EMAT31530, Part 6: Practical optimization in AI

Laurence Aitchison

Now, we know how PyTorch computes gradients. However, to optimize practical neural networks, you need a bit more than just gradients. In this part, we'll see:

- Minibatching.
- SGD with momentum.
- Preconditioning in general.
  - RMSProp
  - Adam
  - AdamW
- Other considerations:
  - Local optima (their mysterious absence)

1 Minibatching

So far, we've computed the full gradient based on all the training data in one big batch computation. In deep learning, this is typically called a "full batch" method. However, remember that GPU memory is very limited, and training datasets are often very, very large. So full batch training only works in small-scale settings.

As an alternative, we usually use minibatches. A minibatch is a small subset of the training data (e.g. 128 images out of 60,000 images in the full dataset). In minibatched training, we calculate the loss, compute the gradients and update the parameters for only the minibatch.

To write down minibatched updates, we need notation for the gradient for an individual datapoint,

$$\mathbf{g}_i = \frac{d\mathcal{L}_i}{d\mathbf{w}} \tag{1}$$

where $\mathcal{L}_i$ is the loss for an individual datapoint. We can use the individual datapoint gradient to write down the minibatched gradient. The minibatched

gradient is just the gradient, averaged over datapoints in the minibatch,

$$\mathbf{g}_{\text{mb}} = \frac{1}{M} \sum_{i \text{ in mb}} \mathbf{g}_i = \frac{1}{M} \sum_{i \text{ in mb}} \frac{d\mathcal{L}_i}{d\mathbf{w}}. \tag{2}$$

Here, M is the minibatch size, and "i in mb" gives the set of indicies of datapoints in the current minibatch. However, it is quite awkward to use this expression directly in PyTorch, as, taken literally, it seems to involve backpropagating through M single-datapoint losses, $\mathcal{L}_i$. In contrast, PyTorch only allows us to (easily) work with a single loss. We therefore need to define a minibatched loss. Note that we again use an *average*, rather than a sum, as it makes the code + analytics slightly easier.

$$\mathcal{L}_{\text{mb}} = \frac{1}{M} \sum_{i \text{ in mb}} \mathcal{L}_i. \tag{3}$$

Using PyTorch to differentiate the minibatch loss gives

$$\frac{d\mathcal{L}_{\text{mb}}}{d\mathbf{w}} = \frac{d}{d\mathbf{w}} \left(\frac{1}{M} \sum_{i \text{ in mb}} \mathcal{L}_i \right) = \frac{1}{M} \sum_{i \text{ in mb}} \frac{d\mathcal{L}_i}{d\mathbf{w}} = \frac{1}{M} \sum_{i \text{ in mb}} \mathbf{g}_i = \mathbf{g}_{\text{mb}}, \tag{4}$$

which gives the required averaged gradients.

The simplest optimizer that uses minibatching is called "stochastic gradient descent", commonly abbreviated to SGD. This is very similar to gradient descent that we've seen previously, but it uses minibatched gradients, rather than the full batch gradient,

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}_{\text{mb};t}. \tag{5}$$

The "stochastic" in SGD refers to the stochasticity induced by randomness in the choice of minibatches.

This additional stochasticity raises a question: does SGD actually work? Specifically, does it converge to the minimum of the loss? Indeed, if you use a fixed learning rate, it will get close to the optimum, but won't actually exactly converge to exactly the optimum. Instead, it bounce around the optimum forever due to the "noisy" minibatched gradient estimates.

However, it turns out we can prove convergence in once setting: when the learning rate, η, goes to zero. Specifically, we can show that if you reduce the learning rate η, and at the same time, consider an increasing number number of iterations, $T(\eta) = 1/\eta$, then you expect the SGD updates to move the parameters to the same place, but the variance decreasing variance, as we in-effect end up averaging over more samples of the minibatch noise.

