Convolutional Support Vector Machines For Image Classification

Benjamin Sugg

Submitted by Benjamin Sugg to the University of Exeter as a thesis for the degree of Masters by Research in Computer Science, May 2018.

Contents

Abstract

The Convolutional Neural Network (CNN) is a machine learning model which excels in tasks that exhibit spatially local correlation of features, for example, image classification. However, as a model, it is susceptible to the issues caused by local minima, largely due to the fully-connected neural network which is typically used in the final layers for classification. This work investigates the effect of replacing the fully-connected neural network with a Support Vector Machine (SVM). It names the resulting model the Convolutional Support Vector Machine (CSVM) and proposes two methods for training. The first method uses a linear SVM and it is described in the primal. The second method can be used to learn a SVM with a non-linear kernel by casting the optimisation as a Multiple Kernel Learning problem. Both methods learn the convolutional filter weights in conjunction with the SVM parameters. The linear CSVM (L-CSVM) and kernelised CSVM (K-CSVM) in this work each use a single convolutional filter, however, approaches are described which may be used to extend the K-CSVM with multiple filters per layer and with multiple convolutional layers. The L-CSVM and K-CSVM show promising results on the MNIST and CIFAR-10 benchmark datasets.

Chapter 1

Introduction

1.1 Motivation

The Support Vector Machine (SVM) (Boser et al., 1992) is an influential machine learning model which has been successfully applied to a broad range of learning tasks. It is an attractive model to researchers due to the fact that it may be trained by optimising a convex error function. Thus, training is guaranteed to find parameters which achieve a globally optimum solution and the same solution will be found each time that the model is trained with the same data.

However, in recent times, the popularity of the SVM has declined in favour of deep learning models. These are models which are characterised by the presence of many processing nodes assembled into a stacked topology. Classification is a black-box procedure for deep learning models as the abundance of nodes makes the internal workings difficult to comprehend. A subclass of deep learning model which has had notable success in the image recognition domain is the Convolutional Neural Network (CNN) (LeCun et al., 1989). The key to the CNN's success is the inclusion of sparsely connected layers named convolutional layers. These allow the model to extract spatial information from its inputs which is unavailable to many other models. The convolutional layers act as feature extractors before an input is passed to a neural network for prediction. The CNN has achieved state-of-the-art results on many benchmark datasets, for example, MNIST (LeCun et al., 1998a), ImageNet Large Scale Visual Recognition Challenge (Russakovsky et al., 2015) and Caltech 256 (Griffin et al., 2007).

Despite its success, the CNN, along with many other forms of deep learning, faces difficulties during training due to the irregularity of its error landscape. The large number of connection weights which are involved make it particularly susceptible to the complications caused by local minima. This work explores the idea of replacing the fully connected neural network, which a CNN typically uses for prediction, with a SVM. Thus the convolutional layers may be considered feature extractors before prediction using a SVM. The aim of this is to create a hybrid classifier which is able to utilise the spatial information of its input values while retaining the convex properties of SVM optimisation.

1.2 Contributions

The principle contributions of this thesis are two algorithms, named the L-CSVM and K-CSVM algorithms, for training the Convolutional SVM hybrid classifier.

The L-CSVM algorithm, described in Algorithm 4, is an iterative, two-step optimisation which may be used to learn a linear SVM with a convolutional feature extractor. It first learns the SVM primal weight vector by fixing the filter coefficients, then it learns the filter coefficients by fixing the weight vector. These steps are repeated until convergence. The novel aspect of the algorithm is the fact that it casts the filter update step as a SVM training problem. By doing this, the filter may be learned using an arbitrary SVM training algorithm. Thus, both of the training steps are convex with respect to the parameters.

The K-CSVM algorithm, presented in Algorithm 5, may be used to learn a kernelised SVM with a convolutional feature extractor. It does this by formulating the optimisation problem as a Multiple Kernel Learning (MKL) optimisation problem. The K-CSVM algorithm is based on the GMKL algorithm, proposed in (Varma and Babu, 2009), but it has been adapted for filter learning. To this authors knowledge, this is the first research in which convolution has been integrated into a kernelised SVM.

1.3 Thesis Organisation

The thesis continues in chapter 2 by providing background to the image recognition domain. The aim of the chapter is to establish notation and to provide the prerequisite information required to understand the concepts proposed in chapters 3 and 4. Chapter 3 presents the Linear Convolutional SVM. It provides details of previous research in this domain and then proposes a training algorithm. The algorithm is tested and analysed using the MNIST dataset. Chapter 4, titled Kernelised Convolutional SVMs, explores how a convolutional filter may be used to augment a kernelised SVM. It presents an algorithm for training and tests the model on the MNIST and CIFAR-10 datasets. The thesis is concluded in chapter 5.

Chapter 2

Background

This chapter begins with a description of supervised learning in section 2.1; the description is model agnostic, with the intention of establishing notation. Section 2.2 introduces the first model for supervised learning: the Linear Perceptron. Section 2.3 shows how the Linear Perceptron may be extended to learn a non-linear hyperplane by introducing the Multi-layer Perceptron. This is further adapted in section 2.4 where it is explained how convolution may be used to specialise the Multi-layer Perceptron for image recognition. Section 2.5 describes an alternative model for supervised learning named the Support Vector Machine. It shows how the primal formulation of a Support Vector Machine may be used to learn a linear hyperplane and then explains how a kernel function may be employed in the dual in order to learn a non-linear hyperplane. Section 2.6 completes the background with a description of Multiple Kernel Learning.

2.1 Supervised Learning

The high-level aim of supervised learning is to obtain a function which can take a set of properties and map them to a target output. The properties, named features, are collated into a vector, $\boldsymbol{x} \in \mathbb{R}^M$, and the target output, y, may be a real value or a class label. The tuple $(\boldsymbol{x}, y)$ forms an instance of some learning problem. The mapping function, denoted f, requires a configuration, $\boldsymbol{\theta}$, which determines the importance of items in $\boldsymbol{x}$ for approximating y. Thus $f : (\boldsymbol{x}, \boldsymbol{\theta}) \mapsto y$. It is the job of a supervised learner to tune $\boldsymbol{\theta}$ using data for which the mapping $x \mapsto y$ is already known. This process is named training and it will be described in more detail as this chapter progresses. Once training has been used to obtain a suitable $\boldsymbol{\theta}$, f may be used to make predictions about data for which y is unknown. The estimation is denoted $\hat{y}$ and it is not necessarily equal to y. Thus:

$$\hat{y} = f(\boldsymbol{x}, \boldsymbol{\theta}). \tag{2.1}$$

As an example of supervised learning, consider a town containing N houses. Three properties are known about each house: the floor space measured in square feet, the total number of floors and the number of bathrooms. Thus each $\boldsymbol{x_n}$ contains three elements where n ranges between 1 and $N + 1$. In this case, $M = 3$. The market value of some of the houses is known, but it is not

known for all. This acts as the target value y_n. In this scenario, the task of a supervised learner may be to find a relationship between the properties of a house and its market value, so that the houses without a known market value can be estimated.

To tune $\boldsymbol{\theta}$, a supervised learner examines a training dataset. This is a set where the true target is known for each of the instances. The training dataset is denoted $\boldsymbol{\mathcal{D}}$ and may be defined as:

$$\boldsymbol{\mathcal{D}} = \{(x_n, y_n)\}_{n=1}^{N}. \tag{2.2}$$

Learning is achieved by finding a $\boldsymbol{\theta}$ which minimises the overall difference between y and $\hat{y}$ for each of the instances in the training dataset. To measure the difference, a loss function $\ell(y, \hat{y})$ is used. The choice of ℓ depends on the type of supervised learner in use; for example, neural networks commonly use cross-entropy $\ell(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$ whereas support vector machines use the hinge loss function $\ell(\hat{y}, y) = \max(0, 1 - \hat{y} \cdot y)$. Loss functions will be described in further detail as new models are introduced. At this point, all that is required is the knowledge that ℓ exists and that it is a measure of the discrepancy between y and $\hat{y}$. The loss function for each of the items in $\boldsymbol{\mathcal{D}}$ may be summed into an error function E:

$$E(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}) = \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x_n}, \boldsymbol{\theta})). \tag{2.3}$$

The error function may alternatively be called the cost function by some literature. The optimum configuration, denoted $\boldsymbol{\theta}^*$, is the $\boldsymbol{\theta}$ which minimises E on the training dataset. It can be notated as:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, E(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}). \tag{2.4}$$

The technique to find $\boldsymbol{\theta}^*$ varies between models. For many it is an iterative process. Again, this will be described in greater detail later as the chapter introduces different models. This section was introduced with a statement that the high-level goal of a supervised learner is to find a function which can take a set of inputs and map them to a target output. This statement may now be made explicit. The goal of learning is to find a $\boldsymbol{\theta}^*$ which minimises E on the training dataset.

Yet, optimising E against the training dataset does not guarantee that the learner will perform well on unseen data. It is also important to tune the flexibility of the learner. A powerful learner will not work for all problems. On the contrary, a powerful learner will often learn poorly on straightforward learning tasks. In a phenomenon named overfitting, the learner may memorise the training data rather than learn which features are useful in prediction. The learner can become tightly coupled with the training data to the point that it learns unhelpful noise.

Overfitting can be tackled by reducing the flexibility of a supervised learner. The rationale is that a less flexible learner will not have the capacity to overfit. One method for this is to reduce the number of tunable parameters by reducing the number of items of $\boldsymbol{\theta}$, for example by reducing the number of hidden nodes in a neural network. Another option is to reduce the size of values within $\boldsymbol{\theta}$ by using a regularisation term. Denoted $E_{reg}(\boldsymbol{\theta})$, a regularisation term is an extension to the error function which takes into account the size of $\boldsymbol{\theta}$. Large $\boldsymbol{\theta}$s are penalised by artificially increasing the output of the error function. This has the effect of smoothing the error landscape and therefore makes it easier for the classifier to learn a $\boldsymbol{\theta}$ which can generalise to unseen data. $E(\boldsymbol{\theta})$ may be reformulated to the following to include a regularisation term:

$$E(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}) = E_{data}(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}) + \lambda E_{reg}(\boldsymbol{\theta}) \tag{2.5}$$

where $E_{data}(\boldsymbol{\theta})$ is the previous formulation of the error function as defined in (2.3) and λ is a free parameter to tune the influence of the regularisation term. An example regularisation term is the weight decay term, also known as the L_2^2 norm:

$$E_{reg}(\boldsymbol{\theta}) = \sum_{i}^{|\boldsymbol{\theta}|} \theta_i^2. \tag{2.6}$$

Weight decay regularisation is motivated by the idea that in many supervised learners the non-linearities which enable overfitting only become significant when the features are large. Thus, penalising the largest values helps to protect against overfitting.

To test how well a model generalises on unseen data, a second dataset, named the testing dataset, is required. The testing dataset is much like the training dataset in that y is known for each of its instances. The difference is that the testing dataset must not be used during training. It is exclusively for checking the effectiveness of a $\boldsymbol{\theta}$ found on the training dataset.

Supervised learning may be further broken down into two main sub-problems: regression and classification. In regression problems the target value y is a real value, whereas for classification problems it is a class name in the set T. At the start of this section, an example problem of calculating the market value of houses in a town was introduced. This is an example of regression. A classification task could be to decide on the type of house, for example, T may include *attached*, *semi-detached* and *detached*. Until now descriptions of the target value have been kept generic and formulas have worked for both regression and classification. The models developed in this thesis will be used for classification so this will be the focus moving forward.

2.2 Linear Perceptrons

The linear perceptron algorithm was an early model for supervised learning. It was first described by Rosenblatt (1957) and was developed in hardware. The linear perceptron aims to learn a linear function for separating binary class data. This is data for which y falls into one of two classes. The linear perceptron achieves classification by learning a hyperplane. Intuitively, this is a linear function which aims to divide the training dataset into homogeneous sets, where a set is defined as homogeneous when each of the instances within it have the same target class. This is not possible in all cases.

Figure 2.1 shows an example training set which contains six feature vectors split into two classes, positive and negative. Each instance in the set contains two properties, and the set has been plotted as a scatter with each axis representing a property. The dashed line shows a hyperplane which has succeeded in splitting the dataset. A hyperplane always has one dimension fewer than the number of dimensions of the feature space. As the feature space in figure 2.1 is two-dimensional, the hyperplane can be plotted as a one-dimensional line.

The hyperplane in a perceptron is determined by a vector of weights, $\boldsymbol{w}$, and a bias b. The weight vector controls the plane's slope and the bias controls its distance from the origin. Together, $\boldsymbol{w}$ and b form $\boldsymbol{\theta}$ for the linear perceptron. Many implementations combine $\boldsymbol{w}$ and b into a single vector by placing the bias in index 0 of the weight vector and assuming that the value of x_0 is 1 for each instance. Thus, $\boldsymbol{w} \in \mathbb{R}^{M+1}$. This scheme will be assumed for the rest of this section. Note that in this case, $\boldsymbol{\theta}$ and $\boldsymbol{w}$ have the same meaning; however, $\boldsymbol{\theta}$ is intended as

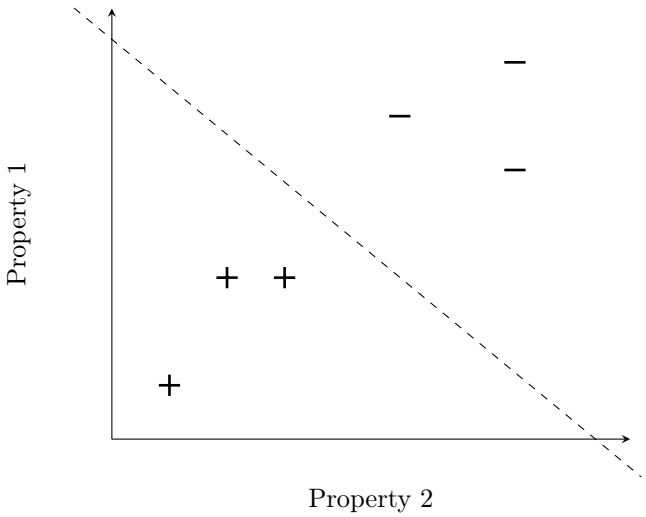

Figure 2.1: An example dataset containing instances split into two target classes. Each axis represents a feature. The dataset is homogeneously split by a linear hyperplane.

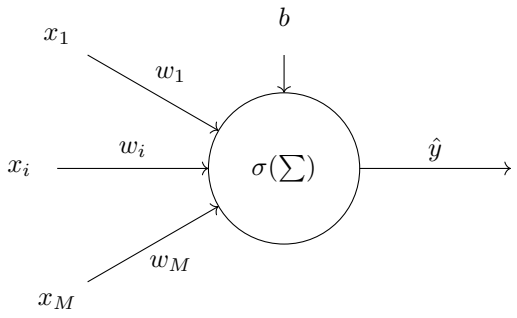

Figure 2.2: A visual depiction of a linear perceptron.

an abstract notion whereas $\boldsymbol{w}$ is concrete. Depicted visually in figure 2.2, the decision function of a linear perceptron can be defined as:

$$\hat{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma(\boldsymbol{w}^\intercal \boldsymbol{x}). \tag{2.7}$$

Hence σ is a function used to convert the output of $\boldsymbol{w}^\intercal \boldsymbol{x}$ into a prediction. The linear perceptron exists in numerous forms, many of which use a different σ. This thesis will look at the choice of σ for perceptrons trained for logistic regression and for large margin perceptrons. Each of these will become relevant further into this chapter.

Logistic regression is a form of linear perceptron which aims to learn a function which not only classifies instances but can also estimate the probability that an instance falls within a class. Logistic regression is a misleading name for the model as it is typically used for classification. Assuming that the two possible classes have been labelled as $+1$ and -1, the output of the perceptron is an estimate of the probability that an instance falls into the class defined as $+1$. Thus $\hat{y} = P(y = 1|\boldsymbol{x})$. If there is a high probability then $\hat{y}$ will be close to 1, if it is unlikely

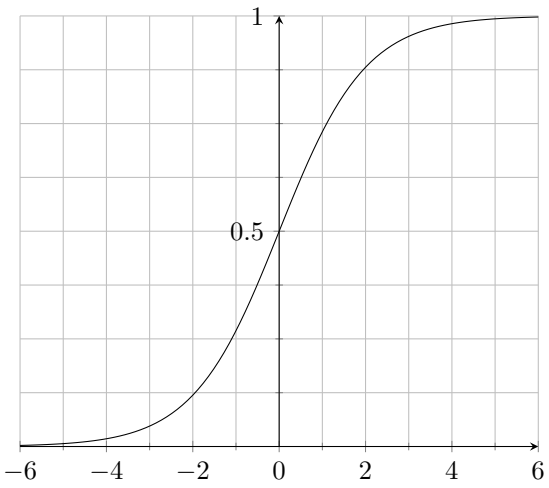

Figure 2.3: The logistic function.

then it will be close to 0. To achieve this, a perceptron can use the logistic function for the link function σ:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{2.8}$$

Figure 2.3 provides a visual depiction of the logistic function. To convert between this probability and a target class, post-processing of $\hat{y}$ is required. If $\hat{y}$ passes a threshold, then it may be classified as $+1$, else it is classified as -1. This threshold is often set at 0.5, though it does not have to be.

As explained in section 2.1, a supervised learner must undergo training before it can be used for prediction. This involves optimising $\boldsymbol{\theta}$ against the error function E, which in turn means minimising a loss function for each of the instances in the training dataset. The loss function used in logistic regression is log loss, also known as cross-entropy. It follows the form

$$\ell(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]. \tag{2.9}$$

The log loss function is largest when the perceptron is confident in a prediction which is incorrect. Even correct classifications incur some loss if they are not confident. If the correct class is predicted with a probability of one then $\ell(\hat{y}, y) = 0$.

The full error function for the training set is the sum of the log loss for each of the instances in $\boldsymbol{\mathcal{D}}$. An interesting property of this error function is that once minimised, the perceptron will be optimised for estimating probabilities rather than for classification. A classifier trained for logistic regression is willing to sacrifice its ability to classify correctly in order to improve its ability to estimate probability. Because of this, the logistic function may be less well suited to tasks which require only classification without needing an estimate of probability. Better suited is a large margin perceptron. Instead of the logistic function, large margin perceptrons use the identity function for σ:

$$\hat{y} = \sigma(a) = a. \tag{2.10}$$

Unlike the logistic function, the identity function does not restrict $\hat{y}$ to a value between 0 and 1. Thus it is not possible to calculate probability using this form of the large margin perceptron. As with the logistic function, post-processing of $\hat{y}$ is required to obtain a classification. As $\hat{y}$ is no

Algorithm 1 Gradient Descent

1: **Inputs** :$\boldsymbol{\mathcal{D}}, \eta$
2: **Initialise** :$\boldsymbol{w}$
3: **while** not converged **do**
4: $\quad \boldsymbol{w} := \boldsymbol{w} - \eta \frac{\partial}{\partial \boldsymbol{w}} E(\boldsymbol{w}, \boldsymbol{\mathcal{D}})$
5: **end while**

longer restricted, it is common practice to classify instances with positive $\hat{y}$ as $+1$ and instances with negative $\hat{y}$ as -1. Because the large margin perceptron does not predict a probability, the log loss function should not be used to measure loss. Instead, the large margin perceptron uses the hinge loss function:

$$\ell(\hat{y}, y) = \max(0, -\hat{y}, y). \tag{2.11}$$

Using hinge loss, correctly classified instances incur no loss, else loss is linear. An interesting property of hinge loss is that once combined into E, it creates a convex function with respect to $\boldsymbol{\theta}$. The significance of this will become apparent shortly.

Until now, little detail has been provided on the algorithm for minimising loss. It has been assumed that this algorithm exists, but no explanation or implementation has been provided. In reality, development of the minimisation algorithm is one of the main challenges faced when developing a model. For the perceptron, a number of learning algorithms exist. Perhaps the most commonly used of these is gradient descent. Gradient descent is an iterative algorithm which aims to minimise E by taking repeated steps in the direction of the negative gradient with respect to $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \frac{\partial}{\partial \boldsymbol{\theta}} E(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}), \tag{2.12}$$

where η is a learning rate, used to control how aggressively the perceptron updates $\boldsymbol{\theta}$ at each iteration. In effect, gradient descent 'walks' down the slope of the error function until a stopping criterion is met. At this point, it is said that the perceptron has converged. A common choice of stopping criterion is to halt training when the gradient becomes close to zero. The full gradient descent algorithm may be seen in algorithm 1.

