The University of Manchester

Machine Learning Lab Report

Lab report for the module of
EEEN4/60151: Machine Learning and Optimization
in the Department of Electronics and Electrical Engineering

YEAR OF SUBMISSION
2021

Jason Dominguez
ID 10154707

THE UNIVERSITY OF MANCHESTER

Contents

1 Single Perceptron **1**

1.1 Aim and understanding 1

1.2 Implementation, Results and Analysis 1

2 Single Layer Perceptrons **3**

2.1 Aim and understanding 3

2.2 Implementation, Results and Analysis 4

3 Multilayer Perceptron **6**

3.1 Aim and understanding 6

3.2 Implementation, Results and Analysis 7

4 Conclusion **8**

5 LeNet 5 for handwritten digit classification (on MNIST dataset) **9**

5.1 Aim and understanding 9

5.2 Implementation, Results and Analysis 10

6 Self-Organizing Map (SOM) **10**

6.1 Aim and understanding 10

6.2 Implementation, Results and Analysis 11

Appendices **13**

A Code **13**

A.1 ImageData class 13

A.2 Model class 18

A.3 Task 1: Single Perceptron 29

A.4 Task 2: Single Layer Perceptrons 31

A.5 Task 3: Multilayer Perceptron 32

A.6 Task 4: LeNet 5 for handwritten digit classification (on MNIST dataset) 33

A.7 Task 5: Self-Organising Map (SOM) 35

1 Single Perceptron

1.1 Aim and understanding

The aim of this experiment was to train a single perceptron model for the binary classification of 'handwritten' digits. These digits were simulated 5×5 images, with black represented by a value of zero and white represented by a one for each pixel.

The input of the single perceptron model is a 25 value feature vector corresponding to a single example image. 25 comes from the 5×5 images being flattened into a vector. The perceptron computes

$$y_{\text{output}} = \text{sign}(\mathbf{x}^{\text{T}}\mathbf{W} + b) \tag{1}$$

[1], where **x** is a single example vector and T denotes the transpose. **W** and b are the weights, of size 25×1, and bias, a scalar, of the perceptron, respectively. The sign function returns a 1, for arguments > 0, or -1, for arguments < 0. A y_{output} of 1 corresponds to a prediction of a one and -1 corresponds to a zero.

The process of training is to update the weights and biases with the aim of improving the accuracy of predicted image labels. First, the perceptron starts with randomly initialized **W** and b. From this, a predicted label for a single training image is calculated, y_{output}. Using this, the weights and bias of the network are updated,

$$\mathbf{W} := \mathbf{W} + \alpha\mathbf{x}(y_{true} - y_{output}), \tag{2}$$

$$b := b + \alpha(y_{true} - y_{output}) \tag{3}$$

[1], where y_{true} is the correct label for the image and α is the learning rate. This optimization method is known as stochastic gradient descent. α must be positive and can be chosen to get the best performance. These new weights and biases are used get a prediction for another image. This is done for each image in the training set, known as a single epoch. This is repeated either for a pre-determined number of epochs, or until a threshold accuracy or error has been passed. Training is performed using training images, whereas some images are left unseen by the model during training. This set of images is called a validation set (or sometimes a test set) and, once trained, the model is used to get predictions for the labels of these images to see if the model can generalize to identify unseen ones and zeros.

1.2 Implementation, Results and Analysis

This task was implemented using the code shown in A.3, making use of the ImageData class and Model class (see A.1 and A.2). These classes were used for sections 1, 2 and 3 of this lab report.

Firstly, the images needed to be preprocessed to be in a format compatible with the single perceprtron model. The images of ones and zeros were represented as $6 \times 5 \times 5$ NumPy arrays as there were 6 examples each of ones and of zeros and the images were 5×5. The corresponding labels, 1 for ones and -1 for zeros, were defined as 6×1 NumPy arrays.

For preprocessing these image and label arrays, an instance of the ImageData class was used. This was responsible for: arranging the ones and zeros images into training and validation sets, with two-thirds of ones and zeros being added to the training set and the last third being added to the validation set; flattening each image into a 1×25; and finally, reshaping the arrays of images to be of size 25×8 and 25×4 for training and validation, respectively.

After preprocessing, the model was trained. This was done using an instance of the Model class. For this task, the model was defined with an input size of 25, the size of the flattened images. The output size and output activation were set to 1 and "sign", respectively.

The model was then trained as described in Section 1.1, using the train method of the Model class object. The train method was also given the validation images and labels, so that, at the end of each epoch, the accuracy of predictions on both the training set and validation set could be calculated and stored for analyzing the results. By setting the 'max_acceptable_error' to be 0, training would stop when training accuracy reached 100%.

One experiment conducted with this single perceptron model was training the model with randomly assigned training and validation image sets fifteen times. The average number of epochs, iterations through entire set of training images, before training accuracy reached 100% and average final validation accuracy are shown in Table 1.

Table 1: Average epochs for training accuracy to reach 100% and average final validation accuracy for fifteen different trained perceptrons. Errors were calculated from standard error of repeated results.

Learning rate	Average epochs until 100% training accuracy	Average validation accuracy (%)
0.1	1.5 ± 0.2	92 ± 4
0.01	2.9 ± 0.2	85 ± 5

Another experiment was to hand engineer the training and validation sets, to see how this affected training and the validation accuracy. The results of this are shown in figure 1.

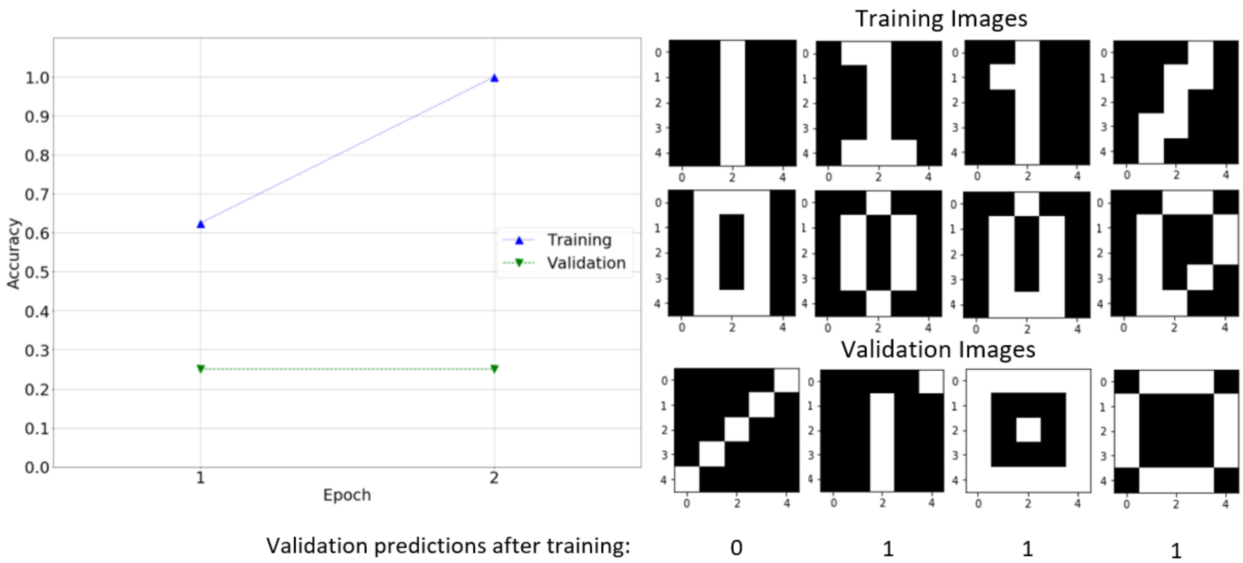

Figure 1: Graph showing the training and validation accuracy of the single perceptron model for each epoch. Next to the graph are the images, grouped into the training and validation set.

From table 1, it can be seen that a smaller learning rate resulted in more training epochs being needed, on average, for the training accuracy to reach 100%. This was expected as a larger learning rate means a larger 'step size' in gradient descent. Also seen in table 1 is that the average validation accuracy upon training completion was less than 100%. When training accuracy is higher than validation accuracy, a model is said to be overfitting the training data. Although it performs well on the training data, it does not generalize as well to unseen data. The results shown in figure 1 help explain this.

Figure 1 shows an example of a particular training and validation image split which was tested and the corresponding accuracy against epoch graph. The graph shows that training accuracy reached 100% at epoch 2, however the validation accuracy is 25%. The training/validation split of images is also shown in figure 1. The label predictions on the validation images shown in figure 1 show that a diagonal one was incorrectly categorized as a zero and both zeros as ones. A possible explanation for the mislabelled diagonal one is that the examples of ones in the training set do not show the same kind of diagonal pattern, whereas some zeros in the training set do show this pattern, such as the bottom right zero in the training images in figure 1. The mislabelled zero with the dot in the middle can be similarly explained as, during training, the perceptron model has never seen a zero with a pixel in the centre and only ones have pixels with value 1 in the centre.

Overall, the number of images for training and validation in this task was small, meaning that performance of the single perceptron model on the unseen images was very dependent on the choice of validation set. In most real-life applications, thousands or millions of images are normally used to help reduce overfitting in image classification.

2 Single Layer Perceptrons

2.1 Aim and understanding

The aim of this experiment was to train a single layer perceptron model to classify "handwritten" digits. The "handwritten" digits used were from the MNIST datset, a datset of $70,000$ 25×25 images of hand drawn digits from 0 to 9.

This task is a multiclass image classification problem. Instead of predicting whether an image was a one or a zero, the model now needed to predict if it was one of ten possible outcomes, digits 0 through to 9. As a result, the single perceptron model needed to be extended to have ten outputs. Each of these outputs corresponds to the prediction of a single outcome. In order for the labelling of images to be compatible with the model, the labels needed to be in a one-hot-encoded (or 1-of-c) format. For example, a label of 3 would be encoded as the vector $[0,0,0,1,0,0,0,0,0,0]$. A single layer perceptron model predicts this label when the 4^{th} output perceptron gives the largest value.

The 10×1 output vector of the single layer perceptron model, y_{output}, is calculated for a single example **x** by

$$\boldsymbol{y}_{\text{output}} = \sigma(\mathbf{x}^{\mathsf{T}}\mathbf{W} + \boldsymbol{b}) = \frac{1}{1 - \exp^{-(\mathbf{x}^{\mathsf{T}}\mathbf{W}+\boldsymbol{b})}}, \tag{4}$$

where σ is the sigmoid function. The weights and biases, **W** and $\boldsymbol{b}$, are of dimensions 625×10 and 1×10. **x** is 625×1 with 625 corresponding to the flattened size of the MNIST 25×25 images.

