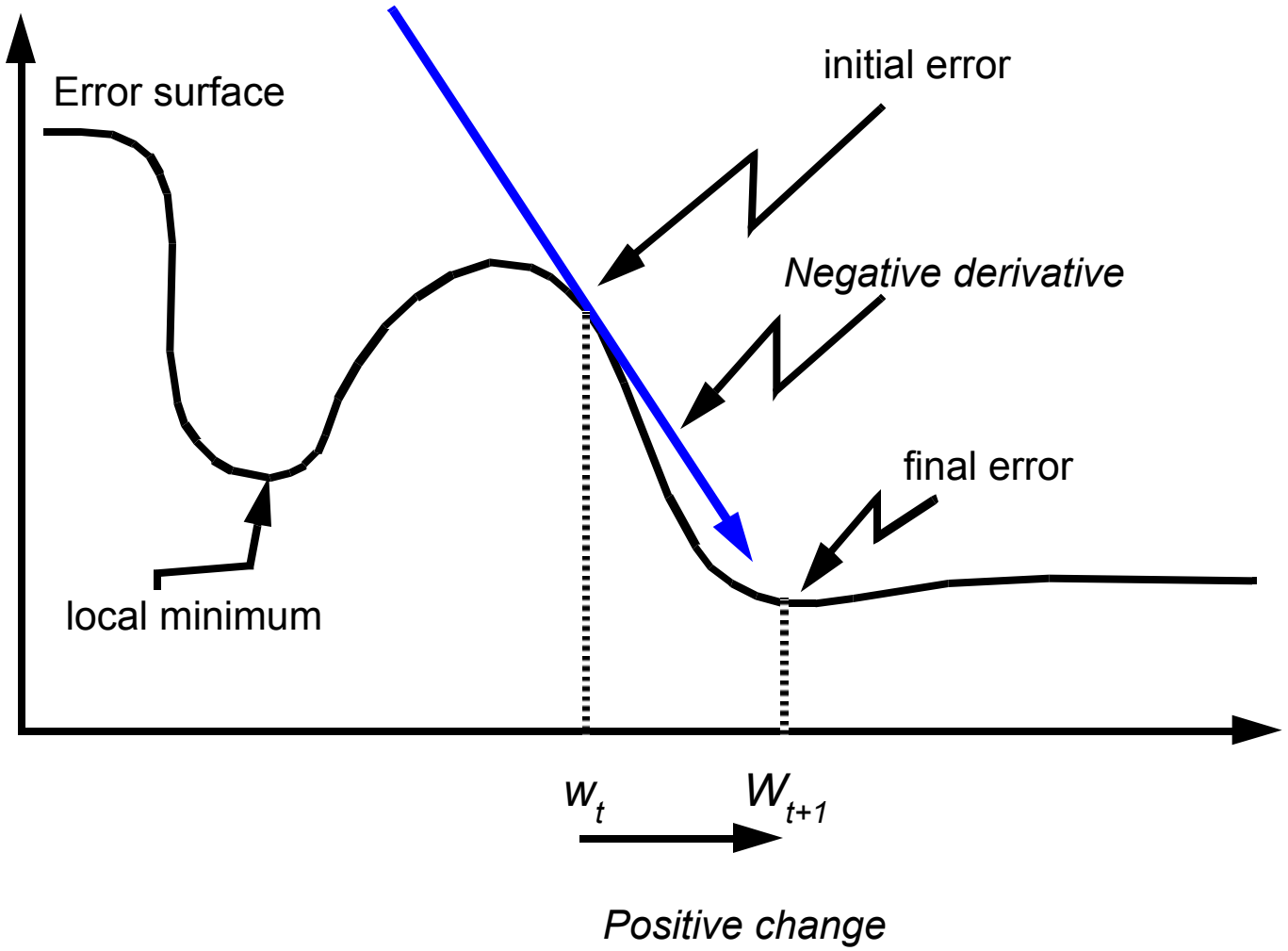
Deep Learning

Gradient Descent & Optimizers

Dr. Omri Allouche
Y-Data Deep Learning Course

Minimizing the Error



Gradient Descent

Suppose we have a **scalar function**

We want to find a local minimum.

Assume our current weight is *w*

$$f(w) : \Re \rightarrow \Re$$

GRADIENT DESCENT RULE:

$$w_{t+1} \leftarrow w_t - \eta \frac{\partial}{\partial w} f(w_t)$$

η is called the LEARNING RATE.

It's a small positive number, e.g. η = 0.05

Gradient Descent in "M" Dimensions

Given $\mathbf{f(w)} : \Re^m \rightarrow \Re$

$$\nabla \mathrm{f}(\mathrm{w}) = \begin{pmatrix} \frac{\partial}{\partial w_1} \mathrm{f}(\mathrm{w}) \\ \vdots \\ \frac{\partial}{\partial w_m} \mathrm{f}(\mathrm{w}) \end{pmatrix}$$ points in direction of steepest ascent.

$|\nabla \mathrm{f}(\mathrm{w})|$ is the gradient in that direction

GRADIENT DESCENT RULE: $\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} - \eta \nabla \mathbf{f}(\mathbf{w_t})$

Equivalently $w_{t+1,j} \leftarrow w_{t,j} - \eta \frac{\partial}{\partial w_{t,j}} \mathbf{f}(\mathbf{w_t})$ … where w_j is the j_{th} weight

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

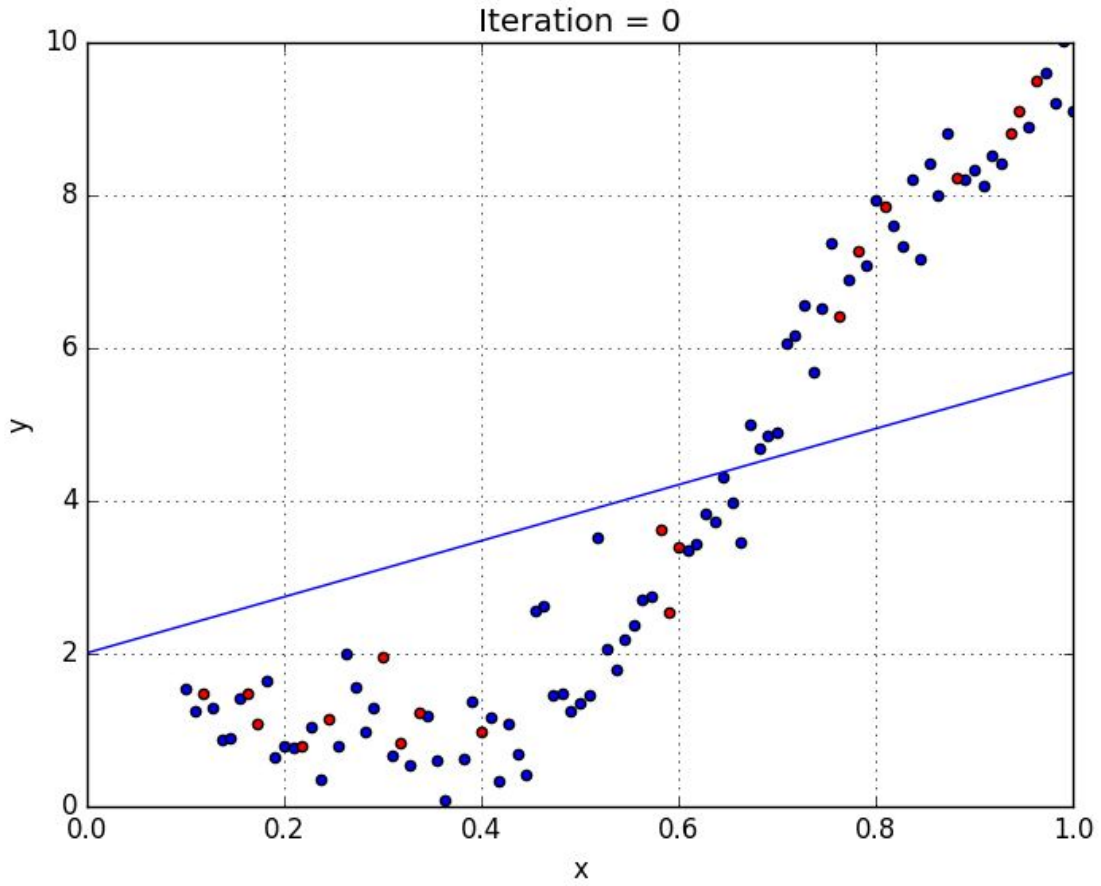
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$

