# Optimization for neural networks

We previously introduced the principle of gradient descent. Now we will discuss specific modifications we make that are useful for deep learning (and potentially other areas).

- Stochastic gradient descent
- Momentum
- Optimizers with adaptive gradients
- Second order techniques
- Challenges in gradient descent

# Batch vs minibatch

Consider a maximum-likelihood classifier, in which we are given examples $(\mathbf{x}^{(i)}, y^{(i)})$ for $i = 1, \ldots, m$.

We would like to use a model, given by $p_{\text{model}}$, and find the parameters, $\theta$, that maximize the likelihood of having seen the data. Our cost function is:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)$$

and its gradient is:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\
&= \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta) \\
&\approx \mathbb{E}\left[\nabla_\theta \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)\right]
\end{aligned}$$

# Batch vs minibatch (cont)

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.

# A few notes about batches

- Larger batch sizes give a more accurate estimate of the gradient (sublinear).
- When doing second order estimation, much larger batch sizes are required to achieve a more accurate Hessian, $\mathbf{H}$. A step by $\mathbf{H}^{-1}\nabla_\theta J(\theta)$ may amplify estimation noise in the gradient, especially if $\mathbf{H}$ is poorly conditioned.
- In practice, it is good to shuffle examples before computing a minibatch to reduce dataset correlations.

# Stochastic gradient descent

Stochastic gradient descent proceeds as follows.

Set a learning rate $\varepsilon$ and an initial parameter setting $\theta$. 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:

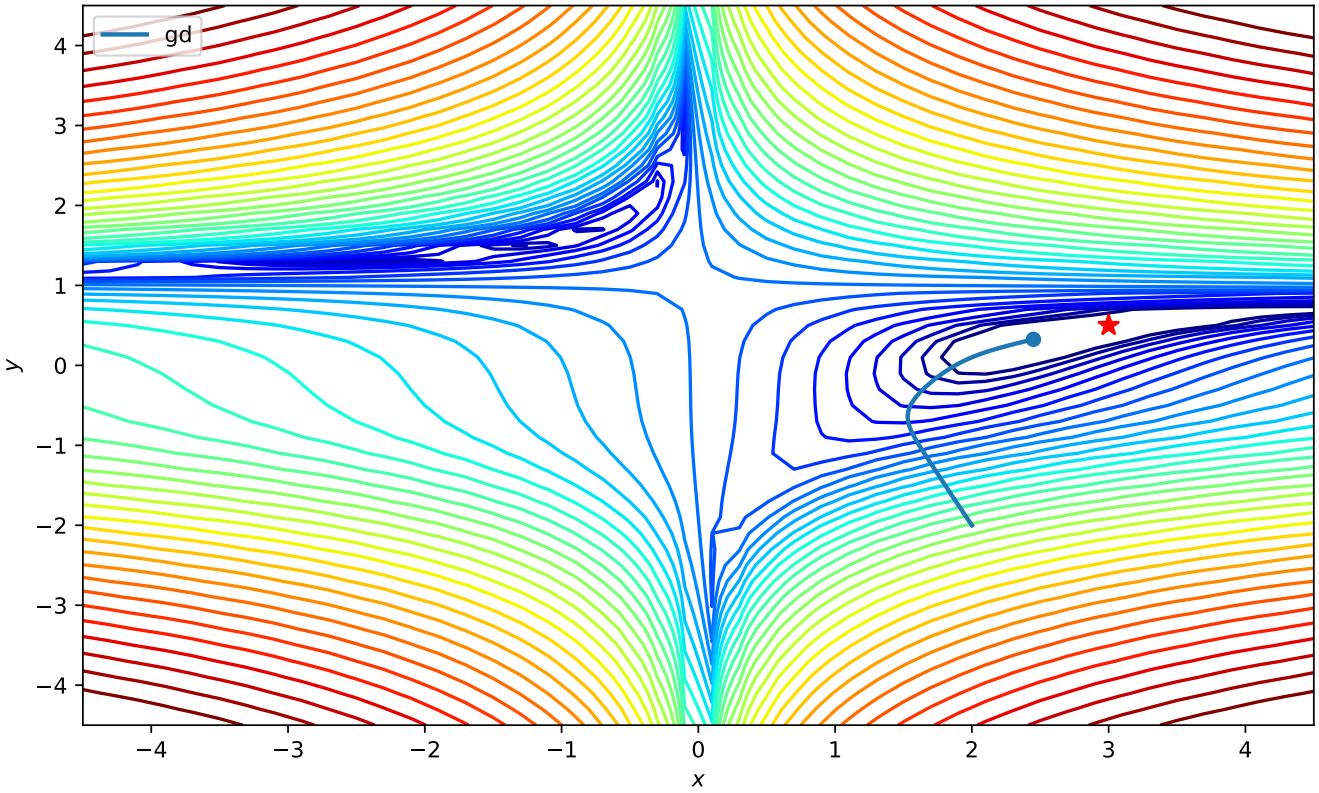
$$\theta \leftarrow \theta - \varepsilon \mathbf{g}$$

A common practice is to apply a decay rule to the learning rate until iteration $\tau$, via:

$$\varepsilon_k = \frac{\tau - k}{\tau} \varepsilon_0 + \frac{k}{\tau} \varepsilon_\tau$$

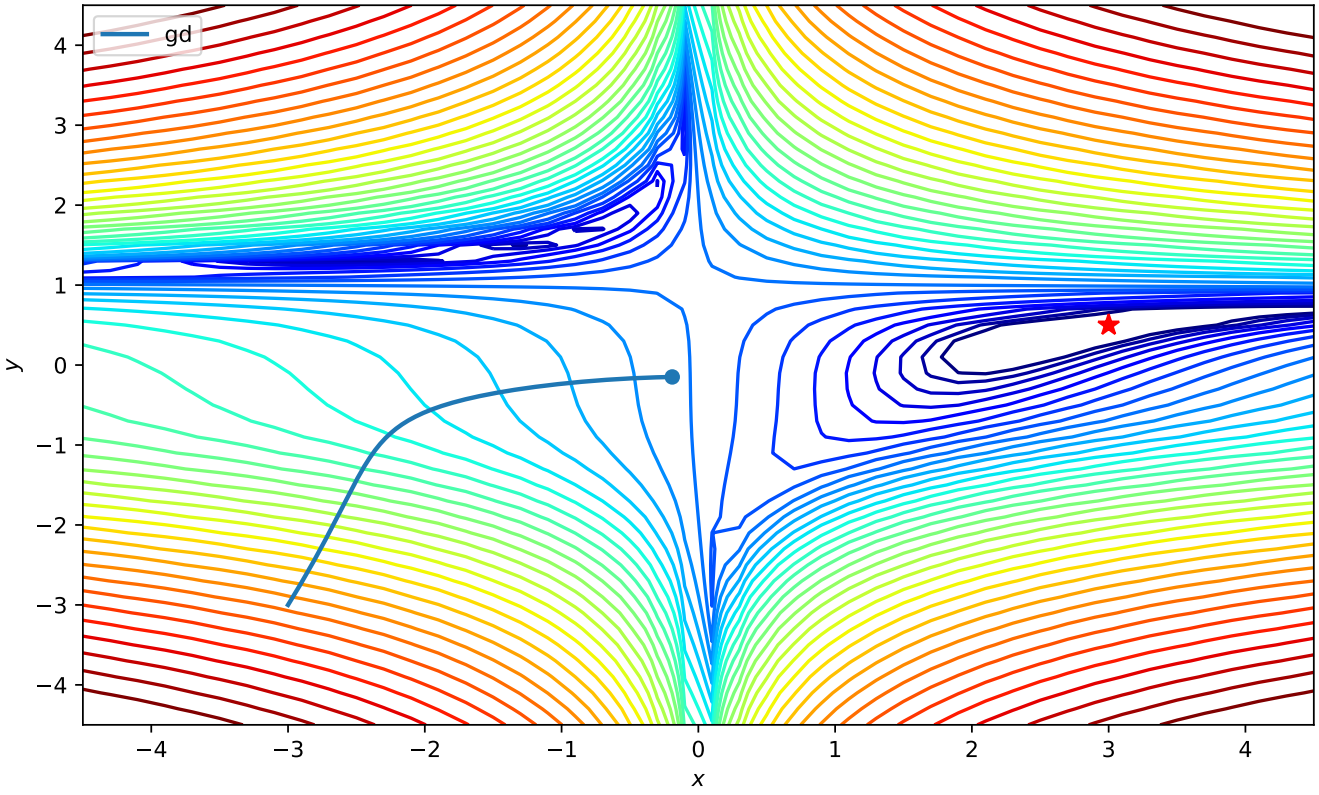
# Stochastic gradient descent (opt 1)

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 (opt 2)




