Evolutionary Computing and Machine Learning

陈辉

Contents

Part I
文献

1 *Population-Based-Training-of-Neural-Networks*

(P1)

1.1 Introduction

Keypoints of The Problem

- Hyperparameters tuning is essential. Wiki.
- Non-stationary problems might lead to non-stationary hyperparameters.
- Tow main tuning tracks:
  - Parallel search.
  - Sequential search.

Brief Introduction to PBT

What PBT can do

1.2 Related Work

2 *Simple-Evolutionary-Optimization-Can-Rival-Stochastic-Gradient-Descent-in-Neural-Networks*

(P1)

Population Based Training of Neural Networks

Max Jaderberg **Valentin Dalibard** **Simon Osindero** **Wojciech M. Czarnecki**

Jeff Donahue **Ali Razavi** **Oriol Vinyals** **Tim Green** **Iain Dunning**

Karen Simonyan **Chrisantha Fernando** **Koray Kavukcuoglu**

DeepMind, London, UK

Abstract

Neural networks dominate the modern machine learning landscape, but their training and success still suffer from sensitivity to empirical choices of hyperparameters such as model architecture, loss function, and optimisation algorithm. In this work we present *Population Based Training (PBT)*, a simple asynchronous optimisation algorithm which effectively utilises a fixed computational budget to jointly optimise a population of models and their hyperparameters to maximise performance. Importantly, PBT discovers a schedule of hyperparameter settings rather than following the generally sub-optimal strategy of trying to find a single fixed set to use for the whole course of training. With just a small modification to a typical distributed hyperparameter training framework, our method allows robust and reliable training of models. We demonstrate the effectiveness of PBT on deep reinforcement learning problems, showing faster wall-clock convergence and higher final performance of agents by optimising over a suite of hyperparameters. In addition, we show the same method can be applied to supervised learning for machine translation, where PBT is used to maximise the BLEU score directly, and also to training of Generative Adversarial Networks to maximise the Inception score of generated images. In all cases PBT results in the automatic discovery of hyperparameter schedules and model selection which results in stable training and better final performance.

1 Introduction

Neural networks have become the workhorse non-linear function approximator in a number of machine learning domains, most notably leading to significant advances in reinforcement learning (RL) and supervised learning. However, it is often overlooked that the success of a particular neural network model depends upon the joint tuning of the model structure, the data presented, and the details of how the model is optimised. Each of these components of a learning framework is controlled by a number of parameters, *hyperparameters*, which influence the learning process and must be properly tuned to fully unlock the network performance. This essential tuning process is computationally expensive, and as neural network systems become more complicated and endowed with even more hyperparameters, the burden of this search process becomes increasingly heavy. Furthermore, there is yet another level of complexity in scenarios such as deep reinforcement learning, where the learning problem itself can be highly non-stationary (*e.g.* dependent on which parts of an environment an agent is currently able to explore). As a consequence, it might be the case that the ideal hyperparameters for such learning problems are themselves highly non-stationary, and should vary in a way that precludes setting their schedule in advance.

Two common tracks for the tuning of hyperparameters exist: *parallel search* and *sequential optimisation*, which trade-off concurrently used computational resources with the time required to achieve optimal results. Parallel search performs many parallel optimisation processes (by *optimisation process* we refer to neural network training runs), each with different hyperparameters, with a view to finding a single best output from one of the optimisation processes – examples of this are grid search and random search. Sequential optimisation performs few optimisation processes in parallel, but does so many times sequentially, to gradually perform hyperparameter optimisation using information obtained from earlier training runs to inform later ones – examples of this are hand tuning and Bayesian optimisation. Sequential optimisation will in general provide the best solutions, but requires multiple sequential training runs, which is often unfeasible for lengthy optimisation processes.

In this work, we present a simple method, Population Based Training (PBT) which bridges and extends parallel search methods and sequential optimisation methods. Advantageously, our proposal has a wall-clock run time that is no greater than that of a single optimisation process, does not require sequential runs, and is also able to use fewer computational resources than naive search methods such as random or grid search. Our approach leverages information sharing across a population of concurrently running optimisation processes, and allows for online propagation/transfer of parameters and hyperparameters between members of the population based on their performance. Furthermore, unlike most other adaptation schemes, our method is capable of performing online adaptation of hyperparameters – which can be particularly important in problems with highly non-stationary learning dynamics, such as reinforcement learning settings. PBT is decentralised and asynchronous, requiring minimal overhead and infrastructure. While inherently greedy, we show that this meta-optimisation process results in effective and automatic tuning of hyperparameters, allowing them to be adaptive throughout training. In addition, the model selection and propagation process ensures that intermediate good models are given more computational resources, and are used as a basis of further optimisation and hyperparameter search.

We apply PBT to a diverse set of problems and domains to demonstrate its effectiveness and wide applicability. Firstly, we look at the problem of deep reinforcement learning, showing how PBT can be used to optimise UNREAL (Jaderberg et al., 2016) on DeepMind Lab levels (Beattie et al., 2016), Feudal Networks (Vezhnevets et al., 2017) on Atari games (Bellemare et al., 2013), and simple A3C agents for StarCraft II (Vinyals et al., 2017). In all three cases we show faster learning and higher performance across a suite of tasks, with PBT allowing discovery of new state-of-the-art performance and behaviour, as well as the potential to reduce the computational resources for training. Secondly, we look at its use in supervised learning for machine translation on WMT 2014 English-to-German with Transformer networks (Vaswani et al., 2017), showing that PBT can match and even outperform heavily tuned hyperparameter schedules by optimising for BLEU score directly. Finally, we apply PBT to the training of Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) by optimising for the Inception score (Salimans et al., 2016) – the result is stable GAN training with large performance improvements over a strong, well-tuned baseline with the same architecture (Radford et al., 2016).

The improvements we show empirically are the result of (a) automatic selection of hyperparameters during training, (b) online model selection to maximise the use of computation spent on promising models, and (c) the ability for online adaptation of hyperparameters to enable non-stationary training regimes and the discovery of complex hyperparameter schedules.

In Sect. 2 we review related work on parallel and sequential hyperparameter optimisation techniques, as well as evolutionary optimisation methods which bear a resemblance to PBT. We introduce PBT in Sect. 3, outlining the algorithm in a general manner which is used as a basis for experiments. The specific incarnations of PBT and the results of experiments in different domains are given in Sect. 4, and finally we conclude in Sect. 5.

2 Related Work

First, we review the methods for sequential optimisation and parallel search for hyperparameter optimisation that go beyond grid search, random search, and hand tuning. Next, we note that PBT inherits many ideas from genetic algorithms, and so highlight the literature in this space.

The majority of automatic hyperparameter tuning mechanisms are sequential optimisation methods: the result of each training run with a particular set of hyperparameters is used as knowledge to inform the subsequent hyperparameters searched. A lot of previous work uses a Bayesian optimisation framework to incorporate this information by updating the posterior of a Bayesian model of successful hyperparameters for training – examples include GP-UCB (Srinivas et al., 2009), TPE (Bergstra et al., 2011), Spearmint (Snoek et al., 2012), and SMAC (Hutter et al., 2011). As noted in the previous section, their sequential nature makes these methods prohibitively slow for expensive optimisation processes. For this reason, a number of methods look to speed up the updating of the posterior with information from each optimisation process by performing early stopping, using intermediate losses to predict final performance, and even modelling the entire time and data dependent optimisation process in order to reduce the number of optimisation steps required for each optimisation process and to better explore the space of promising hyperparameters (György & Kocsis, 2011; Agarwal et al., 2011; Sabharwal et al., 2016; Swersky et al., 2013, 2014; Domhan et al., 2015; Klein et al., 2016; Snoek et al., 2015; Springenberg et al., 2016).

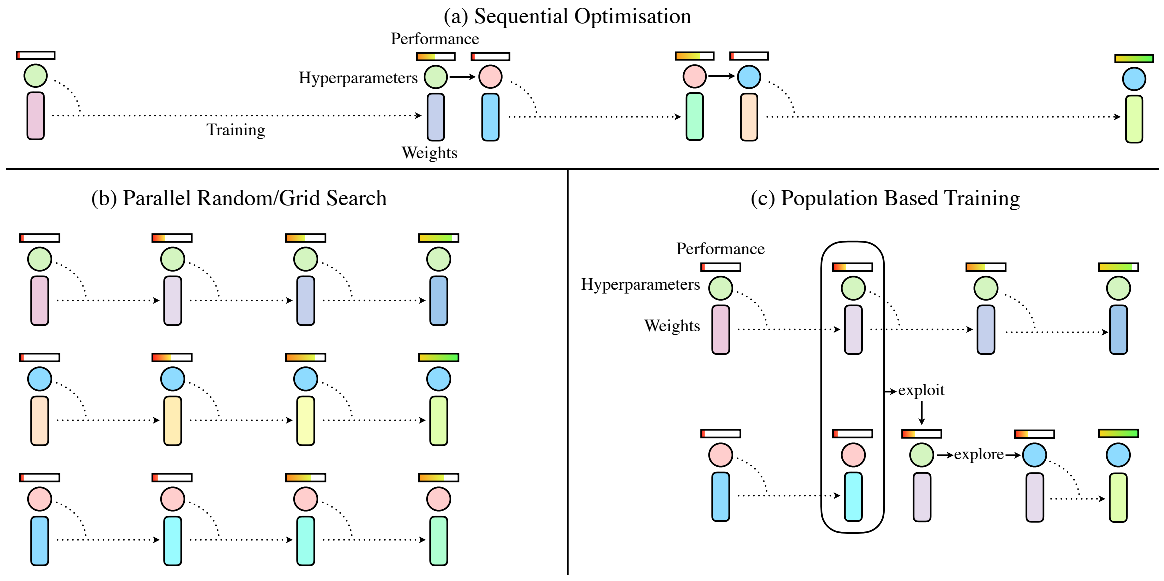


Figure 1: Common paradigms of tuning hyperparameters: *sequential optimisation* and *parallel search*, as compared to the method of *population based training* introduced in this work. (a) Sequential optimisation requires multiple training runs to be completed (potentially with early stopping), after which new hyperparameters are selected and the model is retrained from scratch with the new hyperparameters. This is an inherently sequential process and leads to long hyperparameter optimisation times, though uses minimal computational resources. (b) Parallel random/grid search of hyperparameters trains multiple models in parallel with different weight initialisations and hyperparameters, with the view that one of the models will be optimised the best. This only requires the time for one training run, but requires the use of more computational resources to train many models in parallel. (c) Population based training starts like parallel search, randomly sampling hyperparameters and weight initialisations. However, each training run asynchronously evaluates its performance periodically. If a model in the population is under-performing, it will *exploit* the rest of the population by replacing itself with a better performing model, and it will *explore* new hyperparameters by modifying the better model's hyperparameters, before training is continued. This process allows hyperparameters to be optimised online, and the computational resources to be focused on the hyperparameter and weight space that has most chance of producing good results. The result is a hyperparameter tuning method that while very simple, results in faster learning, lower computational resources, and often better solutions.

Many of these Bayesian optimisation approaches can be further sped up by attempting to parallelise them – training independent models in parallel to quicker update the Bayesian model (though potentially introducing bias) (Shah & Ghahramani, 2015; González et al., 2016; Wu & Frazier, 2016; Rasley et al., 2017; Golovin et al., 2017) – but these will still require multiple sequential model optimisations. The same problem is also encountered where genetic algorithms are used in place of Bayesian optimisation for hyperparameter evolution (Young et al., 2015).

Moving away from Bayesian optimisation, even approaches such as Hyperband (Li et al., 2016) which model the problem of hyperparameter selection as a many-armed bandit, and natively incorporate parallelisation, cannot practically be executed within a single training optimisation process due to the large amount of computational resources they would initially require.

The method we present in this paper only requires a single training optimisation process – it builds upon the unreasonable success of random hyperparameter search which has shown to be very effective (Bergstra & Bengio, 2012). Random search is effective at finding good regions for sensitive hyperparameters – PBT identifies these and ensures that these areas are explored more using partially trained models (*e.g.* by copying their weights). By bootstrapping the evaluation of new hyperparameters during training on partially trained models we eliminate the need for sequential optimisation processes that plagues the methods previously discussed. PBT also allows for adaptive hyperparameters during training, which has been shown, especially for learning rates, to improve optimisation (Loshchilov & Hutter, 2016; Smith, 2017; Massé & Ollivier, 2015).

PBT is perhaps most analogous to evolutionary strategies that employ self-adaptive hyperparameter tuning to modify how the genetic algorithm itself operates (Bäck, 1998; Spears, 1995; Gloger, 2004; Clune et al., 2008) – in these works, the evolutionary hyperparameters were mutated at a slower rate than the parameters themselves, similar to how hyperparameters in PBT are adjusted at a slower rate than the parameters. In our case however, rather than evolving the parameters of the model, we train them partially with standard optimisation techniques (*e.g.* gradient descent). Other similar works to ours are Lamarckian evolutionary algorithms in which parameters are inherited whilst hyperparameters are evolved (Castillo et al., 2006) or where the hyperparameters, initial weights, and architecture of networks are evolved, but the parameters are learned (Castillo et al., 1999; Husken et al., 2000), or where evolution and learning are both applied to the parameters (Ku & Mak, 1997; Houck et al., 1997; Igel et al., 2005). This mixture of learning and evolutionary-like algorithms has also been explored successfully in other domains (Zhang et al., 2011) such as neural architecture search (Real et al., 2017; Liu et al., 2017), feature selection (Xue et al., 2016), and parameter learning (Fernando et al., 2016). Finally, there are parallels to work such as Salustowicz & Schmidhuber (1997) which perform program search with genetic algorithms.

