A Comprehensive Experimental Analysis of Modern CNN Architectures on CIFAR-10

Murat Sahin, Melih Cosgun

***Abstract*—The task of image classification involves assigning a class to each desired image. The CIFAR-10 dataset is a widely used dataset for evaluating systems designed to solve this task. Experiments were conducted on the dataset using four popular architectures from the literature. The architectures were modified and various techniques were applied to observe their effects. The study employed various techniques, including modifying architecture structures, applying different regularization methods, and using various preprocessing techniques on the dataset. The experiments revealed that the effectiveness of most techniques is dependent on the architecture used. Although this was true, some techniques, such as applying data augmentation, improved performance across all configurations. The highest accuracy achieved on the test set was through the utilization of a modified VGG16 network with data augmentation and super resolution techniques, resulting in a test accuracy of 0.9123.**

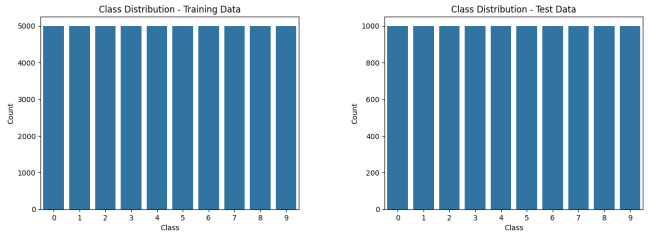
I. Introduction

THE objective of this project is to classify images in the CIFAR-10 dataset [6], a widely used dataset for image classification tasks. Various architectures have been tested on this dataset over the years. We implemented four popular architectures from the literature, namely AlexNet [7], VGG [9], ResNet [5], and GoogLeNet [10], and conducted various experiments. The implementation was carried out using the TensorFlow framework [1]. We also utilized the Weights & Biases framework [2] for logging purposes.

Following the implementation of the initial architectures, we optimized basic hyperparameters, such as learning rates and batch sizes, on the validation dataset that was kept separate for each individual training. Secondly, we investigated various architectural structures by modifying kernel sizes, adding extra layers, and experimenting with different pooling techniques. Additionally, we applied several well-known techniques from the literature, such as dropout, regularization, and batch normalization. Finally, after finalizing the architecture structures, we applied several preprocessing techniques to the data, including normalization, data augmentation, and super resolution. The effects of all experiments were observed methodically and are thoroughly discussed. The ultimate goal is to observe effects and develop a successful network gradually.

The Methods section describes the procedures used. It begins with a subsection on data analysis and is followed by a description of the architectures from the literature that were implemented. Then, it explains the modifications applied to these architectures and configurations. Finally, it describes the preprocessing techniques used for the experiments. The Results section presents the results obtained for each of the settings. The Discussion section analyzes the results and interprets their impact on performance based on the modifications.

Fig. 1. Example image-label pairs from CIFAR-10 dataset



(a) Training set (b) Test set

Fig. 2. Label distribution of the dataset

II. Methods

The methods section is divided into four subsections. The first subsection conducts a simple exploratory data analysis (EDA) on the CIFAR-10 dataset. The second subsection explores the implemented architectures and explains the hyperparameter optimizations. The third section describes the modifications made to those architectures, and the last section discusses the various preprocessing techniques applied to the dataset to further improve performance.

A. Exploratory Data Analysis

The CIFAR-10 dataset consists of 50000 training images and 10000 test images, each being RGB images with sizes 32x32. The images are classified into 10 distinct categories: frog, truck, airplane, automobile, bird, cat, deer, dog, horse, and ship. Example classes from the dataset are shown in Figure 1.

The dataset has a uniform label distribution in both the test and training datasets. Each class has 5000 training samples and 1000 test samples. Figure 2 shows the distribution of class labels. As the labels are highly balanced, there is no need to apply techniques such as random sampling.

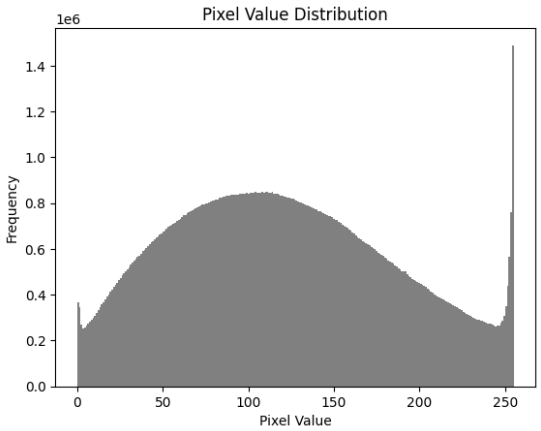


Fig. 3. Distribution of the pixel values over the dataset

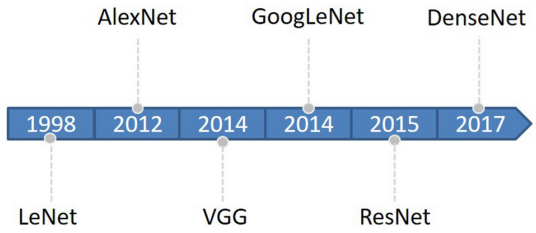


Fig. 4. Timeline of the milestone CNNs, figure from [8]

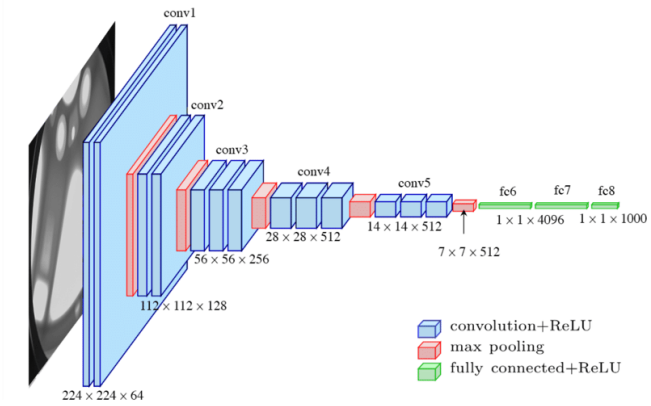


Fig. 5. Original VGG16 architecture, figure from [3]

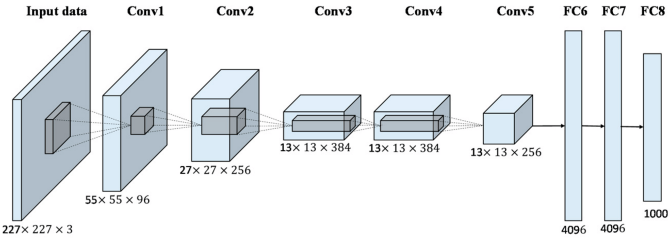


Fig. 6. Original AlexNet architecture, figure from [4]

Upon inspection of the pixel value distribution of the images in the dataset, it can be observed that the distribution closely resembles a normal distribution. This indicates that the images are not biased towards any specific color and therefore, color and intensity processing techniques are not necessary. Please refer to Figure 3 for a visual representation of the pixel value distribution.

B. Exploring Architectures

To initiate our experiments on the CIFAR-10 dataset, we chose four of the most popular and state-of-the-art (at that time) models from the literature and implemented them from scratch. These networks are considered milestones in the timeline, as shown in Figure 4. It is important to note that all of these networks were developed for the ImageNet dataset, which consists of 224x224 images and 1000 classes. Therefore, we modified those architectures beforehand to obtain networks that run properly.

1) AlexNet Architecture: AlexNet was the winning architecture in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It is widely considered to be the network that started the whole era of convolutional neural networks (with the help of ReLU). Despite its apparent simplicity today, training a network like AlexNet was previously impossible.

The architecture comprises 5 convolutional layers and 3 max-pooling layers, followed by 3 fully connected layers. The architecture employs the ReLU activation function for all layers, except for the final fully connected layer, which uses the softmax activation function. The initial two fully connected layers have a dropout ratio of 0.5. The original implementation of the architecture was not appropriate for CIFAR-10 images, so we reduced the stride of the filters to make the network suitable. A detailed representation of the architecture can be seen in Figure 6.

2) VGG16 Architecture: VGG16 has demonstrated outstanding performance in the 2014 ImageNet Large Scale Visual Recognition Challenge, demonstrating its power for image classification tasks by following the path opened by AlexNet. The architecture consists of 16 layers, hence the name VGG16.

It includes 13 convolutional layers, 5 max-pooling layers, and 3 fully connected layers, similar to the AlexNet network. The first two fully connected layers also have a dropout rate of 0.5. All convolutional layers consist of 3x3 filters, which is the biggest difference from AlexNet. All layers include a ReLU activation function except for the final fully connected layer, which has a softmax activation function. A detailed representation of the architecture can be seen in Figure 5.

3) GoogLeNet Architecture: GoogLeNet is a network that was developed in the same year as VGG. After VGG, most research groups attempted to develop deeper networks with more parameters, as this was considered key to success. However, GoogLeNet took a different approach and developed a wider network. The main emphasis of the network was its impressive performance for its efficiency, surpassing the accuracy of VGG16 with far fewer parameters.

