Projection-Free CNN Pruning via Frank-Wolfe with Momentum: Sparser Models with Less Pretraining

Hamza ElMokhtar Shili[1] **Natasha Patnaik**[2] **Isabelle Ruble**[2]
Kathryn Jarjoura[2] **Daniel Suarez Aguirre**[2]

[1]Rice University, Computer Science Department

[2]Rice University, Computational Applied Math & Operations Research Department

Abstract

We investigate algorithmic variants of the Frank–Wolfe (FW) optimization method for pruning convolutional neural networks. This is motivated by the "Lottery Ticket Hypothesis", which suggests the existence of smaller sub-networks within larger pre-trained networks that perform comparatively well (if not better). Whilst most literature in this area focuses on Deep Neural Networks more generally, we specifically consider Convolutional Neural Networks for image classification tasks. Building on the hypothesis, we compare simple magnitude-based pruning, a Frank–Wolfe style pruning scheme, and an FW method with momentum on a CNN trained on MNIST. Our experiments track test accuracy, loss, sparsity, and inference time as we vary the dense pre-training budget from 1 to 10 epochs. We find that FW with momentum yields pruned networks that are both sparser and more accurate than the original dense model and the simple pruning baselines, while incurring minimal inference-time overhead in our implementation. Moreover, FW with momentum reaches these accuracies after only a few epochs of pre-training, indicating that full pre-training of the dense model is not required in this setting.

1 Problem Statement and Notation

Convolutional neural networks are ideal models for image classification and related computer vision tasks. However, large models with many parameters pose serious challenges - both in terms of training time and memory. In this paper, we motivate and outline a series of computational experiments aimed at evaluating different variants of the Frank-Wolfe (FW) algorithm for neural network pruning, intending to find well-performing sparser models.

Notation - From Deep Neural Networks to Convolutional Neural Networks. To represent an arbitrary neural network with N neurons, we follow the notation used in [13] and [14]. Let $\mathbf{x} \in \mathbb{R}^d$ denote the input vector and $\theta_\mathbf{i}$ for $i \in [N] = \{1, ..., N\}$ denote the parameters/ weights associated with the i-th neuron. With these inputs, the i-th neuron can be represented as a function $\sigma(\mathbf{x}, \theta_\mathbf{i})$.

Although this function $\sigma(\cdot, \theta_i)$ can take on general forms, [13] and [14] consider the specific case of a two-layer network for simpler analysis (a representative example of more complex architectures). In the two-layer case, we can write $\sigma(\mathbf{x}, \theta_\mathbf{i}) = b_i \cdot \sigma_+(\mathbf{a_i}^T\mathbf{x})$, where $\theta_\mathbf{i} = [b_i \;\; \mathbf{a_i}]$ denotes the concatenation of the second layer weight, b_i, and the vector of first layer weights, $\mathbf{a_i} \in \mathbf{R}^d$, respectively. The function $\sigma_+(\cdot)$ represents the activation function.

Accordingly, we use the following notation for the output of a neural network containing N neurons:

$$f_{[N]}(\mathbf{x}, \Theta) = \frac{1}{N} \sum_{i=1}^{N} \sigma(\mathbf{x}; \theta_{\mathbf{i}}) \tag{1}$$

Given a dataset $D = \{\mathbf{x}^{(j)}, y^{(j)}\}_{i=1}^{m}$ consisting of m observations, each with attributes $\mathbf{x}^{(j)} \in \mathbb{R}^d$ and desired output $y^{(j)} \in \mathbb{R}$, we can train the neural network to discover the set of weights Θ that minimizes the following loss function:

$$\mathcal{L}\big[f_{[N]}(\cdot, \Theta)\big] = \frac{1}{2} \sum_{j=1}^{m} \big(f_{[N]}(\mathbf{x}^{(j)}, \Theta) - y^{(j)}\big)^2 \tag{2}$$

