Insect Pest Recognition using CNN models

https://github.com/RailgunDotEnc/ML_CNN_Project

Group 9	Ari Hu	Daniel Morandi	Daniel Nguyen	Joel Varghese
	axh200041	drm210004	dnn200003	jjv200001

1 Introduction (5pt)

Image recognition methods have made it easier to detect patterns given a dataset of pictures. One use where Image recognition is helpful in real life is detecting if an insect is a pest, which can help reduce agricultural damage. The dataset we incorporated was IP102, a dataset that insect pest images categorized by pests that exist in daily life. The goal of this report is to use the IP102 dataset [5], three Convolutional Neural Networks(CNN) ResNet18, GoogleNet, and MobileNetV3.

Our approach is to train each model with the dataset and compare the accuracy and other parameters to find the best CNN model for insect pest classification. A motivation for finding the best model is that it can help benefit farmers or gardeners that experience insect damages for there crops. One such person can take periodic pictures of their crops, and feed the pictures to the best evaluated model trained on the IP102 dataset. After the pests are identified, one can make better measures in protecting their agriculture, by using techniques tailored to a specific insect.

The main experiment was running each model with 100 rounds(iterations) and analyzing the data. The result was that the best training accuracy and test accuracy within 3% of margins to minimize over-fitting was GoogleNet at 70.3 and 67.6 respectively. However, MobileNetV3 and ResNet18 also exhibit similar peak accuracy at 66-70 range with Resnet18 taking the least rounds and MobileNetV3 taking the most.

2 Task (5pt)

The primary objective of this project is to deploy Convolutional Neural Networks (CNNs) to accurately identify and classify insect pests from images. This task involves developing a machine learning model capable of discerning between various insect species, specifically those that are detrimental to agriculture. The task can be framed as a multi-class image classification problem where the input is an image of an insect and the output is the classification label identifying the specific type of pest. Each input to the model is a digital image, represented as an array of pixel values. These images are sourced from the IP102 dataset, which includes a variety of insect pest images under natural settings. The output for each image is a categorical label from one of the 102 distinct classes that classify the insect. Each label corresponds to a specific type of insect pest, as identified by agricultural experts.

Let X represent the set of the input images where xi is a member of the set X is an individual image. Correspondingly, let Y represent the set of labels, where each label yi is a member of the set Y is one of the 102 possible categories. The task for the CNN models is to learn the mapping f : X $\rightarrow$ Y. By leveraging the strengths of three different CNN architectures — Resnet18, GoogleNet, and MobileNetV3 — this project aims to evaluate which model performs best in terms of accuracy, computational efficiency, and practical applicability in aiding agricultural practices. This evaluation is crucial as it impacts the recommendations for real world applications, where farmers and gardeners can use the model outputs to make informed decisions about pest control measures.

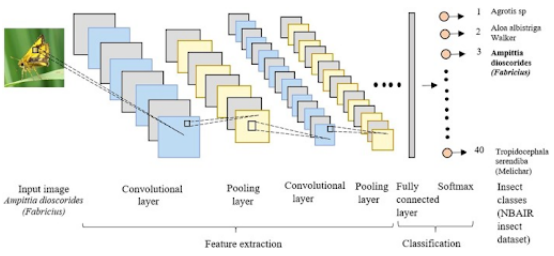

Figure 1: Insect Image to CNN

3 Data (5pt)

The dataset used for our models was IP102_FC_EC obtained from Kaggle [5]. The data includes 75,222 samples of insect pest images belonging to 102 categories. The 102 classes was decided by agricultural experts and structured hierarchical to have the insect pest(sub-class) assign to the the crop(super-class) linked to the pest. The insect pest and crop images were sourced from common search engines with the top 2000 results given each sub-class. The data is then manually filtered to remove data with no pest or more than one pests. The annotations for this data set was gathered from agricultural experts manual categorizing the data.

