Learning by Turning: Neural Architecture Aware Optimisation

Yang Liu*  
yang@abacus.ai  

Jeremy Bernstein*  
bernstein@caltech.edu  

Markus Meister  
meister@caltech.edu  

Yisong Yue  
yyue@caltech.edu  

Abstract

Descent methods for deep networks are notoriously capricious: they require careful tuning of step size, momentum and weight decay, and which method will work best on a new benchmark is a priori unclear. To address this problem, this paper conducts a combined study of neural architecture and optimisation, leading to a new optimiser called \textit{Nero}: the neuronal rotator. Nero trains reliably without momentum or weight decay, works in situations where Adam and SGD fail, and requires little to no learning rate tuning. Also, Nero’s memory footprint is \(\sim\) square root that of Adam or LAMB. Nero combines two ideas: (1) projected gradient descent over the space of balanced networks; (2) neuron-specific updates, where the step size sets the angle through which each neuron’s hyperplane turns. The paper concludes by discussing how this geometric connection between architecture and optimisation may impact theories of generalisation in deep learning.

1 Introduction

Deep learning has brought on a new paradigm in computer science, enabling artificial systems to interact with the world at an unprecedented level of complexity. That said, the core technology relies on various heuristic numerical techniques that are sometimes brittle and often require extensive tuning. A major goal of modern research in machine learning is to uncover the principles underlying learning in neural systems, and thus to derive more reliable learning algorithms.

Part of the challenge of this endeavour is that learning in deep networks is an inherently coupled problem. Suppose that training performance is sensitive to a particular detail of the neural architecture—then it is unclear whether that detail affects the expressivity of the architecture, or just the ability of the descent method to train the architecture.

This observation motivates the \textit{combined study} of architecture and optimisation, and this paper explores several questions at that intersection. First of all:

\(\ast\) What is the right domain of optimisation for a neural network’s weights? Is it \(\mathbb{R}^d\), or something more exotic—such as a Cartesian product of hyperspheres?

Typically, optimisation is conducted over \(\mathbb{R}^d\), while a careful weight initialisation and a tuned weight decay hyperparameter impose a soft constraint on the optimisation domain. Since normalisation schemes such as batch norm \cite{joffe2015batch} render the network invariant to the scale of the weights, weight decay also plays a somewhat subtle second role in modifying the effective learning rate. Hyperparameters with this kind of subtle coupling add to the compounding cost of hyperparameter search.

Furthermore, descent methods such as Adam \cite{kingma2015adam} and LAMB \cite{you2020large} use either synapse-specific or layer-specific gradient normalisation. This motivates a second question:

\(\ast\) At what level of granularity should an optimiser work? Should normalisation occur per-synapse or per-layer—or perhaps, per-neuron?

This paper contends that in deep learning, hyperparameters proliferate because of hidden couplings between optimiser and architecture. By studying the above questions, and distilling the simple rules that govern optimisation and architecture, this paper aims to make deep learning less brittle—and less sensitive to opaque hyperparameters.

Summary of contributions:

1. A new optimiser—\textit{Nero}: the neuronal rotator. Nero performs per-neuron projected gradient descent, and uses \(\sim\) square root the memory of Adam or LAMB.

2. Experiments across image classification, image generation, natural language processing and reinforcement learning, in which Nero’s \textit{out-of-the-box} configuration tends to outperform \textit{tuned} baseline optimisers.

3. Discussion of how the connection between optimisation and architecture relates to generalisation theories, such as PAC-Bayes and norm-based complexity.

*Equal contribution.
2 Related work

This section reviews relevant work pertaining to both neural architecture design and optimisation in machine learning, and concludes with a bridge to the neuroscience literature.

2.1 Neural Architecture Design

The importance of wiring constraints for the stable function of engineered neural systems is not a new discovery. One important concept is that of balanced excitation and inhibition. For instance, Rosenblatt (1958) found that balancing the proportion of excitatory and inhibitory synaptic connections made his perceptron more robust to varying input sizes. Another concept relates to the total magnitude of synapse strengths. For example, Rochester et al. (1956) constrained the sum of magnitudes of synapses impinging on a neuron so as to stabilise the process of learning. Similar ideas were explored by von der Malsburg (1973) and Miller & MacKay (1994). These works are early predecessors to this paper’s definition of balanced networks given in Section 3.1.

Given the resurgence of neural networks over the last decade, the machine learning community has taken up the mantle of research on neural architecture design. Special weight scalings—such as Xavier init (Glorot & Bengio, 2010) and Kaiming init (He et al., 2015)—have been proposed to stabilise signal transmission through deep networks. These scalings are only imposed at initialisation and are free to wander during training—an issue which may be addressed by tuning a weight decay hyperparameter. More recent approaches—such as batch norm (Ioffe & Szegedy, 2015)—explicitly control activation statistics throughout training by adding extra normalisation layers to the network.

Other recent normalisation techniques lie closer to the work of Rosenblatt (1958) and Rochester et al. (1956). Techniques that involve constraining a neuron’s weights to the unit hypersphere include: weight norm (Salimans & Kingma, 2016), decoupled networks (Liu et al., 2017, 2018) and orthogonal parameterised training (Liu et al., 2020). Techniques that also balance excitation and inhibition include centred weight norm (Huang et al., 2017) and weight standardisation (Qiao et al., 2019).

