PCA and Competitive learning

***Abstract*—This lab report describes the process of performing Principal Component Analysis (PCA) to reduce the dimensionality of datasets. Using the MNIST dataset as an example PCA is applied to the 784 dimensional data to reduce it down to 3 dimensions. The best features to represent the data are chosen by inspection using 3-Dimensional plots. Following this, the report describes how competitive learning can be used to cluster the numbers in the MNIST dataset and how the unsupervised training method can be optimised. The experiments found that using the first, third and fourth components separated the MNIST the best in the reduced feature space. Adding noise to the learning was shown to improve the performance of the competitive learning algorithm implemented.**

***Keywords*—Machine Learning, Neural Networks, MNIST, Principal Component Analysis, Competetive Learning**

I. Principal Component Analysis

A. Introduction

Principal Component Analysis is a dimensionality reduction technique which finds a new set of lower dimension feature vectors. This is achieved by transferring some of the information about the data into the basis vectors called the components.

The components maximise how separated the samples are in the lower dimension space. However separated data samples does not imply that the differently labelled samples are separated, this means care needs to taken when choosing the components.

B. Algorithm implemented and the dataset

The MNIST dataset consists of 5000 handwritten numbers encoded in 28x28 pixel images. Using the pixel values as features the images are unravelled and are represented as 784 element vectors. These vectors are transposed to become rows of a dataset matrix X.

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,784} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ x_{5000,1} & x_{5000,2} & \cdots & x_{5000,784} \end{pmatrix}$$

Where $X_{i,j}$ is the value of sample i's feature j.

The aim of the Principal Component Analysis implementation is the reduce the 5000x784 matrix X into a 5000x3 matrix $X_{reduced}$.

$$X_{reduced} = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ \vdots & \vdots & \vdots \\ x_{5000,1} & x_{5000,2} & x_{5000,3} \end{pmatrix}$$

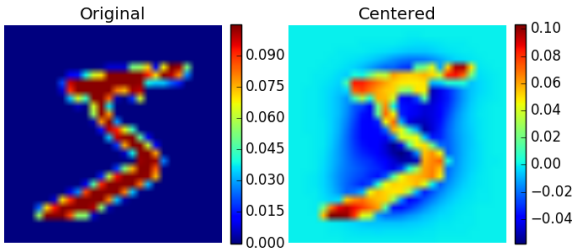

Fig. 1. On the left an instance of a number 5 from the MNIST dataset is being displayed. The right shows an image of the centered version of the same instance of a 5.

The first step when performing PCA is to center the data this means to translate the dataset to be centered around the origin. This is achieved by subtracting the feature value of each sample by the mean value of that feature in the dataset.

Figure 1 shows two images, the image on the left shows an instance of a number 5 in the MNIST dataset. The image on the right shows the same instance but with the data centered around the mean. The centered number shows the points of interest that make a five more unique compared to the average digit. These points of interest are indicated by the red highlighted parts in the corners of the five.

Using the centered data a covariance matrix C is then created. The covariance matrix is a MxM matrix where M is the number of features in the original data. The value of the element $C_{i,j}$ is the covariance of features i and j. The diagonal of the matrix is the variance of those features.

$$C = Cov(X^T) = \begin{pmatrix} var(X_1^T) & \cdots & cov(X_1^T, X_{784}^T) \\ \vdots & \ddots & \vdots \\ cov(X_{784}^T, X_1^T) & \cdots & var(X_{784}^T) \end{pmatrix}$$

The next step in the algorithm is to compute the eigenvectors and eigenvalues of the covariance matrix. The eigenvectors are then sorted by their respective eigenvalues, this is because the aim of PCA is to spread out the data and a larger eigenvalue implies a larger spread when the dimensions have been reduced. The covariance matrix will have 784 eigenvectors,

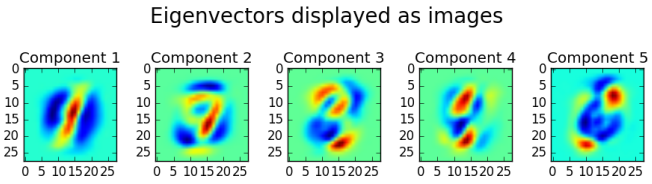

Fig. 2. The first 5 eigenvectors displayed as images. All the images in the MNIST will be approximated using a linear combination of these images. Note these are centered and will be approximating a centered version of the digit.

