Programming and Mathematics for Artificial Intelligence

William Dibble | IN3063 | 01/01/2023

GitHub.com/WDibble/IN3063-MathsAI

Task 1 – Python Path-Finding Game

The objective of this task was to develop a game in Python. The game was required to generate a grid of a specified size, with numbers ranging from 0 to 9 in each square of the grid. The game should then use a pathfinding algorithm to plot the shortest path from the top left square to the bottom right square.

I went through several key steps to achieve the development of this program. Firstly, I imported the two libraries I would need: matplotlib pyplot and numpy. Matplotlib pyplot is a library for creating visualizations, such as charts and graphs, which will be used to create the grid. And numpy is a library for numerical computing in Python, which will be used to develop the pathfinding algorithm.

Next, I declared an np.random.seed function that's called with an argument of 46892. This set the seed for the random number generator in numpy, as setting the seed ensured that the same random values were generated every time the code was run, which was useful for debugging and testing purposes but was also a requirement of the task.

I then created user prompts, the first to enter the grid size, which is stored as the variables grid_width and grid_height. These values are used to determine the size of the grid that will be generated and displayed. The user is also prompted to enter the distribution parameters, which are stored as the variables distribution_mean and distribution_stddev. These values are used to determine the shape of the normal distribution that will be used to generate the random number values for the grid, which is a type of statistical distribution that is symmetrical around the mean and has a bell-shaped curve.

Next, I generated a grid using a nested list comprehension, with the outer list comprehension generating the rows of the grid, and the inner list generating the columns. I then used the np.around function to round the values to the nearest integer, and the np.clip function to ensure that all values are between 0 and 9. The astype function was then used to convert the values to integers. I implemented these steps to ensure that the grid was only filled with random integer values within the specified range. I then defined the start and goal positions as the coordinates (0, 0) and (grid_width - 1, grid_height - 1), respectively. These positions represented the start and end points for the pathfinding algorithm.

After this, I needed to develop the heuristic algorithm as outlined in the task specification. The algorithm did not need to be optimised to perform well but needed to be better than random movements and would be the baseline for the Dijkstra's algorithm that I'd develop next. I started my heuristic algorithm by taking in the start node and the end node as parameters and defined a set that I could use to store the visited nodes, including the start node. I also defined a dictionary in which I could store the came_from values of the nodes. I then created a loop that continues if the current position is not the

end position and within the loop, the function calculates the position of the neighbour using the dx and dy variables and the position of the current node. If the neighbour is out of bounds or has been visited, it’s skipped.

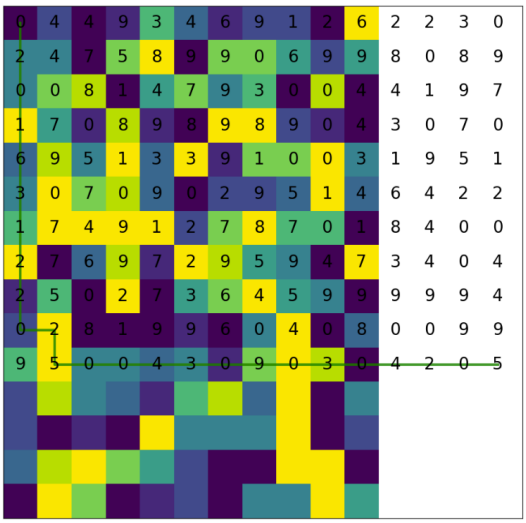
If the neighbour is within bounds and has not been visited, the function adds it to the visited set and sets the came_from value of the neighbour to the current node in the came_from_dict dictionary. The current position is then set to the neighbour and the loop continues. After the loop has finished, the function returns the came_from_dict dictionary, which can be used to construct the shortest path.