Choosing η is an important step in the implementation of a perceptron. If the selected value is too large then the perceptron may not converge and if it is too small then convergence will take considerably longer than necessary. In general, it is better to pick η cautiously as, given enough time, a classifier which has been trained with an excessively low η will converge to a solution close to the solution obtained by a classifier trained with an optimum η. The same cannot be said for a classifier which has been trained with an η which is too large.

Gradient descent is only guaranteed to find the optimum $\boldsymbol{\theta}$ for convex functions. Non-convex functions prove difficult due to the fact that there may be local minima. These are points of the function which are the minimum of a local region of parameter space but are not necessarily the minimum globally. Gradient descent will continue descending a local minimum until it converges, though the solution will likely be non-optimal. As the hinge loss function is a convex, large margin perceptrons are unaffected by local minima.

The version of gradient descent described above is named batch gradient descent because the training dataset may be considered a batch of data. Alternatives to batch gradient descent are stochastic and mini-batch gradient descent. In stochastic gradient descent, only a single

instance is presented at a time, and the gradient calculated with respect to this instance only. The instances are presented in a random order. Stochastic gradient descent makes many small steps towards the minimum, rather than fewer large steps. It is more resistant to local minima than batch gradient descent since the direction of the gradient is more variable. However, the variability makes it more difficult to reach a gradient of zero in stochastic gradient descent. Mini-batch gradient descent is a compromise in which a configured number of instances are used to calculate the gradient per iteration. The configured number is named the batch size. The aim is to select a batch size which allows the algorithm to escape shallow local minima, while also being able to reach a gradient of near zero when a deeper local minimum is found. With a batch size of N, mini-batch gradient descent is equivalent to batch gradient descent, and with a batch size of one, it is equivalent to stochastic gradient descent. In practice, correctly tuned mini-batch gradient descent will often outperform batch or stochastic gradient descent in training time and accuracy. The batch size may be selected through cross-validation.

Gradient descent is not the only possible method of training a perceptron. Quasi-Newton methods are a class of optimisation methods which pursue the stationary point where gradient equals zero. They assume that the current gradient is the side of a quadratic bowl and use second order information to jump straight to the point which achieves the minimum of the bowl. This is usually an incorrect assumption, but in practice, the method often allows a classifier to take larger steps towards the minimum than in gradient descent. Quasi-Newton methods are particularly advantageous for optimising error landscapes with long valleys in which gradient descent performs poorly. BFGS (Fletcher, 1987) is one such implementation of a Quasi-Newton method. It is iterative by nature and uses an approximation of the inverse Hessian to reduce training time. Like gradient descent, Quasi-Newton methods are susceptible to traversing local minima.

2.3 Multi-layer Perceptrons

Figure 2.1 in the previous section introduces an example dataset, and shows how a hyperplane can be used to homogeneously separate the instance space. However, this example is simplistic and does not reflect real-life datasets. For most real datasets, it is impossible to separate all of the datapoints into homogeneous sections using a linear hyperplane. For example, consider the dataset in figure 2.4. There is no way that the points can be separated using a one-dimensional line. The dataset is not linearly-separable.

Due to its simplistic design, the linear perceptron may only ever model a linear decision boundary. While this is acceptable for some datasets, many require a more complex function. Resultantly, the perceptron often performs poorly compared to more flexible learners. Fortunately, an adjustment may be made to the linear perceptron which allows it to model non-linear data. This requires additional terminology. Figure 2.2 in the previous section depicts a linear perceptron visually and within this depiction is a processing unit which calculates the output of σ. This processing unit may be called a node.

To model a non-linear decision boundary, the perceptron may introduce additional nodes. These nodes must be stacked so that the output of at least two nodes are used as the input of another node. Moreover, the function used as σ for the additional nodes cannot be linear. The multilayer perceptron (MLP) (Werbos, 1982) is a model which requires that these additional

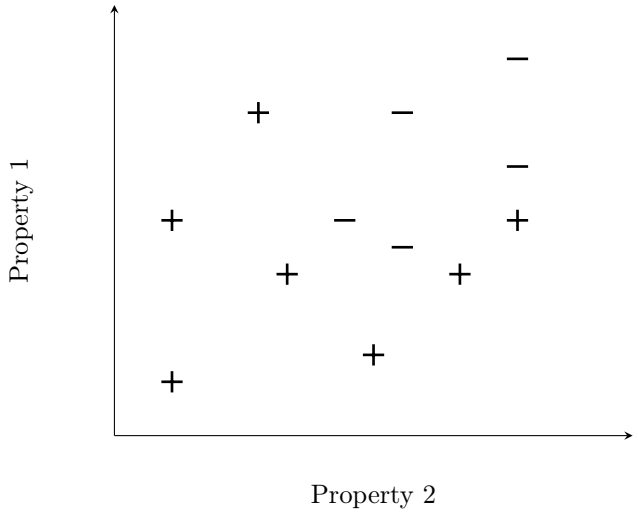

Figure 2.4: An example training dataset containing two classes which cannot be split homogeneously using a linear hyperplane.

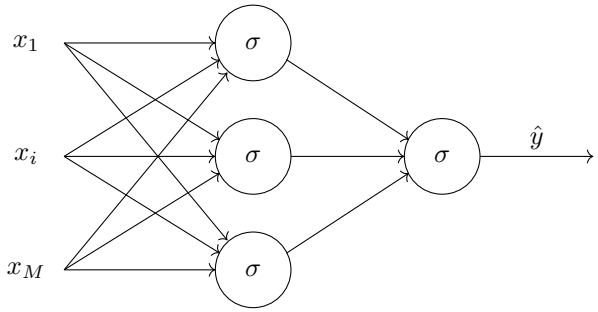

Figure 2.5: A multilayer perceptron with a single output node.

nodes are arranged into layers such that at least one layer of nodes exists between the input feature vector, and an output node used to calculate a final $\hat{y}$. This additional layer is named the hidden layer and the feature vector is named the input layer. Figure 2.5 shows a MLP with a single hidden layer. A MLP may have any number of hidden layers, however, for reasons which will be explained as the section progresses, there is rarely a benefit to having more than two. There is no restriction on the number of output nodes in a MLP, for example, a common scheme used for classification in multiclass problems is one-hot encoding. In one-hot encoding, an output node is added for each of the target classes in T. Each node represents a class, and after running the decision function, the output node with the highest score is the class which is chosen as the classification. A common implementation of MLP is the fully-connected MLP. This is an implementation which requires that each of the nodes within each layer, including the input layer, are connected to each of the nodes in the subsequent layer. This is depicted in figure 2.6. From this point forward, discussions will assume that a MLP is fully connected unless stated otherwise.

Like with the linear perceptron, feature vectors in a MLP are connected to nodes in the

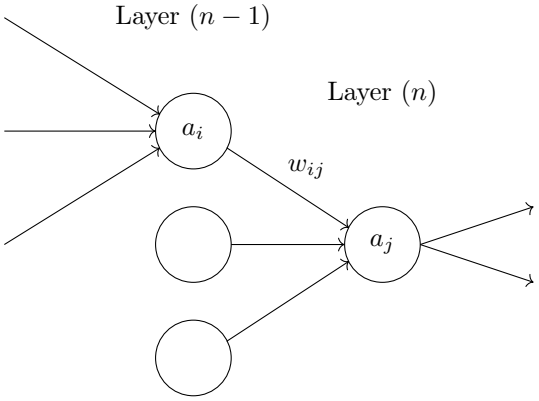

Figure 2.6: A visualisation of a MLP to display notation.

following layer by a set of weights, however, the number of connections is substantially greater. Describing the decision function of a MLP requires further notation. First, a label (n) is required to describe which layer of the network is in discussion. The input layer is described as layer (0) and the total number of layers will be notated as $\mathcal{L}$. The number of nodes within layer (n) will be notated $N^{(n)}$. Also required is a notation for describing the output of σ, known as the activation, for a node. For this, a will be used. The MLP requires $\mathcal{L}-1$ weight arrays such that $\boldsymbol{w}^{(n)}$ contains the weights between each of the nodes in layer (n) and layer $(n+1)$. Each array must now be a two-dimensional matrix rather than a vector and the weight between two nodes, i and j, will be given by $w_{ij}^{(n)}$. Putting this together, the activation of a node j in layer (n) is given by:

$$a_j^{(n)} = \sigma\left(\sum_{i=0}^{N^{(n-1)}} w_{ij}^{(n-1)} a_i^{(n-1)}\right). \tag{2.13}$$

It is assumed that the bias is stored in connection w_{0j} and that and that the activation $a_0^{(n-1)} = 1$. The score $\hat{y}$ is found by recursively calculating activations, starting from nodes in the input layer and finishing at the output layer. Since the output layer may contain multiple nodes, y and $\hat{y}$ may be vectors. The function σ may be any differentiable function, with an additional restriction for nodes in the hidden layer, that the function must be non-linear. Therefore the logistic function is an acceptable choice for hidden nodes, however, the identity function does not meet the criteria for hidden layers. A common σ used by MLPs is the hyperbolic tangent:

$$\sigma(a) = \tanh(a). \tag{2.14}$$

When plotted, the hyperbolic tangent has a similar shape to the logistic function, however the function ranges between values of 1 and -1 rather than 0 and 1. In many cases, the symmetry around the origin helps with the speed of convergence, as explained in (LeCun et al., 1998b). The derivative of the hyperbolic tangent may be calculated as $\frac{\partial}{\partial a}\tanh(a) = 1 - \tanh(a)^2$. The hyperbolic tangent cannot be used in the output layer if a probability estimation is required, however, there is no restriction for σ to be the same function in each layer. It is common to see the hyperbolic tangent used in the hidden layers, and the logistic function used in the output layer in order to estimate probabilities.

The loss function in a MLP depends directly on the output of nodes in the final layer, although this will depend on nodes in the hidden layers. As previously stated, the output layer may contain

multiple nodes, and so y and $\hat{y}$ are vectors rather than single values. The loss function must be updated accordingly. Although more complex schemes exist, a simple yet effective technique is to sum the loss of each of the nodes in the output layer. For example, the MLP variant of log loss for classification may be defined:

$$\ell(\hat{y}, y) = \sum_{j}^{N^{(\mathcal{L})}} (y_j \log \hat{y}_j + (1 - y_j) \log(1 - \hat{y}_j)). \tag{2.15}$$

A MLP may be trained using a form of gradient descent named backpropagation. As with the decision algorithm, this is a recursive algorithm which updates layers one at a time. It requires that the gradient of E is calculated with respect to each of the layers in the MLP using the chain rule. Details of how this is achieved may be found in (Werbos, 1974). Once the gradient has been calculated with respect to each of the layers, the standard gradient descent algorithm defined in 2.12 may be used to update the weight vector within each layer. As with gradient descent, backpropagation may take many iterations before convergence. Methods like momentum (Polyak, 1964) and adaptive learning rates (Schaul et al., 2013) aim to reduce the required number of iterations.

As previously described, adding a hidden layer of nodes allows a MLP to model non-linear decision functions. In fact, it was proved in (Cybenko, 1989) that it is possible to approximate any decision region to arbitrary accuracy using a single hidden layer and any continuous sigmoidal σ. This makes the MLP a flexible model for learning complex functions. However, as a result of this, the MLP is susceptible to overfitting. Section 2.1 described two methods to avoid overfitting. The first of these was to reduce the number of tunable parameters in $\boldsymbol{\theta}$ and the second was to add regularisation to the model. Both of these techniques are applicable to the MLP. In the context of a MLP, $\boldsymbol{\theta}$ can be thought of as the combination of each of the weight vectors. The number of required weights depends on the number of connections between nodes. The more connections, the more flexible a MLP is, and the more likely it is to overfit. The number of connections is controlled by the number of nodes, and by the complexity of the arrangement of these nodes. In particular, the number of hidden layers has a profound effect on the complexity. Adding an extra layer of nodes will usually increase the number of connections by an amount greater than would occur if the same number of nodes were added to an existing layer. Choosing the layout of a MLP, also called the topology, is arguably the main challenge in successful implementation. The optimum layout is different between problems and depends on the intrinsic complexity of the dataset. Cross-validation can be used to select the number of nodes and layers.

The second technique to avoid overfitting is to add a regularisation term. This is an artificial adjustment to the error function which aims to reduce the size of items within $\boldsymbol{\theta}$. For MLPs, this is often achieved by using weight decay regularisation. As touched upon in section 2.1, weight decay regularisation follows the form:

$$E_{reg}(\boldsymbol{\theta}) = \sum_{i=1} \theta_i^2. \tag{2.16}$$

The value of this term, multiplied by a weighting, is added to the total of the error function so that learning will tend towards weight vectors with low values. This helps to guard a MLP from overfitting because the non-linearities which cause overfitting are more significant when features are large.

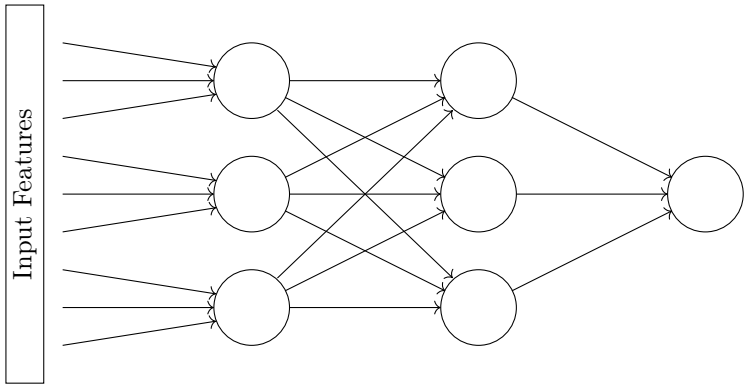

Figure 2.7: A neural network for which the first layer is sparsely connected. The hidden layers remain fully connected.

2.4 Convolutional Neural Networks

The MLP is part of a wider class of model named the Neural Network. Like the MLP, all Neural Networks consists of nodes which are connected by weights, however, there is no restriction for layers to be fully connected. The Convolutional Neural Network (CNN) (LeCun et al., 1989) (LeCun et al., 2015) is a type of Neural Network which specialises in tasks that exhibit local correlation of features, for example, image recognition.

These are tasks for which features within close spatial proximity to one another can form recognisable value patterns. In such tasks, more information may be gained by using the value of a feature relative to its neighbours than can be gained from its value alone. The CNN utilises this additional information with the addition of sparsely connected layers, named convolutional layers, where a layer is considered sparse if it has nodes which are only connected to a local subset of nodes in the following layer. In addition, connections within a convolutional layer are reused at multiple points along the input. To achieve this, the convolutional weights, known collectively as the filter and denoted by $\boldsymbol{f}$, 'slide' along the input, considering only a subset of features at any one time. The number of features which the filter slides over is known as the stride. CNNs still make use of a fully connected network for classification, however, the convolutional layers stand between the input and the fully connected network, as shown in figure 2.7. For a CNN with a single convolutional layer that is the first layer in the network, and for which the stride is one, the number of nodes in the first fully connected layer must equal $|\boldsymbol{x}| - |\boldsymbol{f}| + 1$. For this network, the input, u, to a particular node, i, in the first fully connected layer is given by:

$$u_i^{(1)} = \sum_{j=0}^{|\boldsymbol{f}|} x_{i+j} f_j. \tag{2.17}$$

When performed at each possible index, this operation is named convolution and is denoted by the $*$ operator. The output of convolution is called the feature map and is symbolised as $\boldsymbol{u}$:

$$\boldsymbol{x} * \boldsymbol{f} = \boldsymbol{u}. \tag{2.18}$$

Figure 2.8: An example convolution of an image with an edge detecting kernel. Left shows the original image and right shows the feature map after convolution with a filter containing the values: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$.

CNNs have been shown to be especially effective at image classification tasks, with state-of-the-art results in many benchmark datasets, for example, MNIST (LeCun et al., 1998a), ImageNet Large Scale Visual Recognition Challenge (Russakovsky et al., 2015) and Caltech 256 (Griffin et al., 2007). In these datasets, the input features are the pixel values of each image. Instead of storing features in a vector, they are stored in a tensor which has a rank of two or three, with the third dimension containing colour values if they exist. The filter is usually two-dimensional and during convolution it advances horizontally and vertically. Typically, a different filter is trained for each of the colour channels. The aim of convolution is to extract the defining features of an image, for example, a particular texture or the edges of an object. The movement of the filter allows a CNN to find such features anywhere within an image. It improves the shift invariance of the model which allows it to generalise more effectively to unseen data. As a positive byproduct of sharing filter coefficients, the number of connections in a CNN is low when compared to fully connected networks of an equivalent size. Figure 2.8 shows an example of convolution on a real image.

The description given so far has used a single filter in the convolutional layer, however, many real-world implementations, including the three given above, use multiple stacked filters per layer. Each filter can be trained to highlight a different defining feature. When an input is passed to the convolutional layer, each of the filters is convolved with the input. If the layer contains k filters, then the output will contain k feature maps. As well as having multiple filters per layer, a network may have multiple convolutional layers. In this case, each of the filters in convolutional layer $l^{(n)}$ are applied to each of the feature maps which were created by layer $l^{(n-1)}$. Given a network with two convolutional layers which are the first two layers of the network, the total number of feature maps after the second layer of convolution is given by $k^{(0)} \times k^{(1)}$. Assuming that each of the filters within $l^{(n)}$ is of equal cardinality and that this cardinality is denoted $|\boldsymbol{f^{(n)}}|$, the total number of tunable filter weights is given by $k^{(0)} \times |\boldsymbol{f^{(0)}}| + k^{(0)} \times k^{(1)} \times |\boldsymbol{f^{(1)}}|$.

As the number of filters per layer increases, the total size of a network can quickly become unmanageable. For this reason, many networks include subsampling layers. These are layers which forcibly reduce the size of feature maps by tactically removing non-essential information.

A common method to achieve this is max-pooling. Max-pooling divides a feature map into partitions and then carries forward only the greatest value from each partition into the proceeding layer. The partitions are named pooling windows and for image recognition tasks, each pooling window has a width and a height. Ordinarily, the widths and heights are uniform throughout the image and are referred to as the pooling width, p_w, and pooling height, p_h. Max-pooling reduces the size of feature maps by a factor of $p_w \times p_h$ assuming that the width and height of a feature map are divisible by p_w and p_h respectively. Although the same number of feature maps are passed to the fully connected network, the reduction in the size of feature maps substantially decreases the number of nodes required in the fully connected network. As well as decreasing the number of tunable parameters, max-pooling provides a degree of shift invariance as the output of a pooling window will be consistent regardless of where a feature lies within the pooling window. During backpropagation, the gradient only propagates back through the maximum feature within each pooling window. An alternative to max-pooling is average-pooling, whereby the output is the average over all of the nodes in the pooling window.

The idea of using sliding filters in a neural network is first described in (LeCun et al., 1989), although the networks are described as constrained rather than convolutional, and no subsampling method is employed. It was nine years later in (LeCun et al., 1998a) that CNNs close to their current form are described. The network in (LeCun et al., 1998a) combines the ideas of local receptive fields, shared filter weights and subsampling. It applies the network to an image classification problem, named MNIST, which contains 70,000 images of handwritten numeric characters split into ten target classes. The data is split into a training set, containing 60,000 images, and a test set which contains the remaining 10,000. Each image is of size 28x28, and the contents are centred by mass. The CNN developed in (LeCun et al., 1998a), dubbed LeNet-5, contains seven layers excluding the input layer. Of these, two are convolutional layers, two are subsampling layers and the remaining three form a fully connected network. The convolutional and subsampling layers alternate before feeding into the fully connected network. Using this topology, LeNet-5 is able to achieve a generalisation error of 0.95%, compared to 2.95% obtained by the highest performing fully-connected network described in the paper. In the time since its publication in 1998, MNIST has become a popular dataset for benchmarking models. At this time of writing, a variant of the CNN still yields the greatest accuracy on the dataset, achieving 0.21% generalisation error (Wan et al., 2013a).

