Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Vision-ENCS5343

Project

Text Identification Using Local Feature Extraction Arabic Handwritten Techniques

---

Prepared by:

Student Name	Student NO.
Jana Qutosa	1210331
Shiyar Dar-mousa	1210766

Instructor: Dr.Aziz Qaroush.

Section: 1.

Date: 23/1/2025

Abstract

This Project explores the application of **deep learning techniques** to solve a classification problem, focusing on building, training, and evaluating convolutional neural networks (CNNs). The project is divided into four key tasks.

In **Task 1**, a custom CNN architecture is designed, trained, and optimized through hyperparameter tuning. The network's architecture includes convolutional, pooling, and fully connected layers, while training hyperparameters such as learning rate, batch size, and epochs are adjusted for optimal performance. The results of this task are visualized using accuracy and loss curves.

Task 2 involves enhancing the performance of the selected CNN from Task 1 by applying data augmentation techniques. These techniques are tailored to the provided dataset and aim to improve model generalization. A comparison of the augmented model's performance with the baseline from Task 1 is presented.

In **Task 3**, a well-known published CNN architecture is chosen, trained with augmented data, and evaluated. The performance of this model is compared against the results from Task 2 to highlight the benefits of using a pre-existing architecture.

Task 4 applies transfer learning by fine-tuning a pretrained CNN on the given dataset. The performance of the fine-tuned model is analyzed and compared to that of the published architecture in Task 3.

The report concludes with a comprehensive analysis of the results, including the impact of custom networks, data augmentation, published CNNs, and transfer learning. Accuracy and loss curves, along with visual comparisons, illustrate the effectiveness of each approach. The findings highlight the trade-offs between custom architectures and pretrained models, emphasizing the importance of data augmentation and transfer learning in improving deep learning performance.

Table of Contents

Introduction

Convolutional neural networks (CNNs)

A Convolutional Neural Network (CNN) is an advanced form of an Artificial Neural Network (ANN) specifically designed to process and extract features from grid-like data structures. These are commonly used in tasks involving visual datasets, such as images or videos, where identifying patterns and spatial hierarchies is critical to understanding the data.

Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

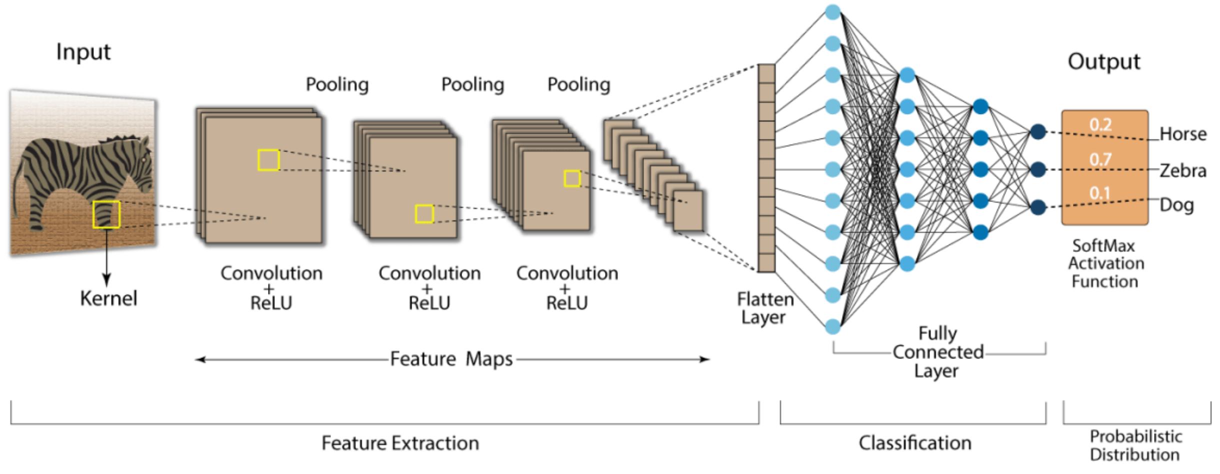


Figure 1: Convolutional neural networks (CNNs)

The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

Layers Used to Build Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN), also known as ConvNet, is composed of a sequence of layers, each transforming the input into more abstract representations through differentiable functions. These layers work together to process and analyze grid-like data, such as images. Let's break down the layers involved in building a ConvNet using an example of a 32x32x3 image (width, height, and depth).

1. **Input Layer**

The input layer serves as the starting point of the network, holding the raw input data. In the case of CNNs, the input is typically an image or a series of images. For an image with dimensions 32x32x3, the width and height are 32 pixels, and the depth represents the three color channels (red, green, and blue). This layer prepares the data for further processing by the network.

2. **Convolutional Layer**

The convolutional layer is responsible for feature extraction. It applies a set of learnable filters, often called kernels, to the input data. These filters are small matrices, typically 2x2, 3x3, or 5x5, that slide over the input image to compute the dot product between the kernel's weights and the corresponding patch of the image. The result is a set of feature maps that highlight specific patterns, such as edges or textures. For instance, if 12 filters are applied to the input image, the resulting output will have dimensions 32x32x12, where 12 represents the number of feature maps.

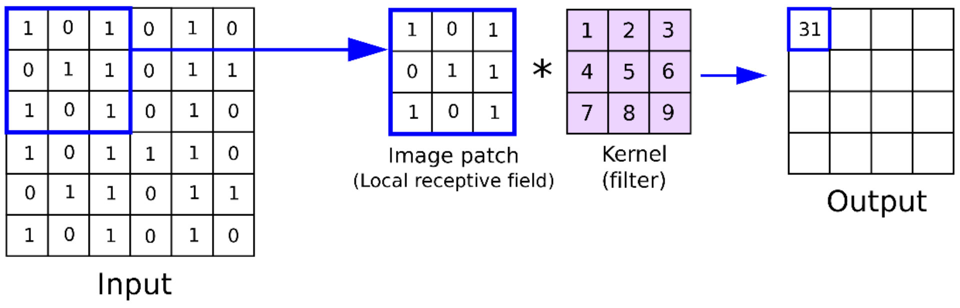


Figure 2: Convolutional Layer

3. **Activation Layer**

The activation layer introduces nonlinearity into the network, allowing it to learn complex patterns. An activation function is applied element-wise to the output of the convolutional layer. Common activation functions include: ReLU (Rectified Linear Unit)**,** Tanh and Leaky ReLU.

The output volume retains the same dimensions as the input; for example, a 32x32x12 volume remains unchanged after the activation layer.

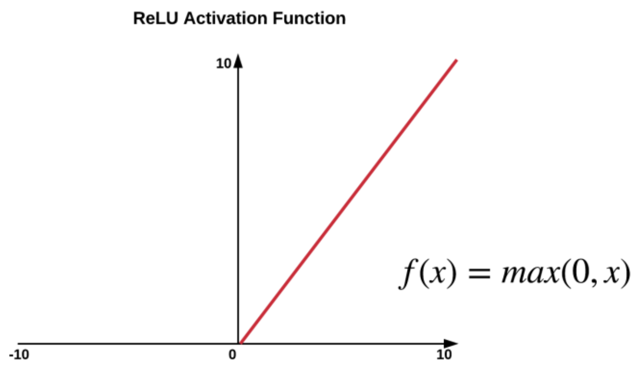


Figure 3: RELU activation function

4. **Pooling Layer**

The pooling layer is used to downsample the feature maps, reducing their spatial dimensions. This helps decrease computation, saves memory, and prevents overfitting. Two common types of pooling are:

Max Pooling: Takes the maximum value from each patch.

Average Pooling: Computes the average value of each patch.

For example, using a 2x2 max pooling layer with a stride of 2 reduces the dimensions of a 32x32x12 feature map to 16x16x12.

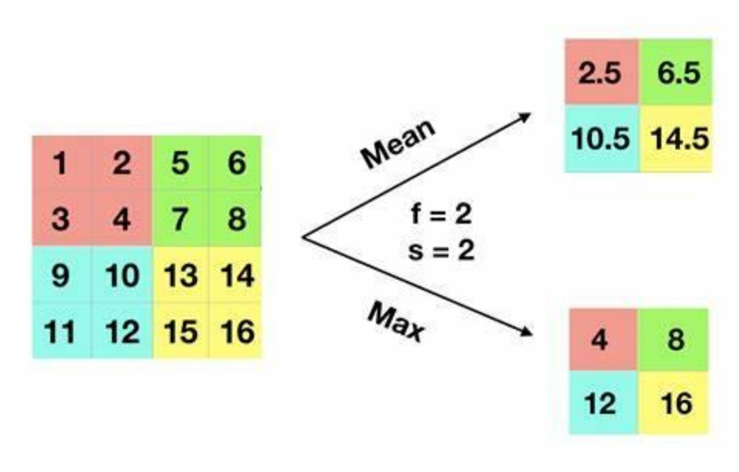


Figure 4: Pooling Layer

5. **Flattening Layer**

After the convolution and pooling layers, the resulting feature maps are flattened into a one-dimensional vector. This step prepares the data for fully connected layers by converting spatial features into a format suitable for classification or regression tasks.

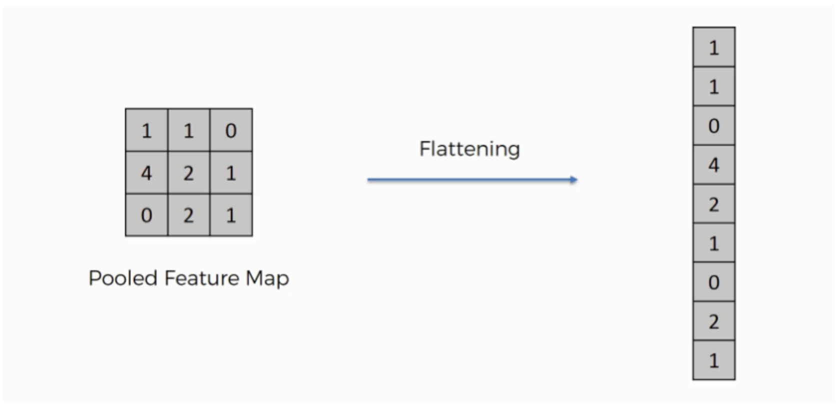


Figure 5: Flattening Layer

6. **Fully Connected Layer**

The fully connected layer processes the flattened input and performs the final task, such as classification or regression. Each neuron in this layer is connected to every element of the previous layer, enabling the network to learn relationships between high-level features.

