Optimizing Neural Networks

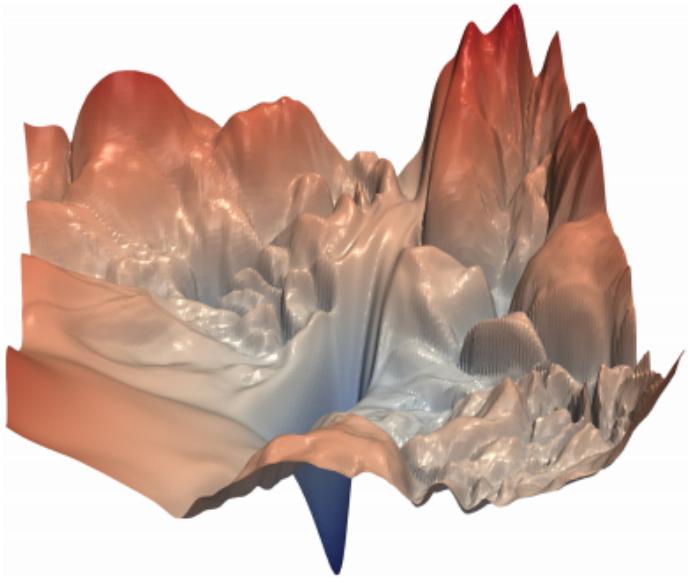
part 1 of 2

Challenges in optimization, and momentum & Adam

This deck embeds the LMU *Introduction to Deep Learning* optimization chunks in full. **Part 1:** challenges → advanced optimization (momentum, Adam, BatchNorm). **Part 2:** initialization & activations.

Deep Learning

Challenges in Optimization



Learning goals

- Ill-Conditioning
- Local Minima
- Saddle Points
- Cliffs and Exploding Gradients

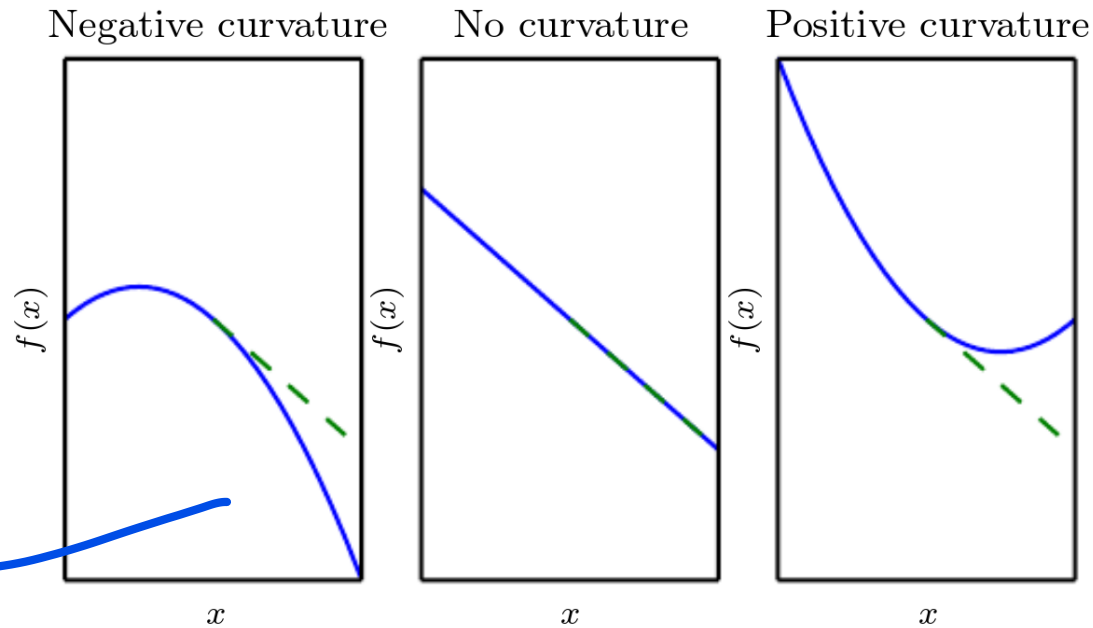
CHALLENGES IN OPTIMIZATION

- In this section, we summarize several of the most prominent challenges regarding training of deep neural networks.
- Traditionally, machine learning ensures that the optimization problem is convex by carefully designing the objective function and constraints. But for neural networks we are confronted with the general nonconvex case.
- Furthermore, we will see in this section that even convex optimization is not without its complications.

Ill-Conditioning

EFFECTS OF CURVATURE

Intuitively, the curvature of a function determines the outcome of a GD step. . .



Source: Goodfellow *et al.*, (2016), ch. 4

Figure: Quadratic objective function $f(\mathbf{x})$ with various curvatures. The dashed line indicates the first order taylor approximation based on the gradient information alone. Left: With negative curvature, the cost function decreases faster than the gradient predicts; Middle: With no curvature, the gradient predicts the decrease correctly; Right: With positive curvature, the function decreases more slowly than expected and begins to increase.

SECOND DERIVATIVE AND CURVATURE

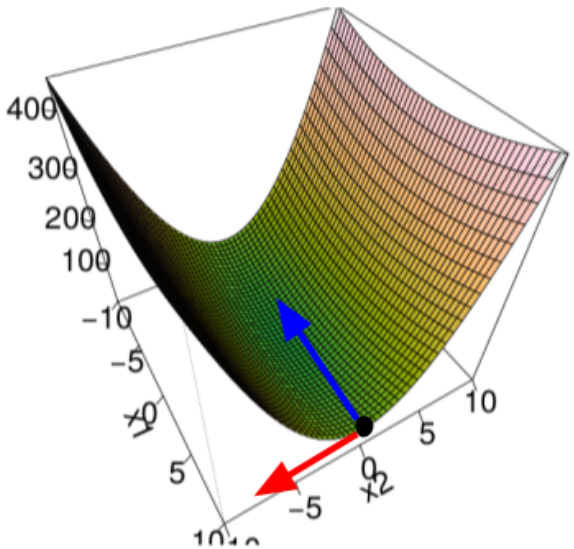
To understand better how the curvature of a function influences the outcome of a gradient descent step, let us recall how curvature is described mathematically:

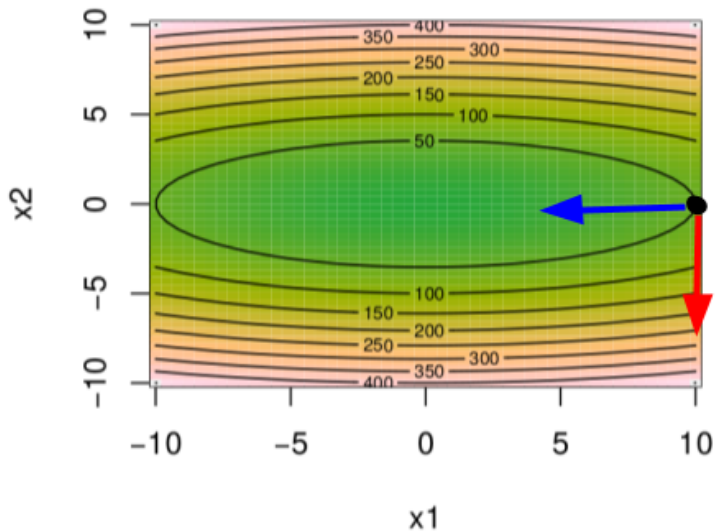
- The second derivative corresponds to the curvature of the graph of a function.
- The **Hessian** matrix of a function $\mathcal{R}(\boldsymbol{\theta}) : \mathbb{R}^m \to \mathbb{R}$ is the matrix of second-order partial derivatives

$$H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} \mathcal{R}(\boldsymbol{\theta}).$$

SECOND DERIVATIVE AND CURVATURE / 2

- The second derivative in a direction $\mathbf{d}$, with $\|\mathbf{d}\| = 1$, is given by $\mathbf{d}^\top \mathbf{H}\, \mathbf{d}$.
- What is the direction of the highest curvature (red direction), and what is the direction of the lowest curvature (blue)?

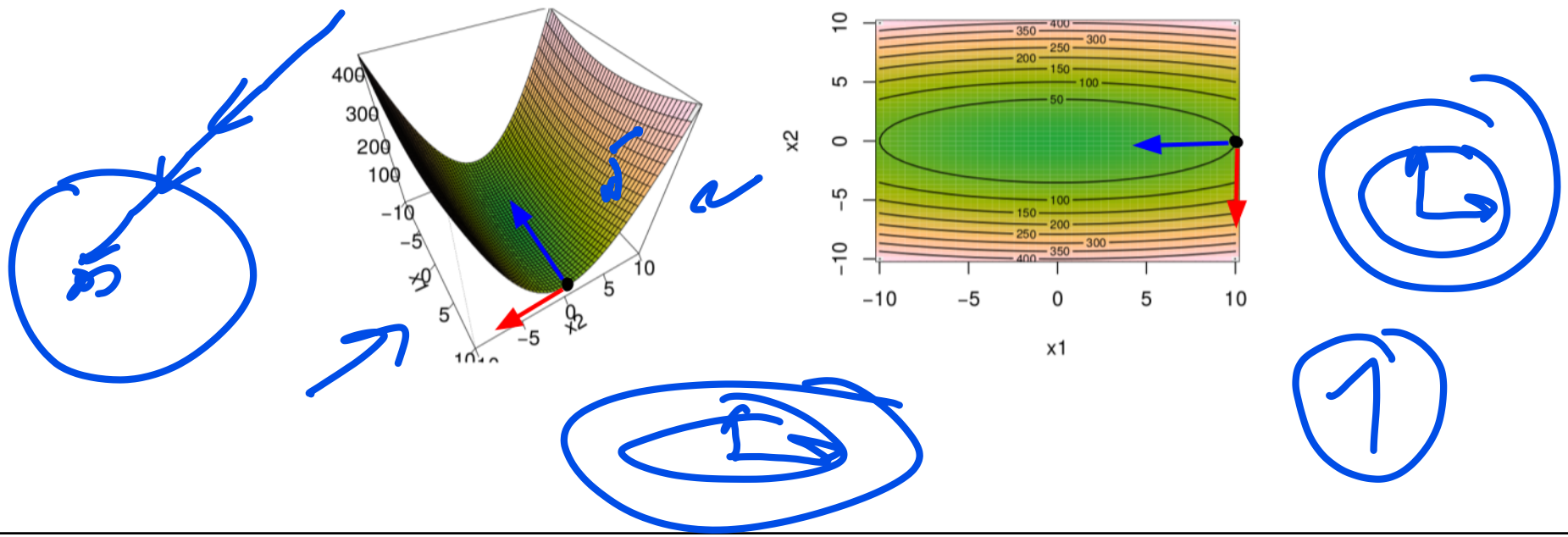




Eigenvectors

SECOND DERIVATIVE AND CURVATURE / 3

- Since $\mathbf{H}$ is real and symmetric (why?), eigendecomposition yields $\mathbf{H} = \mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}$ with $\mathbf{V}$ and $\boldsymbol{\lambda}$ collecting eigenvectors and eigenvalues, respectively.
- It can be shown, that the eigenvector $\boldsymbol{v}_{\text{max}}$ with the max. eigenvalue λ_{max} points into the direction of highest curvature ($\boldsymbol{v}_{\text{max}}^\top \boldsymbol{H}\boldsymbol{v}_{\text{max}} = \lambda_{\text{max}}$), while the eigenvector $\boldsymbol{v}_{\text{min}}$ with the min. eigenvalue λ_{min} points into the direction of least curvature.



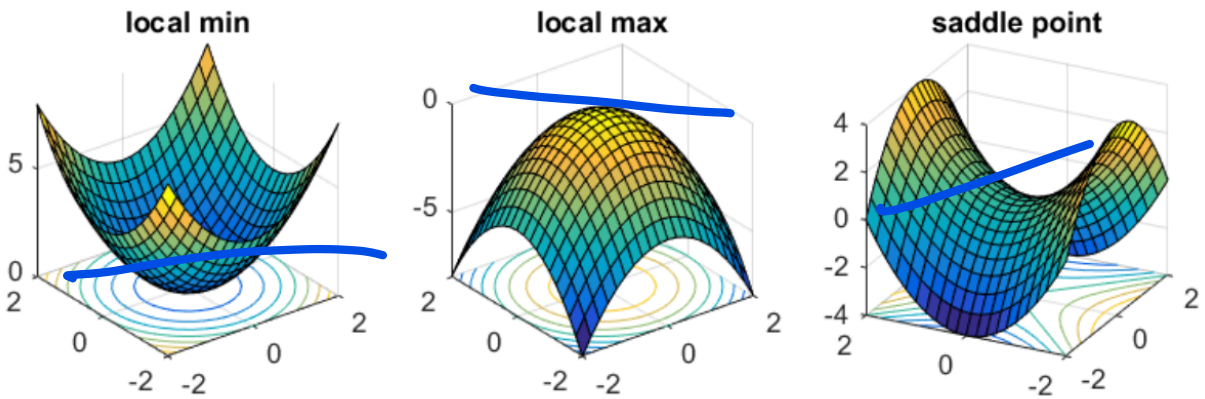
SECOND DERIVATIVE AND CURVATURE / 4

- At a stationary point $\boldsymbol{\theta}$, where the gradient is 0, we can examine the eigenvalues of the Hessian to determine whether the $\boldsymbol{\theta}$ is a local maximum, minimum or saddle point:

$$\forall i : \lambda_i > 0 \,(\mathbf{H} \text{ positive definite at } \boldsymbol{\theta}) \quad \Rightarrow \quad \text{minimum at } \boldsymbol{\theta}$$

$$\forall i : \lambda_i < 0 \,(\mathbf{H} \text{ negative definite at } \boldsymbol{\theta}) \quad \Rightarrow \quad \text{maximum at } \boldsymbol{\theta}$$

$$\exists\, i : \lambda_i < 0 \wedge \exists j : \lambda_j > 0 \,(\mathbf{H} \text{ indefinit at } \boldsymbol{\theta}) \quad \Rightarrow \quad \text{saddle point at } \boldsymbol{\theta}$$

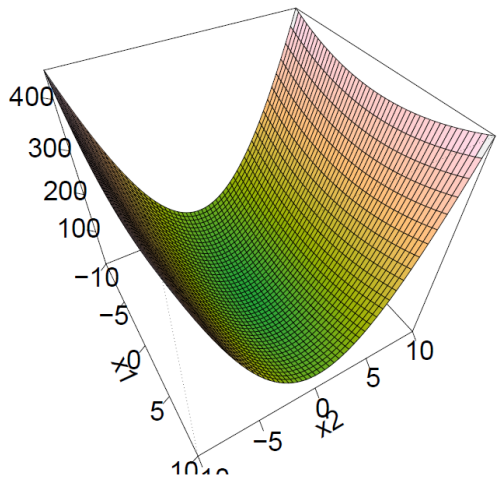


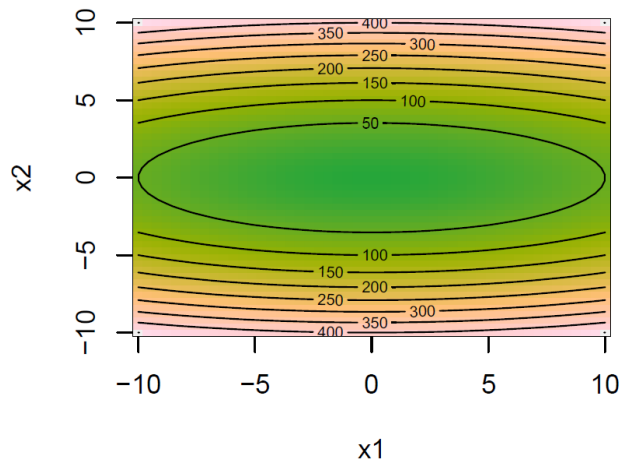
Credit: Rong Ge (2016)

ILL-CONDITIONED HESSIAN MATRIX

The condition number of a symmetric matrix $\boldsymbol{A}$ is given by the ratio of its min/max eigenvalues $\kappa(\boldsymbol{A}) = \frac{|\lambda_{max}|}{|\lambda_{min}|}$. A matrix is called ill-conditioned, if the condition number $\kappa(\boldsymbol{A})$ is very high.

An **ill-conditioned** Hessian matrix means that the ratio of max. / min. curvature is high, as in the example below:





CURVATURE AND STEP-SIZE IN GD

What does it mean for gradient descent if the Hessian is ill-conditioned?

- Let us consider the second-order Taylor approximation as a local approximation of the of $\mathcal{R}$ around a current point $\boldsymbol{\theta}^0$ (with gradient $\mathbf{g}$)

$$\mathcal{R}(\boldsymbol{\theta}) \approx T_2 f(\boldsymbol{\theta}, \boldsymbol{\theta}^0) := \mathcal{R}(\boldsymbol{\theta}^0) + (\boldsymbol{\theta} - \boldsymbol{\theta}^0)^\top \mathbf{g} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}^0)$$

- Furthermore, Taylor's theorem states (proof in Koenigsberger (1997), p. 68)

$$\lim_{\boldsymbol{\theta} \to \boldsymbol{\theta}^0} \frac{\mathcal{R}(\boldsymbol{\theta}) - T_2 f(\boldsymbol{\theta}, \boldsymbol{\theta}^0)}{||\boldsymbol{\theta} - \boldsymbol{\theta}^0||^2} = 0$$

CURVATURE AND STEP-SIZE IN GD / 2

- One GD step with a learning rate α yields new parameters $\boldsymbol{\theta}^0 - \alpha\mathbf{g}$ and a new approximated loss value

$$\mathcal{R}(\boldsymbol{\theta}^0 - \alpha\mathbf{g}) \approx \mathcal{R}(\boldsymbol{\theta}^0) - \alpha\mathbf{g}^\top\mathbf{g} + \frac{1}{2}\alpha^2\mathbf{g}^\top\mathbf{H}\,\mathbf{g}\,.$$

- Theoretically, if $\mathbf{g}^\top\mathbf{H}\mathbf{g}$ is positive, we can solve the equation above for the optimal step size which corresponds to

$$\alpha^* = \frac{\mathbf{g}^\top\mathbf{g}}{\mathbf{g}^\top\mathbf{H}\,\mathbf{g}}\,.$$

CURVATURE AND STEP-SIZE IN GD / 3

- Let us assume the gradient $\mathbf{g}$ points into the direction of $\boldsymbol{v}_{\text{max}}$ (i.e. the direction of highest curvature), the optimal step size is given by

$$\alpha^* = \frac{\mathbf{g}^\top \mathbf{g}}{\mathbf{g}^\top \mathbf{H}\, \mathbf{g}} = \frac{\mathbf{g}^\top \mathbf{g}}{\lambda_{\text{max}} \mathbf{g}^\top \mathbf{g}} = \frac{1}{\lambda_{\text{max}}},$$

  which is very small. Choosing a too large step-size is bad, as it will make us "overshoot" the stationary point.

- If, on the other hand, $\mathbf{g}$ points into the direction of the lowest curvature, the optimal step size is

$$\alpha^* = \frac{1}{\lambda_{\text{min}}},$$

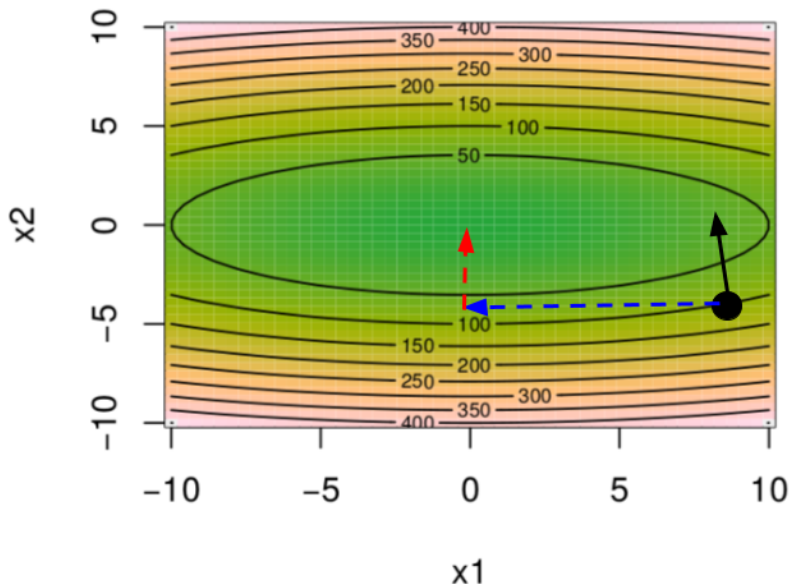
  which corresponds to the largest possible optimal step-size.

