Deep Learning Assignment-1 Report

Pon Vinayak Vickram M
24CS60R52

01 March 2025

1 Introduction

The goal of this assignment is to implement and train a custom VGG16-like neural network architecture for image classification using the MNIST dataset. The key focus is on building custom versions of essential deep learning components, including:

- Batch Normalization (CustomBatchNorm2d)
- ReLU Activation (CustomReLU)
- Max Pooling (CustomMaxPooling2d)
- Dropout (CustomDropout)

1.1 Importance of Custom Components

Building custom implementations of these components is crucial for several reasons:

1. **Deep understanding:** By coding these layers from scratch, we gain insight into their internal workings, which is essential for mastering deep learning concepts.
2. **Flexibility:** Custom components allow for modifications and optimizations tailored to specific use cases or research needs.
3. **Debugging skills:** Implementing these components helps develop the ability to troubleshoot and fine-tune neural networks more effectively.

4. **Performance optimization:** Understanding the low-level operations can lead to more efficient implementations and better resource utilization.
5. **Research and innovation:** The ability to create custom components is vital for developing novel neural network architectures and techniques.

This exercise serves as a practical exploration of the fundamental building blocks of modern neural networks. By implementing these components ourselves, we develop a deeper appreciation for the intricacies of deep learning frameworks and enhance our ability to work with and extend them in the future.

2 Implementation of Custom Components

2.1 CustomBatchNorm2d

The CustomBatchNorm2d implementation follows the standard batch normalization process:

1. Initialize parameters: gamma (scale), beta (shift), running mean, and running variance.
2. In the forward pass:
    - Calculate batch mean and variance
    - Normalize the input using these statistics
    - Apply learnable parameters gamma and beta
    - Update running statistics during training

Key challenges included ensuring correct dimensionality handling for 2D inputs and implementing the running statistics update mechanism.

2.2 CustomReLU

CustomReLU was implemented as a simple element-wise operation:

1. In the forward pass, use torch.max to replace all negative values with 0
2. No learnable parameters or special handling required

The main challenge was to implement this without using the built-in F.relu function, which was overcome by using torch.max with a zero tensor.

2.3 CustomMaxPooling2d

CustomMaxPooling2d implementation involved:

1. Using torch.nn.functional.unfold to create sliding windows over the input
2. Applying max operation over these windows
3. Reshaping the output to the correct dimensions

A key challenge was handling edge cases where input dimensions were smaller than the kernel size, which was addressed by adjusting the kernel size and stride dynamically.

3 VGG16 Model Architecture

The VGG16 model architecture consists of:

1. Five blocks of convolutional layers, each followed by max pooling
2. Three fully connected layers at the end

Key aspects of the implementation:

- Used custom components (CustomBatchNorm2d, CustomReLU, CustomMaxPooling2d) in convolutional blocks
- Maintained the standard VGG16 structure with 13 convolutional layers and 3 fully connected layers
- Adjusted the final fully connected layer to match the number of classes in the MNIST dataset (10 classes)

3.1 Custom Components in Convolutional Blocks

Each convolutional block typically includes:

1. Convolutional layer (nn.Conv2d)
2. CustomBatchNorm2d
3. CustomReLU
4. CustomMaxPooling2d (at the end of each block)

This pattern is repeated throughout the network, with increasing numbers of filters in deeper layers.

3.2 Design Choices

- Activation Functions: CustomReLU was used after each convolutional and batch normalization layer to introduce non-linearity
- Kernel Sizes: Maintained the standard 3x3 kernel size for all convolutional layers, as per the original VGG16 design
- Layer Design: Followed the VGG16 pattern of increasing the number of filters (64, 128, 256, 512) as the network deepens
- Pooling: Used CustomMaxPooling2d with 2x2 windows and stride 2 to reduce spatial dimensions between blocks

These design choices closely follow the original VGG16 architecture while incorporating the custom implementations of key components.

