ECE 763 Computer Vision
Project 03: Babysitting the training of DNN **Rachel Hyo Son**

Introduction

Image classification problem is the task of assigning an input image one label from a fixed set of labels. From this project, develop neural networks model which perform image classification tasks. The approach will have a loss function that can predict scores and compare the predicted score with the ground truth labels. Optimization function will be implemented to minimize the loss function.

The objective of this project is to develop a neural network model, and tune parameters to improve the performance of image classification task. The model is tuned with CIFAR-10 [1] dataset, and the model is tested with the face dataset which is collected by me. At the end, the different architecture of model is also tested.

Methodology and Result

A. Neural Network Model Setup

To develop a robust model, CIFAR-10 dataset is used. In CIFAR-10, there are a training set of N = 50,000 images, each image with 32 x 32 x 3 = 3072 pixels, and the output is 10, since there are 10 different classes.

The linear mapping classifier simply represents:

$$f\,(x_i, W, b) = W\,xi + b$$

x_i : flattened single column of images
W: Weights of the function
b: bias vector

These parameters are contributing to the output scores. The begin model is set up the 3-layer of neural network, like figure 1. It is implemented on the code as shown in Figure 2.

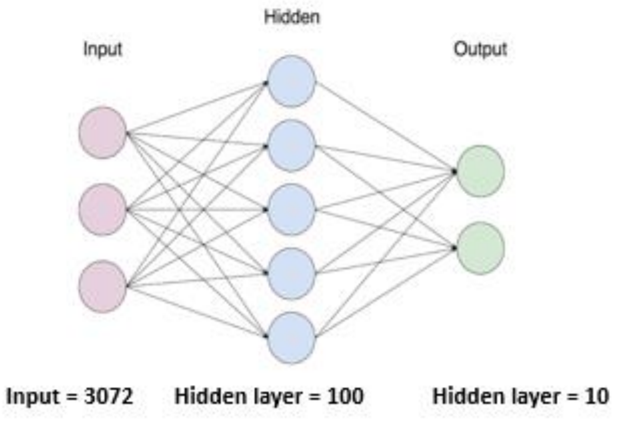
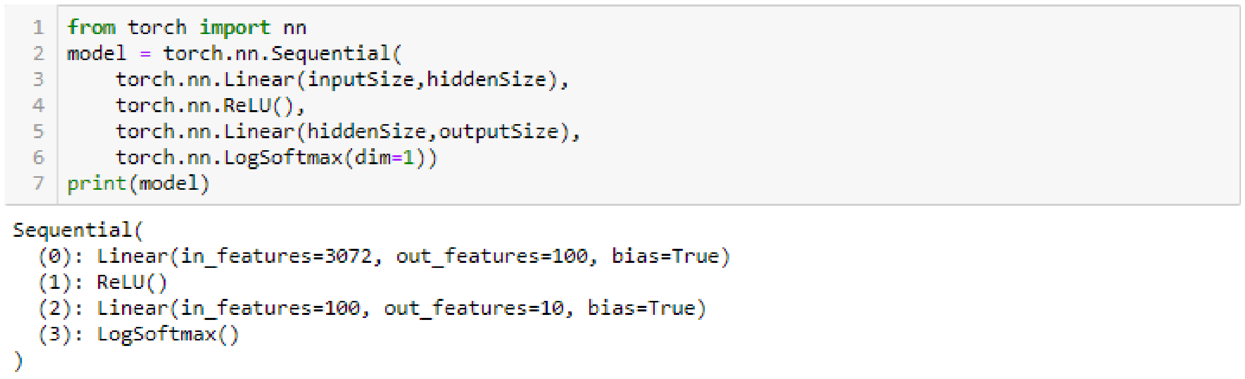


Figure 1 Neural Network Model Setting



```python
from torch import nn
model = torch.nn.Sequential(
    torch.nn.Linear(inputSize,hiddenSize),
    torch.nn.ReLU(),
    torch.nn.Linear(hiddenSize,outputSize),
    torch.nn.LogSoftmax(dim=1))
print(model)
```

```
Sequential(
  (0): Linear(in_features=3072, out_features=100, bias=True)
  (1): ReLU()
  (2): Linear(in_features=100, out_features=10, bias=True)
  (3): LogSoftmax()
)
```

Figure 2 Neural Network Model in Code

B. Activation Function

Activation functions are mathematical equations that determine the output of a neural network. It determines the score of the output from the neural network model. For classification task, the two commonly use ones are sigmoid function and Rectified Linear Unit (ReLU) function.

