Handwritten-Digit-Classification-Using-Convolutional-Neural-Networks-CNNs Project Result

Training the Model

```
model.fit(X_train, y_train, epochs=10, batch_size=64, validation_split=0.1)
```

- **`X_train` and `y_train`**: These are your training features and labels.
- **`epochs=10`**: The model will be trained for 10 passes over the entire training dataset.
- **`batch_size=64`**: During each epoch, the training data will be divided into batches of 64 samples. The model updates its weights after each batch.
- **`validation_split=0.1`**: 10% of the training data will be used as validation data to monitor the model's performance on unseen data during training.

Understanding the Training Results

The output you've shared provides information about the training process and the model's performance over each epoch:

Epoch 1

- **Training Accuracy**: 86.66%
- **Training Loss**: 0.4445
- **Validation Accuracy**: 98.25%
- **Validation Loss**: 0.0580

Epoch 2

- **Training Accuracy**: 98.17%
- **Training Loss**: 0.0578
- **Validation Accuracy**: 98.40%
- **Validation Loss**: 0.0544

Epoch 3

- **Training Accuracy**: 98.83%
- **Training Loss**: 0.0382
- **Validation Accuracy**: 98.87%
- **Validation Loss**: 0.0366

Epoch 4

- **Training Accuracy**: 99.14%
- **Training Loss**: 0.0287
- **Validation Accuracy**: 99.07%
- **Validation Loss**: 0.0329

Epoch 5

- **Training Accuracy**: 99.25%
- **Training Loss**: 0.0223
- **Validation Accuracy**: 98.77%
- **Validation Loss**: 0.0418

Epoch 6

- **Training Accuracy**: 99.38%
- **Training Loss**: 0.0178
- **Validation Accuracy**: 99.10%
- **Validation Loss**: 0.0339

Epoch 7

- **Training Accuracy**: 99.54%
- **Training Loss**: 0.0139
- **Validation Accuracy**: 99.12%
- **Validation Loss**: 0.0350

Epoch 8

- **Training Accuracy**: 99.62%
- **Training Loss**: 0.0111
- **Validation Accuracy**: 99.15%
- **Validation Loss**: 0.0321

Epoch 9

- **Training Accuracy**: 99.62%
- **Training Loss**: 0.0108
- **Validation Accuracy**: 99.05%
- **Validation Loss**: 0.0414

Epoch 10

- **Training Accuracy**: 99.67%
- **Training Loss**: 0.0092
- **Validation Accuracy**: 99.02%
- **Validation Loss**: 0.0405

Key Insights

1. **Accuracy and Loss Trends**:
    - **Training Accuracy**: The training accuracy increases over epochs, indicating that the model is learning and fitting better to the training data.
    - **Training Loss**: The loss decreases, showing that the model is making fewer errors on the training data.
    - **Validation Accuracy**: The validation accuracy also improves, indicating that the model generalizes well to unseen data.
    - **Validation Loss**: There is some fluctuation, but overall, it decreases or remains stable, suggesting that the model is not overfitting significantly.
2. **Early Stopping**:
    - If you observe that the validation accuracy starts to degrade or the validation loss increases, it might be a sign of overfitting. However, in this case, the model shows good generalization with consistent improvement in both accuracy and loss.
3. **Performance**:
    - By the end of training, the model achieves very high accuracy on both training and validation data, indicating strong performance.

Summary

Overall, your model is training well, with high accuracy and low loss values both on the training and validation datasets. The consistent improvement suggests that your hyperparameters, such as epochs and batch size, are well-tuned for this model and dataset.

Implementing Early Stopping and Learning Rate Scheduling

Why Implement Early Stopping and Learning Rate Scheduling?

1. **Early Stopping**:
    - **Purpose**: Prevents overfitting by halting training when the model's performance on a validation set stops improving. This helps in saving time and computational resources, and ensures that the model does not learn noise from the training data.
    - **Implementation**: It monitors the `val_loss` and stops training if there is no improvement for a certain number of epochs (`patience`). It also restores the model weights from the best epoch to avoid retaining overfitted weights.
2. **Learning Rate Scheduling**:

- **Purpose**: Adjusts the learning rate during training to improve convergence. A dynamic learning rate can help the model converge faster and avoid local minima.
- **Implementation**: It reduces the learning rate when the `val_loss` plateaus or increases, allowing for finer adjustments in the later stages of training.

