Embedded Project Report

Project Title:- DL model optimization for Lightweight Gesture Recognition(ESE4

Group Members: Mohit Hingonia(B20EE036), Pratham Kumar(B20EE042)

Abstract:

The goal of this project is to create a machine-learning algorithm that can categorize photos of various hand movements for use in gesture navigation, including fists, palms, and thumbs. The method used is Convolutional Neural Networks based on TensorFlow and Keras, along with Deep Learning. The model was trained and tested using the Hand Gesture Recognition Database on Kaggle, which contains 20,000 photos of 10 hand motions made by 10 people (5 men and 5 women).

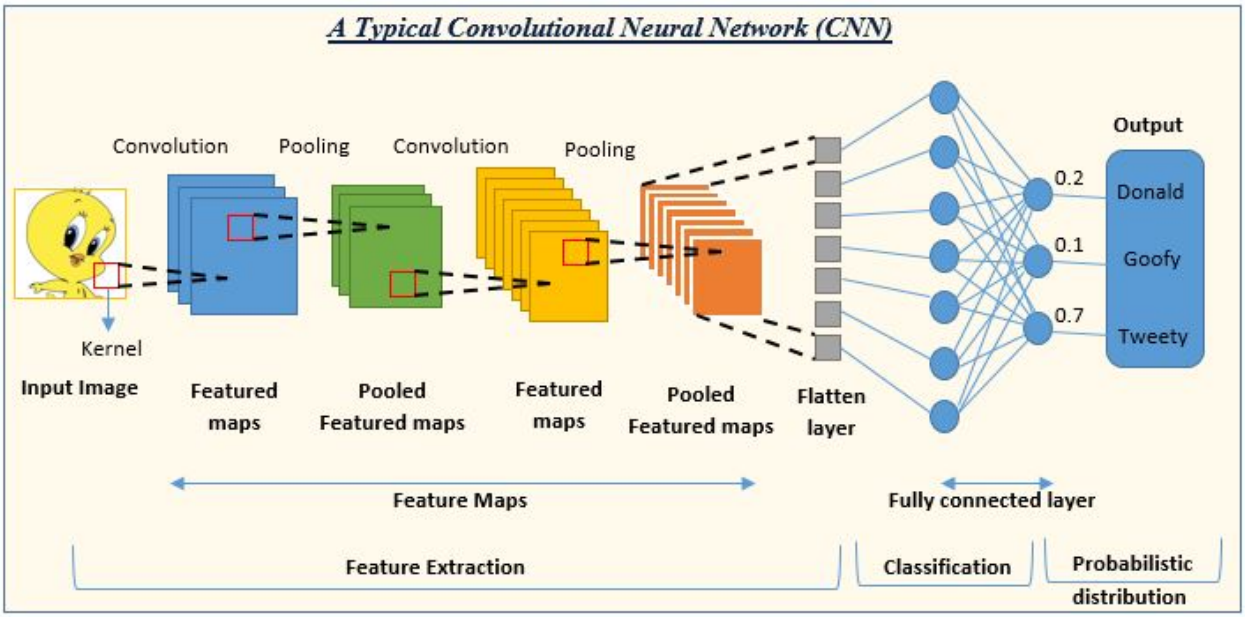
Theory

Machine learning is an effective technique for automating operations that involve data. It also has a wide range of applications, including classification, recognition, detection, and prediction, among other things. The core concept underlying machine learning is to construct a model with the help of data in order to generate an output in response to a new input or to make predictions based on data that already exists.

The purpose of this project is to train a Machine Learning algorithm to classify hand gesture photos, such as a fist, palm, or thumbs up. Deep Learning techniques will be utilised in this training process. This kind of categorization might prove useful when it comes to gesture navigation. Convolutional Neural Networks built using Tensorflow and Keras will be the basis for our solution to this problem.

