```
17/3
```

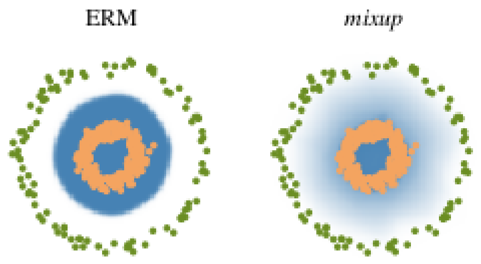
Few Papers related to Batchout

Mixup: BEYOND EMPIRICAL RISK MINIMIZATION (ICLR - 2018)

Mixup is quite similar to batchout. It is a form of regularization (data augmentation. Mixup trains a neural network on convex combinations of pairs of examples and their labels. The paper claims to improve generalization, AR, GAN training stability and reduces the memorization of corrupt labels.

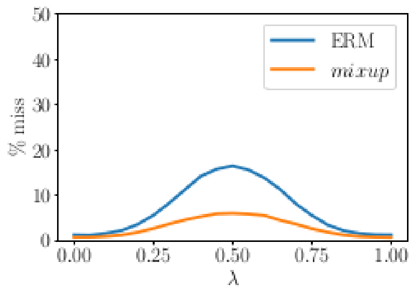
$$\tilde{x} = \lambda x_i + (1-\lambda)x_j$$
$$\tilde{y} = \lambda y_i + (1-\lambda)y_j$$

The major difference is that even the linear combination of 'y' is also done and this is only done in the input space. The following figure is related to a toy example. We see a **smooth transition**, light blue indicates lesser probability.

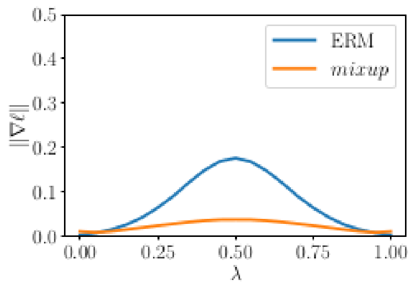


(b) Effect of *mixup* ($\alpha = 1$) on a toy problem. Green: Class 0. Orange: Class 1. Blue shading indicates $p(y=1|x)$.

The following figures were obtained on Cifar10. The figure on left shows an improvement in prediction(prediction refers to the predicted class). As lambda moves from 0(original image) to 1(selected image), the %miss (where a "miss" refers to a prediction when the class does not belong to both the original or selected image) decreases. For me, the presences of a miss itself is surprising because how can other classes come in between the convex combination of two classes? It's possible nonetheless because deep-nets are piecewise linear (with relatively few pieces).



(a) Prediction errors in-between training data. Evaluated at $x = \lambda x_i + (1-\lambda)x_j$, a prediction is counted as a "miss" if it does not belong to $\{y_i, y_j\}$. The model trained with *mixup* has fewer misses.



(b) Norm of the gradients of the model w.r.t. input in-between training data, evaluated at $x = \lambda x_i + (1-\lambda)x_j$. The model trained with *mixup* has smaller gradient norms.

