Understanding Neural Networks Mathematically through Image Recognition

Tan Jie Heng Alfred

Academic Exercise submitted to the
National Institute of Education,
Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Bachelor of Science (Education)

2023

Statement of Originality

I hereby certify that the work embodied in this AAM40A written report is the result of original research and has not been submitted for another degree to any other University or Institution. In addition, I declare that to the best of my knowledge, this written report is free of plagiarism, and contains no material previously published or written by another person, except where due reference has been made in the text.

05/02/2023

Date

Tan Jie Heng Alfred U1960075J

Supervisor Declaration Statement

I have reviewed the content and presentation style of this AAM40A written report, and I declare that it is free of plagiarism and of sucient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the text and/or the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

Date	Dr. Paul Maurice Edmund Shutler

Author Declaration Statement

This AAM40A written report does not contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

05/02/2023

Date

Tan Jie Heng Alfred U1960075J

Acknowledgement

I would like to express my sincerest gratitude to my supervisor Dr Paul Maurice Edmund Shutler for proposing this project after expressing my interest. This research paper would have not been possible without his constant guidance and patience, especially when the project took longer than usual.

I would also like to thank my family and partner for their support and understanding during the course of this research project.

Contents

Chapter 1: Background

Introduction

Artificial intelligence has enhanced many aspects of daily life. One example is the use of artificial intelligence for facial and image recognition. There are many algorithms that allow for this seemingly intelligent behaviour. In this project, we will specifically be looking at three algorithms and architecture: Principal Component Analysis; Fisher Linear Discriminant; and Neural Networks.

1.1 Principal Component Analysis and Fisher Linear Discriminant

Images as Vectors

To deal with images mathematically, we would typically convert images to vectors as shown in Figure 1 below.

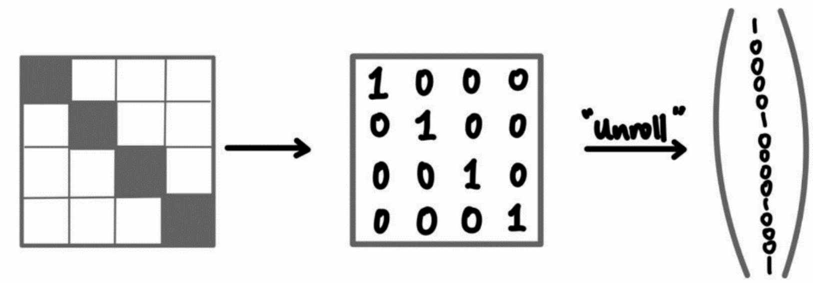


Figure 1: Example of representing a 16-pixel image as a vector in $\mathbb{R}^{16}$

By converting images to vectors, it allows us to look at distance of images diagonally, instead of just looking at each individual pixel value. For example, we could use the Euclidean distance, $\sqrt{\sum_i (u_i - v_i)^2}$, between two images $\boldsymbol{u}$ and $\boldsymbol{v}$ to compare them, where u_i and v_i are the i^{th} component of $\boldsymbol{u}$ and $\boldsymbol{v}$ respectively. Also, by allowing images to be vectors and defining a metric such as Euclidean distance on it, small differences in each pixel can add up to huge distances. Take for example two16-pixel images $\boldsymbol{u}$ and $\boldsymbol{v}$, such that $u_i = \begin{cases} 1, \text{if } i \equiv 1(\text{mod } 2) \\ 0.9, \text{if } i \equiv 0(\text{mod } 2) \end{cases}$ and $v_i = \begin{cases} 1, \text{if } i \equiv 0 \text{ (mod } 2) \\ 0.9 \text{ if } i \equiv 1 \text{ (mod } 2) \end{cases}$, for $1 \leq i \leq 16$. In this case, each pixel value only differ by at most 0.1, which gives the impression that these two images are somehow similar looking to another. However, if we take the Euclidean distance $\|\boldsymbol{u} - \boldsymbol{v}\| = \sqrt{\sum_{i=1}^{16} (u_i - v_i)^2} = 0.4$, they actually differ quite a bit more and may not be as similar as we thought.

Principal Component Analysis

Consider a set of n-pixel images of different faces. Since all faces look somewhat similar, they would form a cluster in the vector space $\mathbb{R}^n$. However, we know that each individual's face look different from one another, and to recognize them, we would try to find these differences. In this case, we could use Principal Component Analysis (PCA) to find these biggest differences (i.e., the most distinguishing features). The idea behind PCA is to project images onto vectors (or subspaces spanned by some set of vectors) to see how spread out they are, then identify the vectors that give the greatest spread. In particular, PCA gives a sequence of eigenvectors of decreasing eigenvalues. The eigenvectors can be thought of as the distinguishing features, while the eigenvalue of each corresponding eigenvector gives us an indication of how distinguishing the eigenvector is. Also, note that the eigenvectors may not consist of the original pixels, but more generally a linear combination of all the pixels.

The reason for PCA is to reduce the dimensionality of each image. In this project, we deal with hypothetical images of relatively low dimensionality, however, a more realistic image would have 256×256 pixels. Considered as a vector, the image would lie in the $\mathbb{R}^{256\times256}$ vector space. However, in facial recognition, only a few eigenvalues say, a hundred of them, are large and the others are small. Therefore, we could project the images onto the subspace spanned by these eigenvectors (of large eigenvalues) thereby reducing the dimensionality of the images and the computing time and cost. In this chapter, we briefly described what PCA is, and the more technical details will be dealt with in subsequent chapters.

Fisher Linear Discriminant

Now, we shall introduce another method of image recognition. Suppose we have images consisting of two classes: faces of Asians and faces of Caucasians. Within each class, there would be differences (the face of one Asian would look different from another Asian), but there would also be overall differences between the two classes (the faces of Asians are generally quite different from the faces of Caucasians). This means that both classes differ to somewhat similar degree and in similar ways, but there is an overall shift in some particular aspects.

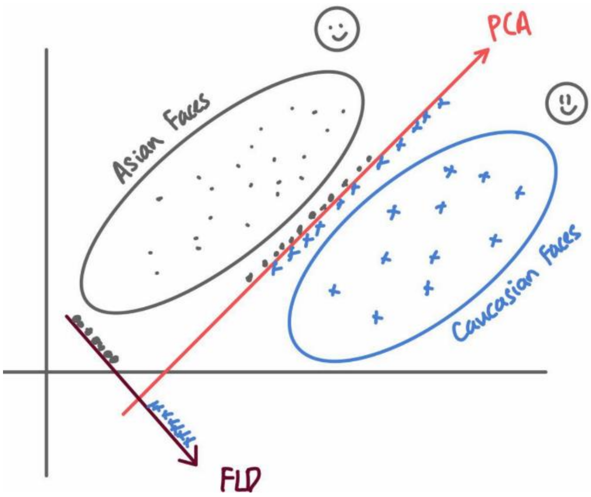


Figure 2: PCA and FLD on two classes of faces (Asian faces and Caucasian faces)

Figure 2 illustrates this in a simple 2-dimensional case, where the grey dots represent Asian faces, while the blue crosses represent Caucasian faces. If we were to use PCA on such a dataset, the red vector labelled “PCA” would be picked. This is because the variation within each cluster is large and in the same direction (i.e., Asian faces differ between each other more than the average Asian faces differ from the average Caucasian faces). This means that the differences between the two groups of faces are a subtle systematic thing (i.e., a translation of sort) acting against a backdrop of larger overall variations which are similar for both groups. Because of these, PCA would favour the larger overall variations within the groups and hence pick the vector labelled “PCA”. The problem is that if the images were to be projected on this vector, the grey dots will mix with the blue crosses, therefore we cannot distinguish and recognize the two faces.

The whole point of the Fisher Linear Discriminant (FLD) method is to overcome this problem, by minimizing the spread within the clusters and maximizing the spread between the different clusters. Therefore, FLD goes for the lesser PCA vectors but does the job better, by picking the maroon vector labelled “FLD”. When the images are then projected onto this vector, the grey dots and the blue crosses will separate out, allowing us to properly recognize the Asian faces from the Caucasian faces. In this chapter, we only briefly described what FLD does, and the more technical details will be discussed in subsequent chapters.

1.2 Neural Networks

What is a Neural Network?

An (artificial) neural network (NN) consists of layers of nodes (i.e., vertices) and some nodes connect with some other nodes by weights (i.e., edges). Each NN consists of an input and output layer of nodes, and some NN may have hidden layers between the input and output layers. In this project, we focus primarily on multi-layer perceptron, which is a kind of feedforward neural network, where all the nodes from the preceding layer is connected to all the nodes in the next layer (see Figure 3).

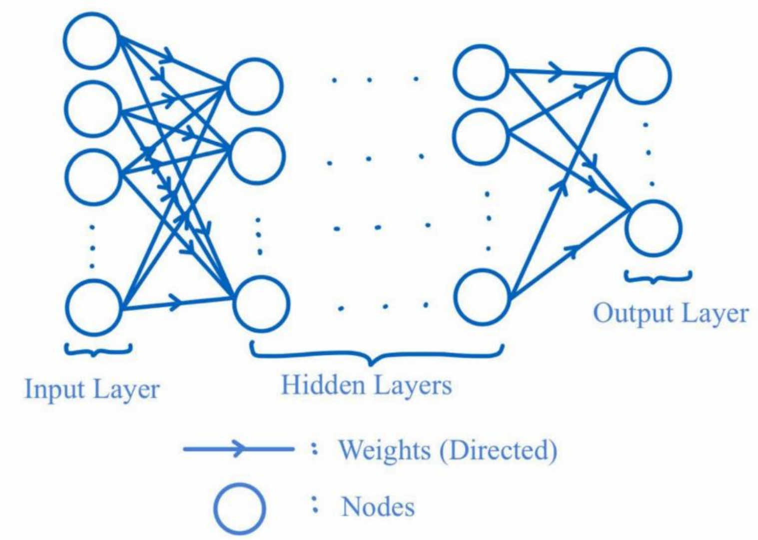


Figure 3: Illustration of neural network (multilayer perceptron)

Note that each weight is directed, from one node of a preceding layer to another node in the next layer. To find the value of each node, we take the weighted sum of the nodes in the preceding layer, add an additional bias term, then put this value into an activation function. The output of the activation function is the value of the node.

Training of Neural Network (Back Propagation)

For a NN to be good at recognizing images, we have to train it against a set of images. Firstly, we calculate the cost, which is the squared differences between the expected output and the value outputted by the NN. The cost acts as a benchmark for how well the NN is performing, and a high cost indicates that the NN is performing poorly. Then, to train the NN to be better at recognizing images, we could change the values of the weights and biases in each layer of the NN to obtain a lower cost for the NN.

Since we are lowering the cost with respect to the weights and biases, we could in principle find $-\nabla C$, where C is the cost, and move in that direction, as ∇C is the direction of the greatest change in C. This can be done via chain rule, since C is a function of the weights and biases in the network. However, in our project, we will obtain lower cost via numerical method, which will be discussed in more details in the subsequent chapters.

What Is the "Best" Neural Network?

The depth of a NN refers to the number of layers of nodes in the neural network, and the width of a NN refers to the maximum number of nodes in each of the layer. By increasing the depth and width, we could get the NN to perform better even in more complicated setting. In theory, though, a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function, given sufficient neurons in the hidden layer (Hornik, 1989). This is known as the Universal Approximation Theorem (UAT). However, in practice, the increase in depth is preferred over the increase in width, as deeper networks have more expressive power (Zhou et al., 2017). We shall illustrate this, as well, in the subsequent chapters. Again, this section serves only as an overview for NNs, the technical details will be discussed in subsequent chapters.

1.3 Literatures and Why This Project?

History of The Different Methods

The Principal Component Analysis was first introduced by Pearson (1901), while Fisher (1936) took it a step further and enhanced PCA to FLD. Then, Turk & Pentland (1991) applied PCA to image recognition and Belhumeur, et al. (1997) applied FLD to image recognition. On the other hand, the earliest version of the artificial neural network was introduced by McCulloch & Pitts (1943) used it for image recognition task, taking inspiration from the primary visual cortex. Since then, neural networks have been used for image recognition tasks, and Grant Sanderson, who produces really good mathematics videos on YouTube, made a series on neural networks applied to image recognition (Sanderson, 2017).

Why This Project?

In Sanderson's video (Sanderson, 2017), weights connecting the first layer and second layer of neurons (see Figure 4) seem to resemble the eigenfaces (see Figure 5) seen in Turk & Pentland (1991), in the sense that they look like the linear combination of their respective images (for Sanderson, the images are handwritten numbers, for Turk & Pentland, they are faces). Furthermore, Turk & Pentland (1991) asserted that there is a parallel between the eigenface approach and a neural network consisting of an input layer, a hidden layer, and an output layer (i.e., total of three layers). In particular, Turk & Pentland (1991) postulated that the weights connecting the input layer and the hidden layer correspond to the eigenfaces. With these observations, our project sets out to understand neural networks by answering the question: Are PCA and FLD the same as NN?

In this project, we use the open-source BASIC interpreter, called Yabasic, to write and run the various programmes. The codes will be appended in the appendix section. Appendix A will contain the programmes that appear in chapter 2, while Appendix B will contain those that appear in chapter 3.

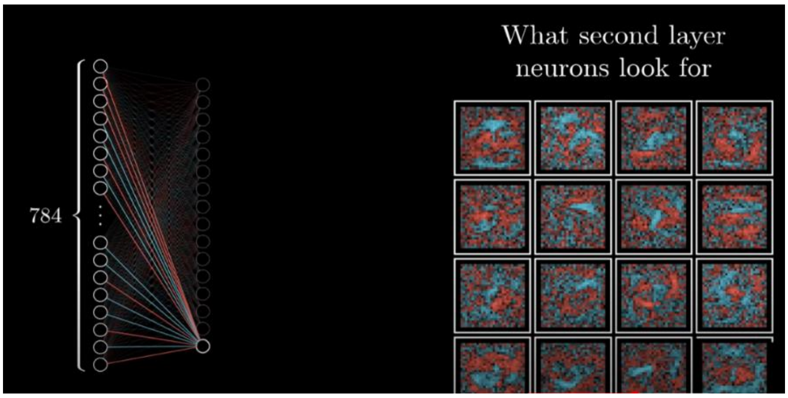


Figure 4: Visualized weights from input layer to first hidden layer in a neural network (Sanderson, 2017)

Figure 5: Illustration of eigenfaces (Turk & Pentland, 1991)

Chapter 2: Principal Component Analysis and 2-layer Neural Networks

2.1 Principal Component Analysis

2.1.1 Theory of PCA

Variance

To illustrate the Principal Component Analysis, let us suppose we have N numbers of 2-pixel images, $\boldsymbol{u}_i$, in our dataset (Figure 6), and we wish to classify them efficiently.

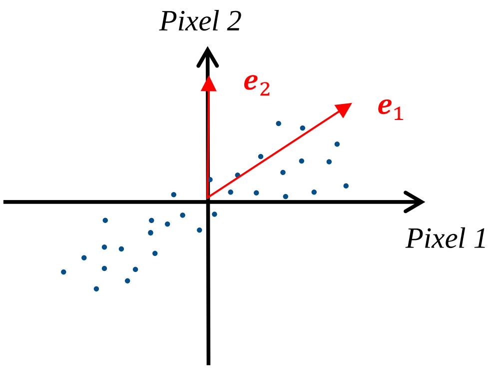


Figure 6: Dataset of Images

Here, we assume that the mean, $\boldsymbol{\mu}$, of the dataset is $\mathbf{0}$. Otherwise, we could always translate the dataset by taking subtracting every image by the mean of the dataset: $\boldsymbol{u}_i - \boldsymbol{\mu}$. We want to project the images onto a vector such that the spread of the dataset is being maximized. By maximizing the spread, the projected images would cluster less together, and hence we retain more information about the original images. Also, since we are only interested in the direction in which we should project the images, we could arbitrarily take the vector centred at the origin. For example, between the two vectors $\boldsymbol{e}_1$and $\boldsymbol{e}_2$, the projection onto $\boldsymbol{e}_1$ would retain more spread between the images, therefore we would pick $\boldsymbol{e}_1$ over $\boldsymbol{e}_2$.

To quantify this spread that we wish to maximize, we would use the variance, F, of the projected training images. Here, F can be written as

$$\text{Variance}, F = \sum_{i=1}^{N} (\boldsymbol{e} \cdot \boldsymbol{u}_i)^2 ,$$

where $\boldsymbol{e}$ is the vector for which the training images will be projected onto. Therefore, we want to find $\boldsymbol{e}$ that maximizes F.

Matrix

Since we want to maximize F with respect to $\boldsymbol{e}$, we can consider how F changes, δF, when $\boldsymbol{e}$ makes a small change, $\delta\boldsymbol{e}$. Therefore, we have

$$F + \delta F = \sum_{i=1}^{N} \big((\boldsymbol{e} + \delta\boldsymbol{e}) \cdot \boldsymbol{u}_i\big)^2$$

To find the maximum F, we set $\delta F = 0$. We write $\boldsymbol{e} = \binom{a}{b}$, $\boldsymbol{\delta e} = \binom{\delta a}{\delta b}$, and $\boldsymbol{u}_i = \binom{x_i}{y_i}$. Then, $F = \sum(\boldsymbol{e} \cdot \boldsymbol{u}_i)^2 = \sum(ax_i + by_i)^2$, and hence

$$\delta F = \sum_{i=1}^{N} \big((\boldsymbol{e} + \delta\boldsymbol{e}) \cdot \boldsymbol{u}_i\big)^2 - (\boldsymbol{e} \cdot \boldsymbol{u}_i)^2$$

$$= \sum_{i=1}^{N} \big((\boldsymbol{e} + \delta\boldsymbol{e}) \cdot \boldsymbol{u}_i + \boldsymbol{e} \cdot \boldsymbol{u}_i\big)\big((\boldsymbol{e} + \delta\boldsymbol{e}) \cdot \boldsymbol{u}_i - \boldsymbol{e} \cdot \boldsymbol{u}_i\big)$$

$$= \sum_{i=1}^{N} (2\boldsymbol{e} \cdot \boldsymbol{u}_i)(\delta\boldsymbol{e} \cdot \boldsymbol{u}_i) + (\delta\boldsymbol{e} \cdot \boldsymbol{u}_i)^2$$

$$= \sum_{i=1}^{N} 2(ax_i + by_i)(\delta a x_i + \delta b y_i) + (\delta a x_i + \delta b y_i)^2$$

$$= \sum_{i=1}^{N} 2(a\delta a x_i^2 + a\delta b x_i y_i + b\delta a x_i y_i + b\delta b y_i^2) + (\delta a^2 x_i^2 + 2\delta a\delta b x_i y_i + \delta b^2 y_i^2)$$

$$\approx \sum_{i=1}^{N} 2(a\delta a x_i^2 + a\delta b x_i y_i + b\delta a x_i y_i + b\delta b y_i^2)$$

$$= \sum_{i=1}^{N} 2(\delta a \quad \delta b)\begin{pmatrix} x_i^2 & x_i y_i \\ x_i y_i & y_i^2 \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix}$$

$$= 2(\delta\boldsymbol{e})^T \left[\sum_{i=1}^{N} \begin{pmatrix} x_i^2 & x_i y_i \\ x_i y_i & y_i^2 \end{pmatrix}\right] \boldsymbol{e}$$

$$= 2(\delta\boldsymbol{e})^T A\boldsymbol{e}, \qquad \text{where } A = \sum_{i=1}^{N} \begin{pmatrix} x_i^2 & x_i y_i \\ x_i y_i & y_i^2 \end{pmatrix}$$

This means that $\nabla F = \begin{pmatrix} \partial \mathrm{F}/\partial \mathrm{a} \\ \partial F/\partial b \end{pmatrix} = 2A\boldsymbol{e}$. Using the same procedure, we can re-write F as,

$$F = \sum_{i=1}^{N} (ax_i + by_i)^2 = \sum_{i=1}^{N} a^2 x_i^2 + 2abx_i y_i + b^2 y_i^2$$
$$= \boldsymbol{e}^T \left[\sum_{i=1}^{N} \begin{pmatrix} x_i^2 & x_i y_i \\ x_i y_i & y_i^2 \end{pmatrix}\right] \boldsymbol{e}^T$$
$$= \boldsymbol{e}^T A\boldsymbol{e}$$

Eigenvectors

Now, note that F gets arbitrarily large as $\|\boldsymbol{e}\|$ increases indefinitely. Since we are only interested in the direction of $\boldsymbol{e}$ we shall add the constraint that $\|\boldsymbol{e}\| = 1$.

$$\therefore \|\boldsymbol{e}\| = 1 \Rightarrow \|\boldsymbol{e}\|^2 = 1 \Rightarrow \boldsymbol{e} \cdot \boldsymbol{e} = 1$$
$$\Rightarrow \delta\boldsymbol{e} \cdot \boldsymbol{e} = 0 \Rightarrow \delta\boldsymbol{e} \perp \boldsymbol{e}, \text{for all } \boldsymbol{e} \in \mathbb{R}^2$$

This means that any small change in $\boldsymbol{e}$ must be in the direction perpendicular to $\boldsymbol{e}$, for all $e \in \mathbb{R}^2$.

Let $\boldsymbol{e}_1$ be the vector that maximizes F. Then, $\delta\boldsymbol{e}_1$ is any small change in $\boldsymbol{e}_1$ constrained to $\|\boldsymbol{e}_1\| = 1$. Therefore,

$$\delta F = 2(\delta\boldsymbol{e}_1)^T A\boldsymbol{e}_1 = (\delta\boldsymbol{e}_1)^T(2A\boldsymbol{e}_1) = (\delta\boldsymbol{e}_1)^T \nabla F = 0$$

$$\therefore \nabla F \cdot \delta\boldsymbol{e}_1 = 0 \Rightarrow \nabla F \perp \boldsymbol{\delta e}_1$$

Since $\delta\boldsymbol{e}_1 \perp \boldsymbol{e}_1$ and $\nabla F \perp \delta\boldsymbol{e}_1$, then $\nabla F \parallel \boldsymbol{e}_1$.

$$\therefore \nabla F = 2A\boldsymbol{e}_1 = \lambda\boldsymbol{e}_1, \text{for some } \lambda \in \mathbb{R}$$
$$\therefore A\boldsymbol{e}_1 = \lambda_1\boldsymbol{e}_1, \text{for some } \lambda_1 \in \mathbb{R}$$

Therefore, by definition, $\boldsymbol{e}_1$ is an eigenvector of A with the corresponding eigenvalue λ_1. Now, note that A may have more than one eigenvectors, but for any unit eigenvector $\boldsymbol{e}_k$ of A, with eigenvalue λ_k, we have

$$F = \boldsymbol{e}_k^T A \boldsymbol{e}_k = \boldsymbol{e}_k^T (\lambda_k \boldsymbol{e}_k) = \lambda_k \boldsymbol{e}_k^T \boldsymbol{e}_k$$
$$= \lambda_k$$

Therefore, by definition of $\boldsymbol{e}_1$, λ_1 is the largest eigenvalue of A, and $\boldsymbol{e}_1$is called the *first principal component*.

Orthogonal Complement, $e_1^\perp$

To find the subsequent principal components, we would focus on the components free of $\boldsymbol{e}_1$. These components will be more efficient in explaining the dataset, since we have already extracted out $\boldsymbol{e}_1$. That is, we want to find principal components orthogonal to $\boldsymbol{e}_1$. Therefore, we shall define $e_1^\perp$ as follows,

$$e_1^\perp = \{\boldsymbol{u} \in \mathbb{R}^2 : \boldsymbol{u} \cdot \boldsymbol{e}_1 = 0\} \subset \mathbb{R}^2$$

Note that $e_1^\perp$ is a proper subspace of the original vector space, $\mathbb{R}^2$, that the images are in.

Now, the second principal component onwards will have to be in $e_1^\perp$, since we want components free of $\boldsymbol{e}_1$. We shall first show that $e_1^\perp$ is invariant to A. That is, for any $\boldsymbol{u} \in e_1^\perp$, when the transformation $A\boldsymbol{u}$ is applied, it is still in $e_1^\perp$. By ensuring this, we could employ the same idea as above to find the remaining principal components.

Proof

For the proof, we first observe that $A = \sum_{i=1}^{N} \begin{pmatrix} x_i^2 & x_i y_i \\ x_i y_i & y_i^2 \end{pmatrix} = \sum_{i=1}^{N} \left(\boldsymbol{u}_i \boldsymbol{u}_i^T\right)$ is symmetric, that is $A = A^T$

We want to show that $A\boldsymbol{u} \in e_1^\perp \Leftrightarrow \boldsymbol{e}_1 \cdot A\boldsymbol{u} = 0$. Therefore,

$$\begin{aligned} \boldsymbol{e}_1 \cdot A\boldsymbol{u} &= \boldsymbol{e}_1^T (A\boldsymbol{u}) = (\boldsymbol{e}_1^T A)\boldsymbol{u} \\ &= (\boldsymbol{e}_1^T A^T)\boldsymbol{u} = (A\boldsymbol{e}_1)^T \boldsymbol{u} \\ &= (\lambda \boldsymbol{e}_1)^T \boldsymbol{u} = \lambda \boldsymbol{e}_1^T \boldsymbol{u} = \lambda(\boldsymbol{e}_1 \cdot \boldsymbol{u}) \\ &= 0 \end{aligned}$$

$\Leftrightarrow A\boldsymbol{u} \in \boldsymbol{e}_1^T$ and we are done.

Finding The Second Principal Component, $\boldsymbol{e}_2$, Restricted to $e_1^\perp$

Since $e_1^\perp$ is A-invariant, we shall define $A|_{e_1^\perp}$ as the matrix A restricted to $e_1^\perp$, and re-write $A_1 = A|_{e_1^\perp}$. That is, A_1 is a submatrix of A. Similarly, we define $F_1 = F|_{e_1^\perp}$ to be the value of F when restricted to $e_1^\perp$.

Therefore, $\boldsymbol{e}_2 = \underset{e \in e_1^\perp}{\operatorname{argmax}}(F_1) = \underset{e \in e_1^\perp}{\operatorname{argmax}}\, \boldsymbol{e}^T A_1 \boldsymbol{e}$. To this end, we find $\max_{\|e\|=1} F$, similar to how we find $\boldsymbol{e}_1$. Now, $\nabla F_1 = 2A_1\boldsymbol{e}$ and $\delta F_1 = 2\boldsymbol{\delta e}^T A_1 \boldsymbol{e}$. When $\boldsymbol{e} = \boldsymbol{e}_2$, we have

$$\delta F_1 = 0 = \boldsymbol{\delta e_2^T} \cdot \nabla F_1 \Rightarrow \nabla F_1 \parallel \boldsymbol{\delta e}_2^T$$

Again, $\boldsymbol{\delta e} \cdot \boldsymbol{e} = 0, \forall \boldsymbol{e} \in e_1^\perp$. In particular, $\boldsymbol{\delta e}_2 \parallel \boldsymbol{e}_2 \Rightarrow \nabla F_1 \parallel \boldsymbol{e}_2 \Rightarrow A_1\boldsymbol{e}_2 = \lambda_2 \boldsymbol{e}_2$, for some $\lambda_2 \in \mathbb{R}$.

$\boldsymbol{e}_2$ As An Eigenvector of A

We will now claim and prove that $\boldsymbol{e}_2$ is in fact an eigenvector of A. Furthermore, λ_2 is the second highest eigenvalue of A, after λ_1.

Proof

Let $V = \{k\boldsymbol{e}_1 : k \in \mathbb{R}\}$, then $V^\perp = e_1^\perp$. Now, by theorem, for any $\boldsymbol{x} \in \mathbb{R}^n, A\boldsymbol{x} = \boldsymbol{v} + \boldsymbol{v}'$, for some $\boldsymbol{v} \in V$ and $\boldsymbol{v}' \in V^\perp$. Let S and S' be the basis of V and $V^\perp$, respectively. Note that $S \cap S' = \emptyset$ and $S \cup S^\perp$ is a basis for $\mathbb{R}^n$. Now, note that since $e_1^\perp$ is A-invariant, then $A\boldsymbol{e}_2 \in e_1^\perp$. In particular, $A\boldsymbol{e}_2$ will be a non-zero linear combination of $\boldsymbol{s}_i' \in S'$. If we consider A and A_1 to be the matrices associated with the linear transformations $T: \mathbb{R}^n \to \mathbb{R}^n$ and $T_1: e_1^\perp \to \mathbb{R}^n$, then for $A\boldsymbol{e}_2$, we have $A\boldsymbol{e}_2 = A_1\boldsymbol{e}_2$, since $A\boldsymbol{e}_2$ has no non-zero components from S. Therefore, $A\boldsymbol{e}_2 = \lambda_2\boldsymbol{e}_2$, for some $\lambda_2 \in \mathbb{R}$. Now, to prove that λ_2 is the second largest eigenvalue of A, we suppose $\boldsymbol{e} \neq \boldsymbol{e}_1$ is a unit eigenvector of A with eigenvalue λ such that $\lambda_2 < \lambda < \lambda_1$. Since $\boldsymbol{e} \neq \boldsymbol{e}_1$ and $\boldsymbol{e}$ is an eigenvector, then $\boldsymbol{e} \perp \boldsymbol{e}_1$ by theorem. Therefore, $\boldsymbol{e} \in V$.

Now, by definition of $\boldsymbol{e}_2$, $F_1(\boldsymbol{e}) = \boldsymbol{e}^T A_1 \boldsymbol{e} = \lambda \leq \lambda_2 = \boldsymbol{e}_2^T A_1 \boldsymbol{e} = F_1(\boldsymbol{e}_2)$, which contradicts the assumption $\lambda_2 < \lambda$. Therefore, λ_2 is the second largest eigenvalue of A, and we are done.