4 Training Strategy

The training strategy for the custom VGG16 model implementation includes the following key components:

4.1 Hyperparameters

- Batch size: 16
- Number of epochs: 3
- Learning rate: 0.001

4.2 Optimizer

The model uses the Adam optimizer, which is well-suited for deep learning tasks due to its adaptive learning rate capabilities.

4.3 Learning Rate Schedule

No explicit learning rate schedule is implemented in the provided code. The learning rate remains constant at 0.001 throughout training.

4.4 Training and Testing Accuracy/Loss Curves

[b]0.48

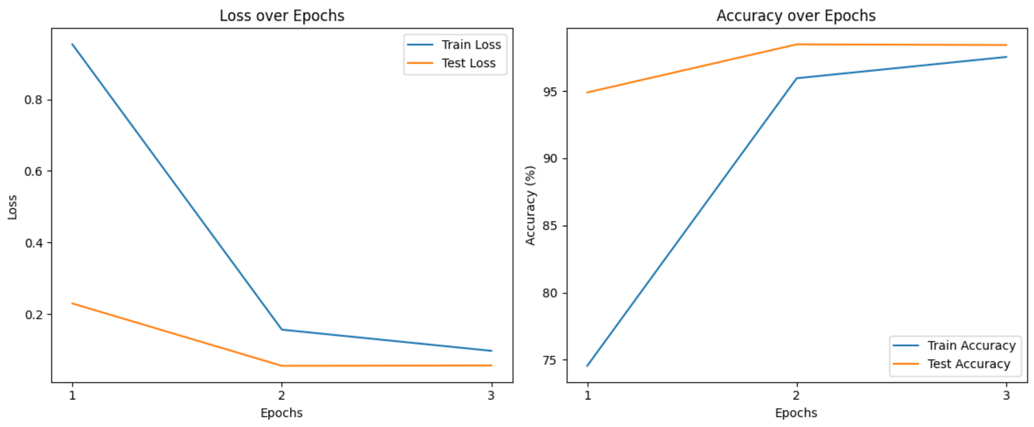

Figure 1: Accuracy and Loss of Custom Model

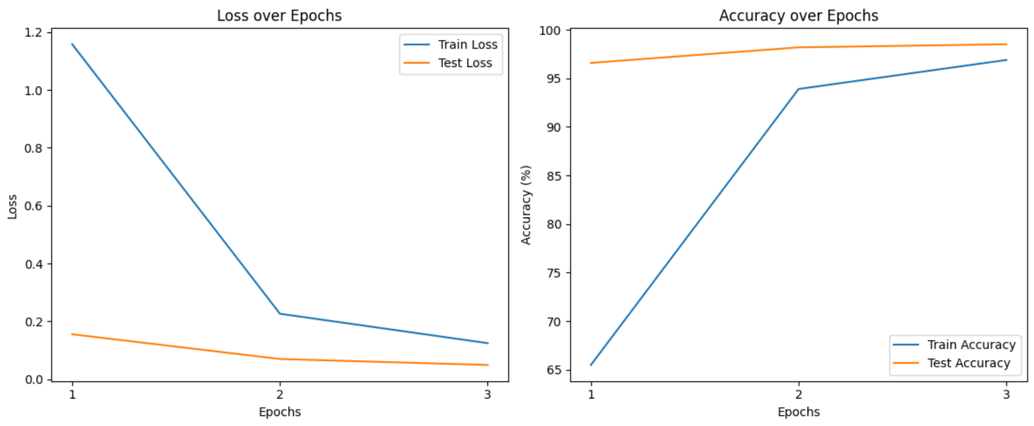

Figure 2: Accuracy and Loss of Built-in Model

4.5 Comparison of Custom Functions vs. PyTorch Built-in Components

Custom Functions:

- Final Test Accuracy: 98.42%

Baseline (PyTorch's built-in components):

- Final Test Accuracy: 98.51%

Comparing the final test accuracies, we can see that the baseline using PyTorch's built-in components slightly outperformed the custom functions implementation by 0.09 percentage points.

