

Mathematics and Programming for AI Coursework (Individual only)

Andrew Youssef |Programming and Math in AI | 18/12/2020

# Task 1

I have decided to compare the algorithms below using the value of the cell as the time spent in the cell. Both algorithms go either horizontally or vertically but not diagonally. Both algorithms do not loop backwards.

My Algorithm:

The algorithm I chose at the beginning before I read about Dijkstra's algorithm was very similar so I tried to implement a simpler algorithm which should be better than chance most of the time.

My algorithm walks along the side of the matrix checking that it does not go beyond the edge and when it arrives at the edge it moves in the opposite direction until it reaches its destination. The algorithm decides if it is going to go across then down or down then across via a random number using numpy.random method that gives either zero or one. A random algorithm might on occasions take the diagonal staggered route and might be faster if the time in the cell is small.

I chose to implement this algorithm as it will work with any matrix size and on average will give good results.

Dijkstra's Algorithm:

Dijkstra's algorithm tries to look at the shortest route between cells. Since the number in the cell represents the time and our aim is to reduce the amount of total time from starting in cell (0,0) to the destination cell. The algorithm first checks the three adjacent cells (above, front and below) to choose the cell which has the lowest value and then moves towards it while checking that it is not going beyond the edge of the matrix until it arrives at the final destination.

Ant Colony Optimisation (ACO:

Unfortunately, I was not able to get a working algorithm for the Ant Colony in time for the submission.

The ACO replicates how ants forage for food and use pheromones to mark their route. A shorter route will mean that the ant will be able to come and go more than the ant on the longer route which means more pheromones. To implement the algorithm, I was trying to get the ant to randomly choose its route through the cells and then computationally mark the route with higher pheromone if it is shorter. This algorithm would have probably predicted the shortest route but would have taken more computational time as it will be randomly choosing the routes at the beginning

Comparison:

I have run several iterations of both algorithms to compare for the shortest time (lower value in cells visited) and for speed of execution of the code. This was done on a 2000 by 2000 cell matrix. See results in the table below.

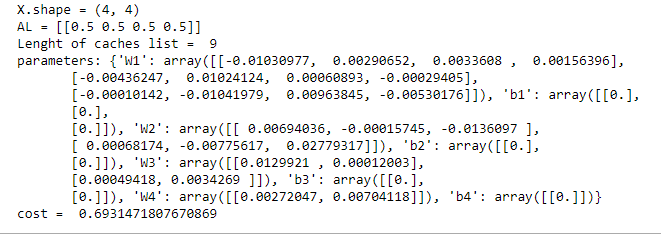
|  |  |  |  |
| --- | --- | --- | --- |
| My Algorithm walk through time | My Algorithm computation time | Dijkstra’s algorithm walk through time | Dijkstra’s algorithm computation time |
| 15,770 | 0.598 | 10,300 | 02.378 |
| 16,003 | 0.627 | 10,859 | 02.186 |
| 16,138 | 0.624 | 10,573 | 02.260 |
| 16,158 | 0.616 | 10,777 | 02.317 |
| 15,855 | 0.623 | 10,809 | 02.204 |
| 16,231 | 0.601 | 10,743 | 02.238 |
| 15,929 | 0.609 | 10,555 | 02.281 |
| 15897 | 0.602 | 10,566 | 02.216 |
| 16,374 | 0.607 | 10,704 | 02.247 |
| 15,640 | 0.603 | 10,340 | 02.242 |

From the table above it is clear that Dijkstra’s algorithm performs better than my simpler algorithm with quite a significant margin. On the other hand, my algorithm was executing much faster. This highlights the importance to strike a balance between speed of execution and efficiency of results. In real life examples (such as delivery route optimisation), the time to travel from one cell to another until destination may be the determining factor due to the cost involved compared to the cost of longer computational time.

