LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Deeplearning

Chapter 3: Optimization

Bernd Bischl

Department of Statistics – LMU Munich

Winter term 2018

HOW LEARNING DIFFERS FROM PURE OPTIMIZATION

- In machine learning we usually act **indirectly**.
- Technically, we would like to minimize the expected generalization error (or risk):

$$\mathcal{R}(\theta) = \mathbb{E}_{(x,y)\sim p_{data}}[L(y, f(x|\theta))]$$

with p_{data} being the true underlying distribution.

  - If we knew p_{data}, the minimization of the risk would be an optimization task!
  - However, when we only have a set of training samples, we deal with a machine learning problem.

HOW LEARNING DIFFERS FROM PURE OPTIMIZATION

- An alternative without directly assuming something about p_{data} is to approximate $\mathcal{R}(\theta)$ based on the training data, by means of the empirical risk:

$$\mathcal{R}_{\text{emp}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)}, \theta))$$

- So rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases as well.
- The empirical risk minimization is prone to overfitting as models with high capacity can simply memorize the training set.
- Thus, we have to tweak our optimization such that the quantity that we actually optimize is even more different from the quantity that we truly want to optimize (in reality we obviously optimize $\mathcal{R}_{reg}(\theta|X, y)$, but to keep things easy we spare that).

REVISION: GRADIENT DESCENT

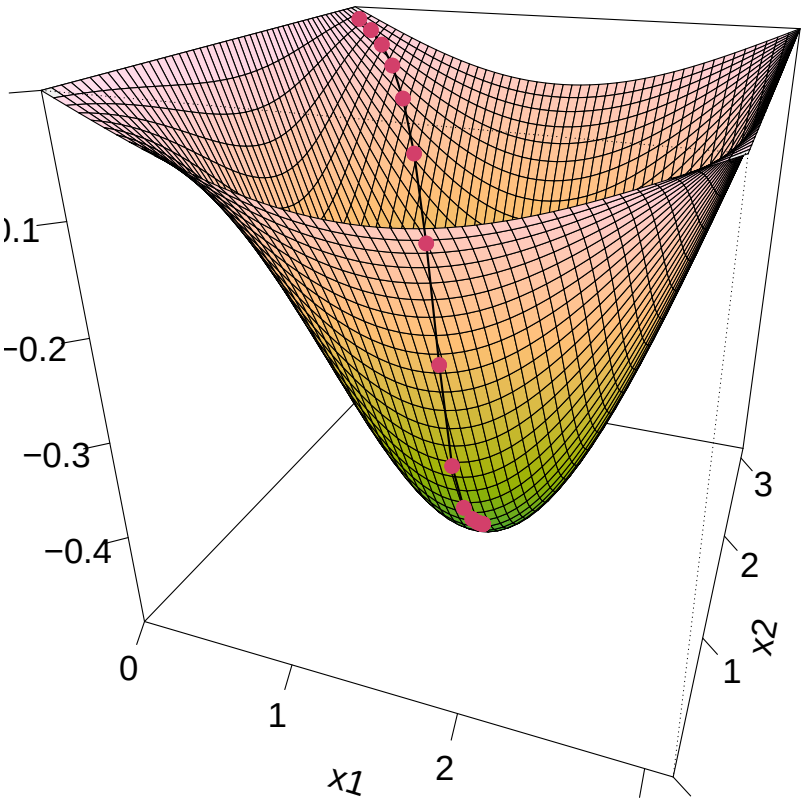
- Let $f(x)$ be an arbitrary, differentiable, unrestricted target function, which we want to minimize.
  - We can calculate the gradient $g = \nabla f(x)$, which always points in the direction of the **steepest ascent**.
  - Thus $-g = -\nabla f(x)$ points in the direction of the **steepest descent**!
- Standing at a point x_k during minimization, we can improve this point by doing the following step:

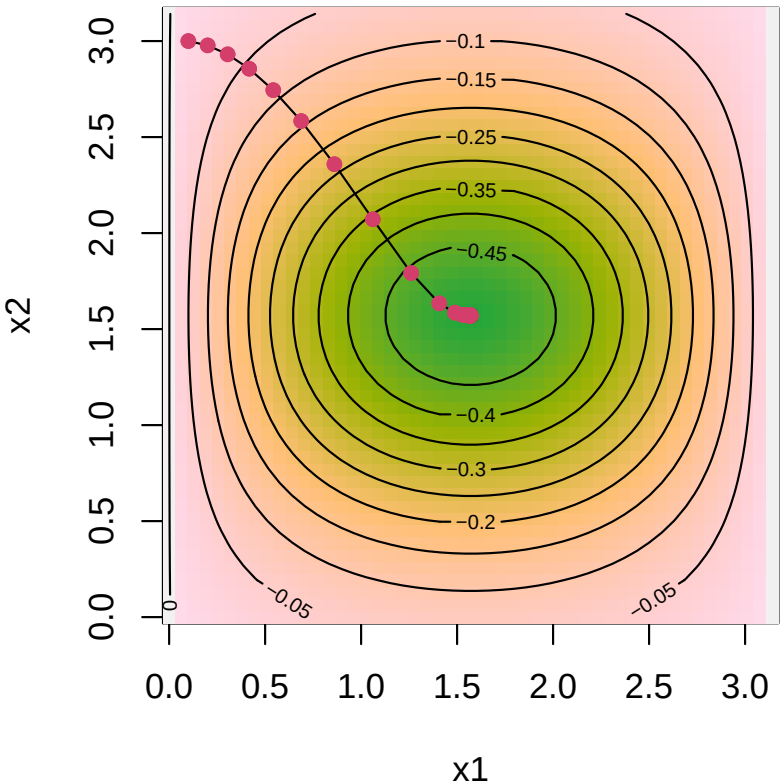
$$f(x_{k+1}) = f(x_k) - \alpha \nabla f(x_k)$$

  "Walking down the hill, towards the valley."
- α determines the length of the step and is called **step size** or in terms of neural networks **learning rate**.

EXAMPLE: GRADIENT DESCENT AND CONVEX FUNCTION

$$f(x_1, x_2) = -\sin(x_1) \cdot \frac{1}{2\pi} \exp\left((x_2 - \pi/2)^2\right)$$





STOCHASTIC GRADIENT DESCENT

- Optimization algorithms that use the entire training set are called **batch** or **deterministic**.
  - All training samples are processed in one huge step.
  - Computationally very costly, particularly in deeplearning!
- Optimization algorithms that use only a single example at a time are called **stochastic** or **online**.
- Most optimization algorithms applied in deeplearning are somewhere in between!
  - Usually more than one, but less than all training samples are used at each iteration. Traditionally called **minibatch**, nowadays **stochastic** methods.
- One **epoch** means one pass of the full training set.
- Since we divide the training set into minibatches, each epoch goes through the whole training set. Each iteration goes through one minibatch.

STOCHASTIC GRADIENT DESCENT

- SGD and its modifications are the most used optimization algorithms for machine learning in general and for deep learning in particular.

Algorithm 1 Basic SGD parameter update at training iteration k

require learning rate α
require initial parameter θ
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: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(y^{(i)}, f(\tilde{x}^{(i)}, \theta))$
    Apply update: $\theta \leftarrow \theta - \alpha \hat{g}$
end while

- Thus, what SGD basically does is computing an unbiased estimate of the gradient by taking the average gradients of a minibatch to update the parameter θ.

STOCHASTIC GRADIENT DESCENT

- For the mnist example of chapter one, we used SGD
  - with a minibatch of size 100
  - and trained for 10 epochs.

```
model = mx.model.FeedForward.create(
  symbol = softmax,
  X = train.x, y = train.y,
  optimizer = "sgd",
  array.batch.size = 100L,
  num.round = 10L
)
```

- Consequently we feed our algorithm successively with 100 training samples before updating the weights.
- An epoch means that the model gets to see the whole training set one time.
- Thus, one epoch involves $\frac{\text{training samples}}{\text{batch size}}$ gradient updates.
- We repeat this procedure for 10 times.

BATCH SIZE AS A HYPERPARAMETER

```
data = mx.symbol.Variable("data")
fc1 = mx.symbol.FullyConnected(data, num_hidden = 128)
act1 = mx.symbol.Activation(fc1, act_type = "relu")
fc2 = mx.symbol.FullyConnected(act1, num_hidden = 64)
act2 = mx.symbol.Activation(fc2, act_type = "relu")
fc3 = mx.symbol.FullyConnected(act2, num_hidden = 32)
act3 = mx.symbol.Activation(fc3, ,act_type = "relu")
fc4 = mx.symbol.FullyConnected(act3, num_hidden = 10)
softmax = mx.symbol.SoftmaxOutput(fc4)
```

Let us train this architecture with six different values for the

$$\text{batch size} \in (16, 17, 20, 100, 128, 1000)$$

on mnist.