- We summarize: We want to perform big steps in directions of low curvature, but small steps in directions of high curvature.

CURVATURE AND STEP-SIZE IN GD / 4

- But what if the gradient does not point into the direction of one of the eigenvectors?
- Let us consider the 2-dimensional case: We can decompose the direction of $\boldsymbol{g}$ (black) into the two eigenvectors $\boldsymbol{v}_{\text{max}}$ and $\boldsymbol{v}_{\text{min}}$
- It would be optimal to perform a **big** step into the direction of the smallest curvature $\boldsymbol{v}_{\text{min}}$, but a **small** step into the direction of $\boldsymbol{v}_{\text{max}}$, but the gradient points into a completely different direction.

CURVATURE AND STEP-SIZE IN GD / 5



ILL-CONDITIONING

- GD is unaware of large differences in curvature, and can only walk into the direction of the gradient.
- Choosing a too large step-size will then cause the descent direction change frequently (“jumping around”).
- α needs to be small enough, which results in a low progress.

ILL-CONDITIONING / 2

- This effect is more severe, if a Hessian has a poor condition number, i.e. the ratio between lowest and highest curvature is large; gradient descent will perform poorly.

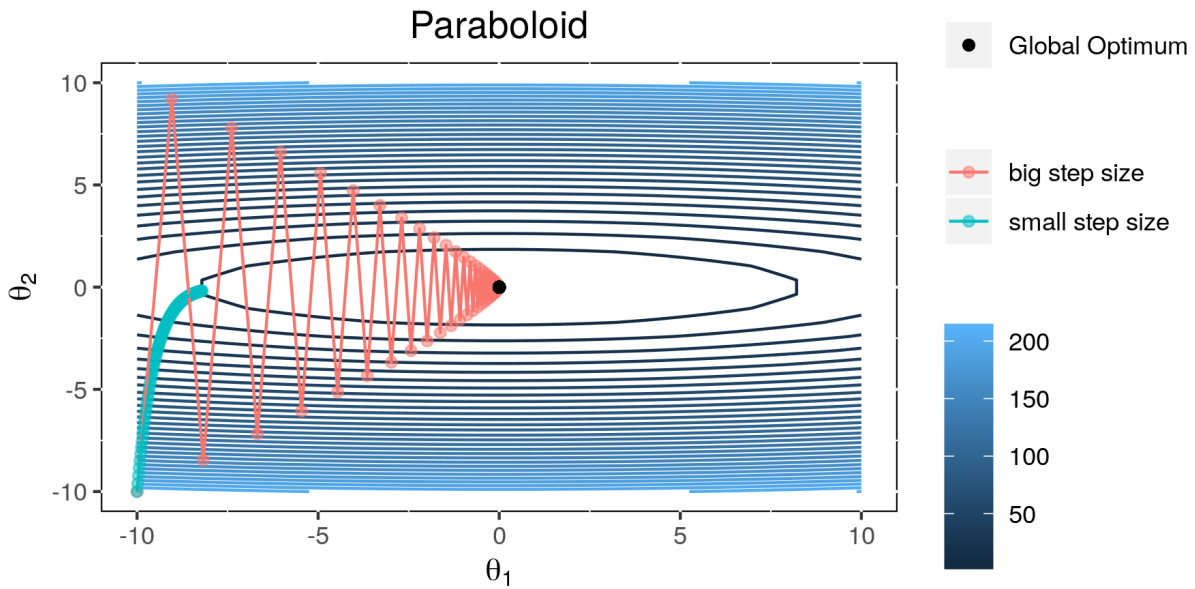


Figure: The contour lines show a quadratic risk function with a poorly conditioned Hessian matrix. The plot shows the progress of gradient descent with a small step-size vs. larger step-size. In both cases, convergence to the global optimum is rather slow.

ILL-CONDITIONING / 3

- In the worst case, ill-conditioning of the Hessian matrix and a too big step-size will cause the risk to increase

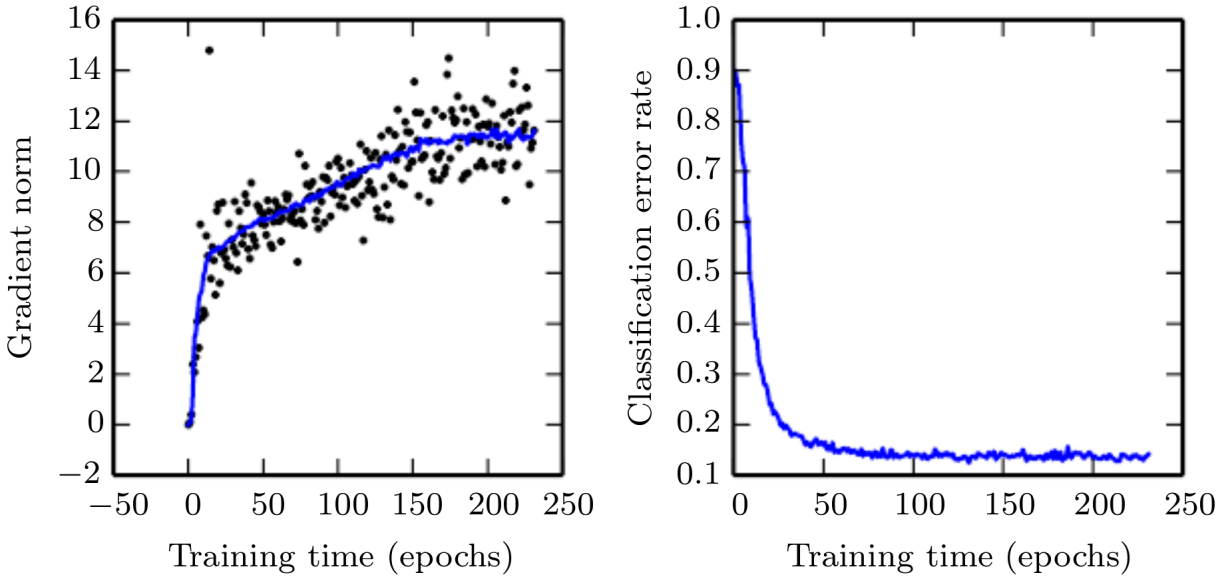
$$\mathcal{R}(\boldsymbol{\theta}^0 - \alpha \mathbf{g}) \approx \mathcal{R}(\boldsymbol{\theta}^0) - \alpha \mathbf{g}^\top \mathbf{g} + \frac{1}{2}\alpha^2 \mathbf{g}^\top \mathbf{H}\, \mathbf{g}\,,$$

  which happens if

$$\frac{1}{2}\alpha^2 \mathbf{g}^\top \mathbf{H}\, \mathbf{g} > \alpha \mathbf{g}^\top \mathbf{g}.$$

- To determine whether ill-conditioning is detrimental to the training, the squared gradient norm $\mathbf{g}^\top \mathbf{g}$ and the risk can be monitored.

ILL-CONDITIONING / 4



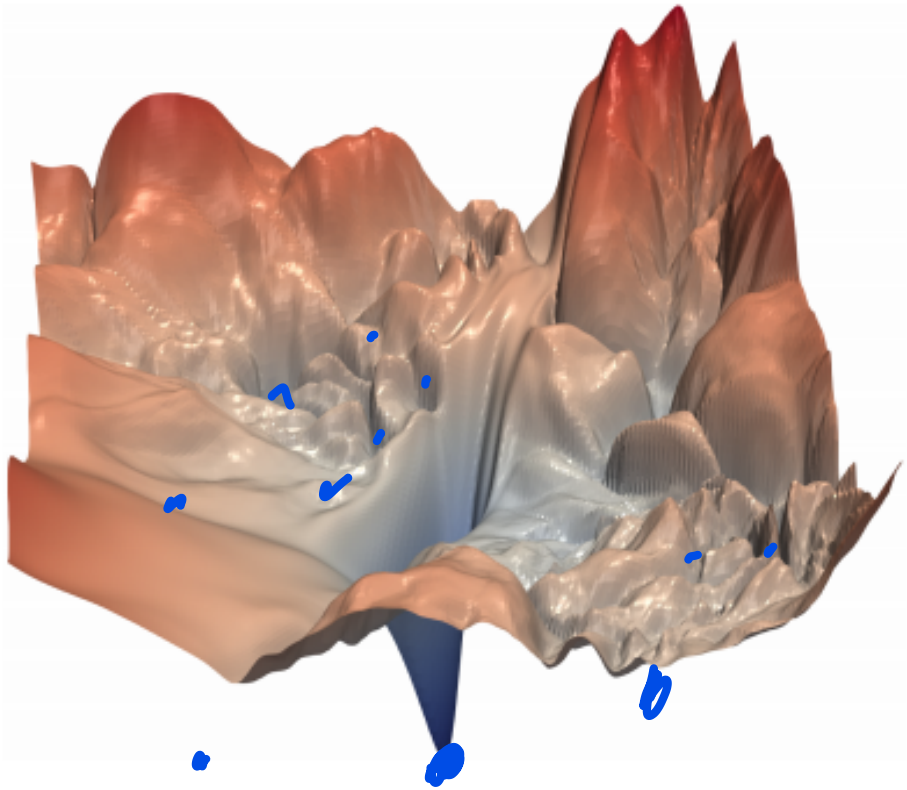
Source: Goodfellow, ch. 6

- Gradient norms **increase** over time, showing that the training process is not converging to a stationary point $\mathbf{g} = 0$.
- At the same time, we observe that the risk is approx. constant, but the gradient norm increases

$$\underbrace{\mathcal{R}(\boldsymbol{\theta}^0 - \alpha\mathbf{g})}_{\text{approx. constant}} \approx \mathcal{R}(\boldsymbol{\theta}^0) - \underbrace{\alpha\mathbf{g}^\top\mathbf{g}}_{\text{increase}} + \frac{1}{2}\underbrace{\alpha^2\mathbf{g}^\top\mathbf{H}\,\mathbf{g}}_{\rightarrow\text{increase}} .$$

Local Minima

UNIMODAL VS. MULTIMODAL LOSS SURFACES



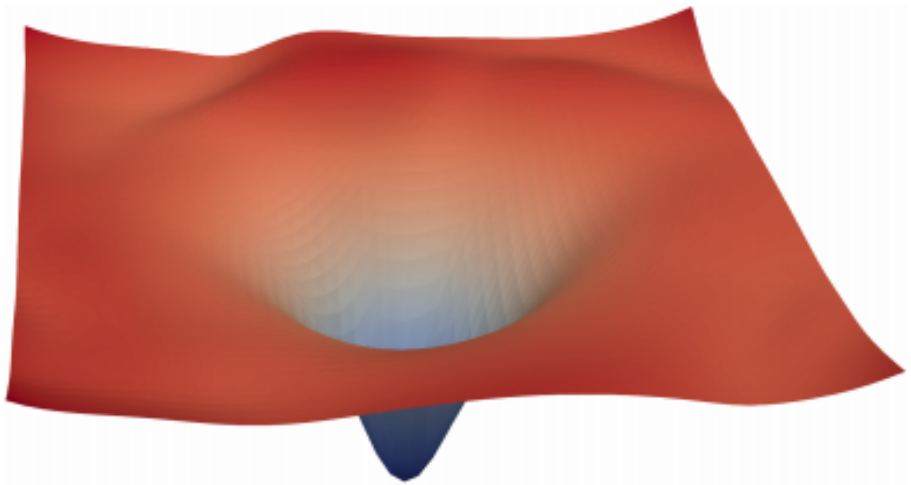
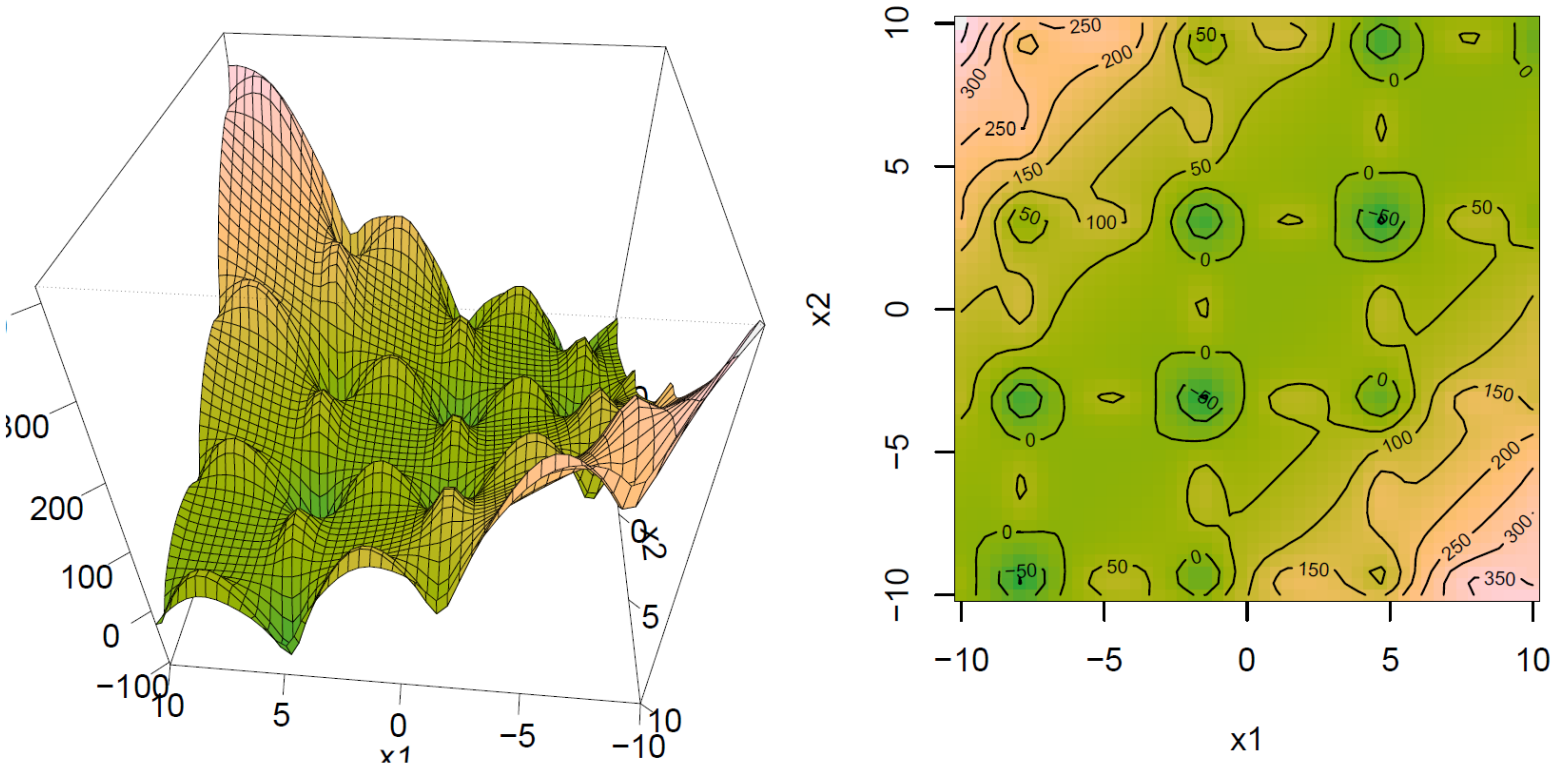


Figure: Left: Multimodal loss surface with saddle points; Right: (Nearly) unimodal loss surface (Hao Li et al. (2017))

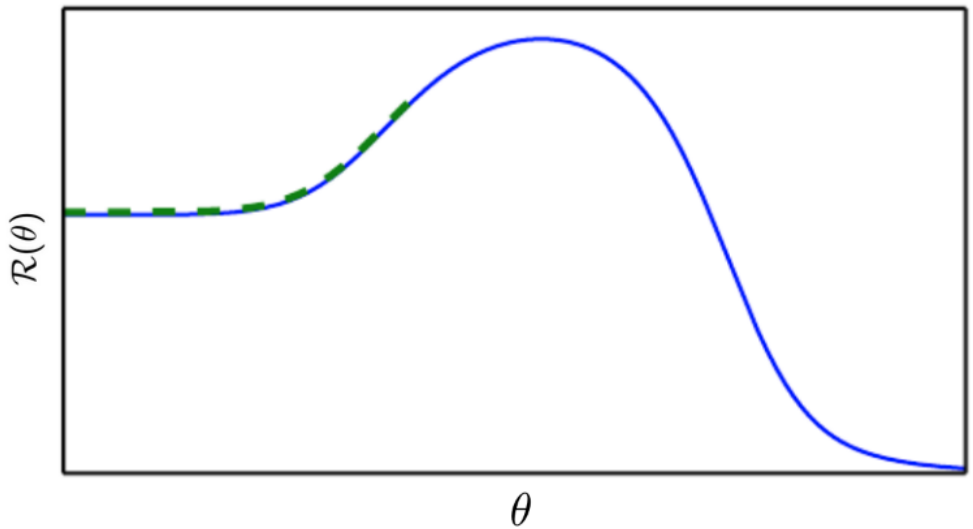
MULTIMODAL FUNCTION

Potential snippet from a loss surface of a deep neural network with many local minima:



ONLY LOCALLY OPTIMAL MOVES

- If the training algorithm makes only *locally* optimal moves (as in gradient descent), it may move away from regions of *much* lower cost.



Source: Goodfellow, Ch. 8

- In the figure above, initializing the parameter on the "wrong" side of the hill will result in suboptimal performance.
- In higher dimensions, however, it may be possible for gradient descent to go around the hill but such a trajectory might be very long and result in excessive training time.

LOCAL MINIMA

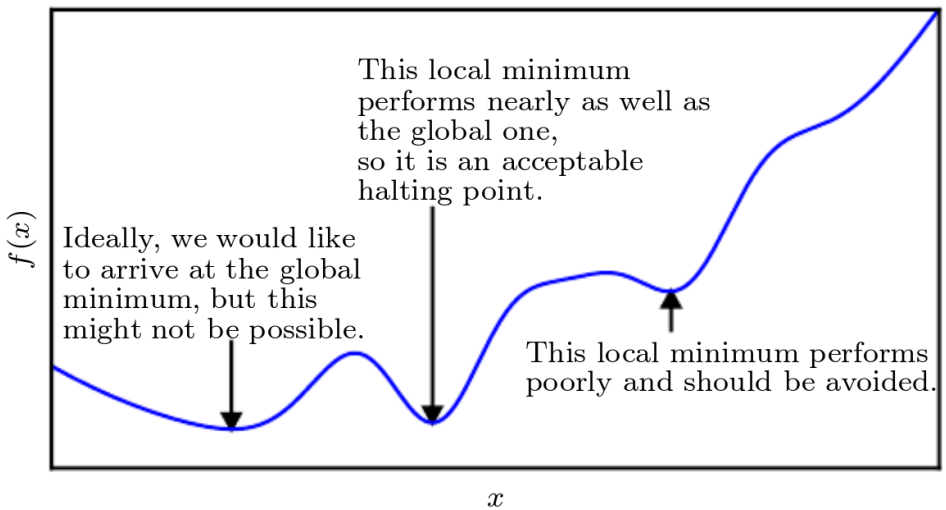
Weight space symmetry:

- If we swap incoming weight vectors for neuron *i* and *j* and do the same for the outcoming weights, modelled function stays unchanged.
  - $\Rightarrow$ with *n* hidden units and one hidden layer there are *n*! networks with the same empirical risk
- If we multiply incoming weights of a ReLU neuron with β and outcoming with $1/\beta$ the modelled function stays unchanged.

$\Rightarrow$ The empirical risk of a NN can have very many minima with equivalent empirical risk.

