Back–Propagation Machine Learning Algorithm

Rabia Akhtar * *StudentID* : 23031641

1 Introduction

Machine learning methods train Artificial Neural Networks (ANNs) via back-propagation. ANNs are inspired by brain neurones. Using simplified methods, computer scientists simulate biological Neural Networks (NNs). Back-propagation optimises supervised learning by modifying weights to decrease error between expected and true outputs [1]. It works with gradient descent optimisation to help the network learn from its mistakes. Forward-propagation sends inputs across the network to calculate output, while back-propagation sends error backward to alter weights.

2 Some Preliminaries

2.1 Biological NNs

A typical neurone has a cell body, nucleus, dendrites, and axon. Thin dendrites protrude from the cell body. Cells extend axons from the cell body. Neurons receive and send messages via dendrites and axons. The fundamental function of synapses is to transmit impulses between neurone axons and dendrites. All neurons are excitable because their membranes maintain voltage gradients.

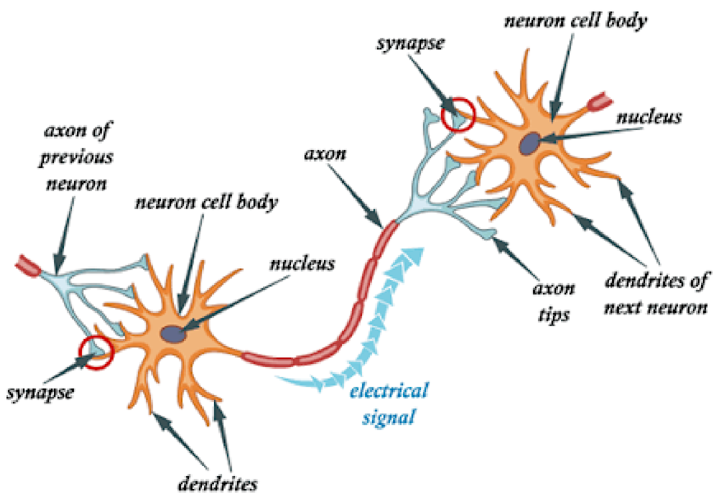


Figure 1: Biological NNs

*Author's github: `https://github.com/Rabia-Akhtr/Back-propagation-Machine-Learning-Tutorial`

2.2 ANN

Any brain network has connected nodes as its neural processes. We call these nodes neurones. A neurone is just a number shown. Any link between synthetic neurones is an axon in a real brain. The strength of ANN connections and the weights of neurones change throughout learning. What's the big deal? We want to discover a set of weights that allow the neural network to classify numbers when trained on training data. Each job and dataset has distinct weights.

2.2.1 Perceptron

A perceptron represents a fundamental model within ANNs, comprising two distinct categories of nodes: input nodes that signify the input attributes, and an output node that denotes the model's output. Figure 1 depicts the fundamental structure of a perceptron, which receives $n + 1$ input attributes, $1, x_1, x_2, \cdots, x_n$, and gives inner product sum z. An attribute x_j represents an input-node which is linked through a weighted connection w_j to the output-node. The activation function $y = f(z)$ like Sigmoid, Tanh, ReLU etc gives the mostly binary values [2]. Every neurone takes in one or multiple inputs, analyses them, and generates an output. Mathematically

$$z = \sum(w_j.x_j) + b$$

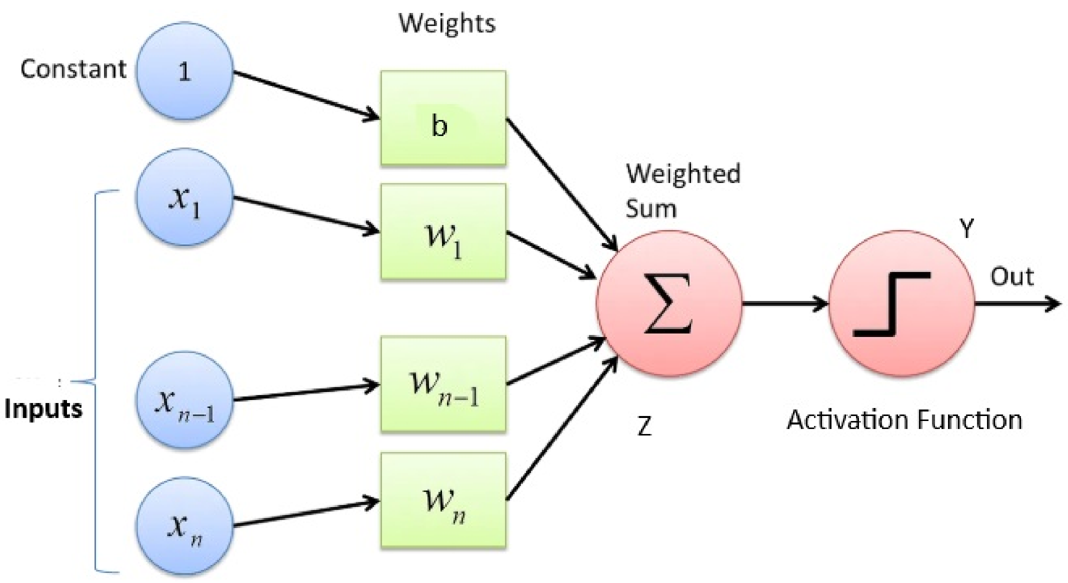


Figure 2: Perceptron Basic Architecture

2.2.2 Multi–Layer Neural Network(MLNN)

An MLNN is a form of ANN consisting of an input, output and hidden-layers.

input-layer: Receives raw data and assigns nodes to data-set features.
Hidden-Layer: Hidden-layers do intermediate computations and have neurons that apply activation functions to change input data non-linearly.

Output-Layer: The output-layer presents the network's outcome.

