Classification of the CIFAR-10 dataset using MobileNetV3

DANIEL MAGUIRE (23222425) AND MALIK E ASHTAR (23044004)

TABLE OF CONTENTS

1. INTRODUCTION

Image classification is a classical problem which intersects the fields of image processing, computer vision and machine learning. In this paper we study image classification using deep learning. We use a convolutional neural network with the MobileNetV3 [1] architecture for this purpose.

In the realm of computer vision, image classification stands as a cornerstone task, aiming to categorize images into predefined classes with the highest accuracy possible. This project explores the application of the MobileNetV3 architecture, a state-of-the-art neural network designed for efficient performance on mobile and embedded devices, to the well-known CIFAR-10 dataset [2]. The CIFAR-10 dataset comprises 60,000 32x32 color images distributed across 10 classes, including animals and vehicles, making it a standard benchmark for evaluating image classification models.

Our approach leverages the lightweight and computationally efficient nature of MobileNetV3, which incorporates advanced techniques such as depthwise, squeeze-and-excitation blocks, and hard-swish activation functions. These innovations are designed to reduce model size and computational demands without compromising the model's accuracy, making MobileNetV3 an ideal candidate for real-time image classification tasks, even on devices with limited computational resources.

Throughout the project, we meticulously train and validate the MobileNetV3 model on the CIFAR-10 dataset, employing a series of data augmentation techniques to enhance the model's generalization capabilities We meticulously monitor key performance metrics , including training and validation loss and accuracy, as well as precision and recall, to ensure a comprehensive evaluation of the model's performance.

Additionally, we delve into visualizing key attributes of the model's learning process, such as the change of loss and accuracy over epochs, confusion matrices, and class activation maps. These visualizations not only provide deeper insights into the model's behaviour but also highlight areas for further optimization.

2. METHODOLOGY

VISUALIZATION OF SOME OF THE KEY ATTRIBUTES

The CIFAR-10 dataset, consisting of 60,000 32x32 color images in 10 classes, presents a diverse set of images for classification tasks. The show_batch function visualizes a batch of pre-processed images, showcasing the effect of applied transformations such as resizing, normalization, and potential augmentation like random flips and rotations. This visualization helps in understanding the dataset's complexity and the preprocessing steps' impact on the input data fed into the MobileNetV3 model.

FEATURE SELECTION, ENGINEERING

In the context of deep learning and specifically with CNNs like MobileNetV3, explicit feature selection is not performed as the model is capable of learning the most relevant features for the task directly from the raw pixel data. However, feature engineering through data augmentation techniques plays a crucial role. The train_transform includes resizing images to 224x224 to match the input layer size of MobileNetV3, random horizontal flipping, and random rotation by up to 10 degrees. These augmentations introduce variability into the training data, aiding the model in learning more generalized features and improving its robustness to variations in unseen data.

```
def load_split_train_test(datadir, valid_size=.2, batch_size=64):
    #Transformations for training data
    #Resizing to 224x224 to match the required size of the input layer of the model
    train_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        #Randomly flip images horizontally
        transforms.RandomHorizontalFlip(),
        #Randomly rotate images by up to 10 degrees
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

Listing 1. load_split_train_test function

CORRELATION

Correlation analysis typically applies to datasets with numerical or categorical features to identify relationships between variables. For image data like CIFAR-10, the concept of correlation is less straightforward. However, understanding inter-class variations and similarities can be insightful. For instance, distinguishing between vehicles (cars and trucks) and animals (cats and dogs) might involve learning correlated features like shapes and textures specific to each category.

3. DATA PRE-PROCESSING

The data pre-processing steps are crucial for preparing the dataset for efficient training with the MobileNetV3 model:

Resizing: Images are resized to 224x224 pixels, aligning with the input size requirement of the MobileNetV3 architecture. This ensures that the model receives input data in a consistent format.

Normalization: Pixel values are normalized using predefined mean and standard deviation values across the RGB channels. This step is vital for stabilizing the learning process by ensuring that the input data has a mean of 0 and a standard deviation of 1, facilitating faster convergence.

Data Augmentation: The training data undergoes augmentation, including random horizontal flips and rotations. These transformations enhance the diversity of the training set, promoting model generalization by simulating various perspectives and orientations.

Splitting Dataset: The CIFAR-10 dataset is split into training, validation, and test sets, with a specified proportion (e.g., 30% for validation as per the valid_size parameter). This separation allows for effective model training, validation during the training process to monitor *overfitting*, and final evaluation on the test set to assess the model's performance on unseen data.

