Exercise 10

Machine Learning I

10A-1.

As the gradient vector is always orthogonal to paths of constant value (contour lines or more formally “Niveaumengen”), it helps to first plot a contour map of the target function:

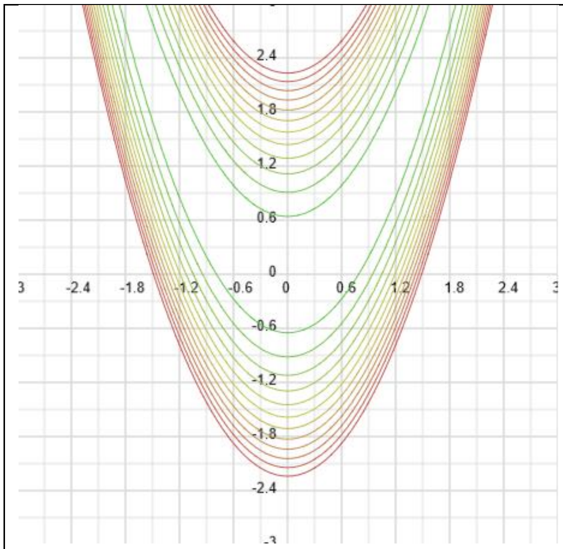


Figure 1 Contour lines of the banana function.

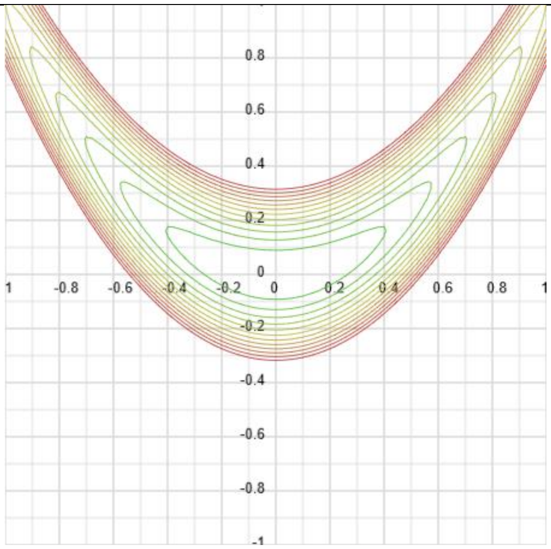
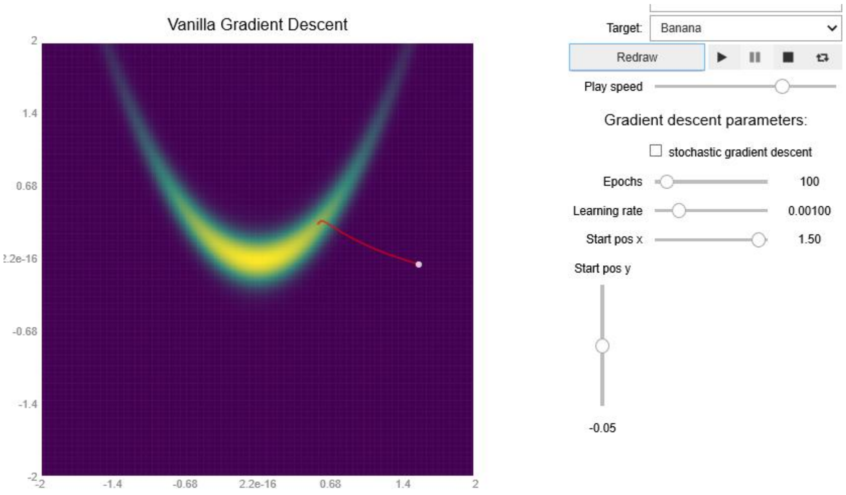


Figure 2 Close up of the contour lines of the banana function. As you can see, the interior contour lines also have the shape of a banana. This should lead to more oscillating behavior in vanilla gradient methods.

As we can see, the contour lines around zero are not very well behaved. Theoretically, this should affect the performance of pure gradient methods such as vanilla gradient descent. However, as the function is convex and $\nabla banana$ is Lipschitz, there will be step sizes for which *every* gradient descent algorithm will converge toward its minimum. But let us see!

Vanilla Gradient descent

Let us first look what happens if we only use the standard parameters:



Target: Banana

Play speed

Gradient descent parameters:

□ stochastic gradient descent

Epochs 100

Learning rate 0.00100

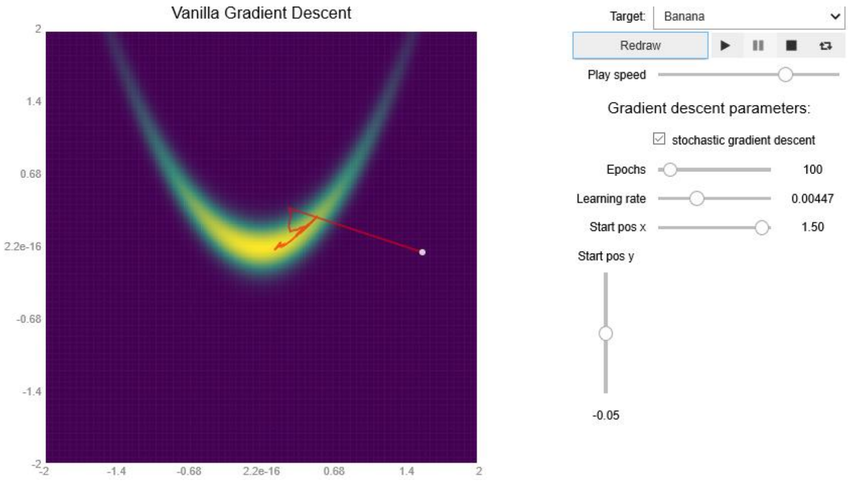
Start pos x 1.50

Start pos y -0.05

Figure 3 As expected, vanilla gradient descent follows the directions orthogonal to the contour lines. This means it first approaches the target from the outside (i.e. is orthogonal to the outside contour lines), until it reaches the upper corners of the interior "banana contour lines", where it will then change direction.

The contour lines near the starting points do not point towards the minimum. This is reflected in the performance. To alleviate the problem, it makes sense to reduce the dependency on the true gradients by selecting "stochastic gradient descent". Also, we are increasing the step size a little, as contour lines near our starting point are similar, i.e. the gradient will also be similar. Note that this does not consistently deliver good results but works well enough for me on average (see Figure 4).

In order to find locations, for which our algorithm does not converge, it makes sense to now classify contour as "good" or "bad". Good thereby means, that vectors normal to the contour lines generally point towards the maximum and bad means that they point somewhere else.



Target: Banana

Play speed

Gradient descent parameters:

☑ stochastic gradient descent

Epochs 100

Learning rate 0.00447

Start pos x 1.50

Start pos y -0.05

Figure 4 By selecting "stochastic gradient descent" and adjusting the learning rate, we are able to get close enough to the origin.