2.5 Support Vector Machines

The Support Vector Machine (SVM) (Boser et al., 1992) (Cristianini and Shawe-Taylor, 2000) is a supervised model for separating binary data-sets where $y \in \{-1, 1\}$. It is an influential and prevalent model with many parallels to the linear perceptron. Like the perceptron, a SVM aims to find a linear hyperplane for classification. It shares the same decision function:

$$\hat{y} = \boldsymbol{w}^\intercal \boldsymbol{x} + b, \tag{2.19}$$

however, the method used for training differs. During training, SVMs focus on separating the data points which are nearest to the decision boundary. These are named the support vectors. The rationale for this is that optimal separation of the most difficult datapoints is equivalent to the optimal separation of the entire dataset. Figure 2.9 shows an example of a hyperplane which achieves maximum separation of the support vectors. Above and below are the planes at which $y(\boldsymbol{w}^\intercal \boldsymbol{x} + b) = 1$. They are known as the boundaries, and it is on these that the support vectors

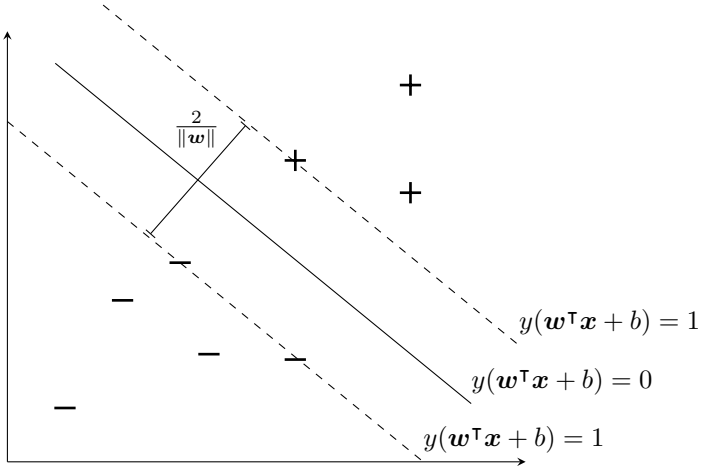

Figure 2.9: A dataset which has been split homogeneously by a SVM hyperplane. The hyperplane achieves maximum separation of most difficult instances within the two target classes. Above and below are the planes, named the boundaries, at which $y(\boldsymbol{w}^\intercal \boldsymbol{x} + b) = 1$.

fall. This is not by chance, it is a requirement of $\boldsymbol{w}$. The distance between the boundaries is known as the margin and is given by $\frac{2}{\|\boldsymbol{w}\|}$. Maximising the margin will maximise the distance between support vectors. Therefore, the optimum $\boldsymbol{w}$ is the smallest $\boldsymbol{w}$ which can satisfy the requirements. On linearly separable data, this can be achieved by minimising the hard-margin error function:

$$E = \|\boldsymbol{w}\|^2 \quad \text{subject to} \quad y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + b) \geq 1 \quad \forall i, \tag{2.20}$$

where $\{\boldsymbol{x}_i, y_i\}$ are the training data.

When the data is not linearly separable, the hard-margin SVM error function will fail to converge as there is no way to satisfy all of the constraints. Instead, the soft-margin error function may be used. The soft-margin function introduces a leniency to the constraints with the addition of slack variables $\boldsymbol{\xi}$. The slack variables alter the constraints such that $y(\boldsymbol{w}^\intercal \boldsymbol{x}_i + b)$ may equal $1 - \xi_i$, rather than 1. If $0 < \xi_i < 1$, then $\boldsymbol{x}_i$ must fall within the margin but still falls on the correct side of the decision boundary. If $\xi_i > 1$ then the $\boldsymbol{x}_i$ falls on the incorrect side of the decision boundary. The introduction of $\boldsymbol{\xi}$ prevents outliers from controlling the decision boundary, as illustrated in figure 2.10. The soft-margin error function can be defined:

$$E = \|\boldsymbol{w}\|^2 + C\sum_{j}^{N} \xi_j \quad \text{subject to} \quad y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + b) \geq 1 - \xi_i \quad \forall i, \tag{2.21}$$

where C is a regularisation parameter to control the extent to which the decision boundary is constrained. A small C will allow the SVM to easily ignore constraints, resulting in a wide margin, but with many support vectors violating the boundaries. A large C heavily penalises the cost of support vectors within the boundaries, resulting in a narrow margin. With very large C, the soft-margin error function is equivalent to the hard-margin function. Both the hard and soft margin error functions are convex and a number of methods exist to optimise these. Perhaps the simplest of these conceptually is to treat the error function as a constrained quadratic programming problem and to solve this directly using Lagrange multipliers, or using an algorithm such as Sequential Minimal Optimisation (SMO) (Platt, 1998). Alternatively, the

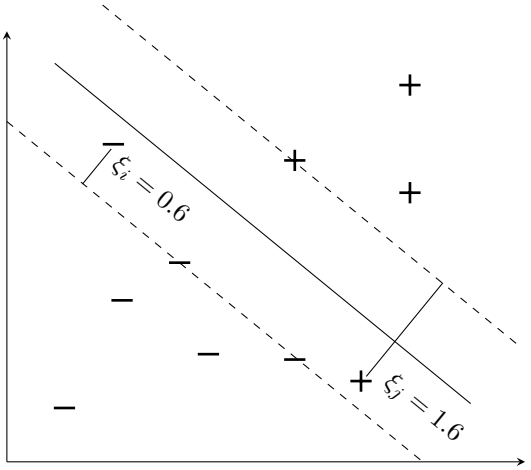

Figure 2.10: An example hyperplane which has been learned by optimising the soft margin error function. Two of the instances fall within the margin because the slack variables relax the SVM constraints.

optimisation problem may be reformulated into an unconstrained version:

$$E = \|\boldsymbol{w}\|^2 + C \sum_{i}^{N} \max(0, 1 - y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + b)). \tag{2.22}$$

The unconstrained optimisation is equivalent to the large-margin perceptron optimisation problem, but with the addition of L_2^2 regularisation on $\boldsymbol{w}$. Therefore, learning may be achieved by using gradient based optimisation approaches, for example, the Pegasos algorithm (Shalev-Shwartz et al., 2011).

The soft-margin error function allows a SVM to converge on data which is not linearly separable but it does not increase the expressive power of the classifier. The formulation described is only flexible enough to describe a linear decision boundary. However, with adjustments, it is possible for the SVM to describe arbitrarily complex boundaries. Moreover, it is possible to do this while retaining the aim of learning a linear hyperplane. Instead of learning a more complex shape than a plane, the SVM opts to alter the input features so that a plane is adequate for classification. It does this by using a function named a feature mapping, denoted by $\boldsymbol{\phi}$. The aim of a $\boldsymbol{\phi}$ is to map a feature vector into a new vector with a greater number of dimensions:

$$\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}\left(\begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ \vdots \\ x_{M'} \end{bmatrix}, \tag{2.23}$$

where $M' \geq M$. M' may even be infinite. The hope is that in the higher dimensional space, the data can be more easily separated. The feature mapping may be considered a preprocessing step before classification by a linear hyperplane. Thus, the decision function is linear with respect to $\boldsymbol{\phi}(\boldsymbol{x})$ but non-linear with respect to $\boldsymbol{x}$. Figure 2.11 shows an example feature mapping which maps between $\mathbb{R}^2$ and $\mathbb{R}^3$. In $\mathbb{R}^2$ the datapoints are inseparable, however, by mapping the inputs into $\mathbb{R}^3$, it is possible to separate the dataset using a linear hyperplane. When using a feature

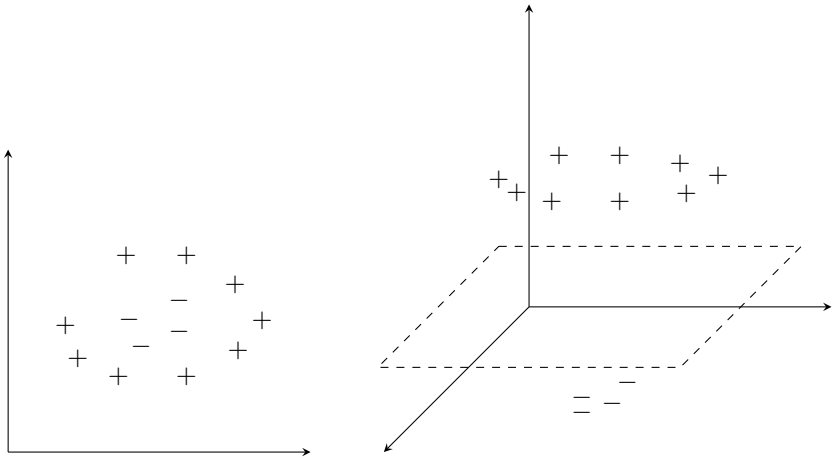

Figure 2.11: Applying an example feature mapping to a dataset containing two-dimensional instances. In two dimensions the dataset is not linearly separable but in three dimensions it is.

mapping, the decision function becomes:

$$\hat{y} = \boldsymbol{w}^\intercal \boldsymbol{\phi}(\boldsymbol{x}) + b. \tag{2.24}$$

Likewise, the error functions seen in equations (2.20), (2.21) and (2.22) remain the same, but with all occurrences of $\boldsymbol{x}$ replaced by $\boldsymbol{\phi}(\boldsymbol{x})$.

While feature mappings provide the means to classify non-linear data using a SVM, in practice, they can prove too computationally expensive for realistic use. For some datasets, the number of output dimensions required by the mapping function may be infinite. So, instead of explicitly working out the high-dimensional space, many SVM formulations choose to take a shortcut. The SVM formulation which is described above is called the primal form. An alternative is the dual form. In the dual form, $\boldsymbol{w}$ does not exist as a concrete variable. Instead, it is calculated as a combination of the data such that $\boldsymbol{w} = \sum_i^N \alpha_i y_i \boldsymbol{\phi}(\boldsymbol{x}i)$, where α_i is a scalar which is learned during training. For training instances which are not considered to be support vectors, α_i will be close to zero. The dual formulation of the decision function may initially be defined as:

$$\hat{y} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)^\intercal \boldsymbol{\phi}(\boldsymbol{x}) + b, \tag{2.25}$$

At this point the dual and primal forms are equivalent, and the issues around computational complexity are still present. The power of the dual form becomes apparent with the addition of a kernel function. Proposed in (Boser et al., 1992), a kernel function, $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, can be used to calculate the output of $\boldsymbol{\phi}(\boldsymbol{x}_i)^\intercal \boldsymbol{\phi}(\boldsymbol{x}_j)$, while avoiding the need to calculate $\boldsymbol{\phi}(\boldsymbol{x}_i)$ or $\boldsymbol{\phi}(\boldsymbol{x}_j)$ directly. Computationally, this is considerably cheaper in most cases. Using Mercer's theorem (Mer, 1909), it can be shown that all positive semi-definite kernels are equivalent to an inner product after mapping with some $\boldsymbol{\phi}$. With the addition of a kernel function, the decision function becomes:

$$\hat{y} = \sum_{i=1}^{n} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b, \tag{2.26}$$

The simplest kernel function is the linear kernel: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^\intercal \boldsymbol{x}_j$. In this case, no feature mapping is performed and $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{x}$. More common choices are the polynomial kernel:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^\intercal \boldsymbol{x}_j + 1)^d \tag{2.27}$$

and the RBF kernel:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2) \tag{2.28}$$

where d and γ are parameters which control the complexity of the feature mapping. Even with the addition of a kernel function, the SVM learning algorithm remains convex. Updating the error function for the dual may be achieved by following a similar process to that used to obtain (2.26). For the regularisation term:

$$\begin{aligned} \|\boldsymbol{w}\|^2 &= \left(\sum_i^N \alpha_i y_i \boldsymbol{\phi}(\boldsymbol{x}_i)\right)^\intercal \left(\sum_j^N \alpha_j y_j \boldsymbol{\phi}(\boldsymbol{x}_j)\right) \\ &= \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j). \end{aligned} \tag{2.29}$$

Inserting this and the representation of the decision function into the primal error function seen in (2.22) gives:

$$E = \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) + C \sum_i^N \max(0, 1 - y_i(\sum_j^N \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) + b). \tag{2.30}$$

However, this is rarely used as the optimisation problem. Instead, it is reframed into an equivalent maximisation problem over $\boldsymbol{\alpha}$ which is easier to solve:

$$\sum_k^N \alpha_k - \frac{1}{2} \sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \quad \text{subject to} \quad 0 \leq \alpha_i \leq C \quad \forall i \quad \text{and} \quad \sum_i^N \alpha_i y_i = 0. \tag{2.31}$$

This is possible due to the representation theorem (Schölkopf et al., 2001) and the fact that the minimisation problem is strongly convex. The solution which solves the dual optimisation problem will also solve the primal problem as long as the Karush-Kuhn-Tucker (KKT) conditions are satisfied (Kuhn and Tucker, 1951).

The SVM implementations discussed so far have been exclusively for binary classification. Although SVM formulations exist which natively support multi-class classification, a more common method for approaching multiclass problems is to use a scheme such as one-versus-one or one-versus-all. In such schemes, no adaption is required to the SVM formulation discussed above. Instead, multiclass is achieved by training multiple classifiers, each designed to classify a particular target or a pair of targets. In a one-versus-all scheme, $|T|$ classifiers are learned, where T is the set of target values. Each classifier is trained with data which has been relabelled so that all instances of one class have $y = +1$ and any instance which is not of this class has $y = -1$. To decide the final classification of an instance, $\hat{y}$ is calculated for each of the classifiers, and the class represented by the classifier with the greatest $\hat{y}$ is selected as the output. An alternative scheme is one-versus-one. In this scheme, a classifier is trained for each pair of target classes, resulting in $\frac{|T|(|T|-1)}{2}$ classifiers in total. The final classification of an instance is obtained by making a class prediction using each of the classifiers and then selecting the class with the greatest number of votes. Although one-versus-one requires the creation and training of many more classifiers than one-versus-all, the overall training time is often lower as fewer instances are used to train each classifier.

2.6 Multiple Kernel Learning

Multiple Kernel Learning (MKL) is an extension to the SVM in which multiple kernel functions are combined into a hybrid. A number of MKL formulations exist, however, all share the rationale that the most effective kernel is unknown and should therefore be learned as part of training. A common form of MKL is where the kernels and their parameters have been predefined and the aim of learning is to decide how they should be combined. The combination function is denoted by $\mathcal{C}$, and it requires a set of weights $\boldsymbol{g}$:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{g}) = \mathcal{C}(\boldsymbol{g}, \{k_m(\boldsymbol{x}_i, \boldsymbol{x}_j)\}_{m=1}^{P}), \tag{2.32}$$

where P represents the number of predefined kernels. An example of this formulation is the weighted sum:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{g}) = \sum_{m=1}^{P} g_m k_m(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{2.33}$$

This is valid since the sum of Mercer kernels is itself a Mercer kernel. Another category of MKL is where the combination function is already known, and instead, the aim is to optimise the parameters which are integrated into the kernels. For example, learning the optimum γ in the RBF kernel or the degree of a polynomial kernel. Here, the kernel functions themselves require the weights stored in $\boldsymbol{g}$. Assuming that each kernel function requires a single weight:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{g}) = \mathcal{C}(\{k_m(\boldsymbol{x}_i, \boldsymbol{x}_j, g_m)\}_{m=1}^{P}). \tag{2.34}$$

This formulation is integral to the classifier developed later in this work. The error function of an MKL problem may be structured in many ways, for example the formulation of (Rakotomamonjy et al., 2008) may be used for learning the weighted sum, whereas (Chapelle et al., 2002) may be used for learning the internal kernel parameters. A general formulation which flexibly describes a range of MKL problems, including the two mentioned above, is Generalised MKL (GMKL). Described in (Varma and Babu, 2009), GMKL aims to optimise the following function:

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{g}} \quad & \frac{1}{2}\boldsymbol{w}^\intercal \boldsymbol{w} + \sum_i l(y_i, \mathcal{C}(\boldsymbol{x}_i)) + r(\boldsymbol{g}) \\ \text{subject to} \quad & \boldsymbol{g} \geq 0 \end{aligned} \tag{2.35}$$

where l is a loss function, for example, hinge loss, and r is a regularisation function which may be any function which is differentiable with respect to $\boldsymbol{g}$. This error function is equivalent to the SVM primal error function but with additional regularisation of the kernel parameters.

GMKL minimises this function by using an iterative two-step optimisation. First $\boldsymbol{g}$ is fixed while an arbitrary SVM optimisation algorithm is used to learn $\boldsymbol{w}$, then $\boldsymbol{g}$ is learned using gradient descent while $\boldsymbol{w}$ is fixed. GMKL aims to minimise the primal function overall but the internal SVM optimiser maximises the dual function with the intention of finding the optimum $\boldsymbol{\alpha}$, denoted $\boldsymbol{\alpha}^*$. After training of the internal SVM is complete, $\boldsymbol{g}$ is optimised by holding $\boldsymbol{\alpha}^*$ constant and using gradient descent. These two steps repeat until convergence. To optimise $\boldsymbol{g}$ using gradient descent, the derivative of the error function $E(\boldsymbol{w}, b, \boldsymbol{g})$, defined in equation (2.35), is required. Varma and Babu (2009) show that this gradient may be calculated using:

$$\frac{\partial E}{\partial g_k} = \frac{\partial r}{\partial g_k} - \frac{1}{2}\boldsymbol{\alpha}^{*\intercal}\frac{\partial \boldsymbol{H}}{\partial g_k}\boldsymbol{\alpha}^*, \tag{2.36}$$

Algorithm 2 Generalised MKL (Varma and Babu, 2009)

1: $n \leftarrow 0$
2: Initialize g^0 randomly
3: **repeat**
4: $K \leftarrow k(\boldsymbol{g}^n)$
5: $\boldsymbol{\alpha}^* \leftarrow$ solve SVM using K.
6: $g_k^{n+1} \leftarrow g_k^n - s^n(\frac{\partial r}{\partial g_k} - \frac{1}{2}\boldsymbol{\alpha}^{*\intercal}\frac{\partial H}{\partial g_k}\boldsymbol{\alpha}^*)$
7: Project $\boldsymbol{g}^{n+1}$ onto the feasible set if any constraints are violated.
8: $n \leftarrow n + 1$
9: **until** Converged

where $\boldsymbol{H}$ is the matrix generated by calculating $y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{g})$ for all combinations of i and j. The requirement of a two-step optimisation stems from the fact that this derivate requires values for $\boldsymbol{\alpha}^*$. After the update of $\boldsymbol{g}$, the values found for $\boldsymbol{\alpha}^*$ are no longer optimum and need to be recalculated. In turn, this causes $\boldsymbol{g}$ to become outdated. Therefore GMKL requires a number of iterations before convergence. As the iteration number increases, the size of the updates to $\boldsymbol{\alpha}$ and $\boldsymbol{g}$ diminish. A step size, s, adds further control to the size of the update of $\boldsymbol{g}$. The full GMKL algorithm is presented in Algorithm 2. Unfortunately, optimisation of $\boldsymbol{g}$ is not generally a convex problem, however, the learning of $\boldsymbol{w}$ remains convex.

2.7 Conclusion

This chapter has provided the prerequisite information which will be required in chapters 3 and 4. It began with a model agnostic description of machine learning and then provided information on Linear Perceptrons, Multilayer-Perceptrons, Convolutional Neural Networks, Suppor Vector Machines and Multiple Kernel Learning.

To summarise, the linear perceptron is an early model for supervised learning. It is arguably the quintessential model for achieving binary classification. Despite this, the linear perceptron performs poorly on many many real datasets due to the lack of flexibility caused by its inherent linearity. However, it is possible to achieve non-linearity by combining a number of perceptrons into a multi-layer perceptron. The convolutional neural network (CNN) is a further development to the multi-layer perceptron which uses sparse layers to utilise spatial information from its inputs and weight sharing to provide leniency regarding the location of this spatial information. This is useful on many tasks for which the location of features is important, for example, image classification. Resultantly, the CNN has achieved state-of-the-art results on many benchmark datasets.

An alternative model for classification is the support vector machine (SVM). The linear SVM is similar to the linear perceptron, even sharing the same decision function. However, the regularisation which is used during the training of a SVM is designed so that the hyperplane achieves optimum separation of the most difficult datapoints, named the support vectors. It is possible for a SVM to achieve non-linearity by using a kernel function to calculate the inner products of its inputs in a higher dimensional space. Optimisation remains a convex problem by using this method. Multiple kernel learning builds on this by providing an approach which may be used to combine kernel functions into a hybrid. Alternatively, it can be used to optimise

internal kernel parameters, for example the degree of a polynomial kernel.

The CNN and SVM are essential for understanding the models which are proposed in chapters 3 and 4. In addition, chapter 4 requires knowledge of multiple kernel learning.

