Examining Lightweight Convolutional Neural Networks and the Efficacy of Pruning Methods

Bo Yang
Georgia Institute of Technology
byang364@gatech.edu

Alestin Sphere
Georgia Institute of Technology
asphere3@gatech.edu

Abstract

This paper analyzes a variety of lightweight convolutional neural networks and attempts to establish a basic method for parameter pruning. We first assemble a shortlist of models by perusing research on lightweight neural network architectures. From the candidates, we pick out four (SpinalNet, SqueezeNet, DenseNet, and Compact Convolutional Transformer) to examine in depth the qualities that are most essential to their structural efficiency. Our first experiment measures the influence of parameter count on SpinalNet and SqueezeNet's classification performance. Our second experiment attempts to prune DenseNet and Compact Convolutional Transformer (CCT) using a combination of unstructured and structured methods. By comparing models with varying parameter counts, we hope to gain insight into the unique, or common, characteristics that high performing and lightweight architectures share.

1. Introduction

There are a wide range of techniques and architectures utilized today to reduce the resource demands of training deep learning models. Some of the primary methods are hardware focused e.g. via specialized processing units such as TPUs (Tensor Processing Units). However, their limitations include high R&D and manufacturing costs, and long development times (not limited to chip fab but building infrastructure for new/increased production).

On the software side, reducing the number of trainable model parameters remains one of the most cost-effective ways to cut down on training time and cost. This type of optimization is also far more accessible for the average researcher, as most organizations cannot afford to buy the latest GPUs en masse, let alone develop specialized hardware.

For our experiment, we evaluated a handful of candidate models on CIFAR-10 [7] to decide which ones to examine further. CIFAR-10 is an image dataset that consists of 60,000 32x32 RGB images across 10 classes, and is one of the most popular benchmarks for low-resolution image classification. Our main experiment uses Food-101 [2], which is a dataset containing 101,000 512x512 RGB images from 101 food classes. Due to training time constraints, the image size was downsampled to 64x64 during preprocessing.

2. Approach and Methodology

The initial step of the project consisted of picking from existing models a set of CNNs that fit the scope of our paper. We looked for lightweight models with open source code that performed well on CIFAR-10 [9], as it is a similar dataset to our choice of a downsampled Food101 dataset. Each model's paper results were verified via training on CIFAR-10 from scratch. A default set of hyperparameters were used for simplicity. The compact transformer architectures we picked out used different settings than regular CNNs and we made a best attempt at simulating their paper's configurations. Using a smaller dataset for this step allowed for much faster training times compared to using ImageNet.

From the candidate models listed in Appendix - Table 2, SpinalNet [6], SqueezeNet [8], DenseNet, and Compact Convolutional Transformer were selected for the main experiments. The goal was to gather a variety of neural network architectures that we had also worked with for previous assignments.

Our default training setup leveraged Georgia Tech's PACE high performance computing cluster to train models considerably faster than using Google Colab. PACE jobs are submitted using SLURM scripts that request partitions with various hardware specifications from compute nodes. Our solver and configuration parser scripts were modified from Assignment 2's framework. In order to accommodate a bigger dataset with larger image sizes for the Food101 experiments, we went a step further and implemented a distributed training setup using Hugging Face's Accelerate API[5]. It wrapped PyTorch's `DistributedDataParallel` module and allowed for

parallel training across multiple nodes. All training was done on Nvidia H100 and H200 GPUs.

Our modified solver creates a separate process for each available GPU and assigns the same batch size. All inter-process communication is handled by the API. After every epoch, only the main process updates metrics and saves the best model as training is the only task sped up by parallelization. Barrier synchronization is implemented whenever all processes must arrive at the same spot before continuing, e.g. before Accelerate's `end_training` call that destroys and cleans up all running processes. Our design logic tries to minimize synchronization and overhead for all non-training tasks to reduce complexity.

In addition to implementing distributed training, we added MLflow functionality to the solver code to help with organizing results and generating plots. We used PACE's head node to run MLflow local server and saved results to the scratch directory. During training, the solver's main process logged metrics for each run. All tunable hyperparameters and relevant metrics gathered during training were recorded. The plots for this paper were generated via csv files using pyplot and seaborn.

Lastly, we created a modified parser that could override the default configuration file settings for each model with command line arguments. The script takes the model name via environment variable exported from the SLURM script, then instantiates and runs the solver with keyword arguments passed from command line and the config file. The final training setup combined with leveraging SLURM job arrays for batch submits were used to perform hyperparameter sweeps for the second experiment.

2.1. Experiment I

In the first experiment, we tuned the models similarly to each respective paper to get a baseline. These models were then modified to accommodate the new dataset inputs (Food-101, 64x64). Despite the training acceleration we were able to achieve, we decided to fallback on tuning in CIFAR-10 and applying results to Food-101. While this was not optimal, it did allow us to rapidly test a variety of tuning parameters and settle on our final parameter configuration for our models in this first experiment.