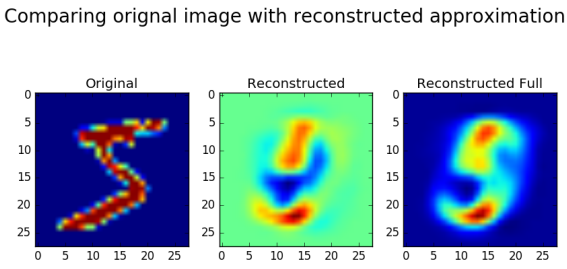

Fig. 3. An example reconstruction of an image in the MNIST dataset using the first 5 eigenvectors. The reconstucted full version contains the means added back onto the digit.

these are stacked next to each other creating a 784x784 square matrix. For the experiments this matrix is reduced to be a 784x5 matrix by selecting the 5 eigenvectors with the highest eigenvalues. Using these vectors the dataset X is projected onto the new basis by multiplying[1] the matrices X and W creating a new dataset with lower dimensionality $X_{projected}$.

$$X_{projected} = XW$$

Figure 2 shows the 5 eigenvectors that were chosen. All the digits in the MNIST dataset will be approximated by using a linear combination of these images when they are being represented in this reduced space. For example a linear combination of a number 5 in the dataset is represented in figure 3.

C. Plotting in 3D

Given the 5D reduced space $X_{projected}$ various 3D subspaces can be selected and plotted[2]. Figures 4, 5, 6, 7 and 8 show these plots with the data points representing an individual sample in the MNIST dataset and the colour indicating the class they belong to. Through inspection, the 1st, 3rd and 4th principal components appear to show the largest separation in the classes, this is also backed up by looking at the mean

[1]Only need to multiply the matrices as the size of each eigenvector is 1, which means they do not need to be rescaled.

[2]Using 5D and then reducing for efficiency. Could have projected the data multiple times using the various eigenvectors. These are equivalent.

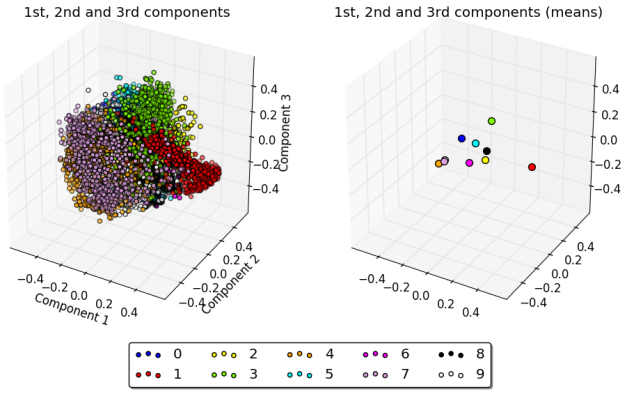

Fig. 4. The MNIST data reduced to 3 dimensions using the first three principal components. By looking at the means, most of the classes have been spread out well. However there is one cluster which contains 3 different labels: 4, 7 and 9 hidden behind the 7. Using these components looks promising but that cluster lets it down.

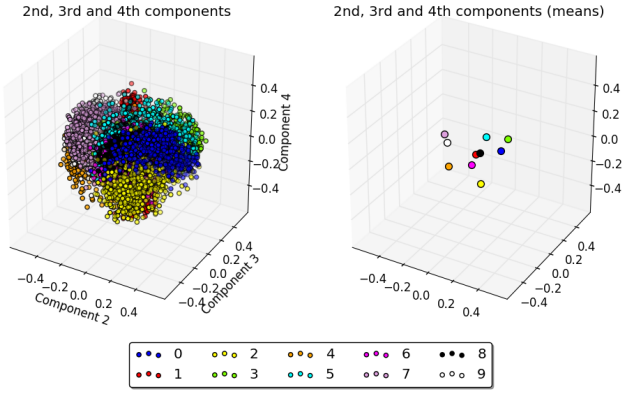

Fig. 5. Plot showing the data projected onto the 2nd, 3rd and 4th principal components. Some labels have been separated well, but there are some labels clustered together.

points. From looking at the mean points all 10 clusters can be seen, however one could argue that the clusters in the centre are too close. PCA guarantees that the first components separate the data the most, here it does not separate the labels the most. Using the first three components resulted in a cluster of three different classes together. Using the other components tended to have points clustered around the origin, making them an unfit choice. Having well separated data is important as this makes it easier to cluster, which will give better classification results.

