

# Neural Networks Project

## Mushroom Image Classifier

### Table of Contents

Dataset and Preprocessing .....	2
Environment and Decisions .....	4
Data Preprocessing .....	5
Model Selection .....	5
<b>Training Methodology</b> .....	6
Results and Observations .....	10
Tracking Metrics .....	10
ResNet18 .....	12
Model Performance Results .....	13
Observations .....	18
Confusion Matrices .....	19
Predictions .....	20
Conclusion .....	21
Bibliography .....	22

All the assets of the project, such as code, dataset and results are also available on my Github [repository](#).

Project by: Ruse Cristian Andrei

Group: 1242A

## Dataset and Preprocessing

I made use of this [dataset](#) from Kaggle, comprised of 27436 photos of 94 species of mushrooms. I decided to use only 30 species for my classes, as running the training models with 94 classes would have taken quite a significant amount of time and 30 is probably more than sufficient to display how a neural network works.

In preparing the dataset for training, it's essential to ensure uniformity in image dimensions. Convolutional Neural Networks (CNNs) require consistent input sizes to function effectively. Resizing all images to a standard square shape not only simplifies the design of the input layer but also enhances computational efficiency.

However, when resizing images, it's crucial to consider the original resolutions to avoid upscaling, which can introduce artifacts and degrade image quality. Therefore, it's advisable to set the target resize dimensions based on the smallest images in the dataset that meet a minimum acceptable resolution threshold. In this dataset, most images have a resolution of 640x480 pixels, but there are some outliers which have low resolutions such as 180x220 pixels, and quite a handful having resolution around 400x400 pixels therefore, I decided that images with resolutions below 400x400 pixels should be removed with a script.

```
C:\Users\ruse\Desktop\ProjectNN\pythonProject7\.venv\Scripts\python.exe
Class: Agaricus augustus, Remaining: 329, Deleted: 12
Class: Amanita amerirubescens, Remaining: 316, Deleted: 20
Class: Amanita augusta, Remaining: 319, Deleted: 12
Class: Amanita brunnescens, Remaining: 284, Deleted: 22
Class: Amanita calyptroderma, Remaining: 323, Deleted: 5
Class: Amanita flavoconia, Remaining: 299, Deleted: 22
Class: Amanita muscaria, Remaining: 291, Deleted: 37
Class: Amanita persicina, Remaining: 246, Deleted: 17
Class: Amanita phalloides, Remaining: 313, Deleted: 8
Class: Amanita velosa, Remaining: 273, Deleted: 10
Class: Armillaria mellea, Remaining: 310, Deleted: 18
Class: Armillaria tabescens, Remaining: 310, Deleted: 43
Class: Artomyces pyxidatus, Remaining: 320, Deleted: 32
Class: Bolbitius titubans, Remaining: 300, Deleted: 21
Class: Boletus pallidus, Remaining: 304, Deleted: 16
Class: Boletus rex-veris, Remaining: 297, Deleted: 11
Class: Cantharellus californicus, Remaining: 304, Deleted: 21
Class: Cantharellus cinnabarinus, Remaining: 307, Deleted: 26
Class: Cerioporus squamosus, Remaining: 532, Deleted: 49
Class: Chlorophyllum brunneum, Remaining: 253, Deleted: 24
Class: Chlorophyllum molybdites, Remaining: 212, Deleted: 24
Class: Galerina marginata, Remaining: 223, Deleted: 11
Class: Ganoderma applanatum, Remaining: 228, Deleted: 95
Class: Ganoderma curtisii, Remaining: 207, Deleted: 53
Class: Ganoderma oregonense, Remaining: 244, Deleted: 3
Class: Ganoderma tsugae, Remaining: 261, Deleted: 8
Class: Phaeolus schweinitzii, Remaining: 280, Deleted: 13
Class: Phlebia tremellosa, Remaining: 257, Deleted: 29
Class: Phyllotopsis nidulans, Remaining: 275, Deleted: 22
Class: Pleurotus ostreatus, Remaining: 283, Deleted: 15
-----
Total images remaining: 8700
Total images deleted: 699

Process finished with exit code 0
```

Fig. 1 Low Resolution Photo Script Remover Output

In total, 699 low resolution photos were deleted, now remaining 8700 photos across the 30 classes. Each class has currently around 250-300 photos on average, whilst, whilst the lowest one has 207, particularly “*Ganoderma Curtisii*”, and the highest one being “*Cerioporus Squamosus*” with 532 photos.

The next step was to divide the dataset into the training, validation and test subsets. I opted for an 80:10:10 split, 80-training, 10-validation, 10-test. For an easier understanding and visualization, I have made histograms for the subsets. On average, it resulted in the train subset having around 250 images per class, and the validation and test having around 30 images per class