3 Population Based Training

The most common formulation in machine learning is to optimise the parameters θ of a model f to maximise a given objective function $\hat{\mathcal{Q}}$, (*e.g.* classification, reconstruction, or prediction). Generally, the trainable parameters θ are updated using an optimisation procedure such as stochastic gradient descent. More importantly, the actual performance metric $\mathcal{Q}$ that we truly care to optimise is often different to $\hat{\mathcal{Q}}$, for example $\mathcal{Q}$ could be accuracy on a validation set, or BLEU score as used in machine translation. The main purpose of PBT is to provide a way to optimise both the parameters θ and the hyperparameters h jointly on the actual metric $\mathcal{Q}$ that we care about.

To that end, firstly we define a function `eval` that evaluates the objective function, $\mathcal{Q}$, using the current state of model f. For simplicity we ignore all terms except θ, and define `eval` as function of trainable parameters θ only. The process of finding the optimal set of parameters that maximise $\mathcal{Q}$ is then:

$$\theta^* = \arg\max_{\theta \in \Theta} \texttt{eval}(\theta). \tag{1}$$

Note that in the methods we are proposing, the `eval` function does not need to be differentiable, nor does it need to be the same as the function used to compute the iterative updates in the optimisation steps which we outline below (although they ought to be correlated).

When our model is a neural network, we generally optimise the weights θ in an iterative manner, *e.g.* by using stochastic gradient descent on the objective function $\hat{\mathcal{Q}}$. Each step of this iterative optimisation procedure, `step`, updates the parameters of the model, and is itself conditioned on some parameters $h \in \mathcal{H}$ (often referred to as hyperparameters).

In more detail, iterations of parameter update steps:

$$\theta \leftarrow \texttt{step}(\theta|h) \tag{2}$$

are chained to form a sequence of updates that ideally converges to the optimal solution

$$\theta^* = \texttt{optimise}(\theta|\boldsymbol{h}) = \texttt{optimise}(\theta|(h_t)_{t=1}^T) = \texttt{step}(\texttt{step}(\ldots\texttt{step}(\theta|h_1)\ldots|h_{T-1})|h_T). \tag{3}$$

This iterative optimisation process can be computationally expensive, due to the number of steps T required to find θ^* as well as the computational cost of each individual step, often resulting in the optimisation of θ taking days, weeks, or even months. In addition, the solution is typically very sensitive to the choice of the sequence of hyperparemeters $\boldsymbol{h} = (h_t)_{t=1}^T$. Incorrectly chosen hyperparameters can lead to bad solutions or even a failure of the optimisation of θ to converge. Correctly selecting hyperparameters requires strong prior knowledge on $\boldsymbol{h}$ to exist or to be found (most often through multiple optimisation processes with different $\boldsymbol{h}$). Furthermore, due to the dependence of $\boldsymbol{h}$ on iteration step, the number of possible values grows exponentially with time. Consequently, it is common practise to either make all h_t equal to each other (*e.g.* constant learning rate through entire training, or constant regularisation strength) or to predefine a simple schedule (*e.g.* learning rate annealing). In both cases one needs to search over multiple possible values of $\boldsymbol{h}$

$$\theta^* = \texttt{optimise}(\theta|\boldsymbol{h}^*),\ \boldsymbol{h}^* = \arg\max_{\boldsymbol{h} \in \mathcal{H}^T} \texttt{eval}(\texttt{optimise}(\theta|\boldsymbol{h})). \tag{4}$$

As a way to perform (4) in a fast and computationally efficient manner, we consider training N models $\{\theta^i\}_{i=1}^N$ forming a *population* $\mathcal{P}$ which are optimised with different hyperparameters $\{\boldsymbol{h}^i\}_{i=1}^N$. The objective

Algorithm 1 Population Based Training (PBT)

```
 1: procedure TRAIN($\mathcal{P}$)                                    ▷ initial population $\mathcal{P}$
 2:     for ($\theta$, $h$, $p$, $t$) $\in \mathcal{P}$ (asynchronously in parallel) do
 3:         while not end of training do
 4:             $\theta \leftarrow$ step($\theta|h$)          ▷ one step of optimisation using hyperparameters $h$
 5:             $p \leftarrow$ eval($\theta$)                                  ▷ current model evaluation
 6:             if ready($p, t, \mathcal{P}$) then
 7:                 $h', \theta' \leftarrow$ exploit($h, \theta, p, \mathcal{P}$)   ▷ use the rest of population to find better solution
 8:                 if $\theta \neq \theta'$ then
 9:                     $h, \theta \leftarrow$ explore($h', \theta', \mathcal{P}$)        ▷ produce new hyperparameters $h$
10:                     $p \leftarrow$ eval($\theta$)                              ▷ new model evaluation
11:                 end if
12:             end if
13:             update $\mathcal{P}$ with new ($\theta, h, p, t+1$)                   ▷ update population
14:         end while
15:     end for
16:     return $\theta$ with the highest $p$ in $\mathcal{P}$
17: end procedure
```

is to therefore find the optimal model across the entire population. However, rather than taking the approach of parallel search, we propose to use the collection of partial solutions in the population to additionally perform meta-optimisation, where the hyperparameters h and weights θ are additionally adapted according to the performance of the entire population.

In order to achieve this, PBT uses two methods called independently on each member of the population (each worker): `exploit`, which, given performance of the whole population, can decide whether the worker should abandon the current solution and instead focus on a more promising one; and `explore`, which given the current solution and hyperparameters proposes new ones to better explore the solution space.

Each member of the population is trained in parallel, with iterative calls to `step` to update the member's weights and `eval` to measure the member's current performance. However, when a member of the population is deemed ready (for example, by having been optimised for a minimum number of steps or having reached a certain performance threshold), its weights and hyperparameters are updated by `exploit` and `explore`. For example, `exploit` could replace the current weights with the weights that have the highest recorded performance in the rest of the population, and `explore` could randomly perturb the hyperparameters with noise. After `exploit` and `explore`, iterative training continues using `step` as before. This cycle of local iterative training (with `step`) and exploitation and exploration using the rest of the population (with `exploit` and `explore`) is repeated until convergence of the model. Algorithm 1 describes this approach in more detail, Fig. 1 schematically illustrates this process (and contrasts it with sequential optimisation and parallel search), and Fig. 2 shows a toy example illustrating the efficacy of PBT.

The specific form of `exploit` and `explore` depends on the application. In this work we focus on optimising neural networks for reinforcement learning, supervised learning, and generative modelling with PBT (Sect. 4). In these cases, `step` is a step of gradient descent (with *e.g.* SGD or RMSProp (Tieleman & Hinton, 2012)), `eval` is the mean episodic return or validation set performance of the metric we aim to optimise, `exploit` selects another member of the population to copy the weights and hyperparameters from, and `explore` creates new hyperparameters for the next steps of gradient-based learning by either perturbing the copied hyperparameters or resampling hyperparameters from the originally defined prior distribution. A member of the population is deemed ready to exploit and explore when it has been trained with gradient descent for a number of steps since the last change to the hyperparameters, such that the number of steps is large enough to allow significant gradient-based learning to have occurred.

By combining multiple steps of gradient descent followed by weight copying by `exploit`, and perturbation of hyperparameters by `explore`, we obtain learning algorithms which benefit from not only local optimisation by gradient descent, but also periodic model selection, and hyperparameter refinement from a process that is more similar to genetic algorithms, creating a two-timescale learning system. An important property of population based training is that it is asynchronous and does not require a centralised process to orchestrate the training of the members of the population. Only the current performance information, weights, and

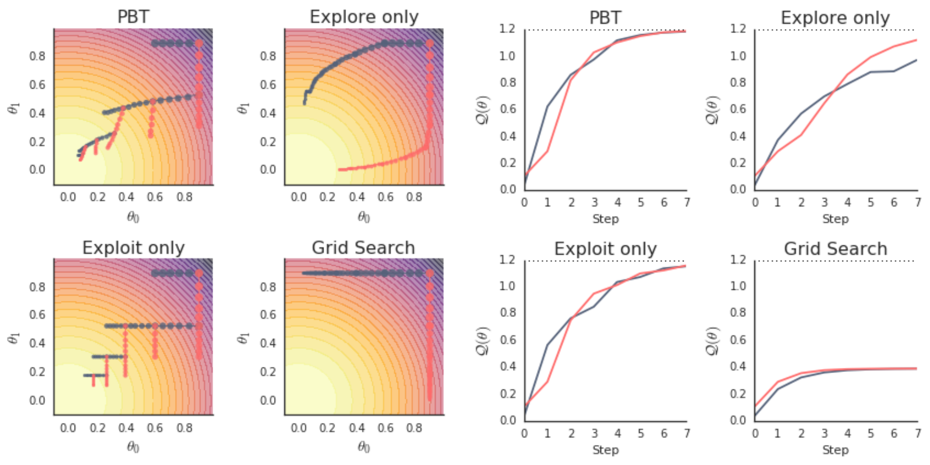


Figure 2: A visualisation of PBT on a toy example where our objective is to maximise a simple quadratic function $\mathcal{Q}(\theta) = 1.2 - (\theta_0^2 + \theta_1^2)$, but without knowing its formula. Instead, we are given a surrogate function $\hat{\mathcal{Q}}(\theta|h) = 1.2 - (h_0\theta_0^2 + h_1\theta_1^2)$ that we can differentiate. This is a very simple example of the setting where one wants to maximise one metric (*e.g.* generalisation capabilities of a model) while only being able to directly optimise other one (*e.g.* empirical loss). We can, however, read out the values of $\mathcal{Q}$ for evaluation purposes. We can treat h as hyperparameters, which can change during optimisation, and use gradient descent to minimise $Q(\theta|h)$. For simplicity, we assume there are only two workers (so we can only run two full maximisations), and that we are given initial $\theta_0 = [0.9, 0.9]$. With grid/random search style approaches we can only test two values of h, thus we choose $[1, 0]$ (black), and $[0, 1]$ (red). Note, that if we would choose $[1, 1]$ we would recover true objective, but we are assuming that we do not know that in this experiment, and in general there is no h such that $\hat{\mathcal{Q}}(\theta|h) = \mathcal{Q}(\theta)$. As one can see, we increase $\mathcal{Q}$ in each step of maximising $Q(\cdot|h)$, but end up far from the actual optimum ($\mathcal{Q}(\theta) \approx 0.4$), due to very limited exploration of hyperparameters h. If we apply PBT to the same problem where every 4 iterations the weaker of the two workers copies the solution of the better one (exploitation) and then slightly perturbs its update direction (exploration), we converge to a global optimum. Two ablation results show what happens if we only use exploration or only exploitation. We see that majority of the performance boost comes from copying solutions between workers (exploitation), and exploration only provides additional small improvement. Similar results are obtained in the experiments on real problems, presented in Sect. 4.4. *Left*: The learning over time, each dot represents a single solution and its size decreases with iteration. *Right*: The objective function value of each worker over time.

hyperparameters must be globally available for each population member to access – crucially there is no synchronisation of the population required.

4 Experiments

In this section we will apply Population Based Training to different learning problems. We describe the specific form of PBT for deep reinforcement learning in Sect. 4.1 when applied to optimising UNREAL (Jaderberg et al., 2016) on DeepMind Lab 3D environment tasks (Beattie et al., 2016), Feudal Networks (Vezhnevets et al., 2017) on the Atari Learning Environment games (Bellemare et al., 2013), and the StarCraft II environment baseline agents (Vinyals et al., 2017). In Sect. 4.2 we apply PBT to optimising state-of-the-art language models, transformer networks (Vaswani et al., 2017), for machine translation task. Next, in Sect. 4.3 we apply PBT to the optimisation of Generative Adversarial Networks (Goodfellow et al., 2014), a notoriously unstable optimisation problem. In all these domains we aim to build upon the strongest baselines, and show improvements in training these state-of-the-art models by using PBT. Finally, we analyse the design space of Population Based Training with respect to this rich set of experimental results to draw practical guidelines for further use in Sect. 4.4.

4.1 Deep Reinforcement Learning

In this section we apply Population Based Training to the training of neural network agents with reinforcement learning (RL) where we aim to find a policy π to maximise expected episodic return $\mathbb{E}_\pi[R]$ within an environment. We first focus on A3C-style methods Mnih et al. (2016) to perform this maximisation, along with some state-of-the-art extensions: UNREAL (Jaderberg et al., 2016) and Feudal Networks (FuN) (Vezhnevets et al., 2017).

