

Lab CudaVision

Learning Vision Systems on Graphics Cards (MA-INF 4308)

Gradient Descent Optimizers

01.06.2021

PROF. SVEN BEHNKE, ANGEL VILLAR-CORRALES

Contact: villar@ais.uni-bonn.de

Revisiting Gradient Descent

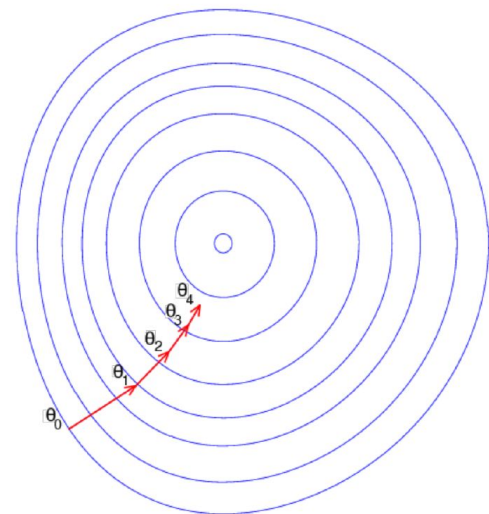
Gradient Descent

- Minimize the empirical risk (L) w.r.t. model parameters (θ)

$$\min_{\theta} \{\mathcal{L}(\mathbf{X}, \mathbf{y}; \theta)\}$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_{\theta} \mathcal{L}(\mathbf{X}, \mathbf{y}; \theta)$$

- Optimize in direction of steepest descent
- Updates proportional to magnitude of the error
- Guaranteed to converge to a (**local**) minimum



Gradient Descent Variants

Batch Gradient Descent

- Gradient of the loss w.r.t. to parameters of the entire dataset all at once

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

- Nice for convex problems with small datasets
- Hardly ever used for Neural Nets:
 - Use all samples for one update
 - Very slow
 - Memory inefficient

Stochastic Gradient Descent (SGD)

- Computes gradient for each training pair $(\mathbf{X}^{(i)}, \mathbf{y}^{(i)})$ independently and makes an update for each pair

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathbf{X}^{(i)}, \mathbf{y}^{(i)})$$

- + Memory efficient
- + Can handle online-learning
- Sensible to outliers
- Loss fluctuates heavily
- Computationally inefficient

Mini-Batch Gradient Descent

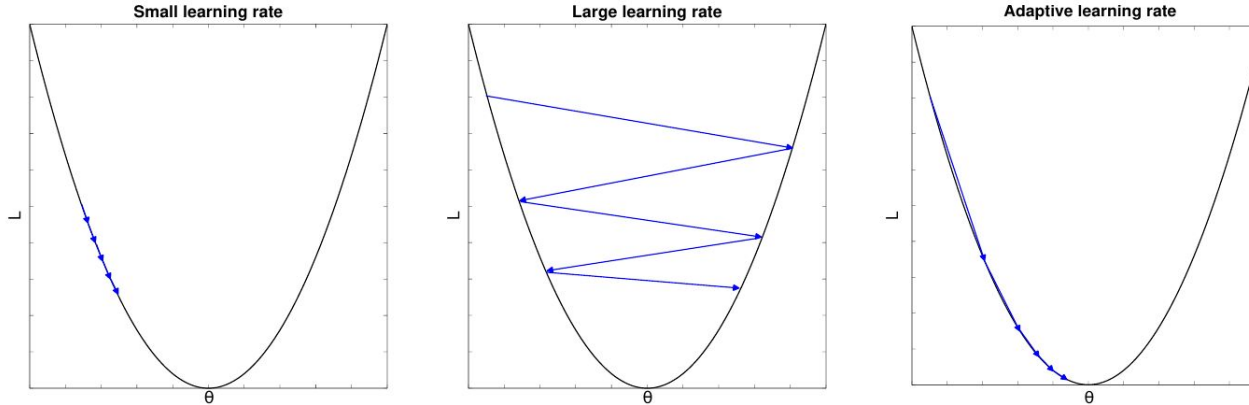
- Performs one update for a subset of the data (*mini-batch*)

$$B^{(i)} = (\mathbf{X}^{(i)}, \mathbf{y}^{(i)}), (\mathbf{X}^{(i+1)}, \mathbf{y}^{(i+1)}), \dots, (\mathbf{X}^{(i+|B|-1)}, \mathbf{y}^{(i+|B|-1)})$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; B^{(i)})$$

- Best of both worlds
 - Fast training
 - Robust to outliers
 - Efficient implementation in GPUs

SGD and Learning Rate



- SGD heavily depends on the choice of hyper-parameters
 - Learning rate
 - Batch size
 - Learning rate decay

Can we do better?

