# COL775: Deep Learning

## Assignment 1.1: ResNet over Convolutional Networks and different Normalization schemes

**Student Name:** Aman Bhardwaj **Entry Number:** 2022ANZ8400

**MODELS URL:**https://drive.google.com/drive/folders/1t5p_gAntjrSz0UV9iumD_b3jejDzoNTF?usp=share_link

## 1 Image Classification using Residual Network

**Residual Network or ResNet** aims to learn residual mapping by providing the input of $t^{th}$ layer to $(t+2)^{th}$ via residual connections. Study shows that learning identity mapping for deeper networks becomes difficult for the models which results in performance degradation for deeper networks as compared to relatively shallower networks.

The skip connections help in mitigating the problem of vanishing gradients that can arise in deep networks. The skip connections enable information to travel directly between non-adjacent layers, allowing for more efficient training of deeper networks.

Table 1: Performance results of Resnet on CIFAR10 with Pytorch inbuilt batch normalization.

| | Micro F1 (%) | Macro F1 (%) | Accuracy (%) |
|---|---|---|---|
| **Without Augmentation** | | | |
| **Train Set** | 99.98% | 99.98% | 99.98% |
| **Val Set** | 83.01% | 82.99% | 83.01% |
| **Test Set** | 81.91% | 81.82% | 81.91% |
| **With Augmentation** | | | |
| **Train Set** | 98.03% | 98.03% | 98.03% |
| **Val Set** | 89.39% | 89.38% | 89.39% |
| **Test Set** | 89.57% | 89.56% | 89.57% |

**Key highlights of implementation and experimentation with Resnet for CIFAR10 classification:**

1. **Hyper-parameters:** Batch Size = 128, Max Epochs = 100, n=2 (total number of layers in the network is 6n+2), r=10 (for CIFAR10).

2. **Augmentation:** For base implementation using pytorch batch normalization, I experiment with augmentation of horizondal flip, padding followed by random cropping, and further compared the performance with no data augmentation.

3. **Criterion:** Cross Entropy Loss

4. **Optimizer:** Default optimizer used for all the experimentation was SGD. Start learning rate = 0.1.

5. **Scheduler:** I experiment with two schedulers namely StepLR and CosineAnnealingLR. However, for the base case **CosineAnnealingLR was giving better performance than StepLR**, therefore, for rest of the experimentation CosineAnnealingLR scheduler was used and LR is reduced for every epoch over 100 epochs which provides smooth loss curves as compared to StepLR for which we can observe sudden drop in loss values. **Stopping Criteria:** Best validation accuracy.

## 1.1 Impact of data augmentation on implementation with Torch Batch Normalization

In this section, we are going to discuss about the impact of data augmentation over model's performance. For the implementation of resnet using inbuilt function for batch normalization from pytorch (**BatchNorn2d**), the following data augmentation has been performed.

**Data Augmentation:** *Padding (4)* $\rightarrow$ *Random Crop (32×32)* $\rightarrow$ *Horizontal Flip.* Table 1 contains the performance metrics for the model trained with and without augmentation.

**Inferences from results Table 1:**

1. **Overfitting:** Without data augmentation the model was extremely overfitted as the training accuracy is 99.98% while the val and test performance are of the order of 81-83%.

2. **Reduction in Overfitting:** Data augmentation brought down the train accuracy to 98.03% from 99.98%. This shows that the model overfitting has slightly reduced due to data augmentation.

3. **Enhanced Performance:** As it can be clearly seen that the data augmentation enhances the test performance to 89.57% which was earlier 81.91%. This is a significant improvement in results.

4. **Difference between val and test performance:** As it can been seen that there is a difference of approximate 1.3% in test and val split. However, post data augmentation this difference comes down to only 0.2%. This shows that data augmentation such as random crop and horizontal flip makes model more robust.

## 1.2 Analysis of Train and Val Loss Curves

Figure 1 illustrates the decrease in loss values with increase in epochs.