To define the notion of the "right expectation", we need to go back to the original full batch setting. Specifically, we're going to write the full batch loss

again as an average,

$$\mathcal{L}_{\text{fb}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i. \tag{6}$$

where N is the number of datapoints in the full dataset. The full batch gradient is again the average of individual datapoint gradients,

$$\mathbf{g}_{\text{fb}} = \frac{d\mathcal{L}_{\text{fb}}}{d\mathbf{w}} = \frac{1}{N} \sum_{i=1}^{N} \frac{d\mathcal{L}_i}{d\mathbf{w}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{g}_i. \tag{7}$$

Now, we can think of the indidivual datapoint gradient as a random variable, which is equally likely to be the gradient corresponding to any individual datapoint,

$$\mathrm{P}\left(\mathbf{g}_{\text{sd}}\right) = \frac{1}{N} \sum_{i=1}^{N} \delta(\mathbf{g}_{\text{sd}} - \mathbf{g}_i) \tag{8}$$

Critically, this random variable has the right expectation,

$$\mathrm{E}\left[\mathbf{g}_{\text{sd}}\right] = \frac{1}{N} \sum_{i=1}^{N} \mathbf{g}_i = \mathbf{g}_{\text{fb}}. \tag{9}$$

And as the minibatched gradient is just a bunch of independent realisations of $\mathbf{g}_{\text{sd}}$,

$$\mathrm{E}\left[\mathbf{g}_{\text{mb}}\right] = \mathbf{g}_{\text{fb}}. \tag{10}$$

i.e. it also has the right expectation.

Thus, we interpreted the single-datapoint, $\mathbf{g}_{\text{sd}}$, or minibatched, $\mathbf{g}_{\text{mb}}$, gradients as random variables, and proved that they have the right expectation, $\mathbf{g}_{\text{fb}}$. In language from statistics, we could consider $\mathbf{g}_{\text{sd}}$ and $\mathbf{g}_{\text{mb}}$ as estimators of $\mathbf{g}_{\text{fb}}$, and because the expectation is right, we say these estimators are "unbiased". It turns out that having unbiased gradient estimates provably works: as the learning rate goes to zero, the noise in each minibatch estimate "averages out", and the method converges. Specifically, as we reduce the learning rate, η, we consider an increasing number of timesteps,

$$T(\eta) = \frac{1}{\eta}. \tag{11}$$

The overall change in the parameter, w is,

$$\Delta w = \eta \sum_{t=1}^{T(\eta)} g_{\text{mb};t}. \tag{12}$$

Writing $\eta = 1/T(\eta)$,

$$\Delta w = \frac{1}{T(\eta)} \sum_{t=1}^{T(\eta)} g_{\mathrm{mb};t} \tag{13}$$

Thus, Δw is just the average of $T(\eta)$ independent realisations of the minibatched gradient, $g_{\mathrm{mb};t}$. The expected change in weights is thus,

$$\mathrm{E}\left[\Delta w\right] = \frac{1}{T(\eta)} \sum_{t=1}^{T(\eta)} \mathrm{E}\left[g_{\mathrm{mb};t}\right] = \frac{1}{T(\eta)} \sum_{t=1}^{T(\eta)} g_{\mathrm{fb}} = g_{\mathrm{fb}}, \tag{14}$$

which is just the full-batch gradient. Moreover, as Δw is the average of $T(\eta)$ independent realisations of the minibatch gradient, $g_{\mathrm{mb};t}$, the variance of Δw is proportional to $1/T(\eta)$, or to η (Eq. 11),

$$\mathrm{Var}\left[\Delta w\right] = \tfrac{1}{T(\eta)} \mathrm{Var}\left[g_{\mathrm{mb}}(t)\right] = \eta \, \mathrm{Var}\left[g_{\mathrm{mb}}(t)\right], \tag{15}$$

Thus, as $\eta \to 0$, the variance in the change in weights $\mathrm{Var}\left[\Delta w\right]$ also goes to zero. Hence, as $\eta \to 0$, the variance in the parameters also converges to zero, $\mathrm{Var}\left[\Delta w\right] \to 0$.

Theory is great, but deep learning is a practical subject, so there's a few points of "wisdom" and/or common practice. Specifically:

- Each pass through the full training dataset is called an "epoch". Deep learning training progress is typically measured in epochs. Each epoch is composed of many minibatches (minibatches per epoch = dataset size / minibatch size).
- Within each epoch, it works slightly better to randomise the minibatches, rather than going through the same fixed set of minibatches at each epoch again and again (that's what `shuffle=True` does).