Chapter 3

Linear Convolutional SVMs

Among others, chapter 2 describes two models: the convolutional neural network and the support vector machine. The CNN excels in tasks which exhibit spatially local correlation of features, for example, image classification, however, it suffers from multiple local minima due to the MLP which it uses in the final layers. This thesis presents a method for replacing the MLP in the final layer with a SVM. The convolutional layers may be considered feature extractors before classification with the SVM. The resulting model will be named the Convolutional Support Vector Machine (CSVM). This chapter will describe an approach which may be used in the primal to augment a linear SVM with convolutional layers. The following chapter will describe an alternative approach which may be used in the dual with a kernelised SVM.

The idea of combining support vector machines and convolutional neural networks into a hybrid classifier has been investigated in previous work. In (Huang and LeCun, 2006) a CNN was trained was on the NORB dataset (LeCun et al., 2004) and once training was complete, the final layers were replaced with a SVM. The NORB dataset is an image recognition benchmark dataset containing pictures of children's toys from many angles. It can be further broken down into two datasets, the normalised-uniform set and the jittered-cluttered set. The normalised-uniform set contains centred images of similar size and on a uniform background, whereas the jittered-cluttered set contains the same images but each has been deliberately perturbed and has been placed on a cluttered background. The hybrid classifier performs well on both datasets: On the jittered-cluttered dataset the hybrid classifier achieves 5.9% error rate, compared to 7.2% for a CNN and 43.3% for a regular SVM. It also achieves 5.9% error rate on the normalised-uniform set, compared to 6.2% for a CNN and 11.6% for a SVM. However, the fact that the fully-connected network is swapped out for a SVM after training, means that the convolutional filter weights are optimised for use with a classifier which is not used at classification time.

Tang (2013) builds upon this approach by proposing a method for training a deep convolutional network in conjunction with SVM parameters. This is achieved by replacing the output layer with a set of SVMs, one per target output. Thus, as in (Huang and LeCun, 2006), the inputs to the SVM are the penultimate activations of the neural network. At prediction time, an instance is classified as the target represented by the SVM which gives the greatest output. Each of the SVM weight vectors is learned using gradient descent of the primal loss function, and connection weights in the deep network are learned by using backpropagation from this

$$\text{CMT}(\boldsymbol{x}) = \begin{bmatrix} x_1 & x_2 & x_D \\ x_2 & x_3 & x_{D+1} \\ x_3 & x_4 & x_{D+2} \\ \vdots & \vdots & \vdots \\ x_{(M-D+1)} & x_{(M-D+2)} & x_M \end{bmatrix}$$

Figure 3.1: An example of a convolutional matrix to achieve one-dimensional convolution. $M = |\boldsymbol{x}|$ and $D = |\boldsymbol{f}|$. In this example $D = 3$.

initial gradient. The initial gradient is calculated with respect to the penultimate activations of the neural network. As the primal loss is used to calculate gradient, the model is restricted to learning linear SVMs. It is up to the deep neural network to induce non-linearity into the prediction. The model, dubbed DLSVM, achieves a greater generalisation accuracy than a CNN using a softmax output layer on the popular MNIST and CIFAR-10 datasets, and on the ICML 2013 Representation Learning Workshop's face expression recognition challenge.

This chapter proposes an alternative approach for learning convolutional filter weights in conjunction with a linear SVM weight vector. Instead of using backpropagation, it proposes an iterative, two-step optimisation. Each of the steps executes a convex optimisation. The chapter will begin by looking at the simplest case: a linear CSVM (L-CSVM) with a single convolutional filter, a stride of one and with no subsampling. In this case, the decision function may be defined in the primal as:

$$\hat{y} = \boldsymbol{w}^\intercal(\boldsymbol{x} * \boldsymbol{f}) + b, \tag{3.1}$$

where $\hat{y}$ is the predicted target of an instance $\boldsymbol{x}$, $\boldsymbol{f}$ denotes a convolutional filter, b symbolises the bias and $\boldsymbol{x} * \boldsymbol{f}$ expresses the convolution operation between $\boldsymbol{x}$ and $\boldsymbol{f}$. The dimension of the weight vector is therefore $|\boldsymbol{w}| - |\boldsymbol{f}| + 1$. The convolution operation aims to increase the distance between positive and negative support vectors so that a greater margin is possible.

Training of the L-CSVM may be achieved by making minor alterations to the SVM error function. First, the SVM constraints are altered to take into account the convolution operation, so that $y_i(\boldsymbol{w}^\intercal(\boldsymbol{x}_i * \boldsymbol{f}) + b) \geq 1 - \xi_i \quad \forall i$. Next, regularisation must be added to the filter weights. Without regularisation, the filter may grow or shrink without bound. To regularise, the squared L_2-norm is used, as for $\boldsymbol{w}$. With these changes, the full L-CSVM error function may be defined:

$$E = \frac{\lambda_{\boldsymbol{w}}}{2}\|\boldsymbol{w}\|^2 + \frac{\lambda_{\boldsymbol{f}}}{2}\|\boldsymbol{f}\|^2 - \frac{1}{N}\sum_{i=1}^{N}\max\{0, (y_i(\boldsymbol{w}^\intercal(\boldsymbol{x} * \boldsymbol{f}) + b)\}, \tag{3.2}$$

where $\lambda_{\boldsymbol{w}}$ and $\lambda_{\boldsymbol{f}}$ take the place of C in equation (2.21) in deciding how much the constraints control the decision boundary.

To simplify the task of optimising this error function, this thesis introduces a function which it calls the Convolution Matrix Transformation (CMT). The CMT function takes a feature vector, $\boldsymbol{x}$, and transforms it into a new vector whose inner product with $\boldsymbol{f}$ produces the same output as convolving $\boldsymbol{x}$ with $\boldsymbol{f}$ directly. The output of $\text{CMT}(\boldsymbol{x}, \boldsymbol{f})$ will be called the convolutional matrix and will be notated by $\boldsymbol{X}$. Thus $\boldsymbol{X} = \text{CMT}(\boldsymbol{x}, \boldsymbol{f})$. The feature vector may have any number of dimensions, for example, it may be a two-dimensional greyscale image or a three-dimensional colour image. The $\boldsymbol{X}$ created by CMT will be set up to perform convolution over each dimension

Algorithm 3 Two-dimensional CMT Algorithm

Inputs: $\boldsymbol{x} \in \mathbb{R}^{m \times n}, \boldsymbol{f} \in \mathbb{R}^{p \times q}$
Initialise: $\boldsymbol{X} \in \mathbb{R}^{((m-p+1)(n-q+1)) \times (pq)}$
for $i = 0$ to m - p + 1 **do**
    for $j = 0$ to n - q + 1 **do**
        for $k = 0$ to p **do**
            for $l = 0$ to q **do**
                $a := i(n - q + 1) + j$
                $b := kq + l$
                $X_{a,b} := x_{(i+k),(j+l)}$
            end for
        end for
    end for
end for
Output: $\boldsymbol{X}$

in the original image, such that:

$$\mathrm{CMT}(\boldsymbol{x}, \boldsymbol{f})^\intercal \boldsymbol{f} = \boldsymbol{X}^\intercal \boldsymbol{f} = \boldsymbol{x} * \boldsymbol{f}. \tag{3.3}$$

Figure 3.1 shows an example of a convolutional matrix which may achieve one-dimensional convolution. If $M = |\boldsymbol{x}|$ and $D = |\boldsymbol{f}|$ then the shape of $\boldsymbol{X}$ is $(M - D + 1)$ by D. The convolutional matrix to achieve one-dimensional convolution is created by stacking subsets of the original instance vector such that the start index of the subset increments by one at each row. Thus $\boldsymbol{X}$ is a staggered matrix containing the elements of $\boldsymbol{x}$. To achieve two-dimensional convolution, whereby a two-dimensional filter is convolved with a two-dimensional input vector, a more complex operation is required than staggering to create the convolutional matrix. In this case, Algorithm 3 may be used. To calculate the output of $\boldsymbol{x} * \boldsymbol{f}$, $\boldsymbol{f}$ must be vectorised. Thus $\mathrm{vec}(\boldsymbol{x} * \boldsymbol{f}) = \boldsymbol{X}^\intercal \mathrm{vec}(\boldsymbol{f})$. Using $\boldsymbol{X}$ instead of $\boldsymbol{x}$, the decision function and error function my be written as:

$$\hat{y} = \boldsymbol{w}^\intercal \boldsymbol{X}^\intercal \boldsymbol{f} + b \tag{3.4}$$

and:

$$E = \frac{\lambda_{\boldsymbol{w}}}{2}\|\boldsymbol{w}\|^2 + \frac{\lambda_{\boldsymbol{f}}}{2}\|\boldsymbol{f}\|^2 - \frac{1}{N}\sum_{i=1}^{N} \max\{0, (y_i(\boldsymbol{w}^\intercal \boldsymbol{X}^\intercal \boldsymbol{f} + b))\} \tag{3.5}$$

respectively.

An interesting property of this error function is that if the filter is held fixed, and the convolution logic is extracted into the setup of the dataset, such that $\boldsymbol{\mathcal{D}}' = \{\boldsymbol{X}^\intercal \boldsymbol{f}\}_{i=1}^N$, then optimisation of the weight vector looks like the usual SVM problem. Likewise, if the weight vector is held fixed, and the weight vector arithmetic is encoded into the training data such that $\boldsymbol{\mathcal{D}}'' = \{\boldsymbol{w}^\intercal \boldsymbol{X}\}_{i=1}^N$, then optimisation of the filter again looks like the usual SVM optimisation problem. Therefore learning may be performed as an iterative two-step optimisation; first by training $\boldsymbol{w}$ with $\boldsymbol{f}$ held fixed, and then by training $\boldsymbol{f}$ with $\boldsymbol{w}$ fixed. Both optimisation steps are convex and may be performed with an arbitrary linear SVM training algorithm, for example, SMO (Platt, 1998) or Pegasos (Shalev-Shwartz et al., 2011). Pseudocode for this is presented in Algorithm 4. The algorithm uses a command named SVM. This command calculates and returns the optimum weight vector for a set of inputs using an arbitrary linear SVM algorithm. It is used in step 6 to calculate the SVM weight vector. However, in step 8 it is used to learn the filter coefficients rather than the weight vector. This works as the cardinality of each instance in $\boldsymbol{\mathcal{D}}''is|\boldsymbol{f}|$.

Algorithm 4 L-CSVM Learning Algorithm

Inputs :$\boldsymbol{\mathcal{D}}, \lambda_{\boldsymbol{w}}, \lambda_{\boldsymbol{f}}$
Initialise :$\boldsymbol{w}, \boldsymbol{f}$
$\boldsymbol{X}_i := \text{CMT}(\boldsymbol{\mathcal{D}}_i, \boldsymbol{f}) \quad \forall i$
while not converged **do**
    $\boldsymbol{\mathcal{D}}' := \{\boldsymbol{X}_i{}^\intercal \boldsymbol{f}\}_{i=1}^N$
    $\boldsymbol{w}, b := \text{SVM}(\boldsymbol{\mathcal{D}}', \lambda_w)$
    $\boldsymbol{\mathcal{D}}'' := \{\boldsymbol{w}^\intercal \boldsymbol{X}_i\}_{i=1}^N$
    $\boldsymbol{f} := \text{SVM}(\boldsymbol{\mathcal{D}}'', \lambda_f)$
end while

3.1 Experimental Results: MNIST

Section 2.4 describes the MNIST image classification dataset (LeCun et al., 1998a). MNIST is widely used as a benchmark dataset and is freely available to researchers. Therefore, it is a natural dataset on which to test the L-CSVM. The results of the L-CSVM are compared against results of a linear SVM, and an equivalent CNN with a single convolutional filter.

3.1.1 Results

In order to implement Algorithm 4, a SVM implementation needed to be chosen. The classifier developed in this research used a custom implementation of the Pegasos algorithm written in Java. Pegasos was chosen due to its ability to converge rapidly in the linear case. The implementation included the optional projection step, although this was found to have little impact on the properties of the trained classifier, nor on the training time. Multiclass classification was achieved using a one-versus-one scheme, and, for simplicity, $\lambda_{\boldsymbol{w}}$ and $\lambda_{\boldsymbol{f}}$ were shared between all of the classifiers rather than selected per classifier. The MNIST target classes are the numerical digits from 0 to 9, and a different filter was learned for each pair of targets, resulting in 45 distinct filters. Each filter was of size 5×5. Cross-validation was used to tune $\lambda_{\boldsymbol{w}}$ and $\lambda_{\boldsymbol{f}}$. The 60,000 training instances were split into 50,000 for training and 10,000 for validation. With so many instances available per target class, k-fold validation was deemed unnecessary. A heatmap of cross-validation results can be seen in figure 3.2. The highest performing classifier had $\lambda_{\boldsymbol{w}} = 0.1$ and $\lambda_{\boldsymbol{f}} = 0.1$, and so these are the parameters which are used to determine generalisation accuracy.

As a comparison, a linear SVM without convolutional filters and a linear implementation of a CNN were also implemented. The SVM used the same custom implementation of Pegasos. In this implementation, the only parameter which required tuning was $\lambda_{\boldsymbol{w}}$, for which the optimum value was found to be 0.1. The CNN was also implemented in Java and used the open source Deeplearning4j library (Skymind, 2014). The architecture of the CNN was designed to be as equivalent as possible to the L-CSVM. It had a single convolutional filter, also of size 5×5, did not perform subsampling, and did not contain any hidden layers. Thus, the convolutional feature maps fed directly into a linear classifier, as was the case in the L-CSVM. It used the sigmoid function in the output layer and used log loss to calculate error gradient. For training, stochastic gradient descent was used with Nesterov's Accelerated Gradient (NAG) (Nesterov, 1983) momentum. NAG momentum achieved superior classification results and resulted in a lower training

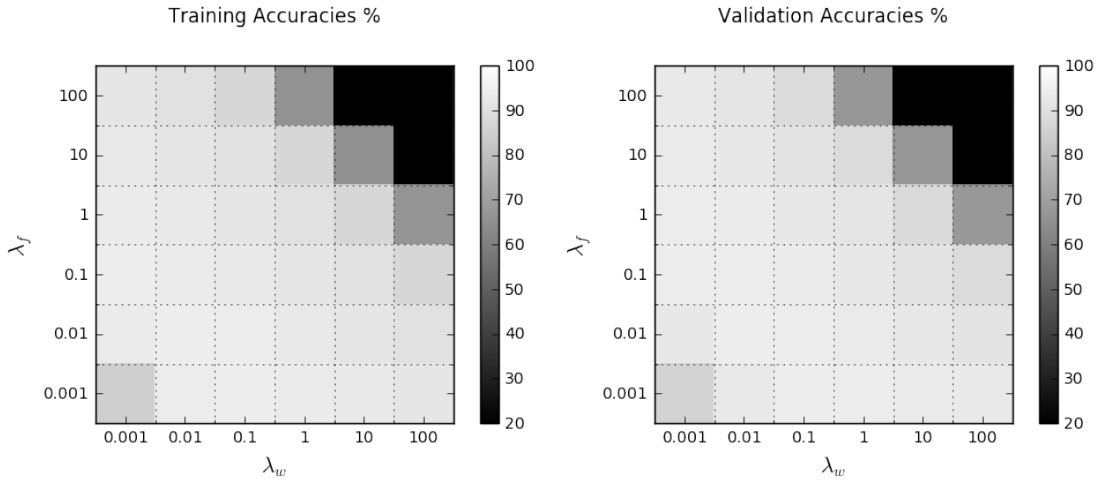

Figure 3.2: A heatmap of cross-validation results on the training and validation sets. Cross-validation aimed to learn effective values of $\lambda_{\boldsymbol{f}}$ and $\lambda_{\boldsymbol{w}}$.

MNIST Results		
Model	Training accuracy %	Testing accuracy %
Linear SVM	93.57 ± 0.053	93.68 ± 0.15
Linear CNN	91.92 ± 0.089	91.60 ± 0.15
L-CSVM	94.06 ± 0.072	94.09 ± 0.14

Table 3.1: Accuracy averaged over ten runs on the MNIST testing set using a variety of classifiers. The error is estimated using the standard deviation.

time than equivalent networks using conventional momentum, and without momentum. The two variables which required tuning were the learning and regularisation rates. Cross-validation on the validation set found the optimum values to be 10^{-3} and 10^{-5} respectively.

Table 3.1 shows the results of each of the classifiers. Accuracies are averaged over ten runs of the model and the error estimate shows the standard deviation. Accuracy was chosen as the performance metric over precision, recall or F1-score as a correct classification of all target classes is of equal importance. As can be seen from the results, the L-CSVM achieves an accuracy of 94.09% on the testing set which is the highest accuracy of each of the tested classifiers. Interestingly, the linear SVM outperforms the linear CNN. This suggests that a linear classifier generalises sufficiently well on MNIST that it does not fully benefit from the additional generalisability provided by the convolutional filter. This is further evidenced by the fact that the SVM achieves a similar accuracy on the testing set as it does on the training set. The conclusion may be drawn that in the linear case, the convex properties of the SVM learning algorithm are more important than the shift invariance provided by a single convolutional filter. Chapter 4 explores whether this is the case for more flexible classifiers.

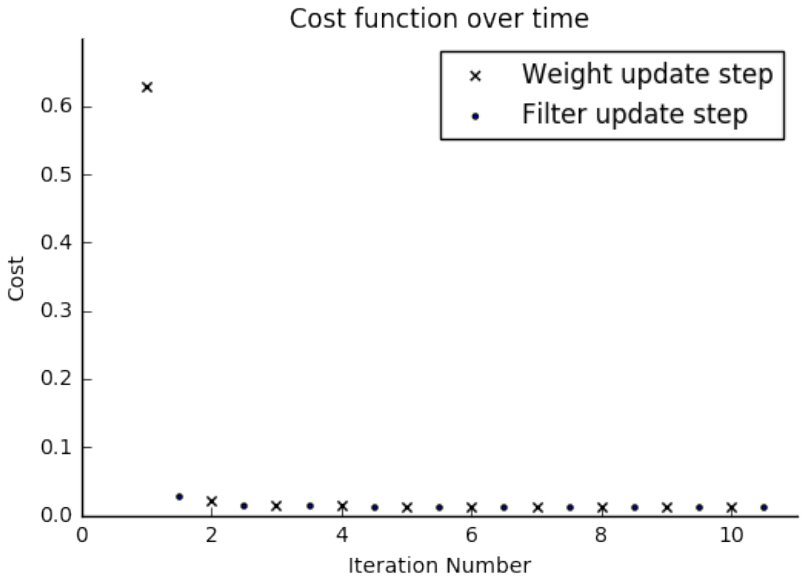

Figure 3.3: L-CSVM error function from equation (3.2) plotted during training. The error is measured twice per iteration, after the weight vector update and then after the filter update.

3.1.2 Analysis

Correct behaviour of the algorithm may be assessed by monitoring the value of the error function during training. The value should decrease each iteration. Moreover, it should decrease after both the weight update and the filter update stages. Figure 3.3 shows this to be the case. After the initial iteration, the weight and filter update steps reduce the cost function by an approximately equal amount each iteration. This confirms that the optimisation procedure is effective. At iteration six, the algorithm sees a small increase in cost function on the weight update step, from 0.01333 to 0.01339. Subsequent iterations continue to rise and fall at around these values. As Pegasos is an approximate solving algorithm and is inherently non-deterministic, this is to be expected. Pegasos itself requires the number of iterations to be specified. A greater number of iterations allows Pegasos to find a solution closer to the true optimum. In turn, this allows the L-CSVM algorithm to continue finding a solution with a reduced cost function for a greater number of iterations. The classifier used to create the graph in figure 3.3 was configured to run Pegasos for 10^6 iterations on both the weight and filter update stages. This value was chosen for demonstrational purposes and is excessive for a real classifier. Since Pegasos is learning a convex function, configuring it to run for an excessive number of iterations does not cause overfitting. The disadvantage is in the increased running time for little benefit. All other experiments in this section have been configured to run Pegasos for 10^5 iterations.

