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.

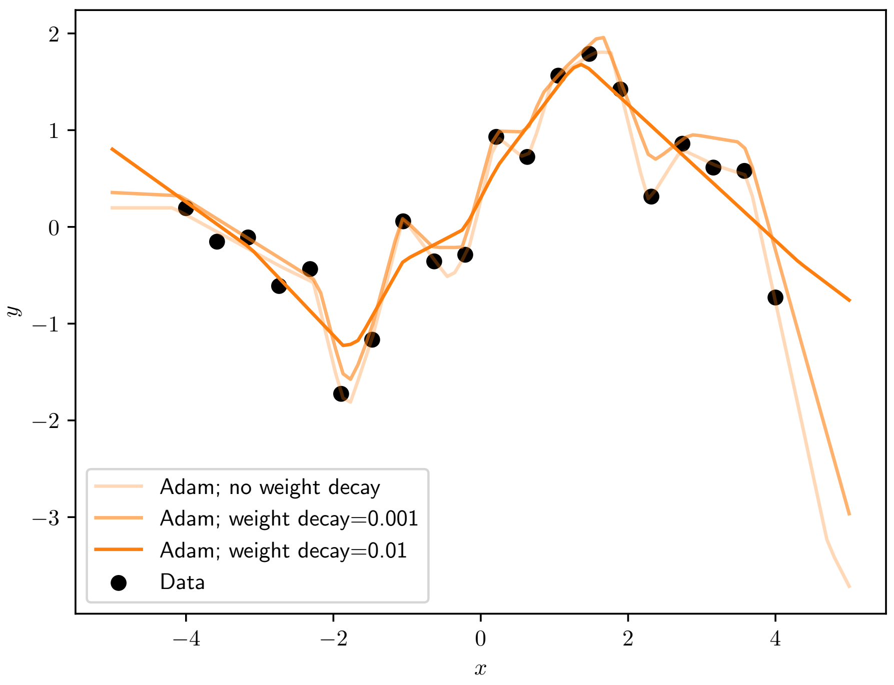
1 Overfitting is not catastrophic in NNs

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 the loss for a single minibatch.) Importantly, Eq. (1) does not yet look like weight "decay".

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

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

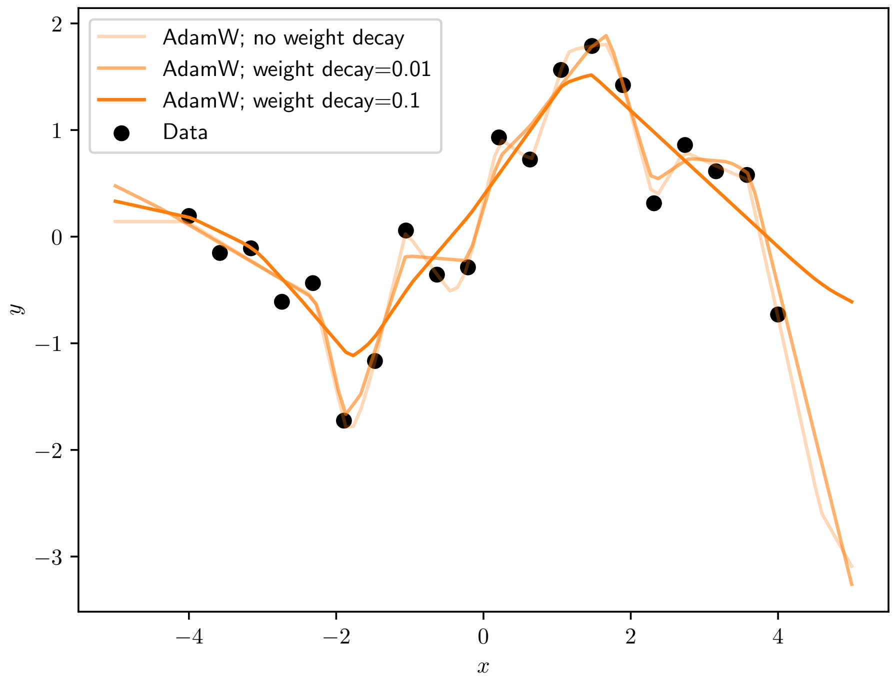
$$g_{\text{reg}} = \frac{d\mathcal{L}_{\text{reg}}}{dw} = \frac{d\mathcal{L}_{\text{unreg}}}{dw} + \lambda w \tag{3}$$

Thus, the SGD update becomes,

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

Thus, the update now contains a decay term, $-\eta\lambda w$, that pushes w towards zero.

