Deep Learning

Ulf Brefeld & Soham Majumder

build: April 30, 2024

Machine Learning Group
Leuphana University of Lüneburg

Implementation and Frameworks

Writing a large neural network from scratch is tedious and error-prone

Multiple frameworks provide libraries of tensor operations and mechanisms to combine them with support for automatic differentiation

	Language(s)	**License**	**Main backer**
PyTorch	Python, C++	BSD 2.0	Facebook
TensorFlow	Python, C++	Apache	Google
JAX	Python	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon

Automatic differentiation

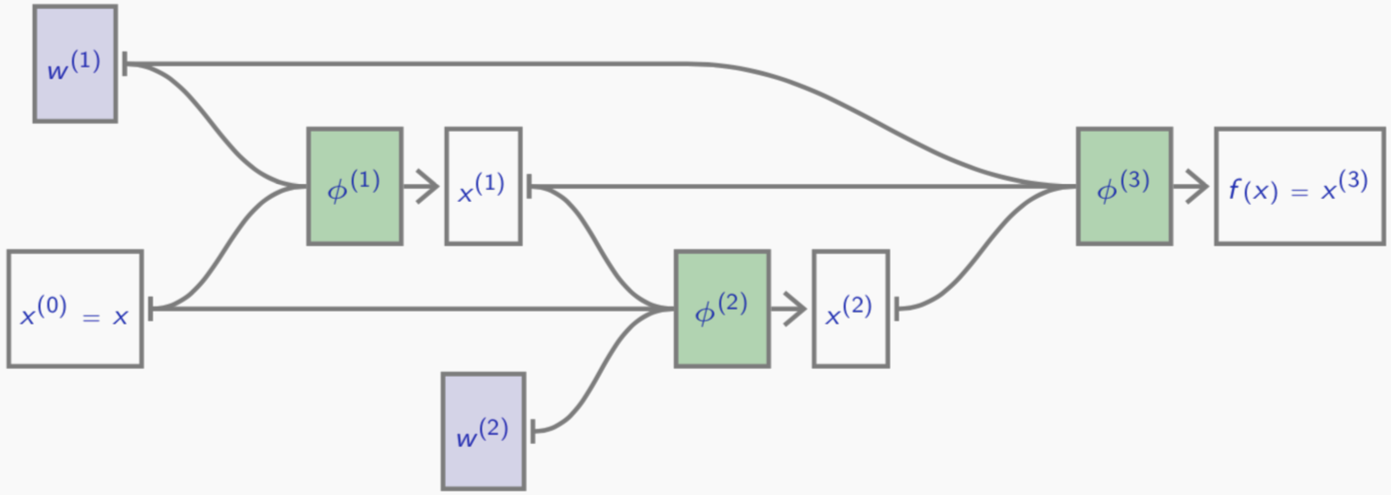
Conceptually, the forward pass is a standard tensor computation, and the directed acyclic graph of tensor operations is required to compute derivatives.

When executing tensor operations, libraries like PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.

This *autograd* mechanism has two main benefits:

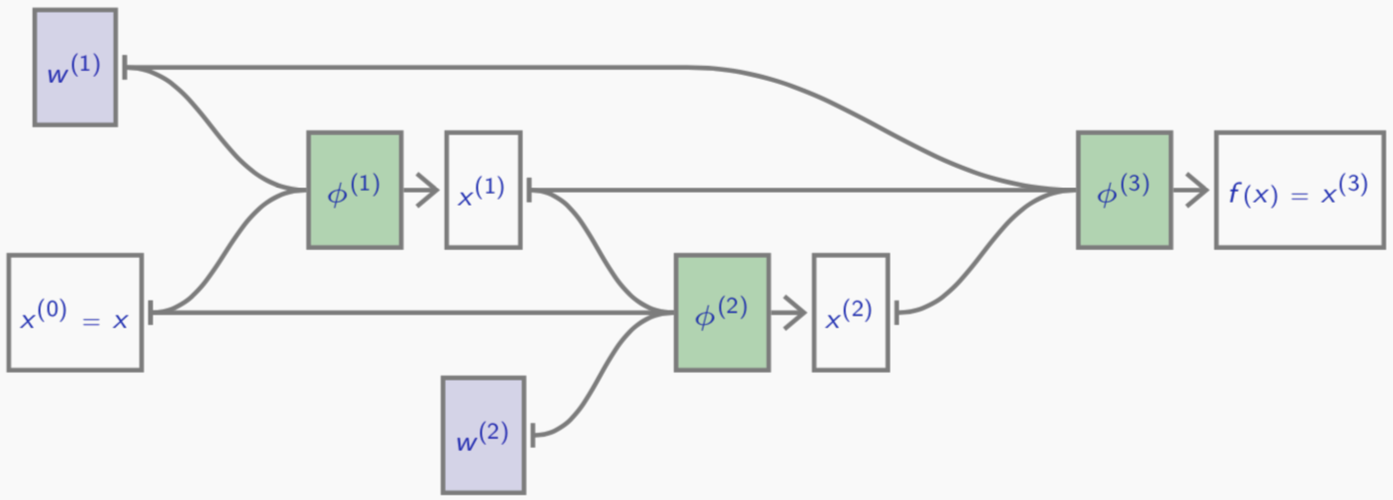
- simpler syntax that requires only the forward pass as a standard sequence of Python operations
- greater flexibility, the forward pass can be dynamically modulated since the graph is not static

Example: In Pytorch a forward/backward pass on



$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)} \cdot x^{(0)}$$
$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)} \cdot x^{(1)}$$
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)} \cdot \left(x^{(1)} + x^{(2)}\right)$$

Example: In Pytorch a forward/backward pass on



```
w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.empty(5).normal_()

x0 = x
x1 = w1 @ x0
x2 = x0 + w2 @ x1
x3 = w1 @ (x1 + x2)

q = x3.norm()

q.backward()
```

Stochastic gradient descent

So far, to minimize the error with loss function V

$$E(w) = \sum_{n=1}^{N} \underbrace{V(y_n, x_n, f_w)}_{E_n(w)}$$

we wanted to use gradient-descent which leads to the batch update

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

While it makes sense to compute the exact batch gradient,

- it takes time
- it is an empirical estimate of a hidden quantity; any partial sum of it would also be an unbiased estimator (though with a greater variance)
- is is computed incrementally

$$\nabla E(w_t) = \sum_n \nabla E_n(w_t)$$

Thus, when we compute E_n, we have already computed $E_1, E_2, \dots, E_{n-1}$ and could have a better estimate of w^* than w_t

