COMP7703 Machine Learning
Assignment 2

Thanat Chokwijitkul 44522328

Question 4.2

In matlab, implement the mean shift clustering algorithm as discussed in lectures and papers. Hand-in your code for this question. To do this, use a "flat" kernel function (you will need to specify the value for the radius parameter, λ). You can choose to implement as either a "blurring" or "non-blurring" process.

```
function centroids = meanShift(X,lambda)
    centroids = [];
    numObservations = size(X,1);
    numClusters = 0;
    initialIndices = (1:numObservations)';
    epsilon = 1e-3*lambda;
    flag = false(numObservations,1);
    numInitialObservations = numObservations;
    votes = zeros(numObservations,1);
    clusterMembers = [];

    while numInitialObservations
        index = initialIndices(ceil((numInitialObservations-1e-6)*rand));
        mu = X(index,:);
        members = [];
        thisVotes = zeros(numObservations,1);
        while true
            sumSquaredDistances = sum(bsxfun(@minus,mu,X).^2,2);
            indices = find(sumSquaredDistances <= lambda^2);
            thisVotes(indices) = thisVotes(indices)+1;
            shift = @(x) mean(x,1);
            mu_0 = mu;
            mu = shift(X(indices,:));
            members = [members; indices];
            flag(members) = true;

            if norm(mu-mu_0) < epsilon
               mergeability = 0;
               for i=1:numClusters
                   distance = norm(mu-centroids(i,:));
                   if distance < lambda/2
                      mergeability = i;
                      break;
                   end
               end

               if mergeability > 0
                    currentMember = numel(members);
                    oldMember = numel(clusterMembers{mergeability});
                    weights = [currentMember;oldMember]/(currentMember+oldMember);
                    clusterMembers{mergeability} =
```

```
42                          unique([clusterMembers{mergeability};members]);
43                      centroids(mergeability,:) = mu*weights(1)+mu_0*weights(2);
44                      votes(:,mergeability) = votes(:,mergeability) + thisVotes;
45                  else
46                      numClusters = numClusters+1;
47                      centroids(numClusters,:) = mu;
48                      clusterMembers{numClusters} = members;
49                      votes(:,numClusters) = thisVotes;
50                  end
51                  break;
52              end
53          end
54          initialIndices = find(~flag);
55          numInitialObservations = length(initialIndices);
56      end
```

The MATLAB code above implements the mean shift clustering algorithm discussed in the lecture and [1] as a *non-blurring* process using a *flat* kernel function. According to [2] and [1], this algorithm is simply an iterative process which shifts each data point to the mean of its neighbourhood based on a window (specified by the parameter λ). The procedure of the above algorithm can be summarised as follows: 1) Create a window around each data point specified by λ; 2) Calculate the average of data points within that window; 3) Move or *shift* the window to the mean and repeat the process until convergence.

The algorithm was implemented as a *non-blurring* process. As opposed to a *blurring* process, the original data points are not simultaneously updated after each iteration. Instead, it keeps track and records the members of each cluster using data indices in which a new mean value can be computed using a subset of the original data points (which are not updated at any point).

Based on the definition of a *flat kernel* defined in [1], let x be a data point and K be a flat kernel representing a characteristic function of a window λ, $K(x)$ is defined as

$$K(x) = \begin{cases} 1 \text{ if } ||x|| \leq \lambda \\ 0 \text{ if } ||x|| > \lambda \end{cases}$$

This characteristic function was implemented in line 19. However, as the algorithm computes the sum of squared distance between each data point and the current mean value, λ^2 must be used instead in this case.

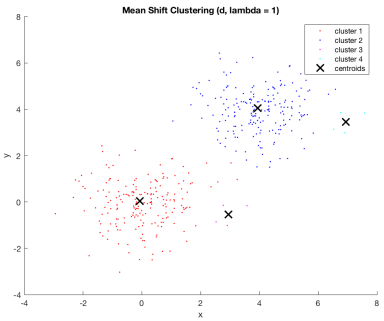
The output of this mean shift clustering algorithm is an n-by-2 matrix containing n resulting centroids which can be used for visualisation later. Each row is a centroid and the two columns represent its x and y axis respectively.

