2024 Design, Automation & Test in Europe Conference (DATE 2024)
Special Initiative "Autonomous Systems Design"

Back to the Future: Reversible Runtime Neural Network Pruning for Safe Autonomous Systems

Danny Abraham[1], Biswadip Maity[1], Bryan Donyanavard[2], and Nikil Dutt[1]
[1] *Department of Computer Science, School of ICS, University of California, Irvine, California, USA*
[2] *Department of Computer Science, San Diego State University, San Diego, California, USA*
{dannya1, maityb, dutt}@uci.edu, bdonyanavard@sdsu.edu

***Abstract*—Neural network pruning has emerged as a technique to reduce the size of networks at the cost of accuracy to enable deployment in resource-constrained systems. However, low-accuracy pruned models may compromise the safety of realtime autonomous systems when encountering unpredictable scenarios, e.g., due to anomalous or emergent behavior. We propose *Back to the Future*: a novel approach that combines pruning with dynamic routing to achieve both latency gains and dynamic reconfiguration to meet desired accuracy at runtime. Our approach enables the pruned model to quickly revert to the full model when unsafe behavior is detected, enhancing safety and reliability. Experimental results demonstrate that our swapping approach is 32× faster than loading the original model from disk, providing seamless reversion to the accurate version of the model, demonstrating its applicability for safe autonomous systems design.**

***Index Terms*—Dynamic networks, pruning, convolutional neural networks, runtime safety**

I. Introduction

Autonomous systems, e.g., autonomous vehicles (AVs) rely on an array of perception tasks such as object detection and lane keeping to navigate the complexities and inconsistencies of modern roads. While vehicles with L2/L3 levels of autonomy can fallback to a human driver if danger is imminent, safety-critical situations can prove catastrophic for driverless L5 AVs if not handled carefully. Thus, the design and development of dependable autonomous systems, even when deploying black-box machine learning models, is critical for safety.

The biggest challenge that AVs face today is they must react to highly dynamic environments under tight realtime latency requirements while accounting for worst-case scenarios (e.g., sensor malfunctions [1]) to ensure safety and reliability. This is achieved using large and complex deep learning models that are challenging to deploy in resource-constrained embedded systems. To meet latency constraints, neural network pruning has emerged as a popular technique for model compression to accommodate the array of complex models in these resource-constrained systems. Pruning aims to reduce the inference latency and memory footprint of deep learning models, but can result in a loss of accuracy that may compromise safety-critical situations where normally a human driver would take over. While fine-tuning techniques can improve the quality of pruned models, they (1) require computationally expensive retraining; (2) lose the original model weights; and (3) the original model architecture cannot be restored without the overhead of a complete reload that can result in missed inferences (shown in Fig. 1). The inability to reconfigure the model at runtime limits its ability to preserve the robustness and safety of the original model in the face of anomalous situations.

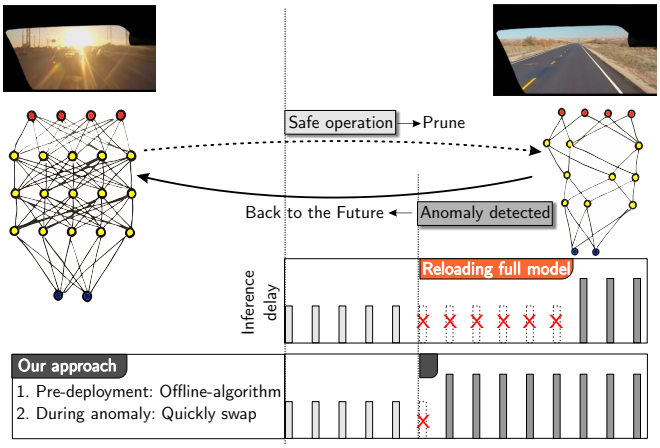

Figure 1: *Back to the Future*: Run pruned model during safe operation but quickly revert to full model during anomaly.

