Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN3150: Pattern Recognition

Assignment 3:

Simple convolutional neural network to perform classification.

Group DeepTrace

Anjula M.K	210368V
Meemanage N.A	210385U
Shamal G.B.A	210599E
Thennakoon T.M.K.R	210642G

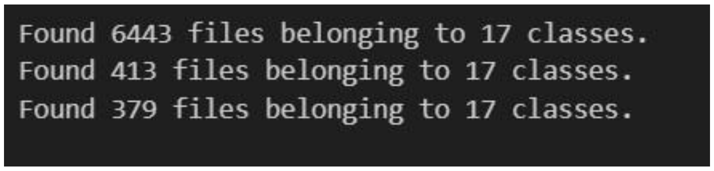
Contents

1.CNN for image classification

In this project we are working with an image classification dataset from the UCI Machine Learning Repository focused on the classification of various insect pests commonly found in jute crops. This dataset consists of 17 distinct classes each representing a different pest such as Beet Armyworm, Black Hairy, Cutworm and Jute Aphid among others. The classes are labeled from 0 to 16 each corresponding to a unique pest type.

The dataset includes 7,235 instances and features 17 categorical attributes making it suitable for classification tasks in the area of biology. The data has been pre-divided into three partitions: training, validation and testing allowing for systematic model training, tuning and evaluation. This dataset will be ideal for constructing and testing a convolutional neural network (CNN) model aiming to accurately classify the insect types based on their images.

1.3



```
Found 6443 files belonging to 17 classes.
Found 413 files belonging to 17 classes.
Found 379 files belonging to 17 classes.
```

The dataset for this project has been effectively organized into three main partitions: training, validation and test sets. This partitioning is essential for developing, tuning and evaluating a robust image classification model. The breakdown is as follows:

- Training Set: Contains 6,443 images representing all 17 insect classes. This set will be used to train the convolutional neural network (CNN) allowing the model to learn distinguishing features of each class.
- Validation Set: Contains 413 images also covering all 17 classes. This set allows for model validation during training enabling tuning of hyperparameters and monitoring of performance to avoid overfitting.
- Test Set: Contains 379 images with images from each of the 17 classes. This independent set is reserved for the final evaluation of the model's accuracy and generalization ability once training is complete.

Having each class represented in all three sets ensures that the model learns patterns across all categories and can generalize well.

1.4

We designed a convolutional neural network (CNN) model to classify images into 17 distinct classes, using TensorFlow and Keras.

a.Input Layer:

- A Convolutional layer with 32 filters (3x3 kernel) and ReLU activation processes input images of size (224, 224, 3) to extract low-level features followed by Batch Normalization for consistent input distribution.

b.Max Pooling Layer:

- A MaxPooling layer with a 2x2 window reduces the spatial dimensions of the feature maps, decreasing computational complexity and capturing spatial hierarchies.

c.Second Convolutional Layer:

- Another Convolutional layer with 64 filters (3x3 kernel) and ReLU activation extracts more complex features followed by Batch Normalization.

d.Second Max Pooling Layer:

- A second MaxPooling layer (2x2 window) further reduces feature map dimensions allowing the model to capture higher-level features.

e.Dropout Layer:

- A Dropout layer with a 0.5 rate mitigates overfitting by randomly dropping 50% of neurons during training.

f.Flattening Layer:

- A Flatten layer converts 2D feature maps into a 1D vector for the fully connected layers.

g.Fully Connected (Dense) Layer:

- A Dense layer with 128 units and ReLU activation learns high-level feature combinations with L2 regularization (0.001) to reduce overfitting. Batch Normalization is also applied.

h.Second Dropout Layer:

- Another Dropout layer (0.5 rate) enhances generalization and reduces overfitting.

i.Output Layer:

- The final Dense layer with 17 units and SoftMax activation converts outputs into probabilities for 17 classes.

1.5

a.Convolutional Layers:

1. ***First Convolutional Layer:***

   Activation Function: ReLU

   Kernel Size: (3, 3)

   Number of Filters: 32

   Input Shape: (224, 224, 3)

1. ***Second Convolutional Layer:***

   Activation Function: ReLU

   Kernel Size: (3, 3)

   Number of Filters: 64

b.Max Pooling Layers:

1. ***First Max Pooling Layer:***
   - Pool Size: (2, 2)
2. ***Second Max Pooling Layer:***
   - Pool Size: (2, 2)

c.Dropout Layers:

1. ***First Dropout Layer:***
   - Dropout Rate: 0.5
2. ***Second Dropout Layer: o***

   Dropout Rate: 0.5

d.Fully Connected (Dense) Layer:

1. Dense Layer:

Number of Units: 128

Activation Function: ReLU

Regularization: L2 with a factor of 0.001

e.Output Layer:

1. Output Dense Layer:

Number of Units: 17

Activation Function: Softmax

1.6

a. ReLU (Rectified Linear Unit) Activation Function:

In the convolutional layers, we use the ReLU (Rectified Linear Unit) activation function because it introduces non-linearity while maintaining computational efficiency. This allows the model to learn complex patterns effectively. The ReLU function outputs zero for negative values and returns the input value for positive values which helps mitigate the vanishing gradient problem during training. Its simplicity and efficiency make it a widely adopted choice in convolutional neural networks (CNNs).

