Image Classification with CIFAR-10 Dataset

Group 4:
Mustafa Mert Çetin 22003953, Samet Senai Işık 21901701,
Ömer Faruk Keskin 21902147, Arda Özgün 21902283

***Abstract*—This project comprehensively explores and evaluates the performance of four advanced convolutional neural network (CNN) architectures — DenseNet, VGGNet, ResNet, and InceptionNet — on the CIFAR-10 image classification task. The CIFAR-10 dataset, a standard benchmark in computer vision, comprises 60,000 32x32 color images, separated into 50.000 train images and 10.000 test images, distributed across ten distinct classes and presents a diverse and challenging environment for assessing the efficacy of various models. The aim is to implement these architectures while gaining a deep understanding of the unique features and design principles inherent to each. The comparative analysis is centered on various metrics, including accuracy, and model complexity. Moreover, the study investigates how modifications and enhancements can be systematically applied to each architecture to optimize performance specifically for the CIFAR-10 dataset. The project endeavors to provide an in-depth understanding of these sophisticated CNN architectures and their implications in practical applications through extensive experimentation and rigorous analysis. Furthermore, this research serves as a foundational step toward understanding the effects of the parameters on sophisticated networks. It aims to develop more efficient and accurate machine-learning models by varying the hyperparameters and observing their effects.**

***Index Terms*—CIFAR-10, Image Classification, ResNet, VGGNet, DenseNet, Inception.**

I. Introduction

This project aims to conduct a comparative study and performance evaluation of four known CNN architectures: DenseNet, VGGNet, ResNet, and Inception Net. Each of these architectures represents a unique approach to deep learning, with distinct structural differences and learning capabilities, and has been selected for its notable features. The objective is to implement, analyze, and optimize these models for the CIFAR-10 dataset, a well-regarded dataset in computer vision.

This research begins with an implementation of each chosen architecture. This process involves not only the construction of the networks but also an in-depth exploration of the unique design principles and features that define each model. For instance, DenseNet is renowned for its feature reuse, VGGNet for its depth and simplicity, ResNet for its innovative use of residual connections, and Inception Net for its multi-level feature extraction. Understanding these characteristics is crucial for the effective implementation of these models and the interpretation of their performance.

Following the implementation, a comparative analysis focuses on several key metrics: accuracy, computational efficiency, and model complexity. Accuracy is measured to determine each model's ability to classify unseen images correctly. Computational efficiency is assessed to understand the resources required to train and deploy each model, an essential factor in real-world applications. Model complexity is analyzed to understand the trade-offs between the depth and breadth of the network and its performance.

Additionally, this study investigates how various modifications and enhancements can be systematically applied to each architecture to optimize performance specifically for the CIFAR-10 dataset. This includes adjustments in layer depth, dropout rates, activation functions, and learning rate. The goal is to identify the best-performing model for this particular task and understand how different architectural and hyperparameter choices impact each network's learning and generalization capabilities.

II. Methods

The methods section of a report is crucial for detailing the frameworks and tools employed in the study. This report involves a deep dive into the CIFAR-10 dataset, which is instrumental in training and testing the machine learning models. Additionally, the section outlines the architectures of various convolutional neural networks (CNNs), such as VGGNet, ResNet, DenseNet, and Inception, which are pivotal for understanding image classification tasks. It provides the theoretical foundation of each model and highlights the specific layers, functions, and innovations that contribute to their performance. The models are elucidated with their unique features, from VGGNet's deep but uniform structure to ResNet's revolutionary skip connections and DenseNet's feature-conserving layers. Furthermore, the methods section elucidates using the PyTorch library, showcasing its DataLoader object for efficient data manipulation, essential for handling complex datasets like CIFAR-10.

A. CIFAR-10 Dataset [1]

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. It is divided into five training batches and one test batch containing 10,000 images. The test batch has exactly 1,000 randomly selected images from each class. The training batches contain the remaining images randomly, with 5,000 images from each class. In this experiment, the train batches were used as one bigger batch, and 10% of the train set was used as the validation set. The split is performed using "PyTorch-torch.utils.data.random_split" function. Figure 1 shows representative pictures from each class of the CIFAR-10 dataset. The dataset includes classes like airplanes, automobiles, birds, cats, and more, with no overlap between similar classes such as automobiles and trucks. The dataset can be reached from [1]. The dataset can be used in Python3 easily. The dataset is encoded with the Python object serialization library

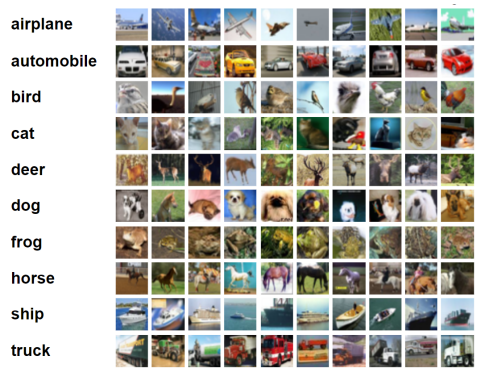

Figure 1: A representative sample from the CIFAR-10 dataset, showcasing the diverse array of classes which include vehicles, animals, and everyday objects. Each class is exemplified by various images that capture the dataset's complexity and the challenges inherent in automated image classification tasks.

"pickle." There is a function to help use the dataset with Python. However, in this project, the dataset was obtained from "torchvision.datasets.CIFAR10" due to its compatibility with PyTorch modules. The dataset was inputted to the below-mentioned models using PyTorch's "DataLoader" object.

B. VGGNet [2]

The VGG-11 architecture is a convolutional neural network (CNN) architecture introduced by the Visual Geometry Group (VGG) at the University of Oxford. It is a deep neural network architecture known for its simplicity and effectiveness in image classification tasks. VGG-11 is a variant of the original VGGNet, which includes 11 weight layers, hence the name.

Here's the theory behind the VGG-11 architecture:

1) Input Layer:: The input to the VGG-11 network is typically an RGB image with a fixed size of 224x224 pixels. Each pixel's RGB values are treated as input features.

2) Convolutional Layers:: VGG-11 consists of 8 convolutional layers, each performing convolutions on the input data. These convolutional layers have small 3x3 filters and use a stride of 1, which means they slide over the input with a small receptive field. The number of filters in these layers increases progressively from 64 to 512, capturing increasingly complex features.

3) Activation Function:: After each convolutional layer, a rectified linear unit (ReLU) activation function is applied. ReLU introduces non-linearity to the model and helps in learning complex patterns.

4) Pooling Layers:: After every two convolutional layers, a max-pooling layer is applied. Max-pooling reduces the spatial dimensions of the feature maps while retaining the most important information. The typical pooling size is 2x2.

5) Fully Connected Layers:: : VGG-11 has three fully connected layers at the end. These layers are similar to traditional neural network layers and are responsible for making predictions based on the high-level features extracted by the preceding convolutional layers. The sizes of these layers are typically 512 and 256, and the number of classes (e.g., 10 for CIFAR-10).

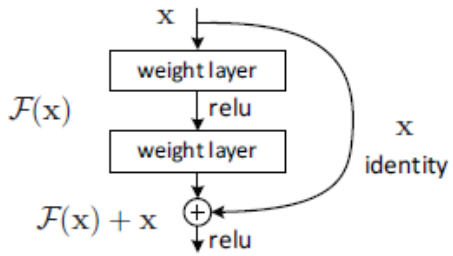

Figure 2: A residual building block used in ResNet. There is a skip connection in this block. Thanks to the skip connection, the gradient can flow without vanishing.

6) Dropout:: Dropout layers are applied after the fully connected layers during training. Dropout helps prevent overfitting by randomly setting a fraction of the neurons to zero during each forward and backward pass.

7) Output:: The final layer produces class probabilities, and the class with the highest probability is considered the predicted class for the input image.

8) Simplicity:: VGG-11 has a straightforward and uniform architecture with small 3x3 filters and pooling layers. Despite its effectiveness, VGG-11 has a large number of parameters, which can make it computationally expensive to train from scratch. However, it has paved the way for deeper and more efficient architectures.

C. ResNet [3]

ResNet (Residual Neural Network) architecture is a convolutional neural network (CNN) architecture introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun that won the 2015 ImageNet Competition [3].

As the deep convolutional neural networks begin to improve by increasing the number of layers, the problem of vanishing/exploding gradients has surfaced. This problem has been largely solved by normalized initialization of weights and intermediate normalization layers. But after this solution, degradation problems started to occur. A degradation problem is when the model's accuracy gets saturated after a point and decreases rapidly, which is not caused by over-fitting. Figure 2 shows the basic ResNet block.

ResNet is proposed to solve the degradation problem by introducing residual blocks into architecture. A residual block maps the input of the block to the output by introducing shortcuts. Shortcut connections skip one or more layers and perform identity mapping, and the input is added to the output of the block. This can be formulated as $H(x) = F(x) + x$ where x is the input and $F(x)$ is the output of the block. These shortcuts do not create any extra parameter or computational complexity [3].

1) Residual Learning: The creators of the models argued that if the added layers in a deep network could be constructed as identity mappings, then a deeper model should not have a greater error compared to a simpler model. However, optimization algorithms struggle to find the appropriate weights for multiple nonlinear layers. To solve this problem, authors have

Block	Layer	Type	Parameters
conv_block1	0	Conv2d	in=3, out=64, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=64, momentum=0.1
	2	ReLU	
	3	MaxPool2d	kernel=2, stride=2
conv_block2	0	Conv2d	in=64, out=128, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=128, momentum=0.1
	2	ReLU	
	3	MaxPool2d	kernel=2, stride=2
conv_block3	0	Conv2d	in=128, out=256, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=256, momentum=0.1
	2	ReLU	
conv_block4	0	Conv2d	in=256, out=256, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=256, momentum=0.1
	2	ReLU	
	3	MaxPool2d	kernel=2, stride=2
conv_block5	0	Conv2d	in=256, out=512, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=512, momentum=0.1
	2	ReLU	
conv_block6	0	Conv2d	in=512, out=512, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=512, momentum=0.1
	2	ReLU	
	3	MaxPool2d	kernel=2, stride=2
conv_block7	0	Conv2d	in=512, out=512, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=512, momentum=0.1
	2	ReLU	
conv_block8	0	Conv2d	in=512, out=512, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=512, momentum=0.1
	2	ReLU	
	3	MaxPool2d	kernel=2, stride=2
fc_block	0	Dropout	p=0
	1	Linear	in=25088, out=512
	2	ReLU	
fc_block1	0	Dropout	p=0
	1	Linear	in=512, out=256
	2	ReLU	
fc_block2		Linear	in=256, out=10

Table 1: VGG11 Architecture

proposed "residual learning formulation," where the model does not try to learn the desired output directly, but the network is trained to learn the residual between the input and the desired output. To achieve this, skip connections between layers have been introduced.

2) Identity Mapping by Shortcuts: Residual learning is achieved by shortcuts in every few stacked layers. The formulation for one block can be written as:

$$y = F(x, W_i) + x \tag{1}$$

where x and y are the input and output vectors corresponding to the block. $F(x, W_i)$ is the function of residual mapping that is to be learned.

3) Architecture Comparison with Other Models: ResNet-18 is a type of residual network model consisting of 4 residual blocks, each consisting of 2 convolutional layers. ResNet-34 is another type where four residual blocks have 3, 4, 6, and 3 convolutional layers, respectively. Figure 3 displays three different models. On the left, VGG-19 which is an older models that does not have shortcuts is displayed. VGG-19 is, thus, considered a plain network. ResNet models have fewer filters and lower complexity compared to VGG-19.

On the middle, a 34-layer plain model is displayed. ResNet-34 is achieved by adding shortcuts between the blocks in this model. In the same block, input and output sizes are equal and element-wise addition can be executed. For the shortcuts between the blocks, as the size of the input does not match the size of the output, two options are possible: (A) The input can be padded with zeros or; (B) the following equation can be used to perform projection:

$$y = F(x, W_i) + W_s X \tag{2}$$

1x1 convolutions do the second option. Both options are performed with a stride of 2.

D. DenseNet [4]

In the convolutional neural networks (CNNs) landscape, DenseNet (Densely Connected Convolutional Networks) represents a significant evolutionary step beyond its predecessors, such as AlexNet and VGGNet. It introduces a novel approach characterized by its dense connectivity pattern, a substantial departure from traditional CNN designs. In DenseNet, each layer is connected to every other layer in a feed-forward fashion, facilitating maximum information flow between layers. This intricate connectivity ensures that each layer can access the gradients from the loss function and the original input signal, leading to implicit deep supervision. Figure 4 illustrates the fundamental architecture of DenseNet, highlighting the efficient flow of information through its densely connected layers.

Layer	Operation	Details
Initial Convolutional Layer		
conv1	Conv2d	(3, 64, (3,3), (1,1), (1,1), bias=False)
bn1	BatchNorm2d	(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
relu	ReLU	inplace=True
Layer 1: BasicBlock (x3)		
0	Conv2d	(64, 64, (3,3), (1,1), (1,1), bias=False)
1	BatchNorm2d	(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
2	ReLU	inplace=True
3	Conv2d	(64, 64, (3,3), (1,1), (1,1), bias=False)
4	BatchNorm2d	(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
Layer 2: BasicBlock (x4)		
0	Conv2d	(64, 128, (3,3), (2,2), (1,1), bias=False)
1	BatchNorm2d	(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
2	ReLU	inplace=True
3	Conv2d	(128, 128, (3,3), (1,1), (1,1), bias=False)
4	BatchNorm2d	(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
Layer 3: BasicBlock (x6)		
0	Conv2d	(128, 256, (3,3), (2,2), (1,1), bias=False)
1	BatchNorm2d	(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
2	ReLU	inplace=True
3	Conv2d	(256, 256, (3,3), (1,1), (1,1), bias=False)
4	BatchNorm2d	(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
Layer 4: BasicBlock (x3)		
0	Conv2d	(256, 512, (3,3), (2,2), (1,1), bias=False)
1	BatchNorm2d	(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
2	ReLU	inplace=True
3	Conv2d	(512, 512, (3,3), (1,1), (1,1), bias=False)
4	BatchNorm2d	(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
Global Average Pooling and Fully Connected Layer		
avg_pool	AdaptiveAvgPool2d	output_size=(1,1)
fc	Linear	in_features=512, out_features=10, bias=True

Table 2: ResNet-34 Architecture

1) Innovation and Performance: DenseNet's innovation lies in its ability to conserve features throughout the network, whereby each layer propagates its feature maps to all subsequent layers. This characteristic improves feature reuse and significantly reduces the number of parameters, as there is no need to relearn redundant feature maps. DenseNets have shown remarkable performance on standard benchmarks like ImageNet and CIFAR-10, often outperforming more complex and computationally heavy architectures. For instance, DenseNet's efficiency and accuracy in image classification tasks have set new precedents, demonstrating the architecture's robustness and potential as a foundation for future explorations.

2) Comparative Analysis: Compared with architectures such as ResNets, DenseNets offer a more compact parameter set due to their feature-conserving nature. Unlike ResNets, which utilize addition-based skip connections to combat the vanishing gradient problem, DenseNets opt for a concatenative approach, merging feature maps from all previous layers. This unique method has been instrumental in establishing DenseNet as a more parameter-efficient alternative without compromising the network's depth or performance.

3) ResNets: ResNets (Residual Networks) introduced the concept of skip connections, where the input to a layer is added to its output, helping to mitigate the vanishing gradient problem by allowing gradients to flow through the network more easily. The equation for a residual block is:

$$\mathbf{x}_l = H_l(\mathbf{x}_{l-1}) + \mathbf{x}_{l-1} \tag{3}$$

where $H_l(\cdot)$ is the transformation (composite) function of the l^{th} layer and $\mathbf{x}_{l-1}$ is the output of the previous layer.

4) Dense Connectivity: DenseNet extends the idea of skip connections by concatenating the feature maps from all previous layers as input to each layer rather than adding them. The equation for a DenseNet block is:

$$\mathbf{x}_l = H_l([\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{l-1}]) \tag{4}$$

where $[\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{l-1}]$ is the concetanation of the layer inputs.

This ensures that each layer receives "collective knowledge" from all preceding layers, making the network more efficient and improving feature reuse.

5) Composite Function: In DenseNet, $H_l(\cdot)$ is defined as a composite function of three consecutive operations: Batch normalization (BatchNorm2d), followed by a rectified linear unit (ReLU), and a 3x3 convolution (Conv2d). The function is:

$$H_l(\cdot) = Conv2d\_3x3(ReLU(BatchNorm2d(\cdot))) \tag{5}$$

This composite function is applied to each layer's input, which includes feature maps from all preceding layers.

6) Transition Layers: DenseNet uses transition layers between its dense blocks to control the number of feature maps and to reduce the size of the feature maps, thus compacting the model and reducing the number of parameters. A transition layer typically consists of batch normalization, followed by a 1x1 convolutional layer and a 2x2 average pooling layer:

$$T(\cdot) = AvgPool2d\_2x2(Conv2d\_1x1(BatchNorm2d(\cdot))) \tag{6}$$

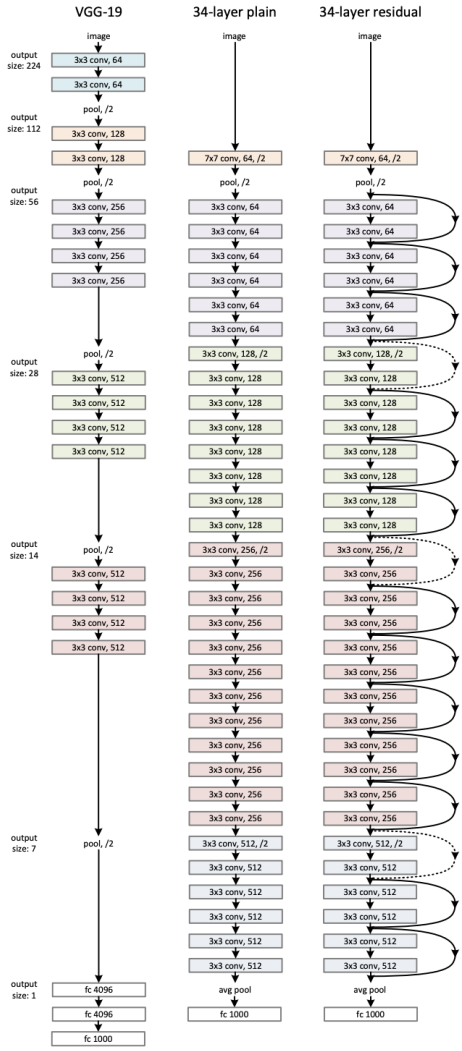

Figure 3: Example Network Architectures. **Left**: The VGG-19 model [2] (19.6 billion FLOPs) as a reference. **Middle**: A plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: A residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions.

