BMI 826 / CS 838 Assignment 2 Report

Wen-Fu Lee (wlee256@wisc.edu)
Huawei Wang (hwang665@wisc.edu)
Shuoxuan Dong (sdong34@wisc.edu)

3.1 Understanding Convolutions

- Implementation

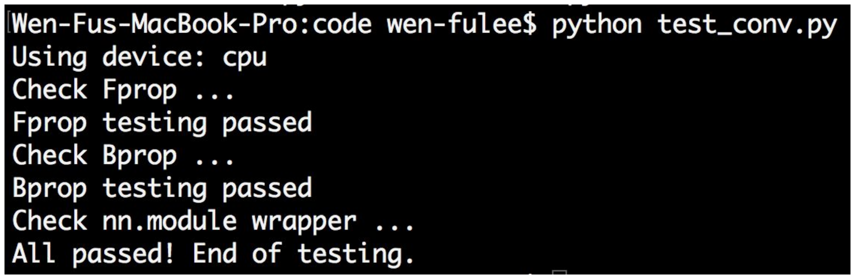
  I mainly referred to the lecture slides of Lecture "5-7 Deep Learning-compressed.pdf" at pages 96 and 98-99 to implement forward and backward propagations, respectively.

  For forward propagation, the ideas is that we unfold the input and weight using torch fold function, and then do matrix multiplication to get unfolded output. Finally, we fold the unfolded output back to the output we want. **One thing the lecture slides don't mention is how to add bias to the output.** I spent some time to figure out that we could just reshape it and add it to the result of matrix multiplication of input and weight. Note that we also save some tensors to ctx to save the time to recalculate them later, such as unfolded input and weight.

  As for backward propagation, it's pretty much the idea of forward propagation implementation. To get input gradient, we unfold the output gradient and multiply it with the unfolded weight already saved in ctx, and then fold the result back. To get weight gradient, we multiply the unfolded output with the unfolded input already saved in ctx, and then fold the result back.

  In this process, I have learned from mistakes about how to choose correct parameters for fold and unfold functions, and know the tensors' size and dimension we need to deal with.

- Screenshot of passing the test code



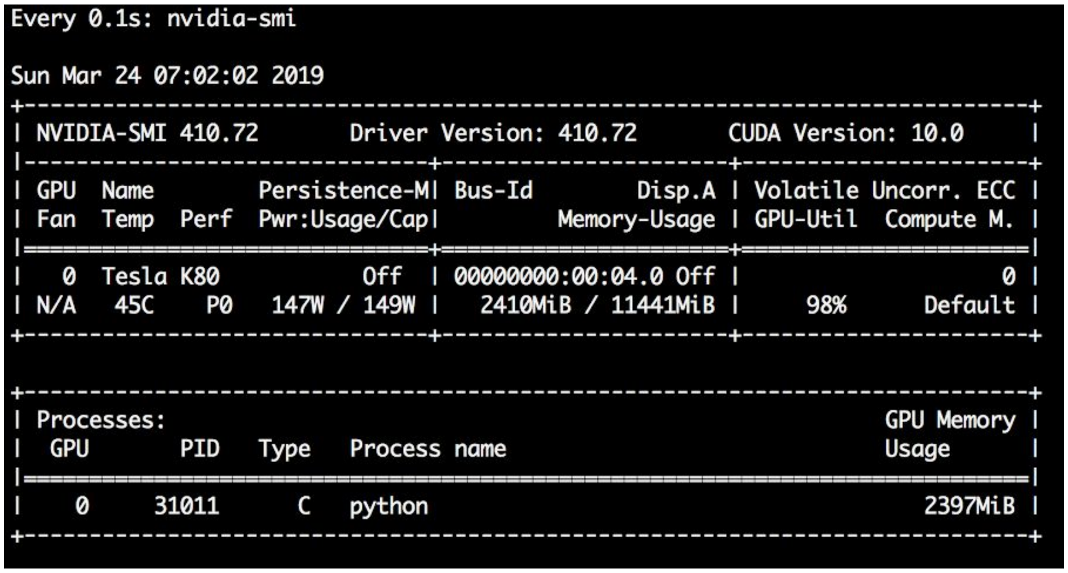
```
Wen-Fus-MacBook-Pro:code wen-fulee$ python test_conv.py
Using device: cpu
Check Fprop ...
Fprop testing passed
Check Bprop ...
Bprop testing passed
Check nn.module wrapper ...
All passed! End of testing.
```

3.2 Design and Train a Convolutional Neural Network

- Section 0

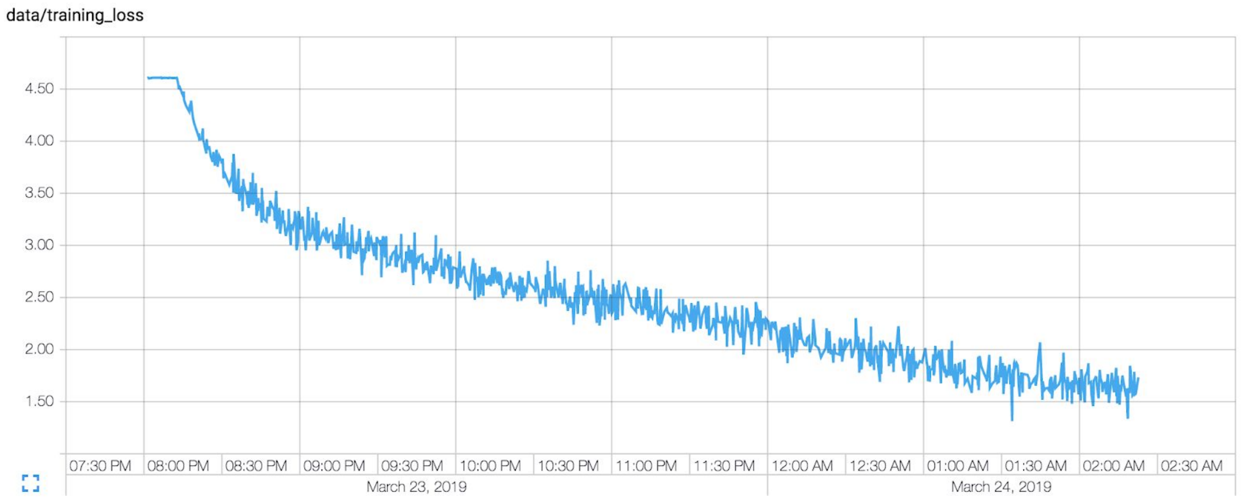
  We trained SimpleNet on GCP, and show its performance below.