Recent pruning techniques are able to maintain accuracy, but the pruning happens pre-deployment and the network cannot be reconfigured at runtime. Anomaly detection engines for AVs already exist [2], and can detect unexpected events and situations at runtime. These engines monitor the behavior of the agent and monitor changes in the environment and can provide signals, sometimes even proactively, when pruned networks may not suffice. These anomaly signals can be leveraged to determine when the accuracy of the pruned network is insufficient to maintain safe operation.

We present *Back to the Future*, a novel reversible approach that combines the benefits of pruning with dynamic routing to achieve both latency gain and dynamic reconfiguration at runtime in response to anomaly detections. Our *Back to the Future* approach prunes neural network architectures for faster inferences during deployment, but switches back to a pre-pruned safe state without the overhead of reloading the entire model. Our approach (shown in Fig. 1) uses traditional pruning and fine-tuning for model compression, but also introduces a

979-8-3503-4859-0/DATE24/© 2024 EDAA

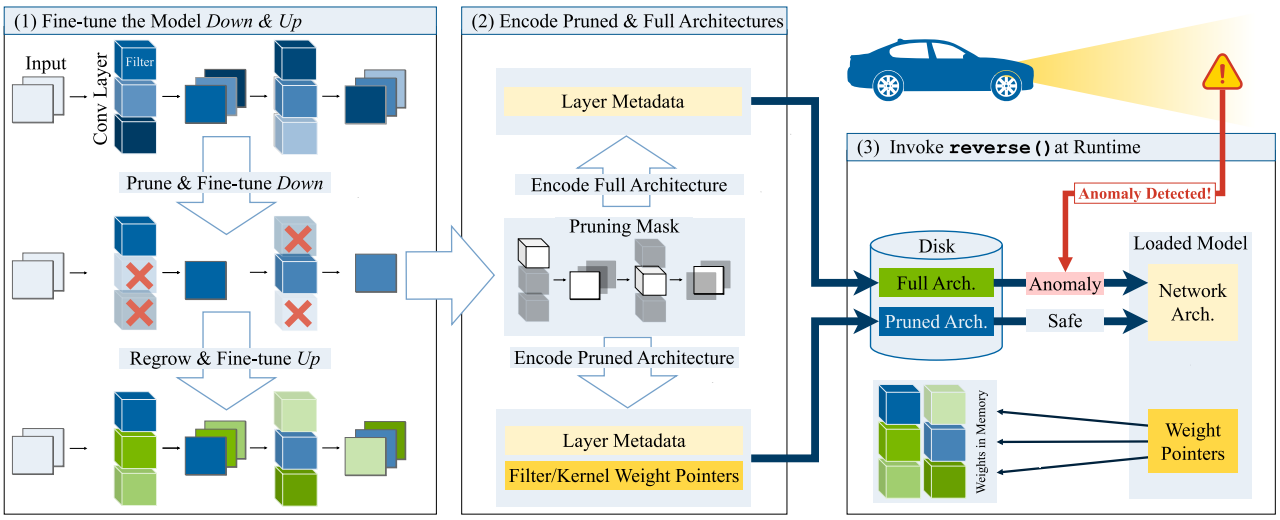

Figure 2: *Back to the Future* approach: (1) fine-tune model down and up, (2) encode pruned and full architectures, and (3) invoke `reverse()` at runtime

novel fine-tuning technique to regrow pruned weights, that we call *fine-tuning up*. *Back to the Future* extends existing pruning techniques to achieve significant model compression, with additional annotated information to decompress the model. At runtime, the annotated information is utilized to preserve robustness by executing a lightweight function and replacing the structure when unsafe or anomalous behavior is detected. By combining the benefits of pruning with dynamic neural architectures, ours is the first approach that enables fast and seamless reversion to the accurate (non-pruned) version of the model, enhancing the safety and reliability of pruned models in real-world scenarios, while also making use of the reduced latency of pruned models.

II. Related Work

Resource-constrained autonomous systems impose the need to reduce the size of deep learning models without sacrificing accuracy or performance. We review two related areas: (1) pruning and (2) dynamic neural networks.