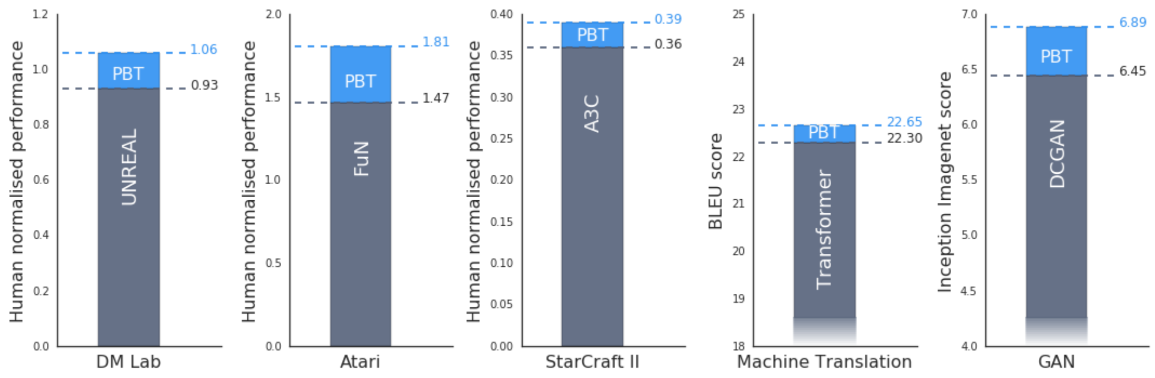


Figure 3: The results of using PBT over random search on different domains. *DM Lab:* We show results for training UNREAL with and without PBT, using a population of 40 workers, on 15 DeepMind Lab levels, and report the final human-normalised performance of the best agent in the population averaged over all levels. *Atari:* We show results for Feudal Networks with and without PBT, using a population of 80 workers on each of four games: Amidar, Gravitar, Ms-Pacman, and Enduro, and report the final human-normalised performance of the best agent in the population averaged over all games. *StarCraft II:* We show the results for training A3C with and without PBT, using a population of 30 workers, on a suite of 6 mini-game levels, and report the final human-normalised performance of the best agent in the population averaged over all levels. *Machine Translation:* We show results of training Transformer networks for machine translation on WMT 2014 English-to-German. We use a population of 32 workers with and without PBT, optimising the population to maximise BLEU score. *GAN:* We show results for training GANs with and without PBT using a population size of 45, optimising for Inception score. Full results on all levels can be found in Sect. A.2.

These algorithms are sensitive to hyperparameters, which include learning rate, entropy cost, and auxiliary loss weights, as well as being sensitive to the reinforcement learning optimisation process which can suffer from local minima or collapse due to the insufficient or unlucky policy exploration. PBT therefore aims to stabilise this process.

4.1.1 PBT for RL

We use population sizes between 10 and 80, where each member of the population is itself a distributed asynchronous actor critic (A3C) style agent (Mnih et al., 2016).

Hyperparameters We allow PBT to optimise the learning rate, entropy cost, and unroll length for UNREAL on DeepMind Lab, learning rate, entropy cost, and intrinsic reward cost for FuN on Atari, and learning rate only for A3C on StarCraft II.

Step Each iteration does a step of gradient descent with RMSProp (Tieleman & Hinton, 2012) on the model weights, with the gradient update provided by vanilla A3C, UNREAL or FuN.

Eval We evaluate the current model with the last 10 episodic rewards during training.

Ready A member of the population is deemed ready to go through the exploit-and-explore process using the rest of the population when between 1×10^6 to 10×10^6 agent steps have elapsed since the last time that population member was ready.

Exploit We consider two exploitation strategies. (a) *T-test selection* where we uniformly sample another agent in the population, and compare the last 10 episodic rewards using Welch's t-test (Welch, 1947). If the sampled agent has a higher mean episodic reward and satisfies the t-test, the weights and hyperparameters are copied to replace the current agent. (b) *Truncation selection* where we rank all agents in the population by episodic reward. If the current agent is in the bottom 20% of the population, we sample another agent uniformly from the top 20% of the population, and copy its weights and hyperparameters.

Explore We consider two exploration strategies in hyperparameter space. (a) *Perturb*, where each hyperparameter independently is randomly perturbed by a factor of 1.2 or 0.8. (b) *Resample*, where each hyperparameter is resampled from the original prior distribution defined with some probability.

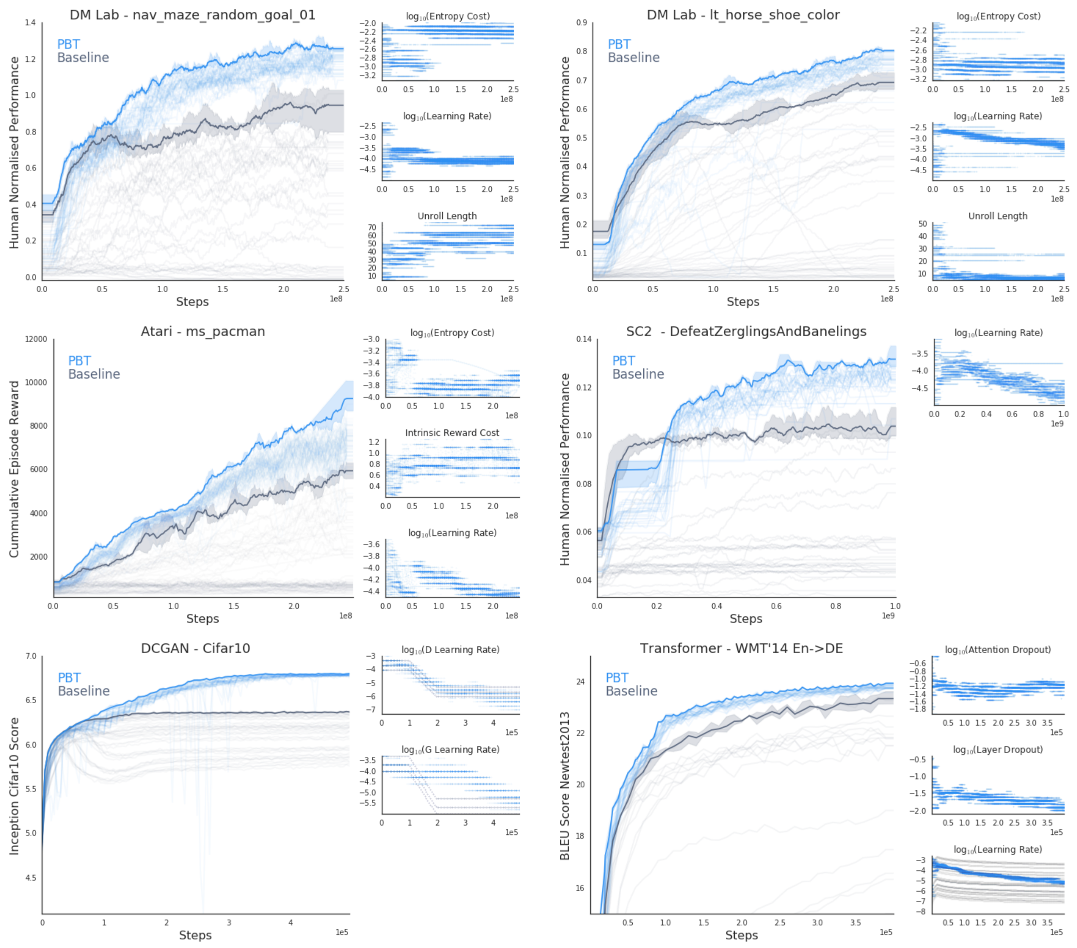


Figure 4: Training curves of populations trained with PBT (blue) and without PBT (black) for individual levels on DeepMind Lab (DM Lab), Atari, and Starcraft II (SC2), as well as GAN training on CIFAR-10 (CIFAR-10 Inception score is plotted as this is what is optimised for by PBT). Faint lines represent each individual worker, while thick lines are an average of the top five members of the population at each timestep. The hyperparameter populations are displayed to the right, and show how the populations' hyperparameters adapt throughout training, including the automatic discovery of learning rate decay and, for example, the discovery of the benefits of large unroll lengths on DM Lab.

4.1.2 Results

The high level results are summarised in Fig. 3. On all three domains – DeepMind Lab, Atari, and StarCraft II – PBT increases the final performance of the agents when trained for the same number of steps, compared to the very strong baseline of performing random search with the same number of workers.

On DeepMind Lab, when UNREAL is trained using a population of 40 workers, using PBT increases the final performance of UNREAL from 93% to 106% human performance, when performance is averaged across all levels. This represents a significant improvement over an already extremely strong baseline, with some levels such as nav_maze_random_goal_01 and lt_horse_shoe_color showing large gains in performance due to PBT, as shown in Fig. 4. PBT seems to help in two main ways: first is that the hyperparameters are clearly being focused on the best part of the sampling range, and adapted over time. For example, on lt_horse_shoe_color in Fig. 4 one can see the learning rate being annealed by PBT as training progresses, and on nav_maze_random_goal_01 in Fig. 4 one can see the unroll length – the number of steps the agent (a recurrent neural network) is unrolled before performing N-step policy gradient and backpropagation through time – is gradually increased. This allows the agent to learn to utilise the recurrent neural network for memory, allowing it to obtain superhuman performance on nav_maze_random_goal_01 which is a level which requires memorising the location of a goal in a maze, more details can be found in (Jaderberg et al., 2016). Secondly, since PBT is copying the weights of good performing agents during the exploitation phase, agents

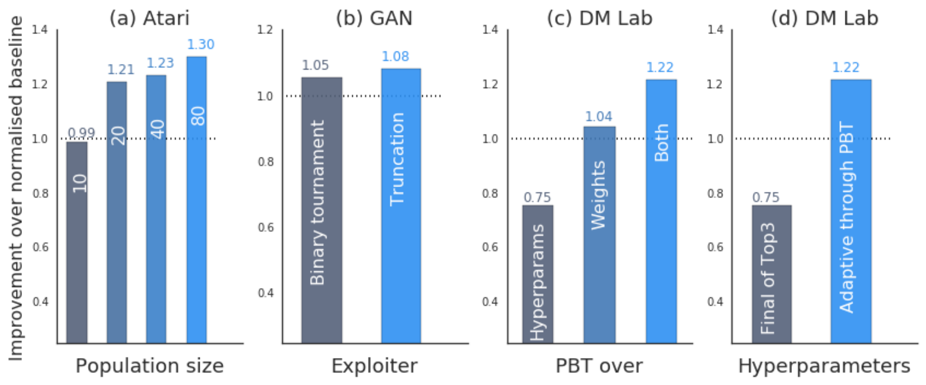


Figure 5: A highlight of results of various ablation studies on the design space of PBT, reported as improvement over the equivalent random search baseline. (a) *Population size:* We evaluate the effect of population size on the performance of PBT when training FuN on Atari. In general we find that a smaller population sizes leads to higher variance and sub-optimal results, however using a population of 20 and over achieves consistent improvements, with diminishing returns for larger populations. (b) *Exploiter:* We compare the effect of using different `exploit` functions – binary tournament and truncation selection. When training GANs, we find truncation selection to work better. (c) *PBT over:* We show the effect of removing parts of PBT when training UNREAL on DeepMind Lab, in particular only performing PBT on hyperparameters (so model weights are not copied between workers during `exploit`), and only performing PBT on model weights (so hyperparameters are not copied between workers or explored). We find that indeed it is the combination of optimising hyperparameters as well as model selection which leads to best performance. (d) *Hyperparameters:* Since PBT allows online adaptation of hyperparameters during training, we evaluate how important the adaptation is by comparing full PBT performance compared to using the set of hyperparameters that PBT found by the end of training. When training UNREAL on DeepMind Lab we find that the strength of PBT is in allowing the hyperparameters to be adaptive, not merely in finding a good prior on the space of hyperparameters.

which are lucky in environment exploration are quickly propagated to more workers, meaning that all members of the population benefit from the exploration luck of the remainder of the population.

When we apply PBT to training FuN on 4 Atari levels, we similarly see a boost in final performance, with PBT increasing the average human normalised performance from 147% to 181%. On particular levels, such as $\mathrm{ms\_pacman}$ PBT increases performance significantly from 6506 to 9001 (Fig. 4), which represents state-of-the-art for an agent receiving only pixels as input.

Finally on StarCraft II, we similarly see PBT improving A3C baselines from 36% huamn performance to 39% human performance when averaged over 6 levels, benefiting from automatic online learning rate adaptation and model selection such as shown in Fig. 4.

More detailed experimental results, and details about the specific experimental protocols can be found in Appendix A.2.

4.2 Machine Translation

As an example of applying PBT on a supervised learning problem, we look at training neural machine translation models. In this scenario, the task is to encode a source sequence of words in one language, and output a sequence in a different target language. We focus on English to German translation on the WMT 2014 English-to-German dataset, and use a state-of-the-art model, Transformer networks (Vaswani et al., 2017), as the baseline model to apply PBT to. Crucially, we also use the highly optimised learning rate schedules and hyperparameter values for our baselines to compare to – these are the result of a huge amount of hand tuning and Bayesian optimisation.

4.2.1 PBT for Machine Translation

We use a population size of 32 workers, where each member of the population is a single GPU optimising a Transformer network for 400×10^3 steps.

Hyperparameters We allow PBT to optimise the learning rate, attention dropout, layer dropout, and ReLU dropout rates.

