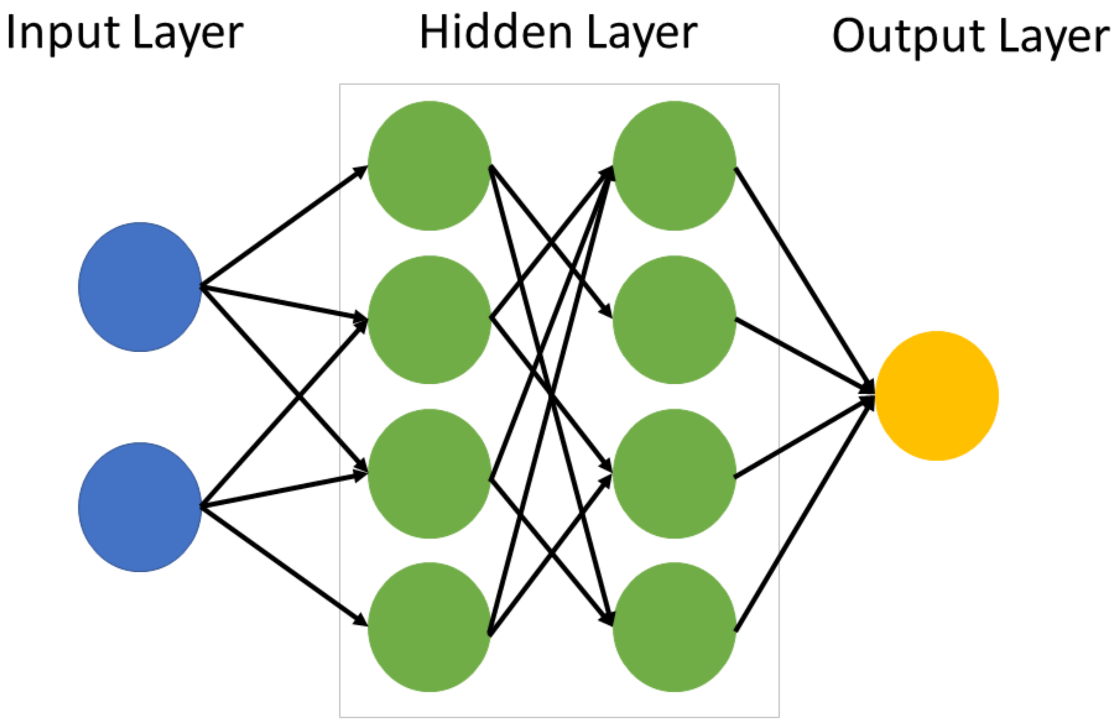
Advanced Data Analytics Algorithms, Machine Learning

Assessment 2

Convolutional Neural Network



Artificial Neural Networks

- **Daniel Braithwaite**

Introduction

For this task, I have decided to create and implement a convolutional neural network from scratch. This task is based off the MNIST dataset, which is a set of 60,000 test images of 28x28 anti-aliased greyscale hand-drawn numbers. Originally, I had intended to create a machine learning implementation to detect digits from a number plate, however, after doing some research into the inner workings of such a network, I have decided that it is beyond the scope of this subject.

MNIST Dataset

The Modified National Institutes of Standard and Technologies (MNIST) dataset is a set of 60,000 training and 10,000 test images, normalised to 28x28 anti-aliased pixels, each representing a digit from 0-9. The format for these images is in a csv file, with 785 values per image. The first value represents the label (0-9), with the other 784 values representing each pixel value (0-255).

Research

My research on this topic had led to implementations of this with a variety of libraries, or implementations from scratch based on 1 hidden layer with biases. Additionally, there was another such implementation with 2 hidden layers, without any biases. As a result, I had decided to implement a learning model which has 2 hidden layers **with** biases, such that it is sufficiently different from other implementations on the same problem.

