ESE 3060 Final Project - Part 1

Your Name

December 6, 2025

1 Proposed Changes

There are two main ways that we can improve the speed of training while maintaining a benchmark accuracy of 94.0. We either need better data, so the model trains better per epoch, or we need to make the training faster.

Our proposed changes are a combination of the two.

1. **Static Precomputation of Image Transforms** – Precomputed normalization, flipping, and padding operations once during `CifarLoader` initialization instead of recomputing every epoch. Static transforms are stored in `self.proc_images` and reused across epochs, eliminating redundant computation while maintaining the same augmentation behavior.
2. **Residual Connections in ConvGroup** – Added skip connections to each `ConvGroup` block with a 1x1 convolution downsample path. The residual pathway enables better gradient flow and faster convergence by allowing the network to learn identity mappings: $output = \text{GELU}(\text{conv\_path}(x) + \text{downsample}(\text{pool}(x)))$. Based on the case example, but it actually made training faster.
3. **Hyperparameter Optimization** – Reduced training epochs from 9.9 to 8.0, increased batch size from 1024 to 1536 (50% increase), and scaled learning rate proportionally from 11.5 to 17.25. The larger batch size improves GPU utilization while the reduced epochs leverage faster convergence from residual connections.
4.

A Appendix

A.1 Change 1: Static Precomputation of Image Transforms

Hypothesis: The original implementation recomputes normalization, flipping, and padding operations every epoch during training. Since normalization and flipping are deterministic transformations that don't depend on random seeds per batch, we can precompute these once during initialization and reuse them across all epochs, eliminating redundant computation.

What Changed: Modified the `CifarLoader.__init__` method to precompute normalized images, flipped images, and padded images once during initialization (lines 135-150 in `airbench-lucky.py`). The `__iter__` method now selects from these precomputed tensors instead of recomputing them each epoch.

Solution: All static transforms (normalization, flipping, padding) are computed once in the `__init__` method within a `torch.no_grad()` context. The normalized images are stored in `self.proc_images['norm']`, flipped images in `self.proc_images['flip']`, and padded images in `self.proc_images['pad']`. During iteration, we only perform the dynamic random crop operation on the precomputed padded tensor, and apply epoch-based flipping by selecting from precomputed tensors. This eliminates redundant normalization and padding operations that were previously executed every epoch.

A.2 Change 2: Residual Connections in ConvGroup

Hypothesis: Adding residual (skip) connections to the ConvGroup blocks will enable deeper gradient flow during backpropagation, potentially allowing the network to learn identity mappings more easily and converge faster. This architectural pattern has been shown to improve training dynamics in deep networks.

What Changed: Added a residual connection pathway to the `ConvGroup` class (lines 241-244, 257-260). The residual path consists of a 1x1 convolution with BatchNorm to match channel dimensions, applied after pooling the identity tensor. The final output is the sum of the main path and the residual path, followed by activation.

Solution: Each `ConvGroup` now includes a `downsample` module that projects the input to match the output channel dimensions. In the forward pass, the identity is preserved, pooled, and passed through the downsample module. The main convolutional path output is then added to this processed identity before the final activation. This creates a residual learning pathway: $\textit{output} = \text{GELU}(\text{conv\_path}(x) + \text{downsample}(\text{pool}(x)))$.

A.3 Change 3: Hyperparameter Optimization

Hypothesis: With the improved architecture (residual connections) and data loading efficiency (static precomputation), we can achieve similar or better accuracy with fewer epochs. Additionally, increasing batch size can improve GPU utilization and training speed, while maintaining training dynamics by scaling the learning rate proportionally.

What Changed: Reduced `train_epochs` from 9.9 to 8.0, increased `batch_size` from 1024 to 1536 (50% increase), and scaled `lr` from 11.5 to 17.25 (proportional to batch size increase: $11.5 \times 1.5 = 17.25$).

Solution: The reduced epoch count leverages faster convergence enabled by residual connections. The larger batch size improves GPU memory utilization and reduces the number of gradient updates per epoch. The learning rate is scaled linearly with batch size to maintain equivalent gradient step magnitudes, following the principle that effective learning rate should scale with batch size when using the same number of epochs.

A.4 Change 4: Optimized Data Loading Pipeline

Hypothesis: Moving data to GPU earlier and using non-blocking transfers can overlap data transfer with computation, reducing overall training time. Additionally, ensuring labels are properly typed and on the correct device avoids runtime type conversions.

What Changed: Modified data loading to use explicit device handling (line 100), convert images to GPU-friendly format immediately (lines 115-121), and ensure labels are properly typed as long integers on the correct device (line 111). Used `non_blocking=True` for asynchronous transfers.

Solution: Images are converted to half precision, normalized, permuted to channels-last format, and moved to GPU in a single pipeline during initialization. Labels are explicitly cast to `long` and moved to the device. This ensures all data is in the optimal format before training begins, eliminating per-batch conversion overhead.