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

# 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:

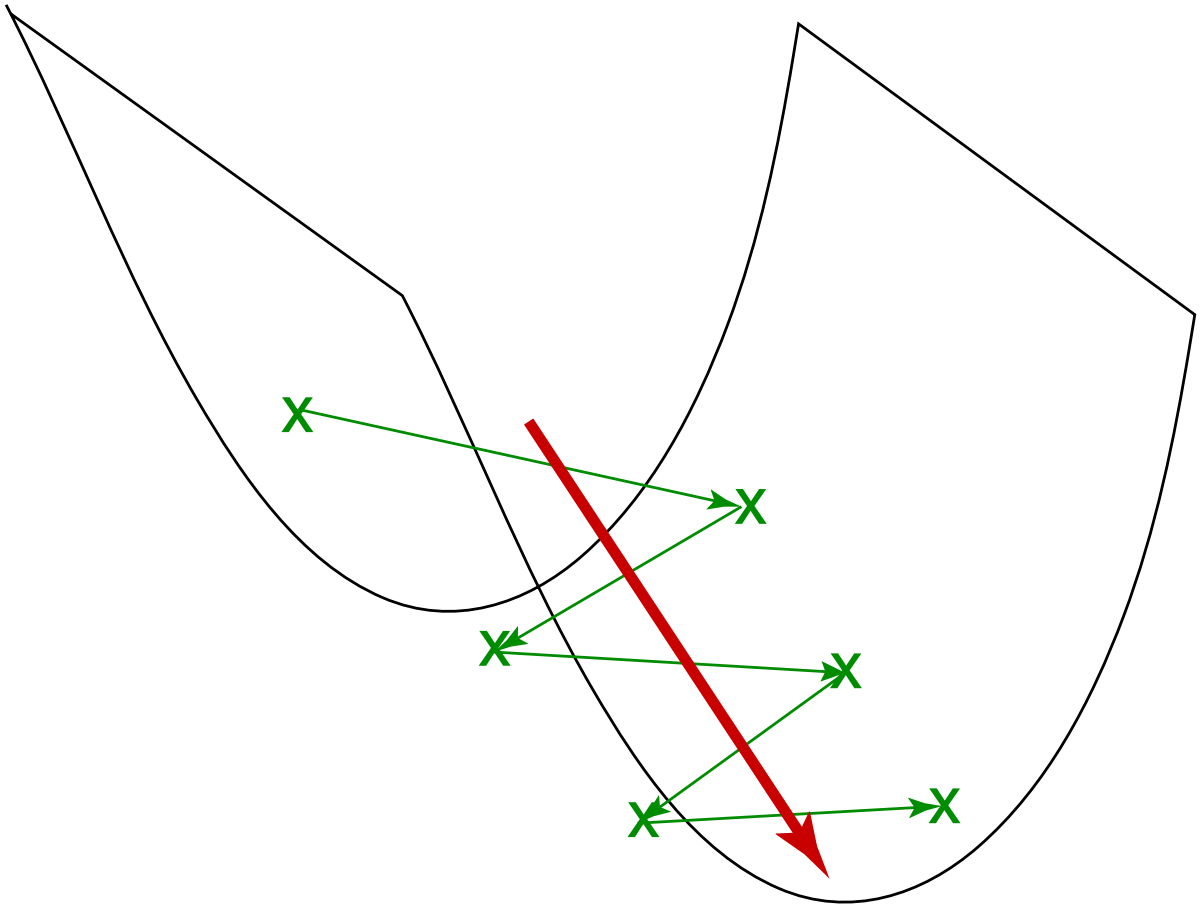
$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \mathbf{g}$$

- Gradient step:

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

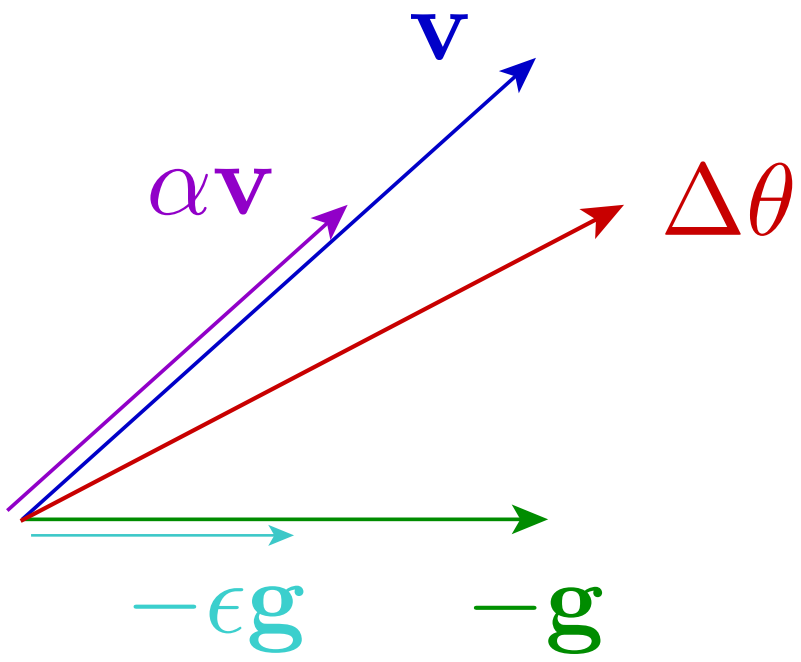
# Momentum (cont.)

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.



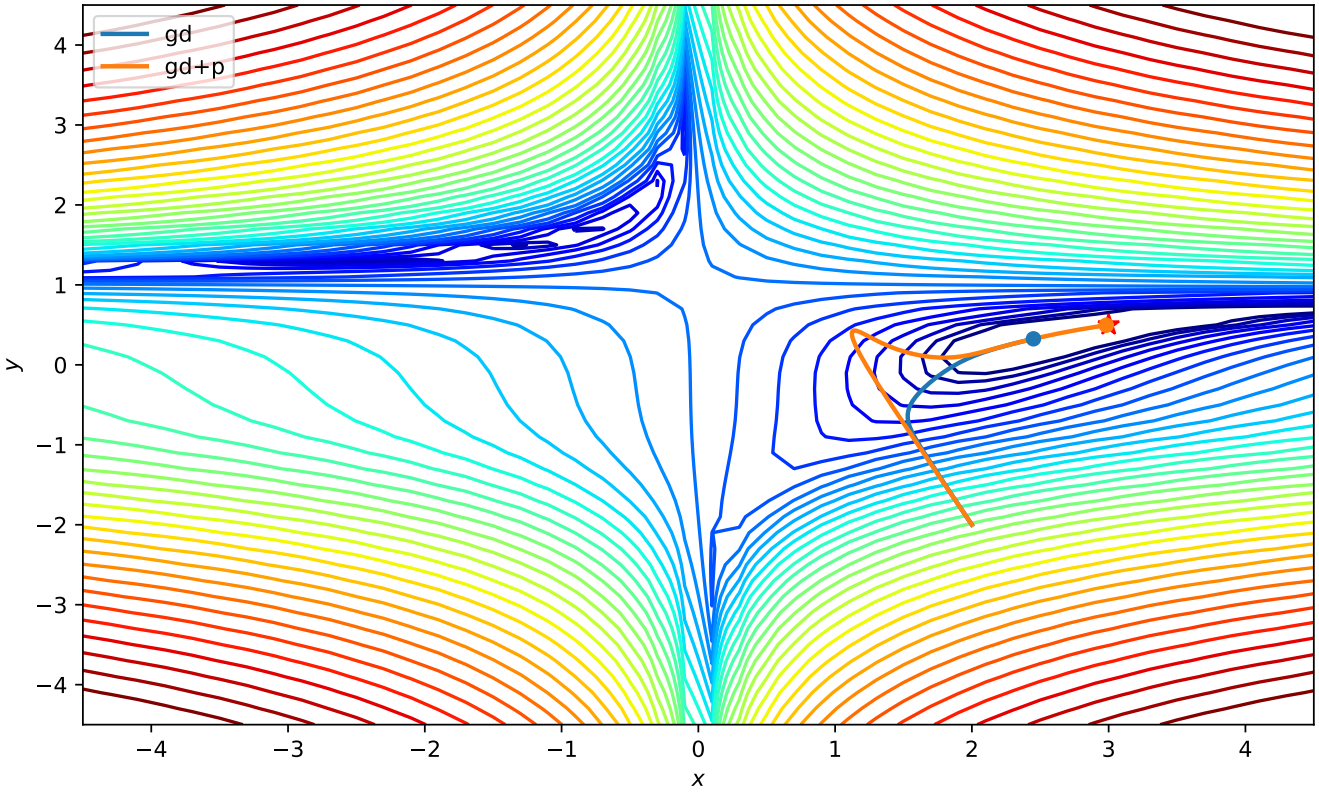
# Momentum (cont.)