While existing analysis in this area uses the representative example of a two-layer network, Convolutional Neural Networks have a more complex architecture, which includes convolutional layers and pooling layers. To extract local features, convolutional layers apply filters that learn to detect specific patterns. The output of a convolutional layer l, denoted $\mathbf{h}^{(l)}$, is obtained by convolving the input feature maps $\mathbf{h}^{(l-1)}$ with learnable filters $\mathbf{W}^{(l)}$ and applying an activation function, ψ, that introduces non-linearity (such as ReLU):

$$\mathbf{h}^{(l)} = \psi\big(\mathbf{W}^{(l)} * \mathbf{h}^{(l-1)}\big) \tag{3}$$

Such convolutional layers are typically succeeded by pooling layers, which reduce spatial dimensions of the feature maps to create a more compact representation while retaining essential information. For instance, max pooling will simply select the maximum value within a given local region:

$$\mathbf{h}_{\text{pool}}^{(l)} = \text{Max}\big\{\mathbf{h}^{(l)}\big\} \tag{4}$$

Three Stage Approach for Neural Network Sparsification The classic Stochastic Gradient Descent (SGD) algorithm is commonly used for Neural Network training, with the model's weights $\Theta = \{\theta_1, ..., \theta_N\}$ typically being unconstrained [11]. As outlined in [8], [13], and [14], the computational cost and memory requirements for training large-scale, dense models as N increases necessitates a more practical three-step procedure for finding smaller sub-networks: *(i) Pretraining* - minimize (1) by performing SGD over Θ, *(ii) Pruning* - finding a sub-network from the original model which performs well, and *(iii) Retraining* - performing SGD again to maintain desired accuracy, either using the newer model's weights as the initial parameters (as in [14]) or starting with a new random parametrization (as in [8]).

In particular, the pruning stage is concerned with finding a subset of neurons $S \subset [N]$ which minimizes the loss of the newly defined sub-network: $f_S(\mathbf{x}, \Theta) = {}^{1}/_{|S|} \sum_{i \in S} \sigma(\mathbf{x}; \theta_{\mathbf{i}})$. In particular, we would like this sub-network's accuracy to be comparable to the original dense model:

$$\mathcal{L}\big[f_{[S]}\big] \geq \mathcal{L}\big[f_{[N]}\big] - \delta \text{ for some } \delta \in \mathbb{R}^+ \tag{5}$$

This allows us to find sub-networks that perform relatively well (as compared to the original dense model) [13]. Furthermore, in [14], they find that pruning dense models guarantees finding more accurate sparse networks than directly training smaller models can, which is in line with the Lottery Ticket Hypothesis [5].

2 Background

Lottery Ticket Hypothesis The Lottery Ticket Hypothesis (LTH), proposed by Frankle and Carbin in 2019, suggests that within randomly-initialized, dense neural networks, there exist sparse subnetworks, or "winning tickets," capable of achieving test accuracy comparable to the original network in a similar number of iterations. These winning tickets benefit from fortuitous initializations that facilitate effective training. Justified by the LTH, a structured three-stage approach to neural network sparsification emerges.

Firstly, in the identification stage, pruning techniques are employed to systematically remove connections from the network, thereby identifying these winning tickets. Pruning enables

significant reductions in model size by removing unnecessary parameters while preserving the network's essential functionality. Frankle and Carbin's reserach consistently found that winning tickets are less than $10 - 20\%$ of the size of several fully-connected and convolutional architectures for MNIST and CIFAR10 datasets [5].

Secondly, in the optimization stage, winning tickets are leveraged to design more efficient training schemes. By focusing on training these sparse subnetworks from the start, the aim is to accelerate convergence and improve overall training dynamics. This stage involves devising novel training algorithms and optimization strategies tailored to the unique properties of winning tickets, enabling faster learning and improved generalization.

Lastly, in the refinement stage, iterative pruning techniques are employed to further enhance the performance of the identified winning tickets. Iterative pruning outperforms one-shot pruning in identifying winning tickets by repeatedly training, pruning, and refining the network over multiple rounds. This iterative process results in networks that learn faster and achieve higher test accuracy, ultimately leading to more efficient and effective deep learning models [5].

These implications suggest that certain sparse architectures with fortuitous initializations possess inherently better training properties and offer opportunities for efficient compression, which involves reducing the size of neural networks while maintaining performance. This structured approach not only advances the field of neural network pruning and optimization but also lays the foundation for more efficient and effective deep learning models.