3 "Decoupled" weight decay, or AdamW



Unfortunately, the simple connection between weight-decay and a modified loss breaks down when we consider RMSProp/Adam. Specifically, consider just using the modified loss (Eq. 1) with RMSProp, (we use RMSProp rather than Adam in our examples as it simplifies things slightly to drop the exponential moving average gradient),

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

And splitting up terms,

$$\Delta w = \underbrace{-\eta \frac{g_{\text{unreg}}}{\sqrt{\langle g_{\text{reg}}^2 \rangle}}}_{\text{original update}} \quad \underbrace{-\eta\lambda \frac{w_i}{\sqrt{\langle g_{\text{reg}}^2 \rangle}}}_{\text{Adam weight decay}} \tag{6}$$

This equation is a bit odd, as the updates don't really seem to really take on a simple "weight decay" form anymore. Instead, the weight decay is also normalized by the exponential moving average gradient-squared. Unfortunately for historical reasons this form is termed the "standard" version of weight decay, so the terms "weight decay", "standard weight decay", "Adam with weight decay" all refer to updates of this form.

That raises a question, what if we "decoupled" the weight decay from the normalizer, and just used,

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

This is "decoupled" weight decay. And Adam with decoupled weight decay is known as AdamW.

What consequences does this have?

First, the AdamW/decoupled weight decay updates cannot be written in terms of minimizing a loss. That makes these updates incredibly weird: almost all of deep learning is framed as/derived as optimizing loss functions. I'm not sure the field has really thought-through the consequences of this point.

Second, we can understand the differences between the Adam and AdamW updates by considering a "toy" setting with 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,

$$0 = g_\text{reg} = g + \lambda w \tag{8}$$

$$w = -\frac{g}{\lambda} \tag{9}$$

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,

$$\Delta w = -\eta \frac{\langle g_\text{unreg} \rangle}{\sqrt{\langle g_\text{unreg}^2 \rangle}} - \eta\lambda w_i. \tag{10}$$

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

$$\frac{\langle g_\text{unreg} \rangle}{\sqrt{\langle g_\text{unreg}^2 \rangle}} \approx \text{sign}(g_\text{unreg}), \tag{11}$$

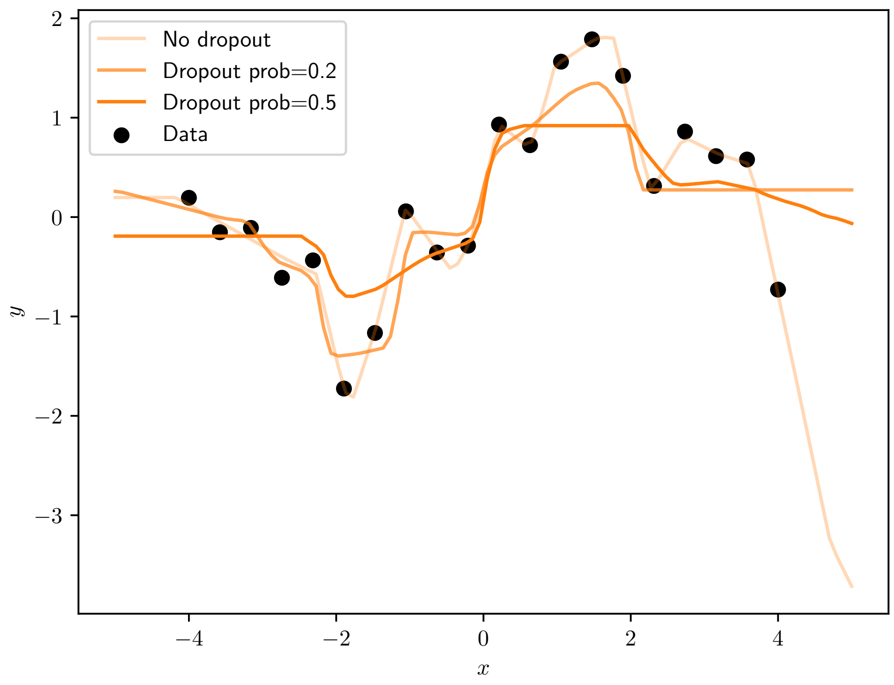
Thus, at steady-state,

$$0 \approx -\eta \text{sign}(g_\text{unreg}) - \eta\lambda w_i, \tag{12}$$

$$w_i \approx \frac{\text{sign}(g_\text{unreg})}{\lambda}. \tag{13}$$

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



Dropout is perhaps the most well-known of a class of methods that add some kind of noise to the training process. The basic idea in dropout is to "turn off" features with some probability (typically 0.5). This seems to regularise neural network training (there are some more specific ideas for what its doing, but I'm not sure I believe any of them). Specifically, dropout samples a binary mask. The probability p is the probability of dropping out a feature (i.e. the probability of the mask being 0!). For instance, if a hidden layer has 14 features, this binary mask might look like:

$$\mathbf{m} = (0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0). \tag{14}$$

