Oxford Internet Institute, University of Oxford

Assignment Cover Sheet

Candidate Number	1037074
Assignment	Machine Learning
Term	Michaelmas
Title/Question	Machine Learning Summative
Word Count	4,432

By placing a tick in this box ☑ I hereby certify as follows:

(a) This thesis or coursework is entirely my own work, except where acknowledgments of other sources are given. I also confirm that this coursework has not been submitted, wholly or substantially, to another examination at this or any other University or educational institution;

(b) I have read and understood the Education Committee's information and guidance on academic good practice and plagiarism at `https://www.ox.ac.uk/students/academic/guidance/skills?wssl=1`.

(c) I agree that my work may be checked for plagiarism using Turnitin software and have read the Notice to Candidates which can be seen at: `http://www.admin.ox.ac.uk/proctors/turnitin2w.shtml`, and that I agree to my work being screened and used as explained in that Notice;

(d) I have clearly indicated (with appropriate references) the presence of all material I have paraphrased, quoted or used from other sources, including any diagrams, charts, tables or graphs.

(e) I have acknowledged appropriately any assistance I have received in addition to that provided by my [tutor/supervisor/adviser].

(f) I have not sought assistance from a professional agency;

(g) I understand that any false claims for this work will be reported to the Proctors and may be penalized in accordance with the University regulations.

Please remember:

- To attach a second relevant cover sheet if you have a disability such as dyslexia or dyspraxia. These are available from the Higher Degrees Office, but the Disability Advisory Service will be able to guide you.

Machine Learning Summative

1037074

1 Introduction

In this document, I conduct a number of machine learning analyses on the MNIST dataset, which is a database of 70,000 black and white images of handwritten digits from 0 through 9. Below, I will conduct a principal component analysis (PCA), train and test support vector machines (SVMs), and train and test multiple neural networks using the MNIST data.

2 Question 1: Principal component analysis

PCA is a statistical procedure for reducing the dimensionality of a feature space to a set of orthogonal (and therefore linearly independent/uncorrelated) components. We do this using singular value decomposition and find components such that our first component explains the greatest amount of variance in the dataset, the second explains the greatest possible amount of the remaining variance, and so on. PCA can produce as many principal components as there are features in the original dataset, although, in practice, we often only use the first few components (see explanation and plots below).

PCA can be used to visualize high-dimensional data more clearly, such as by plotting the data by its loadings on the first two or three components. In certain contexts, the principal components may have a theoretical interpretation, though this is not the case with the MNIST data.

2.1 Data preprocessing

First, I import necessary packages. I then load in the MNIST dataset from Keras, which consists of pixel data for images of 60,000 hand-drawn digits. It is already split into train and test sets in Keras. Each pixel is an RGB value in the range [0, 255], so I divide each pixel by 255 to ensure that we have values between 0 and 1. For later use in the neural network, I create a version of the training and testing labels

(y_train and y_test) with one-hot encoding (that is, the categorical number labels are transformed into a set of binary variables indicating whether a given image is or isn't 1, 2, 3, et cetera).

2.2 PCA

I then flatten the data from a three-dimensional array (of dimension 60,000 x 28 x 28) into a two-dimensional one (of dimension 60,000 x 784) for use in PCA and fit the PCA to the reshaped training data. I then standardize the data to have a mean of zero and a variance of one. This is standard practice for PCA to avoid misleading principal components that occur because one variable has a larger variance than others (which can occur because variables are on different measurement scales, although this isn't the case here) (Murphy 2012: 389). Next, I run a PCA with the maximum number of components (i.e. 784, the total number of pixels in each image) and identify a more optimal number of components using the next set of plots.

2.2.1 Plots:

Figure 1 shows the proportion of total variance in the data explained by each of the first 100 principal components. It begins to level off around 30 components. Figure 2 shows the cumulative variance explained by each number of principal components and levels off around 400 components. Note that even the first principal component explains less than 6 percent of the variation in the data.

Figure 1:

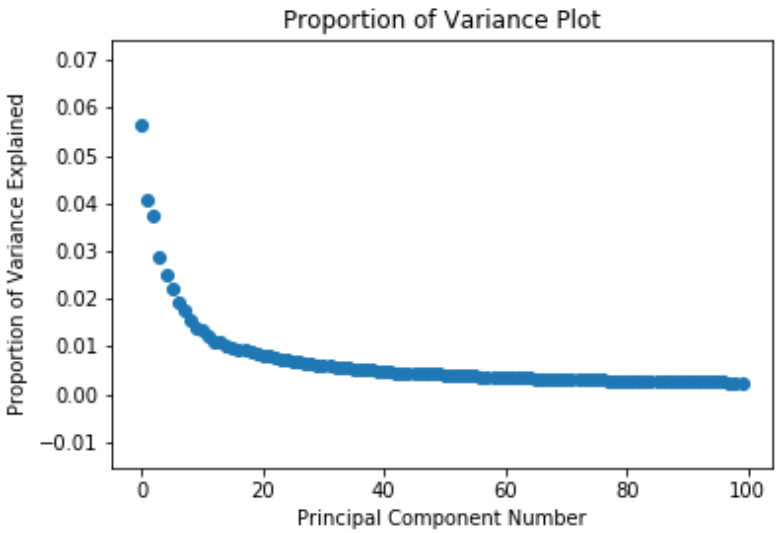
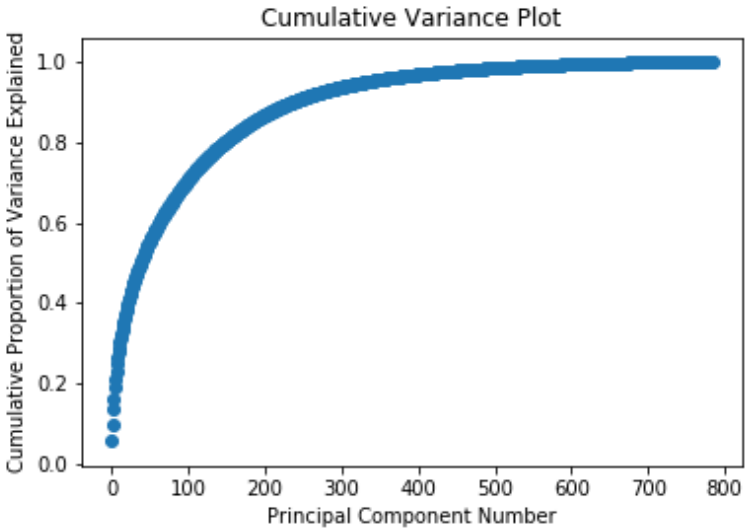
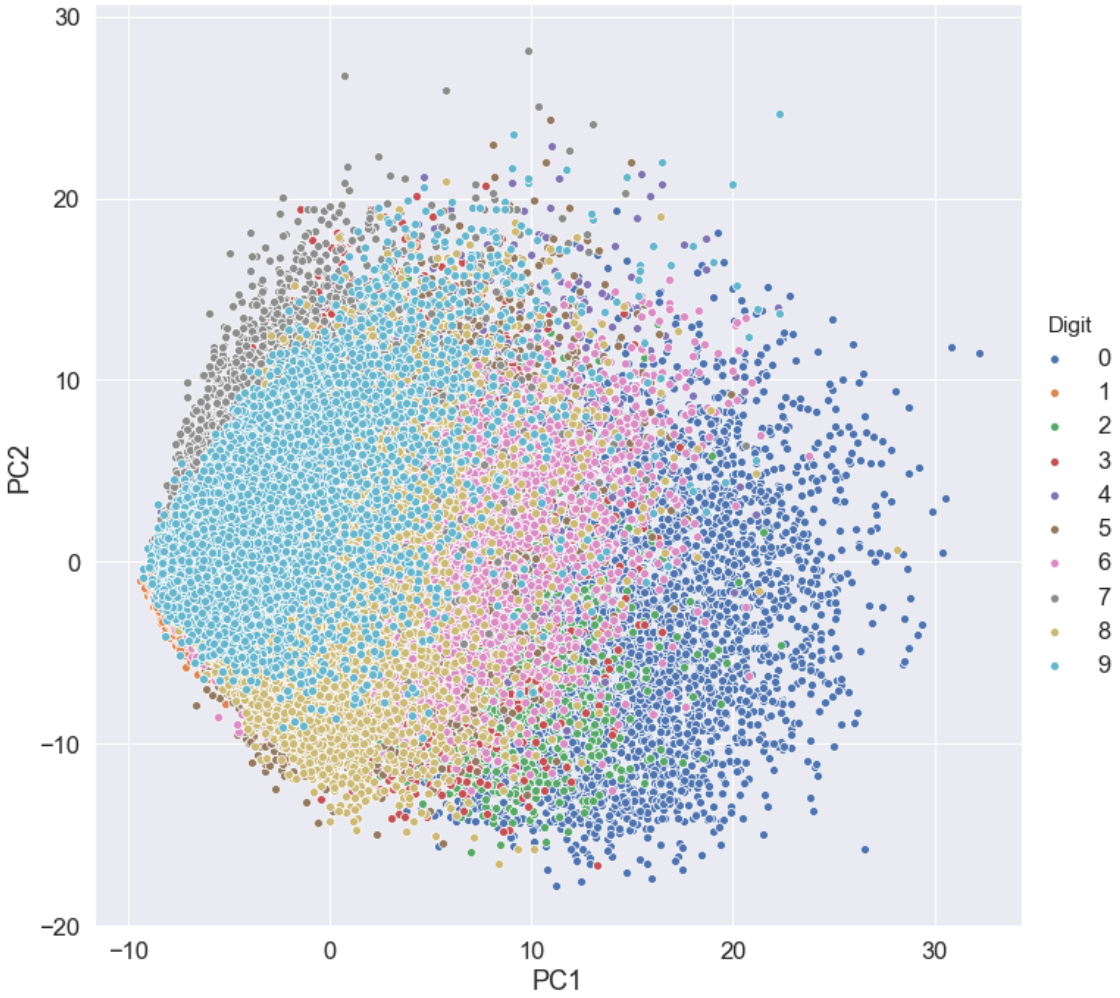


Figure 2:



In Figure 3, I show the original data points plotted on the new orthogonal axes defined by the first two principal components. Given the nature of the dataset—as a set of pixels—the principal components are difficult to interpret theoretically. However, there is clear clustering in Figure 3 for most of the labels. For instance, images of the digit 9 have lower loadings on the first principal component and higher ones on the second, whereas the opposite is true for the digit 0.