Question 4.3

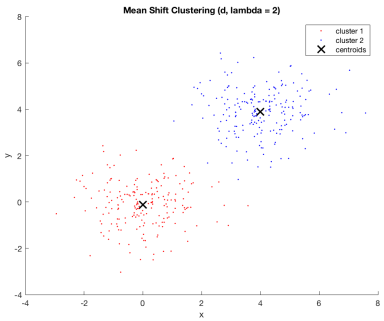
For each of the datasets 'd' and 'e' above, run your algorithm with three different, suitable values of λ.

Recall the testing data, a, b and c are three groups of properly clustered randomly generated data points. d is a combination of a and b, and e is a combination of a, b and c. Figure 1(a), 1(b) and 1(c) visualise the resulting clusters when applying the mean shift clustering algorithm to the dataset d with the lambda (or bandwidth) values of 1, 2 and 5 respectively. The lambda values were obtained via the *trial and error* method. The following list outlines the experimental results on the dataset d (the number of tuples in the result set is the number of clusters):

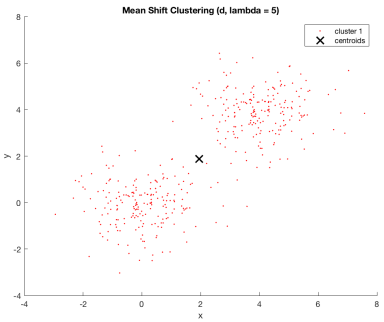
- meanShift(d, 1) $\Rightarrow \{(-0.0645, 0.0455), (3.9355, 4.0455), (2.9455, -0.5460), (6.9455, 3.4540)\}$
- meanShift(d, 2) $\Rightarrow \{(-0.0010, -0.1141), (3.9990, 3.8859)\}$
- meanShift(d, 5) $\Rightarrow \{(1.9559, 1.8753)\}$



(a) Mean shift clustering on d with $\lambda = 1$

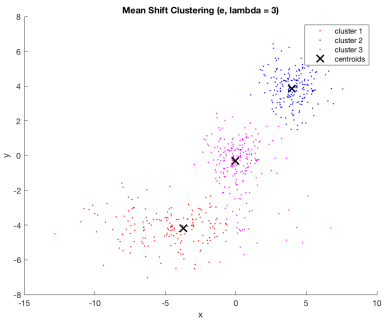


(b) Mean shift clustering on d with $\lambda = 2$

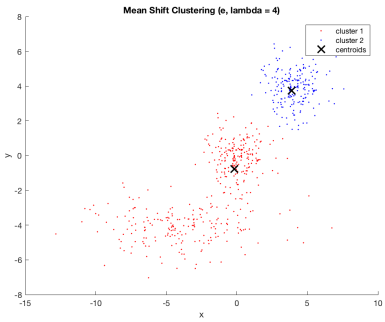


(c) Mean shift clustering on d with $\lambda = 5$

(d) Mean shift clustering on e with $\lambda = 2$



(e) Mean shift clustering on e with $\lambda = 3$



(f) Mean shift clustering on e with $\lambda = 4$

Figure 1: Applying mean shift clustering on the dataset d and e with different values of λ

Figure 1(d), 1(e) and 1(f) visualise the resulting clusters when applying the mean shift clustering algorithm to the dataset e with the lambda (or bandwidth) values of 2, 3 and 5 respectively. The following list outlines the experimental results on the dataset e (the number of tuples in the result set is the number of clusters):

- meanShift(e, 2) $\Rightarrow \{(-0.0170, -0.1559), (-3.8899, -4.2076), (3.9762, 3.9119), (4.4654, -3.8554)\}$
- meanShift(e, 3) $\Rightarrow \{(-3.7296, -4.1701), (3.9674, 3.8657), (-0.0352, -0.3108)\}$
- meanShift(e, 4) $\Rightarrow \{(-0.1837, -0.7678), (3.8759, 3.7292)\}$

According to the experimental results, it can be concluded that the value of the radius parameter λ is critical and needs to be carefully selected. A smaller λ may slow down the convergence and tends to result in the algorithm producing too many clusters while a larger λ may accelerate the convergence but tends to merge distinct clusters which also leads to unexpected clustering results. Most of the time, a λ value in between the two extremes (from *trial and error*) usually results in nicer clusterings.

