RBE 474X
Project 1

Colin Balfour
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

Khang Luu
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

Thinh Nguyen
Worcester Polytechnic Institute
Worcester, Massachussetts 01609

I. Part 1

A. Implementation

Part 1 implemented custom layers for a Multi-Layer Perceptron (MLP), cuda-optimized with PyTorch. The layers implemented are: Linear, ReLU, and Softmax. The forward and backward passes were implemented for each layer. The forward pass computes the output of the layer given the original input (or previous layer's output), and the backward pass computes the gradient of the loss with respect to the input, weights, and bias. The custom implementation was then validated against PyTorch's built-in layers.

B. Results

All implementation yeilded the same results as with PyTorch's built-in layers. The output is in the jupyter notebook.

II. Part 2

A. Implementation

In Part 2, we trained the MLP model using the custom layers implemented in Part 1. The model was trained on the CIFAR-10 dataset with 40 epochs. We used an Adam Optimizer with a learning rate of 1e-4 and a batch size of 32. The custom layers from Part 1 were used to build the MLP model, and were optimized for batching and parallel processing using CUDA. The training loss curve and confusion matrices for the validation set at the first and last epochs are shown in figures 1 - 3.

B. Results

Overall, the network was well-optimized, and only took about 5-10 seconds per epoch on a modern GPU. The training loss curve shows a steady decrease in loss over the 40 epochs, indicating that the model is learning, and achieved a final accuracy of 43.68%.

III. Part 3

A. Implementation

In Part 3, we compared the performance of the MLP implemented in Part 1 with a more modern CNN architechture, using PyTorch's inbuilt layers for simplicity. The model was trained on the same CIFAR-10 dataset with 40 epochs, also using Adam Optimizer with a learning rate of 1e-4 and a batch size of 32. The training loss curve and confusion matrices for the validation set at the first and last epochs are shown in figures 4 - 6.

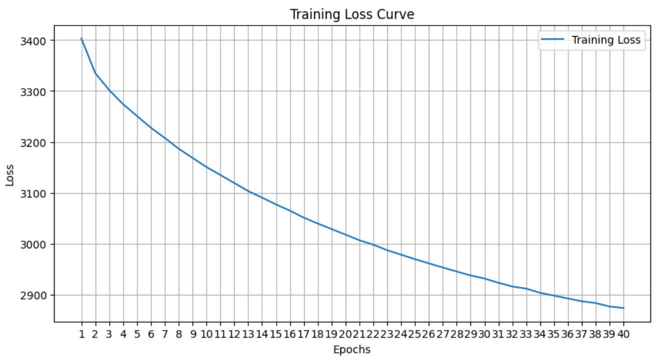

Fig. 1. Training loss curve for part 2

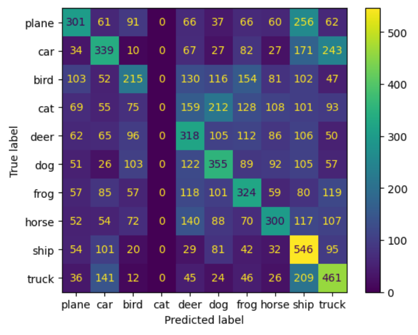

Fig. 2. Confusion matrix for part 2, epoch 1

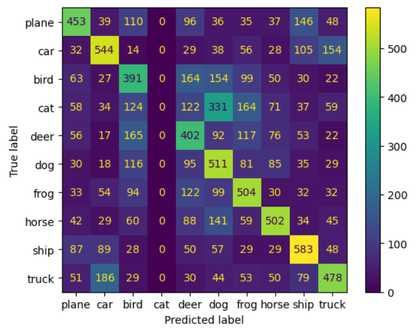

Fig. 3. Confusion matrix for part 2, epoch 39

B. Results

The CNN model performed significantly better than the MLP, achieving an accuracy of 56.79% compared to the MLP's 43.68%. This is due to the CNN's ability to learn spatial features in the images, due to the convolutional layers, where the network learns a kernel filter to apply to the image to extract such features useful in classifying images. The MLP, on the other hand, does not have such structure, and must learn the features from the raw pixel values, which is much more difficult. It also was faster to train, with the CNN taking about 7m 28s to train, compared to the MLP's 8m 36s. This is likely due to efficiencies in PyTorch's optimization compared to our custom implementation. CNNs can be more efficient than MLPs, due to the convolutional layers reducing dimentionality (and therefore the number of parameters), but in this case the CNN also had more layers/parameters, so it's unlikely the network architecture contributed to its efficiency

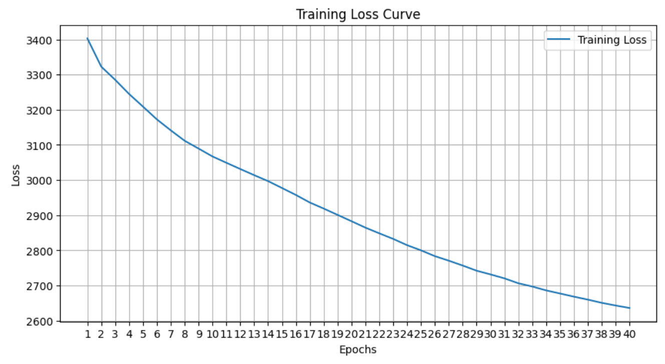

Fig. 4. Training loss curve for part 3

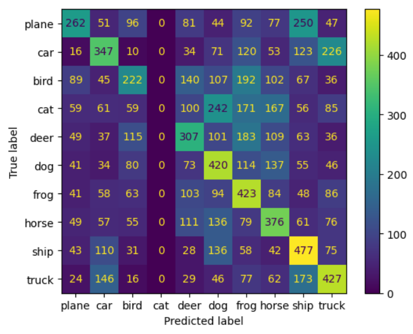

Fig. 5. Confusion matrix for part 3, epoch 1

IV. Part 4

A. Implementation

Part 4 implemented custom layers for a Convolutional Nerual Network (CNN), again cuda-optimized with PyTorch. Like in Part 1, forward and backward passes were implemented for a Convolutional (Conv2d) layer. This included an implementation of a 2d cross-correlation method, which is the main operation performed by the Conv2d layer. The forward

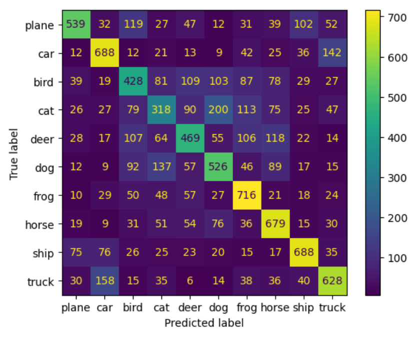

Fig. 6. Confusion matrix for part 3, epoch 39

and backward propogations were implemented for any number of input channels, output channels, kernel size, stride, and batch size. The custom implementation was then validated against PyTorch's built-in layers.

B. Results

All implementation yeilded the same results as with PyTorch's built-in layers. The output is in the jupyter notebook.

V. Part 5

Part 5 was not attempted, despite the custom CNN implementation, due to time constraints, both in the code and in the humans writing it.