The network is composed of multiple Inception blocks, each containing four branches with varying specifications, as shown in Figure 7. Activation maps are computed from the 1x1, 3x3, 5x5 convolution branches, as well as the pooling branch,

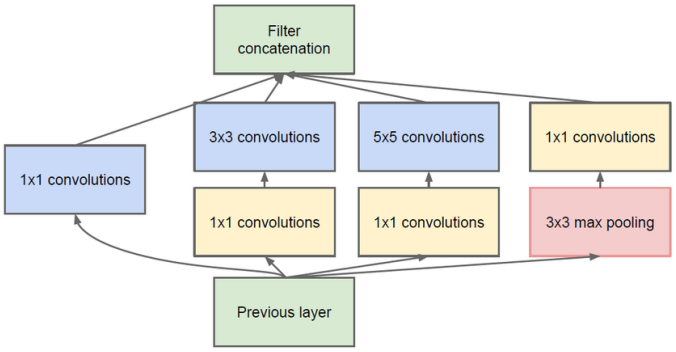


Fig. 7. Example Inception block, figure from [10]

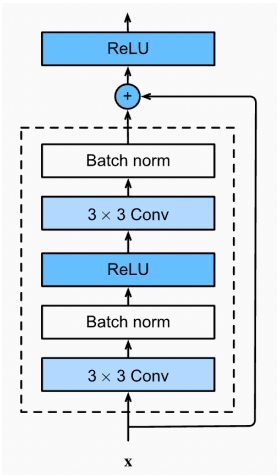


Fig. 8. Example skip connection, figure from [12]

and then concatenated for further processing. The purpose is to merge information that can be obtained from different scales. Note that we dropped some layers from the original architecture to make it suitable for the smaller images that CIFAR-10 has.

4) ResNet Architecture: Back in the 2010s, even though after the AlexNet networks were trainable, they were still suffering from vanishing gradients and this was preventing them from getting more deeper. Residual connections were introduced in ResNet to address this issue, and it outperformed other methods in ILSVC 2015. It gained significant attention due to its ability to train very deep networks. After ResNet, deeper networks became more common.

The main idea behind this network is to allow for better flowing gradients with the help of skip connections, as visualized in Figure 8. The incoming signal is processed and then also added to the outgoing one. There are multiple versions of this network, and our initial optimizations are done with the 50-layer version, namely ResNet-50.

After selecting the networks, we applied hyperparameter tuning to determine the optimal values for the learning rate and batch size. We set aside 10% of the training data for validation and ran our hyperparameter grid search for 100 epochs with an early stopping patience of 10. Several batch sizes and logarithmically spaced learning rates ranging from 1e-6 to 1e-3 were tested. More discussion of these optimizations will be found in the Results section.

C. Architectural Experiments

For architectural experiments, we made modifications and upgrades to the selected architectures from the previous section. Some modifications were applied to all architectures, while others were specific to certain architectures.

We tested various kernel sizes, batch normalization layers, and the effects of dropout and l2 regularization. We fine-tuned the lambda value for l2 regularization to find the optimal regularization strength. The section Results presents the impact of these techniques on the performance.

The following subsections explain all the modifications made to the architectures. Changes mentioned have been optimized using the grid search, not individually.

1) AlexNet Architecture: For the AlexNet architecture, the kernel size was not modified due to its varying kernel sizes for each layer, which could affect the network's purpose. The dropout technique was removed in the first two fully connected layers to observe its impact on performance and also batch normalization was applied before pooling layers. L2 regularization was applied to all layers in the network, and the lambda value was adjusted.

2) VGG16 Architecture: We changed the original 3x3 kernel size to a 5x5 kernel size for the VGG16. To experiment with the pooling techniques, we replaced the original max-pooling layers with average-pooling layers. We also removed the dropout layers and applied batch normalization layers before the pooling layers to observe the effect on performance. L2 regularization was applied to all layers, and the regularization constant was tuned to find the optimal value. Finally, three extra convolutional layers were added to the network, and performance was evaluated.

3) GoogLeNet Architecture: Please recall the various branches in the Inception modules that we have explained. We attempted to modify that module by adding a new 7x7 branch and removing the 5x5 branch. As with the others, we utilized L2 regularization with different lambda values.

4) ResNet Architecture: Our modifications to ResNet were slightly different. Its flexible structure allowed us to do distinct experiments. Not only we tested different kernel sizes (3x3, 5x5), we also tested different depths (ResNet-20, ResNet-32, ResNet-50) and number of filters (8, 16, 32). Likewise, L2 regularization is also optimized.

D. Exploring Preprocessing Techniques

After successfully obtaining architectures through the outlined experiments, we selected the best performing ones from all the networks and experimented with different preprocessing techniques. The following subsections explain the preprocessing techniques that were applied.

1) Min-Max Normalization: The Min-Max Normalization technique was used for normalizing the distribution of the pixel values. Each pixel from the test and training dataset was divided by 255, since the maximum value for the pixels is 255

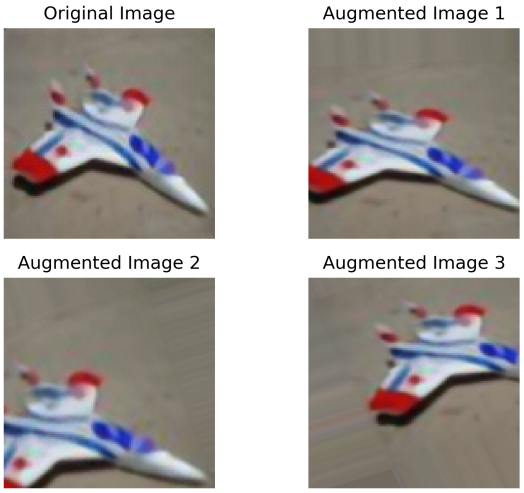


Fig. 9. Original image and several augmented images

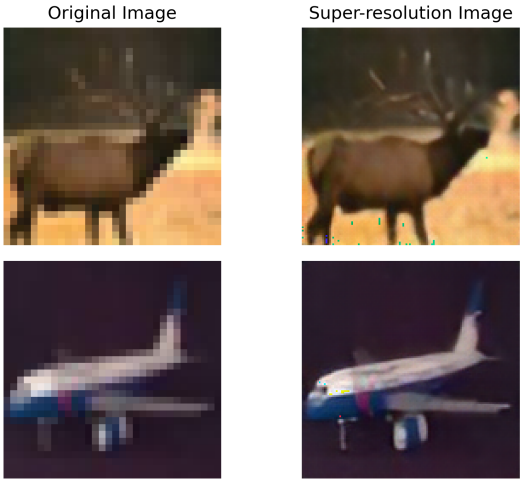


Fig. 10. Original images and their super-resolution pairs

and the minimum value is 0. The process is explained by the formulas below.

$$NormalizedPixel = \frac{X - Min(X)}{Max(X) - Min(X)}$$

$$NormalizedPixel = \frac{X - 0}{255 - 0}$$

$$NormalizedPixel = \frac{X}{255}$$

2) Data Augmentation: For data augmentation, we utilized two different configurations: a *light* configuration and a *heavy* one. The light configuration generated images rotated with 20 degrees of freedom, allowed horizontal flips, and horizontal and vertical shifts with a range of 0.1. The heavy configuration was similar, with 40 degrees of freedom and shifts with a range of 0.2. However, it also utilized shear and zoom operations with a range of 0.1. Refer to Figure 9 for examples of image augmentation results. Please note that this example is given with a super-resolution image for clarity purposes.

3) Super Resolution: We conducted experiments to increase the resolution of the images by preprocessing the dataset to obtain images scaled from 32x32 to 128x128 using ESRGAN [11]. We utilized a pre-trained model downloaded from the TensorFlow Hub for this super-resolution task. The VGG and ResNet networks were then fed with the newly obtained dataset. Please refer to Figure 10 for the example images and their super-resolution pairs.

III. Results

The results section is divided into four subsections. The first subsection describes the experiments conducted on the baseline architectures and their hyperparameter tuning processes. The second subsection presents the results of each modification applied to the networks by using final hyperparameter-tuned architectures. The third subsection describes the experiments conducted on preprocessing techniques for all architectures. Finally, the last section shows the best resulting architecture's parameters and its results on the test data.

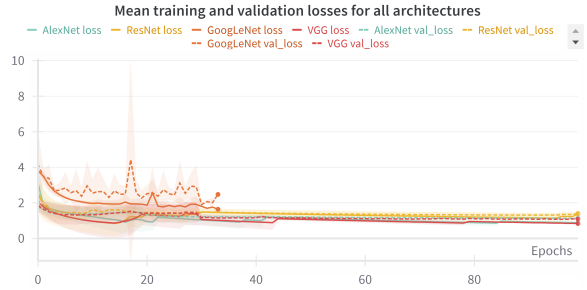


Fig. 11. The figure shows the mean training and validation loss graphs for all architectures. The lighter colored seams in the figure represent the deviation of the values, and it is noticeable that all these networks completed most of their training before the 50th epoch. Also, some training went further and they are most probably the ones with lower learning rate.

A. Hyperparameter Tuning Experiments

For all of the networks we mentioned, learning rate and batch size is optimized. We used grid search that contains batch sizes of 64 and 128, also logarithmically spaced learning rates ranging from 1e-6 to 1e-3. Exact values of the parameters can be seen below:

- Batch Sizes = {64, 128}
- Learning Rates = logspace(-6, -3, num=6)

During the tuning process, 10% of the training data was reserved for validation. We performed this grid search for 100 epochs with an early stopping patience of 10 to avoid overfitting (early stopping is also done during the whole section). Please refer to Figure 11 for the mean training and validation losses during the training process. It is worth noting that the training of some networks was terminated earlier on average.

A higher learning rate leads to more oscillating loss graphs, while a lower learning rate results in a smoother curve, as shown in Figure 12.

Increasing the batch size reduces training time but also increases the GPU time spent accessing memory, as seen in Figures 13 and 14, respectively.