Question 5.1

List the code you have written that implements PCA.

```
function data = p5q1(X,dim)
    sigma = cov(X);
    [eigenvectors,eigenvalues] = eig(sigma);
    [eigenvalues,index] = sort(diag(eigenvalues),'descend');
    eigenvectors = eigenvectors(:,index(1:dim));
    eigenvalues = eigenvalues(1:dim);
    data = X*eigenvectors;
end
```

The MATLAB code above implements Principal Component Analysis (PCA). The function takes a dataset as a matrix X and the targeted number of dimensions *dim* as inputs and returns the reduced dataset stored in the variable *data*. The procedure is outlined as follows:

1. Compute a covariance matrix of X using the *cov* function.
2. Find the corresponding eigenvectors and eigenvalues of the resulting covariance matrix using the *eig* function.
3. Sort the eigenvalues in descending order along with obtaining the indices.
4. Obtain the first *dim* columns of the eigenvector matrix and the first *dim* eigenvalues.
5. Multiply the original dataset X by the *dim* eigenvectors with the largest associated eigenvalues.

The eigenvectors computed using the covariance matrix are the principle components, which is a reduced number of uncorrelated variables from the entire dataset. The result of the above PCA algorithm is a dimensionality-reduced dataset with *dim* dimensions.

Question 5.2

Run your PCA function on the MNIST data.

(a) Produce a plot of the data in the space spanned by the first two principal components. Colour each point by its class.

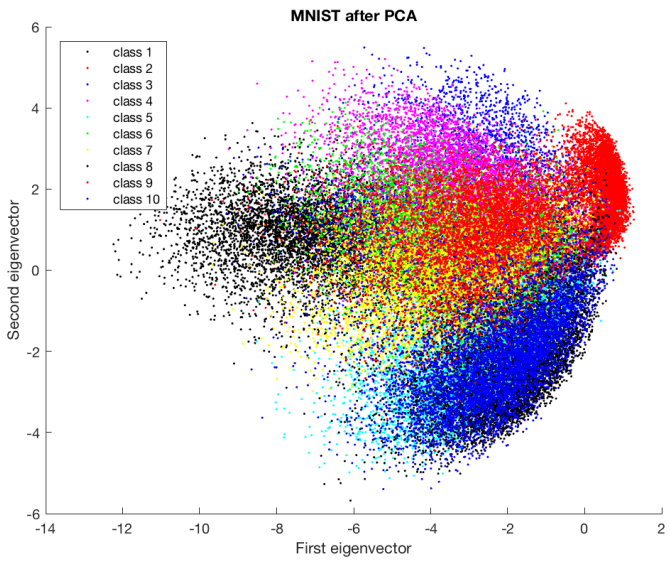


Figure 2: Application of PCA to reduce the dimensionality of the MNIST dataset

Figure 2 visualises a 2-dimensional representation of the MNIST handwritten digit dataset produced by PCA. As shown in the figure, data from most of the classes are intermixed and hardly separable. This observation implies that the two most important dimensions for most of the classes are overlapped. These classes may be effortlessly separable in such higher dimensions as the original dimensions (784 dimensions) but become less separable when the number of dimensions is reduced.