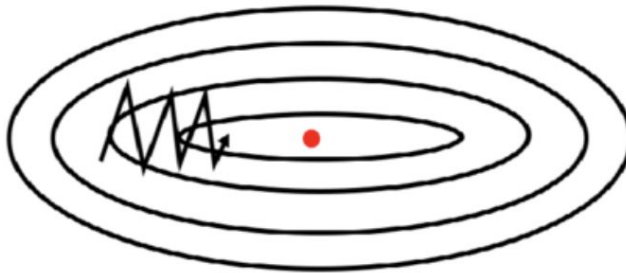


Optimization Algorithms I: Momentum

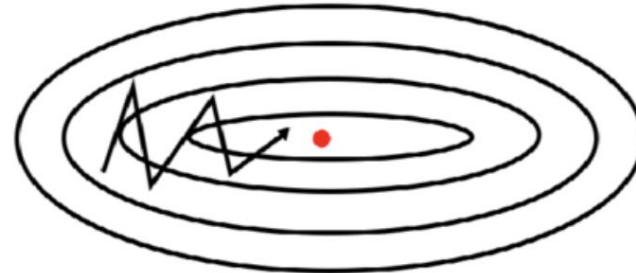
Momentum

- **Problem:** SGD fluctuates when certain dimensions are steeper than others
- **Idea:** Accelerate in directions with persistent gradients

SGD without momentum



SGD with momentum



Momentum

- Parameter updates based on **current and past** gradients

$$\mathbf{v}^{(t)} = \underbrace{\mu}_{\text{momentum}} \cdot \mathbf{v}^{(t-1)} - \eta \cdot \nabla_{\theta} \mathcal{L}(\theta)$$

$$\theta^{(t+1)} = \theta^{(t)} + \mathbf{v}^{(t)}$$

- Typically: $\mu = \{0.9, 0.95, 0.99\}$

 Accelerated convergence

 Learning rate is not adaptive

Nesterov's Momentum

- Measures the error of an estimate of the future parameters

$$\mathbf{v}^{(t)} = \mu \cdot \mathbf{v}^{(t-1)} - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\underbrace{\boldsymbol{\theta} + \mu \cdot \mathbf{v}^{(t-1)}}_{\text{Estimate of next parameters}})$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \mathbf{v}^{(t)}$$

- Takes an anticipatory step that gives us a notion of the loss landscape
- 'Looking ahead' to speed up convergence


Optimization Algorithms II: Adaptive Optimizers

Why Adaptive Optimizers?

- Different features have different needs and frequencies
 - Some are updated very often
 - Some are seldom activated
- We need individual learning rates for each parameter
 - Large η for parameters with infrequent or small gradient updates
 - Small η for parameters with frequent or large gradient updates

Adagrad

- **Adaptive Gradient** optimizer
- Scales each parameter based on past gradients



$$\mathbf{G}_{(t,t)} = \sum_{\tau=1}^t (\nabla_{\theta_{\tau}} \mathcal{L}(\theta_{\tau})) (\nabla_{\theta_{\tau}} \mathcal{L}(\theta_{\tau}))^T$$

Diagonal matrix. Each element is sum of passed squared gradients

$$\theta_{t+1} = \theta_t - \frac{\eta}{\underbrace{\sqrt{\text{diag}(\mathbf{G}_{t,t}) + \epsilon}}_{\text{Each parameter uses a different value to scale the learning rate}}} \odot \nabla_{\theta} \mathcal{L}(\theta_t)$$

Each parameter uses a different value to scale the learning rate

Big Problem!


$t \rightarrow \infty$

$\Rightarrow \mathbf{G}_{(t,t)} \uparrow$

$\Rightarrow \frac{\eta}{\sqrt{\text{diag}(\mathbf{G}_{t,t}) + \epsilon}} \downarrow$

RMSProp

- Solves Adagrad's vanishing learning rate.



$$\mathbf{G}_t = \rho \cdot \mathbf{G}_{t-1} + (1 - \rho) \cdot (\nabla_{\theta_t} \mathcal{L}(\theta_t))(\nabla_{\theta_t} \mathcal{L}(\theta_t))^T$$


Exponential moving average (EMA)

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\text{diag}(\mathbf{G}_{t,t}) + \epsilon}} \odot \nabla_{\theta} \mathcal{L}(\theta_t)$$

- Typically ρ is 0.9
- Scaling factor gives more importance to recent gradients
- Old gradients barely have any importance ($(1-\rho) \rho^{N-1}$)
 - $(1-0.9)0.9^{29} = 0.005$

