EEE-543 - Fall 2023 Projects Group 15: Project 3, Classification on CIFAR-10 Dataset

Salih Deniz Uzel (deniz.uzel@bilkent.edu.tr)
22201382

***Abstract*—In this research a deep learning model architecture and training strategies investigated for achieving state-of-the-art image classification task results on CIFAR-10 data set. Statistical analysis of the pixel distribution of the data set was performed and the effect of the data augmentation method on classification accuracy was examined. By conducting a literature review including current state-of-the-art models, a systematic hyper-parameter search was made on the optimizer type, learning rate, L2 regularization, dropout, batch size parameters on a 20-layer network design with residual connections. The classification accuracy rate of the state-the-art models was approached at the limit of computational power. Experiment results show that, the data augmentation method yielded improvement of up to 12%, and most important factor in training a generalized model was the learning rate selection strategy. 115 different conditions were tested and the results are given in comparison with the models' of the state-of-the-art studies. The proposed architecture in the study achieved the model performance in the reference study with a difference of 1.28% which is 89.97% Top 1 accuracy on the CIFAR-10 test set. It was concluded that data augmentation and learning rate scheduling were the most effective and responsive parameters during training.**

I. Introduction

Image classification task is an extensively examined field in the discipline of computer vision, where continuous research is carried out and the competition among researchers is most intense. To accurately assess and compare methods in this field, the ImageNet dataset was introduced in 2009 [1]. It is quickly became the most widely used benchmarking dataset and the industry standard. Currently, ImageNet is still being used as a main benchmark data in the field of image classification algorithms and machine learning. The data set contains 1,281,167 training images, 50,000 validation images and 100,000 test images belonging to a total of 1000 different classes.

The use of neural networks for image classification was not researched as intensely as it is today due to low computational power and popularity until the 90s. The convolution method, which is being used by many state-of-the-art deep learning models today, was published by Kunihiko Fukushima in 1980. Afterwards, LeCun et. al [2] used the convolution method on the MNIST dataset, which has become a benchmark today, created for the handwritten digit recognition task. The LeNet-5 architecture [2] achieved 99.05% accuracy on the test set. About 20 years later, convolutional neural networks took researchers' attention and gained popularity in 2013 by achieving great success in the competition held for the ImageNet dataset. In 2013, researchers won the ImageNet competition with the Deep Learning Model they created using MLP, CNN, Pooling and Dropout that is called AlexNet [3], with the top 1 accuracy of 56.55%. Since that moment, deep learning models continued to hold the first place in this field. In 2015, ResNet-50 citeresnet model a Deep CNN with residual connections take the first place with 79.26% top 1 accuracy . Afterwards, deep learning models continued to hold the first place in this field. For the ImageNet competition, State of the Art (state-of-the-art) neural network model is OmniVec with 92.400 Top-1 Accuracy [4].

In this study, the image classification task will be performed on the CIFAR-10 dataset [5] with the neural network model designed. CIFAR-10 dataset is labeled subset of 80 million tiny images. In the study, a systematic research on network design, data augmentation, hyper-parameter selection and fine tuning strategy is conducted, process and the outputs are shared. The current state-of-the-art model for CIFAR-10 dataset is a model based on a transformer network architecture called ViT-L/16 with 99.42% Top 1 accuracy on CIFAR-10 dataset [6]. In the literature review, it was concluded that ResNet architecture is the model with the most computationally feasible training cost among other state-of-the-art models. ResNet has an architecture that includes residual connections that prevent gradients from being lost by reintroducing them to the network as a solution to vanishing gradient problems in deep neural networks. For this reason, higher accuracy can be achieved with relatively smaller models in shorter times compared to deeper models. It is deduced for this study, that there is a high probability of achieving higher accuracy values on CIFAR-10 with the limited computational power available. Therefore a ResNet-like network with residual connection was designed. The ResNet-18 model [7] achieved a top 1 accuracy of 91.25 [7]. The lower limit of project goal is to approach at least 91.25% Top 1 accuracy within $\pm 2\%$ margin. Assuming that it is possible to exceed this value due to the following techniques, greater accuracy result outcome is examined in this study.

The different and new approach in this study compared to other studies is hyperparameter search and fine-tuning for the Resnet-like model with Residual connection for the CIFAR-10 dataset. The effects of Adam and the relatively new AdamW optimization methods can achieve faster convergence compare to Stochastic Gradient Descent (SGD) optimizer. Adam and AdamW optimizers are more prone to overfitting due to their implementations. This induced overfitting is tried to be dampened by the dropout regularization technique. It is uniquely used only between the average pooling layer and the output layer for heavy regularization purposes. To investigate this hypothesis, the hyper parameter combinations were extensively investigated in this study. It is aimed to eliminate the overfitting due to less stochastic convergence compare to SGD with dropout regularization and find a better optima with learning rate scheduling.

II. Methodology

In this section, the stages of the CIFAR-10 Image classification task are given in line with the scientific research methodology and explained in detail under their own sections. The literature review is given in detail in the relevant places in the remainder of the research. The neural network model and design decisions of the experiment and the methods are detailed in this section. The experimental setup created for hyper-parameter search is given under the Section IV Lastly, results and interpretation are both shared in the Section V and Section VI.

A. Data

CIFAR-10 dataset [5] consists of 60000 32x32 color images in 10 classes, with 6000 images per class. These classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. And there are 50000 training images and 10000 test images in total.

1) Data Normalization

All images were normalized with the pixel mean and variance of the training set [8]. PyTorch's CIFAR-10 dataset has already scales image pixel values between 0 and 1. The equations used in this process is given in Eq. 1 and Eq. 2. In the equations, I is an array that stores the pixel values of the RGB image. N and M are the dimensions of the image. And m is the number of images in the training set. Calculation applies to each color channel individually.

$$Mean = \mu = \frac{1}{m} \sum_{k=1}^{m} \frac{\sum_{i=1}^{N*M} I_{k,i}}{N*M} \tag{1}$$

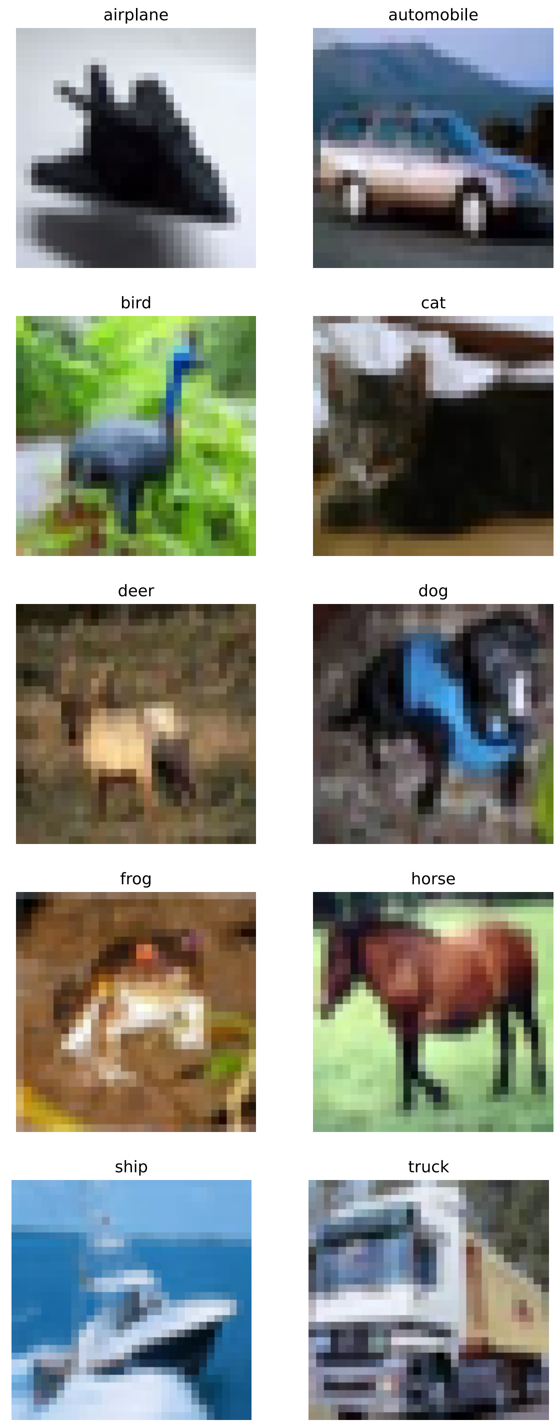
$$Variance = \sigma^2 = \frac{1}{m} \sum_{k=1}^{m} \frac{\sum_{i=1}^{N*M} (I_{k,i} - \mu)^2}{N*M} \tag{2}$$

Rounded μ ve σ values which belongs to CIFAR-10 dataset downloaded via Pytorch are given in Table I.

TABLE I: Values of μ and σ of CIFAR-10 Training Dataset

Method	Mean (μ)			Std. (σ)		
	R	G	B	R	G	B
CIFAR-10	0.0100	0.0098	0.0091	0.0773	0.0759	0.0728

Fig. 1: CIFAR-10 Examples



Class labels are given as 1, 2, .., 10 from left to right top to bottom, respectively .
Images are re-scaled between 0-255.

2) Data Interpolation

To be able to train state-of-the-art models, it is necessary to have an input size of 224x224x3. Therefore, the CIFAR-10 data is scaling with interpolation method provided by PyTorch library. 6 different interpolation method were tested. During the experiments, mean and standard deviation of the each interpolated training set are calculated and normalized. All the seed are fixed and same training set distribution is used. The experiment involved comparing results obtained from scaling the input using six different interpolation methods and training the ResNet18 model for 30 epochs. No statistically significant differences were observed between the interpolation methods. The mean and standard deviation values for the interpolated training sets are presented in Table II. As a result, for the sake of comparison, the default "BILINEAR" interpolation method was used in the experiments.

TABLE II: Values of μ and σ for different interpolation methods on CIFAR-10 Training Dataset

Method	Mean (μ)			Std. (σ)		
	R	G	B	R	G	B
Nearest	0.4887	0.4752	0.4392	0.2431	0.2394	0.2559
Bilinear	0.4890	0.4755	0.4396	0.2364	0.2328	0.2500
Bicubic	0.4887	0.4753	0.4393	0.2413	0.2376	0.2544
Box	0.4887	0.4752	0.4392	0.2431	0.2394	0.2559
Hamming	0.4888	0.4753	0.4393	0.2392	0.2355	0.2524
Lanczos	0.4887	0.4752	0.4393	0.2429	0.2392	0.2558

3) Data Distribution and Training, Validation, and Test Split

The CIFAR-10 Training set contains 50000 samples and Test set contains 10000 samples. The trainin set is divided into 45000 Training and 5000 Validation samples. The 95% ratio is chosen as preferred in the Resnet study [7] in order to reduce the variables in the comparison of model performances. The number of class samples are equal within the training, validation, and test sets.

4) Data Augmentation

During hyper-parameter search, models were trained with and without data augmentation. Data augmentation methodologies and their parameters are given in 2. The following methods implemented and tested. Random crop with 50% probability is used on top of 4 by 4 padding and horizontal flip is applied with 50% probability. During the literature review, it was observed that the deep neural network state-of-the-art models for Image Classification task use this data augmentation strategy, including ResNet. This augmentation strategy provided more than 10% Top1 accuracy improvement. Details are mentioned in the V. As a secondary augmentation method, random rotation was applied between -15 and +15 degrees with a 50% probability. With this method, a decrease in the Top1 Accuracy metric observed on the CIFAR-10 dataset and it was not used in the experiments afterwards.

