Deep Learning and Natural Language Processing Project

Moria Grohar, Shirel Zecharia

March 1, 2024

1 Abstract

In this report, we present a study on implementing a neural network model for the classification of fashion items using the Fashion MNIST dataset.
We compare the performance of the neural network approach with traditional machine learning algorithms such as Logistic Regression, and Multi-Layer Perceptron (MLP).
The report discusses the project methodology, experimental results, and conclusions drawn from the comparison.

2 Introduction

This report explores the performance of various machine learning models for classifying fashion items from the well-known Fashion-MNIST dataset. This dataset consists of grayscale images of various clothing articles, each belonging to a specific category such as T-shirts, trousers, and shoes. The task of classifying these images is considered a classification problem.

The report delves into the implementation and evaluation of different models, including several models from sklearn, neural network from tensorflow and CNN.

The models were evaluated based on their loss function, with the aim of achieving the lowest possible loss to indicate better performance in classifying the fashion items.

The report compares and analyzes the results of each model, providing insights into their strengths and weaknesses for this specific classification task. It also discusses potential areas for further exploration and improvement.

3 Related Work and Required Background

3.1 Softmax Regression

Softmax regression, also known as multinomial logistic regression, is a generalization of logistic regression. While in logistic regression we assumed that the labels were binary: $y(i) \in 0, 1$, In Softmax regression we are able to handle multiple labels $y(i) \in 1, \ldots, K$ where k is the number of classes.

In softmax regression, the goal is to predict the probability that an input sample belongs to each possible class. This is achieved by applying the softmax function to the output of a linear model. The linear model computes a score for each class based on the input features,

and the softmax function converts these scores into probabilities.

Mathematically, given an input vector x, softmax regression computes the probability $P(y = i|x)$ that the input belongs to class i as follows:

$$P(y = i|x) = \frac{e^{x^T \cdot W_i + b_i}}{\sum_{j=1}^{k} e^{x^T \cdot W_j + b_j}}$$

Where:
y: target variable (class)
x: vector of input features
T: denotes the transpose applied to the vector
W_i: the weight vector associated with class i
b_i: the bias term associated with class i

Understanding The Process:

Linear Transformation: For each class, it computes a linear combination of the input features using class-specific weights. This generates a score for each class, indicating its "fit" for the data point.

Softmax Function: This function takes the vector of scores and transforms them into probabilities. It ensures:

- Non-negativity: All probabilities are positive (between 0 and 1).
- Normalization: The sum of all probabilities for a single data point is always 1, representing a valid probability distribution.

The softmax function is often used as the final activation layer in multi-class neural networks as we will discuss later in the article.

3.2 Activation Functions

Activation functions are mathematical functions used to determine the output of a neuron, influencing how information flows through the network and ultimately shaping its learning and decision-making capabilities.

Traditional neural networks without activation functions would simply perform linear transformations on the input data. This limitation restricts them from learning complex patterns and relationships in the data, hindering their ability to perform tasks like image recognition or natural language processing.

A neuron in a neural network receives multiple inputs, each carrying a specific value. These inputs are combined through weighted connections and then passed through an activation function. The function acts like a filter, transforming the combined input into a single output value. This output value then becomes the basis for the neuron's activation, influencing the signals it transmits to other neurons in the network.

The selection of an appropriate activation function depends on the specific task and network architecture. While ReLU and Leaky ReLU are popular choices due to their efficiency, other functions like sigmoid and tanh might be suitable for specific scenarios like output normalization or recurrent neural networks.