  1) Training memory: **2,397 MiB**

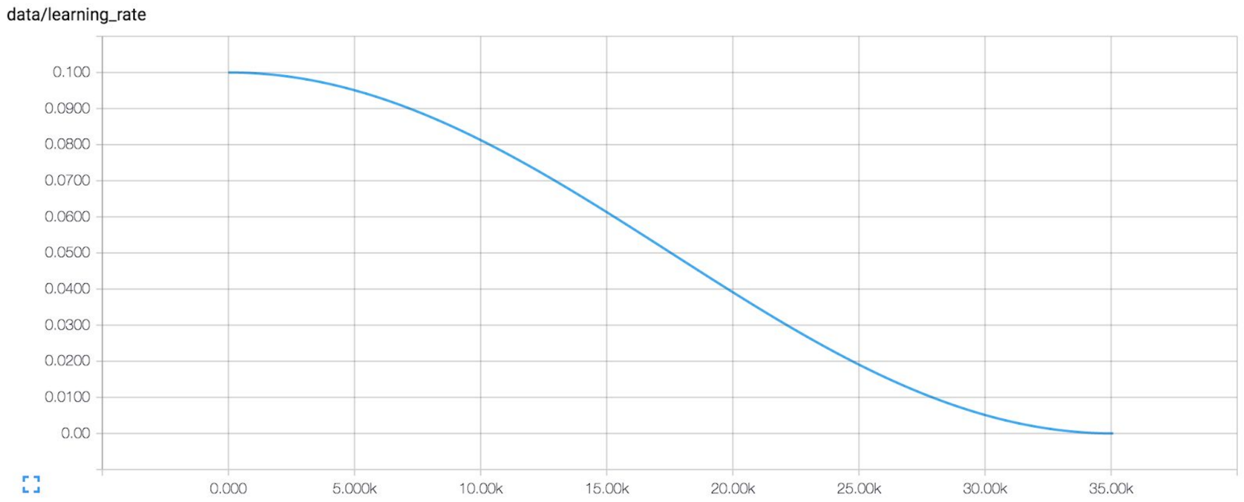


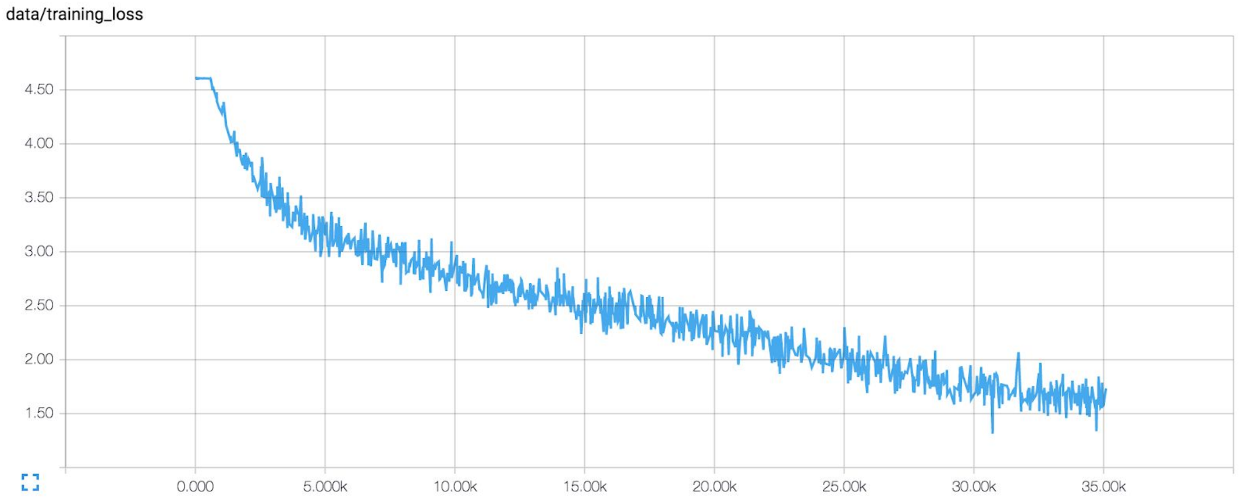
```
Every 0.1s: nvidia-smi

Sun Mar 24 07:02:02 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.72       Driver Version: 410.72       CUDA Version: 10.0      |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   45C    P0   147W / 149W |   2410MiB / 11441MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     31011      C   python                                     2397MiB |
+-----------------------------------------------------------------------------+
```

  2) Training speed: **4.24 min/epoch** (6 hours and 22 min to finish 90 epochs)



