CHAPTER THREE

SUPERVISED CLASSIFICATION

3.1 OVERVIEW

Using machine learning methods to classify data sets is a recognised solution in many remote sensing applications. In this chapter several design considerations are introduced that should be heeded when implementing a supervised classifier. This is important, since less than 30% of new designs are correctly assessed [127]. In the previous chapter it was found that machine learning methods are more readily used in modern research because of the large volumes of data sets becoming readily available to the research community, and the great benefit of analysing these data sets in higher dimensional feature space. This chapter focuses on discussing strong, feasible approaches when a supervised classifier is used to solve real world problems.

3.2 CLASSIFICATION

Classification is the process of finding important similarities between objects and then grouping these objects into several subjective classes (categories).

Conceptual clustering is a modern process of classification by which conceptual descriptions are derived from objects, which is followed by the classification of the object according to these descriptions. Conceptual clustering was promoted from a machine learning background. There are two general methods of categorisation that apply to conceptual clustering, namely supervised and unsupervised learning [98, 128]. Supervised learning is the process of supplying category labels to objects in the machine learning algorithm, while an unsupervised learning algorithm attempts to extract the categories without any labels. The way in which the two learning methods operate are completely different. A supervised learning method uses the labels of multiple objects to extract the information from the descriptions that will accurately predict the correct category. An unsupervised

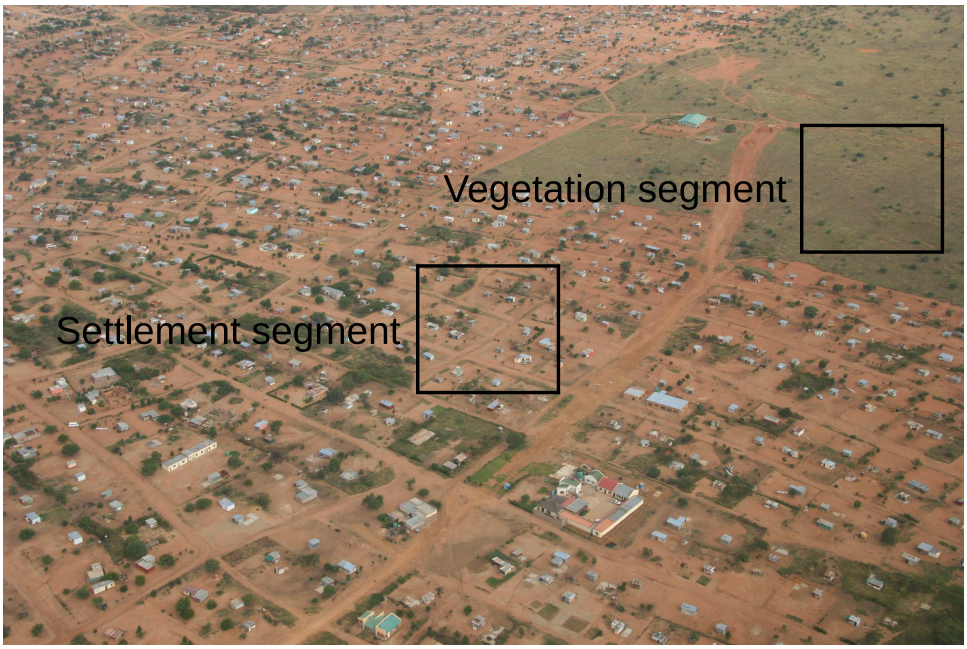

FIGURE 3.1: An aerial photo taken in the Limpopo province, South Africa of two different land cover which are indicated by a natural vegetation segment and settlement segment. A segment is defined as a collection of pixels within a predefined size bounding box.

learning method examines the inherent structure between all objects, to create categories using the most similar descriptions.

3.3 SUPERVISED CLASSIFICATION

Supervised classification is a form of conceptual clustering and is the process of allocating a predefined class label to a certain input pattern. Several concepts will be introduced throughout this thesis in considering a hypothetical problem of separating different land cover types in an image. In figure 3.1, an aerial photo is used to illustrate two different land cover types: natural vegetation and human settlement. Input patterns to the supervised classifier will be labelled as either natural vegetation or human settlement. The supervised classifier is given a set of descriptors to infer a function that assigns a predefined label to each segment of the image. This function produces output values, denoted by y, as either discrete, continuous or probabilistic in nature. The supervised classifier assigns a class label to the output value y that best matches the given input pattern and is denoted by $\mathcal{C}_k, k = 1, 2, \ldots K$, where K is equal to the number of output classes.

Land cover example: In the case of the land cover example shown in figure 3.1, K is equal to two and the output value that the supervised classifier produces will be assigned accordingly to either the natural vegetation class or the human settlement class. □

Observations from different data sources are often grouped together to form an input vector $\vec{x}$, also referred to as an input pattern. These input data sources are usually in descriptive forms that can be interpreted by humans.

Land cover example: In the case of the land cover example, the input data sources provide a colour metric that is either ordinal or real. The input data source in this instance is a set of real number values derived from the green, blue and red colours extracted from the RGB (Red Green Blue) colour buffer of all the pixels within a segment. This input data source is used to form a single input vector with three dimensions, which is defined as

$$\vec{x} = [(\text{Red value})\ (\text{Green value})\ (\text{Blue value})], \tag{3.1}$$

where $\vec{x}$ denotes the input vector. □

3.3.1 Mapping of input vectors

The ability of the supervised classifier to map the input vector $\vec{x}$ to the desired output value y is based on the performance of the learning algorithm. Given a set of input vectors $\{\vec{x}\}$ and the set of corresponding desired output values $\{y\}$, the learning algorithm seeks to infer a function that will satisfy

$$y \approx \mathcal{F}(\vec{x}). \tag{3.2}$$

This implies that the input space is approximately mapped to the desired output space by using a mapping function denoted by $\mathcal{F}$. The mapping function $\mathcal{F}$ is optimised by introducing a scoring function that evaluates the current mapping function's performance.

The learning algorithm tries to find a solution to the mapping function that will maximise the scoring function. There are two general approaches to solving equation (3.2) when a scoring function is used: empirical risk minimisation and structural risk minimisation. Empirical risk minimisation attempts to find the optimal inferred function that will minimise the error in the mapping of the input space to the output space. Structural risk minimisation includes a penalty term that provides control between the bias and variance trade-off within the learning algorithm [129]. Both approaches try to minimise the mapping error between the input and output space.

In regression analysis, the learning algorithm attempts to model the conditional distribution of the desired output values, given a set of input vectors. The desired output values will also be termed target values. Mapping typically uses an error function to determine the goodness of fit between the input and output space, and is based on the principle of maximum likelihood [130, Ch. 6 p. 195]. The likelihood $\mathcal{L}$ is computed as

$$\mathcal{L} = \prod_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{\tilde{x}}^{\,p}) P(\vec{\tilde{x}}^{\,p}), \tag{3.3}$$

where $P(\vec{\tilde{x}}^{\,p})$ denotes the probability of observing the p^{th} input vector and $p(T_{\mathcal{C}}^{p} \mid \vec{\tilde{x}}^{\,p})$ denotes the conditional probability density of observing the target value $T_{\mathcal{C}}^{p}$, given that the input vector $\vec{\tilde{x}}^{\,p}$ is present. The error function $\mathcal{E}$ is derived by converting equation (3.3) into the negative log-likelihood, which is defined as

$$\mathcal{E} = -\ln \mathcal{L} = -\sum_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{\tilde{x}}^{\,p}) - \sum_{p=1}^{P} P(\vec{\tilde{x}}^{\,p}). \tag{3.4}$$

The minimisation of the error in the mapping requires the minimisation of error function $\mathcal{E}$. The minimisation of the error function $\mathcal{E}$ in equation (3.4) will result in the maximisation of the likelihood in equation (3.3). A popular method of defining the error in mapping is the Sum of Squares Error (SSE). The minimisation of the SSE is equivalent to minimising the error function $\mathcal{E}$ in equation (3.4). The SSE equation over P patterns is given as

$$\mathcal{E} = 0.5 \sum_{p=1}^{P} \left\| \mathcal{F}(\vec{\tilde{x}}^{\,p}) - T_{\mathcal{C}}^{p} \right\|^{2}. \tag{3.5}$$

The vector $\vec{\tilde{x}}^{\,p}$ denotes the p^{th} input vector and $T_{\mathcal{C}}^{p}$ denotes the corresponding target value of the supervised classifier.

In regression analysis, the mapping derived by using equation (3.5) is regarded as optimal as long as the following three conditions are met [130, Ch. 6 p. 203]. These three conditions are:

1. The input vector set $\{\vec{\tilde{x}}\}$ is sufficiently large to capture the underlying data structure.
2. The mapping between the input space and the output space is flexible enough.
3. The optimisation of the mapping is done with a good learning algorithm to minimise equation (3.5) effectively.

