EMAT31530: Overfitting, and other topics in NN training

Laurence Aitchison

I advise that you take a look at the iPython notebook for this week first, as it discusses overfitting in the simpler setting of linear models. We take the next step, by discussing Here, we discuss overfitting in the more complex setting of neural networks and how to mitigate overfitting in these models.

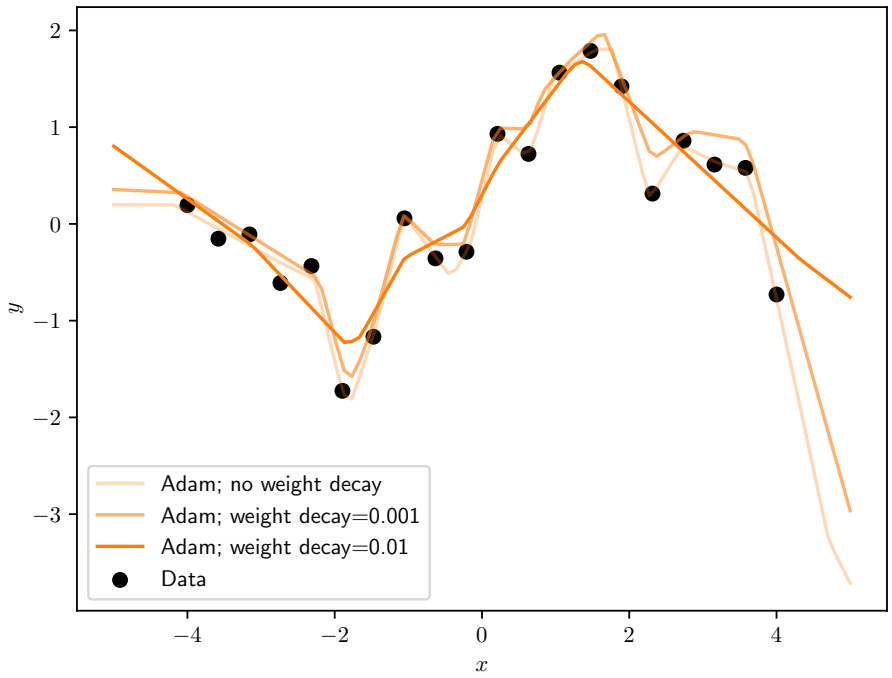
1 Overfitting is not catastrophic in NNs

In the 1990's, neural networks were largely ignored: only a few people like Yann LeCun were still working on them. One of the reasons was the following reasoning:

- Neural networks have loads (and loads) of parameters.
- Models with loads of parameters tend to overfit.
- So neural networks should overfit really badly.

That led many people to expect that overfitting in NNs would be catastropic. In actuality, it turns out that overfitting in NNs isn't catastropic at all: if you ignore overfitting while training your neural net, you should get a pretty reasonable fit. But overfitting is still an issue, and there are a large number of approaches to mitigating overfitting in neural networks. That's what we'll look at this week.

2 Weight decay



Perhaps the most standard approach to mitigating overfitting is weight-decay. In weight decay, we add a term to the loss, penalising large weights,

$$\mathcal{L}_{\text{reg}} = \mathcal{L}_{\text{unreg}} + \tfrac{\lambda}{2} \sum_i w_i^2. \tag{1}$$

Here, $\mathcal{L}_{\text{unreg}}$ is the original unregularised loss (e.g. just the cross-entropy), and $\mathcal{L}_{\text{reg}}$ is the regularised loss. To keep consistency with the PyTorch docs, we assume that $\mathcal{L}_{\text{reg}}$ and $\mathcal{L}_{\text{unreg}}$ are either:

- the loss for a single minibatch.
- the *average* loss across minibatches.

Importantly, this doesn't yet look like weight "decay".