After trial and error, we settled on using the Cosine Annealing scheduler with warm restarts; this scheduler provided better flexibility for the more complex data found in Food-101. The periodic restarts made the scheduler less sensitive to local maxima. We also, found that AdamW performed best, among the other options, for SpinalNet and Squeeze variants.

We did run a few training cycles with label smoothing, but ultimately removed this as we observed lower performance than without it. Since Food-101 inputs are more complex, and we reduced the size considerably, it is possible that label smoothing just added to the uncertainty of the model predictions. For the final training cycles, we included early stopping to reduce run times.

For this experiment, since we were exploring was based on quantitative attributes of the model architectures rather than performance we were able to make each comparison in isolation and then aggregate our findings into categories at the end.

2.2. Experiment II

For initial hyperparameter tuning, our goal was to find an optimal, best-effort setup that could replicate paper results and be used for comparing pruned and default models. Using our distributed training SLURM array setup, we varied learning rate, batch size, weight decay, and epoch count for DenseNet and CCT using values incremented both linearly and logarithmically depending on hyperparameter type. We decided to use the cosine annealing scheduler for all runs as it had less hyperparameters to tune than a step scheduler. We used the SGD optimizer as it seemed the default choice in most papers. For CCT, we resorted to the AdamW optimizer as it worked better for transformer architectures.

The results from the hyperparameter sweeps were used to determine a standard configuration for all subsequent pre and post prune training. The pruning experiments compared only the same models to themselves with different parameter counts, so it was not as important to tune for the absolute best default accuracy than to use the same hyperparameter configuration for all experiments.

While synchronizing training was relatively easy with the Accelerate API, we had more difficulty with determining a distributed pruning framework for the experiment. We needed to decide on how to prune each model, then how to implement the process in a distributed setup. We settled on a two phase approach after some research[1, 11, 10].

In the first phase, the model is trained for half of the epochs as our default configuration. The best model is then pruned using PyTorch's L1 pruning method[10] by zeroing the smallest L1 norm weights in all Conv2D layers. This type of unstructured pruning targets individual weights instead of entire layers. The weights are zeroed but the model structure stays intact.

In the second phase, we prune the model again using Torch-Pruning's[12] structured pruning method. The least essential conv layers are removed by comparing the sum of their L1 norm weights. Afterwards, the pruned model is trained again for the same number of epochs as the first phase using the same weight values post phase one. The best validation accuracy from the second phase is recorded as the best overall validation accuracy.

We initially wanted to manually prune each layer after zeroing the weights, but that proved too time consuming since it required recalculating the in/out shapes for each

subsequent layer. The custom library we found was able to use an example input tensor to determine via a forward pass how the tensor shape changed after each layer was removed. This sped up the experiment significantly and allowed us to compare different pruning settings.

Our final pruning method has two adjustable ratios, one for the initial/unstructured stage that zeroes the lowest L1 weights, and one for the structured pruning stage that removes the lowest L1 norm layers. The first ratio determines the percentage of individual weights to zero, the second ratio determines the percentage of layers to remove. The final fully connected layer is ignored as its weights are essential for classification performance, only convolution layers are pruned for both models.

Integrating the pruning process into our distributed solver involves coordinating between multiple processes to ensure the same model is pruned and loaded after for training. After the first training phase, the solver waits for all processes to arrive at a barrier before calling the pruning method. Since the method is not computationally expensive, only the main process prunes and notifies workers when finished. All processes are synced again before each worker loads the pruned model and restarts training. Results are saved to MLflow at the end of the run.