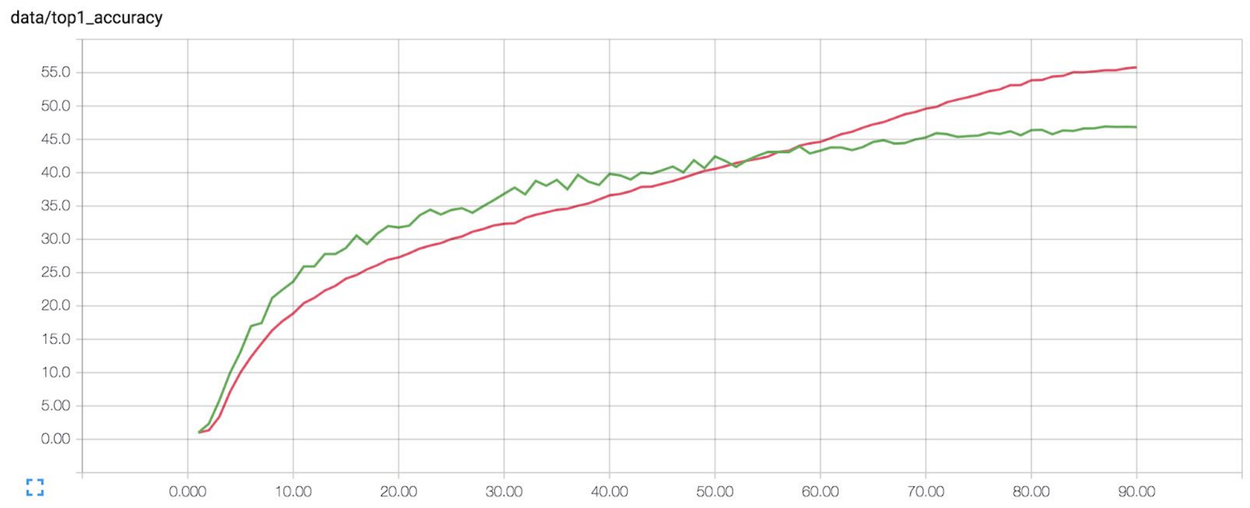
3) Training curves



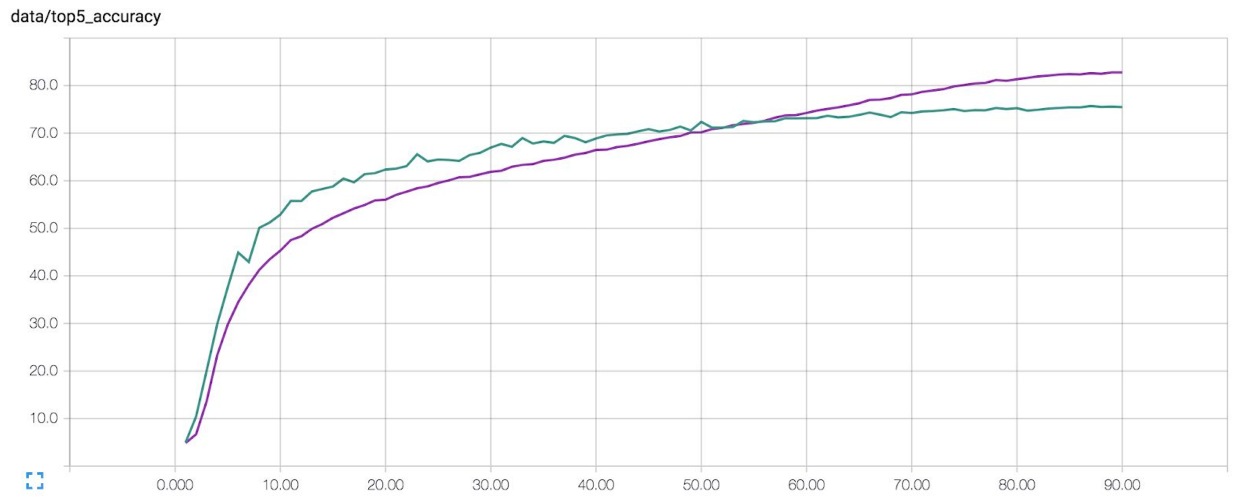


4) Training/validation accuracy

<u>Note: red line: training set; green line: validation set</u>

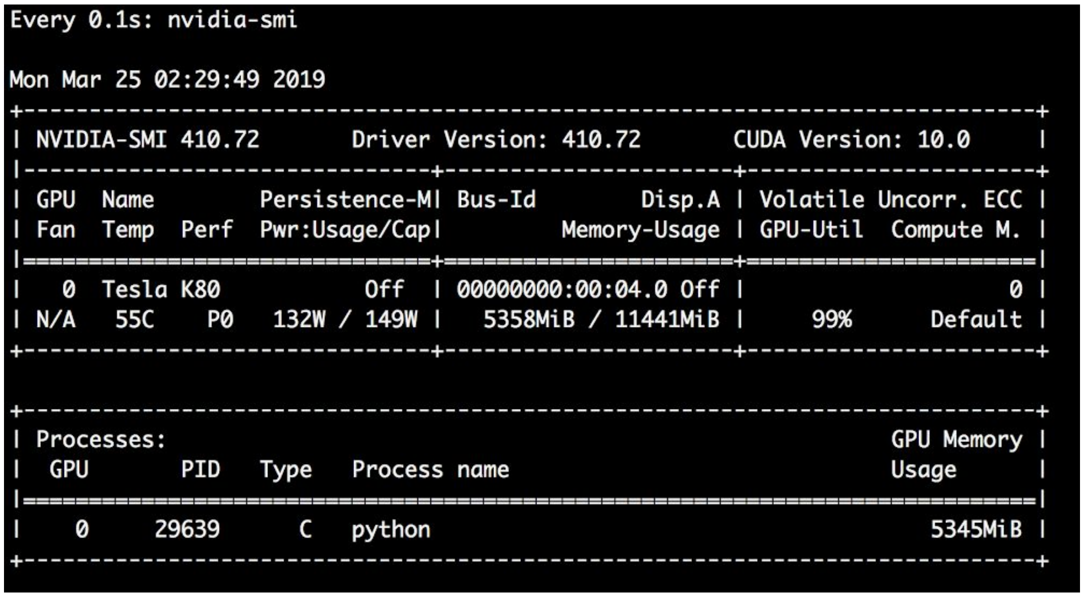


<u>Note: purple line: training set; green line: validation set</u>



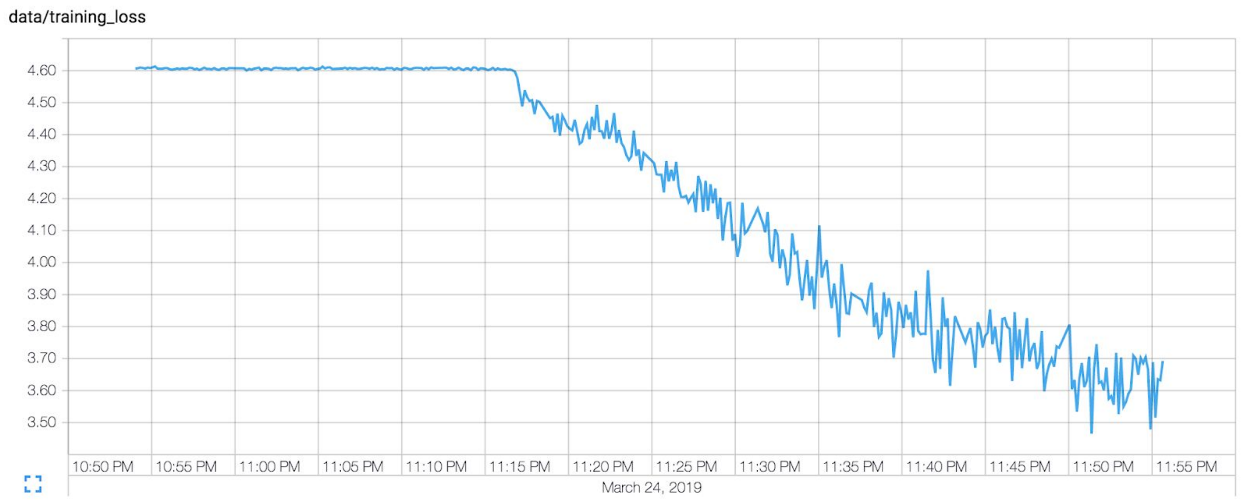
- Section 1
  1) Which loss function/optimization method is used?
     CrossEntropyLoss() is used for loss function; torch.optim.**SGD** with the effects of momentum and weight decay is used for optimization method.
  2) How is the learning rate scheduled?
     The learning rate is scheduled using **cosine learning rate decay**
  3) Is there any regularization used?
     Yes, **weight decay** is used for regularization here.
  4) Why is top-K accuracy a good metric for this dataset?
     Since this dataset contains 100 classes, unlike binary classification (the output is always yes or no), top-K accuracy can more precisely evaluate a trained model's performance. For example, assume that there are two trained models with the same top-1 accuracy, but if one of both models has higher top-5 accuracy, that model is still better than the other one since it prediction of top 5 classes is closer to the ground truth.
  5) Others
     This training is SGD on mini-batches. According to the default setting, there are 90 epochs. In each epoch, there are 390 mini-batches. In each mini-batch, there are 256 training samples.

