Report on

Implementation and Analysis of a Multi-Layer Neural Network for Handwritten Digit Classification using GradientTape and model.fit()

Submitted by

Jannatun Ferdous
Student ID: 1912076147
Dept. of Computer Science and Engineering
University of Rajshahi

Contents

1 Manually draw a neural network having 3 hidden layers for classifying images of 10 English digits.

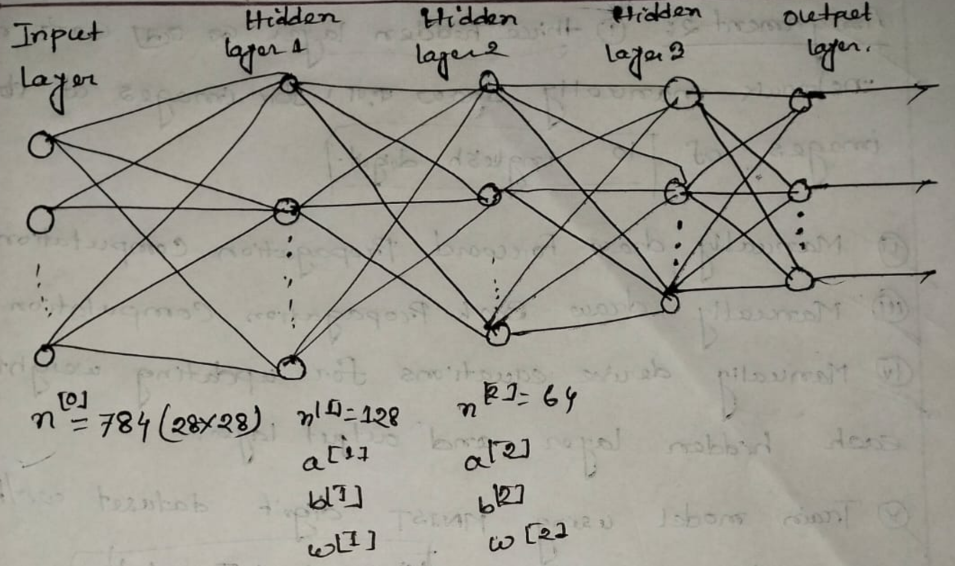


Fig-1: Neural network having 3 Hidden Layers for classifying images of 10 English digits.

