Course Project of CSC591/791:

Efficient Vehicle Plate Recognition

Homak Patel (hppatel4)

1. DNN Model

We were given the Pretrained LPRNet PyTorch Model, a potent and pretrained deep neural network (DNN) designed for License Plate Recognition (LPR), from the github link (https://github.com/sirius-ai/LPRNet_Pytorch) for this project. Even in difficult situations like different angles, dim lighting, and image distortions, LPRNet has proven to be highly effective in identifying license plates from photos. The architecture of the model is made to manage the intricacies of license plate identification, attaining high accuracy while preserving effectiveness.

1.1 Model Architecture Overview

The LPRNet model consists of several key components, each playing an essential role in extracting features from the input images and making accurate predictions. The architecture of LPRNet can be broken down into three primary sections: the **backbone**, the **small_basic_block** modules, and the **final prediction layers**. Below is the whole LPRNet model architecture with blocks and layers. I have generated below architecture in my Google Collab file.

```
Model Architecture:

LPRNet(
  (backbone): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool3d(kernel_size=(1, 3, 3), stride=(1, 1, 1), padding=0, dilation=1, ceil_mode=False)
    (4): small_basic_block(
      (block): Sequential(
        (0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
        (1): ReLU()
        (2): Conv2d(32, 32, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0))
        (3): ReLU()
```

```
      (4): Conv2d(32, 32, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1))
      (5): ReLU()
      (6): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (6): ReLU()
  (7): MaxPool3d(kernel_size=(1, 3, 3), stride=(2, 1, 2), padding=0, dilation=1, ceil_mode=False)
  (8): small_basic_block(
    (block): Sequential(
      (0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
      (2): Conv2d(64, 64, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1))
      (5): ReLU()
      (6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (10): ReLU()
  (11): small_basic_block(
    (block): Sequential(
      (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
```

```
      (2): Conv2d(64, 64, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0))
      (3): ReLU()
      (4): Conv2d(64, 64, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1))
      (5): ReLU()
      (6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (13): ReLU()
  (14): MaxPool3d(kernel_size=(1, 3, 3), stride=(4, 1, 2), padding=0, dilation=1, ceil_mode=False)
  (15): Dropout(p=0.5, inplace=False)
  (16): Conv2d(64, 256, kernel_size=(1, 4), stride=(1, 1))
  (17): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (18): ReLU()
  (19): Dropout(p=0.5, inplace=False)
  (20): Conv2d(256, 68, kernel_size=(13, 1), stride=(1, 1))
  (21): BatchNorm2d(68, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (22): ReLU()
 )
 (container): Sequential(
  (0): Conv2d(516, 68, kernel_size=(1, 1), stride=(1, 1))
 )
)
```

1. **Backbone**:
    - The backbone of LPRNet starts with a convolutional layer that processes the raw input image. This layer extracts low-level features such as edges and textures. It's followed by

batch normalization, which helps stabilize training by normalizing the activations and reducing internal covariate shift. The ReLU activation function is then applied to introduce non-linearity, allowing the network to learn more complex patterns.
    - After the convolutional operation, a max-pooling layer is used to reduce the spatial dimensions of the feature maps, keeping only the most critical information. This pooling layer helps the model become more invariant to small translations and shifts in the input images, which is essential for LPR tasks where plates may not always be centered or aligned in the image.
2. **Small Basic Blocks**:
    - One of the unique aspects of LPRNet is the use of small_basic_block modules. These blocks consist of multiple 1x1 and 3x1 convolutional layers that help the network learn hierarchical feature representations. These convolutions are designed to capture both fine-grained details and more abstract features of the vehicle plates.
    - Each block contains multiple layers of ReLU activations to maintain non-linearity and ensure the model can learn more complex relationships. Batch normalization is applied after each convolution to stabilize the network's training process.
    - These blocks progressively reduce the spatial dimensions of the feature maps while increasing the number of channels (filters) to capture increasingly abstract and complex features. As a result, the model becomes more adept at recognizing the intricate details of license plates.
3. **Pooling, Dropout, and Convolutional Refinement**:
    - As the model progresses, additional max-pooling layers are employed to further downsample the feature maps. This operation helps the model focus on the most important features and reduces the computational load for subsequent layers.
    - Dropout layers are introduced to combat overfitting. During training, these layers randomly drop a fraction of the activations, forcing the model to generalize better and not rely too heavily on any single feature. This helps improve the robustness of the model, making it less sensitive to noise and variations in the input data.
    - The model also incorporates convolutional layers at the later stages to refine the extracted features. These layers aim to produce a more compact and discriminative representation of the vehicle plate, which will later be used for the final character prediction.
4. **Final Layers and Prediction**:
    - The final layers consist of a series of convolutional and batch normalization layers, which further refine the feature maps produced by earlier layers. These refined features are passed through the last convolutional layer, which outputs a prediction of the license plate characters.
    - The final output layer is designed to predict the sequence of characters on the license plate. This output is typically a sequence of character probabilities that are decoded into the recognized license plate number. The model is trained end-to-end to maximize the likelihood of the correct characters, even if the plate is distorted, rotated, or partially occluded.