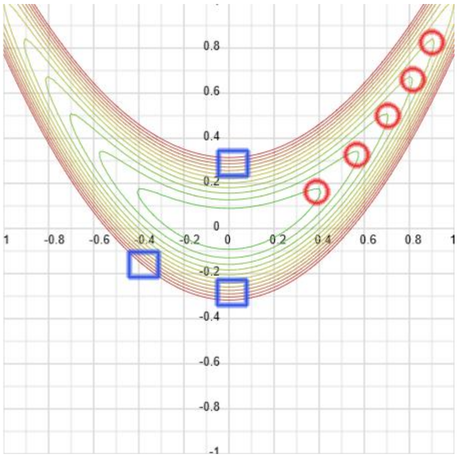
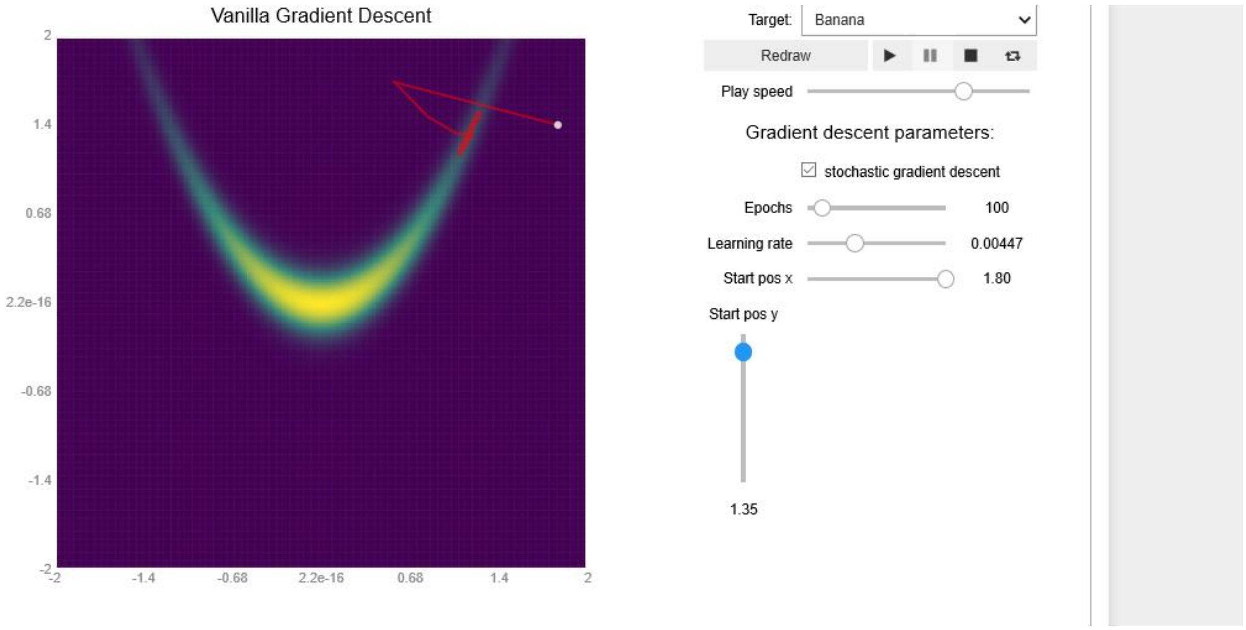


Figure 5 The edges of the interior contour lines (red dots) are responsible for the chaotic behavior if we approach the banana function from the top. The blue regions (square) behave well and generally point towards the minimum.

The edges (red dots) are a problem for our gradient descent, as they deflect the gradient. It is like a Pinball machine. The blue regions are better behaved. Our assumption is validated by changing the initial position to be once on the top right (and the gradients thereby traveling through the red circles) and once on the lower right (thus the gradients passing the well-behaved blue squares).



Target: Banana

Gradient descent parameters:

☑ stochastic gradient descent

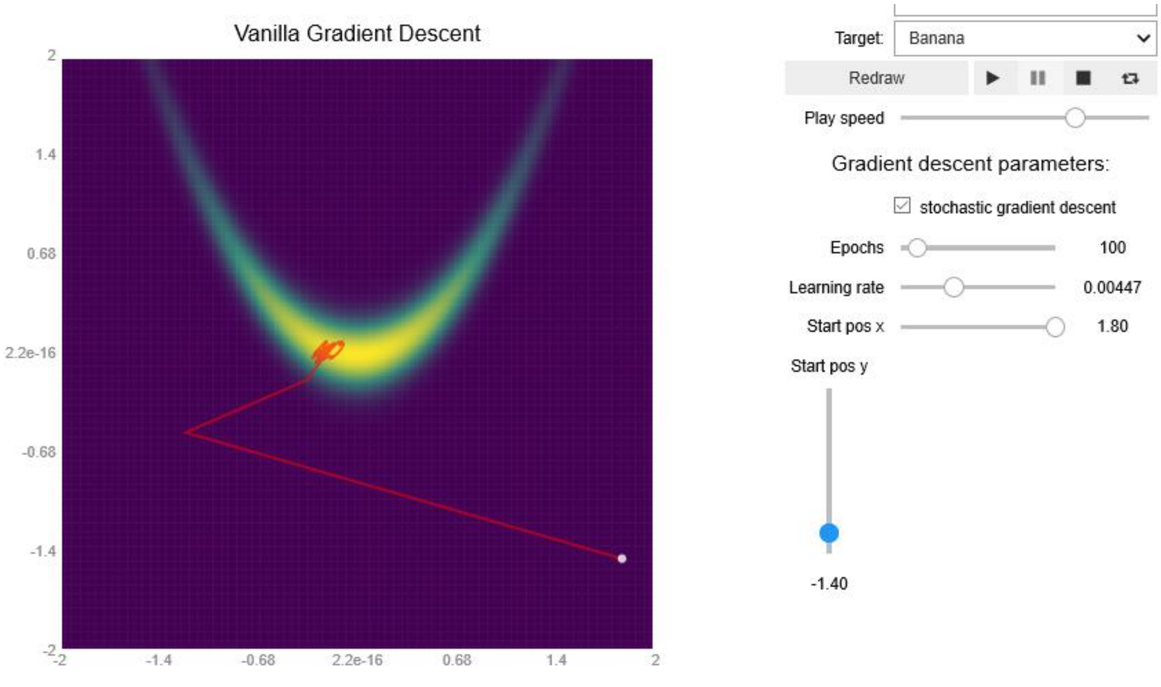
Epochs 100

Learning rate 0.00447

Start pos x 1.80

Start pos y 1.35

Figure 6



Target: Banana

Play speed

Gradient descent parameters:

☑ stochastic gradient descent

Epochs 100

Learning rate 0.00447

Start pos x 1.80

Start pos y -1.40

Figure 7

The important takeaway is, that the learning rate does not only depend on the distance towards the origin but also the path of contour lines.

Momentum

As we now try augment our vanialla gradient with a velocity component, it makes sense to look at the geometry of the banana function.