To work out the weight update rule we need to consider the backward propagation through the single layer perceptron model. The weight update rule has the form

$$\mathbf{W} := \mathbf{W} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}}, \tag{5}$$

$$\boldsymbol{b} := \boldsymbol{b} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}}, \tag{6}$$

where $\mathcal{L}$ is the loss function. Two loss functions were implemented in this task: mean-square error (MSE) and cross entropy/log loss. The MSE loss has the form

$$\mathcal{L} = \frac{1}{2}||\boldsymbol{y}_{\text{true}} - \boldsymbol{y}_{\text{output}}||^2, \tag{7}$$

where y_{true} is the true, one-hot-encoded, image label. By using the chain rule of differentiation, along with the fact that the derivative of the sigmoid function, σ, is $\sigma(1-\sigma)$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}_{\text{output}}} \frac{\partial \boldsymbol{y}_{\text{output}}}{\partial \mathbf{W}} = \mathbf{x}((\boldsymbol{y}_{\text{output}} - \boldsymbol{y}_{\text{true}}) * \boldsymbol{y}_{\text{output}} * (1 - \boldsymbol{y}_{\text{output}})) \tag{8}$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}_{\text{output}}} \frac{\partial \boldsymbol{y}_{\text{output}}}{\partial \boldsymbol{b}} = (\boldsymbol{y}_{\text{output}} - \boldsymbol{y}_{\text{true}}) * \boldsymbol{y}_{\text{output}} * (1 - \boldsymbol{y}_{\text{output}}) \tag{9}$$

[1], where $*$ is element-wise multiplication. Log loss has the form

$$\mathcal{L} = -\sum_i y_{\text{true},i} \log(y_{\text{output},i}) + (1 - y_{\text{true},i}) \log(1 - y_{\text{output},i}), \tag{10}$$

where i corresponds to each output perceptron. From this loss we arrive at

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}\left(\frac{\boldsymbol{y}_{\text{output}} - \boldsymbol{y}_{\text{true}}}{\boldsymbol{y}_{\text{output}} * (1 - \boldsymbol{y}_{\text{output}})} * \boldsymbol{y}_{\text{output}} * (1 - \boldsymbol{y}_{\text{output}})\right) = \mathbf{x}(\boldsymbol{y}_{\text{output}} - \boldsymbol{y}_{\text{true}}) \tag{11}$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}_{\text{output}}} \frac{\partial \boldsymbol{y}_{\text{output}}}{\partial \boldsymbol{b}} = \boldsymbol{y}_{\text{output}} - \boldsymbol{y}_{\text{true}}. \tag{12}$$

2.2 Implementation, Results and Analysis

This task was implemented using the code shown in A.4, making use of the ImageData class and Model class (see A.1 and A.2).

Like in Section 1.2, the images first needed to be preprocessed. The MNIST contained $60,000$ 25×25 grayscale images in a training set and $10,000$ images in a validation set. The only preprocessing needed on the images was flattening and reshaping the arrays to be $625 \times 60,000$ and $625 \times 10,000$. The images also needed to be normalised. This meant dividing the values in the array by 255, the maximum possible pixel value of the images, so that the data values were between 0 and 1. Finally, the image labels were one-hot-encoded.

For this experiment, the input value for the model was 625, the output value was 10 and the output activation was ”sigmoid”. The model was trained using the equations in Section 2.1. As 100% accuracy may not have been achieved in training, training was done for a set number of

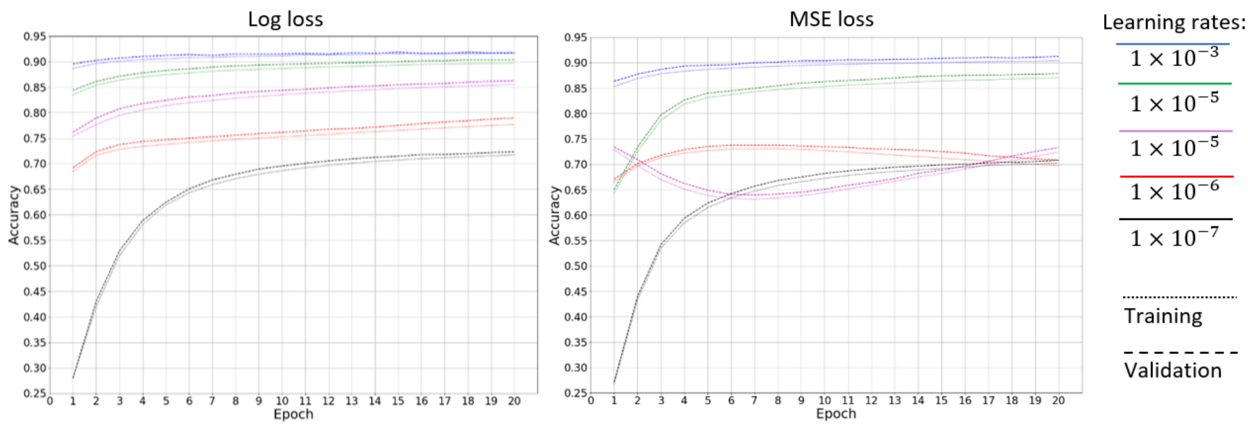

Figure 2: Graphs showing the training and validation accuracy for each training epoch, when using different learning rates. At some points, training and validation accuracy are too close to been seen separately.

epochs. For this task the learning rate was varied from 1×10^{-7} to 1×10^{-3}. The training and validation accuracy results for these learning rates are shown in figure 2, for MSE and log loss.

Alongside this, the effect of too large a learning rate was studied, as well as the effect of using learning rate decay. These results are shown in figure 3.

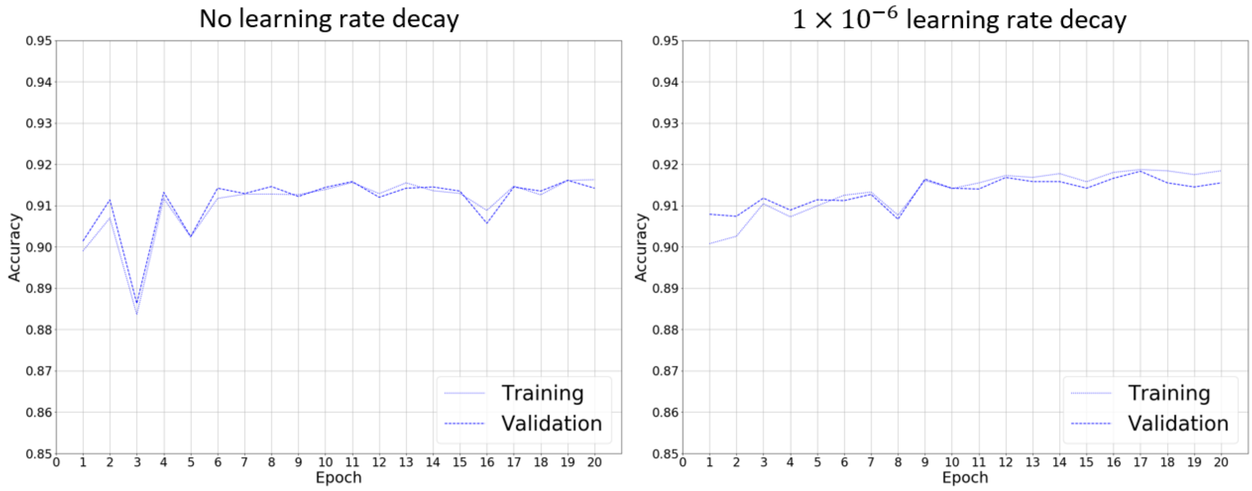

Figure 3: Graphs showing the training and validation accuracy for each training epoch, using an initial learning rate of 0.5. For the graph on the right, a learning rate decay of 1×10^{-6} was used.

Figure 2 shows that larger learning rates lead to both training and validation accuracy beginning to plateau to its highest accuracy at an earlier epoch. Figure 2 also shows that a learning rate that is too small can have adverse effects. This is shown for the MSE loss graph by the decreasing accuracy for 1×10^{-5} and 1×10^{-6} or the plateauing at a much lower accuracy for 1×10^{-7}. The explanation for this is the training of the model becoming stuck in a less optimal local minimum, whereas larger learning rates can effectively skip over these local minima and eventually settle in a more favourable local minimum, with a higher accuracy.

Figure 2 also shows a comparison of the performance of the single layer perceptrons model when using log loss and MSE loss. For all learning rates, log loss showed a slight improvement in highest accuracy achieved and also it did not show the decreasing accuracy effects seen with

MSE loss at small learning rates. For a learning rate of 1×10^{-3}, with MSE loss the training and validation accuracy were 90.4% and 91.2% respectively, whereas for log loss they were 91.6% and 91.8%.

Figure 3 shows how too large a learning rate, such as 0.5, can cause fluctuations in accuracy. This is due to the large learning rate leading to oscillations about a minimum in the loss. A common practice in machine learning is to gradually decrease the learning rate (learning rate decay), to make the most of the early benefits of a large learning rate, but avoid fluctuations when settling at a minimum. Figure 3 shows this smoothing effect using a learning rate decay equation

$$\alpha = \frac{\alpha_{\text{initial}}}{1 + \nu t}, \tag{13}$$

where α_{initial} is the starting learning rate, ν is the decay, set to 1×10^{-6} in this example, and t is the iteration number, which increases after every weight update.

3 Multilayer Perceptron

3.1 Aim and understanding

The aim of this experiment was to train a multilayer perceptron (MLP) model for the multiclass classification the MNIST datset.

The MLP differs from the single layer perceptron model by including a 'hidden' layer of perceptrons before the output layer of perceptron. This layer means there are now two steps in the forward and backward propagation. The output of the hidden layer, $\boldsymbol{y}^{[1]}$ is

$$\boldsymbol{y}^{[1]} = \sigma(\mathbf{x}^{\mathsf{T}}\mathbf{W}^{[1]} + \boldsymbol{b}^{[1]}), \tag{14}$$

where $\mathbf{W}^{[1]}$ and $\boldsymbol{b}^{[1]}$ are the weights and biases for the hidden layer. The dimensions of $\boldsymbol{y}^{[1]}$, $\mathbf{W}^{[1]}$ and $\boldsymbol{b}^{[1]}$ depend on the choice of number of hidden perceptrons, n_{h}. Their dimensions are $1 \times n_{\text{h}}$, for $y^{[1]}$, $625 \times n_{\text{h}}$, for $\mathbf{W}^{[1]}$ and $1 \times n_{\text{h}}$ for $\boldsymbol{b}^{[1]}$, where 625 is the input size of the MLP when using the MNIST images. The output from the hidden layer is then fed-forward into the output layer, where the output of the MLP, $\boldsymbol{y}^{[2]}$ is calculated as

$$\boldsymbol{y}^{[2]} = \sigma(\boldsymbol{y}^{[2]}\mathbf{W}^{[2]} + \boldsymbol{b}^{[2]}), \tag{15}$$

where $\mathbf{W}^{[2]}$ and $\boldsymbol{b}^{[2]}$ are the weights and biases associated with the output layer of perceptrons, of dimensions $n_{\text{h}} \times 10$ and 1×10. The output of the network is a predicted 10 class one-hot-encoded label for the example image.

The weight update rule is derived via backward pass through the MLP. For this task, MSE loss was used. The weight update rule for the output layer is derived using equations 5 and 6 from section 2.2, so that

$$\mathbf{W}^{[2]} := \mathbf{W}^{[2]} + \alpha \boldsymbol{y}^{[1]\mathsf{T}}((\boldsymbol{y}_{\text{true}} - \boldsymbol{y}^{[2]}) * \boldsymbol{y}^{[2]} * (1 - \boldsymbol{y}^{[2]})), \tag{16}$$

$$\boldsymbol{b}^{[2]} := \boldsymbol{b}^{[2]} + \alpha((\boldsymbol{y}_{\text{true}} - \boldsymbol{y}^{[2]}) * \boldsymbol{y}^{[2]} * (1 - \boldsymbol{y}^{[2]})). \tag{17}$$

For updating the weights and biases of the hidden layer, the derivatives $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{[1]}}$ need to be computed, leading to the following hidden layer weight update equations,

$$\mathbf{W}^{[1]} := \mathbf{W}^{[1]} + \alpha \mathbf{x}[((\boldsymbol{y}_{\text{true}} - \boldsymbol{y}^{[2]}) * \boldsymbol{y}^{[2]} * (1 - \boldsymbol{y}^{[2]}))\mathbf{W}^{[2]\mathsf{T}} * \boldsymbol{y}^{[1]} * (1 - \boldsymbol{y}^{[1]})], \tag{18}$$

$$\boldsymbol{b}^{[1]} := \boldsymbol{b}^{[1]} + \alpha[((\boldsymbol{y}_{\text{true}} - \boldsymbol{y}^{[2]}) * \boldsymbol{y}^{[2]} * (1 - \boldsymbol{y}^{[2]}))\mathbf{W}^{[2]\mathsf{T}} * \boldsymbol{y}^{[1]} * (1 - \boldsymbol{y}^{[1]})]. \tag{19}$$

3.2 Implementation, Results and Analysis

Image preprocessing for this experiment was the same as was carried out in section 2.3. For defining the model however, the number of perceptrons in the hidden layer needed to be set. The experiments conducted were: varying the number of perceptrons in the hidden layer, with results shown in figure 4; and using different scale factors for weight initialization, shown in figure 5.