2.2 Descent Methods in Deep Learning

Much classic work in optimisation theory focuses on deriving convergence results for descent methods under assumptions such as convexity (Boyd & Vandenberghe, 2004) and Lipschitz continuity of the gradient (Nesterov, 2004). These simplifying assumptions are often used in the machine learning literature. For instance, Bottou et al. (2018) provide convergence guarantees for stochastic gradient descent (SGD) under each of these assumptions. However, these assumptions do not hold in deep learning (Sun, 2019).

On a related note, SGD is not the algorithm of choice in many deep learning applications, and heuristic methods such as RMSprop (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2015) often work better. For instance, Adam often works much better than SGD for training generative adversarial networks (Bernstein et al., 2020a). Yet the theory behind Adam is poorly understood (Reddi et al., 2018).

A more recent line of work has explored optimisation methods that make relative updates to neural network parameters. Optimisers like LARS (You et al., 2017), LAMB (You et al., 2020) and Fromage (Bernstein et al., 2020a) make per-layer relative updates, while Madam (Bernstein et al., 2020b) makes per-synapse relative updates. You et al. (2017) found that these methods stabilise large batch training, while Bernstein et al. (2020a) found that they require little to no learning rate tuning across tasks.

Though these recent methods partially account for the neural architecture—by paying attention to its layered operator structure—they do not rigorously address the optimisation domain. As such, LARS and LAMB require a tunable weight decay hyperparameter, while Fromage and Madam restrict the optimisation to a bounded set of tunable size (i.e. weight clipping). Without this additional tuning, these methods can be unstable—see for instance (Bernstein et al., 2020a, Figure 2) and (Bernstein et al., 2020b, Figure 3).

The discussion in the previous paragraph typifies the machine learning state of the art: optimisation techniques that work well, albeit only after hyperparameter tuning. For instance, LAMB is arguably the state-of-the-art relative optimiser, but it contains in total five tunable hyperparameters. Since—at least naïvely—the cost of hyperparameter search is exponential in the number of hyperparameters, the prospect of fully tuning LAMB is computationally daunting.

2.3 Homeostatic Control in Neuroscience

Since the brain is a system that must learn stably without hyperparameter do-overs, it is worth looking to neuroscience for inspiration on designing better learning algorithms.

A major swathe of neuroscience research studies mechanisms by which the brain performs homeostatic control. For instance, neuroscientists report a form of homeostasis termed synaptic scaling, where a neuron modulates the strengths of all its synapses to stabilise its firing rate (Turrigiano, 2008). More generally, heterosynaptic plasticity refers to homeostatic mechanisms that modulate the strength of unstimulated synapses (Chistiakova et al., 2015). Shen et al. (2020) review connections to normalisation methods used in machine learning.

These observations inspired this paper to consider implementing homeostatic control via projected gradient descent—leading to the Nero optimiser.
3 Background Theory

In general, an $L$-layer neural network $f(\cdot)$ is a composition of $L$ simpler functions $f_{1}(\cdot),...,f_{L}(\cdot)$:

$$f(x) = f_{L} \circ f_{L-1} \circ ... \circ f_{1}(x).$$  \hfill (forward pass)

Due to this compositionality, any slight ill-conditioning in the simple functions $f_{i}(\cdot)$ has the potential to compound over layers, making the overall network $f(\cdot)$ very ill-conditioned. Architecture design should aim to prevent this from happening, as will be covered in Section 3.1.

The Jacobian $\partial f / \partial f_{i}$, which plays a key role in evaluating gradients, also takes the form of a deep product:

$$\frac{\partial f}{\partial f_{i}} = \frac{\partial f_{L}}{\partial f_{L-1}} \cdot \frac{\partial f_{L-1}}{\partial f_{L-2}} \cdot ... \cdot \frac{\partial f_{1}}{\partial f_{1}}.$$  \hfill (backward pass)

Therefore, it is also important from the perspective of gradient-based optimisation that compositionality is adequately addressed, as will be covered in Section 3.2.

3.1 Balanced Network Architectures

A common strategy to mitigate the issue of compounding ill-conditioning is to explicitly re-normalise the activations at every network layer. Batch norm exemplifies this strategy, and was found to improve the trainability of deep residual networks. Batch norm works by standardising the activations across a batch of inputs at each network layer—that is, it shifts and scales the activations to have mean zero and variance one across a batch.

Although batch norm works well, it adds computational overhead to both the forward and backward pass. To explore how far one can get without explicit re-normalisation, the following definitions are useful:

Definition 1. A neuron is balanced if its weight vector $w \in \mathbb{R}^{d}$ satisfies the following constraints:

$$\sum_{i=1}^{d} w_{i} = 0; \quad (balanced \ excitation \ & \ inhibition)$$

$$\sum_{i=1}^{d} w_{i}^{2} = 1. \quad (\ell_{2} \ constant \ sum \ rule)$$

Definition 2. A network is balanced if all its constituent neurons are balanced.

