GNR 638

Mini-Project I

Fine Grained Classification of Images Using Convolutional Neural Networks

Amruta Parulekar & Hemant Hajare

20d070009 20d070037

Indian Institute of Technology, Bombay

Contents:

1. Problem Statement

Train a CNN model with an upper limit of 10M parameters. Please download CUB dataset. Use default train-test split for this task. Submissions will be evaluated based on parameter efficiency, training time efficiency (no. of iterations) and accuracy. Submissions should include report, code and final model checkpoint. Drive link is fine for checkpoint. Report must include architecture and training details. Training loss and accuracy curves should also be incorporated along with final results. External models: Only ImageNet pretrained models are allowed. The correctness of the code, trade-off between accuracy and model complexity (no of parameters), and presentation of the results. You have to attach the log files.

2. Experimentation

a. Choosing a base model

LITERATURE SURVEY

CNN architecture	No. of params	Properties	Pretrained on Imagenet	Good choice for fine-grained classification
MobileNetV2	3.4M	Designed for mobile and embedded vision applications.	Yes	No
SqueezeNet	<1M	Achieves AlexNet-level accuracy with 50x fewer parameters	No	No
ShuffleNet	<10M	Designed to balance between accuracy and computational efficiency.	Yes	No
DenseNet-121	6.9M	It's relatively compact compared to deeper architectures like ResNet-50.	Yes	DenseNet's dense connectivity pattern allows for feature reuse and gradient flow throughout the network, making it effective at capturing fine details and subtle differences between classes. This dense connectivity can be

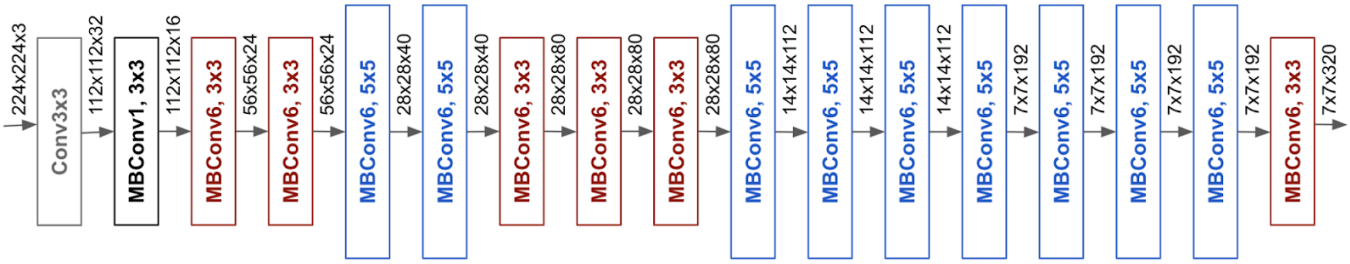
				particularly advantageous for fine-grained classification tasks, where distinguishing between similar object categories is crucial.
MobileNetV3	<5M	Further improves on MobileNetV2 with a focus on efficient and fast inference.	Yes	No
EfficientNet-B0	5.3M	Designed to achieve high accuracy with fewer parameters.	Yes	EfficientNet models are designed to achieve high accuracy with fewer parameters, making them efficient for both training and inference. The balance between model size and performance is crucial for fine-grained classification tasks, especially when computational resources are limited. EfficientNet-B0's optimization for efficiency could make it well-suited for such tasks.
VGG-11	9.1M	Relatively compact compared to deeper variants	Yes	No

EXPERIMENTS

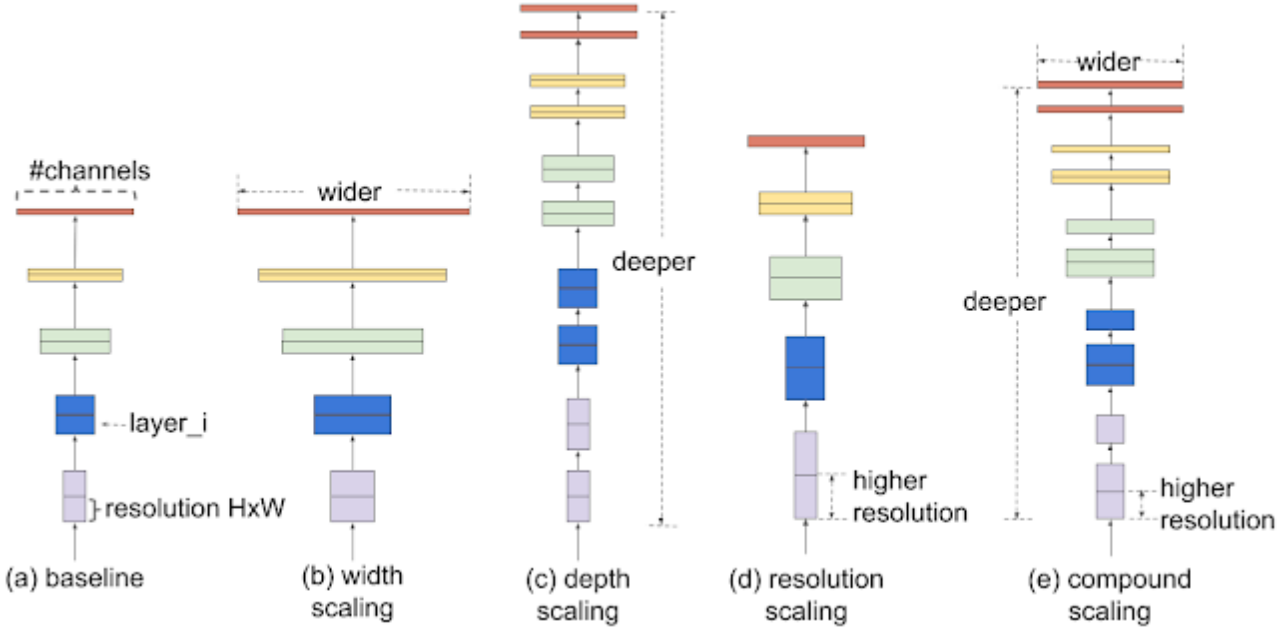
- **We first tried using Densenet121.**
- **However, we were not able to train it fast with the available computational resources.**
- **Hence, we switched to EfficientnetB0 and finetuned the entire model while initializing with imagenet pretrained weights.**
- **Then, to improve accuracy, we shifted to EfficientnetB2 as EfficientNet-B2 generally achieves higher accuracy compared to EfficientNet-B0, especially on large and complex datasets. The larger model size allows EfficientNet-B2 to capture more fine-grained features and learn more intricate patterns in the data.**

ARCHITECTURE FINALIZED

EfficientNet is a convolutional neural network (CNN) architecture that is designed to optimize the network's depth, width, and resolution simultaneously. The architecture uses a combination of neural architecture search and model scaling to achieve this.



