ECE 1508S2: Applied Deep Learning

Chapter 3: Advancing Our Toolbox

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2025

What's Next

Right now, we are familiar with FNN and how to train them

↳ *We potentially can also handle some other architectures*

But, if we try to implement them from scratch, we could get into trouble

We still need to learn more tricks for having a working implementation

---

In this chapter, we advance our bag of tools in four respects

1. *We learn more about optimizers*
   ↳ *more advanced tricks to make gradient descent work*
2. *We learn about hyperparameter tuning*
3. *We learn about data preprocessing*
   ↳ *how to handle data in practice*
4. *Tricks to make training faster and more robust*

Back to Gradient Descent

Let's take a look at *training* in *abstract form* once again

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w}) \qquad \text{(Training)}$$

Recall that $\mathbf{w}$ *includes all weights and biases*; for instance,

> *In FNN of Assignmet 2,* $\mathbf{w}$ *has 3,457 entries! In practice much higher!*

Also recall the gradient descent

1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
2: Choose some small ϵ and η, and set $t = 1$
3: **while** $\Delta > \epsilon$ **do**
4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
6: **end while**

Keep in mind: we almost always use (mini-batch) SGD $\rightsquigarrow$ *let's call it SGD*

Convergence Rate of Optimizers

Recall: we can make gradient descent converging to local minimum if

we set the learning rate η small enough

So is it also with SGD. *But, how fast does the algorithm converge?*

Speed of an optimization algorithm is evaluated by convergence rate

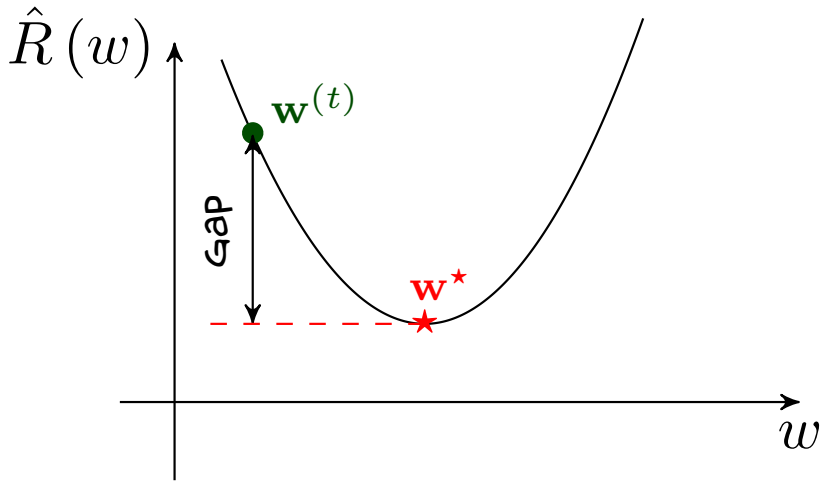
We can understand its meaning from the eyes of *optimality gap*

Optimality Gap in Iteration t

Optimality gap is the gap between $\mathbf{w}^{(t)}$ and local minimizer $\mathbf{w}^\star$, i.e.,

$$\text{optimality gap in iteration } t = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}\,(\mathbf{w}^\star)|$$

Convergence Rate of Optimizers



An optimizer *shrinks this gap gradually*, i.e., *we can approximately say*[1]

$$|\hat{R}(\mathbf{w}^{(t+1)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \ldots \leqslant |\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})|$$

In practice, this drop can occur *with different speeds in terms of* t

[1] It's just approximately correct: *gap may increase in one iteration only! That's no problem*

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

This intuitively means that *for each iteration t, we see*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})|$$

for some $\alpha < 1$: in this case, we can say

$$\begin{aligned}
|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| &\leqslant \alpha |\hat{R}(\mathbf{w}^{(t-1)}) - \hat{R}(\mathbf{w}^{\star})| \\
&\leqslant \alpha(\alpha |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^{\star})|) = \alpha^2 |\hat{R}(\mathbf{w}^{(t-2)}) - \hat{R}(\mathbf{w}^{\star})| \\
&\leqslant \ldots \leqslant \alpha^t |\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})|
\end{aligned}$$

Now say that $|\hat{R}(\mathbf{w}^{(0)}) - \hat{R}(\mathbf{w}^{\star})| = C$.[2] So, we can say

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant C\alpha^t \rightsquigarrow \mathcal{O}(\alpha^t)$$