Where,

n = number of neuron

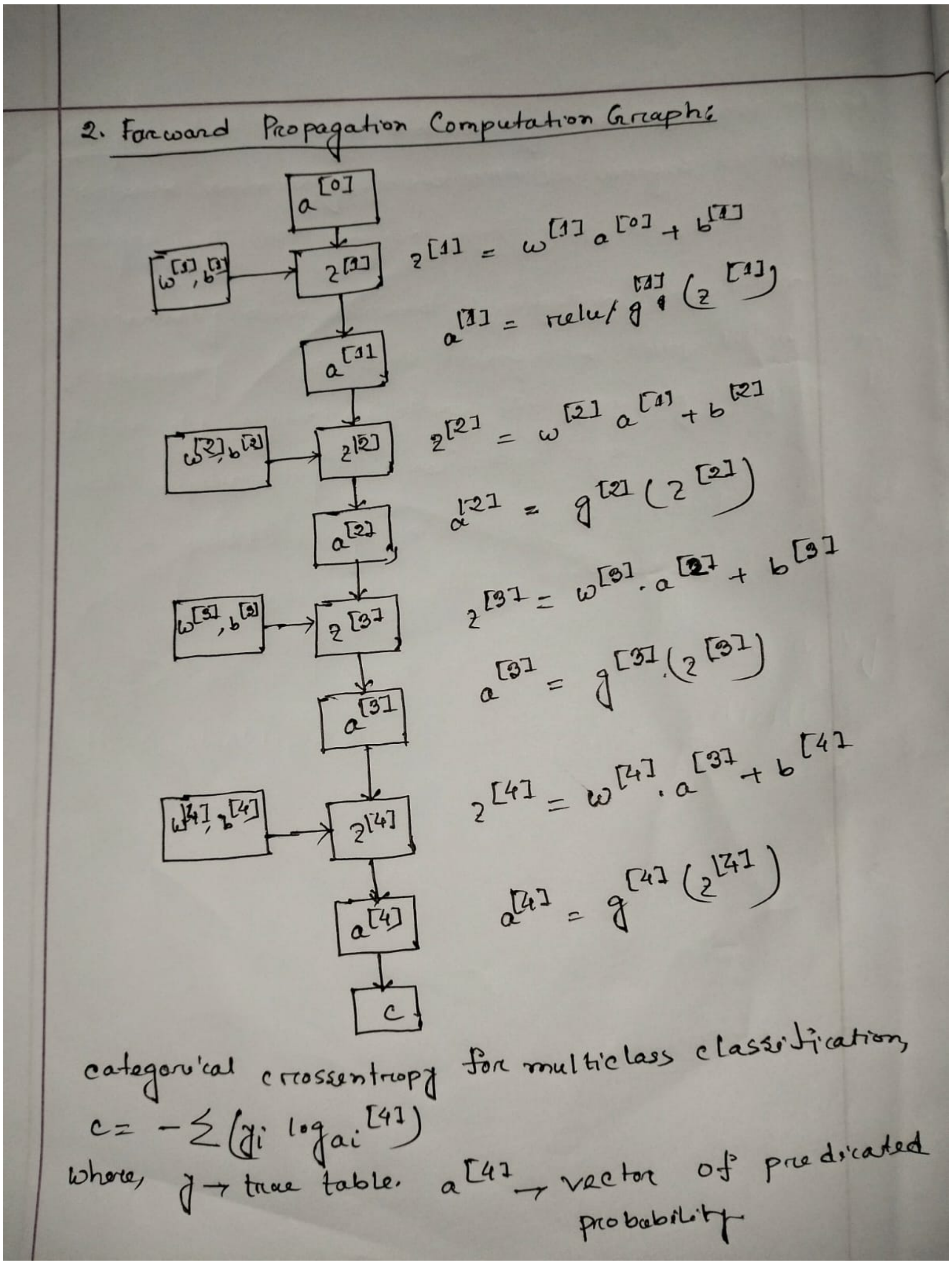
a → vector of activation function

b → vector of biases

w → matrix of weights.

Figure 1: 3 hidden layers for classifying images of 10 English digits.

2 Manually draw Forward Propagation Computation Graph.



2. Forward Propagation Computation Graph:

$$\begin{tikzcd}
 & \boxed{a^{[0]}} \arrow[d] \\
\boxed{w^{[1]}, b^{[1]}} \arrow[r] & \boxed{z^{[1]}} \arrow[d] \\
 & \boxed{a^{[1]}} \arrow[d] \\
\boxed{w^{[2]}, b^{[2]}} \arrow[r] & \boxed{z^{[2]}} \arrow[d] \\
 & \boxed{a^{[2]}} \arrow[d] \\
\boxed{w^{[3]}, b^{[3]}} \arrow[r] & \boxed{z^{[3]}} \arrow[d] \\
 & \boxed{a^{[3]}} \arrow[d] \\
\boxed{w^{[4]}, b^{[4]}} \arrow[r] & \boxed{z^{[4]}} \arrow[d] \\
 & \boxed{a^{[4]}} \arrow[d] \\
 & \boxed{c}
\end{tikzcd}$$

$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = \text{relu}/ g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]})$