- Section 2
  We trained SimpleNet using our own custom convolution on GCP, and show its performance below.
  1) Training memory: **5,345 MiB**

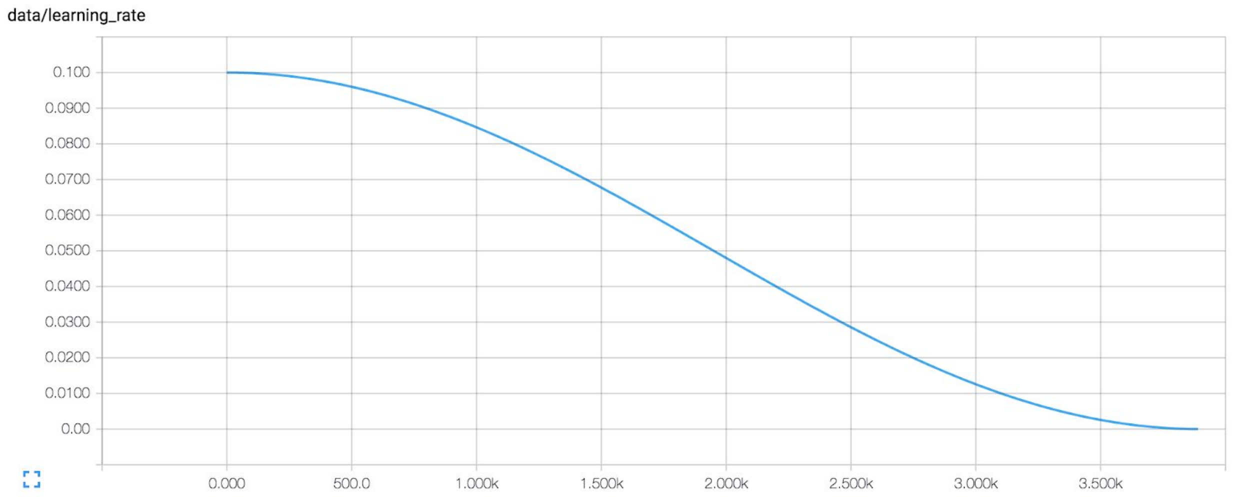


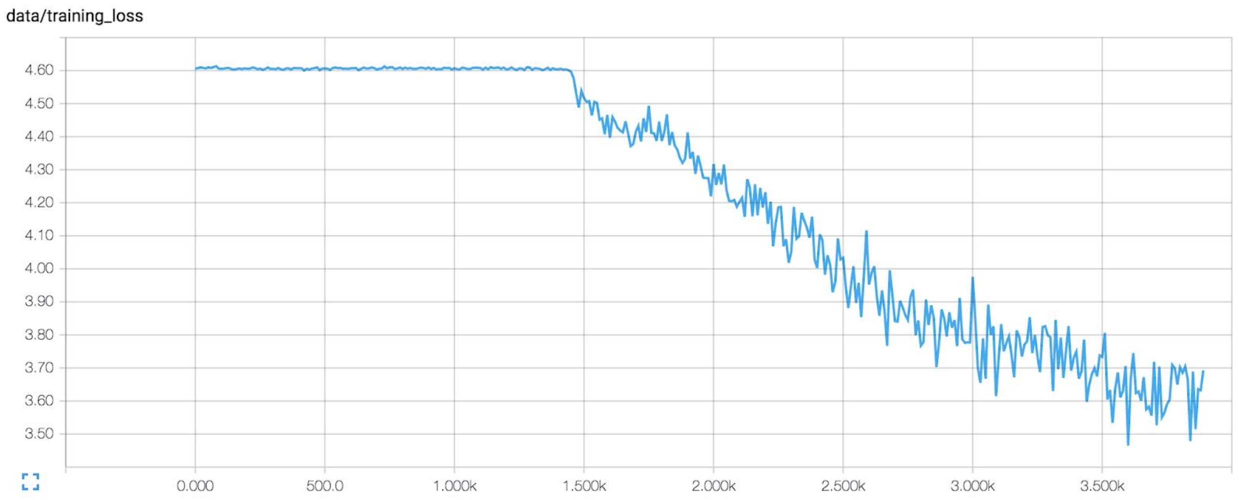
```
Every 0.1s: nvidia-smi

Mon Mar 25 02:29:49 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.72       Driver Version: 410.72       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   55C    P0   132W / 149W |   5358MiB / 11441MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     29639      C   python                                     5345MiB |
+-----------------------------------------------------------------------------+
```

2) Training speed: **6 min/epoch** (1 hour to finish 10 epochs)