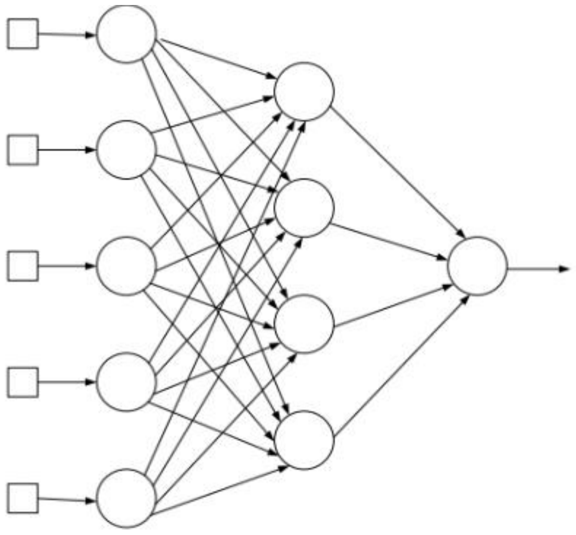


Figure 6: Fully Connected Layer

7. **Output Layer**

The output layer produces the final predictions. For classification tasks, this layer uses activation functions like **softmax** or **sigmoid** to convert the output into probabilities for each class. For example, in a multi-class classification problem, softmax ensures that the probabilities across all classes sum to 1, providing a clear output for decision-making.

Data Augmentation

Data augmentation is a powerful technique used to expand the size and diversity of a training dataset by creating modified versions of existing data. It involves applying transformations such as rotation, scaling, flipping, zooming, or adding noise to the original dataset, thereby generating new variations of the data points. These modifications simulate real-world variations that might occur in unseen data, helping the model learn more robust and generalizable patterns. This approach is particularly valuable when working with small or imbalanced datasets, where the lack of diversity in data can lead to overfitting and poor generalization.

By increasing the variety of training examples, data augmentation improves the model's ability to handle variations and noise, making it more adaptable to different scenarios. It reduces overfitting by exposing the model to a broader range of examples without the need for expensive and time-consuming data collection. This ultimately leads to higher accuracy and better performance on unseen data. Data augmentation is a cost-effective and efficient method to enhance the reliability and robustness of machine learning models, ensuring they perform well even under challenging conditions.

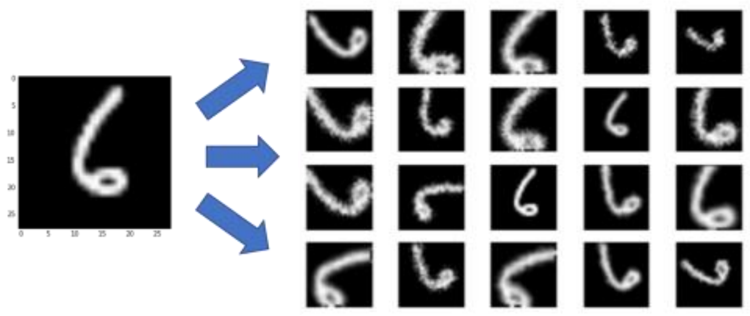


Figure 7: Data Augmentation

Experimental Setup and Results

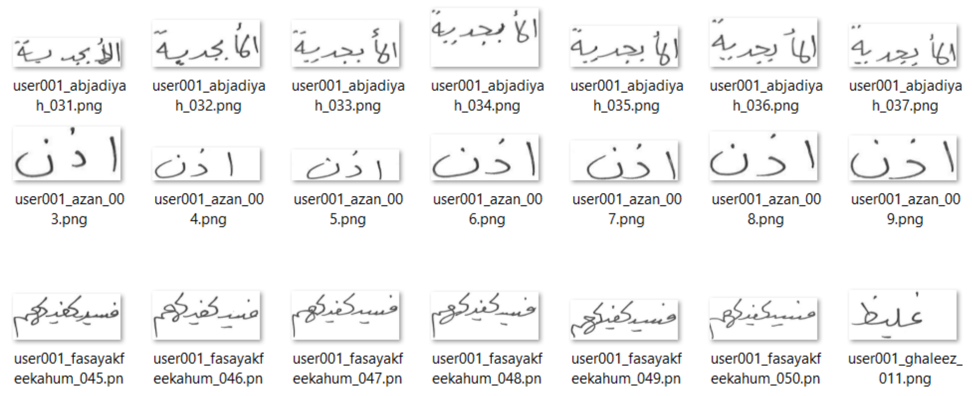
Dataset and Preprocessing

Dataset Overview

The AHAWP dataset, specifically designed for Arabic handwritten text recognition, consists of 8,144 word images from 82 writers. Each writer contributes 10 samples for each of 10 unique Arabic words. This dataset’s diversity in handwriting styles and controlled structure make it suitable for evaluating feature extraction algorithms. The focus on word-level data ensures targeted feature extraction and consistent comparisons.

Key features of the dataset:

1. Number of words: 10 unique words.
2. Number of writers: 82 individuals.
3. Total samples: 8,144 images.
4. Data diversity: Variations in handwriting styles, scale, rotation, and illumination.

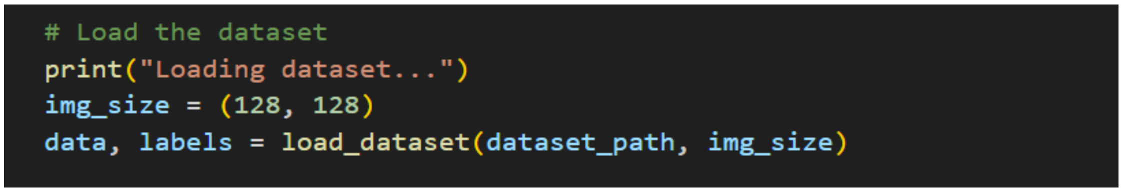


user001_abjadiyah_031.png
user001_abjadiyah_032.png
user001_abjadiyah_033.png
user001_abjadiyah_034.png
user001_abjadiyah_035.png
user001_abjadiyah_036.png
user001_abjadiyah_037.png
user001_azan_003.png
user001_azan_004.png
user001_azan_005.png
user001_azan_006.png
user001_azan_007.png
user001_azan_008.png
user001_azan_009.png
user001_fasayakfeekahum_045.pn
user001_fasayakfeekahum_046.pn
user001_fasayakfeekahum_047.pn
user001_fasayakfeekahum_048.pn
user001_fasayakfeekahum_049.pn
user001_fasayakfeekahum_050.pn
user001_ghaleez_011.png

Figure 8: sample of user001 words

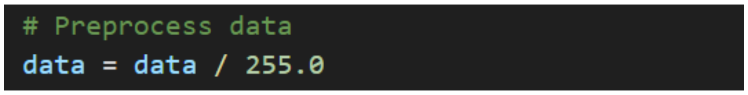
Preprocessing

The dataset comprises grayscale images organized into directories named after unique user IDs, with each image representing an isolated word. This organization facilitates user-specific classification tasks. To standardize input dimensions for the neural network, all images are resized to a fixed size of 128x128 pixels. The resizing ensures uniformity, which is essential for feeding data into a CNN.



```
# Load the dataset
print("Loading dataset...")
img_size = (128, 128)
data, labels = load_dataset(dataset_path, img_size)
```

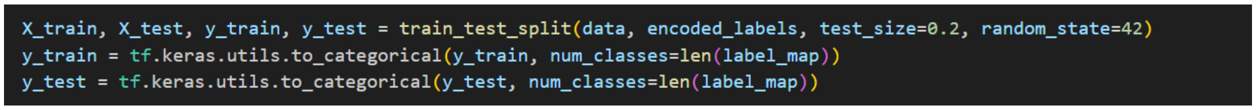
After resizing, the images are normalized to the range [0, 1] by dividing pixel intensity values by 255. This normalization not only accelerates model convergence but also prevents numerical instability during training. The grayscale nature of the images is retained, and a single channel is maintained for simplicity. For consistency with deep learning frameworks, an additional dimension is added to the dataset, making the input shape compatible with CNN layers.



```python
# Preprocess data
data = data / 255.0
```

To handle class labels, each user ID is mapped to a unique integer value. A dictionary is created to establish this mapping, ensuring that each user corresponds to a distinct class index. Labels are then encoded into their respective integers and converted into one-hot encoded vectors, as required for categorical classification tasks.

The dataset is split into training and testing subsets, with 80% of the data allocated for training and 20% reserved for testing. This split allows the evaluation of model performance on unseen data, ensuring a reliable assessment of its generalization capabilities. The split is performed using stratified sampling, preserving the proportion of samples for each class across the subsets.



```python
X_train, X_test, y_train, y_test = train_test_split(data, encoded_labels, test_size=0.2, random_state=42)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=len(label_map))
y_test = tf.keras.utils.to_categorical(y_test, num_classes=len(label_map))
```

Evaluation Metrics

The performance of the models is rigorously assessed using two primary metrics: **loss** and **accuracy**, which together provide a comprehensive evaluation of the model's training progress and generalization ability.

1. **Categorical Cross-Entropy Loss**:

The categorical cross-entropy loss function is a widely used metric for multi-class classification tasks. It measures the dissimilarity between the predicted probability distribution and the true class labels.

For each sample, the model outputs a probability vector across all possible classes. The true label is represented as a one-hot encoded vector, where the correct class is assigned a probability of 1, and all other classes are assigned a probability of 0.

The loss penalizes incorrect predictions by assigning higher penalties when the predicted probability for the correct class is far from 1. Conversely, the loss is minimized when the predicted probability for the true class approaches 1.

Lower cross-entropy loss values during training indicate that the model's predictions are becoming more confident and aligned with the ground truth. Monitoring the loss curve provides insights into whether the model is converging or overfitting.