BATCH SIZE AS A HYPERPARAMETER

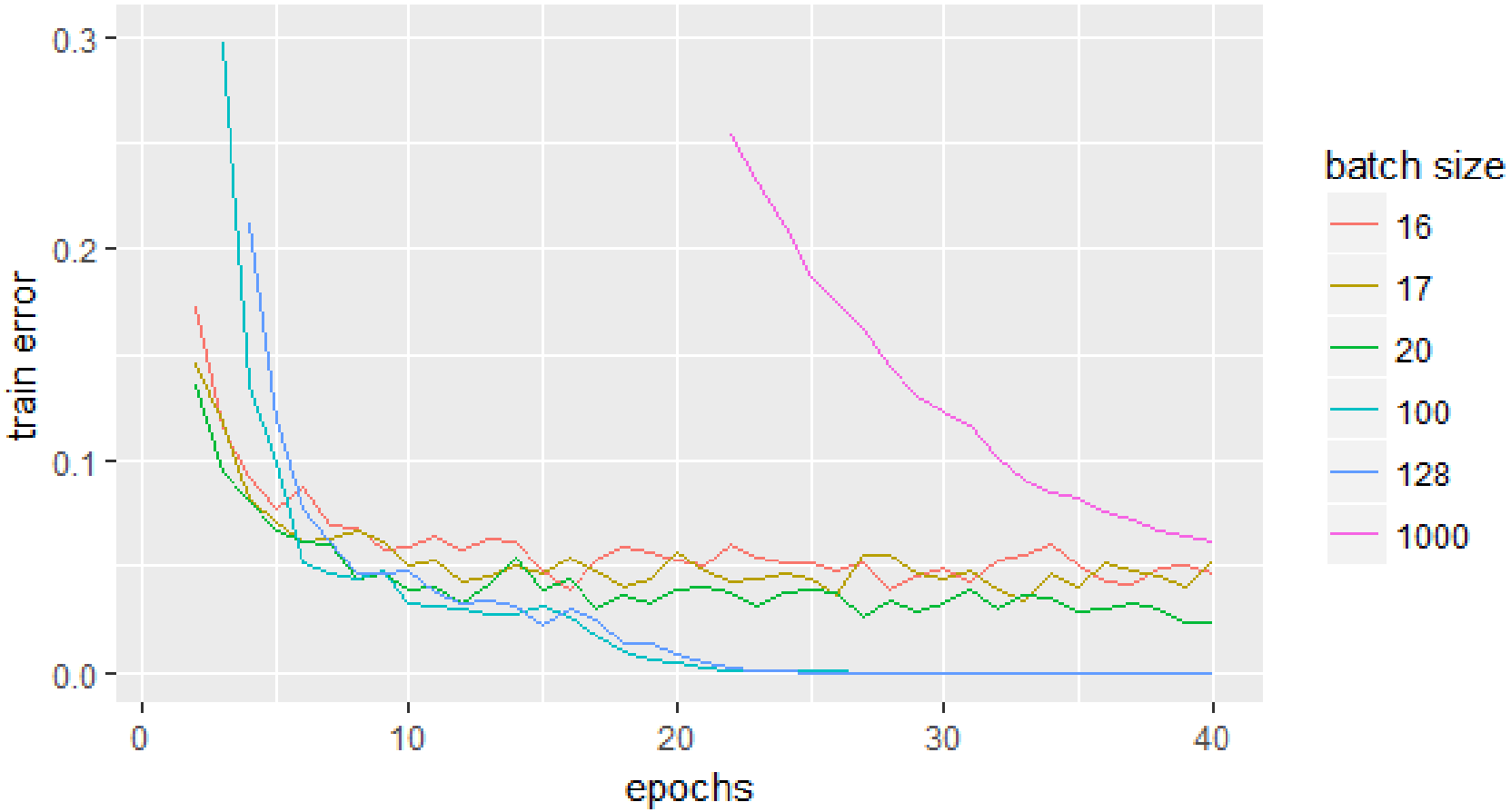


Table: Different batch sizes and training time in seconds

Batch size	16	17	20	100	128	1000
Training time	44.42	44.84	38.84	8.91	7.67	2.59

BATCH SIZE AS A HYPERPARAMETER

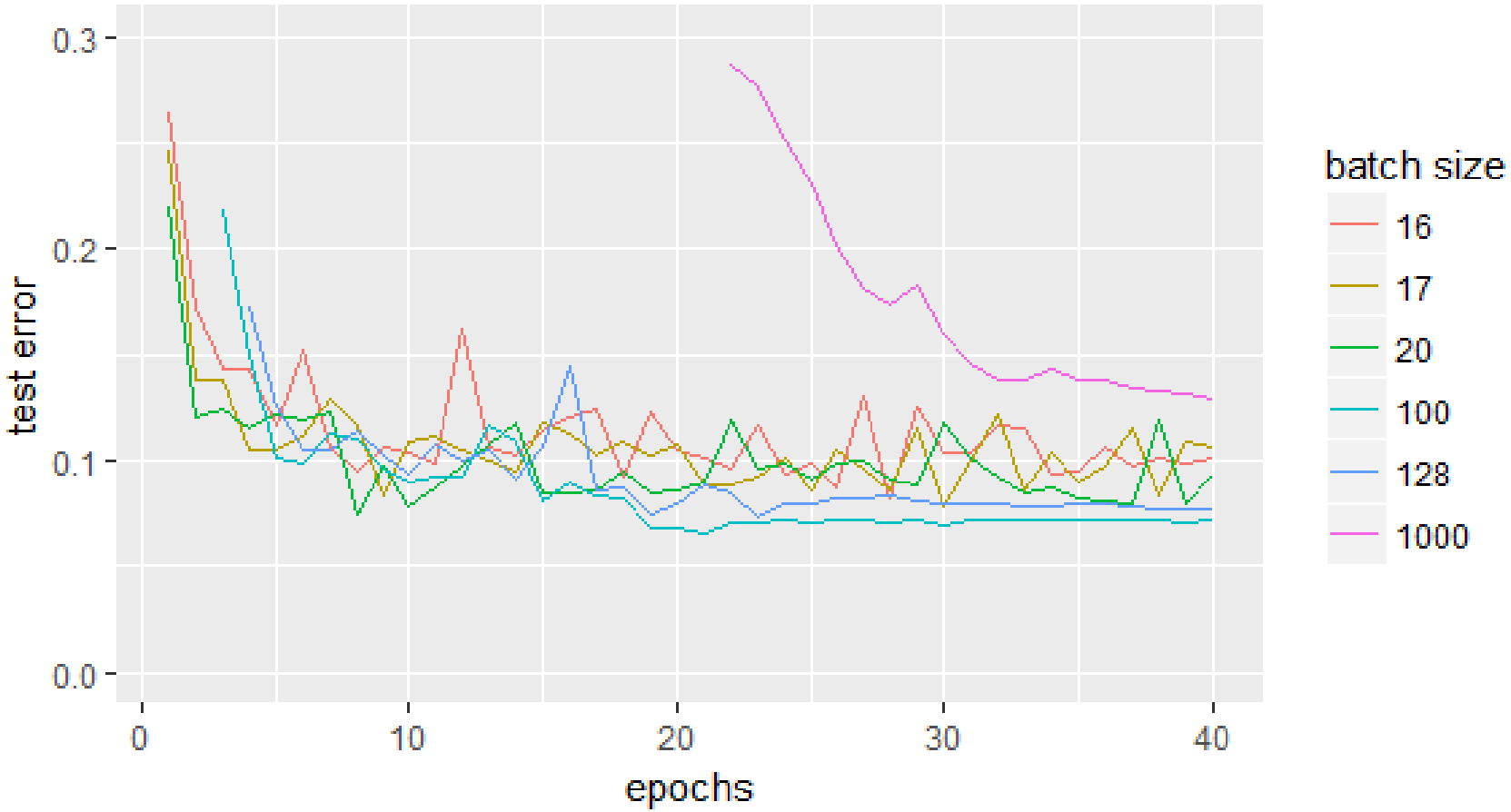


Table: Different batch sizes and training time in seconds

Batch size	16	17	20	100	128	1000
Training time	44.42	44.84	38.84	8.91	7.67	2.59

FLAT REGIONS

- In optimization we look for areas with zero gradient.
- A variant of zero gradient are flat regions such as saddle points.
- For the error surface f of a neural network, the expected ratio of the number of saddle points to local minima typically grows exponentially with n

  $$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

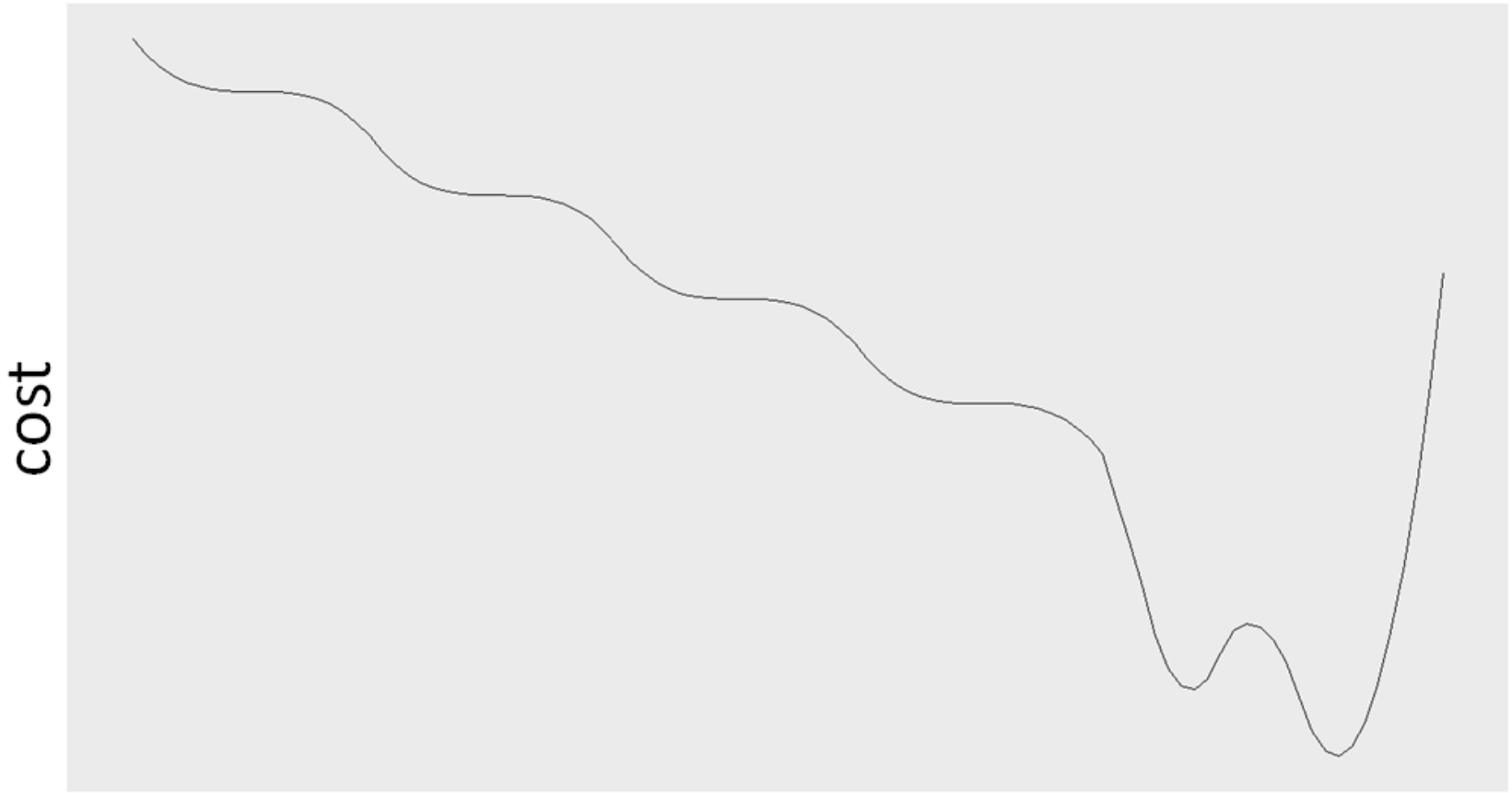
  In other words: deeper networks exhibit alot more saddle points than local minima.
- The Hessian at a local minimum has only positive eigenvalues. At a saddle point it is a mixture of positive and negative eigenvalues.
- Why is that?

FLAT REGIONS

- 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 n-dimensional space, it is exponentially unlikely that all n 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 n 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.

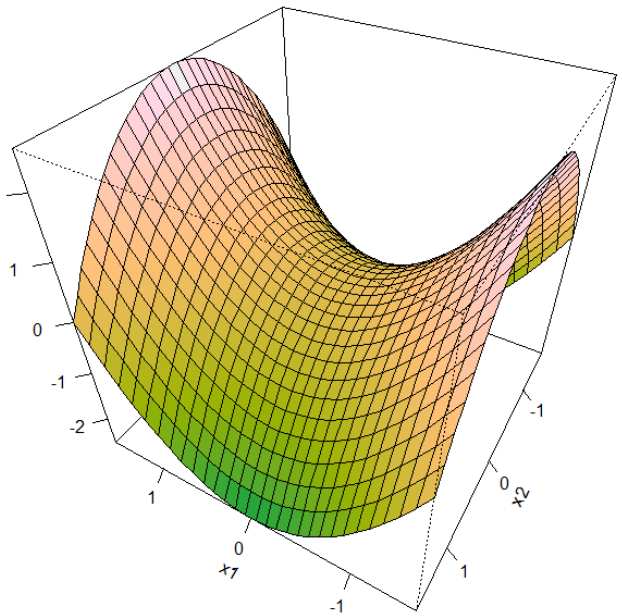
FLAT REGIONS

- “Saddle points are surrounded by high error plateaus that can dramatically slow down learning, and give the illusory impression of the existence of a local minimum” (Dauphin et al. (2014)).



FLAT REGIONS

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

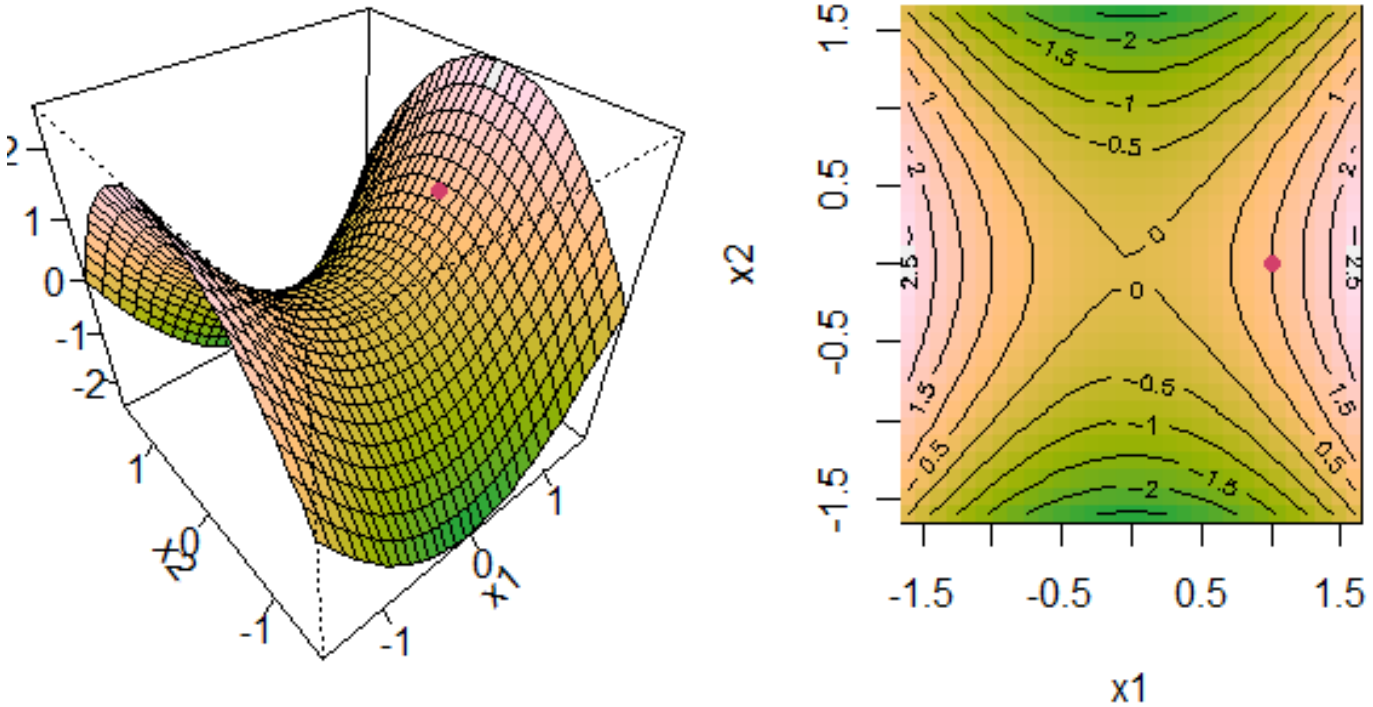


- Along x_1, the function curves upwards (eigenvector of the Hessian with positive eigenvalue). Along x_2, the function curves downwards (eigenvector of the Hessian with negative eigenvalue).