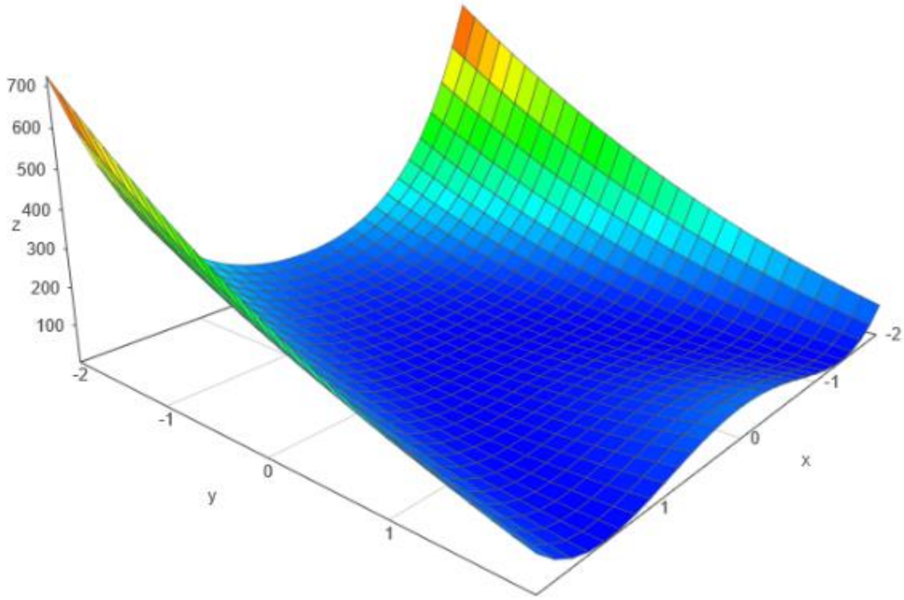
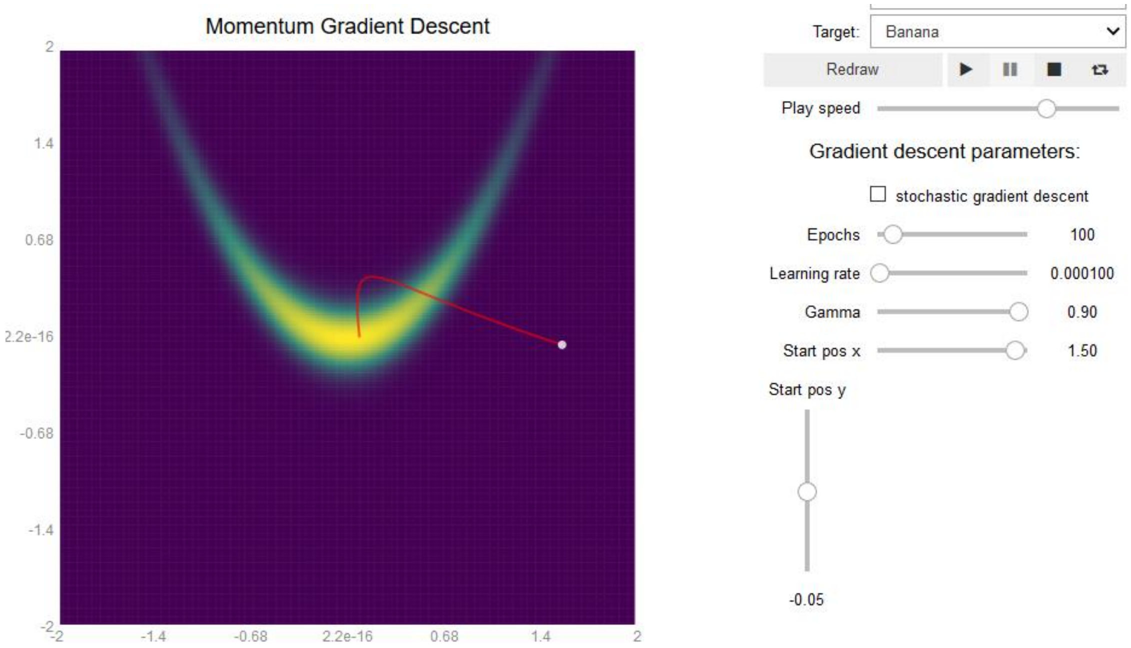


Figure 8

We immediately notice regions which would be ideal starting points if our optimizer simulated a true momentum-based model. In addition, as we require good information about banana's surface, it makes sense to work with the true gradient (e.g. turn off "stochastic gradient descent"). By nearly completely omitting the current gradient and working with velocity only, we receive a good result.



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.000100

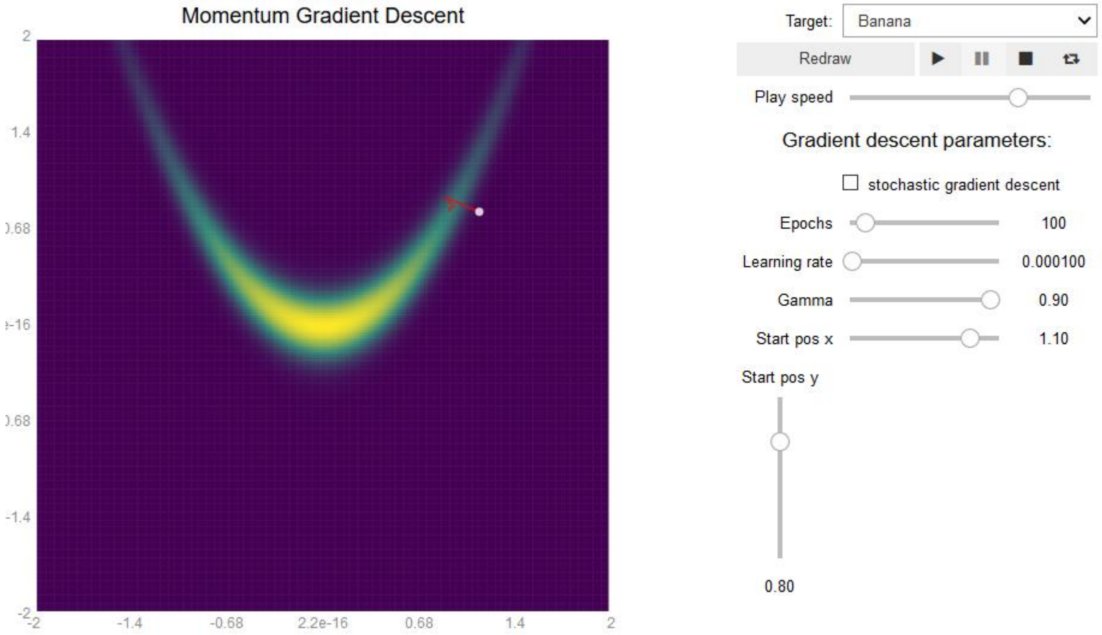
Gamma 0.90

Start pos x 1.50

Start pos y -0.05

Figure 9

This configuration is very dependent on the starting gradient, however. Bad initial placement throws off our path.