Figure 3: PC1 vs. PC2 for the MNIST Data



3 Support Vector Machines

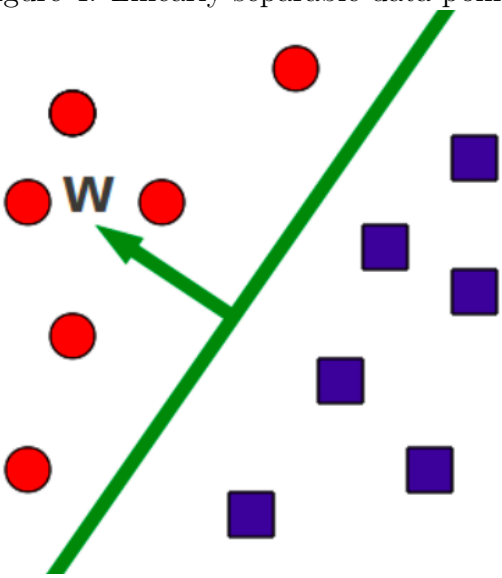
3.1 Theory

Support vector machines (SVMs) separate data into classes by drawing a linear separating hyperplane, or decision boundary, between them. The separating hyperplane is a line in two dimensions, a plane in three, and so on. SVMs can be built by applying the kernel trick to the linear support vector classifiers (SVCs), which in turn arise by relaxing the constraints of a maximum margin classifier. These classifiers are described in greater detail below:

3.1.1 Maximum margin classifiers

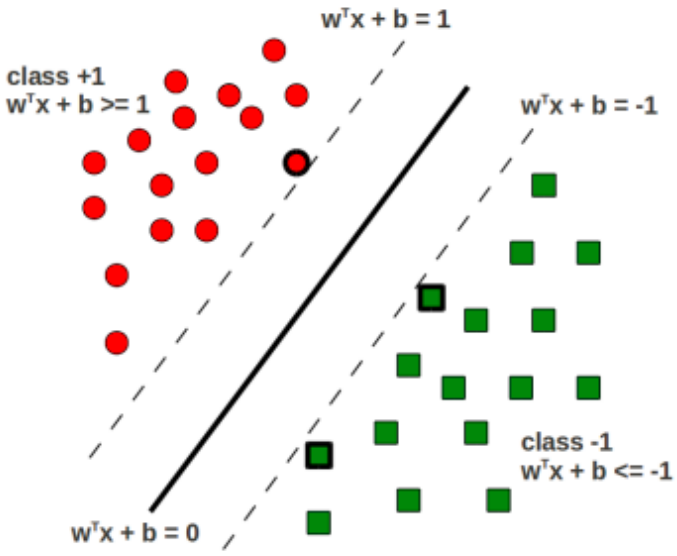
When building a maximum margin classifier, one assumes that data are linearly separable, that is, that there exists a linear separating hyperplane such that all examples with one label (e.g. + 1) will fall one side of it, and all examples with another label (e.g. -1) will fall on the other side. In a D-dimensional feature space, we define our separating hyperplane by an outward-pointing normal vector $\mathbf{w} \in \mathbb{R}^D$, which is orthogonal to any vectors that fall on the hyperplane. Consider the example below (Rai 2011: 1):

Figure 4: Linearly separable data points



We now introduce the concept of the margin, which is shown below. It is denoted by the dotted lines and refers to the area on either side of the separating hyperplane (the solid line) in which no points fall. b is our bias term, and our classification rule is $y = sign(\mathbf{w}^T\mathbf{x} + b)$ (Rai 2011: 2):

Figure 5: Support vectors



To make our decision boundary as generalizable as possible, we pick the hyperplane with the largest margin. The observations which fall on the margin will be called our "support vectors" in a support vector classifier, as they determine (or "support") our decision boundary.

Formally, we can denote our margin as γ, where $\gamma = \frac{1}{\|\mathbf{w}\|}$. We can maximize our margin by minimizing the norm of w. When determining our margin, we minimize:

$$f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \tag{1}$$

subject to the constraint that all of our data points are correctly classified (Rai 2011: 2), i.e.:

$$1 \leq y_n(\mathbf{w}^T\mathbf{x} + b), n = 1 \dots N \tag{2}$$

Note that, in the equation above, we have N total constraints (one for each data point). To solve this optimization problem, we introduce a Lagrange multiplier for each constraint. We will define this as $\alpha_n, n = 1 \dots N$. We now arrive at the primal Lagrangian L_p below (Rai 2011: 3). We'll use this to find our dual Lagrangian, which we'll maximize subject to a set of constraints to arrive at our decision boundary. We want to minimize:

$$L_p(\mathbf{w}, b, \alpha) = \frac{\|\mathbf{w}\|^2}{2} + \sum_{n=1}^{N} \alpha_n\{1 - y_n(\mathbf{w}^T\mathbf{x}_n + b)\} \tag{3}$$

subject to the constraint

$$\alpha_n \geq 0, n = 1 \dots N \tag{4}$$

We will now take the partial derivatives of L_p with respect to $\mathbf{w}$ and b and set them equal to zero so that we can substitute them into L_p. First, we consider $\frac{\partial L_p}{\partial \mathbf{w}}$:

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0 \tag{5}$$

$$\mathbf{w} - \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n = 0 \tag{6}$$

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n \tag{7}$$

Now, we consider $\frac{\partial L_p}{\partial b}$:

$$\frac{\partial L_p}{\partial b} = 0 \tag{8}$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \tag{9}$$