```
batch_size = 64
train_loader, valid_loader, test_loader = load_split_train_test('./data', valid_size=0.3, batch_size=batch_size)
```

Listing 2. Setting batch size and Splitting Data

Data Loading: DataLoader objects are used to load the data in batches, enabling efficient memory usage and parallel processing. The use of num_workers helps in accelerating data loading by utilizing multiple subprocesses.

4. MODEL ARCHITECTURE

We utilized the MobileNetV3 architecture, known for its balance between efficiency and accuracy. Key innovations that contribute to this architectures highly efficient nature include the use of depthwise separable convolutions, which reduces the number of parameters and computational complexity, and the introduction of novel activation functions: HardSigmoid and HardSwish. Another key innovation in the MobileNetV3 architecture is the use of the Squeeze-and-Excitation (SE) block and Linear Bottlenecks. The MobileNetV3 architecture is also separated into MobileNetV3-Large and MobileNetV3-Small, the key difference being the size of the network. Figure 1 [1] shows the structure of both MobileNetV3-Large and MobileNetV3-Small. Separating in this way allows to further tailor the version of MobileNetV3 for specific use cases.

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

MobileNetV3-Large

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

MobileNetV3-Small

Figure 1. MobileNetV3-Large architecture (Above) and MobileNetV3-Small (Below)

DEPTHWISE SEPARABLE CONVOLUTIONS

The Depthwise seperable convolution is an important part of the MobileNet family architecture and was introduced in the first iteration of MobileNet [3]. Compared to a standard convolution, the depthwise seperable convolution does not combine the input channels. Instead it performs convolution on each channel separately and then the pointwise convolution combines the outputs of the depthwise convolution across channels. This is visually demonstrated in figure 2 [4].

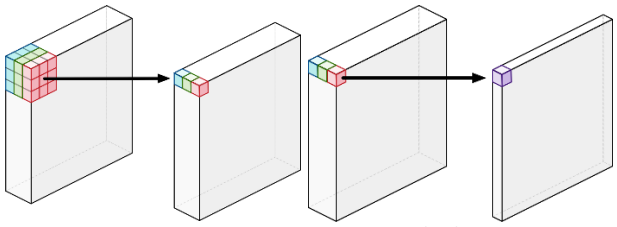

Figure 2. The depth wise convolution applied (left), followed by a pointwise convolution (right) to combine the outputs.

HARD SIGMOID ACTIVATION FUNCTION

```
class HardSigmoid(nn.Module):
    def __init__(self):
        super(HardSigmoid, self).__init__()

    def forward(self, x):
        #Applies the Hard Sigmoid activation
        return F.relu6(x + 3.) / 6.
```

Listing 3. HardSigmoid Implementation

The HardSigmoid function provides a computationally efficient approximation of the sigmoid function, crucial for environments with limited computational resources. A comparision of the hard sigmoid with the standard sigmoid can be seen below in figure. 3 [1].

HARDSWISH ACTIVATION FUNCTION

```
class HardSwish(nn.Module):
    def __init__(self):
        super(HardSwish, self).__init__()

    def forward(self, x):
        #Applies Hard Swish activation
        return x * F.relu6(x + 3.) / 6.
```

Listing 4. HardSwish Implementation

HardSwish, an approximation of the Swish function, combines linear and non-linear properties, allowing the network to learn complex patterns efficiently [1].

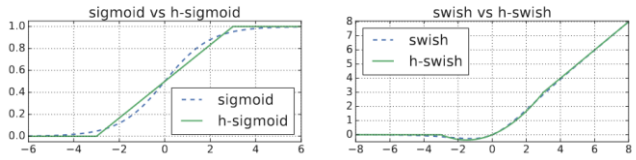

Figure 3. Comparison of the sigmoid to the hard sigmoid (Left) and comparison of the swish to the hard swish (right)

SQUEEZE AND EXCITATION BLOCK

The Squeeze and Excitation (SE) block is a neural network module designed to improve the representational capacity of a network by dynamically recalibrating channel-wise feature responses. The SE block performs two main operations: squeeze, which aggregates the global spatial information of each channel into a single number, and excitation, which captures channel-wise dependencies and scales the feature channels accordingly.

