Hand Gesture Recognition Using Convolutional Neural Networks - Artificial Intelligence for Engineers DAT305

DONNY MARTHEN SITOMPUL, University of Stavanger, Norway

This paper presents a comprehensive approach to developing a hand gesture recognition system using Convolutional Neural Networks (CNN). The system is trained on the Leap Motion Gesture Dataset, which consists of grayscale images of ten different hand gestures. The paper discusses the key steps involved in building the system, including data preprocessing, model architecture, training, and evaluation. Data preprocessing involves resizing the images and splitting the dataset into training, validation, and testing sets. A CNN model is designed to extract relevant features from the images and classify them accurately. The model is trained using appropriate loss functions, optimizers, and evaluation metrics. The paper also highlights the importance of visualizing the training process to monitor model performance and prevent overfitting. By following the guidelines outlined in this paper, researchers and practitioners can effectively develop and deploy hand gesture recognition systems for a variety of applications, such as human-computer interaction and virtual reality. The github repository can be found https://github.com/dms-codes/hand-gesture

1 Introduction

Hand gesture recognition, a vital component of human-computer interaction, is the focus of this project. A gesture recognition system is implemented using Convolutional Neural Networks (CNNs). This system processes grayscale images of hand gestures and employs CNNs to classify them into predefined categories.

2 Data Preprocessing

2.1 Dataset Overview

The dataset used is the Leap Motion Gesture Dataset with 10 gesture classes. Each gesture folder contains grayscale images of hands performing specific gestures. The database can be downloaded from https://www.kaggle.com/datasets/gti-upm/leapgestrecog. See Fig. 1 till Fig. 10 for samples of images.

01_palm: A simple, relaxed hand position with all fingers extended.

02_l : A hand forming the letter "L" shape, with the thumb and index finger extended and touching, while the other fingers are curled.

03_fist : A clenched hand with all fingers curled tightly inward.

04_fist_moved : A clenched fist that is being moved, either in a specific direction or in a general motion.

05_thumb: A thumbs-up gesture, indicating approval, agreement, or positivity.

06_index : A pointed finger, often used to indicate or direct attention.

07_ok : A gesture formed by touching the thumb and index finger to form a circle, while the other fingers are extended. It typically signifies approval or agreement.

08_palm_moved : An open hand that is being moved, either in a specific direction or in a general motion.

Fig. 1. 01_palm

Fig. 2. 02_l

Fig. 3. 03_fist

Fig. 4. 04_fist_moved

Fig. 5. 05_thumb

Fig. 6. 06_index

Fig. 7. 07_ok

Author's Contact Information: Donny Marthen Sitompul, dm.sitompul@stud.uis.no, University of Stavanger, Stavanger, Rogaland, Norway.

Fig. 8. 08_palm_moved

Fig. 9. 09_c

Fig. 10. 10_down

09_c : A hand forming the letter "C" shape, with the thumb and index finger touching to form a curve, while the other fingers are extended.

10_down: A hand with the palm facing downward, often used to indicate a downward motion or a negative gesture.

2.2 Folder Structure and Lookup

Each gesture is stored in subfolders. To handle the labels programmatically, we create a lookup dictionary to map gesture names to numeric labels and create a reverse lookup dictionary for easy interpretation of predictions.

```
def create_lookup(directory: str):
    print("0. Creating Folder Structure and Lookup")
    if not os.path.exists(LOOKUP_FILENAME):
        with open(LOOKUP_FILENAME, 'ab') as resfile:
            lookup = {name: i
            for i, name in enumerate(os.listdir(directory))
            if not name.startswith('.')}
            reverselookup = {i: name for name, i in lookup.items()}
            res = lookup, reverselookup
            pickle.dump(res, resfile)
            resfile.close()
        print(f"File '{LOOKUP_FILENAME}' created and data stored.")
    else:
        resfile = open(LOOKUP_FILENAME, 'rb')
        res = pickle.load(resfile)
        resfile.close()
    return res
```

2.3 Image Loading and Resizing

All images are resized to a consistent shape of 120x320 pixels for efficient model training. Images are converted to grayscale to simplify computation.

```
def load_images_for_gesture(gesture_path: str,
    image_size=(320, 120)):
    return np.array([
        np.array(Image.open(os.path.join(gesture_path,
        image_file)).convert('L').resize(image_size),
        dtype='float32')
        for image_file in filter(lambda f: not f.startswith('.'),
        os.listdir(gesture_path))
    ])
```

2.4 Dataset Creation

The function create_dataset iterates over all gesture folders, loading images and assigning appropriate labels.

```
def create_dataset(lookup, base_path="input/leapGestRecog/"):
    print("1. Creating dataset")
    if not os.path.exists(DATASET_FILENAME):
        with open(DATASET_FILENAME, 'ab') as resfile:
            x_data, y_data = [], []
            for i in range(10):
                folder_path = os.path.join(base_path, f"0{i}")
                for gesture_folder in filter(lambda x: not
                x.startswith('.'), os.listdir(folder_path)):
                    gesture_path = os.path.join(folder_path,
                    gesture_folder)
                    gesture_images =
                    load_images_for_gesture(gesture_path)
                    x_data.extend(gesture_images)
                    y_data.extend([lookup[gesture_folder]] *
                    len(gesture_images))
            dataset = np.array(x_data),
            np.array(y_data).reshape(-1, 1)
            pickle.dump(dataset, resfile)
            resfile.close()
        print(f"File '{DATASET_FILENAME}' created
        and data stored.")
    else:
        resfile = open(DATASET_FILENAME, 'rb')
        dataset = pickle.load(resfile)
        resfile.close()
    return dataset
```

2.5 Data Splitting

We split the data into an 80% training set for model training, a 10% validation set for hyperparameter tuning, and a 10% test set for performance evaluation.

```
def prepare_data(x_data, y_data):
    y_data = to_categorical(y_data)
    x_data = x_data.reshape(-1, 120, 320, 1)
    print("2. Splitting the dataset")
    x_train, x_temp, y_train, y_temp = train_test_split(x_data,
    y_data, test_size=0.2, random_state=42)
    x_validate, x_test, y_validate, y_test =
    train_test_split(x_temp, y_temp,
    test_size=0.5, random_state=42)
```

```
    return x_train, x_validate, x_test, y_train, y_validate,
    y_test
```

y_data = to_categorical(y_data), converts the labels (y_data) into a one-hot encoded format. If y_data contains class labels as integers (e.g., [0, 1, 2, 3]), one-hot encoding transforms each label into a vector where only the index corresponding to the label is 1 and all other indices are 0. For example:

Original y_data: [0, 1, 2]

After one-hot encoding:

[[1, 0, 0], # Class 0

[0, 1, 0], # Class 1

[0, 0, 1]] # Class 2

Neural networks expect output labels in this format when solving classification problems because they use probabilities to assign each input to a specific class.

x_data = x_data.reshape(-1, 120, 320, 1) reshapes the input data (x_data) to include the channel dimension required by Convolutional Neural Networks (CNNs). x_data contains grayscale images, so each pixel is represented by a single intensity value. Adding a channel dimension explicitly specifies that these are single-channel (grayscale) images. The shape is adjusted to (-1, 120, 320, 1). We defined the image dimensions as 120x320 pixels with a single grayscale channel, automatically inferring the number of samples. CNN models typically expect input data with a specific shape, especially the inclusion of a channel dimension. This step ensures compatibility with the model's input layer.

x_train, x_temp, y_train, y_temp = train_test_split(x_data, y_data, test_size=0.2, random_state=42) The dataset was divided into training and temporary sets. Input images (x_data) and their corresponding one-hot encoded labels (y_data) were split, with 20% allocated to the temporary set (x_temp, y_temp) and 80% to the training set (x_train, y_train). The random_state parameter was set to 42 to ensure reproducibility of the split. The training set is used to train the model, while the temporary set will be further split into validation and test sets.

x_validate, x_test, y_validate, y_test = train_test_split(x_temp, y_temp, test_size=0.5, random_state=42) The temporary set was further divided into validation and test sets, each comprising 50% of the temporary data. The validation set, consisting of x_validate and y_validate, is used during training to assess the model's performance on unseen data and prevent overfitting. This aids in tuning hyperparameters. The test set, comprising x_test and y_test, is reserved for the final evaluation of the model's performance after training.

return x_train, x_validate, x_test, y_train, y_validate, y_test This code generates six datasets: training inputs and labels (x_train, y_train), validation inputs and labels (x_validate, y_validate), and test inputs and labels (x_test, y_test). This division is a standard practice in machine learning to ensure unbiased evaluation of the model's performance through distinct training, validation, and testing phases. The prepare_data function formats the data appropriately for neural networks, including one-hot encoding labels and adjusting input shapes. It then divides the data into training, validation, and test sets, establishing a strong foundation for building a robust machine learning model.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 58, 158, 32)	832
max_pooling2d (MaxPooling2D)	(None, 29, 79, 32)	0
max_pooling2d_1 (MaxPooling2D)	(None, 13, 38, 64)	0
conv2d_2 (Conv2D)	(None, 11, 36, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(None, 5, 18, 64)	0
flatten (Flatten)	(None, 5760)	0
dense (Dense)	(None, 128)	737,408
dense_1 (Dense)	(None, 10)	1,290

Total params: 794,954 (3.03 MB)
Trainable params: 794,954 (3.03 MB)
Non-trainable params: 0 (0.00 B)

Table 1. Model Summary CNN

3 Model Design, Training, and Implementation

3.1 Model Architecture

The CNN model, constructed using Keras, comprises convolutional layers to extract features such as edges and textures from images. MaxPooling layers then downsample these feature maps to reduce computational cost and introduce invariance to small translations and distortions. Finally, fully connected layers classify the images based on the extracted features.

```
def build_model_cnn(input_shape):
    print("3. Building the CNN model.")
    model = models.Sequential([
        Input(shape=input_shape),
        layers.Conv2D(32, (5, 5), strides=(2, 2), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    model.summary_report = capture_model_summary(model)
    return model
```

models.Sequential() This creates a linear stack of layers, where each layer's output is the input to the next layer. This approach is ideal for models with a straightforward flow of data.

layers.Conv2D(32, (5, 5), strides=(2, 2), activation='relu', input _shape=input_shape) The addition of a 2D convolutional layer is crucial for extracting spatial features like edges, textures, and shapes from the input image. This layer utilizes 32 filters, each of size 5x5, to learn specific features. The strides parameter of (2, 2) leads to downsampling by moving the filter 2 pixels at a time in both horizontal and vertical directions. The ReLU activation function introduces

non-linearity, enhancing the model's ability to learn complex patterns. The input_shape parameter, set to (120, 320, 1) in this example, specifies the dimensions of grayscale images. Convolutional layers form the foundation of CNNs, allowing them to effectively detect patterns within images.

layers.MaxPooling2D((2, 2)) The max-pooling layer reduces the spatial dimensions of the feature map by selecting the maximum value from a 2x2 region. This downsampling by a factor of 2 helps reduce computational cost and retain the most important features while discarding irrelevant details. Additionally, it prevents overfitting by providing spatial invariance, meaning minor translations or distortions in the image have minimal impact on the results.

layers.Conv2D(64, (3, 3), activation='relu') The addition of another convolutional layer with 64 filters and a 3x3 filter size enhances the model's capacity to learn more complex features. The smaller filter size compared to the previous layer enables the model to focus on finer details. The ReLU activation function, as previously used, introduces non-linearity. Stacking multiple convolutional layers allows the model to progressively detect increasingly abstract patterns. While early layers might identify edges or textures, later layers can detect more intricate shapes or objects.

layers.MaxPooling2D((2, 2)) Another max-pooling layer, similar to the previous one, further reduces the size of the feature maps. This will ensure the feature maps are compact, which reduces computation and helps prevent overfitting.

layers.Conv2D(64, (3, 3), activation='relu') The third convolutional layer, with 64 filters and a 3x3 filter size, maintains the focus on finer details. The ReLU activation function is once again employed to introduce non-linearity. By deepening the network with additional convolutional layers, the model can learn more complex and abstract features.

layers.Flatten() The flattening layer transforms the 2D feature maps into a 1D vector. For instance, a feature map with dimensions 15x40x64 is reshaped into a vector of size 38,400. This step is crucial to prepare the data for the subsequent fully connected (dense) layers, which are designed to process 1D vectors.

layers.Dense(128, activation='relu') The addition of a fully connected (dense) layer with 128 units enables each neuron to learn different combinations of features from the preceding layer. The ReLU activation function allows the model to learn non-linear relationships between these features. Dense layers play a crucial role in combining the features extracted by the convolutional layers and making informed decisions about their significance for classification.

layers.Dense(10, activation='softmax') The output layer consists of 10 units, each corresponding to one of the 10 gesture classes in the dataset. The softmax activation function transforms the raw output (logits) into probabilities, ensuring that they sum up to 1. The class with the highest probability is selected as the predicted class. This softmax activation makes the model's output interpretable as probabilities, making it well-suited for multi-class classification tasks.

