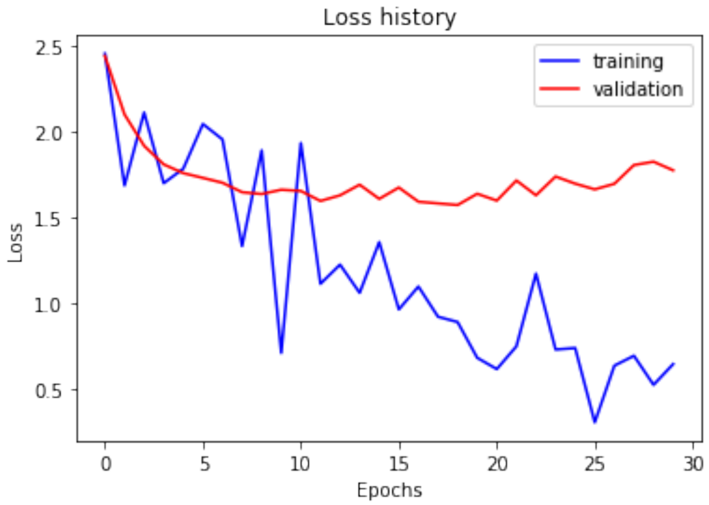
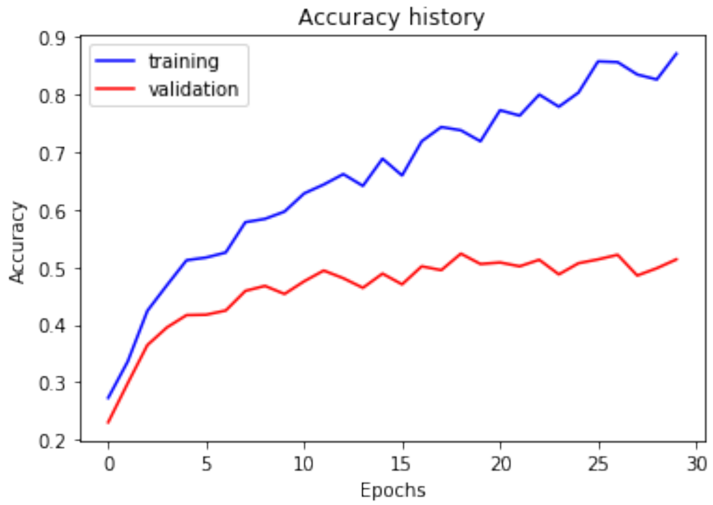
CS 4476 Project 6

Jie Lyu
jie.lyu@gatech.edu
903329676

Part 1: Your Training History Plots





Final training accuracy value: 0.8703517587939699

Final validation accuracy value: 0.5133333333333333

Part 1: Experiment: play around with some of the parameters in nn.Conv2d and nn.Linear, and report the effects for: 1. kernel size; 2. stride size; 3. dim of nn.Linear. Provide observations for training time and performance, and why do you see that?

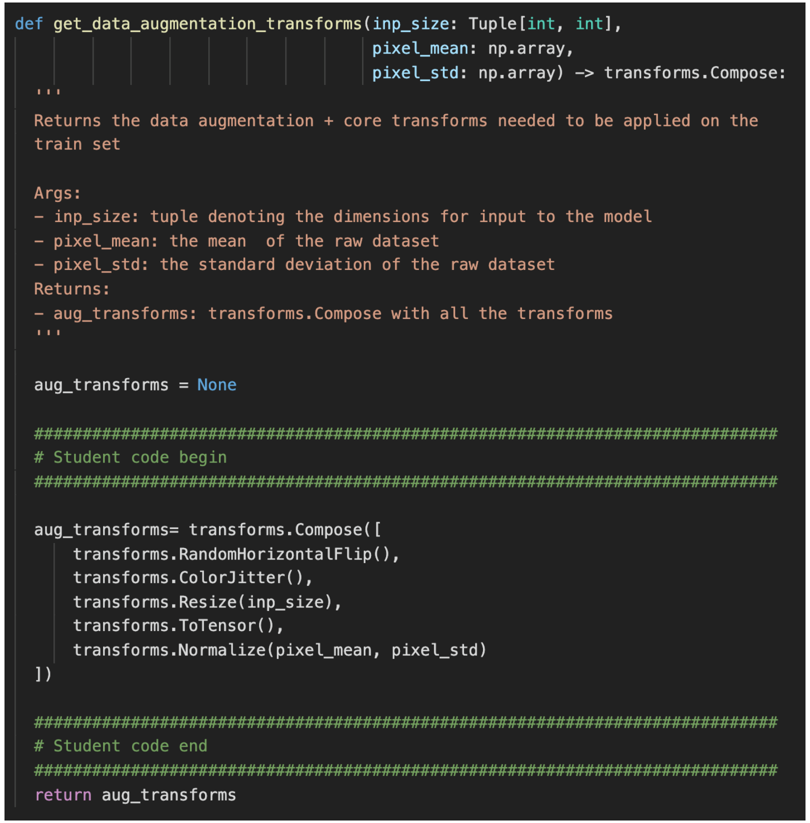
(Base Performance: With default parameters, optimized with Adam wit "lr": 1e-3, "weight_decay": 1e-2, first 5 epochs gives an average loss of 1.2)