Suitable Feasible Regions Frank-Wolfe methods can offer computational efficiency by minimizing a linear objective over the feasible set without needing costly projections [12]. This advantage extends to scenarios with polyhedral and nuclear norm ball constraints, simplifying the linear subproblems and streamlining solutions. Additionally, their projection-free nature makes them popular, particularly in machine learning, where they surpass methods requiring projection at each iteration. As such, Frank-Wolfe methods are appropriate when the optimization problem's constraints define convex, differentiable, and compact feasible regions[6]. Convexity ensures that the feasible region forms a convex set, enabling efficient exploration of the solution space. Differentiability allows for the use of gradient information to guide the optimization process towards the optimal solution. Compactness ensures that the feasible region is bounded, facilitating convergence towards a global optimum.

These essential characteristics of feasible regions crucial for the effective implementation of Frank-Wolfe algorithms, particularly within the realm of large-scale optimization, has been studied as the effectiveness of the algorithm was shown to be intricately tied to the convexity and compactness of the feasible region, coupled with the ability to solve linear optimization subproblems efficiently within this region [3]. Specifically, Frank-Wolfe methods excel in scenarios where constraints define bounded convex regions, enabling swift exploration of the solution space without necessitating costly projection operations. This alignment fits the needs of neural network pruning tasks, where a loss function subject to constraints on network sparsity is minimized. The convexity of the feasible region also ensures a well-defined optimization problem, an essential factor in steering the iterative pruning process efficiently towards an optimal solution. Additionally, the compactness property guarantees convergence towards a global optimum, ensuring the algorithm's resilience in traversing the solution landscape. Furthermore, the differentiability of the feasible region enables the computation of gradients, essential for guiding the pruning process towards optimal sparse architectures efficiently. Compactness ensures a bounded search space, preventing the optimization process from straying towards infeasible solutions.

By satisfying these properties, the optimization problem in neural network pruning mirrors the characteristics of problems well-suited for Frank-Wolfe methods. The iterative nature of Frank-Wolfe-style approaches, where solutions are iteratively refined based on linear approximations of the objective function within the feasible region, aligns with the iterative pruning process. Moreover, the reliance on gradients to navigate the search for optimal sparse architectures underscores the significance of differentiability, a core property of Frank-Wolfe methods.

3 Convolutional Neural Network Pre-training and Pruning

Convolutional Neural Networks Convolutional Neural Networks (CNNs) are a class of neural networks primarily utilized for image classification. The defining feature of CNNs is the convolution operation, which involves sliding filters over the input data to extract and learn important features such as edges and textures. Key elements of CNNs include:

- **Convolutional Layers**: Detect features by applying various filters to the input.
- **Activation Functions**: Introduce non-linearity into the model, enabling it to learn complex patterns. Eg. ReLU
- **Pooling Layers**: Reduce the dimensionality of each feature map while retaining the most important information.
- **Fully Connected Layers**: After the convolutional and pooling layers extract and reduce features, these layers classify the input based on the detected features by outputting probabilities over the classes.

Resource Constrained Neural Networks and Pruning Paradigm While machine learning and deep learning applications are increasing at an exorbitant rate, their real-world implementation is delayed by their high computational requirements in the form of cutting-edge hardware, large memory, and high energy use. [2], [9] With this context, resource-constrained solutions arise. Henceforth, we will show how pruning can help to improve efficiency in training and inference.

Neural network pruning consists of simplifying a neural network by setting some weights to zero according to some rule. For example, a basic pruning rule may consist of removing weights smaller than a certain threshold. Overall, the goal of running is to reduce model size and complexity without compromising performance. In general, the pruning process is the following:

- **Pre-Training**: Train NN to get a baseline model.
- **Pruning**: Introduce non-linearity into the model, enabling it to learn complex patterns.
- **Re-Training**: Reduce the dimensionality of each feature map while retaining the most important information.

Note that pruning and re-training may be done in an iterable manner; this is known as the tunning stage. Also, under some methods, re-training may not be necessary given the needs of the user

Within the perspective of resource-constrained neural networks, a pruned network is smaller than the baseline model and this achieves better efficiency for inference tasks. From the angle of training, pruning may also be beneficial because rather than training a large network to a full extent, the large network is trained only up to a point from which a sufficiently good subnetwork is identified. Therefore, even when considering pre-training and re-training, pruning may lead to using fewer resources as, after pruning, the rest of training is done in a smaller network.