It uses a simple yet highly effective *compound coefficient* to scale up CNNs in a more structured manner. Unlike conventional approaches that arbitrarily scale network dimensions, such as width, depth and resolution, this method uniformly scales each dimension with a fixed set of scaling coefficients.



b. Adding batch normalization

- In fine-grained classification, where distinguishing between visually similar categories is key, Batch Normalization enhances model performance.
- By normalizing activations within each layer, it mitigates the issue of internal covariate shift, ensuring stable optimization during training.
- This stability accelerates convergence and allows for higher learning rates, expediting the learning process.
- Moreover, BatchNorm acts as a form of regularization by adding noise to each mini-batch, thereby preventing overfitting, a common challenge in fine-grained tasks.
- It helps the model learn features that are more generalizable across different samples, enhancing its ability to generalize to unseen data.

Efficientnet already has batch normalized layers, which were used by us while training

c. Freezing layers

- Fine-tuning the entire model allows for learning task-specific features while leveraging pre-trained representations, which is beneficial for datasets with significant differences from the original one. However, this approach demands more computational resources and may risk overfitting with small datasets or highly dissimilar tasks.
- On the other hand, freezing layers preserves pre-trained features, aiding faster convergence and mitigating overfitting, particularly with limited data. While it may not capture task-specific features as effectively, freezing layers proves efficient when computational resources are constrained or when pre-trained features closely align with the new task.

We first tried experiments where we trained the entire model, and then we tried separate experiments where we froze all pretrained (Imagenet) layers except the last few blocks and finetuned the model. The initialized weights in both cases were from imagenet pretraining.

d. Adding Dropout

- Dropout regularization randomly deactivates neurons during training, encouraging the model to learn more robust features, essential for discerning subtle visual differences between categories.
- It introduces noise into the training process, preventing memorization of the training data and promoting better generalization to unseen examples. Additionally, dropout's ensembling effect aids in improving performance and prevents overfitting by averaging predictions from multiple subnetworks.

We tried experiments with 0.2, 0.5 and no dropout in the final fully connected layer of the model.

7. Model specification choices

a. Choosing an optimizer

The Adam optimizer's adaptive learning rate method dynamically adjusts learning rates for each parameter, leading to faster convergence and improved model performance. This is particularly beneficial in fine-grained tasks where subtle visual differences need to be discerned. Additionally, Adam uses momentum to navigate complex and high-dimensional parameter spaces more efficiently. This results in faster training and better generalization.

We used the Adam optimizer for all our experiments

b. Choosing a learning rate scheduler

A learning rate scheduler is crucial to optimize convergence, stability, generalization, and training efficiency by fine-tuning the learning rate. The straightforward and effective StepLR scheduler reduces the learning rate by a factor after a certain number of epochs. ReduceLROnPlateau dynamically adjusts the learning rate based on validation loss or accuracy to prevent the model from getting stuck in local minima or plateau.

We used the StepLR scheduler and ReduceLROnPlateau and observed better convergence as compared to without a scheduler(lr_init=0.001)

c. Choosing the activation function

ReLU is widely used and has been proven effective in various tasks, including fine-grained classification where subtle differences between classes are important. It's computationally efficient and helps in capturing non-linear relationships in the data. Leaky ReLU is used to combat the problem of dying ReLUs

We used the ReLU activation in the FC layer for all our experiments

d. Choosing batch size

For image classification, a batch size between 16 to 64 is commonly effective. Larger batch sizes leverage parallel processing for faster computation but may require more memory and could lead to overfitting with smaller datasets. Smaller batch sizes might generalize better but converge slower.

We experimented with batch size for a few epochs and chose the batch size of 64 as it gave good and fast convergence

e. Choosing a loss function

For multi-class classification, the categorical cross-entropy compares the predicted class probabilities with true class labels, penalizing deviations from the correct classification.

We used the categorical cross entropy loss in all experiments

f. Setting up early stopping

Early stopping prevents overfitting by monitoring the model's performance on a validation set and stopping training when performance starts to degrade, thus helping to find the optimal balance between model complexity and generalization.

We set up our model to stop training if validation accuracy did not improve for 3 consecutive epochs.

8. Results

a. Architecture experiment

EfficientnetB0 – WithoutDropout - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – NoFrozenBlocks

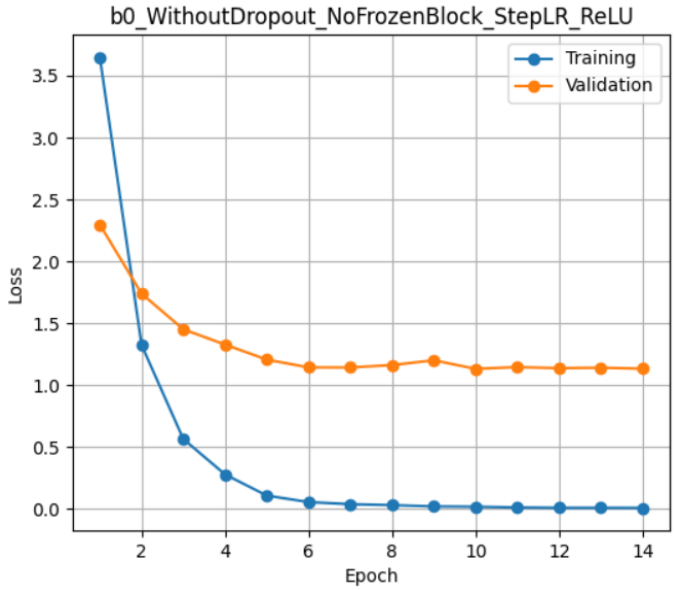
Testing Accuracy: 73.37%

Trainable Parameters: 4263748

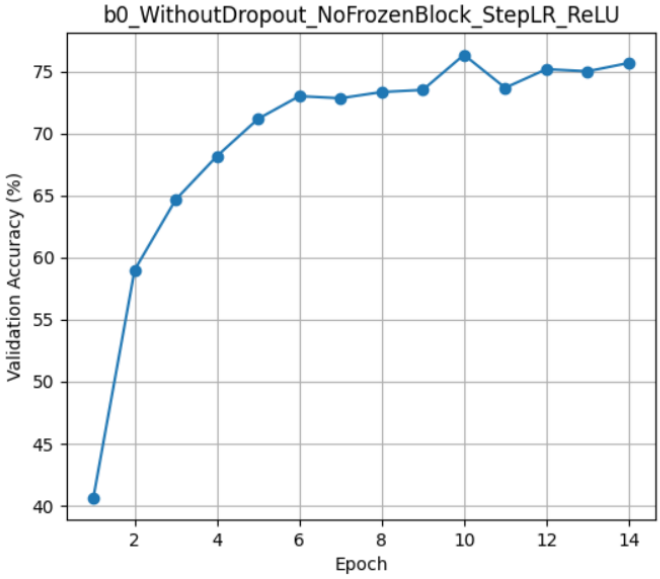
Total Parameters: 4263748

Epochs required:14

Loss plots:



Validation accuracy plot



EfficientnetB2 – WithoutDropout - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – NoFrozenBlocks

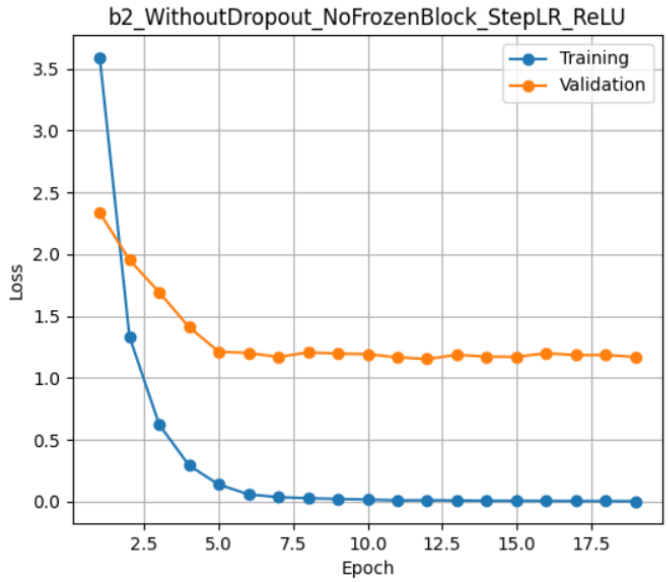
Testing Accuracy: 73.92%

Trainable Parameters: 7982794

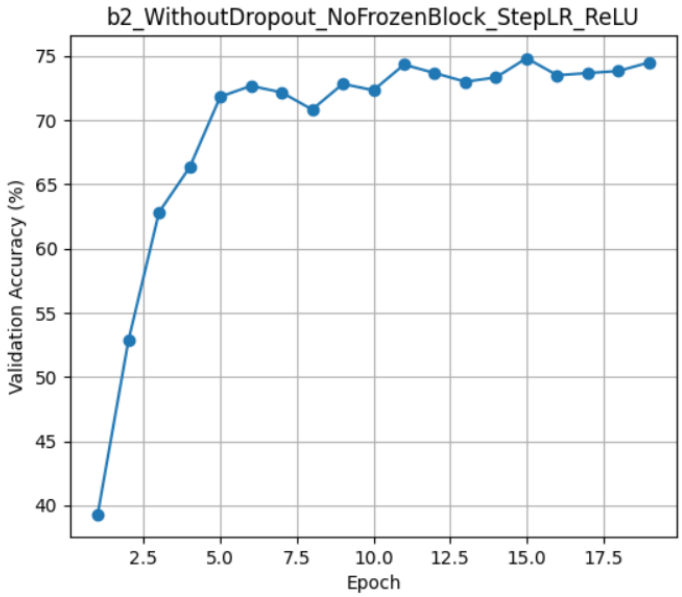
Total parameters: 7982794

Epochs required: 19

Loss plots:



Validation accuracy plot:



b. Layer Freezing experiment

EfficientnetB0 – WithoutDropout - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

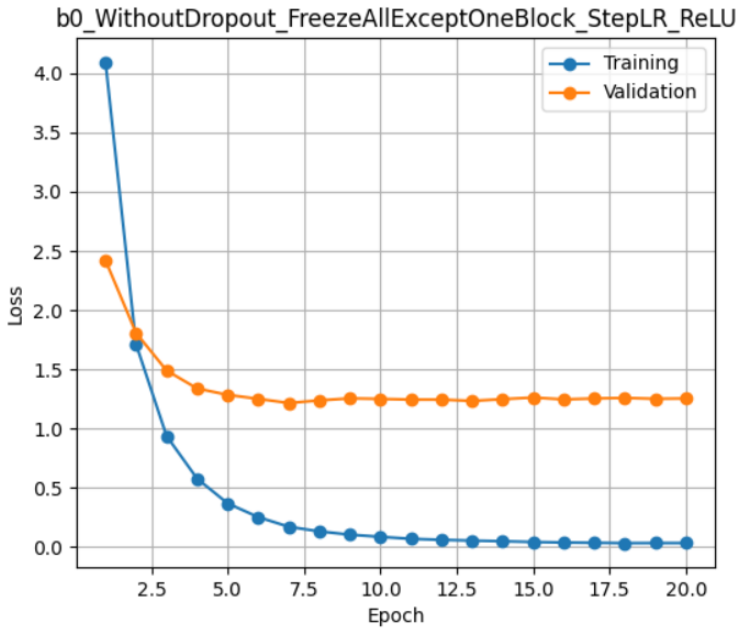
Testing Accuracy: 70.76%

Trainable Parameters: 1010200

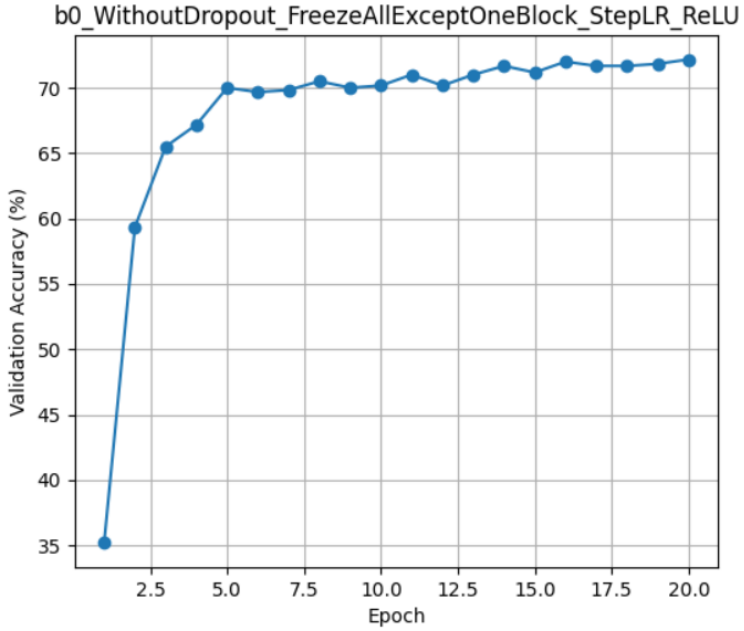
Total Parameters: 4263748

Epochs required: 20

Loss plots:



Validation accuracy plot:



EfficientnetB2 – WithoutDropout - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

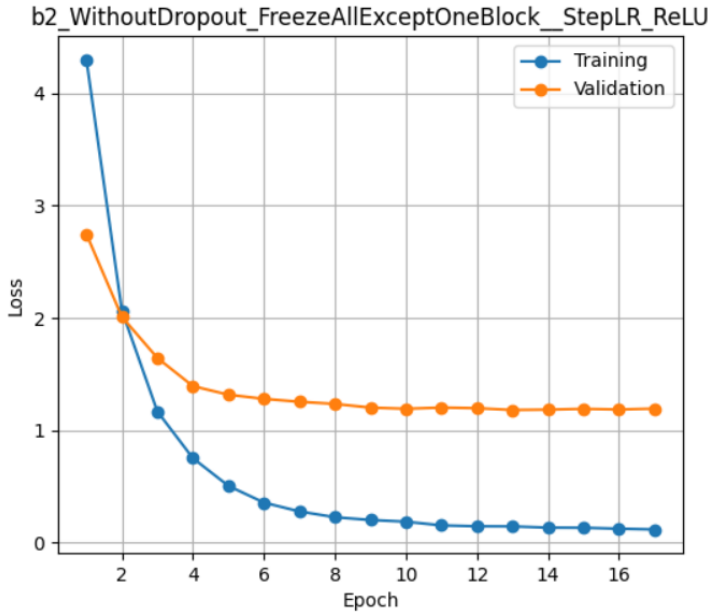
Testing Accuracy: 69.8%

Trainable Parameters: 2229136

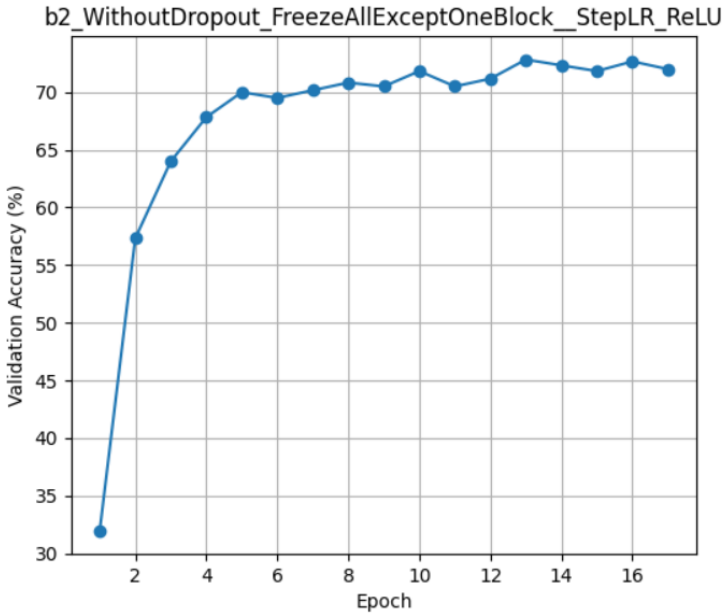
Total Parameters: 7982794

Epochs required: 17

Loss plots:



Validation accuracy plot:



c. Dropout Experiment

EfficientnetB2 – WithDropout0.5 - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

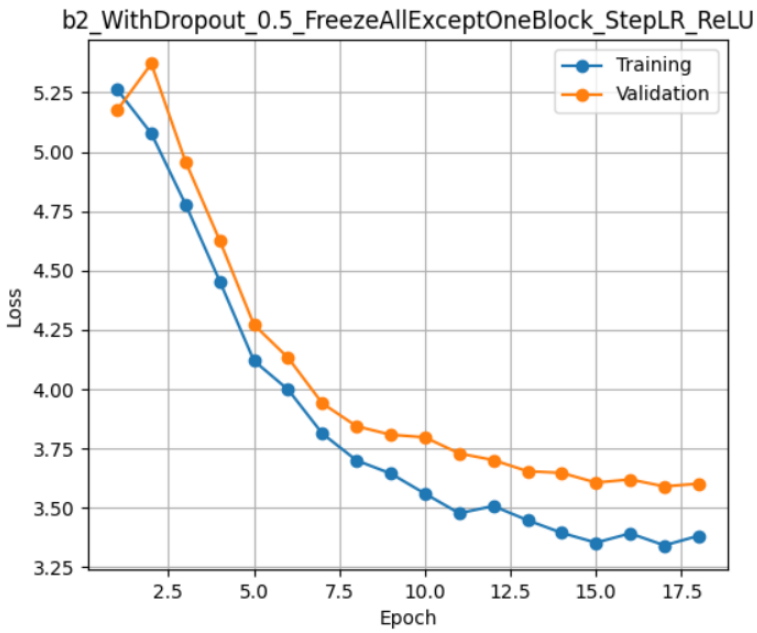
Testing Accuracy: 54.69%

Trainable Parameters: 2229136

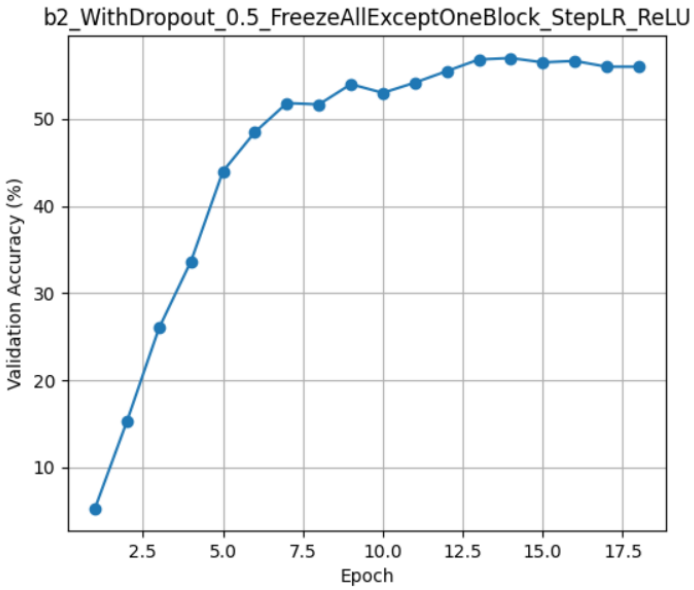
Total Parameters: 7982794

Epochs required: 18

Loss plots:



Validation accuracy plot



EfficientnetB2 – WithDropout0.2 - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

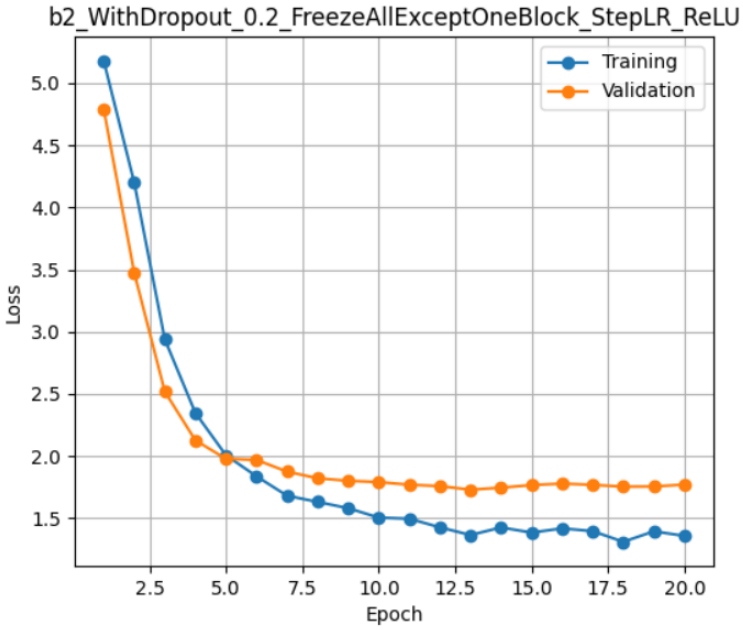
Testing Accuracy: 69.14%

Trainable Parameters: 2229136

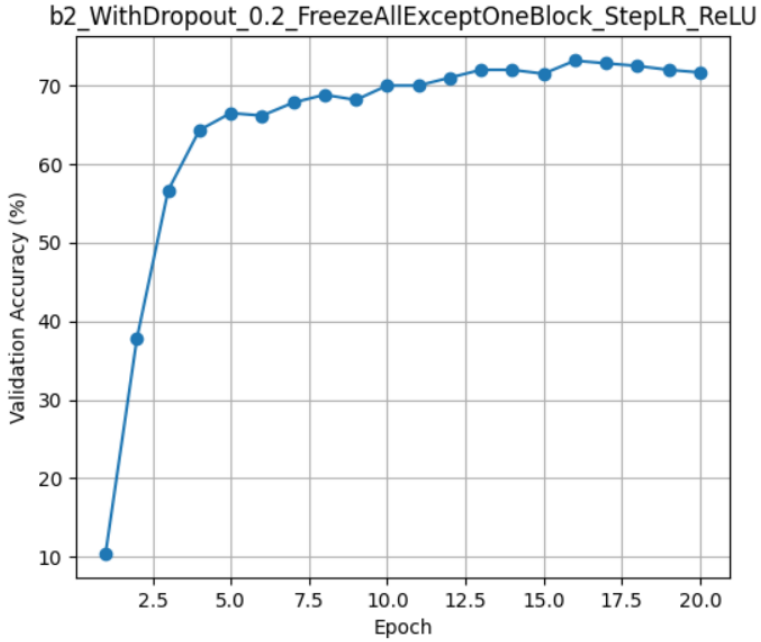
Total Parameters: 7982794

Epochs required: 20

Loss plots:



Validation accuracy plot



d. LR Scheduler experiment

EfficientnetB0 – WithoutDropout - WithBatchNorm – ReduceLROnPlateauscheduler – WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