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:



# Momentum (opt 1)

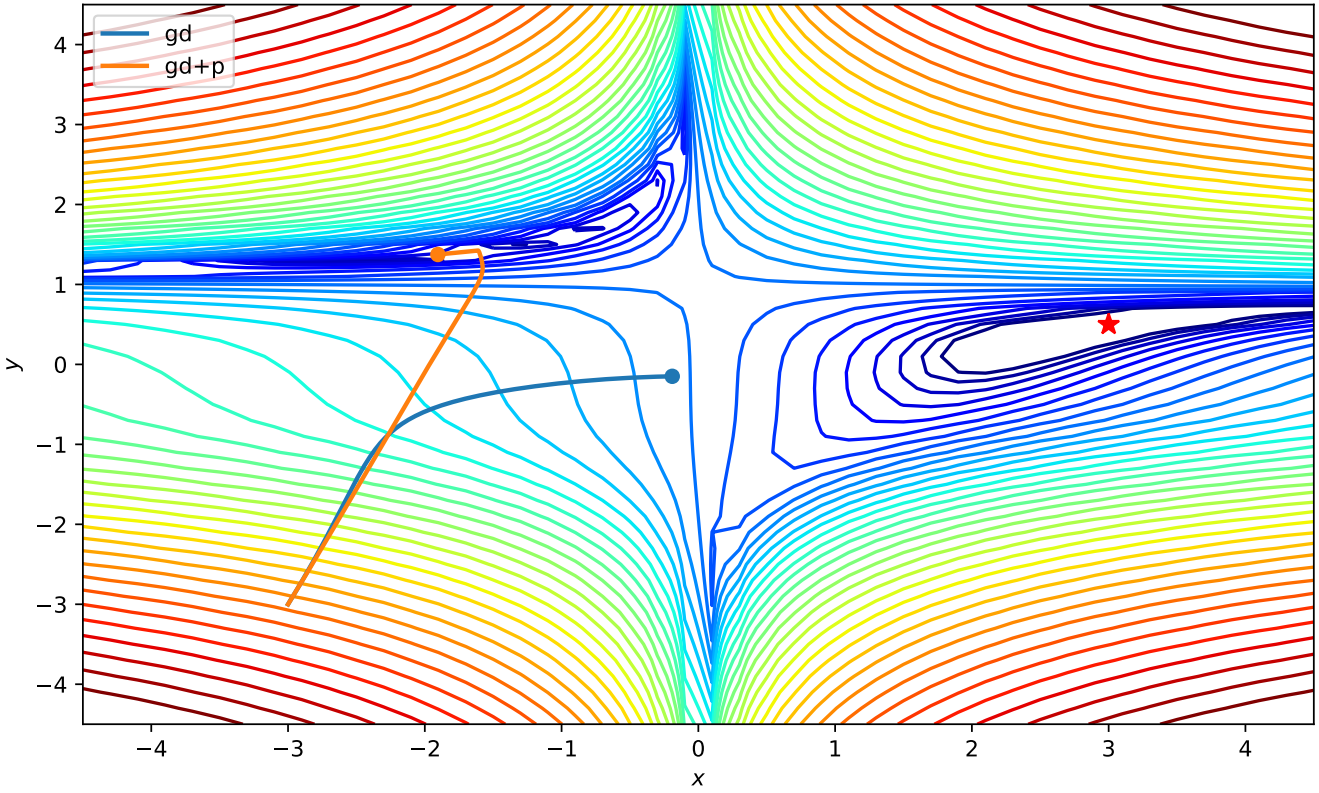
gd+p denotes gradient descent with momentum.




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

# 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`

# 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:

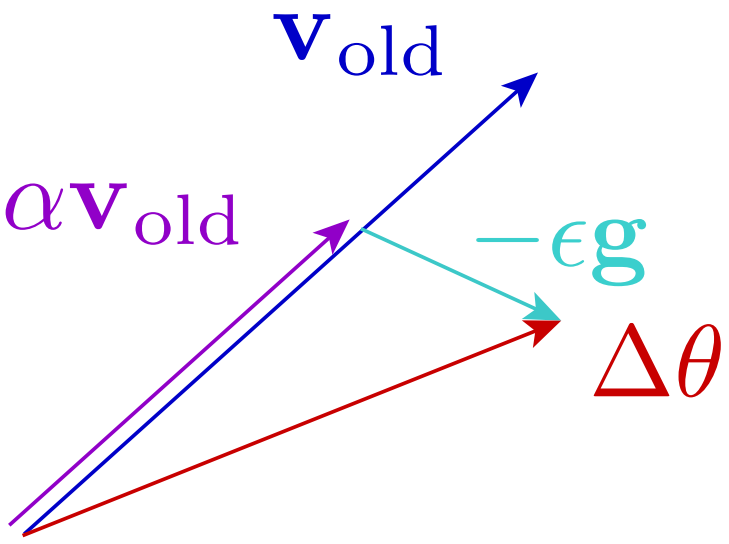
- Update:
$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \nabla_\theta J(\theta + \alpha \mathbf{v})$$
- Gradient step:
$$\theta \leftarrow \theta + \mathbf{v}$$

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:
$$\mathbf{v}_{\text{new}} = \alpha \mathbf{v}_{\text{old}} - \varepsilon \mathbf{g}$$
- Gradient step:
$$\theta_{\text{new}} = \theta_{\text{old}} + \mathbf{v}_{\text{new}} + \alpha(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{old}})$$
- Set $\mathbf{v}_{\text{old}} = \mathbf{v}_{\text{new}}$, $\theta_{\text{old}} = \theta_{\text{new}}$.

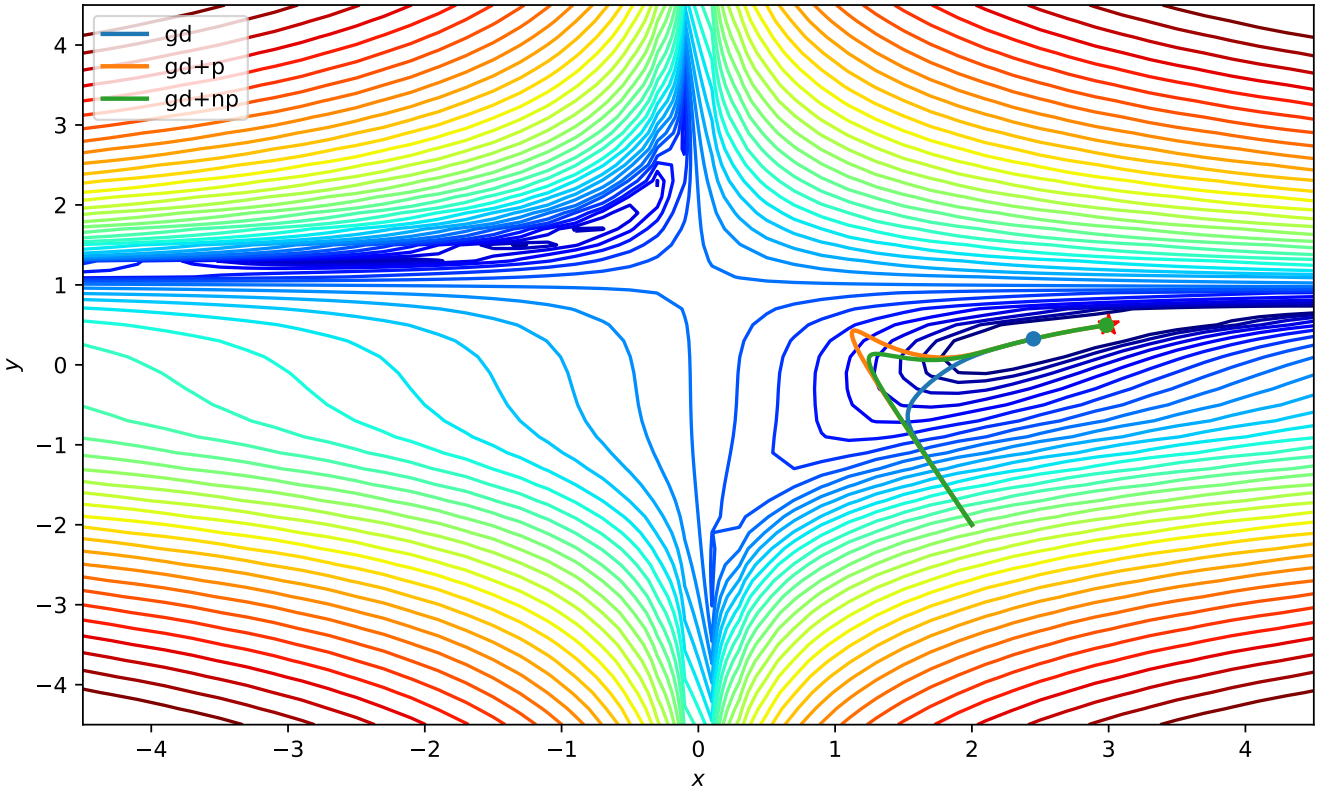
# Nesterov momentum (cont.)