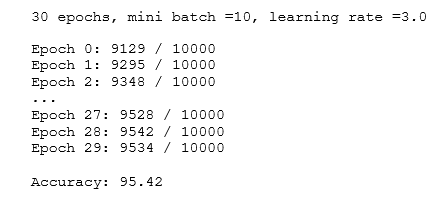
# Task 2

Coding Experience summary- (Code block 1 50% copied, code block 2 90% copied)

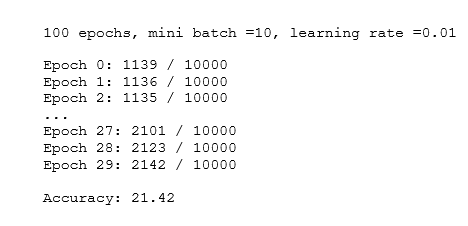
Could not implement fully, due to inability to get the MNIST data to run the code. However, the first block of code executes when given random x and y shapes. The first block utilizes a 2-layer neural network with forward and backwards pass as well as backpropagation. The layers are broken up into one relu and one sigmoid layer. From random implementation without mnist, we achieve:



The second block of code is changed to suit a fully parameterized style for number of layers and types, it is a different style of neural networks than the first implementation as it adds stochastic gradient descent as an optimizer where in the original, I only have gradient descent. I also added softmax output layer. The only limitation I found was that due to the for loops I was only able to implement either sigmoid or relu layers, but not both.

When testing with the MNIST dataset we are expected to reach high levels of accuracy due to the use of backpropagation. While each test can vary due to the use of randomized weights and biases, we find in general the program reaches 95% accuracy with the more epochs we do.

In this test we set a medium learning rate with only 30 epochs however we achieve a high accuracy.



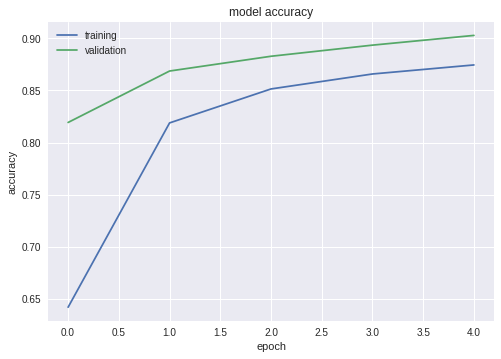
When we change the parameters of the test to a low learning rate but high epochs number, we find that even though we are increasing the learning rate is too low causing it to not increase accuracy fast enough.

This would also be true if we had a high learning rate as we would find the accuracy decreases over time as it would be the same as teaching at random.

Summary and comparison of what I was expecting to find:

**Sigmoid with softmax output**

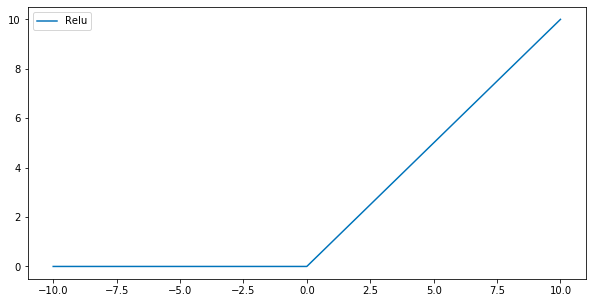
With testing of only having sigmoid and softmax layers, we would find that the model is expected to be between the 87% to 90% accuracy range. While this is much lower than we would hope the model to be it is much better than randomly guessing. The graph below is what I would expect when testing my code.



As our implementation would be looking at fully parametrizable layers, we can also gauge how well the model would do as more hidden layers are added to testing. We find that at a certain threshold the accuracy of the data reduces as more layers are introduced, we take this to be overfitting.

As we introduce backpropagation and optimizers in this case stochastic gradient decent, we find the opposite to overfitting begins and as we increase nodes and epochs the accuracy can increase 95%. However, this increase in accuracy is at the expense of time due to the high number of nodes needed to achieve this, this is why sigmoid is not popular for neural networks.

**Relu with softmax output**

As we test this we would find that the overall convergence and increasing testing accuracy would be reached quicker, this comes in part to relu being computationally efficient to sigmoid, however we would find that sometimes the relu would output very low accuracy, this is due to dying relu,problem where too many activations become zero and stay zero prohibiting learning instead of increasing accuracy.

