Programming Assignment 3: Scaling with SGD

CS4787 — Principles of Large-Scale Machine Learning Systems

Part 0: Summary

In this assignment, I was able to compare the performance of ADAM, Momentum and minibacth SGD as well as regular GD for training a machine learning model when this is framed as an optimization problem. As we have seen before, SGD on average decreases the loss monotonously for a good choice of step size α. Interestingly, this is not the case when we use Nesterov's Momentum update rule, which very often increases the loss function first and might then begin to decrease it although the final loss final will vary a lot with the parameter β. Nota that this might be due to the effects that the update rule has on the models parameter's sizes, since we are using l_2 regularization with parameter $\gamma = 0.0001$ for the **entire** project and exploration, and we can see that although the loss increases, the rate might be decreasing. Finally, ADAM actually does monotonously decrease the loss function, but it also decreases the error rate at a much larger rate than all the other algorithms, while still having a runtime comparable to that of SGD. Lastly, as one would expect, the minibatch versions of each algorithm executed much faster than their non-minibatch counterparts.

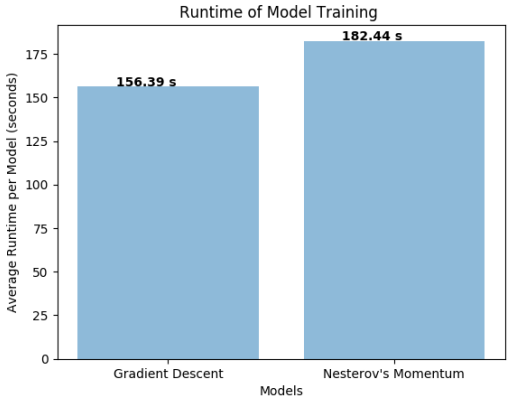
Part I: Momentum and Gradient Descent

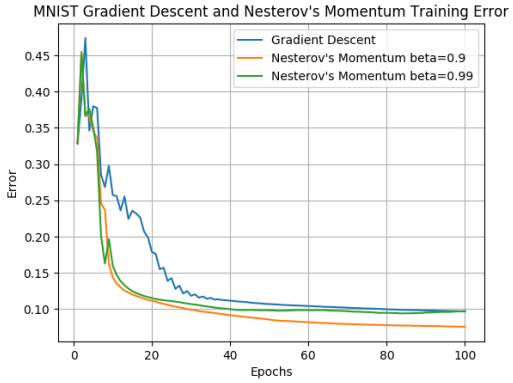
I ran Gradient Descent once with step size $(\alpha = 1.0)$, and Nesterov's Momentum twice using the step size $(\alpha = 1.0)$ for both runs, but varying the momentum parameter β_1 with values $(\beta_1 = 0.9)$ and $(\beta_2 = 0.99)$. Each algorithm ran for a total of 100 epochs, and the model parameters were recorded at each epoch. Then, we computed the training error, test error, and the value of the loss function over the training set for all the outputted models for each run and plotted them. The results can be seen on the right.

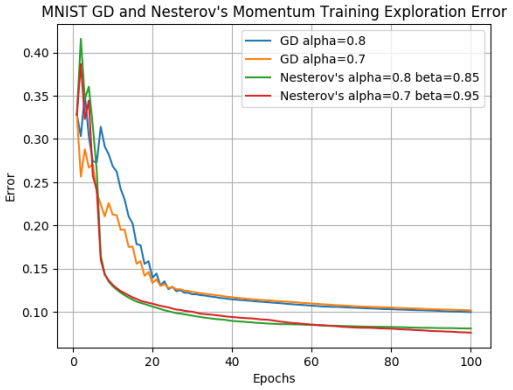
Regular Gradient Descent converges noticeably slower than the runs with Nesterov Momentum update steps, particularly on epochs 10-30, although by the 60th epoch GD and Nesterov's Momentum with $(\beta = 0.99)$ have attained comparable accuracies on the test set. On the other hand, both runs of Nesterov's remain quite close for the first 30 epochs, but the model with $(\beta = 0.99)$ plateaus at around 10% training and test error rates, while for $(\beta = 0.9)$ we see the error rate continue to decrease on the training and test set to about 8%.

