EE-559 – Deep learning

5.2. Stochastic gradient descent

François Fleuret

https://fleuret.org/ee559/

Wed Nov 21 08:24:02 UTC 2018

RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

So far, to minimize a loss of the form

$$\mathscr{L}(w) = \sum_{n=1}^{N} \underbrace{\ell(f(x_n; w), y_n)}_{\ell_n(w)}$$

we have considered the gradient-descent algorithm, of the form

$$w_{t+1} = w_t - \eta \nabla \mathscr{L}(w_t).$$

A straight-forward implementation would be

```
for e in range(nb_epochs):
    output = model(train_input)
    loss = criterion(output, train_target)

    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

A straight-forward implementation would be

```
for e in range(nb_epochs):
    output = model(train_input)
    loss = criterion(output, train_target)

    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

However, the memory footprint is proportional to the full set size. This can be mitigated by summing the gradient through “mini-batches”:

```
for e in range(nb_epochs):
    model.zero_grad()

    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        loss.backward()

    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

While it makes sense in principle to compute the gradient exactly, in practice:

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.
- It is computed incrementally

$$\nabla \mathscr{L}(w_t) = \sum_{n=1}^{N} \nabla \ell_n(w_t),$$

and when we compute ℓ_n, we have already computed $\ell_1, \dots, \ell_{n-1}$, and we could have a better estimate of w^* than w_t.

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times.

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then

$$
\begin{aligned}
\mathscr{L}(w) &= \sum_{n=1}^{N} \ell(f(x_n; w), y_n) \\
&= \sum_{k=1}^{K} \sum_{m=1}^{M} \ell(f(x_m; w), y_m) \\
&= K \sum_{m=1}^{M} \ell(f(x_m; w), y_m).
\end{aligned}
$$

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then

$$
\begin{aligned}
\mathscr{L}(w) &= \sum_{n=1}^{N} \ell(f(x_n; w), y_n) \\
&= \sum_{k=1}^{K} \sum_{m=1}^{M} \ell(f(x_m; w), y_m) \\
&= K \sum_{m=1}^{M} \ell(f(x_m; w), y_m).
\end{aligned}
$$

So instead of summing over all the samples and moving by η, we can visit only $M = N/K$ samples and move by $K\eta$, which would cut the computation by K.

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

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

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

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

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

However this does not benefit from the speed-up of batch-processing.

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

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

However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

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

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

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

However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

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

The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

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

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

However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in "mini-batches", each of a few tens of samples, and updating the parameters each time.

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

The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

The stochastic behavior of this procedure helps evade local minima.

So our exact gradient descent with mini-batches

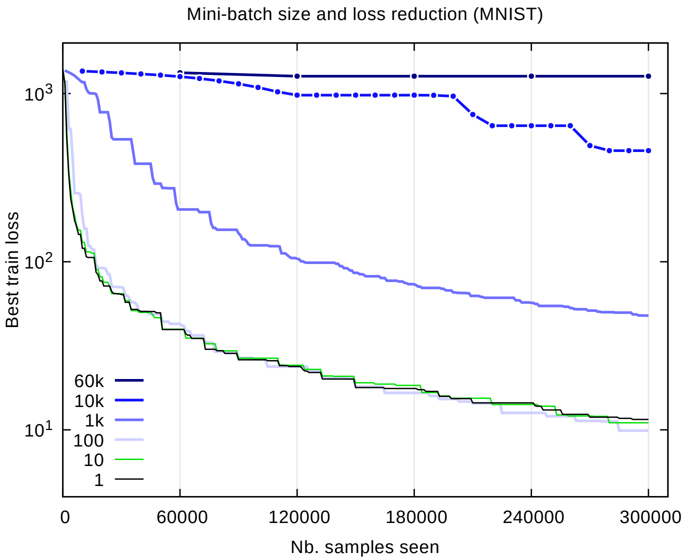
```
for e in range(nb_epochs):
    model.zero_grad()

    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        loss.backward()

    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

So our exact gradient descent with mini-batches

```
for e in range(nb_epochs):
    model.zero_grad()

    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        loss.backward()

    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

can be modified into the mini-batch stochastic gradient descent as follows:

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])

        model.zero_grad()
        loss.backward()
        with torch.no_grad():
            for p in model.parameters(): p -= eta * p.grad
```