This proof generalizes to more eigenvectors. For example, if we already have $\boldsymbol{e}_1, \boldsymbol{e}_2$, and $\boldsymbol{e}_3$, and we are trying to show that $\boldsymbol{e}_4$, which maximizes $F|_{V_3^\perp}$, where $V_3^\perp = \{\boldsymbol{v} \in \mathbb{R}^n : \boldsymbol{v} \cdot \boldsymbol{e_1} = \boldsymbol{v} \cdot \boldsymbol{e_2} = \boldsymbol{v} \cdot \boldsymbol{e_3} = 0\}$. Then, we can also write $A\boldsymbol{e}_4 = \boldsymbol{v} + \boldsymbol{v}'$, where $\boldsymbol{v} \in V_3$ and $\boldsymbol{v}' \in V_3^\perp$. Note that $V_3 = (V_3^\perp)^\perp = \text{span}\{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$, since $\boldsymbol{e}_1, \boldsymbol{e}_2$, and $\boldsymbol{e}_3$ are eigenvectors and therefore $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$ is a basis for V_3. Hence, we can write $A\boldsymbol{e}_4 = \alpha_1 \boldsymbol{e}_1 + \alpha_2 \boldsymbol{e}_2 + \alpha_3 \boldsymbol{e}_3 + \boldsymbol{v}'$, where α_i are some real values.

Therefore, we can apply the same argument recursively: If A is a $n \times n$ matrix (i.e., the images are of n dimension), then the principal components extracted would be the set of eigenvectors $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n\}$ of A. Furthermore, $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, where λ_i is the eigenvalue of the corresponding eigenvector $\boldsymbol{e}_i$. This process of extracting the eigenvectors from the matrix A (also known as the covariance matrix of the dataset) is called *Principal Component Analysis* (PCA).

2.1.2 Codes for PCA

Random Change to $\boldsymbol{e}$

```
label random_change

for i = 1 to dim_images
e(i) = e(i) + ran(2*ss) - ss
next i

return
```

Figure 7: Small changes to *e* subroutine

In this subroutine, $\boldsymbol{e} \in \mathbb{R}^n$ is being updated as $\boldsymbol{e} + \delta\boldsymbol{e}$, where $\delta\boldsymbol{e} = \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix}$, $\delta_i \in [-ss, ss]$, and ss is the maximum value or perturbation (i.e., step size). This is done to get every eigenvector $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$. However, we still need to ensure that e_k is orthogonal to every $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_{k-1}$, where $2 \leq k \leq n$, which will be done in the following subroutine.

Orthogonalization of $\boldsymbol{e}$

```
label orthogonalization

for k = 1 to num_eigenvectors

dot_product = 0
for i = 1 to dim_images
dot_product = dot_product + (e(i) * eigenvec(i,k))
next i

for j = 1 to dim_images
e(j) = e(j) - dot_product*eigenvec(j,k)
next j

next k

return
```

Figure 8: Orthogonalize *e* subroutine

In this subroutine, *num_eigenvectors* represents the number of eigenvectors currently obtained, and *eigenvec(i,k)*, represents the i^{th} component of the k^{th} eigenvector in the k-loop.

For example, suppose we want to orthogonalize $\boldsymbol{e}$ with respect to $\boldsymbol{e}_1$. This subroutine would first find $\boldsymbol{e} \cdot \boldsymbol{e}_1$, which is the length of projection of $\boldsymbol{e}$ onto $\boldsymbol{e}_1$, since $\|\boldsymbol{e}_1\| = 1$. Then, we find $\boldsymbol{e} - (\boldsymbol{e} \cdot \boldsymbol{e}_1)\boldsymbol{e}_1$, to get the projection of $\boldsymbol{e}$ onto $e_1^\perp$.

In general, if we want $\boldsymbol{e}$ to be in $\mathbb{R}^n|_{e_1^\perp, \ldots, e_k^\perp}$, then we do this recursively to get $\boldsymbol{e} - (\boldsymbol{e} \cdot \boldsymbol{e}_1)\boldsymbol{e}_1 - \cdots - (\boldsymbol{e} \cdot \boldsymbol{e}_k)\boldsymbol{e}_k$, the projection of $\boldsymbol{e}$ onto $\mathbb{R}^n|_{e_1^\perp, \ldots, e_k^\perp}$.

Normalization of $\boldsymbol{e}$

```
label normalize_e

E = 0
for i = 1 to dim_images
E = E + e(i)^2
next i
E = sqrt(E)

for i = 1 to dim_images
e(i) = e(i)/E
next i

return
```

Figure 9: Normalize $\boldsymbol{e}$ **subroutine**

In the first half of the subroutine (in box 1), E is calculating $||\boldsymbol{e}||$, while in the second half of the subroutine (in box 2), we are updating $\boldsymbol{e}$ to be $\frac{\boldsymbol{e}}{||\boldsymbol{e}||}$, the unit vector parallel to itself. This is to impose the restriction that $||\boldsymbol{e}|| = 1$, since we are only interested in the direction of $\boldsymbol{e}$.

Since we are trying to find $\boldsymbol{e}$ that maximizes F, therefore, after perturbing, orthogonalizing, and normalizing $\boldsymbol{e}$, we calculate F_{new} with this new $\boldsymbol{e}$. If F_{new} is larger than the original F value, we will update $\boldsymbol{e}$ to this new $\boldsymbol{e}$, otherwise, we go back to randomly changing $\boldsymbol{e}$ again.

2.1.3 Illustration of PCA with Ellipse Dataset

Description of Dataset and PCA

In this project, we will generate artificial dataset, with added noise, instead of processing real images. In this section, we will look at a cluster (i.e., one class) of 1000, 2-pixel images. Firstly, the images will be generated such that they satisfy the equation $\frac{x^2}{4} + y^2 \leq 1$, where x and y represent pixel 1 and 2 of each image respectively. Then, they will be rotated by 45° in the anti-clockwise direction, showing an ellipse that is rotated off the axes (Figure 10).

Since the image vectors will be in $\mathbb{R}^2$, we expect two eigenvectors, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, to be produced. And since the cluster is an ellipse, it is non-symmetrical, therefore the two eigenvalues will not be the same, i.e., $\lambda_1 > \lambda_2$.In this case, the covariance matrix A will be a 2×2 matrix, since each image vector is two-dimensional:

$$A = \sum_{\text{images}} \begin{pmatrix} (x - \mu_1)^2 & (x - u_1)(y - \mu_2) \\ (x - \mu_1)(y - \mu_2) & (y - \mu_2)^{\,2} \end{pmatrix},$$

where $\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}$, is the average of the image vectors in the dataset.

Screenshots

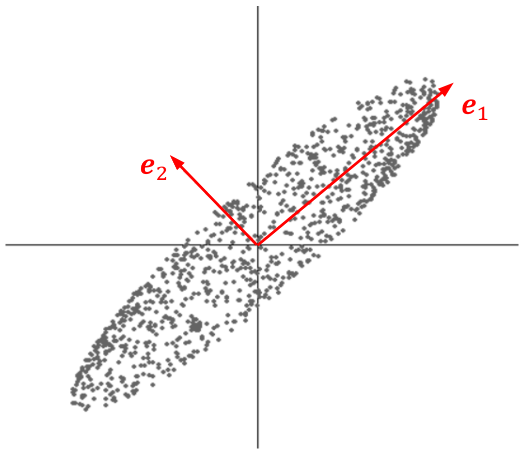


Figure 10: Ellipse dataset and eigenvectors

```
The eigenvector obtained is:
      0.740635029
      0.671907549
F is:
    1406.306061197

Current eigenvectors are:
0.740635 0
0.671908 0

Next eigenvector:
-0.671908
0.740635
The eigenvector obtained is:
     -0.671907549
      0.740635029
F is:
      57.584053259

Current eigenvectors are:
0.740635 -0.671908
0.671908 0.740635
```

Figure 11: Output of F**-values and eigenvectors on ellipse dataset**

Discussions and Comparisons

Let $B = (\boldsymbol{e}_1 \quad \boldsymbol{e}_2)$. Then, $B^T AB$ should be almost diagonalized, and since $F = \boldsymbol{e}_i^T A\boldsymbol{e}_i = \lambda_i$, then $B^T AB \approx \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$.

Indeed, by substituting $B = \begin{pmatrix} 0.741 & -0.672 \\ 0.672 & 0.741 \end{pmatrix}$, and $A = \begin{pmatrix} 797.423 & 671.175 \\ 671.175 & 666.467 \end{pmatrix}$, entries corrected to nearest 3 decimal places, we have

$$\begin{aligned} B^T AB &= \begin{pmatrix} 0.741 & 0.672 \\ -0.672 & 0.741 \end{pmatrix} \begin{pmatrix} 797.423 & 671.175 \\ 671.175 & 666.467 \end{pmatrix} \begin{pmatrix} 0.741 & -0.672 \\ 0.672 & 0.741 \end{pmatrix} \\ &= \begin{pmatrix} 1407.241 & 0.227 \\ 0.227 & 57.622 \end{pmatrix} \approx \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \end{aligned}$$

Spectral Theorem

From here, we can see that $B^T AB = \Omega$, where Ω is a diagonal matrix with eigenvalues of A as its diagonals. Since B is an orthonormal matrix, then $BB^T = I$. Therefore, the spectral theorem, which states that any real symmetric matrix is diagonalizable, is a consequence of this:

$$BB^T ABB^T = B\Omega B^T \Rightarrow A = B\Omega B^T,$$

where A is the real, symmetric matrix.

2.2 Neural Network (2-layer)

2.2.1 Theory for 2-layer Neural Network

Forward Propagation

In this section, we will focus on 2-layer neural network (NN). The figure below (Figure 12) shows a 2-layer NN (2NN) with three input nodes (b_1, b_2, b_3) and two output nodes (a_1, a_2).

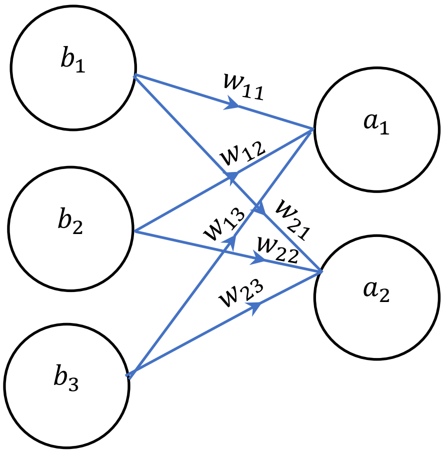


Figure 12: Simple example of a 2NN

Note that $b_k, a_i \in \mathbb{R}$, for $i = 1, 2$, and $k = 1, 2, 3$. Also, each output node, a_i, has three "arrows" pointing in with values $w_{ik} \in \mathbb{R}$ (i.e., the number of nodes in the preceding layer). We shall denote the vector whose components are the weights connecting to the node a_i as $\boldsymbol{w}_i = \begin{pmatrix} w_{11} \\ \vdots \\ w_{ik} \end{pmatrix}$, where k is the number of nodes in the input layer, $\boldsymbol{b}$.

The output layer, $\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$, will be calculated as a linear combination of b_k – with *weights* w_{ik} multiplied to b_k. That is,

$$a_i = \sum_{k=1}^{3} w_{ik} b_k$$
$$\therefore \boldsymbol{a} = W\boldsymbol{b}$$

where $W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$, and $\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$.

From above, $\boldsymbol{a}$ is just a linear function of $\boldsymbol{b}$, the layer preceding it. To relate to our intuition of recognition, where there is a "right" (1) or "wrong" (0), we put $\boldsymbol{a}$ through an *activation function*, $\sigma: \mathbb{R}^2 \to \mathbb{R}^2$, that reduces the range of output (e.g., the range become $(0,1)$ sigmoid and $[0, \infty)$ for ReLU (see below)). Therefore, instead of $\boldsymbol{a}$ as the output, we have $\sigma(\boldsymbol{a})$ as the output. In other words, each node a_i is given as

$$a_i = \sigma\left(\sum_{k=1}^{3} w_{ik} b_k\right)$$

Furthermore, σ is a continuous, non-decreasing function and is usually differentiable. Some examples of activation function include the sigmoid function, the hyperbolic tangent function, and the ReLU function, $\mathrm{ReLU}(x) = \max(x, 0)$. As for this project, we primarily use the sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}}\ , x \in \mathbb{R}$$

Notice that $\sigma(0) = 0.5$. If we were to view this output as a "threshold" value (i.e., "right" or "wrong"), we would want to be able to translate the graph horizontally. Therefore, we add a *bias* term, β_i, to evaluate a_i, and the final output of a_i is

$$a_i = \sigma\left(\sum_{k=1}^{3} w_{ik}\, b_k + \beta_i\right), \text{for } i = 1, 2$$

Here, each bias term is "attached" to each of the output node, and the effect it has is a horizontal translation of the sigmoid graph to change the "threshold" (Figure 13).

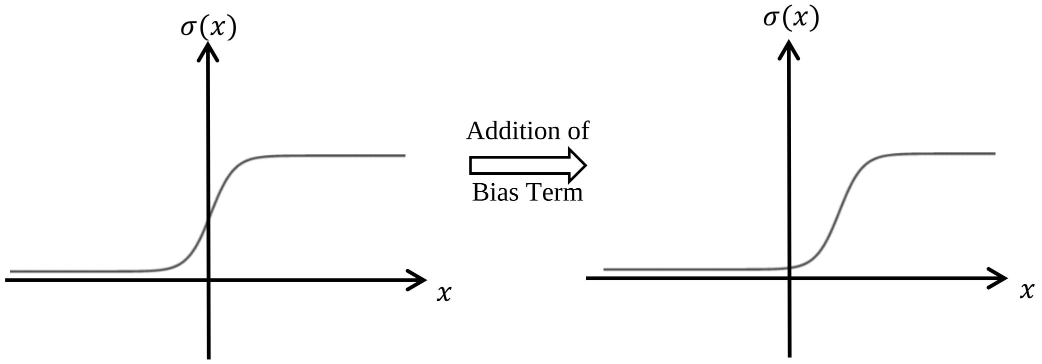


Figure 13: Illustration of effect of bias term

Cost of Network

The output of the network can be interpreted as the likelihood in which the network thinks the image belongs in the i^{th} class (i.e., category). For example, if $\boldsymbol{a} = \begin{pmatrix}1\\0\end{pmatrix}$, then the network believes that the image should be categorized as class 1, instead of class 2. On the other hand, the label, $\boldsymbol{p}$, of the image tells us the actual class (out of r numbers of classes) which the image actually belongs to, and this is provided to the machine. For example, if $r = 3$, and an image belongs to class 3, then its label will be $\boldsymbol{p} = \begin{pmatrix}0\\0\\1\end{pmatrix}$. In general, if $\boldsymbol{p} = \begin{pmatrix}p_1\\ \vdots \\ p_r\end{pmatrix} \in \mathbb{R}^r$, and an image belongs to class j, then $p_i = \begin{cases}0, & \text{if } i \neq j\\ 1, & \text{if } i = j\end{cases}$.

Now, we are able to improve the network by using the *cost* as a benchmark. The cost of an image is how much the network's output differs from the image's label, and is given by

$$\text{Cost} = \sum_{i=1}^{n} (a_i - p_i)^2 = \|\boldsymbol{a} - \boldsymbol{p}\|^2, \text{where } n \text{ is the number of pixels}$$

However, the network could perform poorly on one image and better at another image. Therefore, we would like to obtain an average cost over, say, N images. Therefore,

$$\text{Average Cost} = \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{a}_i - \boldsymbol{p}_i\|^2\ ,$$

where $\boldsymbol{a}_i$ is the network's output for the i^{th} image, and $\boldsymbol{p}_i$ is the label of the i^{th} image.

Extending this idea, we evaluate the average cost over the size of the population, P by taking $N = P$. However, this is an idealized situation and in practice, a very difficult or impossible task to do.

Instead of taking the entire population of images, we could sample a sufficiently large t number of images from the population. These t images will then form our training set, which we would then use to train the NN. Our objective now is to reduce the average cost of the network over the entire training set, C, by changing the weights and biases. In this project, we made use of numerical methods to achieve this.

Since C is locally a quadratic function of the parameters (i.e., every weight and bias in the network), we can make use of the idea that quadratic function has a turning point. Suppose we want to change the weight w_{11}, to minimize C, and its initial value is $w_{11} = w$.

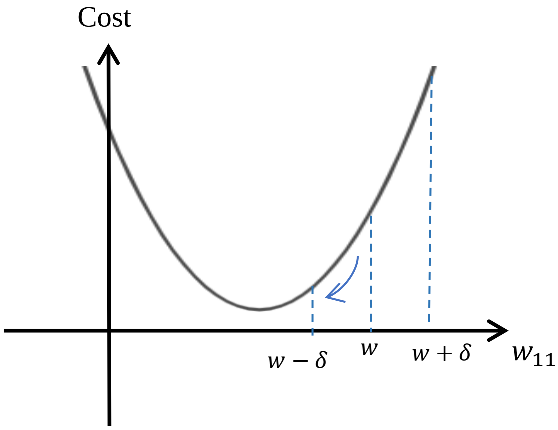


Figure 14: Illustration of updating

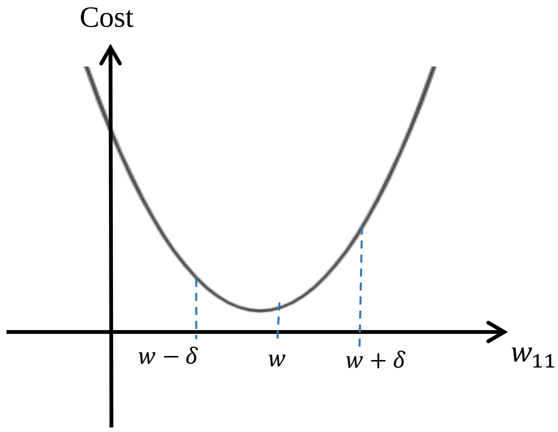


Figure 15: Illustration of no update for w_{11}

Now, we evaluate C at $w_{11} = w - \delta$ and $w_{11} = w + \delta$, for small value δ, which we call the *step size*. Then, we compare the values of C at these three values of w_{11} and "move" in the direction of the lowest C value. In the case shown in Figure 14, $w_{11} = w - \delta$ yields a lower C, so we update w_{11} as $w - \delta$.

However, if $w_{11} = w$ yields the smallest C of the three values, this would suggest that we are "near" the minimum value (Figure 15). In this case, we do not update the value of w_{11}. Now, if this happens to the other weights and biases as well, we will then update the step size to become $\frac{\delta}{2}$ instead of δ.

2.2.2 Codes for 2NN

```
label compute_NN

for i = 1 to anum                          1
a(i) = 0
for j = 1 to bnum
a(i) = a(i) + w(i,j)*b(j)
next j
next i

for i = 1 to anum                          2
a(i) = a(i) + abias(i)
next i

for i = 1 to anum                          3
a(i) = sigmoid(a(i))
next i

return
```

Figure 16: Computing 2NN subroutine

Computing Neural Network

In this subroutine, *anum* and *bnum* represents the number of nodes in the output layer and input layer, respectively. The codes in box 1 in this subroutine calculates each of the output node a_i as the weighted sum of b_j, all the input nodes from the preceding layer, where the weight is w_{ij} for each b_j. Then, the codes in box 2 adds a bias term, $abias_i$, to each a_i.

Lastly, the codes in box 3 in the subroutine inputs the value of a_i, calculated above, into the sigmoid function, $\sigma: \mathbb{R} \to (0,1)$, where $\sigma(a_i) = \frac{1}{1+e^{a_i}}$, for each $i = 1, 2, \dots, anum$.

Calculating Average Cost

In this subroutine, *numimages* is the number of images in the training set, and the subroutine *input_c* retrieves the training images from a text file and store them as arrays of $c(i)$.

We want to highlight two subroutines in this screenshot: *calculate_cost* in box 1, and *average_cost* in box 2.

The *calculate_cost* subroutine in box 1 calculates the cost for a single image. The subroutine *select_p* within this subroutine labels the image with the label $\boldsymbol{p}$, so that the cost could be calculated as

$$\text{Cost} = \sum_{i=1}^{anum} (a_i - p_i)^2$$

The *average_cost* subroutine in box 2 iteratively calls upon the *calculate_cost* subroutine in box 1, for each image, over the entire training dataset (i.e., all the images in the dataset). Then, it takes the average of the cost over the entire training dataset, to find the average cost of the network.

```
label calculate_cost                1

gosub select_p
cost = 0
for i = 1 to anum
cost = cost + (a(i) - p(i))^2
next i

return
```

****

```
label average_cost                  2
open #1, "c_input.txt", "r"
totalcost = 0
for image = 1 to numimages
gosub input_c
gosub compute_NN
gosub calculate_cost
totalcost = totalcost + cost
next image
avg = totalcost/numimages
close #1
return
```

Figure 17: Calculate Cost and Average Cost Subroutines

```
label change_w

for k = 1 to anum
for r = 1 to bnum

gosub average_cost
cost1 = avg

w(k,r) = w(k,r) + ss
gosub average_cost
cost2 = avg

w(k,r) = w(k,r) - 2*ss
gosub average_cost
cost3 = avg

w(k,r) = w(k,r) + ss                1
cost12 = cost1-cost2 : cost13 = cost1-cost3

if(cost12>0) then
w(k,r) = w(k,r) + ss
improvflag = 1
elseif(cost13>0) then
w(k,r) = w(k,r) - ss
improvflag = 1
endif                               2

next r
next k
return
```

Figure 18: Changing weights *w* subroutine

Changing Weights

In this subroutine, we are changing the weight w_{kr} such that the average cost of the network decreases. The codes in box 1 calculates the average cost of the network for the following values of w_{kr}: $w, w + ss$ and $w - ss$, where w is the initial value of w_{kr} and ss is the step size in which change the value of w_{kr}. *cost1*, *cost2*, *cost3* are the average costs at the three values of w_{kr} respectively. *cost12* and *cost13* are the differences between *cost1* and *cost2*, and *cost1* and *cost3*, respectively.

As described above, we update w_{kr} accordingly if the change in w_{kr} reduces the cost (i.e., either *cost12* or *cost13* are positive). If an update is done to w_{kr}, we note it down by updating *improvflag* to 1. A similar subroutine is used to change the biases of $\boldsymbol{a}$.

Changing step_size

This routine changes *ss* if no changes have been made to any of the parameters (i.e., all the weights and biases).

The codes in box 1 are all the subroutines changing the various weights and biases in the network, with an initial step size of 1, so as to reduce the average cost of the network. Again, these subroutines are similar to the one shown above for changing weight w_{kr}. Any changes to any of the weights and biases would update *improvflag* to 1, from 0.

If, after an iteration of trying to change the weights and biases to reduce the average cost, *improvflag* still remains at 0, then none of the parameter has been changed. This means that, with respect to all the parameters, they are all close to the minimum average cost of the network. Therefore, we go back to the point *300* and take half of the previous *ss*, before we continue to iterate through the changes to the parameters.

```
                                               1
        ss = 1
        iterations = 0
300
        ss = ss/2

100

        improvflag = 0
        gosub change_abias
        gosub change_w
        gosub output_current
        gosub output_weights
        gosub average_cost
        print "ss is: ", ss

        iterations = iterations + 1

        if improvflag = 0 then
        print "Reducing ss"
        goto 300
        else
        goto 100
        endif                                  2
```

Figure 19: Changing step_size, *ss*, subroutine

<u>2.2.3 Illustration of 2NN with 2-blob Dataset</u>

Description of Dataset and Initial Parameter Values

The images in this training set are of two pixels, and the training set consists of two different classes of images – class 1 and class 2. This should elicit more interesting behaviours for the 2NN as opposed to images of 1-pixel. Also, we will produce 100 images per class.

Since the images are of two pixels, then $bnum = 2$, where each input node represents one pixel. Also, since we have two classes of images, then $anum = 2$, where each output node will denote the likelihood that the image belongs in a class. Furthermore, the initial weights in which we would use would be $W = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$, and the initial biases are all set to be 0. Finally, the images in class 1 would be generated by adding noise of ± 0.1 to $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, while images in class 2 would be generated by adding noise of ± 0.1 to $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, as shown in Figure 20.

Screenshots

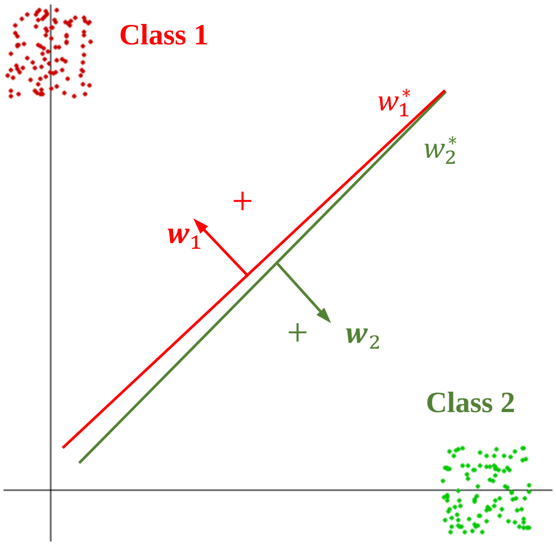
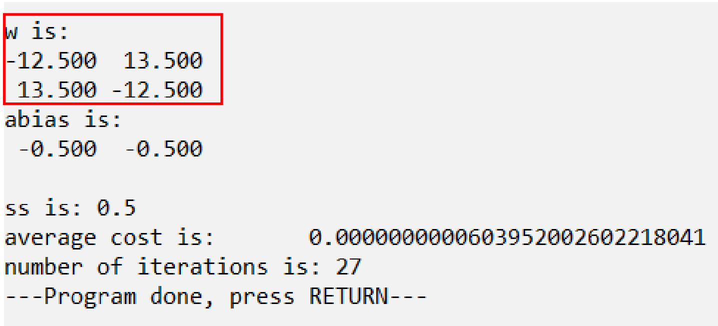


Figure 20: 2-blob dataset with visualized weights



```
w is:
-12.500  13.500
 13.500 -12.500
abias is:
 -0.500  -0.500

ss is: 0.5
average cost is:       0.0000000000603952002602218041
number of iterations is: 27
---Program done, press RETURN---
```

Figure 21: Screenshot of weights, biases, and average cost for 2NN on 2-blob dataset

Discussions

In Figure 14, the lines w_1^* and w_2^* are the "visual" weights (similar to a decision boundaries) in which the network decides to classify the images. In fact, a small change in the components of the weights would visually correspond to the tilting of the "visual" weights in $\mathbb{R}^2$. Note that the weights shown in the box in Figure 21 are such that the first column refers to $\boldsymbol{w}_1 = \begin{pmatrix} w_{11} \\ w_{12} \end{pmatrix}$ and the second column refers to $\boldsymbol{w}_2 = \begin{pmatrix} w_{21} \\ w_{22} \end{pmatrix}$. Therefore, $W = \begin{pmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \end{pmatrix}$. Also, from Figure 21, we can see that the weights $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ change from $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, to $\begin{pmatrix} -12.5 \\ 13.5 \end{pmatrix}$ and $\begin{pmatrix} 13.5 \\ -12.5 \end{pmatrix}$, respectively. In particular, the absolute value of the weights increased, overall. This increase in absolute value results in a scaling of the sigmoid function, making it "steeper" and more like a step-function (see Figure 22).

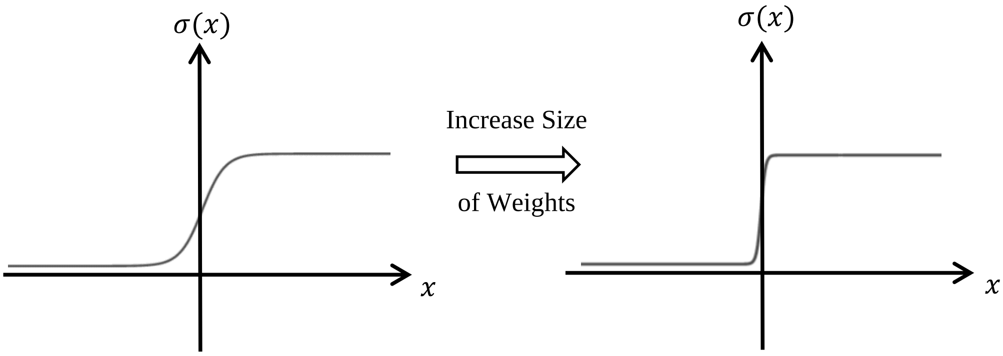


Figure 22: Illustration of effect of size of weights

Also, from Figure 21, we can see that the signs of two weights $\boldsymbol{w}_1 = \begin{pmatrix} -12.5 \\ 13.5 \end{pmatrix}$ and $\boldsymbol{w}_2 = \begin{pmatrix} 13.5 \\ -12.5 \end{pmatrix}$ are opposite. The signs of the weights determine the region in which the network would classify an image as a certain class. For example, referring to Figure 20, any image lying above w_1^* (i.e., in the direction of the red arrow) would be classified as Class 1. Therefore, the opposite signs of the weights w_2 means that the green arrow must be pointing in the opposite direction, hence classifying images in that region as Class 2. In total, weights can be seen as signed "cuts" in $\mathbb{R}^2$ due to its step-function-like property, cutting the space into $+$ and $-$. This can be seen from the sigmoid function as well (Figure 23). It should be noted that the weights $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ correspond to the red arrow and green arrow in Figure 21.