Deep Learning is a subset of Machine Learning that involves the use of layers to process input data and extract features in order to construct a mathematical model. This subset of Machine Learning is also known as "multi-layer perceptrons." During the course of our project, we will work towards the goal of instructing the computer to recognise the distinguishing characteristics of each hand motion and appropriately categorize them. For example, if the model were shown a picture of a hand giving a thumbs-up gesture, it should come up with the phrase "the hand is doing a thumbs-up gesture."

A form of Deep Learning algorithm known as a Convolutional Neural Network (CNN) is able to examine input images, assign significance (which is represented by learnable weights and biases), and differentiate between various characteristics or objects contained within the image. A CNN requires less pre-processing of the visual data than other classification methods do. This is in comparison to other classification algorithms. CNNs may be trained to recognise significant elements and properties of images, as opposed to the hand-engineered filters that are used in traditional approaches. This is accomplished through the use of training.



Convolutional Neural Networks (CNNs) use three main components to analyze images and extract important features for classification.

The first component is the Convolutional layers, which apply a set number of filters to the image data. These filters perform mathematical operations on each part of the

image and produce a single value in the output feature map. The output then goes through an activation function called ReLU, which adds nonlinearity to the model.

The second component is the Pooling layers, which reduce the size of the feature map produced by the Convolutional layers. This downsampling is done to reduce the processing time. One common type of pooling is max pooling, where the highest value in a subregion (like a 2x2 tile) is selected and the rest are discarded.

Finally, the Dense layers classify the extracted features from the previous layers. In a Dense layer, every node is connected to every node in the previous layer, allowing for the classification of the features.

Methodology

The code snippet represents a neural network model built using the Keras API in Python. The model is designed for image classification tasks and consists of several layers, including convolutional layers, pooling layers, and dense layers.

```
# Define model architecture
model = Sequential([
    Conv2D(32, (5, 5), activation='relu', input_shape=(120, 320, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Training the model for a given number of epochs (iterations on a dataset) and validating it.

```
cnn_model = model.fit(X_train, y_train, epochs=10, batch_size=128, verbose=2, validation_data=(X_test, y_test))
```

