**Stochastic gradient descent, and what Adam does**

Stochastic gradient descent (SGD) is the most basic method of gradient optimization, which breaks up the training data into minibatches, and for each minibatch calculates the gradient of the loss with respect to the model parameters, and steps in the opposite direction. It is called stochastic because the minibatches are randomly created each iteration through the dataset. The step size is the gradient multiplied by the learning rate, commonly between 10-1 and 10-5. This is done because backpropagating through the entire dataset to calculate the exact gradient is costs a lot of memory and computational power, and in most cases, SGD is basically as good.

Text, letter

Description automatically generated

Weight update formula for SGD, where eta is the learning rate, C is the cost function

SGD is a 1st order optimizer, relying on only the 1st gradient, which means the optimizer is based on a local linear approximation of the loss. Exploiting curvature in the loss function can speed up optimization but is again very computationally expensive. To get around this, momentum was introduced, which is a way to exploit the curvature in the loss without having to explicitly calculate it.

A picture containing text, clock, watch

Description automatically generated

Weight update formula for SGD with momentum, where beta is a momentum parameter. In practice, there is also a learning rate multiplied onto the momentum (mi) as its added to the weights.

Another problem with SGD is the dual problems of “exploding gradients” and “vanishing gradients,” which is when elements of the gradient become incredibly large or incredibly small and make iterative optimization difficult. This was solved with adaptive optimizers, like Adagrad and RMSProp, which scale elements of the gradient too avoid this.

A picture containing clock

Description automatically generated A picture containing clock

Description automatically generated

Adagrad on the left, RSMProp on the right, where g is the gradient, and squaring and division are done element-wise.

Adam was introduced in a paper in 2014 (can be found at <https://arxiv.org/abs/1412.6980>), and combined elements of Adagrad, RMSProp, and uses momentum, and has become the go-to optimizer for a variety of problems.

A close-up of a document

Description automatically generated with medium confidence

Adam algorithm, as described in the paper. Alpha, beta 1, and beta 2 are the learning rate, and momentum parameters.

**Linear Regression problem, as an introduction**

In order to figure out what Adam (and gradient descent in general) did, I decided to start with a simple case that I could visualize, which was linear regression, i.e. creating a line of best fit. I did this by creating a fake dataset of points randomly dispersed around a line, so that I knew already what the “true” line of best fit is. Then I used a model with a single layer, and MSE loss.

Chart, scatter chart

Description automatically generated Text

Description automatically generated

On the left an example of the test/validation set, and on the right the pytorch code for the model.

As expected, the model converged very easily, and found the line of best fit. What was interesting was when I plotted the change in the model parameters (the slope and y-intercept) over each step that Adam took (every minibatch).

Graphical user interface

Description automatically generated

On the top left, is the loss recorded during training for each minibatch, and the green vertical lines denote each epoch. The top right is the training loss averaged over each epoch, and the bottom left is the test loss for each epoch.

Chart, scatter chart

Description automatically generatedChart, line chart

Description automatically generated

On the left, you can see the predicted line of best fit as the model learns, going from black to yellow. On the right, you can see the parameters as the model learns, with the slope m on the x-axis and the y-intercept b on the y-axis. The center of the green crosshairs is the true values that the dataset was generated based on.

You can see that the initial y-intercept was already close to the true value, but instead of the model going directly towards the optimal parameters, it increases both values, and then the path turns and goes towards the final values. This was unexpected and seemed to happen for almost every trial, so for this case I tried to figure out what kind of loss landscape would cause this, and luckily since there are only two parameters in the model, I plotted the loss at each point in “parameter space,” and overlaid the path. After the fact, it became clear that the reason for this was that the easiest way for the model to reduce the MSE was to shift the line of best fit through the center of the data, and then after that it rotates the line to the optimal slope.

A picture containing background pattern

Description automatically generatedChart, surface chart

Description automatically generated

On the left is the path of a different trial, with the value of the loss plotted as a heatmap underneath. On the right the 3D contour plot of the loss, where you can clearly see the oval shaped global minimum.

I was also able to instrument Adam, and have it return the gradient, momentum, and second momentum at each step, so I could see what Adam was doing during the training. Here I have shown the magnitude of the step Adam took, the magnitude of the gradient, and the magnitude of the momentum at each step. Note that the learning rate here was 0.001, which is why the scale of the step sizes is about 1000 times smaller than that of the gradient and momentum. Initially the gradient and step size is larger, but gets smaller around the same time that the model enters the global minima shown previously.