Also the non-saturating nature of ReLU helps in faster convergence compared to traditional activation functions like sigmoid or tanh especially in deeper networks.

b. Softmax Activation Function:

In the output layer we use the Softmax activation function because it is ideal for multi-class classification problems particularly as this model has 17 classes. Softmax transforms the raw output logits into probabilities for each class ensuring that the sum of these probabilities equals one. This property is essential for interpreting the model's outputs as class probabilities enabling effective decision-making based on the highest probability class.

Also Softmax provides a clear interpretation of the model's confidence in each class allowing for better decision-making based on the highest probability output.

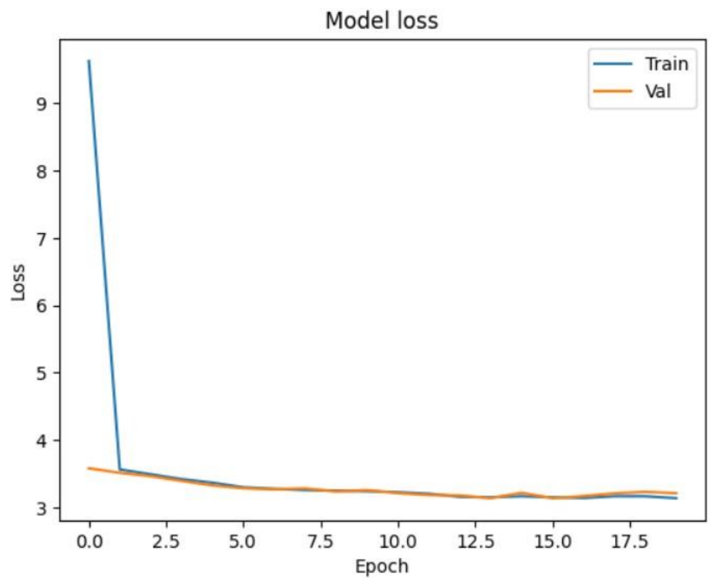
1.7

A convolutional neural network (CNN) was designed with 3 convolutional layers max pooling, dropout and fully connected layers to perform feature extraction and classification. The model was compiled using the Adam optimizer sparse categorical cross-entropy as the loss function, and accuracy as the evaluation metric. It was trained for 20 epochs using the training dataset with validation conducted on the validation dataset. Throughout the training process both training and validation loss and accuracy were monitored to evaluate the model's performance.

Results Observed:

- ***Training Accuracy***: Gradually improved over epochs, reaching around 12%.
- ***Validation Accuracy***: Remained low (~18-19%), showing minimal improvement.
- ***Validation Loss***: Decreased slightly but fluctuated around 3.2, indicating that the model struggled to generalize well to the validation dataset.

The training and validation loss over the epochs were plotted using Matplotlib to visualize the model's performance and assess its learning behavior during training and validation.



1.8

In our model with convolutional and dense layers Adam's adaptive learning rates efficiently adjusted parameter updates for different layers, whereas SGD would require careful manual tuning, potentially delaying the training process.

Adam's built-in momentum accelerated convergence during your 20-epoch training period helping to stabilize parameter updates whereas SGD without momentum might have struggled to make significant progress in the same timeframe.

The training process showed fluctuating validation loss which Adam managed well with its robust gradient adjustments while SGD might have been more sensitive to these fluctuations leading to unstable training.

Also Adam converged more quickly reducing the training loss consistently within 20 epochs while SGD would likely have needed more epochs to achieve similar loss values due to its slower convergence.