```
class SqueezeExcite(nn.Module):
    def __init__(self, exp_size, divide=4):
        super(SqueezeExcite, self).__init__()

        self.fc_R = nn.Sequential(
            nn.Linear(exp_size, exp_size // divide, bias=False),
            nn.ReLU()
        )

        self.fc_HS = nn.Sequential(
            nn.Linear(exp_size // divide, exp_size, bias=False),
            HardSigmoid()
        )

    def forward(self, x):
        batch, channels, height, width = x.size()
        #Global average pooling to squeeze spatial dimensions
        out = F.avg_pool2d(x, kernel_size=[height, width]).view(batch, -1)

        #Two fully connected layers with ReLU and Hard Sigmoid activations,
        out = self.fc_R(out)
        out = self.fc_HS(out)

        #Reshape and scale the input feature map
        out = out.view(batch, channels, 1, 1)
        return out * x
```

Listing 5. Squeeze and Excite Implementation

The SE block first applies global average pooling to generate channel-wise statistics. These statistics are then processed through two fully connected layers with ReLU and Hard Sigmoid activations, respectively, to capture channel-wise dependencies. The output is used to scale the original feature maps, enhancing informative features while suppressing less useful ones. [5]

BOTTLENECK BLOCK

The BottleNeck block, a fundamental building block of the MobileNetV3 architecture, is designed for efficient computation. First introduced to the MobileNet family in MobileNetV2 [6], it incorporates depthwise separable convolutions and optionally integrates the SE block for enhanced feature recalibration.

```
class BottleNeck(nn.Module):
    def __init__(self, inp, out, exp, kernal_size, NL, SE, stride):
        super(BottleNeck, self).__init__()
        self.SE = SE

        #Padding calculation for depthwise convolution
        padding = 0
        if kernal_size == 3:
            padding = 1
        elif kernal_size == 5:
            padding = 2
        else:
            assert False, "Unsupported kernel size"

        #Activation function selection based on NL argument
        if NL == "ReLu":
            activation = nn.ReLU
        elif NL == "h_swish":
            activation = HardSwish
        else:
            assert False, "Unsupported non-linearity"

        #First 1x1 convolution for expanding the number of channels
        self.conv1 = nn.Sequential(
            nn.Conv2d(inp, exp, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(exp),
            activation()
        )

    #Depthwise convolution for spatial filtering
    self.dconv1 = nn.Sequential(
        nn.Conv2d(exp, exp, kernel_size=kernal_size, stride=stride, padding=padding, groups=exp, bias=False),
        nn.BatchNorm2d(exp),
    )

    #Squeeze-and-Excitation block, if SE is True
    self.squeeze = SqueezeExcite(exp) if SE else nn.Identity()

    #Second 1x1 convolution for reducing the number of channels to the desired output size
    self.conv2 = nn.Sequential(
        nn.Conv2d(exp, out, kernel_size=1, stride=1, padding=0, bias=False),
        nn.BatchNorm2d(out),
        activation()
    )

    #Connection flag for choosing whether to use residual connections
    self.connect_flag = (stride == 1 and inp == out)

def forward(self, x):
    x2 = self.conv1(x)
    x2 = self.dconv1(x2)

    if self.SE:
        x2 = self.squeeze(x2)

    x2 = self.conv2(x2)

    #Apply residual connection if needed
    if self.connect_flag:
        return x + x2
    else:
        return x2
```

Listing 6. BottleNeck Block Implementation

The BottleNeck block utilizes an initial 1x1 convolution to expand the number of channels, followed by a depthwise convolution for spatial filtering. If enabled, an SE block is applied to recalibrate the feature maps. A final 1x1 convolution reduces the channel dimensions, and a residual connection is applied if the input and output dimensions match.

5. IMPLEMENTATION OF MOBILENETV3 ARCHITECTURE

The MobileNetV3 model is a culmination of several architectural advancements aimed at optimizing computational efficiency while trying to minimally compromise accuracy.

It incorporates lightweight depthwise separable convolutions, HardSwish and HardSigmoid activation functions, and the novel addition of Squeeze and Excitation (SE) blocks. These elements are strategically combined to enhance the model's ability to capture important features while minimizing computational overhead.

INITIAL CONVOLUTION LAYER

The model begins with an initial convolution layer that reduces the spatial dimensions of the input image and increases the depth to 16 channels. This layer uses a 3x3 kernel, stride of 2, and padding of 1, followed by batch normalization and HardSwish activation:

```
self.conv1 = nn.Sequential(
        nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(16),
        HardSwish(),
)
```

Listing 7. Initial layers

BOTTLENECK BLOCKS, SQUEEZE AND EXCITATION BLOCKS

The SE blocks within the Bottleneck layers serve to recalibrate the channel-wise feature responses, allowing the model to focus on the most informative features. The MobileNetV3 architecture dynamically constructs its network using a series of bottleneck blocks, each meticulously designed with specific parameters including input/output channel sizes, expansion ratios, kernel sizes, choice of nonlinear activation functions, incorporation of Squeeze and Excite blocks, and stride values for effective down sampling.