II. Competitive Learning

A. Introduction

In competitive learning the Neural Network has N output neurons. The aim of the competitive learning algorithm is to have each of the output neurons specialise and fire for a group of inputs. For example a naive network for the MNIST dataset will have 10 output neurons with the aim each of the different

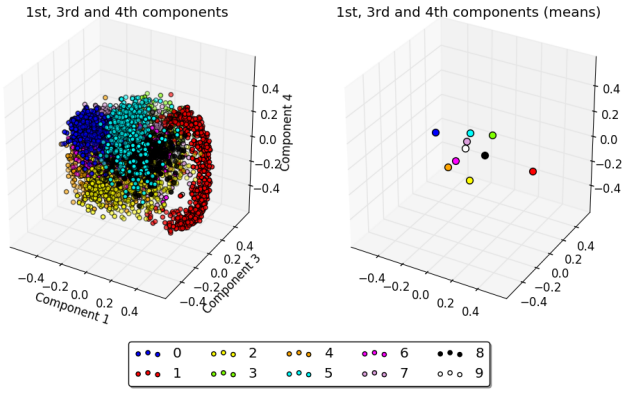

Fig. 6. The MNIST data reduced to 3 dimensions using the 1st, 3rd and 4th component. The points are fairly spread out. Interestingly this set of components really spread of the 2 class. But in general using these components probably yields the best result it terms of spreading out the labels. All the clusters are fairly spread out from looking at the means. Possibly the 7, 9 and 5 classes are too close.

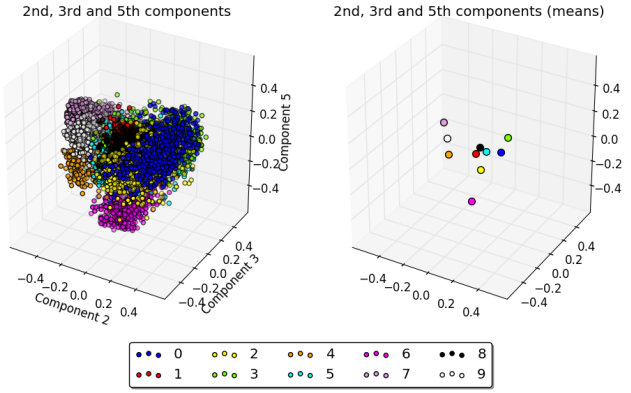

Fig. 7. Data projected onto the 2nd, 3rd and 5th principal components. Some labels here have been separated very well. However the cluster of points in the middle make these components a bad choice.

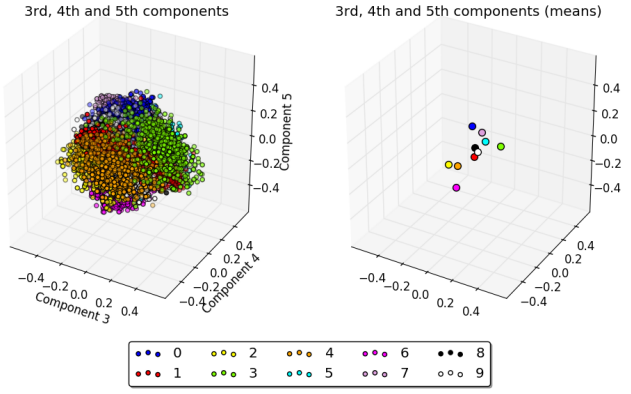

Fig. 8. Plot showing the data projected onto the 3rd, 4th and 5th principal components. The labels are not very well separated using these components making it a bad choice.

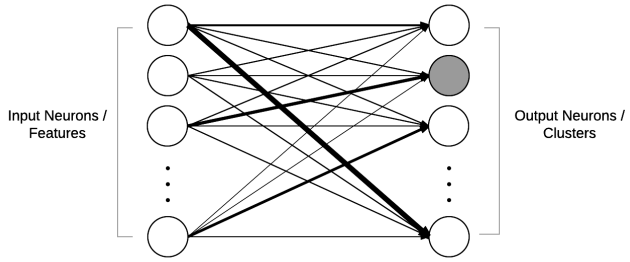

Fig. 9. The simple Neural Network structure for clustering the data. The network has N input neurons, these correspond with the number of features that are being used. There are M output neurons which correspond to the number of different clusters. The network is fully connected, each input neuron connects to each output neuron.

numbers are clustered together and therefore there is an output neuron for the class of 1s, one for the class of 2s and so on. However this is not strictly the case, within classes there are sub classes, for example the number 7 is often written in two different styles. The ideal number of clusters is likely more than 10. Figure 9 shows the simple network created. The neurons on the left represents input data, i.e the features and the neurons on the right represent the output of the network i.e the cluster the input belongs to.

