Deep learning

5.2. Stochastic gradient descent

François Fleuret

https://fleuret.org/dlc/

To minimize a loss of the form

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

the standard gradient-descent algorithm update has the form

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

Notes

Here

- N is the total number of samples,
- x_n is a sample with label y_n,
- ℓ is the function to evaluate how bad the predictor f is on the samples.

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
```


Notes

An "epoch" corresponds to visiting the full training set once. The criterion is the function used to evaluate the prediction (MSE loss, cross-entropy loss, etc.) `p.grad` is the gradient of the loss w.r.t. the parameter.
Using "mini-batches" prevents from evaluating the model on the full data set in one shot, which would be intractable with a large data set.
Both pieces of code produce the exact same result, because `loss.backward()` accumulates the gradient in the `grad` field of the variables, by linearity of the gradient operator, so that the `grad` fields contain the same gradient values after the loop over all the samples.
Processing the training data set by "mini-batches" solves the first issue which is the memory footprint. The mini-batches size can be arbitrarily small.

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 a 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 $\nabla \ell_n$, we have already computed $\nabla \ell_1, \ldots, \nabla \ell_{n-1}$, 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. 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.

Notes

To make it more concrete, we can imagine that the training set of size $N = 1{,}000{,}000$ samples is actually $M = 2{,}000$ samples replicated $K = 500$ times.
Computing the loss on this training set give the same result as computing the loss on the $2{,}000$ samples, and then multiply this partial loss by 500 to get the loss on the full data set, or equivalently one can use $K\eta$ as step size.

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.

Notes

The order $n(t, b)$ to visit the samples can either be:

- sequential, in the natural order of the data set. This can be problematic, if the data set is made such that all the elements of the same class appear all together;
- uniform sampling (usually without replacement), which consists in shuffling the samples before visiting them. If the sampling is done with replacement, the same sample may be used several times in the same epoch, in which case an epoch means using as many samples as the size of the training set.

This "mini-batch gradient descent" is an efficient procedure because:

- it benefits from the speed-up of batch-processing,
- the model is updated more frequently by taking into account the redundancy in the data,
- local minima can be evaded due to the randomness of the process.

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
```

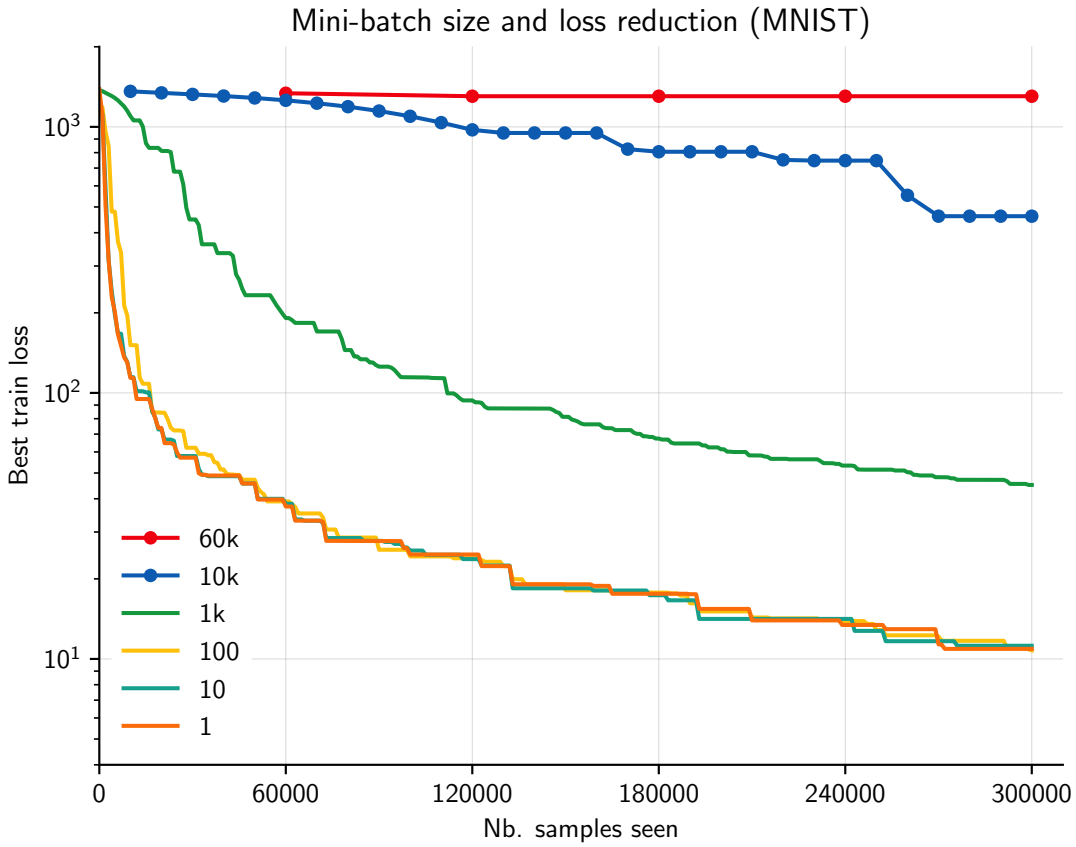
can be modified into the mini-batch stochastic gradient descent as follows:

```
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
```