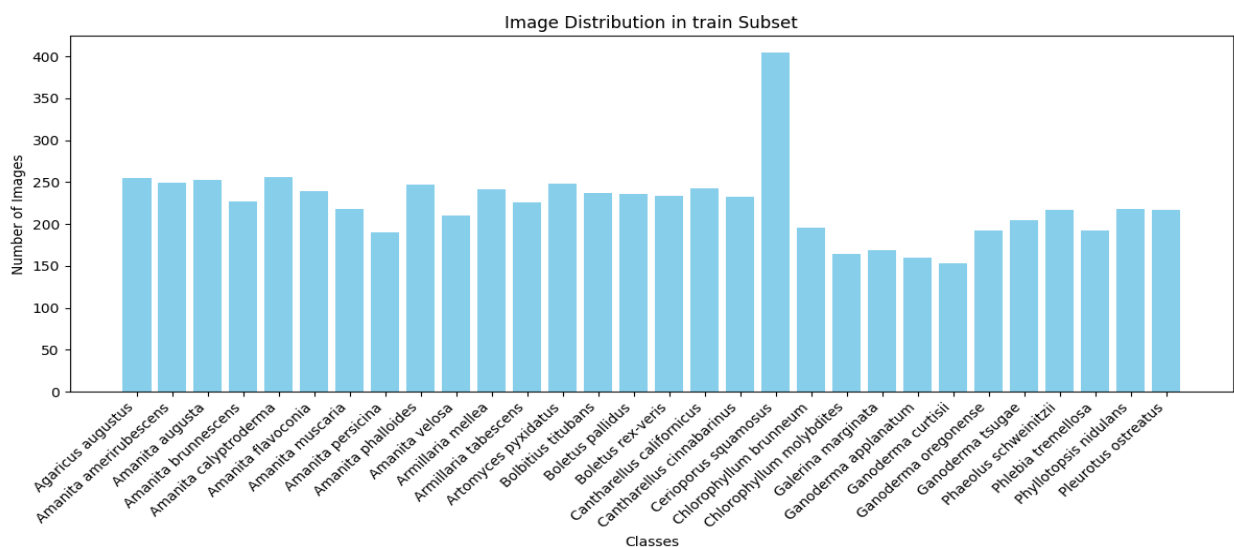


Fig. 2 Train subset Histogram

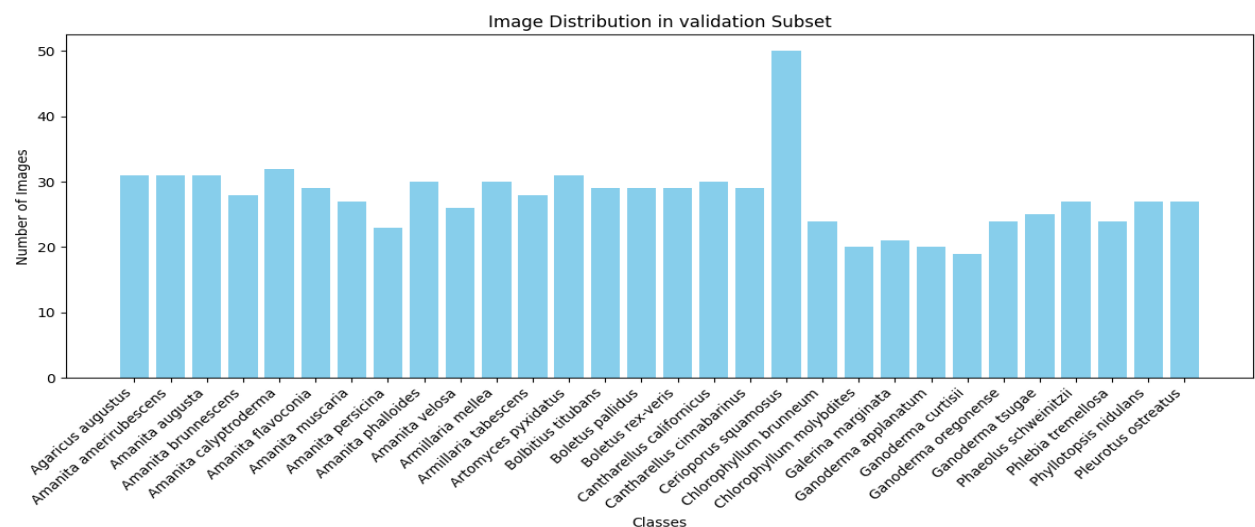


Fig. 3 Validation subset Histogram

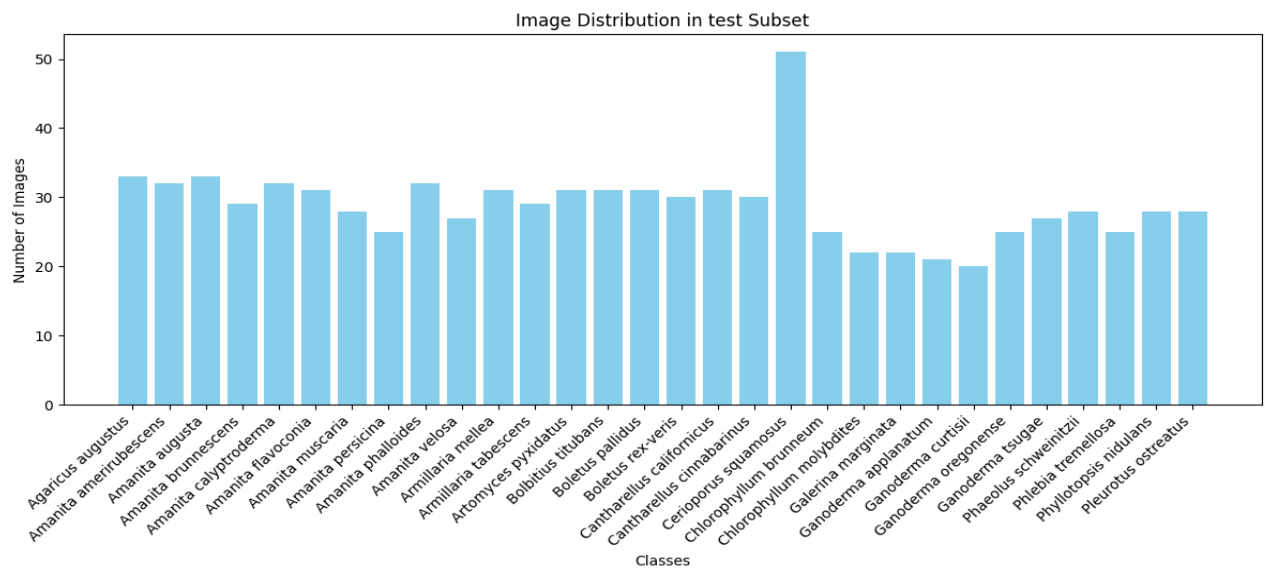


Fig. 4 Test subest Histogram

## Environment and Decisions

For this project, I selected **PyTorch** as the primary deep learning framework due to its flexibility, ease of debugging, and compatibility with pre-trained models. PyTorch is widely recognized for its dynamic computation graphs and straightforward integration with GPU acceleration, making it an excellent choice for tasks involving large datasets and complex neural networks.

I utilized a **dedicated GPU** to accelerate training. GPUs are specifically optimized for parallel processing, allowing them to handle the large matrix computations resulting in significantly faster training times.

```
20
21 # Check for GPU availability
22 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
23 print(f"Using device: {device}")
24 if torch.cuda.is_available():
25     print(f"Number of GPUs: {torch.cuda.device_count()}")
26     for i in range(torch.cuda.device_count()):
27         print(f"GPU {i}: {torch.cuda.get_device_name(i)}")
28
```

Fig. 5 GPU selection for training

## Data Preprocessing

The dataset comprises images from multiple classes, which were processed to ensure consistency and enhance model generalization. The following preprocessing steps were applied:

1. **Uniform Image Dimensions:** Images were resized to **400x400 pixels** to standardize input dimensions across the dataset
2. **Augmentation:** Techniques such as random horizontal flipping and rotation were employed to artificially increase data variability ( for training only ), improving model robustness.
3. **Normalization:** Pixel values were normalized to match the expectations of pre-trained models we are going to use:

```
28
29 # Define data transformations
30 data_transforms = {
31     'train': transforms.Compose([
32         transforms.Resize((img_height, img_width)),
33         transforms.RandomHorizontalFlip(),
34         transforms.RandomRotation(20),
35         transforms.ToTensor(),
36         transforms.Normalize( mean: [0.485, 0.456, 0.406], std: [0.229, 0.224, 0.225])
37     ]),
38     'test': transforms.Compose([
39         transforms.Resize((img_height, img_width)),
40         transforms.ToTensor(),
41         transforms.Normalize( mean: [0.485, 0.456, 0.406], std: [0.229, 0.224, 0.225])
42     ])
43 }
44
```

Fig. 6 Data Preprocessing

## Model Selection

In this project, we utilized two pre-trained convolutional neural networks from PyTorch, each with distinct architectural features and design philosophies:

### MobileNetV2:

1. MobileNetV2 is built for computational efficiency. It uses techniques like depthwise separable convolutions to significantly reduce the number of calculations needed while still achieving good accuracy.
2. This model is more balanced, it supposedly does not have the same level of depth and complexity, but it is an agreeable enough option for the amount of processing power it requires

### ResNet18:

1. **Residual Connections:** ResNet18 employs skip connections, allowing the output of a layer to bypass one or more layers and directly contribute to subsequent layers. This design mitigates the vanishing gradient problem, enabling the training of deeper networks.
2. **Standard Convolutional Layers:** ResNet18 uses traditional convolutional layers organized into blocks, each followed by batch normalization and ReLU activation. This straightforward structure enables effective feature extraction.
3. **Deeper Architecture:** With 18 layers, this model is designed to extract more complex and nuanced features, making it better suited for datasets with higher variance and complexity.

