on a set of training data requires recalculating the value of the partial derivatives of the cost function in every iteration of the gradient descent algorithm. Consequently, if the number of training examples is very large, this calculation becomes very computationally expensive. To avoid this, an extension of the gradient descent algorithm called the *Stochastic Gradient Descent* method, is often used when the training dataset is large. Instead of calculating the derivatives of the total cost function in each iteration, stochastic gradient descent estimates the gradient by calculating the expected value of the partial derivatives from a single training example [30]. Hence, the algorithm becomes:

1. Initialise $\mathbf{x}^0$.
2. For $\tau = 1, 2, \ldots$ do:
$$\mathbf{x}^{(\tau)} = \mathbf{x}^{(\tau-1)} - \alpha\mathbb{E}(\nabla f(\mathbf{x}^{(\tau-1)})) \tag{2.18}$$
$$\text{Calculate} \quad f(\mathbf{x}^{(\tau)})$$
3. Stop when a certain criteria is met.

For instance in the case of the weights in artificial neural networks, the update step (2.18) becomes:

$$\begin{aligned}
\mathbf{W}^{(\tau)} &= \mathbf{W}^{(\tau-1)} - \alpha\mathbb{E}(\nabla C(\mathbf{W}, \mathbf{b})) \\
&= \mathbf{W}^{(\tau-1)} - \alpha\mathbb{E}(\nabla \sum_{i=1}^{n} c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i)) \qquad \text{by (2.11)} \\
&= \mathbf{W}^{(\tau-1)} - \alpha\nabla c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i) \qquad \text{for a random } i \in \{1, 2, \ldots, n\}
\end{aligned}$$

Stochastic gradient descent runs much faster than normal gradient descent; however, it might take longer to converge. Thus, a compromise between normal gradient descent and stochastic gradient descent can be used to combat the computational inefficiency of ordinary gradient descent and the slow convergence of stochastic gradient descent at the same time. This compromise consists of an optimisation method called the *Mini-Batch Gradient Descent*. The idea of mini-batch gradient descent is to estimate the gradient of the cost function from b points from the training dataset, instead of using the whole dataset, like in normal gradient descent, or a single data point, as in stochastic gradient descent. Hence, continuing with the weights of the artificial neural network example, the update step (2.18) becomes:

$$\mathbf{W}^{(\tau)} = \mathbf{W}^{(\tau-1)} - \alpha\nabla(\sum_{i=1}^{b} c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i))$$

2.8.5 Nesterov's Accelerated Gradient Descent

Gradient descent is normally a very efficient algorithm, as it does not take many iterations to converge; however, naturally, the higher the complexity of the objective function is, the greater is the time required for the gradient descent algorithm to converge. One way to speed the optimisation process is to use a version of the gradient descent algorithm called *Nesterov's Accelerated Gradient Descent (NAG)*. The NAG algorithm converges faster than the normal gradient descent method. The NAG algorithm is given below [31]:

1. Initialise $\mathbf{x}^{(0)}, \boldsymbol{\nu}^{(0)}, \mu$.

2. For $\tau = 0, 1, \ldots$ do:

$$\boldsymbol{\nu}^{(\tau+1)} = \mu\boldsymbol{\nu}^{(\tau)} - \alpha\nabla f(\mathbf{x}^{(\tau)} + \mu\boldsymbol{\nu}^{(\tau)})$$

$$\mathbf{x}^{(\tau+1)} = \mathbf{x}^{(\tau)} + \boldsymbol{\nu}^{(\tau+1)}$$

$$\text{Calculate} \quad f(\mathbf{x}^{(\tau+1)})$$

3. Stop when a certain criteria is met.

Where $\boldsymbol{\nu}$ is a velocity vector and μ is a momentum coefficient. The NAG algorithm borrows from the idea of momentum in physics. The principle behind the NAG algorithm and the momentum method, is to iteratively calculate a velocity vector that follows the direction of the persistent reduction in the objective function. Hence, in each update of $\mathbf{x}$, the velocity vector $\boldsymbol{\nu}$ drags the value of $\mathbf{x}$ towards the direction of the reduction of the gradient. The NAG algorithm achieves convergence at a rate of $O(\frac{1}{T^2})$, compared to a rate of $O(\frac{1}{T})$ for the ordinary gradient descent algorithm [31]. Note that, the NAG algorithm can be combined with the mini-batch gradient descent algorithm to further improve the training performance.

2.8.6 Dropout

Because of the large number of parameters in artificial neural networks, they are more prone to over-fitting than other machine learning algorithms. Beside adding regularisation to the cost function, there is another method to combat over-fitting in neural networks. This method involves adding *dropout* to the hidden layers of the network. The idea behind dropout is to randomly remove units (neurons and their connections) from the hidden layers of the network during the training phase to prevent these units from adapting too much to the training data [32]. Dropout can be added to the network by altering the forward propagation equations of the layers in which dropout is to be added. This is done as follows [32]:

$$\begin{aligned}
r_j^{(l-1)} &\sim Bernoulli(p), \qquad \text{for } j \in \{1, 2, \ldots, N_{l-1}\} \\
\mathbf{r}^{(l-1)} &= (r_1^{(l-1)}, r_2^{(l-1)}, \ldots, r_{N_{l-1}}^{(l-1)})^T \\
\tilde{\mathbf{a}}^{(l-1)} &= \mathbf{r}^{(l-1)} \circ \mathbf{a}^{(l-1)} \\
\mathbf{z}^{(l)} &= \mathbf{W}^{(l)} \tilde{\mathbf{a}}^{(l-1)} + \mathbf{b}^{(l)} \\
\mathbf{a}^{(l)} &= f^{(l)}(\mathbf{z}^{(l)})
\end{aligned}$$

Note that, dropout is only done in the training phase; hence, during the testing phase the weights of the dropped units are approximated by averaging out across the thinned network. Having dropout layers in a network not only reduces over-fitting, but also improves the training performance of the network, as the number of parameters are reduced.

2.8.7 Activation Functions

There are various choices of activation functions that could be used in feed-forward and convolutional neural networks. A network can even have different activation functions in different layers. A selection of the most common choices of activation functions and their properties in relation to neural networks is given below:

- **The Identity Function**
  The identity function is simply $f(x) = x$. The identity function is a linear mapping; hence, it does not add complexity to the network. It is often used in the final layer to produce the output; however, it could be used in the hidden layers as well.

- **The Sigmoid Function**
  The sigmoid function $f(x) = \frac{1}{1+e^{-\beta x}}$, where β is a slope coefficient, is a non-linear mapping that maps its input to the interval $[0, 1]$. The sigmoid increases the non-linear properties of the neural network, thus it is often used throughout the hidden layers as well as the output layer. It is often used as a hypothesis function in the output layer for 2-class classification problems as its output can signify the probability of belonging to one of the two classes.

- **The Hyperbolic Tangent Function**
  The hyperbolic tangent function $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, maps its input to the interval $[-1, 1]$. The hyperbolic tangent is a non-linear function, that is often used throughout the layers of the neural network especially in regression problems.

- **Rectified Linear Units (ReLU)**
  The Rectified Linear Units is defined as:

  $$f(x) = \mathrm{ReLU}(x) = \max(0, x)$$

  The Rectified Linear Units is used to increase the non-linear properties of the network. It resembles the biological mechanism of the neurons, as neurons in the brain either fire up (when $x > 0$) or stay dormant (when $x < 0$). It has an advantage over the sigmoid and the hyperbolic tangent functions, as it helps speed up the training of the network.

- **The Softmax Function**
  The softmax function is a multivariate function, defined as follows:

  $$f(x_1, x_2, \ldots, x_n) = \mathrm{softmax}(x_1, x_2, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum\limits_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum\limits_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum\limits_{i=1}^{n} e^{x_i}} \right)$$

  The softmax function maps its inputs to the interval $[0, 1]$ such that the sum of all its outputs equals 1. Hence, it is often used in the final layer of an artificial neural network (as a hypothesis function), for multi-class classification problems, because its outputs can be interpreted as probabilities for mutually exclusive events (classes).

2.8.8 The Cost Function

The choice of the cost function in fully-connected feed-forward and convolutional neural networks, depend primarily on the nature of the learning problem. Since this dissertation is only concerned with supervised regression and classification problems, the following will give a short overview for the choice of cost function for these problems exclusively:

- **Regression Problems**
  A popular choice for the cost function in regression problems, is the square of the $L2$-norm of the difference between a label and the value of the hypothesis function corresponding to this label (i.e. the squared distance):
  $$c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i) = ||h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}) - \mathbf{y}_i||_2^2$$
  The goal in regression problems is to predict the values of a set of continuous variables based on a set of input feature variables. Hence, this cost function insures that the deviation of predictions from the true value is penalised.

- **Classification Problems**
  In classification problems, a cross-entropy function is usually used as a cost function:
  $$c(h_{\mathbf{W},\mathbf{b}}(\mathbf{z}_i^{(L)}), \mathbf{y}_i) = -y \log(h_{\boldsymbol{\beta}}(\mathbf{x})) - (1-y)\log(1 - h_{\boldsymbol{\beta}}(\mathbf{x}))$$
  The cross-entropy function penalises misclassification, making it a desirable choice in classification problems.

Chapter 3

Gender Prediction from Facial Images

3.1 Introduction

The problem of detecting a subject's gender from their face has been studied by many researchers around the world. Many of them, proposed complex algorithms that are able to perform this task. For example, Shirkey and Gupta [33], proposed an algorithm that detects the facial features of the subject and compares them to a database of "Male versus Female" features, making probabilistic conclusions about the subject's gender, based on these features. However, this approach is fairly complex and requires building databases of Male and Female features, which might be expensive to compile. An alternative approach to this method, is to use neural networks to try to learn the facial features of each gender and classify the subject accordingly. Hence, this chapter introduces and compares three network architectures that are used to perform gender classification.

3.2 Methodology

3.2.1 Training and Test Data

The data used in this task comes from the *Facial Recognition Technology (FERET) Database* [34]. The FERET Database contains facial photographs of 994 different subjects, with different facial orientations and facial expressions for each subject. The FERET Database also contains background information about the subjects, labelled as *Ground Truths*. The ground truths files contain information such as, gender, date of birth, date photographed, etc. Hence, the gender information of the subjects were extracted from the these files, and were coded as binary digits, as labels for the image data. The photographs of the subjects in the dataset were all wide shots. Accordingly, before they were loaded into computer memory, they were cropped using an open-source software called SNFaceCrop. Consequently, the cropped facial images were loaded as 8-bit grayscale images, resized to 50×50 pixels and labelled according to the subject's gender. Finally, the labelled images were split into training and testing datasets according to the identities of their subjects, where 100 random subjects where chosen as testing data and the rest were used as training data.

3.2.2 Building the Neural Networks

To build the neural networks, an edited version of a *Python* implementation of artificial neural networks, developed by Dr Ben Graham, was used. This implementation uses a Python library called *Theano* [35] that can manipulate the layers of the network efficiently. A Python script was coded to build three different gender classification neural networks, using this implementation. These networks are classification networks; hence, the cross-entropy function was used as the cost function for this problem. The architectures of the networks, along with a summary of their performances are specified in the consequent sections.

3.3 Gender Classification Network 1 (GCN1)

3.3.1 Architecture

The architecture of GCN1 is fairly deep, consisting of 6 hidden layers, in addition to the input and the output layers (i.e. 8 layers in total). The structures of these layers are specified below:

- **The Input Layer:** is the two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 20 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 50 feature maps.

- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 4.
- **The 5th Hidden Layer:** reshapes the outputs of the previous layer to a one-dimensional array.
- **The 6th Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation.
- **The Output Layer:** is a single neuron layer with a sigmoid activation.

Note that since GCN1 contains two convolutional layers, it is considered a convolutional neural network. Figure C.1 is a visual illustration of the architecture of the network.

3.3.2 Evaluation of the Results

Overall GCN1 managed to classify 92.6% of the testing data correctly. Hence, only 7.4% of the testing samples were misclassified as female when in fact they were male, or as male when in fact they were female. To see the results of the classification more clearly, two bar-plot are produced in Figure 3.1. The first bar-plot shows the distribution of the gender from the output of GCN1, while the second bar-plot shows the actual distribution of the gender in the testing data. To examine the results further, a confusion matrix was produced in Table 3.1.

		Actual Gender		
		Male	Female	Total
Predicted Gender	Male	292	12	304
	Female	25	171	196
	Total	317	183	500

Table 3.1: Confusion matrix for the results of GCN1.

From Table 3.1, GCN1 classified 292 out of 317 males in the testing data correctly; that is approximately 92.1% percent of males in the test set. For the females, GCN1 was slightly more accurate, classifying 171 females out of 183 from the test data correctly; that is an approximately 93.4% accuracy.

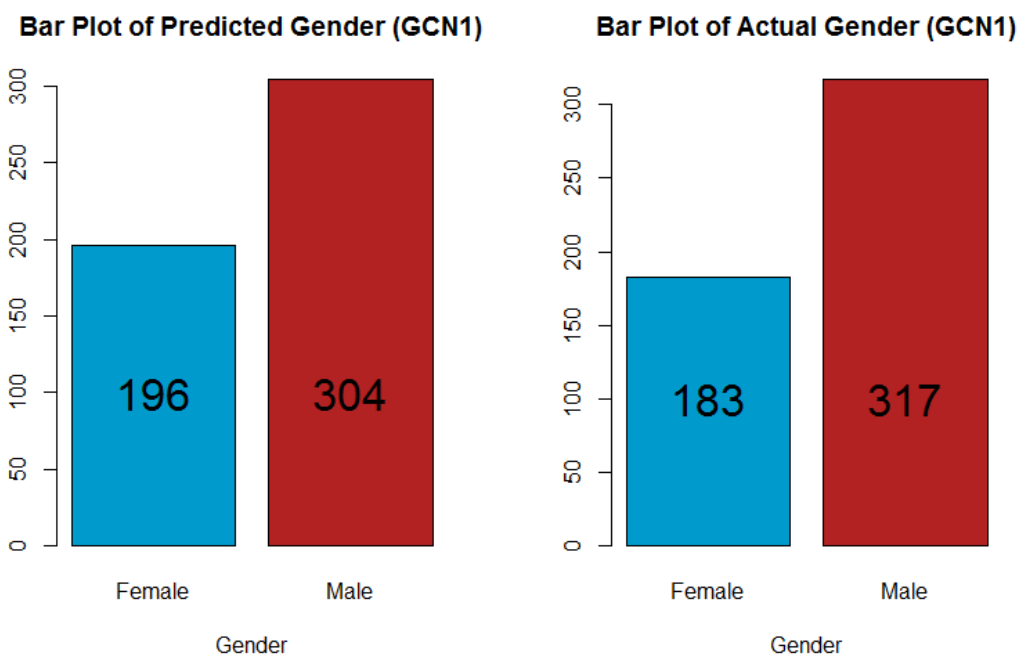


FIGURE 3.1: Bar-Plots showing the distribution of the results of GCN1 versus the actual distribution of gender in the testing data.

3.4 Gender Classification Network 2 (GCN2)

3.4.1 Architecture

GCN2 has a fairly shallow but wide fully-connected architecture, consisting of 5 hidden layers in total. The structures of the layers are specified below:

- **The Input Layer:** is a one dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation.
- **The 2nd Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation.
- **The 3rd Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation.
- **The Output Layer:** is a single neuron layer with a sigmoid activation.

Figure C.2 is a visual illustration of the architecture of the network.

3.4.2 Evaluation of the Results

For GCN2, the classification accuracy was 89.2%; hence, misclassifying 10.8% of the testing data. The distribution of the gender from the output of GCN2 compared to the actual distribution of the gender in the testing data can be seen in the bar-plots in Figure 3.2. To further examine the output of GCN2, a confusion matrix is produced in Table 3.2.