Figure 2: IP102 Dataset

The dataset split is 6 : 1 : 3 for training, validation, and testing at a sub-class level. This leaves 45,095 training, 7,508 validation, and 22,619 testing images for classification. The dataset also has a subset of annotated data, with 15,178 for training and 3,7,98 for testing. IP102 also has a hierarchy system. For example the 8 crops are grouped into two super classes, economic crop(EC) and field crop(FC). The full details of the hierarchy is found in Figure 3.

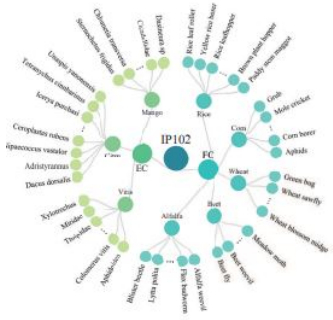

Figure 3: Taxonomy of IP102 dataset

4 Methodology (20pt)

The three models used are Convolutional Neural Networks(CNN). CNN are feed forward networks that can analyze images due to its 2D structure. The images of insect pests therefore can be represented as a array of pixel values. The CNN is also build from multiple hidden layers. The convolution layer performs the convolution operation, which is used for feature extraction. A small sliding window called a kernel is used to scan the input data, and to perform element-wise multiplication and summation to create feature maps. This feature map is fed into the ReLU layer for finding features based on the generated feature map.

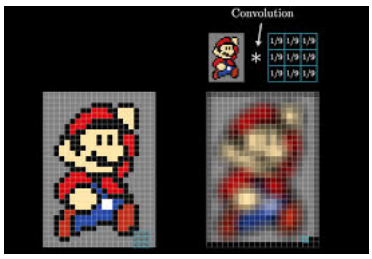

Figure 4: Example of Convolution

The next hidden layer is Pooling, which reduces the dimensions of the feature map to identify parts like corners and edges. There are two types of pooling that are commonly used in CNNs. Average Pooling which calculates average value in window, reduces the dimensions and preserves general info, and Max Pooling which calculates the max value within a pooling window and is used when extracting dominant features or complex patterns. After the hidden layers, the pooled feature map is flattened to a connected layer to get the output [3].

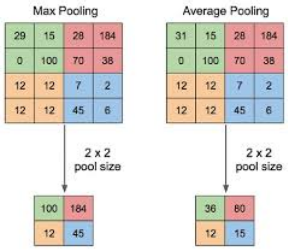

Figure 5: Example of Pooling

The main parameters of CNN that can be used to adjust the model are channels, padding, and strides. Channels, refer to the depth dimension

of a feature map or an image, and allows CNNs to capture information and features. Padding is a technique of adding border pixels to the input data before convolution, that preserves spatial dimensions and avoid edge information loss. Padding can help ensure that the output feature map has the same spatial size as the input. Finally, Strides determine the kernel's step size during convolution, where a Stride of 1 means the kernel moves one pixel at a time. Larger stride values reduce spatial dimensions of the output features.

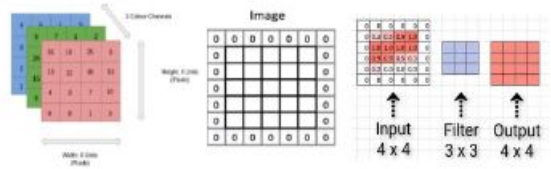

Figure 6: Example of Channels, Padding, and Stride

Another technique we used to make the CNNs faster is Batch Normalization. Batch Normalization is a technique to help coordinate the update of multiple layers in the model by adjusting the layer's output in a way that ensures consistent scaling. The activation's are then re-scaled for each input variable within a mini-batch.

