Lecture 10: Optimization + Convolutional neural networks

Announcements:

- HW #4 is due ~~**Friday, Feb 16**~~ Monday, Feb 19, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

- The midterm exam is during class (2-3:50pm) on Wednesday, Feb 21, 2024.

  - You are allowed 4 cheat sheets (each an 8.5 x 11 inch paper). You can fill out both sides (8 sides total). You can put whatever you want on these cheat sheets.
  - The midterm will cover material up to and including this Wednesday's lecture (Feb 14).
  - Past exams are uploaded to Bruin Learn (under "Modules" —> "past exams").
  - You may bring a calculator to the exam.
  - You may do the exam in pen or pencil.

- Midterm exam review session: Thursday, Feb 15, 6-9pm at WG Young CS50.

  A recording will be uploaded to BruinLearn.

Last lecture summary

In the past lectures, we covered tricks that we can do in initialization, regularization, and data augmentation to improve the performance of neural networks.

But what about the optimizer, stochastic gradient descent? Can we improve this for deep learning?

That's the topic of our next lecture.

Last lecture summary

In the past lectures, we covered tricks that we can do in initialization, regularization, and data augmentation to improve the performance of neural networks.

But what about the optimizer, stochastic gradient descent? Can we improve this for deep learning?

That's the topic of our next lecture.

An aside (label smoothing):

layer of a neural network concatenated with "1" to account for the bias. For a network trained with hard targets, we minimize the expected value of the cross-entropy between the true targets y_k and the network's outputs p_k as in $H(\boldsymbol{y}, \boldsymbol{p}) = \sum_{k=1}^{K} -y_k \log(p_k)$, where y_k is "1" for the correct class and "0" for the rest. For a network trained with a label smoothing of parameter α, we minimize instead the cross-entropy between the modified targets y_k^{LS} and the networks' outputs p_k, where $y_k^{LS} = y_k(1-\alpha) + \alpha/K$.

https://arxiv.org/pdf/1906.02629.pdf

Optimization for neural networks

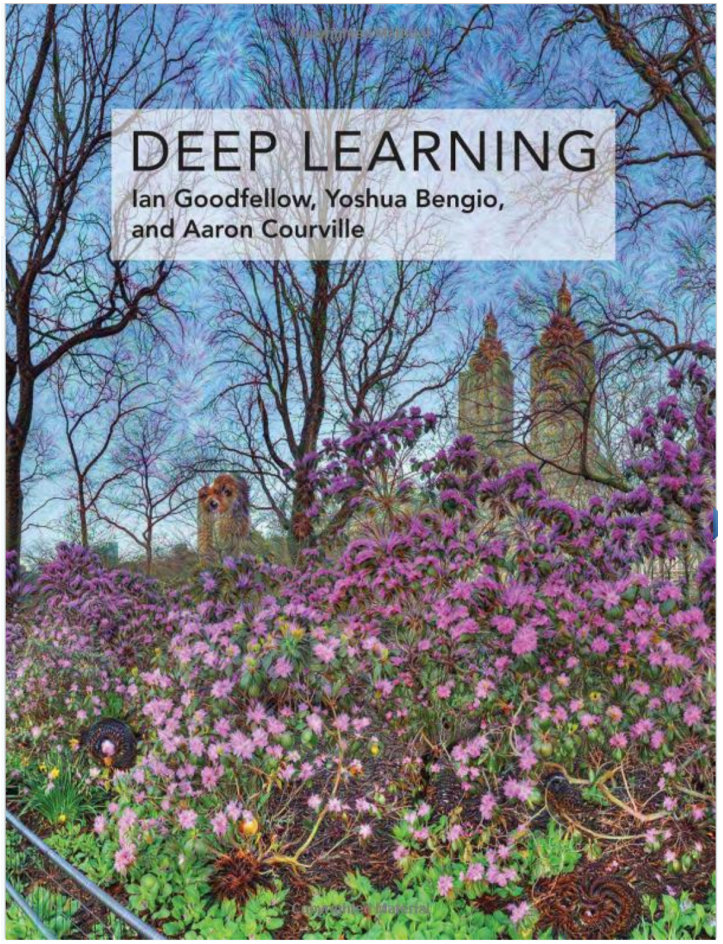
In this lecture, we'll talk about specific techniques in optimization that aid in training neural networks.

- Stochastic gradient descent
- Momentum and Nesterov momentum
- Adaptive gradients
- RMSProp
- Adaptive moments (Adam)
- Overview of second order methods
- Challenges of gradient descent

Reading

Reading:

Deep Learning, Chapter 8 (intro), 8.1, 8.2, 8.3, 8.4, 8.5, 8.6 (skim)



Where we are now

At this point, we know:

- Neural network architectures.
- Hyperparameters and cost functions to use for neural networks.
- How to calculate gradients of the loss w.r.t. all parameters in the neural network. Backprop
- How to initialize the weights and regularize the network in ways to improve the training of the network.

We do know how to optimize these networks with stochastic gradient descent. But can it be improved?

In this lecture, we talk about how to make optimization more efficient and effective.

Gradient descent

A refresher on gradient descent.

- Cost function: $J(\theta)$
- Parameters: θ

Then, the gradient descent step is:

$$\theta \leftarrow \theta - \epsilon \nabla_\theta J(\theta)$$

Stochastic gradient descent

Calculating the gradient exactly is expensive, because it requires evaluating the model on all m examples in the dataset. This leads to an important distinction.

- Batch algorithm: uses all m examples in the training set to calculate the gradient.
- Minibatch algorithm: approximates the gradient by calculating it using k training examples, where $m > k > 1$.
- Stochastic algorithm: approximates the gradient by calculating it over one example.

It is typical in deep learning to use minibatch gradient descent. Note that some may also use minibatch and stochastic gradient descent interchangeably.

A note: small batch sizes can be seen to have a regularization effect, perhaps because they introduce to noise to the training process.

Stochastic gradient descent

Stochastic gradient descent proceeds as follows.

Set a learning rate ε and an initial parameter setting θ. Set a minibatch size of m examples. Until the stopping criterion is met:

- Sample m examples from the training set, $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\}$ and their corresponding outputs $\{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(m)}\}$.
- Compute the gradient estimate:

$$\mathbf{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J(\theta)$$

- Update parameters:

$$\theta \leftarrow \theta - \varepsilon \mathbf{g}$$

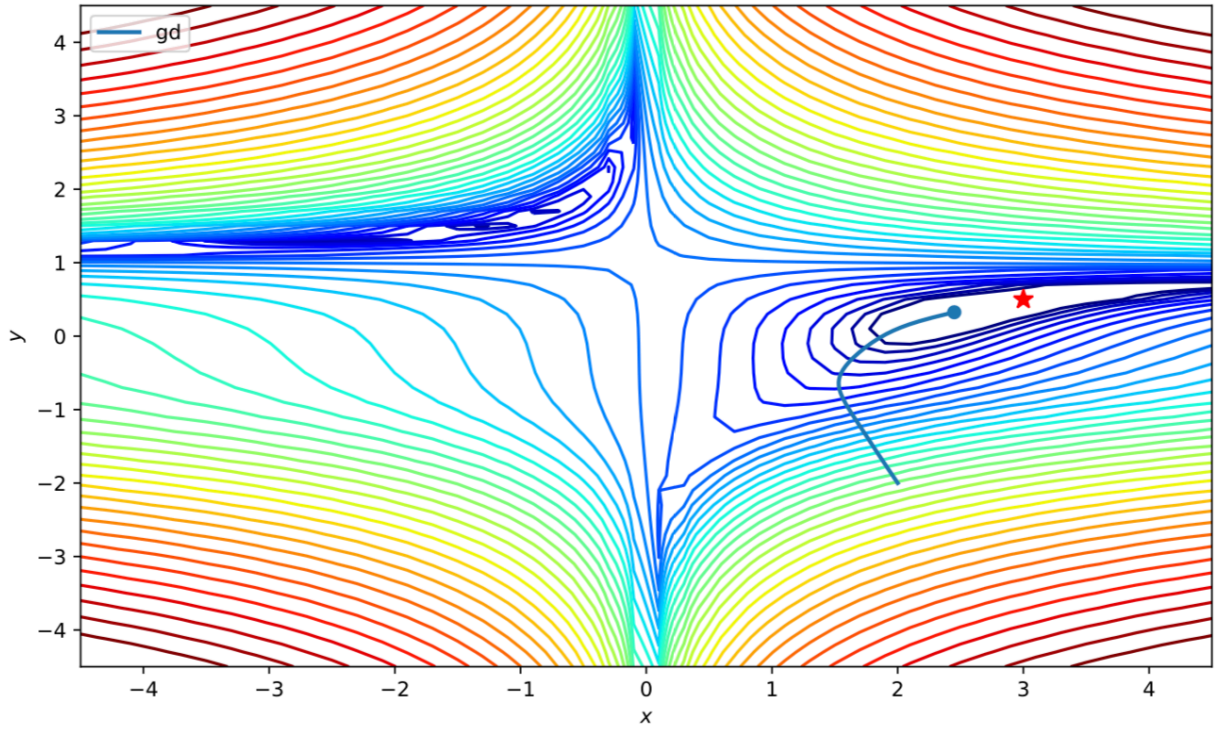
Stochastic gradient descent

softmax.loss_and_grad !)

```
while last_diff > tol:
    cost, g = func(x)
    x -= eps*g
    last_diff = np.linalg.norm(x - path[-1])
```

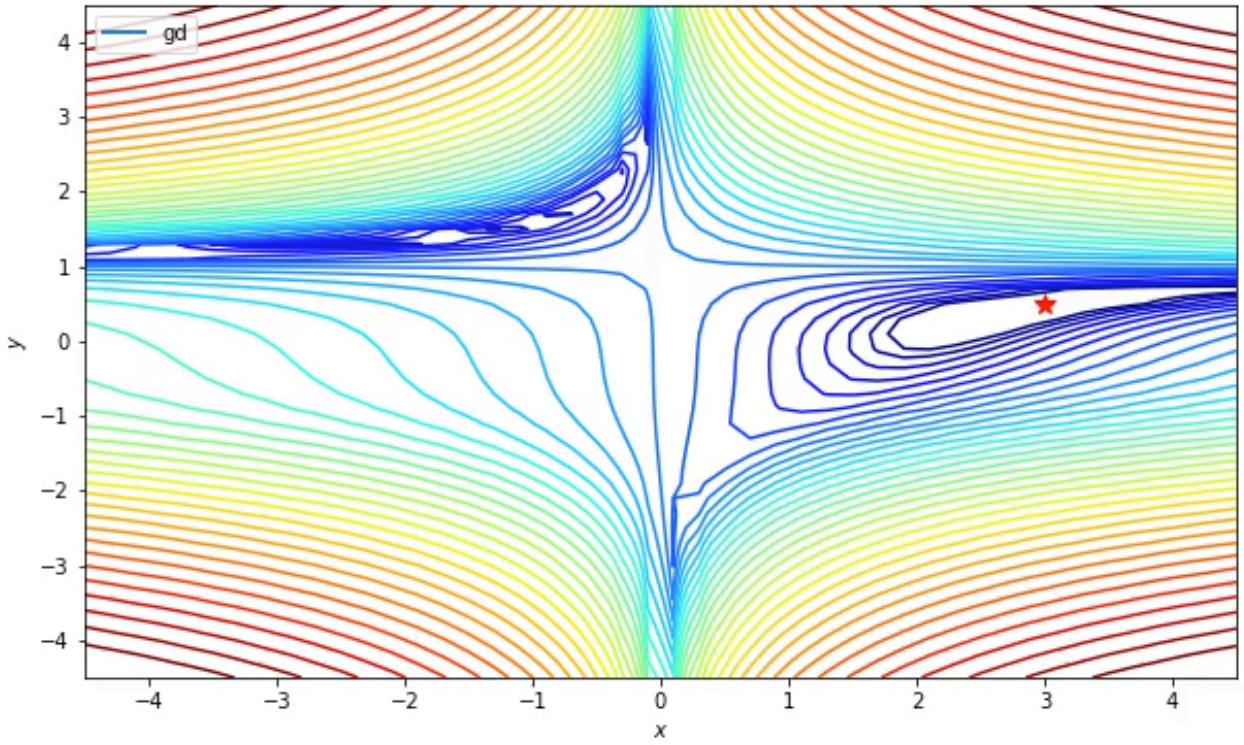
Stochastic gradient descent (opt 1)

The following shows gradient descent applied to Beale's function, for two initializations at $(2, -2)$ and $(-3, 3)$. The two initializations are shown because later on we'll contrast to other techniques. The iteration count is capped at $10,000$ iterations, so gradient descent does not get to the minimum.

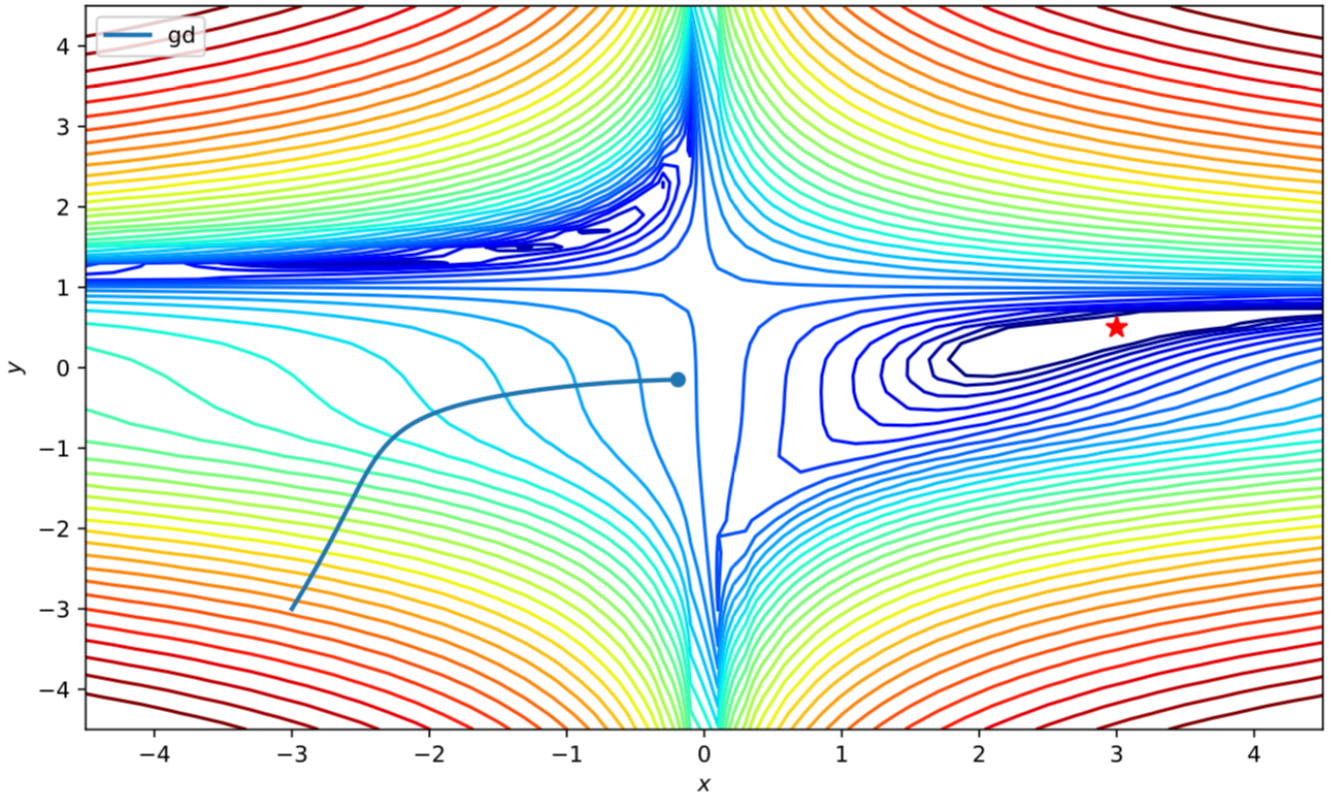


Video: `http://seas.ucla.edu/~kao/opt_anim/1gd.mp4`
Animation help thanks to: `http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/`