```
self.body = []

if large == True:
    print("Large model\n")
    layers = [
        [16, 16, 16, 3, "ReLu", False, 1],
        [16, 24, 64, 3, "ReLu", False, 2],
        [24, 24, 72, 3, "ReLu", False, 1],
        [24, 40, 72, 5, "ReLu", True, 2],
        [40, 40, 120, 5, "ReLu", True, 1],
        [40, 40, 120, 5, "ReLu", True, 1],
        [40, 80, 240, 3, "h_swish", False, 2],
        [80, 80, 200, 3, "h_swish", False, 1],
        [80, 80, 184, 3, "h_swish", False, 1],
        [80, 80, 184, 3, "h_swish", False, 1],
        [80, 112, 480, 3, "h_swish", True, 1],
        [112, 112, 672, 3, "h_swish", True, 1],
        [112, 160, 672, 5, "h_swish", True, 1],
        [160, 160, 960, 5, "h_swish", True, 2],
        [160, 160, 960, 5, "h_swish", True, 1],
    ]

    for inp, out, exp, kernal_size, NL, SE, stride in layers:
        self.body.append(BottleNeck(inp, out, exp, kernal_size, NL, SE, stride))
```

Listing 8. MobileNetV3-Large layer configuration

Final Convolution and Classification Layers

After processing through the Bottleneck blocks, the feature maps undergo further convolution before being passed to a global average pooling layer. The final classification is performed by a 1x1 convolution that reduces the channel depth to the number of output classes, followed by a dropout layer for regularization which can be seen in listing 9 below.

```
self.pconv3 = nn.Sequential(
    nn.Conv2d(960, 1280, kernel_size=1, stride=1),
    HardSwish(),
    nn.Dropout(p=dropout_rate),
    nn.Conv2d(1280, self.num_classes, kernel_size=1, stride=1),
)
```

Listing 9. Final layers

6. Training Procedure

The model was trained using the Adam optimizer with a learning rate of 0.001 and a step learning rate scheduler to adjust the learning rate dynamically. Cross-Entropy Loss was used as the loss function, suitable for multi-class classification tasks.

```
def train(model, train_loader, valid_loader, epochs):
    #Define loss function, CrossEntropyLoss for classification
    criterion = nn.CrossEntropyLoss()
    #Define optimiser
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    #Learning rate scheduler
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.03)

    #Initialise lists to keep track of training and validation losses and accuracies
    train_losses, valid_losses = [], []
    train_accuracies, valid_accuracies = [], []
```

Listing 10. Train function with Adam Optimizer and Cross-Entropy Loss

7. Preparing Images for Prediction

To ensure the model accurately interprets new images, they must first be processed to match the training data's format. This involves resizing images to 224x224 pixels, converting them to tensors, and normalizing their pixel values. The following code snippet outlines the transformation steps applied to each image before prediction.

```
def predict(path, model):
    #Transformations to be applied to the input images
    #This includes resizing the image to 224x224, converting it to a tensor
    test_transforms = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

Listing 11. Resize, normalize and convert to a tensor

Predicting Image Classes

The predict function facilitates the classification of images stored in a specified directory. It iterates over each image, applies the necessary transformations, and forwards the processed image through the trained MobileNetV3 model to obtain a prediction. The function is designed to handle images in batches, enhancing efficiency and scalability. Key components of the predict function include:

Disabling Gradient Computation: For inference, gradient computation is unnecessary and thus disabled to optimize performance.

Processing Each Image: Images are loaded, preprocessed, and batched before being passed to the model.

```
with torch.no_grad():
    #Loop over all images in the specified path with a
    for im_path in glob.glob(join(path, '*.jpg')):
        #Open image file
        image = Image.open(im_path)
        if image.mode != "RGB":
            #Skip images that are not in RGB format
            continue

        #Apply transforms and convert to float
        image_tensor = test_transforms(image).float()
        #Add batch dimension
        image_tensor = image_tensor.unsqueeze_(0)
        #Move the tensor to device (hopefully GPU)
        image_tensor = image_tensor.to(device)
```

Listing 12. For loop to process each image

8. Model Evaluation Setup

The test function is designed to assess the model's performance on the CIFAR-10 test dataset. It operates in the following steps:

Model Preparation: The model is set to evaluation mode, which disables dropout and batch normalization layers allows for experimentation with different configurations to optimize performance

```
def test(model, test_loader):
    all_preds = []
    all_true = []
    #Set model to evaluation mode
    #Disables dropout and batch normalisation layers
    model.eval()
```

Listing 13. Train function with Adam Optimizer and Cross-Entropy Loss

Batch Processing: The test dataset is processed in batches, allowing for efficient computation and evaluation.