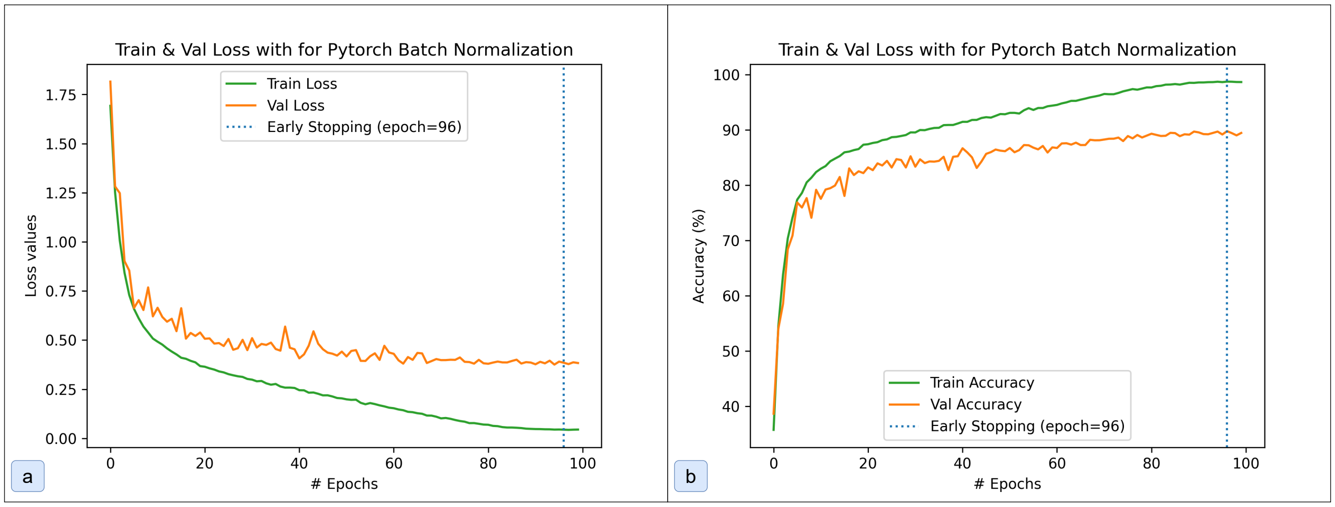

Figure 1: Train and Val (a) loss and (b) accuracy curve for pytorch batch normalization.

**Observations and inferences:**

1. **Smoothness of curves:** Train loss curve decreases quite smoothly, while val loss curve is quite irregular.
2. **Smoothness of val curve beyond a certain epoch:** Val loss curve becomes smoother beyond 65th epoch. This shows that the model has started to reach towards the bottom of minima valley as that change in loss values is very small as compared to earlier epochs.
3. **Decrease in train loss:** Train loss continues to decrease till approximately 90 epochs then become more flat towards the end.
4. **Loss value saturation:** Val loss attains close to minimum value at approx 25th epoch. Beyond that the loss value continues to fluctuate around 0.45 value.
5. **Overfitting:** Train loss continues to decrease while val loss becomes almost constant indicates that model tends to overfit after a certain point.
6. **Trends in accuracy curve:** Accuracy curve shows similar trends as loss curve. It also indicates that the train accuracy continues to increase while val accuracy is almost constant. Therefore, model is over-fitting beyond certain limit.

