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,

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

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.

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

Using PyTorch to differentiate the minibatch loss gives

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

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,

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

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

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

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,

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

Critically, this random variable has the right expectation,

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

And as the minibatched gradient is just the average of a bunch of independent realisations of $\mathbf{g}_{\text{sd}}$ (Eq. 2),

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

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 of $\mathbf{g}_{\text{sd}}$ and $\mathbf{g}_{\text{mb}}$ equals the desired value, $\mathbf{g}_{\text{fb}}$, 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". To prove this kind of convergence, we consider reducing the learning rate η, and at the same time, increasing number of timesteps,

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

Note that we make sure to choose values of η that give integer numbers of steps $T(\eta)$. The overall change in the parameter, w is,

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

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

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

Thus, Δw is just the average of $T(\eta)$ independent realisations of the minibatched gradient, $g_{\text{mb};t}$. Assuming that we consider a small number of iterations, such that the expected gradient doesn't change much across the small part of the trajectory considered (i.e. $\mathrm{E}\left[g_{\text{mb};t}\right] = g_{\text{fb}}$, where g_{fb} doesn't change over timesteps), the expected change in weights becomes,

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

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

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

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

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

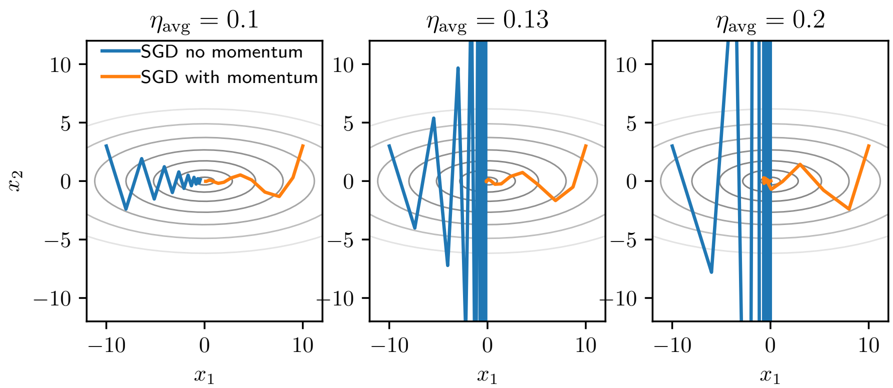
- Each pass through the full training dataset is called an "epoch". Deep learning training progress is typically measured in epochs. Each epoch is composed of many minibatches (minibatches per epoch = dataset size / minibatch size).
- Within each epoch, it works slightly better to randomise the minibatches, 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. That sets up a conflict: for directions with a large gradient, you'd like to use a small learning rate (to avoid instabilities along directions with large gradients). But for for directions with a small gradient, you'd like to use a large learning rate (to converge reasonably quickly). Momentum is one way to mitigate these issues, as it stabilises learning with large-ish learning rates along high gradient directions. That allows you to push up the learning rate a bit, allowing faster convergence in the small-gradient directions, before you hit instabilities in the large-gradient directions. Specifically, the instabilities often look like the diagram below,



If the learning rate is large, SGD can oscillate, as at each step, it jumpts over the "horizontal" in these diagrams. That's okay if the oscillations decay (i.e. left plot below with $\eta = 0.1$), in which case you will still converge. But as the learning rate increases, the oscillations grow, and the learning process diverges (middle and right plots). 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

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

(Note that I'm using two slightly different notions of gradient, η_{trad}, the "traditional" gradient used in standard formulations of SGD with momentum that you'll find on the internet, and η_{avg}: a slightly different notion of momentum that I think makes the effect of momentum far clearer). Taking the mass to be

1 for simplicity, 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 minibatched 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, this form is much easier to interpret, as it is an exponential moving average of minibatch gradients that just estimates the full batch gradient (see exercises for details). That's why we denote this quantity $\mathbf{g}_{\text{mb};t}$ (angle brackets are often used to denote some type of average). Now, we can write an alternative set of momentum updates,

$$\langle\mathbf{g}\rangle_{t+1} = \beta_1\langle\mathbf{g}\rangle_t + (1-\beta_1)\mathbf{g}_{\text{mb};t} \tag{20a}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_{\text{avg}}\langle\mathbf{g}\rangle_{t+1} \tag{20b}$$

