Høgskolen i Østfold

The Neural Net Project

Mushroom Species Classification Using Convolutional Neural Networks

Course Name: Machine Learning and Deep Learning

Group 4
Names:

Moritz Halkjelsvik

Jacob Holth

Magnus Holstad Johansson

Date: 27. 11. 2024

1. Introduction

Identifying mushroom species accurately is essential for applications ranging from ecological research to public safety. This project focuses on developing a neural network to classify mushroom species using a rich dataset of 3,122 images covering 215 distinct species, sourced from Kaggle. Each image is tagged with a species label creating a challenging yet well-defined multiclass classification task.

By exploring several convolutional neural network (CNN) architectures—ResNet, MobileNet, and EfficientNet. With data augmentation techniques the model learns to recognize subtle differences across species improving its robustness. This report outlines the dataset, preprocessing steps, and model performances, demonstrating the practical potential of CNNs for mushroom species identification.

1.1 Objective

This work aims to improve classification accuracy through the use of advanced convolutional neural network (CNN) architectures, such as ResNet, MobileNet, and EfficientNet. Additionally, by leveraging techniques like transfer learning, data augmentation, and optimization strategies, the project seeks to enhance model generalization and robustness.

The practical implications of this research extend to biodiversity monitoring, ecological research, and public safety by enabling reliable differentiation between edible and toxic mushroom species.

2. Literature Review

Mistaking one type of edible mushroom for another, or accidentally consuming toxic species leads to food-related illnesses and fatalities worldwide. As a result of this, it is crucial to accurately identify edible and inedible mushrooms (Wei et al., 2021). Given these risks, developing an accurate and reliable method for mushroom identification is critical. With the significant advancements in convolutional neural networks, for image classification tasks, these models have proven to be powerful tools for developing reliable and accurate methods for visual identification problems, using CNNs such as AlexNet, VGGNet and ResNet (Sultana et al., 2018). Given the need for accurate identification (Wei et al. 2021) and the proven effectiveness and accuracy of CNNs for image classification (Sultana et al. 2018), these models are highly suitable for tasks such as mushroom classification.

2.1 Existing Approaches for Mushroom Classification

Bashir et al.(2024) explored the use of CNNs for mushroom classification, utilizing a dataset of 103 species to demonstrate their applicability in species identification tasks. Similarly, Kiss and Czúni (2021) used a version of the MO106 dataset with 106 mushroom species and implemented EfficientNet, employing techniques like transfer learning and class-specific subnetworks to refine classification performance. These studies illustrate how CNNs can be effectively applied to mushroom classification problems.

2.2 Techniques for Improving Model Performance

Improving CNNs for image classification is all about data augmentation, transfer learning and optimisation. Data augmentation techniques like flipping, cropping and rotation increase dataset diversity and reduce overfitting and generalisation (Shorten & Khoshgoftaar, 2019). Transfer learning uses pre-trained models like ResNet and MobileNet to adapt general features from large datasets like ImageNet and it improves performance on small datasets (Cheung, 2017). Additionally optimisation techniques such as Adam adjusts learning rate dynamically and improves convergence and training stability (Kingma & Ba, 2015).

2.3 Research Gaps and Our Contribution

Mushroom classification with CNNs is limited and most research is on broader biological datasets. Data augmentation is under explored in this domain (Shorten & Khoshgoftaar, 2019). This project addresses these gaps by evaluating CNN architectures (ResNet, MobileNet, EfficientNet) and testing advanced augmentation techniques to improve accuracy and generalization.

3. Dataset and Preprocessing

Data preparation is crucial for training machine learning models, especially neural networks, which are highly sensitive to data quality. Preprocessing standardizes and cleans the data, ensuring it is ready for training while improving generalization by reducing dependence on raw data characteristics. This chapter covers the mushroom classification dataset, the training-validation-test split, and the preprocessing techniques used for convolutional neural networks.

3.1 Dataset

The dataset[1] includes images of various mushroom species. Each image is associated with a single label, identifying the mushroom's species, making this a multiclass classification problem. All code can be found on our GitHub repository[2].

3.2 Data Split

The entire dataset is loaded using the datasets folder, which reads images and their respective labels from the specified data direction (Appendix A.1). The dataset is then split into **training**, **validation**, and **test** sets with an 80/10/10 ratio:

- **Training Set (80%):** This is the largest portion of the data and is used to train the model.
- **Validation Set (10%):** This subset is used to tune the model and evaluate its performance during training without using the test set, helping to prevent overfitting.
- **Test Set (10%):** This final subset is reserved for evaluating the model's performance after training is complete.

[1] https://www.kaggle.com/datasets/daniilonishchenko/mushrooms-images-classification-215
[2] https://github.com/Zuparo1/ML_P2

The split is achieved by calculating the sizes of each subset and using a random split function to divide the dataset accordingly.

3.3 Data Preprocessing

Preprocessing involves applying specific transformations to the images to make the data suitable for training the neural network and to improve generalization. The propressing we did was based on what we thought would be successful, before experimenting. The following are the preprocessing we did on our data initially(Appendix A.1):

- **Resize**: Images are resized to 224x224 pixels, which is a common input size for CNN models like MobileNet and ResNet.
- **CenterCrop**: A center crop of 224x224 is taken to ensure that all images have a uniform size and remove any extraneous parts of the image that might not be relevant.
- **RandomHorizontalFlip**: This randomly flips images horizontally during training, creating variations that help the model generalize better by exposing it to different orientations.
- **ToTensor**: Converts the images into PyTorch tensors, which are needed for training with PyTorch models.
- **Normalize**: The images are normalized with mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225], aligning with the normalization used in the pre-trained models, which helps in making training faster and more stable.

3.4 Data Loaders

After the dataset is split, each subset (training, validation, and test) is loaded into DataLoaders (Appendix A.1):

- **Training DataLoader:** Shuffles the data to randomize the order of images in each batch, which helps prevent the model from learning unintended patterns.
- **Validation and Test DataLoaders:** Do not shuffle, allowing consistent evaluation of the model on these sets.

4. Models and training

Early in the project it was decided to try multiple different models to achieve the most accurate model possible.

4.1 Models

Since there are many CNN models available we decided to experiment with different ones to find the model that gained the best result with our data. All models were ran with the same data preparation

All models used were pretrained. By using a model that has already learned to recognize a wide range of patterns and features, we can leverage that knowledge as a starting point rather than training a model from scratch. This approach, known as transfer learning, saves time, resources and can lead to better results, especially when the target dataset is smaller or more specific than the original training dataset.

4.1.1 Resnet model

We chose to experiment with both ResNet18, and ResNet50. Pre-trained convolutional neural networks, chosen for its balance between efficiency and accuracy in feature extraction.

4.1.2 MobileNet Model

MobileNet models are designed to be lightweight and efficient, often containing fewer parameters compared to deeper models like the ResNet model we used before. Fewer parameters mean that each weight update has a proportionally larger impact. We also lowered the learning rate to allow the model to converge more stable without overshooting the optimal point in the parameter space. We even in some cases tried with the same learning rate as resnet.

4.1.3 EfficientNet Model

We incorporated EfficientNet to evaluate its performance against other models, ResNet and MobileNet. It is a convolutional neural network architecture known for achieving high accuracy with fewer parameters compared to traditional CNN models. This efficiency makes it particularly appealing for image classification tasks where both accuracy and computational resources are essential considerations.

4.2 Training

Training required addressing common issues that could impact the model. Several strategies were implemented to address local optima, error valleys, vanishing gradients and overfitting. Learning rate scheduling adjusted the learning rate during training to prevent the model from getting stuck in suboptimal regions and converge faster. Data augmentation techniques like random flipping and rotation added variety to the training data to reduce the fluctuations in validation loss and generalization.

The details of the code implementation are organized as follows: the training process for the model is detailed in Appendix A.2, the evaluation methods are outlined in Appendix A.3, and the specific parameters unique to each model are provided in Appendix A.4. These sections offer insights into the underlying configurations and steps used to develop and analyze the models.

5. Initial Results

5.1 Resnet

We tested two ResNet variants, their performances are summarized as follows:

- **ResNet18:** This model reached a validation accuracy of approximately **52%**. It provided a good balance between computational efficiency and accuracy but showed limited performance on the complex features in the dataset due to its shallower depth.

- **ResNet50**: With a deeper architecture, It achieved a validation accuracy of **50%**. It demonstrated a worse accuracy than ResNet18 and also came with increased computational costs, which made it slower in training.

Since a deeper architecture resulted in worse accuracy, we saw no point in experimenting with even deeper models of ResNet.
(See Appendix B.1 for detailed results on ResNet18 and ResNet50)

5.2 MobileNet

MobileNet Was tested with three variants:

- **MobileNetV2:** We decided to test with both 0,001 and 0,003 learning rate on this model. Using 0,003 achieved an accuracy of approximately **58%**, while learning rate of 0,003 achieved approximately **51%**. Since a lower learning rate proved better using this model, we used 0,001 on the other MobileNet models as well.
- **MobileNetV3 Large:** This model provided the best performance overall, reaching a validation accuracy of almost **59%.** MobileNetV3 Large's architecture allowed for efficient feature extraction while maintaining high accuracy, probably making it the most suitable model for this dataset.
- **MobileNetV3 Small:** This variant, tailored for ultra-low-compute environments, achieved a validation accuracy of **51%**. However, due to its reduced complexity it was the fastest model we tested.

(See Appendix B.2 for detailed results on the various MobileNet variants)

5.3 EfficentNet

EfficientNet was tested on EfficientNetB0 architecture. Its performance is summarized as follows:

- **EfficientNetB0**: This model achieved a validation accuracy of approximately **52%**, showing moderate effectiveness on the dataset. While its lightweight architecture provided computational efficiency, it struggled to capture the dataset's complex features. This suggests that the simplicity of the B0 variant may limit its capacity to differentiate between the subtle features of the mushroom species in the dataset.

(See Appendix B.3 for detailed results onEfficientNet)

5.4 Best model

All the different models showed a large improvement in validation accuracy across the first epochs, but the improvement flat lines at about 3 epochs and around 45-55% validation accuracy for all of them. Appendix E.1 consists of a detailed graph comparing the validation accuracy of the different models.
The comparison shows that **MobileNetV3_Large** was the most accurate model tested, with a validation accuracy of **58,77%** (Appendix B.2.3). It is also important to note its test accuracy of **54,3%**

6. Improving MobileNetV3_Large

6.1 Data augmentation

Data augmentation is a collection of techniques used to expand an existing dataset and can be implemented in many ways, such as linear or non-linear transformations, adding noise and applying affine transformations like translations, zoom, flip, shear, mirroring and color perturbations. These methods help overfitting and improve performance, especially on smaller datasets (Han et al., 2018).

Given the small size of our dataset, data augmentation was essential to enhance variation and improve model generalization. Transformations like random rotation and flips helped the model handle variations in orientation and environmental conditions. The use of data augmentation will here be discussed further:

- **Removing Cropping:** When we inspected the images of the dataset more closely we noticed that parts of the mushrooms might be cropped out by scaling the pictures as done in chapter xx. Therefore we removed rescaling.
- **More Augmentation:** To further improve the model's ability to generalize, we decided to try augmentation such as random flipping, rotation, gaussian blur, random perspective and color jitter.
- **Fewer Augmentations:** Since the previous augmentation did not work as expected, we decided to remove one type of augmentation, and train the model, until the results got better. Leaving only random rotation.
- **Augmentation Results:** Removing cropping improved accuracy moderately, while random flipping and rotation achieved the best results (67.6% validation, 69.6% test). Complex augmentations like blur and perspective had no effect, summarized in the table below. (See appendix C.2 for detailed results)

Augmentation Technique	**Validation Accuracy**	**Test Accuracy**	**Notes**
Removed Cropping	65.3%	61.3%	Observed significant improvement early on.
Random Flipping & Rotation	67.6%	69.6%	Best-performing technique overall.
Gaussian Blur, Perspective, etc.	No Improvement	No Improvement	Did not contribute to better generalization.

6.2 Optimizers

Optimizing the choice of optimizer is critical for improving model convergence and accuracy. Several optimizers were evaluated to determine their effectiveness for the

MobileNetV3_Large model. The results of these experiments are discussed below, with detailed configurations and outcomes provided in Appendix C.3.

- **ADAM**: The Adam optimizer, known for its adaptive learning rate capabilities, was used as the baseline for comparison.
- **LION**: A relatively new optimization method designed for efficient training, was tested as a potential improvement over Adam.

Optimizer	Validation Accuracy	Test Accuracy	Notes
Adam	67.6%	Consistent	Best performance; stable and generalizable.
LION	66.7%	50.1%	Potential overfitting observed and poor generalization.

Adam was the most effective optimizer, achieving stable performance and good generalization (67.6% validation). LION showed slightly lower validation accuracy (66.7%) but significantly underperformed on the test set (50.1%).
(See Appendix C.3 for detailed results)

6.3 Learning Rate

The experiments in this section are aimed to optimize learning rate strategies to improve the performance of the MobileNetV3_Large model. The Learning rate scheduling approaches we tested were ReduceLROnPlateau and StepLR. The results from Appendix C.4 are summarized below.

6.3.1 Schedulers

- **ReduceLROnPlateau:** Tested with various configurations of patience, threshold, factor, and minimum learning rate. Initial experiments showed no significant improvement.
- **StepLR:** Evaluated by varying step size and gamma

ReduceLROnPlateau with Patience=15 and 100 epochs provided the best results, achieving a validation accuracy of 69.2% and test accuracy of 68.1%. Other configurations, including variations in patience and factors, showed no improvement. StepLR proved ineffective across all tested configurations, offering no noticeable performance gains. The results are summarized in the table below, see Appendix C.4 for detailed results.

Scheduler	Configuration	Validation Accuracy	Test Accuracy	Notes
ReduceLROnPlateau	Patience=3	No Improvement	No Improvement	No noticeable gains.
ReduceLROnPlateau	Patience=15, 100 epochs	69.2%	68.1%	Best result, but required 100 epochs.
ReduceLROnPlateau	Patience=1, Factor=0.9, 100 epochs	No Improvement	No Improvement	Stagnated without significant gains.
StepLR	Various Step Size and Gamma	No Improvement	No Improvement	Ineffective for all configurations.
StepLR	Step Size=10	No Improvement	No Improvement	Ran out of epochs quickly.
StepLR	Step Size=25, 100 epochs	No Improvement	No Improvement	No gains observed.

7. Results

When experimenting with optimizers Adam provided the best balance between validation accuracy and generalization, making it the most effective optimizer for this task.

To better analyze the results we decided to calculate (Appendix D.1) the confidence level of the best results (Appendix D).

Initial comparisons between the chosen models resulted in MobileNetV3_Large being the most accurate one, with a validation accuracy of **58%**. the mean test accuracy was calculated to **54,3%**, and a 95% Confidence Interval of [49.3%, 58.9%] (Appendix D.2).

Experimenting with data augmentation, removing initial cropping and adding random rotation, resulted in an increase of validation accuracy of **67,6%**. We calculated the confidence level of this model and achieved a mean test accuracy of **69,6%** and a 95% confidence interval of [63.7%, 73.2%] (Appendix D.3)

While the experiments with learning rate schedulers improved validation accuracy from approximately **67,6%** to **69,2%**, the computational cost was significant. For instance, training with 100 epochs and the best configuration (ReduceLROnPlateau with patience=15) required approximately 1 hour. We calculated the confidence level of this model and achieved a mean test accuracy of **67,2%** and a 95% confidence interval of [63.7%, 73.2%] (Appendix D.4)
The table below highlights the improvement in performance across experiments.

Experiment	Validation Acurracy	Mean Test Accuracy	95% Confidence Interval	Notes
Baseline (MobileNetV3_Large)	58%	54.3%	[49.3%, 58.9%]	Initial comparisons showed this model was most accurate
With Data Augmentation	67.6%	69.6%	[63.7%, 73.2%]	Removing cropping and adding random rotation.
Learning Rate Schedulers	69.2%	67.2 %	[62,8%, 71,7%]	Minor improvement; possibly due to increased epochs.

To summarize, Data augmentation yielded the most significant boost, increasing validation accuracy from 58% to 67.6% combined test accuracy to 69.6% with a tighter confidence interval than the baseline.

8. Discussion

8.1 Key Results

Tuning and optimization (simplifying augmentations and learning rate scheduling) got MobileNet to 69.6% test accuracy with 95% confidence interval [62,8%, 71,7%]. This shows the model can be robust with targeted tuning. Simplified augmentation techniques like random flipping and rotation instead of cropping, was particularly effective in enhancing the model's generalization capabilities.

Learning rate scheduling also helped with training stability, ReduceLROnPlateau got validation accuracy to 69.2%. As shown in Appendix E.2, these enhancements had a cumulative effect, with augmentation providing an early boost in validation accuracy and learning rate scheduling driving steady improvements over time. But test accuracy didn't improve much, which could suggest overfitting to the validation set.

The reasons behind MobileNetV3_Large's superior performance are clear. MobileNetV3_Large performs better because of its efficient architecture that balances computation with feature extraction. Unlike ResNet50 which is deeper, MobileNet's lightweight design can generalize well on the small mushroom dataset without much computation. MobileNet's simplicity also makes it less prone to vanishing gradients which can hinder training of deeper networks.

8.2 Addressing Challenges

Augmentation helped boost MobileNet performance. Simplified augmentations and reduced generalization, the model learned to handle the variations in the data without relying on the patterns in the training set. So combining a lightweight architecture like MobileNet with carefully crafted augmentations can give good performance even on small datasets.

Several strategies were implemented to overcome key training challenges. Overfitting, local optima, and efficient gradient flow were addressed with several strategies. Simplified augmentations like flipping and rotation reduced overfitting by adding more data without added complexity. Learning rate scheduling especially with ReduceLROnPlateau helped avoid local optima and convergence. MobileNet's lightweight architecture also reduced the risk of vanishing gradients, ensuring effective learning even on a smaller dataset. All these helped improve the model's validation and test performance especially for MobileNetV3_Large.

8.3 Comparison with Related Work

Although we were unable to find any academic sources that used the same dataset as the one used in this paper, we did find some community contributions on Kaggle that used the same dataset, and faced the same difficulties, by having such a small amount of images, spread out on a large amount of species(classes). In the Kaggle notebook post "MobileNet : Mushroom Species Classification"(Dutta, 2023), they used MobileNet and achieved an accuracy of 51%.

In comparison, our implementation of MobileNetV3_Large achieved a validation accuracy of 58% and a test accuracy of 69.6%, showcasing the effectiveness of our optimization strategies, including data augmentation and learning rate adjustments. However, a limitation of our work lies in the absence of advanced techniques such as test-time augmentation and progressive resizing, as these methods are computationally intensive and require additional resources.

Another Kaggle notebook, "83% Accuracy | Mushroom Classification_215_Species"(Goswami, 2023) achieved significantly higher accuracy by leveraging the fastai library and advanced techniques such as test-time augmentation (TTA) and progressive resizing. While we did not incorporate these methods, they demonstrate the potential for further improving performance by focusing on test-time robustness and optimized training workflows. The implication of our work highlights the feasibility of using MobileNetV3_Large for mushroom classification tasks, particularly when computational resources are limited, as it offers a balance between efficiency and accuracy. Future work could explore the integration of advanced techniques like TTA and progressive resizing to further improve accuracy and robustness, as well as the evaluation of additional CNN architectures or ensemble methods to enhance classification performance.

9. Conclusion

In conclusion, while the dataset proved to be more challenging than first anticipated, given its imbalanced size, the use of preprocessing and data augmentation proved to be an invaluable and important step to get the dataset to a usable state. With this dataset, numerous models were tested, and our findings shows that MobileNetV3_Large was the best suited model.

The project highlights how well advanced CNNs can perform on complex classification tasks, emphasizing the importance of picking the most suitable architecture and optimizing it for the most effective performance. Ultimately, this work establishes a solid foundation for using lightweight CNNs in real-world applications like ecological research and public safety.

References

Bashir, Rab Nawaz, Olfa Mzoughi, Nazish Riaz, Muhammed Mujahid, Muhammad Faheem, Muhammad Tausif, and Amjad Rehman Khan. “Mushroom Species Classification in Natural Habitats Using Convolutional Neural Networks (CNN).” *IEEE Access*, 2024, 1–1. https://doi.org/10.1109/ACCESS.2024.3502543.

Dutta, G. (2023). “MobileNet : Mushroom Species Classification.” Accessed November 28, 2024. https://kaggle.com/code/gauravduttakiit/mobilenet-mushroom-species-classification.

Cheung, B. (2017, 14. March). “Deep Transfer Learning on Small Dataset.” Accessed November 27, 2024. https://bennycheung.github.io/deep-transfer-learning-on-small-dataset.

Goswami, H. (2023). “83% Accuracy | Mushroom Classification_215_Species.” Accessed November 28, 2024. https://kaggle.com/code/himgos/83-accuracy-mushroom-classification-215-species.

Han, Dongmei, Qigang Liu, and Weiguo Fan. “A New Image Classification Method Using CNN Transfer Learning and Web Data Augmentation.” *Expert Systems with Applications* 95 (April 2018): 43–56. https://doi.org/10.1016/j.eswa.2017.11.028.

Kingma, Diederik P., and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” arXiv, January 30, 2017. https://doi.org/10.48550/arXiv.1412.6980.