$z^{[3]} = w^{[3]} \cdot a^{[2]} + b^{[3]}$

$a^{[3]} = g^{[3]}(z^{[3]})$

$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$

$a^{[4]} = g^{[4]}(z^{[4]})$

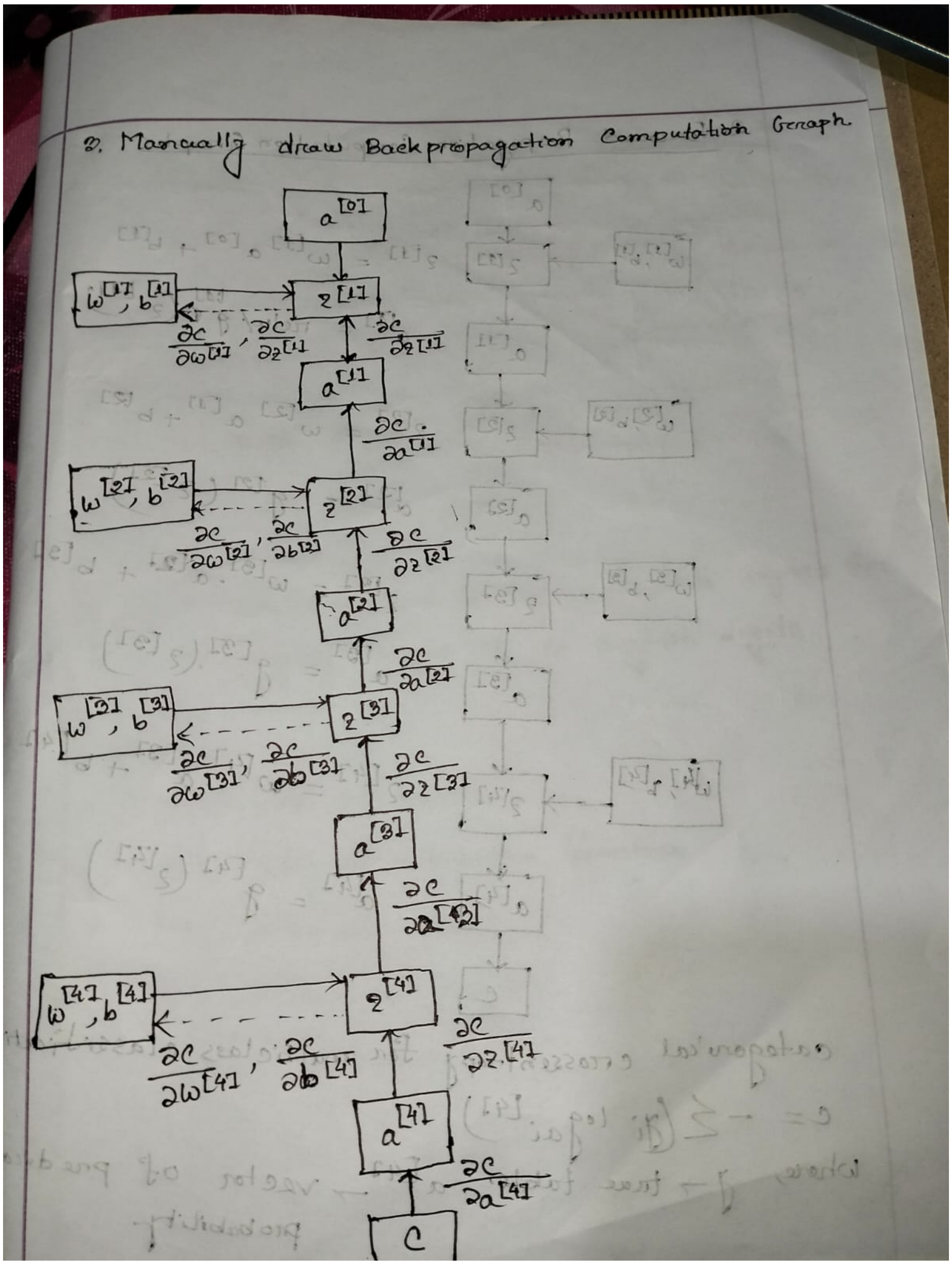
categorical crossentropy for multiclass classification,

$c = -\sum (y_i \log a_i^{[4]})$

where, $y \rightarrow$ true table. $a^{[4]} \rightarrow$ vector of predicated probability