7) Growth Rate: The growth rate in DenseNet, denoted as k, refers to the number of feature maps added by each layer to the collective knowledge. If the initial layer starts with k_0 feature maps, then the l_{th} layer receives the feature maps from all preceding $l-1$ layers, each contributing to k feature maps. Therefore, the total number of feature maps input to the l_{th} layer is $k_0 + k \times (l-1)$, accounting for the initial k_0 feature maps and k additional feature maps from each of the preceding layers. This formulation is fundamental to understanding the incremental growth of the network's capacity and the compounding of features through its depth.

8) Bottleneck Layers: To further improve computational efficiency, DenseNet can include bottleneck layers before each 3x3 convolution, where the number of input feature maps is reduced using 1x1 convolutions. This is commonly called BN-ReLU-Conv(1x1)-BN-ReLU-Conv(3x3). In the bottleneck layer, firstly, batch normalization is applied to the input,

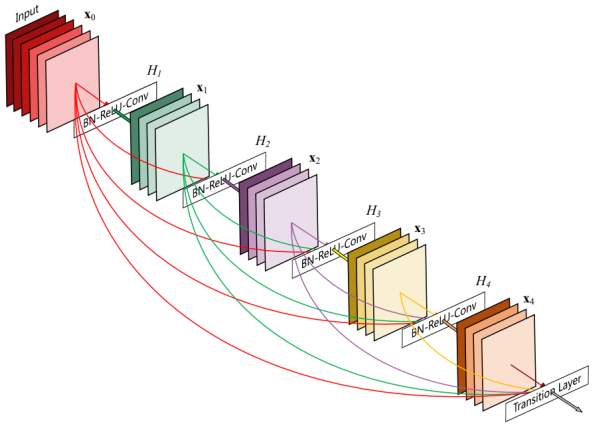

Figure 4: Illustration of DenseNet architecture [4], showcasing the innovative concept of dense connectivity. Each of the five depicted convolutional layers receives concatenated feature maps from all preceding layers, resulting in highly enriched feature representations. These layers are preceded by batch normalization and ReLU activation, with their outputs concatenated to form the subsequent layers' inputs, thus embodying the essence of dense connectivity.

and then ReLU is used for nonlinearity. Then, the output of the ReLU layer is convolved with a 1x1 kernel. The resulting output of the convolution layer with 1x1 kernel has $4k$ output channels. Then, the composite function defined above is applied to this convolution layer's output, resulting in k output channels. The bottleneck layer consists of batch normalization, ReLU, 1x1 convolution, and the composite function. DenseNets containing bottleneck layers are called "DenseNet-B."

9) Compression: To reduce the size of the model, a compression factor (θ) can be applied in the transition layers to reduce the number of feature maps:

$$m \rightarrow transitionlayer \rightarrow \lfloor \theta m \rfloor \tag{7}$$

where m is the number of input feature maps to the transition layer, and $\lfloor \cdot \rfloor$ denotes the floor function, ensuring an integer number of output feature maps.

DenseNets containing this compression operation are called "DenseNet-C," if the network both contains the bottleneck layer and the compression in the transition layers, then the network is called "DenseNet-BC."

10) Implementation: Implementing DenseNet involves stacking multiple densely connected blocks, each followed by a transition layer (except for the last block). Each block consists of multiple layers as defined by the composite function, and the entire network is capped by a global average pooling layer followed by a fully connected layer for classification. This modular and highly parameter-efficient architecture has been shown to achieve excellent performance on image classification tasks while being more computationally efficient than many traditional convolutional networks.

E. Inception [5]

Inception, also known as GoogleNet, is a convolutional neural network architecture developed by Christian Szegedy et al. and is introduced in the paper "Going deeper with convolutions" [5]. It was designed to participate in the ImageNet Large Scale Visual Recognition Challenge 2014 (ILSVRC 2014) and won the competition.

It had several improvements over the initial version that won the ILSVRC 2014, namely Inception-v3, Inception-v4, and Inception-Resnets. The v2 is briefly discussed and directly improved to v3 in the same paper, "Rethinking the Inception Architecture for Computer Vision" [6]. The later improvements are introduced in the paper "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning" [7].

Each improvement of the Inception architecture increases its parameters, starting from 5.491M in the Inception-v1 to 21.640M in the Inception v3, and later to 43M in the inception v4. Inception-Resnets have a similar number of parameters to the v3 and v4. Hence, due to increasing computational demand, only Inception-v1 will be run and discussed in this project.

The detailed architecture of the model is provided in Table 8.

1) Convolution Layers: All the convolution layers in the model, whether in an inception module or stand-alone, are a sequence of a convolution, a batch normalization, and a ReLU.

2) Inception Layers: The key idea behind the Inception model is to have multiple different-sized convolutional filters operate on the same level and then concatenate their outputs as different channels. The resulting network gets *wider* instead of *deeper*. The Inception module introduced in the paper is given in Fig. 5.

It is noted in the paper that embeddings, as in dimension reductions and projections, represent information in a dense, compressed form, and compressed information is harder to model. However, if done judiciously, they can be done without much or any loss in representational power. This introduces the idea of performing 1x1 convolutions to reduce channels before the expensive 3x3 and 5x5 convolutions, significantly reducing the total computational load without sacrificing representativity. Hence, the improved Inception module is shown in Fig. 6, beside the *naive* implementation.

3) Auxiliary Branches: In the original paper, an idea to address the vanishing gradient problem is to add *auxiliary branches* to the network. These are separate outputs from the network branched from earlier layers with their ending sequence, that is, pooling and a fully connected network. The outputs from these auxiliary branches are also subjected to the same loss function and added to the total loss with a discount weight:

$$TotalLoss = Loss_{main} + 0.3 \cdot Loss_{Aux_1} + 0.3 \cdot Loss_{Aux_2} \quad (8)$$

However, these auxiliary branches are only considered during training and are discarded during validation and testing. The later published paper [6], that brings the inception-v3, retrospectively points out that the auxiliary branches act as a regularizer to the network.

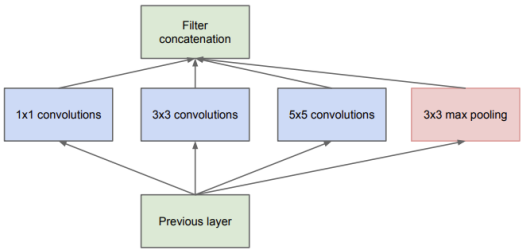

Figure 5: Inception Module, naive version.

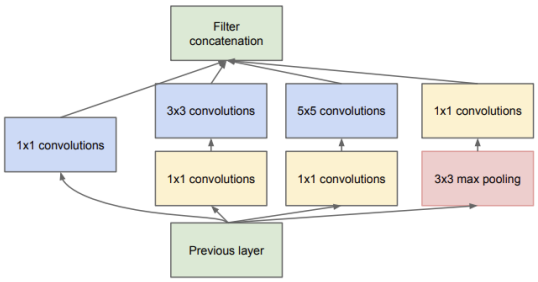

Figure 6: Improved Inception Module with the dimension reductions.

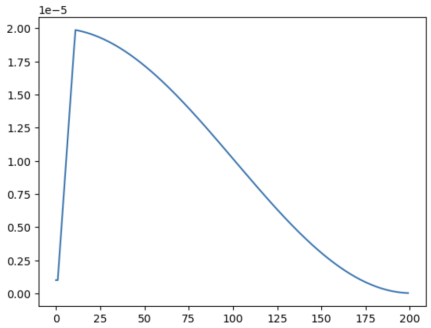

Figure 7: An example cosine annealing with a warm-up of 20 epochs.

4) Label Smoothing: The second paper introduces label smoothing to the Inception-v3 model. Label smoothing is a regularization method that introduces noise to the labels. The idea is to prevent the network from becoming overconfident in its predictions and reduce overfitting. It is done by replacing the hard 0 and 1 classification labels by $\frac{\epsilon}{K-1}$ and $1-\epsilon$, respectively.

5) Cosine Annealing with Warm-up: Cosine Annealing with Warm-up is a learning-rate scheduling technique in which the learning rate is started with a near-zero value and linearly incremented to its base value during the warm-up epochs. Then, the learning rate is determined by the values of cosine for the rest of the epochs such that the rest of the epochs will be a quarter of a period of a cosine, as shown in Fig. 7.

III. Results

In this section, we present a comprehensive analysis of the performance of various convolutional neural network (CNN) architectures, specifically VGGNet, ResNet, DenseNet, and

Model	Parameter Size	Train Acc. (%)	Val. Acc. (%)	Test Acc. (%)	Epoch No.	Dropout	Learning Rate	Momentum	Model Version	Pooling Method
1	22,205,450	97.86	83.96	84.01	29	0.5	0.01	-	VGG11	Max Pooling
2	22,205,450	98.81	80.8	80.3	23	0	0.01	-	VGG11	Max Pooling
3	22,205,450	94.58	83.94	83.74	20	0.5	0.1	-	VGG11	Max Pooling
4	22,205,450	99.86	86.45	86.2	21	0	0.1	-	VGG11	Max Pooling
5	22,205,450	98.19	79.52	79.25	17	0	0.1	0.9	VGG11	Max Pooling
6	22,205,450	99.44	85.38	86.16	16	0	0.1	-	VGG11	Average Pooling
7	33,014,858	99.29	85.48	84.3	19	0	0.1	-	VGG19	Max Pooling

Table 3: Comparison of VGGNet Models

Model	Parameter Size	Train Acc. (%)	Val. Acc. (%)	Test Acc. (%)	Epoch No.	Dropout	Learning Rate	Momentum	Model Version	Pooling Method
1	11,173,962	94.35%	88.58%	89.06%	25	0	0.001	0	Resnet18	Avg. Pooling
2	11,173,962	94.73%	88.48%	88.50%	25	0	0.01	0	Resnet18	Avg. Pooling
3	11,173,962	92.96%	88.54%	88.80%	25	0.5	0.01	0	Resnet18	Avg. Pooling
4	11,173,962	94.95%	88.96%	89.47%	25	0.5	0.001	0	Resnet18	Avg. Pooling
5	11,173,962	92.72%	87.16%	87.29%	25	0.5	0.001	0.9	Resnet18	Avg. Pooling
6	11,173,962	73.06%	59.24%	59.01%	25	0.5	0.001	0	Resnet18	Max Pooling
7	21,282,122	94.76%	88.98%	89.23%	25	0.5	0.001	0	Resnet34	Avg. Pooling

Table 4: Comparison of ResNet Models

Model	Parameter Size	Train Acc. (%)	Val. Acc. (%)	Test Acc. (%)	Epoch No.	Dropout	Learning Rate	Momentum	Model Depth	Model Version
1	4,113,082	89.62	81.62	82.14	18	0.5	0.01	-	L=36, k=24	DenseNet
2	4,113,082	84.8	78.2	78.34	16	0	0.01	-	L=36, k=24	DenseNet
3	4,113,082	92.47	82.64	82.71	24	0.5	0.1	-	L=36, k=24	DenseNet
4	4,113,082	97.24	85.24	85.78	25	0	0.1	-	L=36, k=24	DenseNet
5	4,113,082	94.34	84.3	83.71	27	0	0.1	0.9	L=36, k=24	DenseNet
6	1,979,098	94.26	83.32	84.38	17	0	0.1	-	L=36, k=24	DenseNet-BC
7	9,016,714	88.5	82.9	82.45	15	0	0.1	-	L=54, k=24	DenseNet

Table 5: Comparison of DenseNet Models

Model	Parameter Size	Train Acc. (%)	Val. Acc. (%)	Test Acc. (%)	Epoch No.	Label Smoothing	Auxiliary Branches
1	5,491,000	99.0785	92.7684	92.8385	50	No	No
2	5,491,000	99.5072	92.0974	92.0974	50	No	Yes
3	5,491,000	98.5757	92.3177	92.3778	50	Yes	No
4	5,491,000	99.1647	91.9671	91.9772	50	Yes	Yes

Table 6: Comparison of Inception Models

Inception models, tailored and evaluated on the CIFAR-10 dataset. The experimental setup was designed to explore the effects of different hyperparameters and architectural choices—such as depth, dropout, learning rate, and pooling methods—on the models' training and generalization capabilities. While the experiments were not optimized to achieve peak performance, they provide valuable insights into the models' behavior under various configurations.

A. VGGNet Results

The Table 3 presents a comparative analysis of different VGGNet models on the CIFAR-10 dataset, evaluating their performance in terms of parameter size, accuracy, and training configuration. Models 1 to 7 vary in terms of dropout rate, learning rate, and the use of momentum during training. Models 1 and 2, which employ a lower learning rate of 0.01, show a significant difference in performance when dropout is introduced. Model 1, with a dropout rate of 0.5, achieves a training accuracy of 97.86% and a test accuracy of 84.01%. Model 2, which has no dropout, sees a slight increase in training accuracy to 98.81% but a noticeable decrease in test accuracy to 80.3%. Models 3, 4, and 5, with a higher learning rate of 0.1, display a diverse impact of dropout and momentum. Model 3, with dropout, shows lower training accuracy (94.58%) but maintains a competitive test accuracy (83.74%). Model 4, without dropout, demonstrates the highest training accuracy (99.86%) and test accuracy (86.2%) among the VGG11 models with max pooling. Model 5, which introduces momentum (0.9) experiences a decrease in test accuracy to 79.25%, suggesting that momentum may not contribute positively to generalization for this model configuration. Model 6, employing an average pooling method instead of max pooling, achieves a slightly lower test accuracy (86.16%) compared to the best-performing VGG11 model with max pooling (Model 4). Model 7, which expands the architecture complexity to VGG19, maintains a relatively high training accuracy (99.29%) but exhibits a lower test accuracy (84.3%) compared to the best VGG11 model (Model 4). This indicates that increasing the model depth to 19 layers does not necessarily translate into better performance. All models were trained by using early stopping and they have stopped between 15-20 epochs. All these models were trained using Google Colab with V100 GPU.

B. ResNet Results

Table 4 displays the results of 7 different ResNet models trained with different parameters.

The training of the ResNet-18 models was done using the Adam (short for Adaptive Moment Estimation) optimization algorithm, which is commonly used for training deep neural networks. Two values were tested for learning rate, momentum constant, and dropout probability. Average and max pooling were compared, keeping the other parameters at their best-performing values. The resulting best-performing model parameters were used to train the ResNet-34 model.

A lower learning rate (0.001) has achieved slightly better results than greater learning rates (0.01). A dropout probability of 0.5 has performed better compared to no dropout case. The best of the four cases was again trained with momentum using 0.9 as the constant. The test accuracy in this case has dropped around 2%. Max pooling was tested using a learning rate of 0.001, a dropout probability of 0.5, and no momentum. The resulting accuracy has dropped dramatically compared to previous cases, around 25 to 30%.

When the best-performing parameters were used in training ResNet-34 architecture, it did not result in a big change in the accuracy rate of the model. It was at the same percentile with the best-performing ResNet-18 model, at 89%.

All models were trained for 25 epochs. The models were trained using Google Colab with T4 GPU.

C. DenseNet Results

Table 5 is the training results of seven different DenseNet-based models with slight differences. The experiment's goal was not to obtain the best results, therefore, the parameter selection was made for experimental purposes. Additionally, a dropout layer was included before the classification layer of the DenseNet, which does not exist originally in [4]. Additionally, the parameter L represents the depth of the network. The depth represents the number of DenseBlocks, which is Batch Normalization, ReLU and 3x3 convolution layer, in this experiment. The transition layers are not included in calculating the parameter. The depth count standard is not given in [4]. Therefore, this parameter is interpreted as the number of DenseBlocks. [4] states that they experimented with CIFAR-10 but did not share their models' exact architecture. Their primary focus was ImageNet. Therefore, the CIFAR-10 experiment is not given in detail. They said they used a network with $L = 40$. However, as said before, the standardization of the parameter L is not clearly defined. Therefore, the network used in this experiment is constructed somewhat instinctively. Each model in Table 5 has three DenseLayers, which consist of DenseLayers and use input concatenation at the output after each block. Transition layers determine the limits of the DenseLayers. There are 6 DenseBlocks in models 1-6 and 12 in Model 7. In the second DenseLayer, there are 18 DenseBlocks in models 1-6 and 24 in model 7. Finally, there are 12 DenseBlocks in the third DenseLayer in model 1-6, and 24 DenseBlocks in the third DenseLayer of model 7. The motivation behind this is that the first three DenseLayers in the implementation of DenseNet for ImageNet were copied, omitting the fourth DenseLayer considering that CIFAR-10 is much simpler than the ImageNet and, therefore, the first three layers should be enough to capture the relations between the images. The intuition behind model 7 is that the more DenseBlocks should capture more features, leading to better results. All models except model 6 are in the Default mode, meaning they do not utilize the bottleneck layer and the compression in the transition layer. Model 6 was designed to test the efficiency of the bottleneck layer and the compression in the transition layer. However, testing these in different models might be more convenient to see the individual effects. Due to strict time constraints, the experiments on more models could not be performed. The expansion rate k selection is directly from [4]. Again, experimenting with different k values might be more beneficial to see the effect on the performance of the image classification task. Other parameters are chosen according to the experiments discussed above so that the performance of the models can be compared easily.

Google Colab was utilized to train these models due to its high-performance T4 GPUs. The training parameters are chosen similarly to the implementation of the VGGNet. The batch size 64 and the SGD optimizer from the "PyTorch.optimizer" module were used. Each model was run for 60 epochs, but early stopping based on validation loss is used. The patience was set 7, meaning that the training was stopped if the validation loss did not improve for 7 epochs. Then, the best performing model in terms of validation loss was outputted at the end of the training using the PyTorch's "load_state_dict()" function. Table 5 gives the learning rate and momentum parameters. The first four models for tests to find the best learning rate and dropout rate. Then, the best-performing model, model 4, was tested with momentum in model 5. Model 4 and Model 5 were compared and Model 4 was found to be superior. Again, Model 4 parameters in the DenseNet-BC configuration were tested in Model 6. Model 4 was again found to be superior. Then, the number of DenseBlocks in the DenseLayers increased in Model 7.