As noted by Huang et al. (2017), balanced neurons attain some of the properties of batch norm for free. To see this, consider a linear neuron $y = \sum w_{i}x_{i}$ with inputs $x_{i}$ that are uncorrelated with mean $\mu$ and variance 1. Then the output $y$ is standardised:

$$E[y] = \sum w_{i} E[x_{i}] = \mu \sum w_{i} = 0;$$

$$\text{Var}[y] = \sum w_{i}^{2} \text{Var}[x_{i}] = \sum w_{i}^{2} = 1.$$  \hfill (balanced neuron)

While the assumptions on the inputs $x_{i}$ are unlikely to hold exactly, under more general conditions the constraints may at least encourage the standardisation of activation statistics through the layers of the network [Brock et al., 2021].

3.2 Stable Descent Steps

Since a network is trained via perturbations to its parameters, it is important to know what size perturbations are appropriate. Consider an $L$-layer network with weight matrices $W = (W_{1}, W_{2},...,W_{L})$ and loss function $L(W)$. For a perturbation $\Delta W = (\Delta W_{1}, \Delta W_{2},...,\Delta W_{L})$, the following definition establishes a notion of stable step size:

Definition 3. Let $\theta$ denote the angle between $\Delta W_{l}$ and $-\nabla_{W_{l}} L(W)$. A descent step is stable if for all $l = 1,...,L$:

$$\frac{|\nabla_{W_{l}} L(W + \Delta W) - \nabla_{W_{l}} L(W)|_{F}}{||\nabla_{W_{l}} L(W)||_{F}} < \cos \theta.$$  \hfill (1)

Or in words: for each layer, the relative change in gradient induced by the perturbation should not exceed the cosine of the angle between the perturbation and the negative gradient.

This definition is useful because Bernstein et al. (2020a) proved that a stable descent step is guaranteed to decrease a continuously differentiable loss function $L(W)$. Since Inequality (1) is of little use without a model of its lefthand side, Bernstein et al. (2020a) proposed the following model:

Definition 4. The loss function obeys deep relative trust if for all perturbations $\Delta W = (\Delta W_{1}, \Delta W_{2},...,\Delta W_{L})$:

$$\frac{|\nabla_{W_{l}} L(W + \Delta W) - \nabla_{W_{l}} L(W)|_{F}}{||\nabla_{W_{l}} L(W)||_{F}} \leq \prod_{k=1}^{l} \left(1 + \frac{|\Delta W_{k}|_{F}}{||W_{k}||_{F}}\right) - 1.$$  \hfill (2)

While deep relative trust is based on a perturbation analysis of $L$-layer perceptrons (Bernstein et al. 2020a Theorem 1), the key idea is that its product structure explicitly models the product structure of the network’s backward pass.

The deep relative trust model suggests that a stable descent step should involve small relative perturbations per layer. This motivates the layer-wise family of descent methods (You et al., 2017, 2020). Still, it is unclear whether layers are the right base object to consider. Perhaps a more refined analysis would replace the layers appearing in Definition 3 with individual neurons or even synapses.

Small relative perturbations per-synapse were explored by Bernstein et al. (2020b) and found to slightly degrade training performance compared to Adam and SGD. But this paper will explore the per-neuron middle ground:

Definition 5. A step of size $\eta > 0$ is said to be per-neuron relative if for any neuron with weights $w \in \mathbb{R}^{d}$ and bias $b \in \mathbb{R}$, the perturbations $\Delta w \in \mathbb{R}^{d}$ and $\Delta b \in \mathbb{R}$ satisfy:

$$||\Delta w||_{2}/||w||_{2} \leq \eta \quad \text{and} \quad ||\Delta b||/|b| \leq \eta.$$  \hfill (3)

A per-neuron relative update is automatically per-layer relative. To see this, consider a weight matrix $W$ whose $N$ rows correspond to $N$ neurons $w^{(1)},...,w^{(N)}$. Then:

$$\frac{|\Delta W|_{F}}{||W||_{F}} = \frac{\sqrt{\sum_{i=1}^{N} |\Delta w^{(i)}|_{2}^{2}}}{\sum_{i=1}^{N} |w^{(i)}|_{2}} \leq \sqrt{\frac{\sum_{i=1}^{N} \eta^{2}|w^{(i)}|_{2}^{2}}{\sum_{i=1}^{N} |w^{(i)}|_{2}^{2}}} = \eta.$$  \hfill (4)
4 Nero: the Neuronal Rotator

Following the discussion in Section 3, this paper will consider an optimisation algorithm that makes per-neuron relative updates (Definition 5) constrained to the space of balanced networks (Definition 2).

Since a balanced neuron is constrained to the unit hypersphere, a per-neuron relative update with step size \( \eta \) corresponds to a pure rotation of the neuron’s weight vector by angle \( \approx \eta \). To see this, take \( \eta \) small in the following picture:

\[
\| w + \Delta w \|_2 = 1 \quad \| \Delta w \|_2 = \eta \\
\| w \|_2 = 1
\]

Hence, this paper proposes Nero: the neuronal rotator. Nero’s goal is to reduce the burden of hyperparameter tuning by baking architectural information into the optimiser. More concretely, the anticipated advantages are as follows:

1. Since per-neuron relative updates are automatically per-layer relative by Equation 2, they should inherit the properties of per-layer updates—in particular, stability across batch sizes (You et al., 2017) while needing little to no learning rate tuning (Bernstein et al., 2020a).