This is alot more similar to traditional SGD, in that the parameter updates are again just a learning rate times an estimate of the gradient. 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? Our goal is to find relationships between the so-called "hyperparameters" in the traditional formulation (i.e. μ and η_{trad}) and the hyperparameters in the alternative formulation (i.e. β_1 and η_{avg}) that give exactly the same parameter updates at all timesteps. Specifically, we need the parameter update from the traditional formulation (Eq. 18b) equal to the update from the alternative formulation (Eq. 20b) at timestep t and $t+1$,

$$\mathbf{v}_t = -\eta_{\text{avg}} \langle \mathbf{g} \rangle_t \tag{21}$$

$$\mathbf{v}_{t+1} = -\eta_{\text{avg}} \langle \mathbf{g} \rangle_{t+1}. \tag{22}$$

This indicates that we need the momentum in the traditional formulation to be directly proportional to the average gradient. To achieve that, lets substitute the updates for $\mathbf{v}_{t+1}$ (Eq. 18a) and for $\langle \mathbf{g} \rangle_{t+1}$ (Eq. 20a) into Eq. (22),

$$\begin{aligned} \mu \mathbf{v}_t - \eta_{\text{trad}} \mathbf{g}_{\text{mb};t} &= -\eta_{\text{avg}} \left(\beta_1 \langle \mathbf{g} \rangle_t + (1-\beta_1) \mathbf{g}_{\text{mb};t} \right) \\ &= -\eta_{\text{avg}} \beta_1 \langle \mathbf{g} \rangle_t - \eta_{\text{avg}} (1-\beta_1) \mathbf{g}_{\text{mb};t}. \end{aligned} \tag{23}$$

Now, lets substitute the value of $\mathbf{v}_t$ in terms of the average gradient (Eq. 21),

$$-\mu \eta_{\text{avg}} \langle \mathbf{g} \rangle_t - \eta_{\text{trad}} \mathbf{g}_{\text{mb};t} = -\eta_{\text{avg}} \beta_1 \langle \mathbf{g} \rangle_t - \eta_{\text{avg}} (1-\beta_1) \mathbf{g}_{\text{mb};t}. \tag{24}$$

Now, the minibatch gradient, $\mathbf{g}_{\text{mb};t}$ is a random variable that could (in principle) be anything, and is not e.g. deterministically tied to $\langle \mathbf{g} \rangle_t$. Thus, we're going to consider the equality of the terms proportional to the minibatch gradient, and the terms proportional to $\langle \mathbf{g} \rangle_t$ separately. For the terms proportional to $\langle \mathbf{g} \rangle_t$ to be equal, we need,

$$-\mu \eta_{\text{avg}} \langle \mathbf{g} \rangle_t = -\eta_{\text{avg}} \beta_1 \langle \mathbf{g} \rangle_t. \tag{25}$$

Thus,

$$\mu = \beta_1. \tag{26}$$

And for the terms proportional to $\mathbf{g}_{\text{mb};t}$ to be equal,

$$-\eta_{\text{trad}} \mathbf{g}_{\text{mb};t} = -\eta_{\text{avg}} (1-\beta_1) \mathbf{g}_{\text{mb};t}. \tag{27}$$

Thus,

$$\eta_{\text{trad}} = \eta_{\text{avg}} (1-\beta_1) \qquad \eta_{\text{avg}} = \tfrac{1}{1-\beta_1} \eta_{\text{trad}}. \tag{28}$$

That means if we increase momentum (typically we use $\mu = \beta_1 = 0.9$) 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. Momentum mitigated the problem, by stabilising along high-gradient directions. But it is also possible to just directly reduce the gradients in the high-gradient directions. And that's just what RMSProp gives you! Specifically, RMSProp computes an exponential moving average of the squared gradient, $g^2_{\text{mb};t}$ then uses the root-mean-square-average gradient to normalize (or adapt) the learning rate,

$$\left\langle g^2 \right\rangle_{t+1} = \beta_2 \left\langle g^2 \right\rangle_t + (1-\beta_2) g^2_{\text{mb};t} \tag{29}$$

$$w_{t+1} = w_t + \frac{\eta}{\sqrt{\left\langle g^2 \right\rangle_{t+1} + \epsilon}} g_{\text{mb};t}. \tag{30}$$

Note that ϵ is a small positive constant (usually set to 10^{-8}) which ensures that we never divide-by-zero.

