Stochastic Gradient Descent for Deep Learning

Anubhav Ashok
anubhava

Abhinav Garlapati
agarlapa

Abstract

Deep learning has become very popular in recent times. Deep learning has been applied to almost every field of research successfully. Given the vast impact of deep learning, it is very important to understand why it works well. We believe that powerful optimization techniques such as gradient descent lie at the heart of deep learning. In this project, we study the behaviour of variants of gradient descent and then propose a heuristic which utilizes multiple learning rates in order to get out of badly conditioned areas on the surface of the loss function like local minima, saddle points and valleys.

1 Introduction

Deep learning has become very popular in recent times. The amount of attention it has received is justified by the impressive results it has produced on various tasks. For example, convolutional neural networks which are a class of deep networks are the state-of-the-art in many computer vision tasks. Furthermore, deep networks can be generalized across datasets. A deep network that is trained on one dataset can be used to initialize another, which is then trained on a different dataset for a different task via a process called fine-tuning. Another attractive feature is that they are able to work directly on the input data like images and do not need any external feature engineering.

While their potential is vast, the primary limiting factor is the training procedure. Deep learning models have thousands of parameters which have to be learnt from a given dataset. These parameters are learnt by minimizing a predefined loss function. This learning problem is a very difficult task in practice. Despite being first introduced in 1998 they did not gain popularity till 2011. During this period various developments in gradient descent, back propagation, availability of large datasets and GPUs have lead to success of deep learning.[6], [1].

Several variants of gradient descent have been proposed in the past few years, each addressing various issues faced while training large models. Vanilla gradient descent works well with small networks, but fails to extend to larger networks with many hidden layers. In this project we aim to understand the behaviour of these variants of gradient descent on the training procedure. We compare the most popular methods, namely AdaGrad, RMSprop and ADAM and study their convergence behaviour. We noticed that the pathological curvature of the loss functions introduces various challenges like large number of local minima, saddle points, long narrow valleys and regions of both high and low curvature. We also discuss the pros and cons of moving towards second order methods of optimization.

In this report we use convolutional neural networks as the deep learning model to test our hypothesis because of their relevance to our field of interest Computer Vision. We believe that this kind of analysis would yeild similar results over any other deep learning model. We first introduce these networks in Section 2 and then discuss the various gradient descent algorithms.

2 Background

In this section, we briefly cover the architecture of deep networks and why gradient descent plays an important role in this field.

2.1 Neural Networks

Neural Networks are essentially function approximation models which are inspired by the functioning of the neuron in a human brain. Like the brain these models are structured into various layers, where each layer is made of small units called neurons. Each neuron looks at the responses of all the previous layers and computes the weighted sum. The weights are the parameters which are to be learnt from a given dataset. A small example network is shown in the fig 2.1. Each layer l is represented by the weights W_l, b_l and performs the following computation shown in equation 1.

$$f_{l+1}(x) = g(W_l f_l(x) + b_l) \tag{1}$$

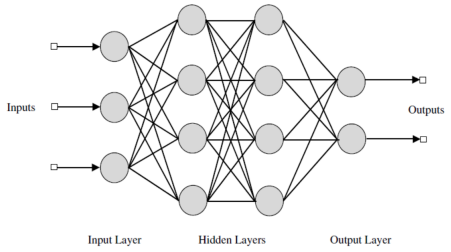

Figure 1: Feed forward network with two hidden layers

2.2 Training

The process of learning the weights from the data is known as training. The weights are learnt by minimizing a loss function. For a given dataset X we search for the best set of parameters which minimize the loss over the dataset. This is essentially an unconstrained optimization problem as shown in the equation 2. $\mathcal{L}$ is a loss function, which computes the loss between the true labels Y and the values predicted by the function f.

$$W^* = \arg\min_W \mathcal{L}(f, X, Y) \tag{2}$$

Gradient descent is one of the most popular method used to solve such optimization problems and is the most common algorithm used for training neural networks. The main idea of gradient descent is to update the parameters in the opposite direction of the gradient as shown in the equation 3.