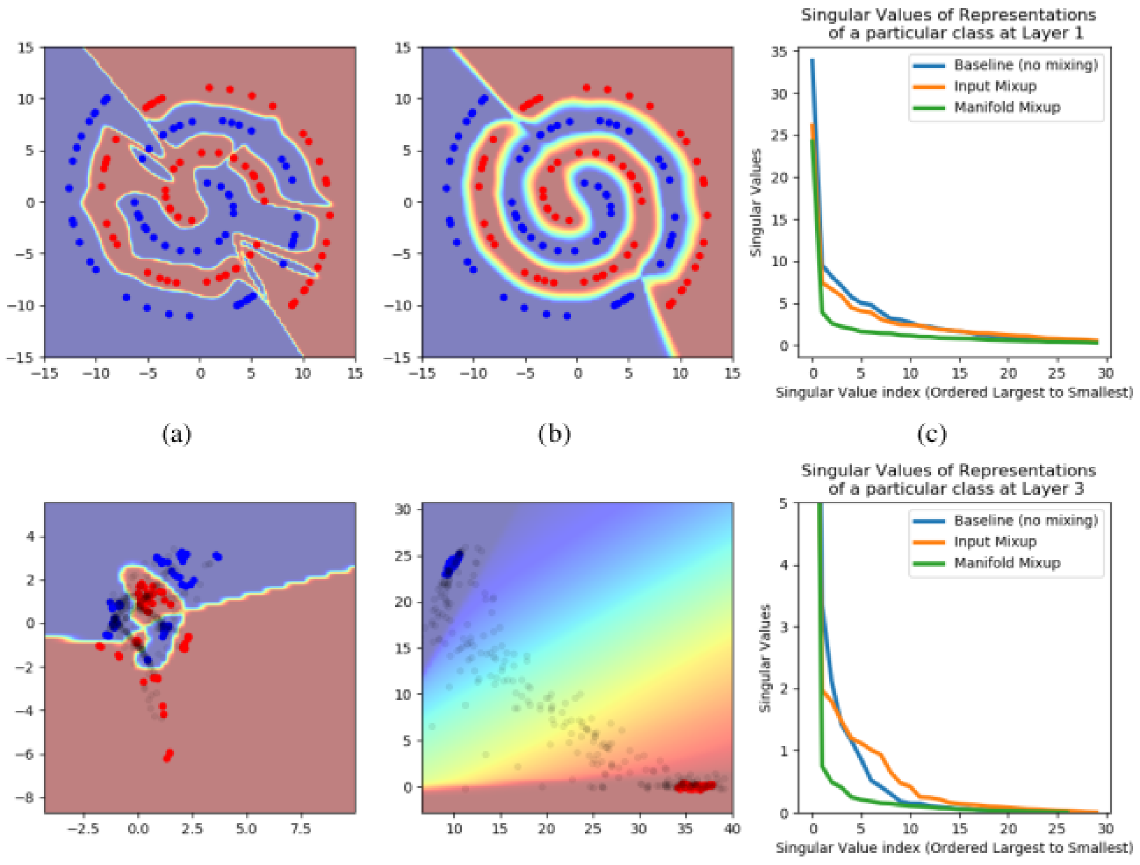
The figure on right talks about the gradient magnitude w.r.t image for differnet lambda. The smaller the norms, the better it is for adversarial robustness.

ManiFold Mixup Better Representations by interpolating Hidden States(ICML 2019)

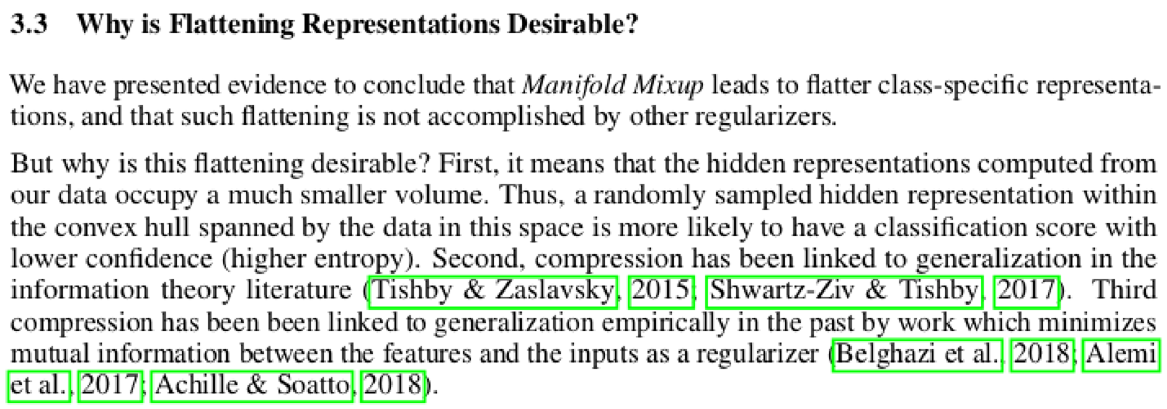
This paper is very much similar to the above paper, except that it also considers assigning mixup in the feature space. Paper claims: Manifold Mixup improves the hidden representations and decision boundaries of neural networks at multiple layers, improves strong baselines in supervised learning, robustness
to **single-step adversarial attacks**. Here they have specifically mentioned robustness to single-step AA such as FGSM, I-FGSM

They apply mixup to only one layer and this layer is choosen randomly (includes input layer as well).

On a toy dataset, manifold mixup does the following:



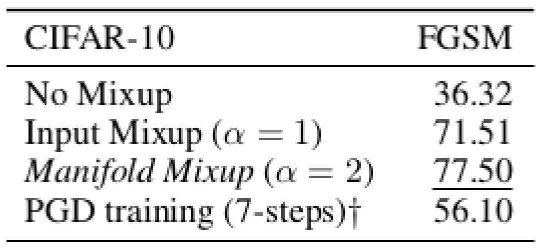
Paper claims that Manifold Mixup Flattens Representations. More specifically, this flattening reduces the number of directions with significant variance (akin to reducing their number of principal components). A lot of Maths and proofs are shown regarding flattening. The concept of Flattening was actually not explained properly in the paper, this was also pointed out in OpenReview.



3.3 Why is Flattening Representations Desirable?

We have presented evidence to conclude that *Manifold Mixup* leads to flatter class-specific representations, and that such flattening is not accomplished by other regularizers.