Adam (almost)

- Combines momentum with parameter-wise adaptive learning rates


$$\begin{aligned}\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \nabla_{\theta_t} \mathcal{L}(\theta_t) \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) (\nabla_{\theta_t} \mathcal{L}(\theta_t) \odot \nabla_{\theta_t} \mathcal{L}(\theta_t))\end{aligned}$$

EMAs of 1st and 2nd order moments

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \cdot \mathbf{m}_t$$

- Typically:
 - $\beta_1 = 0.9$
 - $\beta_2 = 0.999$

Adam

- Moment estimates start at 0 and are computed with a EMA
 - This is a biased estimator!

➡ Let's de-bias the moment estimates

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t + \epsilon}} \cdot \hat{\mathbf{m}}_t$$

Final Thoughts

Too Many Optimizers...

SGD

Momentum

NAG

AdaDelta

AdaGrad

Adam

RMSProp

AdamW

RAdam

NAdam

LBFGS

AdaBelief

AMSBound

AdaBound

SparseAdam

LARS

Adamax



Which Optimizer?

Descending through a Crowded Valley — Benchmarking Deep Learning Optimizers

Robin M. Schmidt^{*1} Frank Schneider^{*1} Philipp Hennig^{1,2}

Abstract

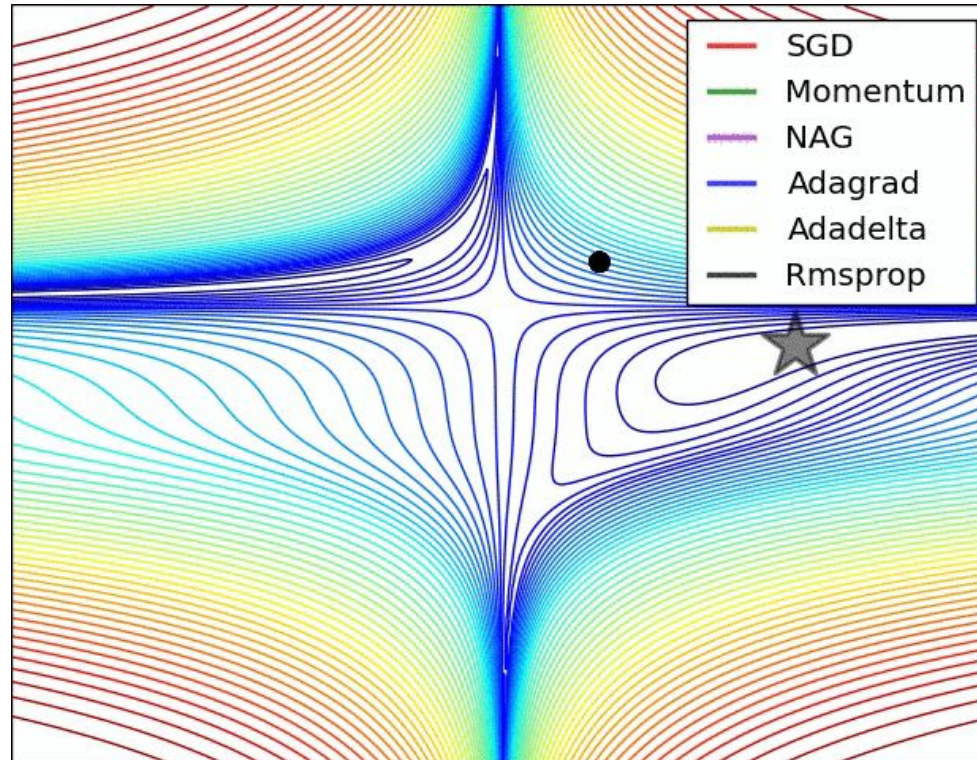
Choosing the optimizer is considered to be among the most crucial design decisions in deep learning, and it is not an easy one. The growing literature now lists hundreds of optimization methods. In the absence of clear theoretical guidance and conclusive empirical evidence, the decision is often made based on anecdotes. In this work, we aim to replace these anecdotes, if not with a conclusive ranking, then at least with evidence-backed heuristics. To do so, we perform an extensive, standardized benchmark of fifteen particularly popular deep learning optimizers while giving a concise overview of the wide range of possible choices. Analyzing more than 50,000 individual runs, we contribute the following three points: (i) Optimizer performance varies greatly across tasks. (ii) We observe that evaluating multiple optimizers with default parameters works approximately as well as tuning the hyperparameters of