Lastly, just like how it shifts the sigmoid function, a change in the bias term would result in a translation of the "visual" weights in $\mathbb{R}^2$.

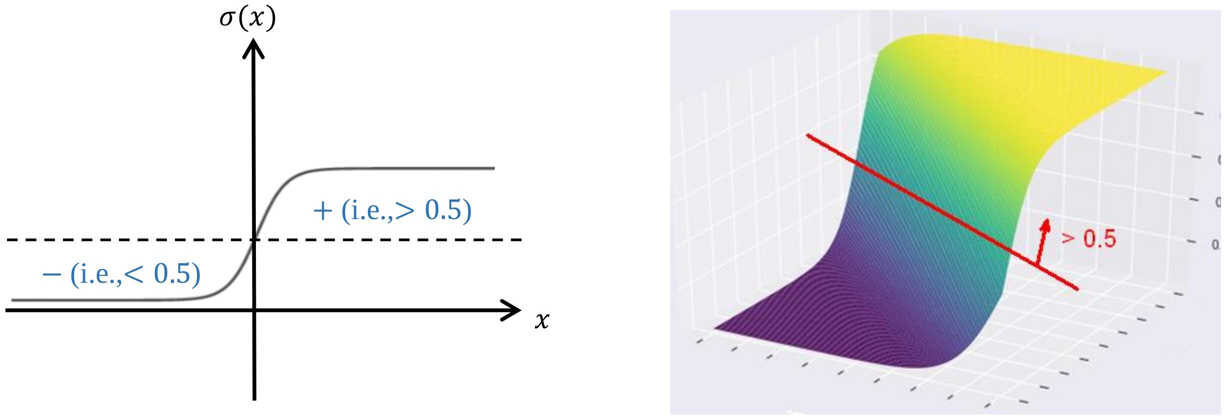


Figure 23: Illustration of signed and step-function-like property of weights (right image adapted from Singh (2019))

2.3 Comparing PCA and 2NN

2.3.1 PCA on 2-blob Dataset

In this section, we shall compare and discuss the results of PCA and 2NN on some datasets.

PCA on Example in 2.2.3

Firstly, we shall look at the results that PCA has on the dataset in Section 2.2.3 (Figure 20). Since the images are of two pixels, we expect two eigenvectors. Similarly, the covariance matrix $A = \sum_{\text{images}} \begin{pmatrix} (x-\mu_1)^2 & (x-\mu_1)(y-\mu_2) \\ (x-\mu_1)(y-\mu_2) & (y-\mu_2)^2 \end{pmatrix}$, where x is the first pixel value, y is the second pixel value, and $\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}$ is the average of the image vectors in the dataset. Figure 24 shows eigenvectors that we obtain from PCA programme and Figure 25 illustrates this on the diagram with the "visual" weights.

From Figure 25, we can see that the actions of the "visual" weights (i.e., the direction of the "+" arrow or $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$) are similar to the eigenvector of the highest eigenvalue, $\boldsymbol{e}_1$. In particular they are almost parallel. This suggests that the best way of separating the two classes of images is to maximize the variance when projected onto a direction vector, which is what PCA does, and the 2NN agrees with this.

```
F is:
     99.422988724

Current eigenvectors are:
0.449423 0.893319
-0.893319 0.449423
```

Figure 24: Eigenvectors for PCA on 2-blob dataset in section 2.2.3

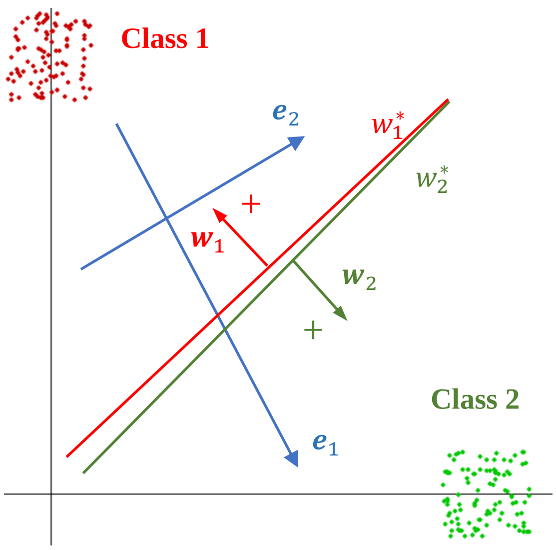


Figure 25: Visualization of eigenvectors and "visual" weights on 2-blob dataset in Figure 20

PCA and 2NN on Isosceles Triangle Datatset in $\mathbb{R}^3$

In this example, we have three classes of images in our dataset and each image is of three pixels (see Figure 26) Images in class 1, 2, and 3 are generated by adding noise of ± 0.2 to $\begin{pmatrix}1\\0\\0\end{pmatrix}$, $\begin{pmatrix}0\\1\\0\end{pmatrix}$, and $\begin{pmatrix}0\\0\\3\end{pmatrix}$, respectively. Each class of image has 100 images. Figure 27 and Figure 28 show the eigenvectors for PCA, and weights and biases for 2NN, on the dataset, respectively. The dataset resembles an isosceles triangle, with each cluster of images from the same class as the vertices (see Figure 26).

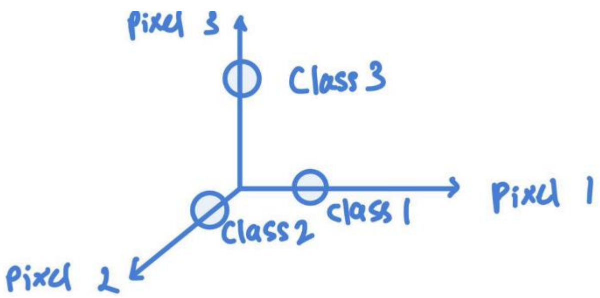


Figure 26 : Visualization of isosceles triangle dataset in $\mathbb{R}^3$

```
F is:
     3.473483105

Current eigenvectors are:
-0.166943 0.687326 0.706903
-0.156861 -0.726348 0.669188
0.973409 0.000830723 0.229074
```

Figure 27: Eigenvectors for PCA on isosceles triangle dataset

We expect three eigenvectors, two of which are on the plane, p, containing $\begin{pmatrix}1\\0\\0\end{pmatrix}$, $\begin{pmatrix}0\\1\\0\end{pmatrix}$, and $\begin{pmatrix}0\\0\\3\end{pmatrix}$, while the third (and last) would be orthogonal to this plane.

```
w is:
 16.500 -15.500 -15.500
-15.500  16.500 -14.500
 -5.500  -6.000  15.500
abias is:
 -0.500  -0.500 -15.500

ss is: 0.5
average cost is:  0.000000000723275546188489926
number of iterations is: 31
---Program done, press RETURN---
```

Figure 28: Weights and biases for 2NN on isosceles triangle dataset

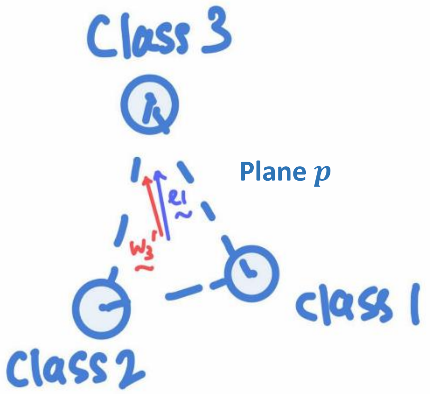


Figure 29: Visualization of plane p with projected weight w_3' and e_1 for isosceles triangle dataset

If we project $\boldsymbol{w}_3$, whose values are shown in the red box of Figure 28, onto the plane p, whose normal vector is $\begin{pmatrix}3\\3\\1\end{pmatrix}$, and normalize it, we get the resulting vector $\boldsymbol{w}_3' = \begin{pmatrix}-0.187\\-0.137\\0.973\end{pmatrix}$. This is very similar to the first eigenvector of $\begin{pmatrix}-0.167\\-0.157\\0.973\end{pmatrix}$ (see Figure 29). In fact, following the same procedure of projecting $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ onto plane p and normalizing them, we see that $\boldsymbol{w}_1' = \begin{pmatrix}0.725\\-0.649\\-0.230\end{pmatrix}$ and $\boldsymbol{w}_2' = \begin{pmatrix}-0.642\\0.725\\-0.250\end{pmatrix}$ are approximately $-\frac{1}{5}\boldsymbol{e}_1 + \boldsymbol{e}_2 = \begin{pmatrix}0.721\\-0.694\\-0.194\end{pmatrix}$ and $-\frac{1}{4}\boldsymbol{e}_1 - \boldsymbol{e}_2 = \begin{pmatrix}-0.645\\0.765\\-0.244\end{pmatrix}$, respectively. This means that $\boldsymbol{w}_3$ is somewhat parallel to $\boldsymbol{e}_1$ while $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are both linear combinations of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. The impact of $\boldsymbol{e}_3$ is insignificant since the eigenvalue is very small.

2.3.2 PCA and 2NN on OXUT Dataset in $\mathbb{R}^9$

Description of Dataset, PCA, and Initial Parameter Values of 2NN

From the two examples above, there seems to be some similarity between what PCA and 2NN do. In this section, we will stretch it and see if PCA and 2NN agree in a higher dimensional space. The images we will be using here are the alphabets “O”, “X”, “U”, and “T”, being represented as 3×3 images (see Figure 30). This means that the images now exist in $\mathbb{R}^9$. Since the images are in $\mathbb{R}^9$, this implies that there are potentially nine eigenvectors for PCA. However, some of the eigenvectors may have eigenvalues that are negligibly small. For PCA, the covariance matrix A will be given by

$$(A)_{ij} = \sum_{\text{images}} (x_i - \mu_i)(x_j - \mu_j),$$

where x_i and x_j are the i^{th} and j^{th} components of an image, and $\boldsymbol{\mu} = \begin{pmatrix}\mu_1\\\vdots\\\mu_9\end{pmatrix}$ is the average of the image vectors. For 2NN, we will have $bnum = 9$ and $anum = 4$, since there are 9 pixels and 4 classes in total. For the initial weights, w_{ij}, and biases, α_k, of the 2NN, we set them as $w_{ij} = \begin{cases}1, \text{if } i = j\\0, \text{if } i \neq j\end{cases}$ and $\alpha_k = 0$, for $k = 1, 2, \dots, anum$.

Screenshots

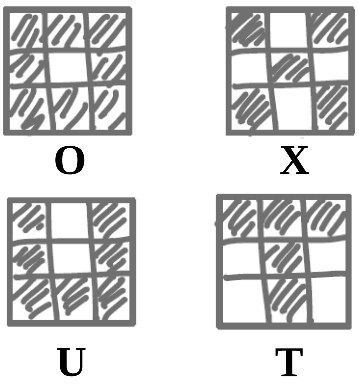


Figure 30: Illustration of O, X, U, T as images

```
w is:
  0.500   0.000   0.000   0.500
  7.500 -16.500 -25.000  12.500
 -4.500   1.000   4.500  -3.500
  2.000 -17.500   8.000 -10.500
 -5.500   7.000 -17.500   4.500
  0.500 -10.000   3.000 -12.000
  0.000  11.000  -1.000 -15.500
 -0.500 -18.000  -0.500  14.000
  0.000   9.500  -1.500 -13.500
abias is:
 -1.500  -2.000   0.500  -0.500

ss is: 0.5
average cost is:  0.004929366656849324179745508 4
number of iterations is: 50
```

Figure 31: Weights and biases for 2NN on OXUT dataset in $\mathbb{R}^9$

```
The eigenvectors and F-values are:

   0.0030    0.0028    0.0314   -0.1790   -0.0656    0.1346    0.6745    0.6537    0.2494
  -0.0889    0.6184    0.7774   -0.0214    0.0080   -0.0528   -0.0420    0.0044    0.0136
  -0.0149    0.0081   -0.0490    0.2816    0.0104   -0.8297    0.2528   -0.1533    0.3769
   0.5108    0.1413   -0.0245   -0.2736    0.1848    0.2005    0.4432   -0.6000    0.1134
  -0.5033   -0.1270    0.0796    0.3867    0.2281    0.4420    0.0958   -0.2541    0.5035
   0.4958    0.1390   -0.0791    0.0591   -0.1705    0.1316   -0.4635    0.2031    0.6501
   0.3159   -0.3870    0.3843    0.4927   -0.5277    0.1230    0.1791   -0.0831   -0.1659
   0.1826    0.5129   -0.3521    0.6359    0.2072    0.1278    0.1311    0.1520   -0.2751
   0.3141   -0.3863    0.3277    0.1152    0.7476   -0.0792   -0.0984    0.2308   -0.0577

 194.5245  105.6320   20.9025    3.3980    3.0453    2.6472    2.4686    2.1488    2.0285
```

Figure 32: Eigenvectors and *F*-values for PCA on OXUT dataset in $\mathbb{R}^9$

Discussions and Comparisons

From Figure 31 and Figure 32, we could see that the number of weights, $\boldsymbol{w}_i$, in the 2NN is less than the number of eigenvectors produced by the PCA. However, not all eigenvectors, $\boldsymbol{e}_i$, are "useful"; only those with non-zero eigenvalue, $\lambda_i \neq 0$. In fact, from the spectral theorem, we can deduce that the rank of A is the number of λ_i such that $\lambda_i \neq 0$. Viewed it this way, PCA tells us the components that spans the image space and remove any linear dependencies.

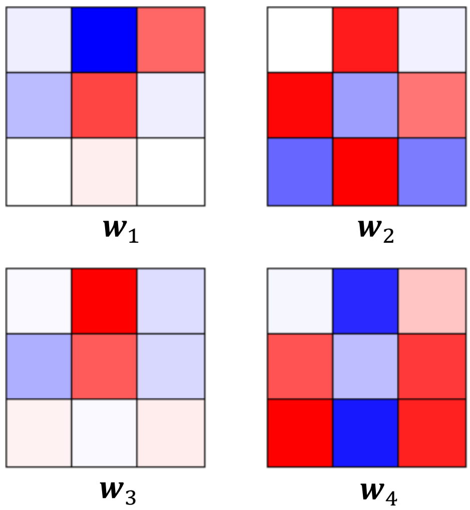


Figure 33: Illustration of weights of 2NN on OXUT dataset as images

As seen in Sanderson's video (Figure 4), we could visualize the weight vectors as images. We did just that for this example, since the other examples are of two or three pixels, which would be rather trivial. Figure 33 shows the weights as images, where each square represents one pixel (i.e., one component of the weights vector). The process of getting these is the reverse of unrolling images into vectors (Figure 1). Pixels coloured in blue have positive values, pixels coloured in red have negative values, and pixels in white have value 0. The darker the intensity, the higher the absolute value of the pixel. For example, a dark red pixel signifies a very negative pixel value (relative to other pixels in the same weights vector). These vectors resemble the eigenfaces, in the sense that we pick up "contrast" between different pixels, and we can almost trace out the distinguishing features between each alphabet. In fact, looking at $\boldsymbol{w}_2$, the red pixels are the pixels in which "X" should not have values in. This distinguishes it quite well from the rest of the images. However, it is not exactly obvious from these images of the weights whether one of the weights is parallel to the first eigenvector $\boldsymbol{e}_1$. In fact, if we look just at the signs of the components of $\boldsymbol{e}_1$ and the weights, they are not exactly parallel. Therefore, we may need another method to discover the relationship between the weights and the eigenvectors.

Because the weight vectors and eigenvectors all exist in $\mathbb{R}^9$, we could not easily visualize like other examples. However, only the first two eigenvectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ from the PCA are important, as their eigenvalues are the largest. Therefore, we could attempt to compare what the weight vectors and eigenvectors are doing by projecting the images and the weight vectors onto the subspace spanned by $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. For simplicity, we shall only project the "perfect" images of (i.e., no noise) "O", "X", "U", and "T"; the noisy images would just be clustering around these "perfect" images. Figure 33 shows the relative positions of the images and the visualized weights. To get the visualized weights, we have to first project the weight vectors onto the subspace and the resulting vector in that subspace would be orthogonal to the visualized weights.

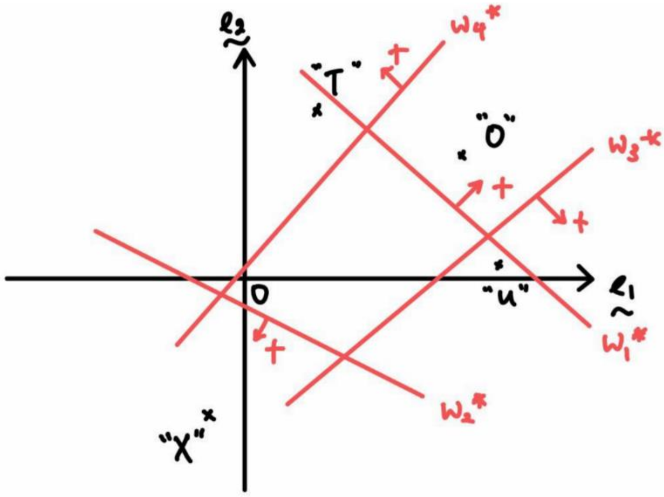


Figure 34: Visualized weights and "perfect" O, X, U, T images projected onto $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$

From Figure 34, we can see that none of the weights $\boldsymbol{w}_i$ are parallel to $\boldsymbol{e}_1$, otherwise, the projected weight vector would lie along the $\boldsymbol{e}_1$-axis. However, we could see that $\boldsymbol{w}_i$ are approximately a linear combination of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. Specifically, $\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3$, and $\boldsymbol{w}_4$ are approximately in the direction of $(\boldsymbol{e}_1 + \boldsymbol{e}_2), (-\boldsymbol{e}_1 - \boldsymbol{e}_2), (\boldsymbol{e}_1 - \boldsymbol{e}_2)$, and $(\boldsymbol{e}_2 - \boldsymbol{e}_1)$, respectively. Note that in the original $\mathbb{R}^9$ vector space, $\boldsymbol{w}_i$ still have components in the direction of $\boldsymbol{e}_3, \boldsymbol{e}_4, \ldots, \boldsymbol{e}_9$ which may be significant. However, because the eigenvalues of these components are much lower compared to the eigenvalues of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, they do not play a significant role in terms of recognition. This discrepancy is however inevitable, since PCA ranks the order of importance amongst the eigenvectors, while NN actually helps us classify the images. Therefore, both algorithms have different ways of segmenting the different classes of images and do not seem to perform identically in this case.

From the three examples, we see that 2NN and PCA only seem to agree on trivial cases and deviate when the complexity of the example increases. In the following section, we will further highlight the differences between these two methods, where one is successful in classifying an example but not the other.

2.3.3 Counterexamples for 2NN and PCA

Two Ellipses Counterexample 1: PCA Fails

In the following example (Figure 35), there are two classes of images of two pixels (i.e., two ellipses). When PCA is carried out, the two eigenvectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are produced (see Figure 35). On the other hand, 2NN would produce weights $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ such that their corresponding "visual" weights, w_1^* and w_2^* are approximately parallel to $\boldsymbol{e}_1$. Additionally, to classify class 1 from class 2, $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ will point in opposite directions.

In this example, however, PCA fails to classify the images appropriately. To classify images using PCA, one has to project the images onto $\boldsymbol{e}_1$. However, when this is done, classes 1 and 2 will overlap, making these two classes indistinguishable (see Figure 35). In fact, it might be better, in terms of recognition, if we project the images onto $\boldsymbol{e}_2$, which would result in less overlap than the projection onto $\boldsymbol{e}_1$.

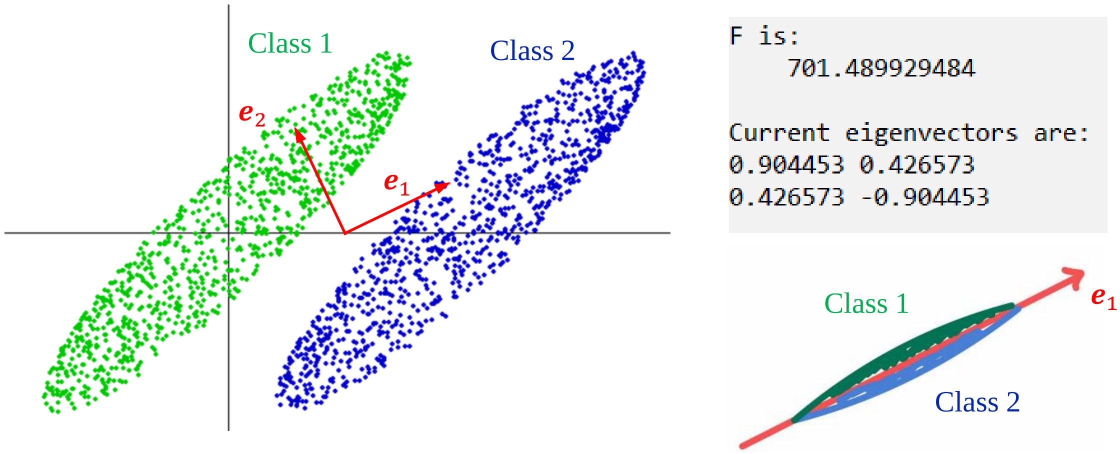


Figure 35: Visualization of two ellipses dataset, eigenvectors, and projected images onto $\boldsymbol{e}_1$ for PCA on the two ellipses dataset

Three-blob Counterexample 2: 2NN Fails

For this example, we have three classes of two-pixel images (see Figure 36). In this case, PCA would easily pick the eigenvector approximately parallel to the three clusters of images (i.e., approximately passing through the centres of all the clusters), and there would be no overlap of images. However, for 2NN, the weights $\boldsymbol{w}_i$ are not able to "cut" and segment the different classes appropriately, such that class 2 could be classified accurately. In fact, looking at Figure 36, the "visual" weight $\boldsymbol{w}_2^*$ is not able to distinguish class 2 from the other classes. This also translates to a relatively high average cost, despite letting the network run 100 iterations.

These two counterexamples exemplify the differences and flaws in both algorithms. In the next chapter, we will suggest two more methods to cover these blindspots.

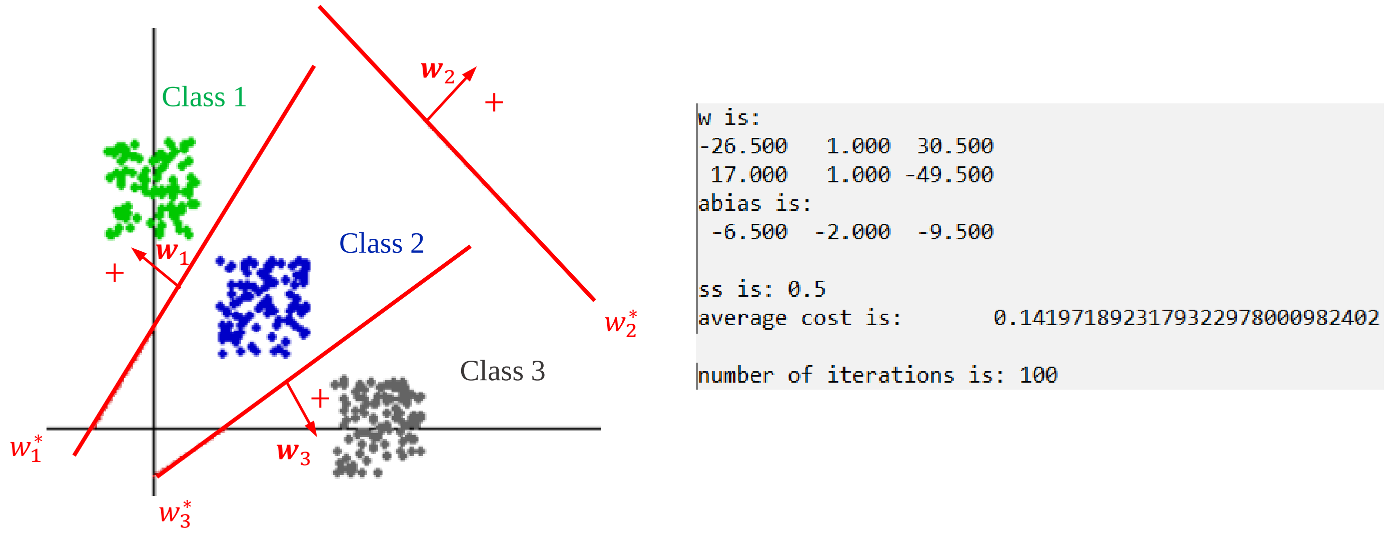


Figure 36: Visualization of three-blob dataset, and weights and biases for 2NN on the three-blob dataset

Chapter 3: Fisher's Linear Discriminant and 3-layer Neural Networks

3.1 Fisher's Linear Discriminant

3.1.1 Theory of Fisher Linear Discriminant

Variance

We shall attempt to resolve the issue faced in 2.3.3, regarding the failure of PCA on a dataset of two classes of two-pixel images. Again, the issue here is that the two clusters will overlap upon projection onto the first principal component produced by PCA, shown in Figure 37 as $\boldsymbol{e}_1$. Instead, choosing projection vector $\boldsymbol{e}$ would be a more ideal choice since the classes will no longer overlap.

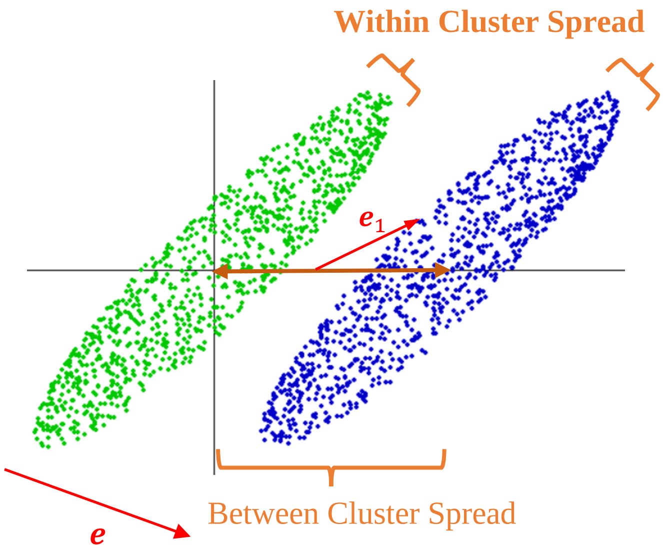


Figure 37: Illustration of within-cluster and between-cluster spread, and projection vectors for two ellipses dataset

To find this projection vector $\boldsymbol{e}$, we first introduce the concept of *within cluster* and *between cluster* spread. The former refers to the spread (or variance) within each class of images, while the latter refers to the variance between different classes of images (see Figure 37 as an illustration).

Now, to get the projection vector, we want to maximize a new function, G,

$$G = \frac{\boldsymbol{e}^T A_0 \boldsymbol{e}}{\boldsymbol{e}^T (\sum_i A_i) \boldsymbol{e}} \,,$$

where A_0 is the between cluster covariance matrix, and A_i are the within cluster covariance matrix, for $i = 1, 2, \ldots, r$, where r is the number of classes of images.

From Chapter 2, we know that $F = \boldsymbol{e}^T A \boldsymbol{e}$ gives us the variance of the dataset when projected onto $\boldsymbol{e}$. Similarly, the numerator of G gives the between cluster variance and the denominator gives the within cluster variance.

Finding $\boldsymbol{e}$ by Maximizing G

By maximizing G, we are trying to simultaneously maximize the between cluster variance and minimize the within cluster variance. Visually, this would keep the images in the same cluster "tight" together, while separating the different clusters from one another, thereby reducing overlaps between projected images of different classes. As in PCA, we will find $\frac{\delta G}{\delta \boldsymbol{e}}$ and set it to $\boldsymbol{0}$. Therefore, by quotient rule, we have

$$\frac{\delta G}{\delta \boldsymbol{e}} = \frac{\overbrace{(\boldsymbol{e}^T \sum_i A_i\, \boldsymbol{e})}^{\text{Scalar}} (\boldsymbol{e}^T A_0) - \overbrace{(\boldsymbol{e}^T A_0 \boldsymbol{e})}^{\text{Scalar}} (\boldsymbol{e}^T \sum A_i)}{(\boldsymbol{e}^T \sum_i A_i\, \boldsymbol{e})^2} = \boldsymbol{0}$$

$$\Rightarrow \boldsymbol{e}^T A_0 = \underbrace{\frac{\boldsymbol{e}^T A_0 \boldsymbol{e}}{\boldsymbol{e}^T \sum_i A_i\, \boldsymbol{e}}}_{G} \left(\boldsymbol{e}^T \sum_i Ai\right)$$

Since A_0 and $\sum_i A_i$ are symmetric,

$$\Rightarrow A_0 \boldsymbol{e} = \lambda \left(\sum_i Ai\right) \boldsymbol{e},$$

where $\lambda = G \in \mathbb{R}$.

The solutions to the above equation are called *generalized eigenvectors*, since if $(\sum_i A_i)^{-1}$ exists, then

$$\left(\sum_i A_i\right)^{-1} A_0 \boldsymbol{e} = \lambda \boldsymbol{e} \Rightarrow \boldsymbol{e} \text{ is an eigenvector of } \left(\sum_i A_i\right)^{-1} A_0.$$

Closed Form Solution for Dataset with Two Classes

We shall use a simple example to illustrate this. In this example, we will have a dataset made up of two classes of two-pixel images (see Figure 38). We can assume that the images in the dataset have mean of $\boldsymbol{0}$. Let A_1 and A_2 be the within cluster covariance matrix for class 1 and 2, respectively. Also, let $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ be the average of the images in class 1 and 2 respectively. We shall also let A_0 be the between cluster covariance matrix, and $\boldsymbol{e}_1$be the projection matrix that maximizes G.