I decided to select these 2 particular models as from the research I've done it appears that MobileNetV2 is a more simplistic model, made for low resource device( ie mobiles, that's where the Mobile comes from), whilst ResNet18 is more advanced, thus I think it might be interesting to see if at the end of training there is a substantial difference between the two models in terms of accuracy of predicting correctly the classes.

### Training Methodology

The training process focused on optimizing model performance while minimizing overfitting. Key aspects of the methodology include:

1. **Early Stopping:** Inspired by TensorFlow's implementation, early stopping was incorporated to monitor validation loss. If validation loss failed to improve over 5 consecutive epochs, training was halted. This approach prevents overfitting and conserves computational resources:

```
111
112     if val_loss < best_val_loss:
113         best_val_loss = val_loss
114         best_epoch = epoch + 1
115         torch.save(model.state_dict(), model_name)
116         print(f"Model saved as {model_name}")
117         early_stop_counter = 0
118     else:
119         early_stop_counter += 1
120
121     if early_stop_counter >= patience:
122         print("Early stopping triggered.")
123         break
124
```

Fig. 7. Early Stopping Code Implementation

2. **Optimizer and Loss Function:** The Adam optimizer was chosen for its adaptive learning rate capabilities, ensuring efficient convergence. Cross-entropy loss was used as the objective function, aligning with the multi-class classification nature of the task.

Here is the full code for the training file, with comment explanations

```
12 base_dir = "mushrooms" # Base directory containing the dataset
13 train_dir = os.path.join(base_dir, "train") # Directory for training data
14 val_dir = os.path.join(base_dir, "validation") # Directory for validation data
15 test_dir = os.path.join(base_dir, "test") # Directory for test data
16
17 # Set parameters
18 img_height, img_width = 400, 400 # Standard size to which all images will be resized
19 batch_size = 32 # Number of samples per batch during training and validation
20
21 # Check for GPU availability
22 device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # Use GPU if available
23 print(f"Using device: {device}")
24 if torch.cuda.is_available():
25     print(f"Number of GPUs: {torch.cuda.device_count()}")
26     for i in range(torch.cuda.device_count()):
27         print(f"GPU {i}: {torch.cuda.get_device_name(i)}")
28
29 # Define data transformations for preprocessing
30 data_transforms = {
31     'train': transforms.Compose([
32         transforms.Resize((img_height, img_width)), # Resize images to standard dimensions
33         transforms.RandomHorizontalFlip(), # Randomly flip images horizontally (augmentation)
34         transforms.RandomRotation(20), # Randomly rotate images up to 20 degrees (augmentation)
35         transforms.ToTensor(),
36         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Normalize using ImageNet statistics
37     ]),
38     'test': transforms.Compose([
39         transforms.Resize((img_height, img_width)), # Resize images to standard dimensions
40         transforms.ToTensor(), # Convert images to PyTorch tensors
41         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # Normalize using ImageNet statistics
42     ])
43 }
44
45 # Load datasets into PyTorch DataLoader objects
46 data_loaders = {
47     'train': DataLoader(datasets.ImageFolder(train_dir, transform=data_transforms['train']), batch_size=batch_size, shuffle=True),
48     'validation': DataLoader(datasets.ImageFolder(val_dir, transform=data_transforms['test']), batch_size=batch_size, shuffle=False),
49     'test': DataLoader(datasets.ImageFolder(test_dir, transform=data_transforms['test']), batch_size=batch_size, shuffle=False)
50 }
51
52 # Determine the number of classes in the dataset
53 num_classes = len(datasets.ImageFolder(train_dir).classes)
54
55 # Define pre-trained models
56 # Model 1: MobileNetV2
57 model1 = models.mobilenet_v2(pretrained=True) # Load pre-trained MobileNetV2
58 model1.classifier[1] = nn.Linear(model1.last_channel, num_classes) # Modify the classifier to match the dataset classes
59 model1 = model1.to(device) # Move the model to the selected device
60
61 # Model 2: ResNet18
62 model2 = models.resnet18(pretrained=True) # Load pre-trained ResNet18
63 model2.fc = nn.Linear(model2.fc.in_features, num_classes) # Modify the fully connected layer to match the dataset classes
64 model2 = model2.to(device) # Move the model to the selected device
65
66 # Define loss function and optimizers
67 criterion = nn.CrossEntropyLoss() # Loss function for multi-class classification
68 optimizer1 = optim.Adam(model1.parameters(), lr=0.001) # Optimizer for MobileNetV2
69 optimizer2 = optim.Adam(model2.parameters(), lr=0.001) # Optimizer for ResNet18
```

Fig. 8 Before Training Preparation Code

```
70
71 # Training function
72 2 usages  ▲ RuseCristian *
73 def train_model(model, optimizer, criterion, train_loader, val_loader, num_epochs=40, patience=5, model_name="model.pth"):
74     """
75     Parameters:
76     - model: PyTorch model to be trained
77     - optimizer: Optimizer for updating model weights
78     - criterion: Loss function
79     - train_loader: DataLoader for training data
80     - val_loader: DataLoader for validation data
81     - num_epochs: Maximum number of training epochs
82     - patience: Number of epochs to wait for improvement before stopping
83     - model_name: File name to save the best model
84
85     Returns:
86     - Training and validation metrics for each epoch
87     """
88     best_val_loss = float('inf') # Initialize the best validation loss
89     early_stop_counter = 0 # Counter for early stopping
90     train_acc, val_acc, train_losses, val_losses = [], [], [], [] # Lists to store metrics
91     best_epoch = 0 # Epoch with the best validation loss
92
93     for epoch in range(num_epochs):
94         print(f"Epoch {epoch+1}/{num_epochs}")
95         model.train() # Set the model to training mode
96         running_loss, correct = 0.0, 0
97
98         # Training loop
99         for inputs, labels in tqdm(train_loader, desc="Training", leave=False):
100             inputs, labels = inputs.to(device), labels.to(device)
101             optimizer.zero_grad() # Reset gradients
102             outputs = model(inputs) # Forward pass
103             loss = criterion(outputs, labels) # Compute loss
104             loss.backward() # Backward pass
105             optimizer.step() # Update weights
106
107             running_loss += loss.item() * inputs.size(0)
108             correct += (outputs.argmax(1) == labels).sum().item()
109
110         train_loss = running_loss / len(train_loader.dataset)
111         train_acc.append(correct / len(train_loader.dataset))
112         train_losses.append(train_loss)
113
114         # Validation loop
115         model.eval() # Set the model to evaluation mode
116         running_loss, correct = 0.0, 0
117         with torch.no_grad():
118             for inputs, labels in tqdm(val_loader, desc="Validation", leave=False):
119                 inputs, labels = inputs.to(device), labels.to(device)
120                 outputs = model(inputs)
121                 loss = criterion(outputs, labels)
122
123                 running_loss += loss.item() * inputs.size(0)
124                 correct += (outputs.argmax(1) == labels).sum().item()
125
126         val_loss = running_loss / len(val_loader.dataset)
127         val_acc.append(correct / len(val_loader.dataset))
128         val_losses.append(val_loss)
```