Kernel size: When I increased kernel size to (15,15) for both Conv2d layers, performance increased significantly with an average loss of 0.85 for the first 5 epochs, and training time decreased. The reason for the increase of performance may be because with a bigger kernel, the filter is able to learn more of a feature than the tiny details in the picture. With a bigger kernel size, the output size will be smaller, thus less computation needed and the training speed is faster.

Stride size: When I increased stride size to 5 for both Conv2d layers, performance increased with an average loss of 0.70 for the first 5 epochs, and training time again decreased. Again, increasing the stride size decreases the output size, thus speeding up the training. However, as I increase it to 100, performance is not as good as 5.

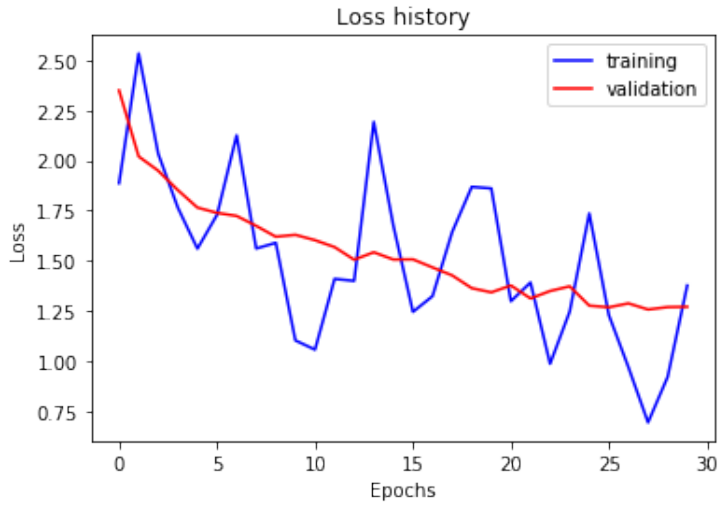
Dim of nn.Linear: When I increased nn.Linear to be 1000 -> 200 -> 15, speed decreased as neuron thus more computation is needed. Training performance is slightly improved, however the loss fluctuates move in the early phase of training.

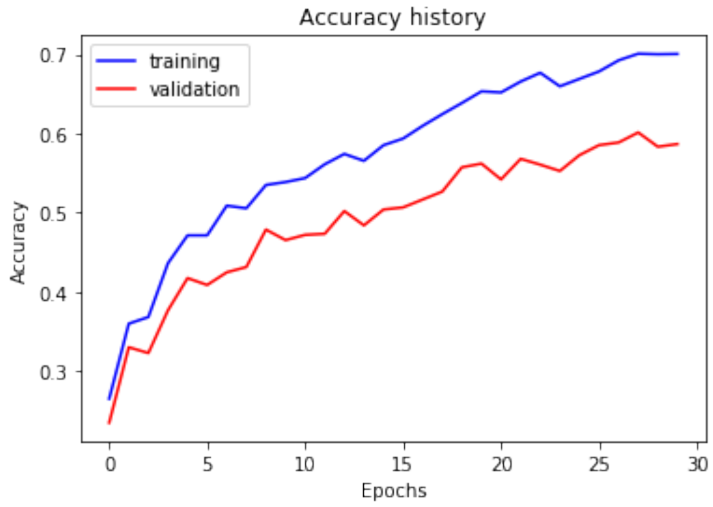
Part 2: Screenshot of your get_data_augmentation_transforms()



```python
def get_data_augmentation_transforms(inp_size: Tuple[int, int],
                                     pixel_mean: np.array,
                                     pixel_std: np.array) -> transforms.Compose:
    '''
    Returns the data augmentation + core transforms needed to be applied on the
    train set

    Args:
    - inp_size: tuple denoting the dimensions for input to the model
    - pixel_mean: the mean  of the raw dataset
    - pixel_std: the standard deviation of the raw dataset
    Returns:
    - aug_transforms: transforms.Compose with all the transforms
    '''

    aug_transforms = None

    #############################################################################
    # Student code begin
    #############################################################################

    aug_transforms= transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.ColorJitter(),
        transforms.Resize(inp_size),
        transforms.ToTensor(),
        transforms.Normalize(pixel_mean, pixel_std)
    ])

    #############################################################################
    # Student code end
    #############################################################################
    return aug_transforms
```