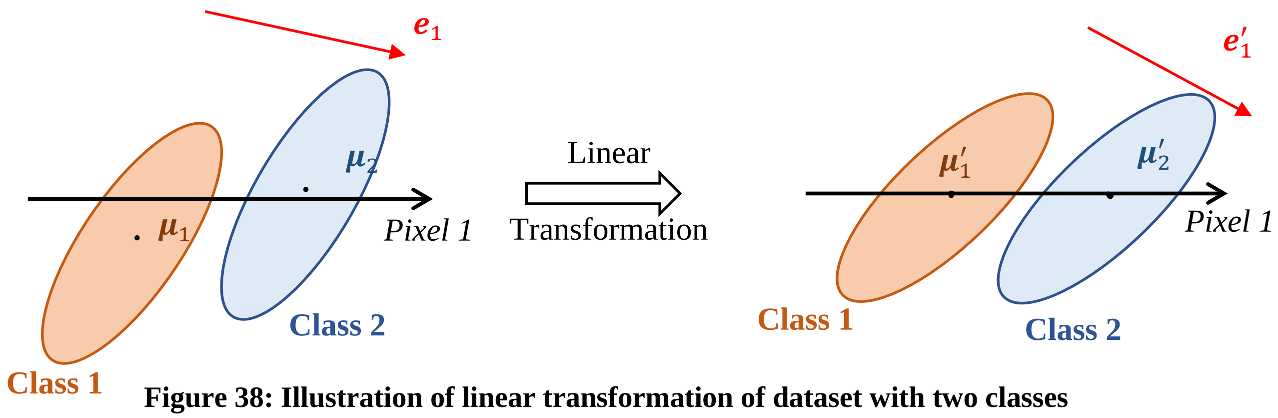


Figure 38: Illustration of linear transformation of dataset with two classes

From above, $\boldsymbol{e}_1$ will satisfy the equation $A_0 \boldsymbol{e}_1 = \lambda(A_1 + A_2)$. Note that A_0, A_1, and A_2 are symmetric matrices. To find $\boldsymbol{e}_1$, we can first do a linear transformation on the dataset, such that $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ end up on the horizontal axis. This linear transformation will also transform A_0, A_1, A_2, and $\boldsymbol{e}_1$. Let their transformed correspondent be A_0', A_1', A_2', and $\boldsymbol{e}_1'$. Then, $\boldsymbol{e}_1'$ will also satisfy the equation $A_0' \boldsymbol{e}_1' = \lambda'(A_1' + A_2')$.

Note that $A_0 = \sum_{i=1}^{2} N_i (\boldsymbol{\mu}_i' - \boldsymbol{\mu}')(\boldsymbol{\mu}_i' - \boldsymbol{\mu}')^T = \sum_{i=1}^{2} N_i \left(\boldsymbol{\mu}_i' \boldsymbol{\mu}_i'^T\right)$, where $\boldsymbol{\mu}'$ is the average of the entire transformed dataset, which is $\boldsymbol{0}$. Since $\boldsymbol{\mu}_i' = \begin{pmatrix} x_i \\ 0 \end{pmatrix}$, then $A_0' = \begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix}$, for some $a \in \mathbb{R}$.

$$\Rightarrow A_0' \boldsymbol{e}_1' = \begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix} \boldsymbol{e}_1' = \beta \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \text{for some } \beta \in \mathbb{R}$$
$$\Rightarrow A_0' \boldsymbol{e}_1' \parallel \boldsymbol{\mu}_1' - \boldsymbol{\mu}_2'$$

This implies that $A_0' \boldsymbol{e}_1'$ is parallel to $\boldsymbol{\mu}_1' - \boldsymbol{\mu}_2'$, independent of their coordinates.

$$\therefore \gamma(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) = \lambda(A_1 + A_2)\boldsymbol{e}_1, \text{ for some } \gamma \in \mathbb{R}$$

Now, assuming $A_1 + A_2$ is invertible, we have

$$\boldsymbol{e}_1 = \frac{\gamma}{\lambda}(A_1 + A_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$$

As in PCA, we are only interested in the direction of $\boldsymbol{e}_1$. Therefore, we can take $\boldsymbol{e}_1$ to be,

$$\therefore \boldsymbol{e}_1 = (A_1 + A_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$$

As in seen here, for the simple case of two classes, two-dimensional images, we have a closed form solution, assuming also that $A_1 + A_2$ is invertible. The process of solving for $\boldsymbol{e}_1$ is known as Fisher's Linear Discriminant (FLD).

In the subsequent sections, we will illustrate the process of finding $\boldsymbol{e}_1$ with a specific example, using our programmes.

3.1.2 Codes for FLD

Generating Clusters for Dataset

```
t = -pi/4

for i = 1 to num_images
u(0,i) = 1
u(1,i) = ran(4)-2
x = sqrt(1-u(1,i)^2/4)
u(2,i) = (ran(2*x)-x)
u(1,i) = cos(t)*u(1,i) - sin(t)*u(2,i)
u(2,i) = -sin(t)*u(1,i) + cos(t)*u(2,i)
next i
```

1

```
for i = 1 to num_images
v(0,i) = 2
v(1,i) = ran(4)-2
x = sqrt(1-v(1,i)^2/4)
v(2,i) = (ran(2*x)-x)
v(1,i) = cos(t)*v(1,i) - sin(t)*v(2,i)
v(2,i) = -sin(t)*v(1,i) + cos(t)*v(2,i)
v(1,i) = v(1,i)+2: v(2,i) = v(2,i) + 0.5
```

2

Figure 39: Generating two clusters subroutine

This subroutine generates the two clusters (or classes) of data. Visually, the images would look like two rotated ellipses, one ellipse for each class. The codes in box 1 generates the first cluster while the codes in box 2 generates the second cluster. In Figure 39, *u(i,k)* and *v(i,k)* are the i^{th} component of the k^{th} image in class 1 and class 2, respectively.

To illustrate the advantage of FLD over PCA, we want to have at least two clusters. Therefore, to ensure no overlap of images in different classes, we can maximize the between cluster spread. We have also chosen to generate clusters that resemble ellipses (Figure 39) instead of, say, circles. The reason being that the latter could easily be classified using PCA instead of FLD, as the dataset would be symmetrical and hence there is a direction in which the projected clusters are not overlapping, provided they do not overlap in the first place.

Covariance Matrices

This subroutine calculates the between cluster matrix (A_0) and within cluster covariance matrices (A_1 & A_2). In Figure 40, *avg(i)*, *u_avg(i)* and *v_avg(i)* are the i^{th} component of $\boldsymbol{\mu}$, $\boldsymbol{\mu}_1$, and $\boldsymbol{\mu}_2$, respectively, where $\boldsymbol{\mu}$ is the average of the dataset and $\boldsymbol{\mu}_k$ is the average of the images in class k. Also, in Figure 40, *C_u(i,k)* and *C_v(i,k)* are the within cluster matrices A_1 and A_2,

```
label calculate_covMs

for i = 1 to dim_images
for k = 1 to dim_images
for j = 1 to num_images


C_u(i,k) = C_u(i,k) + (u(i,j) - u_avg(i))*(u(k,j) - u_avg(k))
C_v(i,k) = C_v(i,k) + (v(i,j) - v_avg(i))*(v(k,j) - v_avg(k))

next j
next k
next i

for i = 1 to dim_images
for k = 1 to dim_images

C_within(i,k) = C_u(i,k) + C_v(i,k)
C_between(i,k) = num_images*((u_avg(i)-avg(i))*(u_avg(k)-avg(k)) + (v_avg(i)-avg(i)*(v_avg(k)-avg(k))

next k
next i

return
```

Figure 40: Calculating covariance matrices subroutine

respectively. Lastly, *C_within(i,k)* and *C_between(i,k)* are $(A_1 + A_2)$ and A_0 respectively.

Note that $A_0 = \sum_{i=1}^{r} N_i(\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T$, where $N_1 = N_2 = num\_images$. Looking at the formula of A_0, we are treating each $\boldsymbol{\mu}_i$ as datapoints, find the covariance matrix associated to each $\boldsymbol{\mu}_i$, and sum them up. Recall that $F = \boldsymbol{e}^T A\boldsymbol{e}$ gives the variance of a dataset with covariance matrix A projected onto the vector $\boldsymbol{e}$, we could similarly use A_0 to find the variance between each $\boldsymbol{\mu}_i$, which can be interpreted as the between clusters variance.

Now, let X_k be the set of images in the k^{th} class. Then, $A_k = \sum_{\boldsymbol{x}_k \in X_k} (\boldsymbol{x}_k - \boldsymbol{\mu}_k)\,(\boldsymbol{x}_k - \boldsymbol{\mu}_k)^T$, which is the formula of covariance matrix, applied to just the k^{th} class.

Calculating G

```
label calculate_G


G = 0
G_between = 0
G_within = 0

for i = 1 to dim_images
for j = 1 to dim_images

G_between = G_between + e(i)*C_between(i,j)*e(j)
G_within = G_within + e(i)*C_within(i,j)*e(j)

next j
next i

G = (G_between)/(G_within)

return
```

Figure 41: Calculating *G* subroutine

This subroutine calculates the value of G for a given $\boldsymbol{e}$. Recall that G is the function (of $\boldsymbol{e}$):

$$G = \frac{\boldsymbol{e}^T A_0 \boldsymbol{e}}{\boldsymbol{e}^T(\sum_{i=1}^{r} A_i)\boldsymbol{e}},$$

where r is the total number of classes of images.

In Figure 41, *G_between* and *G_within* represent $\boldsymbol{e}^T A_0 \boldsymbol{e}$ and $\boldsymbol{e}^T(A_1 + A_2)\boldsymbol{e}$, respectively. These correspond to the numerator and denominator of G respectively.

Recall that we want to find $\boldsymbol{e}$ such that it maximizes G. One way of doing this is to find the solution to the equation:

$$A_0\boldsymbol{e} = \lambda\left(\sum_i Ai\right)\boldsymbol{e}$$

and observe if G is maximized. However, for this project, we will approach this problem numerically. That is, we use the same algorithm as in PCA, of randomly changing $\boldsymbol{e}$, orthogonalizing it, and normalizing it (restrained to $\|\boldsymbol{e}\| = 1$), before finding G and moving in the direction of highest G (see Section 2.1.2).

3.1.3 Illustration of FLD with Two Ellipses Dataset

Description of Dataset and FLD

In this section, we will illustrate the FLD algorithm using a similar dataset as in Section 2.3.3 (Figure 35) – a dataset consisting of two-pixel images from two classes. Note that given r number of classes, we can have a total of $(r-1)$ number of e_i's (Belheumer, p. 714); therefore, we could expect only $\boldsymbol{e}_1$ for this example.

We will check our generalized eigenvectors against that obtained from the closed form solution, since this dataset consists of two clusters in $\mathbb{R}^2$. That is, we will check if the generalized eigenvector produced by our programme is approximately parallel to the vector $(A_1 + A_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$. Therefore, we will also find $(A_1 + A_2), \boldsymbol{\mu}_1$, and $\boldsymbol{\mu}_2$ using our programme, which are *C_within*, *u_avg, and v_avg*. Note that the closed form solution is independent of A_0, therefore we do not need to explicitly output *C_between.*

Screenshots

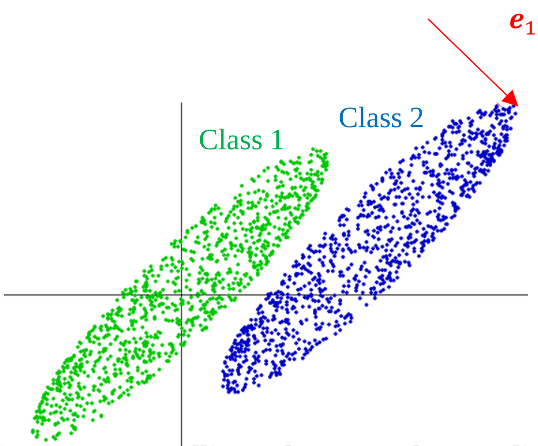
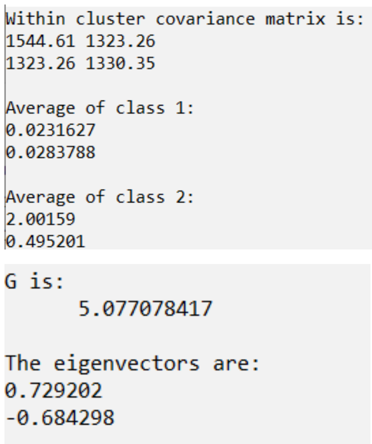


Figure 42: Two ellipses dataset and generalized eigenvector



```
Within cluster covariance matrix is:
1544.61 1323.26
1323.26 1330.35

Average of class 1:
0.0231627
0.0283788

Average of class 2:
2.00159
0.495201
```

```
G is:
       5.077078417

The eigenvectors are:
0.729202
-0.684298
```

Figure 43: Output of FLD for two ellipses dataset

Discussions and Comparisons

From Figure 43, we have $(A_1 + A_2) = \begin{pmatrix} 1544.61 & 1323.26 \\ 1323.26 & 1330.35 \end{pmatrix}$, $\boldsymbol{\mu}_1 = \begin{pmatrix} 0.023 \\ 0.028 \end{pmatrix}$, and $\boldsymbol{\mu}_2 = \begin{pmatrix} 2.002 \\ 0.495 \end{pmatrix}$, then $(A_1 + A_2)^{-1} = \begin{pmatrix} 0.00438 & -0.00435 \\ -0.00435 & 0.00508 \end{pmatrix}$.

$$\therefore (A_1 + A_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) = \begin{pmatrix} -0.00446 \\ 0.00369 \end{pmatrix}$$

Normalizing the vector above and taking its negative, we have $\boldsymbol{e}_1 = \begin{pmatrix} 0.770 \\ -0.637 \end{pmatrix}$, which is quite similar to the vector we obtained from the FLD algorithm, i.e., $\begin{pmatrix} 0.729 \\ -0.684 \end{pmatrix}$. This means that $(A_1 + A_2)^{-1}(\mu_1 - \mu_2)$ is approximately parallel to the generalized eigenvector obtained from our programme.

The closed form solution of $\boldsymbol{e}_1 = (A_1 + A_2)^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ suggests that $\boldsymbol{e}_1$ is the vector pointing in the direction of the vector connecting the means of the clusters (i.e., $\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2$) but tilted at an angle. This extent to which $\boldsymbol{e}_1$ is tilted, is determined by $(A_1 + A_2)^{-1}$. However, in general, for cases more than two classes of images, the closed form solution cannot be used, as the means of the clusters may not be collinear (see Figure 44). Therefore, in such cases, we may want to use the algorithm of finding $\boldsymbol{e}_1$ that maximizes G instead.

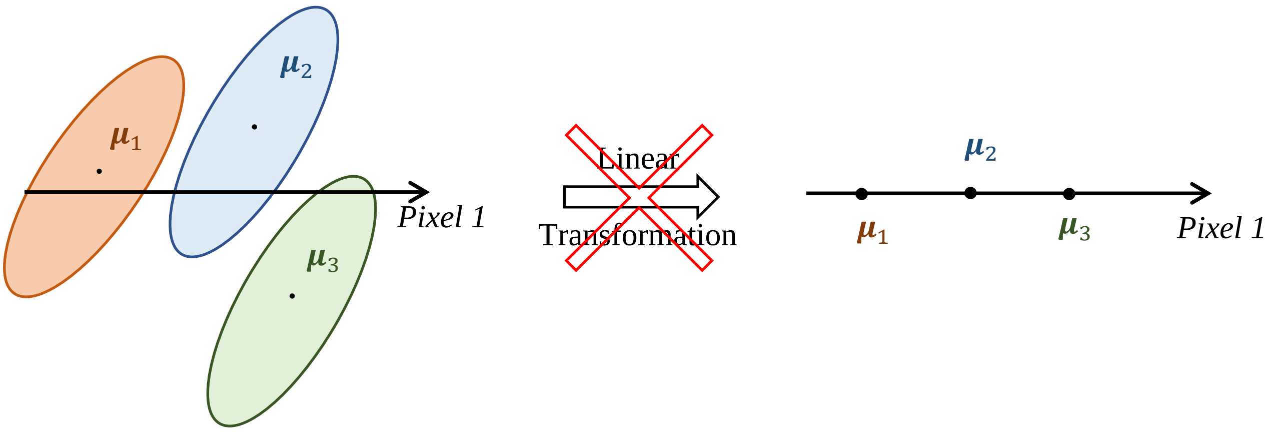


Figure 44: Illustration of the failure of closed form solutions for three clusters

3.2 3-Layer Neural Network

3.2.1 Theory of 3-Layer Neural Network (3NN)

Forward Propagation

In Section 2.2, we looked at 2NN, where there are only two *layers* – the input layer and the output layer. In this section, we will focus on 3NN, where there is an intermediate layer called the *hidden layer*. Figure 45 below shows a 3NN with three nodes each in the input (c_1, c_2, c_3) and hidden layer (b_1, b_2, b_3), and two nodes in the output layer (a_1, a_2)

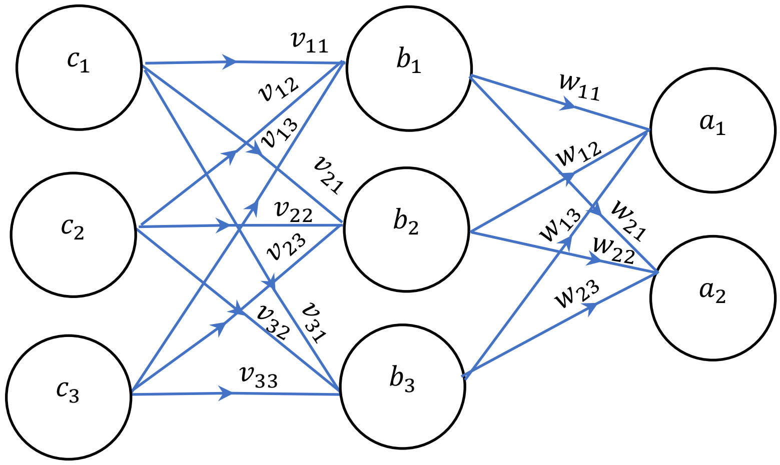


Figure 45: Simple example of 3NN

Because of the additional hidden layer, in this case, $\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$, there are now more weights in the entire network (i.e., v_{ik}, for $i = 1, 2, 3$, and $k = 1, 2, 3$. This would result in a more complex network. Furthermore, $b_i = \sigma_1(\sum_k v_{ik}\, c_k)$ and $a_j = \sigma_2(\sum_r w_{jr} b_r)$, where σ_1 and σ_2 are activation functions, further adding on to the complexity of the network. Note that in principle, we may choose two different activation functions for σ_1 and σ_2, but for this project we set $\sigma_1 = \sigma_2$ to be the sigmoid function. Lastly, with more nodes being included in both activation function σ_1 and σ_2, we will also have more biases which will then control the threshold of each activation function. All in all, we have more parameters (i.e., weights and biases) to tweak, allowing for more refined approximation to the true function.

Back Propagation

Our cost function will be similar to that used in 2NN. That is, will use the average cost of the network as

$$\text{Avg. Cost} = \frac{1}{N} \sum_{images} \|\boldsymbol{a}_i - \boldsymbol{p}_i\|^2,$$

where N is the number of images in the population. Similar to 2NN, we aim to lower the average cost of the network with respect to a training set sampled from the population. To this end, we will use the same numerical approach as in 2NN but doing it now with respect to more parameters (i.e., both the weights V and W, as well as the biases for each b_i and a_j). This is unlike the calculus approach where chain rule is being utilized, therefore, has to be done systematically because some parameters are dependent on others. In this case, we would change all weights and biases in the order: a_bias → W → b_bias → V. This chain of changes forms an iteration. Since we iterate through this order, we are effectively changing all the parameters at the same time.

Comparing 2NN and 3NN

In section 2.2 and 2.3, we saw that the weights in 2NN act as "cuts" in the image space, separating the different classes from one another. However, we have also seen in the example in 2.3.3 that this does not always allow us to correctly classify all classes. Since 3NN has an additional (hidden) layer, it can possibly do more, even if σ_2 is the identity function. The weights in the first layer could probably act as "cuts" like in 2NN but the weights in the second layer might allow for other behaviours, such as synthesis of the different cuts, or even add different kinds of "cuts".

Extending this idea, we could theoretically add more layer, allowing for a richer structure and better predictive power. However, this does not guarantee us always knowing what each layer is doing, as the neural network becomes more of a 'black box'.

3.2.2 Codes for 3NN

Computing The Neural Network

```
label compute_NN

for i = 1 to bnum                1
b(i) = 0
for j = 1 to cnum
b(i) = b(i) + v(i,j)*c(j)
next j
next i

for i = 1 to bnum
b(i) = b(i) + bbias(i)
next i

for i = 1 to bnum
b(i) = sigmoid(b(i))
next i

for i = 1 to anum
a(i) = 0                         2
for j = 1 to bnum
a(i) = a(i) + w(i,j)*b(j)
next j
next i

for i = 1 to anum
a(i) = a(i) + abias(i)
next i


for i = 1 to anum
a(i) = sigmoid(a(i))
next i

return
```

Figure 46: Computing 3NN subroutine

In this subroutine, we compute the value of each a_i given the values of the input nodes, $\boldsymbol{c}$. Here, *cnum* , *bnum*, and *anum* refer to the number of nodes in the input layer $(\boldsymbol{c})$, the hidden layer $(\boldsymbol{b})$, and the output layer $(\boldsymbol{a})$, respectively. Also, *abias(i)*, *bbias(i)*, *v(i,j)*, and *w(i,j)*, refer to the i^{th} component of $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, the weight between the b_i and c_j, and the weight between a_i and b_j, respectively, where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are the biases of $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively.

The codes in box 1 calculates b_i using

$$b_i = \sigma\left(\boldsymbol{v}_i^T \boldsymbol{c} + \beta_i\right),$$

where $\boldsymbol{v}_i$ are the vector of weights connecting to b_i and β_i is the bias for b_i. That is, we place the otherwise affine function $\boldsymbol{v}_i^T c + \beta_i$ into the sigmoid activation function, $\sigma(x) = \frac{1}{1+e^{-x}}$.

Then, the codes in box 2 takes the non-linear output, b_i, and input them into a second layer of sigmoid activation function to calculate a_j as

$$a_j = \sigma\left(\boldsymbol{w}_j^T \boldsymbol{b} + \alpha_j\right),$$

where $\boldsymbol{w}_j$ is the vector of weights connecting to a_j and α_j is the bias for a_j. Therefore, we have two layers of non-linear outputs.

Changing The Parameters

This is a collection of subroutines that change the different parameters. This is executed in the order: $\alpha_i \rightarrow w_{kr} \rightarrow v_{st} \rightarrow \beta_j$, for all $i, k, r, s, t,$ and j. Each of the subroutine is similar to the subroutine shown in Figure 18 in section 2.2.2, i.e., we move each of the weight and bias in the direction that lowers the average cost of the network across the training set. Now, we have more layers of nodes, which means that the runtime for each iteration will be longer, as there are more parameters to work through.

```
gosub change_abias

gosub change_w

gosub change_bbias

gosub change_v
```

Figure 47: Collection of subroutines to change weights and biases

Visualizing Weights and Classification

1

```
label conduct_trials
for trial = 1 to num_trials
c(1) = ran(1.2) : c(2) = ran(1.2)
gosub compute_NN
gosub colour_v
gosub colour_w
next trial
return
```

```
***************************
```

2

```
label colour_v
colouring = 0.5

if abs(b(1)-colouring) < v_threshold then
colour 0,100,200
fill circle H(d(1)), V(d(2)), 2
endif

if abs(b(2)-colouring) < v_threshold then
colour 200,50,50
fill circle H(d(1)), V(d(2)), 2
endif

if abs(b(3)-colouring) < v_threshold then
colour 100,0,200
fill circle H(d(1)), V(d(2)), 2
endif

return
```

Figure 48: Generate random points and colouring points subroutines

We have also used a programme to visualize the different weights. These subroutines are used to trace out the weights and colour the different classifications by the 3NN. Here, *num_trials* and *c(i)* refer to the number of points we will randomly generate and the i^{th} component of the randomly generated point, $\boldsymbol{c}$.

The codes in box 1 uses the Monte Carlo method to "test" our model. Note that we randomly generate point $\boldsymbol{c}$, such that $0 \leq c(i) \leq 1.5$. After we randomly generate a point $\boldsymbol{c}$, we put it through *compute_NN* and colour the point to signify different weights. For example, if a randomly generated point $\boldsymbol{c}$ produces $b(1) = 0.5$, then it would be coloured with a certain colour, dictated by the subroutine in box 2. This colour would show the boundary of v_1^* . Also, *colour_w* (in box 1) is equivalent to colouring the point according to how the 3NN classifies it.

The subroutine *colour_v* (box 2) colours the weights $\boldsymbol{v_i}$ connecting to the node b_i. Here, *colouring* controls the value of b_i that we use to trace the weights, while *v_threshold* controls the thickness of the weights being traced out. Also, we set $colouring = 0.5$, which is the threshold of the sigmoid function, but we could also set $colouring = 1$. For *v_threshold* $= 0.01$ and $colouring = 0.5$ indicates that if $b(i) = 0.5 \pm 0.01$, then we would colour the boundaries of the cut $\boldsymbol{v_i}$. Now, for *v_threshold*$= 0.01$ and $colouring = 1$ indicates that if $b(i) = 1 \pm 0.01$, then we would colour the region in which the weights are pointing in. Through these, we are able to visualize the weights for examples in $\mathbb{R}^2$.

<u>3.2.3: Illustration of 3NN with Three-blob Dataset</u>

Description of Dataset and Initial Parameter Values

To illustrate the 3NN algorithm, we would use the same "3-blob counterexample" example in section 2.3.3, which 2NN fails to perform, and check if 3NN works well on it. Therefore, our dataset would consist of three classes of two-pixel images as shown in Figure 49, copied from section 2.3.3. Because the images are vectors in $\mathbb{R}^2$, then $c\_num = 2$, and since there are three classes, then $a\_num = 3$. Furthermore, we set $b\_num = 3 = a\_num$, the initial weights $v_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$ and $w_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$, and initial biases $abias = bbias = \boldsymbol{0}$. Lastly, the images in class 1, 2, and 3 are generated by adding noise of ± 0.1 to the vectors $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $\begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$, and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, respectively.

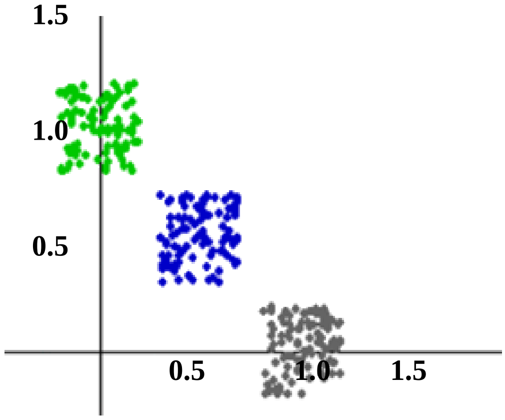


Figure 49 : Three-blob Counterexample from Section 2.3.3

Screenshots

```
v is:
-13.000 -13.000   4.500
  7.500  36.000 -13.000
bbias is:
 -3.000   1.000  -1.500

w is:
 34.000 -34.000 -34.500
-14.000  16.000 -34.500
-35.500 -26.500  36.000
abias is:
 -0.500  -1.500   8.000

ss is: 0.5
average cost is:       0.0000000000871594330240846382
number of iterations is: 71
```

Figure 50: Weights and Biases for 3NN on three-blob dataset

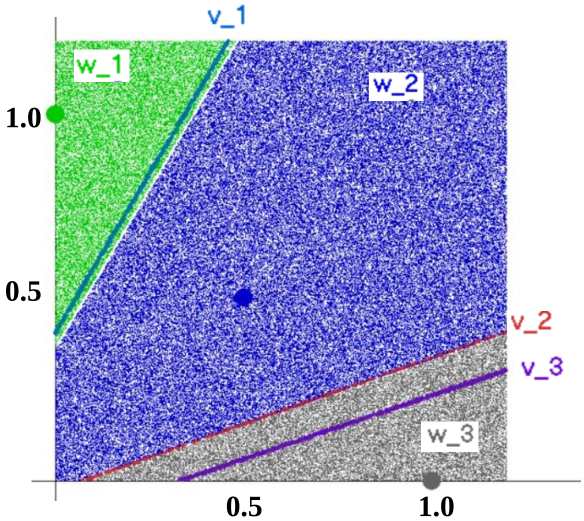


Figure 51: Weights visualized for 3NN on three-blob dataset, where the three dark dots are the centres of the clusters

Discussions

From Figure 50, we see that the average cost is very small at the 71st iteration. This suggests that 3NN seems to succeed in what 2NN fails in. Looking at Figure 51, we see that the weight vectors in the first layer, $\boldsymbol{v}_i$, still act as “cuts” to separate the different clusters into different “patches”. This is consistent with what we discussed about the sigmoid function and the property afforded from the large values for the weights in section 2.2. What is interesting are the weight vectors in the second layer, $\boldsymbol{w}_j$: They seem to “synthesise” the “cuts” and choose the “patch” to classify the clusters accordingly. For example, w_1 is to the “left” of v_1 (Figure 51), and w_1 is the “patch” in which the 3NN classifies images to be in class 1. It is as though v_1 separates $\mathbb{R}^2$ into two regions, and w_1 chooses the “left” region to be classified as class 1. This also suggests that the reason 2NN fails is because 2NN only allows for “cuts” and is unable to identify the “patch” that class 2 is in. We see that 3NN manages to identify the “patch” by taking "to the left of v_2 and to the right of v_1". Now, with this additional property of identifying “patches” along with taking “cuts”, 3NN seems infalliable and could classify any kind of datasets. However, we shall see that this is not really the case – at least not in practice.