The following image is appropriate for Nesterov momentum:



# Nesterov momentum (opt 1)

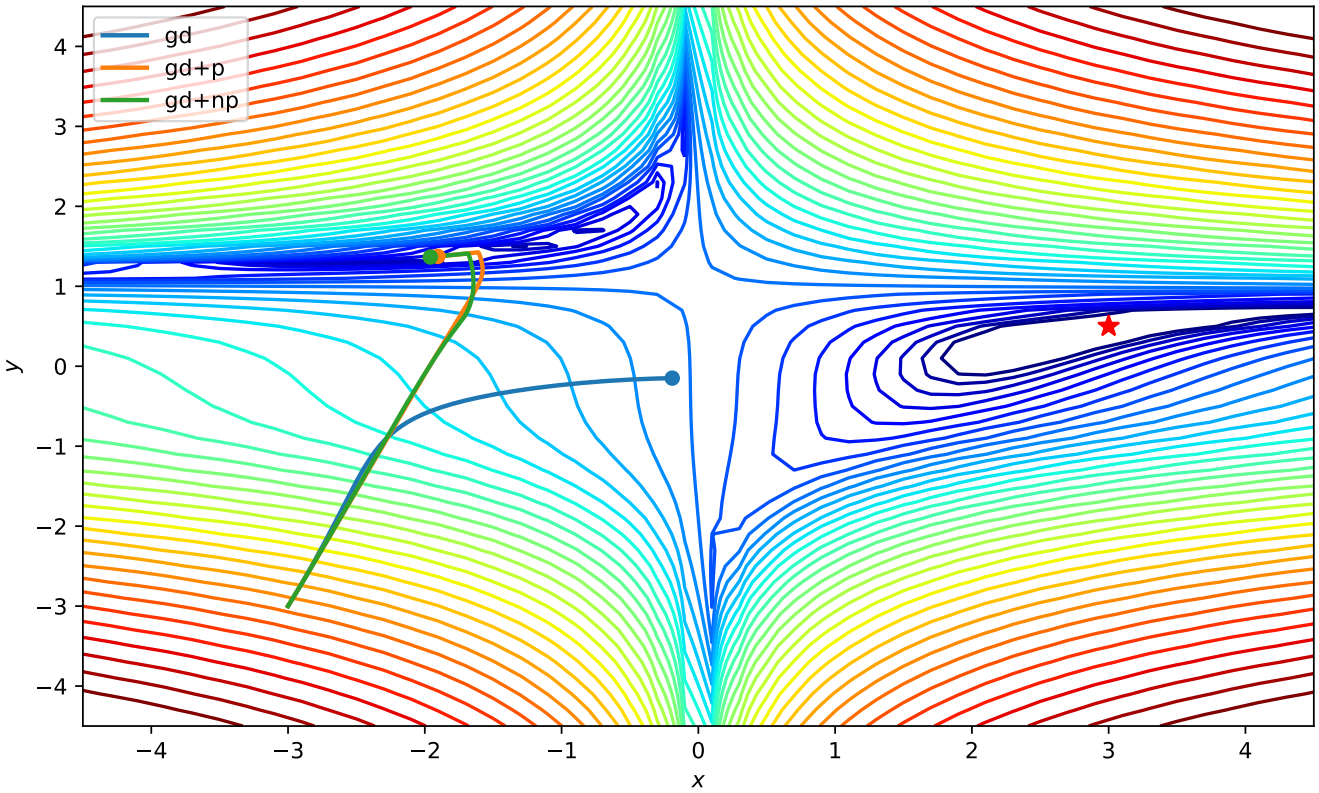
gd+np denotes gradient descent with Nesterov momentum.




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

# Nesterov momentum (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`

# Techniques to adapt the learning rate

Choosing $\varepsilon$ 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, $\varepsilon$ 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 $\nu$ 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:

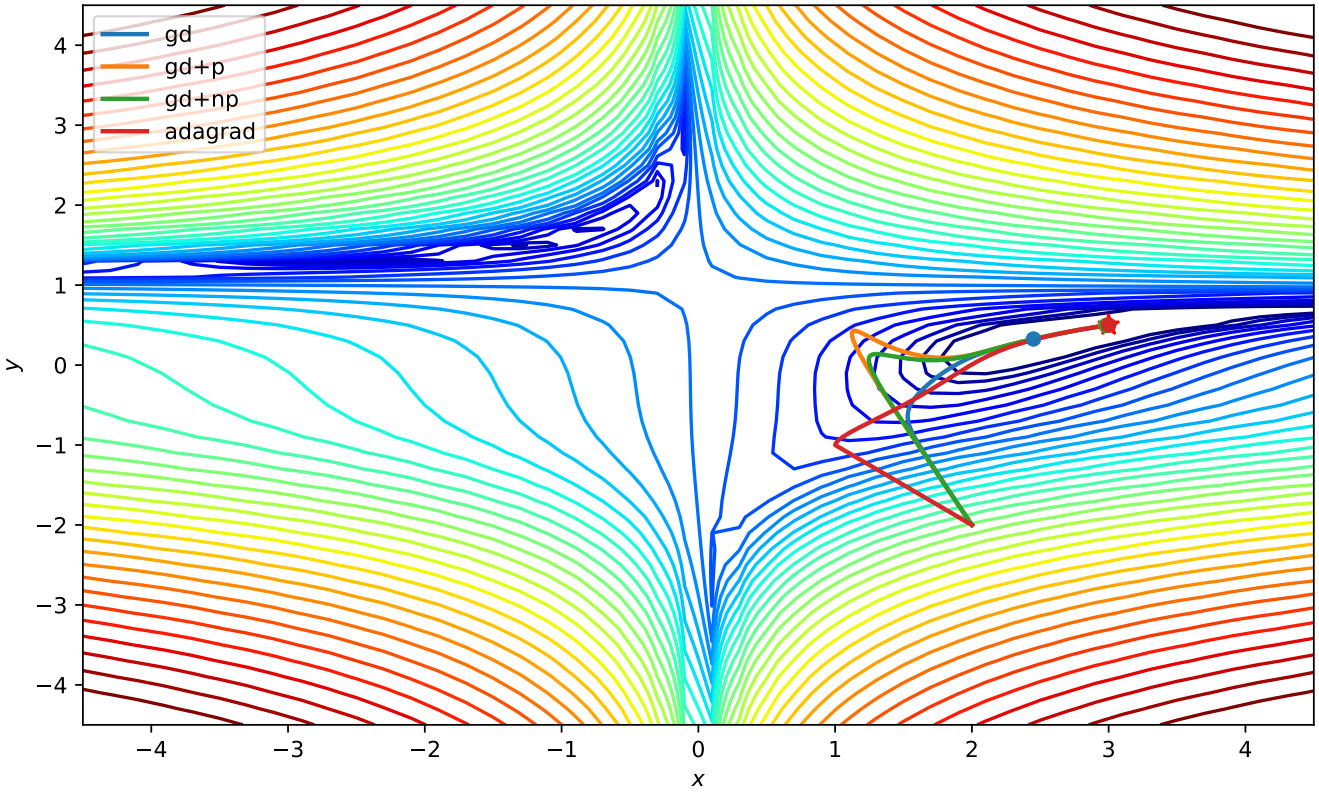
$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

- Gradient step:

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

A con of Adagrad is that eventually the step size will become sufficiently small when $\mathbf{a}$ grows too large such that no more meaningful learning occurs. This motivates the next optimizer, which shrinks $\mathbf{a}$ over time.

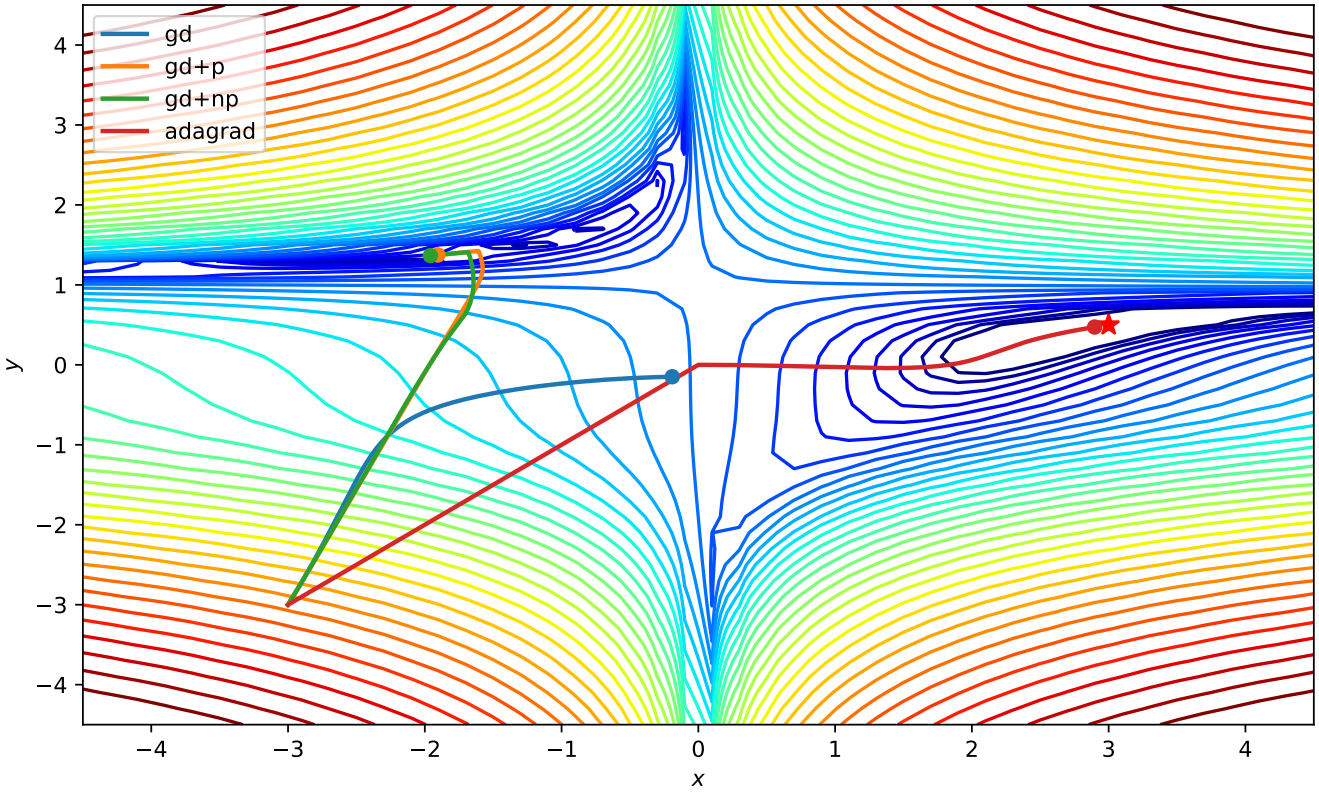
# Adagrad (opt 1)




