MNIST Analysis Project

Abdul Z. Abdulrahim

1. Unsupervised Learning

(a) Perform a principal component analysis on the MNIST dataset. Produce various plots in the dimensionally reduced space using the principal components.

Principal Component Analysis (PCA) is a technique used for data exploration and compression in data analysis and machine learning. It reduces the dimensions of observed data by eliminating redundancy using the explained variance of each component. By reducing the dimensionality, we can construct more demonstrative visualisations using low dimensional data and reduce the processing requirements in high dimensional data. This can be thought of as an unsupervised version of multi-output linear regression, where we observe the high-dimensional response y, but not the low-dimensional 'cause' x. Thus the model has the form $x \rightarrow y$; inferring the latent low-dimensional x from the observed high-dimensional y (Robert, 2014).

Using the scikit-learn library, I implement a PCA (see Appendix). This required preprocessing after loading the data (this comes pre-split from the sklearn and keras libraries). First, I reshape the data into a matrix and convert them to floats. These are then normalised on a scale of 0.0 to 1.0. After implementing the PCA, we can check to see how our components explain the data by checking the explained variance.

We can chart the explained variance which tells us how much information (variance) can be attributed to each of the principal components. This is important to determine how much variance is lost as a result of the dimension reduction. We see below in *figure* 1 and 2, using the attributes `pca.explained_variance_` and `pca.explained_variance_ratio`, that the first principal component explains `9.7 percent` of the variance and the second principal component explains `7.1 percent` of the variance. Together, the two components explain `17 per cent` of the information.

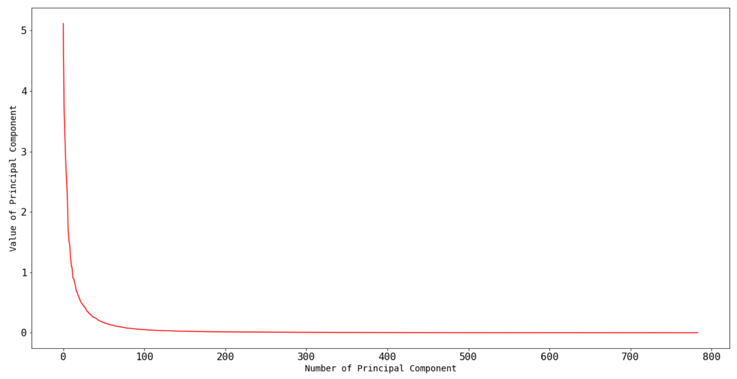

Figure 1: Variance Explained Per Component

Figure 2 below shows the cumulative variance as a function of the number of components. We see that we only reach 85 per cent variance at 58 components, 90 per cent variance at 86 components and 95 per cent variance at 153 components.

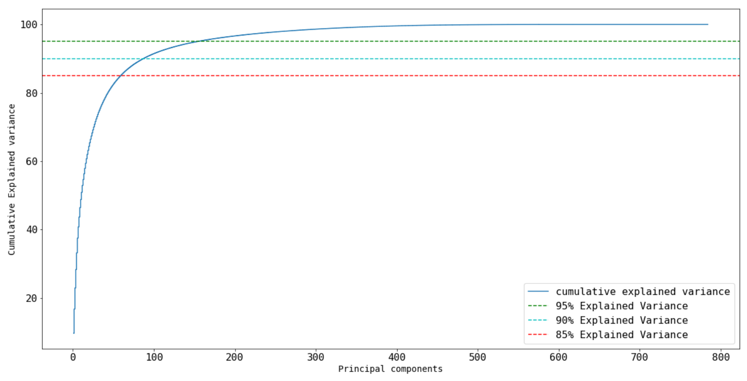

Figure 2: Cumulative Variance as a Function of No. of Components

I also produce various plots below to show how the images are distributed using the first two and second two principal components. *Figure* 3 below provides a plot of all the numbers in a prism spectrum. It gives us a vague idea how well the first two components (left) and the second two components (right) explain the digits in the MNIST data. *Figure* 4 projects the digits with the numbers embedded in the reduced space and shows some of the images plotted in that region. This visualises the performance of the two components. On the left is a plot of the first and the second component, and on the right is a plot of the third and fourth component. We can see each of two components do not sufficiently categorise the numbers, indicating a large number of components would be required to predict the the digits with accuracy and confirming our earlier cumulative variance plot.