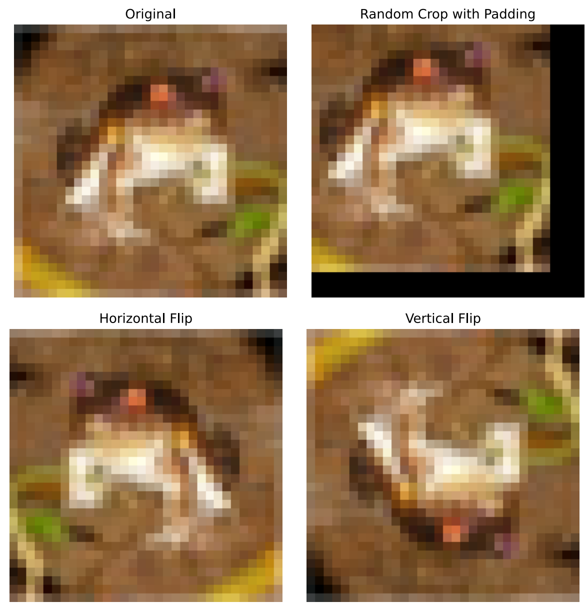
III. Model

A. Model

1) Learning Model with Residual Connections

The model 3 used for classification consists of 2d convolution, 2d batch normalization and relu layers, repectively that are bundled as "ConvBlock" Fig. 4. In every two ConvBlocks', input activations of the first blocks are added to the output activations of the second ConvBlock. This method is called as Residual Learning by He. et. al in the study [7]. Residual layers prevents features of the previous layers from being lost due to the vanishing gradient problem in DNNs. It is shown in

Fig. 2: PyTorch Image Transformation Examples



the Fig. 3 as dotted colored curved lines. The size difference of the activations was eliminated with convolution layer with a stride value of 2. This resizing process is called DownSample Fig. 4. Downsample blocks in the Fig. 4 shares the same colors with dotted Residual Connections in the Fig 3. Each DownSample block consists of 2d Convolution and 2d Batch Normalization, doubling the amount of filter and halving the output size. After the convolution blocks, the output vector was obtained with the Average Pooling Method as 1 (output dimension) by 512 (filter) flattened vector. This vector was passed through the Dropout layer and given to the Fully Connected layer, which is the last layer that will receive 512 inputs and give 10 output values.

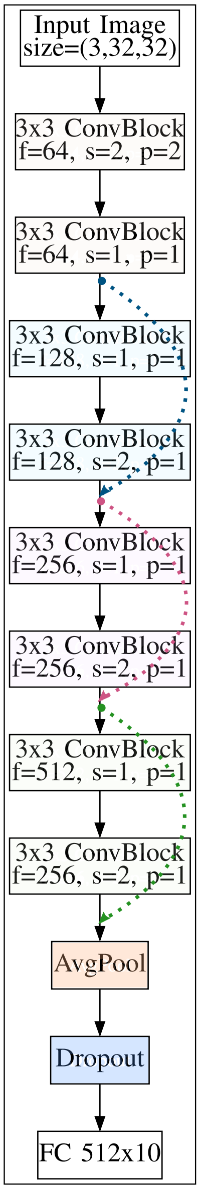
2) Hyper-parameter Search

In order to obtain accuracy close to state-of-the-art models in the CIFAR-10 Image Classification test with the designed model. The scope and type of hyper-parameters were limited by the computing power and project time. The hyper-parameter search space given below.

Optimizer Type	: SGD+Momentum, Adam, AdamW
Learning Rate (LR)	: $0.1, 0.01, 0.001, 0.0001$
L2 Regularization (λ)	: $0.001, 0.0001$
Batch Size	: $32, 64, 256$
Dropout Rate	: $0.1, 0.2, 0.5, 0.7$

Optimizer Type: Three different optimizers were used in the experiments: Stochastic Gradient Descent with Momentum (SGD) [9], Adam (Adaptive Moment Estimation) [10], and AdamW (Adaptive Moment Estimation Weight Decay) [11]. SGD was selected for its stochasticity, which contributes to the model's generalization capability. Momentum was used to shorten the training process and reduce loss oscillation.

Fig. 3: Residual Model Design



The momentum value was chosen as 0.9 and kept constant. Trying different momentum values also increases the Hyper-parameter space as a coefficient. At the same time, SGD with momentum, were used in pioneering studies such as ImageNet [8] and ResNet [7] in the field of image classification with deep learning methods. Therefore it is a strong candidate for an optimizer. Adam is a widely adopted optimizer along with SGD which is created by combining RMSProp [12] and the Momentum method. It is one of the most common optimizer types preferred with its fast convergence feature. AdamW, which has been started to use recently, especially with the use of Large Language Model (LLM), is a method in which weight decay and gradient update are decoupled. AdamW optimizer adjusts weight decay and learning rate separately.

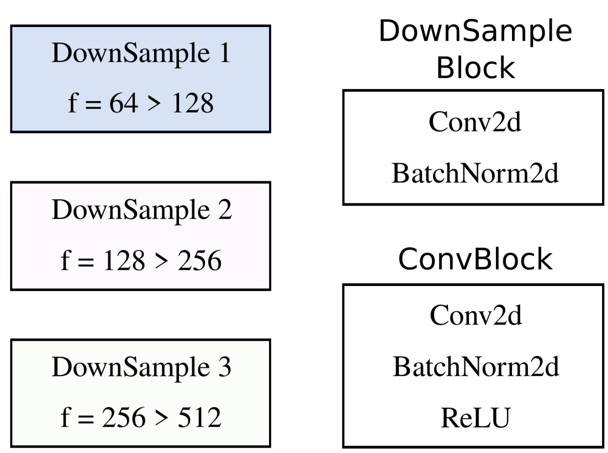
Learning Rate: Learning rate is one of the most important hyper-parameters that affect the model's learning and generalization. It determines how quickly the model approach the optima by scaling the gradient updates. If it is larger than needed, it may cause overshooting as a result model can miss converging to optima. If the learning rate is too small, convergence takes longer time. For these reasons, 3 different optimizer types were trained with 4 different learning rate values: 0.1, 0.01, 0.001, and 0.0001.

L2 Regularization Coefficient: L2 regularization was used to obtain a relatively more balanced weight distribution by penalizing the large weights during model learning. The use of this regularization method in the reference papers was also effective in choosing L2 regularization as the hyper-parameter. L2 regularization coefficient lambda (λ) was chosen as 0.001 and 0.0001 for the search space.

Batch Size: Mini batch sizes are one of the most efficient regularizing technique on the learning process. It prevents overfitting and allows for better generalization on model weights. Model sees fewer samples than the full batch for the case of mini-batches.Therefore mini-batch technique adds a certain amount of stochasticity to the gradient update and contributes to the model finding a good optima. In today's state-of-the-art models, minibatch is used for training even in conditions where the GPU memory is high enough. The sizes of mini batches were chosen as 32, 64, 128, and 256 in this study.

Dropout: Dropout technique has been added to the hyper-parameter search options as another solution to the model's overfit problem. The dropout technique aims to enable different neurons to see and learn the samples by deactivating the neurons with a given probability value in each forward and backward pass [13]. Therefore, overfitting can be prevented by introducing randomness to network and obtaining a regularizing effect on the network. The dropout effect was tested after completing the model trainings with other hyper-parameters and obtaining the results. This strategy selected in order to make a meaningful comparison of the dropout effect. Different dropout rates 0.1, 0.2, 0.5, and 0.7 used on the best model that has the closes training and test accuracy not overfit and the best model that gave the highest test accuracy and overfit.

Fig. 4: Details



IV. Experiments

The experiment design aims to find the hyper-parameters and augmentation method that gives the best results. In this process, the road map is designed according to the time and computational power available in a way that allows us to conduct a systematic review in accordance with scientific research methods. Therefore, the experiments were conducted under 3 main stages.

TABLE III: 3 Main Stages of the Experiments

Stage	Task
1	Finding the optimizer type, and data augmentation method that yields the best test Top 1 accuracy result.
2	Comparative analysis of the impact of different dropout values on the best-selected model and improving its performance with the learning rate scheduling technique.
3	Training state-of-the-art models with CIFAR-10 data-set and comparing the results with the best model.

Training Environment: The code for the experiments were developed on a personal computer and experiments were run on Google Colab environment. The estimated FP32 computing performance of the GPU is around 16 TFLOPS. In the results section, among other parameters, total training time of each model are given in seconds along with the total number of epochs.

Loss Function: All experiments were carried out using the loss function from nn.CrossEntropyLoss class of the Pytorch library. Cross entropy loss also known as log loss or negative log-likelihood loss. Due to Pytorch CrossEntropyLoss implementation [14], first the LogSoftmax function and then Log Loss is applied to the output of the network. For this reason, nn.Softmax was not additionally applied during the design of the custom model. It is stated on the PyTorch developers website, without citing the source, that the log-softmax function provides numerical stability. This situation can be explained mathematically by the following mathematical reasoning. Floating number overflow problem is encountered with current computer architectures when Softmax function is applied to a vector consisting of sufficiently large numbers. The log function applied to Softmax function slows down the accumulation of large numbers during the training a of deep learning model, thus reducing the likelihood of an overflow outcome.

Weight Initialization: Weights and biases of the trained models are initialized as followings. The weights of the convolutions are initialized as in the study made by He. et al [15]. Kaiming Normal initialization provided by PyTorch [16] is scaling the standard deviation of the normal distribution by a factor of $\frac{1}{fan\_mode}$. In this study fan_mode is picked as the number of outgoing connection of the corresponding layer. Batch normalization weights are initialized to 1. All bias terms are initialized to 0.

A. Stage 1: Hyper-parameter and Data Augmentation Search

In the first stage, a hyper parameter search was made without any data augmentation method. The result of these trainings are used for the base line in the study. This baseline is obtained from 36 different hyper-parameter combination and it will be called as the result of the Base Hyper-Parameter Search Space (Base HPSS) throughout the study. Afterwards, the performance of the Base HPSS was tried to be improved by selecting the best performing optimizer and applying various data augmentation techniques.

- ► Base Hyper Parameter Search Space (Base HPSS)
  - ◇ Optimizer type, Batch size, learning rate,
  - ◇ L2 regularization coefficient, best epoch

The data augmentation configurations used in this study are called Data Augmentation Approach (DAA). There are 2 DAAs in total and they will be referred to as DAA-1 and DAA-2 in the remainder of the study.

- ► Data Augmentation Approach 1 (DAA-1)
  - ◇ Padding: 4 pixels from each side
  - ◇ Random square crop: 3x32x32
  - ◇ Random horizontal flip: 50% probability of flipping
  - ◇ Normalization: With the mean and std. of the training set
- ► Data Augmentation Approach 2 (DAA-2)
  - ◇ Random rotation: 50% probability of rotating between $\pm 15°$
  - ◇ DAA-1

In the first stage, a total of 100 different configuration, including Base HPPS (36 runs) Table **??**, V, IX, DAA-1 (32 runs) Table VII, DAA2 (32 runs) Table VIII, were trained for 30 epochs.

B. Stage 2: Testing Dropout Strategy and Learning Rate Scheduling

In this stage, two models were selected according to the results in Stage 1 and the effect of dropout was examined for different probability values. One of the selected models is the model with the best Top 1 accuracy in the test set. A second model was used in this experiment for comparison purposes to evaluate how the Best model behaved with dropout due to overfit. Second model was selected amongst the models with the lowest training set test difference regardless of test set top 1 accuracy.

