DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN3150 Assignment 03

Simple convolutional neural network to perform classification

ABEYGUNATHILAKA T.L.	**200003P**
MANIMOHAN T.	**200377M**
PUSHPAKUMARA H.M.R.M.	**200488E**
TILAKARATHNA U.A.	**200664P**

December 02, 2023

Table of Contents

Introduction

Why CNNs preferable for image classification over multilayered perceptrons (MLPs) or simple feedforward neural networks (NNs)?

- Local receptive fields:
  CNNs can exploit the local receptive field property of images, which means that they can learn to identify patterns in small patches of an image without having to look at the entire image at once. This is a significant advantage over MLPs, which require the entire image to be flattened into a vector before they can be processed.

- Parameter sharing:
  CNNs use parameter sharing, which means that the same weights are used to convolve over different parts of the image. This helps to reduce the number of parameters that the network needs to learn, and it also makes the network more robust to noise and variations in the image.

- Pooling:
  CNNs use pooling layers to reduce the dimensionality of the output of the convolutional layers. This helps to prevent overfitting and makes the network more computationally efficient.

- Translation invariance:
  CNNs are translation invariant, which means that they can recognize objects regardless of their position in the image. This is a valuable property for image classification tasks, as it means that the network does not need to learn a separate model for each possible position of an object.

- Hierarchical feature extraction:
  CNNs can extract features from images at different scales, which allows them to learn more complex features than MLPs. This is important for image classification tasks, as it allows the network to learn to identify the key features that distinguish between different classes of objects.

CNN Model

The architecture consists of three convolutional layers followed by max-pooling layers, and finally, two fully connected layers. The model also includes batch normalization, ReLU activation functions, and dropout for regularization.

Convolutional Layers:

1. **Convolutional Layer 1:**
    - Input: 3 channels (for RGB images)
    - Output Channels: 32
    - Kernel Size: 5x5
    - Stride: 1
    - Padding: 2
    - Batch Normalization and ReLU Activation applied.
2. **Max Pooling Layer 1:**
    - Kernel Size: 2x2
    - Stride: 2
3. **Convolutional Layer 2:**
    - Input Channels: 32
    - Output Channels: 64
    - Kernel Size: 5x5
    - Stride: 1
    - Padding: 2
    - Batch Normalization and ReLU Activation applied.
4. **Max Pooling Layer 2:**
    - Kernel Size: 2x2
    - Stride: 2

5. **Convolutional Layer 3:**
    - Input Channels: 64
    - Output Channels: 128
    - Kernel Size: 5x5
    - Stride: 1
    - Padding: 2
    - Batch Normalization and ReLU Activation applied.
6. **Max Pooling Layer 3:**
    - Kernel Size: 2x2
    - Stride: 2

Fully Connected Layers:

7. **Flatten Layer:**
    - Flattens the output from the last convolutional layer for input to fully connected layers.
8. **Fully Connected Layer 1:**
    - Input Features: 2048 [128 * (32/8) * (32/8)]
    - Output Features: 64
    - ReLU Activation applied.
    - Dropout with probability 0.5

Output Layer:

9. **Fully Connected Layer 2 (Output Layer):**
    - Input Features: 64
    - Output Features: 10 (assuming 10 classes for classification)
    - SoftMax Activation applied for multiclass classification.

Model Parameters:

The model includes several parameters, and the total number of learnable parameters can be calculated for each layer.

Convolutional Layer 1:

- Parameters: (Input Channels * Output Channels * Kernel Size * Kernel Size) + Output Channels + Output Channels (for batch normalization)
- Total Parameters: (3 * 32 * 5 * 5) + 32 + 32 = 2464

Convolutional Layer 2:

- Parameters: (32 * 64 * 5 * 5) + 64 + 64 (for batch normalization)
- Total Parameters: (32 * 64 * 5 * 5) + 64 + 64 = 51328

Convolutional Layer 3:

- Parameters: (64 * 128 * 5 * 5) + 128 + 128 (for batch normalization)
- Total Parameters: (64 * 128 * 5 * 5) + 128 + 128 = 205056

Fully Connected Layer 1:

- Parameters: (128 * (32/8) * (32/8) * 64) + 64
- Total Parameters: (128 * 4 * 4 * 64) + 64 = 131136

Output Layer:

- Parameters: (64 * 10) + 10
- Total Parameters: (64 * 10) + 10 = 650

1. Convolutional Layer 1: 2464 parameters
2. Convolutional Layer 2: 51328 parameters
3. Convolutional Layer 3: 205056 parameters
4. Fully Connected Layer 1: 131136 parameters
5. Output Layer: 650 parameters

The total number of learnable parameters in the entire CNN model is the sum of parameters from all layers:

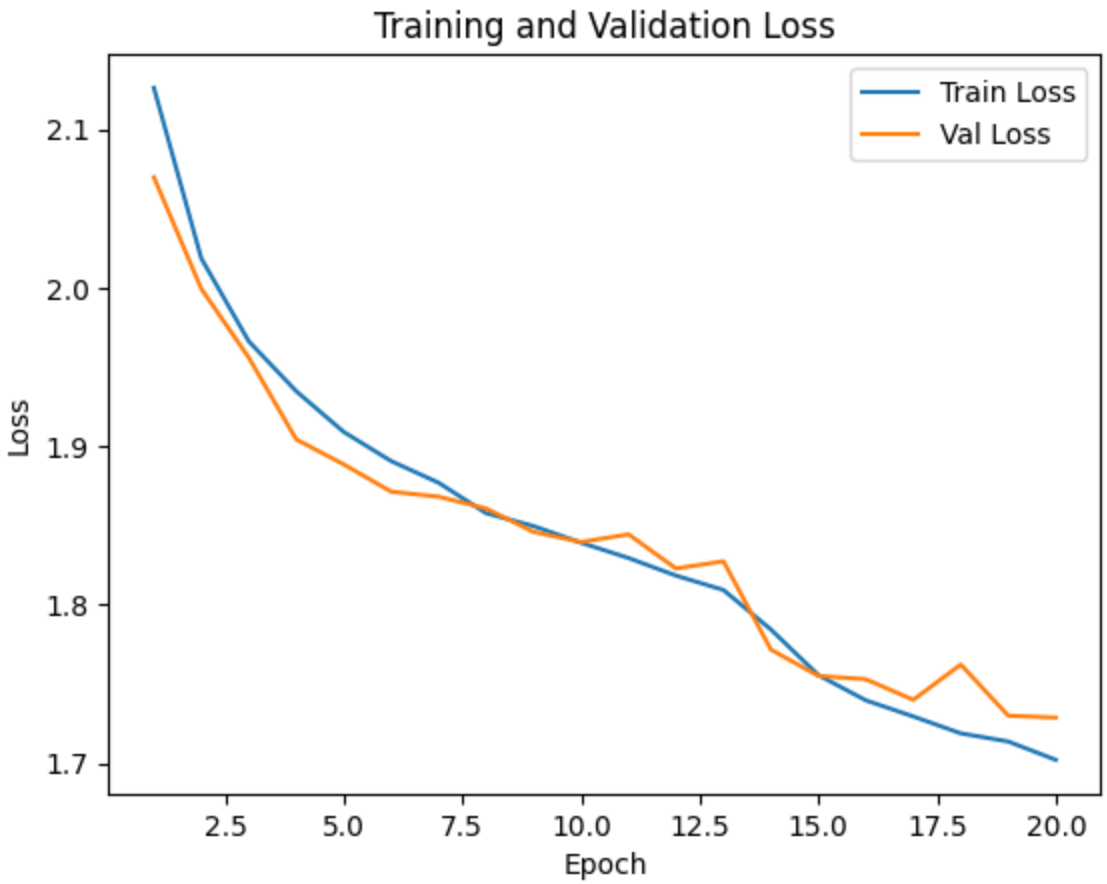
2464+51328+205056+131136+650 = 390634

So, the CNN model has a total of **390,630** learnable parameters.

