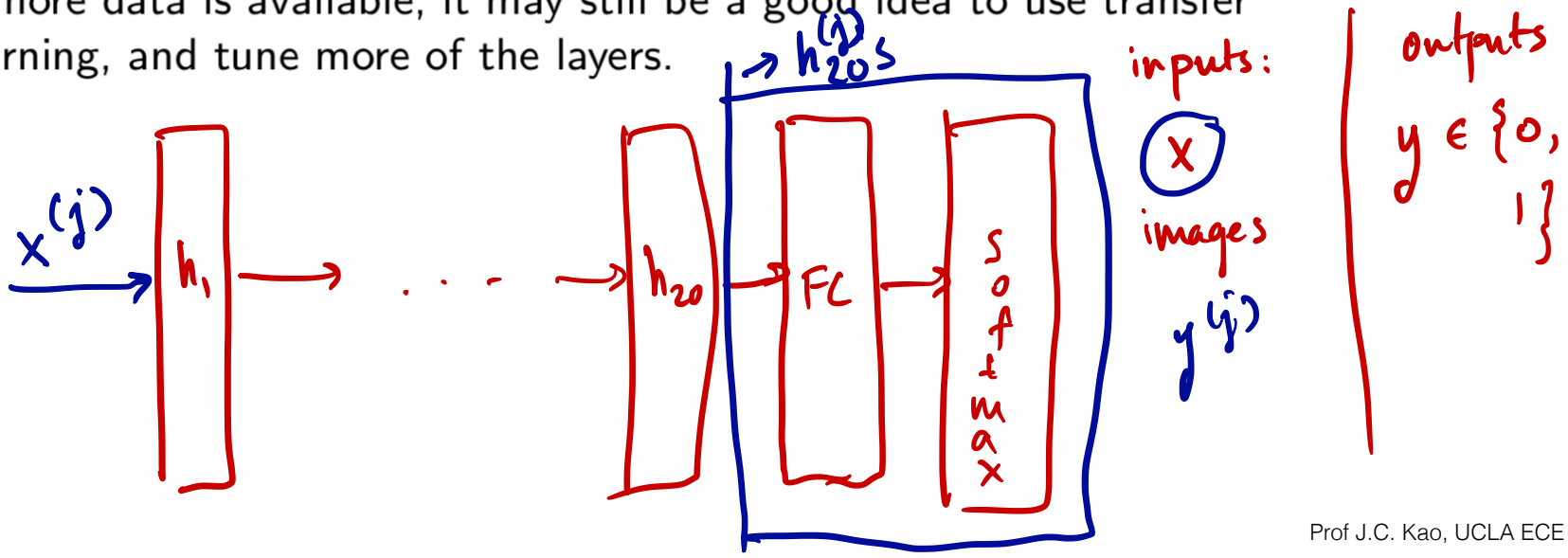
Announcements, 2018-02-07

- HW #3 due tonight, 11:59pm to Gradescope.
- HW #4 released tonight (after lecture) to CCLE.
- A word on the projects, in preparation for the last part of class.
- Midterm will be open book, open python / MATLAB, open CCLE (to get notes), but closed internet.
  - This does not mean that the solutions to questions are exact statements that you will find in the book.
  - In particular, if you find yourself flipping through pages to find a sentence here or there, it's probably not a great use of time.
- A word on the incident that occurred in last lecture.

Transfer learning

We'll discuss this more in the convolutional neural networks lecture, but a related idea is to take neural networks trained in one context and use them in another with little additional training.

- The idea is that if the tasks are similar enough, then the features at later layers of the network ought to be good features for this new task.
- If little training data is available to you, but the tasks are similar, all you may need to do is train a new linear layer at the output of the pre-trained network.
- If more data is available, it may still be a good idea to use transfer learning, and tune more of the layers.



RECAP: ensemble methods

One way to get a boost in performance for very little cognitive work is to use ensemble methods.

The approach is:

1. Train multiple different models
2. Average their results together at test time.

This almost always increases performance by substantial amounts (e.g., a few percentage improvement in testing).

RECAP: ensemble methods

- The basic intuition between ensemble methods is that if models are independent, they will usually not all make the same errors on the test set.
- With k independent models, the average model error will decrease by a factor $\frac{1}{k}$. Denoting ϵ_i to be the error of model i on an example, and assuming $\mathbb{E}\epsilon_i = 0$ as well as that the statistics of this error is the same across all models,

$$\begin{aligned}\mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] &= \frac{1}{k^2}\sum_{i=1}^{k}\mathbb{E}\epsilon_i^2 \\ &= \frac{1}{k}\mathbb{E}\epsilon_i^2\end{aligned}$$

- If the models are not independent, it can be shown that:

$$\frac{1}{k}\mathbb{E}\epsilon_i^2 + \frac{k-1}{k}\mathbb{E}[\epsilon_i\epsilon_j]$$

0 if indep.

which is equal to $\mathbb{E}\epsilon_i^2$ only when the models are perfectly correlated.

Ensemble methods

One implementation of ensemble methods is via bagging.

Bagging

Bagging stands for bootstrap aggregating. It is an ensemble method for regularization. The procedure is as follows:

- Construct k datasets using the bootstrap (i.e., set a data size, N, and draw with replacement from the original dataset to get N samples; do this k times).
- Train k different models using these k datasets.

k new datasets
1: N ex. w/ repl.
2: N ex. w/ repl.
N
K
$1 - \frac{1}{e} \approx 64\%$

N examples in orig. training data

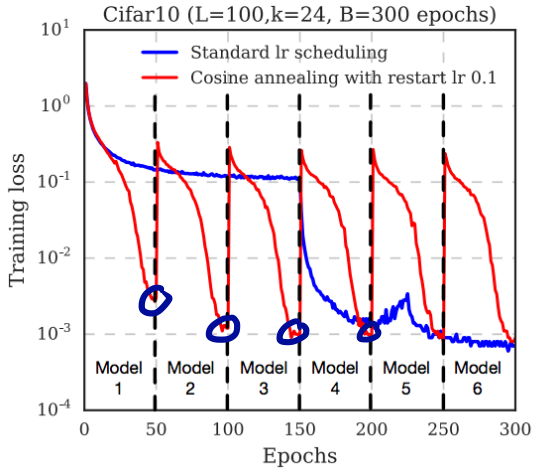
A few notes:

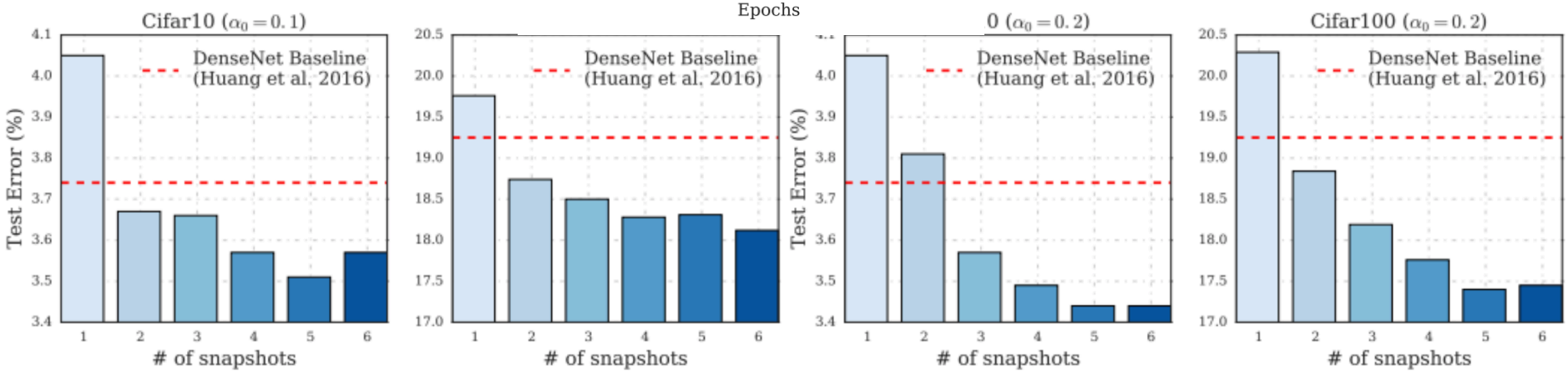
- In practice, neural networks reach a wide variety of solutions given different initializations, hyperparameters, etc., and so in practice even if they are trained from the same dataset, they tend to produce partially independent errors.
- While model averaging is powerful, it is expensive for neural networks, since the time to train models can be very large.

Ensemble methods

Ways to get around computational expense?

You could take snapshots of the model at different local minima and average them together. See Huang, Li et al., ICLR 2017.





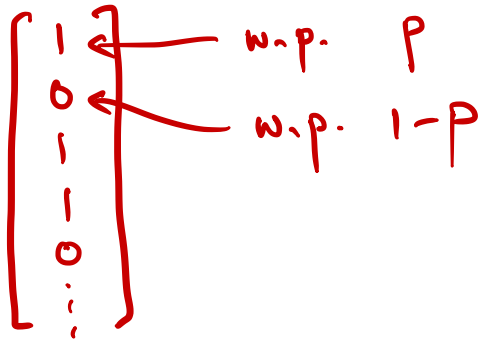
Dropout

Dropout is a computationally inexpensive yet effective method for generalization. It can be viewed as approximating the bagging procedure for exponentially many models, while only optimizing a single set of parameters. The following steps describe the dropout regularizer:

- On a given training iteration, sample a binary mask (i.e., each element is 0 or 1) for all input and hidden units in the network.
  - The Bernoulli parameter, p, is a hyperparameter.
  - Typical values are that $p = 0.8$ for input units and $p = 0.5$ for hidden units.
- Apply (i.e., multiply) the mask to all units. Then perform the forward pass and backwards pass, and parameter update.
- In *prediction*, multiply each hidden unit by the parameter of its Bernoulli mask, p.

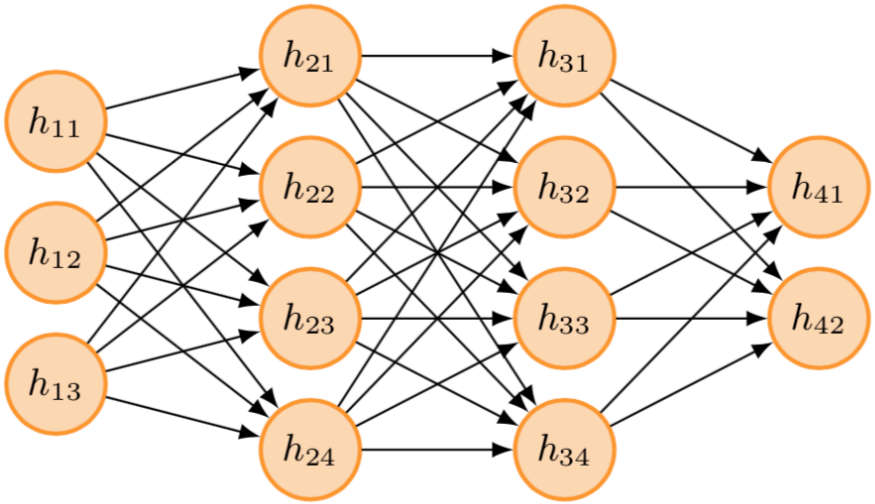
100 neurons / units

100 Bernoulli R.V's w.p. p

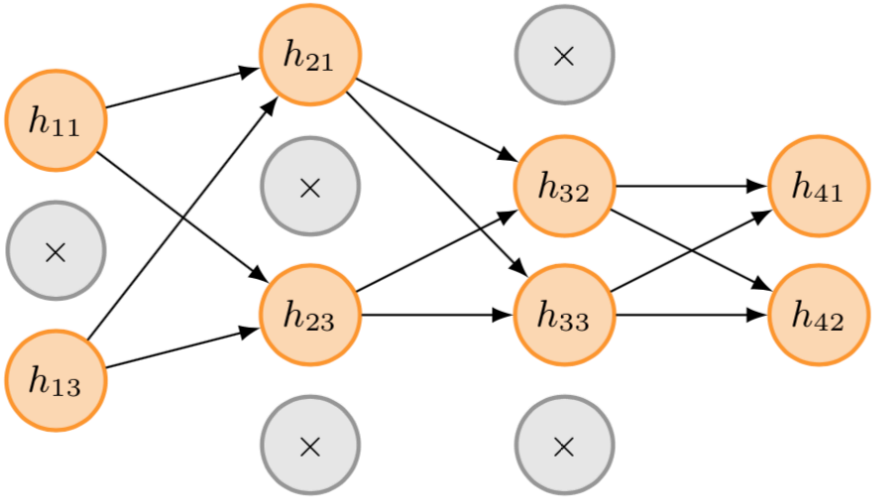


Dropout

Hidden layers of network to be trained



Application of random mask on iteration i



Dropout

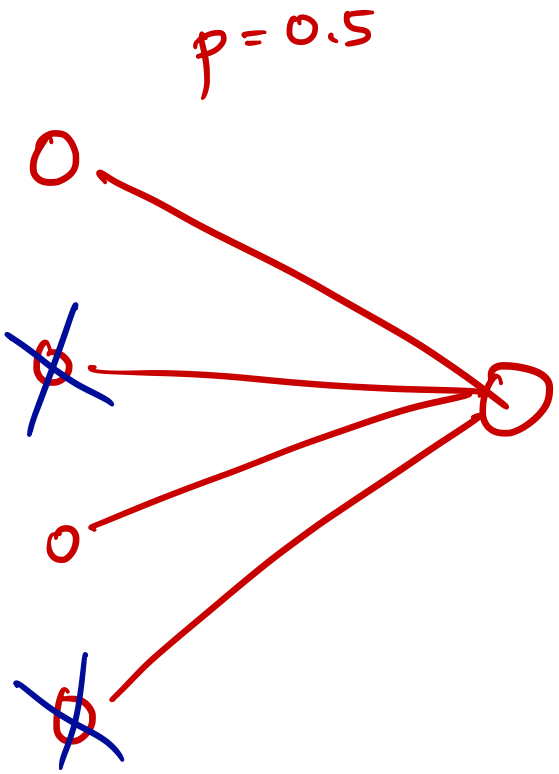
Dropout in code.

