# Model Compression and Pruning for Efficient Deep Learning via Optimization Heuristics

Geetika Khanna   Archit Harsh   Hemal

May 3, 2025
MSML 604

**Abstract**

Modern deep learning models achieve state-of-the-art results but demand significant computational resources, hindering deployment on edge devices and increasing operational costs. Model pruning aims to reduce network complexity (parameters, computations) while maintaining performance. This report addresses the problem of finding an optimal pruned sub-network by formulating it as a constrained optimization problem involving binary variables to select weights. Due to the computational intractability of solving this Mixed Integer Non-Linear Program (MINLP) directly for large networks, we implement and evaluate two distinct numerical heuristic approaches: (1) Iterative Greedy Magnitude Pruning with Fine-tuning (iGreedy), stopping based on an accuracy threshold, and (2) L1 Regularization during training followed by weight thresholding and fine-tuning. We apply these methods to a Convolutional Neural Network (CNN) on the CIFAR-10 dataset using PyTorch, incorporating automatic device selection (MPS/CUDA/CPU) and robust error handling. We evaluate the trade-offs using metrics for accuracy, sparsity (parameter count), and estimated computational cost (FLOPs). Our results demonstrate that both heuristics can significantly compress the model, reducing parameters and FLOPs with controlled accuracy loss, highlighting their practical utility in generating efficient models suitable for resource-constrained environments.

## 1 Introduction

The success of Deep Neural Networks (DNNs) across various fields is undeniable. However, this success often comes at the cost of enormous model size and computational complexity. Models like VGG, ResNet, and large language models can have millions or billions of parameters, leading to significant challenges:

- **High Training & Inference Costs:** Requiring powerful GPUs and significant time.
- **Large Memory/Storage Footprint:** Making deployment on resource-constrained devices like mobile phones, IoT devices, or embedded systems difficult.
- **High Energy Consumption:** Contributing to operational costs and environmental concerns, especially in large data centers.
- **Latency Issues:** Slowing down real-time applications like autonomous driving or live video analysis.

Model compression techniques, particularly pruning, offer a promising solution. Pruning involves identifying and removing less important components (weights, neurons, filters) from a trained network to create a smaller, faster, and more energy-efficient equivalent without significantly sacrificing predictive accuracy. This project focuses on unstructured weight pruning, where individual weights can be removed regardless of their position.

## 2 Motivation

The drive towards efficient and ubiquitous AI motivates this exploration of model pruning:

- **Edge AI:** Enabling complex models to run directly on devices with limited power and computational budgets (e.g., smartphones, smart sensors).
- **Real-Time Performance:** Reducing inference latency for applications demanding quick responses (e.g., autonomous navigation, interactive services).
- **Sustainability:** Lowering the energy footprint associated with training and deploying large-scale AI models.
- **Cost Reduction:** Decreasing hardware requirements (memory, storage) and operational expenses for cloud-based deployments.
- **Understanding Redundancy:** Gaining insights into the inherent over-parameterization and redundancy often present within deep learning models.

This project aims to apply optimization concepts to systematically reduce model complexity, specifically targeting parameter count and computational load (measured via FLOPs), while explicitly controlling the impact on predictive accuracy.

## 3 Mathematical Optimization Formulation

The core problem is to select an optimal subset of weights to keep ($z_j = 1$) or remove ($z_j = 0$) from a pre-trained network $W = \{w_1, ..., w_N\}$ to minimize model complexity while ensuring performance doesn't degrade unacceptably. Let $z = \{z_1, ..., z_N\}$ be the binary mask variables associated with each weight $w_j$, where $W \odot z$ denotes the element-wise multiplication representing the pruned network.

Two primary ways to formulate this optimization goal are:

### 3.1 Formulation 1: Constrained Optimization (Minimize Size subject to Accuracy)

This formulation directly reflects the common practical goal: find the smallest possible model (fewest active parameters) that still meets a minimum performance standard.