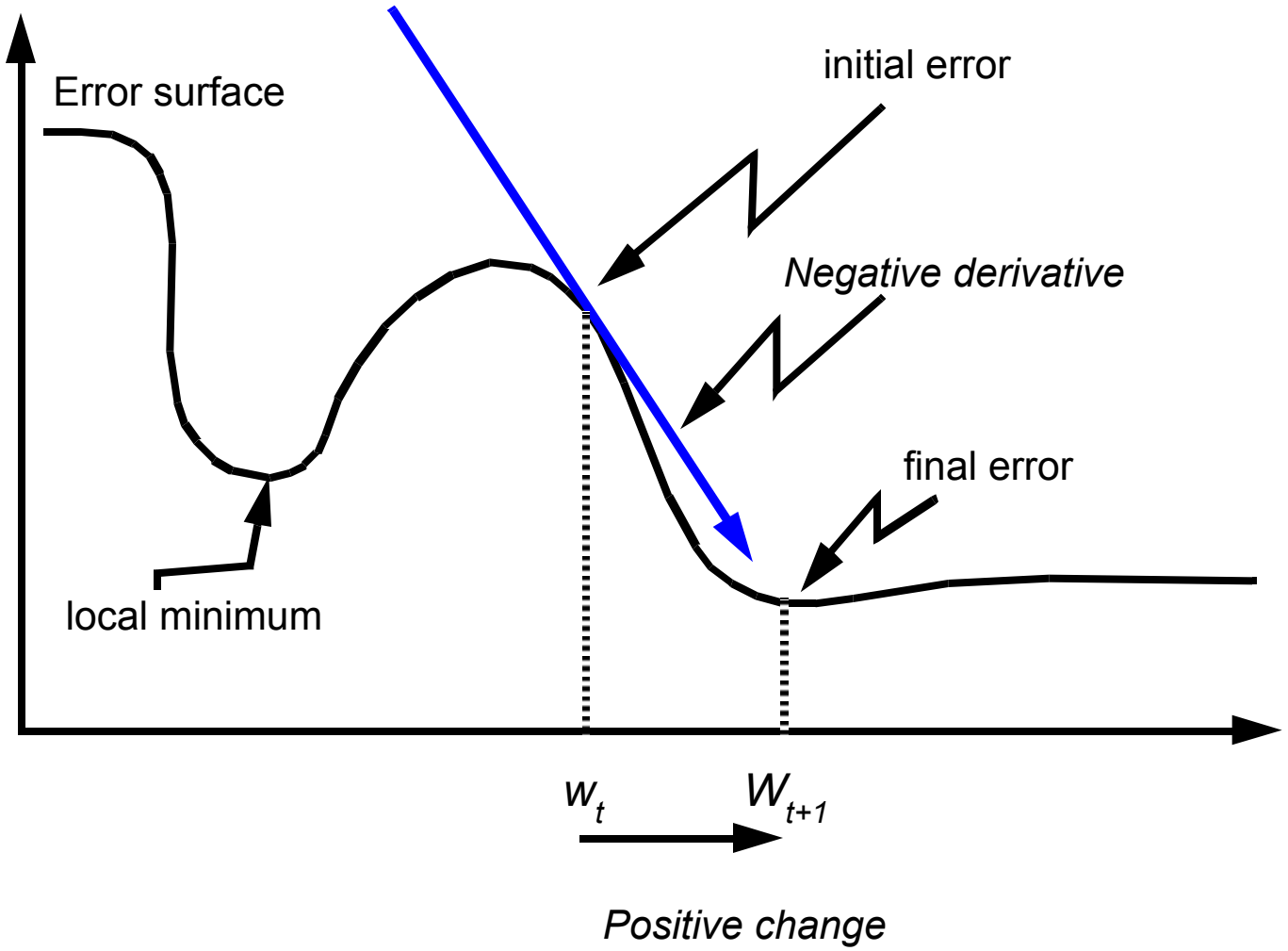
$k \leftarrow k + 1$

end while

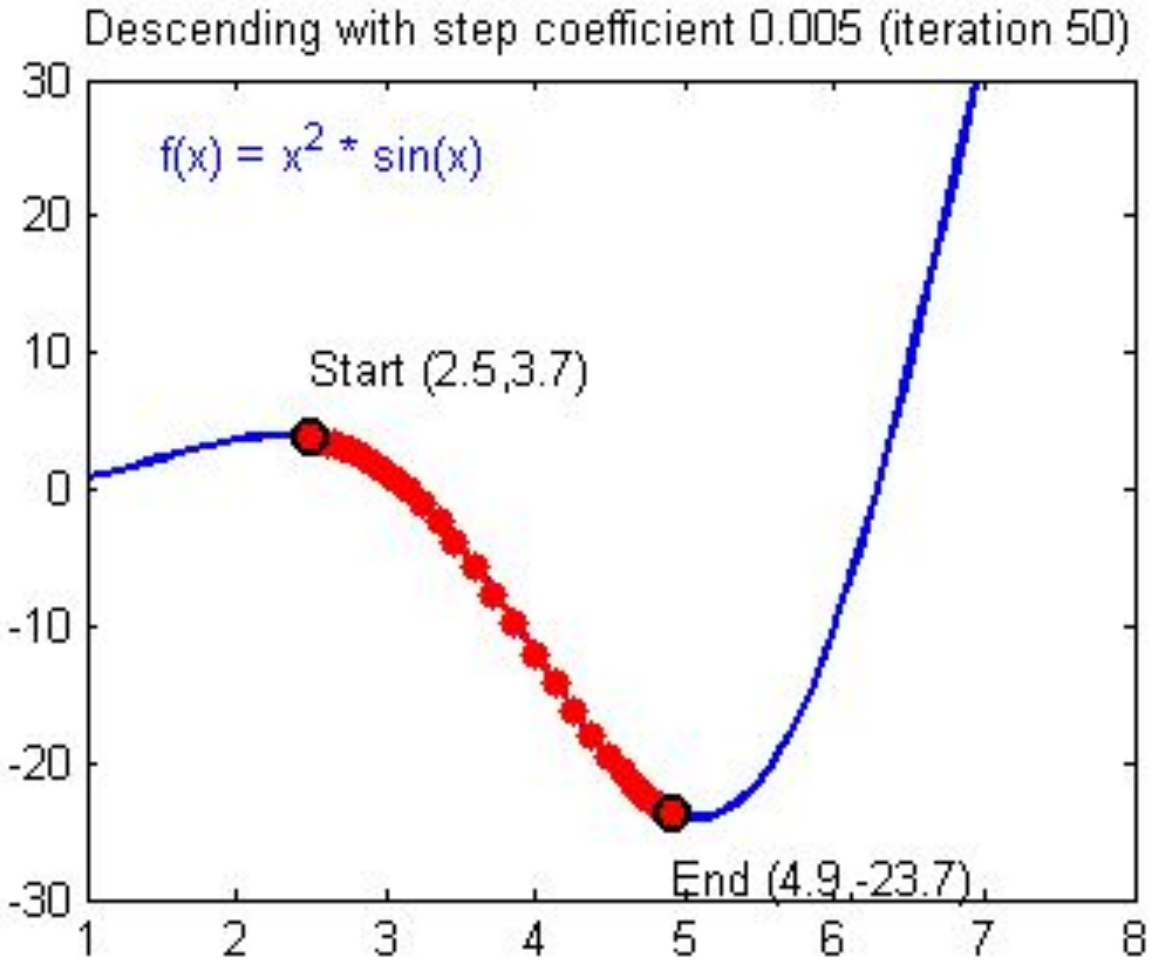
Using Gradient Descent for Solving Linear Regression



Minimizing the Error



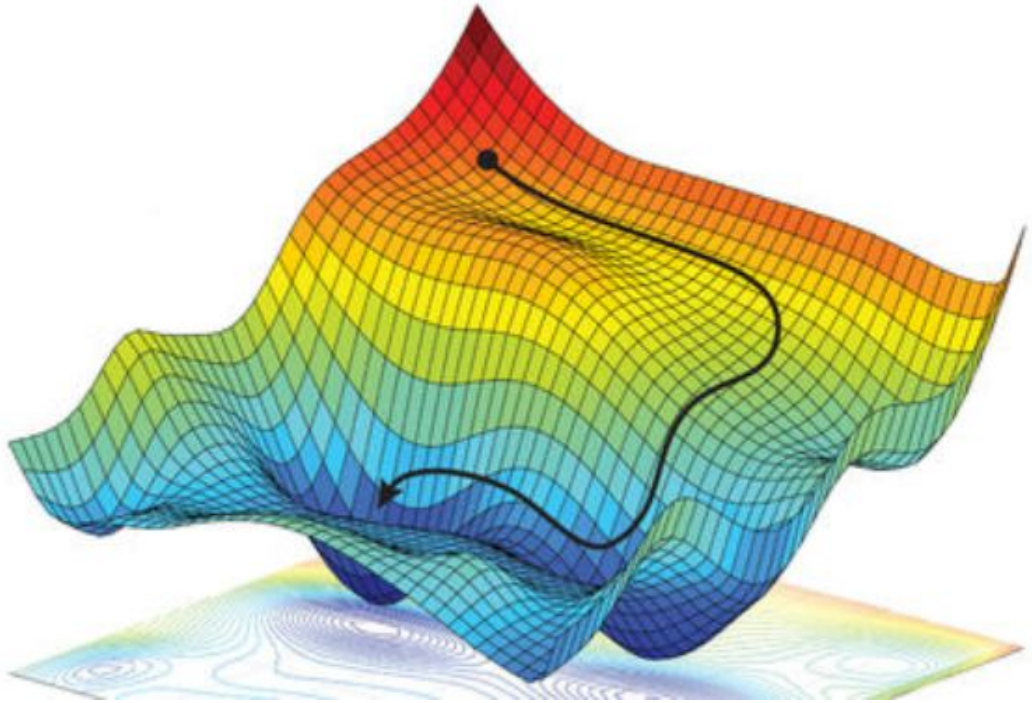
Illustration

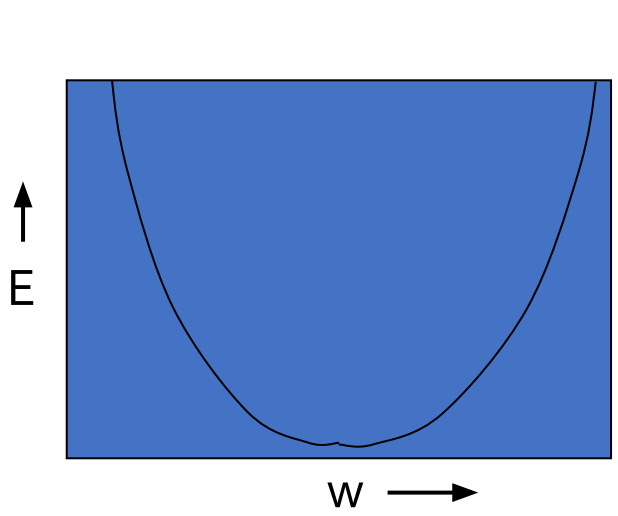


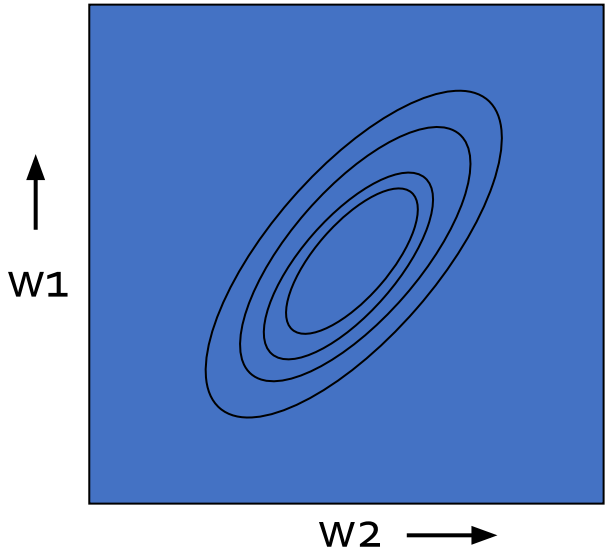
Code in Python can be obtained from: https://www.cs.toronto.edu/~frossard/post/linear_regression/

The error surface

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
  - It is a quadratic bowl
  - Vertical cross-sections are parabolas
  - Horizontal cross-sections are ellipses



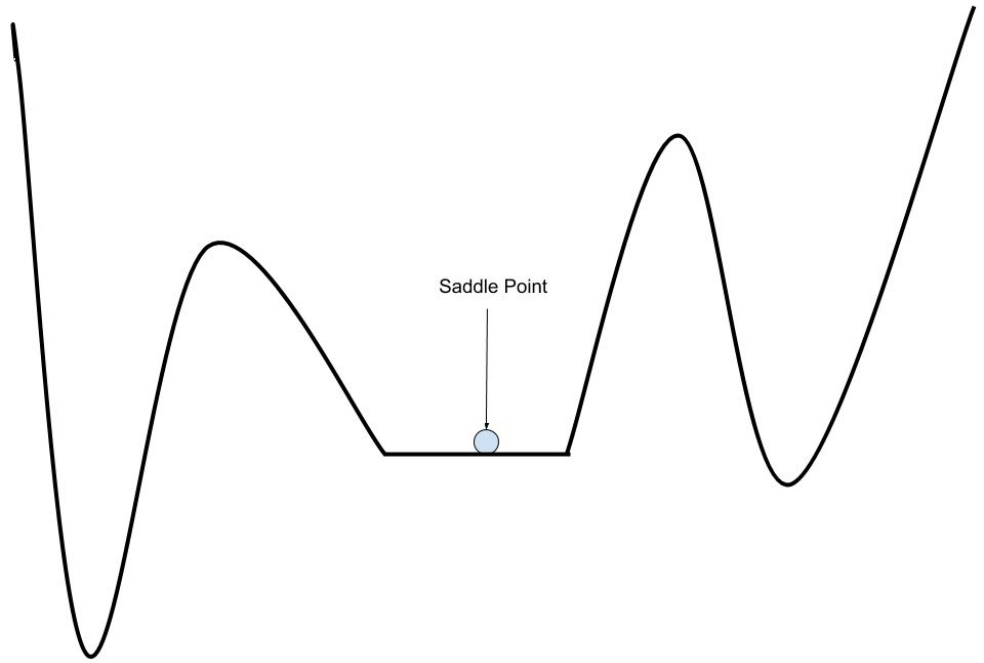




The Error Surface

- We only perceive 3 dimensions, but the dimensionality of the error surface is equal to the number of parameters
- Using gradient descent methods mean we might converge to local minima
- Experience shows that in large networks, there are multiple local minima
- BUT, while small networks show a large variance in the loss of the minima, deep networks show a much lower variance (ie there are several possible solutions, but they are all roughly equal)

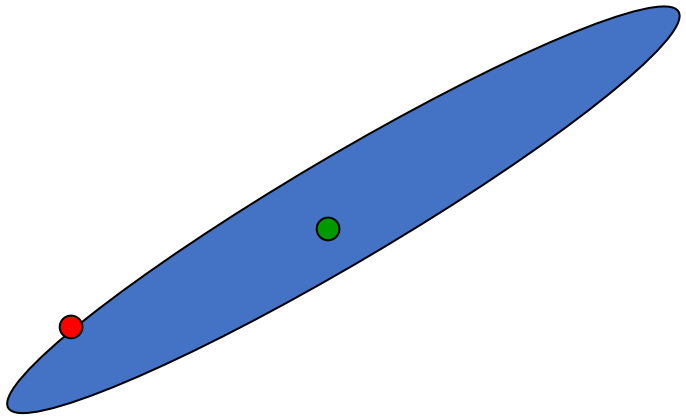
Hinton: "To deal with hyper-planes in a 14-dimensional space, visualize a 3-D space and say "fourteen" to yourself very loudly. Everyone does it."

