CS6777: Gradient Descent and its Variations

Outline

- **Gradient Descent**
- **Momentum Based Gradient Descent**
- **Nesterov Accelerated Gradient Descent**
- **AdaGrad**
- **RMSProp**
- **Adam**

A typical Machine Learning Setup

This brings us to a typical machine learning setup which has the following components…

- **Data**: $\{x_i, y_i\}_{i=1}^{n}$
- **Model**: Our approximation of the relation between x and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$

$$or \quad \hat{y} = \mathbf{w}^T \mathbf{x}$$

- **Parameters**: In all the above cases, **w** is a parameter which needs to be learned from the data
- **Learning algorithm: An algorithm** for learning the parameters **(w)** of the model (for example, **perceptron learning algorithm, gradient descent,** etc.)
- **Objective/Loss/Error function:** To guide the learning algorithm - the learning algorithm should aim to minimize the loss function

For Example:

As an illustration, consider our movie example

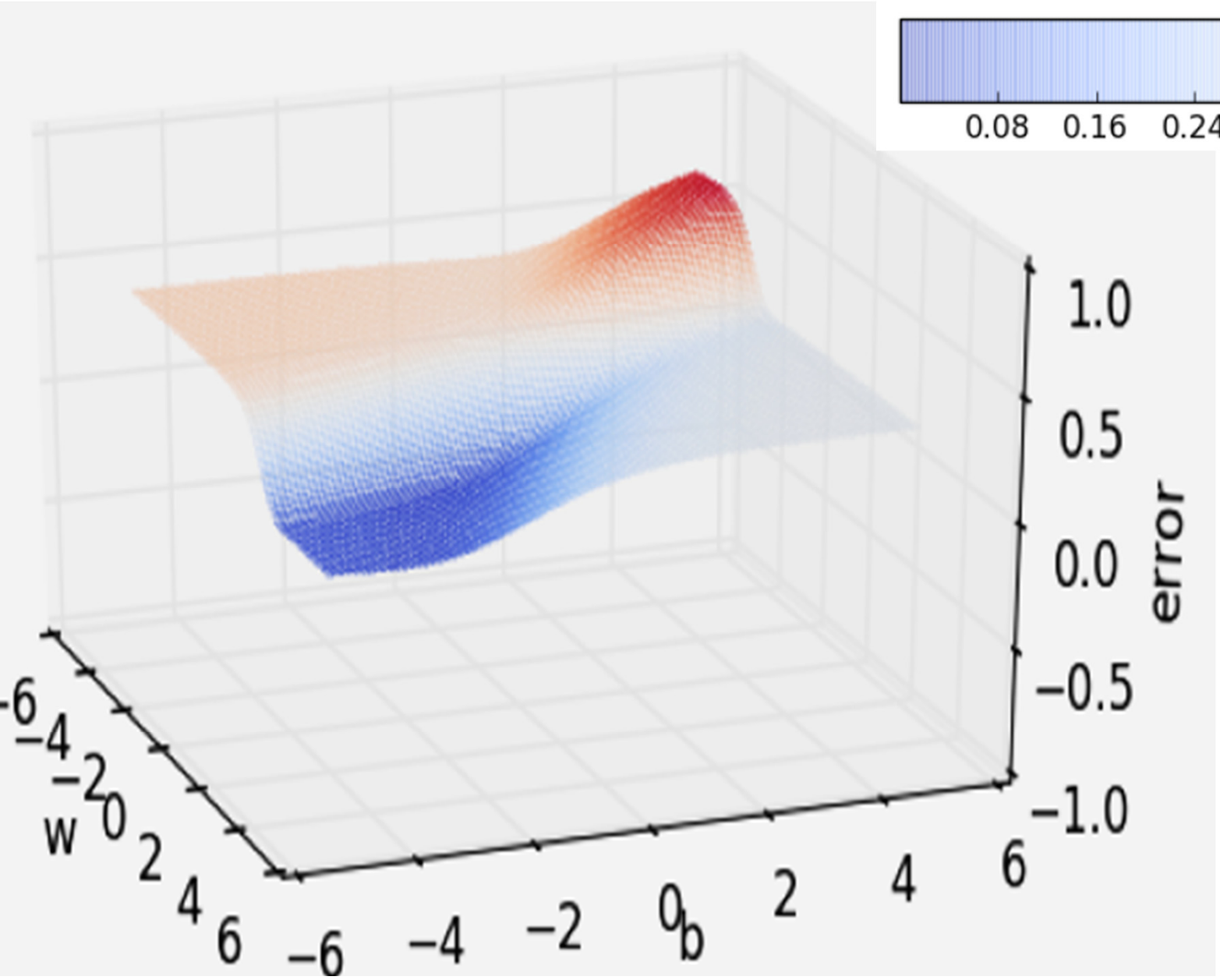
- **Data**: $\{x_i = movie, y_i = like/dislike\}_{i=1}^{n}$
- **Model**: Our approximation of the relation between x and y (the probability of liking a movie).

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$

- **Parameter**: w
- **Learning algorithm**: Gradient Descent
- **Objective/Loss/Error function:** One possibility is

$$\mathscr{L}(\mathbf{w}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

The learning algorithm should aim to find a w which minimizes the above function (squared error between **y** and $\hat{\mathbf{y}}$)



Error Surface

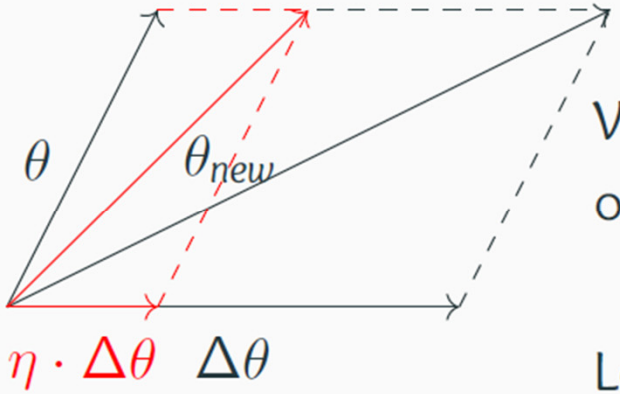
Gradient Descent

vector of parameters, say, randomly initialized

$\theta = [w, b]$

2. $\Delta\theta = [\Delta w, \Delta b]$

change in the values of w, b



We moved in the direction of $\Delta\theta$

Let us be a bit conservative: move only by a small amount η

$$\theta_{new} = \theta + \eta \cdot \Delta\theta$$

Question: What is the right $\Delta\theta$ to use ?

The answer comes from Taylor series

Gradient Descent Continued…

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$\mathscr{L}(\theta + \eta u) = \mathscr{L}(\theta) + \eta * u^T \nabla_\theta \mathscr{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 \mathscr{L}(\theta) u + \frac{\eta^3}{3!} * \ldots + \frac{\eta^4}{4!} * \ldots$$
$$= \mathscr{L}(\theta) + \eta * u^T \nabla_\theta \mathscr{L}(\theta) \text{ [} \eta \text{ is typically small, so } \eta^2, \eta^3, .. \to 0\text{]}$$

Note that the move (ηu) would be favorable only if,

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) < 0 \text{ [i.e., if the new loss is less than the previous loss]}$$

This implies,

$$u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

Gradient Descent Continued…

what is the range of $u^T \nabla_\theta \mathscr{L}(\theta)$?

Let β be the angle between u and $\nabla_\theta \mathscr{L}(\theta)$, then we know that,

$$-1 \leq cos(\beta) = \frac{u^T \nabla_\theta \mathscr{L}(\theta)}{||u|| * ||\nabla_\theta \mathscr{L}(\theta)||} \leq 1$$

multiply throughout by $k = ||u|| * ||\nabla_\theta \mathscr{L}(\theta)||$

$$-k \leq k * cos(\beta) = u^T \nabla_\theta \mathscr{L}(\theta) \leq k$$

Thus, $\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) = u^T \nabla_\theta \mathscr{L}(\theta) = k * cos(\beta)$ will be most negative when $cos(\beta) = -1$ *i.e.*, when β is 180°

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

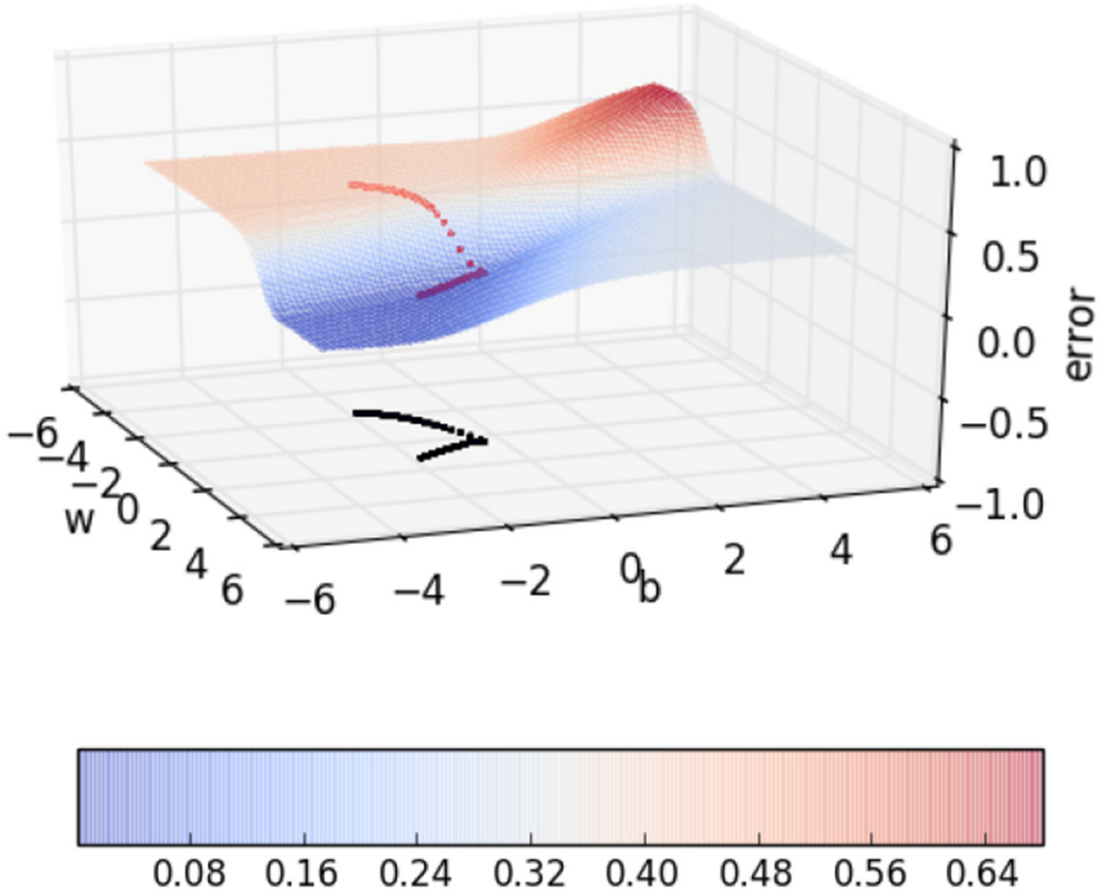
Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$

