Tuning the Tuner

Learning an optimiser's hyperparameters by backpropagation

A report from the Torvane Laboratory for Optimisation Research

Summary

Every gradient-based training run carries settings that somebody has to choose, such as a step size. This report shows that those settings can be differentiated through the update itself and learned along with the weights.

The same trick applies to the settings of the settings, so stacks of any height can be built and trained together.

On dense, convolutional and recurrent workloads the learned optimiser matched a hand-tuned one and needed far fewer sweeps to get there.

Prepared for software testing. The text is fictional.

1 Starting point

People who train models spend a lot of time on the optimiser's own settings: a step size, a momentum term, decay rates. These are usually picked by hand or by a sweep.

The idea here is to differentiate the loss with respect to those settings, so that they can be learned in the same loop as the weights.

On all three architectures we tried — dense, convolutional and recurrent — the learned optimiser remained stable when its starting hyperparameters were badly chosen, and it became more stable as the stack of optimisers grew.

2 The method

Take an update at step t that depends on a setting such as the step size. If that update is written into the graph, the derivative of a later loss with respect to the setting comes out of the same backward pass that supplies the weights' gradients.

Keeping every step's graph would be far too expensive, so after each update the weights are cut loose from the step before. Were the update graph kept whole, backpropagation's linear cost would turn the run quadratic and make it impractical.

Nothing in the model's definition needs to change. What is needed is the update's computation graph, kept long enough for derivatives to pass through it.

2.1 Wrapping an existing optimiser

The implementation wraps an existing optimiser and keeps its interface, so switching from a fixed one to a learned one is a one-line change.
Memory grows with the height of the stack, because the update graph must be retained; the growth is linear in height and small next to the model's activations.
Learned step sizes settle into a recognisable pattern: a quick fall early on, then a plateau, much like a hand-designed decay.

Table 1: Optimisers wrapped, and the settings each exposes.

Optimiser	Learned settings	Extra memory
Plain descent	Step size	Small
Momentum	Step size, momentum	Small
Adam	Step size, decay rates	Moderate
RMSProp	Step size, decay rate	Moderate
AMSGrad	Step size, decay rates	Moderate

3 Workloads

The first workload is a one-hidden-layer dense network on digit images, the second a small convolutional network on colour images, and the third a recurrent network on character text. Splits and base optimiser are the same in every run; the only difference is whether the settings are held fixed or learned.

The experiments in this section each used one GX-7 card.

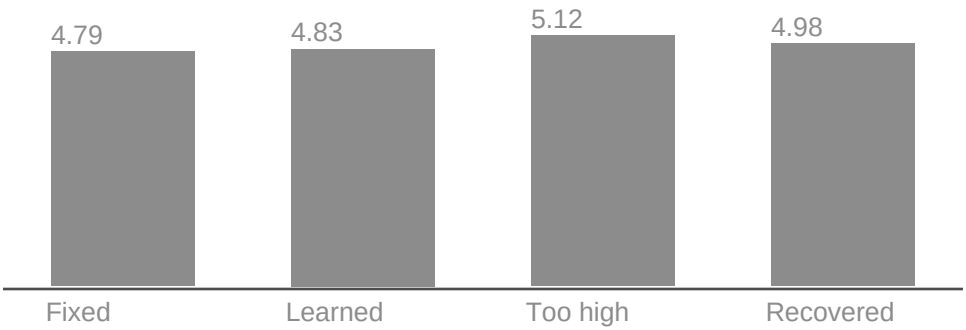
Every workload compares a fixed step size from a sweep with one learned by the method, at the same number of epochs.

Table 2: The three workloads and their base optimiser.