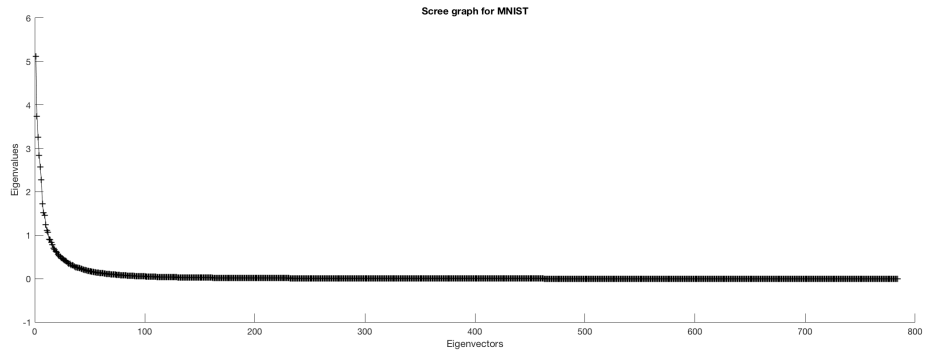
(b) What percentage of the data variance is accounted for by the first two principal components?

Chapter 6.3 of [3] mentions that only some leading eigenvalues should be taken into account while some eigenvalues that have less contribution to variance and should be neglected. Given n principal components and let $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ be a set of eigenvalues, the proportion of variance explained by the leading k principal components with more contribution to variance can be computed by

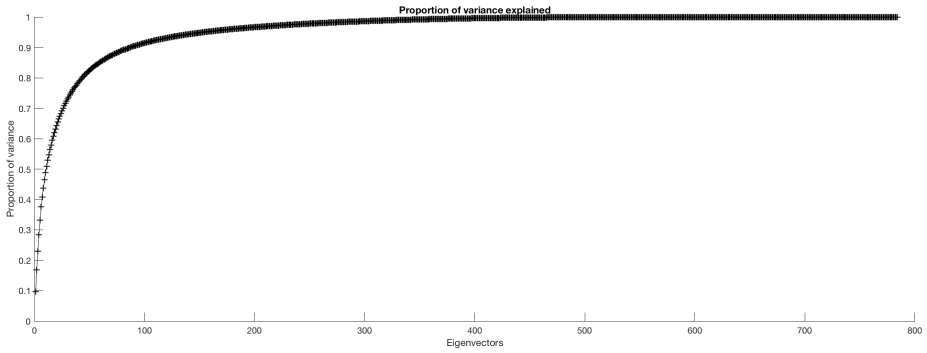
$$\frac{\lambda_1 + \lambda_2 + ... + \lambda_k}{\lambda_1 + \lambda_2 + ... + \lambda_k + ... + \lambda_n}$$

Using the above formula, the result yields that the first two principal components explain **16.8 percent** of the variance.

(c) From the results, produce a Scree graph similar to that shown in Fig 6.4 of the Alpaydin text.



(a) Scree graph for the MNIST dataset



(b) Proportion of variance explained given for the MNIST dataset

Figure 3: Scree graph and proportion of variance explained given for the MNIST dataset

Figure 3(a) is a plot of eigenvalues of the eigenvectors computed using the covariance matrix of the MNIST handwritten digit dataset. Figure 3(b) shows the proportion of variance explained using the formula introduced in Chapter 6.3 of [3]. By conducting a *visual analysis*, these scree graphs can assist in decision-making for an appropriate number of principal components. For data visualisation, one can visually analyse Figure 3(b) in order to decide whether the first two or three principal components explain a sufficient percentage of variance so that the reduced data can be plotted and used for other tasks such as structure or outlier searching. In

the case of the MNIST dataset, the first two principal components explain only 16.8 percent of the variance. This factor may be the reason why the plotted reduced MNIST data using PCA shows less separability, which results in multiple overlapped clusters.

Question 5.6

Provide a screenshot of the 2-dimensional visualisation after 300 iterations. Plot the error at each iteration up to 300 iterations in steps of 10.