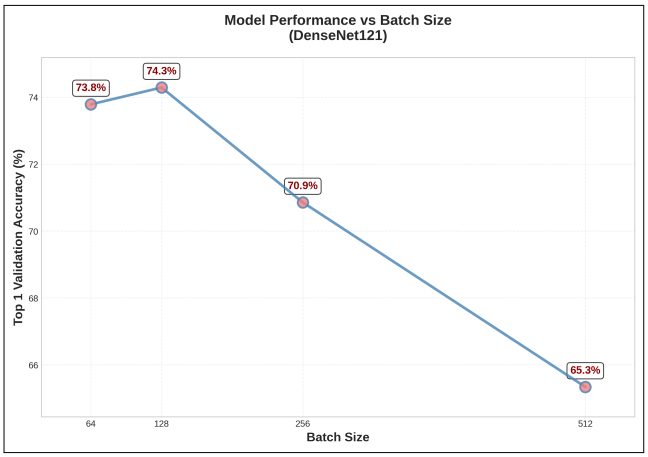
3. Results

3.1. Experiment I

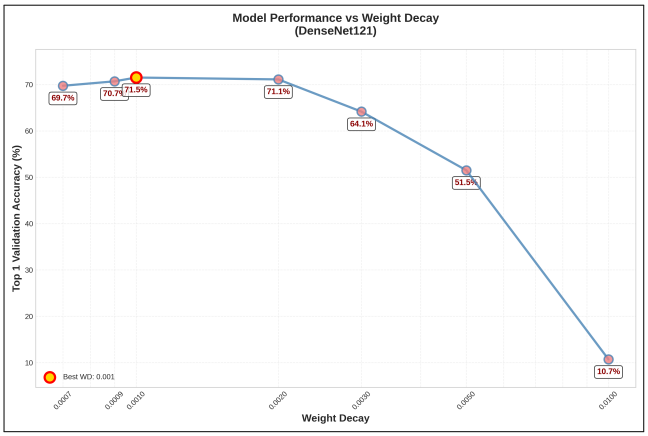
The three primary attributes we explored in this experiment were: Scalability, Compactness, and Novelty. The first two attributes we were able to quantify and our results are discussed in the Appendix, the final attribute was qualitative and we discuss all the findings below.

Scalability, for the purposes of this paper, is defined as "the rate at which trainable parameters grow with respect to the input size and model depth". Then Compactness is defined as "the proportion of fewer trainable parameters a model variant has in comparison to its standard variant"; this attribute was only analyzed for architectures which augmented and existing standard variant rather than implemented a completely unique architecture. The final attribute, Novelty, is defined as the "specific characteristic(s) of an architecture that contributes to its Scalability and/or Compactness", we further determined whether this is a pattern shared among other models or truly unique.

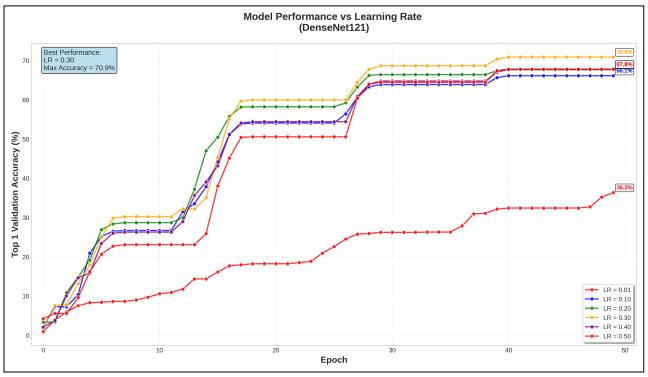
For each model, we tracked their accuracy, the number of trainable parameters, their number of layers, and where applicable, the number of trainable parameters on their standard variants. We tracked all of these metrics for each model and recorded them before and after making modifications to their input sizes, this let us get our measure of scalability. For applicable models, we estimated how many parameters the standard variant would have had to accommodate the larger input, this gave us a baseline to compare compactness.



(a) Accuracy vs. Batch Size



(b) Accuracy vs. Epochs



(c) Accuracy vs. Learning Rate

Figure 1: Hyperparameter tuning results for DenseNet121

In SpinalNet, the core architectural feature that facilitates reduced parameters is Gradual Inputs that split the fully connected layer into multiple "spinal segments". The

segmentation of the fully connected layer slowly introduces the input into subsequent layers where each gets a chunk of the input and output from the previous layer. In practice, this considerably reduces the number of parameters as shown in Appendix - Table 2.

In SqueezeNet, there are two primary contributing structures that massively reduce parameters. The first is a Fire Module, these modules make use of a Squeeze Layer. In a Squeeze Layer they use a "1x1" convolution filter and dial back the output channels resulting in fewer parameters deeper in the network. The second structure, and most impactful, which SqueezeNet utilizes is simply to remove the fully connected layer and replace it with an Average Pooling layer. In many CNNs, the fully connected layer adds a large amount of parameters, so by removing this, SqueezeNet is able to get the results mentioned in their paper of 1/50 the params than AlexNet. From Appendix - Table 2, you can see that SqueezeNet has among the lowest parameters compared with other models we experimented with.

3.2. Experiment II

3.2.1 DenseNet

DenseNet is a CNN architecture from 2016[4] that utilizes dense connections throughout the network to promote gradient flow and feature reuse. Each layer in a DenseNet receives feature maps from all previous layers within the same dense block as input. The idea, while inspired by residual connections, is slightly different as dense connections concatenate feature maps from previous layers rather than adding them. This allows models to access previous feature maps directly and potentially reuse, which leads to more parameter-efficient networks.

Our hyperparameter sweep started with settings adapted from Assignment 2 tuning. After implementing distributed training, we increased batch size along with learning rate to better utilize GPU resources. Searching adjacent values yielded clear choices for all hyperparameters. It was observed that training on more than 100 epochs brought negligible performance improvement. The default configuration for subsequent experiment was set to 128 batch size, 0.25 learning rate, 0.001 reg, and 100 epochs, using cosine scheduler with SGD optimzer. Early stopping was tested but scrapped as 100 epoch runs were already fairly short at 14 minutes per run; we also could not find a satisfactory configuration for patience. Figure 1 shows our results.