Model Training

Learning rate = 0.00025

```
Epoch [1/20], Train Loss: 2.1260, Val Loss: 2.0696
Epoch [2/20], Train Loss: 2.0183, Val Loss: 1.9992
Epoch [3/20], Train Loss: 1.9661, Val Loss: 1.9558
Epoch [4/20], Train Loss: 1.9346, Val Loss: 1.9042
Epoch [5/20], Train Loss: 1.9090, Val Loss: 1.8885
Epoch [6/20], Train Loss: 1.8906, Val Loss: 1.8713
Epoch [7/20], Train Loss: 1.8769, Val Loss: 1.8681
Epoch [8/20], Train Loss: 1.8577, Val Loss: 1.8607
Epoch [9/20], Train Loss: 1.8496, Val Loss: 1.8459
Epoch [10/20], Train Loss: 1.8391, Val Loss: 1.8395
Epoch [11/20], Train Loss: 1.8295, Val Loss: 1.8444
Epoch [12/20], Train Loss: 1.8184, Val Loss: 1.8229
Epoch [13/20], Train Loss: 1.8093, Val Loss: 1.8274
Epoch [14/20], Train Loss: 1.7844, Val Loss: 1.7717
Epoch [15/20], Train Loss: 1.7556, Val Loss: 1.7551
Epoch [16/20], Train Loss: 1.7397, Val Loss: 1.7530
Epoch [17/20], Train Loss: 1.7295, Val Loss: 1.7400
Epoch [18/20], Train Loss: 1.7189, Val Loss: 1.7623
Epoch [19/20], Train Loss: 1.7137, Val Loss: 1.7300
Epoch [20/20], Train Loss: 1.7022, Val Loss: 1.7288
```



Why have we chosen Adam Optimizer over SGD?

Adam and SGD are both optimization algorithms used in training deep learning models. Adam is an adaptive learning rate algorithm, which means it adjusts the learning rate for each parameter during training. This can be beneficial for models with sparse gradients or noisy data. SGD uses a fixed learning rate for all parameters, which can be more difficult to tune but can be more robust to noisy gradients. Adam generally converges faster than SGD, but SGD may perform better in the long run. Adam is also more sensitive to hyperparameters than SGD.

Why have we chosen sparse categorical crossentropy as the loss function?

- Single-Label Classification:
  CIFAR-10 is a single-label classification problem, meaning that each image belongs to one and only one of ten possible classes (e.g., airplane, car, bird, etc.). Sparse categorical crossentropy is a loss function specifically designed for this type of problem, where class labels are represented as integers.

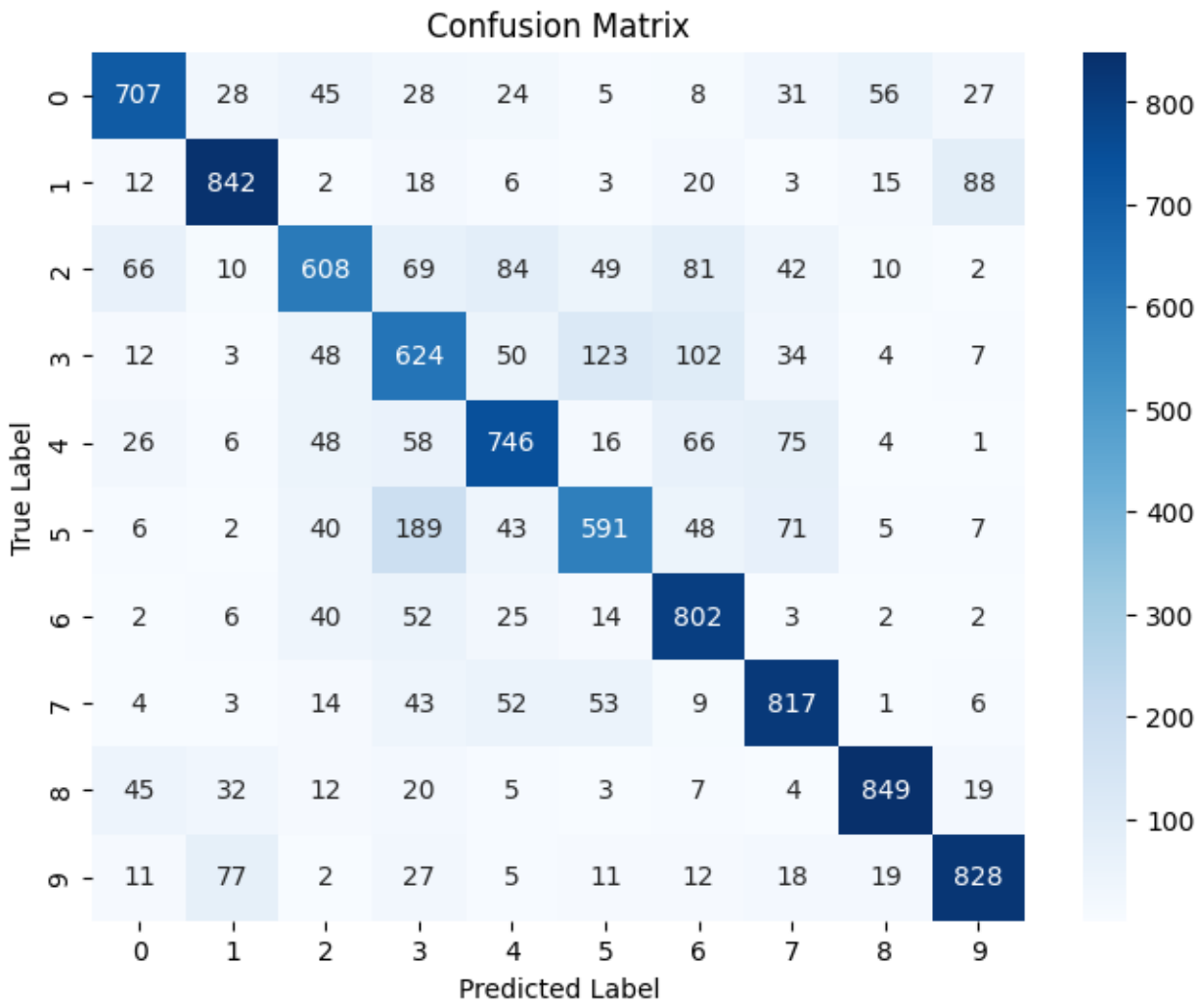
- Integer Labels:
  CIFAR-10 uses integer labels (0 to 9) to represent the ten classes. Sparse categorical crossentropy directly handles these integer labels, eliminating the need for conversion to another format.