Prediction and Label Collection: For each batch, the model predicts the class of each image. These predictions, along with the true labels, are stored for further analysis.

```
    _, predicted = torch.max(outputs.data, 1)

    all_preds.extend(predicted.cpu().numpy())
    all_true.extend(labels.cpu().numpy())
```

Listing 14. Train function with Adam Optimizer and Cross-Entropy Loss

Accuracy Calculation: The total number of correct predictions is compared to the total number of labels to calculate the model's accuracy.

```
#Update total count of labels
total += labels.size(0)
#Compare predicted and true labels and update
correct += (predicted == labels).sum().item()
```

Listing 15. Train function with Adam Optimizer and Cross-Entropy Loss

Precision, Recall, and Accuracy Computation

After collecting all predictions and true labels, we compute precision, recall, and accuracy using the precision_score, recall_score, and accuracy_score functions from the sklearn.metrics module. These metrics are calculated as follows:

Precision: Measures the accuracy of positive predictions.

Recall: Measures the ability of the model to find all the positive samples

Accuracy: Measures the fraction of predictions our model got right.

Function to train and test MobileNetV3

The train_test_mobilenetV3 function encapsulates the entire process, from model initialization through to the final evaluation. This function serves as a streamlined approach to experimenting with different dropout rates and model configurations, facilitating a comprehensive analysis of the MobileNetV3 architecture's performance on the CIFAR-10 dataset.

```
def train_test_mobilenetV3(dropout_rate, epochs, large=True):
    #Init model with dropout rate, and set model size
    model = mobilenet_v3(dropout_rate=dropout_rate, large=large).to(device)

    #Train model
    _, metrics = train(model, train_loader, valid_loader, epochs)
    train_losses, valid_losses, train_accuracies, valid_accuracies = metrics
    #Test model
    test(model, test_loader)
    #Plot results
    title = f"Dropout {dropout_rate}, Epochs {epochs}"
    plot_results(train_losses, valid_losses, train_accuracies, valid_accuracies, title)

    #Predictions and true labels from validation set
    all_preds, all_labels = get_all_preds_and_labels(valid_loader, model, device)
    #Calculate the confusion matrix
    cm = confusion_matrix(all_labels, all_preds)

    #CIFAR-10 dataset class names
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
    #Plot confusion matrix
    plot_confusion_matrix(cm, class_names)
```

Listing 16. Train function with Adam Optimizer and Cross-Entropy Loss

9. MobileNetV3 Small Model Performance

Optimal dropout rate that we found was 0.7 and the number of epochs is 4. If we add more epochs and we start to see early signs of overfitting.

```
train_test_mobilenetV3(dropout_rate=0.7, epochs=4, large=False)
```

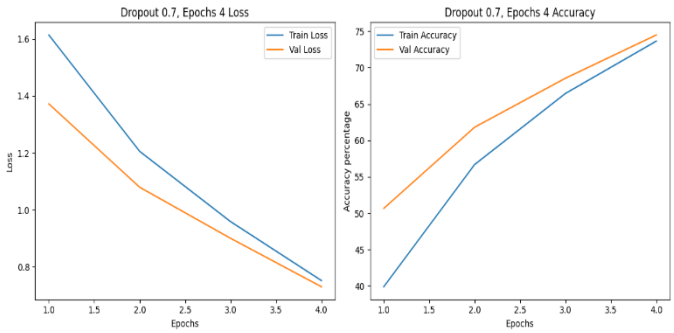

Figure 4. MobileNetV3-Small loss and accuracy when dropout=0.7 and epochs=4

The training process revealed a consistent improvement in both training and validation accuracy over the 4 epochs, with a notable decrease in loss.

Testing Results

Upon evaluating the model on the test dataset, the following metrics were observed:

```
Precision: 0.7458877999762664
Recall: 0.7464999999999999
Test Accuracy: 0.7465
```

The precision and recall metrics, both aligning at 0.746, indicate a balanced performance across the CIFAR-10 classes. The test accuracy closely mirrors the validation accuracy, suggesting that the model has generalized well to unseen data.