Substituting these back into L_p, we arrive at the dual Lagrangian, L_d (Rai 2011:4). We now hope to maximize L_d:

$$L_d(\mathbf{w}, b, \alpha) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m,n=1}^{N} \alpha_m \alpha_n y_m y_n (\mathbf{x}_m^T \mathbf{x}_n) \tag{10}$$

subject to the constraints

$$\sum_{n=1}^{N} \alpha_n y_n = 0, \alpha_n \geq 0, n = 1 \dots N \tag{11}$$

We can now offer a formal definition of our support vectors described above. From our primal Lagrangian L_p, recall that the following will hold for optimal values

of α_n:

$$\alpha_n\{1 - y_n(\mathbf{w}^T\mathbf{x}_n + b)\} = 0 \tag{12}$$

Our support vectors are training examples (vectors) for which the Lagrange multiplier α_n is nonzero, which means that they fall on, and therefore determine, our margin boundaries. If $\alpha_n \neq 0$, then the equation above is only satisfied if:

$$y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1 \tag{13}$$

Since y_n, our class label, is either -1 or 1, this produces our margins:

$$\mathbf{w}^T\mathbf{x}_n + b = 1 \tag{14}$$

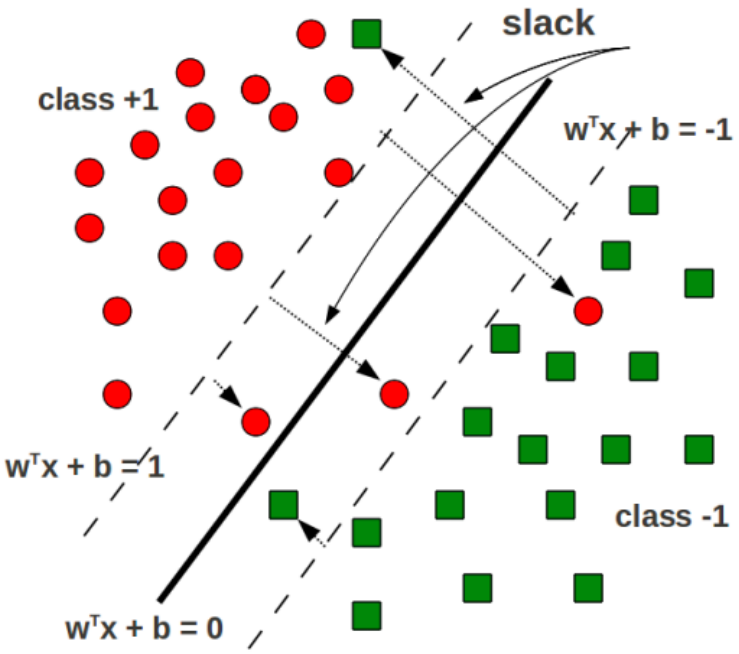
or

$$\mathbf{w}^T\mathbf{x}_n + b = -1 \tag{15}$$

3.1.2 Support vector classifiers:

In practice, our data are rarely completely linearly separable. Therefore, we relax the linear separability assumption and allow decision boundaries where some points fall within the margin and/or on the wrong side of the separating hyperplane. Consider the example below:

Figure 6: Support vector classifier with slack



When training our model (i.e. building our decision boundary), the number of violations and their magnitudes are controlled by variables called the slack variables. The total magnitude of these slack variables is called our budget, C. The resulting classifier is called a support vector classifier.

Formally, we represent this relaxation of the linear separability assumption by adding a slack variable ξ, which represents the distance of a vector $\mathbf{x}$ past the margin (as seen above) A value of $0 < \xi < 1$ indicates that the $\mathbf{x}$ falls within the margin but is correctly classified, while $\xi > 1$ represents a misclassification. Our new constraint is:

$$1 - \xi \leq y_n(\mathbf{w}^T\mathbf{x} + b), n = 1 \dots N \tag{16}$$

We want to minimize the sum of all the slack variables as well as $\frac{\|\mathbf{w}\|^2}{2}$, which leads to our new optimization problem. Our budget C is a constant determining how much we prefer a large margin (smaller C) or fewer misclassifications and examples falling within the margins (larger C):

$$f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} + C\sum_{n=1}^{N}\xi_n \tag{17}$$

subject to the constraint:

$$1 - \xi_n \leq y_n(\mathbf{w}^T\mathbf{x} + b), n = 1 \ldots N, \xi_n \geq 0 \tag{18}$$

We can now introduce a second Lagrange multiplier, β, to arrive at the primal Lagrangian below (Rai 2011: 4):

$$L_p(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{\|\mathbf{w}\|^2}{2} + C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n\{1 - y_n(\mathbf{w}^T\mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n \tag{19}$$

subject to the constraint

$$\alpha_n, \beta_n \geq 0, n = 1 \ldots N \tag{20}$$

As with our maximum margin classifier, we take partial derivatives of L_p with respect to $\mathbf{w}, b$, and ξ_n, set them equal to zero, and substitute them into L_p to derive L_d below (Rai 2011: 4):

$$L_d(\mathbf{w}, b, \xi, \alpha, \beta) = \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{m,n=1}^{N}\alpha_m\alpha_n y_m y_n(\mathbf{x}_m^T\mathbf{x}_n) \tag{21}$$

subject to the constraints

$$\sum_{n=1}^{N}\alpha_n y_n = 0, \alpha_n \geq 0 \geq C, n = 1 \ldots N \tag{22}$$

3.1.3 The kernel trick and support vector machines

Our data are not always linearly separable in the feature space in which they were observed. Even when we relax the constraints of the maximum margin classifier and use an SVC, we may end up with a poorly fitting model. However, our data may be linearly separable in a higher-dimensional feature space. For instance, they may have been observed in $\mathbb{R}^2$ but are only linearly separable in $\mathbb{R}^3$. The data can be transformed into this higher-dimensional space using a function $\phi(\mathbf{x}) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ (where $\mathbf{x} = \{x_1, x_2\}$ is an example in a two-dimensional feature space). Determining the function $\phi(\mathbf{x})$ and computing its value for each point can be computationally expensive, so we instead compute the inner product (the dot product in the example

below) of every vector/data point $\mathbf{x}$ with every other vector/data point $\mathbf{z}$ using a kernel function $k(\mathbf{x}, \mathbf{z})$.