Using this algorithm, I began working on a separate but similar Dijkstra algorithm, as specified by the task. I had to create a simple version that was close to the original and that uses a priority queue. To start, as I did the simple algorithm, I kept the start and end positions as arguments and kept the return of the dictionary of the shortest path. I also kept the visited nodes set to store the nodes that have already been visited. I expanded the queue to a min-priority queue to store the nodes that still need to be visited, with the start node added to the queue with a priority of 0. I created a new dictionary named cost_dict to store the costs of the nodes and kept the original came-from dictionary to store the came_from values.

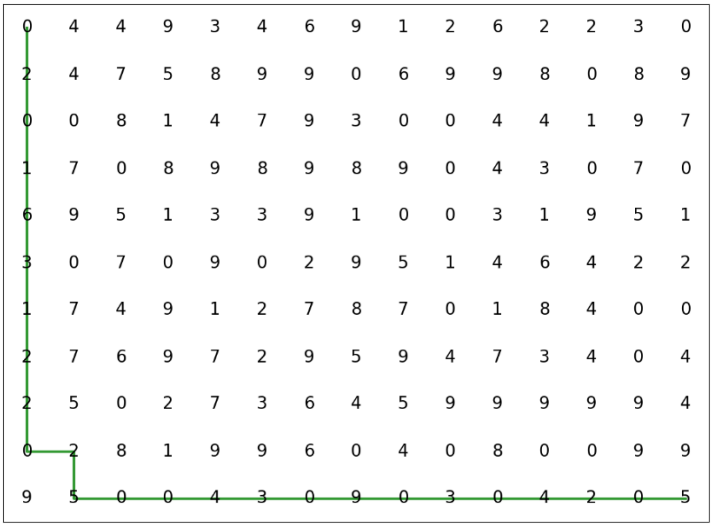
The algorithm then proceeds in much the same way as the simple algorithm, with the main difference being that it considers the costs (distances) of different paths and updates the came_from values of the nodes to construct the shortest path and uses the minimum priority queue to prioritise the nodes with the lowest cost and implements a loop to repeat this until the goal node is reached. Whereas the original simple algorithm starts at the beginning position and moves towards the end position in a straight line until it reaches the end or an obstacle, without considering the same factors that Dijkstra does. This means the simple algorithm would find a path from beginning to end, but it likely would not be the shortest.

Once the Dijkstra’s algorithm loop had finished examining all the neighbours of the current node, it continues to the next iteration, taking the next lowest cost node from the queue and repeating the process until the goal node is reached or the queue becomes empty. If the goal is reached, the function returns the came_from dictionary, which contains the shortest path from the start node to the goal node as a series of nodes that can be followed in reverse order. If the queue becomes empty before the goal is reached, it means that there is no path between the start and goal nodes and the function returns an empty dictionary. After creating the final algorithm, I called it with the input of the start and end nodes and stored the shortest path (came_from_dict) dictionary in a variable called "came_from".

I then visualised the grid and the pathfinding process using the matplotlib library, creating a figure with a specified size using the plt.figure function. This is when I ran into a persistent problem that took some time to resolve. Whenever I tried to display the grid, the shading that represented the values of the numbers would be at a different orientation from the numbers themselves.



To resolve this, I had to rotate the grid using the np.rot90 function, display the grid with an alpha of 0 for the shading, and rotate the grid back to its original position ready to plot the pathfinding line. Finally, the shortest path was plotted on top of the grid using the came_from dictionary and the start and goal positions. The resulting visualisation shows the shortest path from the start node to the goal node on the grid, as well as the values of each cell in the grid.



Task 2 – Neural Network for MNIST Dataset

Implementation of the CNN for CIFAR-10

Firstly, I imported the NumPy and Keras libraries. For this project, I used the Keras libraries to load the dataset and the NumPy libraries to form the neural network.

I loaded the MNIST dataset, which is a dataset of handwritten digits, and split it into training and test sets, represented by the variables X_train, y_train, X_test, and y_test. I then pre-processed the data by reshaping it into a 2D array of size (60000, 784) for the training set and (10000, 784) for the test set, and converted the data to floating point values, normalising it by dividing by 255.