The dropout operation then:

$$\text{dropout}_i^{\text{train}}(\mathbf{h}) = \tfrac{1}{1-p} m_i h_i \tag{15}$$

Alternatively, we can write this operation in vectorised form, using the Hadamard (pointwise) product,

$$\text{dropout}^{\text{train}}(\mathbf{h}) = \tfrac{1}{1-p} \mathbf{m} \odot \mathbf{h}. \tag{16}$$

Importantly, this only happens at *train* time. At test time we often want a determinstic output. And to get a deterministic output the dropout layers stop doing anything:

$$\text{dropout}^{\text{test}}(\mathbf{h}) = \mathbf{h}. \tag{17}$$

Now, we can see where the $\frac{1}{1-p}$ factor comes from. Specifically, we want the expected output of dropout at train-time to be the same as the output of dropout at test-time.

$$\mathrm{E}\left[\text{dropout}_i^{\text{train}}(\mathbf{h})\right] = \tfrac{1}{1-p}\,\mathrm{E}\left[m_i\right] h_i = \tfrac{1-p}{1-p} h_i = h_i = \text{dropout}_i^{\text{test}}(\mathbf{h}). \tag{18}$$

Remember that p is the probability of dropping out a feature (not of keeping the feature). So $\mathrm{E}\left[m_i\right] = 1 = p$,

A network with dropout might look like:

$$\begin{aligned}
\mathbf{a}_1 &= \text{linear}\left(\mathbf{x}, \mathbf{W}_1, \mathbf{b}_1\right) && \text{(19a)}\\
\mathbf{h}_1 &= \text{relu}\left(\mathbf{a}_1\right) && \text{(19b)}\\
\mathbf{d}_1 &= \text{dropout}\left(\mathbf{h}_1\right) && \text{(19c)}\\
\mathbf{a}_2 &= \text{linear}\left(\mathbf{d}_1, \mathbf{W}_2, \mathbf{b}_2\right) && \text{(19d)}\\
\mathbf{h}_2 &= \text{relu}\left(\mathbf{a}_2\right) && \text{(19e)}\\
\mathbf{d}_2 &= \text{dropout}\left(\mathbf{h}_2\right) && \text{(19f)}\\
\mathbf{f} &= \text{linear}\left(\mathbf{d}_2, \mathbf{W}_3, \mathbf{b}_3\right) && \text{(19g)}\\
\mathcal{L} &= \text{sqerr}\left(\mathbf{f}, \mathbf{y}\right). && \text{(19h)}
\end{aligned}$$

Finally, one important point to raise about dropout is that it raises the possibility of lots of irritiating bugs. Specifically, the network now behaves differently at train and test time. So you need to set the network in training mode, or in evaluation model. Using `torch.nn` modules you set the network into training mode using