<u>Sigmoid function</u>

The output of sigmoid function is always going to be in range (0,1). However, the problem is that Y values tends to respond very less to changes in X toward either end of the sigmoid function. It leads to a problem of “vanishing gradient”. Small or vanished gradient refuses to learn further or is drastically slow.

<u>ReLU function</u>

ReLU is nonlinear. The range of ReLu is [0, inf]. This means it can explode the activation. The benefit of ReLU function is that sparse activation is available. The network dropout the neurons getting 0 activation from negative value of x. This will make a fewer neuron are firing and network is lighter. However, the horizontal line in ReLU can make gradient toward zero.

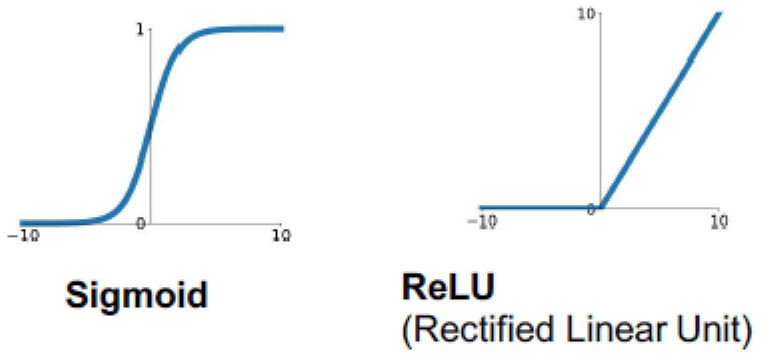


Figure 3 Sigmoid and ReLU function

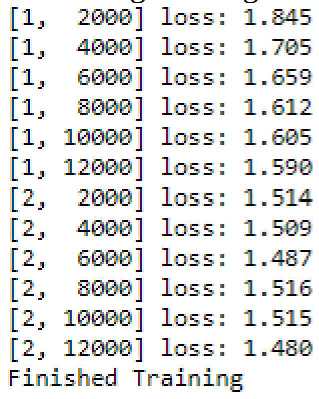
Both activation function has been tried in this project with the same setting of learning rate.
The results with CIFAR-10 dataset is:
- Sigmoid function: test-set accuracy of 42%
- ReLU function: test-set accuracy of 46%

The ReLU function is selected to our model in this development.

C. Learning Rate

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is not easy problem if it is too small may result in taking long processing time to get trained, while a too large value of it cause learning too fast or unstable training process.

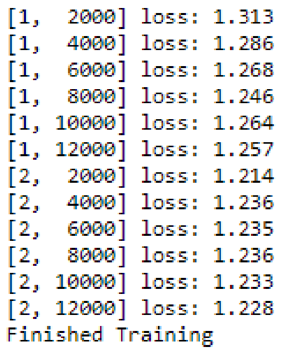
The starting learning rate was 1e-2. The result with learning rate 1e-2 is following:



```
[1,  2000] loss: 1.845
[1,  4000] loss: 1.705
[1,  6000] loss: 1.659
[1,  8000] loss: 1.612
[1, 10000] loss: 1.605
[1, 12000] loss: 1.590
[2,  2000] loss: 1.514
[2,  4000] loss: 1.509
[2,  6000] loss: 1.487
[2,  8000] loss: 1.516
[2, 10000] loss: 1.515
[2, 12000] loss: 1.480
Finished Training
```

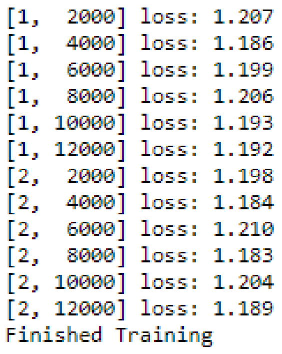
The loss is decreasing reasonable. However, to check different learning rate to see how much loss is changing.