Finally, note that this framework also applies in the case of CNNs, which are the base of our computational study.

Frank-Wolfe for Pruning To find well-performing smaller sub-networks within dense models that are pre-trained via stochastic GD, Frank-Wolfe-style approaches can be used [13] [14], where the desired sparsity level is treated as a constraint. Thus, even though Neural Network training itself is a non-convex optimization problem, the pruning stage can be reformulated as a constrained convex optimization problem, with Frank-Wolfe-style approaches being used [13].

Frank-Wolfe in Machine Learning Context The classic FW algorithm (and related projection-free methods) provides an alternative to projected Gradient Descent for convex-constrained optimization problems. Its low per-iteration complexity and suitability for complicated constraints makes it very effective in the context of large-scale machine-learning problems [10].

FW replaces the projection step of projected GD with a linear minimization sub-problem that can be described as follows:

We form the next iterate by taking a convex combination between (i) the previous iterate and (ii) an extreme point of the feasible region that best approximates the gradient direction. [10] This ensures we retain feasibility within each iteration, without using a projection step. Thus, FW is particularly advantageous when the traditional projection step in projected GD is computationally expensive [6]. See the pseudocode description for the classic deterministic variant below:

Algorithm 1 Classic (Deterministic) Frank-Wolfe Algorithm [4], [6]

Input: $x_0 \in \mathcal{C} :=$ initial starting point; $T :=$ number of iterations
for $t = 0, 1, ..., T$ **do**
  $v_t \leftarrow \text{argmin}_{x_t \in \mathcal{C}} \langle x_t, \nabla f(x_t) \rangle$ ▷ Approximate gradient direction, maintaining feasibility
  $\gamma \leftarrow \frac{2}{t+2}$
  $x_{t+1} \leftarrow (1-\gamma)x_t + \gamma v_t$ ▷ Convex combination: previous iterate and v_t
end for

In Algorithm (1), we can also substitute the linear minimization $v_t = \text{argmin}_{v_t \in \mathcal{C}} \langle x_t, \nabla f(x_t) \rangle$ with an approximate approach, as opposed to solving for the minimizer exactly [6]. Another simple modification is to perform a line search within each iteration t for the optimal γ required to find the best next iterate on the line segment connecting x_t and v_t [6].

4 Pruning Algorithms

Simple Pruning The first algorithm takes a naive approach to pruning. First, all of the weights of the model are retrieved and sorted based on magnitude. A certain percentage of the weights with the lowest magnitude are set to zero based on a user-defined *pruning percentage*.

Algorithm 2 Simple Pruning

Input: $M \in \mathcal{M}$, a neural network model; $p \in [0, 1]$, pruning percentage; D, training dataset; E, epochs for retraining
$W \leftarrow$ weights of M ▷ Retrieve model weights
$n \leftarrow |W|$ ▷ Total number of weights
$n_p \leftarrow \text{round}(p \cdot n)$ ▷ Compute number of weights to prune
$W_f \leftarrow$ flattened W ▷ Concatenate all weights into a vector
idxs $\leftarrow$ indices sorted by $|W_f|$
$I_p \leftarrow$ first n_p elements of idxs
for $i \in I_p$ **do**
  $W_f[i] \leftarrow 0$ ▷ Set pruned weights to zero
end for
Update W of M with W_f
Set optimizer, loss, and metrics for M
Retrain M on D for E epochs
return M

Frank-Wolfe Pruning The second algorithm considers a randomly chosen subsample of the weights and takes gradient information into account when selecting weights to prune. Rather than pruning a fixed percentage of the weights, the pruning mask is applied based on a user-defined *target sparsity level*. This method takes into account cases in which smaller weights are actually more integral to the structure of the CNN and thus is more flexible than the simple pruning algorithm.