The CNN architecture employed in this model comprises convolutional layers for extracting spatial features such as edges, shapes, and textures from the images. MaxPooling layers reduce computational complexity and focus on the most important features. Dense layers combine these extracted features to classify the input into one of 10 gesture classes. Finally, the softmax output layer provides probabilities for multi-class classification, ensuring interpretability of predictions. This compact yet powerful architecture is well-suited for tasks involving image classification, such as gesture recognition. A summary of the CNN models is given in Table 1.

3.2 Model Compilation

The model is compiled using the categorical crossentropy loss function, suitable for multi-class classification tasks. The rmsprop optimizer is employed to adapt the learning rate during training. The accuracy metric is used to evaluate the model's performance.

```
def compile_model(model):
    if model.type == MODEL_TYPE_CNN:
        model.compile(optimizer='rmsprop',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
    elif model.type == MODEL_TYPE_LENET5:
        model.compile(optimizer=Adam(learning_rate=0.001),
              loss="categorical_crossentropy",
              metrics=["accuracy"])
    elif model.type == MODEL_TYPE_ALEXNET:
        model.compile(
            optimizer=Adam(learning_rate=0.001),
            loss="categorical_crossentropy",
            metrics=["accuracy"]
        )
    elif model.type == MODEL_TYPE_VGGNET:
        model.compile(optimizer='adam',
        loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

The model.compile() function is a critical step in preparing a neural network for training in Keras. optimizer='rmsprop' The optimizer is the algorithm used to adjust the weights of the network during training in order to minimize the loss function. In this case, rmsprop (Root Mean Square Propagation) is chosen because it is well-suited for neural networks involving large datasets and it is effective for image-based tasks, like gesture recognition, as it balances speed and accuracy during convergence.. It adapts the learning rate of each parameter based on the average of its recent gradient magnitudes. This makes it especially effective for handling non-stationary objectives (where gradients change over time).

loss='categorical_crossentropy' The loss function measures how far the model's predictions are from the true labels. This is used for multi-class classification problems (i.e., when there are more than two categories, and each input belongs to exactly one category). It compares the predicted probability distribution (output of the model) with the actual one-hot-encoded labels. It ensures the model outputs a probability distribution (sums to 1) across all classes, making it suitable for the gesture recognition problem.

metrics=['accuracy'] The metrics argument specifies what metrics should be evaluated during training and testing. In this case it is accuracy that measures the fraction of predictions that match the true labels. It gives a simple and interpretable measure of how well

the model is performing. By specifying these settings, the model is ready to begin learning from data when you call model.fit().

3.3 Training the Model

We train the model for 10 epochs with a batch size of 64. Validation data is used to monitor performance during training.

```
def train_model(model):
    if model.type == MODEL_TYPE_CNN:
        history = model.fit(x_train, y_train, epochs=10,
        batch_size=64, validation_data=(x_validate, y_validate),
        verbose=1)
    elif model.type == MODEL_TYPE_LENET5:
        history = model.fit(
            x_train, y_train,
            validation_data=(x_validate, y_validate),
            epochs=20,
            batch_size=64)
    elif model.type == MODEL_TYPE_ALEXNET:
        history = model.fit(
            x_train, y_train,
            validation_data=(x_validate, y_validate),
            epochs=20,
            batch_size=64
        )
    elif model.type == MODEL_TYPE_VGGNET:
        history = model.fit(x_train, y_train,
                validation_data=(x_validate, y_validate),
                epochs=20,
                batch_size=32
                )

    model.history = history
    plot_training_history(model)
    return model
```

The function trains the model on the provided data. It iteratively feeds the training data to the model, calculates the loss using the specified loss function, and optimizes the model's weights using the specified optimizer to minimize this loss. After each epoch, the model is evaluated on the validation data (if provided) to monitor its performance.

Inputs to model.fit() x_train and y_train The x_train variable holds the input training data, which in this case consists of preprocessed gesture images. It has a shape of (number of samples, height, width, channels), such as (N, 120, 320, 1) for grayscale images. The y_train variable contains the corresponding labels for x_train, represented as one-hot encoded vectors. For instance, in a 10-class scenario, the label for class 3 would be [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]. epochs=10 Epochs represent the number of times the model will see the entire training dataset. In this case, the training process will loop over the dataset 10 times. Increasing the number of epochs can improve performance but may also lead to overfitting if the model trains too long. batch_size=64 Batch size refers to the number of samples processed by the model before updating its weights. In this specific case, the training data is divided into batches of 64 samples each. The model calculates the loss and adjusts its weights after processing each batch. Using batches offers several advantages: it saves memory by preventing the entire dataset from being loaded

```
› /Users/donnysitompul/.pyenv/versions/3.11.3/bin/python /Users/donnysitompul/Code/hand-gesture/main5.py

Artificial Intelligence for Engineers DAT305
Hand Gesture Recognition
By Donny Marthen Sitompul

0. Creating Folder Structure and Lookup
1. Creating dataset
2. Splitting the dataset
Check if the model is already saved
Saved model not found. Building and training a new model...
3. Building the CNN model.
4. Compiling the model
5. Training the model
Epoch 1/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 41s 152ms/step - accuracy: 0.7917 - loss: 3.1833 - val_accuracy: 0.9970 - val_loss: 0.0131
Epoch 2/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 39s 154ms/step - accuracy: 0.9950 - loss: 0.0225 - val_accuracy: 0.9990 - val_loss: 0.0086
Epoch 3/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 38s 152ms/step - accuracy: 0.9997 - loss: 0.0021 - val_accuracy: 0.9990 - val_loss: 0.0100
Epoch 4/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 38s 153ms/step - accuracy: 0.9984 - loss: 0.0087 - val_accuracy: 0.9995 - val_loss: 0.0057
Epoch 5/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 39s 156ms/step - accuracy: 0.9980 - loss: 0.0131 - val_accuracy: 0.9985 - val_loss: 0.0143
Epoch 6/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 39s 158ms/step - accuracy: 0.9977 - loss: 0.0150 - val_accuracy: 0.9990 - val_loss: 0.0073
Epoch 7/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 39s 157ms/step - accuracy: 0.9993 - loss: 0.0056 - val_accuracy: 0.9975 - val_loss: 0.0131
Epoch 8/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 42s 168ms/step - accuracy: 0.9996 - loss: 0.0014 - val_accuracy: 0.9995 - val_loss: 0.0034
Epoch 9/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 45s 179ms/step - accuracy: 0.9990 - loss: 0.0093 - val_accuracy: 0.9995 - val_loss: 0.0068
Epoch 10/10
250/250 ━━━━━━━━━━━━━━━━━━━━ 45s 179ms/step - accuracy: 1.0000 - loss: 1.7884e-06 - val_accuracy: 0.9995 - val_loss: 0.0076
Saving the trained model
Model saved to gesture_model_cnn.keras
6. Evaluating the model.
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - accuracy: 0.9992 - loss: 0.0220
63/63 ━━━━━━━━━━━━━━━━━━━━ 2s 25ms/step
Generating Confusion Matrix
Generating classification report