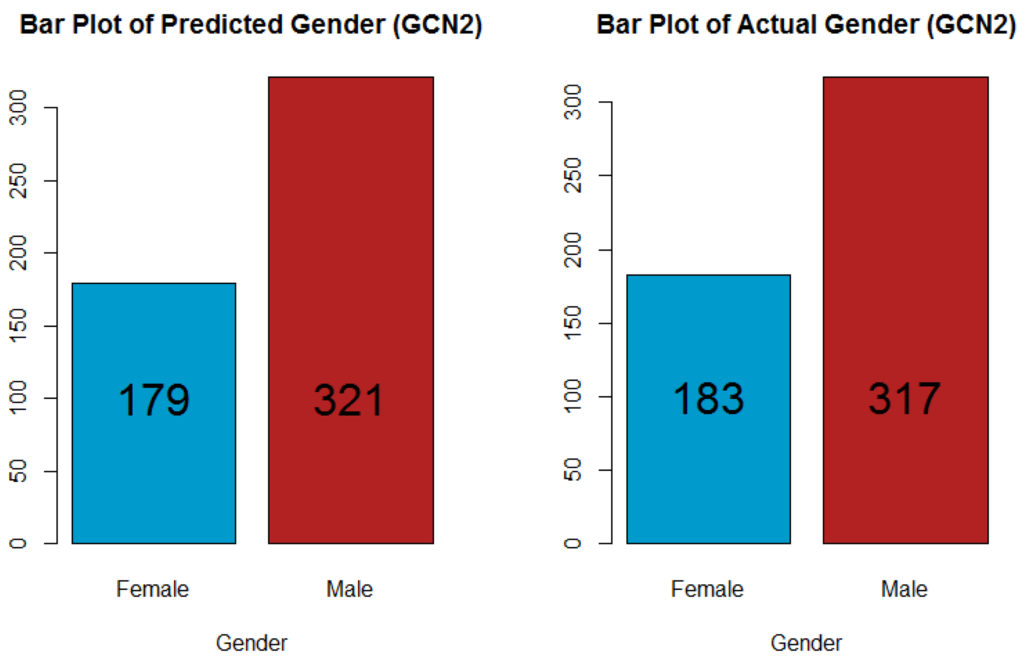


FIGURE 3.2: Bar-Plots showing the distribution of the results of GCN2 versus the actual distribution of gender in the testing data.

		Actual Gender		
		Male	Female	Total
Predicted Gender	Male	292	29	321
	Female	25	154	179
	Total	317	183	500

TABLE 3.2: Confusion matrix for the results of GCN2.

The confusion matrix (Table 3.2) suggests that, GCN2 classified 92.1% of the males in the testing data correctly, while only classifying %84.1 of the females correctly. These results imply that GCN2 is biased towards classifying samples as males; hence, it is not as capable of recognising female features as well as GCN1.

3.5 Gender Classification Network 3 (GCN3)

3.5.1 Architecture

GCN3 has a very similar architecture to GCN2. The only difference is that GCN3 employs dropout throughout its layers. The structures of the layers of GCN3 are specified below:

- **The Input Layer:** is a one dimensional representation of the facial image, with dropout applied at a probability of 0.2.
- **The 1st Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 2nd Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 3rd Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The Output Layer:** is a single neuron layer with a sigmoid activation.

Figure C.3 is a visual illustration of the architecture of the network.

3.5.2 Evaluation of the Results

The classification accuracy of GCN3 was 89%, misclassifying 11% of the testing data. The distribution of the gender from the output of GCN3, in comparison to the actual distribution of the gender in the testing set, is provided in the bar-plots in Figure 3.3.

A quick look at the bar-plots in Figure 3.3 reveals that GCN3 managed to predict the proportion of males to females in the testing data accurately. However, this does not mean that the predictions themselves were accurate. An examination of the prediction performance of GCN3 can be done by referring to the confusion matrix in Table 3.3.

		Actual Gender		
		Male	Female	Total
Predicted Gender	Male	290	28	318
	Female	27	155	182
	Total	317	183	500

TABLE 3.3: Confusion matrix for the results of GCN3.

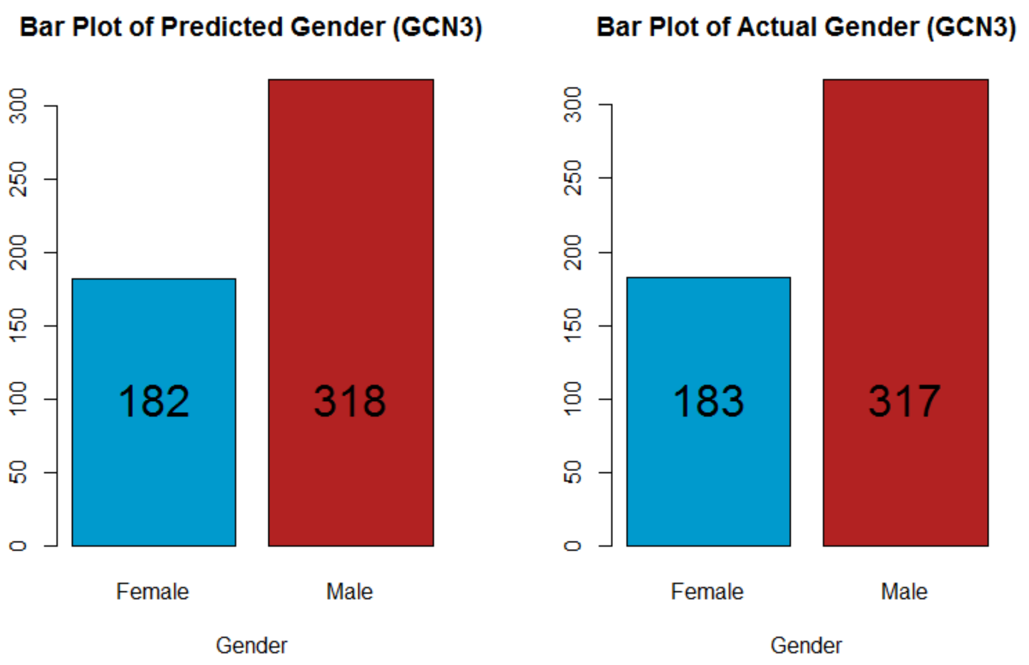


FIGURE 3.3: Bar-Plots showing the distribution of the results of GCN3 versus the actual distribution of gender in the testing data.

From the confusion matrix (Table 3.3), GCN3 classified 91.5% of the males in the testing data correctly, while only classifying 84.7% of the females correctly. These results suggest that, like GCN2, GCN3 is biased towards classifying samples as males and is not as capable of recognising female features as GCN1.

3.6 Comparison of the Networks

Table 3.4 provides a comparison between the three gender classification neural networks. The table suggests that GCN1 with the convolutional architecture performs best overall. Both GCN2 and GCN3, seem to be biased towards classifying males, as the female classification accuracy for both is low compared to GCN1. GCN2 is able to classify males more accurately than GCN3, while GCN3 is slightly more accurate in classifying females than GCN2.

Network	Type	Overall Accuracy	Male Accuracy	Female Accuracy
GCN1	Convolutional	92.6%	92.1%	93.4%
GCN2	Fully-Connected	89.2%	92.1%	84.1%
GCN3	Fully-Connected	89.0%	91.4%	84.7%

TABLE 3.4: Comparison table for the gender classification networks.

Chapter 4

Age Prediction from Facial Images

4.1 Introduction

Unlike other problems concerning facial images, such as gender prediction, facial recognition and facial keypoints detection, the problem of age prediction has not been studied extensively. Only a handful of literature exists on this problem, most of which do utilise artificial neural networks. For example, a pioneering method for age classification proposed by Kwon and da Vitoria Lobo [36], extracts facial features like eyes, mouth, ears, etc. from the images, calculates the ratios between these features and classifies the age into three groups based on the ratios. Few other methods for age estimation utilise artificial neural networks indirectly. For instance, Fukai, *et. al.* [37], propose a method in which features are extracted from the facial images and are fed into a type of unsupervised artificial neural networks called *Self Organising Maps (SOMs)* to produce an estimate of the age. Another method proposed by Thakur and Verma [38], attempts to extract parameters representing facial features and wrinkles from the facial images and then feed these parameters into a feed-forward neural network, classifying the age into four groups. All of the methods mentioned earlier require a great deal of pre-processing, which could prove to be extremely cumbersome. Hence, this chapter aims to introduce an efficient method for age prediction that requires minimum pre-processing, by exploiting the power of convolutional neural networks.

4.2 Methodology

4.2.1 Training and Test Data

For the age prediction problem, the same dataset, FERET, was used as in the gender classification problem (see Section 3.2.1). Again, the images in the this dataset were cropped, loaded as 8-bit grayscale images, resized to 50×50 pixels and labelled according to the age. The age labels were constructed by extracting the year of birth of the subjects and the year their photographs were taken from the ground truths files, and then finding the difference between the two dates to infer the age. Rather than coding the age as a single integer, it was coded as a string of binary digits. This allows for a probabilistic interpretation for the output of the neural network. Consider the random variable X representing the subject's age. X can be redefined as:

$$X = \mathbb{I}_{X \geq 1} + \mathbb{I}_{X \geq 2} + \cdots + \mathbb{I}_{X \geq 100}$$

For ages between 0 and 100, where $\mathbb{I}$ is an indicator function. Hence, the output of the network $\mathbb{E}(X)$ can be defined as:

$$\begin{aligned}\mathbb{E}(X) &= \mathbb{E}(\mathbb{I}_{X \geq 1} + \mathbb{I}_{X \geq 2} + \cdots + \mathbb{I}_{X \geq 100}) \\ &= \mathbb{P}(X \geq 1) + \mathbb{P}(X \geq 2) + \cdots + \mathbb{P}(X \geq 100)\end{aligned}$$

Consequently, coding the age labels as strings of binary digits (e.g. $(1, 1, \ldots, 1, 0, 0, \ldots, 0)$), allows for this probabilistic interpretation.
Note that, for this problem, only images with frontal facial orientations were used. Thus, images in which the subjects were facing sideways or had their heads tilted were discarded. Finally, the remaining images were split into a training dataset and a testing dataset according to the identities of their subjects, where 100 random subjects where chosen as testing data and the rest were used as training data.

4.2.2 Building the Neural Networks

The same neural networks implementation seen in Section 3.2.2 was used to build three different age prediction networks. Since age prediction is a regression problem, these networks were assigned a square distance cost function. The architectures of these networks, along with a summary of their performances are specified in the consequent sections.

4.3 Age Prediction Network 1 (APN1)

4.3.1 Architecture

APN1 is a fully-connected network, consisting of 5 layers in total with dropout utilised throughout its layers. The specifications of the layers of this network are provided below:

- **The Input Layer:** is a one dimensional representation of the facial image, with dropout applied at a probability of 0.2.
- **The 1st Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 2nd Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 3rd Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The Output Layer:** is a 100 neurons layer with a sigmoid activation.

Figure C.4 in Appendix C provides an illustration of the architecture of this network.

4.3.2 Evaluation of the Results

To understand the distribution of the age values that were predicted by APN1 in comparison to the real age values, an enhanced scatter-plot of the predicted age against the real age was produced in Figure 4.1. The box-plots on the side of the axes of the scatter-plot, suggest that the range of the predicted age values is much narrower than the range of the real age data, suggesting that APN1 fails to capture the range of ages present in the testing data. To test how well APN1 performs age prediction, a linear regression model was built regressing the predicted age values on the real age values. The linear model is represented by the solid red line in the scatter-plot in Figure 4.1. Clearly, the linear model deviates significantly from the ideal case, represented by the blue dashed line, in which the predicted age exactly equals the real age (i.e. line with slope 1 and intercept 0). This suggests that APN1 performs the age prediction task poorly.

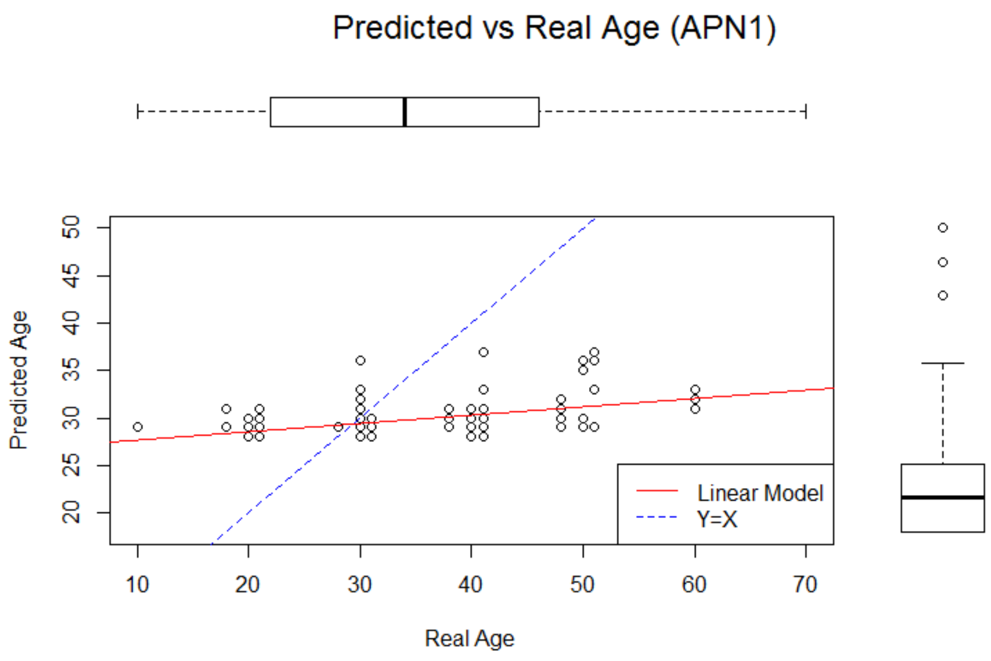


FIGURE 4.1: Enhanced scatter-plot of the predicted age against the real age for APN1.

4.4 Age Prediction Network 2 (APN2)

4.4.1 Architecture

APN2 has a convolutional architecture with a single convolutional layer and a single sub-sampling layer. In addition, APN2 contains two fully-connected layers, plus an input and an output layer. The specifications of the layers of this network are provided below:

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 20 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** reshapes the outputs of the previous layer to a one-dimensional array.
- **The 4th Hidden Layer:** is a fully-connected layer containing 300 neurons with a *ReLu* activation.
- **The 5th Hidden Layer:** is a fully-connected layer containing 300 neurons with a *ReLu* activation.

- **The Output Layer:** is a 100 neurons layer with a sigmoid activation.

Figure C.5 in Appendix C provides an illustration of the architecture of the network.

4.4.2 Evaluation of the Results

The results of APN2 are summarised in the enhanced scatter-plot in Figure 4.2. The box-plots on the axes of the scatter-plot suggest that both the predicted age data and the real age data have similar ranges, with the range of the predicted age only a little narrower than that of the real age. Overall, the distributions seem to be quite similar, suggesting that APN2 performs the age prediction for a wide range of ages. To test the accuracy of the predictions of APN2, a linear regression model of the predicted age against the real age was built. The linear model is represented by the solid red line in the scatter-plot in Figure 4.2. The plot shows that the linear model is fairly close the ideal case, represented by the blue dashed line, suggesting that APN2 is able to predict the ages accurately.

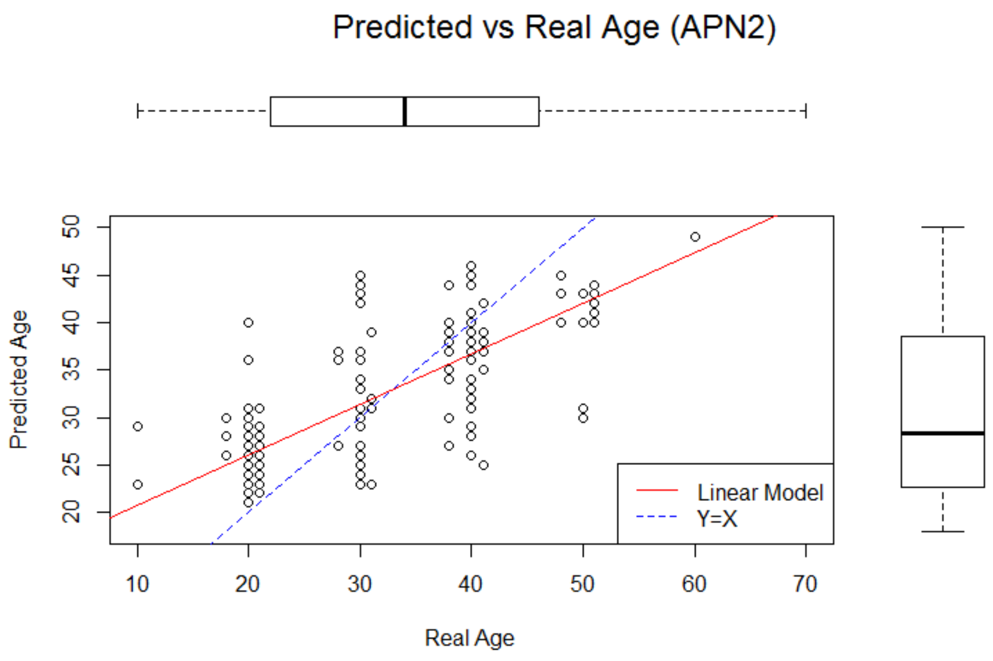


FIGURE 4.2: Enhanced scatter-plot of the predicted age against the real age for APN2.