B. Algorithm Implemented

The first network implemented is the naive solution discussed in the previous section. The network has 784 input neurons, these correspond with each of the features in the input data, these inputs are fully connected to the 10 output neurons. The strengths of the connections between the input and output neurons are stored in a 10x784 matrix[3] W. The value $W_{i,j}$ is the weight of the connection between output neuron i and the input neuron j.

To create the matrix W the values of the matrix must be initialised (usually randomly) and then learnt through training examples. For the experiments the following rule is used.

$$\Delta W_k = \alpha(x - W_k)$$

Where α is the learning rate, W_k is the output neuron which fires with the highest number and x is the input pattern being trained on. In competitive learning only the winning neuron's weights are updated i.e the neuron which fired with the highest number. To find the output neuron that fires the most a weighted sum of each of the input neurons going into each output neuron is calculated. This is simply the dot product of the weight vector for that output neuron and the pattern i.e $W_i \cdot x$. Multiplying the weight matrix W by the input pattern gives a vector of all these dot products, therefore to find k for W_k:

$$k = \underset{k}{\operatorname{argmax}} \, (Wx)_k$$

[3]Ordered this way to simplify the maths (multiplies nicely with x)

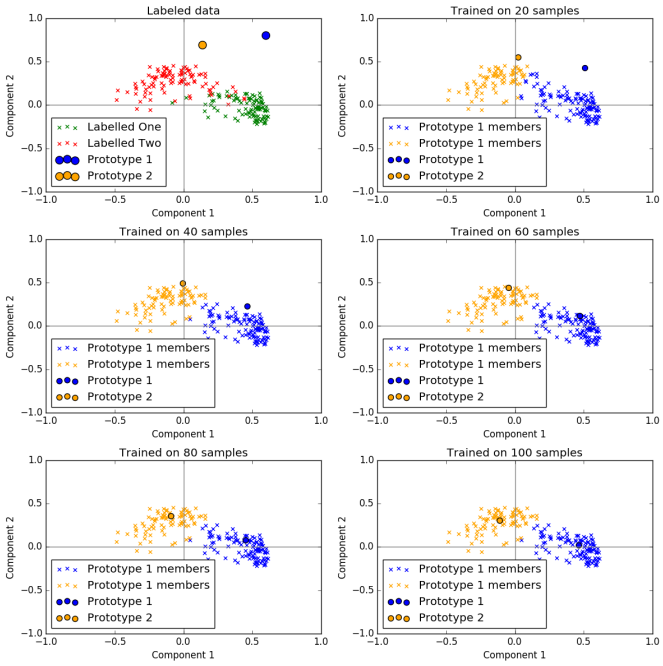

Fig. 10. Figure showing how competitive learning clusters samples. The first image shows the labels of the data and the initial starting points of the two prototypes. The images then progressively show how the weights move towards the data in their clusters. The colours indicate the output neuron which the sample causes to fire the most i.e the cluster they belong to.

The rows in the weight matrix are often called prototypes, this is because they can be geometrically interpreted as a point in the feature-space and will represent a prototype of its members i.e the samples it fires most for.

The learning rule described earlier is an example of an online rule, this means that the weights are adjusted after each input pattern has been seen. Opposed to a batch rule which updates weights after seeing several patterns. Batches are useful when there are performance constraints, they are also less susceptible to noise. However noise can be useful for discovering new opportunities and avoiding local minima.

Figure 10. shows a simplified version of how the algorithm implemented works. Only two different classes are being used and the data has been reduced to a 2D space by using the first 2 principal components to allow for it to be easily displayed. An orange and a blue point have been randomly initialised in the space, these are the prototypes. The network is then trained on samples in the dataset. The samples shown are coloured according to which prototype fires the most for that sample. The images show how the network looks in 20 sample increments, the rule implemented is still an online rule, the intimidate samples are just not shown. This simple demonstration clearly shows the prototypes moving towards the centre of their clusters.

