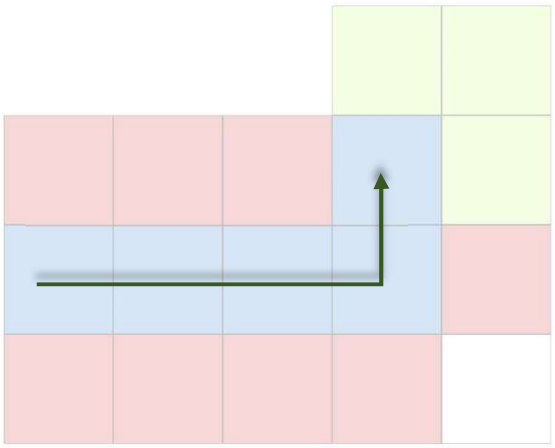
Task 1

a- The algorithm find the shortest bath by the following technique:
A path is created in the start square where it starts to seed to all the adjucent sequare and those paths are saved in a list this list is referred to as Seeding path list "*seedingpath*".
Note : the terms" path is seeding" will be used to refer to a status of finding the adjucent square and seeded paths will be created if the square was not visited before
Each path gets a timing value where it is saved in variable inside the path object
When the main method is called the method start seeding by entering through a loop where it starts "ticking" each tick takes down one from the timing value in the *seedingpath* those paths that reach the value zero are allowed to move on and create their own seeding path and then added to the seedingpath list. It is also crucial to know that once a path reaches a new square, it saves the time estimated to reach this square in a parallel matrix with same size to the original grid.
In case one of the path ends up in closed loop (which is all considered prior to any path seeding) or reached a square that has been visited the path terminate and stored in a different list called terminated path. Also if the path has no other way then an already visited cell by any path it terminates.
A path is not allowed to seed to pass through a cell twice neither it is allowed to pass to any adjucent cell to the cells it passes through it. This is to avoid a path from wasting resources on a path that it could have gone to in a shorter route the illustration below domenstrate this point



If we assume that the path moves along the blue square, then it is not allowed to go to any adjacent location as this would waste of time

After all. I will give a general idea in how this works: A path that has large time is halted while the other who has less time cost moves faster therefore we can ensure that the shortest path to reach the end square is the first path to reach it without the need of checking if there is another shorter path. Thus makes this algorithm fast and always finds the best solution the disadvantage is that the method requires way more resources when the grid size grows as many paths will be seeding.

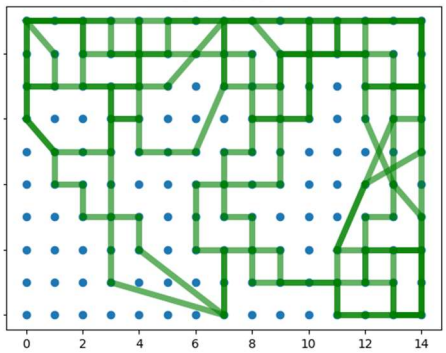
b- Dijkstra algorithm.

This algorithm find the best path by using three parallel list in which the list record the path and the order of each node the algorithm is set to find always the optimal path but yet would consumes more resources then the ant colony and also the to an extent to the algorithm that I have designed as this algorithm might go to any direction and that is where A+ algorithm set to fix

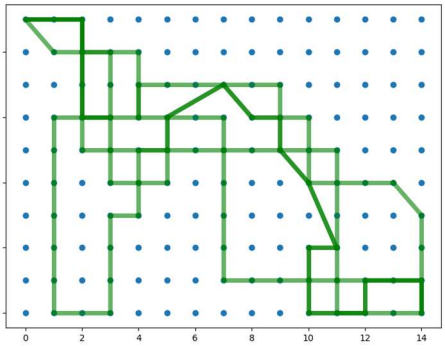
c- Ant Colony algorithm:

By far the ant colony optimization is the least rescourse consuming method between the three above though it is worth to know the for each grid size the same parameter will guarantee a solution as the ant might lose their path and never reach the end point. There is some sort of intuition on how many ants must be in a colony in grid size or the effect of pharmon on the probability and lastly the evaborating. But yet a changing the parameter might result on finding faster solution.

