ملیکا محمدی فخار
۹۹۵۲۲۰۸۶

1.

1.1.

1.1.1.

Keras Tuner is a hyperparameter optimization library for Keras. Hyperparameter tuning is a crucial step in the process of training machine learning models, as it involves finding the optimal set of hyperparameters that result in the best performance of the model. Keras Tuner provides a simple and flexible interface for performing hyperparameter tuning. It allows you to define a search space of hyperparameters and efficiently explores this space to find the combination that maximizes the performance of your model.

1.1.2.

- Define the CNN Model:

defining the CNN model architecture using Keras. Specify the convolutional layers, pooling layers, fully connected layers, and other relevant components.

- Create a Tuner Function:

Define a function that creates and returns the CNN model based on hyperparameters. This function will take hyperparameters as arguments and construct the corresponding model architecture.

- Define the Search Space:

Specify the search space for hyperparameters. For a CNN, this might include parameters such as the number of convolutional layers, filter sizes, kernel sizes, pool sizes, dropout rates, and learning rates.

- Select a Tuner Class:

Choose a tuner class that suits your optimization needs. For CNNs, tuners like RandomSearch or Hyperband are commonly used. The choice of tuner depends on factors like the size of the search space and computational resources available.

- Objective Function:

Define an objective function that trains and evaluates the CNN model using the specified hyperparameters. The objective function should return a metric that reflects the performance of the model on the validation set, such as accuracy or loss.

- Run the Tuner:

Use the selected tuner to search the hyperparameter space and optimize the CNN model. During the tuning process, the tuner will train and evaluate different CNN configurations, updating the hyperparameters to maximize the specified objective.

- Evaluate and Analyze Results:

After the tuning process is complete, examine the results to identify the best set of hyperparameters that optimize the CNN for your classification task. Keras Tuner provides tools for visualizing and analyzing the tuning results, allowing you to understand the impact of different hyperparameter configurations on model performance.

- Refine and Iterate:

Based on the initial tuning results, you may choose to refine the search space and run additional trials to further optimize the CNN. Iterate this process until you achieve satisfactory performance on your classification task.

1.1.3.

- RandomSearch: Randomly samples hyperparameter combinations from the specified search space. It's a simple and computationally efficient tuner.
- Hyperband: Uses a bracketing strategy to concurrently explore multiple configurations in a hyperparameter space, eliminating poor-performing configurations early on.

- BayesianOptimization: Utilizes Bayesian optimization to model the probabilistic relationship between hyperparameters and model performance. It makes informed decisions about where to explore next in the hyperparameter space.

- GridSearch: Exhaustively searches through all possible combinations of hyperparameters within the defined search space.

- TPE (Tree-structured Parzen Estimators): Uses a probabilistic model to guide the search by iteratively selecting hyperparameter configurations that have higher expected improvement.

- Sklearn: Integrates with scikit-learn's GridSearchCV and RandomizedSearchCV, allowing you to use Keras models within scikit-learn's hyperparameter tuning framework.

I used RandomSearch and Here are some considerations for choosing RandomSearch:

Exploration of the Search Space:
RandomSearch is effective for exploring a wide range of hyperparameter configurations randomly. It provides a good initial exploration of the hyperparameter space without making assumptions about the underlying structure.

Computational Efficiency:
Random search is computationally efficient compared to some other methods like grid search or exhaustive search. It allows you to perform a reasonably thorough exploration of the hyperparameter space with a limited number of trials.

Simple Implementation:
The implementation of RandomSearch is straightforward, making it easy to use for users who are new to hyperparameter tuning or when a quick initial exploration is needed.

No Assumptions about Search Space:
RandomSearch doesn't assume any specific structure or relationships within the hyperparameter space. This lack of assumptions can be beneficial when the behavior of the optimization landscape is not well understood.

Suitable for High-Dimensional Spaces:
Random search can be more robust in high-dimensional search spaces where exhaustive search becomes impractical.

1.2.

1.2.1.

The MNIST dataset is a widely used dataset in the field of machine learning and computer vision. The dataset consists of a collection of 28x28 pixel grayscale images of handwritten digits (0 through 9) along with their corresponding labels, indicating the digit that each image represents.

1.2.2.

Dataset Loading and Preprocessing:
Start by loading the MNIST dataset, which consists of handwritten digit images. Preprocess the data by normalizing pixel values to a range between 0 and 1. Additionally, reshape the images to the required input shape (28x28 pixels for MNIST) and one-hot encode the labels if necessary.