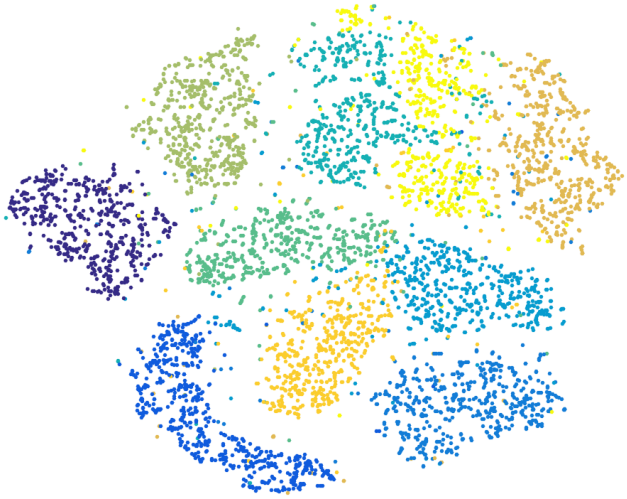


Figure 4: A 2-dimensional visualisation of the MNIST dataset produced by t-SNE

Figure 4 is the visualised result of running the t-Distributed Stochastic Neighbour Embedding (t-SNE) algorithm on 6,000 data points from the MNIST dataset for 300 iterations. Each original data point is in $\mathbb{R}^{784}$. After being projected into a 2-dimensional plane, it is visually apparent that t-SNE is capable of dividing the data into 10 clusters where each cluster corresponds to the different handwritten digit. Nevertheless, the margins of some clusters are hardly visible and some clusters are overlapped or badly separated e.g. the yellow and cyan clusters.

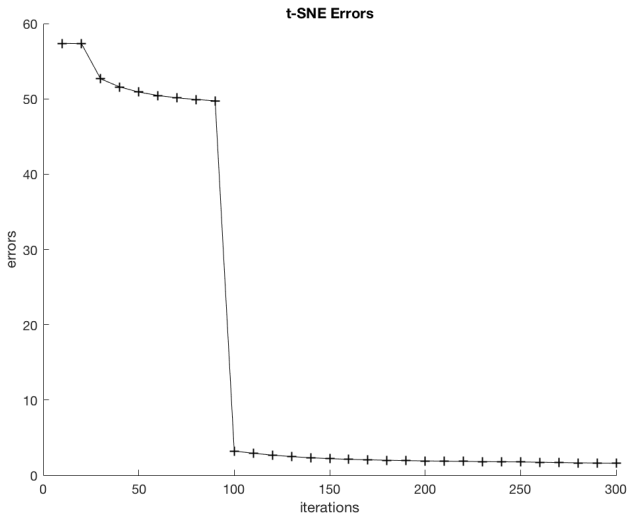


Figure 5: The cost (error) at each iteration up to 300 iterations in steps of 10

Figure 5 is the plot of the error at very 10 iterations up to 300 iterations. As the cost function of t-SNE is non-convex and multiple runs give different results, these error values correspond to the results produced in Figure 4 only. By visually analysing the results, the error value usually gradually decreases over time. However, there also appears to be a rapid change between the 90^{th} and 100^{th} iteration. The error values between these intervals dramatically decreased from 49.7508 to 3.2238. This is perhaps an effect of using the *early exaggeration* method [4] as one of the optimisation methods to control the space between natural clusters in the embedded space. The algorithm uses the early exaggeration method with an exaggeration of 4 for the first 100 iterations. Multiplying the similarity matrix P by such exaggeration constraint results in the natural clusters in the data tending to form tight widely spread clusters in the map. According to the 2-dimensional visualisation of the MNIST dataset produced by t-SNE, this effect creates a lot of empty space, which helps the clusters move around in order to find a good global organisation.

Note: The implementation of the early exaggeration method can found in lines 51-53 and lines 87-89 in the file tsne_p.m downloaded from https://lvdmaaten.github.io/tsne/.

Question 5.8

Comment out lines 51-53 and 87-89. Run t-SNE for 300 iterations for perplexity values ranging from 10 to 300 in steps of 10. Produce a 3D plot with perplexity and iterations on the horizontal axis and cost on the vertical axis. Think of and explain a simple but appropriate heuristic for choosing the perplexity. Provide the visualisation in 2D space after 300 iterations for your chosen perplexity and compare this with your result in (Question 5.6).

