Optimizing Deep Learning Inference on a MacBook Pro

Adejuwon Fasanya
ajf2243@columbia.edu

***Abstract*—As both the size and demand for SOTA deep learning models grows inference optimization becomes and increasingly important field of research. While often intended for use on GPUs some use cases, such as inference on edge devices, requires methods that perform well on CPU. In this paper, we explore three CPU conscious inference time optimizations, model compilation, integer quantization, and structured pruning. We benchmark a convolutional model running on the CPU of a 2018 MacBook Pro, meant to serve as a stand-in for edge devices, before and after applying these optimizations. We demonstrate a 16x speed up of execution time with minimal loss to accuracy, showing the effectiveness of these optimizations allowing even relatively simple hardware to achieve real-time performance.**

I. Introduction & Motivation

Optimizing model inference is an increasingly important area of research in the field of deep learning. The past few years have seen an explosion in both the capability and size of SOTA deep learning models. This has also driven an increase in the demand for these models, now requiring them to be deployed and available and larger scales then ever before. Additionally some models need to be deployed on devices with limited compute hardware, such as bioacoustic models being deployed on audio monitoring systems in the field. Fulfilling these demands often requires significant post-training efforts to optimize a model for inference without major sacrifices in accuracy.

Model compilation is a widely supported method which aims to capture the compute graph of a model and JIT-compile the code into more optimized kernels.

Model pruning attempts to shrink the model's parameter count, and thus its computational demand, by removing unimportant parameters that have minimal impact on performance.

Model quantization converts the parameters of a model into smaller representations that are easy to perform operations on.

In this paper we will apply these three optimization methods and analyze their impacts on the inference times of a deep learning model.

II. Model, Data and Hardware Description

A. Model

Our baseline model is a relatively simple convolutional neural network. It consists of 6 3x3 filter layers with output channel dimensions of
$16 \rightarrow 64 \rightarrow 256 \rightarrow 1024 \rightarrow 2048 \rightarrow 4192$
Each convolutional layer is followed by a 2x2 max pooling operation. All but the last convolutionl layer utilize zero-padding (which ensures the last layer is a 4192x1x1 image). After this the 4192 channels are flattened into a vector and passed through 4 fully-connected linear layers. These layers have the following output dimensions
$2048 \rightarrow 1024 \rightarrow 256 \rightarrow 7$
Outputs of all but the last layer are passed through ReLU activation functions. This is model is intentionally simplistic and overparamaterized for the training data it will be applied to, allowing greater headroom with which to experiment for our optimizations.

B. Data Description

The dataset we train our model on is FER2013. FER2013 is a facial recognition dataset consisting of 48x48 pixel grayscale images of human faces. Each face is labeled with one of 7 possible emotions, Angry, Disgust, Fear, Happy, Sad, Surprise, Neutral. The dataset consists of 28709 training examples and 3589 test examples. For the purpose of our experiments we upscale the images to be 96x96 to make the problem more computationally demanding.

C. Hardware

Our model is trained on a GCP n1-standard-8 instance equipped with an Nvidia Tesla T4 GPU. However all inference benchmarking is performed on the CPU of a 2018 MacBook Pro. The relevant specs are as follows.

- CPU: 2.3 GHz Quad-Core Intel Core I5 (Gen 8)
- 8GB 2133MHz LPDDR3 RAM

III. Training and Profiling Method

A. Training

As previously mentioned model training took place on the GPU of a GCP VM instance. Since this hardware was used solely for the purpose of training and was not the intended target for our optimizations we do not report performance metrics during training. The training curves of the model are displayed in Fig. 1. For the purposes of our experiment we used the took the checkpoint at epoch 6 as our baseline trained model, as test loss began to grow signfificantly after this epoch. This checkpoint achieved a test accuracy of 57.3%. Notably the divergence between training and test loss indicates that our model is likely overfitting due to overparamaterization, which suggests significant headroom for pruning methods.

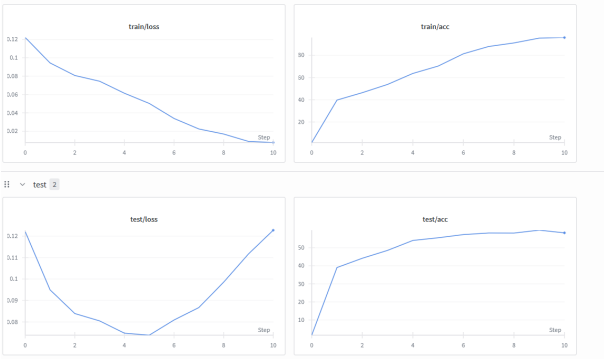

Fig. 1. Loss and Accuracy curves from base model training

B. Profiling Method

For profiling we mainly concern ourselves directly with the inference time of the model. We measure this under two conditions. For one benchmark we simply iterate through the test data and measure model inference time per image. We then use this to determine an average inference time and FPS over the dataset. We also use Pytorch's builtin profiler to track the cpu time and memory usage of individual functions for more granular analysis. Additionally we run a live demo where we use our MacBook's builtin webcam to capture video and have the mode run inference on every frame. We perform similar measurements of inference time and FPS in this demo as our analog for performance in deployment conditions.