FLAT REGIONS

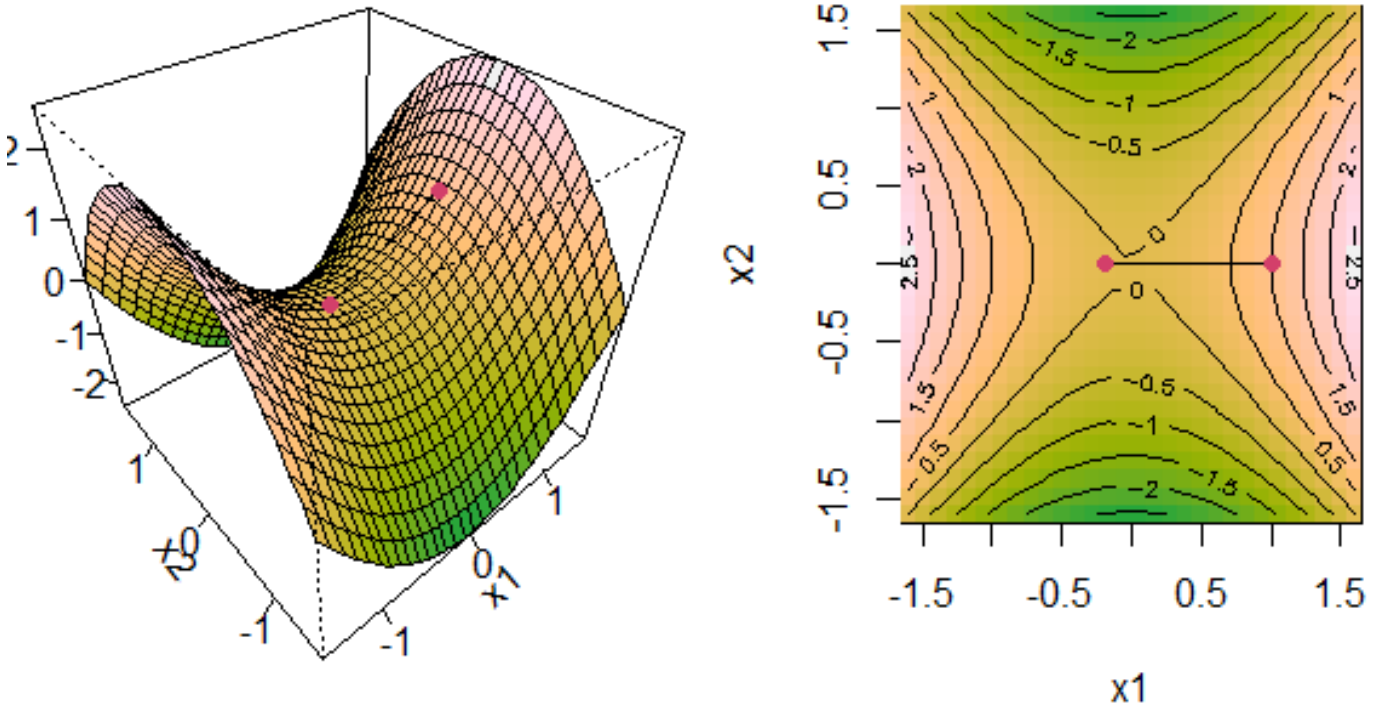
- So how do saddle points impair optimization?
- First-order algorithms that use only gradient information **might** get stuck in saddle points.
- Second-order algorithms experience even greater problems when dealing with saddle points. Newtons method for example actively searches for a region with zero 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 GRADIENT DESCENT



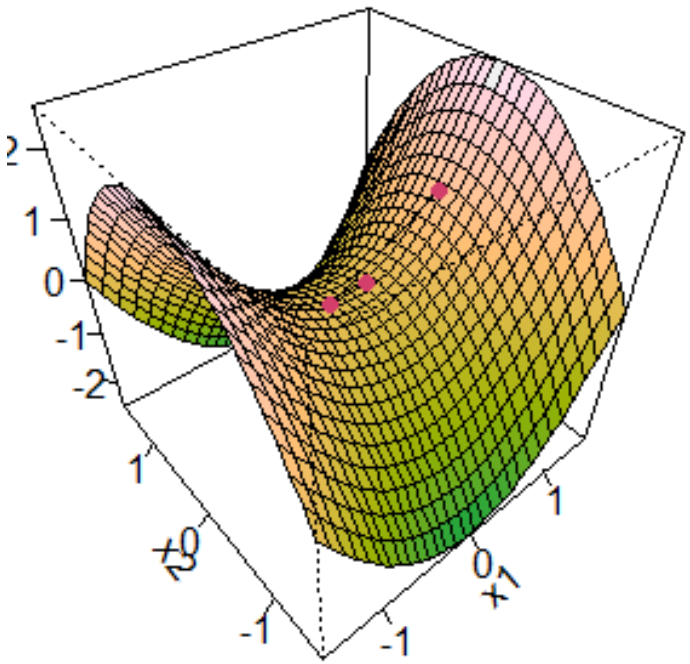
Red dot: starting location

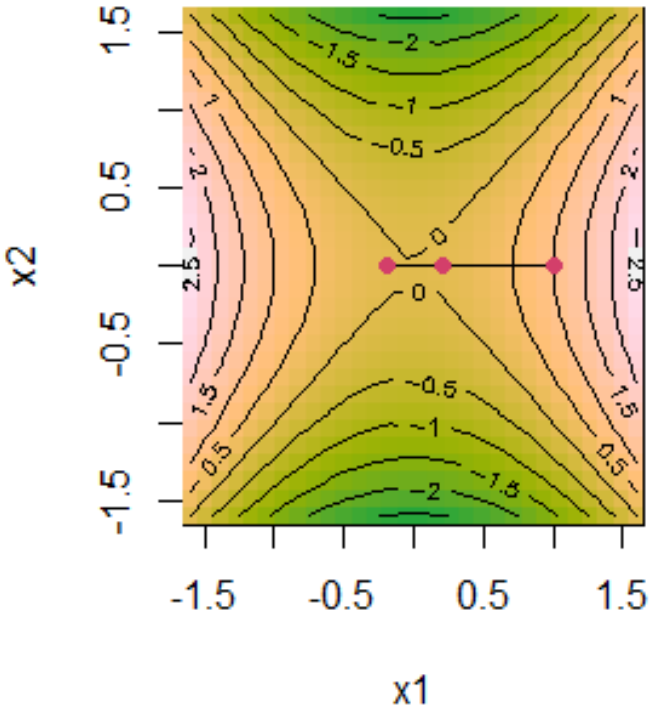
EXAMPLE: SADDLE POINT WITH GRADIENT DESCENT



First step..

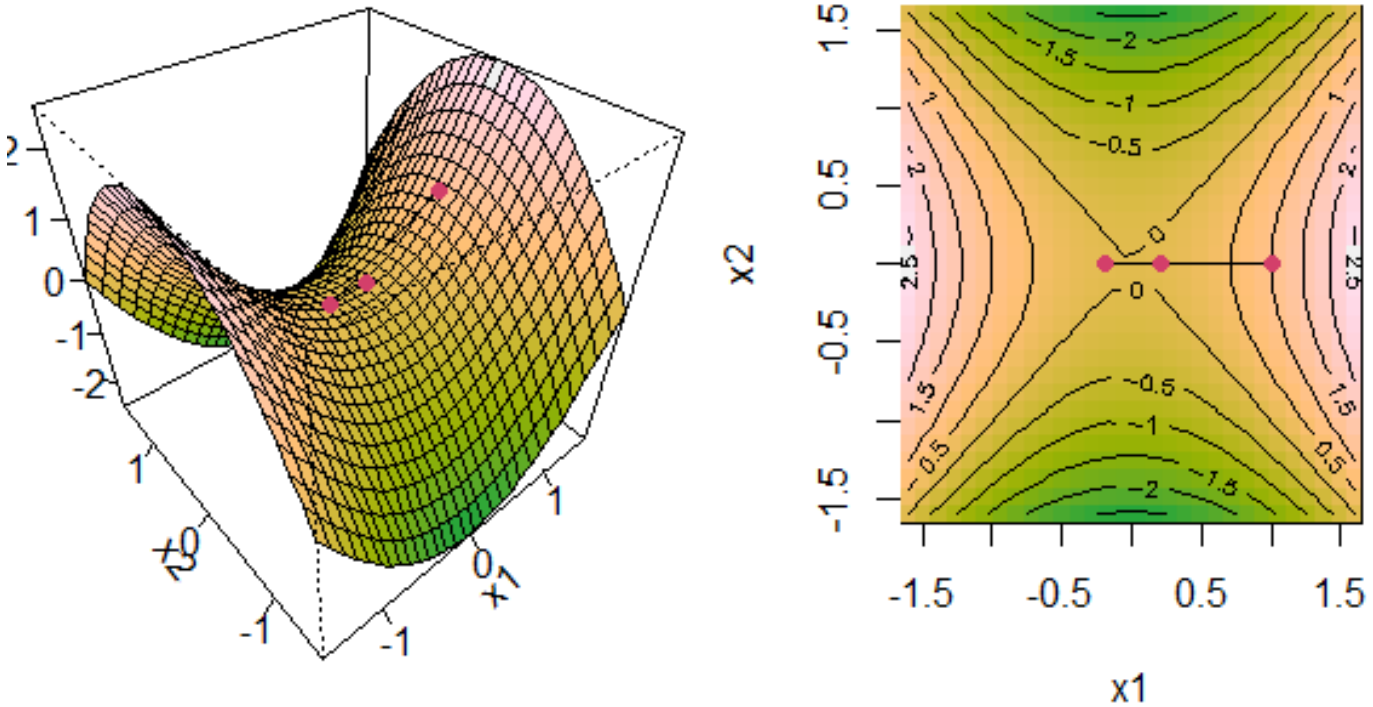
EXAMPLE: SADDLE POINT WITH GRADIENT DESCENT





..second step..

EXAMPLE: SADDLE POINT WITH GRADIENT DESCENT



..tenth step got stuck and can't escape the saddle point!

CLIFFS AND EXPLODING GRADIENTS

- As a result from the multiplication of several parameters, the objective function for highly nonlinear deep neural networks (i.e. recurrent nets, chapter 5) often contain sharp nonlinearities.
  - That may result in very high derivatives in some places.
  - As the parameters get close to such a 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

- 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*:

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

EXAMPLE: CLIFFS AND EXPLODING GRADIENTS

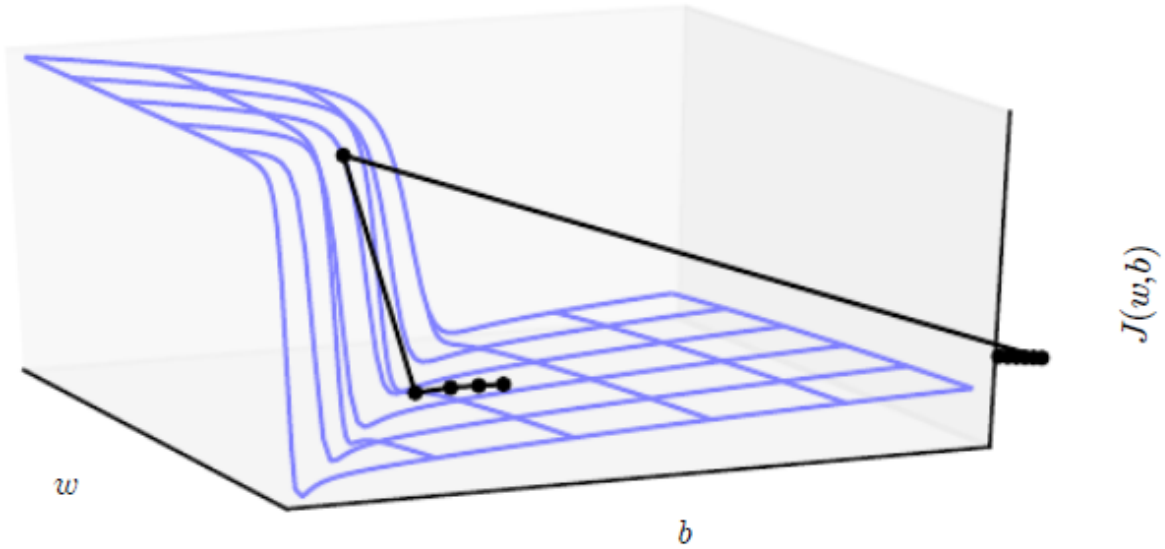
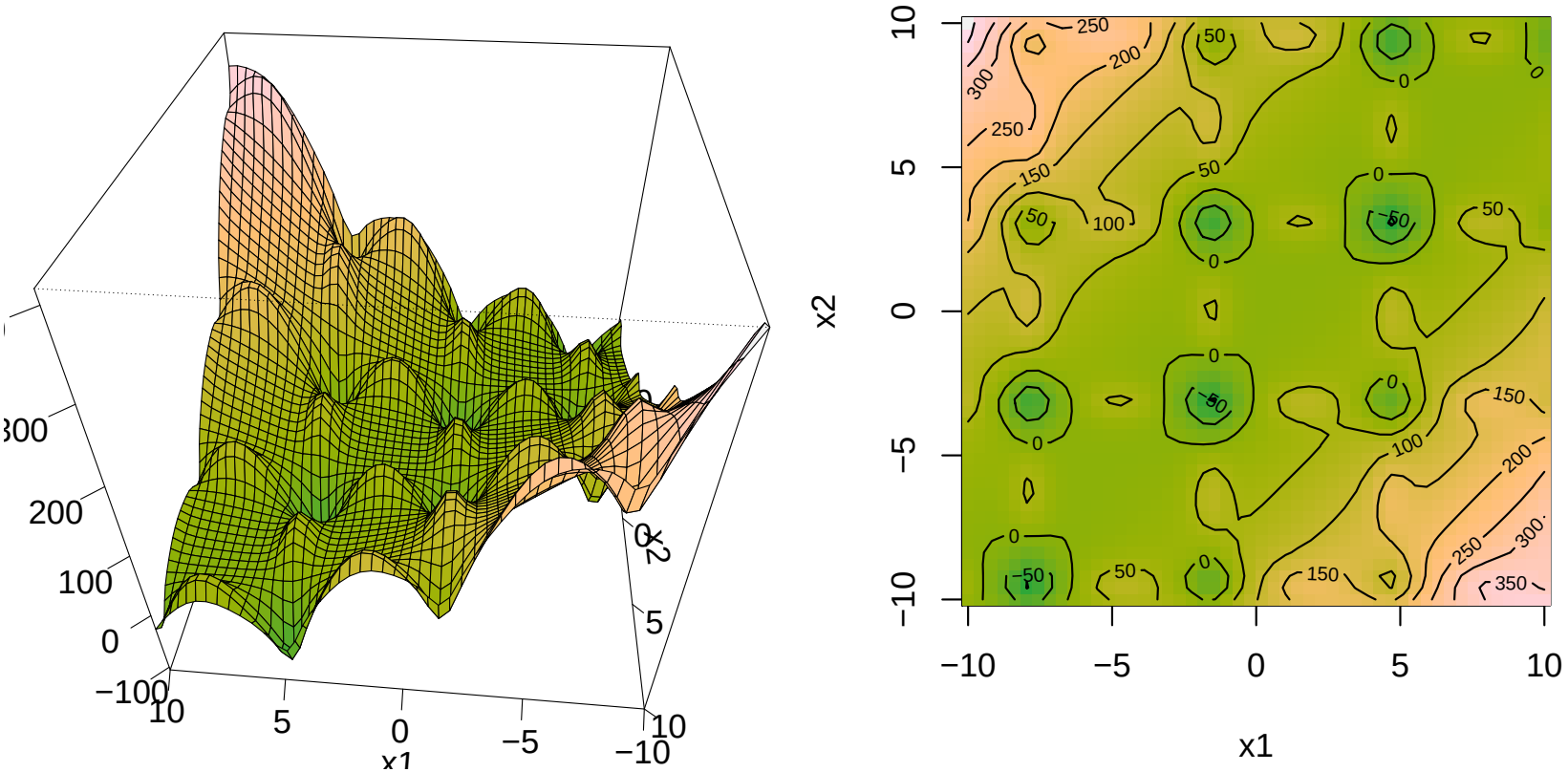


Figure: “The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters 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)).