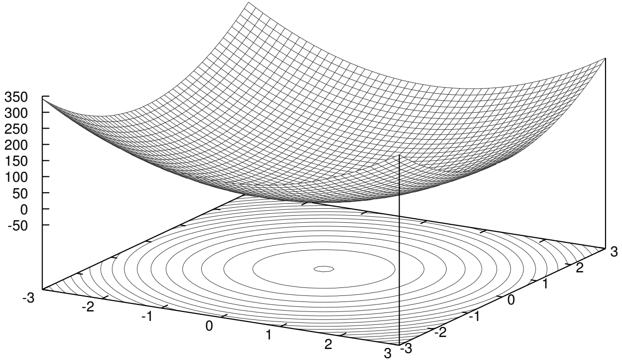


Mini-batch size and loss reduction (MNIST)
Best train loss
10^3
10^2
10^1
60k
10k
1k
100
10
1
0
60000
120000
180000
240000
300000
Nb. samples seen

Limitation of the gradient descent

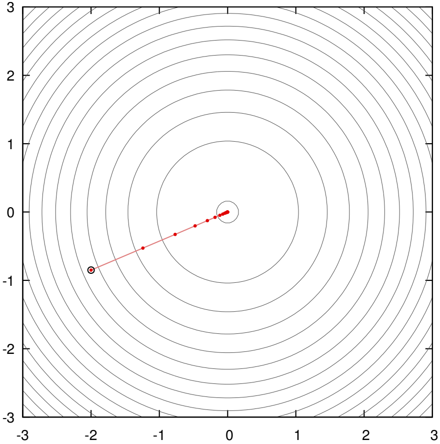
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

The gradient descent method makes a strong assumption about 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.



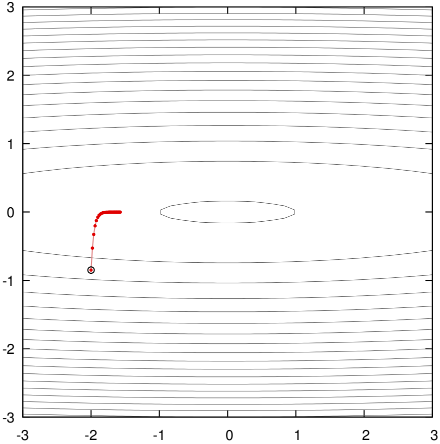
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$\eta = 1.0e - 2$



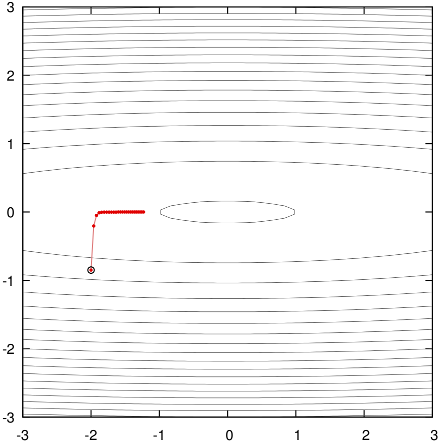
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$\eta = 1.0e - 2$



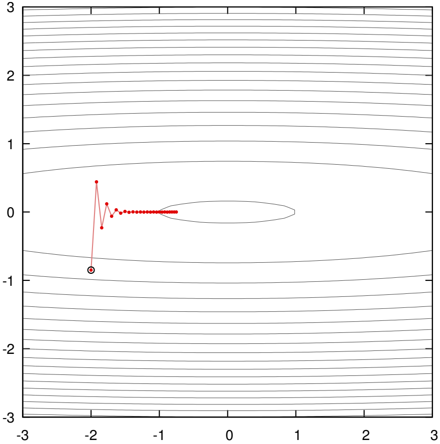
The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$\eta = 2.0e - 2$



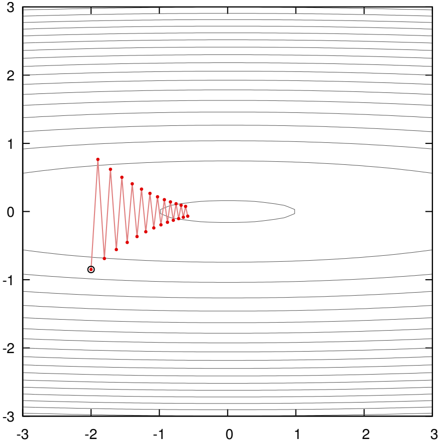
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$\eta = 4.0e - 2$



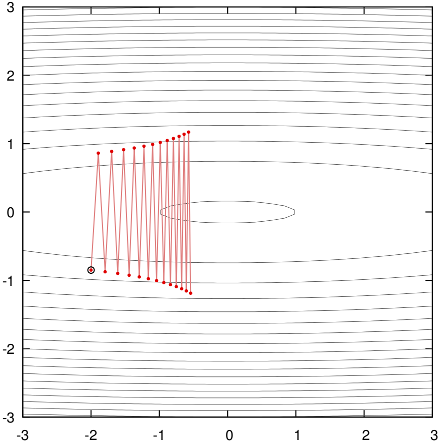
The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.