For learning rate was 1e-3:



```
[1,  2000] loss: 1.313
[1,  4000] loss: 1.286
[1,  6000] loss: 1.268
[1,  8000] loss: 1.246
[1, 10000] loss: 1.264
[1, 12000] loss: 1.257
[2,  2000] loss: 1.214
[2,  4000] loss: 1.236
[2,  6000] loss: 1.235
[2,  8000] loss: 1.236
[2, 10000] loss: 1.233
[2, 12000] loss: 1.228
Finished Training
```

For learning rate was 1e-6:



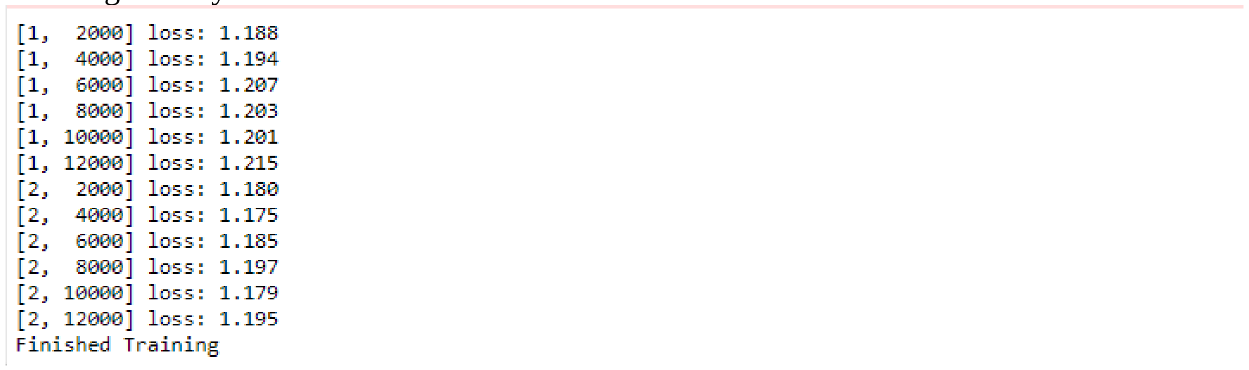
```
[1,  2000] loss: 1.207
[1,  4000] loss: 1.186
[1,  6000] loss: 1.199
[1,  8000] loss: 1.206
[1, 10000] loss: 1.193
[1, 12000] loss: 1.192
[2,  2000] loss: 1.198
[2,  4000] loss: 1.184
[2,  6000] loss: 1.210
[2,  8000] loss: 1.183
[2, 10000] loss: 1.204
[2, 12000] loss: 1.189
Finished Training
```

In case of learning rate 1e-6, the loss is not really decreasing. Based on these rough testing, I can determine some adequate range for learning rate which is [1e-2 to 1e-5].

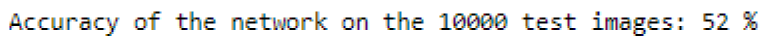
D. Regularization

Regularizations are techniques used to reduce the error by fitting a function appropriately on the given training set and avoid overfitting. It is likely adding bias terms in loss function. The regularization is easily applied with weight decay parameter in optimizer in pytorch. I used the same method with learning rate to find right range of weight decay value.

For weight decay was 1e-5:



```
[1,  2000] loss: 1.188
[1,  4000] loss: 1.194
[1,  6000] loss: 1.207
[1,  8000] loss: 1.203
[1, 10000] loss: 1.201
[1, 12000] loss: 1.215
[2,  2000] loss: 1.180
[2,  4000] loss: 1.175
[2,  6000] loss: 1.185
[2,  8000] loss: 1.197
[2, 10000] loss: 1.179
[2, 12000] loss: 1.195
Finished Training
```



```
Accuracy of the network on the 10000 test images: 52 %
```

For weight decay was 1e-6:

```
[1,  2000] loss: 1.124
[1,  4000] loss: 1.140
[1,  6000] loss: 1.129
[1,  8000] loss: 1.149
[1, 10000] loss: 1.138
[1, 12000] loss: 1.132
[2,  2000] loss: 1.110
[2,  4000] loss: 1.118
[2,  6000] loss: 1.111
[2,  8000] loss: 1.115
[2, 10000] loss: 1.134
[2, 12000] loss: 1.136
Finished Training

Accuracy of the network on the 10000 test images: 53 %
```

The accuracy about 50 % is quite good result from 3-layer neural networks.
I can determine the right value range of the weight decay is [1e-5, 1e-6].