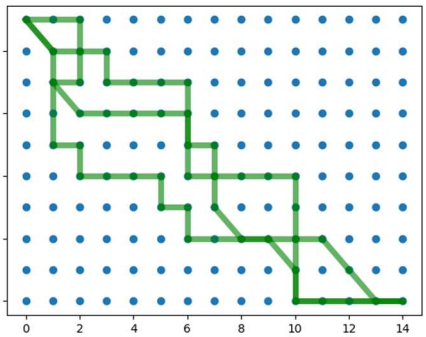
Note: in my implementation I tend to kill those ant who lose their way in a big grid or I added a condtion that does not the ant goes in a endless or extremely long path and I named the feature the ant life span



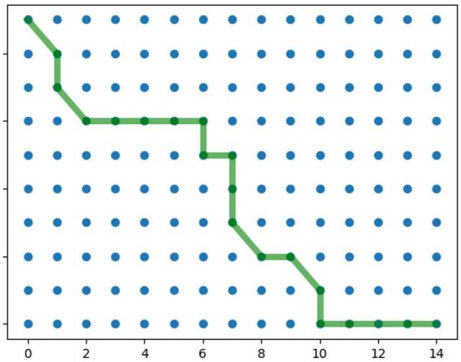
At first ant tend to follow random paths



After few iteration longer path fades as evaboration become obvious



At after many iteration few paths are has lifts



above is the best path which ants found

- The diagnle line is to indicate that the cell has zero value and that is why ant cross it as it cost nothing to cross.

- As a summary it is clear that the ant colony has advantage over both algorithm once the size of the grid grows exponentially and that sets apart. Where both other finds the optimal solution they lack the efficiency once the grid size grows. And with time it manages to find an almost optimal path as the evaporation and works in favor of the probability of choosing the shorter path

Task2:

We will build a neural network that we can change many of its hyper parameter.

- Structure of the neural network which can be given as one dimensional list/ tuple of integer where the first element is the input nodes and the rest represent the nodes in each layer created.
- The second argument is the activation function list which its length is ***number of layers - 1*** lwe can also supply 1 element that indicate the name of the activation function and the program will use this activation function for all layers
- The output function which is set by default to None.
- The stop criterion which will stop epochs if there is no improvement in the accuracy to a certain threshold. The default value is set to .01%
- Cost function a Mean Square Error is the default function and it is fully functioning note: I have not fully developed the others and their result might vary as this is an extra effort
- Key argument in which the default __dict__ will be updated which allows to manipulate other value such as the net work default weights and biases

The neural net work also uses stochastic gradient descent SGD and that in terms allows for a set of different parameter to be changed and they are as follow:

- Training data which is a data and label in tuple format (X , y) Note: the input data must come in an array with one dimension and its length must equal the input nodes stated in the structure list as mention above. Also the the label data must be supplied in One-hot code which means it must be supplied as a zero array with one dimension and a length of the total number of the output classification where there is only one element with 1 value that indicate which class this particular data belong to
- The epochs variable which is an integer number that can be controlled to run the algorithm certain amount of time and keep updating the biases and weight of the network seeking for a better fit
- Mini batches size an integer number to apply the the update using a set of training data rather than one by one
- Learning rate which is a ratio multiplied by the gradient that determine how much movement must be done toward the local minimum. Note the this value needs to be examined to better fit different structure and there is no optimal solution for all structure.
- Test data which again contains the data X and its labeled class y one hot coded

Now let us take a dive on how this neural was built and let us start with the mathematical understanding of the network:

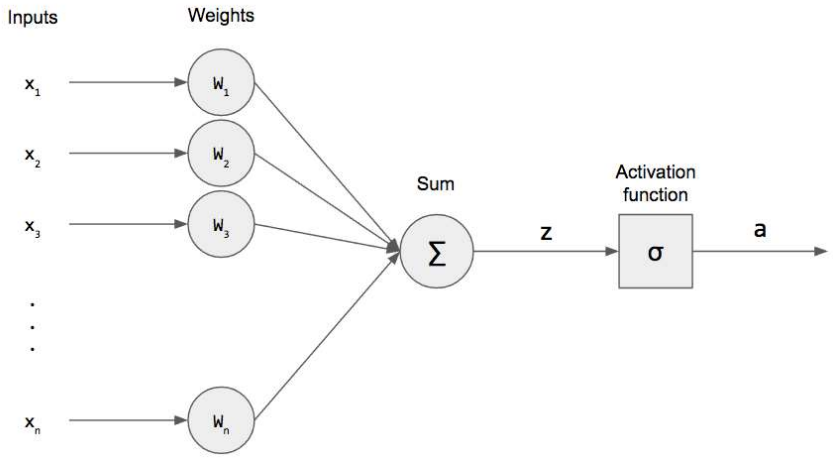
Forward pass:

Each input, in our case it is an image with a size (28,28), will be flatten first through a method named flatten_normalize_matrix in the jupyter book. The method takes an n dimensional array and spits out a flat version of it with a single dimension

