https://doi.org/10.2352/EI.2024.36.3.MOBMU-322

Model Surgery : Run any Neural Network on Embedded Processors

Kunal Ranjan Patel[1,3], Parakh Agarwal[2,3], Manu Mathew[3], Arthur Redfern[3], Debapriya Maji[3], Kumar Desappan[3], Pramod Swami[3], and Do-Kyoung Kwon[3]

[1]k-patel1@ti.com, [2]p-agarwal@ti.com, [3]Texas Instruments

Abstract

We introduce Model Surgery*, a novel approach for optimizing Deep Neural Network (DNN) models for efficient inference on resource-constrained embedded processors. Model Surgery tackles the challenge of deploying complex DNN models on edge devices by selectively pruning or replacing computationally expensive layers with more efficient alternatives. We examined the removal or substitution of layers such as Squeeze-And-Excitation, SiLU, Swish, HSwish, GeLU, and Focus layer to create lightweight "lite" models. Subsequently, these lite models are trained using standard training scripts for optimal performance.*

The benefits of Model Surgery are showcased through the development of several lite models which demonstrate efficient execution on the hardware accelerators of commonly used embedded processors. To quantify the effectiveness of Model Surgery, we conducted a comparison of accuracy and inference time between the original and lite models via training and evaluating them on the Imagenet1K [1] and COCO [6] datasets. Our results suggest that Model Surgery can significantly enhance the applicability and efficiency of DNN models in edge-computing scenarios, paving the way for broader deployment on low-power devices. The source code for model surgery is also publically available as a part of model optimization toolkit at https://github.com/TexasInstruments/edgeai-modeloptimization/tree/main/torchmodelopt .

Motivation

Deep Learning Models have been witnessing significant increase in accuracy over the last few years. This is achieved by use of more training data, improved training techniques, and also by using more sophisticated layers. Many of these models are designed for high power and cloud inference scenarios. These models may not be suitable for efficient inference on the edge using low power embedded devices as they may have suboptimal or unsupported layers. Suboptimal layers could include layers that need high memory throughput compared to the compute time (there by not utilizing the compute resources fully). Suboptimal layers could also include layers that give only slight increase in accuracy compared to the complexity that they bring. Such layers can be replaced or removed to run efficiently on hardware accelerators on embedded devices.

Examples of such layers include Squeeze-And-Excitation [2], SiLU [3], Swish [4], HSwish [12], GeLU [5], Focus [14] etc. This work focuses on methods that can be used to easily remove or replace such suboptimal layers. We call these new models as "lite" models. We call this process of creating lite models as Model Surgery. For example, we could remove Squeeze-And-Excitation, replace SiLU with ReLU, Focus layer with Convolution layer etc.

Here arises the need for the process of "model surgery" to optimize deep learning models for edge devices. This approach involves creating "lite" versions of these models that are more efficient and suitable for low-power embedded devices while trading off some percentage of accuracy. The comparison can be seen in Table 1.

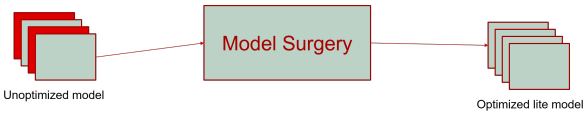

Figure 1: Model Surgery

Proposed Solution

The solution consists of three parts : Iterating through the replacement dictionary, Key Matching using Straight Line Searcher and finally replacing the matched modules using Module Replacer. This heavily uses the functionalities of torch.fx [8] mode. Through this, we are able to generate lite pytorch models which can further be trained and be used for deployment over the edge devices. The individual parts of the solution are explained in the subsections.

TORCH.FX

It is a program capture and transformation library for PyTorch [7] written entirely in Python and optimized for high developer productivity by ML practitioners. It captures programs via symbolic tracing, represents them using a simple 6-instruction python-based IR, and re-generates Python code from the IR to execute it. To avoid the complexities of re-capture for JIT specialization, torch.fx makes no attempt to specialize programs itself, instead relying on the transforms to decide what specializations they want to perform during capture. The process of symbolic tracing can be configured by users to work for more esoteric uses.

