CptS 530: Final Project Report

Colin Greeley, 11567185
Instructor: Dr. Deford

April 29, 2022

Overview

Stochastic gradient descent (SGD) is an iterative method for optimizing the parameters of a differentiable loss function (objective function) from known data points [1]. The idea behind SGD is to randomly sub-sample instances from a dataset where the gradient of the the loss function generated from the sub-sample will loosely represent the gradient of the entire dataset, showing the geometric direction of error at each iteration. The algorithm is called gradient descent because the gradient of the loss function is used to descend down the hyper-plane of the loss function, also referred to as the *loss landscape* [2], with the ultimate goal of finding the global minimum of the objective. By making gradient updates for each batch sample, an objective function can be approximated in relatively short time and for any complexity of function. The recent artificial intelligence revolution would not be possible without SGD since it is the core algorithm used to optimize the parameters of all deep neural networks, where neural networks are also known as universal function approximation models, due to their flexibility and scalability. SGD is effectively the process of how artificial neural networks learn, which is one of the many reasons why the algorithm is so exciting.

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \nabla_w L(w; \hat{y}, y)^{(t)}$$

The equation above is the update method for SGD. w is the parameters (weights) of the differentiable function $f(w; x)$ that is used to model the problem being solved. $\nabla_w L(w; \hat{y}, y)$ is the gradient of the loss function w.r.t the parameters of the function. The loss function is any differentiable distance measuring function to compute the difference between true outputs y and predicted outputs $\hat{y}$, where $\hat{y}$ is produced from $\hat{y} = f(w; x)$. t is the time-step, and η is the step size of the update, also known as the learning rate.

$$w^{(t+1)} \leftarrow w^{(t)} - \frac{\eta}{n} \sum_{i=1}^{n} \nabla_w L_i(w; \hat{y}, y)^{(t)}$$

The equation above is the mini-batch variant of SGD. For all programming implementations, mini-batch SGD is used for smoother gradient updates. The average gradient across many samples makes the process less stochastic.

While SGD is fantastic for quickly descending down the loss landscape of an objective function, it can struggle with parameter optimization in high-dimensional functions and lead to "exploding gradients" in functions of high polynomial degree. Because of this, sub-variants of the famous algorithm have been created to work out the flaws of vanilla SGD. Adaptive Moment Estimation (Adam) [3] is one of the most popular sub-variants of SGD due to its ability to efficiently work with a large number of parameters as well as finding the global minimum in high-dimensional loss landscapes. Default SGD has the problem of getting stuck in local minimas and saddle points of high-dimensional loss functions. This is what inspired momentum in the gradient descent algorithm. The intuition behind Adam is that the current location in the loss landscape should not be treated as just a vector pointing downhill, but more as a "heavy ball" will momentum rolling downhill [3]. This way the local minima and saddle point problem can be avoided and the algorithm will be able to find itself settling closer to the global minimum than it would with the default SGD while also optimizing the parameters faster. Adam used estimates of the first and second moments of the gradients to adapt the learning rate for each parameter individually. The moments are implemented in Adam by estimating the exponentially moving average of the gradients to accelerate the gradient updates towards convergence.

$$m_w^{(t+1)} = \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L(w;\hat{y},y)^{(t)}$$
$$v_w^{(t+1)} = \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L(w;\hat{y},y)^{(t)})^2$$

m and v are the moving averages of the first and second moments, respectively, and β_1 and β_2 are exponential decay rates of the average gradients.

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t},\ \hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t}$$

Since the moving averages of the algorithm are generally biased, the step to produce $\hat{m}$ and $\hat{v}$ is used to help mitigate bias by decreasing the β decay by t.

$$w^{(t+1)} \leftarrow w^{(t)} - \eta\frac{\hat{m}_w}{\sqrt{\hat{v}_w}+\epsilon}$$

Finally, the parameters at time-step $(t+1)$ can be updated via the modified SGD algorithm.

The creators of Adam optimization claim that part of its effectiveness is in the generality of the optimizer. To test the efficiency and generality of Adam, I implemented the Adam optimization method [3] for stochastic gradient descent across two domains and comparing the results to that of standard stochastic gradient descent. The two domains are perceptron classification and polynomial regression. I think that these two domains are good for testing the generality of Adam optimization since the for former domain contains high dimensionality and low polynomial order while the latter contains low dimensionality and high polynomial order. Additionally, both problems are interesting and fun to solve. The motivation for this ideas comes from my increased interest in gradient-based optimization algorithms, as they have fueled the AI revolution by being able to efficiently optimize the parameters of deep neural networks.