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.000100

Gamma 0.90

Start pos x 1.10

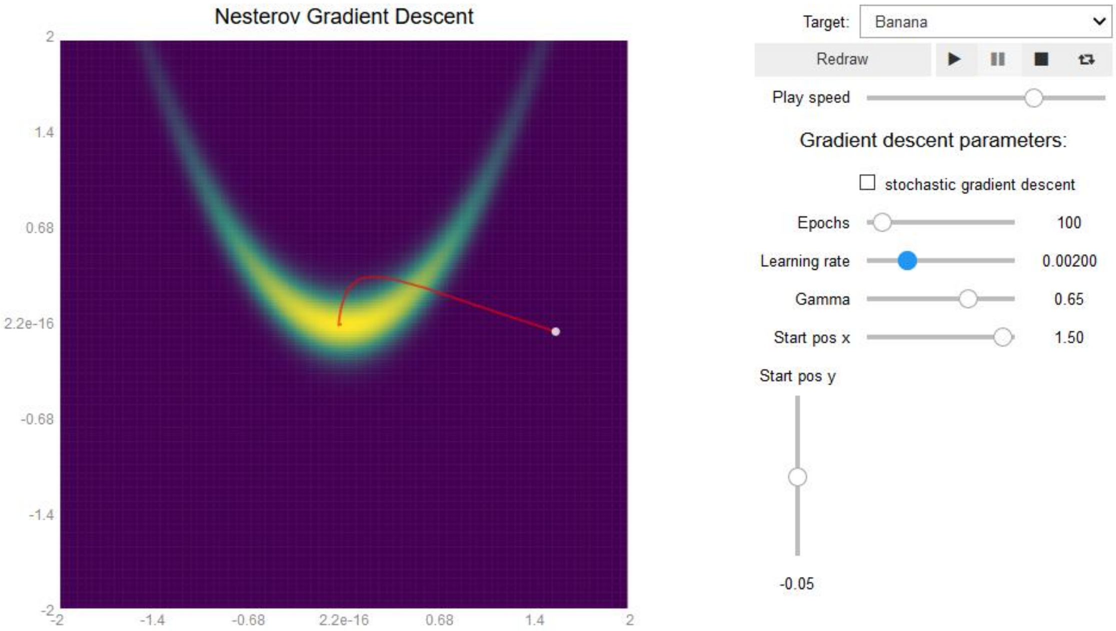
Start pos y 0.80

Figure 10

Nesterov

Instead of looking at the current gradient, we take into account the geometry at the updated position (if the updated position were calculated with standard momentum gradient descent).

This allows us to artificially “damp” our ball, if the geometry suddenly changes (which would imply a minimum between according to the *Intermediate Value theorem*). This is only a valid assumption, if f satisfies certain convexity and smoothness requirements. As we found out at the beginning, however, the banana function meets these requirements.



Target: Banana

□ stochastic gradient descent

Epochs 100

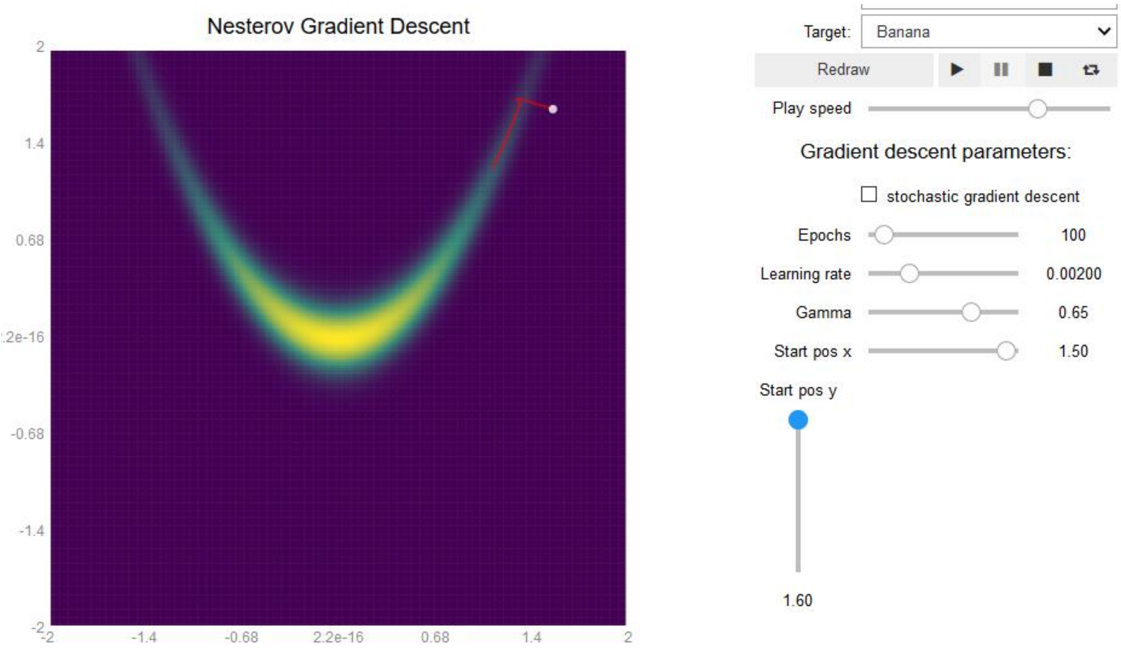
Learning rate 0.00200

Gamma 0.65

Start pos x 1.50

Start pos y -0.05

Figure 11 The learning rate is now higher than in the pure momentum-based model. This is due to banana’s geometry.



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.00200

Gamma 0.65

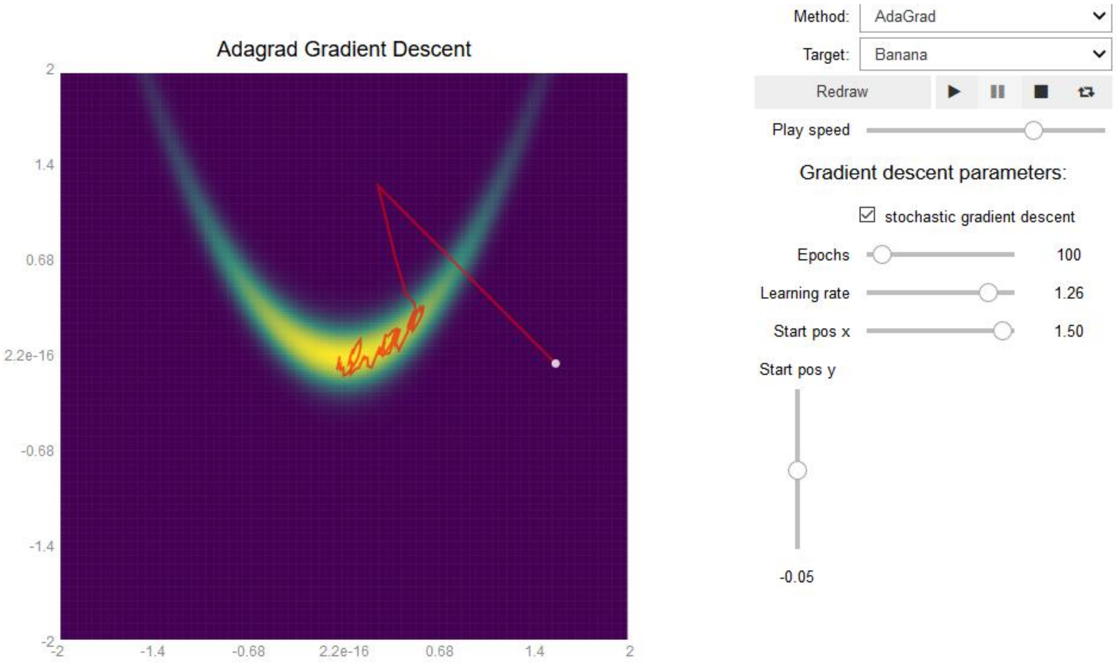
Start pos x 1.50

Start pos y 1.60

Figure 12 As all the other methods, if the contour lines vary greatly gradient descent will perform worse.