LOCAL MINIMA / 2

- In practice only local minima with a high value compared to the global minimium are problematic.



Source: Goodfellow, Ch. 4

- Current literature suspects that most local minima have low empirical risk.

Saddle Points

SADDLE POINTS

- In optimization we look for areas with zero gradient.
- A variant of zero gradient areas are saddle points.
- For the empirical risk $\mathcal{R}$ of a neural network, the expected ratio of the number of saddle points to local minima typically grows exponentially with m

$$\mathcal{R} : \mathbb{R}^m \to \mathbb{R}$$

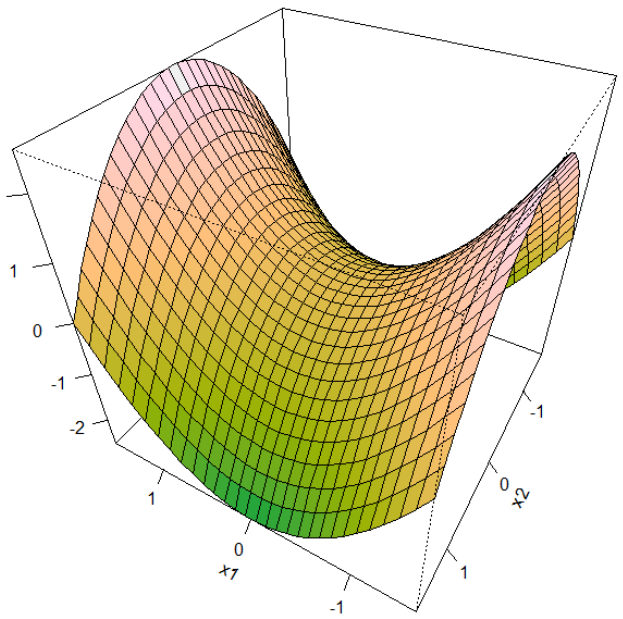
  In other words: Networks with more parameters (deeper networks or larger layers) exhibit a lot more saddle points than local minima.
- Why is that?
- The Hessian at a local minimum has only positive eigenvalues. At a saddle point it is a mixture of positive and negative eigenvalues.

SADDLE POINTS / 2

- Imagine the sign of each eigenvalue is generated by coin flipping:
  - In a single dimension, it is easy to obtain a local minimum (e.g. “head” means positive eigenvalue).
  - In an m-dimensional space, it is exponentially unlikely that all m coin tosses will be head.
- A property of many random functions is that eigenvalues of the Hessian become more likely to be positive in regions of lower cost.
- For the coin flipping example, this means we are more likely to have heads m times if we are at a critical point with low cost.
- That means in particular that local minima are much more likely to have low cost than high cost and critical points with high cost are far more likely to be saddle points.
- See Dauphin et al. (2014) for a more detailed investigation.

SADDLE POINTS: EXAMPLE

$$f(x_1, x_2) = x_1^2 - x_2^2$$

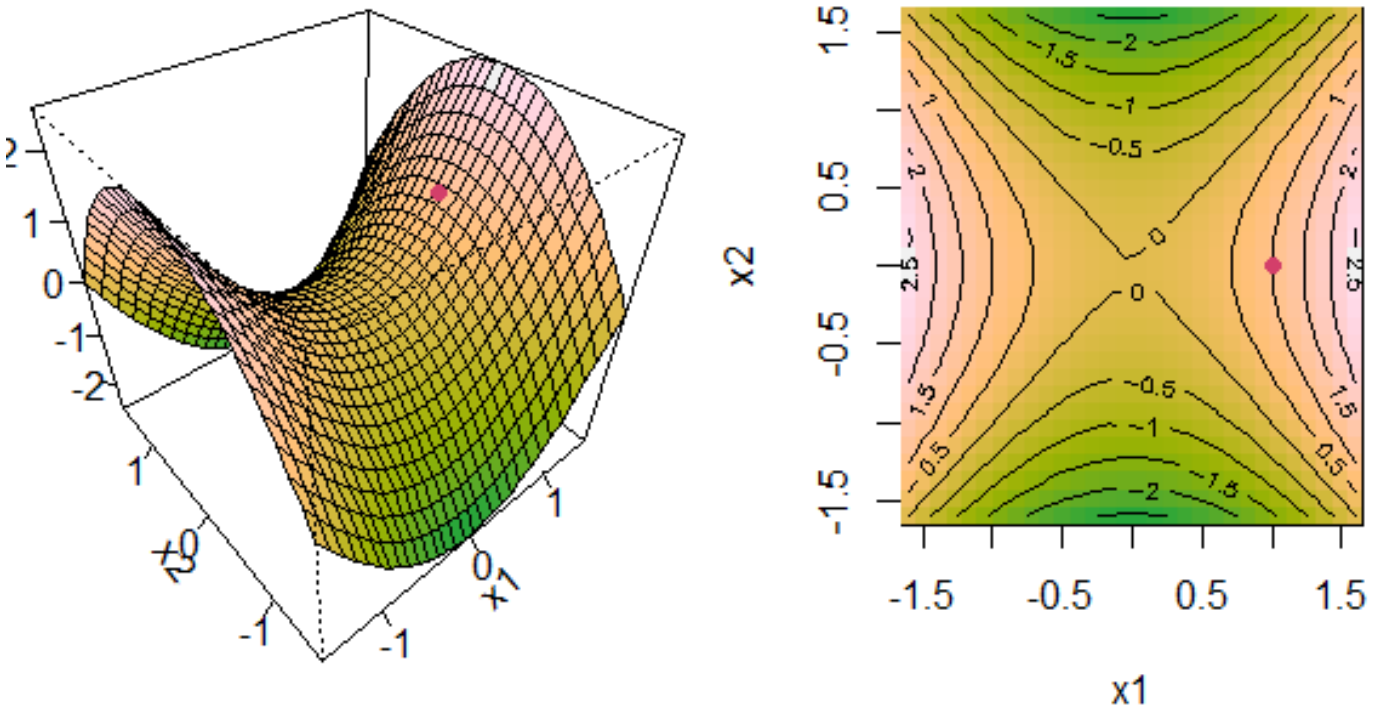


- Along x_1, the function curves upwards (eigenvector of the Hessian with positive eigenvalue). Along x_2, the function curves downwards (eigenvector of the Hessian with negative eigenvalue).

SADDLE POINTS

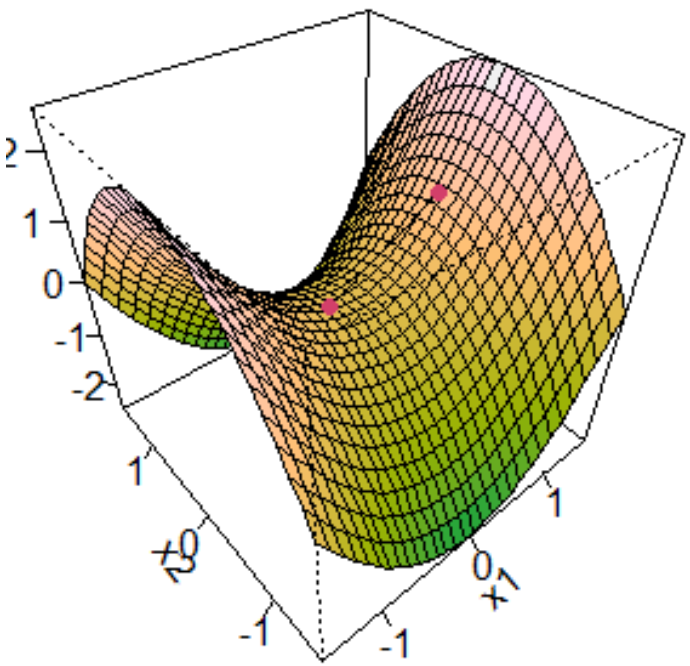
- So how do saddle points impair optimization?
- First-order algorithms that use only gradient information **might** get stuck in saddle points.
- Second-order algorithms experience even greater problems when dealing with saddle points. Newtons method for example actively searches for a region with zero gradient. That might be another reason why second-order methods have not succeeded in replacing gradient descent for neural network training.

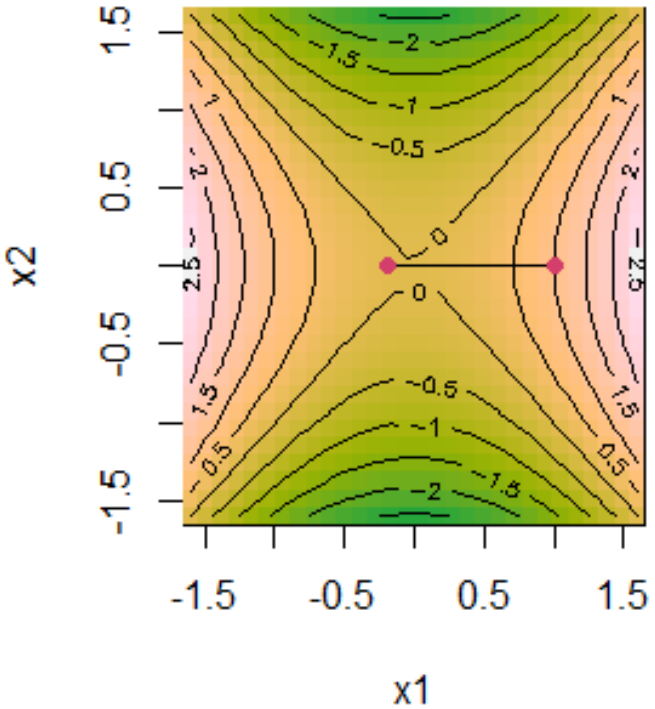
EXAMPLE: SADDLE POINT WITH GD



Red dot: Starting location

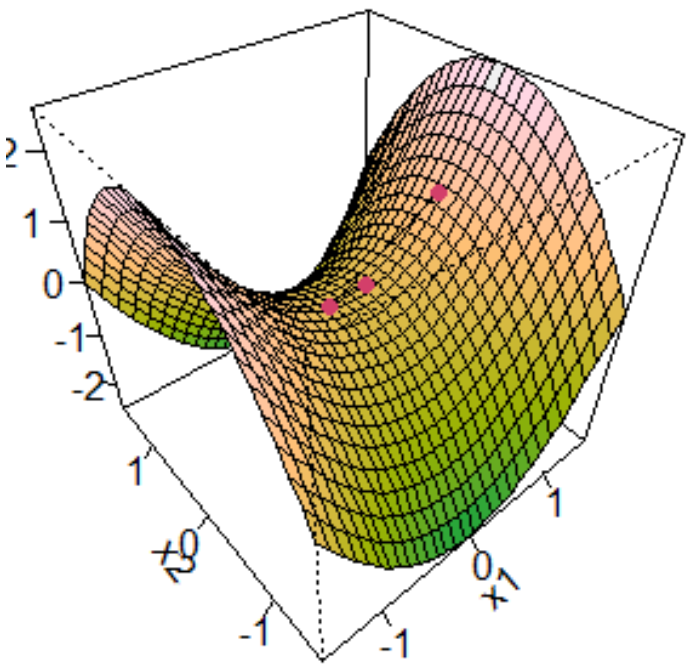
EXAMPLE: SADDLE POINT WITH GD

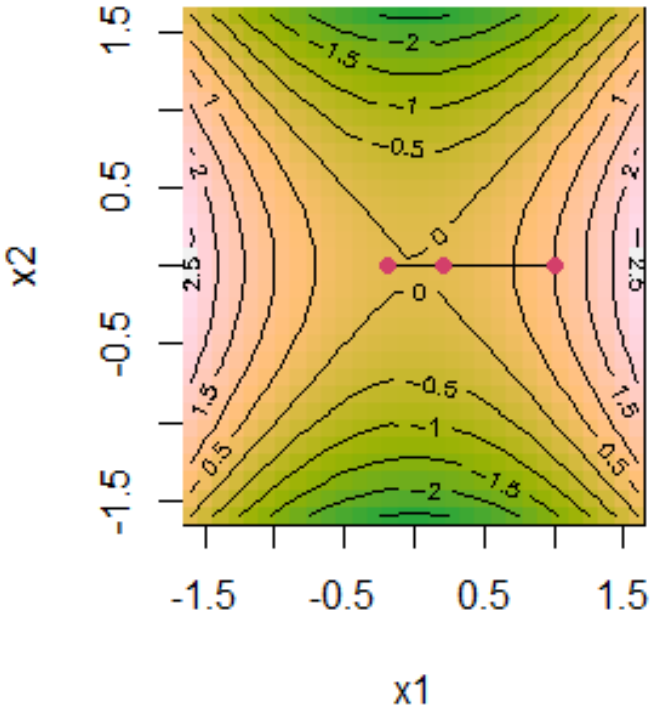




First step...

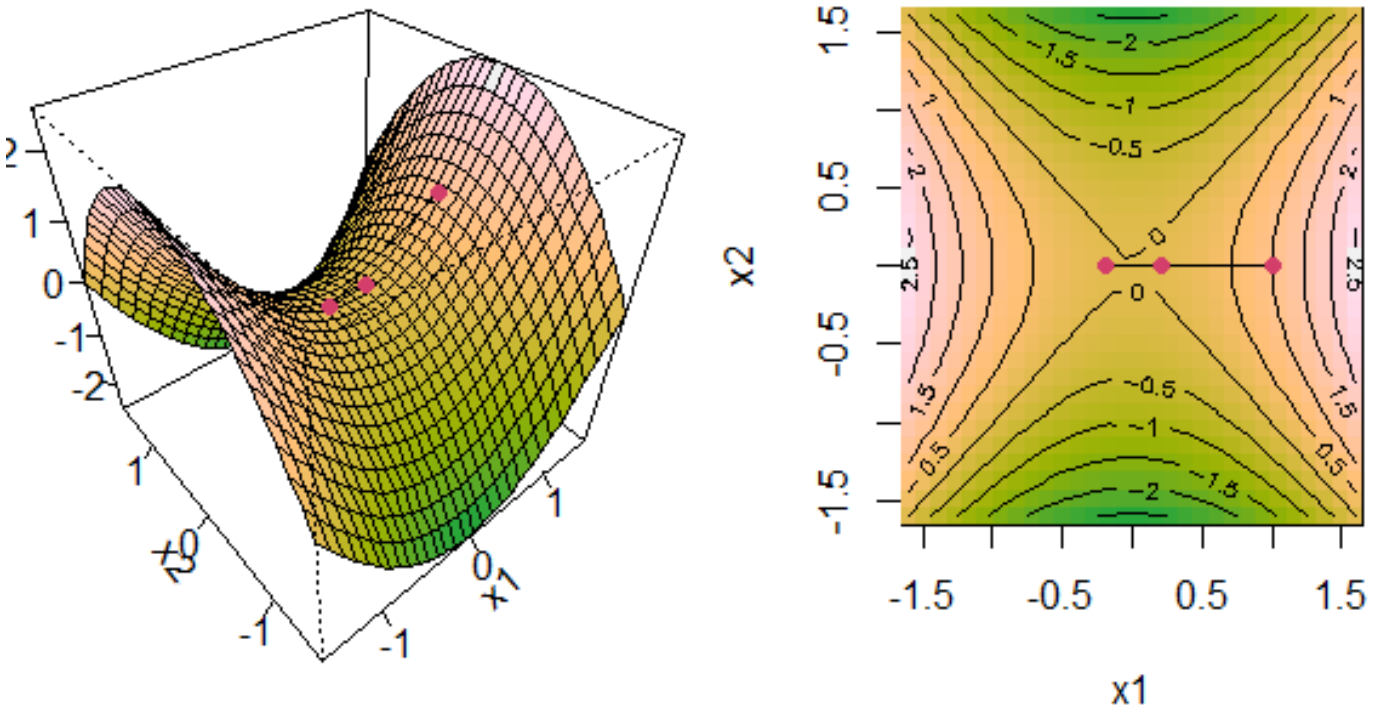
EXAMPLE: SADDLE POINT WITH GD





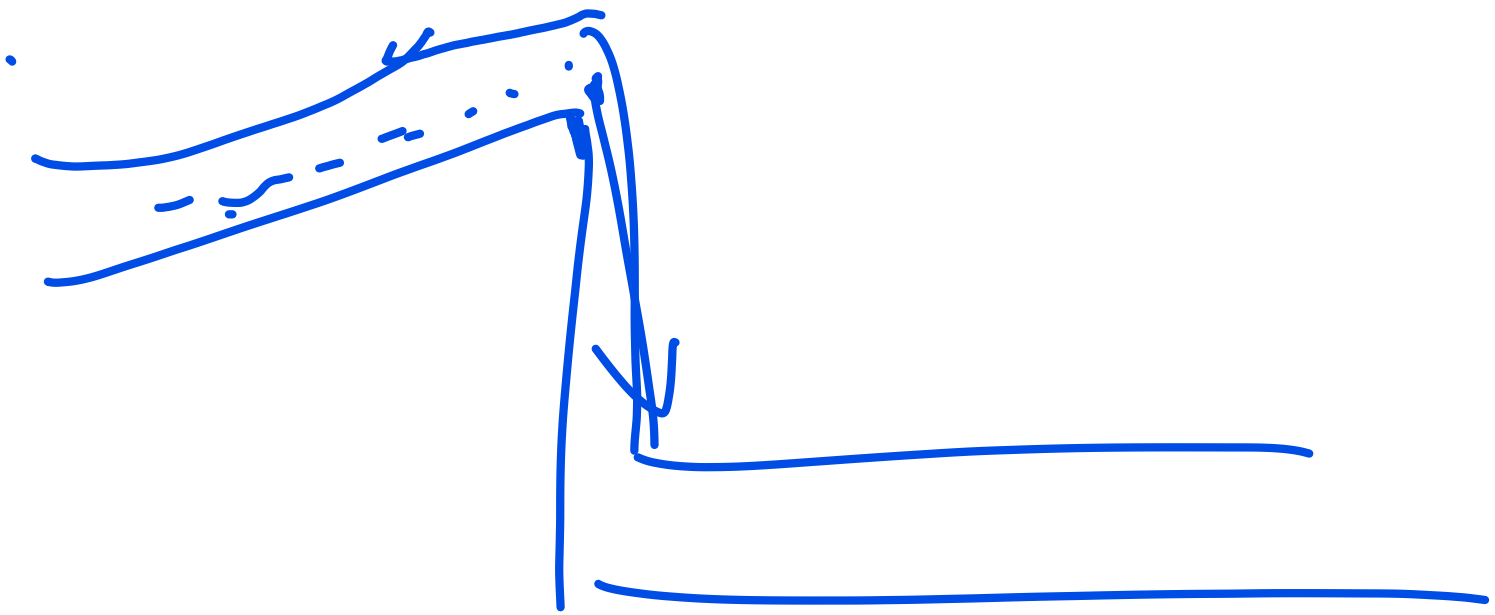
...second step...

EXAMPLE: SADDLE POINT WITH GD

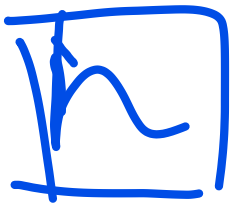


...tenth step got stuck and cannot escape the saddle point!

Cliffs and Exploding Gradients

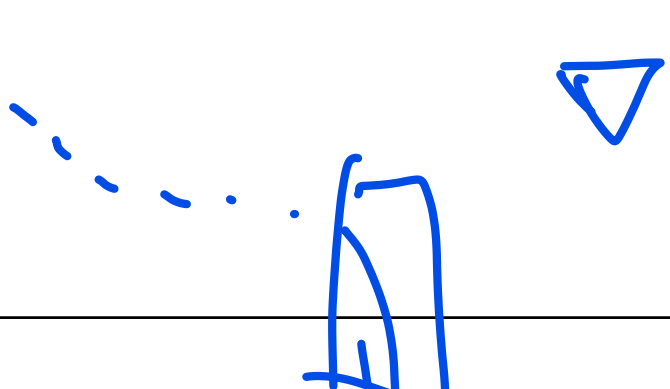


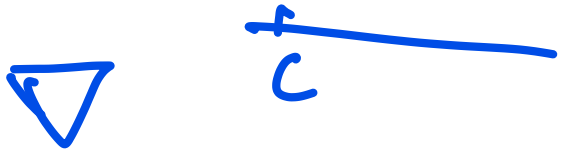
CLIFFS AND EXPLODING GRADIENTS

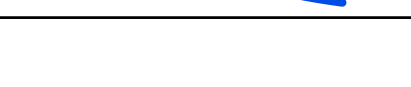


- As a result from the multiplication of several parameters, the emprirical risk for highly nonlinear deep neural networks often contain sharp nonlinearities.
  - That may result in very high derivatives in some places.
  - As the parameters get close to such cliff regions, a gradient descent update can catapult the parameters very far.
  - Such an occurrence can lead to losing most of the optimization work that had been done.