We used the smallest model from the DenseNet paper for pruning. DenseNet121 has 120 conv layers and 1 fully connected layer for 1,035,653 trainable parameters. We were able to replicate its paper results using our setup. Our pruning experiment tested nine combinations of equal pruning ratios from 0.1 to 0.9. The baseline model achieved 76.0% accuracy.

The results from Table 1 and Figure 2 show that the pruned model maintains reasonable performance (73.68%) even with 52% parameter reduction. Only at 80% reduction and beyond does accuracy degrade significantly. We did not expect the pruned model to do so well, especially when its channel and growth rate settings were designed for the smaller CIFAR-10 dataset. This suggests there is a significant excess of conv layers that could be removed without affecting feature learning for Food101. We cannot draw any conclusions about whether this kind of dataset-specific pruning has any generalization benefits. We are also uncertain about whether freezing weights and re-training after pruning is the best way to compare performance. Nonetheless, the extent to which the model could be pruned without affecting performance surprised us.

To verify results, default and pruned models were re-trained and tested on an 80/10/10 train/val/test dataset split. The results are shown in Table 2. While the test accuracy for both models is expectedly worse, their proportional difference supports our previous results.

Table 1: DenseNet121 pruning results

Top 1 Accuracy (%)	Params	Reduction (%)
76.04	1.04M	0.0
75.56	799K	22.8
74.66	639K	38.3
73.68	498K	52.0
72.25	373K	64.0
69.69	277K	73.3
64.83	161K	84.4
57.69	94K	90.9
45.66	45K	95.6
27.09	14K	98.7

Metric	Default Model	Pruned Model
Trainable Parameters	1,035,653	506,525
Test Accuracy	49.6%	43.8%

Table 2: Final test accuracy on Food101

3.2.2 Compact Convolutional Transformer

The second model examined is the Compact Convolutional Transformer[3]. The architecture comes from a more recent paper in 2021, and is a variant of the Vision Transformer (ViT). CCT is designed to achieve high performance without relying on massive pre-training datasets. The architecture employs convolutional layers to embed the input image into token sequences that preserve local information, which contrasts with standard ViT embeddings that flatten image patches and results in losing spatial context.