Methods and Datasets

The dataset for the classification problem will be the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/). The dataset is composed of 10 separate classes, (all single digit integers). The database includes 60,000 images for training and 10,000 for evaluation while each image is 28 x 28 pixels with 1 color channel. Since the model being used to classify the images is a perceptron, the only preprocessing is flattening each image to be a 1-dimensional feature vector with 784 values. The perceptron takes the form $y = Wx + b$, where y is the probability vector representing class predictions, W is the weight matrix of tunable parameters, x is the input feature vector, and b is the bias, which is another tunable parameter. The idea with the perceptron is that possible to optimize the parameters of the function in such a way that the solution is a 10-D hyperplane that accurately separates all data points into their respective classes. The problem sound incredibly complex, but in reality surprisingly easy to optimize using gradient descent.

For the polynomial regression algorithm, the dataset being used will be random (x, y) samples from an input function on an interval. I will demonstrate how both SGD and Adam behave on different intervals. The input function can also be any polynomial or non-polynomial function. I will demonstrate how both optimizers perform on approximating the parameters for set

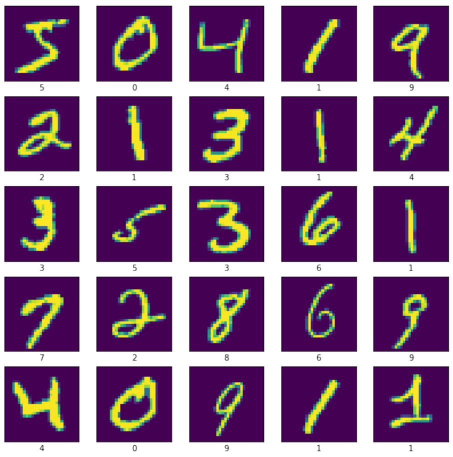

Figure 1: First 25 samples from the training set of MNIST. Each pixel $p \in [0, 1)$.

of diverse functions. The difficultly in testing polynomial regression will be in deciding how to generate random curves to approximate without being biased towards either optimization method.

Stochastic gradient descent works by applying gradient updates to equation parameters though computing the errors between predicted values and true values. For the perceptron classifier, entropy loss is used as the metric to evaluate the performance of the classifier. For the polynomial regressor, mean squared error loss is used as the metric to evaluate the performance of the regressor. Gradients for the optimizers will also be generated using these loss functions, respectively. For reporting the evaluation performance of the perceptron, the optimisation programs will log the evaluation loss along with the evaluation accuracy at each iteration of optimization (performance on test set). Evaluating the performance of the polynomial regressor will involve showing how quickly the error converges during parameter approximation between the two optimization methods as well as showing how much error the approximated curve has after convergence.

$$L_P = -\sum_{i=1}^{n} y_i \log(\hat{y}_i),\ L_R = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Loss functions for each problem type. L_P is multi-class cross entropy loss for the perceptron learning algorithm. L_R is mean squared error loss for the polynomial regression algorithm. $y \in [0, 1]$, $\hat{y} \in [0, 1)$.

Numerical Analysis

The following subsections show the experiments that were carried out in order to test the generality hypothesis of Adam optimization versus standard SGD optimization.

Perceptron Classification

The fist experiment demonstrated below is used for testing the performance difference between Adam and SGD for perceptron classification. For each method, the maximum number of iterations is 100, but the model was forced to stop if the test loss reached convergence, i.e., if it stopped improving after 5 iterations. Additionally, a batch size of 1,000 was each for each method. Through empirical exploration, it was discovered that $\eta = 0.1$ for SGD and $\eta = 0.001$ for Adam resulted int the best performance for each method. The initial value for all trainable parameters of the perceptron is zero.