- However, serious consequences can be easily avoided using a technique called **gradient clipping**.
- The gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region.



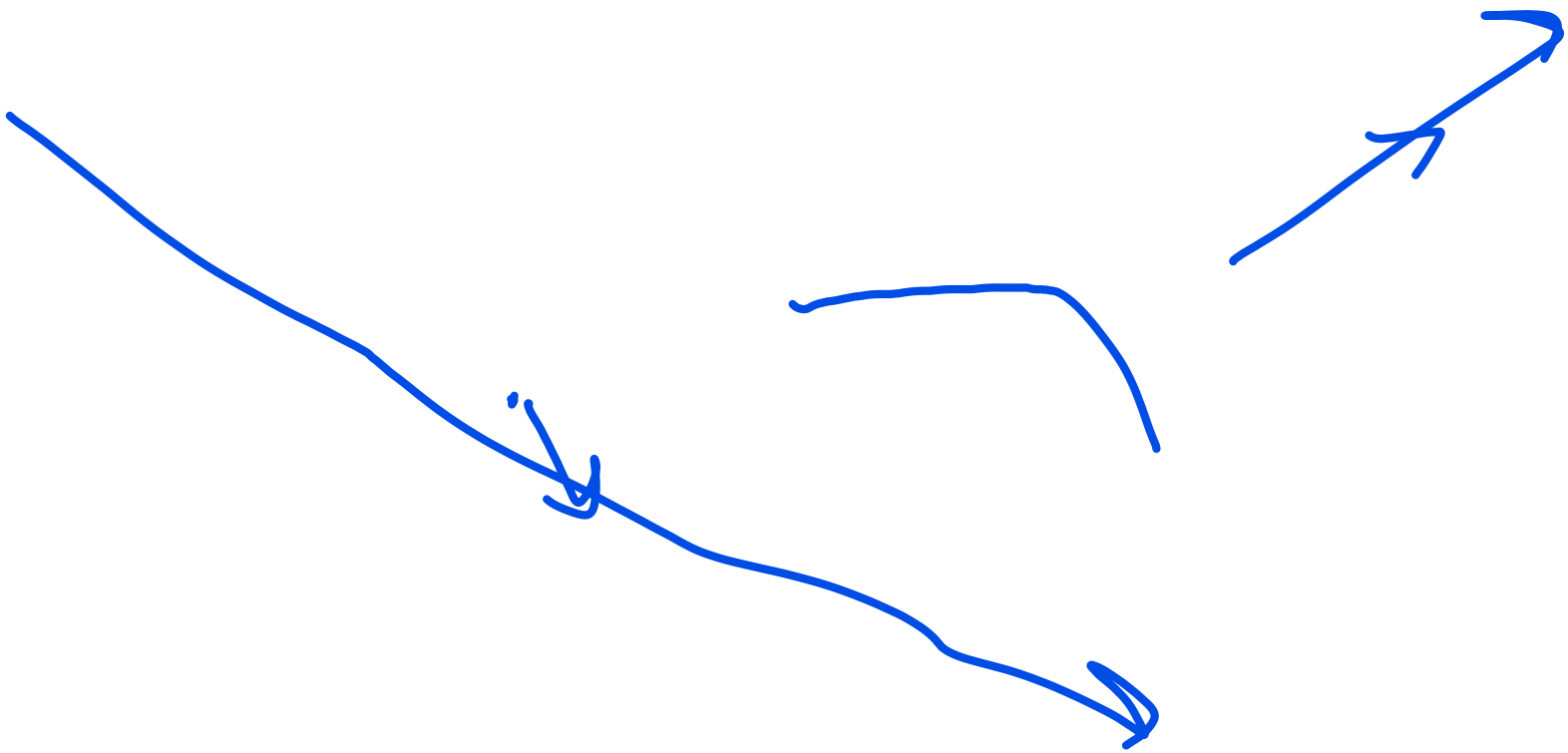




CLIFFS AND EXPLODING GRADIENTS / 2

- Gradient clipping simply caps the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of steepest descent.
- We simply "prune" the norm of the gradient at some threshold h:

$$\text{if } ||\nabla\boldsymbol{\theta}|| > \text{h} : \nabla\boldsymbol{\theta} \leftarrow \frac{h}{||\nabla\boldsymbol{\theta}||}\nabla\boldsymbol{\theta}$$



EXAMPLE: CLIFFS AND EXPLODING GRADIENTS

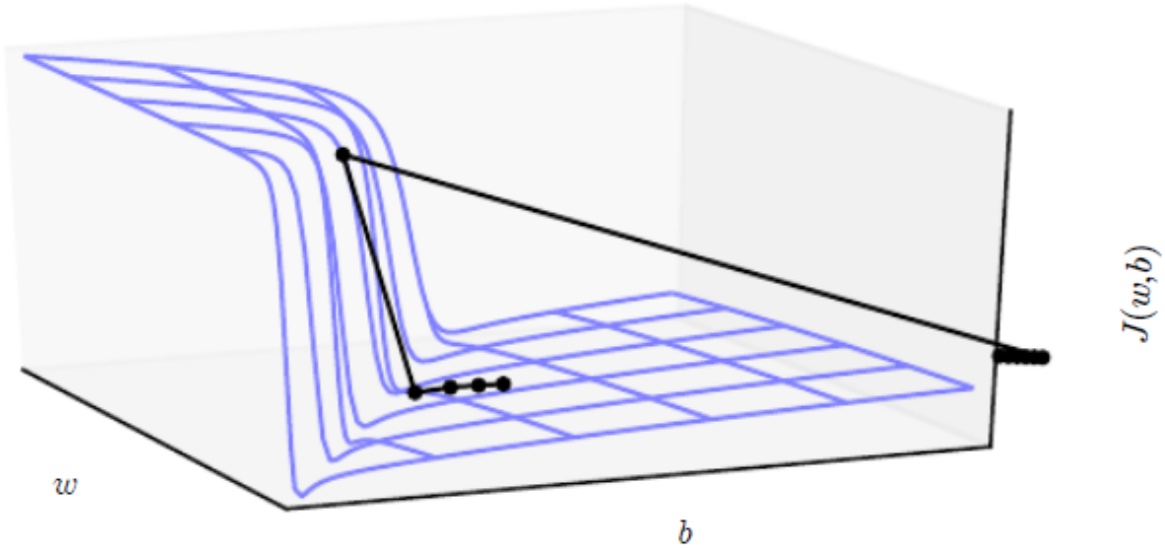


Figure: “The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that had been done” (Goodfellow et al. (2016)).

REFERENCES

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)

Deep Learning

http://www.deeplearningbook.org/

Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio (2014)

Identifying and attacking the saddle point problem in high-dimensional non-convex optimization

https://arxiv.org/abs/1406.2572

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein (2017)

Visualizing the Loss Landscape of Neural Nets

https://arxiv.org/abs/1712.09913

Konrad Koenigsberger (1997)

Analysis 2, Springer

Rong Ge (2016)

Escaping from Saddle Points

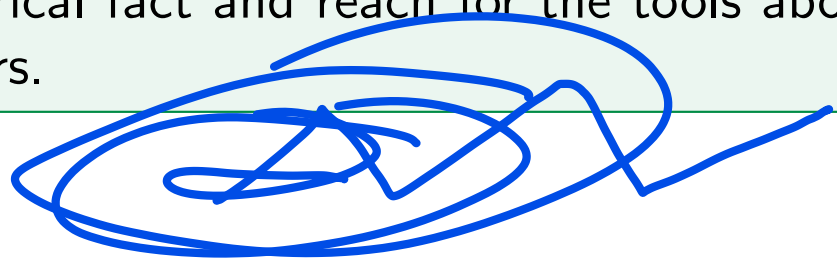
http://www.offconvex.org/2016/03/22/saddlepoints/

The landscape is scary – and training works anyway

LMU just catalogued everything that can go wrong: ill-conditioning, local minima, saddle points, cliffs. Yet SGD trains huge nets reliably. Why?

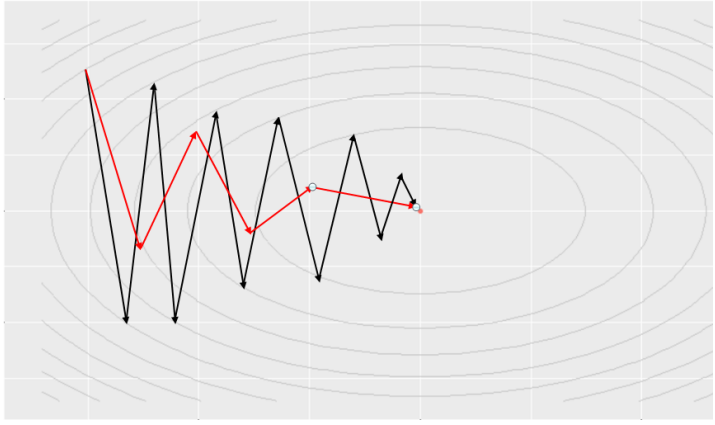
- **Saddles, not traps.** In high dimensions almost every flat spot is a **saddle** (up in some directions, down in others) – and the **noise** in mini-batch SGD nudges you off it. Plain gradient descent from a random start also escapes (Lee et al., 2016), but the gradient near a saddle is tiny, so it can crawl for a long time.
- **Most minima are fine.** Empirically, the many local minima of a big net have *similar*, low loss – you rarely need *the* global one.
- **Cliffs** $\rightarrow$ **gradient clipping** (cap the step). **Ill-conditioning** $\rightarrow$ **momentum / adaptive learning rates** (next section).

Why non-convex SGD works so well is still an open research question – we take it as an empirical fact and reach for the tools above when a specific symptom appears.



Deep Learning

Advanced Optimization



Learning goals

- SGD with Momentum
- Learning Rate Schedules
- Adaptive Learning Rates
- Batch Normalization

Momentum

MOMENTUM

- While SGD remains a popular optimization strategy, learning with it can sometimes be slow.
- Momentum is designed to accelerate learning, especially when facing high curvature, small but consistent or noisy gradients.
- Momentum accumulates an exponentially decaying moving average of past gradients:

$$\begin{aligned} \boldsymbol{\nu} &\leftarrow \varphi\boldsymbol{\nu} - \alpha \underbrace{\nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_i L(y^{(i)}, f(x^{(i)}, \boldsymbol{\theta})) \right]}_{\mathbf{g}_{\boldsymbol{\theta}}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{\nu} \end{aligned}$$

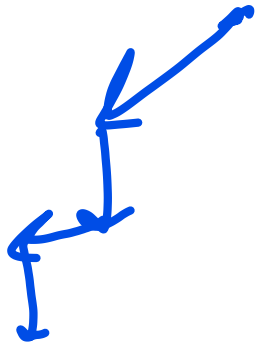
- We introduce a new hyperparameter $\varphi \in [0, 1)$, determining how quickly the contribution of previous gradients decay.
- $\boldsymbol{\nu}$ is called "velocity" and derives from a physical analogy describing how particles move through a parameter space (Newton's law of motion).

MOMENTUM

- So far the step size was simply the gradient **g** multiplied by the learning rate α.
- Now, the step size depends on how **large** and how **aligned** a sequence of gradients is. The step size grows when many successive gradients point in the same direction.
- Common values for φ are $0.5, 0.9$ and even 0.99.
- Generally, the larger φ is relative to the learning rate α, the more previous gradients affect the current direction.
- A very good website with an in-depth analysis of momentum: `https://distill.pub/2017/momentum/`

MOMENTUM: EXAMPLE

$$
\begin{aligned}
\nu_1 &\leftarrow \varphi\nu_0 - \alpha g(\theta^{[0]}) \\
\theta^{[1]} &\leftarrow \theta^{[0]} + \varphi\nu_0 - \alpha g(\theta^{[0]}) \\
\nu_2 &\leftarrow \varphi\nu_1 - \alpha g(\theta^{[1]}) \\
&= \varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]}) \\
\theta^{[2]} &\leftarrow \theta^{[1]} + \varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]}) \\
\nu_3 &\leftarrow \varphi\nu_2 - \alpha g(\theta^{[2]}) \\
&= \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]}) \\
\theta^{[3]} &\leftarrow \theta^{[2]} + \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]}) \\
&= \theta^{[2]} + \varphi^3\nu_0 - \varphi^2\alpha g(\theta^{[0]}) - \varphi\alpha g(\theta^{[1]}) - \alpha g(\theta^{[2]}) \\
&= \theta^{[2]} - \alpha(\varphi^2 g(\theta^{[0]}) + \varphi^1 g(\theta^{[1]}) + \varphi^0 g(\theta^{[2]})) + \varphi^3\nu_0 \\
\theta^{[t+1]} &= \theta^{[t]} - \alpha \sum_{j=0}^{t} \varphi^j g(\theta^{[t-j]}) + \varphi^{t+1}\nu_0
\end{aligned}
$$



MOMENTUM: EXAMPLE / 2

Suppose momentum always observes the same gradient $g(\theta)$:

$$\begin{aligned}
\theta^{[t+1]} &= \theta^{[t]} - \alpha \sum_{j=0}^{t} \varphi^j g(\theta^{[t-j]}) + \varphi^{t+1} \nu_0 \\
&= \theta^{[t]} - \alpha g(\theta) \sum_{j=0}^{t} \varphi^j + \varphi^{t+1} \nu_0 \\
&= \theta^{[t]} - \alpha g(\theta) \frac{1 - \varphi^{t+1}}{1 - \varphi} + \varphi^{t+1} \nu_0 \\
&\to \theta^{[t]} - \alpha g(\theta) \frac{1}{1 - \varphi} \qquad \text{for } t \to \infty.
\end{aligned}$$

Thus, momentum will accelerate in the direction of $-g(\theta)$ until reaching terminal velocity with step size:

$$-\alpha g(\theta)(1 + \varphi + \varphi^2 + \varphi^3 + ...) = -\alpha g(\theta) \frac{1}{1 - \varphi}$$

E.g. a momentum with $\varphi = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

MOMENTUM: ILLUSTRATION

The vector ν_3 (for $\nu_0 = 0$):

$$\begin{aligned} \nu_3 &= \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta^{[0]})) - \alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]}) \\ &= -\varphi^2(\alpha g(\theta^{[0]})) - \varphi(\alpha g(\theta^{[1]})) - \alpha g(\theta^{[2]}) \end{aligned}$$

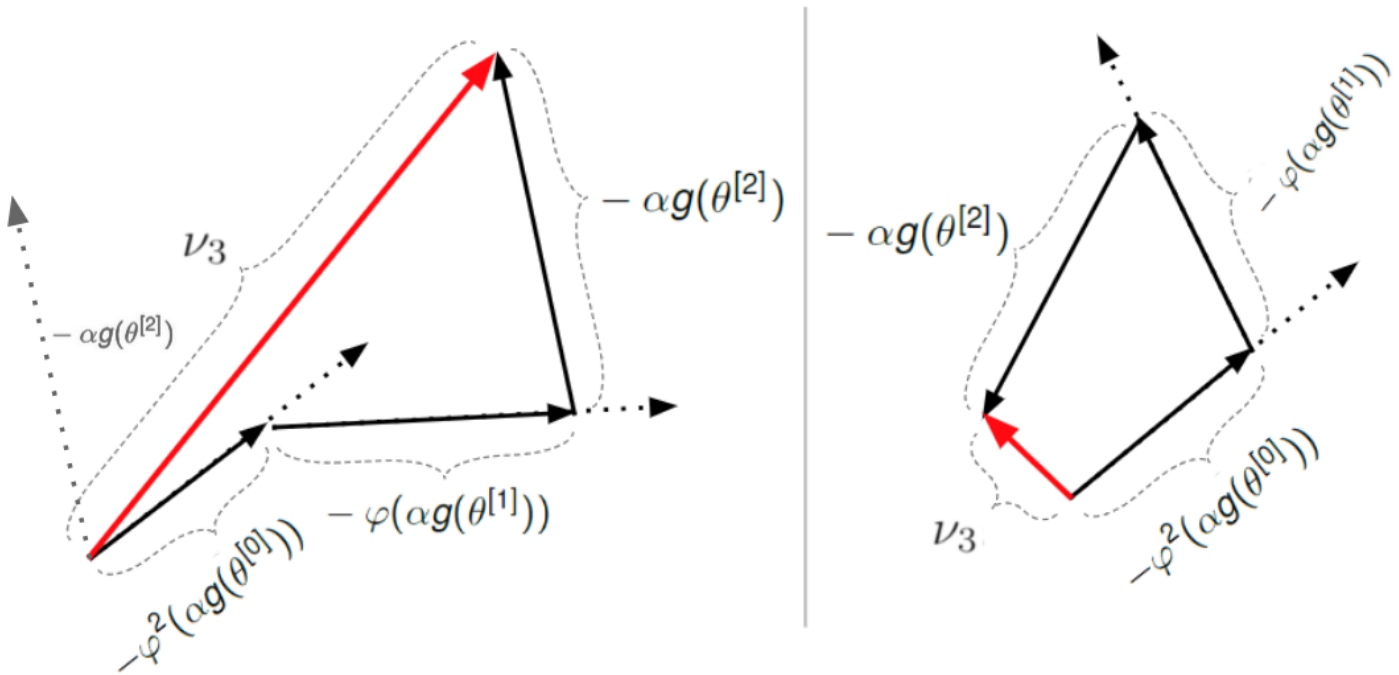


Figure: If consecutive (negative) gradients point mostly in the same direction, the velocity "builds up". On the other hand, if consecutive (negative) gradients point in very different directions, the velocity "dies down".