3.3 Comparison Between FLD and 3NN

In this section, we will compare FLD with 3NN in different examples.

Description of Dataset, FLD, and Initial Parameter Values for 3NN

The dataset for this example consists of two classes of two-pixel images, where the first class of images form a crescent around a core that makes up the second class of images (see Figure 52). Since there are two classes of images, FLD will produce only one generalized eigenvector. Here, $A_0 = \sum N_i(\boldsymbol{\mu}_i - \boldsymbol{\mu})(\boldsymbol{\mu}_i - \boldsymbol{\mu})^T$, where N_i is the number of images in class i, $\boldsymbol{\mu}_i$ is the average of the images in the class i, and $\boldsymbol{\mu}$ is the average over the entire dataset, and $A_k = \sum_{\boldsymbol{x}_k \in X_k}(\boldsymbol{x}_k - \boldsymbol{\mu}_k)(\boldsymbol{x}_k - \boldsymbol{\mu}_k)^T$, where X_k denotes class k, and $\boldsymbol{x}_k$ is an images in class k. Note that $N_1 = N_2 = 10$. Since the images have two pixels, A_0 and A_k are 2×2 symmetric matrices. As for the 3NN, we will set $cnum = 2 =$ number of pixels, $bnum = 3 =$ the number of cuts we think can separate the two classes, and $anum = 2 =$ number of classes. Again, we set the initial biases to be $\mathbf{0}$ and the initial weights to be $v_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$ and $w_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0, \text{ if } i \neq j \end{cases}$.

Screenshots

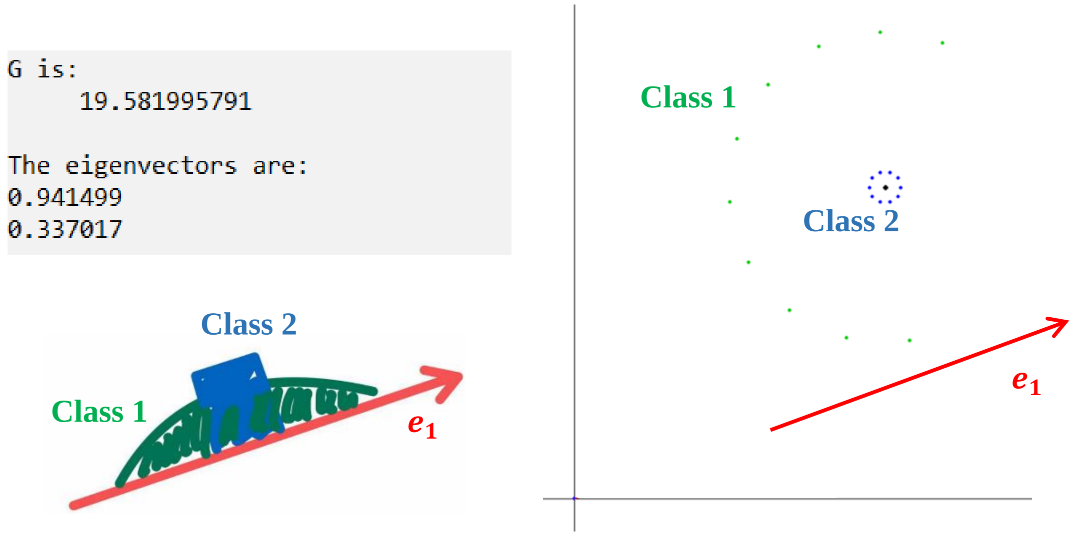


Figure 52: Visualization of crescent surrounding a core dataset, generalized eigenvector, and projected images for FLD on crescent surrounding a core dataset

```
v is:
   -7.250     7.000    -2.000
    7.000    25.750   -10.000
bbias is:
   -1.750   -24.250     6.000

w is:
   35.500   -34.250
  -11.250    12.000
   37.000   -30.500
abias is:
   -0.750    -0.500

ss is: 0.25
average cost is:       0.0000093750467022137726758046
number of iterations is: 153
```

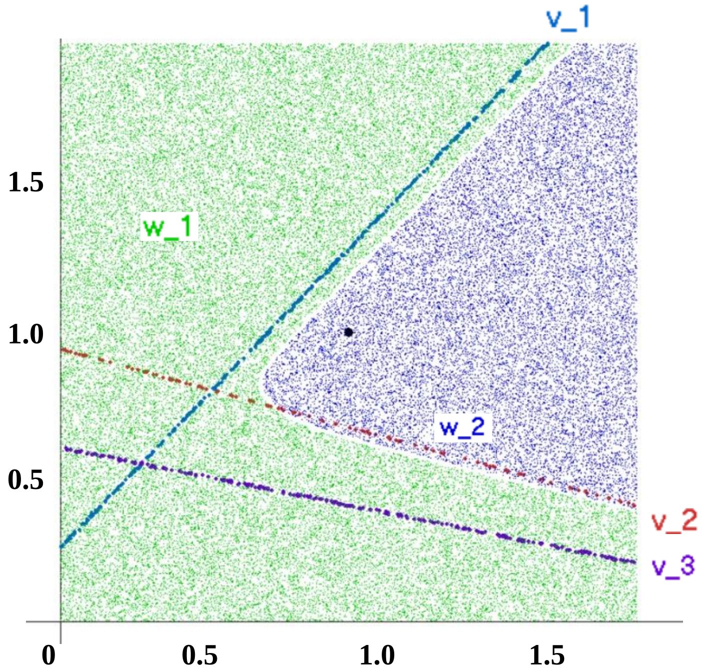


Figure 53: Weights and biases, and their visualization, for 3NN on crescent and a core dataset shown in Figure 52 where the dark dot represents the centre of the core

Discussions and Comparisons

In Figure 53, the dot coloured in black is the point $(1,1)$, where the centre of the core is. By looking at Figure 52 and 53, it is clear that FLD fails to separate the two classes of images upon projection (therefore recognition will fail), but the 3NN are able to achieve a low average cost. Also, the visual weights v_i's manage to segment out class 1 and class 2. From these, we can see that 3NN and FLD are not doing the same thing, as 3NN manages to recognize but FLD does not.

Initially, we expected the visual weights v_i's to form a triangle that encloses the core, so that images from class 1 are separated from class 2. This consideration motivated our choice of $bnum = 3$ so that we have 3 "cuts". However, if we look at v_2 and v_3, they seem to be parallel to one another. Indeed, if we look at the weight vectors $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$, they are approximately parallel but pointing in different direction. This suggests that v_3 is not required for the NN to classify the images accordingly. To test this, we ran the programme for $bnum = 2$ but with initial weights vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ as the ones obtained in Figure 53, i.e., $\boldsymbol{v}_1 = \begin{pmatrix} -7.25 \\ 7.00 \end{pmatrix}$ and $\boldsymbol{v}_2 = \begin{pmatrix} 7.00 \\ 25.75 \end{pmatrix}$. We got similar result but with less iterations (see Figure 54). Even with fewer nodes, the 3NN could still do its thing and obtain a low average cost. Also, the initial weights used are important, as setting the initial weights as $v_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $v_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ could not obtain the same result.

```
v is:
   -9.750    -1.750
    8.500    33.750
bbias is:
   -0.500   -23.250

w is:
   29.250   -29.250
  -18.500    18.500
abias is:
    6.000    -6.000

ss is: 0.25
average cost is:       0.0000095260449696206741521109
number of iterations is: 112
```

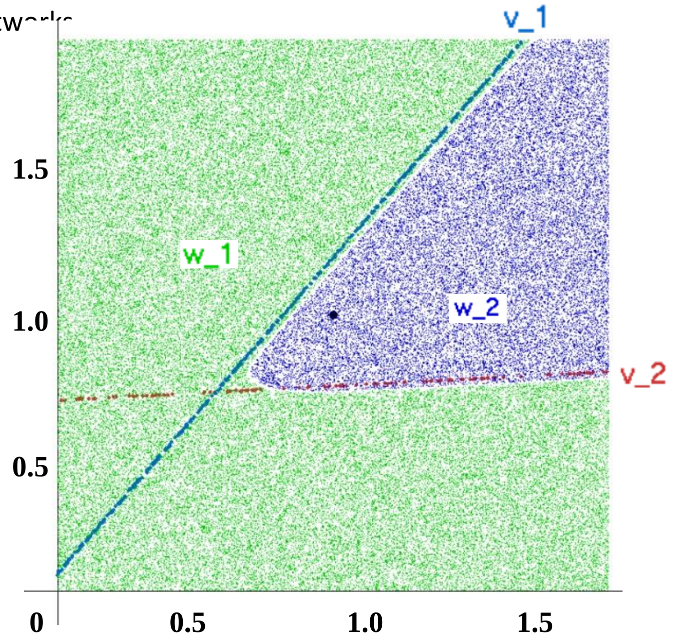


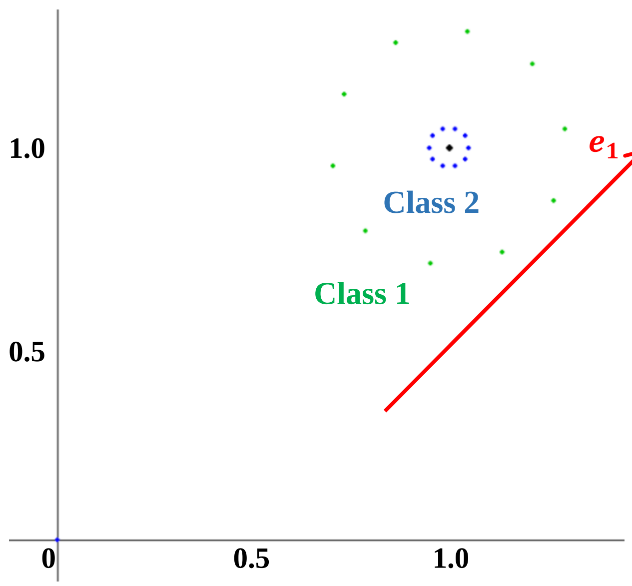
Figure 54: Weights and biases, and their visualization, for 3NN on crescent and a core dataset shown in Figure 52 where the dark dot represents the centre of the core and $bnum = 2$

3.3.2: FLD and 3NN on Annulus with a Core Dataset

Description of Dataset, FLD, and Initial Parameter Values for 3NN

We shall push the example above further and enclose the entire core with an annulus. The dataset now consists of two classes of two-pixel images, where the first class of images form an annulus around a core of images in the second class as shown in Figure 55. Like the example above, our points will be evenly spaced, but dense enough to not be linearly separable. For FLD, we will again expect one generalized eigenvector since there are only two classes of images. And, as we saw above, FLD would probably fail in classification since this is "worse" than the above in the sense that the entire core is now enveloped by the annulus. A modification we made, compared to the other examples, is that we randomly selected the values for the weights connecting from the input layer to the first hidden layer (i.e., $\boldsymbol{v}_i$ for the 3NN). Therefore, $v_{ik} \in [-1,1]$, for $i = 1, 2, \dots, bnm$ and $k = 1, 2, \dots, cnum$. We found that this allows the neural networks to perform better. Visually, it is randomly choosing the initial "cuts". The varying performance is due to the many local minima the cost function has. The rest of the weights and biases are the same as previous examples, i.e., for any weight m_{rs}, $m_{rs} = \begin{cases} 1, \text{if } r = s \\ 0, \text{if } r \neq s \end{cases}$ and for any biases $\boldsymbol{\gamma}$, $\boldsymbol{\gamma} = \boldsymbol{0}$.

Screenshots




```
G is:
     43.243242934

The eigenvectors are:
0.707047
0.707167
```

Figure 55: Annulus and core dataset, generalized eigenvector, and projection of images for FLD on annulus and core dataset, where the black dot is the centre of both the annulus and the core

```
v is:
   -7.651     4.260    -4.089
    2.056   -10.945    -1.561
bbias is:
    3.990     3.019     6.004

w is:
  301.602  -293.401
  564.171  -556.224
 -331.021   323.499
abias is:
   91.745   -89.759

ss is: 4.44089e-16
average cost is:       0.0000000000000000000000000001
number of iterations is: 10000
```

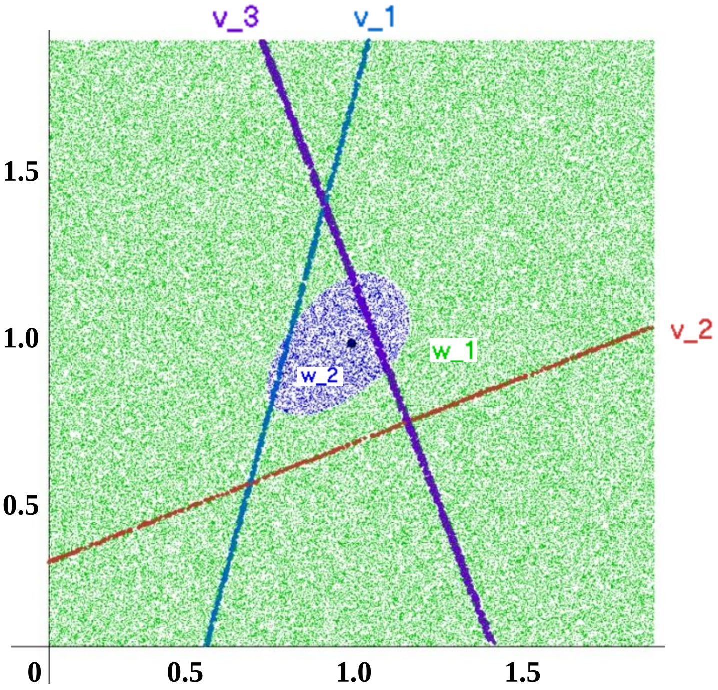


Figure 56: Weights and biases, and their visualization, for 3NN on annulus and core dataset in Figure 55, where the black dot is the centre of the annulus and the core

Discussion and Comparisons

In Figure 55, the black-coloured dot is the centre of the annulus and the core, with coordinates (1,1).

From Figure 55, we can see that FLD fails again in this regard, as the projected images from the two classes, onto $\boldsymbol{e}_1$, overlap with one another. It seems that FLD cannot handle dataset that are not linearly separable, like in the two cases we have seen in section 3.3. Again, looking at the average cost in Figure 56, 3NN seems to be able to classify the images successfully, able to isolate the core out from the annulus, albeit having to run many iterations. It should, therefore, be able to segment out a full, dense, annulus similarly. Furthermore, the visualized weights in Figure 54 fit our intuition that the cuts will form a triangle to isolate out the core from the annulus.

From the examples we have seen, 3NN would sometimes follow our intuition in recognizing the different images, but even when it does not, it is still able to achieve a low average cost and successfully classify the images. Also, 3NN seems to be able to deal with linearly separable dataset well, and even some non-linearly separable ones too. But, as we will see in the next section, 3NN may not be as invincible as it seems to be.

<u>3.3.3 3NN and 4NN on Annulus with An Inner Core and An Outer Core Dataset</u>

Description of Dataset and Initial Parameter Values for 3NN and 4NN

In this section, we will test the limit of 3NN by considering a dataset consisting of two classes of two-pixel images: The first class is made up of an inner and outer core, while the second class is an annulus surrounding the inner core (see Figure 57). We shall pit 3NN against 4NN (see Figure 58) – that is, a 3NN with an additional hidden layer preceding the output layer – and observe whether they will perform any differently.

The parameters for the 3NN will be: $cnum = 2 = \text{number of pixels}, bnum = 8,$ and $anum = 2 = \text{number of classes}$. As for the parameters for the 4NN, we have: $dnum = 2 = \text{number of pixels}, cnum = 4, bnum = 4, anum = 2 = \text{number of classes}$. We made sure that the number of nodes in the hidden layers for both 3NN and 4NN are the same. Lastly, similar to the initialization of weights and biases seen in section 3.3.2, we shall only randomize the weights connecting the input layer and the first input layer for both 3NN and 4NN, while keeping the rest the same.

Screenshots

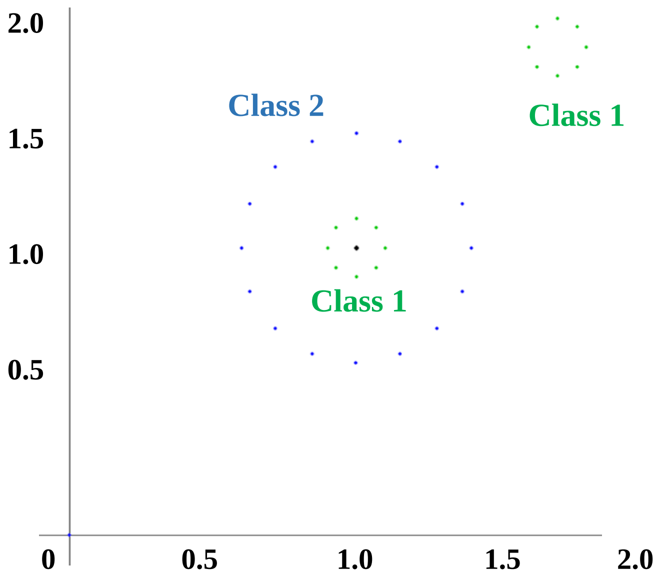


Figure 57: Visualization of annulus and two cores dataset

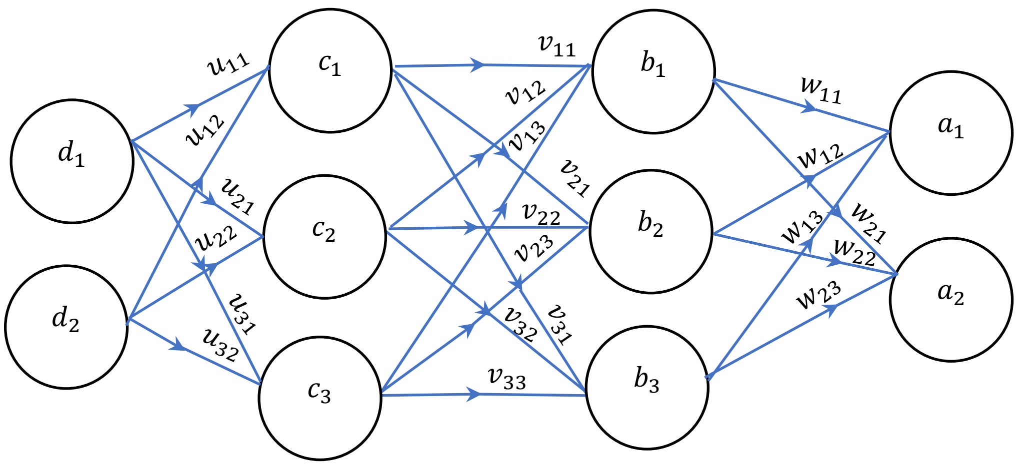


Figure 58: Simple example of 4NN

```
v is:
    9.317    -5.814    -2.955     5.547    -6.334     0.438    -8.274     3.625
   -8.131     4.158    -0.977     0.941     8.871    -1.249    10.101    -1.501
bbias is:
   -2.125    -0.250     0.625    -7.000    -0.062     0.375    -3.125    -3.125

w is:
   15.125   -14.688
 -101.875   101.188
 -123.063   124.563
  -58.625    58.813
   50.875   -50.375
  -38.000    38.313
   20.937   -20.500
   21.750   -22.812
abias is:
   -2.313     1.750

ss is: 0.0625
average cost is:        0.00199300838187984385996628752
highest cost is:        0.0225305498929401 with predictions:    2.00000000    0.10599765    0.89372184
highest cost is:        0.0225305498929401 corresponding to:    2.00000000    1.28284271    1.28284271
number of pixels is: 2
number of iterations is: 1500
```

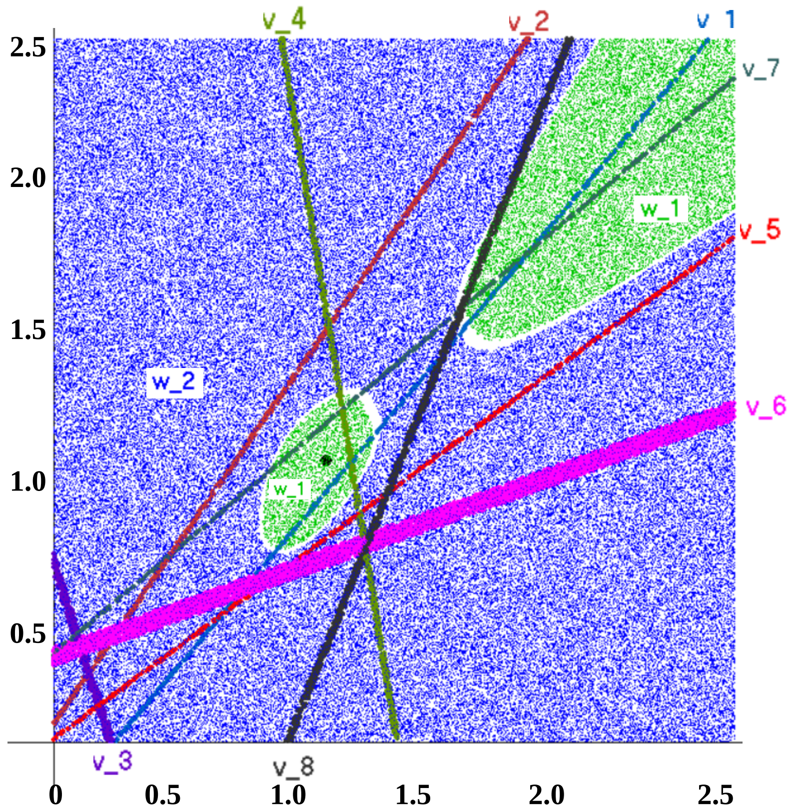


Figure 59: Weights and biases, and their visualization, for 3NN on annulus with two cores dataset in Figure 57, where the black dot is the centre of the annlus and the inner core

```
u is:
   0.527   -1.631    1.651    7.230
   1.696   -0.192    8.975   -5.696
cbias is:
  -3.500    2.250   -7.500   -1.250

v is:
  41.500    3.250  -13.250  -60.500
 -64.500   -5.000   -3.250   11.500
   3.250   -2.250    3.750   79.125
  -0.250    4.000    6.500   53.375
bbias is:
  -2.875    1.250   -1.250  -78.250
w is:
 194.250 -196.250
-174.750  177.375
 118.750 -120.625
  37.125  -37.625
abias is:
 -36.750   37.250

ss is: 0.125
numimages is: 32
average cost is:       0.0000000000000000000002008208
highest cost is:       0.0000000000000000 with predictions:   2.00000000   0.00000000   1.00000000
highest cost is:       0.0000000000000000 corresponding to:   2.00000000   1.28284271   1.28284271
number of pixels is: 2
number of iterations is: 1500
```

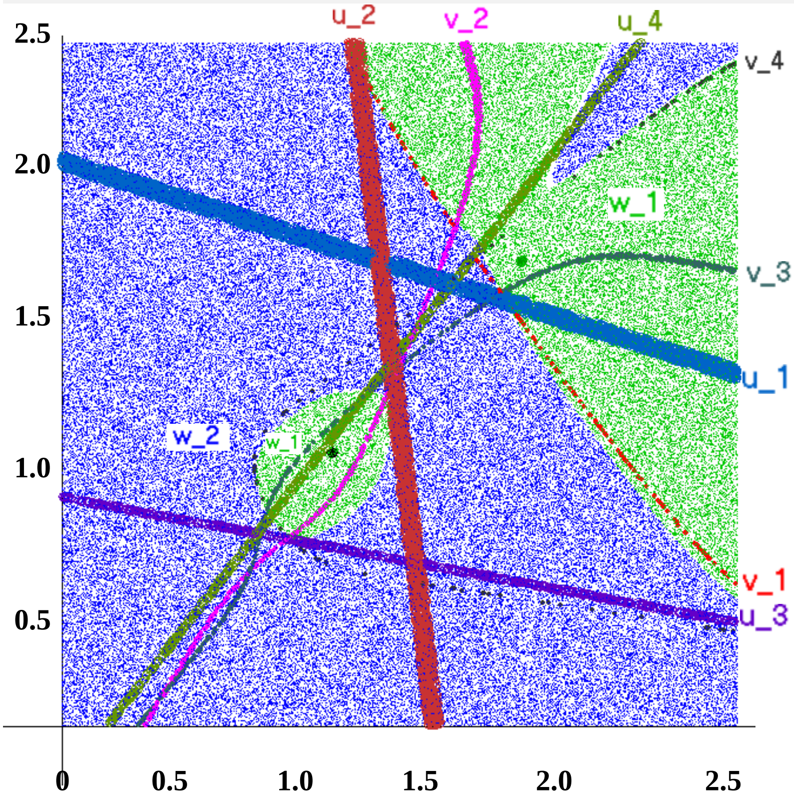


Figure 60: Weights and biases, and their visualization, for 4NN on annulus with two cores dataset in Figure 57, where the black dot is the centre of the annlus and the inner core

Comparisons and Discussions

Both networks ran a total of 1500 iterations. In Figure 59, we have the outputs of the weights and biases for the 3NN, together with the visualized weights. The first output in the red box shows the highest cost and the corresponding predictions given by the 3NN. The three values for the predictions are, the class, the value of $a(1)$, and the value of $a(2)$, respectively. The second output in the red box shows the class, followed by the first- and second-pixel value of the datapoint with the highest cost. Referring to both Figure 59 and 60, the black-coloured dot in the region labelled w_1 indicates the point $(1,1)$, the centre of the annulus and the inner core.

In Figure 59 and 60, we can see that the 4NN gives a much lower average cost as compared to the 3NN. Furthermore, the different weights in 3NN and 4NN look different from one another. In particular, the visualized weights linking the first hidden layer to the second hidden layer (i.e.,v_i) for 4NN are no longer straight lines, but curves. This seems to enable the 4NN to classify the images better, resulting in a lower average cost.

In theory, 3NN should be able to approximate any function, according to the Universal Approximator Theorem. However, the average cost of 3NN here does not seem to be converging as quickly as the 4NN and may require more computing time. There may even be a need to increase the number of nodes in the hidden layer for the 3NN for the network to truly converge to as low as the 4NN here. Therefore, in practice, we may require more than three layers. As we saw in this example, more hidden layer leads to more complex boundaries, such as the nonlinear v_i's in Figure 60. Thus, it may be more efficient to use more layers than a wider 3NN, if the number of nodes remains a constant. However, the more layers we add to a neural network, the more inexplicable its behaviours are.

3.3.4 3NN and 4NN on Double Annuli with a Core

Description of Dataset and Initial Parameter Values for 3NN and 4NN

In this final example, we will look at another interesting dataset consisting of two classes of two-pixel images. The first class of images form the inner core as well as the outermost annulus, while the second class of images form the intermediate annulus, in between the inner core and the outermost annulus, as shown in Figure 61. The centre of both the annuli and the core is the same, at the point $(1,1)$. The parameters for 3NN are $cnum = 2 = \text{number of pixels}$, $bnum = 12, anum = 2 = \text{number of classes}$. Again, we ensured that the hidden layers of 4NN have the same number of nodes as in the hidden layer of 3NN. Therefore, the parameters for 4NN are $dnum = 2 = \text{number of pixels}$, $cnum = 6 = bnum$, and $anum = 2 = \text{number of classes}$. Similar to the initialization of weights and biases seen in section 3.3.2 and 3.3.3, we shall only randomize the weights connecting the input layer and the first input layer for both 3NN and 4NN, while keeping the rest the same.