I converted the labels to categorical variables using the NumPy eye function, which created a matrix with a 1 at the index corresponding to the label and 0s everywhere else. I did this to facilitate the calculation of the cross-entropy loss later. I then initialised the weights and biases for the three layers of the neural network using random values and used dictionaries to store the weights and biases for each layer.

After this, I set the learning rate to 0.001 and the dropout rate to 0.2. I also set the number of epochs to 20 and the batch size to 8. I calculated the number of batches by dividing the number of training examples by the batch size. I also initialized some empty lists to store the mean accuracies and mean losses for each epoch.

I then defined the sigmoid function, which takes in a matrix of input values and returns the sigmoid of each element. I also defined the sgd_optimiser function, which updates the weights and biases using the gradient descent algorithm. After this, I defined the cross_entropy_loss function, which calculates the cross-entropy loss between the true labels and the predicted labels, which was done by calculating the negative sum of the element-wise product of the true labels and the log of the predicted labels. I also included a smoothing factor to avoid division by zero in the log function.

I also defined a test suite to verify that the sigmoid function was working correctly. The first test checked that the sigmoid function returned the expected output for a range of input values and the second tested to see if the sigmoid function was stable when the input was very large, which can be an issue with some activation functions.

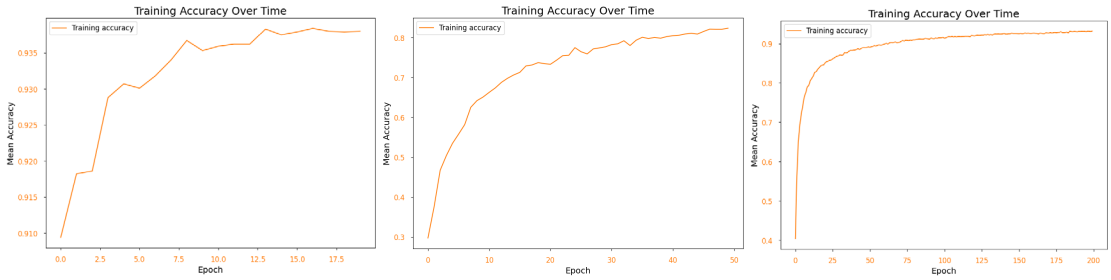
Finally, I began the training loop, which worked as a loop over the number of epochs. In each epoch, the training data was shuffled using a random permutation. This was important as the order of the training examples can have an impact on the optimisation process. I then looped over the number of batches, extracting a batch of training data and labels for each iteration. Next, I performed the forward pass through the network by computing the dot product of the input data and the weights, adding the biases, and applying the sigmoid activation function. I then performed the backward pass by calculating the gradients of the weights and biases using the gradient descent algorithm, and applied a dropout regularisation technique, which randomly set some of the

activations to zero during training to prevent overfitting. Finally, I updated the weights and biases using the optimiser function and calculated the mean accuracy and mean loss for each epoch. This was done to generate graphs that show these factors over time to analyse how the model parameters affected the accuracy outcome.

Overall, the purpose of this program was to train a neural network to classify handwritten digits using the MNIST dataset. The code used the sigmoid and ReLU activation functions in the first and second layers, respectively, and the softmax activation function in the output layer. It also used the SGD optimizer and the cross-entropy loss function to update the weights and biases during training. The code achieved this by performing forward and backward propagation through the network and updating the weights and biases according to the gradients calculated during backward propagation.