Adagrad

Adagrad looks at past gradients in order to dynamically adjust the step size. This works nicely in theory. If many gradients are junk, however, the step size will vary wildly. As previously found out, our initial starting point is near “bad” contour lines. Therefore, we will reduce the importance of the geometry by selecting “stochastic gradient descent”. If we now also start with a good global learning rate, we are able to reach our minimum within 100 epochs on average.



Method: AdaGrad

Target: Banana

Play speed

Gradient descent parameters:

☑ stochastic gradient descent

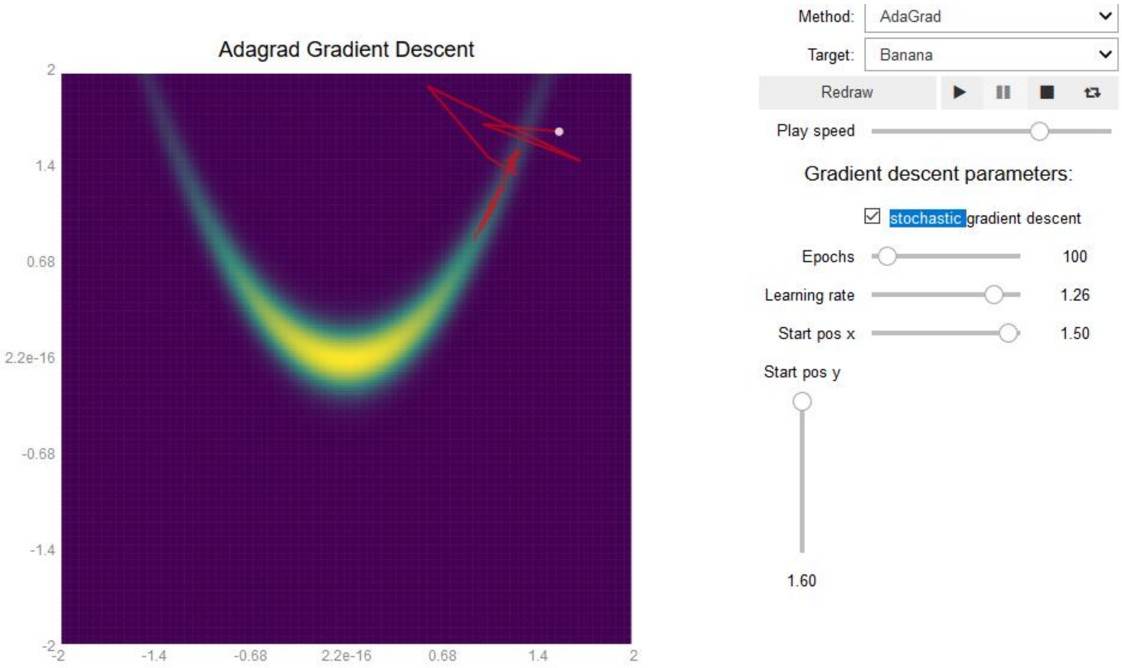
Epochs 100

Learning rate 1.26

Start pos x 1.50

Start pos y -0.05

Figure 13



Method: AdaGrad

Target: Banana

Play speed

Gradient descent parameters:

☑ stochastic gradient descent

Epochs 100

Learning rate 1.26

Start pos x 1.50

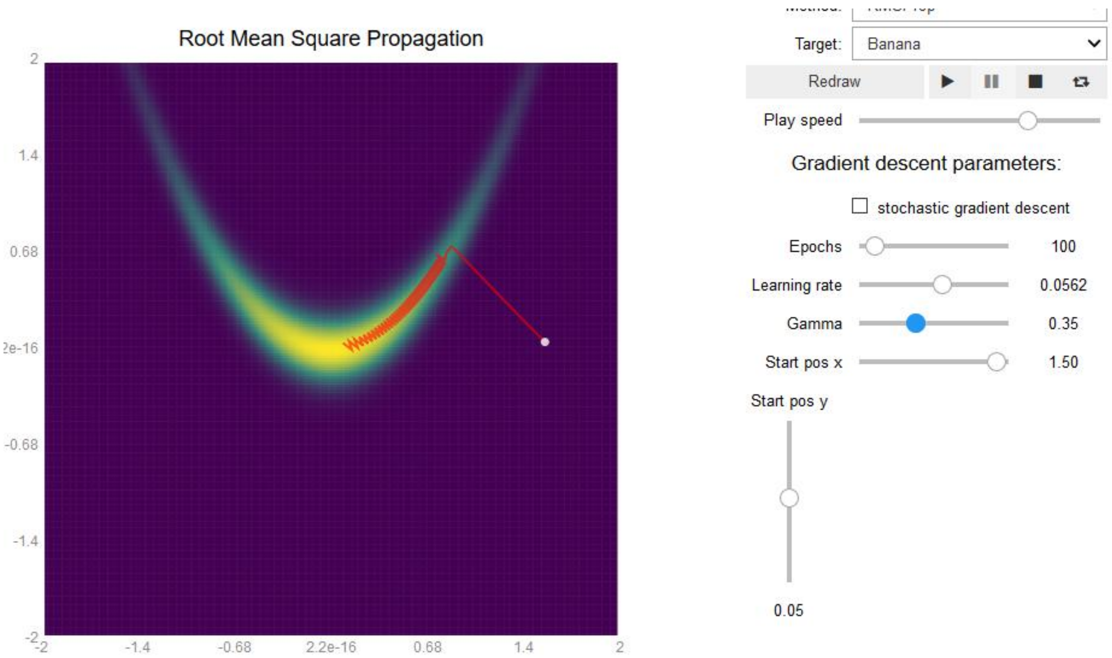
Start pos y 1.60

Figure 14 As expected, performance greatly decreases near the bad contour lines.

RMSProp

RMSProp (which sounds like a boat, see RMS Titanic) uses a running average of past gradients. This way, current geometry is more relevant than past geometry. Interestingly, it is really robust! For many different values of γ, λ we still get a good result. Thanks, Hinton!

If we approach from the top right, we still get our “Pinball Machine Path”. But beware: It may not look like it, but the algorithm is still stable (Pinball paths can be stable, look up e.g. A-Stability for ODE’s)!



