Kaggle Challange #3
Digit Classification

Zack Peters
Dr. DeBruine

Introduction:

This project presents a deep learning approach to classify handwritten digits from the MNIST dataset, utilizing a neural network with data normalization through MinMaxScaler and a sequence of dense and ReLU activation layers. The model is trained and validated over several epochs, with validation accuracy serving as the key performance metric. By graphically analyzing validation accuracy trends, the model's generalization capability was assessed, demonstrating a focused method for improving digit classification accuracy in practical machine-learning applications.

Approach:

Imports

1. Pandas is a Python library used for data manipulation and analysis. It provides powerful data structures like DataFrames, making it easy to read, manipulate, and preprocess data. In the context of machine learning, it's often used to load datasets from various file formats (like CSV), explore data, and prepare it for modeling.

2. train_test_split is a function from the scikit-learn library that splits datasets into training and testing sets. This is crucial for evaluating the performance of machine learning models, as it allows you to train a model on one set of data and test its performance on unseen data.

3. MinMaxScaler is a preprocessing tool from sci-kit-learn that scales the features of your dataset to a given range, typically [0, 1]. This normalization can be important for models that are sensitive to the scale of data, such as neural networks, helping them to converge more quickly and perform better.

4. Sequential is a class from the Keras library (now integrated with TensorFlow) used to create a linear stack of layers for neural networks. It's one of the simplest ways to build a model in Keras, allowing you to easily add layers one after the other.

5. Dense represents a densely connected neural network layer, commonly known as a fully connected layer. Each neuron in a Dense layer receives input from all neurons of the previous layer, making it a core building block for neural networks. It's used for a variety of tasks, including classification and regression.

6. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. Adam is popular due to its effective use of adaptive learning rates, making it more efficient for training deep learning models.

7. Matplotlib is a plotting library for Python and its numerical mathematics extension, NumPy. It provides an object-oriented API for embedding plots into applications. The `plt` submodule is commonly used for creating figures and plots, making it an essential tool for visualizing data

distributions, model performance metrics (like accuracy and loss over epochs), and more in machine learning projects.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import matplotlib.pyplot as plt
```

Loaded the data in, removed the first row, then transposed the data to prepare for matrix multiplication.

```
train_df_path = 'mnist_train.csv\mnist_train.csv'
train_targets_df_path = 'mnist_train_targets.csv'
test_df_path = 'mnist_test.csv\mnist_test.csv'
# Loaded data removing the first row from each CSV
train_df = pd.read_csv(train_df_path, skiprows=1,
header=None).T
train_targets_df = pd.read_csv(train_targets_df_path,
skiprows=1, header=None).T
test_df = pd.read_csv(test_df_path, skiprows=1, header=None).T
```

1. Preprocess the Data: It normalizes the training and, test datasets train_df and test_df, using MinMaxScaler to ensure features contribute equally to model training. The target labels, y_train are also prepared by flattening them into a 1D array.

2. Data Splitting: The normalized training data and labels are split into training and validation sets with train_test_split, allocating 20% of the data to validation. This step is crucial for assessing model performance on unseen data and mitigating overfitting.

```
# Build and Compile the Model
model = Sequential([
    Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer=Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

1. Building the Model: Construct a Sequential neural network with three layers: two dense layers with 128 and 64 neurons, respectively, using ReLU activation, and an output layer with 10 neurons using softmax activation. This structure is designed for classifying inputs into one of ten categories based on the model's input data dimensions.

2. Compiling the Model: The model is compiled with the Adam optimizer and uses sparse categorical cross-entropy as the loss function. It tracks accuracy as a performance metric, preparing the model for training on the dataset by defining how it should learn and be evaluated.

```
# Train the Model
model.fit(X_train, y_train, epochs=95, validation_data=(X_val,
y_val))

# Make Predictions
predictions = model.predict(X_test_scaled)
predicted_classes = predictions.argmax(axis=1)
```

1. Training the Model: Executes the training process for 95 epochs on the preprocessed training data, X_train, and y_train, using a separate validation dataset, X_val, and y_val, to monitor performance. This step iteratively adjusts the model's weights to minimize loss and improve accuracy on both training and validation data.

2. Making Predictions: Uses the trained model to predict the classes of the scaled test dataset, X_test_scaled. The prediction method outputs the likelihood of each class for each sample, and arg max is used to select the class with the highest probability as the final prediction for each sample.

```
# Prepare Submission File
submission_df = pd.DataFrame({'Id': range(1,
len(predicted_classes) + 1), 'Expected': predicted_classes})
submission_df.to_csv(r'C:\Users\zackl\Desktop\678\Project3\mnis
t_submission.csv', index=False)
```

Submitting the data

```
# Fit the model and save the history
history = model.fit(X_train, y_train, epochs=100,
validation_data=(X_val, y_val))

# Plotting the validation accuracy
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.title('Validation Accuracy vs. Epochs')
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.show()
```

```
# Function to create, train the model and return final
validation accuracy
def create_train_model(neurons_layer1, neurons_layer2, X_train,
y_train, X_val, y_val):
    model = Sequential([
        Dense(neurons_layer1, activation='relu',
input_shape=(X_train.shape[1],)),
        Dense(neurons_layer2, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=Adam(),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(X_train, y_train, epochs=95,
validation_data=(X_val, y_val), verbose=0)
    return history.history['val_accuracy'][-1]  # Return the
last validation accuracy
# Neuron configurations to test
neurons_config = [(128, 64), (256, 128), (512, 256), (1024,
512)]
val_accuracies = []
# Iterate over configurations, train models, and store final
validation accuracy
for neurons in neurons_config:
    val_acc = create_train_model(neurons[0], neurons[1],
X_train, y_train, X_val, y_val)
    val_accuracies.append(val_acc)
# Plot the validation accuracy for each configuration
plt.figure(figsize=(10, 6))
plt.plot([str(cfg) for cfg in neurons_config], val_accuracies,
marker='o', linestyle='-', color='b')
plt.title('Validation Accuracy for Different Neuron
Configurations')
```

```
plt.xlabel('Neuron Configuration (Layer1, Layer2)')
plt.ylabel('Final Validation Accuracy')
plt.grid(True)
plt.show()
```

Cross-validation of neurons. This was to test if the validation would increase or decrease based on the number of neurons.