To understand where "weight decay" arises, consider the gradients for a single parameter, w,

$$g_{\text{unreg}} = \frac{d\mathcal{L}_{\text{unreg}}}{dw} \tag{2}$$

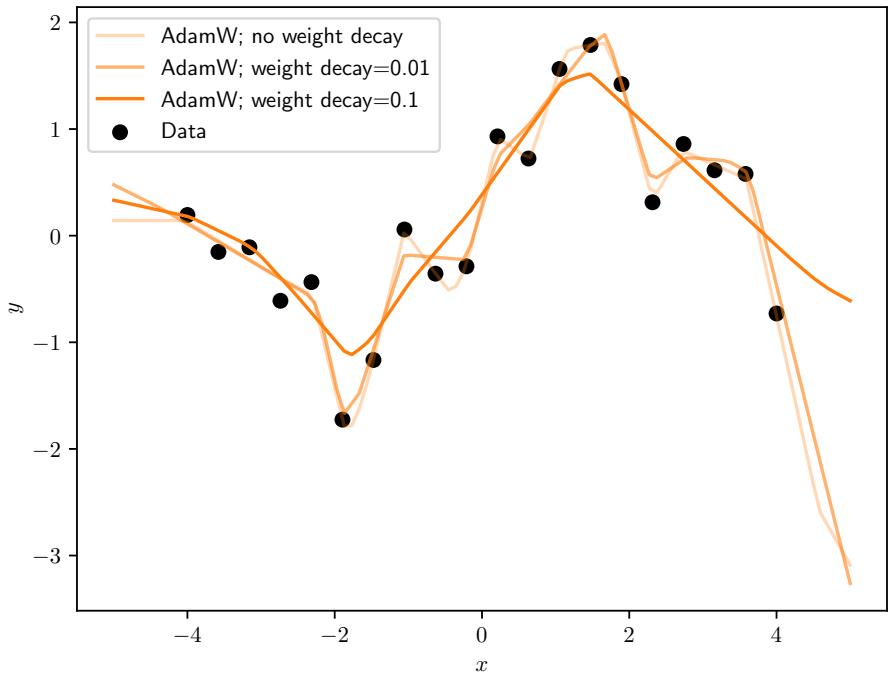
$$g_{\text{reg}} = \frac{d\mathcal{L}_{\text{reg}}}{dw} = \frac{d\mathcal{L}_{\text{unreg}}}{dw} + \lambda w \tag{3}$$

Thus, the update becomes,

$$\Delta w = \underbrace{-\eta \frac{d\mathcal{L}_{\text{unreg}}}{dw}}_{\text{original update}} \underbrace{-\eta\lambda w}_{\text{decay term}} \tag{4}$$

Now, the $-\eta\lambda w$ is a decay term that pushes w towards zero.

3 "Decoupled" weight decay, or AdamW



Almost all of deep learning is framed as optimization of loss functions. That's why weight decay, which is fundamentally defined by Eq. (1) makes alot of sense. However, there is an alternative type of weight decay, which is these days used alot in training / finetuning transformers. In particular, consider what happens with standard weight decay and Adam. Remember that standard weight decay always works in terms of a modified loss. So what happens when we put gradients of this modified loss into RMSProp?

$$\Delta w = -\eta \frac{g_{\text{reg}}}{\sqrt{\langle g_{\text{reg}}^2 \rangle}} \tag{5}$$

Substituting for the gradient of the regularised loss,

$$\Delta w = -\eta \frac{g_{\text{unreg}} + \lambda w}{\sqrt{\langle g_{\text{reg}}^2 \rangle}} \tag{6}$$

And splitting up terms,

$$\Delta w = \underbrace{-\eta \frac{g_{\text{unreg}}}{\sqrt{\langle g_{\text{reg}}^2 \rangle}}}_{\text{original update}} \quad \underbrace{-\eta\lambda \frac{w_i}{\sqrt{\langle g_{\text{reg}}^2 \rangle}}}_{\text{Adam weight decay}} \tag{7}$$