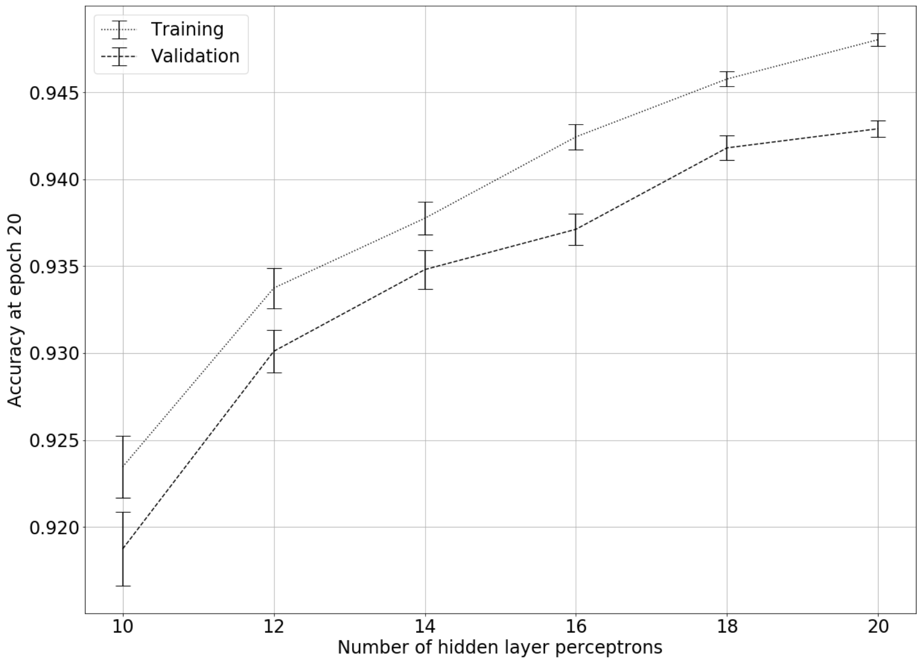

Figure 4: Graph showing the mean training and validation accuracy after twenty epochs for an MLP with a number of hidden perceptrons between 10 and 20. Error bars were calculated from the standard error of six repeated results. learning rate and weight initialization factor of 1×10^{-2} and 1×10^{-3}, respectively, were kept constant.

Figure 4 shows that the accuracy at epoch twenty increased when using more hidden layer perceptrons. This is likely due to the fact that increasing the number of hidden perceptrons increases the complexity of the MLP, meaning that it can model more complex detail. The figure also shows that the training accuracy is higher than the validation accuracy, indicating that overfitting is starting to occur at epoch twenty for all numbers of hidden perceptrons tested, however only slightly.

The most important result from figure 5 is that there was no increase in accuracy for a weight initialization of 10, meaning that weights were initialized with values between -10 and 10. This is evidence of the 'vanishing gradients' problem. With larger weight values, when computing the output of layers in an MLP, the sigmoid function can become saturated. At these points, the gradient of the sigmoid funtion is essentially zero and, as backpropagation is a form of gradient-based learning, no gradient means no learning. Figure 5 justifies why small values are often used to initialize weight values, as for 0.1 or smaller, the performance of the MLP is relatively unaffected, achieving very similar accuracies after twenty epochs. A weight initialization of 1 still shows an increase in accuracy, however is is slower than for smaller weight initializations, again due to the smaller gradient for larger inputs to the sigmoid function.

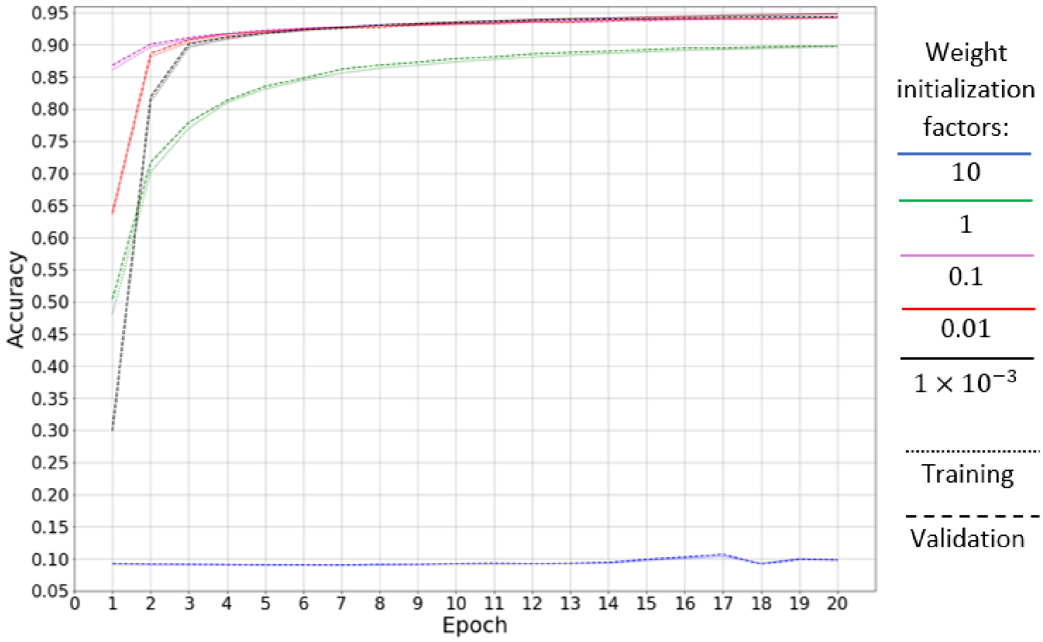

Figure 5: Graph showing the training and validation accuracy as a function of training epoch for different weight initialization factors. The learning rate and number of hidden layer perceptrons were kept constant at 1×10^{-2} and 20, respectively.

4 Conclusion

In conclusion, three different machine learning approaches to the classification of digits were implemented. For the binary classification of 5×5 images of ones and zeros, a single perceptron model was capable of achieving 100% accuracy on training images and unseen images. However, due to the small dataset size for this task, the performance on the unseen images varied depending on which images were chosen for training.

Two different models were trained for the multiclass image classification of the MNIST handwritten digit dataset. Using a single layer perceptron model, the effect of varying learning rate and loss function on training and validation accuracy were studied. The use of log loss, compared to MSE loss, showed a slight increase in accuracy after twenty epochs and no decreasing accuracy for small learning rates. When experimenting with learning rates, it was shown that a value of learning rate that was too small could show adverse affects such as decreasing accuracy and settling to less optimal local minima. Too large a learning rate however could result in overall higher accuracy achieved in the same number of epochs, but flucuations in accuracy. Learning rate decay was shown as one way of gaining the benefits of a larger learning rate whilst reducing these flucuations.

When using an MLP model on the MNIST images, it was shown that a higher accuracy training and validation accuracy could be achieved by using more perceptrons in the hidden layer. Overall, comparing the performance of the single layer perceptron model and the MLP, the MLP could achieve a higher accuracy, 94% compared to 92% validation accuracy for the same learning rate, weight initialization and loss function, but the MLP started to show overfitting. Finally, the vanishing gradient problem was visualized by varying the weight initialization of the MLP, showing that too large an initialization value could result in saturated sigmoid output, meaning small gradients, leading to slower, or no, learning.

5 LeNet 5 for handwritten digit classification (on MNIST dataset)

5.1 Aim and understanding

The aim of this experiment was to use a convolutional neural network (CNN) in the style of LeNet-5, see figure 6, for the classification of digits in the MNIST dataset.

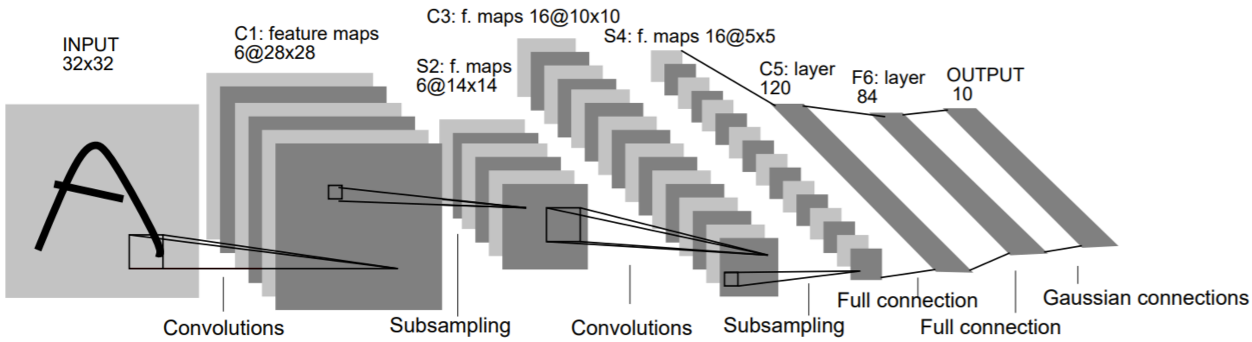

Figure 6: The LeNet-5 CNN architecture, taken from [2].

The first difference between CNNs and MLP is that the input is not a flattened image, but just the image itself. Some of the layers in a CNN consist of filters which perform a convolution to the input, rather than only layers of fully connected perceptrons. Alongside this, convolution layers like these are often followed immediately by pooling layers, where a kernel size is defined. This kernel passes over the input to the layer and outputs either the maximum or average value in that region, depending on whether the pooling is max pooling or average pooling. Figure 7 shows an example of the procedure of convolution and pooling layers. In LeNet 5, after some convolution and pooling layers, the output is flattened and input into a MLP style portion of the network. In the convolution layers of the CNN, the trainable parameters are the values in the filters.

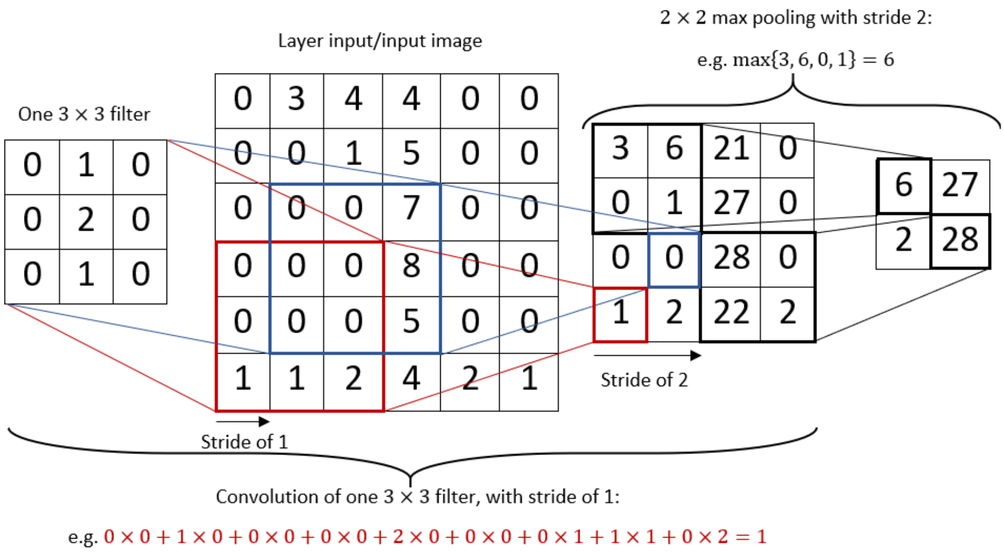

Figure 7: Examples of the mathematical procedure of a convolution and max pooling used in CNNs. Figure taken from [3].