To show that partial sums serve as good estimators, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then,

$$\begin{aligned} E(w) &= \sum_{i=1}^{N} V(y_n, x_n, f_w) \\ &= \sum_{k=1}^{K} \sum_{m=1}^{M} V(y_m, x_m, f_w) \\ &= K \sum_{m=1}^{M} V(y_m, x_m, f_w) \end{aligned}$$

Thus, instead of summing over all samples and moving in steps determined by η, we can visit only $M = N/K$ samples and move by $K\eta$ which would speed-up the computation by K

Although this is an ideal case, it is often useful to exploit the redundancy in practice

The **stochastic gradient descent** consists of updating the parameters w_t direclty after every sample

$$w_{t+1} = w_t - \eta \nabla E_{n(t)}(w_t)$$

However, this is the other extreme and

- does not benefit from the speed-ups from computations in parallel for multiple samples
- the loss function is being optimized with much less information per step

The *mini-batch* stochastic gradient descent is the middle ground.

It visits the samples in mini-batches of size B (a few tens or hundreds of samples), and updating the parameters each time.

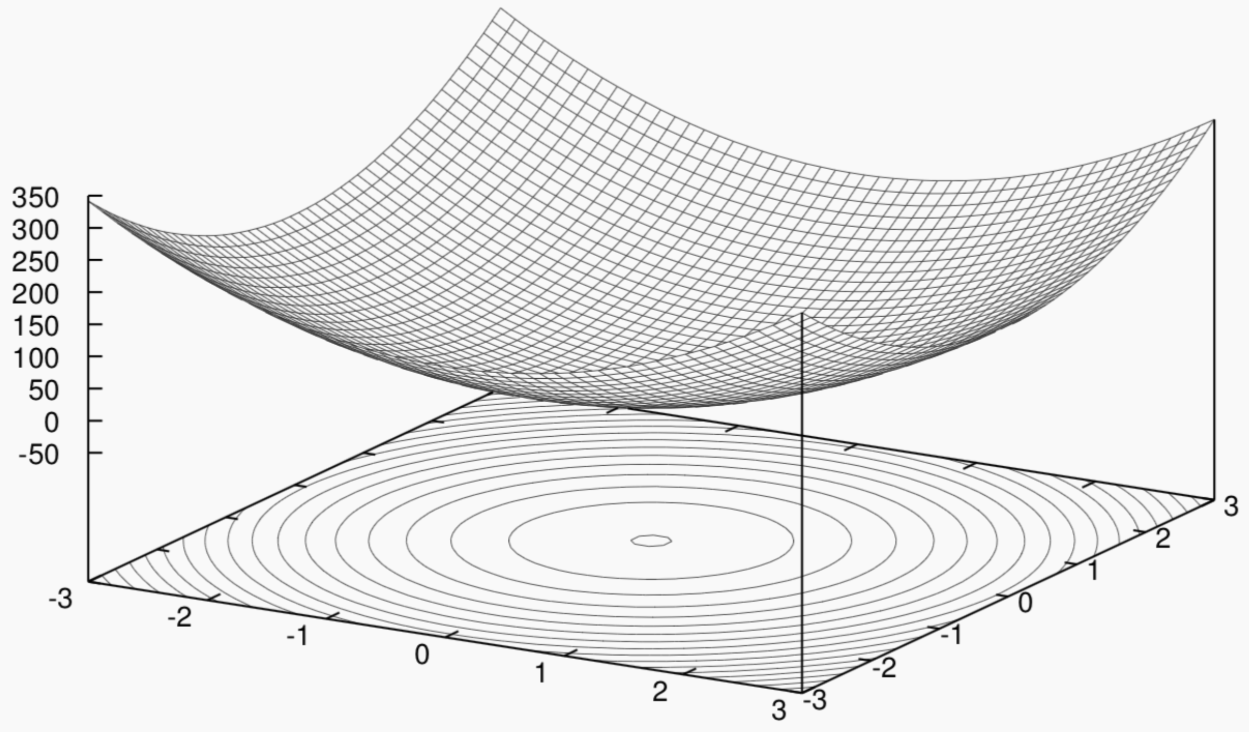
$$w_{t+1} = w_t - \eta \sum_{b=1}^{B} \nabla E_{n(t,b)}(w_t)$$

As in stochastic gradient descent, the order $n(t, b)$ of the samples does not matter much and can be sequential or randomly drawn without replacement