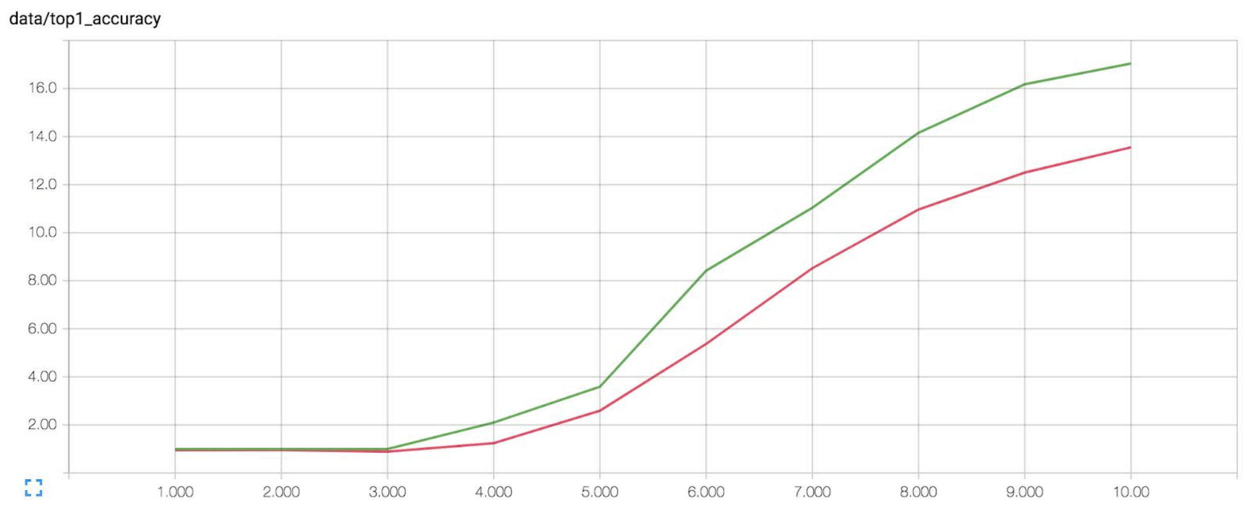
3) Training curves



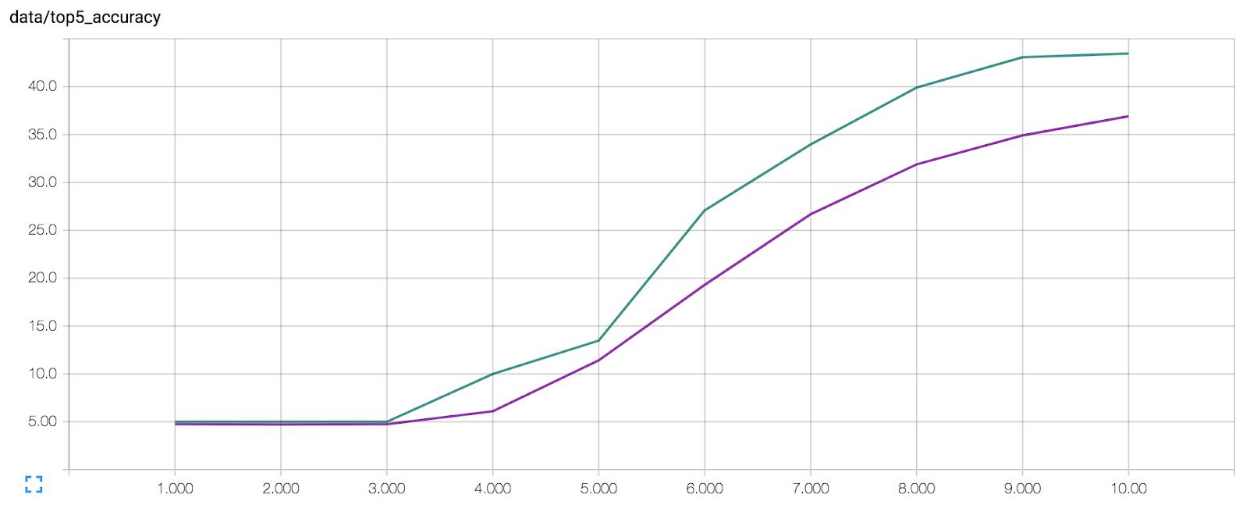


4) Training/validation accuracy

<u>Note: red line: training set; green line: validation set</u>

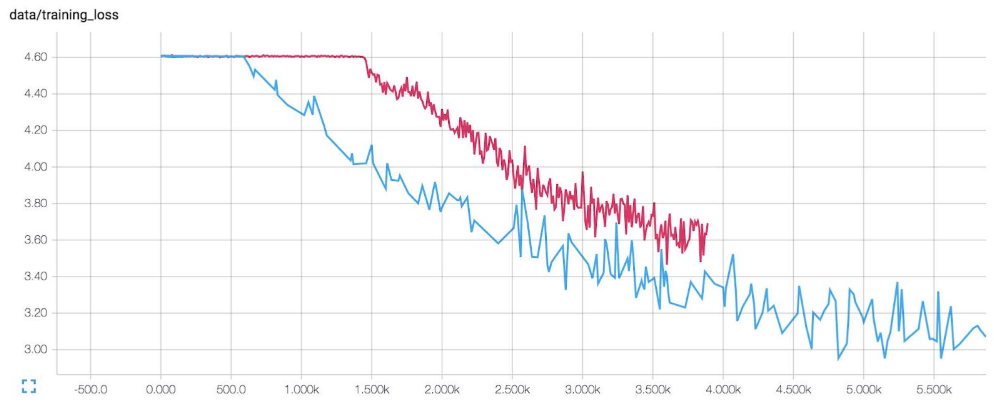


<u>Note: purple line: training set; green line: validation set</u>



5) Comparison with PyTorch's convolution

According to the summary table below, we found that PyTorch's convolution outperforms our own custom convolution in every performance index. We guess that PyTorch must accelerate their convolution using some techniques just like fold/unfold we used in our own custom convolution. Moreover, they must also optimize other convolution in other ways to make it occupy less GPU memory and faster computation speed, which we don't clearly know. As for the convergence rate, since both convolution methods give the same output values, basically their convergence rates are pretty similar.

	PyTorch's convolution	Our own custom convolution
Training memory	**2,397 MiB**	**5,345 MiB**
Training speed	**4.24 min/epoch**	**6 min/epoch**
Convergence rate	 Note: red line: Our own custom convolution; blue line: PyTorch's convolution We can observe the above two lines basically parallel with each other so that their convergence rates are pretty similar although the training loss of using our own convolution is stuck longer before going down.	

6) Lessons learned

In my first version of custom convolution, it is pretty slow since I used lots of **torch.zeros()** and **for loops** which take much computation effort. After studying some example codes in the PyTorch tutorial, the codes finally can be optimized like the current version. I do learn a lot about how to write efficient PyTorch codes in this assignment.

- Section 3

  We made modifications to the original simpleNet.

  Several attempts were made, for specifications please refer to the code:

  1. **CustomNet** - 4 residual blocks were used to build this structure. In each block, two convolution layers and two batch normalization layers were used with the ReLU activation function. This network tends to have overfitting issue
  2. **Modified ResNet**
  3. **Modified GoogleNet**
  4. **Modified SimpleNet** with batch normalizations and dropout layer yields best performance

  Batch normalization after each block for improving the performance and stability of the network. It is used to normalize the input layer by adjusting and scaling the activations.

  Random dropout is added to the network. probability of an element to be zero-ed was set to be 0.5. Randomly zero out entire channels of the input tensor. Each channel will be zeroed out independently on every forward call. with probability 0.5 using samples from a Bernoulli distribution. If adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then i.i.d. dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease.

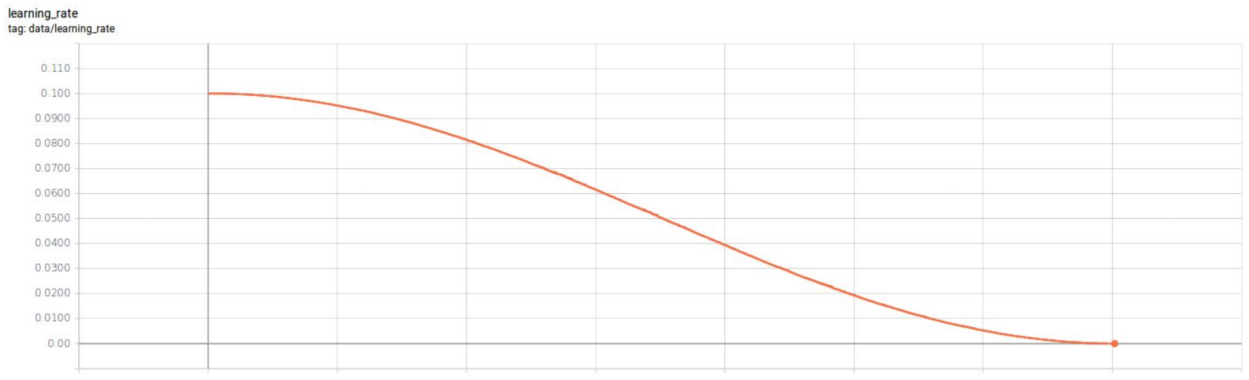
  In this case, nn.Dropout2d() will help promote independence between feature maps and should be used instead.