## 2 Impact of Normalization

### 2.1 Sanity check with comparison of Torch and Custom Batch Normalization

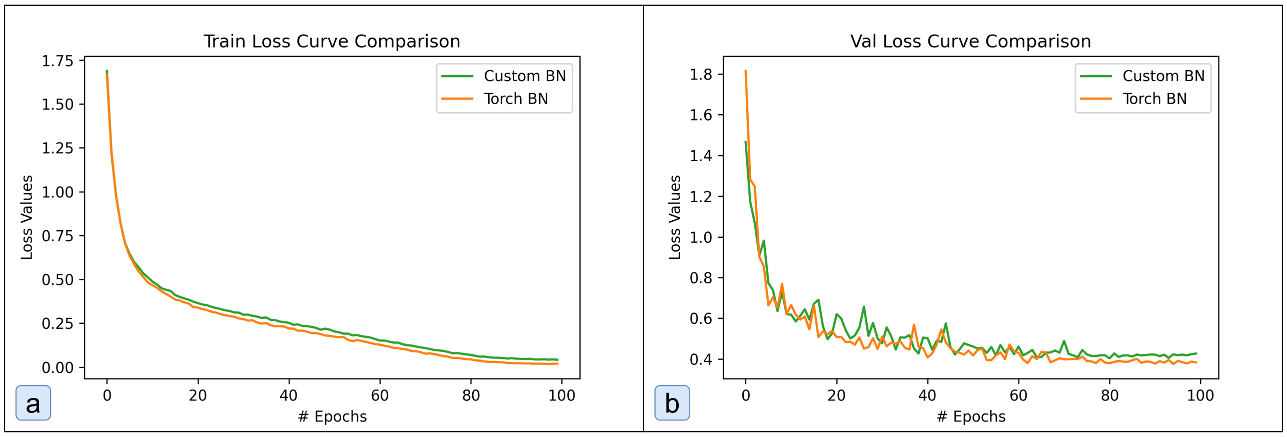

Figure 2: Comparison of (a) train and (b) val loss curves v/s epochs for custom batch normalization and inbuilt Pytorch batch normalization.

Table 2: Performance results of Resnet on CIFAR10 with Pytorch inbuilt batch normalization and all custom implementations of normalization schemes.

| | **Train (%)** | | | **Val (%)** | | | **Test (%)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Micro F1** | **Macro F1** | **Accuracy** | **Micro F1** | **Macro F1** | **Accuracy** | **Micro F1** | **Macro F1** |
| **Torch BN** | **98.03%** | 98.03% | 98.03% | **89.39%** | 89.39% | 89.38% | **89.57%** | 89.57% | 89.56% |
| **Custom BN** | **98.87%** | 98.87% | 98.87% | **89.21%** | 89.21% | 89.27% | **89.14%** | 89.14% | 89.16% |

Figure 2 illustrates the comparison of train and validation loss curves of Pytorch inbuilt `nn.BatchNorm2d` and custom implementation of batch normalization. Table 2 contains the corresponding performance matrics for both implementations on train test and val sets.

**Observations and inferences:**

1. **Similarity of Training Loss curves:** Referring to Figure 2 (a), we can see that the loss curves of both the implementations is similar, however, training loss for the pytorch implementation is slightly lower than that of custom implementation.

2. **Similarity of Validation Loss curves:** Referring to Figure 2 (b) one can observe similar trends as that of training loss. Validation loss for Torch implementation is slightly lower than that of custom normalization.

3. **Test Performance:** I was able to achieve similar test performance for both the implementation as it can be seen in Table 2. Torch BN achieves 0.43% better performance than custom implementation which is negligible.

4. **Train and Val performance:** Train and val performance metrics for both implementations are also similar.

Therefore, we can conclude that the custom batch normalization implementation is **correct and achieves similar loss curves and performance metrics** as that of Pytorch `nn.BatchNorm2D`.

## 2.2 Comparison of loss curves for all 6 Normalizations:

Figure 3 (a) illustrates the decay of loss values over train set and Figure 3 (b) over validation set for all the 6 implementations. We'll be using following abbreviations for the rest of the discussion: • Batch Normalization (BN) • Instance Normalization (IN) • Batch-Instance Normalization (BIN) • Layer Normalization (LN) • Group Normalization (GN) • No Normalization (NN)