5.2 Implementation, Results and Analysis

To implement this experiment, I used the Keras deep learning framework [4], with the code shown in A.6. To preprocess the MNIST images for the Keras LeNet 5 model, the image were required to be in the format of $m_{\text{examples}} \times 25 \times 25 \times 1$. Figure 8 shows the results of using the LeNet 5 model on the MNIST images, compared to using an MLP. For both, the MSE loss funtion was used, as well as a learning rate of 1×10^{-2} and stochastic gradient descent.

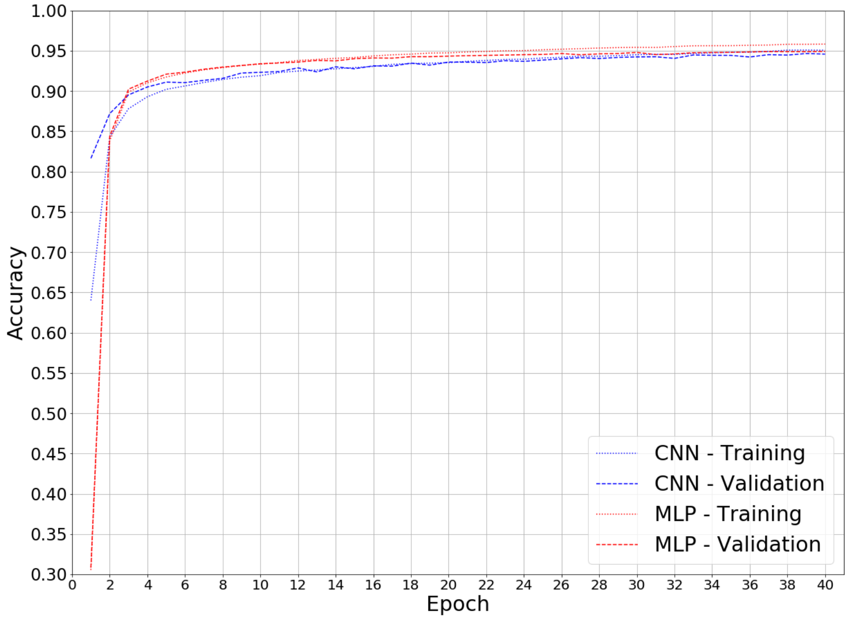

Figure 8: Graph comparing the accuracy on training and validation images for the LeNet-5 CNN and an MLP, for forty epochs.

Figure 8 shows that the performance of an MLP and LeNet-5 was very similar. In the case of the MNIST dataset, the images are very ideal. For example, all the images are centred. As a results, an MLP can achieve a high accuracy on this dataset. However, in more complex image recognition tasks, for example with images where the main object in the image is not centred, CNN can often outperform MLPs.

6 Self-Organizing Map (SOM)

6.1 Aim and understanding

Self-organizing map is a form of dimensional reduction, where objects represented in a high-dimensional space are mapped to a two-dimensional space. The method of learning this mapping starts with a random initialization of weight vectors associated with each point on the SOM, $\boldsymbol{W}_{ij}$. From this, the euclidean distance between a high-dimensional vector, $\boldsymbol{x}$, and each weight vector is computed. The weights are then update via the equation

$$\boldsymbol{W}_{ij} := \boldsymbol{W}_{ij} + \alpha(t)\eta(\boldsymbol{x} - \boldsymbol{W}_{ij}) \tag{20}$$

[1], where t is the iteration number, $\alpha(t)$ is a variable learning rate, varying as

$$\alpha(t) = \frac{100}{200+t} \tag{21}$$

and η is a neighborhood function. The role of η is to ensure that positions closer to the point on the SOM which had the smallest distance to the vector, (u, v), are updated more than others. It has the form

$$\eta = \text{exp} -\frac{(i-u)^2 - (j-v)^2}{2\sigma^2} \tag{22}$$

[1], where σ controls the spread of the neighborhood function, with a larger σ meaning the weights for further away points on the map are affected more. After this weight update, the procedure is repeated for a different high-dimensional vector. Once a set number of iterations has been completed, high-dimensional vectors can be mapped to the two-dimensional SOM using the trained weights.

For this task, the aim was to train an SOM for a series of vectors representing animals, as shown in table 2

Table 2: Table showing the attributes of animals used for creating their vector representations. Taken from [1].

	is			has						likes to			
	small	medium	big	2legs	4legs	hair	hooves	mane	feather	hunt	run	fly	swim
Dove	1	0	0	1	0	0	0	0	1	0	0	1	0
Hen	1	0	0	1	0	0	0	0	1	0	0	0	0
Duck	1	0	0	1	0	0	0	0	1	0	0	1	1
Goose	1	0	0	1	0	0	0	0	1	0	0	1	1
Owl	1	0	0	1	0	0	0	0	1	1	0	1	0
Hawk	1	0	0	1	0	0	0	0	1	1	0	1	0
Eagle	0	1	0	1	0	0	0	0	1	1	0	1	0
Fox	0	1	0	0	1	1	0	0	0	1	0	0	0
Dog	0	1	0	0	1	1	0	0	0	0	1	0	0
Wolf	0	1	0	0	1	1	0	1	0	1	1	0	0
Cat	1	0	0	0	1	1	0	0	0	1	0	0	0
Tiger	0	0	1	0	1	1	0	0	0	1	1	0	0
Lion	0	0	1	0	1	1	0	1	0	1	1	0	0
Horse	0	0	1	0	1	1	1	1	0	0	1	0	0
Zebra	0	0	1	0	1	1	1	1	0	0	1	0	0
Cow	0	0	1	0	1	1	1	0	0	0	0	0	0

6.2 Implementation, Results and Analysis

The SOM was implemented using the code in A.7, following the procedure outlined in the previous section. $sigma$ was varied linearly from an initial value of 3 to 1 at the last iteration in order to help convergence. The SOM was trained for $100,000$ iterations and was plotted every 2000 iterations, shown in figure 9.

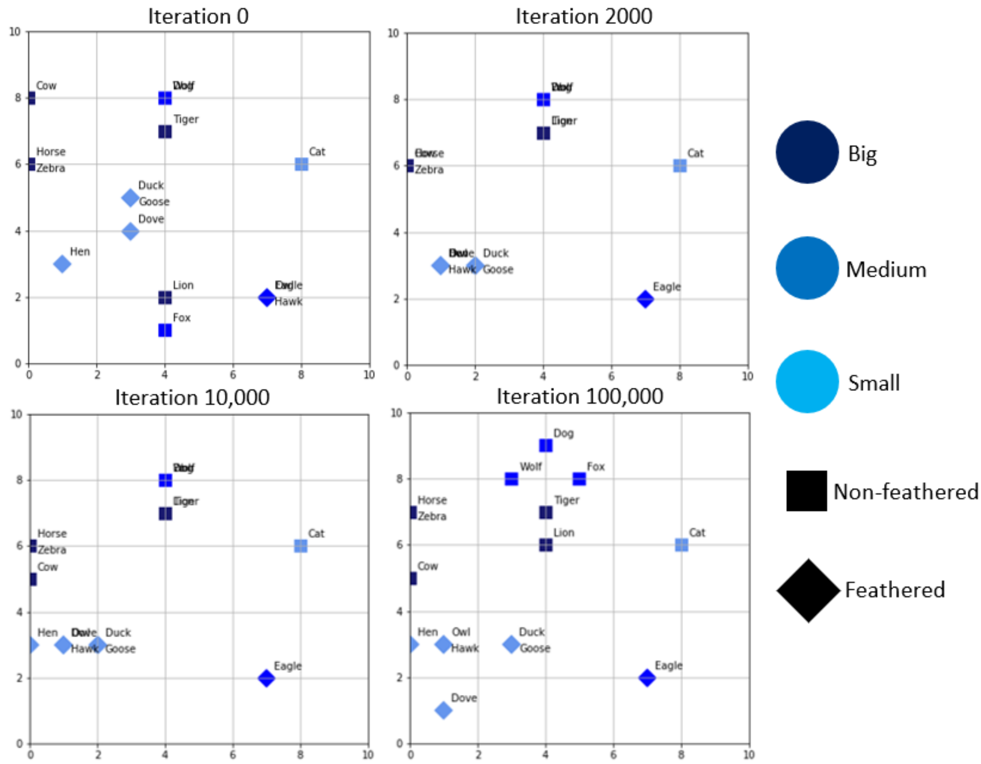

Figure 9: Graphs showing the different stages of learning the SOM mapping, from the first random initialization of weight, until the iteration $100,000$.

Figure 9 shows how similar animals have been grouped together in the SOM, by the last iteration. For example, the bottom half of the SOM contains feathered animals and the top half contains non-feathered. Furthermore, it can be seen that animals such as dog, fox and wolf have been placed very close to eachother, whereas dog and cat are further away. Whilst SOM cannot be directly used for image classification, it does provide a form of dimensional reduction which can be used as the first step in image classification. From this learned mapping, clustering techniques, such as k-nearest-neighbours could be used to then group the animals.

References

[1] H. Yin, "Eeen4/60151 machine learning laboratory: Neural networks for pattern recognition," Blackboard, 2020.

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] J. Dominguez, "Developing a convolutional neural network for the classification of liquid crystal textures and phase transitions," Master's thesis, Department of Physics and Astronomy, The University of Manchester, 2021.

[4] "Keras API," https://keras.io/.

Appendices

A Code

A.1 ImageData class

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 20 17:36:15 2020

@author: Jason
"""

import numpy as np


class ImageData:
    """
    Objects of the ImageData class are used for preprocessing image data and
    their corresponding labels for use in Single Perceptron, Single Layer
    Perceptron, Multilayer Perceptron and LeNet5 task for Machine Learning
    Laboratory (Tasks 1, 2, 3, 4, respectively).
    """

    def __init__(self):
        self.train_data = []
        self.train_labels = []
        self.val_data = []
        self.val_labels = []
        self.unsorted_data1 = []
        self.unsorted_data_labels1 = []
        self.unsorted_data2 = []
        self.unsorted_data_labels2 = []


    def get_data(self, dataset=None, images1=None, image_labels1=None,
                 images2=None, image_labels2=None):
        """
        Method that assigns the input dataset, or unsorted data and labels, to
        their corresponding object attributes.

        Parameters
        ----------
        dataset : Object of some dataset class, optional
            Dataset including sorted training and validation sets of images
            and their corresponding labels (e.g. MNIST dataset, mnist).
            The default is None.
        images1 : np.array, optional
            Array of images belonging to a single label (e.g. all images of
            zeros). The default is None.
        image_labels1 : np.array, optional
            Array of labels corresponding to images1 - all of the same label
            (e.g. a (number_of_examples, 1) array of all zeros). The default
            is None.
        images2 : np.array, optional
```

```python
            Array of images belonging to a single label. The default is None.
        image_labels2 : np.array, optional
            Array of labels corresponding to images1 - all of the same label.
            The default is None.

        Returns
        -------
        None.

        """
        if dataset == None:
            self.unsorted_data1 = images1
            self.unsorted_data_labels1 = image_labels1
            self.unsorted_data2 = images2
            self.unsorted_data_labels2 = image_labels2
        else:
            self.train_data = dataset.train_images()
            self.train_labels = dataset.train_labels()
            self.val_data = dataset.test_images()
            self.val_labels = dataset.test_labels()


    @staticmethod
    def one_hot_encode(labels):
        """
        Static method which one-hot (or 1-to-c) encodes the input labels.
        If an example has the label is 3 and there are 5 possible label values,
        (0,1,2,3,4), the encoded label for that example will be [0,0,0,1,0]

        Parameters
        ----------
        labels : np.array
            Labels which will be one-hot encoded.