1.2 Strengths of LPRNet

LPRNet's architecture has several strengths that make it particularly well-suited for vehicle plate recognition:

- **End-to-End Learning**: Unlike traditional methods that rely on handcrafted features, LPRNet is trained end-to-end, meaning the entire model is optimized for the final task of license plate recognition.
- **Robustness to Variations**: The model's use of multiple convolutional layers, batch normalization, and pooling operations helps it become robust to variations in input images, such as different angles, lighting conditions, and plate designs.
- **Efficient Feature Extraction**: The small_basic_block modules allow for efficient and hierarchical feature extraction. These blocks help the model learn both low-level and high-level features, which are crucial for accurately recognizing vehicle plates.

1.3 Challenges in LPRNet

Despite its strengths, there are some challenges associated with LPRNet:

- **Model Size**: LPRNet is a relatively large model, which can make it challenging to deploy on devices with limited computational resources, such as mobile phones or edge devices.
- **Inference Speed**: While the model performs well in terms of accuracy, its inference speed may not meet the real-time requirements for certain applications, such as on-the-fly license plate recognition in traffic monitoring systems.
- **Generalization**: Although LPRNet is highly accurate, its performance can still be impacted by factors such as extreme distortions, unusual plate designs, or very low-resolution images.

2. Model Optimizations

To enhance the performance of the Pre-trained LPRNet model, I employed two model optimization techniques: fine-grained pruning and dynamic quantization. Each of these techniques was chosen based on their effectiveness in reducing model size and inference time without significantly compromising the model's accuracy.

2.1 Fine-Grained Pruning (Model Optimization - 1)

Why I Chose Fine-Grained Pruning:

Pruning is a well-established technique to reduce the complexity of neural networks by removing unnecessary parameters. Out of the various pruning techniques available, I decided to use fine-grained pruning for its ability to target specific individual weights within a network, rather than removing entire neurons or channels. This approach allows me to maintain the network's structure while still achieving significant reductions in model size and computation.

Intuition Behind Fine-Grained Pruning:

The idea behind fine-grained pruning is to selectively remove weights with lower magnitudes, as these weights typically have less impact on the model's performance. By applying pruning at the level of individual weights rather than entire layers or neurons, we can more effectively reduce the number of parameters while retaining the model's representational power.

I implemented fine-grained pruning using a **sparsity factor** for each layer, where each layer can have its own specific sparsity, meaning some layers might retain more parameters than others based on their importance to the model. This allowed for more flexibility compared to traditional pruning methods where the sparsity is uniform across the network.

Results from Fine-Grained Pruning:

After applying fine-grained pruning to the LPRNet model, I achieved the following results:

- **Number of Non-Zero Parameters:**
    - Before pruning: 446,975 non-zero parameters.
    - After pruning: 225,301 non-zero parameters.

This demonstrates a reduction of about **50%** in the number of non-zero parameters, resulting in a smaller model size.

- **Inference Time:**
    - The inference time of the original model was **195.74 ms**, and after pruning, it was reduced to **109.12 ms**, demonstrating a performance improvement in terms of speed.
- **Model Accuracy:**

    - The pruned model achieved an accuracy of **0.9000**, which is comparable to the original model's performance, indicating that fine-grained pruning did not significantly degrade the model's ability to make predictions.