Our method is based on the torch.fx library. The model is represented as a Directed Acyclic Graph where the nodes can be torch Operators, Functions, or Modules.

```
import torch
from torch.fx import symbolic_trace, GraphModule

def my_func(x):
  return torch.relu(x).neg()

# Program capture via symbolic tracing
traced : GraphModule = symbolic_trace(my_func)
for n in traced.graph.nodes:
  print(f'{n.name} = {n.op} target={n.target} args={n.args}')
"""
x = placeholder target=x args=()
relu = call_function target=<built-in method relu ...> args=(x,)
neg = call_method target=neg args=(relu,)
output = output target=output args=(neg,)
"""

print(traced.code)
"""
def forward(self, x):
    relu = torch.relu(x);  x = None
    neg = relu.neg();  relu = None
    return neg
"""
```

Figure 2: torch.fx captures programs using symbolic tracing into a simple IR and generates Python code from that IR.

Replacement Dictionary

The input for the replacement operations in the model surgery is a replacement dictionary with source patterns as keys and replacement patterns as values. The source as well as the replacement patterns could be a Type (or Class) of a torch Module, an instance of a torch module, an operator, or a function. Here the user need to ensure that the model should be traceable after the the source pattern has been replaced by replacement pattern.

For example, a simple replacement dictionary for MobileNetV3 could be:

```
replacement_dict = {torch.nn.Hardswish(): torch.nn.ReLU(),
                    torch.nn.Hardsigmoid(): torch.nn.ReLU(),
                    torchvision.ops.SqueezeExcitation():torch.nn.Identity()}
```

Figure 3: Example for Replacement Dictionary

All the keys in the replacement dictionary are iterated over, to perform the corresponding replacement operations. Each replacement may happen multiple times in a model as a source pattern may occur multiple times in the model. Figure 4 shows how the replacements are done.

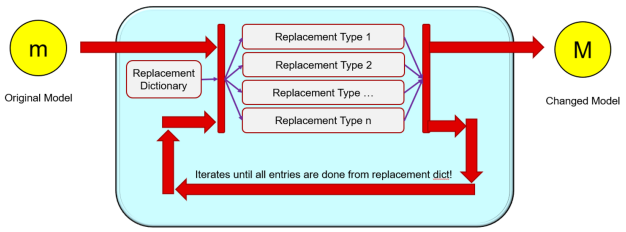

Figure 4: Iterative Approach of the Model Surgery API

This is just an example and in the Figures 5, 6 and 7, we show some of the other possible use cases of this approach.

Straight Chain Searcher

We developed an algorithm to identify straight chain patterns within a graph. The algorithm, straight_chain_searcher, operates on two GraphModule objects (which are the symbolically traced models): the main module and the pattern module.

The straight_chain_searcher algorithm begins by iterating through the nodes of the main module and the pattern module. When a match is found i.e. corresponding nodes are found to be same, the algorithm records the input and output nodes of the matched pattern.

The algorithm uses a sliding window approach, similar to pattern searching in a list, to traverse the main module. If a node does not match the current pattern node, the algorithm resets the pattern index and shifts the window in the main module. If a previously matched pattern is found again, the algorithm records the start of the second match and continues the search from there if the current pattern search fails.

The algorithm returns a list of tuples, each containing the input and output nodes of a matched pattern. This approach allows for efficient and effective identification of straight chain patterns within a graph, contributing to our understanding and manipulation of complex graph structures.

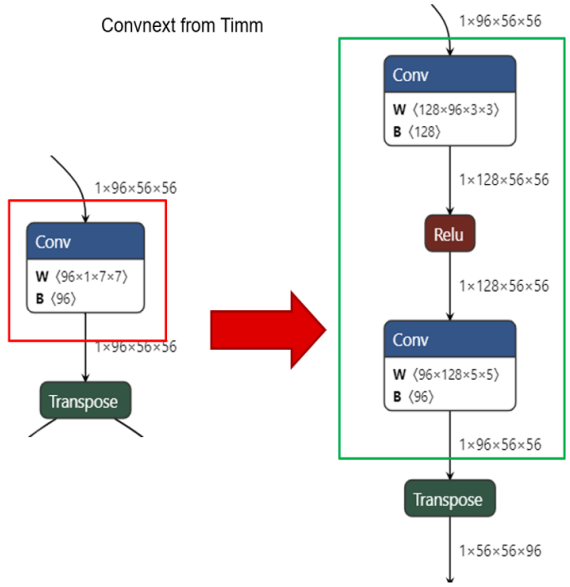

Figure 5: Result of Model Surgery in Convnext Model from TIMM

Module Replacer

Once, we find a match, we replace the pattern with the corresponding value from the replacement dictionary. It operates by identifying a pattern of nodes from the start node to the end node and replacing them with the provided replacement module. This replacement is carried out for all such matched keys in the original model. The function is versatile and can handle different cases based on the type of operation (call_function, call_method, call_module) and the number of operational nodes in the pattern and replacement. In some cases, the replacement pattern is a simple module, while in other cases, it could be a more complicated function. For example, in Figure 5, we replace a 7x7