Define a CNN Architecture:
Design a convolutional neural network architecture suitable for image classification. A typical CNN architecture for MNIST might include convolutional layers with filters, activation functions like ReLU, max-pooling layers for down-sampling, and fully connected layers for classification. Parameterize the architecture by using hyperparameters that can be optimized.

Hyperparameter Tuning with Keras Tuner:
Utilize Keras Tuner to search for the optimal hyperparameters for your CNN. Define a search space for hyperparameters such as the number of filters, kernel sizes, learning rate, and dropout rates. Select a tuner (e.g., RandomSearch or Hyperband) and specify an objective metric to optimize (e.g., validation accuracy). Run the tuner to perform the hyperparameter search over a predefined number of trials.

Training and Validation:
Train the CNN using the training set while validating its performance on a separate validation set. During each trial, the hyperparameters specified by the tuner are used to train a model, and the performance on the validation set is recorded.

Evaluate and Select the Best Model:
After the hyperparameter search is complete, analyze the results to identify the set of hyperparameters that resulted in the best-performing model. Retrieve the best model and evaluate its performance on a held-out test set to ensure generalization to new, unseen data.

Fine-Tuning (Optional):
If desired, perform additional fine-tuning of hyperparameters or architecture based on insights gained from the initial tuning process. Iterate through steps 3 to 5 as needed.

Final Model Deployment:
Once satisfied with the performance of the optimized model, deploy it for making predictions on new, unseen data. The deployed model can be used for classifying handwritten digits into the corresponding categories (0 through 9) with high accuracy.

1.2.3.

Pooling:

Spatial Hierarchical Representation: Pooling helps create a spatial hierarchical representation, emphasizing the most relevant features. It retains the most important information while discarding less important details, making the network more robust to variations in input position and scale.

Reduction in Computational Complexity: Down-sampling reduces the spatial dimensions of the feature maps, which can lead to a reduction in the number of parameters and computations in subsequent layers, making the network computationally more efficient.

Dropout:

Improved Generalization: Dropout introduces a form of model averaging during training. By randomly dropping units, the network learns to rely on different subsets of features for making predictions, leading to improved generalization on unseen data.

Reduction of Co-Adaptation: Dropout helps prevent co-adaptation of hidden units, where some units become overly dependent on the presence of others. This encourages each unit to contribute independently to the learning process, making the network more robust.

Implicit Ensemble Learning: During training, each iteration involves a different subset of neurons being dropped out, effectively training multiple sub-networks. At test time, dropout is typically turned off, but the learned weights are scaled, resulting in an implicit ensemble of the trained sub-networks.

Combined Effect:

Regularization: Both pooling and dropout contribute to regularization, helping prevent overfitting by promoting simpler and more general representations.

Robustness: Pooling enhances spatial robustness by capturing essential features, while dropout promotes robustness in the presence of noisy or irrelevant information.

1.3.

1.3.1.

First I Loaded the MNIST dataset, containing handwritten digits and their labels, Selected a subset of the training data for demonstration purposes, Normalized the pixel values to the range [0, 1] and add a channel dimension and converted labels to one-hot encoding for categorical classification.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

size = -10000
x_train = x_train[:size]
x_val = x_train[size:]
y_train = y_train[:size]
y_val = y_train[size:]

x_train = np.expand_dims(x_train, -1).astype("float32") / 255.0
x_val = np.expand_dims(x_val, -1).astype("float32") / 255.0
x_test = np.expand_dims(x_test, -1).astype("float32") / 255.0

num_classes = 10
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_val = tf.keras.utils.to_categorical(y_val, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

then I designed following model using question information:

```python
def design_model(hp):
    # Create a Sequential model
    model = tf.keras.Sequential()

    # Convolutional layers
    for i in range(hp.Int("num_layers_conv", 1, 5)):
        # Add a convolutional layer with tunable number of filters and specified parameters
        model.add(
            layers.Conv2D(
                filters=hp.Int(f"filters{i}", min_value=32, max_value=256, step=32),
                kernel_size=(3, 3),
                activation='relu',
                padding='same'
            )
        )
        # Add a max pooling layer
        model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    # Flatten the output from convolutional layers
    model.add(layers.Flatten())

    # Dense layers
    for i in range(hp.Int("num_layers", 1, 5)):
        # Add a dense layer with tunable number of units and activation function
        model.add(
            layers.Dense(
                units=hp.Int(f"units_{i}", min_value=32, max_value=256, step=32),
                activation=hp.Choice("activation", ["relu", "tanh"]),
            )
        )

    # Add a dropout layer based on a boolean hyperparameter
    if hp.Boolean("dropout"):
        model.add(layers.Dropout(rate=0.25))

    # Output layer with softmax activation for classification
    model.add(layers.Dense(10, activation="softmax"))

    # Compile the model with a tunable learning rate
    learning_rate = hp.Float("lr", min_value=1e-4, max_value=1e-2, sampling="log")
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
```