Comparing Model 1 (82.14%) and Model 2 (78.34%), the model with dropout probability performed better. However, Model 4 (85.78%) performed better comparing to Model 3 (82.71%) even though Model 4 has no dropout. Comparing learning rates, the ones with the higher learning rates performed better. Model 5 (83.71%) was trained with momentum, but this does not result in an improvement in the test accuracy. DenseNet-BC setup was experimented in Model 6 (84.38%), performed even though it has a 1.9M parameters, which is comperatively very low. Model 7 (82.45%) contains more parameters comparing the to the rest of the DenseNet models, however, it does not performed as expected.

D. Inception Results

The training was done using Stochastic Gradient Descent with 0.9 momentum. Cosine annealing with a warm-up method was used for scheduling with 30% of the total epochs used for warm-up and 0.01 as the base value of the learning rate. Nvidia GeForce RTX 4090 was used as the training GPU.

Metrics such as loss and accuracy, and confusion matrices are provided in Fig. 15-22. A simplified results of the four model variations are provided in Table 6. The more extensive results are provided in Table 7.

All the model variations had relatively close test set accuracy results that no two result is farther away than %1. Best test set accuracy was obtained by the vanilla Inception model by %92.8385. Vanilla Inception is also the best performer in all Top-k accuracy classifications. The second model is with label smoothing but without the auxiliary branches with %92.3778. However, this model variation performed second only in the Top-1 and Top-2 accuracy, and is in the last place for the Top-3

accuracy. The model with auxiliary branches but without label smoothing is in the third place, and the model with both are in the third and fourth place respectively. Their other Top-k accuracy metrics are also in the same fashion, expect the situation with the second model.

All the model variations had pretty similar graphs for their loss and accuracy per epoch. However, it must be noted that direct comparisons of loss values would not be correct since both the auxiliary branches and label smoothing introduce additional loss values into the calculation, and the same model performance otherwise would have different loss values.

IV. Discussion

This experiment was run to classify the CIFAR-10 dataset images into ten classes. The dataset contains 60000 32x32 RGB images. The pre-processing for the data was not deemed necessary since one image has $32x32x3 = 3072$ features. This feature size for a sample seems to be enough for the classification task, considering there are only 10 classes for 60000 images. Convolutional networks are relied upon for their efficacy in accurately classifying images. Therefore, no new feature engineering or analysis is run on the dataset in terms of pre-processing the image. Additionally, the methods used in this experiment are not run on other datasets. However, popular datasets like ImageNet, CIFAR-100, and SVHN were tested in the descendants of the methods implemented in this paper. Especially, ImageNet was tested on various models due to the famous ImageNet competition held annually.

In the study of VGGNet architecture, variations of the learning rate, dropout probability, momentum, and pooling methods reveal different outcomes where each hyperparameter significantly impacts model performance. Adam optimization algorithm has been chosen to optimize the performance of the network. A higher learning rate of 0.1 has shown that a higher learning rate has performed better than 0.01 by reducing training error and boosting test accuracy when coupled with dropout probability 0. A high learning rate, allows higher steps towards convergence and makes it easier to escape from the local minimum. Then, the introduction of the dropout probability 0.5 for these learning rate scenarios has shown that a smaller learning rate 0.01 was better when dropout is introduced compared to 0.1. It allows smaller weight updates and smaller steps towards convergence to avoid overshooting. Dropout regularization compels the neurons with a specific probability to prevent overfitting. For the model, the learning rate is 0.1, and the dropout probability is varied; eliminating half of the neurons leads to a significant loss of information and learning capacity. For the models whose learning rate is 0.01, the introduction of dropout regularization increased the performance of the network by preventing overfitting and increasing the network's generalization ability. Then, momentum was coupled with a high learning rate and 0 dropout probability. The performance of the network has decreased. Introducing the momentum allows weight updates depending on both current and previous updates. For this particular case, momentum has caused the network to reach optimal points during training, leading to less accurate results. Then, the best model was used by changing the max-pooling layers with average pooling, which typically calculates the average instead of taking the maximum value. For this particular model, changing the max pooling layer with average pooling layers remained stable with a slight decrease compared to max pooling. The average pooling layer might cause to forget some critical features that max pooing would capture aggressively. Then, the best model is trained with an expanded network which contains 16 convolution layers rather than 8. This modification has increased the required training time significantly without improving the performance. Adding additional weights to the network might cause being more inclined to overfitting, especially without dropout. This experience underlines the importance of selecting hyperparameters carefully by considering the network.

Experimenting with different configurations of ResNet-18 and ResNet-34 has yielded useful information for comparing different parameters and their effect on the model performance. The choice of learning rate, dropout probability, momentum, and pooling method have influenced the training outcomes. Adam optimization algorithm was chosen and was not replaced as commonly used in CNN models. The variation in learning rates has shown that a lower learning rate of 0.001 has performed slightly worse compared to a higher learning rate of 0.01, which is in line with the expectation as a lower learning rate allows smaller weight updates and smaller steps towards convergence to avoid overshooting. Introducing dropout regularization has proved to improve the performance across most cases compared to scenarios with no dropout. This result is in line with the expectation, as the dropout's role is to prevent overfitting by randomly dropping neurons during training, increasing the network's generalization ability. A slight decrease in accuracy was observed when a lower learning rate and dropout regularization were coupled with momentum. The effect of momentum on gradient descent can be equivalent to higher learning rates, which decreases the model's accuracy rate. When max-pooling was used instead of average pooling after the last block in the ResNet architecture, the accuracy rates dropped dramatically, implying that average pooling preserves more spatial information about the input images compared to max pooling, which results in better overall feature extraction. When the optimized parameters were carried into ResNet-34 architecture, no significant change in the accuracy rate was observed, indicating that the optimization of the model may be saturated after some level of model complexity. Improving the results further might require exploring architectural adjustments or different model architectures.

An analysis was made on the CIFAR-10 dataset using the DenseNet architecture with slight differences. The architecture was offered to capture more insight into the dataset using fewer parameters with skip connections. However, it utilizes skip connections slightly differently from the infamous ResNets. The DenseNet concatenates its inputs to the outputs in some layers so that the information about the input will be used in the later layers of the network. This also promises to solve the vanishing gradient problem in Deep Networks. In addition to the original network architecture offered in [4], a dropout layer

was introduced before the classification layer of the network. This change was made with the intention of preventing overfitting in the complex network structures. As the complexity increases in the network, it has more parameters to memorize the train set. The network also has bottleneck connections, meaning that the features are firstly increased so that the later layer has more data to capture the relations. The network complexity in terms of layer sizes was not clearly specified in the original implementation of the DenseNet; therefore, different size parameters were defined in this implementation. The network also offers a compression method that reduces the parameter size significantly. For example, Model 6 utilizes a compression rate of 0.5 and uses bottleneck connections. However, the number of parameters of Model 6 is found to be 1.9 million. However, Model 4 has the same depth but does not utilize compression. Its parameter size is around 4 million. Also, looking at the test accuracy scores, it can be said that the compression serves the purpose since Model 6 performed similarly to Model 4, even though Model 6 has fewer parameters. The learning differed in the analysis. Model 1 and Model 2 reached early stopping before Model 3 and Model 4, probably because a higher learning rate resulted in more oscillatory behaviors of the weight updates, resulting in a more unstable training process. However, the ones with higher learning rates converged to local minima with better testing accuracy. Another parameter tested is the dropout rate, which originally did not exist in the DenseNet implementation. The effect of dropout is not consistent. Model 1 is superior to Model 2, but Model 3 is inferior to Model 4. The increase might be because the dropout prevents overfitting and results in better test accuracy, or it might be stochasticity since the initialization is from random distributions. The decrease might be because dropouts might prevent learning due to dropped connections in the training. The momentum was tested to achieve better performance in Model 5, but it could not surpass the performance of Model 4. The momentum constant was chosen as 0.9. This might dominate the gradient, resulting in the weight update always going in the same direction, even though the gradient might indicate another direction. A new block structure and compression were tested with Model 6. The parameter size decreased to half. However, the network performance remained unchanged. The model might perform better if the parameter size were set to 4 million. The last model was tested to see the effect of network size. Model 7 has two times the parameters of Model 4. The performance of the model did not increase as expected. Overall, the best performance was obtained from Model 4 with 85.78%. However, in the original implementation, reached error rates of 5.77% with default layers and reached 5.19 % suggesting that this experiment needs further hyperparameter tuning. In addition, the original implementation used 7 million and 15 million parameters, showcasing the effectiveness of the network size. However, these results were obtained when the training was done for 300 epochs, which is not applicable to this experiment due to time limits. Similar results might be obtained if the early stopping was removed and the training was done for 300 epochs.

Experiments with the various Inception models yielded quite unexpected results. Surprisingly, the vanilla setup obtained the best test set accuracy in all accuracy metrics, as shown in Table 7. This is also in contrast with what the original papers suggest since in both [5] and [6], it is argued that both auxiliary branches and label smoothing improve the results. Auxiliary Branches are introduced in the paper [5]; however, they are further discussed in the paper [6]. It is argued in [6] that auxiliary branches have a regulatory role in the model, and their effects only start to be noticeable rather late in the training. It is 100 epochs of training in the original papers in contrast to 50 in my experimentation. Hence, their apparent ineffectiveness might be caused by the lack of late-training epochs in which auxiliary branches perform their role as regularizers and provide better accuracy by reducing overfitting.

Label smoothing also performed ineffectively in contrast to the expectation. It is quite surprising since label smoothing is a somewhat popular method that is known to prevent the network from becoming over-confident to improve its results, and it is used in many state-of-the-art models, including image recognition, language translation, and speech recognition. [8] A similar argument about the lack of late-training epochs can be made for label smoothing since its main role in the model is also as a regularizer.

The vanilla Inception achieved the best test set accuracy of %92.8385 among all models in the project.

In a comparative study of CNN architectures on the CIFAR-10 dataset, the analysis revealed useful conclusions where model complexity, such as that seen in VGG19, did not always correlate with superior performance, as simpler VGG11 configurations often yielded better results. DenseNet models stood out for their efficient parameter usage and robustness, while ResNet models benefited from residual connections but required careful consideration of pooling methods and momentum for optimal generalization. Inception models indicated that additional complexities like auxiliary branches and label smoothing might not enhance performance within certain training limits, as the vanilla Inception setup achieved the highest accuracy. This suggests that for specific datasets like CIFAR-10, simpler architectures or those with thoughtful implementation of complexity, can be more effective, emphasizing the importance of matching the model design intricately with the dataset characteristics.

REFERENCES

[1] *The CIFAR-10 dataset, Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton, University of Toronto, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html*

[2] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition." arXiv, Apr. 10, 2015. doi: 10.48550/arXiv.1409.1556.

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[4] G. Huang, Z. Liu, L. Van Der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 2261-2269, doi: 10.1109/CVPR.2017.243.

[5] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 2015, pp. 1-9, doi: 10.1109/CVPR.2015.7298594.

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 2818-2826, doi: 10.1109/CVPR.2016.308.

[7] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (San Francisco, California, USA) (AAAI'17). AAAI Press, 4278–4284.

[8] Rafael Müller, Simon Kornblith, and Geoffrey Hinton. When does label smoothing help? In Proceedings of the 33rd International Conference on Neural Information Processing Systems, 2019.

APPENDIX

A. Elaborated Results

This part gives detailed results on the microscopic level. The confusion matrices, the training losses, the validation losses, the training accuracy, and the validation accuracy are given in this part with the related model's section.

1) VGGNet: This section gives the results of the VGGNet. Figures 8-14 are the outcomes of the training of 7 different VGGNet models.

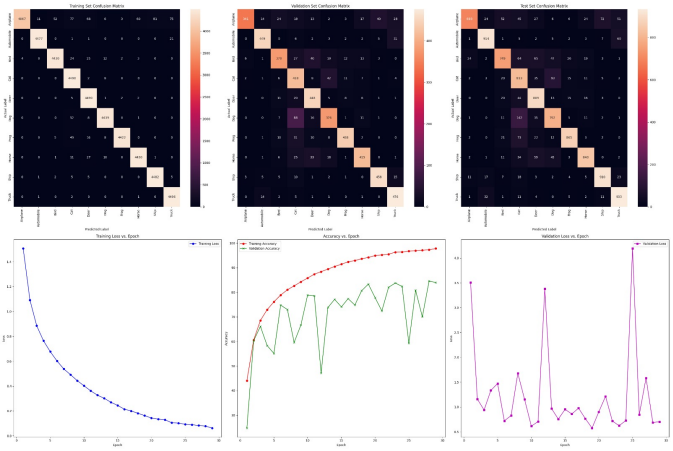

Figure 8: VGGNet Model 1 training results.

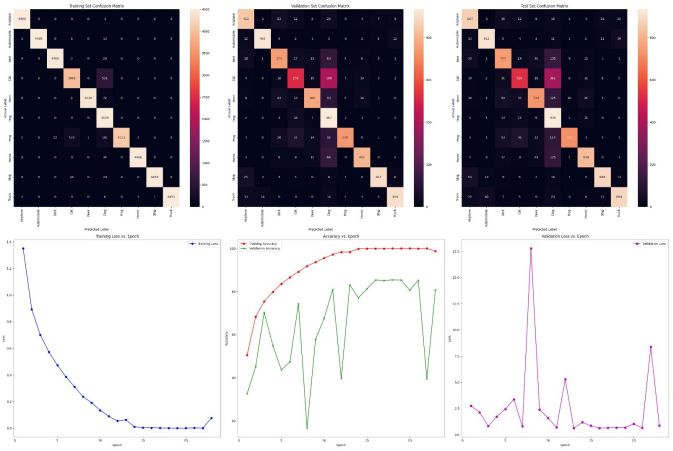

Figure 9: VGGNet Model 2 training results.

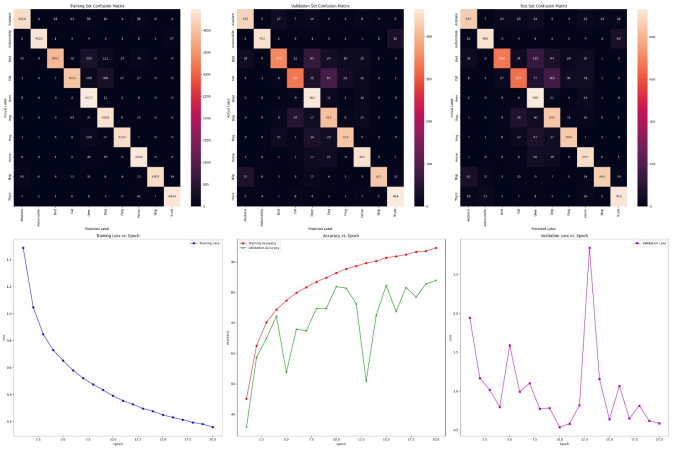

Figure 10: VGGNet Model 3 training results.

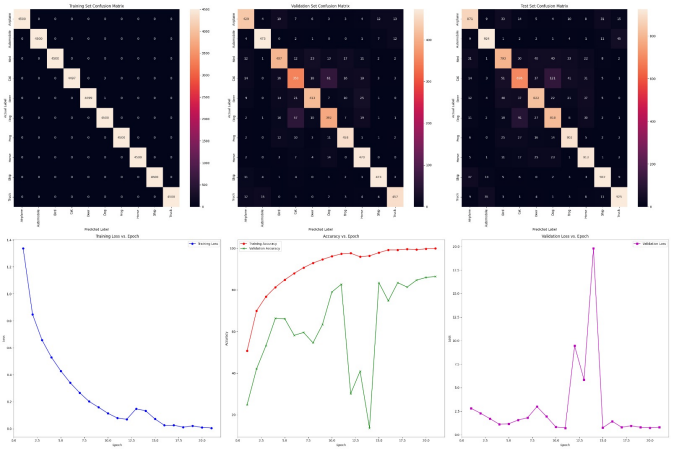

Figure 11: VGGNet Model 4 training results.

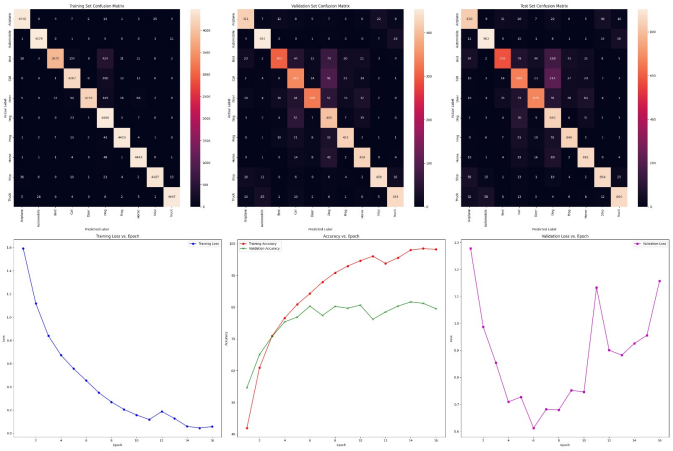

Figure 12: VGGNet Model 5 training results.

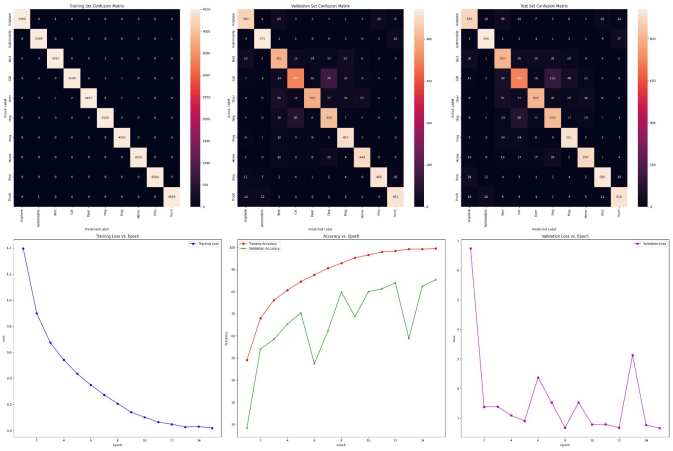

Figure 13: VGGNet Model 6 training results.