DNN Model	Original Version		Lite Version			Speed up Factor on hardware accelerator
Classification Networks	Accuracy (Top-1 Accuracy)	CPU inference time (msec)	Accuracy (Top-1 Accuracy)	CPU inference time (msec)	HW accelerated inference time (msec)	
Mobilenet V2 [11] [9]	72.154	121.23	72.88	119.60	2.27	52.69
MobilenetV3-Large [12] [9]	75.274	103.40	71.7*	88.08	4.40	20.02
EfficientNet_B0 [13] [9]	77.69	219.61	73.57*	163.97	2.63	62.35
EfficientNet_B1 [13] [9]	79.83	314.24	74.49*	243.58	3.52	69.20
Object Detection Networks	Accuracy (mAP Accuracy)	CPU inference time (msec)	Accuracy (mAP Accuracy)	CPU inference time (msec)	HW accelerated inference time (msec)	Speed up Factor on hardware accelerator
YoloX-Tiny [10]	32.8	839.65	30.5	777.59	5.20	149.54

*Networks trained for 150 epochs against the suggested 600 epochs.

convolution layer with a sequence of 3x3 convolution layers, followed by a ReLU and a 5x5 convolution layer after that. Here, the replacement functions generated the replacement pattern on the fly considering the number of channels in the source pattern.

Surgery Examples

Here in Figure 6, we can see the Squeeze and Excitation layer is replaced with an identity layer in the Mobilenet V3 model of Torchvision [9] [12] .

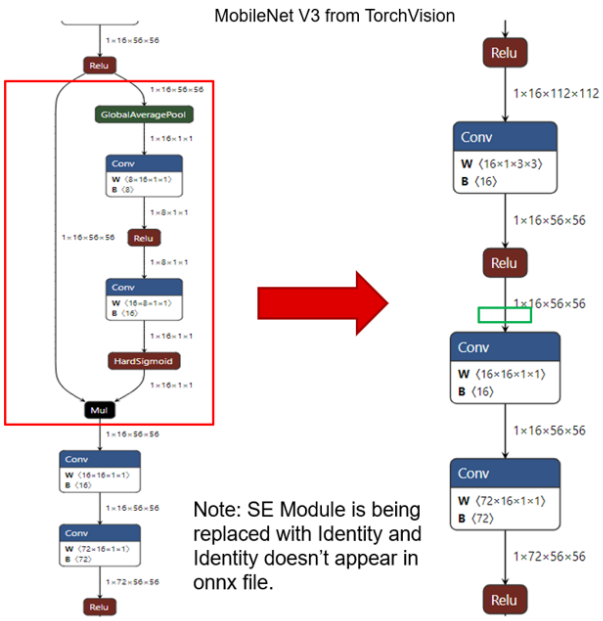

Figure 6: Replacing SE Module to Identity in the Mobilenet V3 Model from Torchvision

Here in Figure 7, the LayerNorm layer is replaced with a BatchNorm layer which is later fused into the above convolution layer of the Convnext model of TIMM.

Summary

Results

The original versions of some of the models are not suitable for Hardware-accelerated inference. In some