- No One-Hot Encoding Required:
  Unlike categorical crossentropy, which requires labels to be one-hot encoded vectors, sparse categorical crossentropy allows you to provide class indices directly. This simplifies label representation and reduces computational overhead.

- Computational Efficiency:
  Sparse categorical crossentropy is more computationally efficient than converting labels to one-hot encoded vectors. It avoids the need for extra memory and computation associated with one-hot encoding.

Model evaluation

Test Accuracy:	**74.14%**
Weighted Precision:	**0.7443**
Weighted Recall:	**0.7414**

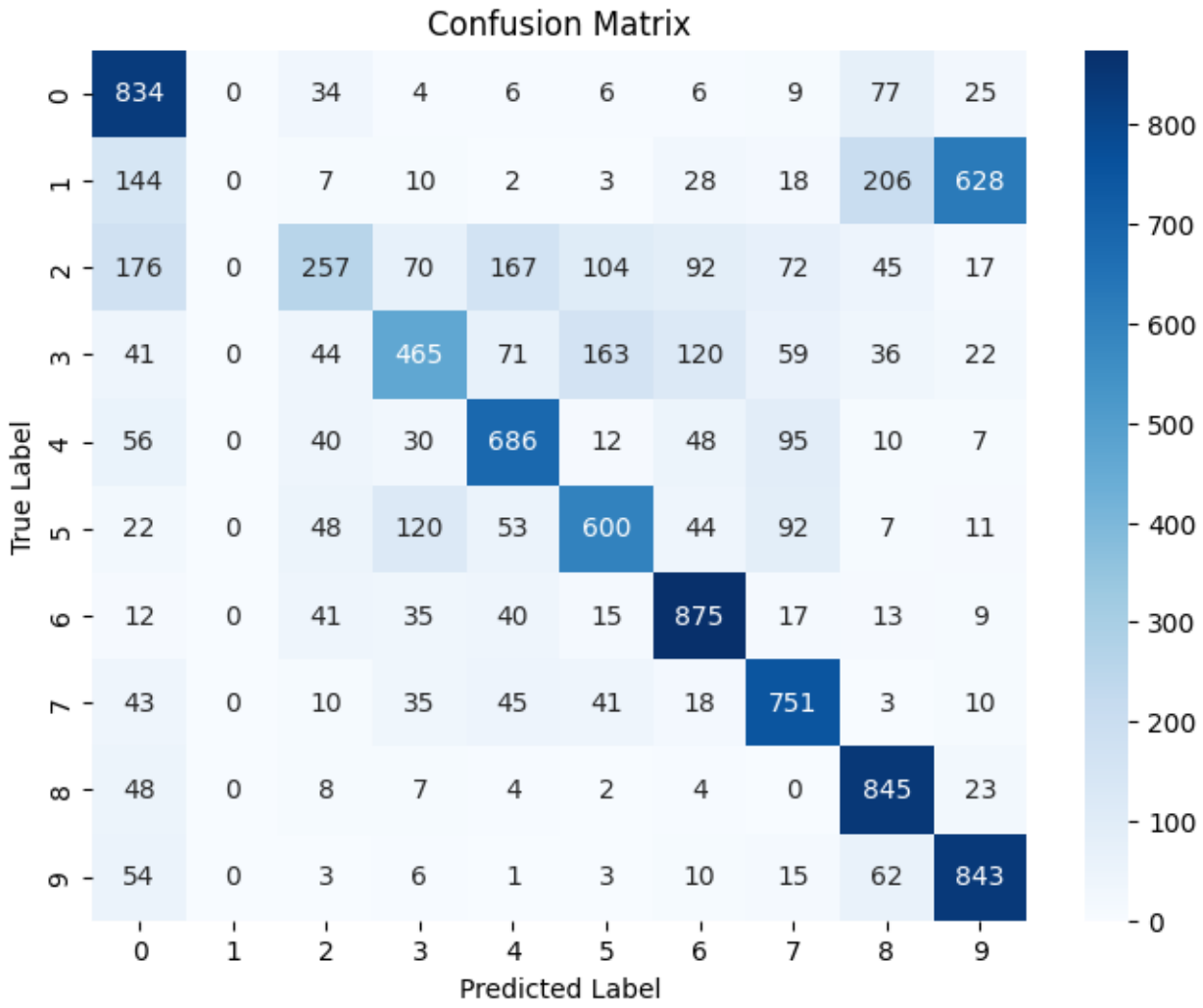


For different learning rates

Learning rate = 0.0001

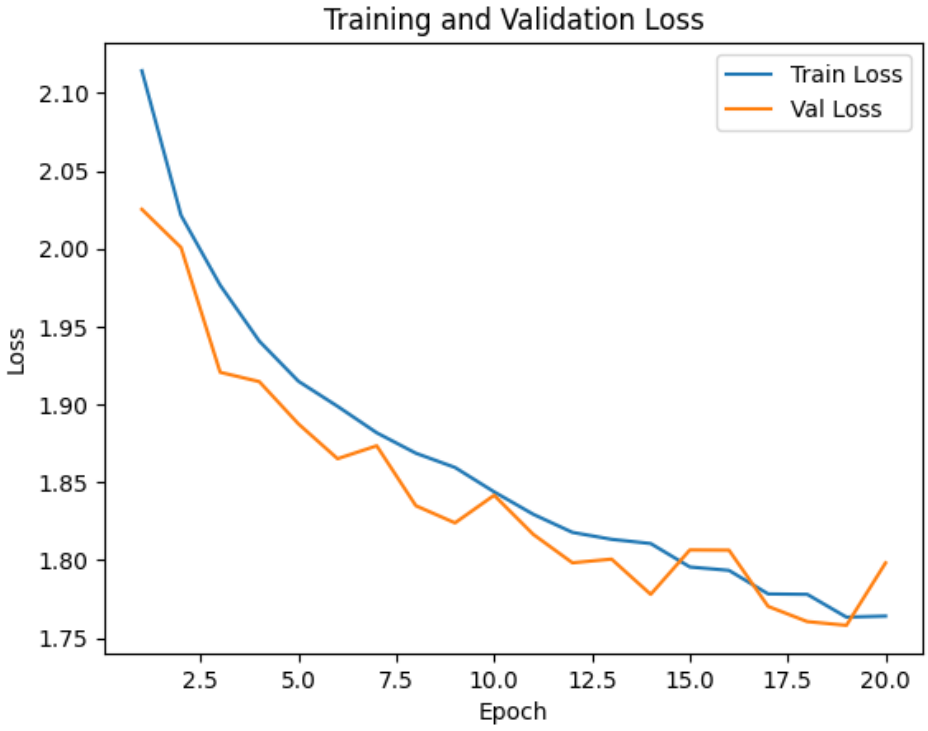
```
Epoch [1/20], Train Loss: 2.1414, Val Loss: 2.0453
Epoch [2/20], Train Loss: 2.0397, Val Loss: 1.9957
Epoch [3/20], Train Loss: 1.9904, Val Loss: 1.9483
Epoch [4/20], Train Loss: 1.9612, Val Loss: 1.9320
Epoch [5/20], Train Loss: 1.9418, Val Loss: 1.9386
Epoch [6/20], Train Loss: 1.9230, Val Loss: 1.9152
Epoch [7/20], Train Loss: 1.9100, Val Loss: 1.8972
Epoch [8/20], Train Loss: 1.8992, Val Loss: 1.8978
Epoch [9/20], Train Loss: 1.8876, Val Loss: 1.8827
Epoch [10/20], Train Loss: 1.8795, Val Loss: 1.8833
Epoch [11/20], Train Loss: 1.8703, Val Loss: 1.8726
Epoch [12/20], Train Loss: 1.8608, Val Loss: 1.8703
Epoch [13/20], Train Loss: 1.8543, Val Loss: 1.8604
Epoch [14/20], Train Loss: 1.8466, Val Loss: 1.8581
Epoch [15/20], Train Loss: 1.8403, Val Loss: 1.8671
Epoch [16/20], Train Loss: 1.8335, Val Loss: 1.8585
Epoch [17/20], Train Loss: 1.8270, Val Loss: 1.8608
Epoch [18/20], Train Loss: 1.8227, Val Loss: 1.8647
Epoch [19/20], Train Loss: 1.8148, Val Loss: 1.8725
Epoch [20/20], Train Loss: 1.8056, Val Loss: 1.8354
```