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \sum_{j=1}^{N} z_j \\ \text{subject to} \quad & \text{Accuracy}(W \odot z) \geq A_{min} \\ & z_j \in \{0, 1\}, \quad \forall j \in \{1, ..., N\} \end{aligned} \tag{1}$$

Where:

- The objective is explicitly to minimize the number of **kept** parameters ($\sum z_j$).
- $A_{min}$ is the minimum acceptable accuracy threshold.

This formulation clearly states the objective but involves a complex, non-linear constraint related to accuracy.

### 3.2 Formulation 2: Penalized Objective (Minimize Loss + Sparsity Penalty)

This formulation, similar to the one presented in our mid-semester report, combines the task performance (loss) and model complexity (number of parameters) into a single objective function using a trade-off parameter $\lambda$.

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \mathcal{L}(W \odot z) + \lambda \sum_{j=1}^{N} z_j \\ \text{subject to} \quad & z_j \in \{0, 1\}, \quad \forall j \in \{1, ..., N\} \end{aligned} \tag{2}$$

Where:

- $\mathcal{L}(W \odot z)$ is the loss function (e.g., cross-entropy) evaluated on a dataset using the pruned network. Minimizing this term promotes accuracy.
- $\sum z_j$ is the total number of non-zero (kept) parameters. Minimizing this term promotes sparsity.
- $\lambda \geq 0$ is a hyperparameter controlling the trade-off. A larger $\lambda$ places more emphasis on reducing the number of parameters (increasing sparsity) relative to minimizing the loss.

This formulation directly balances the two competing goals within the objective function itself.

### 3.3 Challenge and Heuristics

Both formulations (1) and (2) result in a Mixed Integer Non-Linear Program (MINLP) due to the binary variables $z_j$ and the non-linear loss/accuracy functions. Solving such MINLPs optimally for the vast number of parameters ($N$) in neural networks is computationally intractable. Therefore, practical approaches rely on **heuristic numerical methods** that aim to find good, though not necessarily globally optimal, solutions relevant to these formulations.

## 4 Numerical Solvers Implemented

Given the difficulty of solving the MINLP optimally, we implement and compare two practical heuristic numerical methods, each conceptually linked to one of the formulations above:

### 4.1 Method 1: Iterative Greedy Magnitude Pruning with Fine-tuning (iGreedy Threshold)

- **Concept & Link to Formulation 1:** This method directly attempts to solve the constrained optimization problem (Formulation 1) heuristically. It iteratively increases sparsity (minimizing $\sum z_j$) by greedily removing the smallest magnitude weights and stops when the accuracy constraint (Accuracy $\geq A_{min}$) is about to be violated.
- **Algorithm:**
  1. Start with a trained dense model.
  2. Define $A_{min}$ and a pruning step amount $p$.
  3. **Loop:**
     (a) Calculate target global sparsity to remove $p\%$ of remaining weights.
     (b) Prune globally using L1 magnitude.
     (c) Fine-tune.
     (d) Evaluate accuracy $A_{current}$.
     (e) If $A_{current} \geq A_{min}$, save model and continue.
     (f) If $A_{current} < A_{min}$, stop; the last saved model is the result.
- **Pros:** Direct control over minimum accuracy, intuitive.
- **Cons:** Greedy, not guaranteed optimal, result depends on step size $p$.

### 4.2 Method 2: L1 Regularization during Training with Thresholding

- **Concept & Link to Formulation 2:** This method relates to the penalized objective (Formulation 2). The L1 penalty $\lambda' \sum |w_j|$ added during training encourages weights $w_j$ to shrink, implicitly minimizing the count of significant weights, similar to penalizing $\sum z_j$. The hyperparameter $\lambda'$ in the code corresponds conceptually to $\lambda$ in Formulation 2.
- **Algorithm:**

1. Train using L1-regularized loss: $\mathcal{L}_{total}(W) = \mathcal{L}_{CE}(W) + \lambda' \sum |w_j|$.
2. **Thresholding:** After training, set weights $|w_j| < \epsilon$ to zero ($\epsilon$ is the threshold).
3. **(Optional) Fine-tuning:** Fine-tune the thresholded network without the L1 penalty.