  We achieved 50.730% accuracy on test set and 78.88% on training set after 90 epochs of training. Performance of the model is shown in the following figures from tensorboard visualizations.

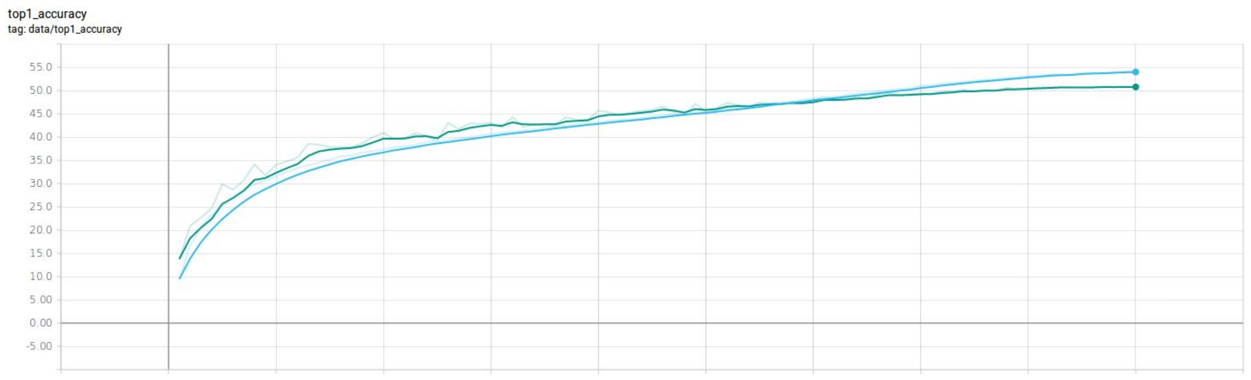
The performance comparison of the attempt models after 90 epochs of training is listed in table.

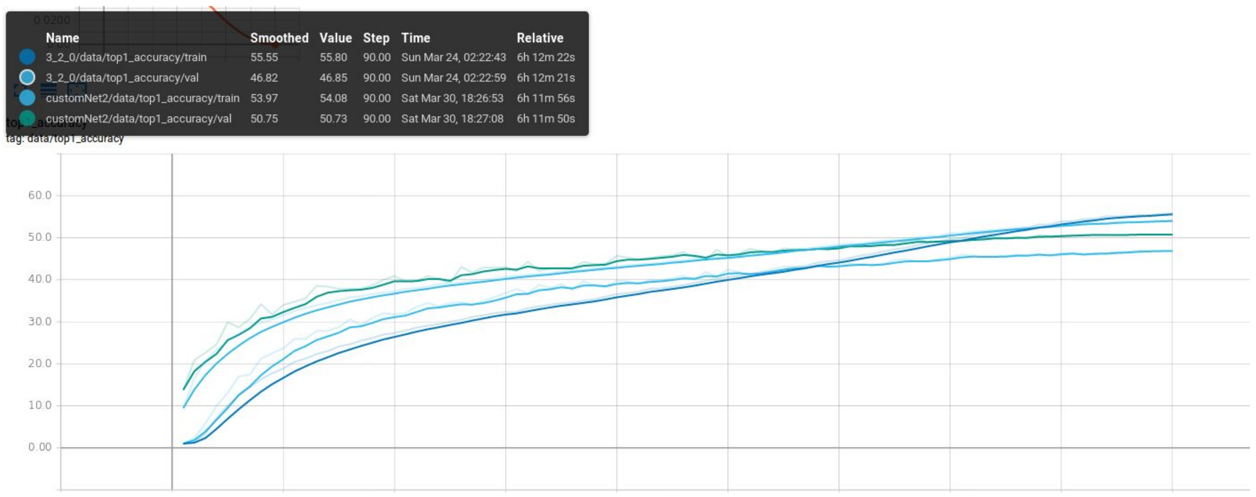
Model	Top 1 accuracy train	Top 1 accuracy val	Top 5 accuracy train	Top 5 accuracy val	Training Loss
SimpleNet	55.80	46.85	82.77	75.50	1.733
ResNet	40.56	43.97	71.06	74.66	2.166
ResNet lr = 0.01	32.91	36.54	63.52	68.00	2.710
GoogleNet	32.43	35.99	62.79	67.06	2.766
CustomNet	60.47	11.38	86.16	33.30	1.489
CustomNet-2	**54.08**	**50.73**	**82.09**	**78.88**	**1.639**