In classification analysis, the learning algorithm tries to model the posterior probability of the class label. The SSE function was not specifically designed for classification problems, as it assumes that the target values are generated from a smooth deterministic function with additive zero-mean Gaussian distributed noise. The decision to use error functions within classification requires discrete class labels with optional corresponding class membership probabilities [130, Ch. 6 p. 222]. Many different approaches have been used to rescale the output values in regression problems to match the

class membership probabilities [130, Ch. 6 p. 223]. The error function shown in equation (3.4) is reformulated for a classification problem as

$$\mathcal{E} = -\sum_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{\tilde{x}}^{p}) - \sum_{p=1}^{P} P(\vec{\tilde{x}}^{p}) = -\sum_{p=1}^{P}\sum_{k=1}^{K} p(\mathcal{C}_k \mid \vec{\tilde{x}}^{p})\delta_{T_{\mathcal{C}}^{p}} - \sum_{p=1}^{P} P(\vec{\tilde{x}}^{p}). \tag{3.6}$$

If the p^{th} input vector $\vec{\tilde{x}}^{p}$ is from class $\mathcal{C}_k$ then $\delta_{T_{\mathcal{C}}^{p}} = 1$, where δ denotes the Kronecker delta symbol. The symbol k denotes the class label of interest and K denotes the number of output classes.

The output values of the supervised classifier correspond to the Bayesian posterior probabilities if the SSE function is minimised as shown in equation (3.6) [131, 132]. In a regression application it is acceptable to assume Gaussian residuals when using the SSE function, but for classification problems the target values are discrete and the additive zero-mean Gaussian distributed noise is not a good description. A more intuitive approach is to use a binomial distribution which leads to the definition of the cross-entropy error function [133].

Cross-entropy starts by observing the probability that the set of target values is $T_{\mathcal{C}_k}^{p} = \delta_{T_{\mathcal{C}}^{p}}$ when the p^{th} input pattern $\vec{\tilde{x}}^{p}$ is from class $\mathcal{C}_k$. This results in the output of a supervised classifier denoting a class membership probability $p(\mathcal{C}_k|\vec{\tilde{x}}^{p})$ [130, Ch. 6 p. 237]. The value of the conditional distribution is then expressed as

$$\mathcal{L} = \prod_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{\tilde{x}}^{p}) P(\vec{\tilde{x}}^{p}) = \prod_{p=1}^{P} \left(\prod_{k=1}^{K} (y^{p})^{T_{\mathcal{C}_k}^{p}} \right) P(\vec{\tilde{x}}^{p}), \tag{3.7}$$

which equates to the cross-entropy error function defined as

$$\mathcal{E} = -\sum_{p=1}^{P}\sum_{k=1}^{K} T_{\mathcal{C}_k}^{p} \ln\left(\frac{y^{p}}{T_{\mathcal{C}_k}^{p}}\right). \tag{3.8}$$

To ensure that the output values of the supervised classifier equates to the posterior probabilities, the following condition must hold, given as [130, 134]

$$\frac{l'(1-y)}{l'(y)} = \frac{1-y}{y}, \tag{3.9}$$

where a class of functions l which satisfies this condition is given by

$$l(y) = \int y^{r}(1-y)^{r-1}\mathrm{d}y. \tag{3.10}$$

Both the cross-entropy error function and SSE function comply with the condition set in equation (3.9). Either of these two error functions can be used in minimising the error in the mapping between the input space and output space for a given classification application. The SSE function is more

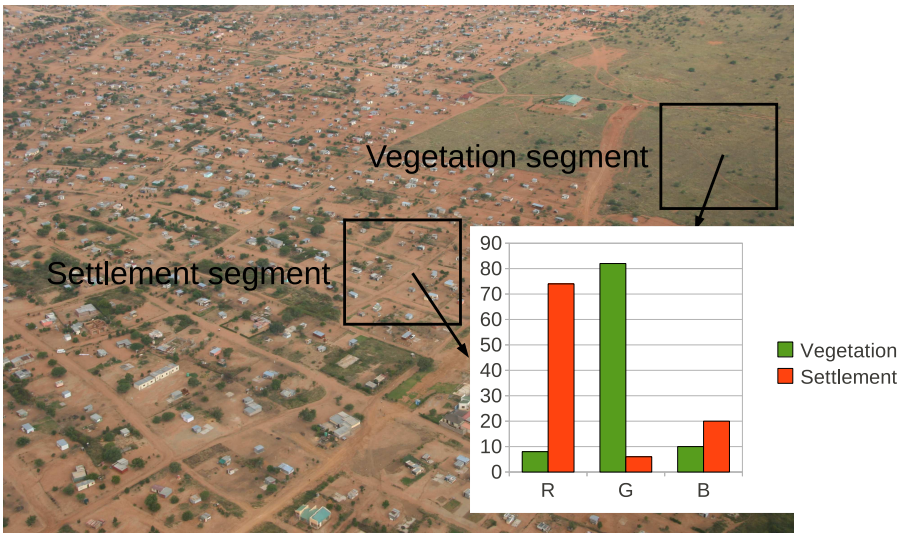

FIGURE 3.2: The same aerial photo over the Limpopo province as shown in figure 3.1, with an RGB histogram overlay showing the attributes of the two segments.

attractive owing to the ease of implementation.

Land cover example: In the case of the land cover example, a mapping of the input space to the output space is planned. The output space has two categories and the class labels are defined as; $\mathcal{C}_k \in \{\mathcal{C}_1, \mathcal{C}_2\} = \{\text{natural vegetation, human settlement}\}$. The input vectors are grouped as shown in equation (3.1). The learning algorithm infers a function that will map the input vector to the corresponding output value. These output values are grouped according to their respective class label for analysis of the supervised classifier. The learning algorithm will attempt to map the correct intensities of the RGB buffer values that will prove to be the most probable match between the input vector and the correct class membership. The learning algorithm uses a scoring system, like the SSE, to minimise the number of incorrect class memberships that are present in the current mapping. To demonstrate the results of the mapping, a histogram of each segment is shown in figure 3.2 with all participating pixels. The supervised classifier assigns segments with dominant red intensity to human settlement and segments with dominant green intensity to natural vegetation. □

The external evaluation of the mapping of the input space to the output space requires sound empirical validation. It was shown that less than 30% of new classifiers and learning algorithms are correctly assessed with proper empirical validation [127]. To ensure proper analysis, the results can be assessed by running the supervised classifier on actual (non-synthetic) data sets. This approach will ensure strong support in using the supervised classifier to solve real problems. A second approach to proper external evaluation is the subdivision of the data set into several partitions. These partitioned

data sets allow proper tuning of the supervised classifier and are used to perform cross-validation [127]. A good method of tuning a supervised classifier is to subdivide the labelled data set (input vectors with known class labels) into three different subsets:

1. A training data set, which is used to train the learning algorithm to derive a mapping function that will minimise the errors on the entire set of input vectors $\{\vec{\tilde{x}}\}$.

2. A validation data set, which is used to test the performance periodically and to mitigate any negative design effects of the supervised classifier [135]. The performance is bounded by the intrinsic noise within the training data [130, Ch. 9 p. 372].

3. A test data set, which is used to verify the performance of the supervised classifier on unseen data. The test data set is used to approximate the generalisation error; this data set is not included in the training phase or optimisation phase of the classifier.

3.3.2 Converting to feature vectors

Preprocessing of the input vector $\vec{\tilde{x}}$ before the learning algorithm and postprocessing of the output vector $\vec{y}$ after the learning algorithm is an optional procedure used to improve an algorithm's performance. The performance improves even when evaluating the outputs derived from the learning algorithm that is using a noisy and inconsistent data set [136]. Let $\vec{x}$ denote the preprocessed version of the input vector $\vec{\tilde{x}}$, and $\vec{\hat{y}}$ denote the postprocessed version of the output vector $\vec{y}$. This processing chain is illustrated in figure 3.3.

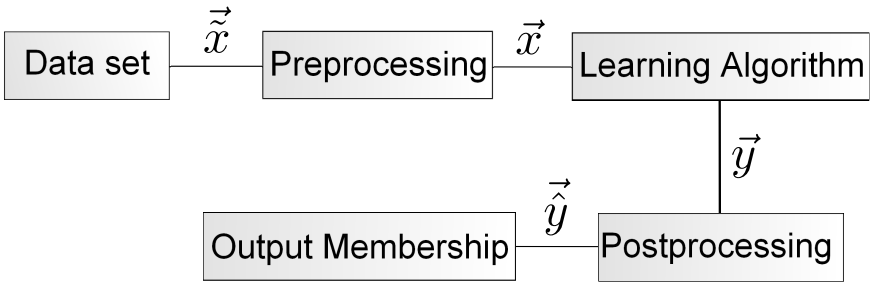

FIGURE 3.3: Flow diagram illustrating the processing steps that includes preprocessing and postprocessing.

The input data set $\{\vec{\tilde{x}}\}$ contains information from several input data sources and the information from each individual source can either be real numbers, ordinal numbers, nominal numbers or an 1-of-c coding. An adjective used to describe the numerical ranking of an object's position in a set is known as an ordinal number. A nominal number is a set of numbers used for labelling purposes alone and do not provide an indication of any other type of measurement. A 1-of-c coding is a vector representation

of the input which is an all-zero vector except in one location. The input data sets must have the same cardinality regardless of the form of the input source.