Target: Banana

□ stochastic gradient descent

Epochs 100

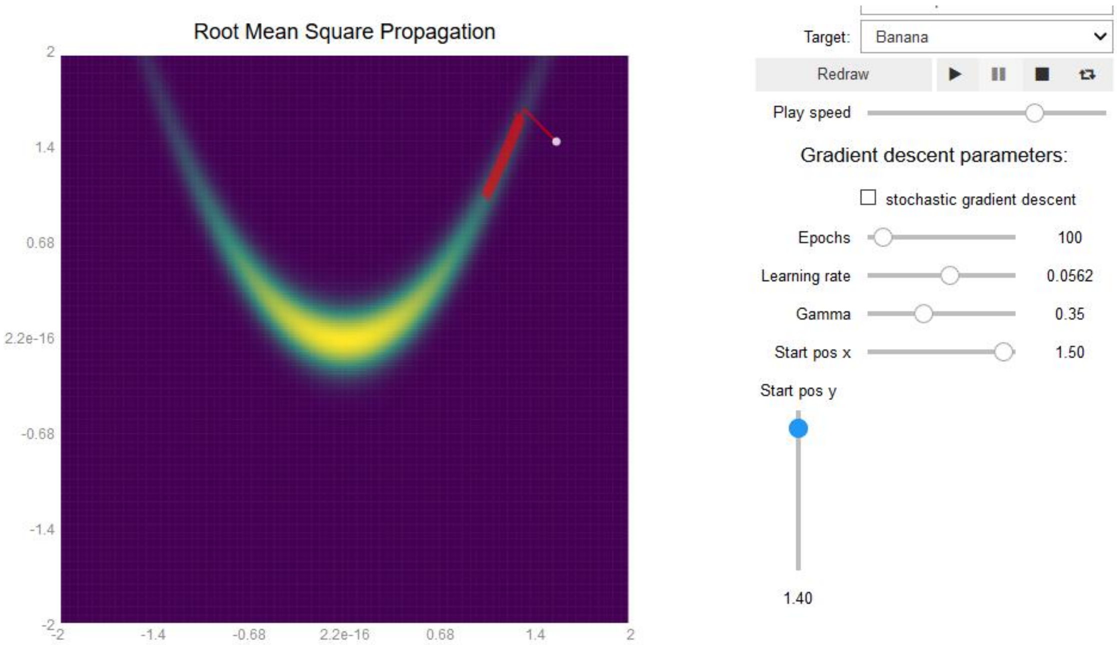
Learning rate 0.0562

Gamma 0.35

Start pos x 1.50

Start pos y 0.05

Figure 15



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.0562

Gamma 0.35

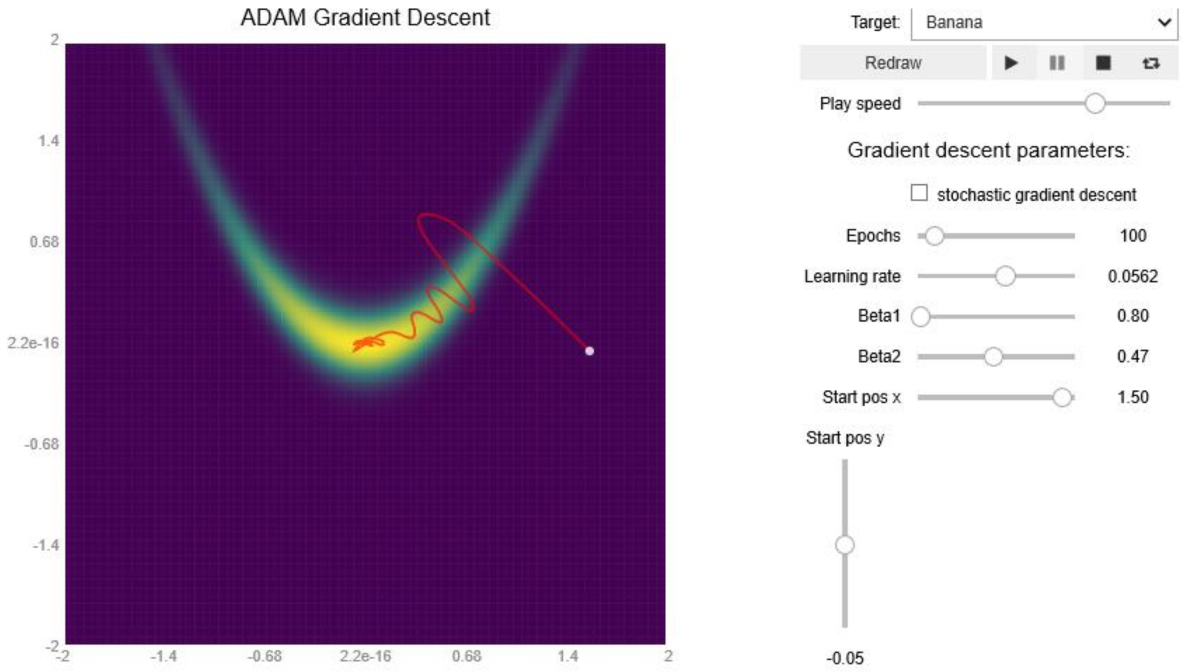
Start pos x 1.50

Start pos y 1.40

Figure 16 Our Pinball machine is still alive and kicking with RMSProp.

Adam

A worthwhile approach is to lower the values of β_0, β_1 to decrease the significance of past gradients. We do this to mimic the effects RMSProp, where we set $\gamma = 0.35$ (which is pretty low).



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.0562

Beta1 0.80

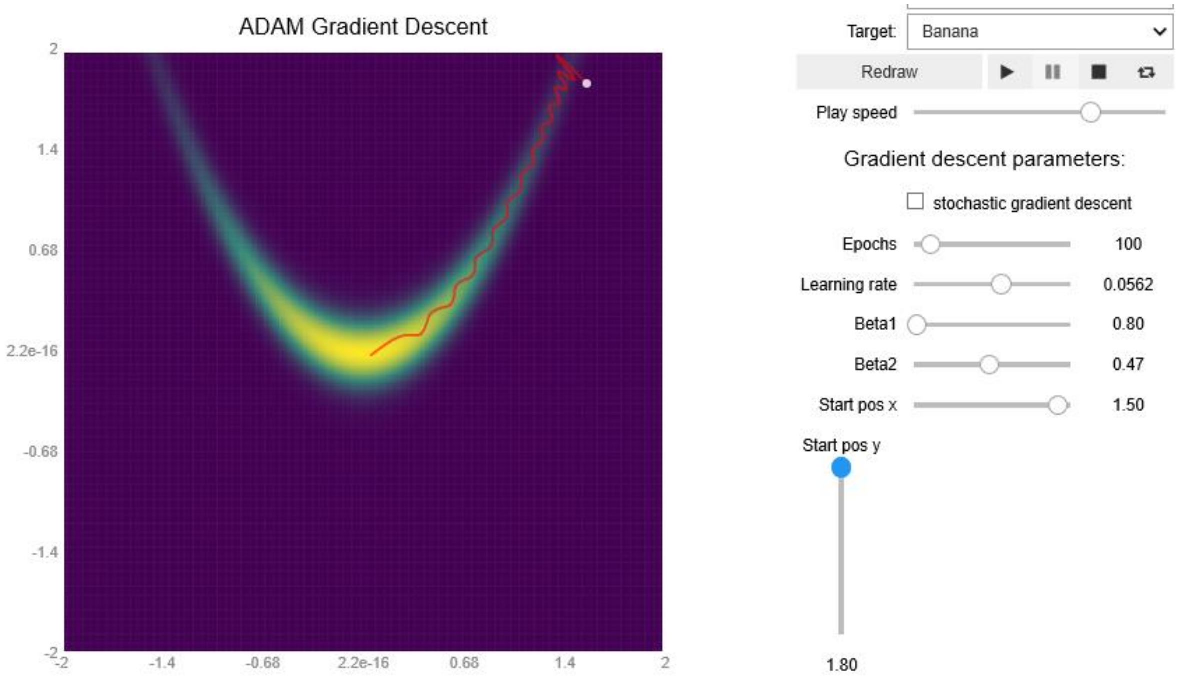
Beta2 0.47

Start pos x 1.50

Start pos y -0.05

Figure 17 Even though it looks like it is orbiting around the origin it is probably is converging.