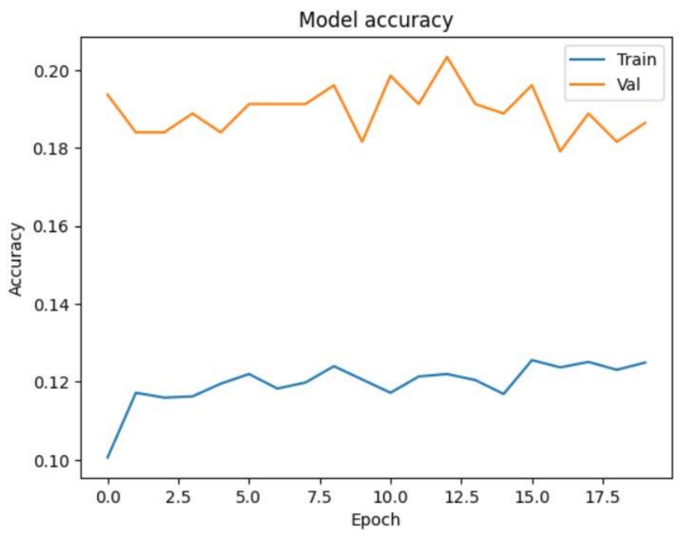
1.9

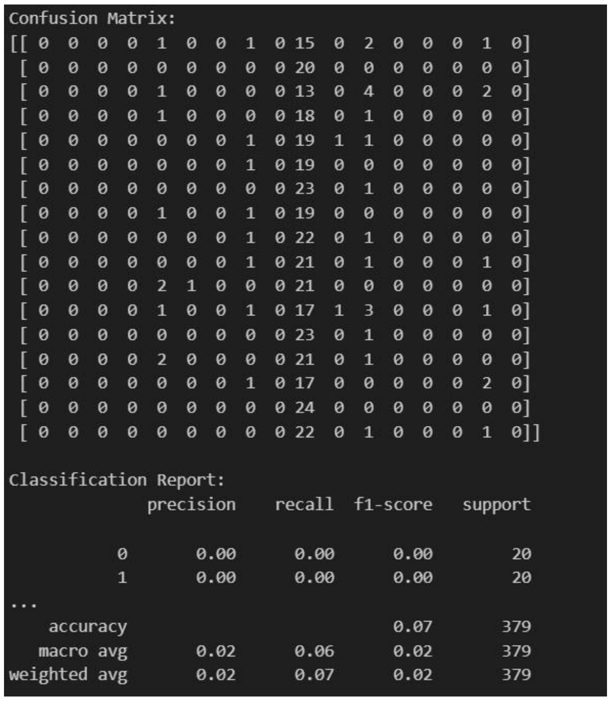
In our dataset the class labels are provided as integers (0, 1, 2, ... 16 for 17 classes). Sparse categorical cross-entropy directly handles these integer labels without requiring them to be onehot encoded simplifying the input preparation.

Model is designed for a multi-class classification task (17 classes) where each input belongs to exactly one class. Sparse categorical cross-entropy is specifically designed for such scenarios.

Sparse categorical cross-entropy is computationally efficient because it avoids the additional step of converting integer labels to one-hot encoded vectors reducing memory and processing overhead.

1.10





```
Confusion Matrix:
[[ 0  0  0  0  1  0  0  1  0 15  0  2  0  0  0  1  0]
 [ 0  0  0  0  0  0  0  0  0 20  0  0  0  0  0  0  0]
 [ 0  0  0  0  1  0  0  0  0 13  0  4  0  0  0  2  0]
 [ 0  0  0  0  1  0  0  0  0 18  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0 19  1  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0 19  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0 23  0  1  0  0  0  0  0]
 [ 0  0  0  0  1  0  0  1  0 19  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0 22  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0 21  0  1  0  0  0  1  0]
 [ 0  0  0  0  2  1  0  0  0 21  0  0  0  0  0  0  0]
 [ 0  0  0  0  1  0  0  1  0 17  1  3  0  0  0  1  0]
 [ 0  0  0  0  0  0  0  0  0 23  0  1  0  0  0  0  0]
 [ 0  0  0  0  2  0  0  0  0 21  0  1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  1  0 17  0  0  0  0  0  2  0]
 [ 0  0  0  0  0  0  0  0  0 24  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0 22  0  1  0  0  0  1  0]]

Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        20
           1       0.00      0.00      0.00        20
...
    accuracy                           0.07       379
   macro avg       0.02      0.06      0.02       379
weighted avg       0.02      0.07      0.02       379
```

Training and Validation Accuracy:

- The training accuracy remained low, around 12%, across the 20 epochs.
- The validation accuracy fluctuated slightly but stayed around 18–20%, indicating that the model struggled to generalize to the validation dataset.

Confusion Matrix:

- The confusion matrix shows that the model failed to correctly classify most classes, with significant misclassifications and very few correct predictions.
- This indicates poor performance across all 17 classes, suggesting the model could not learn meaningful patterns from the data.