Learning rate scheduling is a frequently used method in training state-of-the-art models [7, 8, 17]. In studies [7, 17] researchers states that the learning rate is reduced by a coefficient of 0.1 in after certain epochs. It has been inferred that this specific epoch numbers are based on the researchers' observation. During training, the practitioner can see the points where the model progresses without overfitting and then starts to overfit. In the second part of stage 2, learning rate scheduling was created based on the results in Stage 1 and the best model was trained for 200 epochs.

C. Stage 3: Comparison of the Best Model and state-of-the-art Models on CIFAR-10 Dataset

The results of the best model obtained in the study trained on CIFAR-10 were compared with the results of the state-of-the-art models. In the comparison, ResNet-18 [7] and Densenet-121 [17] were used for state-of-the-art models. The training of Densenet-121 model is computationally very expensive compared to a models such as Resnet18. Therefore the pre-trained model, initialized with the weights which is trained on ImageNet dataset [1], could only be trained for 40 epochs on CIFAR-10 dataset. The ResNet-18 model was trained in 2 different ways. First, the weights were re-initialized with Kaimin-Normal [**?**]as mention in the Section IV and trained on CIFAR-10. Secondly, a pre-trained ResNet-18, initialized with the weights from a ResNet-18 model trained on the ImageNet dataset, was trained on the CIFAR-10 dataset.

V. Results

A. Stage 1

1) Base-HPPS

The Base-HPSS results are given in Table **??**, Table V, Table IX. These tables are visualized in the Fig 5. The figure order is left to right and top to bottom. The rows represents the accuracy, loss, and hyper-parameter visualization respectively. The columns represents the optimizers SGD, Adam, AdamW and models with a test set top1 accuracy value greater than 70 percent in the entire Base-HPSS configuration, respectively. The red dot shows the position of the model with the highest value in the Top 1 test set accuracy on the accuracy and loss plots. The best model is 26'th run, which has 77.08% test set top 1 accuracy.

The colored dots in the 3rd row indicate the initial loss values of each model with their respective batch sizes. Since all trainings started with the same seed value, initial loss values were the same for models with the same batch size. The same colors are arranged on the same y axis, which shows that all conditions for initialization are equal for all models. Line colors represent different learning rates, and whether the line is dotted or dash represents the l2 regularization coefficient. This coloring was done because it was difficult to evaluate 36 different trainings simultaneously in terms of hyper-parameters. By looking at the optimizer-based hyper parameter graphs, it can be said that each configuration is close to each other. However, when the graph is filtered as those above 70% test set top 1 accuracy, it is observed that the graph is completely green. This indicates that among the hyperparameters of learning rate, batch size, L2 regularization coefficient, and optimizer types, learning rate is more decisive in terms of given CIFAR-10 image classification task. Therefore, it can be concluded that using learning rate scheduling can be effective in training a generalized model with high test set accuracy. Looking at the tables of Base-HPSS, it can be said that AdamW gives the highest result. For this reason, it was decided to carry out the tests with adamW, using a greedy approach in which I chose the optimizer with the highest test set top 1 accuracy.

2) Broader Learning Rate Search and Data Augmentation on Base-HPSS

As a result of the experiments carried out in Stage 1, the number of learning rate options was doubled on the optimization type AdamW and the experiments were repeated by applying data augmentation techniques. Although DAA-2 includes random rotation in addition to DAA-1, when the results of DAA-1 and DAA-2 are compared, it is observed that DAA-1 gives slightly better results. Therefore, the best model candidate was selected from Base-HPSS+ DAA-1. The highest test set top 1 accuracy is run number 13 given in Table VII.

"Train Test diff" in Table VII is created as an other comparison metric. This metric gives the difference between train set and test set top 1 accuracy and it is given to express the overfit situation analytically. Run no that gave the lowest difference is chosen for comparison for training with dropout regularization. The lowest run number in table VII is 1. However, run number 9, which comes after run no 1 with a difference of 5 out of 10 thousand, has a 7% higher top 1 test set accuracy. For this reason, Run no 9 along with 13 was chosen as the other best model candidate.

TABLE IV: **Stage 1**, Optimizer **SGD**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 30 Epochs.

Run No	Batch Size	LR (α)	L2 Reg.	BE	Time (s)	Top1 Train Acc.	Top1 Val. Acc.	Top1 Test Acc.
1	32	0.0010	0.0010	14	597.2	99.9	66.0	67.6
2	32	0.0010	0.0001	21	559.5	99.9	66.8	67.1
3	32	0.0001	0.0010	27	560.6	94.2	59.1	57.4
4	32	0.0001	0.0001	29	629.0	95.7	57.2	58.2
5	64	0.0010	0.0010	26	470.0	99.9	61.6	63.1
6	64	0.0010	0.0001	25	473.5	99.9	61.9	63.8
7	64	0.0001	0.0010	25	447.2	94.1	55.2	55.6
8	64	0.0001	0.0001	24	485.9	95.1	56.3	56.5
9	256	0.0010	0.0010	23	394.5	99.5	56.7	58.7
10	256	0.0010	0.0001	21	357.1	99.6	57.8	58.9
11	256	0.0001	0.0010	30	353.2	77.8	50.8	51.6
12	256	0.0001	0.0001	29	362.4	79.3	49.9	51.4

* LR, is learning rate.
* L2, is L2 regularization.
* BE, is Best Epoch.

TABLE V: **Stage 1**, Optimizer **Adam**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 30 Epochs.

Run No	Batch Size	LR (α)	L2 Reg.	BE	Time (s)	Top1 Train Acc.	Top1 Val. Acc.	Top1 Test Acc.
13	32	0.0010	0.0010	27	608.5	90.6	74.3	75.8
14	32	0.0010	0.0001	26	574.2	95.7	75.6	75.8
15	32	0.0001	0.0010	29	623.0	96.8	58.6	62.4
16	32	0.0001	0.0001	23	624.4	97.3	59.2	62.4
17	64	0.0010	0.0010	23	453.8	92.3	72.2	74.5
18	64	0.0010	0.0001	23	451.6	96.1	74.5	74.3
19	64	0.0001	0.0010	28	458.4	98.1	61.4	61.2
20	64	0.0001	0.0001	27	455.4	98.5	60.2	60.6
21	256	0.0010	0.0010	25	364.9	93.8	68.7	70.3
22	256	0.0010	0.0001	27	357.6	96.5	68.5	69.2
23	256	0.0001	0.0010	18	357.8	100.0	58.3	58.5
24	256	0.0001	0.0001	13	358.2	100.0	56.0	58.1

* LR, is learning rate.
* L2, is L2 regularization.
* BE, is Best Epoch.

Fig. 5: Base HPSS Comparisons

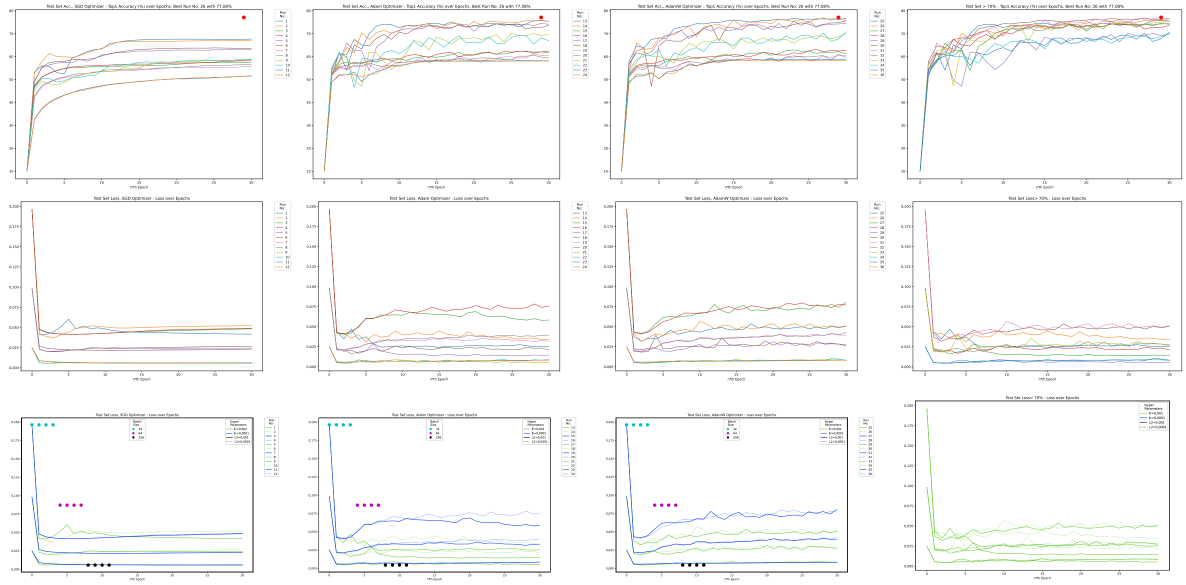


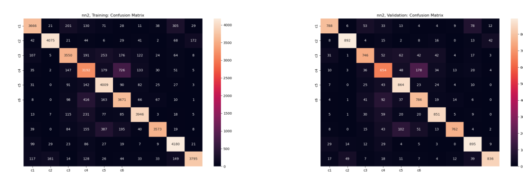
Figure order is left to right, top to bottom.
Row 1: Accuracy metric, Row 2: Loss metric, Row 3: Hyper parameter visualization.
Col. 1: SGD Optimizer, Col. 2: Adam Optimizer, Col. 3: AdamW Optimizer, Col. 4: Models with Test set Top 1 Accuracy value greater than 70%.
The red dot shows the location of the model with the highest value in the Top 1 test set accuracy.
The colored dots in the 3rd row indicate the initial loss values of each model with their respective batch sizes.

TABLE VI: **Stage 1**, Optimizer **AdamW**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 30 Epochs.

Run No	Batch Size	LR (α)	L2 Reg.	BE	Time (s)	Top1 Train Acc.	Top1 Val. Acc.	Top1 Test Acc.
25	32	0.0010	0.0010	28	632.8	98.3	76.9	76.9
26	32	0.0010	0.0001	24	568.2	98.6	77.3	77.1
27	32	0.0001	0.0010	23	565.4	97.6	63.8	62.9
28	32	0.0001	0.0001	30	565.9	97.2	60.7	63.1
29	64	0.0010	0.0010	21	427.2	98.1	74.0	75.5
30	64	0.0010	0.0001	30	427.9	98.8	75.5	75.6
31	64	0.0001	0.0010	29	428.7	98.6	58.9	60.4
32	64	0.0001	0.0001	20	428.7	98.3	59.3	61.0
33	256	0.0010	0.0010	27	357.6	97.9	67.8	70.0
34	256	0.0010	0.0001	30	358.6	97.7	70.3	70.6
35	256	0.0001	0.0010	14	363.3	100.0	58.4	58.8
36	256	0.0001	0.0001	18	384.6	100.0	57.1	58.5

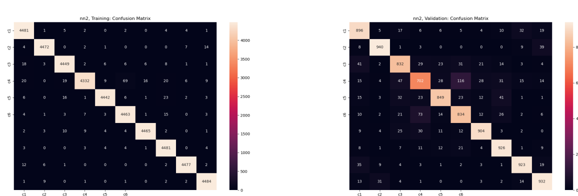
* LR, is learning rate.
* L2, is L2 regularization.
* BE, is Best Epoch.

Fig. 6: Training and Validation Set Confusion Matrices of Run No: 9 for Seed Number 42.



Training(left), Validation(right). Class numbers are increasing from top to bottom, left to right.

Fig. 7: Training and Validation Set Confusion Matrices of Run No: 9 for Seed Number 42.



Training(left), Validation(right). Class numbers are increasing from top to bottom, left to right.