Figure 2: Forward Propagation Computation Graph

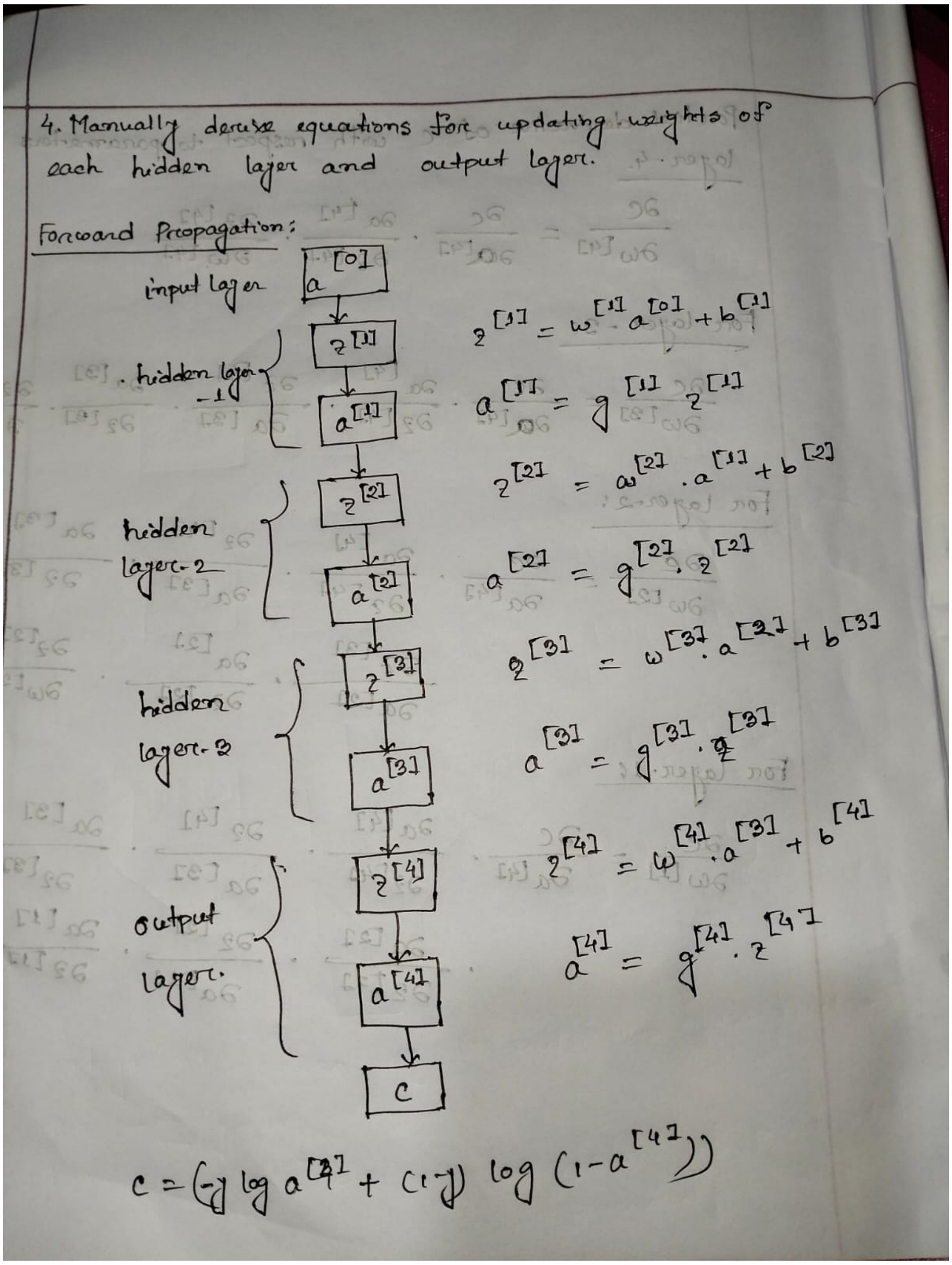
3 Manually draw Backpropagation Computation Graph



3. Manually draw Backpropagation Computation Graph
$a^{[0]}$
$w^{[1]}, b^{[1]}$
$z^{[1]}$
$\frac{\partial c}{\partial w^{[1]}}, \frac{\partial c}{\partial z^{[1]}}$
$\frac{\partial c}{\partial z^{[1]}}$
$a^{[1]}$
$\frac{\partial c}{\partial a^{[1]}}$
$w^{[2]}, b^{[2]}$
$z^{[2]}$
$\frac{\partial c}{\partial w^{[2]}}, \frac{\partial c}{\partial b^{[2]}}$
$\frac{\partial c}{\partial z^{[2]}}$
$a^{[2]}$
$\frac{\partial c}{\partial a^{[2]}}$
$w^{[3]}, b^{[3]}$
$z^{[3]}$
$\frac{\partial c}{\partial w^{[3]}}, \frac{\partial c}{\partial b^{[3]}}$
$\frac{\partial c}{\partial z^{[3]}}$
$a^{[3]}$
$\frac{\partial c}{\partial a^{[3]}}$
$w^{[4]}, b^{[4]}$
$z^{[4]}$
$\frac{\partial c}{\partial w^{[4]}}, \frac{\partial c}{\partial b^{[4]}}$
$\frac{\partial c}{\partial z^{[4]}}$
$a^{[4]}$
$\frac{\partial c}{\partial a^{[4]}}$
C