2. Since balanced networks place hard constraints on the norm of a neuron’s weights, the need for initialisation tuning and weight decay should be removed.

3. Gradients are often normalised by running averages, in order to retain relative scale information between successive minibatch gradients (Tieleman & Hinton, 2012). Along with momentum, this is the main memory overhead of Adam and LAMB compared to vanilla SGD. Per-neuron running averages consume \( \sim \) square root the memory of per-synapse running averages.

4. Since normalisation is local to a neuron, no communication is needed between neurons in a layer (unlike for per-layer updates). This makes the optimiser more distributable—for example, a single layer can be split across multiple compute devices without fuss. For the same reason, the Nero update is biologically plausible.

There is one significant difference between Nero and prior work on balanced networks. In centred weight norm (Huang et al., 2017) and weight standardisation (Qiao et al., 2019), a neuron’s underlying weight representation is an unnormalised vector \( \tilde{w} \in \mathbb{R}^d \)—which is normalised by including the following reparameterisation in the neural architecture:

\[
\text{normalise}(\tilde{w}) := \frac{\tilde{w} - 1^T \tilde{w} \cdot 1/d}{\| \tilde{w} - 1^T \tilde{w} \cdot 1/d \|_2}.
\]

where \( 1 \) denotes the vector of 1s.

Algorithm 1 Nero optimiser. “Out-of-the-box” hyperparameter defaults are \( \eta = 0.01 \) and \( \beta = 0.999 \). The constant \( \sigma_b \in \mathbb{R}_+ \) refers to the initialisation scale of the biases.

**Input:** step size \( \eta \in (0, 1] \), averaging constant \( \beta \in [0, 1) \)

repeat

for each neuron do

\( \triangleright \) get weight & bias gradients \( g_w \in \mathbb{R}^n \) & \( g_b \in \mathbb{R}^m \)

\( \triangleright \) update running averages

\[
\bar{g}_w = \beta \cdot \bar{g}_w + (1 - \beta) \cdot \| g_w \|_2^2
\]

\[
\bar{g}_b = \beta \cdot \bar{g}_b + (1 - \beta) \cdot g_b^2
\]

\( \triangleright \) update weights \( w \in \mathbb{R}^n \) and bias \( b \in \mathbb{R}^m \)

\[
w \leftarrow w - \eta \cdot \bar{g}_w / \bar{g}_w \cdot g_w
\]

\[
b \leftarrow b - \eta \cdot \sigma_b / \bar{g}_b 
\]

\( \triangleright \) project weights back to constraint set

\[
w \leftarrow w - \frac{1}{\eta} \sum_{i=1}^n w_i
\]

end for

until converged

Since the target of automatic differentiation is still the unnormalised vector \( \tilde{w} \), overhead is incurred in both the forward and backward pass. Moreover, there is a subtle coupling between the step size in additive optimisers like Adam and the scale of the unnormalised weights \( \tilde{w} \) (see Section 5.3).

In contrast, Nero opts to implement balanced networks via projected gradient descent. This is lighter-weight than Equation 3 since duplicate copies of the weights are not needed and the network’s backward pass does not involve extra operations. Furthermore, Nero can be used as a drop-in replacement for optimisers like Adam, SGD or LAMB, without the user needing to manually modify the network architecture via the reparameterisation in Equation 3.

Pseudocode for Nero is provided in Algorithm 1. For brevity, the Adam-style bias correction of the running averages is omitted from the pseudocode. But in the Pytorch implementation used in this paper’s experiments, the running averages \( \bar{g}_w \) and \( \bar{g}_b \) are divided by a factor of \( \sqrt{1 - \beta^r} \) before the \( r \)th update. This corrects for the warmup bias stemming from \( \bar{g}_w \) and \( \bar{g}_b \) being initialised to zero (Kingma & Ba, 2015).

While the pseudocode in Algorithm 1 is presented for neurons and biases, in the Pytorch implementation the bias update is applied to any parameters lacking a notion of fan-in—including batch norm gains and biases. Typical initialisation scales are \( \sigma_b = 1 \) for gains and \( \sigma_b = 0.01 \) for biases. The Pytorch implementation of Nero used \( \sigma_b = 0.01 \) for any bias parameter initialised to zero.
5 Experiments

This section begins with targeted experiments intended to demonstrate Nero’s key properties. Then, in Section 5.6, Nero is benchmarked across a range of popular tasks. In all figures, the mean and range are plotted over three repeats. For Nero, out-of-the-box refers to setting $\eta = 0.01$ and $\beta = 0.999$. More experimental details are given in Appendix A.

5.1 Constraints Help Nero

To verify that projecting to the space of balanced networks improves the performance of Nero, an ablation experiment was conducted. As can be seen in Figure 1 when training a VGG-11 image classifier on the CIFAR-10 dataset, Nero performed best with both constraints switched on.

5.2 Per-Neuron Updates are a Good Middle Ground

Since Bernstein et al. (2020b) found that per-synapse relative updates led to slightly degraded performance, while per-layer relative updates typically perform well (You et al., 2017, 2020; Bernstein et al., 2020a), this section compares per-synapse, per-neuron and per-layer relative updates. In particular, Nero is compared to Madam (per-synapse relative) and LAMB (per-layer relative).

