# CNN For CIFAR-10 Dataset

Tomas Pereira (40128504)
Aryamann Mehra (40127106)
Andrei Skachkou (40134189)

December 2022

## 1 Introduction

In this project, 2 challenges were set to train a Convolutional Neural Network on an extremely small sample size of the CIFAR-10 dataset, 50 samples coming from 2 of the 10 image classes, totaling 100 samples all together. In the first challenge, a CNN was trained from scratch using 3 different approaches to tackling the problem of limited data, these being: Data Augmentation through traditional methods, Augmentation through synthetic data generation, and CNN architecture modifications. In the second challenge, pre-trained models were imported and their performance was measured after using fine-tuning and various optimization schedules.

The link to the Google Colab document for this project can be found: Here

## 2 Methodology

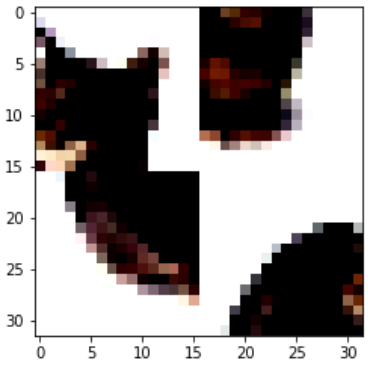
### 2.1 Challenge 1: Data Augmentation Techniques

As this project carried the heavy constraint of using only 100 total training samples, split between 2 classes, a first approach using significant data augmentation was considered to counteract the small size of the dataset. From Connor Shorten and Taghi M. Khoshgoftaar's article "A survey on Image Data Augmentation for Deep Learning"[1], a number of common image data augmentation techniques were studied. Of particular note in this article were: Image Flipping, PatchShuffle Regularization and Image Mixing, which are all claimed to have proven success in reducing the network's error rate when used on the CIFAR-10 dataset. In addition, image rotation and color channel isolation were considered for being simple to execute. The intention behind trying many augmentations techniques was to significantly increase the amount of data available for training and to find an ideal combination of these transformations that would best increase the model's performance in exchange for training time.

Each technique attempted increased the amount of the training data in an amount proportional to the original dataset, this being $N = 100$. The amount by which each technique would increase the size will be discussed now.

- Horizontal image flipping, which creates a new sample that is the flipped version of the original image, creates 1 additional sample per sample and thus makes the dataset of size $2N$.
- Image rotation allows a new sample to be created by rotating the input image by a number of degrees indicated. In theory, this could mean a new sample for each degree of rotation but instead rotations of 90, 180 and 270 degrees were tested. This resulted in 3 additional samples per sample and make the

dataset of size $4N$.
- Color isolation simply creates new samples by isolating each of the RGB channels of the image and thus created 3 new samples per image, making the dataset of size $4N$.
- PatchShuffle regularization is technique that creates a new sample by randomly displacing the pixels of an image given a certain kernel size, resulting in a new image which seperates distinct features as in the figure below. Given the CIFAR-10 images being of size 32x32, a kernel of 16x16 was selected which resulted in a small modification and led to a dataset of size $2N$.

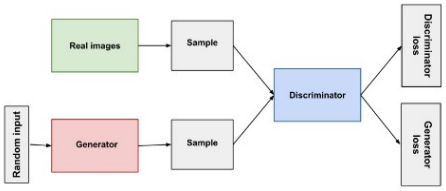



(Figure 1. Image of a cat after being passed through PatchShuffle Regularization.)

- Image mixing creates a new sample by averaging the pixel value of 2 images and assigning the label of the first sample to it. This technique was applied to every combination of samples in the original dataset and resulted in a dataset of size $N^2 + N$.