Consider the example in which we have non-linearly separable data in a two-dimensional feature space. Assume that they are linearly separable in the three-dimensional feature space defined by:

$$\phi(\mathbf{x}) = \{x_1^2, x_2^2, \sqrt{2}x_1x_2\} \tag{23}$$

We could compute the dot product of our vectors $\mathbf{x}$ and $\mathbf{z} = \{z_1, z_2\}$ in our three-dimensional feature space by first mapping them directly into that space, which would give us the vector above and

$$\phi(\mathbf{z}) = \{z_1^2, z_2^2, \sqrt{2}z_1z_2\} \tag{24}$$

Taking their dot product, we get:

$$x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \tag{25}$$

Alternatively, we can use a specific case of the polynomial kernel function:

$$k(\mathbf{x}, \mathbf{z}) = (a < \mathbf{x}, \mathbf{z} > +b)^n \tag{26}$$

Setting $a = 1$, $b = 0$, and $n = 2$ gives us the quadratic kernel:

$$k(\mathbf{x}, \mathbf{z}) = (< \mathbf{x}, \mathbf{z} >)^2 \tag{27}$$

Since our original feature space was two-dimensional, we can expand $\mathbf{x}$ and $\mathbf{z}$:

$$< \{x_1, x_2\}, \{z_1, z_2\} >^2 \tag{28}$$

and calculate this inner product, giving us:

$$(x_1z_1 + x_2z_2)^2 \tag{29}$$

We expand this inner product, which results in:

$$x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \tag{30}$$

This is the same as the inner product calculated above by mapping our data into the feature space defined by $\phi(\mathbf{x})$ and then taking the inner product of $\mathbf{x}$ and $\mathbf{z}$.

However, we were able to do so without explicitly defining $\phi(\mathbf{x})$; this process is the kernel trick. Recall that our dual Lagrangian was:

$$L_d(\mathbf{w}, b, \xi, \alpha, \beta) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m,n=1}^{N} \alpha_m \alpha_n y_m y_n (\mathbf{x}_m^T \mathbf{x}_n) \tag{31}$$

subject to the constraints

$$\sum_{n=1}^{N} \alpha_n y_n = 0, \alpha_n \geq 0 \geq C, n = 1 \dots N \tag{32}$$

We can now calculate our final term, $\mathbf{x}_m^T \mathbf{x}_n$ without having to calculate the mapping $\phi(\mathbf{x})$.

3.1.4 Other kernels

For a function $k(\mathbf{x}, \mathbf{z})$ to be a kernel, it must satisfy Mercer's Condition. It must take in vectors $\mathbf{x}$ and $\mathbf{z}$, defined in a feature space $\mathcal{X}$, and compute their similarity in a higher-dimensional space $\mathcal{F}$. Mercer's condition specifies that the space $\mathcal{F}$ must be a Hilbert space, that is, a vector space for which the kernel function $k(\mathbf{x}, \mathbf{z})$ defines a dot product. This is true if $k(\mathbf{x}, \mathbf{z})$ is a positive definite function. Common kernel functions that satisfy this condition include the polynomial kernel discussed above, the trivial linear kernel, which returns an SVC:

$$k(\mathbf{x}, \mathbf{z}) = < \mathbf{x}, \mathbf{z} > \tag{33}$$

and the exponential kernel below, where σ is a parameter that controls the bias/variance tradeoff for our classifier:

$$k(\mathbf{x}, \mathbf{z}) = exp(\frac{- \|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}) \tag{34}$$

I will use the scikit-learn implementation of both of these kernels below to classify the MNIST data.

3.2 SVMs with various kernels

I use five-fold cross-validation to tune the c hyperparameter for a linear kernel and then for a quadratic kernel SVM. Five fold cross-validation is a specific case of k-fold cross-validation, in which the training data are split into k = 5 "folds," and

the model is tested once on each fold, with the other k - 1 = 4 folds being used as training data. This process is executed once for each value of the c hyperparameter to determine which of the given cost values performs the best. Cross-validation was implemented using the GridSearchCV module from scikit-learn. For the linear and quadratic kernels, I tune only the c hyperparameter; for the exponential kernel, I also tune the γ hyperparameter. For c, I tested the values 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, and 100000. For γ, I tested the values 0.0001, 0.001, 0.01, 0.1, and 1. Full results are shown below; they indicate that the exponential kernel was the most accurate of the three. Surprisingly, the quadratic kernel performed slightly worse than the linear kernel, which could be a result of overfitting.

Table 1: SVM Results

Kernel	**c**	γ	**Accuracy**
Linear	0.01	N/A	0.8791
Quadratic	10.0	N/A	0.8727
Exponential	0.0001	100.0	0.8801

4 Neural networks

4.1 Theory

4.1.1 Definitions

The network: We start by considering a l-layer network in which we have an output layer L, hidden layers $L-1$ through 1, and an input layer 0. This network has an activation level $a_i^{(l)}$ on each node, where l denotes the node's layer, and i indexes the nodes within each layer. I will use $a^{(l)}$ to represent a vector of all activations for a given layer. Our activation function will take a scalar input $z_i^{(l)}$ for each node. Each edge in the network has a weight $w_{i,j}^{(l)}$, which represents a weight feeding into a node i in layer l from a node j in layer $l-1$. The vector of weights feeding into a given node is represented as $w_i^{(l)}$. I will use the notation $w^{(l)}$ to represent a vector of all the weights from layer $l-1$ to layer l. Finally, I will denote a bias term, $b_i^{(l)}$, which is a scalar added to the activation of node i in layer l before it is passed on to the next layer. A vector of all biases for a layer l will be represented as $b^{(l)}$, and a vector of values for a single input will be represented as x.