Popular Activation Functions:

- Rectified Linear Unit: ReLU (Rectified Linear Unit) activation functions serve as the non-linear activation function of choice in most CNN architectures. ReLU introduces non-linearity into the network, enabling it to learn complex mappings between the input and output. Mathematically, ReLU simply outputs the input if it is positive and zero otherwise, which simplifies computations and accelerates convergence during training.
- Leaky Rectified Linear Unit: Leaky ReLU is a variant of the ReLU activation function that addresses the "dying ReLU" problem, where neurons may become inactive and cease to update their weights during training. Unlike ReLU, which sets negative values to zero, Leaky ReLU allows a small, non-zero gradient for negative inputs. This modification ensures that neurons remain active and continue to contribute to the learning process, especially in deeper networks where vanishing gradients are more prevalent.
- Sigmoid: This function squishes the input values between 0 and 1, mimicking the firing behavior of biological neurons. However, its vanishing gradients can hinder learning in deep networks.
- Tanh (Hyperbolic tangent): Similar to sigmoid, tanh maps the input values to a range of -1 to 1. It offers slightly better gradients than sigmoid but suffers from similar limitations.

3.3 Recurrent Neural Networks

3.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specific type of artificial neural network architecture particularly well-suited for analyzing image data. They excel at tasks like image classification, object detection, and image segmentation. Unlike traditional neural networks, CNNs incorporate an additional layer type called a convolutional layer to extract features from the input data. The essence of a CNN lies in its ability to automatically learn spatial hierarchies of features from raw data. These networks consist of several layers, typically including convolutional layers, pooling layers, and fully connected layers.

3.4.1 Convolutional Layers

Convolutional layers are the cornerstone of CNNs, responsible for learning and detecting patterns within the input data. They employ mathematical operations known as convolutions to extract features such as edges, textures, and other distinctive attributes from the input images. Mathematically, convolution involves sliding a filter or a kernel over the input

data, performing element-wise multiplications, and summing the results to produce feature maps.

3.4.2 Pooling Layers

Pooling layers reduce the spatial dimensions of feature maps generated by convolutional layers while retaining important information. Common pooling techniques include max pooling and average pooling, where the former selects the maximum value from a region of the feature map, while the latter computes the average. Pooling helps in achieving translation invariance and reducing the computational burden by downsampling the feature maps.

3.4.3 Fully Connected Layers

These layers function similarly to traditional neural networks, connecting all neurons in one layer to all neurons in the next. They process the extracted features and learn complex relationships to form the final output, such as a class prediction or a set of bounding boxes for object detection.

3.5 Optimizers

Optimizers are used for updating the model's parameters to minimize the loss function during the training process, which ultimately leads to better performance of the model.

3.5.1 Stochastic Gradient Descent (SGD)

The SGD algorithm takes a single data point (or a small batch) at a time, calculates the gradient (the direction of steepest descent in the loss function), and updates the model's parameters in the opposite direction by a small learning rate. This technique is computationally inexpensive, making it suitable for large datasets. But on the other hand it may require many iterations to reach an optimal solution, especially for complex models and datasets. The update rule for SGD can be represented as:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla L(\theta_t)$$

3.5.2 RMSprop (Root Mean Square Propagation)

RMSprop addresses a limitation of SGD, where frequent updates based on noisy gradients can lead to slow convergence. Its key Feature is that it incorporates an exponential moving average of the squared gradients for each parameter. This averages the recent updates and reduces the impact of large, infrequent gradients, leading to smoother convergence. This technique has an increased computational cost compared to SGD and RMSprop.This technique has an increased computational cost compared to SGD.

$$s_{t+1} = \beta \cdot s_t + (1 - \beta) \cdot (\nabla L(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_{t+1}} + \epsilon} \cdot \nabla L(\theta_t)$$

3.5.3 Adam (Adaptive Moment Estimation)

The Adam optimizer combines ideas from SGD, RMSprop, and momentum, aiming to adapt the learning rate for each parameter individually and accelerate convergence. It maintains exponential moving averages of gradients (similar to RMSprop) and squared gradients. It incorporates momentum to account for the history of past updates, addressing the issue of getting stuck in local minima. It adaptively adjusts the learning rate for each parameter based on these estimates, aiming for optimal updates. This technique has an increased computational cost compared to SGD and RMSprop.