A VGG-11 model was trained on the CIFAR-10 dataset. Without constraints, the three optimisers performed similarly, achieving $\sim 12\%$ top-1 validation error (Figure 2 top). Constraining to the space of balanced networks (Definition 2) improved both Nero and LAMB, but did not have a significant effect on Madam (Figure 2 bottom). In both configurations, Nero outperformed Madam and LAMB, demonstrating the viability of per-neuron relative updates.

5.3 The Pitfalls of Reparameterisation

Existing implementations of balanced networks (Definition 2) work via the re-parameterisation given in Equation 3 (Huang et al., 2017, Qiao et al., 2019). This leads to an undesired coupling between the learning rate in optimisers like Adam and the scale of the unnormalised $\bar{w}$ parameters.

To verify this, a network with weights normalised by Equation 3 was trained to classify the MNIST dataset. The initial weights $\bar{w}$ were drawn from $\mathcal{N}(0, \sigma^2)$, and the experiment was repeated for $\sigma = 1$ and $\sigma = 100$. The Adam optimiser was used for training with a fixed learning rate of 0.01. As can be seen in Figure 3 (left), the training performance was sensitive to the weight scale $\sigma$, despite the fact that a weight normalisation scheme was being used.

The unnecessary scale freedom of reparameterisation can lead to other undesired consequences, such as numerical overflow. Nero completely eliminates this issue by implementing balanced networks via projected gradient descent.

Figure 1. Ablating the balanced network constraints. A VGG-11 network was trained on CIFAR-10. The legend denotes which of Nero’s constraints were active. Mean refers to balanced excitation & inhibition, while norm refers to the $\ell_2$ constant sum rule.

Figure 2. Comparing per-synapse (Madam), per-neuron (Nero) and per-layer (LAMB) relative updates. A VGG-11 network was trained to classify CIFAR-10. Top: all optimisers without balanced network constraints. Bottom: all optimisers with constraints.

Figure 3. Left: Training a 5 layer perceptron normalised via reparameterisation (Equation 3) on MNIST. For a fixed Adam learning rate, training is sensitive to the scale $\sigma$ of the raw weights $\bar{w}$. This motivates the different approach taken by Nero. Right: Using Nero to train a 100 layer perceptron—without batch norm or skip connections—to classify MNIST.
5.4 Nero Trains Deeper Networks

Very deep networks are typically difficult to train without architectural modifications such as residual connections [He et al. 2016] or batch norm [Ioffe & Szegedy 2015]. To test whether Nero enables training very deep models without such modifications, Figure 3 (right) shows the results of training a very deep multilayer perceptron (MLP) on the MNIST dataset. Unlike SGD, Adam and LAMB, Nero could reliably train a 100-layer MLP.

5.5 Highly Tuned SGD can Outperform Nero

This section compares Nero out-of-the-box to an SGD implementation with tuned learning rate, weight decay and momentum. The comparison was made for training a ResNet-50 image classifier on the ImageNet dataset. As can be seen in Figure 4, SGD with tuned learning rate, momentum, and weight decay outperformed Nero. However, the optimal set of SGD hyperparameters was brittle, and ablating weight decay alone increased the top-1 validation error by 5%.

5.6 Nero Works Well Out-of-the-Box

This section probes the versatility and robustness of Nero by comparing its optimisation and generalisation performance with three popular alternatives—SGD, Adam, and LAMB—across six tasks. The tasks span the domains of computer vision, natural language processing, and reinforcement learning. A wide spectrum of neural architectures were tested—from convolutional networks to transformers.

To make a fair comparison between optimisers, a fair hyperparameter tuning strategy is needed. In this section:

1. Learning rates were tuned over \{10^{-4}, 10^{-3}, ..., 10^0\}.

2. For Adam, LAMB and SGD, the momentum hyperparameter was tuned to achieve good performance on the most complicated benchmark—cGAN training—and then fixed across the rest of the benchmarks. In each case, the best momentum value for cGAN was 0.

3. \(\beta\) in Nero and \(\beta_2\) in Adam and LAMB were fixed to 0.999 across all experiments, as recommended by Kingma & Ba (2015) and You et al. (2020).

4. Weight decay was not used in any of the experiments.

The results are collated in Table 1. Nero achieved the best validation performance in every experiment—while the runner-up varied across tasks. What’s more, the same learning rate of \(\eta = 0.01\) was optimal for Nero in five out of six experiments. This means that Nero has strong out-of-the-box performance, since Nero’s only other hyperparameter was fixed to \(\beta = 0.999\) across all experiments.

The remainder of this section discusses each experiment in turn. Implementation details are given in Appendix A.
Image synthesis with cGAN  Generative Adversarial Network (Goodfellow et al. 2014, GAN) training is perhaps the most challenging optimisation problem tackled in this paper. Good performance has traditionally relied on extensive tuning: different learning rates are often used in the generator and discriminator (Heusel et al. 2017) and training is highly sensitive to momentum (Brock et al. 2019, p. 35). The class-conditional GAN model in this paper is based on the BigGAN architecture (Brock et al. 2019). This is a heterogeneous network involving a variety of building blocks: convolutions, embeddings, fully connected layers, attention layers, conditional batch norm and spectral norm (Miyato et al. 2018). The results are presented in Figure 7.