Code Explanation

Here's what each part of your updated code does:

Early Stopping

```
python
Copy code
from tensorflow.keras.callbacks import EarlyStopping

# Define Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)
```

- **`monitor='val_loss'`**: Tracks the validation loss to decide when to stop.
- **`patience=3`**: Allows the model to continue training for 3 more epochs after no improvement is observed.
- **`restore_best_weights=True`**: Restores the model weights from the epoch with the best validation loss.

Learning Rate Scheduler

```
python
Copy code
from tensorflow.keras.callbacks import ReduceLROnPlateau

# Define Learning Rate Scheduler
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2,
min_lr=0.0001)
```

- **`monitor='val_loss'`**: Tracks the validation loss to decide when to reduce the learning rate.
- **`factor=0.2`**: Reduces the learning rate by a factor of 0.2 when the validation loss plateaus.
- **`patience=2`**: Allows for 2 epochs of no improvement before reducing the learning rate.
- **`min_lr=0.0001`**: Ensures that the learning rate doesn’t fall below this minimum value.

Updated `model.fit()` Call

```
python
Copy code
history = model.fit(
    X_train, y_train,
    epochs=50,  # Increased epochs
    batch_size=64,
```

```
    validation_split=0.1,
    callbacks=[early_stop, reduce_lr]
)
```

- **`epochs=50`**: Allows training for up to 50 epochs, giving room for early stopping to potentially halt the training earlier.
- **`callbacks=[early_stop, reduce_lr]`**: Includes early stopping and learning rate scheduler in the training process.

Results Analysis

Training Log

```
plaintext
Copy code
Epoch 1/50
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21s 25ms/step - accuracy: 0.9970 -
loss: 0.0082 - val_accuracy: 0.9885 - val_loss: 0.0542 - learning_rate:
0.0010
...
Epoch 8/50
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 19s 23ms/step - accuracy: 1.0000 -
loss: 9.6579e-05 - val_accuracy: 0.9935 - val_loss: 0.0375 - learning_rate:
1.0000e-04
```

- **Initial Epochs**: The accuracy improves and the loss decreases, indicating that the model is learning effectively.
- **Learning Rate Adjustment**: As training progresses, the learning rate decreases (e.g., from 0.0010 to 0.0001) when the `val_loss` plateaus, allowing for more precise adjustments.
- **High Accuracy**: The model reaches nearly perfect accuracy on the training set (`1.0000`), indicating strong performance. However, it's crucial to ensure that the validation accuracy and loss also remain stable and high.

Summary

Your use of early stopping and learning rate scheduling helps to avoid overfitting and improves convergence by dynamically adjusting the learning rate. This can lead to a more robust model that generalizes better to unseen data. The results suggest that the model is training well with good accuracy and low loss, and the callbacks are working as expected to optimize training.

Why Apply Regularization Techniques?

Regularization helps prevent overfitting, where the model learns the training data too well, including noise and irrelevant patterns. This can result in poor generalization to unseen data. By applying regularization, we ensure the model becomes more robust and performs better on new, unseen data.

How Does Regularization Work?

One effective regularization method is **Dropout**. It randomly deactivates a certain percentage of neurons during training, forcing the model to rely on multiple neurons rather than specific ones. This reduces overfitting and improves generalization.

Key Components:

1. **Convolutional Layers**: Extract features from the input images using filters.
2. **Batch Normalization**: Stabilizes learning by normalizing activations, allowing the network to converge faster.
3. **Max Pooling**: Reduces the spatial dimensions of the feature maps, helping prevent overfitting and speeding up the model.
4. **Dropout**:
    - **25% Dropout** is applied after convolutional layers to prevent overfitting in intermediate layers.
    - **50% Dropout** before the final fully connected layer helps regularize the most dense part of the network, ensuring neurons don't overly specialize.

Why Different Dropout Rates?