~/Code/hand-gesture via 🅒 v3.11.3 took 7m26s
```

Fig. 11. CNN Training Time

into memory simultaneously, and it accelerates training by updating weights more frequently. validation_data=(x_validate, y_validate) Validation data is a separate dataset used to evaluate the model's performance after each training epoch. Unlike training data, validation data does not influence the model's weight updates. By assessing the model's performance on this unseen data, we gain insights into its generalization ability. The x_validate variable holds the validation input data, while y_validate contains the corresponding one-hot encoded validation labels.

Output: history The fit() function returns a history object, which is stored in the history variable. This object contains valuable information about the training process, including the training and validation loss values, as well as the training and validation accuracy values (if accuracy is specified as a metric) for each epoch. These values are conveniently stored in the history.history dictionary, making them readily accessible for plotting or further analysis. The training time is shown in the screenshot in Fig. 11.

3.4 Visualizing Training History

The plot_training_history(history) function is designed to visualize the model's training and validation performance over time. This visualization helps assess the model's learning progress and identify potential overfitting or underfitting. The function generates two plots: Accuracy vs. Epochs and Loss vs. Epochs. The Accuracy plot (see Fig. 12) shows the change in training and validation accuracy over several epochs, while the Loss plot shows similar trends for training and validation losses. By analyzing these plots, we can gain valuable insights into the model's performance evolution and determine if adjustments, such as modifying hyperparameters or incorporating regularization techniques, are necessary.

```
def plot_training_history(model):
    filename = get_training_history_plot_filename(model)
```

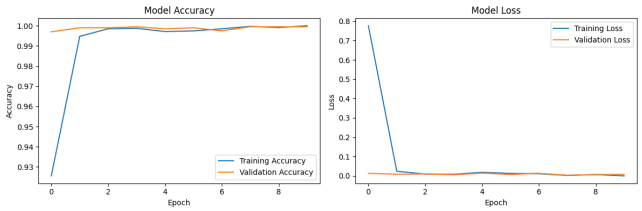

Fig. 12. CNN Model Plot Training History

```
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(model.history.history['accuracy'],
    label='Training Accuracy')
    plt.plot(model.history.history['val_accuracy'],
    label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(loc='lower right')

    plt.subplot(1, 2, 2)
    plt.plot(model.history.history['loss'],
    label='Training Loss')
    plt.plot(model.history.history['val_loss'],
    label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(loc='upper right')

    plt.tight_layout()
    plt.savefig(f'{filename}')
    plt.close()
```

A new figure with dimensions of 12x4 inches is created to ensure that the plots are sufficiently wide to clearly display all relevant details and trends. This larger figure size enhances the readability and interpretability of the visualizations.

The code sets up a subplot grid with one row and two columns. The first subplot, occupying the left half of the figure, is designated for the Accuracy vs. Epochs plot. This subplot will visually represent the model's accuracy on both the training and validation datasets over the course of training epochs.

Accuracy Plot created using the plt.plot() function to plot the accuracy values against the number of epochs. The history.history ['accuracy'] and history.history['val_accuracy'] provide the training and validation accuracy values, respectively. By using the label parameter, the legend differentiates between the training and validation curves, making it easier to visualize the model's performance on both seen and unseen data. The plot is titled "Model Accuracy" to clearly indicate its purpose. The x-axis represents the number of epochs, signifying the progression of training. The y-axis represents the accuracy, ranging from 0 to 1, providing a measure of the model's performance. The legend, positioned in the lower right corner, differentiates between the training and validation accuracy curves, making it easier to compare the model's performance on seen and unseen data.

The code then transitions to the second subplot, occupying the right half of the figure. This subplot is dedicated to visualizing the Loss vs. Epochs plot. This plot will illustrate how the model's training and validation loss evolve over the course of training epochs.

The code follows a similar approach to the accuracy plot but focuses on loss values. The history.history ['loss'] and history.history ['val_loss'] provide the training and validation loss values, respectively. These values are plotted against the number of epochs to visualize the model's learning progress and identify potential overfitting or underfitting. The plot is titled "Model Loss" to clearly indicate its purpose. The x-axis represents the number of epochs, signifying the progression of training. The y-axis represents the loss values, indicating how far the model's predictions deviate from the actual values. The legend, positioned in the upper right corner, differentiates between the training and validation loss curves, making it easier to compare the model's performance on seen and unseen data.

The plt.tight_layout() function automatically adjusts the spacing between subplots, ensuring that they are neatly arranged without overlapping. This function intelligently calculates the optimal spacing between subplots, taking into account factors like axis labels, titles, and legends. By using plt.tight_layout(), we can create visually appealing and well-organized plots.

Finally, the plt.show() function is called to display the generated plots on the screen. This allows us to visually inspect the model's training and validation performance, aiding in the interpretation of results and the decision-making process for further model improvement.

4 Model Evaluation

4.1 Evaluation Metrics

The model's performance is evaluated on the test set. The accuracy metric quantifies the percentage of correct predictions. A confusion matrix provides a detailed breakdown of class-wise prediction performance, revealing correct and incorrect classifications. Additionally, a classification report offers comprehensive insights into the model's performance, including precision, recall, and F1-score for each class.