4.5 Age Prediction Network 3 (APN3)

4.5.1 Architecture

APN3 has a convolutional architecture, consisting of 8 layers in total. The specifications of the layers of this network are provided below:

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 10 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 4. This layer outputs 20 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** reshapes the outputs of the previous layer to a one-dimensional array.
- **The 6th Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation.
- **The 7th Hidden Layer:** is a fully-connected layer containing 300 neurons with a *ReLu* activation.
- **The Output Layer:** is a 100 neurons layer with a sigmoid activation.

Figure C.6 in C provides an illustration of the architecture of APN3.

4.5.2 Evaluation of the Results

The results of APN3 are summarised in the enhanced scatter-plot in Figure 4.3. The box-plots on the axes of the scatter-plot suggest that both the predicted age data and the real age data have similar ranges, suggesting that APN3 manages to capture almost the entire range of ages in the testing data. A regression model was built to test the accuracy of the predictions of APN3. The model is represented by the solid red line in the scatter-plot in Figure 4.3, and suggests that APN3 is fairly accurate, as its predictions do not deviate significantly from the idea case.

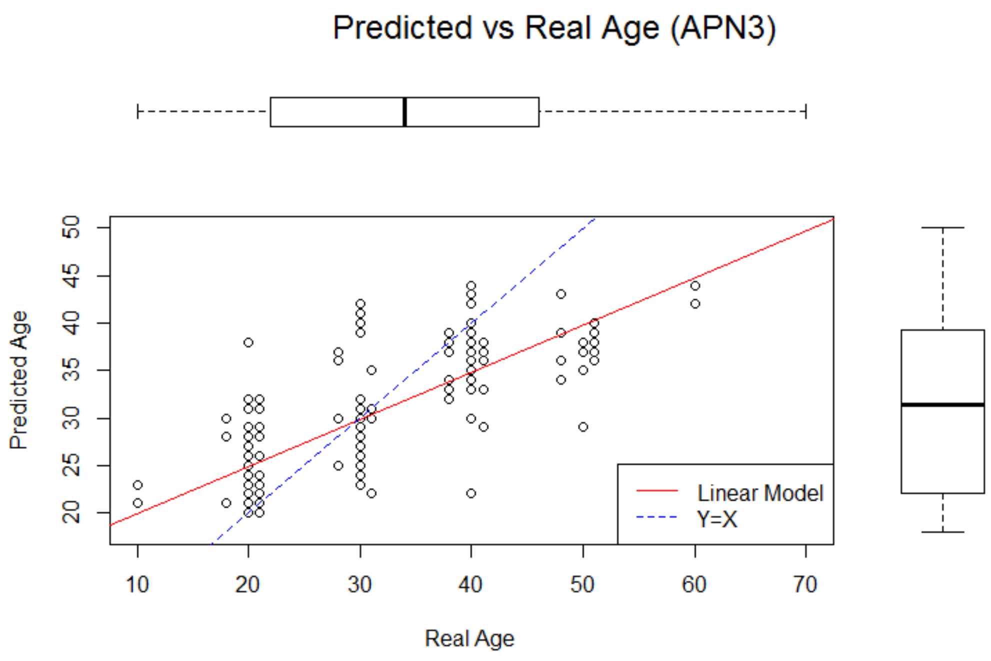


FIGURE 4.3: Enhanced scatter-plot of the predicted age against the real age for APN3.

4.6 Comparison of the Networks

Table 4.1, provides a short comparison of the predictions of the three networks, in comparison to the real age data. As can be seen from the table, APN1 performs poorly, as its predictions have a very small variance compared to the real age, implying that it is unable to predict a wide range of ages. Moreover, the Pearson correlation coefficient between the predictions of APN1 and the real age data, is small compared to the other two networks. On the other hand, both APN2 and APN3 perform well, with variances closer to the real age variance, as well as high correlation coefficients. Furthermore, the *mean of the absolute difference between the real ages and the predicted ages (MADA)*, is lower than that of APN1, confirming that both APN2 and APN3 have better performances. It is worth noting that the performances of APN2 and APN3 are very similar according to the chosen testing benchmarks.

Network	Type	Mean	Variance	Correlation	MADA
Real Age	N/A	30.400	121.333	1	N/A
APN1	Fully-Connected	29.440	3.162	0.540	8.76
APN2	Convolutional	31.635	59.570	0.762	5.695
APN3	Convolutional	30.070	50.588	0.770	5.37

TABLE 4.1: Comparison table for the age prediction networks.

Chapter 5

Non-Linear Dimensionality Reduction of Facial Images

5.1 Introduction

Dimensionality reduction is an important topic in statistical and machine learning. Consider a fixed amount of data from a high dimensional space. As the number of dimensions of this space increases, the volume of this spaces increases very rapidly. Thus, the available data become increasingly more sparse with the increase of its dimension. This problem is often referred to as the *curse of dimensionality.* The curse of dimensionality affects the performance of statistical learning algorithms, as the higher the number of dimensions of the data, the more data is needed to overcome the sparsity. This, in turn, affects the efficiency and performance of the learning algorithm. Hence, dimensionality reduction is often used to counteract the effects of the curse of dimensionality. There is a wide variety of dimensionality reduction techniques, but all of them fall in one of two categories: linear techniques or non-linear techniques. Among the most widely used linear techniques are, *Principal Component Analysis*, *Singular Value Decomposition* and *Independent Component Analysis* [39]. Another approach to dimensionality reduction, is the use non-linear methods. One non-linear dimensionality reduction method, the *Autoencoder*, uses feed-forward artificial neural networks to shrink the dimensions of the input data. This chapter discusses the use of autoencoders on facial image data.

5.2 Background

Autoencoders are multi-layer neural networks with a small central layer. An autoencoder network can be split into two parts: an encoder network, which is a neural network that takes the raw data as input and outputs a low-dimensional representation of this data, and a decoder network, which takes the low-dimensional representation as input and reverses the operations of the encoder to output a new set of data that has the same structure and dimensions as the original data [40]. Hence, the low dimensional representation of the original data is nothing but the central layer of the autoencoder. An example of an autoencoder is given in Figure 5.1. Autoencoders are trained by minimising the difference between the raw inputs and the reconstructed outputs.

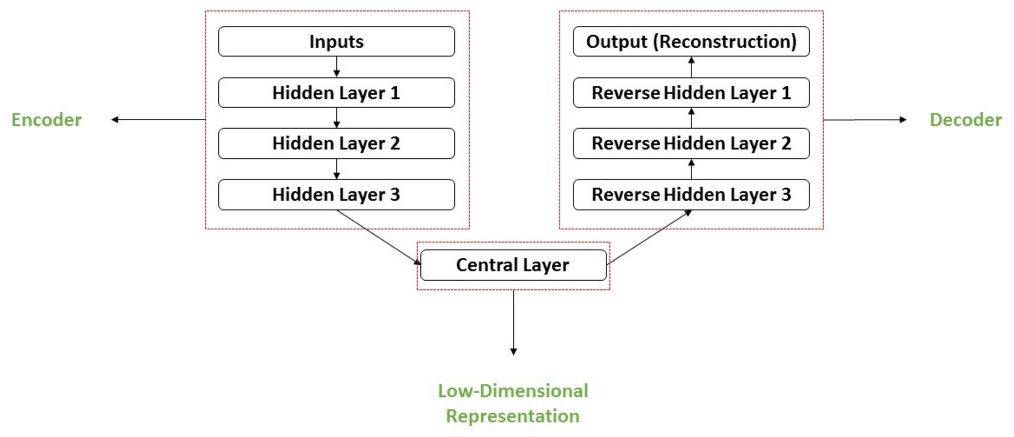


FIGURE 5.1: An example of an autoencoder network.

5.3 Methodology

5.3.1 Training and Test Data

For the dimensionality reduction problem the FERET dataset -seen in the previous problems (see Section 3.2.1)- was used. The images in the this dataset were cropped, resized to 50 pixels and loaded as 8-bit grayscale images. The images were then duplicated, with the duplicates serving as labels for the original images. All available images were used in this problem, regardless of their subject's facial orientation. Finally, the labelled facial images were split into a training dataset and a testing dataset according to the identities of their subjects, where 20 random subjects where chosen as testing data and the rest were used as training data.

5.3.2 Building the Neural Networks

The same neural networks implementation seen in Section 3.2.2 was used to build three different dimensionality reduction networks. The objective of these networks is to minimise the difference between the original input image and the reconstructed output image; hence, a square distance cost function was used. The architectures of these networks, along with a summary of their performances are specified in the consequent sections.

5.3.3 An Evaluation Metric

To quantify the quality of the reconstruction of the facial images, the *Peak Signal-to-Noise Ratio (PSNR)* was used. The PSNR is a metric that is often used to measure the quality of images that are reconstructed after compression [41]. The PSNR is defined as follows:

$$PSNR(\hat{h}, h_0) = 10\log_{10}(\frac{MAX_i^2}{mse(\hat{h}, h_0)}) \tag{5.1}$$

Where, $\log_{10}$ is the logarithm with base 10, $\hat{h}$ is the reconstructed image and h_0 is the original image. In (5.1), $mse(\hat{h}, h_0)$ is the mean squared error between the original image and the reconstructed image and MAX_i is the maximum possible value for the pixels of the image ($MAX_i = 256$ in case of an 8-bit grayscale image as in the training data).

5.4 Architectures of the Networks

All of the three dimensionality reduction networks have convolutional architectures; however, they differ in depth, width and filter sizes. As mentioned before, an autoencoder network consists of an encoder network and a decoder network. The decoder network reverses the operations of the encoder network to produce the reconstruction. Hence, for the sake of brevity, only the architectures of the layers of the encoder will be specified in this section; however, visual representations of the full networks can be found in Figures C.7, C.8 and C.9 in Appendix C.

5.4.1 Dimensionality Reduction Network 1 (DRN1)

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 20 feature maps.

- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 40 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 2. This layer outputs 60 feature maps.
- **The 6th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The Central Layer:** is a convolutional layer with a hyperbolic tangent activation function and a filter of size 3. This layer outputs 200 feature maps.

5.4.2 Dimensionality Reduction Network 2 (DRN2)

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 30 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 60 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 2. This layer outputs 80 feature maps.
- **The 6th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The Central Layer:** is a convolutional layer with a hyperbolic tangent activation function and a filter of size 2. This layer outputs 100 feature maps.

5.4.3 Dimensionality Reduction Network 3 (DRN3)

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 20 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 40 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 2. This layer outputs 60 feature maps.
- **The 6th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 7th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 200 feature maps.
- **The Central Layer:** is a convolutional layer with a hyperbolic tangent activation function and a filter of size 3. This layer outputs 400 feature maps.

5.5 Comparison of the Networks

To compare the three dimensionality reduction networks, the histograms of the PSNR values of the output images of the networks were produced, to allow for the examination and comparison of their distributions. The histograms can be seen in Figures 5.2, 5.3 and 5.4. The histograms reveal that the distributions of the PSNR values for the outputs of the three networks are very similar in shape, with the most noticeable difference being the location of the mean.

To further differentiate between the three networks, a comparison table was compiled and is shown in Table 5.1. In the table, the *Reduction Factor* refers to the ratio of size of the low-dimensional representation of the image to the size of the dimensions of the original image ($50 \times 50 = 2500$). DRN1 has the highest mean PSNR at 17.033; however, it also has the highest reduction factor. On the other hand, DRN3 has the lowest mean PSNR at 15.397, with the lowest reduction factor 0.16. Note that DRN3

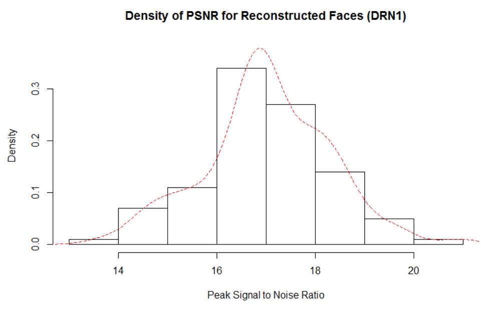


FIGURE 5.2: Distribution of the PSNR values for output of DRN1.

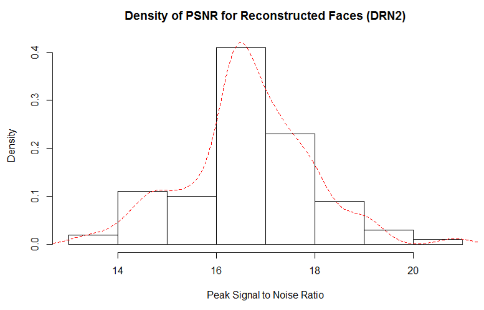


FIGURE 5.3: Distribution of the PSNR values for output of DRN2.

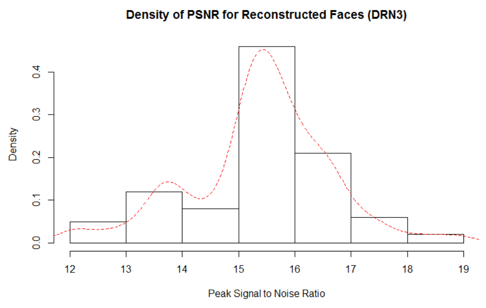


FIGURE 5.4: Distribution of the PSNR values for output of DRN3.

reduces the dimension considerably, while only sacrificing a little of the quality, which might suggest that it has the best performance. That being said, the quality of image reconstruction is mostly subjective; hence, the reader is referred to Figures D.1, D.2 and D.3 in Appendix D, which show samples of the outputs of the three networks, to judge the best performing network.

Network	Mean PSNR	Variance	Dimension Size	Reduction Factor
DRN1	17.033	1.638	1800	0.72
DRN2	16.646	1.611	1600	0.64
DRN3	15.397	1.589	400	0.16

TABLE 5.1: Comparison table for the dimensionality reduction networks.

Chapter 6

Frontal Facial Pose Reconstruction

6.1 Introduction

Most facial recognition systems require the subject's face to be forward oriented. Thus, facial image frontalisation is an important problem in the domain of computer vision. Facial alignment has been studied extensively, and many systems have been proposed to perform the frontalisation. For example, Taigman, et. al. [42], use 3D modelling to align the face as part of their DeepFace facial recognition algorithm. Another approach to this problem, proposed by Cao, *et. al.* [43], uses a regression based method that learns a function to map the original facial image to its frontalised counterpart. The method that is proposed in this chapter is a variation of the regression approach using convolutional neural networks and autoencoders.

6.2 Methodology

6.2.1 Training and Test Data

As before, for this problem, the FERET dataset was used (see Section 3.2.1). The images in the this dataset were cropped and loaded as 8-bit grayscale images. The images were then duplicated and the straight poses were extracted from the duplicates. The extracted frontal poses were then used to label the original images. All the available images were used for this problem, with exactly one straight pose label per subject. The labelled facial images were finally split into a training dataset and a testing dataset according to the identities of their subjects, where 20 random subjects where chosen as testing data and the rest were used as training data.

6.2.2 Building the Neural Networks

The same neural networks implementation seen in Section 3.2.2 was used to build two different frontal pose reconstruction networks. The architectures of these networks, along with a sample of their outputs are provided in the consequent sections. The architectures of these networks are similar to that of autoencoders. The only difference is that in this type of networks the labels are the frontal face images rather than the original images. This incentivises the network to learn the facial features of the face independent from its orientation, as the output is penalised according to its deviation from the frontally oriented version of the face. Hence, a square distance cost function is used for this purpose.

6.3 Architectures of the Networks

Since, the frontal pose reconstruction networks are special versions of autoencoders, only the architectures of the layers of the encoder networks will be specified. The full specifications of these networks can be found in the visual illustrations in Figures C.10 and C.11 in Appendix C.

6.3.1 Frontal Pose Reconstruction Network 1

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 20 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 40 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 2. This layer outputs 60 feature maps.

- **The 6th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The Central Layer:** is a convolutional layer with a hyperbolic tangent activation function and a filter of size 3. This layer outputs 200 feature maps.

6.3.2 Frontal Pose Reconstruction Network 2

- **The Input Layer:** is a two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 20 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 3. This layer outputs 40 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 5th Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 2. This layer outputs 60 feature maps.
- **The 6th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The Central Layer:** is a convolutional layer with a hyperbolic tangent activation function and a filter of size 2. This layer outputs 200 feature maps.

6.4 Results

Since the output facial images are not required to be exact replicas of the frontal facial images, the PSNR cannot be used to test their quality. Moreover, the quality of this reconstruction is highly subjective and can differ from one person to another. For this reason, the reader is asked to judge the quality of the reconstruction subjectively. Hence, a sample of the reconstructed frontal faces is presented in Figures E.1 and E.2 in Appendix E. Each figure consists of three triplets of columns, with each column containing nine images. In each triplet, the column on the right contains the input facial images, the column in the middle contains the required facial images (required straight poses) and the column on the left contains the output facial images (i.e. the straight pose reconstructions).