As we start to add fully parameterized layers with softmax output layer as well as using cross entropy loss function, we find that as the hidden layers increase, we can start achieving 98% accuracy in a lower number of nodes and epochs than the sigmoid implementation.

**Improvements to get better convergence**

For future, I believe the use of adam over stochastic gradient descent as an optimizer would lead to a better convergence. Trying to use different architectures such as tanh, softplus and leaky relu would also be useful in testing accuracy. For example, the use of leaky relu over relu is more beneficial as it eliminates the dead neuron problem linked with its equation.

# Task 3

Coding Experience summary- (1st block of code 50%, 2nd block of code 100%)

For task 3 I attempted to implement NN within Pytorch, I ran into a lot of trouble while trying to find which libraries to use and how to implement it. Mostly I was struggling to find a MNIST downloader that works with what I wanted to make. I was able to find a next journal article with the code on how to implement successfully into pytorch (Second piece of code in task 3) however it was abit more complicated as it had Conventional layer, Relu, with dropout however it lacked parameterizable layers relying only on 2 layers and a logarithmic SoftMax.

I took inspiration from these and created my own pytorch (first block of code) and attempted to use Keras as my MNIST loader, I made a relu sigmoid and softmax Neural network, using Lab 7 as another inspiration, I made my code simple only using 2 hidden layers one relu and another sigmoid with a softmax output layer. I ran into an error “mat1 and mat2 shapes cannot be multiplied” and was unable to solve it. I deciphered it is due to the Keras implementation and x\_train variable.

Summary and comparison of what I was expecting to find:

**Computational speed**

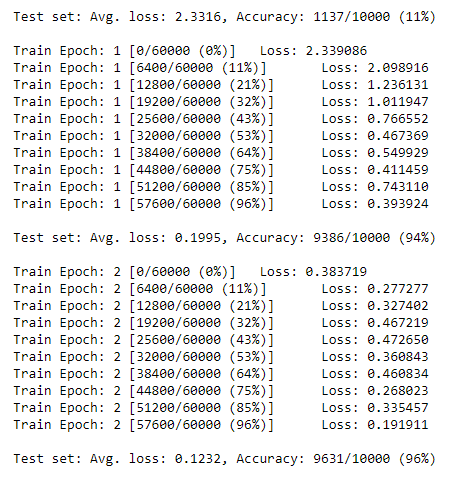
One main advantage pytorch has over numpy is array operations and traversing, this helps in testing and allows for a higher computational speed within pytorch. This means that we can see that pytorch tests finish faster compared to numpy.

The ability to also use your GPU in pytorch is a massive improvement to numpy as it allows for load management, not allowing the cpu to overwork and both to work in tandem.

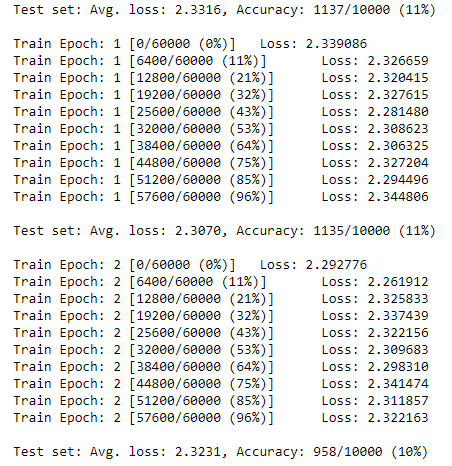
Another advantage to why pytorch is faster is the ability to use GPU’s, due to their higher memory bandwidth and rendering ability they work in tandem with the CPU where the CPU unloads harder tasks such as vector computations.

**Improvements and Updates**

My second program within task 3, is what I would want to strive for in improving my original architecture, the use of multiple layers with both dropout and convolutional neural networks allows for a very high accuracy with very low number of epochs. The results as we see below are for the MNIST set with only 2 layers of relu and log\_softmax.