Algorithm 3 Frank-Wolfe Pruning

Input: $M \in \mathcal{M}$, neural network model; D, training dataset; s, target sparsity level; N, number of pruning iterations; S, subsample size; E, epochs for retraining

Initialize optimizer (SGD)

for $i = 1$ to N **do**
  $G \leftarrow$ zeros (shape of M's trainable weights)
  $subsample \leftarrow$ sample S instances from D
  for each (x, y) in $subsample$ **do**
    $G \leftarrow G + \nabla \text{loss}(M, x, y, \text{optimizer})$
  end for
  if $i \mod \left(\frac{N}{2}\right) = 0$ **then**
    Apply pruning to M based on G and sparsity s
  end if
  Retrain M using D for E epochs
end for
return M

Frank-Wolfe Pruning + Momentum The third algorithm has two main modifications compared to the basic frank-wolfe method: dynamic sparsity and momentum. The *target sparsity level* is determined by the user. However, the sparsity of the model is increased iteratively which potentially minimizes large decreases in model performance at each iteration. Additionally, a user-defined *momentum* vector is applied to the gradients in order to further differentiate the weights and stabilize the algorithm.

Algorithm 4 Frank-Wolfe Pruning + Momentum

Input: $M \in \mathcal{M}$, neural network model; D, training dataset; s_{init}, initial sparsity; s_{final}, final sparsity; m, momentum parameter; N, number of iterations; E, fine-tuning epochs; S, subsample size; Optimizer

$\Delta s \leftarrow \frac{s_{\text{final}} - s_{\text{init}}}{N}$
$s \leftarrow s_{\text{init}}$
for $i = 1$ to N **do**
  $G \leftarrow$ zeros (shape of M's trainable weights)
  $subsample \leftarrow$ sample S instances from D
  for each (x, y) in $subsample$ **do**
    $G \leftarrow G + \nabla \text{loss}(M, x, y, \text{Optimizer})$
  end for
  $G \leftarrow m \cdot G$ ▷ Apply momentum to gradients
  Apply pruning to M based on G and sparsity s
  if $E > 0$ **then**
    Retrain M using D for E epochs
  end if
  $s \leftarrow \min(s + \Delta s, s_{\text{final}})$
end for
return M

5 Numerical Experiments

Datasets For our computational study, we consider image classification problems. Here we used the MNIST [7] dataset as our baseline. This dataset contains handwritten digits, with 60,000 examples in the training set and 10,000 examples on the test set. The digits are centered in 28x28 grayscale images. The only pre-processing applied to the data is normalization of the grayscale by dividing the value of each pixel by 255, ensuring that the pixels used as input for the model are between 0 and 1.

Network Architecture For our numerical analysis, we employ a CNN designed for the classification of 28x28 images into 10 categories, as is the case for the MNIST dataset. The architecture is detailed in Table 1.

Layer Type	Configuration	Purpose
Convolutional	32 filters of size 3x3, ReLU activation	Captures basic features from input images, introduces non-linearity.
Max Pooling	Pool size of 2x2	Reduces spatial dimensions, making the detection of features somewhat invariant to scale and orientation changes.
Convolutional	64 filters of size 3x3, ReLU activation	Increases the complexity of the model to capture more detailed features.
Max Pooling	Pool size of 2x2	Further reduces spatial dimensions, focusing on the most important features.
Flatten	—	Transforms 3D feature maps into 1D feature vectors.
Dropout	Dropout rate of 0.5	Prevents overfitting by randomly setting input units to 0 during training.
Dense	10 units, Softmax activation	Outputs the probability distribution over the ten classes.

Table 1: CNN Architecture for MNIST Classification

Training and pruning setup. We train the base CNN on MNIST using stochastic gradient descent and sparse categorical cross-entropy. We vary the dense pre-training budget from 1 to 10 epochs; for each budget we save a checkpoint of the dense model and then apply each pruning strategy independently to that checkpoint. All pruning methods act only on the convolutional and fully connected layers; pooling, flatten, and dropout layers are left unchanged. The simple pruning baseline removes weights using a greedy backward-selection rule until a target sparsity is reached, followed by a short fine-tuning phase. The Frank–Wolfe (FW) variants use iterative pruning: at each iteration we estimate weight importance on a small subsample of the training data, prune in the FW direction, and fine-tune for a few epochs. In the FW+momentum variant, the FW direction is replaced by an exponential moving average of past gradients, so sparsity increases gradually from an initial value s_{init} to a final target s_{final}. In all subsequent figures, the horizontal axis ("Epochs") should therefore be read as *pre-training budget*: each point corresponds to a different dense checkpoint pruned and fine-tuned independently, not to continued training of a single model.

