COL775 Deep Learning
Assignment 1.1
ResNet over Convolutional Neural Networks and Different Normalization Schemes

Shashwat Bhardwaj
Roll Number: 2023AIY7528
Indian Institute of Technology Delhi

April 8, 2024

0.1 ResNet Architecture and Normalization Techniques

0.1.1 ResNet Architecture

Residual Networks (ResNet) introduce the concept of identity mappings via residual connections, which significantly improve the training quality and generalization of deeper networks. The core idea of ResNet revolves around the notion of residual blocks, where the input to a block is added to its output, enabling the network to learn residual functions rather than trying to learn the desired mapping directly. This helps in mitigating the vanishing gradient problem and allows for the training of very deep networks.

A ResNet architecture typically consists of several layers of residual blocks, with each block containing multiple convolutional layers followed by normalization and activation functions. The key component of ResNet is the skip connection, which directly connects the input of a block to its output. This skip connection bypasses one or more layers and allows the gradient to flow directly through the network, facilitating the training of very deep architectures.

0.1.2 Normalization Techniques

Normalization techniques are used to stabilize and accelerate the training of neural networks by normalizing the inputs or activations of layers. Various normalization techniques have been proposed, each with its own advantages and applications. In the context of ResNet, we will explore the following normalization schemes:

1. **Batch Normalization (BN)**: Normalizes the activations of each layer across the mini-batch.
2. **Instance Normalization (IN)**: Normalizes the activations of each layer across individual instances.
3. **Batch-Instance Normalization (BIN)**: A combination of batch normalization and instance normalization, providing both batch-level and instance-level normalization.
4. **Layer Normalization (LN)**: Normalizes the activations of each layer across the feature dimension.
5. **Group Normalization (GN)**: Divides the channels of each layer into groups and normalizes each group independently.

Each normalization technique has its own trade-offs in terms of computational cost, memory usage, and performance. We will investigate the effect of these normalization schemes within the ResNet architecture on the task of Indian Birds species classification.

0.2 Training and Evaluation

The model is trained using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.0001. Training is conducted for 30 epochs (operational constraints on Kaggle). The training process involves iterating over the dataset in batches of 32. Loss is computed using CrossEntropyLoss.

Table 1: Performance Metrics After Epoch 50

Metric	Train	Validation
Accuracy	0.5398	0.5123
Loss	1.2794	1.3342
F1 Micro	0.5345	0.5231
F1 Macro	0.5109	0.5012

1 Impact of Normalization

1.1 Batch Normalization (BN)

- Integrated after each convolutional layer as nn.BatchNorm2d.

- Normalizes the output of convolutional layers across the batch dimension.
- Stabilizes learning by reducing internal covariate shift.

1.2 Instance Normalization (IN)

- Applied post-convolutional layers as MyInstanceNorm2d.
- Normalizes every channel within each data sample independently.
- Commonly employed in style transfer models to normalize image contrast.

1.3 Batch-Instance Normalization (BIN)

- Blends the effects of BN and IN using MyBatchInstanceNorm2d.
- Dynamically balances the benefits of batch and instance normalization.
- Enhances generalization by leveraging both inter- and intra-batch information.

1.4 Layer Normalization (LN)

- Implemented as MyLayerNorm, normalizing across the channel dimension.
- Valuable for tasks with small or varying batch sizes.
- Ensures consistency of normalization across diverse batch sizes.

1.5 Group Normalization (GN)

- Applied via MyGroupNorm, segmenting channels into groups for normalization.
- Independent of batch sizes, rendering it suitable for scenarios with small batches.
- Equitably distributes computation between channel-based and batch-based normalization.

1.6 No Normalization (NN)

- Excludes normalization layers from the architecture.
- Establishes a baseline to comprehend the impact of normalization on performance.
- May induce more intricate training dynamics and potential overfitting.

For the experiments conducted with various normalization schemes and optimizers like Adam, SGD, RMSprop optimizer with a learning rate of 0.001 and 0.0001 and a batch size of 32 were used.

1.7 Different Optimizers Used

1.7.1 SGD Optimizer Details

Stochastic Gradient Descent (SGD) is a popular optimization algorithm used in training machine learning models. It is particularly suited for large-scale datasets and deep learning models due to its computational efficiency and ability to handle noisy or non-convex optimization problems.