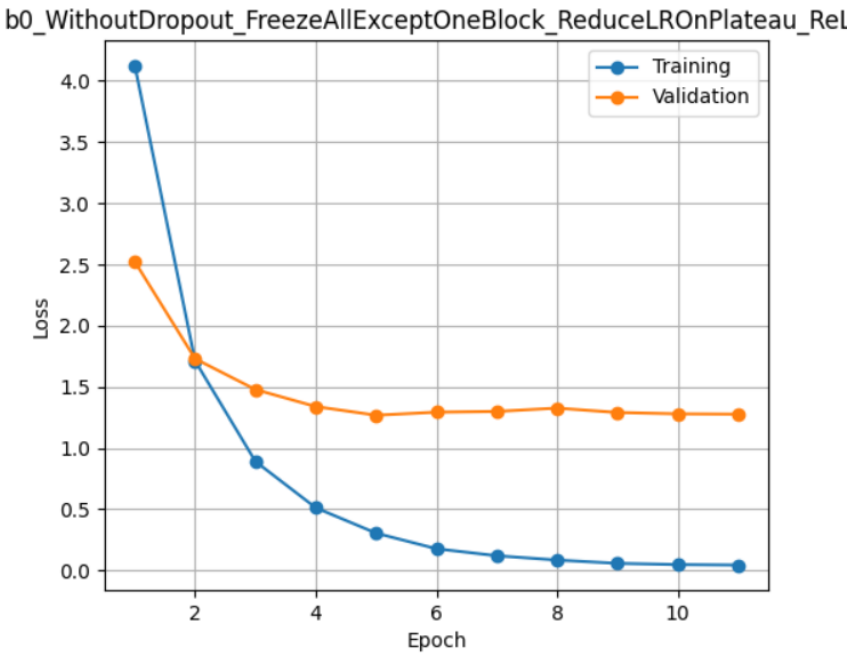
Testing Accuracy: 69.61%

Trainable Parameters: 1010200

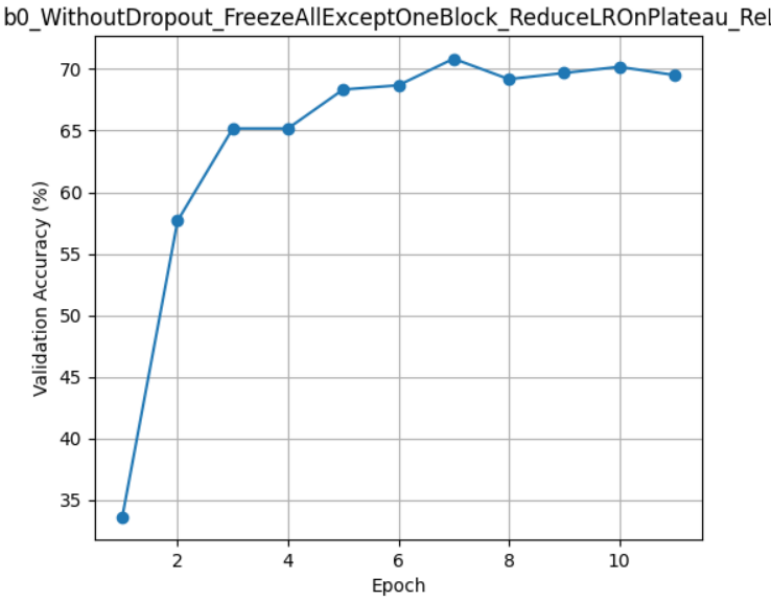
Total Parameters: 4263748

Epochs required: 11

Loss plots:



Validation accuracy plot:



EfficientnetB0 – WithoutDropout - WithBatchNorm – WithoutLRScheduler– WithEarlyStopping – CCELoss – Batchsize64 – ReLUFC – Adam – FreezeAllExceptOneBlock

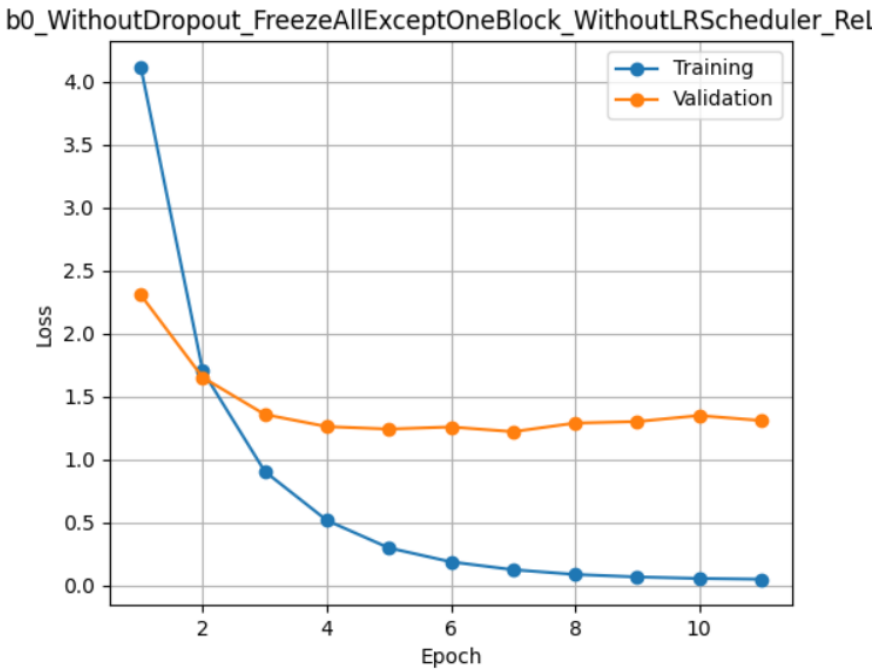
Testing Accuracy: 69.09%

Trainable Parameters: 1010200

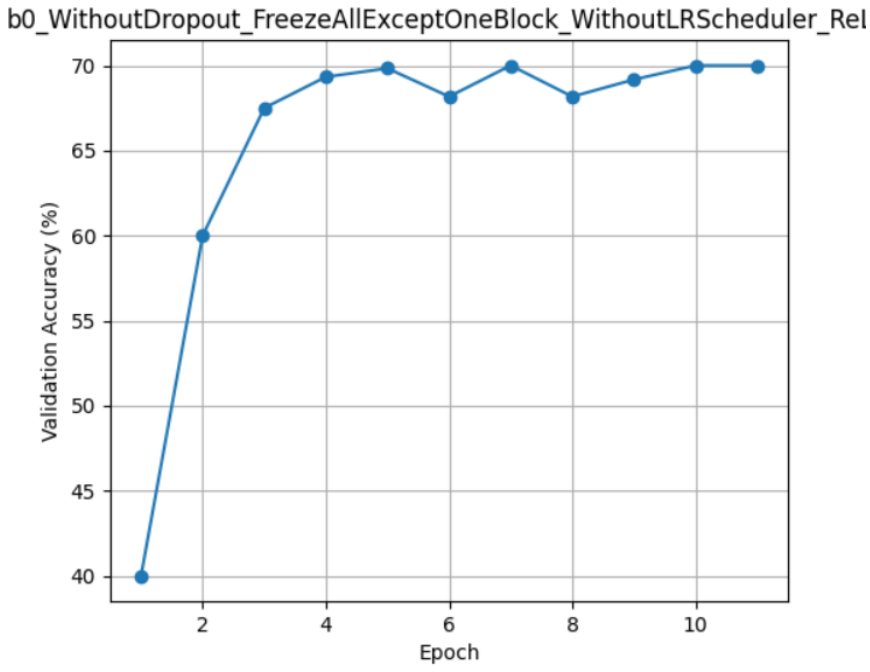
Total Parameters: 4263748

Epochs required: 11

Loss plots:



Validation accuracy plot:



e. Activation function experiment

EfficientnetB0 – WithDropout0.2 - WithBatchNorm – StepLRscheduler – WithEarlyStopping – CCELoss – Batchsize64 – LeakyReLUFC – Adam – FreezeAllExceptOneBlock