Preprocessing is the processing of raw data supplied from the input data set $\{\vec{x}\}$ to another space that can be more effectively analysed. Most machine learning algorithms learn faster and provide better performance if the input data set $\{\vec{x}\}$ is preprocessed. Numerous different methods are used for preprocessing, including: sampling, transformation, denoising, standardisation and feature extraction.

1. Sampling selects representative subsets from a large population of input patterns to perform a range of functions such as generalisation, cross-validation, etc.

2. Transformation translates the raw data set to another mathematical domain.

3. Denoising includes several techniques used to reduce the noise on samples in the input data set.

4. Standardisation refers to the scaling of the variables within the input pattern from multiple input data sources to a common scale. This common scale allows the underlying properties of the input data sources to be compared fairly within a machine learning algorithm.

5. Feature extraction extracts specific characteristics from the input patterns.

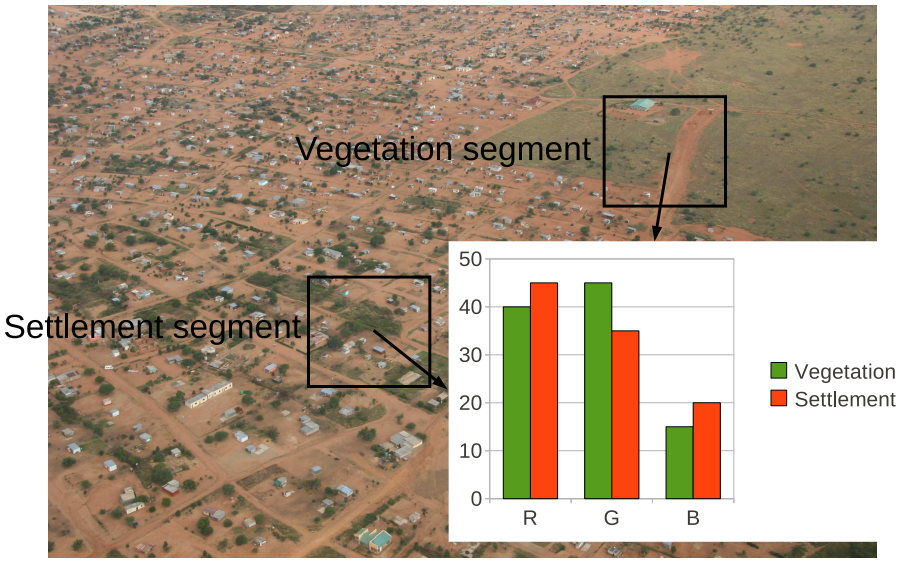

FIGURE 3.4: An alternative selection of natural vegetation and human settlement segments of the aerial photo taken in the Limpopo province using the same input vector.

Land cover example: Revisiting the aerial photo, the advantage of feature extraction as a preprocessing step can be shown when new segments are selected as shown in figure 3.4. High correlation is observed in the histogram of the three RGB buffer values when the new

segments are captured with the original input vector defined in equation (3.1). This results in poor separability within the input space and significant deterioration in the performance of the machine learning algorithm. Both segments appear highly similar in figure 3.4, and will require a complex classifier to separate the segment into the two predefined classes.

A feature extraction method is proposed in the example to extract both the moisture and reflectivity of each segment. Once extracted, these features can be placed into a feature vector $\vec{x}$ of two dimensions, which is defined as

$$\vec{x} = [(\text{Moisture})\ (\text{Reflectivity})]. \tag{3.11}$$

By using the feature vector, the human settlement segment in the example has high reflectivity and low moisture retention due to the bare soil. The natural vegetation segment has high moisture retention and low reflectivity, as shown in figure 3.5. This creates an improved feature space for the classifier to separate the two classes, regardless of the geographical positions of the segments. □

Postprocessing is an important component in the analysis phase of the design [137]. Postprocessing is the procedure of converting the output set $\{\vec{y}\}$, produced by the supervised classifier, back into either the space of the original data set or to a more user-friendly format. This extracts information from the results produced by the learning algorithm and is used to improve the overall system performance.

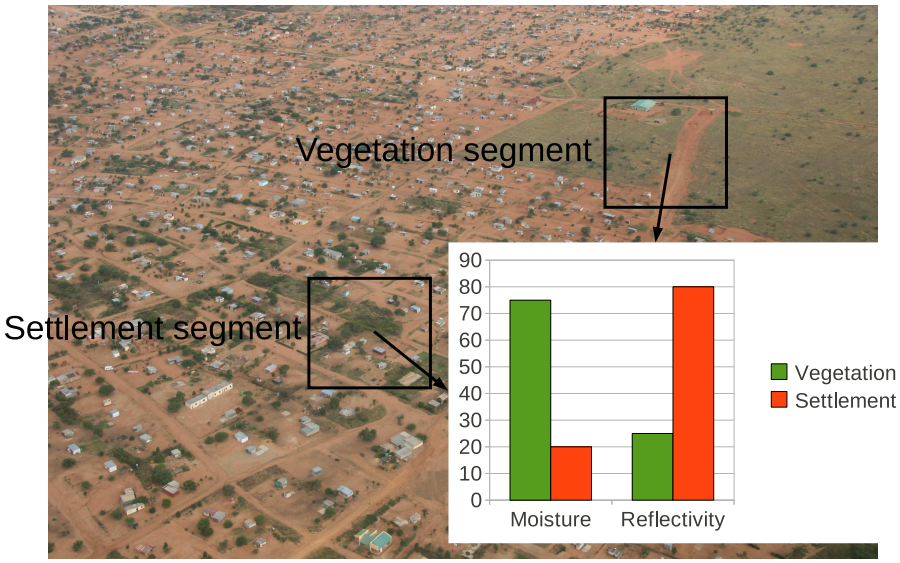

FIGURE 3.5: A new histogram created by extracting the feature vectors of the new segments selected in figure 3.4.

Numerous methods are used for postprocessing, which are categorised as: knowledge filtering, interpretation, evaluation and knowledge integration [137].

1. Knowledge filtering is the filtering of the outputs produced by the supervised classifier. This filtering improves the results when the mapping function in the supervised classifier is sensitive to the noise within the training data set.

2. Interpretation is a form of knowledge discovery where input vectors are processed by the supervised classifier and converted to an user-friendly format for human analysis. These postprocessed outputs are analysed to interpret the effect of the input vectors has on the supervised classifier. This creates a new knowledge base for further improving the results of the supervised classifier for the given application.

3. Evaluation is an approach that transforms the output values into a performance metric that is used to evaluate the performance of the current supervised classifier. Typical performance metrics include: classification accuracy, comprehensibility, computational complexity, visual interpretation, etc.

4. Knowledge integration is the process of including additional selected information sources to improve the performance of the supervised classifier.

Land cover example: In the case of the land cover example, the evaluation approach is used as a postprocessing step. The classification accuracy is used as the performance metric to evaluate the segment classification within the aerial photo. The supervised classifier produces an output vector $\vec{y}$ of either discrete, continuous or probabilistic in nature.

Let the output vector $\vec{y}$ in this example denote the vector containing all the posterior class probability values. The mapping of this vector to a class is expressed as

$$\mathcal{C}_k = \begin{cases} \mathcal{C}_1(\text{natural vegetation}) & \text{if } y_1 > y_2 \\ \mathcal{C}_2(\text{human settlement}) & \text{if } y_2 \geq y_1. \end{cases} \tag{3.12}$$

The output vector $\vec{y}$ is classed as natural vegetation when the largest value in the vector is in the first position and human settlement when in the second position. The classification accuracy is maximised by selection of the most appropriate supervised classifier and feature extraction method. □

The preprocessing of the input vector $\vec{\tilde{x}}$ will produce a new input vector $\vec{x}$ that is commonly referred to as the feature vector. Feature vectors will be used throughout the thesis as it is assumed that with proper feature extraction the overall system performance will improve.

3.4 ARTIFICIAL NEURAL NETWORKS

An artificial neural network (ANN) is a computational learning method that was inspired by the neural activities within the human brain [138]. ANNs have a range of capabilities to operate on non-linear and non-parametric data sets. The advantage of the ANN is that it can model a non-linear relationship between the input and output variables. The ANN is trained on a partial set of known data to perform either classification, estimation, simulation or prediction of underlying structures within the data.