Fig. 9 Training function code part 1



```
124     val_loss = running_loss / len(val_loader.dataset)
125     val_acc.append(correct / len(val_loader.dataset))
126     val_losses.append(val_loss)
127
128     print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_acc[-1]:.4f}")
129     print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc[-1]:.4f}")
130
131     # Save the best model based on validation loss
132     if val_loss < best_val_loss:
133         best_val_loss = val_loss
134         best_epoch = epoch + 1
135         torch.save(model.state_dict(), model_name)
136         print(f"Model saved as {model_name}")
137         early_stop_counter = 0
138     else:
139         early_stop_counter += 1
140
141     # Trigger early stopping
142     if early_stop_counter >= patience:
143         print("Early stopping triggered.")
144         break
145
146     return train_acc, val_acc, train_losses, val_losses, best_epoch
147
148 # Train the models
149 train_acc1, val_acc1, train_loss1, val_loss1, best_epoch1 = train_model(model1, optimizer1, criterion, data_loaders['train'], data_loaders['validation'], model_name="mobilenetv2.pth")
150 train_acc2, val_acc2, train_loss2, val_loss2, best_epoch2 = train_model(model2, optimizer2, criterion, data_loaders['train'], data_loaders['validation'], model_name="resnet18.pth")
```

Fig. 10 Training function code part 2

```
153 # MobileNetV2 Plots
154 plt.figure(figsize=(10, 6))
155 plt.plot(*args: train_acc1, label='Train Accuracy (MobileNetV2)')
156 plt.plot(*args: val_acc1, label='Validation Accuracy (MobileNetV2)')
157 plt.axvline(best_epoch1 - 1, color='red', linestyle='--', label='Early Stopping Trigger (MobileNetV2)')
158 plt.title('MobileNetV2 Accuracy')
159 plt.xlabel('Epochs')
160 plt.ylabel('Accuracy')
161 plt.legend()
162 plt.grid()
163 plt.savefig("mobilenetv2_accuracy.png")
164 plt.show()
165
166 plt.figure(figsize=(10, 6))
167 plt.plot(*args: train_loss1, label='Train Loss (MobileNetV2)')
168 plt.plot(*args: val_loss1, label='Validation Loss (MobileNetV2)')
169 plt.axvline(best_epoch1 - 1, color='red', linestyle='--', label='Early Stopping Trigger (MobileNetV2)')
170 plt.title('MobileNetV2 Loss')
171 plt.xlabel('Epochs')
172 plt.ylabel('Loss')
173 plt.legend()
174 plt.grid()
175 plt.savefig("mobilenetv2_loss.png")
176 plt.show()
177
178 # ResNet18 Plots
179 plt.figure(figsize=(10, 6))
180 plt.plot(*args: train_acc2, label='Train Accuracy (ResNet18)')
181 plt.plot(*args: val_acc2, label='Validation Accuracy (ResNet18)')
182 plt.axvline(best_epoch2 - 1, color='red', linestyle='--', label='Early Stopping Trigger (ResNet18)')
183 plt.title('ResNet18 Accuracy')
184 plt.xlabel('Epochs')
185 plt.ylabel('Accuracy')
186 plt.legend()
187 plt.grid()
188 plt.savefig("resnet18_accuracy.png")
189 plt.show()
190
191 plt.figure(figsize=(10, 6))
192 plt.plot(*args: train_loss2, label='Train Loss (ResNet18)')
193 plt.plot(*args: val_loss2, label='Validation Loss (ResNet18)')
194 plt.axvline(best_epoch2 - 1, color='red', linestyle='--', label='Early Stopping Trigger (ResNet18)')
195 plt.title('ResNet18 Loss')
196 plt.xlabel('Epochs')
197 plt.ylabel('Loss')
198 plt.legend()
199 plt.grid()
200 plt.savefig("resnet18_loss.png")
201 plt.show()
202
```

Fig. 11 Code for Making the Loss Accuracy Plots

```
202
203
204 def evaluate_model(model, test_loader, model_name):
205     model.load_state_dict(torch.load(model_name))
206     model.eval()
207     y_true, y_pred = [], []
208     with torch.no_grad():
209         for inputs, labels in tqdm(test_loader, desc="Testing", leave=False):
210             inputs, labels = inputs.to(device), labels.to(device)
211             outputs = model(inputs)
212             y_true.extend(labels.cpu().numpy())
213             y_pred.extend(outputs.argmax(1).cpu().numpy())
214
215     print(classification_report(y_true, y_pred, target_names=datasets.ImageFolder(test_dir).classes))
216
217
218 evaluate_model(model1, data_loaders['test'], model_name="mobilenetv2.pth")
219 evaluate_model(model2, data_loaders['test'], model_name="resnet18.pth")
220
```

Fig. 11 Code for generating the classification reports

## Results and Observations

### Tracking Metrics

Throughout the training process, I kept track of two key metrics: **accuracy** and **loss** for both training and validation datasets. These helped me evaluate how well the models were learning and generalizing. By monitoring these trends, I could spot overfitting and decided when to stop training using early stopping.