The Figure 2 depicts a MLNN with 2 hidden-layers, along with one input and one output layer. In interactions, each node exhibits a specific degree of bias. The input-layer has n input variables $x_1, x_2, \cdots, x_n$, whereas the output-layer consists of m output variables $y_1, y_2, \cdots, y_m$ [3].

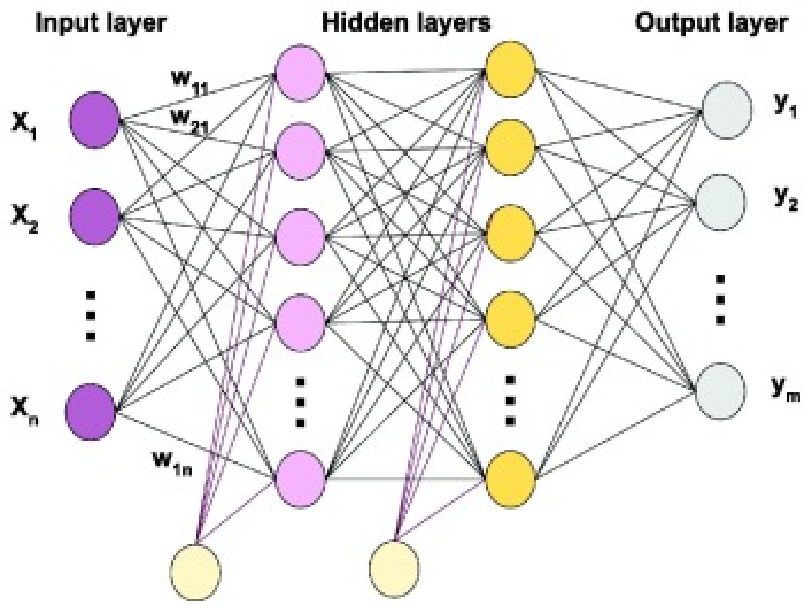


Figure 3: MLNN [3]

3 Back–Propagation Neural Network: Unveiling the Mysteries

3.1 Forward and back-propagation

From input to output, the NN world is a mesmerising ballet of forward and back-propagation. How about we do it one step at a time?

Forward Propagation (The prediction Phase): Forward-propagationstarts the process when data is fed into a NN. An MLNN applies weights and biases to the input values before passing them on to activation functions. Forecasts, or our model's calculated predictions given the data we feed it, are the end product of this process.
Prediction vs. True Values: A loss function, which assesses the performance of our model, quantifies the difference between predicted and true values.
Back-propagation (Learning from Errors): It is time to do back-propagation after we have our forecasts and actual data. In order to minimise the loss function, this technique iteratively modifies the weights and biases. The process is as follows:
Loss Calculation: To find out how far the forecasts are from the actual numbers, use the loss function.
Gradients Computation: The loss gradients with respect to each network weight are computed using a back-propagation algorithm.
Update Weights: By utilising optimisation algorithms such as gradient descent, we may utilise these

gradients to adjust the weights in a way that minimises loss.

Iterative Procedures: This propagation cycle repeats itself, going both ways. The model is improved with each data transfer, leading to a steady minimisation of the loss function. The network learns the data patterns via several cycles, gradually increasing its accuracy.

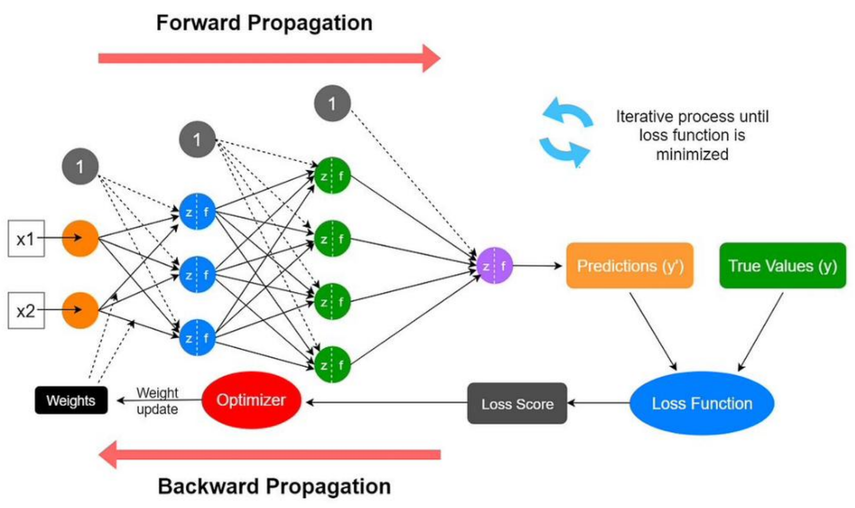


Figure 4: Back-propagation Iterative Process

3.2 Calculus Role in Optimization

A step-by-step derivation of weight updates in a simple NN:
The weighted sum of inputs is computed as:

$$z = W \cdot X + b$$

where W is the weight vector, X is the input vector and b is bias.
The sigmoid activation function has been used as:

$$y = \frac{1}{1 + e^{-z}}$$

The loss function L_0 can be represented as:

$$L_0 = 1 - y$$

To compute gradient of loss function L_0, using chain rule $w.r.t$ the weights:

$$\frac{\partial L_0}{\partial W} = \frac{\partial L_0}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial W} \tag{3.1}$$

Computing the derivative of Loss function $\frac{\partial \mathrm{L}_0}{\partial y} = -1$, the slope of the Sigmoid function $\frac{\partial y}{\partial z} = y \cdot (1 - y)$, and the derivative of the resulting scalar z $w.r.t$ W i.e $\frac{\partial z}{\partial W} = X$. Utilizing these values in (3.1), we get

$$\frac{\partial L_0}{\partial W} = -1 \cdot (y \cdot (1 - y)) \cdot X \tag{3.2}$$