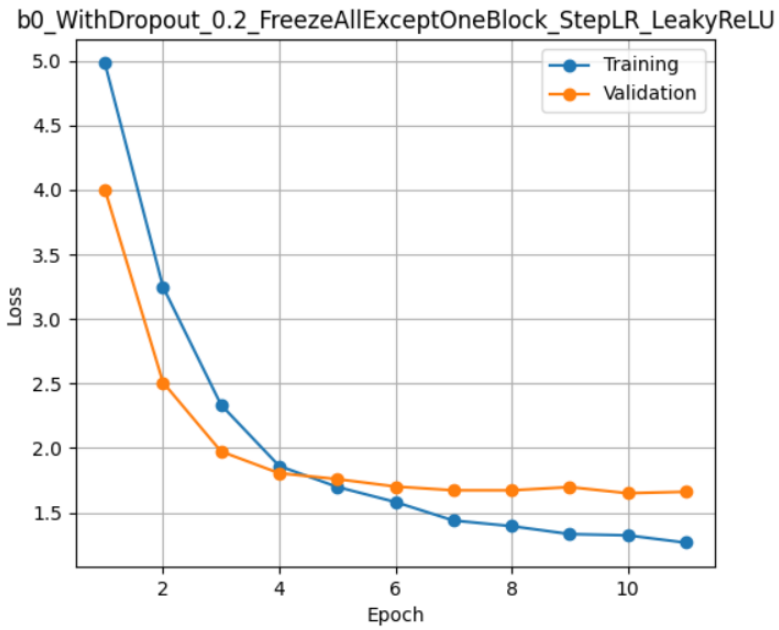
Testing Accuracy: 68.62%

Trainable Parameters: 1010200

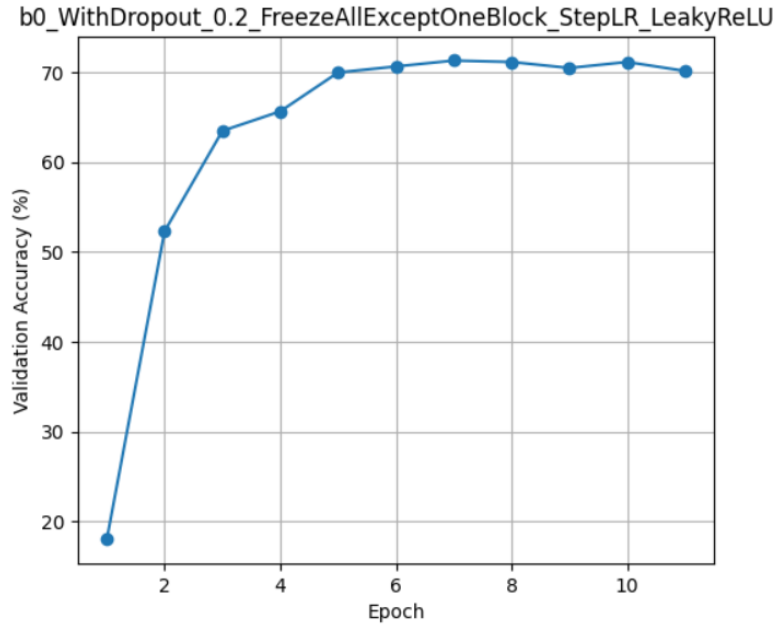
Total Parameters: 4263748

Epochs required: 11

Loss plots:



Validation accuracy plot:



9. Conclusion (Best model, Parameter accuracy tradeoff)

Best accuracy without dropout
Dropout may not have helped due to the complexity of the model, insufficient regularization, or a lack of overfitting.

Best accuracy with batch normalization
Batch normalization helped by reducing internal covariate shift and accelerating convergence by stabilizing the training process.

Best accuracy with StepLR scheduler
StepLR scheduler helped by dynamically adjusting the learning rate at predefined epochs, facilitating faster convergence and preventing oscillations. Plateau LR performed worse than StepLR due to its sensitivity to small fluctuations in loss, potentially leading to premature convergence or oscillations in learning rate.

Best accuracy with early stopping
Early stopping helped by preventing overfitting and improving generalization by halting training when the model's performance on the validation set started to deteriorate.

Best accuracy with categorical cross entropy loss
Categorical cross entropy helped by quantifying the difference between predicted and actual class distributions, optimizing model parameters to minimize this discrepancy during training.

Best accuracy with batch size 64
Batch size 64 was the best due to its balance between computational efficiency and generalization ability, leveraging sufficient batch diversity without overwhelming memory resources.

Best accuracy with ReLU activation in the fully connected layer
Since there weren't outliers, leaky Rely didn't give benefits. It is also slower than ReLU

Best accuracy with Adam optimizer
Adam performed best due to its adaptive learning rate and momentum, efficiently navigating complex optimization landscapes and facilitating rapid convergence.

Best results- Trainable parameter and testing accuracy tradeoff

Model	Trainable params	Testing Accuracy
EfficientnetB0 - Full	4263748	73.37%
EfficientnetB2 - Full	7982784	73.92%
EfficientnetB0 – All except last block frozen	1010200	70.76%

Link to all models and logs : https://drive.google.com/drive/folders/1phTqhcbXiT-V9Rx4KdOh5HBTWWZ1MTU5?usp=sharing