C. Normalisation

Is is important to normalise the weight vectors in competitive learning. Recall that the firing rate of the output neurons is calculated using the dot product i.e. $W_i \cdot x$ which is equal to $|W_i||x|cos\theta$. The size of the pattern vector x is the same for all output neurons and therefore can be ignored here. This means that the size of weight vector W_i is making this neuron more likely to fire. This means that output neurons that have large weight vectors will be more likely to fire even if the angle between them is poor. This will lead to dead neurons as neurons with a small weight vector will struggle to win even though they may have a better cosine angle. The problem worsens too as now, that undeserving neuron is going to have an even larger weight vector the next iteration. Therefore if the weight vector is normalised i.e $|W_j| = 1$ then only the angle between the two points is being used to judge the similarity. Only the angle between the two points is important here because if the vector is scaled up by some scalar all that happens is the intensity of the pixels will change together, which does not create a new number e.g. A faint two is still the same a solid two.

The learning rule described in section B is derived from Oja's rule. Oja's rule has the property that the weight vector's size does not grow infinitely large, instead it converges at a size of 1. Given that the aim of the learning rule is to move the weight vectors towards samples in the dataset and the weight vectors to have a size of 1 it makes sense to have the training data also normalised.

D. Optimisation

After training the networks it can become apparent that some prototypes are never becoming the winners. These are called dead units because they lay dormant in the feature-space and do not move. The number of dead neurons produced will be attempted to be reduced using some of the techniques discussed in [1]. The following configurations of the network are used for experiments.

1) Setting the initial conditions of the weight vectors to be random values drawn from a Gaussian distribution. (Baseline)
2) Setting the initial weights to be random samples from the dataset.
3) Updating the losing output neurons by a lower amount than that of the winner (leaky learning).
4) Adding noise to the output of the neurons. Which means the best neuron does not always win.

The configurations were each run 20 different times, each using a learning rate of 0.05. Each individual experiment consisted of passing through the data 3 complete times. The results are reported in table 1 showing the mean and standard deviation of the number of dead units after running for 20 experiments. The experiments are then repeated again for different amounts of output neurons.

The results show that if the aim is to reduce the number of dead units, then clearly the best way to optimise this is by initialising the weights to be samples in the dataset. However this will produce many prototypes that are capturing the same information. Using a leaky learning rate of 10^{-6} gave a slight improvement over the baseline sometimes but not consistently. Leaky learning did not work well because neurons began

Experiment	10 Clusters	15 Clusters	20 Clusters	25 Clusters	50 Clusters
Baseline	Mean: 0.05 STD: 0.22	Mean: 1.65 STD: 1.06	Mean: 4.25 STD: 1.04	Mean: 7.85 STD: 1.24	Mean: 29.65 STD: 1.93
Noise	Mean: 0.05 STD: 0.22	Mean: 0.40 STD: 0.49	Mean: 1.00 STD: 1.05	Mean: 2.30 STD: 1.23	Mean: 4.55 STD: 1.40
Using samples as initial weights	Mean: 0 STD: 0	Mean: 0 STD: 0	Mean: 0 STD: 0	Mean: 0 STD: 0	Mean: 0 STD: 0
Leaky Learning (rate=10^{-5})	Mean: 2.10 STD: 0.77	Mean: 5.90 STD: 1.26	Mean: 9.90 STD: 0.94	Mean: 14.90 STD: 0.77	Mean: 38.75 STD: 1.13
Leaky Learning (rate=10^{-6})	Mean: 0.15 STD: 0.36	Mean: 2.0 STD: 0.55	Mean: 4.10 STD: 1.55	Mean: 8.55 STD: 1.40	Mean: 29.40 STD: 1.39

TABLE I. TABLE OF RESULTS FOR EXPERIMENTS USING DIFFERENT TRAINING CONFIGURATIONS.

learning a bit of every sample and did not specialise well. Using noise to optimise the network worked well because it allowed the best neurons to win most of the time but also allowed the next best neurons to win some of the time. This stochasticity allows for the opportunity for prototypes that are similar to other prototypes to not get bullied out of being the winner. Once they have won they may then specialise more. Without the noise the unit may not have been able to specialise. The results also indicate that the right amount of clusters to be used for the data is 15. All the experiments got a very low amount of dead units for 10 clusters, this indicates that not enough clusters were chosen to represent all the variety in the data well. Using 20 clusters lead to many dead units in most of the configurations. The following experiments will now use 15 prototypes with adding noise to optimise the number of dead units, as this gave a nice balance between optimising the network and producing unique prototypes.