Analysis of the Results

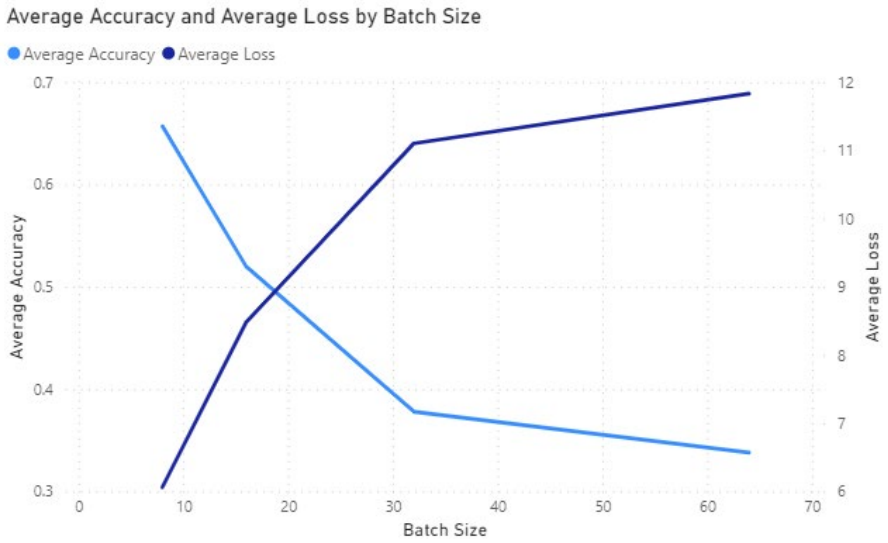
To first analyse and optimise my model, I ran three tests using my default parameters at twenty, fifty, and two hundred epochs. This allowed me to see that the longer I ran my model, the better accuracy the model could produce. It did appear, however, that my model was topping out at around one hundred epochs, with the accuracy increasing very slowly after this point.



Mean accuracy of the model at 20, 50, and 200 Epochs

In my next test, the batch size of the neural network was varied to observe its effect on the accuracy of the model. The batch sizes used were 8, 16, 32, and 64.

The results showed that the accuracy of the network increased as the batch size decreased. One potential reason for this is that using a smaller batch size allows the model to update its weights and biases more frequently, leading to faster convergence and potentially better performance.



It is, however, important to note that using a smaller batch size also increases the computational cost of training the model, as the model must make more updates in each epoch, taking more time for each model to complete. Therefore, for my next test, I set the batch size as 8 for all iterations as this meant I could compute 20 epochs in around 15 minutes.

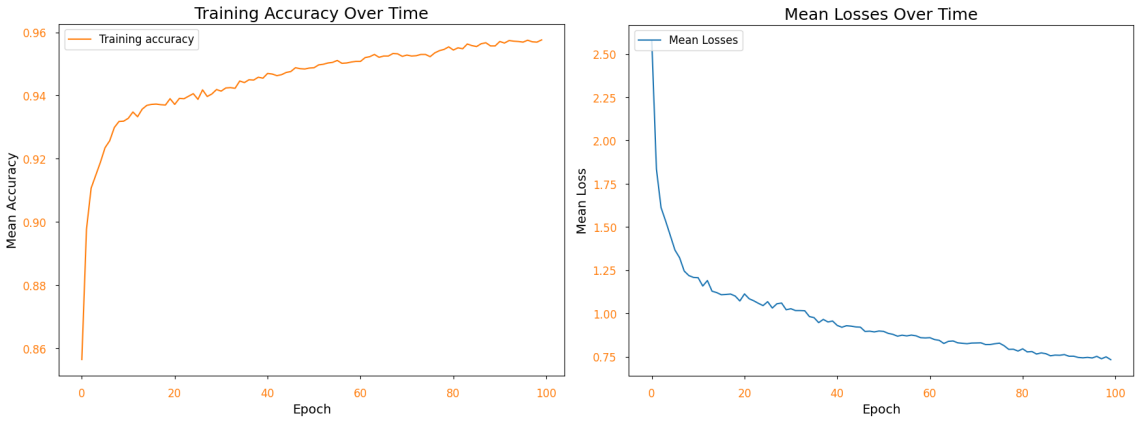
For my next test, I used a variety of learning rates and dropout rates to discover the most optimal parameters for my model. I ran 16 tests in total and plotted the results of the mean accuracy of each iteration on a heatmap.