SGD WITH MOMENTUM

Algorithm Stochastic gradient descent with momentum

1: **require** learning rate α and momentum φ
2: **require** initial parameter $\boldsymbol{\theta}$ and initial velocity $\boldsymbol{\nu}$
3: **while** stopping criterion not met **do**
4: Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
5: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
6: Compute velocity update: $\boldsymbol{\nu} \leftarrow \varphi \boldsymbol{\nu} - \alpha \hat{\mathbf{g}}$
7: Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\nu}$
8: **end while**

SGD WITH MOMENTUM / 2

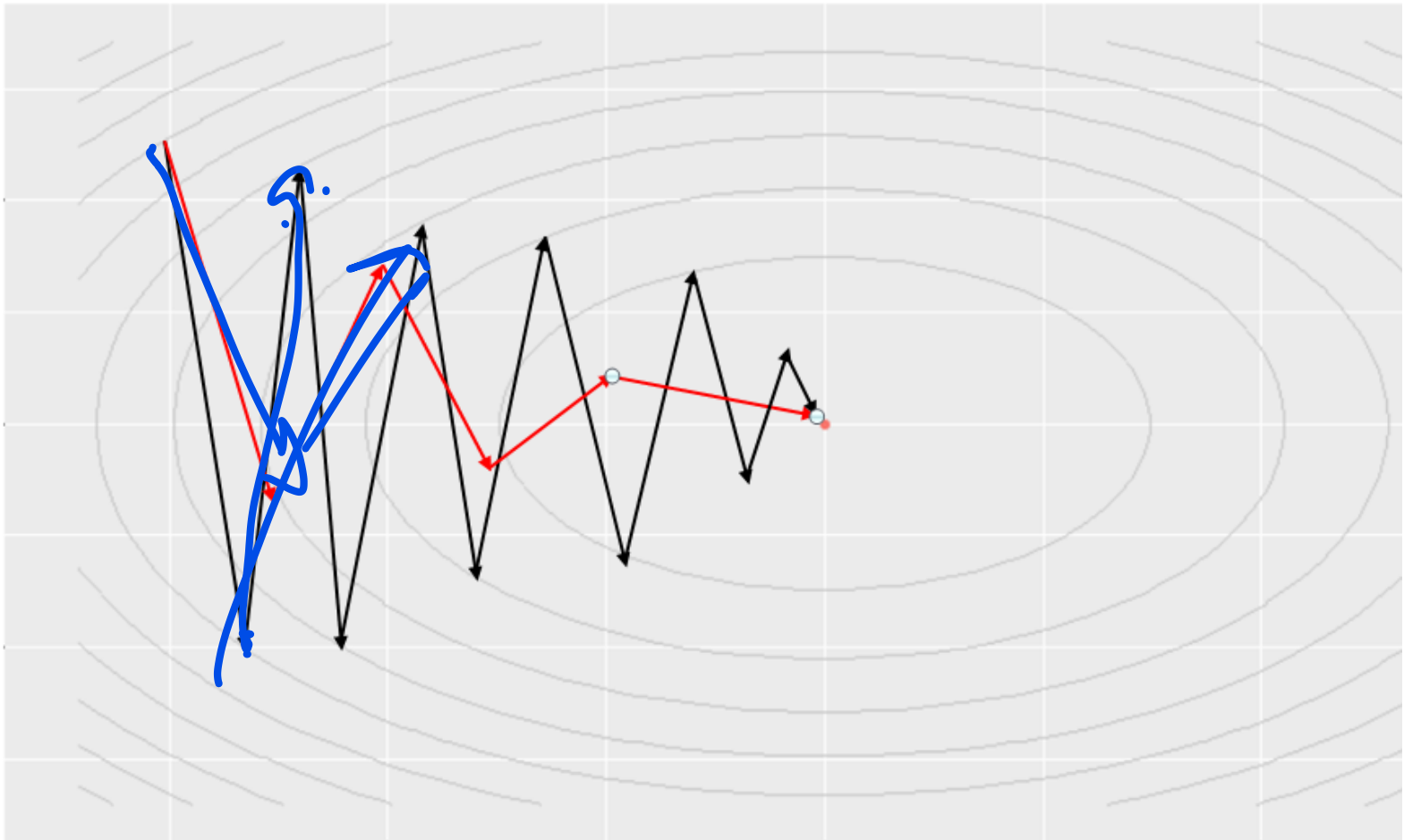


Figure: The contour lines show a quadratic loss function with a poorly conditioned Hessian matrix. The two curves show how standard gradient descent (black) and momentum (red) learn when dealing with ravines. Momentum reduces the oscillation and accelerates the convergence.

SGD WITH AND WITHOUT MOMENTUM

The following plot was created by our Shiny App. On the upper left you can explore different predefined examples. ▸ Click here

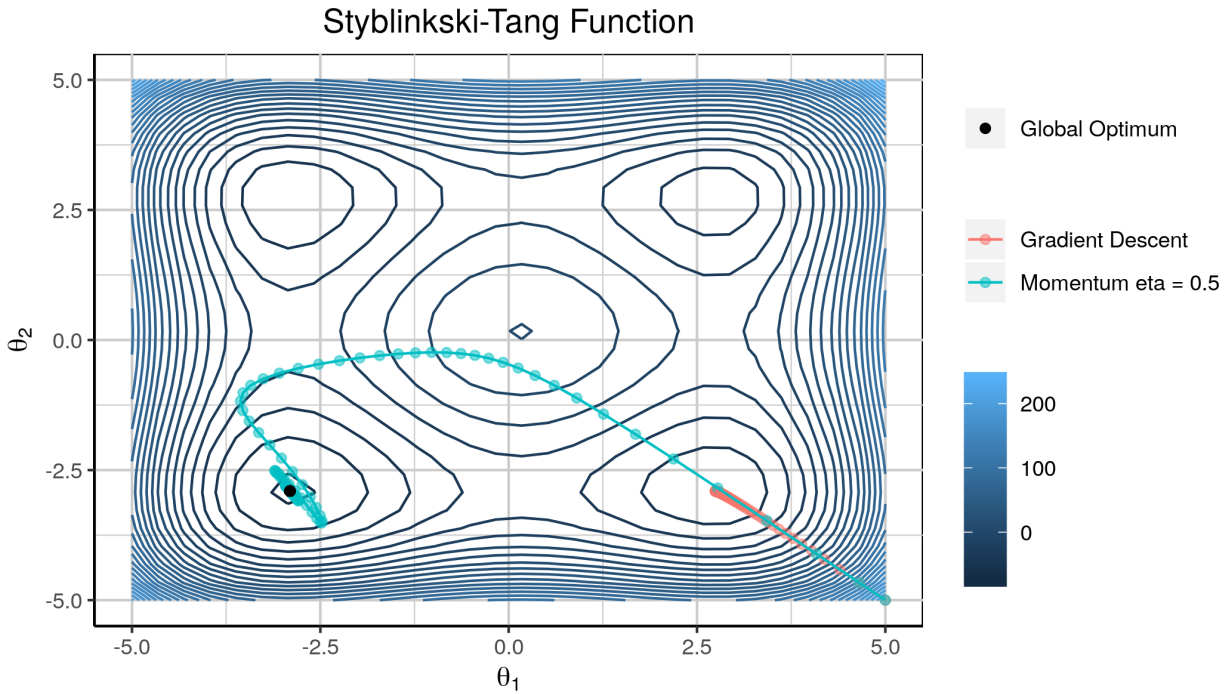
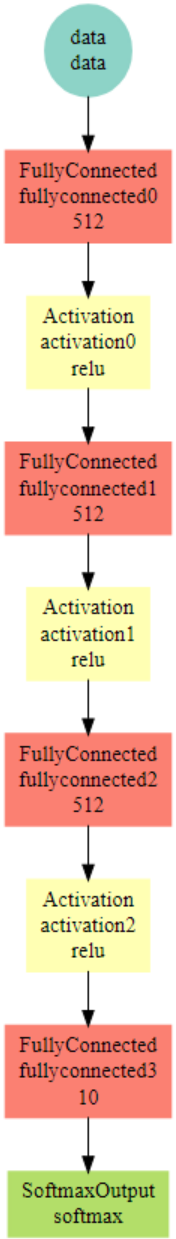


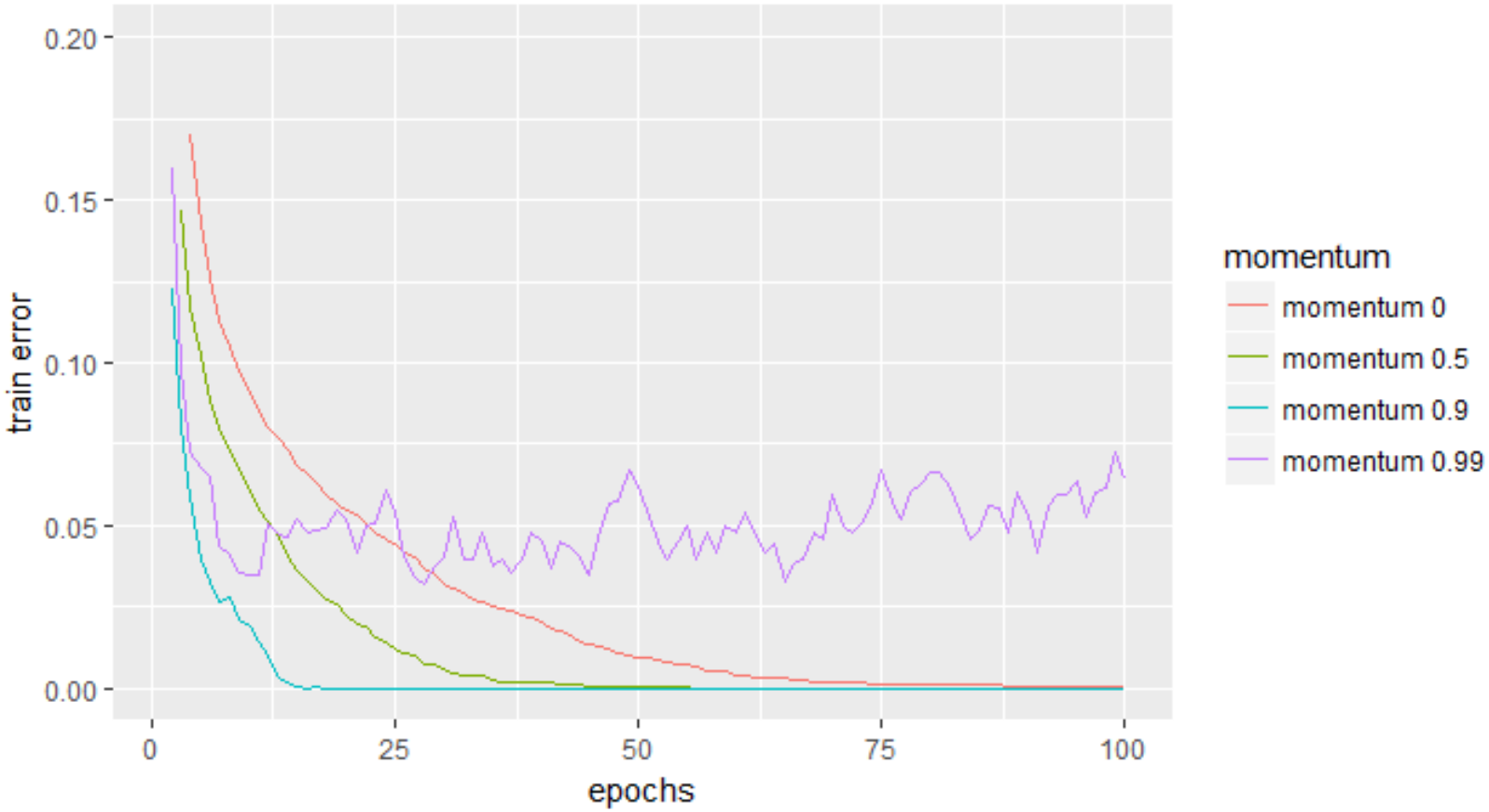
Figure: Comparison of SGD with and without momentum on the Styblinkski-Tang function. The black dot on the bottom left is the global optimum. We can see that SGD without momentum (red line/points) cannot escape the local minimum, while SGD with momentum (blue line/dots) is able to escape the local minimum and finds the global minimum.

MOMENTUM IN PRACTICE

- Lets try out different values of momentum (with SGD) on the MNIST data.
- We apply the same architecture we have used a dozen of times already (note that we used $\varphi = 0.9$ in all computations so far, i.e. in chapter 1 and 2)!

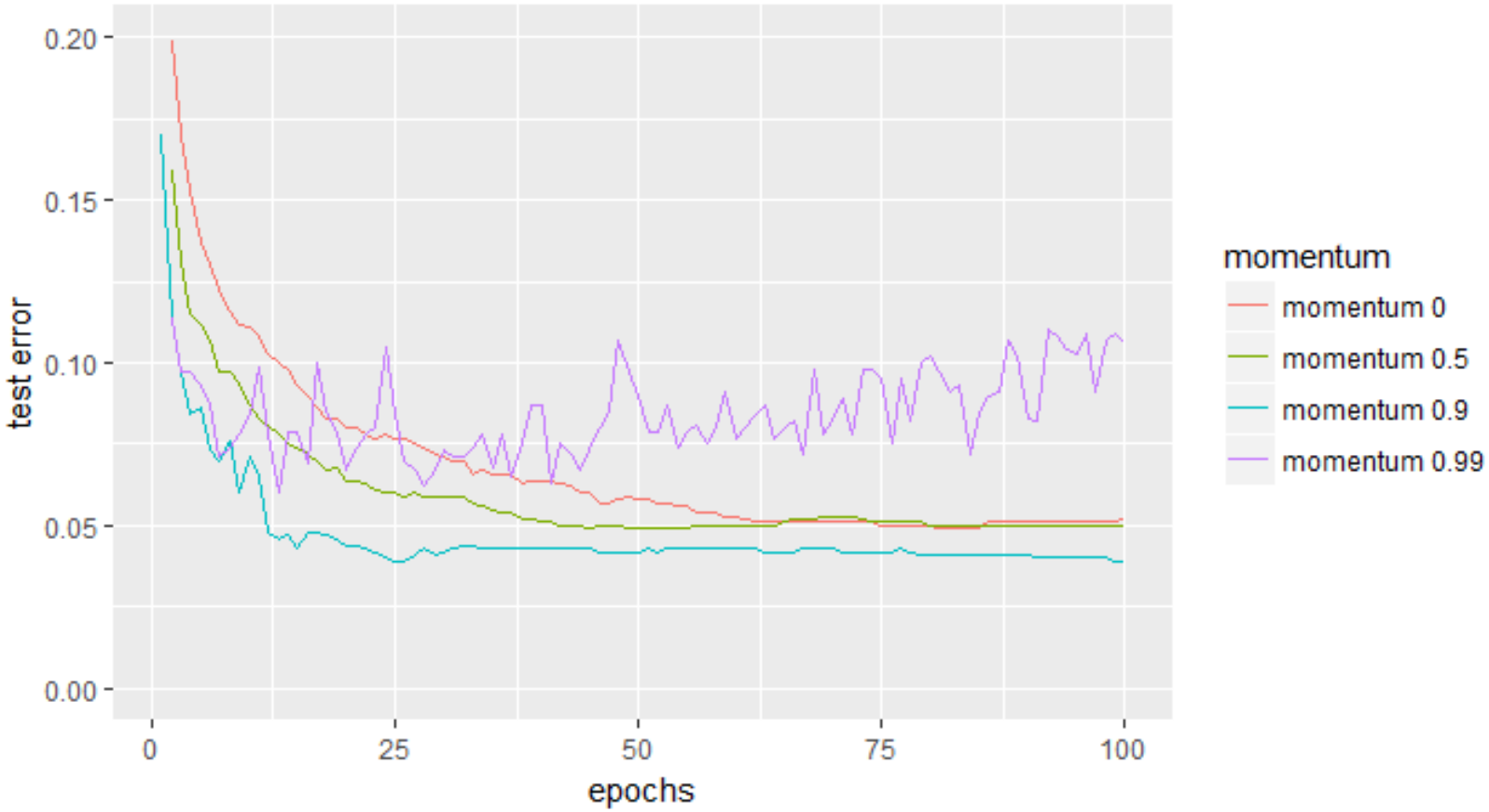


MOMENTUM IN PRACTICE



The higher momentum, the faster SGD learns the weights on the training data, but if momentum is too large, the training and test error fluctuates.

MOMENTUM IN PRACTICE / 2



The higher momentum, the faster SGD learns the weights on the training data, but if momentum is too large, the training and test error fluctuates.

NESTEROV MOMENTUM

- Momentum aims to solve poor conditioning of the Hessian but also variance in the stochastic gradient.
- Nesterov momentum modifies the algorithm such that the gradient is evaluated after the current velocity is applied:

$$\begin{aligned} \boldsymbol{\nu} &\leftarrow \varphi\boldsymbol{\nu} - \alpha\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_i L(y^{(i)}, f(x^{(i)}, \boldsymbol{\theta} + \varphi\boldsymbol{\nu}))\right] \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{\nu} \end{aligned}$$

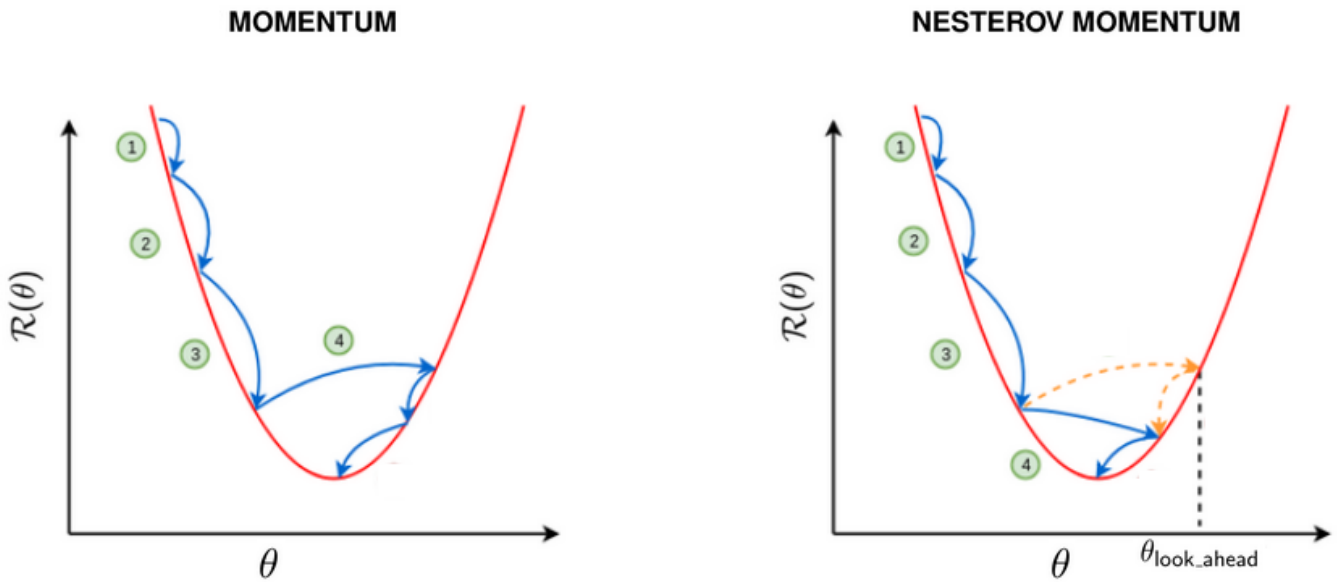
- We can interpret Nesterov momentum as an attempt to add a correction factor to the basic method.
- The method is also called Nesterov accelerated gradient (NAG).

SGD WITH NESTEROV MOMENTUM

Algorithm Stochastic gradient descent with Nesterov momentum

require learning rate α and momentum φ
require initial parameter $\boldsymbol{\theta}$ and initial velocity $\boldsymbol{\nu}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \varphi\boldsymbol{\nu}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m}\nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(y^{(i)}, f(x^{(i)}, \tilde{\boldsymbol{\theta}}))$
    Compute velocity update: $\boldsymbol{\nu} \leftarrow \varphi\boldsymbol{\nu} - \alpha\hat{\mathbf{g}}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\nu}$
end while

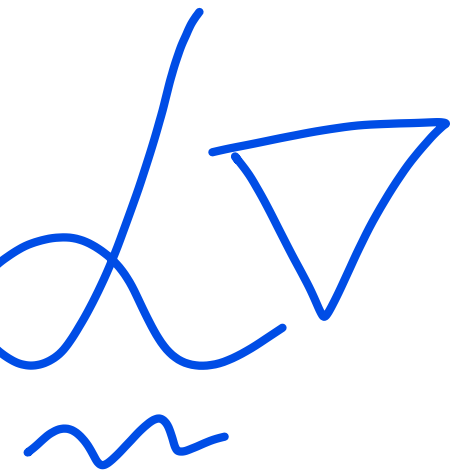
MOMENTUM VS. NESTEROV MOMENTUM



Credits: Chandra (2015)

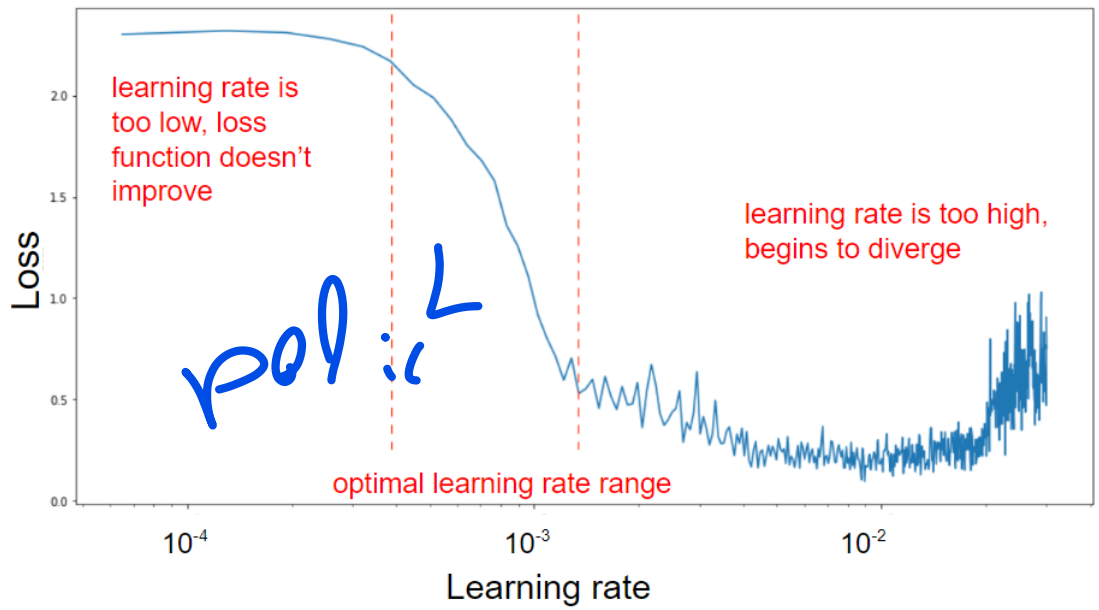
Figure: Comparison GD with momentum (left) and GD with Nesterov momentum (right) for one parameter θ. The first three updates of θ are very similar in both cases and the updates become larger due to momentum (accumulation of previous negative gradients). Update 4 is different. In case of momentum, the update overshoots as it makes an even bigger step due to the gradient history. In contrast, Nesterov momentum first evaluates a "look-ahead" point $\theta_{\text{look\_ahead}}$, detects that it overshoots, and slightly reduces the overall magnitude of the fourth update. Thus, Nesterov momentum reduces overshooting and leads to smaller oscillations than momentum.