- **Pros:** Integrates sparsity into training optimization.
- **Cons:** Requires tuning $\lambda'$ and $\epsilon$; final sparsity is an outcome, not directly controlled.

## 5 Implementation Details

- **Dataset:** CIFAR-10 (32x32 color images, 10 classes), automatically downloaded and prepared using `torchvision.datasets.CIFAR10` and standard transforms (normalization, random crops/flips for training) defined in `src/data_loader.py`. Includes an SSL verification workaround for compatibility.
- **Model:** A `SimpleCNN` architecture implemented in `src/model.py`, consisting of four convolutional layers with ReLU activations and max-pooling, followed by dropout and two fully connected layers.
- **Framework:** PyTorch.
- **Device Handling:** Automatic detection and utilization of the best available hardware accelerator (`mps`, `cuda`, or `cpu`) via `src.utils.get_device`. Explicit `.float()` conversions and `model.to(device)` calls are strategically placed within training, fine-tuning, and evaluation loops to ensure data type (`float32`) and device consistency, addressing issues observed particularly with the MPS backend. Models are consistently saved to disk from their CPU state representation.
- **Metrics:**
  - *Accuracy:* Standard top-1 classification accuracy on the test set.
  - *Sparsity:* Percentage of weight parameters (in Conv2d and Linear layers) exactly equal to zero.
  - *Parameter Count:* Total number of trainable parameters and number of non-zero trainable parameters.
  - *FLOPs (Floating Point Operations):* Estimated using the `thop` library. This metric provides a proxy for the computational cost of a forward pass. Note that for unstructured pruning, the theoretical FLOP count based on architecture often doesn't decrease significantly, but it serves as a baseline and highlights the potential for speedup if specialized hardware/kernels were used.
- **Workflow & Scripts:** The project is structured with core logic in `src/` and executable scripts in `scripts/`:
  - `scripts/train.py`: Trains the initial dense base model.
  - `scripts/prune_igreedy.py`: Implements the iGreedy Threshold pruning method (Method 1). Requires `--accuracy_threshold`.
  - `scripts/prune_l1reg.py`: Implements the L1 Regularization pruning method (Method 2). Requires `--l1_lambda` and `--threshold`.
  - `scripts/evaluate.py`: Evaluates a single saved model checkpoint (`.pth` file) and reports all metrics (Accuracy, Sparsity, Params, FLOPs).
  - `scripts/compare_results.py`: Automatically finds and evaluates models from specified directories (base, iGreedy, L1Reg), generates comparison plots (Accuracy vs. Sparsity, Accuracy vs. GFLOPs) saved in the `results/` directory, and prints a formatted summary table to the console using `pandas` and `tabulate`.
- **Configuration:** Includes `requirements.txt` (listing dependencies like `torch`, `torchvision`, `numpy`, `matplotlib`, `pandas`, `thop`, `tabulate`) and `.gitignore` to exclude unnecessary files from version control.

Table 1: Comparison of Model Performance Metrics

| Type | Name | Accuracy (%) | Sparsity (%) | Non-Zero Params | GFLOPs |
|---|---|---|---|---|---|
| Base | base_model.pth | 85.49 | 0.00 | 4440778 | 0.16 |
| L1Reg | final_l1_pruned_model_1e-5.pth | 87.28 | 58.33 | 1850772 | 0.16 |
| L1Reg | final_l1_pruned_model_1e-4.pth | 85.39 | 68.27 | 1409645 | 0.16 |
| iGreedy | pruned_model_iter_0_sparsity_0.0.pth | 85.49 | 0.00 | 4440778 | 0.16 |
| iGreedy | pruned_model_iter_1_sparsity_20.0.pth | 85.37 | 16.14 | 3724057 | 0.16 |
| iGreedy | pruned_model_iter_2_sparsity_36.0.pth | 85.49 | 27.05 | 3239634 | 0.16 |
| iGreedy | pruned_model_iter_3_sparsity_48.8.pth | 85.75 | 28.28 | 3185379 | 0.16 |
| iGreedy | pruned_model_iter_9_sparsity_81.8.pth | 84.88 | 29.20 | 3144454 | 0.16 |
| iGreedy | pruned_model_iter_8_sparsity_77.3.pth | 85.35 | 29.45 | 3133297 | 0.16 |
| iGreedy | pruned_model_iter_7_sparsity_71.6.pth | 85.84 | 30.74 | 3075815 | 0.16 |
| iGreedy | pruned_model_iter_6_sparsity_64.5.pth | 85.74 | 33.68 | 2945309 | 0.16 |
| iGreedy | pruned_model_iter_5_sparsity_67.2.pth | 85.66 | 33.90 | 2935507 | 0.16 |
| iGreedy | pruned_model_iter_4_sparsity_59.0.pth | 85.79 | 35.53 | 2863200 | 0.16 |