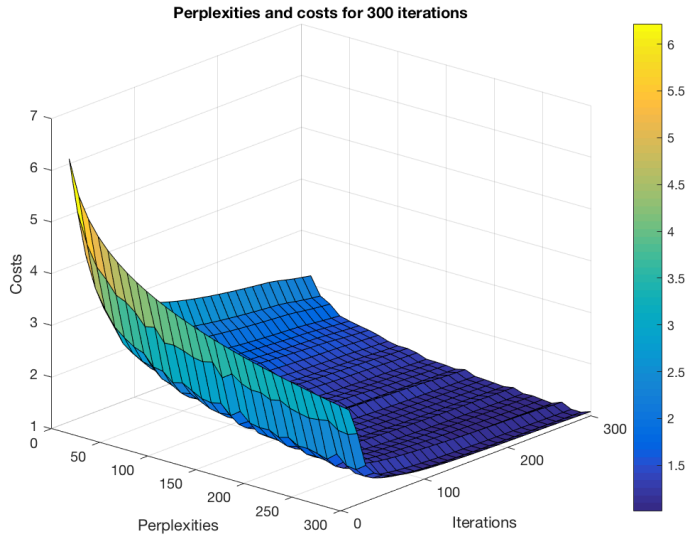


Figure 6: A surface plot of different perplexity values and corresponding costs

Commenting out lines 51-53 and lines 87-89 in the file tsne_p.m means discarding the early exaggeration technique, which results in a smoother representation of error values (without an immediate change after a certain number of iterations which is 100 in this case).

The original paper [4] states that the performance of t-SNE is fairly robust under different values perplexity. The perplexity parameter is related to a guess about the number of nearest neighbours each data point has. Although t-SNE is considered quite insensitive to this parameter, different values of perplexity can have a complex impact on the resulting pictures.

Figure 6 is a 3D plot of different perplexity values and corresponding costs when running t-SNE on the MNIST dataset for 300 iterations. One can observe that the initial error value tends to decrease as the perplexity value increases. Additionally, as the number of iterations increases, it is

quite obvious on the surface plot that the perplexity values in the range [10,50] show more changes in terms of the error values compared with other larger perplexity values.

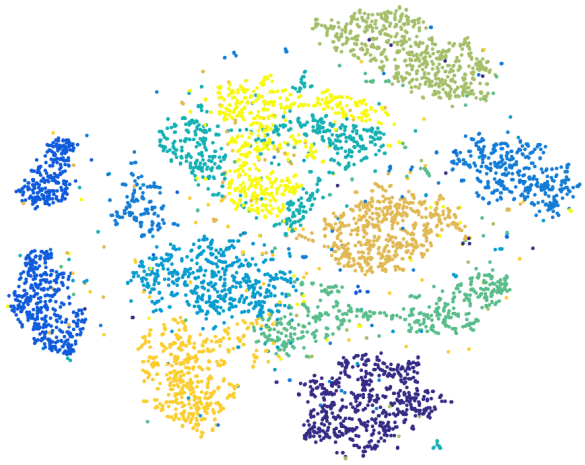


Figure 7: A 2-dimensional visualisation of the MNIST dataset produced by t-SNE using the perplexity value of 50

The paper [4] also suggests that a perplexity value in the range [5,50] tends to result in the output plot showing visually separated clusters. For the MNIST dataset, data points seem to be more properly clustered when the perplexity value is in the range [30,50]. Nevertheless, the most suitable perplexity value is 50 since it shows noticeably better clusters for the dataset when compared to other clustering results produced by t-SNE with other perplexity values. A 2-dimensional visualisation of the MNIST dataset produced by t-SNE using the perplexity value of 50 is shown in Figure 7. The simple heuristic used to find this value is the *trial and error* method based on the theory and recommendation in the original paper [4]. As the selected perplexity value is different from the default value (30) used in Question 5.6, both resulting plots are different in terms of the behaviour of t-SNE when performing clustering. However, the plot produced for this question is considered less well-separated compared to the one produced by Question 5.6 since the algorithm is not capable of properly group the expected clusters. One obvious reason is because the early exaggeration technique, which creates empty spaces and helps the clusters move around to find a good global organisation, has been removed.