Chapter 7

Facial Keypoints Detection

7.1 Introduction

Facial keypoints detection refers to the problem of identifying the locations of key features in a facial image, such as the positions of the eyes mouth, the lips, the tip of the nose, etc. This is an important problem as identifying the locations of these keypoints can help in building 3D facial rendering and facial alignment systems. The facial keypoints detection problem has been studied extensively, and many approaches has been proposed. For instance, Herpers, *et. al.* [44] propose a method for the detection of facial keypoints, that is based detecting edges and lines locally within the facial images, utilising a filtering scheme based on the mathematical concept of *steerable filters*. Another example of a facial keypoints detection system is the Clement, *et. al.* [45] approach. In this approach, Linear Discriminant Analysis is performed on feature vectors that represent the face. This approach is proposed for 3D face scans rather that 2D facial images; hence, it might not perform as well in the case of facial images. Other approaches for this problem involve the use of artificial neural networks. As an example, Nouri [46], advocates the use of convolutional neural networks for detecting the facial features and their locations in the facial image. In this chapter, two approaches for this problem are proposed. These approaches are based on artificial neural networks and are a variation of Nouri's [46] method.

7.2 Methodology

7.2.1 Training and Test Data

For the facial keypoints detection task, the data was obtained from the Facial Keypoints Detection competition, hosted on Kaggle.com [47]. The dataset contains 7049 96×96 images. Each of these images is stored as a vector of length 9216 and labelled with the (x, y) coordinates of 15 facial keypoints. This dataset was loaded and reconstructed so that each image is represented by a an 8-bit 2 dimensional array of size 96×96, with its labels attached to it. Finally, the resulting data was split into a training dataset and a testing dataset with a split ratio of 80:20 training to testing samples.

7.2.2 Building the Neural Networks

The neural networks implementation seen in Section 3.2.2 was used to build two different facial keypoints detection networks. Since, facial keypoints detection is a regression problem, a squared distance cost function was used. The architectures of these networks, along with a sample of their outputs are provided in the consequent sections.

7.3 Facial Keypoints Detection Network 1 (FKDN1)

7.3.1 Architecture

FKDN1 employs a deep and convolutional architecture. It consists of 10 layers including the input and output layers. The structures of the layers of FKDN1 are specified below:

- **The Input Layer:** is the two dimensional representation of the facial image.
- **The 1st Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 10 feature maps.
- **The 2nd Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 2×2.
- **The 3rd Hidden Layer:** is a convolutional layer with a *ReLU* activation function and a filter of size 5. This layer outputs 20 feature maps.
- **The 4th Hidden Layer:** is a sub-sampling layer that performs max-pooling on a pool of size 4.

- **The 5th Hidden Layer:** reshapes the outputs of the previous layer to a one-dimensional array.
- **The 6th Hidden Layer:** is a fully-connected layer containing 1000 neurons with a *ReLu* activation. Dropout applied to this layer at a probability of 0.5.
- **The 7th Hidden Layer:** is a fully-connected layer containing 300 neurons with a *ReLu* activation.
- **The 8th Hidden Layer:** is a fully-connected layer containing 100 neurons with a *ReLu* activation.
- **The Output Layer:** is a 30 neurons layer with a sigmoid activation.

Figure C.12 in Appendix C provides a visual illustration of the architecture of the network.

7.3.2 Results

A sample of the output of FKDN1 is presented in Figures 7.1, 7.2, 7.3 and 7.4. In each sample, the predicted keypoint positions are marked with red crosses, while the correct positions are marked with green crosses. It is evident from the samples that FKDN1 has a very high prediction accuracy. The mean squared error for the predicted positions in regard to the correct positions for FKDN1 is 5.809.

7.4 Facial Keypoints Detection Network 2 (FKDN2)

7.4.1 Architecture

FKDN2 uses a deep fully-connected architecture with dropout employed throughout its layers to speed the training process. It consists of 6 layers including the input and output layers. The structures of the layers of FKDN2 are specified below:

- **The Input Layer:** is a one dimensional representation of the facial image, with dropout applied at a probability of 0.2.
- **The 1st Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 2nd Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.

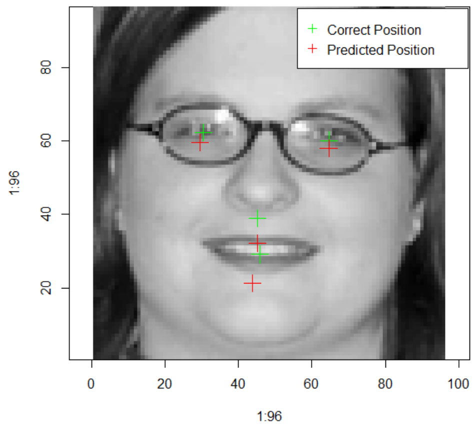


FIGURE 7.1: Sample 1 of the output of FKDN1.

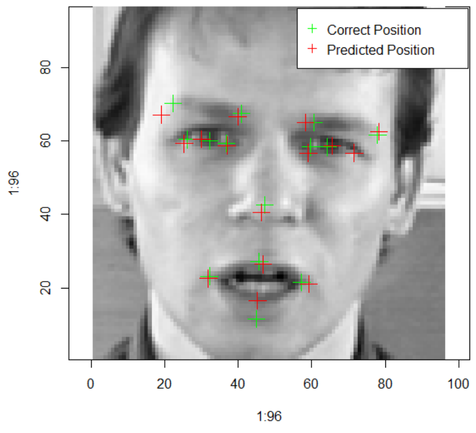


FIGURE 7.2: Sample 2 of the output of FKDN1.

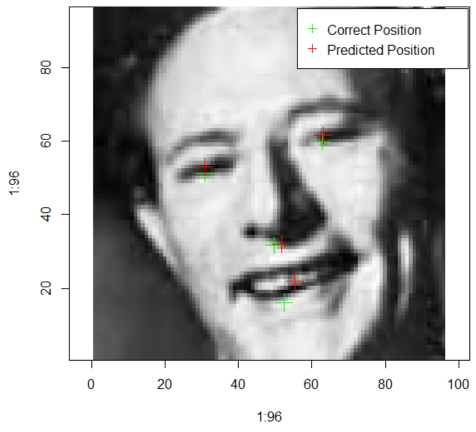


FIGURE 7.3: Sample 3 of the output of FKDN1.

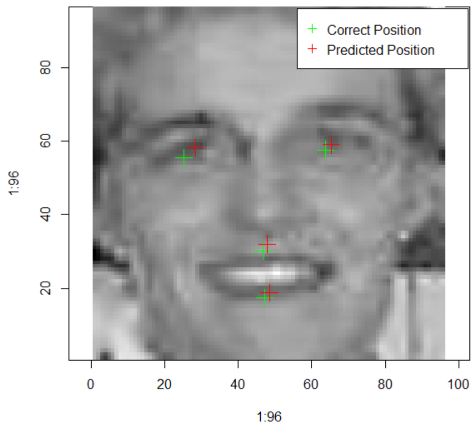


FIGURE 7.4: Sample 4 of the output of FKDN1.

- **The 3rd Hidden Layer:** is a fully-connected layer containing 500 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The 4th Hidden Layer:** is a fully-connected layer containing 100 neurons with a *ReLu* activation. Dropout is applied to this layer at a probability of 0.5.
- **The Output Layer:** is a single neuron layer with a sigmoid activation.

Figure C.13 in Appendix C provides a visual illustration of the architecture of the network.

7.4.2 Results

A sample of the output of FKDN2 is presented in Figures 7.5, 7.6, 7.7 and 7.8. In each sample, the predicted keypoint positions are marked with red crosses, while the correct positions are marked with green crosses. The samples show that FKDN2 has good accuracy; however, it falls short in comparison to FKDN1. The mean squared error for the predicted positions in regard to the correct positions for FKDN2 is 13.582.

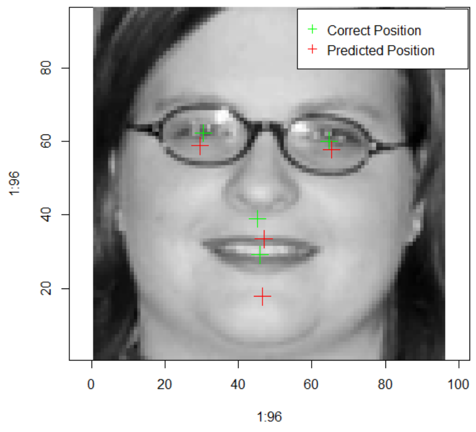


FIGURE 7.5: Sample 1 of the output of FKDN2.

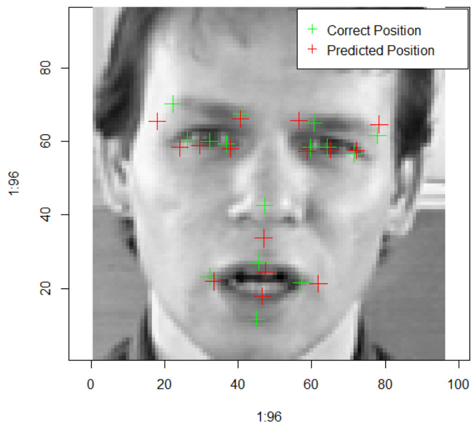


FIGURE 7.6: Sample 2 of the output of FKDN2.

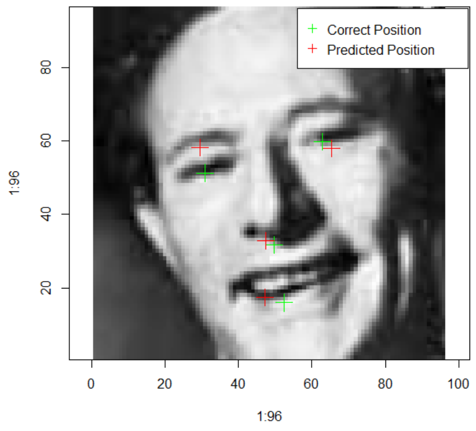


FIGURE 7.7: Sample 3 of the output of FKDN2.

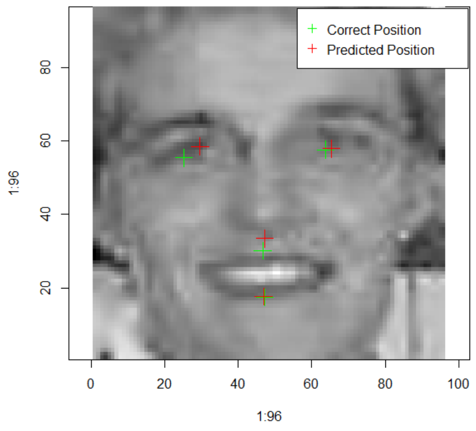


FIGURE 7.8: Sample 4 of the output of FKDN2.

7.5 Comparison of the Networks

Table 7.1 provides a brief comparison between FKDN1 and FKDN2. As mentioned before, FKDN1 is the better performing network. This is confirmed by the fact the its means squared error is less than that of FKDN1. The only advantage of FKDN2 over FKDN1 is the fact that, due to its fully-connected architecture, its training time is significantly quicker than that of FKDN1.

Network	Type	Mean Square Error
FKDN1	Convolutional	5.809
FKDN2	Fully-Connected	13.582

Table 7.1: Comparison table for the facial keypoints detection networks.

Chapter 8

Conclusion

It was possible to achieve state-of-the-art results for the five facial image processing tasks, gender classification, age prediction, non-linear dimensionality reduction, facial frontalisation and facial keypoint detection. The results demonstrated the strength of artificial neural networks, in particular that of convolutional neural networks. The fact that all of the results were obtained on a home laptop computer, using a readily available datasets, shows the practicality of artificial neural networks as a class of statistical learning algorithms. That being said, the availability of more powerful hardware and larger datasets could help in improving the results significantly, by allowing deeper architectures to be feasibly built.

Appendix A

Convergence of Gradient Descent

Theorem:
Let $f(\mathbf{x})$ be differentiable in $\mathbb{R}^n$, and let the gradient of $f(\mathbf{x})$ satisfy the Lipschitz condition:

$$||\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})||_2 \leq L||\mathbf{x} - \mathbf{y}||_2 \tag{A.1}$$

Also, let $f(\mathbf{x})$ be bounded below:

$$f(\mathbf{x}) \geq f^* > -\infty \tag{A.2}$$

And let α satisfy the condition:

$$0 < \alpha < \frac{2}{L} \tag{A.3}$$

Then, in (2.4), the gradient tends to zero:

$$\lim_{\tau \to \infty} \nabla f(\mathbf{x}^{(\tau)}) = 0$$

And the function monotonically decreases:

$$f(\mathbf{x}^{(\tau)}) \leq f(\mathbf{x}^{(\tau-1)})$$

The proof of this theorem is adapted from Polyak's Introduction to Optimization [48].

Proof. The following result is given without a proof:
If $f(\mathbf{x})$ is differentiable on $\mathbb{R}^n$ then:

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \mathbf{y} + \int_0^1 ((f(\mathbf{x}+\nu\mathbf{y}) - \nabla f(\mathbf{x})) \cdot \mathbf{y}) d\nu \tag{A.4}$$

Where is $\mathbf{a} \cdot \mathbf{b}$ is the dot product of $\mathbf{a}$ and $\mathbf{b}$.
From (A.4), substituting $\mathbf{x} = \mathbf{x}^{(\tau)}$ and $\mathbf{y} = -\alpha \nabla f(\mathbf{x}^{(\tau)})$, and applying (A.1):

$$\begin{aligned} f(\mathbf{x}^{(\tau+1)}) =& f(\mathbf{x}^{(\tau)}) - \alpha ||\nabla f(\mathbf{x}^{(\tau+)})||_2^2 \\ & - \alpha \int\limits_0^1 (\nabla f(\mathbf{x}^{(\tau)} - \nu\alpha\nabla f(\mathbf{x}^{(\tau)})) - \nabla f(\mathbf{x}^{(\tau)})) \cdot \nabla f(\mathbf{x}^{(\tau)}) d\nu \\ \leq & f(\mathbf{x}^{(\tau)}) - \alpha ||\nabla f(\mathbf{x}^{(\tau)})||_2^2 + L\alpha^2 ||f(\mathbf{x}^{(\tau)})||_2^2 \int\limits_0^1 \nu^2 d\nu \\ =& f(\mathbf{x}^{(\tau)}) - \alpha(1 - \frac{1}{2}L\alpha) ||f(\mathbf{x}^{(\tau)})||_2^2 \end{aligned}$$

Let $\lambda = \alpha(1 - \frac{1}{2}L\alpha)$ then:

$$f(\mathbf{x}^{(\tau+1)}) \leq f(\mathbf{x}^{(\tau)}) - \lambda ||f(\mathbf{x}^{(\tau)})||_2^2$$

Summing over τ from 0 to s:

$$f(\mathbf{x}^{(s+1)}) \leq f(\mathbf{x}^{(0)}) - \lambda \sum_{\tau=0}^{s} ||\nabla f(\mathbf{x}^{(\tau)})||_2^2$$

Since $\lambda > 0$ by (A.3):

$$\sum_{\tau=0}^{s} ||\nabla f(\mathbf{x}^{(\tau)})||_2^2 \leq \frac{f(\mathbf{x}^{(0)}) - f(\mathbf{x}^{(s+1)})}{\lambda} \leq \frac{f(\mathbf{x}^{(0)}) - f^*}{\lambda} \qquad \text{by (A.2)}$$

Hence, as $\lim_{s\to\infty} \sum_{\tau=0}^{s} ||\nabla f(\mathbf{x}^{(\tau)})||_2^2 = \sum_{\tau=0}^{\infty} ||\nabla f(\mathbf{x}^{(\tau)})||_2^2 < \infty$.
Yielding $||\nabla f(\mathbf{x}^{(\tau)})||_2 \to 0$, as $\tau \to \infty$. $\square$