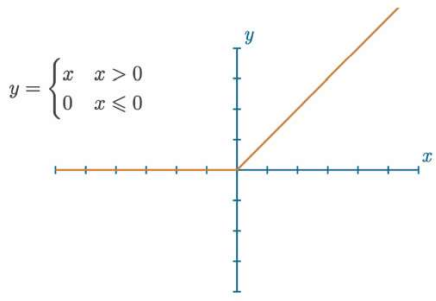
Then in term the data are fed through and multiplied with their associated weights then a bias is added and thus in turn will be z(L) which is the sigma of

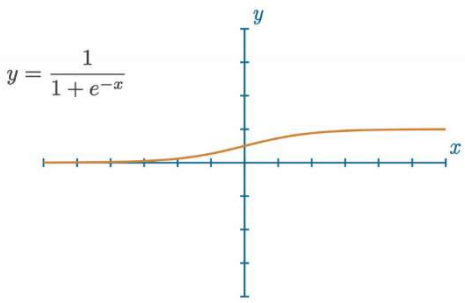
$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

Then for each it will pass through the activation function which can be one of many kinds we will mention the most common two which they are the sigmoid and the ReLU activation function.



The sigmoid function below to the right takes the summation above and place it as value in x :

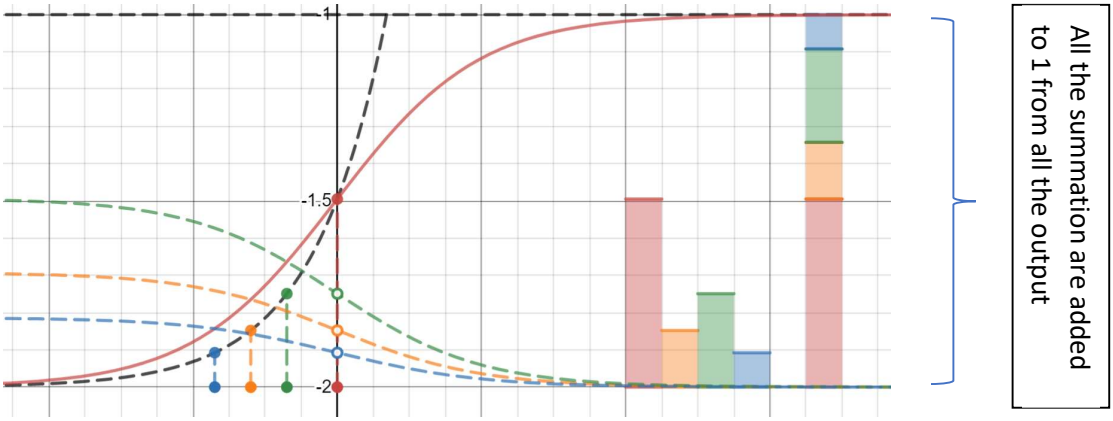




And the figure above to the left shows the ReLU function. This process is pass through all the layers (input layer - hidden layers – output layer) and that is what we call forword pass. It is worth to mention that in classification neural network like the one we are studying the the output layer usually uses softmax function which a function that shows the probabilty of each class. Thus all the output summation will corrsopund to 1.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

And below is a beautiful digram that shows softmax. The link here allows for an interactive version.



In this arbitrary example each color represesnt the output of one neuron

Backward propagation and loss function:

In order to really get significant information from we need to train the network and this happens through the following steps:

- Loss function : In simple term it is a way to calculate how wrong the network is or how bad it is. and this can be applied mathematically via many different ways we will mention here one of the most used one which is the Mean Squared Error aka MSE.

$$L_i = \frac{1}{J}\sum_j (y_{i,j} - \hat{y}_{i,j})^2$$

The y i j means the target value or the actual value. While y hat is the network output.

- Backpropagation: and this is the real mechanism behind training the network and the way this achieved gets a bit more complicated. And the reason behind this is, in order to train the network we need to find a way to estimate the effect of each weight in the network and adjust their value in way that minimizes the loss/error and here where calculus comes to rescue. But first let us describe the whole picture. For example the network that we are using in this coursework has 784 input node. Now let us consider a structure where actually this network perform very well which is 784 , 22 , 16 , 10 so in this way there are more than 17,700 weights to readjust.
    - *784 * 22 + 22 * 16 + 16 * 10 = 17,744 weights*