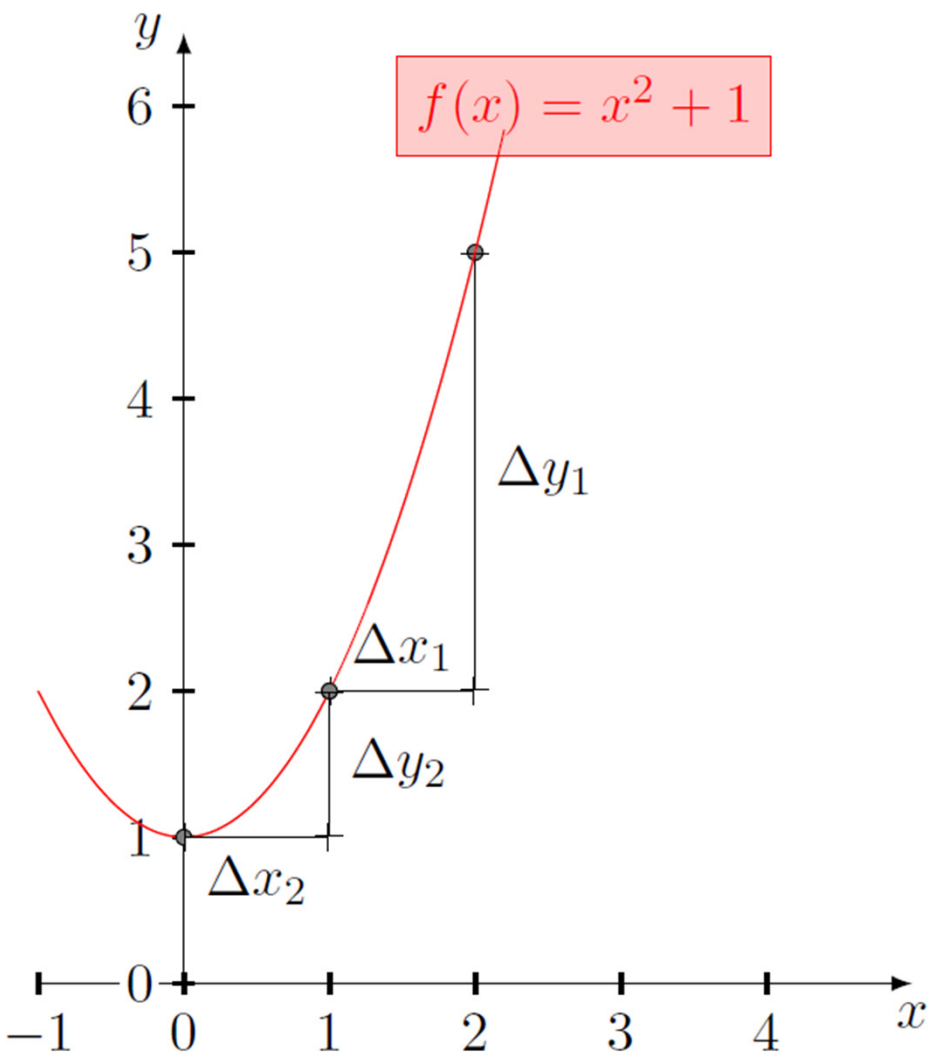
$$b_{t+1} = b_t - \eta \nabla b_t$$

$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\, w = w_t, b = b_t}, \nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\, w = w_t, b = b_t}$$

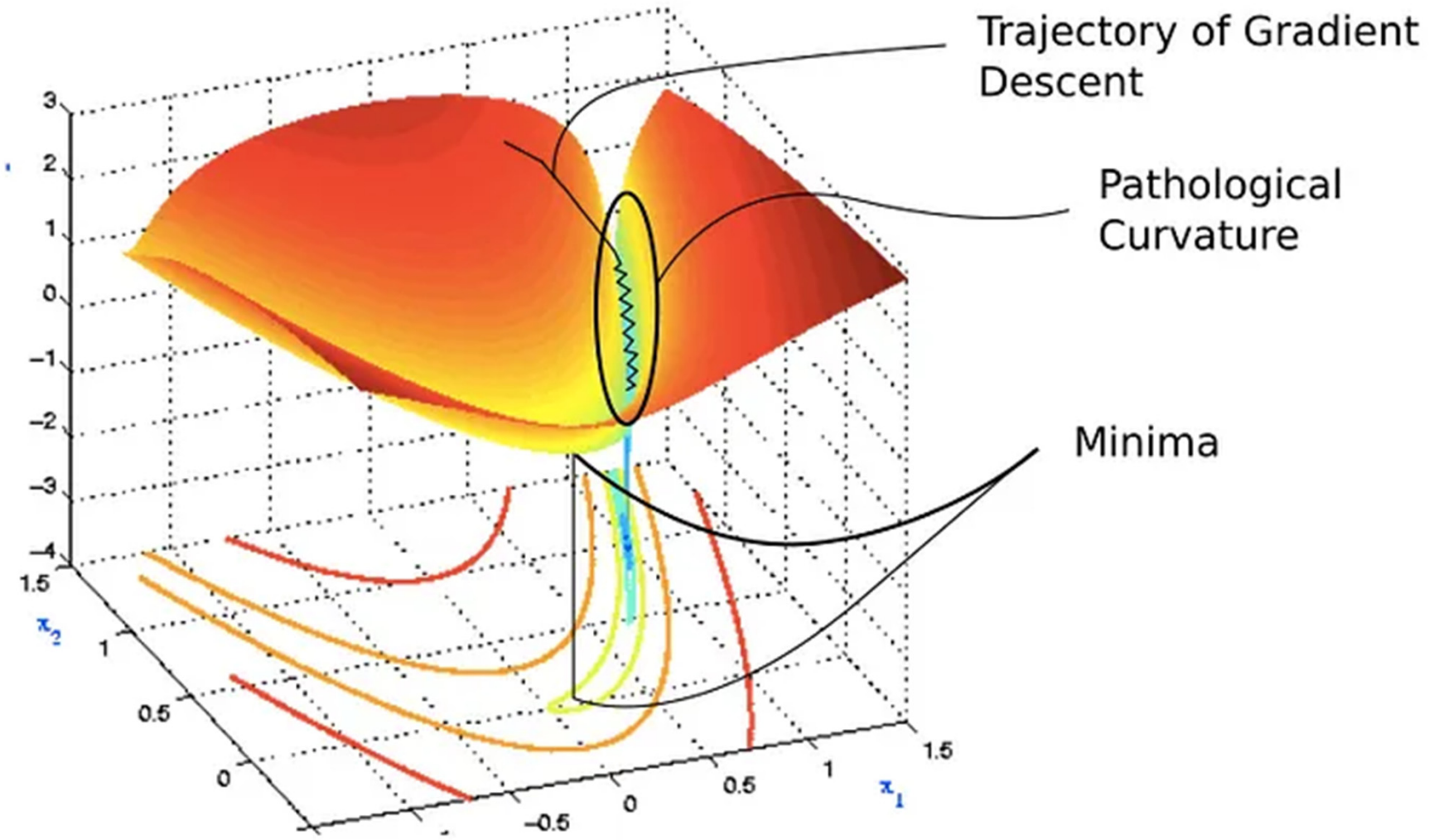
Gradient descent on the error surface



Limitations of Gradient Descent



- When the curve is steep the gradient $(\frac{\Delta y_1}{\Delta x_1})$ is large
- When the curve is gentle the gradient $(\frac{\Delta y_2}{\Delta x_2})$ is small
- Recall that our weight updates are proportional to the gradient $w = w - \eta \nabla w$
- Hence in the areas where the curve is gentle the updates are small whereas in the areas where the curve is steep the updates are large


Trajectory of Gradient Descent
Pathological Curvature
Minima
3
2
1
0
−1
−2
−3
−4
1.5
x_2
1
0.5
0
−0.5
0
0.5
1
1.5
x_1

Some observations about gradient descent

- It takes a lot of time to navigate regions having a gentle slope
- This is because the gradient in these regions is very small
- Can we do something better ?
- Yes, let's take a look at 'Momentum based gradient descent'