Kiss, Norbert, and László Czùni. “Mushroom Image Classification with CNNs: A Case-Study of Different Learning Strategies.” In *2021 12th International Symposium on Image and Signal Processing and Analysis (ISPA)*, 165–70, 2021. https://doi.org/10.1109/ISPA52656.2021.9552053.

Shorten, Connor, and Taghi M. Khoshgoftaar. “A Survey on Image Data Augmentation for Deep Learning.” *Journal of Big Data* 6, no. 1 (July 6, 2019): 60. https://doi.org/10.1186/s40537-019-0197-0.

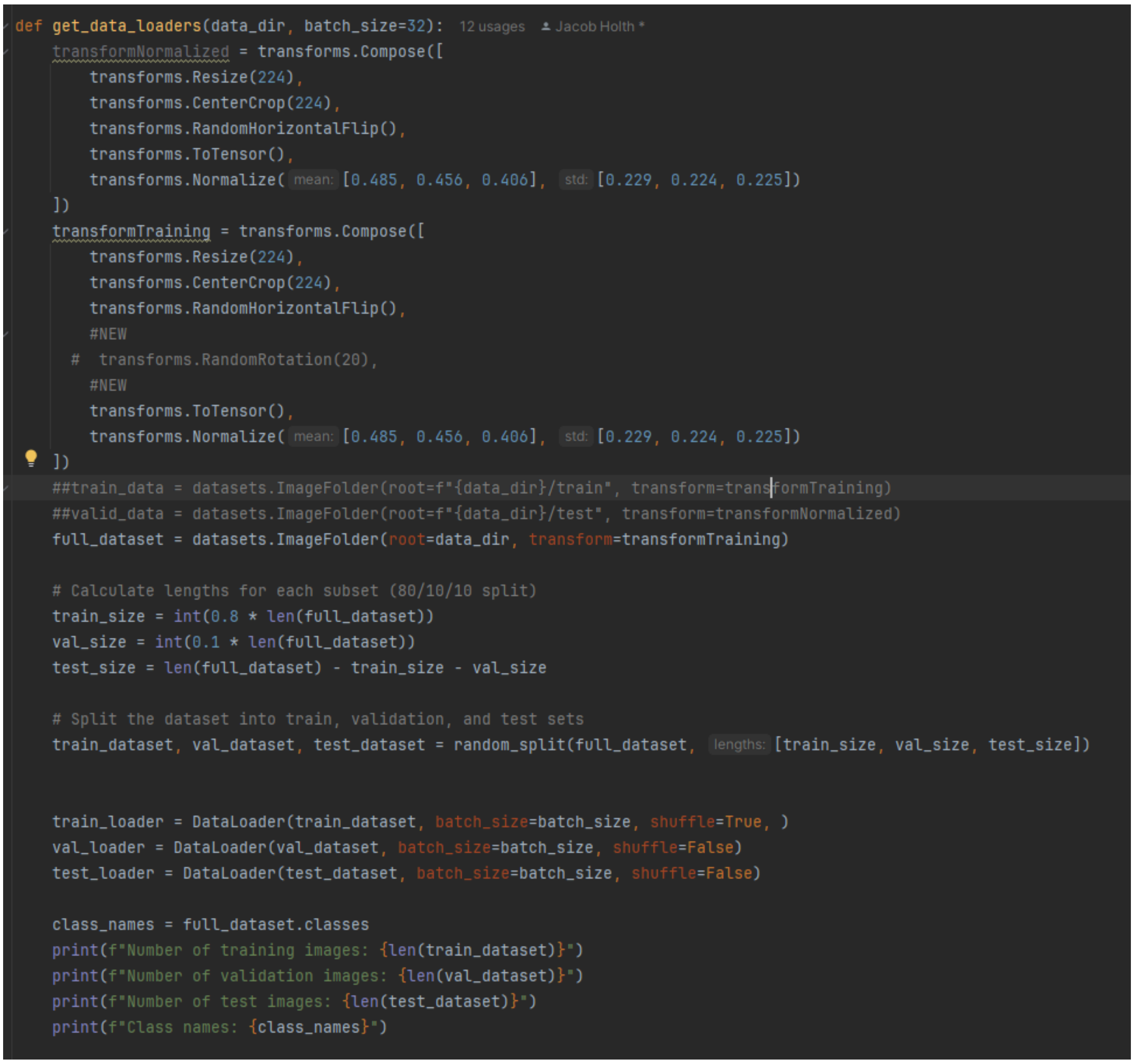
Sultana, Farhana, Abu Sufian, and Paramartha Dutta. “Advancements in Image Classification Using Convolutional Neural Network.” In *2018 Fourth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, 122–29, 2018. https://doi.org/10.1109/ICRCICN.2018.8718718.

Wei, Yuanmiao, Ling Li, Yao Liu, Shuna Xiang, Hanyue Zhang, Lunzhao Yi, Ying Shang, and Wentao Xu. “Identification Techniques and Detection Methods of Edible Fungi Species.” *Food Chemistry* 374 (April 2022): 131803. https://doi.org/10.1016/j.foodchem.2021.131803.

Appendix

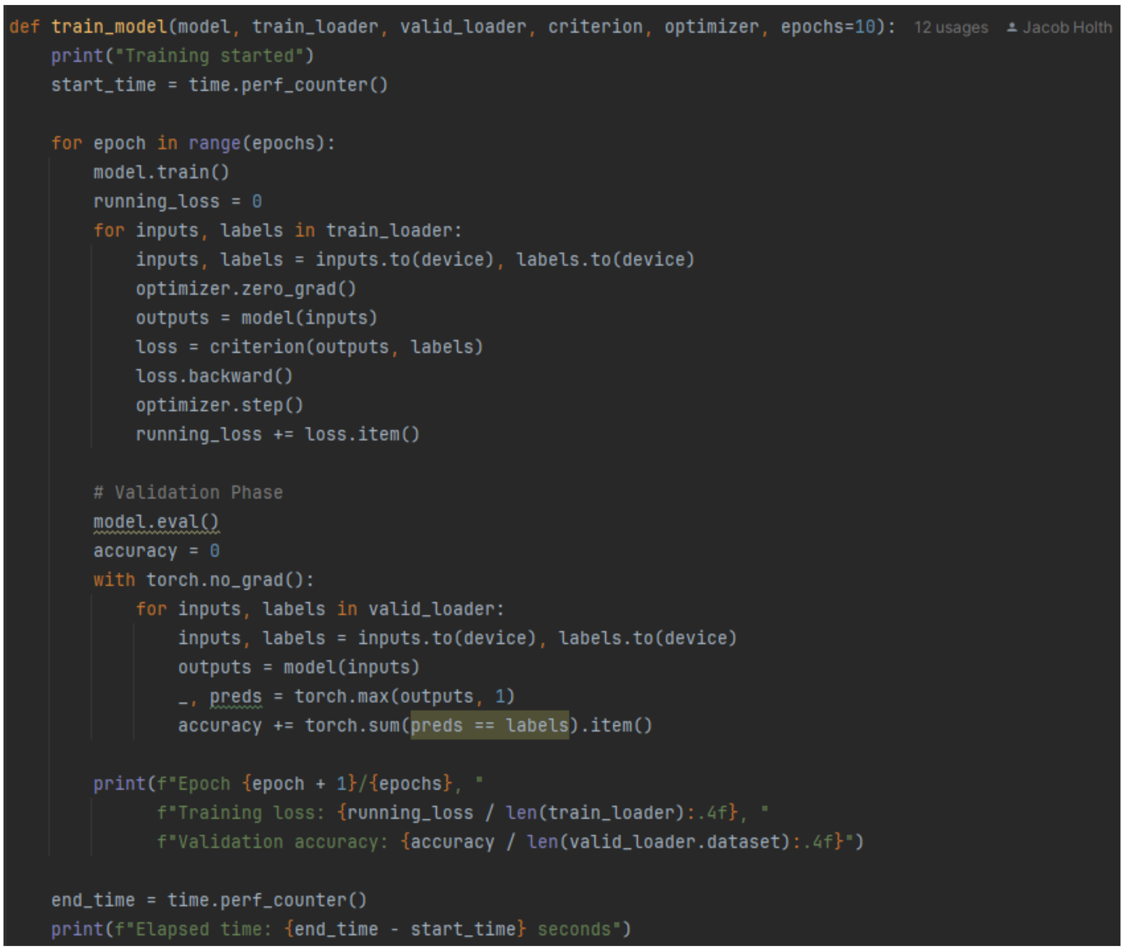
Appendix A - Code

A.1 - Data Loaders

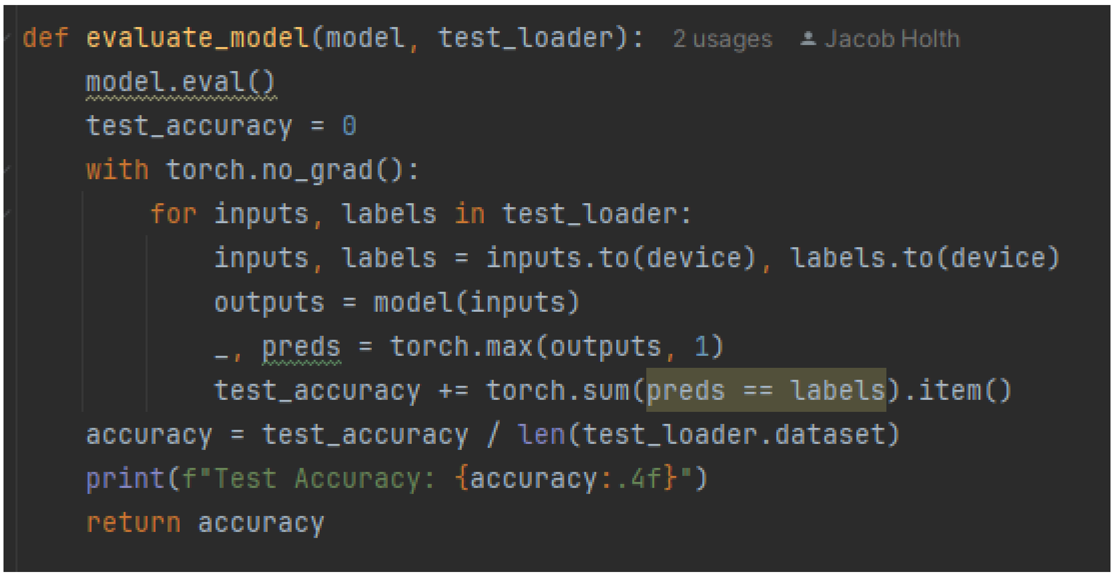


```
def get_data_loaders(data_dir, batch_size=32):  12 usages  Jacob Holth *
    transformNormalized = transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize( mean: [0.485, 0.456, 0.406],  std: [0.229, 0.224, 0.225])
    ])
    transformTraining = transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.RandomHorizontalFlip(),
        #NEW
      #  transforms.RandomRotation(20),
        #NEW
        transforms.ToTensor(),
        transforms.Normalize( mean: [0.485, 0.456, 0.406],  std: [0.229, 0.224, 0.225])
    ])
    ##train_data = datasets.ImageFolder(root=f"{data_dir}/train", transform=transformTraining)
    ##valid_data = datasets.ImageFolder(root=f"{data_dir}/test", transform=transformNormalized)
    full_dataset = datasets.ImageFolder(root=data_dir, transform=transformTraining)

    # Calculate lengths for each subset (80/10/10 split)
    train_size = int(0.8 * len(full_dataset))
    val_size = int(0.1 * len(full_dataset))
    test_size = len(full_dataset) - train_size - val_size

    # Split the dataset into train, validation, and test sets
    train_dataset, val_dataset, test_dataset = random_split(full_dataset,  lengths: [train_size, val_size, test_size])


    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, )
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    class_names = full_dataset.classes
    print(f"Number of training images: {len(train_dataset)}")
    print(f"Number of validation images: {len(val_dataset)}")
    print(f"Number of test images: {len(test_dataset)}")
    print(f"Class names: {class_names}")
```

A.2 - Train Model

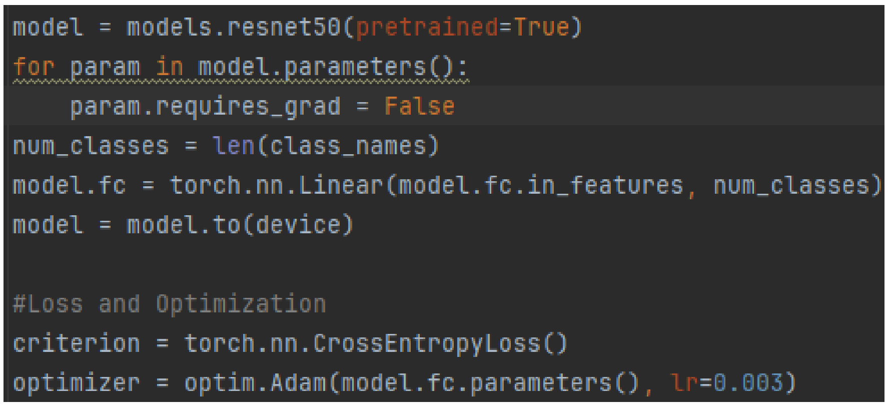

```python
def train_model(model, train_loader, valid_loader, criterion, optimizer, epochs=10):  12 usages  Jacob Holth
    print("Training started")
    start_time = time.perf_counter()

    for epoch in range(epochs):
        model.train()
        running_loss = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # Validation Phase
        model.eval()
        accuracy = 0
        with torch.no_grad():
            for inputs, labels in valid_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                accuracy += torch.sum(preds == labels).item()

        print(f"Epoch {epoch + 1}/{epochs}, "
              f"Training loss: {running_loss / len(train_loader):.4f}, "
              f"Validation accuracy: {accuracy / len(valid_loader.dataset):.4f}")

    end_time = time.perf_counter()
    print(f"Elapsed time: {end_time - start_time} seconds")
```

A.3 - Evaluate Model


```python
def evaluate_model(model, test_loader):  2 usages  Jacob Holth
    model.eval()
    test_accuracy = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            test_accuracy += torch.sum(preds == labels).item()
    accuracy = test_accuracy / len(test_loader.dataset)
    print(f"Test Accuracy: {accuracy:.4f}")
    return accuracy
```

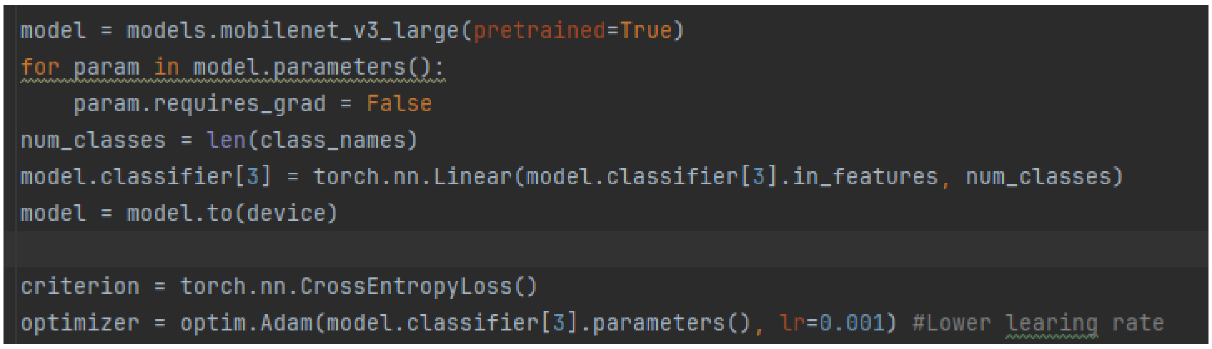
A.4 - Models

A.4.1 - Resnet



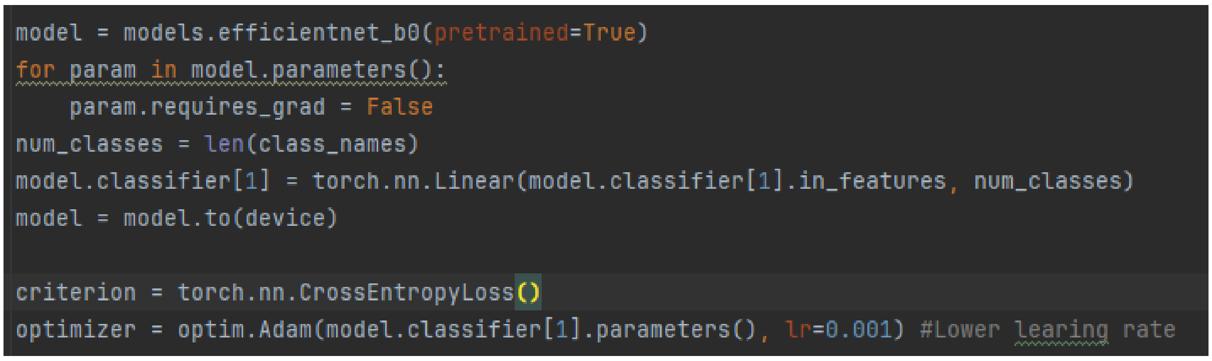
```
model = models.resnet50(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
num_classes = len(class_names)
model.fc = torch.nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)

#Loss and Optimization
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.003)
```

A.4.2 - MobileNet



```
model = models.mobilenet_v3_large(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
num_classes = len(class_names)
model.classifier[3] = torch.nn.Linear(model.classifier[3].in_features, num_classes)
model = model.to(device)

criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier[3].parameters(), lr=0.001) #Lower learing rate
```

A.4.3 - EfficentNet

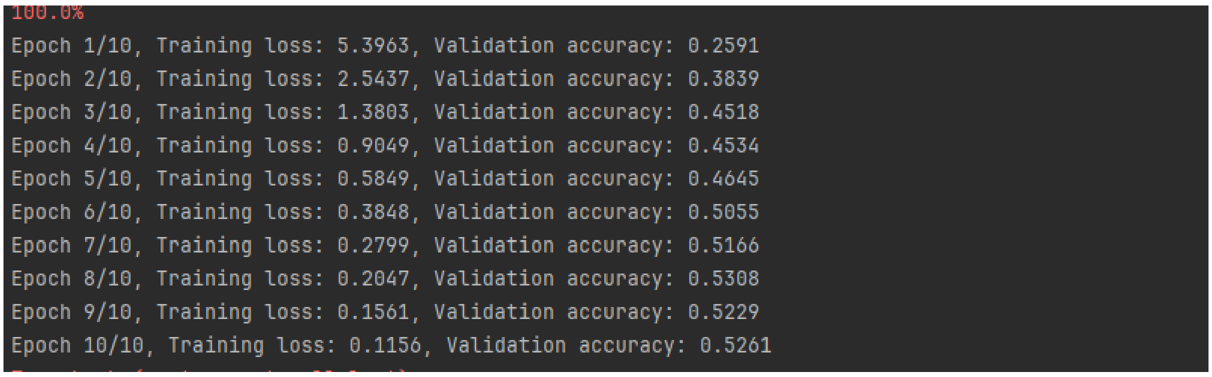


```
model = models.efficientnet_b0(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
num_classes = len(class_names)
model.classifier[1] = torch.nn.Linear(model.classifier[1].in_features, num_classes)
model = model.to(device)

criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier[1].parameters(), lr=0.001) #Lower learing rate
```

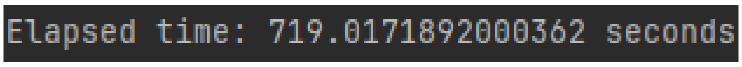
Appendix B - Results

B.1 - Resnet

B.1.1 Resnet 18

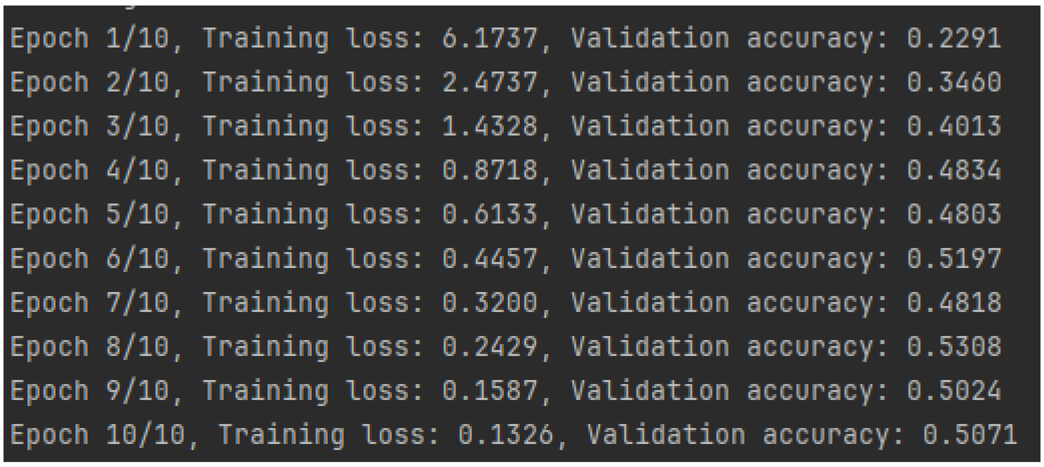


```
100.0%
Epoch 1/10, Training loss: 5.3963, Validation accuracy: 0.2591
Epoch 2/10, Training loss: 2.5437, Validation accuracy: 0.3839
Epoch 3/10, Training loss: 1.3803, Validation accuracy: 0.4518
Epoch 4/10, Training loss: 0.9049, Validation accuracy: 0.4534
Epoch 5/10, Training loss: 0.5849, Validation accuracy: 0.4645
Epoch 6/10, Training loss: 0.3848, Validation accuracy: 0.5055
Epoch 7/10, Training loss: 0.2799, Validation accuracy: 0.5166
Epoch 8/10, Training loss: 0.2047, Validation accuracy: 0.5308
Epoch 9/10, Training loss: 0.1561, Validation accuracy: 0.5229
Epoch 10/10, Training loss: 0.1156, Validation accuracy: 0.5261
```