$$
\begin{aligned}
m_{t+1} &= \beta_1 \cdot m_t + (1 - \beta_1) \cdot \nabla L(\theta_t) \\
v_{t+1} &= \beta_2 \cdot v_t + (1 - \beta_2) \cdot (\nabla L(\theta_t))^2 \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1 - \beta_1^{t+1}} \\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1 - \beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \cdot \hat{m}_{t+1}
\end{aligned}
$$

4 Previous Attempts

4.1 First Attempts with different data

Model Selection and Evaluation:

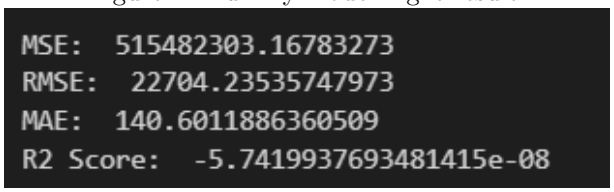
The project explored the performance of various models for predicting flight ticket prices in a dataset containing information on bookings between major Indian cities. The target variable was the "Price" column, indicating a continuous value. Therefore, the problem was classified as a regression task.

The data was split into train and test sets for model evaluation. Several models were implemented and compared based on their performance in reducing the mean squared error (MSE), which served as the loss function.

Model Performances:

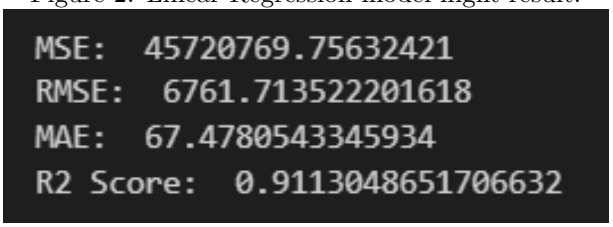
Dummy Model: This baseline model adopted a simple mean strategy, resulting in high loss values.

Figure 1: Dummy model flight result:



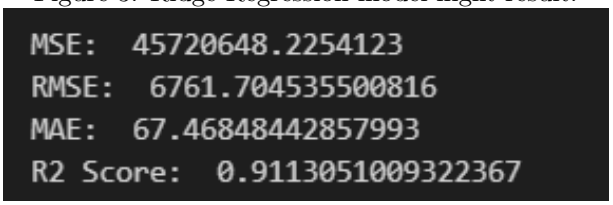
Linear Regression: Compared to the dummy model, linear regression showed improvement but still yielded significant loss.

Figure 2: Linear Regression model flight result:



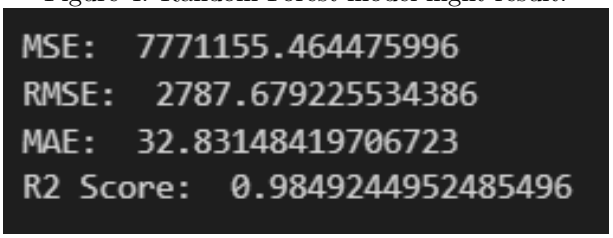
Ridge Regression: A slight reduction in loss compared to linear regression was observed, but the values remained high.

Figure 3: Ridge Regression model flight result:



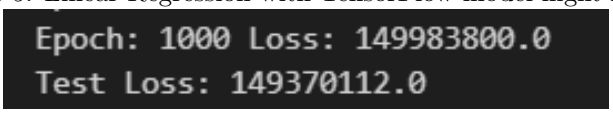
Random Forest: This model achieved a more significant reduction in loss compared to prior models, although the value remained high.

Figure 4: Random Forest model flight result:



Linear Regression with TensorFlow: Implementing linear regression using TensorFlow did not improve performance and even resulted in higher loss compared to the basic implementation.