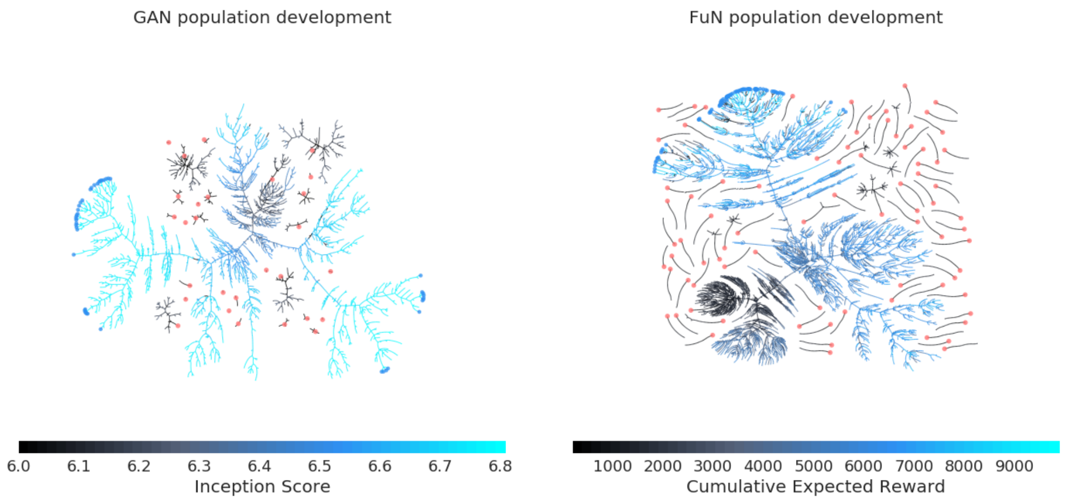

GAN population development
FuN population development
6.0
6.1
6.2
6.3
6.4
6.5
6.6
6.7
6.8
Inception Score
1000
2000
3000
4000
5000
6000
7000
8000
9000
Cumulative Expected Reward

Figure 6: Model development analysis for GAN (left) and Atari (right) experiments – full phylogenetic tree of entire training. Pink dots represent initial agents, blue ones the final ones. Branching of the graph means that the exploit operation has been executed (and so parameters were copied), while paths represent consecutive updates using the step function. Colour of the edges encode performance, from black (weak model) to cyan (strong model).

Step Each step does a step of gradient descent with Adam (Kingma & Ba, 2015).

Eval We evaluate the current model by computing the BLEU score on the WMT *newstest2012* dataset. This allows us to optimise our population for BLEU score, which usually cannot be optimised for directly.

Ready A member of the population is deemed ready to exploit and explore using the rest of the population every 2×10^3 steps.

Exploit We use the *t-test selection* exploitation method described in Sect. 4.1.1.

Explore We explore hyperparameter space by the *perturb* strategy described in Sect. 4.1.1 with 1.2 and 0.8 perturbation factors.

4.2.2 Results

When we apply PBT to the training of Transformer networks, we find that PBT actually results in models which exceed the performance of this highly tuned baseline model: BLEU score on the validation set (*newstest2012*) is improved from 23.71 to 24.23, and on the test set of WMT 2014 English-to-German is improved from 22.30 to 22.65 using PBT (here we report results using the small Transformer network using a reduced batch size, so is not representative of state of the art). PBT is able to automatically discover adaptations of various dropout rates throughout training, and a learning rate schedule that remarkably resembles that of the hand tuned baseline: very initially the learning rate starts small before jumping by three orders of magnitude, followed by something resembling an exponential decay, as shown in the lineage plot of Fig. 7. We can also see from Fig. 4 that the PBT model trains much faster.

4.3 Generative Adversarial Networks

The Generative Adversarial Networks (GAN) (Goodfellow et al., 2014) framework learns generative models via a training paradigm consisting of two competing modules – a *generator* and a *discriminator* (alternatively called a *critic*). The discriminator takes as input samples from the generator and the real data distribution, and is trained to predict whether inputs are real or generated. The generator typically maps samples from a simple noise distribution to a complex data distribution (*e.g.* images) with the objective of maximally fooling the discriminator into classifying or scoring its samples as real.

Like RL agent training, GAN training can be remarkably brittle and unstable in the face of suboptimal hyperparameter selection and even unlucky random initialisation, with generators often collapsing to a single mode or diverging entirely. Exacerbating these difficulties, the presence of two modules optimising competing objectives doubles the number of hyperparameters, often forcing researchers to choose the same hyperparameter

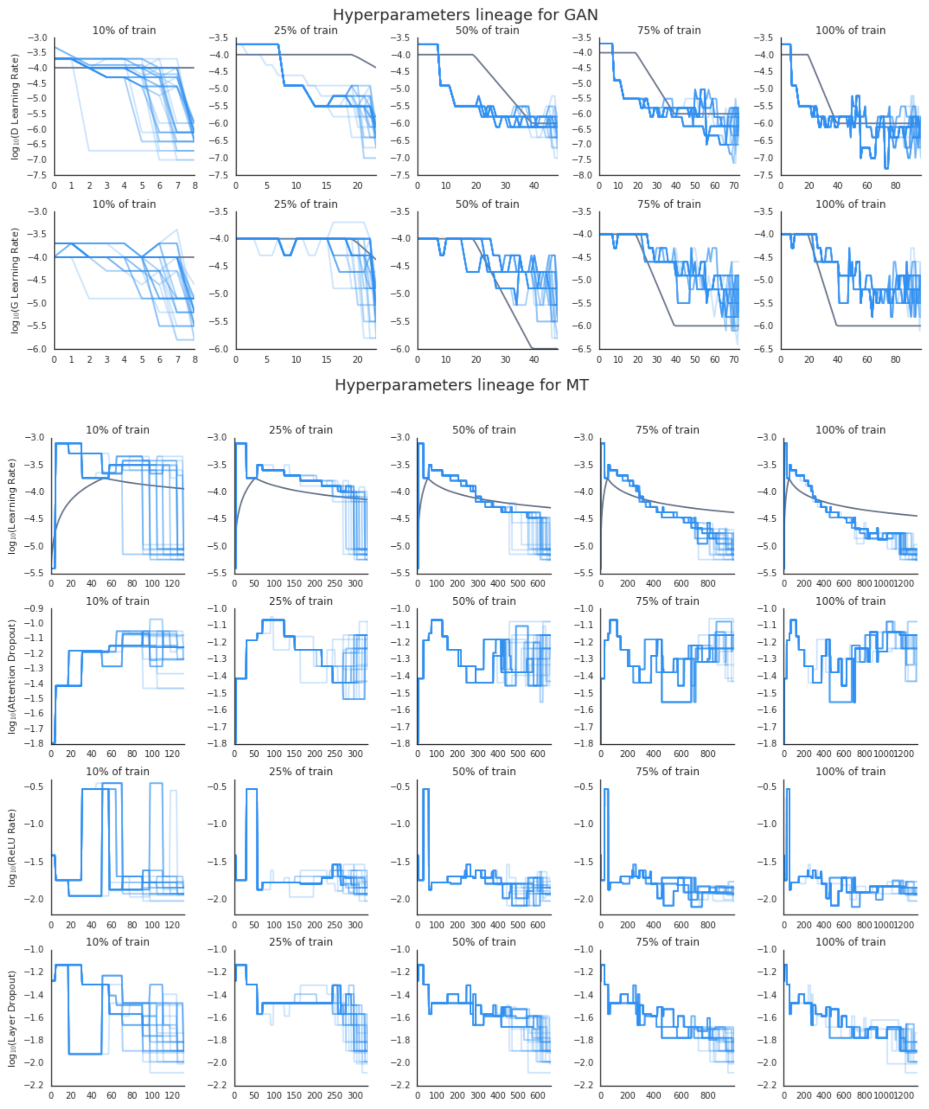


Figure 7: Agent development analysis for GAN (top) and machine translation (bottom) experiments – lineages of hyperparameters changes for the final agents after given amount of training. Black lines show the annealing schedule used by the best baseline run.

values for both modules. Given the complex learning dynamics of GANs, this unnecessary coupling of hyperparameters between the two modules is unlikely to be the best choice.

Finally, the various metrics used by the community to evaluate the quality of samples produced by GAN generators are necessarily distinct from those used for the adversarial optimisation itself. We explore whether we can improve the performance of generators under these metrics by directly targeting them as the PBT meta-optimisation evaluation criteria.

4.3.1 PBT for GANs

We train a population of size 45 with each worker being trained for 10^6 steps.

Hyperparameters We allow PBT to optimise the discriminator's learning rate and the generator's learning rate separately.

Step Each step consists of $K = 5$ gradient descent updates of the discriminator followed by a single update of the generator using the Adam optimiser (Kingma & Ba, 2015). We train using the WGAN-GP (Gulrajani et al., 2017) objective where the discriminator estimates the Wasserstein distance between the real and generated data distributions, with the Lipschitz constraint enforced by regularising its input gradient to have a unit norm. See Appendix A.3 for additional details.

Eval We evaluate the current model by a variant of the Inception score proposed by Salimans et al. (2016) computed from the outputs of a pretrained CIFAR classifier (as used in Rosca et al. (2017)) rather than an ImageNet classifier, to avoid directly optimising the final performance metric. The CIFAR Inception scorer uses a much smaller network, making evaluation much faster.

Ready A member of the population is deemed ready to exploit and explore using the rest of the population every 5×10^3 steps.

Exploit We consider two exploitation strategies. (a) *Truncation selection* as described in Sect. 4.1.1. (b) *Binary tournament* in which each member of the population randomly selects another member of the population, and copies its parameters if the other member's score is better. Whenever one member of the population is copied to another, all parameters – the hyperparameters, weights of the generator, and weights of the discriminator – are copied.

Explore We explore hyperparameter space by the *perturb* strategy described in Sect. 4.1.1, but with more aggressive perturbation factors of 2.0 or 0.5.

4.3.2 Results

We apply PBT to optimise the learning rates for a CIFAR-trained GAN discriminator and generator using architectures similar to those used DCGAN (Radford et al., 2016). For both PBT and the baseline, the population size is 45, with learning rates initialised to $\{1, 2, 5\} \times 10^{-4}$ for a total of $3^2 = 9$ initial hyperparameter settings, each replicated 5 times with independent random weight initialisations. Additional architectural and hyperparameter details can be found in Appendix A.3. The baseline utilises an exponential learning rate decay schedule, which we found to work the best among a number of hand-designed annealing strategies, detailed in Appendix A.3.

In Fig. 5 (b), we compare the *truncation selection* exploration strategy with a simpler variant, *binary tournament*. We find a slight advantage for truncation selection, and our remaining discussion focuses on results obtained by this strategy. Fig. 4 (bottom left) plots the CIFAR Inception scores and shows how the learning rates of the population change throughout the first half of training (both the learning curves and hyperparameters saturate and remain roughly constant in the latter half of training). We observe that PBT automatically learns to decay learning rates, though it does so dynamically, with irregular flattening and steepening, resulting in a learnt schedule that would not be matched by any typical exponential or linear decay protocol. Interestingly, the learned annealing schedule for the discriminator's learning rate closely tracks that of the best performing baseline (shown in grey), while the generator's schedule deviates from the baseline schedule substantially.

In Table 4 (Appendix A.3), we report standard ImageNet Inception scores as our main performance measure, but use CIFAR Inception score for model selection, as optimising ImageNet Inception score directly would potentially overfit our models to the test metric. CIFAR Inception score is used to select the best model after training is finished both for the baseline and for PBT. Our best-performing baseline achieves a CIFAR Inception score of 6.39, corresponding to an ImageNet Inception score of 6.45, similar to or better than Inception scores for DCGAN-like architectures reported in prior work (Rosca et al., 2017; Gulrajani et al., 2017; Yang et al., 2017). Using PBT, we strongly outperform this baseline, achieving a CIFAR Inception score of 6.80 and corresponding ImageNet Inception score of 6.89. For reference, Table 4 (Appendix A.3) also reports the peak ImageNet Inception scores obtained by any population member at any point during training. Fig. 8 shows samples from the best generators with and without PBT.

4.4 Analysis

In this section we analyse and discuss the effects of some of the design choices in setting up PBT, as well as probing some aspects of why the method is so successful.

Population Size In Fig. 5 (a) we demonstrate the effect of population size on the performance of PBT when training FuN on Atari. In general, we find that if the population size is too small (10 or below) we tend to encounter higher variance and can suffer from poorer results – this is to be expected as PBT is a greedy algorithm and so can get stuck in local optima if there is not sufficient population to maintain diversity and scope for exploration. However, these problems rapidly disappear as population size increases and we see improved results as the population size grows. In our experiments, we observe that a population size of between 20 and 40 is sufficient to see strong and consistent improvements; larger populations tend to fare even better, although we see diminishing returns for the cost of additional population members.

Exploitation Type In Fig. 5 (b) we compare the effect of using different `exploit` functions – binary tournament and truncation selection. For GAN training, we find truncation selection to work better. Similar behaviour was seen in our preliminary experiments with FuN on Atari as well. Further work is required to fully understand the impact of different `exploit` mechanisms, however it appears that truncation selection is a robustly useful choice.

PBT Targets In Fig. 5 (c) we show the effect of only applying PBT to subsets of parameters when training UNREAL on DM Lab. In particular, we contrast (i) only performing exploit-and-explore on hyperparameters (so model weights are not copied between workers during `exploit`); and (ii) only performing exploit-and-explore on model weights (so hyperparameters are not copied between workers or explored). Whilst there is some increase in performance simply from the selection and propagation of good model weights, our results clearly demonstrate that in this setting it is indeed the *combination* of optimising hyperparameters as well as model selection that lead to our best performance.