1) Pruning methods: Methods vary in their sparsity structure, scoring method, scheduling, and fine-tuning approaches [3]–[5]. Pruning can be unstructured (individual parameters) or structured (neurons or filters). While both techniques achieve model compression, it is notably more difficult to achieve latency improvements with unstructured pruning. This is because unstructured pruning results in sparse representations (e.g., compressed sparse row formats), and these require special sparse matrix-matrix multiply (SpMM) implementations that only outperform dense matrix-matrix multiply at high sparsities (e.g., 90% or more) [3]. Therefore, in this work we focus solely on structured pruning techniques. Fine-tuning free pruning methodologies have also emerged [6]–[8], but require modifications to the training cycle [8] and are not applicable to pre-trained models [7], and do not support reversibility. More recently, Liu *et al.* proposed a reversible pruning method that simply replaces some of the pruned weights with the mean [6]. While this is somewhat effective, it falls short of the performance achieved by fine-tuning, and cannot fully restore the accuracy on-demand. Alongside pruning, other techniques have garnered attention for their capacity to adapt to changing environments and reduce computational complexity.

2) Dynamic networks: Dynamic networks are an alternative technique to improve the computational efficiency of DNNs during inference. These networks adapt their structures or parameters to the input and can be categorized by the granularity of their adaptation with respect to the inputs, as well as the schedule (inference or training) [9]. In particular, making data-dependent decisions at the inference stage is key to achieving high efficiency. These methods, however, add an inference overhead [10] and in the instance of policy networks, require the training of an additional network [11], or require specialized training to achieve the alternate routes [12]. The overhead of routing decisions during inference can adversely affect real-time constraints and cause unpredictable, and crucially, unsafe behaviors unable to meet realtime constraints.

Our *Back to the Future* approach uses a pruned network at runtime to reduce inference delay, but can dynamically restore the exact model and recover all the lost accuracy during anomalous situations. By combining structured pruning and dynamic routing, our method can be retroactively applied to statically trained networks, offering a compelling solution for achieving computational efficiency without sacrificing accuracy.

III. *Back to the Future* Approach

As shown in Fig. 2, *Back to the Future* consists of an offline portion for pruning and storing the architectures, and a runtime function, `reverse()`, to swap between full and pruned architectures. The offline portion has two main steps: (1) fine-tuning the model down and up to generate a binary mask that partitions the model into a pruned and full model; (2) use the binary mask to encode the pruned model architecture, and encode the full model architecture. At runtime, (3) the `reverse()` function is invoked in response to an anomaly

detection, enabling seamless back-and-forth swapping between the full model and pruned model. Importantly, we only store partial information related to layers that are not shared between the two models, and dynamically swap those stored layers at runtime to load other architectures.

Algorithm 1: Fine-tuning the model down & up

Input : The model architecture $f(x;\cdot)$, the pre-trained model parameters W, and the minimum accuracy threshold t

Output: The fine-tuned weights W_{new} and the binary pruning mask M

1 $M \leftarrow 1^{|W|}$
2 $a \leftarrow eval(f(X_{val}; W))$
3 $M' \leftarrow M, W' \leftarrow W$
4 **while** $a \geq t$ **do**
5 $\quad M \leftarrow M', W \leftarrow W'$
6 $\quad M' \leftarrow prune(M', norm(W'))$
7 $\quad W' \leftarrow fineTune(f(X_{train}; M' \odot W'))$
8 $\quad a \leftarrow eval(f(X_{val}; M' \odot W'))$
9 **end while**
10 $W_u \leftarrow W, W_p \leftarrow W$
11 **for** $i \leftarrow 0$ **to** N **do**
12 $\quad W_p \leftarrow fineTune(f(X_{train}; \overline{M} \odot W_p + M \odot W_u))$
13 **end for**
14 $W_{new} \leftarrow \overline{M} \odot W_p + M \odot W_u$
15 **return** W_{new},M

1) Fine-tuning down & up: The initial phase of *Back to the Future* is partitioning a statically trained network into a pruned model and a full model such that the pruned model's parameters are a subset of the full model's parameters (as shown in Fig. 2). This is to support reversibility and will be discussed in section III-3. The primary aim of our pruning is to mitigate inference latency while supporting reversibility and maintaining accuracy. Consequently, we use structured pruning, employing pruning at the granularity of the filter level. The model representation that results from structured pruning is a small and dense representation. Conventional dense matrix multiply can be used for these filters and has been extensively optimized, providing significant speed-ups even at low levels of sparsity.