The gradient descent rule modifies W in the direction that minimises the loss L_0. In order to accomplish this, a scaled version of the gradient is subtracted:

$$\Delta W = -\mu \frac{\partial L_0}{\partial W}$$

The final updated weight is

$$W \leftarrow W - \mu \frac{\partial L_0}{\partial W}$$

3.3 Understanding Backpropagation Through a Teacher-Student Analogy

Backpropagation is an important idea in training neural networks. It's easier to understand if you think of it as a return loop, which is like a teaching situation. In this analogy, we use the idea of a neural network (NN) as an example of a student learning from a teacher. Imagine that the neural network is a student, and the teacher is the instructor guiding the student to improve their performance.
i. In this comparison, inputs (data) represent lessons taught.
ii. The forward pass is similar to a student taking an exam after studying the material and making predictions.
iii. Loss is the difference between student predictions and accurate responses.
iv. Back-propagation acts as instructor feedback to students. The teacher helps students develop by pointing out areas for improvement and modifying weights for future assessments.
v. The student's learning process involves weight adjustments depending on feedback to better on future tests.

4 Coding for Training and Testing Dataset

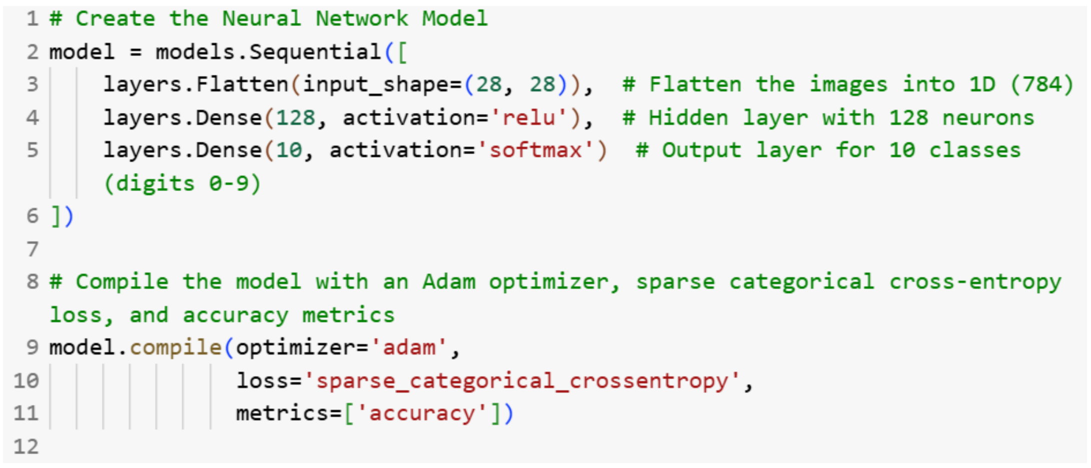
i. Loading and Normalizing the MNIST Dataset
We accelerate training convergence after normalising the data. Data information includes 28×28 pixel

```python
# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()

# Normalize the images to a range of [0, 1]
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Figure 5: Python Code for Loading and Normalizing the MNIST Dataset

grayscale pictures. Each pixel value is an integer between 0 and 255, thus we divide by 255 to normalise them to $[0, 1]$.
ii. Building the NN Model
Create a one-hidden-layer NN model. An ANN architecture based on one hidden-layer will include:

- **Input-layer:** 28×28 picture, flattened into 784-size $1D$ vector.
- **Hidden-Layer:** The Hidden Layer is a dense layer with 128 neurons with ReLU activation.
- **Output-layer:** A dense layer with 10 neurons for each digit class $(0 - 9)$, employing softmax activation for probability distribution.



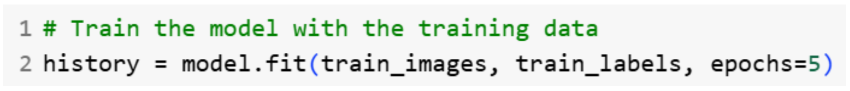
```python
# Create the Neural Network Model
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # Flatten the images into 1D (784)
    layers.Dense(128, activation='relu'),  # Hidden layer with 128 neurons
    layers.Dense(10, activation='softmax')  # Output layer for 10 classes (digits 0-9)
])

# Compile the model with an Adam optimizer, sparse categorical cross-entropy loss, and accuracy metrics
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

```

Figure 6: Python Code for Building the Neural Network Model

iii. Training the Model with Back-propagation

The code given in Figure 7 include:

- The forward pass calculates output from input.
- The loss function calculates error by comparing predicted output to true label.
- Back-propagation generates error gradients for each weight in the network and adjusts weights accordingly.

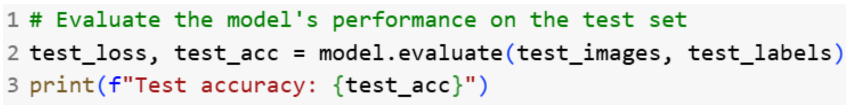


