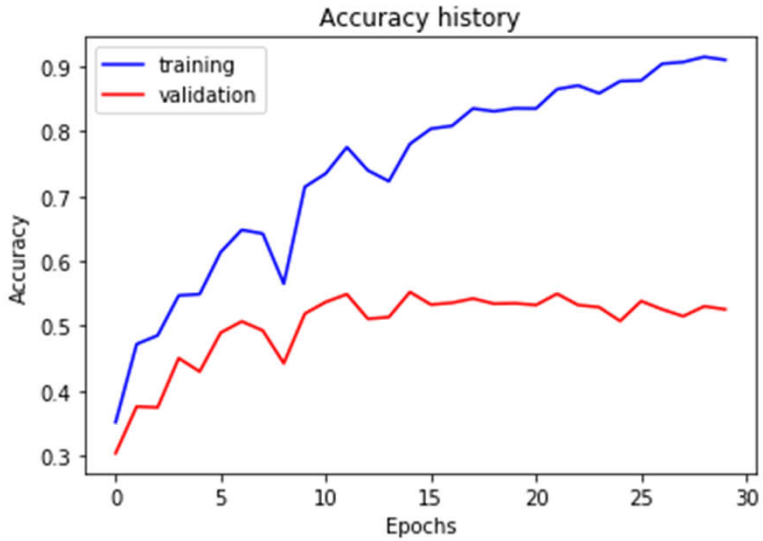
CS 4476 Project 6

Jian Yu Kok
jkok7@gatech.edu
903550380

Part 1: Your Training History Plots

<Loss plot here>

<Accuracy plot here>



Final training accuracy value: 0.9102177554438861

Final validation accuracy value: 0.5253333333333333

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?

Kernel size: Increasing kernel size made training time longer. This is because convolution now occurs over a bigger area on the image, which results in more computation. It also appears to make the training accuracy slightly lower and validation accuracy slightly higher. This might be due to each pixel being more averaged now, which might have helped slightly with overfitting.

Stride size: Increasing Stride size reduces training time. This is because the model now have lesser computations to do. In a 6x6 image, 3x3 kernel runs 16 times with stride 1. But if given a stride of 3, it only runs 4 times. However, both the training and testing accuracy dropped. Training accuracy dropped by around 30% while testing accuracy dropped by 10%. This could be because increasing the stride resulted in a lower resolution, which made the model worst.

Dim of nn.Linear: Increasing Dim of nn.Linear increases the training time and result in an increase in training accuracy. On the other hand, Validation accuracy dropped slightly. This could be because increasing the dimension made the function more complex, which might have resulted in more overfitting compared to a lower dim.

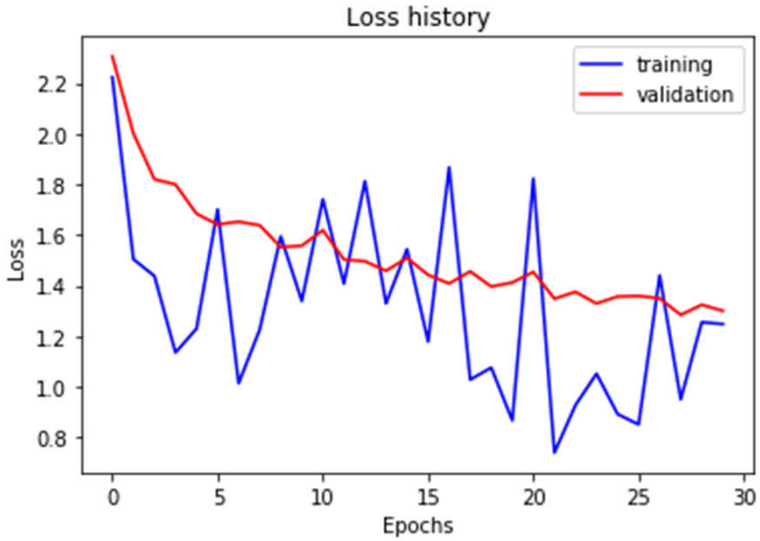
Part 2: Screenshot of your get_data_augmentation_transforms()

<Screenshot here>

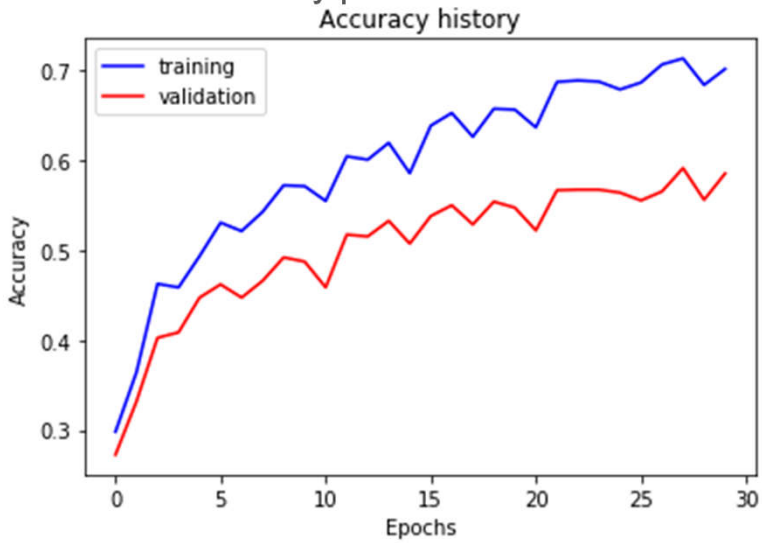
```
aug_transforms = None

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

# Add data augmentation transforms here
# transforms.RandomHorizontalFlip() and transforms.ColorJitter()

aug_transforms = transforms.Compose([
  transforms.RandomHorizontalFlip(),
  transforms.ColorJitter(),
  transforms.Resize((inp_size[0], inp_size[1])),
  transforms.ToTensor(),
  transforms.Normalize(pixel_mean, pixel_std)
])
# Copy over fundamental transforms here
############################################################################
# Student code end
############################################################################
```

Part 2: Your Training History Plots

<Loss plot here>



<Accuracy plot here>



Final training accuracy value: 0.7015075376884422

Final validation accuracy value: 0.5853333333333334

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?