But here comes the shocker: If we change our initial position, the model is still robust! Even from our worst-case placement, where all our previous methods failed!



Target: Banana

□ stochastic gradient descent

Epochs 100

Learning rate 0.0562

Beta1 0.80

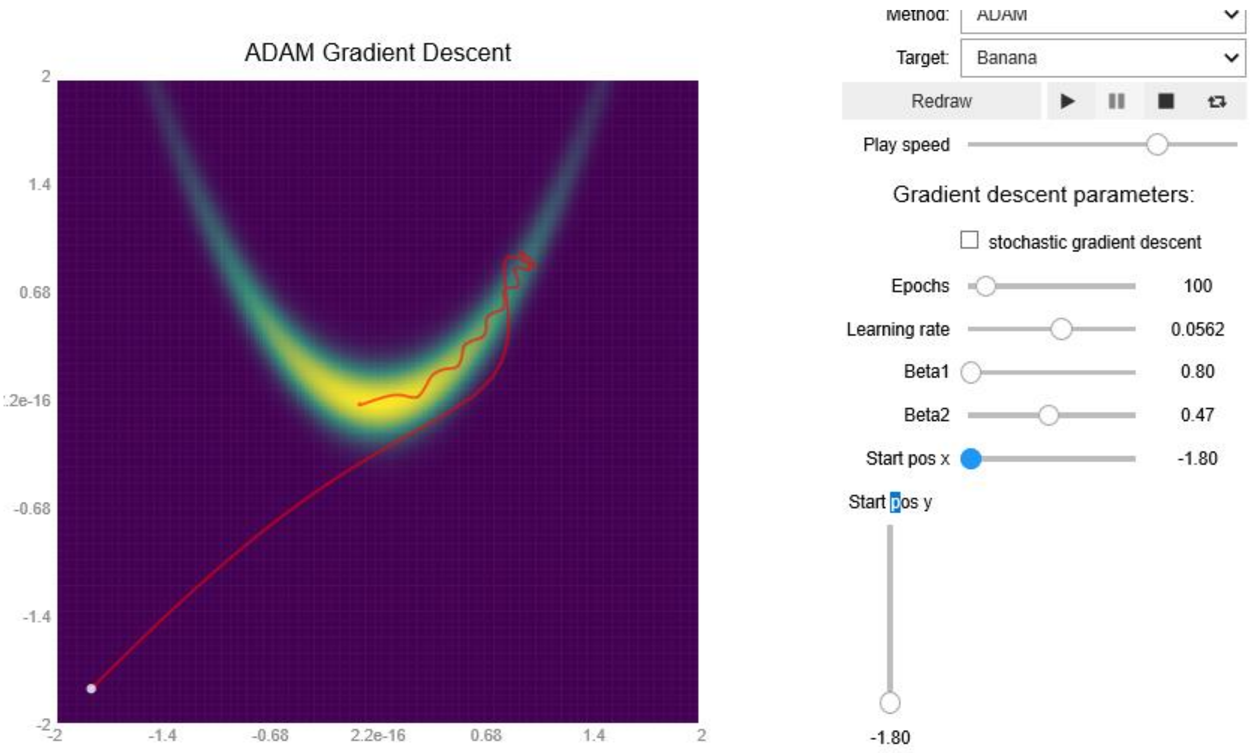
Beta2 0.47

Start pos x 1.50

Start pos y 1.80

Figure 18 Using the same parameters as before, even works from our "Pinball corner".

In fact, Adam is so robust, that I can’t find any placement of the initial values where it does not converge to the minimum!



Method: ADAM

Target: Banana

Gradient descent parameters:

□ stochastic gradient descent

Epochs 100

Learning rate 0.0562

Beta1 0.80

Beta2 0.47

Start pos x -1.80

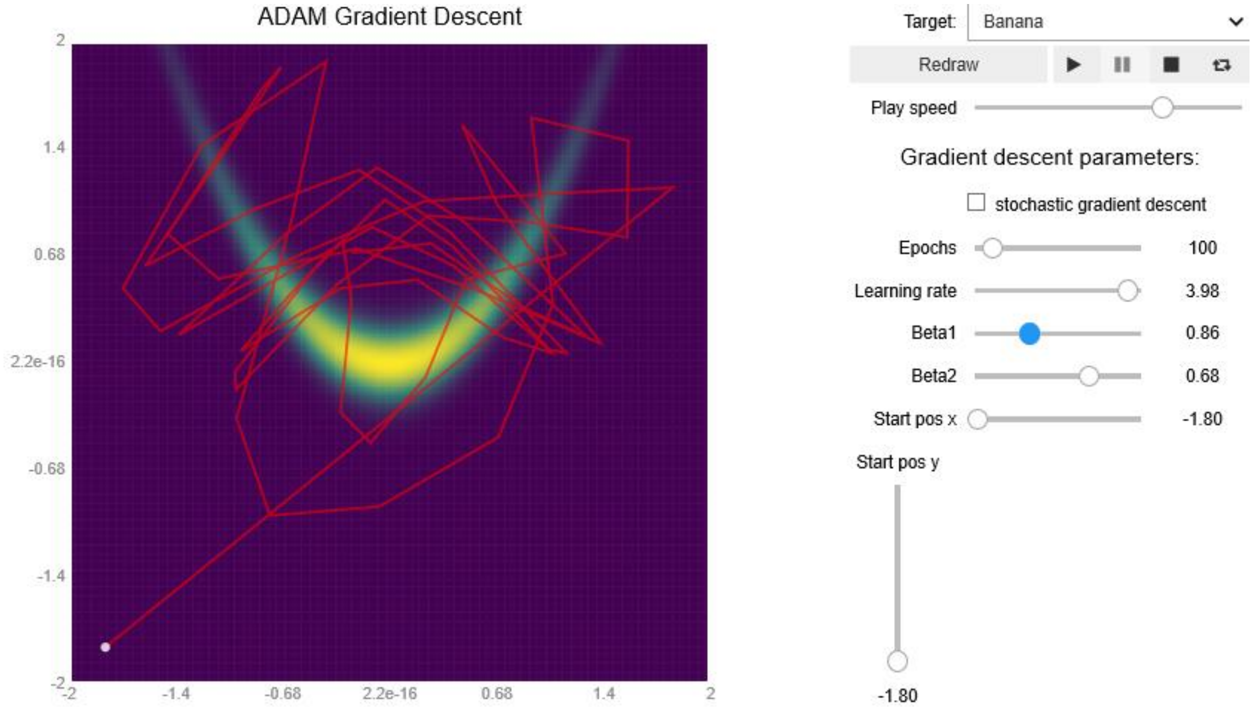
Start pos y -1.80

Figure 19 Lower left corner? No problem for Adam!

Adam is of course not invincible: If we put the starting point too far away from the origin, then it won’t work in 100 epochs. But that is to be expected for any step-based method.