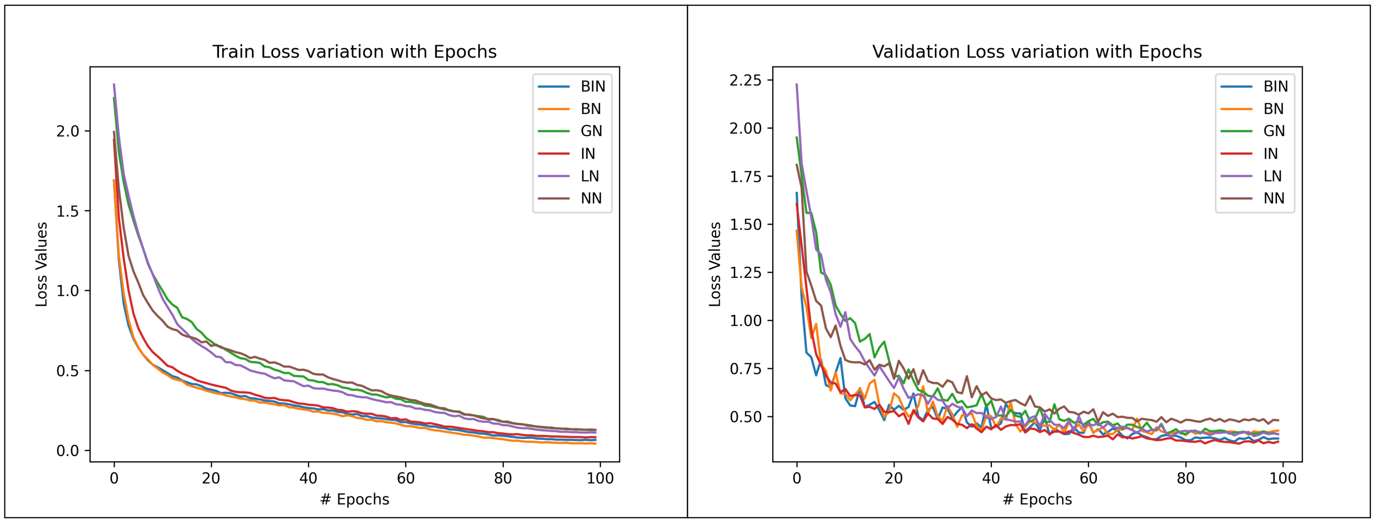

Figure 3: Comparison of (a) train and (b) validation loss curves v/s epochs for custom implementations of all 6 normalization schemes.

**Observations and inferences:**

1. **Loss trends and shapes:** For all the schemes train loss curves show similar behaviour of decay, however, the rate of decay and final convergence varies. It can be visually seen in Figure 3

2. **Comparison of Train Loss:**
    - **Initial rate of decay** of train loss if faster for BN and BIN, followed by IN, NN, LN, and GN respectively.
    - **Final loss values** are in the increasing order from BN, BIN, IN, LN, GN, NN. This shows that NN initially decays faster however ends up with higher train loss values than rest of the schemes.
3. **Comparison of Val Loss Curves:**
    - For all the implementations val loss curves are **fluctuating at the beginning** and then **becomes smoother towards the end** after 65-75 epochs. This shows model starts towards the bottom converge to a local minima valley.
    - **Initial rate of decay** shows similar trends as train loss curves.
    - **Final loss values** are quite higher for NN. This shows that **various normalization significantly improves training as compared to no normalization**.
4. **Generalization:** Val loss curves show that the Batch Normalization should achieve the best performance and NN the least. In next subsection we will compare the performance metrics and verify if this is true.
5. **Overfitting:** as per the comparison of train and val curves, BN tends to overfit slightly more that the other schemes on training data. However, it achieves better val loss values, therefore, if this is overfitting in actual is a bit ambiguous.

### 2.3 Comparison of performance metrics for all 6 Normalizations:

Table 3: Performance results of Resnet on CIFAR10 with Pytorch inbuilt batch normalization and all custom implementations of normalization schemes.