Figure 3: Backward Propagation Computation Graph

4 Manually derive equations for updating weights of each hidden layer and output layer.



4. Manually derive equations for updating weights of each hidden layer and output layer.

Forward Propagation:

$$\begin{tikzcd}
a^{[0]} \arrow[d] \\
z^{[1]} \arrow[d] \\
a^{[1]} \arrow[d] \\
z^{[2]} \arrow[d] \\
a^{[2]} \arrow[d] \\
z^{[3]} \arrow[d] \\
a^{[3]} \arrow[d] \\
z^{[4]} \arrow[d] \\
a^{[4]} \arrow[d] \\
c
\end{tikzcd}$$

input layer
hidden layer-1
hidden layer-2
hidden layer-3
output layer.

$$z^{[1]} = w^{[1]} \cdot a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]} z^{[1]}$$

$$z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]} \cdot z^{[2]}$$

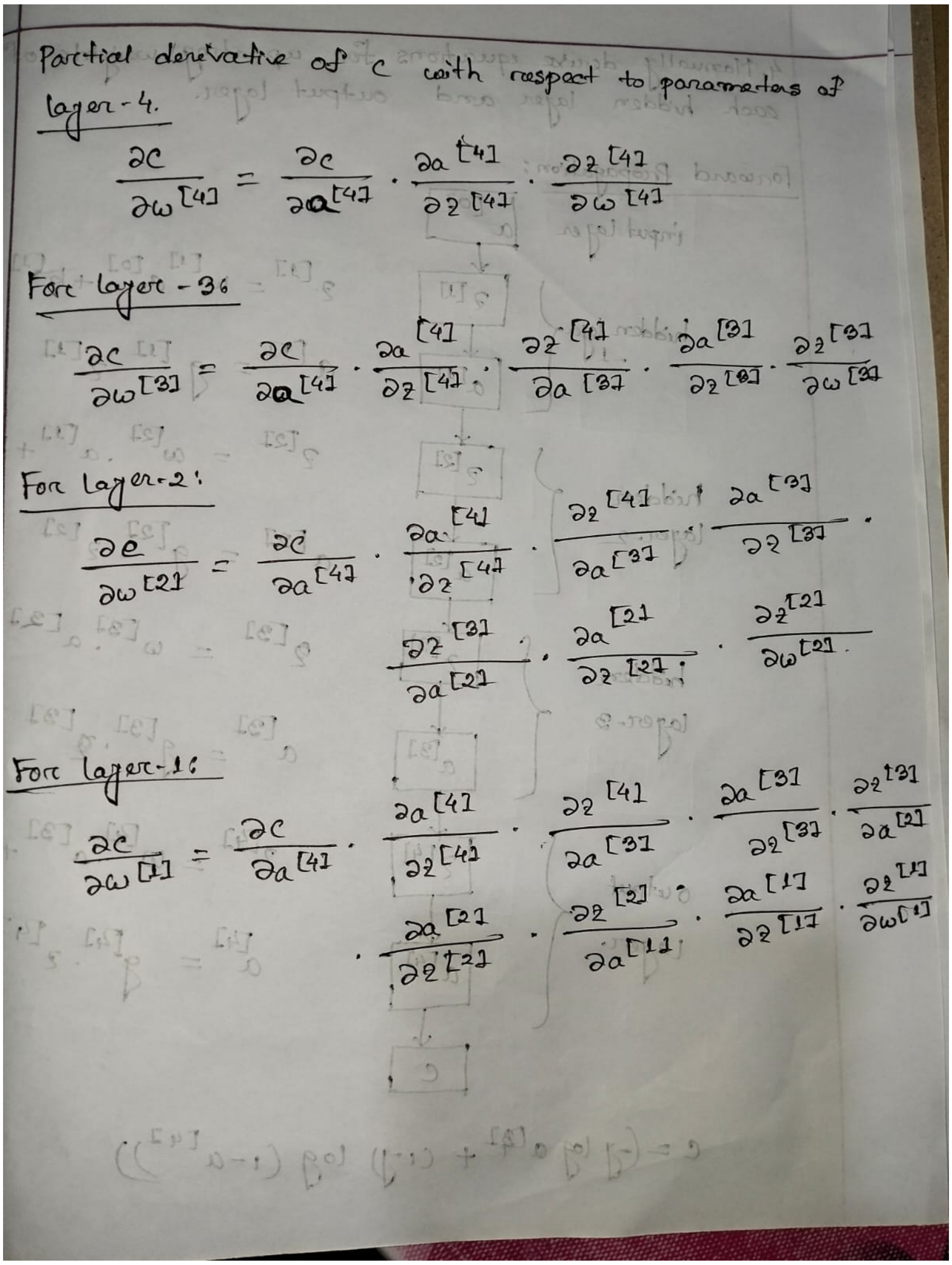
$$z^{[3]} = w^{[3]} \cdot a^{[2]} + b^{[3]}$$

$$a^{[3]} = g^{[3]} \cdot z^{[3]}$$

$$z^{[4]} = w^{[4]} \cdot a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]} \cdot z^{[4]}$$

$$c = (-y) \log a^{[4]} + (1-y) \log (1-a^{[4]}))$$



Partial derivative of c with respect to parameters of layer-4.