$$W = W - \eta \nabla_W \mathcal{L}(f, X, Y) \tag{3}$$

There are three variants of gradient descent,

- Batch gradient descent - This is the vanilla gradient descent, where the gradient of the loss function is estimated by looking at all the entire dataset at once. For each update, the gradient has to be computed again and again. This makes the algorithm very slow and is intractable for large datasets which do not fit in the memory.
- Stochastic gradient descent - SGD is a variant of the vanilla algorithm where only one data point is used to estimate the gradient. Since it does not requires the full dataset for every iteration, it can be used in an online setting and is faster. A feature of SGD is that it is fluctuates across iterations, which prevents it from getting stuck in local minima. SGD typically converges faster than batch gradient descent. The SGD update equation is shown described in 4.

$$W = W - \eta \nabla_W \mathcal{L}(f, x^{(i)}, y^{(i)}) \tag{4}$$

- Mini-batch stochastic gradient descent - This method takes the best of the previous two approaches. In this method, we use a small subset of the dataset to compute the gradient. For each mini-batch of n points, we have the following update equation

$$W = W - \eta \nabla_W \mathcal{L}(f, x^{(i:i+n)}, y^{(i:i+n)}) \tag{5}$$

This reduces the variances in each update of the parameter and is computationally feasible. This is the most commonly used approach in deep learning right now. We use this method and experiment to understand its behaviour.

3 Practical Considerations

The networks which we deal with in practice have parameters in the order of millions and range from 6 to 150 layers. The loss functions which arise from these networks are highly non-convex, and a very challenging function to optimize. The SGD methods tend to slow down way ahead of convergence, which is believed to be because of local minima. Dauphin et al. [2] argue that this also could be because of it being stuck on saddle points. They argue that it highly unlikely that loss function would have an increasing gradient in all million directions. Another issue is that the learning rate should be specific to the parameters. Updating all parameters with the same error rate can cause problems in cases where the data is sparse and different features have different frequencies. Ideally we would like to update the parameters with low frequency with higher learning rates. There are various variants proposed update equations which try to fix some of these issues. We discuss few of these methods,

3.1 Adagrad

AdaGrad proposed in [3] tries to adapt the learning rate to the parameters. It performs larger updates for infrequent and smaller updates for frequent parameters. This makes it ideal for sparse data. In this method, each gradient is divided by the accumulated squares of the gradients.

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{6}$$

The equation 6 shows the update performed in each iteration by AdaGrad. G_t is the diagonal matrix with i, ith element is the sum of the gradients with respect to i parameter until time t, g_t is the gradient vector of the loss with respect to to all the parameters at time t. ϵ is added to avoid division by zeros. The advantages of AdaGrad is that it tunes the learning rate on its own and does not require a separate learning rate decay schedule. The problem with this method is that gradients tend to accumulate to large values causing the gradients to diminish.

3.2 RMSprop

RMSprop is a variant proposed by Geoff Hinton in a Coursera Lecture, which tries to fix the problems of diminishing learning rates in AdaGrad. RMSprop replaces the sum of the squares of gradients with a decaying average.

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\ W_{t+1} &= W_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{7}$$

In equation 7, γ determines the decay of the average.

3.3 Adam

Adaptive Moment Estimation(Adam)[5] is another method that computes adaptive learning rates for each parameter. This method builds on the idea of RMSprop and along with decaying square of the

gradient, it also uses a decaying average of the past gradients like momentum.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2} \\
W_{t+1} &= W_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\end{aligned}
\tag{8}
$$

m_t, v_t are biased towards the initial m and v which are set to zero in general. To prevent this, we normalize and generate $\hat{m}, \hat{v}$ as shown in the above equation. Now the update is done using these unbiased statistics.

4 Proposed Method

Our proposed heuristic takes as input a base learning rate and generates a high and low learning rate by scaling it by some factor $c \leq 10$. On each iteration, it samples the number of steps (run lengths) for each learning rate according to some probability mass function f. Then, it performs stochastic gradient descent and updates the new position with the one that has the lowest loss. The pseudo code is displayed below. In the following, we define one iteration to be one pass through the while loop and one step to be a single update in the regular SGD algorithm.