Screenshots

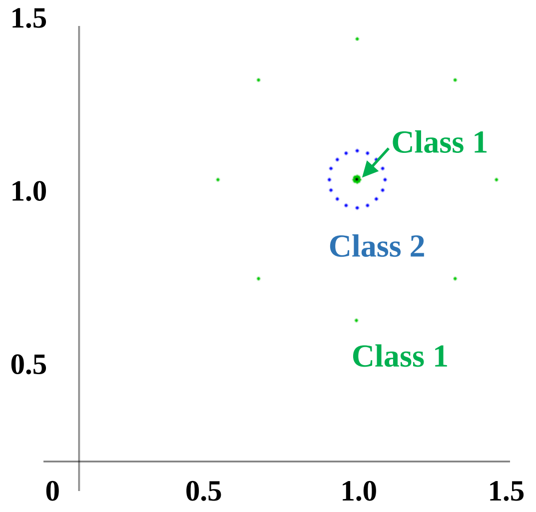


Figure 61: Visualization of double annuli and a core dataset

```
v is:
   1.821   -0.731   -5.951   -3.030   -0.417   -9.990   -1.444   -0.234   -0.711  -151.929   12.887   -0.215
  -0.252   -0.822    0.224    3.969    0.158    3.485   -3.765    3.579   -0.436   149.108  -10.723   -1.995
bbias is:
  -4.415   -0.861    2.746   -2.009   -0.243    3.750    2.685   -4.541    0.260    52.629   -0.577    1.370

w is:
   6.003   -3.435
  -5.307    7.774
  17.392  -18.958
  14.845  -16.840
 -11.878   12.103
  -4.298    3.134
  18.202  -16.878
   3.170   -2.134
  -8.958   11.055
  -3.820    4.134
   6.323   -7.575
  -4.661    4.531
abias is:
   0.259   -0.270

ss is: 0.03125
average cost is:       0.30887818513745668447256775 83
highest cost is:       0.7475397209996752 with predictions:   1.00000000   0.38307249   0.60575586
highest cost is:       0.7475397209996752 corresponding to:   1.00000000   0.99000000   1.00000000
number of pixels is: 2
number of iterations is: 1500
```

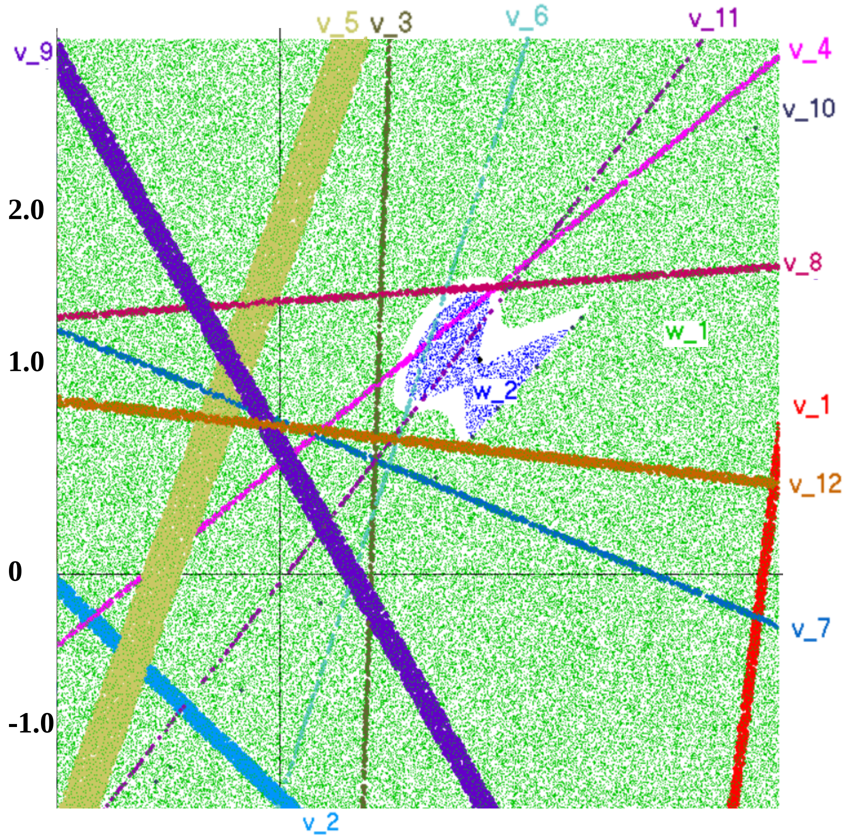


Figure 62: Weights and biases, and their visualization, for 3NN on double annuli and one core dataset in Figure 61, where the black dot is the centre of the annuli and the core

```
u is:
   2.425    2.427    2.019   -2.225   -6.039    0.155
  -1.984   -3.227   -7.857   -4.324    0.771   -1.051
cbias is:
  -1.125    1.625    3.125    2.875    6.375    2.000

v is:
  33.000  -73.125  -19.125   72.500  -16.625   26.375
  -1.375   32.375  -23.875   63.250   12.875   -2.250
 -71.750    2.125   17.125  -51.250   51.500 -114.875
 -30.125   73.250   15.375  -78.125  121.375   82.875
  -0.875  106.750    0.000    6.250   19.750    4.375
  -4.250  -38.500   12.375  -52.250  -23.125   -2.625
bbias is:
  -0.250   -2.375    6.875  -25.500   -5.125   -1.500
w is:
  16.000  -16.250
 -69.250   68.000
 147.250 -149.125
  47.250  -47.000
  52.750  -52.250
  33.875  -33.250
abias is:
 -48.125   48.500

ss is: 0.125
numimages is: 32
average cost is:       0.0000000000002105413711931802
highest cost is:       0.0000000000021147 with predictions:   2.00000000   0.00000093   0.99999888
highest cost is:       0.0000000000021147 corresponding to:   2.00000000   1.07071068   1.07071068
number of pixels is: 2
number of iterations is: 1500
```

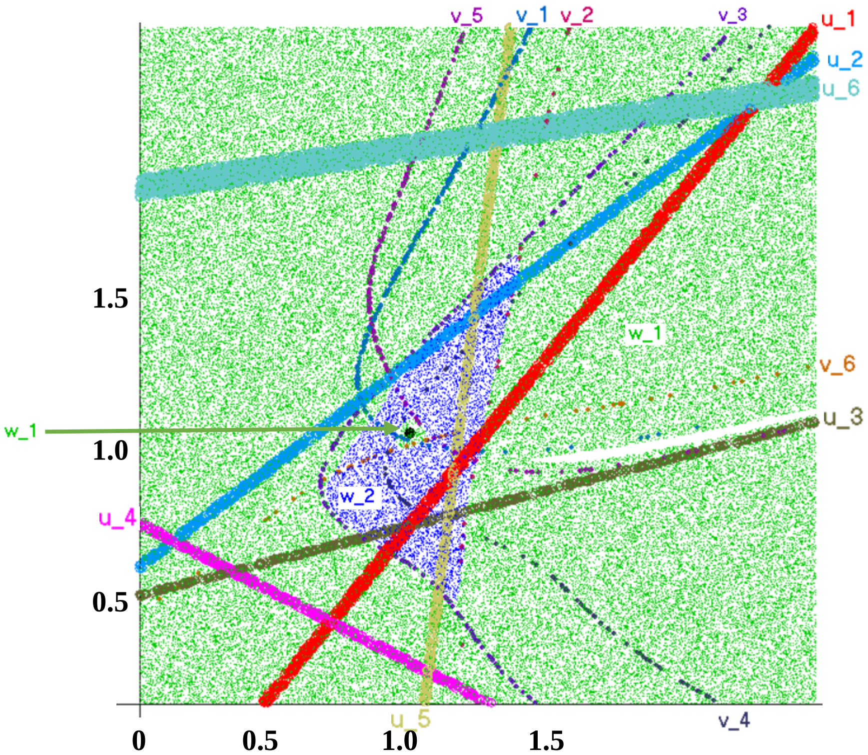


Figure 63: Weights and biases, and their visualization, for 4NN on double annuli and one core dataset in Figure 61, where the black dot is the centre of the annuli and the core

Comparisons and Discussions

For Figure 62, we visualize the area $[-1,2.5] \times [-1,2.5] \subset \mathbb{R}^2$, otherwise v_2 cannot be visualized. Referring to both Figure 62 and 63, the black-coloured dot, in the visualization of weights, is used to denote the centre of the annuli and the core, with coordinates (1,1).

We can see in Figure 62 that the average cost obtained by 3NN is way larger than the average cost obtained by 4NN (in Figure 63). This suggests that the additional hidden layer in 4NN makes a huge difference here; specifically, the use of non-linear "cuts" helps tremendously in separating class 1 and class 2. For example, the v_i's in 4NN (Figure 63) manage to "lasso" the class 1 core for it to be recognized, while if we look at Figure 62 3NN fails to recognize the class 1 images in the core. Despite having the same number of nodes, 3NN could not perform as well as 4NN, and may require much more nodes in the hidden layer. However, again, 4NN seems more inexplicable than 3NN, especially the nonlinear linear boundaries v_i's.

Conclusion

In this project, we set out to understand if neural networks could be explained through the lens of principal component analysis and Fisher linear discriminant. For simple examples as seen in Chapter 2, NNs and PCA seem to agree with one another. However, as the examples become more complex, even FLD does not seem to work as compared to NNs. One reason for this seems to be that PCA and FLD require an additional step to classify the images, after projecting them onto the eigenvectors. Furthermore, NNs can learn from and classify non-linearly separable dataset of images, unlike FLD and PCA. However, in practice, PCA is usually used to lower the dimensionality of the dataset before the images are being used by the NNs. This means that they usually work together to reduce computational time and cost.

Besides investigating the relationship between NNs, PCA, and FLD we also looked at increasingly more complex NNs, as we try to understand how they differ from one another. Specifically, we saw how 3NN overcame some issues that 2NN when it comes to recognizing an example. We also looked at how 3NN did not work as well as 4NN in some examples, given a fixed number of hidden nodes, despite 3NNs being universal approximators. The caveat, for the theorem, is the wideness of the layer must be sufficiently large, which may not be as efficient as using a 4NN.

It is worth noting that there exist non-linear generalizations of the principal component analysis (Scholkopf et al., 1999) and Fisher discriminant (Roth & Steinhage, 1999). These variants may have bridged the differences between NNs and PCA and FLD. Unfortunately, we were unable to investigate this relationship within the scope of this project. Therefore, future studies could be done on this front.

One limitation in our project is that we use hypothetical images, generated (albeit randomly) by our programmes. Furthermore, the images we have are of very low dimensions, containing only either two or three pixels. These make our images very different from realistic images, which are usually much noisier and of higher dimension (for example 256×256 images). Therefore, our findings may not generalize to these images. Moreover, due to the scope of the project, we could only consider one type of neural network, namely the multi-layer perceptron. We were also limited by both the scope and compute time to only consider this type of neural network up to four layers of nodes, including both the input and output layers. This means that our architecture is rather primitive and shallow, compared to other state-of-the-art models out there. Compare them to the popular language model, ChatGPT, developed by OpenAI and can generate text in a human-like manner, which uses a transformer-based neural network and billions of parameters (Brown et al., 2020; Radford et al., 2019); and another model called ResNet-50 developed by Microsoft (He et al., 2015) that uses a convolutional neural network and has a depth of as deep as 152 layers. Therefore, our findings deal with cases on a much smaller and simpler scale and may not generalize to more recent developments. This gap could be filled by future studies, as the prevalence of such complex, powerful systems render it all the more important for them to be understood by humans.

References

Belhumeur, P. N., Hespanha, J. P., & Kriegman, D. J. (1997). Eigenfaces vs. Fisherfaces: Recognition using class specific linear projection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(7), 711-720. https://doi.org/10.1109/34.598228.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. arXiv. https://doi.org/10.48550/arxiv.2005.14165

Fisher, R. A. (1936). The Use of Multiple Measurements in Taxonomic Problems. Annals of Eugenics, 7, 179-188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36, 193–202. https://doi.org/10.1007/BF00344251.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv. https://doi.org/10.48550/ARXIV.1512.03385

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5), 359-366. https://doi.org/10.1016/0893-6080(89)90020-8.

Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. (2017). The Expressive Power of Neural Networks: A View from the Width. arXiv. https://doi.org/10.48550/ARXIV.1709.02540.

McCulloch, W.S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5, 115–133. https://doi.org/10.1007/BF02478259.

Pearson, K. (1901). LIII. On lines and planes of closest fit to systems of points in space. Philosophical Magazine Series 1, 2, 559-572. https://doi.org/10.1080/14786440109462720

Radford, A., Wu, J., Child, R., Laun, D., Amodei, D., & Ilya., S. (2020). Language Models are Unsupervised Multitask Learners. OpenAI. Retrieved from https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Roth, V., & Steinhage, V. (1999, December). Nonlinear discriminant analysis using kernel functions. In Advances in Neural Information Processing Systems (pp. 873-879).

Sanderson, G. (2017). Gradient descent, how neural network learn [Video from “Neural Networks” series]. YouTube. https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&ab_channel=3Blue1Brown

Scholkopf, B., Smola, A. J., & Müller, K.-R. (1999). Kernel principal component analysis. In Advances in Kernel Methods – Support Vector Learning (pp. 327-352). MIT Press.

Singh, J. (2019, March). Constructing a Sigmoid Perceptron in Python. Medium. https://medium.com/hackernoon/codeblog-sigmoid-perceptron-e52c878e0e03.

Turk, M., & Pentland, A. (1991). Eigenfaces for recognition. Journal of Cognitive Neuroscience, 3(1), 71-86. https://doi.org/10.1162/jocn.1991.3.1.71.

Appendix A: Codes for Programmes in Chapter 2

Generating One Ellipses Dataset and Computing Covariance Matrix (Section 2.1.3)

```
        open #1, "PCA_CovMatrix.txt", "w"

dim u(1000,1000), v(100,100), w(100,100), clusters(100,100), C_u(100,100),
C_v(100,100), C_w(100,100), C(100,100), avg(100), u_avg(100), v_avg(100),
w_avg(100), d1(100), d2(100), n(100)

        num_images = 1000
        dim_images = 2
        noise = 0.2
        a$ = "######.##########"

        gosub initialize_clusters
        gosub output_c_input_NN
        gosub calculate_average
        gosub calculate_covMs
        gosub output_CovMatrix

        print "Covariance matrix is: "

        for i = 1 to dim_images
        for k = 1 to dim_images
        print C(i,k);
        next k
        print
        next i

        end
rem     ************************************************
        label initialize_clusters

        t = -pi/4
        for i = 1 to num_images
        u(1,i) = ran(4)-2
        x = sqrt(1-u(1,i)^2/4)
        u(2,i) = (ran(2*x)-x)
        u(1,i) = cos(t)*u(1,i) - sin(t)*u(2,i) : u(2,i) = -sin(t)*u(1,i) +
cos(t)*u(2,i)
        next i

        return
rem     ************************************************
        label output_c_input_NN

        open #2, "c_input.txt", "w"
        for j = 1 to num_images
        for i = 1 to dim_images
        print #2 u(i,j) using a$;
        next i
        print #2
        next j
        close #2
        return

rem     ************************************************
        label calculate_average

        for i = 1 to dim_images
        avg(i) = 0
        for j = 1 to num_images
        avg(i) = avg(i) + (u(i,j))
        next j
        avg(i) = avg(i)/(num_images)
        next i

        return
rem     ************************************************
        label calculate_covMs
```

Visualizing One Ellipse Dataset (Section 2.1.3)

```
        dim freq(1000), b(10), a(10), sigb(10), siga(10)
        dim x(100)
        open #1, "c_input.txt", "r"
        num_images = 1000
        gosub setup_graphics
        gosub conduct_trials

        end
rem     ******************************
        label setup_graphics

        open window 800,800
        window origin "lb"
        hscaling = 100: vscaling = 100
        hoffset = 400: voffset = 400
        line 0,V(0) to 800, V(0)
        line H(0), 0 to H(0), 800

        return
rem     ******************************
        sub H(xx)
        return xx*hscaling + hoffset
        end sub
rem     ********************************
        sub V(yy)
        return yy*vscaling + voffset
        end sub
rem     ******************************
        label conduct_trials

        for k = 1 to num_images

        for i = 1 to 2
        input #1 x(i)
        next i

        colour 100,100,100
        fill circle H(x(1)), V(x(2)), 2

        print "current k, ", k

        next k

        colour 250,0,0
        line H(0),V(0) to H(0.740653),V(0.671908)
        line H(0),V(0) to H(-0.671908), V(0.740635)

        return
```

The codes to visualize other datasets are similar and will be omitted in this Appendix.

```
        for i = 1 to dim_images
        for k = 1 to dim_images
        for j = 1 to num_images
        C(i,k) = C(i,k) + (u(i,j) - avg(i))*(u(k,j) - avg(k))
        next j
        next k
        next i

        return
rem     ***********************************************
        label output_CovMatrix

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C(i,j) using a$;
        next j
        print #1
        next i
        print
        close #1

        return
rem     ***********************************************
        sub N(t)

        k = ran(2*t)-t
        return k

        end sub

rem     ***********************************************
```

PCA Algorithm to Obtain Eigenvectors for One Ellipse Dataset (section 2.1.3)

```
        open #1, "PCA_CovMatrix.txt", "r"
        ss = 1e-3
        dim_images = 2
        a$ = "#######.#########"
        num_eigenvectors = 0
        dim u(10,100), v(10,100), w(10,100), f(10), e(10), improv_e(10),
eigenvec(10,10), avg(10), e1(10), e2(10), C(100,100), F(10)

        gosub input_C
        gosub initial_eigenvector
100
        improvflag = 0
        while improvflag <= 30
        gosub change_e
        wend
        num_eigenvectors = num_eigenvectors + 1
        gosub output_current
        gosub store_eigenvector
        gosub output_eigenvector
        if num_eigenvectors < dim_images then
        gosub choose_next_eigenvector
        pause(4)
        goto 100
        endif
        end
rem     *************************************
        label input_C

        for i = 1 to dim_images
        for j = 1 to dim_images
        input #1 C(i,j)
        print C(i,j)
        next j
        next i

        return
```

Generating Two-blob Dataset and Calculating Covariance Matrix for 2NN and PCA (Section 2.2.3 & 2.3.1)

```
        open #1, "PCA_CovMatrix.txt", "w"

        dim u(100,100), v(100,100), w(100,100), clusters(100,100),
C_u(100,100), C_v(100,100), C_w(100,100), C(100,100), avg(100),
u_avg(100), v_avg(100), w_avg(100), d1(100), d2(100), n(100)

        num_images = 100
        dim_images = 2
        noise = 0.1
        a$ = "######.##########"
        gosub initialize_clusters
        gosub output_c_input_NN
        gosub calculate_covMs
        gosub output_CovMatrix
        print "Covariance matrix is: "
        for i = 1 to dim_images
        for k = 1 to dim_images
        print C(i,k);
        next k
        print
        next i

        end


rem     *************************************
        label initialize_clusters

        for r = 1 to dim_images
        u_avg(r) = 0 : v_avg(r) = 0 : w_avg(r) = 0
        next r
        for i = 1 to num_images
        u(0,i) = 1 : u(1,i) = 0 + N(noise) : u(2,i) = 1 + N(noise)
        v(0,i) = 2 : v(1,i) = 1 + N(noise) : v(2,i) = 0 + N(noise)
        next i
```

```
rem     **************************************
        label initial_eigenvector

        for i = 1 to dim_images
        e(i) = ran(1)
        next i
        gosub normalize_e

        return
rem     *************************************
        label calculate_F

        F = 0
        for i = 1 to dim_images
        for j = 1 to dim_images
        F = F + e(i)*C(i,j)*e(j)
        next j
        next i

        return
rem     **************************************
        label change_e

        gosub calculate_F
        F1 = F
        for i = 1 to dim_images
        improv_e(i) = e(i)
        next i
        gosub delta_e
        gosub orthog_e
        gosub normalize_e
        gosub calculate_F
        F2 = F
        F12 = F1 - F2
        if F12 > 0 then
        improvflag = improvflag + 1
        for i = 1 to dim_images
        e(i) = improv_e(i)
        next i
        else
        improvflag = 0
        endif

        return
rem     *************************************
        label output_current

        gosub calculate_F
        print "The eigenvector obtained is: "
        for i = 1 to dim_images
        print e(i) using a$
        next i
        print "F is: "
        print F using a$
        print

        return
rem     **************************************
        label store_eigenvector

        for k = 1 to dim_images
        eigenvec(k,num_eigenvectors) = e(k)
        next k

        return
rem     **************************************
        label choose_next_eigenvector

        gosub initial_eigenvector
        gosub orthog_e
        gosub normalize_e
        print "Next eigenvector: "
        for i = 1 to dim_images
        print e(i)
```

```
        return
rem     **************************************
        label output_c_input_NN

        open #2, "NN_c_input.txt", "w"
        for k = 1 to 2
        for j = 1 to num_images
        for i = 0 to dim_images
        if k = 1 then
        print #2 u(i,j) using a$;
        elseif k = 2 then
        print #2 v(i,j) using a$;
        endif
        next i
        print #2
        next j
        next k
        close #2

        return
rem     **************************************
        label calculate_average

        for i = 1 to dim_images
        avg(i) = 0
        for j = 1 to num_images
        avg(i) = avg(i) + (u(i,j) + v(i,j))
        next j
        avg(i) = avg(i)/(2*num_images)
        next i

        return

rem     **************************************
        label calculate_covMs

        for i = 1 to dim_images
        for k = 1 to dim_images
        for j = 1 to num_images
        C_u(i,k) = C_u(i,k) + (u(i,j) - avg(i))*(u(k,j) - avg(k))
        C_v(i,k) = C_v(i,k) + (v(i,j) - avg(i))*(v(k,j) - avg(k))

        next j
        next k
        next i
        for i = 1 to dim_images
        for k = 1 to dim_images
        C(i,k) = C_u(i,k) + C_v(i,k)
        next k
        next i

        return
rem     **************************************
        label output_CovMatrix

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C(i,j) using a$;
        next j
        print #1
        next i
        print
        close #1

        return
rem     **************************************
        label calculate_average_class

        for i = 1 to dim_images
        u_avg(i) = 0 : v_avg(i) = 0 : w_avg(i) = 0
        for j = 1 to num_images
        u_avg(i) = u_avg(i) + u(i,j)
        v_avg(i) = v_avg(i) + v(i,j)
        w_avg(i) = w_avg(i) + w(i,j)
```

```
next i

return
rem **************************************
label output_eigenvector

print "Current eigenvectors are: "
for i = 1 to dim_images
for k = 1 to dim_images
print eigenvec(i,k);
next k
print
next i
print

return
rem **************************************
label normalize_e

E = 0
for i = 1 to dim_images
E = E + e(i)^2
next i
E = sqrt(E)
for i = 1 to dim_images
e(i) = e(i)/E
next i

return
rem **************************************
label orthog_e

for k = 1 to num_eigenvectors
dot_product = 0
for i = 1 to dim_images
dot_product = dot_product + (e(i) * eigenvec(i,k))
next i
for j = 1 to dim_images
e(j) = e(j) - dot_product*eigenvec(j,k)
next j
next k

return
rem **************************************
label delta_e

for i = 1 to dim_images
e(i) = e(i) + ran(2*ss) - ss
next i

return
rem **************************************
```

The codes for PCA algorithm to extract the eigenvectors are similar for other datasets, differing only in the dataset input into the programme. Therefore, this algorithm will be further omitted in this Appendix.

```
next j
u_avg(i) = u_avg(i)/num_images
v_avg(i) = v_avg(i)/num_images
w_avg(i) = w_avg(i)/num_images
next i
print "u_avg(1) is: ", u_avg(1)

return
rem **************************************
sub N(t)

k = ran(2*t)-t
return k

end sub
rem **************************************
```

2NN for Two-blob Dataset (Section 2.3.1)

```
dim a(100),b(100),c(100)
dim w(100,100),v(100,100)
dim abias(100), bbias(100)
dim p(100)
bnum = 3
anum = 3
numimages = 300
ransize = 0.1
500
```

Generating Isosceles Triangle Dataset and Computing Covariance Matrix for 2NN and PCA (Section 2.3.1)

```
open #1, "PCA_CovMatrix.txt", "w"
dim u(100,100), v(100,100), w(100,100), clusters(100,100), C_u(100,100), C_v(100,100), C_w(100,100), C(100,100), avg(100), u_avg(100), v_avg(100), w_avg(100), d1(100), d2(100), n(100)

num_images = 100
dim_images = 3
noise = 0.2
a$ = "######.##########"
```

```
	gosub initial_weights
	gosub initial_bias

	gosub output_current
	avg = 0
	ss = 1
	iterations = 0
300
	ss = ss/2
100
	improvflag = 0
	gosub change_abias
	gosub change_w
	gosub output_current
	gosub output_weights
	gosub average_cost
	print "ss is: ", ss
	print "average cost is: ", avg using
"#######.##############################"
	iterations = iterations + 1
	print "number of iterations is: ", iterations
	if avg < 1e-10 then
	goto 400
	endif
	if improvflag = 0 then
	print "Reducing ss"
	pause(2)
	goto 300
	else
	goto 100
	endif
400
	if iterations < 30 then
	if improvflag = 0 then
	print "Reducing ss"
	goto 300
	else
	goto 100
	endif
	else
	goto 400
	endif

	end
rem ****************************
	label initial_weights


	for i = 1 to anum
	for j = 1 to bnum
	if j = i then
	w(i,j) = 1
	else
	w(i,j) = 0
	endif
	next j
	next i

	return
rem ****************************
	label initial_bias

	for i = 1 to anum
	abias(i) = 0
	next i

	return
rem ****************************
	label input_c

	for i = 0 to cnum
	input #1 c(i)
	next i

	gosub initialize_clusters
	gosub output_c_input_NN
	gosub calculate_average
	gosub calculate_covMs
	gosub output_CovMatrix
	print "Covariance matrix is: "
	for i = 1 to dim_images
	for k = 1 to dim_images
	print C(i,k);
	next k
	print
	next i
	end


rem	***************************************
	label initialize_clusters

	for r = 1 to dim_images
	u_avg(r) = 0 : v_avg(r) = 0 : w_avg(r) = 0
	next r
	for i = 1 to num_images
	u(0,i) = 1 : u(1,i) = 1 + N(noise) : u(2,i) = 0 + N(noise) : u(3,i) = 0 +
N(noise)
	v(0,i) = 2 : v(1,i) = 0 + N(noise) : v(2,i) = 1 + N(noise) : v(3,i) = 0 +
N(noise)
	w(0,i) = 3 : w(1,i) = 0 + N(noise) : w(2,i) = 0 + N(noise) : w(3,i) = 3 +
N(noise)
	next i

	return

rem	***************************************
	label output_c_input_NN

	open #2, "NN_c_input.txt", "w"
	for k = 1 to 3
	for j = 1 to num_images
	for i = 0 to dim_images
	if k = 1 then
	print #2 u(i,j) using a$;
	elseif k = 2 then
	print #2 v(i,j) using a$;
	else
	print #2 w(i,j) using a$;
	endif
	next i
	print #2
	next j
	next k
	close #2

	return

rem	***************************************
	label calculate_average

	for i = 1 to dim_images
	avg(i) = 0
	for j = 1 to num_images
	avg(i) = avg(i) + (u(i,j) + v(i,j) + w(i,j))
	next j
	avg(i) = avg(i)/(3*num_images)
	next i

	return

rem	***************************************
	label calculate_covMs

	for i = 1 to dim_images
	for k = 1 to dim_images
	for j = 1 to num_images
```

```
	return
rem ****************************
	label compute_NN

	for i = 1 to anum
	a(i) = 0
	for j = 1 to bnum
	a(i) = a(i) + w(i,j)*b(j)
	next j
	next i
	for i = 1 to anum
	a(i) = a(i) + abias(i)
	next i
	for i = 1 to anum
	a(i) = sigmoid(a(i))
	next i

	return
rem ****************************
	label calculate_cost

	gosub select_p
	cost = 0
	for i = 1 to anum
	cost = cost + (a(i) - p(i))^2
	next i

	return
rem ****************************
	label average_cost

	open #1, "c_input.txt", "r"
	totalcost = 0
	for image = 1 to numimages
	gosub input_c
	gosub compute_NN
	gosub calculate_cost
	totalcost = totalcost + cost
	next image
	avg = totalcost/numimages
	close #1
	return

rem ****************************
	label output_current

	a$ = "######.######"
	open #1, "NN_c_input.txt", "r"
	for image = 1 to numimages
	gosub input_c
	if (numimages - image) <= 5 then
	gosub compute_NN
	gosub calculate_cost
	print c(0) using a$
	for i = 1 to anum
	print a(i) using a$;
	next i
	print cost using a$;
	print:print
	endif
	next image
	close #1

	return
rem ****************************
	label change_w

	for k = 1 to anum
	for r = 1 to bnum
	gosub average_cost
	cost1 = avg
	w(k,r) = w(k,r) + ss
	gosub average_cost
	cost2 = avg
```

```
	C_u(i,k) = C_u(i,k) + (u(i,j) - avg(i))*(u(k,j) - avg(k))
	C_v(i,k) = C_v(i,k) + (v(i,j) - avg(i))*(v(k,j) - avg(k))
	C_w(i,k) = C_w(i,k) + (w(i,j) - avg(i))*(w(k,j) - avg(k))
	next j
	next k
	next i
	for i = 1 to dim_images
	for k = 1 to dim_images
	C(i,k) = C_u(i,k) + C_v(i,k) + C_w(i,k)
	next k
	next i

	return

rem	***************************************
	label output_CovMatrix

	for i = 1 to dim_images
	for j = 1 to dim_images
	print #1 C(i,j) using a$;
	next j
	print #1
	next i
	print
	close #1

	return
rem	***************************************

	label calculate_average_class

	for i = 1 to dim_images
	u_avg(i) = 0 : v_avg(i) = 0 : w_avg(i) = 0
	for j = 1 to num_images
	u_avg(i) = u_avg(i) + u(i,j)
	v_avg(i) = v_avg(i) + v(i,j)
	w_avg(i) = w_avg(i) + w(i,j)
	next j
	u_avg(i) = u_avg(i)/num_images
	v_avg(i) = v_avg(i)/num_images
	w_avg(i) = w_avg(i)/num_images
	next i
	print "u_avg(1) is: ", u_avg(1)

	return

rem	***************************************
	sub N(t)

	k = ran(2*t)-t
	return k

	end sub

rem	*******************************************
```

```
        w(k,r) = w(k,r) - 2*ss
        gosub average_cost
        cost3 = avg
        w(k,r) = w(k,r) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        w(k,r) = w(k,r) + ss
        improvflag = 1
        elseif(cost13>0) then
        w(k,r) = w(k,r) - ss
        improvflag = 1
        endif
        next r
        next k

        return
rem ****************************
        label change_abias

        for k = 1 to anum
        gosub average_cost
        cost1 = avg

        abias(k) = abias(k) + ss
        gosub average_cost
        cost2 = avg
        abias(k) = abias(k) - 2*ss
        gosub average_cost
        cost3 = avg
        abias(k) = abias(k) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        abias(k) = abias(k) + ss
        improvflag = 1
        elseif(cost13>0) then
        abias(k) = abias(k) - ss
        improvflag = 1
        endif
        next k

        return
rem ****************************
        label select_p

        for i = 1 to anum
        if i = c(0) then
        p(i) = 1
        else
        p(i) = 0
        endif
        next i

        return
rem ****************************
        label output_weights

        print
        print "v is: "
        for j = 1 to cnum
        for i = 1 to bnum
        print v(i,j) using "###.###";
        next i
        print
        next j
        print "bbias is: "
        for i = 1 to bnum
        print bbias(i) using "###.###";
        next i
        print
        print "w is: "
        for j = 1 to bnum
        for i = 1 to anum
        print w(i,j) using "###.###";
        next i
```

```
        print
        next j
        print "abias is: "
        for i = 1 to anum
        print abias(i) using "###.###";
        next i
        print
        print

        return
rem     **************************
        sub sigmoid(xx)

        yy = 1/(1+exp(-xx))
        return yy

        end sub
rem ****************************
```

The codes for running 2NN are similar for different dataset and will be omitted further in this Appendix.