Stochastic gradient descent



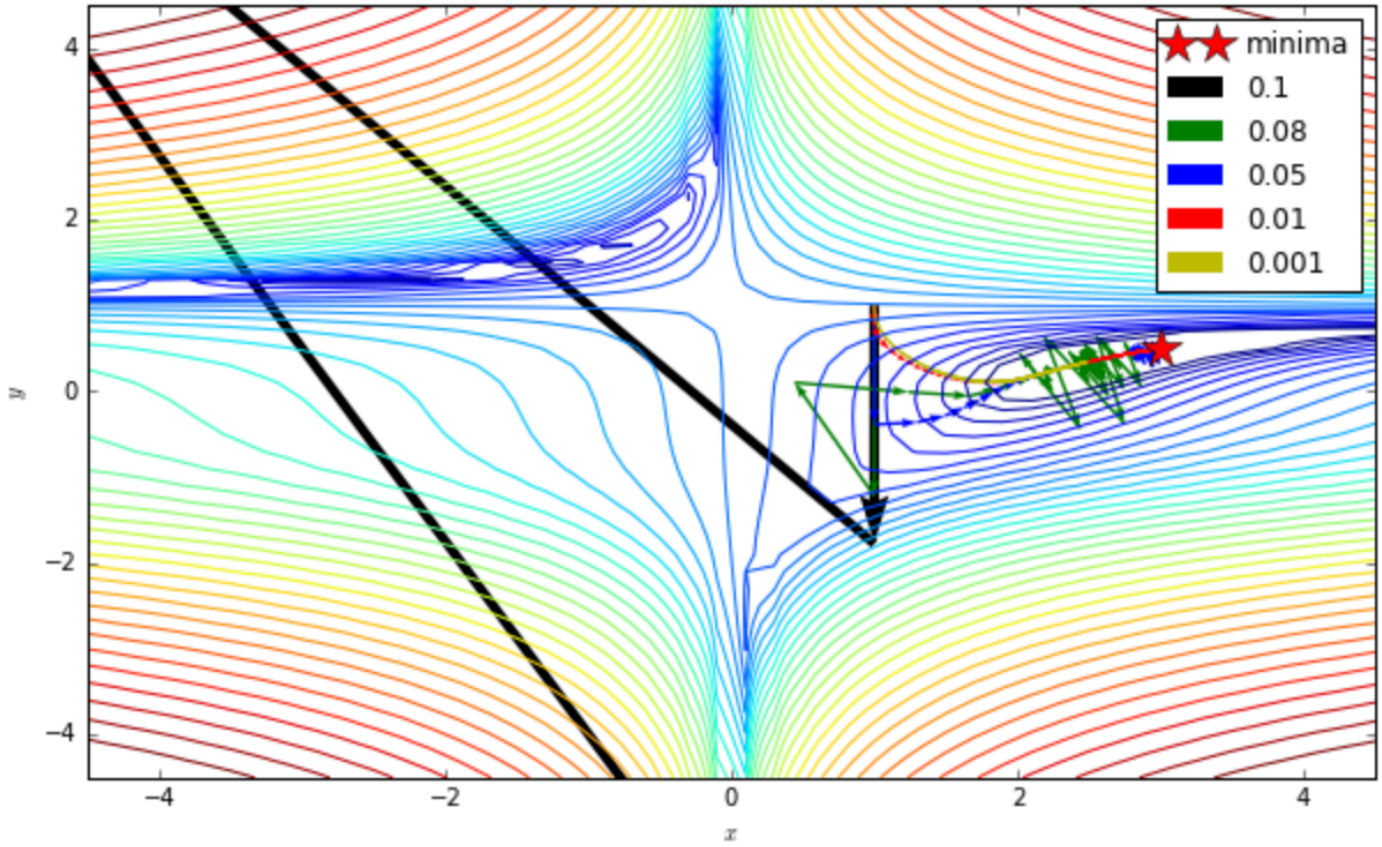
Stochastic gradient descent (opt 2)



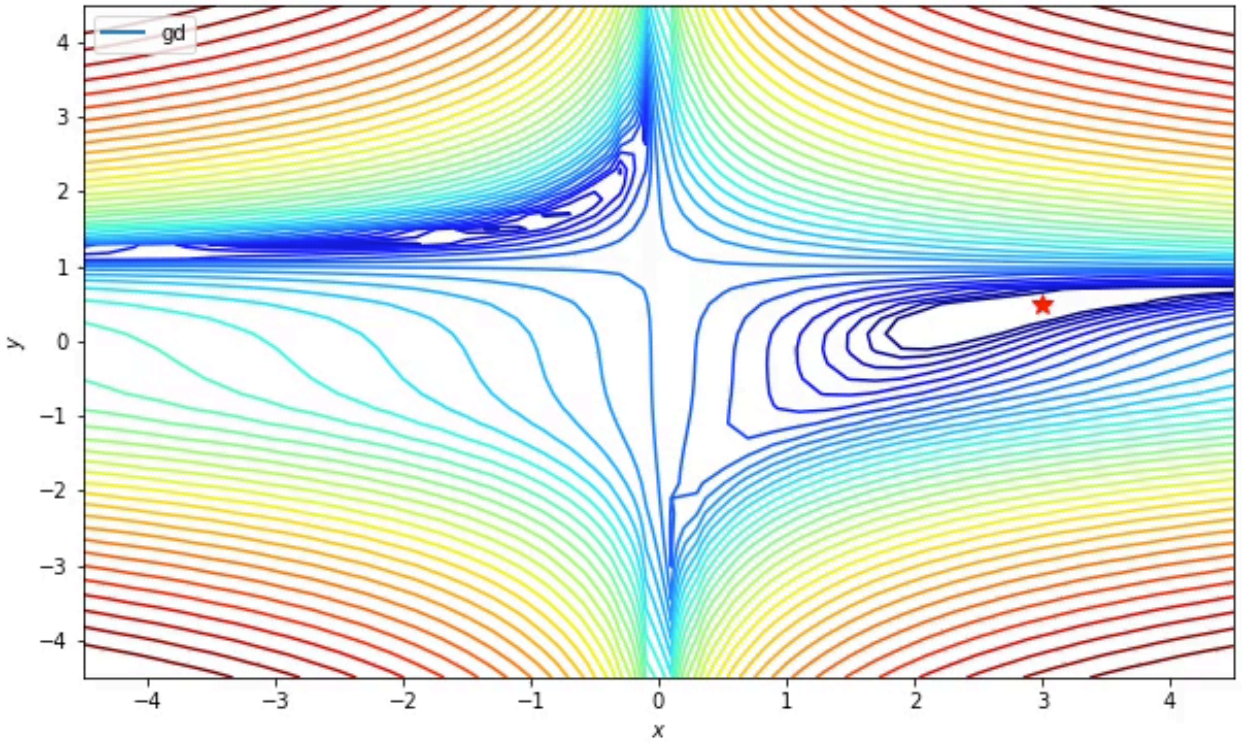
Video: http://seas.ucla.edu/~kao/opt_anim/2gd.mp4

Finding the optimal weights through gradient descent

Varying the learning rate:



Stochastic gradient descent



Momentum

In momentum, we maintain the running mean of the gradients, which then updates the parameters.

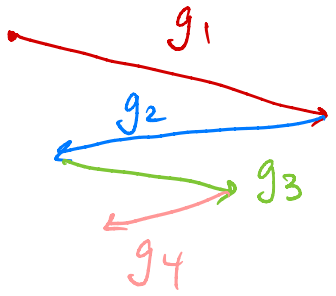
Initialize $\mathbf{v} = 0$. Set $\alpha \in [0, 1]$. Typical values are $\alpha = 0.9$ or 0.99. Then, until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Update:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta + \mathbf{v}$$



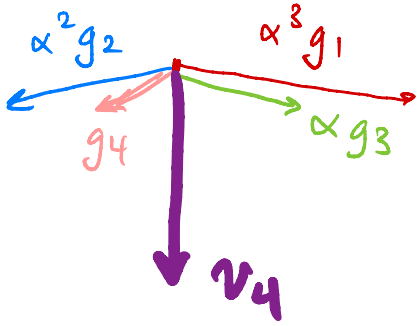
$v_0 = 0$

$v_1 = -\varepsilon g_1$

$v_2 = \alpha v_1 - \varepsilon g_2 = -\alpha\varepsilon g_1 - \varepsilon g_2$

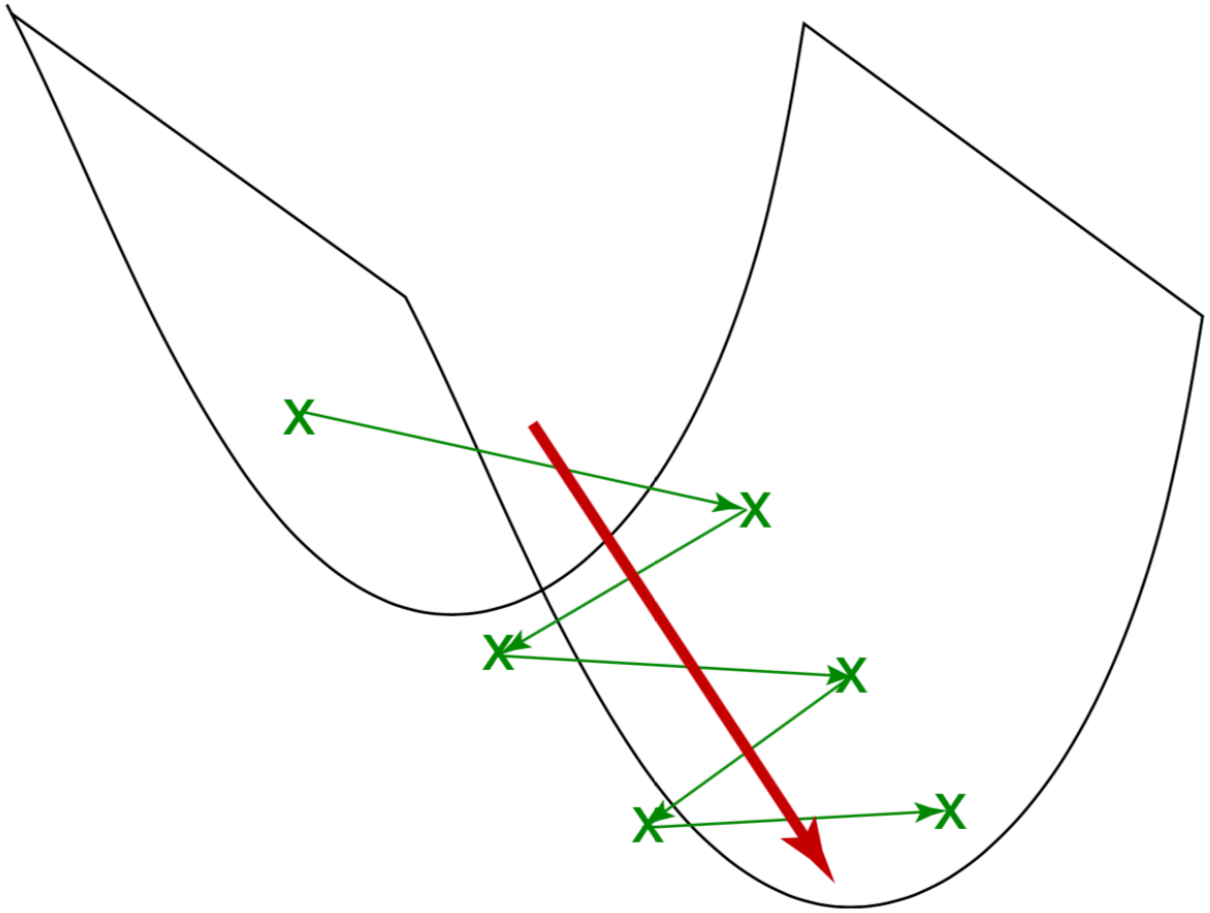
$v_3 = \alpha v_2 - \varepsilon g_3 = -\varepsilon(\alpha^2 g_1 + \alpha g_2 + g_3)$

$v_4 = \alpha v_3 - \varepsilon g_4 = -\varepsilon(\alpha^3 g_1 + \alpha^2 g_2 + \alpha g_3 + g_4)$



Momentum (cont.)

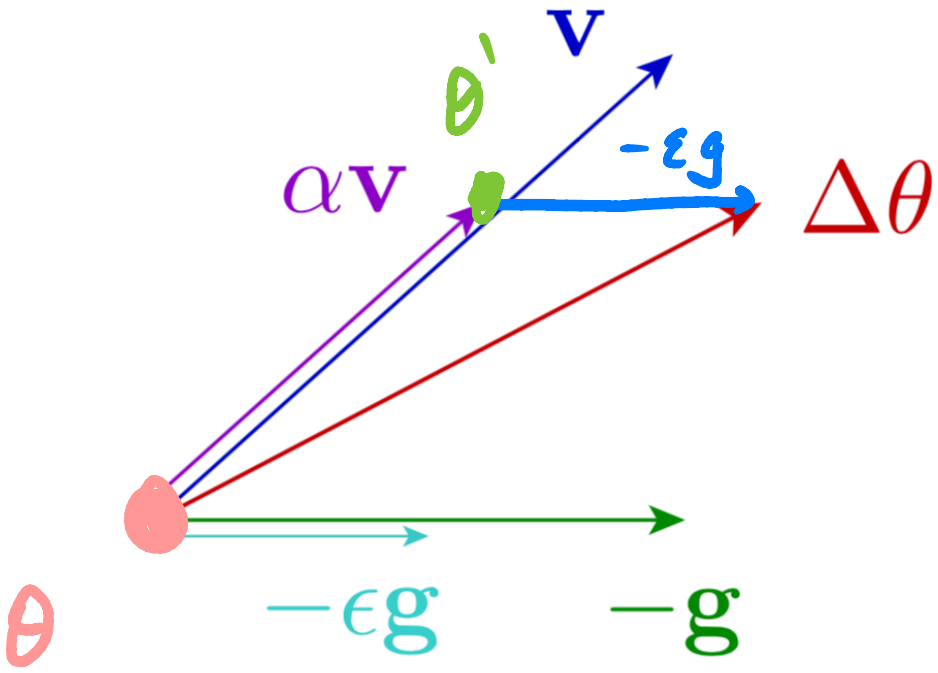
An example of how momentum is useful is to consider a tilted surface with high curvature. Stochastic gradient descent may make steps that zigzag, although in general it proceeds in the right direction. Momentum will average away the zigzagging components.



Momentum (cont.)

This modification augments the gradient with the running average of previous gradients, which is analogous to a gradient "momentum." The following image is appropriate to have in mind:

$\alpha v - \epsilon g$



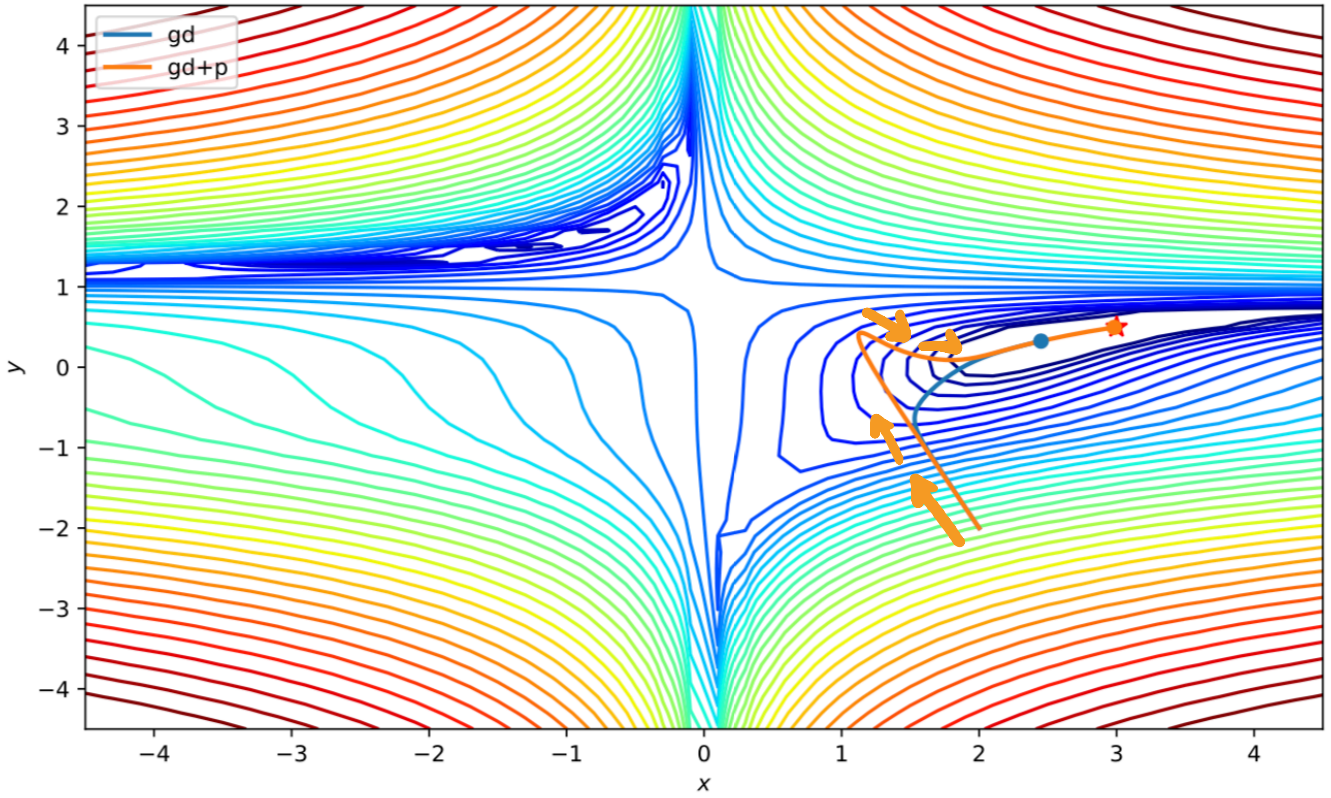
Momentum

```
alpha = 0.9
p = 0
while last_diff > tol:
    cost, g = func(x)
    p = alpha*p - eps*g
    x += p
    last_diff = np.linalg.norm(x - path[-1])
```

p = v

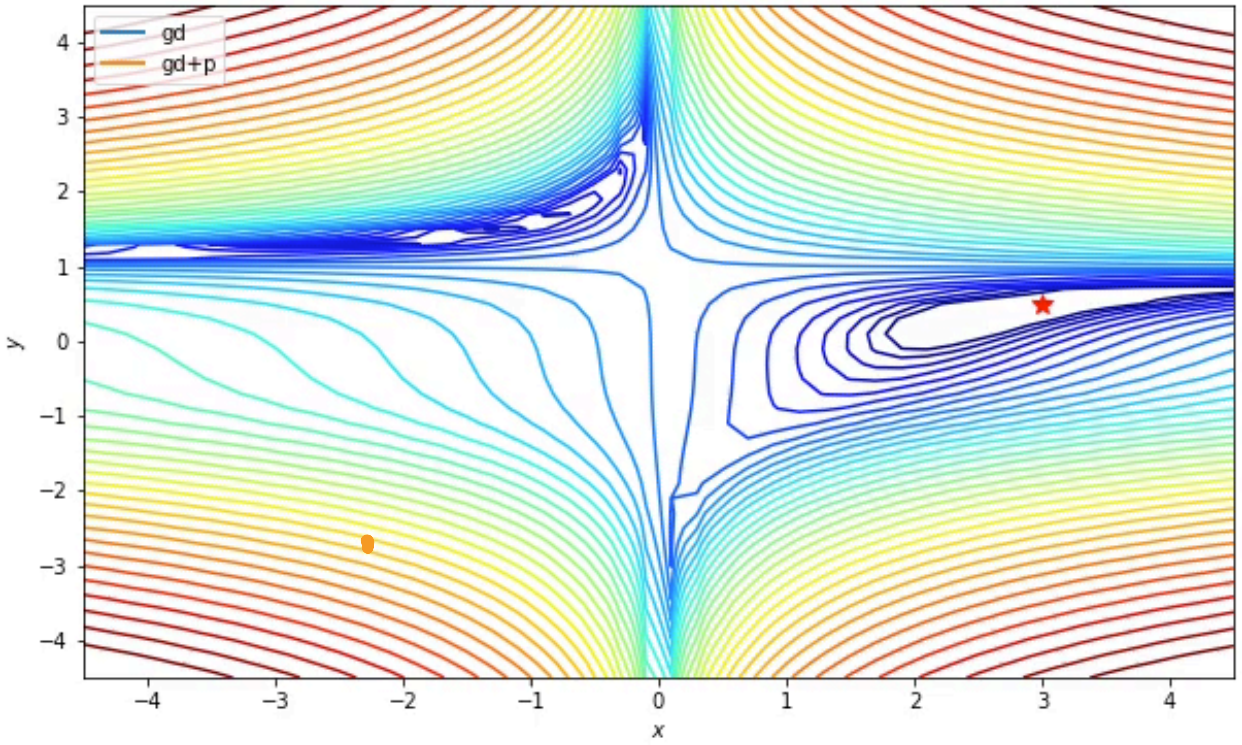
Momentum (opt 1)

gd+p denotes gradient descent with momentum.