Fine-tuning down. The process of producing the pruned model is a conventional pruning and fine-tuning cycle. We delineate this process in Algorithm 1. We define the network architecture as a function $f(x;\cdot)$ and the model with pre-trained parameters W as $f(x;W)$. We first initialize a pruning mask M to all ones (Line 1) such that it has the same shape as the model parameters W. We then begin the pruning cycle, where at every iteration we uniformly prune some portion of the filters with the lowest L^p-norm across all convolutional layers (Line 6). As shown, we do not prune W' directly, instead the pruning scheme is reflected in the pruning mask M'. We then fine-tune the remaining unpruned weights. We accomplish this using the Hadamard product of the model parameters and the mask, $W' \odot M'$, as a new parametrization for the model. The mask fixes the pruned weights to zero to remove their contribution to the model output, consequently, when fine-tuning, only the unpruned weights will be optimized (Line 7). We re-evaluate the pruned network at every step, and continue pruning until the accuracy drops below a given threshold. The fine-tuned weights W, and the mask M, give us our pruned model. The subsequent step involves rebuilding the full model by restoring the pruned weights.

Fine-tuning up. With the pruned model determined, the objective now is to reintegrate the previously pruned weights back into the model, and fine-tune them to complement the pruned model weights. Once again refering to Alg. 1, we begin at Line 10 by storing two copies of the model parameters: W_p denotes the pruned weights, and W_u the unpruned weights. We proceed to fine-tune the model for a number of iterations, but this time we parametrize it with $\overline{M} \odot W_p + M \odot W_u$, such that $\overline{M}$ is the binary complement of M (Line 12). In subsequent fine-tuning steps, only W_p undergoes optimization, while W_u remains unchanged. This complementary masking scheme allows the fine-tuning of the previously pruned weights while preserving the unpruned weights left behind by the *fine-tuning down* phase. The two sets of weights are then combined with the complementary masking scheme to form the unified model parameters W_{new} (Line 14). With these updated parameters and the pruning mask M, we move on to encoding the original and pruned architectures separately.

Algorithm 2: Encoding the pruned architecture

Input : A list of the layers in the neural network $[l_i]$, $i = 1, 2, \cdots, n$, a list of the corresponding masks for each layer $[m_i]$

Output: The pruned architecture encoding: $pruned$

1 $pruned \leftarrow [\,]$
2 **for** $i \leftarrow 0$ **to** $n-1$ **do**
3 $\quad$ **if** $isPruned(l_i)$ ***or*** $isPruned((l_{i-1}))$ **then**
4 $\quad\quad s, p \leftarrow getParameters(l_i)$
5 $\quad\quad k \leftarrow getKernels(m_{i-1})$
6 $\quad\quad f \leftarrow getFilters(m_i)$
7 $\quad\quad pruned.append(s, p, |k|, |f|, k, f)$
8 **end for**
9 **return** $pruned$

2) Encoding the pruned and full architectures: To achieve the desired latency improvements, this step uses the mask to generate an encoding for the pruned architecture. We outline this process in Alg. 2, which outputs the pruned network architecture represented as a 1-dimensional array. We iterate over the network layers and check if they need to be encoded (Line 3). There are two conditions for encoding: (1) if the layer itself has been pruned, or (2) the upstream layer has been pruned. This is because when we prune filters from a convolutional layer, the output channels corresponding to the pruned filters are also eliminated. So, removing filters from one layer necessitates modification in the downstream layers. The algorithm description only considers the simple case where l_i receives inputs from l_{i-1}, but for networks with concatenations and residual blocks, precautions need to be taken to make sure all layers are still compatible with

their immediate upstream and downstream neighbors. We then collect the layer metadata at Line 4 (the stride s, and the padding p), which are necessary to reconstruct the pruned architecture. Next we collect the indices of the kernels to be kept k, and the indices of the unpruned filters f, from the pruning masks (from Alg. 1) of l_{i-1} and l_i respectively (Lines 5-6). This data is collected for each layer and combined into a 1-D array representation of the pruned model.

