# Image Classification

**Alex Rasla**
UC Santa Barbara
alexrasla@ucsb.edu

## 1 Data

For this project, I used the CIFAR-10 [1] dataset which consists of 60,000 32x32 images, split into 50,000 for training and 10,000 for testing. In order to gain more diverse dataset/images and train a model more robust model, I implemented a data augmentation technique on this dataset that included random horizontal flipping and random cropping.

## 2 Libraries

In this project, I used Pytorch to create my CNN and Torchvision to download and use the CIFAR-10 dataset. Within PyTorch, I used Conv2d layers to perform convolutions, ReLU layers to ensure all values are $>=$ 0, and MaxPool2d layers to reduce the dimensions of the image and extract the most important features. I used the Torchvision library to download the CIFAR-10 dataset and normalize the images rather than manually loading and processing the data everytime into my project. Lastly, I used NumPy for basic array operations, matplotlib for plotting metrics, and sklearn to generate a confusion matrix.

## 3 Implementation Details

In order to train an image classification model using my program, first connect to the Google Colab GPU by navigating to research.colab.com and clone the repository https://github.com/alexrasla/ImageClassification.git Next, to edit the model and its hyperparameters, change the values in the config.py. Finally, execute

```
!python3 ./ImageClassification/train.py
```

file to train the image classification model. Once this model is trained, evaluation can be performed by executing:

```
!python3 ./ImageClassification/eval.py
        --model [path to model]
```

The evaluation is performed on the CIFAR-10 testset, and the program generates and saves a confusion matrix, which is used to plot and analyze the results using

```
!python3 ./ImageClassification/plot.py
        --dir [path to model directory]
```

## 4 Results

Throughout this project, I trained two different models, experimented with a variety of different hyperparameters, and tested out different regularization and data augmentation techniques to achieve the most accurate model. In order to compare the models, I used the most common evaluation techniques for image classification models: a confusion matrix. This matrix increments the model's output label (row) to the ground truth label (column) for each test image. Using this method, perfect accuracy is represented by a diagonal matrix. However, more notable metrics using this matrix are the overall accuracy, the commission of error, and the omission of error. The overall accuracy is percentage of correct predictions, the commission of error is the percentage of images that are assigned to a certain class that don't belong to it (overestimation), and the omission of error is the percentage of images that belong to one class but are classified as other classes (underestimation). A table of the overall accuracy, training time per epoch, and validation time per epoch for each model I experimented with is shown in Table 1. The training and validation loss for the best model are shown in Figure 1, the normalized confusion matrix is shown in Figure 2, and the error of commission and omission for this model are shown in Figure 3.

Contrary to my initial assumptions, when trained with the same hyperparamters and data, the base-

| | Overall Accuracy | Train | Validation |
|---|---|---|---|
| Model A | 0.8912 | 42.42 | 5.04 |
| Model B | 0.8605 | 34.98 | 2.09 |
| Model C | 0.8378 | 91.26 | 5.29 |
| Model D | 0.7895 | 41.60 | 3.05 |
| Model E | 0.7773 | 98.42 | 5.03 |
| Model F | 0.6612 | 83.38 | 5.17 |
| Model G | 0.6556 | 45.13 | 3.35 |

Table 1: Model A: Baseline, Data Augmentation, Batch Normalization; Model B: Baseline, 30 epochs, Data Augmentation; Model C: Large, 30 epochs, Data Augmentation; Model D: Baseline, 20 epochs, 3x3 Kernels; Model E: Large, 20 epochs, 3x3 Kernels; Model F: Baseline, 20 epochs, 7x7 Kernels; Model G: Baseline, 40 epochs, 3x3 Kernels. This figure shows the overall accuracy from the confusion matrix for each trained model, along with its average training and validation time per epoch.

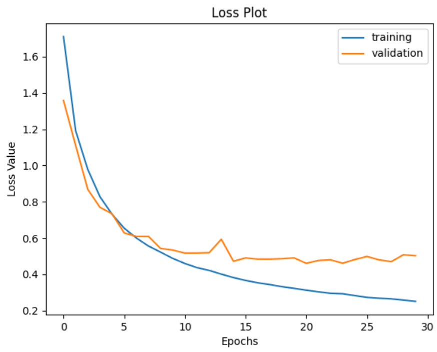


Figure 1: The training and validation loss plots for Model A: Baseline, Data Augmentation, Batch Normalization

line model with fewer layers and convolutions performed better on the overall accuracy when training on more epochs and with data augmentation. After further research, I found that this is in fact expected the expected behavior in CNNs used for image classification.