Part 2: Your Training History Plots





Final training accuracy value: 0.700502512562814

Final validation accuracy value: 0.5866666666666667

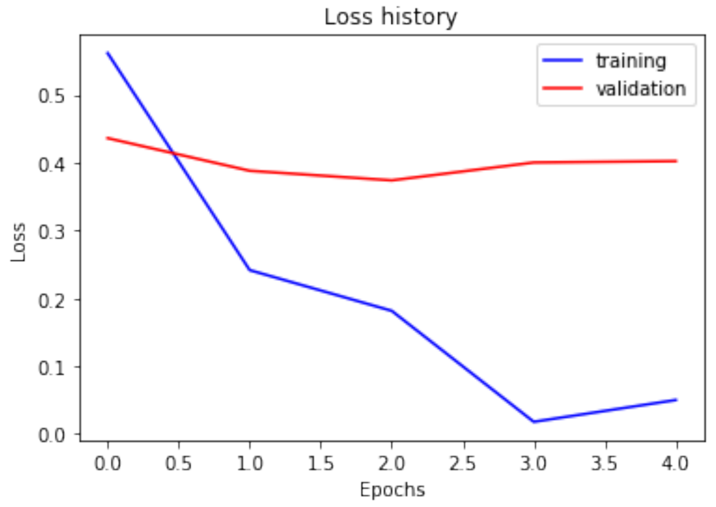
Part 2: Reflection: compare the loss and accuracy for training and testing set, how does the result compare with Part 1? How to interpret this result?

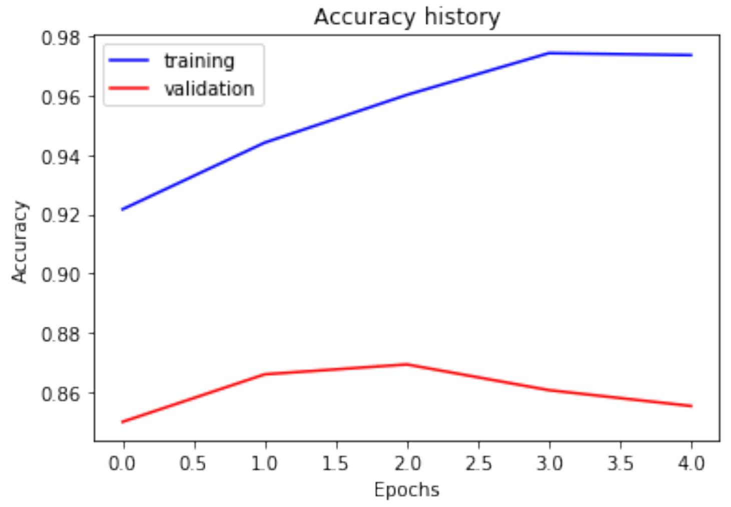
1. Training loss in part2 doesn’t drop as much as training loss in part1, and is more fluctuated. It is because the dropout layer is constantly dropping out weights, so it doesn’t learn the training data well

2. Validation loss in part2 drops continuously, while the validation loss in part1 shows a concave shape indicating an overfit. It is because in part2 we did dropout and transform to minimize the effect of overfitting, so model trained in part2 is less overfitted

3. The gap between training and validation accuracy curve for part2 is smaller than that of part1. This is again because of the overfitting problem in part1. Model in part1 is overfitted for training data, therefore doesn’t generalize well for validation data.

Part 3: Your Training History Plots





Final training accuracy value: 0.9735343383584589

Final validation accuracy value: 0.8553333333333333

Part 3: Reflection: what does fine-tuning a network mean?

It means to train a pre-trained model on our own data set. Training a deep neuron network is expensive, so in this project we used the pre-trained AlexNet. However, the original one has 1000 output neurons while we only have 15 categories, so we need to swap out the final linear layer with 1000 neurons and replace it with a layer of 15 neurons, and learn the weights connected to the new layer. Fine-tuning also make the model more task specific and allow us to achieve a better performance on our dataset. Since the model has been trained extensively on a large dataset, features are wells learnt in terms of weight. It is especially helpful to minimize overfitting when we only have a small training set. Using a pre-trained model and “freeze” some layers, we can also save training time because extensive training has been done already.

Part 3: Reflection: why do we want to “freeze” the conv layers and some of the linear layers in pretrained AlexNet? Why CAN we do this?

We can do this because the AlexNet we have here has been pre-trained on millions of images. Extensive training has been done. Features and weights have been learned. Since it has been trained on images and we are using AlexNet for image classification, tasks are similar and the features and edges learned are similar. We don’t need to modify the previously learnt weights anymore, so we “freeze” them. Also, we want to do it because if we don’t freeze them, training time will be significantly longer because the amount of neurons in this NN is huge.