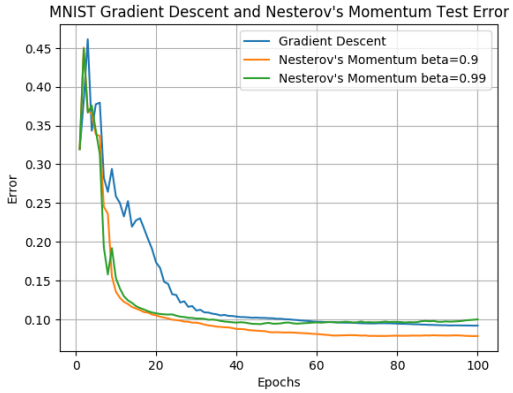
The effect of β on the training loss is quite interesting: while gradient descent monotonously decreases the training loss over the entire training, the moment algorithms actually bring it up abruptly at the beginning and proceed to lower it. Notice however that $(\beta = .9)$ actually decreases it enough to eventually overtake GD, while $(\beta = 0.99)$ will again increase it and finish off at a much higher loss than the other two.

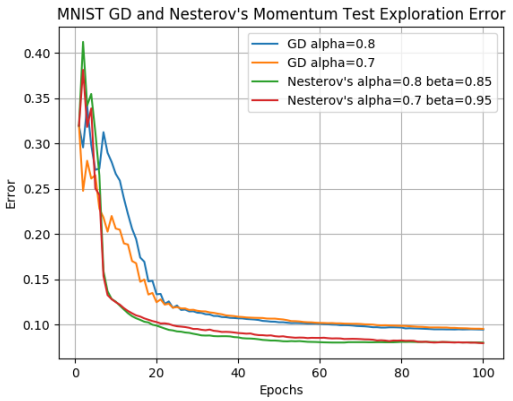
Finally, I measured the runtime of gradient descent, and Nesterov's momentum by training for 100 epochs using the same step size α for both, and $(\beta = 0.9)$ as the momentum parameter. I ran each of these a total of ten times, and averaged the runtimes for each algorithm. The plots can be seen on the right. Momentum evidently takes slightly more time to run, with a ratio of about 5:6 on average. It would seem, nonetheless, that if one has enough compute time, it would be worth it to use momentum since it could be ran for just slightly less iterations than gradient descent and it would likely attain a comparable if not better test error while still satisfying time constraints.

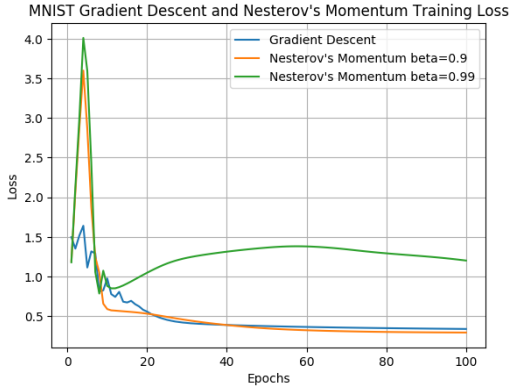


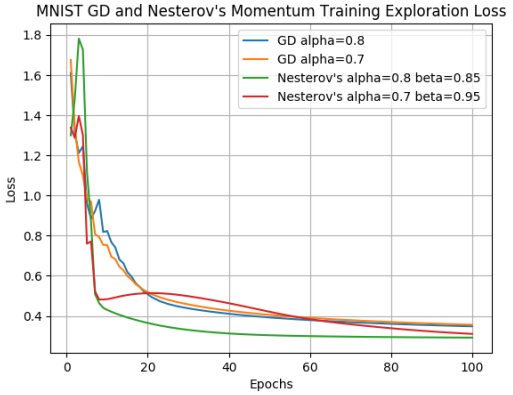












Part I: Exploration

We proceeded to explore other combinations of hyperparameters for these two algorithms and compared the outputs. The plots can be seen below. Both settings of β decreased the error rate at a much faster rate than both settings of gradient descent descent, and are much less noisy. The runs of GD with different values of α actually performed very similarly, and the same was true for their momentum counterparts. Momentum, thus, seems like the better option.