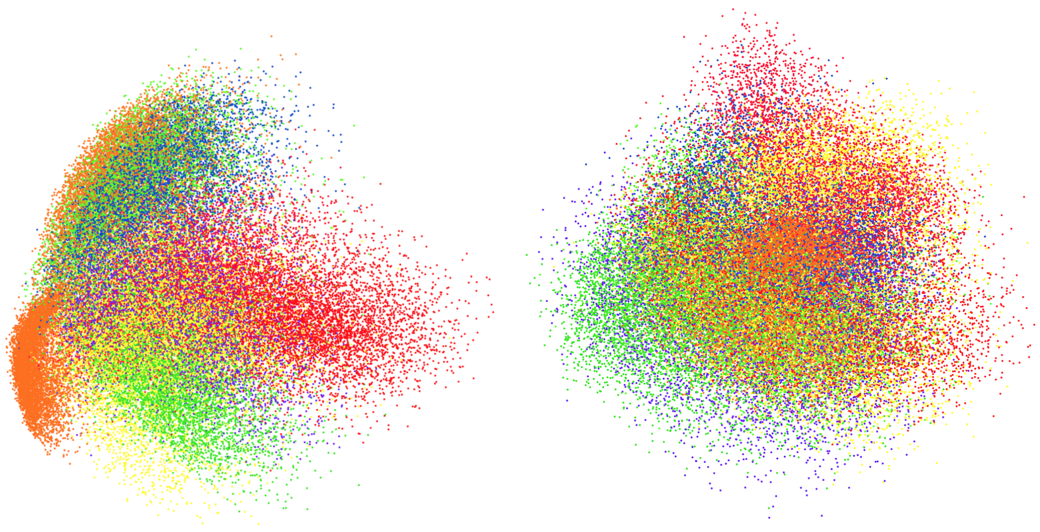

Figure 3: Dimensionally Reduced Projection of Principal Components Using Prism Spectrum

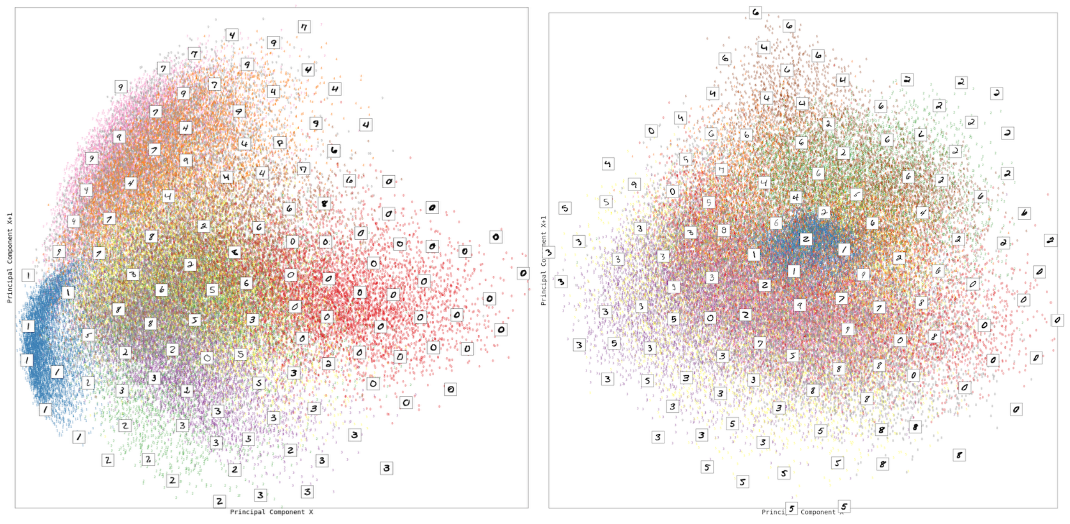

Figure 4: Dimensionally Reduced Projection of Principal Components with Embeddings

2. Support Vector Machines

(a) Theory: Explain either the mathematics or the intuition behind the kernel trick used in SVMs and how it allows to express a non-linear classifier through a linear classifier in a higher-dimensional space.

