REAL-TIME AI WITH MACHINE LEARNING SYSTEMS

Course Project

Hasini Chenchala

hchench

Link to GitHub Repository

https://github.com/HasiniChenchala2/LPRNet_Pytorch.git is the GitHub Repo where the code and details to run the code are present.

Link to Google Colab

RTML_hchench_CourseProject.ipynb - Colab is the Colab link to get the results of model optimizations and MLC optimizations

Model Size is calculated using onnx file and Inference speed and Accuracy are shown in colab.

1. **What is the original DNN model you chose to use?**

   The original model used is a **Deep Neural Network designed for License Plate Recognition (LPR) i.e. LPRNet.** It consists of two main parts: the backbone network and global context aggregation.

   Backbone Network:

   - Starts with a convolutional layer that processes the input image and passes through custom small_basic_block layers. These blocks consist of convolutions and ReLU activations.
   - Max pooling and batch normalization are used throughout to reduce dimensionality and stabilize training. Dropout is applied to prevent overfitting.

   Global Context Aggregation:

   Feature maps are extracted from certain layers, pooled, squared, and normalized to compute global context, which is then concatenated.

   Final Classification:

The aggregated global context is passed through a final convolutional layer to generate logits, which are averaged to produce the output predictions.

Results:

Accuracy: 89.9%
Inference Speed: 0.001 seconds/sample
Model Size using onnx: 1.89MB

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

   I used **model Optimizations** which are **Pruning and Quantization and MLC Optimizations** are **Loop Blocking, Parallelization, and Vectorization.**

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

- **Model Optimizations**
  1. **Pruning:**
     Pruning is the process of removing less important weights (parameters) in the network to make the model more efficient.

     By applying L1 unstructured pruning to the convolutional (Conv2d) and linear (Linear) layers, unnecessary weights are removed, leading to a smaller Model Size using onnx. This reduces both memory usage and computation, speeding up inference.

Impact:

Inference Speed: Pruning contributes to a faster model, with an **inference speed of 0.00061 seconds per sample (around 0.6ms/sample)**.

Model Size using onnx: The pruned Model Size using onnx is reduced to **1.5871 MB**, making it more space-efficient.

Accuracy: After pruning, the model maintains a test accuracy of **89.6%**, which indicates that pruning did not significantly degrade performance.

The **impact of pruning on accuracy** follows a non-linear trend, where accuracy initially increases with moderate pruning due to the **removal of**

redundant or less informative weights, leading to a more focused and generalized model. At a **pruning amount of 0.20**, the model achieved its peak accuracy, as this level of pruning struck a balance between simplification and retaining essential features. However, as the pruning amount increased beyond this point, the accuracy gradually declined. This decline occurred because excessive pruning began removing critical weights necessary for capturing complex patterns in the data, thereby reducing the model's predictive performance. This behavior underscores the importance of identifying an optimal pruning level to maximize accuracy while achieving efficiency gains.

2. **Quantization:**

Quantization for deep learning is the process of approximating a neural network that uses floating-point numbers by a neural network of low-bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

- **Dynamic quantization** was applied to the convolutional (Conv2D) and linear (Linear) layers, reducing the precision of the weights to 8-bit integers (qint8).
- The **Conv2D and BatchNorm2D layers were fused** first for better efficiency. This fusion combines them into a single convolutional layer that integrates the BatchNorm2D behavior, optimizing the model by reducing redundant computations.
- **After fusion**, the model was quantized, where the weights were dynamically adjusted at runtime to fit the 8-bit integer representation.

Fusion of Conv2D and BatchNorm2D Layers:

Fusion of Conv2D and BatchNorm2D layers is an optimization technique that merges these two layers into a single convolutional layer. This reduces the computational load during inference, leading to faster execution.

Final Model Evaluation (Quantized and Optimized Model)

After applying pruning, quantization, and layer fusion, the model underwent significant improvements in terms of efficiency and speed.

Results:

Test Accuracy: 90%

Inference Speed: 0.00059 seconds per sample (0.59ms/sample)

Model Size using onnx: 1.58MB (reduced significantly due to quantization)

Impact:

Quantization helped significantly **improve inference speed and reduce the model's memory footprint**, making it **well-suited for deployment on resource-constrained devices.** The test accuracy remained high, demonstrating that quantization can **reduce computational overhead without sacrificing much accuracy.**

The **impact of quantization on accuracy** is generally minimal when implemented effectively, as seen in this model. By **reducing the precision of weights to 8-bit integers** (qint8), quantization preserves most of the essential information while significantly reducing computational and memory requirements. The model maintained a high **test accuracy of 90%**, indicating that the approximation introduced by quantization did not substantially degrade its predictive performance. This can be attributed to the **dynamic quantization approach,** which adjusts weight values at runtime, ensuring that the reduced precision does not adversely affect critical computations.

- **MLC Optimizations:**

MLC (Machine Learning Compilation) optimizations focus on improving the performance and efficiency of machine-learning models by leveraging hardware-specific features and advanced computational techniques. These

optimizations transform high-level operations into optimized low-level code tailored for specific hardware architectures like CPUs, GPUs, and accelerators.