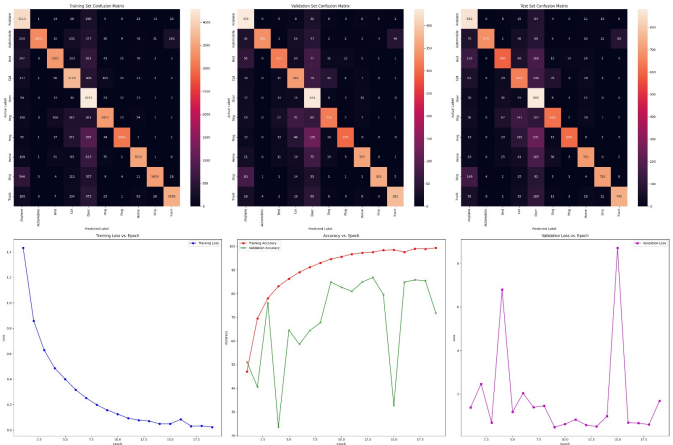

Figure 14: VGGNet Model 7 training results.

2) ResNet: This section gives the results of the ResNet. Figures 15-21 are the outcomes of the training of 7 different ResNet models.

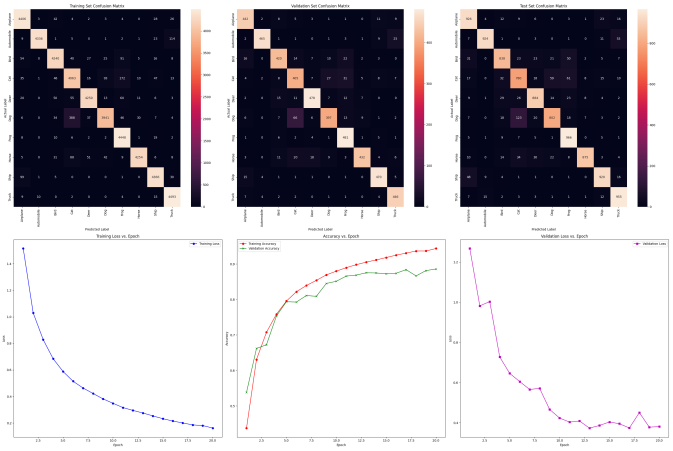

Figure 15: ResNet-18 Model 1 training results.

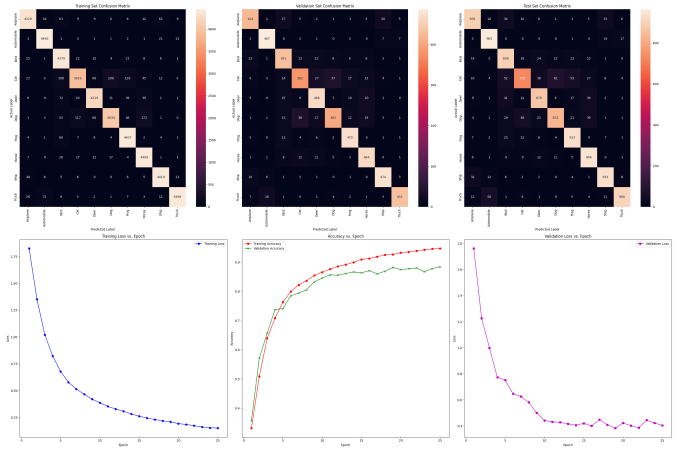

Figure 16: ResNet-18 Model 2 training results.

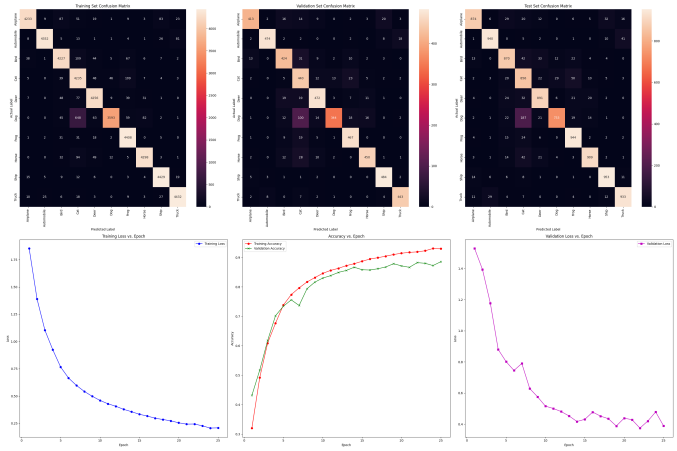

Figure 17: ResNet-18 Model 3 training results.

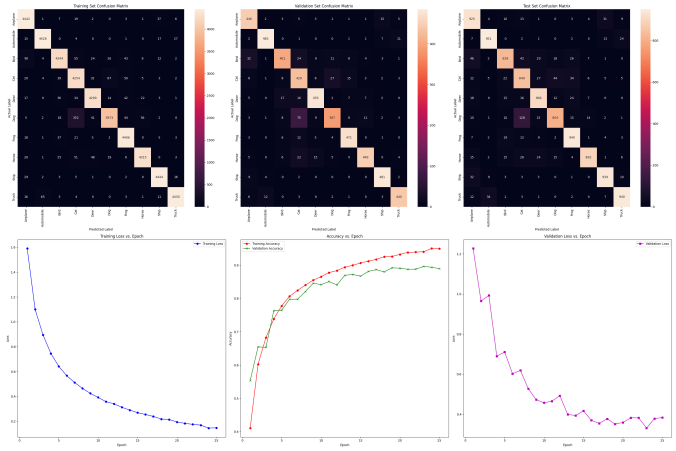

Figure 18: ResNet-18 Model 4 training results.

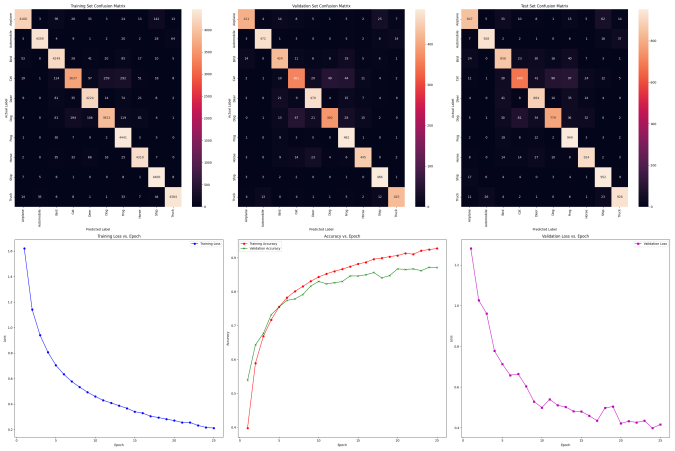

Figure 19: ResNet-18 Model 5 training results.

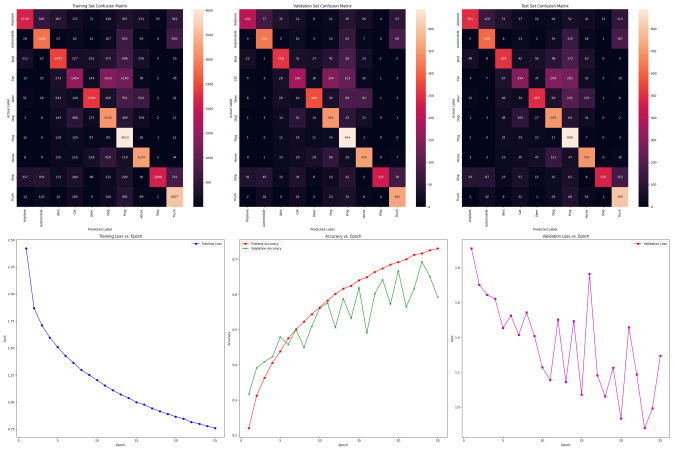

Figure 20: ResNet-18 Model 6 training results.

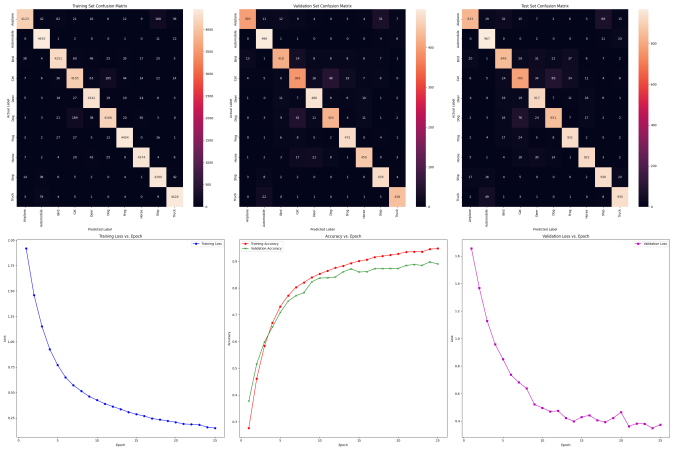

Figure 21: ResNet-34 Model 7 training results.

3) DenseNet: This section gives the results of the DenseNet. Figures 22-22 are the outcomes of the training of 7 different DenseNet models.

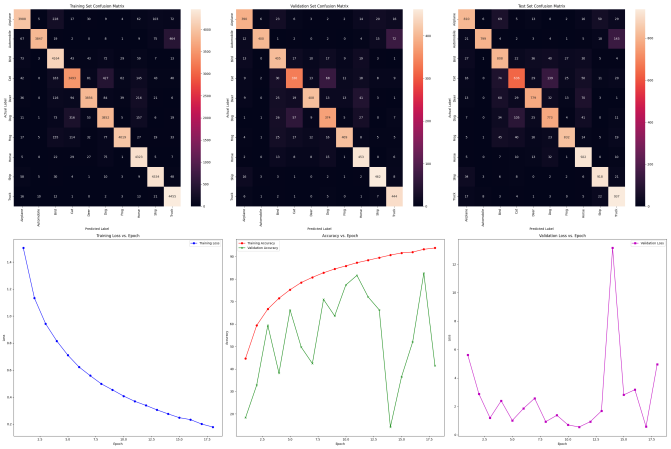

Figure 22: DenseNet Model 1 training results.

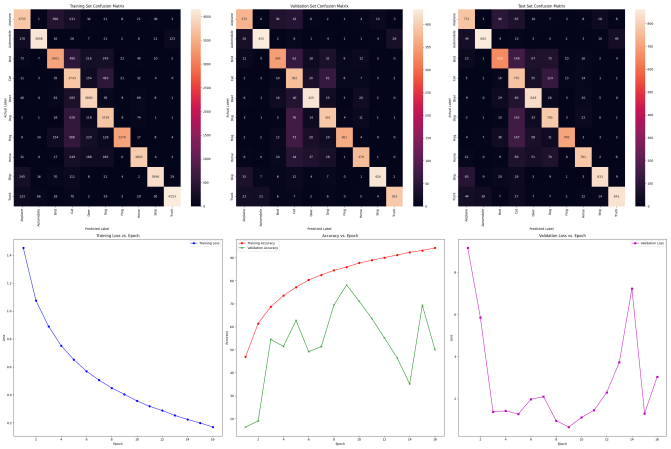

Figure 23: DenseNet Model 2 training results.

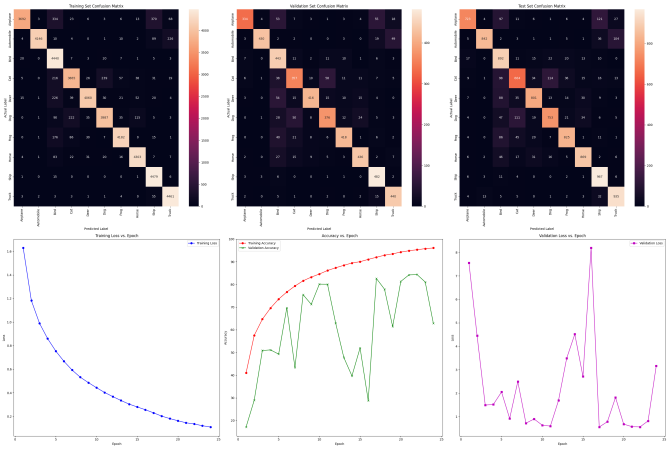

Figure 24: DenseNet Model 3 training results.

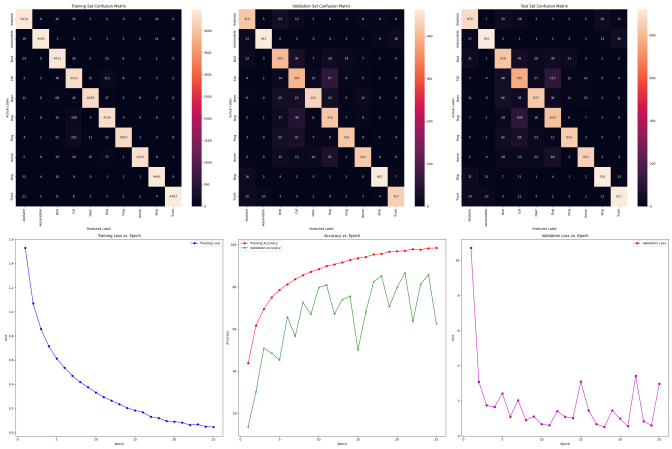

Figure 25: DenseNet Model 4 training results.

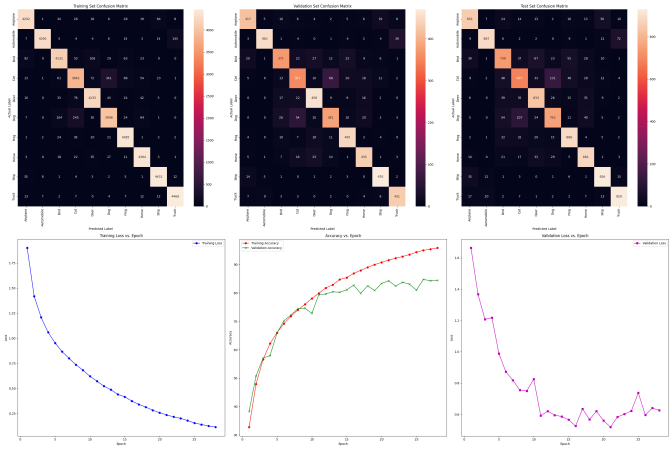

Figure 26: DenseNet Model 5 training results.

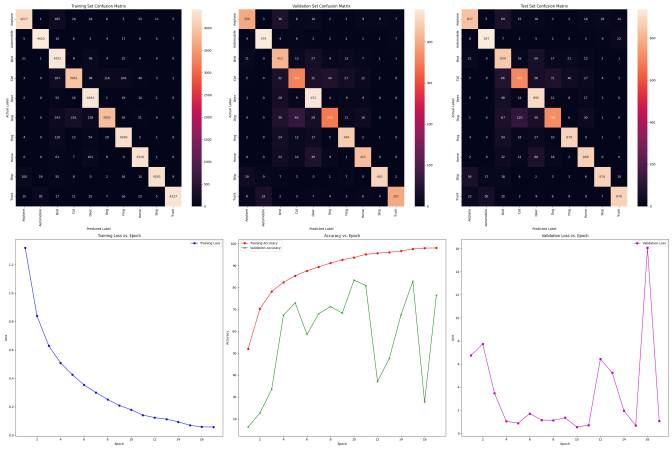

Figure 27: DenseNet Model 6 training results.

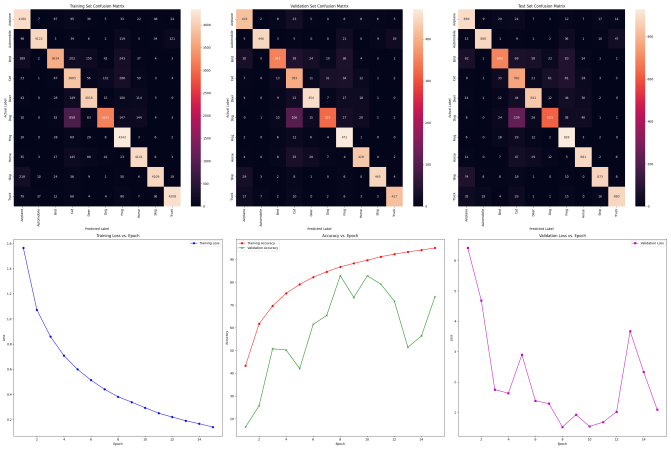

Figure 28: DenseNet Model 7 training results.

4) Inception: This section gives the results of the Inception. Figures 30-36 and Table 7-9 are the outcomes of the training of 4 different Inception models.

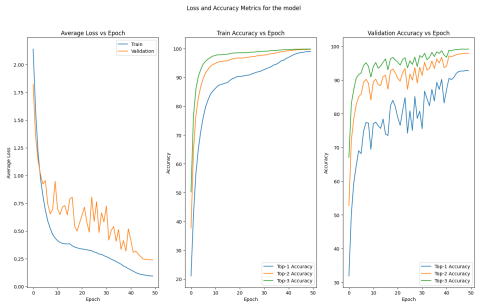

Figure 29: Inception Model 1 training results.

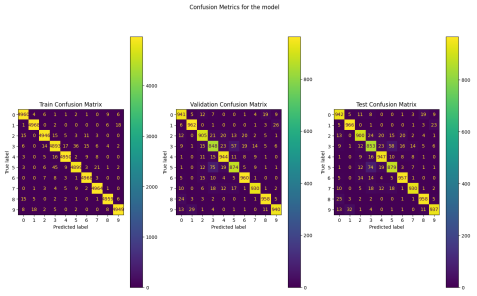

Figure 30: Inception Model 1 confusion matrices.

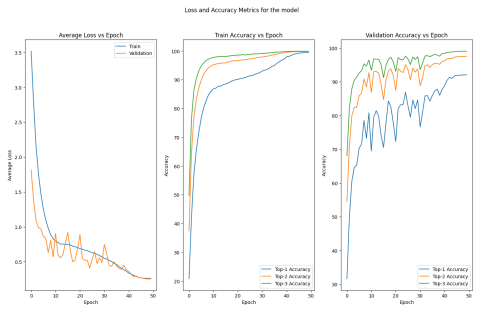

Figure 31: Inception Model 2 training results.

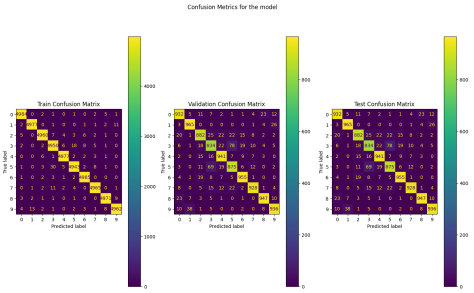

Figure 32: Inception Model 2 confusion matrices.