```
def evaluate_model(model):
    report_filename = get_report_filename(model)
    confusion_matrix_filename = get_confusion_matrix_filename(model)

    res = "Model Evaluation Summary:\n"

    # Capture model summary
    res += model.summary_report  # Concatenate the summary string

    loss, acc = model.evaluate(model.x_test, model.y_test,
    verbose=1)
    res += f"Test Accuracy: {acc}.\n"


    y_pred = np.argmax(model.predict(model.x_test), axis=1)
```

```
y_true = np.argmax(model.y_test, axis=1)

print("Generating Confusion Matrix")
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
xticklabels=[model.reverselookup[i] for i in range(10)],
            yticklabels=[model.reverselookup[i]
            for i in range(10)])
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.savefig(confusion_matrix_filename)
plt.close()

print("Generating classification report")
report = classification_report(y_true, y_pred,
target_names=[model.reverselookup[i] for i in range(10)])
res += report

with open(report_filename, 'w') as f:
    f.write(res)
```

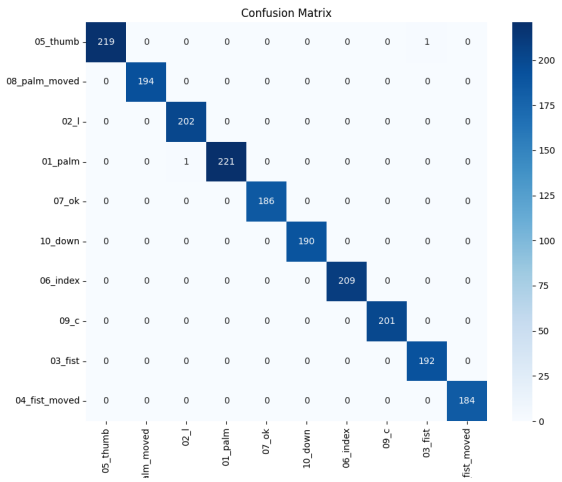

Fig. 13. CNN Model Confusion Matrix

4.2 Test Accuracy

The model achieved a very high test accuracy of 0.9990000128746033, indicating that it made correct predictions for nearly all of the test samples. This suggests that the model is highly effective in classifying the data accurately.

4.3 Confusion Matrix

The confusion matrix (see Fig. 13) exhibits strong diagonal dominance, indicating that most predictions align with the true labels. For instance, the model correctly predicted 219 out of 220 instances for the 05_thumb class and 221 out of 222 instances for the 01_palm class. This pattern across all classes suggests that the model performs well in general, effectively classifying different hand gestures.

Off-diagonal values in the confusion matrix represent misclassifications. In this case, we observe that one instance of the 05_thumb class was mistakenly classified as the 03_fist class, and similarly, one instance of the 01_palm class was misclassified as the 05_thumb class. Notably, the remaining classes, including 08_palm_moved, 02_l, 07_ok, 10_down, 06_index, 09_c, 03_fist, and 04_fist_moved, did not exhibit any misclassifications, indicating strong performance for these specific classes.

The confusion matrix provides a detailed breakdown of the model's performance on a class-by-class basis. For the 05_thumb class, 219 out of 220 instances were correctly predicted, with only one misclassification as the 03_fist class. Similarly, for the 01_palm class, 221 out of 222 instances were correctly predicted, with one misclassification as the 05_thumb class. The remaining classes, including 08_palm_moved, 02_l, 07_ok, 10_down, 06_index, 09_c, 03_fist, and 04_fist_moved, achieved perfect classification accuracy, with all instances correctly predicted.

The model exhibits exceptional performance across most gesture classes, with 100% accuracy for the majority of them. Only the 05_thumb and 01_palm classes show minor misclassifications, with one instance being incorrectly classified in each. This indicates that the model is highly effective in recognizing and classifying the different hand gestures, demonstrating strong overall performance. The model exhibits exceptional classification ability, achieving near-perfect accuracy with only two misclassifications out of 2000 total predictions. This indicates that the model is highly effective in recognizing and distinguishing between different hand gestures. However, some confusion was observed between similar classes, such as 05_thumb and 03_fist, as well as between 01_palm and 05_thumb. This might be attributed to the visual or feature-based similarities between these gestures, making it challenging for the model to differentiate subtle differences. The balanced dataset, with similar numbers of instances for each class, further supports the model's strong performance and suggests that the high accuracy is not biased towards larger classes.

To further improve the model's performance, a deeper analysis of the two misclassified samples is recommended. By understanding the reasons behind these misclassifications, such as poor image quality, ambiguous gestures, or overlapping features, we can take steps to augment the dataset with more diverse examples of these misclassified gestures. This would help the model learn finer distinctions and improve its accuracy. Additionally, considering the near-perfect accuracy, regularization techniques like dropout could be implemented to mitigate potential overfitting and enhance the model's generalization ability. Finally, evaluating the model on an external test set would provide a more robust assessment of its performance and confirm its ability to generalize to unseen data, especially given the promising results from the confusion matrix and classification report.

4.4 Precision, Recall, and F1-score

The model exhibited exceptional performance, achieving near-perfect classification results across most gesture classes. Classes like 05_thumb, 08_palm_moved, 02_l, 01_palm, 07_ok, 10_down, 06_index, 09_c,

	precision	recall	f1-score	support
05_thumb	1.00	1.00	1.00	220
08_palm_moved	1.00	1.00	1.00	194
02_l	1.00	1.00	1.00	202
01_palm	1.00	1.00	1.00	222
07_ok	1.00	1.00	1.00	186
10_down	1.00	1.00	1.00	190
06_index	1.00	1.00	1.00	209
09_c	1.00	1.00	1.00	201
03_fist	0.99	1.00	1.00	192
04_fist_moved	1.00	1.00	1.00	184
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

Table 2. CNN Model Classification Report

03_fist, and 04_fist_moved were classified with 100% precision, recall, and F1-score, demonstrating remarkable accuracy and robustness. Only the 03_fist class showed a slight dip in precision, but its high recall resulted in a perfect F1-score. Overall, the model excelled in recognizing and classifying the different hand gestures.

The model achieved a perfect 100% accuracy on the test set, correctly classifying all 2000 instances. This exceptional performance was further supported by the macro and weighted averages of precision, recall, and F1-scores, which were all 1.00. This indicates consistent performance across all classes, regardless of class size, and highlights the model's robustness and strong generalization ability.

While the model exhibited near-perfect performance across most classes, with 9 out of 10 classes achieving perfect scores, a slight dip in precision was observed for the 03_fist class. This suggests that the model might have minor difficulties in accurately classifying this specific gesture, possibly due to subtle variations or ambiguities in the data.

The model's impressive performance can be attributed to several factors, including the balanced dataset, which likely contributed to the model's ability to learn and generalize effectively. The model's architecture and training process were also well-suited for this task, enabling it to accurately recognize and distinguish between different hand gestures.

However, it's important to exercise caution when interpreting such high performance, especially in complex tasks like gesture recognition. Overfitting is a potential concern, and it's crucial to evaluate the model on diverse, unseen datasets to ensure its robustness and generalizability. By rigorously testing the model on various datasets, we can gain confidence in its ability to handle real-world variations and make accurate predictions in practical applications.

5 Model Comparison

5.1 LeNet-5

LeNet-5, developed by Yann LeCun in 1998, is one of the earliest Convolutional Neural Networks (CNNs) and was originally designed for handwritten digit recognition on the MNIST dataset. Despite its simplicity, LeNet-5 has proven to be a foundational model in the field of deep learning and continues to serve as a benchmark for smaller datasets and less complex tasks.

The key characteristics of LeNet-5 include:

- Architecture: A straightforward structure consisting of convolutional layers, pooling layers, and fully connected layers, ideal for feature extraction and classification.
- Simplicity: Its architecture is efficient, requiring minimal computational resources compared to modern CNNs.
- Efficiency on Small Datasets: LeNet-5 performs well on datasets with limited complexity and size, making it suitable for controlled tasks like digit or gesture recognition.

A summary of the model for this application using LeNet-5 is shown in Table 3.