then after defining a tuner object, the search goes on! you can see the results in below:

```
tuner = keras_tuner.RandomSearch(
    hypermodel=design_model,
    objective="val_accuracy",
    max_trials=3,
    executions_per_trial=2,
    overwrite=True,
    directory="directory_name",
    project_name="Project_name",
)
```

```
tuner.search(x_train, y_train, epochs=2, validation_data=(x_val, y_val))
```

```
Trial 3 Complete [00h 00m 43s]
val_accuracy: 0.9817500114440918

Best val_accuracy So Far: 0.9817500114440918
Total elapsed time: 00h 03m 00s
```

now it’s time to find the best model and train it on the dataset and evaluate it:

```
# Retrieve the top 2 models from the tuner
top_models = tuner.get_best_models(num_models=2)

# Select the best model from the top models
best_model = top_models[0]

# Build the best model using the input shape of the training data
best_model.build(input_shape=x_train.shape)

# Display a summary of the best model's architecture
print("Best Model Summary:")
best_model.summary()
```

```
Best Model Summary:
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (50000, 28, 28, 128)      1280

 max_pooling2d (MaxPooling2  (50000, 14, 14, 128)      0
 D)

 flatten (Flatten)           (50000, 25088)            0

 dense (Dense)               (50000, 96)               2408544

 dense_1 (Dense)             (50000, 256)              24832

 dense_2 (Dense)             (50000, 192)              49344

 dense_3 (Dense)             (50000, 32)               6176

 dropout (Dropout)           (50000, 32)               0

 dense_4 (Dense)             (50000, 10)               330
```

```python
# Train the best model on the dataset
best_model.fit(x_train, y_train, epochs=10, validation_split=0.2)

# Evaluate the best model on the test dataset
test_loss, test_accuracy = best_model.evaluate(x_test, y_test)

# Display the test loss and accuracy
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
```

```
Epoch 1/10
1250/1250 [==============================] - 7s 6ms/step - loss: 0.0404 - accuracy: 0.9887 - val_loss: 0.0758 - val_accuracy: 0.9817
Epoch 2/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.0388 - accuracy: 0.9889 - val_loss: 0.1295 - val_accuracy: 0.9683
Epoch 3/10
1250/1250 [==============================] - 7s 6ms/step - loss: 0.0343 - accuracy: 0.9909 - val_loss: 0.0789 - val_accuracy: 0.9788
Epoch 4/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.0305 - accuracy: 0.9921 - val_loss: 0.0778 - val_accuracy: 0.9806
Epoch 5/10
1250/1250 [==============================] - 8s 6ms/step - loss: 0.0270 - accuracy: 0.9926 - val_loss: 0.0869 - val_accuracy: 0.9814
Epoch 6/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.0302 - accuracy: 0.9918 - val_loss: 0.0983 - val_accuracy: 0.9782
Epoch 7/10
1250/1250 [==============================] - 7s 6ms/step - loss: 0.0278 - accuracy: 0.9923 - val_loss: 0.0996 - val_accuracy: 0.9782
Epoch 8/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.0256 - accuracy: 0.9930 - val_loss: 0.0859 - val_accuracy: 0.9818
Epoch 9/10
1250/1250 [==============================] - 8s 6ms/step - loss: 0.0230 - accuracy: 0.9933 - val_loss: 0.1019 - val_accuracy: 0.9766
Epoch 10/10
1250/1250 [==============================] - 7s 6ms/step - loss: 0.0206 - accuracy: 0.9949 - val_loss: 0.0857 - val_accuracy: 0.9832
313/313 [==============================] - 1s 3ms/step - loss: 0.0883 - accuracy: 0.9825
Test Loss: 0.08826574683189392
Test Accuracy: 0.9825000166893005
```

1.3.2.
here are some reasons why I used 3 * 3 kernels:
Smaller Receptive Field: A 3x3 kernel has a smaller receptive field compared to larger kernels. This allows the network to capture more local and detailed features in the input, which can be crucial for recognizing intricate patterns.