EXAMPLE: MULTIMODAL FUNCTION

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



MOMENTUM

- 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:

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

- We introduce a new hyperparameter $\varphi \in [0, 1)$, determining how quickly the contribution of previous gradients decay.
- ν 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 are.
  - The step size grows when many succesive gradients point in the same direction.
- Common values for φ are 0.5, 0.9 and even 0.99.
- We can think of momentum as the fraction $\frac{1}{1-\varphi}$
  - Then, $\varphi = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.
- 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/`

Example:

$$
\begin{aligned}
\nu_1 &\leftarrow \varphi\nu_0 - \alpha g(\theta_0) \\
\theta_1 &\leftarrow \theta_0 + \varphi\nu_0 - \alpha g(\theta_0)
\end{aligned}
$$

$$
\begin{aligned}
\nu_2 &\leftarrow \varphi\nu_1 - \alpha g(\theta_1) \\
&= \varphi(\varphi\nu_0 - \alpha g(\theta_0)) - \alpha g(\theta_1) \\
\theta_2 &\leftarrow \theta_1 + \varphi(\varphi\nu_0 - \alpha g(\theta_0)) - \alpha g(\theta_1)
\end{aligned}
$$

$$
\begin{aligned}
\nu_3 &\leftarrow \varphi\nu_2 - \alpha g(\theta_2) \\
&= \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta_0)) - \alpha g(\theta_1)) - \alpha g(\theta_2) \\
\theta_3 &\leftarrow \theta_2 + \varphi(\varphi(\varphi\nu_0 - \alpha g(\theta_0)) - \alpha g(\theta_1)) - \alpha g(\theta_2) \\
&= \theta_2 + \varphi^3\nu_0 - \varphi^2\alpha g(\theta_0) - \varphi\alpha g(\theta_1) - \alpha g(\theta_2) \\
&= \theta_2 - \alpha(\varphi^2 g(\theta_0) + \varphi^1 g(\theta_1) + \varphi^0 g(\theta_2)) + \varphi^3\nu_0
\end{aligned}
$$

$$
\theta_{k+1} = \theta_k - \alpha \sum_{j=0}^{k} \varphi^j g(\theta_j) + \varphi^{k+1}\nu_0
$$

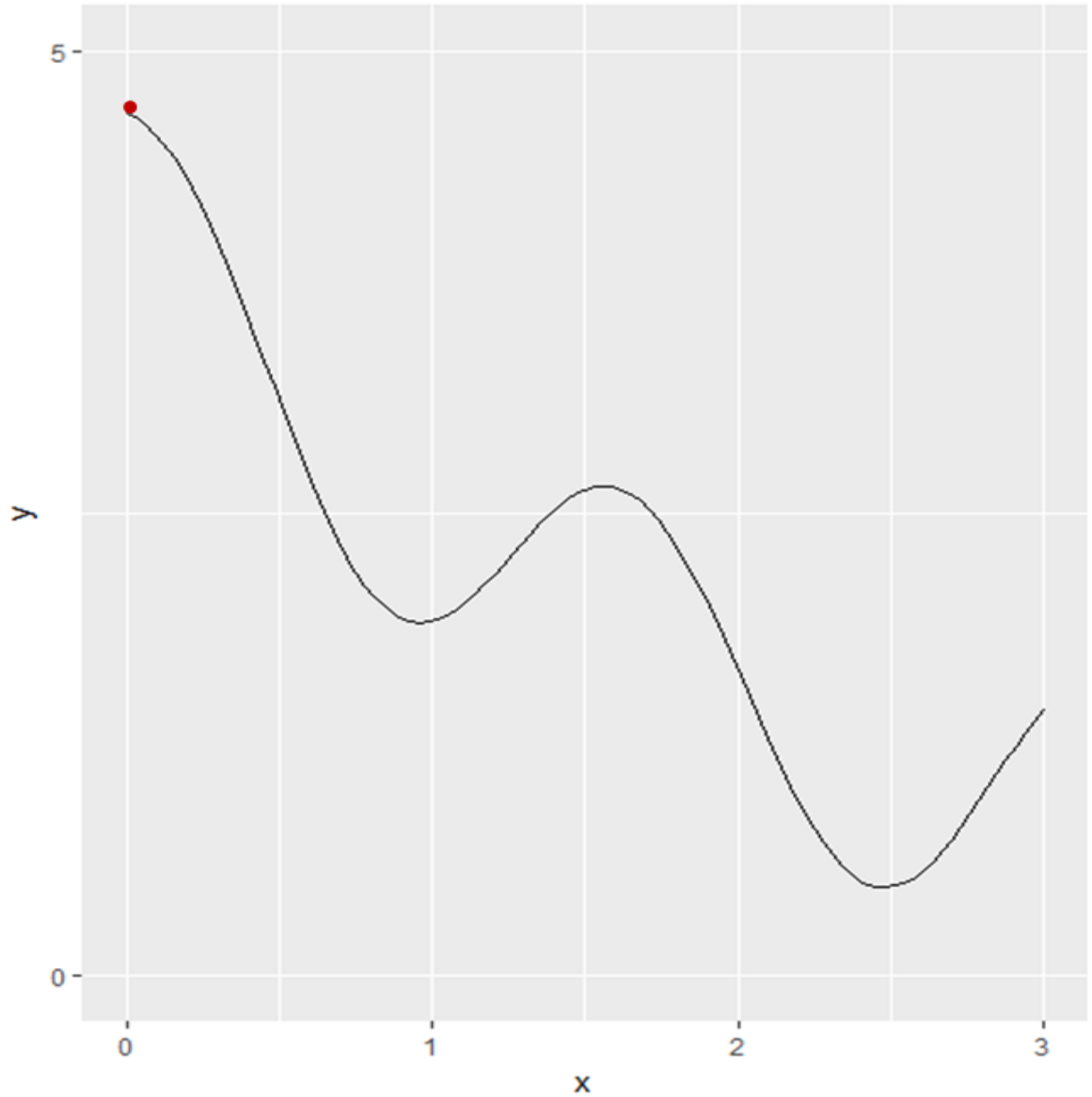
Suppose momentum always observes the same gradient $g(\theta)$:

$$\begin{aligned}
\theta_{k+1} &= \theta_k - \alpha \sum_{j=0}^{k} \varphi^j g(\theta_j) + \varphi^{k+1} \nu_0 \\
&= \theta_k - \alpha g(\theta) \sum_{j=0}^{k} \varphi^j + \varphi^{k+1} \nu_0 \\
&= \theta_k - \alpha g(\theta) \frac{1 - \varphi^{k+1}}{1 - \varphi} + \varphi^k \nu_0 \\
\lim_{k \to \infty} &= \theta_k - \alpha g(\theta) \frac{1}{1 - \varphi}
\end{aligned}$$

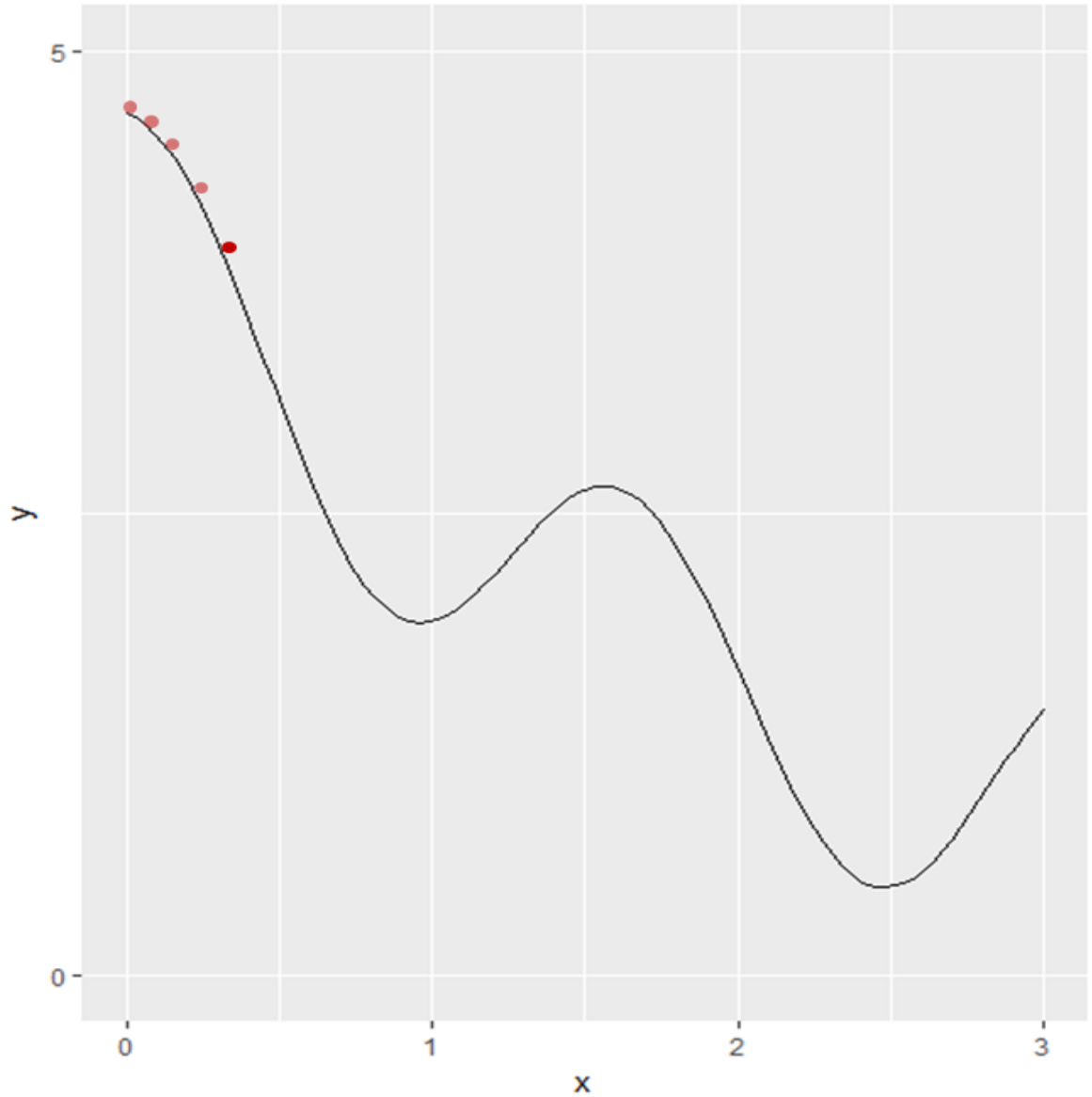
Thus, momentum will accelerate in the direction of $-g(\theta)$ until reaching terminal velocity with step size:

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

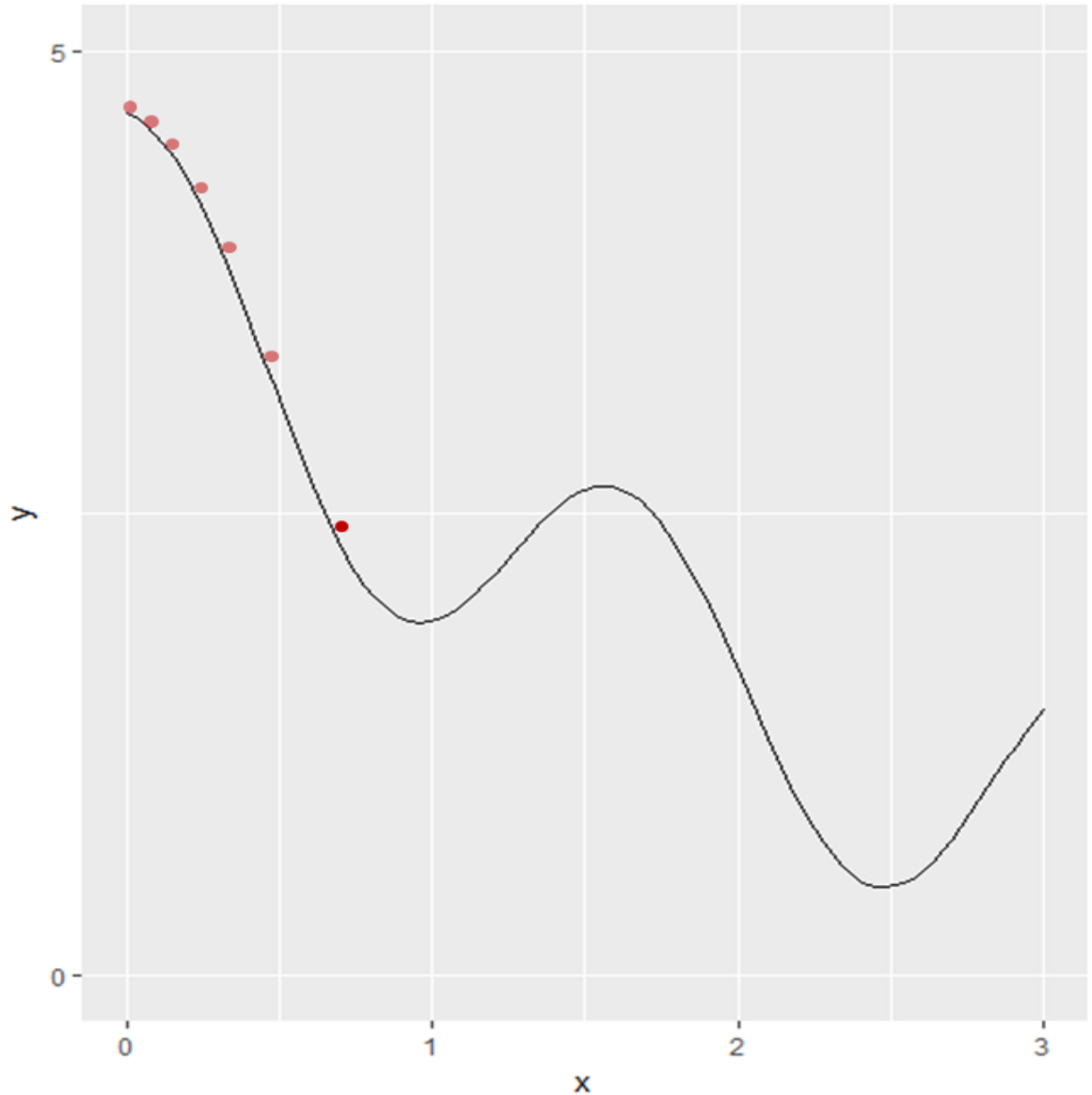
MOMENTUM: PHYSICAL EXPLANATION



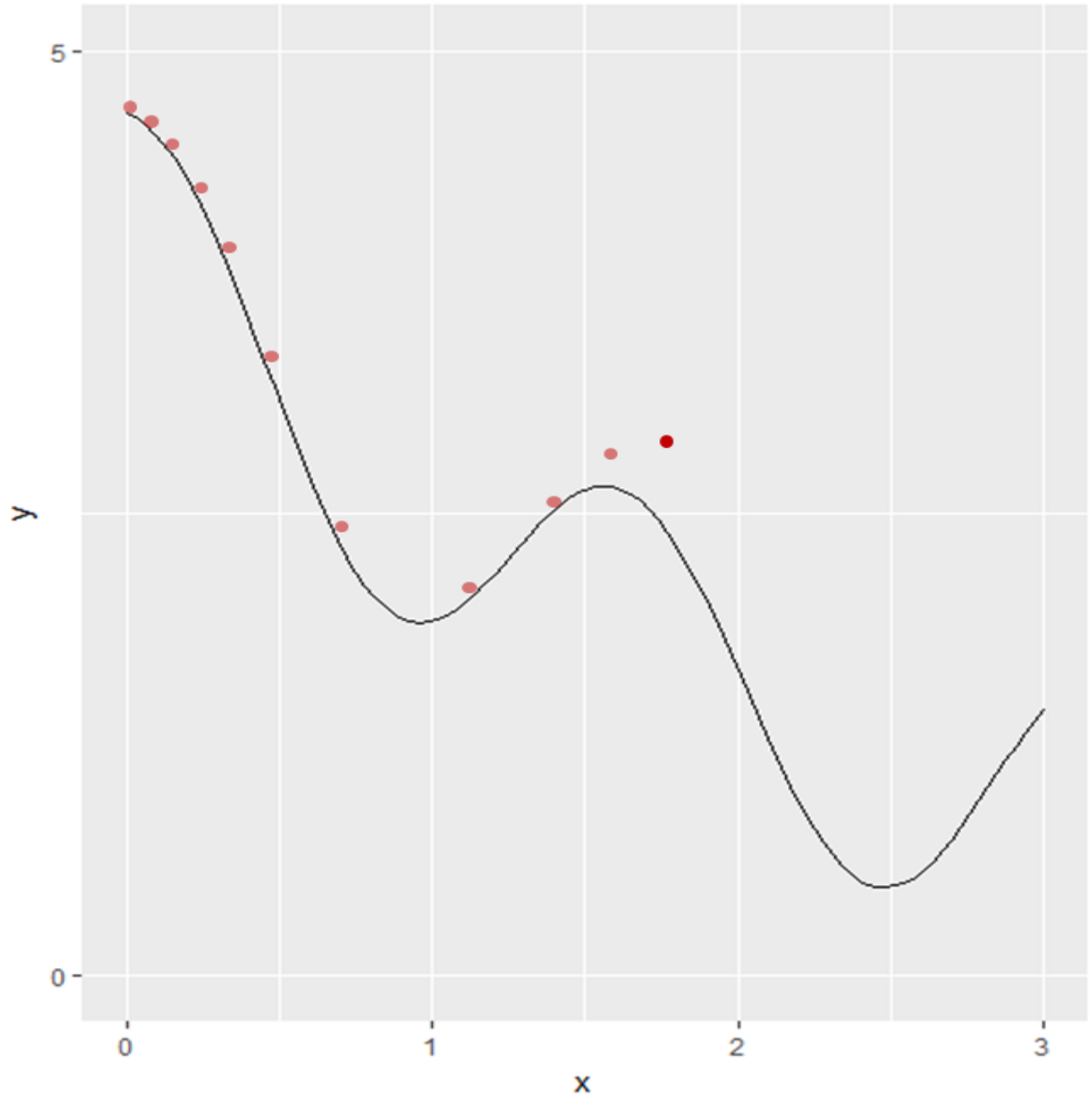
MOMENTUM: PHYSICAL EXPLANATION



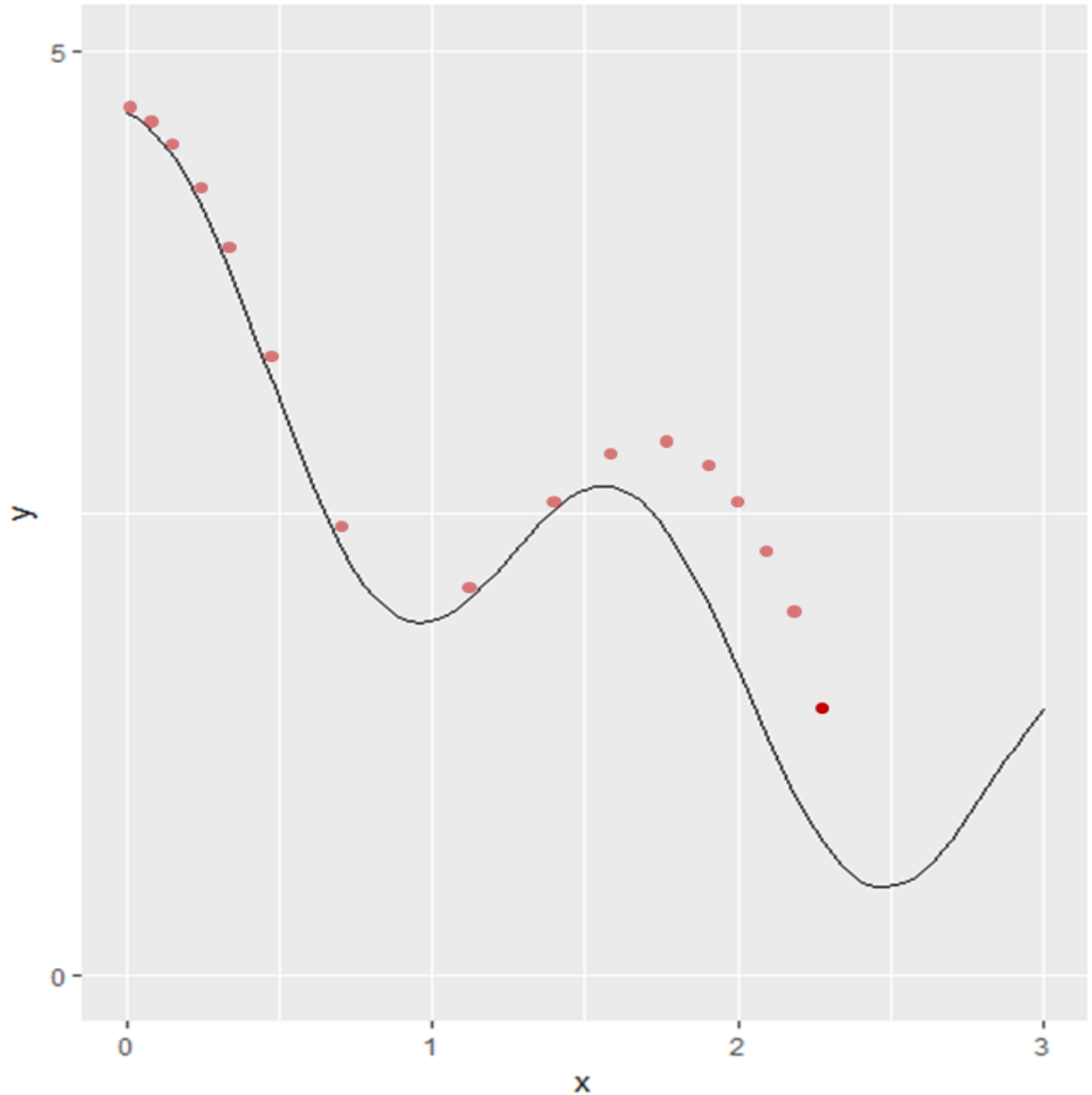
MOMENTUM: PHYSICAL EXPLANATION



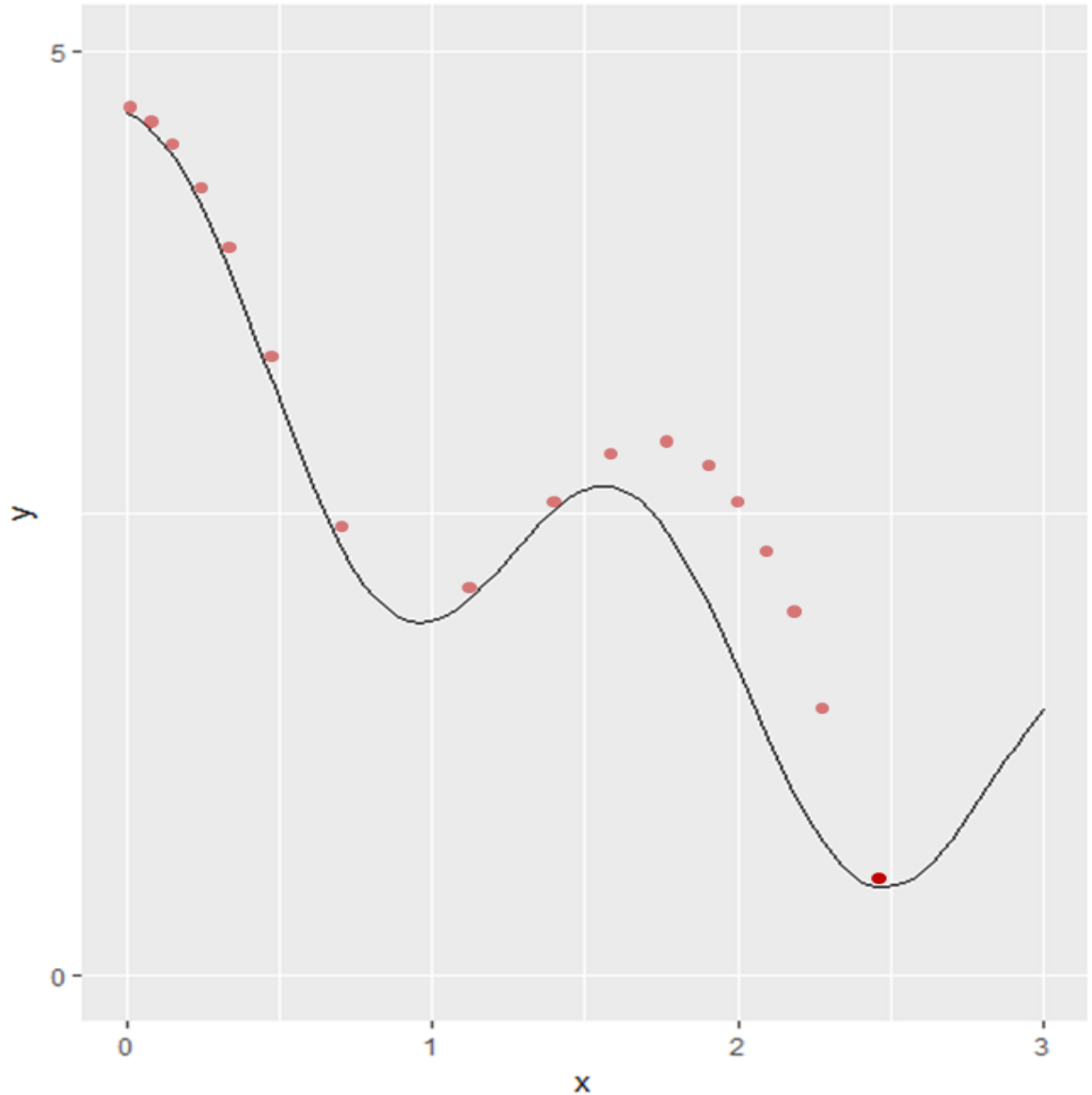
MOMENTUM: PHYSICAL EXPLANATION



MOMENTUM: PHYSICAL EXPLANATION



MOMENTUM: PHYSICAL EXPLANATION



GRADIENT DESCENT VS MOMENTUM

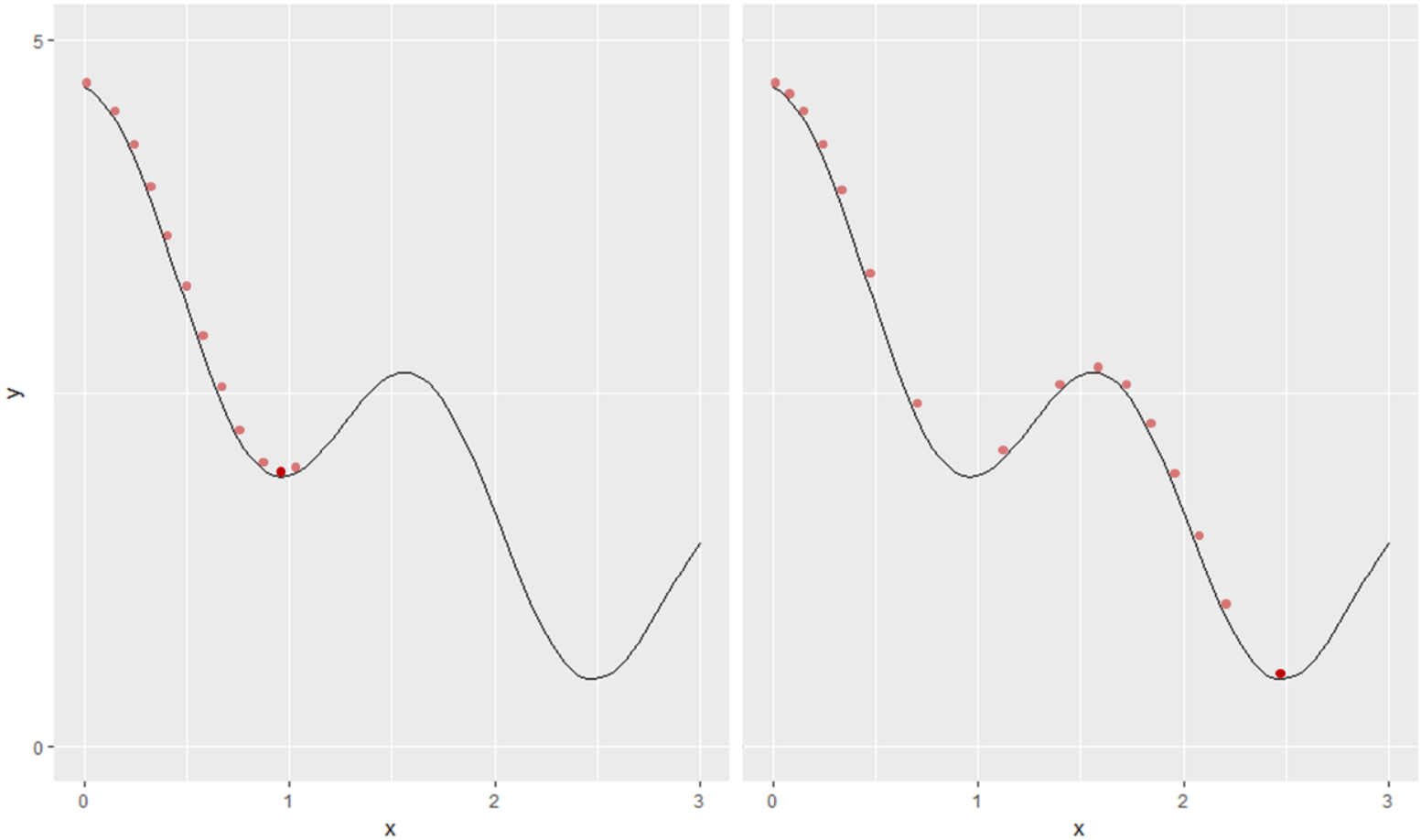


Figure: Gradient descent (left) needs many steps **and** gets stuck in a local minimum Momentum (right) finds the global minimum

SGD WITH MOMENTUM

Algorithm 2 Stochastic gradient descent with momentum

1: **require** learning rate α and momentum φ
2: **require** initial parameter θ and initial velocity ν
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: $\hat{g} \leftarrow +\frac{1}{m} \nabla_\theta \sum_i L(y^{(i)}, f(\tilde{x}^{(i)}, \theta))$
6: Compute velocity update: $\nu \leftarrow \varphi\nu - \alpha\hat{g}$
7: Apply update: $\theta \leftarrow \theta + \nu$