3.4.1 Network architecture

3.4.1.1 Perceptron

The first design consideration that will be evaluated is the network architecture, as several different ANN architectures are proposed in the literature. The simplest architecture is the single-layer perceptron, which is a linear feedforward neural network that was first proposed by Frank Rosenblatt at the Cornell Aeronautical Laboratory in 1957 [139]. The perceptron is discussed, as several other concepts expand on it, as well as the important limitation the perceptron has in terms of the range of functions it can represent. The perceptron is classified as a feedforward network, as the activation of the neuron is propagated in one direction from the feature vector $\vec{x}$ to the output value y. The relationship between the feature vectors and the output is stored within the ANN's weight vector (also referred to as the synaptic strengths within the ANN), and is defined within the network as

$$y = \mathcal{F}(\vec{x}, \vec{\omega}). \tag{3.13}$$

The variable y denotes the corresponding ANN's output value and $\vec{\omega}$ denotes the weight vector. The feature vector presented to the network is denoted by $\vec{x}$ and $\mathcal{F}$ denotes the function inferred by the ANN. The weight vector $\vec{\omega}$ and the feature vector $\vec{x}$ are multiplied such that equation (3.13) expands in the case of the perceptron to

$$y = \mathcal{F}\left(\omega_0 + \sum_{i=1}^{N} x_i\omega_i\right) = \mathcal{F}\left(\omega_0 + \vec{x}\cdot\vec{\omega}\right). \tag{3.14}$$

The symbol $\mathcal{F}$ denotes the activation function and the network inputs are denoted by the feature vector $\vec{x} = \{x_1, x_2, \ldots, x_N\}$. The weight vector for the network is denoted by $\vec{\omega} = \{\omega_1, \omega_2 \ldots, \omega_N\}$ and the neuron bias by ω_0.

The perceptron is trained with the perceptron learning rule, which minimises the error function by evaluating the output value produced for a given feature vector. The perceptron learning rule processes individual feature vectors $\vec{x}$ by presenting them to the network and adjusting the weight

vector $\vec{\omega}$ iteratively to improve the classification accuracy. The perceptron learning rule attempts to fit a linear hyperplane through the feature space. The perceptron learning rule is limited by the network architecture and will only converge if the classes are linearly separable within the feature space [140, 141]. Other applications involving multiple separation regions are catered for by using multiple perceptrons in parallel, with each output value corresponding to a specific region.

3.4.1.2 Multilayer perceptron

A more popular network architecture is the multilayer perceptron (MLP). A MLP is a feedforward ANN model that contains multiple layers of neurons. The multilayer architecture allows the MLP to distinguish feature vectors within a feature space that are not linearly separable. A two-layer network architecture of a MLP, which has one hidden node layer, is illustrated in figure 3.6.

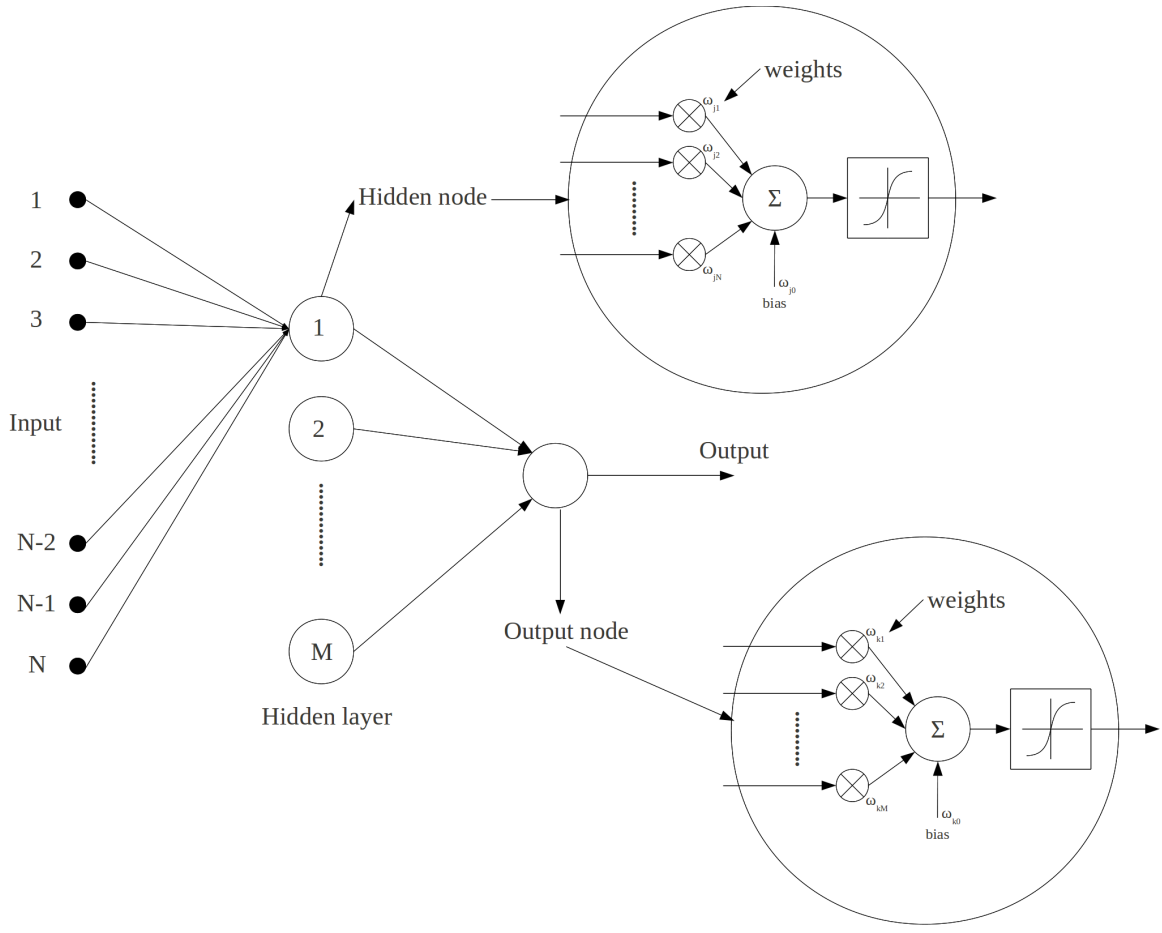

FIGURE 3.6: The topology of a feedforward multilayer perceptron with a single hidden layer.

This fully connected two-layer network's links are mathematically expressed as

$$y_k = \mathcal{F}_2\left(\omega_{k0} + \sum_{j=1}^{M} \omega_{kj}\, \mathcal{F}_1\left(\omega_{j0} + \sum_{i=1}^{N} x_i \omega_{ji}\right)\right), \tag{3.15}$$

which is more compactly expressed in vector notation as a linear multiplication between vectors as

$$y_k = \mathcal{F}_2\left(\omega_{k0} + \vec{\omega}_k \cdot \mathcal{F}_1\left(\omega_{j0} + \vec{x} \cdot \vec{\omega}_j\right)\right). \tag{3.16}$$

The network consists of N input nodes denoted by the vector $\vec{x} = \{x_1, x_2, \ldots, x_N\}$. The weight vector that connects the input nodes to the j^{th} hidden node is denoted by the vector $\vec{\omega}_j = \{\omega_{j1}, \omega_{j2} \ldots, \omega_{jN}\}$, with a corresponding neuron bias denoted by ω_{j0}. Similarly, the weight vector that connects the hidden nodes to the k^{th} output node is denoted by the vector $\vec{\omega}_k = \{\omega_{k1}, \omega_{k2} \ldots, \omega_{kM}\}$, with a corresponding neuron bias denoted by ω_{k0}. The MLP allows the use of multiple output nodes to produces an output vector that expands equation (3.16) to

$$\vec{y}_k = \mathcal{F}_2\left(\omega_{k0} + \vec{\omega}_k \cdot \mathcal{F}_1\left(\omega_{j0} + \vec{x} \cdot \vec{\omega}_j\right)\right), \tag{3.17}$$

with an output vector $\vec{y}_k$ that uses a *one-of-c* coding.

Introducing a unity input on each neuron, $x_0 = 1$, the weight vector is expanded to include the neuron bias as $\vec{\omega}_j = \{\omega_{j0}, \omega_{j1} \ldots, \omega_{jN}\}$ for the hidden nodes and $\vec{\omega}_k = \{\omega_{k0}, \omega_{k1} \ldots, \omega_{kM}\}$ for the weight vector for the output nodes. This simplifies equation (3.17) to

$$\vec{y}_k = \mathcal{F}_2\left(\vec{\omega}_k \cdot \mathcal{F}_1\left(\vec{x} \cdot \vec{\omega}_j\right)\right). \tag{3.18}$$

Monotonic functions are usually used as activation functions. Neural networks typically use a sigmoid activation transfer function in the hidden layers given in equation (3.18) as

$$\mathcal{F}(a) = \frac{1}{1 + e^{-a}}. \tag{3.19}$$

The sigmoid activation function is non-linear and allows the outputs of the neural network to be interpreted as a posterior class probability [130, Ch. 6 p. 234]. If all the activation functions within the network are converted to linear functions, then an equivalent single layer linear network without any hidden layers can be derived. This follows from the observation that the composition of successive linear transformations is itself a linear transformation [130, Ch. 4 p. 121].

By applying a linear transformation to equation (3.19), a tangent activation function is derived as

$$\mathcal{F}(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}. \tag{3.20}$$

The tangent activation function is of interest as through empirical simulations it has been proven to provide faster training of the network (section 3.4.4) [130, Ch. 4 p. 127].

The number of layers and hidden nodes within each layer are flexible design parameters. The general rule is that the layers and nodes are chosen to best model the feature space. It is known from the Kolmogorov theorem that a two-layer network with finitely many discontinuities can closely approximate any decision boundary to arbitrary precision using a sufficient number of hidden nodes with sigmoidal activation functions [142].

