Hyperparameter Tuning and Custom Batch Normalization for ResNet-18 on CIFAR-10

Written by Kusalavan Kirubhaharini (G2503701C)

Abstract

This report investigates the impact of hyperparameter tuning on ResNet-18 performance for CIFAR-10 image classification. Four key aspects are systematically evaluated: learning rate selection, learning rate scheduling, weight decay regularization, and custom batch normalization implementation. The experiments examine how each hyperparameter influences training and generalization capability. A custom batch normalization layer is implemented where mean and variance statistics do not participate in gradient computation, demonstrating the critical role of gradient flow through normalization parameters. The optimal configuration achieves 95.06% validation accuracy and 95.10% test accuracy using learning rate 0.01, cosine annealing schedule, and weight decay 1×10^{-2}.

Introduction

Residual Networks (ResNets) mark a significant innovation in deep convolutional neural network architecture. (He et al. 2016). However, achieving optimal performance requires hyperparameter configuration. Learning rate, learning rate scheduler, and regularization techniques significantly impact both convergence and generalization capability.

This report investigates four critical aspects of ResNet-18 training on CIFAR-10 (Krizhevsky and Hinton 2009): learning rate selection, learning rate scheduling mechanisms, weight decay regularization, and batch normalization gradient flow properties. Through controlled experimentation, each hyperparameter's contribution to overall performance is isolated and quantified. Additionally, a custom batch normalization implementation with modified gradient dynamics is developed and evaluated.

The remainder of this report is organized as follows: Section 2 describes the ResNet-18 architecture and experiment configuration. Sections 3 to 6 present detailed results and analysis for each hyperparameter study. Section 7 discusses model selection methodology and final test set evaluation. Section 8 concludes with key findings.

Description of ResNet

ResNet Architecture and Innovation

Residual Networks address the degradation problem observed in deep neural networks through the introduction of skip connections. These connections enable direct information to flow across layers (He et al. 2016). Prior to ResNet, architectures such as VGG (Simonyan and Zisserman 2014) relied exclusively on sequential layer stacking, which caused training difficulties as network depth exceeded 16-19 layers due to vanishing gradient problems.

The fundamental innovation of ResNet is the residual block, which learns a residual mapping $F(x) = H(x) - x$ rather than attempting to learn the desired underlying mapping $H(x)$ directly. The skip connection allows the network to learn identity mappings when appropriate, substantially simplifying the optimization of very deep networks.

Comparison: VGG-18 vs ResNet-18

The key architectural differences between VGG-18 and ResNet-18 are highlighted below:

Skip Connections: VGG-18 has no skip connections, relying entirely on sequential information flow. ResNet-18 introduces residual skip connections that bypass convolutional layers, allowing gradients to flow directly through the network.

Gradient Flow: In VGG-18, gradients must propagate sequentially through all layers during backpropagation, leading to vanishing gradients in deep networks. ResNet-18's skip connections provide direct gradient pathways, enabling stable training of much deeper architectures.

Depth Scaling: VGG-18 struggles with vanishing gradients when depth increases beyond 16-19 layers, making it difficult to train deeper variants. ResNet-18 uses identity mappings through skip connections, allowing easy scaling to hundreds of layers without degradation.

Parameters: VGG-18 contains approximately 138 million parameters, primarily due to large fully connected layers (FC 4096 $\rightarrow$ FC 4096 $\rightarrow$ FC 10). ResNet-18 has only about 11 million parameters by replacing these with global average pooling followed by a single FC layer (GAP $\rightarrow$ FC 10), reducing parameters by over 90%.

Pooling Strategy: VGG-18 uses max pooling five times throughout the network to progressively downsample feature maps. ResNet-18 has strided convolutions for downsampling and uses global average pooling only once at the end to aggregate spatial information.

Final Layer Design: VGG-18's final layers consist of two large fully connected layers (4096 dimensions each) before the classification layer, contributing heavily to the parameter count. ResNet-18 uses global average pooling to reduce the 4×4×512 feature map to a 512-dimensional vector, followed directly by a single FC layer for classification.

