Date: 24/2/2021

RESULTS of Different Models

Note: The following table accuracies are wrong but they follow the same trend. It can be assumed the accuracies are for e=4/255

Model - VGG19bn	**Test Accuracy**	**FGSM Accuracy (e=8/255)**
Standard model	89.19	32.09
Batchout at last layer (n_2)	86.98	34
Batchout at middle layer (n_1)	86.79	37.81
Batchout at middle layer (n_2)	89.89	45.48
Batchout at middle layer (n_3)	89.52	48.22
Batchout at alternative layers (n_1)	89.69	40.90
Batchout at alternative layers (n_2)	87.91	54.25
Batchout at alternative layers (n_3)	86.73	**69.09**

The best model I obtained has an FGSM accuracy of 69%. The training details and other info can be found in later half. There are four types of models that have been trained:

- Standard model: VGG19 with BN with the difference that ReLU comes before BN unlike the paper implementation
- Single Layer Batchout: Batchout is applied at only one layer, either the last layer or the middle layer. The value of parameter n experimented were [0.1, 0.2, 0.3, 0.4, 0.5]. Middle layer here refers to the layer which gave the best transferability result in the paper. (More on this in subsequent sections)
- Multiple Layer Batchout: Batchout is applied to various *alternative layers*. The reason why I wanted to do alternative layers was just to set a benchmark when I later do it for all layer batchout. Here when n=0.3, the best robust accuracy on FGSM gave 69.09% which is a very good improvement over previous results.

Whenever batchout is mentioned, it refers to the modified batchout and not the batchout implemention from the original paper. Also n_3 refers to n=0.3, where n is the batchout parameter.

The following sections contain information such as training graphs, training parameters which may not be necessary to look into but can be helpful for conducting future experiments. Code and weights can be found at ~/robust/vgg19/

Effect of BatchOut depends on depth of layer

The paper [Feature space perturbation yields more transferable adversarial examples](), uses the l2 distance of features of original image and a target image as loss function and then perturbs the original image using this loss function instead of the usual loss function. As per the paper, there are layers which yield better transferable features. In the following figure x-axis is the depth of the network. We can see a layer in the middle gives better results, than the last layers. This made me feel that batchout will have more effect on certain layers. The y-axis is % of successful attacks and the four figures are in different situations like normal attack, targeted attack.

'VGG19' has the following filters:

[64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', **512**, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'] where 'M' denotes max pooling. In the paper mentioned above, the layer marked **bold** gave the best results and batchout was applied after this layer. (The first two layers were not considered in the paper)

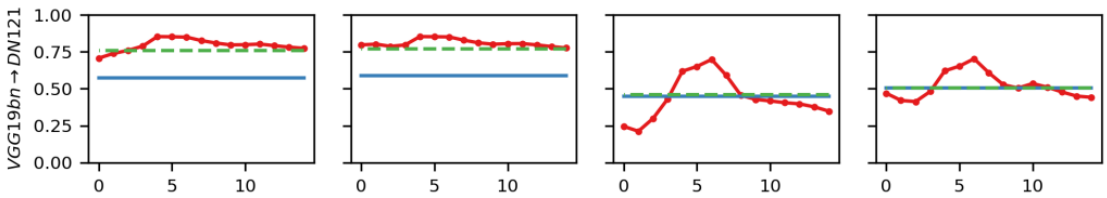
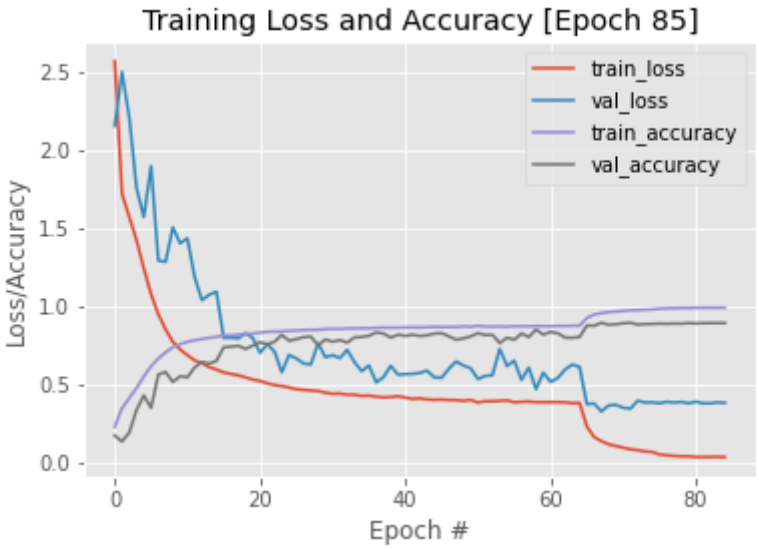


Figure 3: Error, uTR, tSuc, and tTR rates versus depth for multiple transfer scenarios. The top two rows are transfers from a DN121 whitebox model and the bottom two rows are transfers from a VGG19bn whitebox model. Dataset: CIFAR-10.