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = np.random.rand(*H1.shape) < p # Sample random mask
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = np.random.rand(*H2.shape) < p # Sample random mask
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3
```

0, 1

Dropout

How about during test time? What configuration do you use?



Take acts at each layer and scale down by a factor p.

Dropout

How about during test time? What configuration do you use?

*We call this approach the **weight scaling inference rule**. There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.*

In this class, instead of scaling the weights, we'll scale the activations.

Inverted dropout

A common way to implement dropout is *inverted dropout* where the scaling by $1/p$ is done in training. This causes the activations to have the same expected value as if dropout was never been performed.

Thus, testing looks the same irrespective of if we use dropout or not. See code below:

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def train_forward(X):
    H1 = relu(np.dot(W1, X) + b1) # First hidden layer activations
    M1 = (np.random.rand(*H1.shape) < p) / p # Sample random mask AND normalization by p
    H1 *= M1 # Dropout on first hidden layer

    H2 = relu(np.dot(W2, H1) + b2) # Second hidden layer activations
    M2 = (np.random.rand(*H2.shape) < p) / p # Sample random mask AND normalization by p
    H2 *= M2 # Dropout on first hidden layer

    Z = np.dot(W3, H2) + b3

def test(X):
    H1 = relu(np.dot(W1, X) + b1)
    H2 = relu(np.dot(W1, H1) + b1)
    Z = np.dot(W3, H2) + b3
```

Dropout

```
p = 0.5 # probability of dropping out
relu = lambda x: x * (x > 0)

def test(X):
    H1 = relu(np.dot(W1, X) + b1) * p
    H2 = relu(np.dot(W1, H1) + b1) * p
    Z = np.dot(W3, H2) + b3
```

Note: an additional pro of dropout is that in testing time, there is no additional complexity. With *m* ensemble models, our test time evaluation would scale *O(m)* (if not parallelized).

Dropout

How is this a good idea?

1) Dropout approximates bagging, since each mask is like a different model. For a model with N hidden units, there are 2^N different model configurations.

   Each of these configurations must be good at predicting the output.

2) You can think of of dropout as regularizing each hidden unit to work well in many different contexts.

3) Dropout may cause features to be encoded redundantly amongst neurons (e.g., if a feature is critical to classification, dropout will cause it to not be encoded in just one unit).

Lecture summary

Here, we've 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.

Lecture 7: 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
- Overview of second order methods
- Challenges of gradient descent

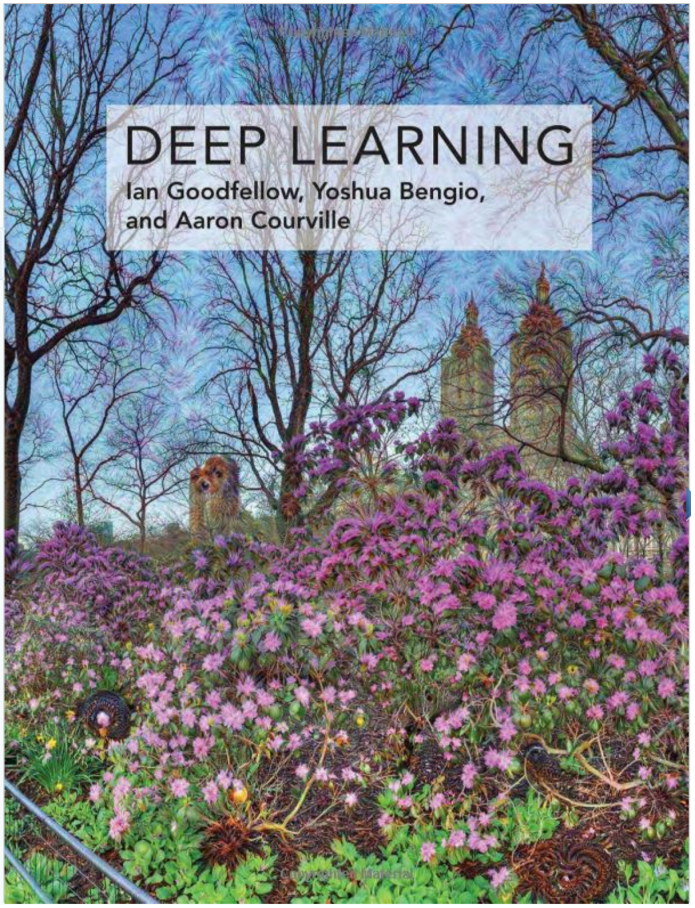
1st order techniques.

← EE 236B/C

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.
- Parameters and cost functions to use for neural networks.
- How to calculate derivatives of all parameters in the neural network.
- 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 (or subgradient) 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: $J(\theta)$ $\mathcal{L}(\theta)$ $\ell(\theta)$
- Parameters: θ

Then, the gradient descent step is:

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

Stochastic gradient descent

is a fn. of the training data

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.

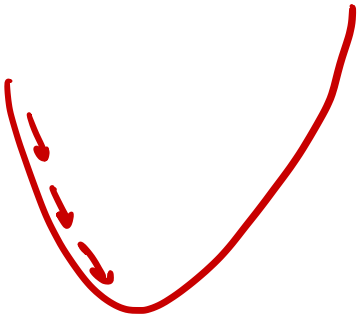
Stochastic gradient descent

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

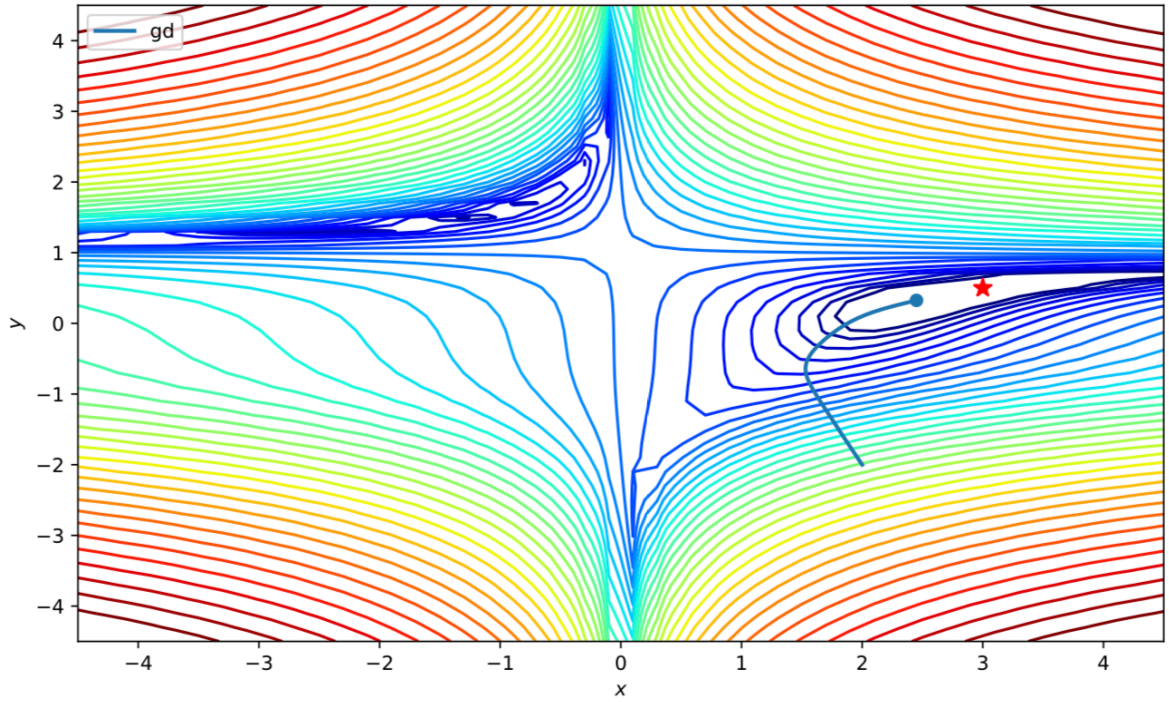
svm / softmax / nn . loss

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

x = θ

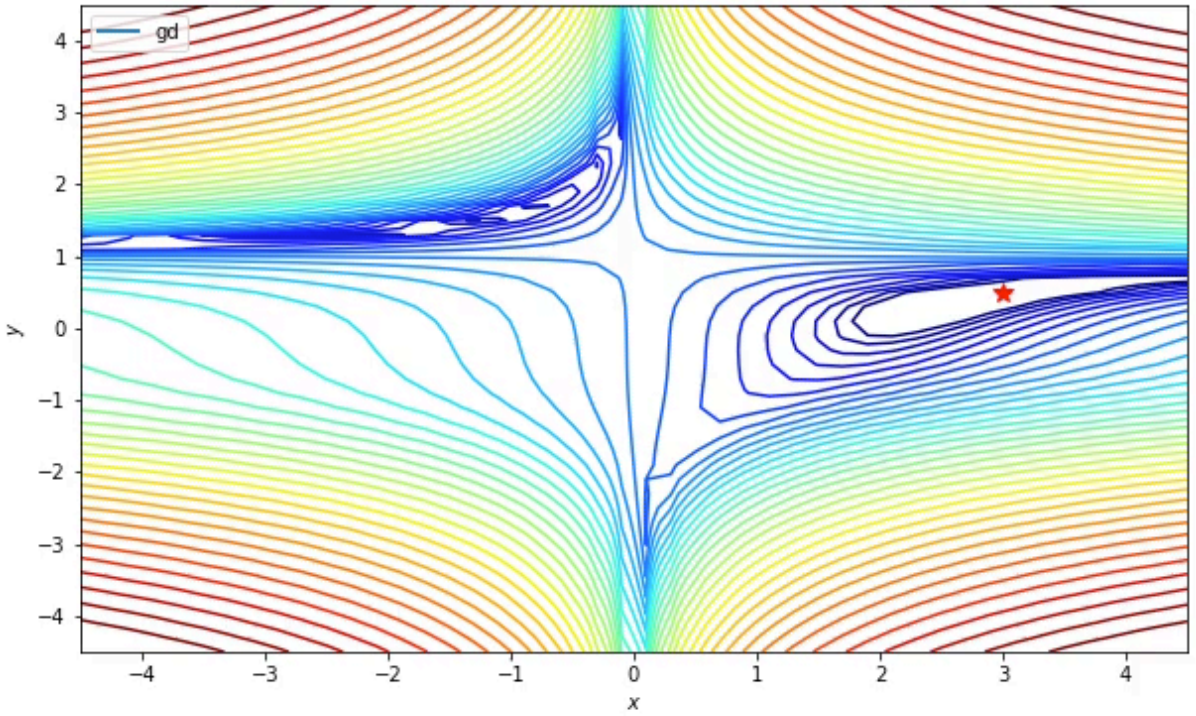
Stochastic gradient descent (opt 1)

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.

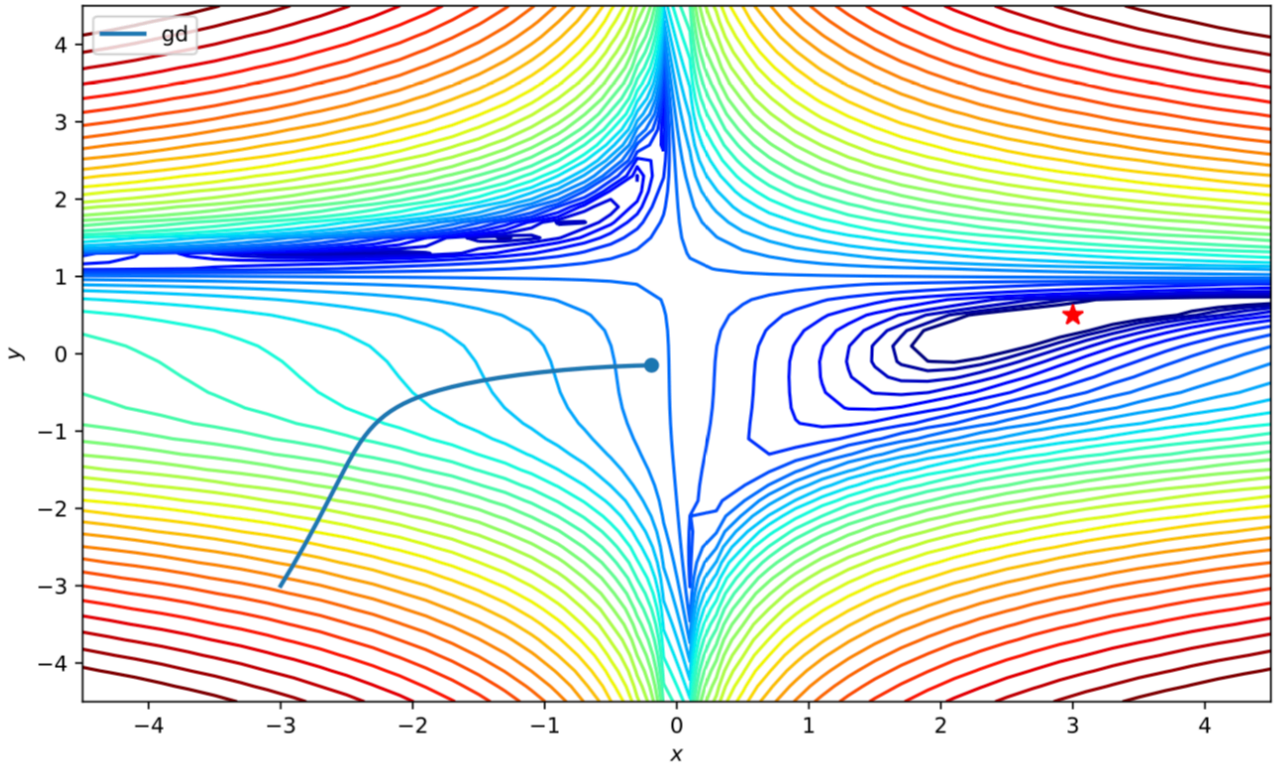


Video: http://seas.ucla.edu/~kao/opt_anim/1gd.mp4
Animation help thanks to: http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/