## 2.2 Challenge 1: Synthetic Data

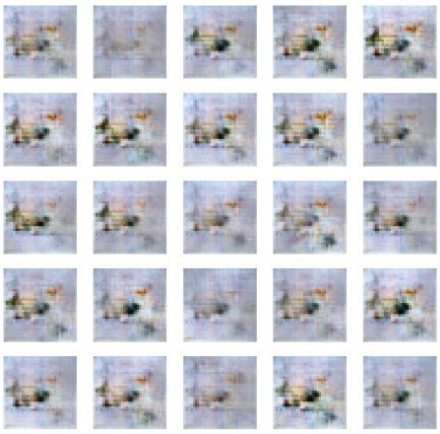
In this section, a method of data augmentation that generates synthetic images using Generative Adversarial Networks (GANs) has been considered. GANs are an architecture for training generative models, such as deep convolutional neural networks for generating images.[1]. We propose a training scheme that uses classical data augmentation to enlarge the training set and then further enlarges the data size and its diversity by applying GAN techniques for synthetic data generating. Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values. GANs achieve this level of realism by pairing a generator, which learns to produce the target output, with a discriminator, which learns to distinguish true data from the output of the generator. The generator tries to fool the discriminator, and the discriminator tries to keep from being fooled (Figure 1)[2].




(Figure 2. picture of the whole system)

Applying this model will discover how to develop a generative adversarial network with deep convolutional networks for small images of objects generating. At the end the final standalone generator model will be used to generate new images and evaluate the performance of the current method. For this method we use an open-source software Keras library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.[3] Keras provides access to the CIFAR10 dataset via the cifar10.load dataset()

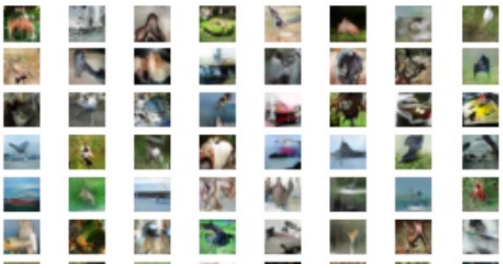
function. It returns two tuples, one with the input and output elements for the standard training dataset, and another with the input and output elements for the standard test dataset. The images of only 100 samples from the training dataset have been taken as the basis for training a Generative Adversarial Network. Specifically, the generator model will learn how to generate new plausible photographs of objects using a discriminator that will try and distinguish between real images from the CIFAR10 training dataset and new images output by the generator model. [4] Also, some best practices were used in defining the discriminator model, such as the use of LeakyReLU instead of ReLU, using Dropout, and using the Adam is an optimization version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5. that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.[5] In order to train the discriminator model, we have to repeatedly retrieve samples of real images and samples of generated images and updating the model for a fixed number of iterations. The train discriminator() function implements this, using a batch size of 128 images, where 64 are real and 64 are fake each iteration. The generator model is responsible for creating new, fake, but plausible small objects. The first step is to generate new points in the latent space. This could be achieved by calling the randn() NumPy function for generating arrays of random numbers drawn from a standard Gaussian distribution. The array of random numbers can then be reshaped into samples, that is n rows with 100 elements per row. The generateLatentPoints() function implements this and generates the desired number of points in the latent space that can be used as input to the generator model. When the discriminator is good at detecting fake samples, the generator is updated more, and when the discriminator model is relatively poor or confused when detecting fake samples, the generator model is updated less. This defines the zero-sum or adversarial relationship between these two models.[4] There may be many ways to implement this using the Keras API, but perhaps the simplest approach is to create a new model that combines the generator and discriminator models. Specifically, a new GAN model can be defined that stacks the generator and discriminator such that the generator receives as input random points in the latent space and generates samples that are fed into the discriminator model directly, classified, and the output of this larger model can be used to update the model weights of the generator [4]. Once a final generator model is selected, it can be used in a standalone manner for your application. This involves first loading the model from file, then using it to generate images. The generation of each image requires a point in the latent space as input. The complete example of loading the saved model and generating images is shown on Figure 2. In this case, we used the model saved after 200 training epochs.