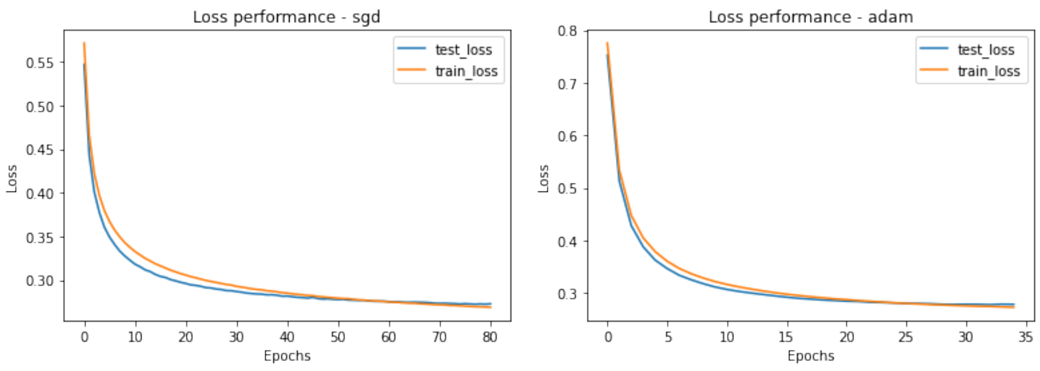

Figure 2: Train set and test set loss performance curves. Each point on the curve is an instance of batch entropy loss computed at the given epoch of training. SGD converged after 81 iterations while Adam converged after only 35 iterations

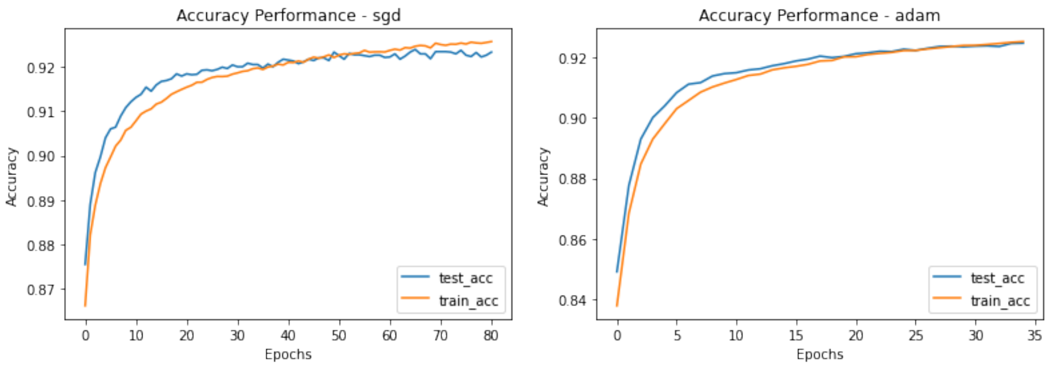

Figure 3: Train set and test set accuracy performance curves. Each point on the curve is an instance of batch classification accuracy computed at the given epoch of training

Perceptron results	SGD	Adam
Train loss	0.2724	**0.2692**
Test loss	**0.2740**	0.2784
Area Under Loss Curve	23.923	**11.22**
Convergence Iteration	81	**35**
Train Acc	0.9246	**0.9267**
Test Acc	0.9223	**0.9244**

Figure 4: Train and test results for both optimization algorithms.

From figures 1-3, it is apparent that Adam is the superior optimization algorithm for training this instance of a perceptron classifier. At convergence, both optimizers ended up with very similar loss and accuracy results, but the big difference is how much faster Adam was able to achieve convergence. The rate of convergence is also quantified in the area under the loss curve calculation.

Polynomial Regression

The second set of experiments shown below are setup with the continued verifying the hypothesis that Adam optimization is more effective and general than SGD optimization. The following experiments are all polynomial regression problems, where a function is defined and random points are sampled on an interval to be used as points for the optimization algorithm to use to learn the parameters of an approximation algorithm to interpolate the defined function. For all the following experiments, the interval for polynomial regression is $[-1, 1]$, since large input values lead to exploding gradients in SGD, especially for high order polynomial equations. It should be noted that Adam is not affected by this nearly as severely. In fact, in any polynomial regression experiment that I setup where the interval was outside the bounds of $[-1, 1]$, optimization from Adam never led to exploding gradients whereas SGD would for high order polynomials. Through empirical exploration, it was discovered that $\eta = 0.1$ for SGD and $\eta = 1$ for Adam resulted int the best performance for each method. I still cannot figure out why Adam needs such a high learning rate for polynomial regression but it handles large gradients very well. Next, the initial value for all trainable parameters is zero. Finally, weight averaging was used for all evaluations and curve approximations, since the trainable parameters in polynomial regression are much more sensitive to new data points are are likely to "forget" parameter approximations that have already been made. The averaged weights are running average of each individual parameter across all opimization iterations, even the poor performing iterations. This method is extremely effective for reducing stochasisity in iterative evaluations and furthermore getting the full use out of every training sample.