Figure 3.4 shows two visual representations of the filters learned by the L-CSVM. Each representation is from a different one-versus-one combination. The first is the filter obtained from training a classifier with instances of target classes zero and one, the second from classes three and seven. Both filters can be recognised as low-pass filters, averaging the pixels to which they are applied. However, the two are oriented differently. The filter which separates instances with

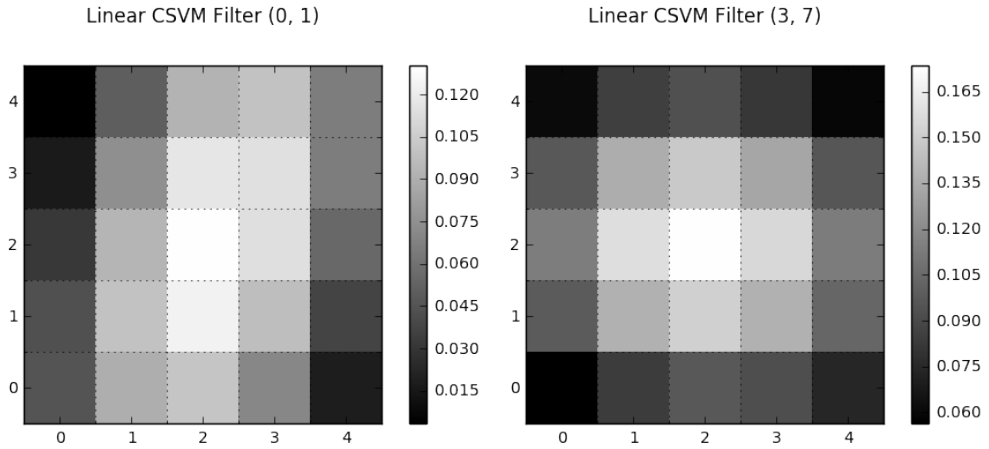

Figure 3.4: A visualisation of the filters learned by the L-CSVM on two different pairs of targets. The filters can both be recognised as low-pass filters, however, the orientation is different in each.

target zero and one is oriented so that it responds well to the vertically oriented pixel arrangements seen in images of ones and less well to circularly symmetric arrangements of zeros. The filters which are learned for each combination are close to identical each time that the algorithm is run, even when $\boldsymbol{w}$ and $\boldsymbol{f}$ are initialised randomly. This is likely because the optimisation problems for both $\boldsymbol{w}$ and $\boldsymbol{f}$ are convex. Further work is required to determine whether the error function is convex with respect to $\boldsymbol{w}$ and $\boldsymbol{f}$ simultaneously.

As a comparison, figure 3.5 displays the filter learned by the linear CNN. To make the comparison fair, the CNN was trained on images with a target of 0 or 1. Both of the two presented CNN filters were trained for 100 epochs. The filters are very different in appearance to the filters learned by the L-CSVM. On each run, the CNN learns a different filter, as shown in the figure. This is likely due to the non-convex nature of the CNN error function. Each filter is produced by a different local minimum and resultantly has less spatial coherence. It could be argued that the filters are almost random in appearance. Without viewing the feature maps produced by a filter, it is difficult to envisage what effect the filter will have on the original image. Figure 3.6 shows a set of example feature maps produced by both the L-CSVM and the CNN. The filter used for creating the L-CSVM feature maps is the (0, 1) filter displayed in figure 3.4. For creating the CNN feature maps, the left-hand filter in figure 3.5 was used. To enhance visual contrast, values have been rescaled between 0 and 256 per image and thus the same colour within two images does not mean that the activations within the feature maps are equivalent. Like the filters themselves, the feature maps created by the two models are very different in appearance. The low-pass L-CSVM filter has a blurring effect on the image, whereas the CNN filter appears to be performing a diagonal edge detection on the rightmost edges as well as blurring. It can be noted that the (0, 1) L-CSVM filter is oriented so as to average along the northeast to southwest diagonal. This corresponds to the direction in which the 1s in the images are oriented. Initially blurring sounds like a disadvantage due to the decrease in detail, however, it increases the shift invariance of the classifier. For a dataset like MNIST which has images without a background, trading contrast for shift invariance is a worthwhile tradeoff, hence the increase of accuracy between the SVM and L-CSVM.

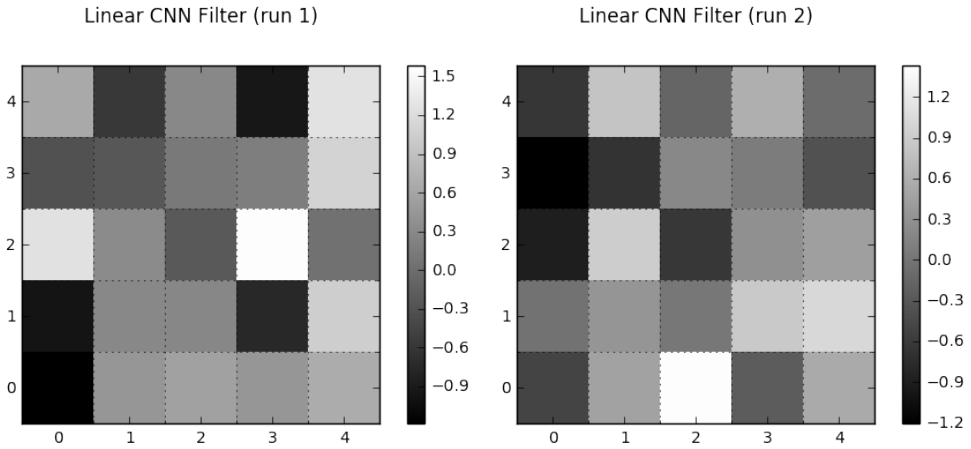

Figure 3.5: A visualisation of two filters learned by the CNN. Both filters aim to separate instances with a target class of zero or one. The filters have less spatial coherence than the filters learned by the L-CSVM. A unique filter is learned each time that the CNN is trained.

3.2 Conclusion

This chapter presents a method which may be used to introduce convolution to the SVM decision function. The convolution operation may be considered a pre-processing step before classification through a linear SVM. The convolutional filter is trained in conjunction with SVM parameters. This is achieved by using a two-step optimisation, first by learning the SVM weight vector while holding the filter weights fixed, then by holding the weight vector constant and training the filter weights. The filter weights are learned by casting the filter update step as an ordinary SVM optimisation problem, but with respect to the filter weights rather than the SVM weight vector. Thus, each of the steps is a convex optimisation with respect to the parameter which requires tuning. It is shown in figure 3.3 that this optimisation procedure successfully minimises the error function.

The L-CSVM is trained and tested on the MNIST dataset. It is found to learn low-pass filters, with an orientation dependant on the input data. For example, in figure 3.4 it is shown that the filter learned by a classifier which aims to separate instances with a target of zero and one is in a different orientation to the filter learned by a classifier which aims to separate instances with targets of three and seven. On the test set, the model achieves a generalisation accuracy of $94.09\% \pm 0.14$. This is greater than the generalisation accuracy achieved by a linear SVM without convolution, or by a linear CNN with a single convolutional filter.

However, the flexibility of the L-CSVM is limited by the linearity of the SVM which it uses for classification. On the MNIST dataset, this is restrictive, and as a result, the linear SVM and L-CSVM fall short of the generalisation accuracy achievable by a kernelised SVM. For example, in (Burges and Schölkopf, 1997) a benchmark SVM with a polynomial kernel of degree 5 and no other adjustments or preprocessing is shown to achieve a generalisation accuracy 98.6%. The focus of the following chapter is to explore how the linear formulation described in this chapter may be adjusted to allow the use of a more flexible, kernelised SVM.

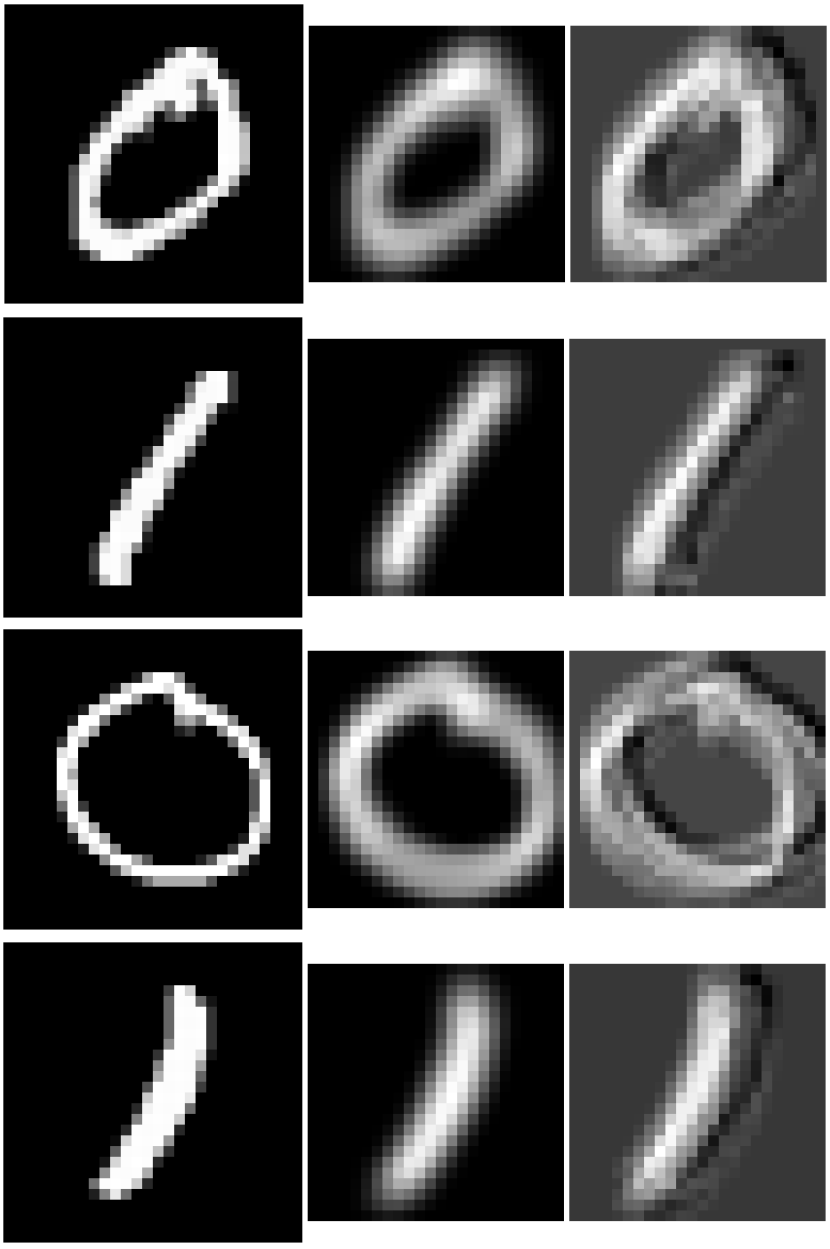

Figure 3.6: Feature maps after convolving with the learned filters. Left column: original image, middle column: image filtered by L-CSVM, right column: image filtered by CNN.

Chapter 4

Kernelised Convolutional SVMs

The previous chapter presents a method which may be used to augment a SVM with a convolutional filter, such that the convolutional filter is used as a preprocessor before classification through the SVM. It presents an algorithm which may be used to optimise the SVM parameters and filter weights in parallel, using an iterative two-step optimisation. However, the algorithm requires that the SVM is linear, thus limiting the model's flexibility. This chapter presents a method which may be used to learn a kernelised SVM while estimating filter parameters using gradient descent. It tests the algorithm on the MNIST and CIFAR-10 benchmark datasets and compares the results with a kernelised SVM and a CNN with a single convolutional layer.

The L-CSVM formulation described in equation (3.5) is presented in the primal. As explained in section 2.5 of the background chapter, the usual approach for implementing a kernelised SVM is to move into the dual. Unfortunately, the L-CSVM formulation is not easily transferred to the dual as the filter update step requires the output of $\boldsymbol{w}^\intercal \boldsymbol{X}$. A possibility would be to remain in the primal and use a feature mapping, however, this would likely result in a classifier for which training is so complex that it could not be used on any dataset of reasonable size. It would also restrict the available kernels to those in a low number of dimensions. Kernels like the RBF kernel, which map instances into an infinite number of dimensions, would be intractable. Instead, a new approach is presented in the dual which leverages the ideas of multiple kernel learning. The resulting model is referred to as the K-CSVM.

To recap, multiple kernel learning is a continuation of the SVM whereby the task is to learn the kernel function as well as the SVM parameters. It exists in two forms. The first form is where a number of kernels are predefined, and the task of learning is to choose how they are combined, as in (Rakotomamonjy et al., 2008). Another form of MKL is where the combination function is predefined and instead, the aim is to learn the parameters which are integrated into each kernel; see for example (Chapelle et al., 2002). As hinted in section 2.6, this MKL formulation is crucial to the K-CSVM. Unless specified otherwise, references to MKL will now mean the second formulation.

4.1 Convolutional Kernels

Although MKL may be used to learn the parameters of multiple kernels, there is no requirement to use multiple kernels. It may be used to learn the parameters of a single kernel. Section 2.6 presented an algorithm named Generalised Multiple Kernel Learning (Varma and Babu, 2009). GMKL uses convex optimisation to find SVM parameters, while also optimising kernel parameters using gradient descent. The algorithm is generalised as it may be used to optimise the parameters of any differentiable kernel with the mild restrictions that the gradient is continuous with respect to its parameters, and that the kernel is positive definite for all valid parameters. The reason that this is of interest is that convolution is a differentiable function with respect to its filter weights. Furthermore, it is possible to move the convolution operation into the kernel function and to treat the filter weights as kernel parameters. In practice, this is equivalent to using convolution as a preprocessing step, but it means that the formulation conforms to the format expected by GKML. Thus, by casting the problem of learning in this way, GMKL may be used to optimise the filter weights using gradient descent. This idea is explored and tested for the polynomial and RBF kernel functions.

The standard polynomial kernel function is as follows:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^\intercal \boldsymbol{x}_j + 1)^d, \tag{4.1}$$

where d is the polynomial degree for controlling the complexity of the feature mapping. This may be adapted to include the convolution operation by making the following alterations:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{f}) = ((\boldsymbol{x}_i * \boldsymbol{f})^\intercal (\boldsymbol{x}_j * \boldsymbol{f}) + 1)^d. \tag{4.2}$$

It may be simplified by using the CMT function described in equation (3.3):

$$\begin{aligned} K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{f}) &= ((\boldsymbol{X}_i \boldsymbol{f})^\intercal (\boldsymbol{X}_j \boldsymbol{f}) + 1)^d \\ &= (\boldsymbol{f}^\intercal \boldsymbol{X}_i^\intercal \boldsymbol{X}_j \boldsymbol{f} + 1)^d, \end{aligned} \tag{4.3}$$

where $\boldsymbol{X}_i = \text{CMT}(\boldsymbol{x}_i, \boldsymbol{f})$. For the filter update step, GMKL requires the derivative of the error function. Varma and Babu (2009) show that this is calculated by using:

$$\frac{\partial r}{\partial f_k} - \frac{1}{2} \boldsymbol{\alpha}^{*\intercal} \frac{\partial \boldsymbol{H}}{\partial f_k} \boldsymbol{\alpha}^*, \tag{4.4}$$

where $\boldsymbol{H}$ is the matrix generated by calculating $y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{g})$ for all combinations of i and j. As shown in the appendix, $\frac{\partial \boldsymbol{H}}{\partial f_k}$ may be calculated by using:

$$\frac{\partial H_{ij}}{\partial f_k} = y_i y_j d(1 + \boldsymbol{f}^\intercal \boldsymbol{X}_i^\intercal \boldsymbol{X}_j \boldsymbol{f})^{d-1} [\boldsymbol{f}^\intercal (\boldsymbol{X}_i^\intercal \boldsymbol{X}_j + \boldsymbol{X}_j^\intercal \boldsymbol{X}_i)]_k \tag{4.5}$$

for the polynomial kernel.

A similar operation may be performed to introduce convolution into the RBF function. The original function is defined:

$$K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \exp(-\gamma \|\boldsymbol{X}_i - \boldsymbol{X}_j\|^2), \tag{4.6}$$

and may be rewritten as:

$$\begin{aligned} K(\boldsymbol{X}_i, X_j, \boldsymbol{f}) &= \exp(-\gamma \|\boldsymbol{X}_i \boldsymbol{f} - \boldsymbol{X}_j \boldsymbol{f}\|^2) \\ &= \exp(-\gamma (\boldsymbol{X}_i \boldsymbol{f} - \boldsymbol{X}_j \boldsymbol{f})^\intercal (\boldsymbol{X}_i \boldsymbol{f} - \boldsymbol{X}_j \boldsymbol{f}) \\ &= \exp(-\gamma \boldsymbol{f}^\intercal (\boldsymbol{X}_i - \boldsymbol{X}_j)^\intercal (\boldsymbol{X}_i - \boldsymbol{X}_j) \boldsymbol{f}). \end{aligned} \tag{4.7}$$

The derivative of $\boldsymbol{H}$ is given by:

$$\frac{\partial H_{ij}}{\partial f_k} = 2\gamma y_i y_j K(\boldsymbol{X}_i, \boldsymbol{X}_j)[\boldsymbol{f}^\intercal(\boldsymbol{X}_i - \boldsymbol{X}_j)^\intercal(\boldsymbol{X}_i - \boldsymbol{X}_j)]_k \tag{4.8}$$

Again, the calculations used to obtain the derivative are shown in the appendix.

4.2 Regularisation

GMKL allows the parameters of the kernel, which in this case means the filter, to be regularised using any differentiable function. Regularisation keeps the filter weights within a feasible range. The only restrictions on the regulariser are that its derivative exists and is continuous. Many regularisers may be developed which satisfy these restrictions. This includes a number of p-norm regularisations, for example, L_1 and L_2 regularisation. The regulariser is parameterised by a weighting, λ, which controls the influence of the regularisation term. The effect of using p-norm regularisation with a selection of λ values is investigated in section 4.4.

As well as investigating p-norm regularisation, this work develops its own form of regularisation. The aim of this regularisation is to encourage the filter to converge so that the value of a chosen form of p-norm regularisation is close to one. This form of regularisation will be called Distance-From-One regularisation and will be denoted by DFO_p, where p is the selected p-norm. DFO_p regularisation may be defined:

$$r(\boldsymbol{f}) = \lambda_f(\max(0, \|\boldsymbol{f}\|_p - 1))^2. \tag{4.9}$$

This term is minimised when $\|\boldsymbol{f}\|_p \leq 1$. As will be discussed in detail later in the chapter, on each of the tested datasets, the K-CSVM prefers to choose filters with a norm $\|\boldsymbol{f}\|_p > 1$ when no regularisation is used. Therefore, it is expected that with DFO_p regularisation, the SVM will choose a filter with norm slightly above, but close to one. Like the constraints of a SVM, the target value is set to one for simplicity, though this choice is suspected to be unimportant. More important is that a target value exists. With a target in place, the learning process must focus on optimising cost by adjusting values relative to each other rather than simply altering the size of $\boldsymbol{f}$. If one value increases then another must decrease so that the norm remains close to the target. Like with the p-norm, this regulariser is parameterised by λ. In this case, λ controls the extent to which the classifier makes an effort to constrain the size of the filter to one. A large λ will force the classifier to learn a filter with a size close to one, whereas a small λ allows the filter to diverge if this is advantageous to the cost function. For use in GMKL the regulariser must be differentiable with respect to the weights which it concerns. The derivative of this function will depend on the choice of p. For example, if $p = 1$ then the derivative is given by:

$$\frac{\partial r(\boldsymbol{f})}{\partial f_i} = \begin{cases} \frac{\lambda f_i(\|\boldsymbol{f}\|_1 - 1)}{|f_i|\|\boldsymbol{f}\|_1} & \text{if } \|\boldsymbol{f}\|_1 > 1 \\ 0 & \text{if } \|\boldsymbol{f}\|_1 \leq 1 \end{cases} \tag{4.10}$$

When $p = 2$, the derivative is:

$$\frac{\partial r(\boldsymbol{f})}{\partial f_i} = \begin{cases} \frac{2\lambda f_i(\|\boldsymbol{f}\|_2 - 1)}{\|\boldsymbol{f}\|_2} & \text{if } \|\boldsymbol{f}\|_2 > 1 \\ 0 & \text{if } \|\boldsymbol{f}\|_2 \leq 1 \end{cases} \tag{4.11}$$

Algorithm 5 K-CSVM Training algorithm

Inputs: $\boldsymbol{\mathcal{D}}$
Initialise: $\boldsymbol{f}$ randomly
for 0 to t **do**
    Set K using (4.3) or (4.7)
    $\boldsymbol{\alpha}^* := \text{SVM}(\boldsymbol{\mathcal{D}}, \boldsymbol{f}, K)$
    Calculate $\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{f}} :=$ using equation (4.5) or equation (4.8).
    $\boldsymbol{f} := \boldsymbol{f} - \eta(\frac{\partial r}{\partial \boldsymbol{f}} - \frac{1}{2}\boldsymbol{\alpha}^{*\intercal}\frac{\partial \boldsymbol{H}}{\partial \boldsymbol{f}}\boldsymbol{\alpha}^*)$
end for

4.3 Learning

Algorithm 5 presents pseudocode which may be used to train a K-CSVM. It is similar to the generic GMKL algorithm shown in Algorithm 2, however, it has been specialised for filter learning. Like the L-CSVM training algorithm, this algorithm uses a command named SVM. In this case, the SVM command determines $\boldsymbol{\alpha}^*$ using an arbitrary linear SVM algorithm and returns it. The K-CSVM in this work used a set number of iterations, t, to decide when training was complete, though more intelligent stopping criteria could have been used.