Inputs to the activation functions and forward propagation: Let us return to our activation function. For our output layer, we'll use the softmax activation function s (to account for the possibility of a multiclass classification problem), and for all other layers, we'll use the sigmoid activation function, σ, resulting in:

$$a_i^{(L)} = s(z_i^{(L)}) \tag{35}$$

for the final layer and

$$a_i^{(l)} = \sigma(z_i^{(l)}) \tag{36}$$

for all other layers, where $l < L$. Let us now define our activation input $z_i^{(l)}$. For all layers after the first hidden layer 1, that is, for all layers $l > 1$, we have:

$$z_i^{(l)} = (w_i^{(l)})^T a^{(l-1)} + b_i^{(l)} \tag{37}$$

where $w_i^{(l)}$ represents all the weights that feed into the given node from the previous layer. We transpose the weights so that we can multiply them by the activations. For the first hidden layer, our activations from the previous layer are a vector for one example (the i^{th} example) from the input x, resulting in

$$z_i^{(1)} = (w_i^{(1)})^T x + b_i^{(1)} \tag{38}$$

Our initial pass through these activation functions—starting with the input x and ending with our output layer activation—is called forward propagation.

The cost function: We can define a cost function $C(y, \hat{y})$, where y is a correct label for a given training example, and $\hat{y}$ is the predicted label. A common cost function is the sum of the squared errors (below), where N is the number of units in the output layer L:

$$C(y, \hat{y}) = \frac{1}{2} \sum_{i=1}^{N} (y - \hat{y})^2 \tag{39}$$

$\hat{y}$ is the output of the softmax function for the final layer as defined above, i.e.:

$$\hat{y} = s(z_i^{(L)}) \tag{40}$$

It is worth noting that other cost functions are possible, such as categorical cross-entropy, which I will use in the implementation section. For the purposes of this

derivation, we do not need to specify a particular cost function.

Weight update rules: We now aim to update our weights by moving them in a direction that decreases our cost with regard to our weights. Let us call the rate at which we update our weights our learning rate α. Since we hope to move our costs in a negative direction, we arrive at the following general update rule for weights connecting nodes in layer l to nodes in layer $l-1$:

$$\Delta w^{(l)} = -\alpha \frac{\partial C}{\partial w^{(l)}} \tag{41}$$

We will now consider three cases for our weight updates: updating our weights in the output layer, updating our weights in any hidden layer after the first one, and updating our weights in the first hidden layer. We separate our derivation into these three cases because our activation function differs for our output layer compared to our hidden layers (softmax vs. sigmoid) and because the input to our activation function, $z_i^{(l)}$ differs for our first hidden layer compared to our other layers (i.e. it includes the input vector x rather than the activations $a^{(l-1)}$).

4.1.2 Weight update for a weight into the output layer

Let us recall our general update rule from (41) and apply it to layer L. That is, we want to find our update rule:

$$\Delta w^{(L)} = -\alpha \frac{\partial C}{\partial w^{(L)}} \tag{42}$$

Using C for our cost function, equation (40) for our predicted values $\hat{y}$, and equation (37) for our activation input $z^{(L)}$, we apply the chain rule and derive the following update rule:

$$\Delta w^{(L)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} \tag{43}$$

4.1.3 Weight update for a weight into an inner hidden layer

Now, we consider the case in which we update the weights in any hidden layers before the first one. Note that we will only apply this rule in networks with more than one hidden layer because, in a single hidden-layer network, the weight update

rule above will apply. Specifically, we are looking for the update rule

$$\Delta w^{(1<l<L)} = -\alpha \frac{\partial C}{\partial w^{(l)}} \tag{44}$$

We expand this using the chain rule until we arrive at $w^{(l)}$:

$$\Delta w^{(l)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \cdots \frac{\partial z^{(l)}}{\partial w^{(l)}} \tag{45}$$

We rewrite the inner term as a product such that we have:

$$\Delta w^{(l)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{l+1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) \frac{\partial z^{(l)}}{\partial w^{(l)}} \tag{46}$$

Note that $\frac{\partial z^{(l)}}{\partial w^{(l)}} = a^{(l-1)}$, so we can rewrite this as:

$$\Delta w^{(l)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{l+1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) a^{(l-1)} \tag{47}$$

4.1.4 Weight update for a weight into the first hidden layer

Finally, we consider the case in which we update a weight in the first hidden layer, $l = 1$, and derive our update rule. This will look nearly identical to our derivation above, with the exception of our final term:

$$\Delta w^{(1)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \cdots \frac{\partial z^{(1)}}{\partial w^{(1)}} \tag{48}$$

Again, we rewrite this as a product:

$$\Delta w^{(1)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) \frac{\partial z^{(1)}}{\partial w^{(1)}} \tag{49}$$

However, there is one key difference: our final term, $\frac{\partial z^{(1)}}{\partial w^{(1)}}$, can be rewritten as x, per equation (38), resulting in our final update rule:

$$\Delta w^{(1)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) x \tag{50}$$