Chart, histogram

Description automatically generated

Below on the left is the normalized dot product (where orthogonal is 0, same direction is 1, and opposite direction is -1) between the gradient and the step Adam took. If this was SGD, the normalized dot product between the step and gradient would always be -1 since SGD steps in opposite direction as the gradient. However, here we can see this is true in the beginning (when the gradient is larger, see above), but as training progresses the gradients tends to be more and more positive. This indicates that the gradient from minibatch to minibatch is less and less uniform, which would happen as the model is very close to optimal and the specific points in the minibatches might have slightly different optimal best fit lines. The normed dot product of the step and momentum being close to -1 makes sense, because it would only be different if elements of the gradient were extraordinarily large or small.

Chart, scatter chart

Description automatically generated

The normalized dot products between steps and gradients, and steps and the momentum.

An interesting test I did quickly was to see how this would look for a system with more local (and global) minima, so I reframed the problem by taking putting the output into a cosine function and adding a cosine activation to the model. Because the cosine function is unchanged when you change the argument by any integer multiple of 2π, there should be an infinite number of optimal global minima. And that is what I found, as well as a number of less optimal local minima, and it is a cool exercise to come up with reasons for the shape of the loss landscape (using symmetries of trig functions, and what happens as m goes to infinity?).

Chart, scatter chart

Description automatically generatedGraphical user interface

Description automatically generated

On the left, the dataset and cosine function of best fit. On the right, the loss landscape where you can clearly see multiple global and local minima.

Making the linear regression problem more complex, by increasing the size of the input and output vectors made it harder to visualize, but I found that I could visualize it using PCA, which takes a series of n-dimensional vectors, and projects them onto a lower dimensional plane which accounts for most of the variance between the vectors. Doing the same linear regression problem but making the input (x) and output (y) each be 10 dimensional vectors increased the number of parameters from 2 to 110, I found that the model converged very quickly to the global minima, but also that the PCA projection into 2D (and even just 1D) accounted for a huge amount of the variance. For this run, which was representative of most trials, the first principal component accounted for around 97.5% of the variance, and the second, 2.23%, which means the PCA projection from 110 dimensions to 2 dimensions retains 99.79% of the variance in the model’s parameters as it trained.

A picture containing chart

Description automatically generatedChart, surface chart

Description automatically generated

On the left, the PCA projection of the model’s parameters over the course of training, with the loss at each point on the plane (which is not exactly the loss at each point in the training, but very close). On the right, the loss on the same plane shown as a 3D contour plot.

This part of the study served two main purposes: first, it was a way for me to learn how to use and test modifications to pytorch code on an example which I could verify, and second, it serves as a control for what a simple optimization with known global minima/minimum to compare with more complex realistic problems. It also was a good introduction into coming up with ways to show the training process, by looking at both a plot of motion in parameter-space (or a PCA projection of it), and by looking at the size and angles of gradients and steps.

**Applying results to MNIST image classification**

The next thing I did was to apply these techniques to a common benchmark problem, which is classifying a set of handwritten numbers from the MNIST dataset, using a simple model with a single hidden layer. The goal was to see how this problem compared to the linear regression problem by using the same tools.

Chart

Description automatically generated

The loss and accuracy (fraction of images correctly recognized) for the training.

The model converges relatively quickly and reaches around 90% accuracy. This model 15910 parameters and using PCA on the model’s parameters over training a to project it down onto 2 dimensions accounts for about 99% of the variance (first principal component: 93.2%; second principal component: 5.6%), which was a surprising result to me, considering that the problem seems much more complex than the linear regression. The method of calculating the loss on the plane formed by the model’s initial state and the first PCA vectors also worked, and clearly showed the gradient that the optimizer was following.

Chart

Description automatically generated Chart, surface chart

Description automatically generated

On the left, the model’s parameters over training (which goes from black to red to white) projected onto 2D using PCA, with the loss calculated on the plane made from the first 2 principal component vectors and the model’s initial state. On the right, the same loss as a 3D contour plot.

However, you can see that the final part of the training does not exactly follow the slope of the loss as shown, which is because this is just a PCA projection and does not capture all of the behavior. I have found that longer training makes the 2D PCA worse, most likely because the model starts moving in different directions near the end of the training than it did in the beginning. If I do a longer run, to try to get the model to a final minima, and do PCA on the end of the training, you can see a minima that is not apparent when you look only at the entire path.