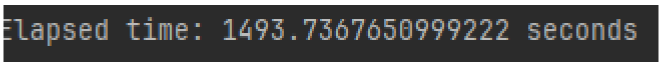


```
Elapsed time: 719.0171892000362 seconds
```

B.1.2 Resnet 50



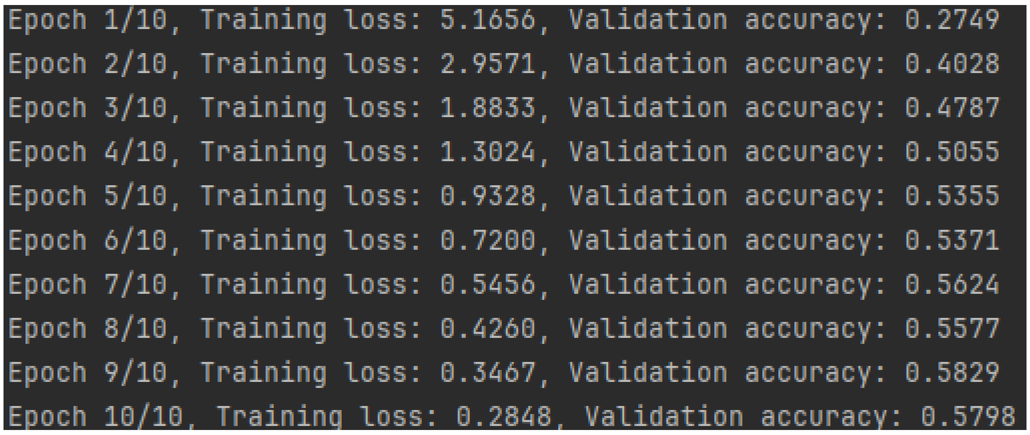
```
Epoch 1/10, Training loss: 6.1737, Validation accuracy: 0.2291
Epoch 2/10, Training loss: 2.4737, Validation accuracy: 0.3460
Epoch 3/10, Training loss: 1.4328, Validation accuracy: 0.4013
Epoch 4/10, Training loss: 0.8718, Validation accuracy: 0.4834
Epoch 5/10, Training loss: 0.6133, Validation accuracy: 0.4803
Epoch 6/10, Training loss: 0.4457, Validation accuracy: 0.5197
Epoch 7/10, Training loss: 0.3200, Validation accuracy: 0.4818
Epoch 8/10, Training loss: 0.2429, Validation accuracy: 0.5308
Epoch 9/10, Training loss: 0.1587, Validation accuracy: 0.5024
Epoch 10/10, Training loss: 0.1326, Validation accuracy: 0.5071
```



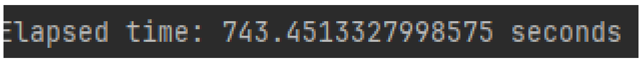
```
Elapsed time: 1493.7367650999222 seconds
```

B.2 - MobileNet

B.2.1 - MobilenetV2 (Learning rate 0.001)

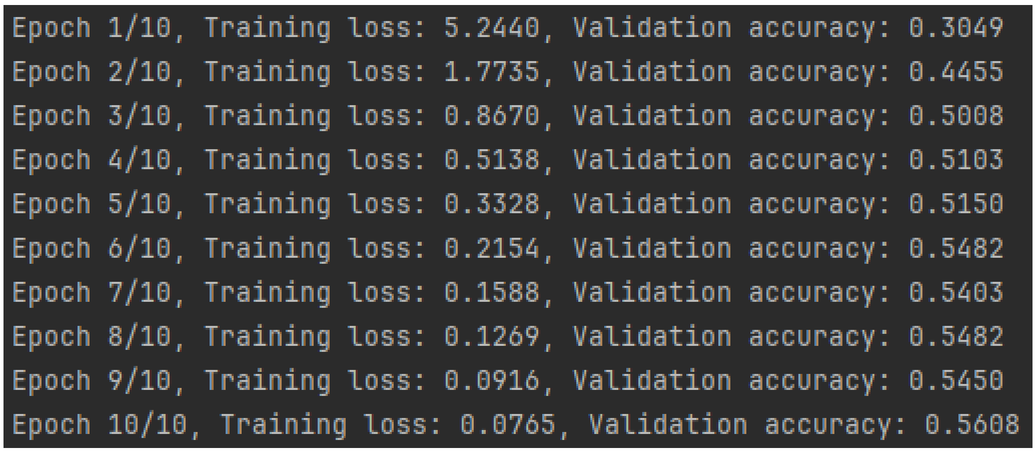


```
Epoch 1/10, Training loss: 5.1656, Validation accuracy: 0.2749
Epoch 2/10, Training loss: 2.9571, Validation accuracy: 0.4028
Epoch 3/10, Training loss: 1.8833, Validation accuracy: 0.4787
Epoch 4/10, Training loss: 1.3024, Validation accuracy: 0.5055
Epoch 5/10, Training loss: 0.9328, Validation accuracy: 0.5355
Epoch 6/10, Training loss: 0.7200, Validation accuracy: 0.5371
Epoch 7/10, Training loss: 0.5456, Validation accuracy: 0.5624
Epoch 8/10, Training loss: 0.4260, Validation accuracy: 0.5577
Epoch 9/10, Training loss: 0.3467, Validation accuracy: 0.5829
Epoch 10/10, Training loss: 0.2848, Validation accuracy: 0.5798
```

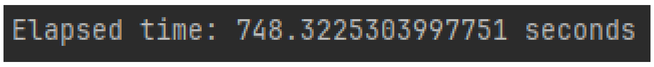


```
lapsed time: 743.4513327998575 seconds
```

B.2.2 - MobilenetV2 (Learning rate 0.003)

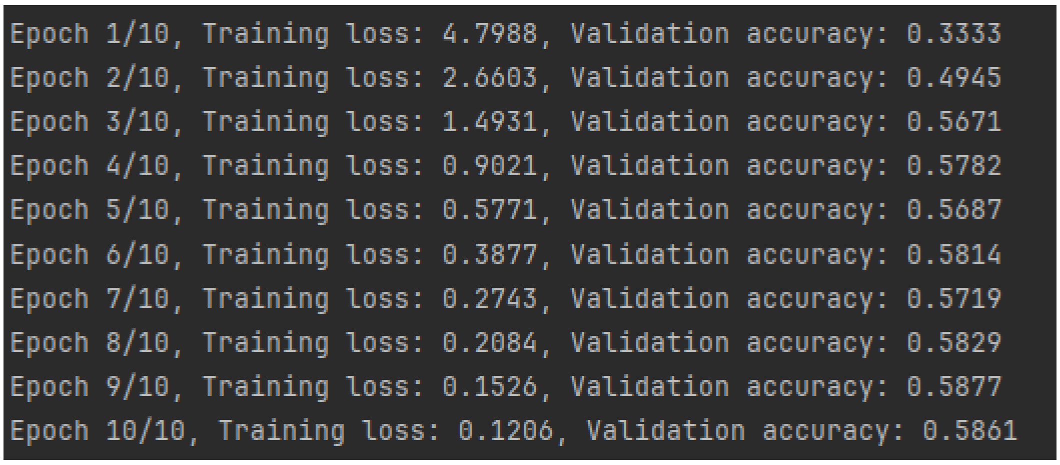


```
Epoch 1/10, Training loss: 5.2440, Validation accuracy: 0.3049
Epoch 2/10, Training loss: 1.7735, Validation accuracy: 0.4455
Epoch 3/10, Training loss: 0.8670, Validation accuracy: 0.5008
Epoch 4/10, Training loss: 0.5138, Validation accuracy: 0.5103
Epoch 5/10, Training loss: 0.3328, Validation accuracy: 0.5150
Epoch 6/10, Training loss: 0.2154, Validation accuracy: 0.5482
Epoch 7/10, Training loss: 0.1588, Validation accuracy: 0.5403
Epoch 8/10, Training loss: 0.1269, Validation accuracy: 0.5482
Epoch 9/10, Training loss: 0.0916, Validation accuracy: 0.5450
Epoch 10/10, Training loss: 0.0765, Validation accuracy: 0.5608
```

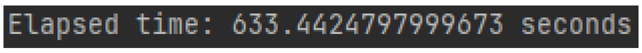


```
Elapsed time: 748.3225303997751 seconds
```

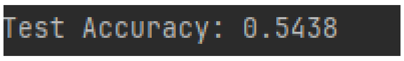
B.2.3 MobilenetV3_Large



```
Epoch 1/10, Training loss: 4.7988, Validation accuracy: 0.3333
Epoch 2/10, Training loss: 2.6603, Validation accuracy: 0.4945
Epoch 3/10, Training loss: 1.4931, Validation accuracy: 0.5671
Epoch 4/10, Training loss: 0.9021, Validation accuracy: 0.5782
Epoch 5/10, Training loss: 0.5771, Validation accuracy: 0.5687
Epoch 6/10, Training loss: 0.3877, Validation accuracy: 0.5814
Epoch 7/10, Training loss: 0.2743, Validation accuracy: 0.5719
Epoch 8/10, Training loss: 0.2084, Validation accuracy: 0.5829
Epoch 9/10, Training loss: 0.1526, Validation accuracy: 0.5877
Epoch 10/10, Training loss: 0.1206, Validation accuracy: 0.5861
```

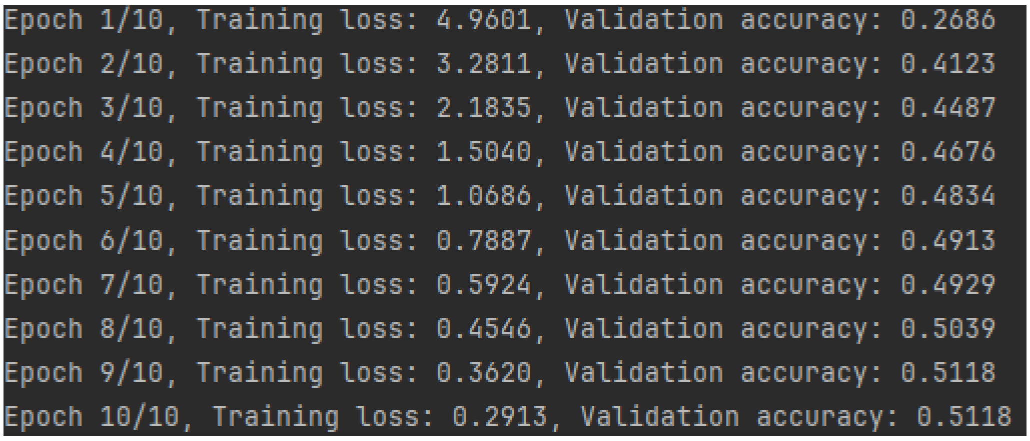


```
Elapsed time: 633.4424797999673 seconds
```

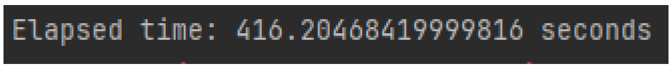


```
Test Accuracy: 0.5438
```

B.2.4 MobilenetV3_Small

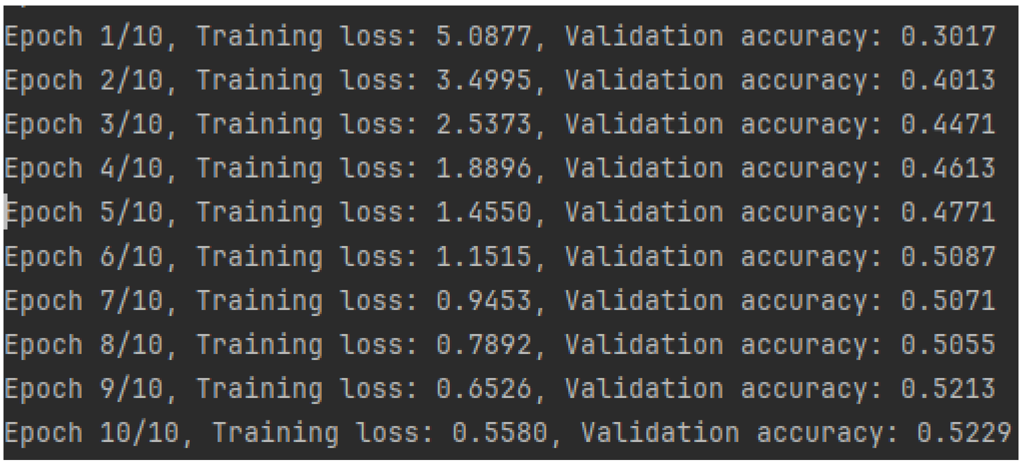


```
Epoch 1/10, Training loss: 4.9601, Validation accuracy: 0.2686
Epoch 2/10, Training loss: 3.2811, Validation accuracy: 0.4123
Epoch 3/10, Training loss: 2.1835, Validation accuracy: 0.4487
Epoch 4/10, Training loss: 1.5040, Validation accuracy: 0.4676
Epoch 5/10, Training loss: 1.0686, Validation accuracy: 0.4834
Epoch 6/10, Training loss: 0.7887, Validation accuracy: 0.4913
Epoch 7/10, Training loss: 0.5924, Validation accuracy: 0.4929
Epoch 8/10, Training loss: 0.4546, Validation accuracy: 0.5039
Epoch 9/10, Training loss: 0.3620, Validation accuracy: 0.5118
Epoch 10/10, Training loss: 0.2913, Validation accuracy: 0.5118
```

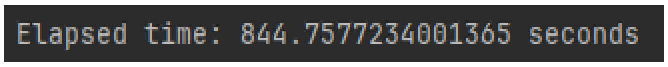


```
Elapsed time: 416.20468419999816 seconds
```

B.3 - Efficient net



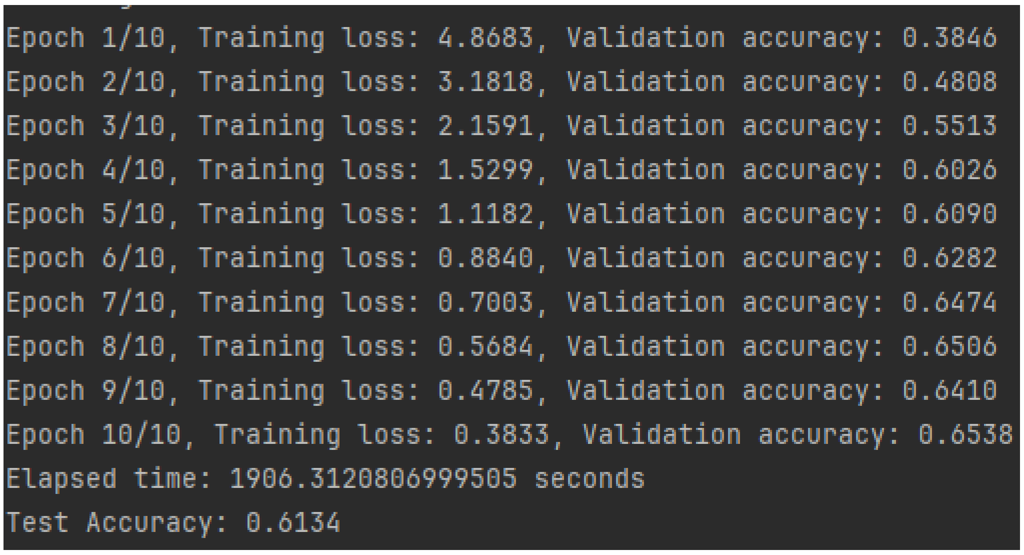
```
Epoch 1/10, Training loss: 5.0877, Validation accuracy: 0.3017
Epoch 2/10, Training loss: 3.4995, Validation accuracy: 0.4013
Epoch 3/10, Training loss: 2.5373, Validation accuracy: 0.4471
Epoch 4/10, Training loss: 1.8896, Validation accuracy: 0.4613
Epoch 5/10, Training loss: 1.4550, Validation accuracy: 0.4771
Epoch 6/10, Training loss: 1.1515, Validation accuracy: 0.5087
Epoch 7/10, Training loss: 0.9453, Validation accuracy: 0.5071
Epoch 8/10, Training loss: 0.7892, Validation accuracy: 0.5055
Epoch 9/10, Training loss: 0.6526, Validation accuracy: 0.5213
Epoch 10/10, Training loss: 0.5580, Validation accuracy: 0.5229
```



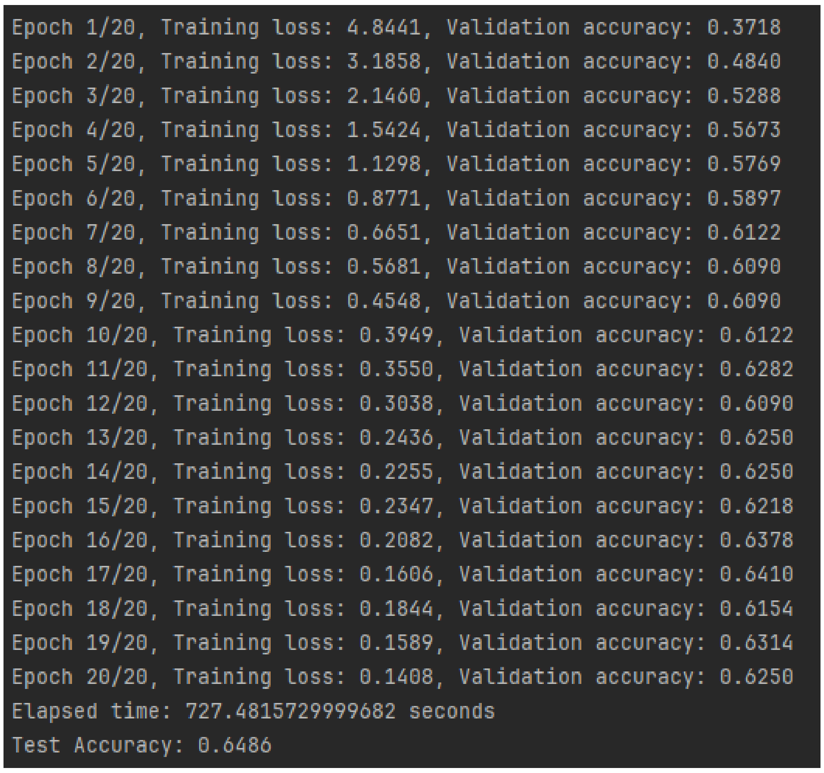
```
Elapsed time: 844.7577234001365 seconds
```

Appendix C - Improvement on MobileNet

C.1 - Remove Cropping



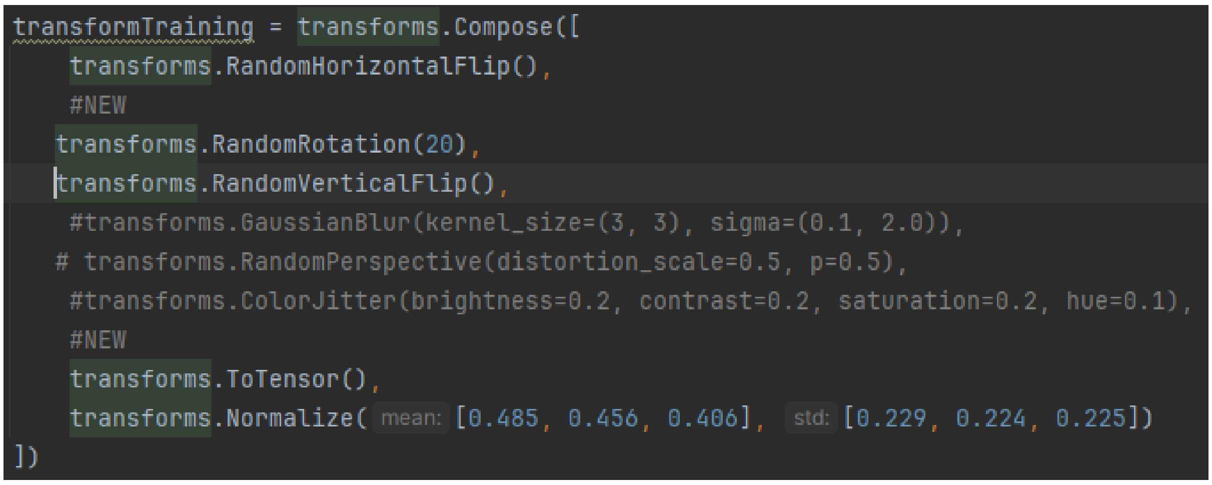
```
Epoch 1/10, Training loss: 4.8683, Validation accuracy: 0.3846
Epoch 2/10, Training loss: 3.1818, Validation accuracy: 0.4808
Epoch 3/10, Training loss: 2.1591, Validation accuracy: 0.5513
Epoch 4/10, Training loss: 1.5299, Validation accuracy: 0.6026
Epoch 5/10, Training loss: 1.1182, Validation accuracy: 0.6090
Epoch 6/10, Training loss: 0.8840, Validation accuracy: 0.6282
Epoch 7/10, Training loss: 0.7003, Validation accuracy: 0.6474
Epoch 8/10, Training loss: 0.5684, Validation accuracy: 0.6506
Epoch 9/10, Training loss: 0.4785, Validation accuracy: 0.6410
Epoch 10/10, Training loss: 0.3833, Validation accuracy: 0.6538
Elapsed time: 1906.3120806999505 seconds
Test Accuracy: 0.6134
```