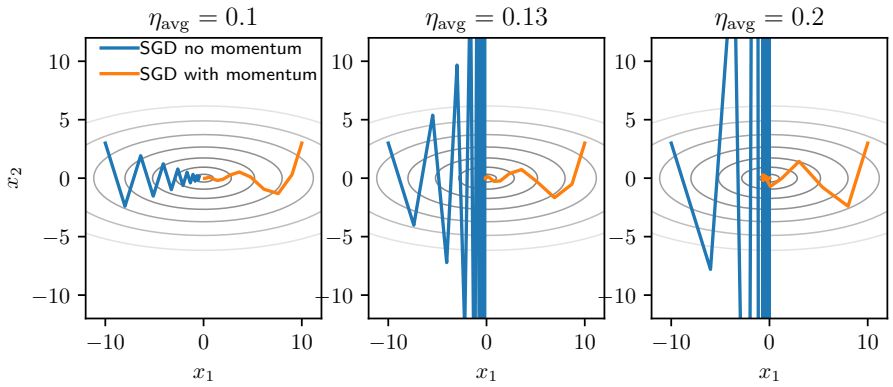
2 Momentum

The simple form of SGD described above is rarely used in practice. The problems all come down to the interactions between the size of gradients and the learning rates. Specifically:

- If all the gradients are small, then we'd like to use large learning rates to converge reasonably quickly.
- If all the gradients are large, then we'd like to use small learning rates to avoid instability.

The problem is in practice, gradients in some directions are large, and gradients in some directions are small. In practice, that means you need to use a small learning rate (to avoid instabilities along directions with large gradients). But

that inevitably means you converge slowly in directions with small gradients. Momentum is one way to mitigate these issues, as it allows you to push up the learning rate a bit, allowing faster convergence in the small-gradient directions, while avoiding instabilities in the large-gradient directions. Specifically, the instabilities often look like the diagram below,



If the learning rate is too large, SGD can jump over the optimum in that direction (in this case, we're looking at the large vertical jumps for SGD depicted in the blue line). That's okay if you end up closer to the middle than you started (i.e. left plot below with $\eta = 0.1$), in which case the gradient next time is smaller, and you will gradually converge on the right answer. But if you end up further from the middle than you started (i.e. middle and left plots below), then you'll blow up. Momentum (orange line) allows you to increase learning rates a bit while avoiding this instability.

How does momentum work? One inuition for momentum (and the origin of the term "momentum") is the analogy with Physics. Specifically, consider a ball rolling around on a hilly landscape, with friction. The objective, $\mathcal{L}(\mathbf{w})$ describes the height of the landscape at any location, $\mathbf{w}$. "Gravity" pushes the ball downhill with a force given by the gradient

$$\mathbf{F}(\mathbf{w}) = -\eta_{\text{trad}}\mathbf{g}_{\text{fb}}. \tag{16}$$

Taking $m = 1$, the equations of motion are,

$$\frac{d\mathbf{v}}{dt} = -(1-\mu)\mathbf{v} + \mathbf{F}(\mathbf{w}) = -(1-\mu)\mathbf{v} - \eta_{\text{trad}}\mathbf{g}_{\text{fb}} \tag{17a}$$

$$\frac{d\mathbf{w}}{dt} = \mathbf{v}. \tag{17b}$$

Here, the $-(1-\mu)\mathbf{v}$ term in the first equation represents friction (μ is between 0 and 1). Discretising these equations with a timestep of 1 and using the mini-

batched gradient, $\mathbf{g}_{\text{mb};t}$ in place of the full batch gradient, $\mathbf{g}_{\text{fb}}$, gives

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \frac{d\mathbf{v}}{dt} = \mu\mathbf{v}_t - \eta_{\text{trad}}\mathbf{g}_{\text{mb};t} \tag{18a}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \frac{d\mathbf{w}}{dt} = \mathbf{w}_t + \mathbf{v}_{t+1}. \tag{18b}$$

This is the parametrisation used in PyTorch (SGD docs). Typically we use a momentum of $\mu = 0.9$, while the learning rate, η_{trad}, needs extensive tuning.

However, I'm really not a fan of this parameterisation: it doesn't help us to understand what's really going on, and conflicts with how other learning rules such as Adam are written down. These other learning rules use an exponential moving average gradient,

$$\langle\mathbf{g}\rangle_{t+1} = \beta_1\langle\mathbf{g}\rangle_t + (1-\beta_1)\mathbf{g}_{\text{mb};t} \tag{19}$$

This looks alot the "velocity" updates in Eq. (18a), in that they both have an exponentially decay, with an additive term that depends on the minibatch gradient (the exponential decay is for $\mathbf{v}$ in Eq. (18a) and for $\langle\mathbf{g}\rangle$ in Eq. (19)). However, the exponential moving average is much easier to interpret, as it is estimates the full batch gradient. Specifically, if we are in a region where the underlying gradient doesn't change much,

$$\mathrm{E}\left[\langle\mathbf{g}\rangle_t\right] = \mathbf{g}_{\text{fb}} \tag{20}$$