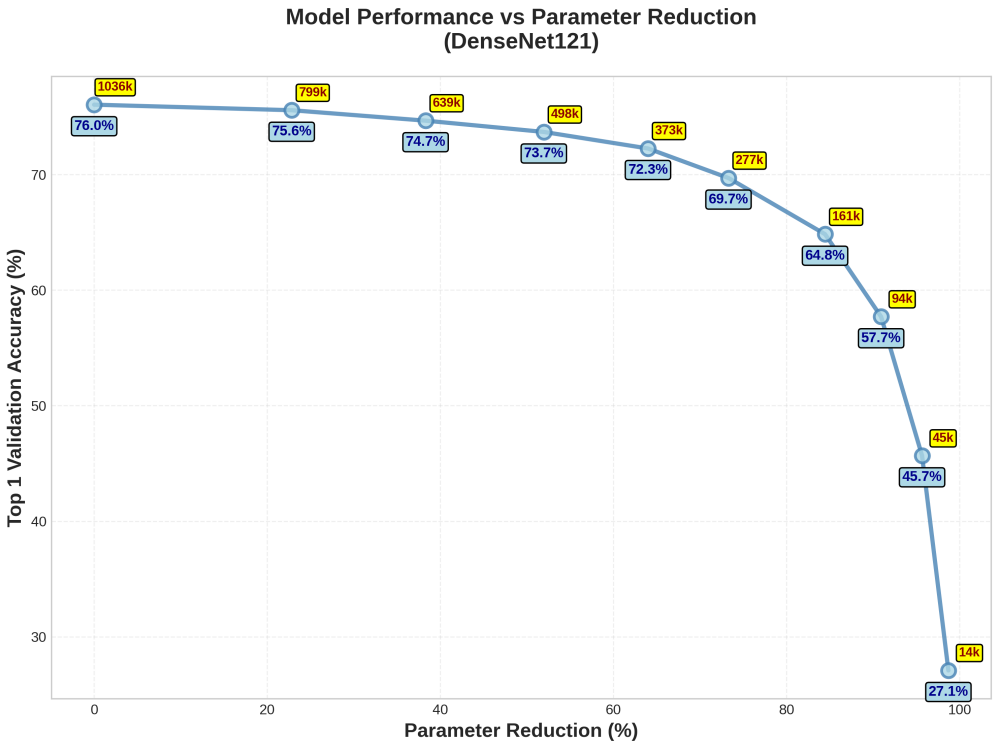


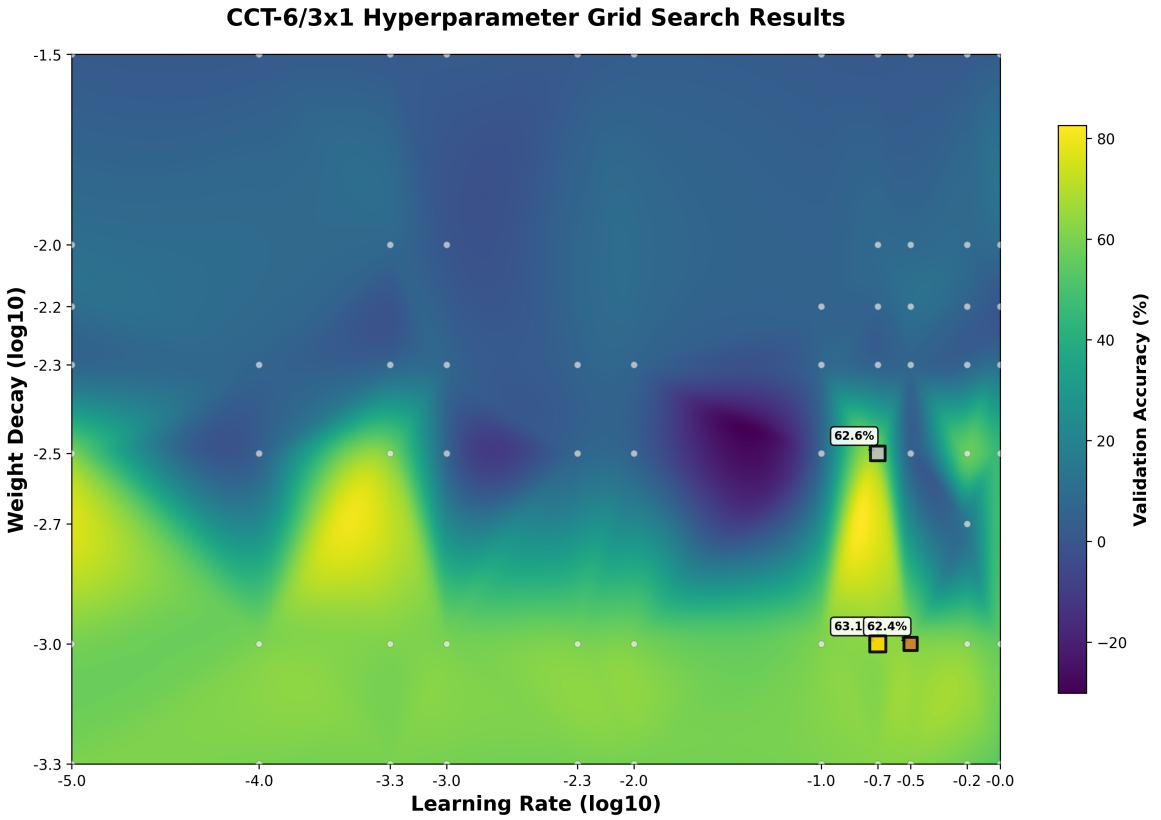
Figure 2: Model performance versus parameter reduction via structured and unstructured pruning. The final parameter count is shown in yellow boxes

The optimizer for this experiment was switched to AdamW to match the paper's training settings. Extra augmentation was done using the source Github repository as reference and included random resized cropping and color jitter. The results are shown in Figure 3a. The final hyperparameter settings were 128 batch size, 100 epochs with 10 warmups, 0.003 learning rate, and 0.2 weight decay. A more extensive sweep was performed for this model as we had little experience with tuning transformers. The best accuracy obtained from the grid search is lower than DenseNet's, mostly due to our choice to cap epochs at 100 to save time. Some initial tuning was done to minimize obvious overfitting and come up with a range for the grid search.