8: **end while**

SGD WITH MOMENTUM

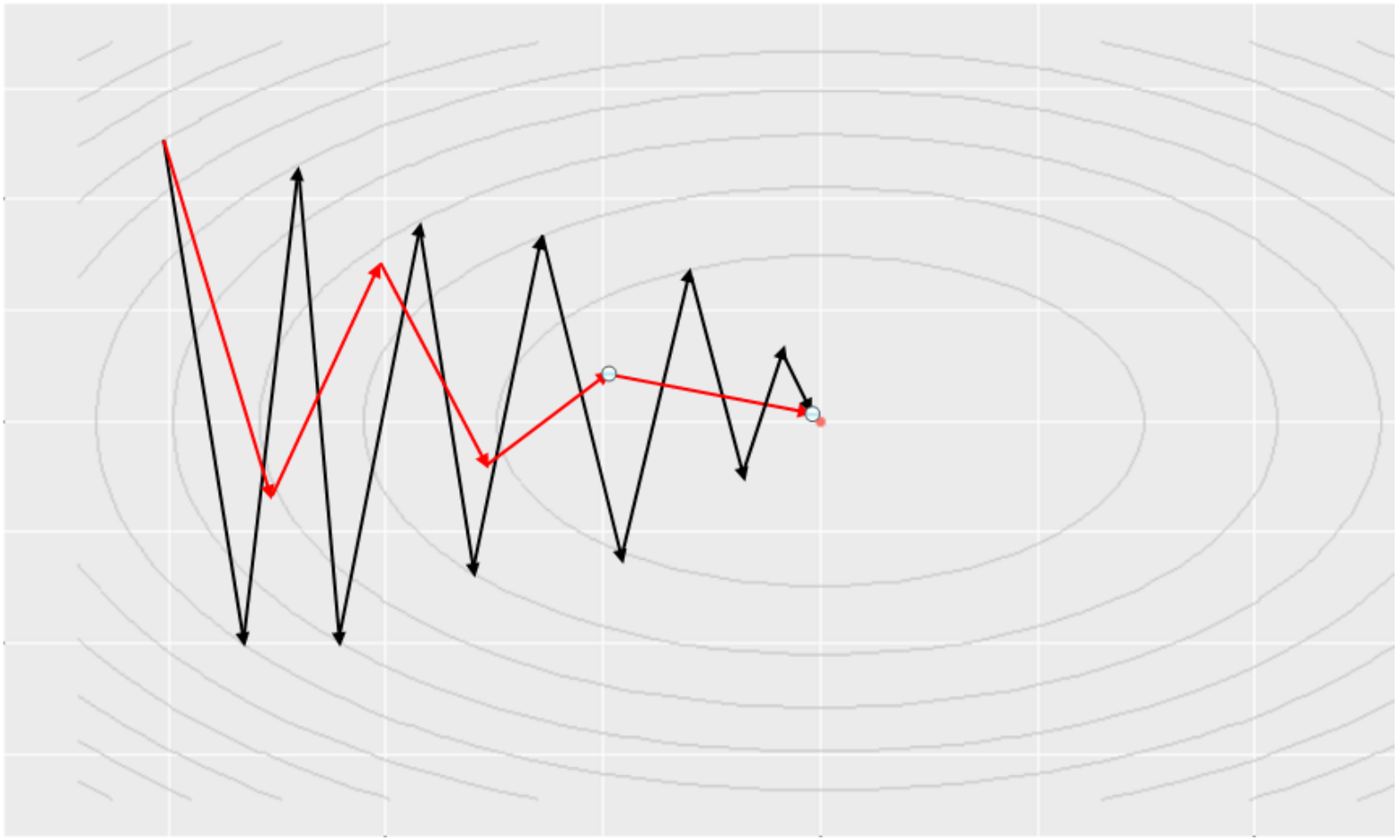


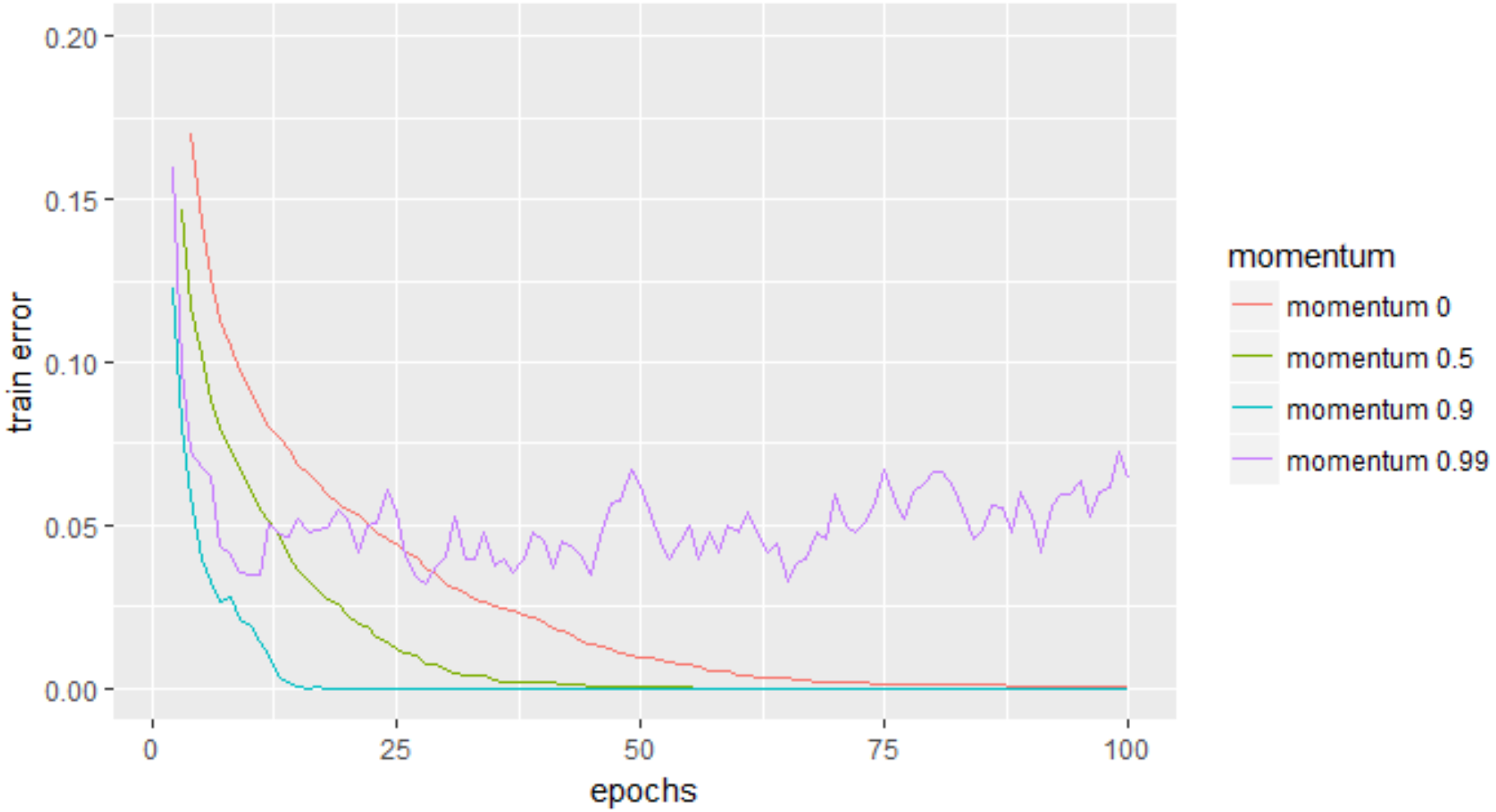
Figure: The contour lines show a quadratic loss function with a poorly conditioned Hessian matrix. The two curves show how standard gradient descent (black) and momentum (red) learn when dealing with ravines.

MOMENTUM IN PRACTICE

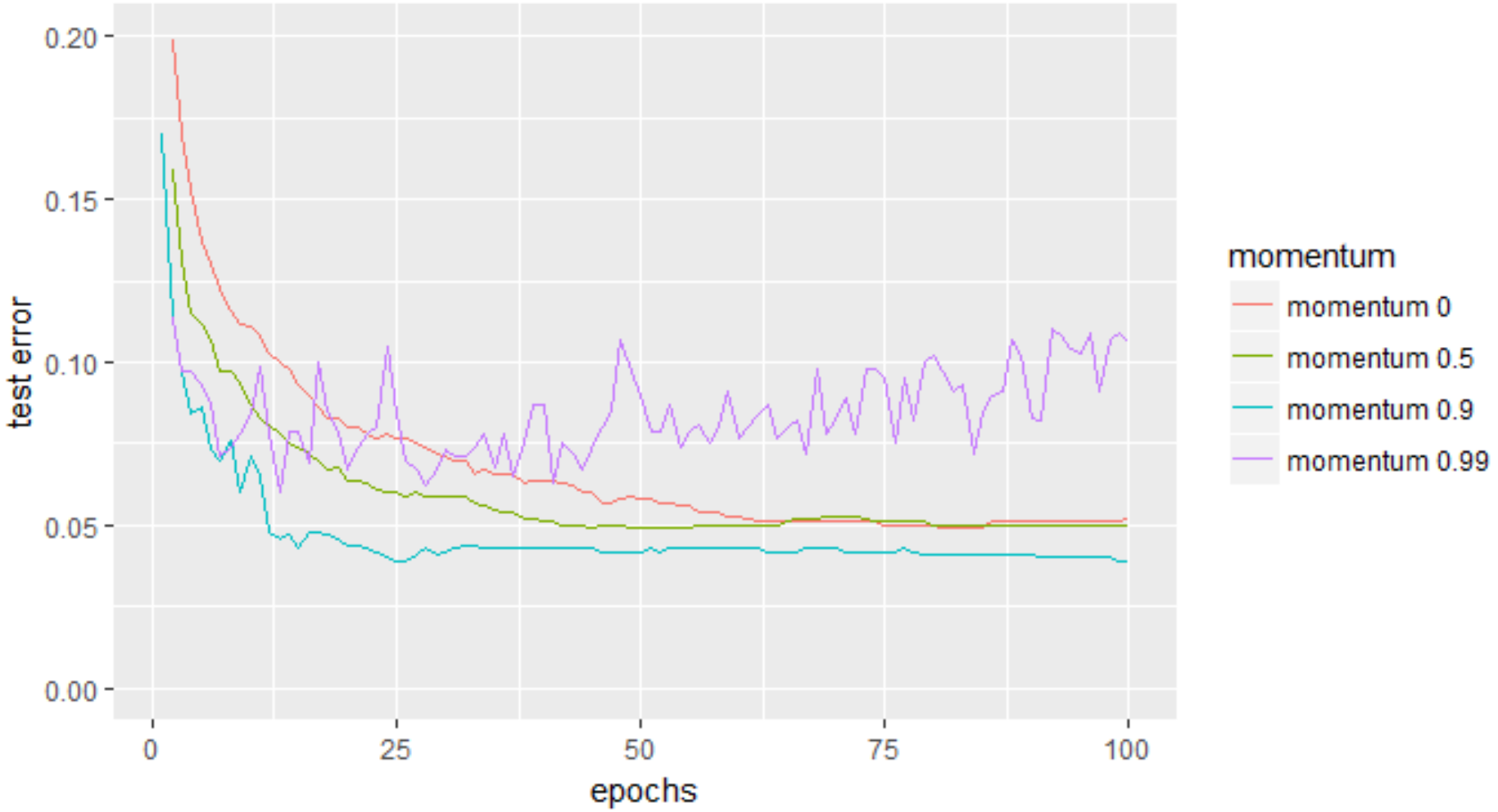
- Lets try out different values of momentum (with SGD) on the MNIST data.
- We apply the same architecture we've used a dozen of times already (note that we used $\varphi = 0.9$ in all computations so far, i.e. in chapter 1 and 2)!

```
fc1 = mx.symbol.FullyConnected(data, num_hidden = 512)
act1 = mx.symbol.Activation(fc1, activation = "relu")
fc2 = mx.symbol.FullyConnected(act1, num_hidden = 512)
act2 = mx.symbol.Activation(fc2, activation = "relu")
fc3 = mx.symbol.FullyConnected(act2, num_hidden = 512)
act3 = mx.symbol.Activation(fc3, activation = "relu")
fc4 = mx.symbol.FullyConnected(act3, num_hidden = 10)
softmax = mx.symbol.SoftmaxOutput(fc4, name = "sm")
```