Hyperparameter Adaptivity In Fig. 5 (d) we provide another demonstration that the benefits of PBT go beyond simply finding a single good fixed hyperparameter combination. Since PBT allows online adaptation of hyperparameters during training, we evaluate how important the adaptation is by comparing full PBT performance compared to using the set of hyperparameters that PBT found by the end of training. When training UNREAL on DeepMind Lab we find that the strength of PBT is in allowing the hyperparameters to be adaptive, not merely in finding a good prior on the space of hyperparameters

Lineages Fig. 7 shows hyperparameter sequences for the final agents in the population for GAN and Machine Translation (MT) experiments. We have chosen these two examples as the community has developed very specific learning rate schedules for these models. In the GAN case, the typical schedule involves linearly annealed learning rates of both generator and discriminator in a synchronised way. As one can see, PBT discovered a similar scheme, however the discriminator annealing is much more severe, while the generator schedule is slightly less aggressive than the typical one.

As mentioned previously, the highly non-standard learning rate schedule used for machine translation has been to some extent replicated by PBT – but again it seems to be more aggressive than the hand crafted version. Quite interestingly, the dropout regime is also non-monotonic, and follows an even more complex path – it first increases by almost an order of magnitude, then drops back to the initial value, to slowly crawl back to an order of magnitude bigger value. These kind of schedules seem easy to discover by PBT, while at the same time are beyond the space of hyperparameter schedules considered by human experts.

Phylogenetic trees Figure 6 shows a phylogenetic forest for the population in the GAN and Atari experiments. A property made noticeable by such plots is that all final agents are descendants of the same, single initial agent. This is on one hand a consequence of the greedy nature of specific instance of PBT used, as `exploit` only uses a single other agent from the population. In such scenario, the number of original agents can only decrease over time, and so should finally converge to the situation observed. On the other hand one can notice that in both cases there are quite rich sub-populations which survived for a long time, showing that the PBT process considered multiple promising solutions before finally converging on a single winner – an exploratory capability most typical optimisers lack.

One can also note the following interesting qualitative differences between the two plots: in the GAN example, almost all members of the population enjoy an improvement in score relative to their parent; however, in the Atari experiment we note that even quite late in learning there can be exploration steps that result in a sharp drop in cumulative episode reward relative to the parent. This is perhaps indicative of some instability of neural network based RL on complex problems – an instability that PBT is well positioned to correct.

5 Conclusions

We have presented Population Based Training, which represents a practical way to augment the standard training of neural network models. We have shown consistent improvements in accuracy, training time and stability across a wide range of domains by being able to optimise over weights and hyperparameters jointly. It is important to note that contrary to conventional hyperparameter optimisation techniques, PBT discovers an adaptive schedule rather than a fixed set of hyperparameters. We already observed significant improvements on a wide range of challenging problems including state of the art models on deep reinforcement learning and hierarchical reinforcement learning, machine translation, and GANs. While there are many improvements and extensions to be explored going forward, we believe that the ability of PBT to enhance the optimisation process of new, unfamiliar models, to adapt to non-stationary learning problems, and to incorporate the optimisation of indirect performance metrics and auxiliary tasks, results in a powerful platform to propel future research.

Acknowledgments

We would like to thank Yaroslav Ganin, Mihaela Rosca, John Agapiou, Sasha Vehznevets, Vlad Firoiu, and the wider DeepMind team for many insightful discussions, ideas, and support.

References

Alekh Agarwal, John C Duchi, Peter L Bartlett, and Clement Levrard. Oracle inequalities for computationally budgeted model selection. In *Proceedings of the 24th Annual Conference on Learning Theory*, pp. 69–86, 2011.

Thomas Bäck. An overview of parameter control methods by self-adaptation in evolutionary algorithms. *Fundamenta Informaticae*, 35(1-4):51–66, 1998.

Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

PA Castillo, V Rivas, JJ Merelo, Jesús González, Alberto Prieto, and Gustavo Romero. G-prop-iii: Global optimization of multilayer perceptrons using an evolutionary algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pp. 942–942. Morgan Kaufmann Publishers Inc., 1999.

PA Castillo, MG Arenas, JG Castellano, JJ Merelo, A Prieto, V Rivas, and G Romero. Lamarckian evolution and the Baldwin effect in evolutionary neural networks. *arXiv preprint cs/0603004*, 2006.

Jeff Clune, Dusan Misevic, Charles Ofria, Richard E Lenski, Santiago F Elena, and Rafael Sanjuán. Natural selection fails to optimize mutation rates for long-term adaptation on rugged fitness landscapes. *PLoS Computational Biology*, 4(9):e1000187, 2008.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, pp. 3460–3468, 2015.

Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pp. 109–116. ACM, 2016.

Bartlomiej Gloger. Self adaptive evolutionary algorithms, 2004.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495. ACM, 2017.

Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. Batch Bayesian optimization via local penalization. In *Artificial Intelligence and Statistics*, pp. 648–657, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, pp. 5763–5773, 2017.

András György and Levente Kocsis. Efficient multi-start strategies for local search algorithms. *Journal of Artificial Intelligence Research*, 41:407–444, 2011.

Christopher R Houck, Jeffery A Joines, Michael G Kay, and James R Wilson. Empirical investigation of the benefits of partial Lamarckianism. *Evolutionary Computation*, 5(1):31–60, 1997.

Michael Husken, Jens E Gayko, and Bernhard Sendhoff. Optimization for problem classes-neural networks that learn to learn. In *Combinations of Evolutionary Computation and Neural Networks, 2000 IEEE Symposium on*, pp. 98–109. IEEE, 2000.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523, 2011.

Christian Igel, Stefan Wiegand, and Frauke Friedrichs. Evolutionary optimization of neural systems: The use of strategy adaptation. *Trends and Applications in Constructive Approximation*, pp. 103–123, 2005.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016.

Kim WC Ku and Man-Wai Mak. Exploring the effects of Lamarckian and Baldwinian learning in evolving recurrent neural networks. In *Evolutionary Computation, 1997., IEEE International Conference on*, pp. 617–621. IEEE, 1997.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Pierre-Yves Massé and Yann Ollivier. Speed learning on the fly. *arXiv preprint arXiv:1511.02540*, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *ICLR*, 2016.

Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. Hyperdrive: Exploring hyperparameters with POP scheduling. In *Proceedings of the 18th International Middleware Conference, Middleware*, volume 17, 2017.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

Mihaela Rosca, Balaji Lakshminarayanan, David Warde-Farley, and Shakir Mohamed. Variational approaches for auto-encoding generative adversarial networks. *arXiv preprint arXiv:1706.04987*, 2017.

Ashish Sabharwal, Horst Samulowitz, and Gerald Tesauro. Selecting near-optimal learners via incremental data allocation. In *AAAI*, pp. 2007–2015, 2016.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, pp. 2234–2242, 2016.

Rafal Salustowicz and Jürgen Schmidhuber. Probabilistic incremental program evolution: Stochastic search through program space. In *ECML*, pp. 213–220, 1997.

Amar Shah and Zoubin Ghahramani. Parallel predictive entropy search for batch global optimization of expensive objective functions. In *Advances in Neural Information Processing Systems*, pp. 3330–3338, 2015.

Leslie N Smith. Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pp. 464–472. IEEE, 2017.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable Bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pp. 2171–2180, 2015.

William M. Spears. Adapting crossover in evolutionary algorithms. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pp. 367–384. MIT Press, 1995.

Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust Bayesian neural networks. In *Advances in Neural Information Processing Systems*, pp. 4134–4142, 2016.

Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task Bayesian optimization. In *Advances in neural information processing systems*, pp. 2004–2012, 2013.

Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw Bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Bernard L Welch. The generalization of student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

Jian Wu and Peter Frazier. The parallel knowledge gradient method for batch Bayesian optimization. In *Advances in Neural Information Processing Systems*, pp. 3126–3134, 2016.

Bing Xue, Mengjie Zhang, Will N Browne, and Xin Yao. A survey on evolutionary computation approaches to feature selection. *IEEE Transactions on Evolutionary Computation*, 20(4):606–626, 2016.

Jianwei Yang, Anitha Kanna, Dhruv Batra, and Devi Parikh. LR-GAN: Layered recursive generative adversarial networks for image generation. In *ICLR*, 2017.

Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, pp. 4. ACM, 2015.

Jun Zhang, Zhi-hui Zhan, Ying Lin, Ni Chen, Yue-jiao Gong, Jing-hui Zhong, Henry SH Chung, Yun Li, and Yu-hui Shi. Evolutionary computation meets machine learning: A survey. *IEEE Computational Intelligence Magazine*, 6(4):68–75, 2011.

A Population Based Training of Neural Networks Appendix

A.1 Practical implementations

In this section, we describe at a high level how one might concretely implement PBT on a commodity training platform. Relative to a standard setup in which multiple hyperparameter configurations are searched in parallel (*e.g.* grid/random search), a practitioner need only add the ability for population members to read and write to a shared data-store (*e.g.* a key-value store, or a simple file-system). Augmented with such a setup, there are two main types of interaction required:

(1) Members of the population interact with this data-store to update their current performance and can also use it to query the recent performance of other population members.

(2) Population members periodically checkpoint themselves, and when they do so they write their performance to the shared data-store. In the exploit-and-explore process, another member of the population may elect to restore its weights from the checkpoint of another population member (*i.e.* exploiting), and would then modify the restored hyperparameters (*i.e.* exploring) before continuing to train.

As a final comment, although we present and recommend PBT as an asynchronous parallel algorithm, it could also be executed semi-serially or with partial synchrony. Although it would obviously result in longer training times than running fully asynchronously in parallel, one would still gain the performance-per-total-compute benefits of PBT relative to random-search. This execution mode may be attractive to researchers who wish to take advantage of commodity compute platforms at cheaper, but less reliable, preemptible/spot tiers. As an example of an approach with partial synchrony, one could imagine the following procedure: each member of the population is advanced in parallel (but without synchrony) for a fixed number of steps; once all (or some large fraction) of the population has advanced, one would call the exploit-and-explore procedure; and then repeat. This partial locking would be forgiving of different population members running at widely varying speeds due to preemption events, etc.

A.2 Detailed Results: RL

FuN - Atari

Table 1 shows the breakdown of results per game for FuN on the Atari domain, with and without PBT for different population sizes. The initial hyperparameter ranges for the population in both the FuN and PBT-FuN settings were the same as those used for the corresponding games in the original Feudal Networks paper (Vezhnevets et al., 2017), with the exception of intrinsic-reward cost (which was sampled in the range $[0.25, 1]$, rather than $[0, 1]$ – since we have found that very low intrinsic reward cost weights are rarely effective). We used fixed discounts of 0.95 in the worker and 0.99 in the manager since, for the games selected, these were shown in the original FuN paper to give good performance. The subset of games were themselves chosen as ones for which the hierarchical abstractions provided by FuN give particularly strong performance improvements over a flat A3C agent baseline. In these experiments, we used *Truncation* as the exploitation mechanism, and *Perturb* as the exploration mechanism. Training was performed for 2.5×10^8 agent-environmnent interactions, corresponding to 1×10^9 total environment steps (due to use of action-repeat of 4). Each experiment condition was repeated 3 times, and the numbers in the table reflect the mean-across-replicas of the single best performing population member at the end of the training run.

Game	**Pop = 10**		**Pop = 20**		**Pop = 40**		**Pop = 80**	
	FuN	**PBT-FuN**	**FuN**	**PBT-FuN**	**FuN**	**PBT-FuN**	**FuN**	**PBT-FuN**
amidar	1742 ±174	1511 ±400	1645 ±158	2454 ±160	1947 ±80	2752 ±81	2055 ±113	3018 ±269
gravitar	3970 ±145	4110 ±225	3884 ±362	4202 ±113	4018 ±180	4481 ±540	4279 ±161	5386 ±306
ms-pacman	5635 ±656	5509 ±589	5280 ±349	5714 ±347	6177 ±402	8124 ±392	6506 ±354	9001 ±711
enduro	1838 ±37	1958 ±81	1881 ±63	2213 ±6	2079 ±9	2228 ±77	2166 ±34	2292 ±16

Table 1: Per game results for Feudal Networks (FuN) with and without PBT for different population sizes (Pop).

UNREAL - DM Lab

Table 2 shows the breakdown on results per level for UNREAL and PBT-UNREAL on the DM Lab domain. The following three hyperparameters were randomly sampled for both methods and subsequently perturbed in the case of PBT-UNREAL: the learning rate was sampled in the log-uniform range [0.00001, 0.005], the entropy cost was sampled in the log-uniform range [0.0005, 0.01], and the unroll length was sampled uniformly between 5 and 50 steps. Other experimental details were replicated from Jaderberg et al. (2016), except for pixel control weight which was set to 0.1.

Game	**Pop = 40**		**Pop = 20**			
	UNREAL	**PBT-UNREAL**	**UNREAL**	**PBT-Hypers-UNREAL**	**PBT-Weights-UNREAL**	**PBT-UNREAL**
emstm_non_match	66.0	66.0	42.6	14.5	56.0	53.2
emstm_watermaze	39.0	36.7	27.9	33.2	34.6	36.8
lt_horse_shoe_color	75.4	90.2	77.4	44.2	56.3	87.5
nav_maze_random_goal_01	114.2	149.4	82.2	81.5	74.2	96.1
lt_hallway_slope	90.5	109.7				
nav_maze_all_random_01	59.6	78.1				
nav_maze_all_random_02	62.0	92.4				
nav_maze_all_random_03	92.7	115.3				
nav_maze_random_goal_02	145.4	173.3				
nav_maze_random_goal_03	127.3	153.5				
nav_maze_static_01	131.5	133.5				
nav_maze_static_02	194.5	220.2				
nav_maze_static_03	588.9	594.8				
seekavoid_arena_01	42.5	42.3				
stairway_to_melon_01	189.7	192.4				

Table 2: Per game results for UNREAL with and without PBT.