Question 6.3

Train an MLP on the MNIST data that we have used previously. You will need to make several "design choices" to do this. Make sure you split your dataset into a training, validation and test set. Your answer for this question should describe the design choices you made to create and train your network and a report of your results. You can also include error graphs or any other output you feel is useful.

Data Preprocessing

The original MNIST handwritten digit dataset has a training set of 60,000 examples and a test set of 10,000 examples. For this question, the training set is divided into two smaller sets including 50,000 examples for training and another 10,000 for validation. The number of the test data remains unchanged.

Each input data point is represented as a 1-by-784 vector. Each dimension (pixel value) is subsequently normalised to be in the range [0,1] (dividing by 255, which is the maximum grey scale value). Each output data point is a scalar in the range [1,10] representing 10 classes. In order

to apply *probabilistic classification* in neural networks, it is a good practice to encode the training output data as *one-hot* vectors. For example, if the output class is 5, it will be encoded as [0,0,0,0,1,0,0,0,0,0].

Multilayer Perceptron Model

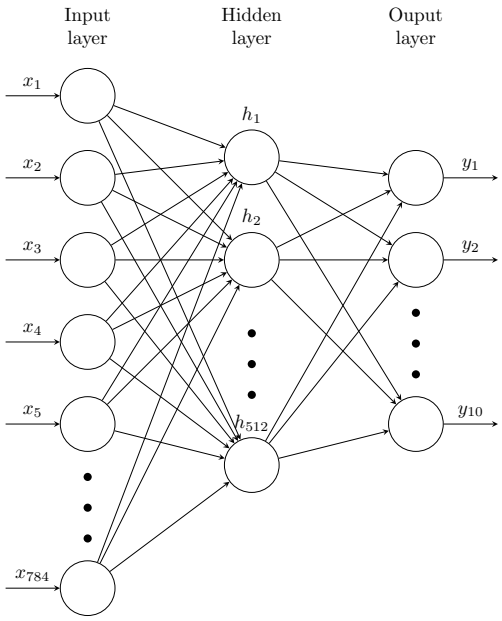


Figure 8: An MLP model for the MNIST handwritten digit classification problem

A multilayer perceptron (MLP) with a single hidden layer as depicted in Figure 8 has been designed as a learning model for this classification problem. The model architecture can be defined as follows:

- **Input layer** - The input layer contains 784 neurons based on the number of dimension of the input data.
- **Hidden layer** - The hidden layer contains 512 hidden neurons with rectified linear units (ReLUs).
- **Output layer** - The output layer contains 10 neurons with the softmax activation function.

Although the appropriate size of a hidden layer can be empirically-derived by trial and error, [5] suggests that the optimal number usually be between the sum of the size of input and output layers. This number needs to be carefully selected as too many neurons may result in the model trying to memorise the data (which causes overfitting) instead of learning it. Many rules of thumb have been tried out, such as the mean of the total number of input and output neurons or even the geometric pyramid rule proposed by [6]. However, after many trials, any number close to $\frac{2}{3}$ of the sum of the number of input neuron and the number of output neurons (532) tends to yield better classification results. For the systematic purpose and neatness, the nearest binary value, namely 512, was selected to be the size of the hidden layer.

Based on the defined neuron network architecture, the activation function associated with the hidden layer is the rectifier or ReLU [7], which is simply defined as

$$\sigma(x) = max(0, x)$$

According to [8], ReLU has range [0, ∞] while the sigmoid function has range [0,1]. The gradient of the sigmoid function tends to vanish as the value of x increases which does not occur in the case of ReLU. The rectifier does not alter the gradient value during the backpropagation process, which alleviates the vanishing gradient problem. The major advantages of ReLU as discussed in [9]

and [10] are 1) convergence acceleration; 2) fast computation; 3) easy optimisation; and 4) better generalisation.