IV. Performance Tuning Methodology

As previously stated we rely on three main methods for optimizing our model's inference speed which we will describe in greater detail here.

A. Model Compilation

In Pytorch model compilation is generally very simple for most models, only requiring a call to the builtin function model.compile(). When the model is called Pytorch will trace the model's compute graph and attempt to optimize it as much as possible through techniques such as JIT-compilation. This can lead to significant speed but comes with an initial overhead cost. Building the compute graph can be expensive and thus initial calls to the model may experience slowdown. Additionally if later calls introduce inputs that require the computational graph to be expanded/modified (e.g due to control flow) these calls will also experience slowdown though is not a concern in our case. To account for this overhead when we profile all versions of our models we first perform an initial pass over the data to serve as a warm start.

B. Pruning

Pruning is a technique that aims to remove low impact parameters from our model reducing computational workload with minimal impact on model accuracy. Pruning can be both structured, in which entire dimensions of a module are removed at once, or unstructured, where individual elements are removed (assigned to zero). The former results in modules with smaller dimensionality while the latter increases parameter sparsity while maintaining their shape. While some modern GPUs are equipped with accelerators to take advantage of sparse matrices our relatively old CPU certainly isn't and thus we rely on structured pruning to get meaningful speedup. This leaves two questions, which dimensions do we prune and how many dimensions do we prune for each module?

For the former we use the dimensions L2 norm as the metric by which we measure importance, with the assumption that dimensions that have small values, and thus small L2 norms, are less important.
For the latter we perform sensitivity study to determine what ratio of dimensions we should remove for each module. In our study we iterated through each module and individually pruned the modules with ratios of of 10-90% in 10% increments and recorded the resulting model accuracy. The results of this can be seen in Fig. 2. A cutoff of 50% accuracy was chosen, so in

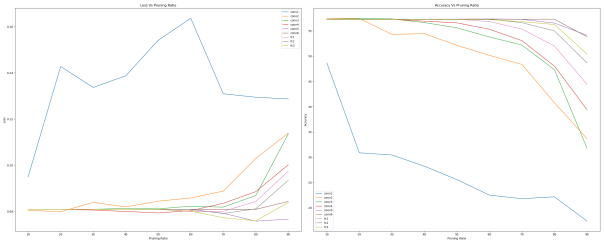

Fig. 2. Results of Sensitivity Study

our pruned model each module is pruned by the largest ratio that didn't drop model accuracy below 50%. Most modules could be pruned by as much as 60-80% showing that the original model was likely overparamterized. After pruning the model accuracy dropped to 36.8% and underwent retraining to compensate for the lost parameters, the results of which can be seen in Fig. 3. Going forward we chose the checkpoint

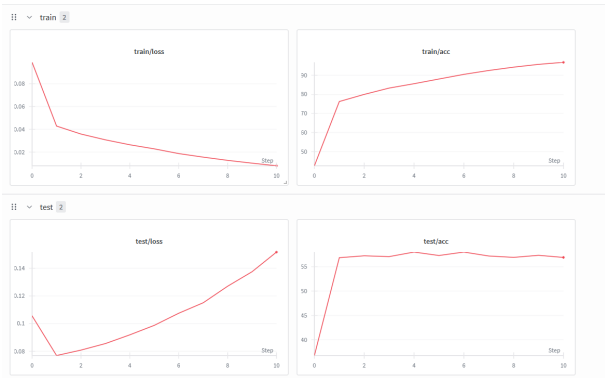

Fig. 3. Loss and Accuracy curves of Pruned Model Training

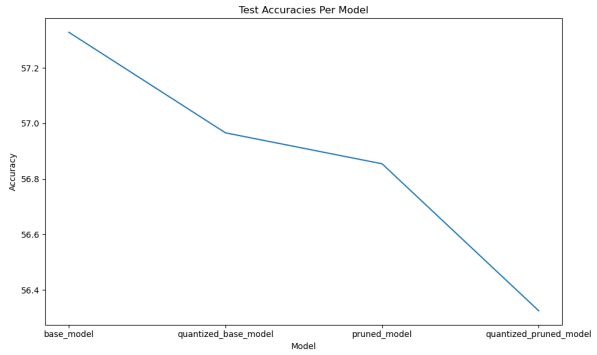

Fig. 4. Test Accuracy of Optimized Models

after just one epoch, as most accuracy was recovered and test loss increased significantly after this point. This checkpoint achieved a test accuracy of 56.8% showing a accuracy drop of just 0.5%. In total the number of parameters in our model was reduced from 109.5M to 5.7M, a decrease of nearly 95%.

C. Quantization

Quantization is a broad set of techniques that aim to convert the individual parameters of a model from their original representation, typically 32-bit floating point numbers, to smaller faster representations, such as 8-bit floats or integers, to speed up inference while maintaining accuracy. In our case integer representations are the most beneficial as

1) Our CPU does not have the same accelerators that modern GPUs have for smaller floating point representations. Some CPUs lack the architecture or instruction set to perform these operation, and thus we'd be adding the overhead of converting back to standard 32-bit floats.
2) SIMD instructions on CPU can take better advantage of smaller integer formats, for similar reasons to above.