Part II: Momentum and SGD

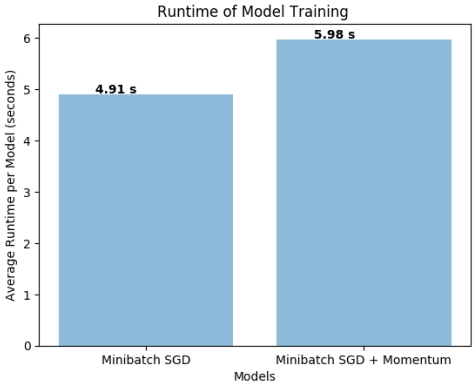
I also compared two sequential variations of gradient descent, namely *Sequential Minibatch SGD*, with and without momentum. I ran regular sequential minibatch SGD once, and its modified momentum version twice, once with momentum parameters $(\beta = 0.9)$ and $(\beta = 0.99)$ respectively. All three executions used a step size $(\alpha = 0.2)$, and went through 10 epochs recording the model parameters every $1/10$ of the epoch (at every full gradient step, with minibatch size $B = 600$ on

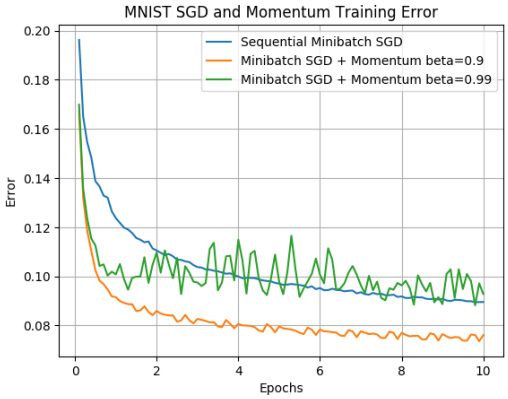
the MNIST training set which contains 60,000 points.

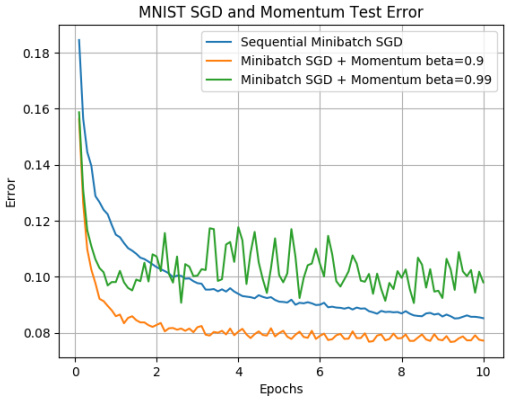
The training and test error as well as the cross-entropy loss on the training set were computed and recorded per model update and the plots are shown on the below. Again, the momentum optimization with $(\beta = 0.9)$ outperformed the others, reaching a training and test error of about 8% while momentum with $(\beta = .99)$ remained at about 10% error on the test set and SGD finished at around 9%. SGD and momentum with $(\beta = .99)$, on average, did continue to decrease the training and test error through all the 10 epochs, but momentum with $(\beta = .9)$ was far ahead by the fouth epoch, and still continued to decrese the error rate although rather slowly while regular SGD was slowly catching up.

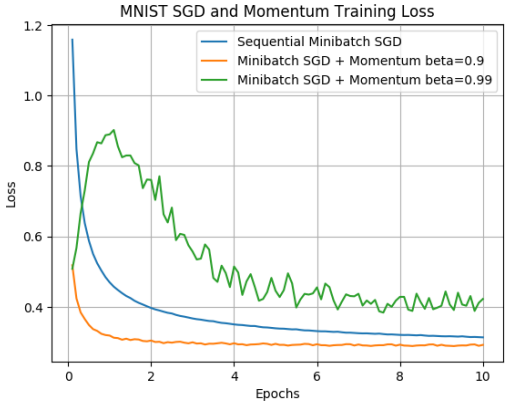
SGD monotonously decreased the loss function throughout training, although rather slowly compared to momentum with $(\beta = .9)$, which decreased the loss monotonously and almost instantaneously and then plateus. On the other hand $(\beta = .99)$ increases the loss to later decrease it gradually and to finish off worst than both of the others. The value of β thus seem to be very crucial for this momentum optimizer to perform.