The significant reduction in model size and inference time, combined with minimal accuracy loss, demonstrates the effectiveness of fine-grained pruning as a model optimization technique.

2.2 Dynamic Quantization (Model Optimization - 2)

Why I Chose Dynamic Quantization:

Out of the many model optimization techniques available, I chose dynamic quantization to further optimize the pruned model. Quantization involves reducing the precision of the model's weights and activations from 32-bit floating-point values to lower bit-width representations (e.g., 8-bit integers). This technique helps reduce both the model size and inference time, particularly on hardware that supports efficient operations on lower-precision values.

I chose **dynamic quantization** over other forms of quantization (e.g., static or post-training quantization) because it provides a good balance between speedup and minimal degradation in accuracy. Dynamic quantization adjusts the precision of weights and activations on the fly, making it well-suited for deployment scenarios where computational efficiency is critical.

Intuition Behind Applying Quantization to the Pruned Model:

The intuition behind applying **dynamic quantization** to the pruned model rather than the original model lies in the fact that pruning already reduces the number of parameters in the network. By quantizing the pruned model, we can take advantage of the reduced size and further enhance the model's efficiency. The pruned model has fewer non-zero weights, which makes it more amenable to quantization, as the savings in memory and computation are more significant on smaller models.

Applying quantization to a pruned model makes sense because the pruned model is already more compact and sparse, so quantizing it further can provide a better trade-off between model size, speed, and accuracy. In contrast, applying quantization directly to the original model, which has more parameters, might not yield the same level of optimization, as the benefits of quantization could be overshadowed by the larger size of the model.

Results from Dynamic Quantization:

After applying dynamic quantization to the pruned model, the following results were observed:

- **Model Size:**
    - The Onnx model size after quantization was **1747.25 KB** (about 1.7 MB), which similar to both the original and pruned models.
- **Inference Time:**

    - The inference time of the quantized model was **107.98 ms**, which is slightly faster than the pruned model (**109.12 ms**) but still offers a modest speedup compared to the original model (**195.74 ms**).
- **Accuracy:**
    - The quantized pruned model achieved an accuracy of **0.8260**, which is a slight decrease compared to the pruned model (**0.9000**). This shows a trade-off between model efficiency and accuracy, but the speedup and reduction in model size are typically worth this small sacrifice in performance, especially when deploying models to edge devices or real-time applications.
- **Speedup:**
    - The speedup after quantization was calculated to be **0.98x**, meaning the quantized model runs slightly faster than the pruned model, with a near-parity in terms of performance.

2.3 Conclusion

The decision to use **fine-grained pruning** and **dynamic quantization** was driven by the goal of reducing both the model size and inference time while maintaining high accuracy.

- **Fine-grained pruning** was chosen because it enables a more targeted and efficient reduction in model size by removing less important weights, which allows the model to retain its essential features while reducing computation.
- **Dynamic quantization** was chosen because it reduces the bit-width of weights and activations dynamically, optimizing the pruned model's memory footprint and speed while minimizing the loss in accuracy.

Both techniques, when applied together, significantly enhanced the model's efficiency without substantial losses in performance. The optimization results show that the combination of pruning and quantization not only reduces the model size (in terms of non zero parameters not onnx model size) and inference time but also ensures that the model is well-suited for deployment on resource-constrained devices.

3. MLC Optimizations

For the MLC optimization of the model, I initially intended to apply manual optimization to the original LPRNet model. However, given the complexity of the LPRNet architecture, I decided to forgo manual optimization. The LPRNet architecture consists of several intricate layers and operations, making it extremely challenging to manually optimize each component effectively. Instead, I opted to leverage **AutoTVM**, an automated tool for optimizing deep learning models, which significantly simplifies the tuning process and ensures that the model performs efficiently on the target hardware.