My batchout experiments included:

1. Standard Training:



Setting: weight_decay: 5e-4. starting lr=1e-2 with drops on epoch 60 and 75.

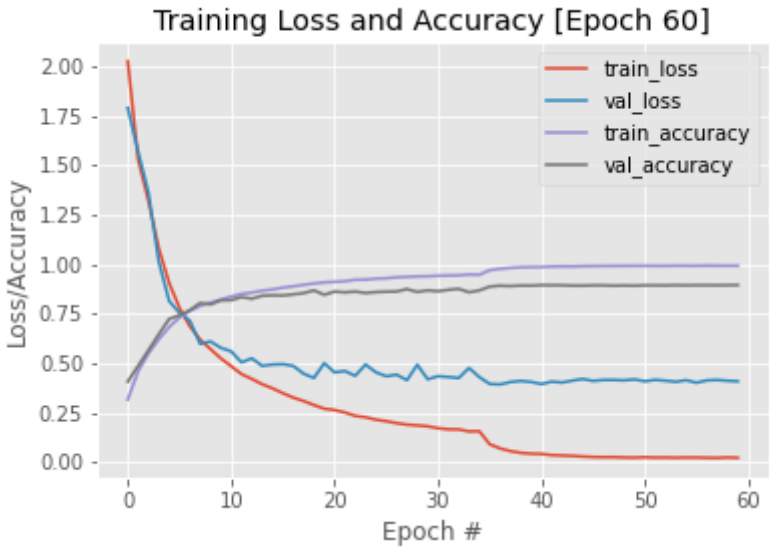
Accuracy: *89.19%*.

2. Batchout on Final layer

Setting: weight_decay: 6e-4 (1e-4 was added because I forgot to change code). The training graph will be uploaded later. `n = 0.2`