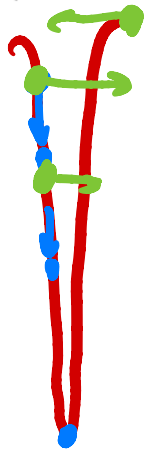
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p.mp4`

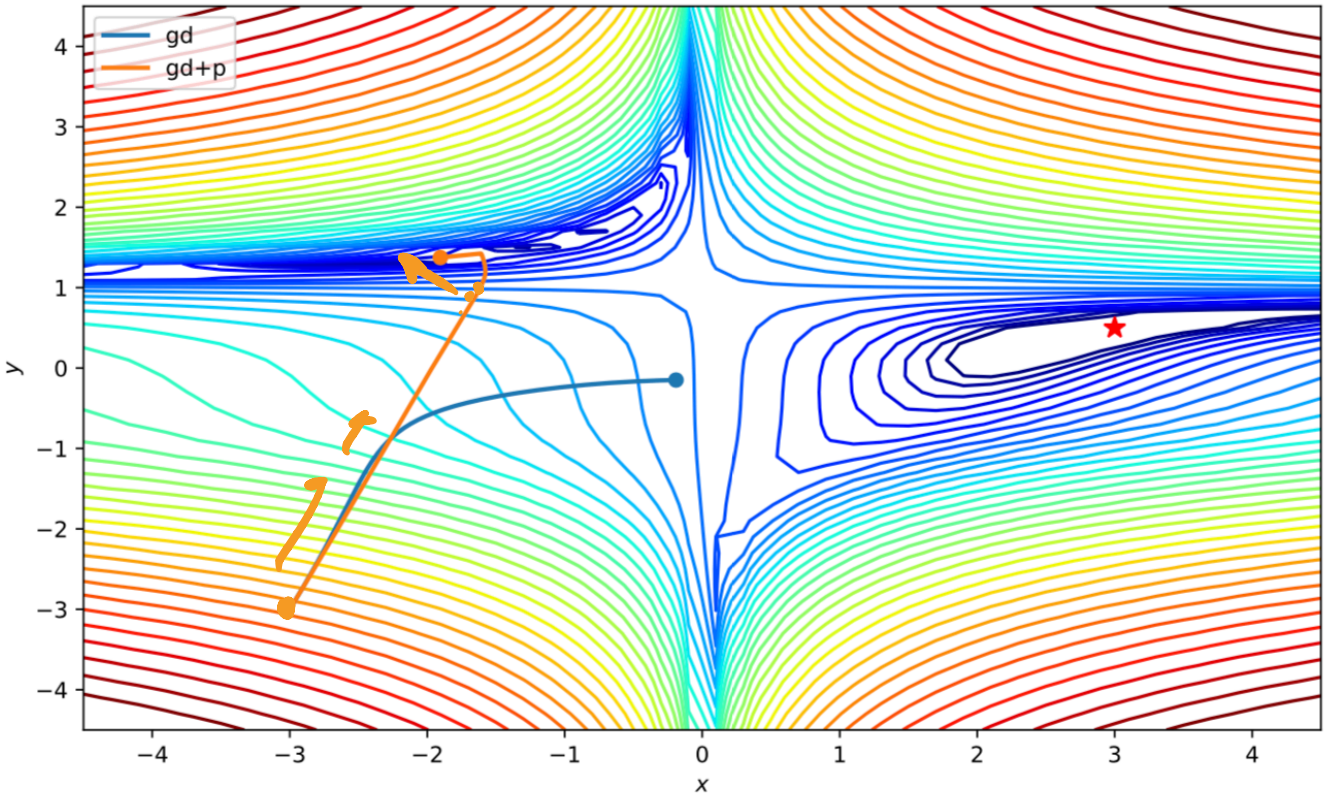
Momentum



Momentum (opt 2)

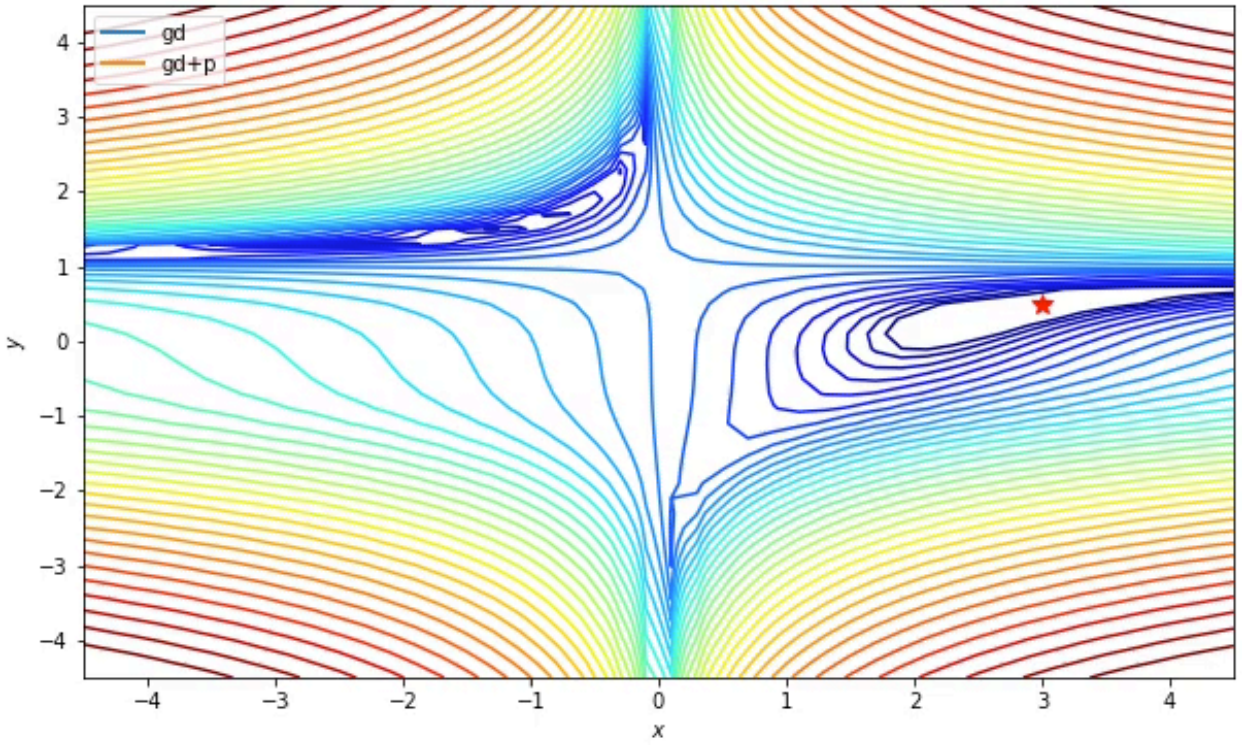
Notice how momentum pushes the descent to find a local, but not global, minimum.





Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p.mp4`

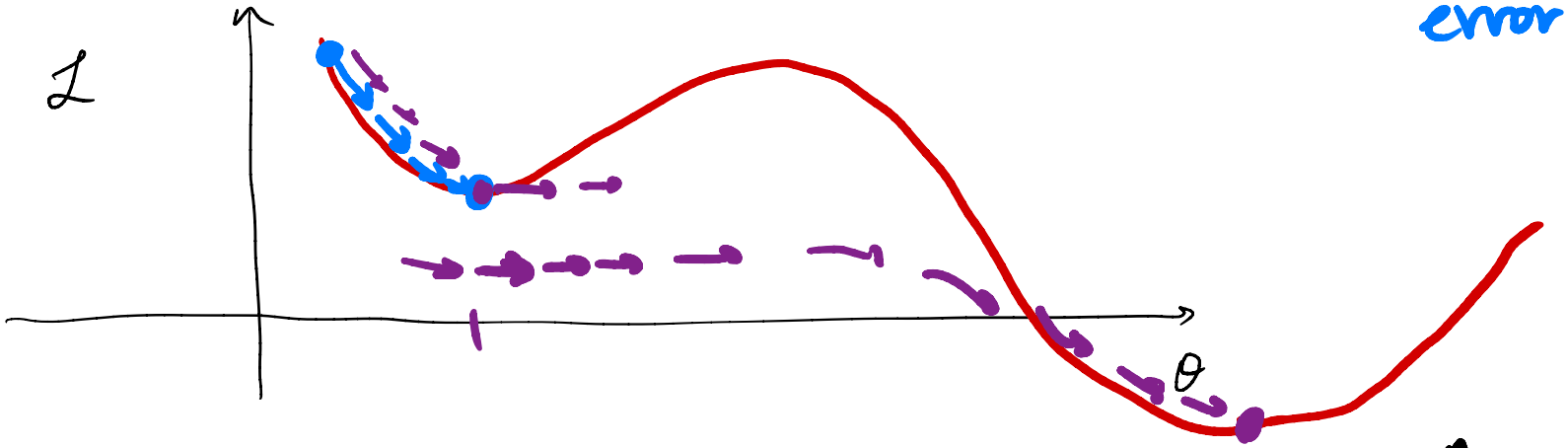
Momentum



Does momentum help with local optima?

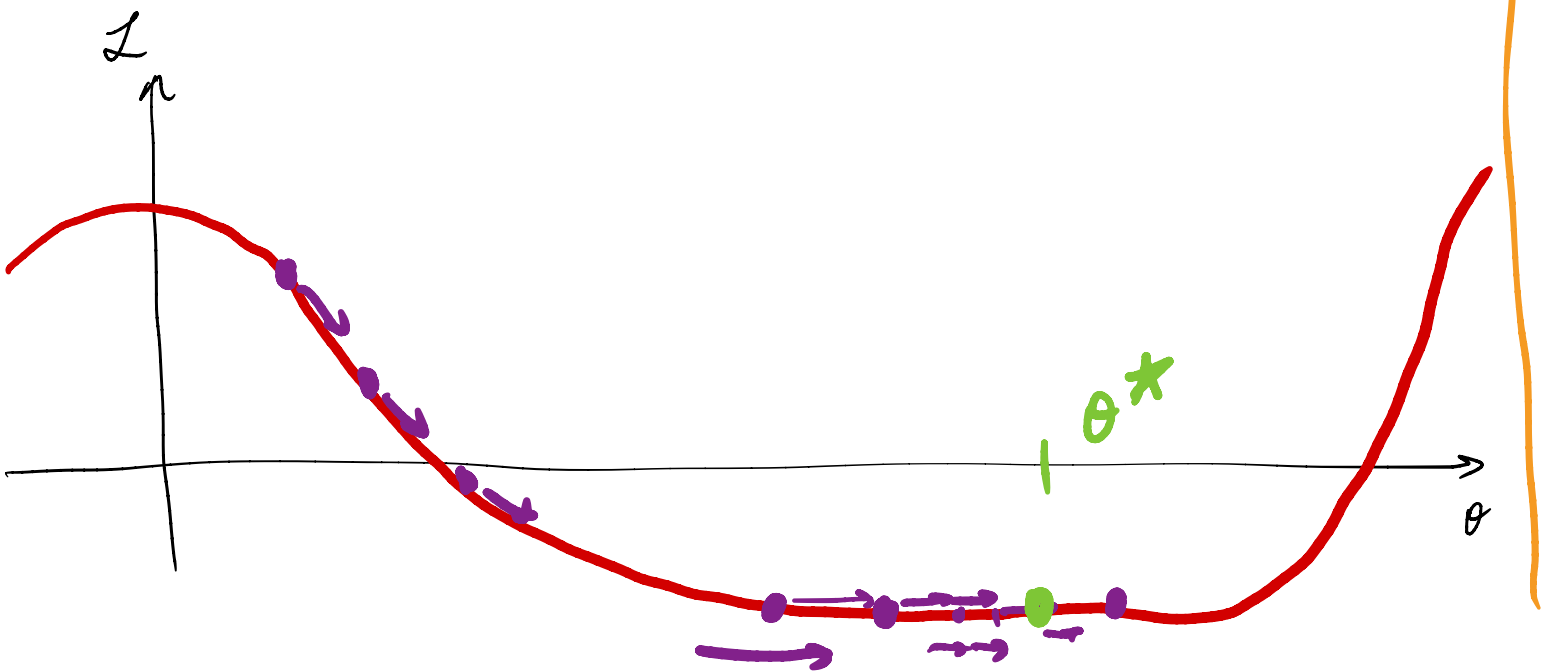
Momentum

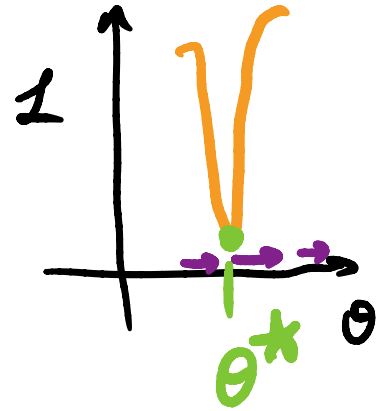
Does momentum help with local optima? (Avoid local optima w/ higher error?)



steep

What kind of local optima does momentum tend to find?





Nesterov momentum

Nesterov momentum is similar to momentum, except the gradient is calculated at the parameter setting after taking a step along the direction of the momentum.

Initialize $\mathbf{v} = 0$. Then, until stopping criterion is met:

- Update:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \nabla_\theta J(\theta + \alpha \mathbf{v})$$

- Gradient step:

$$\theta \leftarrow \theta + \mathbf{v}$$

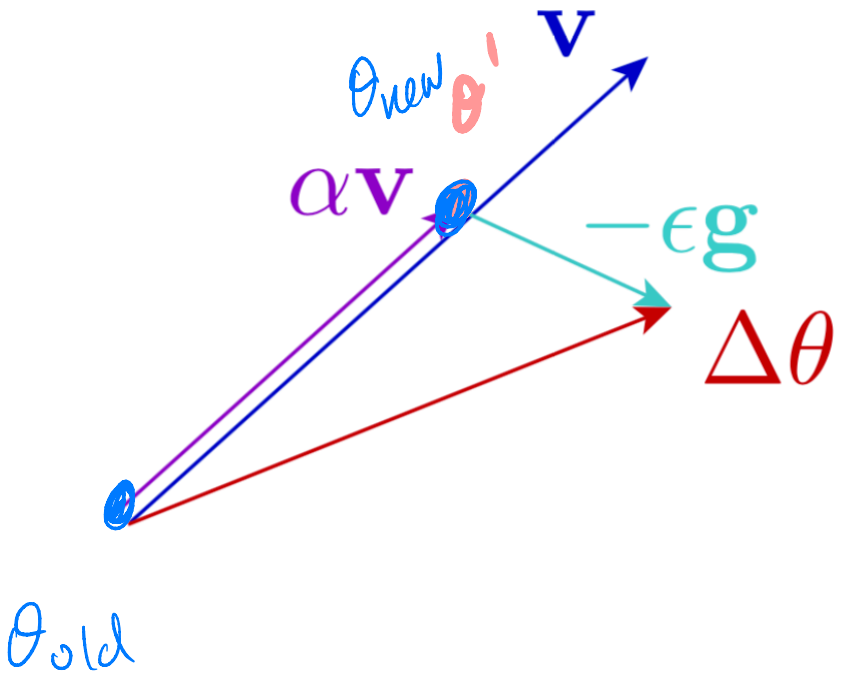
Vanilla Momentum:

$$v \leftarrow \alpha v - \varepsilon \nabla_\theta J(\theta)$$

$\nabla_{\tilde{\theta}} J(\tilde{\theta})$

Nesterov momentum (cont.)