[2] We don't really care how large C is. It's going to shrink at the end

Convergence Rate of Optimizers

An ideal scenario is that the gap drops by a constant factor each iteration

If we wish to *end up somewhere in ϵ-neighborhood of $\mathbf{w}^\star$*; then, we need

$$C\alpha^t \leqslant \epsilon$$

For this to happen, *we need to have at least*

$$t \geqslant \frac{\log 1/\epsilon + \log C}{\log 1/\alpha} \rightsquigarrow \mathcal{O}(\log 1/\epsilon)$$

iterations: the closer we need to get, the more we should iterate

For this required time, we say that the optimizer converges linearly

It's a fast rate, since number of iterations is proportional to logarithm of $1/\epsilon$

Convergence Rate of Optimizers

+ *You said ideal! Isn't gradient descent always converging at this rate?*
– Well! Only when empirical risk is *strongly convex and we do full-batch training*! You can guess it happens almost never for us!
+ *But how it works with realistic NNs and SGD?*

In general, it's hard to characterize exact convergence; however, we know that *when empirical risks are rather smooth functions*[3], *we have*

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| \leqslant \frac{L}{t\eta}$$

for some L *that gets larger as* $\mathbf{w}$ *becomes larger*. So, in practice, we have

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| = \mathcal{O}(1/t)$$

[3] To be rigorous: when it's Lipschitz continuous, but we don't really need details on that

Convergence Rate of Optimizers

In practice, we have

$$|\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{\star})| = \mathcal{O}(1/t)$$

So, if we want to end up somewhere in ϵ-neighborhood of $\mathbf{w}^{\star}$, we need

$$t = \mathcal{O}(1/\epsilon)$$

This can potentially take long! We say in this case that

the optimizer converges sub-linearly

+ *How bad can it be?*
- Just set $\epsilon = 10^{-3}$: *with linear convergence we need time in order of* 3; *with sub-linear one, we need in order of* 1000!

Alternative Optimizers

Moral of Story

Vanilla gradient descent is not what we can use in practice!

In practice, we employ improved versions of gradient descent: *there is a long list of them, but we check a few important ones that are typically used*

- Gradient descent with *learning rate scheduling*
- Gradient descent with *momentum*
- *Rprop: Resilient backpropagation*
- *RMSprop: Root mean square propagation*
- *Adam: Adaptive moment estimation*

\+ *If gradient descent is not used in practice, why we did backpropagation?*

– No worries! They *all use gradient!* This is why these algorithms are commonly referred to as *gradient-based training algorithms*

SGD with Learning Rate Scheduling

We had it in simple words in Chapter 1: *we vary learning rate through time*

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta^{(t-1)} \nabla \hat{R}(\mathbf{w}^{(t-1)})$$

It's called *learning rate scheduling*: *start at large* $\eta^{(0)} = \eta$ *and reduce it with* t