Parameter Efficiency: Using 3x3 kernels is parameter-efficient. They have fewer parameters compared to larger kernels (e.g., 5x5 or 7x7), which reduces the risk of overfitting and makes the network computationally more efficient.

Hierarchical Feature Learning: Stacking multiple 3x3 convolutional layers allows the network to learn hierarchical features. The first layer may capture simple edges and textures, while subsequent layers learn more complex and abstract features. This promotes a progressive abstraction of features through the network.

Compatibility with Pooling: The use of 3x3 kernels is often paired with max pooling layers. The combination of small kernels and pooling layers helps down-sample the spatial dimensions while retaining essential features, facilitating a more hierarchical and efficient representation of the input.

Computational Benefits:
The computational cost of convolutions with 3x3 kernels is often lower than that of larger kernels, making them more efficient for training and inference on hardware with limited resources.

1.3.3.
Pooling and dropout are two effective techniques in convolutional neural networks (CNNs) to mitigate overfitting, each addressing different aspects of the learning process. Pooling layers, such as max pooling, contribute to avoiding overfitting by introducing spatial hierarchies and reducing the dimensionality of feature maps. By down-sampling the spatial resolution, pooling layers focus on retaining essential information while discarding less relevant details. This spatial abstraction helps the network generalize better to unseen data and reduces sensitivity to small variations in input, contributing to a more robust model that is less prone to overfitting on the training set.

On the other hand, dropout serves as a regularization technique by randomly deactivating a fraction of neurons during training. This prevents the network from relying too heavily on specific neurons and encourages the learning of more robust and diverse features. Dropout acts as a form of ensemble learning, training multiple sub-networks during each epoch. As the network learns to adapt to various subsets of neurons being dropped out, it becomes less likely to memorize noise or outliers in the training data, resulting in improved generalization and reduced overfitting. By combining pooling and dropout in a CNN, practitioners create a model that not only captures hierarchical spatial representations efficiently but also learns diverse and robust features, leading to enhanced performance on both training and validation sets.

2.

3.

LSTMs are designed to capture sequential dependencies in data. If your data has a strong temporal structure, it might be beneficial to input it as a series rather than individual time points.
also the length of the sequence is an important factor. If you provide a new input every second, each input can be considered a sequence of its own. The challenge may arise if your sequences are too short for the LSTM to capture meaningful patterns. Longer sequences generally provide more context for the model to learn temporal dependencies.
i think this problem in the code caused negative r2 score.

4.

4-1. Definition and use cases

- Convolutional Neural Networks (CNNs):

Definition: CNNs are primarily designed for processing structured grid-like data, such as images. They consist of convolutional layers that apply filters to local input regions, allowing them to capture hierarchical patterns and spatial dependencies in the data. CNNs are particularly effective in image recognition, object detection, and computer vision tasks.

- Use Cases:
- Image classification: Identifying objects or scenes in images.
- Object detection: Locating and classifying multiple objects within an image.
- Image segmentation: Assigning a label to each pixel in an image.
- Facial recognition: Identifying and verifying individuals based on facial features.

- Recurrent Neural Networks (RNNs):

Definition: RNNs are designed to handle sequential data by maintaining hidden states that capture information about previous inputs. This architecture enables RNNs to model temporal dependencies in the data. RNNs are suitable for tasks where the order of the input sequence matters, such as natural language processing, speech recognition, and time-series prediction.

- Use Cases:
- Natural Language Processing (NLP): Language modeling, text generation, sentiment analysis.
- Speech Recognition: Converting spoken language into text.
- Time-Series Prediction: Forecasting future values based on historical data.

• Handwriting recognition: Recognizing and interpreting handwritten text.

• CNNs are better suited for:
Tasks involving grid-like data, such as images.
Capturing spatial dependencies and hierarchical patterns.
Tasks where translation invariance is essential (e.g., image recognition).

• RNNs are better suited for:
Sequential data where the order of input elements matters.
Tasks involving natural language processing, speech recognition, and time-series prediction.
Problems with variable-length input sequences.