MOMENTUM IN PRACTICE



MOMENTUM IN PRACTICE



NESTOROV 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:

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

- We can interpret Nesterov momentum as an attempt to add a correction factor to the basic method.

STOCHASTIC GRADIENT DESCENT

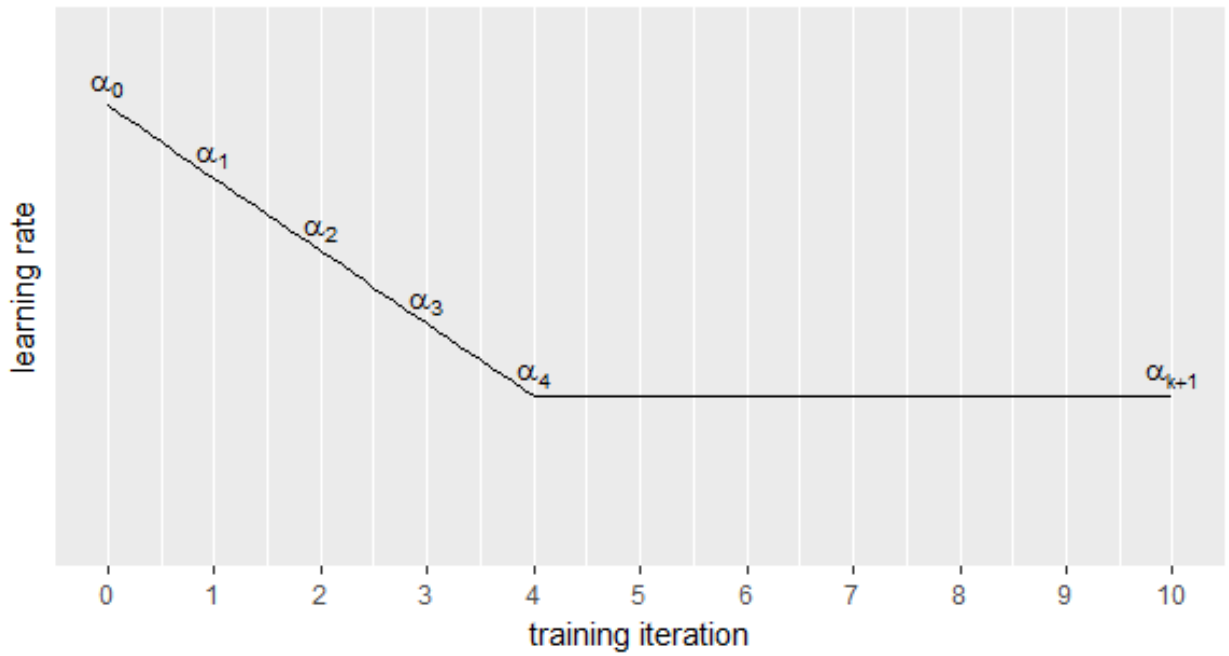
- We would like to force convergence until reaching a local minimum.
- A decisive parameter of SGD is the learning rate α.
- Applying SGD, we have to decrease the learning rate over time, thus α_k (learning rate at training iteration k).
  - The estimator $\hat{g}$ is computed of small batches.
  - Random 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:

$$\begin{aligned} \alpha_k &= (1-\epsilon)\alpha_0 + \epsilon\alpha_\tau \qquad \text{with } \epsilon = \frac{k}{\tau} \text{ and } \alpha_{k+1} \text{ const.} \\ &= k(-\frac{\alpha_0 + \alpha_\tau}{\tau}) + \alpha_0 \end{aligned}$$

STOCHASTIC GRADIENT DESCENT

Example for $\tau = 4$:

iteration	ϵ	α_k
1	0.25	$(1 - \frac{1}{4})\alpha_0 + \frac{1}{4}\alpha_\tau = \frac{3}{4}\alpha_0 + \frac{1}{4}\alpha_\tau$
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	$0 + \alpha_\tau$
...		α_τ
$k+1$		α_τ



ADAPTIVE LEARNING RATES

- Learning rates are 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 automaticaly 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 all of their historical squared values.
  - 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

Algorithm 3 Adagrad

require Global learning rate α
require initial parameter θ
require Small constant β, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $r = 0$
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: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(y^{(i)}, f(\tilde{x}^{(i)}, \theta))$
    Accumulate squared gradient $r \leftarrow r + g \odot g$
    Compute update: $\nabla\theta = -\frac{\alpha}{\beta + \sqrt{(r)}} \odot g$ (division and square root applied element-wise)
    Apply update: $\theta \leftarrow \theta + \nabla\theta$
end while

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

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

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 (Goodfellow et al. (2016)).

RMSPROP

Algorithm 4 RMSProp

require Global learning rate α and decay rate ρ
require initial parameter θ
require Small constant β, perhaps 10^{-6}, to stabilize division by small numbers
Initialize gradient accumulation variable $r = 0$
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: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(y^{(i)}, f(\tilde{x}^{(i)}, \theta))$
    Accumulate squared gradient $r \leftarrow \rho r + (1 - \rho) g \odot g$
    Compute update: $\nabla\theta = -\frac{\alpha}{\beta + \sqrt{(r)}} \odot g$
    Apply update: $\theta \leftarrow \theta + \nabla\theta$
end while

ADAM

- Adaptive Moment Estimationis 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 gradient (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

Algorithm 5 Adam

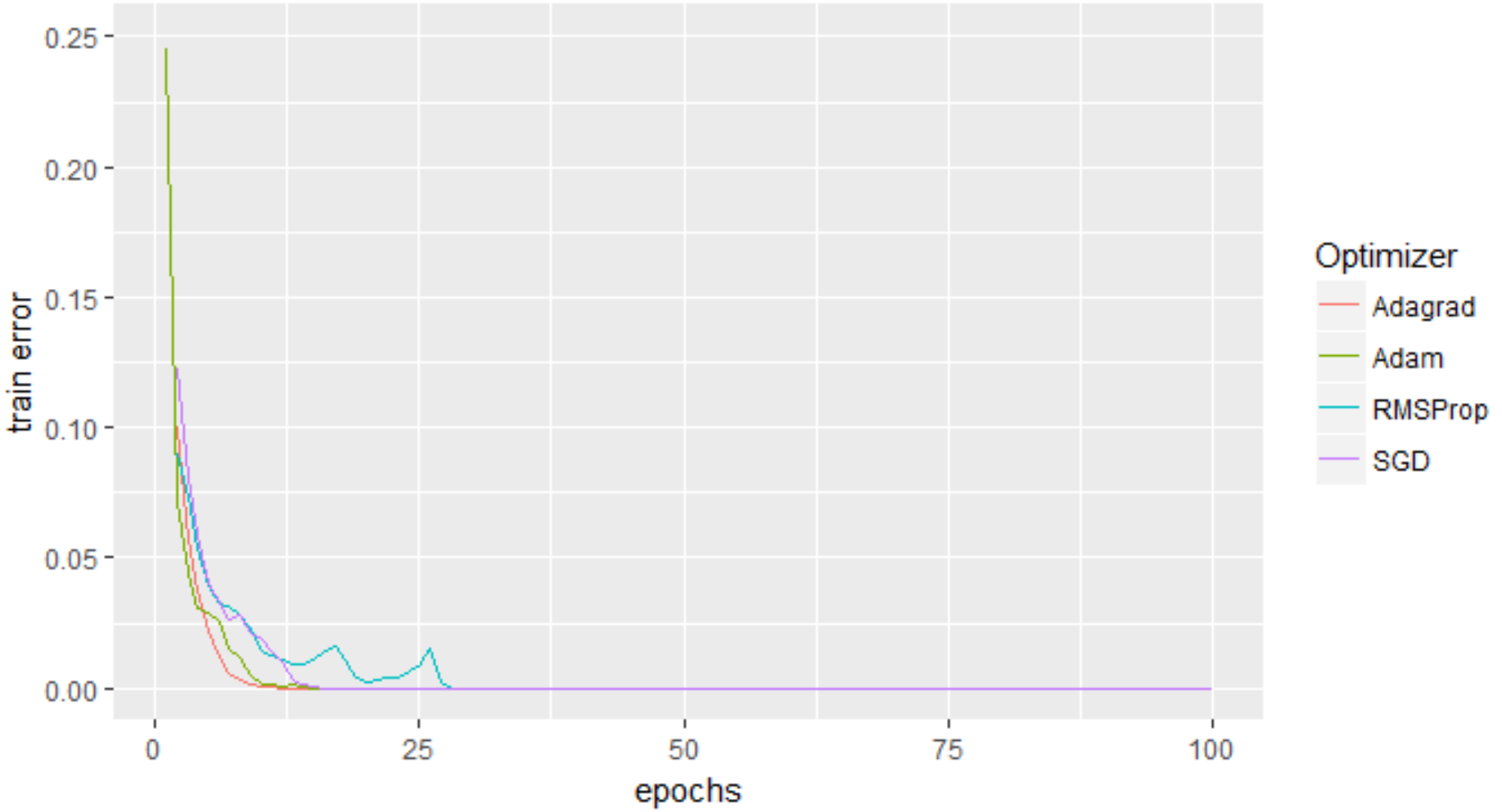
require: Step size μ (suggested default: 0.001)
require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (Suggested defaults: 0.9 and 0.999 respectively)
require: Small constant β (Suggested default 10^{-8})
require: Initial parameters θ
Initialize 1st and 2nd moment variables $s = 0, r = 0$
Initialize time step $t = 0$
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: $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(y^{(i)}, f(\tilde{x}^{(i)}, \theta))$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
    Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
    Correct bias in first moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
    Accumulate squared gradient $r \leftarrow \rho r + (1 - \rho) g \odot g$
    Compute update: $\nabla\theta = -\mu \frac{\hat{s}}{\sqrt{\hat{r}}+\beta}$
    Apply update: $\theta \leftarrow \theta + \nabla\theta$
end while