### MobileNetV2

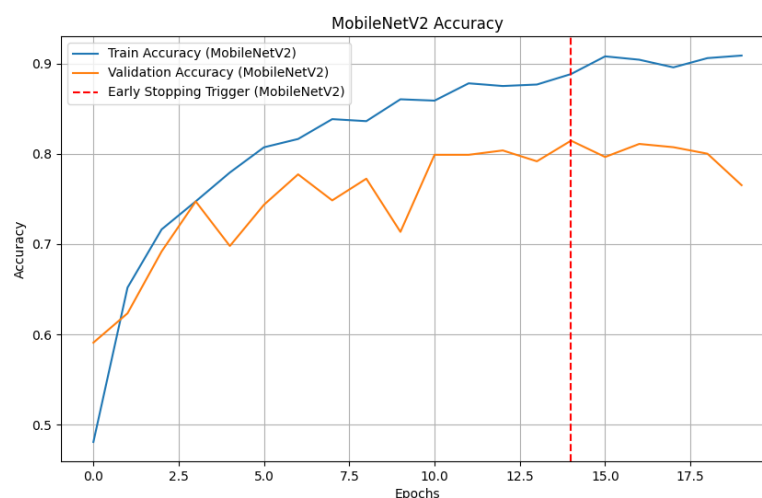


Fig. 12 MobileNetV2 Accuracy Graph

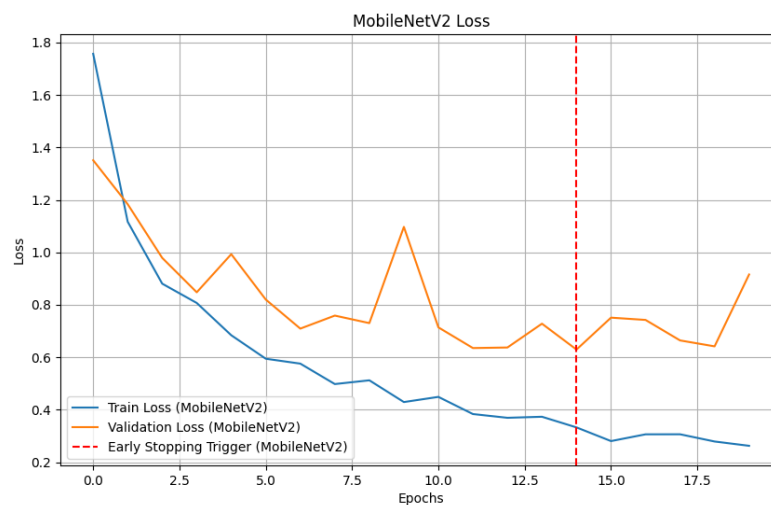


Fig. 13 MobileNetV2 Loss Graph

#### Accuracy:

1. Training accuracy increased steadily, eventually reaching **90%**. This shows the model's ability to fit well to the training data
2. Validation accuracy, on the other hand, plateaued at **80%**, with occasional fluctuations. The gap between training and validation accuracy hinted at overfitting, but the model still managed to improve overall performance until at epoch 14. It continued to run up until epoch 18 where the early stopping decided to stop training as there were 5 continuous epochs where the validation loss did not improve.

#### Loss:

1. Training loss consistently dropped below **0.4**, showing how well the model fit the training data.
2. However, validation loss oscillated around **0.8–1.0**, creating a widening gap between training and validation losses as epochs progressed. This gap is an early indicator of overfitting, as the model becomes increasingly focused on the training data while struggling to generalize to unseen data.
3. Despite the widening loss gap, the model's validation accuracy remained stable, indicating that MobileNetV2 was still improving before overfitting fully set in. Early stopping prevented further overfitting while preserving the model's performance at epoch 14.

## ResNet18

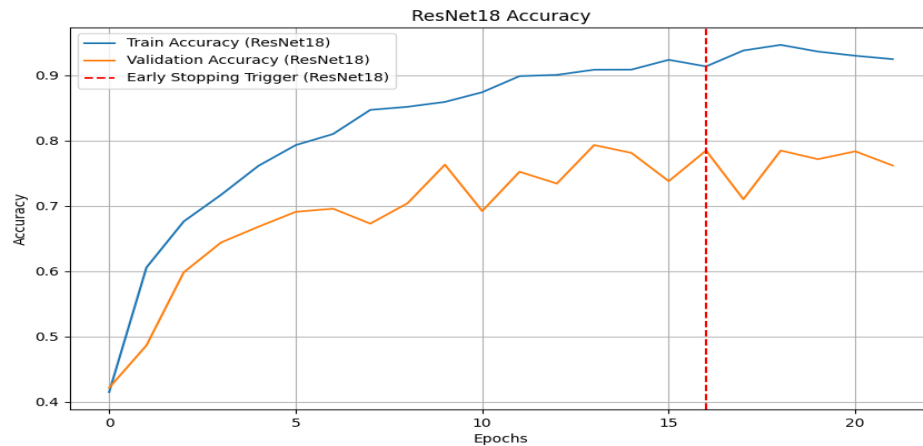


Fig. 14 ResNet18 Accuracy Graph

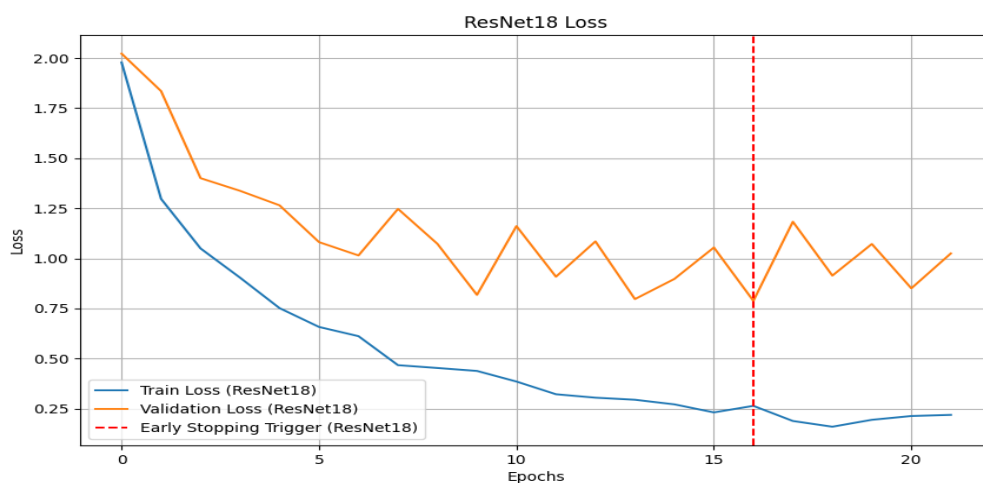


Fig. 15 ResNet18 Loss Graph

### Accuracy:

1. Training accuracy climbed higher than MobileNetV2, reaching **95%**, demonstrating the strength of ResNet18's deeper architecture and residual connections in capturing complex patterns.
2. Validation accuracy remained steady just slightly below **80%** but showed more variability compared to MobileNetV2. The fluctuations reflected the model's attempt to generalize, with occasional signs of overfitting as the gap between training and validation accuracy grew.

#### Loss:

1. Training loss dropped significantly to **0.25**, indicating the model's high confidence in its predictions on the training data.
2. Validation loss decreased but fluctuated between **0.7–1.0**, like MobileNetV2. The growing gap between training and validation losses highlighted overfitting tendencies. However, ResNet18 continued improving its validation accuracy until reaching its best results at epoch 16. Similarly, 5 epochs later, at epoch 21, the early stopping saw no improvement to the loss value compared to the best value, found at epoch 16, and decided to stop the training process.
3. The model's deeper architecture allowed it to perform better on the training data, but this also made it more prone to overfitting, as we can see the validation loss over the later epochs zig-zags up and downs and does not really improves, whilst the training loss improves consistently, thus it was a good idea to stop at epoch 16 due to definitely overfitting if continuing.

Both models showed clear signs of overfitting as training progressed, highlighted by the widening gap between training and validation losses. However, the early stopping mechanism ensured that the models retained their best validation performance.

- **MobileNetV2** demonstrated efficiency and robustness for a lightweight model, with validation accuracy stabilizing around **80%**.
- **ResNet18** achieved higher training accuracy, but its fluctuations in validation accuracy and loss revealed its sensitivity to overfitting due to its more complex architecture.