E. Identifying dead units

To identify the number of dead units that are in a network the numbers of times the unit had fired in the last epoch is used. An epoch consists of training the network on all the samples (5000). If the number of times the neuron fired is zero then the neuron is marked as dead.

F. Weight change over time

Using the new network configuration, 15 output neurons and using noise to optimise. The weight change over time for this network is shown in figure 11. The x-axis and y-axis are on a log scale. Online learning is inherently very noisy, because of this the curve has been smoothed using a window of 1000 samples, to better show the trend. The graph clearly shows that around 10^5 iterations i.e 20 epochs. The network has learnt everything and any further epochs would be pointless as the change is so minimal it will not effect the results, as moving a prototype by a fraction will not change the samples that are closest to it.

G. Prototypes

Figure 12 shows the network using 15 prototypes, from inspecting this image it is clear to see that all 15 of the prototypes are alive. These images represent the average member of their cluster. The weights can be interpreted geometrically like samples can. If so these prototypes will be positioned

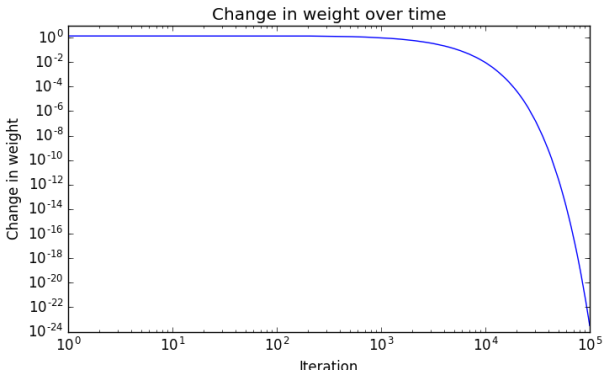

Fig. 11. Plot showing the average change in weight over time of the output neurons. Clearly by 10^5 iterations the network has finished learning, the change in weight after this point is extremely small. Further training after this point will move the prototypes by amounts that are too small to make any difference.

in the middle of the samples that the prototypes fire most for (the members of the cluster). Some of the prototypes are representing different varitions of the same class, for example the 11th and 13th prototype are both representing a 1, however they are clearly representing different variations. The 13th prototype is clustering 1s which have been slanted. Two similar looking 6s are also present in the center of the image. Interestingly from looking at these prototypes there is no clear prototype for clustering the digit 4 and 9. There are a couple prototypes that look like a mixture of the two. The system needs to ideally learn all the digits first before learning variations of the same digits.

H. Correlation matrix of prototype

Through inspecting the prototypes in figure 12 many of the prototypes look similar. Ideally the prototypes should look different otherwise they are redundant as they are capturing the same cluster of data. The similarity between the prototypes can be accessed through calculating the correlation between the samples. To calculate the correlation the training data is passed through the network again, if a prototype fires past a threshold θ the prototype is marked as active for this digit. Similar digits will fire together and therefore give a high correlation. Figure 13 shows the corresponding correlation matrix for the prototypes given in figure 12. To calculate this

matrix, prototypes which fired with a value greater than 0.5 were marked as active with a 1 and prototypes that did not fire more than 0.5 were marked with a -1. Then using the following formula the correlation between prototypes p_i and p_j was calculated:

$$Corr_{i,j} = \frac{1}{N}\sum_{\mu=0}^{N} p_i^{\mu} p_j^{\mu}$$

Where p_k^{μ} is prototype k active state (-1 or 1) for sample μ in the dataset. This formula gives a final NxN matrix where N is the number of prototypes in the network. The highly correlated neurons were then marked by highlighting pairings which have a correlation higher than 0.7. These neurons are very similar and are therefore clustering the same sort of data which means the network could be further optimised.

The matrix on the right in figure 13 backs up the claim made by inspecting the prototypes in figure 12. The two 6s look visually similar and also are highly correlated. The ones that appeared to be clustering different types of 1 were in-fact highly correlated too. Further work needs to be done to reduce this.

I. Combining the PCA and Competitive Learning

The performance of competitive learning might be improved using PCA. In the previous experiments the 784 pixels were used as features. These features are highly correlated with their neighbours. The current neural network does not distinguish the difference between a pixel that is neighbouring and one that is far away. This means the network can be frown off by noise in the data. For example if an unseen image is being clustered and it has random black pixels sparsely positioned around the edges of the image. The current system will be frown off by this because the noise will be equally weighted with all the useful pixels located in the middle. In a reduced eigenspace these surrounding pixels will almost be ignored because no component will weight these pixels highly.