| | **Train (%)** | | | **Val (%)** | | | **Test (%)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Micro F1** | **Macro F1** | **Accuracy** | **Micro F1** | **Macro F1** | **Accuracy** | **Micro F1** | **Macro F1** |
| **Torch Batch Norm (TBN)** | **98.03%** | 98.03% | 98.03% | **89.39%** | 89.39% | 89.38% | **89.57%** | 89.57% | 89.56% |
| **No Norm (NN)** | 95.58% | 95.58% | 95.58% | 86.46% | 86.46% | 86.49% | 86.77% | 86.77% | 86.73% |
| **Batch Norm (BN)** | **98.87%** | 98.87% | 98.87% | **89.21%** | 89.21% | 89.27% | **89.14%** | 89.14% | 89.16% |
| **Batch Instance Norm (BIN)** | 98.26% | 98.26% | 98.26% | 88.79% | 88.79% | 88.83% | 87.65% | 87.65% | 87.62% |
| **Group Norm (GN)** | 95.74% | 95.74% | 95.74% | 87.51% | 87.51% | 87.48% | 86.84% | 86.84% | 86.83% |
| **Instance Norm (IN)** | 97.88% | 97.88% | 97.87% | 88.12% | 88.12% | 88.19% | 87.21% | 87.21% | 87.19% |
| **Layer Norm (LN)** | 96.30% | 96.3% | 96.3% | 87.36% | 87.36% | 87.29% | 87.11% | 87.11% | 87.07% |

Table 3 contains the performance results of models trained with different normalization schemes. I have also included results of default pytorch built in batch normalization in order to provide a comparison with the custom implementations.

**Observations and inferences:**

1. **Test Performance:** As can be seen in the Table 3, BN performs better than the rest of the schemes for both custom and torch implementations. Followed by BIN, IN, LN, GN, and NN. These results are inline with the validation loss values discussed in the previous sub section.
2. **Validation Performance:** Validation performance for all the implementations are similar to that of test performance. Therefore, this shows that after data augmentation, **val set provides true reflection of prospective test performance**.
3. **Train Loss and Overfitting:** BN tends to overfit slightly more that the other schemes on training data. However, it achieves better val and test performance values that others, therefore, if this is overfitting in actual is a bit ambiguous. But the gap between train and val performance show that over 100 epochs all models have gone through slight overfitting.

## 2.4 Impact of Batch Size in BN and GN:

Figure 4 and Figure 5 illustrates comparison of train and val loss for BN and GN with Batch Size = 128 and 8 respectively.