		Learning Rate			
		0.001	0.0033	0.0066	0.01
Dropout	0.2	0.657	0.804	0.816	0.846
	0.3	0.659	0.83	0.885	0.878
	0.4	0.771	0.864	0.889	0.908
	0.5	0.723	0.884	0.913	0.915

The heatmap of the learning rates and dropout rates shows that higher values of both tend to lead to higher accuracy. This makes sense, as higher learning rates can allow the model to make more rapid progress towards the optimal solution, and higher dropout rates can help prevent overfitting by randomly turning off a larger number of neurons during training.

Overall, it was important to carefully tune the hyperparameters of the neural network to achieve the best possible performance. The results of this experiment suggest that it may be beneficial to use smaller batch sizes and higher learning rates and dropout rates to achieve high accuracy. It is, however, important to note that these results likely will not hold for all datasets and models, and it may be necessary to try a range of different hyperparameter values to find the optimal combination.

My final iteration of the model ran for 100 epochs, with a dropout rate of 0.5, a batch size of 8, and a learning rate of 0.01. This yielded an overall mean accuracy of 94.5% and an overall mean loss of only 0.97%, which are very successful results that indicate the neural network is accurate and efficient.



It would have been beneficial to include the option for the user to choose the number of layers in the model as outlined in the specification, because the number of layers can significantly impact the performance of the model. Allowing me to choose the number of layers would have given me more control over the model and potentially resulted in better performance as it would have been another parameter that I could have tweaked to gain the optimal accuracy rate. However, due to time constraints, I was unable to implement this feature.

Task 3 – Neural Network using PyCharm

Implementation of the CNN for CIFAR-10

The first step in this task was to implement a convolutional neural network using PyTorch and either the CIFAR-10 or the Fashion-MNIST dataset. I chose the CIFAR dataset, which consists of 60,000 32x32 colour training images and 10,000 test images, labelled over 10 categories. The first step was to download and load the CIFAR-10 dataset using the torch vision library. I then split the dataset up into a training set and a test set, with the training set comprising 80% of the data and the test set comprising the remaining 20%.

Next, I created data loaders for the training and test sets, which was an iterator that returns a batch of data, with the batch size specified in the code as 8. The data loaders allowed the model to train on the training set and test on the test set in mini batches, rather than all at once.

I then defined the network in the CNN class, which inherited from PyTorch's nn.Module class. The initialisation method of the class defined the layers of the model, which included two convolutional layers, two fully connected layers, and a final output layer. The forward method defined the forward pass of the model, which takes an input image and passes it through the convolutional layers, followed by the fully connected layers, and finally the output layer, which returned the logits for the input image.

The model was then defined as an instance of the CNN class, and a loss function and an optimiser were defined. The loss function used is the cross-entropy loss, which as mentioned in task 2, is a common choice for classification tasks and the optimiser used was stochastic gradient descent (SGD) with a learning rate of 0.001, which is generally a good starting point for learning rates.

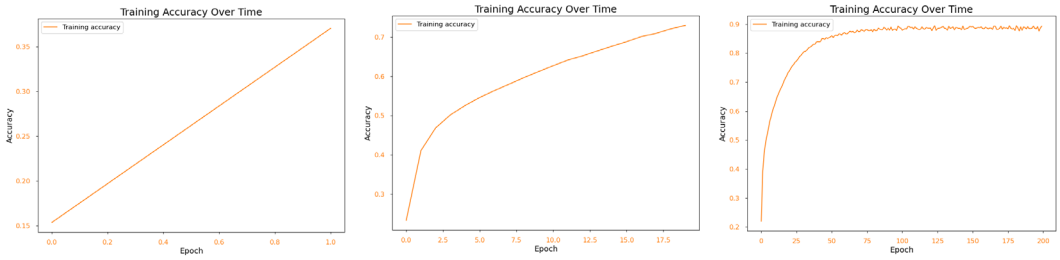
I then trained the model on the training set using mini batch gradient descent by iterating over the training set and optimising the model's parameters using the loss and optimiser defined earlier. I then evaluated the model’s performance on the test set at the end of each epoch, and the training loop continued for a specified number of epochs, with the model's accuracy on the test set printed at the end of each epoch.