Intuition

Support Vector Machines (SVMs) are a learning method used for classification. SVMs aim to find a hyperplane which best separates the n-dimensional data into classes. However, as not all data is easily linearly separable, SVM's use the '*kernel trick*', otherwise known as the kernel-induced feature space, which casts the data into a higher dimensional space where the data is separable as shown in *figure* 5. So rather than defining our feature vector in terms of kernels, we can instead work with the original feature vectors, but modify the algorithm so that it replaces all the inner products with a kernel function (Robert, 2014). Luckily for us, many algorithms can be kernelised this way.

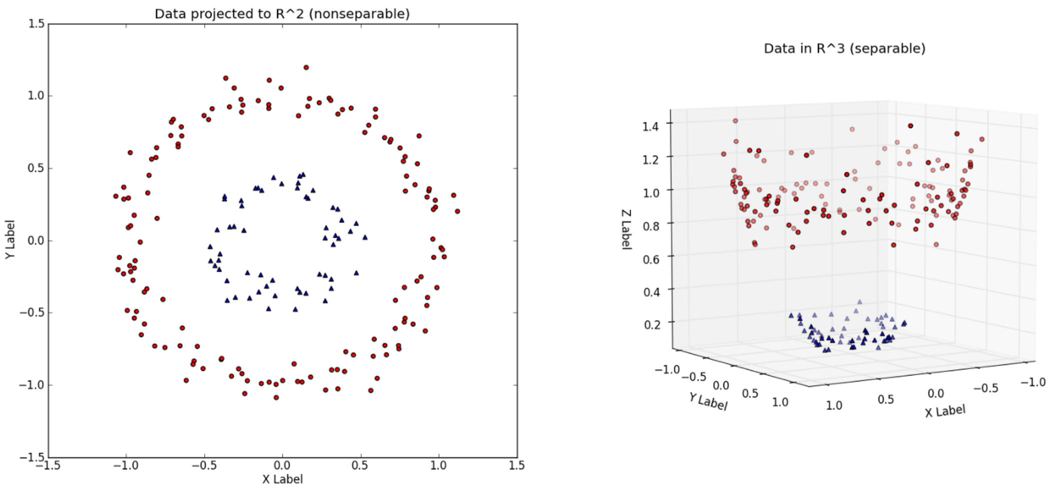

Figure 5: Kernel Induced Feature Space (Kandan, n.d.)

Typically, casting into a higher dimensional space would cause problems computationally but in SVM's this is not dealt with directly as only the formula for the dot-product in that space is needed. Furthermore, it helps to prevent underfitting, i.e., to ensure that the feature vector is sufficiently rich that a linear classifier can separate the data.

Mathematics

Given l training samples $\{\mathbf{x}_i, y_i\}$, $i = 1, \cdots, l$, where each sample has d inputs ($\mathbf{x}_i \in \mathbf{R}^d$), and a class label with one of two values ($y_i \in \{-1, 1\}$), all hyperplanes in $\mathbf{R}^d$ are parameterised by a vector ($\mathbf{w}$), and a constant (b), expressed in the equation

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \tag{1}$$

where $\mathbf{w}$ is the vector orthogonal to the hyperplane.

Given a hyperplane ($\mathbf{w}$,b) that separates the data, this gives the function which classifies the training data and testing data.

$$f(\mathbf{x}) = \mathrm{y}(\mathbf{w} \cdot \mathbf{x} + b)$$

This can also be expressed by all pairs $\{\lambda\mathbf{w}, \lambda b\}$ for $\lambda \in \mathbf{R}^+$ i.e. the *hyperplane* which separates the data by a 'distance' of at least 1.

$$y_i(\mathbf{x_i} \cdot \mathbf{w} + b) \geq 1 \qquad \forall i \tag{2}$$

For a given hyperplane ($\mathbf{w}$,b), all pairs $\{\lambda\mathbf{w}, \lambda b\}$ defines the same hyperplane, but given a data point, each has a different functional distance. We normalise by the magnitude of $\mathbf{w}$ to obtain the geometric distance from the hyperplane to a data point:

$$d\Big((\mathbf{w}, b)\ ,\ \mathbf{x}_i\Big) = \frac{y_i(\mathbf{x_i} \cdot \mathbf{w} + b)}{\parallel \mathbf{w} \parallel} \geq \frac{1}{\parallel \mathbf{w} \parallel}$$