Varma and Babu (2009) use the Armijo rule to select step size. The Armijo rule is an iterative line search which may be used to find a reasonable, but not necessarily optimal step size. If a high degree of convergence is required then the Armijo rule would likely be beneficial, however, for the purposes of this work a constant step size was found to be adequate. The constant step size was controlled by a learning rate, η.

4.4 Experimental Results: MNIST

This section begins by presenting results on the MNIST dataset. As before, it compares the results of the K-CSVM with a SVM and an equivalent CNN. Section 4.4.2 provides an analysis of the model to ensure correct behaviour, for example, by monitoring the error function during training. Information on how the K-CSVM was tuned is provided in section 4.4.3.

4.4.1 Results

Due to the lengthy training time of the K-CSVM, it was not feasible for this work to train on the full MNIST dataset. Instead, the K-CSVM in this work is trained on an MNIST subset, containing each of the instances with a label of three or five which reside within the first 10,000 training instances. This turned out to be 1020 instances with a target of three and 882 instances with a target of five. The targets of three and five were selected as these were found to be the most difficult classes for an ordinary SVM to separate. The hope was that the most difficult combination would provide the greatest opportunity for improvement and that the effect of learning would be more apparent than it would be for an easier combination.

The K-CSVM developed in this section was implemented in Python. Instead of using the

MNIST Results		
Model	Training accuracy %	Testing accuracy %
SVM	100.00 ± 0	98.32 ± 0
CNN	100.00 ± 0	98.38 ± 0.25
K-CSVM	100.00 ± 0	98.59 ± 0.02

Table 4.1: Accuracy on a binary MNIST subset containing instances with target 3 or 5. Each classifier was trained on a subset containing only the instances with target of 3 or 5 within the first 10,000 training instances. The CNN has a single convolutional filter and does not perform subsampling.

custom implementation of Pegasos which was used previously, the K-CSVM used the scikit-learn library (Pedregosa et al., 2011) for the implementation of the SVM. Scikit-learn uses the widely distributed LibSVM algorithm (Chang and Lin, 2011), which is itself a version of Sequential Minimal Optimisation (SMO) (Platt, 1998). SMO is an iterative algorithm, but if required it can be used to solve the quadratic programming problem exactly. Yet this is not the reason for the move away from Pegasos. The reason for this is because Pegasos is designed with the primal problem in mind. For the primal problem, Pegasos achieves a rapid convergence rate, however, for kernelised learning, Pegasos performs rather averagely, as admitted in (Shalev-Shwartz et al., 2011). LibSVM is designed for the dual problem and therefore excels at kernelised learning. An alternative which was investigated was LASVM (Bordes et al., 2005). LASVM is an approximate solver which boasts competitive misclassification rates after a single pass over the data. Ultimately, LibSVM was chosen because using a prebuilt SVM package allowed efforts to be directed at the development of the K-CSVM algorithm. As the focus of this dissertation is aimed at proving the algorithm as a concept, the greater convergence time of LibSVM compared to LASVM is a worthwhile tradeoff for the reduced complexity it offers. However, in future work it would be worthwhile revisiting LASVM as the convergence time of the K-CSVM algorithm had an impact on the datasets which could be tested on.

Table 4.1 shows the accuracy of the K-CSVM on the MNIST subset and compares it against a SVM and an equivalent CNN. Results are averaged over five runs and error is estimated using the standard deviation. The previous section presents kernels and derivatives for both the polynomial and RBF kernels, however, this section focuses primarily on the polynomial kernel. The K-CSVM used to obtain results for table 4.1 used a polynomial degree of two and had a filter size of 5×5. DFO_1 regularisation was used to regularise the filter and λ was configured to 0.01. The learning rate was set to 0.1 and the value of C was greater than every coefficient in $\boldsymbol{\alpha}$. As will be discussed in further detail later in this section, a polynomial kernel with a degree of two did not benefit from regularisation on the weight vector. The classifier was trained for 25 iterations and it did not perform projection other than when the filter was first initialised, at which point it was rescaled so that its L_1 norm was equal to one. Section 4.4.3 provides detail on the process used to select this configuration.

The SVM included in the results was also implemented using the scikit-learn library. It used a polynomial kernel of degree two, and like the K-CSVM, it had C set to a value greater than any of the values in $\boldsymbol{\alpha}$. This was found to be the best performing configuration on the validation set. As with the linear case, the CNN was implemented in Java and used the Deeplearning4j library. It was intended to be as equivalent as possible to the K-CSVM, using a single convolutional filter of size 5×5 and no subsampling. However, unlike before, it used hidden nodes to allow

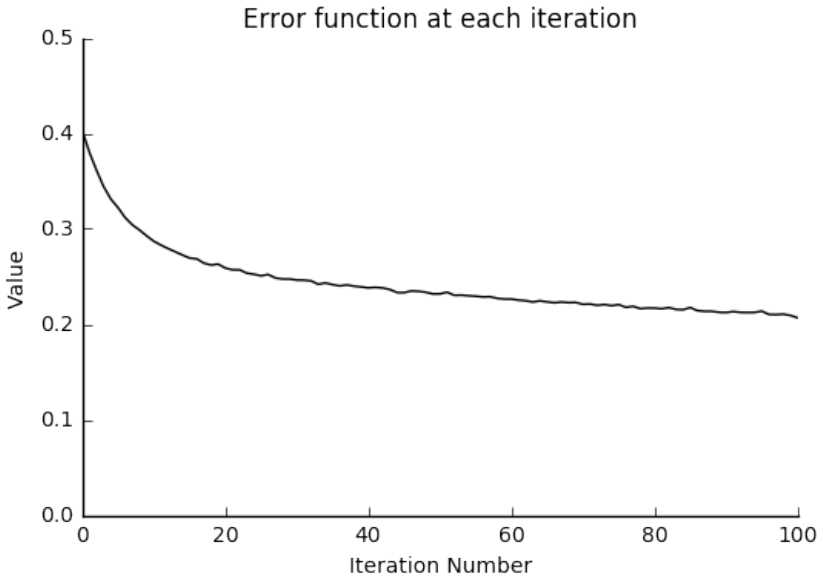

Figure 4.1: The error function after the SVM update of each iteration. The experiment uses DFO_1 regularisation with $\lambda_f = 1$, a learning rate of 0.001 and a polynomial kernel of degree two. The classifier is trained on a subset of MNIST containing the instances with target 3 or 5 within the first 1000 instances.

non-linearity. The output layer used a one-hot encoding scheme with two output nodes. The highest performing CNN had a single hidden layer containing 500 nodes. The nodes in the hidden and output layers used the sigmoid activation function. Nodes in the convolutional layers used the identity activation function. The learning rate was configured to 0.1 and the network was regularised using L_2 regularisation with a weighting of 10^{-5}. Nesterov's momentum was employed with a weighting of 0.9. It could be argued that using momentum gave the CNN an unfair advantage over the K-CSVM which used gradient descent without momentum. Momentum had a considerable impact on the generalisation accuracy of the CNN; without it, the generalisation accuracy decreased from 98.38 ± 0.25 to 96.41 ± 0.48, averaged over five runs. For preprocessing, each of the three classifiers performed a simple rescaling operation so that the input values ranged between zero and one.

4.4.2 Analysis

This section will analyse the behaviour of the algorithm during learning to determine whether it is acting as expected. Perhaps the most effective way to ensure the correct behaviour of the algorithm is to monitor the cost function throughout learning. Figure 4.1 shows the result of doing so. To recap, the error function of the K-CSVM is $\frac{1}{2}\boldsymbol{w}^\intercal\boldsymbol{w} + \sum_i l(y_i, f(\boldsymbol{x}_i)) + r(\boldsymbol{f})$, and the aim is to minimise this function. The figure, which shows the value of the error function each iteration after finding $\boldsymbol{\alpha}^*$, shows that minimisation correctly occurs. The classifier is trained on a subset of MNIST containing the instances with target 3 or 5 within the first 1000 instances. It used the same configuration as the classifier used to obtain the results in table 4.1, apart from

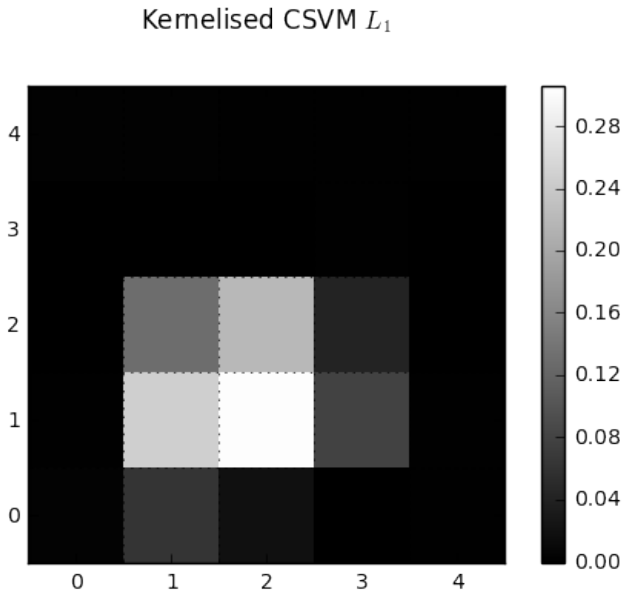

Figure 4.2: A visual depiction of the filter learned by a K-CSVM using DFO_1 regularisation.

the values of the learning rate and of λ which were set to 0.001 and 1 respectively. A reduced learning rate better highlighted the learning process.

Figure 4.2 shows the final filter trained by one of the classifiers used to obtain the results in table 4.1. It is considerably more sparse in appearance than the filters trained by a L-CSVM. However, when the filter is applied to an image, it becomes apparent that it is performing a similar blurring operation, though the blurring is to a lesser degree. This is shown in figure 4.3. On each run of the K-CSVM a different filter was learned. Although the filters were similar in appearance, usually with values on the perimeter close to zero, the inner values varied per run. As will be discussed in section 4.4.3, this work investigated the use of a number of different regularisation types. It was found that the classifier learns filters similar to the one displayed in 4.2 if the chosen regularisation is within the L_1 family. L_1, L_1^2 and DFO_1 were all tested and were found to learn filters of similar appearance and values. However, when using a norm in the L_2 family, the filters resemble those learned by the L-CSVM, as shown in figure 4.4. This is expected as the L-CSVM uses L_2^2 to regularise the filter. The filter displayed in figure 4.4 uses a polynomial kernel with a degree of two. If the degree is increased then the appearance of the filter remains unchanged. However, if the degree is decreased to one so that the classifier is linear, then the learned filter is different again. This is shown in figure 4.5 and the effect it has on the original images is displayed in figure 4.6. The filter appears to be detecting curves in the original image whereas the vertical uprights in the original image have a low activation in the feature map. When using regularisation in the L_2 family, the filters are more stable between runs than when using regularisation in the L_1 family. That is, if two K-CSVMs are trained using the same configuration and training dataset then the appearance of the final filter will be similar for each.

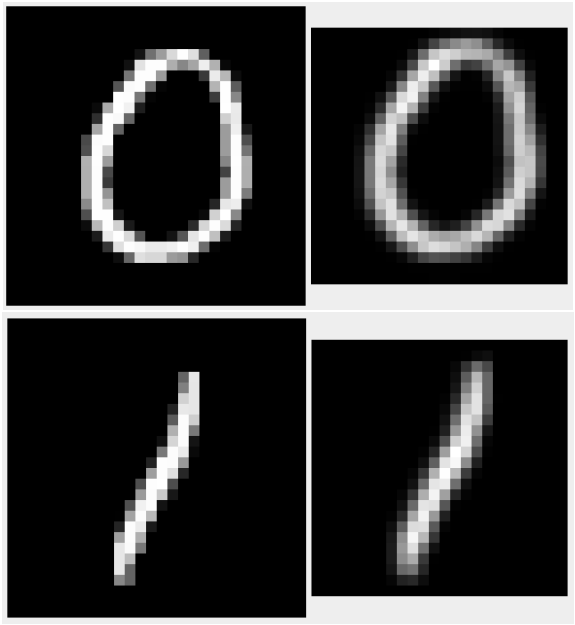

Figure 4.3: The effect of the filter displayed in figure 4.2 on MNIST images. The left column shows the original images and the right column the feature maps. The filter performs a blurring operation, however, the blurring is to a lesser degree than the filter learned by the L-CSVM.

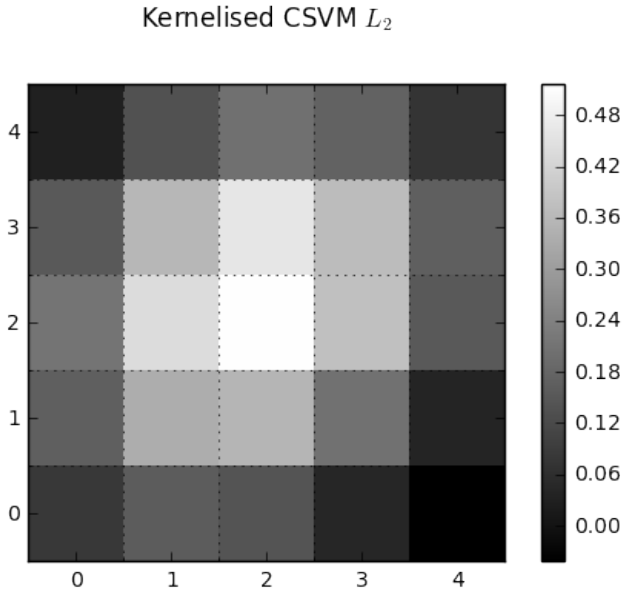

Figure 4.4: A visual depiction of the filter learned by a K-CSVM using DFO_2 regularisation and a polynomial degree of two.

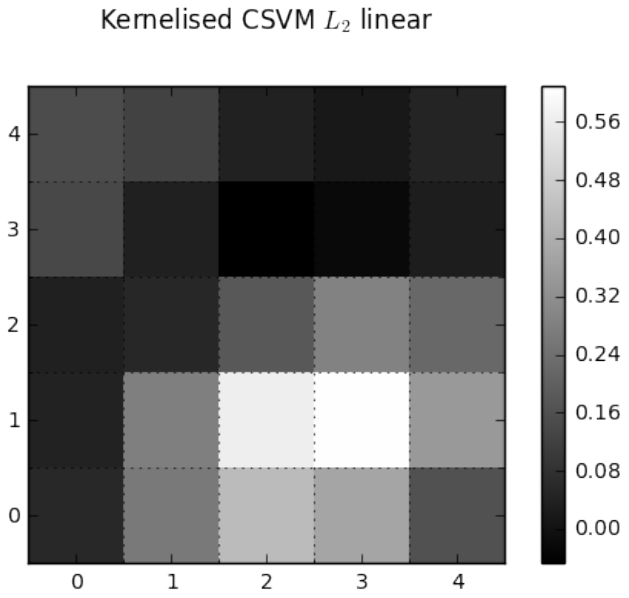

Figure 4.5: A visual depiction of the filter learned by a K-CSVM using DFO_2 regularisation and a polynomial degree of one.

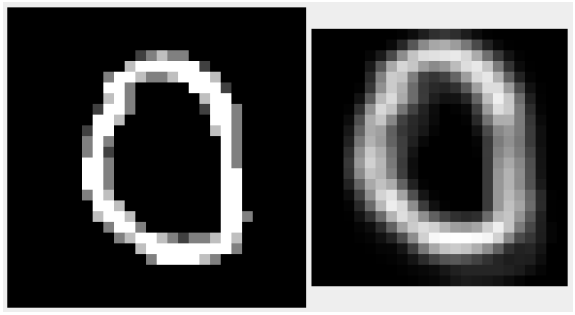

Figure 4.6: The effect of the filter displayed in figure 4.5 on MNIST images. The left column shows the original images and the right column the feature maps.

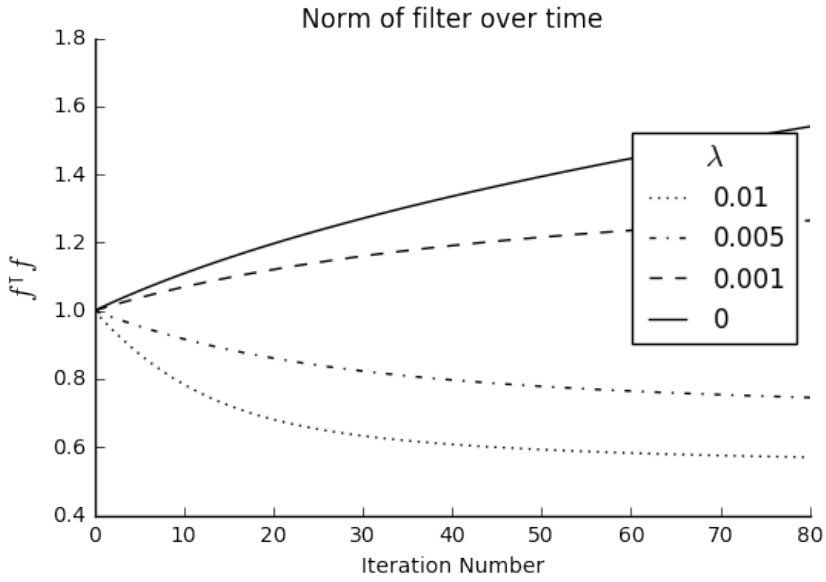

Figure 4.7: A graph of $\|\boldsymbol{f}\|_2^2$ at each iteration for a selection of regularisation values when using L_2 regularisation on the filter.

4.4.3 Tuning

Section 4.4.1 displayed the results of the K-CSVM on the MNIST dataset, however, little information was provided on how the parameters were selected. The free parameters which require tuning are the learning rate, the type of regularisation on the filter and the value of λ, the value of C, and the polynomial degree. Due to the lengthy training time of the K-CSVM, it was not feasible cross-validate all of these parameters. This section gives details of how these were tuned.

The section will begin by looking at the effect of tuning λ, first in the case of p-norm regularisation, and then for DFO_p regularisation. To recap, the aim of filter regularisation is to control the size of the filter. The extent to which this size is controlled is determined by the choice of λ. A large value of λ should be more restrictive on the size of the filter than a small value. Figure 4.7 shows the results of an experiment in which a series of classifiers were trained, each time varying λ. The classifiers used a polynomial kernel of degree two, a filter size of 5×5 and a learning rate of 1. C was set sufficiently high that it was not reached by any item in $\boldsymbol{\alpha}$. Each classifier was trained on the first 5,000 instances of the MNIST dataset. This experiment empirically confirms that greater regularisation values are more restrictive on the the size of the filter. Each of the classifiers is using L_2^2 regularisation, though similar behaviour was observed for L_2, L_1^2 and L_1. Figure 4.8 shows an equivalent graph but the classifier is using DFO_2 regularisation. As before, the results of the experiment exhibit the expected behaviour since large values of λ result in a final filter which is close to one. Small values of λ are allowed more leniency. The trained filters were found to be similar in appearance and value to the filters trained using equivalent p-norm regularisation.