A picture containing chart

Description automatically generatedChart

Description automatically generated

On the left, PCA on the entire path, origin set to initial model state. On the right, PCA on the last 1000 steps, and the origin set to the model’s final state.

**Stochasticity and initialization**

Machine learning is stochastic (random) in two main ways: initialization and minibatch selection.

Randomness from initialization is the fact that the parameters for a model are commonly automatically set to some random values, usually a distribution based on the size of the layer centered around zero. One question that came up during this project was how much the initialization affects the results of training.

The minibatch selection is usually randomized, where for every epoch the dataset is reshuffled into minibatches so that the minibatch gradient estimation is more accurate. However, this means that the minibatches used during training are different for every run, and so the gradients used are different, which means that the model’s parameters do not necessarily follow the same path during training.

To do a quick test of the impact of these, I ran multiple trials of the MNIST classifier, one set with the same initialization, one set with random initialization. You can see most of the runs have slight variations, and there is one run from the set with random initializations that has significantly lower accuracy than the rest.

Graphical user interface

Description automatically generated Graphical user interface

Description automatically generated

On the left, the graphs from training 5 trials of the MNIST classifier with random initializations using pytorch’s default initialization method. On the right, the same but using the same initialization for all 5 trials.

In order to compare the parameters of the models at different points through the training, I used a plotmatrix for the initial states and the final states. Each plot in the plot matrix takes two model states, and each point is the pair (value in model 1, value in model 2). Along the diagonal of the plotmatrix, the plots show all the points along the line y = x because the models are identical.

Diagram

Description automatically generated with medium confidence Diagram

Description automatically generated

The plotmatrices for the trials with random initializations. On the left is the initial states, and you can see the uniform distributions of the parameters from the square shape. On the right is the final states after training.

A picture containing chart

Description automatically generated A picture containing text, tree, surrounded

Description automatically generated

The plotmatrices of the trials with the same initializations. On the left is the initial states, and they are all identical. On the right is the final states, and you can see they are stretched along the line y = x more than the final states for the random initializations.

You can see there is a visible difference in the final states of the model when they use the same initialization, and they are more similar then with the random initializations. However, it is also clear that they are not at the exact same final state either.

**DDplus**

The motivation for this project was to speed up the training of models used for processing data from particle colliders, and one of the models I studied was the DDplus model, which takes measured particle tracks and outputs a Kernel Density Estimate (KDE), which is describes the likelihood of a collision have some z-coordinate (where the beam goes along the z-axis).

My perspective going into this study was roughly this: given a model, a loss function, and a dataset, which in combination describe some high-dimensional loss landscape, how can you improve an optimizer? In other words, I mostly ignored any improvements that could be made through changes in model architecture and the loss function, although I did experiment with the amount of data that is used.

Chart, line chart, histogram

Description automatically generated Chart

Description automatically generated

On the left, an image of the late-stage training, from the word doc “DDplus\_History\_in\_part.docx”. Note that the labels on the right plot are incorrect and is just a linearly scaled version of the one on the left. On the right, an image of the true KDEs (blue) and the predictions from the model at the end of this training (red).

The problem initially described to me was that the model took an incredibly long time to train during the end of training, as shown above. And a note for scale, each epoch took around 800 seconds, and the model has trained for hundreds of hours of computational time to get to this point. However, you can see the model has learned to predict the KDEs relatively well by this point.

For the smaller models previously, the analysis I did was primarily based on looking at the changes in the model between each step the optimizer took, which happened every minibatch. But for larger models run for a longer period, it’s not possible to store the information for every minibatch, just because of memory usage. So instead, I switched to using information from the end of every epoch of training, which often contains hundreds or thousands of steps taken by the optimizer (and therefore takes hundreds or thousands times less memory to store). In particular, PCA is not possible for larger models (or at least not easily), because the amount of memory needed scales with the number of parameters squared.

The term ‘epoch step’ is what I will be using to refer to the change in the model’s parameters over the course of an epoch and is the sum of the steps the optimizer took over the course of an epoch. And one hypothesis I was testing was that because an epoch iterates through the entire dataset, the overall direction of the epoch would be in the direction of the true gradient (as opposed to the minibatch estimations used within the epoch) and so maybe taking steps in the direction of the epoch might speed up training.

I decided to first verify whether consecutive epochs were in the same direction, because if they were then it makes a lot of sense to take an additional step at the end of the epoch to speed up training. The most interesting and useful visualization I made was to plot a grid comparing the angle between every epoch step with every other step during the training.

