Bryant Har (bjh254), Eric Sun (eds228)

Summary

Throughout the programming assignment, I observed how using optimization algorithms, momentum schemes, and normalization can enhance standard SGD. Using Adam, momentum, and batch normalization can greatly accelerate the convergence and stability of SGD. Using these techniques, the loss plot converges/decreases faster and in a much more stable manner. Then, I used grid search and random search to explore the space of hyperparameter sets to find more optimal hyperparameters for the model (step size, beta, batch size, etc.). I saw how these techniques can take extremely long to execute but can yield better and more rigorous results than simple hand-tuning or choosing a hyperparameters set from folklore. This way, I was able to get higher accuracy than what was achieved in the previous sections. Finally, I observed how convolutional neural networks can be used to reduce the number of parameters in a model. Despite how much smaller the network was, the convolutional neural net performed even better and achieved higher accuracy benchmarks than the models from the first part.

Section 1

The labeled plots and wall clock times (in seconds at the top of each image) for each of the four algorithms are shown at the end of the report. All algorithms achieved test accuracies around or exceeding 0.98. Furthermore, each algorithm took between 200-240 seconds to complete, running on Google Colab's server CPU.

**1.6: Report the wall-clock times for training the algorithm. How does the performance of the different algorithms compare? Please explain briefly.

The wall times in seconds are 201.5 for SGD, 216.6 for SGD with momentum, 236.4 with Adam, and 210.8 with batch normalization. Clearly, the standard implementation is fastest, followed by batch normalization, standard momentum, and Adam.

All converged around 0.98 test accuracy, though standard SGD converged the slowest. By the last epoch, its loss graph still was not stable, and the plot was not very flat at the end. By contrast, momentum and Adam had similar accuracies but had much steeper initial curves, indicating faster convergence speeds. Batch normalization converged slightly less fast than momentum but had a much smoother accuracy curve. Batch normalization greatly enhanced the stability of the convergence.

The final test accuracies were all comparable, and the improved algorithms tended to converge much faster and perform better than pure SGD.

**1.7: How does the "approximate" training loss gotten "for free" from the minibatches during training compare to the "real" end-of-epoch training loss? Is this easier-to-compute proxy a good metric for evaluating convergence? Explain briefly.

Clearly from the graphs, they serve as a good analogue and follow the end of epoch training loss relatively well. However, convergence is more difficult to discern based solely on the approximate training loss graph, and it may appear to reach convergence at a different point than what the full end of epoch loss indicates.

Overall, it’s a moderately good proxy to evaluate convergence, though caution should be taken since it does not follow the end of epoch loss plot exactly.Nevertheless, this approximate loss metric is better than nothing, especially since it is collected for free.

Section 2

2.1.

The highest validation accuracy was achieved with a learning rate of 0.1. This is the same hyperparameter used in Part 1. Here is the validation accuracy and loss observed for each value of alpha:

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/lazy.py:180: Us
  warnings.warn('Lazy modules are a new feature under heavy development
100%|██████████| 10/10 [04:45<00:00, 28.54s/it]
Learning Rate: 1.0
Val Loss: 2.3252865505218505, Val Accuracy: 0.10279999673366547
Moving on to next one...
100%|██████████| 10/10 [04:10<00:00, 25.00s/it]
Learning Rate: 0.3
Val Loss: 0.14741767410472675, Val Accuracy: 0.9682999849319458
Moving on to next one...
100%|██████████| 10/10 [03:32<00:00, 21.21s/it]
Learning Rate: 0.1
Val Loss: 0.06442476138630354, Val Accuracy: 0.9836999773979187
Moving on to next one...
100%|██████████| 10/10 [03:17<00:00, 19.76s/it]
Learning Rate: 0.03
Val Loss: 0.0655407006772748, Val Accuracy: 0.9818000197410583
Moving on to next one...
100%|██████████| 10/10 [03:16<00:00, 19.68s/it]
Learning Rate: 0.01
Val Loss: 0.06534248367184774, Val Accuracy: 0.9801999926567078
Moving on to next one...
100%|██████████| 10/10 [03:13<00:00, 19.34s/it]
Learning Rate: 0.003
Val Loss: 0.1296159452246502, Val Accuracy: 0.961899995803833
Moving on to next one...
100%|██████████| 10/10 [03:07<00:00, 18.70s/it]
Learning Rate: 0.001
Val Loss: 0.2455887557566166, Val Accuracy: 0.9305999875068665
Moving on to next one...
```

2.2

We chose to consider the hyperparameters for beta {0.95, 0.9, 0.85}, the number of nodes in the first layer {2048, 1024, 512} (width), and the number of nodes in the second layer {256, 128, 64} (narrow). There are 27 trials in this grid, and we chose similar values for beta because 0.9 seemed to work well. For the layers, we used multiples of 2 for hardware affinities.

Our best validation accuracy and loss was 98.4% and 0.0655 with beta=0.95, width=512, narrow=128.

2.3

We used a uniform distribution across the same range as the grid in Part 2.2. While the range is technically identical, smaller values are not as well-represented. A further extension could be to use a distribution that biases toward smaller values involving a log-norm distribution.

We found that the following hyperparameters worked best: learning rate=0.023, beta=0.912, wide_dim=843, narrow_dim=323. This resulted in a validation loss of 0.0436 and a validation accuracy of 98.7%. The test loss and accuracy were 0.0388 and 98.9% respectively.

2.4

For our experiments, random search had a marginally better accuracy compared to grid search to the point where the results are comparable. However, grid search took 27 trials whereas random search took 10 trials, so there is a preference for random search for the lower computational complexity.

Section 3