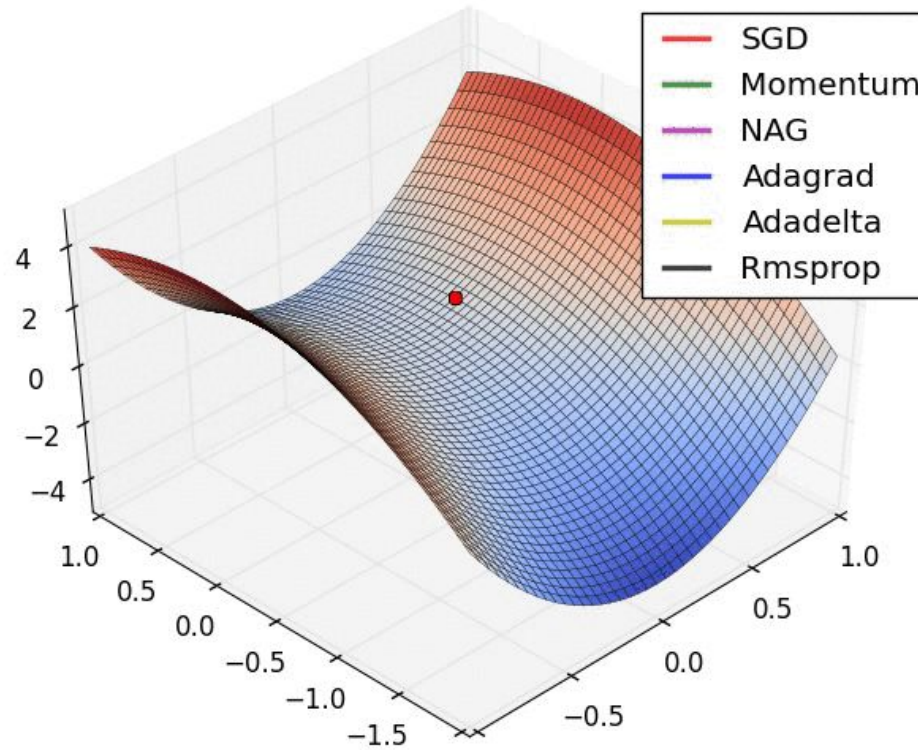
1. Introduction

Large-scale stochastic optimization drives a wide variety of machine learning tasks. Because choosing the right optimization method and effectively tuning its hyperparameters heavily influences the training speed and final performance of the learned model, it is an important, every-day challenge to practitioners. It is probably the task that requires the most time and resources in many applications. Hence, stochastic optimization has been a focal point of research, engendering an ever-growing list of methods (cf. Figure 1), many of them targeted at deep learning. The hypothetical machine learning practitioner who is able to keep up with the literature now has the choice among hundreds of methods (see Table 2 in the appendix), each with their own set of tunable hyperparameters, when deciding how to train a model.

There is limited theoretical analysis that clearly favors one of these choices over the others. Some authors have offered empirical comparisons on comparably small sets of popular methods (e.g. Wilson et al., 2017; Choi et al., 2019; Sivaprasad et al., 2020); but for most optimizers, the only

...we identify a significantly reduced subset of specific optimizers and parameter choices that generally lead to competitive results in our experiments: ADAM remains a strong contender, with newer methods failing to significantly and consistently outperform it

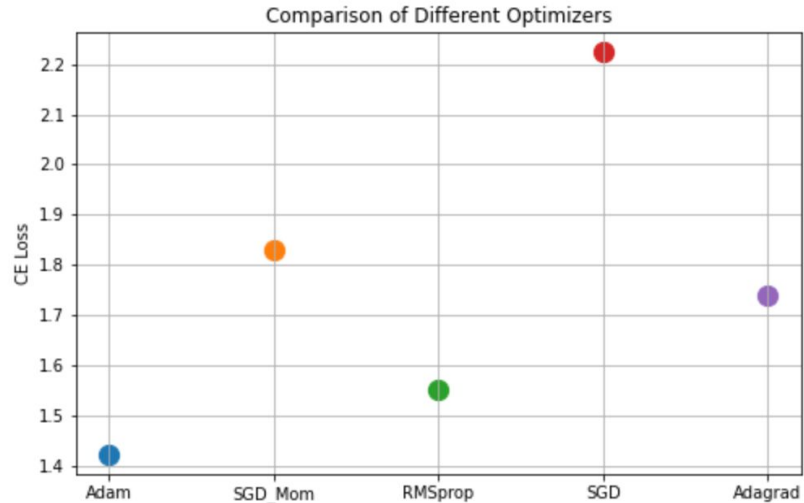




Custom Comparison

- Trained a MLP for CIFAR-10 with different optimizers
 - 10 epochs
 - LR = 0.001
 - Defaults for rest

```
optim_id = trial.suggest_categorical(  
    "optimizer", ["SGD", "SGD_Mom", "Adagrad",  
                  "Adam", "RMSprop"]  
)
```