3.1 MLC Optimization Approach with AutoTVM

1. **Model Preparation**:
    - I chose to apply the MLC optimization to the pruned and quantized model rather than the original LPRNet model. The reason for this decision was that the pruned and quantized version of the model was already providing strong accuracy and performance. Applying MLC optimization to this optimized version ensures that the entire cycle of model optimization, pruning, quantization, and MLC optimization is adhered to, allowing the model to benefit from both reduced size and improved performance.
    - I used the quantized model because it already demonstrated favorable results in terms of accuracy, and optimizing it further through MLC techniques could yield additional benefits without compromising its effectiveness.
2. **Why I Chose AutoTVM Over Manual Optimization**:
    - Given the complexity of the LPRNet architecture, manually tuning its various operations was not only difficult but also time-consuming. The manual process would have involved adjusting numerous parameters for each layer and operation—an impractical approach given the size and complexity of the model.
    - I employed **AutoTVM** to streamline the optimization process. AutoTVM is an advanced tool within the **TVM** framework that automates the process of searching for the best configuration of operations like convolutions, matrix multiplications, and other compute-heavy operations. This tool uses machine learning-based algorithms to intelligently find the most efficient settings, greatly reducing the time and effort required for manual tuning.
    - By using AutoTVM, I was able to ensure that the model's operations were fine-tuned to run optimally on the target hardware, whether on CPU or GPU, without having to manually write complex optimizations.
3. **AutoTVM Tuning Process**:
    - **Exporting the Model to ONNX**: I first exported the pruned and quantized model to **ONNX** (Open Neural Network Exchange) format. ONNX is a widely supported model format that allows for easy integration with various deep learning frameworks, including TVM.

- **Loading the Model into TVM**: After exporting the model to ONNX, I loaded it into TVM. I used the relay.frontend.from_onnx function to convert the ONNX model into a TVM computation graph.
- **AutoTVM Task Extraction and Tuning**: I employed AutoTVM to extract tuning tasks from the model's computation graph. These tasks included optimizing key operations like convolutions and other performance-critical layers. AutoTVM's **XGBTuner** was used to automatically search for the best configurations, adjusting parameters based on hardware specifics. I ran a set of trials (with num_trials=50) to explore different parameter combinations, optimizing the model for the hardware target.
- **Target Hardware**: I employed **GPU** as the primary target for acceleration, but also set up the optimization for CPU as a fallback. This flexibility allowed the model to be optimized for different environments, ensuring broader deployment options.
- **Optimizing the Model**: Once the tuning was complete, I applied the best configurations found by AutoTVM to build the final model. The model was then compiled with these optimizations, ensuring the best performance for inference.

4. **Inference Time and Model Performance**:
    - After applying AutoTVM tuning, I measured the inference time of the optimized model. The average inference time for the MLC-optimized, AutoTuned model was approximately **323.60 ms**, indicating a substantial improvement in speed compared to the baseline model.
    - The AutoTVM tuning helped significantly reduce the computational cost of the operations, leading to faster inference times without compromising the accuracy of the model.
5. **Evaluation of the Optimized Model**:
    - To evaluate the effectiveness of the optimized model, I performed inference on a dataset using a **greedy decoding** approach. This method is typically used in sequence-to-sequence models to predict the most likely sequence based on the output probabilities.
    - During the evaluation, the model achieved an **accuracy of 82.40%**, which was on par with expectations considering the applied optimizations.
    - The speed at which the model processed images was also impressive, with the average time per image being **0.0345 seconds**. This demonstrates that the MLC optimization, combined with AutoTVM tuning, not only maintained high accuracy but also significantly improved the model's efficiency.

3.2 Why AutoTVM Was the Ideal Solution

Given the challenges posed by the LPRNet architecture, I found AutoTVM to be the ideal solution for optimization. It allowed me to avoid the complexity of manual tuning, offering a more efficient and scalable approach to model optimization. The automated nature of AutoTVM not only saved valuable time but also ensured that the model was optimized in a way that would be very difficult to achieve manually, especially given the intricate nature of the operations involved.

In conclusion, by employing AutoTVM for MLC optimization, I was able to effectively optimize the pruned and quantized LPRNet model. This process yielded both improved inference speed and maintained high accuracy. Using AutoTVM ensured that the model was tailored for optimal performance on the target hardware, making it the perfect choice for optimizing complex models like LPRNet.

4. Final Results Summary

The optimization of the LPRNet model was carried out through various approaches, including pruning, quantization, and MLC autotuning. Below is a detailed summary of the outcomes based on test accuracy, inference time, and model size.