2. **Accuracy**:

Accuracy is a straightforward and interpretable metric that quantifies the proportion of correctly classified samples. It is calculated as the ratio of the number of correct predictions to the total number of predictions.

This metric is computed separately for the training and validation datasets, providing a clear picture of the model's ability to generalize to unseen data. A high training accuracy indicates that the model is learning the patterns in the training data, while high validation accuracy suggests effective generalization.

During training, accuracy is monitored at the end of each epoch, enabling the identification of trends such as underfitting (low accuracy on both training and validation sets) or overfitting (high training accuracy but low validation accuracy).

Metric Visualization

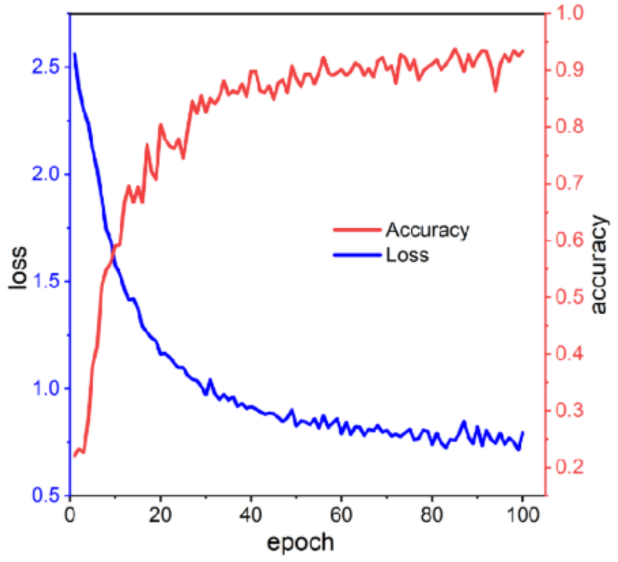
To better understand model performance, loss and accuracy values are plotted over the training epochs for both the training and validation datasets:

1. **Loss Curve**:

The training and validation loss curves are plotted to monitor convergence. Ideally, the training loss should decrease steadily, while the validation loss should stabilize at a low value. If the validation loss increases significantly while the training loss continues to decrease, it is an indication of overfitting.

2. **Accuracy Curve**:

The training and validation accuracy curves are analyzed to assess the model's learning progress. Ideally, both curves should show an upward trend and converge to high values. A large gap between the two curves may indicate overfitting or underfitting, prompting adjustments in the model architecture or training process.



By combining these metrics, the training process is closely monitored to ensure that the model achieves both high accuracy and minimal loss, balancing learning efficiency and generalization performance. The metrics are also compared across different configurations, allowing the identification of the best-performing model.

Hyperparameter Tuning and Training

The process of hyperparameter tuning involves systematically exploring different configurations of the CNN architecture and training parameters to identify the combination that delivers the best performance. This step is crucial as it directly impacts the model's ability to learn meaningful patterns from the data and generalize effectively to unseen samples.

Model Configurations:

Number of Filters: The number of filters in the convolutional layers was varied to control the network's capacity to extract features. Smaller networks began with fewer filters (e.g., [32, 64]), while more complex configurations included additional filters (e.g., [32, 64, 128, 256, 512]).

Dense Layer Units: The number of neurons in the fully connected layer was altered, with smaller networks having fewer units (e.g., 128) and larger ones having more (e.g., 512). This controlled the model's ability to combine extracted features for classification.

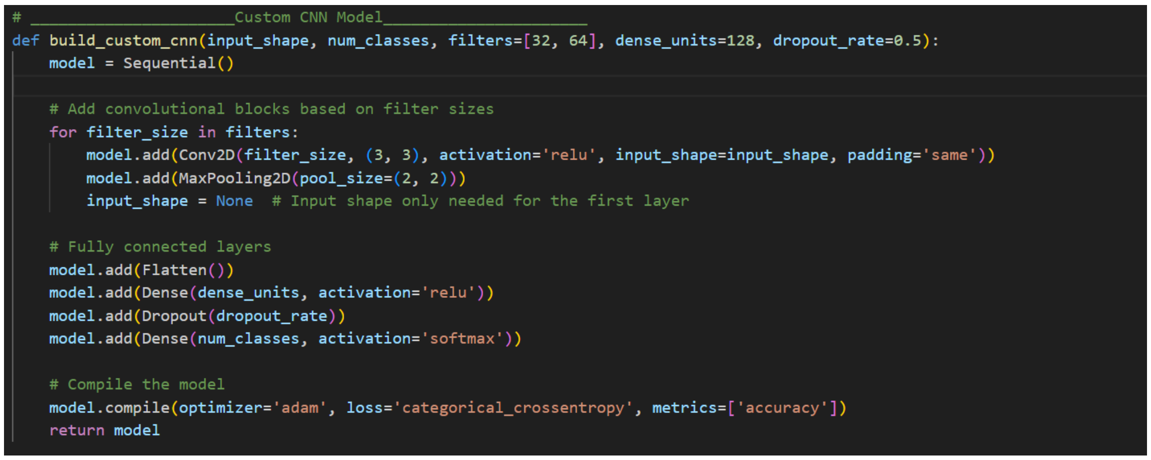
Dropout Rate: Dropout, a regularization technique, was applied to prevent overfitting. The dropout rates tested ranged from 0.2 to 0.5, with lower rates used in larger networks to balance the risk of underfitting and overfitting.

Training Parameters: Hyperparameters such as the number of epochs and batch size were varied to optimize training. Longer training (e.g., 40 epochs) allowed for more thorough learning, while batch sizes (e.g., 32, 64) influenced the stability of gradient updates and convergence speed.

Task 1

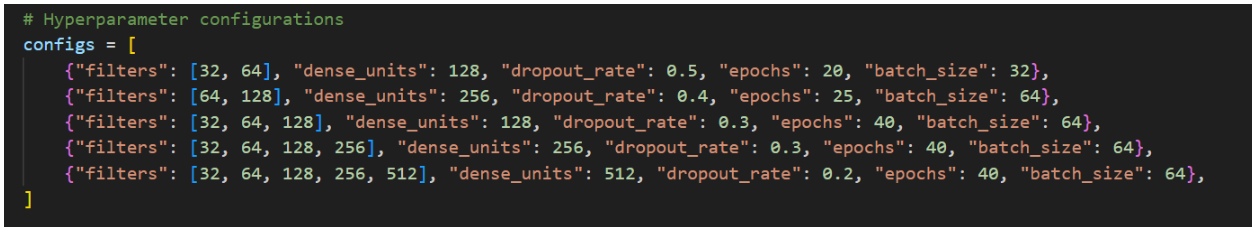
To build and train a custom Convolutional Neural Network (CNN), the architecture must be carefully defined to meet the requirements of the dataset and the classification problem. The architecture begins with an input layer designed to accept images of specific dimensions, followed by a series of convolutional layers that extract features from the images. These layers apply filters (or kernels) to identify patterns, edges, or textures in the data. Each convolutional layer is paired with an **activation function**, such as **ReLU**, which introduces nonlinearity and helps the model learn complex relationships in the data. **Pooling** layers are added to reduce the spatial dimensions of feature maps, thereby decreasing computational costs and minimizing the risk of overfitting. In the example, max pooling with a typical filter size is used to retain the most prominent features while reducing redundancy.

After the convolutional and pooling layers, the output is **flattened** into a one-dimensional vector, which is then passed through fully connected layers. These layers perform the final classification by learning from the high-level features extracted by the preceding layers. Dropout is applied in the fully connected layers to prevent overfitting by randomly disabling a fraction of neurons during training. The **output layer uses a softmax activation function** to generate probabilities for each class, enabling classification. To train the CNN, **hyperparameters such as learning rate, batch size, epochs, and optimizer type are specified**. For example, the Adam optimizer is used in conjunction with categorical cross-entropy as the loss function, ensuring efficient weight updates and robust training. Additionally, the models are evaluated on validation data, and the performance is visualized using loss and accuracy curves to determine the best configuration.



```python
# ______________________Custom CNN Model______________________
def build_custom_cnn(input_shape, num_classes, filters=[32, 64], dense_units=128, dropout_rate=0.5):
    model = Sequential()

    # Add convolutional blocks based on filter sizes
    for filter_size in filters:
        model.add(Conv2D(filter_size, (3, 3), activation='relu', input_shape=input_shape, padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        input_shape = None  # Input shape only needed for the first layer

    # Fully connected layers
    model.add(Flatten())
    model.add(Dense(dense_units, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(num_classes, activation='softmax'))

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

To further optimize the model, hyperparameter tuning is conducted. Various configurations of filter sizes, the number of layers, dropout rates, dense units, and training parameters are explored to find the architecture that achieves the highest validation accuracy.