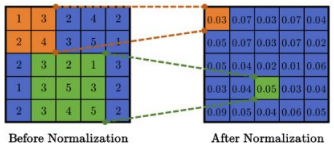

Figure 7: Example of Batch Normalization

ResNet is a deep CNN that integrate features, which can be stacked for higher depth. Previous deep networks like VGG net, experiences degradation, a sharp decrease in accuracy followed by higher depth. Resnet addresses this issue by having a deep residual learning network that consists of a main path, and a skip connection, where one or more layers in the main path can be bypassed. These shortcuts follow a desired underlying mapping, which makes it easier to optimize with no added parameters or computational complexity. The model we would be using is ResNet18. ResNet18 starts with converting the immage to a 7x7 convolutional block. The output of that block is then max pooled and fed to 16 layers of 3x3 convolutional block of differing filter sizes. The output of this chain of blocks or a shortcut of these blocks is then average pooled, fed to a fully connected layer then soft-maxed for probability. [1].

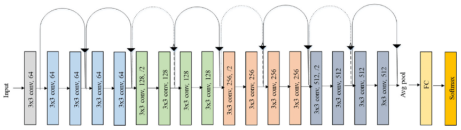

Figure 8: ResNet18

MobileNet is another type of CNN that focuses on high accuracy and mobile vision applications. MobileNetV1 included a depth wise separable convolution, a change of the convolution operation where each channel is convoluted independently with its own filter, to make efficient computations. MobileNetV2 improved on this by adding a resource efficient block with linear bottlenecks, where reducing dimensions of the input feature map is linear to make information flow efficient, and inverted residuals, where the output of the residual block is added to the input to reduce computation. The model we would be using is MobileNetV3 which further improves the previous versions by adding modified swish non-linearity, an upgraded activation function that is cheaper to compute, and Squeeze and Excitation blocks that optimize emphasizing features while suppressing less relevant ones [2].

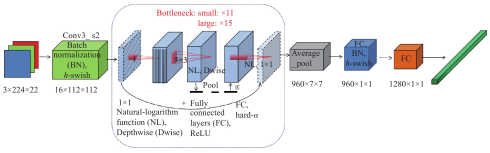

Figure 9: MobileNetV3

GoogleNet is another deep CNN that is used in image classification that significantly decrease the error rate of previous CNNs. GoogleNet adds 1x1 convolution, which reduced the amount of parameters and increases the depth of the architecture, Global Average Pooling, which average features maps to 1x1 to improve accuracy, the Inception module/layer, a parallel convolutional pathway at different scales(1x1,3x3,5x5) to capture local and global features, and Auxiliary Classifiers, convolutional and pooling layers which are added to intermediate layers during training. GoogleNet architecture consists of 2 convolutional layers and 3 inception layers [4].

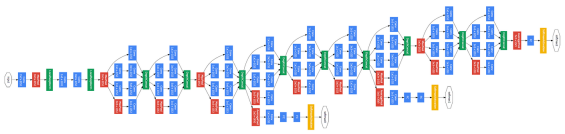

Figure 10: GoogleNet

For training our models, we use an Adam optimizer, which uses momentum to update weights. Our datasets our split with batch size of 256 on both the training and test sets, and the given Net was fed the images. After this, the test and training accuracy is calculated by comparing the output to the given labels, and the loss function is found using Cross Entropy Loss of the output to the labels.

5 Implementations (15pt)

The Resnet Implementation, seen in Listing 1, is implemented using PyTorch. This model architecture is built upon the concept of residual learning where shortcut connections are introduced to skip one or more layers. The architecture starts with a 7x7 convolutional layer that processes the input image, followed by batch normalization to stabilize and speed up the training. ReLU activation is used to provide non-linearity, allowing the model to learn more complex patterns. Max pooling is applied to reduce the dimensionality of input features, helping in making the detection of features invariant to scale and orientation changes. The core of ResNet18 is its residual blocks, defined in the 'make layer' function, which use a series of convolutions and a shortcut that skips these layers. The network concludes with a fully connected layer that transforms the learned "deep features" into the final output classes. ResNet18 benefits from deep residual learning by alleviating the vanishing gradient problem, which allows for deeper networks that are easier to optimize and can gain accuracy from considerably increased depth.

```python
class ResNet18(nn.Module):

    def __init__(self, block, layers, num_classes=1000):
        self.inplanes = 64
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(NUM_CHANNELS, 64, kernel_size=7, stride=2,
        ↪  padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(2)
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Listing 1: ResNet18 Implementation.

The MobileNetV3 implementation, seen in listing 2 is also implemented in PyTorch, and is optimized for mobile and edge devices, focusing on efficiency and speed. It features a configurable architecture that adapts the network size based on the 'config name', making it flexible for different computational power environments. The network uses a combination of depthwise separable convolutions that reduce the computational cost and parameter count. Bottleneck layers incorporate lightweight depthwise convolutions to filter features followed by a pointwise convolution to create new features. Squeeze-and-Excitation blocks re-calibrate channel-wise feature responses by explicitly modeling interdependencies between channels. The classifier consists of a global average pooling followed by a 1x1 convolution to match the number of output classes. MobileNetV3's efficiency in using computations and parameters

makes it particularly suitable for mobile applications without sacrificing much accuracy.