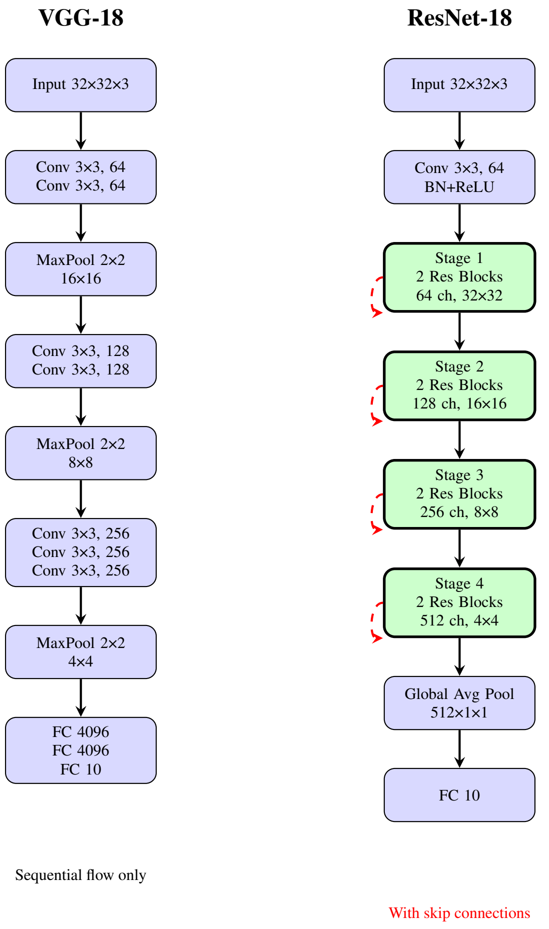

Figure 1: Architecture comparison: VGG-18 (left) vs ResNet-18 (right). Red dashed lines represent skip connections enabling direct gradient flow in ResNet.

Detailed Residual Block Structure

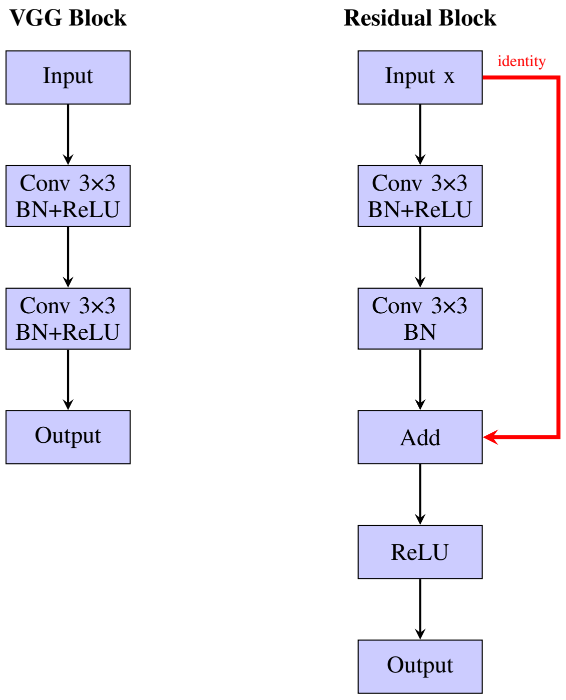

Figure 2: Block-level comparison. VGG blocks learn the direct mapping $H(x)$ through sequential convolutions, while residual blocks learn the residual function $F(x) = H(x) - x$ and reconstruct the final output via $H(x) = F(x) + x$ through a skip connection.

ResNet-18 Complete Structure

ResNet-18 for CIFAR-10 consists of:

- **Initial Convolution:** 3×3 conv (64 filters, stride 1, padding 1) followed by BatchNorm and ReLU, maintaining 32×32 resolution.
- **Residual Stages:**
  - Stage 1 consists of two residual blocks with 64 feature maps and a stride of 1, maintaining a spatial size of 32×32.
  - Stage 2 includes two blocks with 128 channels, using a stride of 2 to reduce the feature map to 16×16.
  - Stage 3 contains two blocks with 256 channels, again downsampling with a stride of 2 to 8×8.
  - Stage 4 applies two blocks with 512 channels, further reducing the resolution to 4×4.

  Each block has two 3×3 convolutions with BatchNorm; the input is added via a skip connection before the final ReLU.
- **Global Average Pooling:** Reduces the $4\times4\times512$ feature map to 512 dimensions.
- **Fully Connected Layer:** Maps 512 features to 10 CIFAR-10 classes.