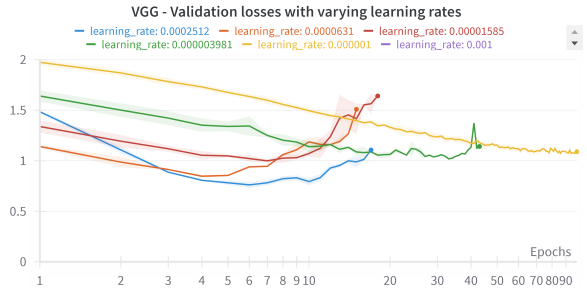


Fig. 12. The figure displays the validation loss graphs for different learning rates on the VGG network. It is important to note that these are the mean values across different batch sizes. The graph shows that a smaller learning rate results in a smoother curve but longer training time, while larger learning rates reach the minimum faster.

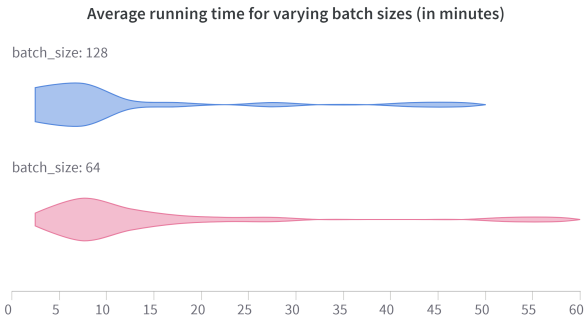


Fig. 13. This figure displays the average running times for all experiments in this subsection, using different batch sizes. It demonstrates that larger batch sizes result in shorter training times.

Overall, Figure 15 demonstrates that learning rate is more important during training.

Figures 16, 17, 18, and 19 illustrate the validation losses obtained with varying batch sizes and learning rates. Table I shows the best combinations and their corresponding validation accuracies. The subsequent subsections will use these best configurations.

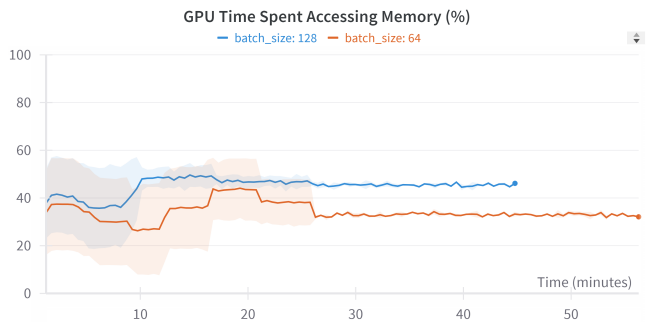


Fig. 14. This figure has the same configuration as Figure 13. However, it displays the time spent by the GPU for memory access. It demonstrates that larger batch sizes result in denser memory accesses.

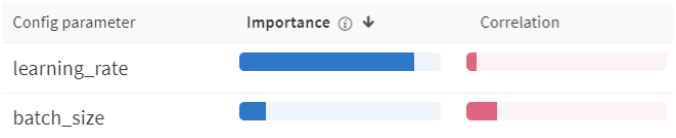


Fig. 15. The figure illustrates the significance of hyperparameters on the validation loss. While the batch size has some impact, the majority of changes are attributed to the learning rate.

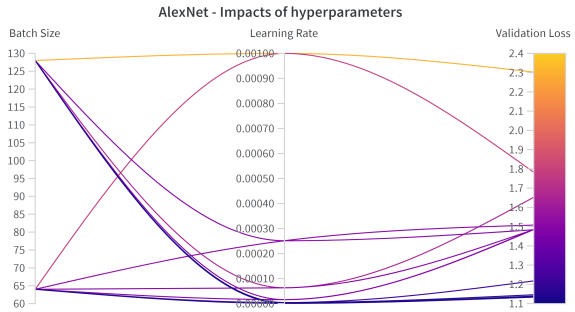


Fig. 16. This figure illustrates the effects of hyperparameters over the validation loss of AlexNet architecture. The best performing case is the one with batch size of 64 and learning rate of 6.3e-05, yielding a loss of 1.485.

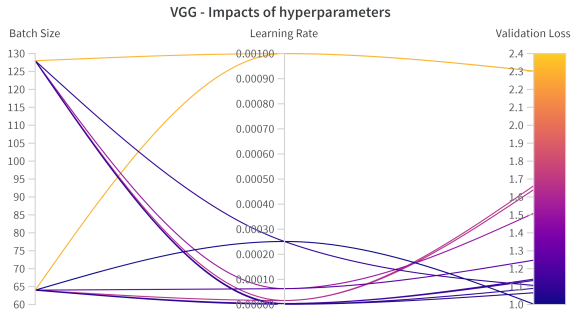


Fig. 17. This figure illustrates the effects of hyperparameters over the validation loss of VGG architecture. The best performing case is the one with batch size of 64 and learning rate of 2.5e-04, yielding a loss of 1.002.

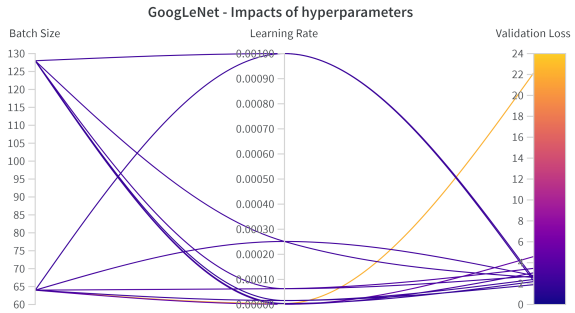


Fig. 18. This figure illustrates the effects of hyperparameters over the validation loss of GoogLeNet architecture. The best performing case is the one with batch size of 128 and learning rate of 1.6e-05, yielding a loss of 1.772.

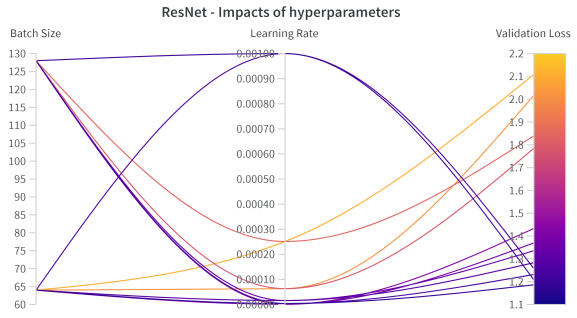


Fig. 19. This figure illustrates the effects of hyperparameters over the validation loss of ResNet architecture. The best performing case is the one with batch size of 64 and learning rate of 1e-03, yielding a loss of 1.257.

Architecture	Batch Size	Learning Rate	Validation Accuracy
AlexNet	64	6.3e-05	0.6454
VGG	64	2.5e-04	0.7894
GoogLeNet	128	1.6e-05	0.6708
ResNet	64	1e-03	0.7438

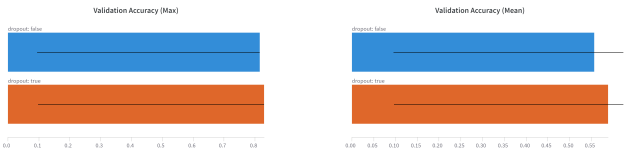
TABLE I
BEST PERFORMING HYPERPARAMETERS FOR THE ARCHITECTURES AND THEIR CORRESPONDING VALIDATION ACCURACIES

B. Architecture Modification Experiments

After obtaining the optimal learning rate and batch size results for all architectures, several experiments are conducted to observe the effects of modifications on the architectures. The experiments were divided into 8 subsections to demonstrate the impact of each modification made to the architectures. A grid search was conducted for all modifications to observe their effects on all possible settings for all architectures. The following subsections illustrate the impact of each modification by examining the mean and maximum results of all runs that contains the modification.

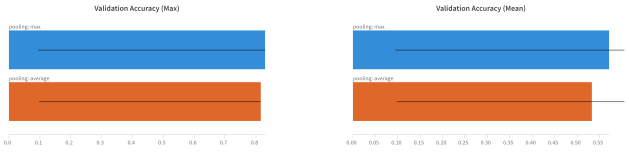
1) Dropout Modification: The impact of dropout on the VGG16 and AlexNet architectures was observed by turning off and on the dropouts at the first two fully connected layers. Figure 20 shows the results of the validation accuracy for all architecture settings.

2) Pooling Layer Modification: To test different pooling layer techniques, we changed the pooling layers of the VGG16 architecture from max pooling to average pooling. The results of the experiments for both max pooling and average pooling cases, for all architecture settings, are shown in Figure 21.



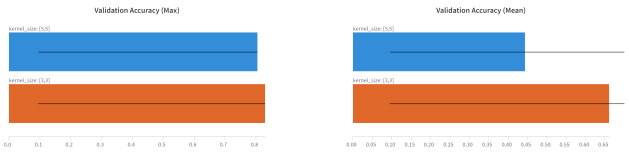
(a) Max values (b) Mean values

Fig. 20. The figures illustrate the impact of dropout modification on validation accuracy for all settings of VGG16 and AlexNet architectures. The figure on the left demonstrates the maximum accuracy values from all settings with and without dropout, while the figure on the right shows the mean accuracy of all settings with and without dropout. The validation accuracy obtained from maximum values figure was 0.8292 with dropout and 0.8148 without dropout. The validation accuracy obtained of mean values figure was 0.5882 with dropout and 0.5559 without dropout.