```
class MobileNetV3(nn.Module):
  def __init__(self, config_name ="large",in_channels = 3,classes =
 ↪  1000, ConvBlock=None,BNeck=None):
      super().__init__()
      config = self.config(config_name)
      self.Layer_Count=[2,4]

      # First convolution(conv2d) layer.
      self.conv = ConvBlock(in_channels, 16, 3, 2, nn.Hardswish())
      # Bneck blocks in a list.
      self.blocks = nn.ModuleList([])
      for c in config:
          kernel_size, exp_size, in_channels, out_channels, se, nl, s =
         ↪  c
          self.blocks.append(BNeck(in_channels, out_channels,
         ↪  kernel_size, exp_size, se, nl, s))

      # Classifier
      last_outchannel = config[-1][3]
      last_exp = config[-1][1]
      out = 1280 if config_name == "large" else 1024
      self.classifier = nn.Sequential(
          ConvBlock(last_outchannel, last_exp, 1, 1, nn.Hardswish()),
          nn.AdaptiveAvgPool2d((1,1)),
          ConvBlock(last_exp, out, 1, 1, nn.Hardswish(), bn=False,
         ↪  bias=True),
          nn.Dropout(0.8),
          nn.Conv2d(out, classes, 1, 1)
      )
      print(self.state_dict().keys())
      self.layers=[]
      for i in range(6):
          if i<self.Layer_Count[0]:
              self.layers.append(f"layer{i+1}")
          else:
              self.layers.append(None)
      print(self.layers)

  def _make_layers(self, in_channels, out_channels, block_cfg,
 ↪  width_multiplier):
      layers = []
      for cfg in block_cfg:
          expansion_factor, num_blocks, stride = cfg
          out_channels = int(out_channels * width_multiplier)
          for _ in range(num_blocks):
              layers.append(self.MobileNetV3Block(in_channels,
             ↪  out_channels, expansion_factor, stride))
              in_channels = out_channels
              stride = 1  # Only the first block in each stage has a
             ↪  stride > 1
      return nn.Sequential(*layers)


  def forward(self, x, volly=None):
      x = self.conv(x)
      x =self.blocks[0](x)
      x =self.blocks[1](x)
      x =self.blocks[2](x)
      x =self.blocks[3](x)
      x =self.blocks[4](x)
      x =self.blocks[5](x)
      x =self.blocks[6](x)
      x =self.blocks[7](x)
      x =self.blocks[8](x)
      x =self.blocks[9](x)
      x =self.blocks[10](x)
      x =self.blocks[11](x)
      x =self.blocks[12](x)
      x =self.blocks[13](x)
      x =self.blocks[14](x)

      x = self.classifier(x)
      y_hat =torch.flatten(x, 1)
      return y_hat
```

Listing 2: MobileNetV3 Implementation.

In the GoogleNet implementation seen in listing 3, the architecture begins with convolutional and max pooling layers that handle the input before it is passed to the inception modules. GoogleNet is renowned for its inception modules, where convolutional filters of different sizes are applied simultaneously to the input. Each inception block applies convolutions of different sizes on the same input and then concatenates the results, allowing the network to capture information at various scales. Auxiliary classifiers are introduced at intermediate points in the network to push useful gradients to the lower layers and improve training. The network also includes dropout for regularization, followed by a fully connected layer that outputs the predictions. GoogleNet's design leverages a deeper and wider network without an explosive growth in computational complexity, thanks to its innovative inception modules.