Advantages over VGG-style networks:

1. Fewer parameters ($\sim$11M vs $>$100M) using global average pooling instead of large FC layers.
2. Skip connections enable stable gradient flow in deep networks.
3. Identity shortcuts simplify optimization and improve convergence.
4. Stronger generalization despite greater depth.

Experimental Configuration

Dataset: CIFAR-10 with 60,000 (32×32) images (10 classes) split into 40,000 training, 10,000 validation, and 10,000 test data.

Augmentation: Random crop with 4-pixel padding and random horizontal flip. Normalization uses per-channel mean (0.4914, 0.4822, 0.4465) and std (0.2023, 0.1994, 0.2010).

Training Setup: SGD with momentum 0.9, batch size 128, and cross-entropy loss on GPU. Learning rate experiments were conducted for 15 epochs; all subsequent experiments (scheduler, weight decay, custom BN) conducted for 300 epochs.

Learning Rate

Learning rate controls parameter update magnitudes during optimization. These values were evaluated: 0.001, 0.01, and 0.1, training for 15 epochs each.

Results

Table 1 shows learning rate 0.01 achieves best validation accuracy at 85.82%.

Table 1: Learning Rate Comparison After 15 Epochs

LR	Train Acc	Val Acc	Train Loss	Val Loss
0.001	85.93	80.28	0.4049	0.5868
0.01	**91.05**	**85.82**	**0.2558**	**0.4320**
0.1	87.20	84.54	0.3658	0.4607

Analysis

Figure 3 shows that training initially converges quickly but later becomes unstable, as large updates cause the optimizer to overshoot the optimum.

LR=0.001 improves steadily but slowly, reaching only 80.28% validation accuracy after 15 epochs. While stable, it is inefficient for the 15-epochs.

LR=0.01 balances speed and stability, achieving 91.05% training and 85.82% validation accuracy with lowest validation loss (0.4320). The curves show smooth, rapid convergence without instability.

This aligns with optimization theory: excessively large rates cause oscillation, while overly small rates result in slow convergence.

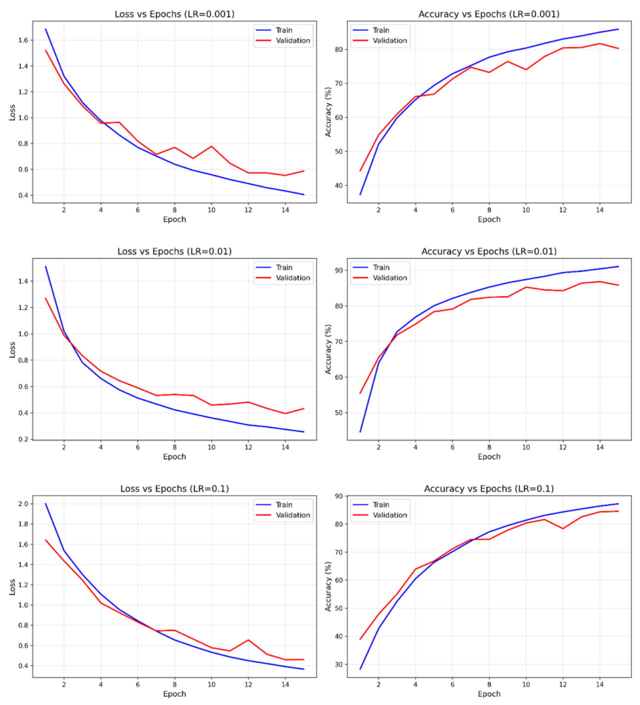

Figure 3: Training and validation curves for different learning rates. LR=0.01 achieves optimal balance between convergence speed and stability.

Learning Rate Scheduler

Learning rate schedules adaptively reduce learning rate during training, potentially improving convergence. The comparison between constant rate against cosine annealing over 300 epochs was done using LR=0.01. All other settings were constant.

Cosine Annealing Schedule

Cosine annealing reduces learning rate following a cosine curve:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right) \quad (1)$$

where $\eta_{max} = 0.01$ (initial learning rate), $\eta_{min} \approx 0$ (minimum learning rate), T_{cur} is the current epoch, and $T_{max} = 300$ is the total number of epochs.