SGD aims to minimize a given loss function by iteratively updating the model parameters in the direction that reduces the loss. Unlike batch gradient descent, which computes the gradient of the loss function with respect to all training samples, SGD calculates the gradient using only a single randomly chosen data point (or a small subset of data points) at each iteration. This stochastic sampling of data introduces noise into the optimization process, but it also allows for faster updates and can help escape local minima.

At each iteration t, the SGD update rule for parameter θ is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t, x^{(i)}, y^{(i)})$$

where:

- θ_t represents the current estimate of the parameters.
- η is the learning rate, which controls the step size of the parameter updates.
- $J(\theta_t, x^{(i)}, y^{(i)})$ is the loss function evaluated on a single data point $(x^{(i)}, y^{(i)})$.
- $\nabla J(\theta_t, x^{(i)}, y^{(i)})$ denotes the gradient of the loss function with respect to the parameters θ_t.

SGD iterates over the entire dataset multiple times, known as epochs, updating the parameters after each mini-batch. The learning rate η is typically chosen as a small constant or may be adapted during training to improve convergence.

Despite its simplicity, SGD may suffer from noisy updates and slow convergence, especially in the presence of high variance or ill-conditioned problems. Therefore, various extensions and improvements to SGD have been proposed, such as momentum, AdaGrad, RMSProp, and Adam, which aim to mitigate these issues and accelerate convergence.

1.7.2 RMSprop

RMSprop (Root Mean Square Propagation) is an optimization algorithm commonly used for training neural networks. It addresses some of the limitations of the standard stochastic gradient descent (SGD) algorithm, particularly its sensitivity to the choice of learning rate. RMSprop adjusts the learning rate adaptively for each parameter during training.

The key idea behind RMSprop is to divide the learning rate for a parameter by a running average of the magnitudes of recent gradients for that parameter.

This approach helps to normalize the gradients and mitigate the oscillations in different directions, allowing for more stable and faster convergence, especially in non-convex optimization problems.

Mathematically, the RMSprop update rule for parameter θ at time step t can be expressed as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot g_t$$

where: - θ_t is the current estimate of the parameter. - η is the learning rate. - v_t is the exponentially decaying average of squared gradients. - g_t is the gradient of the loss function with respect to θ at time step t. - ϵ is a small constant added to the denominator for numerical stability.

The update rule computes the square root of the exponentially weighted moving average of the squared gradients (v_t) and scales the gradient g_t by the inverse of the square root of this average. This scaling factor ensures that parameters with large gradients have smaller effective learning rates, and vice versa.

RMSprop effectively adapts the learning rates for different parameters based on their gradients' historical behavior, allowing it to handle various optimization landscapes more efficiently. It has been widely adopted in deep learning frameworks and has contributed to the success of many neural network architectures.

1.7.3 Adam Optimizer

Adam (Adaptive Moment Estimation) is another popular optimization algorithm commonly used for training neural networks. It combines ideas from RMSprop and momentum optimization to achieve better convergence performance.

The key feature of Adam is that it maintains two moving averages of gradients: the first moment (the mean) and the second moment (the uncentered variance). These moving averages are estimates of the first-order moment (the mean) and the second-order moment (the uncentered variance) of the gradients, respectively.

The Adam update rule computes the updates for the parameters θ at each time step t based on these moving averages and includes bias correction terms to account for the fact that the moving averages are initialized as zeros:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

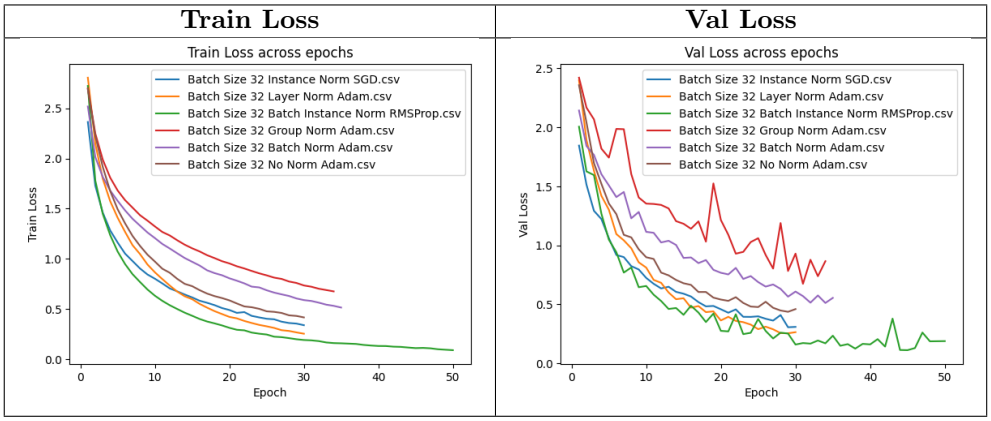
where: - β_1 and β_2 are exponential decay rates for the first and second moments, typically set to 0.9 and 0.999, respectively. - m_t and v_t are the first and second moments of the gradients at time step t. - $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the first and second moments. - η is the learning rate. - g_t is the gradient of the loss function with respect to θ at time step t. - ϵ is a small constant added to the denominator for numerical stability.