The following image is appropriate for Nesterov momentum:



$\theta_{new} = \theta_{t+1}$

$\theta_{old} = \theta_t$

Nesterov momentum

$\theta_{new} = \tilde{\theta}_{new} - \alpha v_{new}$

$\theta_{old} = \tilde{\theta}_{old} - \alpha v_{old}$

By performing a change of variables with $\tilde{\theta}_{\text{old}} = \theta_{\text{old}} + \alpha \mathbf{v}_{\text{old}}$, it's possible to show that the following is equivalent to Nesterov momentum. (This representation doesn't require evaluating the gradient at $\theta + \alpha \mathbf{v}$.)

- Update:

$$\mathbf{v}_{\text{new}} = \alpha \mathbf{v}_{\text{old}} - \varepsilon \nabla_{\tilde{\theta}_{\text{old}}} J(\tilde{\theta}_{\text{old}})$$

- Gradient step:

$v \leftarrow \alpha v - \varepsilon \nabla_{\hat{\theta}} J(\hat{\theta})$

$$\tilde{\theta}_{\text{new}} = \tilde{\theta}_{\text{old}} + \mathbf{v}_{\text{new}} + \alpha(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}})$$

- Set $\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{old}}$, $\tilde{\theta}_{\text{new}} = \tilde{\theta}_{\text{old}}$.

$\tilde{\theta} \leftarrow \tilde{\theta} + v_{new} + \alpha (v_{new} - v_{old})$

$\theta_{new} = \theta_{old} + v_{new}$

$\tilde{\theta}_{new} - \alpha v_{new} = \tilde{\theta}_{old} - \alpha v_{old} + v_{new}$

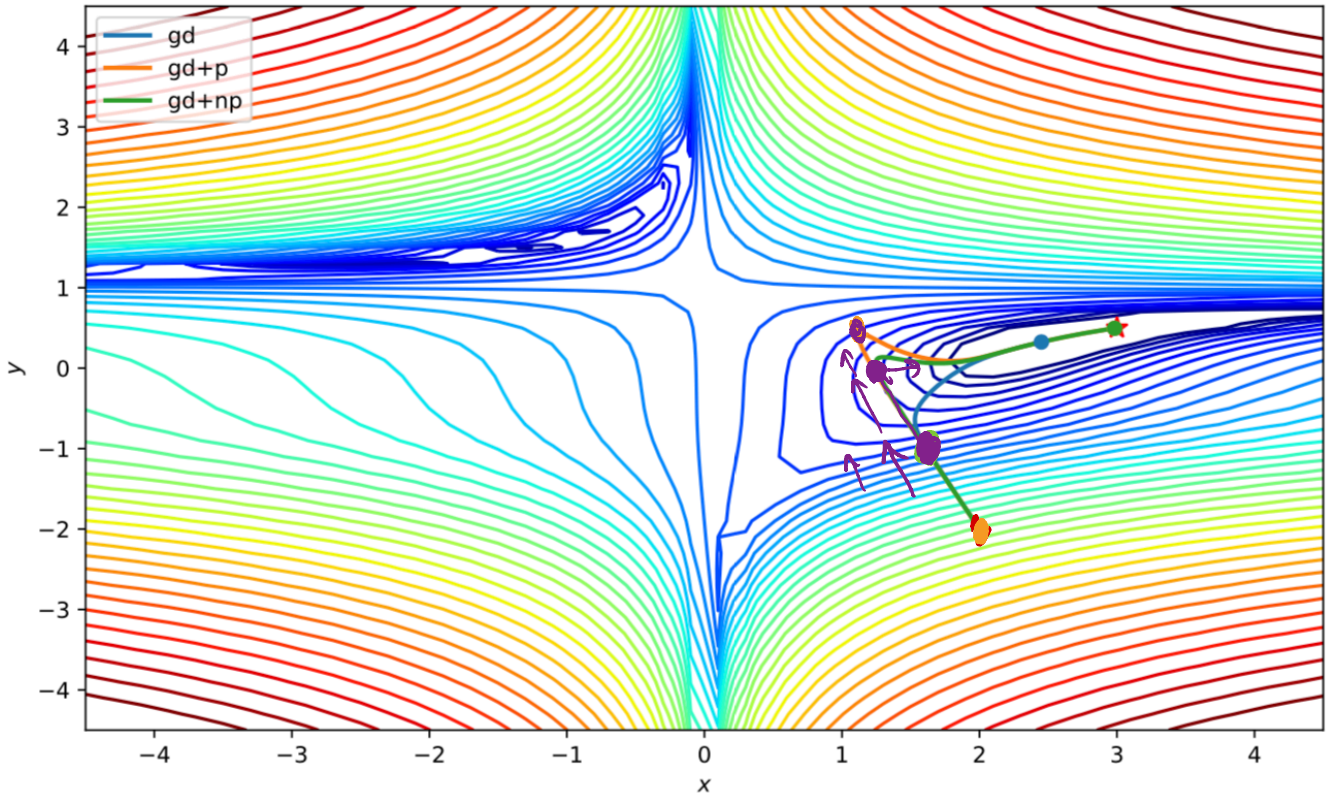
$\Rightarrow \tilde{\theta}_{new} = \tilde{\theta}_{old} + v_{new} + \alpha (v_{new} - v_{old})$

Nesterov momentum

```
alpha = 0.9
p = np.zeros_like(x)
while last_diff > tol:
    cost, g = func(x)
    p_old = p
    p = alpha*p - eps*g
    x += p + alpha*(p-p_old)
    last_diff = np.linalg.norm(x - path[-1])
```

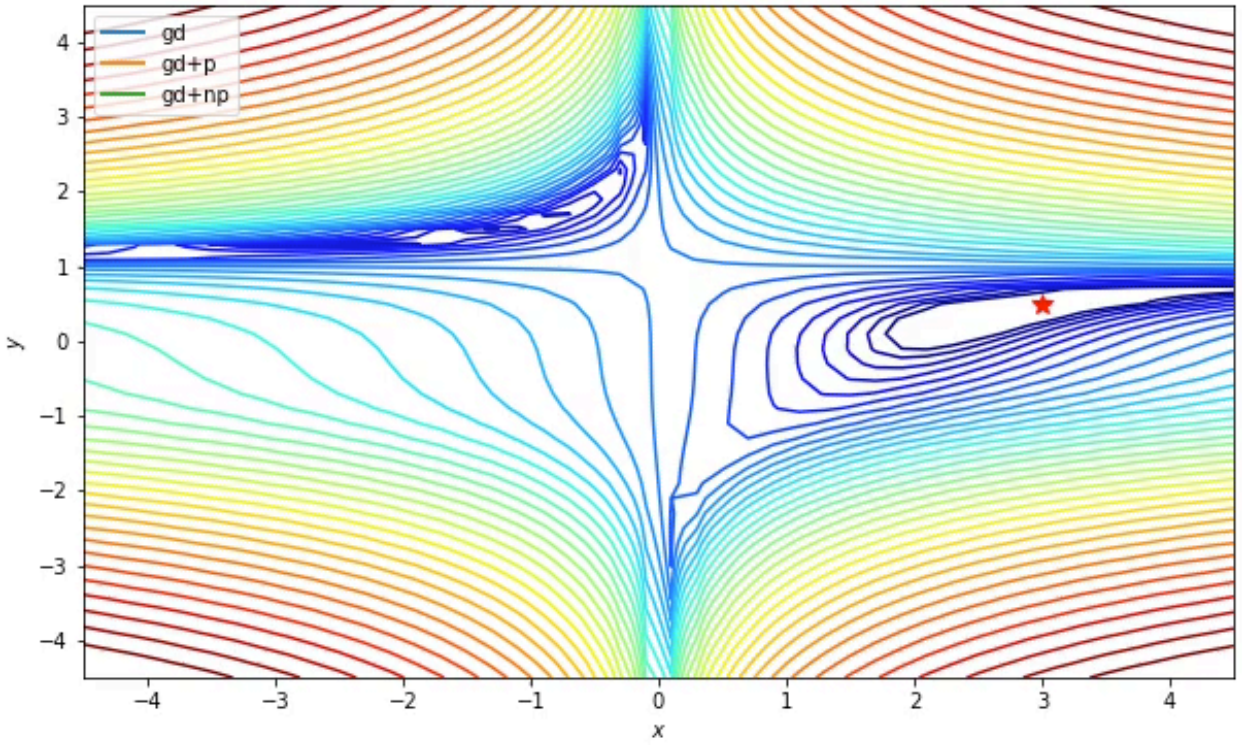
Nesterov momentum (opt 1)

gd+np denotes gradient descent with Nesterov momentum.



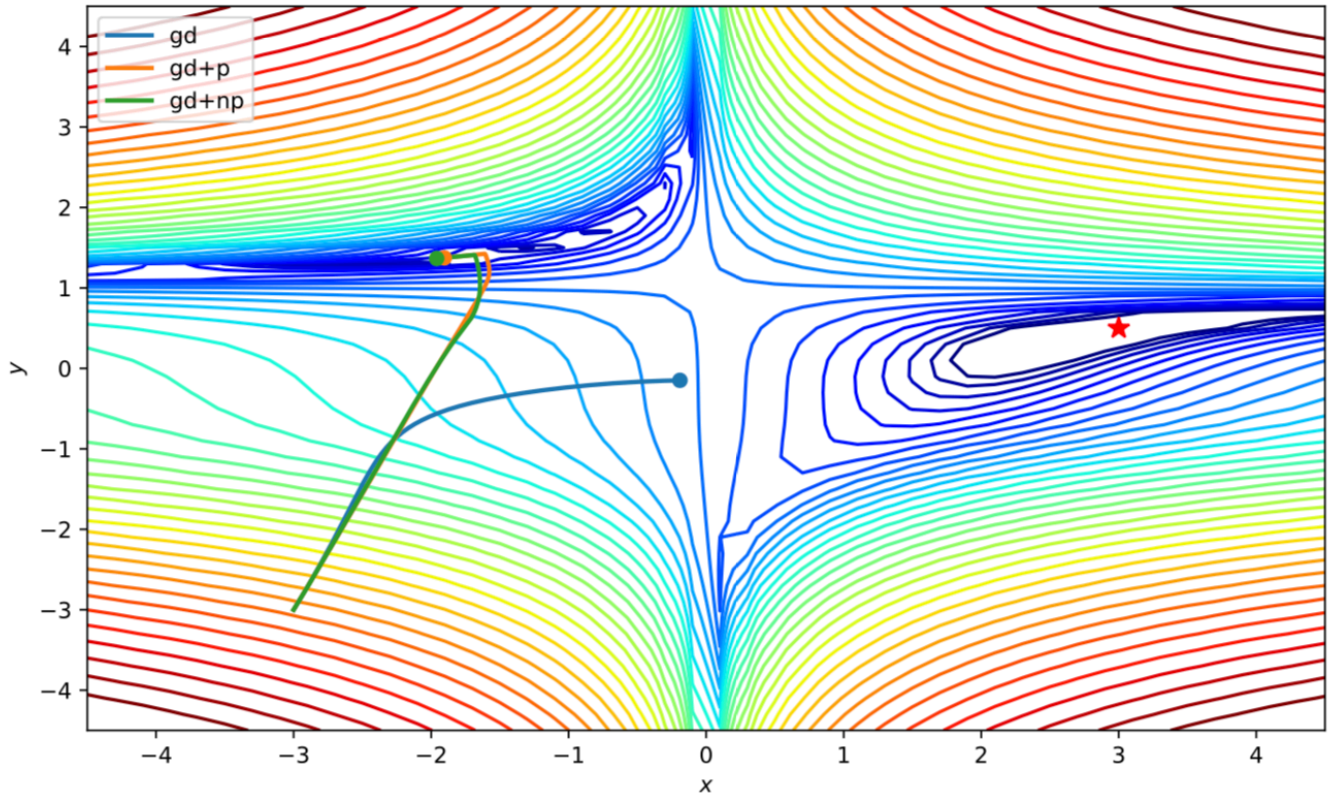
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np.mp4`

Nesterov momentum



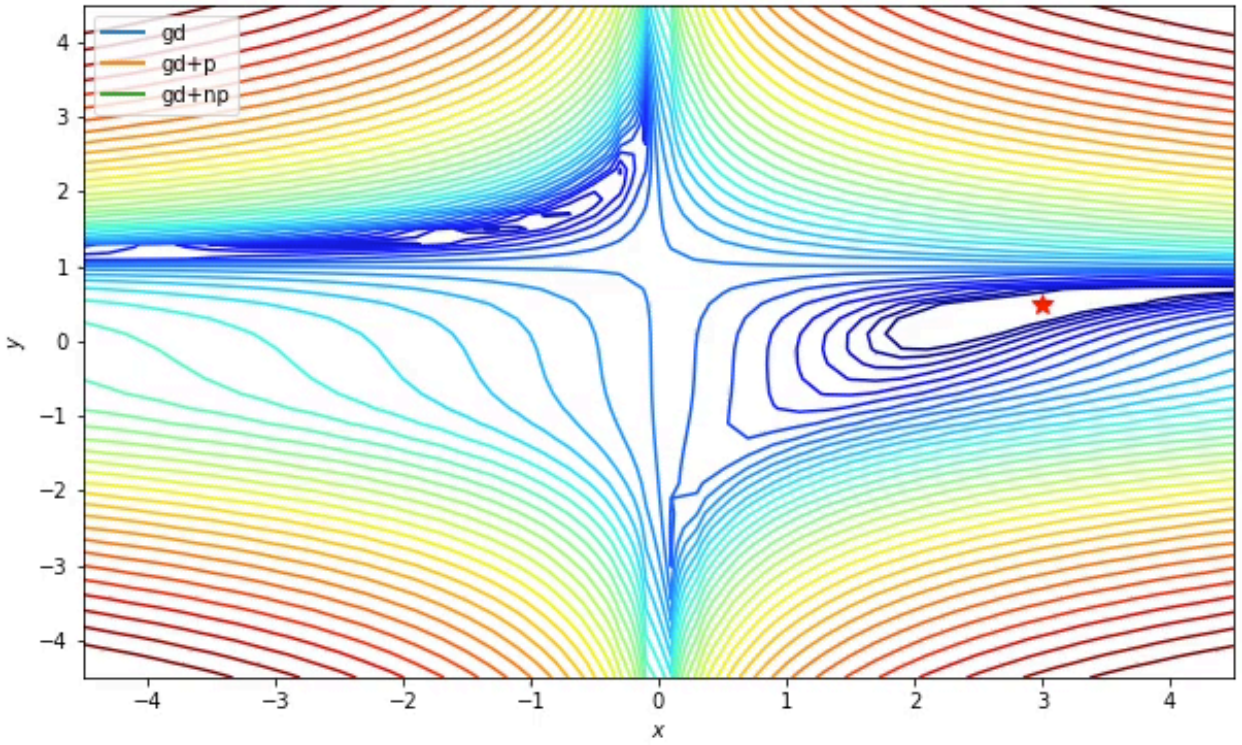
Nesterov momentum (opt 2)

Notice how Nesterov momentum finds the same local minimum as momentum.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np.mp4`

Nesterov momentum



Is there a good way to adapt the learning rule?

Techniques to adapt the learning rate

Choosing ε judiciously can be important for learning. In the beginning, a larger learning rate is typically better, since bigger updates in the parameters may accelerate learning. However, as time goes on, ε may need to be small to be able to make appropriate updates to the parameters. We mentioned before that often times, one applies a decay rule to the learning rate. This is called *annealing*. A common form to anneal the learning rate is to do so manually when the loss plateaus, or to anneal it after a set number of epochs of gradient descent.

Another approach is to update the learning rate based off of the history of gradients.

Adaptive gradient (Adagrad)

Adaptive gradient (Adagrad) is a form of stochastic gradient descent where the learning rate is decreased through division by the historical gradient norms. We will let the variable a denote a running sum of squares of gradient norms.

