Report on MNIST Neural Network Implementation

By Anushree Asthana (id-MST03-0046)

Submitted to Scifor Technologies

UNDER GUIDIANCE OF Urooj Khan

1. Introduction

This report analyzes a Python implementation of a neural network for classifying handwritten digits using the MNIST dataset. The code utilizes TensorFlow and Keras to build, train, and evaluate a simple feedforward neural network.

2. Code Overview

The provided code can be broken down into several key sections:

1. Importing necessary libraries
2. Loading and preprocessing the MNIST dataset
3. Defining a function to display sample images
4. Creating and compiling the neural network model
5. Training the model
6. Evaluating the model's performance
7. Visualizing misclassified examples

3. Data Preparation

3.1 Dataset

The code uses the MNIST dataset, a widely-used benchmark in machine learning. It consists of 70,000 grayscale images of handwritten digits (0-9), split into 60,000 training images and 10,000 test images.

3.2 Preprocessing

The preprocessing steps include:

1. Reshaping the 28x28 pixel images into 1D arrays of 784 elements
2. Converting pixel values to float32 and normalizing them to the range [0, 1] 3. Converting labels to one-hot encoded vectors

```python train_images =
mnist_train_images.reshape(60000, 784) test_images =
mnist_test_images.reshape(10000, 784) train_images =
train_images.astype('float32') test_images =
test_images.astype('float32') train_images /= 255
test_images /= 255 train_labels =
keras.utils.to_categorical(mnist_train_labels, 10)
test_labels = keras.utils.to_categorical(mnist_test_labels,
10)
```

4. Data Visualization

The code includes a `display_sample` function to visualize individual images from the dataset:

```python def display_sample(num):
print(train_labels[num])     label =
train_labels[num].argmax(axis=0)     image =
train_images[num].reshape([28,28])
plt.title('Sample: %d  Label: %d' % (num, label))
plt.imshow(image, cmap=plt.get_cmap('gray_r'))
   plt.show()
```

This function is useful for understanding the data and verifying the preprocessing steps.

5. Model Architecture

The neural network model is a simple feedforward network with two layers:

1. Input layer: 784 neurons (one for each pixel)
2. Hidden layer: 512 neurons with ReLU activation
3. Output layer: 10 neurons with softmax activation

```python
model =
Sequential()
model.add(Dense(5
12, activation='relu',
input_shape=(784,))
)
model.add(Dense(1
0,
activation='softmax'
))
```

This architecture is relatively simple but can be effective for the MNIST classification task.

6. Model Compilation

The model is compiled with the following parameters:

- Loss function: Categorical crossentropy
- Optimizer: RMSprop
- Metric: Accuracy

```python
model.compile(loss='categorical_crossentropy',
        optimizer=RMSprop(),
metrics=['accuracy'])
```

These choices are appropriate for a multi-class classification problem like MNIST.

7. Model Training

The model is trained for 10 epochs with a batch size of 100:

```python history = model.fit(train_images,
train_labels,
                    batch_size=100,
epochs=10,                    verbose=2,
                    validation_data=(test_images, test_labels))
```

The training process uses the test set as validation data, which allows for monitoring of potential overfitting.

8. Model Evaluation

After training, the model's performance is evaluated on the test set:

```python score = model.evaluate(test_images, test_labels,
verbose=0) print('Test loss:', score[0]) print('Test
accuracy:', score[1])
```

This provides a final measure of the model's generalization ability.

9. Error Analysis

The code includes a section to visualize misclassified examples:

```python for x in
range(1000):
    test_image = test_images[x,:].reshape(1,784)
predicted_cat = model.predict(test_image).argmax()
label = test_labels[x].argmax()     if (predicted_cat !=
label):
        plt.title('Prediction: %d Label: %d' % (predicted_cat, label))
plt.imshow(test_image.reshape([28,28]), cmap=plt.get_cmap('gray_r'))
        plt.show()
```

This is valuable for understanding the types of errors the model makes and potentially identifying areas for improvement.

10. Potential Improvements

While the current implementation provides a solid baseline, several enhancements could potentially improve performance:

1. Data augmentation: Applying transformations to the training images could increase the effective size of the dataset and improve generalization.
2. Deeper architecture: Adding more layers to the network might capture more complex features.
3. Convolutional layers: Using convolutional layers could better capture the 2D structure of the images.
4. Regularization: Techniques like dropout or L2 regularization could help prevent overfitting.
5. Hyperparameter tuning: Systematically searching for optimal learning rates, batch sizes, and network architectures could yield better results.

11. Conclusion

This implementation demonstrates a basic approach to solving the MNIST digit classification problem using a neural network. While simple, it provides a foundation for understanding key concepts in deep learning, including data preprocessing, model architecture design, training, and evaluation.

The code structure is clear and well-commented, making it accessible for educational purposes. However, for production use or more challenging datasets, more advanced techniques would likely be necessary.

12. References

1. LeCun, Y., Cortes, C., & Burges, C. J. (2010). MNIST handwritten digit database. AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist
2. Chollet, F. (2017). Deep learning with Python. Manning Publications Co.
3. Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media.