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

# Adagrad (opt 2)

Adagrad proceeds to the global minimum.




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

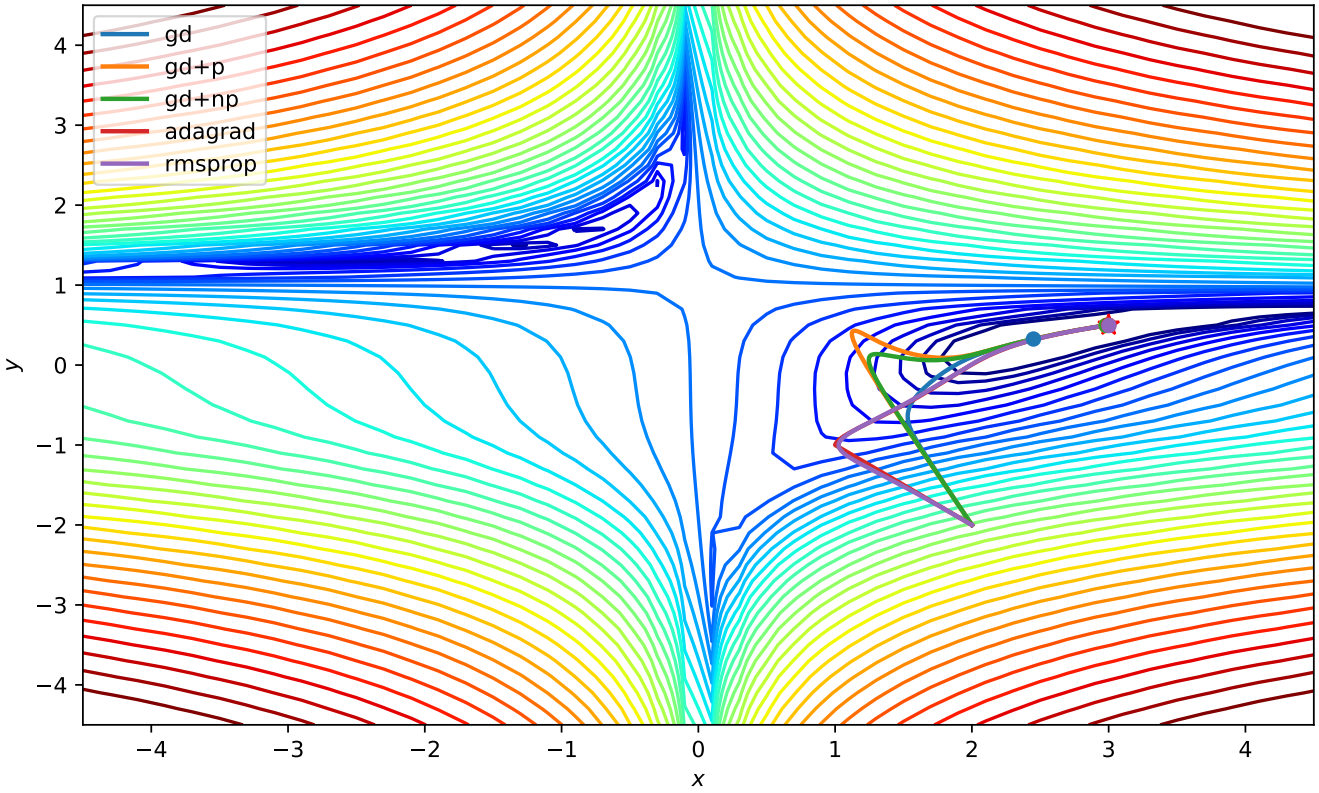
# RMSProp

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

Initialize $\mathbf{a} = 0$ and set $\nu$ to be sufficiently small. Set $\beta$ 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:
$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

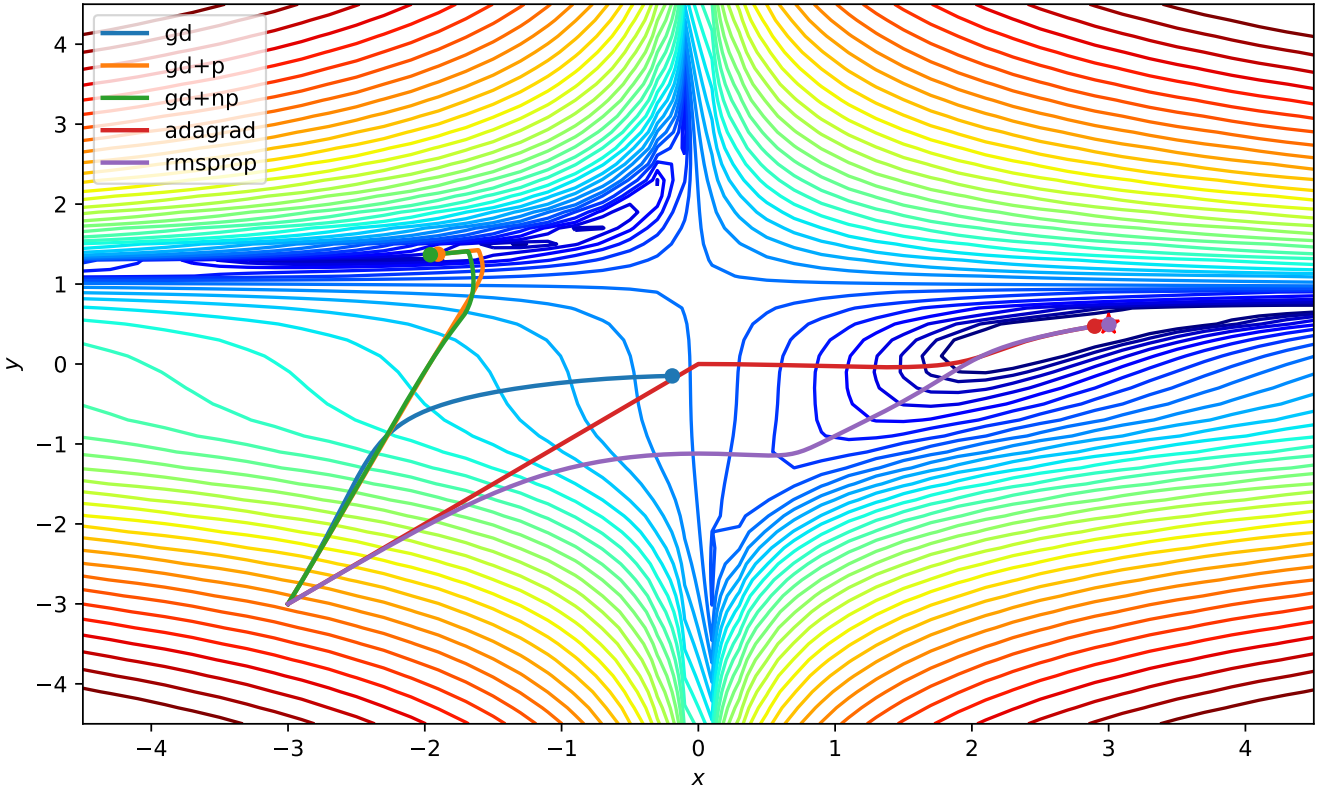
# RMSProp (opt 1)




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

# 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 with momentum

RMSProp can be combined with Nesterov momentum as follows.