Results and Analysis We evaluate the pruning methods in terms of test accuracy, test loss, normalized inference time, and the fraction of non-zero weights, as shown in Figures 1–3.

Accuracy and loss. Figure 1 (top) tracks test accuracy as we vary the dense pre-training budget from 1 to 10 epochs. All methods improve with more pre-training, but the ordering between pruning strategies is consistent. With just a single pre-training epoch, the base model and simple pruning reach only about 0.78–0.79 test accuracy, whereas FW and FW+momentum are already in the mid-0.80s. After 10 epochs, FW+momentum reaches roughly 0.94 accuracy, compared to about 0.88 for FW, 0.85 for simple pruning, and a similar range for the unpruned baseline.

Interpreting the x-axis as pre-training budget makes the efficiency gain clear. After only *1–2 pre-training epochs*, FW+momentum already reaches about 0.86–0.88 test accuracy, matching or exceeding the accuracy of the dense baseline and simple pruning baselines even when they are pre-trained for 8–10 epochs (approximately 0.85–0.86). In other words, FW+momentum lets us stop dense pre-training much earlier while still attaining, and often improving on, the final accuracy of models that were fully pre-trained.

The loss curves in Figure 1 (bottom) exhibit the same pattern. FW+momentum yields the lowest test loss at every pre-training budget, ending around 0.28 after 10 epochs, substantially below both the base model and simple pruning, and noticeably below FW without momentum. Simple pruning, in contrast, never clearly improves on the base network and can slightly worsen both accuracy and loss when pre-training is short.

Model complexity. Figure 2 reports the percentage of non-zero parameters remaining after pruning, relative to the base model. The simple pruning baseline hovers close to 50% of the original weights

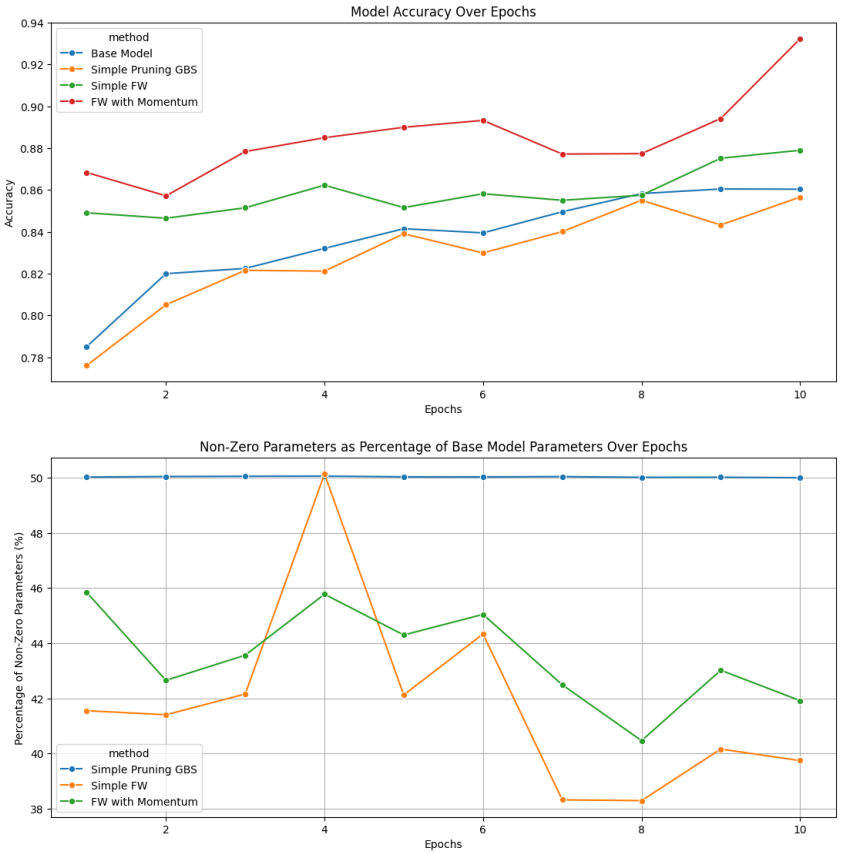

Figure 1: Test accuracy (top) and loss (bottom) of pruned and unpruned models as a function of pre-training budget (epochs).

across all pre-training budgets. Both FW variants produce consistently sparser networks: they remove several additional percentage points of weights beyond the simple method, with FW+momentum always at or below the sparsity achieved by vanilla FW. Even though the absolute gap is modest in this small CNN, it shows that FW+momentum is more aggressive at removing redundant filters while still improving accuracy.