The model's final accuracy on the test set was then calculated and printed. This was an important metric for evaluating the model's performance, as it indicated how well the model generalised to new data.

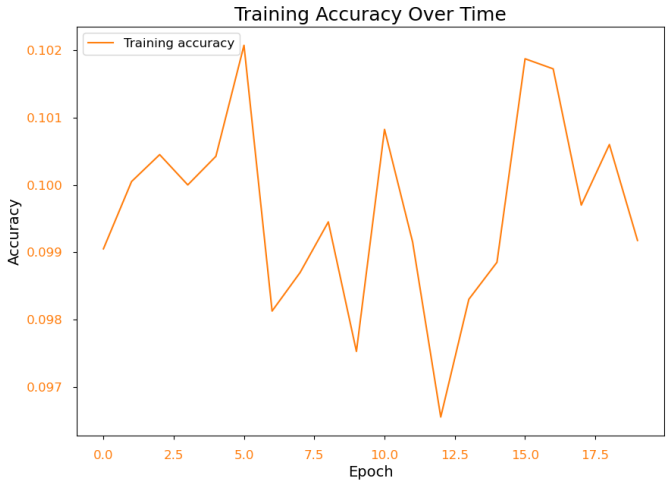
Overall, the program I made demonstrated how I had implemented a simple CNN in PyTorch for the CIFAR-10 dataset, and how I had trained and evaluated the model on the dataset using mini batch gradient descent. I could also modify the code to try different models and hyperparameters, to see if the outcome changed.

Analysis of the Results

I then ran the model at 2, 20, and 200 epochs, to measure how the performance of my model changed over time. The learning rate of the optimiser was set to 0.001 and the batch size was set to 8.

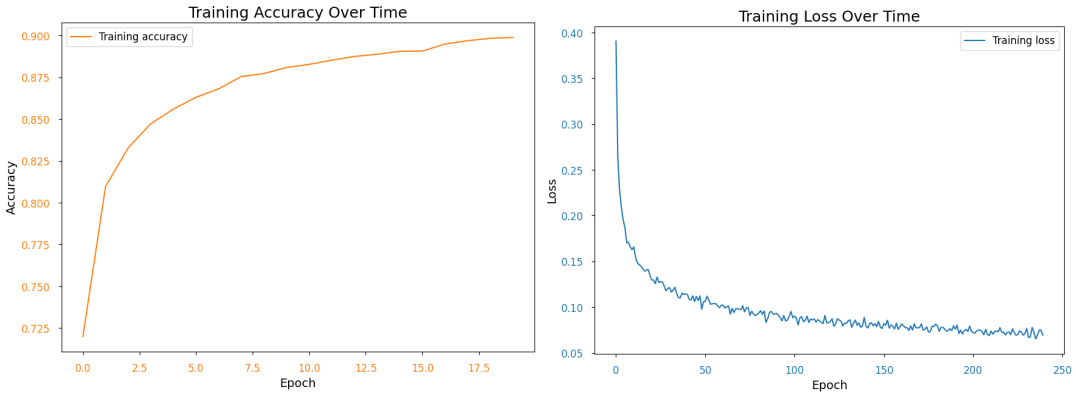


To improve the accuracy score of my model, I tried several changes. The first was increasing the number of layers in my CNN from three to four, but this didn't make much of a difference. Then, I switched to the RMSprop optimiser instead of SGD, but the accuracy level of that was terrible and irregular as seen in the graph below.



Next, I decided to switch back to SGD and add batch normalisation to my CNN to try and improve performance and stability. It supposedly normalises the activations of the neurons in a layer across a mini batch of training data and reduces the internal covariate shift within the network. I applied batch normalisation to the inputs and outputs of the first and second convolutional layers and the third and fourth fully connected layers. But even after doing so, my network was still showing an accuracy of just 56%.