(a) Max values (b) Mean values

Fig. 21. The figures illustrate the impact of pooling layer modification on validation accuracy for all settings of VGG16 architecture. The figure on the left demonstrates the maximum accuracy values from all settings with max pooling and average pooling settings, while the figure on the right shows the mean accuracy of all settings with max pooling and average pooling settings. The validation accuracy obtained from maximum values figure was 0.8292 with max pooling and 0.8154 with average pooling. The validation accuracy obtained of mean values figure was 0.5716 with max pooling and 0.5328 with average pooling.

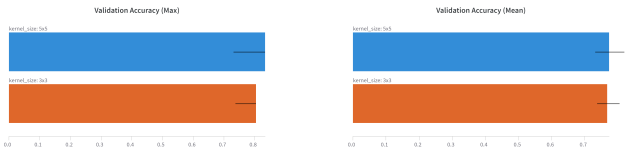


(a) Max values (b) Mean values

Fig. 22. The figures illustrate the impact of kernel size modification on validation accuracy for all settings of VGG16 architecture. The figure on the left demonstrates the maximum accuracy values from all settings with 3x3 and 5x5 kernel size settings, while the figure on the right shows the mean accuracy of all settings with 3x3 and 5x5 kernel size settings. The validation accuracy obtained from maximum values figure was 0.8292 with 3x3 kernel size and 0.8046 with 5x5 kernel size. The validation accuracy obtained of mean values figure was 0.6607 with 3x3 kernel size and 0.4437 with 5x5 kernel size.

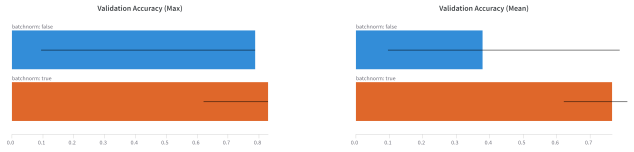
3) Kernel Size Modification: To evaluate the impact of kernel sizes on complex architectures, we conducted experiments using 3x3 and 5x5 kernels on all settings of the VGG16 and ResNet architectures. The kernel sizes of the convolutional layers have been modified from the original 3x3 to a 5x5 version. The results for the VGG16 architecture are shown in Figure 22, while the results for the ResNet architecture are shown in Figure 23.

4) Batch Normalization Modification: To investigate the impact of batch normalization, we added batch normalization layers before the pooling layers in both the VGG16 and



(a) Max values (b) Mean values

Fig. 23. The figures illustrate the impact of kernel size modification on validation accuracy for all settings of ResNet architecture. The figure on the left demonstrates the maximum accuracy values from all settings with 3x3 and 5x5 kernel size settings, while the figure on the right shows the mean accuracy of all settings with 3x3 and 5x5 kernel size settings. The validation accuracy obtained from maximum values figure was 0.8040 with 3x3 kernel size and 0.8336 with 5x5 kernel size. The validation accuracy obtained of mean values figure was 0.7674 with 3x3 kernel size and 0.7729 with 5x5 kernel size.



(a) Max values (b) Mean values

Fig. 24. The figures illustrate the impact of batch normalization layer modification on validation accuracy for all settings of VGG16 and AlexNet architectures. The figure on the left demonstrates the maximum accuracy values from all settings with and without batch normalization, while the figure on the right shows the mean accuracy of all settings with and without batch normalization. The validation accuracy obtained from maximum values figure was 0.8292 with batch normalization and 0.7874 without batch normalization. The validation accuracy obtained of mean values figure was 0.7652 with batch normalization and 0.3789 without batch normalization.

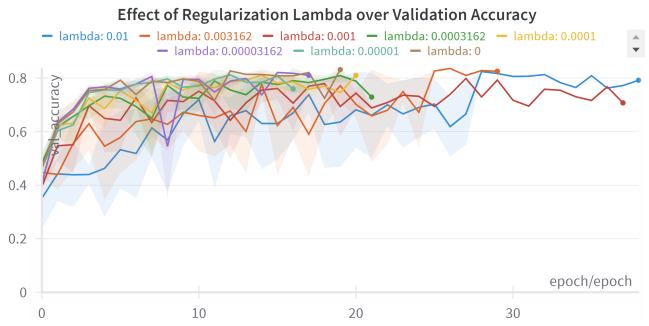


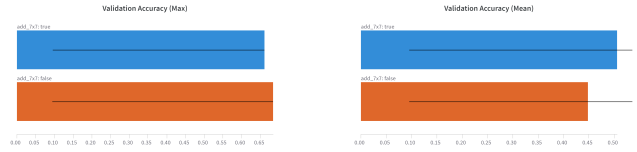
Fig. 25. The figure illustrates the effect of L2 regularization lambda over validation accuracy for all architectures. Y axis represents the validation accuracy while X axis represents the number of epochs. lambda:0 case represents the case with no L2 regularization.

AlexNet architectures. The impact of the batch normalization layer on all settings of both architectures is combined and shown in Figure 24.

5) L2 Regularization Modification: To observe the impact of L2 regularization, we applied the L2 regularization technique to all layers in the network architectures. Then, we tuned the lambda value to find the best performing results on the validation data. Figure 25 shows the impact of the lambda value for the combination of all architecture runs. The case with no regularization technique used is represented by lambda:0 value.

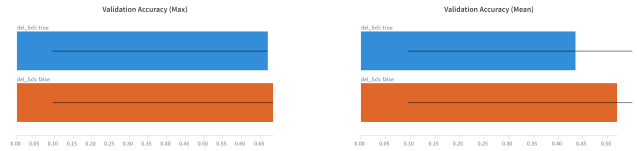
6) GoogLeNet Branch Modification: To experiment with the branch structure of the GoogLeNet architecture, we added a 7x7 filtered branch and removed the existing 5x5 filtered branch. We ran all combinations of the modifications mentioned and reported the results to assess the impact of both modifications. Figure 26 displays the effect of adding a 7x7 filtered branch for all runs of the GoogLeNet architecture, while Figure 27 illustrates the impact of removing the 5x5 filtered branch that is already present in the architecture.

7) Network Depth Modification: To test the impact of network depths, we modified the ResNet and VGG16 architectures. For the VGG16 architecture, we added three additional convolutional layers before the fully connected layers, resulting in the VGG19 architecture. We observed the impact of adding these layers together with all other modifications mentioned. The detailed results are presented in Figure 29. For



(a) Max values (b) Mean values

Fig. 26. The figures illustrate the impact of adding 7x7 filtered branch modification on validation accuracy for all settings of GoogLeNet architecture. The figure on the left demonstrates the maximum accuracy values from all settings with and without adding 7x7 branch, while the figure on the right shows the mean accuracy of all settings with and without adding 7x7 branch. The validation accuracy obtained from maximum values figure was 0.6602 with adding 7x7 branch and 0.6836 without adding 7x7 branch. The validation accuracy obtained of mean values figure was 0.5060 with adding 7x7 branch and 0.4488 without adding 7x7 branch.

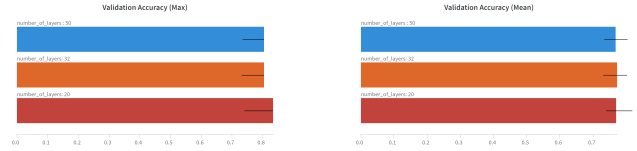


(a) Max values (b) Mean values

Fig. 27. The figures illustrate the impact of deleting the 5x5 filtered branch modification on validation accuracy for all settings of GoogLeNet architecture. The figure on the left demonstrates the maximum accuracy values from all settings with and without deleting 5x5 branch, while the figure on the right shows the mean accuracy of all settings with and without deleting 5x5 branch. The validation accuracy obtained from maximum values figure was 0.6696 with deleting 5x5 branch and 0.6836 without deleting 5x5 branch. The validation accuracy obtained of mean values figure was 0.4353 with deleting 5x5 branch and 0.5195 without deleting 5x5 branch.

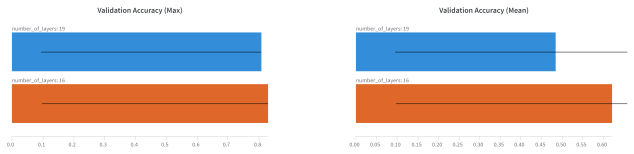
the ResNet architecture, we obtained three different models by increasing and decreasing the number of residual blocks: a 50-layered ResNet, a 32-layered ResNet, and a 20-layered ResNet. We tested all three models with the modifications mentioned and presented the results in Figure 28.

8) Number of Filters Modification: The modification of the number of filters was specific to the ResNet architecture. We altered the number of filters used in the residual blocks and tested it in three different scenarios: 32-filter structure, 16-filter structure, and 8-filter structure. We conducted these tests with all the mentioned modifications and observed the impact of changing filter sizes on the ResNet architecture. The results are shown in Figure 30.



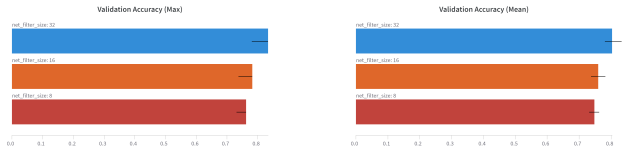
(a) Max values (b) Mean values

Fig. 28. The figures illustrate the impact of changing number of layers on validation accuracy for all settings of ResNet architecture. The figure on the left demonstrates the maximum accuracy values from all settings for number of layers, while the figure on the right shows the mean accuracy of all settings for number of layers. The validation accuracy obtained from maximum values figure was 0.8040 with 50 layer structure, 0.8034 with 32 layer structure and 0.8336 with 20 layer structure. The validation accuracy obtained of mean values figure was 0.7683 with 50 layer structure, 0.7726 with 32 layer structure and 0.7697 with 20 layer structure.