Impact of varying a hyperparameter(s) on MobileNetV3 Small Model

In an effort to explore the limits of the MobileNetV3 small model's performance on the CIFAR-10 dataset, we extended the training duration to 5 epochs, maintaining the previously optimized dropout rate of 0.7. This section examines the impact of the additional epoch on model performance, particularly focusing on signs of overfitting.

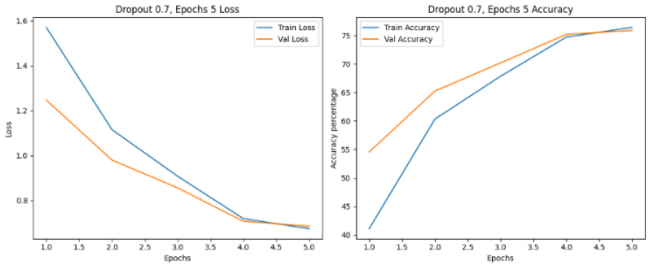

Figure 5. MobileNetV3-Small loss and accuracy when dropout=0.7 and epochs=5

The additional epoch of training led to a slight increase in both precision and recall, suggesting that the model continued to learn useful patterns from the training data. However, the minimal improvement in validation accuracy compared to the increase in training accuracy signals the beginning stages of overfitting. Overfitting occurs when a model learns patterns specific to the training data, which do not generalize well to unseen data, potentially compromising the model's real-world applicability.

10. Analyzing Overfitting and Underfitting (Small Model)

Here we aim to deliberately engineer overfitting and underfitting scenarios in the MobileNetV3 small model trained on the CIFAR-10 dataset. By manipulating the dropout rate and the number of training epochs, we observe the model's behavior under conditions prone to overfitting and underfitting. This analysis provides insights into the model's capacity to generalize and the effectiveness of regularization techniques.

Engineered Overfitting

Here we are setting the dropout_rate to 0 and increase the epochs to engineer overfitting. Dropout is a regularisation technique which turns off a set number of neurons during the training process. This helps the model generalise better, so turning it off by setting the dropout rate to 0 means that we can use this as a method to force the model to overfit the data. Increasing the number of epochs also helps further encourage overfitting as it allows the model to have more iterations to fit to the data.

```
train_test_mobilenetV3(dropout_rate=0.0, epochs=10, large=False)
```

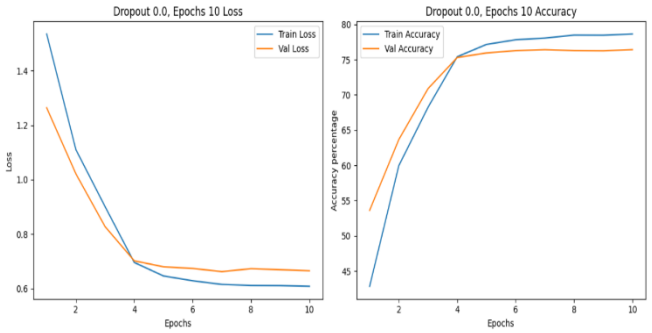

Figure 6. MobileNetV3-Small loss and accuracy when dropout=0 and epochs=10

Observations

- Initial training shows steady improvement in both training and validation accuracy.
- From epochs 6 to 10, training accuracy continues to slightly increase, while validation accuracy plateaus, indicating the onset of overfitting.
- The minimal gap between training and validation loss starts to get wider, suggesting the model is memorizing the training data.

Testing Results

```
Precision: 0.7675353432051456
Recall: 0.7676000000000001
Test Accuracy: 0.7676
```

These metrics, while high, are achieved at the cost of the model's ability to generalize, as evidenced by the stagnation of validation accuracy.

Engineered Underfitting

Here we are decreasing the number of epochs and increasing the dropout rate to engineer underfitting. By decreasing the number of epochs we are essentially cutting the model off before it has the time to learn the data. Setting the dropout rate to be very high can also help encourage underfitting as we are turning off such a large amount of neurons that the network struggles to effectively learn.

```
train_test_mobilenetV3(dropout_rate=0.9, epochs=3, large=False)
```

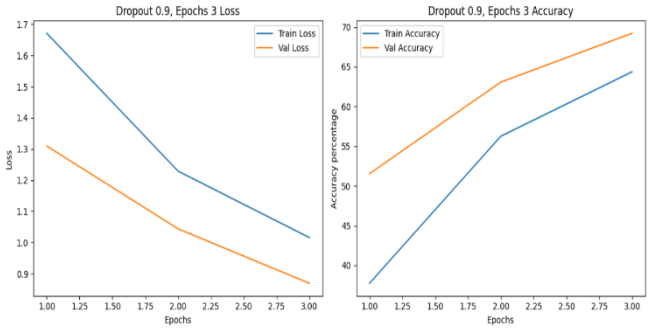

Figure 7. MobileNetV3-Small loss and accuracy when dropout=0.9 and epochs=3

Observations

- The model shows an inability to learn effectively from the training data, indicated by higher training and validation losses compared to the overfitting scenario.
- The premature cessation of training prevents the model from reaching its potential accuracy on both the training and validation sets.