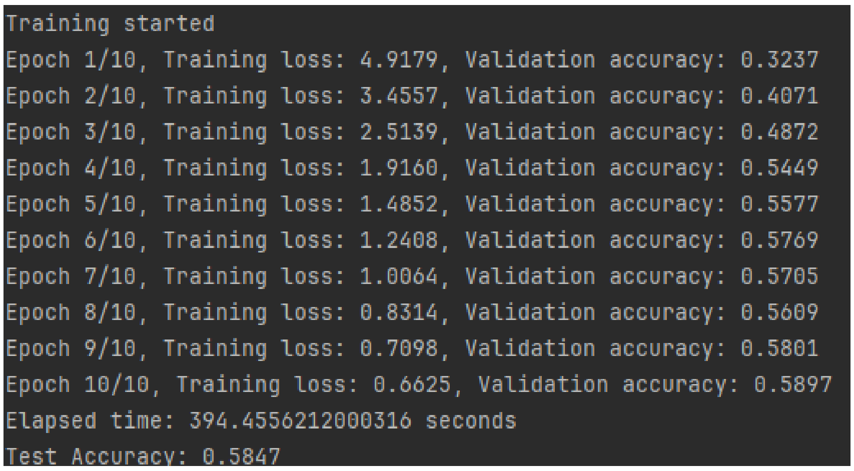
```
Epoch 1/20, Training loss: 4.8441, Validation accuracy: 0.3718
Epoch 2/20, Training loss: 3.1858, Validation accuracy: 0.4840
Epoch 3/20, Training loss: 2.1460, Validation accuracy: 0.5288
Epoch 4/20, Training loss: 1.5424, Validation accuracy: 0.5673
Epoch 5/20, Training loss: 1.1298, Validation accuracy: 0.5769
Epoch 6/20, Training loss: 0.8771, Validation accuracy: 0.5897
Epoch 7/20, Training loss: 0.6651, Validation accuracy: 0.6122
Epoch 8/20, Training loss: 0.5681, Validation accuracy: 0.6090
Epoch 9/20, Training loss: 0.4548, Validation accuracy: 0.6090
Epoch 10/20, Training loss: 0.3949, Validation accuracy: 0.6122
Epoch 11/20, Training loss: 0.3550, Validation accuracy: 0.6282
Epoch 12/20, Training loss: 0.3038, Validation accuracy: 0.6090
Epoch 13/20, Training loss: 0.2436, Validation accuracy: 0.6250
Epoch 14/20, Training loss: 0.2255, Validation accuracy: 0.6250
Epoch 15/20, Training loss: 0.2347, Validation accuracy: 0.6218
Epoch 16/20, Training loss: 0.2082, Validation accuracy: 0.6378
Epoch 17/20, Training loss: 0.1606, Validation accuracy: 0.6410
Epoch 18/20, Training loss: 0.1844, Validation accuracy: 0.6154
Epoch 19/20, Training loss: 0.1589, Validation accuracy: 0.6314
Epoch 20/20, Training loss: 0.1408, Validation accuracy: 0.6250
Elapsed time: 727.4815729999682 seconds
Test Accuracy: 0.6486
```

C.2 - Advanced Data Augmentation

C.2.1 - Rotation and Vertical Flip

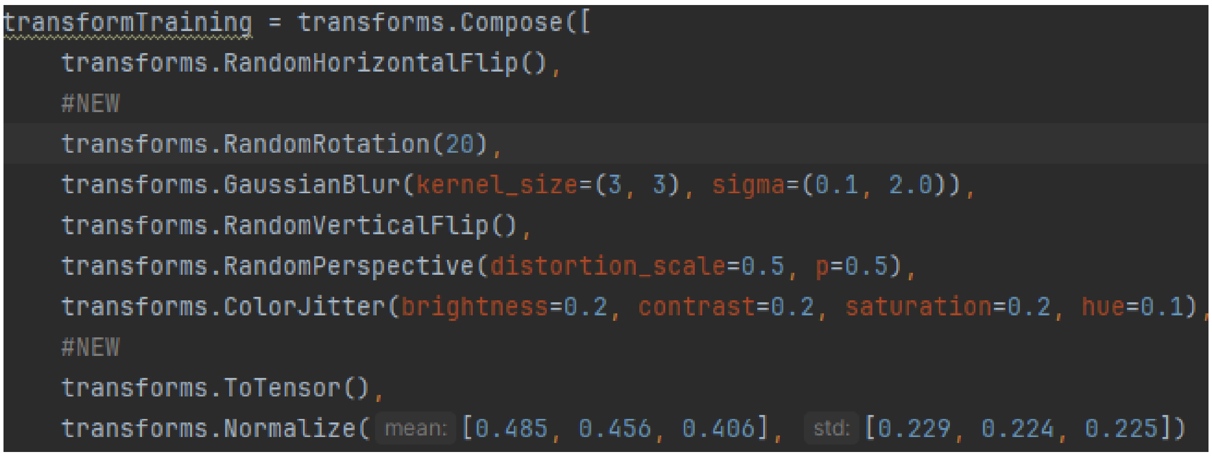


```
transformTraining = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    #NEW
   transforms.RandomRotation(20),
   transforms.RandomVerticalFlip(),
    #transforms.GaussianBlur(kernel_size=(3, 3), sigma=(0.1, 2.0)),
   # transforms.RandomPerspective(distortion_scale=0.5, p=0.5),
    #transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    #NEW
    transforms.ToTensor(),
    transforms.Normalize( mean: [0.485, 0.456, 0.406],  std: [0.229, 0.224, 0.225])
])
```

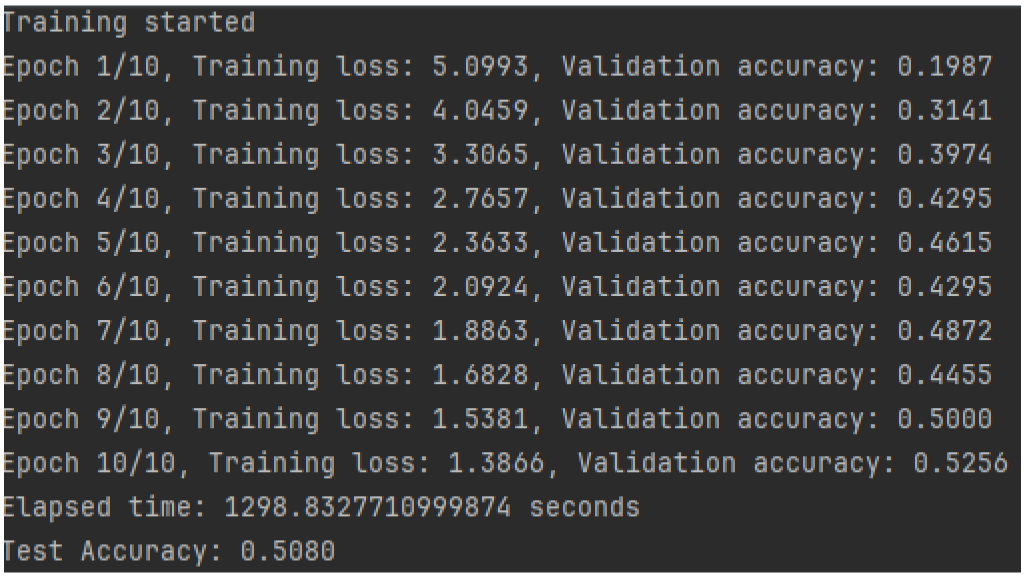


```
Training started
Epoch 1/10, Training loss: 4.9179, Validation accuracy: 0.3237
Epoch 2/10, Training loss: 3.4557, Validation accuracy: 0.4071
Epoch 3/10, Training loss: 2.5139, Validation accuracy: 0.4872
Epoch 4/10, Training loss: 1.9160, Validation accuracy: 0.5449
Epoch 5/10, Training loss: 1.4852, Validation accuracy: 0.5577
Epoch 6/10, Training loss: 1.2408, Validation accuracy: 0.5769
Epoch 7/10, Training loss: 1.0064, Validation accuracy: 0.5705
Epoch 8/10, Training loss: 0.8314, Validation accuracy: 0.5609
Epoch 9/10, Training loss: 0.7098, Validation accuracy: 0.5801
Epoch 10/10, Training loss: 0.6625, Validation accuracy: 0.5897
Elapsed time: 394.4556212000316 seconds
Test Accuracy: 0.5847
```

C.2.2 - Blur, Jitter, Perspective etc.

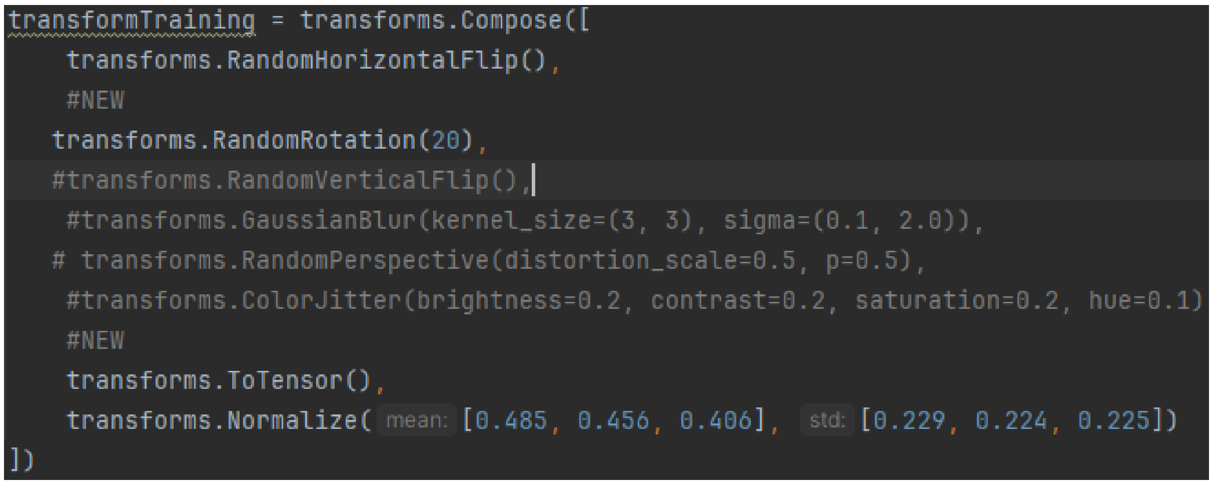


```
transformTraining = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    #NEW
    transforms.RandomRotation(20),
    transforms.GaussianBlur(kernel_size=(3, 3), sigma=(0.1, 2.0)),
    transforms.RandomVerticalFlip(),
    transforms.RandomPerspective(distortion_scale=0.5, p=0.5),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    #NEW
    transforms.ToTensor(),
    transforms.Normalize( mean: [0.485, 0.456, 0.406],  std: [0.229, 0.224, 0.225])
```

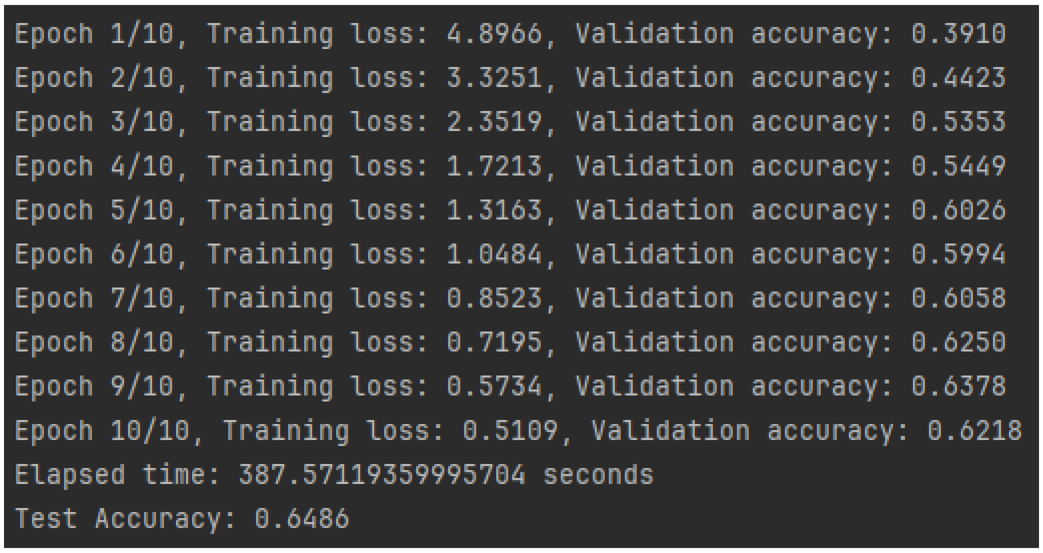


```
Training started
Epoch 1/10, Training loss: 5.0993, Validation accuracy: 0.1987
Epoch 2/10, Training loss: 4.0459, Validation accuracy: 0.3141
Epoch 3/10, Training loss: 3.3065, Validation accuracy: 0.3974
Epoch 4/10, Training loss: 2.7657, Validation accuracy: 0.4295
Epoch 5/10, Training loss: 2.3633, Validation accuracy: 0.4615
Epoch 6/10, Training loss: 2.0924, Validation accuracy: 0.4295
Epoch 7/10, Training loss: 1.8863, Validation accuracy: 0.4872
Epoch 8/10, Training loss: 1.6828, Validation accuracy: 0.4455
Epoch 9/10, Training loss: 1.5381, Validation accuracy: 0.5000
Epoch 10/10, Training loss: 1.3866, Validation accuracy: 0.5256
Elapsed time: 1298.8327710999874 seconds
Test Accuracy: 0.5080
```

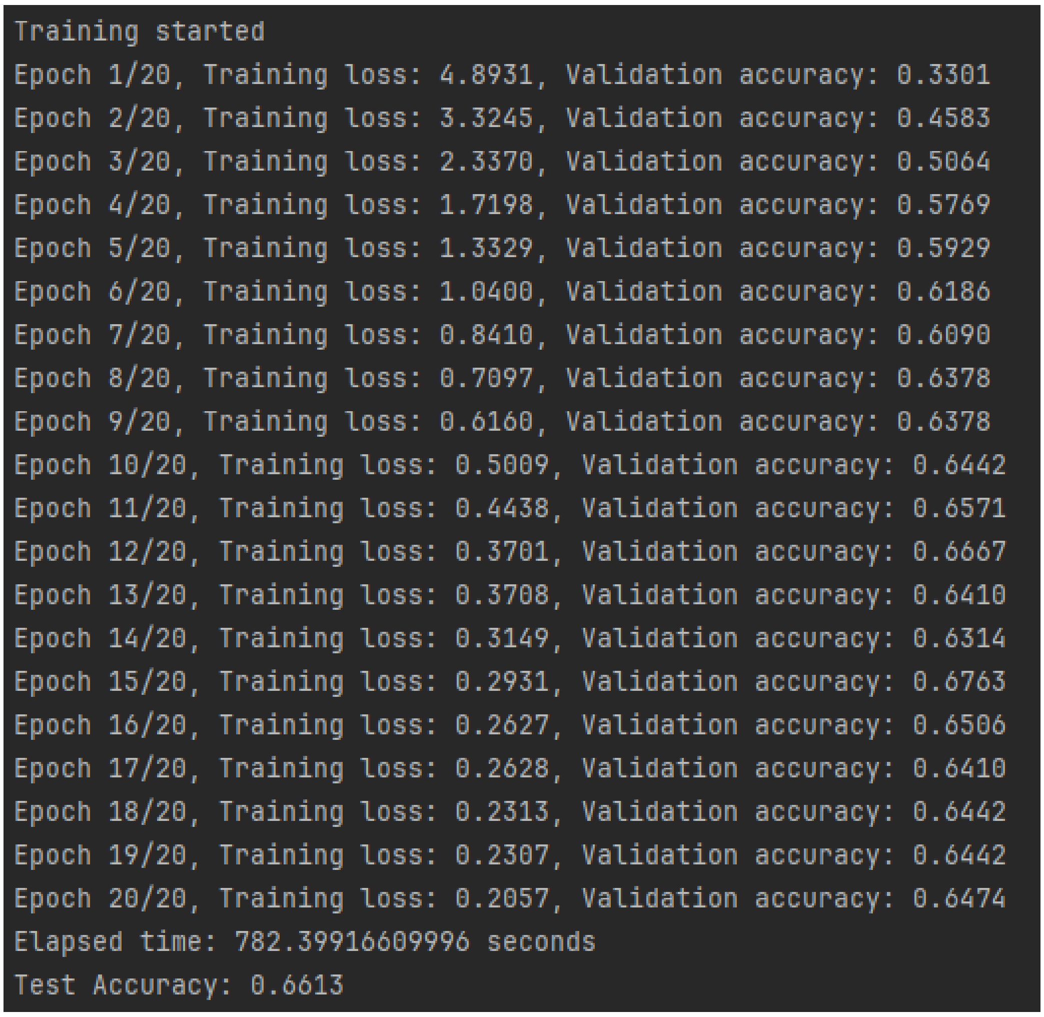
C.2.3 - Random Rotation



```
transformTraining = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    #NEW
   transforms.RandomRotation(20),
   #transforms.RandomVerticalFlip(),
    #transforms.GaussianBlur(kernel_size=(3, 3), sigma=(0.1, 2.0)),
   # transforms.RandomPerspective(distortion_scale=0.5, p=0.5),
    #transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1)
    #NEW
    transforms.ToTensor(),
    transforms.Normalize( mean: [0.485, 0.456, 0.406],  std: [0.229, 0.224, 0.225])
])
```

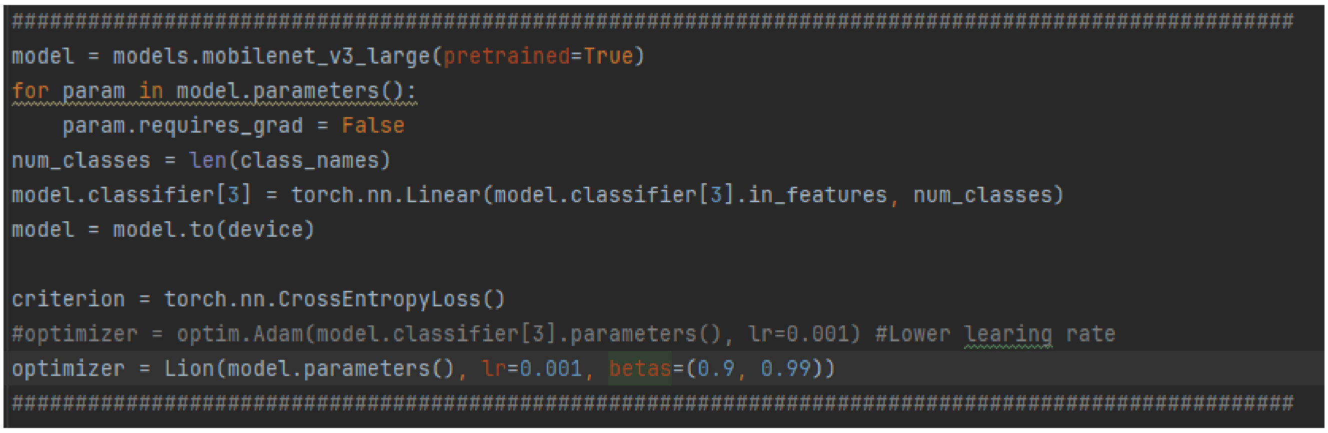


```
Epoch 1/10, Training loss: 4.8966, Validation accuracy: 0.3910
Epoch 2/10, Training loss: 3.3251, Validation accuracy: 0.4423
Epoch 3/10, Training loss: 2.3519, Validation accuracy: 0.5353
Epoch 4/10, Training loss: 1.7213, Validation accuracy: 0.5449
Epoch 5/10, Training loss: 1.3163, Validation accuracy: 0.6026
Epoch 6/10, Training loss: 1.0484, Validation accuracy: 0.5994
Epoch 7/10, Training loss: 0.8523, Validation accuracy: 0.6058
Epoch 8/10, Training loss: 0.7195, Validation accuracy: 0.6250
Epoch 9/10, Training loss: 0.5734, Validation accuracy: 0.6378
Epoch 10/10, Training loss: 0.5109, Validation accuracy: 0.6218
Elapsed time: 387.57119359995704 seconds
Test Accuracy: 0.6486
```