(a) Max values (b) Mean values

Fig. 29. The figures illustrate the impact of changing number of layers on validation accuracy for all settings of VGG16 architecture. The figure on the left demonstrates the maximum accuracy values from all settings for number of layers, while the figure on the right shows the mean accuracy of all settings for number of layers. The validation accuracy obtained from maximum values figure was 0.8074 with 19 layer structure and 0.8292 with 16 layer structure. The validation accuracy obtained of mean values figure was 0.4838 with 19 layer structure and 0.6206 with 16 layer structure.



(a) Max values (b) Mean values

Fig. 30. The figures illustrate the impact of changing number of filters in a residual block on validation accuracy for all settings of ResNet architecture. The figure on the left demonstrates the maximum accuracy values from all settings for number of filters, while the figure on the right shows the mean accuracy of all settings for number of filters. The validation accuracy obtained from maximum values figure was 0.8336 with 32-filter structure, 0.7818 with 16-filter structure and 0.7626 with 8-filter structure. The validation accuracy obtained of mean values figure was 0.8033 with 32-filter structure, 0.7592 with 16-filter structure and 0.7481 with 8-filter structure.

Architecture	Min-Max Normalization	Validation Accuracy	Early Stopped Epoch
AlexNet	On	0.6594	22
AlexNet	Off	0.6498	20
GoogLeNet	On	0.6862	37
GoogLeNet	Off	0.6774	38
VGG16	On	0.7856	20
VGG16	Off	0.7996	15
ResNet	On	0.8000	21
ResNet	Off	0.8146	27

TABLE II
ALL DIFFERENT MIN-MAX NORMALIZATION CASES FOR ALL ARCHITECTURES. FIRST COLUMN REPRESENTS THE ARCHITECTURE NAME, SECOND COLUMN REPRESENTS IF THE NORMALIZATION IS APPLIED, THIRD COLUMN REPRESENTS THE VALIDATION ACCURACY, LAST COLUMN REPRESENTS THE EARLY STOPPED EPOCH NUMBER

Architecture	Data Augmentation	Validation Accuracy	Early Stopped Epoch
AlexNet	heavy	0.7342	54
AlexNet	light	0.7782	60
AlexNet	none	0.6594	21
GoogLeNet	heavy	0.7122	82
GoogLeNet	light	0.7576	55
GoogLeNet	none	0.6686	29
VGG16	heavy	0.8654	46
VGG16	light	0.8798	37
VGG16	none	0.7804	14
ResNet	heavy	0.8436	49
ResNet	light	0.8520	38
ResNet	none	0.8254	27

TABLE III
ALL DIFFERENT DATA AUGMENTATION CASES FOR ALL ARCHITECTURES. FIRST COLUMN REPRESENTS THE ARCHITECTURE NAME, SECOND COLUMN REPRESENTS IF THE DATA AUGMENTATION IS APPLIED, THIRD COLUMN REPRESENTS THE VALIDATION ACCURACY, LAST COLUMN REPRESENTS THE EARLY STOPPED EPOCH NUMBER

C. Preprocessing Techniques Experiments

We applied three preprocessing techniques cumulatively and observed their impact. First, we used the Min-Max normalization technique on the dataset for all best-performing architectures from the previous section. Then, we either enabled or disabled Min-Max normalization and continued with data augmentation based on the architectures' performance. Finally, we applied the super-resolution technique to the dataset using the best performing ResNet and VGG16 architectures. Below sections explain the detailed results for each preprocessing techniques.

1) Experiments on Min-Max Normalization: The CIFAR-10 data underwent Min-Max normalization, as described in the Methods section, and was tested on all of our architectures. The best performing architecture parameters from the previous section were selected and tested with Min-Max normalization. The results are shown in Table II. As can be seen from the table, Min-Max normalization does not has a direct effect on the performance. It worked well for some of the architectures, while it did not for others.

2) Experiments on Data Augmentation: After experimenting with Min-Max normalization, we selected the best performing cases and applied data augmentation to those cases. We applied heavy, light, and no data augmentation to all four architectures and presented the results in Table III. As can be seen from the table, light data augmentation is the best augmentation technique out of our three cases for all experiments. Not applying data augmentation performed worse for all cases.

3) Experiments on Super Resolution: After completing data augmentation, we selected the best performing cases for both the VGG16 and ResNet architectures. Subsequently, we applied super resolution techniques to the CIFAR-10 dataset and compared the results with and without super resolution. Table IV displays the outcomes of our experiments. As can be seen from the table, super resolution technique performed better for VGG16 network while it performed worse for ResNet architecture.

D. Best Performing Configuration

After all of these experiments, our best performing architecture on the validation set was the modified VGG16 with the configuration given in Table V. Visualization of the network is given in Figure 33, note that input shape is 128x128 because of the superresolution operation. This configuration achieved a test loss of 0.284 and test accuracy of 0.912. See Figure 31

Architecture	Super Resolution	Validation Accuracy
VGG16	On	0.9164
VGG16	Off	0.8874
ResNet	On	0.7472
ResNet	Off	0.8478

TABLE IV
ALL DIFFERENT SUPER RESOLUTION CASES FOR RESNET AND VGG16 ARCHITECTURES. FIRST COLUMN REPRESENTS THE ARCHITECTURE NAME, SECOND COLUMN REPRESENTS IF THE SUPER RESOLUTION IS APPLIED, THIRD COLUMN REPRESENTS THE VALIDATION ACCURACY, LAST COLUMN REPRESENTS THE EARLY STOPPED EPOCH NUMBER

Architecture	VGG
Batch Size	64
Learning Rate	0.00025
Number of Layers	16
Kernel Sizes	(3, 3)
Pooling Type	Max Pooling
Batch Normalization	True
Dropout	True
L2 Lambda	0
Normalization	False
Augmentation	Light
Superresolution	True
Test Accuracy	0.9123

TABLE V
CONFIGURATION OF THE BEST PERFORMING CASE AFTER ALL OF THE EXPERIMENTS AND ITS TEST ACCURACY

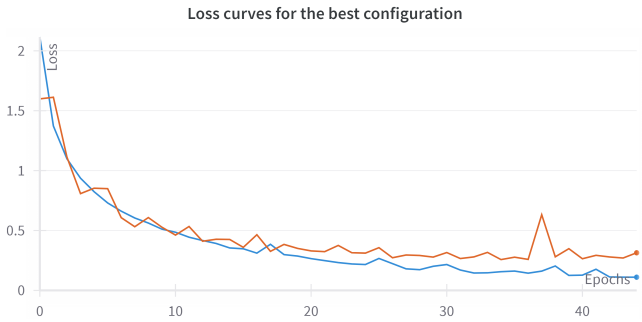


Fig. 31. This figure shows the training loss (blue) and validation loss (orange) over the epochs. It is evident that the network trained well for 34 epochs and early stopping was applied at the appropriate time (after the 44th epoch). The final training loss was 0.11 and the final validation loss was 0.31, with a validation loss of 0.25 at the time of the cut-off.

and 32 for the training loss curves and accuracy curves of this configuration.

IV. DISCUSSION

In this project, we experimented with various popular architectures. During hyperparameter tuning, we observed the effects of different batch sizes. Using a higher batch size sped up our training process (see Figure 13), but at the cost of

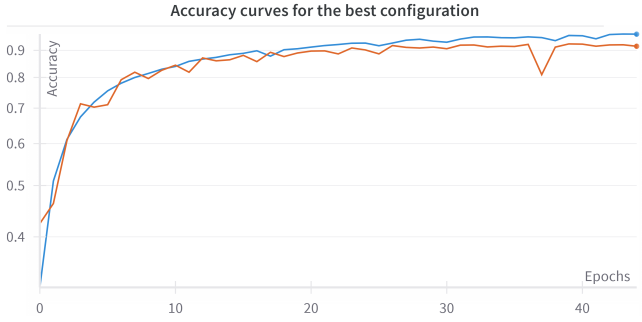


Fig. 32. This figure shows the training accuracy (blue) and validation accuracy (orange) over the epochs. Similar to the curves in Figure 31, network trained well. The final training accuracy was 0.96 and the final accuracy loss was 0.9161, with a validation accuracy of 0.9164 at the time of the cut-off.

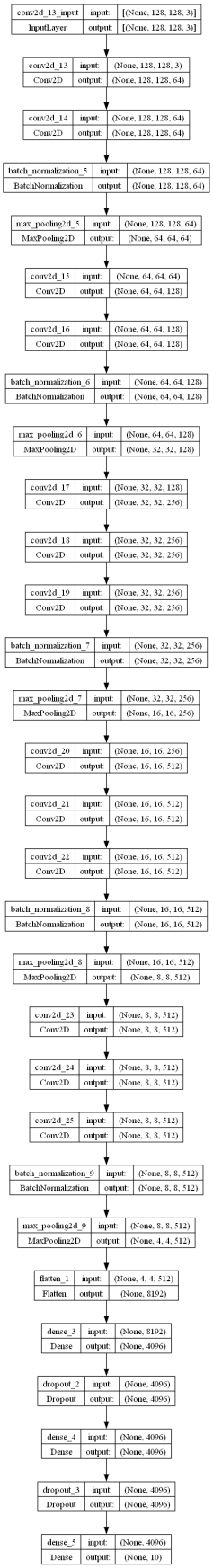


Fig. 33. Final architecture obtained, plotted using plot_model functionality of Keras.

increased memory requirements (see Figure 14). Similarly, we made critical observations about the learning rate. A smaller learning rate led to a smoother training process, but it took more time, as can be seen from Figure 12. Using larger learning rates in conjunction with early stopping and validation led to faster convergence and better results. The effect of different learning rates was observed to be the most critical hyperparameter for all architectures, refer to Figure 15.