A3C - StarCraft II

Table 2 shows the breakdown on results per level for UNREAL on the DM Lab domain, with and without PBT for different population sizes. We take the baseline setup of Vinyals et al. (2017) with feed-forward network agents, and sample the learning rate from the log-uniform range [0, 0.001].

Game	**Pop = 30**	
	A3C	**PBT-A3C**
CollectMineralShards	93.7	100.5
FindAndDefeatZerglings	46.0	49.9
DefeatRoaches	121.8	131.5
DefeatZerglingsAndBanelings	94.6	124.7
CollectMineralsAndGas	3345.8	3345.3
BuildMarines	0.17	0.37

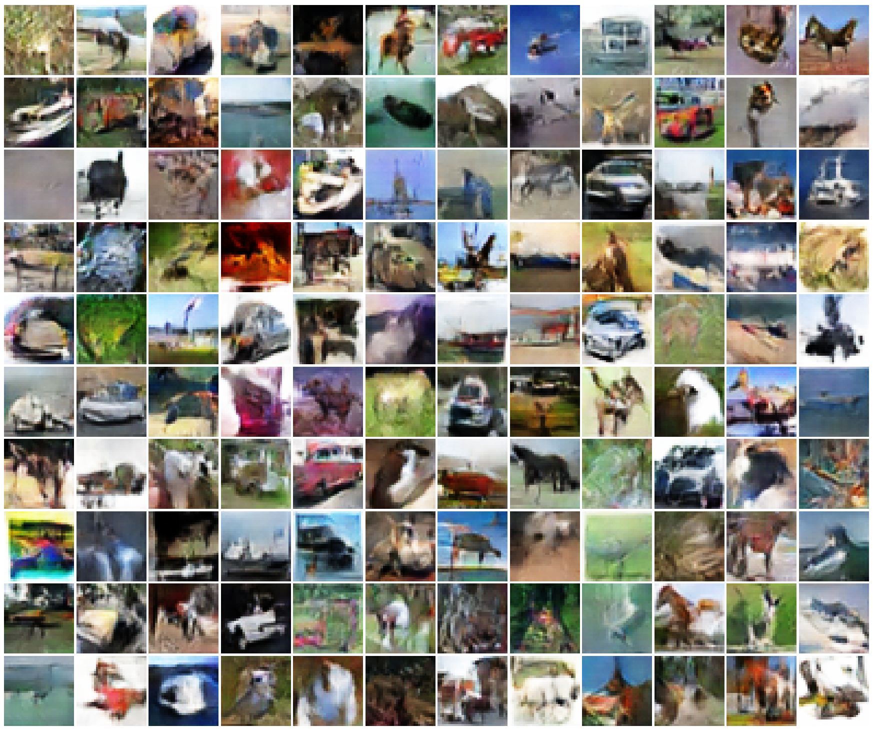
Table 3: Per game results for A3C on StarCraft II with and without PBT.

A.3 Detailed Results: GAN

This section provides additional details of and results from our GAN experiments introduced in Sect. 4.3. Fig. 8 shows samples generated by the best performing generators with and without PBT. In Table 4, we report GAN Inception scores with and without PBT.

Architecture and hyperparameters The architecture of the discriminator and the generator are similar to those proposed in DCGAN (Radford et al., 2016), but with 4×4 convolutions (rather than 5×5), and a smaller generator with half the number of feature maps in each layer. Batch normalisation is used in the generator, but not in the discriminator, as in Gulrajani et al. (2017). Optimisation is performed using the Adam optimiser (Kingma & Ba, 2015), with $\beta_1 = 0.5$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The batch size is 128.

Baseline selection To establish a strong and directly comparable baseline for GAN PBT, we extensively searched over and chose the best by CIFAR Inception score among many learning rate annealing strategies used in the literature. For each strategy, we trained for a total of $2N$ steps, where learning rates were fixed to their initial settings for the first N steps, then for the latter N steps either (a) exponentially decayed to 10^{-2} times the initial setting, or (b) linearly decayed to 0, searching over $N = \{0.5, 1, 2\} \times 10^5$. (Searching over the number of steps N is essential due to the instability of GAN training: performance often degrades with additional training.) In each training run to evaluate a given combination of annealing strategy and choice of training steps N, we used the same number of independent workers (45) and initialised the learning rates to the same values used in PBT. The best annealing strategy we found using this procedure was exponential decay with $N = 1 \times 10^5$ steps. For the final baseline, we then extended this strategy to train for the same


Baseline GAN

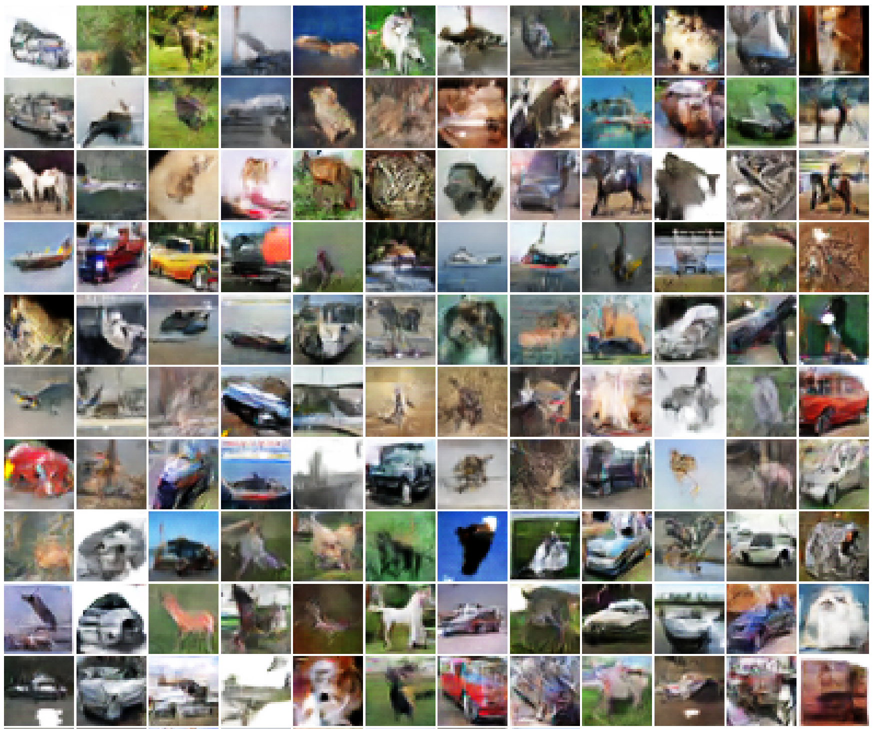

PBT-GAN

Figure 8: CIFAR GAN samples generated by the best performing generator of the Baseline GAN (top) and PBT-GAN (bottom).

	GAN	**PBT-GAN (TS)**	**PBT-GAN (BT)**
CIFAR Inception	6.38±0.03	6.80±0.01	6.77±0.02
ImageNet Inception (by CIFAR)	6.45±0.05	6.89±0.04	6.74±0.05
ImageNet Inception (peak)	6.48±0.05	7.02±0.07	6.81±0.04

Table 4: CIFAR and ImageNet Inception scores for GANs trained with and without PBT, using the *truncation selection* (TS) and *binary tournament* (BT) exploitation strategies. The *ImageNet Inception (by CIFAR)* row reports the ImageNet Inception score for the best generator (according to the CIFAR Inception score) after training is finished, while *ImageNet Inception (peak)* gives the best scores (according to the ImageNet Inception score) obtained by any generator at any point during training.

number of steps as used in PBT (10^6) by continuing training for the remaining 8×10^5 steps with constant learning rates fixed to their final values after exponential decay.

A.4 Detailed Results: Machine Translation

We used the open-sourced implementation of the Transformer framework[1] with the provided `transformer_base_single_gpu` architecture settings. This model has the same number of parameters as the base configuration that runs on 8 GPUs (6.5×10^7), but sees, in each training step, $\frac{1}{16}$ of the number of tokens (2048 vs. 8×4096) as it uses a smaller batch size. For both evolution and the random-search baseline, we searched over learning rate, and dropout values applied to the attention softmax activations, the outputs of the ReLU units in the feed-forward modules, and the output of each layer. We left other hyperparameters unchanged and made no other changes to the implementation. For evaluation, we computed BLEU score based on auto-regressive decoding with beam-search of width 4. We used *newstest2012* and *newstest2013* respectively as the evaluation set used by PBT, and as the test set for monitoring. At the end of training, we report tokenized BLEU score on *newstest2014* as computed by `multi-bleu.pl` script[2]. We also evaluated the original hyperparameter configuration trained for the same number of steps and obtained the BLEU score of 21.23, which is lower than both our baselines and PBT results.

[1] https://github.com/tensorflow/tensor2tensor

[2] https://github.com/moses-smt/mosesdecoder/blob/master/scripts/generic/multi-bleu.perl

Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks

In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2016)*. New York, NY: ACM
Nominated for Best Paper Award in Evolutionary Machine Learning.

Gregory Morse
Department of Computer Science
University of Central Florida
Orlando, FL 32816
gregorymorse07@gmail.com

Kenneth O. Stanley
Department of Computer Science
University of Central Florida
Orlando, FL 32816
kstanley@cs.ucf.edu

ABSTRACT

While evolutionary algorithms (EAs) have long offered an alternative approach to optimization, in recent years backpropagation through stochastic gradient descent (SGD) has come to dominate the fields of neural network optimization and deep learning. One hypothesis for the absence of EAs in deep learning is that modern neural networks have become so high dimensional that evolution with its inexact gradient cannot match the exact gradient calculations of backpropagation. Furthermore, the evaluation of a single individual in evolution on the big data sets now prevalent in deep learning would present a prohibitive obstacle towards efficient optimization. This paper challenges these views, suggesting that EAs can be made to run significantly faster than previously thought by evaluating individuals only on a small number of training examples per generation. Surprisingly, using this approach with only a simple EA (called the *limited evaluation EA* or LEEA) is competitive with the performance of the state-of-the-art SGD variant RMSProp on several benchmarks with neural networks with over 1,000 weights. More investigation is warranted, but these initial results suggest the possibility that EAs could be the first viable training alternative for deep learning outside of SGD, thereby opening up deep learning to all the tools of evolutionary computation.

CCS Concepts

•**Computing methodologies** → **Neural networks; Genetic algorithms;** Artificial intelligence; Supervised learning by regression;

Keywords

neural networks; deep learning; machine learning; artificial intelligence; pattern recognition and classification

GECCO '16, July 20–24, 2016, Denver, Colorado, USA
© 2016 ACM. ISBN 978-1-4503-4206-3/16/07. . . $15.00
DOI: http://dx.doi.org/10.1145/2908812.2908916

1. INTRODUCTION

Artificial neural networks (ANNs) have witnessed a renaissance in recent years within the field of machine learning through the rise of *deep learning* [6, 22, 27, 33]. The main ideas behind this new approach encompass a range of algorithms [5, 22], but a key unifying principle is that an ANN with multiple hidden layers (which make it deep) can encode increasingly complex features in its upper layers. Interestingly, ANNs were historically trained through a simple algorithm called *backpropagation* [40], which in effect applies *stochastic gradient descent* (SGD) or one of its variants to the weights of the ANN to reduce its overall error, but until about 2006 it was widely believed that backpropagation would lose its gradient in a deep network. Yet discoveries in the last few years have proven that in fact with sufficient training data and processing power backpropagation and SGD turn out to be surprisingly effective at optimizing massive ANNs with millions or more connections and many layers [8, 21, 27]. This realization has in turn led to substantive records being broken in many areas of machine learning through the application of backpropagation in deep learning [8, 21, 27], including unsupervised feature learning [4].

The success of a principle as simple as SGD in achieving record-breaking performance is perhaps surprising. After all, in a space of many dimensions, SGD should be susceptible to local optima, and unlike an evolutionary algorithm, all its eggs are essentially in a single basket because it works in effect with a population of one. Yet it turns out empirically that SGD is penetrating farther towards optimality in networks of thousands or millions of weights than any other approach. Attempting in part to explain this phenomenon, Dauphin et al. [11] make the intriguing argument that in fact very high-dimensional ANN weight spaces provide so many possible escape routes from any given point that local optima are actually highly unlikely. Instead, they suggest that the real stumbling blocks for SGD are *saddle points*, or areas of long, gradual error plateaus. This insight has both helped to explain why SGD might not get stuck, and also to improve it to move faster along such saddle points in the search space. There are also variants of SGD such as RMSProp [48] that help to extricate it from similar situations.

While such analyses may help to explain the success of SGD, they also raise an important question for evolutionary computation (EC): If there are indeed so many paths towards relative optimality in a high-dimensional ANN weight space, then why would not the very same benefits received from this scenario by SGD also apply to EC? In fact, maybe evolutionary algorithms (EAs) should even have an advantage. After all, a population is perhaps better suited than a single individual to a situation with many promising branches, and simple EAs are agnostic about the rate of descent with respect to the slope of the gradient, which could in principle avoid the kind of saddle-point problem contemplated by Dauphin et al. [11]. In short, the arguments for why SGD can succeed in extremely high-dimensional spaces seem on the face of it to support or even favor EAs as well.