Generating OXUT Dataset and Covariance Matrix for 2NN and PCA (Section 2.3.2)

```
        open #1, "PCA_CovMatrix.txt", "w"

        dim o(100,100), x(100,100), u(100,100), t(100,100),
clusters(100,100), C_o(100,100), C_x(100,100), C_u(100,100), C_t(100,100),
C(100,100), avg(100), u_avg(100), v_avg(100), w_avg(100), d1(100), d2(100),
n(100)

        num_images = 50
        dim_images = 9
        noise = 0.2
        a$ = "######.##########"
        gosub initialize_clusters
        gosub output_c_input_NN
        gosub calculate_average
        print "finished average"
        gosub calculate_covMs
        gosub output_CovMatrix
        print "Covariance matrix is: "
        for i = 1 to dim_images
        for k = 1 to dim_images
        print C(i,k);
        next k
        print
        next i

        end
rem     ****************************************
        label initialize_clusters

rem     generating O
        for k = 0 to dim_images
        for i = 1 to num_images
        if k = 0 then
        o(k,i) = 1
        elseif k = 5 then
        o(k,i) = N(noise)
        else
        o(k,i) = 1 + N(noise)
        endif
        next i
        next k
rem     generating X
        for k = 0 to dim_images
        for i = 1 to num_images
        if k = 0 then
        x(k,i) = 2
        elseif mod(k,2)=0 then
        x(k,i) = N(noise)
```

Visualizing Weights of 2NN as Images for OXUT Dataset (Section 2.32)

```
        dim w(100,100), w_absmax(100)
        bnum = 9
        anum = 4
        gosub input_w
        gosub max_w
        gosub image_w

        end
rem     *********************************
        label input_w

        open #1, "weights_2.txt", "r"
        for j = 1 to bnum
        for i = 1 to anum
        input #1 w(i,j)
        next i
        next j
        close #1

        return
rem     *********************************
        label setup_graphics

        open window 800,800
        window origin "lb"
        hscaling = 50: vscaling = 50
        hoffset = 100  : voffset = 100
rem     line 0,V(0) to 800, V(0)
rem     line H(0), 0 to H(0), 800

        return
rem     *********************************
        sub H(xx)
        return xx*hscaling + hoffset
        end sub

rem     ********************************
        sub V(yy)
        return yy*vscaling + voffset
        end sub
rem     *****************************
        label colour_w

        w = w_absmax(i)
        if w(i,j) = 0 then
        color 255,255,255
        elseif w(i,j) > 0 then
        color c(w(i,j), w),c(w(i,j),w),255
```

```
        else
        x(k,i) = 1 + N(noise)
        endif
        next i
        next k
rem     generating U
        for k = 0 to dim_images
        for i = 1 to num_images
        if k = 0 then
        u(k,i) = 3
        elseif k = 2 or k = 5 then
        u(k,i) = N(noise)
        else
        u(k,i) = 1 + N(noise)
        endif
        next i
        next k
rem     generating T
        for k = 0 to dim_images
        for i = 1 to num_images
        if k = 0 then
        t(k,i) = 4
        elseif k = 4 or k = 6 or k = 7 or k = 9 then
        t(k,i) = N(noise)
        else
        t(k,i) = 1 + N(noise)
        endif
        next i
        next k

        return
rem     ****************************************
        label output_c_input_NN

        open #2, "NN_c_input.txt", "w"
        for k = 1 to 4
        for j = 1 to num_images
        for i = 0 to dim_images
        if k = 1 then
        print #2 o(i,j) using a$;
        elseif k = 2 then
        print #2 x(i,j) using a$;
        elseif k = 3 then
        print #2 u(i,j) using a$;
        else
        print #2 t(i,j) using a$;
        endif
        next i
        print #2
        next j
        next k
        close #2

        return

rem     ****************************************

        label calculate_average

        for i = 1 to dim_images
        avg(i) = 0
        for j = 1 to num_images
        avg(i) = avg(i) + (o(i,j) + x(i,j) + u(i,j) + t(i,j))
        next j
        avg(i) = avg(i)/(4*num_images)
        next i

        return
rem     ****************************************
        label calculate_covMs

        for i = 1 to dim_images
        for k = 1 to dim_images
        for j = 1 to num_images
```

```
        elseif w(i,j) < 0 then
        color 255,c(w(i,j), w),c(w(i,j),w)
        endif

        return
rem     ********************************

        label max_w

        for k = 1 to anum
        w_absmax(k) = w(k,1)
        for r = 1 to bnum
        w_absmax(k) = max(abs(w_absmax(k)), abs(w(k,r)))
        next r
        next k

        return
rem     ********************************

        label image_w

        gosub setup_graphics
        for i = 1 to anum
        for j = 1 to 9

        if i = 3 then
        t = 4-floor((j-1)/3)
        s = mod(j-1,3) + 1
        colour 0,0,0
        rectangle H(s)-1,V(t)-1 to H(s+1)+1,V(t+1)+1
        gosub colour_w
        fill rectangle H(s),V(t) to H(s+1),V(t+1)

        elseif i = 4 then
        t = 4-floor((j-1)/3)
        s = mod(j-1,3) + 5
        colour 0,0,0
        rectangle H(s)-1,V(t)-1 to H(s+1)+1,V(t+1)+1
        gosub colour_w
        fill rectangle H(s),V(t) to H(s+1),V(t+1)

        elseif i = 1 then
        t = 8-floor((j-1)/3)
        s = mod(j-1,3) + 1
        colour 0,0,0
        rectangle H(s)-1,V(t)-1 to H(s+1)+1,V(t+1)+1
        gosub colour_w
        fill rectangle H(s),V(t) to H(s+1),V(t+1)

        elseif i = 2 then
        t = 8-floor((j-1)/3)
        s = mod(j-1,3) + 5
        colour 0,0,0
        rectangle H(s)-1,V(t)-1 to H(s+1)+1,V(t+1)+1
        gosub colour_w
        fill rectangle H(s),V(t) to H(s+1),V(t+1)
        endif
        next j
        next i

        return
rem     ********************************
        sub c(xx, w)

        return -abs(xx)*(255/w)+255

        end sub
```

```
        C_o(i,k) = C_o(i,k) + (o(i,j) - avg(i))*(o(k,j) - avg(k))
        C_x(i,k) = C_x(i,k) + (x(i,j) - avg(i))*(x(k,j) - avg(k))
        C_u(i,k) = C_u(i,k) + (u(i,j) - avg(i))*(u(k,j) - avg(k))
        C_t(i,k) = C_t(i,k) + (t(i,j) - avg(i))*(t(k,j) - avg(k))
        next j
        next k
        next i
        for i = 1 to dim_images
        for k = 1 to dim_images
        C(i,k) = C_o(i,k) + C_x(i,k) + C_u(i,k) + C_t(i,k)
        next k
        next i

        return
rem     ****************************************
        label output_CovMatrix

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C(i,j) using a$;
        next j
        print #1
        next i
        print
        close #1

        return
rem     ****************************************
        sub N(t)

        rand = ran(2*t)-t
        return rand

        end sub
rem     ****************************************
```

Generating Two Ellipses Dataset and Covariance Matrix for 2NN and PCA (Section 2.3.3)

```
        open #1, "PCA_CovMatrix.txt", "w"

        dim u(1000,1000), v(1000,1000), w(100,100), clusters(100,100),
C_u(100,100), C_v(100,100), C_w(100,100), C(100,100), avg(100),
u_avg(100), v_avg(100), w_avg(100), d1(100), d2(100), n(100)

        num_images = 1000
        dim_images = 2
        noise = 0.2
        a$ = "######.##########"
        gosub initialize_clusters
        gosub output_c_input_NN
        gosub calculate_average
        gosub calculate_covMs
        gosub output_CovMatrix
        print "Covariance matrix is: "
        for i = 1 to dim_images
        for k = 1 to dim_images
        print C(i,k);
        next k
        print
        next i

        end
rem     ********************************************
        label generate_clusters

        t = -pi/4
        for i = 1 to num_images
        u(0,i) = 1
        u(1,i) = ran(4)-2
```

Generating Three-blob Dataset 2NN and PCA (Section 2.3.3)

```
        open #1, "c_input.txt", "w"

        dim c(100), count(1000)
        noise = 0.2
        num_images = 100
        anum = 3
        a$ = "##.######"
        for k = 1 to 3
        for j = 1 to num_images
        c(0) = k
        if k = 1 then
        c(1) = 0 + N(noise): c(2) = 1 + N(noise)
        elseif k = 2 then
        c(1) = 0.5 + N(noise) : c(2) = 0.5+N(noise)
        else
        c(1) = 1 + N(noise) : c(2) = 0 + N(noise)
        endif
        for i = 0 to 2
        print #1 c(i) using a$;
        next i
        print #1
        next j
        next k
        close #1

        end


rem     ********************************************
        sub N(t)
```

```
x = sqrt(1-u(1,i)^2/4)
u(2,i) = (ran(2*x)-x)
u(1,i) = cos(t)*u(1,i) - sin(t)*u(2,i)
u(2,i) = -sin(t)*u(1,i) + cos(t)*u(2,i)
next i

for i = 1 to num_images
v(0,i) = 2
v(1,i) = ran(4)-2
x = sqrt(1-v(1,i)^2/4)
v(2,i) = (ran(2*x)-x)
v(1,i) = cos(t)*v(1,i) - sin(t)*v(2,i)
v(2,i) = -sin(t)*v(1,i) + cos(t)*v(2,i)
v(1,i) = v(1,i)+2
next i

return
rem     ******************************************
label output_c_input_NN

open #2, "c_input.txt", "w"
for j = 1 to num_images
for i = 0 to dim_images
print #2 u(i,j) using a$;
next i
print #2
next j
for j = 1 to num_images
for i = 0 to dim_images
print #2 v(i,j) using a$;
next i
print #2
next j
close #2

return
rem     ******************************************
label calculate_average

for i = 1 to dim_images
avg(i) = 0
for j = 1 to num_images
avg(i) = avg(i) + (u(i,j) + v(i,j))
next j
avg(i) = avg(i)/(2*num_images)
next i

return
rem     ******************************************

label calculate_covMs
for i = 1 to dim_images
for k = 1 to dim_images
for j = 1 to num_images
C_u(i,k) = C_u(i,k) + (u(i,j) - avg(i))*(u(k,j) - avg(k))
C_v(i,k) = C_v(i,k) + (v(i,j) - avg(i))*(v(k,j) - avg(k))
next j
next k
next i
for i = 1 to dim_images
for k = 1 to dim_images
C(i,k) = C_u(i,k) + C_v(i,k)
next k
next i

return

rem     ******************************************
label output_CovMatrix

for i = 1 to dim_images
for j = 1 to dim_images
print #1 C(i,j) using a$;
next j
l = ran(2*t)-t
return l

end sub

rem     ******************************************
```

```
        print #1
        next i
        print
        close #1

        return
rem     ********************************************
        sub N(t)

        k = ran(2*t)-t
        return k

        end sub
rem     ********************************************
```

Appendix B: Codes for Programmes in Chapter 3

Generating Two Ellipses Dataset and Covariance Matrices for FLD (Section 3.1.3)

```
	open #1, "FLD_CovMatrix.txt", "w"
	open #2, "c_input.txt", "w"

	dim u(1000,1000), v(1000,1000), w(100,100), clusters(100,100),
C_u(100,100), C_v(100,100), C_w(100,100), C_between(100,100),
C_within(100,100),avg_u(100), v_avg(100), w_avg(100)
	dim avg(100), avg_u(100), avg_v(100)

	num_images = 1000
	dim_images = 2
	noise = 0.1
	a$ = "######.##########"
	gosub initialize_clusters
	gosub calculate_covMs
	gosub output_C
	for i = 1 to dim_images
	for j = 1 to dim_images
	print #1 C_within(i,j) using a$;
	next j
	print #1
	next i
	for i = 1 to dim_images
	for j = 1 to dim_images
	print #1 C_between(i,j) using a$;
	next j
	print #1
	next i
	close #1
	for j = 1 to num_images
	for i = 0 to dim_images
	print #2 u(i,j) using a$;
	next i
	print #2
	next j
	for j = 1 to num_images
	for i = 0 to dim_images
	print #2 v(i,j) using a$;
	next i
	print #2
	next j
	close #2

	end


rem	**********************************************
	label initialize_clusters

	for r = 1 to dim_images
	avg_u(r) = 0 : v_avg(r) = 0
	next r

	t = -pi/4
	for i = 1 to num_images
	u(0,i) = 1
	u(1,i) = ran(4)-2
	x = sqrt(1-u(1,i)^2/4)
	u(2,i) = (ran(2*x)-x)
	u(1,i) = cos(t)*u(1,i) - sin(t)*u(2,i)
	u(2,i) = -sin(t)*u(1,i) + cos(t)*u(2,i)
	for k = 1 to dim_images
	avg_u(k) = avg_u(k) + u(k,i)
	next k

	next i

	for i = 1 to num_images
	v(0,i) = 2
```

FLD Algorithm to Obtain Generalized Eigenvectors for Two Ellipses Dataset (Section 3.1.3)

```
	open #1, "FLD_CovMatrix.txt", "r"

	ss = 1e-3
	num_clusters = 3
	dim_images = 2
	a$ = "#######.#########"
	num_eigenvectors = 0

	dim u(10,100), v(10,100), w(10,100), f(10), e(10), improv_e(10),
eigenvec(10,10), avg(10), e1(10), e2(10), C_within(100,100),
C_between(100,100), check_between(10), check_within(10), lambda(10)


	gosub input_C
	gosub initial_eigenvector
100
	improvflag = 0
	while improvflag <= 25
	gosub change_e
	wend
	num_eigenvectors = num_eigenvectors + 1
	gosub output_current
	gosub store_eigenvector
	gosub output_eigenvector
	if num_eigenvectors < dim_images-1 then
	gosub choose_next_eigenvector
	pause(4)
	goto 100
	endif

	end
rem	***************************************
	label input_C

	for i = 1 to dim_images
	for j = 1 to dim_images
	input #1 C_within(i,j)
	print C_within(i,j);
	next j
	print
	next i
	for i = 1 to dim_images
	for j = 1 to dim_images
	input #1 C_between(i,j)
	print C_between(i,j);
	next j
	print
	next i

	return
rem	***************************************
	label initial_eigenvector

	for i = 1 to dim_images
	e(i) = ran(1)
	next i
	gosub normalize_e
	print "Initializing eigenvector: "
	print e(1)
	print e(2)

	return
rem	***************************************
	label calculate_G

	G = 0
	G_between = 0
	G_within = 0
```

```
            v(1,i) = ran(4)-2
            x = sqrt(1-v(1,i)^2/4)
            v(2,i) = (ran(2*x)-x)
            v(1,i) = cos(t)*v(1,i) - sin(t)*v(2,i) : v(2,i) = -sin(t)*v(1,i) +
cos(t)*v(2,i)
            v(1,i) = v(1,i)+2: v(2,i) = v(2,i) + 0.5
            for k = 1 to dim_images
            avg_v(k) = avg_v(k) + v(k,i)
            next k
            next i
            for k = 1 to dim_images
            avg(k) = avg_u(k) + avg_v(k)
            avg_u(k) = avg_u(k)/num_images
            avg_v(k) = avg_v(k)/num_images
            avg(k) = avg(k)/(2*num_images)
            next k

            return
rem         ********************************************
            label calculate_covMs

            for i = 1 to dim_images
            for k = 1 to dim_images
            for j = 1 to num_images
            C_u(i,k) = C_u(i,k) + (u(i,j) - avg_u(i))*(u(k,j) - avg_u(k))
            C_v(i,k) = C_v(i,k) + (v(i,j) - avg_v(i))*(v(k,j) - avg_v(k))
            next j
            next k
            next i
            for i = 1 to dim_images
            for k = 1 to dim_images
            C_within(i,k) = C_u(i,k) + C_v(i,k)
            C_between(i,k) = num_images*((avg_u(i)-avg(i))*(avg_u(k)-
avg(k)) + (v_avg(i)-avg(i))*(v_avg(k)-avg(k)))
            next k
            next i

            return
rem         ********************************************
            label output_C

            print "Within cluster covariance matrix is: "
            for i = 1 to dim_images
            for k = 1 to dim_images
            print C_within(i,k);
            next k
            print
            next i
            print
            print "Average of class 1: "
            for i = 1 to dim_images
            print avg_u(i)
            next i
            print
            print "Average of class 2: "
            for i = 1 to dim_images
            print avg_v(i)
            next i
            print

            return
rem         ********************************************
            sub N(t)

            k = ran(2*t)-t
            return k

            end sub

rem         ********************************************

            for i = 1 to dim_images
            for j = 1 to dim_images

            G_between = G_between + e(i)*C_between(i,j)*e(j)
            G_within = G_within + e(i)*C_within(i,j)*e(j)

            next j
            next i

            G = (G_between)/(G_within)

            return
rem         **************************************

            label change_e

            gosub calculate_G
            G1=G
            for i = 1 to dim_images
            improv_e(i) = e(i)
            next i
            gosub delta_e
            gosub orthog_e
            gosub normalize_e
            gosub calculate_G
            G2= G
            G12 = G1 - G2
            if G12 > 0 then
            improvflag = improvflag + 1
            for i = 1 to dim_images
            e(i) = improv_e(i)
            next i
            else
            improvflag = 0
            endif

            return
rem         **************************************
            label output_current

            gosub calculate_G
            print "The eigenvector is: "
            print e(1) using a$
            print e(2) using a$
            print "G is: "
            print G using a$
            print

            return
rem         **************************************
            label store_eigenvector

            for k = 1 to dim_images
            eigenvec(k,num_eigenvectors) = e(k)
            next k

            return

rem         **************************************

            label choose_next_eigenvector

            gosub initial_eigenvector
            gosub orthog_e
            gosub normalize_e
            print "Next eigenvector: "
            for i = 1 to dim_images
            print e(i)
            next i

            return
rem         **************************************
```

```
        label output_eigenvector

        print "The eigenvectors are: "
        for i = 1 to dim_images
        for k = 1 to dim_images -1
        print eigenvec(i,k);
        next k
        print
        next i
        print

        return
rem     **************************************

        label normalize_e

        E = 0
        for i = 1 to dim_images
        E = E + e(i)^2
        next i
        E = sqrt(E)
        for i = 1 to dim_images
        e(i) = e(i)/E
        next i

        return

rem     **************************************
        label orthog_e

        for k = 1 to num_eigenvectors
        dot_product = 0
        for i = 1 to dim_images
        dot_product = dot_product + (e(i) * eigenvec(i,k))
        next i
        for j = 1 to dim_images
        e(j) = e(j) - dot_product*eigenvec(j,k)
        next j
        next k

        return
rem     ***************************************
        label delta_e

        for i = 1 to dim_images
        e(i) = e(i) + ran(2*ss) - ss
        next i

        return
rem     ***************************************
```

The codes for the FLD algorithm to obtain generalized eigenvectors are similar for different examples, differing only in the dataset used. Therefore, it will be omitted further in this Appendix.

3NN for Three-blob Dataset (Section 3.2.3)

```
    dim a(100),b(100),c(100)
    dim w(100,100),v(100,100)
    dim abias(100), bbias(100)
    dim p(100)
    dim vv(100,100), ww(100,100)
    dim bbbias(100), aabias(100)
    cnum = 2
    bnum = 3
    anum = 3
    numimages = 300
    ransize = 0.1
500
```

Weights Visualization for 3NN on Three-blob Dataset (Section 3.2.3)

```
        dim freq(1000), d(100), c(100), b(100), a(100), v(100,100),
w(100,100), u(100,100), cbias(100), bbias(100), abias(100)

        cnum = 2
        bnum = 3
        anum = 3
        num_trials = 5000
        w_threshold = 0.4
        u_threshold = 0.01
        v_threshold = 0.01
        gosub setup_graphics
        gosub import_weights_biases
        gosub check_weights_biases
```

```
	gosub initial_weights
	gosub initial_bias
	gosub output_current
	avg = 0
	ss = 1
	iterations = 0
300
	ss = ss/2
100
	improvflag = 0
	gosub change_abias
	gosub change_w
	gosub change_bbia
	gosub change_v
	gosub output_current
	gosub output_weights
	gosub average_cost
	print "ss is: ", ss
	print "average cost is: ", avg using
"#######.############################"
	iterations = iterations + 1
	print "number of iterations is: ", iterations
	if avg < 1e-10 then
	goto 400
	endif
	if improvflag = 0 then
	print "Reducing ss"
	pause(2)
	goto 300
	else
	goto 100
	endif
400
	gosub export_weights_biases

	end
rem ****************************
	label initial_weights

	for i = 1 to bnum
	for j = 1 to cnum
	if j = i then
	v(i,j) = 1
	else
	v(i,j) = 0
	endif
	next j
	next i

	for i = 1 to anum
	for j = 1 to bnum
	if j = i then
	w(i,j) = 1
	else
	w(i,j) = 0
	endif
	next j
	next i

	return
rem ****************************
	label initial_bias

	for i = 1 to anum
	abias(i) = 0
	next i

	for i = 1 to bnum
	bbias(i) = 0
	next i

	return
rem ****************************
	label input_c
```

```
	colour 0,100,200
	text 100,780,"v_1","lc","swiss18"
	colour 200,50,50
	text 100,760,"v_2","lc","swiss18"
	colour 100,0,200
	text 100,740,"v_3","lc","swiss18"
	colour 100,200,0

	colour 0,200,0
	text 150,780,"w_1","lc","swiss18"
	colour 0,0,200
	text 150,760,"w_2","lc","swiss18"
	colour 100,100,100
	text 150,740,"w_3","lc","swiss18"

	for k = 1 to 100
	gosub conduct_trials
	input a$
	if a$ = "end" then
	colour 0,200,0
	fill circle H(0), V(1), 10
	colour 0,0,200
	fill circle H(0.5), V(0.5), 10
	colour 100,100,100
	fill circle H(1),V(0), 10
	endif
	next k

	end
rem	******************************
	label setup_graphics
	open window 800,800
	window origin "lb"
	hscaling = 400: vscaling = 400
	hoffset = 50: voffset = 50
	line 0,V(0) to 800, V(0)
	line H(0), 0 to H(0), 800

	return
rem	*****************************
	sub H(xx)
	return xx*hscaling + hoffset
	end sub
rem	******************************
	sub V(yy)
	return yy*vscaling + voffset
	end sub
rem	*****************************
	label import_weights_biases

	open #2, "weight_biases.txt", "r"
	for j = 1 to cnum
	for i = 1 to bnum
	input #2 v(i,j)
	next i
	next j
	for i = 1 to bnum
	input #2 bbias(i)
	next i
	for j = 1 to bnum
	for i = 1 to anum
	input #2 w(i,j)
	next i
	next j
	for i = 1 to anum
	input #2 abias(i)
	next i

	return
rem	***************************
	label check_weights_biases
```

```
        for i = 0 to cnum
        input #1 c(i)
        next i

        return
rem ****************************
        label compute_NN

        for i = 1 to bnum
        b(i) = 0
        for j = 1 to cnum
        b(i) = b(i) + v(i,j)*c(j)
        next j
        next i
        for i = 1 to bnum
        b(i) = b(i) + bbias(i)
        next i
        for i = 1 to bnum
        b(i) = sigmoid(b(i))
        next i

        for i = 1 to anum
        a(i) = 0
        for j = 1 to bnum
        a(i) = a(i) + w(i,j)*b(j)
        next j
        next i
        for i = 1 to anum
        a(i) = a(i) + abias(i)
        next i
        for i = 1 to anum
        a(i) = sigmoid(a(i))
        next i

        return
rem ****************************
        label average_cost

        open #1, "c_input.txt", "r"
        totalcost = 0
        for image = 1 to numimages
        gosub input_c
        gosub compute_NN
        gosub calculate_cost
        totalcost = totalcost + cost
        next image

        avg = totalcost/numimages
        close #1

        return
rem ****************************
        label output_current

        a$ = "######.######"
        open #1, "c_input.txt", "r"
        for image = 1 to numimages
        gosub input_c
        if (numimages - image) <= 5 then
        gosub compute_NN
        gosub calculate_cost
        print c(0) using a$
        for i = 1 to anum
        print a(i) using a$;
        next i
        print cost using a$;
        print:print
        endif
        next image
        close #1

        return
rem ****************************
```

```
        print
        print "v is: "
        for j = 1 to cnum
        for i = 1 to bnum
        print v(i,j) using "###.###";
        next i
        print
        next j
        print "bbias is: "
        for i = 1 to bnum
        print bbias(i) using "###.###";
        next i
        print
        print "w is: "
        for j = 1 to bnum
        for i = 1 to anum
        print w(i,j) using "###.###";
        next i
        print
        next j
        print "abias is: "
        for i = 1 to anum
        print abias(i) using "###.###";
        next i
        print
        print

        return
rem     ***************************
        label conduct_trials

        for trial = 1 to num_trials
        c(1) = ran(1.2) : c(2) = ran(1.2)
        gosub compute_NN
        gosub colour_v
        gosub colour_w
        next trial

        return
rem     ***************************
        label compute_NN

        for i = 1 to bnum
        b(i) = 0
        for j = 1 to cnum
        b(i) = b(i) + v(i,j)*c(j)
        next j
        next i
        for i = 1 to bnum
        b(i) = b(i) + bbias(i)
        next i
        for i = 1 to bnum
        b(i) = sigmoid(b(i))
        next i

        for i = 1 to anum
        a(i) = 0
        for j = 1 to bnum
        a(i) = a(i) + w(i,j)*b(j)
        next j
        next i
        for i = 1 to anum
        a(i) = a(i) + abias(i)
        next i
        for i = 1 to anum
        a(i) = sigmoid(a(i))
        next i

        return
rem     ***************************
        label colour_v

        colouring = 0.5
        if abs(b(1)-colouring) < v_threshold then
```

```
        label change_w

        for k = 1 to anum
        for r = 1 to bnum
        gosub average_cost
        cost1 = avg
        w(k,r) = w(k,r) + ss
        gosub average_cost
        cost2 = avg
        w(k,r) = w(k,r) - 2*ss
        gosub average_cost
        cost3 = avg
        w(k,r) = w(k,r) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        w(k,r) = w(k,r) + ss
        improvflag = 1
        elseif(cost13>0) then
        w(k,r) = w(k,r) - ss
        improvflag = 1
        endif
        next r
        next k

        return
rem ****************************
        label change_v

        for k = 1 to bnum
        for r = 1 to cnum
        gosub average_cost
        cost1 = avg
        v(k,r) = v(k,r) + ss
        gosub average_cost
        cost2 = avg
        v(k,r) = v(k,r) - 2*ss
        gosub average_cost
        cost3 = avg
        v(k,r) = v(k,r) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        v(k,r) = v(k,r) + ss
        improvflag = 1
        elseif(cost13>0) then
        v(k,r) = v(k,r) - ss
        improvflag = 1
        endif
        next r
        next k

        return
rem ****************************
        label change_abias

        for k = 1 to anum
        gosub average_cost
        cost1 = avg
        abias(k) = abias(k) + ss
        gosub average_cost
        cost2 = avg
        abias(k) = abias(k) - 2*ss
        gosub average_cost
        cost3 = avg
        abias(k) = abias(k) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        abias(k) = abias(k) + ss
        improvflag = 1
        elseif(cost13>0) then
        abias(k) = abias(k) - ss
        improvflag = 1
        endif
        next k
```

```
        colour 0,100,200
        fill circle H(c(1)), V(c(2)), 2
        endif

        if abs(b(2)-colouring) < v_threshold then
        colour 200,50,50
        fill circle H(c(1)), V(c(2)), 2
        endif

        if abs(b(3)-colouring) < v_threshold then
        colour 100,0,200
        fill circle H(c(1)), V(c(2)), 2
        endif

        return
rem     ***************************
        label colour_w

        if abs(a(1) - 1) < w_threshold then
        colour 0,200,0
        dot H(c(1)), V(c(2))
        endif

        if abs(a(2) - 1) < w_threshold  then
        colour 0,0,200
        dot H(c(1)), V(c(2))
        endif

        if abs(a(3) - 1) < w_threshold  then
        colour 100,100,100
        dot H(c(1)), V(c(2))
        endif

        return
rem     *******************************
        sub sigmoid(xx)

        yy = 1/(1+exp(-xx))
        return yy
        end sub

rem     *******************************
```

The codes to visualize the weights are similar for other examples, differing only in the weights used (obtained from the NN). Therefore, we omit these codes further in the Appendix. We will however include the codes for the visualization of weights for 4NN in the subsequent part.

```
            return
rem ****************************
            label change_bbias

            for k = 1 to bnum
            gosub average_cost
            cost1 = avg
            bbias(k) = bbias(k) + ss
            gosub average_cost
            cost2 = avg
            bbias(k) = bbias(k) - 2*ss
            gosub average_cost
            cost3 = avg
            bbias(k) = bbias(k) + ss
            cost12 = cost1-cost2 : cost13 = cost1-cost3
            if(cost12>0) then
            bbias(k) = bbias(k) + ss
            improvflag = 1
            elseif(cost13>0) then
            bbias(k) = bbias(k) - ss
            improvflag = 1
            endif
            next k

            return

rem ****************************
            label select_p

            for i = 1 to anum
            if i = c(0) then
            p(i) = 1
            else
            p(i) = 0
            endif
            next i

            return
rem ****************************
            label calculate_cost

            gosub select_p
            cost = 0
            for i = 1 to anum
            cost = cost + (a(i) - p(i))^2
            next i

            return
rem ****************************
            label output_weights

            print
            print "v is: "
            for j = 1 to cnum
            for i = 1 to bnum
            print v(i,j) using "###.###";
            next i
            print
            next j
            print "bbias is: "
            for i = 1 to bnum
            print bbias(i) using "###.###";
            next i
            print
            print "w is: "
            for j = 1 to bnum
            for i = 1 to anum
            print w(i,j) using "###.###";
            next i
            print
            next j
            print "abias is: "
            for i = 1 to anum
            print abias(i) using "###.###";
```

```
        next i
        print
        print

        return
rem     *************************
        label export_weights_biases

        open #2, "weight_biases.txt", "w"
        for j = 1 to cnum
        for i = 1 to bnum
        print #2 v(i,j) using "###.###";
        next i
        print #2
        next j
        for i = 1 to bnum
        print #2 bbias(i) using "###.###";
        next i
        print #2
        for j = 1 to bnum
        for i = 1 to anum
        print #2 w(i,j) using "###.###";
        next i
        print #2
        next j
        for i = 1 to anum
        print #2 abias(i) using "###.###";
        next i
        print #2 : print #2
        print #2 "*****************************************"
        close #2

        return
rem     *************************
        sub sigmoid(xx)

        yy = 1/(1+exp(-xx))
        return yy

        end sub
rem ***************************
```

The code for 3NN is similar for the different examples, differing only in the dataset used. Therefore, the codes for this algorithm will be omitted from this Appendix after this.