Intuitive Explanation: The cosine function creates a smooth, wave-like decay pattern. At training start ($T_{cur} = 0$), the cosine term equals 1, so $\eta_t = \eta_{max} = 0.01$. At the midpoint ($T_{cur} = 150$), the cosine term equals 0, giving $\eta_t = \frac{\eta_{max}}{2} \approx 0.005$. At training end ($T_{cur} = 300$), the cosine term equals -1, yielding $\eta_t = \eta_{min} \approx 0$.

The cosine annealing schedule slowly reduces the learning rate in a smooth curve, preventing sudden drops that might disrupt training. Unlike step decay, which changes the rate in sharp intervals, cosine annealing makes the transition gradual and predictable. This helps the model explore the parameter space more broadly at the start with higher learning rates, then fine-tune later as the rate decreases. Because of the cosine shape, the learning rate falls faster in the early stages and slows down toward the end, giving the model more time to refine its performance.

Results

Table 2 compares performance with and without cosine annealing learning rate scheduler.

Table 2: Learning Rate Scheduler Comparison After 300 Epochs

Config	Train Acc	Val Acc	Train Loss	Val Loss
No Scheduler	99.95	92.38	0.0016	0.5071
Cosine Anneal	**100.00**	**92.59**	**0.0001**	**0.4547**

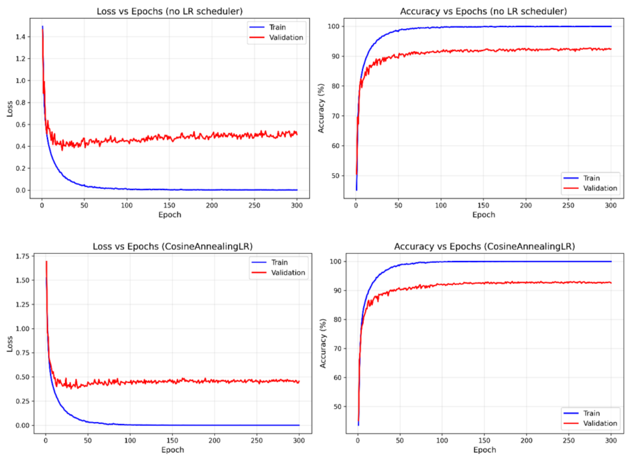

Figure 4: Comparison of training with and without cosine annealing. The scheduler achieves lower validation loss through fine-grained late-stage optimization.

Analysis

The constant learning rate flattens out around epoch 50, as the optimizer tends to oscillate near a minimum. In contrast, the gradual decay of cosine annealing allows smaller, more precise updates and smoother convergence. Higher learning rates early in training help the model explore quickly, while the slower decay later on lets it fine-tune more effectively and reach better generalization.

Weight Decay

Weight decay (L2 regularization) penalizes large parameters, encouraging simpler models. Weight decay values of 5×10^{-4} and 1×10^{-2} were compared using LR=0.01 with cosine annealing for 300 epochs.

Results

Table 3 presents results for both weight decay values.

Table 3: Weight Decay Comparison After 300 Epochs

Weight Decay	Train Acc	Val Acc	Train Loss	Val Loss
5×10^{-4}	**100.00**	94.12	**0.0012**	0.2448
1×10^{-2}	**100.00**	**95.06**	0.0229	**0.1919**

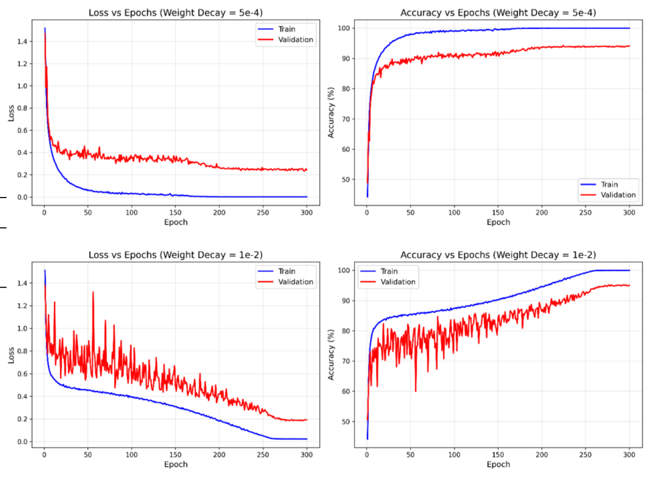

Figure 5: Weight decay comparison showing that stronger regularization (WD=1×10^{-2}) achieves better validation performance despite higher training loss.

