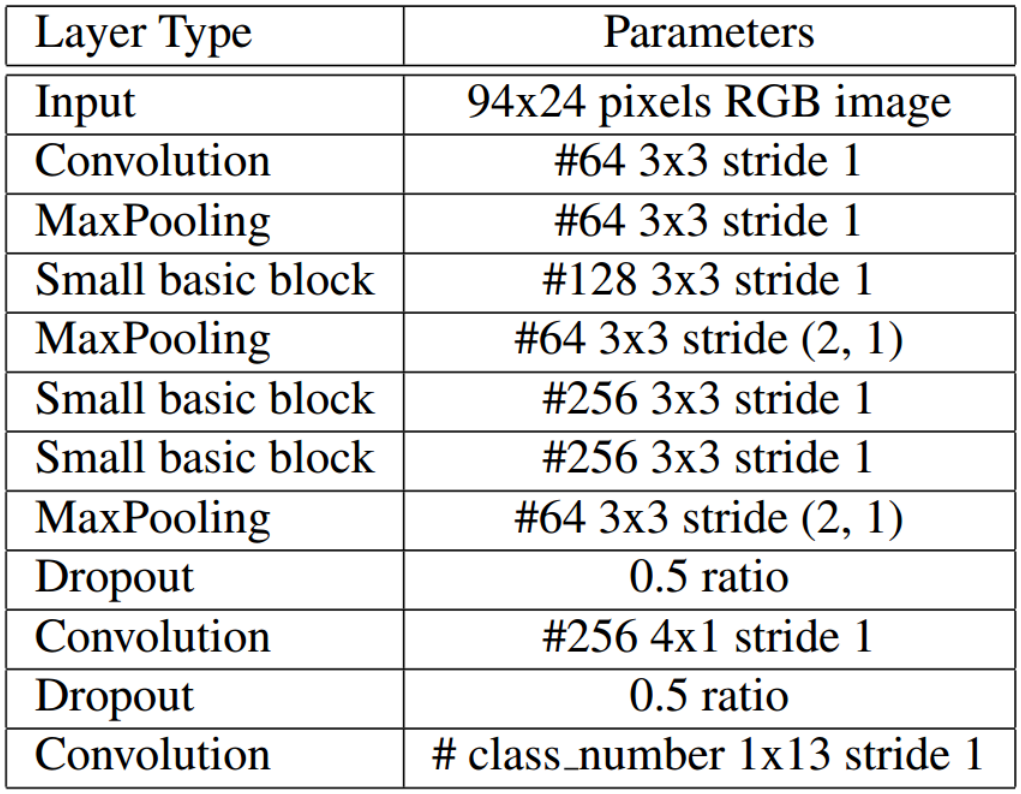
CSC 791 Real-Time AI and Machine Learning Systems
Final Report
Obblivignes KV

Applying Model and MLC Optimizations on the LPRNet Model

Link to GitHub Repository: https://github.com/vignes-12/csc-791-realtime-ai-final-project

What is the original DNN model you chose to use?

The original DNN model that I chose to use was the original LPRNet structure, which can be seen in the screenshot below. I ran all experiments on the CPU in order to run on Google Colab as there was limited compute power available on CUDA or other environments.



Layer Type	Parameters
Input	94x24 pixels RGB image
Convolution	#64 3x3 stride 1
MaxPooling	#64 3x3 stride 1
Small basic block	#128 3x3 stride 1
MaxPooling	#64 3x3 stride (2, 1)
Small basic block	#256 3x3 stride 1
Small basic block	#256 3x3 stride 1
MaxPooling	#64 3x3 stride (2, 1)
Dropout	0.5 ratio
Convolution	#256 4x1 stride 1
Dropout	0.5 ratio
Convolution	# class_number 1x13 stride 1

What optimizations did you apply to the model to improve its accuracy, speed, and space efficiency?

I chose to apply 4 different optimization techniques, two model optimizations and two MLC optimizations. With regards to the model optimizations, as we did not have a dataset to train on to recover the model accuracy, I decided to apply filter-level pruning without any further training. After testing different sparsity level, I was only able to reduce the sparsity factor by 1.45%. By applying this method, I was able to reduce the inference time by 11.7

seconds while maintaining the same accuracy of 89.9 percent and model size of 1.707 MB on ONNX and 1.827 MB on PyTorch.

Next, I applied post-training static quantization, where I needed to encapsulate the MaxPool3D in a contiguous block in order to enable precise control over quantized and dequantized data flow. I also calibrated the model across the dataset in order to prepare the model for quantization before quantizing it using the fbgemm backend to convert it to a lower-precision int8 inference. By applying quantization, I was able to reduce the inference time dramatically to 27.5 seconds however the accuracy also decreased to 72.3%. However, due to the way ONNX saves model, the parameter changes do not get affected so the model size is still reflected as 1.707 MB. However, by saving it using PyTorch, I get a significant model size decrease to 0.536 MB.