- **25% in early layers**: These layers typically capture low-level features, and we want the model to retain most of this information.
- **50% in dense layers**: Dense layers tend to have more parameters, making them more prone to overfitting, so we apply a higher dropout rate here.

By combining dropout with batch normalization and other techniques, we create a more regularized model that generalizes well across different datasets.

Why Hyperparameter Tuning?

Hyperparameter tuning is crucial because it helps in finding the best possible configuration for your model's hyperparameters, leading to better performance and generalization. Hyperparameters are settings that are not learned from the data but set before training begins, such as the number of layers, learning rate, and batch size. Tuning these can significantly impact the model's accuracy and efficiency.

Explanation of the Result

Trial 10 Complete [00h 03m 22s]:

- The training process for the 10th hyperparameter configuration took 3 minutes and 22 seconds.
- During this trial, the **validation accuracy** achieved was **0.9859** (or 98.59%), meaning that the model performed well on the unseen validation data, correctly predicting 98.59% of the examples.

Best val_accuracy So Far: 0.9926 (99.26%)

- Across all trials, the best validation accuracy achieved so far is **99.26%**, meaning that one of the previous hyperparameter configurations outperformed the 10th trial in terms of generalization.
- This metric is used to evaluate how well the model generalizes to new, unseen data.

Total elapsed time: 09h 37m 13s:

- The total time spent across all 10 trials was **9 hours, 37 minutes, and 13 seconds**. Tuning hyperparameters can be time-consuming, but it's essential to find the best configuration for optimal model performance.

Optimal Hyperparameters Found

After running multiple trials, the Keras Tuner has identified the best hyperparameters:

1. **Number of filters in the first Conv2D layer: 128**
    - The tuner determined that using 128 filters in the first convolutional layer yields the best performance.
    - Filters are responsible for detecting features in images, and 128 filters help in identifying more intricate patterns in the dataset.
2. **Number of filters in the second Conv2D layer: 192**
    - The second convolutional layer uses 192 filters, which allows the model to capture even more detailed patterns as it goes deeper into the network.
3. **Number of units in the Dense layer: 256**
    - The fully connected dense layer is responsible for combining the features detected by the convolutional layers. Having 256 units in this layer balances complexity and model capacity.
4. **Optimizer: Adam**

- The **Adam optimizer** is selected as the best optimizer. Adam (Adaptive Moment Estimation) is commonly used because it combines the advantages of other optimizers like RMSprop and Stochastic Gradient Descent (SGD) with momentum, making it more efficient and robust for different types of models.

Summary

- **Trial 10** achieved a validation accuracy of **98.59%**, but the best configuration across all trials reached a validation accuracy of **99.26%**.
- The optimal architecture for the convolutional layers was found to use **128 filters in the first layer** and **192 in the second**, followed by a dense layer with **256 units**.
- The model performed best using the **Adam optimizer**, which balances fast convergence and stability.

Implementing Residual Networks (ResNet) – Explanation

Why Use ResNet?

- **Residual Networks (ResNet)** were introduced to address the **vanishing gradient problem** that occurs in deep neural networks, where the gradient diminishes as it is backpropagated through the layers. This makes it difficult for the network to learn in very deep architectures.
- ResNet solves this by introducing **skip connections**, which allow the model to learn the identity function more easily. By adding the input (`shortcut`) directly to the output of a block, ResNet ensures that important information passes through layers even if some intermediate transformations fail.

Model Summary:

The model has several key components:

1. **Input Layer:**
   - Shape: `(None, 28, 28, 1)`, indicating grayscale images (MNIST digits) with dimensions 28x28 pixels.
   - No parameters to train at this stage.
2. **Convolutional and Residual Blocks:**
   - The first block applies a convolution (`conv2d_12`) followed by batch normalization (`batch_normalization_12`), which helps stabilize and accelerate training.
   - Then, the residual block is applied: two convolution layers (`conv2d_13` and `conv2d_14`), both followed by batch normalization, and a skip connection that adds the input to the output of these layers. This skip connection creates the residual block.

- Activation is applied after the addition to introduce non-linearity.