Finally, I also averaged the runtime of both, sequential minibatch SGD and it's momentum counterpart over five runs, each of which completed 10 epochs of training as summarized above. the results are summarized on the plot below: both algorithms ran much faster than their non-minibatch counterparts, reaching lower training and test errors in just 5-6 seconds with the non-momentum version finishing faster.





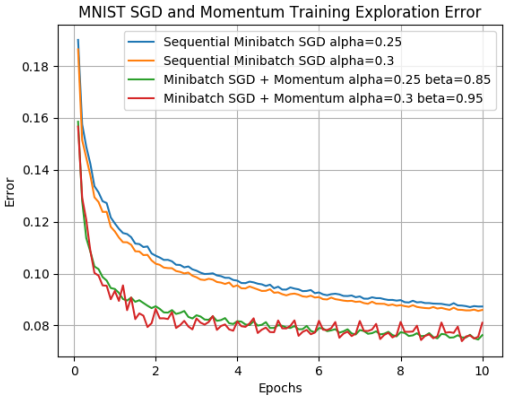


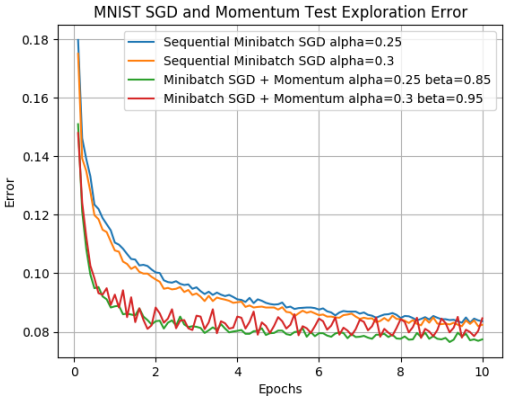


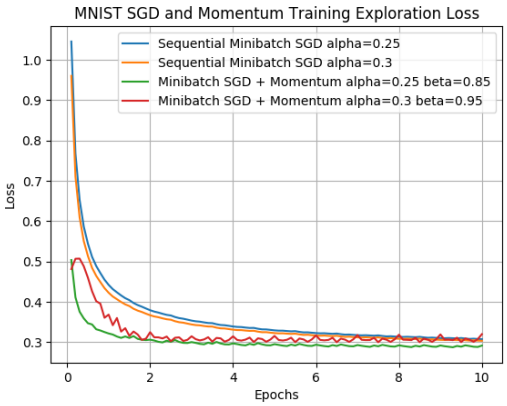
Part II: Exploration

A few more settings of α and β were tested, and the plots of the training and test errors, as well as the cross-entropy loss on the training set are plotted below for each of the optimizations. Over all these parameter settings, SGD with momentum still performs better, with $(\beta = .85)$ bringing the test error below 8%. The sequential algorithms are rather close, nonetheless, giving a comparable test error to that of the rather

noisy momentum run with $(\beta = .95)$. The non-momentum optimizers slowly and monotonously decrease the loss over all training, while $(\beta = .85)$ decreased it again almost instantaneously over the first epoch On the other hand, $(\beta = 0.95)$ actually briefly increases the loss, and then decreases it monotonously to achieve similar results, althought its trajectory is clearly more noisy in terms of both error rates and the cross entropy loss.