After seemingly failing to get a positive result, I decided to switch to the FashionMNIST dataset to see how the results would compare. And after changing the input sizes to match, I ran the model again and immediately, even after only 20 epochs, the network pulled an amazing 88% accuracy.

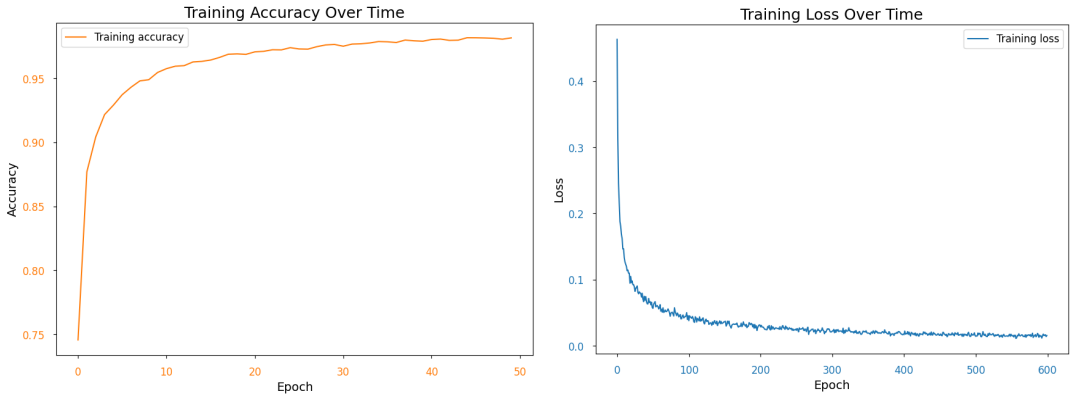


Finally, I tested my neural network against a dataset that was not specified in the coursework, the KMNIST dataset. Which is a collection of 28x28 grayscale images of handwritten Japanese characters consisting of 60,000 training images and 10,000 test images, each labelled with one of the 10 classes representing each of the 10 characters.

I trained the network using the KMNIST dataset, using a dropout rate of 0.5, a learning rate of 0.0066, and a batch size of 8. Running the model with these hyperparameters resulted in a very high accuracy of 96%. The dropout rate of 0.5 may have contributed to the high accuracy by reducing overfitting, as it randomly drops out 50% of the neurons in each training iteration to prevent the network from relying too heavily on any one feature, improving its generalisation.

The learning rate of 0.0066 may have also contributed to the high accuracy, as it determines the step size at which the optimiser adjusts the weights and biases of the network during training. A lower learning rate can lead to slower convergence but can also result in a more accurate model. The batch size of 8 may have also contributed to the high accuracy by allowing the network to see a larger variety of data during each training iteration, which helped to improve generalisation.

Overall, the combination of these hyperparameters may have led to very high accuracy by effectively preventing overfitting and allowing the network to converge on a good solution.



References

The following references are pieces of documentation that assisted in the development of my coursework.

Applying Convolutional Neural Network on MNIST dataset (2021) *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applying-convolutional-neural-network-on-mnist-dataset/ (Accessed: December 28, 2022).*

Build the Neural Network - PyTorch Tutorials 1.13.1 Documentation. *Available at: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html (Accessed: December 29, 2022).*

Data visualization using matplotlib (2022) *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/data-visualization-using-matplotlib/ (Accessed: December 27, 2022).*

Dijsktra's algorithm (2022) *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ (Accessed: December 28, 2022).*

Navone, E.C. (2022) Dijkstra's shortest path algorithm - a detailed and visual introduction, *freeCodeCamp.org*. freeCodeCamp.org. *Available at: https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/ (Accessed: December 28, 2022).*

Pykes, K. (2022) Pytorch tutorial: Building a simple neural network from scratch, DataCamp. *DataCamp. Available at: https://www.datacamp.com/tutorial/pytorch-tutorial-building-a-simple-neural-network-from-scratch (Accessed: December 29, 2022).*