Several different network architectures exist and are constructed on similar concepts. The focus of this chapter will be on the MLP, but different ANNs will be briefly discussed in this chapter.

3.4.2 Regression using a multilayer perceptron

Regression analysis is a method for modelling and analysing a set of variables that focuses on the mapping relationship between a dependent variable and multiple independent variables. This extends to the understanding of inherent changes in the dependent variable when any one of the independent variables is altered. An ANN is seen as a flexible non-linear regression method, which is readily deduced from equation (3.18), where the network uses a training algorithm to find a weight $\vec{\omega}$ to map a relationship between the feature vectors and the output vectors.

The training algorithm trains the network by presenting the patterns of the training set to the network, and adjusting the weights (synapse strengths) to minimise the error function. The training algorithm derives the optimal weight by using the error function given in equation (3.4) as

$$\vec{\omega}_{opt} = \underset{\vec{\omega}\in\Omega}{\operatorname{argmin}}\{\mathcal{E}\} = \underset{\vec{\omega}\in\Omega}{\operatorname{argmin}}\left\{ -\sum_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{x}^{p}) - \sum_{p=1}^{P} P(\vec{x}^{p}) \right\}. \tag{3.21}$$

The vector $\vec{\omega}_{opt}$ denotes the optimised weight that provides the optimal fit for the mapping that is found within the weight space Ω. $P(\vec{x}^{p})$ denotes the probability of observing the p^{th} feature vector and $p(T_{\mathcal{C}}^{p} \mid \vec{x}^{p})$ denotes the conditional probability density of the target value $T_{\mathcal{C}}^{p}$ given that the feature vector $\vec{x}^{p}$ is present. The probability of observing the p^{th} feature vector denoted by $P(\vec{x}^{p})$ is an additive constant in equation (3.21), and can not be improved through the network architecture or learning algorithm procedures [130, Ch. 6 p. 195]. This term is dropped to simplify equation (3.21) to

$$\vec{\omega}_{opt} = \underset{\vec{\omega}\in\Omega}{\operatorname{argmin}}\left\{ -\sum_{p=1}^{P} p(T_{\mathcal{C}}^{p} \mid \vec{x}^{p}) \right\}. \tag{3.22}$$

The SSE function given in equation (3.5) is usually used as the error function in the MLP and is substituted into equation (3.22) to compute the optimised weight as

$$\vec{\omega}_{opt} = \underset{\vec{\omega} \in \Omega}{\operatorname{argmin}} \left\{ 0.5 \sum_{p=1}^{P} \left\| \mathcal{F}(\vec{x}^p, \vec{\omega}) - T_{\mathcal{C}}^p \right\|^2 \right\}. \tag{3.23}$$

The symbol $\mathcal{F}$ denotes the MLP's inferred map and $\vec{x}^p$ denotes the p^{th} feature vector with the corresponding target value denoted by $T_{\mathcal{C}}^p$. The training algorithm attempts to find the optimal weight $\vec{\omega}_{opt}$ that provides the smallest error function value $\mathcal{E}$.

3.4.3 Classification using a multilayer perceptron

The case was made that an ANN can be interpreted as a non-linear regression model in section 3.4.2. A regression model is used to construct a classifier, which is used to interpret the dependent variable as a posterior class membership probability. These posterior probabilities yield the most likely class for each feature vector.

The reconstruction of the regression model to behave like a classifier starts by using a 1-of-c coding output vector as shown in equation (3.18). The output layer responds like a logistic regression model when sigmoid activation functions are used in each output node [130, Ch. 6 p. 232].

By setting the target value for each training pattern to the desired posterior class probability, with a 1-of-c coding , the MLP is trained in the same manner as a regression model to obtain the optimal weight $\vec{\omega}_{opt}$. Using the optimal weight $\vec{\omega}_{opt}$, the ANN maps the feature vectors to their corresponding desired posterior class probabilities.

Since each MLP output node represents the posterior class probability for each class, a mapping function is used to select the class that has the largest posterior probability. The mapping function $\mathcal{Z}$ is expressed as

$$\mathcal{C}_k = \mathcal{Z}(\vec{y}), \tag{3.24}$$

where $\mathcal{C}_k$ denotes the class membership and $\vec{y}$ denotes the MLP output vector.

Deriving the optimal weight $\vec{\omega}_{opt}$ will assign the highest posterior class probability to the correct class membership $\mathcal{C}_k$ for the corresponding feature vector $\vec{x}$ and is expressed as

$$P(\mathcal{C}_k = \mathcal{C}_f | \vec{x}) > P(\mathcal{C}_k = \mathcal{C}_g | \vec{x}) \qquad \forall (f \neq g), \tag{3.25}$$

where $P(\mathcal{C}_k = \mathcal{C}_f | \vec{x})$ denotes the probability of class membership of $\mathcal{C}_k$ being equal to $\mathcal{C}_f$, given the feature vector $\vec{x}$ was presented to the MLP.

The probability of error is equal to the probability of falling within the incorrect decision region [143]. The probability of error for the class membership $(\mathcal{C}_k = \mathcal{C}_c)$ of the MLP is computed as

$$P_e = 1 - \int_{\mathcal{R}_c} p(\vec{x}|\mathcal{C}_k = \mathcal{C}_c)P(\mathcal{C}_k = \mathcal{C}_c)\mathrm{d}\,\vec{x}. \tag{3.26}$$

The procedure of minimising the probability of error P_e on the global population group of feature patterns, requires that the complete population's class memberships be known. This is not possible for most actual data sets (non-synthetic), as acquiring the class membership on all feature vectors is infeasible. The objective of the training algorithm is to minimise the probability of error P_e on the global population by only using a subset of feature vectors with known class membership.

An external evaluation process is used for minimising the probability of error, as discussed in section 3.3.1, that is used to improve overall system performance. The subdivision of the labelled data set (feature vectors with known class memberships) for the MLP is briefly discussed:

1. A training data set is used to train the ANN to minimise the mapping errors on the data set by means of adaptation of the weights. A popular method of calculating the error in the mapping is the SSE shown in equation (3.5). The minimisation of the error is accomplished by initialisation the weights with random values, followed by presenting the training data set to the network to adjust the weights accordingly. Several different training algorithms exist in the literature that attempts to minimise the error on the training data set.

2. A validation data set is periodically used to test the network performance to mitigate the effects of overfitting [135]. A neural network with more hidden nodes has the ability to learn a more complex mapping [144]. A complex mapping in the feature space has the ability to isolate complex regions [145]. If proper design of the MLP is not adhered to, the network not only extracts the characteristics of the feature space, but also memorises the noise within the training data set.

3. A test data set is used to validate the performance of the MLP. The test data set is used to estimate the generalisation error, and this data set is not included in the training phase or optimisation phase.

3.4.4 Training of neural networks

As stated previously, the MLP network relies on the weights to assign the feature vector to the class membership that has the largest posterior probability. This is under the assumption that the optimal weight $\vec{\omega}_{opt}$ is used to provide the decision regions. The design of a proper MLP requires the estimation of a weight $\vec{\omega}$ that will minimise the error function and generalisation error for an application.

The error function $\mathcal{E}(\vec{\omega})$ is improved with a training algorithm by searching through the weight space Ω, that uses the SSE metric given in equation (3.5), which is continuous and twice differentiable in $\mathbb{R}^{|\vec{\omega}|}$, where $|\vec{\omega}|$ denotes the total number of weights in the network.

A local minimum of $\mathcal{E}(\vec{\omega})$ is defined as a vector $\vec{\omega}_{\text{local}}$, such that $\mathcal{E}(\vec{\omega}_{\text{local}}) \leq \mathcal{E}(\vec{\omega})$ for all $|\vec{\omega}_{\text{local}} - \vec{\omega}| < D_{\vec{\omega}}$ in $\mathbb{R}^{|\vec{\omega}|}$, where $D_{\vec{\omega}}$ is a predefined constant.

It is possible that $\mathcal{E}(\vec{\omega})$ may contain multiple local minima. Let S_{local} denote the set of all such local minima of $\mathcal{E}(\vec{\omega})$ on $\mathbb{R}^{|\vec{\omega}|}$. The global minimiser of $\mathcal{E}(\vec{\omega})$ is then defined as

$$\vec{\omega}^* = \underset{\vec{\omega} \in S_{\text{local}}}{\operatorname{argmin}} \mathcal{E}(\vec{\omega}). \tag{3.27}$$

Note that $\mathcal{E}(\vec{\omega}^*) \leq \mathcal{E}(\vec{\omega}), \forall\, \vec{\omega} \in \mathbb{R}^{|\vec{\omega}|}$. In addition, the derivative of the error function, $\nabla\mathcal{E}(\vec{\omega})$, is zero for all $\vec{\omega} \in S_{\text{local}}$.

Owing to the non-linear nature of the error function $\mathcal{E}(\vec{\omega})$, no closed form solution can be obtained. Many iterative algorithms can be applied to minimise the error function $\mathcal{E}(\vec{\omega})$, most of which iteratively adjust the current weight $\vec{\omega}_i$ such that

$$\vec{\omega}_{(i+1)} = \vec{\omega}_i + \Delta\vec{\omega}_i, \tag{3.28}$$

where $\Delta\vec{\omega}_i$ is typically chosen such that $\mathcal{E}(\vec{\omega}_{i+1}) < \mathcal{E}(\vec{\omega}_i)$. The manner in which $\Delta\vec{\omega}_i$ is determined at each epoch i, will allow the algoirthm to converge to either a local minimum or a global minimum of the error function $\mathcal{E}(\vec{\omega})$.