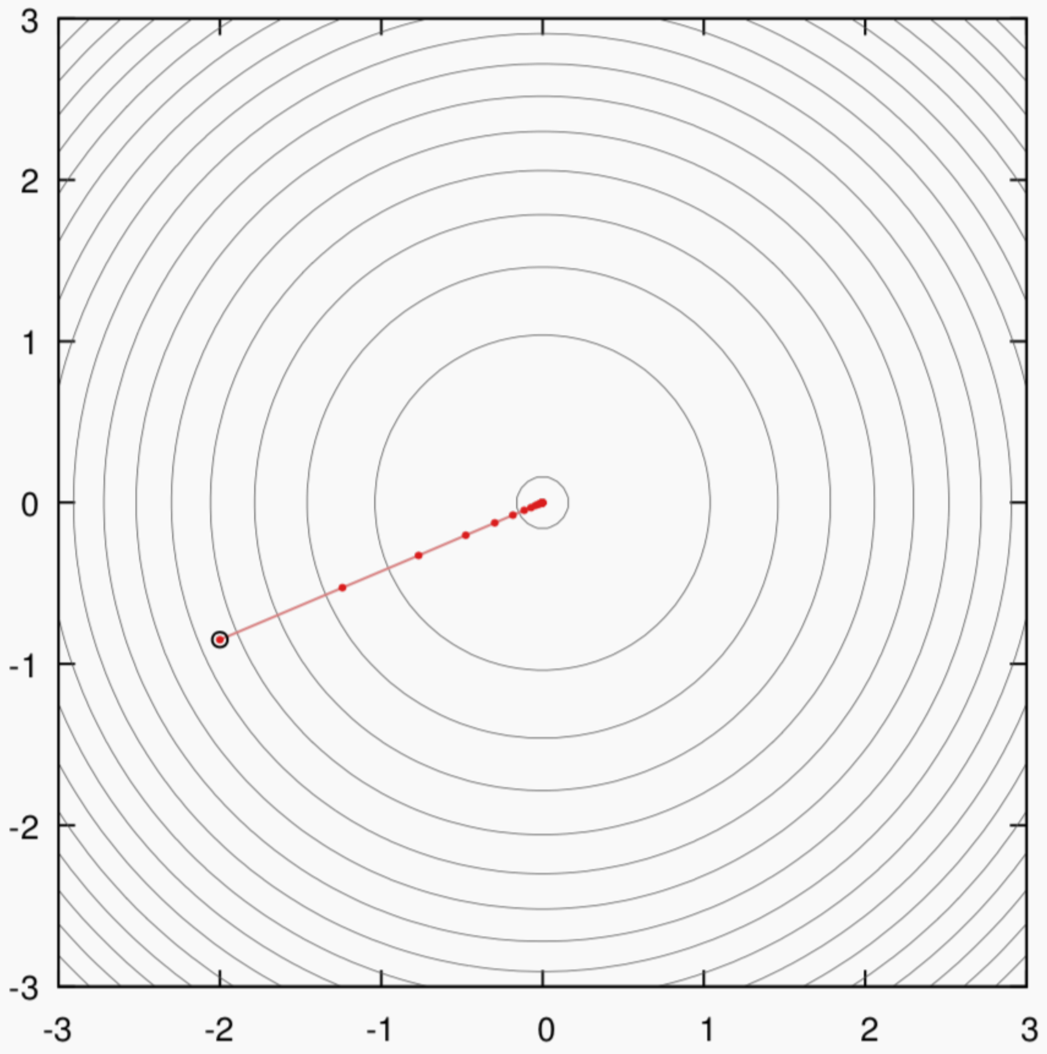
In practice, the stochastic nature of this procedure helps to avoid local minima and tends to work as a regularizer (we will come back to this)

Limitations of gradient descent

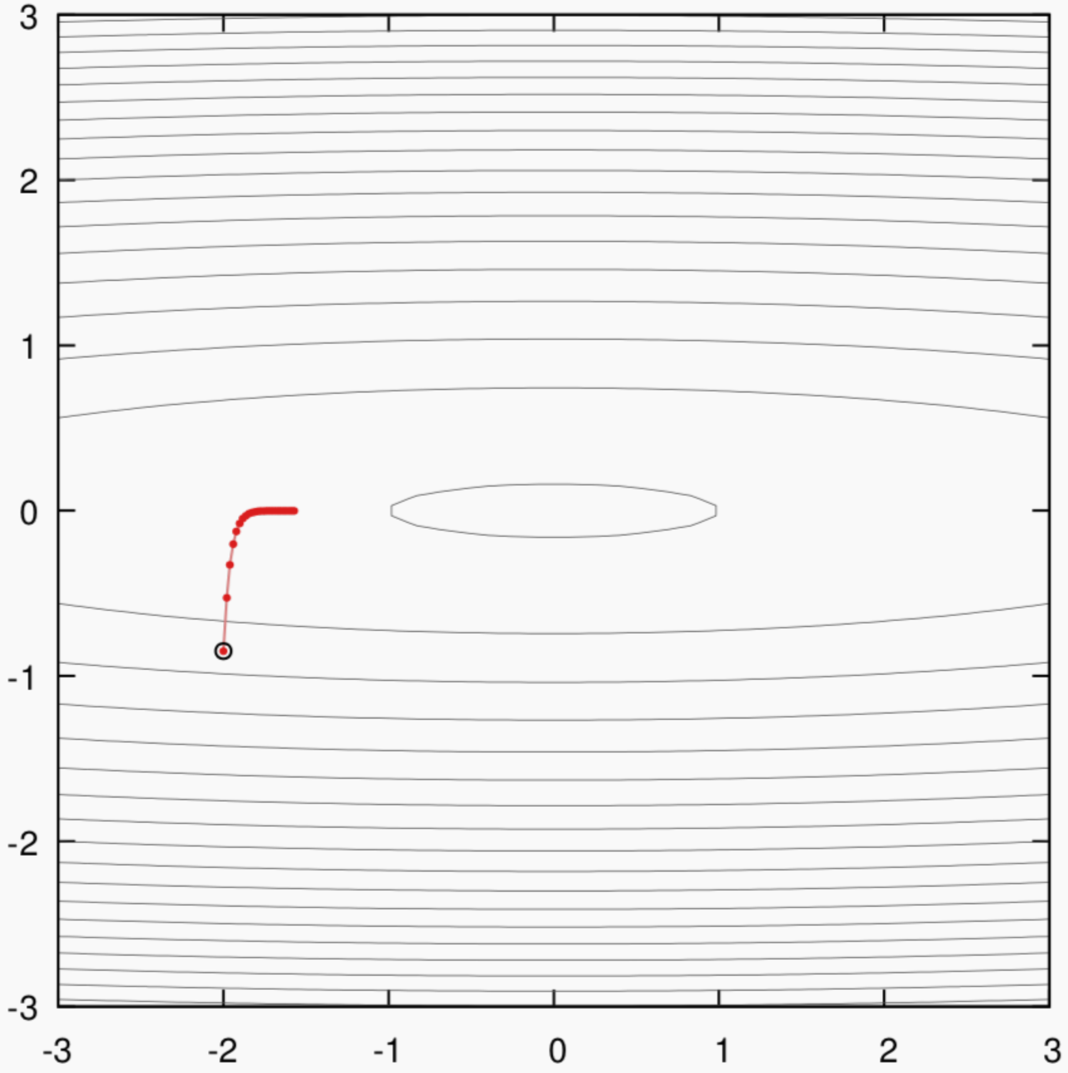
Gradient descent puts a strong assumption on the magnitude of the *local curvature* to fix the step size, and about its isotropy so that the same step size makes sense in all directions.