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated Chart, line chart

Description automatically generatedA picture containing chart

Description automatically generated

Lin. Reg. (100D input and output vectors): On the top left, the training loss over each epoch. On the top right, the PCA projection (accounting for 99.8% of the variance) of the training, with the origin on the mean of the path over training. On the bottom left, the norm of the epoch steps. On the bottom right, the angle grid for this run. The color scale on the right shows the angle (in radians) that each color represents.

Above, you can see what this shows for the linear regression problem. The top left corner, in blue, shows that the first 10 epochs are roughly in the same direction, and then the next 50 or so epochs after that are also all roughly in the same direction. And because the top edges are orange, which is roughly π/2, you can see that the first 10 epoch steps are all orthogonal to the next 50 steps, which is visible in the PCA plot of the path. The last around 35 epochs are clearly not in the same direction, and you can see the loss stops increasing at this point, which makes sense because the model is jumping back and forth around the global minimum at this point.

Chart, line chart

Description automatically generated Shape

Description automatically generated with low confidence Chart

Description automatically generatedA picture containing chart

Description automatically generated

MNIST: On the top left, the training loss over each epoch. On the top right, the PCA projection (accounting for 99% of the variance) of the training. On the bottom left, the norm of each epoch step. On the bottom right , the angle grid of the training.

For MNIST, you can immediately see that the angle grid looks very different. Instead of having regions of training orthogonal to each like you could see for the linear regression PCA projection and angle grid, here you can see the motion in parameter space curves gradually, and you can also see that from the angle grid. Consecutive epochs are still relatively collinear, but 15 epochs apart from each other are much less collinear.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated Chart

Description automatically generated

On the top, the training loss for each epoch over training. On the bottom left, the norm of each epoch step. On the bottom right, the angle grid for this run.

Above is the same analysis for DDplus training, and the learning process is more complex. The region until about 20 epochs looks like the MNIST training, especially because the size of the epoch steps is also descreasing. However, unlike the MNIST training at around 20 epochs the size of the steps starts to increase and the model’s path in parameter-space starts to curve more (because it is more red). However, you can see that both the angles between the epoch steps, the size of the epoch steps, and the training loss starts to jump near the end. This is even more apparent if you overlay the training loss on top of the angle grid.

Chart, histogram

Description automatically generated

The training loss on top of the angle grid. You can see the changes in loss roughly correspond to changes in the angles between epoch steps.

This is only during the first part of training however and looking at part of the training when the model is already very good shows that the behavior is quite different.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

A picture containing histogram

Description automatically generated

On the top left, the training loss for each epoch. On the top right, the norm of the epoch steps. On the bottom, the angle grid for this training.

Above you can see the same analysis done when the model is already very trained (notice how the loss is around 1, instead of around 10). Instead of jumping down in loss after a period of almost no change, here we can see the model is constantly, but slowly, decreasing in loss. The angles between epoch steps and the size of epoch steps look very similar to the MNIST analysis, which correlated to the model’s moving on a smooth curve through the parameter-space.

I also briefly looked at some of the other models used in the group, the first being the AllCNN model, which is used to go from KDEs to histograms (instead of tracks to KDEs like DDplus), and you can immediately see massive qualitative differences

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

Graphical user interface

Description automatically generated

On the top left, the training loss. On the top right, the norm of the epoch steps over training. On the bottom, the angle grid between the epoch steps over training (the color shows the angle in radians).

The AllCNN model’s training already looks very qualitatively different than the any of the previous models. One similarity I will note is the end of the training here (after about 50 epochs) is similar to the training of the well trained DDplus model, and somewhat similar to the MNIST training.

At this point, I started testing some ideas to try to speed up the training for these cases, given what I had learned from the analysis of the training. The first algorithm I experimented with was the EVE algorithm, which was based on an idea from Dr. Sokoloff. The idea of this algorithm was to speed up training by guessing and verifying that the model would improve by continuing to change the parameters in the overall direction of any given epoch, which we can see some evidence for in the previous analysis. In practice, what I did was calculate a series of parameter states by taking the final state of a model at the end of an epoch and the change in parameters over the course of that epoch, and then adding some scalar multiple of that change to the final state. Then I evaluated the model’s loss for a variety of scalar multiples, and finally set the model’s state to be whichever of those had the minimum loss.