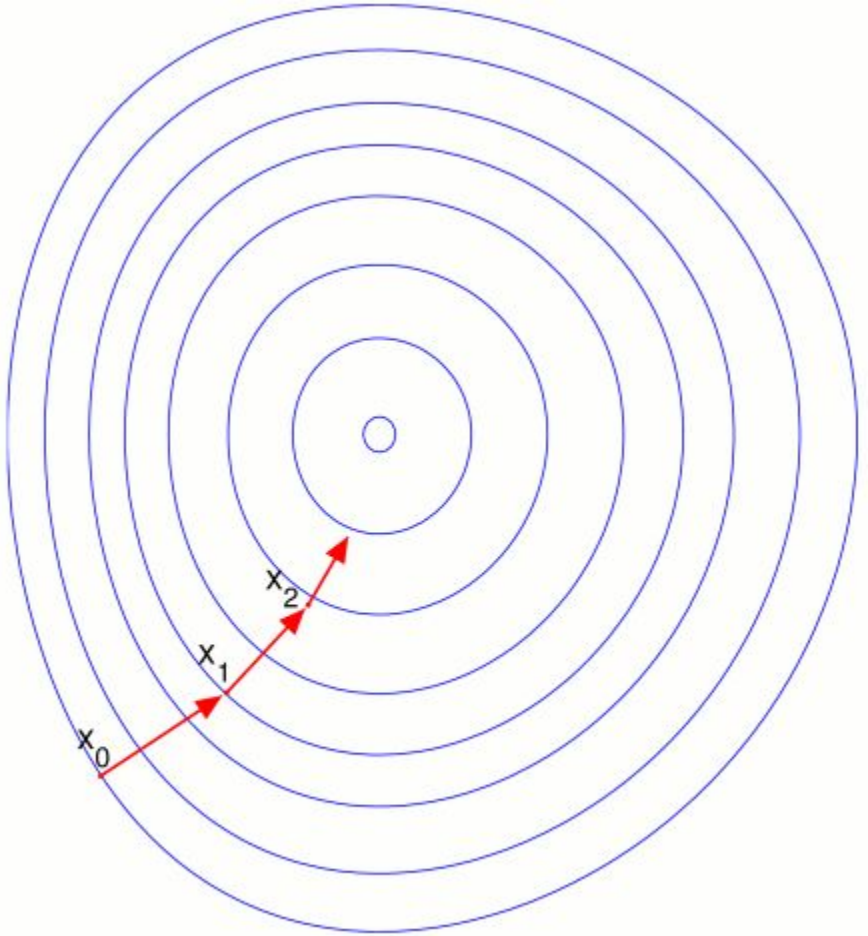


Choromanska et al 2015 https://arxiv.org/abs/1412.0233

Illustration of Gradient Descent

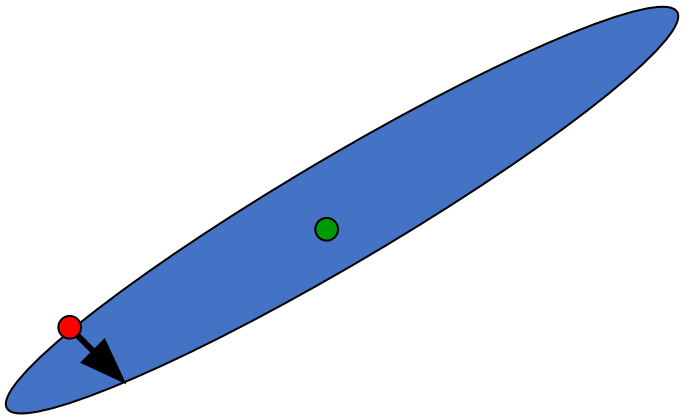
- How would gradient descent behave if the error surface is an elongated ellipse?





Convergence Speed

- The direction of steepest descent does not point at the minimum unless the ellipse is a circle
  - The gradient is large in the direction in which we only want to travel a small distance
  - The gradient is small in the direction in which we want to travel a large distance
- Scaling variables helps turn ellipses into circles

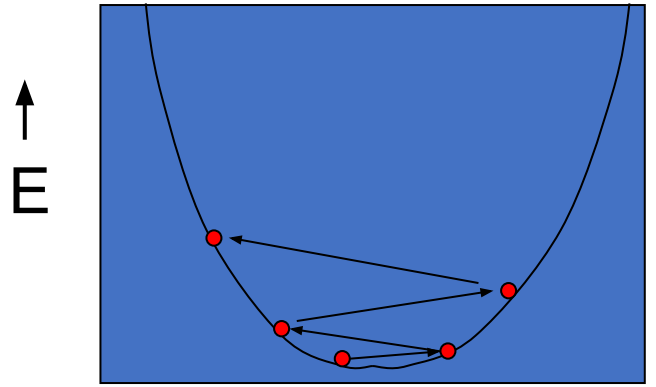


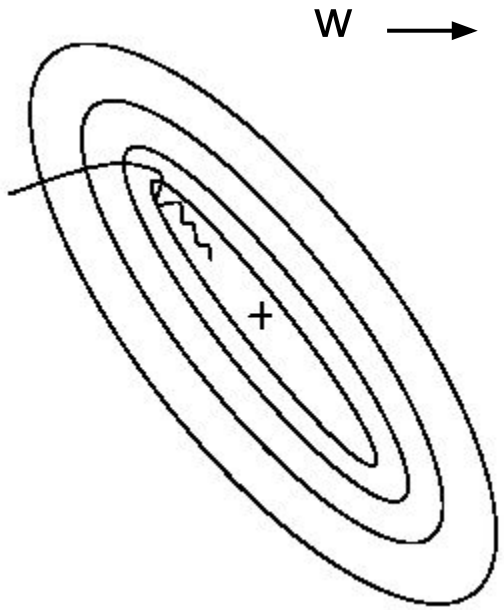
How learning can go wrong

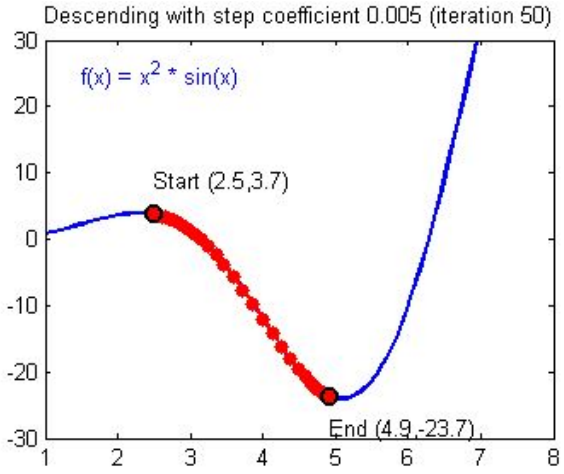
If the learning rate is too big, it sloshes to and from across the ravine.

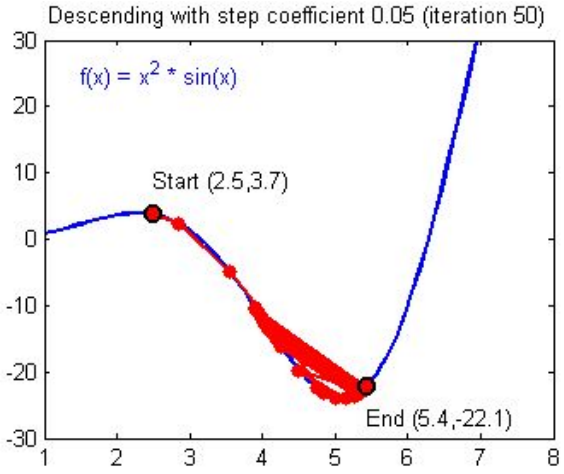
Solution:

- Use small learning rate
- But: if learning rate is too small, it may be slow to coverage