Appendix B

Python Code for Building the Neural Networks

B.1 Gender Classification Networks

```python
from multicolumnNN import *

#Loading the Dataset
dataset=StaticFERETGenderDataset()

#Gender Classification Network 1
if False: #Toggle to True to use this network
        y=dataset.variables[1] #Labels
        x=InitialLayer(dataset,0) #Input Layer
        x=LeNetConvPoolLayer(x,20,5,2) #Convolution+Maxpool
        x=LeNetConvPoolLayer(x,50,5,2) #Convolution+Maxpool
        x=HiddenLayer(Flatten(x),500)
         ↪  #Flattening+Fully-Connected Layer
        nn=LogisticRegression(x,y,2)
         ↪  #Logistic Regression X-Entropy Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestClassification(learningRate=0.001,
         ↪  learningRateDecay=0, momentum=0.99, weightDecay=0,
         ↪  restart=True, savePredictions=True)#Training

#Gender Classification Network 2
if False: #Toggle to True to use this network
        y=dataset.variables[1] #Labels
        x=InitialLayer(dataset,0)
        x=Flatten(x) #Input Layer
        x=HiddenLayer(x,1000) #Fully-Connected Layer
        x=HiddenLayer(x,1000) #Fully-Connected Layer
        x=HiddenLayer(x,1000) #Fully-Connected Layer
        nn=LogisticRegression(x,y,2)
         ↪  #Logistic Regression X-Entropy Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestClassification(learningRate=0.0001,
         ↪  learningRateDecay=0, momentum=0.99, weightDecay=0,
         ↪  restart=False, savePredictions=True)#Training
```

```

#Gender Classification Network 3
if False: #Toggle to True to use this network
        y=dataset.variables[1] #Labels
        x=InitialLayer(dataset,0)
        x=DropoutLayer(Flatten(x),0.2) #Input Layer+Dropout
        x=DropoutLayer(HiddenLayer(x,1000),0.5)
        ↪  #Fully-Connected Layer+Dropout
        x=DropoutLayer(HiddenLayer(x,1000),0.5)
        ↪  #Fully-Connected Layer+Dropout
        x=DropoutLayer(HiddenLayer(x,1000),0.5)
        ↪  #Fully-Connected Layer+Dropout
        nn=LogisticRegression(x,y,2)
        ↪  #Logistic Regression X-Entropy Cost
        nag=NAGtrainer(dataset,nn)
        ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestClassification(learningRate=0.0001,
        ↪  learningRateDecay=0, momentum=0.99, weightDecay=0,
        ↪  restart=True, savePredictions=True)#Training
```

B.2 Age Prediction Networks

```
from multicolumnNN import *

#Loading the Dataset
dataset=StaticFERETAgeDataset()

#Age Prediction Network 1
if False: #Toggle to True to use this network
        x=InitialLayer(dataset,0)
        x=Flatten(x) #Input Layer
        x=DropoutLayer(x,0.2) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,100,f=T.nnet.sigmoid) #Fully-Connected Layer
        y=InitialLayer(dataset,1) #Labels
        nn=L2Distance(y,x,1,x) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestL2Distance(learningRate=0.1**4,momentum=0.99,
         ↪  weightDecay=0.01,restart=True, savePredictions = False)
         ↪  #Training

#Age Prediction Network 2
if False: #Toggle to True to use this network
        x=InitialLayer(dataset,0) #Input Layer
        x=LeNetConvPoolLayer(x,20,5,2)  #Convolution+Maxpool
        x=Flatten(x) #Flatenning Layer
        x=HiddenLayer(x,300,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,300,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,100,f=T.nnet.sigmoid) #Fully-Connected Layer
        y=InitialLayer(dataset,1) #Labels
        nn=L2Distance(y,x,1,x) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn) #Nestrov Accelerated Gradient Descent
        nag.TrainTestL2Distance(learningRate=0.1**3,momentum=0.99, weightDecay=0,restart=True , savePredictions = True) #Training

#Age Prediction Network 3
if False: #Toggle to True to use this network
        x=InitialLayer(dataset,0) #Input Layer
        x=LeNetConvPoolLayer(x,10,5,2) #Convolution+Maxpool
        x=LeNetConvPoolLayer(x,20,4,2) #Convolution+Maxpool
        x=Flatten(x) #Flatenning Layer
        x=HiddenLayer(x,1000,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,300,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,100,f=T.nnet.sigmoid) #Fully-Connected Layer
        y=InitialLayer(dataset,1) #Labels
        nn=L2Distance(y,x,1,x) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn) #Nestrov Accelerated Gradient Descent
        nag.TrainTestL2Distance(learningRate=0.1**4,momentum=0.99, weightDecay=0.01,restart=True, savePredictions = True) #Training
```

B.3 Dimensionality Reduction Networks

```
from multicolumnNN import *

#Loading the Dataset
dataset=StaticFERETDataset(batchSize=100)

#Dimensionality Reduction Network 1
if False: #Toggle to True to use this network
        a=InitialLayer(dataset,0) #Input Layer
        b=LeNetConvPoolLayer(a,20,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,40,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,60,2,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,200,3,1)#Convonvolution
        c=LeNetUnConvPoolLayerC(b,60,3,1)#Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c,40,2,2)
         ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,20,3,2)
         ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,1,3,2,T.tanh)
         ↪  #Reverse Convonvolution+Maxpool
        nn=L2Distance(a,c,1,a) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestEmbedding(b,0.001,momentum=0.99,weightDecay=0.01
         ↪  ,restart=True, saveReconstruction = True) #Training

#Dimensionality Reduction Network 2
if False: #Toggle to True to use this network
        a=InitialLayer(dataset,0) #Input Layer
        b=LeNetConvPoolLayer(a,30,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,60,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,80,2,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,100,2,1)#Convonvolution
        c=LeNetUnConvPoolLayerC(b,80,2,1)#Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c,60,2,2)
         ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,30,3,2) #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,1,3,2,T.tanh) #Reverse Convonvolution+Maxpool
        nn=L2Distance(a,c,1,a) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn) #Nestrov Accelerated Gradient Descent
        nag.TrainTestEmbedding(b,0.001,momentum=0.99,weightDecay=0.01,restart=True, saveReconstruction = True) #Training

#Dimensionality Reduction Network 3
if False: #Toggle to True to use this network
        a=InitialLayer(dataset,0) #Input Layer
        b=LeNetConvPoolLayer(a,20,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,40,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,60,2,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,200,3,1)#Convonvolution
        b=LeNetConvPoolLayer(b,400,3,1)#Convonvolution
        c=LeNetUnConvPoolLayerC(b,200,3,1) #Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c, 60,3,1)#Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c,40,2,2) #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,20,3,2) #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,1,3,2,T.tanh) #Reverse Convonvolution+Maxpool
        nn=L2Distance(a,c,1,a) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn) #Nestrov Accelerated Gradient Descent
        nag.TrainTestEmbedding(b,0.001,momentum=0.99,weightDecay=0.01,restart=True, saveReconstruction = True) #Training
```

B.4 Frontal Pose Reconstruction Networks

```
from multicolumnNN import *

#Loading the Dataset
dataset=StaticFERETDataset(batchSize=100)

#Frontal Pose Reconstruction Network 1
if False: #Toggle to True to use this network
        a=InitialLayer(dataset,0) #Input Layer
        d=InitialLayer(dataset,1) #Labels
        b=LeNetConvPoolLayer(a,20,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,40,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,60,2,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,200,3,1) #Convolution
        c=LeNetUnConvPoolLayerC(b,60,3,1) #Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c,40,2,2)
         ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,20,3,2)
         ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,1,3,2,T.tanh)
         ↪  #Reverse Convonvolution+Maxpool
        nn=L2Distance(d,c,1,a) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestEmbedding(b,0.01,momentum=0.99,weightDecay=0.01,
         ↪  restart=False) #Training

#Frontal Pose Reconstruction Network 2
if False: #Toggle to True to use this network
        a=InitialLayer(dataset,0) #Input Layer
        d=InitialLayer(dataset,1) #Labels
        b=LeNetConvPoolLayer(a,20,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,40,3,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,60,2,2) #Convonvolution+Maxpool
        b=LeNetConvPoolLayer(b,200,2,1) #Convonvolution
        c=LeNetUnConvPoolLayerC(b,60,2,1) #Reverse Convonvolution
        c=LeNetUnConvPoolLayerC(c,40,2,2)
        ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,20,3,2)
        ↪  #Reverse Convonvolution+Maxpool
        c=LeNetUnConvPoolLayerC(c,1,3,2,T.tanh)
        ↪  #Reverse Convonvolution+Maxpool
        nn=L2Distance(d,c,1,a) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
        ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestEmbedding(b,0.01,momentum=0.99,weightDecay=0.01,
        ↪  restart=False) #Training
```

B.5 Facial Keypoint Detection Networks

```
from multicolumnNN import *

#Loading the Dataset
dataset=StaticFacialDataset()

#Facial Keypoint Detection Network 1
#Good! Number 15 on Kaggle 6th April 2015!!!!
if True: #Toggle to True to use this network
        x=InitialLayer(dataset,0) #Input Layer
        x=LeNetConvPoolLayer(x,10,5,2) #Convolution+Maxpool
        x=LeNetConvPoolLayer(x,20,5,2) #Convolution+Maxpool
        x=Flatten(x) #Flatenning Layer
        x=HiddenLayer(x,1000,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,300,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,100,f=ReLu) #Fully-Connected Layer
        x=HiddenLayer(x,30,f=T.nnet.sigmoid) #Fully-Connected Layer
        y=InitialLayer(dataset,1) #Labels
        nn=L2Distance(y,x,0.01,x) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestL2Distance(learningRate=0.0**6,momentum=0.99,
         ↪  weightDecay=0.01,restart=True, savePredictions = True)
         ↪  #Training

#Facial Keypoint Detection Network 2
if False: #Toggle to True to use this network
        x=InitialLayer(dataset,0)
        x=Flatten(x) #Input Layer
        x=DropoutLayer(x,0.2) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,500,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,100,f=ReLu) #Fully-Connected Layer
        x=DropoutLayer(x,0.5) #Dropout
        x=HiddenLayer(x,30,f=T.nnet.sigmoid) #Fully-Connected Layer
        y=InitialLayer(dataset,1) #Labels
        nn=L2Distance(y,x,0.01,x) #L2 Distance Cost
        nag=NAGtrainer(dataset,nn)
         ↪  #Nestrov Accelerated Gradient Descent
        nag.TrainTestL2Distance(learningRate=0.1**6,momentum=0.99,
         ↪  weightDecay=0.01,restart=True, savePredictions = True)
         ↪  #Training
```

Appendix C

Visual Representations of Neural Network Architectures

C.1 Gender Classification Networks

C.1.1 GCN1

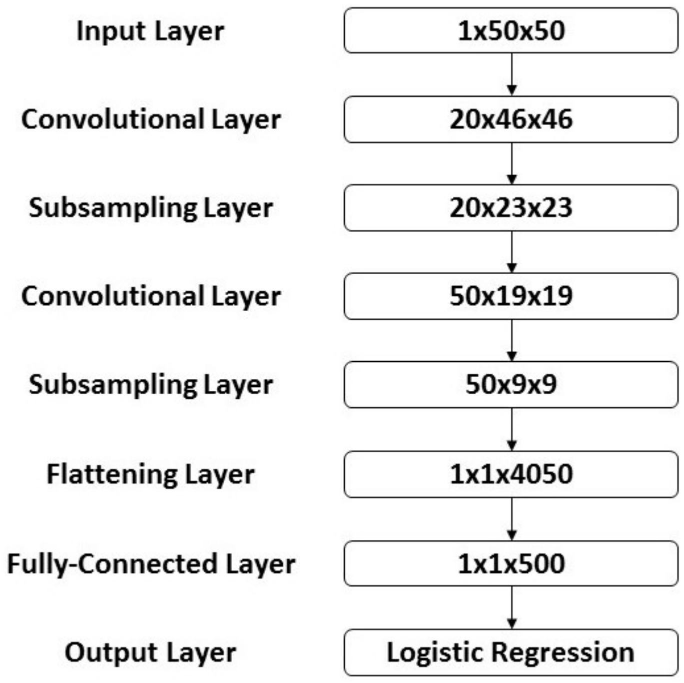


Figure C.1: Network architecture for GCN1.

C.1.2 GCN2

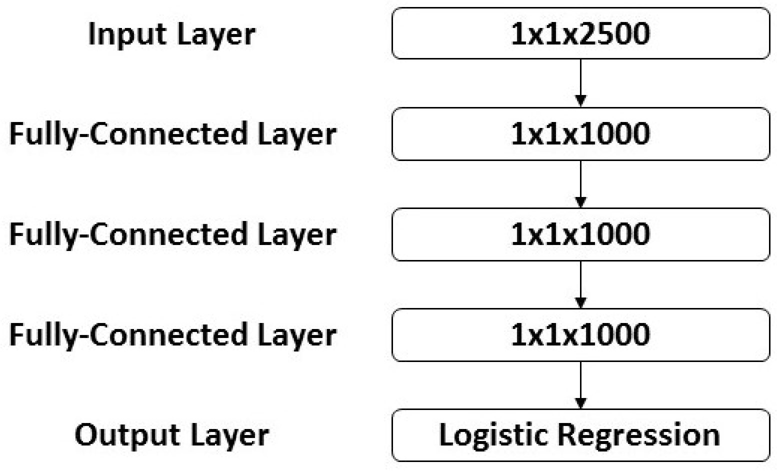


FIGURE C.2: Network architecture for GCN2.

C.1.3 GCN3

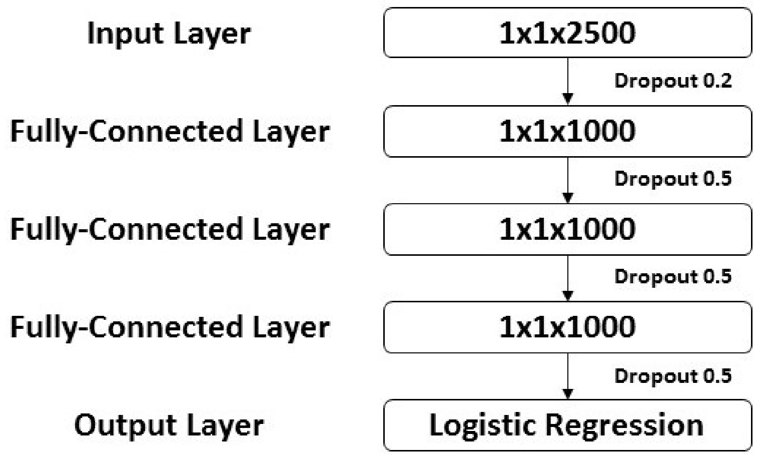


FIGURE C.3: Network architecture for GCN3.

C.2 Age Prediction Networks

C.2.1 APN1

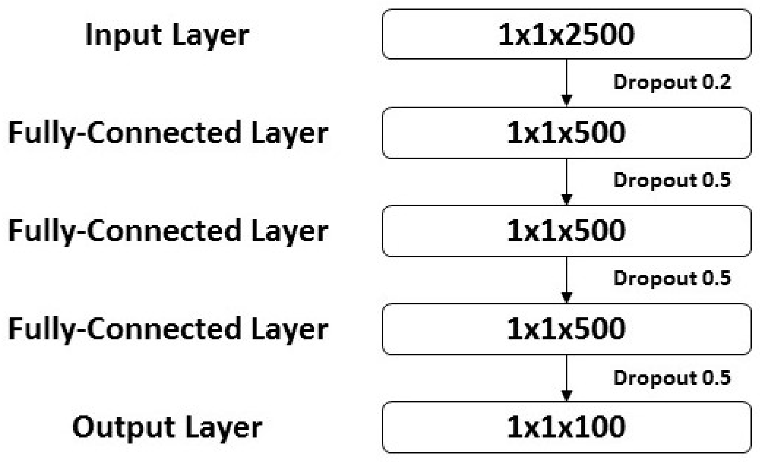


FIGURE C.4: Network architecture for APN1.

C.2.2 APN2

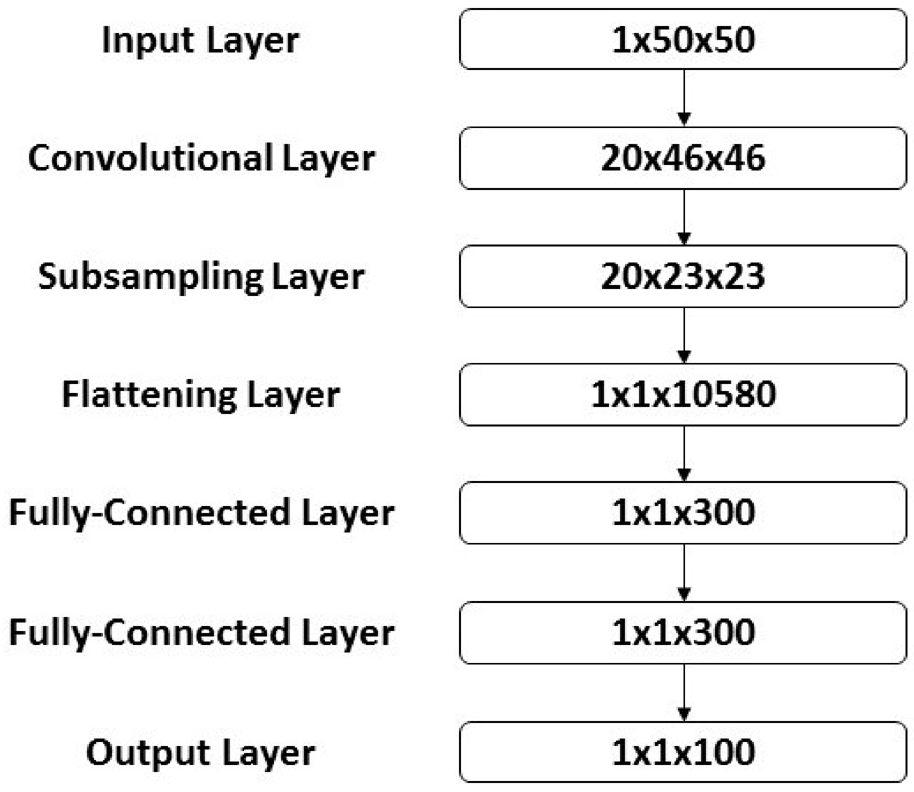


FIGURE C.5: Network architecture for APN2.