Results:

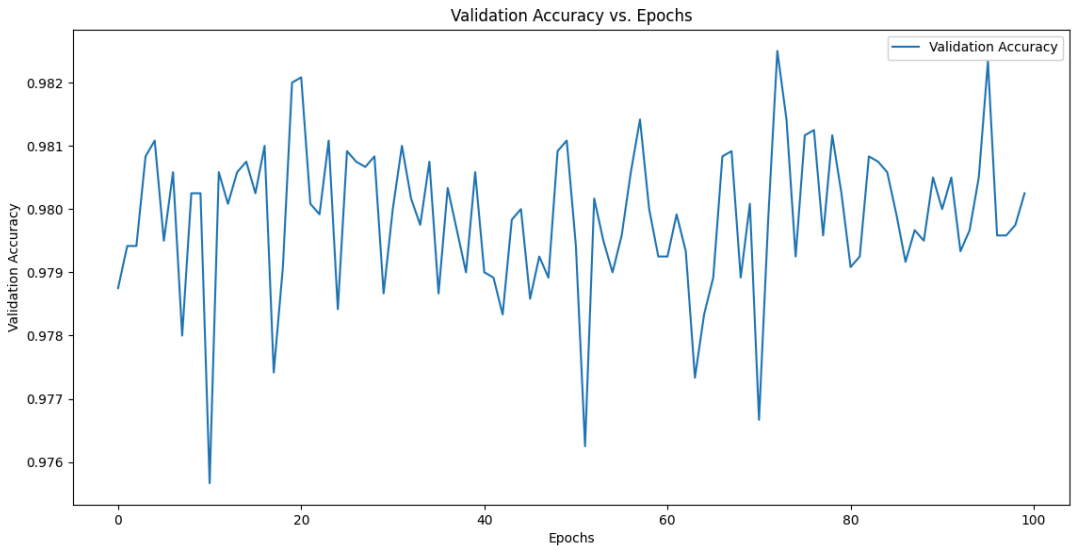

Figure 1: Epochs vs Validation Accuracy

My final score was 0.98166 accuracy using 95 epochs as noted in the graph.

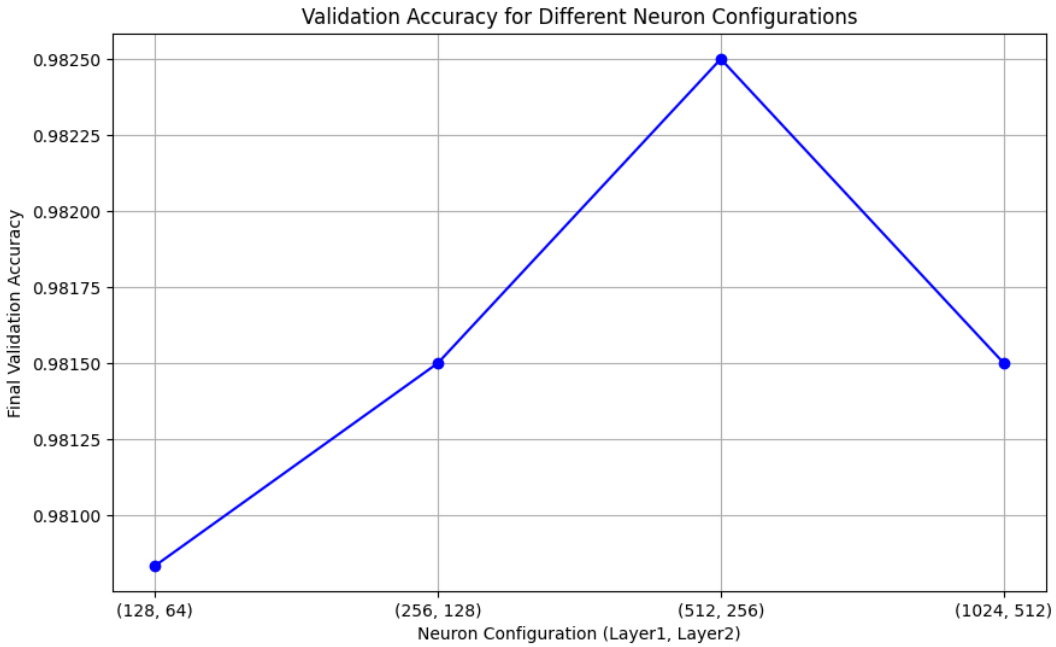

Figure 2: Cross Validation of Neuron Configuration

The Neuron configuration is laid out in Figure 2. I wanted to see if the number of neurons made a difference in the validation accuracy of the project or not and it appears like it hardly did.

Conclusion:

In conclusion, using a sequential, dense stack of neurons with ADAM optimization, the ideal number of epochs was found by graphing them out and finding the absolute max of validation accuracy. This has many strengths and close to no weaknesses. There are specific numbers in the test set to confuse the model, so this is about as accurate as you can possibly get without examining the data by hand. Increasing or decreasing the neurons hardly made a difference, so I chose the lowest possible configuration to reduce computing power.