## Model Performance Results

The classification reports for the MobileNetV2 and ResNet18 models on the test dataset are presented below. These reports include precision, recall, F1-score, and support for each class.

#### MobileNetV2 Classification Report

Class	Precision	Recall	F1-Score	Support
Agaricus augustus	0.84	0.82	0.83	33

Class	Precision	Recall	F1-Score	Support
Amanita amerirubescens	0.76	0.69	0.72	32
Amanita augusta	0.90	0.85	0.88	33
Amanita brunnescens	0.62	0.90	0.73	29
Amanita calyptroderma	0.74	0.78	0.76	32
Amanita flavoconia	0.88	0.94	0.91	31
Amanita muscaria	0.81	0.93	0.87	28
Amanita persicina	0.79	0.44	0.56	25
Amanita phalloides	0.83	0.75	0.79	32
Amanita velosa	0.88	0.85	0.87	27
Armillaria mellea	0.72	0.74	0.73	31
Armillaria tabescens	0.80	0.83	0.81	29
Artomyces pyxidatus	0.97	1.00	0.98	31
Bolbitius titubans	0.82	0.87	0.84	31
Boletus pallidus	0.90	0.87	0.89	31
Boletus rex-veris	0.88	0.73	0.80	30
Cantharellus californicus	0.88	0.97	0.92	31
Cantharellus cinnabarinus	1.00	0.93	0.97	30
Cerioporus squamosus	0.87	0.90	0.88	51
Chlorophyllum brunneum	0.64	0.92	0.75	25
Chlorophyllum molybdites	0.67	0.55	0.60	22
Galerina marginata	0.95	0.82	0.88	22
Ganoderma applanatum	0.89	0.81	0.85	21
Ganoderma curtisii	0.75	0.60	0.67	20
Ganoderma oregonense	0.53	0.76	0.62	25

Ruse Cristian Andrei  
Group: 1242A

Class	Precision	Recall	F1-Score	Support
Ganoderma tsugae	0.82	0.52	0.64	27
Phaeolus schweinitzii	0.79	0.96	0.87	28
Phlebia tremellosa	0.95	0.76	0.84	25
Phyllotopsis nidulans	0.89	0.89	0.89	28
Pleurotus ostreatus	0.85	0.79	0.81	28
<b>Overall Accuracy</b>			<b>0.81</b>	868
<b>Macro Average</b>	0.82	0.81	0.81	
<b>Weighted Average</b>	0.82	0.81	0.81	

#### ResNet18 Classification Report

Class	Precision	Recall	F1-Score	Support
Agaricus augustus	0.81	0.64	0.71	33
Amanita amerirubescens	0.75	0.66	0.70	32
Amanita augusta	0.75	0.64	0.69	33
Amanita brunnescens	0.75	0.72	0.74	29
Amanita calyptroderma	0.71	0.69	0.70	32
Amanita flavoconia	0.86	0.81	0.83	31
Amanita muscaria	0.81	0.93	0.87	28
Amanita persicina	0.92	0.48	0.63	25
Amanita phalloides	0.58	0.81	0.68	32
Amanita velosa	0.77	0.85	0.81	27
Armillaria mellea	0.58	0.68	0.63	31
Armillaria tabescens	0.65	0.59	0.62	29
Artomyces pyxidatus	1.00	0.94	0.97	31

Ruse Cristian Andrei  
Group: 1242A

Class	Precision	Recall	F1-Score	Support
Bolbitius titubans	0.79	0.84	0.81	31
Boletus pallidus	0.68	0.97	0.80	31
Boletus rex-veris	0.89	0.80	0.84	30
Cantharellus californicus	1.00	0.71	0.83	31
Cantharellus cinnabarinus	0.83	1.00	0.91	30
Cerioporus squamosus	0.94	0.88	0.91	51
Chlorophyllum brunneum	0.85	0.68	0.76	25
Chlorophyllum molybdites	0.68	0.68	0.68	22
Galerina marginata	1.00	0.50	0.67	22
Ganoderma applanatum	0.80	0.95	0.87	21
Ganoderma curtisii	0.49	0.90	0.63	20
Ganoderma oregonense	0.71	0.60	0.65	25
Ganoderma tsugae	0.84	0.59	0.70	27
Phaeolus schweinitzii	0.92	0.86	0.89	28
Phlebia tremellosa	0.77	0.96	0.86	25
Phyllotopsis nidulans	0.90	0.96	0.93	28
Pleurotus ostreatus	0.63	0.79	0.70	28
<b>Overall Accuracy</b>			<b>0.77</b>	868
<b>Macro Average</b>	0.79	0.77	0.77	
<b>Weighted Average</b>	0.79	0.77	0.77	