Algorithm 1 Our heuristic

$x \leftarrow$ rand() {Initialize position and learning rates}
$\alpha_{\text{low}} \leftarrow \alpha_{\text{base}}/c$
$\alpha_{\text{high}} \leftarrow \alpha_{\text{base}} * c$
while NOT CONVERGED **do**
  $n_1 \leftarrow f(n, \text{'low'})$ {Sample run lengths uniformly}
  $n_2 \leftarrow f(n, \text{'base'})$
  $n_3 \leftarrow f(n, \text{'high'})$
  $x_1 \leftarrow \text{sgd}(x, n_1, \alpha_{\text{low}})$ {Perform steps}
  $x_2 \leftarrow \text{sgd}(x, n_2, \alpha_{\text{base}})$
  $x_3 \leftarrow \text{sgd}(x, n_3, \alpha_{\text{high}})$
  $i \leftarrow \text{argmin loss}(x_i)$ {Update position and iteration number}
  $x = x_i$
  $n = n + n_i$
end while

4.1 Run length sampling

Ideally we want to pick a run length that best achieves our goals of either breaking out of local minima, moving quickly along smooth regions or converging to a minima without oscillating. A few run length sampling schemes are proposed in this section.

4.1.1 Fixed

The simplest approach would be to fix the run lengths to be some constant for each learning rate. One advantage of this approach is that since the number of steps is deterministic, we have greater control over the total number of steps the algorithm should run. This approach however makes simplifying assumptions about the surface we are trying to optimize on.

4.1.2 Sample from fixed distribution

Randomly sampling the run lengths might be a better approach since we have no knowledge of the actual surface generated by the deep network before runtime. Stochastically picking run lengths makes it more likely to pick a good learning rate for a given locality than a fixed value.

4.1.3 Decay

A better method is to sample from a distribution that biases lower learning rates as the number of iterations increases. This method shares the same intuition as decaying the learning rate, i.e. we want lower learning rates as we get closer to convergence.

4.2 Computational complexity

Suppose that stochastic gradient descent converges in n steps. Then in the worst case, our method will perform $3n$ gradient and function computations as opposed to n of them. As shown in the figure below, if we pick the run lengths to be equal, we might have to perform n steps for each learning rate. While this is still linear in time complexity, it seems like it adds substantial overhead to the training time.

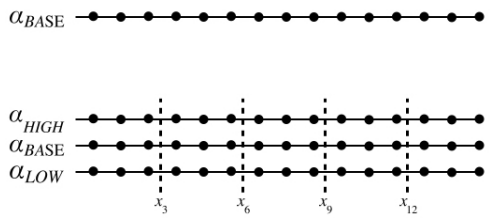

Figure 2: Worst case complexity of GD vs. Our method

Fortunately, as we will see in the practical advantages section the worst case often does not occur in practice.

5 Practical advantages

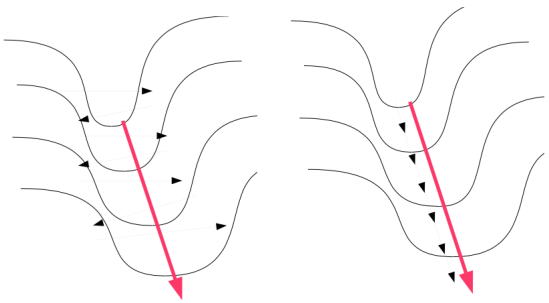

Figure 3: Optimization in long narrow valley(Image source [7])

As discussed earlier, the loss function of deep networks have a large number of local minima and saddle points. These saddle points cause a lot of issues and the optimizers tend to get stuck in them. It is important to be able to adapt the learning rate according to the local curvature of the loss function.

The high learning rate serves two practical purposes. First, to speed convergence by moving quickly in regions of constant gradient and second, to break out of regions containing local minima.

The low learning rate serves the practical purpose of preventing oscillation as we get close to convergence.

One situation where the importance of an adaptive learning rate can be seen in the example shown in the figure 3. This figure shows a long narrow valley, where the base has low curvature and has high curvature along the sides. The small black arrows represent the updates with small and large learning rates respectively. Most first order gradient descent methods tend to oscillate around these curves and tend to stop learning. Our method addresses this by trying both high and low learning rates to progress quickly along the valley while not oscillating along its sides.

While this algorithm seems to have a worse time complexity than regular gradient descent, we observe that in practice, it converges quicker since it moves faster along smoother regions and oscillates less near the optimum. We will now look at theoretical convergence guarantees.

6 Derivation of gradient descent

In order to understand the convergence guarantees of our method, we will first look at those of the regular gradient descent algorithm Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ which is convex and differentiable, gradient descent aims to solve the following optimization problem:

$$x^* = \arg\min_x f(x)$$

where x^* is the optimal value of x that minimizes f.