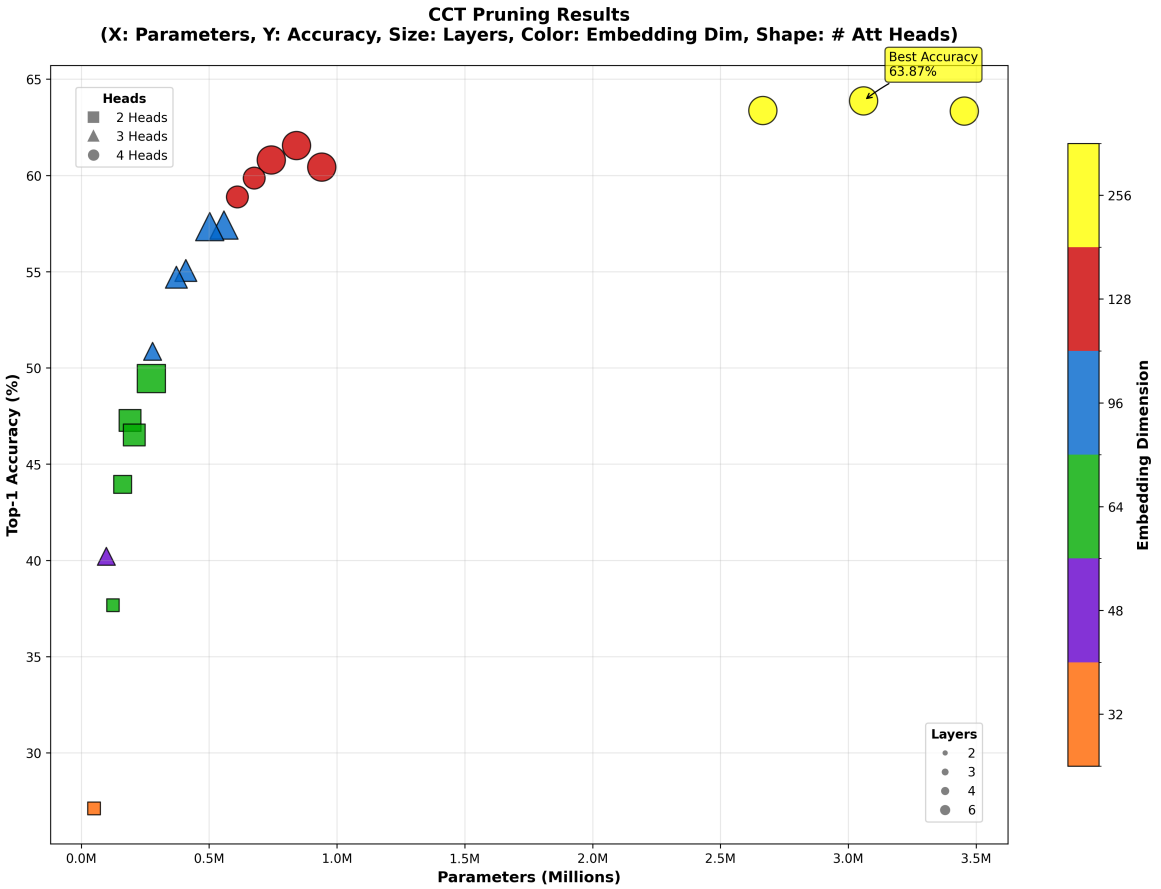
The CCT architecture was significantly more difficult to prune than DenseNet's. We could not get Torch Pruning's methods to work even when ignoring all but conv2d layers. The library could not identify the correct post-pruning tensor shapes and we ran into trouble when trying to restart training. We eventually modified the model's class constructor to allow adjustments for embedding dimension, number of attention heads, layers, and MLP ratio. The embedding dimension governs the size of the input embedding, the layer/attention heads parameters determine the depth/width of the attention blocks, MLP ratio determines the size of the fully connected layer matrices that are passed through nonlinear activations at the end of each attention block.

We used our batch training script to perform another grid search on those parameters. The result from twenty combinations are shown in Figure 3b. We find, once again, that significant parameter reduction does not lead to worse performance. From the default parameter count of 3.5 million, the model can be compressed to under a million parameters before dipping below 60% accuracy.

Like before, we postulate that the results are highly specific to dataset. This process essentially downsamples the model architecture to fit dataset complexity. As most models are used to generalize to unseen data, we do not know what the benefits are, if any, of pruning on a single training dataset. The compressed model can presumably only be applied to similar data with comparable image dimensions. Nonetheless, identifying which weights and layers remain after pruning can provide insight into which features are most essential. The learned features could be extracted and used for transfer learning applications. In cases where the target data is well-characterized, pruning, or compression, can help find optimal model sizes for specific tasks, and provide potential gains in cost and efficiency.



(a) Hyperparameter sweep results.



(b) Pruning results.

Figure 3: Combined analysis plots for CCT.

References

[1] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems*, MLSys '20, 2020. 2

[2] L. Bossard, M. Guillaumin, and L. Van Gool. Food-101 - mining discriminative components with random forests. In *European Conference on Computer Vision (ECCV)*, 2014. 1

[3] Ali Hassani, Steven Walton, Nikhil Shah, Abulikemu Abuduweili, Jiachen Li, and Humphrey Shi. Escaping the big data paradigm with compact transformers, 2021. 4

[4] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 4

[5] Hugging Face. Accelerate documentation. https://huggingface.co/docs/accelerate/en/index, 2025. Accessed: 2025-07-24. 1

[6] H. M. Dipu Kabir, M. Abdar, S. Mohammad Jafar Jalali, A. Khosravi, A. F. Atiya, S. Nahavandi, and D. Srinivasan. Spinalnet: Deep neural network with gradual input. GitHub, 2020. 1

[7] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. 1

[8] F. N. Landola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016. 1

[9] Papers with Code. Image classification on cifar-10. https://paperswithcode.com/sota/image-classification-on-cifar-10, 2025. Accessed 2025. 1

[10] PyTorch Team. Pruning tutorial. https://pytorch.org/tutorials/intermediate/pruning_tutorial.html, 2024. Accessed: 2025-07-25. 2