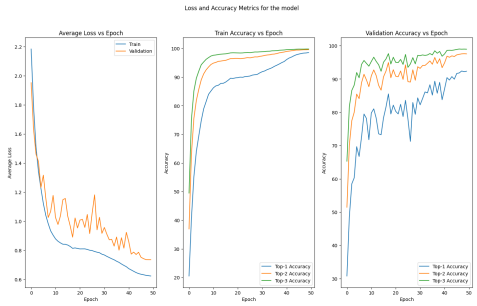

Figure 33: Inception Model 3 training results.

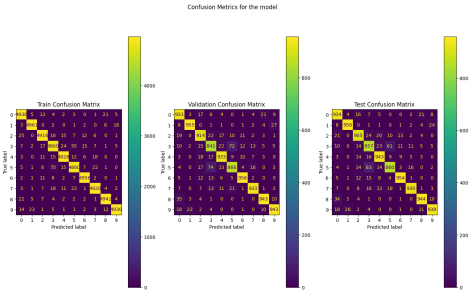

Figure 34: Inception Model 3 confusion matrices.

Model	Parameter Size	Train Top-1 Acc. (%)	Train Top-2 Acc. (%)	Train Top-3 Acc. (%)	Epoch No.	Label Smoothing	Auxiliary Branches
1	5,491,000	99.0785	99.8097	99.9079	50	No	No
2	5,491,000	99.5072	99.8898	99.9399	50	No	Yes
3	5,491,000	98.5757	99.5693	99.7937	50	Yes	No
4	5,491,000	99.1647	99.7496	99.8618	50	Yes	Yes
Model	**Parameter Size**	**Val. Top-1 Acc. (%)**	**Val. Top-2 Acc. (%)**	**Val. Top-3 Acc. (%)**	**Epoch No.**	**Label Smoothing**	**Auxiliary Branches**
1	5,491,000	92.7684	97.9367	99.1987	50	No	No
2	5,491,000	92.0974	97.4259	99.0385	50	No	Yes
3	5,491,000	92.3177	97.5361	98.9383	50	Yes	No
4	5,491,000	91.9671	97.3958	99.0084	50	Yes	Yes
Model	**Parameter Size**	**Test Top-1 Acc. (%)**	**Test Top-2 Acc. (%)**	**Test Top-3 Acc. (%)**	**Epoch No.**	**Label Smoothing**	**Auxiliary Branches**
1	5,491,000	92.8385	97.8966	99.1486	50	No	No
2	5,491,000	92.0974	97.4259	99.0385	50	No	Yes
3	5,491,000	92.3778	97.5361	98.9383	50	Yes	No
4	5,491,000	91.9772	97.4359	99.0184	50	Yes	Yes

Table 7: Detailed Results of Inception Models

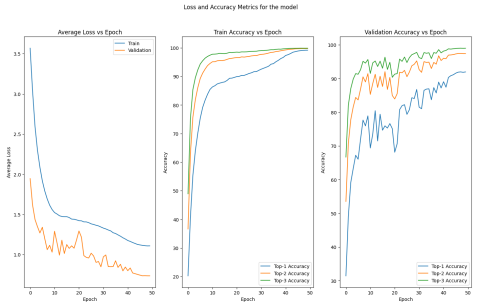

Figure 35: Inception Model 4 training results.

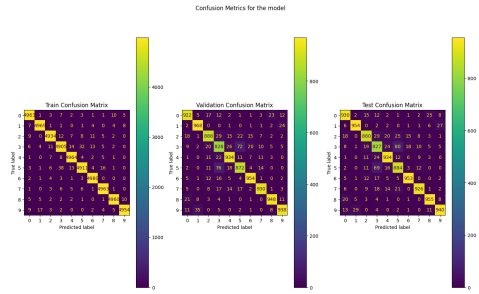

Figure 36: Inception Model 4 confusion matrices.

B. VGGNet Python Implementation

```
# -*- coding: utf-8 -*-
"""Untitled0.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/197GaKS_e1nU-7GIpSq3fu_t7UfMHhL_lO
"""

# Import necessary libraries for neural network construction, optimization, and dataset handling.
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import torch.nn.functional as F
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, random_split
import torch.optim as optim
from torchvision import datasets, transforms
from tqdm import tqdm
from torch.utils.data.sampler import SubsetRandomSampler
from torch.utils.data import Subset

# Define normalization transformation for the CIFAR10 dataset.
normalize = transforms.Normalize(
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2023, 0.1994, 0.2010],
    )

# Compose the transformations: resizing the images, converting them to tensor, and normalizing.
transform = transforms.Compose([
        transforms.Resize((227,227)),
        transforms.ToTensor(),
        normalize,
    ])
# Function to split the dataset by class, ensuring each class is represented in both the train and validation sets.

def split_by_class(dataset, val_percentage):
    # Determine the number of classes and initialize indices for each class
    class_indices = {i: [] for i in range(10)}

    # Iterate through the dataset and assign indices to the respective class
    for idx, (_, label) in enumerate(dataset):
        class_indices[label].append(idx)

    # Initialize train and validation indices
    train_indices, val_indices = [], []

    # Split the indices for each class into train and validation sets
    for indices in class_indices.values():
        np.random.shuffle(indices)
        split = int(len(indices) * val_percentage)
        val_indices.extend(indices[:split])
        train_indices.extend(indices[split:])

    return Subset(dataset, train_indices), Subset(dataset, val_indices)

# Load the train set
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
```

Block	Layer	Type	Parameters
Convolution Block	0	Conv2d	in=3, out=192, kernel=3x3, stride=1, padding=1
	1	BatchNorm2d	features=192, eps=0.001, momentum=0.1
	2	ReLU	
Inception Module	1x1 branch	1x1 Convolution	in=192, out=64, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=192, out=96, kernel=1x1, stride=1
		3x3 Convolution	in=96, out=128, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=192, out=16, kernel=1x1, stride=1
		5x5 Convolution	in=16, out=32, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=192, out=32, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=256, out=128, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=256, out=128, kernel=1x1, stride=1
		3x3 Convolution	in=128, out=192, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=256, out=32, kernel=1x1, stride=1
		5x5 Convolution	in=32, out=96, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=256, out=64, kernel=3x3, stride=1
Max Pooling	0	Max Pooling	kernel=3x3, stride=2, padding=1
Inception Module	1x1 branch	1x1 Convolution	in=480, out=192, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=480, out=96, kernel=1x1, stride=1
		3x3 Convolution	in=96, out=208, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=480, out=16, kernel=1x1, stride=1
		5x5 Convolution	in=16, out=48, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=480, out=64, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=512, out=160, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=512, out=112, kernel=1x1, stride=1
		3x3 Convolution	in=112, out=224, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=512, out=24, kernel=1x1, stride=1
		5x5 Convolution	in=24, out=64, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=512, out=64, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=512, out=192, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=512, out=128, kernel=1x1, stride=1
		3x3 Convolution	in=128, out=256, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=512, out=24, kernel=1x1, stride=1
		5x5 Convolution	in=24, out=64, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=512, out=64, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=512, out=112, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=512, out=144, kernel=1x1, stride=1
		3x3 Convolution	in=144, out=288, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=512, out=32, kernel=1x1, stride=1
		5x5 Convolution	in=32, out=64, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=512, out=64, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=528, out=256, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=528, out=160, kernel=1x1, stride=1
		3x3 Convolution	in=160, out=320, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=528, out=32, kernel=1x1, stride=1
		5x5 Convolution	in=32, out=128, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=528, out=128, kernel=3x3, stride=1

Table 8: Inception Architecture

Max Pooling	0	Max Pooling	kernel=3x3, stride=2, padding=1
Inception Module	1x1 branch	1x1 Convolution	in=832, out=256, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=832, out=160, kernel=1x1, stride=1
		3x3 Convolution	in=160, out=320, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=832, out=32, kernel=1x1, stride=1
		5x5 Convolution	in=32, out=128, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=832, out=128, kernel=3x3, stride=1
Inception Module	1x1 branch	1x1 Convolution	in=832, out=384, kernel=3x3, stride=1, padding=1
	3x3 branch	1x1 Convolution	in=832, out=192, kernel=1x1, stride=1
		3x3 Convolution	in=192, out=384, kernel=3x3, stride=1
	5x5 branch	1x1 Convolution	in=832, out=48, kernel=1x1, stride=1
		5x5 Convolution	in=48, out=128, kernel=5x5, stride=1
	Pooling Branch	3x3 Max Pooling	kernel=3x3, stride=1, padding=1
		1x1 Convolution	in=832, out=128, kernel=3x3, stride=1
Adaptive Average Pooling	0	Adaptive Average Pooling	
Dropout		p=0.2	
Fully Connected Block	0	Linear	in=1024, out=10
	1	ReLU	

Table 9: Inception Architecture Continued

```
# Split the dataset ensuring each class is equally represented
train_data, val_data = split_by_class(trainset, val_percentage=0.1)

# Load the test set
testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

# Load and transform the CIFAR10 dataset for training, validation, and testing.
trainloader = DataLoader(train_data, batch_size=64, shuffle=True)
valloader = DataLoader(val_data, batch_size=64, shuffle=True)
testloader = DataLoader(testset, batch_size=64, shuffle=False)
# Print the number of images in each part of the dataset to verify the splits.

print(len(train_data))
print(len(val_data))
print(len(testset))

# Define the VGG16 neural network architecture with convolutional and fully connected layers.
class VGG16(nn.Module):
    def __init__(self):
        super(VGG16, self).__init__()
        # Define convolutional blocks with batch normalization and max pooling

        self.conv_block1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))

        self.conv_block2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_features=128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))


        self.conv_block3 = self._create_block(128, 256)
        self.conv_block4 = self._create_block(256, 256, pool=True)

        self.conv_block5 = self._create_block(256, 512)
        self.conv_block6 = self._create_block(512, 512, pool=True)

        self.conv_block7 = self._create_block(512, 512)
        self.conv_block8 = self._create_block(512, 512, pool=True)

        self.fc_block = nn.Sequential(
            nn.Dropout(0),
            nn.Linear(7*7*512, 512),
            nn.ReLU())
        self.fc_block1 = nn.Sequential(
            nn.Dropout(0),
            nn.Linear(512, 256),
            nn.ReLU())
        self.fc_block2 = nn.Linear(256, 10)

    def forward(self, x):
        # Apply each convolutional block in sequence
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)
        x = self.conv_block5(x)
        x = self.conv_block6(x)
        x = self.conv_block7(x)
        x = self.conv_block8(x)

        # Flatten the output for the fully connected layer
        x = x.view(x.size(0), -1)

        # Apply the fully connected layers
        x = self.fc_block(x)
        x = self.fc_block1(x)
        x = self.fc_block2(x)
        return x

    def _create_block(self, in_channels, out_channels, pool=False):
        layers = [
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()]
```

```
        if pool:
            layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
        return nn.Sequential(*layers)

# Instantiate the VGG16 model
vgg16_model = VGG16()

# Print the model architecture
print(vgg16_model)
# Install and import torchsummary for a detailed summary of the model.

from torchsummary import summary
summary(vgg16_model, input_size=(3,32,32))

import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, random_split
import torch.optim as optim
from torchvision import datasets, transforms
from tqdm import tqdm

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
# Move the model to the GPU and set up the loss function and optimizer.

model = VGG16()
model.to(device)  # Move the model to the GPU


# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.0)
# Define variables for tracking progress and early stopping.

epochs = 30
train_loss_values = []
train_accuracy_values = []
val_loss_values = []
val_accuracy_values = []

best_val_loss = float('inf')
epochs_no_improve = 0
n_epochs_stop = 10  # number of epochs to continue after no improvement

for epoch in range(epochs):
    # Training phase
    model.train()
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)  # Move data to GPU

        optimizer.zero_grad()

        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total_predictions += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    epoch_loss = running_loss / len(trainloader)
    epoch_accuracy = 100 * correct_predictions / total_predictions
    train_loss_values.append(epoch_loss)
    train_accuracy_values.append(epoch_accuracy)

    # Validation phase
    model.eval()
    val_running_loss = 0.0
    val_correct_predictions = 0
    val_total_predictions = 0

    with torch.no_grad():
        for data in valloader:
            inputs, labels = data[0].to(device), data[1].to(device)

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            val_total_predictions += labels.size(0)
            val_correct_predictions += (predicted == labels).sum().item()

            loss = criterion(outputs, labels)
            val_running_loss += loss.item()

    val_loss = val_running_loss / len(valloader)
    val_accuracy = 100 * val_correct_predictions / val_total_predictions
    val_loss_values.append(val_loss)
    val_accuracy_values.append(val_accuracy)
    print(f'Epoch: {epoch + 1}, Training Loss: {epoch_loss:.3f}, Training Accuracy: {epoch_accuracy:.2f}%, Validation Loss: {val_loss:.3f}, Validation Accuracy: {val_accuracy:.2f}%')
# Plotting training and validation metrics

    # Early stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        if epochs_no_improve == n_epochs_stop:
            print('Early stopping!')
            break


print('Finished Training')

print(train_accuracy_values)

plt.subplot(1, 2, 1)
plt.plot(range(1, epoch+2), train_loss_values, marker='o', linestyle='-', color='b')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Epoch')

plt.subplot(1, 2, 2)
plt.plot(range(1, epoch + 2), train_accuracy_values, marker='o', linestyle='-', color='r')
plt.xlabel('Epoch')
plt.ylabel('Train Accuracy')
plt.title('Train Accuracy vs. Epoch')

plt.tight_layout()
plt.show()

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

```
correct = 0
total = 0
# Set the model to evaluation mode
model.eval()

# Initialize lists to store true and predicted labels
true_labels = []
predicted_labels = []

with torch.no_grad():
    for i, data in enumerate(testloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)  # Move data to GPU

        # Forward pass
        outputs = model(inputs)

        # Get predicted labels
        _, predicted = torch.max(outputs.data, 1)

        # Append true and predicted labels to the lists
        true_labels.extend(labels.cpu().numpy())
        predicted_labels.extend(predicted.cpu().numpy())
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
test_accuracy = correct / total
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

def calculate_confusion_matrix(dataloader):
    model.eval()  # Set the model to evaluation mode
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for data in dataloader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    return confusion_matrix(all_labels, all_preds)

# Calculate confusion matrices for the datasets
train_confusion_matrix = calculate_confusion_matrix(trainloader)
val_confusion_matrix = calculate_confusion_matrix(valloader)
test_confusion_matrix = calculate_confusion_matrix(testloader)


def plot_all(train_cm, val_cm, test_cm, train_loss_values, train_accuracy_values, val_accuracy_values, val_loss_values, epochs):
    class_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

    fig, axes = plt.subplots(2, 3, figsize=(30, 20))  # 2 rows, 3 columns
    # Define plots for confusion matrices and performance metrics

    # Plotting confusion matrices
    sns.heatmap(train_cm, annot=True, fmt="d", ax=axes[0, 0],xticklabels=class_names, yticklabels=class_names)
    axes[0, 0].set_title('Training Set Confusion Matrix')
    axes[0, 0].set_ylabel('Actual Label')
    axes[0, 0].set_xlabel('Predicted Label')

    sns.heatmap(val_cm, annot=True, fmt="d", ax=axes[0, 1],xticklabels=class_names, yticklabels=class_names)
    axes[0, 1].set_title('Validation Set Confusion Matrix')
    axes[0, 1].set_ylabel('Actual Label')
    axes[0, 1].set_xlabel('Predicted Label')

    sns.heatmap(test_cm, annot=True, fmt="d", ax=axes[0, 2],xticklabels=class_names, yticklabels=class_names)
    axes[0, 2].set_title('Test Set Confusion Matrix')
    axes[0, 2].set_ylabel('Actual Label')
    axes[0, 2].set_xlabel('Predicted Label')

    # Plotting training loss
    axes[1, 0].plot(range(1, epochs + 2), train_loss_values, marker='o', linestyle='-', color='b', label='Training Loss')
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Loss')
    axes[1, 0].set_title('Training Loss vs. Epoch')
    axes[1, 0].legend()

    # Plotting training and validation accuracy
    axes[1, 1].plot(range(1, epochs + 2), train_accuracy_values, marker='o', linestyle='-', color='r', label='Training Accuracy')
    axes[1, 1].plot(range(1, epochs + 2), val_accuracy_values, marker='x', linestyle='-', color='g', label='Validation Accuracy')
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Accuracy')
    axes[1, 1].set_title('Accuracy vs. Epoch')
    axes[1, 1].legend()