```
class GoogLeNetClient(nn.Module):
  def __init__(self, in_channels=3,
 ↪  num_classes=1000,conv_block=None,Inception_block=None):
      super(GoogLeNetClient, self).__init__()
      self.conv1 = conv_block(in_channels=in_channels, out_channels =
     ↪  64, kernel_size=7, stride =2, padding = 3)

      self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
      self.conv2 = conv_block(64, 192, kernel_size =3, stride=1,
     ↪  padding=1)
      self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
      self.inception3a = Inception_block(192, 64, 96, 128, 16, 32, 32)
      self.inception3b = Inception_block(256, 128, 128, 192, 32, 96,
     ↪  64)
      self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

      self.inception4a = Inception_block(480, 192, 96, 208, 16, 48, 64)
      self.inception4b = Inception_block(512, 160, 112, 224, 24, 64,
     ↪  64)
      self.inception4c = Inception_block(512, 128, 128, 256, 24, 64,
     ↪  64)
      self.inception4d = Inception_block(512, 112, 144, 288, 32, 64,
     ↪  64)
      self.inception4e = Inception_block(528, 256, 160, 320, 32, 128,
     ↪  128)
      self.maxpool4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
      self.inception5a = Inception_block(832, 256, 160, 320, 32, 128,
     ↪  128)
      self.inception5b = Inception_block(832, 384, 192, 384, 48, 128,
     ↪  128)

      self.avgpool = nn.AvgPool2d(kernel_size=2, stride=1)
      self.dropout = nn.Dropout(p=0.4)
      self.fc1 = nn.Linear(1024,num_classes)