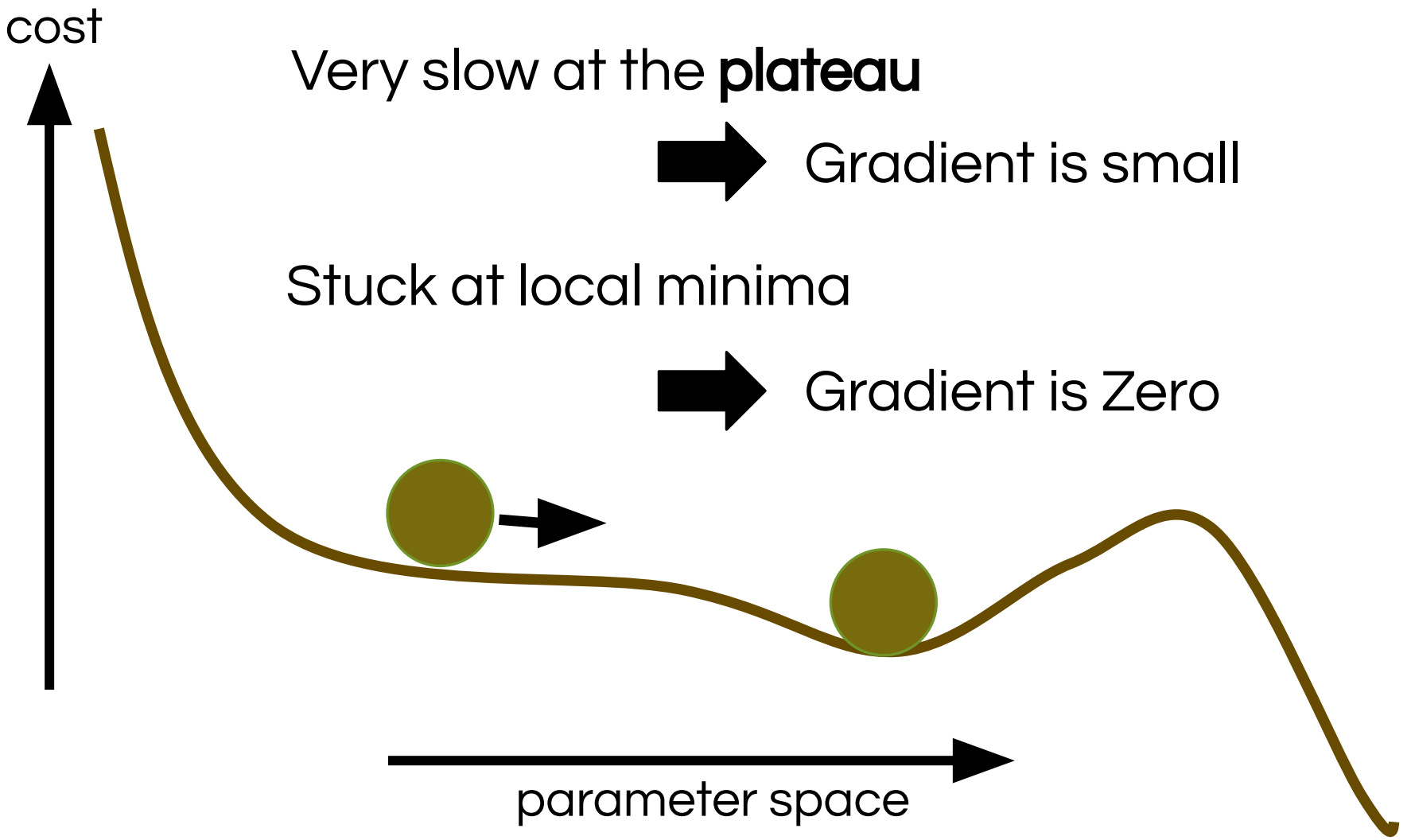






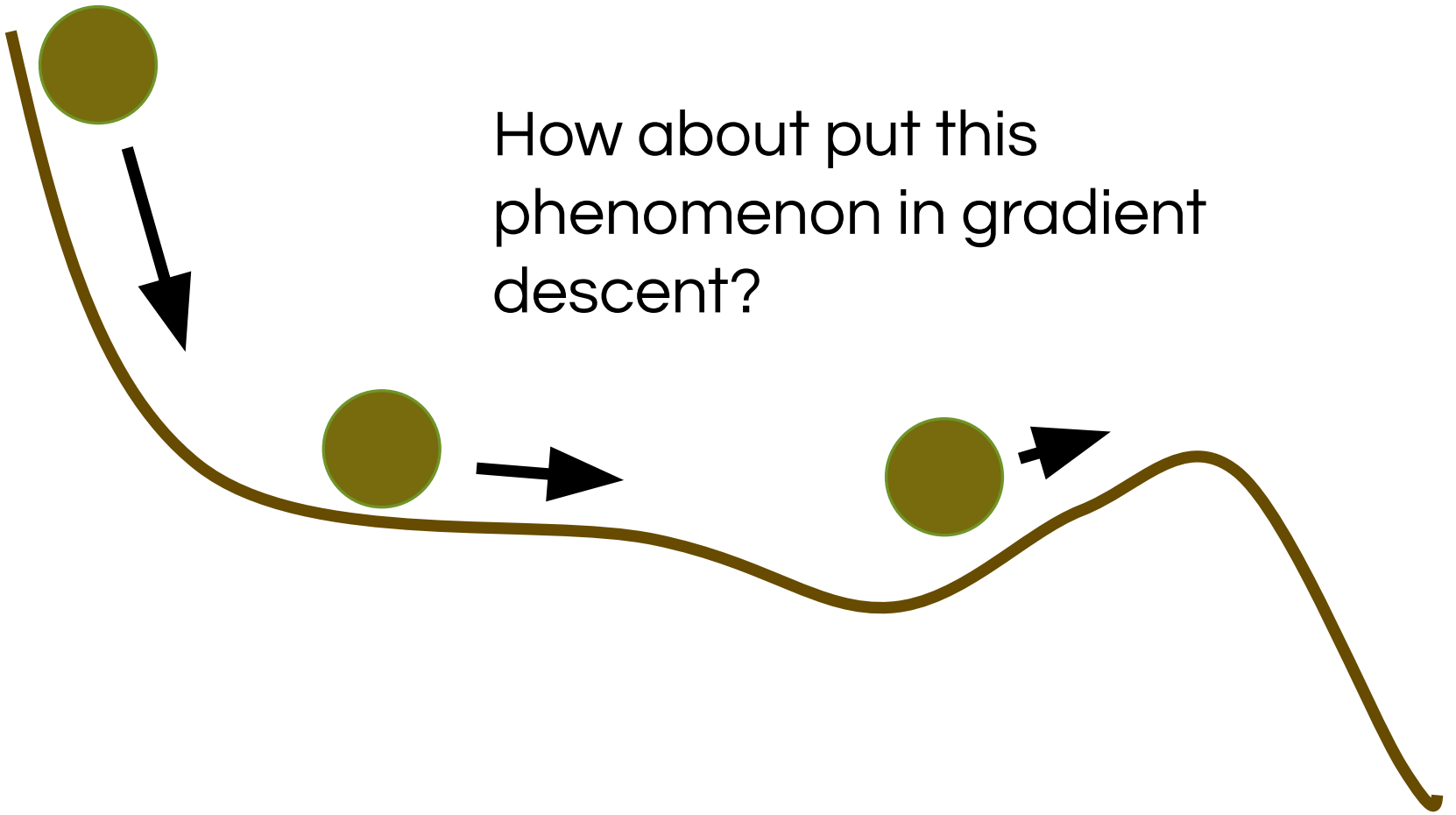
How can we improve convergence?

