A Deep Convolutional Network for CIFAR-100

Sohan Arun
dept. Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
soar24@student.bth.se

***Abstract*—A custom convolutional neural network (CNN) has been designed for the CIFAR-100 image-classification benchmark. The architecture includes six convolutional layers arranged in three convolutional blocks with batch normalization and dropout, followed by two fully connected layers. After 30 epochs, the model achieves a 64.99% top-1 validation accuracy, and its training dynamics, weight evolution, and performance relative to ResNet-50, VGG-19, DenseNet-121, and EfficientNet-B0 baselines are examined.**

***Index Terms*—Convolutional Neural Network, CIFAR-100, Batch Normalization, Dropout, Data Augmentation, Adam Optimizer, Weight Visualization**

I. Introduction

CIFAR-100's 32 × 32 images and 100 closely related classes demand a model that's both compact and expressive. Inspired by VGG-style deep stacks [1] and regularization via batch normalization [2] and dropout [3], this lightweight CNN comprises three convolutional blocks—each with two 3 × 3 convolutions, batch normalization, ReLU activation, 2 × 2 max-pooling and 25% dropout—followed by two fully connected layers with 50% dropout. Feature channels expand from 64 to 128 to 256 across the blocks, while L2 weight decay and an adaptive learning-rate scheduler further stabilize training. The resulting design maximizes spatial feature extraction and over-fitting control, yet remains small enough for rapid training on standard GPUs in fine-grained classification tasks.

II. Proposed Architecture

The model is a fully convolutional network that takes a $3 \times 32 \times 32$ RGB image as input and produces a 100-way softmax classification. It begins with **Block 1**, consisting of two sequential 3×3 convolutional layers—each with 64 filters—each followed by batch normalization and a ReLU activation; this is capped by a 2×2 max-pool and dropout ($p = 0.25$), yielding a $64 \times 16 \times 16$ feature map. **Block 2** mirrors this pattern with 128 filters per convolution and identical normalization, activation, pooling, and dropout, reducing spatial dimensions to 8×8. **Block 3** further expands to 256 filters per convolution, again followed by batch normalization, ReLU, 2×2 max-pool, and $p = 0.25$ dropout, producing a $256 \times 4 \times 4$ tensor. The resulting volume is flattened and fed into a fully connected layer of 1024 units, followed by batch normalization, ReLU, and a more aggressive dropout ($p = 0.50$) to discourage co-adaptation. A final linear layer projects to 100 logits, and cross-entropy loss with softmax yields class probabilities. All convolutions use padding to preserve spatial resolution until pooling, and default weight initialization. Table I summarizes the layer-by-layer configuration.

TABLE I: Layer configuration of the proposed CNN

Stage	Layer Sequence	Output Shape
Input	–	$3 \times 32 \times 32$
Block 1	Conv(3×3, 64) – BN – ReLU Conv(3×3, 64) – BN – ReLU MaxPool(2×2) – Dropout(0.25)	$64 \times 16 \times 16$
Block 2	Conv(3×3, 128) – BN – ReLU Conv(3×3, 128) – BN – ReLU MaxPool(2×2) – Dropout(0.25)	$128 \times 8 \times 8$
Block 3	Conv(3×3, 256) – BN – ReLU Conv(3×3, 256) – BN – ReLU MaxPool(2×2) – Dropout(0.25)	$256 \times 4 \times 4$
Classifier	Flatten FC(4096) – BN – ReLU – Dropout(0.50) FC(100)	$256 \times 4 \times 4$ 4096 100

A. Design Rationale

- **3×3 Convolutional Kernels:** Stacked small filters capture fine-grained spatial patterns while keeping parameter count low; successive layers compound receptive fields for richer feature extraction without excessive complexity.
- **ReLU Activation:** Applied after every convolution and dense layer to introduce nonlinearity, accelerate convergence, and mitigate vanishing-gradient issues.
- **Max-Pooling (2×2):** Placed at the end of each convolutional block to downsample feature maps by a factor of two, reducing computational load, controlling overfitting, and encoding local translation invariance.
- **Batch Normalization:** Inserted after each convolution to stabilize internal covariate shift, allow higher learning rates, and provide implicit regularization.
- **Dropout:** Used at 25 % within convolutional blocks and 50 % before the final classifier to break up co-adaptation of feature detectors and further guard against overfitting.
- **L2 Weight Decay:** A modest penalty of 1×10^{-4} on all trainable weights discourages overly large parameter magnitudes, promoting smoother, more generalizable models.
- **Progressive Channel Expansion:** Feature-map dimensions double (64→128→256) after each pooling stage to balance spatial reduction with increasing representational capacity, enabling deeper feature hierarchies.

