A visual analysis of optimization algorithms

Andrea Cerutti, 385030
EPFL, Switzerland

Abstract—**In this project, a methodology to obtain a visual representation of loss functions is presented. These representations allow for a comparison between the behaviors of various optimization algorithms in real-world scenarios rather than just simple problems where the function to approximate is fully known. After defining this methodology, classical gradient descent (GD), stochastic gradient descent (SGD), and Adaptive Moment Estimation (Adam) are compared in different learning rate and batch size scenarios to highlight their differences.**

I. Introduction

Optimization algorithms are crucial in training neural networks, significantly impacting their performance and generalization capabilities. This report aims to analyze the shape of local minima found by different optimization algorithms in a real-world optimization task. The challenge is to find a suitable representation of the loss function, which is non-trivial when the loss is multi-dimensional as well as the target function. The chosen method to achieve this representation is presented in [1]. To further analyze and compare the different algorithms, the same method with appropriate corrections was used to produce an approximation of the path followed by the model during training, allowing easier visualization of the effects of different learning steps, batch sizes, and momentum.

II. Methods

A. Explanation of terms

Filter-wise normalization: This technique adjusts each filter in a random direction vector to match the norm of the corresponding filter in the network parameters. This normalization helps ensure that the sharpness of the visualized loss surface correlates better with the generalization error, preventing misleading artifacts due to scale invariance in network weights.

Convexity: A function is convex if the line segment between any two points on the function's graph lies above or on the graph. In optimization, convex functions are significant because they ensure that any local minimum is also a global minimum, simplifying the optimization process. **Smoothness**: This refers to the continuity and differentiability of a function. A smooth loss function allows for more reliable and stable gradient-based optimization since small changes in parameters lead to small changes in the loss.

B. Visualization methodologies

The baseline of this work relies on the study presented in [1], where neural network loss surfaces are plot in a 2D space defined by random direction vectors. To avoid misleading artifacts due to scale invariance in network weights, they introduce a "filter-wise normalization" where each filter in a random direction vector d is scaled to match the norm of the corresponding filter in the network parameters θ. This ensures that the visualized sharpness of the loss surface correlates better with generalization error. The 2D plots are generated using the formula $f(\alpha, \beta) = L(\theta^* + \alpha v + \beta\eta)$, where v and η are normalized direction vectors. In short the loss is represented as a function of the distance from the model along v and η. In this project, this method is extended to also represent the training steps performed during training. To obtain this representation, the model is first trained on the dataset to obtain the final trained model. Then, a new model with the same initialization undergoes equivalent training where at each step, the position of the current model with respect to the final one is computed in terms of α and β, effectively obtaining the training trajectory in these coordinates.

C. Dataset and Model

For the objective of the project was important to have a dataset representative of a complex tasks that can still be approached with limited computational resources. A suitable compromise was found by using the CIFAR-10 dataset [2]. This dataset present an image classification task over 10 classes and is usually tackled using convolutional neural networks (CNNs); however to focus the results on the optimizer a simpler multi layer perceptron (MLP) was used. The implemented MLP is composed by four linear layers of sizes 3072x64, 64x64 and 64x32 and 32x10. Between each layer a ReLU activation function was used. This simple model was able to achieve accuracy close to 50% on the test split of the dataset, demonstrating the learning capabilities provided by the optimization algorithm.

D. Study methodologies

Once the visualization techniques, datasets, and models were defined, the next step was to define which algorithms and hyper-parameters to compare. The chosen optimizers were classical GD, SGD, and Adam. Specifically, the results were obtained using PyTorch's implementation (torch.optim.SGD and torch.optim.Adam), where GD was achieved by setting the batch size equal to the entire dataset in SGD. Further studies on the effects of batch size, learning rate, and momentum were performed on the SGD implementation. All experiments on Adam and SGD were conducted over 20 epochs, while GD was limited to 1000 steps. An early stopping mechanism was employed to halt training when the loss reached NaN or did not decrease for four consecutive epochs.

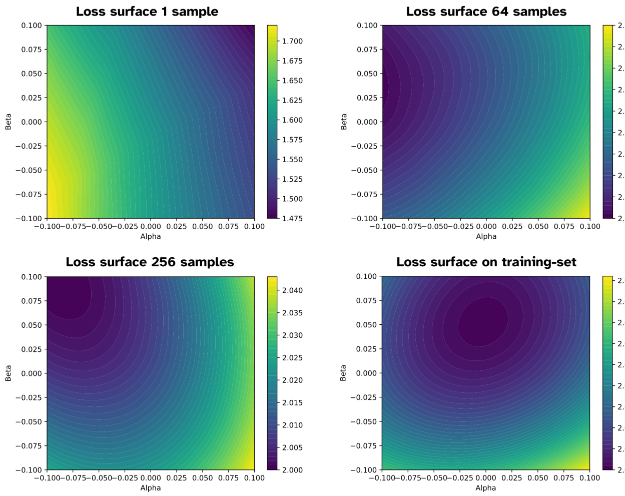

Fig. 1. Local loss representation on the first batch of size 1 (upper left), 64 (upper right), 256 (lower left) and the real loss on the whole dataset (lower right)