Momentum Based Gradient Descent

Update rule for momentum based gradient descent

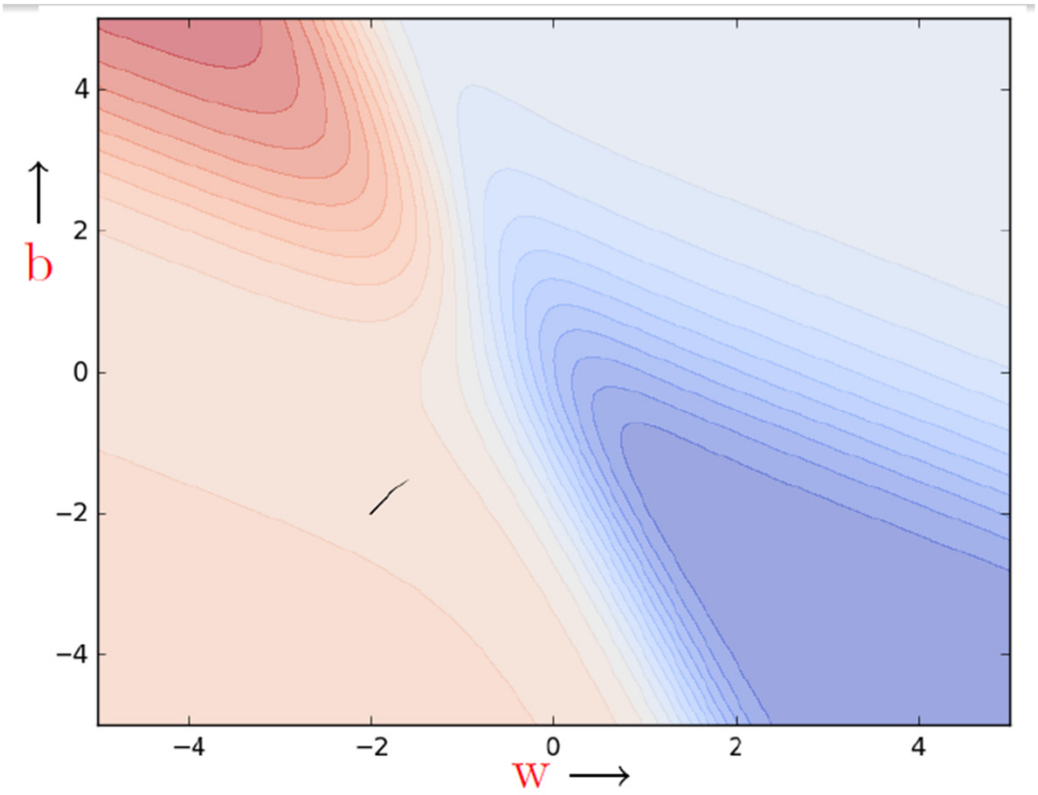
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

- In addition to the current update, also look at the history of updates.

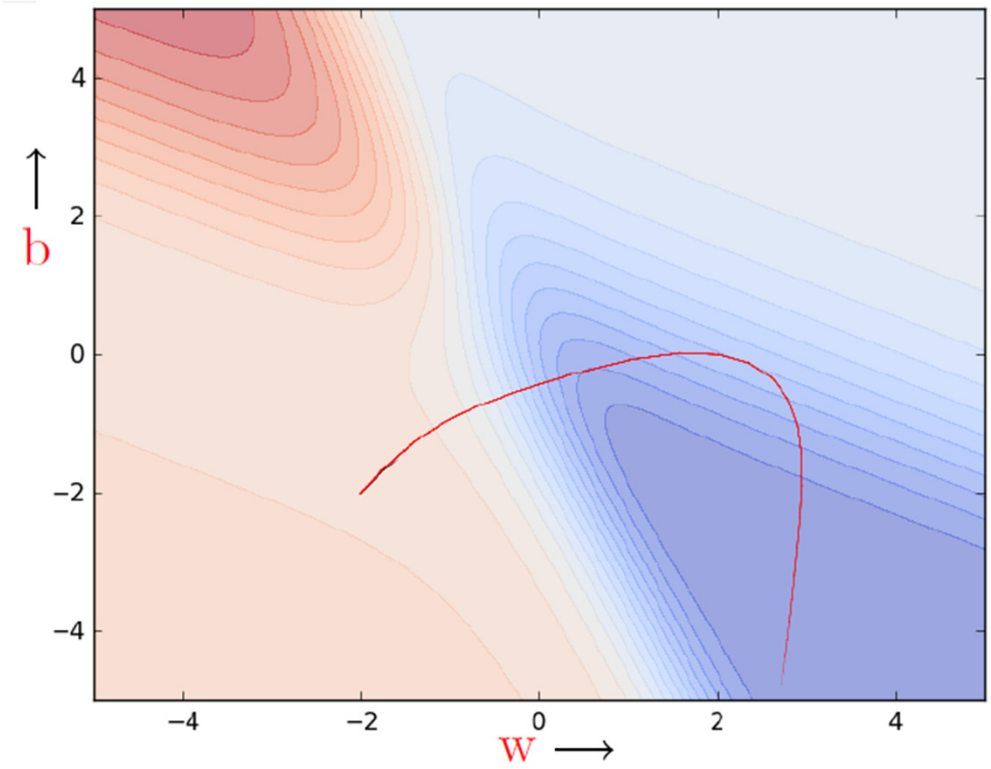
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

$$\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot update_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + ... + \eta \nabla w_t
\end{aligned}$$