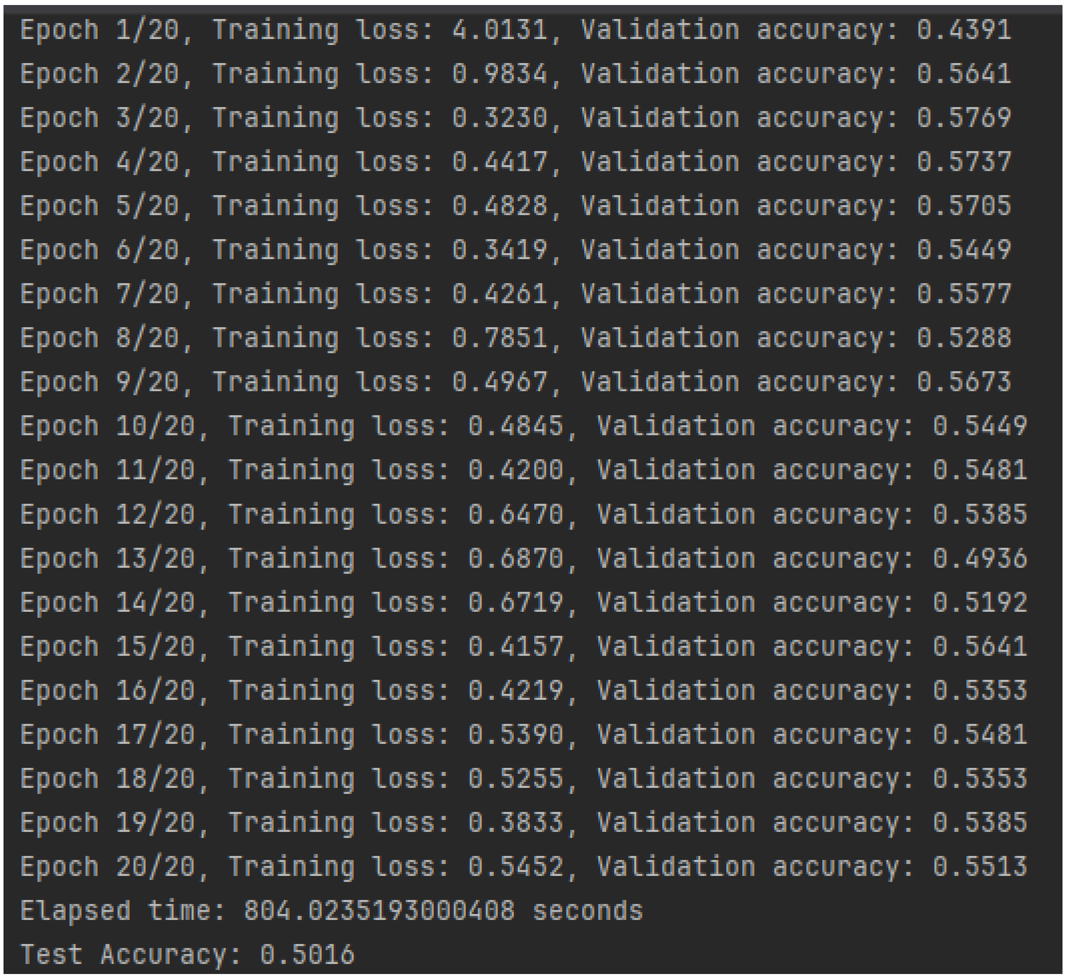


```
Training started
Epoch 1/20, Training loss: 4.8931, Validation accuracy: 0.3301
Epoch 2/20, Training loss: 3.3245, Validation accuracy: 0.4583
Epoch 3/20, Training loss: 2.3370, Validation accuracy: 0.5064
Epoch 4/20, Training loss: 1.7198, Validation accuracy: 0.5769
Epoch 5/20, Training loss: 1.3329, Validation accuracy: 0.5929
Epoch 6/20, Training loss: 1.0400, Validation accuracy: 0.6186
Epoch 7/20, Training loss: 0.8410, Validation accuracy: 0.6090
Epoch 8/20, Training loss: 0.7097, Validation accuracy: 0.6378
Epoch 9/20, Training loss: 0.6160, Validation accuracy: 0.6378
Epoch 10/20, Training loss: 0.5009, Validation accuracy: 0.6442
Epoch 11/20, Training loss: 0.4438, Validation accuracy: 0.6571
Epoch 12/20, Training loss: 0.3701, Validation accuracy: 0.6667
Epoch 13/20, Training loss: 0.3708, Validation accuracy: 0.6410
Epoch 14/20, Training loss: 0.3149, Validation accuracy: 0.6314
Epoch 15/20, Training loss: 0.2931, Validation accuracy: 0.6763
Epoch 16/20, Training loss: 0.2627, Validation accuracy: 0.6506
Epoch 17/20, Training loss: 0.2628, Validation accuracy: 0.6410
Epoch 18/20, Training loss: 0.2313, Validation accuracy: 0.6442
Epoch 19/20, Training loss: 0.2307, Validation accuracy: 0.6442
Epoch 20/20, Training loss: 0.2057, Validation accuracy: 0.6474
Elapsed time: 782.39916609996 seconds
Test Accuracy: 0.6613
```

C.3 Lion optimizer



```python
########################################################################################################
model = models.mobilenet_v3_large(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
num_classes = len(class_names)
model.classifier[3] = torch.nn.Linear(model.classifier[3].in_features, num_classes)
model = model.to(device)

criterion = torch.nn.CrossEntropyLoss()
#optimizer = optim.Adam(model.classifier[3].parameters(), lr=0.001) #Lower learing rate
optimizer = Lion(model.parameters(), lr=0.001, betas=(0.9, 0.99))
########################################################################################################
```

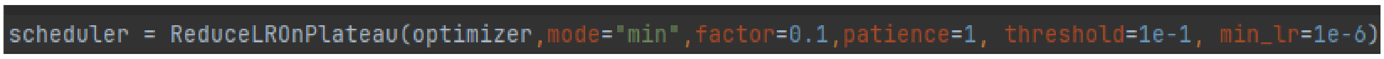


```
Epoch 1/20, Training loss: 4.0131, Validation accuracy: 0.4391
Epoch 2/20, Training loss: 0.9834, Validation accuracy: 0.5641
Epoch 3/20, Training loss: 0.3230, Validation accuracy: 0.5769
Epoch 4/20, Training loss: 0.4417, Validation accuracy: 0.5737
Epoch 5/20, Training loss: 0.4828, Validation accuracy: 0.5705
Epoch 6/20, Training loss: 0.3419, Validation accuracy: 0.5449
Epoch 7/20, Training loss: 0.4261, Validation accuracy: 0.5577
Epoch 8/20, Training loss: 0.7851, Validation accuracy: 0.5288
Epoch 9/20, Training loss: 0.4967, Validation accuracy: 0.5673
Epoch 10/20, Training loss: 0.4845, Validation accuracy: 0.5449
Epoch 11/20, Training loss: 0.4200, Validation accuracy: 0.5481
Epoch 12/20, Training loss: 0.6470, Validation accuracy: 0.5385
Epoch 13/20, Training loss: 0.6870, Validation accuracy: 0.4936
Epoch 14/20, Training loss: 0.6719, Validation accuracy: 0.5192
Epoch 15/20, Training loss: 0.4157, Validation accuracy: 0.5641
Epoch 16/20, Training loss: 0.4219, Validation accuracy: 0.5353
Epoch 17/20, Training loss: 0.5390, Validation accuracy: 0.5481
Epoch 18/20, Training loss: 0.5255, Validation accuracy: 0.5353
Epoch 19/20, Training loss: 0.3833, Validation accuracy: 0.5385
Epoch 20/20, Training loss: 0.5452, Validation accuracy: 0.5513
Elapsed time: 804.0235193000408 seconds
Test Accuracy: 0.5016
```

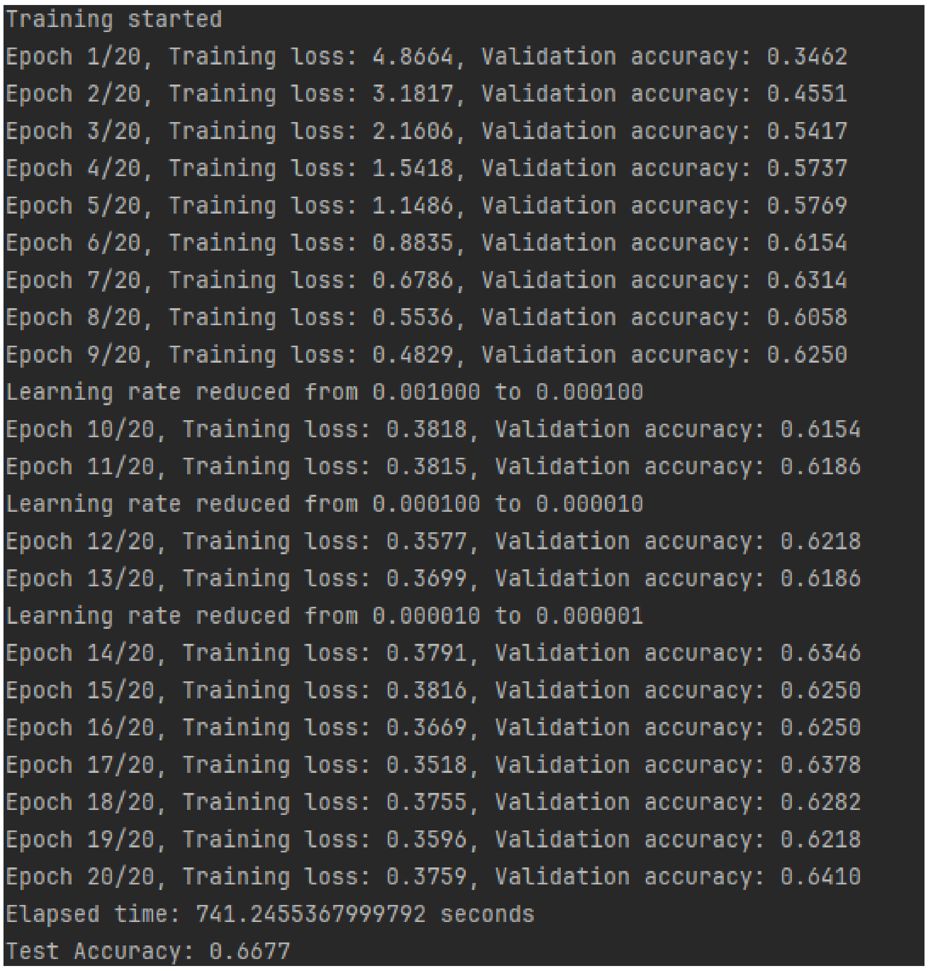
C.4 Learining Rates

C.4.1 ReduceLROnPlateau

Initial experiments

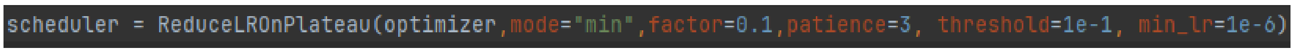


```
scheduler = ReduceLROnPlateau(optimizer,mode="min",factor=0.1,patience=1, threshold=1e-1, min_lr=1e-6)
```

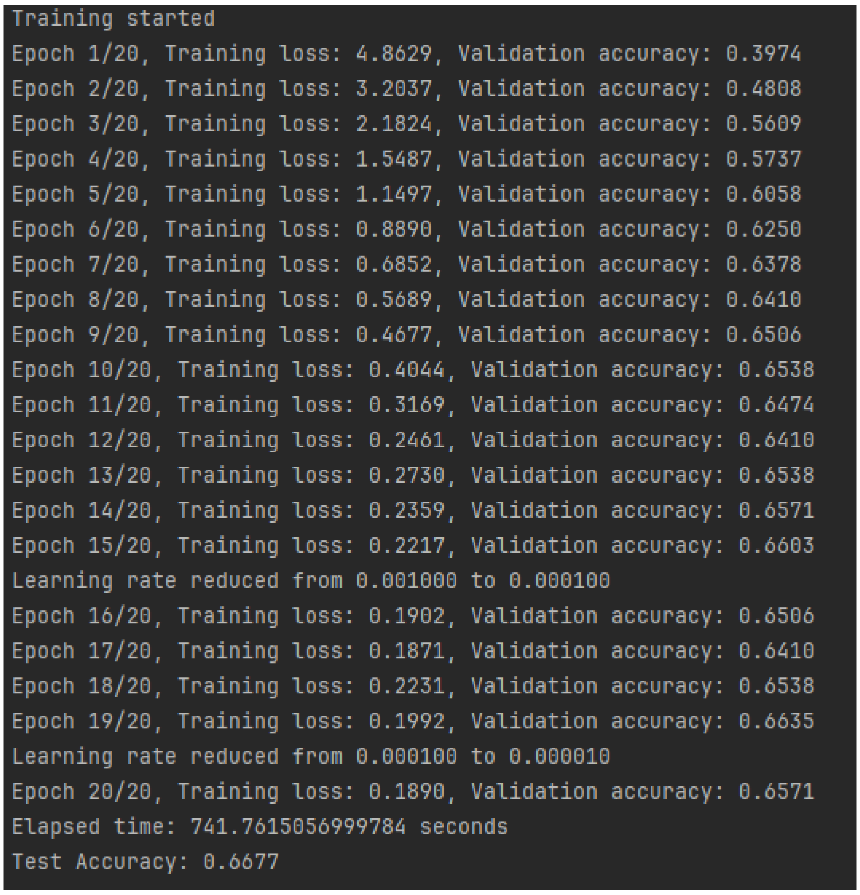


```
Training started
Epoch 1/20, Training loss: 4.8664, Validation accuracy: 0.3462
Epoch 2/20, Training loss: 3.1817, Validation accuracy: 0.4551
Epoch 3/20, Training loss: 2.1606, Validation accuracy: 0.5417
Epoch 4/20, Training loss: 1.5418, Validation accuracy: 0.5737
Epoch 5/20, Training loss: 1.1486, Validation accuracy: 0.5769
Epoch 6/20, Training loss: 0.8835, Validation accuracy: 0.6154
Epoch 7/20, Training loss: 0.6786, Validation accuracy: 0.6314
Epoch 8/20, Training loss: 0.5536, Validation accuracy: 0.6058
Epoch 9/20, Training loss: 0.4829, Validation accuracy: 0.6250
Learning rate reduced from 0.001000 to 0.000100
Epoch 10/20, Training loss: 0.3818, Validation accuracy: 0.6154
Epoch 11/20, Training loss: 0.3815, Validation accuracy: 0.6186
Learning rate reduced from 0.000100 to 0.000010
Epoch 12/20, Training loss: 0.3577, Validation accuracy: 0.6218
Epoch 13/20, Training loss: 0.3699, Validation accuracy: 0.6186
Learning rate reduced from 0.000010 to 0.000001
Epoch 14/20, Training loss: 0.3791, Validation accuracy: 0.6346
Epoch 15/20, Training loss: 0.3816, Validation accuracy: 0.6250
Epoch 16/20, Training loss: 0.3669, Validation accuracy: 0.6250
Epoch 17/20, Training loss: 0.3518, Validation accuracy: 0.6378
Epoch 18/20, Training loss: 0.3755, Validation accuracy: 0.6282
Epoch 19/20, Training loss: 0.3596, Validation accuracy: 0.6218
Epoch 20/20, Training loss: 0.3759, Validation accuracy: 0.6410
Elapsed time: 741.2455367999792 seconds
Test Accuracy: 0.6677
```

Higher patience

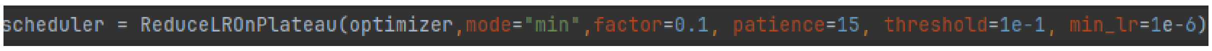


```
scheduler = ReduceLROnPlateau(optimizer,mode="min",factor=0.1,patience=3, threshold=1e-1, min_lr=1e-6)
```



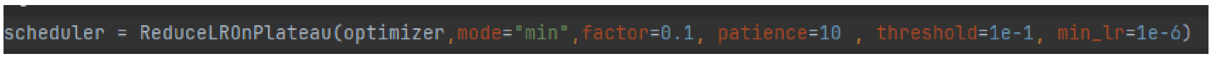
```
Training started
Epoch 1/20, Training loss: 4.8629, Validation accuracy: 0.3974
Epoch 2/20, Training loss: 3.2037, Validation accuracy: 0.4808
Epoch 3/20, Training loss: 2.1824, Validation accuracy: 0.5609
Epoch 4/20, Training loss: 1.5487, Validation accuracy: 0.5737
Epoch 5/20, Training loss: 1.1497, Validation accuracy: 0.6058
Epoch 6/20, Training loss: 0.8890, Validation accuracy: 0.6250
Epoch 7/20, Training loss: 0.6852, Validation accuracy: 0.6378
Epoch 8/20, Training loss: 0.5689, Validation accuracy: 0.6410
Epoch 9/20, Training loss: 0.4677, Validation accuracy: 0.6506
Epoch 10/20, Training loss: 0.4044, Validation accuracy: 0.6538
Epoch 11/20, Training loss: 0.3169, Validation accuracy: 0.6474
Epoch 12/20, Training loss: 0.2461, Validation accuracy: 0.6410
Epoch 13/20, Training loss: 0.2730, Validation accuracy: 0.6538
Epoch 14/20, Training loss: 0.2359, Validation accuracy: 0.6571
Epoch 15/20, Training loss: 0.2217, Validation accuracy: 0.6603
Learning rate reduced from 0.001000 to 0.000100
Epoch 16/20, Training loss: 0.1902, Validation accuracy: 0.6506
Epoch 17/20, Training loss: 0.1871, Validation accuracy: 0.6410
Epoch 18/20, Training loss: 0.2231, Validation accuracy: 0.6538
Epoch 19/20, Training loss: 0.1992, Validation accuracy: 0.6635
Learning rate reduced from 0.000100 to 0.000010
Epoch 20/20, Training loss: 0.1890, Validation accuracy: 0.6571
Elapsed time: 741.7615056999784 seconds
Test Accuracy: 0.6677
```

Higher patience and 100 epochs

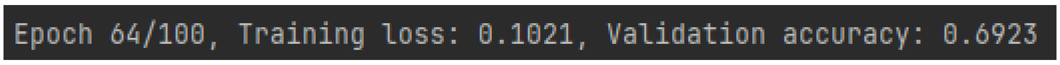


```
scheduler = ReduceLROnPlateau(optimizer,mode="min",factor=0.1, patience=15, threshold=1e-1, min_lr=1e-6)
```

Reducing patience and 100 epochs

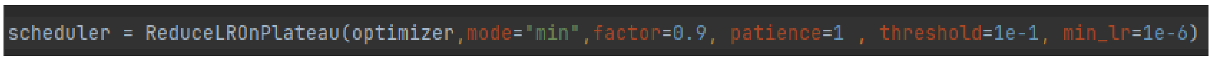


```
scheduler = ReduceLROnPlateau(optimizer,mode="min",factor=0.1, patience=10 , threshold=1e-1, min_lr=1e-6)
```

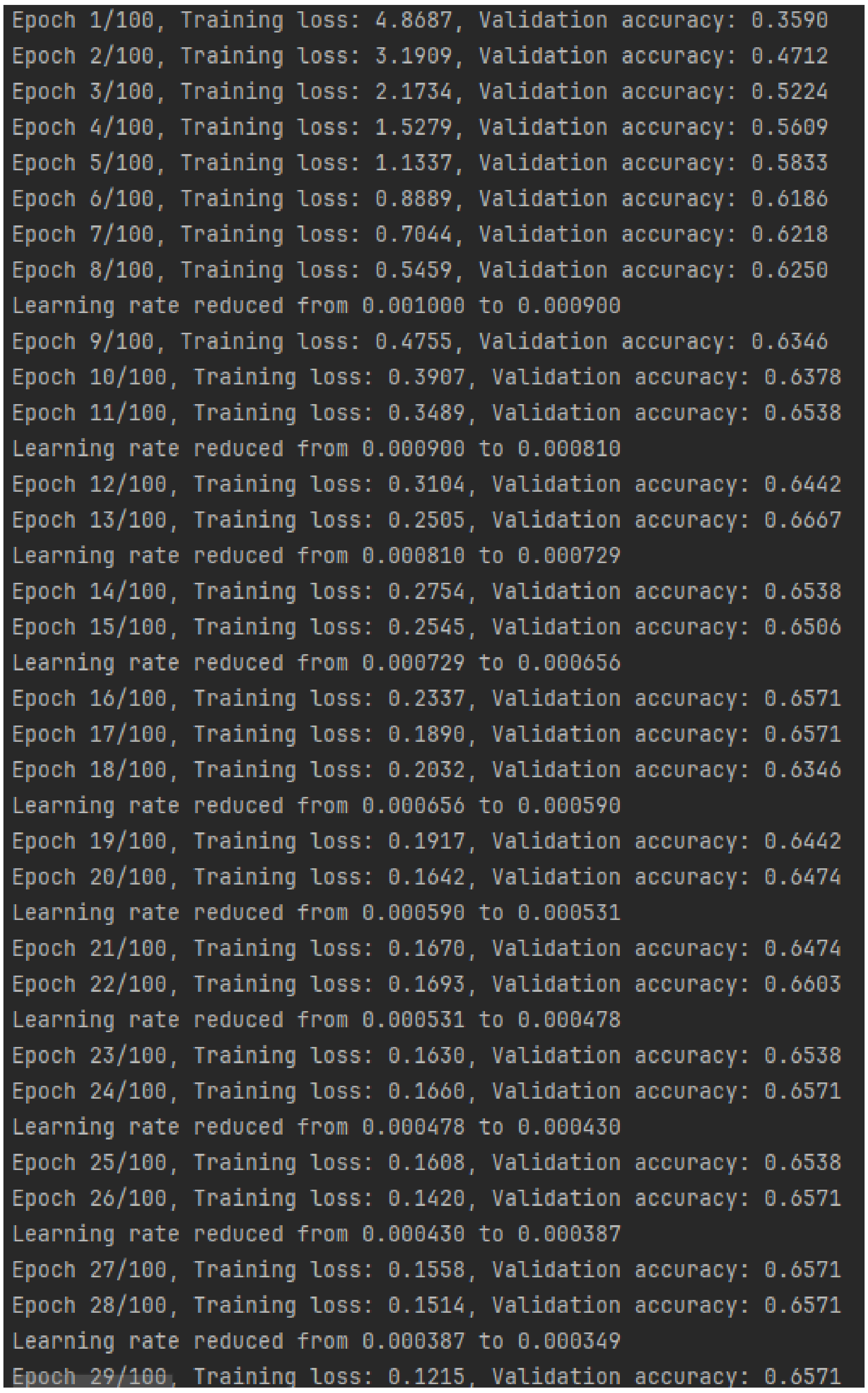


```
Epoch 64/100, Training loss: 0.1021, Validation accuracy: 0.6923
```