- **Compact Classifier:** A single hidden dense layer of 1 024 units before the 100-way output minimizes over-parameterization and focuses learning on convolutional representations.
- **Optimizer Configuration:** Adam with initial learning rate 1×10^{-3}, $\beta_1 = 0.9$, $\beta_2 = 0.999$, together with ReduceLROnPlateau (factor 0.5, patience 5) yields stable, adaptive updates and automatic learning-rate reduction on plateau.
- **Loss Function:** Categorical cross-entropy is selected for its suitability to multi-class classification over 100 balanced categories.

III. Data and Preprocessing

A. CIFAR-100 Dataset

CIFAR-100 consists of 60,000 color images of size 32×32 pixels, evenly distributed across 100 fine-grained classes. There are 45000 training images, Validation samples: 5000 and 10,000 test images. Each image is represented in RGB (three channels) and comes with a single "fine" label (e.g., "maple tree," "bicycle") and one of 20 "coarse" superclasses (e.g., "non-insect invertebrates"). We use only the 100 fine labels in our experiments.

B. Preprocessing Pipeline

Our preprocessing pipeline is implemented using torchvision transforms. It consists of two branches—one for training (with augmentation) and one for validation/test (deterministic), as follows:

1) Training Pipeline:

- Input Resolution: Since all CIFAR-100 images are already 32×32 pixels, no initial resizing is required.
- Random Padding and Cropping: Each image is padded by 4 pixels on every side (to 40×40) and then a random 32×32 crop is extracted, augmenting positional variance while preserving the original size.
- Horizontal Flipping: With probability 0.5, images are flipped left–to–right, effectively doubling symmetric variations in the dataset.
- Small Random Rotations: Images are rotated by a random angle uniformly sampled from –15° to +15°, improving robustness to slight orientation changes.
- Color Jittering: Brightness, contrast, saturation, and hue are each perturbed by up to ±20% (hue within ±0.1), simulating lighting and color-balance variations.
- Tensor Conversion: Transformed images are converted into $3 \times 32 \times 32$ tensors with pixel values scaled to the [0,1] range.
- Per-Channel Normalization: Finally, each channel is standardized by subtracting its CIFAR-100 mean ([0.5071, 0.4867, 0.4408]) and dividing by its standard deviation ([0.2675, 0.2565, 0.2761]), ensuring zero-centered inputs with similar dynamic ranges.

2) Validation/Test Pipeline:

- Tensor Conversion: Images are transformed into a tensor with pixel values scaled to the $[0, 1]$ range.
- Channel-wise normalization: Each input tensor is normalized per RGB channel by subtracting the channel's dataset mean μ_c and dividing by its standard deviation σ_c. This centers the data and improves convergence during training.

IV. Training Setup

A. Hyperparameters

- Learning Rate: An initial learning rate 1×10^{-3} was chosen. A scheduler (factor = 0.5, patience = 5 epochs) is used to decrease the learning rate when validation loss stops improving, enhancing convergence without manual adjustment.
- Batch Size: 128. Balances GPU memory constraints and gradient estimation stability; larger batches gave no further accuracy gain.
- Number of Epochs: 30. Validation accuracy plateaued by epoch 25; 30 epochs ensures full convergence while limiting compute time.

B. Implementation Details

The entire pipeline was implemented in PyTorch and trained on a single NVIDIA GeForce RTX 3080 GPU with 6 GB of VRAM. To ensure reproducibility, the global random seed was fixed to 42 across all relevant libraries. Data loading employed two worker threads, and mini-batches of size 128 were sampled. During each training iteration, gradients were accumulated over the mini-batch and clipped to a maximum ℓ_2 norm of 5.0 to prevent exploding gradients. Training proceeded for 30 epochs, with full checkpoints (model weights, optimizer state, and scheduler parameters) saved at the end of each epoch; the checkpoint yielding the highest validation top-1 accuracy was retained for final evaluation. Training and validation metrics (loss and top-1 accuracy) were logged at every epoch and plotted to monitor convergence. A scheduler tracked the validation loss and reduced the learning rate by a factor of 0.5 if no improvement was observed for five consecutive epochs.