Initialize $\mathbf{a} = 0$. Set ν at a small value to avoid division by zero (e.g., $\nu = 1e-7$). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

$\theta \in \mathbb{R}^n$, n params

$$g \in \mathbb{R}^n = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}$$

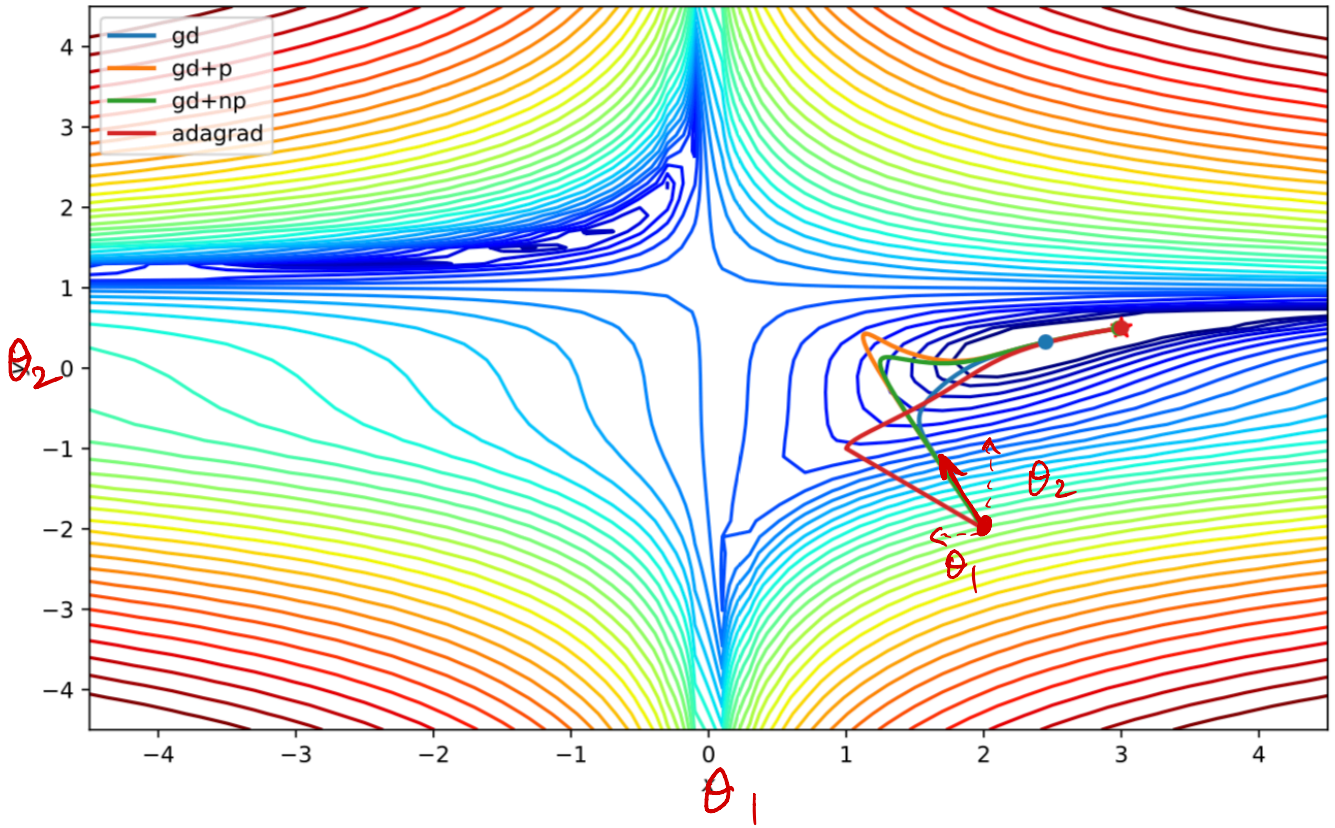
$$\frac{\varepsilon}{\sqrt{a} + \nu} \odot g = \begin{bmatrix} \frac{\varepsilon g_1}{\sqrt{a_1} + \nu} \\ \frac{\varepsilon g_2}{\sqrt{a_2} + \nu} \\ \vdots \end{bmatrix}$$

$$g \odot g = \begin{bmatrix} g_1^2 \\ g_2^2 \\ \vdots \\ g_n^2 \end{bmatrix}$$

Adagrad

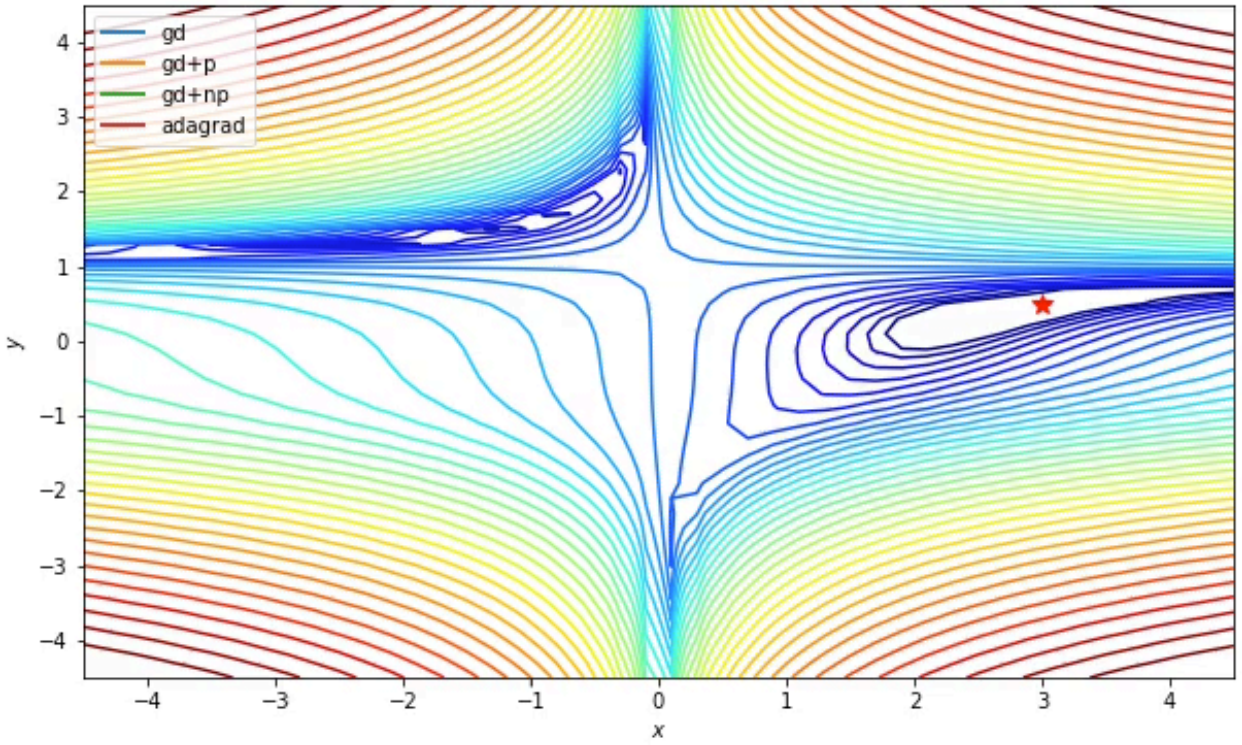
```
a = 0
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a += g*g
    x -= eps * g / (np.sqrt(a) + nu)
    last_diff = np.linalg.norm(x - path[-1])
```

Adagrad (opt 1)



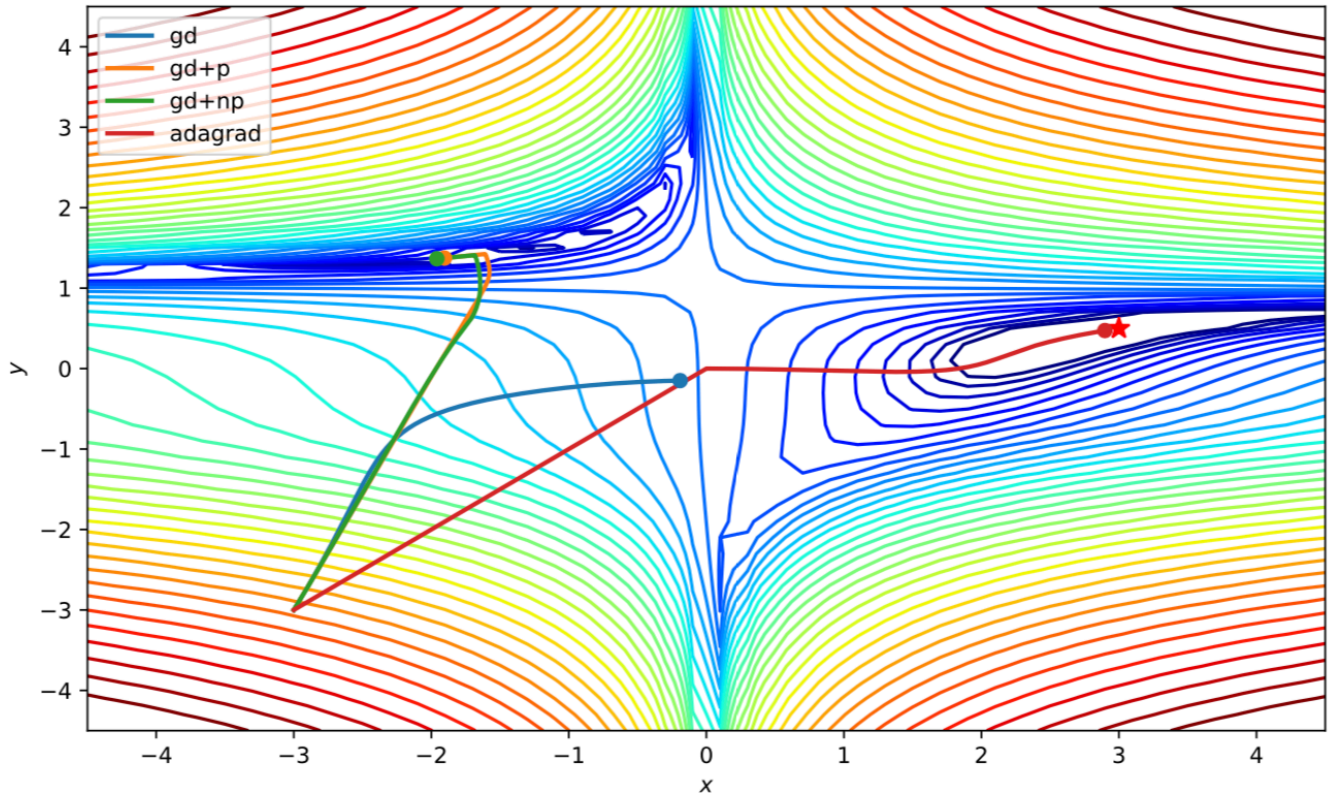
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad.mp4`

Adagrad



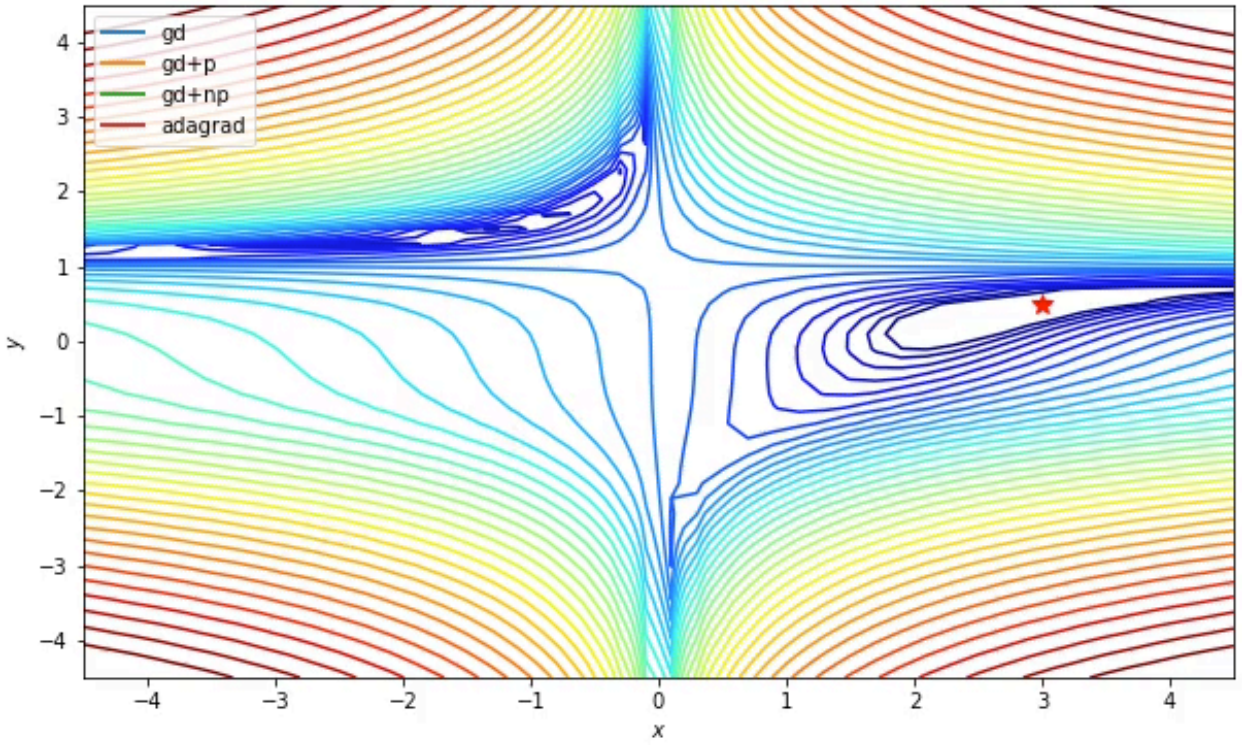
Adagrad (opt 2)

Adagrad proceeds to the global minimum.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad.mp4`

Adagrad



Adagrad

$a_1 = a_1 + g_1^2$

Is there a problem with adagrad?

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

$a_1^{(100)} = a_1^{(99)} \cdot 0.99 + 0.01 g_1^2$

RMSProp

RMSProp augments Adagrad by making the gradient accumulator an exponentially weighted moving average.

Initialize $\mathbf{a} = 0$ and set ν to be sufficiently small. Set β to be between 0 and 1 (typically a value like 0.99). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \beta \mathbf{a} + (1 - \beta)\mathbf{g} \odot \mathbf{g}$$

- Gradient step:

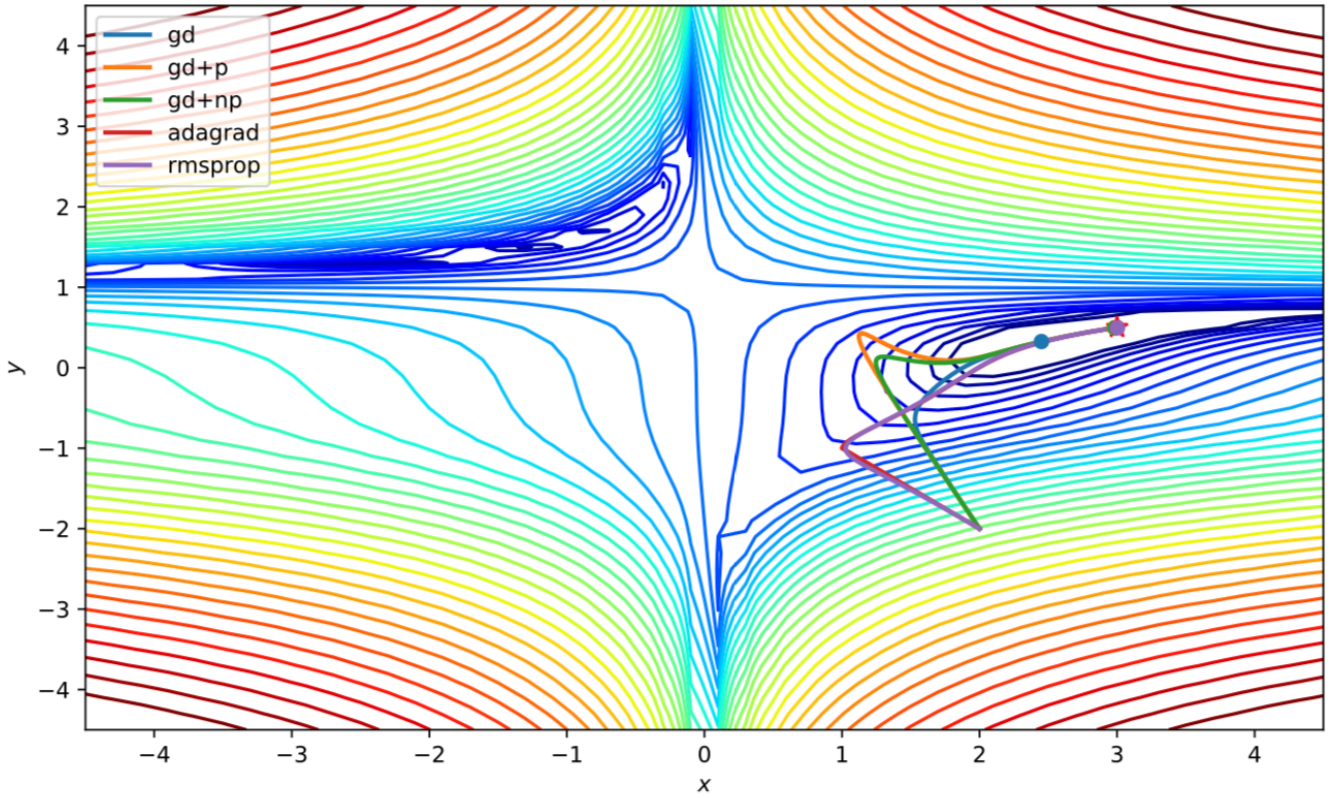
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