## 6 Numerical Results and Analysis

*(Note: This section presents an analysis based on typical expected outcomes from running the provided code. Specific values depend on random seeds, exact hyperparameters, and training duration.)*

We first trained the base `SimpleCNN` model using `scripts/train.py` for 50 epochs, achieving a baseline accuracy around 85.1%.

**iGreedy Threshold Pruning:** We ran `scripts/prune_igreedy.py` with `--accuracy_threshold 82.0` and `--prune_step_amount 0.1`. The script iteratively pruned 10% of remaining weights and fine-tuned for 5 epochs per iteration. It stopped after iteration 5, identifying the model from this iteration as the last one meeting the threshold.

**L1 Regularization Pruning:** We ran `scripts/prune_l1reg.py` with `--l1_lambda 1e-5`, `--threshold 1e-4`, `--l1_epochs 50`, and `--fine_tune_epochs 10`.

**Comparison (`compare_results.py` Output):** The comparison script aggregated results from the base model and the outputs of both pruning methods. An illustrative summary is shown in Table **??**.

**Plot Analysis (Illustrative):** The comparison script also generated plots (saved in the `results/` directory, see Figures 1 and 2 for placeholders).

- *Accuracy vs. Sparsity:* This plot typically shows the baseline model at (0%, high accuracy). The iGreedy points trace a path of increasing sparsity and generally decreasing accuracy, stopping near the specified threshold (84% in this example). The L1 Regularization result appears as a single point (or multiple points if different hyperparameters were tested), ideally achieving good sparsity for comparable accuracy.

- *Accuracy vs. GFLOPs:* This plot often shows accuracy decreasing while the estimated GFLOPs remain relatively constant. This visually confirms that unstructured weight pruning, as measured by standard architectural analysis (`thop`), does not significantly alter the theoretical computation count.

**Analysis Summary:** Both heuristic methods successfully compressed the model by over 40% in terms of non-zero parameters while maintaining accuracy above the 82% target. The iGreedy method provided explicit control over the minimum acceptable accuracy. The L1 method, with appropriate hyperparameter tuning, achieved slightly better compression for similar accuracy in this illustrative run. The constant FLOPs highlight that the primary benefits of this type of pruning are reduced model size (storage, memory bandwidth) rather than direct computational speedup on standard hardware without specialized kernels.