Initialize $\mathbf{a} = 0$. Set $\alpha, \beta$ to be between $0$ and $1$. 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:

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

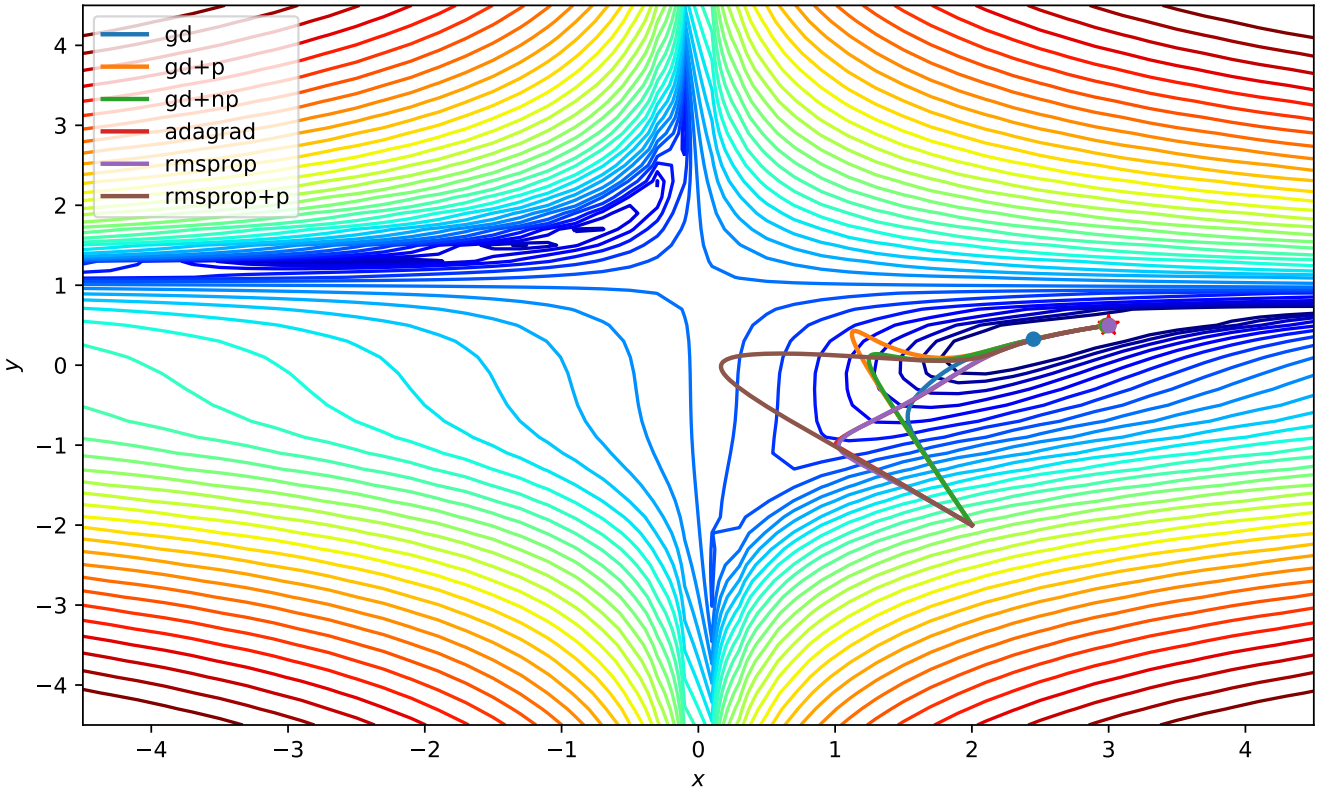
- Gradient step:

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

It is also possible to RMSProp this with Nesterov momentum.

# RMSProp with momentum (opt 1)

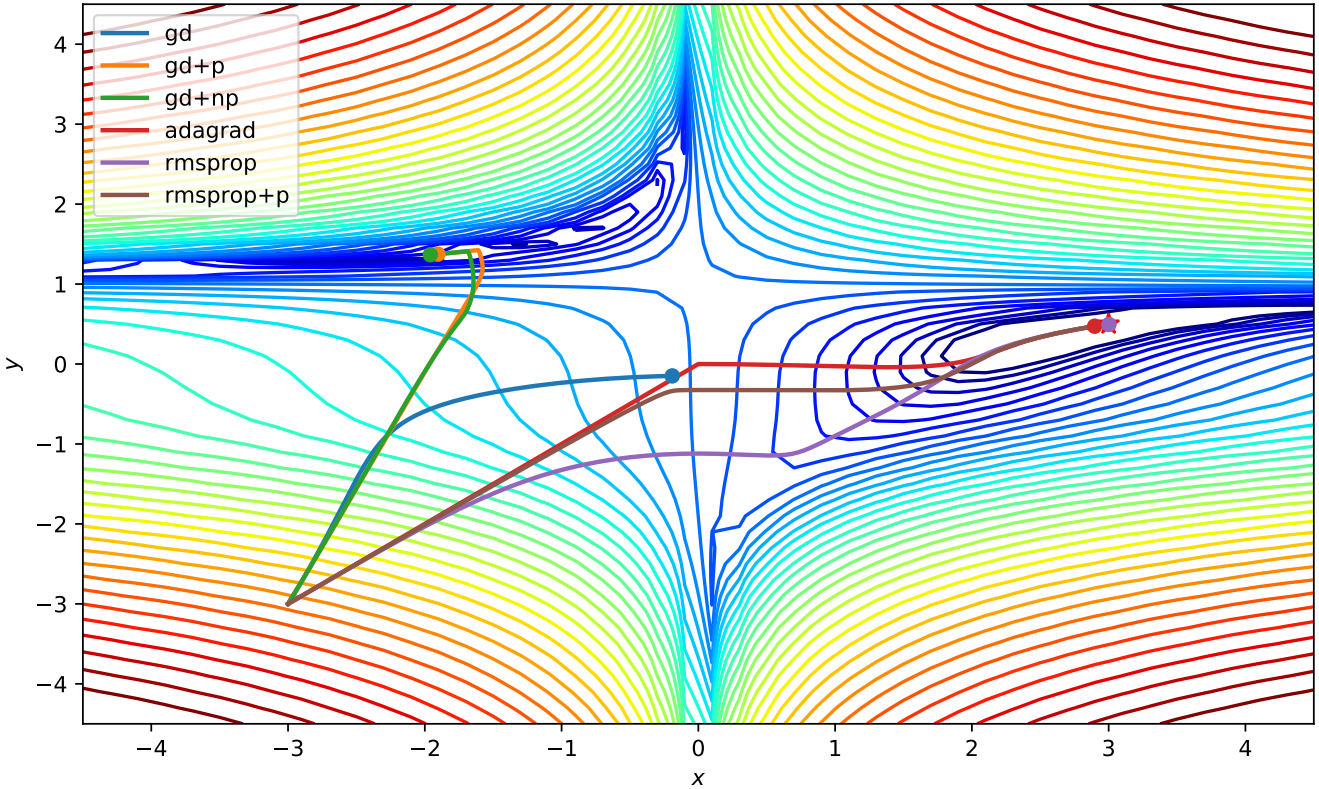
rmsprop+p denotes RMSProp with momentum. Though it makes a larger excursion out of the way, it gets to the optimum more quickly than all other optimizers. This is more apparent in the 2nd optimizer.




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

# RMSProp (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`

# 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.

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set $\beta_1$ and $\beta_2$ to be between $0$ and $1$. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize $\nu$ 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 gradient steps at the start of optimization.

# Adam (cont.)

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set $\beta_1$ and $\beta_2$ to be between $0$ and $1$. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize $\nu$ 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):

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

- Second moment update (gradient normalization):

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

- Bias correction in moments:

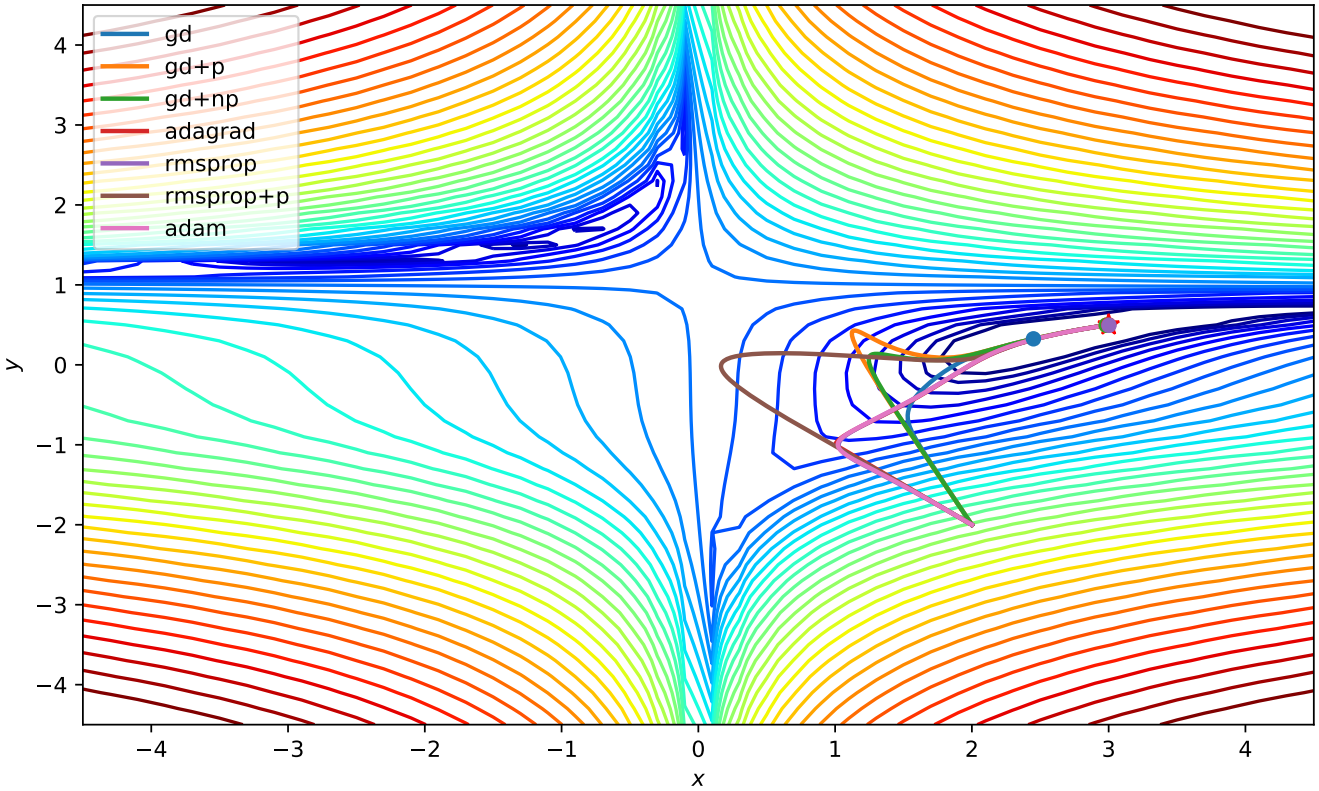
$$\begin{aligned} \tilde{\mathbf{v}} &= \frac{1}{1 - \beta_1^t}\mathbf{v} \\ \tilde{\mathbf{a}} &= \frac{1}{1 - \beta_2^t}\mathbf{a} \end{aligned}$$

- Gradient step:

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

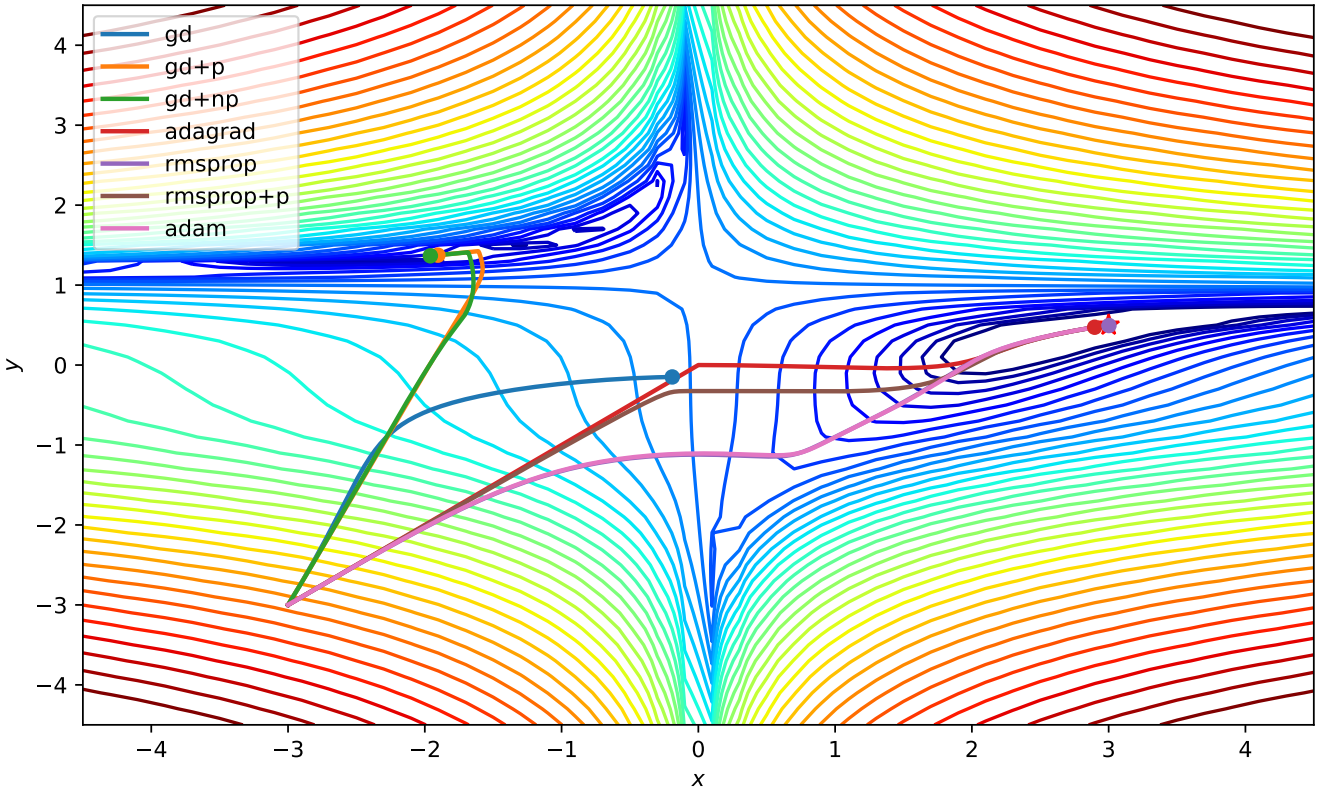
# Adam (opt 1)

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`

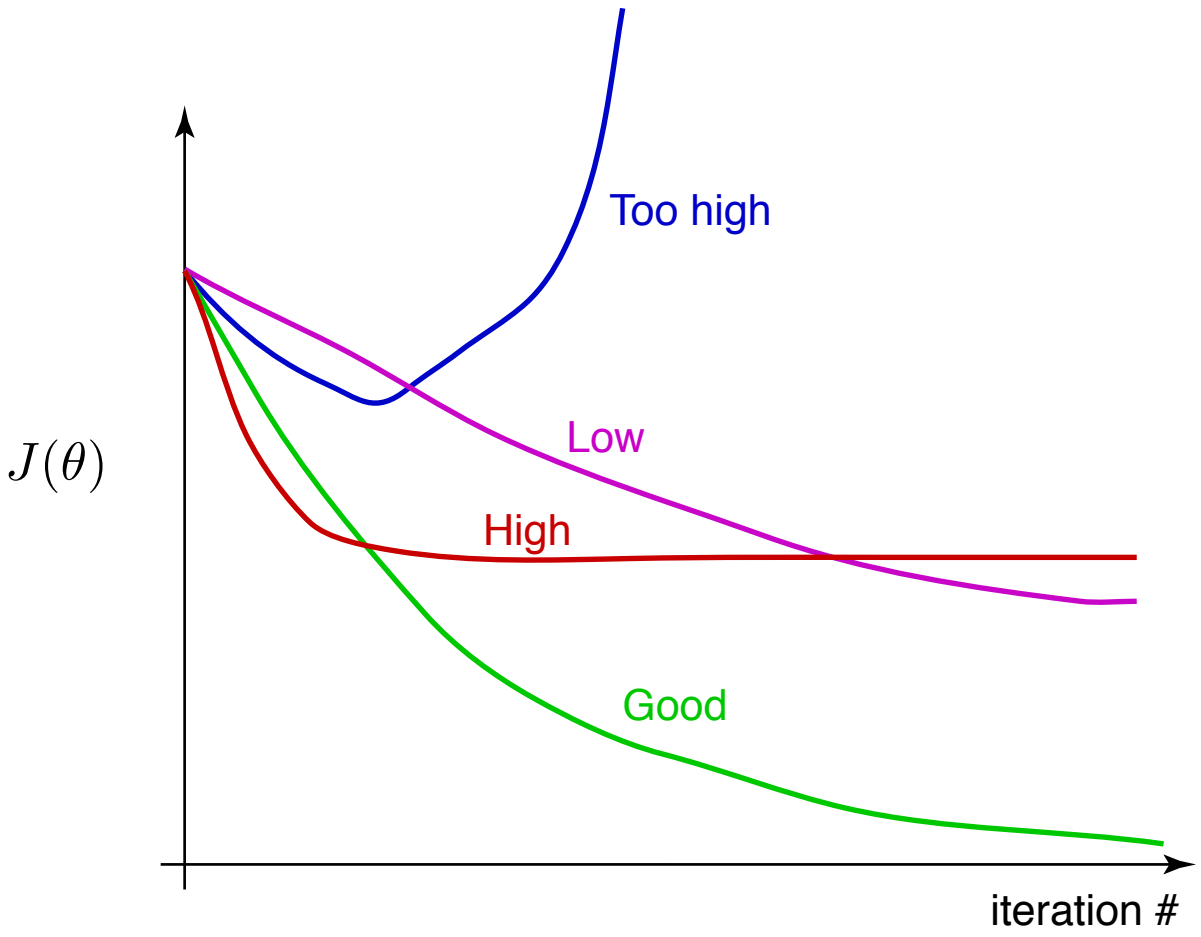
# Adam (opt 2)