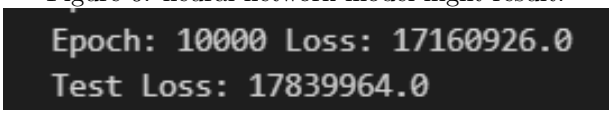
Figure 5: Linear Regression with TensorFlow model flight result:



Neural Network Exploration:

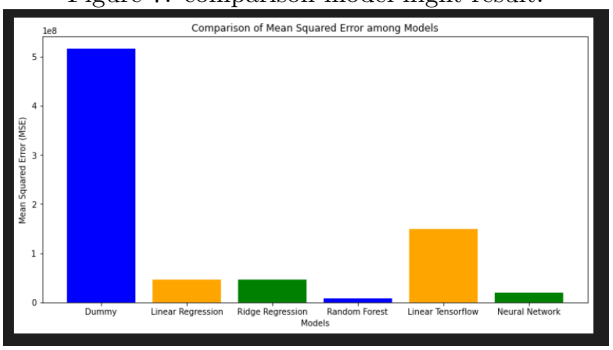
A neural network model with 2 hidden layers and 80 hidden units was constructed using TensorFlow. Hyperparameter tuning involved adjusting the learning rate and number of epochs, ultimately leading to the configuration with the lowest observed loss.

Figure 6: neural network model flight result:



Comparison and Conclusion:

Figure 7: comparison model flight result:



While the neural network achieved competitive results compared to simpler models after hyperparameter adjustments, the random forest remained the best performing model based on the achieved loss values.

4.2 Second Attempts with Fashion-MNIST data (our data)

Model Selection and Evaluation:

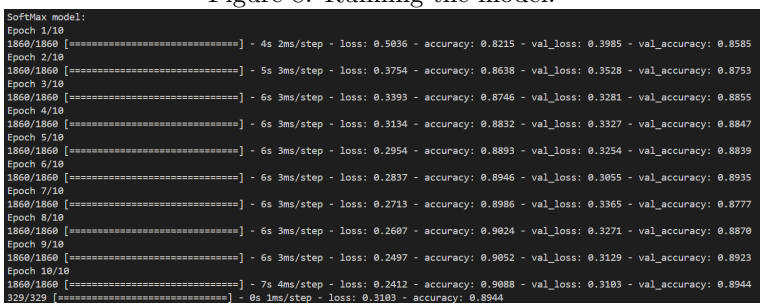
The project explored the performance of various models for classification of a picture of cloth to a various category. Therefore, the problem was classified as a classification task.

The data was split into train and test sets for model evaluation, in some models also to validation. Several models were implemented and compared based on their performance in reducing the loss function.

Model Performances:

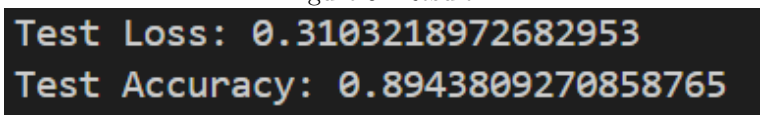
Soft max: a function that turns a vector of K real values into a vector of K real values that sum to 1. In an effort to optimize the performance of the model's performance, various hyperparameters were explored. The number of epochs controlling the training iterations was adjusted. Different optimizers responsible for updating the model weights were investigated. The split of data for training, validation and testing was also investigated. The aim of this approach was to achieve the best possible result with the smallest loss function and the highest accuracy. The accuracy calculation process

Figure 8: Running the model:



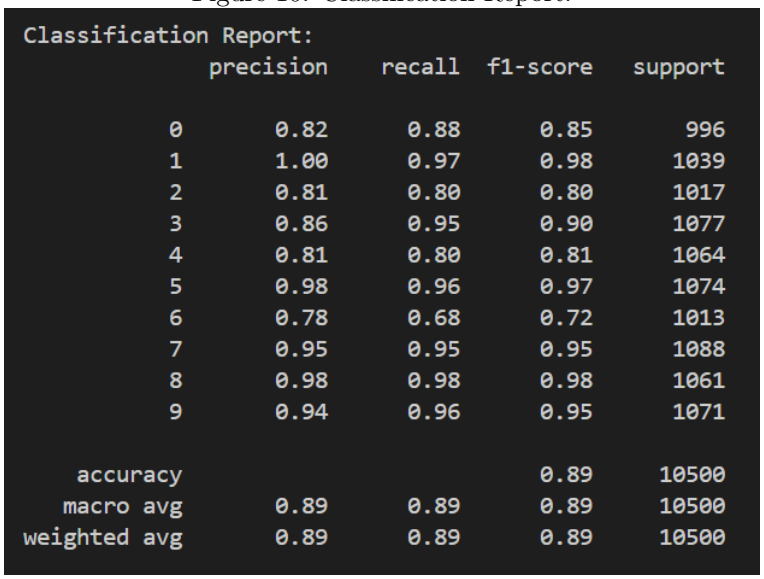
The result of the run:

Figure 9: Result:



The classification report in concentrated form:

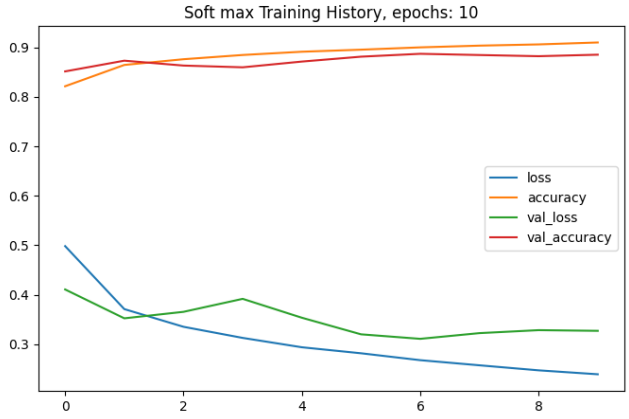
Figure 10: Classification Report:



	precision	recall	f1-score	support
0	0.82	0.88	0.85	996
1	1.00	0.97	0.98	1039
2	0.81	0.80	0.80	1017
3	0.86	0.95	0.90	1077
4	0.81	0.80	0.81	1064
5	0.98	0.96	0.97	1074
6	0.78	0.68	0.72	1013
7	0.95	0.95	0.95	1088
8	0.98	0.98	0.98	1061
9	0.94	0.96	0.95	1071
accuracy			0.89	10500
macro avg	0.89	0.89	0.89	10500
weighted avg	0.89	0.89	0.89	10500

All the data of the epochs can be shown in a graph that shows the loss function and the accuracy in every run.

Figure 11: Soft max training history:



CNN: a function that turns a vector of K real values into a vector of K real values that sum to 1.

Figure 12: Running the model:

```
CNN model:
Epoch 1/8
1860/1860 [==============================] - 41s 21ms/step - loss: 0.4501 - accuracy: 0.8406 - val_loss: 0.3607 - val_accuracy: 0.8690
Epoch 2/8
1860/1860 [==============================] - 43s 23ms/step - loss: 0.3181 - accuracy: 0.8864 - val_loss: 0.4119 - val_accuracy: 0.8583
Epoch 3/8
1860/1860 [==============================] - 54s 29ms/step - loss: 0.2777 - accuracy: 0.9000 - val_loss: 0.3165 - val_accuracy: 0.8833
Epoch 4/8
1860/1860 [==============================] - 60s 32ms/step - loss: 0.2489 - accuracy: 0.9088 - val_loss: 0.2881 - val_accuracy: 0.8993
Epoch 5/8
1860/1860 [==============================] - 67s 36ms/step - loss: 0.2285 - accuracy: 0.9160 - val_loss: 0.2519 - val_accuracy: 0.9069
Epoch 6/8
1860/1860 [==============================] - 74s 40ms/step - loss: 0.2093 - accuracy: 0.9222 - val_loss: 0.2704 - val_accuracy: 0.9014
Epoch 7/8
1860/1860 [==============================] - 60s 32ms/step - loss: 0.1929 - accuracy: 0.9279 - val_loss: 0.2730 - val_accuracy: 0.9056
Epoch 8/8
1860/1860 [==============================] - 69s 37ms/step - loss: 0.1827 - accuracy: 0.9302 - val_loss: 0.3491 - val_accuracy: 0.8859
```