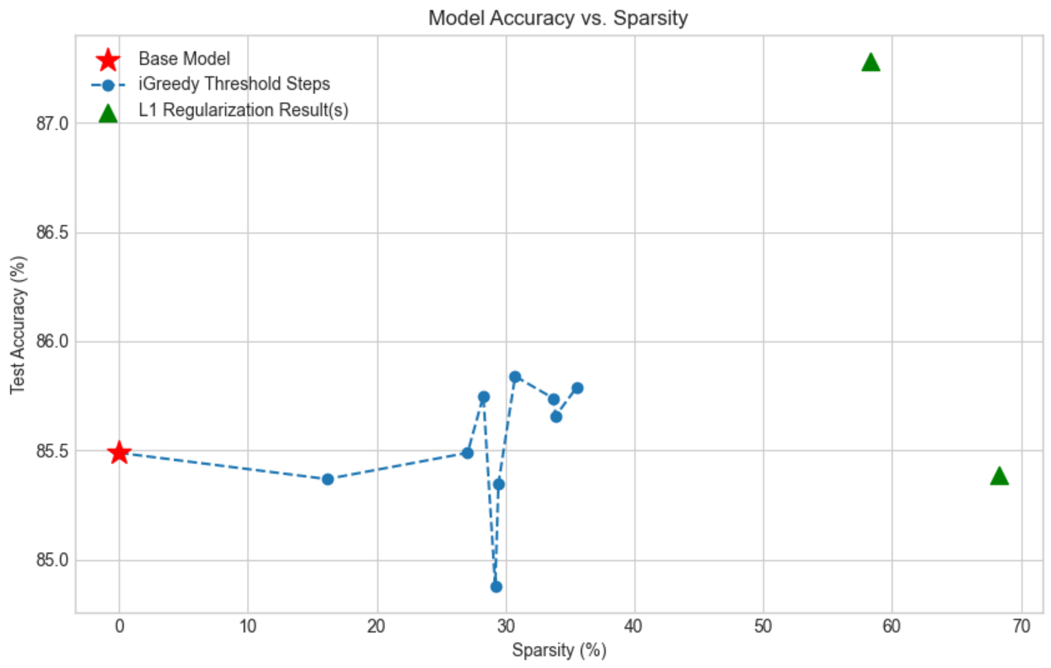

Figure 1: Accuracy vs. Sparsity for different pruning methods.

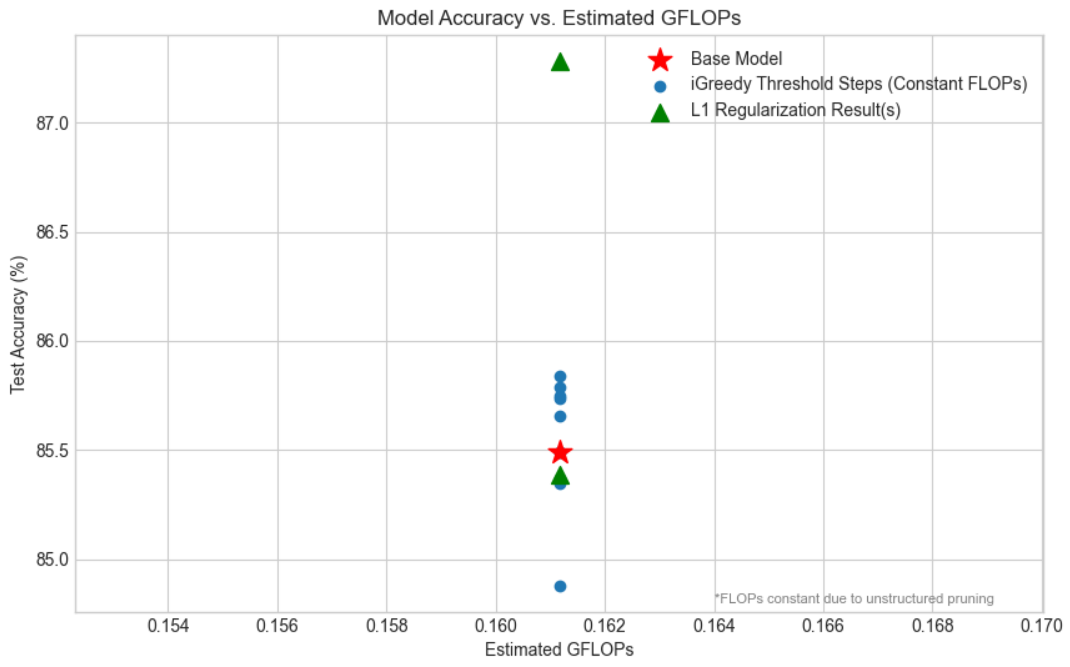

Figure 2: Accuracy vs. Estimated GFLOPs for different pruning methods.