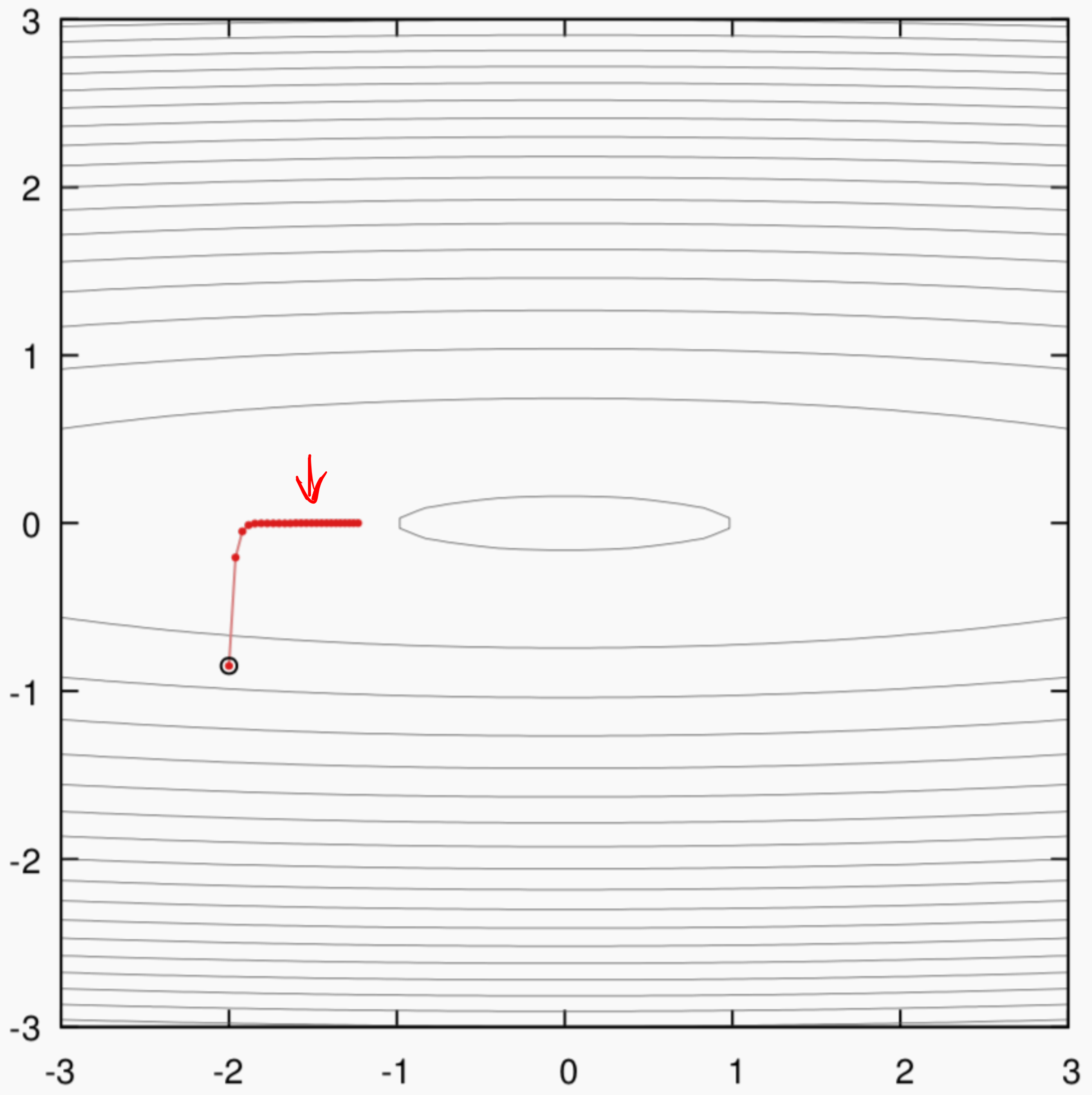
$\eta = 1.0e-2$



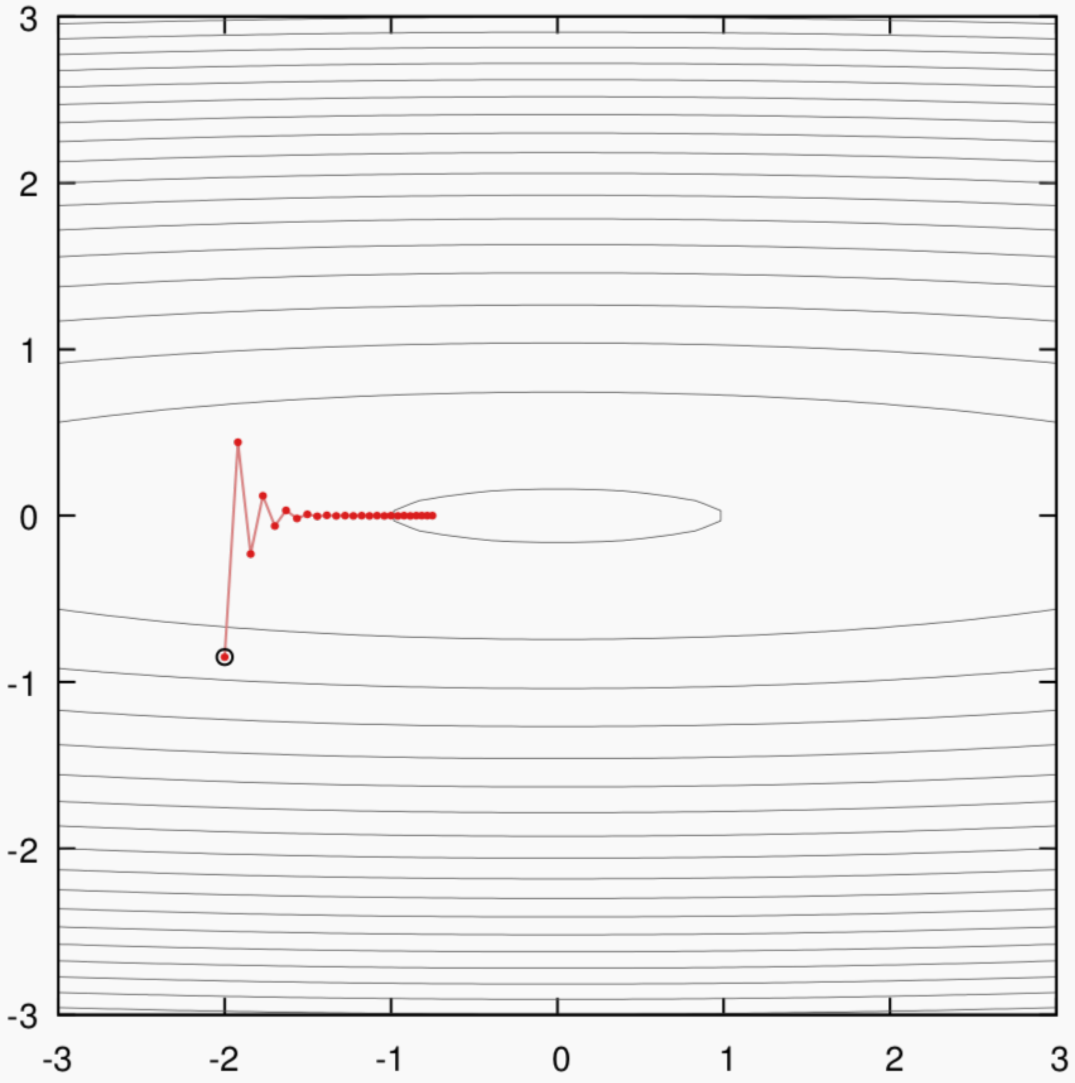
$\eta = 1.0e-2$



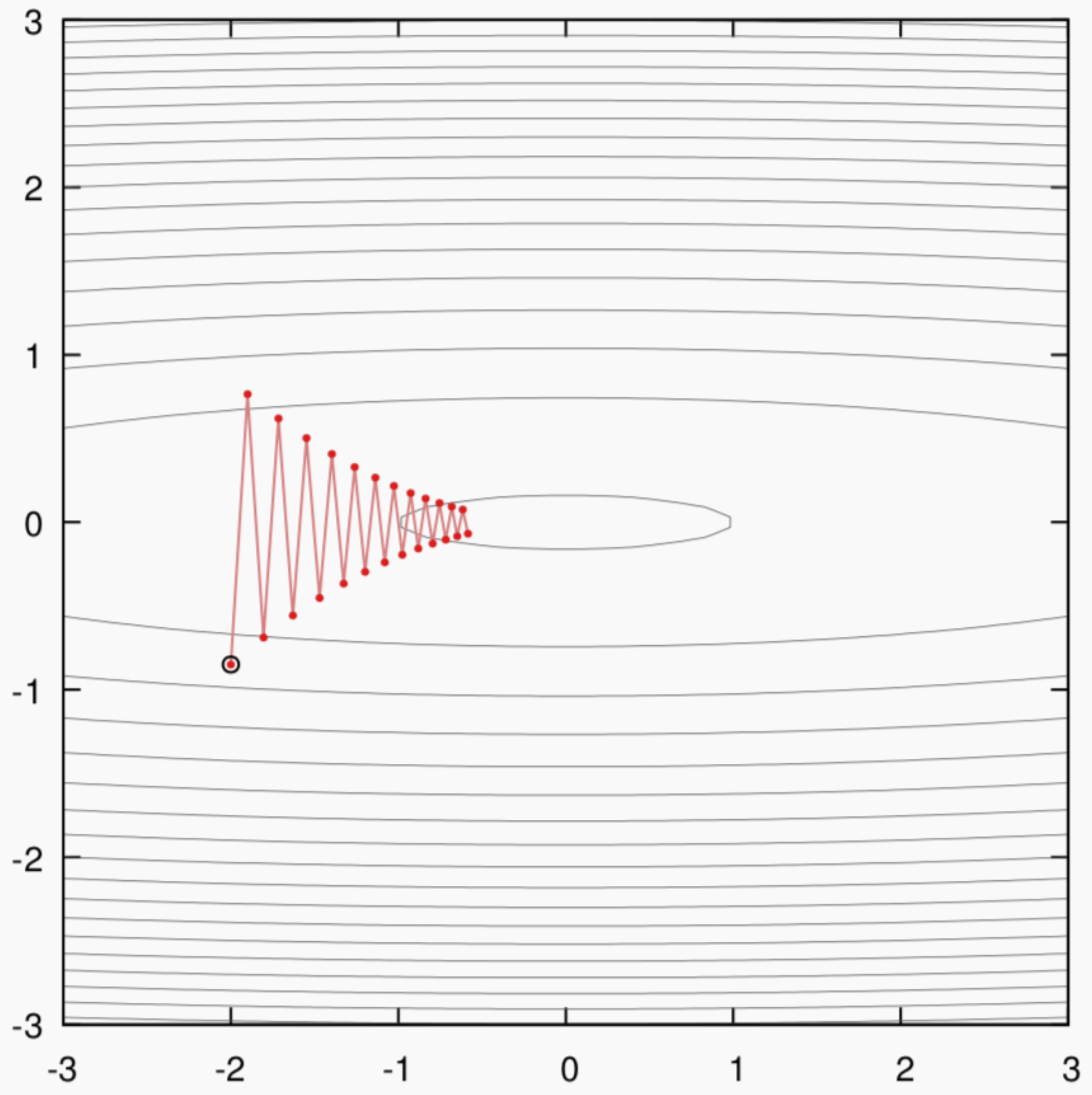
$\eta = 2.0e-2$



$\eta = 4.0e-2$



$\eta = 5.0e-2$

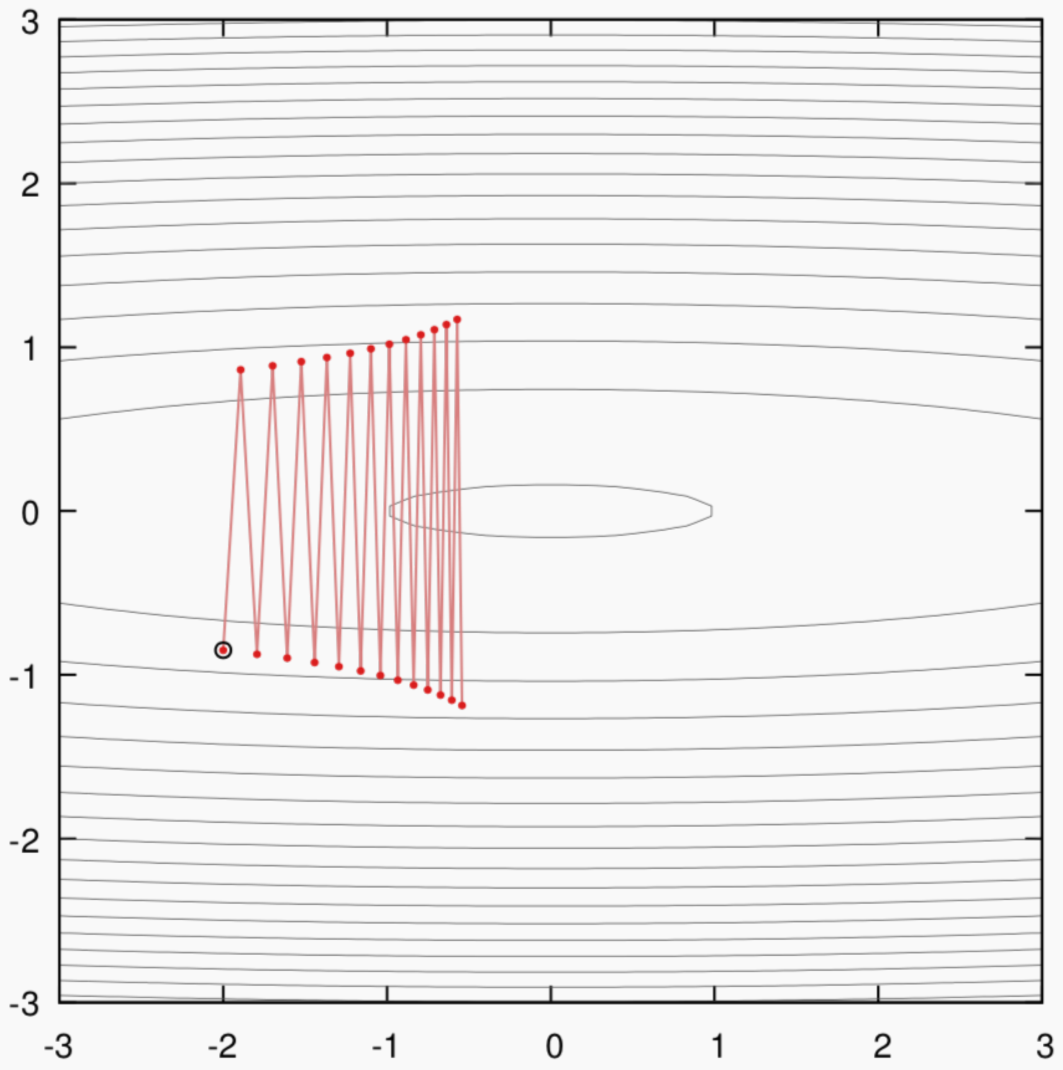


$x \in \{[-1000, 1000], [0, 1]\}$

$\eta = 5.3e-2$

$wx =$

$w_1 x_1 + w_2 x_2$



Some optimization methods leverage second order moments, to use a more accurate local model of the functional to optimize