The custom components affected the model's performance in the following ways:

1. **Initial Performance:** In the first epoch, the custom functions achieved higher train and test accuracies (74.55% and 94.89% respectively) compared to the baseline (65.49% and 96.59%). This suggests that the custom implementation may have led to faster initial learning.

2. **Convergence:** Both implementations showed rapid improvement in the second epoch, but the custom functions reached a higher train accuracy (95.95% vs 93.90%) while the baseline achieved a better test accuracy (98.19% vs 98.47%).

3. **Final Performance:** By the third epoch, the baseline slightly surpassed the custom functions in both train and test accuracies. The custom functions showed signs of potential overfitting, with a higher train accuracy (97.53% vs 96.89%) but lower test accuracy (98.42% vs 98.51%).

4. **Loss Behavior:** The custom functions consistently achieved lower training loss values across all epochs, which didn't necessarily translate to better test performance. This could indicate that the custom implementation might be more prone to overfitting.

5. **Stability:** The baseline implementation showed more consistent improvement in test accuracy across epochs, while the custom functions had a slight decrease in the final epoch (98.47% to 98.42%). This suggests that the baseline might offer more stable performance.

5 Conclusion and Future Work

5.1 Key Findings and Insights

1. **Custom Implementation Performance:** The custom VGG16 implementation using custom components (CustomDropout, CustomBatchNorm2d, CustomReLU, CustomMaxPooling2d) achieved a final test accuracy of 98.42% after 3 epochs. This demonstrates that the custom

components were able to effectively learn and generalize on the MNIST dataset.

2. **Rapid Convergence:** The model showed quick improvement, with test accuracy jumping from 94.89% in the first epoch to 98.47% in the second epoch. This indicates that the custom implementations, particularly the CustomBatchNorm2d, may be contributing to faster learning.

3. **Slight Overfitting:** There's a small sign of overfitting as the training accuracy (97.53%) is higher than the test accuracy (98.42%) in the final epoch. However, the gap is relatively small, suggesting that the model generalizes well.

4. **Effective Custom Components:** The custom implementations of dropout, batch normalization, ReLU, and max pooling were able to work together effectively within the VGG16 architecture, demonstrating that these fundamental neural network components can be successfully implemented from scratch.

5.2 Areas for Improvement

1. **CustomMaxPooling2d Edge Cases:** The current implementation could be improved to handle edge cases more robustly, particularly when input dimensions are smaller than the kernel size. Adding more comprehensive input validation and adjusting the pooling strategy for small inputs would enhance its versatility.

2. **Optimization for Large Datasets:** The current implementation might face performance issues with larger datasets or deeper networks. Optimizing the custom components for speed and memory efficiency would be beneficial for scaling to more complex tasks.

3. **CustomDropout Implementation:** While functional, the CustomDropout class could be enhanced to use a more memory-efficient approach, possibly by using in-place operations to modify the input tensor directly.

4. **Learning Rate Scheduling:** Implementing a learning rate scheduler could potentially improve convergence and final accuracy, especially for longer training runs.

5. **Data Augmentation:** Adding data augmentation techniques could help improve generalization and potentially push the test accuracy even higher, especially given the relatively small size of the MNIST dataset.

6. **Handling of Different Input Sizes:** The current implementation assumes a fixed input size (224x224). Making the network more flexible to handle various input sizes would increase its applicability to different datasets.

7. **Comparative Analysis:** Conducting a more thorough comparison between the custom implementations and PyTorch's built-in components in terms of both performance and computational efficiency would provide valuable insights into the effectiveness of the custom implementations.

These improvements would further enhance the robustness and efficiency of the custom VGG16 implementation, potentially leading to better performance on more complex datasets and tasks.