A similar encoding is produced for the full architecture. However, only 4 values need to be stored per layer: the stride, padding, number of input channels, and number of output channels. The two encodings will be used by `reverse()` to swap the pruned and full models at runtime.

3) Invoke `reverse()` at runtime: Our assumption is that the pruned model will suffice for most cases, and the original model will only be needed when an anomaly is detected (e.g., depth sensor malfunction due to excessive fog, or unexpected stall in pipeline performance [1]). Most of the memory footprint of a convolutional neural network is dominated by the weights, while other parameters, although essential, have a much smaller memory footprint. We keep the weights (shared by both models) in memory, and load the architecture in and out of memory as shown in Fig. 2. While all the weights are loaded in memory at all times, when the pruned architecture is loaded only some of the weights will be executed. The full architecture encoding will be stored on-disk. This encoding is sufficient to rebuild the model since all we have to do for the original architecture is load all the weights into the model. However, the process is slightly more complex for the pruned architecture. In addition to all the information we need to rebuild the original architecture, we also need pointers to the weights in memory to know which subset of weights to load; this is where the kernel and filter indices collected in Alg. 2 will be used. As shown in Alg. 2, we only store data for the layers that exhibit differences between the pruned and full models. The shared and unchanged layers do not need to be stored separately. When switching from the full and pruned models at runtime, we only swap out the layers that need dynamic modifications, while the rest remain unchanged.

IV. Evaluation and Results

A. System Overview

We now evaluate the efficacy of *Back to the Future* to achieve efficient neural network inferencing without sacrificing accuracy when anomalous behavior is detected. Our primary objective is to demonstrate the rapid reversion to the full network by invoking our custom `reverse()` from the pruned model, thereby eliminating the need for reloading the complete network from disk. We utilized the PyTorch framework in Python to implement our proposed approach. All inferences are measured on an Intel Xeon X5680 based workstation.

B. Models and Datasets

We evaluate on two tasks, classification and object detection, and we select two extensively used datasets for these evaluations respectively, CIFAR-100 [13], and COCO [14]. The classification task involves correctly categorizing a sample image into 1 of 100 classes, and the evaluation metric used is accuracy. The object detection task involves recognizing and localizing individual instances of objects within an image and the evaluation metric used is the mean Average Precision (mAP). CIFAR-100 consists of 50K training samples and 10K test samples with an input image size of $32 \times 32 \times 3$. COCO consists of 118K training samples and 5K validation samples. We use pre-trained VGG and ResNet models from [15] for CIFAR100, and we use pre-trained YOLO-v5 models from [16] for COCO. VGG is a feed-forward family of architectures whereas ResNet and YOLO contain residual blocks.

C. Fine-tuning Down & Up

In this section we cover the process of partitioning our pre-trained models into pruned and full using *Back to the Future's* fine-tuning down & up technique. We perform the fine-tuning on eight classification models, four VGG models (VGG11,13,16,19) and four ResNet models (ResNet20,32,44,56), and on the YOLOv5-small (YOLOv5s) object detection model. For the classifiers, we uniformly prune 5% of the filters with the smallest L_∞-norms in all convolution layers. We opt for the L_∞-norm as the importance score due to it outperforming other metrics in our experiments. We then fine-tune the remaining weights for a small number of epochs. For every pruned filter, we also prune the corresponding kernels in the immediate downstream layer, if it exists. Fine-tuning down is repeated iteratively while the drop in accuracy does not exceed 3%. We then regrow the pruned weights and fine-tune up until the accuracy of the model is restored. A similar process is done for YOLOv5s, except the fine-tuning down ends when the mAP drops below the mAP of the YOLOv5-nano model (YOLOv5n, the next model size down in the YOLOv5 family of models).