PCA creates more abstract features that describe the data, such as curves. These are more robust to variations in the data. Choosing the right amount of features to optimise this is important, too few components will lead to not enough abstract features to describe the full array of possible digits well enough. Too many components will lead to the same issue before with using pixels. Using PCA will also make the learning process more efficient because the weight matrix will be reduced resulting is less calculations that need to be performed.

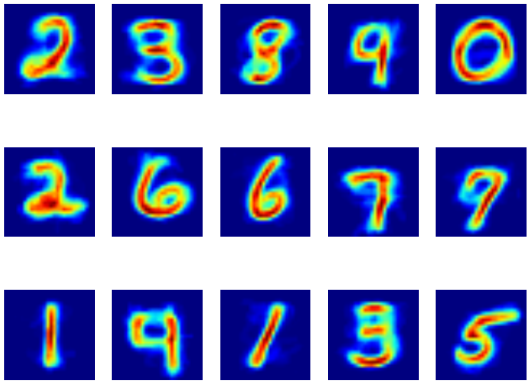

Fig. 12. Figure showing the 15 prototypes (weights going into output neurons) of the network. These images represent an average member of the cluster i.e it is the center of the cluster.

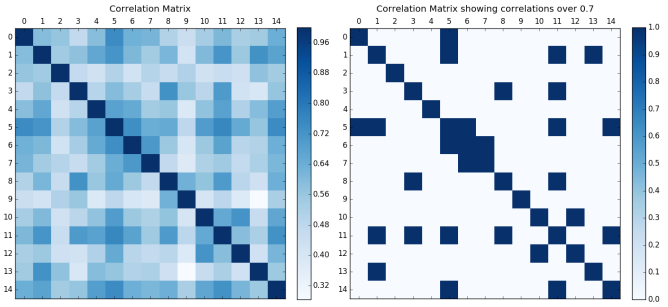

Fig. 13. Correlation matrix for the prototypes shown in figure 12. Two neurons are considered to fire together if they both fire about a threshold of 0.5. The image on the left shows the correlation values pairing all the prototypes. The image on the right highlight the pairings that have a high correlation ($corr > 0.7$).

APPENDIX A
REPRODUCING RESULTS

To show the 3D plots shown in section 1 run `python3 plot_pca.py`.

To run the experiments listed in Table 1. run `python3 optimisation_experiments.py`. This will run all the experiments 20 times and output the mean and standard deviation value for each experiment.

To view the correlation matrix, prototypes and weight change curve run `python3 competitive_learning.py`. This file also has parameters that can be tweaked to produce variants of the figures shown in the report. e.g show without noise being added.

To reproduce the figure demonstrating how competitive learning works run `python3 plot_learning.py`.

APPENDIX B
CODE

The core of the code is located in two classes, PCA and Neural Network. The other files created use these classes to do tasks such as plotting. The code snippets below have been slightly altered to be more understandable out of context of the class they reside.

PCA - Fit

```
    # Calc mean shape=(features,)
    mean = np.mean(dataset, axis=0)
    # Centre the data around origin
    # shape=(samples, features)
    centered = dataset - mean

    # Calc covariance matrix
    # shape=(features, features)
    cov = np.cov(centered, rowvar=False)

    # Calc eigenvectors and values
    eig_values, eig_vectors
            = np.linalg.eigh(cov)

    # Sort eig vectors by eig values
    indices = np.argsort(eig_values)[::-1]
    new_basis = eig_vectors[:, indices]
```

Fig. 14. Calculates the new basis vectors vectors which will be used to project data into the new space. The dataset is first centered, then the covariance matrix is calculated. From this covariance matrix the eigvectors and eigenvalues are calculated, the eigenvectors are then sorted by their eigenvalues.

PCA - Transform

```
if isinstance(components, int):
    # If int choose the first n
    new_basis = new_basis[:, :components]
elif isinstance(components, list):
    # If a list (of ints), use those components
    new_basis = new_basis[:, components]
else:
    msg = 'Components must be a list or an int'
    raise ValueError(msg)

projected = np.dot(dataset - mean, new_basis)
```

Fig. 15. Selects the basis vectors to use from an int or a list. If a list of ints past in e.g [0,1,3] then use 1st, 2nd and 4th eigenvectors. If an int is past in then just you the first n eigenvectors. The dataset is centered and then projected onto the basis vectors that were calculated previously. The projection is just matrix multiplication because the size of the eigenvectors are 1.