## 5 Discussion

For my experimentation in this project, I implemented two different models — one large and one small — and analyzed the effect of different kernel sizes, number of epochs, learning rate, and data augmentation. In order to perform this experimentation, I first trained the baseline model that is described in the model section. This model gave me insight into how accurate a CNN can be without any hyperparameter tuning or modifications. I trained this model for 20 epochs with a learning rate

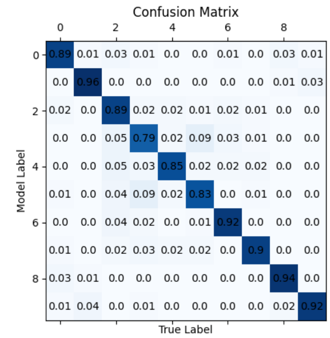


Figure 2: The confusion matrix for Model A

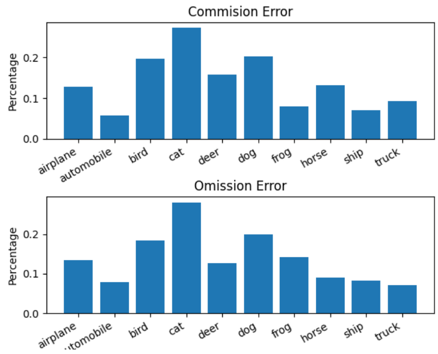


Figure 3: The commission and omision of error for Model A

of 0.001 and a kernel size of 3x3. This produced an overall accuracy of 0.7895 for the CIFAR-10 testset.

Once this baseline model generated reasonable results, I decided to test whether or not a larger model (Model E) with more layers and convolutions would increase the performance and accuracy. I trained the model with identical hyperparameters in order to be able to most effectively compare the baseline and larger model. Contrary to my hypothesis, this did not perform better than its simpler counterpart — it achieved an overall accuracy of 0.7773. Furthermore, this model took twice as long to train and was 1GB in size (compared to 67MB) because of the number of parameters.

Because I noticed a slight decrease in performance with the larger model (and a huge increase in size), I decided to build upon and experiment with the baseline model. I first experimented with a kernel size of 7x7 with a padding of 3 so every pixel can be processed. This model performed noticeably worse than the original baseline model with 3x3 kernels, decreasing the overall accuracy to 0.6612. This is likely because the images in the CIFAR-12 dataset are only 32x32. Using a 7x7 kernel for convolutions for images this small makes it hard for the model to extract important features and characteristics; the features that differentiate the images from each other and allow them to be classified correctly are smaller and more detailed. In other words, a smaller kernel size can better specify and recognize a pixel's relation to a classification rather than a bigger kernel size which has a harder time determining these details and relationships.

Next, I experimented with the number of epochs and learning rate for the baseline model to see if tuning these hyperparameters would improve performance. In order to test this, I trained the baseline model for 40 epochs with a learning rate of 0.0005. This would run the data over the entire dataset twice as much, but update the model's weights by half as much as the original model. Unfortunately, changing these parameters drastically decreased the performance again to an overall accuracy of 0.6556, but maintained a similar train and validation time per epoch. This is likely because the model overfitted the training data, and thus performed poorly on the testing and validation data. This was proven by the diverging validation loss, as well as the poor overall accuracy.

My last attempt at improving the model itself was to use regularization techniques since the dataset is relatively small and they are proven to improve many types of models for many types of applications. I decided to focus on experimenting with dropout layers and batch normalization. Throughout my experimentation with a dropout of 0.15, I noticed that even one dropout layer in the neural network prevented the model from converging at all. This was very surprising to me since I thought regularization techniques can only improve model performance, given they are not overused. I suspect it failed to improve performance because dropping convolution weights with a kernel size of 3 are already small enough that dropping additional neurons is ineffective and prevents it from learning. On the other hand, batch normalization after convolutional layers proved to increase the model's overall performance to by 0.03. This is likely because standardizing the output of the convolutional layers prevents overfitting the model to the training set.

Finally, I decided to experiment with data augmentation. In order to perform data augmentation, the CIFAR-10 images I used needed to be modified in some way to diversify the dataset. To modify the images, I used torchvision's transform module to randomly flip the images horizontally with p=0.5 and randomly crop the image. These transformations ensured the model was able to find more robust features that represented a certain image classification rather than simply learning the dataset and overfitting the data. As a result of this data augmentation, the overall accuracy of the baseline model improved by nearly 0.08, and the accuracy of the large model improved nearly 0.05. This is a significant improvement from the initial models, which truly shows the impact that data augmentation has on improving image classification.

Building upon the baseline model with the combination of batch normalization and data augmentation, I was able to improve the model's overall accuracy by 0.1, or $10\%$, to 0.8912.

## References

[1] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.