However, I was unable to convert this quantization method to TVM due to it's lack of support for certain layers, such as QuantizedBatchNorm2D. Hence, I decided to apply the baseline ONNX model to TVM to determine how it can optimize on the baseline model. By converting the ONNX model to TVM, I already achieved significant improvements in inference time (which decreased from 203.9 ms to 47.9 ms), but accuracy reduced slightly as well from 89.9% to 89.7%. After converting the ONNX model to TVM, I applied a manual complex computation graph optimization pipeline in two optimization stages. The first stage included graph simplification with the SimplifyInference (which simplifies inference computations such as BatchNorm folding), FoldConstant (which folds constant computation into one), and FoldScaleAxis (which folds scaling factors) operators, as well as graph pruning with the DeadCodeElimination (which eliminates unused code) operator. This optimization actually helped improve my model somewhat as well, as the accuracy improved from 89.9% to 90.1% and decreasing the inference time from 47.9 ms to 42.1 ms. The second stage included computation optimizations with FuseOps (fusing Conv2D, BatchNorm, and ReLU), AlterOpLayout (which alters the operation layout based on the target machine), EliminateCommonSubexpr (which eliminates the need to calculate the same subexpression multiple times), and ConvertLayout (which converts the layout of Convolution Layers for better CPU optimization). This optimization further improved the model's performance, although slightly, as the inference time went to 41.8 ms from 42.1 ms and the accuracy went slightly higher from 90.1% to 90.2%.

After applying those optimizations, I applied AutoTVM to optimize the hardware on the CPU beyond what I had manually adjusted, but Google Colab kept crashing each time that I ran the tuning process. I believe that this is due to lack of enough resources to run the auto-tuning process, so I decided to neglect that part of TVM optimizations.

What is the performance (the three metrics) of the original model and the optimized model?

Unfortunately, saving the model to ONNX did not change the model size due to the way that ONNX stores the models, which does not take into account the static quantization performed. Hence, I decided to save the models in PyTorch as well, which clearly shows the difference in model size for quantization. Nonetheless, the performance of all levels of optimization are shown in the tables below.

Model Optimizations

Optimization Level	Accuracy (%)	Inference Time (ms)	Model Size ONNX (MB)	Model Size PyTorch (MB)
Baseline Model	89.9	203.9	1.707	1.827
After Filter-Level Pruning	89.9	183.2	1.707	1.827
After Post-Training Static Quantization	72.3	**27.5**	1.707	**0.536**

MLC Optimizations:

Optimization Level	Accuracy (%)	Inference Time (ms)	Model Size ONNX (MB)
Baseline Model	89.9	203.9	1.707
After Importing to TVM	89.7	47.9	1.707
After Graph Simplification/Pruning TVM Optimizations	90.1	42.1	1.707
After Computation/Layout TVM Optimizations	**90.2**	41.8	1.707

As we can see from the tables below, I believe that pruning had the least impact out of all the optimizations, as it only reduced the inference time. On the other hand, quantization improved the inference time and model size significantly, although at the cost of accuracy. With regards to MLC, bringing the model into TVM itself improved performance, but our own optimizations, especially the Graph simplification ones, further improved on the model itself. As I have marked in bold, I believe that quantization performed the best with inference time and model size within PyTorch, but the TVM optimizations retained the highest accuracy while also reducing inference time from the baseline model.

Unfortunately, due to some compatibility issues with some of the Quantized Layers, I was unable to bring the quantized model into TVM to get the benefits of both worlds due to a lack of support for quantized layers in TVM. I would believe that if it were possible, we could have compounded the improvements in model size with quantization and the inference time improvements with MLC by importing the model to TVM.

What lessons have you learned through the project?

One of my biggest challenges working with this project was with Google Colab. Unfortunately, I have had several setbacks from using Google Colab, as it required me to use the CPU for all optimizations since GPU resources were quite limited. This required several steps to take much longer, such as identifying an optimal place to prune the model or calibrating the model before quantization. Moreover, I had issues trying to utilize AutoTVM to optimize the model beyond the manual optimizations that I had originally done, since Google Colab would crash with an unexpected error, which would force me to reset and run the environment once more. If I had the opportunity to complete this assignment again, I would have tested running on other systems so that I would have a backup to utilize in case Google Colab had issues.

Another challenge I had was with regards to the model itself. Unfortunately, there was no provided dataset, so all of my optimizations had to use post-training methodologies, which isn't as effective as training the model after performing either pruning or quantization. I believe that if I had started the assignment earlier, I would have been able to find an appropriate dataset to label and use for training purposes so that I could have used other strategies like Quantization-Aware Training or Fine-Tuning after pruning the model.

Furthermore, I could have been better prepared for setbacks if I had started earlier on the project. One of the biggest challenges that I ended up not being able to solve was importing the pruned and quantized model to TVM. Some of the layers that my optimization strategy such as BatchNorm2D got converted into QuantizedBatchNorm2D, which TVM did not support. Unfortunately, I could not get this part working, so I decided to settle with optimizing the base model to see what optimizations TVM could perform on that. I also faced other issues regarding applying pruning, quantization and TVM strategies, but those I was ultimately able to resolve.

Although I was faced with several setbacks during the course of the project, it made me realize that I should have prepared a bit more for this project and tried to run it on several different systems rather than only on Google Colab. This final project was an encapsulation of everything that we had learned this semester, and I believe that I have learned a lot about the importance of MLC and model optimizations within the ML/AI fields by applying all the knowledge learned in class into this project, which I am truly grateful for.