4.1.5 Summary

To summarize, our final three update rules are:

$$\Delta w^{(L)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} \tag{51}$$

for an update to a weight feeding into the output layer,

$$\Delta w^{(l)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{l+1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) a^{(l-1)} \tag{52}$$

for a weight feeding into a generic hidden layer $l > 1$, and

$$\Delta w^{(1)} = -\alpha \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} (\prod_{i=L}^{1} \frac{\partial z^{(i)}}{\partial a^{(i-1)}} \frac{\partial a^{(i-1)}}{\partial z^{(i-1)}}) x \tag{53}$$

for a weight leading into the first hidden layer. We can train our network using these update rules by initializing our weights to random values, our biases to zeros, and repeating the feedforward process and the backpropagation weight update described here over e epochs. For each epoch e, we will start with the weights and biases from the previous epoch $e - 1$ until we hit a specified cost threshold, which I implement in the next section.

4.2 Neural network class implementation details

I now implement a neural network based on the derivation above. The full code is shown in the attached .ipynb file. In this implementation, I include one hidden layer with 100 neurons as specified in the instructions. I will describe my high-level design choices below; low-level choices are explained in comments within the code.

4.2.1 Activation functions

As in the derivation above, I use the sigmoid activation function and its derivative for non-output layers and the softmax activation function for the output layer. While I did not have to define these functions for the derivation above, I will do so explicitly here. The sigmoid function, $\sigma(z)$ is below:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{54}$$

To take its first derivative, we use the quotient rule:

$$\sigma'(z) = \frac{(0)(1+e^{-z}) - (1)(-e^{-z})}{(1+e^{-z})^2} \tag{55}$$

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} \tag{56}$$

$$(\frac{1}{1+e^{-z}})(\frac{e^{-z}}{1+e^{-z}}) \tag{57}$$

Rewriting our lefthand term as $\sigma(z)$, we get:

$$\sigma'(z) = (\sigma(z))(\frac{e^{-z}}{1+e^{-z}}) \tag{58}$$

We now expand the numerator:

$$\sigma'(z) = (\sigma(z))(\frac{(1+e^{-z}) - 1}{1+e^{-z}}) \tag{59}$$

$$\sigma'(z) = (\sigma(z))(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}) \tag{60}$$

$$\sigma'(z) = (\sigma(z))(1 - \frac{1}{1+e^{-z}}) \tag{61}$$

Rewriting the last term as $\sigma(z)$, we arrive at the sigmoid derivative used in the implementation below:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \tag{62}$$

In our final layer, we use the softmax activation function, which takes in a vector of input activations and returns a probability vector (i.e. a vector in which all the entries sum to 1). We then take the maximum value from this probability vector as our class label. Given an input vector $z^{(L)}$ of activations from the final layer L, the softmax $s(z_i^{(L)})$ output for a single neuron i is:

$$s(z^{(L)})_i = \frac{e^{z_i^{(L)}}}{\sum_{j=1}^{J} e^{z_j^{(L)}}} \tag{63}$$

where J is the number of neurons in the output layer.

4.2.2 Cost function

For my cost function, I use categorical cross-entropy because I have a multiclass classification problem. Given a vector of true labels y and a vector of predicted labels $\hat{y}$, and n classes, categorical cross-entropy $C(y, \hat{y})$ is defined as:

$$C(y, \hat{y}) = -\sum_{i=1}^{N} y_i log(\hat{y}_i) \tag{64}$$

4.2.3 Training and testing

I train the network using stochastic gradient descent. That is, I pass in one training example at a time and use it to update the weights, rather than passing in the entire dataset (batch gradient descent). Stochastic gradient descent has been shown to be more efficient than batch gradient descent because it avoids recomputing the gradient for similar examples before updating the weights and biases (Ruder 2016: 2). Using one example at a time causes fluctuation in the objective function, which can lead to the discovery of potentially better local minima than those found by batch gradient descent (Ruder 2016: 2). When training my neural network, each pass over all of the training data using stochastic gradient descent constitutes a single epoch.

The final trained network is tested using the scikit-learn classification_report function. It achieves a weighted average precision and recall of 0.93, and, consequently, an f1-score of 0.93 as well. Full results are shown in the attached code in the "Neural network testing" section.

4.3 Practical

I now use Keras build a neural network with two hidden layers of 1200 units each. In Keras, a Sequential model consists of a linear stack of layers (Keras 2018). I initialize this model and then add a Flatten layer to reshape the data from (60000, 28, 28) to (60000, 784). I then add a Dense (i.e. fully connected) layer, where every neuron in the input/previous layer is connected to every neuron in the current layer. I use the rectified linear unit, or ReLU, activation function in my dense layers; this function is defined as: $f(x) = x^+ = \max(0, x)$ and has been shown to improve the time required for convergence in stochastic gradient descent by a factor of six relative to the inverse tangent activation function (Krizhevsky et al. 2012: 3).

Next, I add a Dropout layer. Dropout is a method for minimizing overfitting in neural networks by randomly removing units from a layer during training (i.e. setting their weights to zero), resulting in a series of "thinned" networks (Srivastava et al. 2014: 1929). A single neural network, which consists of scaled version of the weights from the thinned networks, is used during testing. This provides an alternative to averaging the outputs of multiple neural networks, which is computationally expensive (Srivastava et al. 2014: 1929). In my Keras implementation, we pick the dropout rate, which specifies the probability that a given neuron is dropped out in a thinned network.