Gradient descent does this by iteratively choosing a new value for x that is likely to minimize the function by moving along the negative gradient.

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

To see why this is the case, let us consider the value of f at some new point x_{t+1} close to x_t. We do this by performing a quadratic expansion of f around x_{t+1} as follows.

$$f(x_{t+1}) \approx f(x_t) + \nabla f(x_t)^\top (x_{t+1} - x_t) + \frac{1}{2}(x_{t+1} - x_t)^\top \nabla^2 f(x_t)(x_{t+1} - x_t)$$

Since the hessian is hard to compute in practice, we can replace it by $\frac{1}{2\alpha} I$, where α is the learning rate of the algorithm. With the goal being to pick a learning rate at each iteration that best approximates the hessian of the underlying function. Our expression now becomes

$$f(x_{t+1}) \approx f(x_t) + \nabla f(x_t)^\top (x_{t+1} - x_t) + \frac{1}{2\alpha} ||x_{t+1} - x_t||^2$$

Since we want to find the x_{t+1} that minimizes f, we get the following

$$\begin{aligned} x^*_{t+1} &= \arg\min_{x_{t+1}} f(x_t) + \nabla f(x_t)^\top (x_{t+1} - x_t) + \frac{1}{2\alpha} ||x_{t+1} - x_t||^2 \\ &= \arg\min_{x_{t+1}} ||x_{t+1} - (x_t - \alpha \nabla f(x_t))||^2 \end{aligned}$$

Thus it is apparent that the best value for x_{t+1} is $x_t - \alpha \nabla f(x_t)$.

7 Convergence analysis of gradient descent

As before, say $f : \mathbb{R}^n \mapsto \mathbb{R}$ is convex and differentiable. To get convergence bounds, we further assume that f is Lipschitz continuous, i.e.

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y||$$

for any x, y and $L > 0$.

Since the gradient of f is Lipschitz continuous, the largest eigenvalue of hessian of f is bounded by L. In the following, we denote this by the expression $\nabla^2 f \preccurlyeq LI$. Furthermore, since f is convex, its hessian is positive definite. Combining these properties, we get that $LI - \nabla^2 f$ is positive semi-definite, i.e. $\forall x, y, z$

$$(x-y)^\top (LI - \nabla^2 f(z))(x-y) \geq 0$$
$$L||x-y||^2 \geq (x-y)^\top \nabla^2 f(z)(x-y)$$

Using Taylor expansion on $f(y)$ around some point x and the Lagrange form of the remainder, we have $\forall x, y, \exists z \in [x, y]$ such that

$$\begin{aligned} f(y) &= f(x) + \nabla f(x)^\top (y-x) + \frac{1}{2}(x-y)^\top \nabla^2 f(z)(x-y) \\ &\leq f(x) + \nabla f(x)^\top (y-x) + \frac{L}{2}||y-x||^2 \end{aligned}$$

We now use the result of the derivation in section 1, $x_{t+1} = x_t - \alpha \nabla f(x_t)$

$$\begin{aligned} f(x_{t+1}) &\leq f(x_t) + \nabla f(x_t)^\top (x_{t+1} - x_t) + \frac{L}{2}||x_{t+1} - x_t||^2 \\ &= f(x_t) - \nabla f(x_t)^\top (\alpha \nabla f(x_t)) + \frac{L}{2}||\alpha \nabla f(x_t)||^2 \\ &= f(x_t) - (1 - \frac{\alpha L}{2})\alpha ||\nabla f(x_t)||^2 \end{aligned}$$

For $\alpha \leq \frac{1}{L}$, we can write

$$f(x_{t+1}) \leq f(x_t) - \frac{\alpha}{2}||\nabla f(x_t)||^2$$

Since f is convex, $f(x_t) \leq f(x^*) + \nabla f(x_t)^\top (x_t - x^*)$. Combining this with the above, we get