The problem of gradient descent

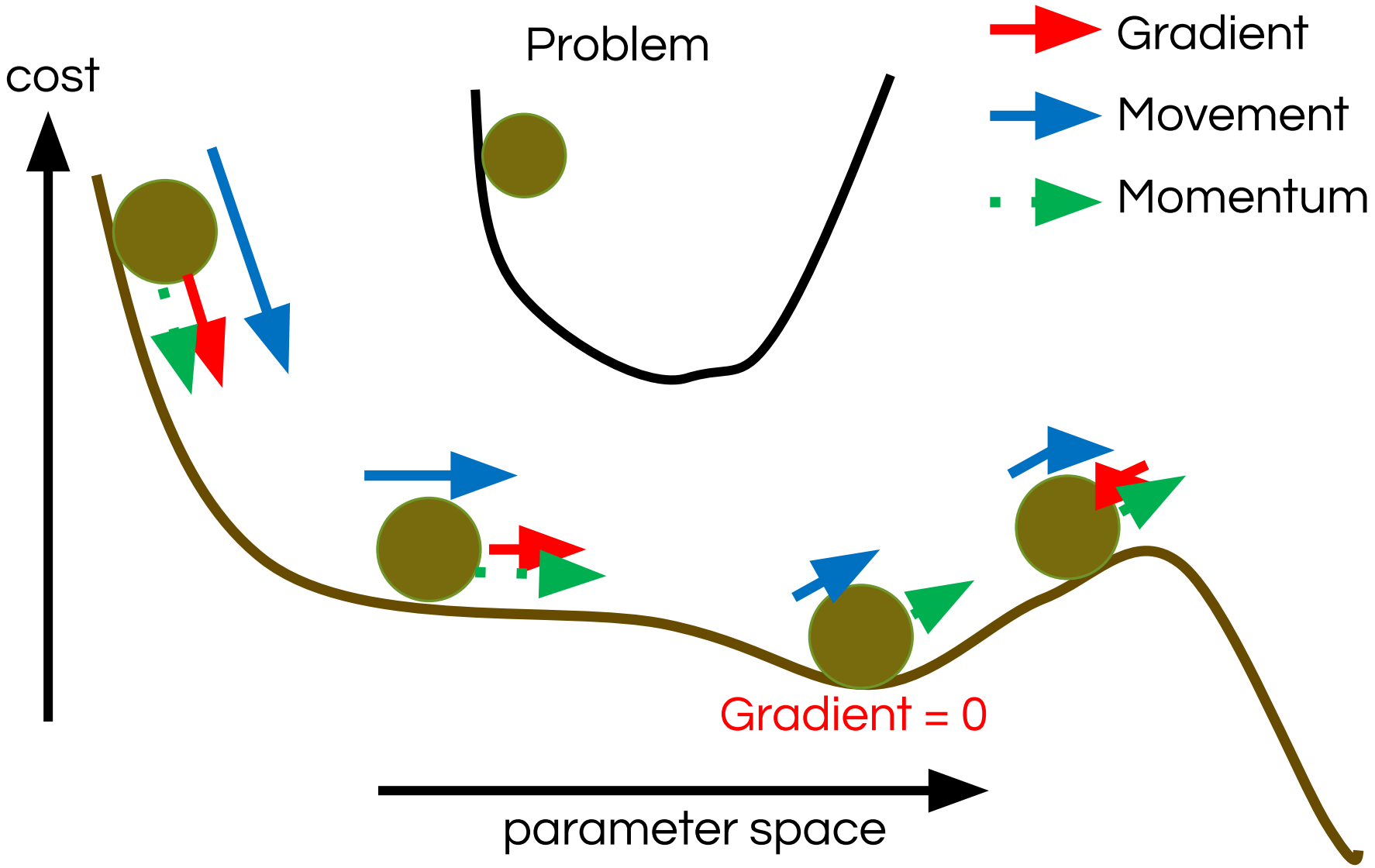


In the physical world ……

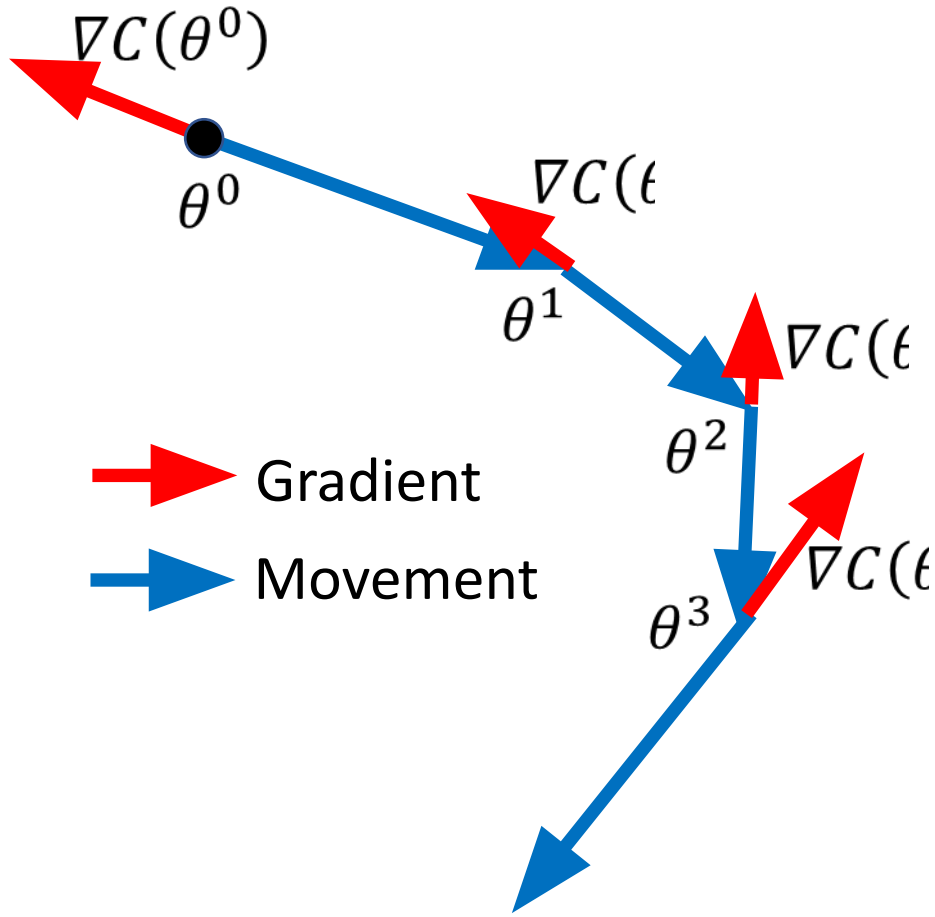
- Momentum



Gradient descent with Momentum



Original Gradient descent



Start at position θ^0

Compute gradient at θ^0

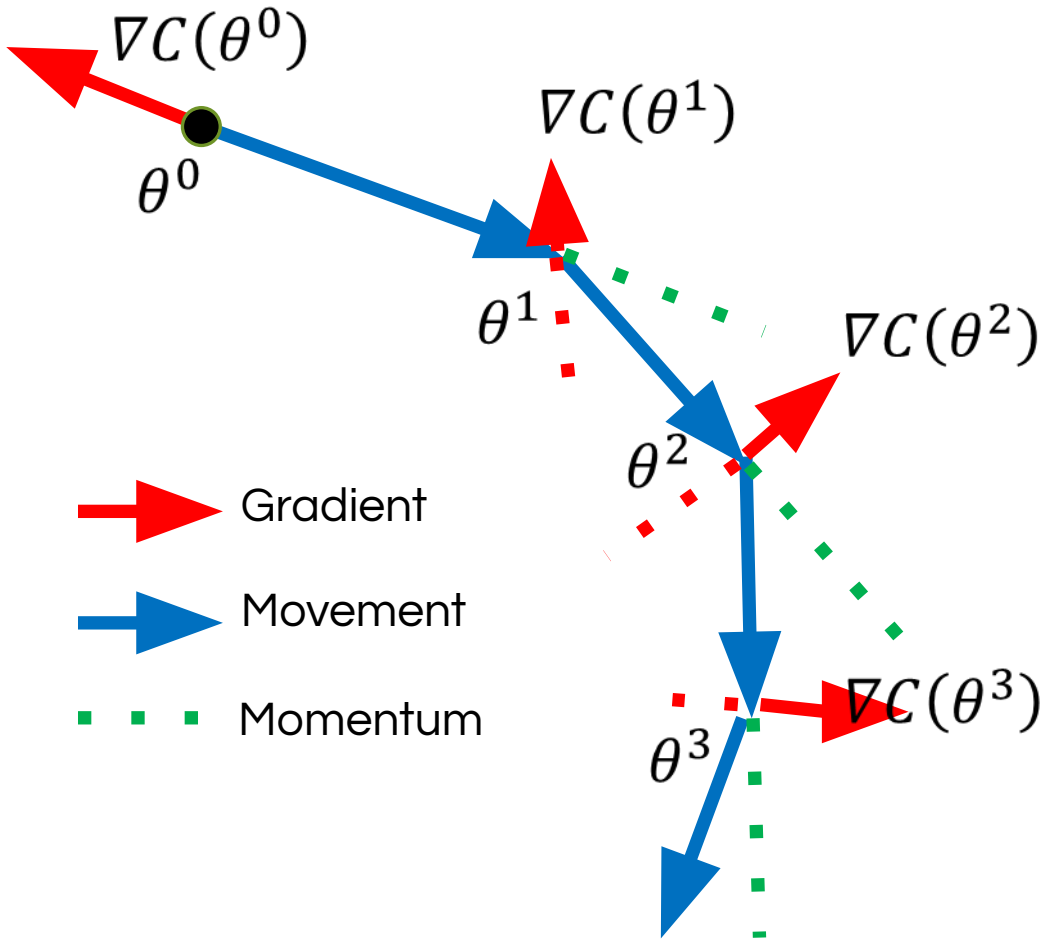
Move to $\theta^1 = \theta^0 - \mu\nabla C(\theta^0)$

Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \mu\nabla C(\theta^1)$

⋮ ⋮

Gradient descent with Momentum (Heavy ball acceleration)



Start at point θ^0

Momentum $v^0=0$

(v^i : movement at the i-th update)

Compute gradient at θ^0

Momentum $v^1 = \lambda v^0 - \mu \nabla C(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Momentum $v^2 = \lambda v^1 - \mu \nabla C(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

……

Gradient descent with Momentum

v^i is actually the weighted sum of all the previous gradient: $\nabla C(\theta^0), \nabla C(\theta^1), \ldots \nabla C(\theta^{i-1})$

$v^0=0$

$v^1 = -\mu \nabla C(\theta^0)$

$v^2 = -\lambda \mu \nabla C(\theta^0) - \mu \nabla C(\theta^1)$

……

Start at point θ^0

Momentum $v^0=0$

(v^i : movement at the i-th update)

Compute gradient at θ^0

Momentum $v^1 = \lambda v^0 - \mu \nabla C(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Momentum $v^2 = \lambda v^1 - \mu \nabla C(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

…

…

SGD with Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

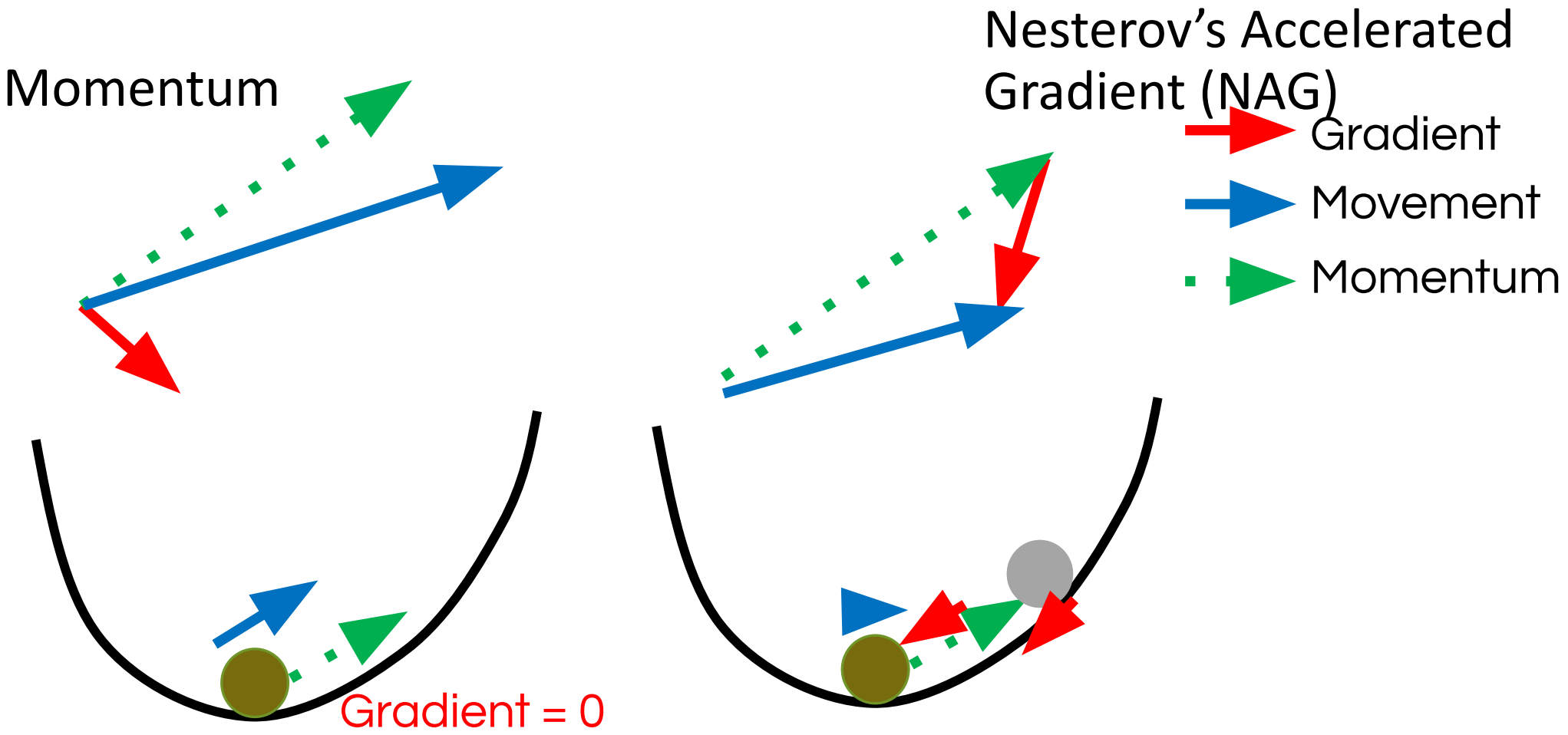
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Nesterov's Accelerated Gradient (NAG)



SGD with Nesterov Momentum

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.

Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

AdaGrad (Adaptive Gradient)

Divide the learning rate by the "***average" gradient***

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma} g^t$$

σ: Average gradient of parameter w

Estimated while updating the parameters

If w has small average gradient σ → Larger learning rate

If w has large average gradient σ → Smaller learning rate

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.

Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

AdaGrad Benefits

- Improves convergence for sparse data (e.g. NLP, Image recognition)
- The accumulated gradient norm increases at every iteration, and so the learning rate shrinks
- AdaGrad eliminates the need to update the learning rate – most often a value of 0.01 is used

RMSProp (Root Mean Square Propagation)

RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates

Algorithm:

Step 1. $G^{(t+1)} = \gamma G^{(t)} + (1-\gamma) \cdot \left(g^t\right)^2$

G is a moving exponential average of the square gradient.

Step 2. $w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{G^{(t+1)}} + 10^{-7}} \cdot g^t$

Scale the learning rate by $1/\sqrt{G}$ and follow the negative gradient direction.

Default settings: $\gamma = 0.9, \eta = 0.001.$

RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

RMSProp with Momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

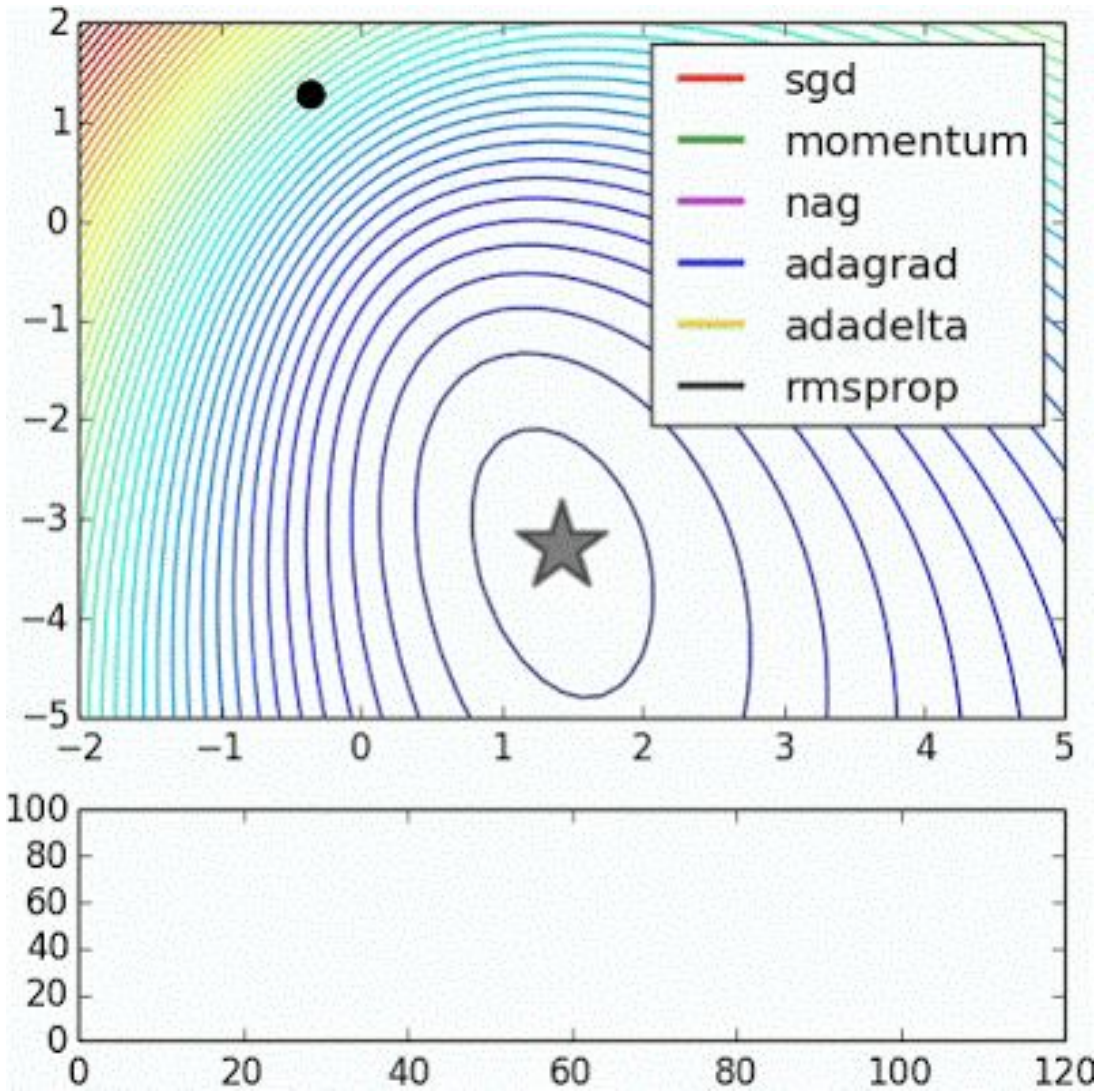
Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