(Figure 3. images generated by GAN model using 50 samples of one class(airplanes))

As it's seen from the figure above, in this case, the

results are low quality, although we can see some difference between background and foreground with a blog in the middle of each image. The reason of this is the sample size is too low. This can be concluded by comparing the poor result above with the one that was generated using all images training CIFAR10 dataset of 50k samples (Figure 3)



(Figure 4. images generated by GAN model using 50000 CIFAR10 samples)

Here after 100 epochs, we are starting to see plausible photographs with blobs that look like birds, dogs, cats, and horses. The objects are familiar and CIFAR-10-like.

### 2.3 Challenge 1: CNN Modifications

In this approach, we tried to modify the baseline CNN in order to better fit the model to the problem at hand. Since in our case the data available for training is limited, we tried modifications to the CNN that would increase generalization and reduce over fitting due to a low number of training samples. With this constraint, we researched few shot learning methods[1] and found that we can add regularization to the model layers to increase generalization ability. To this end, we attempted to implement BatchNorm in the Convolutional Layers, so as to add some sort of regularizing effect which would prevent our model from over fitting the training samples and performing worse at test time. Another way of modifying the model to better suit the problem is changing the Convolutional Layer sizes. We also tried to tweak the size of the kernels in some convolutional layers to better find patterns during training. This was achieved by increasing the kernel size and decreasing the overall size after convolutions. Since our images are already low resolution, if the kernel sizes were small, they would pick up on very minute patterns (which would be even lower resolution). For an already small input dataset, this would possibly lead to worse performance during test time. Thus, kernel sizes in the 2 middle convolutional layers was increased to 5x5. This made the model learn higher resolution patterns, which contributes to the improved accuracy of the model. A learning rate search was also performed with among the learning rates [0.1,0.01,0.008,0.001], wherein lr=0.008 yielded the highest average accuracy as calculated in the test bed.

### 2.4 Challenge 2: Pretrained Model Fine-tuning

In this method, we attempted to fine tune the fully connected layers in a pretrained ResNet18 model, in order to improve the accuracy in our CIFAR classification task (used [7] as a reference to fine tuning). Modifying kernel sizes in the first convolutional layer was attempted but was found to negatively effect performance, possibly due to this reducing the effectiveness of the pretrained kernels in those layers. A hidden layer was added to the FC layers of the pretrained ResNet model, in order to better adapt to our data while still reaping benefit from the pre-learned kernels. This was found to be more successful in adapting the model to the CIFAR dataset. We added a Sigmoid non linearity in the FC hidden layer and tried this approach with/without batchnorm and found that accuracy was significantly higher with batchnorm, so we decided to keep batch-

norm as a fine tuning measure. We tried different hidden layer widths (wide = 18 and narrow = 3) and found that the wider network performed marginally faster and more accurate so that width=18 was decided as the finetuning measure. Initially, the learning rate, and batch size was kept to 0.1,10 respectively. When we tested these optimizations with the ideal learning rate and batch size (on default pretrained nets) found in Challenge 2, the overall accuracy dropped. Thus, we went with the initial learning rates of 0.1 and 10 as the hyperparams for this optimization method. Thus, in this method, we fine tuned by adding a hidden FC layer (shape=(512,18)) over the pre-trained ResNet18, with a sigmoid non linearity and batchnorm regularization. The output layer had shape=(18,10).

### 2.5 Challenge 2: Optimization Scheduling

In this approach for the second challenge, it was decided to study how the hyper-parameters passed to the model's optimizer were affecting model performance and training time. To this end, the AlexNet pre-trained model was used with the classification layer being fine-tuned given the different variables.

To begin, the Stochastic Gradient Descent optimizer was used as the baseline with combinations of learning rates: 0.001, 0.01, 0.05 and 0.1, and batch sizes: 10, 32, 64, 128. In the end, a combination of learning rate 0.01 and batch size 128 was selected as the baseline. From here, it was decided to test an adaptive learning rate on the optimizer to improve the rate at which the loss would converge. This would be implemented through PyTorch's built-in learning rate schedulers. Both the LambdaLR and ExponentialLR scheduler's were tested along with SDG and Adagrad optimizers.