Intuitively, we want the hyperplane that maximises the geometric distance to the closest data points and from the equation we see this is accomplished by minimizing $\parallel \mathbf{w} \parallel$ (subject to the distance constraints). As such, the problem is transformed into:

$$\text{minimize:} \quad W(\alpha) = -\sum_{i=1}^{l} \alpha_i + \tfrac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j (\mathbf{x_i} \cdot \mathbf{x_j}) \tag{3}$$

$$\text{subject to:} \quad \sum_{i=1}^{l} y_i \alpha_i = 0 \tag{4}$$

$$0 \leq \alpha_i \leq C \quad (\forall i) \tag{5}$$

where α is the vector of l non-negative Lagrange multipliers to be determined, and C is a constant. So, the kernel trick is essentially redefining the dot product $\mathbf{x_i} \cdot \mathbf{x_j}$ as a kernel function $K(\mathbf{x_i}, \mathbf{x_j})$ because we only need the result of the dot product. We can now calculate the result of a dot product performed in another space and can change the kernel function in order to classify non-linearly separable data.

There are various ways we could use a kernel function to classify data. Some common examples include:

(a) *Linear*: A linear kernel which works well for text classification

$$K(x_i, x_j) = x_i \cdot x_j$$

(b) *Polynomial*: A polynomial kernel with a constant c and a degree of freedom d

$$K(x_i, x_j) = (x_i \cdot x_j + c)^d$$

(c) *RBF*: A radial basis function (RBF) kernel for more complex decision boundaries with a parameter γ

$$K(x_i, x_j) = exp(-\gamma \parallel x_i - x_j \parallel^2)$$

Note that when $C = \infty$, the optimal hyperplane is the one that completely separates the data. For finite C, this changes the problem to finding a '*soft-margin*' classifier, which allows for some of the data to be misclassified. Finite values of C are useful in situations where the data is not readily separable, perhaps because the input data $\{\mathbf{X}_i\}$ is noisy. It can also be shown that the expected out-of-sample error, Π, is bound by $\frac{N_s}{l-1}$. However, if the data cannot be separated, this is not the case and is a potential setback for SVMs (Robert, 2014).

(b) Practical: Build a linear support vector classifier in Python for the MNIST classification task. Use various kernels to build a better non-linear classifier.

Using the code provided in the Appendix, I build a linear classifier to classify the digits, which does so with an accuracy score of `94.04 percent`. I then implemented grid search using the sklearn library, a process of scanning the parameter space to determine the optimal values for parameters C and gamma (γ) for the RBF kernel (see figure 6 (Sopyła, n.d.)).

This procedure took approximately 3 days to complete on the server but generated `C = 5` and γ `= 0.05` as one of the best parameters, which in turn produce a test score of `98.37 percent`. Similar test scores can be achieved using γ `= 0.05` and `10 or 15` for `C` but due to time constraints this isn't shown here.

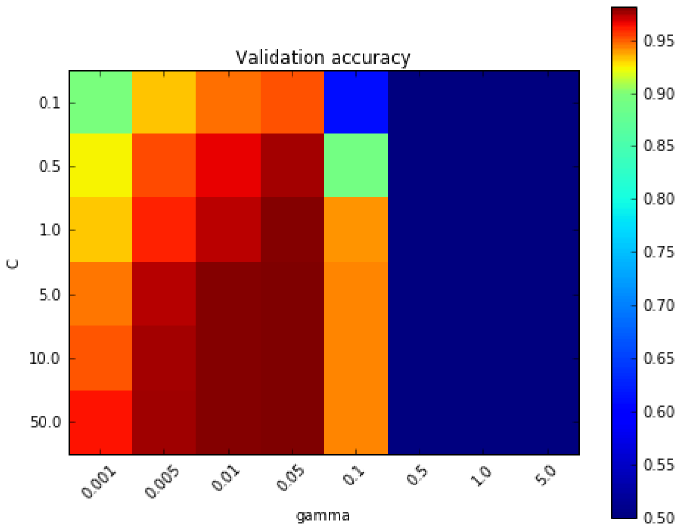

Figure 6: Plot of Grid Search Results for Selected Parameter Space (Sopyla, n.d.)