$$\frac{\partial c}{\partial w^{[4]}} = \frac{\partial c}{\partial a^{[4]}} \cdot \frac{\partial a^{[4]}}{\partial z^{[4]}} \cdot \frac{\partial z^{[4]}}{\partial w^{[4]}}$$

For layer-3:

$$\frac{\partial c}{\partial w^{[3]}} = \frac{\partial c}{\partial a^{[4]}} \cdot \frac{\partial a^{[4]}}{\partial z^{[4]}} \cdot \frac{\partial z^{[4]}}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

For Layer-2:

$$\frac{\partial e}{\partial w^{[2]}} = \frac{\partial c}{\partial a^{[4]}} \cdot \frac{\partial a^{[4]}}{\partial z^{[4]}} \cdot \frac{\partial z^{[4]}}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

For layer-1:

$$\frac{\partial c}{\partial w^{[1]}} = \frac{\partial c}{\partial a^{[4]}} \cdot \frac{\partial a^{[4]}}{\partial z^{[4]}} \cdot \frac{\partial z^{[4]}}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

Figure 4: Manual derivation of weight update equations for hidden and output layers.

5 Train model using the MNIST digit dataset with the help of Tensorflow's tf.GradientTape

The compile jupyter notebbok for this implementation can be found here: [**Link**]

6 Methodology

The experiment was conducted using the MNIST handwritten digit dataset, consisting of 60,000 training images and 10,000 testing images. Each image is a 28×28 grayscale digit (0–9), normalized to the range [0,1] before training. The dataset was further split into training and validation sets to evaluate model generalization.
A multi-layer feedforward neural network was designed with three fully connected hidden layers. Each hidden layer employed the ReLU activation function to introduce non-linearity, and the output layer used a softmax activation function to produce probability distributions over the ten digit classes.
Two training approaches were implemented:

1. **Custom Training Loop with `tf.GradientTape`:**
   - Forward propagation and loss computation were explicitly defined.
   - Gradients were calculated with `tf.GradientTape` and applied using the Adam optimizer.
   - This method provided flexibility in monitoring the training process and experimenting with custom update rules.
2. **High-Level Training with `model.fit()`:**
   - The model was compiled with the Adam optimizer and categorical cross-entropy loss.
   - Training was executed using TensorFlow's built-in `model.fit()` function, which abstracts the training loop.
   - This method required fewer lines of code and reduced implementation complexity.

Training was carried out for 5 epochs with a batch size of 128. Both approaches were executed on the same architecture and data set to ensure a fair comparison.

7 Compare the performance of the model trained by `tf.GradientTape()` with `TensorFlow`'s `model.fit()`

The model achieved strong classification performance on the MNIST dataset using both training approaches. Table 1 summarizes the final test accuracy after 5 epochs:

Training Method	**Final Test Accuracy**
`tf.GradientTape` (Custom Loop)	0.9773
`model.fit()` API	0.9750

Table 1: Final test accuracy after 5 epochs.

Both methods converged to nearly identical results, with only a minor difference in accuracy (0.23%). This difference can be attributed to stochastic factors such as random initialization and mini-batch shuffling.
The results confirm that while `model.fit()` provides simplicity and rapid prototyping, the `tf.GradientTape` method offers flexibility and transparency, making it more suitable when custom loss functions or non-standard training procedures are required.

8 Conclusion

This study compared two training approaches for a multi-layer neural network on the MNIST dataset: a custom loop using `tf.GradientTape` and TensorFlow's high-level `model.fit()` API. The results confirm that both methods effectively implement the same backpropagation process while using the gradient descent optimization algorithm. Both achieved nearly identical accuracy, indicating that performance remains consistent regardless of the training approach. However, `model.fit()` provides simplicity and rapid prototyping, whereas `tf.GradientTape` offers greater flexibility and control, making it more suitable for customized training tasks and experimental research.