## 3 Experimental Results

### 3.1 Data Augmentation Techniques

In testing the various augmentation techniques selected, many trials were conducted with different combinations of techniques. For comparison, each of these datasets were trained using the CNN provided in the project testbed and used its performance on the original 100 samples, which was $71.69 \pm 6.71$, as a benchmark.

In the first augmentation attempt, all of the described methods, except color isolation, were put together in an attempt to maximize the amount of training data. This means that all of the new samples were created and then passed through image mixing. In all, this resulted in a datset of size $(6N)^2 + 6N$, or 360600 training samples. The training of 5 seeds using this dataset took just over 1 1/2 hours and had a very lacking performance of $59.60 \pm 0.55$ on the test set.

After this attempt, it was considered that image mixing after having used PatchShuffle regularization may have been creating images which were too chaotic to learn from. In the second trial, only non-PatchShuffled images were mixed, creating a dataset of size $(5N)^2 + 6N$, or 250600. 5 seeds of training on this dataset took 1 hour and also did not lead to any performance enhancement with accuracy of $61.17 \pm 0.93$ on the test set. In the performance of both these sets, it was found that the training did not seem to be changing very much, which explained the poor performance.

By this stage, it seemed that increasing the size of the training set significantly was only harming performance. To look into this, the next attempt cut out Image mixing entirely as an augmentation method, which led to a much smaller dataset of $6N$, or 600 training samples. Here the loss did begin converging and after a few minutes had a test time accuracy of $65.69 \pm 1.86$ over 100 training epochs

and 10 seeds. It is predicted that given more epochs, this performance would have only improved.
In the fourth attempt, all of the augmentations were used once again but this time kept independent from each other so as to not bloat the training data, resulting in a set of size of $N^2 + 6N$, or 10600. This time, training loss was once converging but start significantly lower than in the last, starting near 0.7 rather than 2.2. The final test time accuracy over 10 seeds was $64.03 \pm 1.12$.
Finally, all of the augmentations were tested individually to see whether any of them seemed to improve performance more that the rest. The results will be shown in point form:
- Image Mixing Independent Performance $N^2 + N$: $59.00 \pm 1.71$ test time accuracy.
- Horizontal Flipping Independent Performance $2N$: $66.11 \pm 2.71$ test time accuracy.
- Image Rotation Independent Performance $4N$: $65.77 \pm 1.94$ test time accuracy.
- PatchShuffle Independent Performance $2N$: $59.03 \pm 1.27$ test time accuracy.
- Color Isolation Independent Performance $4N$: $59.14 \pm 2.65$ test time accuracy.
In the end, the best results were found in using horizontal image flipping as a singular augmentation technique, however this still did not outperform the original 100 samples.

### 3.2 Synthetic Data

For the synthetic dataset 1000 training samples of generated synthetic images will be used in addition to 50 per class of initial images. As being said, we have 2 classes; Therefore, the training set will be 2100 images in total. Evaluation will be performed on 700 random labeled images picking from targeted classes. For model training, a simple Convolutional neural network with 2 hidden layers and RELU activation function has been used (Figure 4).

```
cnn = keras.Sequential([
    layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

cnn.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])
```

(Figure 5. CNN using for training)

The testing has been performed on 50 splits and using two different optimizers: • adam • SGD(lr=0.01, momentum=0.9))

Firstly, as it's seen from Figure 6, using additional synthetic data increases training accuracy rate only with lesser epochs. Only with training on 7 epochs the model has slightly higher accuracy on evaluation set (+6%). With 15, 30, 45 epochs the accuracy is even lower when using the additional synthetic data. Secondly, switching from 'Adam' optimizer to 'SGD with lr=0.01, momentum=0.9' doesn't change anything in terms of accuracies ratios. It has the same tend to be higher with initial 100 samples than with 2100 images. Also, during the training it's noticeable that with synthetic data the model shows the worse results on evaluation set. Therefore, this method of synthetic data generation does not help much to solve the issue of limited data, because the GNA model still need more data to generate a decent images as we have seen in example of Figure 3.