Owing to the inherent difficulty of reliably locating the global minimum $\vec{\omega}^*$ of the error function $\mathcal{E}(\vec{\omega})$, most algorithms instead attempt to find the best local minimum, given a finite number of iterations, which may be called an *acceptable local minimum* for a given training data set.

Another important aspect that should be considered is that the global minimum of the error function $\mathcal{E}(\vec{\omega})$ on a given training data set may not necessarily result in the best generalisation performance for the application, hence it is typically sufficient to find an *acceptable local minimum* [130, Ch. 6 p. 194]. Several different approaches to calculating the weight update set $\vec{\omega}_i$ in equation (3.28) will now be discussed.

3.4.5 First order training algorithms

3.4.5.1 Gradient descent

The gradient of the error function $\mathcal{E}(\vec{\omega})$ always points in the direction in which $\mathcal{E}(\vec{\omega})$ will decrease most rapidly in its local vicinity. Algorithms that exploit the gradient information can typically locate a minimum in fewer iterations than algorithms that do not use gradients. The gradient descent algorithm

propagates along the negative slope of the error function [146]. The weight update $\Delta\vec{\omega}_i$ given in equation (3.28) is iteratively computed in the gradient descent approach at each epoch i as

$$\Delta\omega_i = -\mathfrak{L}_i \nabla\mathcal{E}|_{\vec{\omega}_i} + \mathcal{M}\Delta\omega_{(i-1)}. \tag{3.29}$$

The variable $\mathfrak{L}_i$ denotes the learning rate and $\mathcal{M}$ denotes the momentum parameter. The derivative of the error surface evaluated at weight $\vec{\omega}_i$ is denoted by $\nabla\mathcal{E}|_{\vec{\omega}_i}$. The algorithm incorporates a learning rate parameter $\mathfrak{L}_i$ that scales the rate of propagation of the weight down the negative slope. The correct adjustment of the learning rate improves the convergences onto a local minimum of $\mathcal{E}(\vec{\omega})$. If the learning rate is set too high, the algorithm has difficulty in stabilising the weight and might cause $\vec{\omega}_i$ to oscillate around the minimum, preventing convergence. When the learning rate is set too low, the algorithm takes a long time to converge. Common practice states a gradual decrease in the learning rate $\mathfrak{L}_i$ during training minimises the chance of oscillations within the training process.

Additional information for the training algorithm is acquired from the eigenvalues of the Hessian matrix of the error. The learning rate can be set to $\mathfrak{L}_i = (2/\lambda_{\max})$ to improve the performance further, where $\lambda_{\max}$ denotes the largest eigenvalue in the Hessian matrix [147]. The disadvantage is that the Hessian matrix varies as the weight is updated at each iteration with $\Delta\omega_i$ and calculating the Hessian matrix is computationally expensive.

If the Hessian matrix is calculated, a metric is defined for characterising the expected rate of convergence of steepest descent. This metric is the ratio of the smallest eigen value $\lambda_{\min}$ and the largest eigen value $\lambda_{\max}$ and is expressed as

$$R(\lambda) = \frac{\lambda_{\min}}{\lambda_{\max}}. \tag{3.30}$$

A very small value of $R(\lambda)$ usually means that the error surface contours are highly elongated elliptical in shape and the progress to the minimum will be extremely slow when using steepest gradient descent. The momentum parameter $\mathcal{M}$ is used for compensating when the ratio $R(\lambda)$ is small [148]. The momentum term leads to faster convergence towards the minimum without causing divergent oscillations, which may appear when the learning rate is too large. The momentum parameter acts as a lowpass filter to incorporate recent trends in movement along the error surface. Inclusion of momentum generally leads to a significant improvement in the performance of gradient descent.

3.4.5.2 Resilient backpropagation

Resilient backpropagation (RPROP) is a first-order heuristic algorithm that is used for training a feedforward neural network [149]. The RPROP algorithm is based on the notion that the optimal

step size, at a given iteration, will differ for each dimension of $\vec{\omega}_i$. RPROP thus maintains a separate weight update step $\Delta\vec{\omega}_{i,j}$ for each dimension j. A heuristic is employed to adjust each $\Delta\vec{\omega}_{i,j}$ at every epoch as follows; if the sign of the gradient dimension j has changed from that of the previous epoch, reduce the step size $\Delta\vec{\omega}_{i,j}$ and reverse its sign, otherwise increase the step size $\Delta\vec{\omega}_{i,j}$.

The reasoning is that the gradient sign in dimension j will change if the algorithm has moved over a local minimum, thus the algorithm must take smaller steps in the following iterations to approach the minimum. This is analogous to implementing standard steepest descent, but with a separate adaptive learning rate for each dimension.

3.4.5.3 Quickprop

The last heuristic first order training algorithm that will be discussed in the section is the Quickprop algorithm [150]. Quickprop treats each weight within the network as quasi-independent. The idea is to approximate the error surface with a quadratic polynomial function. The gradient information derived with backpropagation is used to determine the coefficients of the polynomial. The step sizes are fixed within the weight to ensure that the algorithm will converge to a minimum. The Quickprop algorithm uses a local quadratic surface and cannot distinguish between propagating upwards or downwards on the error surface. This drawback is easily overcome by determining the propagation direction by using an algorithm such as the gradient descent algorithm in the first epoch.

3.4.5.4 Line search

The line search is a one dimensional minimisation problem, which finds the minimum of the error function along a particular search direction [151]. It is used in several different algorithms to reduce computational complexity and will be discussed briefly. Suppose that a certain algorithm is considering a particular search direction $\vec{d}_i$ through the weight space for a potential future weight update (equation (3.28)), the minimum along that particular search direction is calculated as

$$\vec{\omega}_{(i+1)} = \vec{\omega}_i + \Delta_d \vec{d}_i, \tag{3.31}$$

where the step size parameter Δ_d is calculated as

$$\mathcal{E}(\Delta_d) = \underset{\Delta_d \in \mathbb{R}}{\operatorname{argmin}}\, \mathcal{E}(\vec{\omega}_i + \Delta_d \vec{d}_i). \tag{3.32}$$

In summary, the line search finds the optimal step size for a selected search direction. The line search algorithm itself has several constraints, as every line minimisation involves several internal error function evaluations, which could be computationally expensive. Line search introduces additional

parameters whose values will determine the termination criterion for each line search.

3.4.5.5 Conjugate gradient

The concept of choosing improved search directions is the main principle behind the conjugate gradient algorithm [130, 152]. The conjugate gradient algorithm evaluates the performance of conjugate directions with line search algorithms. The conjugate gradient algorithm is an iterative approach and is applied with ease to applications having feature vectors with several dimensions. The conjugate gradient algorithm operates under the assumption of a quadratic error function with a positive definite Hessian matrix [130, Ch. 7 p. 276].

Owing to the fact that most data sets have a non-quadratic error surface, there is a high probability that if the step size is small enough, the evaluation of $\mathcal{E}(\vec{\omega}_i + \Delta\vec{\omega}_i)$ will fall on an error surface that is approximately quadratic in its local vicinity. This may lead to fast convergence to a minimum. Under similar reasoning, if the local vicinity of the error surface is non-quadratic, the conjugate gradient algorithm will converge slowly to the minimum.

The performance of the conjugate gradient algorithm is dependent on the type of line search algorithm used. Line search allows the conjugate gradient algorithm to find the step size without evaluating the Hessian matrix.

3.4.6 Second order training algorithms

The successive use of the local gradient vector as the search direction does not always result in the most optimal search trajectory. The local gradient does not necessarily point directly at the minimum, which may cause oscillating behaviour in a steepest descent algorithm. This slow progression to the minimum can even be present with a quadratic error surface for poorly conditioned networks. The convergence speed can be improved by evaluating and choosing superior search directions while propagating down the error surface.

3.4.6.1 Newton method

The Newton method is an algorithm that calculates the Newton direction by assuming a positive definite Hessian matrix and a quadratic error surface. The trajectory from the current weight to a nearby minimum is known as the Newton direction. There are three obstacles when using the Newton method [130, Ch. 7 p. 286]:

1. The calculation of the Hessian matrix is computationally expensive for a non-linear MLP which requires $\mathcal{O}(P|\vec{\omega}|^2)$ operations to compute, where P is the number of feature vectors to evaluate

and $|\vec{\omega}|$ is the dimension of the weights.

2. The calculation of the inverted Hessian matrix is also computationally expensive, as it requires $\mathcal{O}(|\vec{\omega}|^3)$ iterations to compute.

3. Regardless of whether the Hessian matrix is positive definite, the Newton direction can point to either a maximum or a minimum.