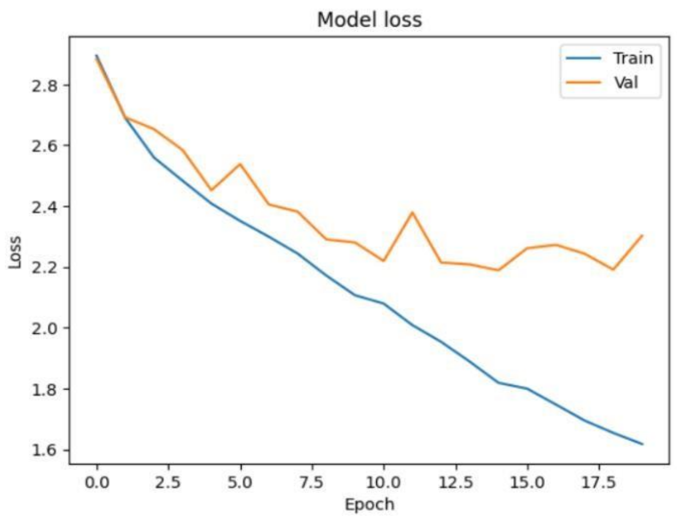
Classification Summary:

- ***Precision, Recall, and F1-Score:***
  - All metrics are very low (close to 0) for most classes, indicating that the model performs poorly in identifying the correct class and managing false positives and false negatives.
  - The macro average precision, recall and F1-score are extremely low (~0.02), showing no substantial performance improvement across the classes.

- ***Overall* Accuracy**:
    - The test accuracy is only **7%**, far below the expected performance, reflecting the model's inability to classify correctly on the testing dataset.

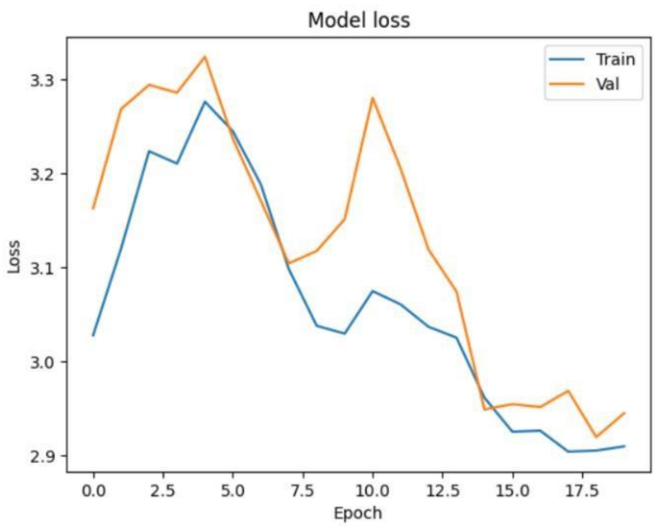
1.11

a. *Learning Rate = 0.0001:*



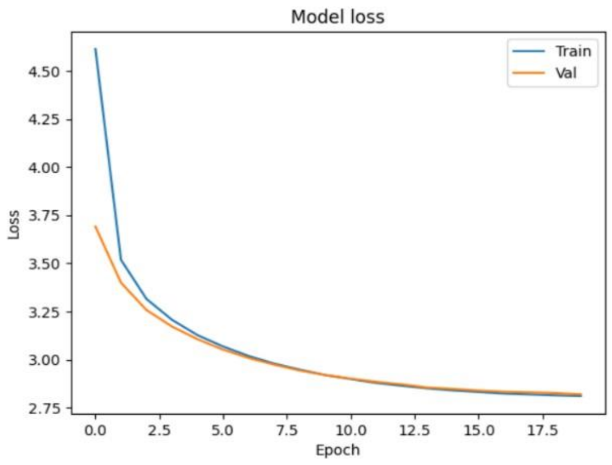
Both training and validation loss curves show a smooth and steady decrease indicating consistent learning and generalization without overfitting.

b.*Learning Rate = 0.001*:



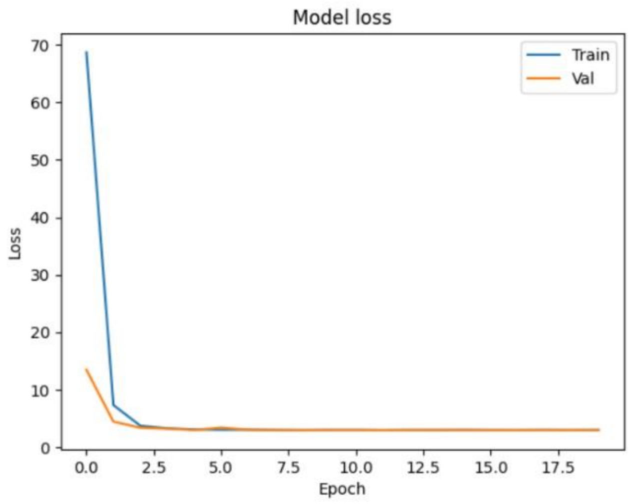
The loss decreases faster than for 0.0001, but validation loss has higher fluctuations suggesting some degree of overfitting or learning instability.