Comparing convergence of GD and Momentum based GD



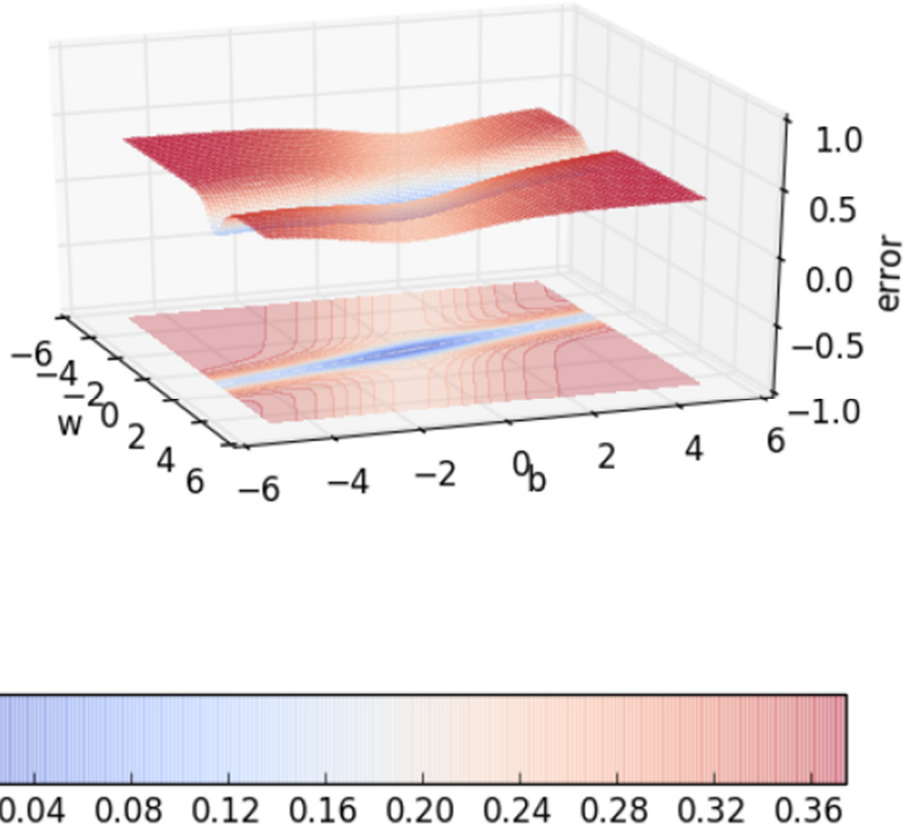
Gradient Descent

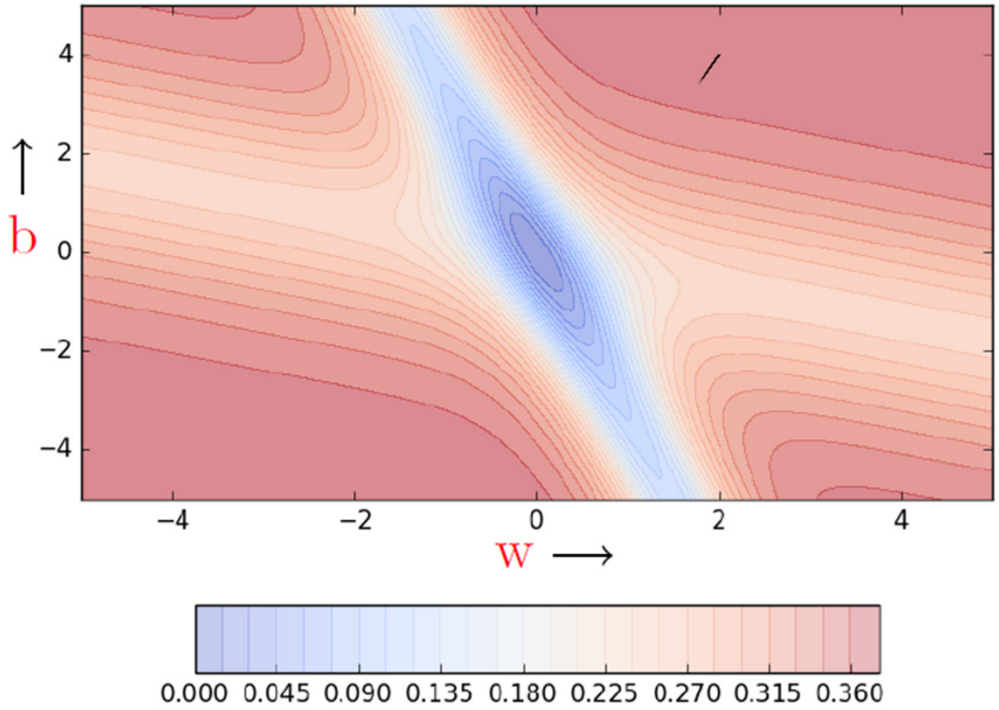


Momentum Based Gradient Descent

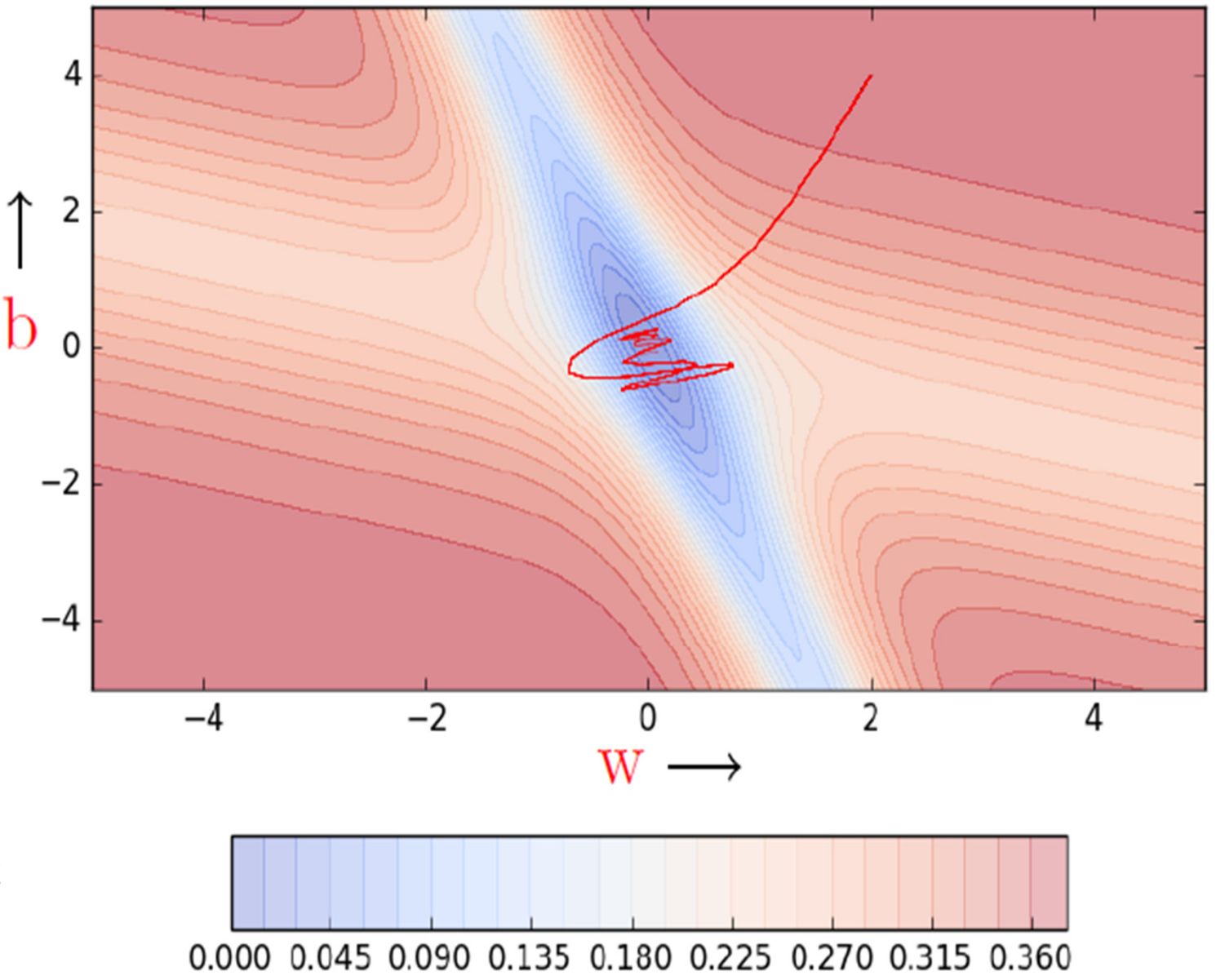
Observations

- Is moving fast always Good?
- Can we overshoot our target?





Observations



- Takes lots of U-turns/oscillations before converging.
- Despite oscillations, it is faster than gradient descent.
- Can we do something about it?

Nesterov Gradient Descent

Intuition

- Look before you leap
- Recall that $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ and then a bit more by $\eta \nabla w_t$
- Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w ($w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$) instead of calculating it using the current value w_t

Update rule for NAG

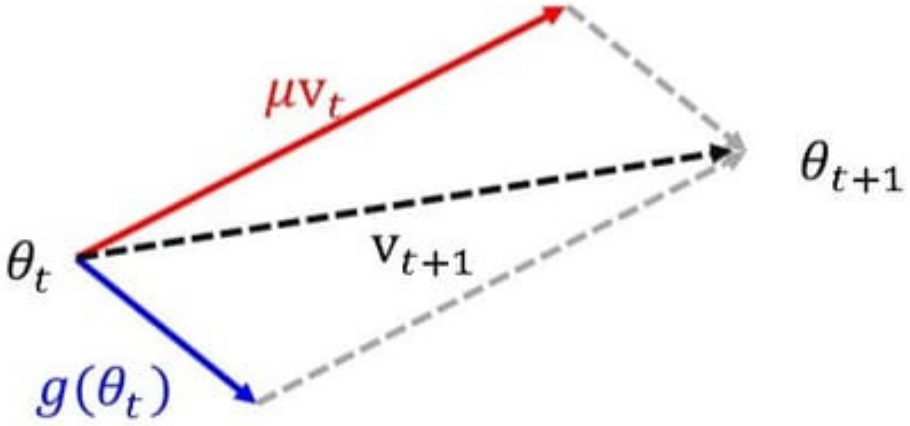
$$
\begin{aligned}
w_{look\_ahead} &= w_t - \gamma \cdot update_{t-1} \\
update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead} \\
w_{t+1} &= w_t - update_t
\end{aligned}
$$

We will have similar update rule for b_t

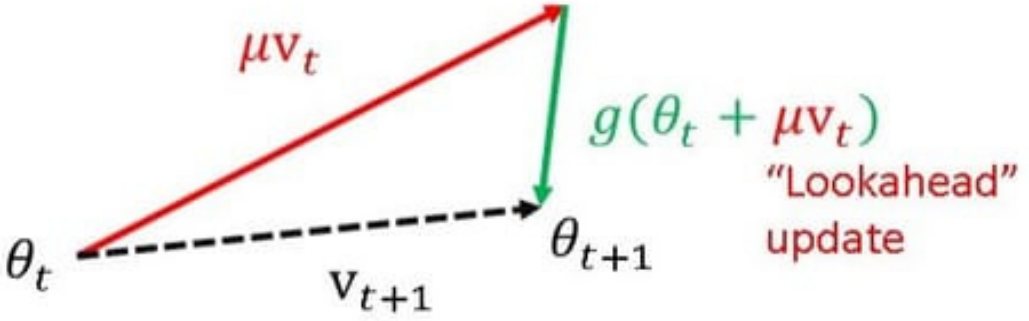
$$v_{t+1} = \mu v_t - \eta \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Why NAG works?

Momentum



NAG

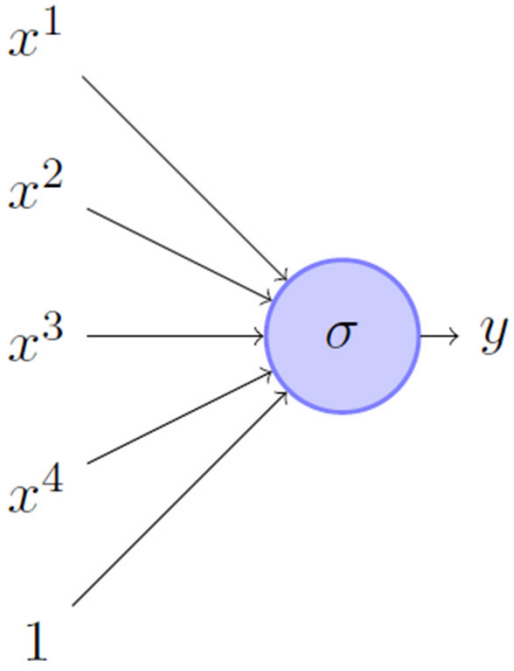


$$v_t = (1-\beta)v_{t-1} + \beta g$$

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

Adaptive Learning Rate Based Methods

- What if instead of applying this momentum, we would have used a higher value of learning rate like 10?
- It will blow up the gradient in both gentle and steep regions
- Ideally we would like to use a large learning rate in gentle regions and small learning rate in steep regions.



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

- The gradients would be as follows:
- $\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$
- $\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$ ... so on

- What happens if the feature x^2 is very sparse?
- ∇w^2 will be 0 for most inputs (see formula) and hence w^2 will not get enough updates
- Can we have a different learning rate for each parameter which takes care of the frequency of features ?

AdaGrad (Adaptive Gradient Algorithm)

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

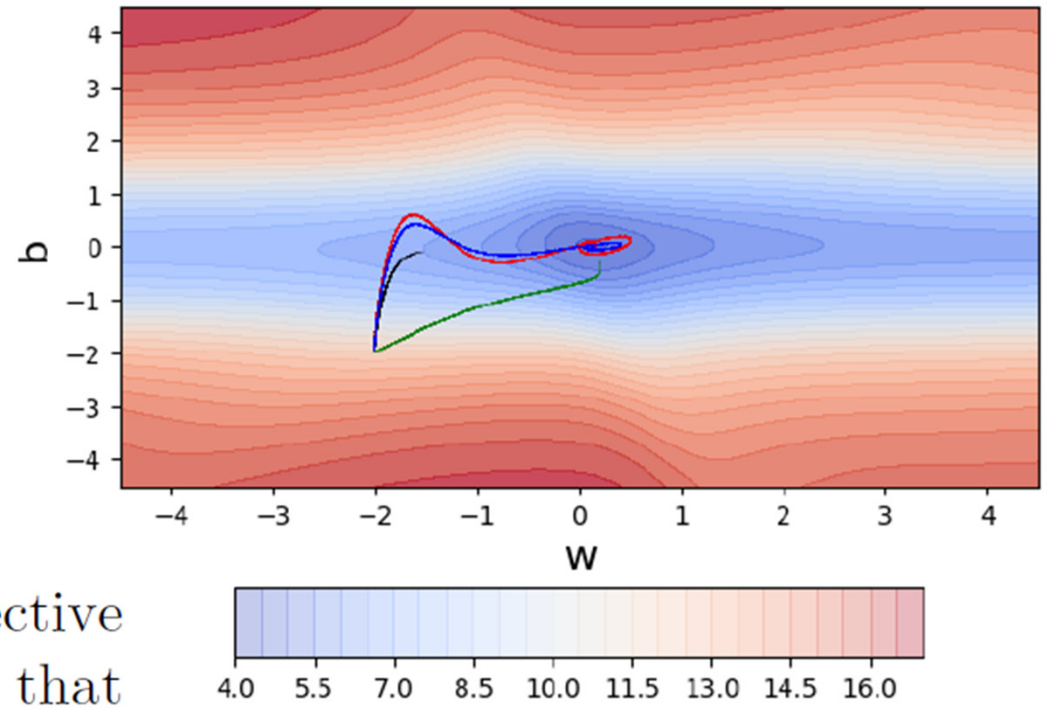
$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