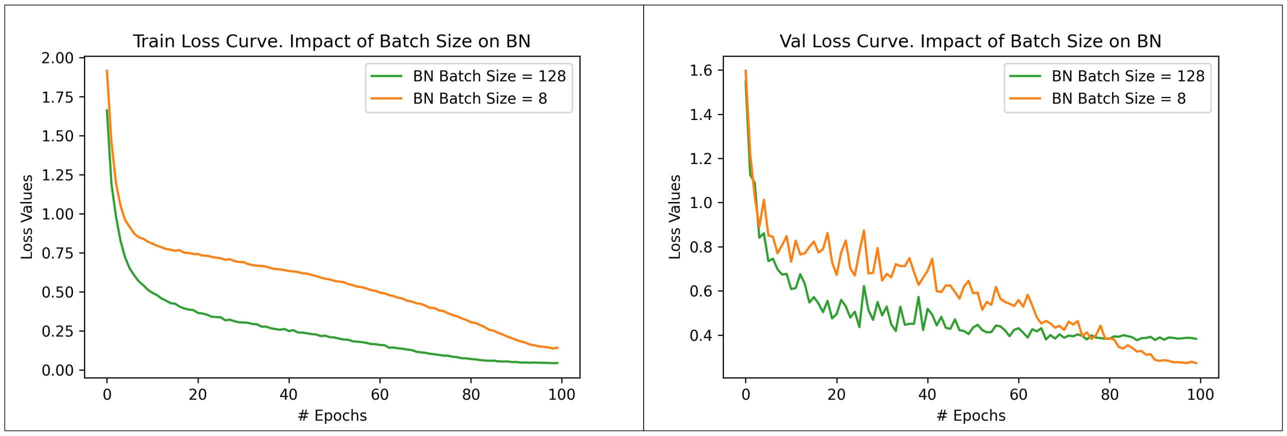

Figure 4: Impact of batch size on BN. (a) Train loss curve, (b) Val loss curve

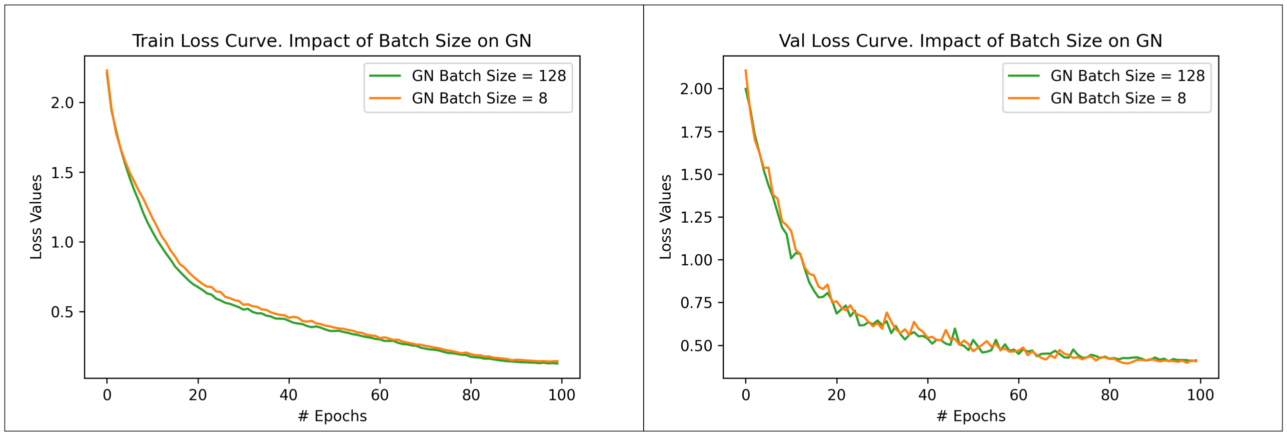

Figure 5: Impact of batch size on GN. (a) Train loss curve, (b) Val loss curve

**Observations and inferences:**

1. **Impact on loss curves for GN:** As it can be seen from Figure 5 that there is **no impact** of batch size on train and val loss curves. Therefore, this validates the claims by *[Wu and He, 2020]*.

2. **Impact on loss curves for BN:**

   - **Train Loss:** As it can be seen for train loss, for batch size = 128 the train loss decreases at a faster rate as compared to batch size = 8, therfore the curve for batch size = 8 remains lower than batch size = 128.
   - **Val Loss:** It reduces faster for batch size = 128. However, towards the end the val loss for batch size = 8 goes lower than that of batch size = 128 after 80th epoch.

3. **Impact on performance metrics for GN:** (refer Table 4) It can be seen that there is no change in performance of GN for both the batch sizes. This again validates that the performance of GN remains similar for both the cases.

4. **Impact on performance metrics for BN:** As the loss curves show, the test accuracy for batch size = 8 improves by 1.58% over batch size = 128. Also, the train accuracy for batch size = 8 is lower than that of batch size = 128 by 2.56%. This shows that smaller batch size **reduces overfitting and improves generalization capability** of the model in case of BN.

Table 4: Impact of batch size on Train and Test performance for BN and GN.

| | **Train Accuracy (%)** | **Test Accuracy (%)** |
|---|---|---|
| BN (Batch Size = 128) | 98.88% | 89.39% |
| BN (Batch Size = 8) | 96.32% | **90.97%** |
| GN (Batch Size = 128) | 95.52% | 86.78% |
| GN (Batch Size = 8) | 95.25% | 86.28% |

## 2.5 Evolution of feature distributions via quantile plots

In order to perform a sanity check I first compare the feature quantile plots of default Pytorch BN and custom BN in Figure 6.