Learning Rates



LEARNING RATE

- The learning rate is a very important hyperparameter.
- To systematically find a good learning rate, we can start at a very low learning rate and gradually increase it (linearly or exponentially) after each mini-batch.
- We can then plot the learning rate and the training loss for each batch.
- A good learning rate is one that results in a steep decline in the loss.



Credit: jeremyjordan

LEARNING RATE SCHEDULE

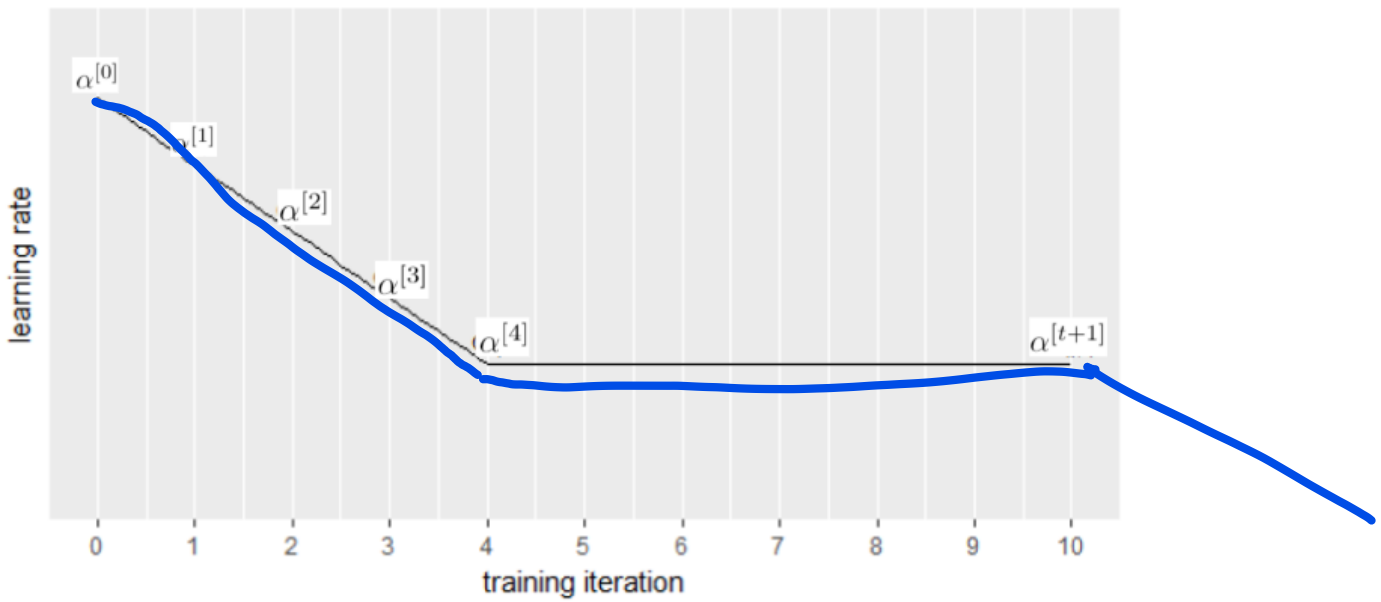
- We would like to force convergence until reaching a local minimum.
- Applying SGD, we have to decrease the learning rate over time, thus $\alpha^{[t]}$ (learning rate at training iteration *t*).
  - The estimator $\hat{\mathbf{g}}$ is computed based on small batches.
  - Random sampling *m* training samples introduces noise, that does not vanish even if we find a minimum.
- In practice, a common strategy is to decay the learning rate linearly over time until iteration τ:

$$\alpha^{[t]} = \begin{cases} \left(1 - \frac{t}{\tau}\right)\alpha^{[0]} + \frac{t}{\tau}\alpha^{[\tau]} = t\left(-\frac{\alpha^{[0]}+\alpha^{[\tau]}}{\tau}\right) + \alpha^{[0]} & \text{for } t \leq \tau \\ \alpha^{[\tau]} & \text{for } t > \tau \end{cases}$$

LEARNING RATE SCHEDULE / 2

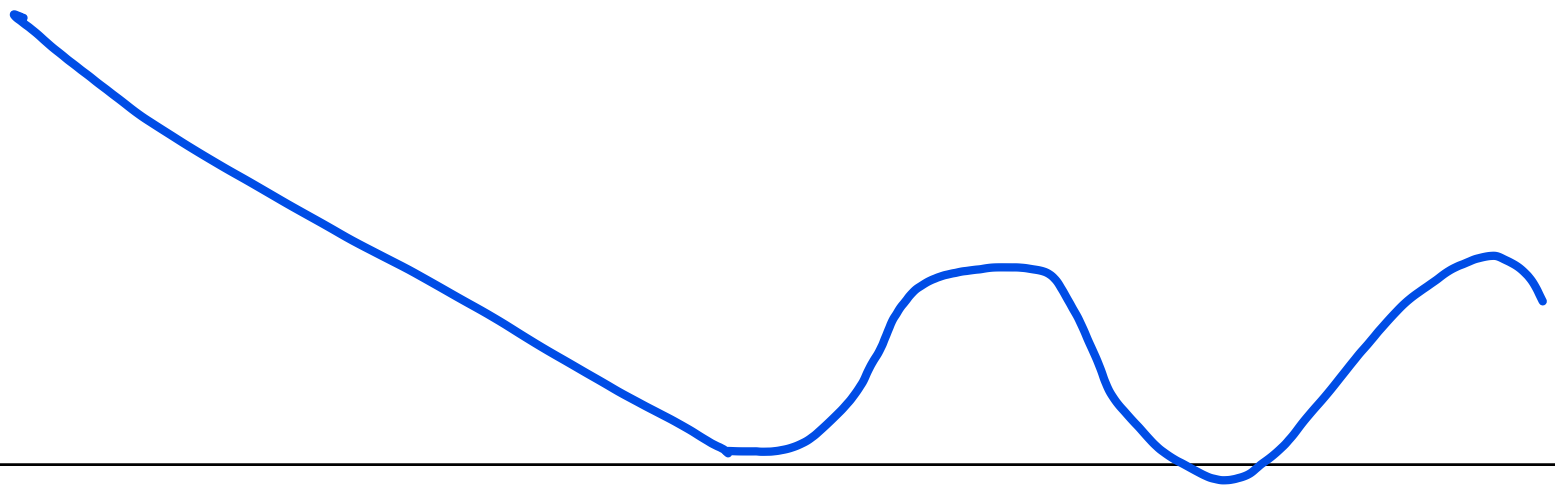
Example for $\tau = 4$:

iteration t	t/τ	$\alpha^{[t]}$
1	0.25	$\left(1 - \frac{1}{4}\right)\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]} = \frac{3}{4}\alpha^{[0]} + \frac{1}{4}\alpha^{[\tau]}$
2	0.5	$\frac{2}{4}\alpha^{[0]} + \frac{2}{4}\alpha^{[\tau]}$
3	0.75	$\frac{1}{4}\alpha^{[0]} + \frac{3}{4}\alpha^{[\tau]}$
4	1	$0 + \alpha^{[\tau]}$
...		$\alpha^{[\tau]}$
$t+1$		$\alpha^{[\tau]}$



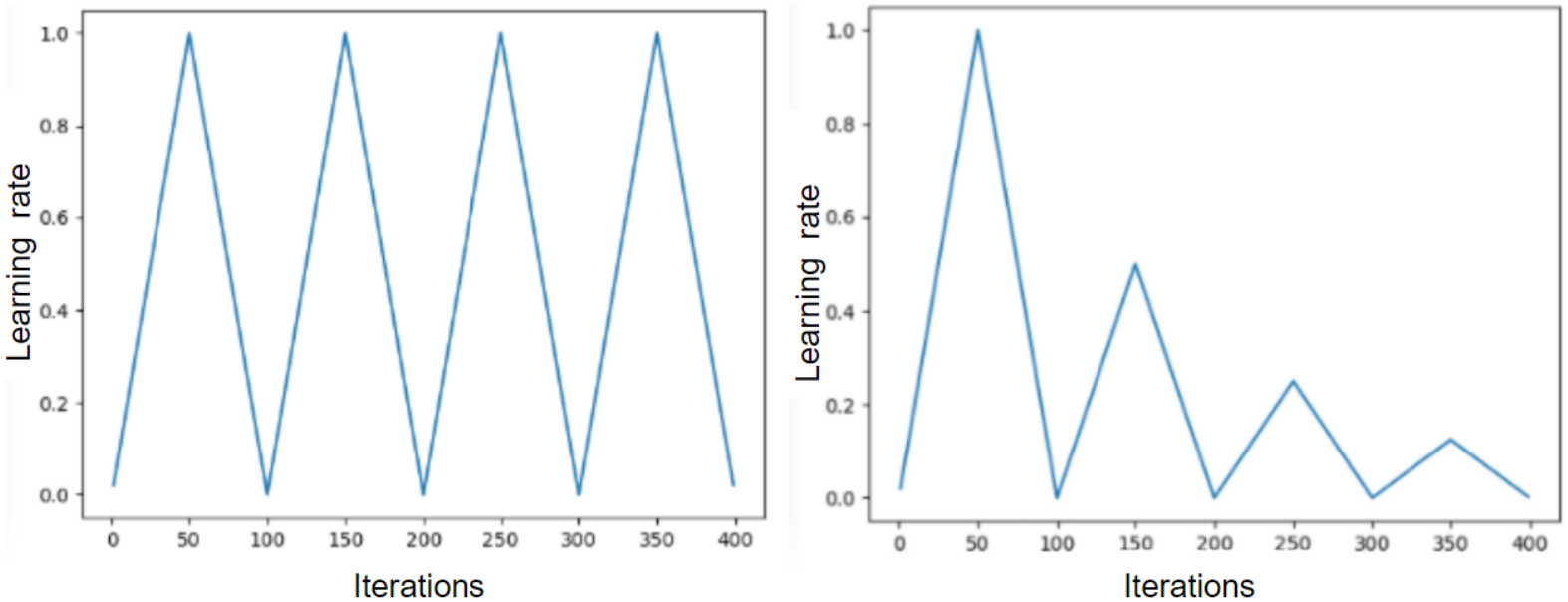
CYCLICAL LEARNING RATES

- Another option is to have a learning rate that periodically varies according to some cyclic function.
- Therefore, if training does not improve the loss anymore (possibly due to saddle points), increasing the learning rate makes it possible to rapidly traverse such regions.
- Recall, saddle points are far more likely than local minima in deep nets.
- Each cycle has a fixed length in terms of the number of iterations.



CYCLICAL LEARNING RATES

- One such cyclical function is the "triangular" function.

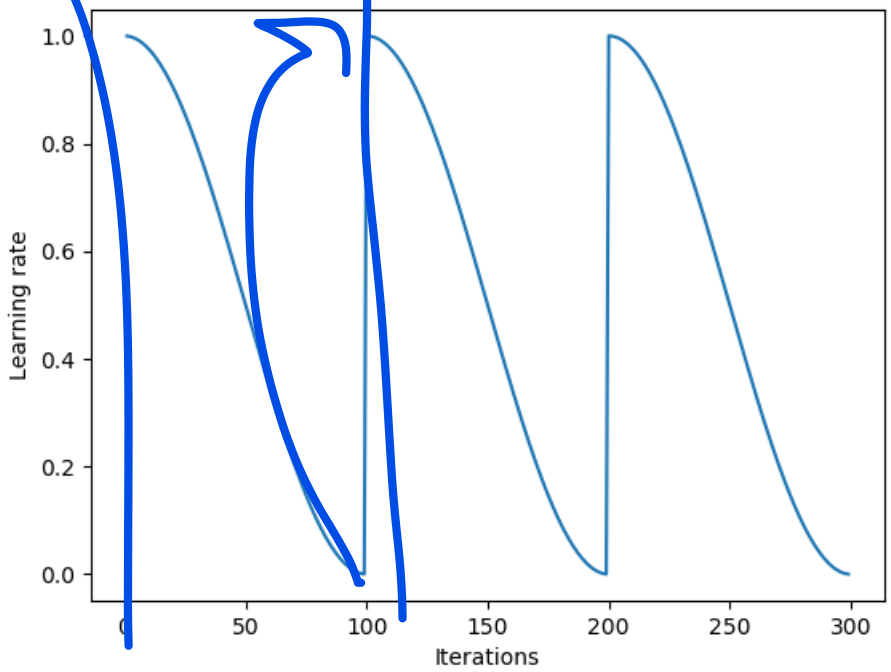


Credit: Hafidz Zulkifli

- In the right image, the range is cut in half after each cycle.

CYCLICAL LEARNING RATES

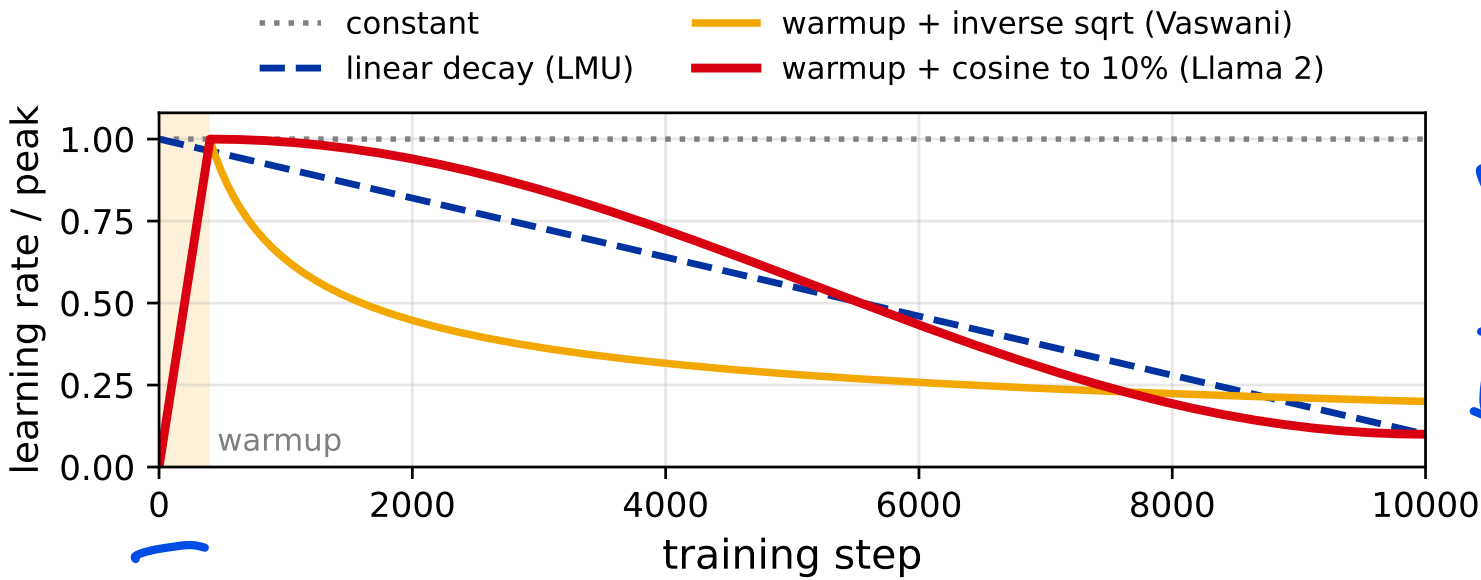
- Yet another option is to abruptly "restart" the learning rate after a fixed number of iterations.
- Loshchilov et al. (2016) proposed "cosine annealing" (between restarts).



Credit: Hafidz Zulkifli

Warmup, then decay: the schedule modern models use

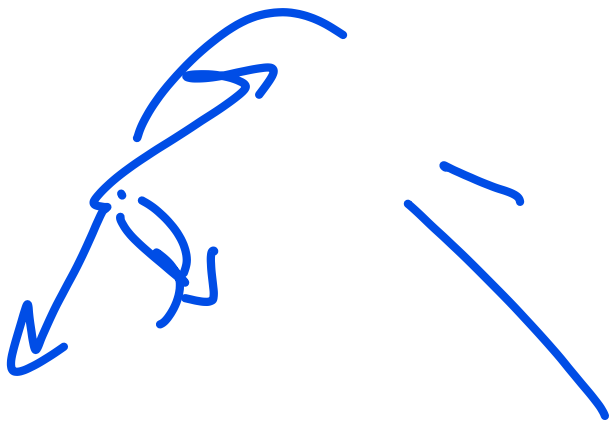
One piece LMU's schedule pages skip: **warmup**.



- **Warmup:** start near 0 and ramp up linearly. The network changes fastest early in training, and full-size steps there can derail it (Goyal et al., 2017). The original transformer warmed up for 4000 steps (Vaswani et al., 2017).
- **Then decay:** big steps to make progress, small steps to settle. Llama 2 used 2000 warmup steps, then cosine decay to 10% of the peak (Touvron et al., 2023).

Shapes on a made-up 10,000-step run; the step counts above are the papers' own.

Algorithms with Adaptive Learning Rates



ADAPTIVE LEARNING RATES

- The learning rate is reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on the models performance.
- Naturally, it might make sense to use a different learning rate for each parameter, and automatically adapt them throughout the training process.

ADAGRAD

- Adagrad adapts the learning rate to the parameters.
- In fact, Adagrad scales learning rates inversely proportional to the square root of the sum of the past squared derivatives.
  - Parameters with large partial derivatives of the loss obtain a rapid decrease in their learning rate.
  - Parameters with small partial derivatives on the other hand obtain a relatively small decrease in their learning rate.
- For that reason, Adagrad might be well suited when dealing with sparse data.
- Goodfellow et al. (2016) say that the accumulation of squared gradients can result in a premature and overly decrease in the learning rate.

ADAGRAD / 2

Algorithm Adagrad

require Global learning rate α
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ (division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

- "$\odot$" is called Hadamard or element-wise product.
- Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, \quad \text{then } A \odot B = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 6 \\ 3 \cdot 7 & 4 \cdot 8 \end{bmatrix}$$

RMSPROP

- RMSprop is a modification of Adagrad.
- It's intention is to resolve Adagrad's radically diminishing learning rates.
- The gradient accumulation is replaced by an exponentially weighted moving average.
- Theoretically, that leads to performance gains in non-convex scenarios.
- Empirically, RMSProp is a very effective optimization algorithm. Particularly, it is employed routinely by deep learning practitioners.

