CS4803 EML Lab 0

Saad Amin

September 2025

0.1 Lab 2

I utilized a loop to iterate through various hyperparameter configurations.

1. It appears that for MNSIT, schedulers don't change the final accuracy too much. Optimizers also don't affect the final output, with the exception of SGD which has much lower performance (as expected). Learning rate doesn't affect accuracy either, but a very high learning rate like 0.1 makes the accuracy go down. In experiments with $LR = 1.0$ which was suggested in the Notebook's original comments, I found that the network did not converge at all. From my experience tuning model architecture in other contexts, as long as the architecture isn't very unreasonable (like very shallow, or too deep without residual connections) then the accuracy will remain high. In other words, any small-to-medium change to a high accuracy architecture usually does not change accuracy too much.

2. I conducted CIFAR-10 testing with much fewer epochs, since this assignment is due in a few hours and even A100s are too slow to train a ResNet18 fast. Thus the data collected here is not 100% representative of real scenarios. I found that the optimal learning rate is 0.001, which is not too large or small. SGD, as expected, is a terrible optimizer, and RMSProp lags behind Adam and AdamW which are tied for first place. Consine annealing LR had a surprisingly larger accuracy than the other schedulers.

Conclusion: always use Adam with a learning rate of 0.001 or 0.0001.

0.2 Lab 3

1. I like to implement linear bottlenecks with `nn.Sequential`:

```
    self.conv = nn.Sequential(
        nn.Conv2d(in_planes, hidden_dim, 1, bias=False),
        nn.BatchNorm2d(hidden_dim),
        nn.SiLU(),

        nn.Conv2d(hidden_dim, hidden_dim, 3, stride=
            stride, padding=1,
```

```
                groups=hidden_dim, bias=False),
        nn.BatchNorm2d(hidden_dim),
        nn.SiLU(),

        nn.Conv2d(hidden_dim, out_planes, 1, bias=False)
            ,
        nn.BatchNorm2d(out_planes)
    )
```

2. I was able to get down to $\sim$ 130M MACs without accuracy reduction. I was able to get down to $\sim$ 30M MACs with a $\sim$ 10% accuracy drop.

3. My real world measurements actually indicated that despite the MACs reduction, the "more efficient" models were far slower than expected, and `torch.compile` caused the models to become even slower. I have a few explanations:

   (a) I noticed from small scale experiments (not included in Notebook results) that the more efficient models marginally beat the original ResNet when in FP32, as opposed to FP16 which I trained and tested the models in. Furthermore, the original ResNet model performance stayed very similar regardless of precision while the FP16 performance of the MobileNets was way slower than FP32. I don't have a concrete explanation why, but I think `torch.autocast` or something CPU side might be doing things inefficiently.

   (b) Furthermore, depthwise convolutions are strongly memory bound. I tested on an A100, which has strong compute capabilities but is often bottlenecked by memory accesses. In that case, any efficient form of convolution like point-wise or depth-wise has the same time cost as a dense convolution.

   (c) Although MobileNets have less MACs, they are more complex from a computational graph perspective. They consist of more sequential operations, which is harder to parallelize on the GPU efficiently.

   (d) Although I didn't use `torch.compile` for all experiments, I think there may have been some form background optimization, where cuDNN or some other library that PyTorch relies on was optimizing calls on the fly. They may have been swapping out ResNet kernels for highly optimized hand-written kernels, whereas these kernels might not have existed for MobileNet.

   (e) I think the `torch.compile` overhead probably came from the fact that we were CPU bound. Profiling in NSight Systems showed that our GPU was idle for 90% of the time in larger batch sizes, waiting for data to be transfer to the device and compiled functions to execute.