Recommended Reading

An overview of gradient descent optimization algorithms*

Sebastian Ruder
 Insight Centre for Data Analytics, NUI Galway
 Aylien Ltd., Dublin
ruder.sebastian@gmail.com

Abstract

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This article aims to provide the reader with intuitions with regard to the behaviour of different algorithms that will allow her to put them to use. In the course of this overview, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms, review architectures in a parallel and distributed setting, and investigate additional strategies for optimizing gradient descent.

1 Introduction

On the importance of initialization and momentum in deep learning

Ilya Sutskever¹
 James Martens
 George Dahl
 Geoffrey Hinton

[ILYASU@GOOGLE.COM](mailto:ilyasu@google.com)
[JMARTENS@CS.TORONTO.EDU](mailto:jmartens@cs.toronto.edu)
[GDAHL@CS.TORONTO.EDU](mailto:gdahl@cs.toronto.edu)
[HINTON@CS.TORONTO.EDU](mailto:hinton@cs.toronto.edu)

Abstract

Deep and recurrent neural networks (DNNs and RNNs respectively) are powerful models that were considered to be almost impossible to train using stochastic gradient descent with momentum. In this paper, we show that when stochastic gradient descent with momentum uses a well-designed random initialization and a particular type of slowly increasing schedule for the momentum parameter, it can train both DNNs and RNNs (on datasets with long-term dependencies) to levels of performance that were previously achievable only with Hessian-Free optimization. We find that both the initialization and the momentum are crucial since poorly initialized networks cannot be trained with momentum and well-initialized networks perform markedly worse when the momentum is absent or poorly tuned.

Our success training these models suggests that previous attempts to train deep and recurrent neural networks from random initializations have likely failed due to poor initialization schemes. Furthermore, carefully tuned momentum methods suffice for dealing with the curvature issues in deep and recurrent network training objectives without the need for sophisticated second-order methods.

widespread use until fairly recently. DNNs became the subject of renewed attention following the work of Hinton et al. (2006) who introduced the idea of greedy layerwise pre-training. This approach has since branched into a family of methods (Bengio et al., 2007), all of which train the layers of the DNN in a sequence using an auxiliary objective and then “fine-tune” the entire network with standard optimization methods such as stochastic gradient descent (SGD). More recently, Martens (2010) attracted considerable attention by showing that a type of truncated-Newton method called Hessian-free Optimization (HF) is capable of training DNNs from certain random initializations without the use of pre-training, and can achieve lower errors for the various auto-encoding tasks considered by Hinton & Salakhutdinov (2006).

Recurrent neural networks (RNNs), the temporal analogue of DNNs, are highly expressive sequence models that can model complex sequence relationships. They can be viewed as very deep neural networks that have a “layer” for each time-step with parameter sharing across the layers and, for this reason, they are considered to be even harder to train than DNNs. Recently, Martens & Sutskever (2011) showed that the HF method of Martens (2010) could effectively train RNNs on artificial problems that exhibit very long-range dependencies (Hochreiter & Schmidhuber, 1997). Without resorting to special types of memory units, these problems were considered to be impossibly difficult for first-order optimization methods due

References

1. Sebastian Ruder. *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747, 2016.
2. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
3. Yoshua Bengio. *Practical recommendations for gradient-based training of deep architectures*. In Neural networks: Tricks of the trade, pages 437–478. Springer, 2012.
4. Ning Qian. *On the momentum term in gradient descent learning algorithms*. *Neural networks*, 12(1):145–151, 1999.
5. John Duchi, Elad Hazan, and Yoram Singer. *Adaptive subgradient methods for online learning and stochastic optimization*. *Journal of machine learning research*, 12(7), 2011.
6. Matthew D Zeiler. *Adadelta: an adaptive learning rate method*. arXiv preprint arXiv:1212.5701, 2012.

References

7. Yoshua Bengio and MONTREAL CA. *Rmsprop and equilibrated adaptive learning rates for nonconvex optimization*. corr abs/1502.04390, 2015.
8. Diederik P Kingma and Jimmy Ba. *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980, 2014.
9. Schmidt, Robin M., Frank Schneider, and Philipp Hennig. *Descending through a Crowded Valley--Benchmarking Deep Learning Optimizers*. arXiv preprint arXiv:2007.01547 (2020).