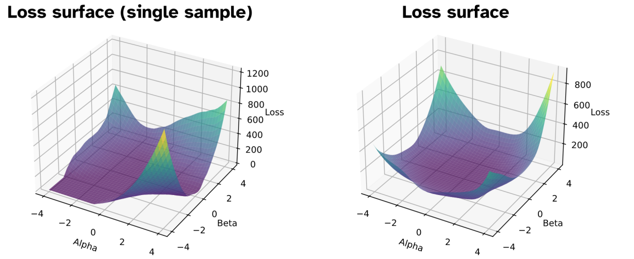

Fig. 2. loss representation for a single batch on the left and for the whole training set on the right. Images obtained at the end of a training with SGD.

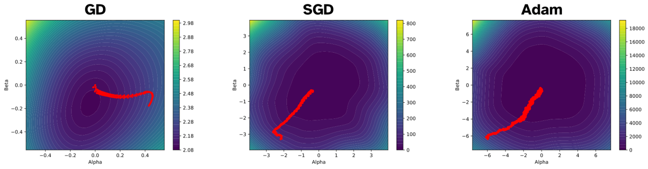

Fig. 3. Comparison of loss function and training of GD (lr=0.1), SGD (lr=2e-3, bs=64, m=0.9), and Adam (lr=5e-4) achieved accuracy

III. Results

A. Loss representation

Using the method described in [1] we where able to obtain a representation of the loss as a function of changes in model parameters along α and β directions. The implementation used in this project allows for the calculation of loss for a given sample or a batch, effectively allowing portrayal of the loss approximation during optimization with different batch sizes. Exploiting this property, Figure 1 illustrates that the loss in the proximity of the model depends on the sample batch used to calculate it. When the batch is small, the gradient direction may significantly differ from the actual one, but with increasing batch sizes, the loss quickly becomes similar to the actual loss. In particular the gradient (calculated in the center of the image) points toward the right for the single batch, towards the upper left for batch sizes of 64 and 256, while points up for the real loss function.

From figure 1, it is clear that the loss calculated with a small batch size differs greatly from the real loss on the whole dataset. This visual proof explains why classical SGD is noisy and why using mini-batches improves performance in practice. Another interesting plot is to obtain a 3D representation of the loss function at the end of training. From this representation we can see if the loss presents multiple minima or how close the representation is to a convex and smooth function Figure 2 shows how the obtained loss function is almost convex and how the loss shape can differ when considering small batches.

B. Optimizer Comparison

Gradient Descent minimizes an objective function $f(\theta)$ by iteratively updating parameters θ in the direction of steepest descent defined by the negative gradient:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t)$$

where η is the learning rate.

Stochastic Gradient Descent is a variant of GD that updates the model parameters using only a single or a small mini-batch of training examples at each iteration. This introduces noise into the parameter updates but can lead to faster convergence and better generalization. Momentum is often used to stabilize convergence:

$$\theta_{t+1} = \theta_t - \eta_t v_t, \quad v_t = \beta v_{t-1} + (1-\beta)\nabla f_i(\theta_t)$$

where η_t is the learning rate, v_t is the velocity, and β is the momentum coefficient.

Adam [3] combines the advantages of momentum and adaptive learning rates. Adam computes adaptive learning rates for each parameter by maintaining exponentially decaying averages of past gradients (m_t) and squared gradients (v_t):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t),$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla f(\theta_t))^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \quad \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where β_1, β_2 control decay rates, and ϵ prevents division by zero.

Table I and figure 3 show the comparison obtained by training over 20 epochs for SGD and Adam, while considering 1000 steps for GD. All trainings were close to convergence.

	GD	**SGD**	**Adam**
Accuracy	38.33%	48.16%	47.65%

TABLE I
Comparison of GD (lr=0.1), SGD (lr=2e-3, bs=64, m=0.9), and Adam (lr=5e-4) achieved accuracy

All optimizers where able to proceed towards a minima allowing the network to learn how to perform the classification task. The obtained graphs in figure 3 loss function is almost

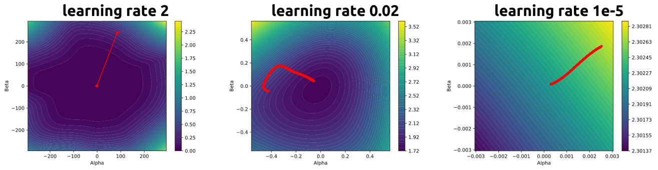

Fig. 4. SGD update steps and losses for varying learning rates, fixed batch size at 64 and no momentum

convex for all optimizers with a single minima. The best performance where obtained using SGD but, we can consider them comparable with Adam results. On the other hand the worst performing algorithm was GD especially when considering the computational burden required to complete 1000 steps. It is important to note that the main difference between these optimizer does not lie on their ability to converge but more on the number of steps required.