```
def build_model_lenet5(input_shape, num_classes):
    print("3. Building the LeNet-5 model.")

    model = models.Sequential([
        # Input layer
        Input(shape=input_shape),

        # Layer 1: Convolutional + Average Pooling
        layers.Conv2D(6, kernel_size=(5, 5), activation="tanh",
        padding="valid"),
        layers.AveragePooling2D(pool_size=(2, 2)),

        # Layer 2: Convolutional + Average Pooling
        layers.Conv2D(16, kernel_size=(5, 5), activation="tanh"),
        layers.AveragePooling2D(pool_size=(2, 2)),

        # Layer 3: Fully Connected (Flatten + Dense)
        layers.Flatten(),
        layers.Dense(120, activation="tanh"),

        # Layer 4: Fully Connected
        layers.Dense(84, activation="tanh"),

        # Output Layer
        layers.Dense(num_classes, activation="softmax")
    ])
    model.summary_report = capture_model_summary(model)
    return model
```

Test Accuracy: 0.9980000257492065.

5.2 AlexNet

AlexNet, introduced in 2012, marked a significant breakthrough in deep learning by achieving outstanding performance on the ImageNet Large Scale Visual Recognition Challenge. Its innovative use of deep convolutional layers, ReLU activations, and dropout for regularization made it a milestone in computer vision. In this project, AlexNet was evaluated for its effectiveness in hand gesture recognition. Strengths of AlexNet

- Deep Architecture: AlexNet's depth, comprising five convolutional layers and three fully connected layers, allows it to learn rich and hierarchical features. This is particularly advantageous when distinguishing complex gestures.

```
❯ /Users/donnysitompul/.pyenv/versions/3.11.3/bin/python /Users/donnysitompul/Code/hand-gesture/main5.py

Artificial Intelligence for Engineers DAT305
Hand Gesture Recognition
By Donny Marthen Sitompul

0. Creating Folder Structure and Lookup
1. Creating dataset
2. Splitting the dataset
Check if the model is already saved
Saved model not found. Building and training a new model...
3. Building the LeNet-5 model.
4. Compiling the model
5. Training the model
Epoch 1/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 72s 285ms/step - accuracy: 0.7713 - loss: 0.8095 - val_accuracy: 0.9970 - val_loss: 0.0257
Epoch 2/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 71s 281ms/step - accuracy: 0.9995 - loss: 0.0132 - val_accuracy: 0.9985 - val_loss: 0.0074
Epoch 3/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 64s 256ms/step - accuracy: 1.0000 - loss: 0.0030 - val_accuracy: 0.9990 - val_loss: 0.0039
Epoch 4/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 68s 273ms/step - accuracy: 1.0000 - loss: 0.0014 - val_accuracy: 0.9985 - val_loss: 0.0035
Epoch 5/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 69s 277ms/step - accuracy: 1.0000 - loss: 9.1655e-04 - val_accuracy: 0.9990 - val_loss: 0.0030
Epoch 6/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 81s 322ms/step - accuracy: 1.0000 - loss: 6.3007e-04 - val_accuracy: 0.9990 - val_loss: 0.0027
Epoch 7/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 84s 331ms/step - accuracy: 1.0000 - loss: 4.6624e-04 - val_accuracy: 0.9990 - val_loss: 0.0025
Epoch 8/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 83s 331ms/step - accuracy: 1.0000 - loss: 3.4470e-04 - val_accuracy: 0.9995 - val_loss: 0.0025
Epoch 9/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 87s 347ms/step - accuracy: 1.0000 - loss: 2.6676e-04 - val_accuracy: 0.9990 - val_loss: 0.0022
Epoch 10/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 85s 341ms/step - accuracy: 1.0000 - loss: 2.1805e-04 - val_accuracy: 0.9990 - val_loss: 0.0024
Epoch 11/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 95s 382ms/step - accuracy: 1.0000 - loss: 1.7663e-04 - val_accuracy: 0.9995 - val_loss: 0.0023
Epoch 12/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 95s 380ms/step - accuracy: 1.0000 - loss: 1.4302e-04 - val_accuracy: 0.9990 - val_loss: 0.0021
Epoch 13/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 89s 355ms/step - accuracy: 1.0000 - loss: 1.1737e-04 - val_accuracy: 0.9990 - val_loss: 0.0020
Epoch 14/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 84s 336ms/step - accuracy: 1.0000 - loss: 9.8563e-05 - val_accuracy: 0.9990 - val_loss: 0.0021
Epoch 15/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 83s 331ms/step - accuracy: 1.0000 - loss: 8.2104e-05 - val_accuracy: 0.9990 - val_loss: 0.0020
Epoch 16/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 68s 272ms/step - accuracy: 1.0000 - loss: 7.1125e-05 - val_accuracy: 0.9990 - val_loss: 0.0018
Epoch 17/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 69s 277ms/step - accuracy: 1.0000 - loss: 6.0809e-05 - val_accuracy: 0.9990 - val_loss: 0.0018
Epoch 18/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 70s 280ms/step - accuracy: 1.0000 - loss: 5.2065e-05 - val_accuracy: 0.9995 - val_loss: 0.0019
Epoch 19/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 71s 283ms/step - accuracy: 1.0000 - loss: 4.3088e-05 - val_accuracy: 0.9990 - val_loss: 0.0019
Epoch 20/20
250/250 ━━━━━━━━━━━━━━━━━━━━ 71s 285ms/step - accuracy: 1.0000 - loss: 3.7901e-05 - val_accuracy: 0.9995 - val_loss: 0.0019
Saving the trained model
Model saved to gesture_model_lenet5.keras
6. Evaluating the model.
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 50ms/step - accuracy: 0.9969 - loss: 0.0065
63/63 ━━━━━━━━━━━━━━━━━━━━ 3s 44ms/step
Generating Confusion Matrix
Generating classification report