Key concepts

They key concepts that must be researched within this project are:

- Weighting
- Activation functions
- Linear regression
- Gradient descent optimisation
- Bias values

What is a convolutional neural network

According to ChatGPT, a large language model, “Convolutional Neural Networks (CNNs) are a class of deep learning models designed for processing and analyzing visual data, such as images and video. They have revolutionized tasks like image classification, object detection, and image segmentation. CNNs are inspired by the human visual system and are particularly well-suited for tasks that involve detecting patterns and features in images.”.

The main components in the model I have built are as follows:

1. Initialisation
2. Sigmoid function
3. Softmax function
4. Forward propagation
5. Backward propagation
6. Accuracy computation
7. Prediction

Network architecture

The network architecture for this CNN is as follows:

1. Input Layer:
    - INPUTSIZE: 784
    - The input layer consists of 784 neurons, corresponding to the 28x28 pixels in an image from the MNIST dataset.
2. Hidden Layers:
    - HIDDEN1: 128 neurons
    - HIDDEN2: 64 neurons
    - These two hidden layers are fully connected (dense) and use the sigmoid activation function. The number of neurons in each layer is a hyperparameter that can be adjusted based on the problem and dataset.

3. Output Layer:

   - OUTPUTSIZE: 10 neurons
   - The output layer has 10 neurons, corresponding to the 10 possible digits (0 to 9). It uses the softmax activation function to produce a probability distribution over the 10 classes. The network predicts the class with the highest probability as the final classification.

Mathematical theory

In this section I will be discussing the theory and calculations behind each part of the CNN

Forward propagation:

Forward propagation involves the sequential processing of data through the network's layers, with each layer's neurons performing weighted computations and applying activation functions until the final prediction or output is obtained.

The first activation function, $A^{[0]}$, is simply the input layer. This is a matrix of size $784 \times n$, as there are 784 inputs for n sets.

From here, the pre-activation value $Z^{[1]}$ of size $128 \times n$ is calculated as the dot product between $W^{[1]}$(128×784) and $A^{[0]}$($784 \times n$). Additionally, a bias value $B^{[1]}$ (128×1) is added to each of the 128 nodes in $Z^{[1]}$ to add more flexibility to the model. This equation looks like:

$$Z^{[1]} = W^{[1]} \bullet A^{[0]} + B^{[1]}$$

In order to create a CNN, and not just a complicated linear regression model, an activation function is applied to the pre-activation value. This is done to each of the layers so that it is not simply a linear combination of the layers before it. This is calculated as:

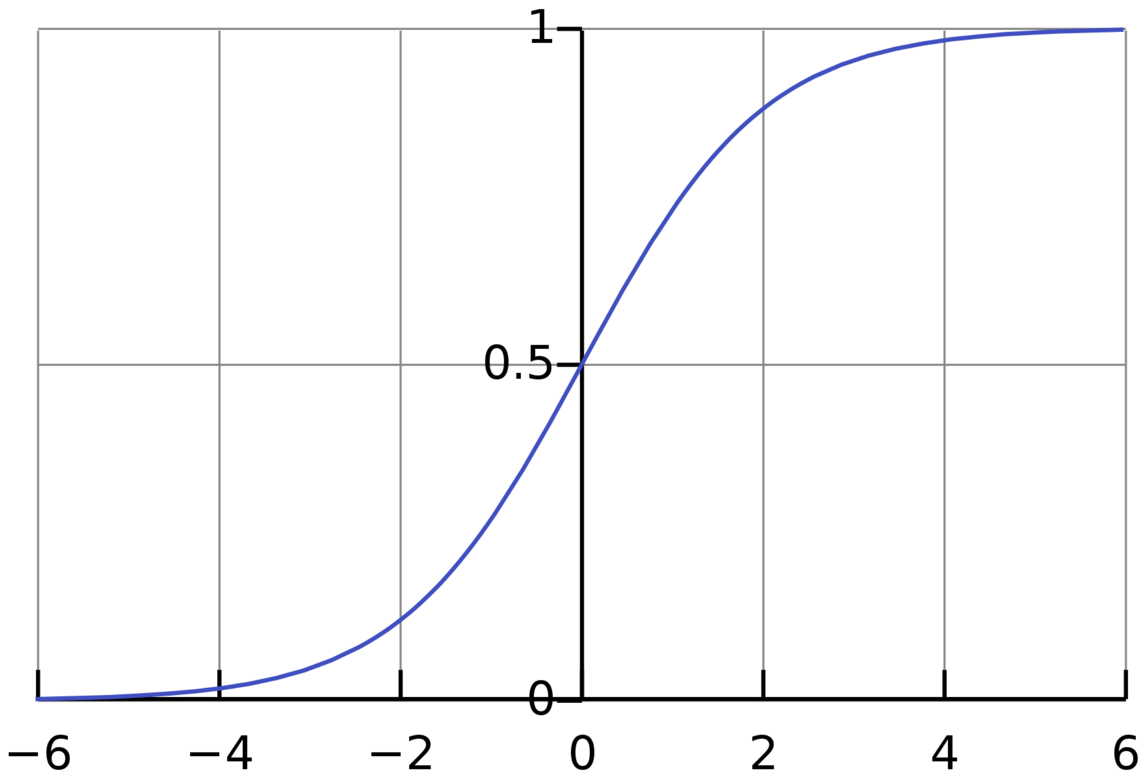
$$A^{[1]} = \sigma(Z^{[1]})$$

where $A^{[1]}(128 \times n)$, meaning that the sigmoid function is applied to every value of $A^{[1]}$.

The sigmoid function σ is defined as:

$$f(x) = \frac{1}{(1+e^{-x})}$$

The sigmoid function compresses the pre-activation value to a value between 0 and 1, mapped in such a way:



Similarly, this is done for the second hidden layer. The pre-activation value $Z^{[2]}(64 \times n)$ is again calculated through the dot product, this time of $W^{[2]}(64 \times 128)$ and $A^{[1]}(128 \times n)$, with $A^{[1]}$ being the previous layer's activation function. Once again, the bias value $B^{[2]}(64 \times 1)$ is added, resulting in the equation:

$$Z^{[2]} = W^{[2]} \bullet A^{[1]} + B^{[2]}$$

Once again, the sigmoid function is applied to $Z^{[2]}$ to compress the value to between 0 and 1 in a non-linear way:

$$A^{[2]} \ = \ \sigma(Z^{[2]})$$

Finally, to go from the second hidden layer to the output layer, the pre-activation value $Z^{[3]}$($10 \times n$) is once again calculated by the dot product of $W^{[3]}$(10×64) and $A^{[2]}$($64 \times n$) with $A^{[2]}$ being the previous layer's activation function. A third bias value $B^{[3]}$(10×1) is once again added, resulting in this equation:

$$Z^{[3]} = W^{[3]} \bullet A^{[2]} + B^{[3]}$$

However, as this is the activation function for the output layer, the softmax function (σ) will be used instead of the sigmoid function. The softmax function transforms the raw, unnormalised scores into a probability distribution over multiple classes (in this case 0-9). As such, $A^{[3]}$ becomes:

$$A^{[3]} \ = \ \sigma(Z^{[3]})$$

Where σ is $g(x)$:

$$f(x) \ = \ e^{x - max(x)}$$

Here, $max(x)$ represents the largest value of x, i.e. the highest probability of x

$$g(x) \ = \ \frac{f(x)}{\Sigma f(x)}$$

Thus, $A^{[3]}$ will return a matrix of size 10×1, with the elements of the matrix summing to 1. The element inside the matrix with the highest value (probability) is taken to be the output of the neural network.

Backward propagation:

Backward propagation involves propagating gradients backward through the network to update the model's parameters. The process continues iteratively, gradually improving the model's ability to make accurate predictions by reducing the loss.