Image classification In Section 5.5, Nero out-of-the-box was shown to outperform SGD without weight decay when training ResNet-50 on ImageNet. Due to limited computational resources, the authors of this paper were unable to run the LAMB and Adam baselines on ImageNet. Experiments were run across all baselines on the smaller CIFAR-10 dataset instead. The networks used were the vanilla, convolutional VGG-11 network (Simonyan & Zisserman 2015) and the batch-normalised, residual ResNet-18 network (He et al. 2015). The results are presented in Figure 8.

Natural language processing Much recent progress in natural language processing is based on the transformer architecture (Vaswani et al. 2017). Transformers process information via layered, all-to-all comparisons—without recourse to recurrence or convolution. This paper experimented with a smaller transformer (19 tensors) trained on the Wikitext-2 dataset, and a larger transformer (121 tensors) trained on WMT2016 English–German translation. The results are presented in Figures 7 and 8.

Reinforcement learning Many reinforcement learning algorithms use neural networks to perform function approximation. Proximal Policy Optimization (Schulman et al. 2017, PPO) is one example, and PPO has gained increasing popularity for its simplicity, scalability, and robust performance. This paper experimented with PPO on the Atari Pong video game. The results are presented in Figure 9.

While LAMB failed to train on this task, further investigation revealed that setting LAMB’s momentum hyperparameter to 0.9 enabled LAMB to learn. This demonstrates LAMB’s sensitivity to hyperparameters.

Table 1. Validation results for the best learning rate $\eta$. The best result is shown in bold, while the runner-up is underlined.

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Model</th>
<th>Metric [%]</th>
<th>Nero</th>
<th>SGD</th>
<th>Adam</th>
<th>LAMB</th>
<th>Nero $\eta$</th>
<th>SGD $\eta$</th>
<th>Adam $\eta$</th>
<th>LAMB $\eta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image synthesis with cGAN</td>
<td>CIFAR-10</td>
<td>BigGAN-like</td>
<td>FID (%)</td>
<td>17.43 ± 0.37</td>
<td>35.07 ± 0.42</td>
<td>25.42 ± 0.85</td>
<td>16.32 ± 0.04</td>
<td>0.11</td>
<td>0.10</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>Classification</td>
<td>CIFAR-10</td>
<td>VGG11</td>
<td>Top-1 Error (%)</td>
<td>11.16% ± 0.17</td>
<td>12.61% ± 0.21</td>
<td>12.86% ± 0.34</td>
<td>13.66% ± 0.55</td>
<td>0.01</td>
<td>0.10</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>Language Model</td>
<td>Wikitext-2</td>
<td>Transformer</td>
<td>Perplexity (%)</td>
<td>172.99 ± 0.51</td>
<td>181.76 ± 0.49</td>
<td>178.35 ± 0.96</td>
<td>200.54 ± 1.53</td>
<td>0.01</td>
<td>1.00</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>Translation</td>
<td>WMT16 En–De</td>
<td>Transformer</td>
<td>Perplexity (%)</td>
<td>11.35 ± 1.20</td>
<td>92.40 ± 89.48</td>
<td>12.63 ± 0.34</td>
<td>16.30 ± 0.09</td>
<td>0.01</td>
<td>0.001</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
<tr>
<td>PPO</td>
<td>Atari Pong</td>
<td>vanilla CNN</td>
<td>Reward (%)</td>
<td>20.62 ± 0.05</td>
<td>11.99 ± 0.85</td>
<td>15.32 ± 3.42</td>
<td>−19.46 ± 0.10</td>
<td>0.01</td>
<td>0.10</td>
<td>0.0001</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

Figure 7. Training a language model on the Wikitext-2 dataset. A small transformer network was used, composed of 19 tensors. Nero achieved the best anytime performance.

Figure 8. Training an English–German translation model on WMT16. A larger transformer network was used, composed of 121 tensors. The optimisers with gradient normalisation—Nero, Adam, and LAMB—performed best in training this model. Training with SGD was unstable and led to significantly worse perplexity.

Figure 9. Training a policy network to play Pong. Proximal Policy Optimisation (PPO) was used. Pong’s reward is bounded between ±21. While investigating LAMB’s failure to train the policy network, it was discovered that adjusting the $\beta_1$ momentum hyperparameter from 0 to 0.9 improved LAMB’s performance.
6 Discussion: Rotation and Generalisation

The results in this paper may have a bearing on the generalisation theory of neural systems—an area of research that is still not settled. Consider the following hypothesis:

Hypothesis 1. Deep learning generalises because SGD is biased towards solutions with small norm.

This hypothesis is well-known, and is alluded to or mentioned explicitly in many papers (Wilson et al., 2017; Zhang et al., 2017; Bansal et al., 2018; Advani et al., 2020). But in light of the results in Table 1, Hypothesis 1 encounters some basic problems. First, for some tasks—such as the GAN and translation experiment—SGD simply performs very poorly. And second, Nero is able to find generalising solutions even when the norm of the network is constrained. For instance, the VGG-11 network and the Wikitext-2 transformer model have no gain parameters so, under Nero, the norm of the weights (though not the biases) is fixed and cannot be “adapting to the data complexity”.