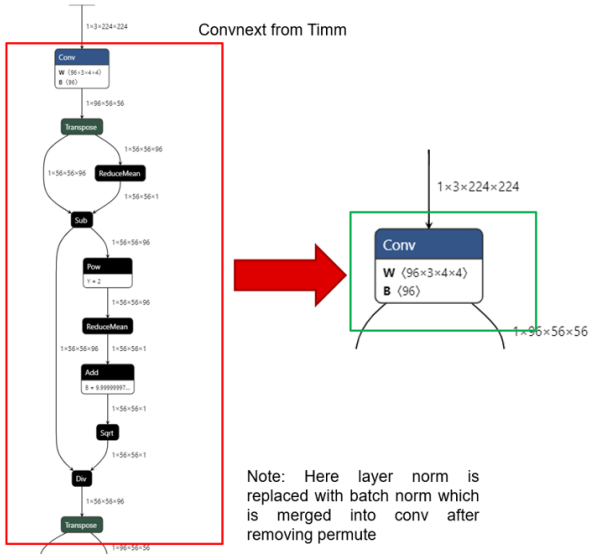

Figure 7: Replacing LayerNorm to BatchNorm in Convnext Model from TIMM

cases, they run with very high inference time while in other cases it may not run at all. The speedup factor is the comparison of the inference time of the original model and the lite model. The inference was run on AM68A SoC from Texas Instruments.

Here we show the results for a few image classification networks [9] and YoloX-Tiny for object detection [10]. The same training regime as original model were used, however for MobilenetV3 [12] [9], and EfficientNet [13] [9] networks, instead of 600 epochs for training, 150 epochs were used which results in accuracy degradation. Some part of this could be reclaimed on training for 600 epochs. The classification models have been trained on the Imagenet1K dataset using the torchvision [9] source code and the object detection models have been trained on the COCO dataset using mmyolo [10] source code. The Top 1 accuracy metric has been used for the classification networks whereas the mAP metric [0.5:0.95]% is reported as defined by the COCO dataset.

Conclusion

The ability to do the surgery without manually modifying the model, has been implemented, and will be a great advantage for the user because the models that would not have otherwise run on the devices, can now run efficiently with model surgery.

It is model-independent and could be used for a wide variety of different models as the user can also define the replacement patterns (source and replacement patterns) according to the specific use case as well. This helps to support a much larger set of models efficiently compared to what was possible without this method.

The method searches for each of the keys in the model's Pytorch graph and replaces them with the corresponding value keeping all the connections in the model intact. This facilitates changing the required connections without the actual need to interfere with the Python code. Through this method, any module/function can be changed to any module/function as long as the inputs and outputs match. This makes this model, model/module independent. The implementation of this method has been released open source and the models will also be made available open source.

The Model Zoo could be collaborative space to host embedded friendly models. We have not tested transformer networks, LSTM based networks yet. We also plan on implementing the method smart enough, that it can automatically change the required modules with the ones that are supported by mobile devices without the need to manually specify both the patterns.

References

[1] J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, ImageNet: A large-scale hierarchical image database, 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[2] J. Hu, L. Shen and G. Sun, "Squeeze-and-Excitation Networks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 7132-7141, doi: 10.1109/CVPR.2018.00745.

[3] Elfwing, S., Uchibe, E., & Doya, K. (2017). Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning. Neural networks : the official journal of the International Neural Network Society, 107, 3-11.

[4] Ramachandran, P., Zoph, B., & Le, Q.V. (2018). Searching for Activation Functions. ArXiv, abs/1710.05941.

[5] Hendrycks, D., & Gimpel, K. (2016). Gaussian Error Linear Units (GELUs). arXiv: Learning.

[6] Tsung-Yi Lin et al., 2014.Microsoft COCO: Common Objects in Context. CoRR, abs/1405.0312

[7] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc.

[8] James K. Reed , Zachary DeVito , Horace He , Ansley Ussery , Jason Ansel Torch.fx: Practical Program Capture and Transformation for Deep Learning in Python, MLSys 2022.

[9] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the machine-vision package of torch. In Proceedings of the 18th ACM international conference on Multimedia (MM '10). Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[10] MMYOLO Contributors. MMYOLO: OpenMMLab YOLO series toolbox and benchmark. In github.com/open-mmlab/mmyolo, 2022

[11] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. -C. Chen, *MobileNetV2: Inverted Residuals and Linear Bottlenecks*, 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 4510-4520, doi: 10.1109/CVPR.2018.00474.

[12] A. Howard et al., Searching for MobileNetV3, 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Korea (South), 2019, pp. 1314-1324, doi: 10.1109/ICCV.2019.00140.

[13] Mingxing Tan, Quoc V. Le, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, InInternational conference on machine learning 2019 May 24 (pp. 6105-6114). PMLR.

[14] C. -Y. Wang, A. Bochkovskiy and H. -Y. M. Liao, YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors, 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Vancouver, BC, Canada, 2023, pp. 7464-7475, doi: 10.1109/CVPR52729.2023.00721.