We apply quantization to both the base and pruned versions of our models, the results of which will be discussed in the next section.

V. Experimental Results with Analysis

First we will compare the test accuracy of our 4 models (base model, quantized model, pruned model, pruned + quantized model). The results of this can be seen in Fig. 4 We can see that our optimizations result in minor drops in accuracy, with our final model (pruned + quantized) achieving an accuracy of about 56.3% just 1% lower than the base model. In Fig. 5 we can see the resulting speedups for each version of the model, split between compiled and uncompiled versions of the models.

Somewhat surprisingly we see that for the base model model compilation significantly slowed down inference time while for the other models its impact was minimal but often still ran slower. Given that we ran our warm start before taking measurements it is possible there is still some existing overhead associated with model compilation that, for our

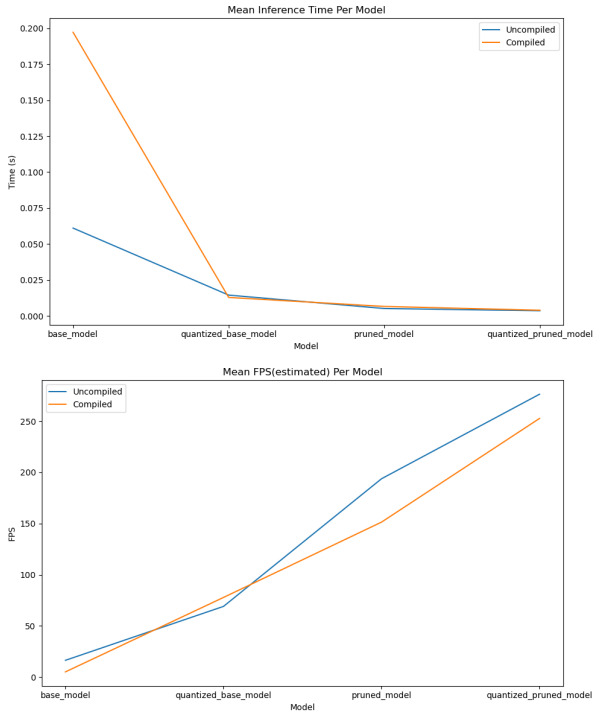

Fig. 5. Loss and Accuracy curves of Pruned Model Training

particular machine and model, slows down inference instead of accelerating it. We focus on analysis of the uncompiled model going forward.

However we can see that the pruning and quantization methods are both very successful in speeding up inference with specific values reported in Table . I. We can see

TABLE I
Execution Times for Uncompiled Models

	Base	**Quantized**	**Pruned**	**Both**
Inference Time(s)	0.0611	0.0145	0.0051	0.0036
FPS	16.37	69.03	193.77	276.38

that for our experiments pruning had greater individual impact on performance than quantization which is somewhat expected given that pruning removed 95% of our parameters/computations. However pruning relies on our model being somewhat overparamaterized for the problem and becomes less impactful if our model complexity is more aligned with the problem and thus can't afford to prune as many parameters. However even in this scenario quantization can still provide significant speed up as seen in our experiments where quantizing the already pruned model provides additional speedup with minimal cost to accuracy.

Results from the live demo, shown in Fig. 6, are a mixed bag. On one hand we do see meaningful speedup going from single digit FPS in the base model to 200+ FPS in

Fig. 6. Base and Optimized Live Demo

the optimized model. However accuracy of all models are fairly poor in the demo. This is likely caused by distributional shift from the training set to the live demo which could be caused by a variety of differences such as camera quality, down sampling method, grayscale conversion, environment etc. Our dataset, model and training procedure were all fairly simple so this failure to generalize is understandable. Still from a performance standpoint we more than achieve our goal of real time performance.

VI. Conclusion Future Work

In this report we explored the impact of three common inference optimization techniques, model compilation, pruning and quantization. We apply these methods to CNN trained on the FER2013 dataset and benchmark CPU inference performance on a 2018 MacBook Pro. We observe significant speed up with minimal accuracy loss when applying quantization and pruning while model compilation generally has minimal or sometimes negative impact for our specific scenario.

There remains space for significant future work on this project. As previously mentioned the accuracy of our base model could be greatly improved, both on it's test data and the live demo, with more sophisticated models and training procedures. Another investigation of interest would be trying to track down why model compilation failed to produce performance gains in our scenario and what could be done to improve it. Finally there are additional optimizations such as distillation, quantization aware training or more advanced pruning methods that all could be applied and experimented with on our machine.

References

[1] Dumitru, Ian Goodfellow, Will Cukierski, and Yoshua Bengio. Challenges in Representation Learning: Facial Expression Recognition Challenge. https://kaggle.com/competitions/challenges-in-representation-learning-facial-expression-recognition-challenge, 2013. Kaggle.

[2] Gongfan Fang and Xinyin Ma and Mingli Song and Michael Bi Mi and Xinchao Wang. DepGraph: Towards Any Structural Pruning. https://arxiv.org/abs/2301.12900 2023