Analysis

Both models reach 100% training accuracy, but validation performance diverges. Stronger regularization (1×10^{-2}) yields higher training loss (0.0229 vs 0.0012) yet better validation accuracy (95.06% vs 94.12%) and lower validation loss (0.1919 vs 0.2448), which is a 21.6% drop. The smaller weight decay (5×10^{-4}) fits training data smoothly but overfits slightly, while the larger value adds volatility yet generalizes better. This shows that accepting minor training degradation improves robustness and overall generalization.

Custom Batch Normalization

Standard batch normalization (Ioffe and Szegedy 2015) normalizes activations using batch statistics. A custom variant where mean and variance do not participate in gradient computation was implemented.

The key implementation detail is using `detach()` on batch_mean and batch_var, preventing PyTorch from building computational graph nodes for these operations.

```python
# ============================================================
# Custom Batch Normalization Implementation
# ============================================================
class CustomBatchNorm2d(nn.Module):
    """Custom BN where statistics are detached from gradient
        flow"""
    def __init__(self, channels, eps=1e-5, mom=0.1):
        super(CustomBatchNorm2d, self).__init__()
        self.channels = channels
        self.eps = eps
        self.mom = mom

        # Learnable scale and shift parameters
        self.scale = nn.Parameter(torch.ones(channels))
        self.shift = nn.Parameter(torch.zeros(channels))

        # Non-trainable buffers for running statistics
        self.register_buffer('run_mean', torch.zeros(channels))
        self.register_buffer('run_var', torch.ones(channels))
        self.register_buffer('batch_count', torch.tensor(0,
             dtype=torch.long))

    def forward(self, inp):
        # inp: (batch, channels, height, width)
        if self.training:
            # Calculate batch-wise statistics per channel
            mean_batch = inp.mean(dim=[0, 2, 3])
            var_batch = inp.var(dim=[0, 2, 3], unbiased=False)

            # CRITICAL: Detach to block gradient propagation
            mean_batch = mean_batch.detach()
            var_batch = var_batch.detach()

            # Exponential moving average for inference
                 statistics
            with torch.no_grad():
                self.run_mean = (1 - self.mom) * self.run_mean
                     + self.mom * mean_batch
                self.run_var = (1 - self.mom) * self.run_var +
                     self.mom * var_batch
                self.batch_count += 1

            # Standardize: (input - mean) / sqrt(variance +
                 epsilon)
            normalized = (inp - mean_batch.view(1, -1, 1, 1))
                 / torch.sqrt(var_batch.view(1, -1, 1, 1) +
                 self.eps)
        else:
            # Evaluation mode: use accumulated running
                 statistics
            normalized = (inp - self.run_mean.view(1, -1, 1,
                 1))/torch.sqrt(self.run_var.view(1, -1, 1,
                 1) + self.eps)

        # Affine transform: scale * normalized + shift
        result = self.scale.view(1, -1, 1, 1) * normalized +
             self.shift.view(1, -1, 1, 1)
        return result
# ============================================================
# Residual Block Using Custom Batch Normalization
# ============================================================
class ResidualBlock(nn.Module):
    expansion = 1
    def __init__(self, in_ch, out_ch, stride=1):
        super(ResidualBlock, self).__init__()

        # First convolution pathway
        self.conv_1 = nn.Conv2d(in_ch, out_ch, kernel_size=3,
             stride=stride, padding=1, bias=False)
        self.norm_1 = CustomBatchNorm2d(out_ch)

        # Second convolution pathway
        self.conv_2 = nn.Conv2d(out_ch, out_ch, kernel_size=3,
             stride=1, padding=1, bias=False)
        self.norm_2 = CustomBatchNorm2d(out_ch)

        # Identity shortcut or projection
        self.skip = nn.Sequential()
        if stride != 1 or in_ch != self.expansion * out_ch:
            self.skip = nn.Sequential(nn.Conv2d(in_ch,
                 self.expansion * out_ch,
                 kernel_size=1,stride=stride,
                 bias=False),CustomBatchNorm2d(self.expansion
                 * out_ch)
            )

    def forward(self, inp):
        residual = F.relu(self.norm_1(self.conv_1(inp)))
        residual = self.norm_2(self.conv_2(residual))
        residual += self.skip(inp)  # Add skip connection
        return F.relu(residual)
# ============================================================
# ResNet Architecture with Custom BatchNorm
# ============================================================
class ResNet(nn.Module):
    def __init__(self, block_type, layer_config, n_classes=10):
        super(ResNet, self).__init__()
        self.current_channels = 64
        # Initial convolution (stem layer)
        self.stem_conv = nn.Conv2d(3, 64, kernel_size=3,
                                   stride=1, padding=1,
                                        bias=False)
        self.stem_norm = CustomBatchNorm2d(64)
        # Four residual stages with progressive downsampling
        self.stage_1 = self._build_stage(block_type, 64,
                                         layer_config[0],
                                              stride=1)
        self.stage_2 = self._build_stage(block_type, 128,
                                         layer_config[1],
                                              stride=2)
        self.stage_3 = self._build_stage(block_type, 256,
                                         layer_config[2],
                                              stride=2)
        self.stage_4 = self._build_stage(block_type, 512,
                                         layer_config[3],
                                              stride=2)


        # Classification head
        self.classifier = nn.Linear(512 *
             block_type.expansion, n_classes)

    def _build_stage(self, block_type, out_ch, n_blocks,
         stride):
        stride_list = [stride] + [1] * (n_blocks - 1)
        stage_layers = []
        for s in stride_list:
            stage_layers.append(block_type(self.current_channels,
                 out_ch, s))
            self.current_channels = out_ch *
                 block_type.expansion
        return nn.Sequential(*stage_layers)

    def forward(self, inp):
        features = F.relu(self.stem_norm(self.stem_conv(inp)))
        features = self.stage_1(features)
        features = self.stage_2(features)
        features = self.stage_3(features)
        features = self.stage_4(features)
        features = F.avg_pool2d(features, 4)
        features = features.view(features.size(0), -1)
        logits = self.classifier(features)
        return logits

def ResNet18():
    return ResNet(ResidualBlock, [2, 2, 2, 2])
```