c. ***Learning Rate = 0.01:***



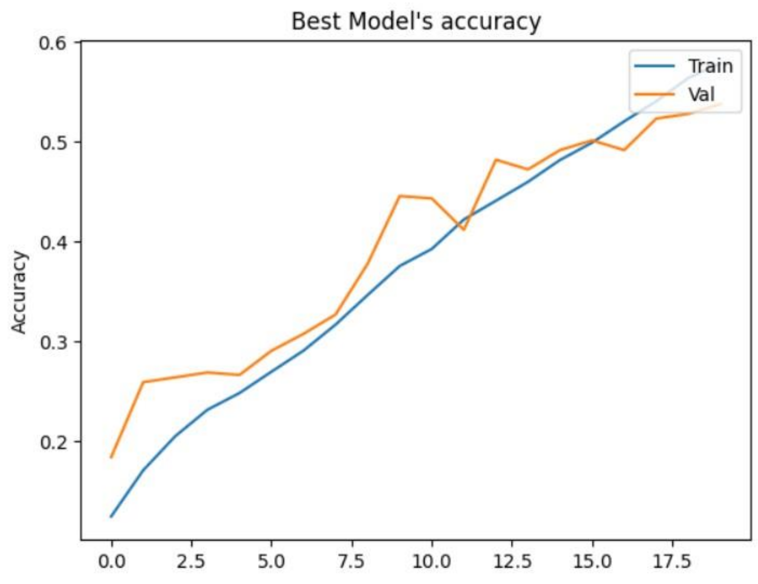
The training loss decreases rapidly but with significant fluctuations and the validation loss shows high instability. This indicates that the model struggles to converge and generalize well.

d.***Learning Rate = 0.1:***



The training loss drops very quickly at the beginning but does not stabilize and the validation loss barely decreases after the first few epochs. This suggests divergence due to an excessively high learning rate.

The best model selection:



The model trained with a learning rate of **0.0001** is the best choice. This learning rate ensures stable convergence and balanced performance on both training and validation sets, achieving training accuracy of 57.89% and validation accuracy of 53.75%. The steady decrease in loss without significant fluctuations demonstrates its capability to generalize well to unseen data, making it the optimal learning rate for this experiment.

2.Compare the network with state-of-the-art networks.

Instead of training a convolutional neural network (CNN) from scratch which would be challenging given the dataset's relatively small size we can employ **transfer learning** by utilizing **DenseNet121** (a pre-trained CNN) . This model was pre-trained on the large **ImageNet** dataset which contains 1.2 million images across 1,000 categories.

In our implementation:

1. **Base Model**: DenseNet121 was loaded with pre-trained ImageNet weights. This pre-trained model serves as the backbone for feature extraction.
2. **Transfer Learning**: The base model's weights were initially frozen meaning these layers were treated as a fixed feature extractor. Then we added custom layers for our specific dataset and fine-tuning the network to classify the 17 classes in the dataset.
3. **Fine-Tuning**: After the initial training we unfroze the last few layers of DenseNet121 and re-trained them on our dataset to improve performance further. This step allowed the model to adapt to domain-specific features while retaining the knowledge from ImageNet.

This approach highlights the efficiency and practicality of transfer learning. By leveraging a pre-trained model we can avoid the need for extensive data and computational resources required to train a CNN from scratch. Instead we can effectively use the pre-trained DenseNet121 to achieve better performance.

1.12

DenseNet121 was selected as a state-of-the-art pre-trained model as it have some key benefits.

*1. **Efficient Feature Utilization***

- DenseNet121 have dense connections where each layer directly accesses the feature maps of all preceding layers. This allows it to reuse features effectively reducing redundancy and making feature learning more efficient.

*2. **Parameter Efficiency***

- DenseNet121 has fewer parameters compared to other deep architectures with similar or better performance. This makes it computationally efficient while maintaining high accuracy especially when resources are limited.

3. ***Strong Representational Power***

- Pre-trained on ImageNet DenseNet121 provides robust feature representations that generalize well to new tasks and datasets. This strong initialization enhances the performance of transfer learning tasks.

4. ***Gradient Flow and Training Stability***

- The dense connections help mitigate the vanishing gradient problem allowing gradients to flow more easily through the network during training. This improves optimization and makes fine-tuning more effective.

5. ***Versatility Across Tasks***

- DenseNet121 has shown competitive performance across various computer vision tasks making it a versatile choice for both classification and feature extraction in diverse applications.

6. ***Pre-Trained Weights Availability***

- DenseNet121's pre-trained weights on ImageNet are readily available which simplifies implementation and reduces the need for extensive computational resources to train a large network from scratch.

7. ***Improved Generalization***

- By using dense connections and efficient feature reuse DenseNet121 tends to generalize better to datasets with limited samples as it effectively captures complex patterns in the data.

These advantages make DenseNet121 a practical and powerful choice for fine-tuning in transfer learning scenarios.