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

# Interpreting the cost

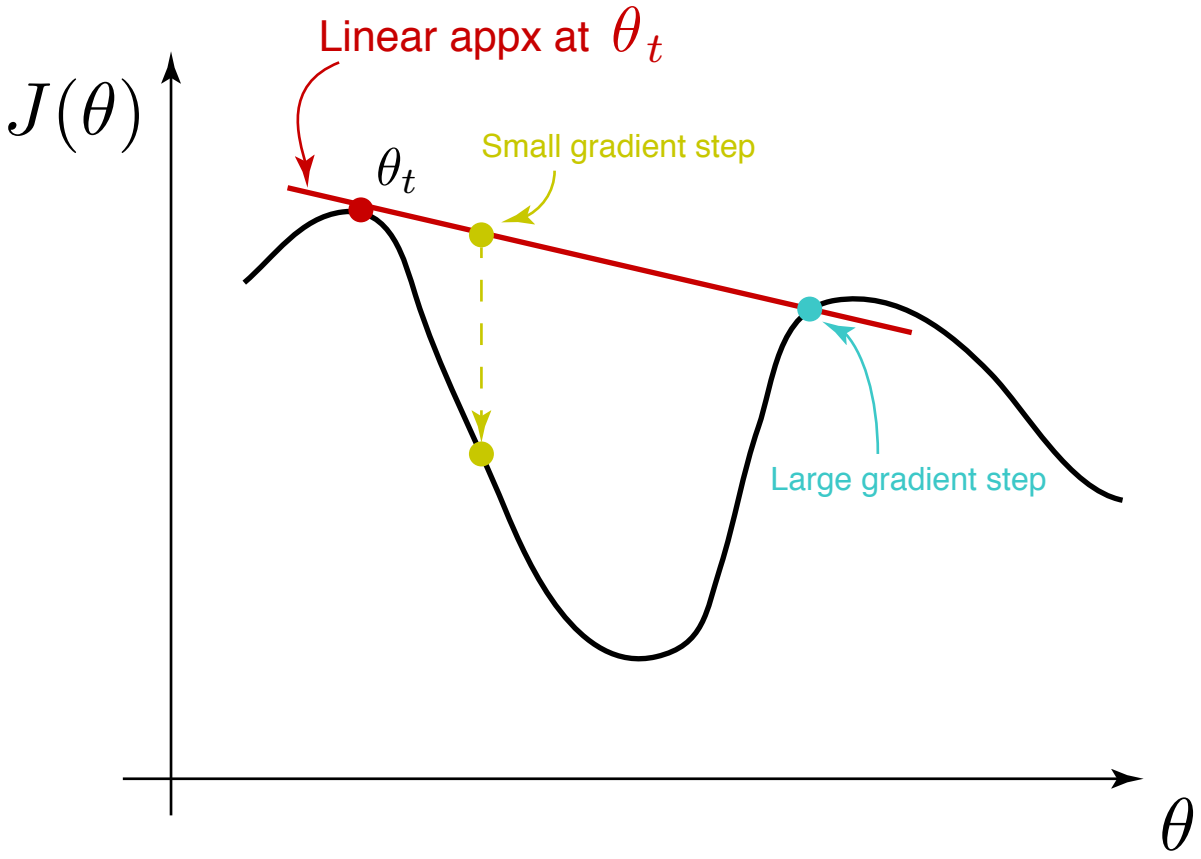
The cost function can be very informative as to how to adjust your step sizes for gradient descent.



# First order vs second order methods

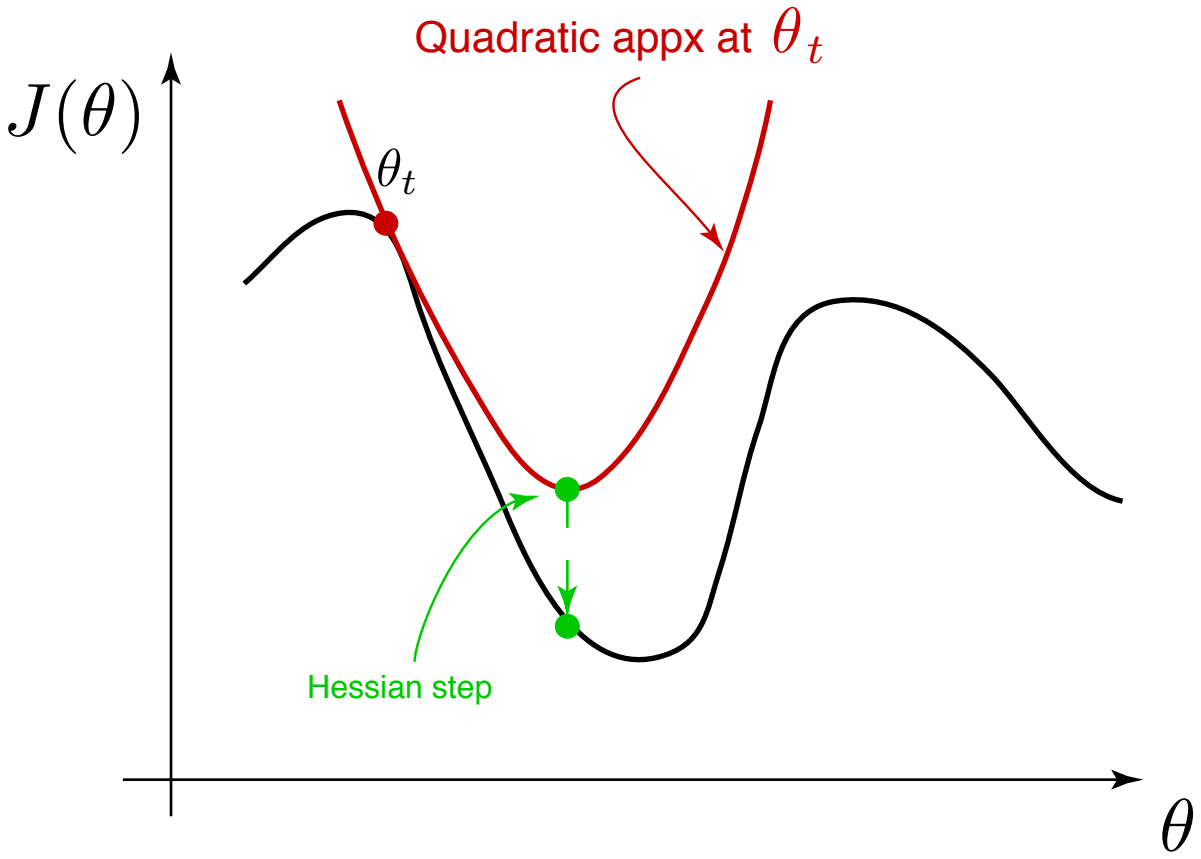
SGD, and optimizers used to augment the learning rate (Adagrad, RMSProp, and Adam), are all *first order* methods and have the learning rate $\varepsilon$ as a hyperparameter.

- 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:



# 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 $\theta_0$ 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 $\theta_0$ 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:

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

# 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 $\alpha$ becomes larger, this turns into first order gradient descent with learning rate $1/\alpha$.
- Newton's method finds any critical points, including saddle points. The algorithm as written will get stuck at these points, and not descend any further.
- 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 with gradient descent

- **Ill-conditioning of the Hessian matrix.**
  At $\theta_0$, a gradient step along $-\varepsilon\nabla J(\theta)$ will modify the cost function as:

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

  and thus when

  $$\frac{1}{2}\varepsilon^2\nabla J(\theta)^T\mathbf{H}J(\theta) > \varepsilon\nabla J(\theta)^T\nabla J(\theta)$$

  the cost will increase. Intuitively, when the curvature becomes too large, then the step size, $\varepsilon$, must be decreased.
- **Local minima.**
  Local minima are always a concern. Empirically, for large networks, most local minima have a low cost function value. The intuition is that for large networks, the system is so high-dimensional that minima where stepping in no direction will decrease the cost function are exponentially rare. Hence, local minima tend to be close to global minima.

  A check for local minima is to evaluate the gradient norm, and see if it shrinks. If it does, you are encountering a local minima (or saddle point).

# Challenges with gradient descent (cont.)

- **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.
- **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 $\lambda_i^t$. Architectural decisions, as well as appropriate regularization, can deal with vanishing gradients.