However, because the population in an EA is in effect an approximation of the gradient, it may seem that EAs could be significantly disadvantaged by the fact that they do not compute *exact* gradients, which is precisely what SGD does. However, results have been reported to suggest that the exactitude of the gradient in SGD is not the crux of its success. For example, recalling the application of mutation in EAs, Lillicrap et al. [31] report the surprising discovery that the error signal in a deep network can be multiplied by *random synaptic weights* (entirely breaking the precision of the gradient computation) with little detriment to learning. This result suggests that in fact there are so many viable paths in the high-dimensional space that exactitude is *not* the key causal explanation for reaching near-optimality. Furthermore, given that any search space can be deceptive [18, 49], it is likely often the case that the steepest descent at any given point is not on the shortest path to the optimum anyway. Perhaps it would be even better to maintain a population of options to avoid any premature committal to the best-looking path of the moment.

In fact, the smooth application of SGD in deep learning remains a work in progress. Many tricks have been developed to improve its performance, such as the introduction of rectified linear units (ReLUs) for activation functions, which improves the passing of the gradient from layer to layer over sigmoidal units [38]. Yet even then, researchers continue to observe challenges with finding the right parameters to make such structures learn smoothly, leading to complicated work-arounds like interpolating between different architectures over the course of learning [1] and the recent *highway networks* architecture of Srivastava et al. [43] that in effect turns some neurons on and off over the course of learning, which is reminiscent of evolutionary algorithms that learn structure like NEAT [44]. Thus there remains ample room for new approaches, yet few have considered that such new approaches might come from *outside* SGD.

Perhaps one reason simple EAs have not yet been applied widely to optimizing the weights of deep networks is that most problems in deep learning encompass a large number of training examples. Just as an example, the MNIST image classification database [28] includes 60,000 training examples. While SGD can cycle through these examples on its single learner and adjust its weights based on every individual example, in an EA every individual in the population at every generation must seemingly be evaluated on all the examples to assess its fitness on the training set. Thus a single generation of e.g. 100 individuals would process *six million* examples only to facilitate a single step of the search algorithm.

However, the algorithm introduced in this paper, called the *limited evaluation evolutionary algorithm* (LEEA), is based on a novel insight into the analogy between EAs and SGD that implies that in fact just as an iteration of SGD does not necessitate passing through the entire training set, *neither does a generation of an EA*. Instead, consider that if SGD can compute an error gradient from a single instance (or a small batch of them) then a generation of evolution can be tasked with doing precisely the same. That is, a generation of the EA can be considered analogous to a single iteration of SGD, aiming merely to adjust the weights of the best current approximation(s) to improve with respect to a single instance or small set of them. In this view, the population of 100 might only need to process 100 instances in one generation (instead of six million), which with simple parallelization could in principle be done in the same time it takes to process a single example (and no backpropagation of error need be computed either). Thus the EA begins to look computationally comparable to SGD.

Experiments in this paper on high-dimensional optimization of ANNs will indeed reveal the surprising conclusion that a simple EA appears about as effective as backpropagation through state-of-the-art SGD in problems of over 1,000 dimensions. The competitive performance of the EA in these problems suggests that further research in higher-dimensional neural network optimization is warranted because of the potential for an alternative training strategy in deep learning. This possibility is not just about leveling the playing field with SGD. Rather, it is exciting because EAs bring with them an entirely new toolbox that suddenly becomes applicable to the field of deep learning. Whereas in deep learning researchers apply tricks like regularization for sparsity [16] or dropout [42], EAs have distinctly different options completely unavailable to SGD such as architecture evolution like in NEAT [44], diversity maintenance techniques like novelty search [29], or indirect encodings for ANNs like in HyperNEAT [15, 47]. Thus the entrance of EAs as an alternative to SGD in deep learning would carry with it a broad set of new possibilities.

2. BACKGROUND

The application of EAs to optimizing neural networks is often called *neuroevolution* by its practitioners [12, 44]. As documented in reviews such as by Yao [53] and later Floreano et al. [12], the field of neuroevolution originated in the 1980s (e.g. Montana and Davis [34]), at a time when backpropagation was on the rise [40]. Interest in neuroevolution really picked up in the 1990s, during which a wide variety of approaches were introduced [53]. In these early years, many researchers focused on the intriguing idea (now for the most part long abandoned) that evolution might in fact exceed the capabilities of backpropagation.

In fact, a number of early studies showcase neuroevolution through a variety of EAs matching [51] or even outperforming backpropagation in classification problems [17, 34, 39]. In fact, in these early years enthusiasm was high in part because the future potential of evolving topology along with connection weights seemed to some a significant possible advantage for neuroevolution. As Mühlenbein [37] put it, "We conjecture that the next generation of neural networks will be genetic neural networks which evolve their structure." Many researchers echoed this enthusiasm [7, 10]. Others touted the potential for combining topology evolution with backpropagation [3, 50].

However, as computational capabilities increased over the ensuing decade, researchers began to recognize that the apparent advantages of neuroevolution on relatively simple, low-dimensional problems (i.e. requiring relatively small networks) with small amounts of data might be eclipsed as the amount of data and size the neural networks increases. For example, even before deep learning really began to showcase the power of SGD with big data, Mandischer [32] begins to articulate the more negative outlook (focusing on Evolution Strategies, or ESs, which are a kind of EA): "We will see that ESs can only compete with gradient-based search in the case of small problems and that ESs are good for training neural networks with a non-differentiable activation function." (It is important to note of course that researchers at the time had not tried the idea in the present paper of only evaluating on a very small number of instances per generation.)

As SGD and backpropagation increasingly dominated the world of classification, especially after the advent of deep learning [5, 22], a significant shift in attention away from classification took hold in the neuroevolution literature. Researchers began to focus on types of problems where backpropagation is more challenging to apply, such as reinforcement learning problems requiring recurrent connections and specialized architectures [2, 13, 20, 35]. As the focus in neuroevolution largely shifted towards reinforcement learning and away from classification, a new generation of neuroevolution algorithms such as NEAT [44, 46], HyperNEAT [15, 47], and a modified CMA-ES [25] gained popularity in part by focusing on the daunting challenge of finding the right architecture for complicated control and decision-making problems, for which SGD provides little answer. Thus the aspiration of EAs to compete directly with SGD in training neural networks for state-of-the-art classification and supervised learning has gradually become only a memory.

Nevertheless, the classic results from the 1990s where neuroevolution does outperform SGD on simple supervised problems [17, 34, 39] remain an intriguing prelude to the ensuing decades of SGD dominance in supervised learning. Despite the seeming clarity of SGD's subsequent dominance in deep learning, the question of why the early promising results of neuroevolution so dramatically fizzled out is actually not well understood. It may seem that neuroevolution is simply definitively not suited to high-dimensional optimization (even with statistical approaches such as in CMA-ES [25] or EDAs [26], which still do not compute exact error gradients), but the support for such a hypothesis is largely speculative because we do not fully understand how or why the structure of neural network search spaces is necessarily vastly more favorable to SGD, which faces its own perils with local optima and saddle points [11]. In short, why should an approximation of the gradient (provided by an EA's population) be any less useful than the single exact gradient computation of a single individual in SGD, which is subject to deception? Both must be imperfect, but given that SGD still succeeds despite the danger of deception, the high-dimensional landscape of neural optimization appears to offer a forgiving landscape of many viable paths, which might be similarly favorable to evolution. This paper therefore revives the old hope from the 1990s that even simple neuroevolution can compete with SGD.

3. APPROACH

A key property of SGD is that the gradient does not have to be calculated for the network over the entire training set. Instead, the gradient can be calculated for a single or very few training examples at a time, which greatly reduces computational cost compared to training on all training examples at once and reduces the chance of becoming stuck in local optima. Interestingly, this ability to adjust weights after very few examples is not necessarily exclusive to gradient descent. Rather, it can in principle apply to any algorithm that traditionally evaluates its solutions against an entire training set, such as EAs.

LEEA implements this idea for the first time in a very simple traditional generational EA. Instead of evaluating the population against the entire training set each generation, the population is evaluated against only a limited number of training examples each generation. This lean approach to evaluation greatly relieves the computational load, particularly on large training sets.

However, one potential weakness of LEEA is that performance on a single example may not correspond to performance across all examples. In SGD, this problem of deceptive examples is tempered by the learning rate and the fact that the population of in effect one individual is never "replaced" by a defective mutant, which prevents the network from shifting too far towards a globally poor configuration during any given evaluation. In contrast, an EA does not inherently contain such safeguards against such deceptive training examples. For example, in the EA, a species well suited to only the present example could displace one that had mastered all the examples before. Two strategies in LEEA mitigate this problem. The first is simple: the population is evaluated against more than one example per generation, though still very few. This strategy reduces the potential damage caused to the population by a single deceptive training example.

It should be noted that while the technique of evaluating the population against a small set of examples each generation may appear to be analogous to a technique employed by SGD called "mini-batching" [9], its motivation in LEEA is different. SGD employs mini-batches primarily to better utilize parallel computational resources, while LEEA employs these mini-batches for more stable population dynamics.

The way the examples are selected for each mini-batch naturally can influence the effectiveness of the algorithm. If all of the examples selected for a given mini-batch happen to have a similar expected output, then even degenerate networks that output a constant value may achieve a high fitness. Instead, if the examples are selected so that they have a diversity of expected outputs, then networks will only be highly rewarded when they can also produce a diversity of outputs that match the expected values. This approach thereby prevents degenerate networks from ever achieving a high fitness and rewards networks that exhibit heterogeneous behavior.

Even with carefully selected mini-batches, LEEA might still lose individuals who are relatively fit in a global sense, but weak on the examples encountered during any single mini-batch. This danger is particularly acute during early evolution, when the best individuals may only succeed on a small percentage of all examples. To further combat this complication, the second key strategy in LEEA is that fitness is calculated based on both the performance on the current mini-batch and the performance of the individual's *ancestors* against their mini-batches. As long as each step of evolution is not changing the behavior of each network in a radical way, this *fitness inheritance* builds up for those individuals who are more likely to be more globally fit than their peers, regardless of how they performed on the current mini-batch.

It is important to note that this form of fitness inheritance is inspired by though differs from previous approaches to fitness inheritance that aimed to avoid directly evaluating a portion of the population [14, 41]. To implement fitness inheritance in LEEA, the fitness for individuals produced by sexual reproduction and asexual reproduction, respectively, is given by

$$f' = \frac{f_{p_1} + f_{p_2}}{2}(1-d) + f, \text{and} \tag{1}$$

$$f' = f_{p_1}(1-d) + f, \tag{2}$$

where f' is the individual's modified fitness, f is the fitness of the individual against the current mini-batch, f_{p_n} is the fitness of parent n, and d is a constant decay value.

The introduction of output-diverse mini-batching and fitness inheritance enables LEEA to take advantage of the computational benefits of SGD within the framework of evolutionary computation. Other than these two strategies, LEEA is just a simple generational EA with a mutation power decay (which is analogous to learning rate decay in SGD):

Algorithm 1 LEEA

```
1: while $gen < maxGenerations$ do
2:     select mini-batch of training examples
3:     evaluate population
4:     modify fitnesses based on fitness inheritance
5:     select parent(s) from a truncated list with roulette
       wheel selection
6:     create offspring – sexual reproduction with uniform
       crossover (without mutation) or asexual reproduc-
       tion with uniformly distributed mutation
7:     reduce maximum mutation power by multiplying by
       decay constant
8:     $gen = gen + 1$
9: end while
```

This algorithm contains only the bare essentials required for a more traditional EA to work, along with the modifications necessary to combat the complications caused by limited evaluations per generation.

4. EXPERIMENT

To assess the effectiveness of LEEA as an alternative to more traditional neural network optimization methods, performance is tested in three domains against a traditional generational EA (TGEA), SGD, and RMSProp, which is a cutting-edge variant of gradient descent based on the idea that following shallow gradients can often be as useful as following steep ones [48]. The TGEA works like LEEA, but with *all* training examples evaluated in each generation (instead of a mini-batch), and no fitness inheritance mechanism. In RMSProp, the enhancement to SGD, the current gradient information is divided by a running average of recent gradients. This technique allows the algorithm to escape from plateaus with tiny gradients more quickly. All three domains are chosen because they can benefit from a neural network of moderate dimensionality, i.e. over 1,000 connection weights that must be optimized. In general it is not a common view that a simple EA is suited to optimizing so many parameters in a neural network as effectively as SGD. Therefore these experiments demand an unusual ability for EAs that is rarely contemplated in deep learning literature.

Sample data from each domain is divided into a training set, a validation set, and a test set. All algorithms evaluate performance against the validation set to test for overfitting. The evolution-based algorithms additionally require this validation performance data to determine which individual from the population is selected for evaluation against the test set. All reported results are thus based on performance against the test set and for evolution the individual tested is the one that performed best against the validation set.

The first domain, with 800 training examples, is a function approximation task, where the function is given by the equation

$$f(x,y) = \frac{sin(5x(3y+1)) + 1}{2}. \tag{3}$$

Preliminary tests indicate that the surface generated by this function is sufficiently difficult to approximate that networks with over 1,000 connections have an advantage over smaller networks, thus making a good test for the effective use of high dimensions.