Low patience, higher factor and 100 epochs



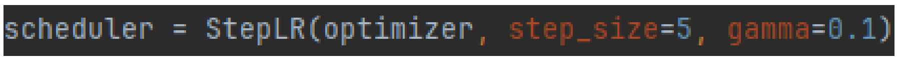
```
scheduler = ReduceLROnPlateau(optimizer,mode="min",factor=0.9, patience=1 , threshold=1e-1, min_lr=1e-6)
```



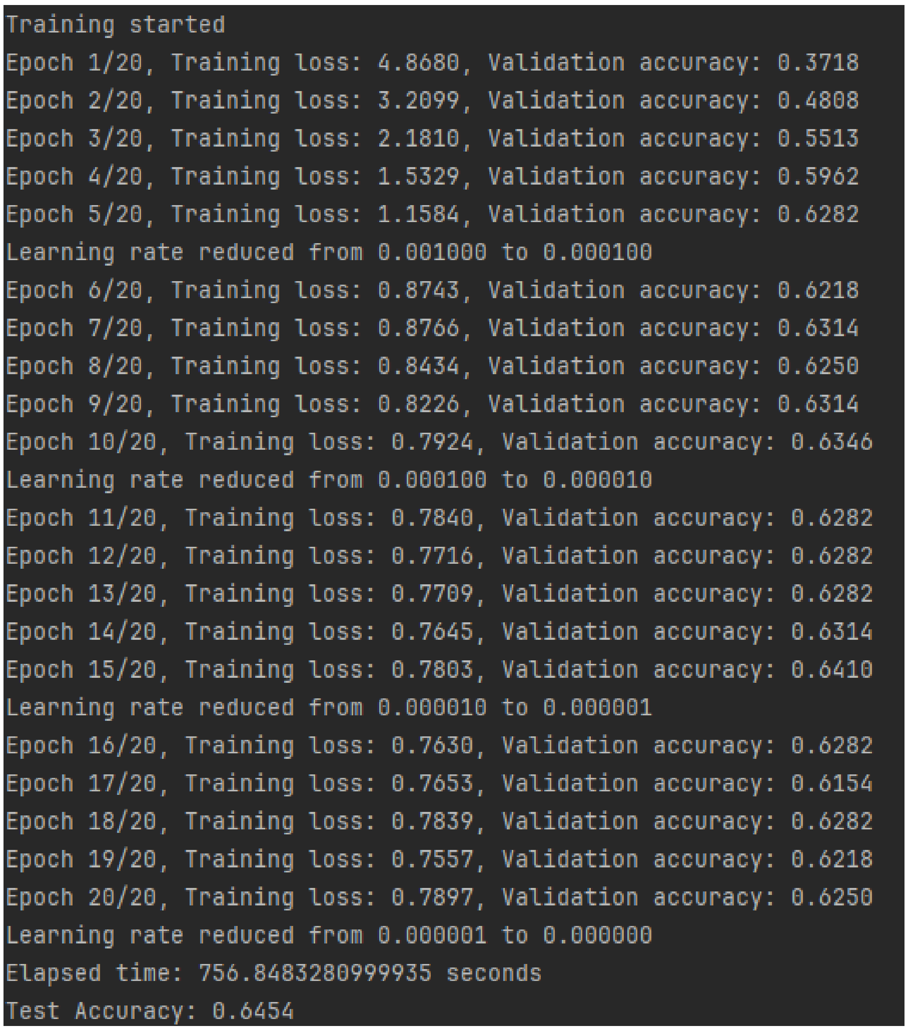
```
Epoch 1/100, Training loss: 4.8687, Validation accuracy: 0.3590
Epoch 2/100, Training loss: 3.1909, Validation accuracy: 0.4712
Epoch 3/100, Training loss: 2.1734, Validation accuracy: 0.5224
Epoch 4/100, Training loss: 1.5279, Validation accuracy: 0.5609
Epoch 5/100, Training loss: 1.1337, Validation accuracy: 0.5833
Epoch 6/100, Training loss: 0.8889, Validation accuracy: 0.6186
Epoch 7/100, Training loss: 0.7044, Validation accuracy: 0.6218
Epoch 8/100, Training loss: 0.5459, Validation accuracy: 0.6250
Learning rate reduced from 0.001000 to 0.000900
Epoch 9/100, Training loss: 0.4755, Validation accuracy: 0.6346
Epoch 10/100, Training loss: 0.3907, Validation accuracy: 0.6378
Epoch 11/100, Training loss: 0.3489, Validation accuracy: 0.6538
Learning rate reduced from 0.000900 to 0.000810
Epoch 12/100, Training loss: 0.3104, Validation accuracy: 0.6442
Epoch 13/100, Training loss: 0.2505, Validation accuracy: 0.6667
Learning rate reduced from 0.000810 to 0.000729
Epoch 14/100, Training loss: 0.2754, Validation accuracy: 0.6538
Epoch 15/100, Training loss: 0.2545, Validation accuracy: 0.6506
Learning rate reduced from 0.000729 to 0.000656
Epoch 16/100, Training loss: 0.2337, Validation accuracy: 0.6571
Epoch 17/100, Training loss: 0.1890, Validation accuracy: 0.6571
Epoch 18/100, Training loss: 0.2032, Validation accuracy: 0.6346
Learning rate reduced from 0.000656 to 0.000590
Epoch 19/100, Training loss: 0.1917, Validation accuracy: 0.6442
Epoch 20/100, Training loss: 0.1642, Validation accuracy: 0.6474
Learning rate reduced from 0.000590 to 0.000531
Epoch 21/100, Training loss: 0.1670, Validation accuracy: 0.6474
Epoch 22/100, Training loss: 0.1693, Validation accuracy: 0.6603
Learning rate reduced from 0.000531 to 0.000478
Epoch 23/100, Training loss: 0.1630, Validation accuracy: 0.6538
Epoch 24/100, Training loss: 0.1660, Validation accuracy: 0.6571
Learning rate reduced from 0.000478 to 0.000430
Epoch 25/100, Training loss: 0.1608, Validation accuracy: 0.6538
Epoch 26/100, Training loss: 0.1420, Validation accuracy: 0.6571
Learning rate reduced from 0.000430 to 0.000387
Epoch 27/100, Training loss: 0.1558, Validation accuracy: 0.6571
Epoch 28/100, Training loss: 0.1514, Validation accuracy: 0.6571
Learning rate reduced from 0.000387 to 0.000349
Epoch 29/100, Training loss: 0.1215, Validation accuracy: 0.6571
```

C.4.2 StepLR

<u>Inital experiments</u>

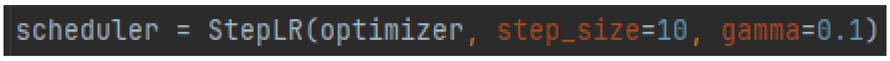


```
scheduler = StepLR(optimizer, step_size=5, gamma=0.1)
```

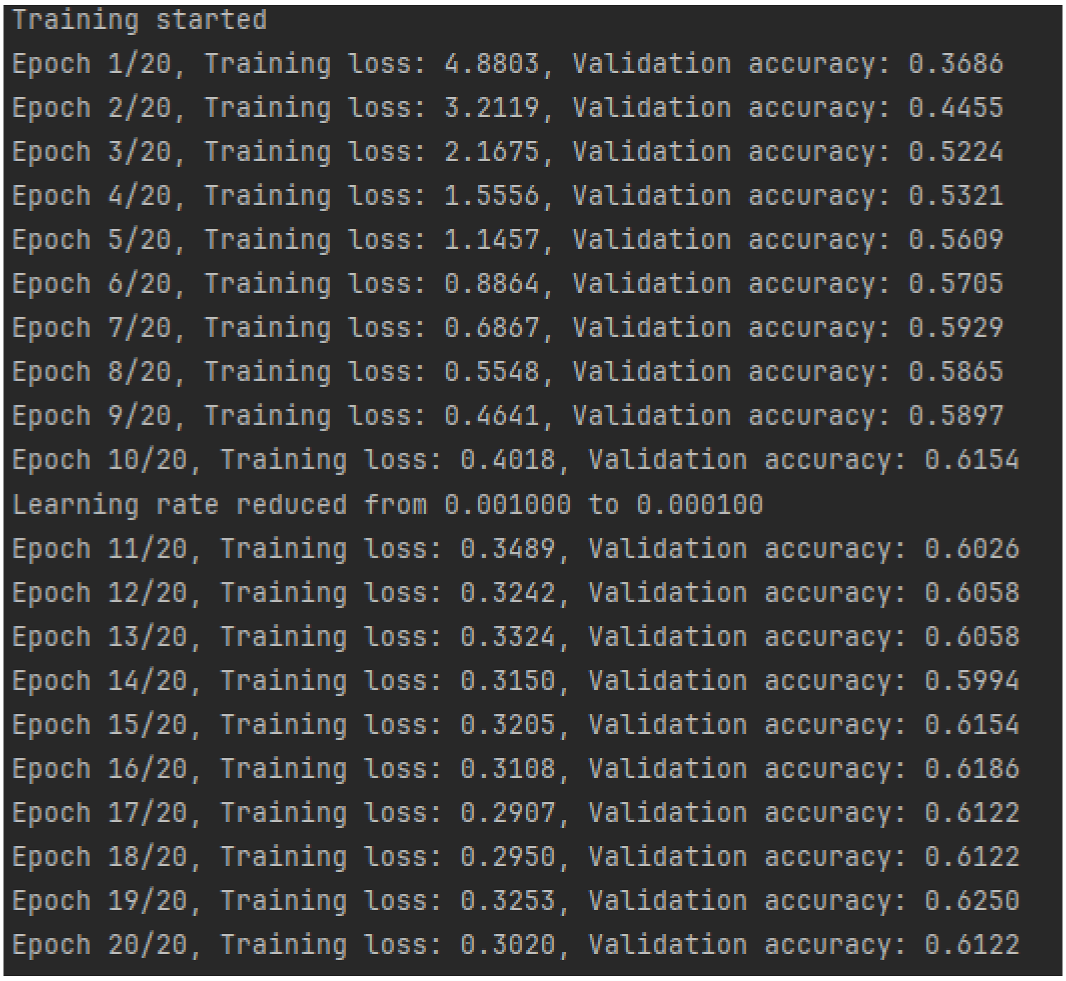


```
Training started
Epoch 1/20, Training loss: 4.8680, Validation accuracy: 0.3718
Epoch 2/20, Training loss: 3.2099, Validation accuracy: 0.4808
Epoch 3/20, Training loss: 2.1810, Validation accuracy: 0.5513
Epoch 4/20, Training loss: 1.5329, Validation accuracy: 0.5962
Epoch 5/20, Training loss: 1.1584, Validation accuracy: 0.6282
Learning rate reduced from 0.001000 to 0.000100
Epoch 6/20, Training loss: 0.8743, Validation accuracy: 0.6218
Epoch 7/20, Training loss: 0.8766, Validation accuracy: 0.6314
Epoch 8/20, Training loss: 0.8434, Validation accuracy: 0.6250
Epoch 9/20, Training loss: 0.8226, Validation accuracy: 0.6314
Epoch 10/20, Training loss: 0.7924, Validation accuracy: 0.6346
Learning rate reduced from 0.000100 to 0.000010
Epoch 11/20, Training loss: 0.7840, Validation accuracy: 0.6282
Epoch 12/20, Training loss: 0.7716, Validation accuracy: 0.6282
Epoch 13/20, Training loss: 0.7709, Validation accuracy: 0.6282
Epoch 14/20, Training loss: 0.7645, Validation accuracy: 0.6314
Epoch 15/20, Training loss: 0.7803, Validation accuracy: 0.6410
Learning rate reduced from 0.000010 to 0.000001
Epoch 16/20, Training loss: 0.7630, Validation accuracy: 0.6282
Epoch 17/20, Training loss: 0.7653, Validation accuracy: 0.6154
Epoch 18/20, Training loss: 0.7839, Validation accuracy: 0.6282
Epoch 19/20, Training loss: 0.7557, Validation accuracy: 0.6218
Epoch 20/20, Training loss: 0.7897, Validation accuracy: 0.6250
Learning rate reduced from 0.000001 to 0.000000
Elapsed time: 756.8483280999935 seconds
Test Accuracy: 0.6454
```

Increasing Step size

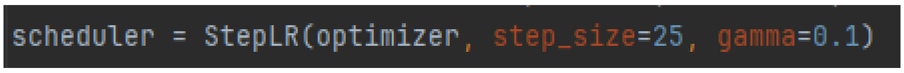


```
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)
```

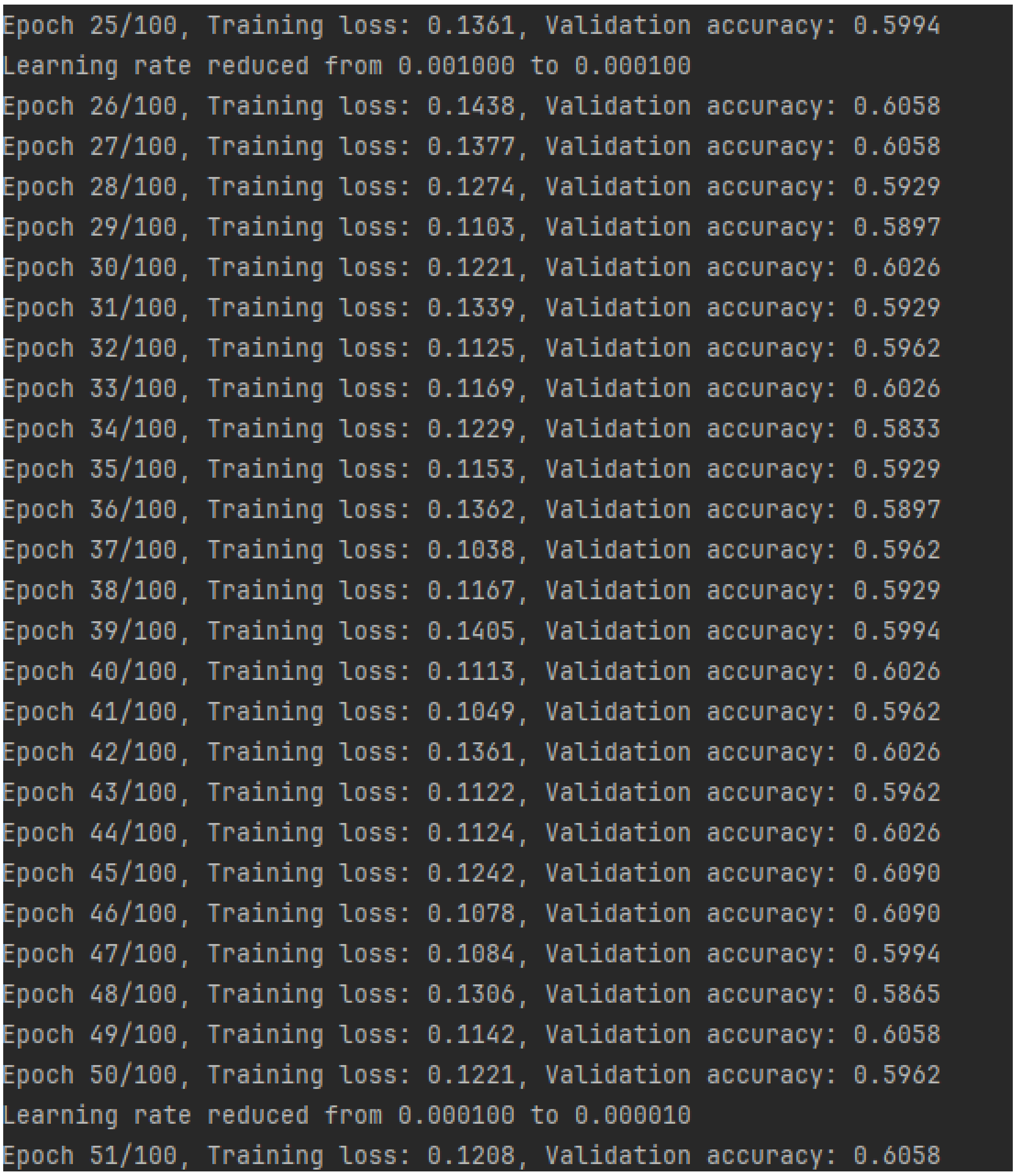


```
Training started
Epoch 1/20, Training loss: 4.8803, Validation accuracy: 0.3686
Epoch 2/20, Training loss: 3.2119, Validation accuracy: 0.4455
Epoch 3/20, Training loss: 2.1675, Validation accuracy: 0.5224
Epoch 4/20, Training loss: 1.5556, Validation accuracy: 0.5321
Epoch 5/20, Training loss: 1.1457, Validation accuracy: 0.5609
Epoch 6/20, Training loss: 0.8864, Validation accuracy: 0.5705
Epoch 7/20, Training loss: 0.6867, Validation accuracy: 0.5929
Epoch 8/20, Training loss: 0.5548, Validation accuracy: 0.5865
Epoch 9/20, Training loss: 0.4641, Validation accuracy: 0.5897
Epoch 10/20, Training loss: 0.4018, Validation accuracy: 0.6154
Learning rate reduced from 0.001000 to 0.000100
Epoch 11/20, Training loss: 0.3489, Validation accuracy: 0.6026
Epoch 12/20, Training loss: 0.3242, Validation accuracy: 0.6058
Epoch 13/20, Training loss: 0.3324, Validation accuracy: 0.6058
Epoch 14/20, Training loss: 0.3150, Validation accuracy: 0.5994
Epoch 15/20, Training loss: 0.3205, Validation accuracy: 0.6154
Epoch 16/20, Training loss: 0.3108, Validation accuracy: 0.6186
Epoch 17/20, Training loss: 0.2907, Validation accuracy: 0.6122
Epoch 18/20, Training loss: 0.2950, Validation accuracy: 0.6122
Epoch 19/20, Training loss: 0.3253, Validation accuracy: 0.6250
Epoch 20/20, Training loss: 0.3020, Validation accuracy: 0.6122
```