    # Plotting validation loss
    axes[1, 2].plot(range(1, epochs + 2), val_loss_values, marker='s', linestyle='-', color='m', label='Validation Loss')
    axes[1, 2].set_xlabel('Epoch')
    axes[1, 2].set_ylabel('Loss')
    axes[1, 2].set_title('Validation Loss vs. Epoch')
    axes[1, 2].legend()

    plt.tight_layout()
    plt.show()


plot_all(train_confusion_matrix, val_confusion_matrix, test_confusion_matrix, train_loss_values, train_accuracy_values, val_accuracy_values, val_loss_values, epoch)
```

C. ResNet Python Implementation

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
```

```
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import CIFAR10


class BasicBlock(nn.Module):
    """
    BasicBlock: A basic building block for the ResNet architecture.

    Args:
        in_channels (int): Number of input channels.
        out_channels (int): Number of output channels.
        stride (int, optional): The stride used for the first convolutional layer (default: 1).

    Attributes:
        - conv1: First convolutional layer.
        - bn1: Batch normalization after the first convolution.
        - relu: ReLU activation function.
        - conv2: Second convolutional layer.
        - bn2: Batch normalization after the second convolution.
        - shortcut: Shortcut connection to match dimensions if needed.

    Methods:
        - forward(x): Forward pass through the basic block.

    Notes:
        - The residual connection is added to the output before the final ReLU activation.
        - If the stride is not 1 or the number of input channels doesn't match the output channels,
          a shortcut connection is added to match dimensions.

    References:
        - Original ResNet Paper: https://arxiv.org/abs/1512.03385
        - torchvision.models.resnet implementation: https://pytorch.org/vision/stable/_modules/torchvision/models/resnet.html
    """

    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += self.shortcut(residual)
        out = self.relu(out)
        return out


class ResNet(nn.Module):
    """
    ResNet: Implementation of the ResNet architecture.

    Args:
        block (nn.Module): The basic building block for the ResNet architecture.
        layers (list): List of integers representing the number of blocks in each layer. The length of the list
                       determines the total number of layers in the network.
        num_classes (int, optional): Number of classes in the classification task (default: 10).
        dropout_prob (float, optional): Probability of dropout for regularization (default: 0.5).

    Attributes:
        - conv1: Initial convolutional layer.
        - bn1: Initial batch normalization layer.
        - relu: ReLU activation function.
        - layer1, layer2, layer3, layer4: Residual layers.
        - avg_pool: Adaptive average pooling layer.
        - fc: Fully connected layer for classification.
        - dropout: Dropout layer for regularization.

    Methods:
        - forward(x): Forward pass through the network.

    Notes:
        - This implementation follows the ResNet architecture with configurable layers and block types.
        - The network structure is based on the ResNet18 and ResNet34 variants with different numbers of layers.
        - For ResNet18, set layers=[2, 2, 2, 2].
        - For ResNet34, set layers=[3, 4, 6, 3].

    References:
        - Original ResNet Paper: https://arxiv.org/abs/1512.03385
        - torchvision.models.resnet implementation: https://pytorch.org/vision/stable/_modules/torchvision/models/resnet.html
    """
    def __init__(self, block, layers, num_classes=10, dropout_prob=0.5):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
```

```
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1,1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)
        self.dropout = nn.Dropout(p=dropout_prob)

    def _make_layer(self, block, out_channels, blocks, stride):
        layers = [block(self.in_channels, out_channels, stride)]
        self.in_channels = out_channels * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels, stride=1))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc(x)
        return x

def test_model(model, test_loader, device):
    """
    Evaluate the performance of a trained model on a test dataset.

    Args:
        model (nn.Module): The trained model to be evaluated.
        test_loader (DataLoader): DataLoader for the test dataset.
        device (torch.device): The device (cuda or cpu) on which the evaluation should be performed.

    Notes:
        - The model is set to evaluation mode using `model.eval()`.
        - The function iterates through the test dataset, computes predictions, and calculates accuracy.
        - The accuracy is printed to the console.
        - Ensure that the model and test_loader are properly configured and loaded before calling this function.
        - The device argument determines whether the evaluation is performed on GPU or CPU.

    """
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = correct / total
    print(f'Test Accuracy: {accuracy * 100:.2f}%')

def main():
    """
    Main function for training and evaluating a ResNet model on the CIFAR-10 dataset.

    This function performs the following steps:
    1. Initializes a ResNet model (ResNet18) with specified configurations.
    2. Sets up data transformations and creates training and validation datasets.
    3. Defines the loss function (CrossEntropyLoss) and optimizer (SGD) for training.
    4. Iterates through epochs, training the model and evaluating on the validation set.
    5. Prints training and validation metrics during each epoch.
    6. Evaluates the trained model on the test set and generates confusion matrices.
    7. Plots confusion matrices and training/validation curves.

    Notes:
        - Ensure that the required modules (torch, torchvision, etc.) are installed.
        - The ResNet model, dataset paths, and hyperparameters can be modified as needed.
        - Confusion matrices and training curves are visualized using the plot_all function.
        - The model is trained on GPU if available, otherwise, it falls back to CPU.
    """

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    resnet18 = ResNet(BasicBlock, [2, 2, 2, 2], dropout_prob=0.5).to(device)


    batch_size = 64
    learning_rate = 0.001
    momentum = 0
    epochs = 25

    transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                    transforms.RandomCrop(32, padding=4),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    train_dataset = CIFAR10(root='./data', train=True, download=True, transform=transform)
    torch.manual_seed(0)

    trainset_size = len(train_dataset)
    valset_size = int(0.1 * trainset_size)
    trainset_size = trainset_size - valset_size

    train_subset, val_subset = random_split(train_dataset, [trainset_size, valset_size])

    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True, num_workers=2)
```

```
    valloader = DataLoader(val_subset, batch_size=64, shuffle=False)


    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(resnet18.parameters(), lr=learning_rate, momentum=momentum)

    train_loss_values = []
    train_accuracy_values = []
    val_loss_values = []
    val_accuracy_values = []

    for epoch in range(epochs):

        resnet18.train()
        total_train_correct = 0
        total_train_samples = 0
        total_train_loss = 0.0

        for i, (images, labels) in enumerate(train_loader):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = resnet18(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            _, predicted = torch.max(outputs, 1)
            total_train_samples += labels.size(0)
            total_train_correct += (predicted == labels).sum().item()
            total_train_loss += loss.item()

            if (i + 1) % 100 == 0:
                print(f'Training - Epoch [{epoch+1}/{epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')

        train_accuracy = total_train_correct / total_train_samples
        train_loss = total_train_loss / len(train_loader)
        train_conf_matrix = calculate_confusion_matrix(resnet18, train_loader, device)

        resnet18.eval()
        total_val_correct = 0
        total_val_samples = 0
        total_val_loss = 0.0

        with torch.no_grad():
            for images, labels in valloader:
                images, labels = images.to(device), labels.to(device)
                outputs = resnet18(images)
                loss = criterion(outputs, labels)
                _, predicted = torch.max(outputs, 1)
                total_val_samples += labels.size(0)
                total_val_correct += (predicted == labels).sum().item()
                total_val_loss += loss.item()

        val_accuracy = total_val_correct / total_val_samples
        val_loss = total_val_loss / len(valloader)
        val_conf_matrix = calculate_confusion_matrix(resnet18, valloader, device)

        print(f'Epoch [{epoch+1}/{epochs}] - Training Accuracy: {train_accuracy * 100:.2f}%, Validation Accuracy: {val_accuracy * 100:.2f}%')
        train_loss_values.append(train_loss)
        train_accuracy_values.append(train_accuracy)
        val_loss_values.append(val_loss)
        val_accuracy_values.append(val_accuracy)

    print("Training finished.")

    test_batch_size = 10
    test_dataset = CIFAR10(root='./data', train=False, download=True, transform=transform)
    test_loader = DataLoader(test_dataset, batch_size=test_batch_size, shuffle=False, num_workers=2)

    resnet18.eval()
    test_model(resnet18, test_loader, device)
    test_conf_matrix = calculate_confusion_matrix(resnet18, test_loader, device)
    plot_all(train_conf_matrix, val_conf_matrix, test_conf_matrix, train_loss_values, train_accuracy_values,
             val_accuracy_values, val_loss_values, epochs, model_no=18, show=True)


if __name__ == '__main__':
    main()
```

D. DenseNet Python Implementation

```
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, random_split
import torch.optim as optim
from torchvision import datasets, transforms
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import copy
from sklearn.metrics import confusion_matrix
from tqdm import tqdm
import csv
from torchsummary import summary
from collections import OrderedDict


# Transition Layer to connect Dense Layers
class Transition(nn.Sequential):
    """
    Transition layer used in DenseNet between dense blocks.

    This layer performs convolution and pooling operations to reduce the size of the feature maps.

    Parameters:
    in_channels (int): Number of input channels.
    compression_factor (float): Compression factor to reduce the feature map size.
    version (str): Specifies the version of the transition layer. 'Default' version
                   keeps the number of channels unchanged, while 'C' applies compression.
    """
```

```
    def __init__(self, in_channels, compression_factor=0, version='Default'):
        super(Transition, self).__init__()
        if version == 'Default':
            self.add_module('batchnorm', nn.BatchNorm2d(in_channels))
            self.add_module('conv1x1', nn.Conv2d(in_channels, in_channels, kernel_size=1, bias=False))
            self.add_module('avgpool2x2', nn.AvgPool2d(kernel_size=2))

        if version == 'C':
            output_channel = int(in_channels * compression_factor // 1)

            self.add_module('batchnorm', nn.BatchNorm2d(in_channels))
            self.add_module('conv1x1', nn.Conv2d(in_channels, output_channel, kernel_size=1, bias=False))
            self.add_module('avgpool2x2', nn.AvgPool2d(kernel_size=2))

# The Dense Block is the main contribution of the paper
class DenseBlock(nn.Module):
    """
    DenseBlock in DenseNet, consisting of batch normalization, ReLU activation, and convolution.

    This block forms the core building block of DenseNet, with each DenseBlock layer receiving additional feature maps from all preceding layers.

    Parameters:
    in_channels (int): Number of input channels.
    growth_rate (int): How many filters to add per dense layer (growth rate).
    version (str): Version of the DenseBlock. 'Default' is the standard block,
                   while 'B' includes bottleneck layers.
    """
    def __init__(self, in_channels, growth_rate, version='Default'):
        super(DenseBlock, self).__init__()

        if version == 'Default':
            # Default Dense Block
            self.dense_block = nn.Sequential(OrderedDict([
                ('batchnorm0', nn.BatchNorm2d(in_channels)),
                ('relu0', nn.ReLU(inplace=True)),
                ('conv3x3', nn.Conv2d(in_channels, growth_rate, kernel_size=3, padding=1, bias=False))
            ]))

        if version == 'B':
            # B type Dense Block
            # Bottleneck layers
            self.dense_block = nn.Sequential(OrderedDict([
                ('batchnorm0', nn.BatchNorm2d(in_channels)),
                ('relu0', nn.ReLU(inplace=True)),
                ('conv1x1', nn.Conv2d(in_channels, 4 * growth_rate, kernel_size=1, bias=False)),
                ('batchnorm1', nn.BatchNorm2d(4 * growth_rate)),
                ('relu1', nn.ReLU(inplace=True)),
                ('conv3x3', nn.Conv2d(4 * growth_rate, growth_rate, kernel_size=3, padding=1, bias=False))
            ]))
    def forward(self, x):
        """
        Forward pass of the DenseBlock.

        Concatenates the input with the output feature maps of the dense block.

        Parameters:
        x (Tensor): Input tensor to the DenseBlock.

        Returns:
        Tensor: Concatenated output after passing through the block.
        """
        # Forwards the convolutional layers, and concatenate it with the input.
        return torch.cat([x, self.dense_block(x)], 1)

# The class below defines cascaded Dense Blocks
class DenseLayer(nn.Module):
    """
    Layer of DenseNet that consists of multiple DenseBlocks.

    Parameters:
    in_channels (int): Number of input channels.
    growth_rate (int): Growth rate of the dense blocks.
    num_blocks (int): Number of dense blocks in the layer.
    version (str): Version of DenseBlocks used in the layer.
    """
    def __init__(self, in_channels, growth_rate, num_blocks, version='Default'):
        super(DenseLayer, self).__init__()

        blocks = []
        for i in range(num_blocks):
            if version == 'Default':
                block = DenseBlock(in_channels + i * growth_rate, growth_rate, version)
            elif version == 'B':
                block = DenseBlock(in_channels + i * growth_rate, growth_rate, version)
            blocks.append(block)

        self.blocks = nn.Sequential(*blocks)

    def forward(self, x):
        """
        Forward pass of the DenseLayer.

        Parameters:
        x (Tensor): Input tensor to the DenseLayer.

        Returns:
        Tensor: Output tensor after passing through all dense blocks.
        """
        for block in self.blocks:
            x = block(x)
        return x

# The main network connecting the above methods.
class DenseNet(nn.Module):
    """
    DenseNet architecture for image classification.
```

```
    Parameters:
    growth_rate (int): Number of filters to add per dense layer (growth rate).
    compression_factor (float): Compression factor for Transition layers.
    layer_widths (List[int]): Number of blocks in each dense layer.
    num_classes (int): Number of output classes for classification.
    dropout_rate (float): Dropout rate for the final fully connected layer.
    version (str): Version of DenseNet ('Default', 'B', 'BC', etc.).
    """
    def __init__(self, growth_rate, compression_factor, layer_widths, num_classes, dropout_rate=0.0, version='Default'):
        super(DenseNet, self).__init__()

        # Assuming the input is RGB images
        num_channels = 3


        dense_version = 'Default'
        transition_version = 'Default'
        comp_fac = 1

        if (version == 'B') or (version == 'BC'):
            dense_version = 'B'

        if (version == 'BC') or (version == 'C'):
            transition_version = 'C'
            comp_fac = compression_factor

        # Initial convolution
        self.features = nn.Sequential(OrderedDict([
            ('conv0', nn.Conv2d(num_channels, growth_rate * 2, kernel_size=3, stride=1, padding=1, bias=False)),
            ('norm0', nn.BatchNorm2d(growth_rate * 2)),
            ('relu0', nn.ReLU(inplace=True))
        ]))

        num_channels = growth_rate * 2

        # Dense blocks and transition layers
        for i, num_blocks in enumerate(layer_widths):
            # Add DenseBlock
            dense_block = DenseLayer(num_channels, growth_rate, num_blocks, dense_version)
            self.features.add_module('denseblock%d' % (i + 1), dense_block)

            num_channels += num_blocks * growth_rate

            if i != len(layer_widths) - 1:
                # Add Transition Layer
                trans_layer = Transition(num_channels, comp_fac, transition_version)
                self.features.add_module('transition%d' % (i + 1), trans_layer)
                num_channels = int(num_channels * comp_fac)

        # Final batch norm
        self.features.add_module('norm5', nn.BatchNorm2d(num_channels))

        # Global average pooling
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))

        # Classifier
        self.classifier = nn.Sequential(OrderedDict([
            ('dropout', nn.Dropout(p=dropout_rate)),
            ('fc', nn.Linear(num_channels, num_classes))
        ]))

    def _initialize_weights(self):
        """
        Initialize weights of the network using specific schemes for different types of layers.
        """
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        """
        Forward pass of the DenseNet.

        Parameters:
        x (Tensor): Input tensor to the network.

        Returns:
        Tensor: Output tensor after classification.
        """
        x = self.features(x)
        x = self.global_avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x


def train_model(model, trainloader, valloader, optimizer, criterion, device, epochs, patience, model_no):
    """
    Train a model with given data and hyperparameters, implementing early stopping.

    Parameters:
    model (nn.Module): The neural network model to be trained.
    trainloader (DataLoader): DataLoader for the training set.
    valloader (DataLoader): DataLoader for the validation set.
    optimizer (Optimizer): Optimizer for the training process.
    criterion (Loss Function): Loss function used during training.
    device (str): Device to train the model on ('cpu' or 'cuda').
    epochs (int): Total number of epochs for training.
    patience (int): Patience for early stopping (number of epochs to wait without improvement).
    model_no (int): Identifier for the model, used for saving metrics.

    Returns:
```

```
    tuple: Tuple containing the trained model, and lists of training and validation losses and accuracies.
    """
    model.to(device)

    train_loss_values = []
    train_accuracy_values = []
    val_loss_values = []
    val_accuracy_values = []

    best_val_loss = float('inf')
    best_model_params = copy.deepcopy(model.state_dict())  # Save the initial model parameters
    epochs_no_improve = 0

    epoch_metrics = []
    for epoch in range(epochs):
        # Training phase
        model.train()
        running_loss = 0.0
        correct_predictions = 0
        total_predictions = 0
        for i, (inputs, labels) in enumerate(trainloader):
            inputs, labels = inputs.to(device), labels.to(device)  # Move data to GPU

            optimizer.zero_grad()

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total_predictions += labels.size(0)
            correct_predictions += (predicted == labels).sum().item()

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            if (i + 1) % 100 == 0:
                print(f'Iteration {i+1}: Loss = {running_loss / (i+1):.4f}')


        epoch_loss = running_loss / len(trainloader)
        epoch_accuracy = 100 * correct_predictions / total_predictions
        train_loss_values.append(epoch_loss)
        train_accuracy_values.append(epoch_accuracy)

        # Validation phase
        model.eval()
        val_running_loss = 0.0
        val_correct_predictions = 0
        val_total_predictions = 0

        with torch.no_grad():
            for inputs, labels in valloader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                val_total_predictions += labels.size(0)
                val_correct_predictions += (predicted == labels).sum().item()

                loss = criterion(outputs, labels)
                val_running_loss += loss.item()

        val_loss = val_running_loss / len(valloader)
        val_accuracy = 100 * val_correct_predictions / val_total_predictions
        val_loss_values.append(val_loss)
        val_accuracy_values.append(val_accuracy)

        print(f'Epoch {epoch+1}/{epochs}, Train Loss: {epoch_loss:.4f}, Train Acc: {epoch_accuracy:.2f}%, Val Loss: {val_loss:.4f}, Val Acc: {val_accuracy:.2f}%')
        # Save metrics for the current epoch
        epoch_metrics.append([epoch + 1, epoch_loss, epoch_accuracy, val_loss, val_accuracy])

        # Early stopping check
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_params = copy.deepcopy(model.state_dict())
            epochs_no_improve = 0
        else:
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping!')
                break

    model.load_state_dict(best_model_params)
    print('Finished Training')

    # Write metrics to a CSV file
    csv_file = f'training_metrics_{model_no}.csv'
    with open(csv_file, 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Epoch', 'Train Loss', 'Train Accuracy', 'Validation Loss', 'Validation Accuracy'])
        writer.writerows(epoch_metrics)

    return model, train_loss_values, train_accuracy_values, val_loss_values, val_accuracy_values


def cal_acc(model, testloader, device):
    """
    Calculate the accuracy of a trained model on a given dataset.

    Parameters:
    model (nn.Module): The trained neural network model.
    testloader (DataLoader): DataLoader for the test set.
    device (str): Device on which the model is loaded ('cpu' or 'cuda').

    Returns:
    tuple: Tuple containing the test accuracy and the predicted labels.
    """
    correct = 0
    total = 0
    # Set the model to evaluation mode
    model.to(device)
    model.eval()

    # Initialize lists to store true and predicted labels
    true_labels = []
    predicted_labels = []
```

```
    with torch.no_grad():
        for i, data in enumerate(testloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)  # Move data to GPU

            # Forward pass
            outputs = model(inputs)

            # Get predicted labels
            _, predicted = torch.max(outputs.data, 1)

            # Append true and predicted labels to the lists
            true_labels.extend(labels.cpu().numpy())
            predicted_labels.extend(predicted.cpu().numpy())
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    test_accuracy = correct / total

    return test_accuracy, predicted_labels


def calculate_confusion_matrix(model, dataloader, device):
    """
    Compute the confusion matrix for a model's predictions on a dataset.

    Parameters:
    model (nn.Module): The trained neural network model.
    dataloader (DataLoader): DataLoader for the dataset.
    device (str): Device on which the model is loaded ('cpu' or 'cuda').

    Returns:
    ndarray: Confusion matrix of the model's predictions.
    """
    model.eval()  # Set the model to evaluation mode
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for data in dataloader:
            inputs, labels = data[0].to(device), data[1].to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    return confusion_matrix(all_labels, all_preds)



def plot_all(train_cm, val_cm, test_cm, train_loss_values, train_accuracy_values, val_accuracy_values, val_loss_values, epochs, model_no, show=False):
    """
    Plot and save various training and evaluation metrics including confusion matrices and accuracy/loss curves.

    Parameters:
    train_cm (ndarray): Confusion matrix for the training set.
    val_cm (ndarray): Confusion matrix for the validation set.
    test_cm (ndarray): Confusion matrix for the test set.
    train_loss_values (list): List of training loss values for each epoch.
    train_accuracy_values (list): List of training accuracy values for each epoch.
    val_accuracy_values (list): List of validation accuracy values for each epoch.
    val_loss_values (list): List of validation loss values for each epoch.
    epochs (int): Number of epochs.
    model_no (int): Identifier for the model, used for saving the plot.
    show (bool): If True, display the plot. Defaults to False.
    """
    class_names = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

    fig, axes = plt.subplots(2, 3, figsize=(30, 20))  # 2 rows, 3 columns

    # Plotting confusion matrices
    sns.heatmap(train_cm, annot=True, fmt="d", ax=axes[0, 0],xticklabels=class_names, yticklabels=class_names)
    axes[0, 0].set_title('Training Set Confusion Matrix')
    axes[0, 0].set_ylabel('Actual Label')
    axes[0, 0].set_xlabel('Predicted Label')

    sns.heatmap(val_cm, annot=True, fmt="d", ax=axes[0, 1],xticklabels=class_names, yticklabels=class_names)
    axes[0, 1].set_title('Validation Set Confusion Matrix')
    axes[0, 1].set_ylabel('Actual Label')
    axes[0, 1].set_xlabel('Predicted Label')

    sns.heatmap(test_cm, annot=True, fmt="d", ax=axes[0, 2],xticklabels=class_names, yticklabels=class_names)
    axes[0, 2].set_title('Test Set Confusion Matrix')
    axes[0, 2].set_ylabel('Actual Label')
    axes[0, 2].set_xlabel('Predicted Label')

    # Plotting training loss
    axes[1, 0].plot(range(1, epochs + 1), train_loss_values, marker='o', linestyle='-', color='b', label='Training Loss')
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Loss')
    axes[1, 0].set_title('Training Loss vs. Epoch')
    axes[1, 0].legend()

    # Plotting training and validation accuracy
    axes[1, 1].plot(range(1, epochs + 1), train_accuracy_values, marker='o', linestyle='-', color='r', label='Training Accuracy')
    axes[1, 1].plot(range(1, epochs + 1), val_accuracy_values, marker='x', linestyle='-', color='g', label='Validation Accuracy')
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Accuracy')
    axes[1, 1].set_title('Accuracy vs. Epoch')
    axes[1, 1].legend()

    # Plotting validation loss
    axes[1, 2].plot(range(1, epochs + 1), val_loss_values, marker='s', linestyle='-', color='m', label='Validation Loss')
    axes[1, 2].set_xlabel('Epoch')
```

```
    axes[1, 2].set_ylabel('Loss')
    axes[1, 2].set_title('Validation Loss vs. Epoch')
    axes[1, 2].legend()

    plt.tight_layout()
    if show:
        plt.show()
    plt.savefig(f'model_results_{model_no}.png')  # Save the figure to a file

# Check whether the GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Using device:", device)


# Defining the models.
model1 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], dropout_rate=0.5, num_classes=10)
criterion1 = nn.CrossEntropyLoss()
optimizer1 = optim.SGD(model1.parameters(), lr=0.01, momentum=0.0)

model2 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], num_classes=10)
criterion2 = nn.CrossEntropyLoss()
optimizer2 = optim.SGD(model2.parameters(), lr=0.01, momentum=0.0)

model3 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], dropout_rate=0.5, num_classes=10)
criterion3 = nn.CrossEntropyLoss()
optimizer3 = optim.SGD(model3.parameters(), lr=0.1, momentum=0.0)

model4 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], num_classes=10)
criterion4 = nn.CrossEntropyLoss()
optimizer4 = optim.SGD(model4.parameters(), lr=0.1, momentum=0.0)

model5 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], num_classes=10)
criterion5 = nn.CrossEntropyLoss()
optimizer5 = optim.SGD(model5.parameters(), lr=0.1, momentum=0.9)

model6 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[6, 18, 12], num_classes=10, version='BC')
criterion6 = nn.CrossEntropyLoss()
optimizer6 = optim.SGD(model5.parameters(), lr=0.1, momentum=0.0)

model7 = DenseNet(growth_rate=24, compression_factor=0.5, layer_widths=[12, 24, 18], num_classes=10)
criterion7 = nn.CrossEntropyLoss()
optimizer7 = optim.SGD(model7.parameters(), lr=0.1, momentum=0)



transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)


# Set the seed for reproducibility
torch.manual_seed(0)

# Size of trainset
trainset_size = len(trainset)

# Calculate validation set size (10% of trainset size)
valset_size = int(0.1 * trainset_size)

# Calculate train set size by subtracting validation set size from total trainset size
trainset_size = trainset_size - valset_size

# Splitting the dataset
train_subset, val_subset = random_split(trainset, [trainset_size, valset_size])

# Create data loaders for each set
trainloader = DataLoader(train_subset, batch_size=64, shuffle=True)
valloader = DataLoader(val_subset, batch_size=64, shuffle=False)

testset = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=False)

# Defining model, criterions, and optimizers.
models = [model1, model2, model3, model4, model5, model6, model7]
criterions = [criterion1, criterion2, criterion3, criterion4, criterion5, criterion6, criterion7]
optimizers = [optimizer1, optimizer2, optimizer3, optimizer4, optimizer5, optimizer6, optimizer7]


for i in range(len(models)):
    model = models[i]
    criterion = criterions[i]
    optimizer = optimizers[i]

    # Prints the summary of the model. Gives information about the model parameters, layers etc.
    summary(model, input_size=(3,32,32))

    # Train the model according to the function called "train_model"
    trained_model, train_loss_values, train_accuracy_values, val_loss_values, val_accuracy_values =  train_model(model, trainloader, valloader, optimizer, criterion, device, 60, 7, i+1)

    # Saves the pretrained model if needed later
    torch.save(trained_model.state_dict(), f'model_{i+1}_state.pth')

    # Calculates accuracy scores and predicted labels.
    train_acc, train_pred = cal_acc(trained_model, trainloader, device)
    val_acc, val_pred = cal_acc(trained_model, valloader, device)
    test_acc, test_pred = cal_acc(trained_model, testloader, device)

    print(f'train accuracy: {train_acc}\nvalidation accuracy: {test_acc}')

    # Calculates confusion matrix for the train, validation and test set.
```

```
    train_cm = calculate_confusion_matrix(trained_model, trainloader, device)
    val_cm = calculate_confusion_matrix(trained_model, valloader, device)
    test_cm = calculate_confusion_matrix(trained_model, testloader, device)

    # Plots confusion matrices, loss and accuracy figures.
    plot_all(train_cm, val_cm, test_cm, train_loss_values, train_accuracy_values, val_accuracy_values, val_loss_values, len(train_accuracy_values), i+1, show=False)
```