C. Hyper-parameters Analysis

The hyper-parameter analysis was completed on modifying SGD parameters. This choice is justified by the simplicity of SGD, the possibility to tune all the the chosen hyper-parameters of interest (batch size, momentum and learning rate). This analysis took into consideration all possible combinations of learning rates of 2, 2e-3 and 1e-5, batch sizes of 1, 64 and 256, and momentum of 0 and 0.9. The summarized results are shown in table II. From these results, it is clear that the most impactful hyper-parameter is the learning rate. When it is too large, the optimization process no longer converges, causing the model to jitter around the minima or even diverge due to overshooting. Conversely, when the learning rate is too small, the training becomes more precise but too slow. This typical behaviour is clearly visible in figure 4 where the oscillations decreases with the learning rate. The effect of choosing higher values for batch size make the training progressively more stable at the cost of more computational power for each step. Choosing higher batch sizes makes training progressively more stable (as justified in loss representation paragraph) at the cost of more computational power for each step. The effects of changing the batch size on the trajectory are reported in 5.

		lr = 2	**lr = 0.002**	**lr = 1e-05**
bs=1	**m=0**	10.00%	27.04%	10.00%
bs=1	**m=0.9**	10.00%	16.44%	11.58%
bs=64	**m=0**	10.00%	34.64%	10.00%
bs=64	**m=0.9**	10.00%	48.16%	15.19%
bs=256	**m=0**	10.00%	17.92%	9.99%
bs=256	**m=0.9**	10.00%	41.34%	10.14%

TABLE II
ACCURACIES OF SGD WITH DIFFERENT PARAMETER COMBINATIONS

Momentum modifies SGD by averaging new updates based on previous ones, mimicking momentum in physics. This helps the model avoid local minima by allowing it to slightly overshoot before changing direction. One visible effect of momentum is smoother training, as shown in figure 6. From the results in table II momentum usually helps the training, however when batch size is too small it may introduce disrupting behaviours.

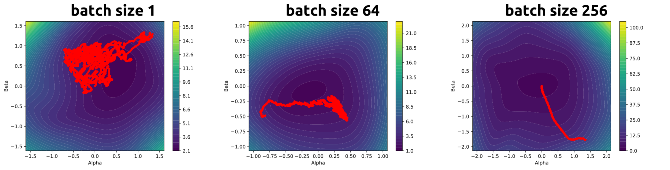

Fig. 5. SGD update steps and losses for varying batch sizes and learning rate of 0.02 and momentum of 0.9.

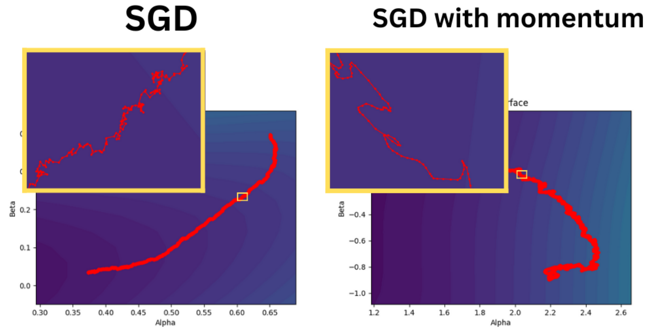

Fig. 6. SGD update steps for batch size of 64 and learning rate of 0.002 with and without momentum.

IV. Limitations

This study primarily focuses on a simple MLP model and the CIFAR-10 dataset, which may limit the generalizability of the results to more complex models and datasets. Future work could involve extending the analysis to convolutional neural networks and other datasets. A further problem with the presented comparison is the difficulty to compare different optimizer in the same conditions by setting correct hyper-parameters, in this project was chosen to present the version of these algorithms that achieved better performances which can be regarded as an arbitrary choice. An in depth analysis would also require the study to cover more values of the hyper-parameters. Another limitation lies in the difficulty of representing loss functions in a low dimensional space, which intrinsically introduce some distortions.

V. Conclusion

This project provided an analysis of loss function representation and used these representations to understand the behavioral changes induced by different optimizers and hyper-parameters. The obtained visuals are almost always in accordance with theoretical expectations, demonstrating the effectiveness of known optimization theory in real-world tasks where typical constraints such as convexity or smoothness are not necessarily respected. Despite the interesting insights on optimization, the scope of this project is still limited. However, it can be easily adapted to analyze different datasets, networks, and optimizers, given that the pipeline is agnostic to these factors.

References

[1] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, “Visualizing the loss landscape of neural nets,” 2018.

[2] A. Krizhevsky, “Learning multiple layers of features from tiny images,” *University of Toronto*, 05 2012.

[3] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.