Some of the modifications applied to the architectures improved performance, such as batch normalization and dropout, as shown in Figures 24 and 20. The mean values in the figures demonstrate the direct impact of the modifications. Changing the kernel size from 3x3 to 5x5 resulted in worse performance for the VGG16 architecture, while it improved performance for the ResNet architecture. Therefore, we decided to continue with this version of the ResNet architecture instead of original one.

Most of our modifications to the original architectures did not perform well. For instance, changing the pooling layer from max-pooling to average-pooling performed worse for the VGG16 architecture, as can be seen in Figure 21. Similarly, adding or removing branches from ResNet's residual blocks and altering the original structure resulted in worse performance, as seen in Figures 26 and 27. It can be inferred that these architectures are the product of a significant and historical effort, and that most of the parameters are optimal for image classification tasks. However, some parameters may need to be adjusted for optimal performance on the CIFAR-10 dataset. For instance, altering the depth of the ResNet architecture resulted in improved performance on this dataset. The ResNet50 architecture was successful in classifying the ImageNet dataset. However, the CIFAR-10 dataset, which contains low-resolution images, may not require such a complex architecture. By reducing the number of layers from 50 to 20, we achieved better results for this dataset, as shown in Figure 28.

Applying various preprocessing techniques was our primary strategy for this project, and it significantly improved our results. As anticipated, normalization did not yield significant improvements, as can be seen from Table II. However, data augmentation had a drastic impact on our results. It is important to note that selecting the optimal augmentation strength (light vs heavy) is crucial for achieving the best results, see Table III. Furthermore, using a super-resolution network beforehand further improved the results for one of our networks (VGG), see Table IV, leading us to the best performing configuration. This configuration is also presented in Table V.

Overall, we successfully achieved the goals set for this project. We analyzed various architectures, parameters, modifications, and techniques, all of which were sensible in their own way. In particular, the use of super-resolution is a growing domain, and our observations were promising. Ultimately, we developed a network that can make accurate predictions.

Everything we did in this project is available in our Weights & Biases project page as experiment logs, notebooks etc. Please refer to wandb.ai/takim for further exploration.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Z. Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Christopher Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *ArXiv*, abs/1603.04467, 2016.

[2] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[3] Max Ferguson, Ronay Ak, Yung-Tsun Tina Lee, and Kincho H. Law. Automatic localization of casting defects with convolutional neural networks. *2017 IEEE International Conference on Big Data (Big Data)*, pages 1726–1735, 2017.

[4] Xiaobing Han, Yanfei Zhong, Liqin Cao, and Liangpei Zhang. Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification. *Remote. Sens.*, 9:848, 2017.

[5] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

[6] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.

[8] Tingting Shi, Yongmin Liu, Xinying Zheng, Kui Hu, Hao Huang, Hanlin Liu, and Hongxu Huang. Recent advances in plant disease severity assessment using convolutional neural networks. *Scientific Reports*, 13, 2023.

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[10] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, D. Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2014.

[11] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Chen Change Loy, Yu Qiao, and Xiaoou Tang. Esrgan: Enhanced super-resolution generative adversarial networks. In *ECCV Workshops*, 2018.

[12] Aston Zhang, Zachary Chase Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *Journal of the American College of Radiology : JACR*, 2020.

APPENDIX

Appendix starts from the next page. It contains the cumulative Jupyter Notebook code for our experiments.

Notebook Description

This notebook shows the last state of our code after cumulative additions. More detailed code for each experiment can be reached from https://wandb.ai/takim/CIFAR-10_Classification

Common Data Loading Steps

In [ ]:

```
import pickle
import numpy as np
from tensorflow.keras.utils import to_categorical

def load_cifar10_batch(file_path):
    with open(file_path, 'rb') as file:
        batch = pickle.load(file, encoding='bytes')
    return batch

def load_cifar10_data(folder_path):
    train_data = []
    train_labels = []

    for i in range(1, 6):
        batch_file = f"{folder_path}/data_batch_{i}"
        batch = load_cifar10_batch(batch_file)
        train_data.append(batch[b'data'])
        train_labels.extend(batch[b'labels'])

    test_batch_file = f"{folder_path}/test_batch"
    test_batch = load_cifar10_batch(test_batch_file)
    test_data = test_batch[b'data']
    test_labels = test_batch[b'labels']

    train_data = np.vstack(train_data)
    train_labels = np.array(train_labels)
    test_labels = np.array(test_labels)

    return train_data, train_labels, test_data, test_labels

def preprocess_data(train_data, train_labels, test_data, test_labels):
    train_data = train_data.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
    test_data = test_data.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)

    train_labels_onehot = to_categorical(train_labels)
    test_labels_onehot = to_categorical(test_labels)

    return train_data, train_labels_onehot, test_data, test_labels_onehot

cifar10_folder = 'cifar-10-batches-py'

train_data, train_labels, test_data, test_labels = load_cifar10_data(cifar10_folder)

x_train, y_train, x_test, y_test = preprocess_data(
    train_data, train_labels, test_data, test_labels
)

print("Train Data Shape:", x_train.shape)
print("Train Labels Shape:", y_train.shape)
print("Test Data Shape:", x_test.shape)
print("Test Labels Shape:", y_test.shape)
```

EDA

In [ ]:

```
import matplotlib.pyplot as plt

# Define the class labels
class_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']

# Function to plot sample images
def plot_sample_images(images, labels):
    fig, axes = plt.subplots(2, 5, figsize=(12, 6))
    axes = axes.ravel()

    for i in range(10):
        axes[i].imshow(images[i])
        axes[i].set_title(class_labels[labels[i]])
        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

# Plot sample images
plot_sample_images(x_train, np.argmax(y_train, axis=1))
```

In [ ]:

```
import seaborn as sns

# Plot class distribution of training data
sns.countplot(x=np.argmax(y_train, axis=1))
plt.title('Class Distribution - Training Data')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()

# Plot class distribution of test data
sns.countplot(x=np.argmax(y_test, axis=1))
plt.title('Class Distribution - Test Data')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```

In [ ]:

```
import matplotlib.pyplot as plt

# Flatten the image data
flattened_data = x_train.flatten()

# Plot the histogram
plt.hist(flattened_data, bins=256, color='gray')
plt.title('Pixel Value Distribution')
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
plt.show()
```

Weights & Biases Related Inputs

In [ ]:

```
import wandb
from wandb.keras import WandbMetricsLogger
```

In [ ]:

```
wandb.login()
```


Super Resolution Preprocessing

Note that the usage of super resolution images is not given in this notebook, since they were not tested on some architectures. However, it is just a matter of replacing the variables.

In [ ]:

```
import tensorflow_hub as hub
from tqdm import tqdm

supres_model = hub.load('https://tfhub.dev/captain-pool/esrgan-tf2/1')
x_train_superres = supres_model(tf.cast(x_train[0:100], tf.float32), training=False).num
py().astype('uint8')
for i in tqdm(range(100, x_train.shape[0], 100)):
    result = supres_model(tf.cast(x_train[i:i+100], tf.float32), training=False).numpy()
.astype('uint8')
    x_train_superres = np.concatenate((x_train_superres, result), axis=0)

x_test_superres = supres_model(tf.cast(x_test[0:100], tf.float32), training=False).numpy
().astype('uint8')
for i in tqdm(range(100, x_test.shape[0], 100)):
    result = supres_model(tf.cast(x_test[i:i+100], tf.float32), training=False).numpy().
astype('uint8')
    x_test_superres = np.concatenate((x_test_superres, result), axis=0)

np.save('x_train_superres.npy', x_train_superres)
np.save('x_test_superres.npy', x_test_superres)
```

Model Definitions

Following blocks will have TWO merged blocks that contains the configuration and model definition for a specific architecture, run the correct configuration and model definition

AlexNet

In [ ]:

```
sweep_config = {
    'method': 'grid'
    }

metric = {
    'name': 'val_loss',
    'goal': 'minimize'
    }

sweep_config['metric'] = metric

parameters_dict = {
    'augmentation': {
          'values': ['none', 'light', 'heavy']
        }
    }

sweep_config['parameters'] = parameters_dict

parameters_dict.update({
    'earlystopping_patience': {
        'value': 10},
    'epochs': {
        'value': 100},
    'learning_rate': {
        'value': 0.000063
        },
    'batch_size': {
          'value': 64
        },
```

```
    'dropout': {
          'value': True
        },
    'batchnorm': {
          'value': True
        },
    'regularization': {
          'value': False
        },
    'normalization': {
        'value': True}
    })
```

In [ ]:

```
import tensorflow as tf

# Define the AlexNet architecture
def create_model(dropout, batchnorm, regularization):

    model = tf.keras.Sequential()

    if regularization:
        model.add(tf.keras.layers.Conv2D(filters=96, kernel_size=(11, 11), strides=(2, 2
), activation='relu', input_shape=(32, 32, 3), kernel_regularizer=tf.keras.regularizers.
l2(0.001)))
    else:
        model.add(tf.keras.layers.Conv2D(filters=96, kernel_size=(11, 11), strides=(2, 2
), activation='relu', input_shape=(32, 32, 3)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))


    if regularization:
        model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(5, 5), strides=(1, 1)
, activation='relu', padding="same", kernel_regularizer=tf.keras.regularizers.l2(0.001))
)
    else:
        model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(5, 5), strides=(1, 1)
, activation='relu', padding="same"))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))


    if regularization:
        model.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same", kernel_regularizer=tf.keras.regularizers.l2(0.001))
)
        model.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same", kernel_regularizer=tf.keras.regularizers.l2(0.001))
)
        model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same", kernel_regularizer=tf.keras.regularizers.l2(0.001))
)
    else:
        model.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same"))
        model.add(tf.keras.layers.Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same"))
        model.add(tf.keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(1, 1)
, activation='relu', padding="same"))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(3, 3), strides=(1, 1)))


    model.add(tf.keras.layers.Flatten())
```

```
    model.add(tf.keras.layers.Dense(units=4096, activation='relu'))
    if dropout:
        model.add(tf.keras.layers.Dropout(0.5))


    model.add(tf.keras.layers.Dense(units=4096, activation='relu'))
    if dropout:
        model.add(tf.keras.layers.Dropout(0.5))


    model.add(tf.keras.layers.Dense(10, activation='softmax'))

    return model
```

VGG

In [ ]:

```
sweep_config = {
    'method': 'grid'
    }

metric = {
    'name': 'val_loss',
    'goal': 'minimize'
    }

sweep_config['metric'] = metric

parameters_dict = {
    'augmentation': {
          'values': ['none', 'light', 'heavy']
        }
    }

sweep_config['parameters'] = parameters_dict

parameters_dict.update({
    'earlystopping_patience': {
        'value': 10},
    'epochs': {
        'value': 100},
    'learning_rate': {
        'value': 0.00025118864
        },
    'batch_size': {
          'value': 64
        },
    'kernel_size': {
        'value': (3, 3)
        },
    'dropout': {
          'value': True
        },
    'pooling': {
          'value': 'max'
        },
    'batchnorm': {
          'value': True
        },
    'a_layers': {
          'value': 16
        },
    'reg_alpha': {
        'value': 0
        },
    'normalization': {
        'value': False}
    })
```

In [ ]:

```
import tensorflow as tf

def create_model(kernel_size, dropout, pooling, batchnorm, n_layers, reg_alpha):
    model = tf.keras.Sequential()

    model.add(tf.keras.layers.Conv2D(64, kernel_size, activation='relu', padding='same',
input_shape=(32, 32, 3), kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(64, kernel_size, activation='relu', padding='same',
kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    if pooling == 'max':
        model.add(tf.keras.layers.MaxPooling2D((2, 2)))
    else:
        model.add(tf.keras.layers.AveragePooling2D((2, 2)))

    model.add(tf.keras.layers.Conv2D(128, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(128, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    if pooling == 'max':
        model.add(tf.keras.layers.MaxPooling2D((2, 2)))
    else:
        model.add(tf.keras.layers.AveragePooling2D((2, 2)))

    model.add(tf.keras.layers.Conv2D(256, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(256, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(256, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    if pooling == 'max':
        model.add(tf.keras.layers.MaxPooling2D((2, 2)))
    else:
        model.add(tf.keras.layers.AveragePooling2D((2, 2)))

    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    if pooling == 'max':
        model.add(tf.keras.layers.MaxPooling2D((2, 2)))
    else:
        model.add(tf.keras.layers.AveragePooling2D((2, 2)))

    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='same'
, kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
    if batchnorm:
        model.add(tf.keras.layers.BatchNormalization())
    if pooling == 'max':
        model.add(tf.keras.layers.MaxPooling2D((2, 2)))
    else:
        model.add(tf.keras.layers.AveragePooling2D((2, 2)))

    if n_layers == 19:
        model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='s
ame', kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
        model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='s
```

```
ame', kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
        model.add(tf.keras.layers.Conv2D(512, kernel_size, activation='relu', padding='s
ame', kernel_regularizer=tf.keras.regularizers.l2(reg_alpha)))
        if batchnorm:
            model.add(tf.keras.layers.BatchNormalization())


    model.add(tf.keras.layers.Flatten())

    model.add(tf.keras.layers.Dense(units=4096, activation='relu', kernel_regularizer=tf
.keras.regularizers.l2(reg_alpha)))
    if dropout:
        model.add(tf.keras.layers.Dropout(0.5))

    model.add(tf.keras.layers.Dense(units=4096, activation='relu', kernel_regularizer=tf
.keras.regularizers.l2(reg_alpha)))
    if dropout:
        model.add(tf.keras.layers.Dropout(0.5))

    model.add(tf.keras.layers.Dense(10, activation='softmax'))

    return model
```

GoogLeNet

In [ ]:

```
sweep_config = {
    'method': 'grid'
    }

metric = {
    'name': 'val_loss',
    'goal': 'minimize'
    }

sweep_config['metric'] = metric

parameters_dict = {
    'augmentation': {
          'values': ['none', 'light', 'heavy']
        }
    }

sweep_config['parameters'] = parameters_dict

parameters_dict.update({
    'earlystopping_patience': {
        'value': 10},
    'epochs': {
        'value': 1},
    'learning_rate': {
        'value': 0.000016},
    'batch_size': {
        'value': 128},
    'reg_alpha': {
          'value': 0.00001
        },
    'del_5x5': {
          'value': False
        },
    'add_7x7': {
          'value': False
        },
    'normalization': {
          'value': True}
    })
```

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Lambda, Add, Input,
GlobalAveragePooling2D, Flatten, Dense, Softmax, MaxPooling2D, AveragePooling2D, Dropout
from tensorflow.keras import Model
from tensorflow.keras.layers import concatenate
from tensorflow.keras.regularizers import l2


def InceptionBlock(x, f1, f3_reduce, f3, f5_reduce, f5, pool_reduce, reg_alpha, del_5x5,
add_7x7):

    f7_reduce = f5_reduce
    f7 = f5

    p1_x = Conv2D(filters=f1, kernel_size=(1, 1), strides=(1, 1), padding='same', kernel
_regularizer=l2(reg_alpha))(x)
    p1_x = ReLU()(p1_x)

    p3_x = Conv2D(filters=f3_reduce, kernel_size=(1, 1), strides=(1, 1), padding='same',
kernel_regularizer=l2(reg_alpha))(x)
    p3_x = ReLU()(p3_x)
    p3_x = Conv2D(filters=f3, kernel_size=(3, 3), strides=(1, 1), padding='same', kernel
_regularizer=l2(reg_alpha))(p3_x)
    p3_x = ReLU()(p3_x)

    if not del_5x5:
        p5_x = Conv2D(filters=f5_reduce, kernel_size=(1, 1), strides=(1, 1), padding='sa
me', kernel_regularizer=l2(reg_alpha))(x)
        p5_x = ReLU()(p5_x)
        p5_x = Conv2D(filters=f5, kernel_size=(5, 5), strides=(1, 1), padding='same', ke
rnel_regularizer=l2(reg_alpha))(p5_x)
        p5_x = ReLU()(p5_x)

    if add_7x7:
        p7_x = Conv2D(filters=f7_reduce, kernel_size=(1, 1), strides=(1, 1), padding='sa
me', kernel_regularizer=l2(reg_alpha))(x)
        p7_x = ReLU()(p7_x)
        p7_x = Conv2D(filters=f7, kernel_size=(7, 7), strides=(1, 1), padding='same', ke
rnel_regularizer=l2(reg_alpha))(p7_x)
        p7_x = ReLU()(p7_x)

    pool_x = MaxPooling2D(pool_size=(3, 3), strides=(1, 1), padding='same')(x)
    pool_x = Conv2D(filters=pool_reduce, kernel_size=(1, 1), strides=(1, 1), padding='sa
me', kernel_regularizer=l2(reg_alpha))(pool_x)
    pool_x = ReLU()(pool_x)

    if del_5x5 and add_7x7:
        x = concatenate(inputs=[p1_x, p3_x, p7_x, pool_x], axis=-1)
    elif del_5x5 and not add_7x7:
        x = concatenate(inputs=[p1_x, p3_x, pool_x], axis=-1)
    elif not del_5x5 and add_7x7:
        x = concatenate(inputs=[p1_x, p3_x, p5_x, p7_x, pool_x], axis=-1)
    elif not del_5x5 and not add_7x7:
        x = concatenate(inputs=[p1_x, p3_x, p5_x, pool_x], axis=-1)

    return x


# Define the GoogLeNet architecture
def create_model(config):

    reg_alpha = config['reg_alpha']
    del_5x5 = config['del_5x5']
    add_7x7 = config['add_7x7']

    inputs = Input(shape=(32, 32, 3))

    x = Conv2D(filters=64, kernel_size=(7, 7), strides=(2, 2), padding='same', kernel_re
gularizer=l2(reg_alpha))(inputs)
    x = ReLU()(x)
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)
```

```
    x = Conv2D(filters=64, kernel_size=(1, 1), strides=(1, 1), padding='same', kernel_re
gularizer=l2(reg_alpha))(x)
    x = ReLU()(x)
    x = Conv2D(filters=192, kernel_size=(3, 3), strides=(1, 1), padding='same', kernel_r
egularizer=l2(reg_alpha))(x)
    x = ReLU()(x)
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

    x = InceptionBlock(x, 64, 96, 128, 16, 32, 32, reg_alpha, del_5x5, add_7x7)
    x = InceptionBlock(x, 128, 128, 192, 32, 96, 64, reg_alpha, del_5x5, add_7x7)
    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

    x = InceptionBlock(x, 192, 96, 208, 16, 48, 64, reg_alpha, del_5x5, add_7x7)

    # Auxilary loss-output
    x_aux1 = AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same')(x)
    x_aux1 = Conv2D(filters=128, kernel_size=(1, 1), strides=(1, 1), padding='same', ker
nel_regularizer=l2(reg_alpha))(x_aux1)
    x_aux1 = ReLU()(x_aux1)
    x_aux1 = Flatten()(x_aux1)
    x_aux1 = Dense(1024, kernel_regularizer=l2(reg_alpha))(x_aux1)
    x_aux1 = ReLU()(x_aux1)
    x_aux1 = Dropout(0.7)(x_aux1)
    x_aux1 = Dense(10)(x_aux1)
    x_aux1 = Softmax(name='aux1_out')(x_aux1)

    x = InceptionBlock(x, 160, 112, 224, 24, 64, 64, reg_alpha, del_5x5, add_7x7)
    x = InceptionBlock(x, 128, 128, 256, 24, 64, 64, reg_alpha, del_5x5, add_7x7)
    x = InceptionBlock(x, 112, 144, 288, 32, 64, 64, reg_alpha, del_5x5, add_7x7)

    # Auxilary loss-output
    x_aux2 = AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same')(x)
    x_aux2 = Conv2D(filters=128, kernel_size=(1, 1), strides=(1, 1), padding='same', ker
nel_regularizer=l2(reg_alpha))(x_aux2)
    x_aux2 = ReLU()(x_aux2)
    x_aux2 = Flatten()(x_aux2)
    x_aux2 = Dense(1024, kernel_regularizer=l2(reg_alpha))(x_aux2)
    x_aux2 = ReLU()(x_aux2)
    x_aux2 = Dropout(0.7)(x_aux2)
    x_aux2 = Dense(10)(x_aux2)
    x_aux2 = Softmax(name='aux2_out')(x_aux2)

    x = InceptionBlock(x, 256, 160, 320, 32, 128, 128, reg_alpha, del_5x5, add_7x7)

    x = MaxPooling2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

    x = InceptionBlock(x, 256, 160, 320, 32, 128, 128, reg_alpha, del_5x5, add_7x7)
    x = InceptionBlock(x, 384, 192, 384, 48, 128, 128, reg_alpha, del_5x5, add_7x7)

    x = GlobalAveragePooling2D()(x)
    x = Dropout(0.4)(x)
    x = Dense(10)(x)
    x = Softmax(name='og_out')(x)

    outputs = [x_aux1, x_aux2, x]

    model = Model(inputs, outputs, name='GoogLeNet')

    return model
```