Mathematical Formulation

Standard batch normalization for mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$:

Batch Statistics:

$$\mu_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma_\mathcal{B}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \tag{2}$$

Normalization:

$$\hat{x}_i = \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \tag{3}$$

Affine Transformation:

$$y_i = \gamma \hat{x}_i + \beta \tag{4}$$

where γ (scale) and β (shift) are learnable.

In standard BN, gradients flow through all operations, including $\mu_\mathcal{B}$ and $\sigma_\mathcal{B}^2$. My custom implementation detaches these statistics using PyTorch's `detach()` function, preventing gradient flow through mean and variance computation while preserving gradients to input x and parameters γ, β.

Results

Table 4 shows the results of custom batch normalization.

Table 4: Custom Batch Normalization Results After 300 Epochs

Configuration	Train Acc	Val Acc	Train Loss	Val Loss
Custom BN	21.95	21.65	1.9707	1.9775

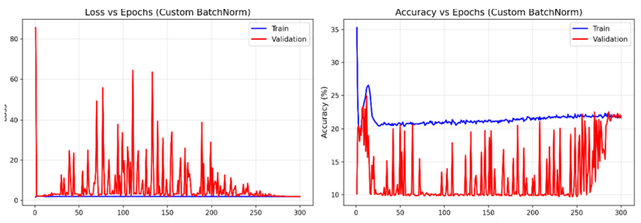

Figure 6: Custom batch normalization training curves showing failure to learn. Accuracy plateaus around 22% , and loss remains near initial values throughout training.

Analysis

Figure 6 shows the train and validation instabilities. The training curves show severe instability throughout 300 epochs. Training loss fluctuates from near zero to above 80, and accuracy varies between 10% and 25%. Validation accuracy remains steady around 22%, close to random guessing for CIFAR-10. Final loss values near 2.0 (close to $-\log(0.1) \approx 2.3$) confirm that the model outputs nearly uniform predictions.