4.1 Quality (Test Accuracy)

The original model achieved a test accuracy of **0.898**, showcasing strong baseline performance. After pruning, the accuracy slightly improved to **0.9**, reflecting the effectiveness of pruning in simplifying the model while preserving or even enhancing accuracy. However, quantizing the pruned model resulted in a minor accuracy drop to **0.826**. Similarly, applying MLC optimization slightly reduced the accuracy further to **0.824**, demonstrating the trade-offs between computational efficiency and precision.

4. 2 Inference Time

The inference time showed significant improvements after pruning and quantization. The original model's inference time was **195.74 ms**, which reduced drastically to **109.12 ms** after pruning. Quantization further improved this to **107.98 ms**, emphasizing the computational benefits of these optimizations. However, MLC optimization increased the inference time to **323.59 ms**, indicating that while it may enhance certain aspects, it introduces additional computational overhead in this case.

4.3 Model Size

Though the ONNX model size remained constant at **1747.25 KB** across all stages, a critical observation was the reduction in the number of non-zero parameters after pruning. This indicates a significant improvement in model sparsity and efficiency. The reduced number of non-zero parameters contributed to faster inference times, especially in the pruned and quantized versions, by reducing computational requirements without increasing storage size.

4.4 Observations and conclusions

The results highlight key trade-offs in model optimization. Pruning proved highly effective, reducing the number of non-zero parameters, improving inference time, and even slightly increasing accuracy. Quantization added further benefits in inference speed but introduced a minor accuracy drop. In contrast, MLC optimization provided insights into automatic tuning but resulted in a considerable increase in inference time, making it less practical for latency-sensitive applications.

It is also important to note that the metrics reported here were obtained by running the code in Google Colab. Due to the variability in runtime environments and hardware configurations in Colab, the reported metrics (accuracy, inference time, etc.) may vary each time the notebook is executed. Therefore, the metrics provided in this report should not be considered exact or definitive, but rather indicative of trends and relative performance changes.

Through this project, I employed advanced optimization techniques, including pruning and quantization, to significantly enhance the computational performance of LPRNet while maintaining acceptable accuracy levels. These efforts demonstrate the importance of selecting and tailoring optimization strategies to achieve a balance between model efficiency and task-specific performance goals.

5. Lessons Learned

This project provided valuable insights into optimizing complex machine learning models and highlighted several important lessons:

1. **Balancing Trade-Offs Between Accuracy and Efficiency**
   I learned that optimizing machine learning models often involves trade-offs between accuracy, inference speed, and model size. For instance, pruning and quantization improved inference time significantly but caused minor reductions in accuracy. This reinforced the importance of prioritizing task-specific requirements when selecting optimization techniques.
2. **Model Sparsity and Computational Efficiency**
   By employing pruning, I observed that reducing the number of non-zero parameters can have a direct and substantial impact on inference speed, even when the model's size (e.g., ONNX file) does not visibly decrease. This highlighted the critical role of sparsity in improving computational efficiency.
3. **Challenges of Complex Architectures**
   Working with the LPRNet architecture demonstrated the complexities of optimizing intricate models. I realized that while manual optimization can provide more control, it is impractical for such architectures. Automated approaches like pruning and quantization are better suited to streamline the process while still achieving significant gains.
4. **Hardware and Runtime Variability**
   Running the experiments in Google Colab revealed the variability in metrics such as accuracy and inference time due to changes in runtime environments. This taught me to approach reported metrics with caution, as they can differ when experiments are run in different hardware setups or sessions.
5. **Optimization Automation**
   Leveraging MLC autotuning gave me insights into automated optimization strategies. While it introduced additional computational overhead, it also highlighted the benefits of automation in finding optimal configurations for deployment. This experience underscored the value of balancing automation with manual fine-tuning.
6. **Flexibility and Adaptability**
   Throughout this project, I employed various optimization strategies, adapting to the challenges presented by the model and the tools used. This experience reinforced the importance of flexibility and iterative problem-solving when working with real-world machine learning systems.
7. **Real-World Applications of Optimization**
   The improvements in inference time achieved through pruning and quantization emphasized their practical significance for deploying models in resource-constrained environments or for applications requiring real-time processing. I learned how these techniques can make models more efficient and usable in such scenarios.

This project provided hands-on experience with advanced optimization techniques and a deeper understanding of the trade-offs involved, preparing me to address similar challenges in future machine learning tasks.