But why is this flattening desirable? First, it means that the hidden representations computed from our data occupy a much smaller volume. Thus, a randomly sampled hidden representation within the convex hull spanned by the data in this space is more likely to have a classification score with lower confidence (higher entropy). Second, compression has been linked to generalization in the information theory literature (Tishby & Zaslavsky, 2015; Shwartz-Ziv & Tishby, 2017). Third compression has been been linked to generalization empirically in the past by work which minimizes mutual information between the features and the inputs as a regularizer (Belghazi et al., 2018; Alemi et al., 2017; Achille & Soatto, 2018).



CIFAR-10	FGSM
No Mixup	36.32
Input Mixup ($\alpha = 1$)	71.51
Manifold Mixup ($\alpha = 2$)	$\underline{77.50}$
PGD training (7-steps)†	56.10

It is seen that by perturbing in the feature space, there is some improvement. Also surprising is that it has surpassed Madry et Al(PGD training) on FGSM accuracy. This is what is written in paper:

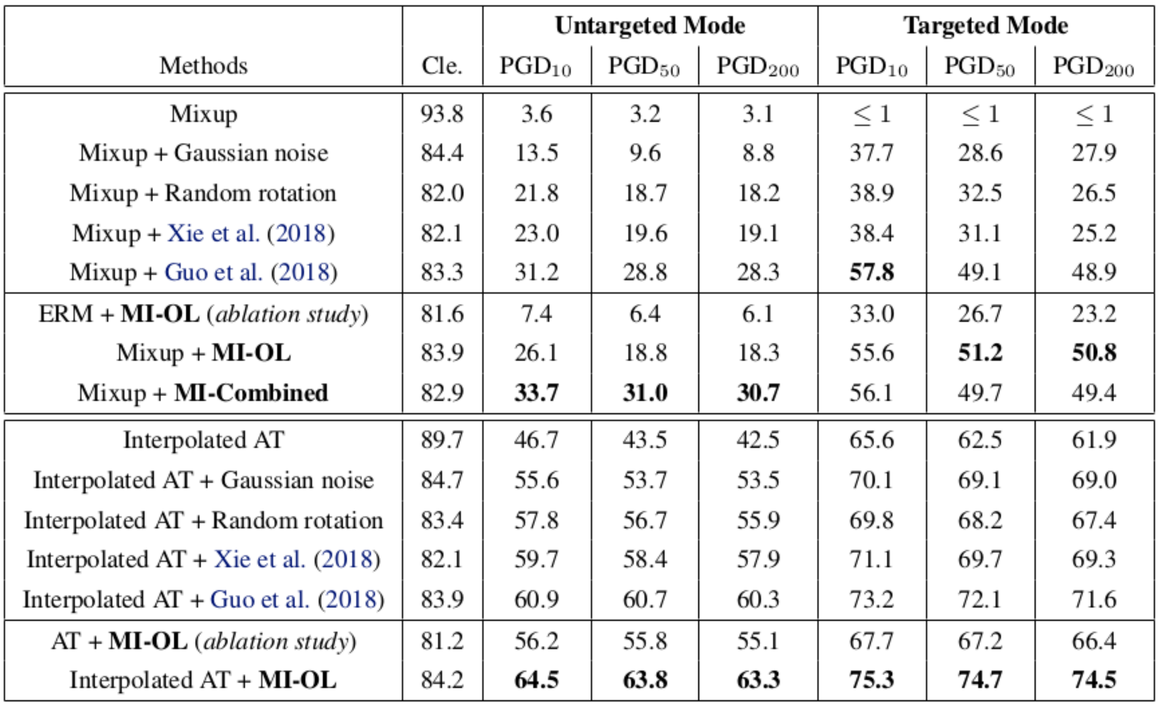
While we found that using Manifold Mixup improves the robustness to single-step FGSM attack (especially over Input Mixup), we found that Manifold Mixup did not significantly improve robustness against stronger,
multi-step attacks such as PGD

I have seen in various places that some models which have very good accuracy on FGSM might fail in PGD, for example a model had 90% FGSM accuracy but 0 % PGD whereas in contrast Madry et Al model has 56% FGSM accuracy and around 44% PGD. Another thing to note from here is, in my batchout experiments I achieved accuracy of 52% on FGSM and around 2-3% on PGD (standard model-0%).

A lot of questions where asked in openreview and might give better explaination of what is flattening, manifolds etc.