ResNet

In [ ]:

```
sweep_config = {
    'method': 'grid'
    }

metric = {
    'name': 'val_loss',
```

```
    'goal': 'minimize'
    }

sweep_config['metric'] = metric

parameters_dict = {
    'augmentation': {
          'values': ['none', 'light', 'heavy']
        }
    }

sweep_config['parameters'] = parameters_dict

parameters_dict.update({
    'earlystopping_patience': {
        'value': 10},
    'epochs': {
        'value': 100},
    'learning_rate': {
        'value': 0.001},
    'batch_size': {
        'value': 64},
    'kernel_size': {
          'value': '5x5'},
    'net_filter_size': {
          'value': 32},
    'net_n': {
          'value': 3},
    'reg_alpha': {
          'value': 0.0001},
    'normalization': {
          'value': False}
    })
```

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, BatchNormalization, ReLU, Lambda, Add, Input,
GlobalAveragePooling2D, Flatten, Dense, Softmax
from tensorflow.keras import Model
from tensorflow.keras.regularizers import l2


def ResidualBlock(x, filter_size, is_switch_block, kernel_size, reg_alpha):

    # note that if is_switch_block true, it means that output will not be the same as the
input
    # so while merging the residual connection, we need to adapt to it
    # this adaptation could be with a conv layer, or a simple downsampling + padding is e
nough.

    x_skip = x # save original input to the block

    if not is_switch_block:
        x = Conv2D(filter_size, kernel_size=kernel_size, strides=(1, 1), padding='same',
kernel_regularizer=l2(reg_alpha))(x)
    else:
        x = Conv2D(filter_size, kernel_size=kernel_size, strides=(2, 2), padding='same',
kernel_regularizer=l2(reg_alpha))(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(filter_size, kernel_size=kernel_size, strides=(1, 1), padding='same', ker
nel_regularizer=l2(reg_alpha))(x)
    x = BatchNormalization()(x)

    if is_switch_block: # takes every second element to half(v) spatial dimension and the
n adds padding to each side for matching filter (last) dimension
        x_skip = Lambda(lambda x: tf.pad(x[:, ::2, ::2, :], tf.constant([[0, 0,], [0, 0]
, [0, 0], [filter_size//4, filter_size//4]]), mode="CONSTANT"))(x_skip)
```

```
    x = Add()([x, x_skip])
    x = ReLU()(x)

    return x

def ResidualBlocks(x, filter_size, n, kernel_size, reg_alpha):
    for group in range(3): # a stack of 6n layers, 3×3 convolutions, feature maps of size
s {4fs, 2fs, fs}, 2n layers for each size
        for block in range(n):
            if group > 0 and block == 0: # double filter size
                filter_size *= 2
                is_switch_block = True
            else:
                is_switch_block = False

            x = ResidualBlock(x, filter_size, is_switch_block, kernel_size, reg_alpha)

    return x

# Define the ResNet architecture
def create_model(config):

    filter_size = config['net_filter_size']
    n = config['net_n']
    kernel_size = (3, 3) if config['kernel_size'] == '3x3' else (5, 5)

    reg_alpha = config['reg_alpha']

    inputs = Input(shape=(32, 32, 3))
    x = Conv2D(filter_size, kernel_size=kernel_size, strides=(1, 1), padding='same', ker
nel_regularizer=l2(reg_alpha))(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = ResidualBlocks(x, filter_size, n, kernel_size, reg_alpha)
    x = GlobalAveragePooling2D()(x)
    x = Flatten()(x)
    x = Dense(10)(x)
    outputs = Softmax()(x)

    model = Model(inputs, outputs, name=f"ResNet-{n*6+2}")
    return model
```

W&B creating sweep (for the grid search)

In [ ]:

```
import pprint

pprint.pprint(sweep_config)
```

In [ ]:

```
sweep_id = wandb.sweep(sweep_config, project="CIFAR-10_Classification")
```

Training Code

This is for VGG, others are similar too

In [ ]:

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split

def train(config = None):
    with wandb.init(config=config):
```

```
        config = wandb.config

        do_normalization = config['normalization']
        do_augmentation = config['augmentation'] != 'none'

        x_train_to_use = (x_train.astype('float32') / 255) if do_normalization else x_tr
ain
        x_test_to_use = (x_test.astype('float32') / 255) if do_normalization else x_test

        tf.keras.backend.clear_session()
        model = create_model(config["kernel_size"], config["dropout"], config["pooling"]
, config["batchnorm"], config["a_layers"], config["reg_alpha"])
        model.compile(
            optimizer = Adam(learning_rate=config["learning_rate"]),
            loss = "categorical_crossentropy",
            metrics = ["accuracy", tf.keras.metrics.TopKCategoricalAccuracy(k=3, name='t
op@3_accuracy')]
        )

        early_stopping = EarlyStopping(monitor='val_loss',
                                       patience=config["earlystopping_patience"],
                                       restore_best_weights=True)

        if not do_augmentation:
            history = model.fit(x_train_to_use, y_train,
                                epochs=config["epochs"],
                                batch_size=config["batch_size"],
                                validation_split=0.1,
                                callbacks=[
                                    WandbMetricsLogger(log_freq='epoch'),
                                    early_stopping
                                ], verbose=1
                                )
        else:
            if config['augmentation'] == 'light':
                datagen = ImageDataGenerator(
                    rotation_range=20,
                    horizontal_flip=True,
                    width_shift_range=0.1,
                    height_shift_range=0.1,
                    fill_mode='nearest'
                )
            else:
                datagen = ImageDataGenerator(
                    rotation_range=40,
                    horizontal_flip=True,
                    width_shift_range=0.2,
                    height_shift_range=0.2,
                    shear_range=0.1,
                    zoom_range=0.1,
                    fill_mode='nearest'
                )

            x_tr, x_vl, y_tr, y_vl = train_test_split(x_train_to_use, y_train, test_size
=0.1, random_state=42)

            train_datagen = datagen.flow(x_tr, y_tr, batch_size=config["batch_size"])
            history = model.fit(train_datagen,
                                epochs=config["epochs"],
                                batch_size=config["batch_size"],
                                validation_data=(x_vl, y_vl),
                                callbacks=[
                                    WandbMetricsLogger(log_freq='epoch'),
                                    early_stopping
                                ], verbose=1
                                )


        test_stats = model.evaluate(x_test_to_use, y_test)
        wandb.log({"test_loss": test_stats[0]})
        wandb.log({"test_acc": test_stats[1]})
```

```
        val_loss_history = history.history['val_loss']
        val_acc_history = history.history['val_accuracy']

        best_epoch_num = -1 if (len(val_loss_history) == 100 or len(val_loss_history) <=
10) else (len(val_loss_history) - 11)

        wandb.log({"best_val_loss": val_loss_history[best_epoch_num]})
        wandb.log({"best_val_acc": val_acc_history[best_epoch_num]})
```

Starting the grid search

In [ ]:

```
wandb.agent(sweep_id, train)
```