Test Accuracy:	61.56%
Weighted Precision:	0.5488
Weighted Recall:	0.6156

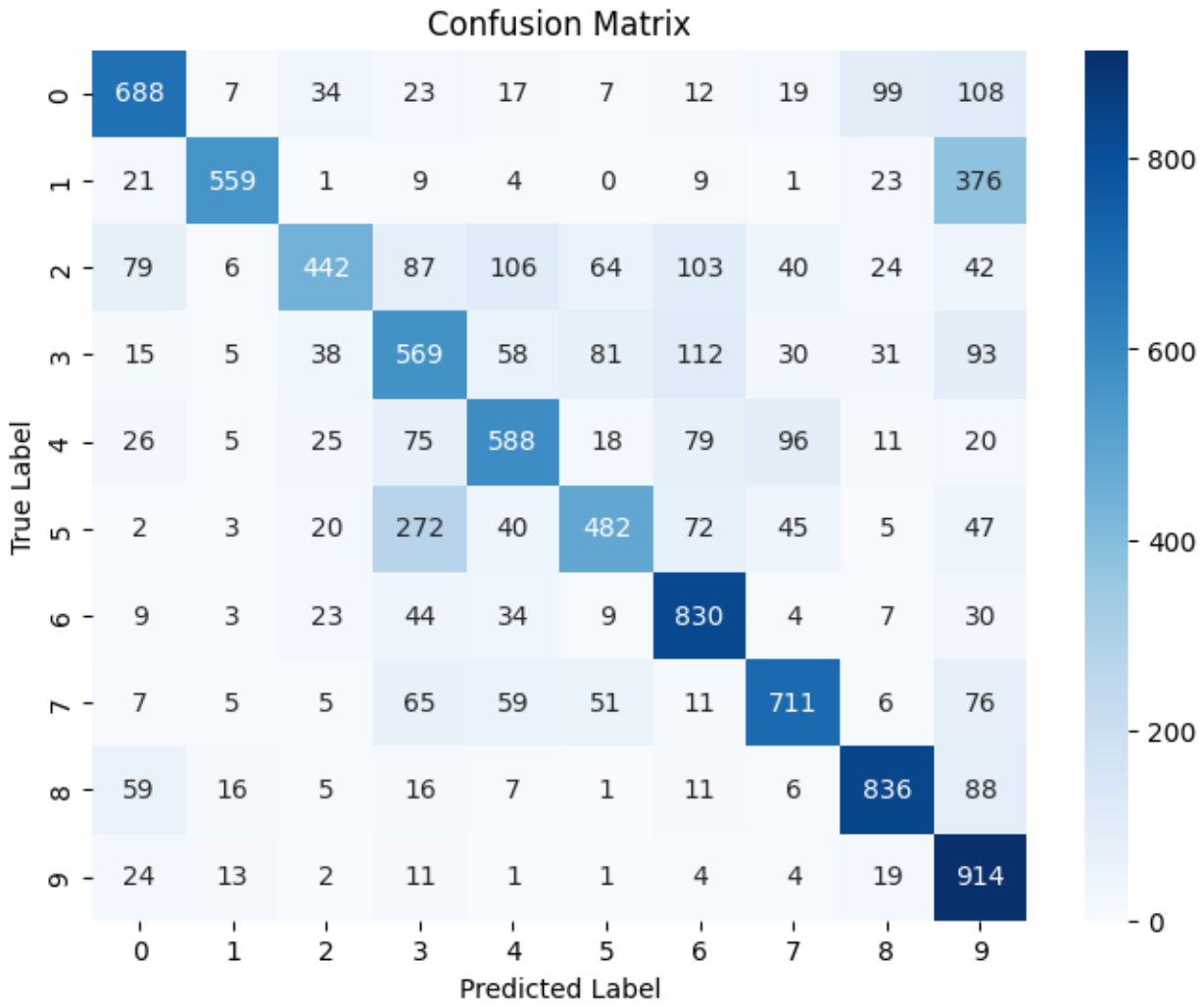


Learning rate = 0.001

```
Epoch [1/20], Train Loss: 2.1142, Val Loss: 2.0252
Epoch [2/20], Train Loss: 2.0214, Val Loss: 2.0007
Epoch [3/20], Train Loss: 1.9765, Val Loss: 1.9206
Epoch [4/20], Train Loss: 1.9406, Val Loss: 1.9146
Epoch [5/20], Train Loss: 1.9148, Val Loss: 1.8874
Epoch [6/20], Train Loss: 1.8987, Val Loss: 1.8651
Epoch [7/20], Train Loss: 1.8817, Val Loss: 1.8734
Epoch [8/20], Train Loss: 1.8686, Val Loss: 1.8349
Epoch [9/20], Train Loss: 1.8594, Val Loss: 1.8238
Epoch [10/20], Train Loss: 1.8438, Val Loss: 1.8416
Epoch [11/20], Train Loss: 1.8294, Val Loss: 1.8165
Epoch [12/20], Train Loss: 1.8178, Val Loss: 1.7982
Epoch [13/20], Train Loss: 1.8133, Val Loss: 1.8006
Epoch [14/20], Train Loss: 1.8106, Val Loss: 1.7779
Epoch [15/20], Train Loss: 1.7955, Val Loss: 1.8065
Epoch [16/20], Train Loss: 1.7933, Val Loss: 1.8063
Epoch [17/20], Train Loss: 1.7782, Val Loss: 1.7702
Epoch [18/20], Train Loss: 1.7779, Val Loss: 1.7605
Epoch [19/20], Train Loss: 1.7633, Val Loss: 1.7581
Epoch [20/20], Train Loss: 1.7640, Val Loss: 1.7982
```