RMSPROP / 2

Algorithm RMSProp

require Global learning rate α and decay rate $\rho \in [0, 1)$
require Initial parameter $\boldsymbol{\theta}$
require Small constant β, perhaps 10^{-6}, for numerical stability
Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Accumulate squared gradient $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
    Compute update: $\nabla\boldsymbol{\theta} = -\frac{\alpha}{\beta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

ADAM

- Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
- Adam uses the first and the second moments of the gradients.
  - Adam keeps an exponentially decaying average of past gradients (first moment).
  - Like RMSProp it stores an exponentially decaying average of past squared gradients (second moment).
  - Thus, it can be seen as a combination of RMSProp and momentum.
- Basically Adam uses the combined averages of previous gradients at different moments to give it more “persuasive power” to adaptively update the parameters.

ADAM / 2

Algorithm Adam

require Step size α (suggested default: 0.001)
require Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (suggested defaults: 0.9 and 0.999 respectively)
require Small constant β (suggested default 10^{-8})
require Initial parameters $\boldsymbol{\theta}$
Initialize time step $t = 0$
Initialize 1st and 2nd moment variables $\mathbf{s}^{[0]} = 0, \mathbf{r}^{[0]} = 0$
while stopping criterion not met **do**
    $t \leftarrow t + 1$
    Sample a minibatch of m examples from the training set $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(m)}\}$
    Compute gradient estimate: $\hat{\mathbf{g}}^{[t]} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(y^{(i)}, f\left(\tilde{\boldsymbol{x}}^{(i)} \mid \boldsymbol{\theta}\right)\right)$
    Update biased first moment estimate: $\mathbf{s}^{[t]} \leftarrow \rho_1 \mathbf{s}^{[t-1]} + (1 - \rho_1)\hat{\mathbf{g}}^{[t]}$
    Update biased second moment estimate: $\mathbf{r}^{[t]} \leftarrow \rho_2 \mathbf{r}^{[t-1]} + (1 - \rho_2)\hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}$
    Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}^{[t]}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}^{[t]}}{1-\rho_2^t}$
    Compute update: $\nabla\boldsymbol{\theta} = -\alpha \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\beta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla\boldsymbol{\theta}$
end while

ADAM / 3

- Adam initializes the exponentially weighted moving averages $\mathbf{s}$ and $\mathbf{r}$ as $\mathbf{0}$ (zero) vectors.
- As a result, they are biased towards zero.
- This means $\mathbb{E}[\mathbf{s}^{[t]}] \neq \mathbb{E}[\hat{\mathbf{g}}^{[t]}]$ and $\mathbb{E}[\mathbf{r}^{[t]}] \neq \mathbb{E}[\hat{\mathbf{g}}^{[t]} \odot \hat{\mathbf{g}}^{[t]}]$ (where the expectations are calculated over minibatches).
- To see this, let us unroll the computation of $\mathbf{s}^{[t]}$ for a few time-steps:

$$\mathbf{s}^{[0]} = 0$$

$$\mathbf{s}^{[1]} = \rho_1 \mathbf{s}^{[0]} + (1 - \rho_1)\hat{\mathbf{g}}^{[1]} = (1 - \rho_1)\hat{\mathbf{g}}^{[1]}$$

$$\mathbf{s}^{[2]} = \rho_1 \mathbf{s}^{[1]} + (1 - \rho_1)\hat{\mathbf{g}}^{[2]} = \rho_1(1 - \rho_1)\hat{\mathbf{g}}^{[1]} + (1 - \rho_1)\hat{\mathbf{g}}^{[2]}$$

$$\mathbf{s}^{[3]} = \rho_1 \mathbf{s}^{[2]} + (1 - \rho_1)\hat{\mathbf{g}}^{[3]} = \rho_1^2(1 - \rho_1)\hat{\mathbf{g}}^{[1]} + \rho_1(1 - \rho_1)\hat{\mathbf{g}}^{[2]} + (1 - \rho_1)\hat{\mathbf{g}}^{[3]}$$

- Therefore, $\mathbf{s}^{[t]} = (1 - \rho_1) \sum_{i=1}^{t} \rho_1^{t-i} \mathbf{g}^{[i]}$.
- Note that the contribution of the earlier $\hat{\mathbf{g}}^{[i]}$ to the moving average shrinks rapidly.

ADAM / 4

- The expected value of $\mathbf{s}^{[t]}$ is:

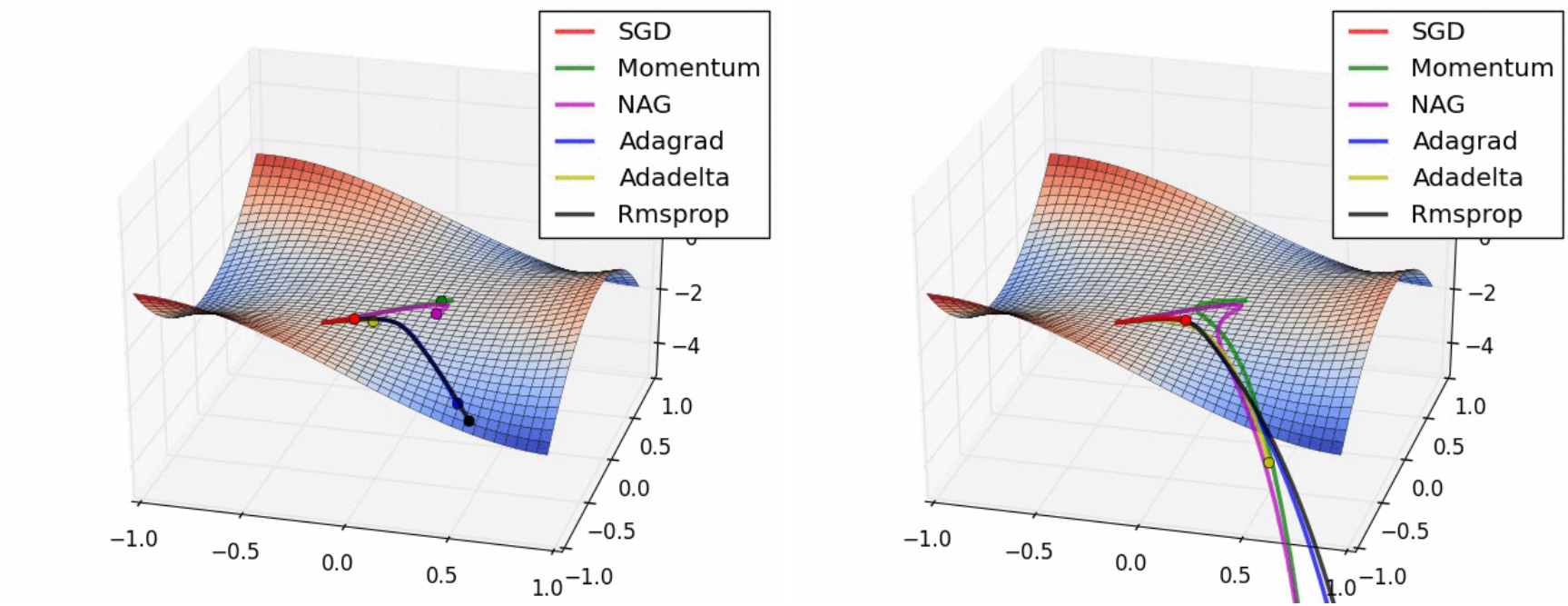
$$\mathbb{E}[\mathbf{s}^{[t]}] = \mathbb{E}[(1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i}\hat{\mathbf{g}}^{[i]}]$$

$$= \mathbb{E}[\hat{\mathbf{g}}^{[t]}](1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i} + \zeta$$

$$= \mathbb{E}[\hat{\mathbf{g}}^{[t]}](1-\rho_1^t) + \zeta$$

  where we approximate $\hat{\mathbf{g}}^{[i]}$ with $\hat{\mathbf{g}}^{[t]}$ which allows us to move it outside the sum. ζ is the error that results from this approximation.
- Therefore, $\mathbf{s}^{[t]}$ is a biased estimator of $\hat{\mathbf{g}}^{[t]}$ and the effect of the bias vanishes over the time-steps (because $\rho_1^t \to 0$ for $t \to \infty$).
- Ignoring ζ (as it can be kept small), we correct for the bias by setting $\hat{\mathbf{s}}^{[t]} = \frac{\mathbf{s}^{[t]}}{(1-\rho_1^t)}$.
- Similarly, we set $\hat{\mathbf{r}}^{[t]} = \frac{\mathbf{r}^{[t]}}{(1-\rho_2^t)}$.

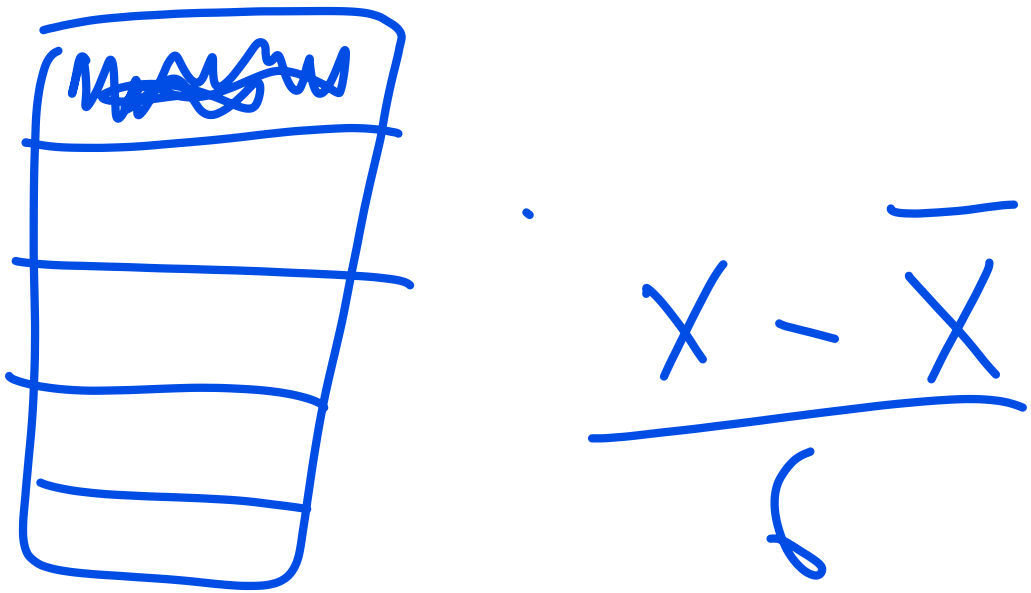
COMPARISON OF OPTIMIZERS: ANIMATION



Credits: Dettmers (2015) and Radford

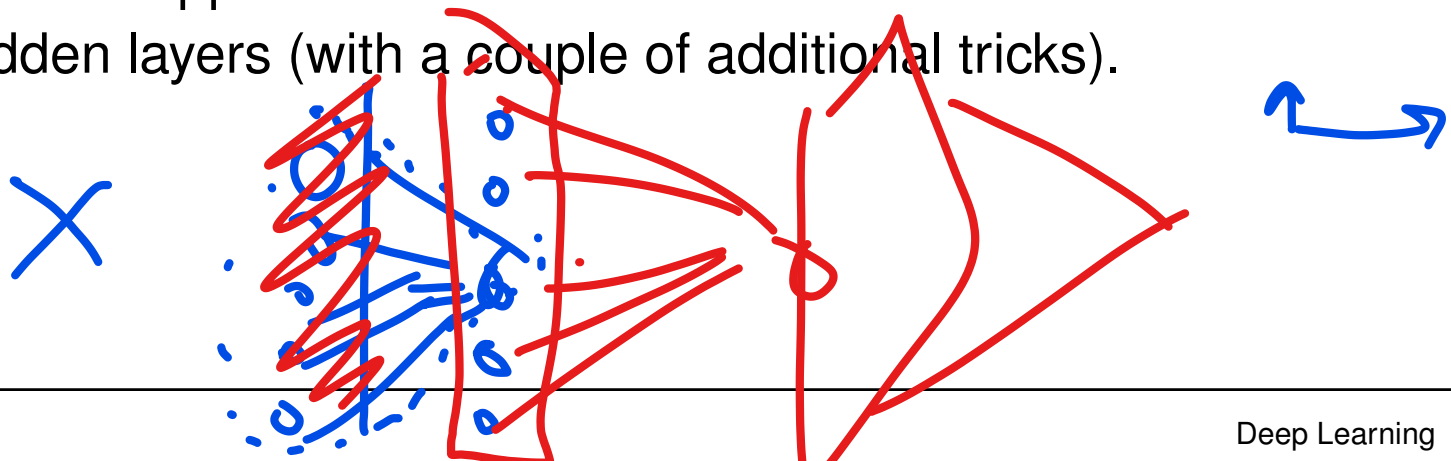
Figure: Excerpts from an animation to compare the behavior of momentum and other methods compared to SGD for a saddle point. Left: After a few seconds; Right: A bit later. The animation shows that all showed methods accelerate optimization compared to the standard SGD. The highest acceleration is obtained using Rmsprop followed by Adagrad as learning rate strategies. You can find the animation here or click on the images above.

Batch Normalization



BATCH NORMALIZATION

- Batch Normalization (BatchNorm) is an extremely popular technique that improves the training speed and stability of deep neural nets.
- It is an extra component that can be placed between each layer of the neural network.
- It works by changing the "distribution" of activations at each hidden layer of the network.
- We know that it is sometimes beneficial to normalize the inputs to a learning algorithm by shifting and scaling all the features so that they have 0 mean and unit variance.
- BatchNorm applies a similar transformation to the activations of the hidden layers (with a couple of additional tricks).



BATCH NORMALIZATION

- For a hidden layer with neurons z_j, $j = 1, \ldots, J$, BatchNorm is applied to each z_j by considering the activations of z_j **over a given minibatch** of inputs.
- Let $z_j^{(i)}$ denote the activation of z_j for input $x^{(i)}$ in the minibatch (of size m).
- The mean and variance of the activations are

$$\mu_j = \frac{1}{m} \sum_i^m z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_i^m (z_j^{(i)} - \mu_j)^2$$

- Each $z_j^{(i)}$ is then normalized

$$\tilde{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

where a small constant, ϵ, is added for numerical stability.

BATCH NORMALIZATION

- It may not be desirable to normalize the activations in such a rigid way because potentially useful information can be lost in the process.
- Therefore, we commonly let the training algorithm decide the "right amount" of normalization by allowing it to re-shift and re-scale $\tilde{z}_j^{(i)}$ to arrive at the batch normalized activation $\hat{z}_j^{(i)}$:

$$\hat{z}_j^{(i)} = \gamma_j \tilde{z}_j^{(i)} + \beta_j$$

- γ_j and β_j are learnable parameters that are also tweaked by backpropogation.
- $\hat{z}_j^{(i)}$ then becomes the input to the next layer.
- Note: The algorithm is free to scale and shift each $\tilde{z}_j^{(i)}$ back to its original (unnormalized) value.

BATCH NORMALIZATION: ILLUSTRATION

- Recall: $z_j = \sigma(W_j^T x + b_j)$
- So far, we have applied batch-norm to the activation z_j. It is possible (and more common) to apply batch norm to $W_j^T x + b_j$ *before* passing it to the nonlinear activation σ.

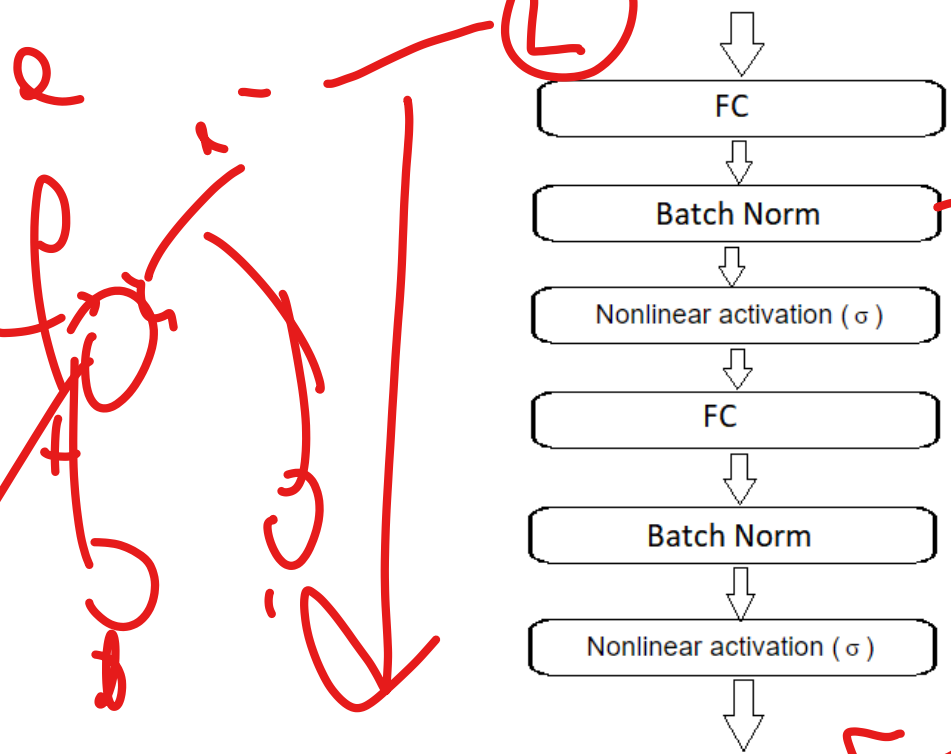
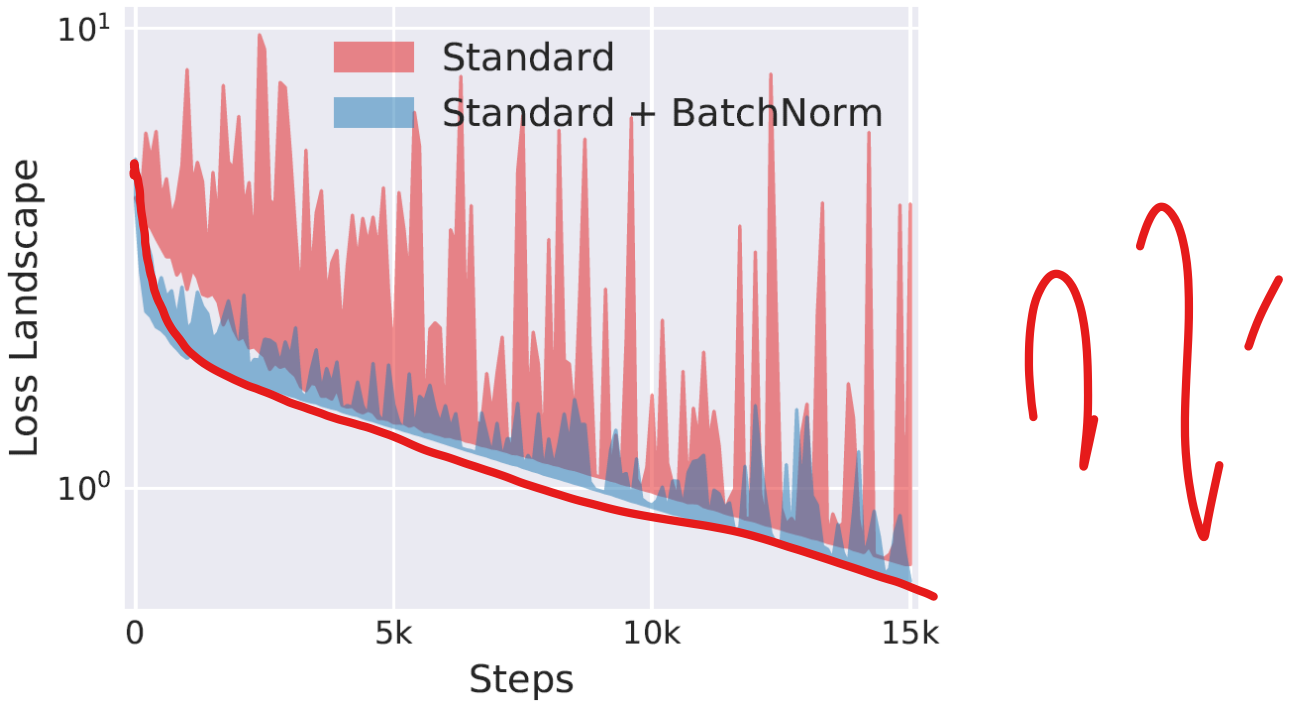


Figure: FC = Fully Connected layer. BatchNorm is applied *before* the nonlinear activation function.