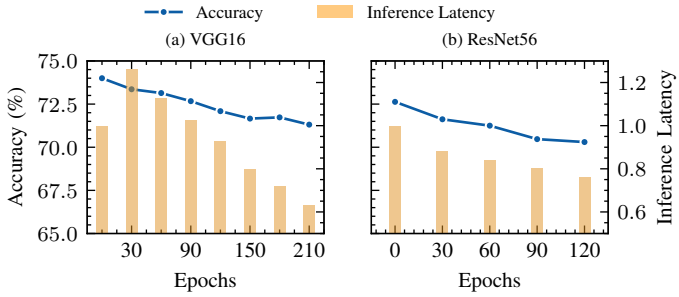

Figure 3: Inference latency drop and accuracy drop while fine-tuning down (a) VGG16 and (b) ResNet56

We document the fine-tuning down procedure for VGG16 and ResNet56 in Fig. 3. Epoch # is shown on the x-axis, accuracy is reported as a percentage, and the inference latency is reported as a proportion of the original model's inference latency. The study highlights that while pruning and fine-tuning filters result in accuracy degradation, the corresponding reduction in inference time is disproportionately greater. For < 3 drop in percentage points of accuracy we observe a 24% and 37% drop in inference time for VGG16 and ResNet56 respectively. Similar fine-runing procedures are completed for all models, and the results are reported in Table I.

	ResNet20	ResNet32	ResNet44	ResNet56	VGG11	VGG13	VGG16	VGG19	YOLOv5-small	YOLOv5-nano
Original Accuracy/mAP@50 (%)	68.83	70.16	71.63	72.63	70.78	74.63	74.0	73.87	56.8	45.7
Sparsity (%)	28.15	35.61	35.69	35.74	66.76	55.31	56.15	56.55	58.59	-
Accuracy/mAP@50 (%)	66.13	67.31	68.86	70.3	68.24	72.03	71.31	70.87	46.8	-
Inference Latency	0.81	0.77	0.77	0.76	0.51	0.64	0.63	0.63	1.15	1.0

Table I: Resulting sparsities and latencies after the fine-tuning down stage for each of the ResNet, VGG, and YOLOv5s models.

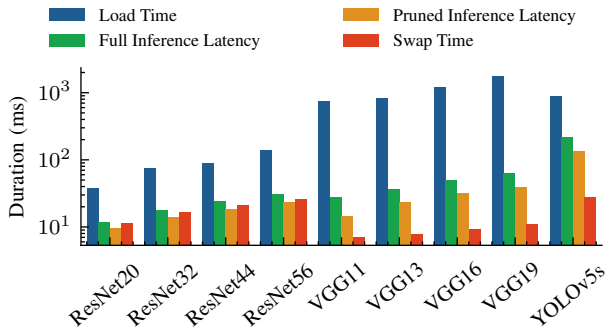

Figure 4: Performance overhead for (1) loading the model from disk, (2) running inference, (3) switching the pruned model, for the CIFAR-100 models, and YOLOv5s

The quality metric reported in Table I is accuracy for the CIFAR models, and mAP@50 for the YOLO models. The inference time of the ResNet and VGG models is reported as a proportion to the original model's inference latency, whereas that of the YOLO models is reported as a proportion of the nano model's inference latency. This is because we are attempting to replace the need for the nano model with the pruned small model. Table I shows disproportionate inference latency gains across the board. For a $< 3\%$ drop in accuracy we achieve anywhere from a 19% to 49% drop in inference latency for the CIFAR models, while also achieving high sparsities ranging from 28.15% to 66.76%. With the pruned YOLOv5s, we achieve a sparsity of nearly 60% and slightly outperform the YOLOv5-nano model, while also achieving a similar inference latency ($1.15\times$). Also, not reported in Table I, the pruned YOLOv5s model achieves $0.60\times$ the inference latency of the original small model. Table I shows that filter pruning can achieve significant inference speedups while only slightly sacrificing the quality of the model.