1.13

Steps for loading Fine-Tuning a Pre-Trained Model

1. Load the Pre-Trained Model.

```
# Load the pre-trained ResNet50 model
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(img_height, img_width, 3))
```

- DenseNet121 is loaded with pre-trained weights from ImageNet.
- include_top=False excludes the final dense (classification) layers of DenseNet making it possible to add the custom classification head.
- input_shape=(img_height, img_width, 3) adapts the model to take input images of shape 224×224×3224×224×3 (consistent with the dataset).

This step ensures the base DenseNet121 model is initialized with rich feature representations learned from ImageNet which can transfer to the jute_pest_dataset.

2. Freeze the Pre-Trained Layers Initially

Before training the pre-trained layers are frozen so their weights don't get updated.

```
base_model.trainable = False  # Freeze base model's layers
```

- This ensures that the pre-trained convolutional layers act as a fixed feature extractor during initial training.
- The custom layers (added next) are trained on top of these frozen DenseNet121 features.

3. Add Custom Layers for the Dataset

We added a custom classification head on top of DenseNet121.

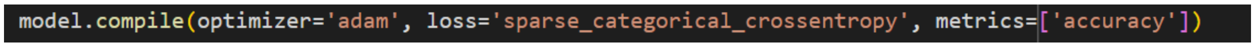
```
model = models.Sequential([
    base_model,
    layers.GlobalAveragePooling2D(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation='softmax')
])
```

- GlobalAveragePooling2D: Converts the feature maps from the DenseNet121 backbone into a single vector per image.
- Dense(128, activation='relu'): Adds a fully connected layer with 128 neurons and ReLU activation.
- Dropout(0.5): Adds regularization to prevent overfitting.
- Dense(num_classes, activation='softmax'): Adds an output layer with num_classes neurons (17 in this dataset) for multi-class classification using the softmax activation.

This head is specific to our jute_pest_dataset and adapts DenseNet121 to the new task.

4. Compile the Model.

The model is compiled with the Adam optimizer and categorical cross-entropy loss:

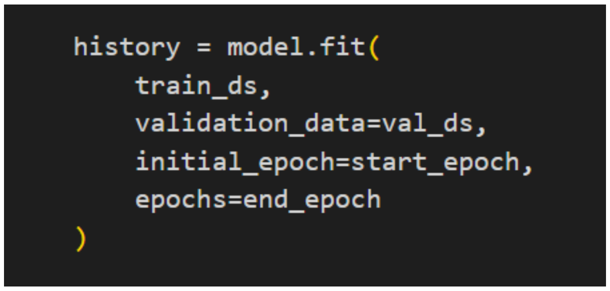


```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

- Loss Function: Since we are working with multi-class classification, sparse_categorical_crossentropy is used.
- Metric: Accuracy is tracked during training and validation.

5. Train the Model Initially

The model is first trained with the pre-trained DenseNet121 layers frozen.

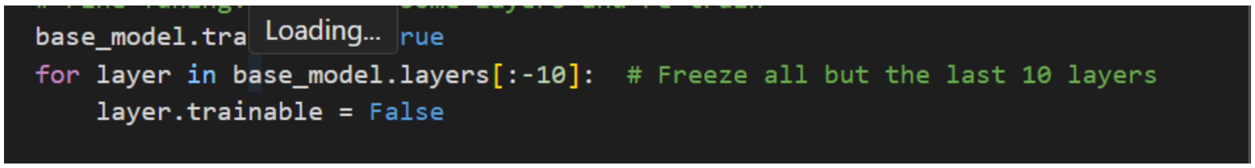


```
history = model.fit(
    train_ds,
    validation_data=val_ds,
    initial_epoch=start_epoch,
    epochs=end_epoch
)
```

- During this phase only the custom classification layers added to the DenseNet121 model are trained.
- The weights of the pre-trained DenseNet121 layers remain fixed.

6. Fine-Tune the Model

After initial training the DenseNet121 layers are partially unfrozen for fine-tuning:



```
base_model.tra Loading... rue
for layer in base_model.layers[:-10]:  # Freeze all but the last 10 layers
    layer.trainable = False
```

- Here all but the last 10 layers of DenseNet121 are frozen allowing the last few convolutional layers to be fine-tuned.
- Fine-tuning is done with a lower learning rate to preserve the pre-trained weights while adjusting them slightly for our dataset.

The fine-tuned model is then recompiled with a smaller learning rate. Fine-tuning allows the model to adjust its high-level features for jute pest classification leading to improved performance on the dataset.

1.15

Initial Training (Epochs 1-10):

- **Epoch 1**: Training Loss = 4.1250, Validation Loss = 2.4836
- **Epoch 2**: Training Loss = 2.5669, Validation Loss = 2.3079
- **Epoch 3**: Training Loss = 2.3861, Validation Loss = 2.1801
- **Epoch 4**: Training Loss = 2.2732, Validation Loss = 2.0830
- **Epoch 5**: Training Loss = 2.1571, Validation Loss = 1.9807
- **Epoch 6**: Training Loss = 2.0588, Validation Loss = 1.9273
- **Epoch 7**: Training Loss = 2.0205, Validation Loss = 1.9157
- **Epoch 8**: Training Loss = 1.9302, Validation Loss = 1.9061
- **Epoch 9**: Training Loss = 1.8813, Validation Loss = 1.8535
- **Epoch 10**: Training Loss = 1.8834, Validation Loss = 1.8714

Fine-Tuning (Epochs 11-20):

- **Epoch 11**: Training Loss = 3.434, Validation Loss = 2.3643
- **Epoch 12**: Training Loss = 2.8276, Validation Loss = 2.2376
- **Epoch 13**: Training Loss = 2.5696, Validation Loss = 2.1282
- **Epoch 14**: Training Loss = 2.4165, Validation Loss = 2.0542
- **Epoch 15**: Training Loss = 2.2692, Validation Loss = 2.0047
- **Epoch 16**: Training Loss = 2.2003, Validation Loss = 1.9708
- **Epoch 17**: Training Loss = 2.1251, Validation Loss = 1.9477
- **Epoch 18**: Training Loss = 2.1119, Validation Loss = 1.9305

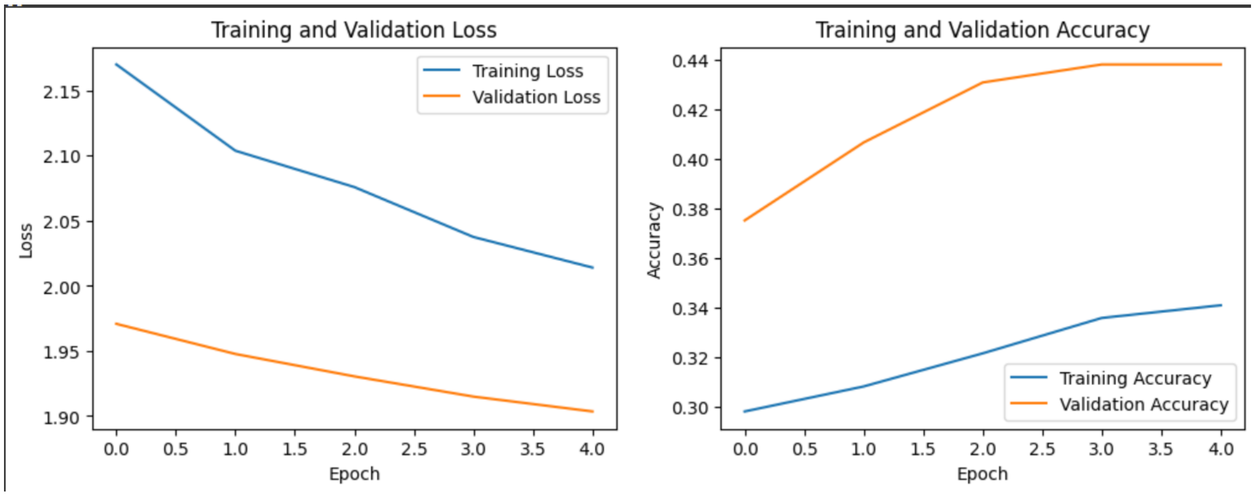
- **Epoch 19**: Training Loss = 2.0544, Validation Loss = 1.9150
- **Epoch 20**: Training Loss = 2.0331, Validation Loss = 1.9035

These loss values track the model's performance over each epoch reflecting the changes in both training and validation losses during the training and fine-tuning stages.

1.16

The evaluation result shows that the fine-tuned model performed with the following metrics:

- **Test Accuracy**: 46%
- **Test Loss**: 1.76



Observations from the plot:

1. **Training and Validation Loss**:
   - The training loss is decreasing over the epochs indicating the model is learning and fitting better to the training data.
   - The validation loss also decreases although at a slower rate which suggests that the model is generalizing well to unseen data.
2. **Training and Validation Accuracy**:
   - The training accuracy increases steadily which is typical as the model learns over time.
   - The validation accuracy shows improvement as well but it is slightly lower than the training accuracy. This is a sign that the model is not overfitting and is performing reasonably well on the validation set.

1.17

Comparison of Test Accuracy: Custom CNN vs. Fine tuned state-of-the-art-model.

1. ***Custom CNN Model***:

    - **Test Accuracy**: 7%
    - The custom CNN model shows extremely low performance. The accuracy is significantly low which is a sign that the model is not able to learn the underlying patterns in the data. Additionally the confusion matrix and classification metrics (precision, recall, F1-score) further emphasize that the model struggles with correctly classifying the samples. The fluctuations in the validation accuracy and low overall metrics suggest poor generalization and the model could not effectively learn meaningful features from the dataset.

2. ***Fine- tuned state-of-the-art-model:***

    - **Test Accuracy**: 46%
    - The fine-tuned model shows a much higher test accuracy of 46% indicating better generalization and performance on the testing dataset. While this result still leaves room for improvement the fine-tuned model outperforms the custom CNN by a significant margin, demonstrating the effectiveness of transfer learning in utilizing pretrained models to enhance performance.

Analysis:

- The custom CNN model struggles with both training and validation accuracy indicating it has not captured any meaningful patterns from the data. The poor results across multiple metrics (accuracy, precision, recall, F1-score) suggest that either the architecture of the model or the training process (such as hyperparameters or dataset size) needs significant improvement.
- On the other hand the fine-tuned model leverages pretraining on ImageNet which provides a robust feature extractor that helps the model perform better on the given task. The substantial increase in accuracy demonstrates the power of transfer learning when fine-tuned for a specific task.

Conclusion:

The fine-tuned model significantly outperforms the custom CNN model in terms of test accuracy (46% vs. 7%). This highlights the advantages of using pre-trained models for tasks with limited data or when developing custom models from scratch is challenging. Transfer learning as demonstrated can lead to a more effective and efficient solution in many deep learning tasks.

1.18

Trade-offs, Advantages and Limitations of Using a Custom Model vs. a Pre-Trained Model

Custom Model

Advantages:

1. ***Tailored Architecture***: A custom model can be designed specifically for the problem at hand allowing flexibility in choosing the architecture, number of layers, and hyperparameters. This can lead to better optimization for the specific task.
2. ***Control Over Training***: With a custom model we have full control over the training process including how the model learns from the data, the data augmentation techniques used and the way we handle overfitting or underfitting.
3. ***No Dependency on External Data***: Custom models don't require pre-trained datasets like ImageNet so there are no external data dependencies which may be useful when we have a unique or highly specific dataset.

***Limitations*:**

1. ***Data Efficiency***: Training a custom model from scratch typically requires a large amount of labeled data. If the dataset is small the model may not generalize well leading to poor performance as seen with the low test accuracy of the custom CNN in this case.
2. ***Longer Training Time***: Custom models require significant time and computational resources to train from scratch particularly for deep architectures. This may not be feasible in time-sensitive applications.
3. ***High Risk of Overfitting***: Without sufficient data or regularization techniques a custom model is prone to overfitting as it might learn specific patterns in the training data that don’t generalize well to new unseen data.

Pre-Trained Model (e.g., ResNet, DenseNet, VGG)

Advantages:

1. ***Faster Training***: Since the model is pre-trained on large datasets like ImageNet we can fine-tune the model rather than train it from scratch which significantly reduces training time and computational costs.
2. **Better Generalization**: Pre-trained models benefit from learning on a large and diverse dataset which enables them to capture robust, generalized features. Fine-tuning these features on your specific task can yield much better performance especially when data is limited.

3. ***Improved Accuracy with Transfer Learning****:* Pre-trained models often achieve better accuracy as seen in the results because they leverage knowledge from large-scale datasets which custom models without sufficient data may fail to learn.
4. ***Lower Data Requirements****:* Since pre-trained models already contain knowledge from large datasets they can often perform well with smaller amounts of task-specific data making them more suitable for scenarios with limited labeled data.

Limitations:

1. ***Less Flexibility***: Pre-trained models are not as customizable as custom-built models as we are constrained by the architecture and the features learned from the original dataset. While fine-tuning can help adapt the model to a new task it may not always fully align with the specific nuances of the problem.
2. ***Large Model Size****:* Pre-trained models like ResNet and DenseNet are often quite large which can be a concern for deployment in environments with limited computational resources or memory. This can affect inference speed and model efficiency in real-time applications.
3. ***Risk of Negative Transfer****:* If the pre-trained model's original dataset is too dissimilar from the target dataset fine-tuning might not work well and the model may actually perform worse than a custom model trained from scratch. This is particularly the case in tasks where the domain of the source and target data differs significantly.
4. ***Dependency on External Datasets***: Using pre-trained models means depending on external datasets (e.g., ImageNet). In some cases the model may carry biases or limitations from the dataset it was trained on which may not be optimal for the specific problem.

Conclusion:

- ***Custom Model***: Best when you have a large and domain-specific dataset and want complete control over the architecture. It might be the better choice for highly specialized tasks but is prone to underperformance with limited data and a longer training process.
- ***Pre-Trained Model****:* Ideal when you have limited data need quicker training and want better generalization. It often leads to faster results and better performance due to the transfer of learned features from large datasets. However it comes with a trade-off in terms of flexibility and dependency on the original dataset.

4 .GitHub Profiles:

1. https://github.com/AvindaShamal/Jute-Pest-CNN

2. https://github.com/nipunanisha/state-of-the-art-convolutional-neural-networks