| | Performance (Accuracy) optimizer='adam', loss='sparse_categorical_crossentropy' | | | | |
|---|---|---|---|---|---|
| | 7 epochs | st.dev | 15 epochs | st.dev | 30 epochs |
| **Limited data (100 samples)** | 0.66 | +-0.02 | 0.79 | +-0.04 | 0.81 |
| **Extended data (100 samples + 2000 of generated data)** | 0.72 | +-0.03 | 0.74 | +-0.03 | 0.74 |
| Result (increase/decrease) | 6% | | -5% | | -7% |
| | | | | | |
| | | | | | |
| | | | | | |
| | Performance (Accuracy) optimizer=SGD(lr=0.01, momentum=0.9), loss='sparse_categorical_crossentropy' | | | | |
| | 7 epochs | st.dev | 15 epochs | st.dev | 30 epochs |
| **Limited data (100 samples)** | 0.5 | +-0.04 | 0.56 | +-0.02 | 0.52 |
| **Extended data (100 samples + 2000 of generated data)** | 0.51 | +-0.02 | 0.51 | +-0.03 | 0.68 |
| Result (increase/decrease) | 1% | | -5% | | 16% |

(Figure 6. Evaluation results )

### 3.3 CNN Modifications

The changes made in this method, to the CNN, cause slightly worse performance when running the testbed on CPU (1 min for the baseline vs approximately 1min20seconds for our modification). With all the tweaks discussed here, we achieve average accuracy around $77\% \pm 6\%$ which is a decent increase over the baseline accuracy with not a huge impact on performance. The results of the learning rate search are in Figure 7. Basis this, 0.008 was chosen as the learning rate for the final submission.

```
Learning Rate (0.1): [68.39999999999999, 10.182177306150072]

Learning Rate (0.01): [78.95714285714286, 6.743114401872082]

Learning Rate (0.008): [79.37142857142857, 6.774562264644928]

Learning Rate (0.001): [73.37142857142858, 8.331450127348166]
```

(Figure 7. Learning rate search results)

We then experimented to see if increasing the kernel size (as discussed in section 2.3) was a viable way to boost accuracy. This was done by running 2 models, one with the original kernel sizes (only modifying the baseline by adding batchnorm to the 2 middle convolutional layers) and one with modified kernel sizes (also with batchnorm applied to the same layers). The results of our testing showed a slight accuracy jump when using a slightly bigger kernel size compared to the baseline (results in Figure 8). The model with baseline kernel sizes is pictured in Figure 9.

```
Acc over 10 instances with modified Conv kernels: 78.84 +- 7.43
Acc over 10 instances with baseline kernels: 77.60 +- 8.29
```

(Figure 8. Results with vs without modifying the kernel sizes)

```
class Net2(torch.nn.Module):
    def __init__(self):
        super(Net2, self).__init__()

        # MODEL WITH BASELINE KERNEL SIZE + BATCHNORM
        self.layers = nn.ModuleList()

        self.layers+=[nn.Conv2d(3, 16,  kernel_size=3) ,
                      nn.ReLU(inplace=True)]
        self.layers+=[nn.Conv2d(16, 16,  kernel_size=3, stride=2),
                      nn.BatchNorm2d(16),
                      nn.ReLU(inplace=True)]
        self.layers+=[nn.Conv2d(16, 32,  kernel_size=3),
                      nn.BatchNorm2d(32),
                      nn.ReLU(inplace=True)]
        self.layers+=[nn.Conv2d(32, 32,  kernel_size=3, stride=2),
                      nn.ReLU(inplace=True)]
        self.fc = nn.Linear(32*5*5, 32)

        self.layerout=nn.Linear(32,10)


        # add one hidden layer with non linearity in the Fully Connected layers
    def forward(self, x):
        for i in range(len(self.layers)):
          x = self.layers[i](x)
        x = x.view(-1, 32*5*5)
        rel=torch.nn.ReLU()
        x = self.fc(x)
        x=rel(x)
        x=self.layerout(x)

        return x
```

(Figure 9. Baseline model with only batchnorm2d applied)

The model we ended up with after modifying the baseline CNN is shown in Figure 10.