This model uses categorical cross entropy as its loss function. This is incorporated within the backwards propagation phase and is not explicitly calculated or stored. The adjustment of the weights within the third layer is done with the following formula:

$$error^{[3]} = 2 \times \frac{(pred - true)}{n} \times g'(Z^{[3]})$$

Where $pred$ is the prediction of the neural network, $true$ is the ground truth, n is the number of samples and $g'(Z^{[3]})$ is the derivative of the softmax function, mentioned previously. The 2 is there to scale the output by a factor of 2, making the adjustment of the network slightly faster. The derivative $g'(Z^{[3]})$ is calculated as follows:

$$g(x) = \frac{f(x)}{\Sigma f(x)} \times \frac{1 - f(x)}{\Sigma f(x)}$$

With $f(x)$ from the forwards propagation section.

From there, the difference weight $W^{[3]}$ is to be adjusted is calculated using outer matrix multiplication with respect to the error rate. This is done by calculating the gradient of the loss with respect to the weights, and creates a matrix with the direction and magnitude of the adjustment needed to reduce the loss.

$$dW^{[3]} = error^{[3]} \otimes A^{[2]}$$

$A^{[2]}$ is used here to calculate how the loss should influence the weights of the output layer.

From here, the error of the second layer can be calculated in a similar manner, using the derivative of the sigmoid function as opposed to the softmax function. This is done as the second and first hidden layers use the sigmoid function, and the derivative of the activation function is required in order to effectively calculate loss.

$$error^{[2]} = W^{[3]T} \bullet error^{[3]} \times \sigma'(Z^{[2]})$$

Which is then used to calculate how the weights should be adjusted:

$$dW^{[2]} = error^{[2]} \otimes A^{[1]}$$

Here, the weight matrix $W^{[3]}$ is transposed to match the dimensions so that the dot product of the previous error or gradient of loss can occur. This is done to create a new error signal that represents how much each neuron in this hidden layer contributed to the error in the output layer.

This is once again done for hidden layer 1:

$$error^{[1]} = W^{[2]T} \bullet error^{[2]} \times \sigma'(Z^{[1]})$$

And once again used to calculate the adjustment vector:

$$dW^{[1]} = error^{[1]} \otimes A^{[0]}$$

Additionally, the error contribution of the biases are calculated during backwards propagation. As the bias rate affects all neurons in a given layer equally, the gradients are calculated by accumulating the error signals of the neuron they are connected to. These are calculated in the following ways:

$$bias^{[3]} = \Sigma(error^{[3]})$$

$$bias^{[2]} = \Sigma(error^{[2]})$$

$$bias^{[1]} = \Sigma(error^{[1]})$$

And thus, gradient descent optimisation using categorical cross entropy is implemented.

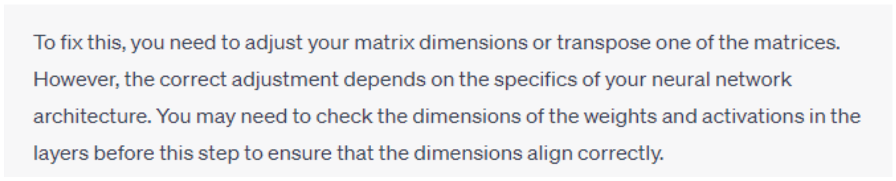
Accuracy computation:

The accuracy of the neural network is calculated by averaging out the comparison between the predicted outputs and the ground truth over the separate development split. This ensures that the model is being tested against unseen data, and is thus able to be applied in a general case.