```
# Hyperparameter configurations
configs = [
    {"filters": [32, 64], "dense_units": 128, "dropout_rate": 0.5, "epochs": 20, "batch_size": 32},
    {"filters": [64, 128], "dense_units": 256, "dropout_rate": 0.4, "epochs": 25, "batch_size": 64},
    {"filters": [32, 64, 128], "dense_units": 128, "dropout_rate": 0.3, "epochs": 40, "batch_size": 64},
    {"filters": [32, 64, 128, 256], "dense_units": 256, "dropout_rate": 0.3, "epochs": 40, "batch_size": 64},
    {"filters": [32, 64, 128, 256, 512], "dense_units": 512, "dropout_rate": 0.2, "epochs": 40, "batch_size": 64},
]
```

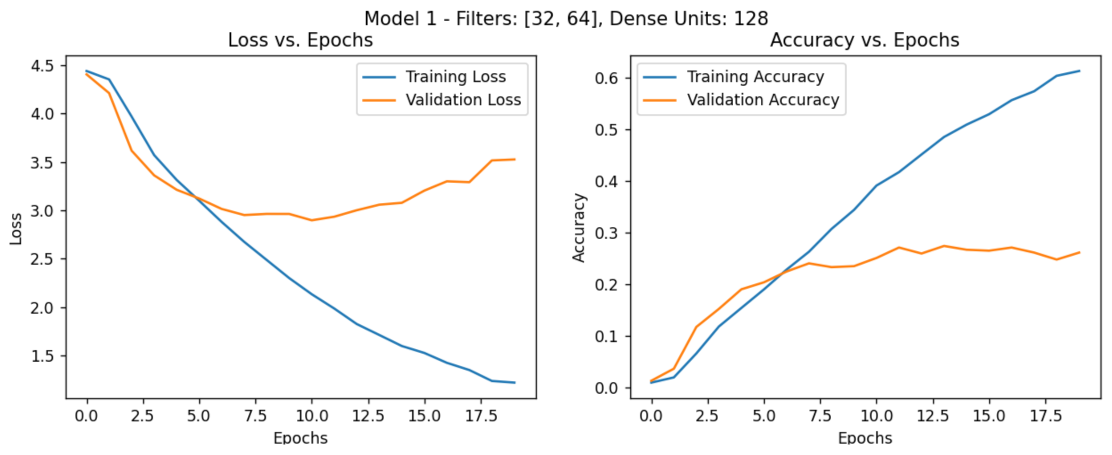
The model with **filters: [32, 64, 128, 256, 512], dense_units: 512, dropout_rate: 0.2, epochs: 40, and batch_size: 64** is expected to perform the **best** due to its deeper architecture and larger number of parameters.

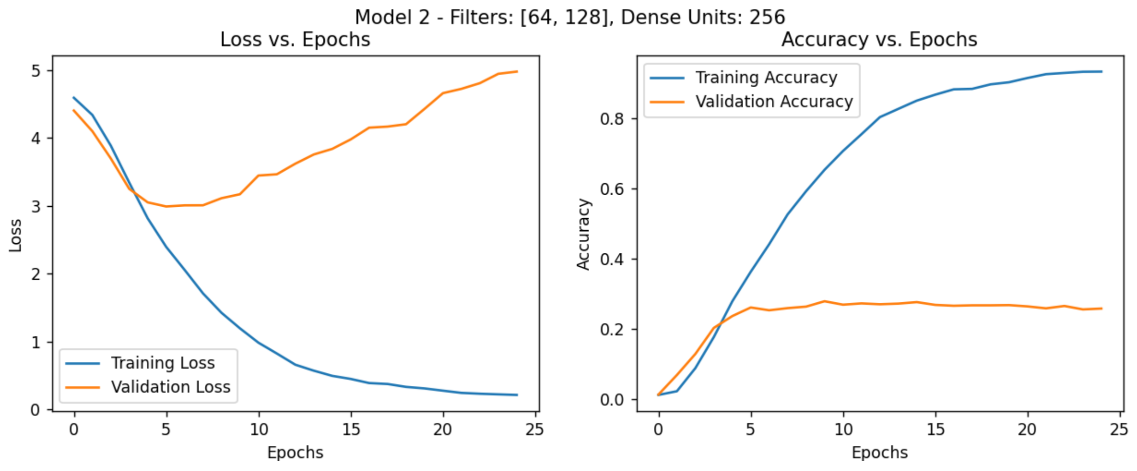
This configuration uses five convolutional layers with an increasing number of filters, which allows the model to extract more complex features from the dataset. The larger dense layer with **512** units further enhances its ability to **capture high-level abstractions in the data**, especially for tasks requiring complex decision boundaries.

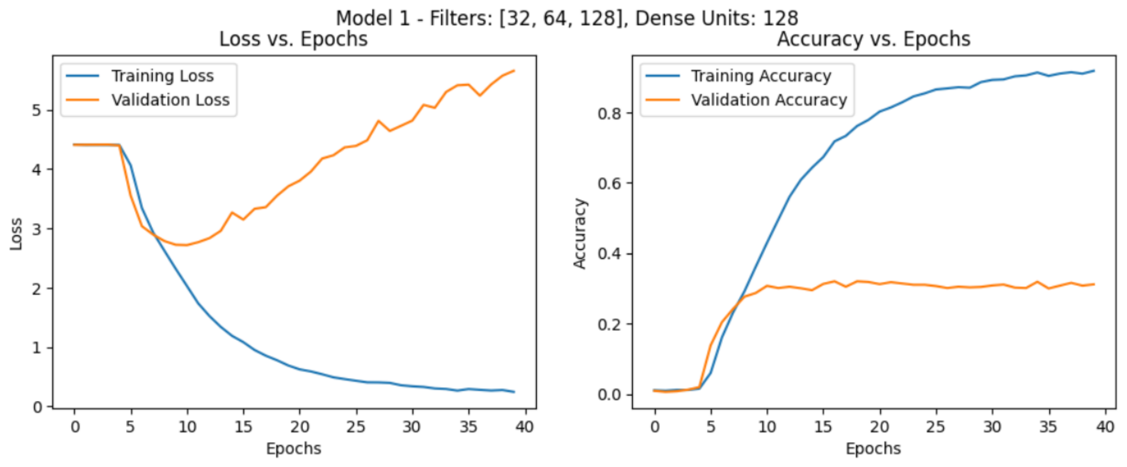
The **lower dropout rate of 0.2** balances regularization and feature retention, preventing excessive information loss during training. The higher epoch count ensures sufficient training time, while **the batch size of 64** provides a balance between stability and computational efficiency. This model's complexity makes it well-suited for larger, more intricate datasets, but its performance will depend on avoiding overfitting and ensuring sufficient computational resources.

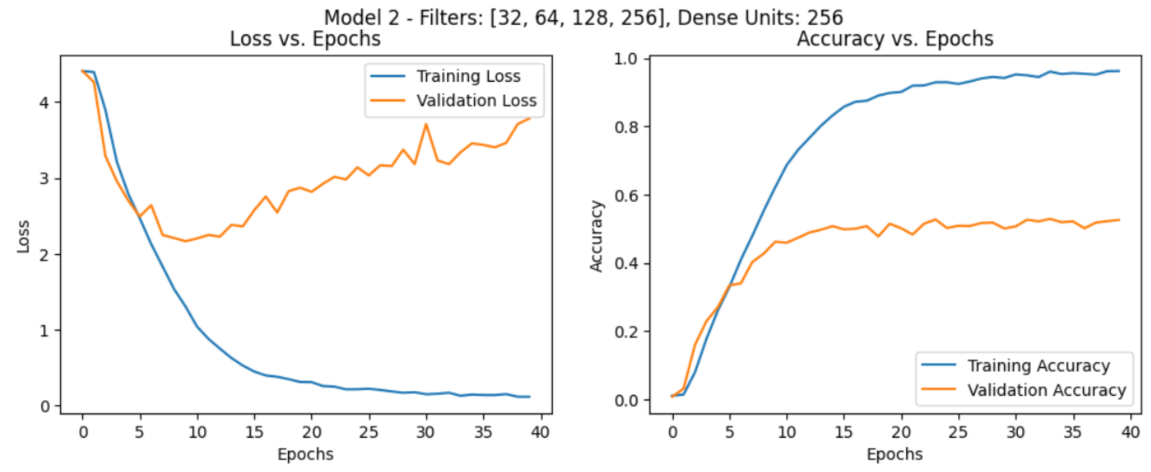
Task 1 Results

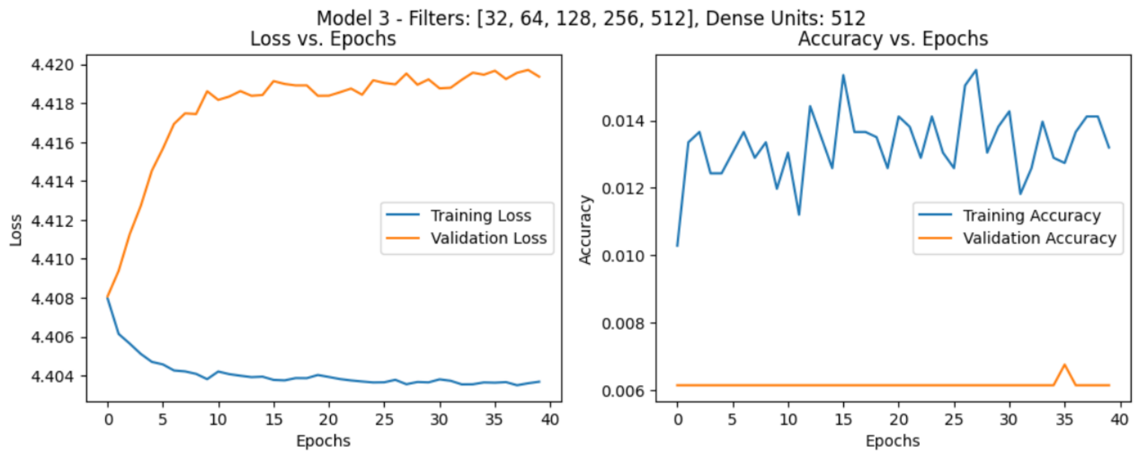
The loss and accuracy curves for several configurations:



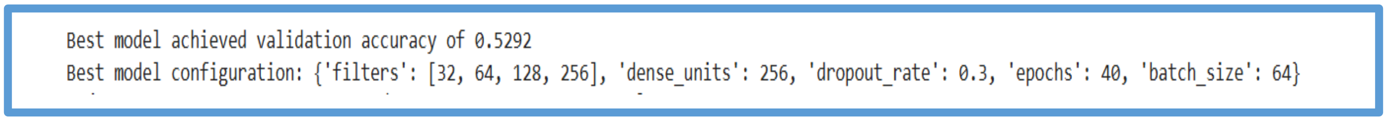








Best model:

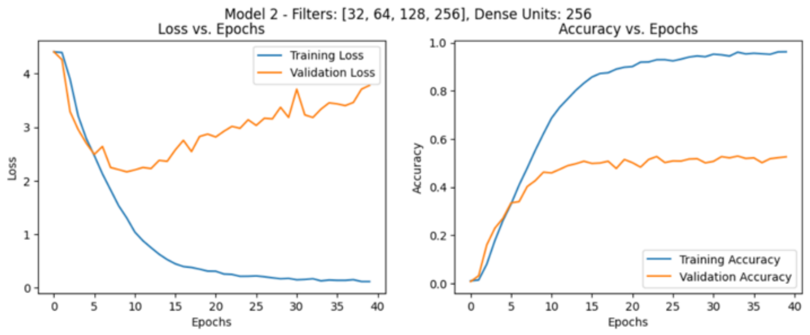


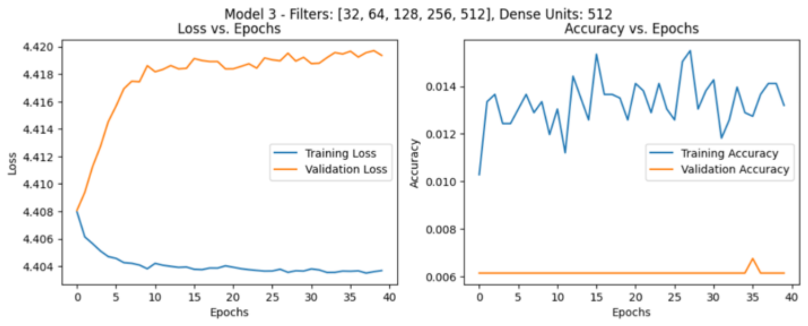
```
Best model achieved validation accuracy of 0.5292
Best model configuration: {'filters': [32, 64, 128, 256], 'dense_units': 256, 'dropout_rate': 0.3, 'epochs': 40, 'batch_size': 64}
```

The best-performing model achieved a **validation accuracy of 0.5292** with the configuration **{'filters': [32, 64, 128, 256], 'dense_units': 256, 'dropout_rate': 0.3, 'epochs': 40, 'batch_size': 64}.** This result indicates that the chosen architecture effectively captured key features in the dataset, balancing complexity and generalization. The use of four convolutional layers with increasing filter sizes ([32, 64, 128, 256]) enabled the model to extract progressively higher-level features from the images, contributing to its performance.

The dense layer with 256 units provided sufficient capacity for learning complex representations, while the dropout rate of 0.3 helped mitigate overfitting by introducing regularization during training. The batch size of 64 balanced computational efficiency and training stability, ensuring effective updates to the model weights. The 40 training epochs allowed the model enough time to learn without overfitting, as evident from the validation accuracy.

The validation accuracy of 0.5292 shows some success but leaves room for improvement. This might be due to limited dataset diversity, class imbalances, or the model's architecture not being strong enough to handle new data. Improving the results could involve adding more data augmentation, using techniques like L2 regularization to prevent overfitting, or adjusting settings like the learning rate or optimizer. These changes could help the model perform better and generalize more effectively.





The expected model would show a balanced learning pattern, where both training and validation loss decrease steadily, and accuracy improves for both sets, indicating effective generalization. However, based on the results, the best model (Model 2) demonstrates overfitting. The training loss decreases consistently, and training accuracy reaches high levels, but the validation loss starts to increase after about 15 epochs, and validation accuracy plateaus at around 50%. This suggests that while the model learns the training data well, it struggles to generalize to unseen data. In contrast, Model 3 fails to learn effectively, as shown by consistently high validation loss and poor validation accuracy throughout training, likely due to excessive model complexity or other limitations. This highlights the gap between the expected and actual performance, emphasizing the need for better regularization, tuning, or data adjustments.

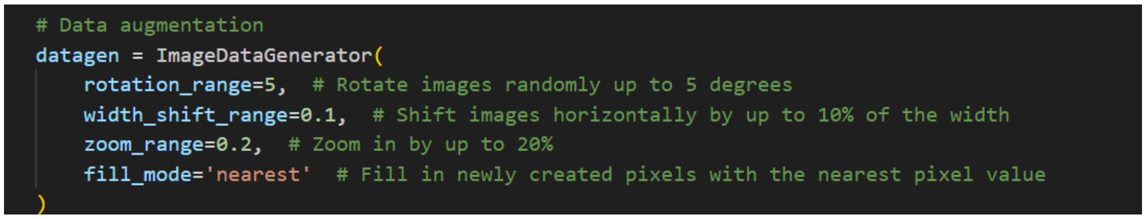
Task2

The task involves retraining the selected best-performing network from Task 1 using data augmentation to enhance the diversity of the training data. Data augmentation applies various transformations, such as rotation, zooming, and shifting, to artificially increase the dataset's size and variability. This helps the model generalize better to unseen data and improves overall performance.

The process begins by loading and preprocessing the dataset, resizing grayscale images to a uniform shape, and normalizing pixel values. Labels are encoded into numerical values, and the data is split into training and testing sets. The model architecture from Task 1, which achieved the best results, is used as the base for retraining. This architecture consists of multiple convolutional layers, fully connected dense layers, and dropout for regularization.

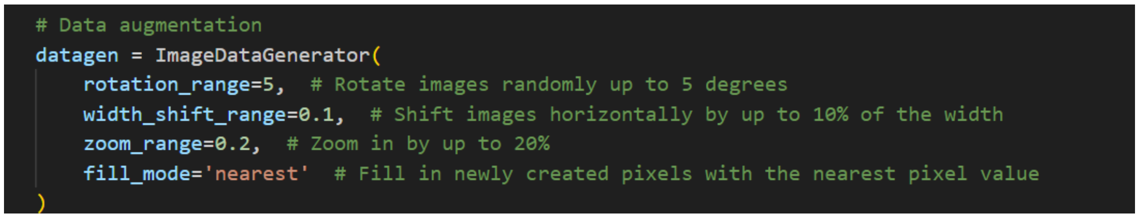
To incorporate data augmentation, transformations such as small rotations, width shifts, and zooming are applied to the training images dynamically during training. The augmented data is fed into the network using the **ImageDataGenerator** class, ensuring that each epoch sees varied versions of the training data. The model is then retrained using these augmented data batches for the specified number of epochs.

Finally, the performance of the model trained with data augmentation is evaluated and compared to its performance from Task 1. Validation accuracy before and after augmentation is analyzed, and the results are visualized through plots of loss and accuracy over epochs to highlight improvements achieved through data augmentation.



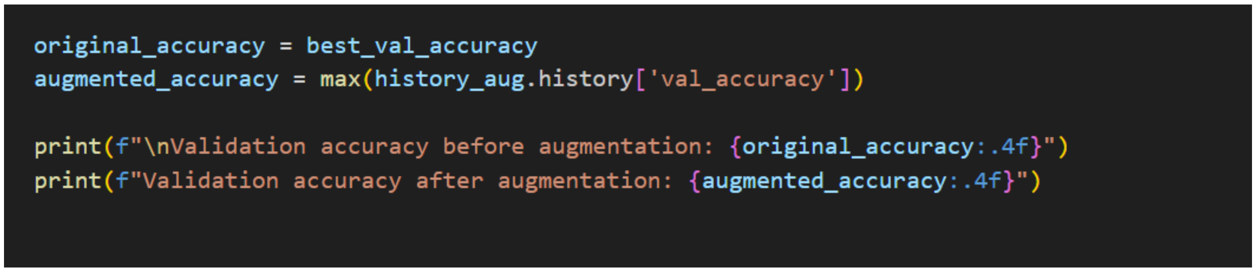
```python
# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=5,  # Rotate images randomly up to 5 degrees
    width_shift_range=0.1,  # Shift images horizontally by up to 10% of the width
    zoom_range=0.2,  # Zoom in by up to 20%
    fill_mode='nearest'  # Fill in newly created pixels with the nearest pixel value
)
```

Data augmentation is a technique used to artificially expand the diversity of a dataset by applying random transformations to the original data, which helps improve the generalization of machine learning models. In the given code, **ImageDataGenerator** is used to perform several types of augmentation on images. The **rotation_range=5** parameter randomly rotates the images by up to 5 degrees. The **width_shift_range=0.1** shifts the images horizontally by up to 10% of their width, and **zoom_range=0.2** allows for random zooming in on the images by up to 20%. Finally, the **fill_mode='nearest'** parameter ensures that any newly created pixels during these transformations are filled with the value of the nearest pixel. These augmentations help create more varied training data, making the model more robust and less likely to overfit to the original data.