C.2.3 APN3

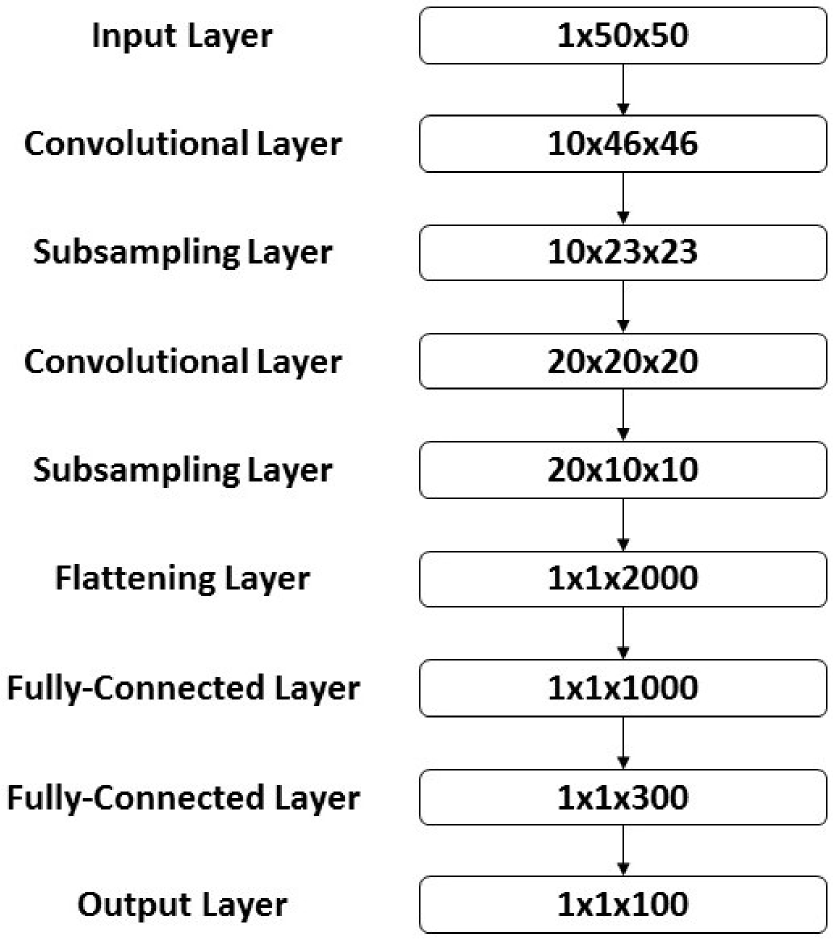


FIGURE C.6: Network architecture for APN3.

C.3 Dimensionality Reduction Networks

C.3.1 DRN1

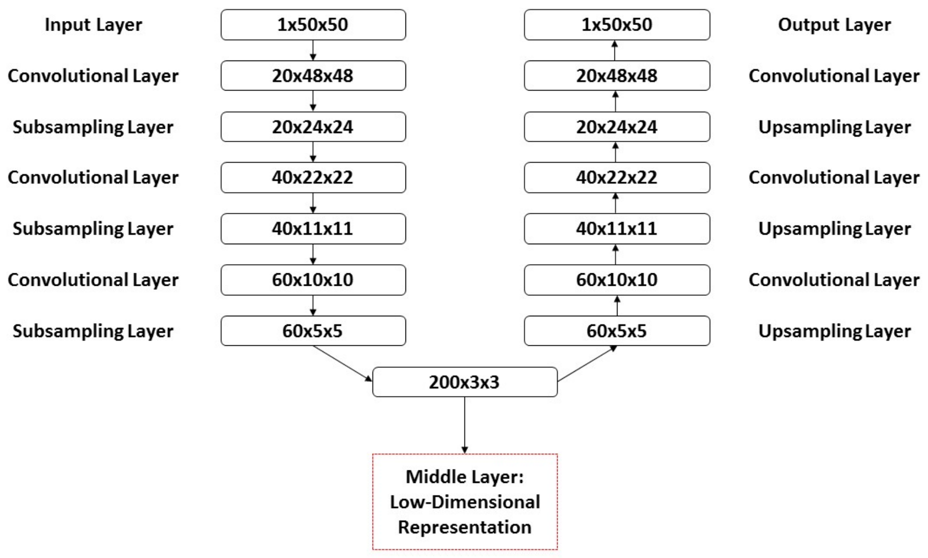


FIGURE C.7: Network architecture for DRN1.

C.3.2 DRN2

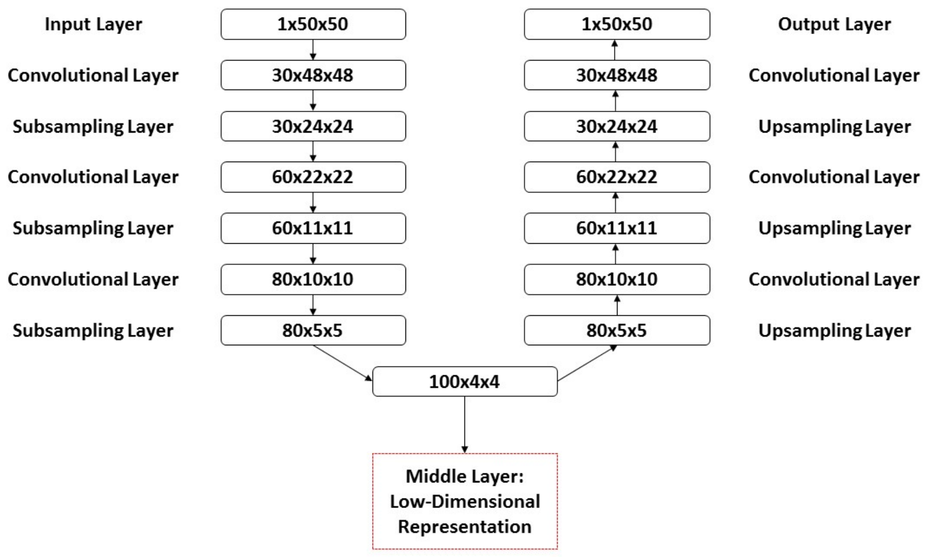


FIGURE C.8: Network architecture for DRN2.

C.3.3 DRN3

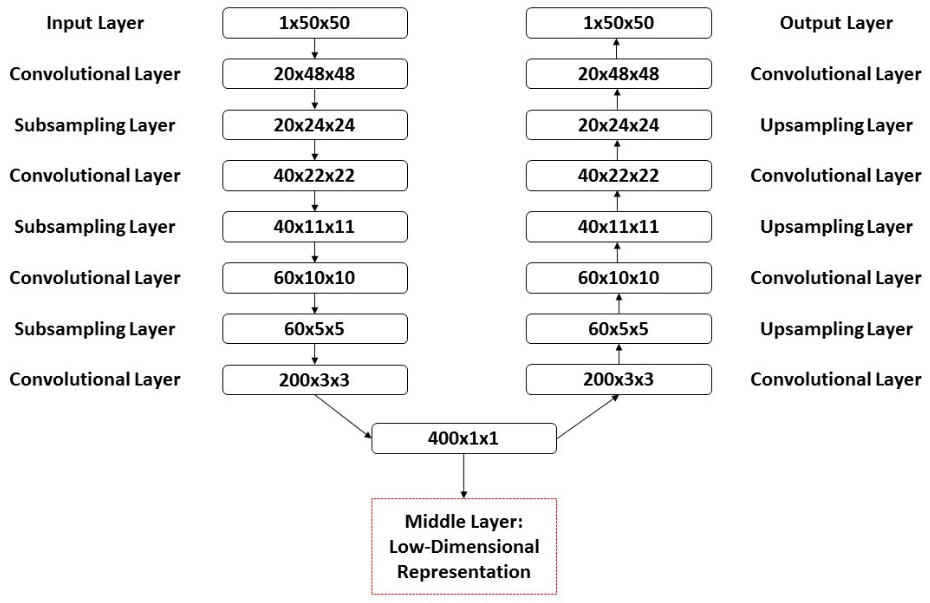


FIGURE C.9: Network architecture for DRN3.

C.4 Frontal Pose Reconstruction Networks

C.4.1 FPRN1

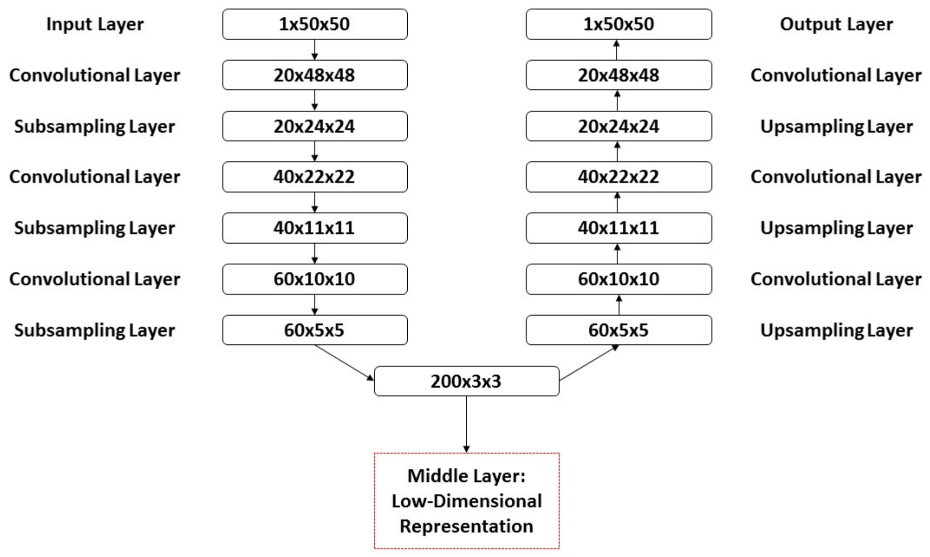


FIGURE C.10: Network architecture for FPRN1.

C.4.2 FPRN2

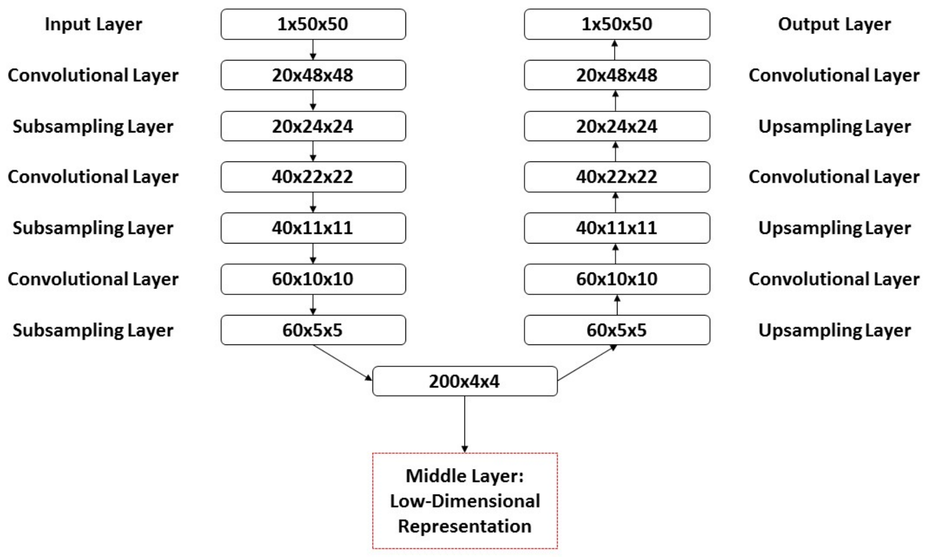


Figure C.11: Network architecture for FRN2.

C.5 Facial Keypoint Detection Networks

C.5.1 FKDN1

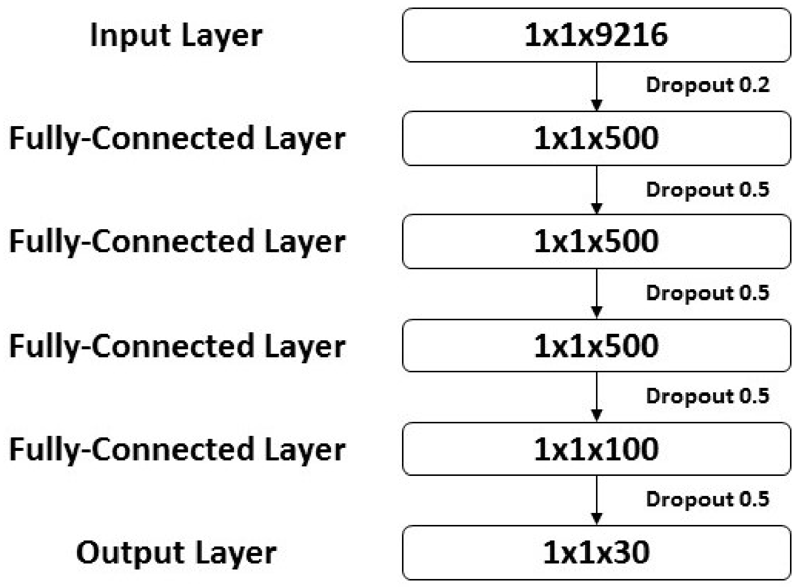


Figure C.12: Network architecture for FKDN1.

C.5.2 FKDN2

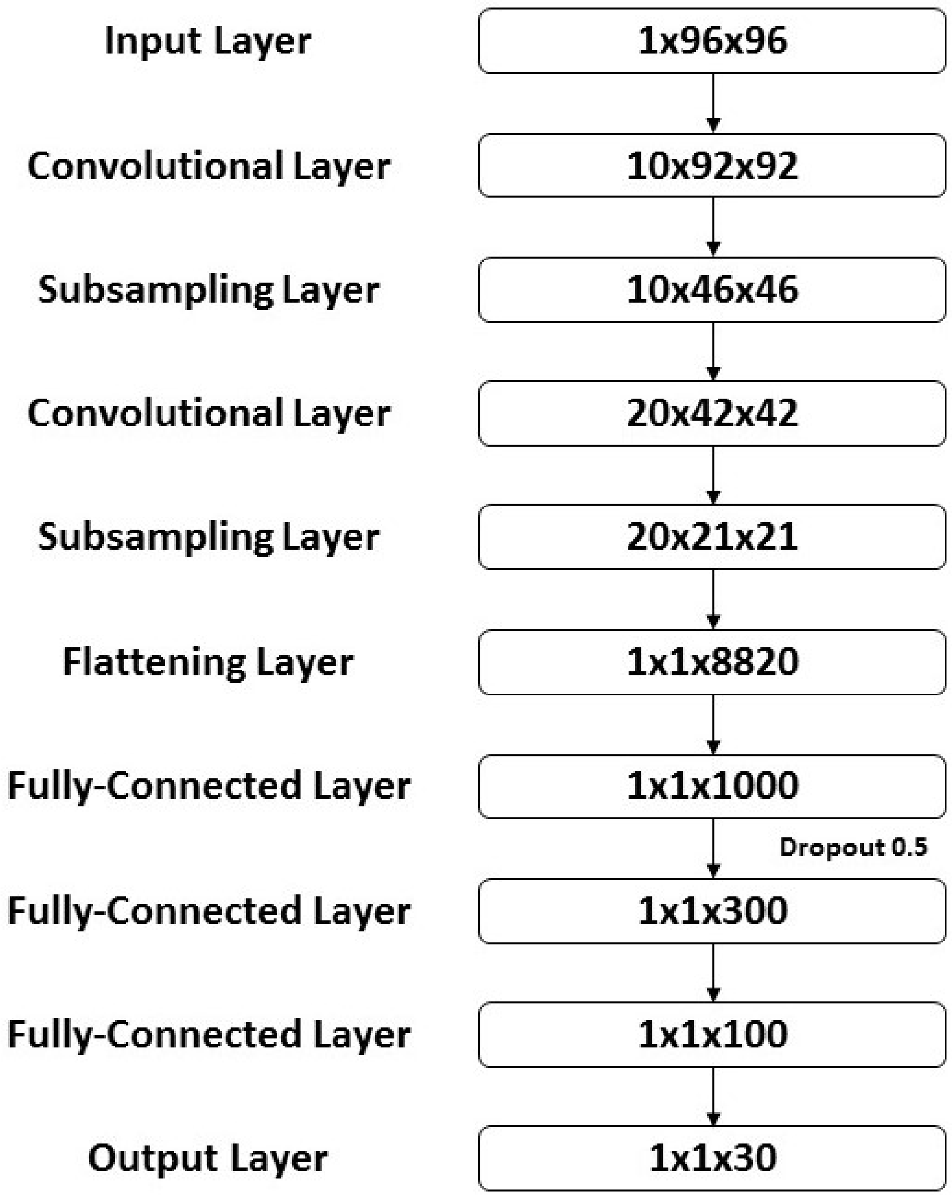


Figure C.13: Network architecture for FKDN2.