Test Accuracy:	66.19%
Weighted Precision:	0.6897
Weighted Recall:	0.6619



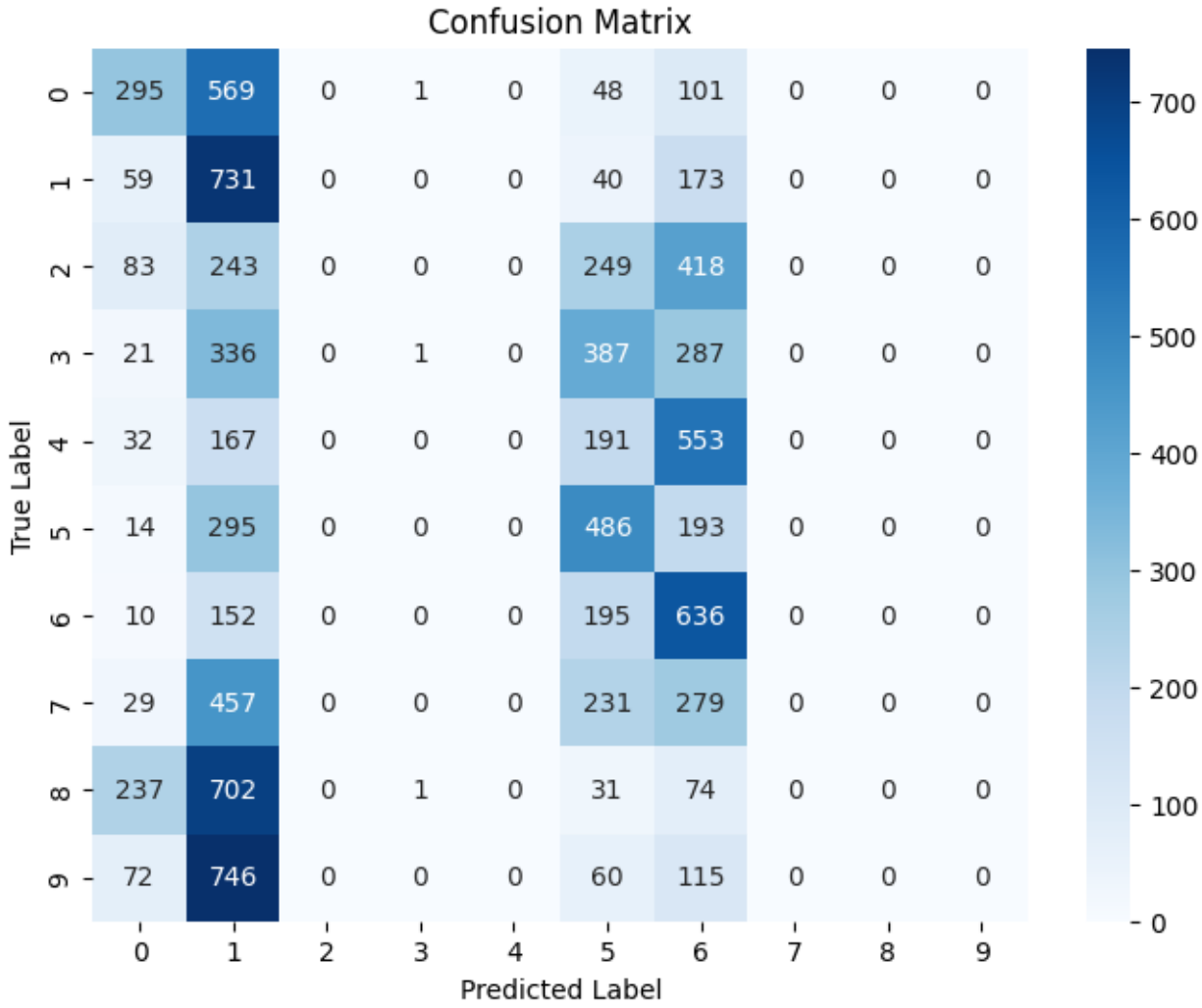
Learning rate = 0.01

```
Epoch [1/20], Train Loss: 2.2400, Val Loss: 2.2680
Epoch [2/20], Train Loss: 2.2799, Val Loss: 2.3015
Epoch [3/20], Train Loss: 2.2736, Val Loss: 2.2646
Epoch [4/20], Train Loss: 2.2555, Val Loss: 2.2531
Epoch [5/20], Train Loss: 2.2567, Val Loss: 2.2228
Epoch [6/20], Train Loss: 2.2398, Val Loss: 2.2294
Epoch [7/20], Train Loss: 2.2505, Val Loss: 2.2904
Epoch [8/20], Train Loss: 2.2670, Val Loss: 2.2874
Epoch [9/20], Train Loss: 2.2737, Val Loss: 2.2598
Epoch [10/20], Train Loss: 2.2646, Val Loss: 2.2767
Epoch [11/20], Train Loss: 2.2719, Val Loss: 2.2606
Epoch [12/20], Train Loss: 2.2991, Val Loss: 2.3621
Epoch [13/20], Train Loss: 2.3253, Val Loss: 2.2942
Epoch [14/20], Train Loss: 2.2723, Val Loss: 2.2484
Epoch [15/20], Train Loss: 2.2526, Val Loss: 2.2373
Epoch [16/20], Train Loss: 2.2562, Val Loss: 2.2816
Epoch [17/20], Train Loss: 2.2857, Val Loss: 2.2888
Epoch [18/20], Train Loss: 2.2712, Val Loss: 2.2490
Epoch [19/20], Train Loss: 2.2638, Val Loss: 2.2667
Epoch [20/20], Train Loss: 2.2492, Val Loss: 2.2446
```