```python
# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=5,  # Rotate images randomly up to 5 degrees
    width_shift_range=0.1,  # Shift images horizontally by up to 10% of the width
    zoom_range=0.2,  # Zoom in by up to 20%
    fill_mode='nearest'  # Fill in newly created pixels with the nearest pixel value
)
```

The model is trained using data augmentation, which introduces variations to the training images by applying random transformations. During each training step, augmented versions of the images are generated and fed to the model. The batches consist of 64 images, created from the training data, and these augmented images are used for training. The number of batches processed in each epoch is calculated by dividing the total number of training samples by the batch size. The model will undergo 40 epochs, and after each one, the model's performance will be evaluated using the validation data. This approach helps the model learn from a diverse set of images, improving its ability to generalize to unseen data.



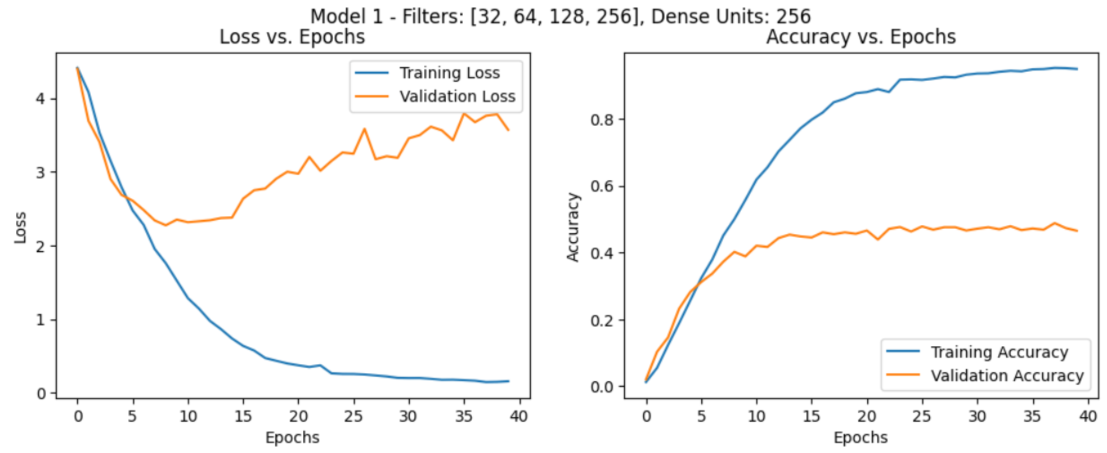
```
original_accuracy = best_val_accuracy
augmented_accuracy = max(history_aug.history['val_accuracy'])