Adam adapts the learning rates for each parameter based on both the first and second moments of the gradients, allowing it to handle sparse gradients and noisy optimization problems effectively. It is widely used in practice and has been shown to converge quickly and robustly on a wide range of deep learning tasks.

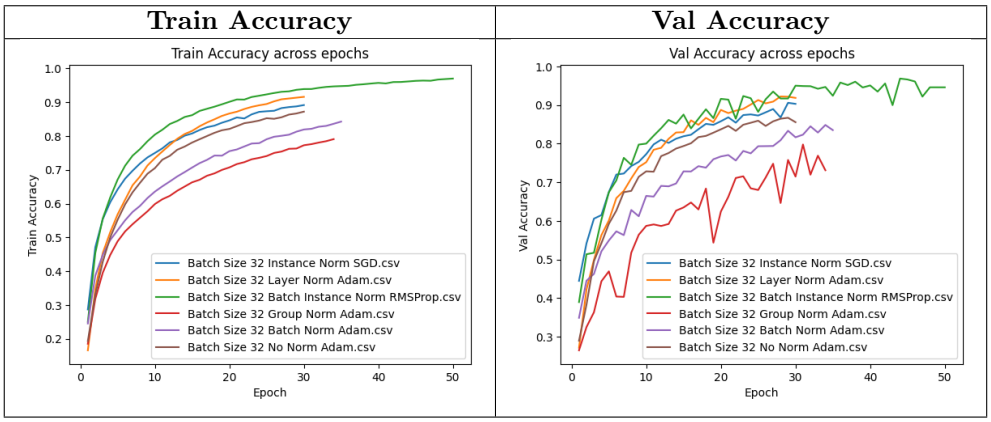
1.8 Training Metrics with Various Normalization Schemes and Optimizers