E. Hyperparameter Optimization

Based on the study previous sections, training for loop is set up as following for hyperparameter Optimization. The random uniform function will provide random number for the weight decay and learning rate in the range base. Figure 4 shows how the hyperparameter optimization set up to run.

```
max_count = 100
for count in range(max_count):
    weight_decay = 1**np.random.uniform(-6,-2)
    learning_rate = 1**np.random.uniform(-5,-6)
```

Figure 4 Hyperparameter Optimization

With the assumption that the model is good enough over 50% of accuracy, the results with over 54% accuracy are listed in here:

```
- count : 17 | epoch : 1 batch: 4000 lr : 0.000008 wd: 0.000640, loss: 1.07 accuracy: 54.21
- count : 37 | epoch : 2 batch: 4000 lr : 0.000007 wd: 0.000441, loss: 1.06 accuracy: 54.21
- count : 44 | epoch : 2 batch: 4000 lr : 0.000006 wd: 0.003821, loss: 1.08 accuracy: 54.22
- count : 47 | epoch : 2 batch: 4000 lr : 0.000007 wd: 0.000411, loss: 1.08 accuracy: 54.21
- count : 63 | epoch : 1 batch: 10000 lr : 0.000003 wd: 0.000003, loss: 1.08 accuracy: 54.24
- count : 64 | epoch : 1 batch: 2000 lr : 0.000003 wd: 0.000014, loss: 1.06 accuracy: 54.20
- count : 67 | epoch : 1 batch: 10000 lr : 0.000009 wd: 0.000362, loss: 1.07 accuracy: 54.21
- count : 68 | epoch : 2 batch: 8000 lr : 0.000004 wd: 0.000131, loss: 1.08 accuracy: 54.20
- count : 76 | epoch : 1 batch: 4000 lr : 0.000004 wd: 0.000390, loss: 1.05 accuracy: 54.22
- count : 80 | epoch : 2 batch: 12000 lr : 0.000004 wd: 0.000084, loss: 1.07 accuracy: 54.22
- count : 82 | epoch : 1 batch: 4000 lr : 0.000006 wd: 0.000178, loss: 1.05 accuracy: 54.24
- count : 82 | epoch : 2 batch: 8000 lr : 0.000006 wd: 0.000178, loss: 1.06 accuracy: 54.21
- count : 87 | epoch : 2 batch: 10000 lr : 0.000003 wd: 0.000281, loss: 1.05 accuracy: 54.20
- count : 88 | epoch : 2 batch: 2000 lr : 0.000006 wd: 0.000006, loss: 1.06 accuracy: 54.27
- count : 89 | epoch : 1 batch: 8000 lr : 0.000004 wd: 0.000928, loss: 1.05 accuracy: 54.26
- count : 90 | epoch : 1 batch: 6000 lr : 0.000002 wd: 0.001040, loss: 1.07 accuracy: 54.22
- count : 91 | epoch : 1 batch: 4000 lr : 0.000001 wd: 0.000007, loss: 1.06 accuracy: 54.20
- count : 92 | epoch : 2 batch: 6000 lr : 0.000010 wd: 0.000060, loss: 1.07 accuracy: 54.25
- count : 93 | epoch : 1 batch: 10000 lr : 0.000002 wd: 0.000021, loss: 1.06 accuracy: 54.25
- count : 94 | epoch : 1 batch: 8000 lr : 0.000006 wd: 0.004167, loss: 1.07 accuracy: 54.25
- count : 95 | epoch : 1 batch: 8000 lr : 0.000001 wd: 0.000357, loss: 1.08 accuracy: 54.23
```

Based on the Optimization, to validate the model performance with count #88, the model has run with learning rate 6e-6 and weight decay 0.004167. It achieved overall 50% of accuracy with test dataset. The accuracies of each classes are as follow:

```
Accuracy of plane : 56 %
Accuracy of   car : 57 %
Accuracy of  bird : 40 %
Accuracy of   cat : 35 %
Accuracy of  deer : 44 %
Accuracy of   dog : 39 %
Accuracy of  frog : 56 %
Accuracy of horse : 55 %
Accuracy of  ship : 66 %
Accuracy of truck : 54 %
```

F. Face Dataloader

The face dataset from the project #1 is used to test the neural network model. In order to use pytorch, created data is provided through torch.Dataloader, which can handle transofrms and batch size. The batch size is left in 4 same as I have used in previous model building.

```
from torchvision import transforms
transforms = transforms.Compose([Resize((32,32)),
                                 ToTensor(),
                                 Normalize([0.485,0.456,0.406],[0.229,0.2245,0.225])])
```

```
trainset = torchvision.datasets.ImageFolder(root=os.path.join(root_dir,'train_dir'),
                                            transform=transforms)
```