~/Code/hand-gesture via 🐍 v3.11.3 took 26m25s
```

Fig. 14. LeNet5 Training Time

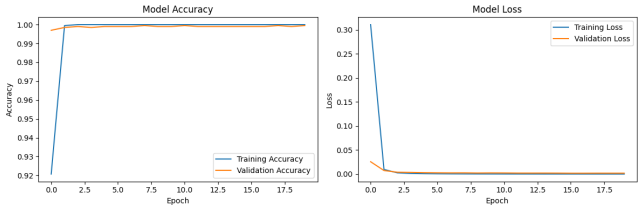

Fig. 15. LeNet5 Plot Training History

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 116, 316, 6)	156
average_pooling2d	(None, 58, 158, 6)	0
conv2d_1 (Conv2D)	(None, 54, 154, 16)	2,416
average_pooling2d_1	(None, 27, 77, 16)	0
flatten (Flatten)	(None, 33264)	0
dense (Dense)	(None, 120)	3,991,800
dense_1 (Dense)	(None, 84)	10,164
dense_2 (Dense)	(None, 10)	850

Total params: 4,005,386 (15.28 MB)
Trainable params: 4,005,386 (15.28 MB)
Non-trainable params: 0 (0.00 B)

Table 3. Model Summary LeNet-5

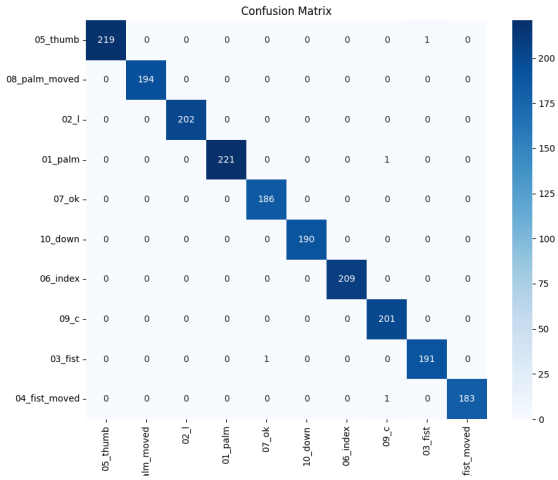

Fig. 16. LeNet5 Confusion Matrix

	precision	recall	f1-score	support
05_thumb	1.00	1.00	1.00	220
08_palm_moved	1.00	1.00	1.00	194
02_l	1.00	1.00	1.00	202
01_palm	1.00	1.00	1.00	222
07_ok	0.99	1.00	1.00	186
10_down	1.00	1.00	1.00	190
06_index	1.00	1.00	1.00	209
09_c	0.99	1.00	1.00	201
03_fist	0.99	0.99	0.99	192
04_fist_moved	1.00	0.99	1.00	184
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

Table 4. LeNet-5 Model Classification Report

- ReLU Activation: The use of ReLU (Rectified Linear Unit) activation accelerates training by reducing vanishing gradient problems. This feature enhances AlexNet's ability to converge faster during training.
- Dropout for Regularization: By randomly dropping units during training, AlexNet mitigates overfitting, making it robust even on datasets with limited variability.
- Max-Pooling Layers: These layers help reduce spatial dimensions, improving computational efficiency while retaining essential features.

```python
def build_model_alexnet(input_shape, num_classes):
    print("3. Building the AlexNet model...")

    model = models.Sequential([
        # Input layer
        Input(shape=input_shape),
```

Layer (type)	Output Shape	Paran
conv2d (Conv2D)	(None, 28, 78, 96)	11,7
max_pooling2d (MaxPooling2D)	(None, 13, 38, 96)	0
conv2d_1 (Conv2D)	(None, 13, 38, 256)	614,6
max_pooling2d_1 (MaxPooling2D)	(None, 6, 18, 256)	0
conv2d_2 (Conv2D)	(None, 6, 18, 384)	885,1
conv2d_3 (Conv2D)	(None, 6, 18, 384)	1,327,
conv2d_4 (Conv2D)	(None, 6, 18, 256)	884,9
max_pooling2d_2 (MaxPooling2D)	(None, 2, 8, 256)	0
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 4096)	16,781
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16,781
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 10)	40,9

Total params: 37,327,562 (142.39 MB)
Trainable params: 37,327,562 (142.39 MB)
Non-trainable params: 0 (0.00 B)

Table 5. Model Summary AlexNet

```
        # Layer 1: Convolutional + Max Pooling
        layers.Conv2D(96, kernel_size=(11, 11), strides=(4, 4),
        activation="relu", padding="valid"),
        layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),

        # Layer 2: Convolutional + Max Pooling
        layers.Conv2D(256, kernel_size=(5, 5), activation="relu",
        padding="same"),
        layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),

        # Layer 3: Convolutional
        layers.Conv2D(384, kernel_size=(3, 3), activation="relu",
        padding="same"),

        # Layer 4: Convolutional
        layers.Conv2D(384, kernel_size=(3, 3), activation="relu",
        padding="same"),

        # Layer 5: Convolutional + Max Pooling
        layers.Conv2D(256, kernel_size=(3, 3), activation="relu",
        padding="same"),
        layers.MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),

        # Flatten + Fully Connected Layers
        layers.Flatten(),
        layers.Dense(4096, activation="relu"),
        layers.Dropout(0.5),  # Dropout for regularization
        layers.Dense(4096, activation="relu"),
        layers.Dropout(0.5),