The third obstacle can be resolved by using a model trust region approach that adds a positive definite symmetrical matrix to the Hessian matrix [130, Ch. 7 p. 287], which is expressed as

$$\mathbf{H}_{\text{new}} = \mathbf{H}_{\text{old}} + A\mathbf{I}. \tag{3.33}$$

The matrix $\mathbf{H}_{\text{old}}$ is the current Hessian matrix and $\mathbf{H}_{\text{new}}$ is the adjusted Hessian matrix. The identity matrix is denoted by $\mathbf{I}$ and A denotes a constant factor. Equation (3.33) provides the Newton direction if the constant factor A is set to a small value or it can provide the negative gradient descent direction if the constant factor A is set to a large value [130, Ch. 7 p. 287].

The last consideration is the step size along the Newton direction. The step size calculated within the Newton method is made under the assumption that the error surface is quadratic in shape. Most real data sets have non-quadratic error surfaces and when the step size is too large, the algorithm may fail to converge.

3.4.6.2 Quasi-Newton method

A more practical implementation of the Newton method is the Quasi-Newton method. The Quasi-Newton method is an approximation of the Newton method, as the Hessian matrix is computationally expensive for complex neural networks [153]. The Quasi-Newton method approximates the inverted Hessian matrix over several iterations, using only the first derivative of the error function. After each iteration the estimated inverse Hessian matrix approximates more closely the real inverse Hessian matrix for a given weight.

A popular quasi-Newton algorithms is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. The BFGS algorithm updates the estimated Hessian matrix in each epoch to converge to the actual Hessian matrix. The algorithm starts with the identity matrix to ensure that the minimum is tracked and not the maximum. The length of the Newton step is calculated using a proper line search to ensure stability. The accuracy of the line search is not as critical as it was with the conjugate gradient algorithm [154].

The disadvantages of the Newton and the Quasi-Newton methods are the storage requirements and the number of iterations to approximate the Hessian matrix [130, Ch. 7 p. 289]. Because of the

non-quadratic error surface of most data sets, the approximate Hessian matrix must be estimated after each weight update to ensure correct minimisation of the error function. The second disadvantage of these methods is the introduction of the model trust region constant factor A and the correct scaling of this constant.

3.4.6.3 Levenberg-Marquardt algorithm

The last second order training algorithm that will be discussed in the section is the Levenberg-Marquardt algorithm [155, 156]. The Levenberg-Marquardt algorithm is an approach to derive the second-order derivative without computing the Hessian matrix, as with the Quasi-Newton method. The Levenberg-Marquardt algorithm is specifically designed to minimise the SSE. This is accomplished by approximating the function in equation (3.5) with linearisation as

$$\mathcal{F}(\vec{x}_i, \vec{\omega}_i + \Delta\omega_i) \approx \mathcal{F}(\vec{x}_i, \vec{\omega}_i) + \vec{J}_i \Delta\omega_i. \tag{3.34}$$

The vector $\vec{J}_i$ is a gradient row vector of $\mathcal{F}$ with respects to $\vec{\omega}_i$ and is computed as

$$\vec{J}_i = \frac{\partial \mathcal{F}(\vec{x}_i, \vec{\omega}_i)}{\partial\, \vec{\omega}_i}. \tag{3.35}$$

Substituting the approximation of equation (3.34) into equation (3.5) is expressed as

$$\mathcal{E}(\vec{\omega} + \Delta\omega_i) = 0.5 \sum_{p=1}^{P} \left\| \mathcal{F}(\vec{x}^p, \vec{\omega}) + \vec{J}_i \Delta\omega_i - T_{\mathcal{C}}^p \right\|^2. \tag{3.36}$$

By setting the derivative as

$$\frac{\partial \mathcal{E}(\vec{\omega} + \Delta\omega_i)}{\partial\, \Delta\omega_i} = 0, \tag{3.37}$$

equation (3.36) can be expressed as

$$(\mathbf{J}^{\mathrm{T}}\mathbf{J})\Delta\omega_i = \mathbf{J}^{\mathrm{T}} \left(0.5 \sum_{p=1}^{P} \left\| \mathcal{F}(\vec{x}^p, \vec{\omega}) - T_{\mathcal{C}}^p \right\|^2 \right). \tag{3.38}$$

The Jacobian matrix is denoted by $\mathbf{J}$, with each row containing $\vec{J}_i$. This Jacobian matrix contains the first derivatives of the neural network's error. Levenberg added a non-negative damping factor λ_{damp}, which is adjusted at each epoch. This is expressed as

$$(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda_{\mathrm{damp}}\mathbf{I})\Delta\omega_i = \mathbf{J}^{\mathrm{T}} \left(0.5 \sum_{p=1}^{P} \left\| \mathcal{F}(\vec{x}^p, \vec{\omega}) - T_{\mathcal{C}}^p \right\|^2 \right). \tag{3.39}$$

A smaller damping factor λ_{damp} value allows the algorithm to behave more like the Newton method, while a larger damping factor λ_{damp} value allows the algorithm to behave like the gradient descent method.

If the damping factor λ_{damp} value is set too high, the inversion of $(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda_{\text{damp}}\mathbf{I})$ contributes nothing to the algorithm. Marquardt then contributes a variable that will scale each component of the gradient according to the curvature. This results in the Levenberg-Marquardt equation given as

$$(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda_{\text{damp}}\text{diag}(\mathbf{J}^{\mathrm{T}}\mathbf{J}))\Delta\omega_i = \mathbf{J}^{\mathrm{T}}\left(0.5\sum_{p=1}^{P}\left\|\mathcal{F}(\vec{x}^{\,p},\vec{\omega}) - T_{\mathcal{C}}^{p}\right\|^2\right), \tag{3.40}$$

where the identity matrix $\mathbf{I}$ in equation (3.39) is replaced to ensure larger propagation in the desired direction when the gradient becomes smaller.

3.5 OTHER VARIANTS OF ARTIFICIAL NEURAL NETWORKS USED FOR CLASSIFICATION

3.5.1 Radial basis function network

The radial basis function (RBF) network is another ANN that is discussed in this chapter [130, 157]. In the case of the MLP, the hidden neurons create multi-dimensional hyperplanes to separate different classes within the feature space. In the case of the RBF network, the network uses local kernel functions, which are represented by a prototype vector within each hidden neuron to model different classes. The activation of the hidden neurons is based on the distance from the prototype vector, which in effect creates a spherical multi-dimensional hypersphere. The RBF network can be used for classification; the posterior class probabilities of the network at the output is computed as

$$p(\mathcal{C}_k|\vec{x}) = \sum_{d=1}^{D}\vec{\omega}_{kd}\varphi_d(\vec{x}). \tag{3.41}$$

The RBF uses D basis functions that are denoted by φ_d. The φ_d basis function in the network's hidden neurons is expressed as a normalised basis function given by

$$\varphi_d(\vec{x}) = \frac{p(\vec{x}\,|\,d)P(d)}{\sum_{e=1}^{E}p(\vec{x}\,|\,e)P(e)} = p(d\,|\,\vec{x}). \tag{3.42}$$

The d^{th} basis function evaluating feature vector $\vec{x}$ is denoted by $\varphi_d(\vec{x})$ [130, Ch. 5 p. 181]. The denominator is used to normalised the basis function by iterating through all the basis functions within the network with variable e. The outputs of all the radial basis functions are linearly combined with a weight vector to form an output vector. The weight vector for each output node is given by

$$\vec{\omega}_{kd} = \frac{p(d \,|\, \mathcal{C}_k) P(\mathcal{C}_k)}{P(d)} = p(\mathcal{C}_k \,|\, d). \tag{3.43}$$

The radial basis function network can be designed in a fraction of the time required to train a MLP, but requires a large sample of input vectors to train reliably [158].

3.5.2 Self organising map

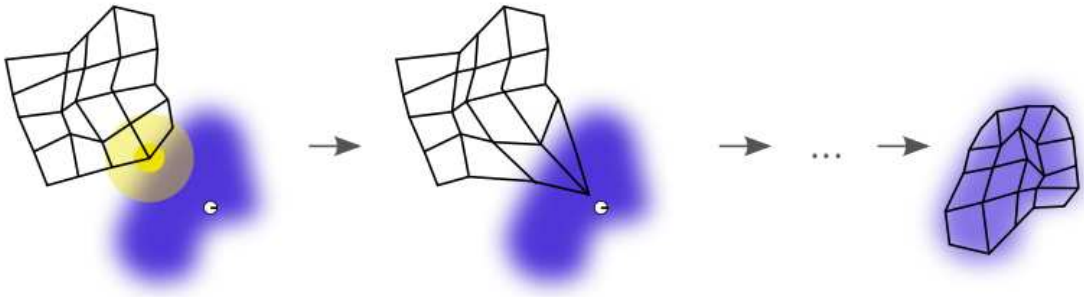

FIGURE 3.7: The training of the SOM will map the gridded topological map to the training data set.

Another popular ANN design is the Self Organising Map (SOM) [159, 160]. The SOM is trained with an unsupervised learning algorithm to convert a high dimensional data set to a lower dimensional representation of the data, typically two-dimensional. The SOM converts the higher dimensional data set to a lower dimension using a topological map that comprises prototype neurons. This topological map is used to illustrate the relationship between feature vectors by placing similar feature vectors in close vicinity to each other on the map and dissimilar feature vectors further apart. Each prototype neuron has a prototype vector; these are comparable to weights in other ANNs, and are initialised to either random samples or uniform subsampling of the feature vector set.