Observations

- Initially, all three algorithms are moving mainly along the vertical (b) axis and there is very little movement along the horizontal (w) axis
- Why? Because in our data, the feature corresponding to w is sparse and hence w undergoes very few updates ...on the other hand b is very dense and undergoes many updates

GD (black), momentum (red) and NAG (blue)



- What's the flipside? over time the effective learning rate for b will decay to an extent that there will be no further updates to b

RMSProp (Root Mean Squared Propagation)

Intuition

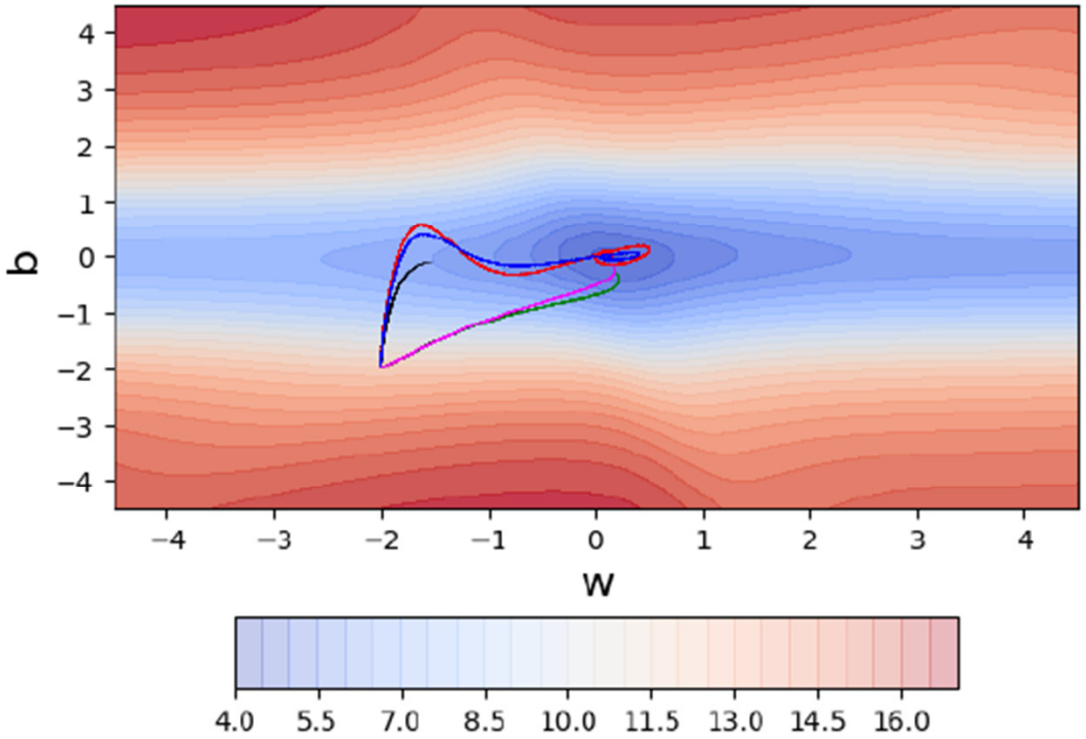
- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate
- To avoid this why not decay the denominator and prevent its rapid growth

Update rule for RMSProp

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

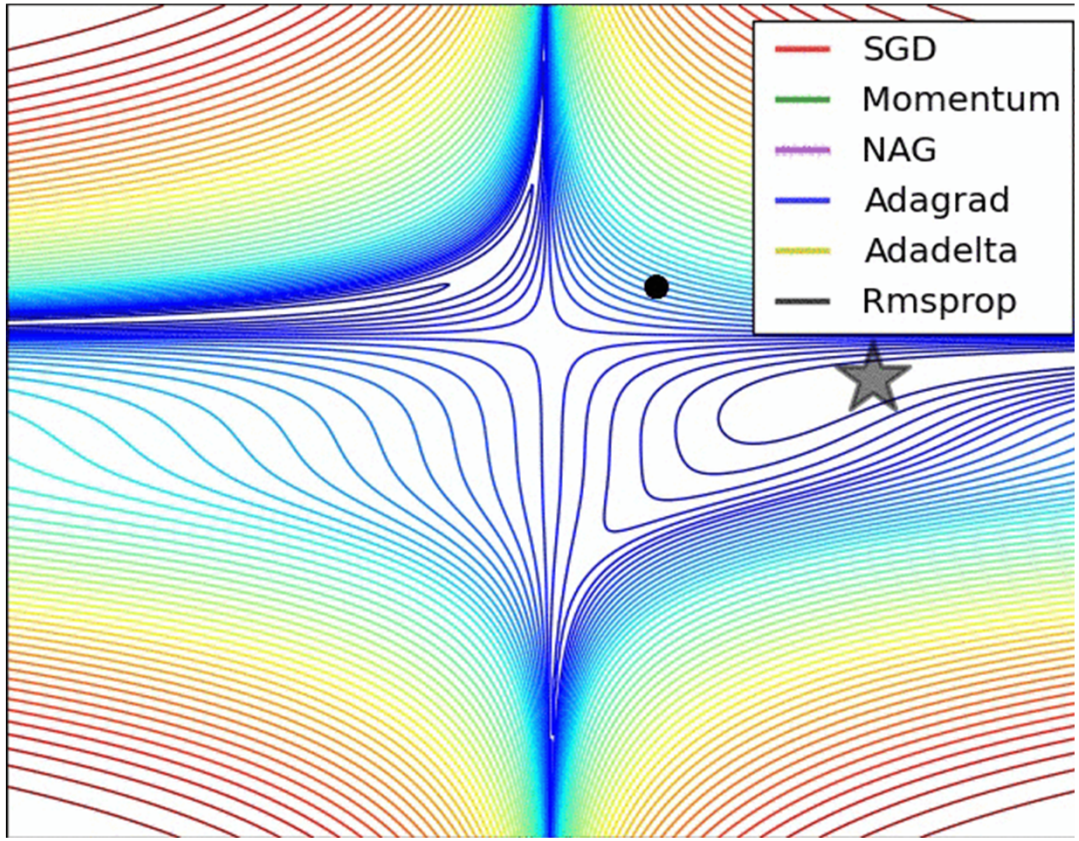
... and a similar set of equations for b_t

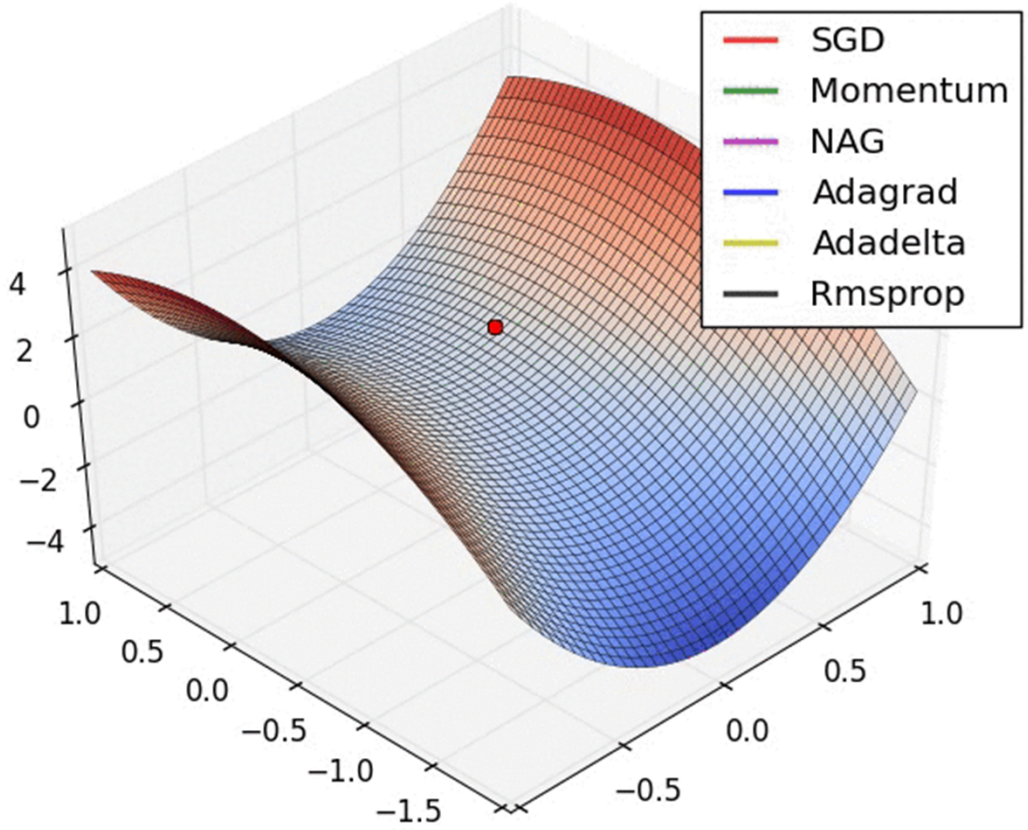


- Green: AdaGrad, Pink: RMSProp

GD (black), momentum (red) and NAG (blue)

- RMSProp overcomes this problem by being less aggressive on the decay


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
4
2
0
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
0.5
0.0
−0.5

Adam (Adaptive Moment Estimation)