1. **Parallelization and Vectorization:**

Code-Level Implementation of Parallelization

Parallelization in the LPRNet model was achieved by designing the network to take advantage of PyTorch's built-in parallelism. PyTorch automatically dispatches operations to multiple threads or GPU cores when available. Below are some implementation highlights:

Multi-Threaded Operations

PyTorch operations like nn.Conv2d and torch.matmul are inherently parallelized.

For example, in the small_basic_block:

```
self.block = nn.Sequential(

nn.Conv2d(ch_in, ch_out // 4, kernel_size=1), # Parallelized kernel operations

nn.ReLU(),

nn.Conv2d(ch_out // 4, ch_out // 4, kernel_size=(3, 1), padding=(1, 0)),

nn.ReLU(),

nn.Conv2d(ch_out // 4, ch_out // 4, kernel_size=(1, 3), padding=(0, 1)),

nn.ReLU(),

nn.Conv2d(ch_out // 4, ch_out, kernel_size=1),

)
```

Each convolutional layer utilizes GPU cores to parallelize its kernel computations.

Batch-Level Parallelism

During inference, the model processes data in batches to maximize parallel throughput:

```
test_batch_size = 100
```

```
logits = model(input_tensor) # Executes inference in parallel across batch samples
```

Code-Level Implementation of Vectorization

Vectorization ensures that scalar operations are grouped into vectors for efficient execution using SIMD instructions. Examples include:

Vectorized Feature Maps

In the forward pass, pooling and normalization steps were vectorized. Operations like torch.pow, torch.mean, and torch.div process feature maps as tensors rather than looping over individual elements:

```
f_pow = torch.pow(f, 2) # Vectorized element-wise power computation

f_mean = torch.mean(f_pow) # Computes mean over the entire feature map

f = torch.div(f, f_mean) # Element-wise division performed in a vectorized manner
```

Global Context Concatenation

Feature maps from different stages were concatenated efficiently using PyTorch's torch.cat, enabling vectorized processing:

```
x = torch.cat(global_context, 1) # Concatenates along the channel dimension
```

Optimization of Layer Operations

Layer fusion combined Conv2D and BatchNorm2D into a single operation, reducing memory access overhead and redundant computations. PyTorch's torch.jit.trace was used to trace the optimized computation graph for inference.

Results After Optimization

Test Accuracy: 89.9% (No significant drop due to optimizations)

Inference Speed: 0.00062 seconds per sample (0.62ms/sample)

Model Size using onnx: 1.58709 MB (Lightweight and efficient)

Impact:

By applying MLC optimizations like parallelization and vectorization, the LPRNet model achieved faster inference times and a reduced memory footprint without compromising accuracy. These techniques make the model suitable for deployment on resource-constrained devices, offering both performance and efficiency.

The **impact of parallelization and vectorization on accuracy** is negligible, as these optimizations primarily focus on improving computational efficiency without altering the model's architecture or parameters. By leveraging **multi-threaded operations and SIMD instructions**, these techniques speed up computations while maintaining the integrity of the underlying mathematical operations. The model achieved a **test accuracy of 89.9%**, demonstrating that these performance enhancements did not compromise its predictive capabilities. Efficient processing of feature maps through vectorized operations and batch-level parallelism ensures consistent accuracy across varying workloads. This confirms that parallelization and vectorization are effective in enhancing speed while preserving model accuracy.

2. **Loop Blocking:**

   Loop Blocking is a technique that optimizes memory usage and computational efficiency by breaking a large tensor operation into smaller, manageable blocks, leveraging cache locality. Below are the implementation and results for loop blocking in the context of LPRNet:

   Optimized Code for Loop Blocking:

   The implementation modifies the forward method and global context pooling in the LPRNet_loop_blocking class:

   Global Context with Loop Blocking:

   Pooling operations are split into smaller "blocks" of data to enhance cache reuse.

```
# Apply loop blocking during pooling
if i in [0, 1]: # Example: First few feature maps
for h in range(0, f.shape[2], block_size):
for w in range(0, f.shape[3], block_size):
block = f[:, :, h:h + block_size, w:w + block_size]
block = F.avg_pool2d(block, kernel_size=5, stride=5)
else:
# Larger feature maps
for h in range(0, f.shape[2], block_size):
for w in range(0, f.shape[3], block_size):
block = f[:, :, h:h + block_size, w:w + block_size]
block = F.avg_pool2d(block, kernel_size=(4, 10), stride=(4, 2))
```

Block Size Optimization:

The block size is a tunable parameter (e.g., block_size = 32 or smaller for GPU compatibility).

Blocks are processed individually, avoiding overloading the memory.

Vectorized Normalization:

Each block is vectorized for operations like mean calculation and normalization:

```
# Normalize blocks
f_pow = f ** 2
f_mean = torch.mean(f_pow, dim=[1, 2, 3], keepdim=True)
f = f / (f_mean + 1e-8) # Avoid division by zero
```

Concatenation and Further Processing:

Blocked and normalized feature maps are concatenated to form the global_context.