D. Swapping Performance

To show that our approach is favorable, we compare the time delay of swap time to the time delay of loading the models from disk, and to the respective model's inference delay. The goal is to demonstrate that the overhead of loading the model from disk is many times greater than the inference time, and grows with the size of the model, whereas our swapping approach is even faster than a single inference. This is important because the objective of our approach is to make running approximate models viable for systems where safety and/or resilience are a concern. As shown in Fig. 4, for each model we measure the time it takes to (1) load from disk, (2) perform one inference, and (3) to swap in the full model in place of the pruned model currently running (or vice versa), by `reverse()`. The inference time is calculated by running inference on the respective model's validation set with a batch size of 1 and dividing by the total samples. All metrics are reported in ms on the y-axis and they are grouped together by model on the x-axis. We make two important observations here. First, as expected, *the model load times are many times greater than the respective inference times.* On average, the model load time is $11\times$ greater than the full inference time of the same model. Second, *the swap time is at least on par with the inference time in the case of all the models*, and is even many times less than a single inference in the case of The VGG models, and YOLOv5s. To illustrate the importance of these observations, consider a scenario where the nano model is loaded in memory, but an anomaly is detected, requiring fast loading of a small model for better accuracy. Loading the small model from memory would cost us multiple lost inferences in a potentially dangerous context. When an anomaly is detected the full model needs to be in-memory quickly to ensure safety. With our approach, safety is ensured by having the approximate model be a pruned version of the full model enabling the `reverse()` to complete quickly, taking only 27ms to do so. This represents a $32\times$ reduction compared to loading the full model from disk. It is important to remember that there are two alternatives to our approach: (1) serving two models, one approximate to make use of the reduced resource requirements, and one full model when safety is a concern and the full accuracy is required, and (2) have both models loaded in memory at all times, which is a significant overhead that many resource-constrained systems running multiple applications cannot afford. We discuss this overhead further and the overhead of our approach, in Section IV-F. As shown in Fig. 4, Our approach is able to benefit from running a pruned model, while guaranteeing a quick reversion to the full model when needed, and while only having the equivalent of one model loaded in memory at all times.

E. Effect of No Fine-tuning

Back to the Future necessitates the pruned model weights be a subset of the full weights for quick reversion. Our two-staged fine-tuning approach guarantees this. However, we could meet this requirement if we forego fine-tuning alltogether. In this study, we compare the degradation of the mAP of YOLOv5s while pruning filters, with and without fine-tuning the remaining weights. To give the no fine-tuning approach the best chance, at every pruning step, we greedily prune the filters that have the smallest effect on the validation mAP. The fine-tuned approach is implemented identically to IV-C with uniform pruning across all layers. We use COCO128 for validation. Fig. 5 shows the mAP of the model at each pruning increment.

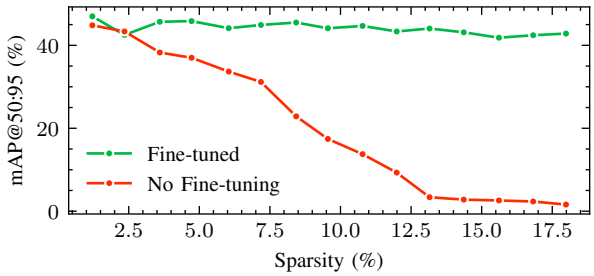

Figure 5: Quality degradation of YOLOv5 while pruning with and without fine-tuning

The x-axis denotes model sparsity, and the y-axis represents the mAP. We observe that the fine-tuned approach maintains a mAP above 40 even at sparisities exceeding 17%, whereas the fine-tuning free pruning causes the mAP to completely degrade by that point. We conclude that in order to achieve any extent of sparsity, certainly the levels we achieve in IV-C, fine-tuning is necessary, and in the absence of fine-tuning neither a significant compression ratio, nor, consequently, a significant inference speedup can be achieved.