Accuracy: *86.98%*

3. Batchout on a middle layer:

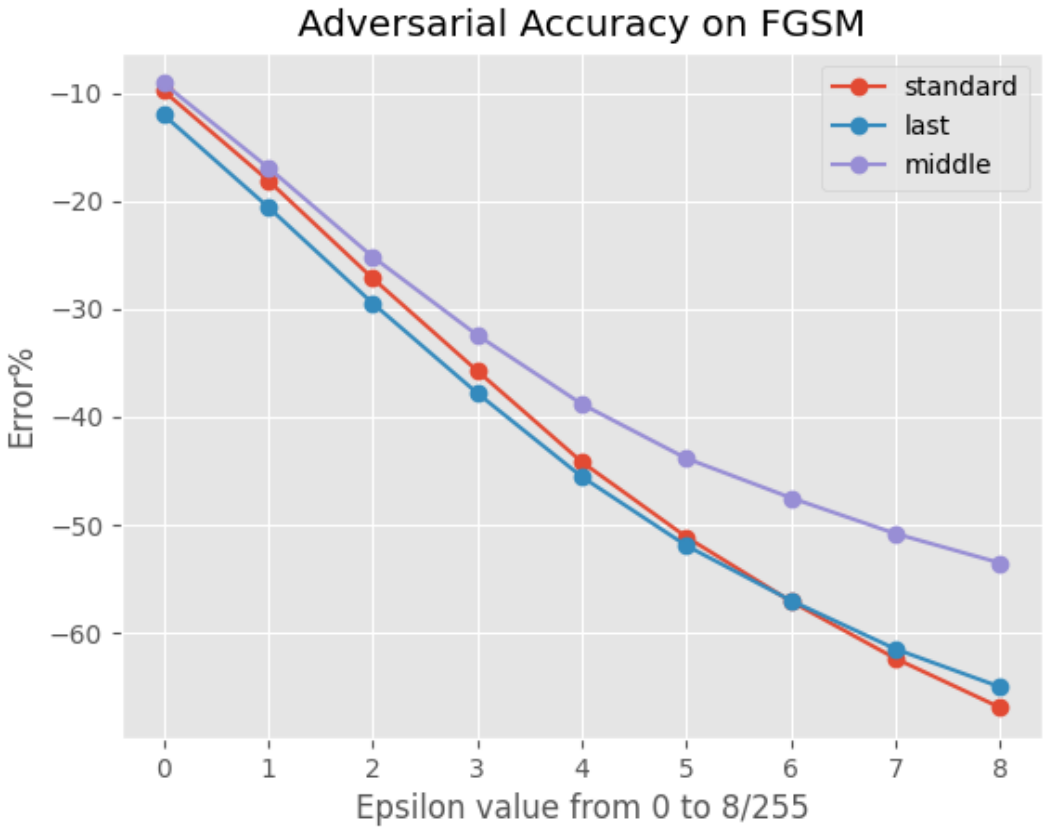


Setting: weight_decay: 6e-4. starting learning rate=1e-2 dropped at 40 and 55 epochs. `n=0.2`

Accuracy: *89.89%*

Robust Accuracies:

Applying batchout on a layer which gave good results to the paper mentioned above, gave better robust accuracies as well.



Conclusion: There are certain layers in a model that can give better effect on batchout than the last layers.

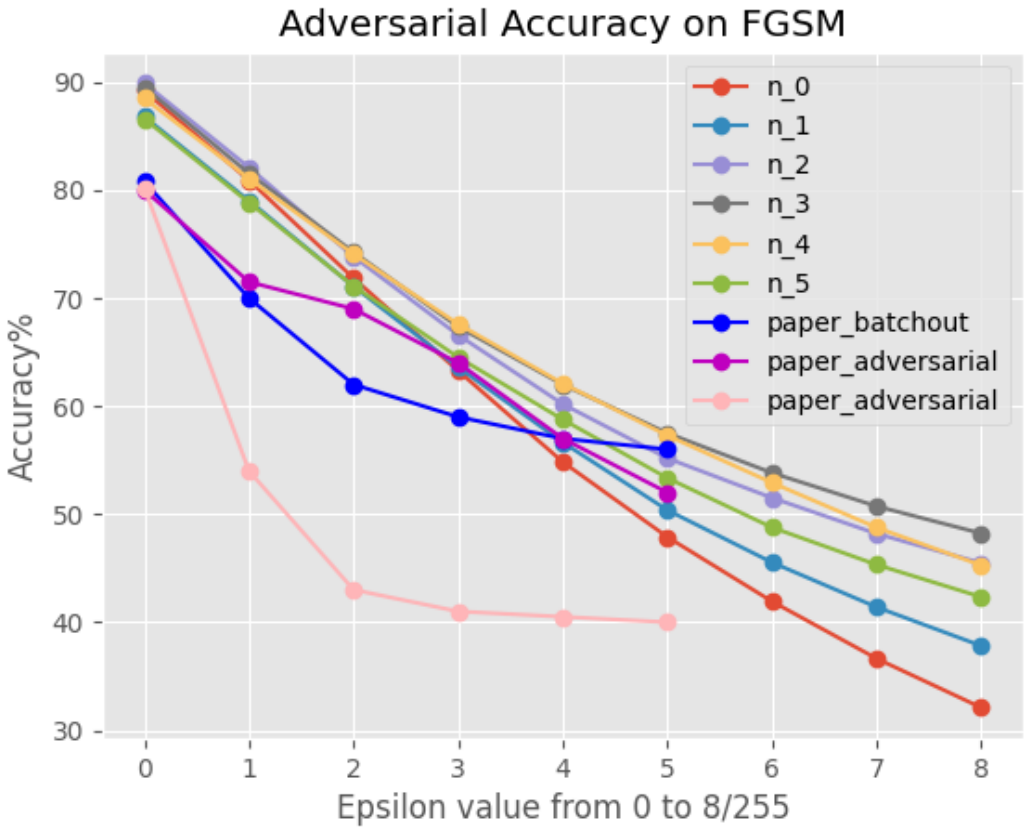
Effect of n on robustness

Batchout update equation: `x = original_feature + n * (added_feature - original_feature)`

```
x = (1 - n) * original_feature + n * added_feature
```

Clearly n cannot be greater than 0.5, otherwise the added_feature is going to dominate giving wrong results. n = 0 refers to standard training.