To understand how RMS prop functions, we assume that we're in a small region, where the gradient is always the same, $g_{\text{mb};t} = g_{\text{fb}}$, and if we're in this region for long enough, $\left\langle g^2 \right\rangle_{t+1} = g^2_{\text{fb}}$. Now, the weight update is,

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

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

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

Setting $\epsilon = 0$ for simplicity,

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

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

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

4 Adam

In practice, RMSProp is rarely, if ever used. Instead, Adam is perhaps the most common optimizer in deep learning. Critically, Adam is just RMSProp + Momentum + Bias correction. Note that the name "Adam" breaks down as "Ada-m", with the "Ada" for "adaptive" learning rates, and the "m" for momentum. To combine RMSProp with momentum, we just replace the gradient,

$g_{\text{mb};t}$, in Eq. (30) with the exponential moving average gradient,

$$\langle g \rangle_{t+1} = \beta_1 \langle g \rangle_t + (1-\beta_1) g_{\text{mb};t} \tag{35}$$

$$\left\langle g^2 \right\rangle_{t+1} = \beta_2 \left\langle g^2 \right\rangle_t + (1-\beta_2) g^2_{\text{mb};t} \tag{36}$$

$$w_{t+1} = w_t + \frac{\eta}{\sqrt{\left\langle g^2 \right\rangle_{t+1}} + \epsilon} g_{\text{mb};t}. \tag{37}$$

However, there's another problem with these updates. Specifically, $\left\langle g^2 \right\rangle_{t+1}$ is too small for about the first 1000 timesteps. In particular, we typically use a long timescale for the exponential moving average for $\left\langle g^2 \right\rangle_{t+1}$: the default is $\beta_2 = 0.999$, which implies a timescale of about 1000 timesteps. Moreover, in the absence of any better information, we initialize $\left\langle g^2 \right\rangle_0 = 0$. Combining all this, we have $1 - \beta_2 = \frac{1}{1000}$, so

$$\left\langle g^2 \right\rangle_1 = \tfrac{1}{1000} g^2_{\text{mb};t=1}, \tag{38}$$

thus the first estimate of $\left\langle g^2 \right\rangle$ is about 1000 times too small. To fix this issue, the full Adam updates do bias correction:

$$\langle g \rangle_{t+1} = \beta_1 \langle g \rangle_t + (1-\beta_1) g_{\text{mb};t} \tag{39}$$

$$\left\langle g^2 \right\rangle_{t+1} = \beta_2 \left\langle g^2 \right\rangle_t + (1-\beta_2) g^2_{\text{mb};t} \tag{40}$$

$$\hat{m}_{t+1} = \frac{\left\langle g^2 \right\rangle_{t+1}}{1-(\beta_1)^t} \tag{41}$$

$$\hat{v}_{t+1} = \frac{\langle g \rangle_{t+1}}{1-(\beta_2)^t} \tag{42}$$

$$w_{t+1} = w_t + \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1}. \tag{43}$$

To understand what bias correction is doing, see Exercises.

5 Exercises

Exercise 1. *Consider an exponential moving average of random variables X_{t+1},*

$$\langle X \rangle_{t+1} = \beta \langle X \rangle_t + (1-\beta) X_{t+1} \tag{44}$$

where we initialize at,

$$\langle X \rangle_0 = 0 \tag{45}$$

and the means of all the X_{t+1}'s are μ,

$$\mathrm{E}\left[X_{t+1}\right] = \mu. \tag{46}$$

Part 1: What is $\mathrm{E}\left[\langle X \rangle_{t+1}\right]$ after an arbitrary number of timesteps?