10. Appendix

a. More about model layers and freezing strategy

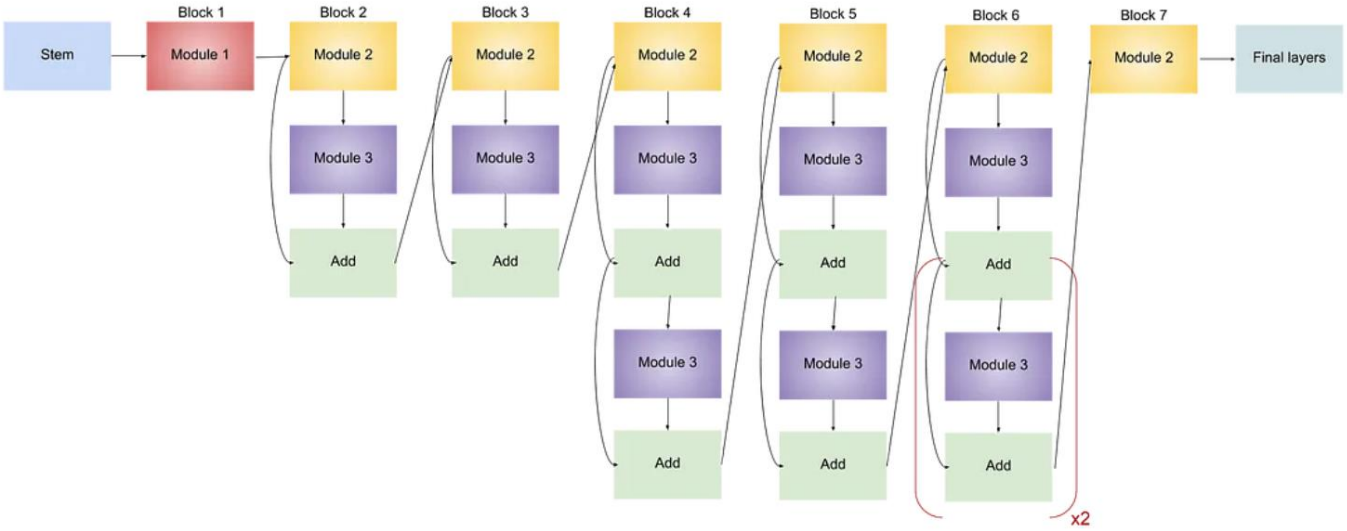


Fig A: EfficientnetB0

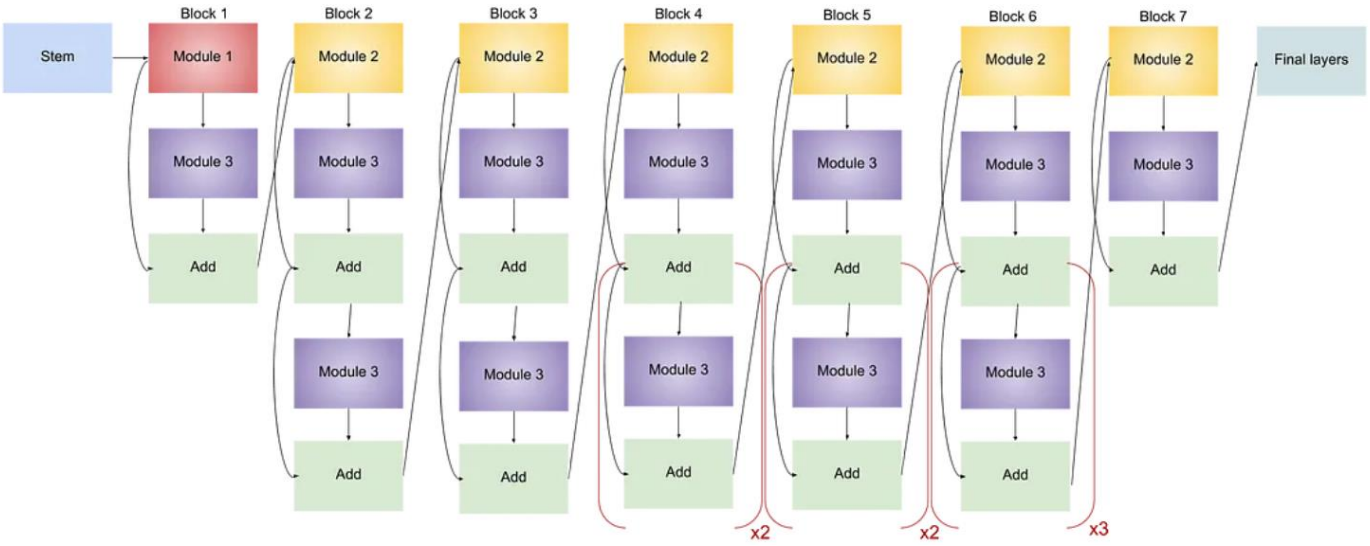


Fig B: EfficientnetB2

For model freezing, first, we froze all parameters of the model and we selectively unfroze specific parameters in the model, including the last few layers of the model's blocks, fully connected layers, and batch normalization layers. Different variations of layer freezing was tried out.

b.Code snippets

Code to freeze layers:

```
# Freeze all the wights of the model
for name, param in model.named_parameters():
    param.requires_grad = False
# Un-freeze the last block, fully connected layer and batch normalization layers
for name, param in model.named_parameters():
    if (name.split('.')[0] == "_blocks" and int(name.split('.')[1]) >= len(model._blocks) - 1) or ("fc" in name) or ("bn" in name) :
        param.requires_grad = True
```

Code to add dropout:

```
model._fc = nn.Sequential(
    nn.Linear(model._fc.in_features, 200),  # Add a fully connected layer
    nn.ReLU(inplace=True),  # Add activation function
    nn.Dropout(0.2),  # Add dropout
    )
```

Code to test:

```
# Testing loop
def test(model, test_loader, device):
    model.eval()
    test_correct = 0
    test_total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)  # Move data to CUDA if available
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            _, predicted = torch.max(outputs, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()
    test_accuracy = 100 * test_correct / test_total
    print(f'Testing Accuracy: {test_accuracy:.2f}%, Epochs: {epoch}')
    f.write(f'Testing Accuracy: {test_accuracy:.2f}%, Epochs: {epoch}\n')
```

Code to load data:

```
# Dataset path
dataset = datasets.ImageFolder(root='/kaggle/input/cub-200-2011/CUB_200_2011/images', transform=transform)

train_dataset, val_dataset, test_dataset = test_train_val_split(dataset)

# Dataloaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Code for loss function, optimizer and LR scheduler:

```
# Define loss function, optimizer and learning rate scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = StepLR(optimizer, step_size=1,gamma=0.9)
```

Code for early stopping:

```
# Stoping criteria
if(val_accuracy > max_val_accuracy):
    max_val_accuracy = val_accuracy
    torch.save(model.state_dict(), '/kaggle/working//efficientnet_b0_cub.pth')
    stop_count = 0
else:
    stop_count += 1
```

Code for validation loop(inside training loop):

```
# Validation loop
model.eval()
val_correct = 0
val_total = 0
with torch.no_grad():
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)  # Move data to CUDA if available
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        val_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        val_total += labels.size(0)
        val_correct += (predicted == labels).sum().item()

val_loss = val_loss/len(val_loader)
VALIDATION_LOSS.append(val_loss)
```

Code for training loop(excluding validation part):

```
# Training loop
for epoch in range(num_epochs):
    if(stop_count > 3 or epoch == num_epochs):
        break

    model.train()
    for batch_idx, (inputs, labels) in enumerate(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()

        if (batch_idx + 1) % print_every == 0 or (batch_idx + 1) == total_batches:
            avg_loss = running_loss / print_every
            print(f'Training Epoch - [{epoch+1}/{num_epochs}], Batch [{batch_idx+1}/{total_batches}], Loss: {avg_loss:.4f}')
            f.write(f'Training Epoch - [{epoch+1}/{num_epochs}], Batch [{batch_idx+1}/{total_batches}], Loss: {avg_loss:.4f}\n')
            epoch_loss += running_loss
            running_loss = 0
    epoch_loss = epoch_loss / len(train_loader)
    TRAINING_LOSS.append(epoch_loss)

    print(f'Epoch Loss: {epoch_loss:.4f}')
    f.write(f'Epoch Loss: {epoch_loss:.4f}\n')

    epoch_loss = 0
```