1. Learning rate



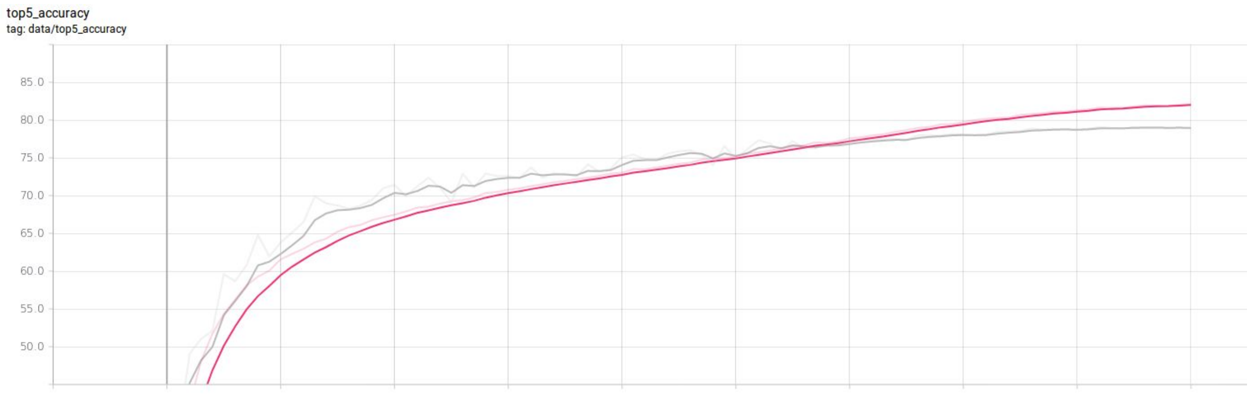
2. Top1 accuray

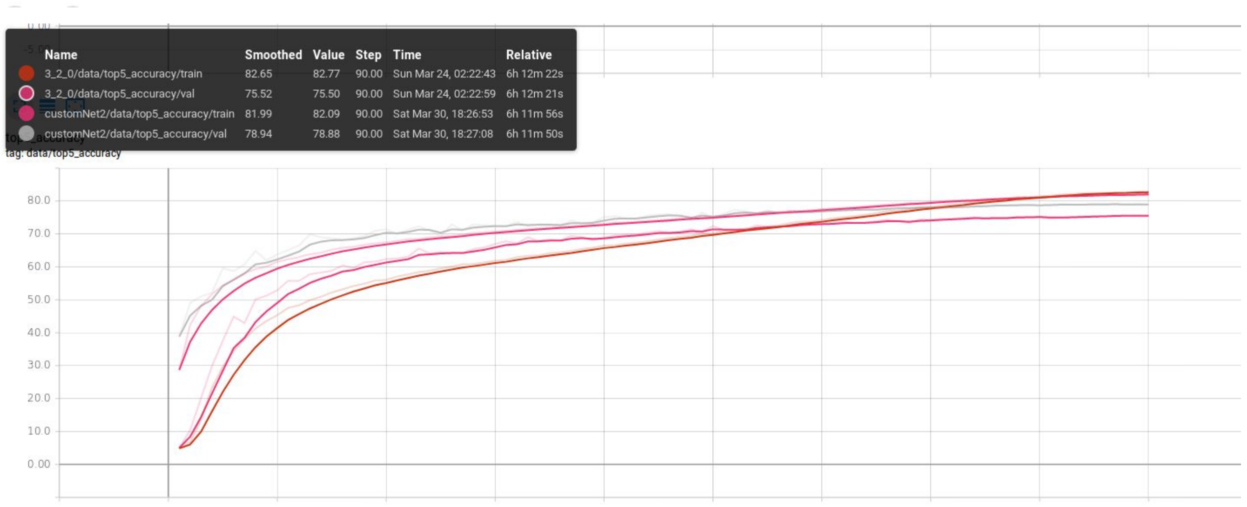




	Name	Smoothed	Value	Step	Time	Relative
●	3_2_0/data/top1_accuracy/train	55.55	55.80	90.00	Sun Mar 24, 02:22:43	6h 12m 22s
●	3_2_0/data/top1_accuracy/val	46.82	46.85	90.00	Sun Mar 24, 02:22:59	6h 12m 21s
●	customNet2/data/top1_accuracy/train	53.97	54.08	90.00	Sat Mar 30, 18:26:53	6h 11m 56s
●	customNet2/data/top1_accuracy/val	50.75	50.73	90.00	Sat Mar 30, 18:27:08	6h 11m 50s

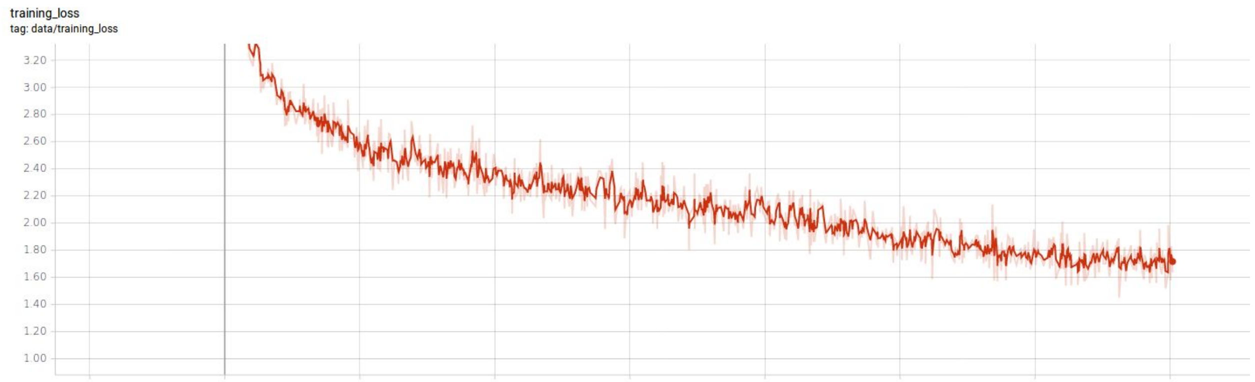
3. Top5 accuray

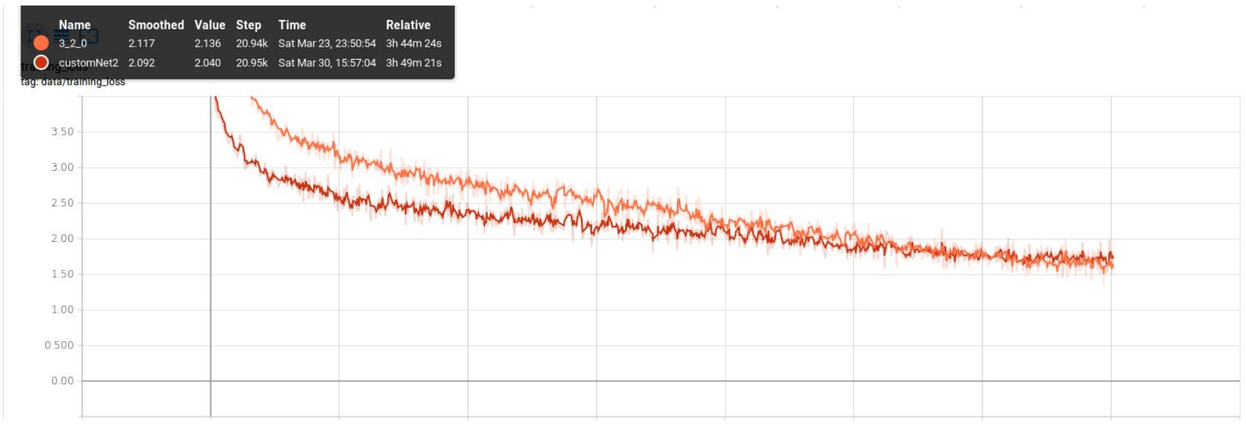




	Name	Smoothed	Value	Step	Time	Relative
●	3_2_0/data/top5_accuracy/train	82.65	82.77	90.00	Sun Mar 24, 02:22:43	6h 12m 22s
●	3_2_0/data/top5_accuracy/val	75.52	75.50	90.00	Sun Mar 24, 02:22:59	6h 12m 21s
●	customNet2/data/top5_accuracy/train	81.99	82.09	90.00	Sat Mar 30, 18:26:53	6h 11m 56s
●	customNet2/data/top5_accuracy/val	78.94	78.88	90.00	Sat Mar 30, 18:27:08	6h 11m 50s

4. Training loss




	Name	Smoothed	Value	Step	Time	Relative
●	3_2_0	2.117	2.136	20.94k	Sat Mar 23, 23:50:54	3h 44m 24s
●	customNet2	2.092	2.040	20.95k	Sat Mar 30, 15:57:04	3h 49m 21s

- [Bonus] MiniPlaces Challenge
  For this part, we put our best model **CustomNet-2** into results folder. Also, its size is 5.5 MB, which is smaller than 10 MB.