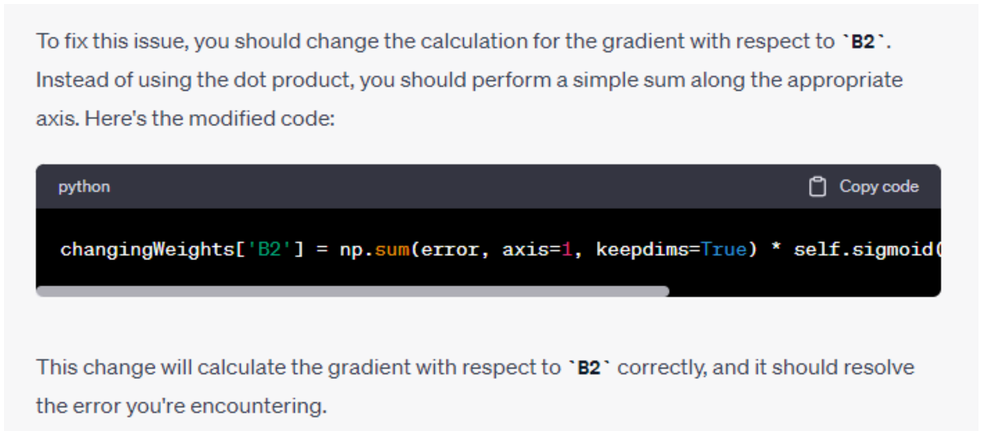
Use of generative AI:

Generative AI (ChatGPT) was used during this task to both explain/clarify concepts and to assist with error correction within the programming phase. More specifically, it was used to identify and help solve issues regarding matrix sizes when back propagating. This, in addition with liberal use of the $print(array.shape)$ function has allowed me to create a functioning program.

Attached are some of the conversations I had:



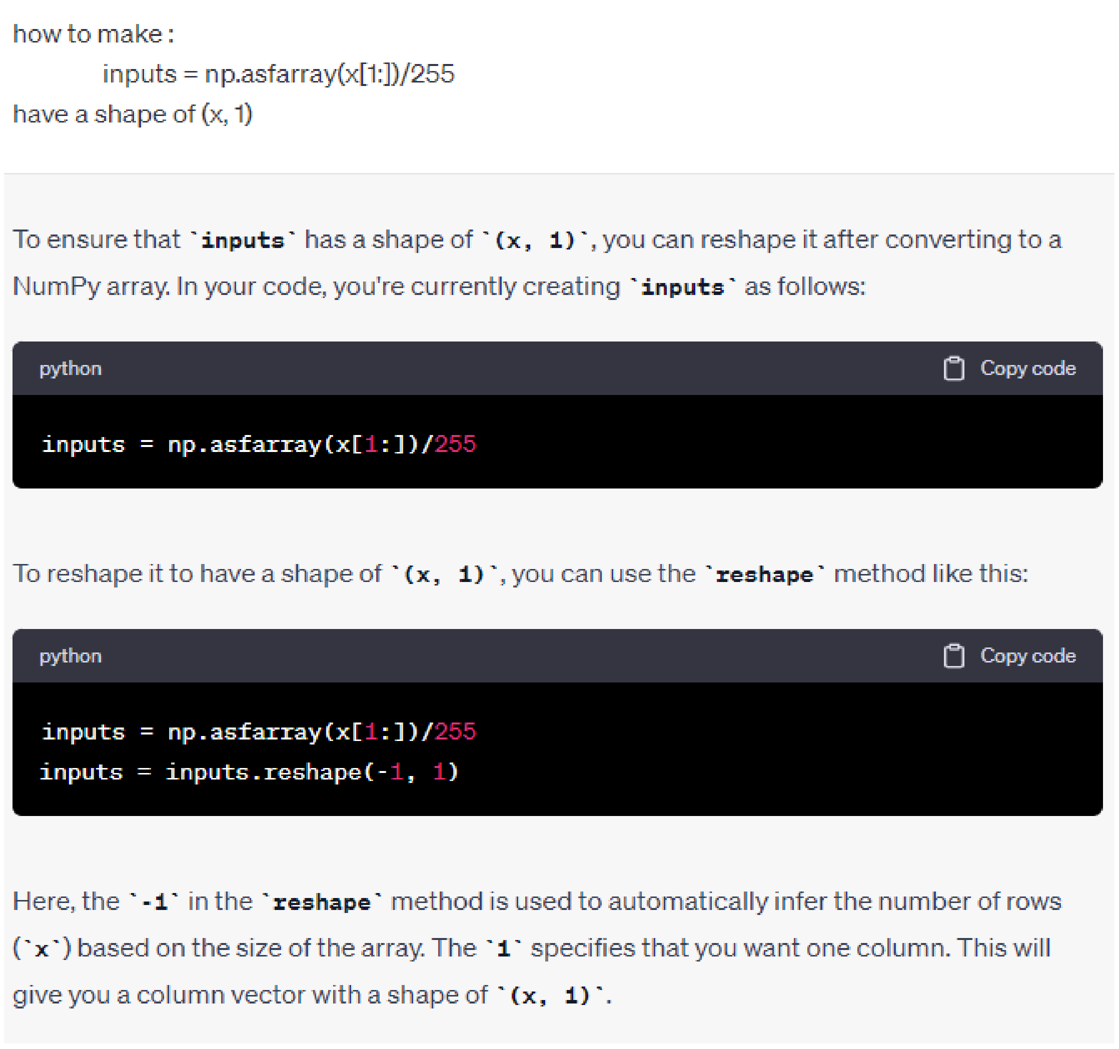
To fix this, you need to adjust your matrix dimensions or transpose one of the matrices. However, the correct adjustment depends on the specifics of your neural network architecture. You may need to check the dimensions of the weights and activations in the layers before this step to ensure that the dimensions align correctly.