MIXUP INFERENCE (ICLR 2020)

This paper applies mixup in inference phase (Called MI) and claims that MI can further improve the adversarial robustness for the models trained by mixup and its variants. If we want to classify a image x on classifier f, we do f(x) but in this method we need to do f(x, a) where 'a' is a batch of images and 'b' corresponding class. Then mixup is applied between x and 'a' with a suitable value of 'alpha' the convex combination parameter. Average of the results are taken as prediction. The paper is short but the paper was slighlty difficult to read because of excessive use of symbols. The good thing about the method is that it can be used on any model.



Methods	Cle.	Untargeted Mode PGD$_{10}$	Untargeted Mode PGD$_{50}$	Untargeted Mode PGD$_{200}$	Targeted Mode PGD$_{10}$	Targeted Mode PGD$_{50}$	Targeted Mode PGD$_{200}$
Mixup	93.8	3.6	3.2	3.1	≤ 1	≤ 1	≤ 1
Mixup + Gaussian noise	84.4	13.5	9.6	8.8	37.7	28.6	27.9
Mixup + Random rotation	82.0	21.8	18.7	18.2	38.9	32.5	26.5
Mixup + Xie et al. (2018)	82.1	23.0	19.6	19.1	38.4	31.1	25.2
Mixup + Guo et al. (2018)	83.3	31.2	28.8	28.3	**57.8**	49.1	48.9
ERM + **MI-OL** (*ablation study*)	81.6	7.4	6.4	6.1	33.0	26.7	23.2
Mixup + **MI-OL**	83.9	26.1	18.8	18.3	55.6	**51.2**	**50.8**
Mixup + **MI-Combined**	82.9	**33.7**	**31.0**	**30.7**	56.1	49.7	49.4
Interpolated AT	89.7	46.7	43.5	42.5	65.6	62.5	61.9
Interpolated AT + Gaussian noise	84.7	55.6	53.7	53.5	70.1	69.1	69.0
Interpolated AT + Random rotation	83.4	57.8	56.7	55.9	69.8	68.2	67.4
Interpolated AT + Xie et al. (2018)	82.1	59.7	58.4	57.9	71.1	69.7	69.3
Interpolated AT + Guo et al. (2018)	83.9	60.9	60.7	60.3	73.2	72.1	71.6
AT + **MI-OL** (*ablation study*)	81.2	56.2	55.8	55.1	67.7	67.2	66.4
Interpolated AT + **MI-OL**	84.2	**64.5**	**63.8**	**63.3**	**75.3**	**74.7**	**74.5**

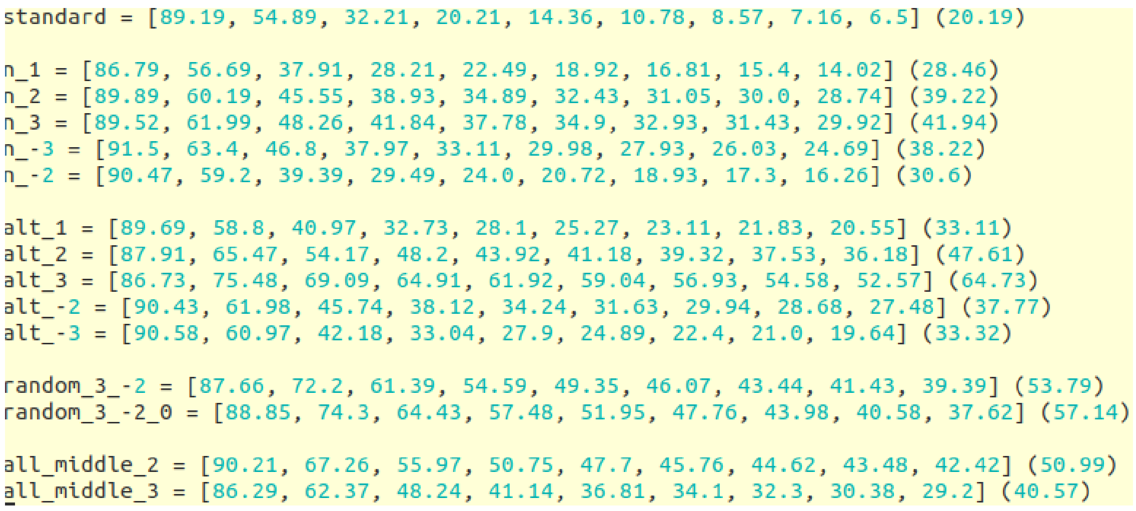
Interpolated here refers to a method that was proposed by the authors Manifold Mixup. I haven't read it but it looks similar to mixup. Also this link should be checked that gives a list of interpolation related papers. There are various other papers on mixup as well that I found by checking the result of mixup in OpenReview.