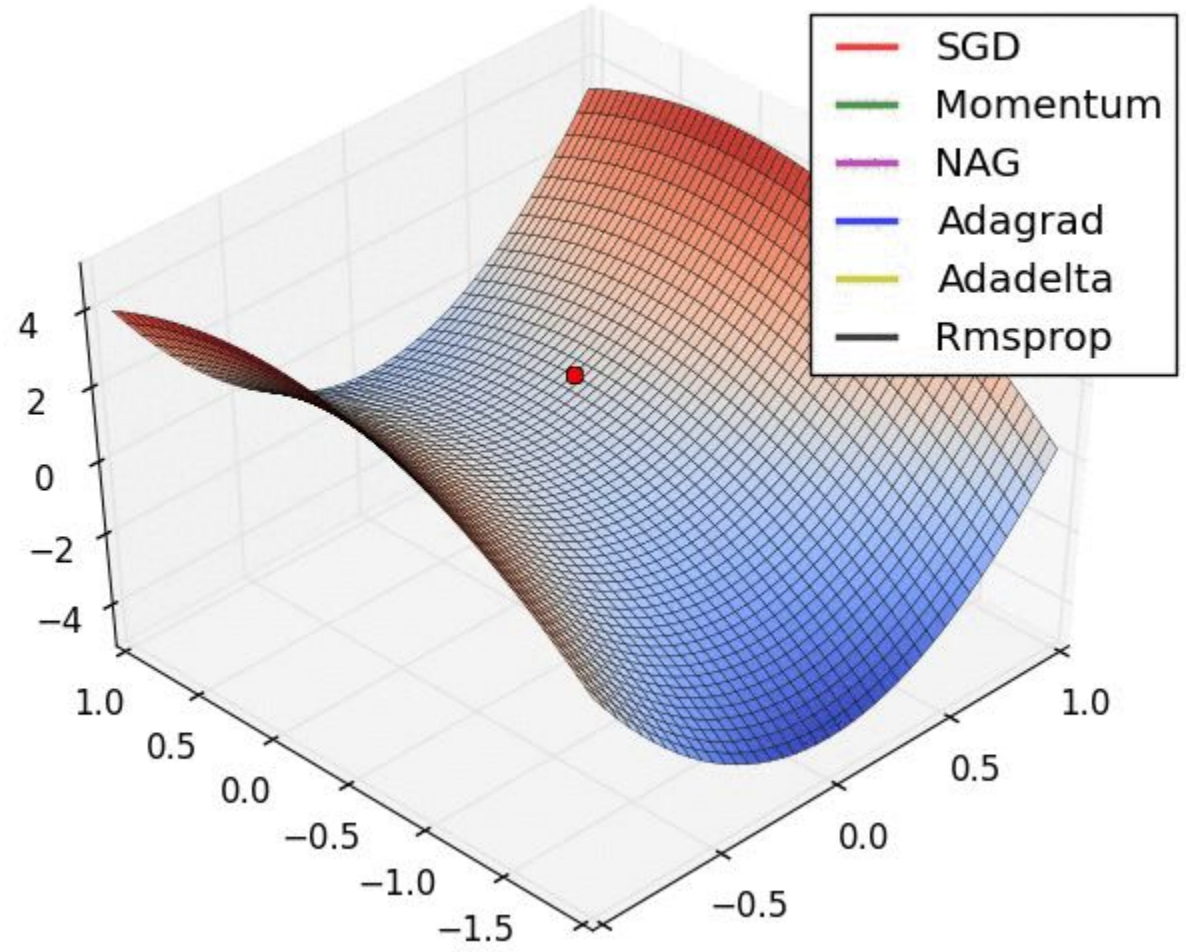
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

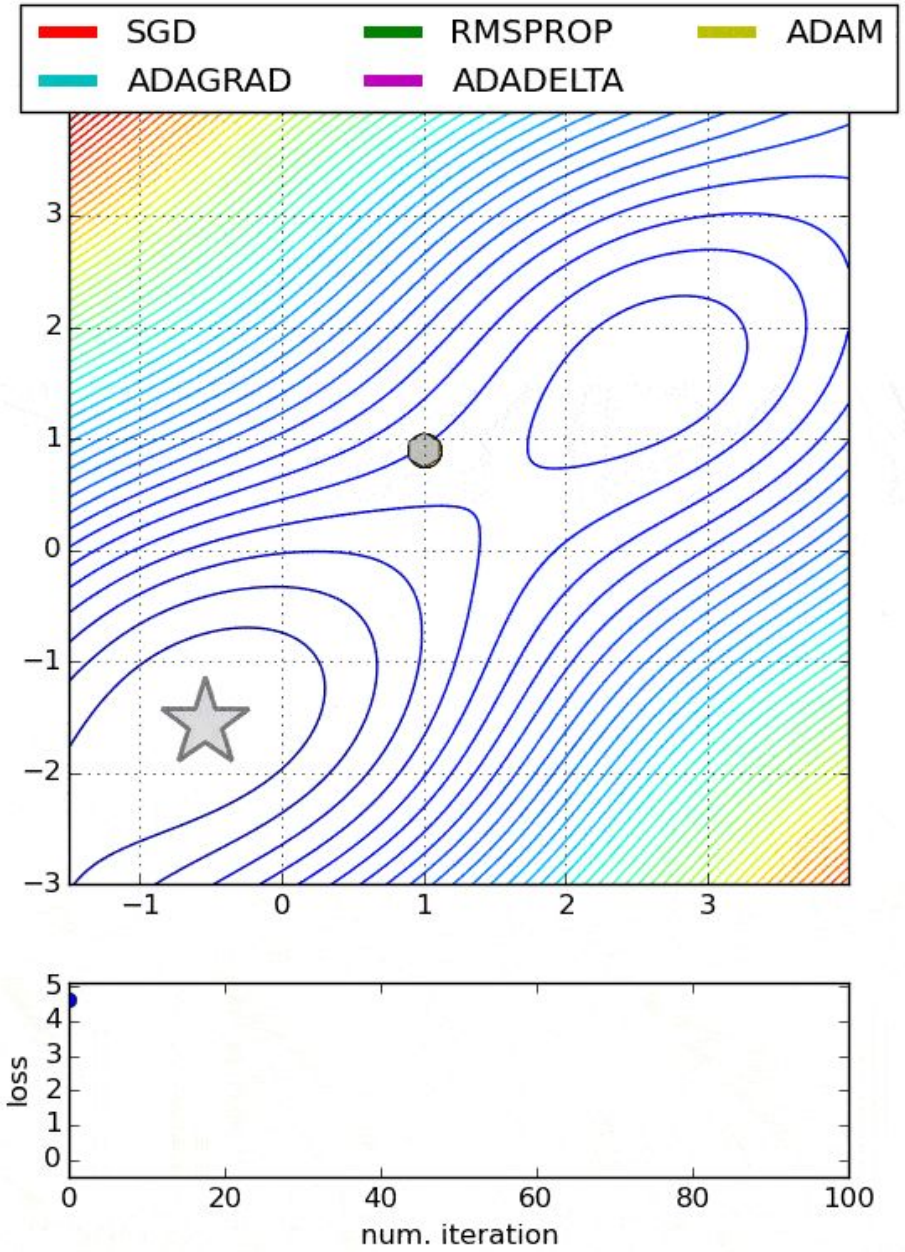
end while

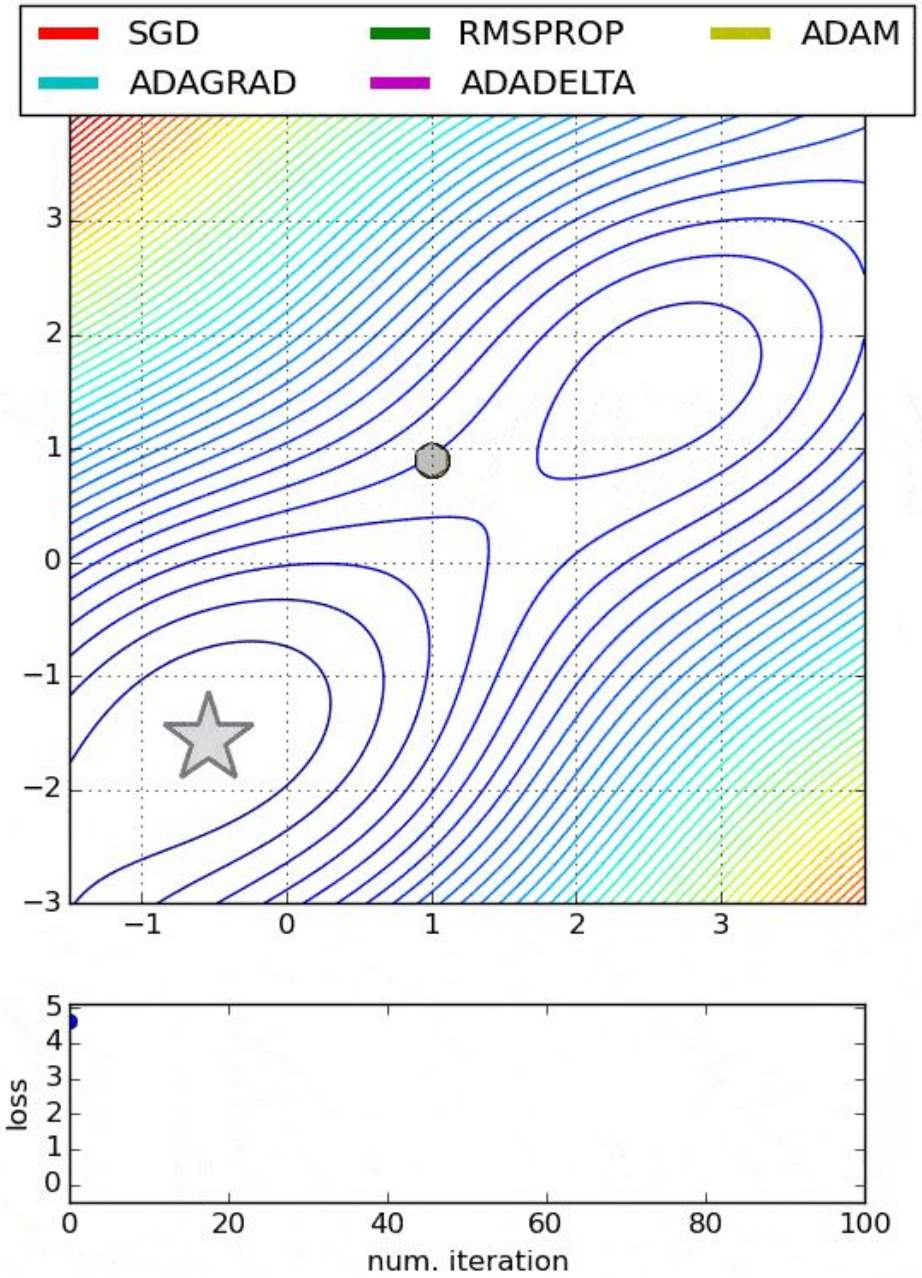
Many Flavors of Gradient Descent



Many Flavors of Gradient Descent




SGD
RMSPROP
ADAM
ADAGRAD
ADADELTA
3
2
1
0
−1
−2
−3
−1
0
1
2
3
loss
5
4
3
2
1
0
0
20
40
60
80
100
num. iteration