```python
# Train the model with the training data
history = model.fit(train_images, train_labels, epochs=5)
```

Figure 7: Python Code for Training the Model with Back-propagation

iv. Evaluating Model Performance

The back-propagation model modifies its weights to reduce the error on the training data-set and we

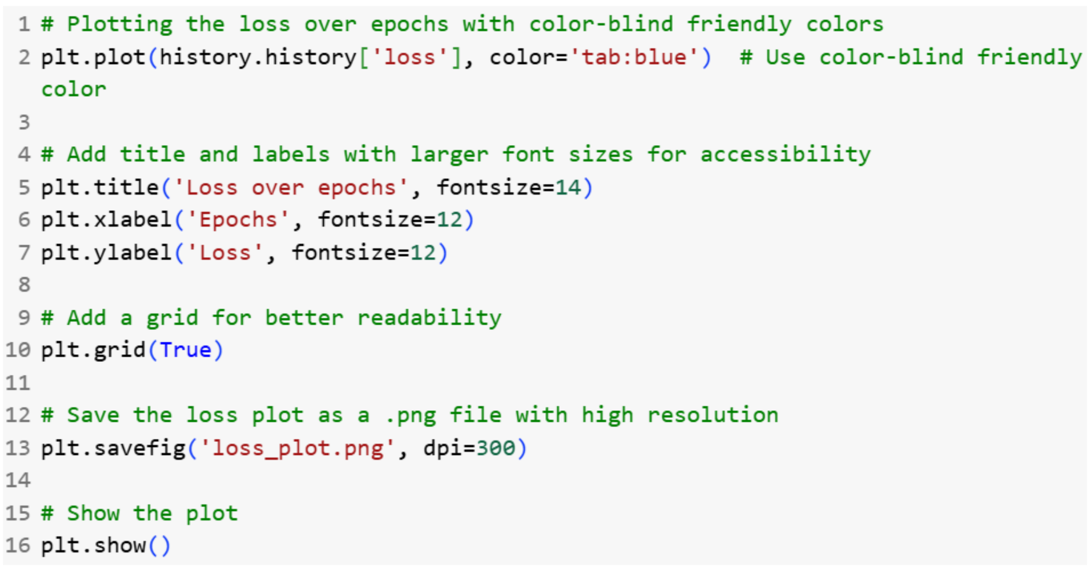


```python
# Evaluate the model's performance on the test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

Figure 8: Python Code for Evaluating Model Performance

can see the code in Figure 8. Subsequent to training, examine the model on the test set to determine its efficacy on unknown data. This will provide the test accuracy and test loss.

v. Visualizing Loss Reduction Over Epochs



```python
# Plotting the loss over epochs with color-blind friendly colors
plt.plot(history.history['loss'], color='tab:blue')  # Use color-blind friendly color

# Add title and labels with larger font sizes for accessibility
plt.title('Loss over epochs', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Loss', fontsize=12)

# Add a grid for better readability
plt.grid(True)

# Save the loss plot as a .png file with high resolution
plt.savefig('loss_plot.png', dpi=300)

# Show the plot
plt.show()
```

Figure 9: Python Code for Visualizing Loss Reduction Over Epochs

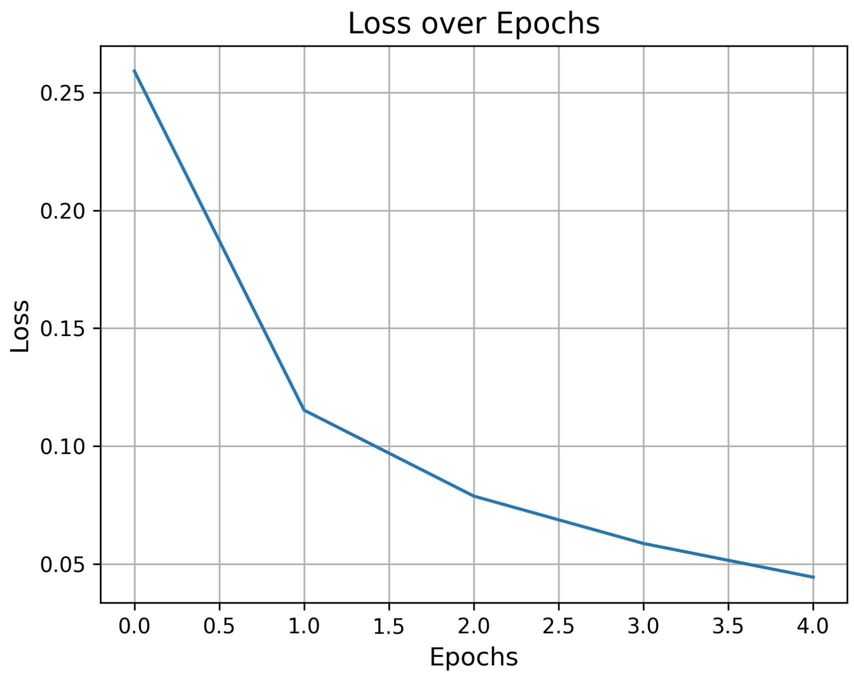


Figure 10: Plot for Loss over Epochs

The loss across epochs may be visualised to show how back-propagation reduces error over time. As the model modifies its weights, a gradual reduction in loss is expected.

vi. Visualizing Model Predictions

```python
# Display the test image and its predicted label
plt.imshow(test_images[0], cmap=plt.cm.binary)  # Grayscale image, binary
colormap

# Add title with high contrast for accessibility
predicted_label = model.predict(test_images[0:1]).argmax()
plt.title(f"Predicted label: {predicted_label}", fontsize=14, color='black')  #
Black title for contrast

# Add grid for better readability (optional)
plt.grid(False)  # We may not want a grid on image plots

# Save the image with the prediction title
plt.savefig('predicted_image.png', dpi=300)  # Save as PNG with high resolution

# Show the plot
plt.show()
```