Stochastic gradient descent

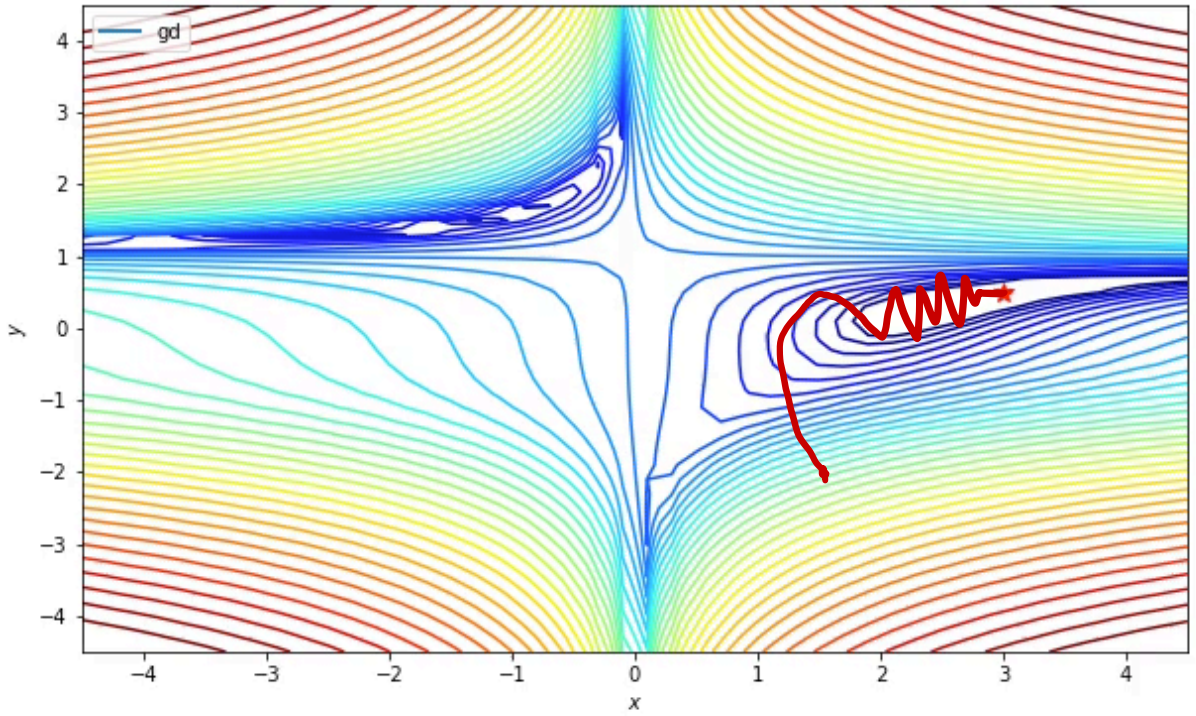


Stochastic gradient descent (opt 2)



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

Stochastic gradient descent



Momentum

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

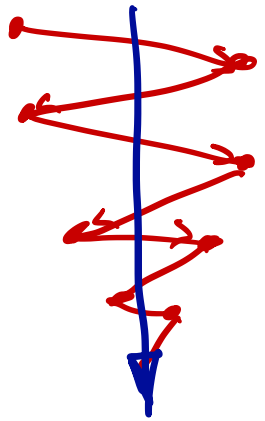
Initialize $\mathbf{v} = 0$. Set $\alpha \in [0, 1]$. Typical values are $\alpha = 0.9$ or 0.99. Then, until stopping criterion is met:

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

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

- Gradient step:

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

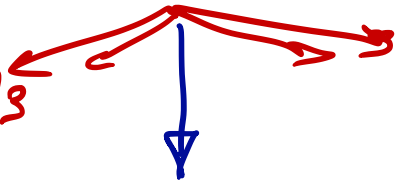


$v_0 = 0$

$v_1 = -\varepsilon g_1$

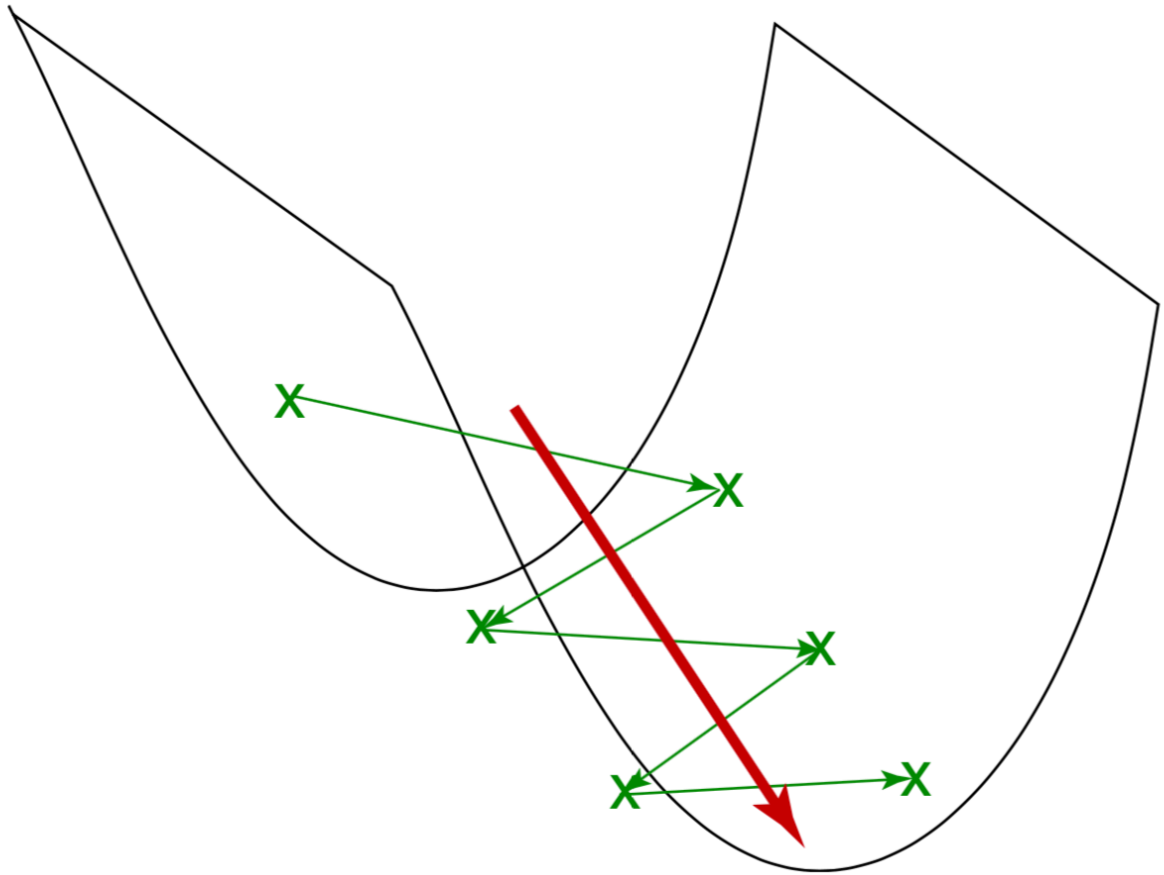
$v_2 = \alpha v_1 - \varepsilon g_2 = -\alpha \varepsilon g_1 - \varepsilon g_2$

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



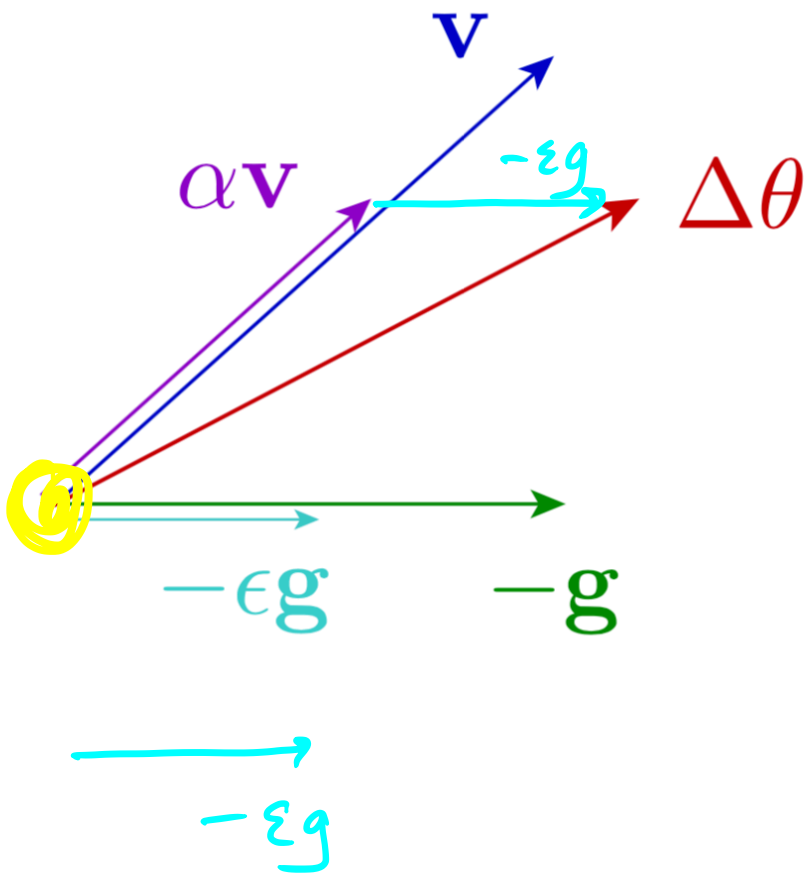
Momentum (cont.)

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:



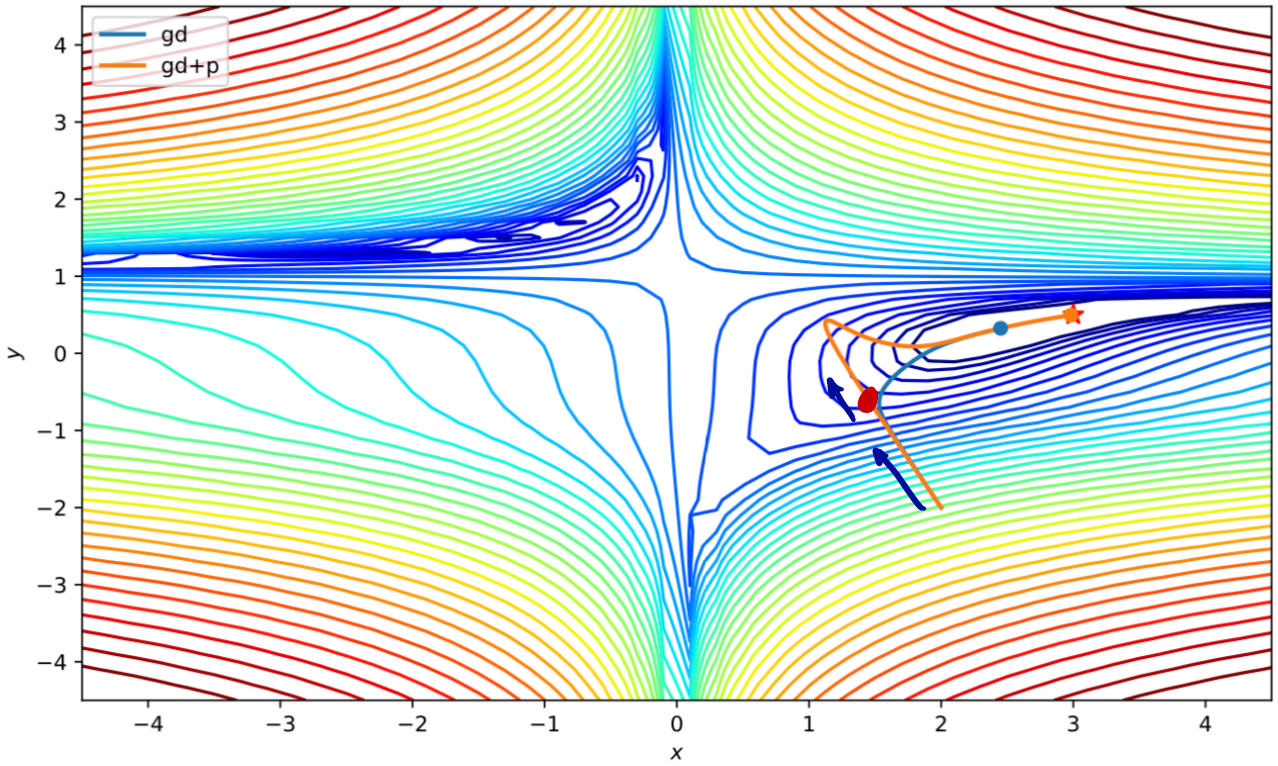
Momentum

```
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])
```

v = momentum

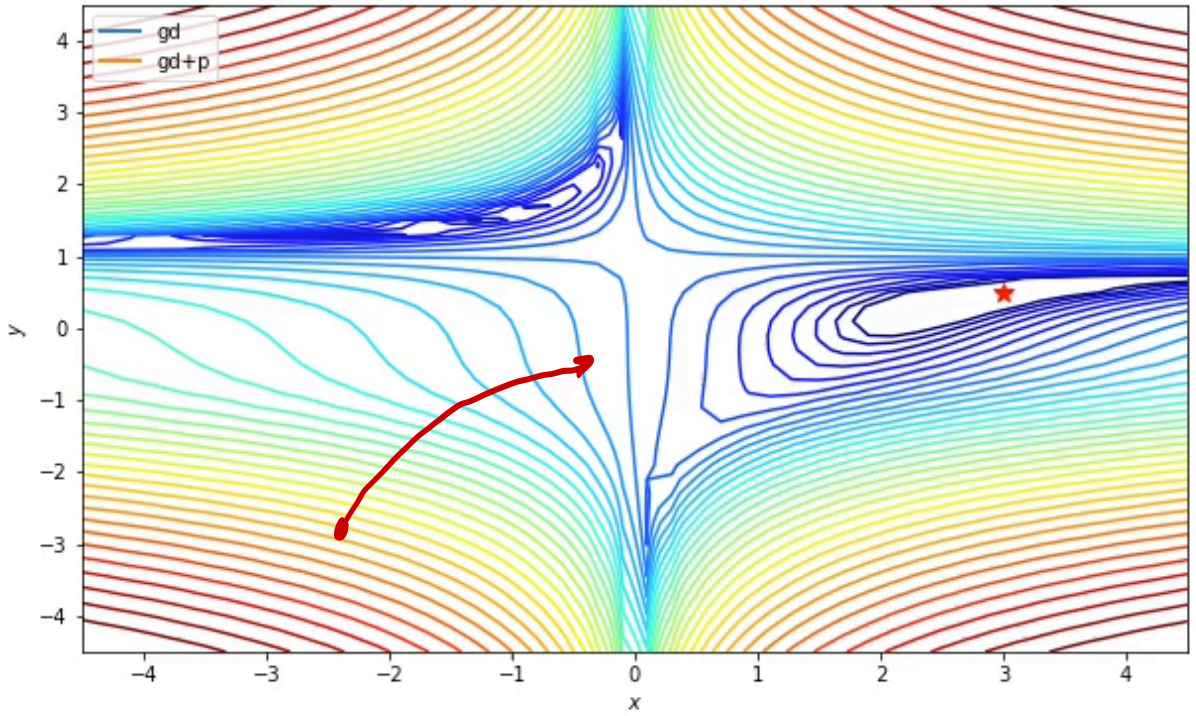
Momentum (opt 1)

gd+p denotes gradient descent with momentum.