Nevertheless, this does not automatically make Adam superior. Compared to other methods, it has more parameters. And it is very sensitive to changes to them (Figure 20).

So here the question of finding $\beta_0, \beta_1, \lambda$ is even more urgent. How would you approach it?



Target: Banana

Play speed

Gradient descent parameters:

□ stochastic gradient descent

Epochs 100

Learning rate 3.98

Beta1 0.86

Beta2 0.68

Start pos x -1.80

Start pos y -1.80

Figure 20 Different parameters give rise to different paths.

10A-2.

We are trying to minimize our banana function using Newton's method. As $f\colon \mathbb{R}^m \to \mathbb{R}$, we have the following iteration rule:

$$\mathbf{x}_{t+1} = \mathbf{x}_n - \left(\mathbf{H}(\mathbf{x}_\mathrm{t})\right)^{-\mathbf{1}} \nabla f$$

Where $\mathbf{H}$ is the Hessian. This follows the notation from here: https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization#Higher_dimensions

It is numerically more stable (and quicker) to solve the following system of equations:

$$\mathbf{H}(\mathbf{x}_\mathrm{t})(\mathbf{v}) = \nabla f$$

for $\mathbf{v} = \mathbf{x}_{t+1} - \mathbf{x}_n$. As $\mathbf{H}_{banana}$ is only 2×2, we can directly calculate the inverse.

Some auxiliary calculation leads to:

$$\nabla f = \begin{pmatrix} 2x - 80x(y - x^2) \\ y - x^2 \end{pmatrix}$$

$$\mathbf{H}(\mathbf{x}_\mathrm{t}) = \begin{pmatrix} -80y + 240x^2 + 2 & -80x \\ -80x & 40 \end{pmatrix}$$

$$\left(\mathbf{H}(\mathbf{x}_{\mathrm{t}})\right)^{-\mathbf{1}} = \frac{1}{40\cdot(-80y+240x^2+2)-6400x^2}\begin{pmatrix}40 & 80x\\ 80x & -80y+240x^2+2\end{pmatrix}$$

Because for-loops are not optimized in Python, we will first code the iteration in C++ and call the function from Python.

newton_banana.cpp

```
extern "C" void newton_banana(double* x){
        int iter=0;
        while(iter<100){
                double g1=(2*x[0]-80*(x[1]-x[0]*x[0])*x[0]);
                double g2=x[1]-x[0]*x[0];
                double temp1=40*g1+80*x[0]*g2;
                double temp2=80*x[0]*g1+(-80*x[1]+240*x[0]*x[0]+2)*g2;
                double norm=1/((-80*x[1]+240*x[0]*x[0]+2)*40-6400*x[0]*x[0]);
                x[0]=x[0]-norm*temp1;
                x[1]=x[1]-norm*temp2;
                iter=iter+1;
        }
}
```

source.py

```
import ctypes
import numpy.ctypeslib as ctl
import numpy as np

_sum = ctypes.CDLL('mypath’//newton_banana.so')
_sum.newton_banana.argtypes = [ctl.ndpointer(np.float64,flags='aligned, c_contiguous')]

def newton_banana(x_old):
   global _sum
   _sum.newton_banana(x_old)


x_0=np.array([2,0], dtype=np.float64)
newton_banana(x_0)
print(x_0)
```

What we can see, is that Newtons method converges way quicker than any of the gradient descent methods. In my case, sometimes only 10 iterations were enough to have a good approximation of the root.
Newtons method roughly converges quadratically with respect to $d_t = \|\mathbf{x}_{true} - \mathbf{x}_t\|^2$, i.e. $d_{t+1} \approx d_t^2$, $0 < d < 1$, faster than the gradient based methods in general.

In its most basic form it also has no parameters: A big advantage in high-dimensional spaces, where it is hard to fine tune parameters based on intuition.

The only downside is the need for $\mathbf{H}$. But quasi methods exist, that make use of finite-difference schemes or even automatic differentiation.

Less relevant but still noteworthy, is that Newtons method handles itself well with respect to units. If our functions satisfies Clairut's theorem of partials (which they mostly do), we see that:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left[\frac{f}{\partial x_j}\right] = \frac{\partial}{\partial x_j}\left[\frac{f}{\partial x_i}\right]$$

This leads to:

$$\mathbf{units}\,\frac{\partial}{\partial x_i}\left[\frac{f}{\partial x_j}\right] \quad = \frac{\mathbf{units}\ f}{\mathbf{units}\ x_i\ \mathbf{units}\ x_j}$$

As $\mathbf{H}^{-1}\mathbf{H} = \mathbf{H}\mathbf{H}^{-1} = \mathbf{I}$ and $\mathbf{I}$ is unitless, we get:

$$\begin{aligned}\mathbf{units}\ \mathbf{H}_{ij}^{-1} \quad &= \frac{1}{\mathbf{units}\ \mathbf{H}_{ij}} \\ &= \frac{\mathbf{units}\ x_i\ \mathbf{units}\ x_j}{\mathbf{units}\ f}\end{aligned}$$

For each row i, this leads to:

$$\begin{aligned}\mathbf{units}\ (\mathbf{H}\nabla f)_i \quad &= \sum_{j=1}^{n} \underbrace{\frac{\mathbf{units}\ x_i\ \mathbf{units}\ x_j}{\mathbf{units}\ f}}_{units\ H_{ij}} \cdot \frac{\mathbf{units}\ f}{\mathbf{units}\ x_j} \\ &= \mathbf{units}\ x_i\end{aligned}$$

10A-3.

Momentum / Vanilla:

We have:

$$\theta_{n+1} = \theta_n - \gamma \mathbf{v}_n - \lambda \nabla f$$

If we set $\gamma = 0$, we receive the same updates as normal gradient descent. You can try that out in the notebook!

Adam/RMSProp

$$\theta_{i,n+1} = \theta_{i,n} - \frac{\lambda}{\sqrt{\frac{m_{2,n+1}}{1-\beta_2^{n+1}} + \epsilon}} \frac{m_{1,n+1}}{1-\beta_1^{n+1}}$$

with

$$m_{1,n+1} = \beta_1 m_{1,n} + (1-\beta_1)\nabla f$$

$$m_{2,n+1} = \beta_2 m_{2,n} + (1-\beta_2)\nabla f$$

If we set $\beta_1 = 0$, we have near equivalence to RMSProp. Then, the only term that is different is a small term in the adaptive step size part, i.e. compare

$$\underbrace{\frac{\lambda}{\sqrt{\frac{m_{2,n+1,i}}{1-\beta_2^{n+1}} + \epsilon}}}_{Adam\ step\ size} \approx \underbrace{\frac{\lambda}{\sqrt{g_{i,n}^2 + \epsilon}}}_{RMSProp\ step\ size}$$

But as $\lim_{n\to\infty} \frac{1}{1-\beta_2^{n+1}} = 1$, we will get complete equivalence the more steps we take.

What do these equivalences mean for us?

Because these equivalences involve specific values of the global parameters, they generally do not occur by accident during the calculations.

But if we are building a framework that compares and evaluates different gradient descent methods, we can skip “subset models”, i.e. those methods, that are already within others.