        Returns
        -------
        labels_enc : np.array
            The one-hot encoded labels.

        """
        m = labels.shape[0] # Number of example labels to be encoded
        values_to_encode = np.unique(labels)
        size = len(values_to_encode) # Number of values to be encoded

        labels_enc = np.zeros((m, size))
        for i in range(m):
            labels_enc[i, int(labels[i])] = 1

        return labels_enc


    def train_val_split(self, pre_shuffle=False):
        """
        Method that combines the unordered_data1 and unordered_data2 and
        random splits into training and validation sets with a split ratio of
        2/3 training, 1/3 validation. The labels are correspondingly split.

        Parameters
        ----------
```

```python
        pre_shuffle : boolean, optional
            Indicates whether shuffling before splitting data is required.
            The default is False.
        Returns
        -------
        None.

        """
        # If unsorted image data is not in a flattened format it must be
        # flatten before combining
        if len(self.unsorted_data1.shape) > 2:
            # Get height and width of images
            h1 = self.unsorted_data1.shape[1]
            w1 = self.unsorted_data1.shape[2]
            self.unsorted_data1 = self.unsorted_data1.reshape((-1, h1*w1))
            # Now in format (examples, height*width)
        if len(self.unsorted_data2.shape) > 2:
            h2 = self.unsorted_data2.shape[1]
            w2 = self.unsorted_data2.shape[2]
            self.unsorted_data2 = self.unsorted_data2.reshape((-1, h2*w2))

        m1 = self.unsorted_data1.shape[0]
        # Number of examples belonging to dataset 1
        m2 = self.unsorted_data2.shape[0]

        # Combine images and labels for consistent shuffling, so labels stay
        # with their associated image
        data_with_labels1 = np.hstack((self.unsorted_data_labels1,
                                       self.unsorted_data1))
        data_with_labels2 = np.hstack((self.unsorted_data_labels2,
                                       self.unsorted_data2))

        if pre_shuffle:
            # Want the particular images in training and validation sets to be
            # random each time
            np.random.shuffle(data_with_labels1)
            np.random.shuffle(data_with_labels2)

        # 2/3 of total images and their labels will be put into training set,
        # the rest into validation set
        train_ratio = 2/3
        train_split1 = int(train_ratio * m1)
        train_split2 = int(train_ratio * m2)

        train_labels1 = data_with_labels1[:train_split1,
                                          0].reshape(train_split1, 1)
        train_data1 = data_with_labels1[:train_split1, 1:]
        val_labels1 = data_with_labels1[train_split1:,
                                        0].reshape(m1 - train_split1, 1)
        val_data1 = data_with_labels1[train_split1:, 1:]

        train_labels2 = data_with_labels2[:train_split2,
                                          0].reshape(train_split2, 1)
        train_data2 = data_with_labels2[:train_split2, 1:]
        val_labels2 = data_with_labels2[train_split2:,
                                        0].reshape(m2 - train_split2, 1)
        val_data2 = data_with_labels2[train_split2:, 1:]

        self.train_data = np.vstack((train_data1, train_data2))
```

```
        self.train_labels = np.vstack((train_labels1, train_labels2))
        self.val_data = np.vstack((val_data1, val_data2))
        self.val_labels = np.vstack((val_labels1, val_labels2))

        # Return images to their original unflatten format (they'll be
        # flattened again in image_preprocess method if required)
        self.train_data = self.train_data.reshape((self.train_data.shape[0],
                                                   h1, w1))
        self.val_data = self.val_data.reshape((self.val_data.shape[0],
                                               h2, w2))


    def image_preprocess(self, split_data=False, pre_shuffle=False,
                         normalize=True, flatten=False, pad=False,
                         img_first_format=True):
        """
        Method which preprocesses the train_data and val_data attributes of
        the ImageData object. Possible preprocessing includes: splitting
        unordered data into training and validation sets, flattening images,
        padding images and reshaping into (image, examples) array format.

        Parameters
        ----------
        split_data : boolean, optional
            Indicates whether splitting the data into training and validation
            sets is required. The default is False.
        pre_shuffle : boolean, optional
            Indicates whether shuffling before splitting data is required.
            The default is False.
        normalize : boolean, optional
            Indicates whether normalizing the images by 1/255 is required.
            The default is True.
        flatten : boolean, optional
            Indicates whether flattening the images from (height, width) into
            a (height*width) vector is required. The default is False.
        pad : boolean, optional
            Indicates whether padding the images is required.
            The default is False.
        img_first_format : boolean, optional
            Indicates whether image first format (image, examples) is required.
            The alternative is image last format (examples, image).
            The default is True.

        Returns
        -------
        None.

        """
        # If data is not yet sorted, split and sort into training and
        # validation sets
        if split_data:
            self.train_val_split(pre_shuffle)

        # Normalize images
        if normalize:
            self.train_data = self.train_data/255
            self.val_data = self.val_data/255

        # Flatten images (no further flattening or dimension expansion needed
```

```
        # if data was split into training and validation sets)
        if flatten:
            img_height = self.train_data.shape[1]
            img_width = self.train_data.shape[2]
            self.train_data = self.train_data.reshape((-1,
                                                       img_height*img_width))
            self.val_data = self.val_data.reshape((-1, img_height*img_width))
        elif not flatten:
            # If not flattening images, need a 4D-array of size
            # (examples, img_height, imag_width, n_channels) for Keras input
            self.train_data = np.expand_dims(self.train_data, 3)
            self.val_data = np.expand_dims(self.val_data, 3)

        # Pad images with a 2 pixel thick border of zeros
        if pad:
            self.train_data = np.pad(self.train_data,((0,0),(2,2),(2,2),(0,0)))
            self.val_data = np.pad(self.val_data,((0,0),(2,2),(2,2),(0,0)))

        # Get images into the correct format, either (examples, images) or
        # (images, examples)
        if img_first_format:
            self.train_data = self.train_data.T
            self.val_data = self.val_data.T

        print("\nPreprocessed images shapes:")
        print("Training images shape: " + str(self.train_data.shape))
        print("Validation images shape: " + str(self.val_data.shape))


    def label_preprocess(self, one_hot_encode=True):
        """
        Method which preprocesses the train_labels and val_labels attributes
        of the ImageData object. This may include one-hot-encoding the labels
        if required.

        Parameters
        ----------
        one_hot_encode : boolean, optional
            Indicates whether labels should be one-hot-encoded.
            The default is True.

        Returns
        -------
        None.

        """
        # Ensure that labels are in the required (examples, label) format
        m_train = len(self.train_labels)
        m_val = len(self.val_labels)
        self.train_labels = self.train_labels.reshape((m_train, 1))
        self.val_labels = self.val_labels.reshape((m_val, 1))

        # One-hot encode the labels
        if one_hot_encode:
            # Resultant labels array will be of size (examples, number of
            # unique labels)
            self.train_labels = self.one_hot_encode(self.train_labels)
            self.val_labels = self.one_hot_encode(self.val_labels)

```

```python
        print("\nPreprocessed labels shapes:")
        print("Training labels shape: " + str(self.train_labels.shape))
        print("Validation labels shape: " + str(self.val_labels.shape))


    def data_preprocess(self, split_data=False, pre_shuffle=False,
                        normalize=False, flatten=False, pad=False,
                        img_first_format=True, one_hot_encode=False):
        """
        Method that performs image and label preprocessing in sequence to get
        data and labels consistent and ready for any machine learning using
        this data.

        Parameters
        ----------
        split_data : boolean, optional
            Indicates whether splitting the data into training and validation
            sets is required. The default is False.
        pre_shuffle : boolean, optional
            Indicates whether shuffling before splitting data is required.
            The default is False.
        normalize : boolean, optional
            Indicates whether normalizing the images by 1/255 is required.
            The default is True.
        flatten : boolean, optional
            Indicates whether flattening the images from (height, width) into
            a (height*width) vector is required. The default is False.
        pad : boolean, optional
            Indicates whether padding the images is required.
            The default is False.
        img_first_format : boolean, optional
            Indicates whether image first format (image, examples) is required.
            The alternative is image last format (examples, image).
            The default is True.
        one_hot_encode : boolean, optional
            Indicates whether labels should be one-hot-encoded.
            The default is True.

        Returns
        -------
        None.

        """
        self.image_preprocess(split_data, pre_shuffle, normalize,
                              flatten, pad, img_first_format)
        self.label_preprocess(one_hot_encode)
```

A.2 Model class

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 20 18:15:09 2020

@author: Jason
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```python


class Model:
    """
    Objects of the Model class can be used to train a single perceptron,
    single layer of perceptrons or two-level multilayer perceptron machine
    learning model/network.
    """

    def __init__(self, input_size, output_size, hidden_size=None,
                 output_activation="sigmoid"):
        """
        Initialization method called upon creating an object of the class.

        Parameters
        ----------
        input_size : int
            The number of input features for the network.
        output_size : int
            The number of output values from the network. e.g for a binary
            classification this will be 1 (0 or 1), but for n class
            classification it will be n.
        hidden_size : int, optional
            If a MLP network is required, this is the number of perceptron in
            the hidden layer. The default is None.
        output_activation : string, optional
            The activation function for the output perceptrons. For binary
            classification, "sign" can be used. For single layer perceptrons
            or MLP, use "sigmoid". The default is "sigmoid".

        Returns
        -------
        None.

        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.output_activation = output_activation
        self.train_acc_log = []
        self.val_acc_log = []


    def initialize_parameters(self, factor):
        """
        Method to initialize the weights and biases of the model network.
        Initialization is random, from a normal distribution between factor
        and -factor. If the model is initialized with no hidden layer, then
        only one weight matrix and bias vector is intialized. If not, two
        weight matrices and bias vectors are initialized corresponding to
        the hidden and output layers.

        Parameters
        ----------
        factor : float
            If using random intialization this number scales the random
            value to be between -factor and factor.

        Returns
```

```
        -------
        W : np.array
            Weight matrix corresponding to output layer, of size (n_in, n_out).
        b : np.array
            Bias vector corresponding to the output layer, of size (1, n_out).
        Or
        W1 : np.array
            Weight matrix corresponding to hidden layer,
            of size (n_in, n_hidden).
        b1 : np.array
            Bias vector corresponding to the hidden layer,
            of size (1, n_hidden).
        W2 : np.array
            Weight matrix corresponding to output layer,
            of size (n_hidden, n_out).
        b2 : np.array
            Bias vector corresponding to the output layer, of size (1, n_out).

        """
        n_in = self.input_size # Number of input features
        n_hidden = self.hidden_size # Number of hidden layer perceptrons
        n_out = self.output_size # Number of output perceptrons

        if n_hidden == None:
            # Network only has an output layer
            W = np.random.randn(n_in, n_out)*factor
            b = np.random.randn(1, n_out)*factor

            return W, b

        else:
            # Network has a hidden layer and output layer
            W1 = np.random.randn(n_in, n_hidden)*factor
            b1 = np.random.randn(1, n_hidden)*factor
            W2 = np.random.randn(n_hidden, n_out)*factor
            b2 = np.random.randn(1, n_out)*factor

            return W1, b1, W2, b2


    @staticmethod
    def sigmoid(x):
        """
        Method that computes the sigmoid of the input x.

        Parameters
        ----------
        x : np.array
            Input array for which the element-wise sigmoid is required.

        Returns
        -------
        s : np.array
            Array corresponding to the element-wise sigmoid of the elements
            of x.

        """
        s = 1/(1 + np.exp(-x))
        return s
```

```python


    def forward_propagation(self, data, W1, b1, W2=None, b2=None):
        """
        Computes the forward propagation through the model using one or more
        training examples and the weights and biases of the model.

        Parameters
        ----------
        data : np.array
            DESCRIPTION.
        W1 : np.array
            Weight matrix corresponding to first layer.
        b1 : np.array
            Bias vector corresponding to the first layer.
        W2 : np.array, optional
            Weight matrix corresponding to second layer, if MLP.
            The default is None.
        b2 : np.array, optional
            Bias vector corresponding to the second layer, if MLP.
            The default is None.

        Returns
        -------
        y1 : np.array or float
            Output of first layer of model for the image(s) in data.
        y2 : np.array
            Output of second layer of model for the image(s) in data., if MLP.

        """
        m = data.shape[1] # Number of examples in the input data
        hidden = self.hidden_size # Number of hidden perceptrons in the
                                  # network
        outputs = self.output_size # Number of output perceptrons in the
                                   # network

        if hidden == None:
            # Network only has an output layer
            # Apply the chosen activation function to the linear function of
            # the form x_transpose W + b, where x is the input data
            if self.output_activation == "sign":
                y1 = np.sign(np.dot(data.T, W1) + b1).reshape(m, outputs)
                # Reshapes are used to ensure after manipulation that the
                # array shape is correct
            elif self.output_activation == "sigmoid":
                y1 = self.sigmoid(np.dot(data.T, W1) + b1).reshape(m, outputs)

            return y1
        else:
            # Network has a hidden layer and output layer
            y1 = self.sigmoid(np.dot(data.T, W1) + b1).reshape(m, hidden)

            # Outputs of the output perceptrons are computed as the sigmoid of
            # the linear function hidden_activation W2 + b2
            y2 = self.sigmoid(np.dot(y1, W2) + b2).reshape(m, outputs)

            return y1, y2