Intuition

- Do everything that RMSProp does to solve the decay problem of Adagrad
- Plus use a cumulative history of the gradients
- In practice, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

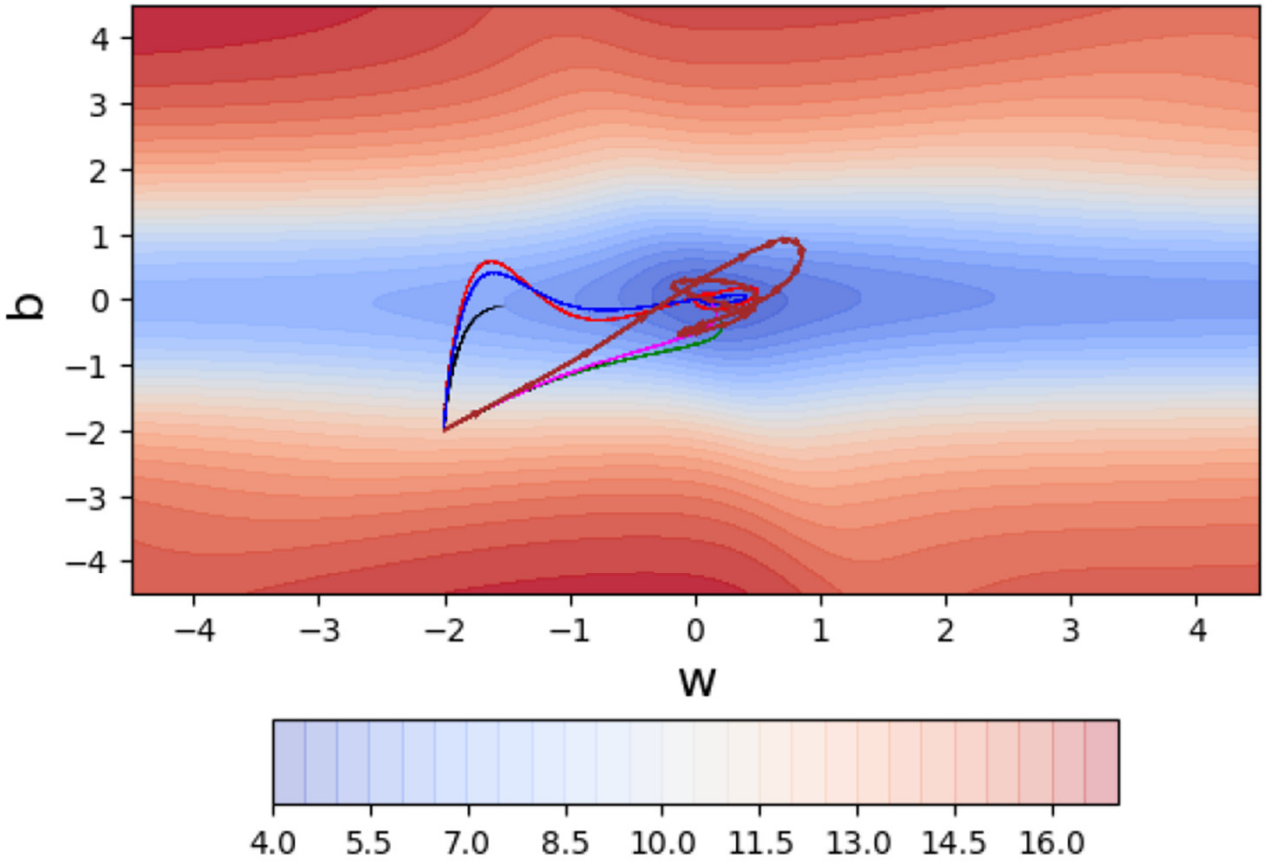
Update rule for Adam

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$



Green: AdaGrad, Pink: RMSProp

GD (black), momentum (red) and NAG (blue)

Update equations

Method	Update equation
SGD	$g_t = \nabla_{\theta_t} J(\theta_t)$ $\Delta\theta_t = -\eta \cdot g_t$ $\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Table: Update equations for the gradient descent optimization algorithms.

SGD(I_t, η_t)

$$\theta_{t+1} = \theta_t - \eta_t \nabla \ell(\theta_t)$$

MOMENTUM(I_t, η_t, γ)

$$v_0 = 0$$
$$v_{t+1} = \gamma v_t + \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}$$

NESTEROV(I_t, η_t, γ)

$$v_0 = 0$$
$$v_{t+1} = \gamma v_t + \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta_t \left(\gamma v_{t+1} + \nabla \ell(\theta_t)\right)$$

RMSPROP$(I_t, \eta_t, \gamma, \rho, \epsilon)$

$$v_0 = 1, m_0 = 0$$
$$v_{t+1} = \rho v_t + (1-\rho)\nabla \ell(\theta_t)^2$$
$$m_{t+1} = \gamma m_t + \frac{\eta_t}{\sqrt{v_{t+1} + \epsilon}} \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - m_{t+1}$$

RMSTEROV$(I_t, \eta_t, \gamma, \rho, \epsilon)$

$$v_0 = 1, m_0 = 0$$
$$v_{t+1} = \rho v_t + (1-\rho)\nabla \ell(\theta_t)^2$$
$$m_{t+1} = \gamma m_t + \frac{\eta_t}{\sqrt{v_{t+1} + \epsilon}} \nabla \ell(\theta_t)$$
$$\theta_{t+1} = \theta_t - \left[\gamma m_{t+1} + \frac{\eta_t}{\sqrt{v_{t+1} + \epsilon}} \nabla \ell(\theta_t)\right]$$

ADAM$(I_t, \alpha_t, \beta_1, \beta_2, \epsilon)$

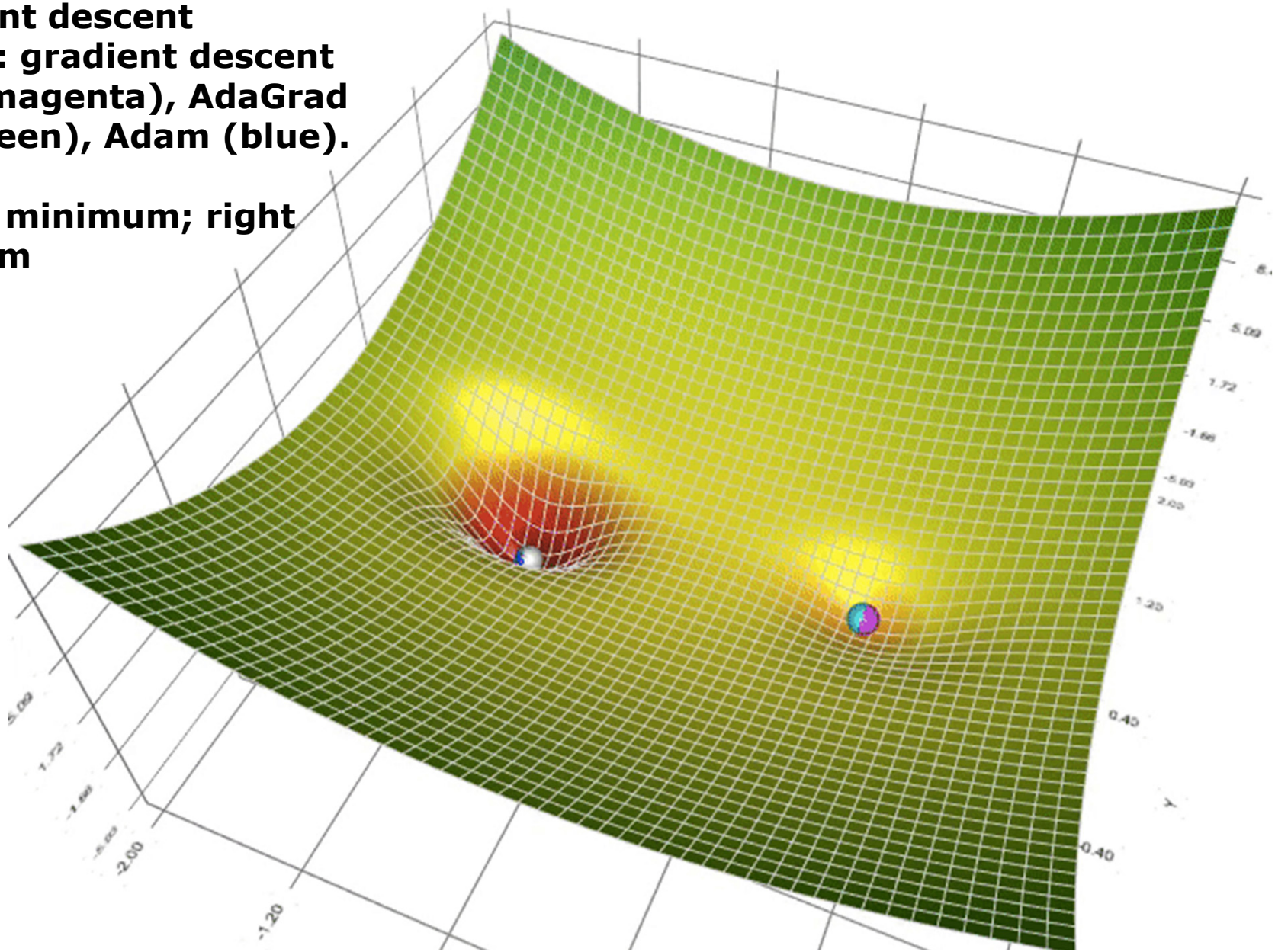
$$m_0 = 0, v_0 = 0$$
$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla \ell(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla \ell(\theta_t)^2$$
$$b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$$
$$\theta_{t+1} = \theta_t - \alpha_t \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