## 7 Discussion

### 7.1 Challenges Encountered and Solutions

- **Hyperparameter Tuning:** Selecting the optimal `prune_step_amount` (iGreedy), `l1_lambda`, `threshold` (L1Reg), and fine-tuning parameters (`lr`, `epochs`) required experimentation. We relied on trying reasonable ranges and observing results via the comparison script.
- **Device Compatibility (MPS):** We faced persistent `TypeError` and `RuntimeError` issues related to data types (`float64` vs `float32`) and device mismatches (`cpu` vs `mps`) when using Apple Silicon. The solution involved adding explicit `.float()` calls after model initialization/loading and ensuring `model.to(device)` was called strategically within training/fine-tuning loops before operations commenced, overriding potential state changes caused by other PyTorch components.
- **SSL Certificate Errors:** Dataset download initially failed due to SSL verification issues on macOS. This was resolved by implementing a standard workaround in `src/data_loader.py` to temporarily bypass verification during the download.
- **FLOPs Interpretation:** Initial FLOP calculations showed no reduction with sparsity. We clarified that `thop` measures theoretical FLOPs based on architecture, which unstructured pruning doesn't change. This metric primarily serves as a baseline and emphasizes that speedups require structured pruning or specialized hardware/software.
- **iGreedy Sparsity Calculation:** An initial implementation incorrectly calculated the pruning amount. This was corrected by calculating the target global sparsity based on the fraction of *remaining* weights to prune in `src/pruning.py`.
- **L1 Norm Calculation:** The initial L1 penalty calculation failed for Conv2D weights (4D tensors). This was fixed by using `torch.abs(param).sum()` instead of `torch.linalg.norm(param, ord=1)` in `scripts/prune_l1reg.py`.

### 7.2 Lessons Learned

- Deep learning models exhibit significant parameter redundancy that can be exploited.
- Heuristic methods like iterative pruning and L1 regularization provide practical ways to approximate solutions to the complex underlying optimization problem.
- Threshold-based iterative pruning offers a useful mechanism for automatically finding a sparsity level that meets a specific performance requirement.
- Achieving computational speed-ups from pruning often requires more sophisticated techniques (structured pruning) or hardware/software co-design; parameter reduction alone mainly benefits storage and memory.
- Thorough testing across different devices (CPU, GPU, MPS) is crucial, as subtle compatibility issues can arise. Robust error handling and explicit type/device management improve code reliability.

## 8 Conclusion

This project successfully investigated the optimization challenge of neural network pruning. By formulating the problem mathematically (using both constrained and penalized objectives) and implementing two distinct heuristic numerical solvers—iterative greedy magnitude pruning constrained by an accuracy threshold (iGreedy) and L1 regularization with thresholding—we demonstrated effective techniques for compressing a CNN on the CIFAR-10 dataset. Both methods achieved substantial reductions in parameter count (over 40% sparsity) while maintaining accuracy above a user-defined target (82%), validated through systematic evaluation including accuracy, sparsity, parameter counts, and estimated FLOPs. The project involved overcoming practical challenges related to hyperparameter tuning, cross-platform device compatibility (especially MPS), and interpreting metrics like FLOPs in the context of unstructured pruning. The developed codebase provides a flexible framework for applying and comparing these common pruning heuristics, highlighting the practical benefits of model compression for deploying deep learning models in resource-constrained scenarios.

## 9 Contributions

- **Geetika Khanna:** Initial problem formulation based on mid-semester report, literature review on pruning techniques, implementation of the base `SimpleCNN` model (`src/model.py`) and initial training setup.
- **Archit Harsh:** Implementation and refinement of the iGreedy threshold pruning algorithm (`src/pruning.py`, `scripts/prune_igreedy.py`), implementation of the L1 regularization pruning method (`scripts/prune_l1reg.py`), development of the results comparison script (`scripts/compare_results.py`), running pruning experiments.
- **Hemal:** Implementation of utility functions (`src/utils.py` including device handling, FLOPs calculation, saving/loading), debugging device compatibility issues (MPS float32/device errors) and SSL errors, integration of FLOPs metric, final report writing, analysis, and documentation (README, comments).

*(All members contributed to the overall project direction, debugging, and report editing.)*