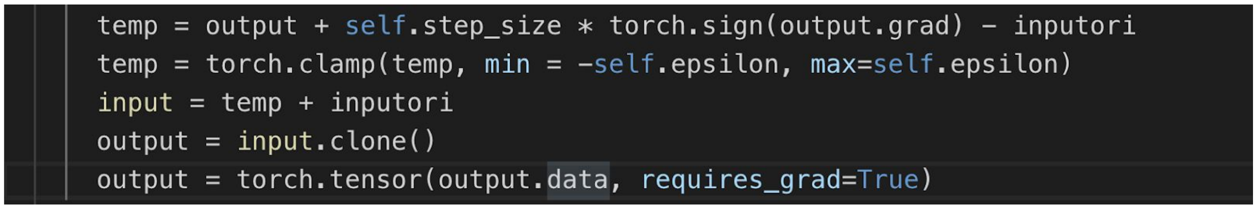
3.3 Attention and Adversarial Samples

- Attention

As described in [1], this is a technique for visualizing the class models. It generates numerically image which is representatives of the class in terms of neural network class scores. The computation of the image-specific saliency map for a single class is extremely quick, since it only requires a single back-propagation pass.

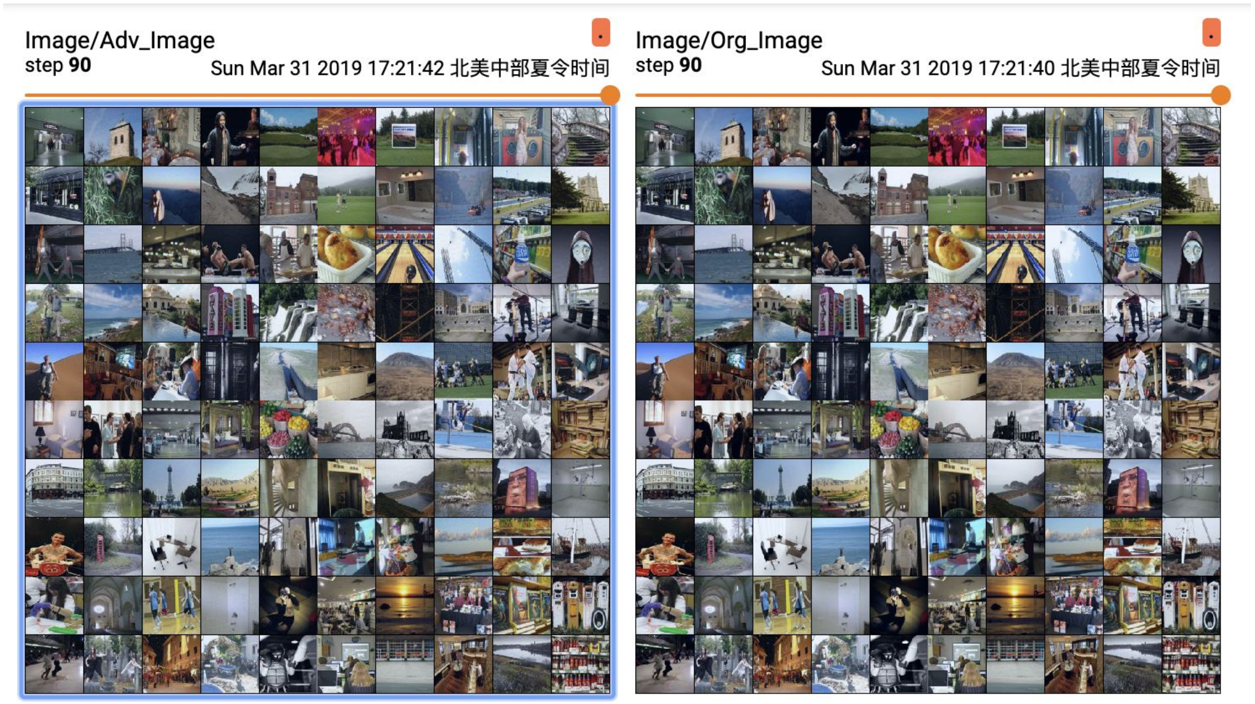
- Adversarial Samples

For this part, we clone a new tensor in the computation graph such that the gradient operation is not recorded in the corresponding tensors. The cloned tensor's **required_grad property needed to set to true** in each iteration such that the gradient can be acquired to conduct corresponding projected gradient descent(PGD) update operation. One small problem we meet in the implementation is that the gradient still can't be acquired even if we set the required_grad to true explicitly. Finally, we get the grad by created a new tensor with required grad property in each iteration.
According to [2], the projected gradient descent can be regarded as multiple FSGM operations. The updated input tensor needed to be projected to the epsilon-nearest neighborhood if it is out of boundary in this dimension**.** So we just implement the as follows:



```
temp = output + self.step_size * torch.sign(output.grad) - inputori
temp = torch.clamp(temp, min = -self.epsilon, max=self.epsilon)
input = temp + inputori
output = input.clone()
output = torch.tensor(output.data, requires_grad=True)
```

The key operation here in PGD is that we compare the updated image with the original image and see **whether the disturb is inside the epsilon ball, if not we cast the corresponding disturbance to this scale and add to the updated tensor.**
The adversarial image and original image looks almost the same and you can't identify the difference with human eyes. We have pasted the tensorboard display results as follows:


Image/Adv_Image
step 90
Sun Mar 31 2019 17:21:42 北美中部夏令时间
Image/Org_Image
step 90
Sun Mar 31 2019 17:21:40 北美中部夏令时间

We compare the results all at SimpleNet's 87 epch model.

Originally the accuracy for top 1 is 47.65%, for top5 is 76.41%. After applying the adversarial example, the accuracy becomes 0.09% for top1, and 7.61% for top5.

Please clearly identify the contribution of all the team members.

Name	Contribution
Wen-Fu Lee	1. For 3.1, finished coding for **forward** and **backward** functions in student_code.py, and passed the testing code 2. For 3.2 section 0, finished training **SimpleNet** in student_code.py 3. For 3.2 section 2, finished training SimpleNet **using our own convolution** in student_code.py 4. For 3.2 section 3, tried to build up a new CNN and tested ResNet and GoogleNet 5. Built up a github repo for teammates to submit their codes 6. Shared experience about how to set up a GCP with teammates 7. Finished report 3.1 and 3.2 section 0-2
Huawei Wang	1. Finish section 3.3 attention map, finish corresponding part in assignment report. 2. Finish section 3.3 adversarial samples, read reference articles and implement the project gradient descent. Discuss the respective results in the report.
Shuoxuan Dong	1. For 3.2 section 3, build a new CNN based on the simpleNet architecture with random dropout and batch normalizations in each block.

Reference

[1] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In ICLR, 2014.
[2] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In ICLR, 2018.