(see exercises for details). Now, we can write an alternative set of momentum updates,

$$\langle\mathbf{g}\rangle_{t+1} = \beta_1\langle\mathbf{g}\rangle_t + (1-\beta_1)\mathbf{g}_{\text{mb};t} \tag{21a}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_{\text{trad}}\langle\mathbf{g}\rangle_{t+1} \tag{21b}$$

This is alot more similar to traditional SGD, in that the parameter updates are again just a learning rate times an estimate of the graident. In SGD, we just use a single minibatched gradient estimate, while here, we use an exponential-moving-average over previous minibatch gradient estimates. This averaging allows us to reduce alternating gradients (as in the original diagram), and stabilise learning at larger learning rates.

This raises a question: how does this "average" parameterisation relate to the "traditional" parameterisation? It turns out that the "velocity" in the traditional formulation is just proportional to the exponential moving average gradient,

$$\mathbf{v}_t = -\frac{\eta_{\text{trad}}}{1-\mu}\langle\mathbf{g}\rangle_t. \tag{22}$$

To prove this assertion, we can substitute this into the traditional momentum updates (Eq. 18a),

$$\mathbf{v}_{t+1} = \mu\mathbf{v}_t - \eta_{\text{trad}}\mathbf{g}_{\text{mb};t} \tag{23}$$

$$-\frac{\eta_{\text{trad}}}{1-\mu}\langle\mathbf{g}\rangle_{t+1} = -\frac{\eta_{\text{trad}}}{1-\mu}\mu\langle\mathbf{g}\rangle_t - \eta_{\text{trad}}\mathbf{g}_{\text{mb};t} \tag{24}$$

And dividing everything by $-\frac{\eta_{\text{trad}}}{1-\mu}$,

$$\langle \mathbf{g} \rangle_{t+1} = \mu \langle \mathbf{g} \rangle_t + (1-\mu)\mathbf{g}_{\text{mb};t}. \tag{25}$$

This is exactly the exponential moving average gradient in Eq. (21a), with $\beta_1 = \mu$.

Additionally, we can relate the learning rates in the two formulations. Specifically, the traditional parameter updates (Eq. 18b) are,

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1} = \mathbf{w}_t - \frac{\eta_{\text{trad}}}{1-\mu}\langle \mathbf{g} \rangle_{t+1}. \tag{26}$$

This is equivalent to the average parameter updates (Eq. 21b), if we set

$$\eta_{\text{avg}} = \frac{\eta_{\text{trad}}}{1-\mu}. \tag{27}$$

That means if we increase momentum (typically we use $\mu = 0.9$) while keeping the traditional momentum, η_{trad}, the same, then we are in effect increasing the learning rate, η_{avg}.

3 RMSProp

If you remember, the original problem we were solving with momentum was that there are big gradients in some directions, and small gradients in other directions. You ideally want a big learning rate in directions with a small gradient, to converge quickly. At the same time, you want a small learning rate in directions with a big gradient, to avoid instability. RMSProp gives you just that! Specifically, RMSProp updates for a single parameter, w, are,

$$\left\langle g^2 \right\rangle_{t+1} = \beta_2 \left\langle g^2 \right\rangle_t + (1-\beta_2) g_t^2 \tag{28}$$

$$\hat{v}_{t+1} = \frac{g_{\text{mb};t}^2}{1-(\beta_2)^t} \tag{29}$$

$$w_{t+1} = w_t + \eta \frac{g_{\text{mb};t}}{\sqrt{\hat{v}_{t+1}} + \epsilon} \tag{30}$$

where g_t is the minibatch gradient at iteration t. You can understand this as being like SGD, but where

so $\left\langle g^2 \right\rangle_{t+1}$ is an exponential moving average of the squared gradients, g^2. Now, the update to the weights is always about η. If we assume that we're in a small region, where the gradient is always the same, $g_t = g_{\text{fb}}$, then, $\left\langle g^2 \right\rangle_{t+1} = g_{\text{fb}}^2$, so the weight update is,

$$w_{t+1} = w_t + \eta \frac{g_{\text{fb}}}{\sqrt{g_{\text{fb}}^2} + \epsilon} \tag{31}$$

And $\sqrt{g_{\text{fb}}^2} = |g_{\text{fb}}|$,

$$w_{t+1} = w_t + \eta \frac{g_{\text{fb}}}{|g_{\text{fb}}| + \epsilon} \tag{32}$$

For very small ϵ,

$$w_{t+1} = w_t + \eta \, \text{sign}(g_{\text{fb}}) \tag{33}$$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } 0 < x \\ 0 & \text{if } 0 = x \\ -1 & \text{if } 0 > x \end{cases} \tag{34}$$

So the magnitude of the updates is always around η, no matter how big the gradients are.