```

```python
    def update_weights(self, y1, x, d, loss, lrn_rate,
                       W1, b1, W2=None, b2=None, y2=None):
        """
        Updates the the current weights and biases of the model using the
        stochastic or batch gradient descent learning rule. This can be done
        for log loss or MSE loss.

        Parameters
        ----------
        y1 : np.array or float
            y1 : np.array or float
            Output of first layer of model for the image(s) in x.
        x : np.array
            Image or images used for updating the weights.
        d : np.array or int
            True labels of image/images in x.
        loss : string
            The loss function used for defining the weight update rules. Can
            be either "log_loss" or mean-square error loss, "mse.
        lrn_rate : float
            The learning rate for the weight update rule.
            Indicates how large steps are taken.
        W1 : np.array
            Current weights for the first layer of the model.
        b1 : np.array
            Current bias(es) for the first layer of the model.
        W2 : np.array, optional
            Current weights for the second layer of the model, if MLP.
            The default is None.
        b2 : np.array, optional
            Current biases for the second layer of the model, if MLP.
            The default is None.
        y2 : np.array, optional
            Output of second layer of model for the image(s) in x, if MLP
            The default is None.

        Returns
        -------
        W_new : np.array
            Updated weights for the first layer of the model.
        b_new : np.array
            Updated biases for the first layer of the model.
        or
        W1_new : np.array
            Updated weights for the first layer of the model, if MLP.
        b1_new : np.array
            Updated biases for the first layer of the model, if MLP.
        W2_new : np.array
            Updated weights for the second layer of the model, if MLP.
        b2_new : np.array
            Updated biases for the second layer of the model, if MLP.

        """
        m = x.shape[1]

        if self.hidden_size == None:
            # Network only has an output layer
            # Single layer perceptron
            diff = d - y1
```

```

            # Perform the weight update rule depending on what
            # loss function is being used
            if loss == "log_loss":
                W_new = W1 + lrn_rate*(1/m)*np.dot(x, diff)
                b_new = b1 + lrn_rate*(1/m)*np.sum(diff,
                                                   axis=0, keepdims=True)
            elif loss == "mse":
                W_new = W1 + lrn_rate*(1/m)*np.dot(x, diff*y1*(1 - y1))
                b_new = b1 + lrn_rate*(1/m)*np.sum(diff*y1*(1 - y1),
                                                   axis=0, keepdims=True)

            return W_new, b_new
        else:
            # Network has a hidden layer and output layer (MLP)
            diff = d - y2

            # Perform the weight update rule depending on what
            # loss function is being used
            if loss == "log_loss":
                W2_new = W2 + lrn_rate*np.dot(y1.T, diff).reshape(W2.shape)
                b2_new = b2 + lrn_rate*np.sum(
                    diff, axis=0, keepdims=True).reshape(b2.shape)
                W1_new = W1 + lrn_rate*np.dot(
                    x, np.dot(diff, W2.T)*y1*(1 - y1)).reshape(W1.shape)
                b1_new = b1 + lrn_rate*np.sum(
                    np.dot(diff, W2.T)*y1*(1 - y1),
                    axis=0, keepdims=True).reshape(b1.shape)
            elif loss == "mse":
                W2_new = W2 + lrn_rate*np.dot(
                    y1.T, diff*y2*(1 - y2)
                    ).reshape(W2.shape)
                b2_new = b2 + lrn_rate*np.sum(
                    diff*y2*(1 - y2), axis=0, keepdims=True).reshape(b2.shape)
                W1_new = W1 + lrn_rate*np.dot(
                    x,np.dot(diff*y2*(1 - y2), W2.T)*y1*(1 - y1)
                    ).reshape(W1.shape)
                b1_new = b1 + lrn_rate*np.sum(
                    np.dot(diff*y2*(1 - y2), W2.T)*y1*(1 - y1),
                    axis=0, keepdims=True).reshape(b1.shape)

            return W1_new, b1_new, W2_new, b2_new


    def evaluate(self, epoch,  train_or_val, data, true_labels,
                 W1, b1, W2=None, b2=None):
        """
        Evaluate the performance (accuracy) of the model on the entire
        training or validation dataset, using the current weights and biases
        of the model. Values of the accuracy are stored in arrays for writing
        to a csv file after training and plotting graphs.

        Parameters
        ----------
        epoch : int
            At which epoch the accuracy is being evaluated, used for storing
            accuracies for plotting later.
        train_or_val : string
            Indicated whether that data is training or validation, used for
```

```
            storing accuracies for plotting later. Possible values are
            "train" or "val".
        data : np.array
            Image data, either entire training or validation set.
        true_labels : np.array
            The true labels corresponding to images in data.
        W1 : np.array
            Current weights for the first layer of the model.
        b1 : np.array
            Current bias(es) for the first layer of the model.
        W2 : np.array, optional
            Current weights for the second layer of the model, if MLP.
            The default is None.
        b2 : np.array, optional
            Current biases for the second layer of the model, if MLP.
            The default is None.

        Returns
        -------
        acc : float
            Accuracy of model predictions on all data images, between 0 and 1.

        """
        m = data.shape[1]

        if self.hidden_size == None:
            # Network only has an output layer, no hidden layer
            # Single layer perceptron
            output = self.forward_propagation(data, W1, b1)
        else:
            # Network has a hidden layer and output layer (MLP)
            _, output = self.forward_propagation(data, W1, b1, W2, b2)

        if output.shape[1] > 1:
                # For multiclass problems, one-hot-encoded labels are used, so
                # to compute accuracy argmax must see which element is maximum
                # and hence the correct label.
                # e.g. output = [0,0,1,0] ----> output_labels = [2]
                output = np.argmax(output, axis = 1).reshape(m, 1)
                true_labels = np.argmax(true_labels, axis = 1).reshape(m, 1)

        # Compute the accuracy
        acc = float(np.sum(output == true_labels))/m

        # Record and display the accuracy
        if train_or_val == "train":
            # Forward propagation using training data
            # Store the training accuracy in train_acc_log for plotting etc
            # after training
            self.train_acc_log.insert(epoch - 1, acc)

        elif train_or_val == "val":
            # Forward propagation using validation data
            # Store the validation accuracy in val_acc_log for plotting etc
            # after training
            self.val_acc_log.insert(epoch - 1, acc)
            # Display the prediction vectors and true label vectors
            #print("True labels:")
            #print(true_labels)
```

```
            #print("Output:")
            #print(output)

        return acc


    def train(self, train_data, train_labels, val_data, val_labels,
              init_factor=1e-3, loss = "log_loss", lrn_rate=0.01, lr_decay=None,
              optimizer="sgd", epochs=20, max_accept_error=0, print_epochs=1):
        """
        Train the model for either a set number of epochs or until a maximum
        acceptable error has been reached. Training is done via forward
        and backward propagation through the model and using gradient-based
        weight update.

        Parameters
        ----------
        train_data : np.array
            The images used for training the network, of shape
            (features, examples).
        train_labels : np.array
            Labels corresponding to the images in train_data, of
            shape (examples, n_output).
        val_data : np.array
            The images used for validation of the network, of shape
            (features, examples).
        val_labels : np.array
            Labels corresponding to the images in val_data, of
            shape (examples, n_output).
        init_factor : float, optional
            For weight intialization this number scales the random values to
            be between 0 and factor.
            The default is 1e-3.
        loss : string, optional
            The loss function used for defining the weight update rules. Can
            be either "log_loss" or mean-square error loss, "mse.
            The default is "log_loss".
        lrn_rate : float, optional
            The learning rate for the weight update rule.
            Indicates how large steps are taken. The default is 0.01.
        lr_decay : float, optional
            The value used in the weight decay equation to gradually reduce
            the learning rate from its initial value after each weight update.
            The default is None.
        optimizer : string, optional
            The optimization technique used for updating the weights. Can
            be either batch gradient descent "batch_gd", looking at all
            training images before updating the weights, or stochastic
            gradient descent "sgd", looking at a single training images, then
            updating the weights. The default is "sgd".
        epochs : int, optional
            Number of iterations through whole dataset should
            before training is complete. The default is 20.
        max_accept_error : float, optional
            When training error reaches this value, training will stop.
            The default is 0.
        print_epochs : int, optional
            Number of how often the accuracies and epoch number
            should be displayed. The default is 1.
```

```
        Returns
        -------
        None.

        """
        m = train_data.shape[1]
        initial_lrate = lrn_rate # This is needed if using learing rate decay
                                 # so as to not ovewrite the initial learning
                                 # rate

        # Initialize the weights of the network
        if self.hidden_size == None:
            # Network only has an output layer
            W, b = self.initialize_parameters(init_factor)
        else:
            # Network has a hidden layer and output layer
            W1, b1, W2, b2 = self.initialize_parameters(init_factor)


        print("\nTraining has started...")
        i = 0 # Variable to increment upon weight updates, used for weight
              # decay equation
        # Each epoch use all the training data to update the weights of the
        # network
        for epoch in range(1, epochs + 1):

            if epoch%print_epochs == 0:
                # Only print the interation number every print_epochs
                # iteration
                print("Epoch " + str(epoch) + ":")

            if self.hidden_size == None:
                # Network has no hidden layer, just an output
                if optimizer == "sgd":
                    for example in np.random.permutation(m):
                        # Select images in random order
                        # Forward propagation to compute output
                        # Feed in an image at a time and update the weights
                        train_example = train_data[:, example].reshape(
                            train_data.shape[0], 1)
                        train_label = train_labels[example, :].reshape(
                            1,train_labels.shape[1])
                        y = self.forward_propagation(train_example,
                                                     W, b)
                        # Backward propagation to update weights
                        if lr_decay != None:
                            # Using learning rate decay, so calculated reduced
                            # learning rate based on what iteration it is
                            lrn_rate = initial_lrate*(1/(1 + lr_decay*i))
                        W, b = self.update_weights(y, train_example,
                                                   train_label,
                                                   loss, lrn_rate, W, b)
                        i += 1
                elif optimizer == "batch_gd":
                    # Update weights using all examples at a time
                    # Forward propagation to compute output
                    y = self.forward_propagation(train_data, W, b)
                    # Backward propagation to update weights
```

```
                W, b = self.update_weights(y, train_data, train_labels,
                                           loss, lrn_rate, W, b)

            # Calculate, store and display the training and
            # validation accuracy each epoch
            train_acc = self.evaluate(epoch, "train", train_data,
                                      train_labels, W, b)
            val_acc = self.evaluate(epoch, "val", val_data,
                                    val_labels, W, b)

            if epoch%print_epochs == 0:
                # Only display the accuracy every print_epochs interation
                print("train_accuracy = " + str(train_acc))
                print("validation_accuracy = " + str(val_acc))
            train_error = 1 - train_acc
            if train_error <= max_accept_error:
                # When max_accept_error is given exit the epoch loop
                # prematurely based on whether the error is small enough
                break

        else:
            # Network has a hidden layer
            if optimizer == "sgd":
                for example in np.random.permutation(m):
                    # Select images in random order
                    # Forward propagation to compute output
                    # Feed in an image at a time and update the weights
                    train_example = train_data[:,example].reshape(
                        train_data.shape[0], 1)
                    train_label = train_labels[example, :].reshape(
                        1, train_labels.shape[1])
                    y1, y2 = self.forward_propagation(train_example,
                                                      W1, b1, W2, b2)
                    # Backward propagation to update weights
                    if lr_decay != None:
                        # Using learning rate decay, so calculated reduced
                        # learning rate based on what iteration it is
                        lrn_rate = initial_lrate*(1/(1 + lr_decay*i))
                    W1, b1, W2, b2 = self.update_weights(y1, train_example,
                                                         train_label,
                                                         loss, lrn_rate,
                                                         W1, b1,
                                                         W2, b2, y2)
                    i += 1
            elif optimizer == "batch_gd":
                # Update weights using all examples at a time
                # Forward propagation to compute output
                y1, y2 = self.forward_propagation(train_data,
                                                  W1, b1, W2, b2)
                # Backward propagation to update weights
                W1, b1, W2, b2 = self.update_weights(y1, train_data,
                                                     train_labels,
                                                     loss, lrn_rate,
                                                     W1, b1,
                                                     W2, b2, y2)

            # Calculate, store and display the training and validation
            # accuracy each epoch and return error to compare with
            # max_accept_error, if given
```

```
                train_acc = self.evaluate(epoch, "train", train_data,
                                          train_labels, W1, b1, W2, b2)
                val_acc = self.evaluate(epoch, "val", val_data,
                                        val_labels, W1, b1, W2, b2)

                if epoch%print_epochs == 0:
                    # Only display the accuracy every print_epochs interation
                    print("train_accuracy = " + str(train_acc))
                    print("validation_accuracy = " + str(val_acc))
                train_error = 1 - train_acc
                if train_error <= max_accept_error:
                    # When max_accept_error is given exit the epoch loop
                    # prematurely based on whether the error is small enough
                    break

        print("\nTraining has finished")
        print("\nFinal training accuracy: " + str(self.train_acc_log[-1]))
        print("\nFinal validation accuracy: " + str(self.val_acc_log[-1]))


    def plot(self, file_path, epoch_steps=1):
        """
        Method to plot the training and validation accuracy,
        for each epoch, against the epoch number and save the image.

        Parameters
        ----------
        file_path : string
            File path where the csv will be saved.
        epoch_steps : int, optional
            The interval steps on the x axis of the graph.
            The default is 1.

        Returns
        -------
        None.

        """
        # Retrieve the training accuracy and validation accuracy data
        acc = self.train_acc_log
        val_acc = self.val_acc_log

        epochs = range(1, len(acc) + 1)

        # Plot the graph
        plt.figure(figsize=(20,15))
        plt.plot(epochs, acc, 'r', label="Training accuracy")
        plt.plot(epochs, val_acc, 'b', label="Validation accuracy")
        plt.legend(loc=0, fontsize=18)
        plt.grid(True)
        plt.xticks(np.arange(0, epochs[-1] + 1, step=epoch_steps),
                   fontsize=18)
        plt.xlim(1, epochs[-1] + 1)
        plt.xlabel("Epochs", fontsize=20)
        plt.yticks(np.arange(0, 1.1, step=0.1), fontsize=18)
        plt.ylim(0, 1)
        plt.ylabel("Accuracy", fontsize=20)

        # Save the file and display
```

```python
        plt.savefig(file_path)
        print("\nTraining and validation accuracy graph printed successfully!")
        plt.show()


    def save(self, file_path):
        """
        Method to save training and validation accuracy data,
        for each epoch, as a csv file.

        Parameters
        ----------
        file_path : string
            File path where the csv will be saved.

        Returns
        -------
        None.

        """
        # Retrieve the training accuracy and validation accuracy
        acc = np.array(self.train_acc_log).reshape(len(self.train_acc_log), 1)
        val_acc = np.array(self.val_acc_log).reshape(len(self.val_acc_log), 1)
        training_log = np.hstack((acc, val_acc))

        # Store data into a pandas dataframe and save to a csv file
        column_headers = ["Training accuracy", "Validation accuracy"]
        df = pd.DataFrame(data = training_log, columns = column_headers)
        df.index += 1
        df.to_csv(file_path)
        print("\nData saved in a csv file to file path successfully!")
```