$$\begin{aligned} f(x_{t+1}) &\leq f(x_t) - \frac{\alpha}{2}||\nabla f(x_t)||^2 \\ &\leq f(x^*) + \nabla f(x_t)^\top (x_t - x^*) - \frac{\alpha}{2}||\nabla f(x_t)||^2 \\ &= f(x^*) + \nabla f(x_t)^\top (x_t - x^*) - \frac{\alpha}{2}||\nabla f(x_t)||^2 + \frac{1}{2\alpha}||x_t - x^*||^2 - \frac{1}{2\alpha}||x_t - x^*||^2 \\ &= f(x^*) + \frac{1}{2\alpha}(2\alpha \nabla f(x_t)^\top (x_t - x^*) - \alpha^2 ||\nabla f(x_t)||^2 + ||x_t - x^*||^2 - ||x_t - x^*||^2) \\ &= f(x^*) + \frac{1}{2\alpha}(||x_t - x^*||^2 - (\alpha^2 ||\nabla f(x_t)||^2 - 2\alpha \nabla f(x_t)^\top (x_t - x^*) + ||x_t - x^*||^2)) \\ &= f(x^*) + \frac{1}{2\alpha}(||x_t - x^*||^2 - ||x_t - x^* - \alpha \nabla f(x_t)||^2) \\ &= f(x^*) + \frac{1}{2\alpha}(||x_t - x^*||^2 - ||x_{t+1} - x^*||^2) \\ f(x_{t+1}) - f(x^*) &\leq \frac{1}{2\alpha}(||x_t - x^*||^2 - ||x_{t+1} - x^*||^2) \end{aligned}$$

Summing over all iterations,

$$\begin{aligned} \sum_{i=1}^{k} f(x_i) - f(x^*) &\leq \sum_{i=1}^{k} \frac{1}{2\alpha}(||x_{i-1} - x^*||^2 - ||x_i - x^*||^2) \\ &= \frac{1}{2\alpha}(||x_0 - x^*||^2 - ||x_k - x^*||^2) \\ &\leq \frac{1}{2\alpha}(||x_0 - x^*||^2) \end{aligned}$$

Since we showed above that $f(x_{t+1}) \leq f(x_t) - \frac{\alpha}{2}||\nabla f(x_t)||^2$, i.e. the sequence is non-increasing,

$$\begin{aligned} f(x_k) - f(x^*) &\leq \frac{1}{k}\sum_{i=1}^{k} f(x_i) - f(x^*) \\ &\leq \frac{||x_0 - x^*||^2}{2k\alpha} \end{aligned}$$

Thus, we can see that the gradient descent algorithm converges at a rate of $O(\frac{1}{k})$

8 Convergence guarantees

First, we assume that $\alpha_{\text{BASE}} \leq \frac{1}{L}$ since the standard gradient descent algorithm would theoretically diverge otherwise. Our method differs from gradient descent in that it tries 3 different guesses of the learning rate, compares them, and then picks the next best point. Since $\alpha_{\text{LOW}} < \alpha_{\text{BASE}} \leq \frac{1}{L}$, we are guaranteed that at least 2 of the learning rates meet the constraint. In regions that α_{HIGH} exceeds $\frac{1}{L}$, under our assumptions, it will produce a loss worse than the other two, so we will discard the update. Additionally, at regions where α_{HIGH} is closer to $\frac{1}{L}$, it will produce a better update. Hence, at the worst case, our method also has a convergence rate of at least $O(\frac{1}{k})$.

The learning rate that gives the optimal convergence rate based on our assumptions is $\alpha = \frac{1}{L}$. However, in practice the surface is likely non-convex and it is often hard to know exactly what a suitable value for L is. By trying a few learning rates at each local region, we are more likely to find a learning rate that is close to $\frac{1}{L}$. Thus our method is likely to converge faster.

9 Experiments

In all our experiments we use a Convolutional Neural network implemented using the Caffe framework[4]. We compare our proposed methods with the vanilla SGD update equations, ADAM, AdaGrad and RMSprop. We test our hypothesis on two datasets, MNIST and CIFAR-10.

9.1 MNIST

Figure 4: Sample images from MNIST

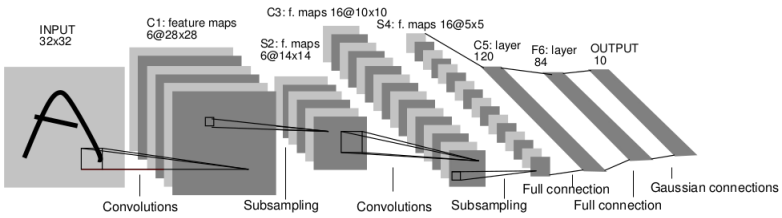

Figure 5: LeNet architecture.