4 Adam

Adam is just RMSProp + Momentum.

5 Appendix [Non examinable]

5.1 Exponential moving averages

The exponential moving average gradient estimator was,

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1 \langle \mathbf{g} \rangle_t + (1 - \beta_1) \mathbf{g}_t \tag{35}$$

But in what sense is this an "exponential moving average"?

We know,

$$\langle \mathbf{g} \rangle_t = \beta_1 \langle \mathbf{g} \rangle_{t-1} + (1 - \beta_1) \mathbf{g}_{t-1} \tag{36}$$

$$\langle \mathbf{g} \rangle_{t-1} = \beta_1 \langle \mathbf{g} \rangle_{t-2} + (1 - \beta_1) \mathbf{g}_{t-2} \tag{37}$$

$$\langle \mathbf{g} \rangle_{t-2} = \beta_1 \langle \mathbf{g} \rangle_{t-3} + (1 - \beta_1) \mathbf{g}_{t-3} \tag{38}$$

Now, substituting $\langle \mathbf{g} \rangle_t$ into $\langle \mathbf{g} \rangle_{t+1}$,

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1 (\beta_1 \langle \mathbf{g} \rangle_{t-1} + (1 - \beta_1) \mathbf{g}_{t-1}) + (1 - \beta_1) \mathbf{g}_t \tag{39}$$

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1^2 \langle \mathbf{g} \rangle_{t-1} + (1 - \beta_1)(\mathbf{g}_t + \beta_1 \mathbf{g}_{t-1}). \tag{40}$$

Now, substituting $\langle \mathbf{g} \rangle_{t-1}$ into this expression,

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1^2 (\beta_1 \langle \mathbf{g} \rangle_{t-2} + (1 - \beta_1) \mathbf{g}_{t-2}) + (1 - \beta_1)(\mathbf{g}_t + \beta_1 \mathbf{g}_{t-1}) \tag{41}$$

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1^3 \langle \mathbf{g} \rangle_{t-2} + (1 - \beta_1)(\mathbf{g}_t + \beta_1 \mathbf{g}_{t-1} + \beta^2 \mathbf{g}_{t-2}). \tag{42}$$

If we continued doing this, we'd get,

$$\langle \mathbf{g} \rangle_{t+1} = \beta_1^{t+1} \langle \mathbf{g} \rangle_1 + (1 - \beta_1) \sum_{\tau=0}^{t-1} \beta_1^\tau \mathbf{g}_{t-\tau} \tag{43}$$

Additionally, we assume that we initialize $\langle \mathbf{g} \rangle_1 = \mathbf{0}$,

$$\langle \mathbf{g} \rangle_{t+1} = (1 - \beta_1) \sum_{\tau=0}^{t-1} \beta_1^\tau \mathbf{g}_{t-\tau} \tag{44}$$

This can be interpreted as a weighted average,

$$\langle \mathbf{g} \rangle_{t+1} = \sum_{\tau=0}^{t-1} w_\tau \mathbf{g}_{t-\tau} \tag{45}$$

with weights,

$$w_\tau = (1 - \beta_1)\beta_1^\tau \tag{46}$$

For this to be an exponential moving average, we need the sum of the weights to be 1.

$$\sum_{\tau=0}^{t-1} w_\tau = (1 - \beta_1) \underbrace{\sum_{\tau=0}^{t-1} \beta_1^\tau}_{\text{geometric series}} \tag{47}$$

For geometric series, we know,

$$\sum_{k=0}^{n-1} r^k = \frac{1 - r^n}{1 - r} \tag{48}$$

(for $0 \leq r < 1$). Therefore,

$$\sum_{\tau=0}^{t-1} w_\tau = (1 - \beta_1) \frac{1 - \beta_1^t}{1 - \beta_1} = 1 - \beta_1^t \tag{49}$$

As $0 \leq \beta_1 < 1$, it will quickly approach 0 as t gets large. In that case, the sum of weights approaches 1, and it makes sense to regard Eq. (19) as an exponential weighted average.