E. Inception Python Implementation

```
import os
from argparse import ArgumentParser

import torch

from pytorch_lightning import Trainer, seed_everything
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning.loggers import WandbLogger, TensorBoardLogger

from data import CIFAR10Data
from module import CIFAR10Module



def main(args):

    seed_everything(0)
    os.environ["CUDA_VISIBLE_DEVICES"] = args.gpu_id


    logger = TensorBoardLogger("cifar10", name="inception")

    checkpoint = ModelCheckpoint(monitor="acc/val", mode="max", save_last=False)


    trainer = Trainer(
        fast_dev_run=bool(args.dev),
        logger=logger if not bool(args.dev + args.test_phase) else None,
        gpus=-1,
        deterministic=True,
        weights_summary=None,
        log_every_n_steps=50,
        check_val_every_n_epoch=1,
        max_epochs=args.max_epochs,
        checkpoint_callback=checkpoint,
        precision=args.precision,
    )

    model = CIFAR10Module(args)
    data = CIFAR10Data(args)

    if bool(args.pretrained):
        state_dict = os.path.join("state_dicts", "inception" + ".pt"
        )
        model.model.load_state_dict(torch.load(state_dict))

    if bool(args.test_phase):
        trainer.test(model, data.test_dataloader())
    else:
        trainer.fit(model, data)
        trainer.test()

    model.save_metrics()



if __name__ == "__main__":
    parser = ArgumentParser()

    # PROGRAM level args
    parser.add_argument("--data_dir", type=str, default="data/cifar10")
    parser.add_argument("--download_weights", type=int, default=0, choices=[0, 1])
    parser.add_argument("--test_phase", type=int, default=0, choices=[0, 1])
    parser.add_argument("--dev", type=int, default=0, choices=[0, 1])
    parser.add_argument(
        "--logger", type=str, default="tensorboard", choices=["tensorboard", "wandb"]
    )

    # TRAINER args
    parser.add_argument("--pretrained", type=int, default=0, choices=[0, 1])

    parser.add_argument("--precision", type=int, default=32, choices=[16, 32])
    parser.add_argument("--batch_size", type=int, default=256)
    parser.add_argument("--max_epochs", type=int, default=50)
    parser.add_argument("--num_workers", type=int, default=8)
    parser.add_argument("--gpu_id", type=str, default="0")

    parser.add_argument("--learning_rate", type=float, default=1e-2)
    parser.add_argument("--weight_decay", type=float, default=1e-2)

    parser.add_argument("--aux_branches", type=bool, default=False)
    parser.add_argument("--label_smoothing", type=bool, default=False)

    args = parser.parse_args()
    main(args)
```

```
from typing import Any, List
import pytorch_lightning as pl
import torch
from torchmetrics import Accuracy

import numpy as np
import pickle


from label_smoothing_loss import LabelSmoothingLoss
from model.inception import InceptionModel

from schduler import WarmupCosineLR
```

```
class CIFAR10Module(pl.LightningModule):
    def __init__(self, hparams):
        super().__init__()
        self.hparams = hparams

        self.label_smoothing = self.hparams.label_smoothing
        self.aux_branches = self.hparams.aux_branches

        #Loss function selection, with label smoothing or without.
        if self.label_smoothing:
            self.criterion = LabelSmoothingLoss(10)
        else:
            self.criterion = torch.nn.CrossEntropyLoss()


        self.accuracy = Accuracy()
        self.accuracy2 = Accuracy(top_k=2)
        self.accuracy3 = Accuracy(top_k=3)


        self.model = InceptionModel(aux_logits=self.aux_branches)


        self.epoch_metrics_train = []
        self.epoch_metrics_val = []

        self.cur_epoch_train_metrics = []
        self.cur_epoch_val_metrics = []

        self.cur_train_preds = []
        self.cur_val_preds = []
        self.cur_test_preds = []

        self.train_preds = []
        self.val_preds = []
        self.test_preds = []



    def forward(self, batch):
        images, labels = batch
        predictions = self.model(images)

        #This is to handle multiple outputs if auxiliary branches exist.
        if self.training and self.aux_branches:
            loss_main = self.criterion(predictions[0], labels)
            loss_aux_2 = self.criterion(predictions[1], labels)
            loss_aux_1 = self.criterion(predictions[2], labels)



            accuracy = self.accuracy(predictions[0], labels)
            accuracy2 = self.accuracy2(predictions[0], labels)
            accuracy3 = self.accuracy3(predictions[0], labels)

            predictions = predictions[0]

        else:
            loss_main = self.criterion(predictions, labels)
            loss_aux_2 = 0
            loss_aux_1 = 0

            accuracy = self.accuracy(predictions, labels)
            accuracy2 = self.accuracy2(predictions, labels)
            accuracy3 = self.accuracy3(predictions, labels)

            predictions = predictions


        #The losses from the auxiliary branches are added with discount weights, the values are 0 if they do not exist.
        loss = loss_main + 0.3 * loss_aux_2 + 0.3 * loss_aux_1

        return loss, accuracy * 100, accuracy2 * 100, accuracy3 * 100, predictions



    def on_train_epoch_start(self):

        self.cur_train_preds = []
        self.cur_val_preds = []
        self.cur_test_preds = []


    def training_step(self, batch, batch_nb):
        loss, accuracy, acc2, acc3, preds = self.forward(batch)
        self.log("loss/train", loss)
        self.log("acc/train", accuracy)

        metrics = {
                'loss':loss,
                'acc1':accuracy,
                'acc2':acc2,
                'acc3':acc3
            }

        self.cur_epoch_train_metrics.append(metrics)


        pred_probs = preds.cpu().detach().numpy()
        preds = np.argmax(pred_probs, axis=1)

        outs = {
            'preds':preds,
            'pred_probs':pred_probs,
            'labels':batch[1]
        }

        self.cur_train_preds.append(outs)


        return loss



    def validation_step(self, batch, batch_nb):
        loss, accuracy, acc2, acc3, preds = self.forward(batch)
        self.log("loss/val", loss)
        self.log("acc/val", accuracy)


        metrics = {
                'loss':loss,
                'acc1':accuracy,
                'acc2':acc2,
                'acc3':acc3
```

```
            }

        self.cur_epoch_val_metrics.append(metrics)



        pred_probs = preds.cpu().detach().numpy()
        preds = np.argmax(pred_probs, axis=1)

        outs = {
            'preds':preds,
            'pred_probs':pred_probs,
            'labels':batch[1]
        }

        self.cur_val_preds.append(outs)


    def on_train_epoch_end(self, outputs):

        epoch_loss = 0
        acc1 = 0
        acc2 = 0
        acc3 = 0
        for metric in self.cur_epoch_train_metrics:
            epoch_loss += metric['loss']
            acc1 += metric['acc1']
            acc2 += metric['acc2']
            acc3 += metric['acc3']
        epoch_loss /= len(self.cur_epoch_train_metrics)
        acc1 /= len(self.cur_epoch_train_metrics)
        acc2 /= len(self.cur_epoch_train_metrics)
        acc3 /= len(self.cur_epoch_train_metrics)

        epoch_metrics = {
            'loss':epoch_loss,
            'acc1':acc1,
            'acc2':acc2,
            'acc3':acc3
        }

        self.cur_epoch_train_metrics = []
        self.epoch_metrics_train.append(epoch_metrics)


        preds_all = np.empty(1)
        pred_probs_all = np.empty((1, 10))
        labels_all = np.empty(1)
        for out in self.cur_train_preds:
            preds = out['preds']
            pred_probs = out['pred_probs']
            labels = out['labels'].cpu().detach().numpy()

            preds_all = np.concatenate((preds_all, preds))
            pred_probs_all = np.concatenate((pred_probs_all, pred_probs))
            labels_all = np.concatenate((labels_all, labels))


        preds_all = preds_all[1:]
        pred_probs_all = pred_probs_all[1:,:]
        labels_all = labels_all[1:]


        out = {
            'preds':preds_all,
            'pred_probs':pred_probs_all,
            'labels':labels_all
        }

        self.train_preds = out



    def on_validation_epoch_end(self):

        epoch_loss = 0
        acc1 = 0
        acc2 = 0
        acc3 = 0
        for metric in self.cur_epoch_val_metrics:
            epoch_loss += metric['loss']
            acc1 += metric['acc1']
            acc2 += metric['acc2']
            acc3 += metric['acc3']
        epoch_loss /= len(self.cur_epoch_val_metrics)
        acc1 /= len(self.cur_epoch_val_metrics)
        acc2 /= len(self.cur_epoch_val_metrics)
        acc3 /= len(self.cur_epoch_val_metrics)

        epoch_metrics = {
            'loss':epoch_loss,
            'acc1':acc1,
            'acc2':acc2,
            'acc3':acc3
        }

        self.cur_epoch_val_metrics = []
        self.epoch_metrics_val.append(epoch_metrics)



        preds_all = np.empty(1)
        pred_probs_all = np.empty((1, 10))
        labels_all = np.empty(1)
        for out in self.cur_val_preds:
            preds = out['preds']
            pred_probs = out['pred_probs']
            labels = out['labels'].cpu().detach().numpy()

            preds_all = np.concatenate((preds_all, preds))
            pred_probs_all = np.concatenate((pred_probs_all, pred_probs))
            labels_all = np.concatenate((labels_all, labels))


        preds_all = preds_all[1:]
        pred_probs_all = pred_probs_all[1:,:]
        labels_all = labels_all[1:]


        out = {
            'preds':preds_all,
            'pred_probs':pred_probs_all,
            'labels':labels_all
        }

        self.val_preds = out



    def on_test_epoch_end(self):

        preds_all = np.empty(1)
        pred_probs_all = np.empty((1, 10))
        labels_all = np.empty(1)
        for out in self.cur_test_preds:
            preds = out['preds']
```

```
            pred_probs = out['pred_probs']
            labels =
            ↪  out['labels'].cpu().detach().numpy()

            preds_all = np.concatenate((preds_all,
            ↪  preds))
            pred_probs_all =
            ↪  np.concatenate((pred_probs_all,
            ↪  pred_probs))
            labels_all =
            ↪  np.concatenate((labels_all, labels))


        preds_all = preds_all[1:]
        pred_probs_all = pred_probs_all[1:,:]
        labels_all = labels_all[1:]


        out = {
            'preds':preds_all,
            'pred_probs':pred_probs_all,
            'labels':labels_all
        }

        self.test_preds = out



    def get_metrics(self):

        train_metrics = self.epoch_metrics_train
        val_metrics = self.epoch_metrics_val[1:]

        train_outs = self.train_preds
        val_outs = self.val_preds
        test_outs = self.test_preds


        return train_metrics, val_metrics,
        ↪  train_outs, val_outs, test_outs


    def save_metrics(self):

        train_metrics, val_metrics, train_outs,
        ↪  val_outs, test_outs = self.get_metrics()

        save = {
            'train_metrics':train_metrics,
            'val_metrics':val_metrics,
            'train_outs':train_outs,
            'val_outs':val_outs,
            'test_outs':test_outs
        }

        with open('metrics/saved_metrics.pkl',
        ↪  'wb') as f:
            pickle.dump(save, f)


    def load_metrics():

        with open('metrics/saved_metrics.pkl',
        ↪  'rb') as f:
            loaded_metrics = pickle.load(f)

        return loaded_metrics['train_metrics'],
        ↪  loaded_metrics['val_metrics'],
        ↪  loaded_metrics['train_outs'],
        ↪  loaded_metrics['val_outs'],
        ↪  loaded_metrics['test_outs']



    def test_step(self, batch, batch_nb):
        loss, accuracy, acc2, acc3, preds =
        ↪  self.forward(batch)
        self.log("acc/test", accuracy)



        pred_probs = preds.cpu().detach().numpy()
        preds = np.argmax(pred_probs, axis=1)

        outs = {
            'preds':preds,
            'pred_probs':pred_probs,
            'labels':batch[1]
        }

        self.cur_test_preds.append(outs)


    def configure_optimizers(self):
        optimizer = torch.optim.SGD(
            self.model.parameters(),
            lr=self.hparams.learning_rate,
            weight_decay=self.hparams.weight_decay,
            momentum=0.9,
            nesterov=True,
        )
        total_steps = self.hparams.max_epochs *
        ↪  len(self.train_dataloader())
        scheduler = {
            "scheduler": WarmupCosineLR(
                optimizer,
                ↪  warmup_epochs=total_steps *
                ↪  0.3, max_epochs=total_steps
            ),
            "interval": "step",
            "name": "learning_rate",
        }
        return [optimizer], [scheduler]
```

```

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix,
↪  ConfusionMatrixDisplay
import pickle
from torchmetrics import Accuracy
import torch


def load_metrics():

    with open('metrics/saved_metrics/lbl_v1.pkl',
    ↪  'rb') as f:
        loaded_metrics = pickle.load(f)

    return  loaded_metrics['train_metrics'],
    ↪  loaded_metrics['val_metrics'],
    ↪  loaded_metrics['train_outs'],
    ↪  loaded_metrics['val_outs'],
    ↪  loaded_metrics['test_outs']



train_metrics, val_metrics, train_outs, val_outs,
↪  test_outs = load_metrics()




fig, axs = plt.subplots(1, 3)

# Loss per epoch

train_losses = []
val_losses = []
index = []
for i in range(len(train_metrics)):
```

```
    loss_train = train_metrics[i]['loss'].cpu().detach().numpy().item()
    loss_val = val_metrics[i]['loss'].cpu().detach().numpy().item()

    index.append(i)
    train_losses.append(loss_train)
    val_losses.append(loss_val)


axs[0].plot(index, train_losses)
axs[0].plot(index, val_losses)
axs[0].set(xlabel='Epoch', ylabel='Average Loss')
axs[0].set_title('Average Loss vs Epoch')
axs[0].legend(['Train', 'Validation'])


#################


# Train Accuracy per Epoch

train_accuracy1 = []
train_accuracy2 = []
train_accuracy3 = []
index = []
for i in range(len(train_metrics)):
    acc1 = train_metrics[i]['acc1'].cpu().detach().numpy().item()
    acc2 = train_metrics[i]['acc2'].cpu().detach().numpy().item()
    acc3 = train_metrics[i]['acc3'].cpu().detach().numpy().item()

    index.append(i)
    train_accuracy1.append(acc1)
    train_accuracy2.append(acc2)
    train_accuracy3.append(acc3)

axs[1].plot(index, train_accuracy1)
axs[1].plot(index, train_accuracy2)
axs[1].plot(index, train_accuracy3)
axs[1].set(xlabel='Epoch', ylabel='Accuracy')
axs[1].set_title('Train Accuracy vs Epoch')