Generating Crescent and Core Dataset and Covariance Matrices for 3NN and FLD (Section 3.3.1)

```
        open #1, "FLD_CovMatrix.txt", "w"
        open #2, "c_input.txt", "w"

        dim u(1000,1000), v(1000,1000), w(100,100), clusters(100,100),
C_u(100,100), C_v(100,100), C_w(100,100), C_between(100,100),
C_within(100,100),avg_u(100), v_avg(100), w_avg(100)
        dim avg(100), avg_u(100), avg_v(100)

        num_images = 10
        dim_images = 2
        noise = 0.1
        a$ = "######.##########"
        gosub initialize_clusters
        gosub calculate_covMs
        gosub output_C
        gosub export_C_c

        end
rem     *************************************
        label initialize_clusters
```

Generating Annulus with a Core Dataset and Covariance Matrices for 3NN and FLD (Section 3.3.2)

```
        open #1, "FLD_CovMatrix.txt", "w"
        open #2, "c_input.txt", "w"

        dim u(1000,1000), v(1000,1000), w(100,100), clusters(100,100),
C_u(100,100), C_v(100,100), C_w(100,100), C_between(100,100),
C_within(100,100),avg_u(100), v_avg(100), w_avg(100)
        dim avg(100), avg_u(100), avg_v(100)

        num_images = 10
        dim_images = 2
        noise = 0.1
        a$ = "######.##########"
        gosub initialize_clusters
        gosub calculate_covMs
        gosub output_C
        gosub export_C_c

        end
rem     *************************************
        label initialize_clusters
```

```
a$ = "##.######"
for r = 1 to dim_images
avg_u(r) = 0 : v_avg(r) = 0
next r
for i = 1 to num_images
theta = 1.3*pi*(i/num_images) + pi/4
xx = cos(theta)
yy = sin(theta)
u(0,i) = 1
u(1,i) = 0.5*xx + 1
u(2,i) = 0.5*yy + 1
for k = 1 to dim_images
avg_u(k) = avg_u(k) + u(k,i)
next k
next i
for i = 1 to num_images
theta = 2*pi*(i/num_images)
xx = cos(theta)
yy = sin(theta)


v(0,i) = 2
v(1,i) = 0.05*xx + 1
v(2,i) = 0.05*yy + 1
for k = 1 to dim_images
avg_v(k) = avg_v(k) + v(k,i
next k
next i
for k = 1 to dim_images
avg(k) = avg_u(k) + avg_v(k)
avg_u(k) = avg_u(k)/num_images
avg_v(k) = avg_v(k)/num_images
avg(k) = avg(k)/(2*num_images)
next k

return
rem ******************************************
label calculate_covMs

for i = 1 to dim_images
for k = 1 to dim_images
for j = 1 to num_images
C_u(i,k) = C_u(i,k) + (u(i,j) - avg_u(i))*(u(k,j) - avg_u(k))
C_v(i,k) = C_v(i,k) + (v(i,j) - avg_v(i))*(v(k,j) - avg_v(k))
next j
next k
next i
for i = 1 to dim_images
for k = 1 to dim_images
C_within(i,k) = C_u(i,k) + C_v(i,k)
C_between(i,k) = num_images*((avg_u(i)-avg(i))*(avg_u(k)-avg(k)) + (v_avg(i)-avg(i))*(v_avg(k)-avg(k)))
next k
next i

return
rem ******************************************
label output_C

print "Within cluster covariance matrix is: "
for i = 1 to dim_images
for k = 1 to dim_images
print C_within(i,k);
next k
print
next i
print
print "Average of class 1: "
for i = 1 to dim_images
print avg_u(i)
next i
print
print "Average of class 2: "
for i = 1 to dim_images
```

```
a$ = "##.######"
for r = 1 to dim_images
avg_u(r) = 0 : v_avg(r) = 0
next r
for i = 1 to num_images
theta = 1.3*pi*(i/num_images) + pi/4
xx = cos(theta)
yy = sin(theta)
u(0,i) = 1
u(1,i) = 0.5*xx + 1
u(2,i) = 0.5*yy + 1
for k = 1 to dim_images
avg_u(k) = avg_u(k) + u(k,i)
next k
next i
for i = 1 to num_images
theta = 2*pi*(i/num_images)
xx = cos(theta)
yy = sin(theta)


v(0,i) = 2
v(1,i) = 0.05*xx + 1
v(2,i) = 0.05*yy + 1
for k = 1 to dim_images
avg_v(k) = avg_v(k) + v(k,i)
next k
next i
for k = 1 to dim_images
avg(k) = avg_u(k) + avg_v(k)
avg_u(k) = avg_u(k)/num_images
avg_v(k) = avg_v(k)/num_images
avg(k) = avg(k)/(2*num_images)
next k

return
rem **************************************
label calculate_covMs

for i = 1 to dim_images
for k = 1 to dim_images
for j = 1 to num_images
C_u(i,k) = C_u(i,k) + (u(i,j) - avg_u(i))*(u(k,j) - avg_u(k))
C_v(i,k) = C_v(i,k) + (v(i,j) - avg_v(i))*(v(k,j) - avg_v(k))
next j
next k
next i
for i = 1 to dim_images
for k = 1 to dim_images
C_within(i,k) = C_u(i,k) + C_v(i,k)
C_between(i,k) = num_images*((avg_u(i)-avg(i))*(avg_u(k)-avg(k)) + (v_avg(i)-avg(i))*(v_avg(k)-avg(k)))
next k
next i

return
rem **************************************
label output_C

print "Within cluster covariance matrix is: "
for i = 1 to dim_images
for k = 1 to dim_images
print C_within(i,k);
next k
print
next i
print
print "Average of class 1: "
for i = 1 to dim_images
print avg_u(i)
next i
print
print "Average of class 2: "
for i = 1 to dim_images
```

```
        print avg_v(i)
        next i
        print

        return
rem     ******************************************
        label export_C_c

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C_within(i,j) using a$;
        next j
        print #1
        next i
        for i = 1 to dim_images
        for j = 1 to dim_images

        print #1 C_between(i,j) using a$;
        next j
        print #1
        next i
        close #1
        for j = 1 to num_images
        for i = 0 to dim_images
        print #2 u(i,j) using a$;
        next i
        print #2
        next j
        for j = 1 to num_images
        for i = 0 to dim_images
        print #2 v(i,j) using a$;
        next i
        print #2
        next j
        close #2

        return
rem     ******************************************
        sub N(t)

        k = ran(2*t)-t
        return k

        end sub
rem     ******************************************
```

```
        print avg_v(i)
        next i
        print

        return
rem     **************************************
        label export_C_c

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C_within(i,j) using a$;
        next j
        print #1
        next i

        for i = 1 to dim_images
        for j = 1 to dim_images
        print #1 C_between(i,j) using a$;
        next j
        print #1
        next i
        close #1
        for j = 1 to num_images
        for i = 0 to dim_images
        print #2 u(i,j) using a$;
        next i
        print #2
        next j
        for j = 1 to num_images
        for i = 0 to dim_images
        print #2 v(i,j) using a$;
        next i
        print #2
        next j
        close #2

        return
rem     **************************************
        sub N(t)

        k = ran(2*t)-t
        return k

        end sub
rem     **************************************
```

Generating Annulus with an Inner and An Outer Core Dataset (Section 3.3.3)

```
        open #1, "c_input.txt", "w"
        dim c(100), count(1000)

rem     each class has 16 images
        num_images = 16
        num_images1 = 8
        num_images2 = num_images - num_images1
        a$ = "####.########################"

rem     inner core
        for k = 1 to num_images1

        d = 2*pi*k/num_images1
        c(0) = 1

        c(1) = 1 + 0.1*cos(d) : c(2) = 1 + 0.1*sin(d)

        for i = 0 to 2
        print #1 c(i) using a$;
        next i
        print #1

        next k
```

4NN for Annulus with an Inner and An Outer Core Dataset (Section 3.3.3)

```
        dim a(100),b(100),c(100), d(100)
        dim w(100,100),v(100,100), u(100,100)
        dim abias(100), bbias(100), cbias(100)
        dim p(100)
        dim vv(100,100), ww(100,100)
        dim bbbias(100), aabias(100)

        open #2, "weight_biases4NN.txt", "w"
        dnum = 2
        cnum = 4
        bnum = 4
        anum = 2
        num_images_generate_c = 16
        numimages = num_images_generate_c*2
        ransize = 0.1
        costfactor = 1
        num_iterations = 1500
500
        gosub initial_weights
        gosub initial_bias
        gosub output_current
        avg = 0
```

```
rem     annulus

        for k = 1 to num_images
        d = 2*pi*k/num_images
        c(0) = 2
        c(1) = 1 + 0.4*cos(d) : c(2) = 1 + 0.4*sin(d)

        for i = 0 to 2
        print #1 c(i) using a$;
        next i
        print #1

        next k

rem     outer core
        for k = 1 to num_images2
        d = 2*pi*k/num_images2
        c(0) = 1
        c(1) = 1.7 + 0.1*cos(d) : c(2) = 1.7 + 0.1*sin(d)

        for i = 0 to 2
        print #1 c(i) using a$;
        next i
        print #1

        next k

        close #1

        end
```

```
        ss = 1/4
        for iteration = 1 to num_iterations
        improvflag = 0
        gosub change_abias
        gosub change_w
        gosub change_bbias
        gosub change_v
        gosub change_cbias
        gosub change_u
        gosub output_current
        gosub output_weights
        gosub average_cost
        gosub high_cost
        print "ss is: ", ss
        print "numimages is: ", numimages
        print "average cost is: ", avg using
"#######.############################"
        d(1) = highest_1 : d(2) = highest_2
        gosub compute_NN
        print "highest cost is: ", highest_cost using
"#######.################", " with predictions: ", highest_0 using
"###.########", a(1) using "###.########", a(2) using "###.########"
        print "highest cost is: ", highest_cost using
"#######.################", " corresponding to: ", highest_0 using
"###.########", highest_1 using "###.########", highest_2 using
"###.########"
        avg1 = avg
        print "number of pixels is: ", dnum
        iterations = iterations + 1
        print "number of iterations is: ", iteration
        gosub export_weights_biases
        if improvflag = 0 then
        print "Reducing ss"
        ss = ss/2
        pause(2)
        endif
        if iteration = 30 and avg > 0.33 then
        goto 500
        endif
        next iteration
        close #2

        end
rem ****************************
        label high_cost

        highest_cost = 0
        open #1, "c_input.txt", "r"
        for r = 1 to numimages
        gosub input_c
        gosub compute_NN
        gosub calculate_cost
        highest_cost = max(highest_cost, cost)
        if highest_cost = cost then
        highest_0 = d(0) : highest_1 = d(1) : highest_2 = d(2)
        endif
        next r
        close #1

        return
rem ****************************
        label initial_weights

        for i = 1 to cnum
        for j = 1 to dnum
        u(i,j) = ran(2) - 1
        next j
        next i

        for i = 1 to bnum
        for j = 1 to cnum
        if i = j then
        v(i,j) = 1
        else
```

```
v(i,j) = 0
endif
next j
next i

for i = 1 to anum
for j = 1 to bnum
if i = j then
w(i,j) = 1
else
w(i,j) = 0
endif
next j
next i

return
rem ****************************
label initial_bias

for i = 1 to cnum
cbias(i) = 0
next i

for i = 1 to bnum
bbias(i) = 0
next i

for i = 1 to anum
abias(i) = 0
next i

return
rem ****************************
label input_c

for i = 0 to dnum
input #1 d(i)
next i

return
rem ****************************
label compute_NN


for i = 1 to cnum
c(i) = 0
for j = 1 to dnum
c(i) = c(i) + u(i,j)*d(j)
next j
next i
for i = 1 to cnum
c(i) = c(i) + cbias(i)
next i
for i = 1 to cnum
c(i) = sigmoid(c(i))
next i

for i = 1 to bnum
b(i) = 0
for j = 1 to cnum
b(i) = b(i) + v(i,j)*c(j)
next j
next i
for i = 1 to bnum
b(i) = b(i) + bbias(i)
next i
for i = 1 to bnum
b(i) = sigmoid(b(i))
next i

for i = 1 to anum
a(i) = 0
for j = 1 to bnum
a(i) = a(i) + w(i,j)*b(j)
```

```
        next j
        next i
        for i = 1 to anum
        a(i) = a(i) + abias(i)
        next i
        for i = 1 to anum
        a(i) = sigmoid(a(i))
        next i

        return
rem ***************************

        label average_cost
        open #1, "c_input.txt", "r"
        totalcost = 0
        for image = 1 to numimages
        gosub input_c
        gosub compute_NN
        gosub calculate_cost
        totalcost = totalcost + cost
        next image
        avg = totalcost/numimages
        close #1

        return
rem ***************************
        label output_current

        a$ = "######.######"
        open #1, "c_input.txt", "r"
        for image = 1 to numimages
        gosub input_c
        if (numimages - image) <= 5 then
        gosub compute_NN
        gosub calculate_cost
        print d(0) using a$
        for i = 1 to anum
        print a(i) using a$;
        next i
        print cost using a$;
        print:print
        endif
        next image
        close #1

        return
rem ***************************
        label change_w

        for k = 1 to anum
        for r = 1 to bnum
        gosub average_cost
        cost1 = avg
        w(k,r) = w(k,r) + ss
        gosub average_cost
        cost2 = avg
        w(k,r) = w(k,r) - 2*ss
        gosub average_cost
        cost3 = avg
        w(k,r) = w(k,r) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        w(k,r) = w(k,r) + ss
        improvflag = 1
        elseif(cost13>0) then
        w(k,r) = w(k,r) - ss
        improvflag = 1
        endif
        next r
        next k

        return

rem **************************
```

```
label change_v

for k = 1 to bnum
for r = 1 to cnum
gosub average_cost
cost1 = avg
v(k,r) = v(k,r) + ss
gosub average_cost
cost2 = avg
v(k,r) = v(k,r) - 2*ss
gosub average_cost
cost3 = avg
v(k,r) = v(k,r) + ss
cost12 = cost1-cost2 : cost13 = cost1-cost3
if(cost12>0) then
v(k,r) = v(k,r) + ss
improvflag = 1
elseif(cost13>0) then
v(k,r) = v(k,r) - ss
improvflag = 1
endif
next r
next k

return

rem ****************************
label change_u

for k = 1 to cnum
for r = 1 to dnum
gosub average_cost
cost1 = avg
u(k,r) = u(k,r) + ss
gosub average_cost
cost2 = avg
u(k,r) = u(k,r) - 2*ss
gosub average_cost
cost3 = avg
u(k,r) = u(k,r) + ss
cost12 = cost1-cost2 : cost13 = cost1-cost3
if(cost12>0) then
u(k,r) = u(k,r) + ss
improvflag = 1
elseif(cost13>0) then
u(k,r) = u(k,r) - ss
improvflag = 1
endif
next r
next k

return

rem ****************************
label change_abias

for k = 1 to anum
gosub average_cost
cost1 = avg
abias(k) = abias(k) + ss
gosub average_cost
cost2 = avg
abias(k) = abias(k) - 2*ss
gosub average_cost
cost3 = avg
abias(k) = abias(k) + ss
cost12 = cost1-cost2 : cost13 = cost1-cost3
if(cost12>0) then
abias(k) = abias(k) + ss
improvflag = 1
elseif(cost13>0) then
abias(k) = abias(k) - ss
improvflag = 1
endif
```

```
        next k

        return

rem ***************************
        label change_bbias

        for k = 1 to bnum
        gosub average_cost
        cost1 = avg
        bbias(k) = bbias(k) + ss
        gosub average_cost
        cost2 = avg
        bbias(k) = bbias(k) - 2*ss
        gosub average_cost
        cost3 = avg
        bbias(k) = bbias(k) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        bbias(k) = bbias(k) + ss
        improvflag = 1
        elseif(cost13>0) then
        bbias(k) = bbias(k) - ss
        improvflag = 1
        endif
        next k

        return

rem ****************************
        label change_cbias

        for k = 1 to cnum
        gosub average_cost
        cost1 = avg
        cbias(k) = cbias(k) + ss
        gosub average_cost
        cost2 = avg
        cbias(k) = cbias(k) - 2*ss
        gosub average_cost
        cost3 = avg
        cbias(k) = cbias(k) + ss
        cost12 = cost1-cost2 : cost13 = cost1-cost3
        if(cost12>0) then
        cbias(k) = cbias(k) + ss
        improvflag = 1
        elseif(cost13>0) then
        cbias(k) = cbias(k) - ss
        improvflag = 1
        endif
        next k

        return

rem ****************************
        label select_p

        for i = 1 to anum
        if i = d(0) then
        p(i) = 1
        else
        p(i) = 0
        endif
        next i

        return
rem ***************************
        label calculate_cost

        gosub select_p
        cost = 0
        for i = 1 to anum
        cost = cost + (a(i) - p(i))^2
        next i
```

```
            return
rem ***************************
            label output_weights

            print
            print "u is: "
            for j = 1 to dnum
            for i = 1 to cnum
            print u(i,j) using "####.###";
            next i
            print
            next j
            print "cbias is: "
            for i = 1 to cnum
            print cbias(i) using "####.###";
            next i
            print
            print "v is: "
            for j = 1 to cnum
            for i = 1 to bnum
            print v(i,j) using "####.###";
            next i
            print
            next j
            print "bbias is: "
            for i = 1 to bnum
            print bbias(i) using "####.###";
            next i
            print
            print "w is: "
            for j = 1 to bnum
            for i = 1 to anum
            print w(i,j) using "####.###";
            next i
            print
            next j
            print "abias is: "
            for i = 1 to anum
            print abias(i) using "####.###";
            next i
            print
            print
            return
rem         *************************
            label export_weights_biases

            for j = 1 to dnum
            for i = 1 to cnum
            print #2 u(i,j) using "####.###";
            next i
            print #2
            next j
            for i = 1 to cnum
            print #2 cbias(i) using "####.###";
            next i
            print #2
            for j = 1 to cnum
            for i = 1 to bnum
            print #2 v(i,j) using "####.###";
            next i
            print #2
            next j
            for i = 1 to bnum
            print #2 bbias(i) using "####.###";
            next i
            print #2
            for j = 1 to bnum
            for i = 1 to anum
            print #2 w(i,j) using "####.###";
            next i
            print #2
            next j
```

```
	for i = 1 to anum
	print #2 abias(i) using "####.###";
	next i
	print #2
	gosub compute_NN
	gosub average_cost
	print #2 "network cost is: ", avg using "####.###"
	print #2 : print #2
"*******************************************"

	return
rem	*************************
	sub sigmoid(xx)

	yy = 1/(1+exp(-xx))
	return yy

	end sub
rem ***************************
```

The codes for 4NN on different examples are similar, differing only on the dataset used. Therefore, this will be excluded going forward in this Appendix.

Visualizing Weights for 4NN on Annulus with an Inner and An Outer Core Dataset (Section 3.3.3)

```
	dim freq(1000), d(100), c(100), b(100), a(100), v(100,100),
w(100,100), u(100,100), cbias(100), bbias(100), abias(100)

	dnum = 2
	cnum = 4
	bnum = 4
	anum = 2
	num_trials = 5000
	w_threshold = 0.4
	u_threshold = 0.01
	v_threshold = 0.01

	gosub setup_graphics
	gosub import_weights_biases
	gosub check_weights_biases
	fill circle H(1), V(1), 5
	colour 0,150,0
	fill circle H(1.7), V(1.7), 5

	colour 0,100,200
	text 50,780,"u_1","lc","swiss18"
	colour 200,50,50
	text 50,760,"u_2","lc","swiss18"
	colour 100,0,200
	text 50,740,"u_3","lc","swiss18"
	colour 100,150,0
	text 50,720,"u_4","lc","swiss18"

	colour 255,0,0
	text 100,780,"v_1","lc","swiss18"
	colour 255,0,255
	text 100,760,"v_2","lc","swiss18"
	colour 50,100,100
	text 100,740,"v_3","lc","swiss18"
	colour 50,50,50
	text 100,720,"v_4","lc","swiss18"

	colour 0,200,0
	text 200,780,"w_1","lc","swiss18"
	colour 0,0,255
	text 200,760,"w_2","lc","swiss18"

	for k = 1 to 100
	gosub conduct_trials
	input dummy
	next k
```

Generating Double Annuli with a Core Dataset for 3NN and 4NN (Section 3.3.3)

```
	open #1, "c_input.txt", "w"
	dim c(100), count(1000)

rem	each class has 16 images
	num_images = 16
	num_images1 = 8
	num_images2 = num_images - num_images1
	a$ = "####.########################"

rem	inner core
	for k = 1 to num_images1

	d = 2*pi*k/num_images1
	c(0) = 1

	c(1) = 1 + 0.01*cos(d) : c(2) = 1 + 0.01*sin(d)

	for i = 0 to 2
	print #1 c(i) using a$;
	next i
	print #1

	next k

rem	middle annulus

	for k = 1 to num_images
	d = 2*pi*k/num_images
	c(0) = 2
	c(1) = 1 + 0.1*cos(d) : c(2) = 1 + 0.1*sin(d)

	for i = 0 to 2
	print #1 c(i) using a$;
	next i
	print #1

	next k

rem	outer annulus
	for k = 1 to num_images2
	d = 2*pi*k/num_images2
	c(0) = 1
	c(1) = 1 + 0.5*cos(d) : c(2) = 1 + 0.5*sin(d)

	for i = 0 to 2
	print #1 c(i) using a$;
```

```
        end

rem     *******************************
        label setup_graphics
        open window 800,800
        window origin "lb"
        hscaling = 230: vscaling = 230
        hoffset = 50: voffset = 50
        line 0,V(0) to 800, V(0)
        line H(0), 0 to H(0), 800

        return

rem     ******************************
        sub H(xx)
        return xx*hscaling + hoffset
        end sub

rem     ********************************
        sub V(yy)
        return yy*vscaling + voffset
        end sub

rem     ******************************
        label import_weights_biases

        open #2, "weight_biases4.txt", "r"
        for j = 1 to dnum
        for i = 1 to cnum
        input #2 u(i,j)
        next i
        next j
        for i = 1 to cnum
        input #2 cbias(i)
        next i
        for j = 1 to cnum
        for i = 1 to bnum
        input #2 v(i,j)
        next i
        next j
        for i = 1 to bnum
        input #2 bbias(i)
        next i
        for j = 1 to bnum
        for i = 1 to anum
        input #2 w(i,j)
        next i
        next j

        for i = 1 to anum
        input #2 abias(i)
        next i

        return
rem     **************************
        label check_weights_biases

        print
        print "u is: "
        for j = 1 to dnum
        for i = 1 to cnum
        print u(i,j) using "###.###";
        next i
        print
        next j
        print "cbias is: "
        for i = 1 to cnum
        print cbias(i) using "###.###";
        next i
        print
        print "v is: "
        for j = 1 to cnum
        for i = 1 to bnum
```

```
next i
print #1

next k

close #1

end
```

```
        print v(i,j) using "###.###";
        next i
        print
        next j
        print "bbias is: "
        for i = 1 to bnum
        print bbias(i) using "###.###";
        next i
        print
        print "w is: "
        for j = 1 to bnum
        for i = 1 to anum
        print w(i,j) using "###.###";
        next i
        print
        next j
        print "abias is: "
        for i = 1 to anum
        print abias(i) using "###.###";
        next i
        print
        print

        return
rem     **************************
        label conduct_trials

        for trial = 1 to num_trials
        d(1) = ran(2.5) : d(2) = ran(2.5)
        gosub compute_NN
        gosub colour_u
        gosub colour_v
        gosub colour_w
        next trial

        return
rem     ***************************
        label compute_NN

        for i = 1 to cnum
        c(i) = 0
        for j = 1 to dnum
        c(i) = c(i) + u(i,j)*d(j)
        next j
        next i
        for i = 1 to cnum
        c(i) = c(i) + cbias(i)
        next i
        for i = 1 to cnum
        c(i) = sigmoid(c(i))
        next i

        for i = 1 to bnum
        b(i) = 0
        for j = 1 to cnum
        b(i) = b(i) + v(i,j)*c(j)
        next j
        next i
        for i = 1 to bnum
        b(i) = b(i) + bbias(i)
        next i
        for i = 1 to bnum
        b(i) = sigmoid(b(i))
        next i

        for i = 1 to anum
        a(i) = 0
        for j = 1 to bnum
        a(i) = a(i) + w(i,j)*b(j)
        next j
        next i
        for i = 1 to anum
        a(i) = a(i) + abias(i)
        next i
```

```
        for i = 1 to anum
        a(i) = sigmoid(a(i))
        next i

        return
rem     ***************************
        label colour_u

        if abs(c(1)-0.5) < u_threshold then
        colour 0,100,200
        circle H(d(1)), V(d(2)), 4
        endif

        if abs(c(2)-0.5) < u_threshold then
        colour 200,50,50
        circle H(d(1)), V(d(2)), 4
        endif

        if abs(c(3)-0.5) < u_threshold then
        colour 100,0,200
        circle H(d(1)), V(d(2)), 4
        endif

        if abs(c(4)-0.5) < u_threshold then
        colour 100,150,0
        circle H(d(1)), V(d(2)), 4
        endif

        if abs(c(5)-0.5) < u_threshold then
        colour 200,200,100
        circle H(d(1)), V(d(2)), 4
        endif

        if abs(c(6)-0.5) < u_threshold then
        colour 100,200,200
        circle H(d(1)), V(d(2)), 4
        endif

        return
rem     ***************************
        label colour_v
        colouring = 0.5

        if abs(b(1)-colouring) < v_threshold then
        colour 255,0,0
        fill circle H(d(1)), V(d(2)), 2
        endif

        if abs(b(2)-colouring) < v_threshold then
        colour 255,0,255
        fill circle H(d(1)), V(d(2)), 2
        endif

        if abs(b(3)-colouring) < v_threshold then
        colour 50,100,100
        fill circle H(d(1)), V(d(2)), 2
        endif

        if abs(b(4)-colouring) < v_threshold then
        colour 50,50,50
        fill circle H(d(1)), V(d(2)), 2
        endif

        if abs(b(5)-colouring) < v_threshold then
        colour 200,0,100
        fill circle H(d(1)), V(d(2)), 2
        endif

        if abs(b(6)-colouring) < v_threshold then
        colour 200,100,0
        fill circle H(d(1)), V(d(2)), 2
        endif

        return
```

```
rem     **************************
        label colour_w

        if abs(a(1) - 1) < w_threshold and abs(a(2)) < w_threshold then
        colour 0,200,0
        dot H(d(1)), V(d(2))
        endif

        if abs(a(2) - 1) < w_threshold and abs(a(1)) < w_threshold then
        colour 0,0,255
        dot H(d(1)), V(d(2))
        endif

        return
rem     ******************************
        sub sigmoid(xx)

        yy = 1/(1+exp(-xx))
        return yy
        end sub
rem     ******************************
```

The code for visualizing the weights for 4NN are similar, differing only on the weights being used (which depends on the dataset). Therefore, going forward, the codes for this will be omitted in this Appendix.