CNN For CIFAR10 Dataset Challenge

Network Architecture

We were asked to build a CNN classifier for the CIFAR10 dataset. In my implementation, the network architecture is as follows:

- Input layer – 3x32x32 nodes (images of size 32x32 with 3 channels)
- Conv. Layer –
    - 16 filters, 3x3 kernel size, padding = 1
    - ReLU activation
    - Max Pooling (2x2)
- Conv. Layer –
    - 32 filters, 3x3 kernel size, padding = 1
    - ReLU activation
    - Max Pooling (2x2)
- Conv. Layer –
    - 48 filters, 3x3 kernel size, padding = 1
    - ReLU activation
- Fully Connected Layer – Size 3072 (8x8 sized filter maps (after two /2 in size due to Max Pooling), times 48 kernels from previous conv. layer)
- Dropout Layer – with probability = 0.5
- Output layer – 10 nodes (one node for each class), classification using Softmax

Let us display the number of trainable parameters in the model above. Using the function:

```
model = myCNN().to(device)
print('Number of parameters: ', sum(param.numel() for param in model.parameters()))
```

We get the following number of parameters (**< 50,000**, as asked):

```
Number of parameters:  49882
```

The Training Procedure

A (short) summary of the process: First, download the CIFAR10 dataset, as well as transform and augment the data (see data augmentation below). Second, start the training loop while: calculate the loss (Cross Entropy Loss) for each batch (batch size = 128, images are shuffled), back-propagate, and perform a gradient descent (Adam) step (with learning rate = 0.001). Do that for 100 epochs (But break when the error on the full test set is less than 0.2).

*To conclude the hyper-parameters: batch_size=128, learning_rate=0.001, epochs=100.

*For further information regarding the implementation, please refer to the code (documented).

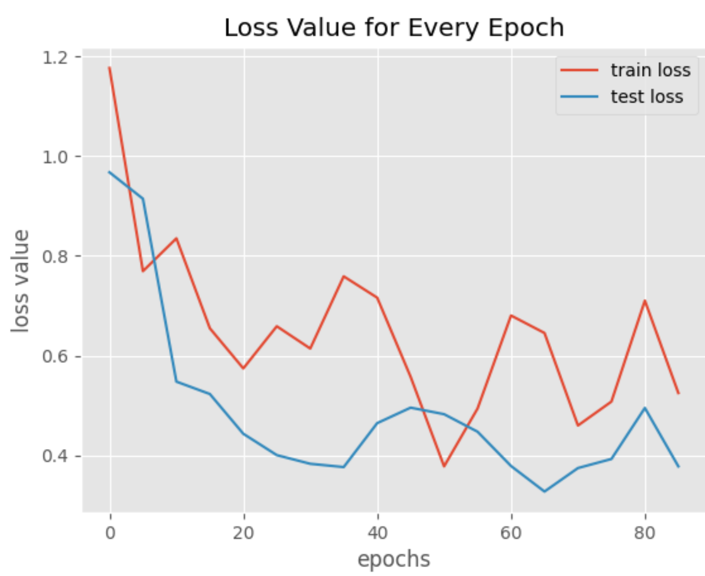
Data augmentation: We transformed the training data set as follows:

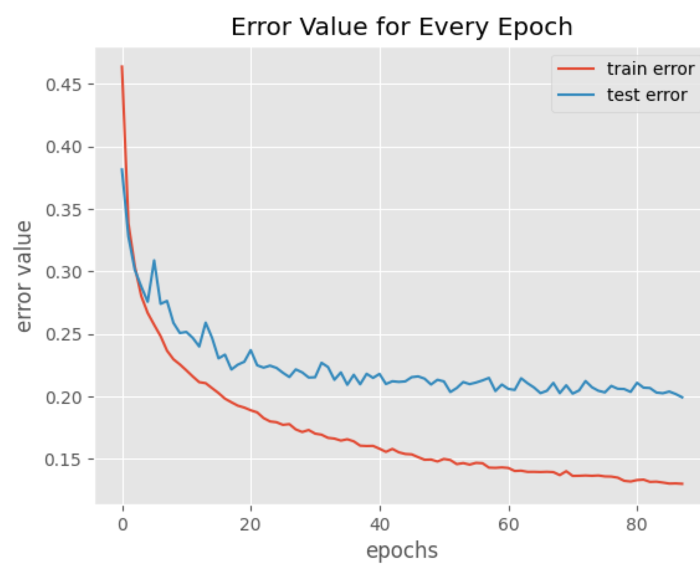
- Randomly flip images horizontally with probability 0.5

- Randomly modify the brightness of the image by up to 10%.

- Normalize all images with mean (0.491, 0.482, 0.447) and STD (0.247, 0.243, 0.262). See "Normalization Appendix" for the code and calculation used to find the mean and STD.

Convergence Graphs





In addition, in the run shown above, we exit the code after epoch 88 since the model has already received a test error < 0.2 (accuracy **> 80%** on the test set). The output is as follows:

```
Stopped at epoch: 88
Finished Training

Number of parameters:  49882
Test Accuracy of the model on the 10000 test images: 80.07%
```

Discussion

First, as we can see in the graphs above, both loss and error converge (on both training and test sets). However, the loss convergence is a bit noisy and unsmooth, specifically on the train set.

Attempts: During the implementation, I went through multiple combinations of different hyper-parameters and architectures. Tuning the learning rate resulted in poor performances, thus I kept using the initial 0.001 one. Several architectures I tried include increasing number of filters in each conv. layer, which proved to perform rather poorly as well. Eventually, I tried architectures with decreasing number of filters, and after a process of trial and error I reached the model arch. mentioned above. It's worth mentioning that I initially didn't include a dropout layer, but my results improved rather drastically once I did. However, I still had difficulty getting to 80% accuracy with my model, so I decided to try data augmentation. After some research online, I found out about the methods used to train models more effectively using image manipulation. The manipulations mentioned above are the ones I ended up using to get 80% accuracy.

Normalization Appendix

I decided not to include this code in the final file (since it doesn't have much to do with the training of the model), but I'll display the code I used below:

```
# finding train set images' mean and std
temp_transform = transforms.Compose([transforms.ToTensor()])

temp_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=False, transform=temp_transform)
temp_loader = torch.utils.data.DataLoader(temp_dataset, batch_size=1, shuffle=True)

# find mean
num_images = 0
summ = np.array([torch.zeros((32, 32)), torch.zeros((32, 32)), torch.zeros((32, 32))])
for image, _ in temp_loader:
    num_images += 1

    for channel in range(3):
        summ[channel] += image[0][channel]

image_mean = summ / num_images
image_mean = image_mean.tolist()

# find std
sum_squared = np.array([torch.zeros((32, 32)), torch.zeros((32, 32)), torch.zeros((32, 32))])
for image, _ in temp_loader:
    for channel in range(3):
        sum_squared[channel] += torch.square(image[0][channel] - image_mean[channel])

image_var = sum_squared / num_images
image_var = image_var.tolist()
image_std = [torch.sqrt(x) for x in image_var]

print(f"IMAGE MEAN: {image_mean}")
print(f"IMAGE STD: {image_std}")
```