print(f"\nValidation accuracy before augmentation: {original_accuracy:.4f}")
print(f"Validation accuracy after augmentation: {augmented_accuracy:.4f}")
```

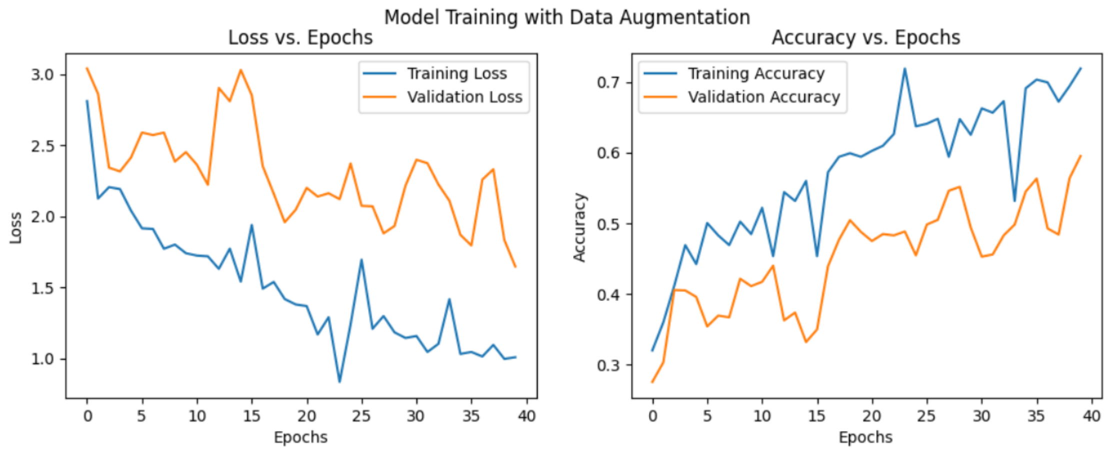
original_accuracy stores the best validation accuracy from the previous task, which was achieved without using data augmentation. augmented_accuracy holds the highest validation accuracy from the training process that involved data augmentation.

Comparing these two values helps assess the effect of data augmentation on the model's performance by showing how it influences accuracy compared to the model trained without augmentation.

Task 2 Results

The loss and accuracy curves for the best model got from task 1 without augmentation, and the augmentation effect:

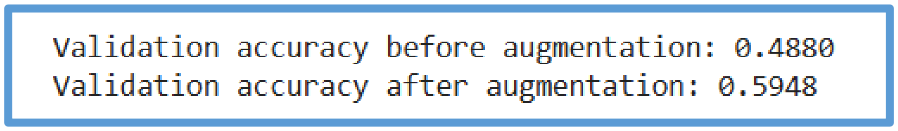




The first two curves show the model's loss and accuracy before data augmentation, and the second set shows the loss and accuracy after data augmentation.

Before augmentation, the training loss decreases, meaning the model is improving on the training data. However, the validation loss starts to level off and then increase a little, which suggests the model is overfitting to the training data. As the training accuracy goes up, the validation accuracy increases more slowly, meaning the model might be memorizing the training data but not performing as well on the validation data.

After adding data augmentation, both training and validation loss decrease more smoothly, and the validation loss doesn't fluctuate as much. This shows that the model is overfitting less. The accuracy curves also improve, with both training and validation accuracy rising more steadily. The validation accuracy gets closer to the training accuracy, showing that data augmentation helped the model generalize better and perform well on new, unseen data.



```
Validation accuracy before augmentation: 0.4880
Validation accuracy after augmentation: 0.5948
```

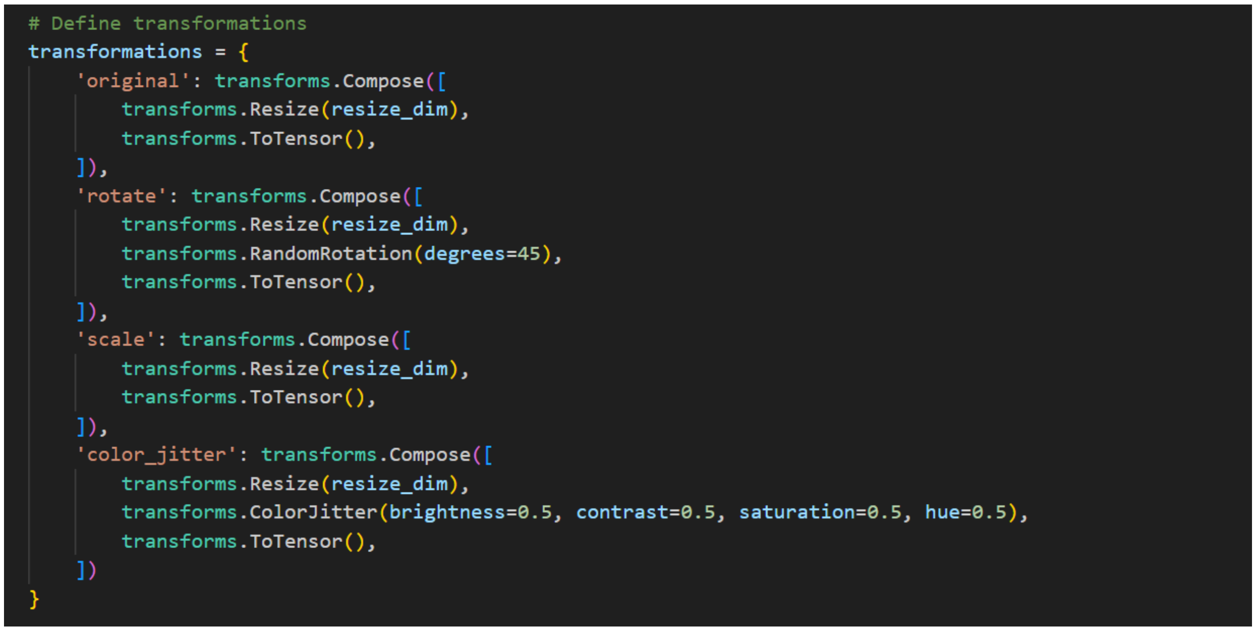
The improvement in validation accuracy **from 0.4880 to 0.5948** demonstrates a significant boost in the model's ability to generalize after incorporating data augmentation. This kind of enhancement is often observed when the model has the opportunity to train on more diverse variations of the data, rather than just the original training samples. Data augmentation, which applies transformations such as rotation, shifting, and zooming, exposes the model to different perspectives of the same data, preventing it from memorizing specific patterns and overfitting to the original dataset.

Before augmentation, the model might have been overfitting, as it was only exposed to the same set of images during training, which limits its ability to generalize to new, unseen data. With augmentation, the model experiences a broader range of input variations, which can improve its robustness and ability to handle variations in real-world data.

The increase in validation accuracy suggests that the augmented dataset helped the model better capture the underlying patterns in the data, making it more adaptable to different conditions. This is particularly important in tasks where the data can vary in subtle ways, such as object recognition in images or other visual tasks. The 0.5948 validation accuracy after augmentation indicates that the model is now performing better and is more likely to maintain this performance when exposed to new, unseen data.

Task 3

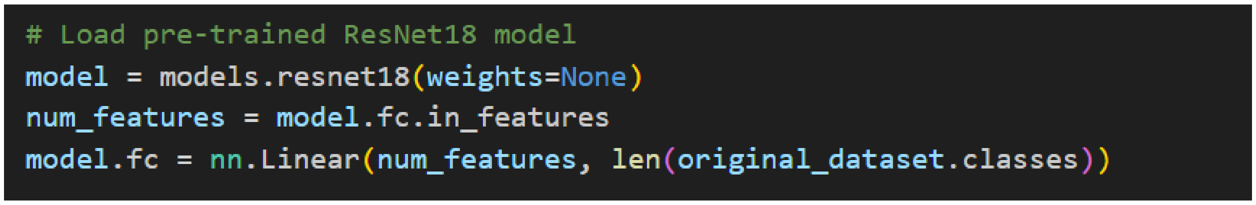
Creating Augmented Datasets



```
# Define transformations
transformations = {
    'original': transforms.Compose([
        transforms.Resize(resize_dim),
        transforms.ToTensor(),
    ]),
    'rotate': transforms.Compose([
        transforms.Resize(resize_dim),
        transforms.RandomRotation(degrees=45),
        transforms.ToTensor(),
    ]),
    'scale': transforms.Compose([
        transforms.Resize(resize_dim),
        transforms.ToTensor(),
    ]),
    'color_jitter': transforms.Compose([
        transforms.Resize(resize_dim),
        transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
        transforms.ToTensor(),
    ])
}
```

For each of the additional transformations (rotation, scaling, and color jittering), a new dataset is created by applying the corresponding transformation. These augmented datasets simulate variations in the input data, making the model more robust to changes.

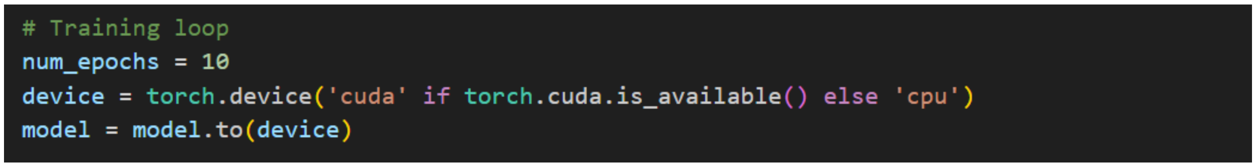
Loading a Pre-Trained Model