However, second order information comes at a computational cost and, for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the optimization is often worse in practice

Deep learning generally relies on a smarter use of the gradient, using statistics over its past values to make a smarter step with the current one

The *vanilla* mini-batch stochastic gradient descent (SGD) consists of

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

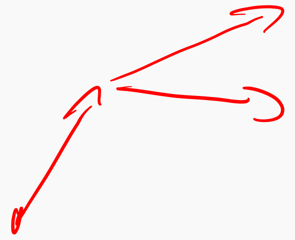
where

$$g_t = \sum_{b=1}^{B} \nabla E_n(t,b)(w_t)$$

is the gradient summed over a mini-batch

A first improvement is the use of a *momentum* to add inertia in the choice of the step direction

$$u_t = \underline{\gamma u_{t-1}} + \eta g_t$$
$$w_{t+1} = w_t - u_t$$



For $\gamma = 0$, this is identical to vanilla SGD

For $\gamma > 0$, this update has three nice properties:

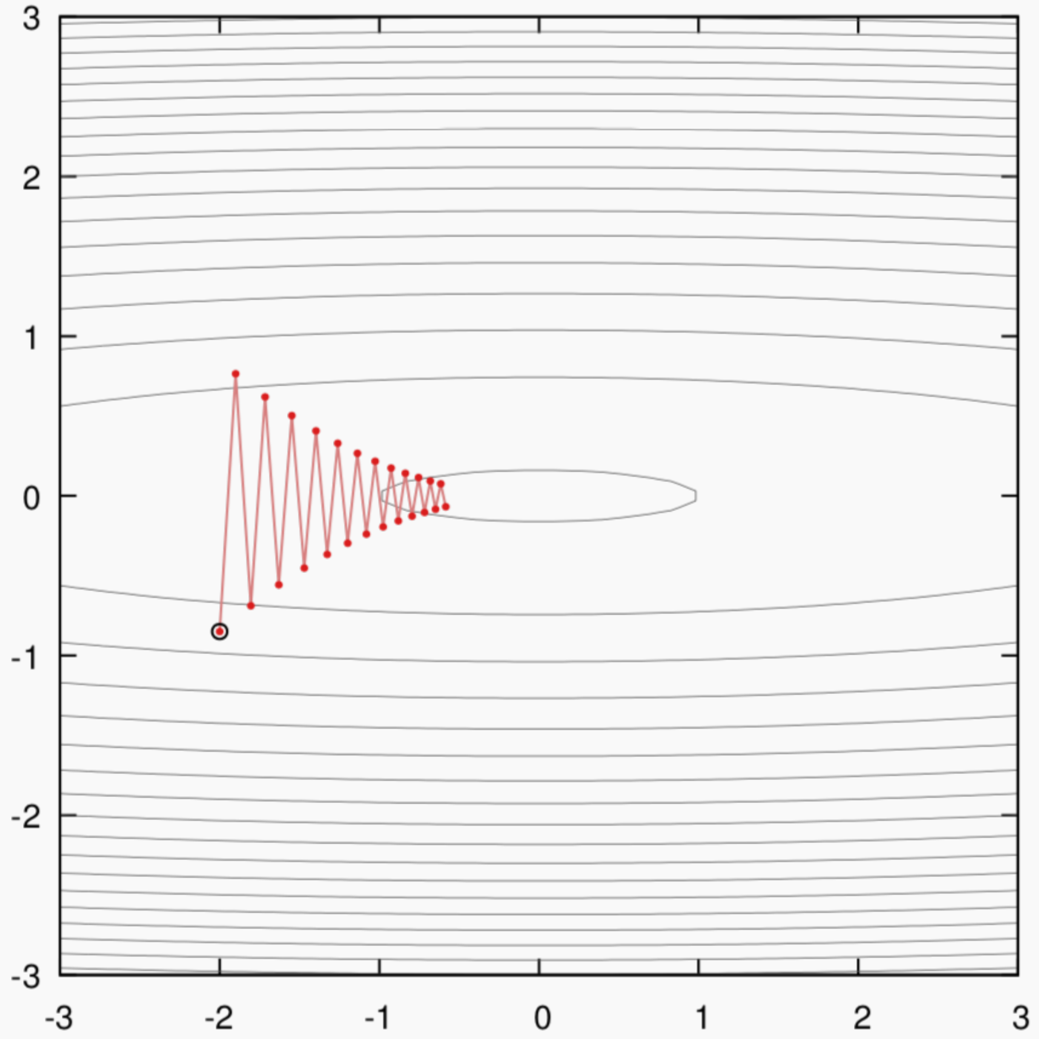
- it can circumvent optimization *barriers*
- it accelerates if the gradient does not change much