Figure 11: Python Code for Visualizing Model Predictions

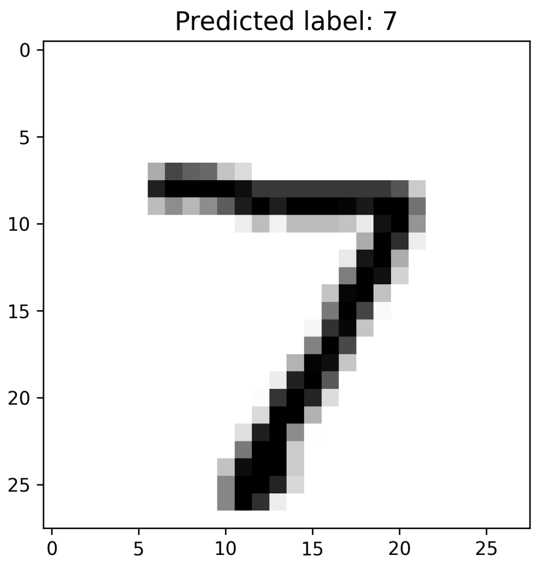


Figure 12: Plot for Predicted label: 7

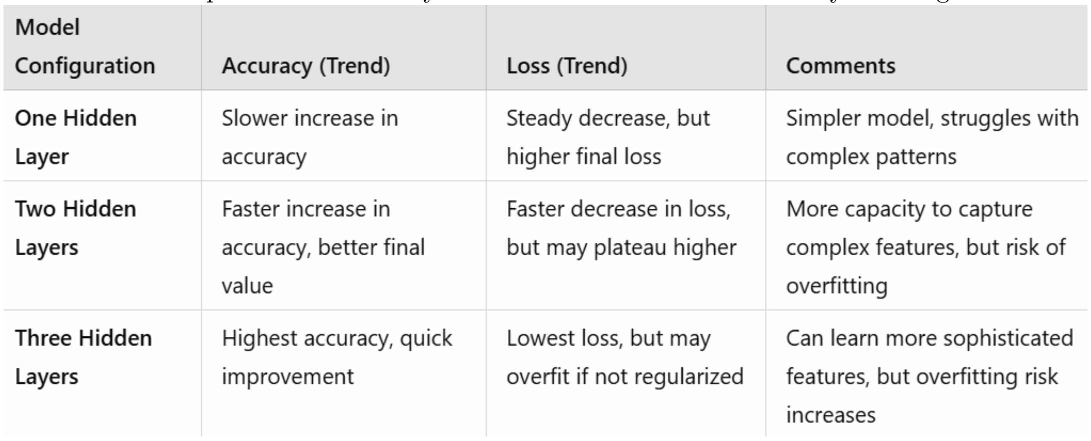
To demonstrate how well the trained model works, see a test image together with its expected label. The model's prediction relies on the weights that were modified throughout the training process using back-propagation. The label that is predicted aligns with the digit class that exhibits the highest probability as determined by the softmax function.

5 Discussion on Training and Testing

5.1 Comparison of Accuracy and Loss for Various Hidden-Layers Configurations

From the table 1, we can see the difference in accuracy and loss across models with one, two, and three hidden layers in this visual: As shown in Figure 13, the accuracy improves with the increase of number of hidden-layers. This improvement reflects the model's capacity to learn more complicated patterns. Figure 14 shows that the loss becomes smaller with increasing numbers of hidden-layers, which means that learning and error reduction become better.

Table 1: Comparison of Accuracy and Loss for Different Hidden Layer Configurations



Model Configuration	Accuracy (Trend)	Loss (Trend)	Comments
One Hidden Layer	Slower increase in accuracy	Steady decrease, but higher final loss	Simpler model, struggles with complex patterns
Two Hidden Layers	Faster increase in accuracy, better final value	Faster decrease in loss, but may plateau higher	More capacity to capture complex features, but risk of overfitting
Three Hidden Layers	Highest accuracy, quick improvement	Lowest loss, but may overfit if not regularized	Can learn more sophisticated features, but overfitting risk increases

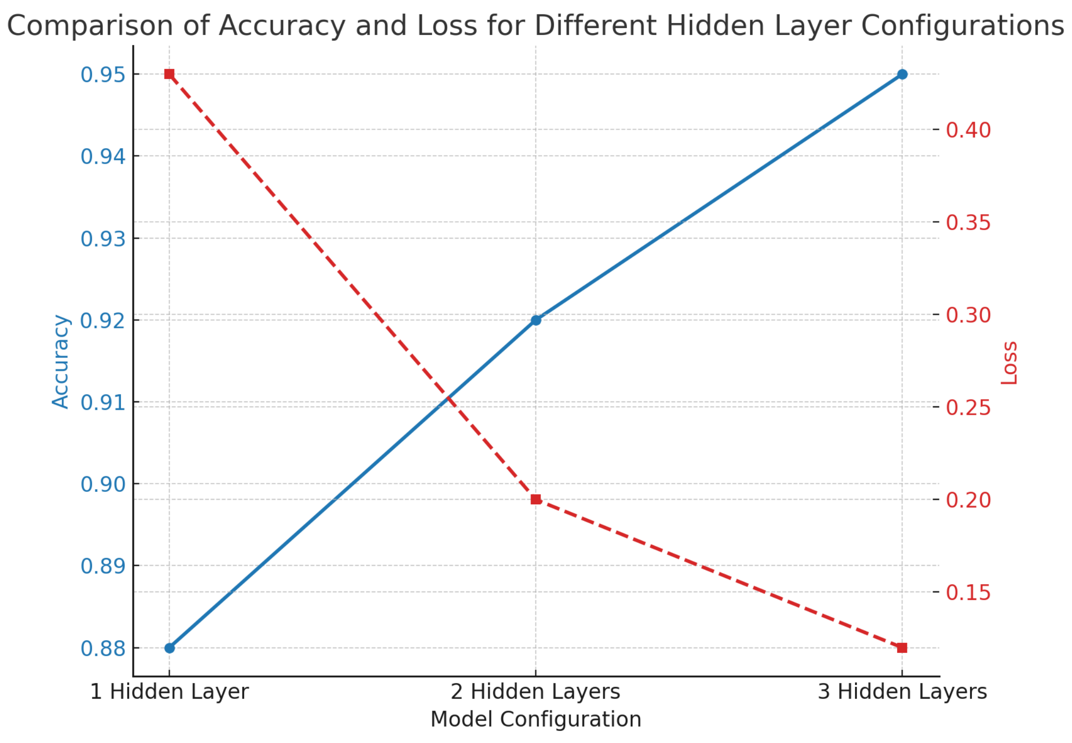


Figure 13: Comparison of Accuracy and Loss for Various Hidden Layer Configuration

