Assignment 1\_2

CSCE 790 – Edge and Neuromorphic Computing

Mark Shperkin

**Overview:**

In this assignment, we were tasked with designing a convolutional neural network inspired by the VGG architecture for classifying 32x32 images into 10 different categories. The dataset was divided into training, validation, and test sets. Additionally, we participated in a competition where the final score was determined by dividing the average accuracy by the average latency, making it crucial to optimize both accuracy and inference speed.

**Solution:**

**Data Preprocessing & Augmentation**

To improve the model’s generalization ability, data augmentation was applied to the training set using a variety of transformation techniques. The augmentation process included:

* Translation (Shifting the image up to +-4 pixels)
* Rotation (Random rotation up to +-30 degrees)
* Flipping (Horizontal flipping with a 50% probability)
* Minor Transformations (Slight rotation up to +=15 degrees combined with small translations)
* Color Jittering (Random brightness, contrast, and saturation changes)
* Perspective Transformations (Random warping to simulate different viewpoints)

Each augmentation technique added 7,000 new samples, resulting in an additional 42000 images, significantly increasing the dataset size. This augmentation strategy was designed to increase generalization and prevent overfitting.

**Model Architecture**

I designed a simplified VGG-inspired model, considering that the original VGG architecture would be too complex for this relatively small-scale classification task. To achieve a balance between accuracy and inference speed, I reduced the number of convolutional layers and neurons while still maintaining performance.

The final model consists of:

**Convolutional Layers**

* Conv2d(3, 32, kernel\_size=3, padding=1) -> ReLU -> MaxPool(2x2)
* Conv2d(32, 64, kernel\_size=3, padding=1) -> ReLU -> MaxPool(2x2)
* Conv2d(64, 128, kernel\_size=3, padding=1) -> ReLU -> MaxPool(2x2)

**Fully Connected Layers**

* Linear(2048, 512) -> ReLU -> Dropout(0.3)
* Linear(512, 128) -> ReLU -> Dropout(0.3)
* Linear(128, 10)

This lighter version of VGG reduces computational complexity while maintaining high classification accuracy.

**Training Strategy**

To train the model efficiently, I used:

* Batch Size: 256
* Epochs: 80
* Learning Rate: 0.0001
* Loss Function: CrossEntropyLoss
* Optimizer: Adam

**Model Checkpointing**

To ensure that the best model is saved during training, I implemented model checkpointing. Where the model’s performance was monitored using the validation accuracy. Each time a new best validation accuracy was achieved, the model was saved. This prevented the model from being overwritten by worse-performing iterations and allowed training to continue for a longer duration.

**Inference Optimization & Evaluation**

Since the competition score was accuracy divided by inference time, I optimized the model’s inference time with the following techniques:

* GPU Warm-Up: Running a dummy input before inference to stabilize GPU performance. (solved a problem where the first inference time was significantly high)
* Using perf\_counter() Instead of time(): The time() function resulted in some inference times being recorded as 0.0 ms, likely due to lower resolution and system-dependent precision. Instead, I used time.perf\_counter(), which provides higher precision and accuracy for measuring execution time.

After evaluating the model, I generated the final **score.csv** submission file.

**Challenges & Future Work**

Initially, I intended to search for an optimal architecture automatically but faced time constraints due to environment setup issues, including getting CUDA and Kaggle’s GPU to function correctly. This limited the amount of experimentation I could perform.

**Conclusion**

This project provided hands-on experience in designing a compact yet effective CNN, implementing data augmentation, and optimizing inference efficiency. By balancing model complexity and speed, I achieved competitive results, improving generalization through augmentation while ensuring efficient inference. After figuring out how to use the GPU Kaggle provided, I realized that running the testing on my own local GPU produced better results. This led me to suspect that my local testing environment might have given me an unfair edge in the competition, as inference speeds could vary depending on hardware. Additionally, I believe that the data split provided in Kaggle, and the dataset generated by get\_dataset.py may not have been split identically. If a model was trained using one of these splits and then tested using the other, it could result in the model being evaluated on partially seen data, potentially providing another unfair advantage.