Test Results and Impact for Loop Blocking:

1. Model Size using onnx: 1.5877 MB

The optimized model using loop blocking maintains a size of approximately 1.59 MB, slightly **reduced from the original size. This reduction is due to the efficient use of operations and feature map management during the optimization process. Despite this reduction, there is no compromise in model performance or capacity.**

2. Test Accuracy: 0.894

The loop-blocking implementation achieved a test accuracy of 0.894, which is nearly on par with the original model.

3. Inference Speed: 0.0007203 seconds/sample

The optimized model demonstrated an inference speed of 0.0007203 seconds/sample, a significant improvement over the original. This faster inference is attributed to efficient memory access and enhanced data locality introduced by loop blocking, making it well-suited for low-latency applications.

Loop blocking has a consistent impact on **maintaining model accuracy** while optimizing computational efficiency. By d**ividing large tensor operations into smaller blocks**, it improves cache locality and reduces memory access overhead, ensuring that the core computations remain unaffected. The model achieved a test **accuracy of 89.4%**, showing that the optimization preserved its predictive capabilities. These results highlight how loop blocking efficiently enhances execution speed without compromising the accuracy necessary for reliable predictions.

4. What lessons have you learned through the project?

Working on this project has been an enriching experience, offering insights into both model optimization techniques and broader computational efficiency strategies. Below are the key lessons learned:

1. Deep Understanding of Model Optimization Techniques

Pruning and Quantization:

Pruning enabled the identification and removal of redundant model parameters, **leading to a smaller and more efficient network without a significant loss in accuracy.** It emphasized the importance of balancing model complexity and performance.

Quantization showcased how converting high-precision parameters into lower-bit representations (e.g., from 32-bit to 8-bit) can dramatically **reduce storage requirements and inference time while maintaining satisfactory accuracy levels.**

2. Application of MLC Optimizations

Loop Blocking:

Loop blocking helped enhance data locality by **restructuring the computation to work on smaller data chunks, reducing cache misses, and improving inference speed.** This demonstrated the value of algorithmic restructuring for better hardware utilization.

Parallelization and Vectorization:

Parallelization revealed how workloads could be **efficiently distributed across multiple computational units, reducing execution time significantly.** It highlighted the importance of leveraging modern hardware capabilities like multi-core processors or GPUs.

Vectorization, by leveraging **SIMD (Single Instruction, Multiple Data) operations**, showed how batch operations could be performed simultaneously, leading to **faster execution of repetitive tasks.**

3. Trade-offs Between Efficiency and Accuracy

Optimization techniques often require striking a balance between computational efficiency and model performance. Through pruning, quantization, and other optimizations, I learned how to evaluate trade-offs effectively and decide on acceptable thresholds for performance metrics like accuracy and latency.

4. Profiling and Analysis Are Key

Before applying optimizations, profiling the model to understand its bottlenecks is critical. The project highlighted the **importance of analyzing memory usage, computational overhead, and data flow** to target specific areas for improvement.

5. Importance of Scalability

The optimizations applied in this project not only improved the current model but also set a foundation for scalability. **Techniques like parallelization and vectorization are transferable to larger, more complex models, enabling broader applicability.**

6. Real-World Deployment Considerations

The smaller Model Size using onnx, improved inference speed, and resource efficiency achieved through this project align with deployment requirements for edge devices and real-world applications. This reinforced the need to consider deployment environments during model design and optimization.

7. Collaborative Problem-Solving and Iterative Improvements

The iterative nature of optimizing, testing, and refining models taught me the value of a structured development cycle. Collaboration with tools, frameworks, and methodologies was crucial to success.

Overall, this project deepened my understanding of **computational efficiency, highlighted the importance of trade-offs in machine learning, and strengthened my ability to solve practical challenges in deploying optimized deep learning models.**

5.Overall Effects of the Combination of the Best of the Optimizations:

The combination of **quantization and parallelization & vectorization** led to significant improvements in both model efficiency and performance. Quantization reduced **the precision of the weights to 8-bit integers, significantly lowering memory usage and computational cost**, with the Model Size using onnx reduced to 1.58 MB. Despite this reduction, the model maintained a high test accuracy of 90%, showing that the **loss of precision did not impact predictive performance**. Vectorization further optimized the model **by enabling SIMD instructions to**

process feature maps more efficiently, speeding up operations like pooling and normalization. As a result, the inference time was reduced to 0.00059 seconds per sample, demonstrating the effectiveness of combining quantization with vectorization in enhancing deep learning models for deployment on resource-constrained devices without sacrificing accuracy.

When **pruning was combined with parallelization and vectorization**, the model exhibited a substantial improvement in memory efficiency and computational speed. Pruning reduced the Model Size using onnx to 1.587 MB **by removing less important weights, leading to a more compact model.** Parallelization further sped up processing by leveraging multi-threaded operations and GPU cores, while vectorization **optimized mathematical operations, processing multiple data points simultaneously.** These combined optimizations resulted in a faster inference time of 0.00062 seconds per sample and a test accuracy of 89.9%. By reducing both memory usage and computation, the combination of pruning with parallelization and vectorization made the model more suitable for deployment on devices with limited resources, without compromising accuracy.