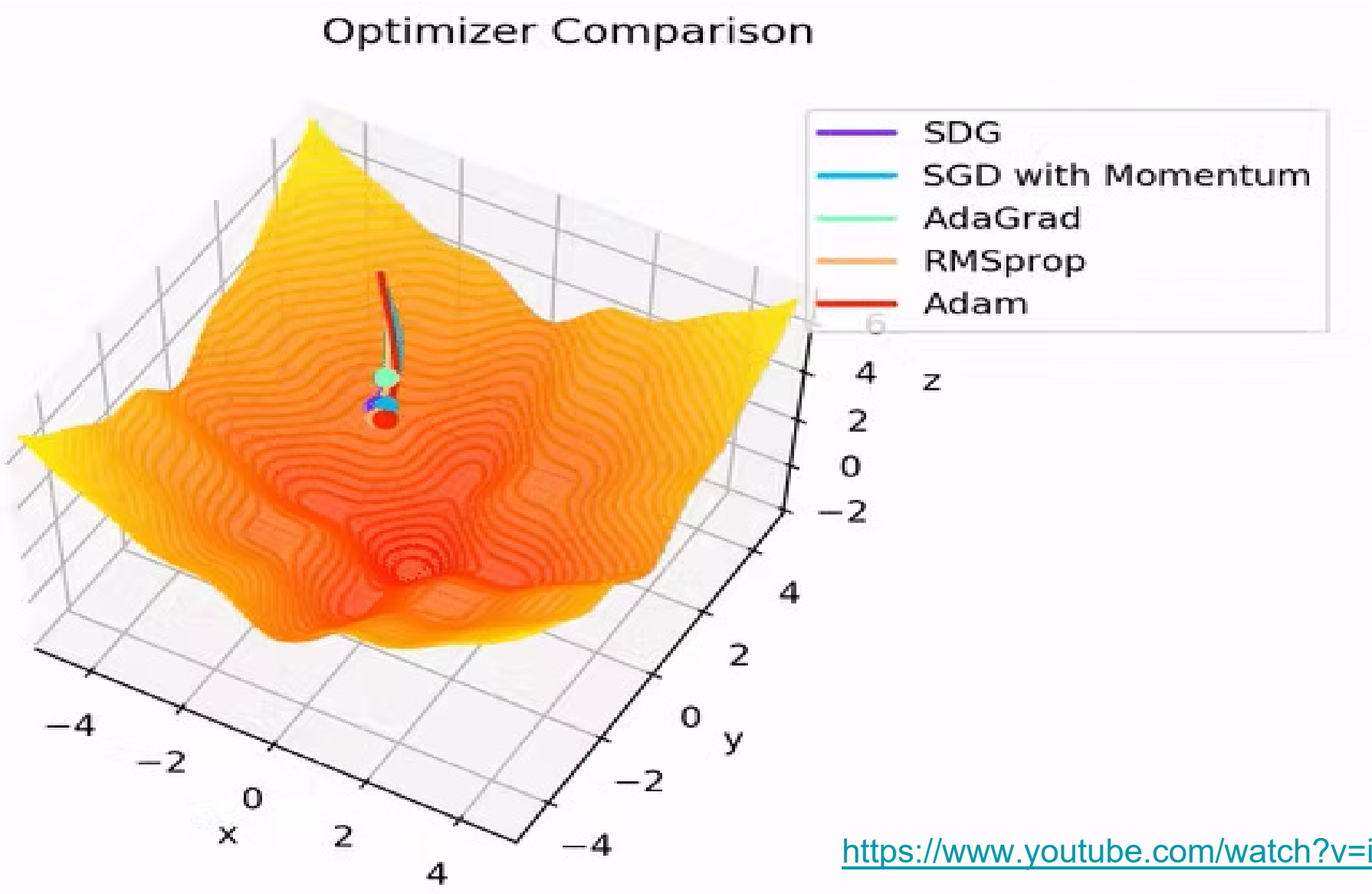
NADAM$(I_t, \alpha_t, \beta_1, \beta_2, \epsilon)$

$$m_0 = 0, v_0 = 0$$
$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla \ell(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla \ell(\theta_t)^2$$
$$b_{t+1} = \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}}$$
$$\theta_{t+1} = \theta_t - \alpha_t \frac{\beta_1 m_{t+1} + (1-\beta_1)\nabla \ell(\theta_t)}{\sqrt{v_{t+1}} + \epsilon} b_{t+1}$$

Animation of 5 gradient descent methods on a surface: gradient descent (cyan), momentum (magenta), AdaGrad (white), RMSProp (green), Adam (blue).

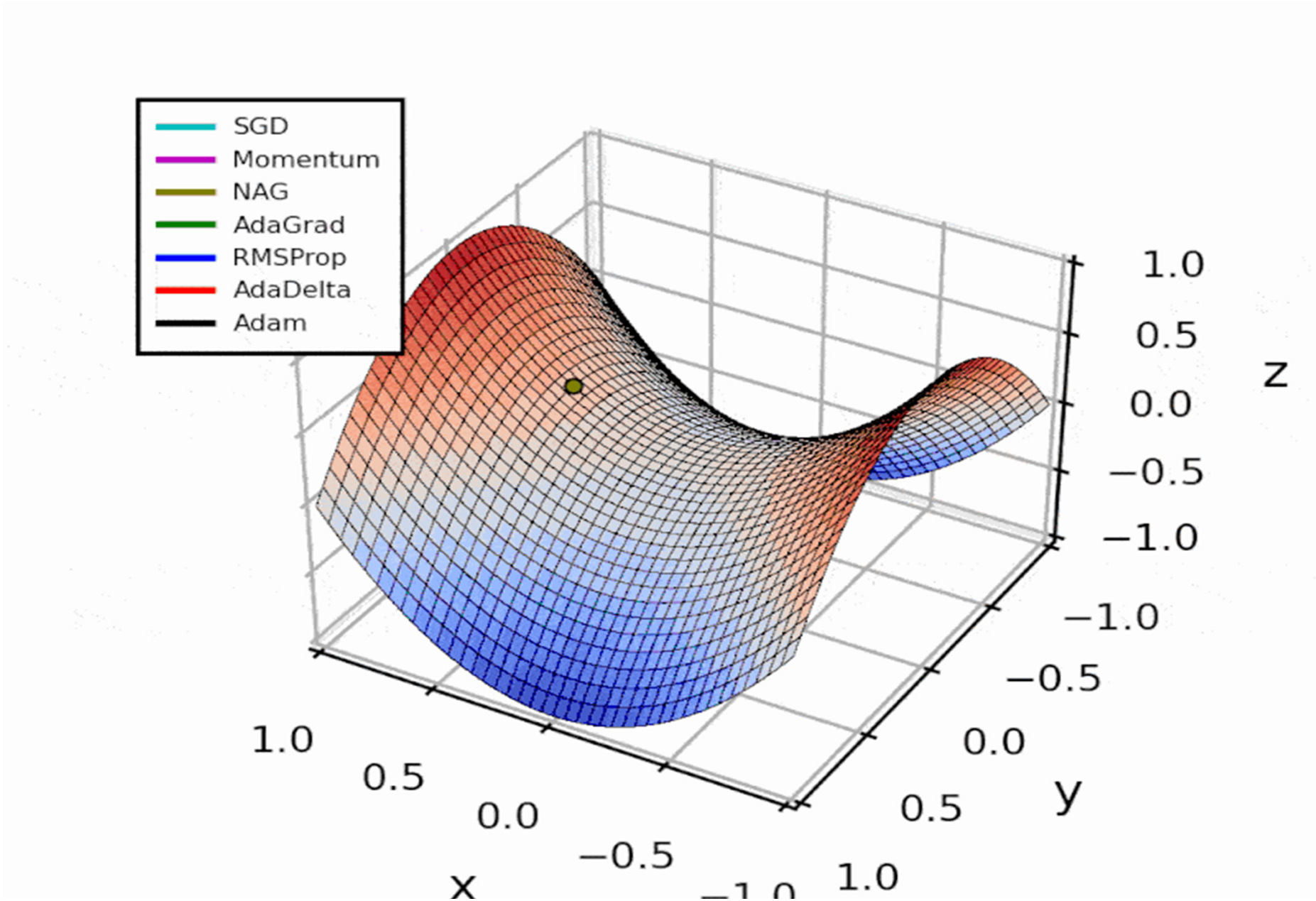
Left well is the global minimum; right well is a local minimum