Previously, n=0.3 was used on ResNet and it gave also gave good results on batchout. In the batchout paper, n = U[0, 0.15] was frequently used where U is uniform distribution. I don't know why I used such a large value of n=0.3 and thus reduced it to n=0.2 for this experiments on VGG. I experiment with different values of n to check which is giving good results.

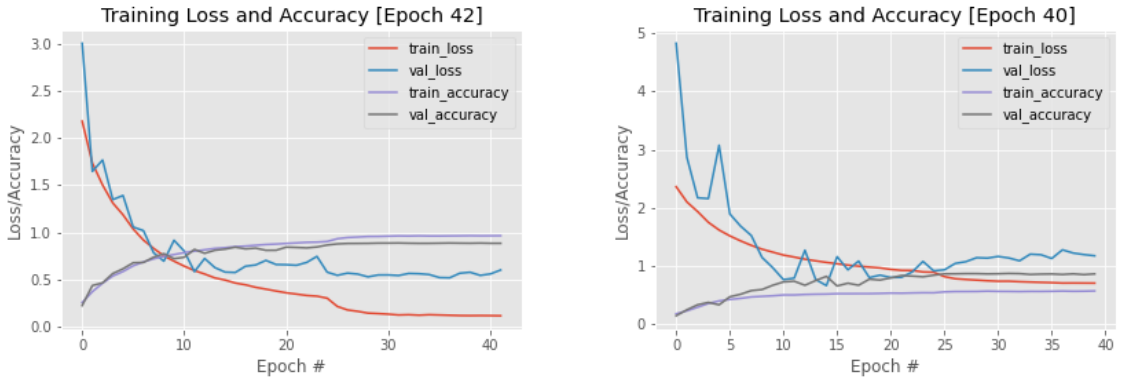


It can be seen that for `n=0.3`, we get the **best results**. Also we can see the results comparison with the paper. The difference between paper and my implementation are:

1. Different architectures (mine is bigger - vgg19)
2. Paper uses batchout at various locations but I used it at one location.

A particularly interesting case is when n=0.5. When n=0.5, the batchout equation turns into

```
x = original_feature + n * (added_feature - original_feature)
# The above equation converts to:
x = 0.5 * (original_feature + added_feature) # i.e the average of both
```



Training when n=0.4 (left) and n=0.5 (right) .

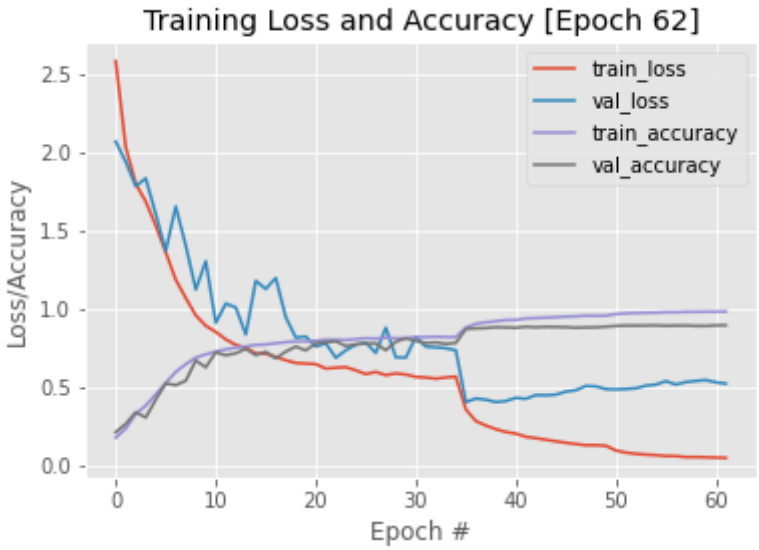
The model is clearly overfitting with n=0.5 (The training was done in the proper way of trying to reduce overfitting). Infact, the training accuracy of perturbed images is just 57.37 % but the test accuracy is 86.6%. Surprisingly, model trained with n=0.5 is more robust than standard trained model and n=0.1 trained model even though it has poor training accuracy. When trained with n=0.6, the best result obtained was 60% training accuracy and 75% test accuracy, with clear overfitting.