B. Stage 2: Dropout Parameter Search and Learning Rate Scheduling

The two selected models in the first stage, were retrained for different dropout values p = $0.1, 0.2, 0.5, 0.7$. The outputs of Run no 9 are given in Table XIII, the outputs of Run no 13 are given in Table XIV. By looking at the Train Test diff values in Table XIII, we can say that it dropout regularization is effective in . However, there was no improvement in the test accuracy. In Table XIV, a regularization effect that is directly proportional to the increasing dropout value is not observed.

1) Hypothesis

Fig. 6 and Fig. In 7 are the confusion matrices of the training and validation sets for run no 9 and run no 13. In the matrix, left top corner starts from label 1 ends at the right bottom with label number 10. It can be easily seen that the training set confusion matrix of Run no 13 is overfit. The same amount of heat is not observed on the diagonal of the confusion matrix on its validation set. However, the confusion

TABLE VII: **Stage 1**, Optimizer **AdamW**, Augmentation **DAA-1**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 30 Epochs.

Run No	Batch Size	LR (α)	L2 (10^x)	Best Epoch	Top1 Train Acc.	Top1 Val. Acc.	Top1 Tes Acc.	Train Test diff
1	32	0.1000	-3	30	72.0	70.0	70.5	1.47
2	32	0.1000	-4	29	82.2	78.6	79.1	3.07
3	32	0.0100	-3	30	92.0	85.0	86.0	5.96
4	32	0.0100	-4	24	91.9	85.5	85.5	6.37
5	32	0.0010	-3	23	93.0	85.7	86.9	6.07
6	32	0.0010	-4	27	92.9	85.4	86.5	6.44
7	32	0.0001	-3	29	88.7	84.0	82.4	6.34
8	32	0.0001	-4	29	88.8	81.6	82.7	6.09
9	**64**	**0.1000**	**-3**	**30**	**78.4**	**75.0**	**77.0**	**1.43**
10	64	0.1000	-4	26	83.1	78.4	79.4	3.75
11	64	0.0100	-3	27	90.4	83.6	84.1	6.26
12	64	0.0100	-4	29	92.2	86.0	85.4	6.74
13	**64**	**0.0010**	**-3**	**30**	**93.1**	**85.6**	**86.6**	**6.54**
14	64	0.0010	-4	27	93.0	85.4	85.8	7.17
15	64	0.0001	-3	28	85.1	77.5	80.0	5.11
16	64	0.0001	-4	29	84.8	77.0	79.5	5.26
17	128	0.1000	-3	25	79.4	75.3	77.5	1.86
18	128	0.1000	-4	30	84.7	80.1	81.5	3.20
19	128	0.0100	-3	25	90.5	84.4	85.4	5.13
20	128	0.0100	-4	29	91.9	85.3	85.6	6.34
21	128	0.0010	-3	23	90.6	84.1	85.4	5.16
22	128	0.0010	-4	20	89.6	82.5	84.3	5.36
23	128	0.0001	-3	29	81.8	75.6	76.9	4.91
24	128	0.0001	-4	30	82.4	76.1	77.3	5.03
25	256	0.1000	-3	30	79.8	77.3	77.1	2.68
26	256	0.1000	-4	30	81.9	77.2	78.4	3.52
27	256	0.0100	-3	30	90.0	84.0	83.7	6.23
28	256	0.0100	-4	24	89.3	82.8	83.2	6.07
29	256	0.0010	-3	24	88.8	82.6	84.2	4.56
30	256	0.0010	-4	23	90.6	82.6	84.7	5.85
31	256	0.0001	-3	26	77.6	70.7	73.1	4.49
32	256	0.0001	-4	27	77.7	71.2	72.9	4.79

* LR, is learning rate.
* L2, is L2 regularization. It is shortened to the value 10^value.
* BE, is Best Epoch.

matrices of the validation set for run no 9 and run no 13 have similar distributions. This gives a clue about the problem that the model is having difficulty learning. Wrong classifications occurs between classes 4, 5, and 6. Label 4 belongs to cat, label 5 belongs to value and label 6 belongs to dog 1. Separating cats and dogs with similar features is also a another computer vision problem investigated by researchers. Because the cause of overfit is due to another computer vision problem, these close results of the two best studies indicate that only a few points of improvement can be made from this point.

2) Suggestion

A reasoning made for the different outcomes in Table XIV and Table XIII as follows. Right after the initialization, gradient updates multiplied by the learning rate defines how the process will be and where the model converge. In this case, models are converging towards different optima, since all the remaining training conditions are equal. However, the values in the tables do not give an idea as to why results close to overfit and underfit are obtained. Therefore Fig. 8 with Fig. 9 compared and interpreted. It can be seen that the loss curve in run no 9 flattens and then oscillates, while the overfitted run no 13 gives a flattening curve with relatively less oscillation. In this case, at the point where the curve begins to flatten

TABLE VIII: **Stage 1**, Optimizer **AdamW**, Augmentation **DAA-2**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 30 Epochs.

Run No	Batch Size	LR (α)	L2 (10^x)	BE	Top1 Train Acc.	Top1 Val. Acc.	Top1 Tes Acc.	Train Test diff
1	32	0.1000	-3	28	72.1	70.3	70.5	1.59
2	32	0.1000	-4	30	78.2	73.0	75.7	2.58
3	32	0.0100	-3	21	88.8	82.4	84.0	4.76
4	32	0.0100	-4	28	89.7	82.3	84.4	5.23
5	32	0.0010	-3	25	90.8	84.1	85.2	5.58
6	32	0.0010	-4	25	90.8	84.8	85.3	5.53
7	32	0.0001	-3	29	86.2	80.7	81.2	4.94
8	32	0.0001	-4	28	85.9	80.7	81.0	4.98
9	**64**	**0.1000**	**-3**	**28**	**78.8**	**75.8**	**77.2**	**1.64**
10	64	0.1000	-4	29	81.5	79.4	78.0	3.48
11	64	0.0100	-3	28	90.3	84.2	84.2	6.06
12	**64**	**0.0100**	**-4**	**27**	**91.5**	**83.5**	**85.4**	**6.17**
13	64	0.0010	-3	26	89.9	85.0	84.7	5.21
14	64	0.0010	-4	29	91.2	85.0	84.9	6.26
15	64	0.0001	-3	30	83.3	77.3	78.6	4.75
16	64	0.0001	-4	30	83.6	77.3	78.7	4.95
17	128	0.1000	-3	30	79.1	77.5	76.3	2.85
18	128	0.1000	-4	25	80.9	77.3	77.5	3.43
19	128	0.0100	-3	28	90.0	82.8	84.1	5.91
20	128	0.0100	-4	30	89.6	83.2	83.9	5.65
21	128	0.0010	-3	23	88.1	83.3	83.5	4.61
22	128	0.0010	-4	23	90.5	84.3	84.6	5.88
23	128	0.0001	-3	30	80.7	74.0	75.7	5.00
24	128	0.0001	-4	29	80.2	72.7	75.5	4.69
25	256	0.1000	-3	28	81.3	76.3	77.9	3.40
26	256	0.1000	-4	26	80.4	77.0	77.8	2.61
27	256	0.0100	-3	26	87.9	82.8	82.9	5.06
28	256	0.0100	-4	28	89.0	84.0	83.0	6.09
29	256	0.0010	-3	29	88.2	81.5	83.0	5.19
30	256	0.0010	-4	28	88.0	82.4	83.0	4.98
31	256	0.0001	-3	30	74.6	68.7	71.0	3.53
32	256	0.0001	-4	30	75.5	70.3	71.0	4.53

* LR, is learning rate.
* L2, is L2 regularization. It is shortened to the value 10^value.
* BE, is Best Epoch.

TABLE IX: **Stage 1**, Optimizer **AdamW**, Augmentation **DAA-1**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 200 Epochs.

Run No	Batch Size	LR (α)	L2 (10^x)	BE	Top1 Train Acc.	Top1 Val. Acc.	Top1 Tes Acc.	Train Test diff
9	64	0.100	-3	139	92.89	85.98	86.28	6.61
13	64	0.001	-3	77	99.43	89.56	89.4	10.03

* LR, is learning rate.
* L2, is L2 regularization. It is shortened to the value 10^value.
* BE is, Best Epoch.

TABLE X: **Stage 1**, Optimizer **AdamW**, Augmentation **DAA-1**, **Run No: 9**

Acc.	Training	Validation	Test
Top1	92.89	85.98	86.28
Top2	97.98	94.44	94.55
Top3	99.14	97.22	97.3

for run number 9, the possibility of the model overshooting due to the high learning rate should be investigated. The spike in the graph of the experiment with a dropout value of 0.7

TABLE XI: **Stage 1**, Optimizer **AdamW**, Augmentation **DAA-1**, **Run No: 13**

Acc.	Training	Validation	Test
Top1	99.43	89.56	89.4
Top2	99.96	96.16	96.23
Top3	99.99	98.2	98.37

TABLE XII: **Run No: 9**, **Dropout: 0.2**, Validation Set Confusion Matrix Example

Label	Cat (4)	Deer (5)	Dog (6)
Cat (4)	654	48	178
Deer (5)	43	864	23
Dog (6)	92	37	786

strengthens this impression.

As a result, although the run no 13 overfitted, it gave the highest top 1 test set accuracy result. Dropout does not seem to be effective on Run no: 13. This may be because the model settled at local optima with the learning rate of 0.001. In order to improve the best model result obtained, same approach with the state-of-the-art studies applied. The learning rate value was reduced at the best validation set accuracy. In this way, it is aimed to push the final accuracy value a little further. Run no 13 was initialized with a learning rate of 0.001. The learning rate was updated to 0.0001 when the validation set accuracy exceeded 89% accuracy. The model trained for 200 epochs. And achieved 0.57% better result which is 89.97% Top 1 accuracy on test set Table XV.

TABLE XIII: **Stage 2**, Optimizer **AdamW**, Augmentation **DAA-1 Dropout**, **Run No: 9**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 200 Epochs.

Dp (p)	Batch Size	LR (α)	L2 (10^x)	BE	Top1 Train Acc.	Top1 Val. Acc.	Top1 Tes Acc.	Train Test diff
0.0	64	0.1	-3	139	92.89	85.98	86.28	6.61
0.1	64	0.1	-3	134	82.93	80.50	79.97	2.96
0.2	64	0.1	-3	200	83.06	80.18	81.04	2.02
0.5	64	0.1	-3	97	80.18	78.60	78.11	2.07
0.7	64	0.1	-3	158	69.91	69.56	69.23	0.68

* Dp, is Dropout, (p)robability. BE is, Best Epoch.
* LR, is learning rate.
* L2, is L2 regularization. It is shortened to the value 10^value.

C. Comparisons with State-of-the-Art Models

ResNet-18 and DenseNet-121 networks are compared with the best model obtained. Both of them are models that have achieved state-of-the-art results in the ImageNet classification task. Resnet-18 is approximately 6.5 times smaller than DenseNet121 in terms of layers. Therefore performances of both shallow and very deep models will be compared. Two models were trained for 200 epochs with the same parameters published in their papers, with a learning rate of 0.1. Pre-Trained models are initialized with the weights that is trained on ImageNet data set [1]. Pre-trained models are trained on CIFAR-10 dataset without any initialization[5]. Re-initialized models are initialized with kaiming-normal [16] and trained on CIFAR-10 dataset. In the Table XV, DenseNet-121 gives the lowest performance. This result is due to the fact that a very deep model was trained very short time with limited data.