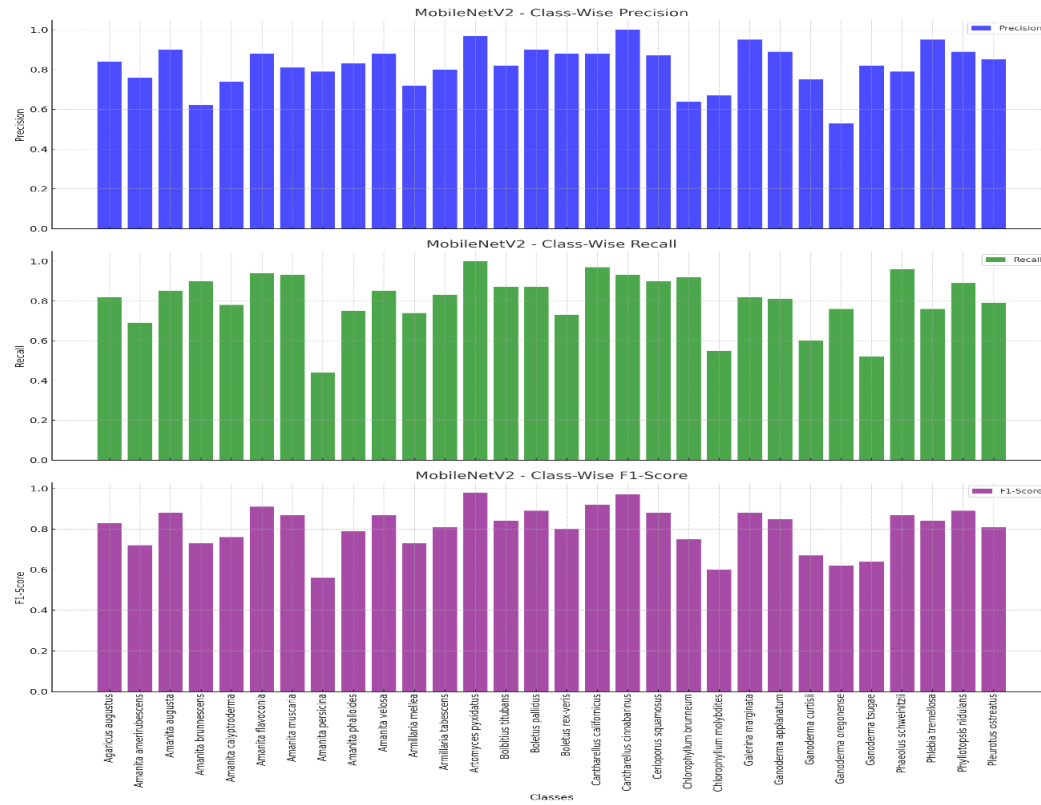


Fig. 16 Histogram for MobileNetv2

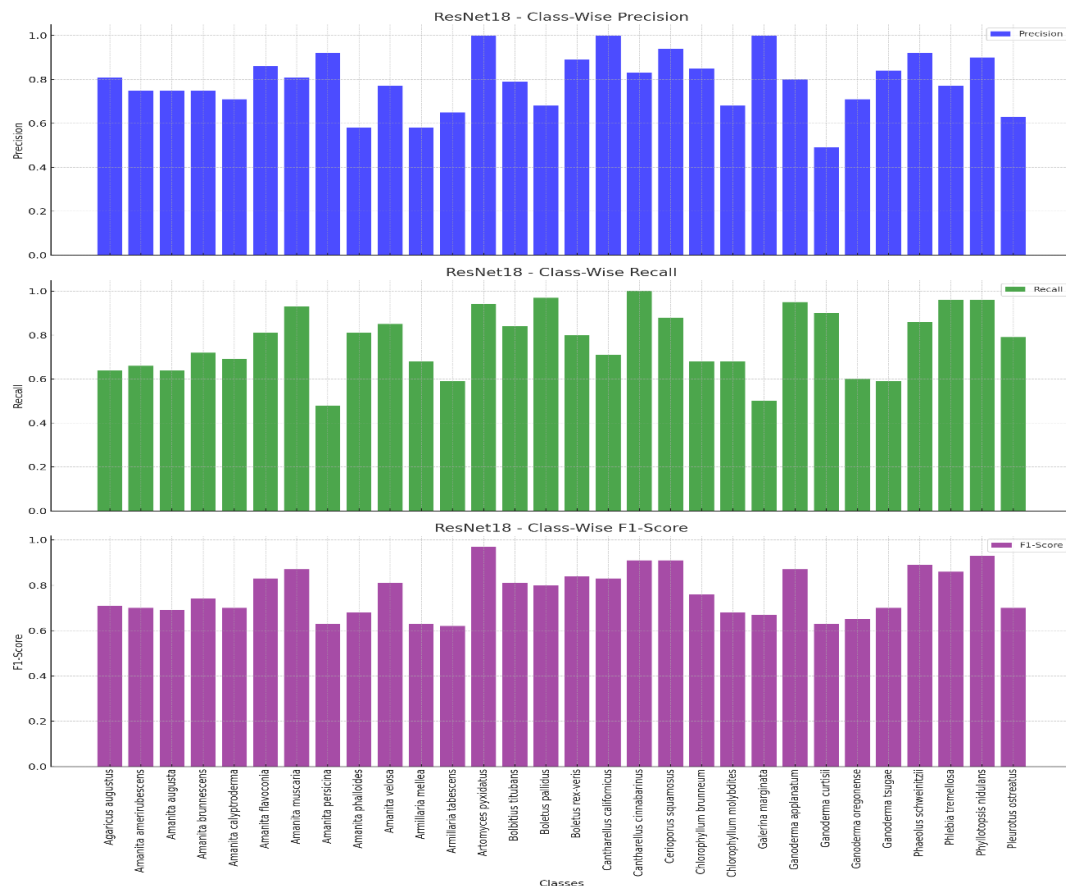


Fig. 17 Histogram for ResNet18

What each parameter represents:

Precision =  $\text{True Positives} / (\text{True Positives} + \text{False Positives})$

Recall =  $\text{True Positives} / (\text{True Positives} + \text{False Negatives})$

F1-Score =  $2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$

Support = Number of true instances for each class in the dataset.

Accuracy =  $\text{Correct Predictions} / \text{Total Predictions}$

## Observations

- **MobileNetV2** achieved an overall accuracy of **81%**, with notable performance in classes such as *Artomyces pyxidatus* (F1-Score: 0.98) and *Cantharellus cinnabarinus* (F1-Score: 0.97). However, it struggled with some classes like *Amanita persicina* (F1-Score: 0.56) due to lower recall.
- **ResNet18** attained an overall accuracy of **77%**, slightly lower than MobileNetV2. While its performance was comparable for certain classes, such as *Artomyces pyxidatus* (F1-Score: 0.97), it showed weaknesses in classes like *Galerina marginata* (F1-Score: 0.67) due to lower recall and precision.