$$(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1-\gamma} g\right)$$

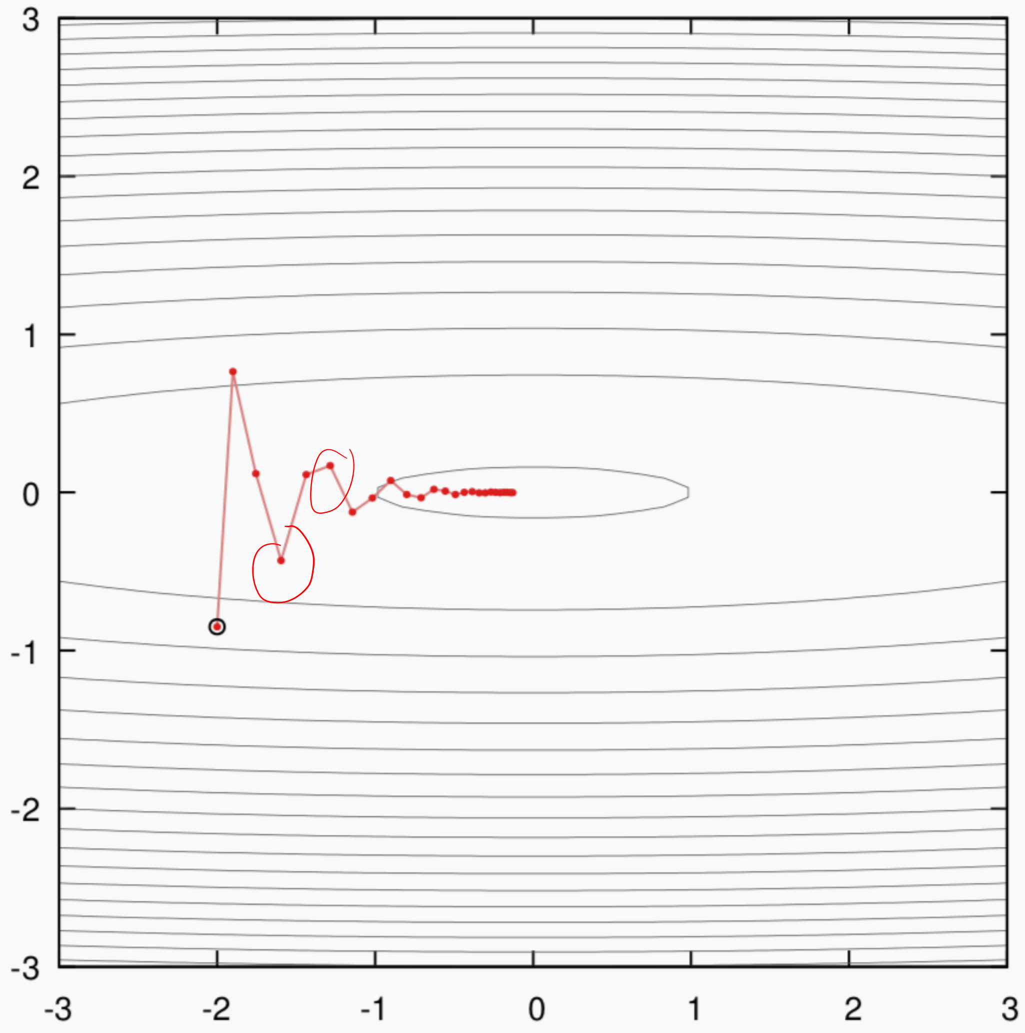
- it dampens oscillations in narrow valleys

$\eta = 5.0e-2, \gamma = 0$

SGD



$\eta = 5.0e - 2, \gamma = 0.5$



Momentum

Another class of methods exploits the statistics over the previous steps to compensate for the direction of the gradient

The Adam algorithm uses moving averages of each dimension and its square to rescale dimensions separately (Kingma & Ba, 2015)