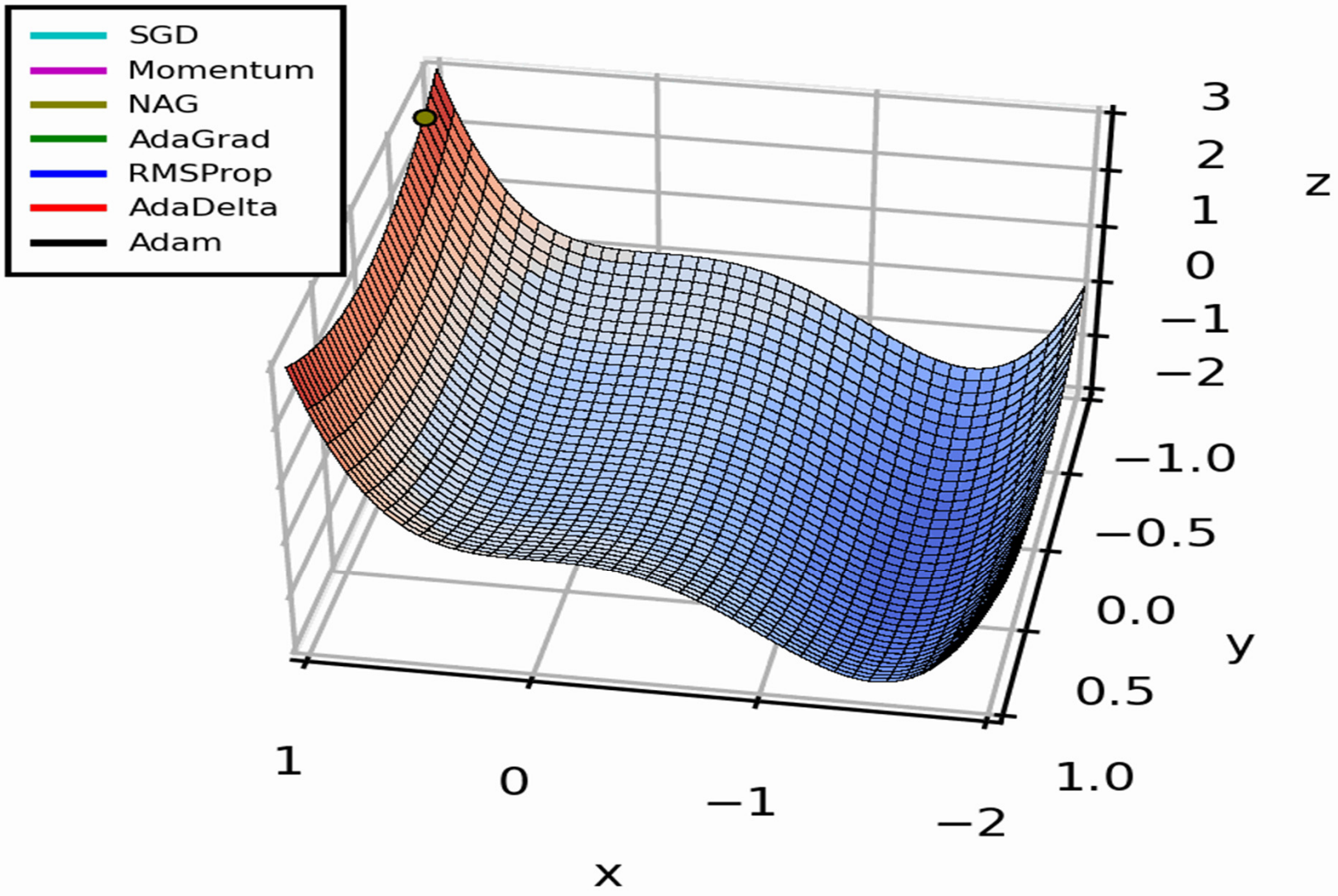


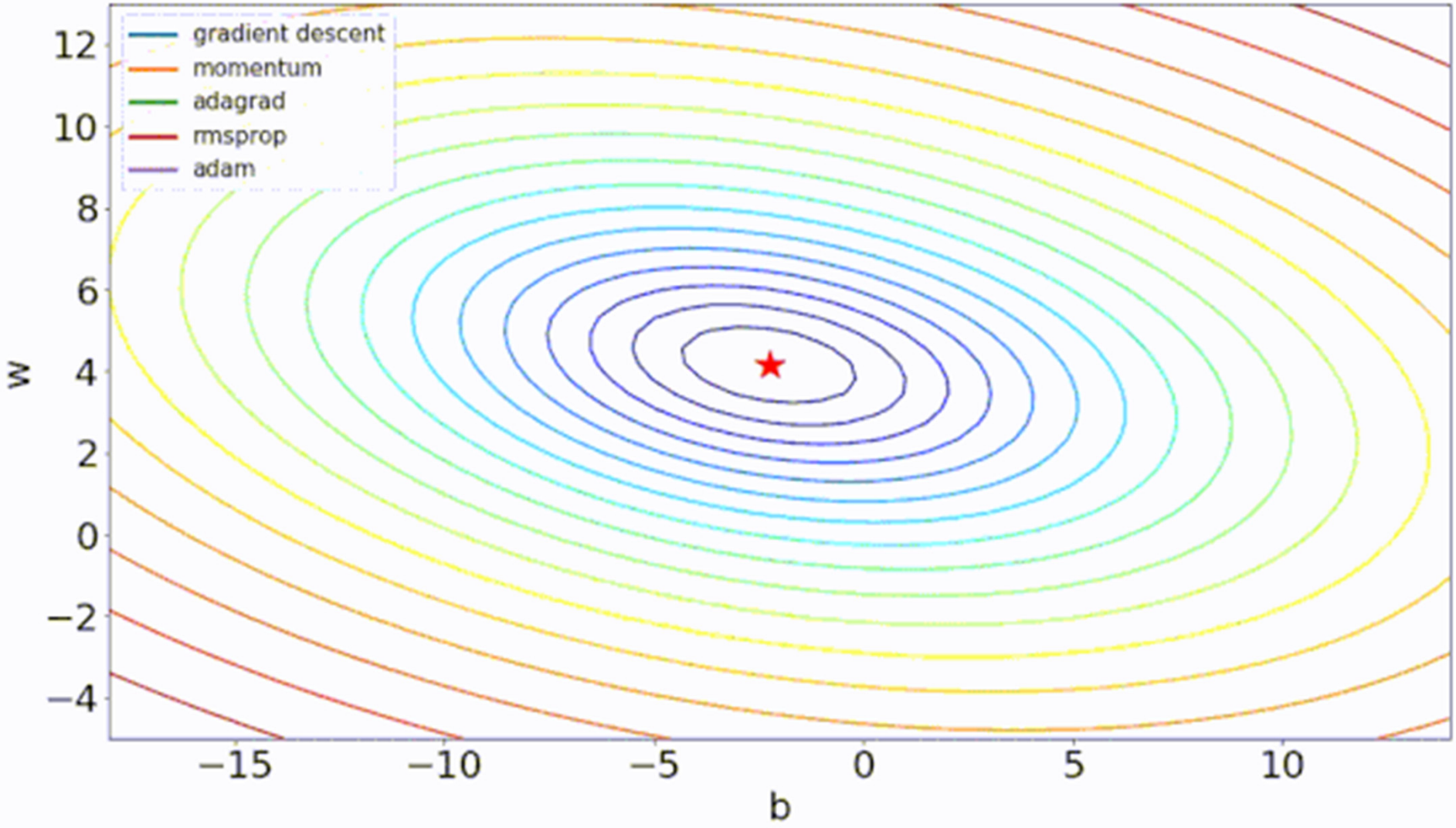


https://www.youtube.com/watch?v=ilYd4TAzNoU

https://www.youtube.com/watch?v=DwKC5S7MceU


SGD
Momentum
NAG
AdaGrad
RMSProp
AdaDelta
Adam
1.0
0.5
0.0
−0.5
−1.0
z
1.0
0.5
0.0
−0.5
−1.0
y
1.0
0.5
0.0
−0.5
−1.0
x


SGD
Momentum
NAG
AdaGrad
RMSProp
AdaDelta
Adam
3
2
1
0
−1
−2
z
−1.0
−0.5
0.0
0.5
1.0
y
1
0
−1
−2
x


gradient descent
momentum
adagrad
rmsprop
adam
12
10
8
6
4
2
0
−2
−4
w
−15
−10
−5
0
5
10
b

When training a neural network, what reasons are there for choosing an optimizer from the family consisting of stochastic gradient descent (SGD) and its extensions (RMSProp, Adam, etc.) instead of from the family of Quasi-Newton methods (including limited-memory BFGS, abbreviated as L-BFGS)?

https://en.wikipedia.org/wiki/Limited-memory_BFGS

https://www.dropbox.com/s/qavnl2hr170njbd/NumericalOptimization2ndedJNocedalSWright%282006%29.pdf?dl=0

When training a neural network the ***number of training examples is so vast compared to the number of weights*** being trained that simply evaluating the gradient is a bottleneck. **SGD methods allow you to work with much cheaper approximations of the gradient** (typically, summing the contributions of a different small subset of examples at each iteration) instead. What separates stochastic gradient descent from regular gradient descent is that, at each iteration, you randomly sample the training data and evaluate the cost/loss function (and its gradient) on just the sample, not all the data.

Can you still reuse the gradient information in a quasi-Newton method?
But that's what Adam is doing. Presumably it is using gradient information from previous iterations when the cost function is changing every iteration. You can do the same with Quasi-Newton methods ?

It's because of ***memory issues*** (e.g. LBFGS requires storing about 20-100 previous gradient evaluations) and more importantly it ***does not work in stochastic setting*** (e.g. minibatches which is very important since a full pass through a dataset is very expensive and a lot of progress can be done with small minibatches). There have been many tryouts to make LBFGS work in stochastic setting, but none that work well.

In machine learning "evaluating the gradient" means sweeping over the whole training set. For simple models, stochastic gradient descent will have found a good fit after one or two passes over the dataset. (For neural nets, tens to hundreds of passes over the dataset may be needed.) It's ***hard for batch L-BFGS to do much in the same number of gradient and function evaluations.***

Challenge is on making stochastic or mini-batch versions of algorithms like L-BFGS. It's possible, but not entirely straightforward, which is why "dumb" stochastic gradient descent is often used.

Second order methods are way more complex, i.e., harder to implement without bugs. DL systems are increasingly becoming a small part of huge data processing pipelines. Also, ***harder to optimize for distributed computing on heterogeneous hardware***, which is becoming more and more common.

Another issue with **Deep Learning optimization are saddle points**. It's becoming abundantly clear that "bad" local minima are not an issue in Deep Learning, but saddle points are. Newton's method does have a tendency to be attracted to saddle points.

Instead, if just switching to distributed computing makes the first-order method as fast as, or faster, than second-order methods, I don't see the reason to use a more complicated optimization algorithm.

2nd order methods are way more expensive in terms of iteration cost (not number) and memory occupation, thus they introduce a considerable overhead. ***Current architectures (GPUs) are more memory-bound that computation-bound. The increase in iteration cost and memory occupation is steeper, the more high-dimensional the problem is***. Optimization in Deep Learning is arguably one of the most high-dimensional optimization problems, so it's not clear that second order methods would have a clear advantage in terms of computational time (not iteration count, which is not what we really care about) wrt first-order methods.

Another issue with second order methods is that ***for most common loss functions, it's easy to use mini-batches to get an estimator which converges to the actual gradient***. It is much more complicated to build a sampling-based estimator for the approximation to the inverse of the Hessian. In other words, second order methods introduce a lot of complexity and extra memory occupation, but stochastic second order methods introduce even more complexity.

Do we need even more extra hyperparameters, or do we need robust optimization methods? Keep in mind that in Deep Learning, as explained very well by Shai Shalev-Shwartz [??], when something goes wrong, it's very difficult to understand how to fix it.

Thank You!