A.3 Task 1: Single Perceptron

```python
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 20 17:48:37 2020

@author: Jason
"""

# EEEN4/60151 Machine Learning Laboratory
# 1. Single Perceptron


import numpy as np
from Model_class import Model
from ImageData_class import ImageData

# Create the 0 and 1 images as numpy array and the corresponding labels
ones = np.array([[[0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0]],
                 [[0,1,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
```

```
                   [0,1,1,1,0]],
                  [[0,0,1,0,0],
                   [0,1,1,0,0],
                   [0,0,1,0,0],
                   [0,0,1,0,0],
                   [0,0,1,0,0]],
                  [[0,0,0,1,0],
                   [0,0,1,1,0],
                   [0,0,1,0,0],
                   [0,1,1,0,0],
                   [0,1,0,0,0]],
                  [[0,0,0,0,1],
                   [0,0,0,1,0],
                   [0,0,1,0,0],
                   [0,1,0,0,0],
                   [1,0,0,0,0]],
                  [[0,0,0,0,1],
                   [0,0,1,0,0],
                   [0,0,1,0,0],
                   [0,0,1,0,0],
                   [0,0,1,0,0]]
                  ])

zeros = np.array([[[0,1,1,1,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,1,1,1,0]],
                   [[0,0,1,0,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,0,1,0,0]],
                   [[0,0,1,0,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,1,0,1,0],
                    [0,1,1,1,0]],
                   [[0,0,1,1,0],
                    [0,1,0,0,1],
                    [0,1,0,0,1],
                    [0,1,0,1,0],
                    [0,1,1,0,0]],
                   [[1,1,1,1,1],
                    [1,0,0,0,1],
                    [1,0,1,0,1],
                    [1,0,0,0,1],
                    [1,1,1,1,1]],
                   [[0,1,1,1,0],
                    [1,0,0,0,1],
                    [1,0,0,0,1],
                    [1,0,0,0,1],
                    [0,1,1,1,0]]
                   ])

one_labels = np.array([[1],
                        [1],
                        [1],
                        [1],
```

```
                        [1],
                        [1]
                       ])

zero_labels = np.array([[-1],
                         [-1],
                         [-1],
                         [-1],
                         [-1],
                         [-1]
                        ])

# Preprocess the images and labels using the ImageData class
# For this task the image will need to be in the format
# (image, example) with flattened images
# The training set will have 8 examples (4 1s, 4 0s) and the
# validation set will have 4 examples (2 1s, 2 0s)
data = ImageData()
data.get_data(images1=ones,
              image_labels1=one_labels,
              images2=zeros,
              image_labels2=zero_labels
              )
data.data_preprocess(split_data=True,
                     normalize=False,
                     flatten=True,
                     img_first_format=True,
                     one_hot_encode=False
                     )

# Define and train a single perceptron model for learning this dataset, then
# plot the training and validation accuracy as a function of epoch and save
# this data to a csv file
input_size = data.train_data.shape[0]
output_size = data.train_labels.shape[1]
save_dir = "C:/Users/Jason/Documents/"
graph_save_path = save_dir + "graph.png"
data_save_path = save_dir + "data.csv"

single_layer_perceptrons = Model(input_size,
                                 output_size,
                                 hidden_size=None,
                                 output_activation="sign"
                                 )
single_layer_perceptrons.train(data.train_data,
                               data.train_labels,
                               data.val_data,
                               data.val_labels,
                               init_factor=0.1,
                               lrn_rate=1e-2,
                               max_accept_error=0
                               )
single_layer_perceptrons.plot(graph_save_path)
single_layer_perceptrons.save(data_save_path)
```

A.4 Task 2: Single Layer Perceptrons

```
# -*- coding: utf-8 -*-
"""
```

```
Created on Fri Nov 20 18:00:48 2020

@author: Jason
"""

# EEEN4/60151 Machine Learning Laboratory
# 2. Single Layer Perceptrons

import mnist
from ImageData_class import ImageData
from Model_class import Model

# Preprocess the images and labels using the ImageData class
# For this task the image will need to be in the format
# (image, example) with flattened images and one-hot-encoded
# labels (of size (n_examples, 10) as there are 10 possible labels 0-9)
# MNIST has 60,000 training examples and 10,000 validation examples
data = ImageData()
data.get_data(dataset=mnist)
data.data_preprocess(pre_shuffle=False,
                     normalize=True,
                     flatten=True,
                     img_first_format=True,
                     one_hot_encode=True
                     )

# Define and use a single layer of perceptrons model for learning the MNIST
# data, then plot the training and validation accuracy as a function of epoch
# and save this data to a csv file
input_size = data.train_data.shape[0]
output_size = data.train_labels.shape[1]
save_dir = "C:/Users/Jason/Documents/"
save_name = "save_name"
graph_save_path = save_dir + save_name + "-graph.png"
data_save_path = save_dir + save_name + "-data.csv"

single_layer_perceptrons = Model(input_size,
                                 output_size,
                                 hidden_size=None,
                                 output_activation="sigmoid"
                                 )
single_layer_perceptrons.train(data.train_data,
                               data.train_labels,
                               data.val_data,
                               data.val_labels,
                               init_factor=1e-3,
                               loss="mse",
                               lrn_rate=5e-1,
                               lr_decay=1e-6,
                               epochs=20
                               )
single_layer_perceptrons.plot(graph_save_path, epoch_steps=20)
single_layer_perceptrons.save(data_save_path)
```

A.5 Task 3: Multilayer Perceptron

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 20 18:07:44 2020
```

```python
@author: Jason
"""

# EEEN4/60151 Machine Learning Laboratory
# 3. Multilayer Perceptrons

import mnist
from Model_class import Model
from ImageData_class import ImageData

# Preprocess the images and labels using the ImageData class
# For this task the image will need to be in the format
# (image, example) with flattened images and one-hot-encoded
# labels (of size (n_examples, 10) as there are 10 possible labels 0-9)
# MNIST has 60,000 training examples and 10,000 validation examples
data = ImageData()
data.get_data(dataset=mnist)
data.data_preprocess(pre_shuffle=False,
                     normalize=True,
                     flatten=True,
                     pad=False,
                     img_first_format=True,
                     one_hot_encode=True
                     )

# Define and use a multi layer perceptrons (MLP) model for learning the MNIST
# data, then plot the training and validation accuracy as a function of epoch
# and save this data to a csv file
input_size = data.train_data.shape[0]
hidden_size = 20
output_size = data.train_labels.shape[1]
save_dir = "C:/Users/Jason/Documents/"
save_name = "save_name"
graph_save_path = save_dir + save_name + "-graph.png"
data_save_path = save_dir + save_name + "-data.csv"

multi_layer_perceptrons = Model(input_size=input_size,
                                output_size=output_size,
                                hidden_size=hidden_size,
                                output_activation="sigmoid"
                                )
multi_layer_perceptrons.train(data.train_data,
                              data.train_labels,
                              data.val_data,
                              data.val_labels,
                              init_factor=1e-3,
                              loss="mse",
                              lrn_rate=1e-2,
                              optimizer="sgd",
                              epochs=20
                              )
multi_layer_perceptrons.plot(graph_save_path, epoch_steps=1)
multi_layer_perceptrons.save(data_save_path)
```

A.6 Task 4: LeNet 5 for handwritten digit classification (on MNIST dataset)