Workload	Data	Metric	Base optimiser
Dense	Digit images	Error rate	Adam
Convolutional	Colour images	Error rate	Adam
Recurrent	Character text	Perplexity	Adam

3.1 What the learned rate buys

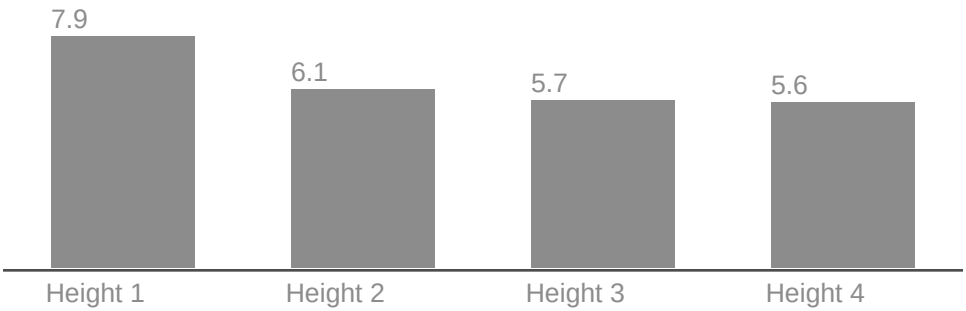
For the recurrent workload the metric is held-out perplexity, and for the other two it is test error. Against a grid search with the same budget, the learned optimiser matched or beat the best fixed setting while using fewer runs.



Held-out perplexity of the recurrent workload: a swept step size, the learned one, a deliberately bad start, and that bad start with learning.

3.3 Taller stacks

The same idea stacks: one optimiser tunes another, which tunes the model, and so on to any height.
A tall stack is awkward to set by hand, since every level carries its own step size, but it is more forgiving once the levels are started sensibly.
Each level starts below the level beneath it, so that changes at the top happen more slowly than changes at the bottom.



Held-out perplexity of the recurrent workload as the stack grows, from a deliberately poor starting rate.

3.4 Stacks on top of stacks

Section 2.1 describes how to wrap an optimiser inside another. To check that this scales, every workload was rerun with stacks of rising height and with bottom-level rates spread over several orders of magnitude.

When the starting hyper-step sizes are far too large, training blows up on anything bigger than the small dense task. Each level therefore starts below the one beneath it. Our schedule looks like this: the rates for the upper levels step from □= 105 down to 101, taking the values [□· 3, □ · 1, □· 0.3], and the bottom level is held at □= 102.

Starting step sizes spanning several orders of magnitude, in both directions from a hand-picked value, were tried to see how the stack copes.

The taller the stack, the flatter the curve of quality against the starting rate, which is the property the method was designed to produce.

4 Cost

A small library accompanies this report, with wrappers for the three workloads above and the scripts behind each figure.
Cost rises with stack height and with the size of the update graph, and each extra level adds just 1–2% to the runtime over a run with no hyperoptimiser.
Another is the assumption that the update is differentiable in its settings, which holds for the optimisers tested here but not for all adaptive methods.

Table 3: Runtime of the recurrent workload by stack height.

Height	Runtime	Over the baseline
None	1.00 h	-
1	1.01 h	1%
2	1.03 h	3%
3	1.05 h	5%

5 In practice

One drawback is a slower start: for the first few hundred steps the top-level rates are large and the estimates noisy.

The wrapper holds the update from the previous step and detaches it, so one step's graph is never joined back into the next.

A few ready-made modules simply cannot be wrapped, because they rewrite the computation graph without saying so while keeping state of their own.

For those we supply an explicit cell with the same interface and no hidden graph changes.

6 Closing remarks

The optimiser's settings can be learned by the very gradient descent that learns the model, provided the update is written into the graph and cut free between steps.

The method is cheap, stacks to any height, and makes training less sensitive to a person's first guess.

Where a sweep over learning rates is expensive, it should pay for itself; the code is released for others to try on their own models.

References

[1] A. Author, B. Author and C. Author. Learning by gradient descent to learn by gradient descent. In Proceedings of the Conference on Neural Information Processing Systems, 2016.
[2] A. Author and B. Author. Gradients through the update. A technical report, 2018.
[3] A. Author, B. Author, C. Author and D. Author. Adapting the rate online. A technical report, 2019.
[4] A. Author. A library for differentiable optimisation. A technical report, 2020.