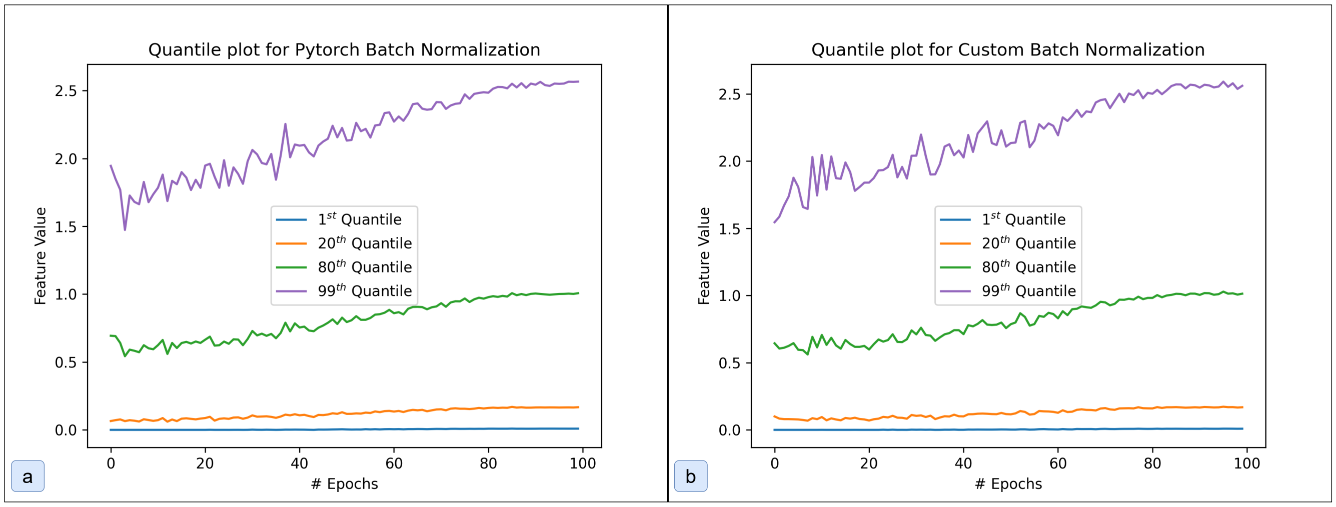

Figure 6: Plot of 1st, 20th, 80th, and 99th of (a) Pytorch inbuilt and (b) custom Batch Normalization.

**Comparison for Pytorch BN and custom BN quantile plots**

1. All quantile values i.e. $1^{st}$, $20^{th}$, $80^{th}$, and $99^{th}$ for both the implementations of BN shows similar trends as it can be seen in Figure 6.

2. Features $1^{st}$ quantile is close to zero, while $20^{th}$ is initially close to zero by increases gradually to a small value.

3. Values of $80^{th}$, and $99^{th}$ feature quantiles values vary significantly with increase in epochs ranging from o.5-1.0 and 1.5-2.5 respectively.

**Comparison of $1^{st}$, $20^{th}$, $80^{th}$, and $99^{th}$ feature quantile plots for all 6 normalizations** Figure 7 shows variation of different feature quantile values for all 6 normalization schemes.
**Observations and inferences:**

1. **Initialization:** For all the schemes, **quantile values at first epoch** show that initially $1^{st}$ quantile values are close to zero, followed by $20^{th}$, $80^{th}$, and $99^{th}$ values ranging from 0.2 to 1.8. This shows that model param initialization ranges from 0 to 1.8 approximately.

2. $1^{st}$ quantile value remain close to zero throughout for NN, BN, GN, and LN. This shows that some feature values remains close to zero throught the training. While the $1^{st}$ quantiles for BIN and IN increase during training.

3. For NN, even $20^{th}$ percentile is close to zero. While the values of $80^{th}$, and $99^{th}$ quantiles increase significantly throught the training. This shows that NN relies on comparatively lesser number of features making them more significant during training to predict the outcome. Therefore, achieves lesser performance than rest of the schemes.

4. One can infer that, using different normalization schemes we can control the feature distribution across the training differently for each of the schemes.