To fix this issue, you should change the calculation for the gradient with respect to `B2`. Instead of using the dot product, you should perform a simple sum along the appropriate axis. Here's the modified code:

```python
changingWeights['B2'] = np.sum(error, axis=1, keepdims=True) * self.sigmoid(
```

This change will calculate the gradient with respect to `B2` correctly, and it should resolve the error you're encountering.


how to make :

inputs = np.asfarray(x[1:])/255

have a shape of (x, 1)

To ensure that `inputs` has a shape of `(x, 1)`, you can reshape it after converting to a NumPy array. In your code, you're currently creating `inputs` as follows:

```python
inputs = np.asfarray(x[1:])/255
```

To reshape it to have a shape of `(x, 1)`, you can use the `reshape` method like this:

```python
inputs = np.asfarray(x[1:])/255
inputs = inputs.reshape(-1, 1)
```

Here, the `-1` in the `reshape` method is used to automatically infer the number of rows (`x`) based on the size of the array. The `1` specifies that you want one column. This will give you a column vector with a shape of `(x, 1)`.

Main issues

The main issue with my approach to creating this model is the use of the sigmoid function. Although easy to implement, the sigmoid function is not the most efficient activation function that can be used in the hidden layer. This is because it suffers from what is known as the vanishing gradient function, which can slow down neural network training speeds. I had decided to approach the problem with the sigmoid function simply because I was more familiar with it.

Another issue would be overfitting of the network to training data. This can occur in one of several ways:

1. Learning rate is too large
2. Data set is too small
3. Training algorithm is run too many times

As the MNIST dataset contains 60,000 training samples, I determined that a training and development split of 58,000:2,000 was sufficient to test the generalisation of the model. This is also in addition to the fact that the dataset has provided a separate test set, which I have kept exclusively to test and validate the model after it has been trained.

This leaves us with issues 1 and 3, both of which come down to adjusting parameters of the model. This simply comes down to evaluating the algorithm after the adjustment of these “hyperparameters”.

Repeatability

Given enough computational power, a sufficiently large dataset, and a disregard for time, this model could theoretically be trained to 100% accuracy. However, as I am a humble university student with a laptop, I have adjusted the hyperparameters in such a way as to give me an approximate 85% accuracy on an unseen test set. The weight and bias values for the system were then saved in a pickled python file for easy importation. This allows me to configure the network and emulate a fully trained network, without the time requirement of training the network from scratch each time.

Links

Here is the link to a public repository of my project:

https://github.com/16Bocchi/CNN

For the purpose of demonstration, the colab notebook has the training commented out and the parameter loading activated. Here is the link to the google colab notebook:

13556384 Demo.ipynb