The output layer of the model uses the softmax activation function [11], which is defined as

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}}$$

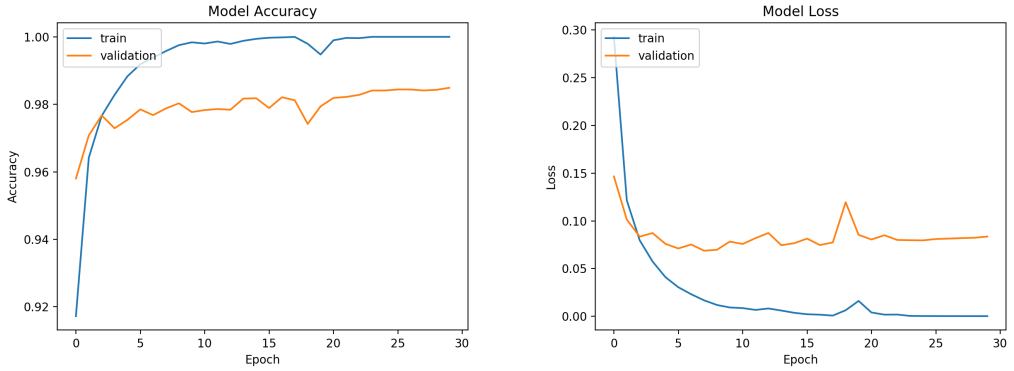
The softmax function is designed to transform any real-valued vector into a probability vector, which is suitable for this multiclass classification problem as its output can be used to represent the probability distribution over K possible outcomes.

The model uses *cross-entropy* as the loss function as it is a better choice when dealing with this probabilistic classification problem because it primarily focuses on the model's confidence in the correct class.

Instead of using the standard gradient descent or stochastic gradient descent, the optimisation technique used in the model is *Adaptive Moment Estimation (Adam)* [12] with the recommended learning rate of 0.001 and the convergence threshold (ϵ) of $1e-08$. Adam, as introduced in [12], is a stochastic gradient-based optimisation technique with adaptive learning rates and momentum on each parameter, designed to efficiently deal with machine learning problems with high-dimensional parameter space or large datasets such as the MNIST dataset.

Ultimately, there are some other hyperparameters that need to be specified, including the number of *epochs* and *batch size*. The number of epochs is the number of times the learning algorithm iterates through the entire training data and updates the weights. The batch size is the number of examples in a single forward/backward pass. After multiple experimental trials, the model yields the best results when training for 30 epochs with the batch size of 128 because a higher number of epochs tends to result in an increase of the cross-entropy error.

Experimental Results



(a) Model accuracy

(b) Model loss

Figure 9: Model accuracy and model loss

Figure 9(a) and 9(b) illustrate the training performance of the model in terms of accuracy and cross-entropy error. After 30 epochs, the model achieved accuracies of 0.9993 and 0.9849, and cross-entropy errors of 0.0001 and 0.0835 for the training set and validation set respectively.

The trained model achieved an accuracy of 0.9844 with 0.0156 classification error, and a cross-entropy error of 0.0722 when performing classification on the test set. In terms of micro-average recall, precision and F-measure, the trained model achieved recall of 0.9841, precision of 0.9848 and F-measure of 0.9844, which demonstrates a respectable generalisation capability of such simple classification model.

Model Summary

Multilayer Perceptron
Input layer: 784 neurons
Hidden layer: 512 neurons with rectified linear units (ReLUs)
Output layer: 10 neurons with the softmax function

Algorithm
Training algorithm: Adaptive Moment Estimation (Adam)
Cost function: Cross-Entropy

Hyperparameters
Number of epochs: 30
Batch size: 128
Learning rate: 0.001
Convergence threshold: $1e-08$

References

[1] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[2] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.

[3] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[4] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[5] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[6] T. Masters, *Practical neural network recipes in C++*. Morgan Kaufmann, 1993.

[7] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, no. 6789, pp. 947–951, 2000.

[8] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010, pp. 807–814.

[9] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean *et al.*, "On rectified linear units for speech processing," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2013, pp. 3517–3521.

[10] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 30, no. 1, 2013.

[11] C. M. Bishop, "Pattern recognition," *Machine Learning*, vol. 128, pp. 1–58, 2006.

[12] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] S. Marsland, *Machine learning: an algorithmic perspective*. CRC press, 2015.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.