One thing you'll notive is that the updates no longer have a "simple" weight decay structure. Instead, the weight decay is normalized by the squared gradient. "Decoupled" weight decay (or AdamW) is what you get when you remove the normalization (and you use the unregularised gradients in the denominator),

$$\Delta w = \underbrace{-\eta \frac{\langle g_{\text{unreg}} \rangle}{\sqrt{\langle g^2_{\text{unreg}} \rangle}}}_{\text{original update}} \underbrace{-\eta \lambda w_i}_{\text{"decoupled" weight decay}} \tag{8}$$

This is *super-weird* from the perspective of standard deep learning, as it can no longer be understood as gradient descent on a modified objective. So why bother introducing AdamW? Well, it seems to work well in particular cases like transformers, which is reason-enough for the deep learning community.

But we can go a bit further than that. In particular, consider a single parameter, with a gradient, g, that's constant across minibatches and constant across a large range of parameters. We can now solve for the steady-state value of w. For Adam, as everything is framed in terms of an objective. Thus, we can find the steady-state by finding the location at which the gradients are zero,

$$0 = g_{\text{reg}} = g + \lambda w \tag{9}$$

$$w = -\frac{g}{\lambda} \tag{10}$$

So stronger gradients imply bigger weights. In contrast, AdamW can't be framed in terms of an objective, so we need to work with the full updates,

$$\Delta w = -\eta \frac{\langle g_{\text{unreg}} \rangle}{\sqrt{\langle g^2_{\text{unreg}} \rangle}} - \eta \lambda w_i. \tag{11}$$

Following the analysis of RMSProp last week, we're going to take,

$$\frac{\langle g_{\text{unreg}} \rangle}{\sqrt{\langle g^2_{\text{unreg}} \rangle}} \approx \text{sign}(g_{\text{unreg}}), \tag{12}$$