The result of the run:

Figure 13: Result:

```
Test Loss: 0.3103218972682953
Test Accuracy: 0.8943809270858765
```

The classification report in concentrated form:

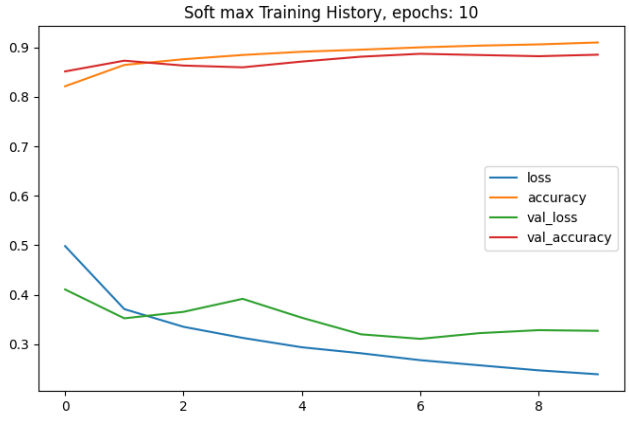
Figure 14: Classification Report:

Classification Report:

	precision	recall	f1-score	support
0	0.82	0.88	0.85	996
1	1.00	0.97	0.98	1039
2	0.81	0.80	0.80	1017
3	0.86	0.95	0.90	1077
4	0.81	0.80	0.81	1064
5	0.98	0.96	0.97	1074
6	0.78	0.68	0.72	1013
7	0.95	0.95	0.95	1088
8	0.98	0.98	0.98	1061
9	0.94	0.96	0.95	1071
accuracy			0.89	10500
macro avg	0.89	0.89	0.89	10500
weighted avg	0.89	0.89	0.89	10500

All the data of the epochs can be shown in a graph that shows the loss function and the accuracy in every run.

Figure 15: Soft max training history:



Neural Network Exploration:

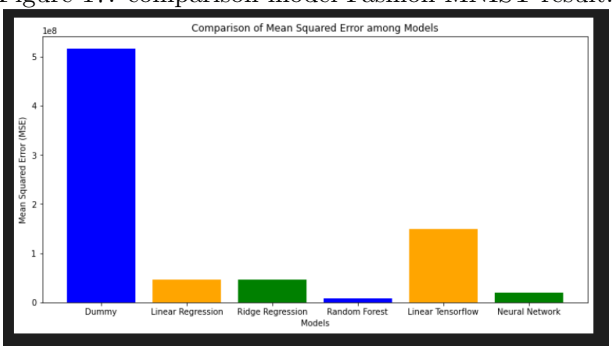
A neural network model with 2 hidden layers and 80 hidden units was constructed using TensorFlow. Hyperparameter tuning involved adjusting the learning rate and number of epochs, ultimately leading to the configuration with the lowest observed loss.

Figure 16: neural network model Fashion MNIST result:

```
Epoch: 10000 Loss: 17160926.0
Test Loss: 17839964.0
```

Comparison and Conclusion:

Figure 17: comparison model Fashion MNIST result:



While the neural network achieved competitive results compared to simpler models after hyperparameter adjustments, the random forest remained the best performing model based on the achieved loss values.

5 Project Description

Your project description text here.

6 Simulation Results

6.1 Softmax Regression Model

Figure 18: Result:

```
Test Loss: 0.3103218972682953
Test Accuracy: 0.8943809270858765
```

6.2 RNN Model

6.3 Neural Network Model

7 Code

7.1 CNN Model

The CNN architecture defined in this function follows a common pattern used for image classification tasks. It starts with convolutional layers that learn low-level features such as edges and textures. As the network deepens, subsequent layers learn increasingly complex and abstract features. Max pooling layers help in reducing spatial dimensions while retaining important features. Finally, the flattened feature vectors are fed into fully connected layers for classification. The use of ReLU activation functions helps in introducing non-linearity, and the softmax activation in the output layer enables the model to output class probabilities, making it suitable for multi-class classification tasks. Overall, this CNN architecture efficiently learns hierarchical representations from image data, enabling accurate classification.