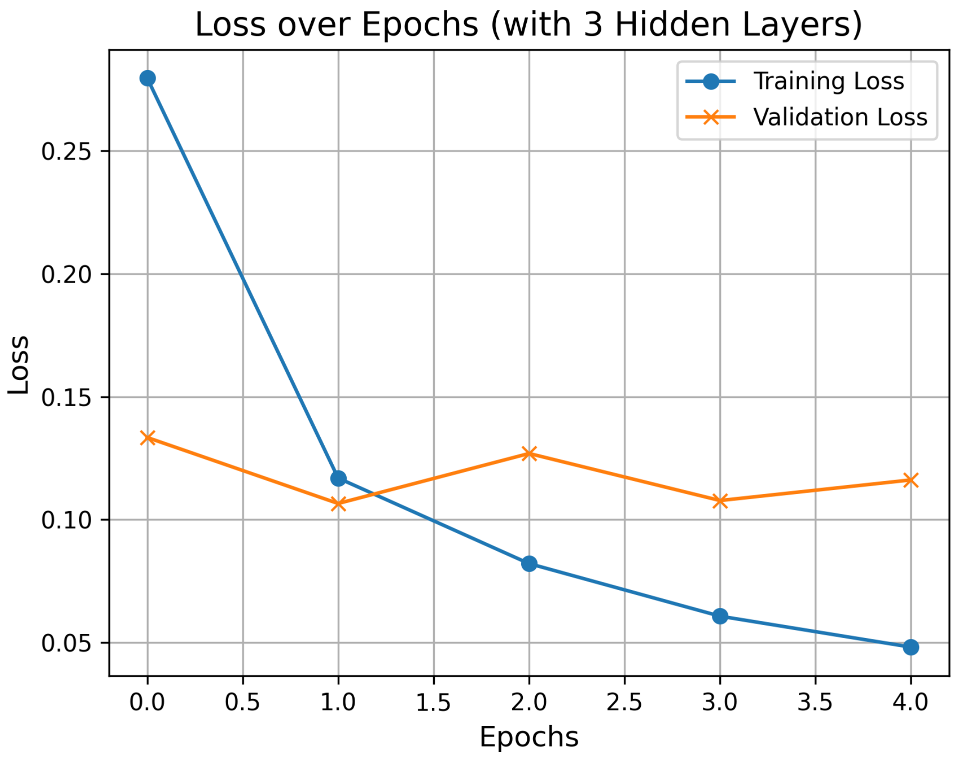


Figure 14: Comparison of Loss over Epochs (with 3 Hidden layers)

5.2 Comparison of Architectures (Hidden Layers + Neurons)

In Table 3 and Figure 15, the relationship between neurones and hidden layers is clearly shown. Since back-propagation properly adjusts weights, adding hidden layers and neurones improves the model's ability to recognise nuanced patterns. Insufficient layers or neurones might underfit, whereas excessive complexity can overfit. The best performance was achieved with three hidden layers and 256 neurones, proving that more complex networks operate. Back-propagation balances training and validation accuracy by using model complexity.

Table 2: Comparison of Architectures with Different Hidden Layers and Neurons

Hidden Layers	**Neurons per Layer**	**Training Accuracy (%)**	**Validation Accuracy (%)**	**Interpretation**
1	64	85.0	79.0 - 81.0	**Underfitting:** The model is too simple, underperforming on the validation set, suggesting it doesn't capture enough complexity.
1	128	87.5	81.0 - 82.0	**Better Fit:** Increasing neurons improves performance slightly, but the model still shows signs of underfitting with a noticeable gap between training and validation accuracy.
1	256	90.0	83.0 - 84.0	**Improved Performance:** More neurons help improve both training and validation accuracy, but it's still relatively simple.
2	64	88.0	81.5 - 83.0	**Balanced Performance:** The model is better able to capture patterns, but may still be too simple, as the validation accuracy doesn't significantly surpass 85%.
2	128	90.5	84.0 - 85.0	**Optimal Configuration:** This configuration strikes a good balance, showing high training and validation accuracy. It generalizes well.
2	256	92.0	85.0 - 86.0	**Good Generalization:** Increasing neurons further improves performance, although validation accuracy shows diminishing returns after a certain point.
3	64	90.5	82.5 - 83.5	**More Complexity, Still Underfitting:** The model has more layers, but the performance on validation data still lags behind the best configurations, showing signs of underfitting.
3	128	93.0	86.0 - 87.0	**Better Fit:** With 3 hidden layers, the model improves accuracy on both training and validation, demonstrating it is capable of learning more complex patterns.
3	256	94.5	88.0 - 88.5	**Best Performance**: This model configuration provides the highest training and validation accuracy, achieving the best generalization and capturing complex patterns.