For the fist polynomial regression experiment, I created a relatively simple objective function to see how well the the optimizers perform in low polynomial environments. The objective function is defined as follows: $f(x) = 4x^4 - 8x^3 + 5x^2 + 3x - 12$. The batch size for this experiment is set to 1 to keep things simple. Each approximation was trained for 1,000 iterations.

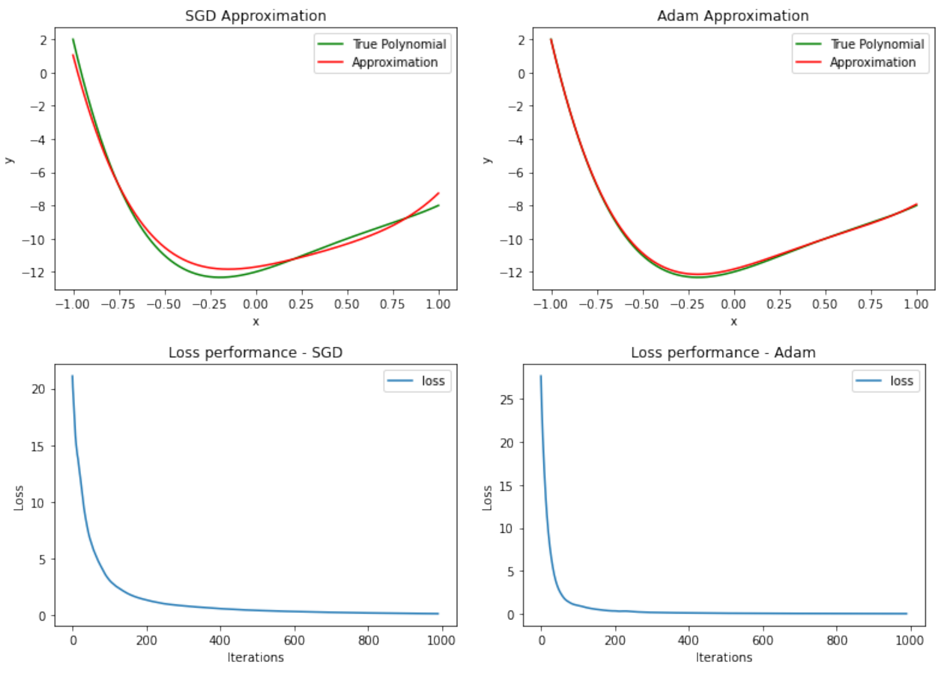

Figure 5: Polynomial Regression with SGD and Adam optimization for low order polynomial interpolation.

The next experiment is setup to test the approximation performance of the optimizers on high polynomial equations. The objective function is generated by randomly choosing the order of the polynomial between 10 and 100. The corresponding coefficient for each polynomial is also randomly generated from a uniform distribution on the interval $[-10, 10]$. These intervals are designed to be random but controlled, to show that the results are reproducible for any generated polynomial equation. These parameters can be tweaked in the included jupyter notebook for polynomial regression. The regressor has the same polynomial order as the randomly generated objective function. The batch size for both methods is 100.