These results indicate that removing gradient flow through batch statistics completely disrupts training stability. The network fails to maintain consistent gradient updates, leading to three main problems:

Gradient Instability: Without gradients flowing through the mean and variance, updates become uncontrolled, causing repeated spikes and collapses in loss.

Unstable Optimization: Detaching statistics removes the smoothing effect of batch normalization, creating a highly irregular loss surface where the optimizer overshoots and undershoots minima.

Broken Gradient Coordination: Normal batch statistics align gradients across samples, but detaching them causes conflicting updates that prevent coherent learning.

The near-identical train and validation accuracies (21.95% vs 21.65%) show that the network learns nothing meaningful. This confirms that gradient flow through batch statistics is essential for stable optimization and effective learning in deep networks.

Model Selection

Comparison Across Experiments

Table 5 summarizes validation performance across all configurations.

Table 5: Model Selection Summary

Configuration	Val Acc	Val Loss
Initial (LR=0.01)*	85.82	0.4320
Baseline - No Scheduler (LR=0.01)	92.38	0.5071
Cosine Annealing	92.59	0.4547
WD 5×10^{-4}	94.12	0.2448
WD 1×10^{-2}	**95.06**	**0.1919**
Custom BatchNorm	21.65	1.9775

*Initial trained for 15 epochs; all others for 300 epochs

We use validation set performance as the model selection criterion, choosing the model with highest validation accuracy as this indicates best generalization to unseen data. The progression reveals cumulative benefits of proper hyperparameter selection. Learning rate 0.01 establishes a strong performance (85.82% after 15 epochs). Extending training to 300 epochs without scheduling reaches 92.38% (baseline). Adding cosine annealing provides modest improvement to 92.59%. Introducing weight decay 5×10^{-4} yields substantial gains to 94.12%. Finally, stronger weight decay 1×10^{-2} achieves 95.06% validation accuracy. This is a 2.68% improvement in validation accuracy compared to the baseline. It also has the lowest validation loss of 0.1919.

Based on validation performance, the selected model for final evaluation was the one trained with learning rate 0.01, cosine annealing schedule, and weight decay 1×10^{-2}. This configuration demonstrates the best balance between training efficiency and generalization capability. The best model during training with these settings was at epoch 291, with a validation accuracy of 95.29%. This model was saved for the final test data evaluation in the next section.

Final Test Set Evaluation

Following standard practice, the test set was exclusively reserved for final model evaluation. This ensures test performance provides an unbiased estimate of real-world generalization.

The selected model achieves:

Test Accuracy: 95.10%

Test Loss: 0.1846

The test accuracy closely matches validation performance (95.06%), confirming proper model selection without overfitting to the validation set. This performance represents the expected accuracy on unseen CIFAR-10 images. The close alignment between validation and test metrics validates the experimental methodology and hyperparameter selection process.

Conclusion

This report has explored and tuned the ResNet-18 model for the CIFAR-10 dataset through controlled experiments. The main observations are as follows:

Learning Rate: A moderate rate of 0.01 provided the best balance between convergence speed and stability.

Scheduling: Cosine annealing performed better than a constant rate, as its gradual decay allowed smoother and more precise optimization.

Weight Decay: Stronger regularization (1×10^{-2}) improved generalization even though it led to a slightly higher training loss.

Batch Normalization: Detaching batch statistics completely disrupted training, confirming that gradient flow through normalization layers is essential for stable learning.

Using the optimal settings, the model reached 95.06% validation accuracy and 95.10% test accuracy, improving by 9.24 percentage points over the initial model (85.82% from the 15-epoch LR=0.01 tuning).

Although the custom batch normalization experiment failed to train successfully, it provided valuable insight into how gradient flow supports deep learning optimization. This process of isolating and testing each hyperparameter proved to be an effective and practical way to fine-tune deep learning models.

References

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Ioffe, S.; and Szegedy, C. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, 448–456.

Krizhevsky, A.; and Hinton, G. 2009. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto.

Simonyan, K.; and Zisserman, A. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*.