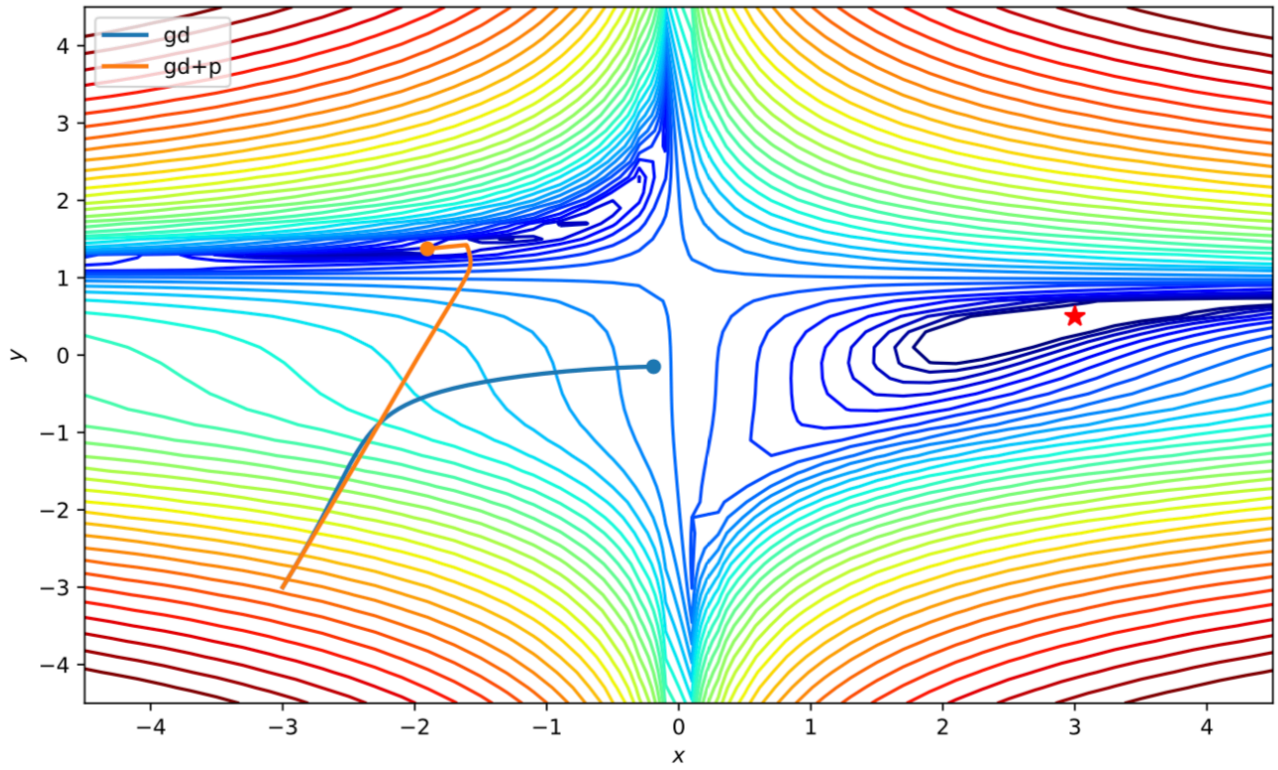
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p.mp4`

Momentum



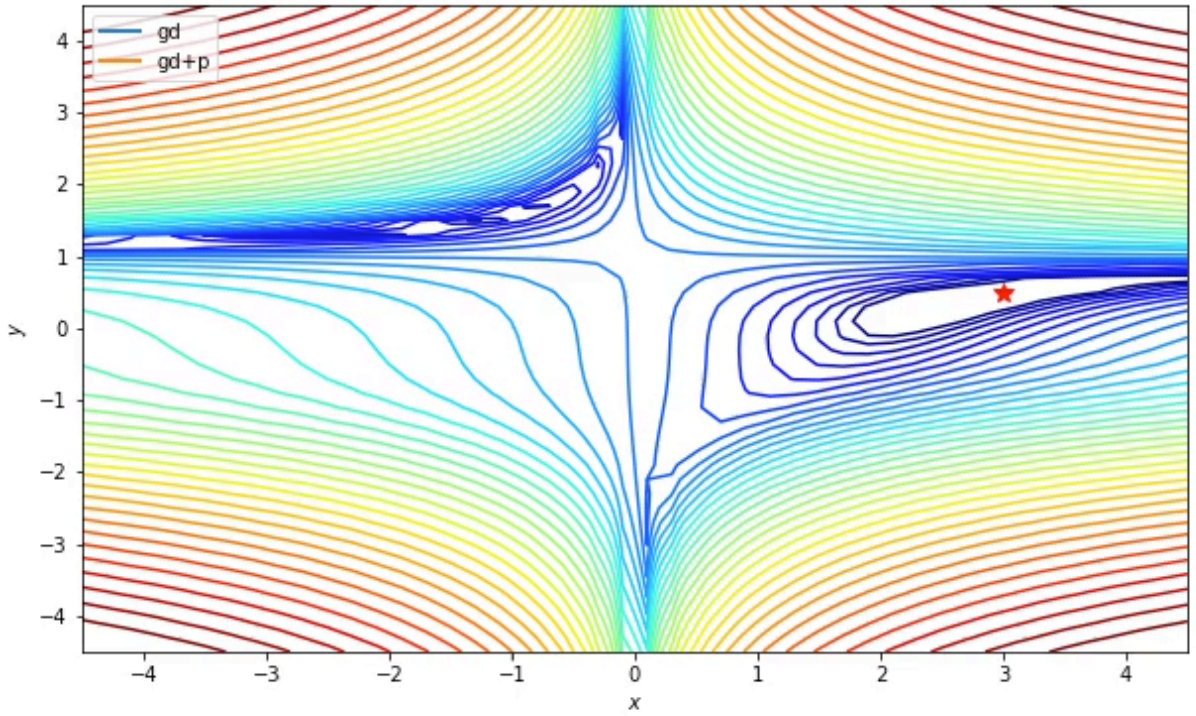
Momentum (opt 2)

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`

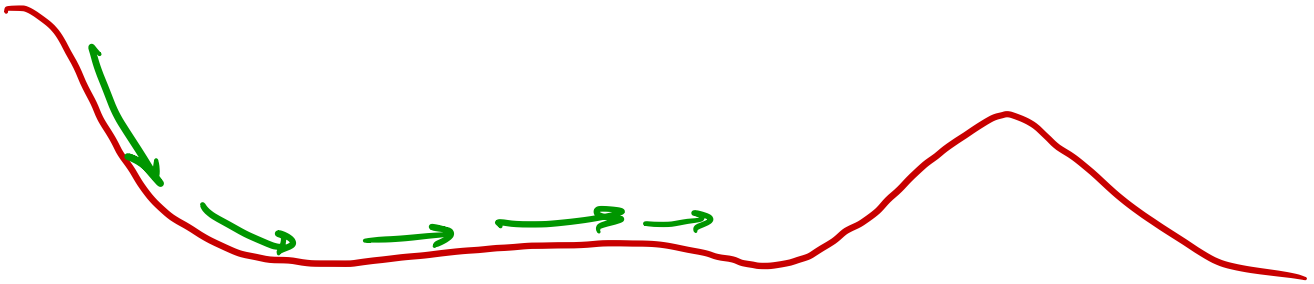
Momentum



Does momentum help with local optima?

Does momentum help with local optima?

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:

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

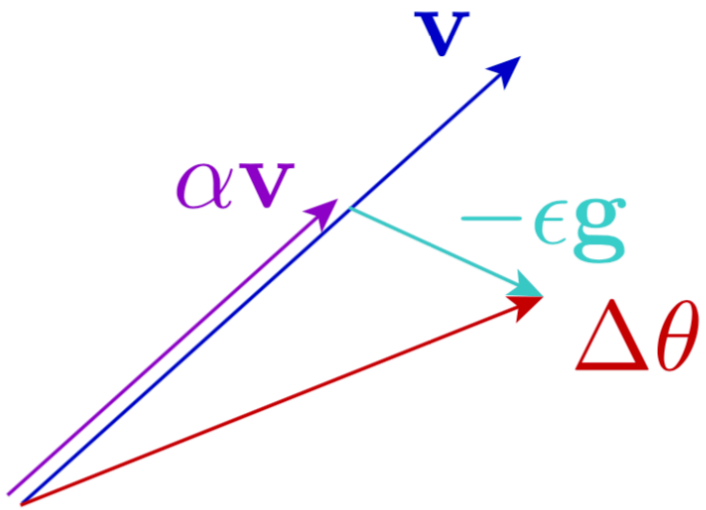
in momentum $-\varepsilon \nabla_\theta J(\theta)$

- Gradient step:

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

Nesterov momentum (cont.)

The following image is appropriate for Nesterov momentum:



Nesterov momentum

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:

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

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

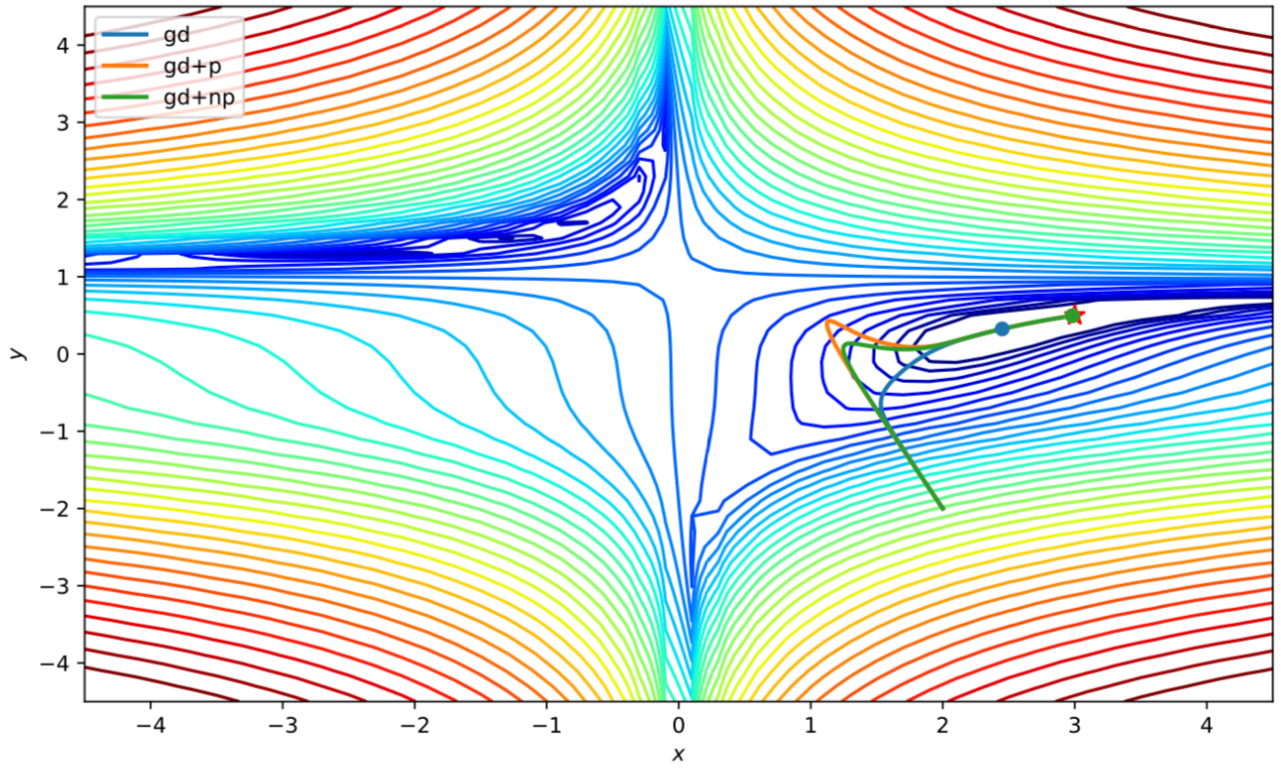
Nesterov momentum

```
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])
```

calc. of v_{new}

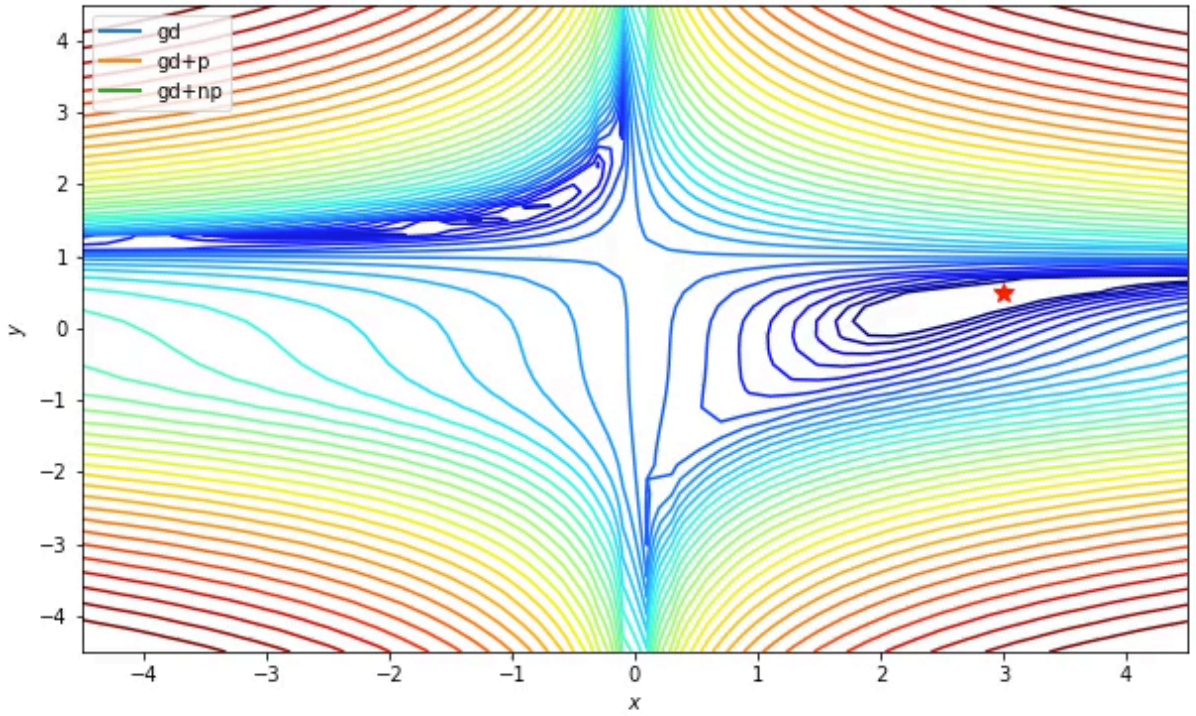
Nesterov momentum (opt 1)

gd+np denotes gradient descent with Nesterov momentum.