Test Accuracy: 21.49%

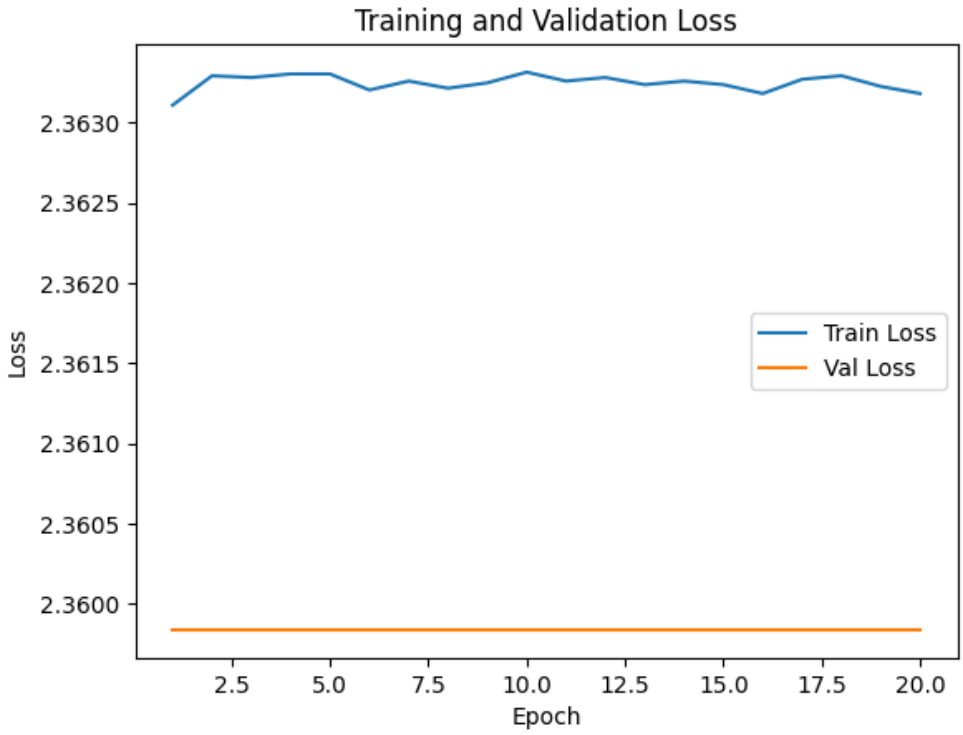
Weighted Precision: 0.1335

Weighted Recall: 0.2149



Learning rate = 0.1

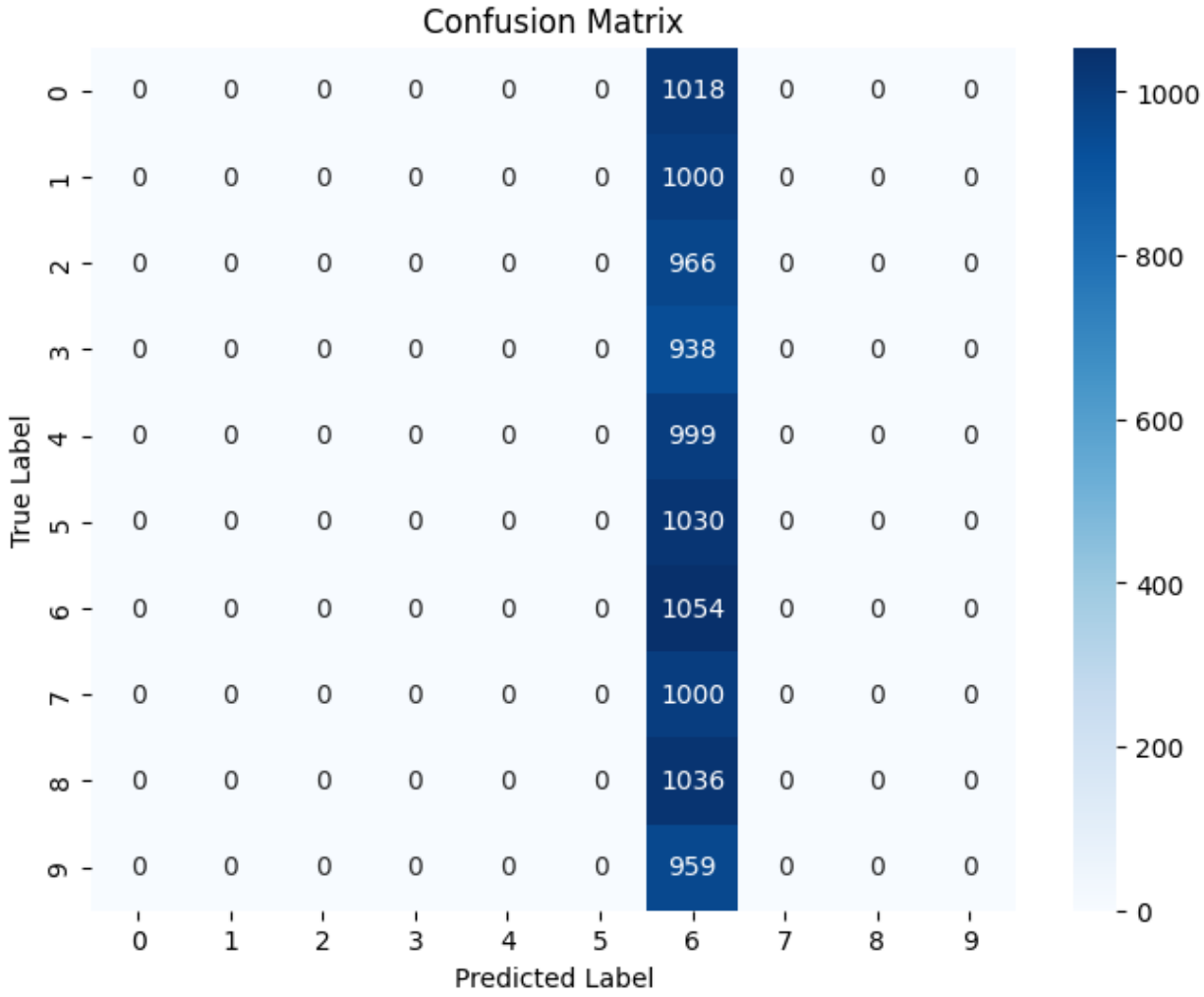
```
Epoch [1/20], Train Loss: 2.3631, Val Loss: 2.3598
Epoch [2/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [3/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [4/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [5/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [6/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [7/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [8/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [9/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [10/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [11/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [12/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [13/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [14/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [15/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [16/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [17/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [18/20], Train Loss: 2.3633, Val Loss: 2.3598
Epoch [19/20], Train Loss: 2.3632, Val Loss: 2.3598
Epoch [20/20], Train Loss: 2.3632, Val Loss: 2.3598
```



Test Accuracy: 10.54%

Weighted Precision: 0.0111

Weighted Recall: 0.1054



Pre-trained Models

Choosed state-of-the-art pre-trained models

1. **ResNet**
2. **Densenet**
3. **Googlenet**

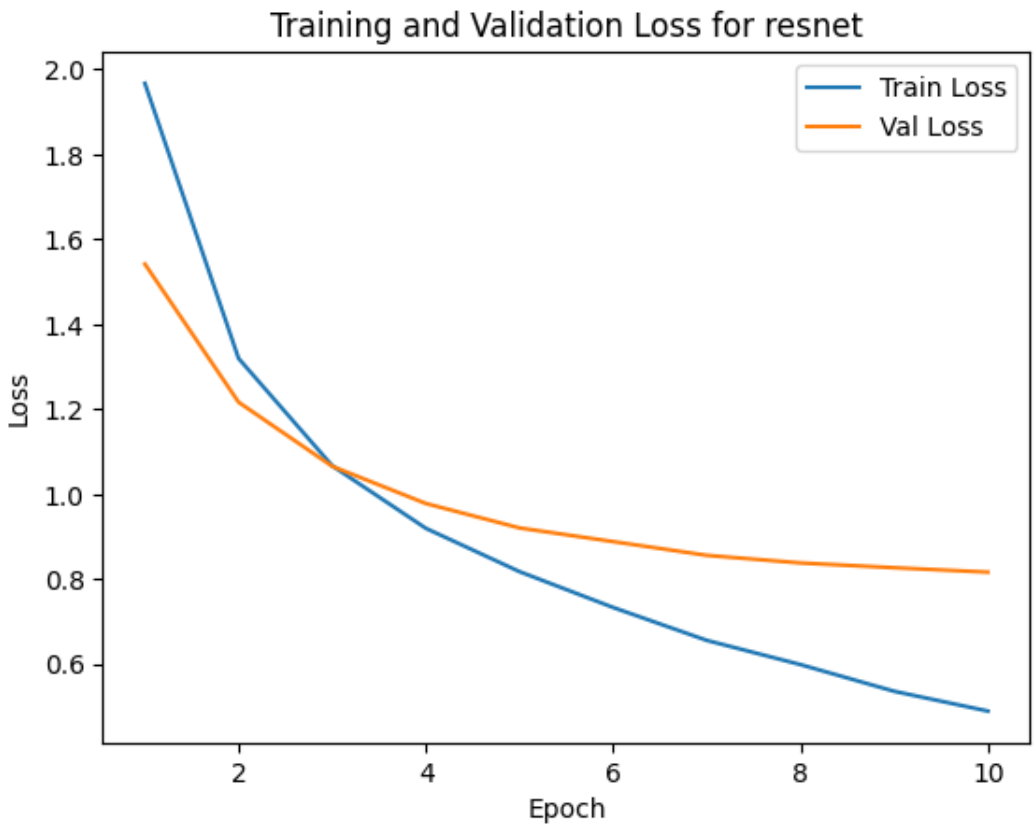
We used only 10 epochs when fine-tuning these models on the CIFAR-10 dataset because using more epochs led to overfitting. Overfitting is when a model learns the training data too well and is unable to generalize to new data. By limiting the training to 10 epochs, we were able to prevent overfitting and improve the models' performance on the test dataset.