Appendix D

Samples from the Dimensionality Reduction Networks

D.1 DRN1

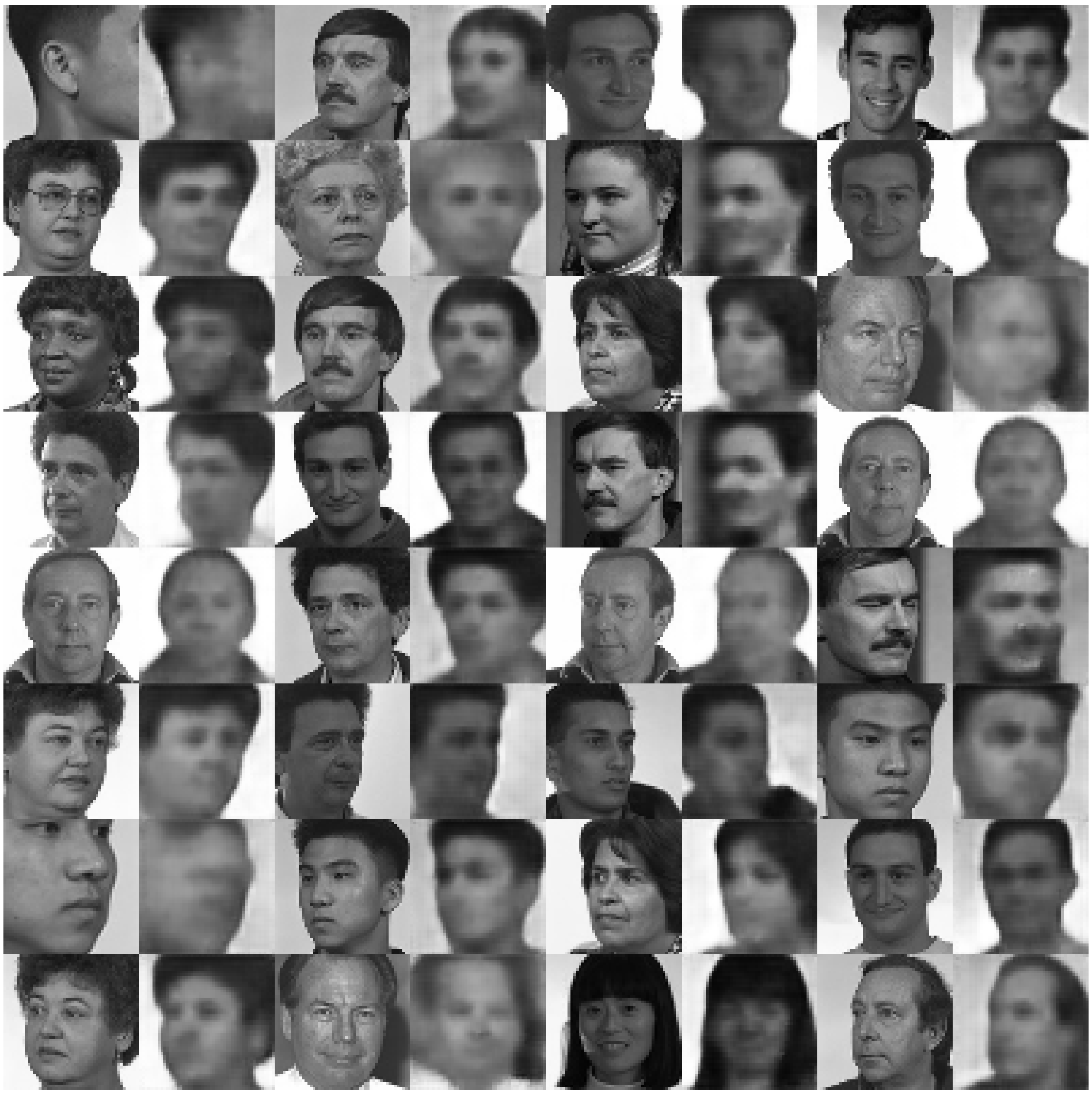


FIGURE D.1: A sample of the original images compared to the reconstructed images for DRN1. In each pair of columns, the images on the right are the originals and the images on the left are the reconstruction.

D.2 DRN2

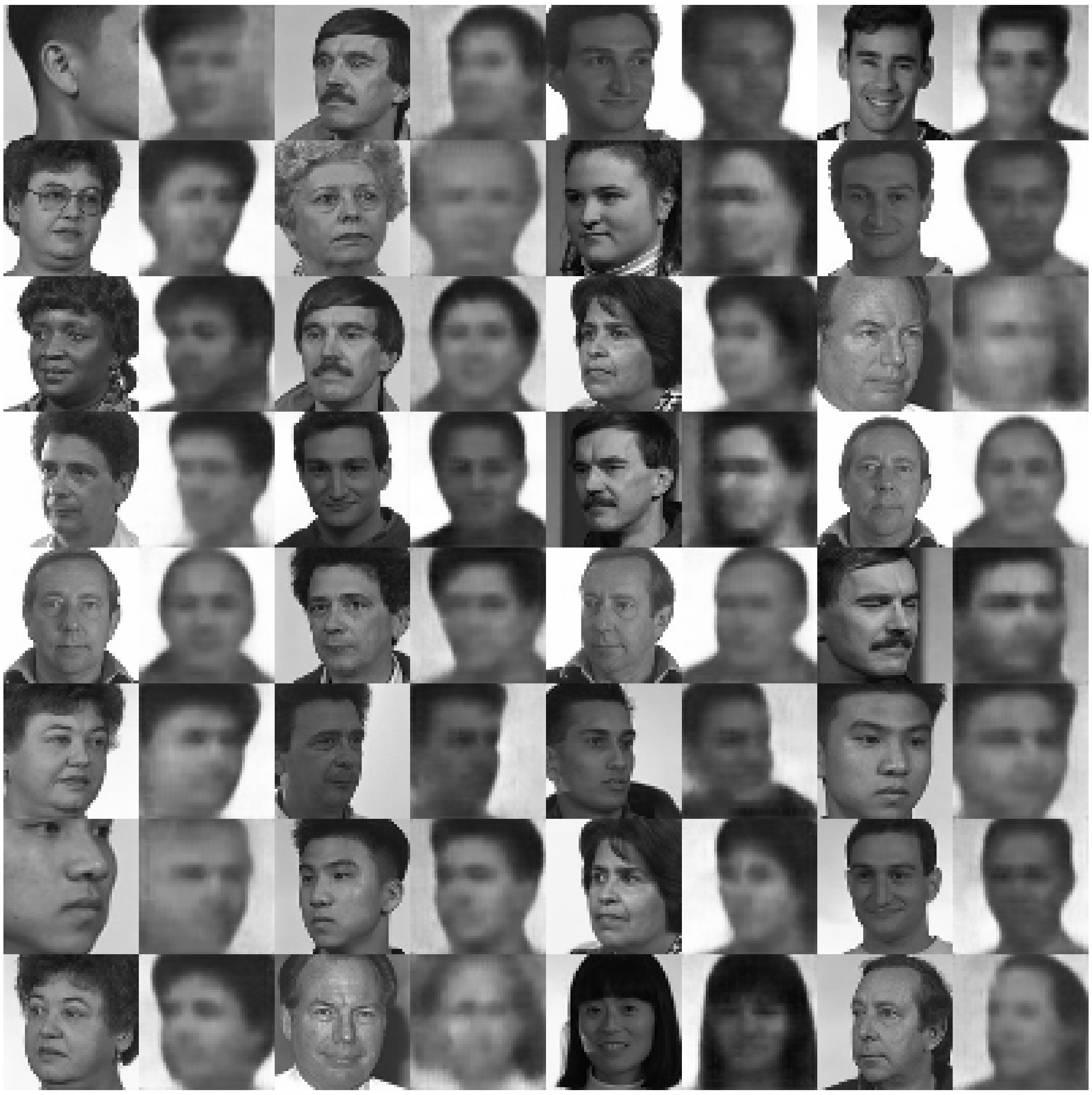


FIGURE D.2: A sample of the original images compared to the reconstructed images for DRN2. In each pair of columns, the images on the right are the originals and the images on the left are the reconstruction.

D.3 DRN3

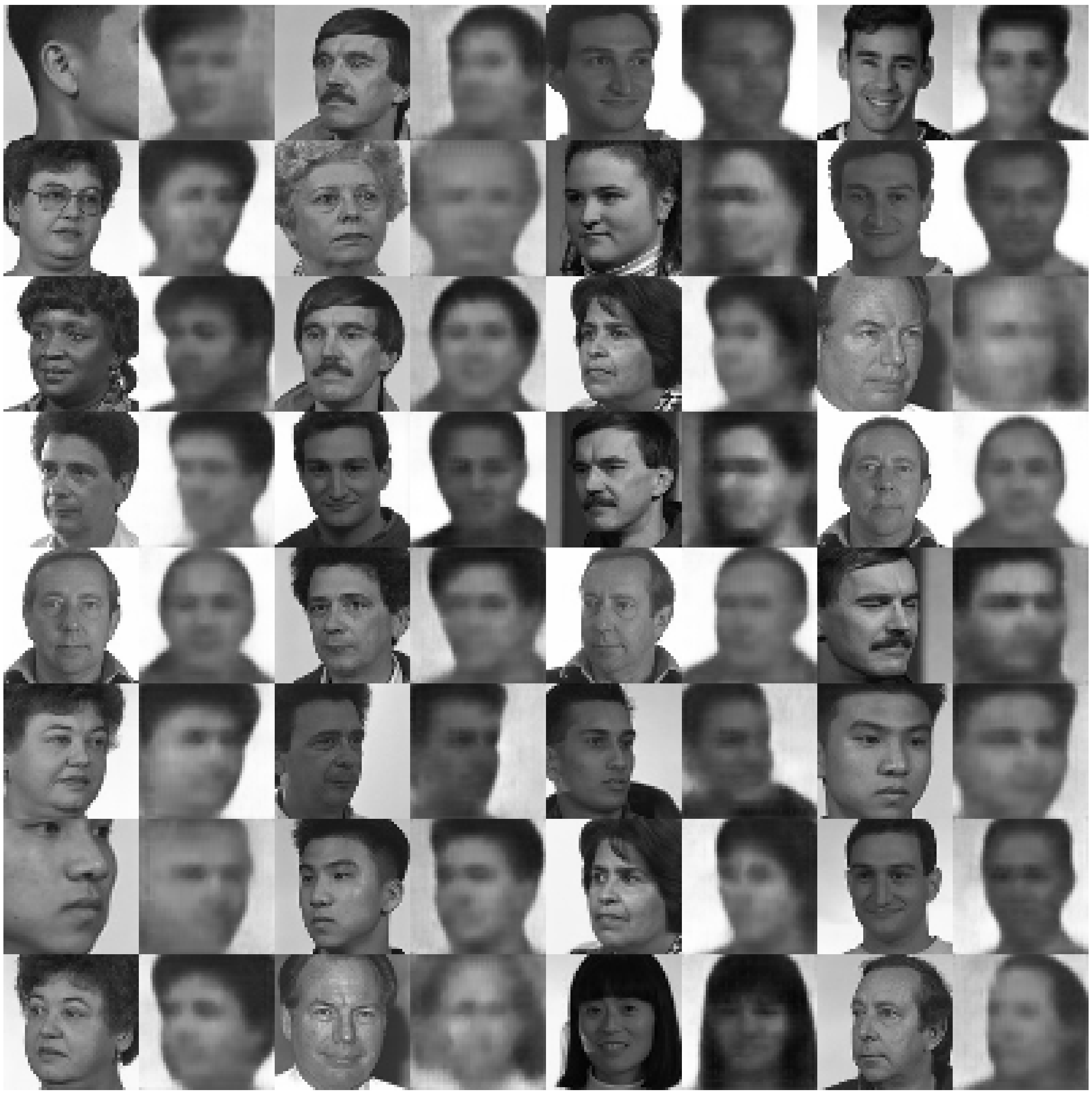


FIGURE D.3: A sample of the original images compared to the reconstructed images for DRN3. In each pair of columns, the images on the right are the originals and the images on the left are the reconstruction.

Appendix E

Samples from the Frontal Pose Reconstruction Networks

E.1 FPRN1

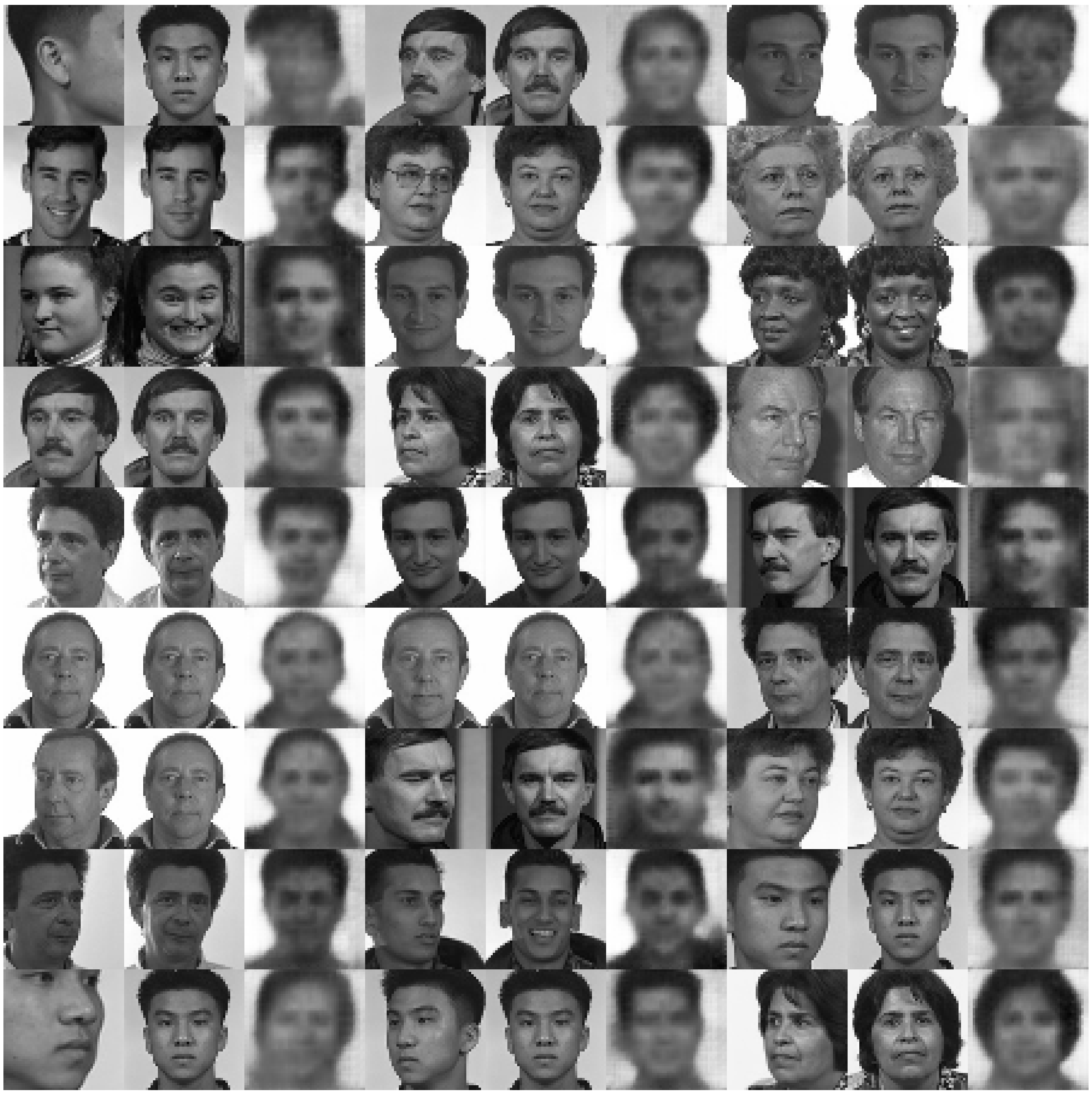


FIGURE E.1: A sample comparing the original images to the reconstructed images for FPRN1.