3. Neural Networks

(a) Theory: By applying the chain law of derivatives to the objective function of a neural network, derive the backpropagation algorithm.

Considering a neural network consisting of L layers, excluding the input layer, an arbitrary layer, ℓ, will have N_ℓ neurons $X_1^{(\ell)}$, $X_2^{(\ell)}$, $\ldots$, $X_{N_\ell}^{(\ell)}$, each with a transfer function $f^{(\ell)}$. The neurons receive signals from the preceding layer, $\ell - 1$ i.e. neuron $X_j^{(\ell)}$ receives a signal from $X_i^{(\ell-1)}$ with a weight factor $w_{ij}^{(\ell)}$. This gives us a $N_{\ell-1}$ by N_ℓ weight matrix, $\mathbf{W}^{(\ell)}$, whose elements are given by $W_{ij}^{(\ell)}$, for $i = 1, 2, \ldots, N_{\ell-1}$ and $j = 1, 2, \ldots, N_\ell$. The neuron $X_j^{(\ell)}$ also has a bias given by $b_j^{(\ell)}$, and an activation $a_j^{(\ell)}$. As such, we have $n_j^{(\ell)} (= y_{\text{in},j})$ to denote the net input into neuron $X_j^{(\ell)}$, which is given by

$$n_j^{(\ell)} = \sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}, \qquad j = 1, 2, \ldots, N_\ell. \tag{1}$$

and the activation function of each neuron $X_j^{(\ell)}$

$$a_j^{(\ell)} = f^{(\ell)}(n_j^{(\ell)}) = f^{(\ell)}(\sum_{i=1}^{N_{\ell-1}} a_i^{(\ell-1)} w_{ij}^{(\ell)} + b_j^{(\ell)}). \tag{2}$$

For any given input vector, we can use these equations to find the activation for each neuron, given a set of weights and biases.

Backpropagation Algorithm

Assuming a set of training pairs (samples and targets): $\mathbf{s}^{(q)} : \mathbf{t}^{(q)}, q = 1, 2, \ldots, Q$, the training vectors are presented one at a time to the network at time t of the training process. So, a training vector $\mathbf{s}^{(q)}$ for a particular q is presented as input, $\mathbf{x}(t)$, to the network. The input can be propagated forward through the network using the equations above to obtain the current set of weights and biases and the corresponding network output, $\mathbf{y}(t)$. These weights and biases are then adjusted for example using the steepest descent algorithm or cross-entropy (*back propagated*) to minimise the error for this training vector. In this case we use the steepest descent algorithm:

$$E = \|\mathbf{y}(t) - \mathbf{t}(t)\|^2, \tag{3}$$

This error E is a function of all the weights and biases of the entire network since $\mathbf{y}(t)$ depends on them. We then need to find the set of updating rules for them based on the steepest descent algorithm:

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha \, \frac{\partial E}{\partial w_{ij}^{(\ell)}(t)}$$

$$b_j^{(\ell)}(t+1) = b_j^{(\ell)}(t) - \alpha \, \frac{\partial E}{\partial b_j^{(\ell)}(t)},$$

where $\alpha (> 0)$ is the learning rate.

To optimise, we compute the partial derivatives of the activations of the last layer, $\mathbf{a}^{(L)}$, and the net input into the $L-$th layer, $\mathbf{n}^{(L)}$. Thus, $\mathbf{n}^{(L)}$ is given by the activations of the preceding layer and the weights and biases of layer L, which gives the explicit relation:

$$E = \|\mathbf{y} - \mathbf{t}(t)\|^2 = \|\mathbf{a}^{(L)} - \mathbf{t}(t)\|^2 = \|f^{(L)}(\mathbf{n}^{(L)}) - \mathbf{t}(t)\|^2$$

$$= \left\| f^{(L)} \left(\sum_{i=1}^{N_{L-1}} a_i^{(L-1)} w_{ij}^{(L)} + b_j^{(L)} \right) - \mathbf{t}(t) \right\|^2 . \tag{4}$$