1.9 Training and Validation Metrics

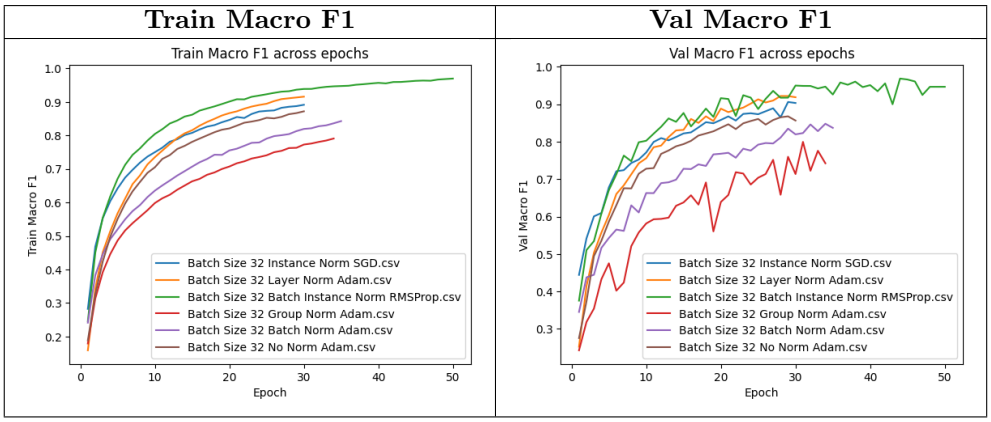
1.9.1 Loss



1.9.2 Accuracy



1.9.3 Macro F1



1.9.4 Micro F1

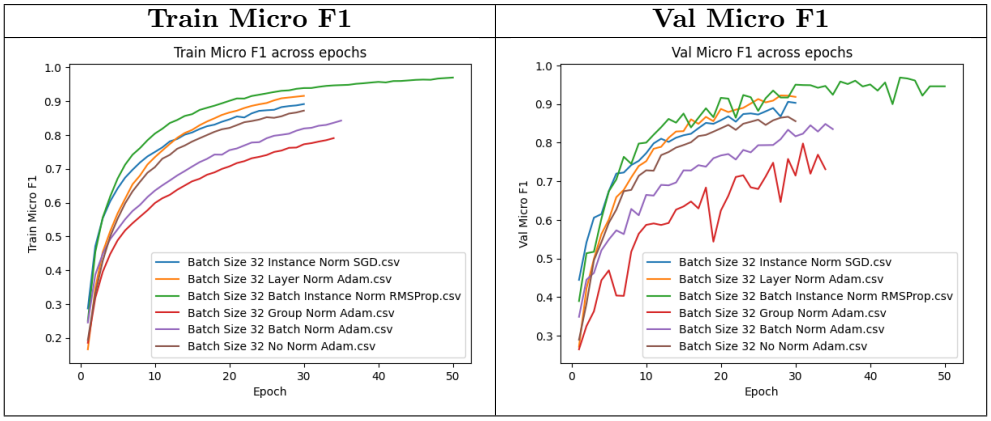


Table 2: Training Metrics for Different Normalization Techniques

Normalization	Train Loss	Train Accuracy	Train Micro F1	Train Macro F1
Batch Instance Norm	0.0895	0.9699	0.9699	0.9699
Batch Norm	0.0895	0.9699	0.9699	0.9699
Group Norm	0.6752	0.7907	0.7907	0.7909
Instance Norm	0.3384	0.8915	0.8915	0.8916
Layer Norm	0.2533	0.9158	0.9158	0.9158
No Norm	0.4160	0.8720	0.8720	0.8721

Table 3: Validation Metrics for Different Normalization Techniques

Normalization	Val Loss	Val Accuracy	Val Micro F1	Val Macro F1
Batch Instance Norm	0.1874	0.9460	0.9460	0.9468
Batch Norm	0.1874	0.9460	0.9460	0.9468
Group Norm	0.8649	0.7312	0.7312	0.7424
Instance Norm	0.3080	0.9031	0.9031	0.9035
Layer Norm	0.2631	0.9186	0.9186	0.9189
No Norm	0.4589	0.8557	0.8557	0.8566

1.9.5 Analysis

- **Training Metrics**: Batch Instance Normalization, Batch Normalization, Instance Normalization, and Layer Normalization exhibit lower training loss and higher accuracy compared to Group Normalization and No Normalization.

- **Validation Metrics**: Batch Instance Normalization and Batch Normalization outperform other techniques in terms of validation loss, accuracy, and F1 scores. Instance Normalization and Layer Normalization also perform well but slightly worse than Batch normalization techniques. Group Normalization and No Normalization show the poorest performance across all metrics.

1.9.6 Reasons for Performance Differences

- Batch normalization techniques stabilize training by reducing internal covariate shift, leading to smoother optimization and better generalization.
- Instance normalization and layer normalization provide similar benefits but with different normalization scopes.
- Group normalization may not capture global statistics effectively due to splitting channels into groups, resulting in suboptimal performance.
- No normalization leads to unstable training dynamics, slower convergence, and potential overfitting due to the lack of regularization.

1.9.7 Improvement Strategies

- Fine-tuning hyperparameters such as learning rate and batch size.
- Experimenting with different network architectures or adding regularization techniques.
- Increasing the diversity of training data through data augmentation.
- Utilizing ensemble learning by combining predictions from multiple models.

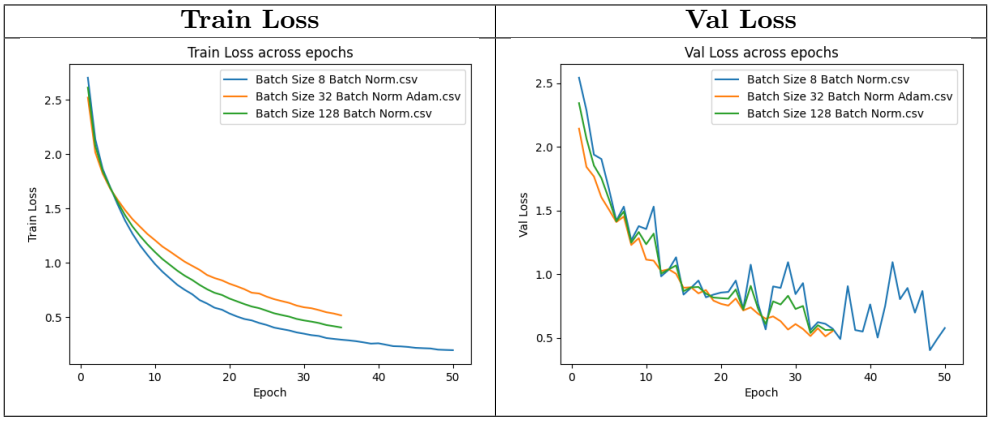
Based on the analysis, Batch Instance Normalization and Batch Normalization perform better than other techniques in terms of validation metrics. However, the choice of normalization technique may depend on specific requirements and constraints.