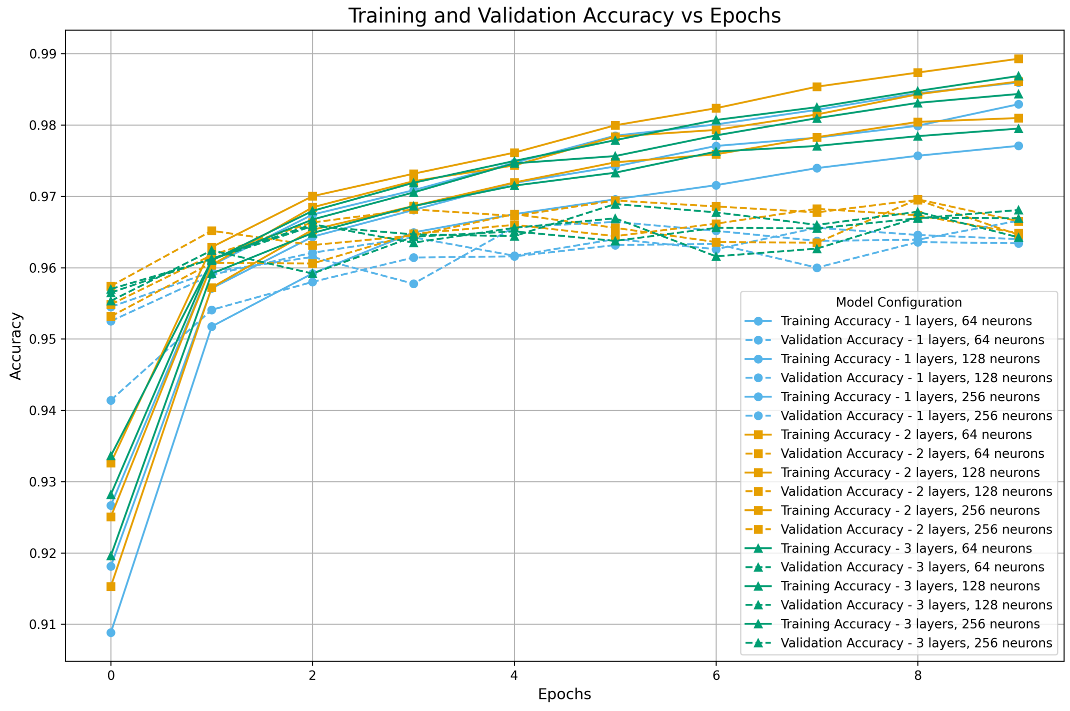


Figure 15: Comparison of Training and Validation Accuracy vs Epochs

5.3 Comparison of Optimization Algorithms in NNs

The Training Loss Comparison graphic shows each optimizer's epoch-long training loss decrease. The blue line SGD initially loses more and converges more slowly, giving it more epochs to develop. Adam (orange line) and RMSprop (green line) reduce loss faster, with Adam reducing loss the fastest. SGD starts with moderate accuracy and improves with time, but its convergence is slower, thus it takes longer to reach high validation accuracy. Adam achieves early validation accuracy, accelerating convergence and generalisation. RMSprop operates similarly to Adam, but with a little flaw, delivering a well-rounded effectiveness. We conclude that Adam outperforms SGD and RMSprop in convergence speed and generalisation, while SGD requires more epochs to attain optimal results.

Table 3: Comparison of Optimization Algorithms in NNs

Optimizer	**Test Accuracy**	**Test Loss**	**Final Training Accuracy**	**Final Training Loss**	**Convergence Speed**	**Key Characteristics**
SGD	0.9720	0.1119	0.9791	0.0757	Slow	Starts with higher loss, gradually improves, slower convergence
Adam	0.9711	0.1033	0.9864	0.0479	Fast	Converges faster, high validation accuracy due to adaptive learning rate and momentum
RMSprop	0.9740	0.1033	0.9837	0.0561	Moderate	Balances convergence speed and accuracy, adapts learning rate based on past gradients

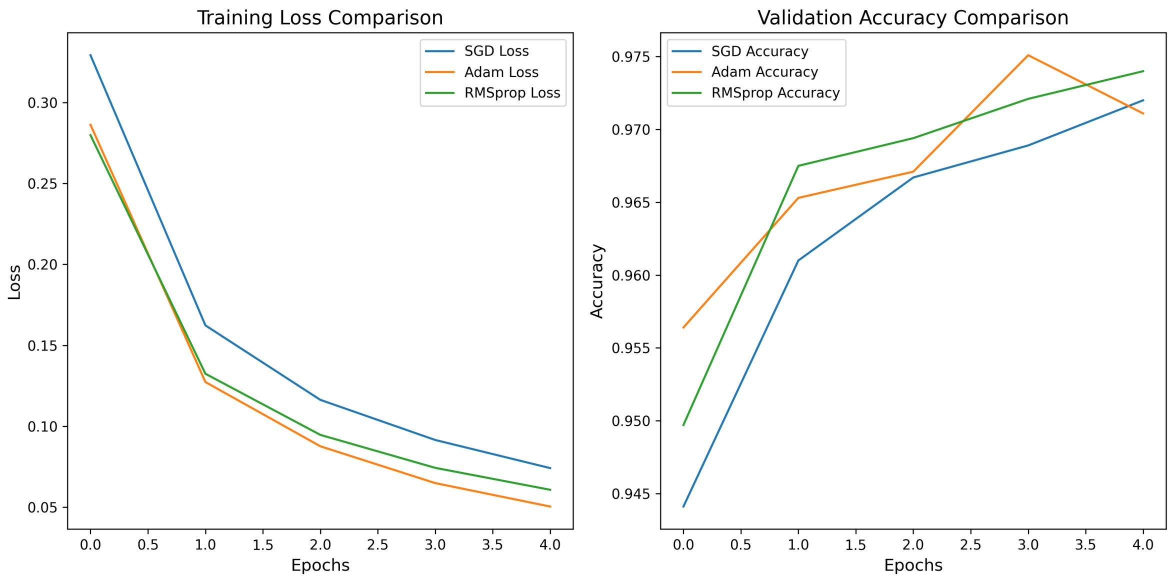
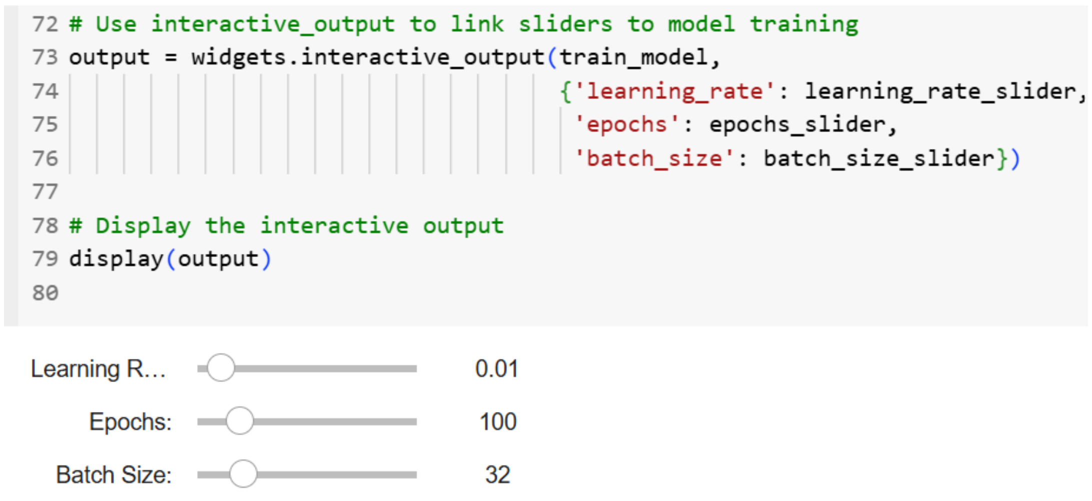