E.2 FPRN2

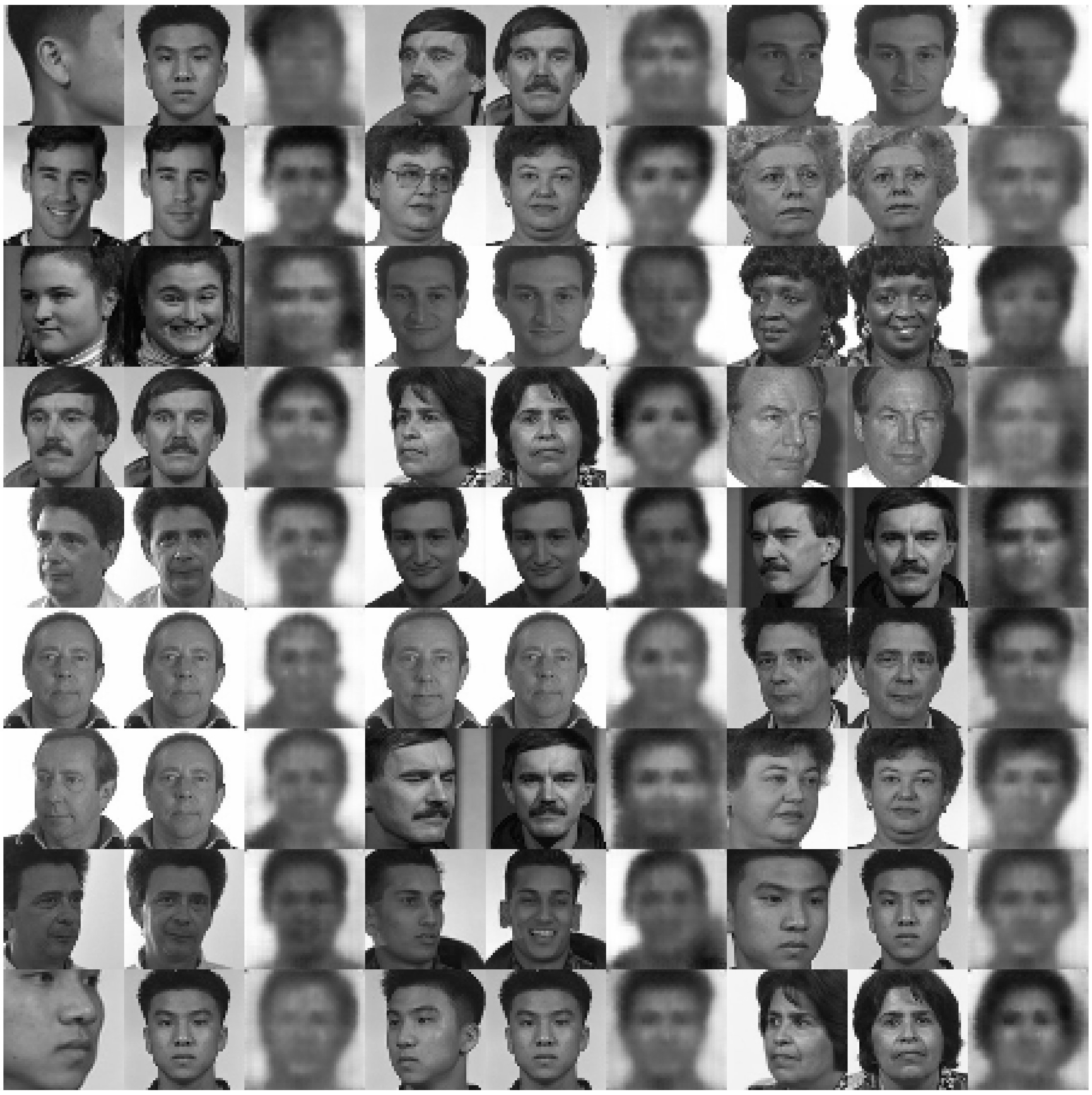


FIGURE E.2: A sample comparing the original images to the reconstructed images for FPRN2.

Appendix F

R Code for the Analysis of the Results

F.1 Gender Classification Networks

```r
#Functions
usePackage <- function(p) {
        if (!is.element(p, installed.packages()[,1]))
                install.packages(p, dep = TRUE)
        require(p, character.only = TRUE)
}

#Loading the data
data <- read.csv(file = 'predictions.csv', header = FALSE, col.names=
 ↪  c('predicted', 'real'))
attach(data)

#Barplots
par(mfrow = c(1,2))
bp1 <- barplot(table(predicted), names.arg = c('Female', 'Male'),
 ↪  main = 'Bar Plot of Predicted Gender', xlab='Gender', col=c(
 ↪  'deepskyblue3', 'firebrick'))
text(bp1, 80, round(table(predicted), 1),cex=2,pos=3)
bp2 <- barplot(table(real), names.arg = c('Female', 'Male'), main =
 ↪  'Bar Plot of Actual Gender', xlab='Gender', col=c('deepskyblue3',
 ↪  'firebrick'))
text(bp2, 80, round(table(real), 1),cex=2,pos=3)
dev.off()

#Confusion
usePackage('caret') #UserDefined
confusion <- confusionMatrix(predicted, real)
confusion$table

#Misclassification
mis <- sum(abs(real-predicted))
mis/length(real)*100
100-mis/length(real)*100
```

F.2 Age Prediction Networks

```r
#Functions
usePackage <- function(p) {
        if (!is.element(p, installed.packages()[,1]))
                install.packages(p, dep = TRUE)
        require(p, character.only = TRUE)
}

#Loading the data
data <- read.csv(file = 'comp.csv', header = FALSE, col.names= c(
 ↪  'predicted', 'real', 'abs.diff'))
attach(data)

#Scatter
plot(real, predicted, xlab = 'Real Age', ylab = 'Predicted Age', main
 ↪  = 'Real vs Predicted Age'
     , xlim = c(10,70), ylim = c(10,50))
abline(0, 1, col='red')

#Box Plot
boxplot(predicted, real, names=c('Predicted', 'Real'), ylab='Age',
 ↪  main='Box Plot of Real and Predicted Age')
abline(,0,30, col='red', lty=2)

#Histograms
par(mfrow = c(1,2))
hist(predicted, main = 'Histogram of Predicted Age', xlab='Age',
 ↪  probability = TRUE)
lines(density(predicted), col='red', lty=2)
hist(real, main = 'Histogram of Real Age', xlab='Age', , probability
 ↪  = TRUE)
lines(density(real), col='red', lty=2)
dev.off()

#Linear Model
model <- lm(real~predicted)
summary(model)
```

```r
coef(model)
plot(predicted, real, xlab = 'Predicted Age', ylab = 'Read Age', main
  ↪  = 'Real vs Predicted Age'
      , xlim = c(18,50), ylim = c(10,70))
abline(model, col='red')
abline(0,1, col ='blue', lty=2)
legend(43, 20, c('Linear Model', 'Y=X') ,lty = c(1,2), col = c('red',
  ↪  'blue'))

#Correlation
cor(real,predicted)

#Mean
mean(real)
mean(predicted)

#Variance
var(real)
var(predicted)

#Confusion Matrix
binned.real = cut(real, breaks=c(0, 20, 25, 30, 35, 40, 45, 50, 100))
binned.predicted = cut(predicted, c(0, 20, 25, 30, 35, 40, 45, 50,
  ↪  100))

usePackage('caret') #UserDefined
confusion <- confusionMatrix(binned.predicted, binned.real)
confusion$table
```

F.3 Dimensionality Reduction Networks

```r
#Loading the data

recon <- read.csv(file = 'output.csv', header = FALSE)
labels <- read.csv(file = 'labels.csv', header = FALSE)

#MSE

diff <- recon - labels
diff2 <- diff^2
mse <- rowSums(diff2)/ncol(diff2)

#Peak Signal to Noise Ratio

psnr <- 10*log10(1/mse)

#Analysis of Results
mean(psnr)
var(psnr)
hist(psnr, main='Density of PSNR for Reconstructed Faces', xlab=
↪  'Peak Signal to Noise Ratio', probability = TRUE)
lines(density(psnr), col='red', lty=2)
boxplot(psnr, ylab='PSNR Value', main=
↪  'Box Plot for the PSNR for the Reconstructed Faces')
```

F.4 Facial Keypoint Detection Networks

```r
#Functions
usePackage <- function(p) {
        if (!is.element(p, installed.packages()[,1]))
                install.packages(p, dep = TRUE)
        require(p, character.only = TRUE)
}

#Loading Data
labels <- read.csv(file = 'labels.csv', header = FALSE)*96
x.lab <- labels[, seq(1,30,by=2)]
y.lab <- labels[, seq(2,30,by=2)]

predictions <- read.csv(file = 'predictions.csv', header = FALSE)*96
x.pred <- predictions[, seq(1,30,by=2)]
y.pred <- predictions[, seq(2,30,by=2)]

#samples
usePackage('png')

sample1 <- t(matrix(rev(readPNG('image1.png')), 96, 96))

image(1:96, 1:96, sample1, col=gray((0:255)/255))

points(96 - x.lab[624,], 96 - y.lab[624,], col = 'green', pch=3)
points(96 - x.pred[624, -which(is.na(x.lab[624,]))], 96 - y.pred[624,
 ↪  -which(is.na(x.lab[624,]))], col = 'red', pch=3)
legend('bottomright', pch = c(3,3), c('Correct Position',
 ↪  'Predicted Position'), col = c('green', 'red'))

sample2 <- t(matrix(rev(readPNG('image2.png')), 96, 96))

image(1:96, 1:96, sample2, col=gray((0:255)/255))

points(96 - x.lab[848,], 96 - y.lab[848,], col = 'green', pch=3)
points(96 - x.pred[848,], 96 - y.pred[848,], col = 'red', pch=3)
```

```r
legend('bottomright', pch = c(3,3), c('Correct Position',
 ↪  'Predicted Position'), col = c('green', 'red'))

sample3 <- t(matrix(rev(readPNG('image3.png')), 96, 96))

image(1:96, 1:96, sample3, col=gray((0:255)/255))

points(96 - x.lab[254,], 96 - y.lab[254,], col = 'green', pch=3)
points(96 - x.pred[254, -which(is.na(x.lab[254,]))], 96 - y.pred[254,
 ↪  -which(is.na(x.lab[254,]))], col = 'red', pch=3)
legend('bottomright', pch = c(3,3), c('Correct Position',
 ↪  'Predicted Position'), col = c('green', 'red'))

sample4 <- t(matrix(rev(readPNG('image4.png')), 96, 96))

image(1:96, 1:96, sample4, col=gray((0:255)/255))

points(96 - x.lab[590,], 96 - y.lab[590,], col = 'green', pch=3)
points(96 - x.pred[590, -which(is.na(x.lab[590,]))], 96 - y.pred[590,
 ↪  -which(is.na(x.lab[590,]))], col = 'red', pch=3)
legend('bottomright', pch = c(3,3), c('Correct Position',
 ↪  'Predicted Position'), col = c('green', 'red'))

#Mean Squared Error
mse <- mean((predictions - labels)^2, na.rm=TRUE)
```

Bibliography

[1] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R (Springer Texts in Statistics)*. Springer, 1 edition, 2013.

[2] L. Breiman. Statistical modeling: The two cultures. *Statistical Science*, 16(3): 199–231, 2001.

[3] R. Pearl, L. J. Reed, and J. F. Kish. The logistic curve and the census count of 1940. *Science*, 92(2395):486–488, 1940.

[4] J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):pp. 370–384, 1972. URL `http://www.jstor.org/stable/2344614`.

[5] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[6] T. Hastie and R. Tibshirani. Generalized additive models. *Statistical Science*, 1(3): pp. 297–310, 1986. URL `http://www.jstor.org/stable/2245459`.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.

[8] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[9] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1 edition, 1995.

[10] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[11] D. O. Hebb. *The Organization of Behavior*. Wiley, 1949.

[12] S. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach)*. Prentice-Hall, Inc., 1 edition, 1995.

[13] L.N. Kanal. Perceptrons. In Neil J. Smelser P. B. Baltes, editor, *International Encyclopedia of the Social & Behavioral Sciences*, pages 11218–11221. Pergamon, Oxford, 2001.

[14] L. Minsky and S.A. Papert. *Perceptrons*. MIT Press, 1988.

[15] P.J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. A Wiley Interscience Publication. Wiley, 1994.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

[17] G. E Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[18] Y. LeCun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages 97–104. IEEE, 2004.

[19] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[20] J. Gao. Machine learning applications for data center optimization. `http://research.google.com/pubs/pub42542.html`, 20. Accessed: 2014-03-27.

[21] Lung cancer classification using neural networks for {CT} images. *Computer Methods and Programs in Biomedicine*, 113(1):202–209, 2014.

[22] K. Fukui and O. Yamaguchi. Face recognition using multi-viewpoint patterns for robot vision. In *Robotics Research. The Eleventh International Symposium*, pages 192–201. Springer, 2005.

[23] F. Alonso-Martín, M. Malfaz, J. Sequeira, J. F. Gorostiza, and M. A. Salichs. A multimodal emotion detection system during human–robot interaction. *Sensors*, 13 (11):15549–15581, 2013.

[24] J.S. Coffin and D. Ingram. Facial recognition system for security access and identification, 1999. URL http://www.google.com/patents/US5991429. US Patent 5,991,429.

[25] V. N. Vapnik. An overview of statistical learning theory. *Neural Networks, IEEE Transactions on*, 10(5):988–999, 1999.

[26] A. NG. Machine learning, 2011. URL https://class.coursera.org/ml-006.

[27] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks*, 3361:310, 1995.

[28] Convolutional neural networks (lenet), 2015. URL http://deeplearning.net/tutorial/lenet.html.

[29] Linear neural networks. http://www.willamette.edu/~gorr/classes/cs449/linear2.html. Accessed: 2015-03-30.

[30] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[31] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[33] D. Shirkey and S.R. Gupta. An image mining system for gender classification & age prediction based on facial features. *IOSR Journal of Computer Engineering*, 10 (6):21–29, 2013.

[34] Phillips, Harry Wechsler, Jeffery Huang, and Patrick J. Rauss. The FERET database and evaluation procedure for face-recognition algorithms. *Image and Vision Computing*, 16(5):295–306, 1998.

[35] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010. Oral Presentation.

[36] Y. H. Kwon and N. da Vitoria Lobo. Age classification from facial images. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 762–767. IEEE, 1994.

[37] H. Fukai, Y. Nishie, K. Abiko, Y. Mitsukura, M. Fukumi, and M. Tanaka. An age estimation system on the aibo. In *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, pages 2551–2554, 2008.

[38] S. Thakur and L. Verma. Identification of face age range group using neural network. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):250–254, 2012.

[39] I. K Fodor. A survey of dimension reduction techniques, 2002.

[40] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[41] T. Bourlai, A. Ross, and A. K. Jain. Restoring degraded face images: A case study in matching faxed, printed, and scanned photos. *Information Forensics and Security, IEEE Transactions on*, 6(2):371–384, 2011.

[42] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1701–1708. IEEE, 2014.

[43] X. Cao, Y. Wei, F. Wen, and J. Sun. Face alignment by explicit shape regression. *International Journal of Computer Vision*, 107(2):177–190, 2014.

[44] R. Herpers, M. Michaelis, K. Lichtenauer, and G. Sommer. Edge and keypoint detection in facial regions. In *In Killington, VT*, pages 212–217. IEEE Computer Society Press, 1996.

[45] C. Creusot, N. Pears, and J. Austin. A machine-learning approach to keypoint detection and landmarking on 3d meshes. *International Journal of Computer Vision*, 102(1-3):146–179, 2013.

[46] D. Nouri. Using convolutional neural nets to detect facial keypoints tutorial. http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/, 2014. Accessed: 2014-03-30.

[47] Facial keypoint detection. https://www.kaggle.com/c/facial-keypoints-detection/data. Accessed: 2014-10-10.

[48] B. T. Polyak. *Introduction to Optimization*. Translation Series in Mathematics and Engineering. Optimization Software, Inc., 1987.