Batchout on Alternative Layers

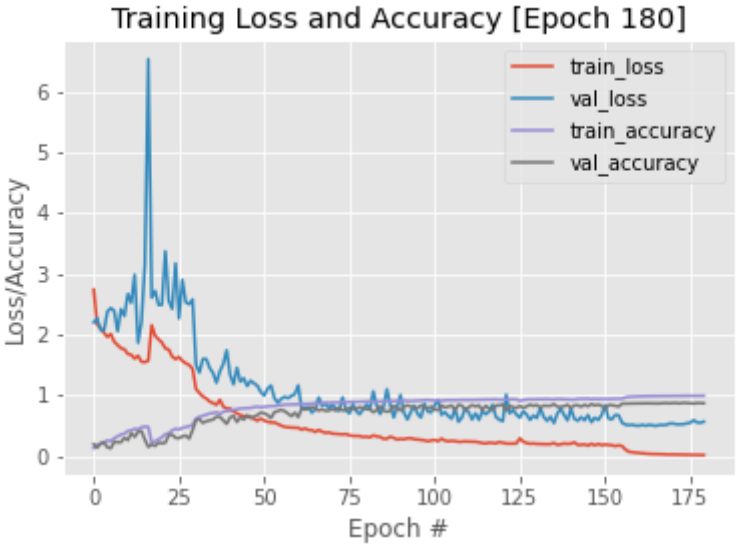
In the below experiments, there is a slight difference from the actual paper implementation. Here I ensure that the features of the same datapoint from the batch are added in subsequent batchout layers. However this datapoint is chosen randomly.

```
    def forward(self, x, y):
         x = self.features1(x) #0
         x = self.features2(x) #1
         x = self.features3(x) #2 R
         x = self.features4(x) #3 0
         x = self.features5(x) #4 1
         x = self.features6(x) #5 R
         x = self.features7(x) #6 2
         x = self.features8(x) #7 3
         x = self.features8(x) #8 4
         x = self.features8(x) #9 5
         x = self.features9(x) #10 R

         x = self.features10(x) #11 6

         if self.training:
             x, r = self.batchout1(x, y)

         x = self.features11(x) #12
         x = self.features11(x) #13

         if self.training:
             x, _ = self.batchout1(x, y, r)

         x = self.features11(x) #14
         x = self.features12(x) #15 R

         if self.training:
             x, _ = self.batchout1(x, y, r)

         x = self.features11(x) #16
         x = self.features11(x) #17

         if self.training:
             x, _ = self.batchout1(x, y, r)

         x = self.features11(x) #18
         x = self.features11(x) #19

         if self.training:
             x, _ = self.batchout1(x, y, r)

         x = self.features12(x) #20 R
         x = x.view(x.size(0), -1)
         x = self.classifier(x)

         return x
```

- For `n=0.1`, the model is trained similar to standard model. 89.7% test accuracy. {initial lr=0.1, decay=5e-4, momentum=0.9}. Robust error = 59.10% {e=8/255 - FGSM} which is actually less. But the training was completed easily just like standard training.

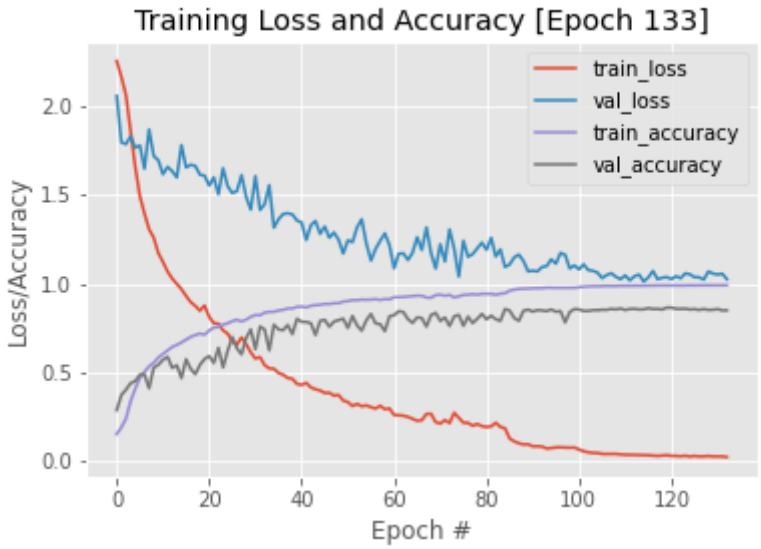


- For `n=0.2` the model couldn't be trained. There was clear overfitting and the best accuracy I achieved was 60%. But then I realised that I was decreasing the learning rates too quickly. (This was probably the same reason why I didn't get the required results when implementing the batchout paper.)