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

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # added BatchNorm2d to the hidden Convolutional Layers, Reduced dimensionality
        self.layers = nn.ModuleList()

        self.layers+=[nn.Conv2d(3, 16,  kernel_size=3) ,
                      nn.ReLU(inplace=True)]
        self.layers+=[nn.Conv2d(16, 16,  kernel_size=5, stride=2),
                      nn.BatchNorm2d(16),
                      nn.ReLU()]
        self.layers+=[nn.Conv2d(16, 32,  kernel_size=5),
                      nn.BatchNorm2d(32),
                      nn.ReLU(inplace=True)]
        self.layers+=[nn.Conv2d(32, 32,  kernel_size=3, stride=2),
                      nn.ReLU(inplace=True)]
        self.fc = nn.Linear(32*4*4, 32)

        self.layerout=nn.Linear(32,10)


        # add one hidden layer with non linearity in the Fully Connected layers
    def forward(self, x):
        for i in range(len(self.layers)):
          x = self.layers[i](x)
        x = x.view(-1, 32*4*4)
        rel=torch.nn.ReLU()
        x = self.fc(x)
        x=rel(x)
        x=self.layerout(x)

        return x
```

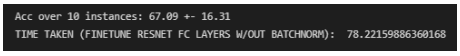
(Figure 10. Model after modifying baseline)

The final results (running the TestBed with all the changes to the CNN and learning rate) are shown in Figure 11. In conclusion, this seems to have been an effective way of boosting our model's accuracy, owing to the added generalizational ability of our model, especially considering the fact that the added performance only slows down training on the testbed by around 20 seconds (around 33%).


(Figure 11. Model after modifying baseline)

### 3.4 Pretrained Model Finetuning

The accuracy and time taken in the first attempt (only adding an FC Layer + non-linearity) is summarized in Figure 12.




(Figure 12. Fine Tuning by adding one hidden layer + non linearity)

After adding batchnorm to the modified FC Layer (before the Sigmoid non linearity), the results improved to $88.06\% \pm 4.58\%$ (See Figure 13),


(Figure 13. Adding BatchNorm1d to the fine tuned FC Layer)

One further way of improving accuracy was found to be increasing the Fully Connected Hidden Layer width. Earlier, we thought that because we have fewer samples and 10 output classes, higher layer widths would cause overfitting and be counter productive to model accuracy. However, this was found to be false - upon increasing the FC Hidden Layer width to 18, accuracy jumped by 1.5% to 2.0% on average, and the impact on performance wasn't significant either - so we kept the hidden layer width as 18 in this method. (See Figure 14 for final accuracy + execution time for this optimization method).


(Figure 14. Final Results with FC Hidden Layer Width = 18 + BatchNorm + Sigmoid non-linearity)

### 3.5 Optimization Scheduling

In doing the initial hyper-parameter search for the ideal learning rates and batch sizes, it was found that all of the combinations performed relatively equally with some performing just slightly better than the others. In general batch sizes of 128 tended to have a higher accuracy variance than the others and learning rate of 0.001 tended to have the highest average accuracy of all the choices.

With the initial hyper-parameters of LR=0.01 and BatchSize=128 selected, the learning rate scheduler LambdaLR was used and trained for 20 epochs. This achieved a performance of $90.73 \pm 6.42$, which was a similar average accuracy as the baseline but with a slightly higher variance. The next trial used the same method but instead started the learning rate off at 0.001, which had the highest average in the baseline testing, and as predicted resulted in a boost in average performance as well as reduction in variance with a performance of $92.23 \pm 5.78$ over 10 seeds. Seeing that reducing the initial learning rate had a positive effect of performance, the next trial used 0.0001 as the starting point and trained over 40 epochs to give the loss enough time to converge.

The performance of this trial over 10 seeds was $92.21 \pm 5.42$, very similar to the previous, and so the results were not worth the extra training time that this required.