[11] Anh Tuan. Introduction to pruning. https://medium.com/@anhtuan_40207/introduction-to-pruning-4d60ea4e81e9, May 2022. Accessed: 2025-07-25. 2

[12] F. Wang. Torch-pruning: Towards any structural pruning. https://github.com/VainF/Torch-Pruning, 2023. 2

4. Appendix

4.1. Experiment I Issues

When implementing the modifications for models in Experiment I in order to run on Food-101. We were able to get the models running however we ran into several issues. First, the training runs took too long, we were able to reduce this with the 64x64 input size, utilizing early stopping along with use of accelerator, and most importantly testing out a max pooling layer before the fully connected layer. This led to the second issue. While the max pooling layer allowed training to run within the PACE resource limit of 16 GPU hours per job, we found performance was very low. The other approach tested out was resizing some of the earlier layers to accommodate 64x64 and some combination of this and max pooling.

4.2. Additional Data

Our repo: https://github.gatech.edu/asphere3/cs7643-paella

Model	Paper Accuracy	Experimental Accuracy	Number of Parameters	Implementation Link
CCT-7/3x1		88.59%	4,679,435	Paper / Experimental
CVT 2/4		74.37%	209,163	Paper / Experimental
PReactResNetB		93.17%	451,626	Paper / Experimental
EfficientNet		90.74%	3,599,686	Paper / Experimental
GoogLeNet		94.98%	6,166,250	Paper / Experimental
MobileNetV2		93.63%	2,296,922	Paper / Experimental
RegNetX$_2$00MF		94.66%	2,321,946	Paper / Experimental
DenseNet121		94.73%	1,000,618	Paper / Experimental
PyramidNet	95.42%	95.08%	1,772,706	Paper / Experimental
ResNet56	93.03%	93.74%	853,018	Paper / Experimental
GhostNetV3 1.0	93.30%	85.94%	6,156,908	Paper / Experimental
ShuffleNetV2		92.35%	1,263,854	Paper / Experimental
ThriftyNet	90.95%			Paper / Experimental
Spinal CNN		91.56%	2,230,282	Paper / Experimental
SpinalResNet18	91.42%	90.38%	3,609,130	Paper / Experimental
SpinalResNet34		91.57%	6,138,026	Paper / Experimental
SpinalVGG11		90.74%	10,559,242	Paper / Experimental
SpinalVGG13		92.54%	10,744,138	Paper / Experimental
SpinalVGG16		92.35%	16,056,394	Paper / Experimental
SpinalVGG19	91.40%	92.43%	21,368,650	Paper / Experimental
SqueezeNet	80.3%		$\sim$ 1,200,000	Paper / Experimental

Table 3: All considered models and implementations

Table 4: Initial Hyperparameter Tuning Results for DenseNet-121. All runs used the SGD optimizer with a cosine learning rate scheduler.

Top 1 Acc. (%)	**Batch Size**	**Epochs**	**LR**	**Weight Decay**	**Sec/Epoch**
76.75%	128	100	0.25	0.0010	8.39
76.72%	128	200	0.25	0.0010	8.24
74.29%	128	50	0.25	0.0010	8.37
73.79%	64	50	0.30	0.0010	9.75
73.71%	128	50	0.35	0.0010	8.10
73.47%	128	50	0.20	0.0010	8.33
71.51%	256	50	0.30	0.0010	8.52
71.34%	256	300	0.30	0.0005	8.26
71.11%	256	50	0.30	0.0020	8.45
70.90%	256	250	0.30	0.0005	8.45
70.86%	256	50	0.30	0.0010	8.97
70.73%	256	150	0.30	0.0005	8.43
70.69%	256	50	0.30	0.0009	8.26
70.67%	256	200	0.30	0.0005	8.50
69.73%	256	50	0.30	0.0007	8.41
69.11%	256	100	0.30	0.0005	8.47
69.09%	256	50	0.30	0.0005	8.46
68.68%	256	50	0.30	0.0005	9.01
67.77%	256	50	0.50	0.0005	9.05
67.73%	256	50	0.40	0.0005	8.48
67.63%	256	50	0.20	0.0005	9.00
66.11%	256	50	0.10	0.0005	8.29
65.34%	512	50	0.30	0.0010	10.19
64.15%	256	50	0.30	0.0030	8.39
51.49%	256	50	0.30	0.0050	8.28
36.34%	256	50	0.01	0.0005	8.46
10.70%	256	50	0.30	0.0100	8.49

Table 5: Partial Hyperparameter Tuning Results for CCT-6/3x1. All runs used the AdamW optimizer with a cosine learning rate scheduler for 100 epochs.

