# Lab 3: Evaluating normalization Techniques in CNNs using TensorFlow

**Thi Hien Nguyen**
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
AZ, USA
tn598@nau.edu

The code for this experiment is available at `https://github.com/hienngt/IST597_Fall2019_TF2.0`.

## 1 Problem 1

This report explores the impact of various normalization methods, specifically batch normalization, layer normalization, and weight normalization, on the training dynamics of a convolutional neural network (CNN) utilizing the MNIST dataset. The comparison focuses on both custom implementations and built-in versions of these normalization techniques, with an emphasis on evaluating their effects on training accuracy and loss. To maintain consistency and reproducibility in the experiments, a fixed random seed of 2701 was established for all relevant libraries used in the study.

### 1.1 Model architecture

The architecture of the CNN was developed using TensorFlow's low-level APIs, allowing for a high degree of customization. The network comprises several key components that work together to process input data effectively. Initially, the model includes a convolutional layer that employs the ReLU activation function, which introduces non-linearity to the learning process, followed by a max-pooling layer that helps reduce the spatial dimensions of the feature maps. Following this, a fully connected (dense) layer is incorporated, which can optionally be enhanced through various normalization techniques, including batch normalization, layer normalization, or weight normalization. Finally, the architecture concludes with an output layer, which may also be subject to normalization, depending on the chosen configuration.

### 1.2 Normalization support

In this study, batch normalization and layer normalization are applied after the hidden dense layer, while weight normalization is implemented directly on the weights of the final layer. This approach allows for a comprehensive analysis of how each normalization method influences the training process. Both custom and built-in versions of these normalization techniques are included in the comparison, facilitating a thorough evaluation of their respective performances.

### 1.3 Custom normalization implementations

Batch normalization normalizes across the batch dimension, ensuring that the mean and variance are consistent across different samples. Layer normalization, on the other hand, normalizes across feature dimensions for each individual sample, allowing for greater stability in training. Weight normalization reparameterizes weights into direction and magnitude, which can improve convergence rates and enhance the overall training dynamics. All custom implementations are constructed using raw TensorFlow operations to clarify their mathematical foundations.

### 1.4 Experimental variants

Six model variants were trained to assess the effects of normalization methods on performance. These variants include configurations with no normalization, custom and built-in batch normalization, custom and built-in layer normalization, and custom weight normalization only.

### 1.5 Results and observations

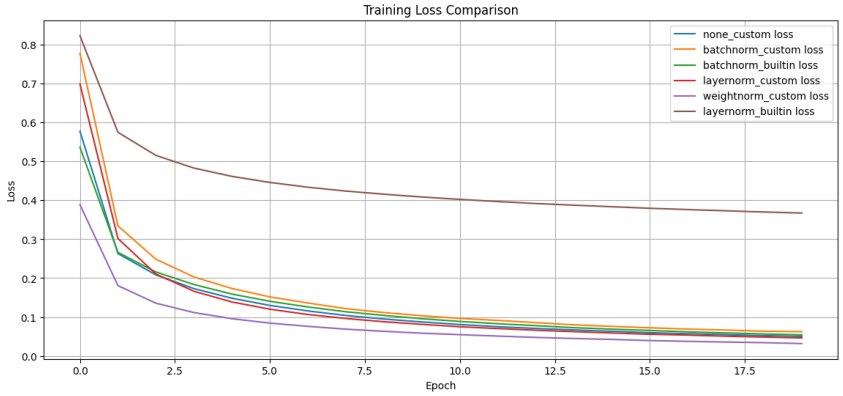


Figure 1: Training loss comparison for all methods.

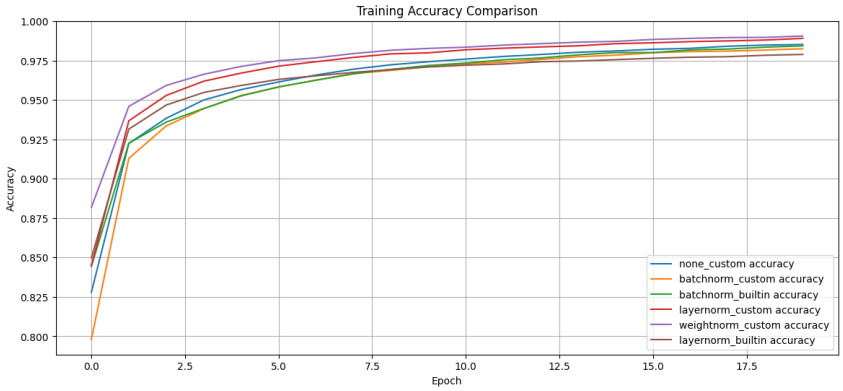


Figure 2: Training accuracy comparison for all methods.

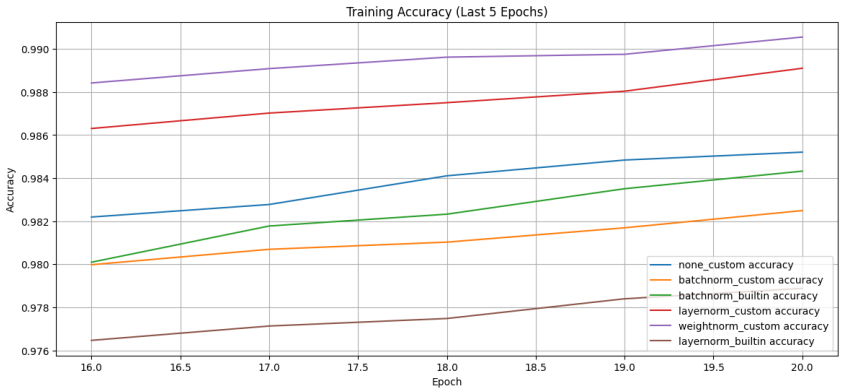


Figure 3: Training accuracy (last 5 epochs).


Figure 4: Training loss (last 5 epochs).

All normalization methods significantly enhance both the convergence speed and the final accuracy of the model when compared to configurations lacking any normalization. However, contrary to the initial assessment, the custom implementations generally outperform their built-in counterparts in this experiment. Weight normalization, implemented only in its custom form, demonstrates not just stable training dynamics but achieves the highest accuracy levels among all methods tested.

Among the various normalization techniques, custom weight normalization consistently achieves the highest accuracy during the final five epochs, starting at approximately 0.988 and reaching just above 0.991 by epoch 20. The custom layer normalization implementation performs second best, maintaining accuracy between 0.986 and 0.989. It is noteworthy that the built-in layer normalization significantly underperforms relative to all other methods, exhibiting the lowest accuracy throughout the training process (below 0.978).

## 2 Conclusion

In conclusion, normalization techniques are essential for stabilizing and accelerating the training of neural networks. Based on the results shown, the custom implementation of weight normalization outperformed all other methods, achieving the highest training accuracy, exceeding 0.99 in the final epochs. The custom layer normalization also demonstrated strong and consistent improvements, maintaining accuracy just below 0.99. In contrast, TensorFlow's built-in normalization layers, particularly built-in layer normalization, lagged behind their custom counterparts in both accuracy and convergence speed. These findings highlight the effectiveness of carefully implemented custom normalization strategies, with weight normalization being especially impactful in this experiment.