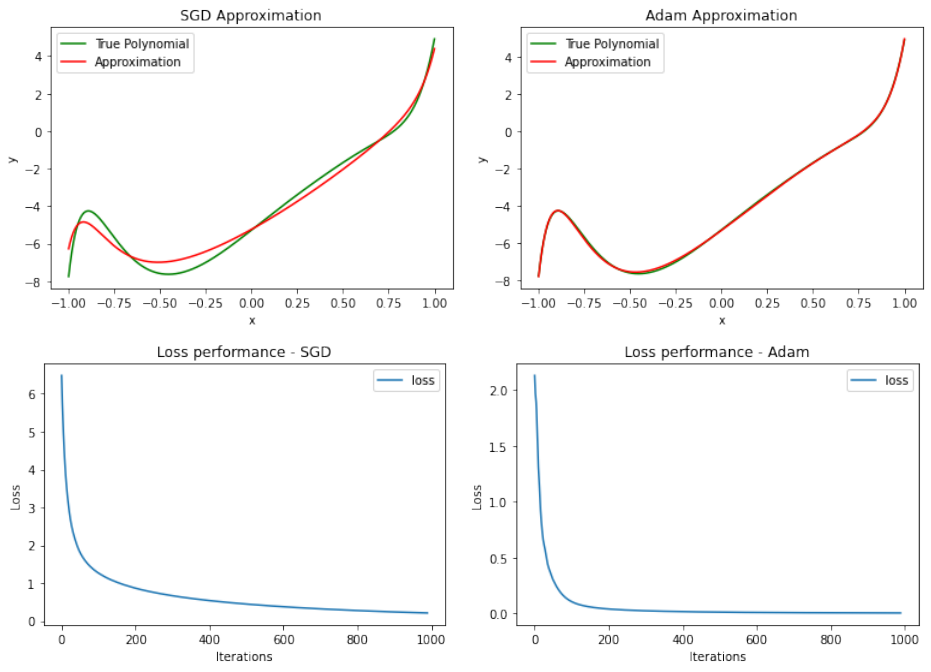

Figure 6: Polynomial Regression with SGD and Adam optimization for high order polynomial interpolation.

The next experiment is the same as the last, except for the regressor polynomial order, which is set to 10. The purpose of this experiment is to see how the regressor learns to approximate functions that are a much higher polynomial order than itself, and how well each optimizer does in optimizing the parameters. Again, the batch size for both methods is 100.

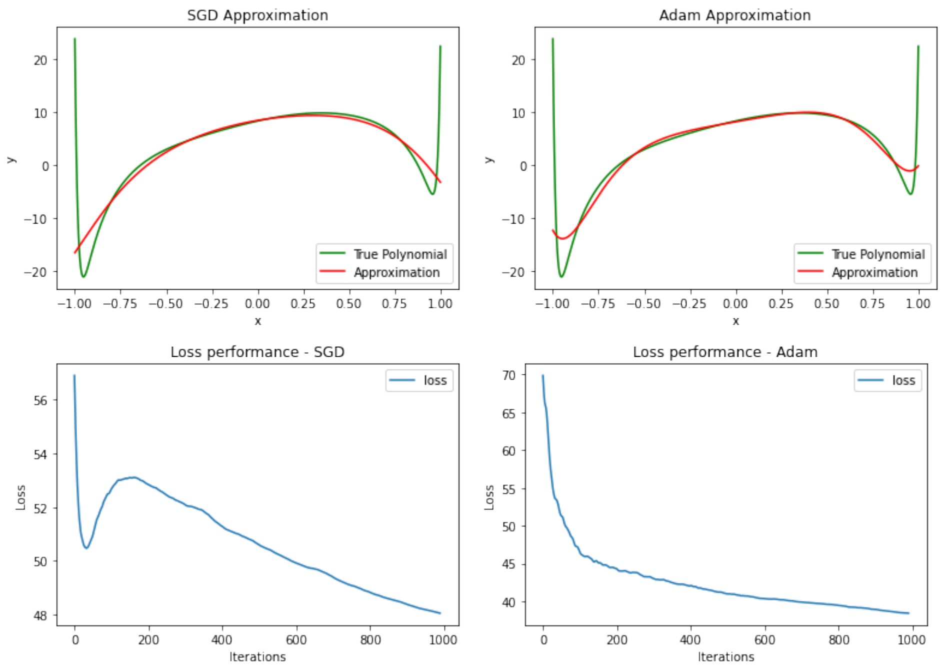

Figure 7: Polynomial Regression with SGD and Adam optimization for high order polynomial interpolation.

The final experiment is setup to test how well the polynomial regressor performs on interpolating non-polynomial functions. In this case, the objective function is $f(x) = \sin(10x)$ on the interval $[-1, 1]$. The polynomial order of the regressor is set to 20, but the order could easily by modified for better results. The batch size for this experiment is set to 1,000 and the models run for a total of 10,000 iterations.