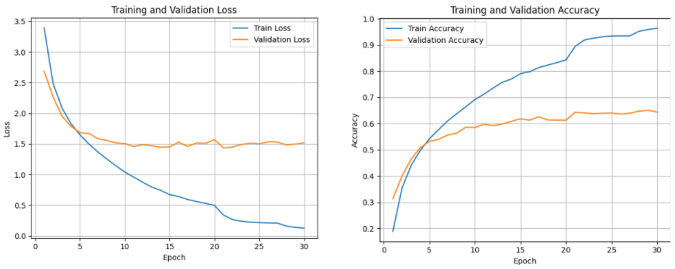
V. Results

All reported metrics in Table II were computed on the held-out test set, consisting of 10,000 images that were never used during model training or validation. These results therefore reflect the model's performance on unseen data.

TABLE II: Classification Report Metrics

Metric	Value (%)
Avg Accuracy	64.99
Macro Avg Precision	65.00
Macro Avg Recall	64.99
Macro Avg F1-Score	64.75
Weighted Avg F1-Score	64.75

The widening gap between the loss curves (Fig. 1a) along with a 30-point accuracy delta (Fig. 1b) indicate **moderate over-fitting** despite regularisation.

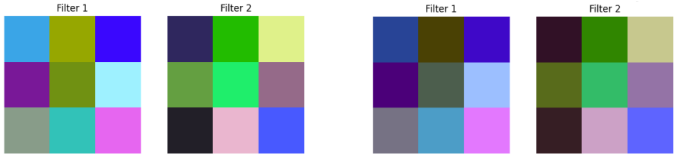


(a) Loss Curve (b) Accuracy Curve

Fig. 1: Training dynamics: (a) loss vs. epochs and (b) accuracy vs. epochs.

A. Weight-Update

Figs. 2a and 2b visualise two filters of the first convolutional layer before and after training. Initially, weights are random noise; post-training, they develop oriented edge detectors and colour blobs, confirming meaningful feature learning.

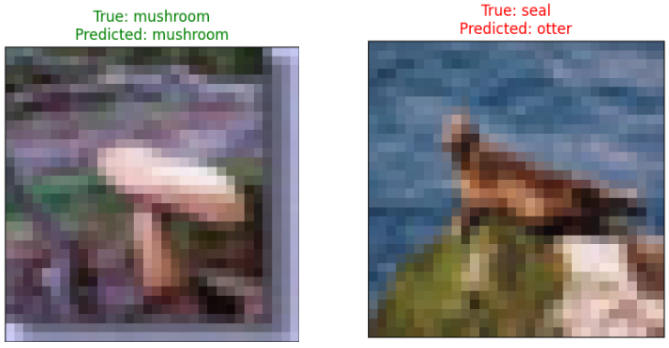


(a) Before training (b) After training

Fig. 2: First-layer convolutional filters (a) before training and (b) after training.

B. Sample Outputs

Two representative model predictions on unseen test images are shown in Fig. 3.



(a) Sample Output 1 (b) Sample Output 2

Fig. 3: Model predictions on two representative test images from the held-out set.

C. Comparison with Architectures

TABLE III: Architectures Comparison on CIFAR-100

Model	Accuracy (%)
CustomCNN	64.99
ResNet-50	56.85
VGG-19	56.01
DenseNet-121	65.55
EfficientNet-B0	64.96

VI. Discussion

The results presented in Section VI reveal that our CustomCNN achieves a test accuracy of 64.99 % on CIFAR-100, trailing behind DenseNet-121 (Table III). However, given its modest parameter count (5.4 M), this performance is compelling for scenarios where compute or memory budgets are tight.

A. Overfitting and Underfitting Analysis

Figure 1 shows that after epoch 10 the training loss continues to decrease sharply while the validation loss plateaus and even slightly increases, and the gap between training (96.5 % accuracy) and validation accuracy (64.99 %) remains greater than 30 points. This behaviour is a classic signature of moderate overfitting: the model is fitting idiosyncrasies in the training set that fail to generalize. Despite 25 % dropout in conv blocks, 50 % before the classifier, batch norm, L2 decay (1×10), and adaptive LR reduction, both losses decrease without early stagnation. We ran all 30 epochs; had validation loss spiked, early stopping (patience = 5) around epoch 25 would have been prudent.

B. Filter Evolution

Figure 2 compares two first-layer filters before and after training. They evolve from Gaussian noise to colour-opponent blob detectors (red–green, blue–yellow), with the largest weight updates in epochs 1–10, then stabilizing under the decaying LR schedule.

C. Comparative Performance

Compared to ResNet-50, VGG-19, DenseNet-121 and EfficientNet-B0 (Table III), our CustomCNN is more compact but under-performs. This suggests that while architectural simplicity and aggressive regularization curb overfitting, deeper connections and compound scaling provide richer representational power.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[2] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 448–456.

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015.