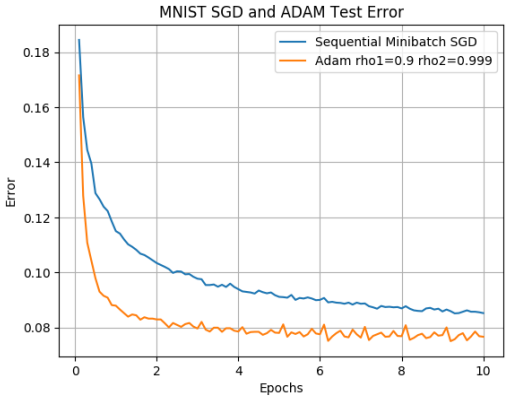


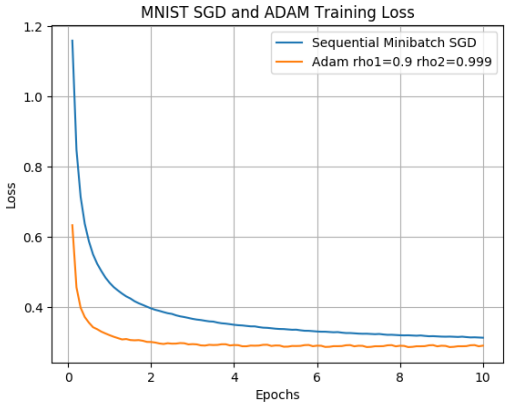
Part III: ADAM

Finally, I also compared our old friend, *Sequential Minibatch SGD* with the grandfather of adaptive algorithms, *ADAM*. I executed another independent run of SGD to the one shown in Part II, with $\alpha = 0.2$ over 10 epochs, computing the training and test error as well as the value of the loss function on the training set 10 times per epoch. ADAM was executed in a similar fashion, but using step size $(\alpha = 0.01)$ and respective first and second decay rates $(\rho_1 = 0.9)$ and $(\rho_2 = 0.999)$. The results are summarized in the plots below.

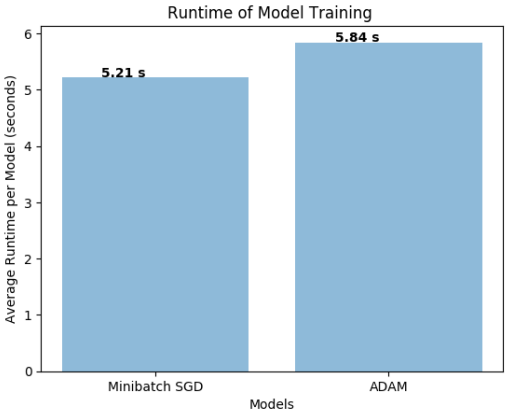
ADAM is able to bring the training and test error down, at an incredibly fast rate over the first epoch (at least comparatively), but the test error slows down and almost plateaus right away as well. In fact, it would seems as thought the training error for ADAM is just the training error for SGD shifted down by two percent, but we can see that the same does not hold for the test error. Nonetheless, the test error is still about 1% smaller for ADAM than it is for SGD.

The training losses, just like the training errors, have extremely similar shapes, but ADAM's is shifted down. In addition, both SGD and ADAM monotonously bring the loss down, unlike SGD with momentum as we previously saw. Nonetheless, both optimizers have a comparable final loss. Nonetheless, this difference in loss translatates to about a 1% difference in test error.





I also measured the runtime for both algorithms, but now over a total of 15 independent executions since the results seemed to vary largely. Both algorithms performed comparatively, with SGD being only slight faster at about 5.2:5.8 seconds. This is great news, if there's any time constraints, we could just run ADAM since it seems like for any allotted time Δt at which SGD might give reasonable good results, ADAM is likely to perform better by the plots below.



Part II: Exploration

I explored other settings of α and β that would give reasonable accuracy, as shown below. As we would expect, ADAM performed better for any of the four parameter settings tested. Interestingly, as shown on the plots, all four parameter settings performed very similarly on the test set, and each pair of ADAM optimizers with same value for α behaved almost indistinguiably in terms of the loss. This suggests that determining a good value for α is still very important. In terms of accuracy and runtime, then, ADAM seems like the clear winner.