$\eta = 5.0e - 2$



The gradient descent method makes a strong assumption about 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.

$\eta = 5.3e - 2$



Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worst.

Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worst.

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.

Momentum and moment estimation

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

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

where

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

is the gradient summed over a mini-batch.

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

$$
\begin{aligned}
u_t &= \gamma u_{t-1} + \eta g_t \\
w_{t+1} &= w_t - u_t .
\end{aligned}
$$

(Rumelhart et al., 1986)

With $\gamma = 0$, this is the same as vanilla SGD.

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

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

(Rumelhart et al., 1986)

With $\gamma = 0$, this is the same as vanilla SGD.

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

- it can "go through" local barriers,

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

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

(Rumelhart et al., 1986)

With $\gamma = 0$, this is the same as vanilla SGD.

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

- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

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

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

$$
\begin{aligned}
u_t &= \gamma u_{t-1} + \eta g_t \\
w_{t+1} &= w_t - u_t .
\end{aligned}
$$

(Rumelhart et al., 1986)

With $\gamma = 0$, this is the same as vanilla SGD.

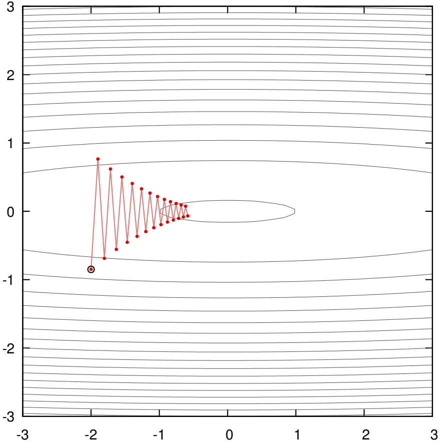
With $\gamma > 0$, this update has three nice properties:

- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

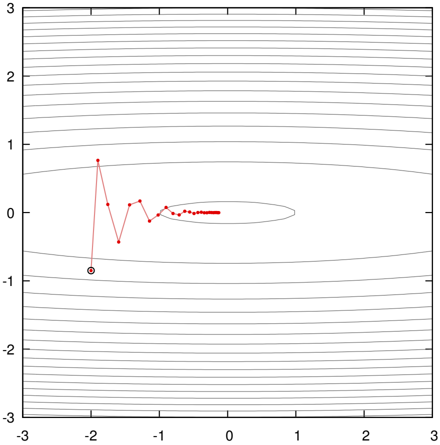
$$
(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1-\gamma} g \right),
$$

- it dampens oscillations in narrow valleys.

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



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



Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the mapping.

The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the mapping.

The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

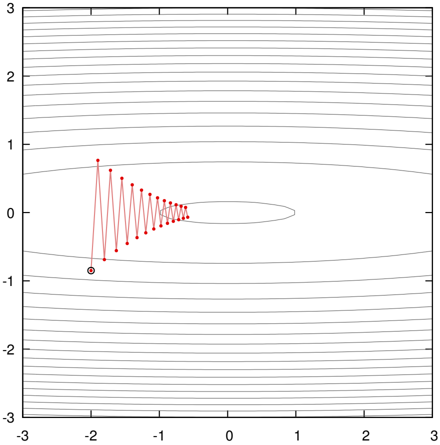
The update rule is, **on each coordinate separately**

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

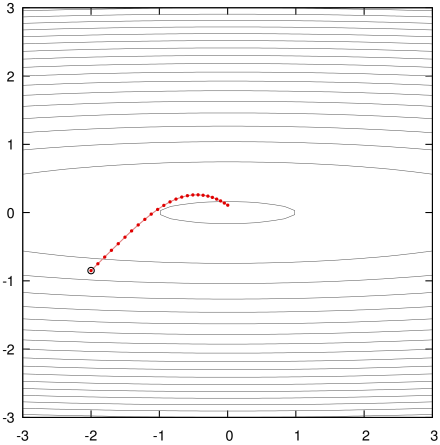
(Kingma and Ba, 2014)

This can be seen as a combination of momentum, with $\hat{m}_t$, and a per-coordinate re-scaling with $\hat{v}_t$.

$\eta = 5.0e - 2$



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



These two core strategies have been used in multiple incarnations:

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

The end

References

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(9):533–536, 1986.