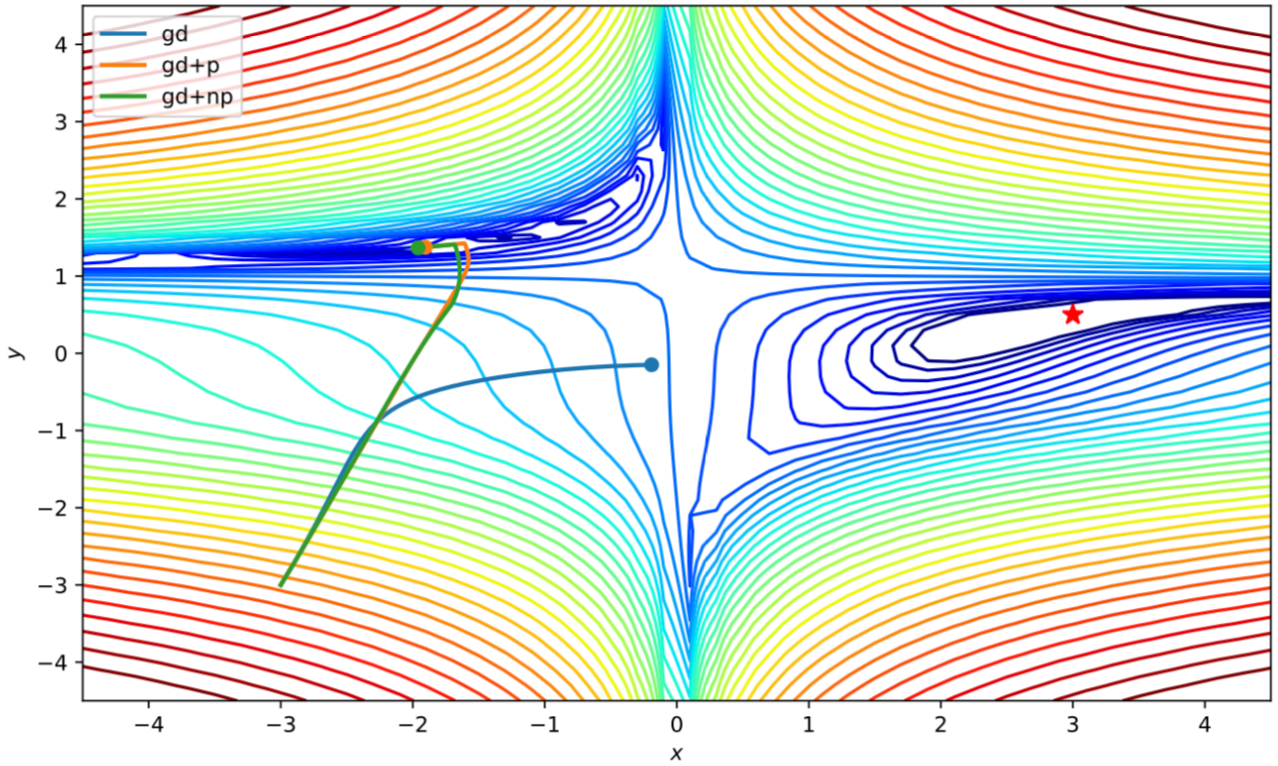
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np.mp4`

Nesterov momentum



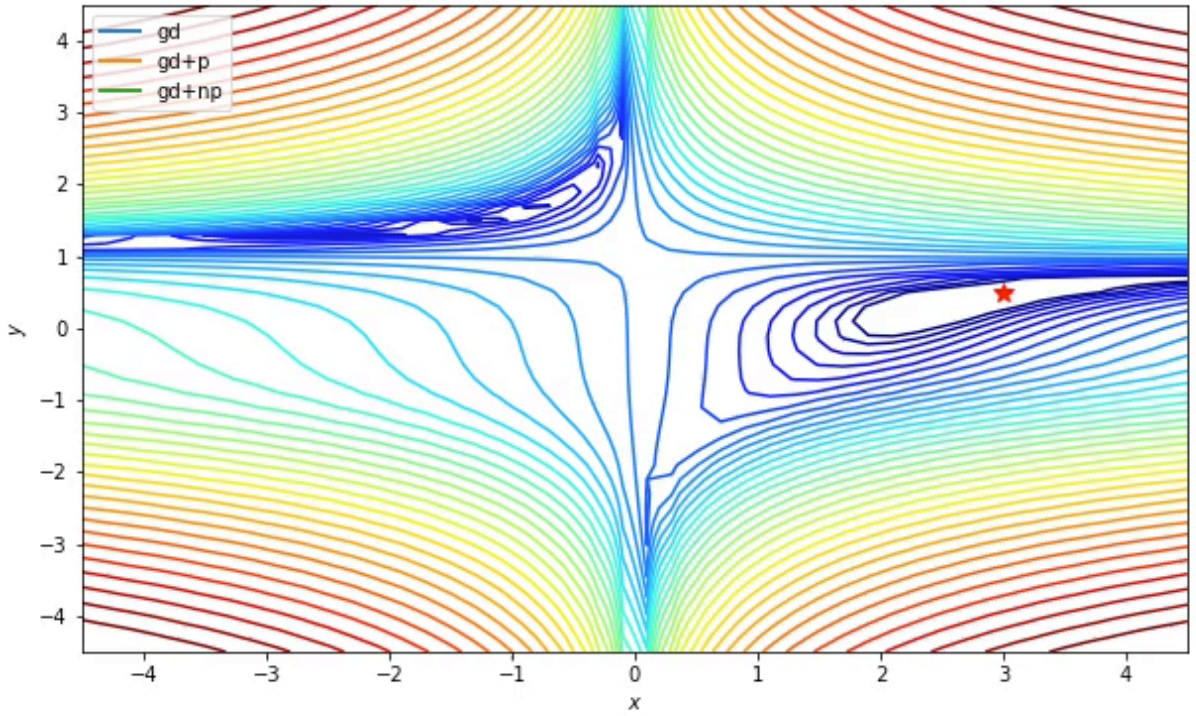
Nesterov momentum (opt 2)

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)

John Duchi 2011

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:

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

$g \in \mathbb{R}^n$

$a \in \mathbb{R}^n$

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

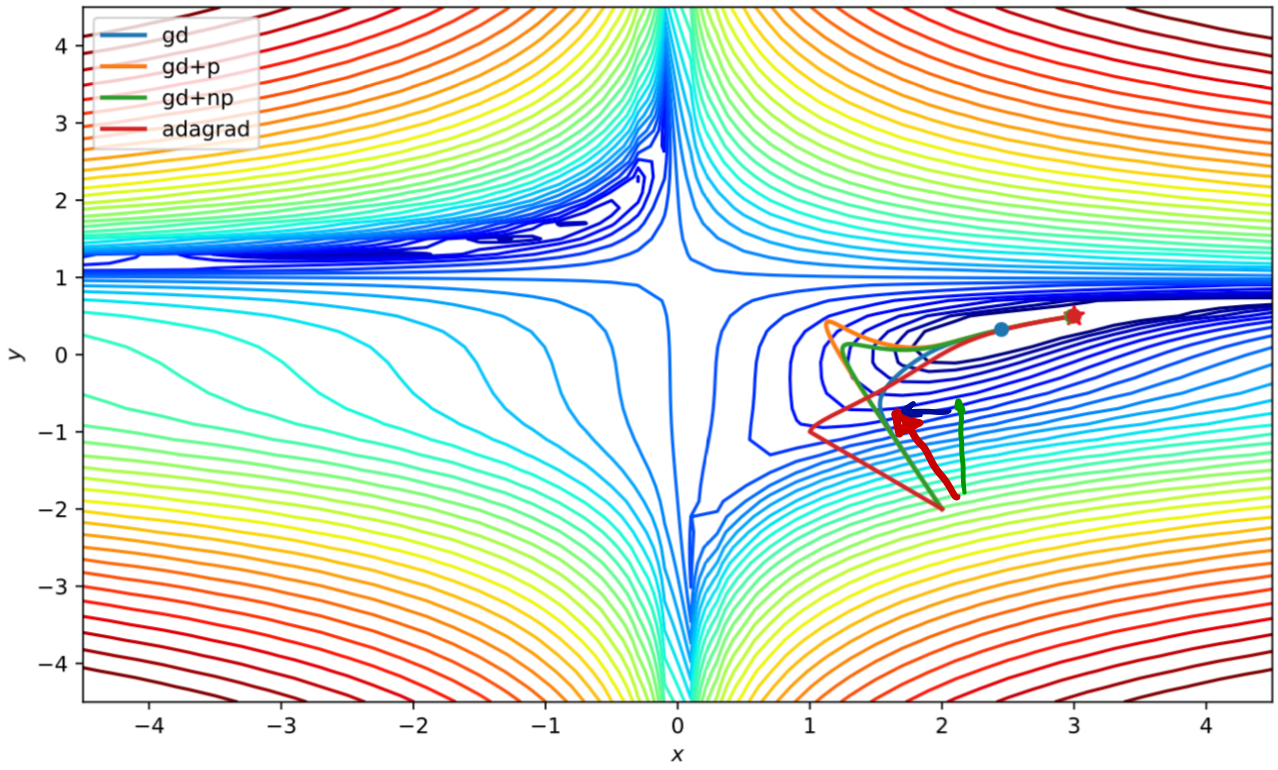
$$\frac{\varepsilon}{\sqrt{a} + \nu} = \begin{bmatrix} \frac{\varepsilon}{\sqrt{a_1} + \nu} \\ \frac{\varepsilon}{\sqrt{a_2} + \nu} \\ \vdots \\ \frac{\varepsilon}{\sqrt{a_n} + \nu} \end{bmatrix} \in \mathbb{R}^n$$

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

Adagrad

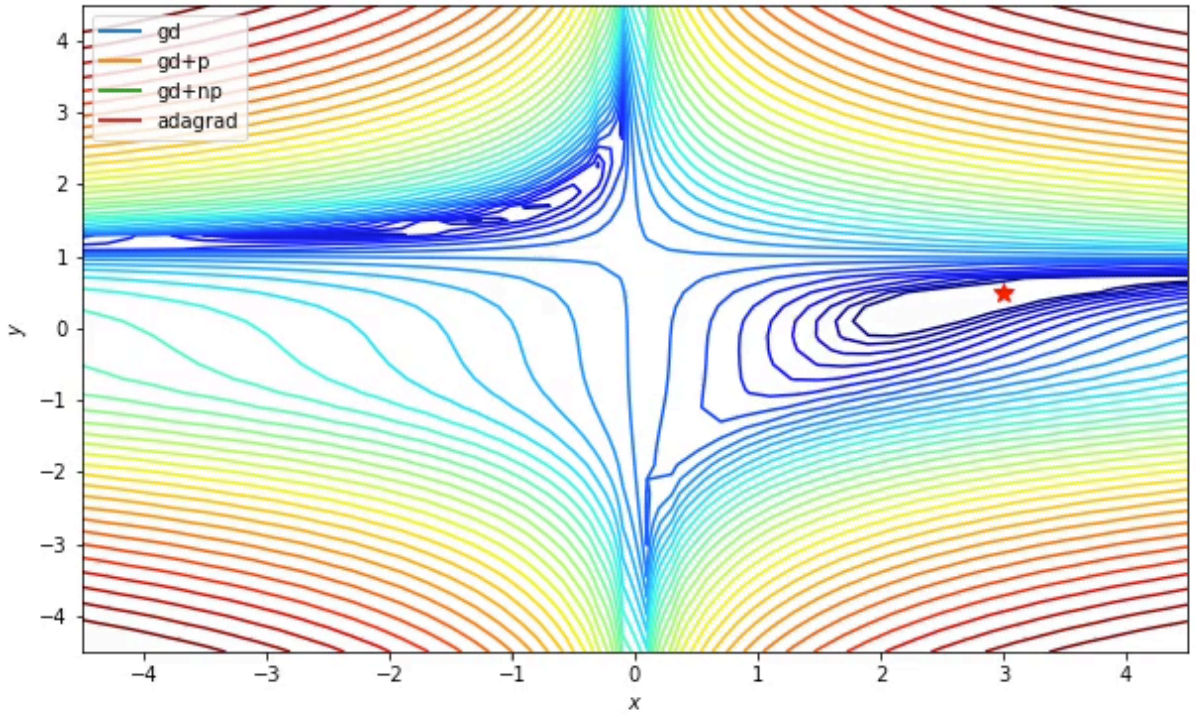
```
a = 0
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a += g*g
    x -= eps * g / (np.sqrt(a) + nu)
    last_diff = np.linalg.norm(x - path[-1])
```