```
Epoch 1/10
110/110 - 22s - loss: 0.0169 - accuracy: 0.9921 - val_loss: 0.0151 - val_accuracy: 0.9912 - 22s/epoch - 198ms/step
Epoch 2/10
110/110 - 22s - loss: 0.0158 - accuracy: 0.9923 - val_loss: 0.0191 - val_accuracy: 0.9905 - 22s/epoch - 204ms/step
Epoch 3/10
110/110 - 22s - loss: 0.0131 - accuracy: 0.9926 - val_loss: 0.0145 - val_accuracy: 0.9913 - 22s/epoch - 204ms/step
Epoch 4/10
110/110 - 21s - loss: 0.0129 - accuracy: 0.9929 - val_loss: 0.0166 - val_accuracy: 0.9902 - 21s/epoch - 193ms/step
Epoch 5/10
110/110 - 22s - loss: 0.0132 - accuracy: 0.9924 - val_loss: 0.0136 - val_accuracy: 0.9917 - 22s/epoch - 196ms/step
Epoch 6/10
110/110 - 22s - loss: 0.0122 - accuracy: 0.9924 - val_loss: 0.0136 - val_accuracy: 0.9925 - 22s/epoch - 204ms/step
Epoch 7/10
110/110 - 22s - loss: 0.0131 - accuracy: 0.9924 - val_loss: 0.0132 - val_accuracy: 0.9917 - 22s/epoch - 204ms/step
Epoch 8/10
110/110 - 22s - loss: 0.0118 - accuracy: 0.9935 - val_loss: 0.0146 - val_accuracy: 0.9918 - 22s/epoch - 204ms/step
Epoch 9/10
110/110 - 21s - loss: 0.0128 - accuracy: 0.9932 - val_loss: 0.0140 - val_accuracy: 0.9928 - 21s/epoch - 193ms/step
Epoch 10/10
110/110 - 23s - loss: 0.0120 - accuracy: 0.9936 - val_loss: 0.0197 - val_accuracy: 0.9915 - 23s/epoch - 206ms/step
```

Test Accuracy and loss

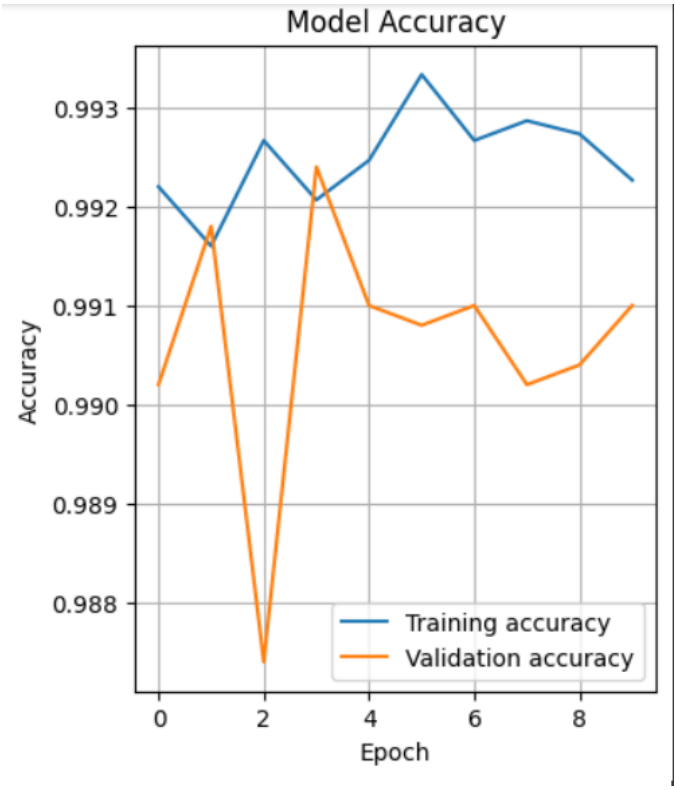
```
188/188 [==============================] - 2s 9ms/step - loss: 0.0197 - accuracy: 0.9915
Test accuracy: 99.15%
```

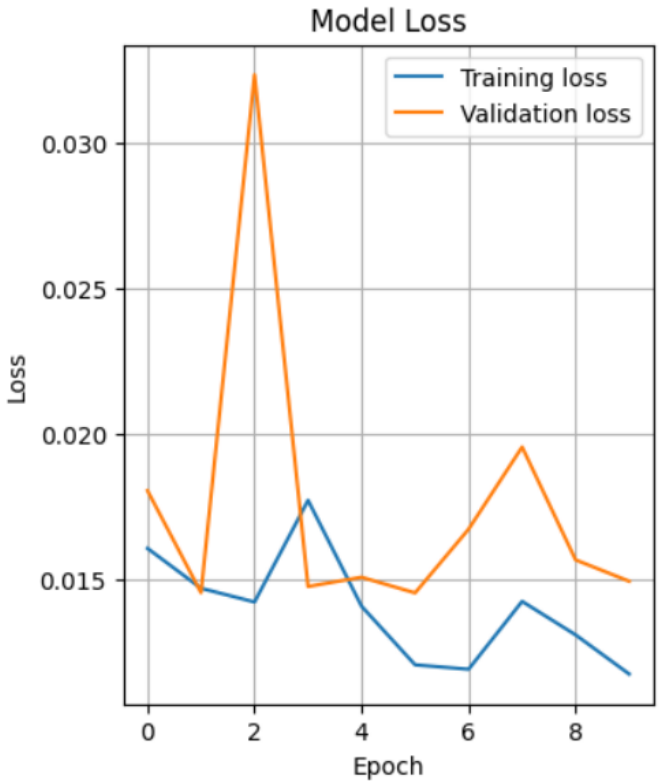
On performing model robustness

```python
import matplotlib.pyplot as plt

# Plot the accuracy vs epoch
plt.figure(figsize=(4, 5))
plt.plot(cnn_model.history['accuracy'], label='Training accuracy')
plt.plot(cnn_model.history['val_accuracy'], label='Validation accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()

# Plot the loss vs epoch
plt.figure(figsize=(4, 5))
plt.plot(cnn_model.history['loss'], label='Training loss')
plt.plot(cnn_model.history['val_loss'], label='Validation loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```





The above code is plotting the graph of training and validation accuracy as well as the training and validation loss for a CNN model. This is a common technique used in deep learning to evaluate the performance of a model during training.