Batchout

Batchout is slightly different with mixup. We only take convex combination w.r.t images only. Intuitively in my opinion both are acceptable. Mixup tries to make smoother transition. Batchout tries to push the decision boundary away. I am confident that by training on ResNet, the accuracy reported in Mixup(71%) and and perhaps Manifold Mixup(77%) can be achieved (Currently I have 53% with VGG).

Experiements Conducted

1. Usually value of n choosen was from 0.2 or 0.3. This time I applied negative values of -0.2 and -0.3 and the results were surprisingly good but that as good as 0.2 or 0.3. Why did I apply negative values? It was because of some misunderstanding why I was going through TRADES paper. Below we can see alt-*2 and alt*-3 gave 27% and 19% respectively, better than the standard model 6.5% at epsilon 8/255. The values in "[]" refers to accuracies from 0 to 8/255 epsilon and value in "()" is accuracy at l2 FGSM attack.



```
standard = [89.19, 54.89, 32.21, 20.21, 14.36, 10.78, 8.57, 7.16, 6.5] (20.19)

n_1 = [86.79, 56.69, 37.91, 28.21, 22.49, 18.92, 16.81, 15.4, 14.02] (28.46)
n_2 = [89.89, 60.19, 45.55, 38.93, 34.89, 32.43, 31.05, 30.0, 28.74] (39.22)
n_3 = [89.52, 61.99, 48.26, 41.84, 37.78, 34.9, 32.93, 31.43, 29.92] (41.94)
n_-3 = [91.5, 63.4, 46.8, 37.97, 33.11, 29.98, 27.93, 26.03, 24.69] (38.22)
n_-2 = [90.47, 59.2, 39.39, 29.49, 24.0, 20.72, 18.93, 17.3, 16.26] (30.6)

alt_1 = [89.69, 58.8, 40.97, 32.73, 28.1, 25.27, 23.11, 21.83, 20.55] (33.11)
alt_2 = [87.91, 65.47, 54.17, 48.2, 43.92, 41.18, 39.32, 37.53, 36.18] (47.61)
alt_3 = [86.73, 75.48, 69.09, 64.91, 61.92, 59.04, 56.93, 54.58, 52.57] (64.73)
alt_-2 = [90.43, 61.98, 45.74, 38.12, 34.24, 31.63, 29.94, 28.68, 27.48] (37.77)
alt_-3 = [90.58, 60.97, 42.18, 33.04, 27.9, 24.89, 22.4, 21.0, 19.64] (33.32)

random_3_-2 = [87.66, 72.2, 61.39, 54.59, 49.35, 46.07, 43.44, 41.43, 39.39] (53.79)
random_3_-2_0 = [88.85, 74.3, 64.43, 57.48, 51.95, 47.76, 43.98, 40.58, 37.62] (57.14)

all_middle_2 = [90.21, 67.26, 55.97, 50.75, 47.7, 45.76, 44.62, 43.48, 42.42] (50.99)
all_middle_3 = [86.29, 62.37, 48.24, 41.14, 36.81, 34.1, 32.3, 30.38, 29.2] (40.57)
```