Fig. 8: Dropout effect on Run no: 9

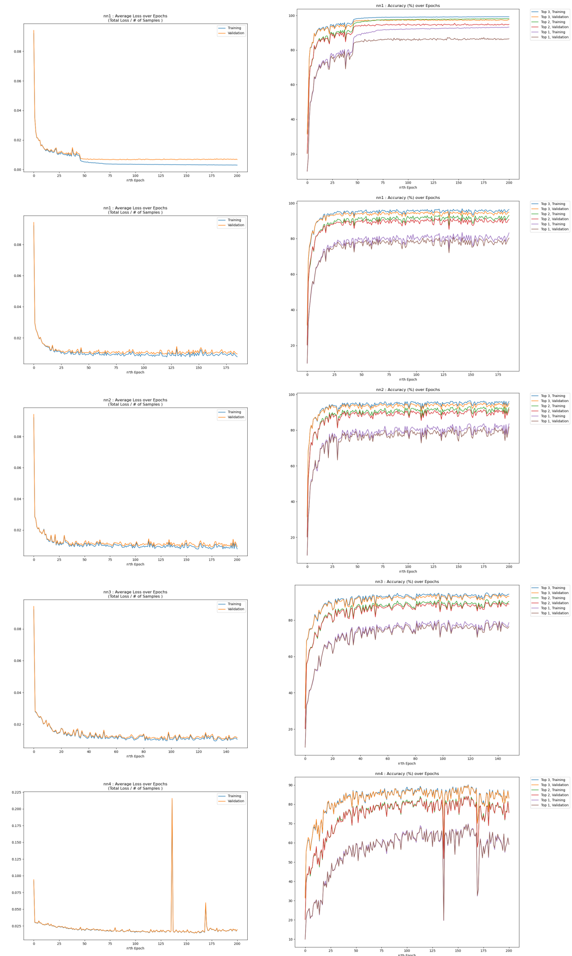


Figure order is left to right, top to bottom.
Row 1: dropout = 0, Row 2: dropout = 0.1, Row 3: dropout = 0.2, Row 4: dropout = 0.5, Row 5: dropout = 0.7.
Col. 1: Loss metric, Col. 2: Accuracy metric (Top 1, Top 2, Top 3).

TABLE XIV: **Stage 2**, Optimizer **AdamW**, Augmentation **DAA-1**, **Dropout**, **Run No: 13**: Training Parameters and Top1 Best Training, Validation, and Test Set Results for 200 Epochs.

Dp (p)	Batch Size	LR (α)	L2 (10^x)	BE	Top1 Train Acc.	Top1 Val. Acc.	Top1 Tes Acc.	Train Test diff
0.0	64	0.001	-3	77	99.43	89.56	89.4	10.03
0.1	64	0.001	-3	175	99.09	88.62	88.44	10.65
0.2	64	0.001	-3	183	99.23	88.66	87.95	11.28
0.5	64	0.001	-3	169	98.99	88.52	87.99	11.00
0.7	64	0.001	-3	200	99.03	88.14	88.16	10.87

* Dp, is Dropout, (p)robability. BE is, Best Epoch.
* LR, is learning rate.
* L2, is L2 regularization. It is shortened to the value 10^value.

Fig. 9: Dropout effect on Run no: 13

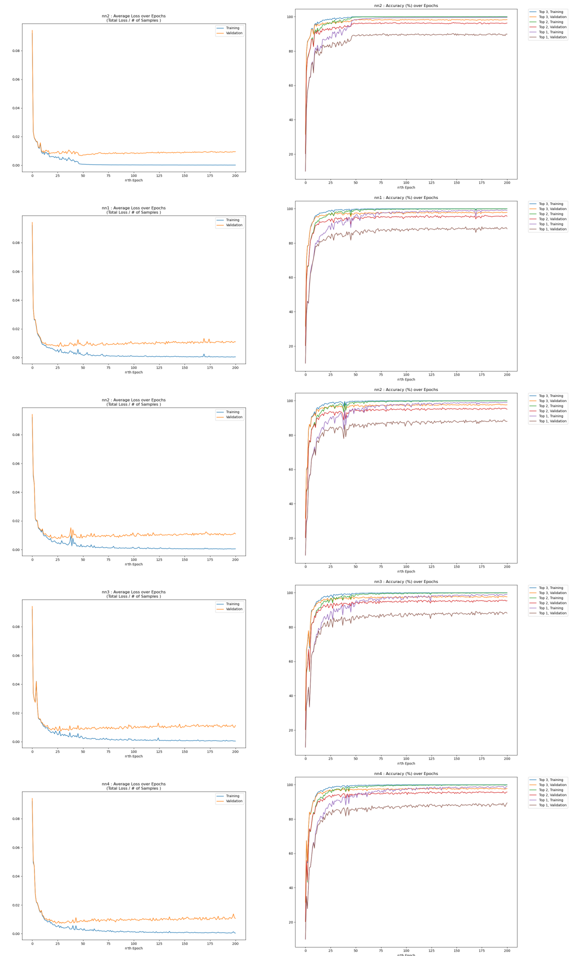
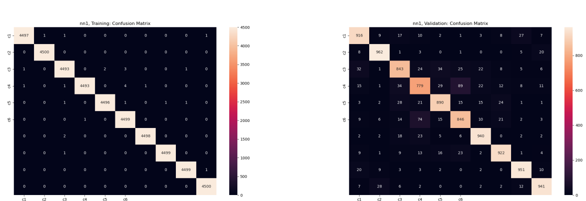


Figure order is left to right, top to bottom.
Row 1: dropout = 0, Row 2: dropout = 0.1, Row 3: dropout = 0.2, Row 4: dropout = 0.5, Row 5: dropout = 0.7.
Col. 1: Loss metric, Col. 2: Accuracy metric (Top 1, Top 2, Top 3).

Deep models like Densenet-121 can achieve high performance when trained with millions of data for a long period of time. Pre-trained and kaiming-initialized ResNet18 model gave relatively close and high results. There is less than 1% difference between the CIFAR-10 test set accuracy value published for the 20-layer model in the ResNet [7] study and the ResNet-18 model trained from scratch in this study. Lastly the best model obtained in this study reached 89.97% test set top1 accuracy.

Fig. 10: Training and Validation Set Confusion Matrices of **Improved** Run No: 13 for Seed Number 42.



Training(left), Validation(right). Class numbers are increasing from top to bottom, left to right.

TABLE XV: Training, Validation, and Test Set Top 1 Accuracy Comparisons with State of the Art Models trained on CIFAR-10 Data Set for 200 Epochs.

	Accuracy(%), Top 1		
Model Name	Training	Validation	Test
Improved Run No:13	99.8955	90.1199	89.97
Published Resnet-18 Results	-	-	91.25
kaiming-initialized Resnet-18	97.7044	89.9399	90.28
Pre-trained ResNet-18	97.5177	89.4199	89.9
Pre-trained DenseNet-121	87.4822	84.86	83.85

* Published Resnet-18 result is the ResNet model with 20 layers performance trained on CIFAR-10 dataset given in the ResNet study [7].

VI. Discussion

In this study, custom and state-of-the-art models were trained on the CIFAR-10 dataset consisting of 32 by 32 images with RGB channels. During the training of the state-of-the-art models are images are interpolated to meet the 3x224x224 input size condition. The images were interpolated by 6 different interpolation method. Mean and standard deviation of the 6 different interpolated training data set produced the same results up to 3 decimal places and did not cause a statistically significant difference in model accuracies. When the prediction patterns of the trained models were examined, it was observed that the labels of cats, dogs and deer were mostly detected incorrectly. Prediction of these classes, which have similar features, is also studied as another computer vision problem.

The designed model and the trained state-of-the-art models gave very similar results to the published results. The comparison with the densenet121 model proved that there is no need for a deep model to solve a 10-class classification problem (Table XV). During the hyper parameter search, models were complex enough to be able to overfit with close to 100% training set top1 accuracy. The learning rate hyper-parameter created a statistically significant difference between the optimizer type, learning rate, L2 regularization parameter, batch size, and dropout hyper parameters. This effect of learning rate also correlated with batch size. The samples in the mini batch and the size of the batch affect the gradient update to be made. Therefore learning rate that scales the gradient update was the most researched hyper parameter in this study. Interestingly, the dropout method did not cause deterministic regularization in the model. While the dropout showed a regularization effect in run number 9, it was not effective in run number 13 (Table XIII, refdp13). The model was proven to be complex enough due to the fact that the run no 13 with overfitted nearly 100

It should be taken into consideration that all trainings were started under the same conditions with the same seed values. In this study, all variables causing randomization were fixed with a seed value of 42. The best model obtained during hyper parameter search reached 89.97% test set top1 accuracy. The difference in accuracy between the ResNet18/CIFAR-10 result published in the ResNet [7] study is 1.28%. Considering that the model in the ResNet study was trained for a longer period of time, there is a possibility that these values in state-of-the-art studies can be exceeded. The most deterministic improvement was achieved with data augmentation. An improvement of up to 12% was achieved in almost all results.

In conclusion, different learning rate scheduling techniques should be compared and further investigated in order to improve the results. Data augmentation is a powerful technique and determines what results the parameters used for model training will yield. The gradient updates are directly depend on the statistical properties of the data in a batch. Therefore, it has been observed that the learning rate and batch size are the most responsive hyper-parameters along with data augmentation techniques.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[2] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[4] "Papers with Code - OmniVec: Learning robust representations with cross modal sharing — paperswithcode.com," https://paperswithcode.com/paper/omnivec-learning-robust-representations-with, [Accessed 19-12-2023].

[5] "CIFAR-10 and CIFAR-100 datasets — cs.toronto.edu," https://www.cs.toronto.edu/ kriz/cifar.html, [Accessed 19-12-2023].

[6] "Papers with Code - An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale — paperswithcode.com," https://paperswithcode.com/paper/an-image-is-worth-16x16-words-transformers-1, [Accessed 19-12-2023].

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[9] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, 1964. [Online]. Available: https://api.semanticscholar.org/CorpusID:120243018

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[11] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[12] G. Hinton, *Coursera Neural Networks for Machine Learning lecture 6*, 2018.

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, jan 2014.

[14] "CrossEntropyLoss &x2014; PyTorch 2.1 documentation — pytorch.org," https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html, [Accessed 05-01-2024].

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[16] "torch.nn.init &x2014; PyTorch 2.1 documentation — pytorch.org," https://pytorch.org/docs/stable/nn.init.html, [Accessed 05-01-2024].