  And as complicated as it sound actually implementing those change are rather easier than it seems by following a systematic approach. This approach is what back propagation s all about is to find the griedient slop of the error equation and move with step toward minimizing it and now let us take a look at the equation. Let see first the output layer and and how a change of its any weight would change the out come and which in term changes the error.

$$new\ weight\ (x)\ = old\ weight\ (x)\ -\ Learning\ rate * \left(\frac{\partial(error)}{\partial\ (weight\ x)}\right)$$

Let us first find the derviteve of the errer with respect to an arbitrary weight connected to that node we will use the chain rule which say to find the dervitive of a function to a variable that is not connected directly to that function we need to multiply all the sub function dervetives that connect us to the intendent variable.

$$\left(\frac{\partial(error)}{\partial\ (weight\ x)}\right) = \frac{dC}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}}$$

dc/da is the dervitive of the cost function and the second term is the derivative of the output function to its input which it is in our case it can be derivative of ReLU of the sigmoid function and lastly is derivative of the sigma operation to the given weight which will be according to the above equation the output of the pervious layer connected to this weight: giving that all other output are constant with respect to this weight and therefore there derivative equals zero. Let us take a look on how the equation would look like.

$$\varphi(z) = \frac{1}{1 + e^{-z}} \qquad \frac{d\varphi(z)}{dz} = \varphi(z)(1 - \varphi(z))$$

And for ReLU function the derivative is simply:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

For biases the equation above is similar with the only difference that the last term equals 1

And now let us yet take a deeper dive and look on how change the weight in the pervious layer and actually and this is achieved by the estimating the error gradient at the node in the previous layer and just as the last equation term equals ***w*** and this basically how we propagate backward and we repeat the same thing for the previous weights and biases.

- Mini batched and average value: rather than doing the above samples by samples we tend to perform the training using minibatches instead and applying the average on the value and we can think of intuitively as faster but less accurate toward a local or global minimum than precise yet slow descending

And that pretty much the general idea behind neural network and from there we can consider yet more complicated structure and function

As much as easy it might sound nueral network proves to be a great and easy solution to easy yet hard to program problem such as the MNIST Handwriting set that I will discuss below

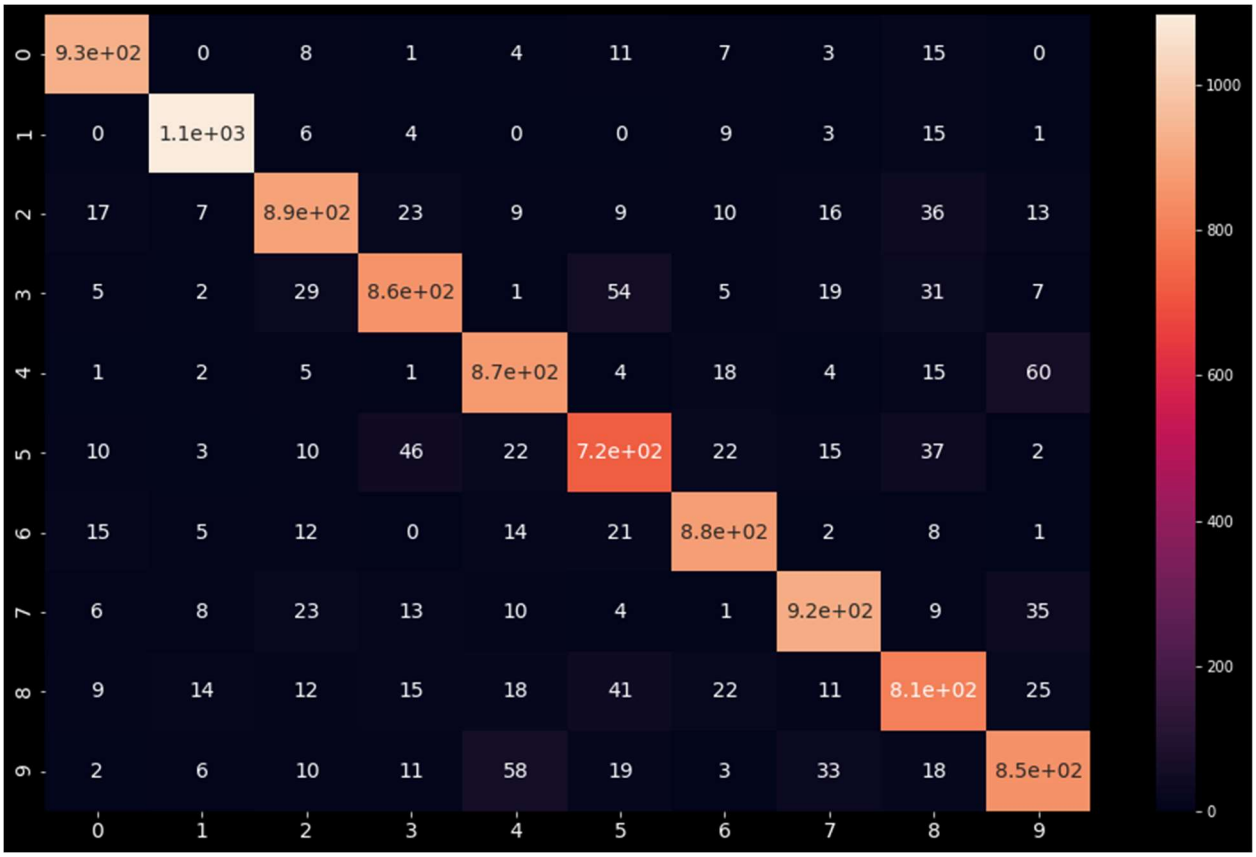
Task 3:

Comparing different neural network structure:

It is abvious that the deeper the network “the more hidden layer” is the better result it can obtain but this is not the case as deepening the network seems not as effective after certain amount of nodes and layer and it seems that it carried out so little improvement.

Secondly. The accuracy improves dramatically when applying CNN convolution neural network as filtering seems to improve the accuracy even when using less neurons than the one used in ANN as ANN reach it is maximum accuracy point

Confusion Matrix:



We can see here while the accuracy was high the recall and precision was not as high as accuracy was in all classes and in few cases they were higher: let us build basic example on what is recall and precision

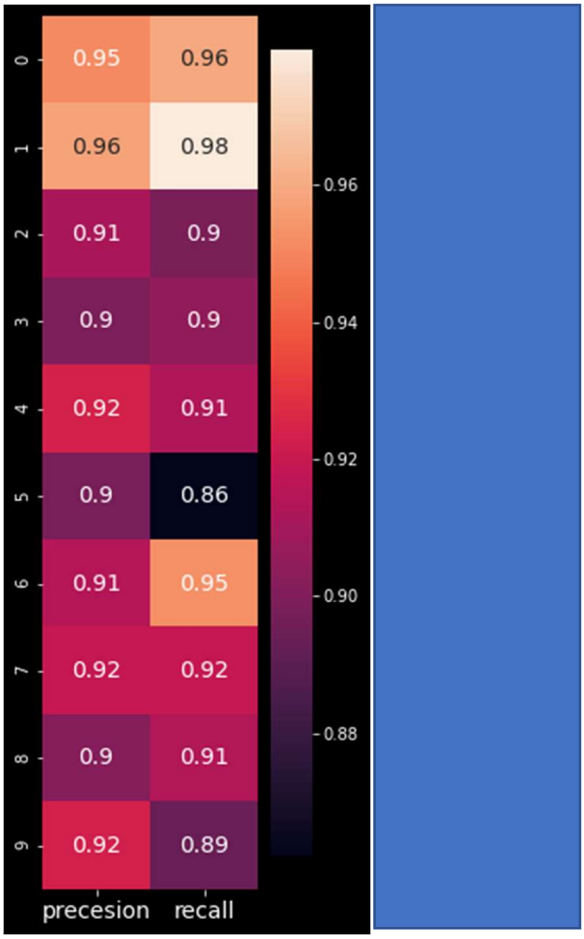
Precision — Out of all the examples that predicted as certain class, how many really belong to this class

$$\boldsymbol{precision} = \frac{tp}{tp + fp}$$

Recall — Out of all the one class examples, how many are predicted

$$\boldsymbol{recall} = \frac{tp}{tp + fn}$$

Thus, we see number 5 has the very low precision giving many value who predicted to 5 are actually not. Let us print the precision for each class.



We can see the image contains number 5 scored in general the lowest both in precision and sensitivity “Recall” and the somehow understandable as the number has many common feature with the other numbers. In contrary, number 1 scored really for the opposite reason. Though why number 2 scores higher than one is some how unclear for me

Task 4 :

Inspired by 3brown1blue YouTube channel as it was referenced during our classes I build using python a library to create a grid in a 3D space then to have a 3 * 3 Matrix that can manipulate the space and visual it is effect. And how it would squash space to a lower dimension or flip it the 3brown1blue matrices videos was showing the effect on 2 by 2 space and it was really interesting to see the similar effect on a 3D space. All you have to input in the program the 3D matrix and watch the effect of it also the program will show the base vectors and eigen base vector the determent and more