```
# Load pre-trained ResNet18 model
model = models.resnet18(weights=None)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, len(original_dataset.classes))
```

A ResNet18 model, a popular deep learning architecture for image classification, is initialized. Its final layer is replaced with a new one matching the number of classes in the dataset, allowing the model to predict the specific categories in this task.

Training the Model

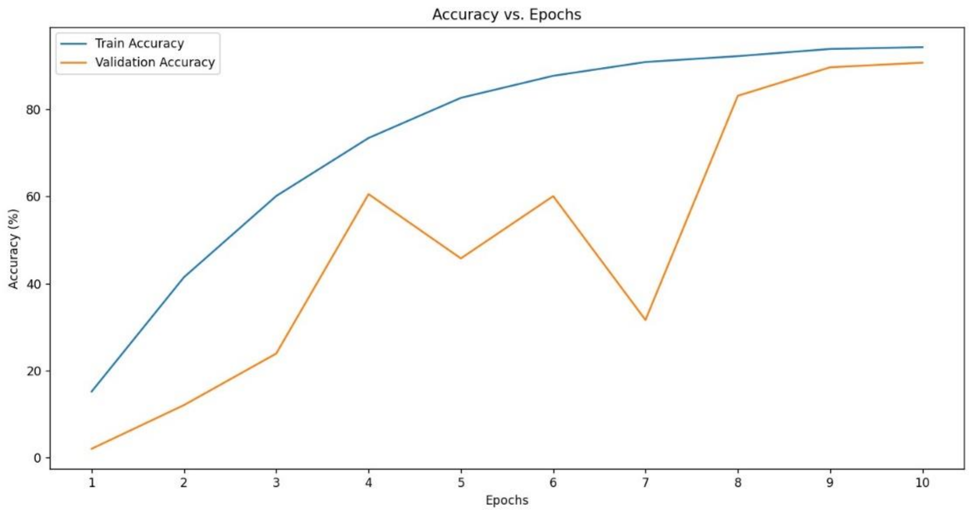


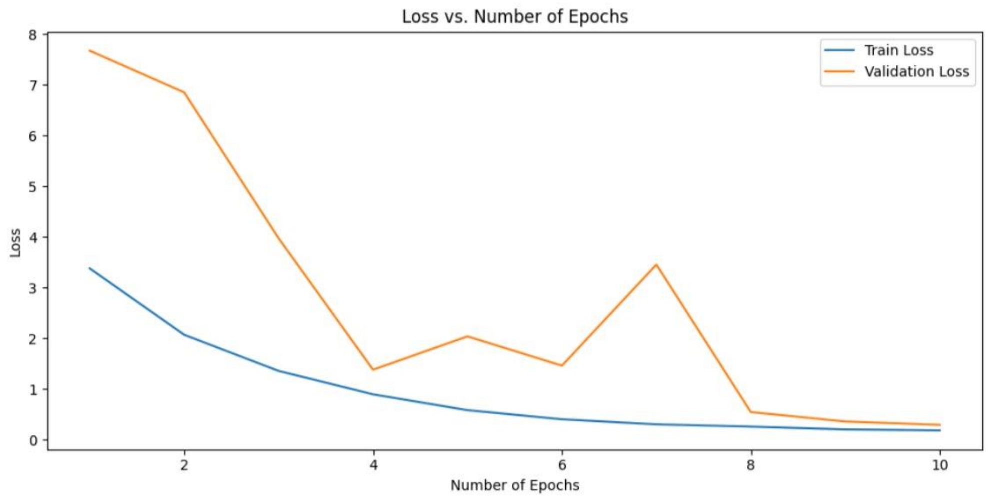
```
# Training loop
num_epochs = 10
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

The model is trained over 10 cycles (epochs). For each epoch, the training data is used to update the model's weights. Images and their labels are passed through the model, the loss is calculated, and the optimizer adjusts the weights. Metrics like training loss and accuracy are recorded to track progress.

Task 3 Results

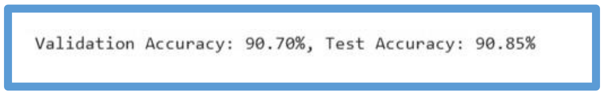
The loss and accuracy curves for several configurations:





The visualizations depict the accuracy and loss trends for the training and validation datasets over ten epochs, providing insight into the model's learning process and generalization capabilities. In the first graph, both training and validation accuracy improve steadily, with the training accuracy surpassing 90% by the final epoch. The validation accuracy follows a similar trajectory but displays occasional fluctuations, which might indicate slight overfitting or variations due to data complexity. The consistent upward trend, however, highlights effective learning, with the model generalizing well to unseen data.

The second graph shows the loss decreasing for both training and validation datasets, signifying improved predictions over time. While the training loss reduces smoothly, the validation loss exhibits occasional spikes. These fluctuations could be due to the challenging nature of the validation dataset or noise in the data. Overall, the steady decline in both metrics confirms that the model is successfully minimizing error during training and retaining its ability to generalize. These results demonstrate a well-trained model, though future adjustments, such as fine-tuning hyperparameters, might help smooth out validation inconsistencies.



```
Validation Accuracy: 90.70%, Test Accuracy: 90.85%
```

The results show that the model achieved a test accuracy of 90.85% and a validation accuracy of 90.7%. This indicates that the model has been trained effectively and performs consistently on both the validation and test datasets. The close alignment between these two metrics suggests that the model has generalized well to unseen data, which is a strong indicator that overfitting is minimal. The use of data augmentation techniques, such as rotation and color jittering, likely contributed to this robustness by exposing the model to a variety of scenarios and preventing it from simply memorizing the training data.

However, while the results are promising, they also leave room for improvement. Achieving over 90% accuracy demonstrates that the model can classify the data with a high degree of reliability, but it is not perfect. There may still be edge cases or specific categories where the model struggles, which could lower performance in real-world applications. The consistent gap between training and validation accuracy should be monitored in future experiments to ensure that it doesn’t widen with changes to the data or training process.

Additionally, the results suggest the potential for further fine-tuning. Techniques such as optimizing hyperparameters, experimenting with different architectures, or using transfer learning from a pre-trained model with weights could yield even higher accuracy. Finally, a more detailed analysis of errors, such as examining misclassified samples, could provide insights into specific weaknesses and guide improvements in both data preprocessing and model design.

Comparison between Task 2 and Task 3

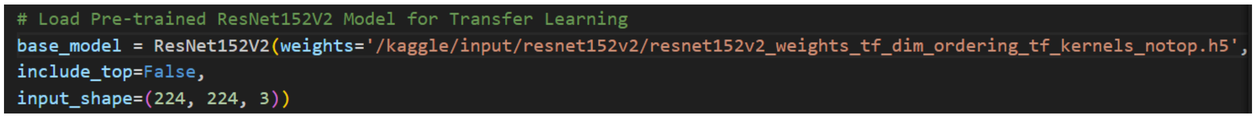
Both Task 2 and Task 3 explore the role of data augmentation in improving a model's performance, but they differ in focus, methodology, and results. Task 2 primarily aimed to analyze how augmentation impacts a model's ability to generalize. The results showed a clear improvement in validation accuracy from 48.80% to 59.48%, demonstrating that exposing the model to diverse variations like rotations and zooming significantly enhanced its robustness. Before augmentation, the model likely suffered from overfitting because it only trained on the same set of unaltered images. Augmentation provided a broader range of input variations, allowing the model to learn more generalized patterns instead of memorizing specific ones. This was especially beneficial given the potential limitations in the original dataset's diversity.

Task 3, on the other hand, incorporated data augmentation into the training of a more advanced model, ResNet18. Here, techniques like rotation and color jittering played a similar role in improving the model's ability to generalize, but the use of ResNet18 added significant value. The results were much stronger, with the model achieving a validation accuracy of 90.7% and a test accuracy of 90.85%. These high and consistent accuracy levels indicate that combining data augmentation with a well-designed CNN architecture can yield impressive performance. While augmentation helped expose the model to various scenarios, the deep architecture of ResNet18 likely contributed to its ability to learn complex patterns, resulting in its superior performance compared to Task 2.

.

Task 4

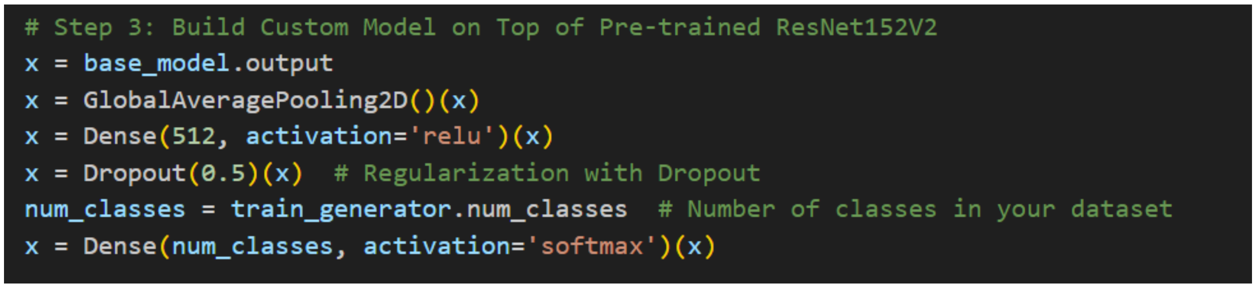
Step 1: Loading a Pretrained Model for Transfer Learning



```
# Load Pre-trained ResNet152V2 Model for Transfer Learning
base_model = ResNet152V2(weights='/kaggle/input/resnet152v2/resnet152v2_weights_tf_dim_ordering_tf_kernels_notop.h5',
include_top=False,
input_shape=(224, 224, 3))
```

A ResNet152V2 model is used as the base for transfer learning. This model is already trained on a large dataset and has learned general features that can be adapted to the new dataset. By using the pretrained weights, the model can learn faster and achieve better results even with a smaller dataset. To prevent disrupting the learned features, the first 100 layers of the ResNet model are frozen, keeping them unchanged during training. The remaining layers are made trainable so they can fine-tune to the new dataset's specific characteristics.