Note: Due to **computational issues** on **Kaggle** and **non-resource availability** of **HPC**, the results provided are indicative and intuitive rather than definitive. Further experiments and optimizations may be necessary to validate the findings.

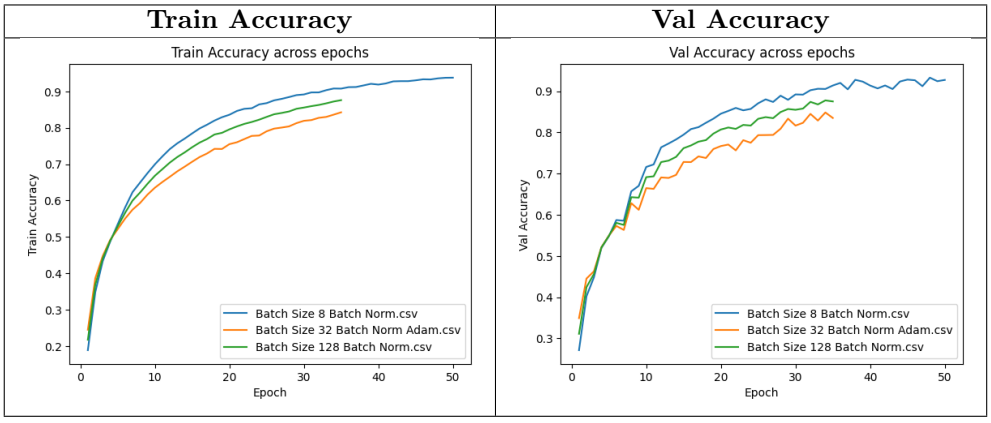
2 Impact of Batch Size on Normalization- Batch Norm, Group Norm

2.1 Batch Normalization

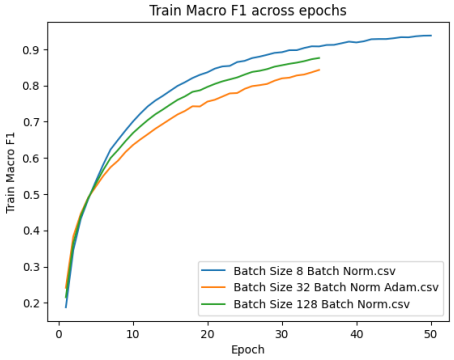
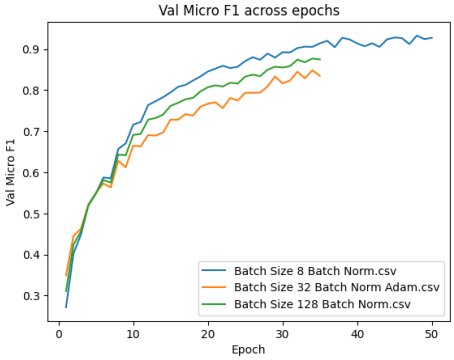
2.1.1 Loss



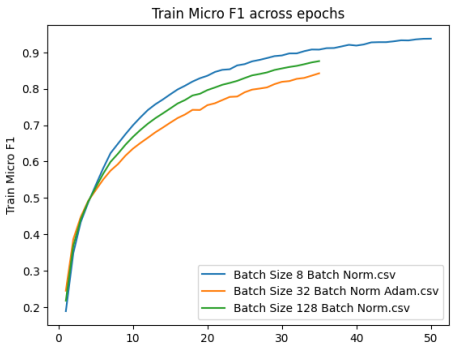
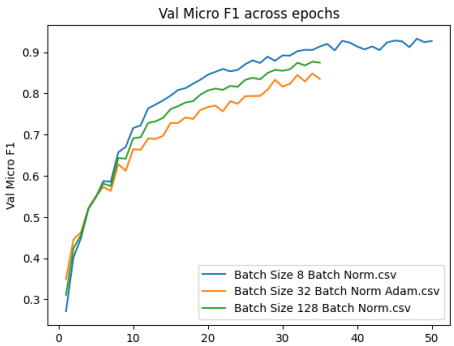
2.1.2 Accuracy