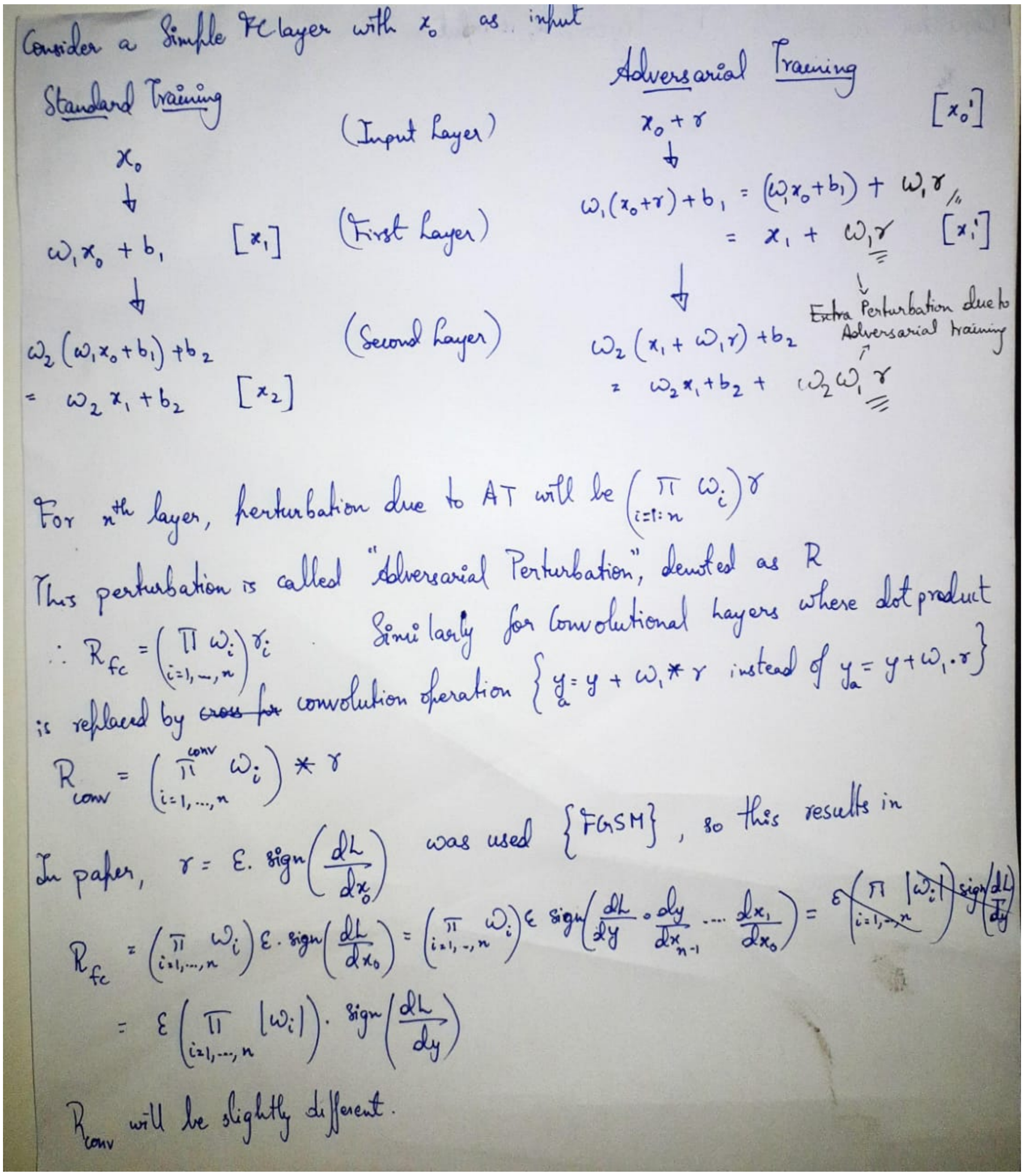
I tried a random combination of +3 and -2, but the results were somewhere in between as can be seen in random_3-*2 and random_3*-2_0. But all these models fail with PGD.

2. I applied the batchout attack specifically on a pair of classes and implemented PGD targeted attack. The accuracy was poor here as well perhaps because our method is not strong with PGD.the
3. The original implementation of batchout did not ensure that the two images that will be perturbed will be of same class. I applied batchout in both the cases, when it is ensured both classes are of same class and when it is not ensured. I see that strangely when it is not ensured, I achieve better accuracy(6% better accuracy). This can give some insight such as: **The linear space between images/features of same class is not necessarily classified as the class?** I tried to check whether mixup ensures both classes are different, but I didn't get any info, will need to look into it.
4. Batchout was applied to all layers after a specific layer. As we can see in above figure, it didn't give the best result and also higher values of n gave poorer result.
5. Early stopping did give good results some time. But perhaps more experiments should be done to check whether this works (This is not related to batchout)

Why I think batchout or Mixup fail with PGD can perhaps be expalined by a paper Tramer et al(2017). I have written more about this later in other file. The gist is that Tramer et al finds around 25 directions in which Adversarial examples exist for MNIST. This number I think will be larger as dataset size increases. But Batchout or Mixup can at max consider 9 direction (each direction in the direction of other class) for CIFAR10. Perhaps this is the main reason why it is giving good (and sometimes better than PGD training) on FGSM but poor on PGD.

Layer-wise Adversarial Training Approach to Improve Adversarial Robustness

AT is designed to only manipulate input. Since intermediate layers also play an important role, layer perturbation is introduced by this method, i.e for each layer there will be a separate perturbation to the input it recieves from the previous layer.



Consider a Simple FC layer with x_0 as input

Standard Training		Adversarial Training
x_0	(Input Layer)	$x_0 + \gamma$ $[x_0']$
$\downarrow$		$\downarrow$
$\omega_1 x_0 + b_1$ $[x_1]$	(First Layer)	$\omega_1(x_0+\gamma)+b_1 = (\omega_1 x_0 + b_1) + \omega_1\gamma$ $= x_1 + \underline{\underline{\omega_1\gamma}}$ $[x_1']$
$\downarrow$		$\downarrow$
$\omega_2(\omega_1 x_0 + b_1) + b_2$ $= \omega_2 x_1 + b_2$ $[x_2]$	(Second Layer)	$\omega_2(x_1 + \omega_1\gamma) + b_2$ $= \omega_2 x_1 + b_2 + \underline{\underline{\omega_2\omega_1\gamma}}$