Testing Results

```
Precision: 0.6941715863943863
Recall: 0.693
Test Accuracy: 0.6930
```

These metrics reflect the model's underperformance, primarily due to insufficient training and excessive regularization, hindering its learning capacity.

11. MobileNetV3 Large Model Performance

Following the exploration of overfitting and underfitting scenarios with the MobileNetV3 small model, we extend our investigation to the large model variant. This experiment aims to assess how the increased complexity and capacity of the large model influence its susceptibility to overfitting and underfitting under identical hyperparameter settings used for the small model.

```
train_test_mobilenetV3(dropout_rate=0.7, epochs=4, large=True)
```

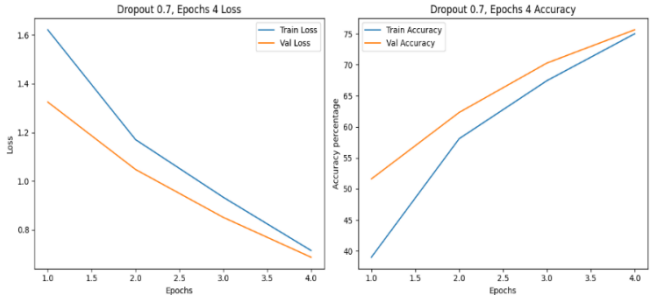

Figure 8. MobileNetV3-Large loss and accuracy when dropout=0.7 and epochs=4

Similar to the small model, the large variant shows consistent improvement across epochs, with significant gains in both training and validation accuracy.

Testing Results

```
Precision: 0.7624977661099234
Recall: 0.7618
Test Accuracy: 0.7618
```

These results indicate a slight improvement in precision, recall, and test accuracy compared to the small model, suggesting that the increased model size contributes positively to the model's ability to generalize.

12. Impact of varying a hyperparameters on the MobileNetV3 Large Model

Following the investigation into the MobileNetV3 small model's behavior under engineered overfitting and underfitting conditions, we extend our analysis to the large model variant. This experiments goal was to understand the impact of model size on the effectiveness of similar hyperparameter adjustments, specifically focusing on dropout rate and training duration.

Here we examine the impact of the additional epoch on model performance, particularly focusing on signs of overfitting.

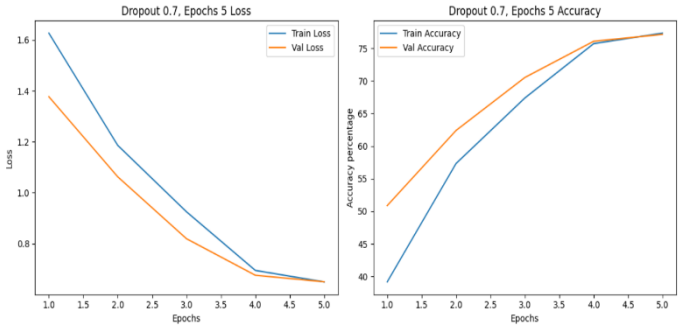

Figure 9. MobileNetV3-Large loss and accuracy when dropout=0.7 and epochs=5

Testing Results

```
Precision: 0.7726209060673617
Recall: 0.7745999999999998
Test Accuracy: 0.7746
```

Adding an additional epoch further enhances the model's performance, with increased precision, recall, and test accuracy. Unlike the small model, the large variant shows continued improvement without immediate signs of overfitting, suggesting that the larger model capacity can effectively utilize the additional training epoch. This could be due to the models larger size better being able to represent the complexity of the data compared to the smaller model.