The training algorithm used on the SOM is a competitive learning algorithm which searches for the part of the network that strongly responds to the given feature vector. The response is evaluated by presenting a feature vector $\vec{x}$ to the SOM's prototype neurons to determine the Euclidean distances to all prototype vectors. The prototype neuron with the most similar prototype vector is termed as the best matching unit (BMU). The prototype vector within the BMU is adjusted towards the feature vector. The prototype neurons in close vicinity of the BMU in the topological map are known as the neighbouring neurons and are also updated to a certain degree towards the current feature vector. The magnitude of the adaptation of the neighbouring neurons decreases with epochs and distance from the BMU.

A SOM is trained in batch mode, where all the feature vectors are presented to the network and only the BMU is trained. A monotonically increasing penalty factor is added to that feature vector to

ensure that a particular feature vector does not dominate the training algorithm. In the beginning of the training phase, the neighbourhood relationship within the topological map is large, but with each epoch the mapping of neighbourhood size shrinks within the map and the network converges (Figure 3.7). The creation of a topological map, particularly if the data are not intrinsically two-dimensional, may lead to suboptimal placement of the feature vectors [130, Ch. 5 p. 188].

3.5.3 Hopfield networks

The third ANN briefly discussed is the Hopfield network. A Hopfield network is a recurrent network with feedback loops between the outputs and the inputs [161–163]. The neurons in the Hopfield network have binary threshold activation functions and the internal state of the network evolves to a stable state that is a local minimum of the Lyapunov function. The Lyapunov function is a monotonically decreasing energy function that puts less emphasis on the previous set of feature vectors than on the current set of feature vectors. A Hopfield network is an associative memory, which enables it to train on a set of target vectors, and when a new set of feature vectors are presented it will cause the network to settle into an activation pattern corresponding to the most closely resembling target vector presented in the training phase. The drawback of the Hopfield network is that it can only retrieve all the fundamental memorised target vectors [164].

3.5.4 Support vector machine

A Support Vector Machine (SVM) is a supervised learning algorithm that was developed in the AT&T Bell laboratories in 1995. SVM is based on the principle of structural risk minimisation, which involves constructing a non-linear hyperplane with kernel functions to separate the feature space into several output regions [129].

The SVM training algorithm attempts to fit a non-linear hyperplane through the feature space. It focuses on maximising the distance between the decision boundary and the sets of feature vectors. The SVM is a maximum margin classifier and does this by identifying the feature vectors within the feature space that prohibits the training algorithm from increasing the margin between the output regions. These feature vectors are called the support vectors within the feature space.

The method by which the SVM handles non-separable feature vectors is relaxing the constraints on the hyperplane that maximises the separability. This is accomplished by including a cost function into the separating marginal regions and penalises the feature vectors that severely hinders the SVM's performance.

The advantage of a SVM is that it uses a weighted sum of kernel functions to separate the feature vectors in the feature space. The kernel functions reduce the number of dimensions and decouples the

computational complexity of the SVM from the feature vector's dimensionality. Another advantage is that it is less prone to overfitting. If the hyperplanes are properly designed, the results of the SVM are similar to a properly designed MLP classifier [165].

A disadvantage in the SVM is that the choice of kernel used in the algorithm is very important. Several adjustable dimensions of the parameters are encapsulated within the kernel, which only leaves the penalty parameter available for adjustment. Proper choice of kernel is still an active research field; using prior knowledge during kernel selection usually improves performance. Further disadvantages are potentially slow training and substantial memory usage during training. It is observed that the speed is significantly reduced when training on larger data sets [129].

The last design consideration is the proper setting of the penalty term used to classify non-separable feature vectors. This penalty term must be optimised either through brute force searching or any other heuristic search methods.

3.6 DESIGN CONSIDERATION: SUPERVISED CLASSIFICATION

In this section a brief overview is given of some considerations when designing a supervised classifier. The first consideration is the investigation of the input vector set $\{\vec{x}\}$ and the desired output vector set $\{\vec{y}\}$. The first question is whether a plausible mapping function exists that can successfully map the input space to the output space with meaningful descriptors. Should the input vector set $\{\vec{x}\}$ be preprocessed into a feature vector set $\{\vec{x}\}$ and should the output vector set $\{\vec{y}\}$ be postprocessed to improve overall performance? This analysis provides insight into all further design decisions.

On completing the analysis, the next step is finding a suitable supervised classifier. The choice of ANN and the corresponding training algorithm is critical in finding acceptable performance in the mapping. The reason why only acceptable performance is pursued, rather than optimal, is that finding the best feature vector set and the optimal supervised classifier requires an exhaustive search, which is not feasible in terms of computational costs.

The adaptation for using a supervised classifier optimally entails the use of a proper training algorithm. Training algorithms typically focus on monotonically decreasing the value of the error function. Unfortunately, this type of training algorithm is more prone to becoming trapped in a local minimum when a small incremental steps are used. If the incremental step size is too large, the training algorithm will overshoot the minima. The convergence rate of the training algorithm is hindered even more when the direction of the propagation in the error surface does not point to the minimum. Several different training algorithms try to find the direction to the minimum since the local gradient does not always point straight at the minimum.

The training algorithm utilises training patterns in two general methods: iterative and batch

learning. Batch learning is an offline learning method that evaluates all the available training patterns before adapting the network. Iterative learning can either be online or offline, as it only evaluates sequentially a single training pattern before adapting the network [166]. An offline system stores all its patterns in a data set, while an online system processes and discards a pattern.

Another important consideration is that most ANNs are prone to overfit. This can be controlled by proper implementation of an early stopping criteria. The most common methods of stopping a training algorithm are:

1. The preset number of epochs is reached.
2. The predetermined computational time has expired in the execution of the training algorithm.
3. The training algorithm is stopped when a predefined lower threshold of the error function is reached.
4. The training algorithm is stopped when the first derivative of error function falls below a predefined lower threshold.
5. The error on the validation data set (section 3.4.3) is minimised.

It is commonly believed that a MLP with many hidden neurons has a high generalisation error, as the network is more prone to overfit [130, Ch. 1 p. 14]. This excess capacity (large number of hidden neurons) offers the MLP the ability to learn more complex models. If too much training is applied on a MLP, with excess capacity, it starts to learn the intrinsic noise within the data set. This is an undesirable property in most applications of a supervised classifier and much emphasis is placed on limiting the capacity of the network to prevent overfitting (Occam Razor's principle). It is also commonly believed that a MLP network with a large number of hidden neurons requires a large number of training vectors (section 3.4.3) to find a suitable mapping function between the feature and output space [167].

This common knowledge was questioned when a contradiction was shown by Caruana *et al.* [168]. They showed that a MLP with excess capacity has better generalisation error than a MLP with sufficient capacity. A MLP can be trained to map highly non-linear regions with a large number of hidden neurons, but still have the ability to retain a proper mapping of the linear regions [168] with a limited number of training patterns.

The concept is based on a slowly converging training algorithm that will first train the linear regions and then progress to the non-linear regions. If a good stopping criterion is adhered to, the training algorithm will terminate properly before it overfits. Some second-order methods, e.g. conjugate gradient descent algorithm, do not exhibit this property, as they have fast convergence, and will indeed overfit if the network has excess capacity.

This behaviour is intrinsically built into the slower training algorithms, as the set of weights $\{\vec{\omega}\}$ is usually initialised with small non-zero values and only after many epochs do certain values within the weights tend to large values. This implies that the MLP first considers simple mapping functions before exploring more complex functions [168, 169].

Small initial values are used within the weights to ensure that there is no saturation of the sigmoidal activation function. This initialisation ensures that contours are created on the error surface when backpropagation is applied in the training phase, otherwise the saturation of the sigmoidal activation functions will create a very flat error surface.

The last design consideration is the choice of initial weights, which is very important in achieving good results. A suitable initial choice has the potential of allowing the training algorithm to train fast and efficiently. Even stochastic algorithms, such as gradient descent, which have the possibility of escaping from local minima, can be sensitive to the initial weights used. This results in the rule of thumb to run several training phases with different initial weights in parallel to evaluate the performance of different minima [130, Ch. 7 p. 260].

The ANN used in this thesis is the MLP with a stochastic gradient descent as used by Caruana *et al.* [168]. The gradient descent uses a learning and momentum parameter in the training process to speed up convergences and a validation data set to apply proper early stopping.

3.7 SUMMARY

This chapter presented a methodology for designing a supervised classifier for real world applications. Emphasis was placed on the design of a proper mapping function between the input and output space. The mapping function's fit was then measured using a suitable error function. The performance of the classifier improves when a training method is used which adapts the network to minimise the error function.

This can be seen as a regression approach to determine the relationship between the dependent and independent variables within the network. The output values produced by the network can be interpreted as a set of posterior class probabilities under certain assumptions. The chapter concludes with a range of good practice notes on how to design and develop a good supervised classifier.