Finally, the ExponentialLR scheduler was used using an initial learning rate of 0.001 over 20 epochs given that this was the best performer as of yet. The scheduler was instructed to scale the learning rate by a factor of 0.95 per epoch to slowly reduce it over the training period. The performance of this trial was an average accuracy of $91.09 \pm 5.87$, which was an improvement over the initial trials but was not enough to beat out the LambdaLR scheduler with initial LR of 0.001.

# 4 Conclusions

In the use of data augmentation techniques to enhance the size of the dataset, it was surprising to find that the original 100 samples performed better than any set created using augmentations. Working under the assumption that a greater number of training samples would aid in the models generalization capabilities, many techniques were used but none resulted in performance increases.These augmentations varied in difficulty of implementation and using each incurred an increase in training time, which did prove worthwhile.

The method of synthetic data generation does not provide a good solution to overcome the problem of limited data. The accuracy results of using synthetic data in addition to initial dataset are lower during the evaluation process, for the reason that the model generates poor quality syntithic data. In order to fix that issue the generator needs more images for particular class in order to produce the proper synthetic data.

Modifying the CNN Architecture seemed to be an effective way to boost accuracy for this challenge. By modifying the CNN to generalize better, we improve the accuracy compared to the baseline, and the performance impact is not as heavy as the Data Augmentation approach. In this approach, we did realize that due to the constraint of number of training samples, adding more parameters didn't necessarily improve accuracy or performance, rather, by reducing the number of parameters, we are able to use these reduced amount of parameters to learn more meaningful patterns which leads to stable performance and higher accuracy.

Fine tuning the CNN for Challenge 2 didn't lead to a huge improvement over just modifying the pretrained network to work with data of the given problem's dimensions. That is probably because the pre trained network has already learned most of the meaningful patterns it can with the given number and resolution of input samples. Adding batchnorm and increasing the hidden fc layer width did lead to slight gains in performance and as a result did improve our final result slightly, without adding too much performance overhead. In the second challenge, it was found that using a LambdaLR optimization scheduler alongside and initial learning rate of 0.001 lead to the greatest improvements in the model's accuracy after fine-tuning.

# 5 References

1. Shorten, C., Khoshgoftaar, T.M. A survey on Image Data Augmentation for Deep Learning. J Big Data 6, 60 (2019). https://doi.org/10.1186/s40537-019-0197-0

2. E. E. (LEDU), "Understanding few-shot learning in machine learning," Medium, 08-Jun-2021. [Online]. Available: https://medium.com/quick-code/understanding-few-shot-learning-in-machine-learning-bede251a0f67. [Accessed: 17-Dec-2022].

3. Archit Yadav, "Generating/Expanding your datasets with synthetic data"

Aug 10, 2021. [Online]. Available: https://towardsdatascience.com/generating-expanding-your-datasets-with-synthetic-data-4e27716be218.

4. Google Developers blog, "Machine Learning Advanced courses" Last updated 2022-07-18 UTC. [Online]. Available: https://developers.google.com/machine-learning/gan/discriminator.

5. Jason Brownlee, "How to Develop a GAN to Generate CIFAR10 Small Color Photographs", July 1, 2019. [Online]. Available: https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/.

6. Jason Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning", July 3, 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

7."Finetuning torchvision models," Finetuning Torchvision Models - PyTorch Tutorials 1.2.0 documentation. [Online]. Available: https://pytorch.org/tutorials/beginner/finetuning-torchvision-models-tutorial.html [Accessed: 18-Dec-2022] (change the '-' to '_' to in the link to access the documentation).