---

We may schedule learning rate with various approaches

- We could have *linear decay*

$$\eta^{(t)} = \frac{\eta}{t+1}$$

- We could have *polynomial decay* with power P

$$\eta^{(t)} = \frac{\eta}{(t+1)^P}$$

- We could have *exponential decay* with some exponent rate $\kappa > 0$

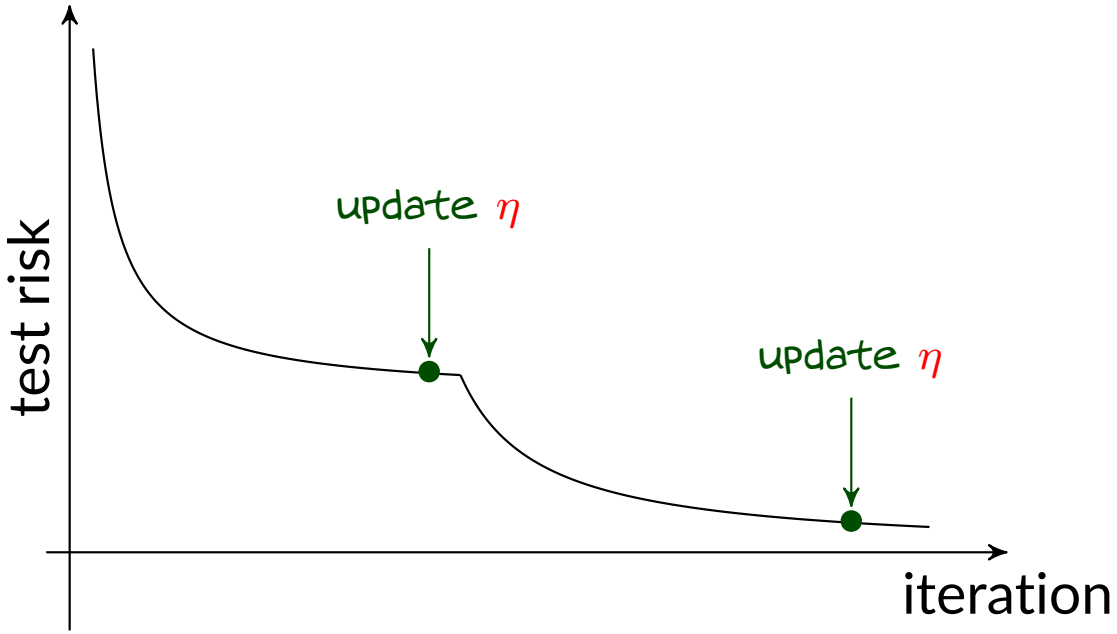
$$\eta^{(t)} = \eta \, e^{-\kappa t}$$

SGD with Learning Rate Scheduling

There is also a *more practical trick* for learning rate scheduling:

1. *Use a fixed learning rate η until the test risk saturates*
2. *Reduce the learning rate as $\eta \leftarrow \alpha\eta$ for $\alpha < 1$*
3. *Repeat the above steps until test risk arrives at a desired level*

A good choice of α is around $\boxed{\alpha = 0.1}$



SGD with Learning Rate Scheduling

Learning rate scheduling can lead us to a better local minima; however,

it does not change the convergence rate

It is nevertheless a *good approach for easy problems*

We can access pre-implemented scheduling techniques in PyTorch

```
>> import torch
>> torch.optim.lr_scheduler
```

Momentum: *Moving Average*

Momentum is one of the key approaches to robust SGD

The idea is simple: *we replace the gradient*[4] *with its moving average*

Say we are in iteration t and let the computed gradient to be $\mathbf{g}^{(t)}$, i.e.,

$$\mathbf{g}^{(t)} = \texttt{estimator}\left\{\nabla \hat{R}(\mathbf{w}^{(t)})\right\}$$ e.g., we computed by SGD

With standard gradient descent we update as

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)}\mathbf{g}^{(t)}$$

In momentum approach, we replace $\mathbf{g}^{(t)}$ with its moving average

[4] Which can largely variate, especially with small mini-batches

Momentum: *Moving Average*

Moving Average $\sim$ Momentum

Moving average with factor β in iteration t is

$$\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1-\beta)\,\mathbf{g}^{(t)}$$

Moving average has less fluctuations; hence, *it's an estimator of true gradient with less variance*

SGD with momentum does the following update

Initiate $\mathbf{m}^{(0)} = \mathbf{0}$
for $t = 1, \ldots$ **do**
  $\ldots$
  $\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1-\beta)\,\mathbf{g}^{(t)}$
  $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \mathbf{m}^{(t)}$
  $\ldots$
end for

Nesterov Momentum: *Accelerated Gradient Computation*

Nesterov approach adds an *intermediate step:when we compute gradient, we use the already-calculated momentum to estimate future weights*

Initiate $\mathbf{m}^{(0)}$
for $t = 1, \ldots$ **do**
  $\ldots$
  $\hat{\mathbf{w}} = \mathbf{w}^{(t)} - \beta \mathbf{m}^{(t-1)}$ `# approximate next point`
  Compute gradient for weights $\hat{\mathbf{w}}$: call it $\hat{\mathbf{g}}^{(t)}$
  $\mathbf{m}^{(t)} = \beta \mathbf{m}^{(t-1)} + (1 - \beta)\, \hat{\mathbf{g}}^{(t)}$
  $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta^{(t)} \mathbf{m}^{(t)}$
  $\ldots$
end for

You can combine these lines into a single line of update

that may looks a bit more complicated, but it's the same thing!

SGD with Momentum: *Implementation*

In PyTorch, SGD is already implemented with momentum

```
>> import torch
>> torch.optim.SGD()
```

When using this implementation, we can *specify momentum factor*, *i.e.*, β, and choose *whether Nesterov being applied or not*

Typical choice of momentum factor is $\beta = 0.9$, *and remember*

with $\beta = 0$, *we return to the standard SGD*

This is the default value in PyTorch

Rprop: *Resilient Backpropagation*

Rprop was introduced by *Riedmiller and Braun* in 1992; *check the paper here*

> *Riedmiller and Braun noticed that gradient descent can be improved if we could have individual learning rate in each dimension of* $\mathbf{w}$*: you may recall the first question in Assignment 1!*