**NOTE: Quantile comparison plot for all 6 schemes is on next page due to Latex auto-formatting.**

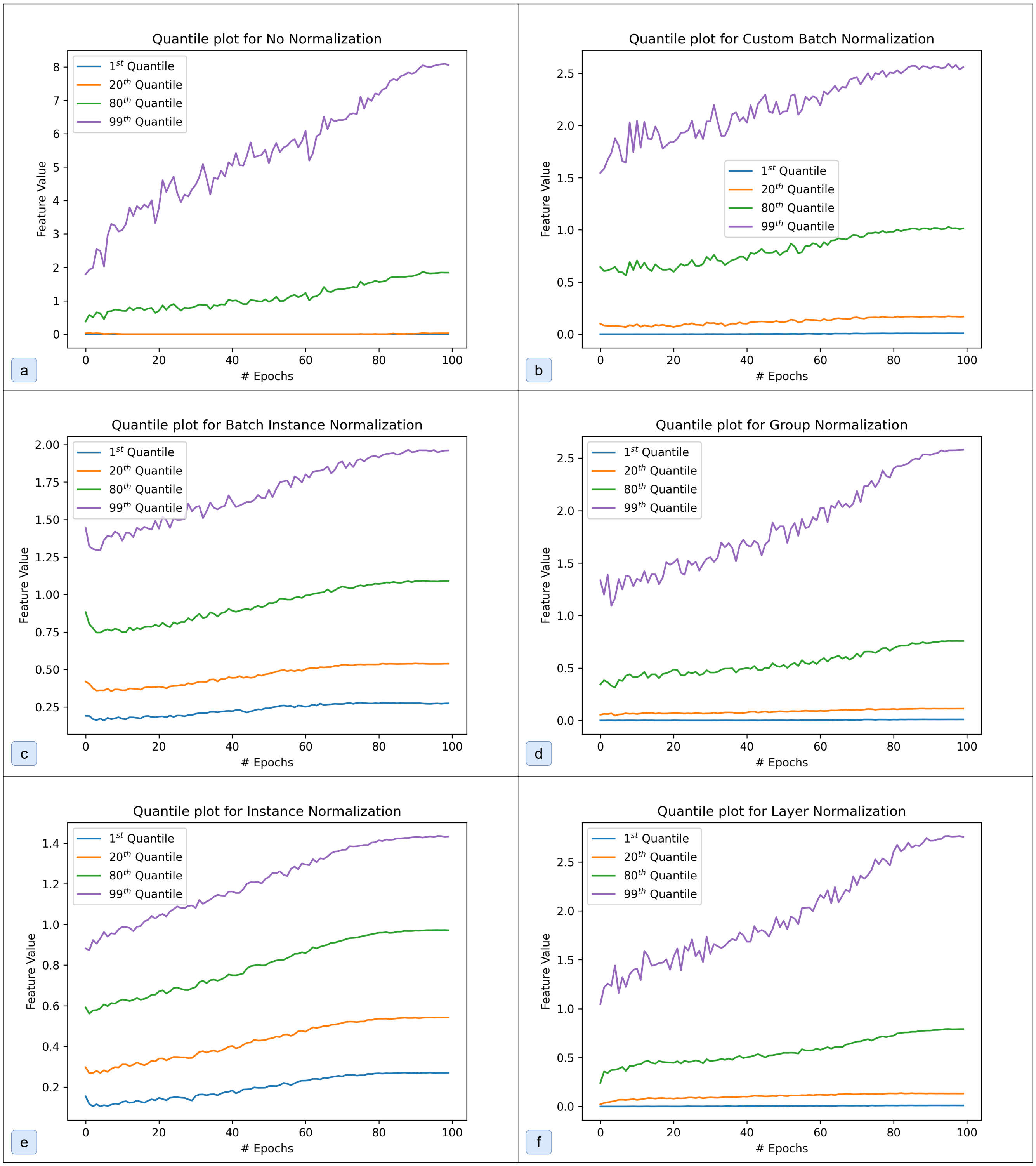

Figure 7: Comparison of $1^{st}$, $20^{th}$, $80^{th}$, and $99^{th}$ of all 6 normalization schemes.