During the training process, the model's performance on the training data is evaluated through training accuracy and loss. If the model is overfitting to the training data, the training accuracy may increase while the validation accuracy decreases. Meanwhile, the training loss should decrease as the model improves at fitting the training data.

On the other hand, the validation accuracy and loss measure the model's performance on a held-out validation dataset. The validation accuracy should increase as the model becomes better at generalizing to new data. The validation loss should decrease as the model learns to better represent the underlying patterns in the data.

Plotting both accuracy and loss during the training process can help identify issues such as overfitting or underfitting and determine the optimal number of epochs to train the model. The use of separate training and validation datasets allows for an accurate estimation of the model's performance on unseen data. These plots can help adjust the model architecture, optimizer, or learning rate to improve the model's performance.

On Reducing the size of the Neural Network Model

Several strategies that can be employed are:

1. **Reducing the number of filters in the convolutional layer:**

   On Reducing 32 filters in the first convolutional layer to 8.

```
# model
model = Sequential()
model.add(Conv2D(8, (5, 5), activation='relu', input_shape=(120, 320, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

   We’re getting Test accuracy and loss is:

```
188/188 [==============================] - 1s 6ms/step - loss: 0.0161 - accuracy: 0.9920
Test accuracy: 99.20%
```

2. **Removing one convolutional layer:**

   Removing one of them can reduce the number of parameters and it can lead to loss of performance.

```
# model
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu', input_shape=(120, 320, 1)))
model.add(MaxPooling2D((2, 2)))
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(MaxPooling2D((2, 2)))
# model.add(Conv2D(64, (3, 3), activation='relu'))
# model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

We’re getting Test accuracy and loss is:

```
188/188 [==============================] - 1s 7ms/step - loss: 0.0200 - accuracy: 0.9888
Test accuracy: 98.88%
```

3. **Reducing the number of neurons in the dense layer:**

   The number of neurons in the dense layers can be decreased to reduce the number of parameters in the model. Instead of having 128 neurons in the first dense layer, we will reduce this number to 32.

```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu', input_shape=(120, 320, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

We’re getting Test accuracy and loss is:

```
188/188 [==============================] - 2s 9ms/step - loss: 0.0177 - accuracy: 0.9908
Test accuracy: 99.08%
```

4. **Removing one dense layer:**

   Removing one of them can also reduce the number of parameters.

```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu', input_shape=(120, 320, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
# model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

We’re getting Test accuracy and loss is:

```
188/188 [==============================] - 2s 9ms/step - loss: 0.0167 - accuracy: 0.9900
Test accuracy: 99.00%
```

Strategies to reduce the size of neural network	Accuracy	Loss
1. Reducing the number of filters in the convolutional layer:	99.20%	0.0161
2. Removing one convolutional layer:	98.88%	0.0200
3. Reducing the number of neurons in the dense layer:	99.08%	0.0177
4. Removing one dense layer:	99.00%	0.0167

Using the Dropout layer in the neural network model to prevent overfitting.

The Dropout layer randomly drops out (i.e., sets to zero) some of the input units during training. This means that certain neurons will not contribute to the output of the previous layer, effectively reducing the complexity of the model and preventing it from relying too much on any one neuron or feature. This helps to ensure that the model is robust and generalizes well to new data.