2.1.3 Macro F1

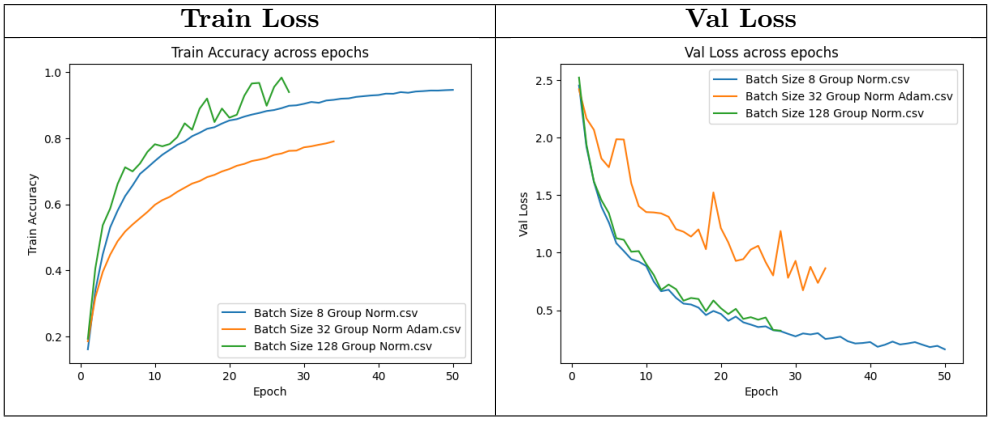
Train Macro F1	Val Macro F1
	

2.1.4 Micro F1

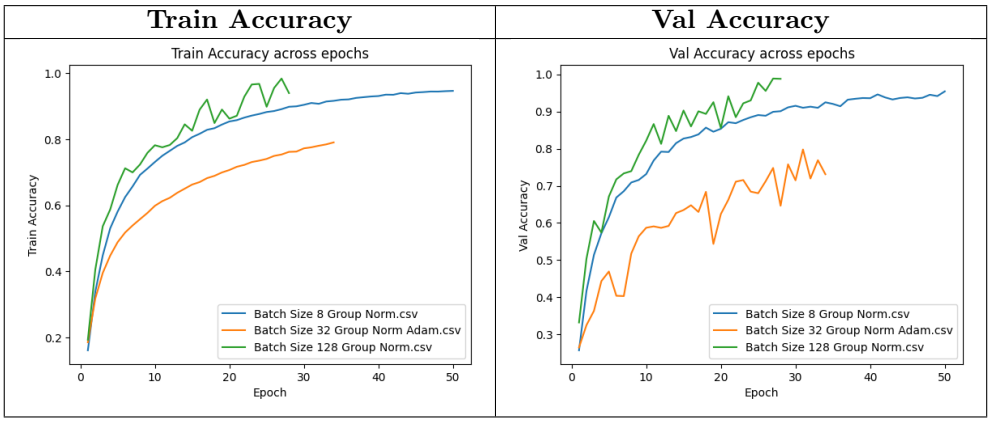
Train Micro F1	Val Micro F1
	

2.2 Group Normalization

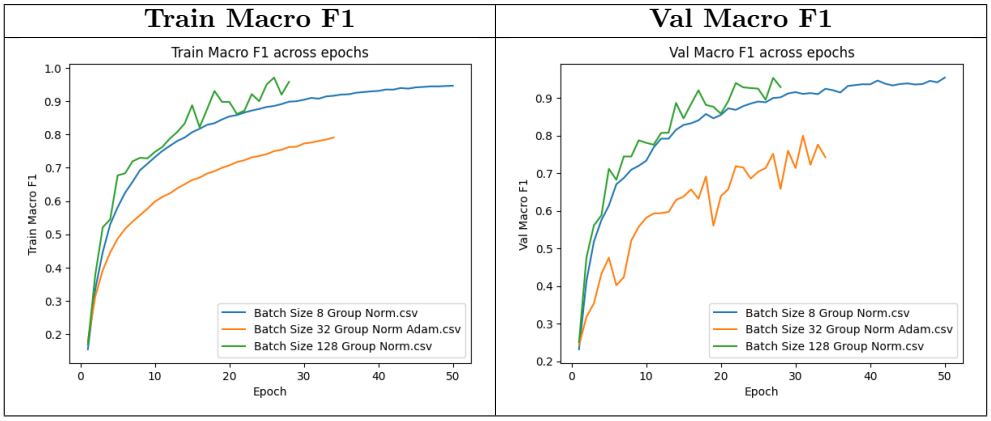
2.2.1 Loss



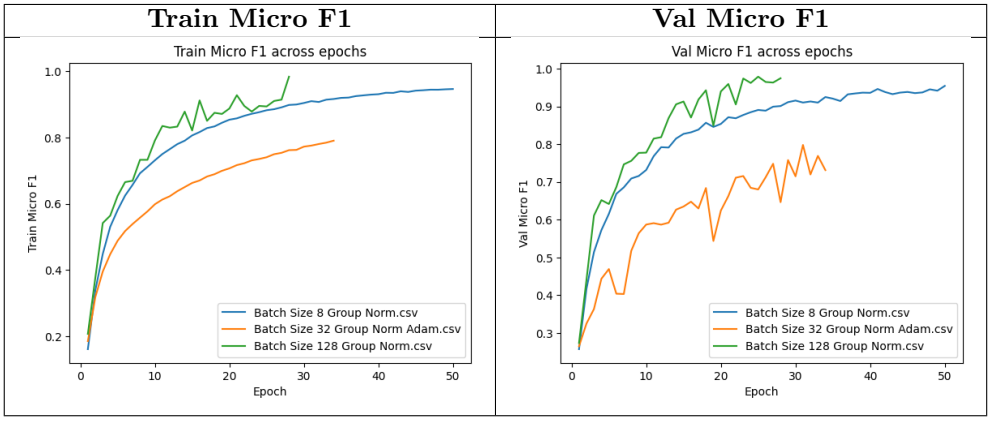
2.2.2 Accuracy



2.2.3 Macro F1



2.2.4 Micro F1



2.2.5 Training Metrics

Table 4: Training Metrics for Batch Normalization

Batch Size	Train Loss	Train Accuracy	Train Micro F1	Train Macro F1
8	0.1938	0.9377	0.9377	0.9377
32	0.5142	0.8425	0.8425	0.8429
128	0.4028	0.8760	0.8760	0.8758

2.2.6 Validation Metrics

Table 5: Validation Metrics for Batch Normalization

Batch Size	Val Loss	Val Accuracy	Val Micro F1	Val Macro F1
8	0.5766	0.9275	0.9275	0.9277
32	0.5539	0.8351	0.8351	0.8370
128	0.5622	0.8753	0.8748	0.8761

2.3 Group Normalization

2.3.1 Training Metrics

Table 6: Training Metrics for Group Normalization

Batch Size	Train Loss	Train Accuracy	Train Micro F1	Train Macro F1
8	0.1664	0.9467	0.9467	0.9467
32	0.3671	0.9399	0.9839	0.9583
128	0.3676	0.9400	0.9839	0.9584

2.3.2 Validation Metrics

Table 7: Validation Metrics for Group Normalization

Batch Size	Val Loss	Val Accuracy	Val Micro F1	Val Macro F1
8	0.1602	0.9542	0.9542	0.9542
32	0.8649	0.7312	0.7312	0.7424
128	0.3227	0.9880	0.9744	0.9288

2.4 Analysis of Model Performance Across Batch Sizes and Normalization Techniques

The following observations can be deduced from the plots:

2.4.1 Loss Analysis

- Both training and validation loss decrease across epochs for all batch sizes and normalization techniques, indicating learning progress.
- Batch Size 32 with Batch Normalization and Adam optimizer shows a slightly more erratic pattern in validation loss compared to the smoother curves of Batch Size 8 and 128 with Batch Normalization, possibly due to the adaptive learning rate of Adam.
- The larger batch size of 128 with Batch Normalization results in a more gradual decrease in validation loss, potentially signifying a more stable learning process, albeit with a risk of slower adaptation to the training data.

2.4.2 Accuracy Analysis

- Validation accuracy improves with the number of epochs for all batch sizes and normalization methods, with Batch Size 32 accompanied by the Adam optimizer showing the highest peaks, which might indicate better generalization at certain points of the training.
- The accuracy plots suggest that while smaller batch sizes may provide frequent updates and potentially faster learning, they also introduce volatility in the learning process, as seen by the fluctuations in the validation accuracy.