4.2. Number of Parameters:

• CNNs:
Typically, CNNs have a fixed number of parameters per filter, which are shared across the entire input.
The number of parameters in a CNN is determined by the size of the filters, the depth of the network, and the number of filters in each layer.
CNNs are generally more parameter-efficient than fully connected networks when dealing with grid-like data such as images.

• RNNs:
The number of parameters in RNNs depends on the size of the hidden state and the number of recurrent layers.
As the RNN processes sequences, the number of parameters does not depend on the sequence length but rather on the recurrent structure.
RNNs may have more parameters compared to CNNs for tasks involving sequential data, especially when dealing with long sequences.

Parallelism:
• CNNs:
CNNs are inherently parallelizable because convolutions can be applied independently to different regions of the input.
Modern hardware, such as Graphics Processing Units (GPUs), is well-suited for parallel processing, and CNNs can take advantage of this to speed up training and inference.
Batch processing is also efficient in CNNs, allowing multiple inputs to be processed simultaneously.

• RNNs:
RNNs are more challenging to parallelize because each step in the sequence depends on the previous one.
While some parallelization is possible (e.g., processing different sequences simultaneously), the inherently sequential nature of RNNs limits the degree of parallelism.
Techniques like mini-batch processing and model parallelism can be used to introduce some parallelization in RNN training.

5.

Conv (padding = same):

$$params = filters(kernelSize \,.\, kernelSize \,.\, inputChannelSize + 1)$$
$$outputWidth = inputWidth$$
$$outputChannels = filters$$

Dilated Conv:

$$params = filters(kernelSize \,.\, kernelSize \,.\, inputChannelSize + 1)$$
$$outputWidth = ((inputWidth + 2.padding_{d}elationRate \,.\, (kernelSize - 1) - 1) \div stride) + 1$$
$$outputChannels = filters$$

MaxPool:

no params!

$$outputWidth = ((inputWidth - poolSize) \div stride) + 1$$

(rows are layers)
pytorch:

output dimensions	params
256, 256, 64	64 (3 . 3 . 3 + 1)
124, 124, 32	32 (5 . 5 . 64 + 1)
62, 62, 32	0
62, 62, 128	128 (3 . 3 . 32 + 1)
24, 24, 64	64 (5 . 5 . 128 + 1)
12, 12, 64	0
12, 12, 256	256 (3 . 3 . 64 + 1)
-9, -9, 28	128 (5 . 5 . 256 + 1)
-4, -4, 128	0

if we use tensorflow(stride doesn’t consider with dilation):

output dimensions	params
256, 256, 64	64 (3 . 3 . 3 + 1)
248, 248, 32	32 (5 . 5 . 64 + 1)
124, 124, 32	0
124, 124, 128	128 (3 . 3 . 32 + 1)
108, 108, 64	64 (5 . 5 . 128 + 1)
54, 54, 64	0
54, 54, 256	256 (3 . 3 . 64 + 1)
22, 22, 128	128 (5 . 5 . 256 + 1)
11, 11, 128	0

6.

6.1.

- The statement is false. The primary benefits of batch normalization include improved convergence during training and the ability to use higher learning rates. Batch normalization does not specifically speed up the processing of a single batch. Instead, it helps the entire training process by making it more stable and faster.
- Correct. Batch normalization adjusts the input values for each layer (which essentially correspond to the output from the preceding layer) in a mini-batch. This adjustment involves subtracting the mean and dividing by the standard deviation.
- False. Batch normalization may have an impact on the scale and distribution of weights indirectly due to the normalization of inputs, but its main focus is on the normalization of activations, not the explicit control of weights being close to zero.

6.2.

code is attached.

6.3.

A small constant, denoted as ϵ, is added to the variance estimate to prevent division by zero. This inclusion ensures that the denominator is never zero, contributing to the stabilization of the normalization process.

6.4.

Normalization Issues: Statistics calculated over a single sample may not accurately represent the data distribution, resulting in less effective normalization.

Instability during Training: Sensitivity to individual sample characteristics can lead to training instability, with small input changes causing large normalized value changes.

Lack of Statistical Robustness: Batch normalization relies on larger batch sizes for statistical stability, and using one contradicts this principle.

Loss of Regularization Effect: The regularization effect provided by batch normalization is diminished with a batch size of one, as there is no variation in statistics across samples.

Performance and Speed: Larger batch sizes are often more efficient, and using one can result in slower training times due to reduced parallelism.