Similar structures are repeated in the later layers with increasing filter sizes, from 64 to 128 filters.

3. **Max Pooling and Dropout:**
   - After each residual block, a max pooling layer reduces the spatial dimensions of the feature maps, and dropout layers prevent overfitting by randomly setting a fraction of the input units to 0 during training.
4. **Dense Layers:**
   - After flattening the feature maps, a dense (fully connected) layer with 256 units and a final output layer with 10 units (for 10 digit classes) is used. Batch normalization and dropout are applied before the final output layer.
5. **Total Parameters:**
   - The model has **2,055,306 parameters** in total, with the vast majority (1,605,888) concentrated in the fully connected dense layer. About **7.83 MB** of memory is used for these parameters.

Training Results:

- **Epoch 1:**
  - **Accuracy:** 83.78%
  - **Loss:** 0.5341
  - **Validation Accuracy:** 98.46%
  - **Validation Loss:** 0.0497
  - **Learning Rate:** 0.0010
  - The model starts with a relatively high accuracy (83.78%) on the training set. The validation accuracy is even higher (98.46%) after the first epoch, showing that the model is learning quickly but still has some room for improvement.
- **Epoch 2:**
  - **Accuracy:** 97.01%
  - **Loss:** 0.0991
  - **Validation Accuracy:** 98.68%
  - **Validation Loss:** 0.0478
  - The training accuracy improves significantly to 97.01%, and the validation accuracy increases slightly. This shows that the model is generalizing well without overfitting, and the validation loss has decreased to 0.0478, indicating better performance on unseen data.
- **Epoch 3:**
  - **Accuracy:** 97.69%
  - **Loss:** 0.0745
  - **Validation Accuracy:** 98.49%
  - **Validation Loss:** 0.0458
  - Continued improvement in training accuracy, but the validation accuracy stays around 98.49%. The model is nearing its optimal performance, as both training and validation metrics converge.

Overall Insights:

- The model is learning effectively, as indicated by the steady improvement in training accuracy and decreasing loss values.
- The **batch normalization** and **dropout** layers are likely helping prevent overfitting, as shown by the stable validation accuracy and minimal gap between training and validation performance.
- **Residual connections** are also contributing to the model's ability to train deeply without suffering from vanishing gradients.

Purpose of the Confusion Matrix:

The confusion matrix evaluates how well the model predicts each digit from 0 to 9, showing both correct and incorrect classifications. Each row corresponds to the true label (actual digit), and each column corresponds to the predicted label (model's output). It helps in understanding:

- **True Positives** (correct classifications): Values on the diagonal.
- **False Positives and False Negatives** (misclassifications): Off-diagonal values.

Detailed Analysis of the Confusion Matrix:

- **Correct Predictions (Diagonal values):** These are high across the matrix, which is excellent. For instance:
  - **Digit 0**: Correctly classified 975 times.
  - **Digit 1**: Correctly classified 1,135 times.
  - **Digit 2**: Correctly classified 1,025 times, and so on.
- **Misclassifications (Off-diagonal values):**
  - **Digit 6** has the most notable confusion:
    - Misclassified as **digit 0** (5 times).
    - Misclassified as **digit 4** (3 times).
    - Misclassified as **digit 1** (4 times).
  - **Digit 8** is occasionally misclassified as **digit 2** and **digit 5** (2 instances each).
  - **Digit 5** was misclassified as **digit 3** five times.

These misclassifications are expected since certain digits (like 6 and 0, or 8 and 3) share visual similarities, making them harder to distinguish.

Metrics for Model Evaluation:

1. **Accuracy:** Given that the diagonal values are dominant, your model has high accuracy. Accuracy is calculated as the ratio of correct predictions to the total predictions, which would be very close to 99%.

2. **Precision, Recall, F1-Score:** These additional metrics are not shown in the confusion matrix itself but can be derived from it:
    - **Precision** measures how many of the predicted positives were correct.
    - **Recall** measures how many of the actual positives were correctly identified.
    - **F1-Score** is the harmonic mean of precision and recall, giving a balanced measure of the model's performance.