Confusion Matrices

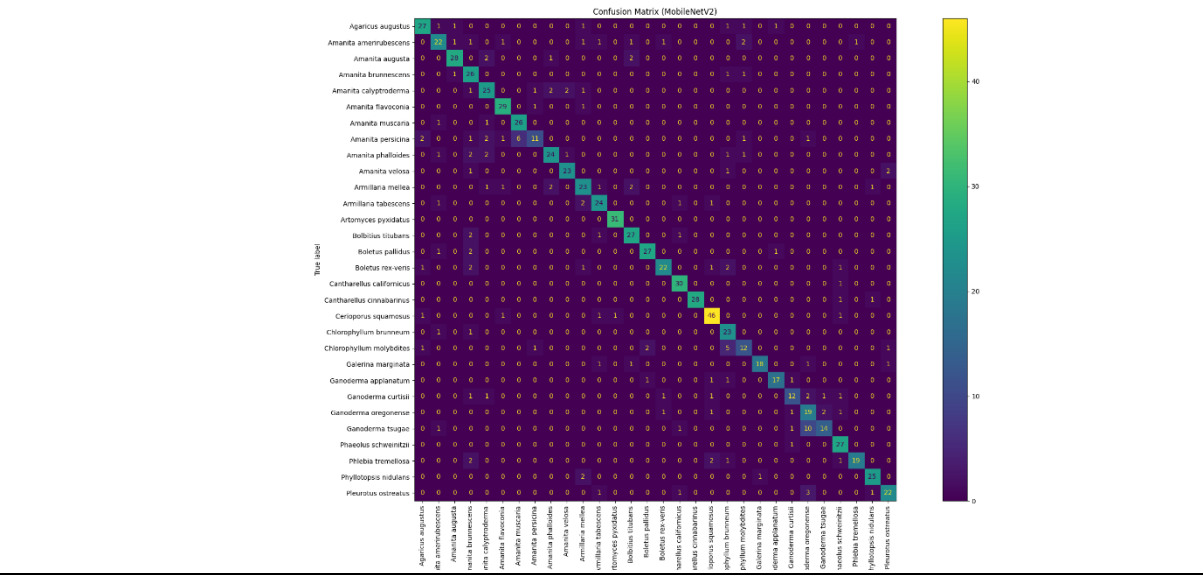


Fig. 18 MobileNetV2 Confusion Matrix

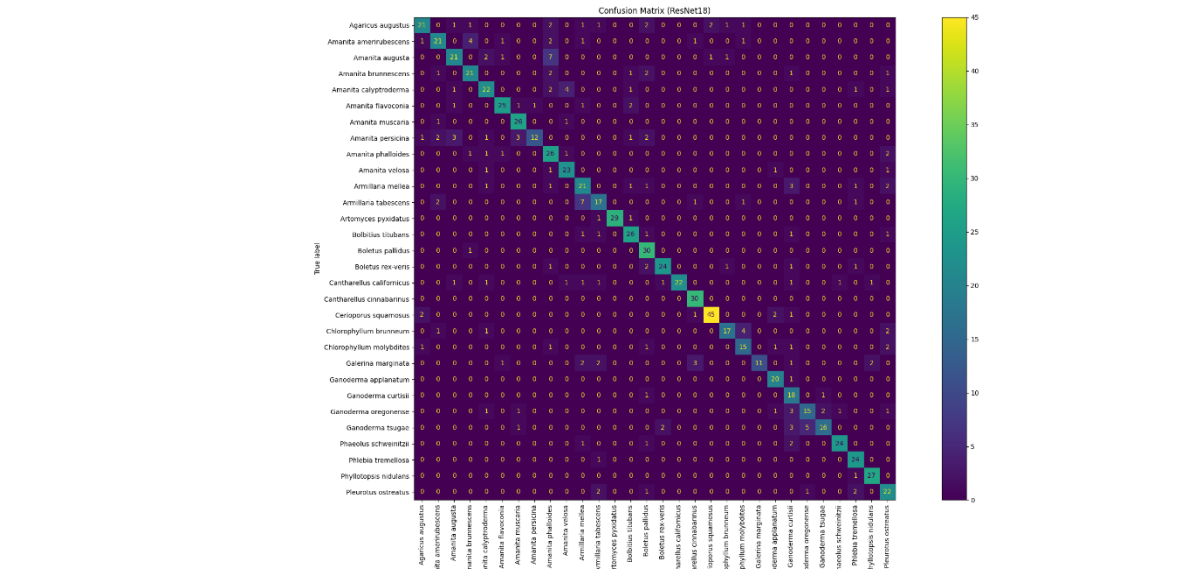


Fig. 19 ResNet18 Confusion Matrix

- **MobileNetV2:** This model shows strong overall performance, with many classes like *Artomyces pyxidatus* and *Pleurotus ostreatus* being classified almost perfectly. However, it struggles with visually similar classes, such as *Amanita flavoconia* and *Bolbitius titubans*.
- **ResNet18:** ResNet18 performs well on complex classes like *Cerioporus squamosus*, leveraging its deeper architecture. However, it also misclassifies certain challenging classes, showing that its complexity can lead to overfitting in some cases.

## Predictions

Both models were tested on 15 images from the test subset, therefore none of the images it was asked to predict were used in its training. This will show more of a real-world example of their performances.



Fig. 20 MobileNetV2 Predictions

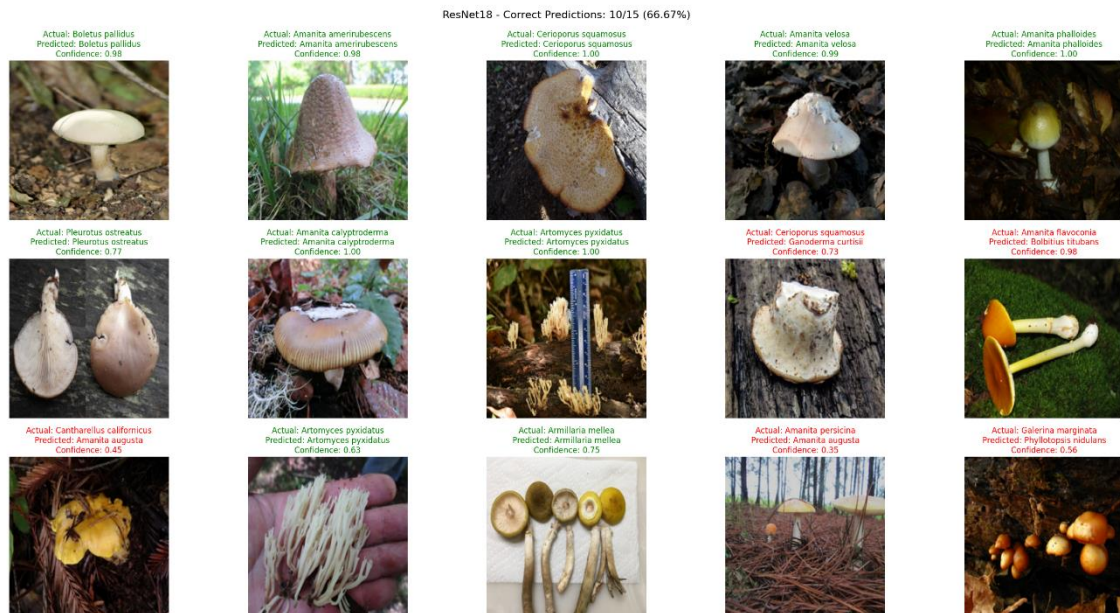


Fig. 21 ResNet18 Predictions

- **MobileNetV2:** Correctly predicted 12/15 images (80% accuracy). It handled simpler classes effectively but misclassified ambiguous images like *Amanita calyptroderma*. Misclassified predictions had noticeably lower confidence scores, which showcase that even though it misclassified 3 of the images, it was quite unsure of them to begin with.
- **ResNet18:** Correctly predicted 10/15 images (66.67% accuracy). While it performed well on complex features, it struggled with certain ambiguous classes, resulting in slightly more misclassifications than MobileNetV2, which is in accordance with the previous results as ResNet18 performed slightly worse in the accuracy metrics.

## Conclusion

The results highlight a distinction in the performance of MobileNetV2 and ResNet18. MobileNetV2 demonstrated stronger accuracy and reliability, particularly for simpler classifications, with fewer errors and lower confidence in its misclassifications. This suggests that MobileNetV2 might be a better choice for tasks requiring efficiency and consistent performance, especially when working with smaller datasets. While I did not test these models on larger datasets, the lightweight design of MobileNetV2 could make it more suitable for such scenarios. ResNet18, on the other hand, excelled at capturing complex features but showed slightly higher misclassification rates, indicating that it may be more appropriate for datasets with greater complexity or diversity, but that also would probably require bigger training datasets to make sure that the misclassifications do not occur so often.

## Bibliography

1. PyTorch Documentation. (2025). *PyTorch: An Open Source Machine Learning Framework*. Retrieved from <https://pytorch.org/docs/>
2. Scikit-learn Documentation. (2025). *Classification Metrics and Tools*. Retrieved from <https://scikit-learn.org/>
3. TQDM Documentation. (2025). *A Fast Progress Meter for Python*. Retrieved from <https://tqdm.github.io/>
4. Mushroom dataset <https://www.kaggle.com/datasets/iftekhara08/mo-106>
5. ResNet18 documentation <https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html>
6. MobileNetv2 documentation <https://pytorch.org/vision/main/models/mobilenetv2.html>