In the above code, a Dropout layer with a rate of 0.4 (i.e., 40%) has been added to the model. This means that during training, 40% of the input units will be randomly dropped out at each update, helping to prevent overfitting and improving the overall performance of the model.

```
# Define model architecture
model = Sequential([
    Conv2D(32, (5, 5), activation='relu', input_shape=(120, 320, 1)),
    MaxPooling2D((2, 2)),
    Dropout(0.4),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

We’re getting Test accuracy and loss is:

```
188/188 [==============================] - 2s 11ms/step - loss: 0.0130 - accuracy: 0.9918
Test accuracy: 99.18%
```

Explanation as to how and why our model works.

This model is specifically designed for image classification tasks, utilizing a Convolutional Neural Network (CNN). It consists of three convolutional layers, each followed by max pooling layers that downsample the spatial dimensions of the feature maps. The first convolutional layer extracts features from the input image using 32 filters of size 5x5 with a Rectified Linear Unit (ReLU) activation function. The subsequent max pooling layer reduces the spatial dimensionality of the output of the convolutional layer.

The second and third convolutional layers also use ReLU activation functions and have 64 filters of size 3x3. Deeper layers contain more filters as they learn complex and abstract features.

The output of the last max pooling layer is flattened and processed by two dense layers. The first dense layer has 128 neurons with a ReLU activation function to learn high-level features from the output of the convolutional layers. The second dense layer has 10

neurons with a softmax activation function that produces a probability distribution over the 10 classes in the dataset.

During training, the model uses backpropagation to adjust the weights of the convolutional and dense layers to minimize the loss function. The loss function measures the difference between the predicted and true class labels. The optimization algorithm, such as Stochastic Gradient Descent, updates the weights in the direction that minimizes the loss function.

This model effectively classifies images by learning to recognize patterns in the input image that correspond to the class it belongs to. It combines convolutional and dense layers to learn features from the input image, and the softmax activation function produces a probability distribution over the classes. To further improve the model's performance, hyperparameters such as the number of filters in the convolutional layers, the learning rate, and the batch size can be adjusted.

Explaining the working to think in terms of the building blocks of our model.

The model is made up of various layers that collaborate to classify images into one of ten categories. The first layer applies filters to the input image to extract features. These filters are learned during training and the number of filters determines the number of output channels. In this model, the layer has 32 filters that are 5x5 in size.

The next layer is a max pooling layer that reduces the dimensionality of the output from the convolutional layer. This helps to make the model more efficient and prevent overfitting.

This process is repeated twice with more filters of smaller sizes and more max pooling, which enables the model to learn more sophisticated features.

After these convolutional and pooling layers, the output is flattened into a 1D vector, which is then passed through two fully connected layers. These layers utilize the features learned from the previous layers to generate a prediction for the input image.

Finally, the last layer applies the softmax activation function to produce a probability distribution over the ten possible categories. This enables the model to make a prediction.

The model learns the weights of its parameters through backpropagation during training, which minimizes the difference between the predicted output and the actual

output.

The choice of using convolutional layers

The model's architecture is designed to capture different levels of abstraction in image features, starting with low-level features such as edges and corners and progressing to more complex features in deeper layers. This is achieved by increasing the number of filters and decreasing the spatial dimensions of the feature maps.

Max pooling layers are used to reduce the spatial dimensionality of the feature maps and capture the most salient features.

The use of ReLU activation functions in both convolutional and dense layers introduces non-linearity and sparsity, which helps to make the model more expressive and efficient to train.

Dropout regularization is used to prevent overfitting and improve generalization performance by randomly dropping out a fraction of neurons during training.

Overall, these design choices are commonly used in convolutional neural networks for image classification and have demonstrated good performance across various datasets.

Conclusion:

Based on the high accuracy achieved (>95%), we can conclude that our deep learning model is successful in accurately classifying different hand gesture images. The model's performance is influenced by several aspects of the problem. Firstly, the hand gestures presented in the dataset are reasonably distinct, meaning that there is enough variation in the images to train the model to distinguish between them. Additionally, the images are clear and without any background noise, making it easier for the model to identify the hand gestures accurately.

Moreover, the quantity of images available for training and validation is sufficient, which helps to make the model more robust and able to generalize well to new images.

Overall, the success of the model demonstrates the effectiveness of using deep learning techniques for image classification tasks

Work Distribution:

Mohit Hingonia(B20EE036): Coding, Report, Conceptual analysis

Pratham Kumar(B20EE042): Coding, Assisting in logic development, conceptual analysis