Computational efficiency. Figure 3 shows the per-example inference time on the test set, normalized by the unpruned model. Because our implementation does not exploit sparse kernels, all pruned models incur a runtime overhead of roughly 2–2.5$\times$ compared to the dense baseline. Within that range, however, the FW-based methods are consistently faster than simple pruning, and FW+momentum is typically the fastest of the three pruning strategies. Thus FW-style pruning does not pay an extra runtime penalty over greedy backward selection in our setup; if anything, it slightly reduces inference time for a given level of sparsity.

Key takeaway. Across all pre-training budgets, FW+momentum is the only method that simultaneously increases accuracy, decreases test loss, and yields the sparsest models among the algorithms considered. Simple greedy pruning acts more like a noisy perturbation than a principled compression method: it rarely beats the unpruned baseline, barely reduces the parameter count, and is slower at inference time. For practitioners constrained to short pre-training schedules or limited hardware, FW+momentum therefore appears to be the safest default choice among the pruning strategies we tested.

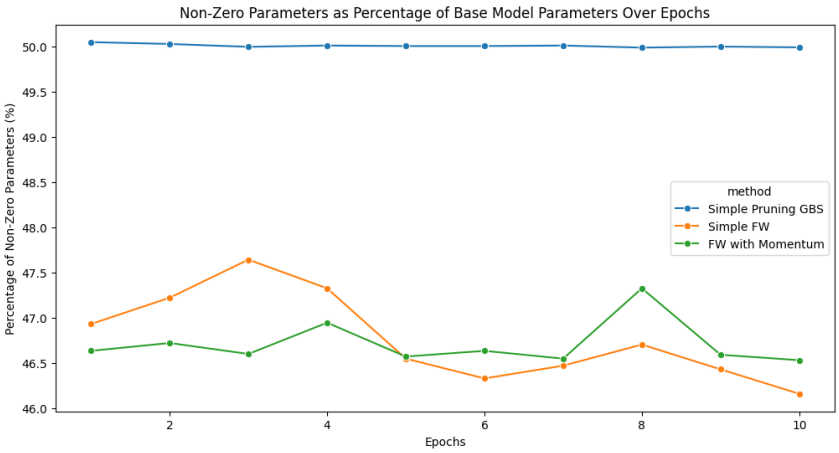

Figure 2: Percentage of non-zero parameters in neural networks post-pruning, relative to the unpruned base model, across different pre-training budgets (epochs).

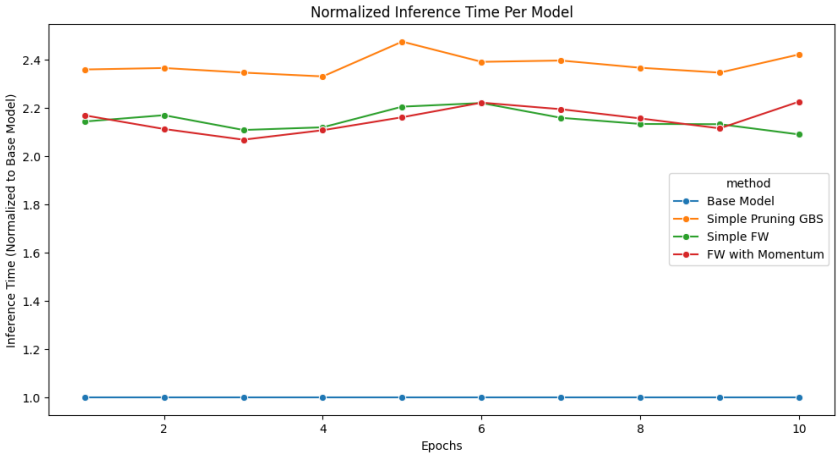

Figure 3: Normalized inference time on the test set (lower is better) for each pruning method, as a function of pre-training budget (epochs).