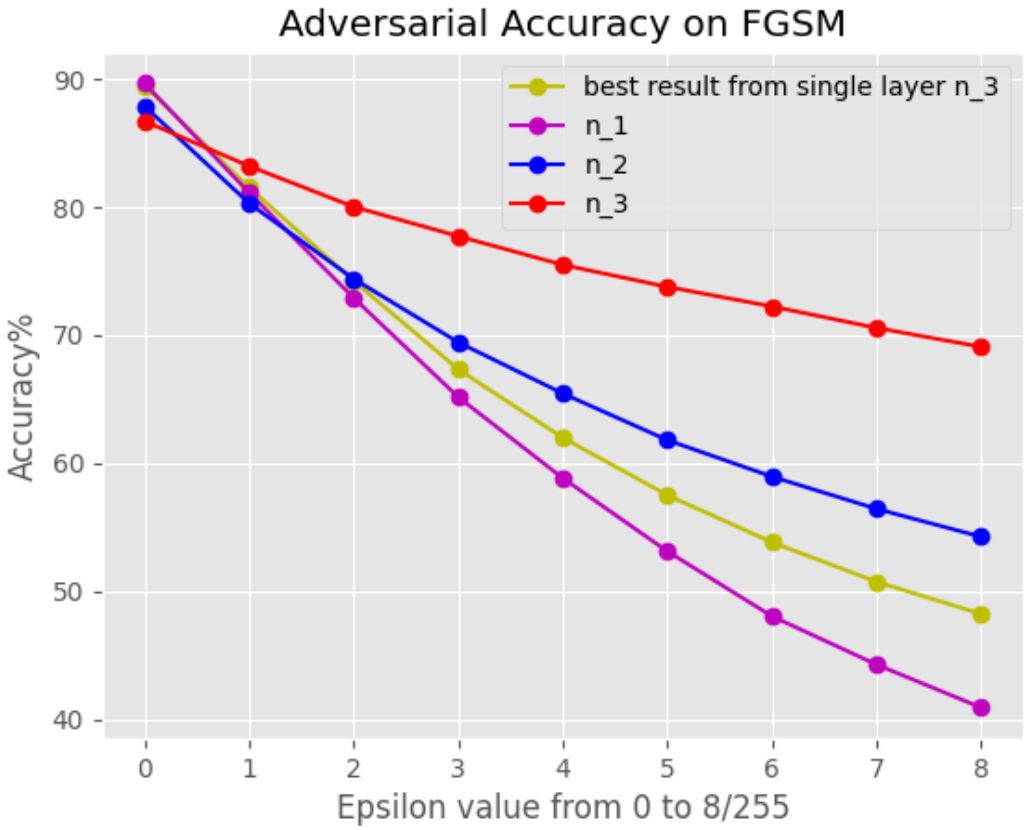


  {initial lr=0.1 (droped only till 1e-3 but don't remember exactly when), decay=5e-4, momentum=0.9}. Accuracy=87.91%. Robust error =45.75%. This is a very good improvement. But the training required almost 3x more epochs

- For `n=0.3` initially about 2-3 times the training process didn't give proper results. Observing high overfitting, I increased the regularization from 5e-4 to 1e-3. Also the momentum was changed from 0.9 to 0.5 and later when learning rate was changed to 1e-4, it was increased to 0.9.



Accuracy: 86.73%, **Robust error=30.89%. This is a very good improvement.** For comparison of above results and the best result obtained by me in past, the following figure will be helpful:



Conclusions and Thoughts

1. For `n=0.3` I obtained the best results.
2. Applying batchout to multiple layers and especially with large values such as n=0.2 or n=0.3 made the training to complete in more epochs but gave better results as well.
3. Evaluation on AutoAttack would give a good comparision but due to some strange errors I couldn't. I will look into the errors.
4. The intuition of batchout is that we are perturbing the features of a class in the direction of another class. An experiment can be performed where batchout is only applied to a pair of

classes and evaluation can be done with targeted attacks for the chosen class and other classes. Example: In cifar, there are 10 classes. Let us chose bird and deer. We apply batchout only to datapoints of bird and the perturbed feature will always be deer. Intuitively, we are perturbing the features of bird class in direction of deer class. After training, if we perform targeted attack on deer class, the bird class should give better results than the other classes(frog, automobile, dog etc).

5. Further experiments can include applying batchout to all layers or trying it with ResNet which is more popular as there are more instances of it in AutoAttack leaderboard.