As we can see from the training and test sets that after the initial random test being 11% only just one epoch, we reach 94% accuracy. We achieved this with a learning rate 0.01, however when we increase this rate to above one, we have an extreme case of dead relu where the accuracy decreases to 11%.



Other improvements I would like to add would be the use of LTSM, long and short term memory networks, their use of gates to store previous values in layers.

References

## websites used

Task 1

Task 1 - Nothing copied and pasted Resources used as a guide

My Algorithm

<https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.randint.html> <https://www.askpython.com/python/examples/add-a-newline-character-in-python> <https://numpy.org/doc/stable/user/basics.indexing.html> <https://www.w3schools.com/python/python_while_loops.asp> <https://numpy.org/doc/stable/reference/generated/numpy.array.html> <https://stackoverflow.com/questions/43901096/numpy-indexing-getting-a-specific-row-for-each-column> <https://www.w3schools.com/python/numpy_array_indexing.asp> <https://www.datacamp.com/community/tutorials/python-numpy-tutorial#make> <https://machinelearningmastery.com/numpy-axis-for-rows-and-columns/> <https://www.w3schools.com/python/python_while_loops.asp> <https://stackoverflow.com/questions/19522990/python-catch-exception-and-continue-try-block#:~:text='continue'%20is%20allowed%20within%20an,loop%20to%20call%20your%20function>. <https://www.w3schools.com/python/python_try_except.asp> <https://pythonbasics.org/multiple-return/> <https://note.nkmk.me/en/python-multi-variables-values/> <https://www.tutorialspoint.com/What-is-difference-between-self-and-init-methods-in-python-Class#:~:text=__init__%20method,the%20attributes%20of%20the%20class>. <https://cs231n.github.io/python-numpy-tutorial/>

Dijkstra algorithm

<https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/> <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/> <https://stackoverflow.com/questions/22897209/dijkstras-algorithm-in-python> <https://www.educative.io/edpresso/how-to-implement-dijkstras-algorithm-in-python> <https://www.geeksforgeeks.org/building-an-undirected-graph-and-finding-shortest-path-using-dictionaries-in-python/> <https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/> <https://pymotw.com/2/collections/deque.html> <https://www.geeksforgeeks.org/deque-in-python/> <https://numpy.org/doc/stable/reference/generated/numpy.ones.html>

Ant colony

<https://github.com/pjmattingly/ant-colony-optimization> <https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms#Example_pseudo-code_and_formula> <https://towardsdatascience.com/using-ant-colony-and-genetic-evolution-to-optimize-ride-sharing-trip-duration-56194215923f> <https://github.com/Akavall/AntColonyOptimization> <https://github.com/Vampboy/Ant-Colony-Optimization/blob/master/AntColonycode_forjupiternotebook.ipynb> <https://numpy.org/doc/stable/reference/generated/numpy.array.html>

Task 2

<http://neuralnetworksanddeeplearning.com/chap1.html#implementing_our_network_to_classify_digits> Task 2 70% code block 2

<https://pylessons.com/Deep-neural-networks-part3/> Task 2 40% code block 1

<https://mlfromscratch.com/neural-network-tutorial/#/> Task 2 10%

<https://colab.research.google.com/drive/1Xsv6KtSwG5wD9oErEZerd2DZao8wiC6h> Used to get an idea of doing SGD

<https://stackoverflow.com/questions/32109319/how-to-implement-the-relu-function-in-numpy> Task 2 ReLU code

<https://datascience.stackexchange.com/questions/11704/reshaping-of-data-for-deep-learning-using-keras>

https://github.com/chokkan/deeplearningclass/blob/master/mnist.ipynb

<https://medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3> Used for analysis

Task 3

<https://www.kaggle.com/riteshsinha/pytorch-building-a-basic-neural-network-mnist>

<https://nextjournal.com/gkoehler/pytorch-mnist> 2nd block of code 100% copy

<https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627>

<https://www.pluralsight.com/guides/building-your-first-pytorch-solution>

https://github.com/chokkan/deeplearningclass/blob/master/mnist.ipynb