SGD
RMSPROP
ADAM
ADAGRAD
ADADELTA
3
2
1
0
−1
−2
−3
−1
0
1
2
3
loss
5
4
3
2
1
0
0
20
40
60
80
100
num. iteration

Adam (Adaptive Moment Estimation)

- Adaptive learning rate for each parameter
- Keeps exponentially decaying average of past gradients
- Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface

Adam (Adaptive Moment Estimation)

Algorithm [Simplified Version]:

Step 1. $m^{(t+1)} = \beta_1 m^{(t)} + (1 - \beta_1) \cdot g^t$

m is the exponential moving average of the gradients (similar as the speed in the momentum method);

Step 2. $v^{(t+1)} = \beta_2 v^{(t)} + (1 - \beta_2) \cdot (g^t)^2$

v is the exponential moving average of the square gradients.

Step 3. $w^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{v^{(t+1)}} + 10^{-8}} \cdot m^{(t+1)}$

Default settings: $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \eta = 0.001.$

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (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 δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

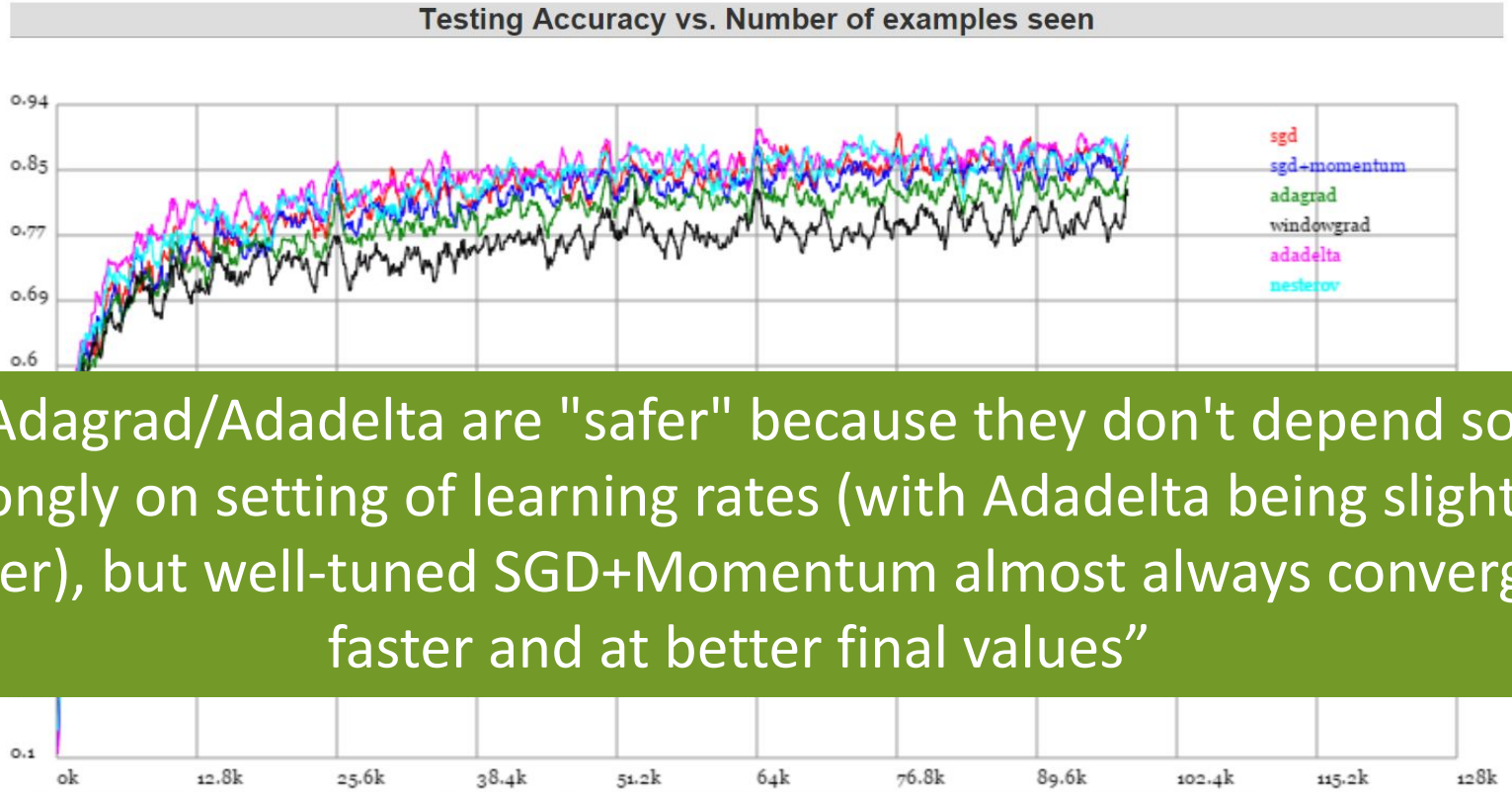
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

CNN training with different optimization algorithms



"Adagrad/Adadelta are "safer" because they don't depend so strongly on setting of learning rates (with Adadelta being slightly better), but well-tuned SGD+Momentum almost always converges faster and at better final values"

Karpathy, MNIST demo: http://cs.stanford.edu/people/karpathy/convnetjs/demo/trainers.html

Comparing GD Methods

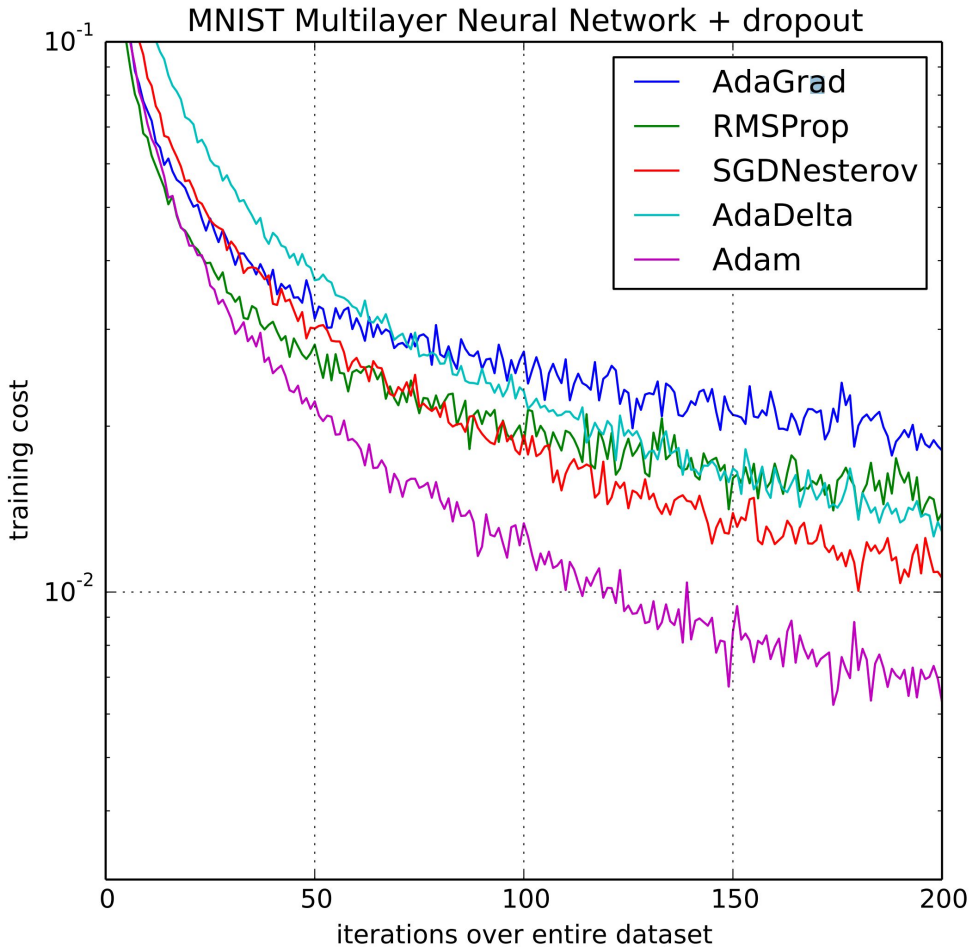


Image credit: Kingma & Ba, 2015

Network configurations:

Input layer: 28x28

2 hidden layers (1000 ReLUs each)

Output: 10 SoftMax

Cross-entropy loss.

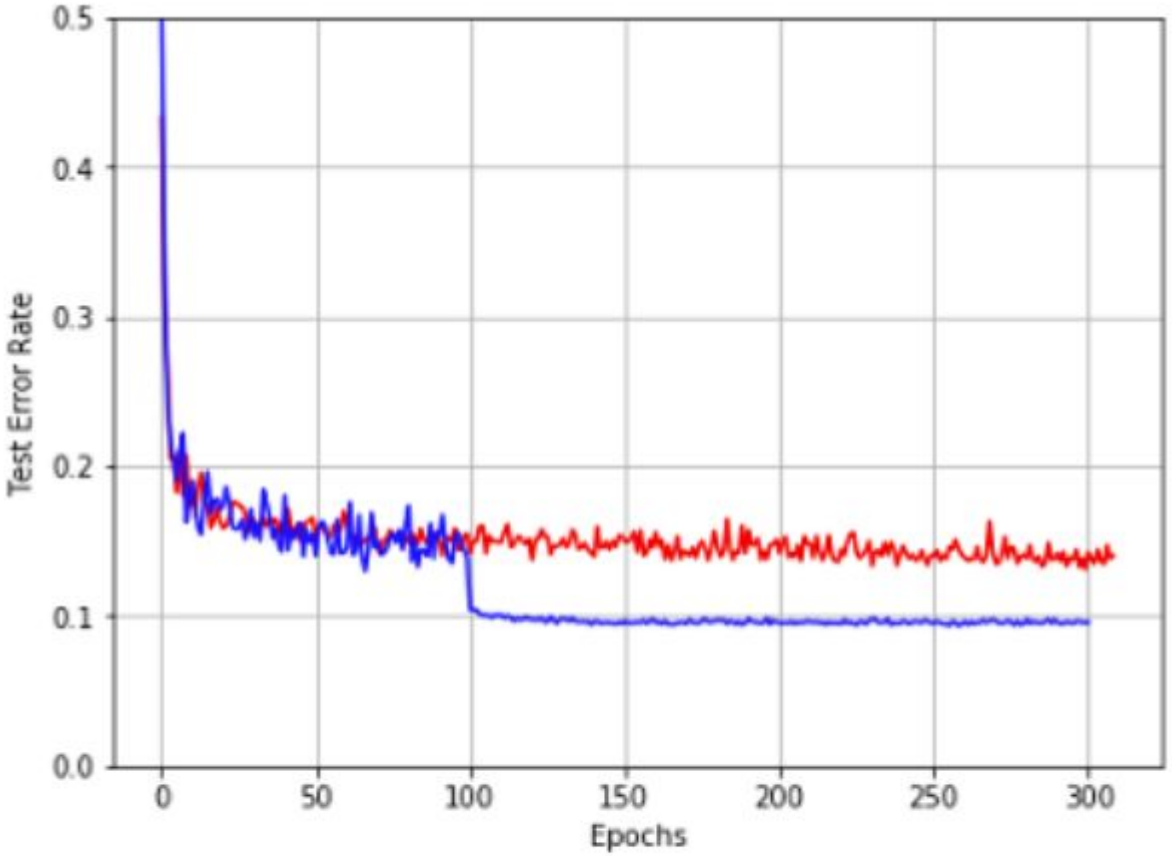
In practice, Adam is the default choice to use