Figure 16: Comparison of Training Loss and Validation Accuracy

6 Teaching Back–propagation with Interactive Learning

NN training is affected by hyper-parameters such as learning rate, epochs, and batch size. Understanding how the NN learns and updates its weights through back-propagation helps reduce the loss function.

The most significant hyper-parameters of a NN are learning rate, epochs, and batch size. Learning rate affects model convergence (or divergence). Each epoch affects the model's overfitting or underfitting and learning. Stability and efficiency depend on batch size while updating weights.



```
# Use interactive_output to link sliders to model training
output = widgets.interactive_output(train_model,
                                    {'learning_rate': learning_rate_slider,
                                     'epochs': epochs_slider,
                                     'batch_size': batch_size_slider})

# Display the interactive output
display(output)

```

```
Learning R...   0.01
Epochs:         100
Batch Size:     32
```

Figure 17: Slider for Fine Tune of Learning Rate, Epochs and Batch Size

Figure 17 represents how changing the learning rate, epochs, and batch size affects the training process with the sliders. Figure 18 shows the training loss across epochs for a model trained with a

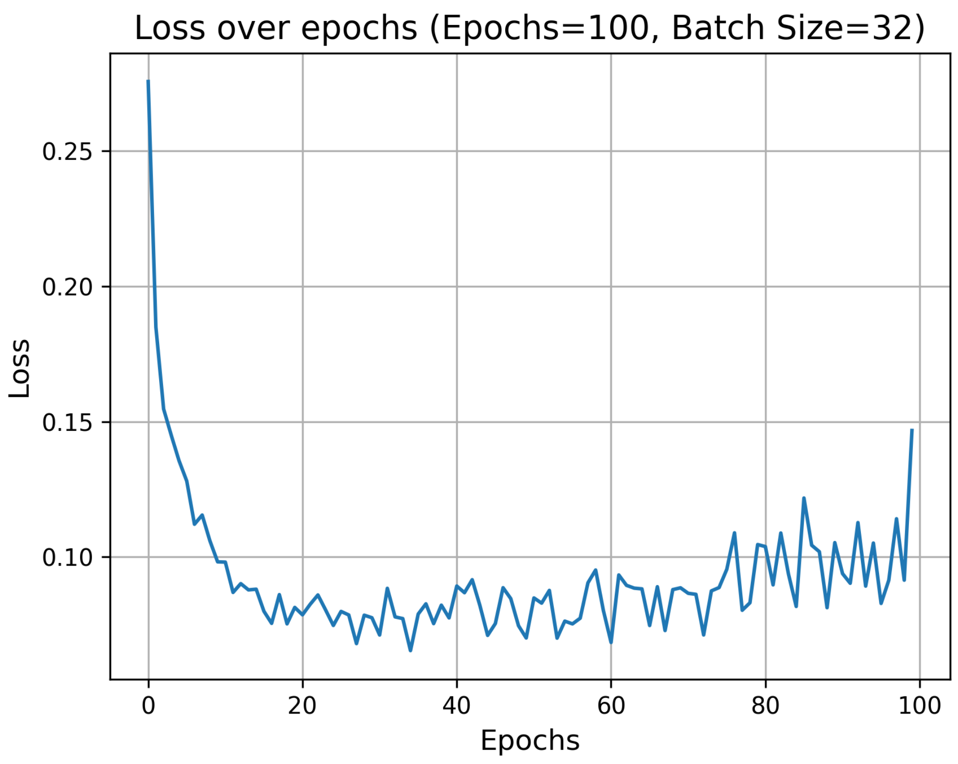


Figure 18: Comparison of Training Loss and Validation Accuracy

learning rate of 0.01, 100 epochs, and 32 batch size. The $x - axis$ shows training iterations as epochs.

The model's training data performance is shown by the loss on the $y - axis$. A reduced loss value means the model's predictions match labels better.

References

[1] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[2] Kumar V Stinbach M and Karpatne A. *Introduction to data mining.* Springer, 2006.

[3] Kit Yan Chan, Bilal Abu-Salih, Raneem Qaddoura, Al-Zoubi AlaM, Vasile Palade, Duc-Son Pham, Javier Del Ser, and Khan Muhammad. Deep neural networks in the cloud: Review, applications, challenges and research directions. *Neurocomputing*, 545:126327, 2023.