I tested it on the linear regression problem:

Graphical user interface

Description automatically generated

Chart

Description automatically generated with low confidence A picture containing chart

Description automatically generated

On the top, the training loss for each epoch. On the bottom left, the angle grid for training the linear regression model with Adam, and on the bottom right is the same with the EVE algorithm at the end of each epoch.

You can see that the EVE algorithm does decrease the loss per epoch over training, and the angle grid shows that the model takes consecutive steps that are roughly orthogonal to each other.

Another thing to see is plotting the loss in the direction of each epoch from the beginning of the epoch as a function of the scalar multiple, e.g. the point on the x-axis 2 represents the model’s state if the model’s parameters changed by twice the amount as they did during the epoch.

A picture containing diagram

Description automatically generated

The loss in the direction of each epoch for Adam.

Diagram

Description automatically generated

The loss in the direction of each epoch for Adam with EVE.

As you can see, the minimum of the loss is uniformly at 1 for Adam with EVE, which means that the EVE algorithm is correctly choosing the step size for each epoch.

One of the biggest problems with the EVE algorithm is that evaluating the loss can take a long time with a large dataset, and so to use the algorithm with DDplus I spit the training dataset into a large subset to use to calculate the gradients and iterate through, and a smaller subset that is used to calculate the loss for the EVE algorithm.

A picture containing histogram

Description automatically generatedChart, histogram

Description automatically generated

On the left, the angle grid for training DDplus with Adam, and on the left is training with Adam and EVE every epoch.

You can see a similar effect with DDplus, particularly when the learning rate is relatively low (1e-5), as shown above. The region where consecutive epoch steps are mostly collinear on the left is replaced by a region where consecutive epoch steps are mostly close to orthogonal, and you can see that it takes about half as many epochs before the loss starts to drop below about 11.8.

Chart, line chart

Description automatically generated Chart, line chart

Description automatically generated

Graphical user interface

Description automatically generated Chart

Description automatically generated with low confidence

On the top left, the training loss for AllCNN with Adam, on the top right, the same but with EVE. On the bottom left, the angle grid training the AllCNN model with Adam, and on the bottom right the same but with EVE.

There is a similar effect with AllCNN, and you can see that parts of the training are sped up and that the consecutive epoch steps are more orthogonal with the EVE algorithm than with Adam. Again, you can also see that certain features seen on the angle grid for Adam are shrunk on the angle grid for EVE, which I concluded that the EVE algorithm speed up parts of the training that would have otherwise taken more epochs without it.

I next looked at the initial DDplus training and tried to address the problem of the loss jumping repeatedly, instead of monotonically decreasing.

Chart, histogram

Description automatically generated

For the above set of runs, I first ran the Adam algorithm with a learning rate of 1e-5 for 700 epochs on a small dataset (18 thousand events) to see how it performed, which is the blue line you can see. After that, I started again at 450 epochs (the vertical dashed line) but added a learning rate schedule to reduce the learning rate to see if it could prevent the model from jumping, and it seemed to work. Careful Adam is what I called an automatic learning rate scheduler I implemented. Specifically, if the training loss increased by more than 2% over an epoch, the learning rate was decreased by 20%. And this seemed to work even better than the learning rate scheduler and speeding up the training.

Chart, line chart

Description automatically generated

For the above set of runs, I then tried some runs on a larger dataset, 160 thousand events, and this time Adam had a learning rate of 1e-4, and you can see the model already trains quicker for both optimizers (even though each epoch here took about 10 times longer than for the 18k event dataset). I started the runs from the same initial state, and you can again see that the careful algorithm helped the model converge much more quickly.

The careful Adam algorithm was a way to automatically decrease the learning rate if it was too high but based on the previous analysis, we could see in many cases that there are regions where you would want to increase the learning rate to speed up the training. In particular, the hypothesis I had was that if consecutive epoch steps were mostly collinear, then it would be beneficial for the model to take larger steps. So, I implemented an algorithm I called adaptive Adam, which increases the learning rate by some factor proportional to the normalized dot product between the two most recent epoch steps, using the following formula:

where the normalized dot product is calculated from two vectors with

Chart

Description automatically generated Chart

Description automatically generated

On the left, the angle grid from training AllCNN with Adam, and on the right is with the adaptive algorithm. Both models had the same initial state

Chart, line chart

Description automatically generated

Here is the loss over training the AllCNN model with Adam and adaptive Adam. Technically, the careful algorithm was used too, but there was never a large enough increase in the training loss for it to do anything, so it was a non-factor in this training,

Since the AllCNN model had a lot of consecutive epochs that were relatively collinear, I figured that the adaptive algorithm would speed up the training, so I trained the AllCNN with Adam and adaptive Adam, starting from the same initial state, and as you can see the training was significantly faster. I used an increase factor of 15% here, so if two consecutive epochs were perfectly collinear, the learning rate would increase by 15%. And you can also see that the angle grid for the training with adaptive Adam shows that the model’s consecutive epoch steps are much less collinear than for the training with Adam, which in this case verifies my hypothesis.