It is then possible to compute the partial derivatives of E with respect to the elements of $\mathbf{w}^{(L)}$ and $\mathbf{b}^{(L)}$ using the chain rule for differentiation. First, we tackle $\mathbf{w}^{(L)}$:

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = \sum_{n=1}^{N_L} \frac{\partial E}{\partial n_n^{(L)}} \frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}}. \tag{5}$$

To apply the chain rule and obtain the sum of the equation, we first start by defining the sensitivity vector for a general layer ℓ to have components

$$s_n^{(\ell)} = \frac{\partial E}{\partial n_n^{(\ell)}} \qquad n = 1, 2, \ldots, N_\ell.$$

This is called the sensitivity of neuron $X_n^{(\ell)}$ as it gives the change in the output error, E, per unit change in the net input it receives (Lee & To, 2010). Computing the sensitivity vector directly for layer L we get

$$s_n^{(L)} = 2 \left(a_n^{(L)} - t_n(t) \right) \dot{f}^{(L)}(n_n^{(L)}), \qquad n = 1, 2, \ldots, N_L.$$

where $\dot{f}$ denotes the derivative of the transfer function f.

Now, since

$$n_n^{(\ell)} = \sum_{m=1}^{N_{\ell-1}} a_m^{(\ell-1)} w_{mn}^{(\ell)} + b_n^{(\ell)}, \qquad j = 1, 2, \ldots, N_\ell,$$

we can derive

$$\frac{\partial n_n^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial}{\partial w_{ij}^{(L)}} \left(\sum_{m=1}^{N_{L-1}} a_m^{(L-1)} w_{mn}^{(L)} + b_n^{(L)} \right) = \delta_{nj} a_i^{(L-1)}.$$

Giving us

$$\frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \delta_{nj} a_i^{(\ell-1)},$$

We therefore have get the partial derivative of $\mathbf{w}^{(L)}$ as

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} \, s_j^{(L)}.$$

Similarly for the biases $\mathbf{b}^{(L)}$,

$$\frac{\partial E}{\partial b_j^{(L)}} = \sum_{n=1}^{N_L} \frac{\partial E}{\partial n_n^{(L)}} \frac{\partial n_n^{(L)}}{\partial b_j^{(L)}}, \tag{6}$$

and since

$$\frac{\partial n_n^{(L)}}{\partial b_j^{(L)}} = \delta_{nj},$$

we get

$$\frac{\partial E}{\partial b_j^{(L)}} = s_j^{(L)}.$$

Therefore, for a general layer, ℓ, we get the equations:

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial w_{ij}^{(\ell)}}.$$

$$\frac{\partial E}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} \frac{\partial E}{\partial n_n^{(\ell)}} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}} = \sum_{n=1}^{N_\ell} s_n^{(\ell)} \frac{\partial n_n^{(\ell)}}{\partial b_j^{(\ell)}}.$$

and as such:

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = a_i^{(\ell-1)}\, s_j^{(\ell)}, \qquad \frac{\partial E}{\partial b_j^{(\ell)}} = s_j^{(\ell)}. \tag{7}$$

Finally, the updating rules for the weights and biases are (putting back the dependency on the step index t)

$$w_{ij}^{(\ell)}(t+1) = w_{ij}^{(\ell)}(t) - \alpha\, a_i^{(\ell-1)}(t)\, s_j^{(\ell)}(t),$$

$$b_j^{(\ell)}(t+1) = b_j^{(\ell)}(t) - \alpha\, s_j^{(\ell)}(t).$$

(b) Coding: Implement from scratch, as a Python class, a feedforward neural network with 1 hidden layer of 100 hidden units merely using basic linear algebra computations in Numpy. Based on your derivation of back-propagation, or otherwise, implement a train method based on stochastic or batch gradient descent. Test your implementation on the MNIST classification task.

Using the hand-written neural network provided in the Appendix, we train the network to predict our digits. The network is implemented as follows:

(a) *Feed-Forward*: Taking images as a matrix input (i.e. array of numbers that represent digits), we multiply the input by a set of weights using matrix multiplication and apply a sigmoid activation function.