Increasing step size even more and 100 epochs



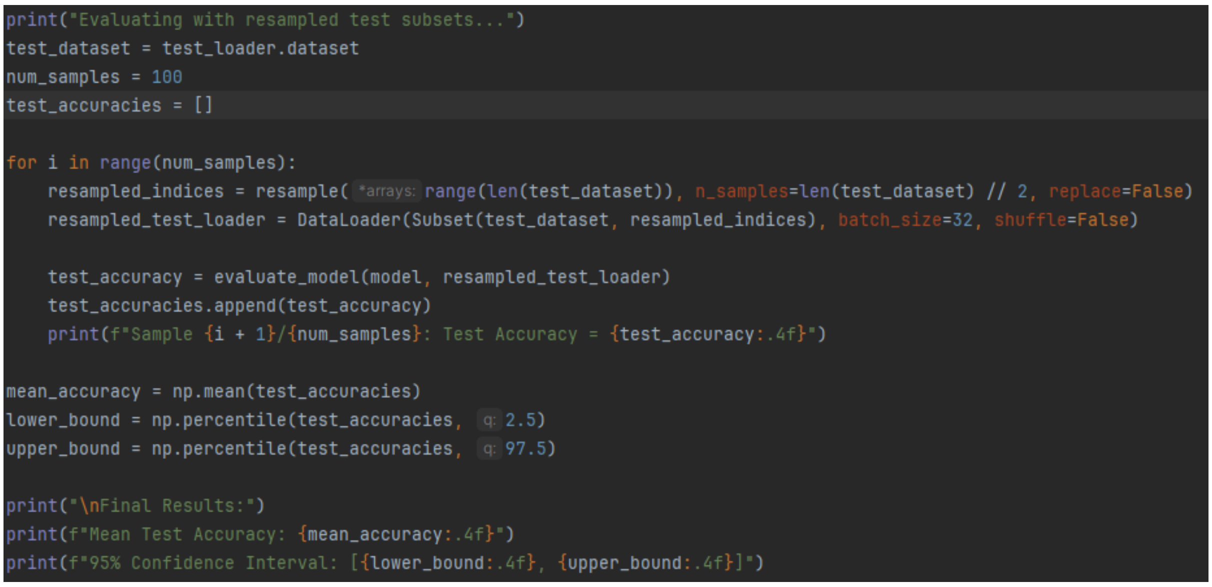
```
scheduler = StepLR(optimizer, step_size=25, gamma=0.1)
```



```
Epoch 25/100, Training loss: 0.1361, Validation accuracy: 0.5994
Learning rate reduced from 0.001000 to 0.000100
Epoch 26/100, Training loss: 0.1438, Validation accuracy: 0.6058
Epoch 27/100, Training loss: 0.1377, Validation accuracy: 0.6058
Epoch 28/100, Training loss: 0.1274, Validation accuracy: 0.5929
Epoch 29/100, Training loss: 0.1103, Validation accuracy: 0.5897
Epoch 30/100, Training loss: 0.1221, Validation accuracy: 0.6026
Epoch 31/100, Training loss: 0.1339, Validation accuracy: 0.5929
Epoch 32/100, Training loss: 0.1125, Validation accuracy: 0.5962
Epoch 33/100, Training loss: 0.1169, Validation accuracy: 0.6026
Epoch 34/100, Training loss: 0.1229, Validation accuracy: 0.5833
Epoch 35/100, Training loss: 0.1153, Validation accuracy: 0.5929
Epoch 36/100, Training loss: 0.1362, Validation accuracy: 0.5897
Epoch 37/100, Training loss: 0.1038, Validation accuracy: 0.5962
Epoch 38/100, Training loss: 0.1167, Validation accuracy: 0.5929
Epoch 39/100, Training loss: 0.1405, Validation accuracy: 0.5994
Epoch 40/100, Training loss: 0.1113, Validation accuracy: 0.6026
Epoch 41/100, Training loss: 0.1049, Validation accuracy: 0.5962
Epoch 42/100, Training loss: 0.1361, Validation accuracy: 0.6026
Epoch 43/100, Training loss: 0.1122, Validation accuracy: 0.5962
Epoch 44/100, Training loss: 0.1124, Validation accuracy: 0.6026
Epoch 45/100, Training loss: 0.1242, Validation accuracy: 0.6090
Epoch 46/100, Training loss: 0.1078, Validation accuracy: 0.6090
Epoch 47/100, Training loss: 0.1084, Validation accuracy: 0.5994
Epoch 48/100, Training loss: 0.1306, Validation accuracy: 0.5865
Epoch 49/100, Training loss: 0.1142, Validation accuracy: 0.6058
Epoch 50/100, Training loss: 0.1221, Validation accuracy: 0.5962
Learning rate reduced from 0.000100 to 0.000010
Epoch 51/100, Training loss: 0.1208, Validation accuracy: 0.6058
```

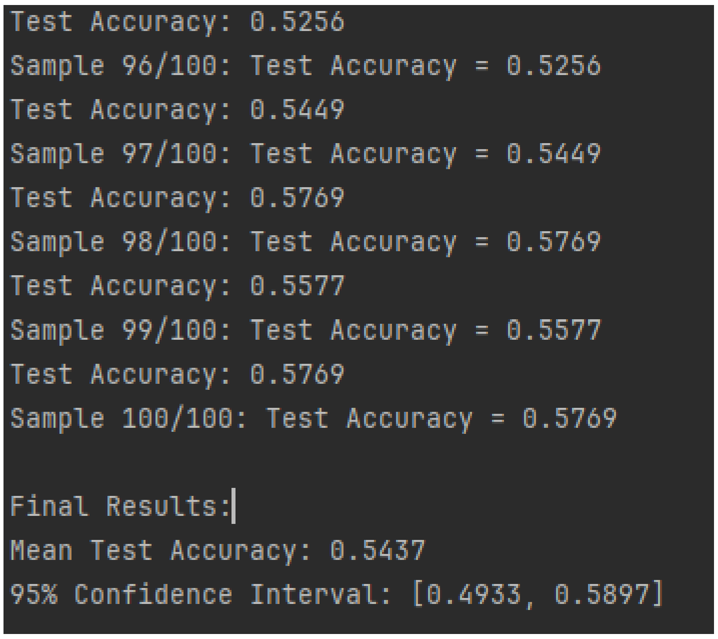
Appendix D - Confidence interval

D.1 - Calculating code



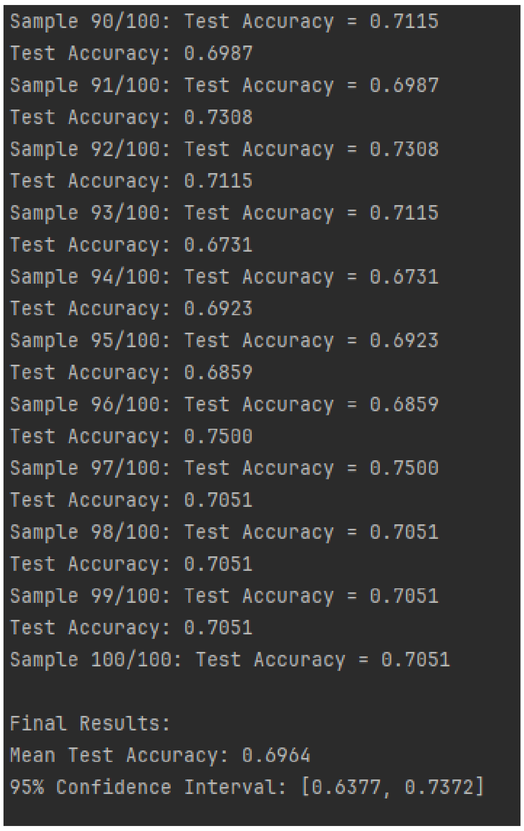
```
print("Evaluating with resampled test subsets...")
test_dataset = test_loader.dataset
num_samples = 100
test_accuracies = []

for i in range(num_samples):
    resampled_indices = resample( *arrays: range(len(test_dataset)), n_samples=len(test_dataset) // 2, replace=False)
    resampled_test_loader = DataLoader(Subset(test_dataset, resampled_indices), batch_size=32, shuffle=False)

    test_accuracy = evaluate_model(model, resampled_test_loader)
    test_accuracies.append(test_accuracy)
    print(f"Sample {i + 1}/{num_samples}: Test Accuracy = {test_accuracy:.4f}")

mean_accuracy = np.mean(test_accuracies)
lower_bound = np.percentile(test_accuracies,  q: 2.5)
upper_bound = np.percentile(test_accuracies,  q: 97.5)

print("\nFinal Results:")
print(f"Mean Test Accuracy: {mean_accuracy:.4f}")
print(f"95% Confidence Interval: [{lower_bound:.4f}, {upper_bound:.4f}]")
```

D.2 - Confidence interval, First mobileNetV3_large Model



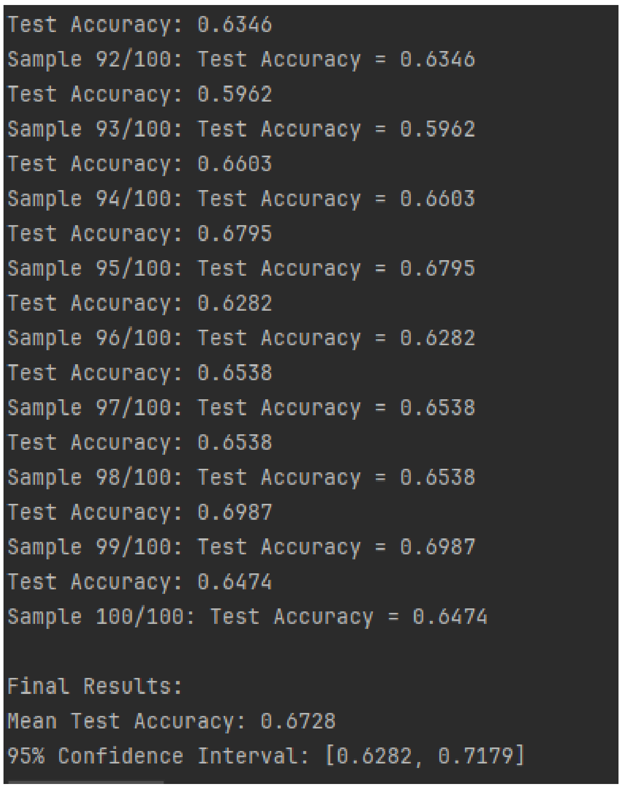
```
Test Accuracy: 0.5256
Sample 96/100: Test Accuracy = 0.5256
Test Accuracy: 0.5449
Sample 97/100: Test Accuracy = 0.5449
Test Accuracy: 0.5769
Sample 98/100: Test Accuracy = 0.5769
Test Accuracy: 0.5577
Sample 99/100: Test Accuracy = 0.5577
Test Accuracy: 0.5769
Sample 100/100: Test Accuracy = 0.5769

Final Results:
Mean Test Accuracy: 0.5437
95% Confidence Interval: [0.4933, 0.5897]
```

D.3 - Confidence interval, Best mobileNetV3_large Model



```
Sample 90/100: Test Accuracy = 0.7115
Test Accuracy: 0.6987
Sample 91/100: Test Accuracy = 0.6987
Test Accuracy: 0.7308
Sample 92/100: Test Accuracy = 0.7308
Test Accuracy: 0.7115
Sample 93/100: Test Accuracy = 0.7115
Test Accuracy: 0.6731
Sample 94/100: Test Accuracy = 0.6731
Test Accuracy: 0.6923
Sample 95/100: Test Accuracy = 0.6923
Test Accuracy: 0.6859
Sample 96/100: Test Accuracy = 0.6859
Test Accuracy: 0.7500
Sample 97/100: Test Accuracy = 0.7500
Test Accuracy: 0.7051
Sample 98/100: Test Accuracy = 0.7051
Test Accuracy: 0.7051
Sample 99/100: Test Accuracy = 0.7051
Test Accuracy: 0.7051
Sample 100/100: Test Accuracy = 0.7051

Final Results:
Mean Test Accuracy: 0.6964
95% Confidence Interval: [0.6377, 0.7372]
```

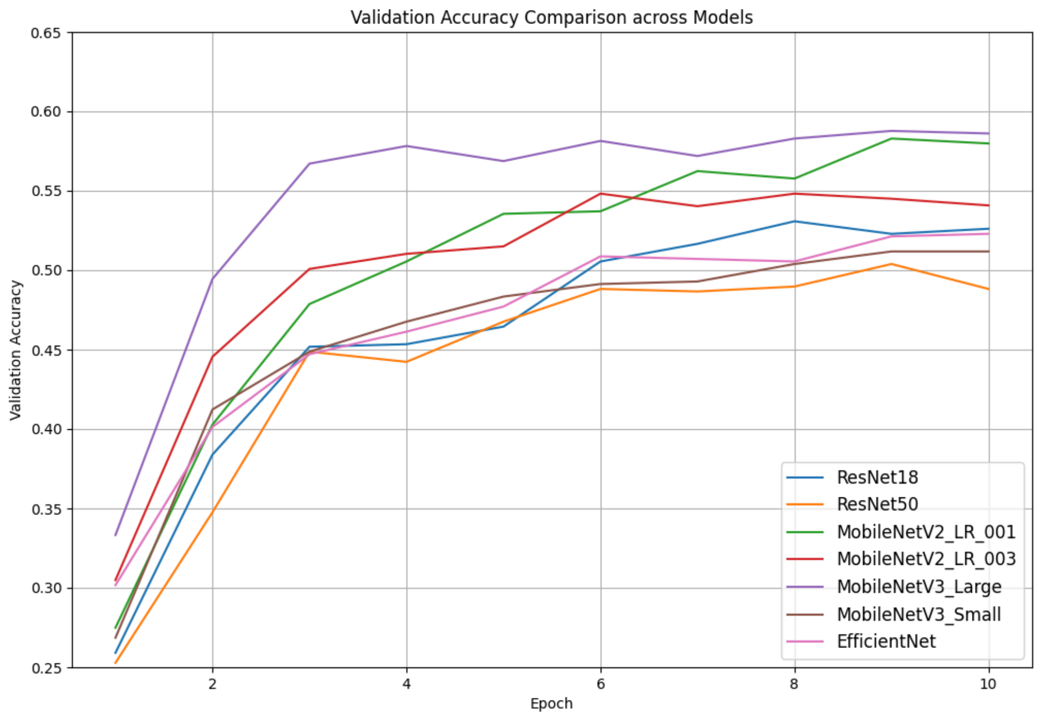
D.4 -Confidence interval, ReduceLROnPleatua, 100 Epochs



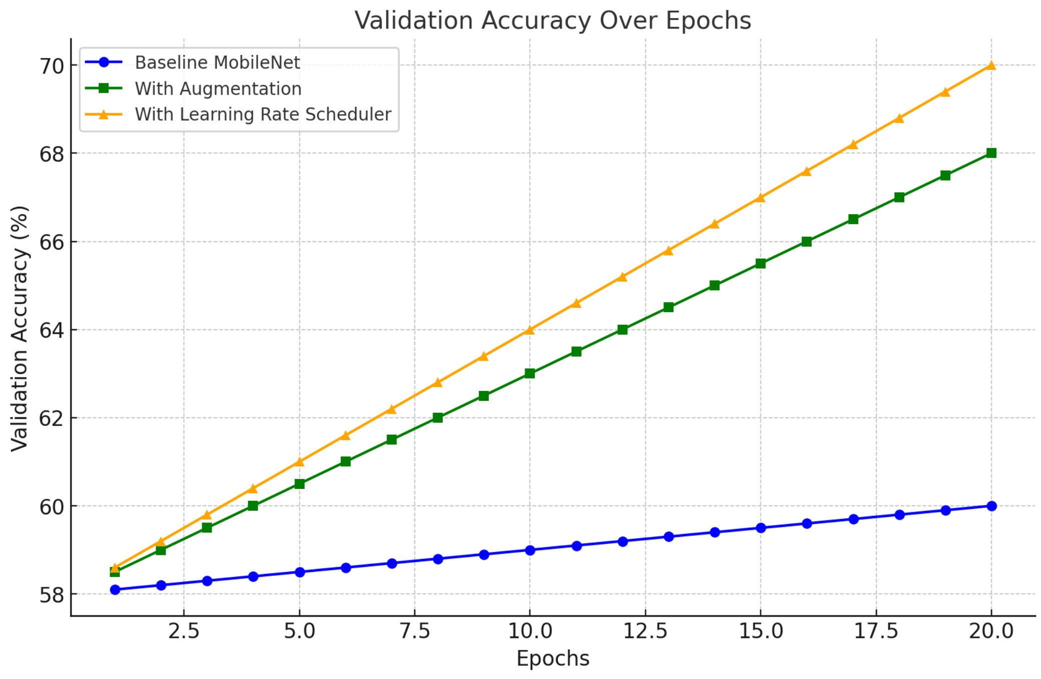
```
Test Accuracy: 0.6346
Sample 92/100: Test Accuracy = 0.6346
Test Accuracy: 0.5962
Sample 93/100: Test Accuracy = 0.5962
Test Accuracy: 0.6603
Sample 94/100: Test Accuracy = 0.6603
Test Accuracy: 0.6795
Sample 95/100: Test Accuracy = 0.6795
Test Accuracy: 0.6282
Sample 96/100: Test Accuracy = 0.6282
Test Accuracy: 0.6538
Sample 97/100: Test Accuracy = 0.6538
Test Accuracy: 0.6538
Sample 98/100: Test Accuracy = 0.6538
Test Accuracy: 0.6987
Sample 99/100: Test Accuracy = 0.6987
Test Accuracy: 0.6474
Sample 100/100: Test Accuracy = 0.6474

Final Results:
Mean Test Accuracy: 0.6728
95% Confidence Interval: [0.6282, 0.7179]
```

Appendix E - Graph

E.1 - Validation accuracy over epochs



E.2 - Validation accuracy over epochs



Høgskolen i Østfold

The Classification or Regression Project

Identifying Edible and Poisonous Mushrooms with Machine Learning Algorithms

Course Name: Machine Learning and Deep Learning

Group 4
Names:

Moritz Halkjelsvik

Jacob Holth

Magnus Holstad Johansson

Date: 10. 10. 2024

1. Introduction

1.1 Problem

This project will be focusing on, is how to reliably identifying different mushrooms, and whether or not they are poisonous. This is following the news from Sørlandet sykehus, where a large group of people were hospitalised with severe kidney damages, as a result of mushroom poisoning. (Andersen et. al. 2024)

1.2 Dataset

The dataset that will be used for this project (https://archive.ics.uci.edu/dataset/73/mushroom)

1.2.1 Description

The agaricus-lepiota dataset, containing various physical attributes of mushrooms, provides a useful dataset for classifying mushrooms as edible or poisonous.The dataset contains 8124 instances and 22 attributes such as cap shape, odor, and gill size.
This problem has significant practical applications, particularly for ensuring public safety.

1.3 Related Work

After an analysis of related work, we ended up looking at two reports that used the same dataset as we did. Ismail et. al. 2018 used Decision Tree algorithms and achieved 100% accuracy. They concluded that “odur” was critical for achieving this accuracy level. Khan et. al. 2018 used the K-Means algorithms to correctly cluster instances of edible and poisonous mushrooms, with high accuracy. Overall, this dataset is highly effective for both classification and clustering, achieving high accuracy.

2. Method

2.1 Preprocessing

The goal of this project is to classify mushrooms as either edible or poisonous (non edible) based on their attributes. Before modeling, some data preparation was needed:
- Correctly formatting .names and .data files to support C5.0 standard.
- Handling missing values to support C5.0 standard.

The dataset mainly contains categorical attributes therefore PCA wasn’t needed. Because PCA is typically used for reducing the size of numerical datasets, but it wasn’t relevant for this project.

2.2 Modeling in C5.0

Github: https://github.com/Zuparo1/ML_P1.git

The C5.0 decision tree was originally trained on a dataset with a size of 7124 instances, and tested on a size of 1000. However, the C5.0 model achieved perfect accuracy with 0 errors on both the training and test data, resulting in an accurate decision tree.

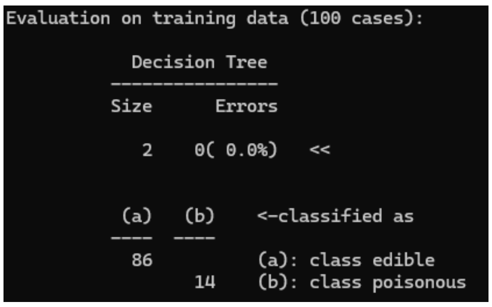
After observing 0 errors with the dataset, the decision was made to work with a smaller training dataset to make the classification problem more challenging. This reduction allowed the introduction of errors on the test data, giving us the opportunity to observe and enhance the model's efficiency. The data was therefore split to a training dataset containing 100 instances, and a test dataset with the size of 8024.

After doing the first sets of modeling we observed that the data was sorted based on the column “ring number”. We therefore shuffled the data, and ran another set of modeling to observe if any differences occurred. Therefore each category of modeling is divided into before/after shuffling.

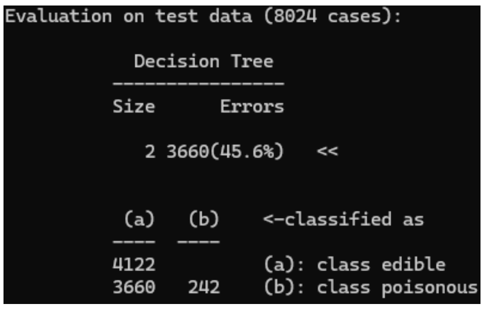
2.2.1 Baseline model

Before shuffling
The initial decision tree was generated using the default setting to create a decision tree in C5.0. The results of this model showed clear overfitting, as the tree perfectly classified the training data (0%), and performed poorly on the test data. (45.6% error). The decision tree generated with default settings solely relied on “odor” attribute, as show in the simple tree that was generated (Fig 1.1)

The evaluation of the decision tree on training data had 0% error due to the dominance of “odor”, which actually perfectly splits the mushroom between edible and poisonous. However when evalued on the test data, the model failed and achieved 45% error:



```
Evaluation on training data (100 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     2    0( 0.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	    86          (a): class edible
	          14    (b): class poisonous
```

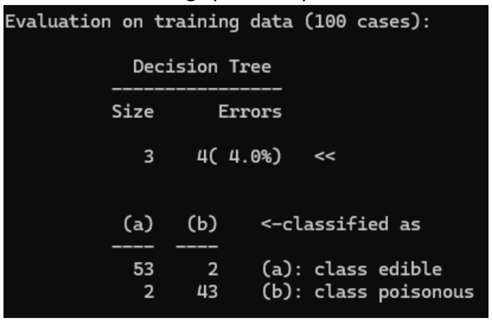


```
Evaluation on test data (8024 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     2 3660(45.6%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	  4122          (a): class edible
	  3660   242    (b): class poisonous
```

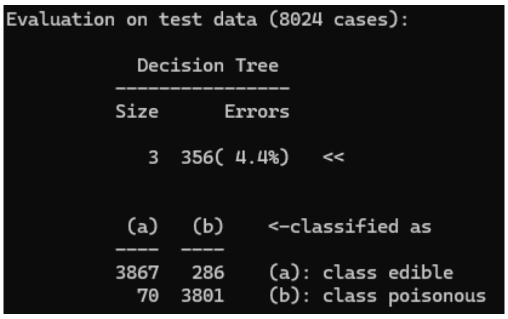
(Evaluation on training data) (Evaluation on test data)

After shuffling
After shuffling, the decision tree no longer solely relied on the “odor” attribute. The model started to incorporate other predictors, improving generalization and reducing the overfitting that occurred before shuffling (Fig 1.2).