$a_1^{(100)} \leftarrow 0.99 \cdot a_1^{(99)} + 0.01 g_1^2$

RMSProp

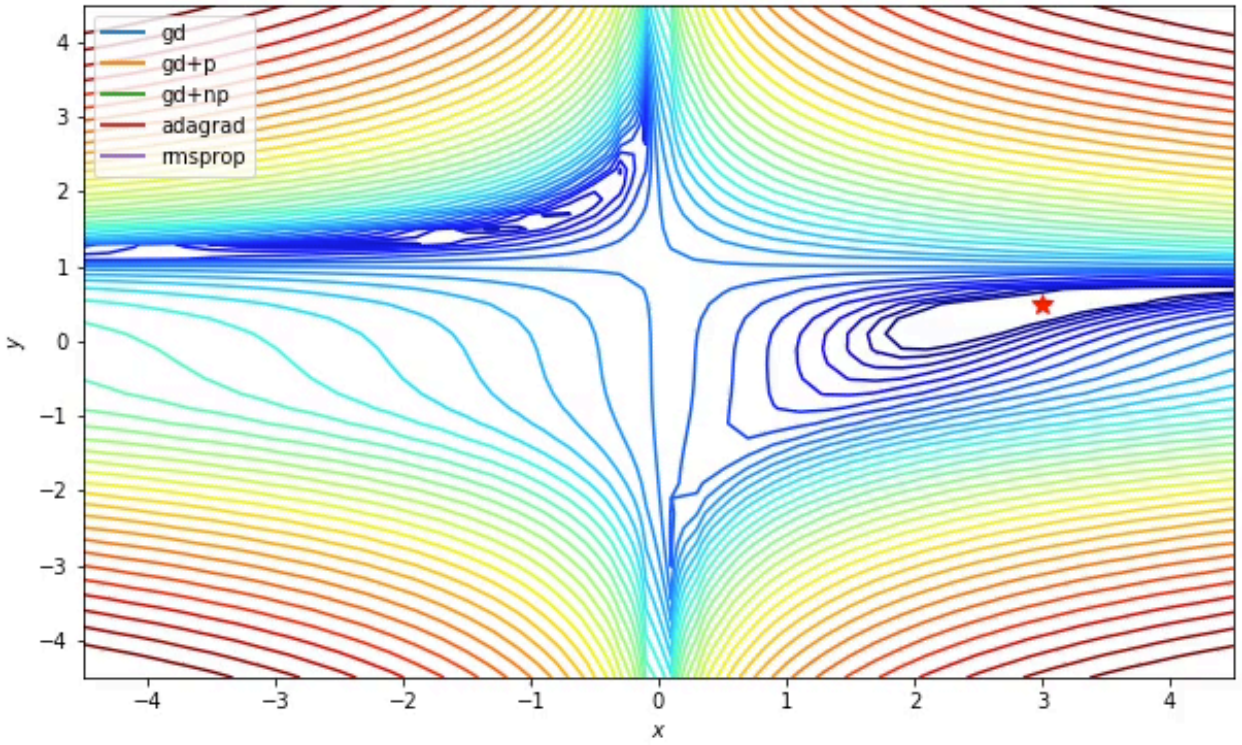
```
a = 0
beta = 0.99
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a = beta * a + (1-beta) * g * g
    x -= eps * g / (np.sqrt(a + nu))
    last_diff = np.linalg.norm(x - path[-1])
```

RMSProp (opt 1)



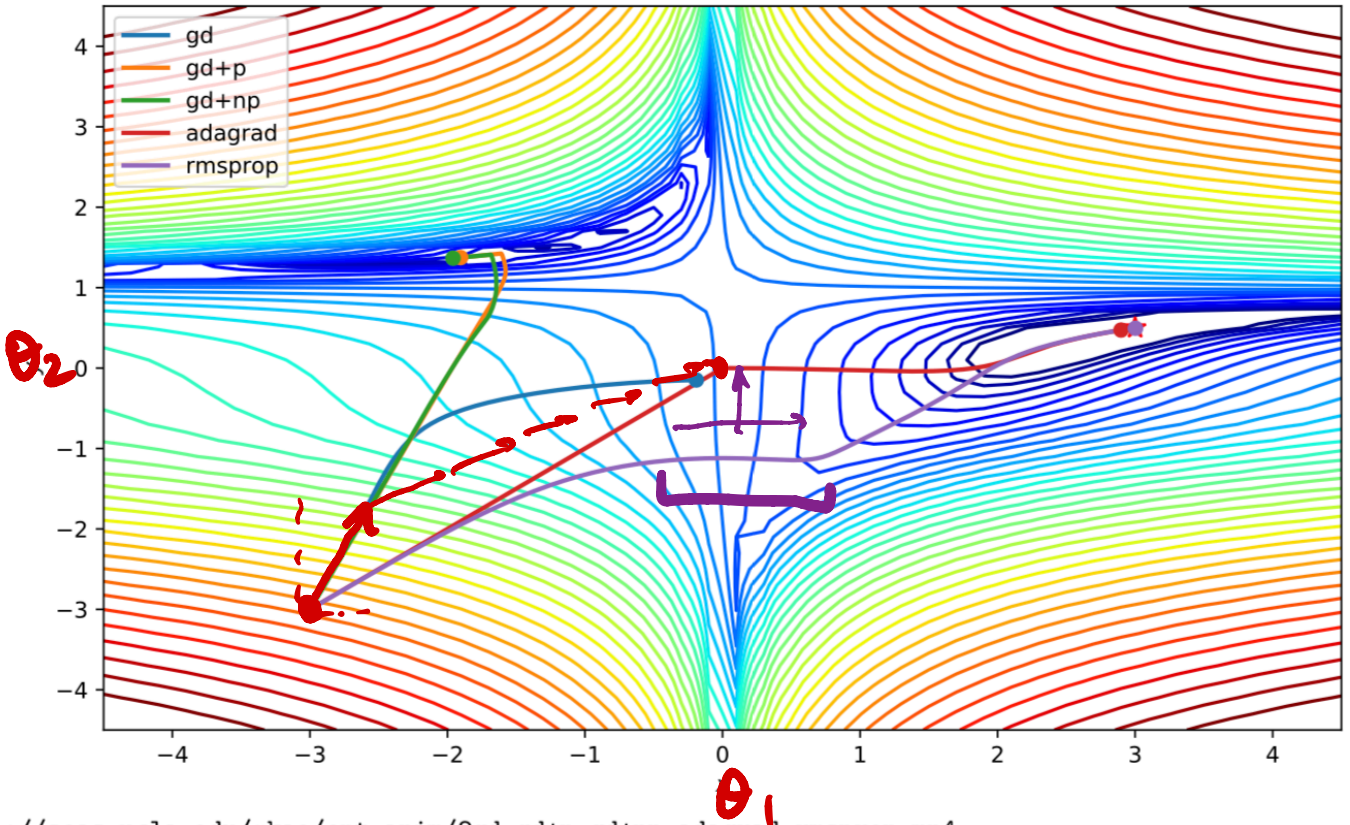
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop.mp4`

RMSProp



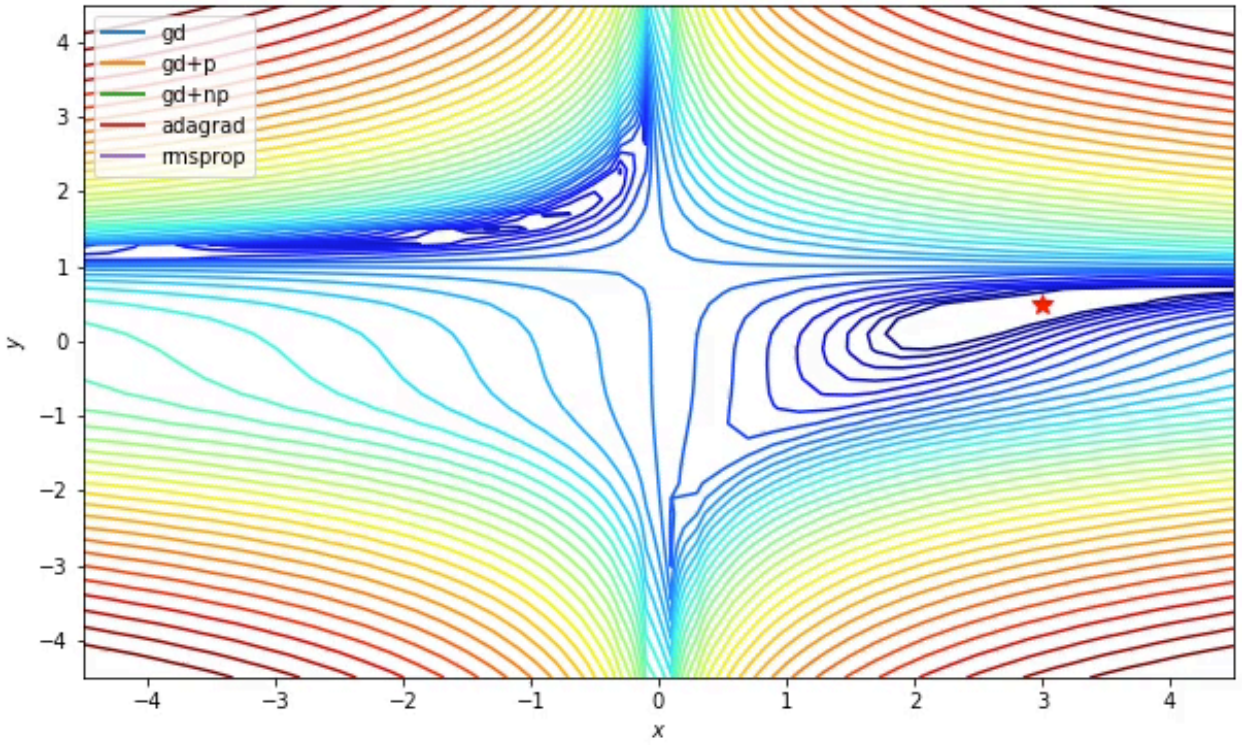
RMSProp (opt 2)

RMSProp proceeds to the global minimum, and in the video you can see it does so more quickly than Adagrad.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop.mp4`

RMSProp



RMSProp with momentum

RMSProp can be combined with momentum as follows.

Initialize $\mathbf{a} = 0$. Set α, β to be between 0 and 1. Set $\nu = 1e-7$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Accumulate gradient:

$$\mathbf{a} \leftarrow \beta \mathbf{a} + (1-\beta)\mathbf{g} \odot \mathbf{g}$$

- Momentum:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta + \mathbf{v}$$

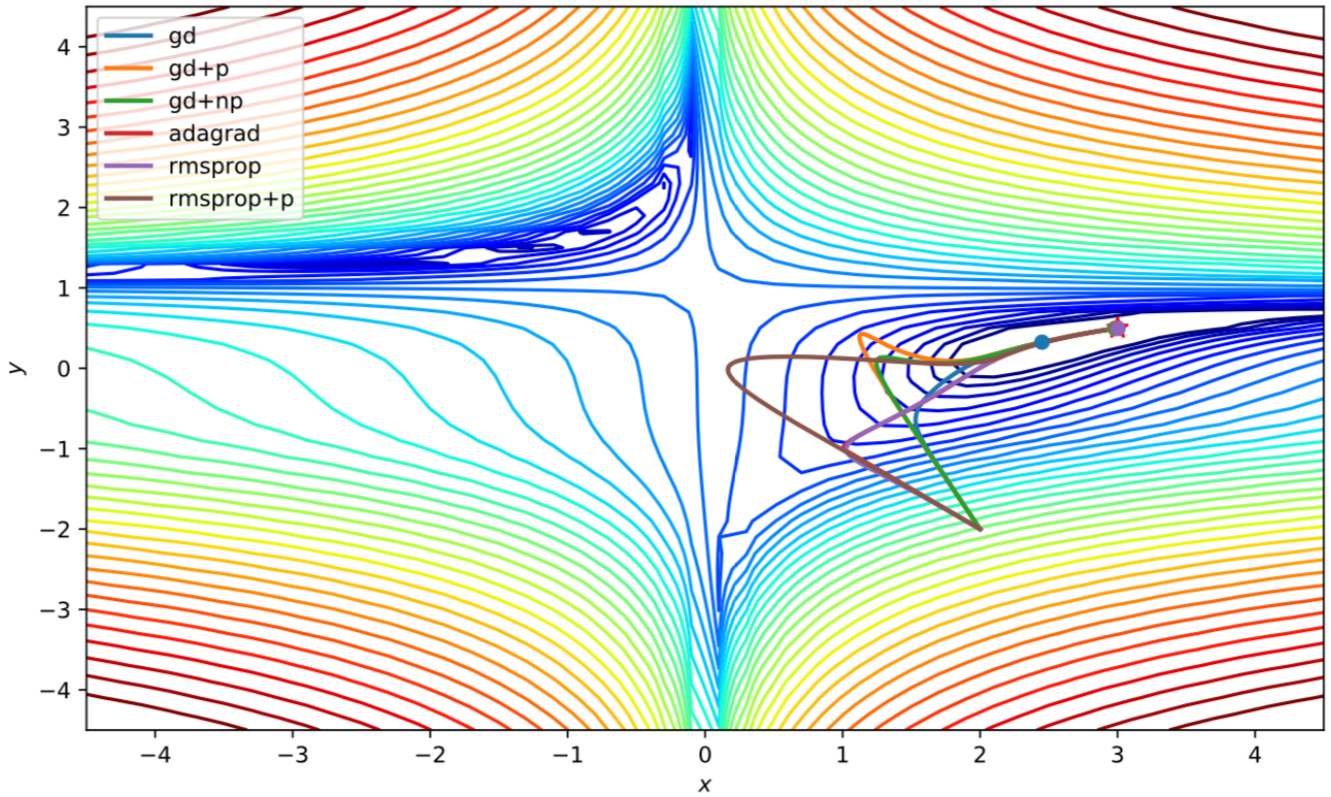
It is also possible to RMSProp with Nesterov momentum.

RMSProp + momentum

```
a = 0
p = 0
alpha = 0.9
beta = 0.99
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a = beta * a + (1-beta) * g * g
    p = alpha * p - eps * g / (np.sqrt(a) + nu)
    x += p
    last_diff = np.linalg.norm(x - path[-1])
```

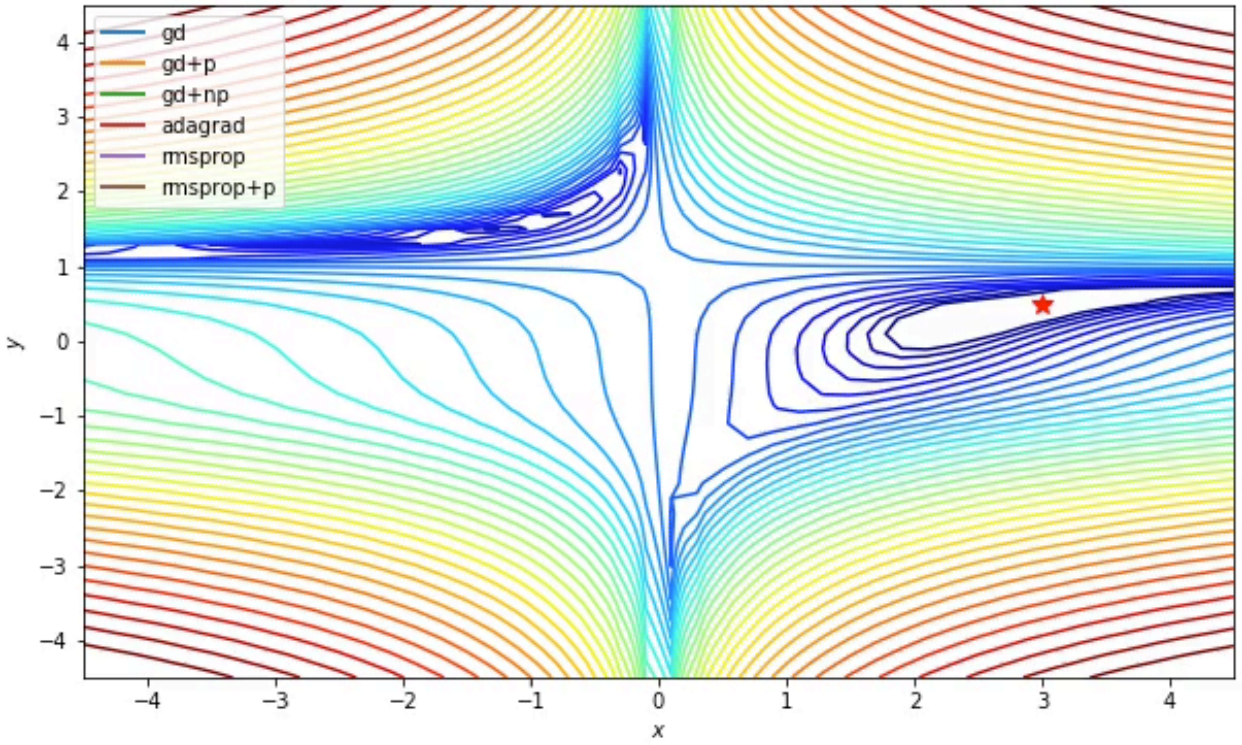
RMSProp with momentum (opt 1)

rmsprop+p denotes RMSProp with momentum. Though it makes a larger excursion out of the way, it gets to the optimum more quickly than all other optimizers. This is more apparent in the 2nd optimizer.