Competitive Learning - Initialisation

```
if init == 'random':
    W = np.random.rand(output_neurons, no_features)
elif init == 'samples':
    # initialise matrix using
    # random samples from the dataset
    indices = np.arange(dataset.shape[0])
    # choose n random indices
    selected_idx = np.random.choice(indices,output_neurons)
    W = dataset[selected_idx]
```

Fig. 16. Weights can be either initialised randomly or through selecting samples from the dataset. If initialised randomly then the weights are randomly chosen real values between 0-1.

Competitive Learning - Normalisation

```
W = W / np.linalg.norm(W, axis=1).reshape((-1, 1))
```

Fig. 17. The weight vectors are normalised such that the size of the vectors are 1. This is simply achieved by dividing the weight vectors by their size i.e the l_2-norms.

Competitive Learning - Learning

```
# Get the similarities for this pattern to the prototypes
similarities = np.dot(W, pattern)
# Normalise similarities
similarities = similarities / number_output_neurons

# Add noise to the similarities if specified
if use_noise:
    # Generate noise
    noise = np.random.rand(number_output_neurons) / 200
    # Apply noise
    similarities = similarities + noise

# Find best prototype for the sample
max_idx = np.argmax(similarities)

# Calc the change using the online rule
change = learning_rate * (pattern - W[max_idx])

# Apply the change to winner
W[max_idx] = W[max_idx] + change
```

Fig. 18. The code above is the online training implemented. Given a sample this is how the weights are updated. First the firing rates of each of the output neurons is calculated by multiplying the weight matrix W with the pattern. This is an analogous to similarities in k-means. The similarities are then normalised using the number of clusters. If noise is being used it is then added onto the similarities to add some variation into the learning. After this the most similar neuron is found. Using this information the learning rule is applied to calculate the change in weight. This change will then just be applied to the winning neuron.

Competitive Learning - Leaky Learning

```
# Move prototypes that lose,
# by a smaller amount than if they win
# Needs to be smaller to
# work correctly (ideally by a lot)
assert loser_factor < learning_rate
# create mask for the winner
winner_mask = np.zeros_like(similarities, dtype=bool)
winner_mask[max_idx] = True
# Opposite to winner mask is the loser mask
losers_mask = ~winner_mask

# Select all the prototypes that are not the winner
losers = W[losers_mask]
# Calc change for all losers
changes = loser_factor * (pattern - losers)
# Add the change
W[losers_mask] = losers + changes
```

Fig. 19. Leaky learning requires a much smaller learning rate for losing neurons than that is used on the winner. To update the losing neurons a mask is first created, which selects the winning neuron. The negation of this therefore selects all the losing neurons. The learning rule is then applied to all the losing neurons using the smaller learning rate.

Running average

```
    abs_change = np.abs(change).sum()

    # Make running average
    if samples_seen == 1:
        # Have the weight change start at the first one
        weight_change_average[samples_seen-1] = abs_change
    else:
        window_size = 1000
        prev = weight_change_average[samples_seen-2]
        weight_change_average[samples_seen-1] =
                    prev*((window_size-1)/window_size
                    + abs_change/window_size)
```

Fig. 20. Code for computing the running average. The absolute value is used because the amount changed is what is useful not the direction. This is then sumed to get the total change in weights of the input neurons going into this output neuron.

Correlation Matrix

```
# Holder for corr matrix
corr = np.zeros((clusters, clusters))
threshold = 0.5

for sample in dataset:
    # Get neuron firings
    output = np.dot(weights, sample)

    output_states = np.zeros_like(output)
    output_states[output <= threshold] = -1
    output_states[output > threshold] = 1

    corr += np.outer(output_states, output_states)

corr /= dataset.shape[0]
```

Fig. 21. A correlation matrix is first initialised with all zeroes. The dataset is then looped through one by one. The output of each neuron is calculated, if a neuron is above the threshold 0.5 the neuron is set to 1, if it less or equal it is set to -1. The outer product with the same vector is then used to give a MxM matrix, where M is the number of output neurons. The outer product gives a matrix of all the pairs of output neurons multiplied together. The sum of this over all the samples divided by the number of samples gives the correlation matrix.

REFERENCES

[1] John A Hertz, Anders Krogh, and Richard G Palmer. *Introduction to the theory of neural computation*. 1st ed. Perseus Books, 1999.