Notes

In the "mini-batch gradient descent", the gradient is computed on a mini-batch and the model is then updated.



Notes

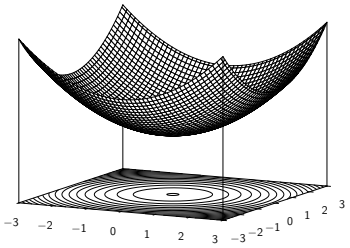
This graph shows the training loss of a simple LeNet model trained on MNIST as a function of the number of training samples seen. The training set contains 60,000 samples.
The loss is computed on the full training set, and is the "best so far" to make the curves easier to read: if after an update, the training loss increases, we keep the previous value to plot. This is why the curves are decreasing with plateaus.
Each curve represents a training done with one particular mini-batch size, that is the number of samples used to make one update. Note that a batch of size 60k corresponds to classical "non-stochastic" gradient descent, where the full training set is used for an update. A batch size of 1 is pure stochastic gradient descent where we update the model after each sample.
These curves show that:

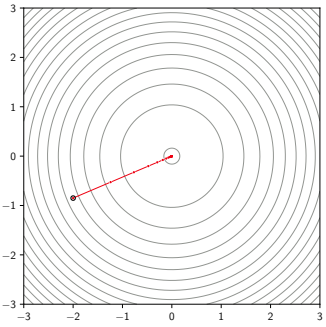
- it works far better to update the parameters more often, that is using only a small number of samples for each update,
- there is no use at using very small batches or to look at samples individually: a mini-batch of 100 samples is as efficient as stochastic gradient descent, but benefits from the speed-up of batch processing.

Note that learning rates were optimized for each mini-batch size.

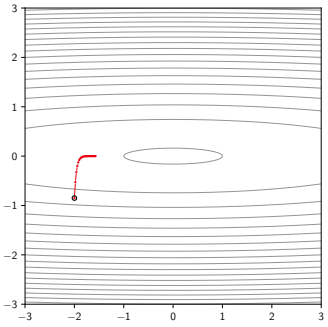
Limitation of the gradient descent

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.