Adagrad (opt 1)



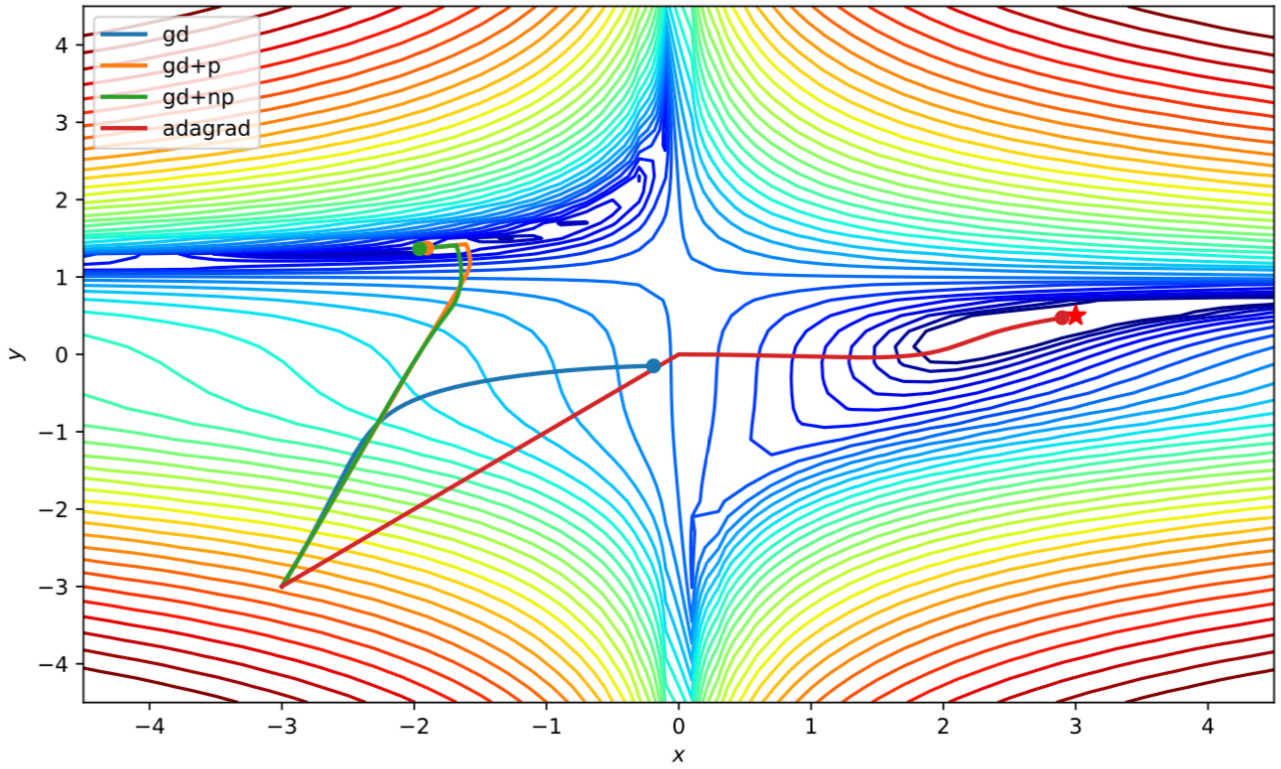
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad.mp4`

Adagrad



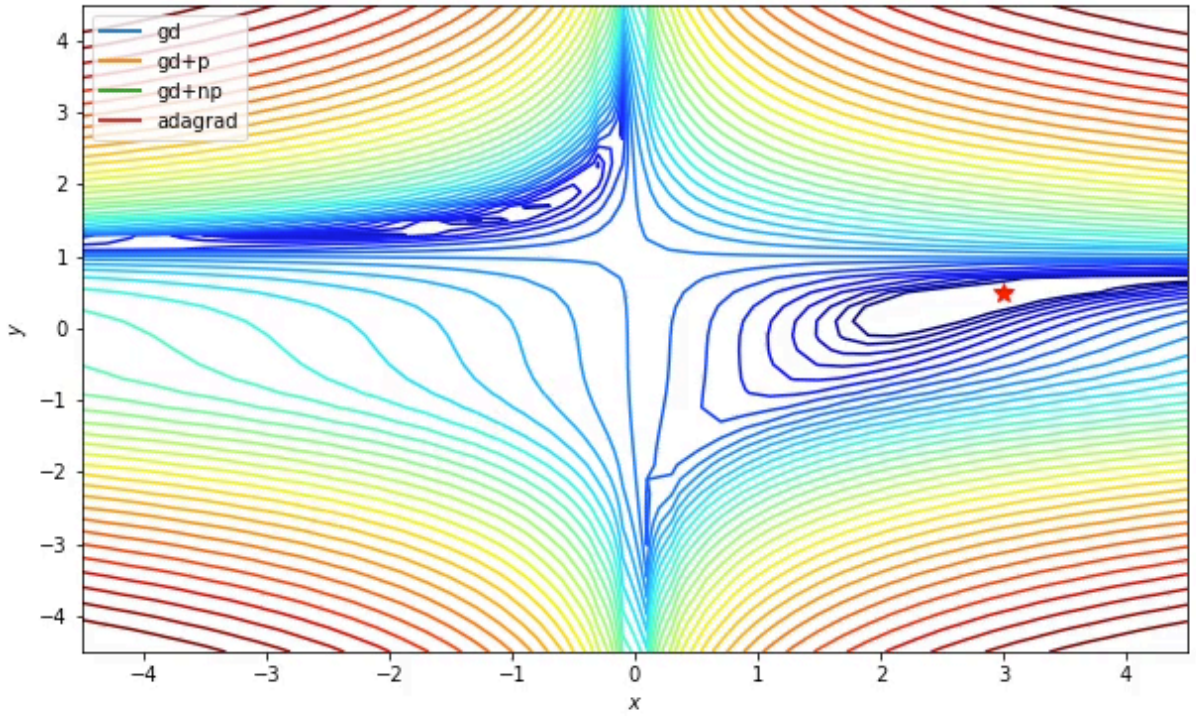
Adagrad (opt 2)

Adagrad proceeds to the global minimum.



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

Adagrad



Adagrad

Is there a problem with adagrad?

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

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

- Gradient step:

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

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:

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

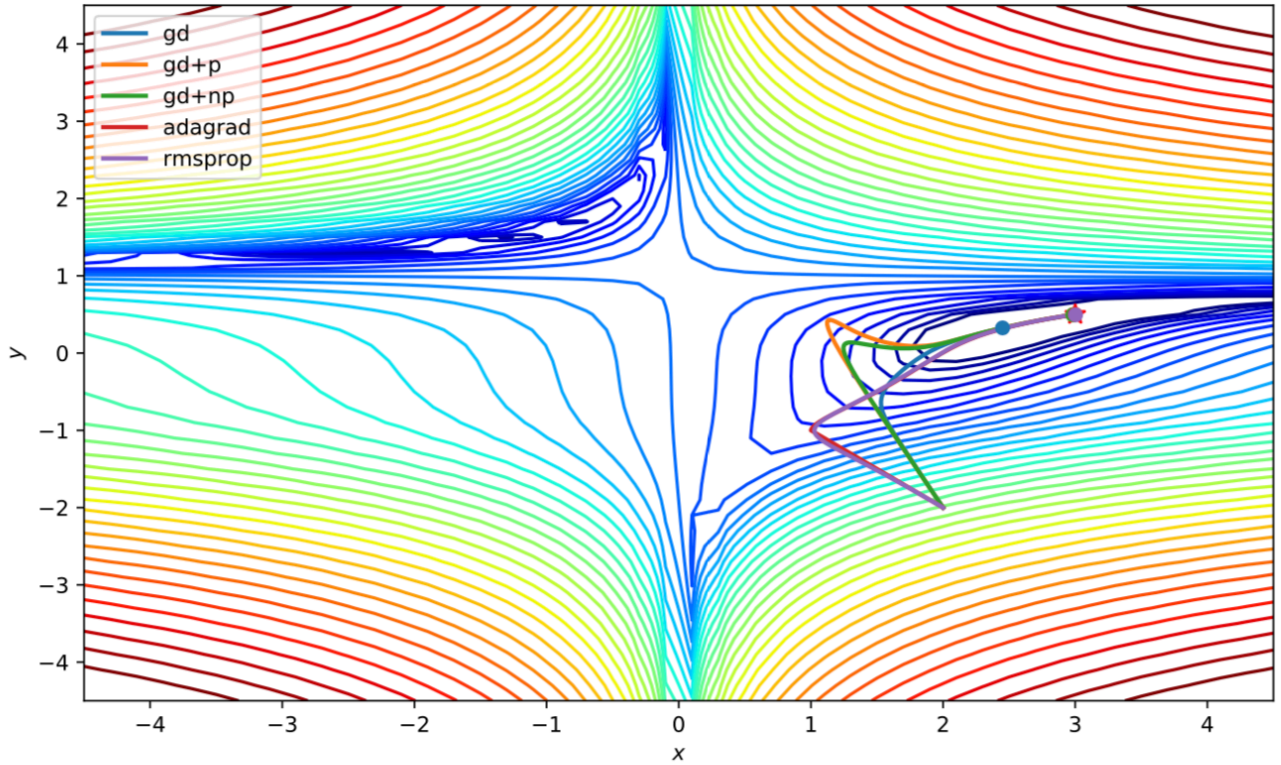
- Gradient step:

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

RMSProp

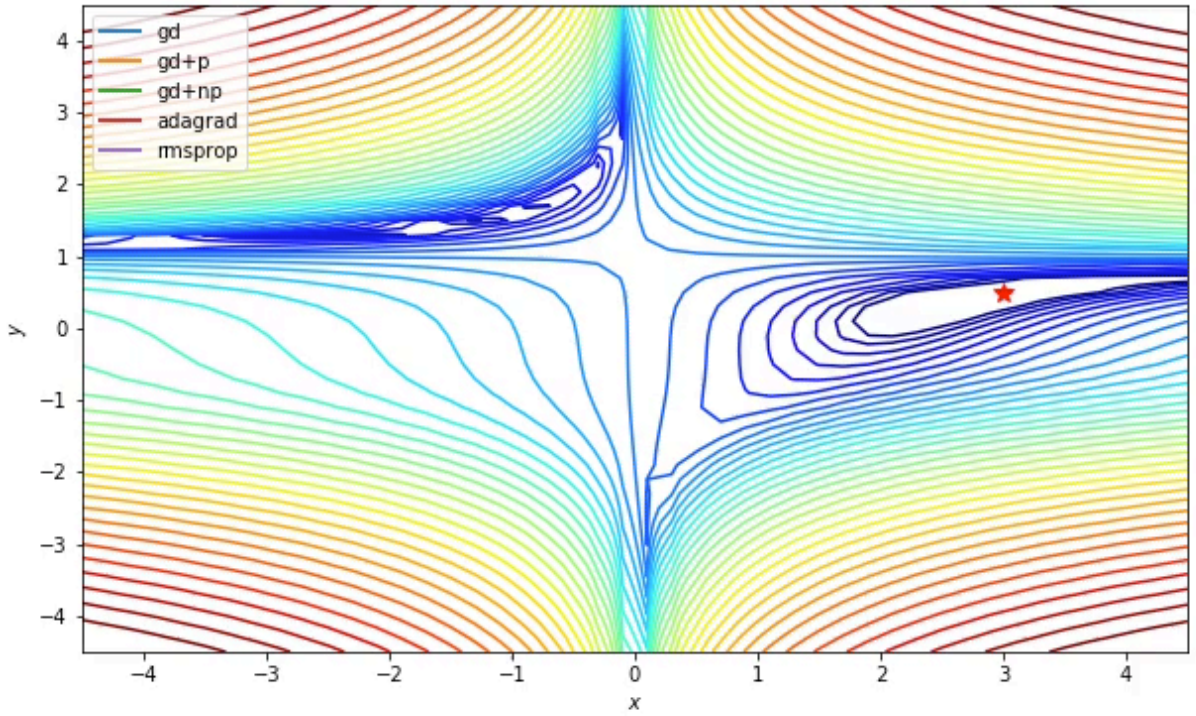
```
a = 0
beta = 0.99
nu = 1e-7
while last_diff > tol:
    cost, g = func(x)
    a = beta * a + (1-beta) * g * g
    x -= eps * g / (np.sqrt(a + nu))
    last_diff = np.linalg.norm(x - path[-1])
```

RMSProp (opt 1)



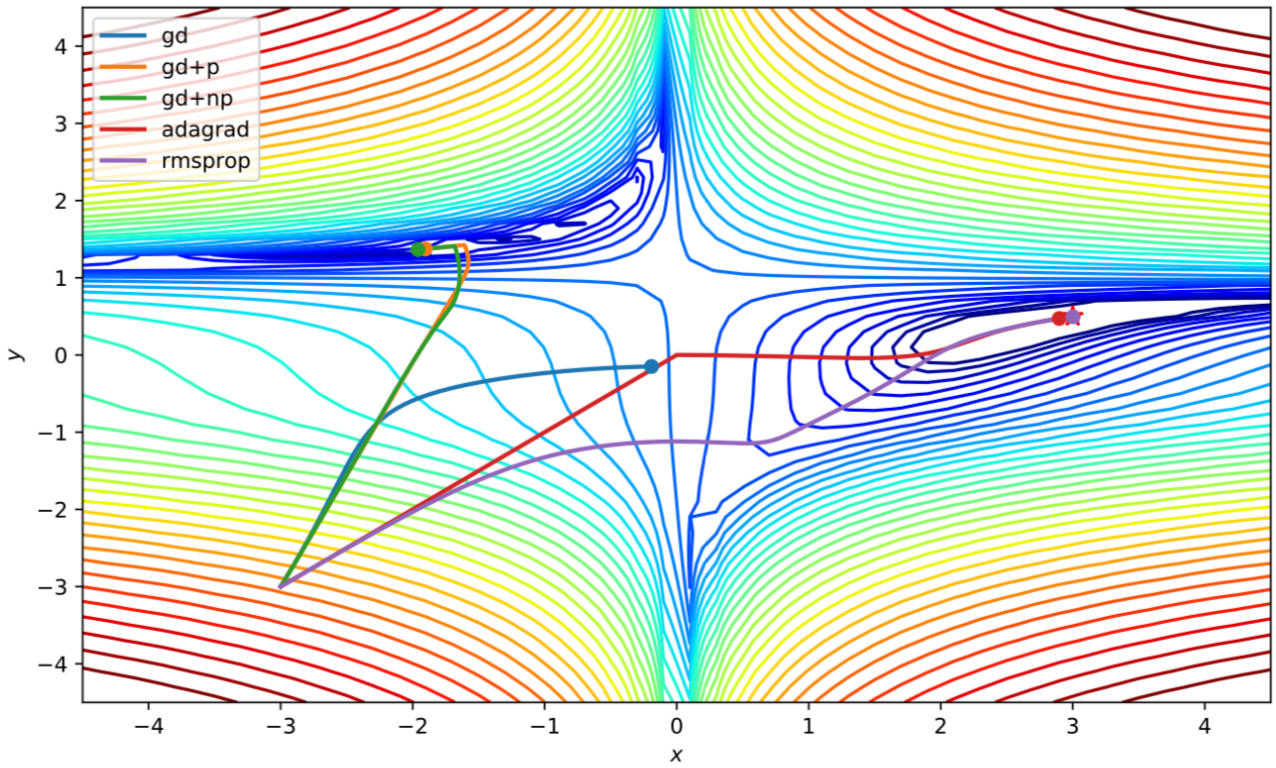
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop.mp4`