Then it seems right to consider an alternative theory:

Hypothesis 2. Deep learning generalises because the space of networks that fit the training data has large measure.

This hypothesis is essentially the PAC-Bayesian generalisation theory (McAllester, 1998; Langford & Seeger, 2001) applied to deep learning. Valle-Perez et al. (2019) have developed this line of work, proving the following result:

Theorem 1 (Realisable PAC-Bayes). First, fix a probability measure \( \mathbb{P} \) over the weight space \( \Omega \) of a classifier. Let \( S \) denote a training set of \( n \) iid datapoints and let \( V_S \subset \Omega \) denote the version space—that is, the subset of classifiers that fit the training data. Consider the population error rate \( 0 \leq \varepsilon(w) \leq 1 \) of weight setting \( w \in \Omega \), and its average over the version space \( \varepsilon(V_S) := \mathbb{E}_{w \in V_S} [\varepsilon(w) | w \in V_S] \). Then, for a proportion \( 1 - \delta \) of random draws of the training set \( S \),

\[
\varepsilon(V_S) \leq \ln \frac{1}{1 - \varepsilon(V_S)} \leq \frac{\ln \frac{1}{\mathbb{P}[V_S]} + \ln \frac{2n}{n - 1}}{n - 1}.
\]

The intuition is that for a larger measure of solutions \( \mathbb{P}[V_S] \), less information needs to be extracted from the training data to find just one solution, thus memorisation is less likely.

A simple formula for \( \mathbb{P}[V_S] \) is possible based on this paper’s connection between optimisation and architecture, since the problem is reduced to hyperspherical geometry. Consider a balanced network (Definition 2) composed of \( m \) neurons each with fan-in \( d \). Then the optimisation domain is isomorphic to the Cartesian product of \( m \) hyperspheres:

\[
\Omega \cong S^{d-2} \times \ldots \times S^{d-2},
\]

while \( \mathbb{P} \) can be fixed to the uniform distribution on \( \Omega \).

By restricting deep relative trust (Definition 4) or its antecedent (Bernstein et al., 2020a, Theorem 1) to balanced networks, the following definition becomes natural:

Definition 6. A solution that attains zero training error is \( \alpha \)-robust if all neurons may be simultaneously and arbitrarily rotated by up to angle \( \alpha \) without inducing an error.

Geometrically, an \( \alpha \)-robust solution is the product of \( m \) hyperspherical caps. If the version space consists of \( K \) non-intersecting \( \alpha \)-robust solutions, then its measure is:

\[
\mathbb{P}[V_S] = K \cdot \mathbb{P}[\text{cap}_{d-2}(\alpha)]^m \geq \frac{K}{2^m} \sin^{m(d-2)} \frac{\alpha}{2},
\]

where \( \text{cap}_{d-2}(\alpha) \) denotes an \( \alpha \)-cap of \( S^{d-2} \), and the inequality follows from (Ball, 1997, Lemma 2.3). Combining Inequality 4 with Inequality 4 yields the following generalisation bound for neural networks:

\[
\varepsilon(V_S) \leq \frac{m \ln 2 + m(d - 2) \ln \frac{1}{\sin \frac{\alpha}{2}} + \ln \frac{2n}{n - 1}}{n - 1}.
\]

Focusing on the dominant terms, the bound suggests that the average test error \( \varepsilon(V_S) \) over the space of solutions \( V_S \) is low when the number of datapoints \( n \) exceeds the number of parameters \( md \) less the entropy \( \ln K \) of the multitude of distinct solutions. The theory has two main implications:

1. In the “over-parameterised” regime \( md \gg n \), generalisation can still occur if the number of distinct solutions \( K \) is exponential in the number of parameters \( md \). In practice, \( \ln K \) might be increased relative to \( md \) by constraining the architecture based on the symmetries of the data—e.g. using convolutions for image data.

2. All else equal, solutions with larger \( \alpha \)-robustness may generalise better. In practice, \( \alpha \) might be increased by regularising the training procedure (Foret et al., 2021).

Future work might investigate these ideas more thoroughly.

7 Conclusion

This paper has proposed the Nero optimiser based on a combined study of optimisation and neural architecture. Nero pairs two ingredients: (1) projected gradient descent over the space of balanced networks; and (2) per-neuron relative updates. Taken together, a Nero update turns each neuron through an angle set by the learning rate.

Nero was found to have strong out-of-the-box performance. In almost all the experiments in this paper—spanning GAN training, image classification, natural language processing and reinforcement learning—Nero trained well using its default hyperparameter settings. The two exceptions were the 100 layer MLP and the WMT16 En–De transformer, for which Nero required a reduced learning rate of \( \eta = 0.001 \). Thus Nero has the potential to accelerate deep learning research and development, since the need for time and energy intensive hyperparameter search may be reduced.
References


Appendix A  Experimental Details

All code is to be found at github.com/jxbz/nero. This appendix records important details of the implementations and their hyperparameters.