The plot and wall time for the part three model is shown below. It took 820 seconds with Google Colab gpus using the provided architecture.

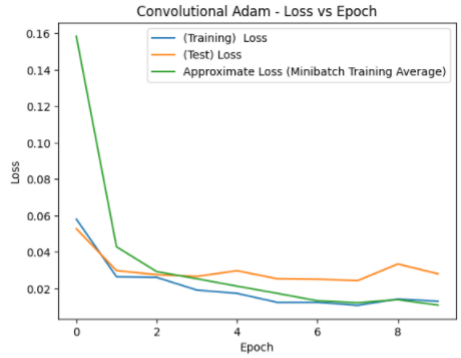
**3.4 : How does the number of parameters used in this network compare to the number of parameters used in the MLPs you studied in Part 1?

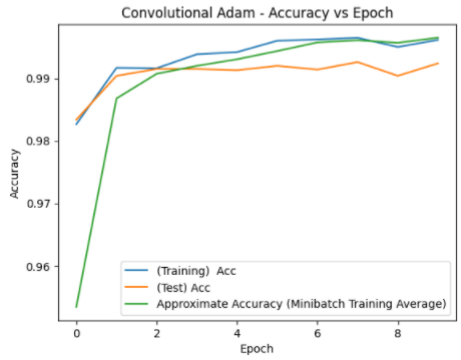
The convolutional neural network uses far fewer parameters than in the MLPs of the first part. Most of our layers only take in 32 or 16 inputs compared to the thousands in the fully connected MLP. CNNs are able to sparsify the model by leveraging small learnable filters/kernels to capture local patterns in the image. These convolution filters can transform the image by blurring, embossing, edge detection, etc..

**3.5 :Can you find a way to further improve this CNN architecture, either by improving its hyperparameters, decreasing its parameter count by adjusting its architecture, or some other method? Explain your approach and reasoning

There are several ways to improve the CNN depending on the desired end result. To decrease the number of parameters, we could use a 32-channel convolutional layer followed by a 16-channel convolutional layer. Generally, having fewer convolutional layers and narrowing convolutional layers would dramatically reduce training time and decrease parameter count, as there would be fewer layers and layers would be smaller. We can also add more linear layers at the end to increase the expressivity of the model. The current model only has two linear layers (one of which maps to the 10 output classes), so enlarging this MLP at the end could enhance the expressivity of the model. Further, we can also add batch normalization in the end MLP or apply layer normalization techniques more broadly to accelerate convergence and improve generalization. My approach depends on what particular result I wanted to prioritize. Since training took very long, I found that reorganizing the architecture such that there were only two convolutional layers (32 and 16) where the second layer was narrower than the first greatly sped up the training process.

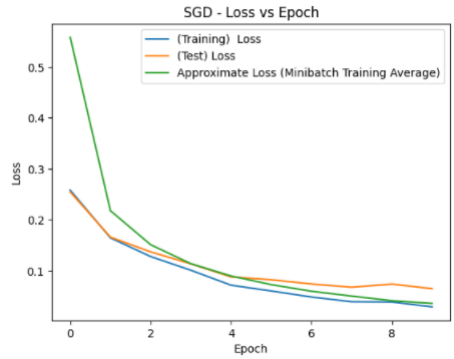
```
conv adam
820.3390882015228
```

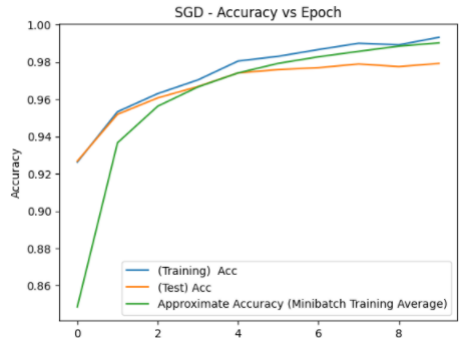




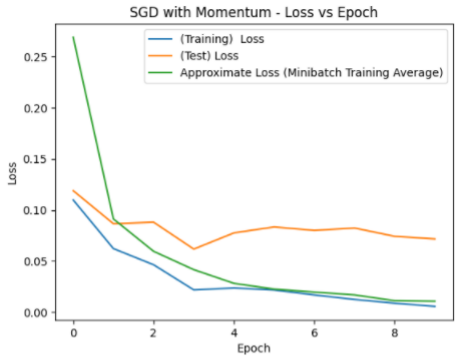
Part 1 PLOTS & WALL RUNTIMES (SECONDS)

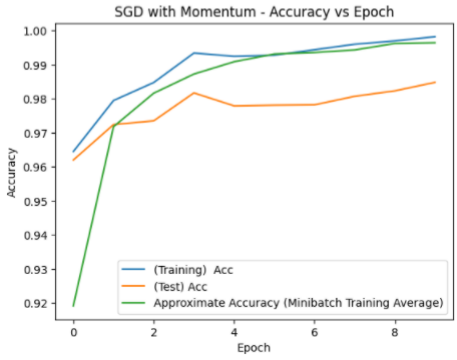
```
sgd
201.54192733764648
```



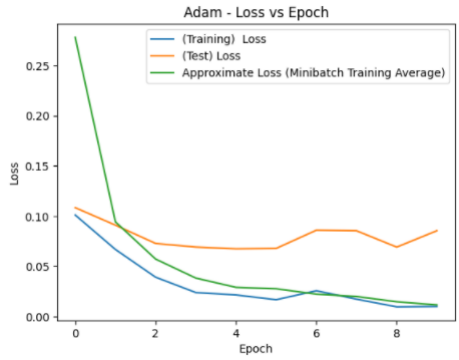


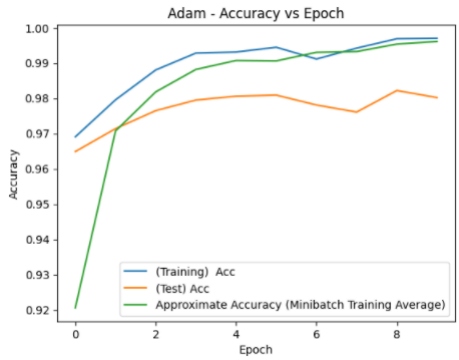
sgd momentum
216.6392765045166



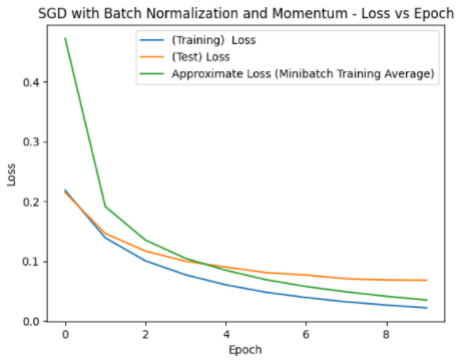


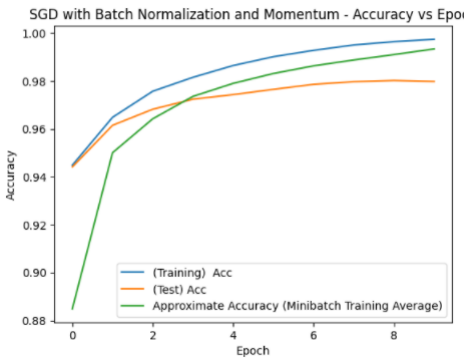
adam
236.41820192337036





bnorm
210.8349528312683





**