RMSProp



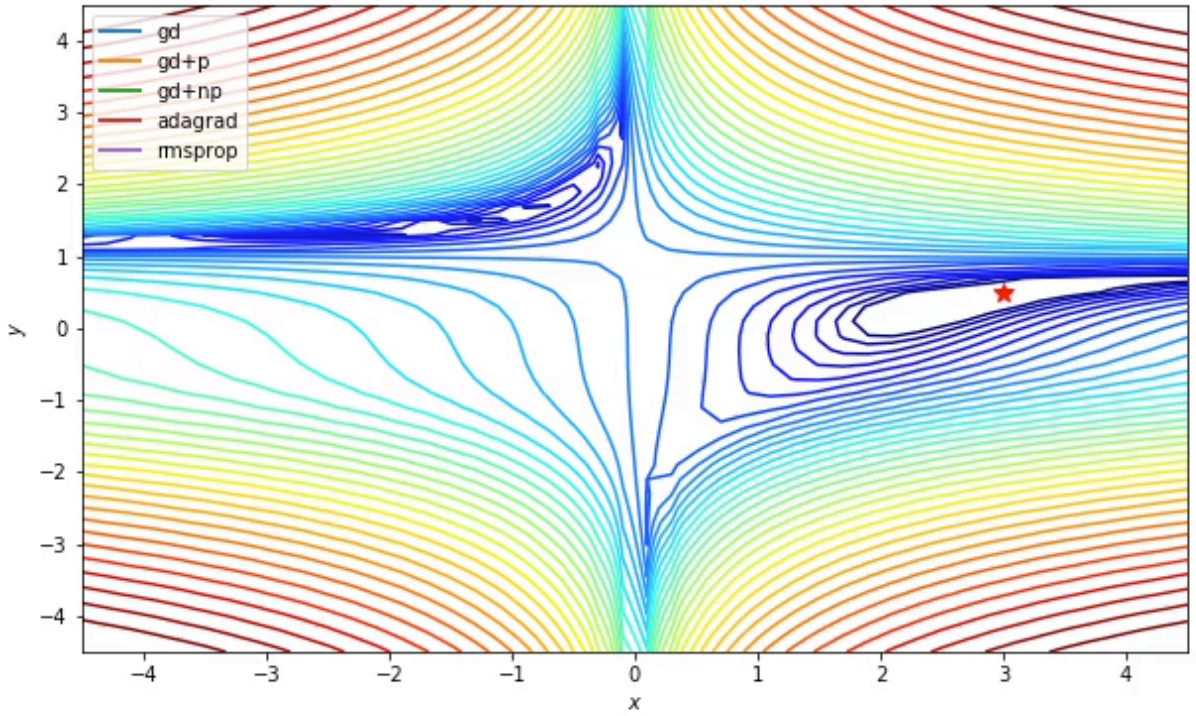
RMSProp (opt 2)

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:

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

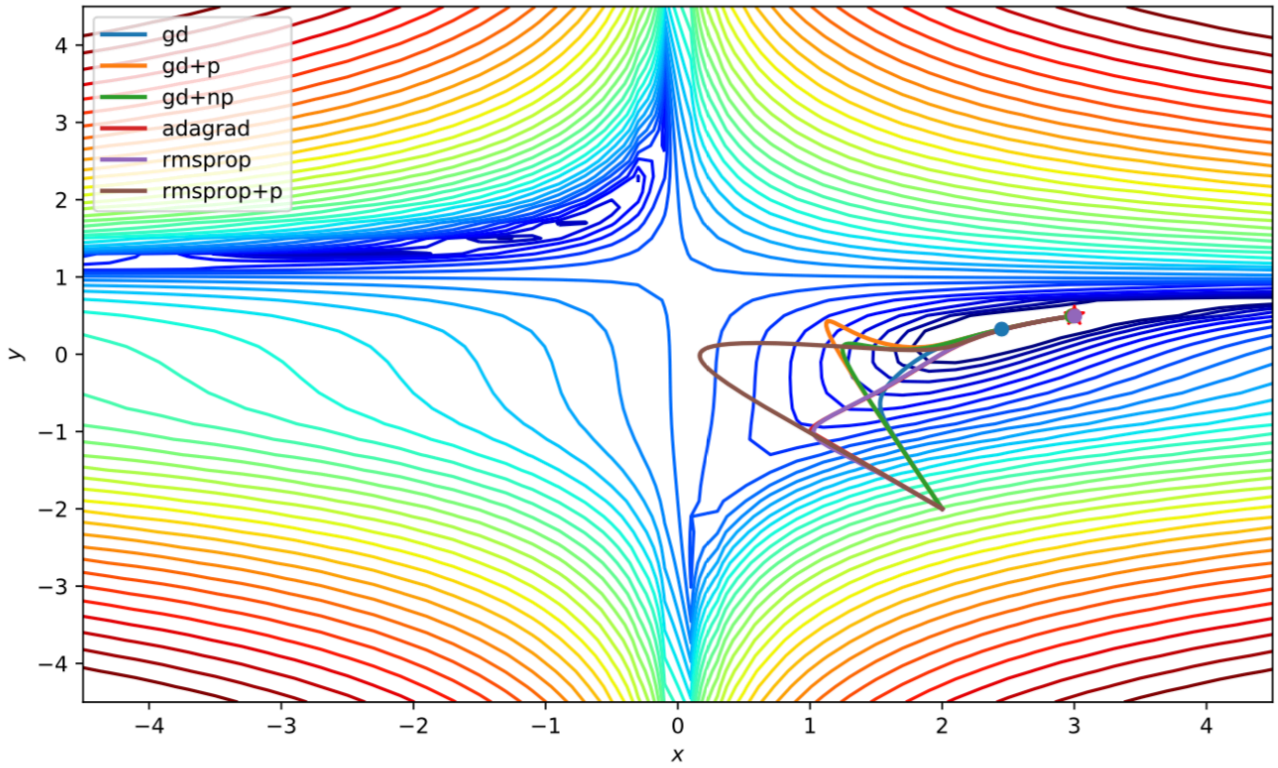
It is also possible to RMSProp with Nesterov momentum.

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])
```

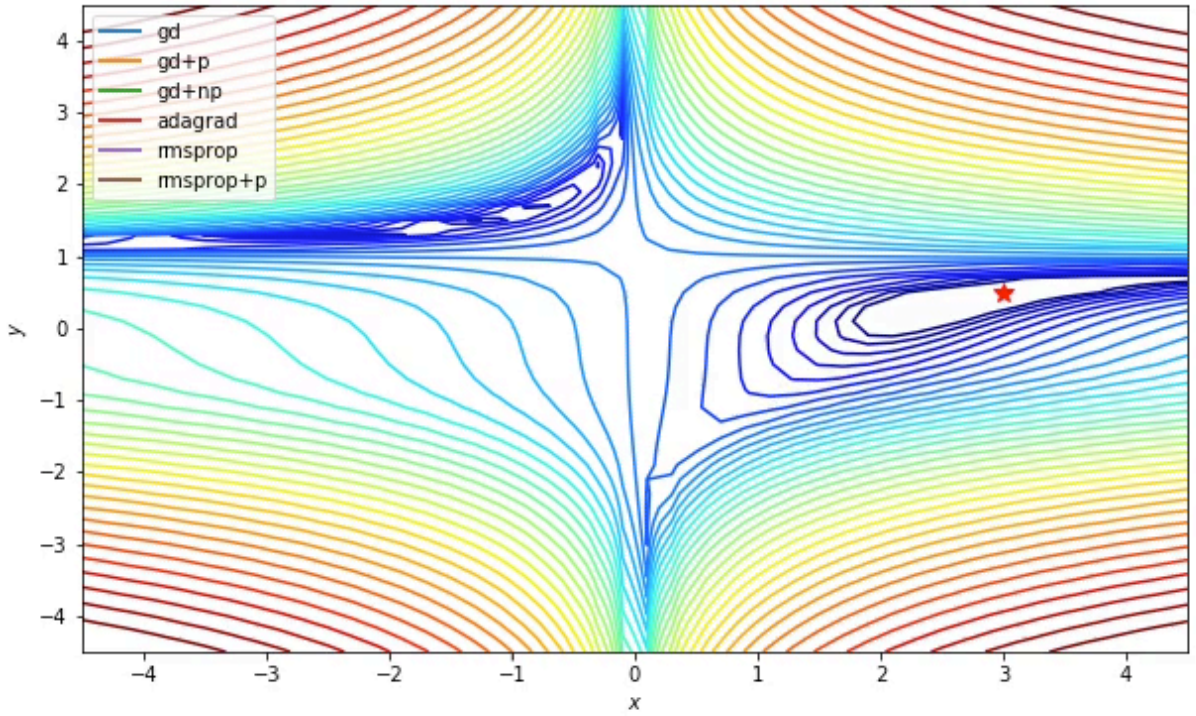
RMSProp with momentum (opt 1)

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.



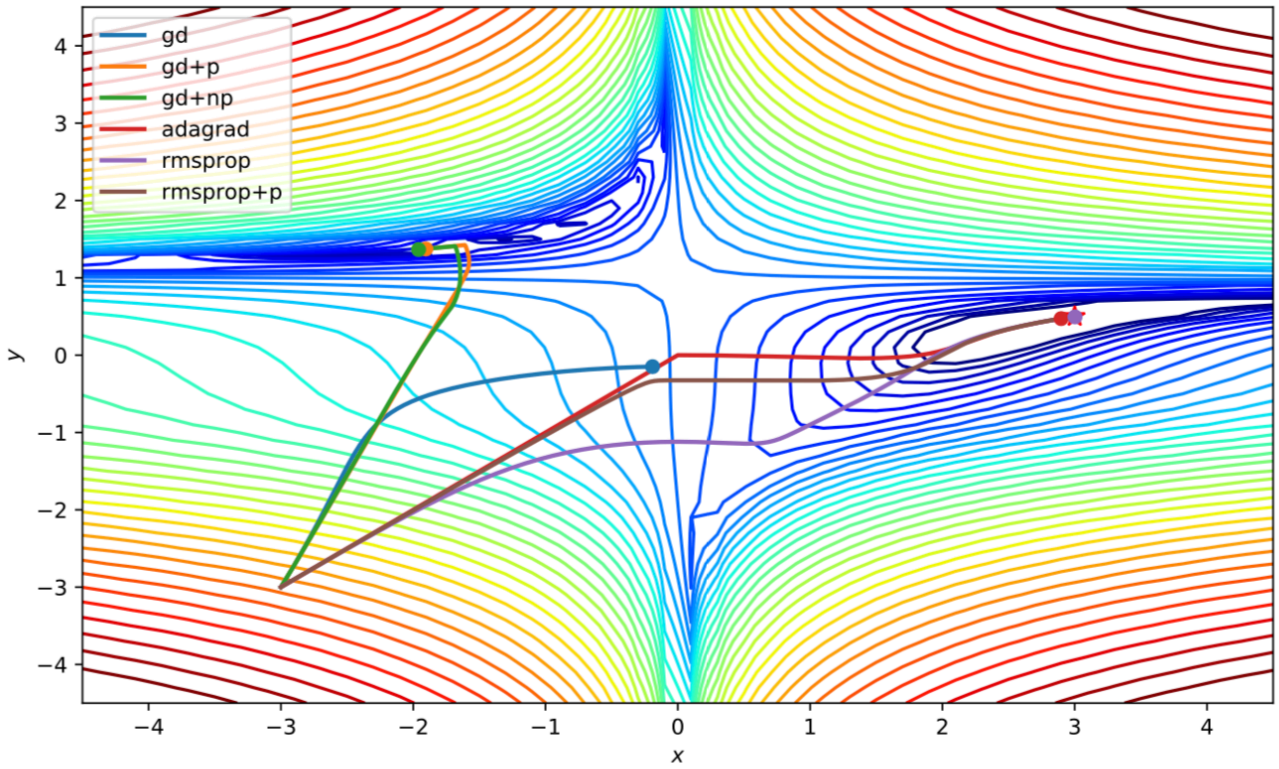
Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p.mp4`

RMSProp + momentum



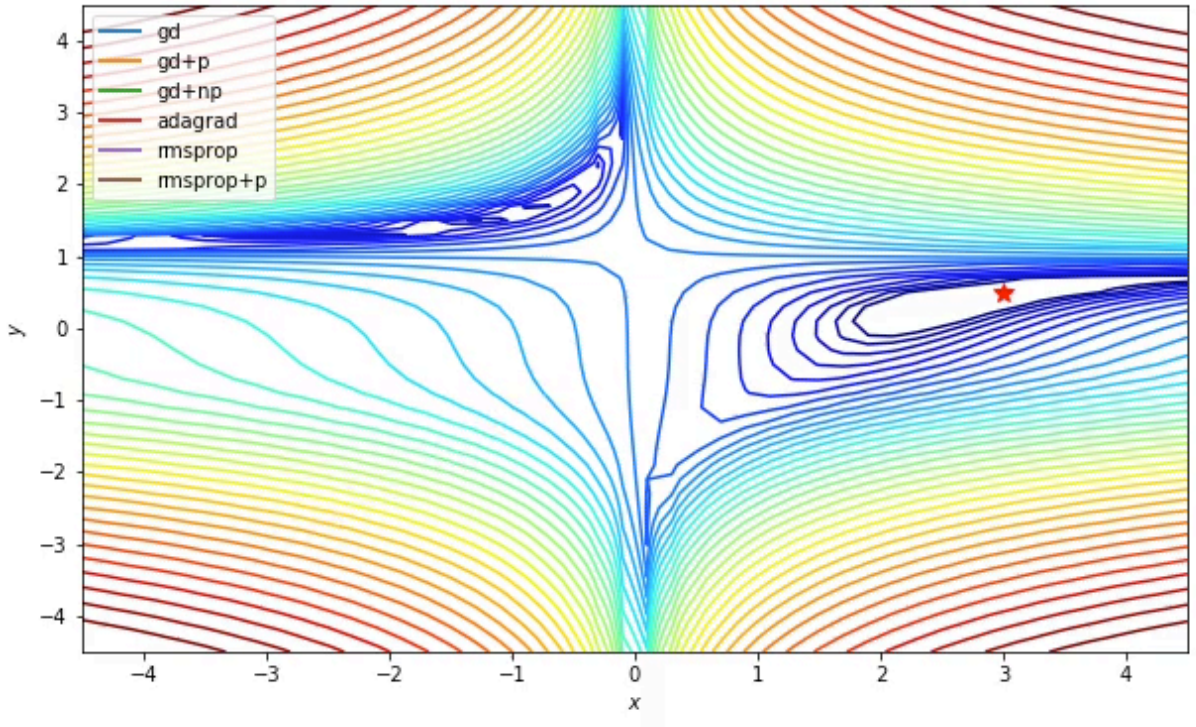
RMSProp (opt 2)

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):

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

- Second moment update (gradient normalization):

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

- 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:

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

$t \to \infty$

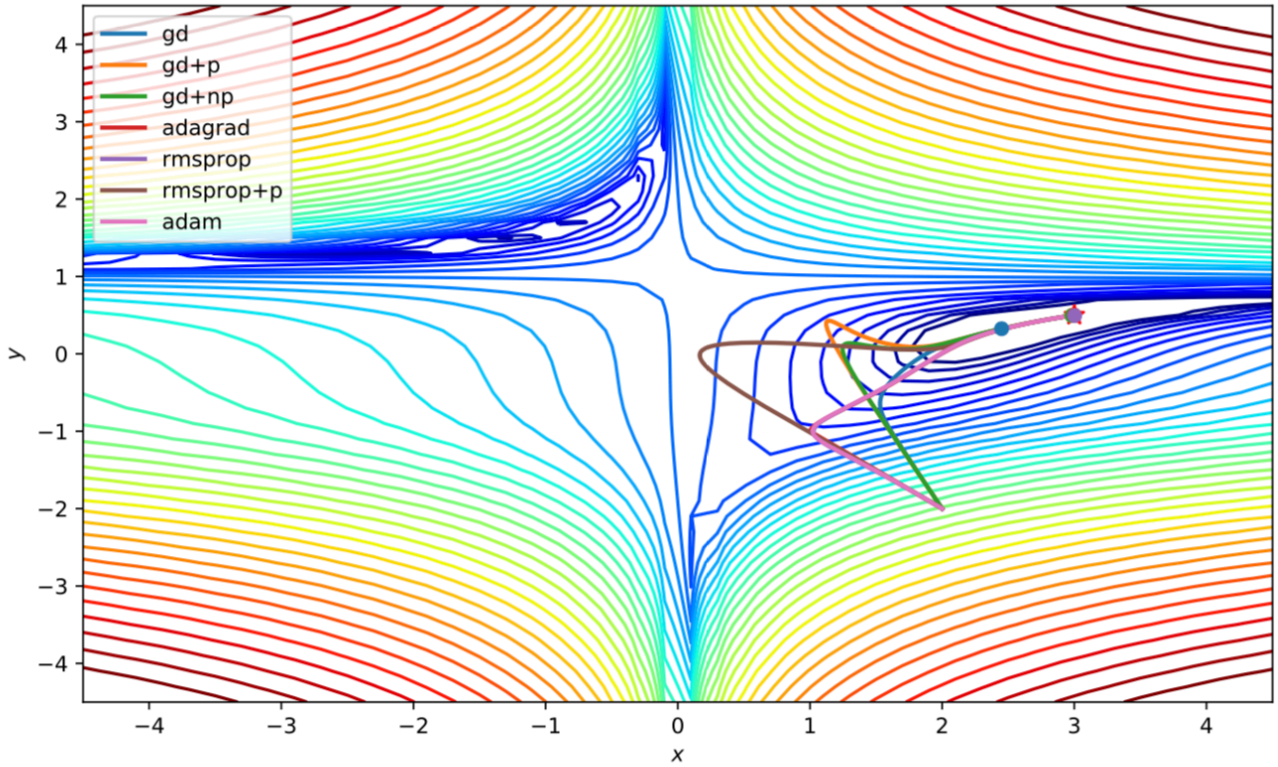
$t = 3$

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])
```