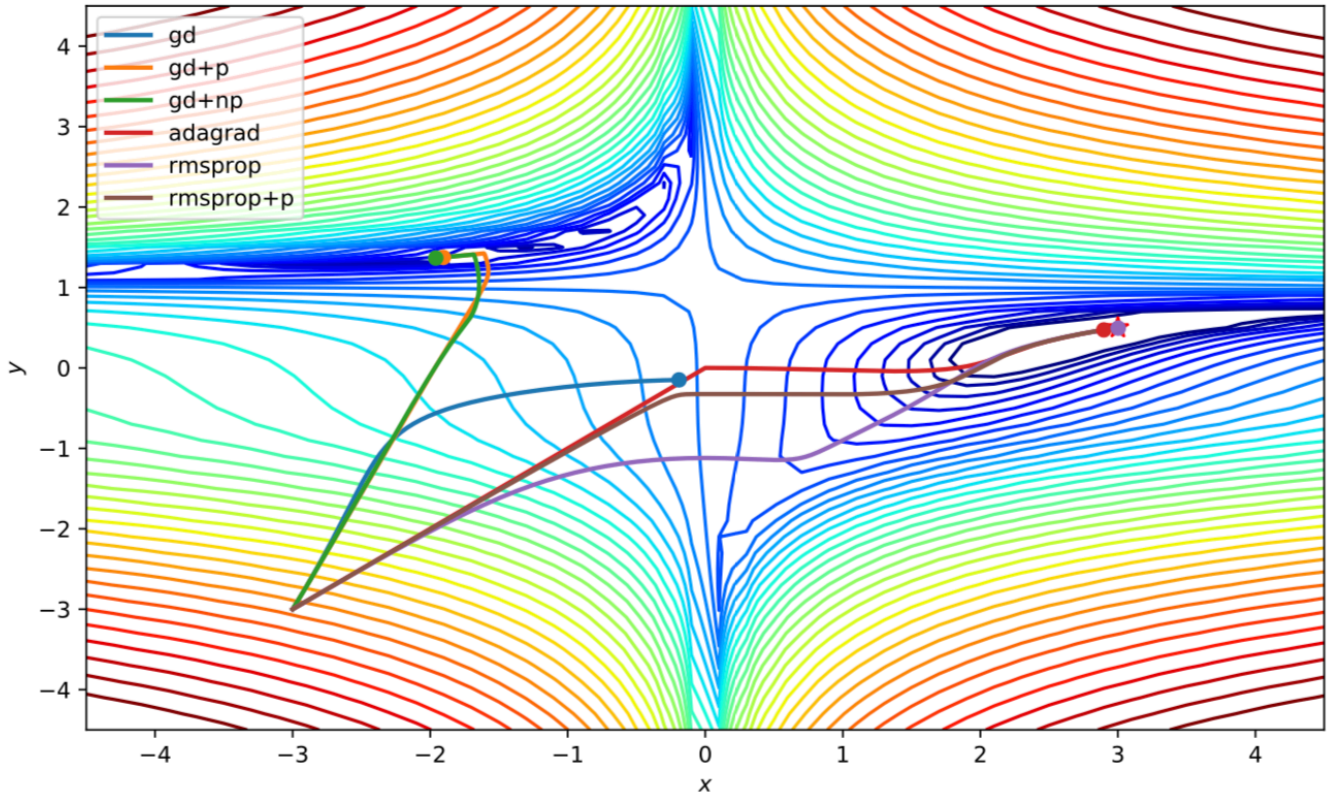
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p.mp4`

RMSProp + momentum



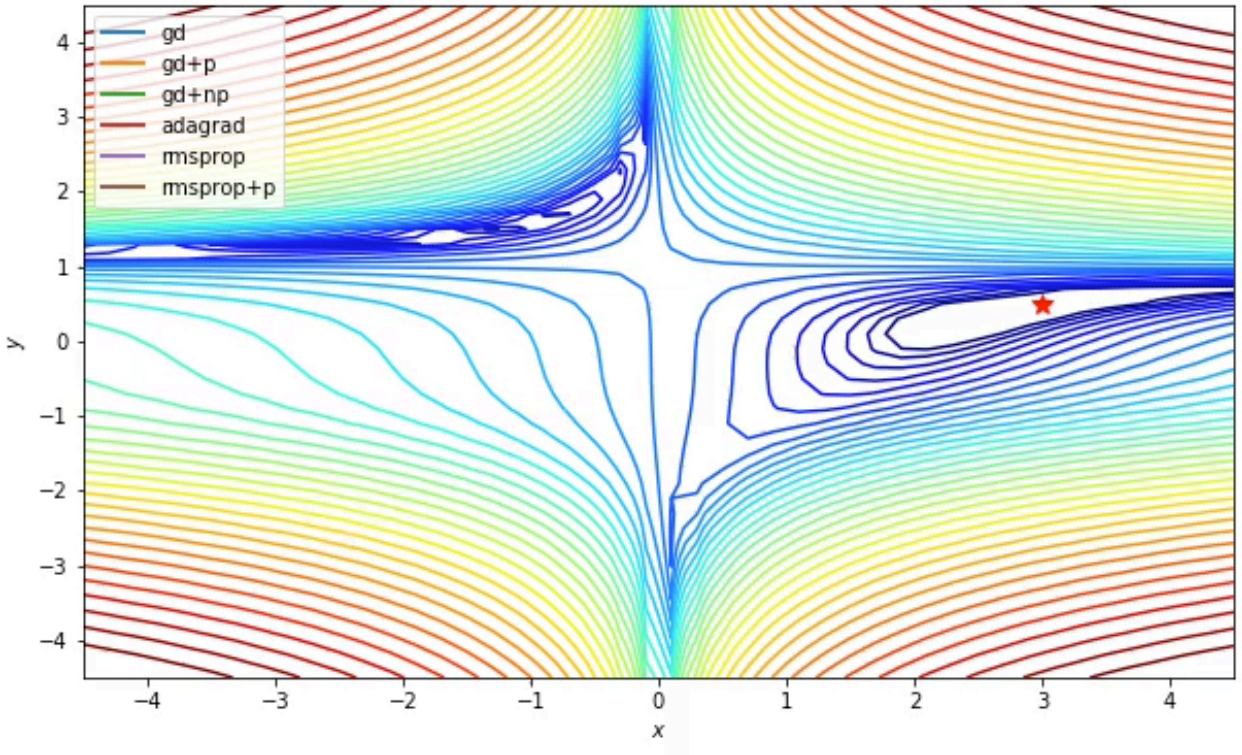
RMSProp (opt 2)

RMSProp proceeds to the global minimum, and in the video you can see it does so more quickly than Adagrad.



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop+p.mp4`

RMSProp + momentum



Adam

Adaptive moments without bias correction

The adaptive moments optimizer (Adam) is one of the most commonly used (and robust to e.g., hyperparameter choice) optimizers. Adam is composed of a momentum-like step, followed by an Adagrad/RMSProp-like step. For intuition, we first present Adam without a bias correction step.

Adam with no bias correction

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set β_1 and β_2 to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize ν to be sufficiently small. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{v}$$

Adaptive moments (Adam)

Adam incorporates a bias correction on the moments. The intuition for the bias correction is to account for initialization; these bias corrections amplify the second moments, so that extremely large steps are not taken at the start of the optimization.

Adam (cont.)

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set β_1 and β_2 to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize ν to be sufficiently small. Initialize $t = 0$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Time update: $t \leftarrow t + 1$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

$\mathbb{E}[g]$

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

$\mathbb{E}[g^2]$

- Bias correction in moments:

$$\tilde{\mathbf{v}} = \frac{1}{1 - \beta_1^t}\mathbf{v}$$

$$\tilde{\mathbf{a}} = \frac{1}{1 - \beta_2^t}\mathbf{a}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\tilde{\mathbf{a}}} + \nu} \odot \tilde{\mathbf{v}}$$

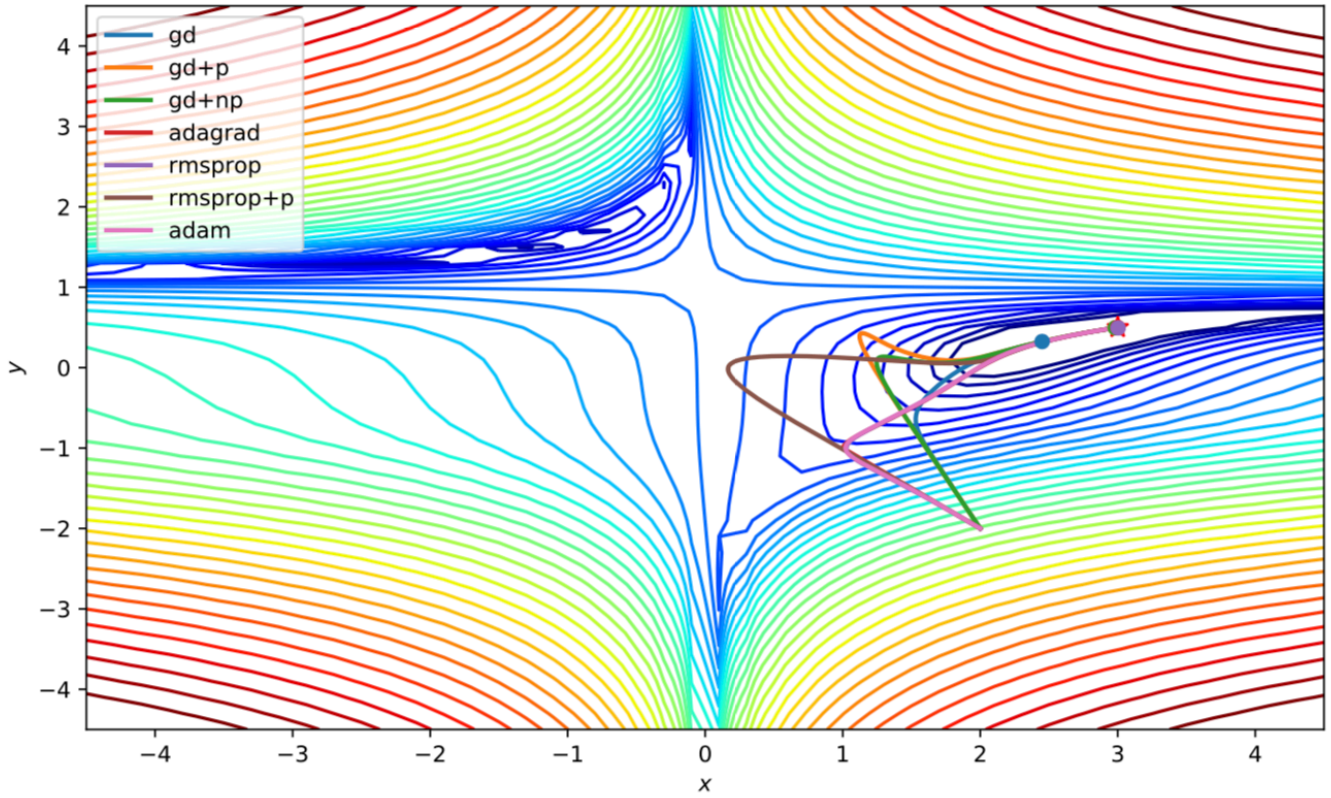
A resource on bias correction: https://www.coursera.org/lecture/deep-neural-network/bias-correction-in-exponentially-weighted-averages-XjuhD

Adam

```
a = 0
p = 0
beta1 = 0.9
beta2 = 0.99
nu = 1e-7
t = 0
while last_diff > tol:
    cost, g = func(x)
    t += 1
    p = beta1 * p + (1-beta1) * g
    a = beta2 * a + (1-beta2) * g * g
    p_u = p / (1 - beta1**t)
    a_u = a / (1 - beta2**t)
    x -= eps * p_u / (np.sqrt(a_u) + nu)
    last_diff = np.linalg.norm(x - path[-1])
```

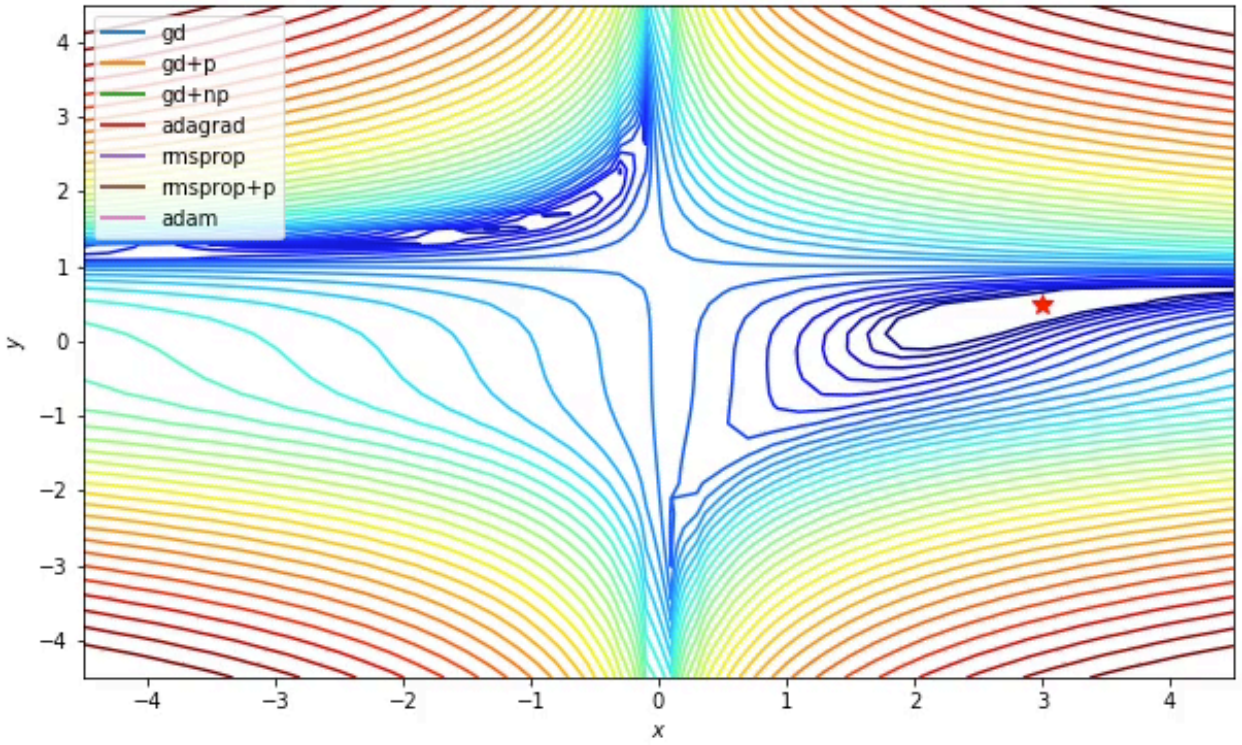
Adam (opt 1)

In both example optimizations, Adam is just slightly slower than RMSProp.