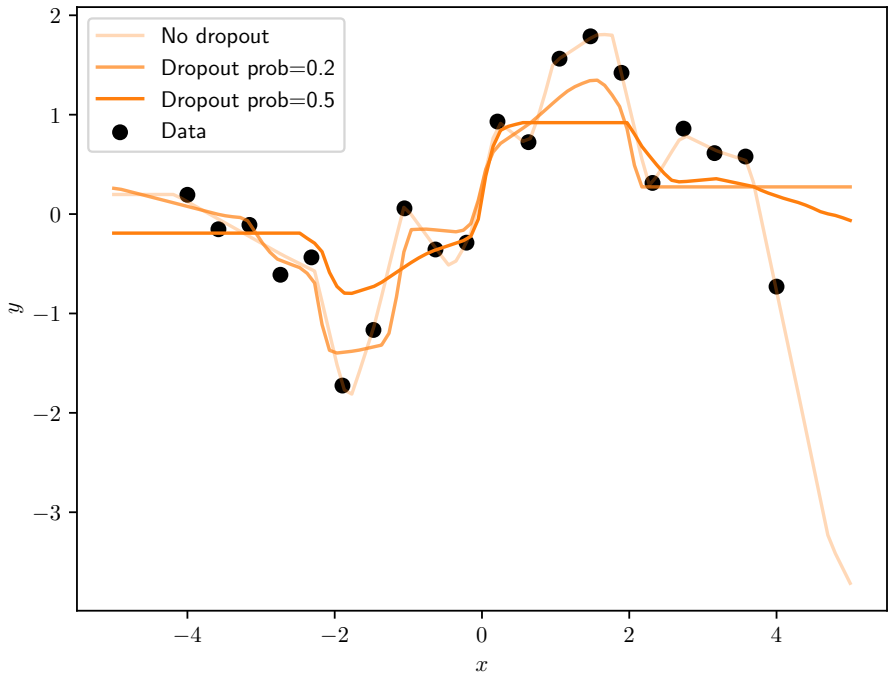
Thus, at steady-state,

$$0 \approx -\eta \text{sign}(g_{\text{unreg}}) - \eta \lambda w_i, \tag{13}$$

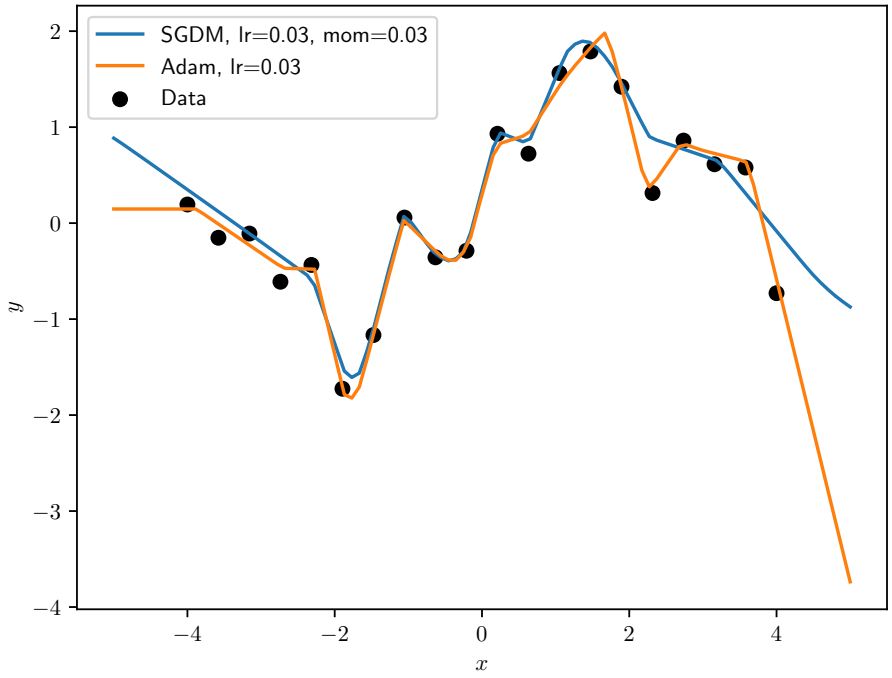
$$w_i \approx \frac{\text{sign}(g_{\text{unreg}})}{\lambda}. \tag{14}$$

So the steady-state weights for AdamW point in the same direction as the gradients, but they don't scale with the gradients!

4 Dropout


No dropout
Dropout prob=0.2
Dropout prob=0.5
Data
2
1
0
−1
−2
−3
y
−4
−2
0
2
4
x

5 Adam vs SGD



Here, I ran both Adam and SGD for a long time (1000 iterations). It seems that:

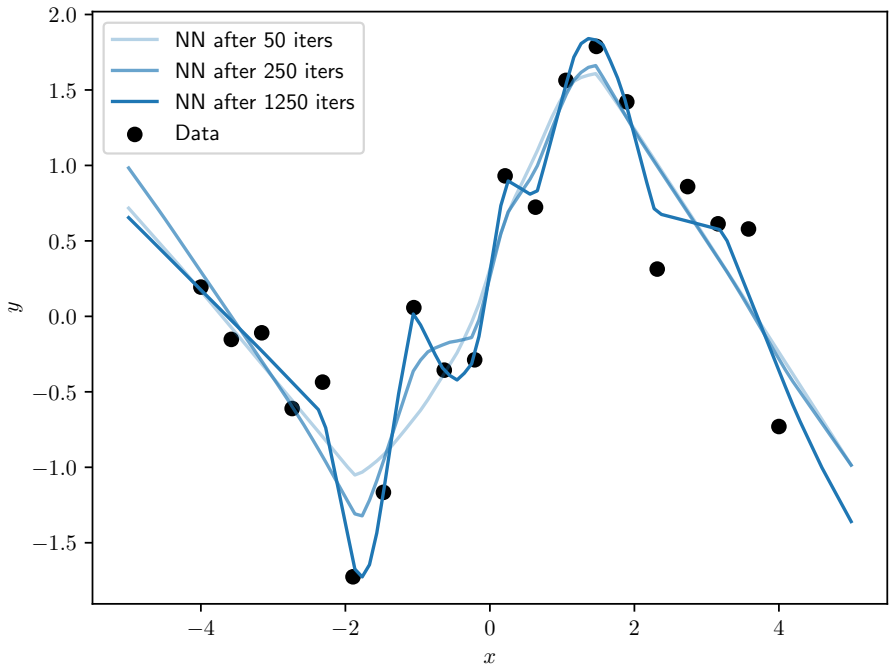
- Adam is better at capturing "fine details" than SGD.
- Adam tends to overfit more than SGD.