[17] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

Appendix

Code Listing 1: Python Code: PyTorch Model

```
import torch
import torch.nn as nn


class SimpleConvPack(nn.Module):
    def __init__(self, in_channels, out_channels,
                       kernel_size, stride, padding
                ):
        super(SimpleConvPack, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size,
            stride=stride, padding=padding)
        self.bn1 = nn.BatchNorm2d(out_channels, eps=1e-05, affine=True,
            track_running_stats=True)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        #print(f"Simple Conv: {x.shape=}, {out.shape=}")
        return out

def down_sample(in_channels, out_channels):
    """
    in should be smaller block size
    out should be bigger block size
    """
    # TO DO: check default conv padding val
    downsample = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2, padding=0),
        nn.BatchNorm2d(out_channels, eps=1e-05, affine=True, track_running_stats=True)
    )
    #print(f"Downsample: {in_channels=}, {out_channels=}")
    return downsample


class myResnetStyleModel(nn.Module):
    def __init__(self, *, keepdims):
        super(myResnetStyleModel, self).__init__()
        self.relu = nn.ReLU()
        self.input_block_1 = SimpleConvPack(3, 64, 3, 2, 2)
        self.input_block_2 = SimpleConvPack(64, 64, 3, 1, 1)

        self.res_block_1_1 = SimpleConvPack(64, 128, 3, 1, 1)
        self.res_block_1_2 = SimpleConvPack(128, 128, 3, 2, 1)
        self.down_sample_1 = down_sample(64, 128)

        self.res_block_2_1 = SimpleConvPack(128, 256, 3, 1, 1)
        self.res_block_2_2 = SimpleConvPack(256, 256, 3, 2, 1)
        self.down_sample_2 = down_sample(128, 256)

        self.res_block_3_1 = SimpleConvPack(256, 512, 3, 1, 1)
        self.res_block_3_2 = SimpleConvPack(512, 512, 3, 2, 1)
        self.down_sample_3 = down_sample(256, 512)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(keepdims)
        self.fc = nn.Linear(512, 10)

    def forward(self, x):

        out = self.input_block_1(x)
        out = self.input_block_2(out)

```

```python
        #residual block 1
        identity = out # does this carry the object or copy ?
        out = self.res_block_1_1(out)
        out = self.res_block_1_2(out)
        identity = self.down_sample_1(identity)
        out = out + identity
        out = self.relu(out)

        # residual block 2
        identity = out
        out = self.res_block_2_1(out)
        out = self.res_block_2_2(out)
        identity = self.down_sample_2(identity)
        out = out + identity
        out = self.relu(out)

        # residual block 3
        identity = out
        out = self.res_block_3_1(out)
        out = self.res_block_3_2(out)
        identity = self.down_sample_3(identity)
        out = out + identity
        out = self.relu(out)

        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.dropout(out)
        out = self.fc(out)
        return out

    def initialize_weights(self):
        import torch.nn.init as init
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                # Apply He initialization to weights of Conv and FC layers
                init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    # Initialize biases to zero
                    init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                # Initialize BatchNorm scaling parameter to 1 and shift parameter to 0
                init.constant_(m.weight, 1)
                init.constant_(m.bias, 0)
```

Code Listing 2: Python Code: Training Script

```python
import os

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

from torch.utils.data import DataLoader
from torch.utils.data import DataLoader, Subset
from sklearn.model_selection import StratifiedShuffleSplit
from torchvision.datasets import CIFAR10

from torchvision.models import resnet18

from torchvision.transforms import InterpolationMode # type: enum.EnumMeta
from PIL import Image
from tqdm import tqdm

import time
import random
import numpy as np

import utils
```

```
import plots

import cmodel
import json

from torch.optim.lr_scheduler import ReduceLROnPlateau, MultiStepLR

cur_run_seed = 42

class CustomLrSheduler():
    def __init__(self, optimizer, mode, beta=0.3, patience=20, cool_down=5,
        improvement_threshold_coef=0.0001):
        self.optim = optimizer
        self.mode = mode
        self.patience = patience
        self.cool_down = cool_down
        self.improvement_threshold_coef = improvement_threshold_coef
        self.beta = beta
        self.best_val = None

        self.patience_counter = None
        self.cool_down_counter = None

        self.init_vals()

    def reset_counters(self):
        self.patience_counter = self.patience
        self.cool_down_counter = self.cool_down

    def init_vals(self):
        if self.mode == "max":
            self.best_val = -1
        if self.mode == "min":
            self.best_val = float("inf")

        self.reset_counters()

    def get_improvement_val(self, val):
        return val * (1+self.improvement_threshold_coef)

    def lr_update(self, cur_val, verbose=True):
        imp_val = self.get_improvement_val(cur_val)

        if self.mode == "max":
            if imp_val < self.best_val:
                if self.cool_down_counter < 1:
                    self.patience_counter -= 1

        if self.mode == "min":
            if imp_val > self.best_val:
                if self.cool_down_counter < 1:
                    self.patience_counter -= 1

        if self.patience_counter < 1:
            for g in self.optim.param_groups:
                lr_val = g['lr']
                if lr_val > 0.00009:
                    g['lr'] = lr_val * self.beta
                    if verbose:
                        print("! Learning Rate Updated. Now: ", g['lr'], "\n")

            self.reset_counters()

        self.cool_down_counter -= 1
        self.best_val = max(self.best_val, cur_val)

    def _invoke_update(self, cur_val, verbose=True):
        imp_val = self.get_improvement_val(cur_val)

```

```python
        if self.mode == "max":
            if imp_val < self.best_val:
                if self.cool_down_counter < 1:
                    self.patience_counter -= 1

        if self.mode == "min":
            if imp_val > self.best_val:
                if self.cool_down_counter < 1:
                    self.patience_counter -= 1

        if True:
            for g in self.optim.param_groups:
                g['lr'] = g['lr'] * self.beta
                if verbose:
                    print("! Learning Rate Updated. Now: ", g['lr'], "\n")

            self.reset_counters()

        self.cool_down_counter -= 1
        self.best_val = max(self.best_val, cur_val)


    def help(self):
        s ="""
        Improvement threshold works as follows:
            bestVal > bestVal * (1 + improvement_threshold_coef)
            For Loss: // min case
                if loss bestVal is 0.094, minimum improvement should be less than
                0.094 * (1+0.0001) = 0.940094

            For Acc:  // max case
                if acc bestVal is 94.87%, minimum improvement should be bigger than
                94.870 * (1+0.0001) = 94.879487
        """
        print(s, "\n")

class CosineAnnealingLRWithWarmup(torch.optim.lr_scheduler.CosineAnnealingLR):
    def __init__(self, optimizer, T_max, eta_min=0, warmup_steps=0, warmup_factor=0.01,
        last_epoch=-1):
        self.warmup_steps = warmup_steps
        self.warmup_factor = warmup_factor
        super(CosineAnnealingLRWithWarmup, self).__init__(optimizer, T_max, eta_min,
            last_epoch)

    def get_lr(self):
        if self.last_epoch < self.warmup_steps:
            # Linear warmup
            return [base_lr * (self.warmup_factor + (1.0 - self.warmup_factor) *
                self.last_epoch / self.warmup_steps) for base_lr in self.base_lrs]
        else:
            return [self.eta_min + (base_lr - self.eta_min) * (1 + math.cos(math.pi *
                (self.last_epoch - self.warmup_steps) / (self.T_max - self.warmup_steps))) / 2
                for base_lr in self.base_lrs]

class DNN:
    _shared_container = dict()
    _exp_no = 0

    @staticmethod
    def get_shared_container():
        return DNN._shared_container

    @staticmethod
    def get_exp_no():
        return DNN._exp_no

    def update_exp_no(self):
        DNN._exp_no += 1
```

```
    def __init__(self, name: str, nn_model, nn_optim, nn_crit, device):
        self.update_exp_no()

        self.name: str = name
        self.model = nn_model
        self.optimizer = nn_optim
        self.criterion = nn_crit
        self.device = device

        self.lr = None
        self.batch_size = None
        self.epochs = None
        self.l2_lambda = None
        self.momentum = None
        self.opt = None
        self.keepdims = None

        # training, test datas
        self.epoch_train_loss = []
        self.epoch_train_acc = []

        self.epoch_val_loss = []
        self.epoch_val_acc = []

        self.epoch_test_loss = []
        self.epoch_test_acc = []

        # hold model parameters and optimizer parameters
        self.checkpoint = None

        self.best_acc = -1
        self.best_epoch = -1

    def eval_all(self, epoch, train_loader, val_loader, test_loader, train=True, val=True,
        test=True):
        def last_conversion(data_metrics):
            tm = []
            for val in data_metrics:
                tm.append(val.item())
                if isinstance(val, list):
                    tmp = []
                    for val_inner in val:
                        tmp.append(val_inner.item())
                    tm.append(tmp.copy())
            return tm

        self.model.eval()
        if train:
            training_metrics = utils.calculate_statistics(self.model, self.criterion,
                self.device, train_loader)
            training_metrics = utils.gpu_to_cpu_tensor_to_np(training_metrics)
            training_metrics = last_conversion(training_metrics)
            if (epoch%3) == 0:
                utils.prompt_statistics("Training", epoch, self.epochs, *training_metrics)

            self.epoch_train_loss.append(training_metrics[0])
            self.epoch_train_acc.append(training_metrics[1:])

        if val:
            val_metrics = utils.calculate_statistics(self.model, self.criterion, self.device,
                val_loader)
            val_metrics = list(utils.gpu_to_cpu_tensor_to_np(val_metrics))
            val_metrics = last_conversion(val_metrics)
            if (epoch%3) == 0:
                utils.prompt_statistics("Validation", epoch, self.epochs, *val_metrics)
            self.epoch_val_loss.append(val_metrics[0])
            self.epoch_val_acc.append(val_metrics[1:])
            #print("---")

```

```python
        if test:
            test_metrics = utils.calculate_statistics(self.model, self.criterion, self.device,
                test_loader)
            test_metrics = list(utils.gpu_to_cpu_tensor_to_np(test_metrics))
            test_metrics = last_conversion(test_metrics)
            if (epoch%3) == 0:
                utils.prompt_statistics("Test", epoch, self.epochs, *test_metrics)
            self.epoch_test_loss.append(test_metrics[0])
            self.epoch_test_acc.append(test_metrics[1:])
            #print("---")

        return val_metrics[1]


    def fit(self, case:str, train_gen, val_gen, test_gen, epochs, lr, batch_size,
        early_stopping,
            momentum, l2_lambda, opt, keepdims):

        self.lr = lr
        self.batch_size = batch_size
        self.epochs = epochs
        self.momentum = momentum
        self.l2_lambda = l2_lambda
        self.opt = opt
        self.keepdims = keepdims

        #print("Initial metrics...")
        _ = self.eval_all(0, train_gen, val_gen, test_gen)

        # Train the model on the modified CIFAR-10 dataset

        #lr_scheduler = CosineAnnealingLRWithWarmup(self.optimizer, T_max=self.epochs,
            warmup_steps=5, eta_min=0.0001*self.lr)
        #lr_scheduler = ReduceLROnPlateau(self.optimizer, mode='max', cooldown=3)

        # # Set milestones for learning rate decay in terms of batches
        # milestones = [32000, 48000]
        # milestones_in_batches = []
        # train_size =  45000
        # for mile in milestones:
        #      milestones_in_batches.append(int((mile*batch_size)/train_size))
        # print(f"{milestones_in_batches=}")

        #lr_sc = CustomLrSheduler(self.optimizer, "max", beta=1)
        #lr_sc._invoke_update(-5)

        start = time.perf_counter()
        for epoch in range(1, self.epochs+1):
            self.model.train()
            for inputs, labels in train_gen:
                self.optimizer.zero_grad()

                inputs = inputs.to(self.device)
                labels = labels.to(self.device)

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

            # Evaluate the model on the training and test set
            self.model.eval()
            val_top1_acc = self.eval_all(epoch, train_gen, val_gen, test_gen)

            if val_top1_acc > (self.best_acc + 1e-5):
                self.checkpoint = utils.get_best_params_checkpoint(self.model, self.optimizer)
                self.best_acc = val_top1_acc
                self.best_epoch = epoch