(b) *Compute Loss*: The error/loss is calculated by taking the difference (log-likelihood) of the expected output and the predicted output. In this case, I use categorical cross-entropy (multiclass loss function) which predicts for n classes their hypothetical occurrence probabilities. This is used to alter the weights.

(c) *Back Propagation*: Our weights are then adjusted slightly according to the error.

I also implement a mini-batch descent using an additional '*for loop*' inside the pass-through for each epoch. At each pass, the training set is randomly shuffled then iterated through in chunks. Lastly, to add momentum, we keep a moving average of our gradients instead of updating our parameters (Trevor, Robert, & JH, 2009; Weisberg, n.d.).

After training the network with 12 epochs, we can achieve an accuracy of `98 percent`, with a loss of `0.08`, performing better than the linear SVM and closely matching the RBF SVM.

(c) Practical: Use TensorFlow (potentially using a frontend such as tflearn or keras) to implement a 2 layer neural network with 1200 and 1200 hidden units. Use RELU activation function instead of sigmoids, as well as add regularisation methods such as dropout to produce a model which has a test error of below 2%.

Now using keras, we train a 2-layer neural network using the specifications outlined in the question to predict the digits. I use the RELU activation function instead of sigmoids, and for regularisation, I use dropout to ignore randomly selected neurons. We get better performance at `98.46 per cent` accuracy and a loss of `0.06`, which falls just below the test error threshold of `2 percent`.

In my implementation which can be seen in the Appendix, the dropout regularisation is used to reduce overfitting as it forces the model to learn multiple independent representations of the same data by randomly disabling neurons in the learning phase (Katariya, n.d.). This is arbitrarily set to 20 per cent of the neurons. As for the last layer, softmax activation is used. Softmax, an extension of the multiclass function, allows us to calculate the output of the network based on the probabilities. Each class of the digits is assigned a probability, and the class with the maximum probability is the model's output for the input. Finally, I use the

Adadelta as my gradient descent optimiser. Adadelta is an extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done.

4. Data Science Challenge

This is a problem to challenge you. Any reasonable attempt will grant you full points. For this problem (and only for this problem) you are allowed to work in groups of up to 3 students handing in the same solution. The idea is that you use this last problem as a competition between your peers or peer groups in the course. It is a continuation of problem 3.c. Using a high-level frontend for tensorflow such as tflearn or keras, try to improve your classifier by means of deeper networks and/or convolution layers which are readily available as plug&play building blocks. Depending on interest and time we can use the more challenging CIFAR10 dataset for this last part as opposed to the simpler to predict MNIST dataset. In the latter case, it would also make sense to use GPUs.

Finally, using a simple 2-D Convolutional Neural Network (CNN) model from keras, we can further increase the predictive accuracy in 3(c) to `99.15 percent` (see Appendix). This required reshaping our data to fit into the CNN, changing the input layer to a 2-D CNN and adding a MaxPooling layer to down-sample the input to enable the model to make assumptions about the features to reduce over-fitting. After creating the convolutional layers, I flatten them to ensure they can act as an input to the Dense layers. The full implementation can be seen in the Appendix.

References

Katariya, Y. (n.d.). *Applying convolutional neural network on the mnist dataset.* Retrieved from `https://yashk2810.github.io/yashk2810.github.io/Applying-Convolutional-Neural-Network-on-the-MNIST-dataset/`, accessed on 2018-12-30.

Lee, M.-C., & To, C. (2010). Comparison of support vector machine and back propagation neural network in evaluating the enterprise financial distress. *arXiv preprint arXiv:1007.5133*.

Robert, C. (2014). *Machine learning, a probabilistic perspective.* Taylor & Francis.

Sopyła, K. (n.d.). *Mnist digit classification with scikit-learn and support vector machine (svm) algorithm.* Retrieved from `https://github.com/ksopyla/svm_mnist_digit_classification`, accessed on 2018-12-31.

Trevor, H., Robert, T., & JH, F. (2009). *The elements of statistical learning: data mining, inference, and prediction.* New York, NY: Springer.

Weisberg, J. (n.d.). *Building a neural network from scratch: Part 1.* Retrieved from `https://jonathanweisberg.org/post/`, accessed on 2018-12-10.