Engineering Overfitting in MobileNetV3-Large

To explore the effects of overfitting in deep learning models, we conducted an experiment with the MobileNetV3 large model on the CIFAR-10 dataset. By setting the dropout rate to 0 and increasing the number of training epochs to 10, we aimed to create conditions conducive to overfitting

```
train_test_mobilenetV3(dropout_rate=0, epochs=10, large=True)
```

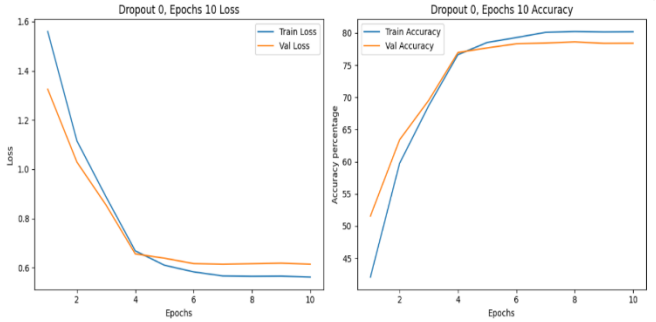

Figure 10. MobileNetV3-Large loss and accuracy when dropout=0 and epochs=10

Observations

- Initial epochs showed rapid improvements in both training and validation accuracy, with training loss decreasing more sharply than validation loss.
- By the mid-point of the training (Epoch 5 onwards), while training accuracy continued to improve, validation accuracy gains became marginal, suggesting the onset of overfitting.
- Despite slight fluctuations in validation loss and accuracy in the later epochs, the model's training accuracy remained consistently high, further indicating overfitting.

Testing Results

```
Precision: 0.7871469448523931
Recall: 0.7882999999999999
Test Accuracy: 0.7883
```

These results demonstrate a high level of performance on the test set, with precision and recall closely aligned, indicating balanced class-wise performance. However, the

close tracking of training and validation accuracy, along with the minimal dropout, suggests the model may be beginning to overfit to the training data.

Engineering Underfitting

In contrast to our previous experiment on overfitting, we now turn our attention to deliberately engineering underfitting within the MobileNetV3 large model trained on the CIFAR-10 dataset. By significantly increasing the dropout rate and reducing the number of training epochs, we aim to examine the model's performance under conditions that limit its learning capacity.

This setup is designed to severely restrict the model's ability to learn from the training data, simulating an underfitting scenario.

```
train_test_mobilenetV3(dropout_rate=0.9, epochs=3, large=True)
```

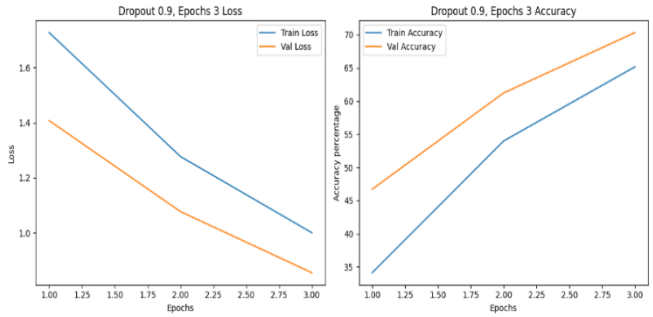

Figure 11. MobileNetV3-Large loss and accuracy when dropout=0.9 and epochs=3

Observations

- The high dropout rate led to a substantial loss of information during training, as reflected in the relatively high training and validation losses across epochs.
- Despite some improvement over the three epochs, both training and validation accuracies remained significantly lower than in the overfitting scenario, indicating the model's struggle to adequately learn the dataset's underlying patterns.

Testing Results

```
Precision: 0.7105702294239662
Recall: 0.7094
Test Accuracy: 0.7094
```

These metrics, while respectable, are indicative of the model's underperformance due to the engineered constraints.

Analysis

The experiment with the MobileNetV3 large model under conditions conducive to underfitting yielded several key insights:

- The significant dropout rate effectively hampered the model's learning process, as evidenced by the lower accuracy metrics compared to the baseline and overfitting scenarios.
- The limited number of training epochs prevented the model from fully adapting to the training data, further contributing to the underfitting condition.
- Despite these constraints, the model still managed to achieve a test accuracy of over 70%, highlighting the inherent robustness of the MobileNetV3 architecture.

References

[1] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. Le, and H. Adam, "Searching for MobileNetV3," in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 1314-1324.

[2] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Master's thesis, Dept. of Comput. Sci., Univ. of Toronto, Toronto, ON, Canada, 2009.

[3] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv preprint arXiv:1704.04861, 2017.

[4] Hollemans, M. (2017) Google's MobileNets on the iPhone. Available at: https://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/ (Accessed: 12 March 2024).

[5] J. Hu, L. Shen, and G. Sun, "Squeeze-and-Excitation Networks," IEEE Conference on Computer Vision and Pattern Recognition, 2018

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510-452