The range of acceptable learning rate values was impacted by the choice of λ. Larger values

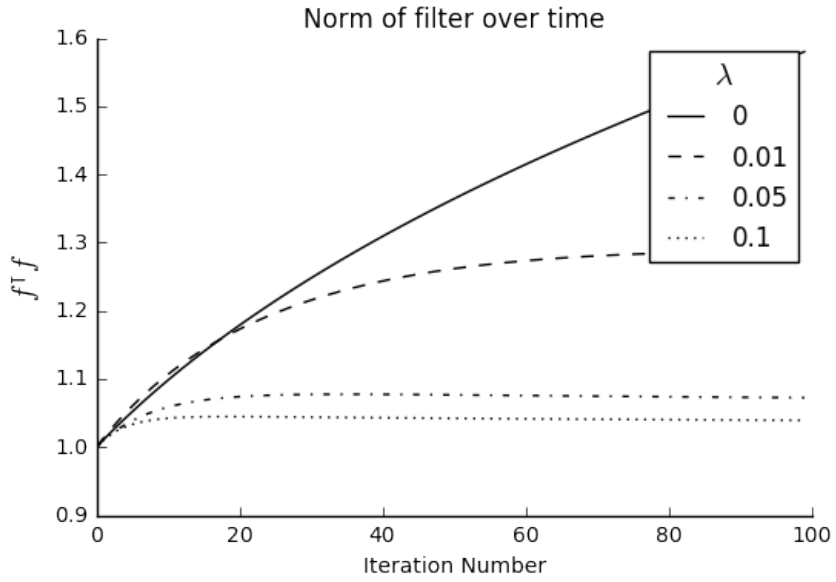

Figure 4.8: A graph of $\|\boldsymbol{f}\|_2^2$ at each iteration for a selection of regularisation values when using DFO_2 regularisation.

		λ		
		0.01	0.1	1
η	0.01	97.73 ± 0.10	97.95 ± 0.05	97.95 ± 0.04
	0.1	97.83 ± 0	97.89 ± 0.06	98.01 ± 0.15
	1	97.99 ± 0.06	98.01 ± 0.041	97.83 ± 0.20

Table 4.2: Cross-validation to find the effect of tuning λ and the learning rate. Reported values show the accuracy and the error is estimated using standard deviation. A result of n/a indicates failure to converge. Each classifier was trained on a subset containing instances with target 3 or 5 which fall within the first 5,000 instances.

of λ required a smaller learning rate in order to converge. If the learning rate was set too high then the size of the filter was found to increase per iteration. This happened if the magnitude of $s\nabla \boldsymbol{w}$ was approximately twice the magnitude of $\boldsymbol{f}$ or greater. In this scenario, the filter values oscillated between positive and negative, growing in size at each iteration.

Table 4.2 shows the results of cross-validation over λ and the learning rate. The model was trained on a subset of the instances of target 3 or 5 which fall within the first 5,000 instances of the training dataset. It is using DFO_1 regularisation. The aim of this experiment is to determine whether the selection of λ makes a difference to the classification ability of the model. It is not to obtain optimum values for λ and the learning rate. The table shows the average precision over five runs of the model and error is estimated using standard deviation. The validation set here is the set of instances with target 3 or 5 which fall within the 5,000 instances following the training subset. It was expected that this experiment would display that the value of λ is irrelevant with a correctly tuned training rate. Arguably, the results show this to be the case. Although there are differences in the accuracy achieved when altering values of λ, these are small. Figure 4.9

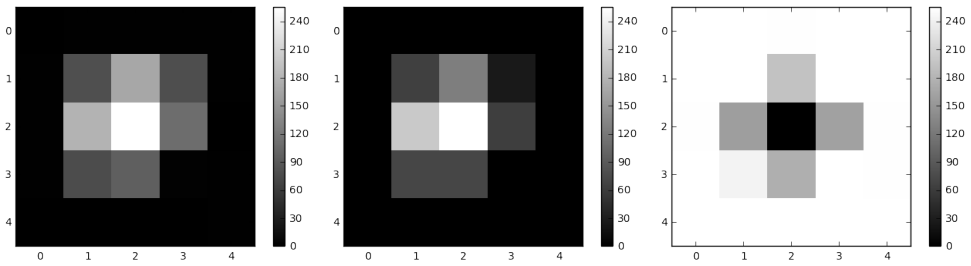

Figure 4.9: A visual depiction of the filters learned by a classifier using DFO_1 regularisation for a selection of values of λ. Left: $\lambda = 0.01$, middle: $\lambda = 0.1$, right: $\lambda = 1$.

shows an example of the filters which are learned by each of the values of λ using the highest performing learning rate. Each filter is similar in appearance, although values in the $\lambda = 1$ filter are negative, hence its inverted appearance. The K-CSVM with $\lambda = 1$ did not learn correctly for the larger learning rates due to the issue discussed previously whereby filter values oscillate between positive and negative, each time growing in value. When using a learning rate of 0.01, the weight update of the first iteration was greater than the size of the filter, thus causing an inversion, however, the size of the inverted filter was not sufficient for the second iteration to return filter values to positive. From this point, the filter remained negative. In this case, the learning rate is too large for the selected value of λ.

To conclude, the choice of λ does not appear to be significant, as long as care is taken to tune this alongside the learning rate so that convergence occurs. This is true for each of the regularisation types tested as part of this work. The filters learned when using the DFO_p regularisations developed in this work are similar in appearance to those learned using p-norm regularisation within the same family. However, when using regularisation with a different p, the filters are different in appearance. For example, the filter learned when using L_2 regularisation will likely appear different than when using L_1 regularisation.

The next parameter which required tuning was C. C is a regularisation parameter of the internal SVM which controls the width of the margin. A large C will result in a narrow margin with few instances selected as support vectors, whereas a smaller C increases the margin by increasing the number of support vectors, potentially allowing more instances to fall between the boundaries. It achieves this by limiting the values within $\boldsymbol{\alpha}$ such that $\alpha_i \leq C \quad \forall i$. As briefly discussed in section 4.4.1, it was found that C regularisation was unhelpful on MNIST. The highest achieving classifier had C set sufficiently high that it did not constrain any of the values within $\boldsymbol{\alpha}$. It was found that even without regularisation, a large proportion of the instances are selected as support vectors, and that values within $\boldsymbol{\alpha}$ rarely rose above a value of 10^{-3}. When training with the subset described above, any value of C small enough to restrict the size of values in $\boldsymbol{\alpha}$ was found to decrease accuracy on the validation set. This was true for both the K-CSVM and for an ordinary SVM. This is likely because even the subset of MNIST has a sufficiently large number of instances that it is difficult to overfit when using a low degree polynomial. When using a polynomial degree of four or greater, a light restriction on $\boldsymbol{\alpha}$ gave a small increase in generalisation accuracy, though this accuracy was still lower than using a degree of two and no regularisation. By reducing the size of the subset to include only the 3s and 5s within the first 400 instances, it was possible to gain an increase in the generalisation accuracy by tuning C, even when $d = 2$.

d	DFO_1	DFO_2	SVM
1	93.58 ± 0.17	93.63 ± 0.09	93.67
2	98.32 ± 0.04	98.20 ± 0.02	98.06
3	98.28 ± 0.03	97.65	98.01
4	98.01	97.44	97.60
5	97.39	96.88	96.46

Table 4.3: Cross-validation to find the optimum kernel degree for DFO_1 and DFO_2 regularisation. The classifiers were trained on the MNIST subset containing 1020 instances with a target of 3 and 882 instances with a target of 5. Results are averaged over five runs and error is measured using the standard deviation. The error value is omitted for entries with a standard deviation of 0.

To select a value for the polynomial degree and to choose the regularisation type, cross-validation was used. The results of this are shown in table 4.3. Each classifier was trained on the MNIST subset containing the instances with a target of 3 or 5 which fall within the first 10,000 instances. The validation set contained the instances with equivalent targets which fall within the following 10,000 instances. The results show the average value and the standard deviation calculated over five runs of the model. The experiments for DFO_1 regularisation had $\lambda = 0.01$ and a learning rate of 0.1. DFO_2 regularisation was found to require lower values to converge for all degrees. Instead, these experiments had $\lambda = 0.001$ and a learning rate of 0.01.

4.5 Experimental Results: CIFAR-10

As well as testing on MNIST, this work collects results on the CIFAR-10 dataset (Krizhevsky, 2009). Like MNIST, CIFAR-10 is an image classification benchmark dataset which is freely available to researchers. The instances are of size 32×32 and contain colour photos of real-world objects. The dataset contains 60,000 images, portioned into 50,000 training instances and 10,000 testing instances. The instances are split into ten target classes and there are exactly 6,000 instances per class. As with MNIST, this work uses only two of the CIFAR-10 target classes. CIFAR-10 is a considerably more difficult dataset than MNIST, and therefore it is not necessary to train on the most difficult pairing. Instead, this work uses the first two target classes: aeroplanes and automobiles. To utilise the RGB colour channels, this work takes the simple approach of vertically appending the red, green and blue images into a combined image of size 96 by 32. It does so for each of the classifiers which are tested on CIFAR-10. The convolutional filter of the K-CSVM passes over the entire combined image such that the same filter is used to find features within each of the colour channels.

4.5.1 Results

As with MNIST, this work trained on a subset of the CIFAR-10 dataset due to the extensive training time of the full dataset. Krizhevsky (2009) has split the CIFAR-10 training dataset into five separate data files, each of 10,000 instances. This work trains on only the instances with a target class of aeroplane or automobile which fall within the first of the five data files. This results in 1005 aeroplanes and 974 automobiles.

CIFAR-10 Results		
Model	Training accuracy %	Testing accuracy %
SVM	94.19 ± 0	85.05 ± 0
CNN	94.54 ± 2.09	80.15 ± 1.09
K-CSVM	91.99 ± 0	85.56 ± 0.02

Table 4.4: Accuracy on a binary CIFAR-10 subset containing instances with a target of 'aeroplane' or 'automobile'. Each classifier was trained on a subset containing 1005 aeroplanes and 974 automobiles. Results are averaged over five runs and error is estimated using the standard deviation. The CNN has a single convolutional filter and does not perform subsampling.

The K-CSVM used to obtain results on the CIFAR-10 dataset was similar in structure to the one used on MNIST, although parameter values differed. The highest performing classifier was found to have a polynomial degree of three. It used DFO_2 regularisation and had both η and λ set as 0.1. The filter was of size 5×5. Unlike the classifiers trained on MNIST, the CIFAR-10 classifiers were found to gain benefit by correctly tuning C. However, it was found that the optimal value of C declined through several orders of magnitude during training. Moreover, if the value of C was set too low for the initial iterations then training did not converge successfully. For this reason, C was only utilised on the final iteration, at which point the optimal value was found to be 10^{-11}. Section 4.5.3 shows how this configuration was decided.

The SVM trained for comparison used a polynomial degree of 3, however, it required a larger value for C, with a value of 10^{-8} achieving the greatest accuracy on the validation set. This configuration was found by using cross-validation. Cross-validation assessed the polynomial degree between the values of two and five in increments of one; it selected which values of C to evaluate by determining the point at which C began to restrict the size of values in $\boldsymbol{\alpha}$, then iteratively decremented the exponent of C by one until all of the values in $\boldsymbol{\alpha}$ were restricted. The CNN used to predict CIFAR-10 was equivalent to the CNN described in table 4.1, however, it used 800 nodes in the hidden layer rather than 500 and the value of η was 10^{-4}. Again, cross-validation was used to find these values; it tested hidden layers of size 100 to 1000 in increments of 100, and η values of 10^{-3}, 10^{-4} and 10^{-5}. A number of configurations were tested which employed multiple hidden layers, however, the accuracy achieved by each of these fell short of the accuracy achieved by the highest performing network with a single hidden layer.

As can be seen in table 4.4, the K-CSVM is the highest achieving classifier of those tested. It achieves an average generalisation accuracy of 85.56%. This is a little greater than a SVM and considerably greater than a CNN. Despite having a greater generalisation accuracy, the K-CSVM achieves a decidedly lower training accuracy when compared to the other classifiers. However, this is not a shortcoming. On the contrary, the reduced disparity between training and testing accuracies demonstrates the effectiveness of the convolutional filter for preventing overfitting. Interestingly, the same cannot be said for the CNN, which achieves the greatest average training accuracy, but the lowest generalisation accuracy. Since a convolutional filter is shown to help a SVM, the overfitting has likely been caused by the fully connected network. Thus, the results highlight that switching out the fully connected network for a SVM is a useful tactic for improving the models ability to generalise between training and testing data.

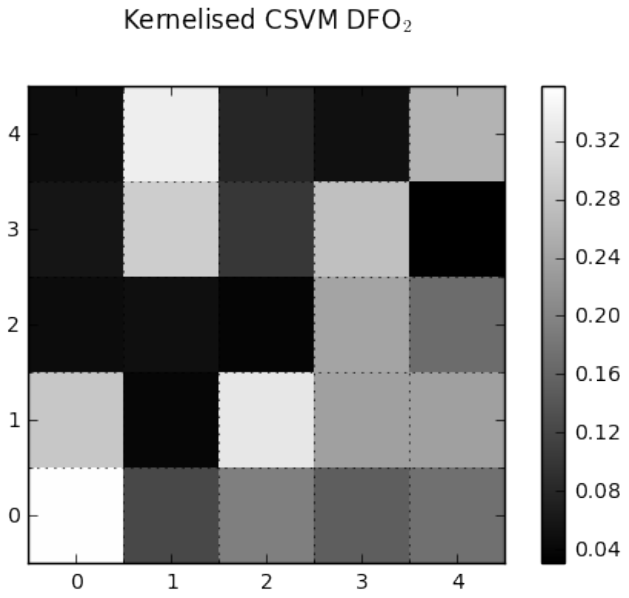

Figure 4.10: A visualisation of the filter trained by the K-CSVM during the experiment used to obtain results for table 4.4. Each of the five runs learned filters which were close to identical.

4.5.2 Analysis

Figure 4.10 shows the filter which was trained by the K-CSVM during the experiment used to obtain results for table 4.4. Over the five runs used to obtain the results, each of the learned filters was close to identical. This is interesting as the classifier used to obtain results on MNIST learned a different filter on each run. One of the main differences between the two experiments is the regularisation type. This experiment used DFO_2 whereas the previous experiment on MNIST used DFO_1 regularisation. The finding that DFO_2 learns more stable filters between runs than DFO_1 regularisation is consistent with the findings in section 4.4.3. Figure 4.11 shows the effect of the filter on four example images. Each image is of size 96×32 and contains the vertical concatenation of the RGB channels; these have each been visualised in grayscale. The learned filter can be recognised as a low-pass filter, however, the blurring is to an even greater degree than the low-pass filter shown in figure 4.4 in the previous section. This is due to the low central value of the filter. The K-CSVM likely benefited from an extreme blur on this dataset because the features are not centred within the image and they are of inconsistent size.

4.5.3 Tuning

Table 4.5 shows the results of cross-validation over the kernel degree and C. Each classifier used DFO_1 regularisation and was otherwise set up using the same configuration as the classifier used to obtain the results in table 4.4. As mentioned previously, the CIFAR-10 training set is split into five data files, each containing 10,000 instances. The set used for training was the first of the data files and the set used for validation was the second of the training files. The results show the

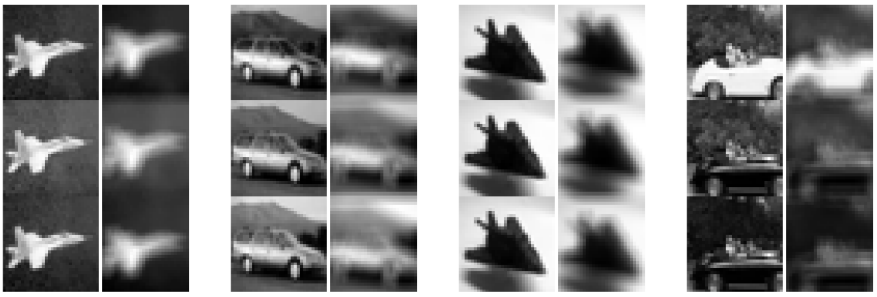

Figure 4.11: Feature maps after convolving with the learned filter. Four images and their corresponding feature maps are displayed. Each image is of size 96×32 and contains the vertical concatenation of the RGB channels. The channels have been plotted in grayscale.

		d			
		2	3	4	5
C	10^{-7}				
	10^{-3}	82.82 ± 0.21			
	10^{-4}	83.22 ± 0.18			
	10^{-5}	82.87 ± 0.09			
	10^{-6}	81.20 ± 0.12			
	10^{-7}		84.61 ± 0.03		
	10^{-8}		83.04 ± 0.03		
	10^{-9}		80.38 ± 0.03	84.36 ± 0.20	
	10^{-10}			84.58 ± 0.36	
	10^{-11}			83.29 ± 0.12	82.22 ± 0.05
	10^{-12}			80.23 ± 0.03	83.98 ± 0
	10^{-13}				83.77 ± 0
	10^{-14}				82.77 ± 0

Table 4.5: Cross-validation to find the optimum values for d and C when using DFO_1 regularisation on the filter. The classifiers were trained on the first of the CIFAR-10 training data files, and validated on the second of the data files.

mean accuracy averaged over three runs and error is estimated using standard deviation. Table 4.6 shows an equivalent cross-validation, however, it uses DFO_2 regularisation. Other than the regularisation, the classifier was set up the same as the DFO_1 classifier. Overall, the results show that the highest validation accuracy is achieved by a classifier using DFO_2 regularisation, d of 3 and C of 10^{-11}. Thus, this was the configuration used to obtain results on the testing sets in table 4.4.

4.6 Conclusion

In this section, a method is proposed which may be used to augment a kernelised SVM with a convolutional filter. The thesis calls the resulting model the K-CSVM and proposes an algo-

		d			
		2	3	4	5
C	10^{-5}	82.47			
	10^{-6}	85.23			
	10^{-7}	84.13			
	10^{-8}	81.87			
	10^{-9}		83.17		
	10^{-10}		84.88		
	10^{-11}		86.04		
	10^{-12}		83.17		
	10^{-13}			83.02	
	10^{-14}			84.28	
	10^{-15}			85.72*	
	10^{-16}			83.88	
	10^{-17}				
	10^{-18}				84.43
	10^{-19}				85.03
	10^{-20}				83.58
	10^{-21}				79.91

Table 4.6: Cross-validation to find the optimum values for d and C when using DFO_2 regularisation on the filter. The classifiers were trained on the first of the CIFAR-10 training data files, and validated on the second of the data files. The results are averaged over five runs, however, the standard deviation was found to be 0 for each value apart from the value marked by an asterisk for which the standard deviation was 0.03.

rithm for training. The algorithm works by casting the learning problem as a Multiple Kernel Learning (MKL) problem. To do this requires that the convolution operation is moved into the kernel function with the filter weights as arguments. In this form, there are many MKL solving algorithms which can be used to optimise both the SVM parameters and the filter weights. This thesis uses the GMKL algorithm. GMKL is able to find globally optimum SVM parameters, while also learning filter weights using gradient descent. It is shown in figure 4.1 that the loss function is correctly minimised during learning. To the knowledge of this author, this is the first research in which convolution has been applied to a kernelised SVM.

The K-CSVM is tested on the MNIST and CIFAR-10 datasets. It achieves a greater generalisation accuracy on both datasets than a SVM or an equivalent CNN with a single convolutional filter. The model learns a low-pass filter on both of the datasets, however, the extent of the blurring differs on each; on MNIST, the filter performs a little blurring, whereas on CIFAR-10 it is drastic.

In this work, the K-CSVM uses a single filter. An opportunity for improvement could be to learn a library of filters, each of which could be used to extract a different feature from within the input. To achieve this, the CMT function could be reformulated so that it satisfies:

$$\mathrm{CMT}(\boldsymbol{x}, \boldsymbol{\mathcal{F}})^{\intercal}\mathrm{vec}(\boldsymbol{\mathcal{F}}) = \mathrm{vec}\left(\begin{bmatrix} \boldsymbol{x} * \boldsymbol{\mathcal{F}}_1 \\ \vdots \\ \boldsymbol{x} * \boldsymbol{\mathcal{F}}_n \end{bmatrix}\right) \quad \text{where} \quad \boldsymbol{\mathcal{F}} = \begin{bmatrix} \boldsymbol{f_1} \\ \vdots \\ \boldsymbol{f_n} \end{bmatrix}. \tag{4.12}$$

This could be accomplished by diagonally arranging the convolutional matrix required for each

$$\mathrm{CMT}(\boldsymbol{x}, \boldsymbol{\mathcal{F}}) = \begin{bmatrix} CMT(\boldsymbol{x}, \mathcal{F}_1) & 0 & 0 \\ 0 & CMT(\boldsymbol{x}, \mathcal{F}_2) & 0 \\ 0 & 0 & CMT(\boldsymbol{x}, \mathcal{F}_n) \end{bmatrix}$$

Figure 4.12: A convolutional matrix to allow convolution using multiple filters. The filters are arranged into diagonal blocks.

filter as shown in figure 4.12. This would likely be more effective when using a filter regularisation in the L_1 family than from the L_2 family because experiments in this chapter have shown that L_2 regularisation learns a similar filter on each run of the model. There would be little benefit to learning multiple filters which all extract the same feature. Alternatively, the K-CSVM could continue using filter regularisation in the L_2 family, but could include a selection of different filter sizes in the hope is that larger filters may pick out different features from smaller filters. Further work is required to determine whether this is an effective strategy.