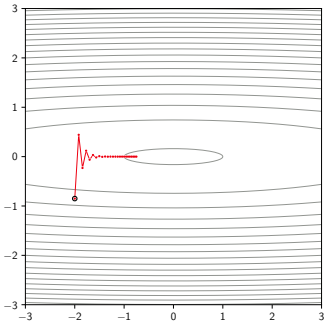




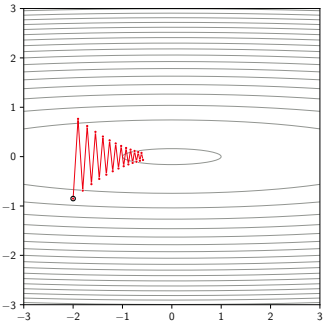
$\eta = 1.0e-2$



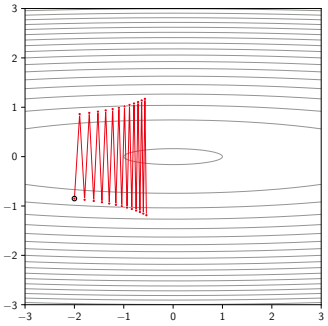
$\eta = 1.0e-2$



$\eta = 4.0e-2$



$\eta = 5.0e-2$



$\eta = 5.3e-2$

Notes

This example illustrates the drawback of the gradient descent when the loss is not isotropic. The black lines are the level lines of the $\mathbb{R}^2 \to \mathbb{R}$ mapping to minimize, and the red polygonal line shows the trajectory of gradient descent starting at the black circle.

- When the curvature is the same in both x and y directions (top middle), the gradient descent goes straight to the minimum.
- When the curvature is not isotropic (top right and all bottom), the descent goes faster where the slope is steeper, here in the y direction.
- Increasing the learning rate to reach the minimum in x is better (bottom left).
- But having a step size which suits the x direction creates oscillations (bottom center).
- And eventually diverges in the y direction.

This example shows that gradient descent should have a different learning rate for each direction.

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 worse.

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.

Notes

Here

- w_t is the current estimate of the parameter,
- w_{t+1} is the update of the estimate,
- g_t is the gradient computed over a mini-batch of samples with parameter w_t,
- $n(t, b)$ is the index of the bth sample of the mini-batch used at iteration t.

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)

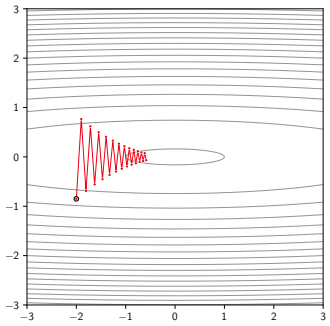
With $\gamma = 0$, this is the same as vanilla SGD.

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

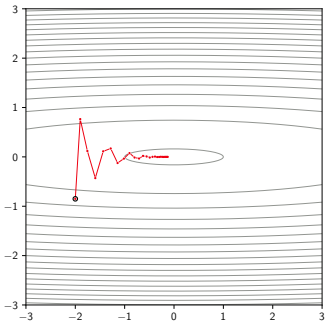
- it can "go through" local barriers,
- it accelerates if the gradient does not change much:

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

- it dampens oscillations in narrow valleys.



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



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

Notes

We take our example of a non-isotropic quadratic function to illustrate the difference between gradient descent with and without momentum.
The left image shows the successive locations of a standard vanilla gradient descent: there is no momentum ($\gamma = 0$). We have oscillations in the y direction, and it converges very slowly in the x direction.
With a momentum of $\gamma = 0.5$,

- the oscillations in the y direction are dampened due to the constant alternative sign which makes the average gradient close to 0 in that direction, and
- the moves in the x direction are larger than before, because the gradient is rather constant in this direction which causes momentum to accelerate.

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

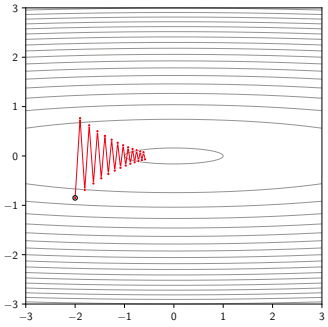
The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

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

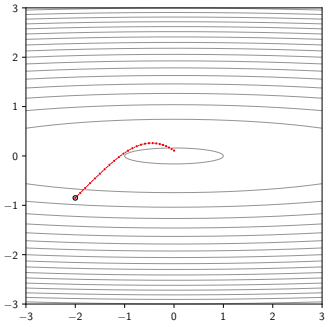
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

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

Notes

These parameter values for Adam here are the standard ones.

These two core strategies have been used in multiple incarnations:

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

There is unfortunately no best general optimizer. Although a default choice such as Adam with default parameter values usually gives good results, it can be beneficial to test alternatives and optimize meta-parameters.

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.