Model Strengths:

- **High overall accuracy:** Most of the digits are predicted correctly, with very few misclassifications.
- **Consistent performance across all digits:** The high diagonal values suggest the model performs well across all digit classes.

Areas for Improvement:

- **Misclassification patterns:** The confusion between certain digits (like 6 and 0, 8 and 5) indicates areas where the model can be improved. Techniques such as:
    - Further tuning the hyperparameters.
    - Increasing model complexity.
    - Using advanced regularization techniques.

could help in reducing these errors.

Conclusion:

Your model performs well on the MNIST dataset, with minimal misclassifications. The few errors that exist are likely due to the visual similarity between certain digits, which is common in this dataset. Exploring precision, recall, and F1-score metrics would give a more comprehensive understanding of how the model handles different classes of digits.

lassification Report Analysis

This classification report gives a detailed breakdown of your model's performance on each digit from 0 to 9. Let's walk through each of the metrics and their significance.

Metrics:

1. **Precision**: This measures the proportion of true positives (correct predictions) out of all the predictions made for a given class.
    - A precision score of 0.99 or 1.00 for all digits indicates that when the model predicts a digit (e.g., predicts "0"), it is almost always correct. There are very few false positives.

2. **Recall**: This measures the proportion of actual positives that were correctly identified by the model.
    - All the digits have a recall close to or equal to 1.00, which means that the model captures almost all of the true instances of each digit. For example, for digit "1", the recall of 1.00 means the model found **all** the instances of "1" correctly.
3. **F1-Score**: This is the harmonic mean of precision and recall, giving a balanced view of the model's performance for each digit.
    - With all F1-scores at 0.99 or higher, this confirms that your model balances both precision and recall extremely well.
4. **Support**: This indicates how many true instances of each class (digit) are present in the dataset.
    - For example, there are 980 images of the digit "0" and 1135 images of the digit "1".

Accuracy:

- **Overall Accuracy: 99%**: This confirms that the model correctly predicted the digits in 99% of the 10,000 test samples. It's a strong indicator of the model's performance.

Macro and Weighted Averages:

- **Macro avg**: The simple average of precision, recall, and F1-score for all classes.
    - A score of 0.99 shows that each class (digit) performs almost equally well.
- **Weighted avg**: Takes into account the support (number of instances) of each class, providing a more balanced view when some classes are more common than others.
    - A score of 0.99 for the weighted average confirms that even though some digits (like "1") appear more often than others, the model handles the imbalanced dataset very well.

Insights:

- **Strengths**:
    - The model is extremely effective at both predicting and identifying each digit correctly, which is reflected in the consistently high precision, recall, and F1-scores.
    - There is a minimal margin for error, making the model highly reliable for recognizing handwritten digits in the MNIST dataset.
- **Potential for Improvement**:
    - Even though precision and recall are near-perfect, some digits like "6" and "5" show slightly lower recall (0.98 and 0.99, respectively). This suggests a few misclassifications that may be caused by the visual similarities between certain digits.

Explanation of the Code:

1. **Model Loading**:

- `model = tf.keras.models.load_model('your_model.h5')`: This line loads the trained Convolutional Neural Network (CNN) model from your `.h5` file. Make sure to replace `'your_model.h5'` with the actual path to your model file.

2. **File Upload**:
   - `st.file_uploader()`: Streamlit provides a user-friendly interface for uploading files, allowing users to upload an image from their device.
3. **Image Preprocessing**:
   - The uploaded image is converted to grayscale and resized to 28x28 pixels, which matches the MNIST dataset format used during training.
   - `img_array = np.array(image) / 255.0`: The image is normalized to have pixel values between 0 and 1.
   - `img_array = img_array.reshape(1, 28, 28, 1)`: The array is reshaped to the format expected by the model (batch size of 1, 28x28 image size, and 1 color channel).
4. **Prediction**:
   - The model predicts the digit from the uploaded image using `model.predict(img_array)`.
   - `np.argmax(prediction, axis=1)[0]`: The predicted digit is extracted from the prediction array (the index with the highest probability).
5. **Displaying Results**:
   - The uploaded image is displayed using `st.image()`.
   - The predicted digit is shown using `st.write()`.