Listing 1: CNN Functions code example

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
layers, models = tf.keras.layers, tf.keras.models
BatchNormalization, Dropout, LeakyReLU = tf.keras.layers.BatchNormalization, tf.keras

def create_cnn_model(input_shape, num_classes):

    """
    Description:
    This function constructs a CNN model using the Keras Sequential API.
    The model architecture consists of several layers:

    Convolutional Layers: These layers apply a convolution operation to the input ima
    extracting various features through filters.
    The activation function used is ReLU,
    which introduces non-linearity into the model, enabling it to learn complex patte

    Batch Normalization: This technique is used to improve the training stability and
    by normalizing the inputs of each layer.

    MaxPooling Layers: Max pooling reduces the spatial dimensions of the feature maps
    effectively downsampling the input.
    It retains the most important features while reducing computational
    complexity and preventing overfitting.

    Flatten Layer: This layer flattens the 2D feature maps into a 1D vector,
    preparing the data for input into a fully connected neural network.

    Dense Layers: Fully connected layers that perform classification based on the lea
    The first dense layer has 64 units with ReLU activation, enabling the network to
    patterns in the flattened feature vectors.
```

```
    The final dense layer has units equal to the number of classes
    in the dataset, with softmax activation, which outputs probabilities for each cla
    indicating the likelihood of the input image belonging to each class.

    Dropout: This layer was added to prevent overfitting.
    It works by randomly setting a fraction of input units to zero during training,
    which helps to prevent the network from relying too much on specific neurons
    and encourages it to learn more robust features.
    We've defined a Dropout layer with a dropout rate of 0.2.
    This means during training, 20% of the input units to the Dropout layer will be r

    Parameters:
    - input_shape: Tuple specifying the shape of the input images (height, width, ch
    - num_classes: Integer indicating the number of classes in the classification tas

    notes:
        - I was experiencing with ReLU and Leaky ReLU and the results were slightly
        - Deepening the layers more with MaxPooling2D is impossible because the input

    """
    # activation = 'relu'
    activation = LeakyReLU(alpha=0.1)
    dropout_rate = 0.3

    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation=activation, input_shape=input_shape),
# 32 filters of size 3x3
        BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=activation),
        BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation=activation),
        BatchNormalization(),
        layers.Flatten(),
        layers.Dense(64, activation=activation),
        Dropout(dropout_rate),
        layers.Dense(num_classes, activation='softmax')
    ])

    return model


def train_cnn_model(X_train, y_train, X_test, y_test, num_classes):
    """
    notes: I was experiencing with different parameters:
    - number of epochs: 8, 10, 12, 16
      results were best with 8
```

```
- optimizers: Adam, SGD, RMSprop
  results were best with Adam
- loss functions:

"""
# Reshape the input data for CNN
X_train = np.array(X_train).reshape(-1, 28, 28, 1)
X_test = np.array(X_test).reshape(-1, 28, 28, 1)

# Create the model
model = create_cnn_model((28, 28, 1), num_classes)  # grayscale image of size 28x

# Compile the model
# optimizer = tf.keras.optimizers.legacy.RMSprop(learning_rate=0.001, momentum=0
optimizer = 'adam'
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
epochs = 8
model_history = model.fit(X_train, y_train, epochs=epochs, batch_size=32, validat

# Display the training history
pd.DataFrame(model_history.history).plot(figsize=(8, 5))
plt.title('CNN Training History, epochs: ' + str(epochs))
plt.show()
plt.savefig("imgFolder/CNN_fig")

return model
```

8 Conclusion

Your conclusion text here.