They hence came up with Rprop: *let's see first one-dimensional case*

```
Rprop():
    Initiate η^(0) and choose μ^+ > 1, μ^- < 1, η_max and η_min
    for t = 1 : T do
        ...
        Compute gradient at w^(t) and call it g^(t)
        Update learning rate as η^(t) ← Rprop_Scheduler(η^(t-1), g^(t), g^(t-1))
        Update weight w^(t+1) = w^(t) − η^(t) sign(g^(t))  # only sign of gradient
        ...
    end for
```

Rprop: *Learning Rate Scheduler*

They key point of Rprop is its scheduler

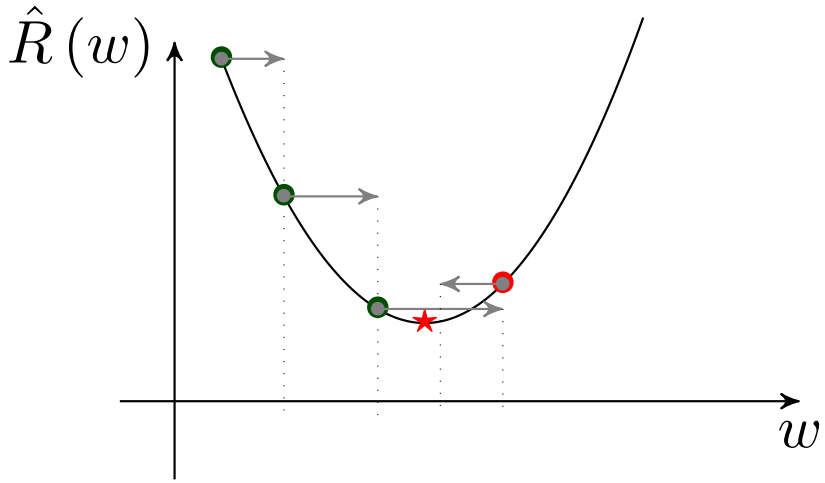
```
Rprop_Scheduler():
    Use μ⁺ > 1 and η_max as well as μ⁻ < 1 and η_min
    if sign(g^(t)) = sign(g^(t−1)) then
        Update learning rate η^(t) = min{μ⁺ η^(t−1), η_max}      # go faster
    else
        Update learning rate η^(t) = max{μ⁻ η^(t−1), η_min}      # slow down
    end if
```

It follows an intuitive strategy

- ↳ *Keep going faster as the sign of gradient is not changing*
- ↳ *Slow down only when the sign changes: you have passed the minimum!*

Rprop: *Resilient Backpropagation*

Let's look at it geometrically



In simple words, with Rprop

we keep on increasing until we pass the minimum

Rprop: *Resilient Backpropagation*

+ *How does it extend to multi-dimensional case that we have in training?*
– We apply this idea on every entry individually

Initiate η and choose μ^+, μ^-, $\eta_{\max}$ and $\eta_{\min}$
for $t = 1 : T$ **do**
  ...
  Compute gradient $\mathbf{g}^{(t)}$
  for every entry i of $\mathbf{g}^{(t)}$ **do**
    Use the sign of entry and update $\eta_i^{(t)}$ via `Rprop_Scheduler()`
    Apply one-dimensional `Rprop()` to update entry i of $\mathbf{w}^{(t)}$
  end for
  ...
end for

Rprop: *Resilient Backpropagation*

Typical choices of parameters for this algorithm are

- initial learning rate $\eta = 0.01$
- factors $\mu^+ = 1.2$ and $\mu^- = 0.5$
- $\eta_{\max}$ and $\eta_{\min}$ are less important: *they get automatically regulated*

It's better to avoid choices that $1/\mu^+ = \mu^-$, because if we pass the minimum, we don't like to move exactly the previous point

---

We can again access Rprop through module `optim` *in PyTorch*

```
>> import torch
>> torch.optim.Rprop()
```

RMSprop: *Root Mean Square Propagation*

It turns out the Rprop only works fine with full-batch training

↳ *it's because, it ignores the magnitude of gradient*

To understand why this happens, consider the *following dummy example*

Consider a one-dimensional case, i.e., $\mathbf{w} = w$*: we break the full batch into 4 mini-batches and come up with the following derivatives calculated in each step of mini-batch SGD*

$$(1) \leftrightsquigarrow 0.1 \qquad (2) \leftrightsquigarrow 0.1 \qquad (3) \leftrightsquigarrow 0.1 \qquad (4) \leftrightsquigarrow -0.5$$

We may approximately say that the derivative of full-batch risk, at the very first choice of w*, was close to zero or negative; however, Rprop*

1. *takes first three steps with larger and larger learning rates*
2. *only comes back with smaller step at last iteration*

It could be hence already lost at the last iteration!