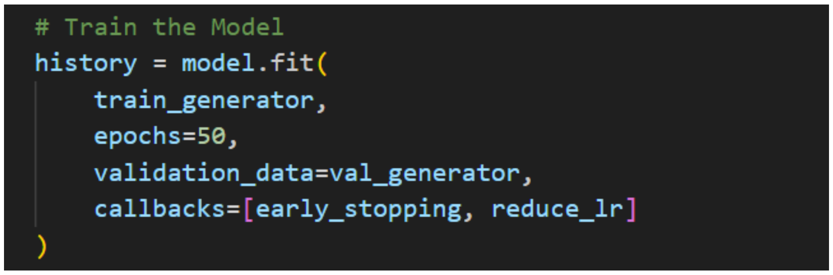
Step 2: Building a Custom Model for Classification



```
# Step 3: Build Custom Model on Top of Pre-trained ResNet152V2
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)  # Regularization with Dropout
num_classes = train_generator.num_classes  # Number of classes in your dataset
x = Dense(num_classes, activation='softmax')(x)
```

On top of the ResNet152V2 base model, additional layers are added to specialize the model for the current dataset. A global average pooling layer reduces the feature maps to a compact representation, followed by a fully connected layer with 512 units and ReLU activation to learn complex patterns. A dropout layer is included to prevent overfitting by randomly deactivating neurons during training. Finally, a softmax layer is added to classify the images into their respective categories.

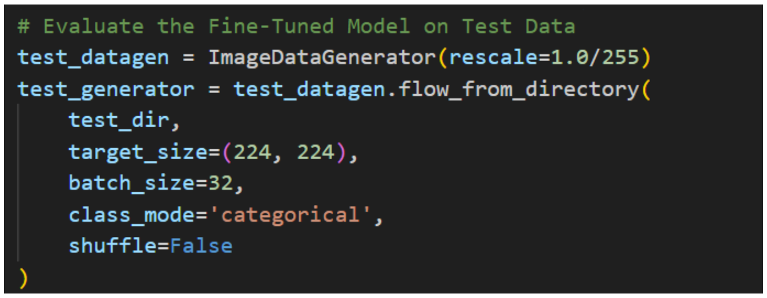
Step 3: Training the Model



```
# Train the Model
history = model.fit(
    train_generator,
    epochs=50,
    validation_data=val_generator,
    callbacks=[early_stopping, reduce_lr]
)
```

The model is trained using the augmented training dataset, while its performance is monitored on the validation dataset. The validation data helps evaluate how well the model generalizes to unseen data during training. The callbacks ensure that the model converges efficiently without overfitting.

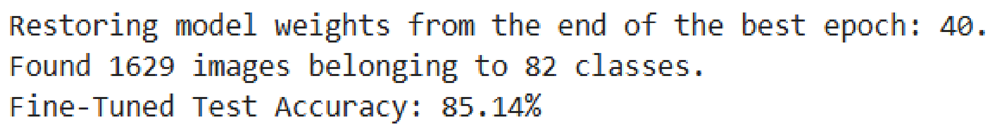
Step 4: Evaluate the Fine-Tuned Model on Test Data



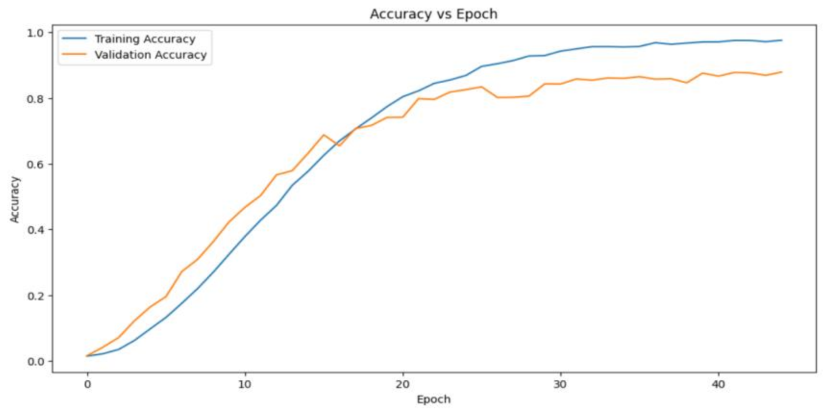
```
# Evaluate the Fine-Tuned Model on Test Data
test_datagen = ImageDataGenerator(rescale=1.0/255)
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)
```

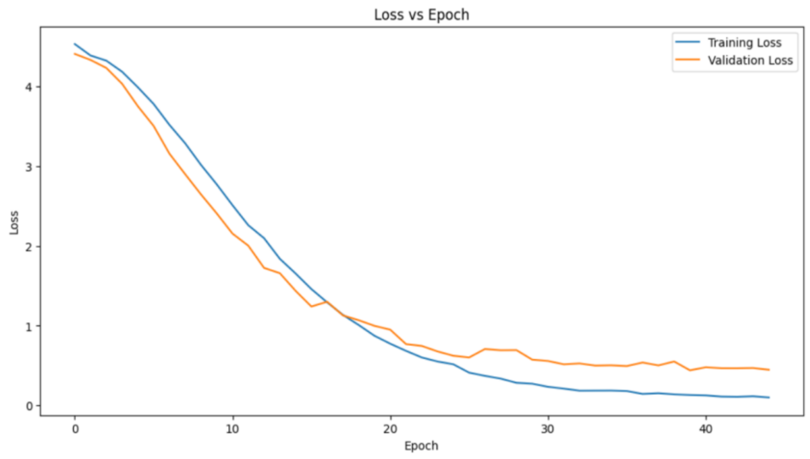
To evaluate the fine-tuned model on unseen data, the test dataset is prepared by rescaling pixel values to a range between 0 and 1, ensuring compatibility with the model's training scale. A test data generator loads the images from the test directory and resizes them to 224x224 pixels, matching the input shape required by the model. The data is organized into batches of 32 images, and the categorical class mode is used to handle multi-class labels. The shuffle parameter is disabled to maintain a consistent order between the predictions and their corresponding labels, facilitating an accurate performance assessment. This process ensures that the evaluation reflects how well the model can generalize to entirely new data.

Task 4 Results



```
Restoring model weights from the end of the best epoch: 40.
Found 1629 images belonging to 82 classes.
Fine-Tuned Test Accuracy: 85.14%
```





The results show that using **a pre-trained ResNet152V2 model** and fine-tuning it on the dataset was very effective. The dataset had 1,629 images across 82 classes, which makes this a challenging task because there are many categories and a relatively small number of images. Despite this, the model achieved a validation accuracy of **85.14%**, showing it performed well at recognizing the classes.

The accuracy and loss graphs provide a clear picture of the training process. **The accuracy graph shows both training and validation accuracy improving steadily over tim**e, with validation accuracy staying close to training accuracy. **This means the model is learning well without overfitting**. **In the loss graph, the training and validation loss decrease steadily, showing that the model is improving consistently as it learns.**

These results highlight the power of transfer learning. By starting with a model that has already learned features from a large dataset, we saved time and achieved good performance even with fewer images. This approach worked well for adapting the model to the specific task, making it a great choice for datasets with limited data.

Comparison between Task 3 and Task 4

Task 3 and Task 4 both involve training convolutional neural networks (CNNs) for image classification, but they use different approaches and architectures, which led to distinct outcomes. Task 3 used ResNet18, a well-known CNN, trained from scratch with data augmentation. The model achieved impressive results, with a test accuracy of 90.85% and a validation accuracy of 90.7%. The close alignment of these metrics suggests strong generalization and minimal overfitting. Data augmentation, such as rotation and color jittering, was key to this performance, helping the model handle variations in the dataset. However, Task 3 also highlighted areas for improvement, such as optimizing hyperparameters and exploring fine-tuning to handle edge cases and boost accuracy further.

In contrast, Task 4 leveraged transfer learning by fine-tuning a pre-trained ResNet152V2 model, a deeper and more complex architecture, on a dataset with 1,629 images across 82 classes. Despite the dataset's limited size and high number of categories, the model achieved a validation accuracy of 85.14%. The use of transfer learning allowed the model to build on features learned from a large, generic dataset, saving training time and improving performance with limited data. The training and validation accuracy improved steadily throughout, and the loss consistently decreased, showing effective learning without overfitting. These results highlight the efficiency of transfer learning, particularly for tasks with small datasets, as it enables the model to adapt quickly to specific problems while leveraging previously learned knowledge.

Conclusion

Across the four tasks, each approach highlighted different aspects of training deep learning models and demonstrated their strengths and limitations. Task 1 achieved a validation accuracy of 52.92% using a custom architecture with four convolutional layers and appropriate regularization. While the result reflected the model's ability to extract meaningful features, it also underscored the challenges of limited dataset diversity and potential architectural constraints. Enhancements like more robust data augmentation or fine-tuning hyperparameters could further improve performance.

Task 2 emphasized the impact of data augmentation, showcasing a significant improvement in validation accuracy from 48.80% to 59.48%. This demonstrated how transformations such as rotation and scaling helped the model generalize better by exposing it to a wider range of variations. While the improvement was evident, the task highlighted the limitations of relying solely on data augmentation when working with simpler architectures.

Task 3 introduced a pre-trained ResNet18 model, which was trained from scratch with data augmentation, achieving impressive test and validation accuracies of 90.85% and 90.7%, respectively. This approach balanced the benefits of a well-designed architecture and robust data augmentation, showcasing strong generalization and minimal overfitting. However, fine-tuning and further analysis of misclassifications could potentially push the performance even higher.

Finally, Task 4 utilized transfer learning with a ResNet152V2 model, fine-tuned on a challenging dataset of 1,629 images across 82 classes. Despite the dataset's complexity and size, the model achieved a validation accuracy of 85.14%. This highlighted the power of transfer learning in leveraging pre-trained knowledge to adapt effectively to specific tasks. The approach proved particularly efficient for datasets with limited images, offering an excellent balance between performance and resource efficiency.

Overall, the four tasks underline the importance of tailoring model design, training strategies, and data preprocessing to the specific problem. Custom architectures can be useful for controlled experiments, data augmentation plays a crucial role in boosting generalization, and leveraging pre-trained models through transfer learning provides a robust solution for challenging datasets.