```

```
            _loss = 1000

            if _loss < 1e-7:
                break

            # early stopping condition
            if (epoch - self.best_epoch) > early_stopping:
                break


            # if epoch == 70:
            #     for g in self.optimizer.param_groups:
            #         g['lr'] = g['lr'] * 0.1
            #         print(g['lr'])

            #lr_scheduler.step(self.epoch_val_acc[-1])
            #lr_sc.lr_update(val_top1_acc, True) # [Top1 Top2 Top3][0]

        end = time.perf_counter()
        elapsed_time = end - start # in seconds

        self.record(case, elapsed_time)

    def record(self, case: str, elapsed_time: float):
        global cur_run_seed

        # Finalize
        cont = DNN.get_shared_container()
        exp_no = DNN.get_exp_no()
        cont[exp_no] = dict()

        # Store meta-data types
        cont[exp_no]["seed"] = cur_run_seed
        cont[exp_no]["params"] = dict()
        cont[exp_no]["results"] = dict()
        cont[exp_no]["quick_results"] = None

        # Store Parameters
        cont[exp_no]["params"]["case"] = "case"
        cont[exp_no]["params"]["keepdims"] = self.keepdims
        cont[exp_no]["params"]["lr"] = self.lr

        cont[exp_no]["params"]["beta"] = self.momentum
        cont[exp_no]["params"]["l2"] = self.l2_lambda
        cont[exp_no]["params"]["opt"] = self.opt


        cont[exp_no]["params"]["epochs"] = self.epochs
        cont[exp_no]["params"]["best_epoch"] = self.best_epoch
        cont[exp_no]["params"]["batch_size"] = self.batch_size
        cont[exp_no]["params"]["act1"] = "-1"
        cont[exp_no]["params"]["act2"] = "-1"
        cont[exp_no]["params"]["time"] = elapsed_time


        # Store Meta-data parameters
        cont[exp_no]["results"]["losses"] = list()
        cont[exp_no]["results"]["losses"].append(self.epoch_train_loss.copy())
        cont[exp_no]["results"]["losses"].append(self.epoch_val_loss.copy())
        cont[exp_no]["results"]["losses"].append(self.epoch_test_loss.copy())
        cont[exp_no]["results"]["acc"] = list()
        cont[exp_no]["results"]["acc"].append(self.epoch_train_acc.copy())
        cont[exp_no]["results"]["acc"].append(self.epoch_val_acc.copy())
        cont[exp_no]["results"]["acc"].append(self.epoch_test_acc.copy())

        # Store Meta-data parameters
        # this contains the final train, validation, test accuracies
        _accs_test = np.array(self.epoch_test_acc.copy())
        _accs_val = np.array(self.epoch_val_acc.copy())
```

```python
        _accs_train = np.array(self.epoch_train_acc.copy())
        _idx = _accs_test[:,0].argmax(axis=0)
        cont[exp_no]["quick_results_top1"] = [_accs_train[_idx, 0], _accs_val[_idx, 0],
            _accs_test[_idx, 0]]
        cont[exp_no]["quick_results_top2"] = [_accs_train[_idx, 1], _accs_val[_idx, 1],
            _accs_test[_idx, 1]]
        cont[exp_no]["quick_results_top3"] = [_accs_train[_idx, 2], _accs_val[_idx, 2],
            _accs_test[_idx, 2]]


def print_Res(inst, model, opt, crit, train_loader, test_loader):
    #pred_confusion_matrix("Test Results", model, data.x_test, data.y_test)
    dir_name = "mini_p3_results/"
    if not os.path.exists(dir_name):
        os.mkdir(dir_name)

    dir_name += "res_" + inst.name + "/"
    os.mkdir(dir_name)
    idx = inst.name[2:]

    last_layer_weights = utils.gpu_to_cpu_tensor_to_np(list(model.parameters())[-2])
    # print(f"{len(last_layer_weights)=}\n"
    #       f"{last_layer_weights=}"
    #       f"{last_layer_weights[0].shape=}")
    pp = plots.plot_weights(np.array(last_layer_weights).reshape(10, -1), "W_last")
    pp.savefig(dir_name + "W_last")
    plots.close_p(pp)

    pp = plots.plot_loss(inst.name, inst.epoch_train_loss, inst.epoch_val_loss, "Validation")
    pp.savefig(dir_name + f"losses{idx}")
    plots.close_p(pp)

    pp = plots.plot_acc(inst.name, inst.epoch_train_acc, inst.epoch_val_acc, "Validation")
    pp.savefig(dir_name + f"accs{idx}")
    plots.close_p(pp)

    ######################################################################
    # save best test accuracy parameters
    utils.save_best_parameters(dir_name, inst.checkpoint)

    # confusion matrix for the best result
    utils.load_best_params(dir_name, model, opt)

    _, _, preds, targets = utils.prediction(model, crit, inst.device, train_loader)
    cm = utils.pred_confusion_matrix("", preds, targets)
    pp = plots.plot_cm_hm(inst.name + ",␣Training", cm)
    pp.savefig(dir_name + f"trainingConfMatrix{idx}")
    plots.close_p(pp)

    _, _, preds, targets = utils.prediction(model, crit, inst.device, test_loader)
    cm = utils.pred_confusion_matrix("", preds, targets)
    pp = plots.plot_cm_hm(inst.name + ",␣Validation", cm)
    pp.savefig(dir_name + f"validationConfMatrix{idx}")
    plots.close_p(pp)

def set_seeds(seed=42):
    """
    This code copied from internet
    # Call this function before initializing your models and training loops
    """

    # Set seed for Python random module
    random.seed(seed)

    # Set seed for NumPy
    np.random.seed(seed)

    # Set seed for PyTorch
    torch.manual_seed(seed)
```

```python

    # Set seed for CUDA operations if available
    if torch.cuda.is_available():
        #print("\nCUDA IS AVAILABLE.", end=" ")
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)

    # Set random number generator to deterministic mode for cudnn
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False  # If set to True, it may improve training speed for some configurations, but may not be reproducible
    print("Seeds are set.\n")

def transform_w_interpolate(mod: str):
    """
    mods: nearest, bilinear, bicubic, box, hamming, and lanczos

    """
    enum_meta = InterpolationMode
    d = {"bilinear":enum_meta.BILINEAR,
         "nearest":enum_meta.NEAREST,
         "bicubic":enum_meta.BICUBIC,
         "box":enum_meta.BOX,
         "hamming":enum_meta.HAMMING,
         "lanczos":enum_meta.LANCZOS
        }

    import norms
    ns = norms.d_norms[mod]

    if mod not in d:
        raise ValueError

    t = transforms.Compose([
            transforms.Resize((256, 256), interpolation=d[mod]),
            transforms.RandomCrop(size=224),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ToTensor(),
            transforms.Normalize(ns['mu'], ns['std'])
            ])
    return t

def cifar32_transform():
    import norms
    n32 = norms.d_norms_32['32']
    t = transforms.Compose([
            transforms.RandomCrop(size=32, padding=4),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ToTensor(),
            transforms.Normalize(n32['mu'], n32['std'])
            ])
    return t

def get_model():
    # Load pre-trained ResNet model
    resnet_model = resnet18(pretrained=True)

    # Modify the size of the output layer for CIFAR-10
    resnet_model.fc = nn.Linear(resnet_model.fc.in_features, 10)
    _ = resnet_model
    return resnet_model

def get_cmodel(*, dp_val):
    m = cmodel.myResnetStyleModel(keepdims=dp_val)
    m.initialize_weights()
    return m

def _gen_cond():
    batch_sizes = [32, 64, 128, 256]
```

```
    lr_values = [0.1, 0.01, 0.001, 0.0001]
    l2_reg = [0.001, 0.0001]
    dropout_vals = [0.0] #, 0.5, 0.8]
    d_optim = {"adamw": optim.AdamW}
    for opt_name_str, opt_foo in d_optim.items():
        for batch_size in batch_sizes:
            for lr in lr_values:
                for l2lambda in l2_reg:
                    for dp_rate in dropout_vals:
                        if False:
                            print(f"{batch_size=}\n"
                                  f"{momentum=}\n"
                                  f"{opt_foo=}\n"
                                  f"{lr=}\n"
                                  f"{l2lambda=}\n"
                                  f"{dp_rate=}\n")

                        yield {"BATCH_SIZE": batch_size,
                               "MOMENTUM"  : 0.9,
                               "OPT_FOO"   : (opt_name_str, opt_foo),
                               "LR"        : lr,
                               "L2LAMBDA"  : l2lambda,
                               "KEEP_DIMS" : dp_rate}


def gen_cond():
    batch_sizes = [64]
    lr_values = [0.1, 0.01]
    dropout_vals = [0.1, 0.2, 0.5, 0.7] #, 0.5, 0.8]
    d_optim = {"adamw": optim.AdamW}
    for opt_name_str, opt_foo in d_optim.items():
        for batch_size in batch_sizes:
            for lr in lr_values:
                for dp_rate in dropout_vals:
                    if False:
                        print(f"{batch_size=}\n"
                              f"{momentum=}\n"
                              f"{opt_foo=}\n"
                              f"{lr=}\n"
                              f"{l2lambda=}\n"
                              f"{dp_rate=}\n")

                    yield {"BATCH_SIZE": batch_size,
                           "MOMENTUM"  : 0.9,
                           "OPT_FOO"   : (opt_name_str, opt_foo),
                           "LR"        : lr,
                           "L2LAMBDA"  : lr*0.01,
                           "KEEP_DIMS" : dp_rate}


def resnet_gen_cond():
    batch_sizes = [128]
    lr_values = [0.1]
    dropout_vals = [0.0]
    d_optim = {"sgd": optim.SGD}
    for opt_name_str, opt_foo in d_optim.items():
        for batch_size in batch_sizes:
            for lr in lr_values:
                for dp_rate in dropout_vals:
                    if False:
                        print(f"{batch_size=}\n"
                              f"{momentum=}\n"
                              f"{opt_foo=}\n"
                              f"{lr=}\n"
                              f"{l2lambda=}\n"
                              f"{dp_rate=}\n")

                    yield {"BATCH_SIZE": batch_size,
                           "MOMENTUM"  : 0.9,
```

```python
                            "OPT_FOO"   : (opt_name_str, opt_foo),
                            "LR"        : lr,
                            "L2LAMBDA"  : 0.0001,
                            "KEEP_DIMS" : dp_rate}


def run_sims(device, train_dataset, val_dataset, test_dataset, models:dict):

    NUM_EPOCHS = 200
    EARLY_STOPPING = 50
    DEBUG = 0

    total_runs = len(list(gen_cond()))
    c = 0
    for get_inputs in tqdm(resnet_gen_cond(), total=total_runs, desc='Run No'):
        c += 1
        set_seeds()

        BATCH_SIZE = get_inputs["BATCH_SIZE"]
        MOMENTUM   = get_inputs["MOMENTUM"]
        OPT_NAME   = get_inputs["OPT_FOO"][0]
        OPT_FOO    = get_inputs["OPT_FOO"][1]
        LR         = get_inputs["LR"]
        L2LAMBDA   = get_inputs["L2LAMBDA"]
        KEEP_DIMS  = get_inputs["KEEP_DIMS"]

        train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)
        val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)
        test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)

        model = get_model().to(device)
        criterion = nn.CrossEntropyLoss().to(device)
        optimizer = None
        if OPT_NAME == "sgd":
            optimizer = OPT_FOO(model.parameters(), lr=LR, momentum=MOMENTUM, weight_decay=L2LAMBDA)
        else:
            optimizer = OPT_FOO(model.parameters(), lr=LR, weight_decay=L2LAMBDA)

        nn_instance = DNN(f"nn{c}", model, optimizer, criterion, device)
        models[c] = nn_instance

        nn_instance.fit(f"nn{c}", train_loader, val_loader, test_loader, epochs=NUM_EPOCHS, lr=LR, batch_size=BATCH_SIZE, early_stopping=EARLY_STOPPING,
                        momentum=MOMENTUM, l2_lambda=L2LAMBDA, opt=OPT_NAME, keepdims=KEEP_DIMS)
        print_Res(nn_instance, model, optimizer, criterion, train_loader, test_loader)

        with open(f'./json_outputs/data42-{c}.json','w') as file_handle_final:
            entry = {
                str(c): {
                    'prm': get_inputs,
                    'res': DNN._shared_container[c]["results"],
                    'quick_res_t1': DNN._shared_container[c]["quick_results_top1"],
                    'quick_res_t2': DNN._shared_container[c]["quick_results_top2"],
                    'quick_res_t3': DNN._shared_container[c]["quick_results_top3"]
                        }
                    }
            entry[str(c)]['prm']['OPT_FOO'] = entry[str(c)]['prm']['OPT_FOO'][0]
            json.dump(entry, file_handle_final)


def dataset_split(full_dataset, bigger_chunk_size_coef):
    full_dataset_list = list(range(len(full_dataset)))
    labels = [label for _, label in full_dataset]
    sss = StratifiedShuffleSplit(n_splits=1, test_size=bigger_chunk_size_coef, random_state=42)
```

```python
    val_idx, train_idx = next(sss.split(full_dataset_list, labels))
    # Create Subset datasets using the selected indices
    train_dataset = Subset(full_dataset, train_idx)
    val_dataset = Subset(full_dataset, val_idx)
    print(f"LENGTH {len(val_idx)=}, {len(train_idx)=}")
    return val_dataset, train_dataset

if __name__ == '__main__':
    # check and set cpu or gpu
    device_state = "cuda" if torch.cuda.is_available() else "cpu"
    device = torch.device(device_state)
    print("The device is ...", device_state.upper(), "...\n\n")


    transform = transform_w_interpolate("bilinear")
    #transform = cifar32_transform()

    # Download and Load CIFAR-10
    tv_dataset = CIFAR10(root='./data', train=True, download=True, transform=transform)
    test_dataset = CIFAR10(root='./data', train=False, download=True, transform=transform)
    val_dataset, train_dataset = dataset_split(tv_dataset, bigger_chunk_size_coef=0.9)

    models = dict()
    run_sims(device, train_dataset, val_dataset, test_dataset, models)

    d = DNN._shared_container
    # Store the data in a JSON file
    with open('data42.json', 'w') as json_file:
        json.dump(d, json_file)

    print("\nFinished.\n")
```