F. Back to the Future Storage Overhead

We now discuss our approach's storage overhead and demonstrate the overhead is offset many times over by the storage saved. *Back to the Future* stores the difference between the pruned and the full model, but in order to achieve this extra metadata is needed, namely, the architecture encodings discussed in Section III-2. The storage overhead for each model configuration is shown in Table II. We see the size of the encoded architecture files on disk for YOLOv5s is 5.04 kB and 106.35 kB for the full and pruned architectures respectively. The encoded full architecture requires less space on disk than the pruned architecture because of the necessary filter and kernel index metadata present in the encoded pruned layers. The total overhead of the encoded architecture files, for YOLOv5s, is 111.39 kB. *This is many orders of magnitude smaller than the space required for the full model on disk*, 27.93 MB. A Similar trend is observed in the VGG and ResNet models. We argue that even this negligible overhead is offset by the space saved by not storing two separate networks. The advantage of our approach is the ability to make use of a smaller network to benefit from the reduced latency, while also being able to revert to the full network when safety may be compromised. The alternative to our approach would be running two separate networks with no shared weights. That strategy would either require storing two weight files on disk, or having two models loaded in memory, as discussed at the end of Section IV-D. *Back to the Future*'s strategy of storing two versions of a single network incurs a storage overhead on the order of 100s of kB, whereas storing an additional network will certainly incur an overhead of 10s of MB or more, either on disk, or in memory.

		Our Overhead	
Model	**Full**	**Encoded Full**	**Encoded Pruned**
ResNet (Averaged)	2.27 MB	4.19 kB	20.97 kB
VGG (Averaged)	53.11 MB	1.74 kB	37.33 kB
Yolov5-small	27.93 MB	5.04 kB	106.35 kB

Table II: Storage overhead associated with our approach

V. Conclusions and Future work

We presented *Back to the Future*, a novel approach for deploying deep learning models in resource-constrained real-time autonomous systems that enables seamless switching between approximate and full accuracy models at runtime. *Back to the Future* combines neural network pruning with dynamic routing to achieve both latency gains and dynamic reconfiguration at runtime. Our approach allows the pruned model to quickly revert to the full model when unsafe behavior is detected. This grants safety-critical systems the ability to run approximate pruned models without compromising safety. Experimental results demonstrated that our swapping approach is 32× faster than loading the full model from disk, providing seamless reversion to the accurate version of the model. Our approach opens up new possibilities for deploying efficient and adaptable deep learning models in resource-constrained environments, particularly in the context safety-critical autonomous systems that must respond in real-time to anomalies. paving the way for improved performance and usability of deep learning applications in a wide range of domains. Our future work will focus on better pruning methods and further optimizing the swapping process for different domains.

References

[1] B. Maity *et al.*, "Chauffeur: Benchmark suite for design and end-to-end analysis of self-driving vehicles on embedded systems," *ACM TECS*, vol. 20, no. 5s, 2021.
[2] C. Segler *et al.*, "Anomaly Detection for Advanced Driver Assistance Systems Using Online Feature Selection," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 578–585.
[3] T. Hoefler *et al.*, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.
[4] T. Liang *et al.*, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.
[5] D. Blalock *et al.*, "What is the state of neural network pruning?" in *Proc. of MLSys*, vol. 2, 2020, pp. 129–146.
[6] D. Liu *et al.*, "Pruning on-the-fly: A recoverable pruning method without fine-tuning," *arXiv:2212.12651*, 2022.
[7] Y. Guo *et al.*, "Dynamic network surgery for efficient dnns," *Advances in neural information processing systems*, vol. 29, 2016.
[8] D. Molchanov *et al.*, "Variational dropout sparsifies deep neural networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2498–2507.
[9] Y. Han *et al.*, "Dynamic neural networks: A survey," *IEEE TPAMI*, vol. 44, no. 11, pp. 7436–7456, 2021.
[10] L. Yang *et al.*, "Resolution adaptive networks for efficient inference," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2369–2378.
[11] Z. Chen *et al.*, "You look twice: Gaternet for dynamic filter selection in cnns," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9172–9180.
[12] J. Yu *et al.*, "Slimmable neural networks," *arXiv preprint arXiv:1812.08928*, 2018.
[13] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," 2009.
[14] T.-Y. Lin *et al.*, "Microsoft COCO: Common Objects in Context," in *Proc. of ECCV*. Springer, 2014, pp. 740–755.
[15] Y. Chen, "Pytorch cifar models," https://github.com/charlespwd/project-title, 2023, accessed: June 19, 2023.
[16] Ultralytics, "YOLOv5: PyTorch-based Object Detection," https://github.com/ultralytics/yolov5, accessed: May 19, 2023.