RMSprop: *Root Mean Square Propagation*

Geoffrey Hinton in his *lecture notes*[5] came up with a solution: *we can use the idea of moving average to further normalize the learning rate according to average gradient magnitude*

this way we do not completely ignore the magnitude of gradient

---

Hinton looks differently at update rule of Rprop: recall the update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \text{sign}(\mathbf{g}^{(t)})$$

We can write alternatively as

learning rates are updated entry-wise

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \boldsymbol{\eta}^{(t)} \odot \frac{\mathbf{g}^{(t)}}{|\mathbf{g}^{(t)}|}$$

where $|\cdot|$ operates entry-wise

[5] Click to check out the lecture notes!

RMSprop: *Root Mean Square Propagation*

Hinton suggests that *we replace the denominator with*

moving average of root mean square of the gradients

This mean: *starting with some* $\mathbf{v}^{(0)} = \mathbf{0}$ *we determine in iteration* t

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta) \left|\mathbf{g}^{(t)}\right|^2$$

for some $\beta < 1$ *and normalize* $\mathbf{g}^{(t)}$ *with* $\sqrt{\mathbf{v}^{(t)}}$

Initiate $\mathbf{v}^{(0)} = \mathbf{0}$
for $t = 1 : T$ **do**
  ...
  $\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta) \left|\mathbf{g}^{(t)}\right|^2$ `# compute moving average`
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\mathbf{g}^{(t)}}{\sqrt{\mathbf{v}^{(t)}}}$ `# normalize by RMS`
  ...
end for

RMSprop: *Root Mean Square Propagation*

We may note that RMSprop can be observed as

SGD with Rprop-inspired learning rate scheduling

Note that the exact form of RMSprop has more details!

In typical implementations of RMSprop, we set

- learning rate to a some constant: *typical choice* $\eta = 0.01$
- β to be close to 1: *typical choice* $\beta > 0.9$

The complete form of RMSprop is also available in module `optim` *of PyTorch*

```
>> import torch
>> torch.optim.RMSprop()
```

Adaptive Momentum Estimation

Most recent implementations use the optimizer

Adaptive Momentum Estimation: Adam

that was proposed by *Kingma and Ba in 2015*[6]

The idea of Adam is straightforward: *it combines RMSprop with momentum*

it combines the strength of both approaches

In simple words: *Adam suggests that we use*

- *momentum for updating the weights*
- *Rprop-inspired approach for normalization* $\equiv$ *scheduling*

[6] Click to check out the original paper!

Adaptive Momentum Estimation

We can think of Adam as below

Initiate $\mathbf{m}^{(0)}$ and $\mathbf{v}^{(0)}$
for $t = 1 : T$ **do**
$\cdots$
$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1)\, \mathbf{g}^{(t)}$ # compute momentum
$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1 - \beta_2)\, |\mathbf{g}^{(t)}|^2$ # compute RMS
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \dfrac{\mathbf{m}^{(t)}}{\sqrt{\mathbf{v}^{(t)}}}$ # move with normalized momentum
$\cdots$
end for

Attention

Pseudo codes in this lecture give the main ideas: there are further details and numerical tricks to make these algorithms robust and stable in practice

Adaptive Momentum Estimation

In typical implementations of Adam, we set

- learning rate to a some constant: *typical choice* $0.001 < \eta < 0.01$
- β_1 to be close to 1: *typical choice* $\beta_1 = 0.9$
- β_2 to be closer to 1: *typical choice* $\beta_2 = 0.99$

Just check out the PyTorch implementation in the module `optim`

```
>> import torch
>> torch.optim.Adam()
```

Other Optimizers: *First Order vs Second Order*

There is a long list of modified gradient descents

Press Tab after typing `torch.optim.` *to see how long it is!*

In some particular applications, we may need to learn a new one: *it is hence good to know these two terms*

- First-order optimizers *that use only gradient, i.e., first-order derivatives*
  - ↳ *What we had in this section were all first-order*
- Second-order optimizers *also use Hessian, i.e., second-order derivatives*
  - ↳ These approaches are inspired by Newton's method *that shows convergence of gradient descent is boosted if we multiply gradient with inverse of Hessian*
  - ↳ They have typically *better convergence behavior*
  - ↳ Finding Hessian is a huge computation: *practical algorithms usually approximate Hessian; but, they still need high computation*