```python
# -*- coding: utf-8 -*-
"""
```

```python
Created on Fri Nov 20 18:09:20 2020

@author: Jason
"""

# EEEN4/60151 Machine Learning Laboratory
# 4. LeNet 5 for handwritten digit classification
#    (on MNIST dataset)

import mnist
import keras
from keras import layers
from ImageData_class import ImageData
from Model_metric_plotter_saver import save_history_to_csv

# Preprocess the images and labels using the ImageData class
# For this task the image will need to be in the format
# (example, img_height, img_width, 1) with padding to make the
# images 32x32 rather than 28x28.
# Labels will be one-hot-encoded (of size (n_examples, 10) as
# there are 10 possible labels 0-9)
# MNIST has 60,000 training examples and 10,000 validation
# examples
data = ImageData()
data.get_data(dataset=mnist)
data.data_preprocess(split_data=False,
                     flatten=False,
                     pad=True,
                     img_first_format=False,
                     one_hot_encode=True
                     )

#  Defin a LeNet 5 inspired CNN model using the keras deep learning library
LeNet5 = keras.Sequential([
    layers.Conv2D(filters=6, kernel_size=(5,5),
                  name="C1", input_shape=(32,32,1)),
    layers.AveragePooling2D((2,2), (2,2), name="S2"),

    layers.Conv2D(filters=16, kernel_size=(5,5), name="C3"),
    layers.AveragePooling2D((2,2), (2,2), name="S4"),

    layers.Flatten(),

    layers.Dense(units=120, activation="tanh", name="C5"),

    layers.Dense(units=84, activation="tanh", name="F6"),

    layers.Dense(units=10, activation="softmax", name="OUTPUT")
])

LeNet5.compile(optimizer="sgd", loss="mse", metrics=["accuracy"])
history = LeNet5.fit(x=data.train_data,
                     y=data.train_labels,
                     epochs=40,
                     validation_data=(data.val_data, data.val_labels)
                    )


# Plot the training and validation accuracies as a function of epoch
```

```python
# and save the graph
save_dir = "C:/Users/Jason/Documents/"
save_path = save_dir + "LeNet5.csv"


save_history_to_csv(history, save_path, ["accuracy"])
```

A.7 Task 5: Self-Organising Map (SOM)

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 21 21:52:43 2020

@author: Jason
"""

# EEEN4/60151 Machine Learning Laboratory
# 5. Self-Organizing Map (SOM)


import numpy as np
import matplotlib.pyplot as plt


# FUNCTION DEFINITIONS

def generate_vector(dims, index_list):
    """
    Funtion for generating the vectors which represent the animals.
    Parameters
    ----------
    dims : int
        Desired dimensions of the output vector.
    index_list : list
        List of which indices in the output vector are to be set to 1.

    Returns
    -------
    vector : np.array
        Vector representing an animal, made of 0s expect where index_list
        specified there should be a 1.

    """
    vector = np.zeros((dims,1))
    for index in index_list:
        vector[index,0] = 1
    return vector


def initialize_weights(height, width, depth):
    """
    Random initializes a weight tensor of size (l,m,n) for a
    self-organizing map (SOM)

    Parameters
    ----------
    height : int
        height of weight tensor (l)
    width : int
```

```
        Width of weight tensor (m)
    depth : int
        Depth of weight tensor - to be equal to SOM input vector dimensions(n)

    Returns
    -------
    np.array
        The randomly initialized weight tensor.

    """
    return np.random.rand(height, width, depth)


def winning_neuron(input_vec, weights):
    """
    Selects the i and j index values of weights which has the smallest
    Euclidean distance to input_vec

    Parameters
    ----------
    input_vec : np.array
        Animal vector.
    weights : np.array
        SOM weights.

    Returns
    -------
    u : int
        The ith component index of the winning neuron.
    v : int
        The jth component index of the winning neuron.

    """
    # Compute the Euclidean distance between input_vec and each of the
    # vectors at position ij in the weights tensor.
    d = np.zeros((weights.shape[0], weights.shape[1]))
    for i in range(weights.shape[0]):
        for j in range(weights.shape[1]):
            weight_ij = weights[i,j,:].reshape(input_vec.shape)
            diff = input_vec - weight_ij
            d[i,j] = np.sqrt(np.dot(diff.T, diff))
    # Select the winning neuron
    u, v = np.unravel_index(np.argmin(d, axis=None), d.shape)

    return u, v


def update_weights(u, v, t, sigma, input_vec, old_weights):
    """
    Update the weights corresponding to position in the SOM map.

    Parameters
    ----------
    u : int
        The ith component index of the winning neuron.
    v : int
        The jth component index of the winning neuron.
    t : int
        Iteration number
```

```python
    sigma : TYPE
        DESCRIPTION.
    input_vec : np.array
        Vector of the animal being used to update the SOM weights.
    old_weights : np.array
        Current weight tensor for the SOM.

    Returns
    -------
    new_weights : np.array
        Weight tensor with updated values.

    """
    alpha = 100/(200 + t)
    height = old_weights.shape[0]
    width = old_weights.shape[1]

    new_weights = np.zeros(old_weights.shape)
    for i in range(height):
        for j in range(width):
            old_weight_ij = old_weights[i,j,:].reshape(input_vec.shape)
            eta =  np.exp(-((i - u)**2 + (j - v)**2) / 2*sigma**2)
            diff = input_vec - old_weight_ij
            new_weights[i,j,:] = (old_weight_ij + alpha*eta*(diff)).reshape(
                new_weights[i,j,:].shape)

    return new_weights


def select_rand_row(input_vectors):
    """
    Selects a random rows of a stack of vectors, i.e chooses a random animal
    vector.

    Parameters
    ----------
    input_vectors : np.array
        Stack of animal vectors.

    Returns
    -------
    random_vec : np.array
        One animal vector randomly chosen from input_vectors.

    """
    rand_int = np.random.randint(0, input_vectors.shape[1])
    random_vec = input_vectors[:,rand_int].reshape(input_vectors.shape[0],1)

    return random_vec


def plot_SOM(input_vectors, input_labels, label_colors, label_markers,
             trained_weights, save_path, iteration):
    """


    Parameters
    ----------
    input_vectors : np.array
```

```python
        Stacked animal vectors.
    input_labels : list
        List of animal names corresponding to the components of input_vectors.
    label_colors : list
        List of colors for plotting points corresponding to each animal.
    label_markers : list
        List of maker types for plotting point corresponding to each animal.
    trained_weights : np.array
        The trained weights for the SOM.
    save_path : string
        Where to save the plots of the SOM.
    iteration : int
        Iteration number, used for the save name of the plotted graph.

    Returns
    -------
    None.

    """
    fig, ax = plt.subplots(figsize = (6,6))

    # Plot points for each animal vector on SOM
    for i in range(input_vectors.shape[1]):
        label = input_labels[i]

        marker = label_markers[i]

        color = label_colors[i]

        input_vec = input_vectors[:,i]
        u, v = winning_neuron(input_vec, trained_weights)
        ax.scatter(v, u, s=150, marker=marker, color=color)

        if label == "Zebra" or label == "Hawk" or label == "Goose":
            plt.annotate(label, (v, u),
                         xytext=(8,-8), textcoords="offset points")
        else:
            plt.annotate(label, (v, u),
                         xytext=(8,8), textcoords="offset points")


    plt.xlim(0,10)
    plt.ylim(0,10)
    plt.grid(True)

    plt.savefig(save_path + "Pass_" + str(iteration))
    plt.show()


def train_and_plot_SOM(save_path, input_vectors, labels,
                       label_colors, label_markers,
                       map_size=(10,10), start_sigma=3,
                       num_iter=100001, plot_iter=2000):
    """
    Train the SOM and plot the SOM multiple times during training.

    Parameters
    ----------
    save_path : string
```

```
        Where to save the plots of the SOM.
    input_vectors : np.array
        Stacked animal vectors.
    labels : list
        List of animal names corresponding to the components of input_vectors.
    label_colors : list
        List of colors for plotting points corresponding to each animal.
    label_markers : list
        List of maker types for plotting point corresponding to each animal.
    map_size : tuple, optional
        Size of the SOM. The default is (10,10).
    start_sigma : float, optional
        The starting value of sigma, which will be gradually decreased to 1.
        The default is 3.
    num_iter : int, optional
        How many iteration of training the SOM. The default is 10000.
    plot_iter : int, optional
        After how many iterations the SOM should be plotted and saved.
        The default is 2000.

    Returns
    -------
    None.

    """
    print("Training has started...")


    height, width = map_size
    n = input_vectors.shape[0]

    weights = initialize_weights(height, width, n)

    for t in range(num_iter):
        # Vary sigma from 3 to 1 as iterations go on
        sigma = start_sigma - (start_sigma - 1)*(t/num_iter)

        input_vec = select_rand_row(input_vectors)
        u, v = winning_neuron(input_vec, weights)
        weights = update_weights(u, v, t, sigma, input_vec, weights)

        if t%1000 == 0:
            print("Iteration: " + str(t))
        if t % plot_iter == 0:
            plot_SOM(input_vectors,labels,label_colors,label_markers,
                     weights,save_path,t)

    print("\nTraining Completed!")



# MAIN

# Represent the animals as 13-dimensional vectors
dims = 13
Dove = generate_vector(dims, [0,3,8,11])
Hen = generate_vector(dims, [0,3,8])
Duck = generate_vector(dims, [0,3,8,11,12])
Goose = generate_vector(dims, [0,3,8,11,12])
```

```python
Owl = generate_vector(dims, [0,3,8,9,11])
Hawk = generate_vector(dims, [0,3,8,9,11])
Eagle = generate_vector(dims, [1,3,8,9,11])
Fox = generate_vector(dims, [1,4,5,9])
Dog = generate_vector(dims, [1,4,5,10])
Wolf = generate_vector(dims, [1,4,5,7,9,10])
Cat = generate_vector(dims, [0,4,5,9])
Tiger = generate_vector(dims, [2,4,5,9,10])
Lion = generate_vector(dims, [2,4,5,7,9,10])
Horse = generate_vector(dims, [2,4,5,6,7,10])
Zebra = generate_vector(dims, [2,4,5,6,7,10])
Cow = generate_vector(dims, [2,4,5,6])

# Put vector representation together in shuffled matrix
animal_inputs = np.hstack((Dove,Hen,Duck,Goose,Owl,Hawk,Eagle,Fox,Dog,Wolf,
                          Cat,Tiger,Lion,Horse,Zebra,Cow))

# Define lists for plotting and annotating SOM plot
animal_labels = ["Dove","Hen","Duck","Goose","Owl","Hawk","Eagle","Fox","Dog",
                 "Wolf","Cat","Tiger","Lion","Horse","Zebra","Cow"]
color1 = "cornflowerblue"
color2 = "blue"
color3 = "midnightblue"
label_colors = [color1,color1,color1,color1,color1,color1,color2,color2,
                color2,color2,color1,color3,color3,color3,color3,color3]
label_markers = ["D","D","D","D","D","D","D",
                 "s","s","s","s","s","s","s","s","s"]

# Train, plot and save SOM
save_path = "C:/Users/Jason/Documents/SOM7-"
train_and_plot_SOM(save_path, animal_inputs, animal_labels,
                   label_colors, label_markers)
```