Of course, these are really just two sides of the same coin: overfitting *is* capturing the fine details (specifically, fine details in the noise). Moreover, note that these patterns obviously depend on the "hyperparameters" (e.g. learning rate and momentum). But this is the general "rule of thumb" that does seem to hold very generally in NNs.

So which is best Adam or SGD? Depends on whether you care about mitigating overfitting or capturing fine-details.

- SGD tends to be better for ConvNets (which we'll look at next week) for image recognition, as it mitigates overfitting.
- Adam tends to be better for transformers (which we'll look at in a couple of weeks), RL where routing information in complex ways is important.

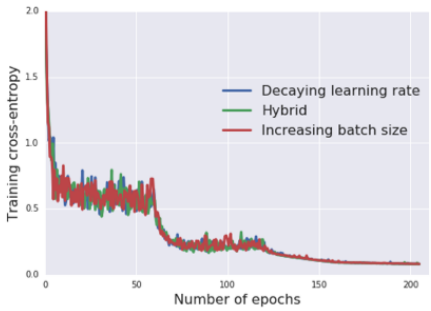
6 Early stopping



Here, I fitted a three-layer network with width 100 hidden layer, to 20 datapoints drawn from $\sin(x) + \text{noise}$. I used SGD with a learning rate of 0.01, and momentum of 0.8. I then plotted the prediction at different points throughout training.

Late on in training (darker lines), the line basically goes through all the datapoints. Strictly speaking, this is overfitting, as its fitting the noise rather than the "real underlying" sin function (especially see the "dip" about $x = -1$. But this is still a pretty reasonable prediction, e.g. nothing has blown up. Importantly, earlier in training (lighter lines) the function is smooth, it gradually gets more "details" as training progresses. So we can use "early stopping" as a regularisation technique! And early stopping is great because it gives us a nice excuse to cut training short (so training with early-stopping takes less time).

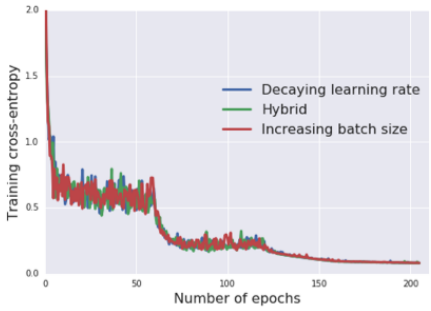
7 Minibatch noise



(a)

Typical practice in training neural networks involves starting with a high learning rate, then decaying the learning rate in steps. In the above plot (blue line), the learning rate is decayed by a factor of 5 in steps at epochs 60 and 120 (epochs is a measure of time through training; one epoch means we've gone over all datapoints once). Notice that as you decay the learning rate, the training cross-entropy drops quickly, then becomes somewhat static. What seems to be going on is that minibatch noise stops you from converging to the actual optimum. to the real Instead, minibatch noise causes you to bounce around the solution. You can reduce the effective amount of noise by reducing the learning rate (see previous section on Optimizers). Alternatively (red line), you can reduce the effective noise by increasing the batch size, so each batch averages over more datapoints. However, we usually reduce the learning rate rather than increasing the batch size, as we don't have the GPU memory to increase the batch size.

8 High learning rates



(a)

The previous plot would suggest that we should aggressively reduce learning rates. But that isn't true! Specifically, loss landscapes look

9 Exercises

Exercise 1.

Answer 1.