Chart

Description automatically generated A screenshot of a computer

Description automatically generated with low confidence

On the left, the angle grid for the epochs 25-50 for careful Adam training. On the right, the same thing but for adaptive Adam training.

Chart, line chart

Description automatically generated

I tried the same algorithms combined on DDplus, and as you can see it did not help that much. I am guessing that this was because the consecutive steps were not as collinear as in the AllCNN training, and so the training was not really sped up that much for the first 40 or so epochs. My conclusion from this was that the adaptive Adam method is not necessarily the best method for all models, and maybe to make it more generalizable it is better to have the increase factor be lower, so it is a more subtle effect.

Based on the experiments and observations above, and over the course of the entire summer, I came up with a list of the improvements I thought could be useful to implement to speed up the training of models:

1. The **EVE** algorithm, which works at the end of each epoch. It takes steps in the overall direction of the epoch and evaluates the loss at those points. If it finds that the loss is smaller at one of those steps, then it takes a step in that direction (Credit to Dr. Sokoloff for this idea). The goal of this is to be able to speed up the training in regions where the consecutive epoch steps are in the same direction.
2. An **epoch step**ping algorithm, which takes a step in the direction of an epoch at the end of the epoch based on how collinear the previous epoch step(s) are. A less complicated and faster to implement version of the EVE algorithm, which has the same goal.
3. A **careful Adam**, which detects when the loss jumps up from one epoch to the next and reduces the learning rate. The goal of this is to get rid of the jumps in loss that you can see in the DDplus training, which I am hypothesizing are the result of jumping out of a saddle point and not a local minimum.
4. A **more careful Adam**, which detects when the loss jumps from one epoch to the next, reduces the learning rate, and restarts the epoch. This has the same goal as the careful Adam above but is more extreme and basically prohibits any large increase in the loss.
5. An **adaptive learning rate**, which changes based on how collinear consecutive epochs are. If consecutive epochs are in the same direction, then the learning rate increases. This is a way to speed up the training if epoch steps are all in the same direction, like the EVE and epoch stepping algorithms. My hope is that combining this with the careful Adam algorithms will provide a way for the learning rate to both increase and decrease automatically, depending on how the model is training.

I did not have time to fully test all of these ideas, but to conclude this I will include some plots comparing these algorithms on initial DDplus training. Note that fixedcareful Adam and careful2 Adam are versions of careful Adam that reset the model’s parameters (but not the optimizer state) if the model’s loss increases too much.

These plots can be made from “various\_DDplus\_runs\_compared.ipynb” and “Sep14\_DDplus\_runs\_compared.ipynb” in pv-finder/baileyds\_stuff/DOC\_code/DDplus/

(from <https://github.com/sullybd/pv-finder>)

Graphical user interface, chart

Description automatically generated

This is a plot of runs comparing training with an 18 thousand event dataset. The spike in the green and yellow lines are due to increasing the size of the dataset, and so can be somewhat ignored. From this plot, it looks like careful Adam algorithms easily outperform Adam.

Chart

Description automatically generated

This plot compares runs with a 80 thousand event dataset, and unfortunately I did not do a control run with just Adam. However, you can again see that careful Adam did the best, while the adaptive + careful algorithms performed similarly. The green line at the top did not go down either due to initialization or a low learning rate.

Chart

Description automatically generated

This plot compares runs with a 160 thousand event dataset, and you can again see that the careful Adams outperformed both Adam and Adam with EVE.

Chart

Description automatically generated

Here is a plot comparing the careful Adam algorithms with varying dataset sizes, plotting the loss versus roughly the computational time. It makes sense that the 80k and 160k outperform the 18k, but it is strange that adding more data makes it worse in the 320k case.

Chart, line chart

Description automatically generated

Finally, here was a short comparison between most of the methods I described, using 80k events. Following a recurring pattern, the careful Adam algorithms performed the best here. Interestingly, you can see that EVE and the epoch step algorithms did speed up the convergence of Adam when the learning rate is lower.