Conclusion: briefly discuss what you have learned from this project.

It is nice to see the power of NN. Even a simple one like SimpleNet gives us great performance and a lot of space to tweak around like implementing the optimizer, adjusting hyper-parameters in NN etc. I learnt a lot from part2. I have heard of techniques like dropout and flipping the image, but it is my first time doing it myself. I was surprised how easy that was. The performance for AlexNet surprised me. I learnt how fine-tune a pre-trained NN. I think the idea of fine-tuning is useful because I can achieve a great performance without spending a large amount of time training.

Code and Misc. (DO NOT modify this page)

Part 1

Part 2

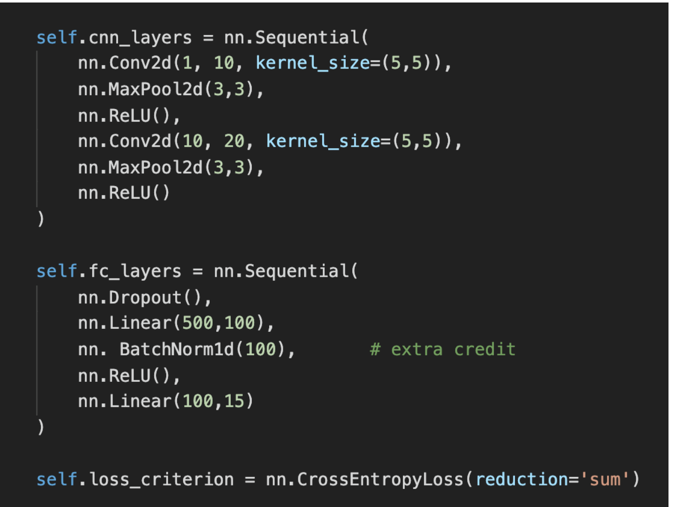
Part 3

Late hours

Violations

Extra Credit

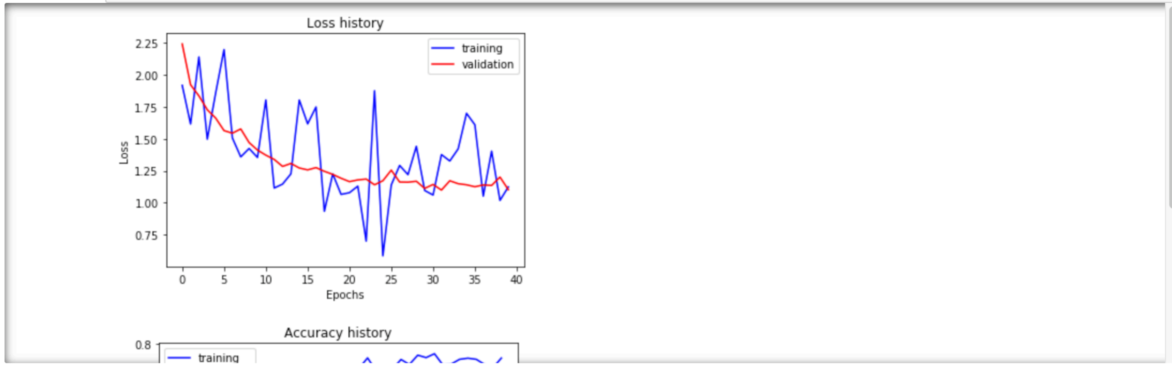
I added a BatchNorm1d layer in FC layer and achieved a validation accuracy of 0.6353



```
self.cnn_layers = nn.Sequential(
    nn.Conv2d(1, 10, kernel_size=(5,5)),
    nn.MaxPool2d(3,3),
    nn.ReLU(),
    nn.Conv2d(10, 20, kernel_size=(5,5)),
    nn.MaxPool2d(3,3),
    nn.ReLU()
)

self.fc_layers = nn.Sequential(
    nn.Dropout(),
    nn.Linear(500,100),
    nn. BatchNorm1d(100),       # extra credit
    nn.ReLU(),
    nn.Linear(100,15)
)

self.loss_criterion = nn.CrossEntropyLoss(reduction='sum')
```

the plots onto the report, and answer the questions accordingly.

```
In [15]: trainer.plot_loss_history()
         trainer.plot_accuracy()
```



```
In [16]: train_accuracy = trainer.train_accuracy_history[-1]
         validation_accuracy = trainer.validation_accuracy_history[-1]
         print('Train Accuracy = {}; Validation Accuracy = {}'.format(train_accuracy, validation_accuracy))

         Train Accuracy = 0.7668341708542713; Validation Accuracy = 0.6353333333333333
```