Resnet

training and validation loss values for each epoch

```
Fine-tuning resnet for CIFAR-10:
Epoch 1/10, Train Loss: 1.9670, Validation Loss: 1.5417
Epoch 2/10, Train Loss: 1.3195, Validation Loss: 1.2165
Epoch 3/10, Train Loss: 1.0668, Validation Loss: 1.0656
Epoch 4/10, Train Loss: 0.9195, Validation Loss: 0.9780
Epoch 5/10, Train Loss: 0.8176, Validation Loss: 0.9205
Epoch 6/10, Train Loss: 0.7332, Validation Loss: 0.8883
Epoch 7/10, Train Loss: 0.6557, Validation Loss: 0.8559
Epoch 8/10, Train Loss: 0.5988, Validation Loss: 0.8381
Epoch 9/10, Train Loss: 0.5359, Validation Loss: 0.8269
Epoch 10/10, Train Loss: 0.4894, Validation Loss: 0.8164

Final resnet Training  Loss: 0.4894
Final resnet Validation Loss: 0.8164
```

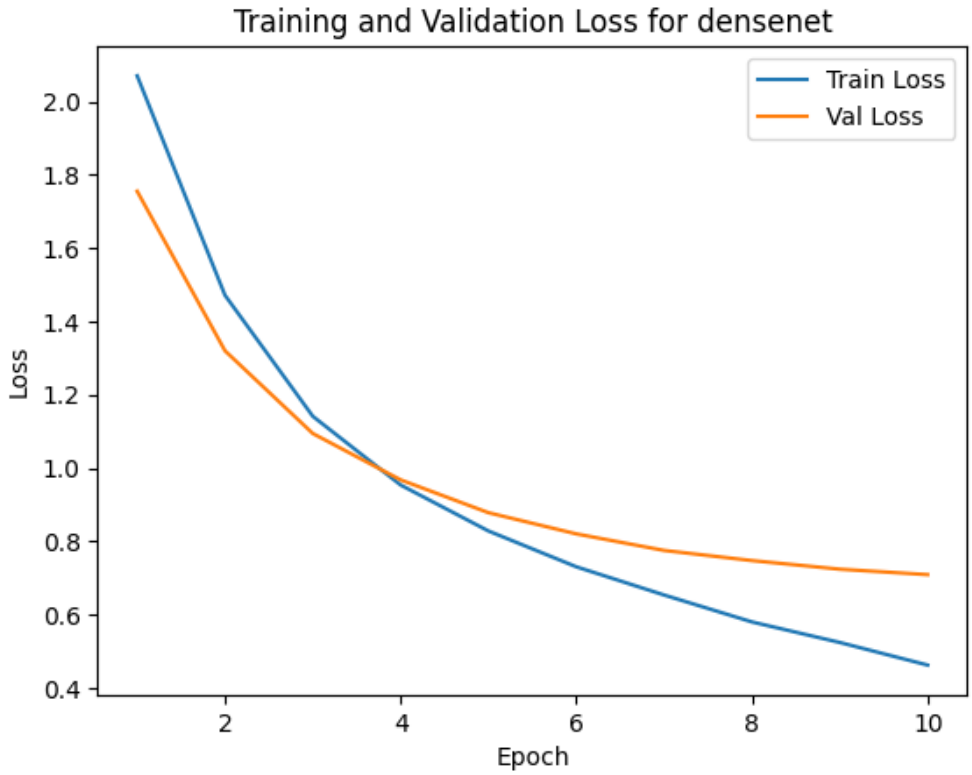


Densenet

training and validation loss values for each epoch

```
Fine-tuning densenet for CIFAR-10:
Epoch 1/10, Train Loss: 2.0702, Validation Loss: 1.7553
Epoch 2/10, Train Loss: 1.4715, Validation Loss: 1.3200
Epoch 3/10, Train Loss: 1.1412, Validation Loss: 1.0943
Epoch 4/10, Train Loss: 0.9537, Validation Loss: 0.9678
Epoch 5/10, Train Loss: 0.8283, Validation Loss: 0.8782
Epoch 6/10, Train Loss: 0.7306, Validation Loss: 0.8204
Epoch 7/10, Train Loss: 0.6536, Validation Loss: 0.7752
Epoch 8/10, Train Loss: 0.5801, Validation Loss: 0.7476
Epoch 9/10, Train Loss: 0.5241, Validation Loss: 0.7241
Epoch 10/10, Train Loss: 0.4624, Validation Loss: 0.7092

Final densenet Training Loss: 0.4624
Final densenet Validation Loss: 0.7092
```