MNIST is a large handwritten digit dataset. It comprises of small binary images of the 10 numeric digits, a sample of which is shown in the Fig 9.1. To classify these images, we use a convolutional neural network architecture known as LeNet[6]. This network shown in the Fig 9.1, has 6 layers, 2 convolution layers, 2 sub-sampling layers and 2 fully connected layers. We train this network with different SGD variants and compare the obtained trajectories. The results of which are shown in the Fig 6. We notice that our method very quickly converges to a lower loss compared to the other methods. Our method is a lot more stable compared to the other methods. It is to be noted that we also have compared our method with RMSprop, but it converges to a loss which is an order larger than these methods, hence has been omitted from the plot.

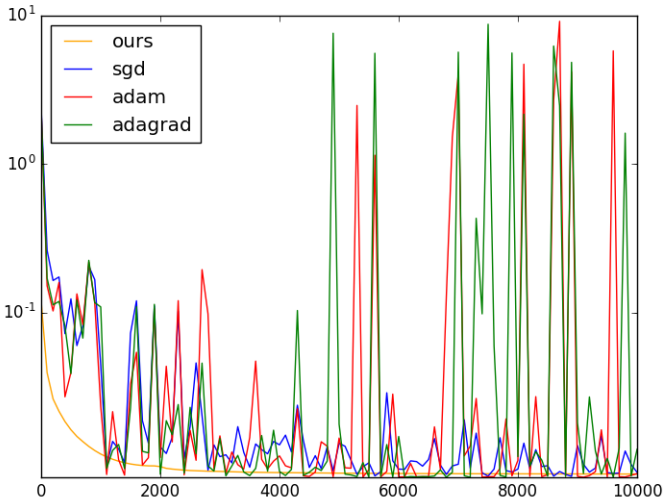

Figure 6: Loss during the training of the LeNet model on MNIST dataset using various SGD methods

9.2 CIFAR

We next use the CIFAR-10 dataset, which has tiny images of different objects as shown in Fig 9.2. The results of this are shown in the Fig 8. We still observe that our method reaches a lower loss much faster than other methods. Other methods take up to 50k iterations to reach the loss reached by our method in 4k iterations. In this experiment, we clearly see the improvement exhibited by our heuristic.

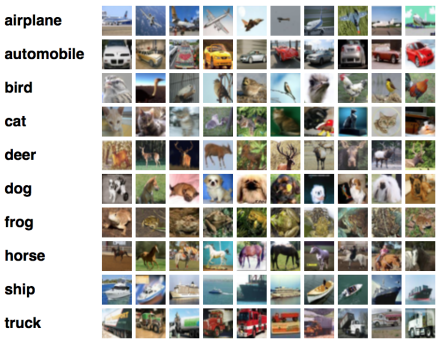

Figure 7: CIFAR-10 sample images

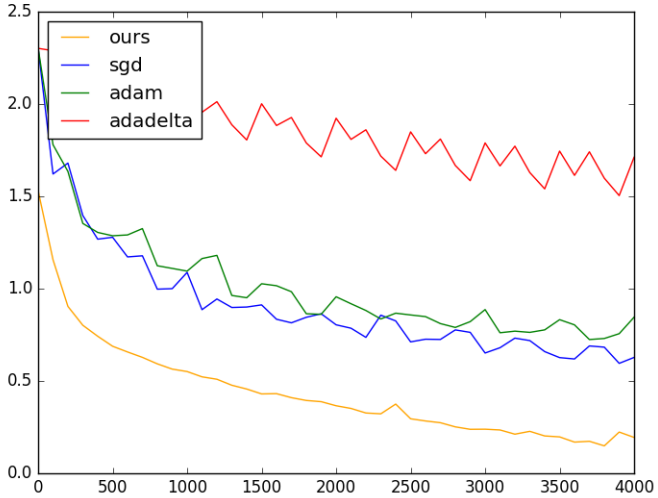

Figure 8: Loss during the training on CIFAR-10 dataset using various SGD methods.

10 Conclusion

In this project, we first studied the various gradient descent techniques popularly used for learning in deep learning. We then analysed their strengths and weaknesses and propose a heuristic which reaches better convergence faster but at a higher computational cost. Furthermore, we observed that our method produces a better optimization despite letting the other methods run for an equivalent number of steps. In these experiments, we applied our heuristic to the vanilla SGD method. However, we expect to obtain superior results when used in combination with stronger optimizers like ADAM and AdaGrad.

References

[1] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[2] Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[5] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.