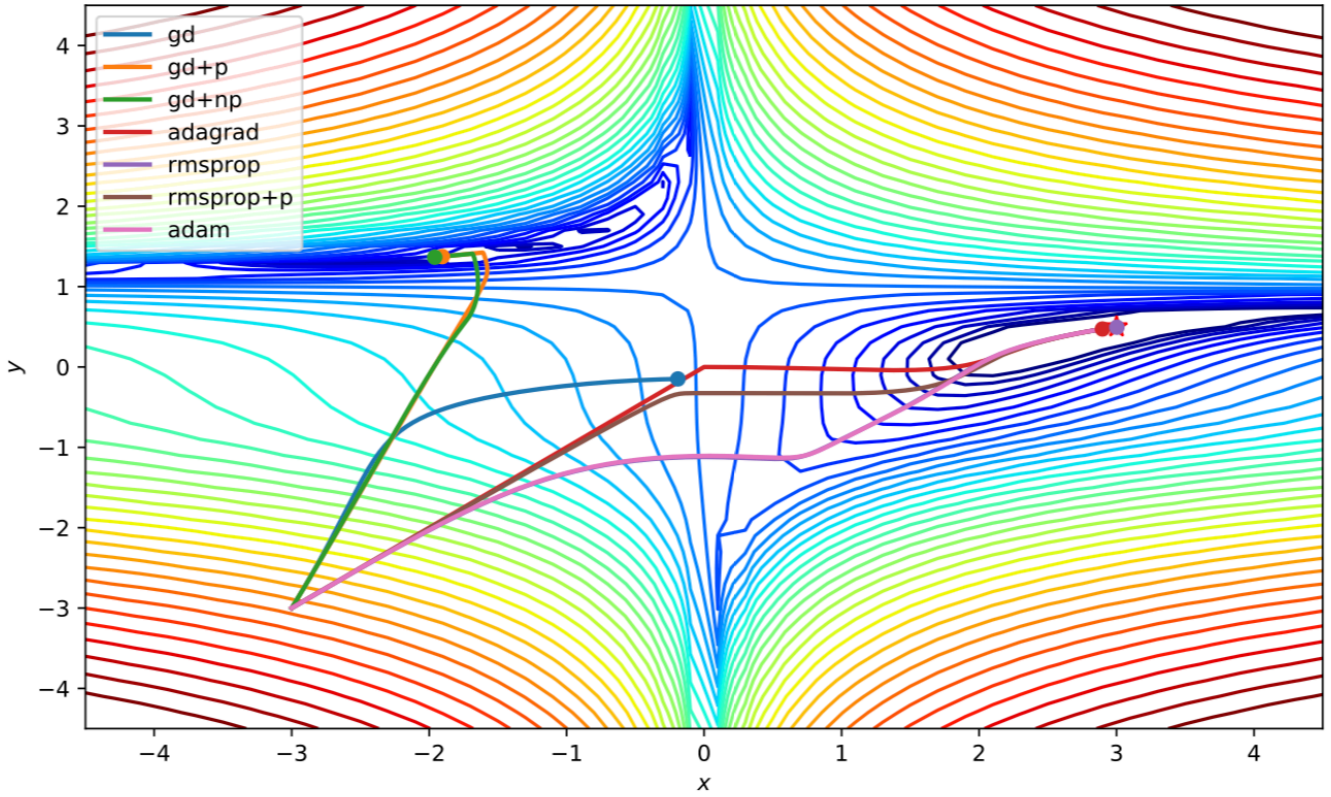


Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p_adam.mp4`

Adam

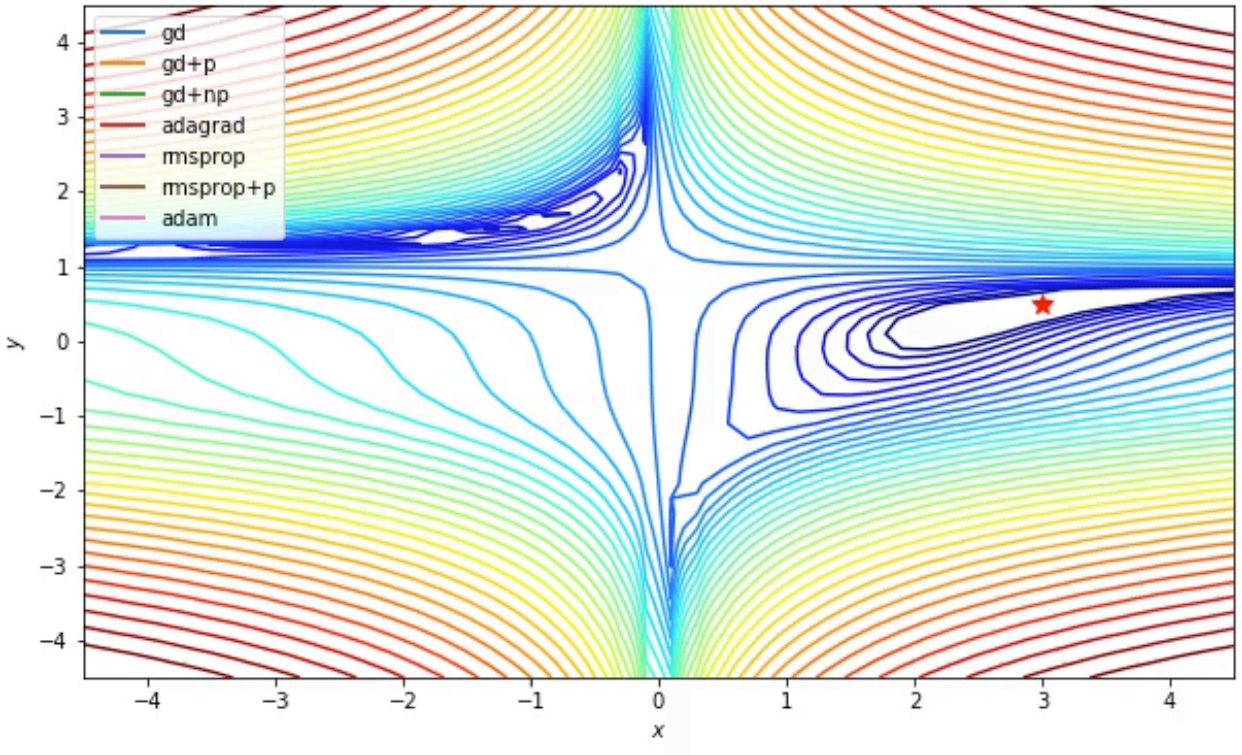

gd
gd+p
gd+np
adagrad
rmsprop
rmsprop+p
adam
4
3
2
1
0
−1
−2
−3
−4
y
−4
−3
−2
−1
0
1
2
3
4
x

Adam (opt 2)



Video: `http://seas.ucla.edu/~kao/opt_anim/2gd_gd+p_gd+np_adagrad_rmsprop+p_adam.mp4`

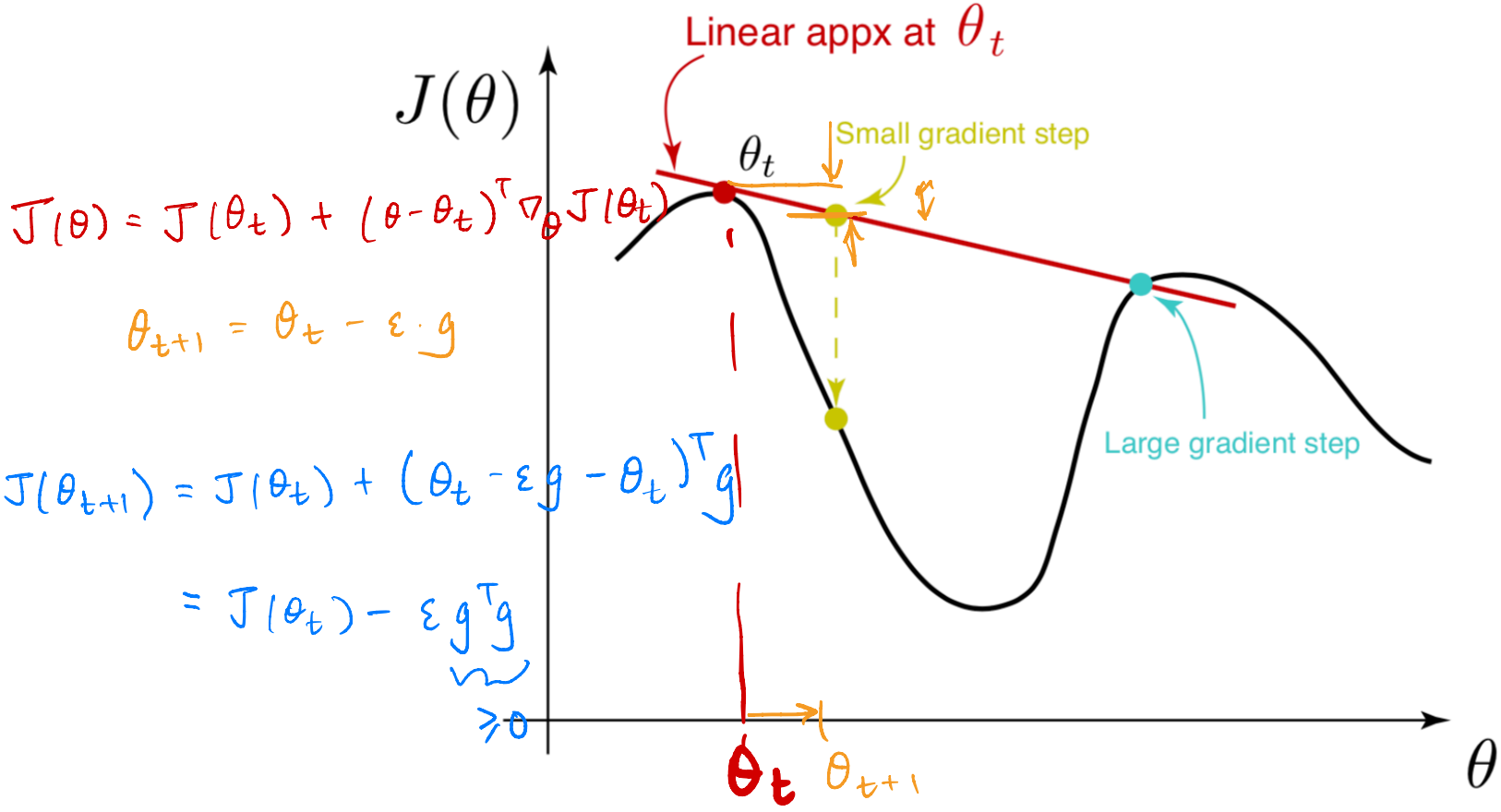
Adam


gd
gd+p
gd+np
adagrad
rmsprop
rmsprop+p
adam
y
x
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

First order vs second order methods

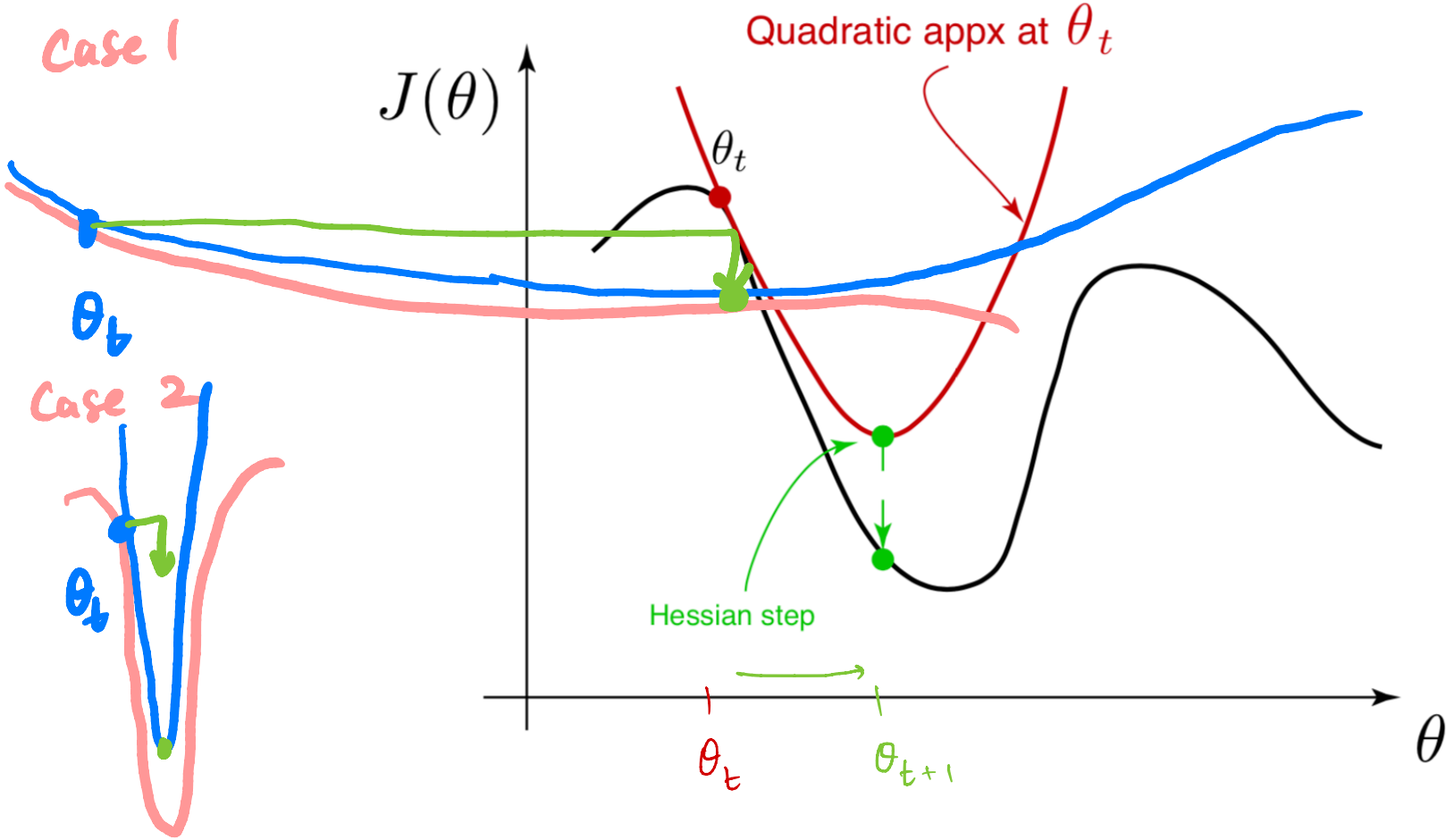
SGD, and optimizers used to augment the learning rate (Adagrad, RMSProp, and Adam), are all *first order* methods and have the learning rate ε as a hyperparameter.

- First-order refers to the fact that we only use the first derivative, i.e., the gradient, and take linear steps along the gradient.
- The following picture of a first order method is appropriate.



First order vs second order methods (cont)

It is possible to also use the *curvature* of the cost function to know how to take steps. These are called second-order methods, because they use the second derivative (or Hessian) to assess the curvature and thus take appropriate sized steps in each dimension. See following picture for intuition:



Newton's method

NOT TESTED

Second order optimization

The most widely used second order method is Newton's method. To arrive at it, consider the Taylor series expansion of $J(\theta)$ around θ_0 up to the second order terms, i.e.,

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \mathbf{H}(\theta - \theta_0)$$

If this were just a second order function, we could minimize it by taking its derivative and setting it to zero:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta J(\theta_0) + \mathbf{H}(\theta - \theta_0) \\ &= \mathbf{0} \end{aligned}$$

This results in the Newton step,

$$\theta = \theta_0 - \mathbf{H}^{-1} \nabla_\theta J(\theta_0)$$

If the function is quadratic, this step takes us to the minimum. If not, this approximates the function as quadratic at θ_0 and goes to the minimum of the quadratic approximation.

Does this form of step make intuitive sense?

Newton's method

Newton's method, by using the curvature information in the Hessian, does not require a learning rate.

Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Compute Hessian: $\mathbf{H}$
- Gradient step:

$$\theta \leftarrow \theta - \mathbf{H}^{-1}\mathbf{g}$$

1. Memory: storing H is expensive

$n = 10^6$:

grad: 3.8 MBytes

Hessian: 3.6 TB

2. Inverting the Hessian: $O(n^3)$

3. Hessian typically requires very large batch sizes

NOT TESTED

Newton's method (cont.)

A few notes about Newton's method:

- When the Hessian has negative eigenvalues, then steps along the corresponding eigenvectors are gradient ascent steps. To counteract this, it is possible to regularize the Hessian, so that the updates become:

$$\theta \leftarrow \theta - (\mathbf{H} + \alpha \mathbf{I})^{-1} \nabla_\theta J(\theta_0)$$

  As α becomes larger, this turns into first order gradient descent with learning rate $1/\alpha$.

- Newton's method requires, at every iteration, calculating and then inverting the Hessian. If the network has N parameters, then inverting the Hessian is $\mathcal{O}(N^3)$. This renders Newton's method impractical for many types of deep neural networks.

Quasi-Newton methods

To get around the problem of having to compute and invert the Hessian, quasi-Newton methods are often used. Amongst the most well-known is the BFGS (Broyden Fletcher Goldfarb Shanno) update.

- The idea is that instead of computing and inverting the Hessian at each iteration, the inverse Hessian $\mathbf{H}_0^{-1}$ is initialized at some value, and it is recursively updated via:

$$\mathbf{H}_k^{-1} \leftarrow \left(\mathbf{I} - \frac{\mathbf{s}\mathbf{y}^T}{\mathbf{y}^T\mathbf{s}}\right)\mathbf{H}_{k-1}^{-1}\left(\mathbf{I} - \frac{\mathbf{y}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}\right) + \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}$$

  for

$$\mathbf{s} = \theta_k - \theta_{k-1} \quad \text{and} \quad \mathbf{y} = \nabla J(\theta_k) - \nabla J(\theta_{k-1})$$

- The proof of this result is beyond the scope of this class; if you'd like to learn more about this (and about optimization in general), consider taking ECE 236C.

Quasi-Newton methods (cont.)

- An important aspect of this update is that the inverse of any Hessian can be reconstructed from the sequence of $\mathbf{s}_k$, $\mathbf{y}_k$, and the initial $\mathbf{H}_0^{-1}$. Thus a recurrence relationship can be written to calculate $\mathbf{H}_k^{-1}\mathbf{x}$ without explicitly having to calculate $\mathbf{H}_k^{-1}$. However, it does require iterating over $i = 0, \ldots, k$ examples.
- A way around this is to use limited memory BFGS (L-BFGS), where you calculate the inverse Hessian using just the last m examples assuming $\mathbf{H}_{k-m}^{-1}$ is some $\mathbf{H}_0^{-1}$ (e.g., it could be the identity matrix).
- Quasi-Newton methods usually require a full batch (or very large minibatches) since errors in estimating the inverse Hessian can result in poor steps.

Conjugate gradient methods

CG methods are also beyond the scope of this class, but we bring it up here in case helpful to look into further. Again, ECE 236C is recommended if you'd like to learn more about these techniques.

- CG methods find search directions that are *conjugate* with respect to the Hessian, i.e., that $\mathbf{g}_k^T \mathbf{H} g_{k-1} = 0$.
- It turns out that these derivatives can be calculated iteratively through a recurrence relation.
- Implementations of "Hessian-free" CG methods have been demonstrated to converge well (e.g., Martens et al., ICML 2011).

Challenges in gradient descent

- **Exploding gradients.**
  Sometimes the cost function can have "cliffs" whereby small changes in the parameters can drastically change the cost function. (This usually happens if parameters are repeatedly multiplied together, as in recurrent neural networks.) Because the gradient at a cliff is large, an update can result in going to a completely different parameter space. This can be ameliorated via gradient clipping, which upper bounds the maximum gradient norm.

Challenges in gradient descent

- **Vanishing gradients.**
  Like in exploding gradients, repeated multiplication of a matrix $\mathbf{W}$ can cause vanishing gradients. Say that each time step can be thought of as a layer of a feedforward network where each layer has connectivity $\mathbf{W}$ to the next layer. By layer t, there have been $\mathbf{W}^t$ multiplications. If $\mathbf{W} = \mathbf{U}\Lambda\mathbf{U}^{-1}$ is its eigendecomposition, then $\mathbf{W}^t = \mathbf{U}\Lambda^t\mathbf{U}^{-1}$, and hence the gradient along eigenvector $\mathbf{u}_i$ is shrunk (or grown) by the factor λ_i^t. Architectural decisions, as well as appropriate regularization, can deal with vanishing gradients.