Test-time Inconsistency: Accumulated statistics from a single-sample batch may not accurately represent the population, leading to discrepancies between training and testing. Overall, it is generally recommended to use larger batch sizes for more effective batch normalization.

6.5.
For a fully connected layer with 10 inputs and 20 outputs, there are 10×20=200 weights. Additionally, there are 20 biases corresponding to the 20 output nodes.
So, the total number of adjustable parameters for weights and biases without batch normalization is 220.

With the incorporation of batch normalization, every output of the fully connected layer will be linked to two extra parameters (gamma and beta).
So trainable parameters, considering batch normalization, will be 220 + 2 × 20 = 260.

7. first i downloaded the dataset:

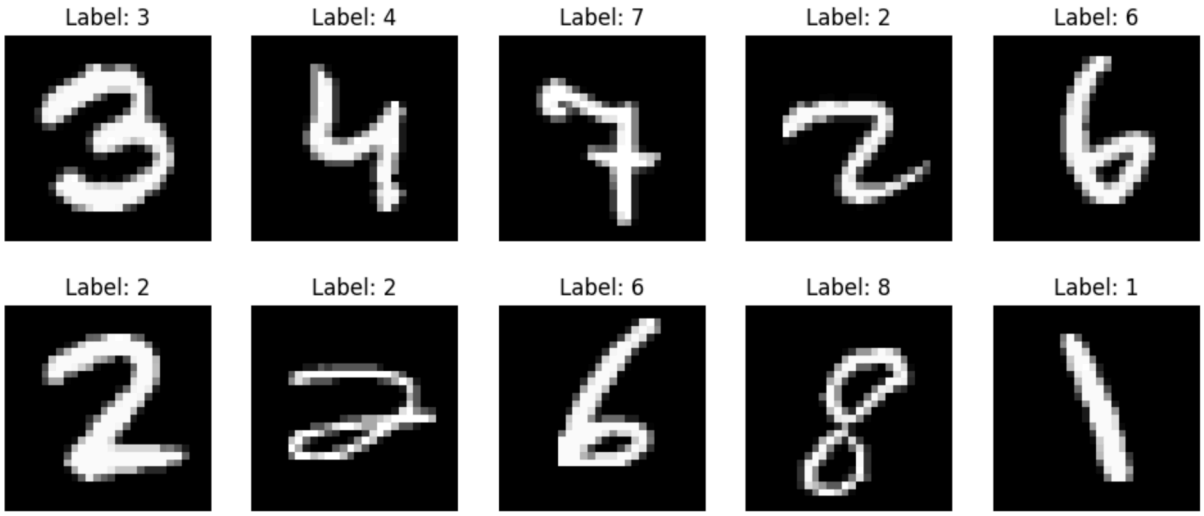
```
# 1 - download the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

then, i shuffled the train data and printed out the dimentions. as it is clear, there are 60000 images and labels for train and 10000 for test.

then i

```
# 2 - Shuffle the training data
```

```
# 3 - Display the first 10 training data samples with labels
plt.figure(figsize=(12, 5))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(x_train[i].squeeze(), cmap='gray')
    plt.title(f"Label: {y_train[i]}")
    plt.axis('off')

plt.show()
```




```
print("Labels (y_train):", y_train.shape)
print("Features (x_test):", x_test.shape)
print("Labels (y_test):", y_test.shape)
```

```
Dimensions of Shuffled Training Data:
Features (x_train): (60000, 28, 28)
Labels (y_train): (60000,)
Features (x_test): (10000, 28, 28)
Labels (y_test): (10000,)
```

displayed the first ten images of training data and the labels.

```
# 4 - normalization
x_train = x_train / 255.0
x_test = x_test / 255.0
```

```
# 5 - Convert labels to categorical representation (one-hot encoding)
from keras.utils import to_categorical

num_classes = 10  # Assuming there are 10 classes (digits 0-9)
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

after that i normalized the pixel values and converted the labels to the categorial representation (one-hot encoding)

i defined the model with layers, optimizer and loss function which question asked for:

```
# 6 - design the convolutional model
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Define the model
model = Sequential()

# Convolutional layer 1
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
# Max pooling layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))

# Convolutional layer 2
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'))
# Max pooling layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))

# Convolutional layer 3
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same', name='last_conv_layer'))
# Max pooling layer 3
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten layer
model.add(Flatten())

# Fully connected layer 1
model.add(Dense(128, activation='relu'))

# Output layer
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

train and validation step:

```python
# 7 - train and evaluate the model
history = model.fit(x_train, y_train, epochs=15, batch_size=64, validation_data=(x_test, y_test))
```

```
Epoch 1/15
938/938 [==============================] - 7s 6ms/step - loss: 0.1788 - accuracy: 0.9450 - val_loss: 0.0508 - val_accuracy: 0.9846
Epoch 2/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0529 - accuracy: 0.9833 - val_loss: 0.0435 - val_accuracy: 0.9855
Epoch 3/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0367 - accuracy: 0.9885 - val_loss: 0.0319 - val_accuracy: 0.9896
Epoch 4/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0275 - accuracy: 0.9912 - val_loss: 0.0344 - val_accuracy: 0.9888
Epoch 5/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0235 - accuracy: 0.9926 - val_loss: 0.0347 - val_accuracy: 0.9884
Epoch 6/15
938/938 [==============================] - 7s 7ms/step - loss: 0.0191 - accuracy: 0.9938 - val_loss: 0.0291 - val_accuracy: 0.9908
Epoch 7/15
938/938 [==============================] - 8s 9ms/step - loss: 0.0168 - accuracy: 0.9945 - val_loss: 0.0244 - val_accuracy: 0.9911
Epoch 8/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0127 - accuracy: 0.9957 - val_loss: 0.0287 - val_accuracy: 0.9911
Epoch 9/15
938/938 [==============================] - 7s 7ms/step - loss: 0.0128 - accuracy: 0.9958 - val_loss: 0.0287 - val_accuracy: 0.9911
Epoch 10/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0089 - accuracy: 0.9969 - val_loss: 0.0289 - val_accuracy: 0.9914
Epoch 11/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0089 - accuracy: 0.9971 - val_loss: 0.0308 - val_accuracy: 0.9912
Epoch 12/15
938/938 [==============================] - 5s 5ms/step - loss: 0.0075 - accuracy: 0.9974 - val_loss: 0.0295 - val_accuracy: 0.9928
Epoch 13/15
938/938 [==============================] - 5s 6ms/step - loss: 0.0074 - accuracy: 0.9975 - val_loss: 0.0335 - val_accuracy: 0.9906
Epoch 14/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0067 - accuracy: 0.9975 - val_loss: 0.0308 - val_accuracy: 0.9928
Epoch 15/15
938/938 [==============================] - 4s 5ms/step - loss: 0.0069 - accuracy: 0.9975 - val_loss: 0.0343 - val_accuracy: 0.9914
```

Then I implemented Grad-CAM (Gradient-weighted Class Activation Mapping), a technique **used for visualizing and understanding the regions of an image that a convolutional neural network (CNN) focuses on when making a prediction. Grad-CAM generates a heatmap overlay on the original image, highlighting the important regions.**

- grad_cam function:

This function takes an image (img) as input. It extracts the last convolutional layer (last_conv_layer) from the model.
It creates a new model (heatmap_model) that takes the same input as the original model but outputs both the output of the last convolutional layer and the final predictions.
It uses a tf.GradientTape to record the operations for gradient calculation.
It computes the loss by selecting the predicted class probability for the given image.
It calculates the gradients of the loss with respect to the output of the last convolutional layer.
The gradients are then averaged over the spatial dimensions to obtain a weight for each channel in the last convolutional layer.
The final heatmap is obtained by multiplying these channel-wise weights with the corresponding output of the last convolutional layer and taking the mean across channels.
The heatmap values are normalized between 0 and 1.

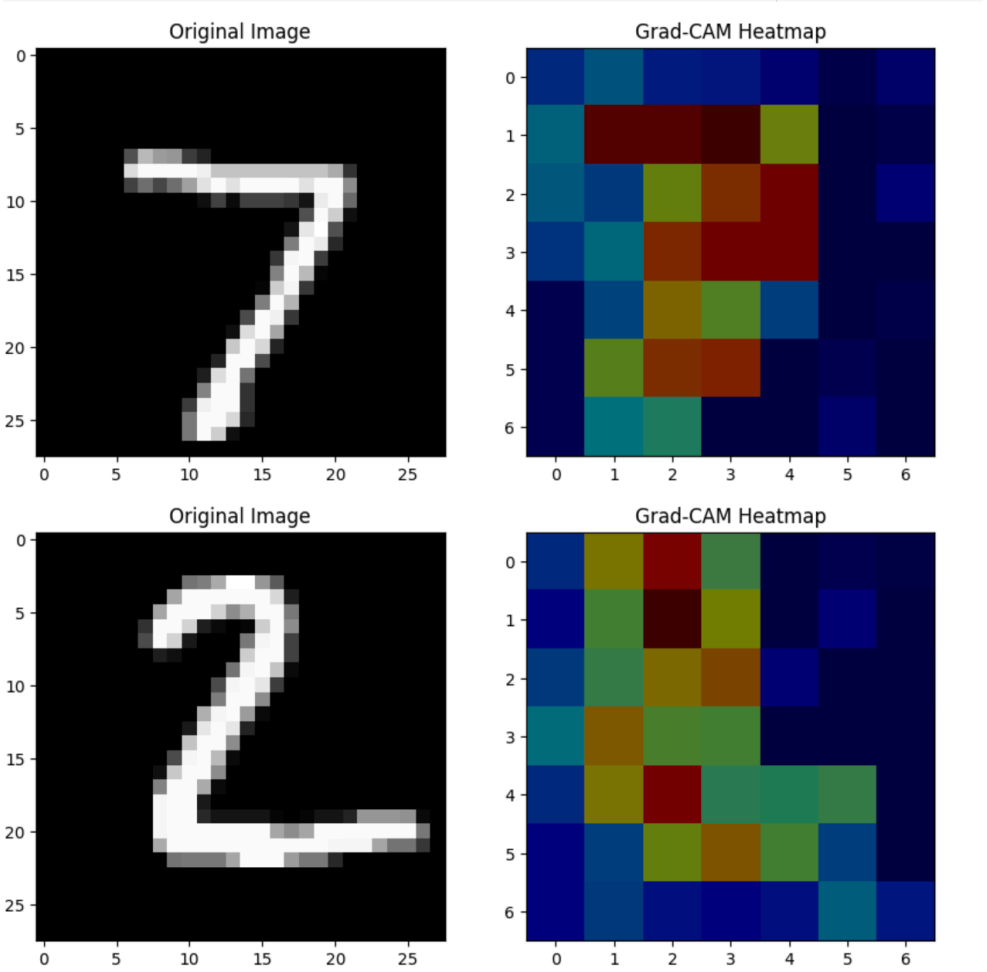
- Visualization loop (for i in range(10)) for the first 10 images in the test set (x_test): For each image, it plots two subplots using Matplotlib.

The first subplot displays the original grayscale image.
The second subplot displays the original image with the Grad-CAM heatmap overlaid.