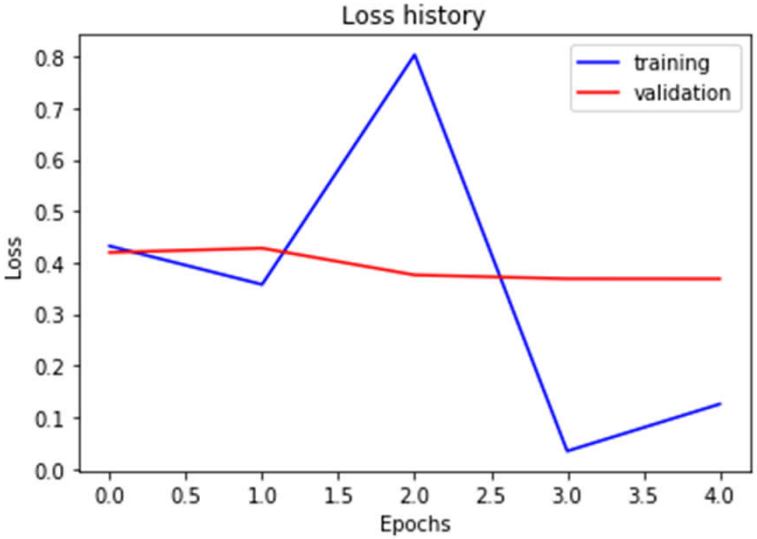
Accuracy is higher with the training set compared to testing set.

Loss is also generally lower for the training set compared to testing set.

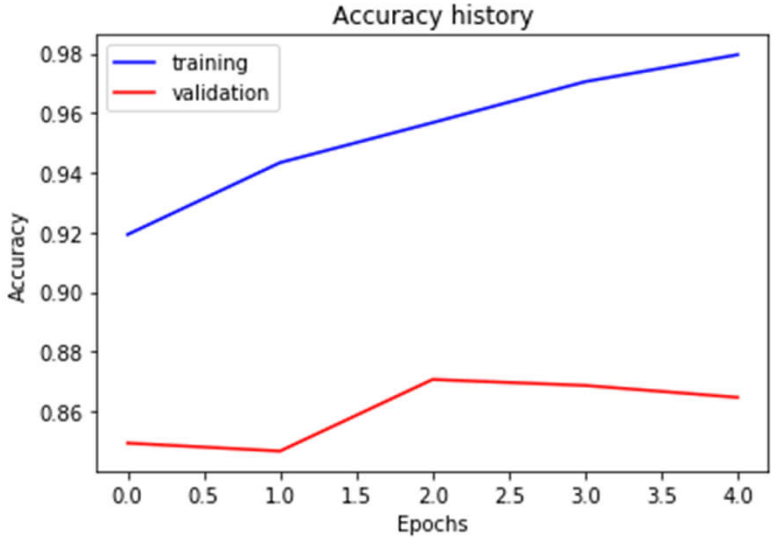
However, compared to part 1, the difference between the accuracy and loss for both training set and testing set is smaller. This shows that the model is not as overfitted as compared to part 1, as the model now doesn’t perform as well for training data, but performs better for testing data.

Part 3: Your Training History Plots

<Loss plot here>



<Accuracy plot here>



Final training accuracy value: 0.9795644891122278

Final validation accuracy value: 0.8646666666666667

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

Fine-tuning a network is to make minor/small changes to the network to make it work better for your situation. For our case, it will be to modify AlexNet, which outputs up to 1000 classes, to output 15 classes instead.

For this case. since AlexNet has already been pre-trained, rather than retraining the whole network ourselves, we utilised the pre-trained weights and make minor adjustments final layer of the network, and only train that layer. This allows us to get better results in a shorter amount of time as we do not have to run the computationally heavy training as long.

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 do this because we wish to utilise the pre-trained weights that we know works well. The images and outputs AlexNet make is also relatively similar to ours. Therefore, we can freeze all the layers except the final linear layer as the features that have been retrieved by AlexNet before classification should be sufficient to make our prediction.

If it ever comes to a case where the images and output are extremely different from what AlexNet was built for, for example, trying to predict if someone has cancer given image of cells, freezing the layers will probably result in a case where the model will probably not perform well as the features learned will not translate well to the new situation.

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

I have learned how to utilize pytorch to create a deep learning image classifier. I now know how to create a brand new neural network, and also learned how to utilise existing pre-train model to make my own image classifier.

Code and Misc. (DO NOT modify this page)

Part 1

Part 2

Part 3

Late hours

Violations

Extra Credit

<Discuss what extra credit you did and analyze it. Include images of results as well >