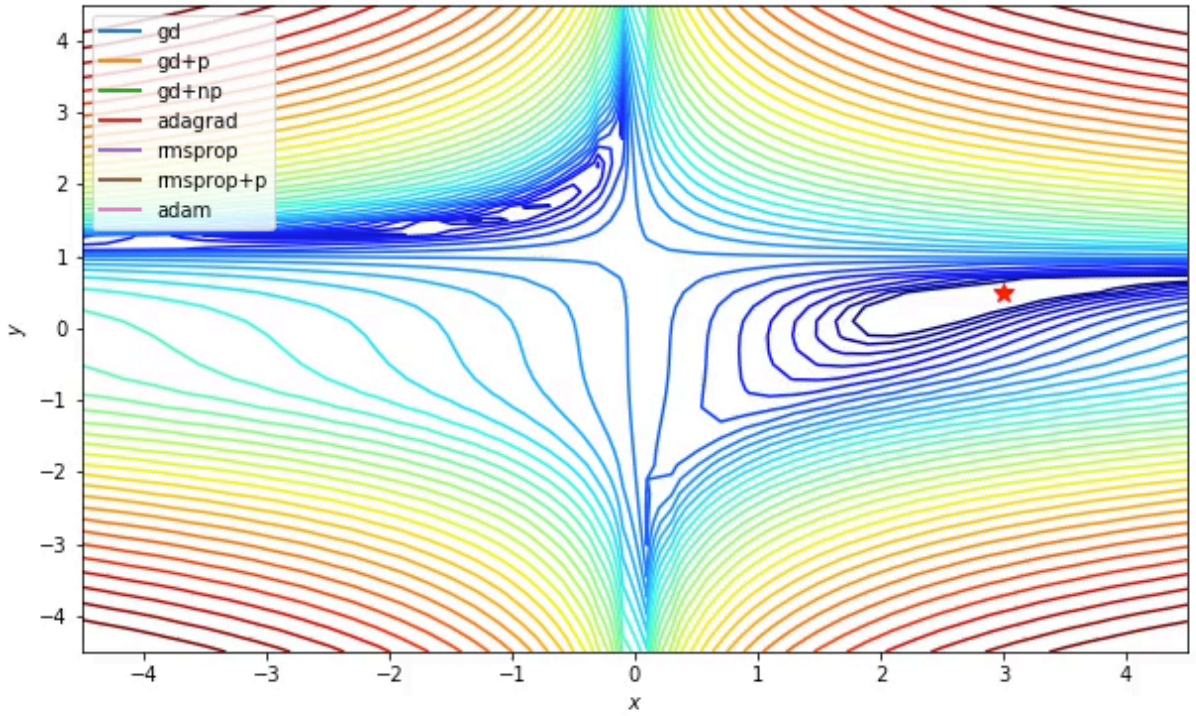
Adam (opt 1)

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

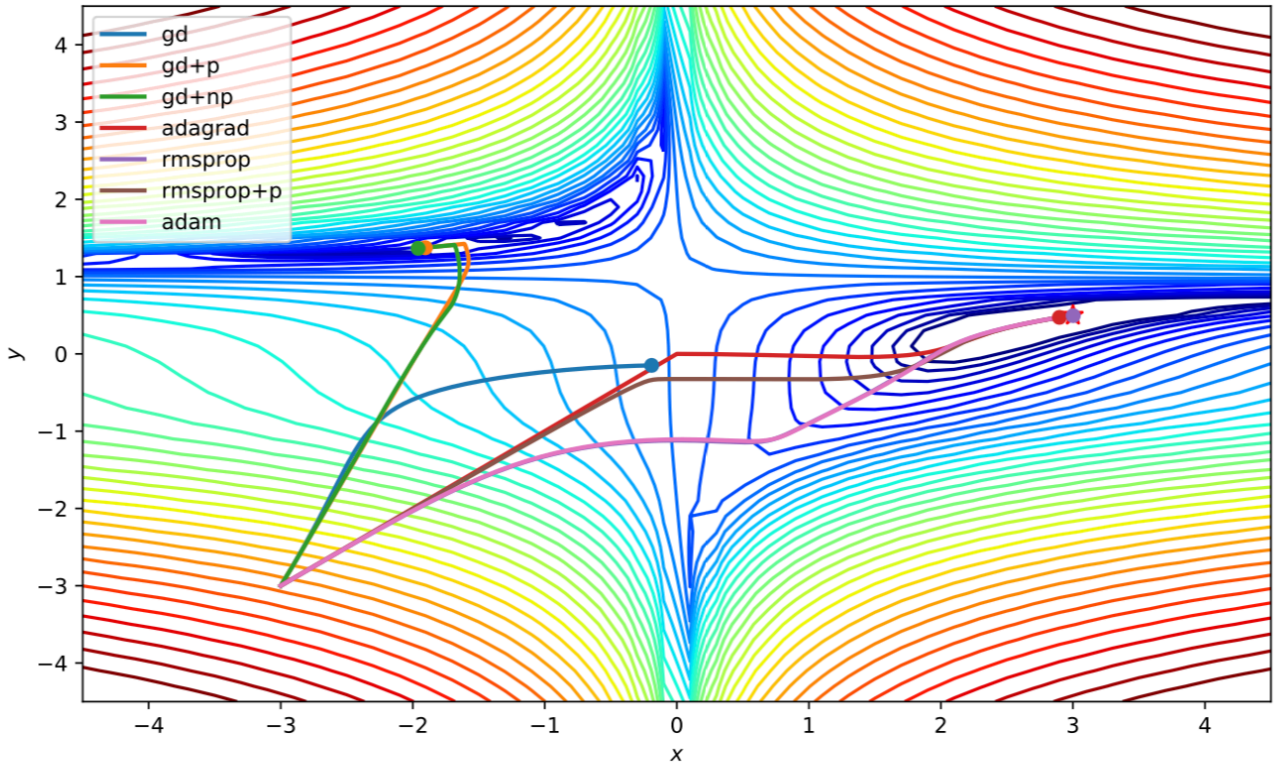


Video: `http://seas.ucla.edu/~kao/opt_anim/1gd_gd+p_gd+np_adagrad_rmsprop+p_adam.mp4`

Adam

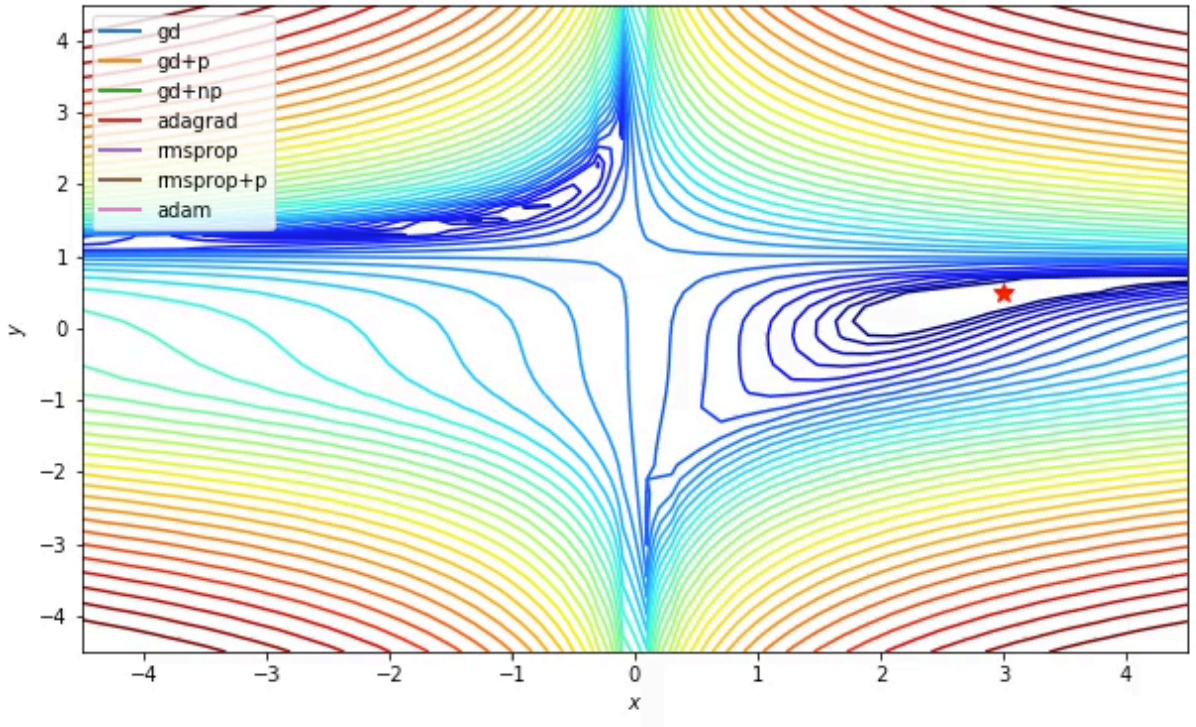

gd
gd+p
gd+np
adagrad
rmsprop
rmsprop+p
adam
y
x
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

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