```
def grad_cam(img):
  last_conv_layer = model.get_layer('last_conv_layer')
  heatmap_model = tf.keras.models.Model(inputs=model.inputs, outputs=[last_conv_layer.output, model.output])


  with tf.GradientTape() as tape:
      conv_outputs, predictions = heatmap_model(x_test[:10].reshape(-1, 28, 28, 1))
      loss = predictions[:, tf.argmax(predictions[0])]
      grads = tape.gradient(loss, conv_outputs)
      gradients_pool = K.mean(grads, axis = (0, 1, 2))
  heatmap = tf.reduce_mean(tf.multiply(gradients_pool, conv_outputs), axis=-1)
  heatmap = np.maximum(heatmap, 0)
  heatmap /= np.max(heatmap)
  return heatmap


for i in range(10):
    img = x_test[i]
    plt.figure(figsize=(10, 5))

    # Original Image
    plt.subplot(1, 2, 1)
    plt.imshow(img.squeeze(), cmap='gray')
    plt.title('Original Image')

    # Grad-CAM Heatmap
    plt.subplot(1, 2, 2)
    plt.imshow(img.squeeze(), cmap='gray')
    plt.imshow(grad_cam(img)[i], cmap='jet', alpha=0.5)
    plt.title('Grad-CAM Heatmap')

    plt.show()
```

here are the results:

Using Grad-CAM can



provide
insights into which regions of the input images are influencing the model's predictions.
MNIST digits are relatively simple, and the model typically focuses on localized features such as edges, curves, and intersections. Grad-CAM can reveal which pixels or regions are crucial for the model to recognize a specific digit.
also Grad-CAM helps interpret the decisions made by the model. By visualizing the heatmap overlaid on the original image, you can see which parts of the digit contribute the most to the model's prediction.