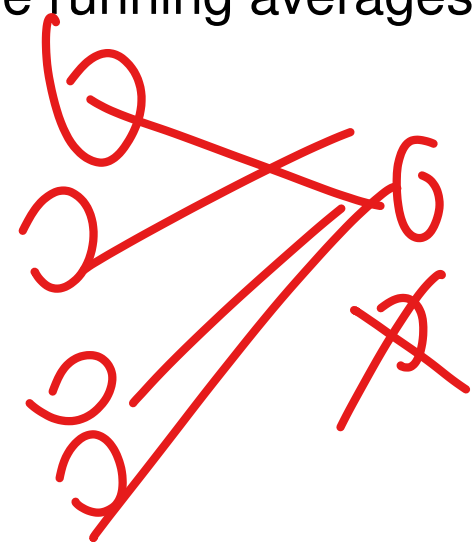
BATCH NORMALIZATION

- The key impact of BatchNorm on the training process is this: It reparametrizes the underlying optimization problem to **make its landscape significantly more smooth**.
- One aspect of this is that the loss changes at a smaller rate and the magnitudes of the gradients are also smaller (see Santurkar et al. 2018).



BATCH NORMALIZATION: PREDICTION

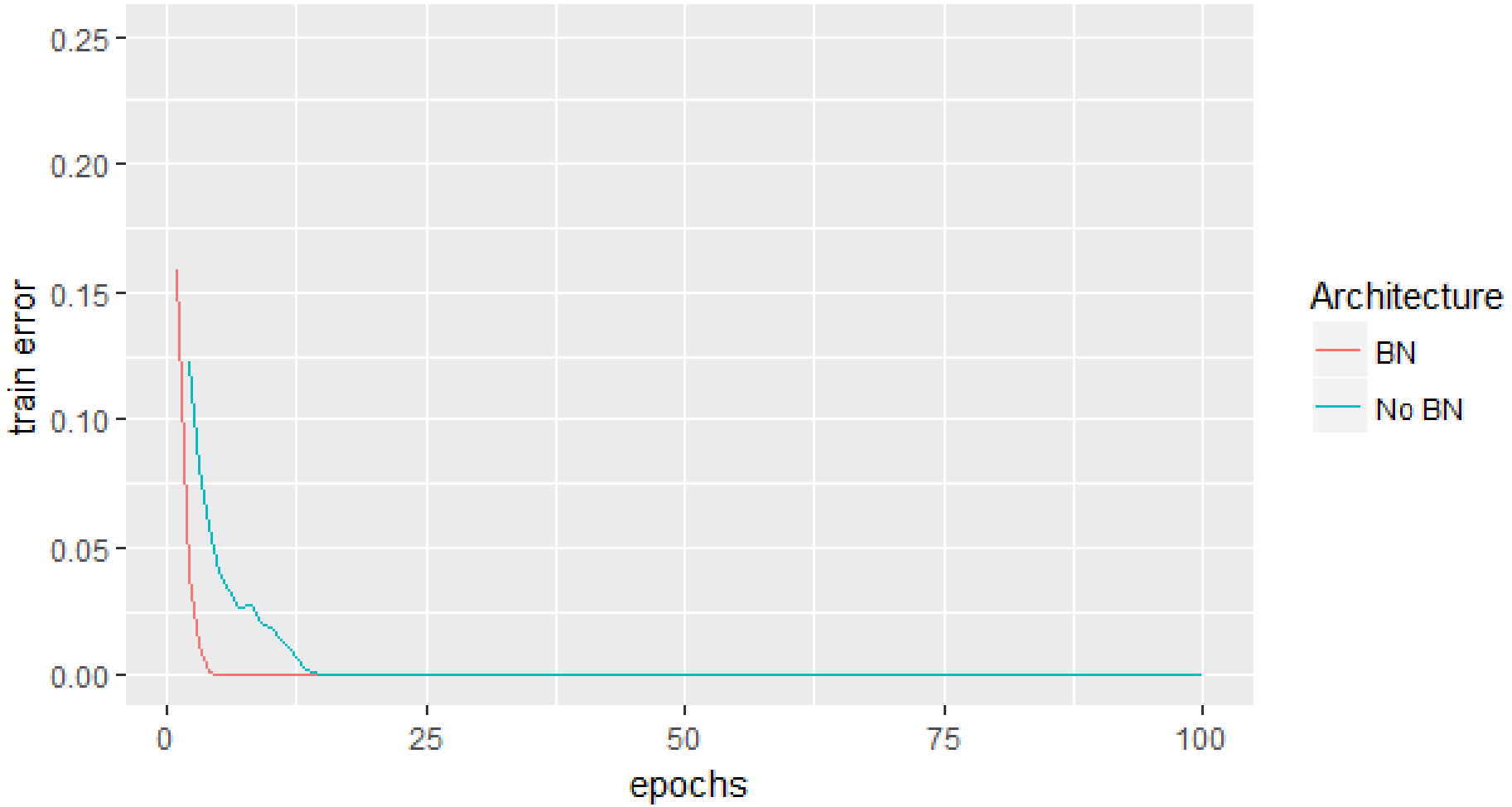
- Once the network has been trained, how can we generate a prediction for a single input (either at test time or in production)?
- One option is to feed the entire training set to the (trained) network and compute the means and standard deviations.
- More commonly, during training, an exponentially weighted running average of each of these statistics over the minibatches is maintained.
- The learned γ and β parameters are then used (in conjunction with the running averages) to generate the output.



BATCH NORMALIZATION

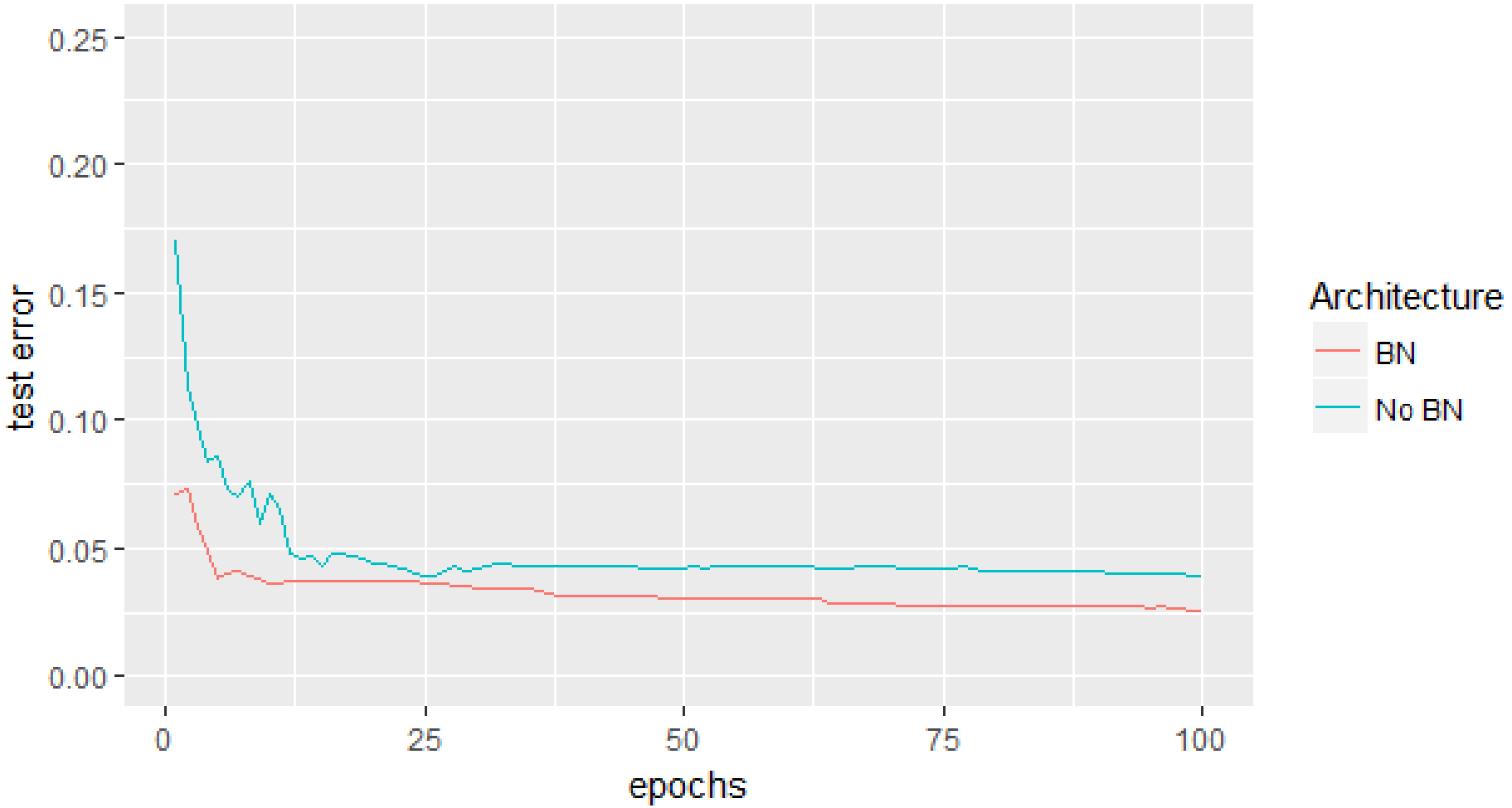
- For our final benchmark in this chapter we compute two models to predict the mnist data.
- One will extend our basic architecture such that we add batch normalization to all hidden layers.
- We use SGD as optimizer with a momentum of 0.9, a learning rate of 0.03 and weight decay of 0.001.

BATCH NORMALIZATION / 2



Batch Normalization accelerated learning.

BATCH NORMALIZATION / 3



Batch Normalization resulted in a lower test error.

REFERENCES

Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)

Deep Learning

http://www.deeplearningbook.org/

Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, Yoshua Bengio (2014)

Identifying and attacking the saddle point problem in high-dimensional non-convex optimization

https://arxiv.org/abs/1406.2572

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein (2017)

Visualizing the Loss Landscape of Neural Nets

https://arxiv.org/abs/1712.09913

Tim Dettmers (2015)

Deep Learning in a Nutshell: History and Training

https://devblogs.nvidia.com/deep-learning-nutshell-history-training/

REFERENCES / 2

Hafidz Zulkifli (2018)

Understanding Learning Rates and How It Improves Performance in Deep Learning

https://towardsdatascience.com

Ilya Loshchilov, Frank Hutter (2016)

SGDR: Stochastic Gradient Descent with Warm Restarts

https://arxiv.org/abs/1608.03983

Jeremy Jordan (2018)

Setting the learning rate of your neural network

https://www.jeremyjordan.me/nn-learning-rate/

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, Aleksander Madry (2018)

How Does Batch Normalization Help Optimization?

https://arxiv.org/abs/1805.11604

REFERENCES / 3

Akshay Chandra (2015)

Learning Parameters, Part 2: Momentum-Based & Nesterov Accelerated Gradient Descent

https://towardsdatascience.com/learning-parameters-part-2-a190bef2d12

Which optimizer? A practical cheat-sheet

LMU derived SGD, momentum, Nesterov, AdaGrad, RMSProp, Adam and BatchNorm. In practice the decision is short:

- **Start with AdamW** (ρ_1=0.9, ρ_2=0.999, α=10^{-3}): Adam with *decoupled* weight decay. Adam's built-in L2 penalty gets rescaled weight by weight; AdamW decays every weight at the same rate (next three frames). This is the modern default, not plain Adam.
- **SGD + momentum** (φ=0.9) can generalize *better* once tuned – common for large-scale vision.
- **Add a schedule** (warmup + cosine decay) when squeezing the last few points.
- **Add BatchNorm** between layers: it smooths the loss landscape, so training is faster and tolerates higher learning rates. Its sibling **LayerNorm** normalises each example over its *own* features instead of over the batch – the norm transformers use (LLM chapter).

The full math (momentum, Adam's moment estimates) is here and in the course's *optimization module*. The practical default: **AdamW, then tune.**

AdamW (1/3): two ways to shrink the weights

The regularization deck treated the *L2 penalty* and *weight decay* as one idea. They are two different recipes:

L2 penalty add $\frac{\lambda}{2}\|\boldsymbol{\theta}\|^2$ to the loss, so the *gradient* becomes $\nabla L + \lambda\boldsymbol{\theta}$

weight decay leave the gradient alone and shrink the *weights* directly: $\boldsymbol{\theta} \leftarrow (1 - \alpha\lambda)\,\boldsymbol{\theta}$

With plain SGD they give exactly the **same update**:

$$\boldsymbol{\theta} - \alpha\,(\nabla L + \lambda\boldsymbol{\theta}) \;=\; (1 - \alpha\lambda)\,\boldsymbol{\theta} - \alpha\,\nabla L$$

Example: $\alpha = 0.1$, $\lambda = 0.01$. Every step first multiplies *every* weight by 0.999, whatever its gradient, then takes the usual gradient step.

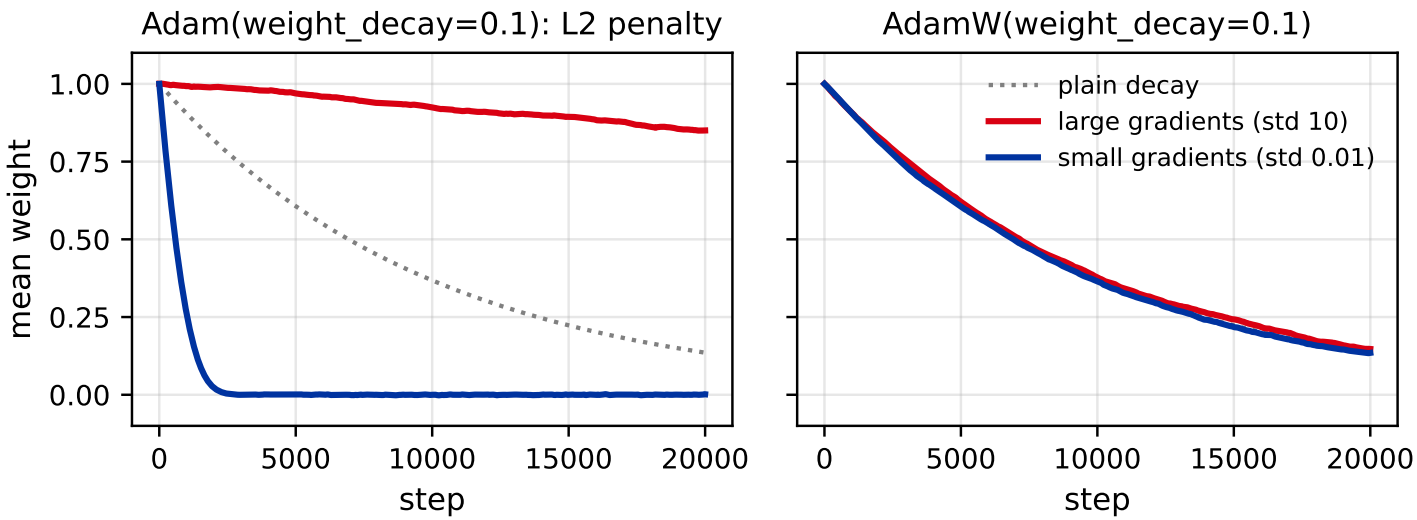
That is why the two names were used interchangeably for years. It stops being true as soon as the optimizer does more to the gradient than multiply it by α – and Adam does.

AdamW (2/3): Adam breaks the equivalence

Adam divides each weight's step by $\sqrt{\hat{r}_i}$, the size of *that weight's* past gradients. With an L2 penalty the decay term $\lambda\theta_i$ sits inside the gradient, so it is divided too:

$$\Delta\theta_i = -\alpha \frac{\hat{s}_i}{\sqrt{\hat{r}_i} + \beta}, \qquad \hat{s}_i \approx \overline{\nabla_i L} + \lambda\theta_i \quad \Rightarrow \quad \text{decay} \approx \alpha \frac{\lambda\,\theta_i}{\sqrt{\hat{r}_i}}$$

Large gradients, big divisor, **little decay**. Small gradients, tiny divisor, **strong decay**:



PyTorch's own optimizers; two groups of 100 weights start at 1 and get pure-noise gradients (nothing to learn); $\alpha = 10^{-3}$, $\lambda = 0.1$. After 20,000 steps: Adam 0.850 vs. 0.002, AdamW 0.147 vs. 0.135.

AdamW (3/3): decay the weights outside Adam

AdamW (Loshchilov & Hutter, 2017; ICLR 2019) shrinks the weights directly, then runs Adam on the loss gradient *alone* (LMU's notation):

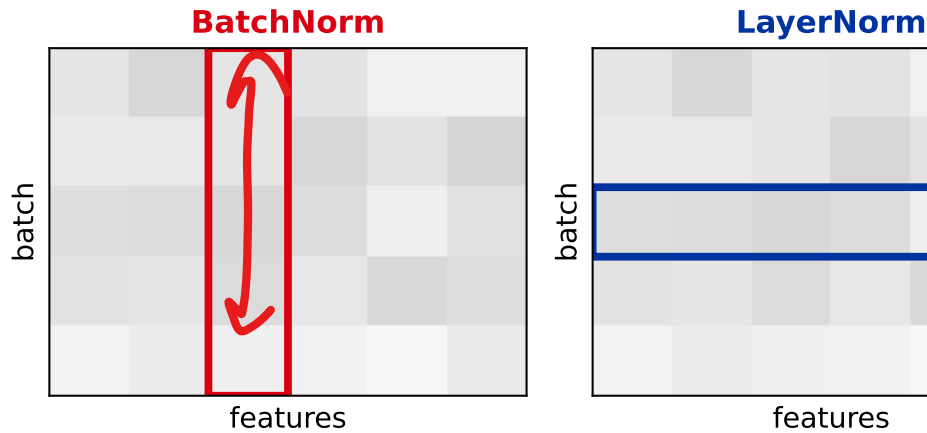
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\lambda\,\boldsymbol{\theta}$	decay: the same rate for every weight
$\mathbf{s},\ \mathbf{r} \leftarrow$ Adam's moment updates on ∇L	no $\lambda\boldsymbol{\theta}$ in the gradient
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\,\hat{\mathbf{s}} \,/\, (\sqrt{\hat{\mathbf{r}}} + \beta)$	the usual Adam step

- **In code:** `AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)` – PyTorch's default; GPT-3 and Llama 2 used 0.1.
- **Footgun:** `torch.optim.Adam(..., weight_decay=0.01)` is the L2-penalty version from the previous frame, unless you also pass `decoupled_weight_decay=True`.
- The decay is multiplied by the learning rate, so a learning-rate schedule schedules the decay too.

AdamW trains essentially every transformer today (BERT, GPT, Llama). The current challenger is **Muon** (Jordan et al., 2024), used to train Kimi K2 and GLM-4.5.

BatchNorm's sibling: LayerNorm

Both standardise activations. They differ in **what they average over**:



BatchNorm (Ioffe & Szegedy, 2015): one feature, averaged over the batch. Needs a big enough batch, and running statistics at test time.

LayerNorm (Ba, Kiros & Hinton, 2016): one example, averaged over its own features. Batch-independent – identical at train and test, fine at batch size 1.

That independence is why **every transformer uses LayerNorm**, not BatchNorm: sequences vary in length and batches are small. Today's open models use the cheaper **RMSNorm** (root mean square; Zhang & Sennrich, 2019), which drops the mean subtraction – Llama, Mistral. Details in the LLM chapter.

Two more choices: initialization & activations

We now have the hard part – **how** to descend (momentum, Adam, schedules, BatchNorm). Two more choices decide whether descent even *starts* well:

- **Initialization** – where the weights begin sets whether signal and gradients survive depth.
- **Activations** – ReLU vs the saturating sigmoids, and how to choose.

Both are in **part 2** – after which the neural-networks chapter is complete.