Code Listing 3: Python Code: Utils

```python
import numpy
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import numpy as np


#####   METRICS   #####
def calc_occurence(data_loader):
    from collections import defaultdict
    class_counts = defaultdict(int)
    for data in data_loader:
        inputs, labels = data
        for label in labels:
            class_counts[label.item()] += 1
    class_counts = dict(sorted(class_counts.items()))
    print(class_counts)

def get_top_n_correct(preds: torch.Tensor, target: torch.Tensor, top_n: int):
    _class, indices = preds.topk(top_n, 1, True, True)
    correct_preds = indices.eq(target.view(-1,1).expand_as(indices))
    total_correct = correct_preds.sum(1).sum(0)
    return _class, total_correct

def calculate_statistics(_model, _criterion, _device, data_loader, debug=0):
    total_loss, total_samples, combined_preds, combined_targets = prediction(_model,
        _criterion, _device, data_loader)

    _, top_1_acc = get_top_n_correct(combined_preds, combined_targets, top_n=1)
    _, top_2_acc = get_top_n_correct(combined_preds, combined_targets, top_n=2)
    _, top_3_acc = get_top_n_correct(combined_preds, combined_targets, top_n=3)

    if debug:
        print(f"{total_loss=}\n"
```

```
                f"{total_samples=}\n"
                f"{top_1_acc=}\n"
                f"{top_2_acc=}\n"
                f"{top_3_acc=}\n"
        )

    return total_loss/total_samples, (top_1_acc*100)/total_samples,
        (top_2_acc*100)/total_samples, (top_3_acc*100)/total_samples

def prompt_statistics(name:str, cur_epoch:int, total_epochs:int, top1_avg_loss:float,
                      top1_acc:float, top2_acc:float, top3_acc:float):
        print(f"\nEpoch␣[{cur_epoch}/{total_epochs}],␣"
                f"{name}␣\nAvg.␣(Top1)␣Loss:␣{top1_avg_loss:.4f},␣"
                f"Total␣Accuracy␣->␣"
                f"(Top1):␣{top1_acc:.4f}%,␣"
                f"(Top2):␣{top2_acc:.4f}%,␣"
                f"(Top3):␣{top3_acc:.4f}%")
        return



###   PREDICTION   ###
def prediction(_model, _criterion, _device, data_loader):
    """
    RETURNS: loss, m:(total_sample_size), preds, labels
    """
    total_loss = 0
    total_samples = 0
    batch_preds = list()
    batch_targets = list()
    _model.eval()
    with torch.no_grad():
        for inputs, labels in data_loader:
            inputs = inputs.to(_device)
            labels = labels.to(_device)
            outputs = _model(inputs)
            total_loss += _criterion(outputs, labels)
            predicted = outputs.data
            total_samples += labels.size(0)
            batch_targets.append(labels)
            batch_preds.append(predicted)
    combined_targets = torch.cat(batch_targets, dim=0).view(-1,1)
    combined_preds = torch.cat(batch_preds, dim=0)
    return total_loss, total_samples, combined_preds, combined_targets


###   SAVE/LOAD   PARAMETERS ###
def save_parameters(dir_name, _model, _optimizer):
    torch.save(_model.state_dict(), dir_name+"/wb")
    torch.save(_optimizer.state_dict(), dir_name+"/optims")

def save_best_parameters(dir_name, checkpoint: dict):
    torch.save(checkpoint['model_state_dict'], dir_name+"/wb")
    torch.save(checkpoint['optimizer_state_dict'], dir_name+"/optims")

def load_best_params(dir_name, _model, _optimizer):
    _model.load_state_dict(torch.load(dir_name+"/wb"))
    _optimizer.load_state_dict(torch.load(dir_name+"/optims"))

def get_best_params_checkpoint(_model, _optimizer):
    checkpoint = {
        'model_state_dict': _model.state_dict(),
        'optimizer_state_dict': _optimizer.state_dict()
    }
    return checkpoint


###   CONFUSION MATRIX   ###
def gpu_to_cpu_tensor_to_np(arg1):
```

```
    """given a list of tensor it returns lis of numpy arrays"""
    arr = []
    for obj in arg1:
        if isinstance(obj, torch.Tensor):
            arr.append(obj.cpu().detach().numpy())
        else:
            arr.append(obj)
    return arr

def confusion_matrix(y_pred, y_true):
    # Calculate confusion matrix
    num_class = 10
    conf_matrix = np.zeros((num_class, num_class), dtype=np.int64)

    for pred, true_val in zip(y_pred, y_true):
        conf_matrix[true_val, pred] += 1

    return conf_matrix

def pred_confusion_matrix(name: str, preds, targets, verbose=False):
    _, indices = preds.topk(1, 1, True, True)
    conf_matrix = confusion_matrix(gpu_to_cpu_tensor_to_np(indices),
        gpu_to_cpu_tensor_to_np(targets))

    if verbose:
        print(f"\nConfusion␣Matrix␣of␣{name}:")
        print(conf_matrix)
    return conf_matrix


# watch -n 1 nvidia-smi
```

Code Listing 4: Python Code: Normalization Values

```
d_norms = {
'nearest':
    {
    'mu' : [0.48872008919715881347656625, 0.47523540258407592773437500,
        0.43924531340599060058593750],
    'std': [0.24306835234165191650390625, 0.23936371505260467529296875,
        0.25593858957290649414062500]
    },
'bilinear':
    {
    'mu' : [0.48899888992309570312500000, 0.47554847598075866699218750,
        0.43956559896469116210937500],
    'std': [0.23639316856861114501953125, 0.23279730975627899169921875,
        0.24997927248477935791015625]
    },
'bicubic':
    {
    'mu' : [0.48873317241668701171875000, 0.47526192665100097656250000,
        0.43927314877510070800781250],
    'std': [0.24130378663539886474609375, 0.23760847747325897216796875,
        0.25435203313827514648437500]
    },
'box':
    {
    'mu' : [0.48872008919715881347656625, 0.47523540258407592773437500,
        0.43924531340599060058593750],
    'std': [0.24306835234165191650390625, 0.23936371505260467529296875,
        0.25593858957290649414062500]
    },
'hamming':
    {
    'mu' : [0.48876085877418518066406250, 0.47529646754264831542968750,
        0.43930974602699279785156250],
    'std': [0.23915398120880126953125000, 0.23551589250564575195312500,
        0.25244525074958801269531250]
```

```
    },
'lanczos':
    {
    'mu' : [0.4887157082557678222656250, 0.4752416908740997314453125,
        0.4392519891262054443359375],
    'std': [0.2429319024085998535156250, 0.2392167747020721435546875,
        0.2558041214942932128906250]
    }
}


d_norms_32 = {
'32':
    {
    'mu' : [0.0100285699591040611267090, 0.0098399734124541282653809,
        0.0091128842905163764953613],
    'std': [0.0772939026355743408203125, 0.0759121179580688476562500,
        0.0728098824620246887207031]
    }
}
```

Code Listing 5: Python Code: Plots

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


def plot_cm_hm(name, cm):
    _ = plt.figure(figsize=(12, 8))
    sns.heatmap(cm, annot=True, fmt="d",
                xticklabels=['c1', 'c2', 'c3', 'c4', 'c5', 'c6'],
                yticklabels=['c1', 'c2', 'c3', 'c4', 'c5', 'c6'])
    plt.title(f"{name}: Confusion Matrix")
    return _

def plot_loss(name, train_data, test_data, comparison_data_name: str):
    _ = plt.figure(figsize=(12, 8))

    x_arr = [i for i in range(len(train_data))]

    plt.plot(x_arr, train_data, label='Training')
    plt.plot(x_arr, test_data, label=f'{comparison_data_name}')

    plt.xlabel("n'th Epoch")
    plt.title(f'{name} : Average Loss over Epochs\n(Total Loss / # of Samples )')
#    plt.xticks(x_arr)

    plt.legend()

    return _

def plot_acc(name, train_data, test_data, comparison_data_name: str):
    _ = plt.figure(figsize=(12, 8))

    train_data = np.array(train_data).T
    test_data = np.array(test_data).T

    x_arr = [i for i in range(len(train_data[0]))]

    # Plotting accuracy in blue
    for i in reversed(range(len(train_data))):
        plt.plot(x_arr, train_data[i], label=f'Top {i+1}, Training')
        plt.plot(x_arr, test_data[i], label=f'Top {i+1}, {comparison_data_name}')

    plt.xlabel("n'th Epoch")
    plt.title(f'{name} : Accuracy (%) over Epochs')
    #plt.xticks(x_arr)
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
```

```python
    plt.tight_layout()

    return _

def plot_outputs(data_dict, name:str):

    _ = plt.figure()

    first_columns = [value[:, 0] for key, value in data_dict.items()]

    data_matrix = np.vstack(first_columns)

    cmap = 'magma'

    plt.imshow(data_matrix, aspect='auto', cmap=cmap)

    plt.colorbar(label='First␣Column␣Values')

    plt.xlabel('Hidden␣Layer␣(Tanh)␣Outputs')
    plt.ylabel('Time␣Step')
    plt.title(f'{name}:␣Activations␣for␣Each␣Time␣Step␣')

    return _

def plot_weights(data_matrix, name: str):
    _ = plt.figure()

    cmap = 'magma'
    plt.imshow(data_matrix, aspect='auto', cmap=cmap, extent=[0, data_matrix.shape[1], 0,
        data_matrix.shape[0]])
    cbar = plt.colorbar(label='Values')

    plt.xlabel('Neuron␣weights:␣w1,w2,␣...␣wn')
    plt.ylabel('Neurons')
    plt.title(f'{name}:␣Outputs␣Heatmap')

    return _

def close_p(f):
    plt.close(f)
```