Top 1 Acc. (%)	Batch Size	LR	Weight Decay
63.11%	64	0.0010	0.2000
62.55%	128	0.0030	0.2000
62.40%	128	0.0010	0.3000
62.18%	64	0.0005	0.2000
62.16%	64	0.0010	0.3000
62.08%	64	0.0005	0.3000
61.56%	256	0.0010	0.3000
60.93%	128	0.0010	0.2000
60.88%	64	0.0010	0.0005
60.61%	128	0.0010	0.1000
60.53%	64	0.0005	0.1000
60.46%	64	0.0005	1e-05
60.43%	128	0.0010	0.6000
60.40%	128	0.0010	0.0005
60.24%	64	0.0010	0.1000
60.09%	128	0.0010	0.0001
60.09%	64	0.0005	0.0005
60.03%	128	0.0005	0.0001
59.84%	256	0.0010	0.1000
59.81%	128	0.0005	0.2000
59.80%	128	0.0010	0.0050
59.79%	256	0.0010	0.6000
59.79%	64	0.0010	0.0010
59.79%	128	0.0005	1e-05
59.70%	64	0.0010	1e-05
59.69%	64	0.0010	0.0100
59.63%	128	0.0005	0.3000
59.62%	64	0.0030	1e-05
59.59%	128	0.0030	0.0005
59.58%	128	0.0005	0.0010
59.49%	64	0.0010	0.0050
59.46%	128	0.0005	0.0100
59.39%	128	0.0010	0.0010
59.32%	64	0.0010	0.0001
59.27%	64	0.0005	0.0050
59.23%	128	0.0005	0.1000
59.18%	64	0.0005	0.0010
59.17%	128	0.0005	0.0005
59.06%	256	0.0010	0.0010
59.05%	128	0.0010	1e-05
59.04%	128	0.0005	0.0050
59.03%	256	0.0010	0.0050
59.02%	128	0.0010	0.0100
58.99%	256	0.0010	1e-05
58.97%	64	0.0005	0.0001
58.89%	256	0.0010	0.0100
58.89%	256	0.0005	0.0001
58.88%	256	0.0005	0.0100
58.86%	256	0.0005	0.0050
58.82%	256	0.0005	1e-05
58.73%	256	0.0010	0.0005
58.66%	256	0.0005	0.2000

Table 6: CCT Pruning Results. All models were trained for 100 epochs with a batch size of 128, learning rate of 0.003, and weight decay of 0.2.

Top 1 Acc. (%)	Parameters	MLP Ratio	Heads	Layers	Emb. Dim.
63.87%	3 059 814	1.5	4	6	256
63.37%	2 665 830	1.0	4	6	256
63.34%	3 453 798	2.0	4	6	256
61.55%	841 830	1.5	4	6	128
60.80%	743 142	1.0	4	6	128
60.43%	940 518	2.0	4	6	128
59.86%	676 326	2.0	4	4	128
58.88%	610 534	1.5	4	4	128
57.42%	557 958	2.0	3	6	96
57.34%	502 374	1.5	3	6	96
55.06%	408 966	2.0	3	4	96
54.71%	371 910	1.5	3	4	96
50.86%	278 886	1.0	3	3	96
49.44%	273 702	2.0	2	6	64
47.27%	190 630	1.5	2	4	64
46.51%	207 142	2.0	2	4	64
43.95%	161 478	1.5	2	3	64
40.21%	98 022	1.0	3	3	48
37.67%	124 070	1.0	2	2	64
27.11%	49 798	1.0	2	2	32

Student Name	Contributed Aspects	Details
Alestin Sphere	Project Setup	Created the initial GitHub repo, brought together training files from Bo, figured out the dev env setup on PACE, documented the steps, setup the jupyter notebook (commit mainline branch)
	Initial Food101 Processing	I first tried pulling Food101 from another site but the dataset was stored in '.h5' format and they had downsampled it to 32x32. When I tried testing on my Default model, the dataset was too low res for us to train on, so we went with a different dataset. (this is in the debug branch) **NOTE: after this point, our branches mostly diverged and we implemented our own improvements (though we shared advice and some snippets)**
	CIFAR-10 Results	Results from the models I tested can be found here
	Misc. Improvements	1) Added image preview for convenient viewing of how transforms affected the dataset 2) Setup Norm calculator to add to dataset transformer 3) Performance Improvements 4) Early Stopping, Label Smoothing, AdamW
	Implemented a Few Models for Experimenting	1) Default model (for testing the PACE job and trainer code) (file) 2) CNN Spinal (file) 3) SpinalResNet18 & SpinalResNet34 (file) 4) Spinal: VGG11, VGG13, VGG16, VGG19 (file) 5) SqueezeNet (file)
	Report	Wrote 50% of the report, contributed to each section (except for Experiment II)
	Experiment I	My primary contribution to the whole project was all aspects of Experiment I. Some of the results from tuning and training runs can be found here
	Results	1) Experiment I was my contribution to this project 2) I was not as successful at reimplementing my models for Food101, I could not get good performance and mention this in the Appendix. But I was able to still discuss the characteristics which contributed to my analysis on why these models are able to perform with less parameters and show comparisons of their params
Bo Yang	Rest of the project	Did everything else

Table 7: Contributions of team members.