Extra Perturbation due to Adversarial training

For n^{th} layer, perturbation due to AT will be $\left(\prod_{i=1:n} \omega_i\right)\gamma$

This perturbation is called "Adversarial Perturbation", denoted as R

$\therefore R_{fc} = \left(\prod_{i=1,\dots,n} \omega_i\right)\gamma_i$. Similarly for Convolutional layers where dot product is replaced by ~~cross for~~ convolution operation $\{y_a = y + \omega_1 * \gamma$ instead of $y_a = y + \omega_1 \cdot \gamma\}$

$$R_{conv} = \left(\prod_{i=1,\dots,n}^{conv} \omega_i\right) * \gamma$$

In paper, $\gamma = \varepsilon . \operatorname{sign}\left(\frac{dL}{dx_0}\right)$ was used $\{FGSM\}$, so this results in

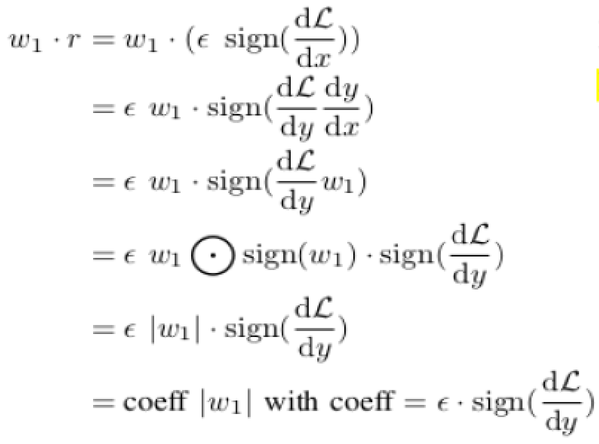
$$R_{fc} = \left(\prod_{i=1,\dots,n} \omega_i\right)\varepsilon . \operatorname{sign}\left(\frac{dL}{dx_0}\right) = \left(\prod_{i=1,\dots,n} \omega_i\right)\varepsilon \operatorname{sign}\left(\frac{dL}{dy} \circ \frac{dy}{dx_{n-1}} \cdots \frac{dx_1}{dx_0}\right) = \cancel{\varepsilon\left(\prod_{i=1,\dots,n} |\omega_i|\right) \operatorname{sign}\left(\frac{dL}{dy}\right)}$$

$$= \varepsilon\left(\prod_{i=1,\dots,n} |\omega_i|\right) . \operatorname{sign}\left(\frac{dL}{dy}\right)$$

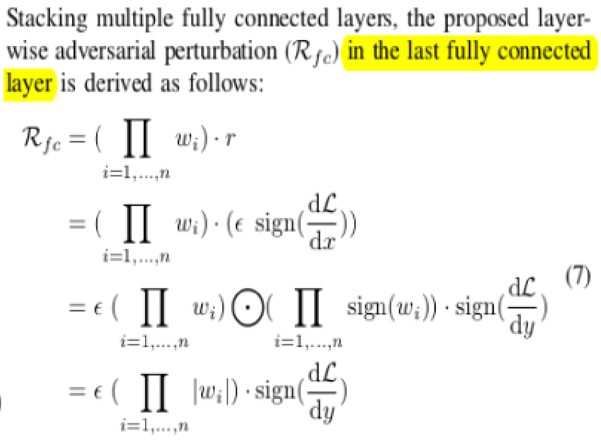
R_{conv} will be slightly different.

R_{conv} will be slightly different because of the presence of the convolution operator.

Few pictures from the paper:



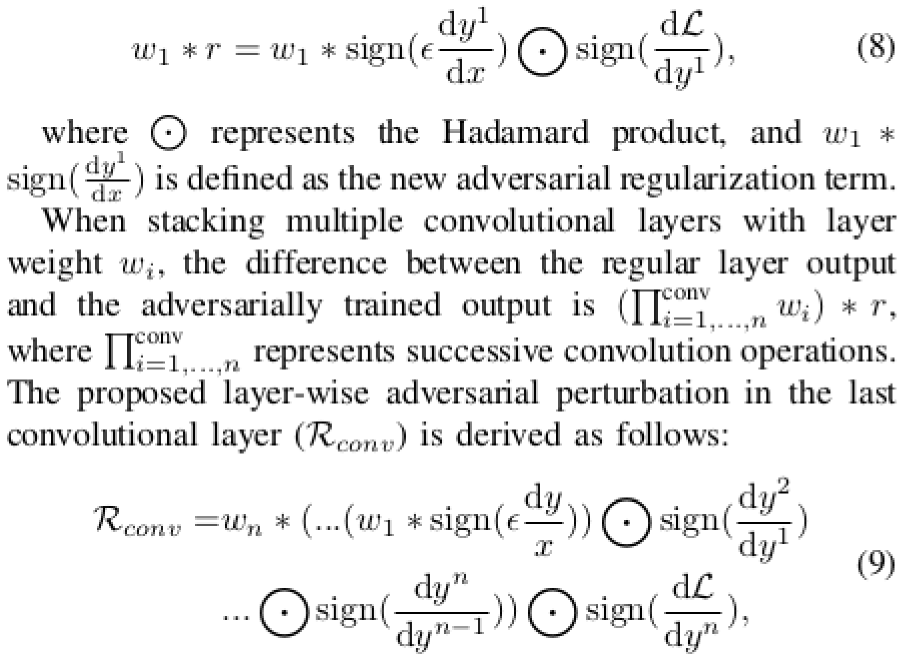
$$\begin{aligned} w_1 \cdot r &= w_1 \cdot (\epsilon \ \operatorname{sign}(\frac{d\mathcal{L}}{dx})) \\ &= \epsilon \ w_1 \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}\frac{dy}{dx}) \\ &= \epsilon \ w_1 \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy} w_1) \\ &= \epsilon \ w_1 \odot \operatorname{sign}(w_1) \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}) \\ &= \epsilon \ |w_1| \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}) \\ &= \text{coeff} \ |w_1| \text{ with coeff} = \epsilon \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}) \end{aligned}$$



Stacking multiple fully connected layers, the proposed layer-wise adversarial perturbation ($\mathcal{R}_{fc}$) in the last fully connected layer is derived as follows:

$$\begin{aligned} \mathcal{R}_{fc} &= (\prod_{i=1,\dots,n} w_i) \cdot r \\ &= (\prod_{i=1,\dots,n} w_i) \cdot (\epsilon \ \operatorname{sign}(\frac{d\mathcal{L}}{dx})) \\ &= \epsilon \ (\prod_{i=1,\dots,n} w_i) \odot (\prod_{i=1,\dots,n} \operatorname{sign}(w_i)) \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}) \\ &= \epsilon \ (\prod_{i=1,\dots,n} |w_i|) \cdot \operatorname{sign}(\frac{d\mathcal{L}}{dy}) \end{aligned} \quad (7)$$

This is for a FC layer. For convolution layer it will be slightly different because we take the convolution operator instead of simple dot product.



$$w_1 * r = w_1 * \text{sign}(\epsilon \frac{\mathrm{d}y^1}{\mathrm{d}x}) \bigodot \text{sign}(\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y^1}), \qquad (8)$$

where $\bigodot$ represents the Hadamard product, and $w_1 * \text{sign}(\frac{\mathrm{d}y^1}{\mathrm{d}x})$ is defined as the new adversarial regularization term.

When stacking multiple convolutional layers with layer weight w_i, the difference between the regular layer output and the adversarially trained output is $(\prod_{i=1,\dots,n}^{\text{conv}} w_i) * r$, where $\prod_{i=1,\dots,n}^{\text{conv}}$ represents successive convolution operations. The proposed layer-wise adversarial perturbation in the last convolutional layer ($\mathcal{R}_{conv}$) is derived as follows:

$$\begin{aligned}\mathcal{R}_{conv} =& w_n * (...(w_1 * \text{sign}(\epsilon \frac{\mathrm{d}y}{x})) \bigodot \text{sign}(\frac{\mathrm{d}y^2}{\mathrm{d}y^1}) \\ & ... \bigodot \text{sign}(\frac{\mathrm{d}y^n}{\mathrm{d}y^{n-1}})) \bigodot \text{sign}(\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y^n}),\end{aligned} \qquad (9)$$

Note that for the 1st layer equation (8) in the above figure is nothing but the perturbation due to FGSM adversary.

Training procedure:

1. A batch of images is sent and forward pass is done. In the forward pass the values R_{fc} and R_{conv} is added for each FC layer and conv layer respectively.
2. Backward pass is done. Here the values of R_{fc} and R_{conv} are updated as well.
3. Process repeats

The results show positive result. The best positive result is when the perturbations are added to all the layers (including the starting layer).

Few points:

1. This is like FGSM training done cumulatively on each layer. Few of the methods that improved FGSM training in the past are Random initializations and methods used in Fast Adversarial training paper.
2. So we are perturbing the features at each layer, this is somewhat similar to batchout. Batchout gave better results when applied to alternative layers, perhaps this method may also give better results when applied at alternative layers.
3. Eventually this method tries to perturb the value that would be added to a layer in AT. However, the value added to one layer will be taken as input to the other layer and this results in a difference in the perturbation. This difference, however gave better results.
4.