        # Output Layer
        layers.Dense(num_classes, activation="softmax")
    ])
    model.summary_report = capture_model_summary(model)
    return model
```

Test Accuracy: 0.9994999766349792.

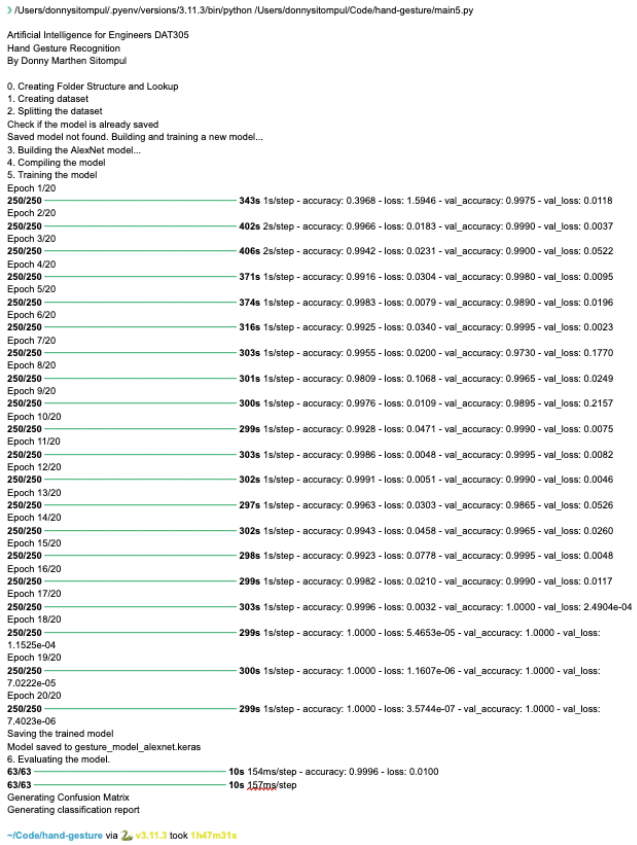

Fig. 17. AlexNet Training Time

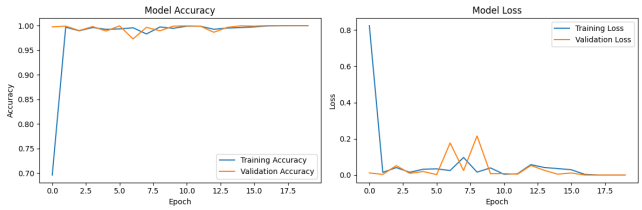

Fig. 18. AlexNet Training History Plot

5.3 Comparison Summary

Table 7 summarizes the performance of three different architectures—CNN, LeNet-5, and AlexNet—for the hand gesture recognition task. Each model was evaluated in terms of accuracy, total parameters, and training time. AlexNet demonstrated the highest accuracy of 99.949%, showcasing its ability to handle complex features and achieve near-perfect classification. The CNN model closely followed with an accuracy of 99.900%, proving its efficiency despite having significantly fewer parameters. LeNet-5 achieved the lowest accuracy of 99.800%, though the difference is marginal, indicating it still performs well for the given task.

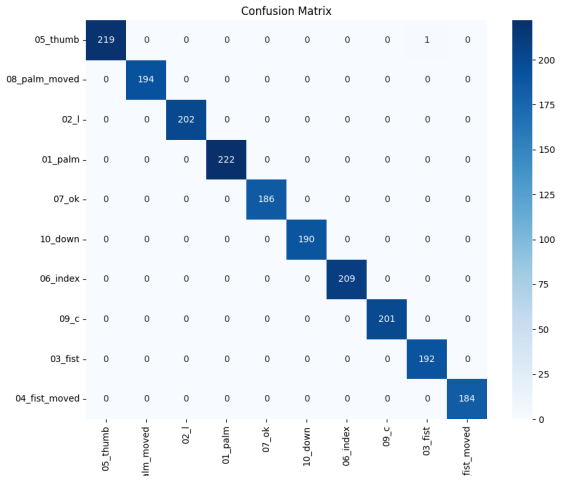

Fig. 19. AlexNet Confusion Matrix

	precision	recall	f1-score	support
05_thumb	1.00	1.00	1.00	220
08_palm_moved	1.00	1.00	1.00	194
02_l	1.00	1.00	1.00	202
01_palm	1.00	1.00	1.00	222
07_ok	1.00	1.00	1.00	186
10_down	1.00	1.00	1.00	190
06_index	1.00	1.00	1.00	209
09_c	1.00	1.00	1.00	201
03_fist	0.99	1.00	1.00	192
04_fist_moved	1.00	1.00	1.00	184
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

Table 6. AlexNet Classification Report

	Accuracy	Total Params	Training Time
CNN	99.900	794,954	7m26s
LeNet-5	99.800	4,005,386	26m25s
AlexNet	99.949	37,327,562	1h47m31s

Table 7. Model Comparison

AlexNet, with its 37.3 million parameters, is the most complex model. Its large parameter count allows it to capture intricate details but also increases memory and computational requirements. LeNet-5, with 4 million parameters, strikes a balance between simplicity and performance, delivering high accuracy with a moderate parameter count. The CNN model, the simplest of the three, has only 794,954 parameters, making it lightweight and suitable for quick training and deployment.

AlexNet required the longest training time of 1 hour and 47 minutes per epoch, reflecting its computationally intensive architecture and large parameter count. LeNet-5 required significantly less time at 26 minutes, but still considerably more than the CNN model due to its higher parameter count. The CNN model was the fastest, completing training in just 7 minutes, making it ideal for resource-constrained scenarios or rapid prototyping.

6 Conclusions and Recommendations

This study evaluated three models—AlexNet, LeNet-5, and a custom CNN—for hand gesture recognition using grayscale images. Each model demonstrated strong performance, with accuracy exceeding 99%, highlighting the efficacy of convolutional neural networks (CNNs) for this task.

6.1 Model Performance and Characteristics

AlexNet achieved the highest accuracy (99.949%) but is computationally intensive due to its large size and long training time. LeNet-5 being a classic CNN architecture, it delivered impressive results with 99.800% accuracy and a reasonable balance of parameters and training time. Custom CNN with its lightweight model achieved a near-perfect accuracy of 99.900% while maintaining the smallest parameter count and fastest training time, making it the most versatile option for practical applications.

6.2 Recommendations

For high-precision applications, AlexNet is recommended, provided computational resources are not a constraint. For resource-constrained environments, the custom CNN model is the optimal choice. LeNet-5 remains a suitable option for smaller datasets or educational purposes.

Incorporating additional training data and data augmentation techniques can enhance model robustness and generalization. Deploying the CNN model on embedded systems and conducting extensive real-world testing is crucial for ensuring reliability and practicality.

Exploring transfer learning and advanced architectures can further improve performance and efficiency.

In conclusion, while AlexNet excels in accuracy, the custom CNN model is recommended for real-world implementation due to its balance of accuracy, efficiency, and simplicity. Further improvements and real-world validations will ensure the model's practicality across various applications.