In addition to arranging filters horizontally, many state-of-the-art CNNs stack filters vertically by using multiple convolutional layers. This helps to improve the translation invariance of the model, meaning that it is more robust to changes in the location of the feature within the inputs. If additional convolutional layers were introduced to the K-CSVM then it would be possible to train them by using backpropagation, similar to the approach taken in (Tang, 2013). To recap, backpropagation is an extension to gradient descent in which the gradient is recursively passed back through a number of layers. All that is required to train using backpropagation is that the gradient may be calculated in the penultimate layer. Since the K-CSVM uses gradient descent to update filter weights, the penultimate gradient is available. Thus, backpropagation is a viable training option. This would likely increase generalisation accuracy, while also reducing the training time, as each convolutional layer reduces the size of the final feature maps. Training time could be further reduced by introducing subsampling layers.

With stacked convolutional layers, the K-CSVM may be able to gain a further improvement in its ability to generalise through the introduction of either dropout or DropConnect. Dropout, described in (Srivastava et al., 2014), is a technique which may be used to guard against overfitting. It is a method of regularisation which works by setting the value of random activations to zero during training. For a convolutional layer, this means setting the activation of random feature map values to zero. DropConnect (Wan et al., 2013b) is a generalisation of dropout, where instead a random subset of connection weights are set to zero. Both dropout and drop connect aim to reduce overfitting by preventing co-adaption between connections. Both methods have achieved state-of-the-art results on a number of benchmark datasets. Although either method may be used with a single convolutional layer, it is likely that a greater benefit would be gained with the increased number of connection weights which would result from stacked convolutional layers.

Chapter 5

Conclusion

In this work, two methods are presented which may be used to utilise convolution during the SVM classification process. The first method may be used to introduce a convolutional filter to a linear SVM, whereas the second method makes it possible to use a kernelised SVM. A training algorithm is presented for each method. The first algorithm, which the thesis names the L-CSVM algorithm, uses an iterative and alternating two-step optimisation to achieve convergence. It first learns the SVM primal weight vector while holding the filter weights fixed, then it learns the convolutional filter with the weight vector fixed. The filter update step is cast as a SVM optimisation problem and may be learned using an arbitrary linear SVM optimisation algorithm. The second algorithm, dubbed the K-CSVM algorithm, formulates the optimisation problem as a MKL optimisation. It is able to learn globally optimal SVM parameters while learning reasonable filter weights by using gradient descent. The algorithm is based on GMKL, but it has been specialised for filter learning.

The L-CSVM and K-CSVM algorithms each show promising results. The L-CSVM is tested on the MNIST dataset and achieves a greater generalisation accuracy than a linear SVM or a linear CNN with a single convolutional filter. It learns a low-pass filter which creates feature maps that are blurred in appearance. The blurring operation makes the model less sensitive to inconsistencies in the location of features within the inputs. As a result, the L-CSVM is able to generalise to unseen data more effectively than a SVM. However, the linear nature of the model is restrictive on MNIST, and as a result, the accuracy is inferior when compared to non-linear algorithms. The K-CSVM algorithm makes use of a kernelised SVM, and therefore achieves non-linearity. It is tested on subsets of the MNIST and CIFAR-10 datasets and is found to learn a low-pass filter on each, though the blurring is to a greater degree on CIFAR-10. It achieves superior generalisation accuracies on both datasets when compared to a kernelised SVM and a CNN with a single convolutional filter.

However, the K-CSVM could not be tested on the full MNIST or CIFAR-10 datasets due to the training time and memory requirements of the algorithm. The focus of this work was on proving the K-CSVM as a concept and little effort was expended in enhancing its efficiency. There are many improvements which could be made to accelerate the algorithm and these could shape the direction of future work. A number of these will be discussed in the following text.

For optimising the filter weights, this work uses the GMKL algorithm. GMKL is able to learn $\boldsymbol{\alpha}^*$ using an arbitrary SVM optimiser, however, the internal kernel parameters, in this case, the filter weights, are optimised using gradient descent. Gradient descent is a well-researched algorithm and many optimisations have been developed to decrease training time. Such optimisations could be introduced to the K-CSVM. For example, in chapters 3 and 4, NAG momentum was found to reduce the number of steps required to train a CNN. It was also found to increase the generalisation accuracy of the final classifier as it provides resilience against local minima. Introducing momentum to the K-CSVM would be no more difficult than for a neural network, and since the K-CSVM converges to a similar solution each time, particularly when using regularisation in the L_2 family, momentum would likely be even more effective for reducing the training time of this model. Another performance optimisation which could have been adopted is to select the step size using an Armijo line search, as in (Varma and Babu, 2009). The dynamic nature of the Armijo rule would likely let the algorithm take greater steps towards the optimum, while also allowing fine-grained weight updates when close to the solution. Alternatively, gradient descent could be replaced altogether by a higher order Newtonian method for selecting direction. For example, in (Jain et al., 2012), gradient descent is replaced with Spectral Gradient Descent (SPG). The resulting algorithm is named SPG-GMKL. SPG uses higher order information to take approximate steps. It is well suited to large-scale problems as it efficiently builds a coarse approximation with little memory overhead. Empirical results of SPG-GMKL show an order-of-magnitude decrease in training time on large datasets. For example, Jain et al. (2012) show that when learning a product of kernels on a dataset of 20,000 training instances, SPG-GMKL converges in 0.66 hours, compared to 18.89 hours for GMKL. This is due to the reduced number of iterations required before convergence. This work decided against using SPG-GMKL as the approximate nature of SPG would have made it more difficult to monitor the progress of the algorithm. The predictability of gradient descent was more valuable to the aims of this work than a decrease in training time. However, now that the K-CSVM has been shown to learn effectively using vanilla gradient descent, SPG-GMKL would likely be a productive avenue for future work to explore.

Another adjustment which could be made to the learning algorithm would be to train on mini-batches at each iteration rather than the full dataset. Currently the entire dataset is used to calculate the gradient per iteration, however, this may be unnecessary. The number of calculations required for the filter update step is in the order of quadratic compared to the number of training instances. In addition, the algorithm must learn a kernelised SVM on each iteration. It is known that the number of iterations required to train LibSVM is greater than linear compared to the number of training instances (Chang and Lin, 2011). Thus, both of the update steps in the K-CSVM would likely gain more than a linear reduction in training time by decreasing the size of training batches. As discussed in section 2.2, mini-batch gradient descent can also be more resistant to local minima than batch gradient descent.

In section 4 a SVM optimisation algorithm named LASVM is briefly described. LASVM is a SVM training algorithm which boasts competitive performance after a single pass over the training data. In (Bordes et al., 2005) the results of LASVM are compared against LibSVM. LibSVM is shown to achieve an error rate of 1.36% and takes 17,400 seconds to train. After a single pass over the training data, LASVM achieves an error rate of 1.42%. This takes 4950 seconds to train. A subsequent pass over the training data brings the error to 1.36% with a training time of 12210 seconds. For the K-CSVM, it is unclear how beneficial a high level of convergence is for the internal optimiser. A lower level of convergence may be adequate during training. If the internal optimiser was LASVM, then perhaps a single pass could be used per

iteration until the final iteration which could use multiple passes.

As well as optimising the efficiency of the algorithm, future work could fine tune the code and the hardware used to execute the code. The code in this work was single threaded and it was run on a machine with a 2.2 GHz Intel Core i7 processor and 16GB of RAM. Many of the modules in the code could be parallelised, for example, in the multiclass problem, the classifiers used to learn different OVO pairs could be trained on separate cores. In addition, the code was executed on a CPU, and so it was not designed with GPU acceleration in mind. Implementing each of these ideas would likely result in a substantial decrease in the required training time.

In section 4.6 of the previous chapter, it is suggested that the K-CSVM would benefit from using stacked convolutional and subsampling layers. Using multiple convolutional layers has been a successful technique for increasing the generalisation accuracy of the CNN on many datasets. The primary reason for this is that each convolutional layer or subsampling layer provides a degree of translation invariance. This effect is cumulative to a point. Using multiple convolutional and subsampling layers would also be advantageous to the training time due to the reduced size of the final feature maps which are provided to the internal SVM. Section 4.6 suggests that backpropagation may be used to learn a K-CSVM with multiple convolutional layers. As well as using multiple layers, the CSVM may benefit from using multiple filters per layer. Each filter can learn to extract a separate feature from the input, thus learning a library of feature maps. Section 4.6 proposes a method to achieve this by adapting the CMT method to arrange the convolutional matrices diagonally.

Overall, this work has shown that augmenting a SVM with a convolutional filter is an effective technique to improve the model's ability to generalise to unseen data. The convolutional filter allows the CSVM to extract spatial information, which is unavailable to a SVM, from input features. Experimental results show this to be beneficial to the generalisation accuracy in both the linear and the non-linear cases. As well as comparing against a SVM, this work contrasts the CSVM with the CNN. Like the CSVM, the CNN is intrinsically able to utilise the spatial information from within its inputs. However, the CNN uses a MLP for prediction and therefore it does not guarantee that an optimal hyperplane is learned. This work shows the L-CSVM and K-CSVM to outperform equivalent CNNs on the MNIST and CIFAR-10 datasets. It would be interesting to see future research in which a K-CSVM with a greater number of convolutional filters and layers is tested using the full extent of these datasets.

Appendices

.1 K-CSVM Derivatives

Section 4.1 in chapter 4 introduces convolutional kernels. These are mercer kernels for which the convolutional filter is integrated as an internal kernel parameter. In this form GMKL may be used to optimise the filter coefficients. To use the learning schemes of the Varma and Babu (2009), the following derivative is required:

$$\frac{\partial \boldsymbol{H}}{\partial g_k} = \frac{\partial\, \boldsymbol{Y}\boldsymbol{K}\boldsymbol{Y}}{\partial\, g_k} \tag{1}$$

where $\boldsymbol{K} \in \mathbb{R}^{N\times N}$ is the matrix of kernels evaluated for each pair of observations, $\boldsymbol{Y}$ is the diagonal matrix whose elements are the labels and g_k are the kernel parameters, in this case the filter coefficients, f_k. The following two sections show how this derivative may be calculated for the polynomial and RBF kernels respectively. However, both sections reach a point where the derivative of a quadratic form $S = \boldsymbol{f}^\intercal \Delta \boldsymbol{f}$ for some $F \times F$ matrix Δ is required. This is calculated here.

One way to calculate the derivative of the quadratic form is to consider S explicitly and to differentiate with respect to f_k. Thus:

$$S = \boldsymbol{f}^\intercal \Delta \boldsymbol{f} \tag{2}$$

$$= \sum_{i=1}^{F}\sum_{j=1}^{F} \Delta_{ij} f_i f_j \tag{3}$$

and

$$\frac{\partial S}{\partial f_k} = \frac{\partial}{\partial f_k}\sum_{i=1}^{F}\sum_{j=1}^{F} \Delta_{ij} f_i f_j \tag{4}$$

$$= \sum_{i=1}^{F}\sum_{j=1}^{F} \Delta_{ij} \frac{\partial f_i f_j}{\partial f_k}. \tag{5}$$

The term $\frac{\partial f_i f_j}{\partial f_k}$ is zero unless $i = k$ or $j = k$. If $i = k$, then the result is $\sum_{j=1}^{F} \Delta_{jk} f_j = [\boldsymbol{f}\Delta^\intercal]_k$, where $[\boldsymbol{z}]_k$ means the kth element of the vector $\boldsymbol{z}$. Similarly, if $j = k$ then the result is $\sum_{i=1}^{F} \Delta_{ik} f_i = [\boldsymbol{f}\Delta]_k$. So

$$\frac{\partial S}{\partial f_k} = \sum_{j=1}^{F} \Delta_{kj} f_j + \sum_{i=1}^{F} \Delta_{ik} f_i \tag{6}$$

$$= [\boldsymbol{f}\Delta^\intercal]_k + [\boldsymbol{f}\Delta]_k \tag{7}$$

Putting the results together for each k gives

$$\frac{\partial S}{\partial \boldsymbol{f}} = \boldsymbol{f}^\intercal(\Delta^\intercal + \Delta) \tag{8}$$

If Δ is symmetric:

$$\frac{\partial S}{\partial \boldsymbol{f}} = 2\boldsymbol{f}^\intercal \Delta \tag{9}$$

.1.1 Polynomial kernel

Here the kernel is

$$K_{\boldsymbol{f}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = (1 + (\boldsymbol{X}_m \boldsymbol{f})^\intercal (\boldsymbol{X}_n \boldsymbol{f}))^d \tag{10}$$

$$= (1 + \boldsymbol{f}^\intercal \boldsymbol{X}_m^\intercal \boldsymbol{X}_n \boldsymbol{f})^d \tag{11}$$

Therefore, using the result (6) with $\Delta = \boldsymbol{X}_m^\intercal \boldsymbol{X}_n$, which is not symmetric, gives

$$\frac{\partial}{\partial f_k} K_{\boldsymbol{f}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = d\,(1 + \boldsymbol{f}^\intercal \boldsymbol{X}_m^\intercal \boldsymbol{X}_n \boldsymbol{f})^{d-1} \frac{\partial}{\partial f_k} (1 + \boldsymbol{f}^\intercal \boldsymbol{X}_m^\intercal \boldsymbol{X}_n \boldsymbol{f}) \tag{12}$$

$$= d\,(1 + \boldsymbol{f}^\intercal \boldsymbol{X}_m^\intercal \boldsymbol{X}_n \boldsymbol{f})^{d-1} \left[\boldsymbol{f}^\intercal (\boldsymbol{X}_m^\intercal \boldsymbol{X}_n + \boldsymbol{X}_n^\intercal \boldsymbol{X}_m)\right]_k \tag{13}$$

Therefore

$$\frac{\partial H_{mn}}{\partial f_k} = y_m y_n d\,(1 + \boldsymbol{f}^\intercal \boldsymbol{X}_m^\intercal \boldsymbol{X}_n \boldsymbol{f})^{d-1} \left[\boldsymbol{f}^\intercal (\boldsymbol{X}_m^\intercal \boldsymbol{X}_n + \boldsymbol{X}_n^\intercal \boldsymbol{X}_m)\right]_k \tag{14}$$

The most efficient way of constructing $\frac{\partial \boldsymbol{H}}{\partial f_k}$ will be to construct all F of them simultaneously, only computing $\boldsymbol{X}_m^\intercal \boldsymbol{X}_n$ once for each m and n.

.1.2 RBF kernel

The kernel between two observations $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$ is given by

$$K_{\boldsymbol{f}}(\boldsymbol{x}_m, \boldsymbol{x}_n) = \exp\left\{-\gamma \|\boldsymbol{X}_m \boldsymbol{f} - \boldsymbol{X}_n \boldsymbol{f}\|^2\right\} \tag{15}$$

$$= \exp\left\{-\gamma (\boldsymbol{X}_m \boldsymbol{f} - \boldsymbol{X}_n \boldsymbol{f})^\intercal (\boldsymbol{X}_m \boldsymbol{f} - \boldsymbol{X}_n \boldsymbol{f})\right\} \tag{16}$$

$$= \exp\left\{-\gamma \boldsymbol{f}^\intercal (\boldsymbol{X}_m - \boldsymbol{X}_n)^\intercal (\boldsymbol{X}_m - \boldsymbol{X}_n) \boldsymbol{f}\right\} \tag{17}$$

$$= \exp\left\{-\gamma \boldsymbol{f}^\intercal \Delta \boldsymbol{f}\right\} \tag{18}$$

where γ is a width parameter and $\Delta = (\boldsymbol{X}_m - \boldsymbol{X}_n)^\intercal (\boldsymbol{X}_m - \boldsymbol{X}_n)$. This can be used to differentiate (18):

$$\frac{\partial K_{\boldsymbol{f}}(\boldsymbol{x}_m, \boldsymbol{x}_n)}{\partial f_k} = \frac{\partial}{\partial f_k} \exp\left\{-\gamma \boldsymbol{f}^\intercal \Delta \boldsymbol{f}\right\} \tag{19}$$

$$= \exp\left\{-\gamma \boldsymbol{f}^\intercal \Delta \boldsymbol{f}\right\} \frac{\partial}{\partial f_k} [-\gamma \boldsymbol{f}^\intercal \Delta \boldsymbol{f}] \tag{20}$$

$$= -\gamma K_{\boldsymbol{f}}(\boldsymbol{x}_m, \boldsymbol{x}_n) \left[\boldsymbol{f}^\intercal \Delta^\intercal + \boldsymbol{f}^\intercal \Delta\right]_k \tag{21}$$

Therefore

$$\frac{\partial H_{mn}}{\partial f_k} = -2\gamma y_m y_n K_{\boldsymbol{f}}(\boldsymbol{x}_n, \boldsymbol{x}_m) [\boldsymbol{f}^\intercal \Delta(\boldsymbol{x}_n, \boldsymbol{x}_m)]_k \tag{22}$$

Again, the results of each $\boldsymbol{X}_m^\intercal \boldsymbol{X}_n$ should be stored so that they are only computed once.

Bibliography

(1909). Xvi. functions of positive and negative type, and their connection the theory of integral equations. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 209(441-458):415–446.

Bordes, A., Ertekin, S., Weston, J., and Bottou, L. (2005). Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6(Sep):1579–1619.

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

Burges, C. J. and Schölkopf, B. (1997). Improving the accuracy and speed of support vector machines. In *Advances in neural information processing systems*, pages 375–381.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27.

Chapelle, O., Vapnik, V., Bousquet, O., and Mukherjee, S. (2002). Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1):131–159.

Cristianini, N. and Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314.

Fletcher, R. (1987). Practical methods of optimization. *Perth Meridien Observations*.

Griffin, G., Holub, A., and Perona, P. (2007). Caltech-256 object category dataset.

Huang, F. J. and LeCun, Y. (2006). Large-scale learning with svm and convolutional for generic object categorization. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 284–291. IEEE.

Jain, A., Vishwanathan, S. V., and Varma, M. (2012). Spg-gmkl: generalized multiple kernel learning with a million kernels. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 750–758. ACM.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif. University of California Press.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Effiicient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK. Springer-Verlag.

LeCun, Y. et al. (1989). Generalization and network design strategies. *Connectionism in perspective*, pages 143–155.

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

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Platt, J. (1998). Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report.

Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. 4:1–17.

Rakotomamonjy, A., Bach, F. R., Canu, S., and Grandvalet, Y. (2008). Simplemkl. *Journal of Machine Learning Research*, 9(Nov):2491–2521.

Rosenblatt, F. (1957). *The Perceptron, a Perceiving and Recognizing Automaton Project Para.* Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Schaul, T., Zhang, S., and LeCun, Y. (2013). No more pesky learning rates. In *International Conference on Machine Learning*, pages 343–351.

Schölkopf, B., Herbrich, R., and Smola, A. (2001). A generalized representer theorem. In *Computational learning theory*, pages 416–426. Springer.

Shalev-Shwartz, S., Singer, Y., Srebro, N., and Cotter, A. (2011). Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30.

Skymind (2014). Deeplearning4j: Open-source, distributed deep learning for the jvm. `http://deeplearning4j.org/`. accessed 09-12-2017.

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

Tang, Y. (2013). Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*.

Varma, M. and Babu, B. R. (2009). More generality in efficient multiple kernel learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1065–1072. ACM.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013a). Regularization of neural networks using dropconnect. In Dasgupta, S. and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1058–1066. JMLR Workshop and Conference Proceedings.

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. (2013b). Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pages 1058–1066.

Werbos (1974). Beyond regression : new tools for prediction and analysis in the behavioral sciences /.

Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In Drenick, R. F. and Kozin, F., editors, *System Modeling and Optimization*, pages 762–770, Berlin, Heidelberg. Springer Berlin Heidelberg.