MNIST classification These experiments used a multilayer perceptron (MLP) network. An L-layer architecture consisted of \((L - 1)\) layers of dimension \(784 \times 784\) followed by an output layer of dimension \(784 \times 10\). A “scaled relu” nonlinearity was used, defined by \(\phi(x) := \sqrt{2} \cdot \max(0, x)\). The factor of \(\sqrt{2}\) was motivated by Kaiming init [He et al., 2015] and was not tuned. The reparameterisation experiment used \(L = 5\) layers and trained for 5 epochs without learning rate decay. The very deep MLP used \(L = 100\) layers and trained for 50 epochs with the learning rate decayed by a factor of 0.9 at the end of every epoch, and with the initial learning rate tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\). Training took place on an unknown Google Colab GPU. On an NVIDIA Tesla P100 GPU, the 5-layer MLP took \(\sim 1\) minute to train and the 100-layer MLP took \(\sim 30\) minutes.

CIFAR-10 eGAN Equal learning rates were used in the generator and discriminator. The initial learning rate was tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\) for all optimisers. The networks were trained for 120 epochs, with the learning rate decayed by a factor of 10 at epoch 100. The momentum parameter in SGD and \(\beta_1\) in Adam and LAMB were tuned over \(\{0.0, 0.9\}\). Nero’s \(\beta\) and \(\beta_2\) in Adam and LAMB were set to 0.999 without tuning. Training took around 3 hours on an NVIDIA RTX 2080Ti GPU.

CIFAR-10 classification All models were trained for 200 epochs, with 5 epochs of linear learning rate warm-up and learning rate decay by a factor of 0.2 at epochs 100, 150 and 180. The initial learning rates were tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\). Training was performed on an NVIDIA RTX 2080Ti GPU. Training time for the VGG-11 network was \(\sim 1\) hour, and for ResNet-18 was \(\sim 2\) hours.

Since the experiments in Figures 1 and 2 were intended to probe the fundamental properties of optimisers rather than their performance under a limited tuning budget, a more fine-grained learning rate search was conducted. Specifically, the learning rates were tuned over \(\{0.01, 0.02, 0.04, 0.06, 0.08, 0.1\}\). The best results are listed in the following table:

<table>
<thead>
<tr>
<th>Optimiser</th>
<th>Fix Mean</th>
<th>Fix Norm</th>
<th>Top 1 Error</th>
<th>Best (\eta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nero</td>
<td>✓</td>
<td>✓</td>
<td>12.17% ± 0.08</td>
<td>0.02</td>
</tr>
<tr>
<td>Nero</td>
<td>✓</td>
<td></td>
<td>11.99% ± 0.14</td>
<td>0.01</td>
</tr>
<tr>
<td>Nero</td>
<td>✓</td>
<td>✓</td>
<td>10.03% ± 0.24</td>
<td>0.02</td>
</tr>
<tr>
<td>Nero</td>
<td>✓</td>
<td>✓</td>
<td>8.61% ± 0.22</td>
<td>0.02</td>
</tr>
<tr>
<td>Madam</td>
<td>✓</td>
<td>✓</td>
<td>12.77% ± 0.20</td>
<td>0.02</td>
</tr>
<tr>
<td>Madam</td>
<td>✓</td>
<td>✓</td>
<td>12.06% ± 0.12</td>
<td>0.06</td>
</tr>
<tr>
<td>LAMB</td>
<td>✓</td>
<td>✓</td>
<td>12.73% ± 0.10</td>
<td>0.02</td>
</tr>
<tr>
<td>LAMB</td>
<td>✓</td>
<td>✓</td>
<td>8.88% ± 0.08</td>
<td>0.06</td>
</tr>
</tbody>
</table>

ImageNet classification For training with SGD + momentum + weight decay, the initial learning was set to 0.1, momentum was set to 0.9 and weight decay was set to 0.0001. These settings follow He et al. [2016]. One epoch of linear learning rate warm-up was used, followed by 89 epochs of cosine annealing. The batch size was set to 200 for ResNet-50 to fit the GPU vRAM budget, and this was in the range known to yield good performance [Goyal et al., 2017]. This paper’s implementation surpassed the target ImageNet top-1 accuracy of 76.3% for ResNet-50 [Goyal et al., 2017; You et al., 2020]. The training was distributed over four NVIDIA RTX 2080Ti GPUs, taking \(\sim 45\) hours per training run. The total number of GPU hours for all ImageNet experiments in this paper was \(\sim 1500\).

Wikitext-2 language model The small transformer model was trained for 20 epochs, with the learning rate decayed by a factor of 0.1 at epoch 10. The initial learning rate was tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\). The batch size was set to 20. Training on an NVIDIA RTX 2080Ti GPU took \(\sim 15\) minutes.

WMT16 En–De translation The large transformer model was trained for 100 epochs, with a linear warm-up from epoch 0 to 50, and linear annealing from epoch 50 to 100. The maximum learning rate was tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\). A batch size of 128 was used. Training took \(\sim 1\) hour on an NVIDIA RTX 2080Ti GPU.

Reinforcement learning Hyperparameter settings followed Kostrikov [2018], except for the initial learning rate and the total number of environment steps. The number of steps was fixed to 5 million, and the initial learning rate was tuned over \(\{0.0001, 0.001, 0.01, 0.1, 1.0\}\). The policy network combined convolutional image feature extractors with dense output layers. Training was performed on an NVIDIA RTX 2080Ti GPU, and the training time was \(\sim 1.5\) hours.