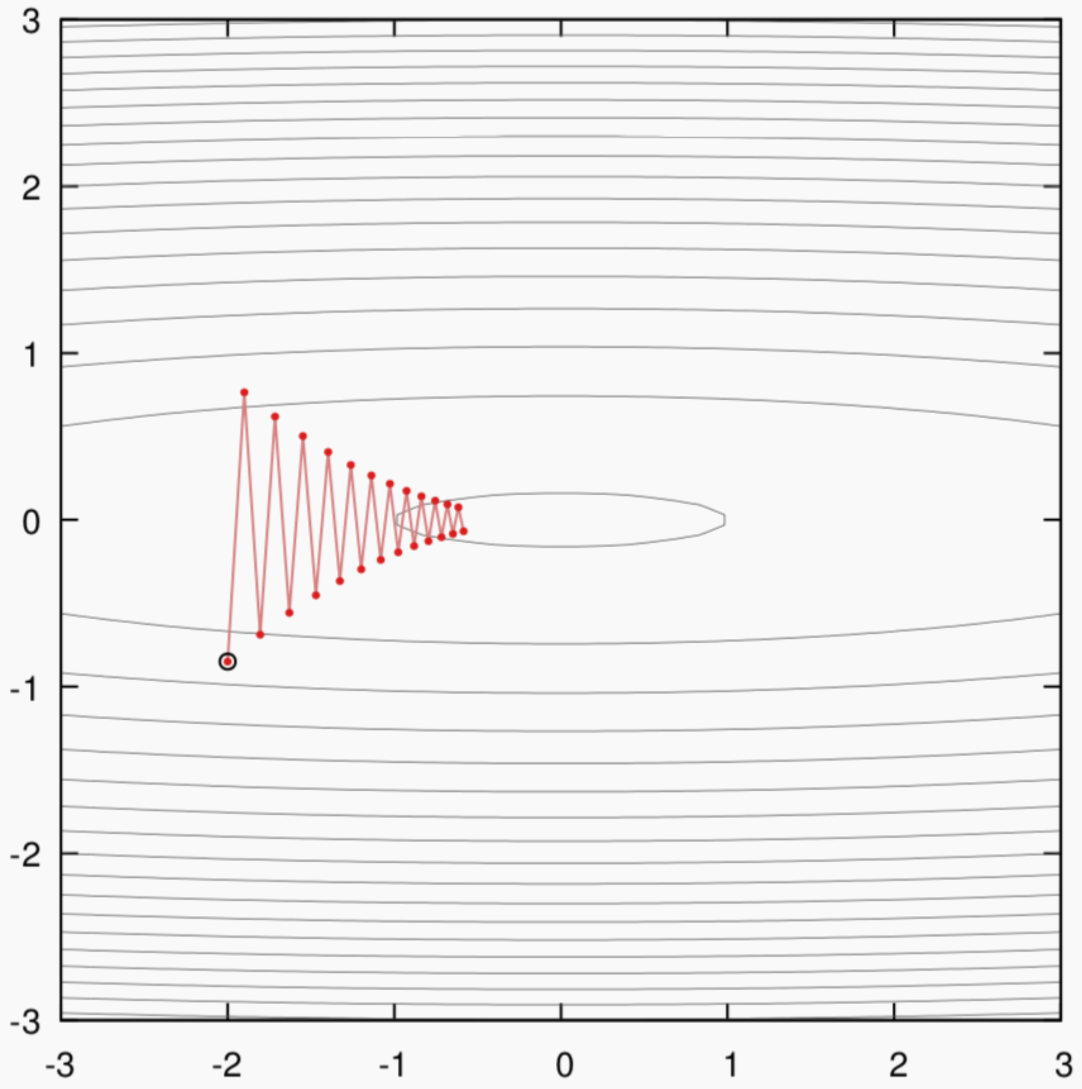
The update rule performs **individual corrections for each dimension**

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t
\end{aligned}
$$

$\beta_1 \in [0, 1]$

Intuitively, Adam combines momentum $\hat{m}_t$ and a per-dimension re-scaling by $\hat{v}_t$

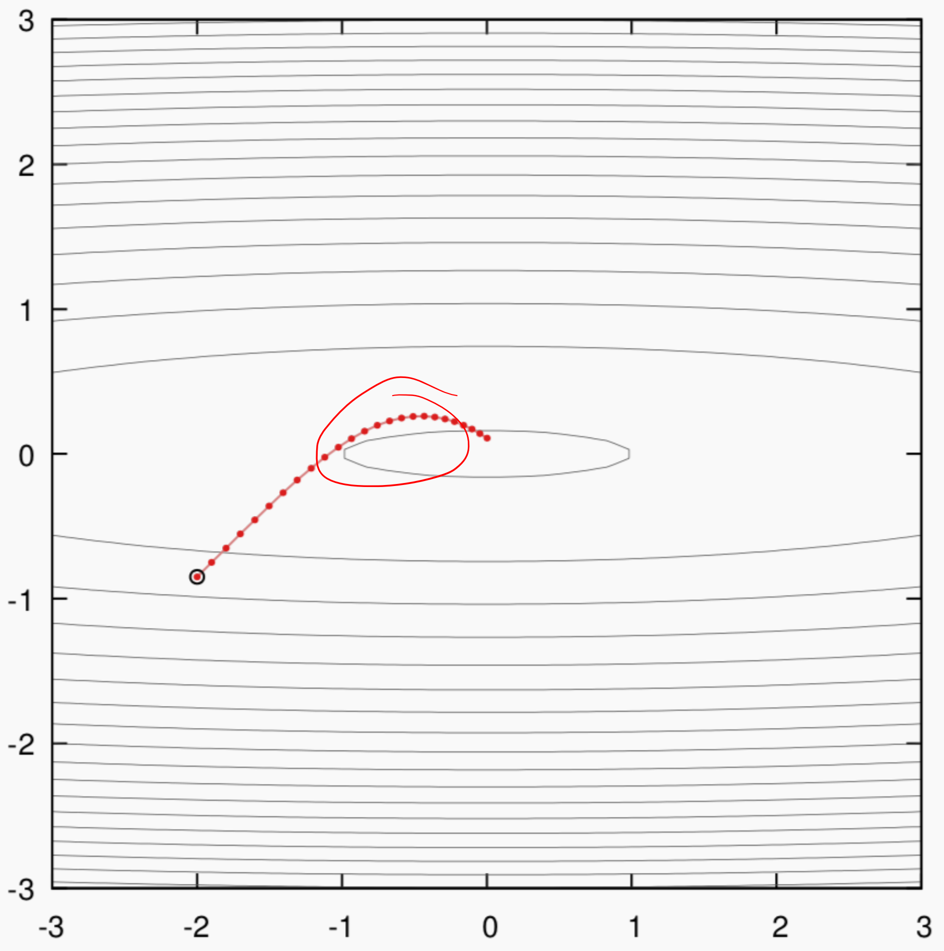
$\eta = 5.0e-2$



SGD

Adam, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8, \eta = 1.0e-1$

!



Adam

There are many alternatives proposing to improve vanilla SGD, e.g.,

- Nesterov's accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- AdaMax
- Nadam

References

Kingma, D. P., and Ba, J. (2015). Adam: A Method for Stochastic Optimization. In International Conference on Learning Representations.