  def forward(self, x):
      x = self.conv1(x)
      x = self.maxpool1(x)
      x = self.conv2(x)
      x = self.maxpool2(x)
      #Layer3
      x = self.inception3a(x)
      x = self.inception3b(x)
      x = self.maxpool3(x)
      x = self.inception4a(x)
      x = self.inception4b(x)
      x = self.inception4c(x)
      x = self.inception4d(x)
      x = self.inception4e(x)
      x = self.maxpool4(x)
      x = self.inception5a(x)
      x = self.inception5b(x)
      x = self.avgpool(x)
      x = x.reshape(x.shape[0],-1)
      x = self.dropout(x)
      x = self.fc1(x)
      return x
```

Listing 3: GoogleNet Implementation.

6 Experiments and Results (45pt)

Development and Test Results How we evaluated the model is by running each and collecting round, acc_train, acc_test, Loss, and Total Time data. We then output the results to an excel file using pandas Dataframes. Next, tables were

added showcasing select rounds(1,25,50,75,100) for each of the models. Observations of the tables show that Resnet18 took the less time and exhibit the highest acc_train in 100 rounds with the smallest loss at round 25, MobileNetV3 took the most time with the worst acc_train and acc_test in 100 rounds with consistent loss throughout the rounds, and GoogleNet took medium amount of time with the highest acc_test in 100 rounds.

Resnet18 Select Results :				
Round	acc_train	acc_test	Loss	Time(s)
1	59.6	45.8	0.74	8.9
25	83.4	71.8	0.59	80.0
50	92.3	73.7	0.75	163.4
75	95.6	75.8	0.88	241.6
100	96.6	75.4	0.98	319.2

MobileNetV3 Select Results :				
Round	acc_train	acc_test	Loss	Time(s)
1	54.9	45.8	0.69	8.1
25	67.4	65.2	0.62	88.3
50	75.5	68.8	0.62	169.1
75	83.3	70.2	0.66	253.1
100	88.7	72.3	0.77	333.0

GoogleNet Select Results :				
Round	acc_train	acc_test	Loss	Time(s)
1	57.9	45.8	0.69	3.2
25	82.7	72.4	0.64	82.6
50	91.3	76.1	0.72	165.0
75	94.6	76.7	0.79	245.3
100	96.1	76.5	0.94	325.6

Next, a table was created that recorded the best acc_train and acc_test data within 3% of each other to reduce over-fitting. The results we found were that GoogleNet exhibit the best accuracy at 70.3 and 67.7, with Resnet18 at second with 69.3 and 66.4, and MobileNetV3 with worst accuracy at 68.6 and 66.0. Resnet18 was the fastest to reach its peak accuracy at 7 rounds, compared to GoogleNet's 9 and MovileNetV3 32.

Best Accuracy given Model (within 3%) :			
Model	Round	acc_train	acc_test
Resnet18	7	68.6	66.0
MobileNetV3	32	69.3	66.4
GoogleNet	9	70.3	67.6

A hypothesis on why MobileNetV3 was the slowest and lowest accuracy given the margins was that it had the most layers with 28 compared with GoogleNet's 22 layers and ResNet18 18 layers. Models with more layers are slower, but they can capture more intricate details. However with the cost of capturing more details is the prone of overfitting where the 3% margin was broken at Round 33. But as a benefit for the extra layers, MobileNetV3 exhibits less Cross Entropy Loss compared to ResNet18 and GoogleNet on select rounds.

Error Analysis We conducted extensive error analysis to identify and address the limitations faced by each model. Adjusting hyper parameters such as learning rate and kernel size proved crucial in optimizing the performance of these CNNs. An improperly set learning rate would lead to slow convergence or unstable training trajectories. High rates would cause models to overshoot optimal weights while very low rates would prolong the training period unnecessarily.

The choice of kernel size in convolutional layers also had a significant impact on the model's ability to capture relevant features. Larger kernels would capture broader features but miss some critical details, while smaller kernels focused on finer details but overlooked contextual information. Incorrect hyper parameter configurations would frequently lead to under-fitting or over-fitting. Models would either fail to generalize effectively or they would capture excessive noise. The main classes of errors were over-fitting to training data, visual similarity errors, and scale and orientation errors. Unknown words did not seem to be a problem in our model.

Speed Analysis In theory, MobileNetV3 should be a lightweight architecture that would out perform Resnet18 and GoogleNet in terms of speed. This is because MobileNet consolidates the requirements through the use of Depth-wise and point-wise convolution. However in practice, the test results show that Resnet offers more accuracy compared to MobileNet with the same amount of speed. This is because depth-wise convolution is not supported with most GPU firmware. From the results, we can see that GoogleNet outperforms Resnet in terms of accuracy but has a trade-off in the amount of time required.

7 Conclusion (5pt)

The purpose of this project is to find the best CNN model for the IP102 dataset. We did this by

running each model with 100 rounds and recording select results in a table. We then extracted results that had less than 3% of margin between training accuracy and test accuracy.

By comparing the capabilities of our distinct CNN architecture out of the three to classify insect pests, we came to the following conclusions. GoogleNet proved to be the most effective model, achieving the highest test accuracy at 67.6 percent and training accuracy at 70.3 percent. ResNet18, while slightly less accurate, demonstrated remarkable efficiency by reaching peak accuracy faster than its counterparts. MobileNetV3, despite its design for mobile use which suggests high efficiency, took longer to train and offered slightly lower accuracy, highlighting a potential trade-off between design intentions and practical performance.

Our error analysis revealed several common challenges among the three architectures that called for strategic hyper parameter adjustments. The practical implications of these findings are substantial for the agricultural sector. Farmers and gardeners could leverage the insights provided by the most accurate model which is GoogleNet, to detect and manage pest threats more effectively.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[2] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019.

[3] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[5] Xiaoping Wu, Chi Zhan, Yu-Kun Lai, Ming-Ming Cheng, and Jufeng Yang. Ip102: A large-scale benchmark dataset for insect pest recognition. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8779–8788, June 2019.