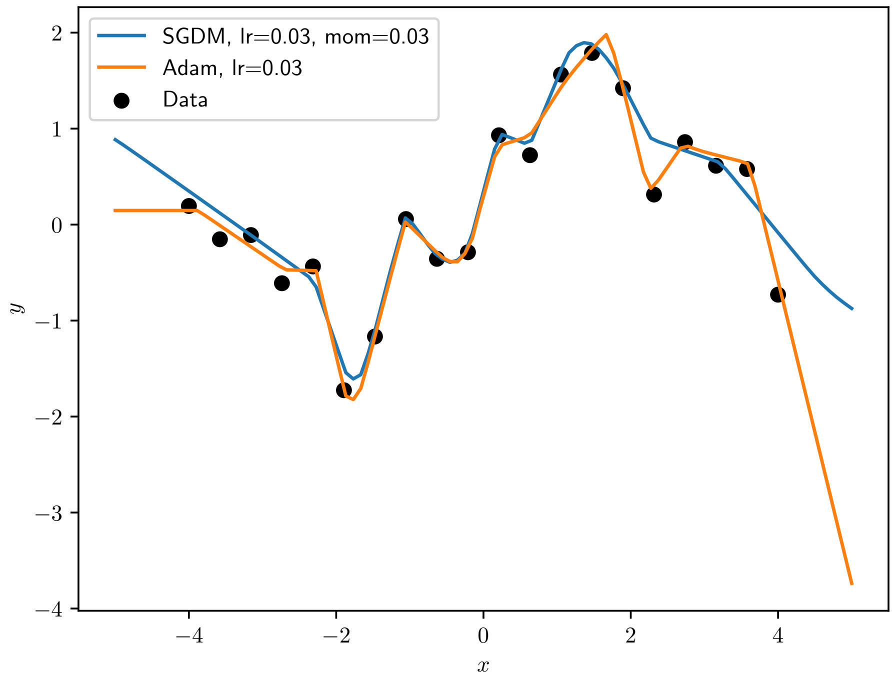
```
net.train()
```

and you set the network into eval mode using

```
net.eval()
```

Obviously, its super-easy to forget how to do this. And if you do forget, it won't be obvious (it'll still work, just not quite as well as it might have done).

5 Adam vs SGD



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

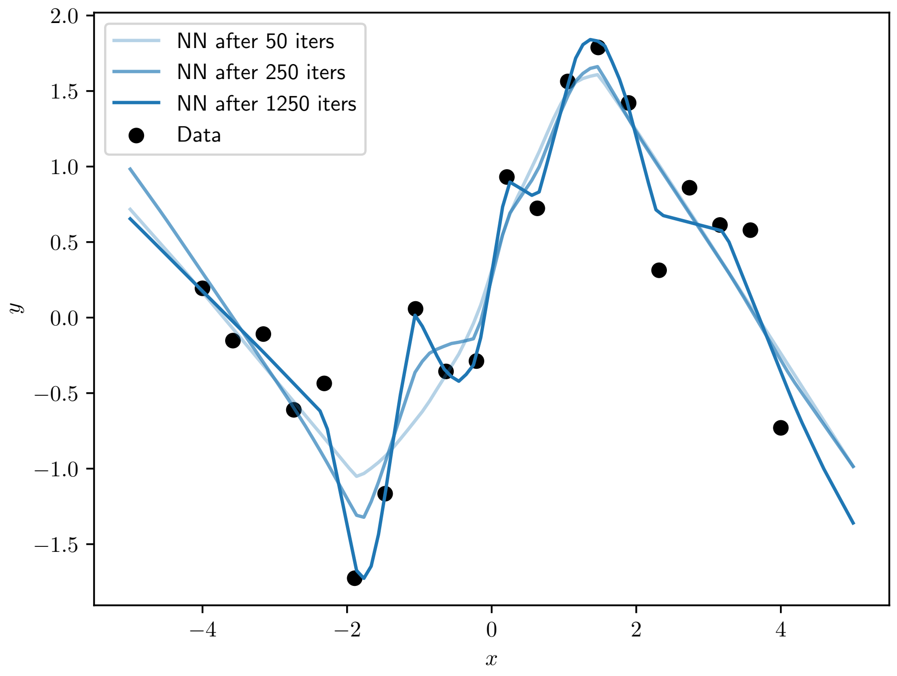
- Adam is better at capturing "fine details" than SGD.
- Adam tends to overfit more than SGD.

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) and RL where routing information in complex ways is important.

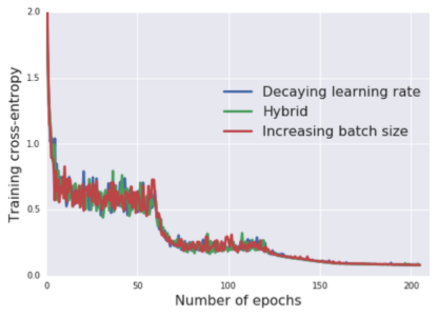
6 Early stopping



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.

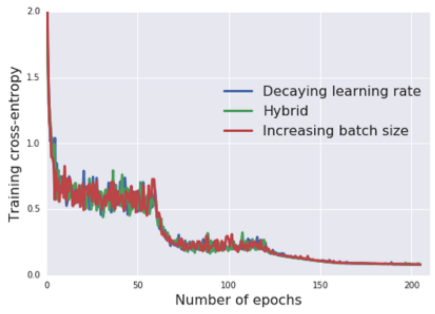
7 Minibatch noise



(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.