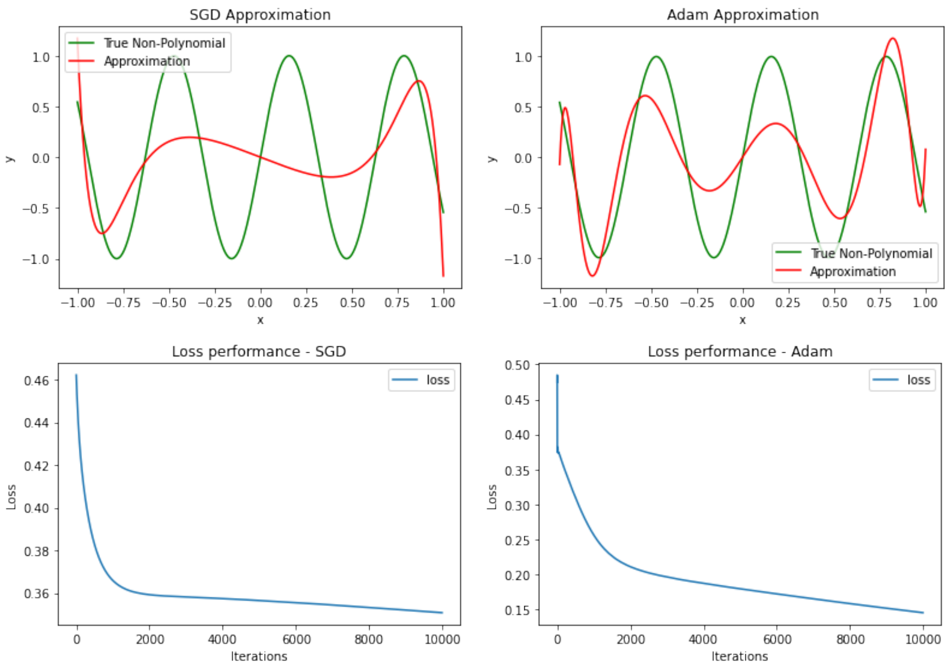

Figure 8: Polynomial Regression with SGD and Adam optimization for non-polynomial interpolation.

Polynomial Regression Results	SGD	Adam
Experiment 1		
Area Under Loss Curve	1379.76	**693.18**
Final MSE loss	0.1534	**0.0112**
Experiment 2		
Area Under Loss Curve	668.05	**66.94**
Final MSE loss	0.2101	**0.0033**
Experiment 3		
Area Under Loss Curve	50096.74	**42078.31**
Final MSE loss	48.0592	**38.4268**
Experiment 4		
Area Under Loss Curve	3591.04	**1926.45**
Final MSE loss	0.3508	**0.1452**

Figure 9: Loss metrics for all polynomial regression experiments. Area under the loss curve represents the relative rate of convergence.

Again, Adam significantly outperforms SGD in the polynomial regression setting for all experiments. One observation might be that Adam has an unfair advantage in this setting since the learning rate is 1 while the learning rate of SGD is 0.1. Every experiment I tried with a learning rate of 1 for SGD resulted in exploding gradients and the loss quickly diverging towards infinity. Adam optimization is simply more robust and can handle large gradient updates better than SGD.

Conclusion

All the results from both the perceptron classification and polynomial regression experiments clearly support the claim that the Adam optimization method is more effective that standard SGD across multiple domains. While both optimization methods required η-tuning to achieve their best performance, Adam was able to handle a wide range a learning rates and sill perform well. The complexity of both methods is effectively the same, since the gradient update computation for any differentiable function is cheap. Adam just requires two extra steps where the only operations involved are multiplication, division, and a square root. All the experiments carried out had slight bias since the hyper-parameters for the optimization were tuned, i.e., the learning rate and batch size. It would be interesting to continue this experiment while comparing different learning rate and batch size values for a large set of random controlled experiments. The polynomial regression experiments could definitely be improved by being more controlled, or simply ran many times with the random equation generation and then reported averaged results. There are many values within the polynomial regression experiments that are more or less arbitrarily chosen. Regardless of how the experiments were carried out, the results are still fascinating. Optimization via gradient update methods is one of my favorite algorithmic domains simply due to the scalability and power of the methods, so long as there is data available to learn from. It is becoming increasingly more popular in today's world since data has become so abundant. All that is required for the method to be effectively applied is a sufficiently large set of data correlated to a problem, and a differentiable function that can be used to describe that problem.

References

[1] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[2] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets, 2017.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.