Part 2: Can I multiply by a time-dependent constant, c_{t+1} to form a "debiased" estimate,

$$d_{t+1} = c_{t+1} \langle X \rangle_{t+1} \tag{47}$$

such that

$$\mathrm{E}\left[d_{t+1}\right] = \mu. \tag{48}$$

Answer 1. *Part 1: We begin by taking the expectation of the update (Eq. 44)*

$$\mathrm{E}\left[\langle X \rangle_{t+1}\right] = \beta\,\mathrm{E}\left[\langle X \rangle_t\right] + (1-\beta)\,\mathrm{E}\left[X_{t+1}\right]. \tag{49}$$

Substituting Eq. (46)*,*

$$\mathrm{E}\left[\langle X \rangle_{t+1}\right] = \beta\,\mathrm{E}\left[\langle X \rangle_t\right] + (1-\beta)\mu. \tag{50}$$

To make the notation slightly "lighter", use $a_t = \mathrm{E}\left[\langle X \rangle_t\right]$,

$$a_{t+1} = \beta a_t + (1-\beta)\mu. \tag{51}$$

We also have,

$$a_t = \beta a_{t-1} + (1-\beta)\mu. \tag{52}$$
$$a_{t-1} = \beta a_{t-2} + (1-\beta)\mu. \tag{53}$$

Substituting a_t into a_{t+1},

$$a_{t+1} = \beta(\beta a_{t-1} + (1-\beta)\mu) + (1-\beta)\mu. \tag{54}$$
$$a_{t+1} = \beta^2 a_{t-1} + (1-\beta)(1+\beta)\mu \tag{55}$$

Substituting a_{t-1},

$$a_{t+1} = \beta^2(\beta a_{t-2} + (1-\beta)\mu) + (1-\beta)\mu(1+\beta) \tag{56}$$
$$a_{t+1} = \beta^3 a_{t-2} + (1-\beta)\mu(1+\beta+\beta^2). \tag{57}$$

Repeating until we get to a_0,

$$a_{t+1} = \beta^3 a_0 + (1-\beta)\mu \sum_{\tau=0}^{t} \beta^\tau. \tag{58}$$

As we initialize at $a_0 = 0$,

$$a_{t+1} = (1-\beta)\mu \sum_{\tau=0}^{t} \beta^\tau. \tag{59}$$

Now, the $\sum_{\tau=0}^{t} \beta^\tau$ *term is a geometric series. You can look geometric series up. Or we can find the value ourselves. There's a "trick" to the proof for a geometric series, which is to consider,*

$$(1-\beta) \sum_{\tau=0}^{t} \beta^\tau = \sum_{\tau=0}^{t} \beta^\tau - \beta \sum_{\tau=0}^{t} \beta^\tau \tag{60}$$

Now we can absorb the β *into the sum,*

$$(1-\beta) \sum_{\tau=0}^{t} \beta^\tau = \sum_{\tau=0}^{t} \beta^\tau - \sum_{\tau=1}^{t+1} \beta^\tau \tag{61}$$

Now, notice that almost all the terms cancel, except those for $\tau = 0$ *and* $\tau = T + 1$*,*

$$(1-\beta) \sum_{\tau=0}^{t} \beta^\tau = 1 - \beta^{t+1}. \tag{62}$$

Thus,

$$\sum_{\tau=0}^{t} \beta^\tau = \frac{1 - \beta^{t+1}}{1 - \beta}. \tag{63}$$

Substituting this back into our form for a_{t+1} *gives,*

$$a_{t+1} = (1 - \beta^{t+1})\mu \tag{64}$$

Of course, assuming that $0 < \beta < 1$*, then as* $t + 1$ *goes to infinity, we have,*

$$\lim_{t\to\infty} \beta^{t+1} = 0 \lim_{t\to\infty} a_{t+1} \qquad = \mu \tag{65}$$

The problem is what happens for smaller numbers of timesteps.

Part 2: Therefore, we can choose,

$$c_{t+1} = \frac{1}{1 - \beta^{t+1}} \tag{66}$$

Now, the expectation of our debiased estimator, d_{t+1} *becomes,*

$$\mathrm{E}\left[d_{t+1}\right] = c_{t+1} \, \mathrm{E}\left[\langle X \rangle_{t+1}\right] \tag{67}$$

Substituting $a_{t+1} = \mathrm{E}\left[\langle X \rangle_{t+1}\right]$

$$\mathrm{E}\left[d_{t+1}\right] = c_{t+1} a_{t+1} \tag{68}$$

Substituting for c_{t+1} *from Eq.* (66) *and* a_{t+1} *from Eq.* (64)*,*

$$\mathrm{E}\left[d_{t+1}\right] = \frac{1}{1 - \beta^{t+1}} (1 - \beta^{t+1})\mu = \mu. \tag{69}$$

So d_{t+1} *has the correct expectation.*