The evaluation on the training data had 4% errors and the test data contained 4,4%. This shows slight increase in the (0%) an extreme improvement on the test data compared to before shuffling (45,6%).



```
Evaluation on training data (100 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3    4( 4.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	    53     2    (a): class edible
	     2    43    (b): class poisonous
```



```
Evaluation on test data (8024 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3  356( 4.4%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	  3867   286    (a): class edible
	    70  3801    (b): class poisonous
```

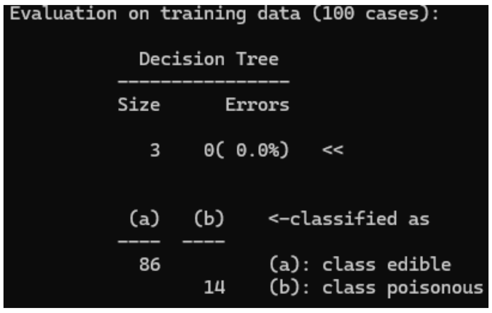
(Evaluation on training data) (Evaluation on test data)

2.2.2 Model with pruning

Before shuffling

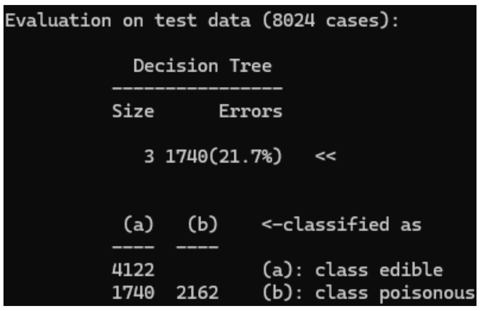
To reduce overfitting, we applied pruning with a confidence level of 0.01. This helped prevent the tree from growing too complex by removing branches that are not significant. The decision tree produced with pruning, became slightly more complex, and the test error dropped significantly from 45.6% to 21.7% (Fig 1.3)

It is also important to note that the addition of “gill-size” attribute as a second splitting criteria improved generalization and reduced the models reliance on the “odor” attribute from 100% to 25%.



```
Evaluation on training data (100 cases):

          Decision Tree
        ----------------
        Size      Errors

           3    0( 0.0%)   <<


         (a)   (b)    <-classified as
        ----  ----
          86          (a): class edible
                14    (b): class poisonous
```

(Training data, with pruning)



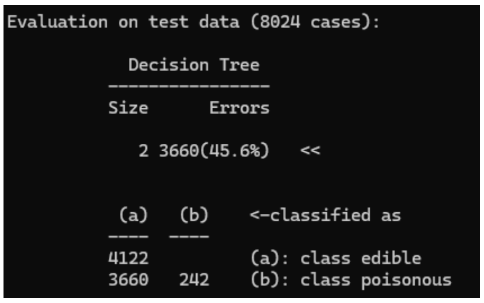
```
Evaluation on test data (8024 cases):

          Decision Tree
        ----------------
        Size      Errors

           3 1740(21.7%)   <<


         (a)   (b)    <-classified as
        ----  ----
        4122          (a): class edible
        1740  2162    (b): class poisonous
```

(Test data, with pruning)

<u>After shuffling</u>
By adjusting the confidence level, we observed a trade off between tree simplicity and generalization. We saw that as the confidence level increased we get a more accurate model (Fig 2) , and with 100% confidence level we achieve the most accurate model yet with 2,7% error on the test data (Fig 2.3).

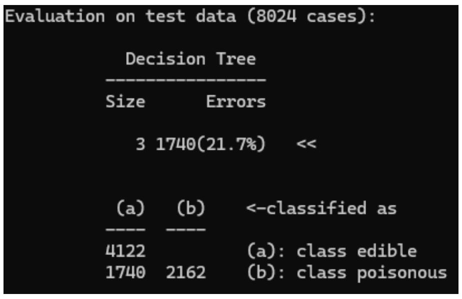
2.2.3 Boosting attempts

<u>Before shuffling</u>
Next we attempted to use boosting to improve the models accuracy by creating a group of trees. Boosting normally focuses on the errors made by the previous trees, however in our case boosting did not provide any improvements because the training data already was at 0%.



```
Evaluation on test data (8024 cases):

          Decision Tree
        ----------------
        Size      Errors

           2 3660(45.6%)   <<


         (a)   (b)    <-classified as
        ----  ----
        4122          (a): class edible
        3660   242    (b): class poisonous
```

(Boosting, without pruning)

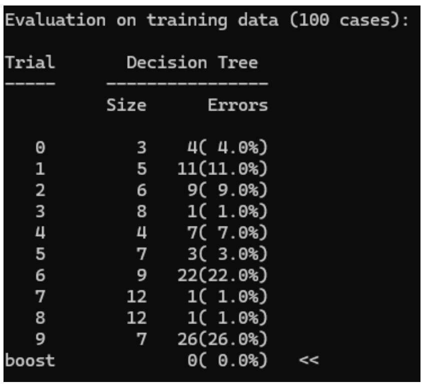


```
Evaluation on test data (8024 cases):

          Decision Tree
        ----------------
        Size      Errors

           3 1740(21.7%)   <<


         (a)   (b)    <-classified as
        ----  ----
        4122          (a): class edible
        1740  2162    (b): class poisonous
```

(Boosting, with pruning)

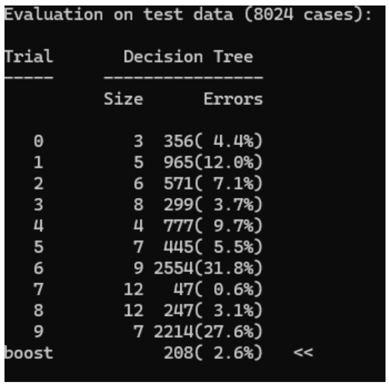
The results of our boosting attempts indicated that boosting has no effect in this scenario, likely because the model had 0% error on the training data, and no further gains could be made by focusing on correcting errors.

<u>After shuffling</u>
Boosting on the shuffled dataset caused improved generalization, with a drop in test error to 2,6%, from 4,4% in the baseline model. This is probably due to the multiple trees focusing harder to classify the instacers the initial tree misclassified.



```
Evaluation on training data (100 cases):

Trial         Decision Tree
-----       ----------------
            Size      Errors

   0           3     4( 4.0%)
   1           5    11(11.0%)
   2           6     9( 9.0%)
   3           8     1( 1.0%)
   4           4     7( 7.0%)
   5           7     3( 3.0%)
   6           9    22(22.0%)
   7          12     1( 1.0%)
   8          12     1( 1.0%)
   9           7    26(26.0%)
boost                0( 0.0%)   <<
```

(Boosting on training data)



```
Evaluation on test data (8024 cases):

Trial         Decision Tree
-----       ----------------
            Size      Errors

   0           3   356( 4.4%)
   1           5   965(12.0%)
   2           6   571( 7.1%)
   3           8   299( 3.7%)
   4           4   777( 9.7%)
   5           7   445( 5.5%)
   6           9  2554(31.8%)
   7          12    47( 0.6%)
   8          12   247( 3.1%)
   9           7  2214(27.6%)
boost              208( 2.6%)   <<
```

(Boosting on test data)

2.2.4 Classification

The dataset we chose contains two main outcomes, “edible” and “poisonous”. The consequences of misclassification could differ depending on the outcome:

- False positive: This would be classing an edible mushroom as poisonous. The cost of this is a missed opportunity, this is non lethal but still undesirable.
- False negative: Classifying a poisonous mushroom as edible would have a much more severe cost, and could in some situations be life-threatening

Minimizing false negatives is therefore the most critical, because of the consequences of someone consuming a poisonous moshroom are far larger than in the opposite case.

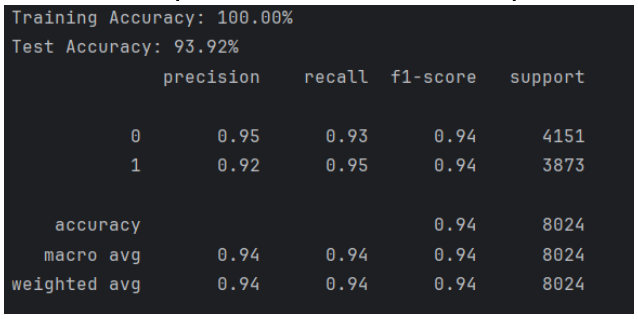
Apply cost matrix in c5.0
Unfortunately the C5.0 version on the ITstud server we were using did not seem to support the appliance of .cost files. We still created a file to demonstrate how it would have been included (Fig 3.1)

2.3 Modeling in Python

Github: https://github.com/mahojo99/mushroom-machine-learning-project

After initially working with the C5.0 algorithm, it was decided to explore Pythons scikit-learn library to determine whether further improvements could be made to the model’s performance. Using a combination of techniques such as Decision Trees, Random Forests, and instance weighting, the goal was to enhance the models accuracy and generalization.

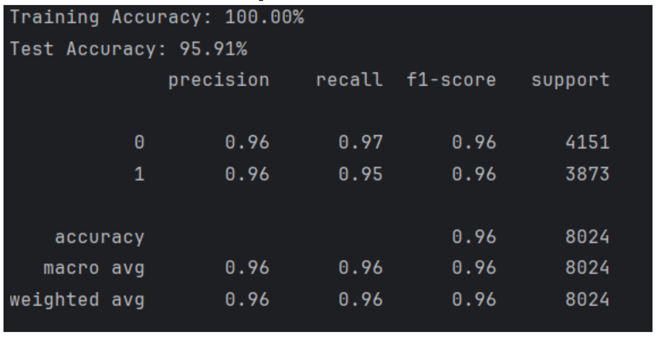
Base Model (Decision Tree Classifier);



```
Training Accuracy: 100.00%
Test Accuracy: 93.92%
              precision    recall  f1-score   support

           0       0.95      0.93      0.94      4151
           1       0.92      0.95      0.94      3873

    accuracy                           0.94      8024
   macro avg       0.94      0.94      0.94      8024
weighted avg       0.94      0.94      0.94      8024
```

A decision tree was used as the baseline model. While the model achieved high accuracy, decision trees can easily overfit smaller datasets, making this an important starting point for further improvement.

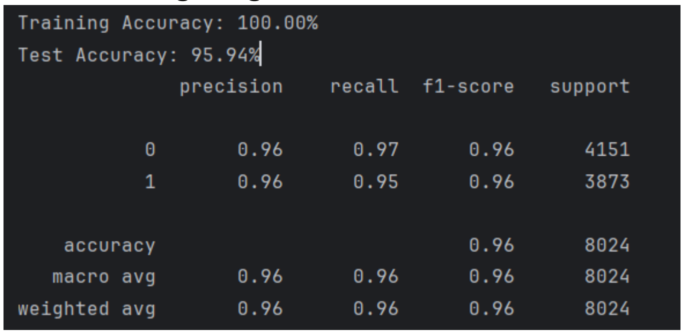
Random Forest Implementation:



```
Training Accuracy: 100.00%
Test Accuracy: 95.91%
              precision    recall  f1-score   support

           0       0.96      0.97      0.96      4151
           1       0.96      0.95      0.96      3873

    accuracy                           0.96      8024
   macro avg       0.96      0.96      0.96      8024
weighted avg       0.96      0.96      0.96      8024
```

Random forest was implemented to reduce overfitting and improve generalization. By averaging predictions across multiple trees, the random forest achieved better results compared to base decision tree although subtle given the already high accuracy.

Instance weighting:



```
Training Accuracy: 100.00%
Test Accuracy: 95.94%
              precision    recall  f1-score   support

           0       0.96      0.97      0.96      4151
           1       0.96      0.95      0.96      3873

    accuracy                           0.96      8024
   macro avg       0.96      0.96      0.96      8024
weighted avg       0.96      0.96      0.96      8024
```

Poisonous mushrooms were assigned a higher weight (2), making the model focus more on correctly identifying poisonous mushrooms, and therefore avoiding false negatives for this critical class.

Tree Pruning and Data Shuffling:
Tree Pruning was used to reduce overfitting and therefore the tree depth was limited. While Data Shuffling ensured that data was randomly distributed,
avoiding any bias from the original data order.

3. Discussion

The projects highlighted the challenges of dealing with an extremely dominant feature like "Odor", which caused significant overfitting in the C5.0 model. Shuffling the data improved generalization by forcing the model to use additional attributes. Pruning reduced overfitting by simplifying the model, though at a risk of losing important details. Boosting was only effective after shuffling, demonstrating how important preprocessing is in a project like this.

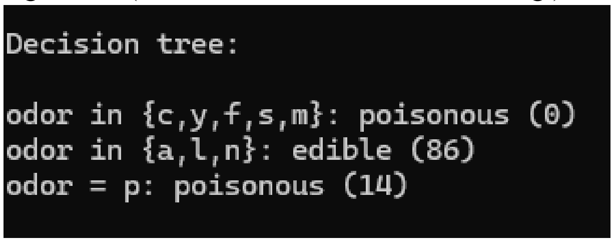
4. Conclusion

In the project decision trees and Random Forests to classify mushrooms as either edible or poisonous was successful. To start the C5.0 model overfit the data but after using techniques like shuffling, pruning, and boosting, the performance improved notably. The use of Random Forests and instance weighting also helped increase accuracy, especially in reducing the chances of misclassifying poisonous mushrooms. Future work would focus on better handling of cost-sensitive classification, particularly to reduce the false negative when classifying poisonous mushrooms.

Appendix

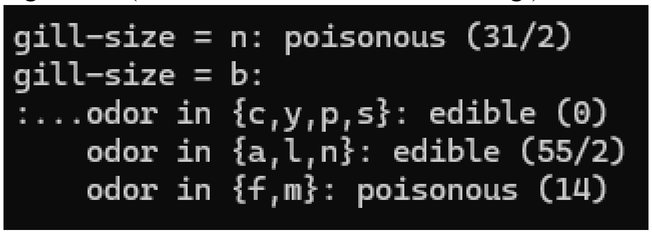
1 Decision trees

Figure 1.1 (Base decision tree before shuffling)



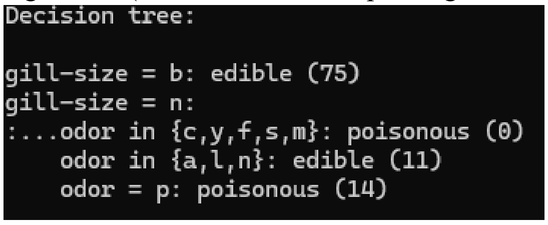
```
Decision tree:

odor in {c,y,f,s,m}: poisonous (0)
odor in {a,l,n}: edible (86)
odor = p: poisonous (14)
```

Figure 1.2 (Base decision tree after shuffling)



```
gill-size = n: poisonous (31/2)
gill-size = b:
:...odor in {c,y,p,s}: edible (0)
    odor in {a,l,n}: edible (55/2)
    odor in {f,m}: poisonous (14)
```

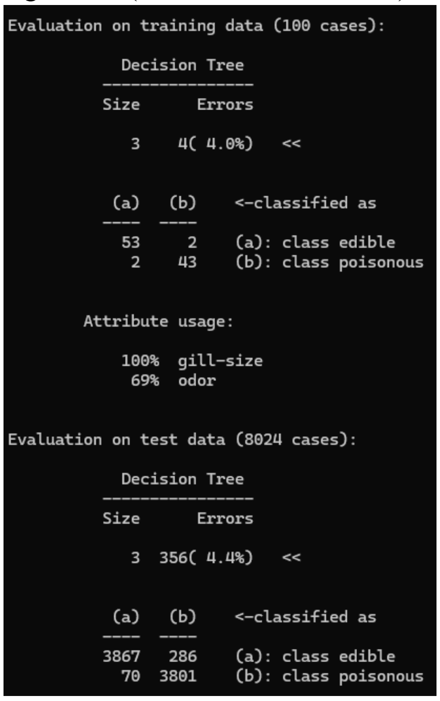
Figure 1.3 (Decision tree, with pruning before shuffling)



```
Decision tree:

gill-size = b: edible (75)
gill-size = n:
:...odor in {c,y,f,s,m}: poisonous (0)
    odor in {a,l,n}: edible (11)
    odor = p: poisonous (14)
```

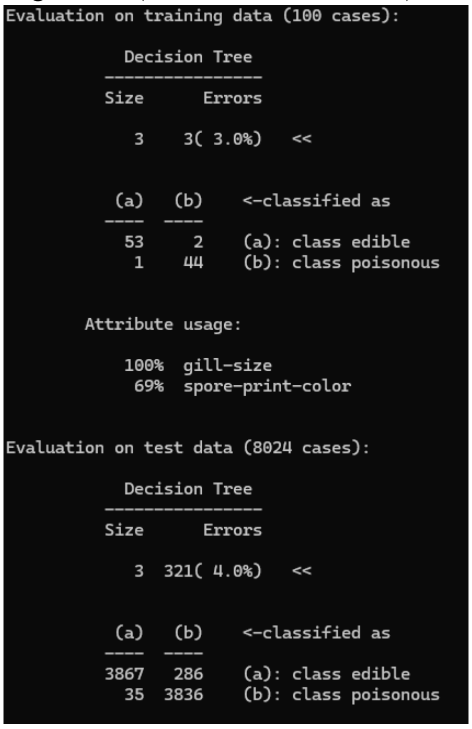
2. Pruning after shuffling

Figure 2.1 (Confidence level 20%)



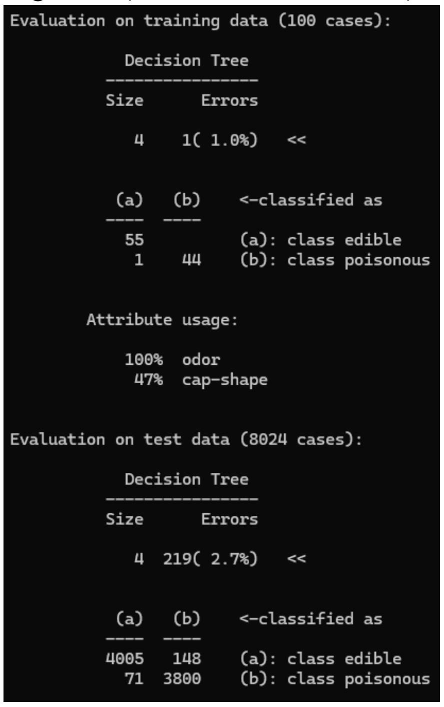
```
Evaluation on training data (100 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3    4( 4.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	    53     2    (a): class edible
	     2    43    (b): class poisonous


	Attribute usage:

	    100%  gill-size
	     69%  odor


Evaluation on test data (8024 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3  356( 4.4%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	  3867   286    (a): class edible
	    70  3801    (b): class poisonous
```

Figure 2.2 (Confidence level 50%)



```
Evaluation on training data (100 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3    3( 3.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	    53     2    (a): class edible
	     1    44    (b): class poisonous


	Attribute usage:

	    100%  gill-size
	     69%  spore-print-color


Evaluation on test data (8024 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     3  321( 4.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	  3867   286    (a): class edible
	    35  3836    (b): class poisonous
```

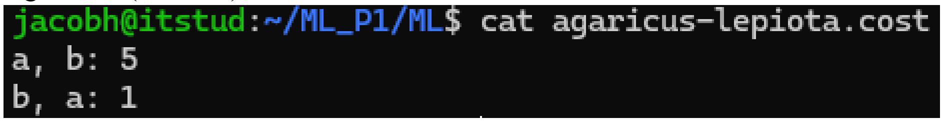
Figur 2.3 (Confidence level 100%)



```
Evaluation on training data (100 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     4    1( 1.0%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	    55          (a): class edible
	     1    44    (b): class poisonous


	Attribute usage:

	  100%  odor
	   47%  cap-shape


Evaluation on test data (8024 cases):

	    Decision Tree
	  ----------------
	  Size      Errors

	     4  219( 2.7%)   <<


	   (a)   (b)    <-classified as
	  ----  ----
	  4005   148    (a): class edible
	    71  3800    (b): class poisonous
```

3 Files

Figure 3.1 (cost file)



```
jacobh@itstud:~/ML_P1/ML$ cat agaricus-lepiota.cost
a, b: 5
b, a: 1
```

References

Andersen, Elise Rønnevig, and Anders Ihle Tovan (2024, 11. September). "Flere forgiftet av giftig sopp i Norge." *VG*, https://www.vg.no/nyheter/i/kw87Ov/flere-forgiftet-av-giftig-sopp-i-norge.

Khan, A Ameer Rashed, Dr S Shajun Nisha, and Dr M Mohamed Sathik. "CLUSTERING TECHNIQUES FOR MUSHROOM DATASET" 05, no. 06 (2018).

Ismail, Shuhaida, Amy Rosshaida Zainal, and Aida Mustapha. "Behavioural Features for Mushroom Classification." In *2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, 412–15, 2018. https://doi.org/10.1109/ISCAIE.2018.8405508.