axs[1].legend(['Top-1 Accuracy', 'Top-2 Accuracy', 'Top-3 Accuracy'])


#################


# Val Accuracy per Epoch

val_accuracy1 = []
val_accuracy2 = []
val_accuracy3 = []
index = []
for i in range(len(val_metrics)):
    acc1 = val_metrics[i]['acc1'].cpu().detach().numpy().item()
    acc2 = val_metrics[i]['acc2'].cpu().detach().numpy().item()
    acc3 = val_metrics[i]['acc3'].cpu().detach().numpy().item()

    index.append(i)
    val_accuracy1.append(acc1)
    val_accuracy2.append(acc2)
    val_accuracy3.append(acc3)

axs[2].plot(index, val_accuracy1)
axs[2].plot(index, val_accuracy2)
axs[2].plot(index, val_accuracy3)
axs[2].set(xlabel='Epoch', ylabel='Accuracy')
axs[2].set_title('Validation Accuracy vs Epoch')
axs[2].legend(['Top-1 Accuracy', 'Top-2 Accuracy', 'Top-3 Accuracy'])
fig.suptitle("Loss and Accuracy Metrics for the model")
plt.show()


#################

fig, axs = plt.subplots(1, 3)


# Train Confusion Matrix

cm = confusion_matrix(train_outs['labels'], train_outs['preds'])

disp = ConfusionMatrixDisplay(cm)
disp.plot(ax=axs[0])
disp.ax_.set_title('Train Confusion Matrix')


#################



# Val Confusion Matrix

cm = confusion_matrix(val_outs['labels'], val_outs['preds'])

disp = ConfusionMatrixDisplay(cm)
disp.plot(ax=axs[1])
disp.ax_.set_title('Validation Confusion Matrix')


#################



# Test Confusion Matrix

cm = confusion_matrix(test_outs['labels'], test_outs['preds'])

disp = ConfusionMatrixDisplay(cm)
disp.plot(ax=axs[2])
disp.ax_.set_title('Test Confusion Matrix')
fig.suptitle("Confusion Metrics for the model")
plt.show()

#################


# Train Accuracy

measureAcc1 = Accuracy(top_k=1)
measureAcc2 = Accuracy(top_k=2)
measureAcc3 = Accuracy(top_k=3)

train_acc1 = measureAcc1(torch.from_numpy(train_outs['pred_probs']), torch.from_numpy(train_outs['labels']).int())
train_acc2 = measureAcc2(torch.from_numpy(train_outs['pred_probs']), torch.from_numpy(train_outs['labels']).int())
train_acc3 = measureAcc3(torch.from_numpy(train_outs['pred_probs']), torch.from_numpy(train_outs['labels']).int())

print("[Train set] Top-1 Accuracy: {acc1:.4f}, Top-2 Accuracy: {acc2:.4f}, Top-3 Accuracy: {acc3:.4f},".format(acc1=train_acc1*100, acc2=train_acc2*100, acc3=train_acc3*100))

#################
```

```

# Val Accuracy


val_acc1 = measureAcc1(torch.from_numpy(val_outs['pred_probs']), torch.from_numpy(val_outs['labels']).int())
val_acc2 = measureAcc2(torch.from_numpy(val_outs['pred_probs']), torch.from_numpy(val_outs['labels']).int())
val_acc3 = measureAcc3(torch.from_numpy(val_outs['pred_probs']), torch.from_numpy(val_outs['labels']).int())

print("[Validation set] Top-1 Accuracy: {acc1:.4f}, Top-2 Accuracy: {acc2:.4f}, Top-3 Accuracy: {acc3:.4f},".format(acc1=val_acc1*100, acc2=val_acc2*100, acc3=val_acc3*100))

#################


# Test Accuracy


test_acc1 = measureAcc1(torch.from_numpy(test_outs['pred_probs']), torch.from_numpy(test_outs['labels']).int())
test_acc2 = measureAcc2(torch.from_numpy(test_outs['pred_probs']), torch.from_numpy(test_outs['labels']).int())
test_acc3 = measureAcc3(torch.from_numpy(test_outs['pred_probs']), torch.from_numpy(test_outs['labels']).int())

print("[Test set] Top-1 Accuracy: {acc1:.4f}, Top-2 Accuracy: {acc2:.4f}, Top-3 Accuracy: {acc3:.4f},".format(acc1=test_acc1*100, acc2=test_acc2*100, acc3=test_acc3*100))

#################
```

```
import math
import warnings
from typing import List

from torch.optim import Optimizer
from torch.optim.lr_scheduler import _LRScheduler


class WarmupCosineLR(_LRScheduler):
    def __init__(
        self,
        optimizer: Optimizer,
        warmup_epochs: int,
        max_epochs: int,
        warmup_start_lr: float = 1e-8,
        eta_min: float = 1e-8,
        last_epoch: int = -1,
    ) -> None:

        self.warmup_epochs = warmup_epochs
        self.max_epochs = max_epochs
        self.warmup_start_lr = warmup_start_lr
        self.eta_min = eta_min

        super(WarmupCosineLR, self).__init__(optimizer, last_epoch)

    def get_lr(self) -> List[float]:
        if not self._get_lr_called_within_step:
            warnings.warn(
                "To get the last learning rate computed by the scheduler, "
                "use `get_last_lr()`.",
                UserWarning,
            )

        if self.last_epoch == 0:
            return [self.warmup_start_lr] * len(self.base_lrs)
        elif self.last_epoch < self.warmup_epochs:
            return [
                group["lr"]
                + (base_lr - self.warmup_start_lr) / (self.warmup_epochs - 1)
                for base_lr, group in zip(self.base_lrs, self.optimizer.param_groups)
            ]
        elif self.last_epoch == self.warmup_epochs:
            return self.base_lrs
        elif (self.last_epoch - 1 - self.max_epochs) % (
            2 * (self.max_epochs - self.warmup_epochs)
        ) == 0:
            return [
                group["lr"]
                + (base_lr - self.eta_min)
                * (1 - math.cos(math.pi / (self.max_epochs - self.warmup_epochs)))
                / 2
                for base_lr, group in zip(self.base_lrs, self.optimizer.param_groups)
            ]

        return [
            (
                1
                + math.cos(
                    math.pi
                    * (self.last_epoch - self.warmup_epochs)
                    / (self.max_epochs - self.warmup_epochs)
                )
            )
            / (
                1
                + math.cos(
                    math.pi
                    * (self.last_epoch - self.warmup_epochs - 1)
                    / (self.max_epochs - self.warmup_epochs)
                )
            )
            * (group["lr"] - self.eta_min)
            + self.eta_min
            for group in self.optimizer.param_groups
        ]

    def _get_closed_form_lr(self) -> List[float]:
        if self.last_epoch < self.warmup_epochs:
            return [
                self.warmup_start_lr
                + self.last_epoch
                * (base_lr - self.warmup_start_lr)
                / (self.warmup_epochs - 1)
                for base_lr in self.base_lrs
            ]

        return [
            self.eta_min
            + 0.5
            * (base_lr - self.eta_min)
            * (
                1
                + math.cos(
```

```
                    math.pi
                    * (self.last_epoch -
                    ↪  self.warmup_epochs)
                    / (self.max_epochs -
                    ↪  self.warmup_epochs)
                )
            )
            for base_lr in self.base_lrs
        ]
```

```
import os
from collections import namedtuple

import torch
import torch.nn as nn
import torch.nn.functional as F

__all__ = ["InceptionModel", "InceptionModel"]


_InceptionModelOuputs = namedtuple(
    "InceptionModelOuputs", ["logits",
    ↪  "aux_logits2", "aux_logits1"]
)


def InceptionModel(pretrained=False, progress=True,
↪  device="cpu", aux_logits=False, **kwargs):

    model = InceptionModel(aux_logits=aux_logits)
    if pretrained:
        script_dir = os.path.dirname(__file__)
        state_dict = torch.load(
            script_dir +
            ↪  "/state_dicts/InceptionModel.pt",
            ↪  map_location=device
        )
        model.load_state_dict(state_dict)
    return model



class InceptionModel(nn.Module):

    def __init__(self, num_classes=10,
    ↪  aux_logits=False):
        super(InceptionModel, self).__init__()
        self.aux_logits = aux_logits


        self.conv1 = BasicConv2d(3, 192,
        ↪  kernel_size=3, stride=1, padding=1)

        self.inception1 = Inception(192, 64, 96,
        ↪  128, 16, 32, 32)
        self.inception2 = Inception(256, 128, 128,
        ↪  192, 32, 96, 64)

        self.maxpool1 = nn.MaxPool2d(3, stride=2,
        ↪  padding=1, ceil_mode=False)

        self.inception3 = Inception(480, 192, 96,
        ↪  208, 16, 48, 64)
        self.inception4 = Inception(512, 160, 112,
        ↪  224, 24, 64, 64)
        self.inception5 = Inception(512, 128, 128,
        ↪  256, 24, 64, 64)
        self.inception6 = Inception(512, 112, 144,
        ↪  288, 32, 64, 64)
        self.inception7 = Inception(528, 256, 160,
        ↪  320, 32, 128, 128)

        self.maxpool2 = nn.MaxPool2d(3, stride=2,
        ↪  padding=1, ceil_mode=False)

        self.inception8 = Inception(832, 256, 160,
        ↪  320, 32, 128, 128)
        self.inception9 = Inception(832, 384, 192,
        ↪  384, 48, 128, 128)

        if aux_logits:
            self.aux1 = InceptionAux(512,
            ↪  num_classes)
            self.aux2 = InceptionAux(528,
            ↪  num_classes)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(1024, num_classes)


    def forward(self, x):

        x = self.conv1(x)

        x = self.inception1(x)
        x = self.inception2(x)

        x = self.maxpool1(x)

        x = self.inception3(x)

        #First auxiliary branch.
        if self.training and self.aux_logits:
            aux1 = self.aux1(x)

        x = self.inception4(x)
        x = self.inception5(x)
        x = self.inception6(x)

        #Second auxiliary branch.
        if self.training and self.aux_logits:
            aux2 = self.aux2(x)

        x = self.inception7(x)

        x = self.maxpool2(x)

        x = self.inception8(x)
        x = self.inception9(x)

        x = self.avgpool(x)

        x = x.view(x.size(0), -1)

        x = self.dropout(x)
        x = self.fc(x)

        if self.training and self.aux_logits:
            return _InceptionModelOuputs(x, aux2,
            ↪  aux1)
        return x


#This is the implementation of the Inception
↪  Module. Improved implementation is used rather
↪  than the naive one.
#Every convolution has a 1x1 convolution beforehand
↪  to reduce channels, except the pooling branch.
#Pooling is performed first, then the 1x1
↪  convolution.
#The outputs are concetenated at the end as
↪  different channels.
class Inception(nn.Module):
    def __init__(self, in_channels, ch1x1,
    ↪  ch3x3red, ch3x3, ch5x5red, ch5x5,
    ↪  pool_proj):
        super(Inception, self).__init__()

        self.branch1 = BasicConv2d(in_channels,
        ↪  ch1x1, kernel_size=1)
```

```
        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, ch3x3red,
            ↪  kernel_size=1),
            BasicConv2d(ch3x3red, ch3x3,
            ↪  kernel_size=3, padding=1),
        )

        self.branch3 = nn.Sequential(
            BasicConv2d(in_channels, ch5x5red,
            ↪  kernel_size=1),
            BasicConv2d(ch5x5red, ch5x5,
            ↪  kernel_size=3, padding=1),
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1,
            ↪  padding=1, ceil_mode=True),
            BasicConv2d(in_channels, pool_proj,
            ↪  kernel_size=1),
        )

    def forward(self, x):
        branch1 = self.branch1(x)
        branch2 = self.branch2(x)
        branch3 = self.branch3(x)
        branch4 = self.branch4(x)

        outputs = [branch1, branch2, branch3,
        ↪  branch4]
        return torch.cat(outputs, 1)


class BasicConv2d(nn.Module):
    def __init__(self, in_channels, out_channels,
    ↪  **kwargs):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels,
        ↪  out_channels, bias=False, **kwargs)
        self.bn = nn.BatchNorm2d(out_channels,
        ↪  eps=0.001)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)


#This is the ending sequence of the Auxiliary
↪  Branches.
#The outputs are treated as normal model outputs to
↪  calculate loss in module.py, expect only in
↪  training.
class InceptionAux(nn.Module):
    def __init__(self, in_channels, num_classes):
        super(InceptionAux, self).__init__()
        self.conv = BasicConv2d(in_channels, 128,
        ↪  kernel_size=1)

        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, num_classes)

    def forward(self, x):
        x = F.adaptive_avg_pool2d(x, (4, 4))

        x = self.conv(x)

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x), inplace=True)

        x = F.dropout(x, 0.7,
        ↪  training=self.training)

        x = self.fc2(x)

        return x
```

```
import os
import zipfile

import pytorch_lightning as pl
import requests
from torch.utils.data import DataLoader, Subset
from torchvision import transforms as T
from torchvision.datasets import CIFAR10
from tqdm import tqdm


class CIFAR10Data(pl.LightningDataModule):
    def __init__(self, args):
        super().__init__()
        self.hparams = args
        self.mean = (0.4914, 0.4822, 0.4465)
        self.std = (0.2471, 0.2435, 0.2616)


    def train_dataloader(self):
        transform = T.Compose(
            [
                T.RandomCrop(32, padding=4),
                T.RandomHorizontalFlip(),
                T.ToTensor(),
                T.Normalize(self.mean, self.std),
            ]
        )
        dataset =
        ↪  CIFAR10(root=self.hparams.data_dir,
        ↪  train=True, transform=transform)
        #subset = Subset(dataset, range(256))

        dataloader = DataLoader(
            dataset,
            batch_size=self.hparams.batch_size,
            num_workers=self.hparams.num_workers,
            shuffle=True,
            drop_last=True,
            pin_memory=True,
        )
        return dataloader

    def val_dataloader(self):
        transform = T.Compose(
            [
                T.ToTensor(),
                T.Normalize(self.mean, self.std),
            ]
        )
        dataset =
        ↪  CIFAR10(root=self.hparams.data_dir,
        ↪  train=False, transform=transform)
        #subset = Subset(dataset, range(256))

        dataloader = DataLoader(
            dataset,
            batch_size=self.hparams.batch_size,
            num_workers=self.hparams.num_workers,
            drop_last=True,
            pin_memory=True,
        )
        return dataloader

    def test_dataloader(self):
        return self.val_dataloader()
```

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

```
from torch.nn.modules.loss import _WeightedLoss


class LabelSmoothingLoss(nn.Module):
    def __init__(self, classes, smoothing=0.1):
        super(LabelSmoothingLoss, self).__init__()
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.cls = classes

    def forward(self, pred, target):
        assert 0 <= self.smoothing < 1
        pred = pred.log_softmax(-1)

        with torch.no_grad():
            dist = torch.zeros_like(pred)
            dist.fill_(self.smoothing / (self.cls -
            ↪  1))
            dist.scatter_(1,
            ↪  target.data.unsqueeze(1),
            ↪  self.confidence)
        return torch.mean(torch.sum(-dist * pred,
        ↪  dim=self.dim))
```