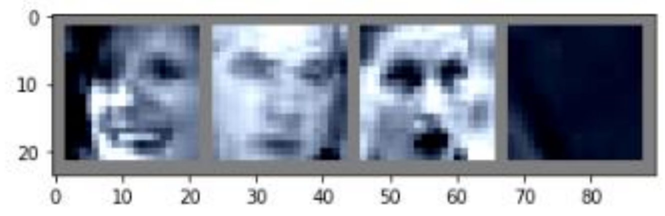
```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
```

```
testset = torchvision.datasets.ImageFolder(root=os.path.join(root_dir,'test_dir'),
                                           transform=transforms)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

```
classes = ('face','nonface')
```

With face data, the model achieved the accuracy 97%.



```
face  face  face nonface
```

Figure 5 Some of images from face data set

G. Data Augmentation

Data augmentation is a strategy that enables to increase the diversity of data available for training models, without collecting new data. The techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks.

Here, I have tested with two different augmentations: Random Crops and Horizontal Flip.

<u>Random Crops</u>

What random crops do is that crop the randomly in the region of images by set size here 20x20. The random cropping helps data augmentation while the semantics of the image are preserved. For this reason, the image cropped region should contain enough features of the face. With only randomly cropped image dataset, it cannot achieve the high number of accuracies, but including them into training dataset would increase diversity and would make robustness of the model [2].

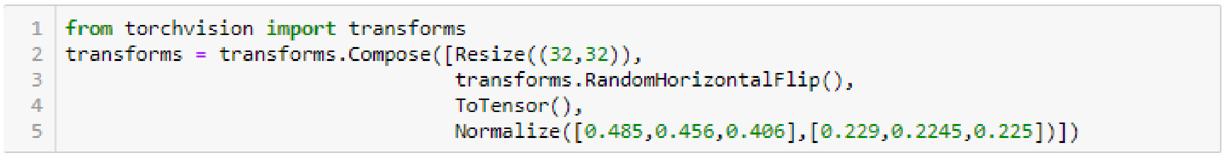
Since the model cannot reduce the loss in iteration, learning rate is changed to 1e-2 , weight decay to 0.01. This gives 57% of accuracy.

```python
from torchvision import transforms
from torchvision.transforms import RandomCrop
transforms = transforms.Compose([RandomCrop(20,20),
                                 ToTensor(),
                                 Normalize([0.485,0.456,0.406],[0.229,0.2245,0.225])
                                ])
```

<u>Horizontal Flip</u>

The horizontal flip shows the same image but flip by the vertical line. This transformation is helpful to increase variability sustaining the quality of dataset. However, for robustness of the model, it may not strongly help the face detection task because of the training the model with only front side of well cropped images. If side view of face dataset is given as test dataset, it is low chance to detect as the face, because the model has not been trained with that kind of images.

The horizontal flip transformation is applied using pytorch RandomHorizontalFlip() as shown below:



```
from torchvision import transforms
transforms = transforms.Compose([Resize((32,32)),
                                 transforms.RandomHorizontalFlip(),
                                 ToTensor(),
                                 Normalize([0.485,0.456,0.406],[0.229,0.2245,0.225])])
```

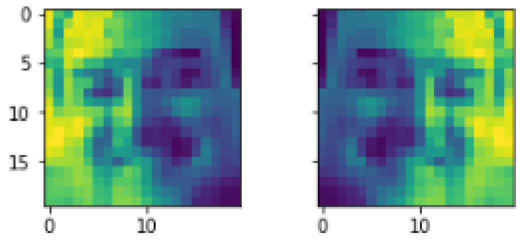


Figure 6 Image Horizontal Flip Example

Based on the result of data augmentation, only the random cropped data set has limited to get a good result of classification, but horizontal flip data set only itself can perform similar to the normal dataset. The training results will be improved if we include these two data set combined to the normal one.

H. Architecture Selection

The different model has been tested to the same dataset using the same learning rate and weight decay. The additional hidden layer is added, which makes total 4-layer model. Made assumption that determined parameters are usable in different architecture, the additional hidden layer is applied and tested the result of the model.

The new model is structured as shown in figure: 3072 input layer, 100 hidden layer, 50 hidden layer, 2 output layer for face data and 10 output layer for CIFAR-10 data.

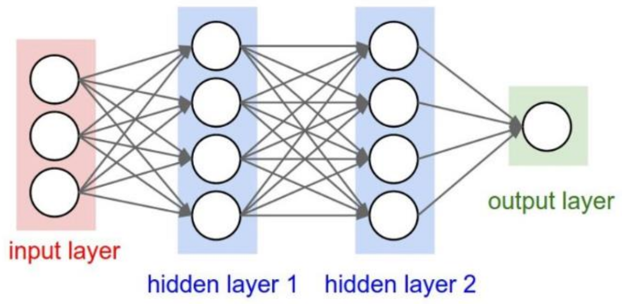
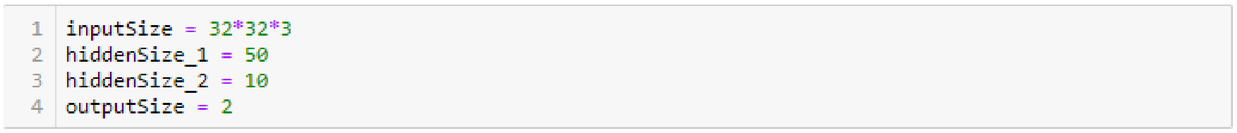
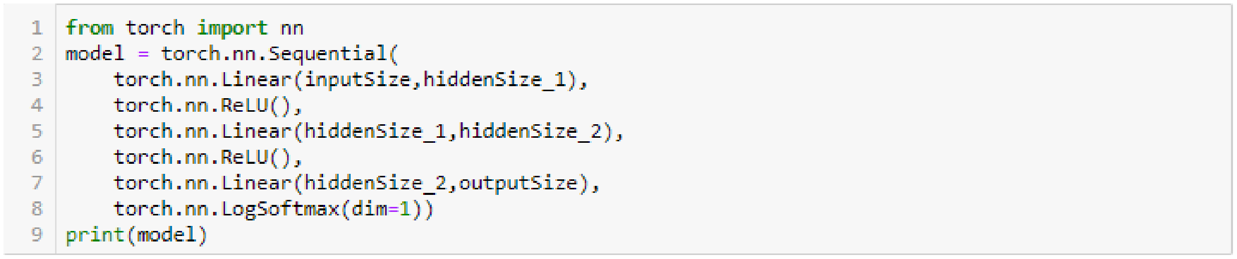


Figure 7 4-layer Neural Network Model



```
inputSize = 32*32*3
hiddenSize_1 = 50
hiddenSize_2 = 10
outputSize = 2
```



```
from torch import nn
model = torch.nn.Sequential(
    torch.nn.Linear(inputSize,hiddenSize_1),
    torch.nn.ReLU(),
    torch.nn.Linear(hiddenSize_1,hiddenSize_2),
    torch.nn.ReLU(),
    torch.nn.Linear(hiddenSize_2,outputSize),
    torch.nn.LogSoftmax(dim=1))
print(model)
```

Conclusion

The project has successfully developed classification model for 10 different objects as well as 2 binary face and non-face detection. The project begins with the simple model with 3-layer of neural network. While the model is simple, the selecting activation function and tuning the parameter leads to increase the performance, which result in low loss and give a high accuracy. Since ReLU function can have sparse activation, it can reduce the time to training the model. However, in duration of training, the loss can be exploded when learning rate is not quite right performed. Hyperparameter optimization has been completed with for loop and found good learning rate and weight decay pair and finalize the model parameter. The good enough model retains train model accuracy 60% and test accuracy 56% with CIFAR-10 dataset. The model is tested on the face dataset I have previously prepared. The face data is binary classification task, and it is much easier to detect with the same model, achieved 97% accuracy of face classification.

Reference

[1] Krizhevsky, Alex, and Geoff Hinton. "Convolutional deep belief networks on cifar-10." Unpublished manuscript 40.7 (2010): 1-9
[2] Takahashi, Ryo, Takashi Matsubara, and Kuniaki Uehara. "Data augmentation using random image cropping and patching for deep CNNs." IEEE Transactions on Circuits and Systems for Video Technology (2019).
[3] Dr. Wu, “ECE 763 Computer Vision Lecture Note”
.