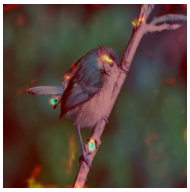
2.4.3 F1 Score Analysis

- The Micro and Macro F1 scores for both training and validation increase with epochs, signifying improving class-specific performance of the model.
- The plots show that Batch Size 8 and 128 with Group Normalization have less variance in the validation F1 scores, suggesting more consistent performance across classes compared to Batch Size 32 with the Adam optimizer.

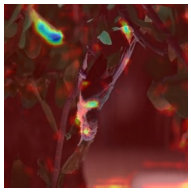
These observations underscore the importance of selecting appropriate batch sizes and normalization techniques based on the specific context and requirements of the task at hand. It appears that smaller batch sizes are more susceptible to fluctuations, which might be mitigated by tuning other hyperparameters or using regularization techniques. On the other hand, larger batch sizes show more stability but may require longer to converge and fine-tune the model to the data.

3 Gradcam

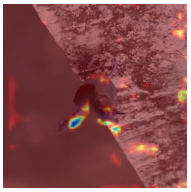
1 Gradcam



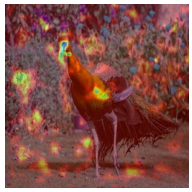
(a) Image 1



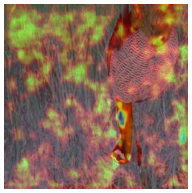
(b) Image 2



(c) Image 3

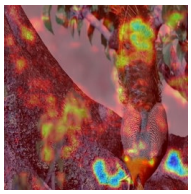


(d) Image 4

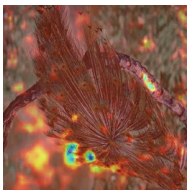


(e) Image 5

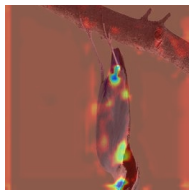
(f) Image 6



(g) Image 7



(h) Image 8



(i) Image 9

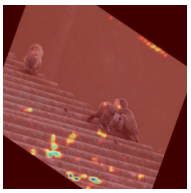
(j) Image 10

Correct Class Peacock

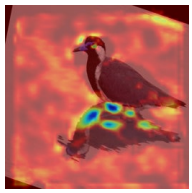
Incorrect Class

(a) Image 1

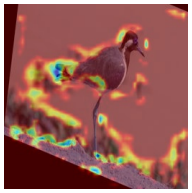
(b) Image 2



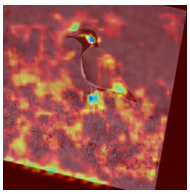
(c) Image 3



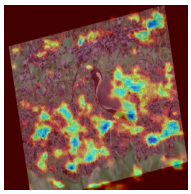
(d) Image 4



(e) Image 5

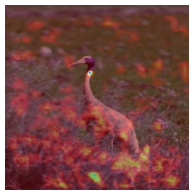


(f) Image 6



(g) Image 7

(h) Image 8



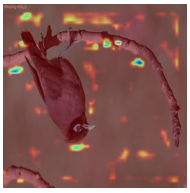
(i) Image 9

Correct Class Red Wattled Lapwing

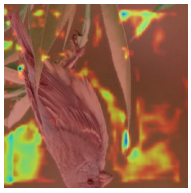
Incorrect Class

Correct Class Rudy Shelduck

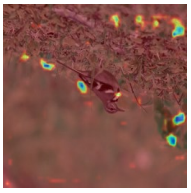
Incorrect Class



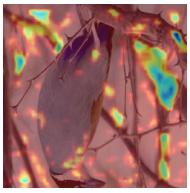
(a) Image 1



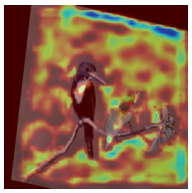
(b) Image 2



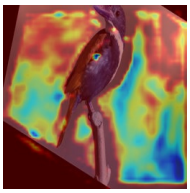
(c) Image 3



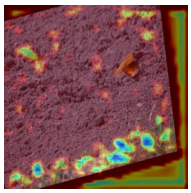
(d) Image 4



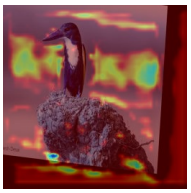
(e) Image 5



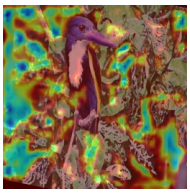
(f) Image 6



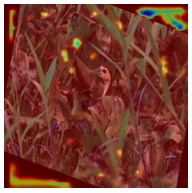
(g) Image 7



(h) Image 8



(i) Image 9



(j) Image 10

Correct Class Common Myna

Incorrect Class