SGD VS. ADAPTIVE LEARNING RATES

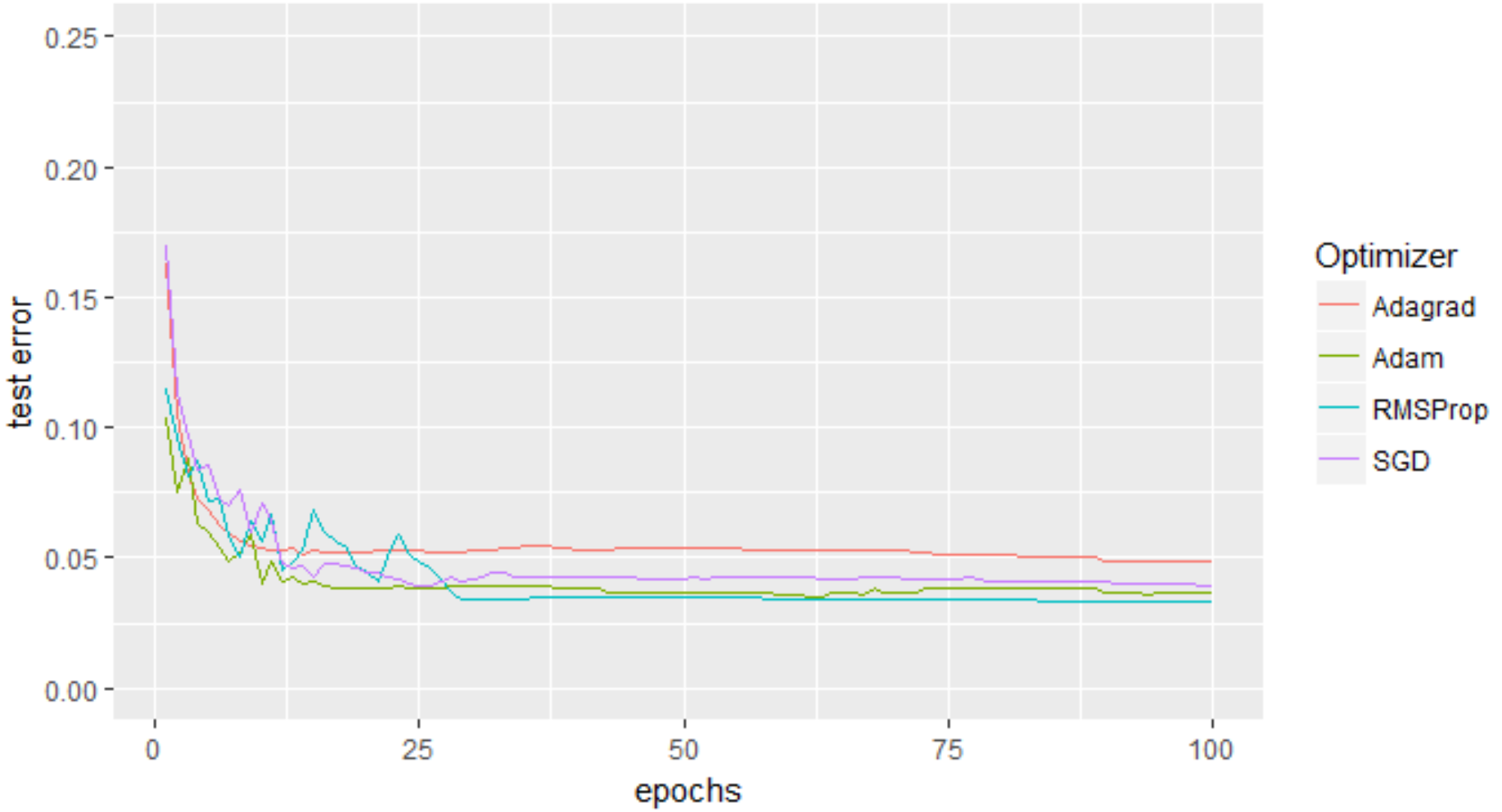
- Let us apply our standard architecture on the mnist problem and try our different optimizers.
- These are:

```
mx.opt.sgd(wd = 0.001, learning.rate = 0.03,
  momentum = 0.9)

mx.opt.adagrad(wd = 0.001, learning.rate = 0.05,
  epsilon = 1e-08)

mx.opt.rmsprop(wd =  0.001, learning.rate = 0.002,
  gamma1 = 0.95, gamma2 = 0.9)

mx.opt.adam(wd = 0.001, learning.rate = 0.001,
  beta1 = 0.9, beta2 = 0.999, epsilon = 1e-08)
```

SGD VS. ADAPTIVE LEARNING RATES



SGD VS. ADAPTIVE LEARNING RATES



BATCH NORMALIZATION

Motivation:

As shown earlier, neural networks learn a nonlinear transformation of the input space such that in the last layer(s), a simple classification is sufficient to seperate our data well. To do so, especially in deep networks, we need to coordinate updates between the layers. Batch normalization forces the model to learn a nonlinear transformation in a layer by removing changes in mean and standard deviation from the layers output.

BATCH NORMALIZATION

- Batch Normalization is no algorithm, but rather a technique to improve optimization in certain situations.
- It is an extra component that can be placed between each layer of the neural network.
- That component takes the activation of the antecedent layer and normalizes it before sending it to the next “actual layer”.

$$H' = \frac{H - \mu}{\sigma}$$

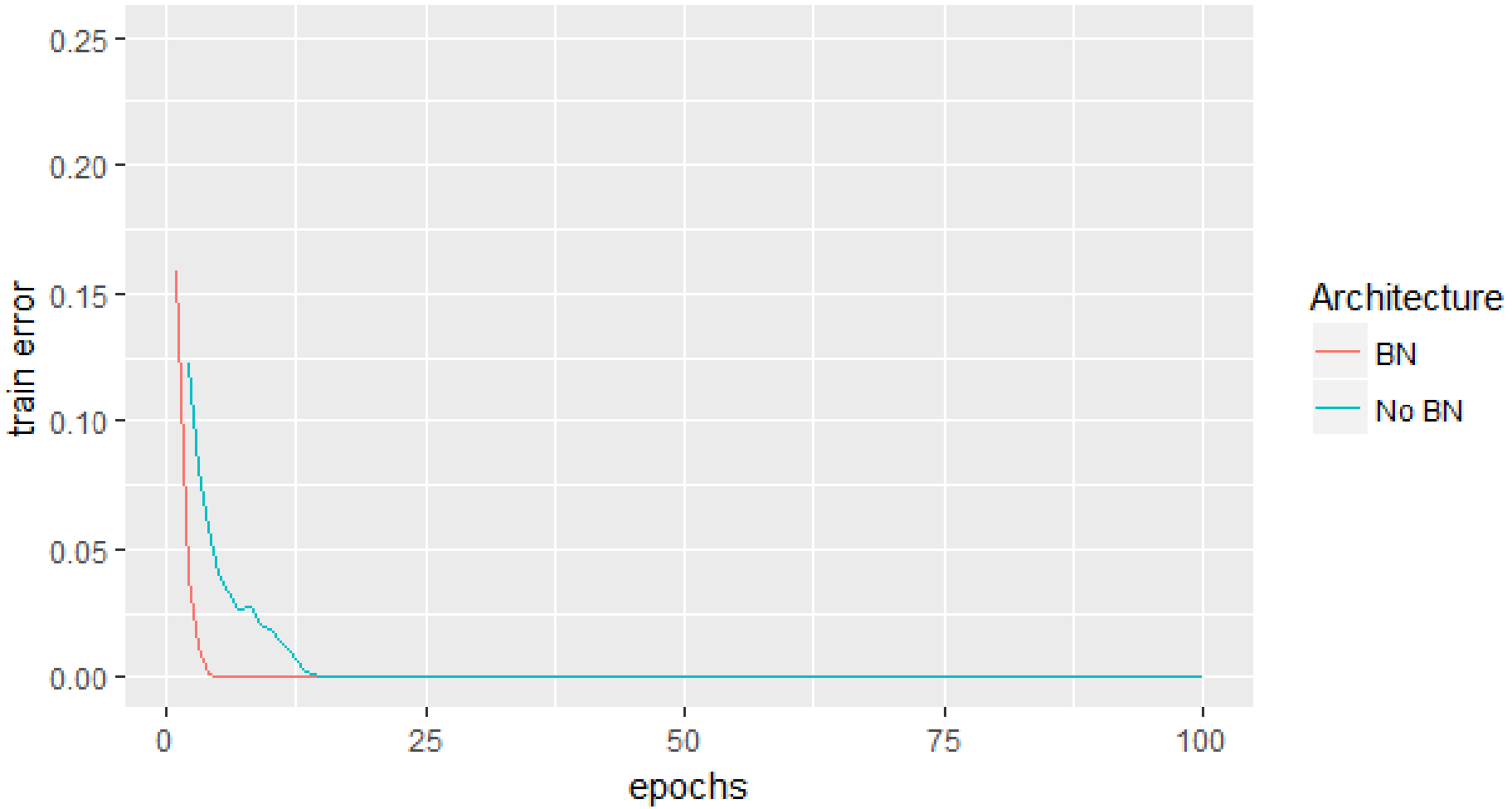
with H being the activated minibatch of the previous layer, μ the mean and σ the standard deviation of each unit.

BATCH NORMALIZATION

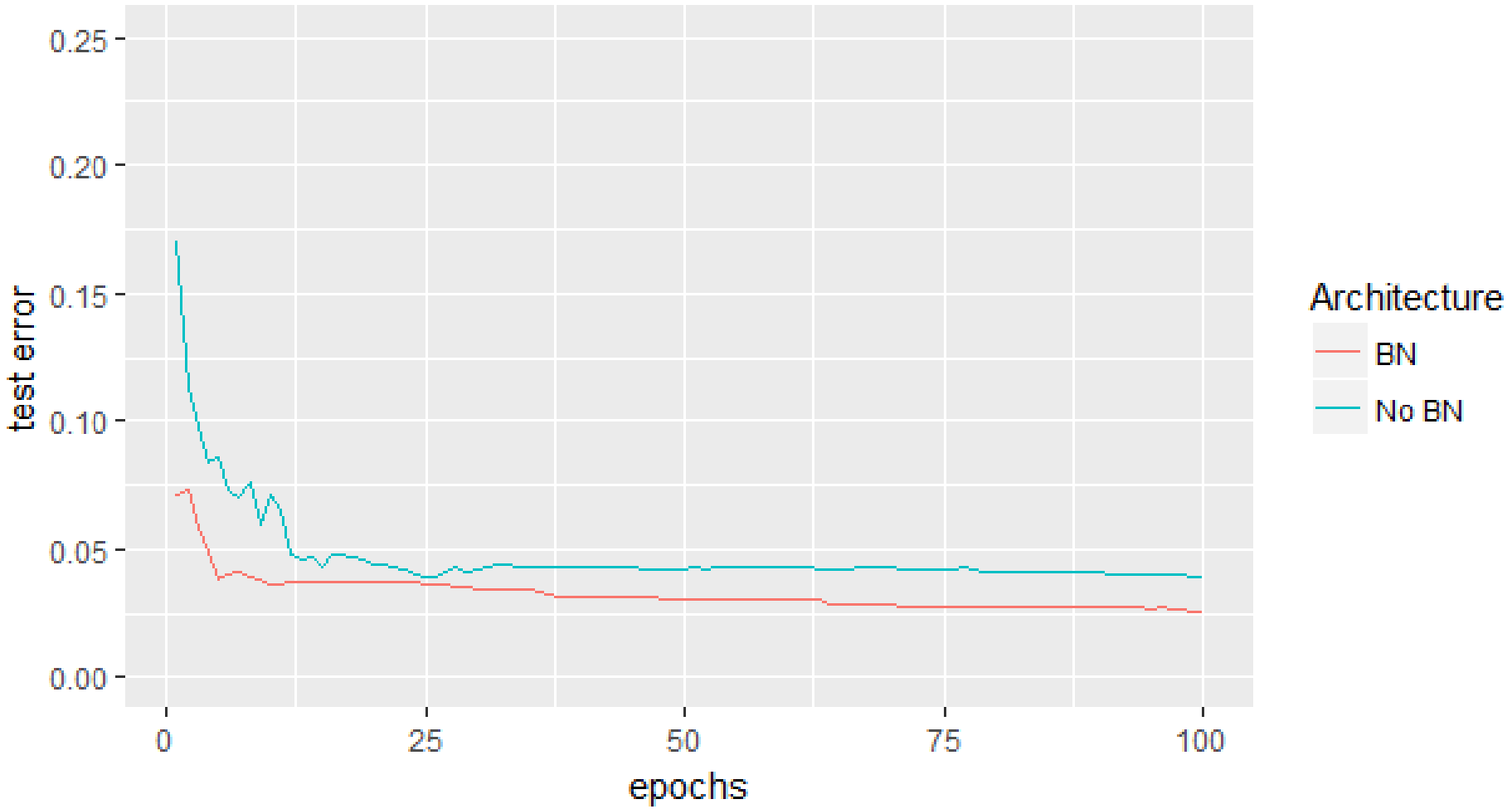
- For our final benchmark in this chapter we compute two models to predict the mnist data.
- One will extend our basic architecture such that we add batch normalization to all hidden layers.
- We use SGD as optimizer with $momentum = 0.9$, $learning.rate = 0.03$ and $wd = 0.001$.

```
fc1 = mx.symbol.FullyConnected(data, num_hidden = 512)
act1 = mx.symbol.Activation(fc1, act_type = "relu")
bn1 = mx.symbol.BatchNorm(act1)
fc2 = mx.symbol.FullyConnected(bn1, num_hidden = 512)
act2 = mx.symbol.Activation(fc2, act_type = "relu")
bn2 = mx.symbol.BatchNorm(act2)
fc3 = mx.symbol.FullyConnected(bn2, num_hidden = 512)
act3 = mx.symbol.Activation(fc3, act_type = "relu")
bn3 = mx.symbol.BatchNorm(act3)
fc4 = mx.symbol.FullyConnected(bn3, num_hidden = 10)
softmax = mx.symbol.SoftmaxOutput(fc4, name = "sm")
```

BATCH NORMALIZATION



BATCH NORMALIZATION



SECOND ORDER OPTIMIZATION

- Second order optimization methods include information about the curvature but require computing the hessian matrix.
  - Very expensive for large networks!
  - Think of the network we applied in the regularization chapter:

```
layer1 = mx.symbol.FullyConnected(data, num_hidden = 512L)
activation1 = mx.symbol.Activation(layer1, act_type = "relu")
layer2 = mx.symbol.FullyConnected(activation1, num_hidden = 512L)
activation2 = mx.symbol.Activation(layer2, act_type = "relu")
layer3 = mx.symbol.FullyConnected(activation2, num_hidden = 512L)
activation3 = mx.symbol.Activation(layer3, act_type = "relu")
layer4 = mx.symbol.FullyConnected(activation3, num_hidden = 10L)
softmax = mx.symbol.SoftmaxOutput(layer4, name = "softmax")
```

- This model has a huge amount of parameters:

$$
\begin{aligned}
&= \underbrace{784 \cdot 512}_{\text{input to 1st layer}} + \underbrace{512^2}_{\text{1st to 2nd layer}} + \underbrace{512^2}_{\text{2nd to 3rd layer}} + \underbrace{512 \cdot 10}_{\text{3rd to output}} \\
&= 3.285.504
\end{aligned}
$$

SECOND ORDER OPTIMIZATION

- A first order method would need 3.285.504 partial derivatives to compute the gradients.
- Second order methods on the other hand require

$$3.285.504^2 = 10.794.536.534.016$$

  partial derivatives!
- For small problems/networks one might consider the BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm, which computes an approximation of the hessian (still expensive!).
  - Very difficult to implement, alot of programming know-how necessary to create an efficient implementation.
- For huge networks even gradient descent becomes too expensive.
  - Solution: parameters are instead grouped into mini-batches (stochastic gradient descent)

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