6 Conclusions

We studied algorithmic variants of Frank–Wolfe for pruning convolutional neural networks, comparing a simple magnitude-based pruning baseline, an FW-style pruning scheme, and an FW method with momentum. On a CNN trained on MNIST, FW+momentum consistently produced the best trade-off among the methods considered: it achieved higher test accuracy and lower loss than both the dense baseline and the simple pruning method, while also yielding the sparsest models and incurring no additional inference-time overhead in our implementation.

Perhaps most importantly, treating epochs as pre-training budget shows that FW+momentum can be applied after only 1–2 epochs of dense training and still match or surpass the accuracy of fully pre-trained baselines. This suggests that full pre-training of the dense model is unnecessary in this setting, which is especially attractive when compute is limited or rapid model iteration is required. Overall, our results indicate that Frank–Wolfe with momentum is a promising and practical tool for compressing CNNs in resource-constrained environments.

7 Further Research

Generalizability and Comparison with Other Pruning Methods Despite the positive results attained with our numerical experiments, it is still to be seen if these results can be generalized to other settings. With access to more computational resources, the performance of FW pruning may be evaluated by using optimizers other than Stochastic Gradient Descent (SGD), such as ADAM. Moreover, to further test the generalizability of our FW pruning methods, their performance could be tested on established architectures such as ResNet, AlexNet, or VGG. Likewise, one could also test the generalizability from varying the dataset by using datasets such as Fashion-MNIST, CIFAR-10, or ImageNet. Overall, despite promising results, more computational resources are needed to test the performance of our pruning methods in a general setting.

Moreover, performing these tests would allow us to compare performance against other pruning methods. To do this, we should suggest using a wide benchmark, such as the one available with ShrinkBench [1], to allow a systematic review of the models' performance.

Structured Pruning The pruning paradigm we considered before, which masks certain weights as zeros, is known as unstructured pruning. On the other hand, we have structured pruning, which removes entire units, such as neurons, layers, or filters from the NN/CNN. That is, structured pruning changes the architecture of the network rather than masking weights. Combining the algorithms developed in this paper with structured pruning may lead to even stronger numerical results as model size and complexity could be reduced to a greater extent.

Greedy Forward Selection [14]. Another pruning methodology that could be considered for future research is greedy forward selection. Instead of traditional pruning methods that use backward elimination to remove redundant and Byzantine neurons from the larger network, greedy forward selection begins with an empty model, and sequentially adds neurons from the original network that result in the greatest immediate decrease in the loss function [14]. The main advantage of this scheme is that there is no need to prune after pre-training, as a subnetwork is found while training. Hence, this approach may be of particular interest in resource-constrained environments.

8 Acknowledgements

This work originated from a project carried out in Spring 2024 under the supervision of Prof. Anastasios Kyrillidis at Rice University. We thank him for helpful feedback and suggestions.

References

[1] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[2] Ting-Wu Chin, Cha Zhang, and Diana Marculescu. Layer-compensated pruning for resource-constrained convolutional neural networks, 2018.

[3] Lijun Ding and Madeleine Udell. Frank-wolfe style algorithms for large scale optimization, 2018.

[4] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956.

[5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[6] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 427–435, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[8] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018.

[9] Mariusz Pietrołaj and Marek Blok. Resource constrained neural network training, Jan 2024.

[10] Sebastian Pokutta. The frank-wolfe algorithm: A short introduction. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 126:3–35, 2024.

[11] Sebastian Pokutta, Christoph Spiegel, and Max Zimmer. Deep neural network training with frank-wolfe. 2020.

[12] Haoyue Wang, Haihao Lu, and Rahul Mazumder. Frank-wolfe methods with an unbounded feasible region and applications to structured learning, 2021.

[13] Cameron R. Wolfe, Fangshuo Liao, Qihan Wang, Junhyung Lyle Kim, and Anastasios Kyrillidis. How much pre-training is enough to discover a good subnetwork? 2023.

[14] Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. 2020.