[Fei-Fei Li, Andrej Karpathy, Justin Johnson]

No Free Lunch

- RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the nominator update rule. Adam, finally, adds bias-correction and momentum to RMSprop.
- RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances
- Adam has become the default algorithm used
- But there are cases in which GD and SGD provide much better results:
  - The number of parameters exceeds the number of data points
  - The rate decay of SGD is well-tuned
- Some modern methods combine Adam for initial stages with SGD for later ones

Wilson et al., The Marginal Value of Adaptive Gradient Methods in Machine Learning.



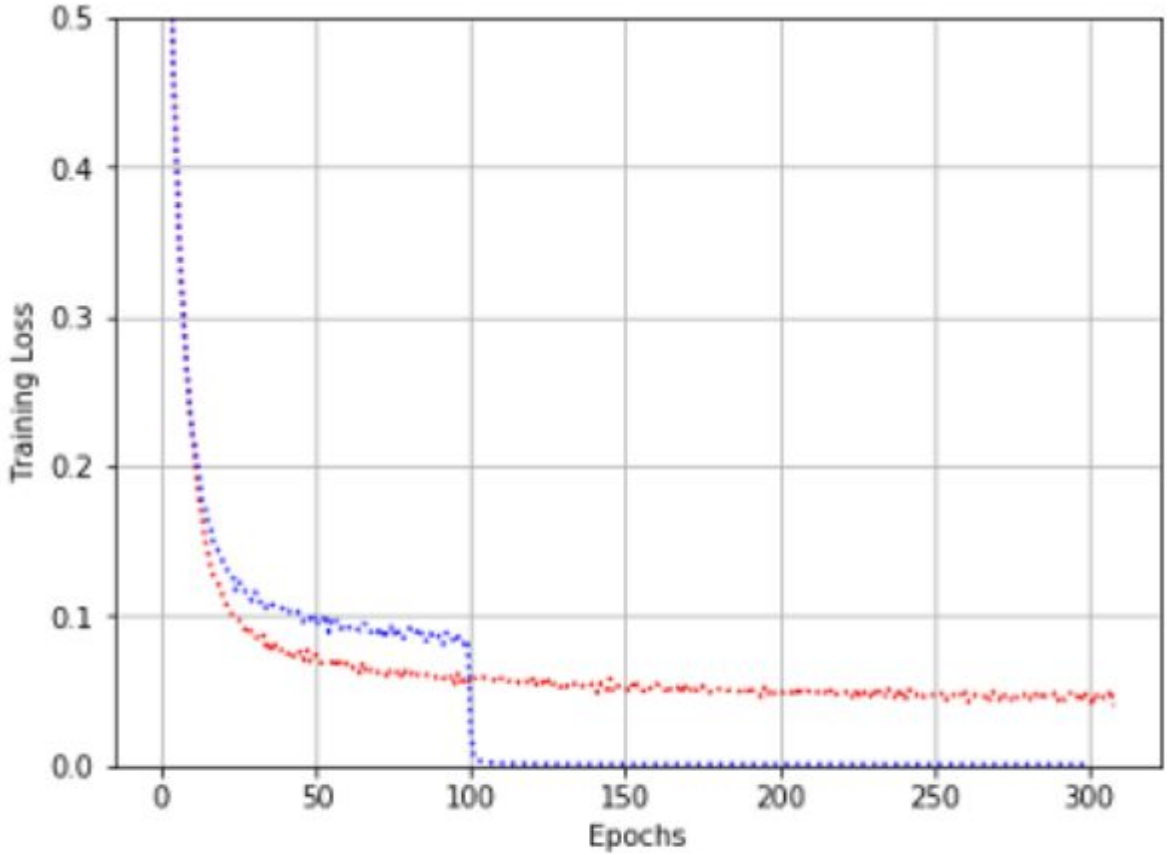


Fig. 5: Comparison of the results obtained by tuning and not tuning the learning rate. **Red lines**: the results without tuning; **Blue lines**: the results with tuning.

AdamW

- Modifies the typical implementation of weight decay in Adam, by decoupling weight decay from the gradient update

Algorithm 2 Adam with L_2 regularization and Adam with weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w\boldsymbol{x}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{x}_t$

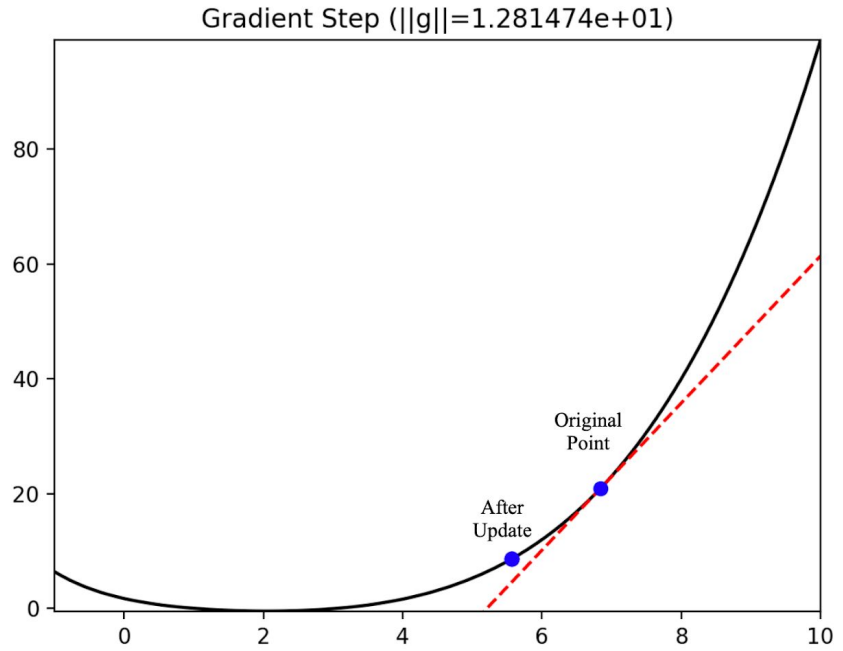
Taken from "Fixing Weight Decay Regularization in Adam" by Ilya Loshchilov, Frank Hutter.

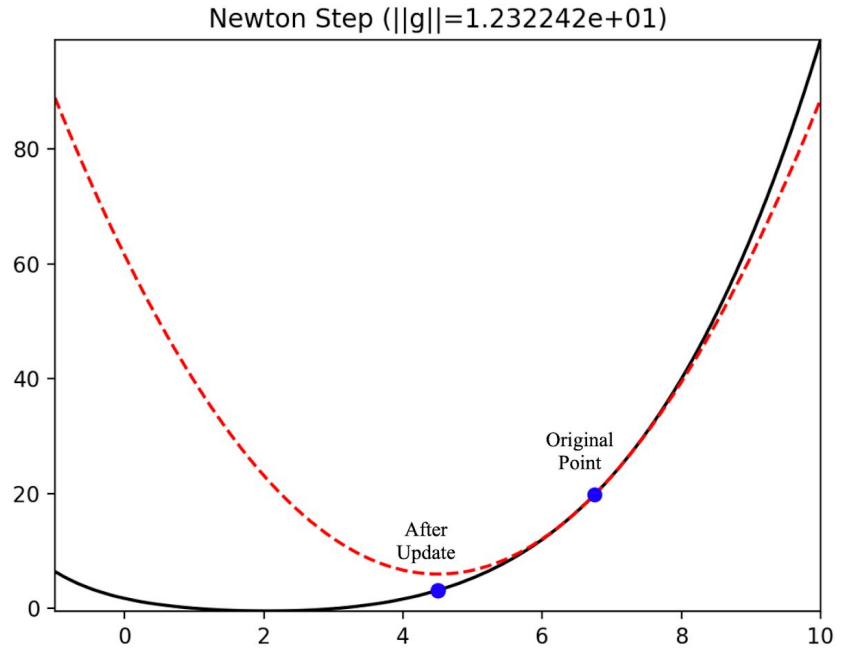
Alternatives to Gradient Descent

- The gradient provides the slope of the function in each dimension
- More information is included in the Hessian – the square matrix of second order derivatives
- This is called *Newton's method*

$$x \leftarrow x - [Hf(x)]^{-1} \nabla f(x)$$

- The Hessian describes the local curvature of the loss function, which allows us to perform a more efficient update
- Leads to more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature
- Note that there is no learning rate hyperparameter(s) – an advantage





Using the Hessian – Newton's Method

- Computing the Hessian requires computing many parameters – assuming 1M parameters, the Hessian has 1Mx1M values
- Current research focuses on approximating the inverse Hessian, preferably on mini-batches and not the entire dataset
- E.g. L-BFGS uses historical gradient information to approximate the Hessian
- In practice, SGD+Momentum methods are much more common for optimizing neural networks
- The good news – you generally don't have to worry about any of that

What it means for you?

- A good optimizer can help you squeeze more juice from the lemon
- But too often people start optimizing too early
- Hyper-parameter optimization should be among the last things you do
  - Explore the data and errors
  - Consider getting more data
  - Evaluate your loss function
  - …

Neural Networks in PyTorch

- We define the loss function and the optimizer:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

- PyTorch Optimizers

- `public torch::optim::Adagrad` (Class Adagrad)
- `public torch::optim::Adam` (Class Adam)
- `public torch::optim::AdamW` (Class AdamW)
- `public torch::optim::LBFGS` (Class LBFGS)
- `public torch::optim::RMSprop` (Class RMSprop)
- `public torch::optim::SGD` (Class SGD)

Training the Network is done in a loop

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Per-layer Optimizer Params

```
optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

This means that `model.base`'s parameters will use the default learning rate of `1e-2`, `model.classifier`'s parameters will use a learning rate of `1e-3`, and a momentum of `0.9` will be used for all parameters.

Suggested Reading

- Chapter 8 of the book “Deep Learning” – https://www.deeplearningbook.org/contents/optimization.html

Recommeded Resources

- https://hackernoon.com/some-state-of-the-art-optimizers-in-neural-networks-a3c2ba5a5643
- https://ruder.io/optimizing-gradient-descent/
- AdaBound – https://medium.com/syncedreview/iclr-2019-fast-as-adam-good-as-sgd-new-optimizer-has-both-78e37e8f9a34