For the output layer, I specify a size of 10 units—one for each of the possible MNIST digit labels—and use the softmax function as my final activation function since I have a multiclass classification problem.

I compile the model using the categorical cross-entropy as my loss function and Adam as my optimizer. Categorical cross-entropy provides a method of measuring the difference between two probability distributions—in this case, y and $\hat{y}$. Our goal when training the network is to minimize this loss function. The Adam optimizer allows us to adjust our learning rate for each parameter using exponentionally weighted moving averages of the mean and variance of our gradients (Ruder 2016: 7).

Finally, I train the model on the 60,000 images in the training set with an 80/20 training/validation split (i.e. 20 percent of the data are held out for validation). After training for 20 epochs—that is, after executing backpropagation 20 times—I achieve 98.43 percent accuracy on the test set.

5 Data science challenge

5.1 Theory and model design

We now attempt to improve on our previous model by creating a convolutional neural network (CNN), a type of neural network that is frequently used in image processing. CNNs introduce the idea of convolutional filters, which can be used to detect features in images, such as curves or edges. Below, I describe the intuition behind feature learning and the types of layers involved in a typical CNN architecture, which we have used in our model.

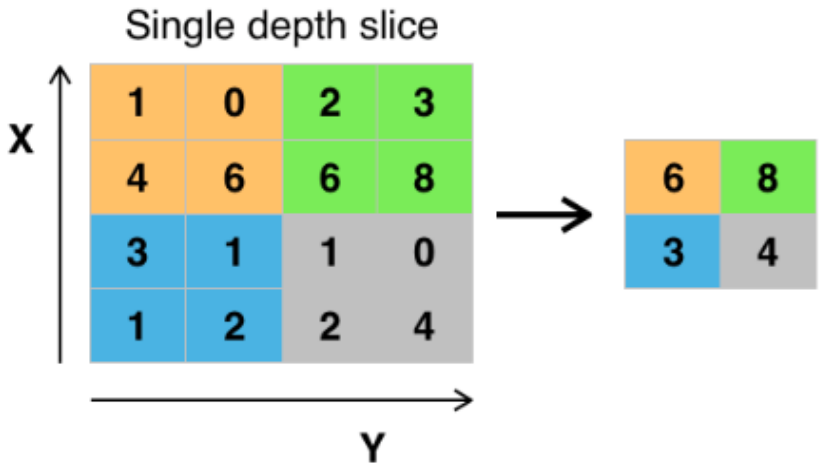
5.1.1 Feature learning layers

Convolution + ReLU: In a convolution + ReLU layer, we apply the convolutional filter described above and then pass the convolved features to an activation function, typically the ReLU (which I will describe below). In a CNN, filters are learned during training—much as our edge weights were above—rather than being explicitly specified. This means that each filter in a layer could learn to detect a slightly different feature, such as an edge or a curve (though features may not always be that interpretable). Once these filters are applied, the values from the convolved feature matrix are used to compute the activation of each neuron using the ReLU function described above, that is, $f(x) = x^+ = \max(0, x)$.

Pooling: Pooling layers are used to prevent overfitting and reduce the number of parameters learned in the model (and thereby the amount of computation required). They do this by downsampling the image, that is, reducing the width and height of each of its tensor representations.

I will now review one of the most common pooling operations, max pooling with a 2×2 filter. We slide a 2×2 square across a depth slice of the image. We take the maximum value in each 2×2 square, as shown below (Zohren 2018: 10):

Figure 7: Max pooling



5.1.2 Classification layers

The classification layers are similar to the layers described in the Keras and hand-coded neural networks described above. Before being classified, the three-dimensional output of the feature learning layers must be passed through a Flatten layer, which reshapes it to a one-dimensional array as in the Keras neural network above. Ex-

amples then pass through a fully connected layer (in Keras, a Dense layer) with the softmax activation function to allow for multiclass classification.

5.2 Data preprocessing

Unlike with our data preprocessing for our neural networks above, we add a dimension to our data rather than flattening it. Instead of having the shape (784, 1), each of our training examples will now have the shape (28, 28, 1), where the first two values represent the width and height of the image in pixels, and the third value represents the number of color channels in our image. This value is usually 3 for RGB images, which have a red, green, and blue color channel; it is 1 here because we have greyscale images and therefore only a black/white color channel.

5.3 Model and performance

When building our CNN, we generally follow the architecture described above. For feature learning, we include a convolutional + ReLU (Conv2D) layer with 32 filters, each of size 5×5, and a max pooling layer with a 2×2 filter. As with our Keras neural network above, we also add a dropout layer to further reduce overfitting. Our classification layers include a Flatten layer and then two dense layers with the same hyperparameters as in our earlier network. Similarly, we compile our model with categorical cross-entropy loss and the Adam optimizer. Our new model achieves an accuracy of 98.93 percent, which is slightly higher than the previous model's performance of 98.43 percent.

References

Keras. (2018). Retrieved from `https://keras.io/`

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017, May). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. Retrieved 2019-01-12, from `http://dl.acm.org/citation.cfm?doid=3098997.3065386` doi: 10.1145/3065386

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective.* Cambridge, MA: MIT Press.

Ruder, S. (2016, September). An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*. Retrieved 2019-01-12, from `http://arxiv.org/abs/1609.04747` (arXiv: 1609.04747)

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (n.d.). Dropout: A Simple Way to Prevent Neural Networks from Overtting. , 30.

Zohren, S. (2018). *Machine Learning: Lecture 8.* University of Oxford.