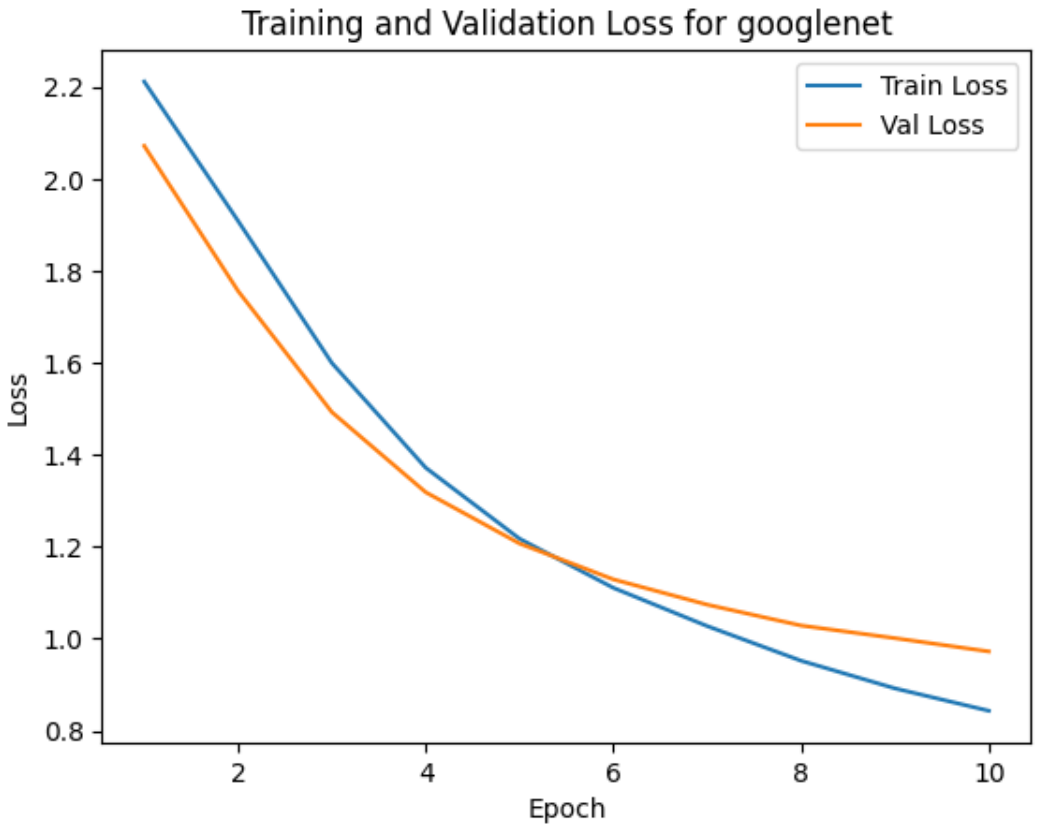


Googlenet

training and validation loss values for each epoch

```
Fine-tuning googlenet for CIFAR-10:
c:\Python311\cv\Lib\site-packages\torchvision\models\_utils.py:223:
  warnings.warn(msg)
Epoch 1/10, Train Loss: 2.2113, Validation Loss: 2.0716
Epoch 2/10, Train Loss: 1.9084, Validation Loss: 1.7561
Epoch 3/10, Train Loss: 1.5993, Validation Loss: 1.4925
Epoch 4/10, Train Loss: 1.3717, Validation Loss: 1.3184
Epoch 5/10, Train Loss: 1.2173, Validation Loss: 1.2069
Epoch 6/10, Train Loss: 1.1105, Validation Loss: 1.1289
Epoch 7/10, Train Loss: 1.0273, Validation Loss: 1.0739
Epoch 8/10, Train Loss: 0.9520, Validation Loss: 1.0286
Epoch 9/10, Train Loss: 0.8919, Validation Loss: 1.0011
Epoch 10/10, Train Loss: 0.8432, Validation Loss: 0.9722

Final googlenet Training Loss: 0.8432
Final googlenet Validation Loss: 0.9722
```



Model Evaluation

ResNet test accuracy:	73.05%
Densenet test accuracy:	76.18%
GoogleNet test accuracy:	66.47%

```
Test Accuracy (ResNet): 73.05%
Test Accuracy (Densenet): 76.18%
Test Accuracy (GoogLeNet): 66.47%
```

Comparison

Compare the test accuracy of your custom CNN model with that of the fine-tuned state-of-the-art models.

1. **Custom CNN Model (Test Accuracy: 74.14%):**
    - Simplicity: Our custom CNN model is relatively simple, which might be beneficial for the CIFAR-10 dataset.
    - Customization: We have control over the architecture, allowing us to tailor it to the specifics of the dataset.
    - **Possible Limitations:**
        - Limited Complexity: The model might not capture intricate patterns in the data compared to more complex architectures.
        - Data Augmentation: Further augmentation techniques could potentially improve performance.

2. **ResNet (Test Accuracy: 73.05%):**
    - Residual Connections: Help mitigate vanishing gradient problems, facilitating training of deep networks.
    - Feature Reuse: Dense connectivity allows features to be reused across layers.
    - **Possible Limitations:**
        - Complexity: ResNet may be too complex for the relatively small CIFAR-10 dataset, *leading to overfitting or suboptimal generalization.*

3. **DenseNet (Test Accuracy: 76.18%):**
    - Dense Connectivity: Facilitates feature reuse, enhancing gradient flow and learning.
    - Parameter Efficiency: It tends to be parameter-efficient due to the sharing of features.
    - **Possible Reasons for Higher Accuracy:**

- Dense Connectivity: CIFAR-10 benefits from the enhanced feature reuse provided by DenseNet.
- Efficient Parameter Usage: DenseNet might be utilizing parameters more effectively for this task.

4. **GoogleNet (Test Accuracy: 66.47%):**
    - Inception Modules: Capture multi-scale features effectively.
    - Auxiliary Classifiers: Aid in mitigating the vanishing gradient problem.
    - **Possible Limitations:**
        - Architecture Complexity: GoogleNet's intricate architecture may not be well-suited for the simplicity of CIFAR-10.

Conclusion:

- DenseNet achieved the highest accuracy (76.18%), suggesting that the dense connectivity and parameter efficiency are well-suited for CIFAR-10.
- Our custom CNN model performed competitively, likely benefiting from its simplicity.
- GoogleNet's lower accuracy could be due to its complex architecture, which might not be necessary for this dataset.

Discussion

The decision of whether to use a custom model or a pre-trained model depends on several factors, including the specific task, the available data, the computational resources, and the expertise available.

Custom Models

Advantages:

- **Domain Specificity:** Custom models can be tailored specifically for the task or domain at hand, ensuring they are optimized for the specific requirements of the problem we are trying to solve.
- **Control over Architecture:** We have full control over the architecture of the model. This allows us to design a network structure that is well-suited for the intricacies of our data.
- **Performance Optimization:** Fine-tuning hyperparameters and optimizing the model's performance on our specific dataset can lead to better results compared to a pre-trained model that might have been trained on a different distribution of data.
- **Data Efficiency:** If we have a sufficiently large and representative dataset, training a custom model from scratch can lead to better generalization and performance.

Limitations:

- **Data Dependency:** Training a custom model requires a substantial amount of labeled data. In domains where obtaining labeled data is difficult or expensive, pre-trained models might be a more practical choice.
- **Computational Resources:** Training a deep learning model from scratch demands significant computational resources and time, especially for large datasets and complex architectures.
- **Expertise Required:** Developing and training a custom model demands expertise in machine learning and deep learning.

Pre-trained Models

Advantages:

- **Transfer Learning:** Pre-trained models, especially those trained on large and diverse datasets, can be fine-tuned on smaller datasets for specific tasks. This often results in faster convergence and improved performance.
- **Time and Resource Efficiency:** Training a model from scratch can be computationally expensive and time-consuming. Pre-trained models save time and computational resources, as they have already learned useful features from extensive datasets.
- **Generalization:** Pre-trained models have often learned generic features from a broad range of data, which can lead to good generalization across various tasks.
- **Community Support:** Popular pre-trained models often have strong community support, with many resources, tutorials, and tools available for implementation and fine-tuning.

Limitations:

- **Domain Mismatch:** Pre-trained models might not perform well if there's a significant mismatch between the data they were trained on and our specific task. This is particularly relevant when dealing with highly specialized or domain-specific tasks.
- **Limited Flexibility:** Pre-trained models have a fixed architecture, limiting our ability to customize the model for specific requirements. If our task demands a unique network structure, a custom model might be more appropriate.
- **Overfitting Risk:** Fine-tuning a pre-trained model on a small dataset carries the risk of overfitting, especially if the pre-trained model has already captured a broad set of features that may not be relevant to our specific task.

The choice between a custom model and a pre-trained model is a complex one, and there is no one-size-fits-all answer. The best approach will depend on the specific requirements of our task, the available data, and the resources at our disposal.

In general, custom models are a good choice when we have a lot of relevant data and the expertise to train a model, while pre-trained models are a good choice when we need a model quickly and easily or when we do not have a lot of data or expertise.

Here is a table summarizing the key trade-offs between custom and pre-trained models:

Feature	Custom Model	Pre-trained Model
Domain Specificity	High	Medium – Low
Control over Architecture	High	Low
Data Requirements	High	Low
Computational Requirements	High	Low
Expertise Requirements	High	Low
Ease of Use	Low	High
Time Efficiency	Low	High
Generalization	Medium – High	Medium
Interpretability	Medium – High	Medium – Low
Scalability	Medium – High	Medium – Low