The second domain is a time series prediction task with 1,200 training examples where the time series is generated using the Mackey-Glass chaotic time series equation

$$\frac{dx}{dt} = \beta \frac{x_r}{1 + {x_r}^n} - \gamma x, \quad \gamma, \beta, n > 0 \tag{4}$$

where $\gamma = 1$, $\beta = 2$, $r = 2$, $n = 9.65$, $x_0 = 1.1$, and $x_1 = 1.2$. The prediction for time t is based on values at $t-6$, $t-12$, $t-18$, and $t-24$. This task has been employed to test neural networks in the past [24], and is also sufficiently complex to potentially benefit from over 1,000 connections.

The third domain is the California housing dataset, which is a housing value prediction task with 10,000 training examples also previously given to neural networks [24, 30]. This task contains observations of housing data where each observation consists of eight input attributes for a particular block of housing (median income, median house age, etc.) and one output for the median house value.

All learning algorithms train with the same network topology, which contains two hidden layers with 50 nodes in the first layer and 20 nodes in the second layer, as well as a single output. Because of differences in the number of inputs, the resultant networks have 1,170, 1,270, and 1,470 connections for the first, second, and third domains, respectively (networks include also a bias node). LEEA and TGEA are both evolved using a direct encoding (i.e. one floating-point gene per connection in the network) with no ability to change the network topology.

To evaluate the effectiveness of each algorithm, they are exposed to the same total number of examples during training. That way, the number of generations given to TGEA equals the number of epochs given to SGD and RMSProp, while LEEA is given a proportionately higher number of generations (which of course still means the same number of evaluations) than TGEA based on the ratio of total training examples to the number of examples evaluated per generation. Because the EAs must evaluate all members of their

	Function Approximation	**Time Series**	**California Housing**
TGEA	0.1643 ± 0.0121	0.2068 ± 0.0107	0.1216 ± 0.0013
LEEA	0.0711 ± 0.0116	$\mathbf{0.1080 \pm 0.0340}$	0.1147 ± 0.0020
SGD	$\mathbf{0.0539 \pm 0.0087}$	0.1336 ± 0.0380	0.1097 ± 0.0016
RMSProp	0.0636 ± 0.0138	0.1227 ± 0.0410	$\mathbf{0.1092 \pm 0.0007}$

Table 1: RMSE with standard deviation for each algorithm on each domain averaged over 10 runs. (Lower is better.)

population, on a single processor their execution time becomes greater proportionally to the population size. However, parallelization can potentially benefit EAs to a greater degree because each such evaluation is independent. As the capacity for parallelization increases, the instrumental issue thus shifts from population size to the ability to effectively learn from each training example, which is why performance is measured here based on the number of examples evaluated.

Parameters for all algorithms were selected through a parameter sweep on the first domain. There were two key differences found between the ideal parameters of LEEA compared to TGEA. For LEEA, the ideal starting mutation power (the maximum size of a weight mutation) is 0.03, compared to 0.1 for TGEA. This difference reflects that it is better to take smaller steps when evaluating individuals on a small set of training examples. In addition, while TGEA performs best with fitness sharing-based speciation [19] in the spirit of NEAT [44] based on genetic similarity (which maintains diversity), LEEA performs best without any speciation. This observation makes sense because the changing mini-batch in each generation is a force for diversity on its own.

Parameters in common between LEEA and TGEA are population size (1,000), mutation power decay (0.99), mutation rate (0.04), selection proportion (0.4), and sexual reproduction proportion (0.5). Selection proportion refers to the ratio of individuals (sorted by fitness) that are eligible for reproduction.

Ideal mini-batch size in LEEA was determined experimentally to be 2. In all three domains, there is no notion of output classes, but rather a single real number output in the range $\{0, 1\}$. To apply the idea of diversity in mini-batches, the output range is divided into two equally sized sections, and examples selected so that each mini-batch contains one example for both sections of the range. A new mini-batch is randomly generated at the beginning of each generation (while ensuring the diversity of the expected outputs). The ideal fitness inheritance decay rate for LEEA was determined experimentally to be $d = 0.2$.

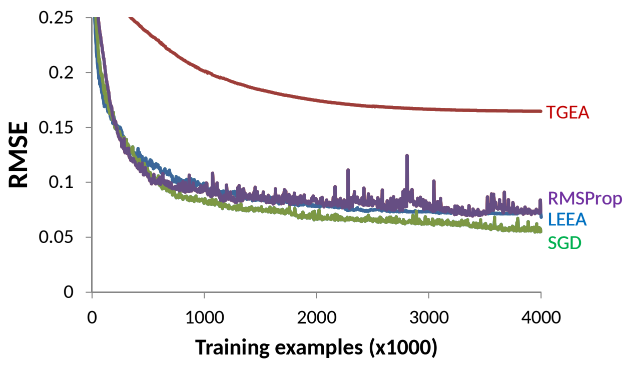
SGD and RMSProp were implemented as online learning (i.e. one example at a time). Mini-batching with these methods is not included in the comparison because there is not a consensus on an empirical advantage for mini-batching on a per-epoch basis, and some have even suggested perhaps a slight disadvantage [52]. Therefore, the online case serves as a more transparent baseline that avoids introducing any confounding factors that might come with mini-batching. The training examples are reshuffled at the start of each epoch. The initial learning rate for SGD and RMSProp is 0.3 and 0.001, respectively. The learning rate is decayed exponentially by 0.0001 per epoch for both SGD and RMSProp. RMSProp maintains a queue of 250 recent magnitudes for each parameter for calculating the per-parameter learning rate.

All neurons in every setup are sigmoidal based on the function $f(x) = \frac{1}{1+e^{-x}}$. That way, all methods are compared in equivalent conditions. Of course it is known that other functions like rectified linear units [38] can sometimes help deep networks under SGD, but the aim here is to establish how these methods behave under equivalent controlled conditions to get a sense of their relative capabilities in general. Source code for the experiments in this paper can be found at http://eplex.cs.ucf.edu/uncategorised/software.

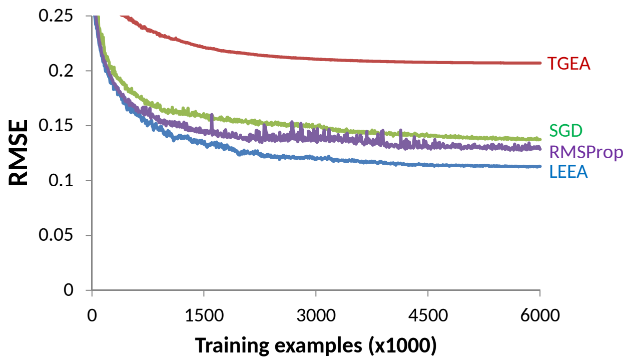
5. RESULTS

Table 1 shows testing results for each algorithm on each domain. Overall test performance in the table is measured as the root mean square error (RMSE) of the individual who performs best against the validation set across the run, where individuals are tested against the validation set at regular intervals based on the number of training examples in the domain (every 4,000 samples for function approximation, every 6,000 samples for the time series, and every 30,000 samples for California Housing). The individual tested in the evolutionary runs is the one who performs best on validation from the population at the current generation. It is important to note that overall, each algorithm experiences precisely the same number of examples before each validation check, so the amount of training data experienced is always equivalent. Test performance is averaged over 10 runs for each algorithm/domain combination. Some results did not exhibit a normal distribution, so significance between results is calculated with the Mann-Whitney U test. On the function approximation domain, LEEA's performance is not statistically significantly different from RMSProp, but it performs slightly though significantly worse than SGD ($p < 0.01$) and significantly better than TGEA ($p < 0.01$). On the time series domain, though LEEA performs best by a small margin, there is no statistical difference between LEEA, SGD, and RMSProp, and all three algorithms perform significantly better than TGEA ($p < 0.01$). On the California housing domain, LEEA performs slightly though significantly worse than SGD and RMSProp ($p < 0.01$) and significantly better than TGEA ($p < 0.01$). As a whole, comparing results between LEEA and TGEA in all the domains confirms that EA performance can be significantly enhanced by evaluating a limited number of examples per generation. They also show that differences between LEEA, SGD, and RMSProp range from very small to insignificant, with LEEA even performing best (though insignificantly) in one domain, supporting the conclusion that LEEA is a viable option for training neural networks.

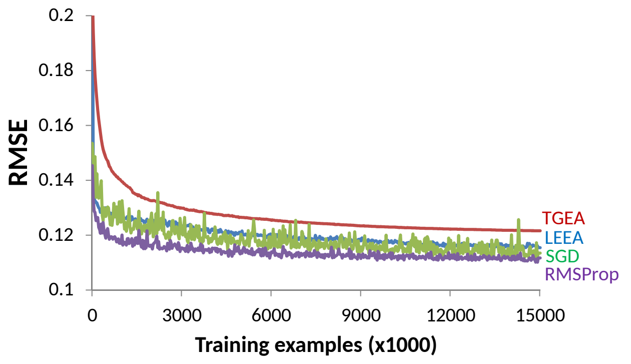
To provide further insight into these results, figure 1 shows average testing accuracy across whole runs, with the x-axis normalized such that all algorithms are evaluated against an



(a) Function approximation



(b) Time series



(c) California housing

Figure 1: Average root mean square error on the test set over time (lower is better). The average performance over total training examples seen so far as measured against the test set is displayed for each algorithm on the (a) function approximation, (b) time series, and (c) California housing domains. The main result is that the performance of LEEA closely matches that of SGD and RMSProp.

equal number of training examples on the same problem at the same point along x. For the evolution-based algorithms, the whole population is evaluated against the validation set at each measurement interval and the individual with the best validation performance is selected for testing. While TGEA learns more slowly and converges to a less optimal solution than SGD and RMSProp, LEEA exhibits a qualitatively similar training curve to these conventional deep learning training algorithms. This observation provides further evidence that LEEA offers an evolution-based alternative for training complex neural networks that is potentially competitive with the state of the art.

6. DISCUSSION AND CONCLUSION

The results suggest that a simple EA with limited evaluations in each generation (the LEEA) can optimize neural networks of over 1,000 dimensions about as effectively and in about the same number of iterations as gradient descent algorithms that currently dominate the field of deep learning. While these results do not prove that such EAs will continue to work well on the neural networks of millions or more weights that now feature in the most cutting-edge results in deep learning [21], they are an intriguing hint that the potential for EAs in this area may be greater than previously believed. At the least it suggests that investigation of such algorithms on larger networks is warranted.

The core question on much larger networks is whether somehow the gradient over the high number of dimensions becomes too hard to approximate for the EA. For example, if such high-dimensional optimization requires effectively sampling changes along nearly every dimension, the EA population might fail to sample densely enough. Yet if it is true that there are many paths to near-optimality as has been argued even in deep learning literature [11], it may not ultimately be essential to sample even a large proportion of the possible paths. In fact, sparse sampling could be a winning trade-off as the price to gain diversity, which conventional SGD lacks. In any case, only further empirical investigation on more complex domains such as MNIST [28] can settle this question.

Interestingly, even if sampling is ultimately too sparse, there is always the option of expanding the population size. Continuing advances in hardware and the availability of graphics processing units (GPUs) foreshadow the possibility of parallelizing much larger populations in the future. While GPUs have recently been celebrated for their parallelism largely in the context of SGD within deep learning, EAs where individuals can be evaluated separately are particularly suited to large-scale parallelization. In the future, GPUs and EAs could present a particularly powerful marriage. Furthermore, performance might be improved in the future also by refining the method for computing fitness inheritance.

The potential for EAs to offer an alternative to SGD for training neural networks is compelling because EAs are a genuinely different paradigm for specifying a search problem. That is, the real payoff of such a novel option is not that it might perform better in some scenario, but that it allows researchers to approach problems in entirely different ways and capitalize on different forms of regularization, thereby greatly expanding the toolbox available to neural network researchers.

For example, expressing the loss function for SGD to target precisely the desired behavior can be harder than for a fitness function. Consider a two-class classification problem where a factory product is either *working* or *defective*. While it is relatively straightforward to configure SGD to minimize error on the classification, it would be much harder to minimize *monetary cost* for making different kinds of mistakes: misclassifying a defective product as working might be a lot more expensive than misclassifying a working product as defective. The advantage of the fitness function is that it can *directly* minimize even an indirect cost because it is intrinsically more expressive.

Furthermore, EAs offer options unavailable to SGD like diversity maintenance [36], the evolution of topology [44], and indirect encoding [15, 45]. These can all act as powerful regularizers different from those in SGD. Problems facing SGD in backpropagation like vanishing gradients through multiple layers or through recurrence also would not even exist for neuroevolution through EAs, perhaps enabling radically different architectures to be learned than the ones designed for SGD like LSTMs [23], or even autoencoders [4].

The possibility of such a dramatically different training paradigm is intriguing, and the empirical evidence in this paper offers a hint that it is at least sufficiently conceivable to warrant further serious investigation. The properties of search spaces with millions of dimensions are still not fully understood, leaving open the chance that something quite different than backpropagation can play a constructive role in such spaces as well.

Acknowledgments

This work was supported by the National Science Foundation under grant no. IIS-1421925. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Special thanks to the UCF Stokes cluster for providing computation resources.

References

[1] D. Almeida and N. Sauder. GradNets: Dynamic interpolation between neural architectures. *ArXiv e-prints*, abs/1511.06827, 2015.

[2] P. J. Angeline, G. M. Saunders, and J. B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[3] R. K. Belew, J. McInerney, and N. N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In *Artificial Life II: Proceedings of the Workshop on Artificial Life*, 1990.

[4] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1): 1–127, 2009.

[5] Y. Bengio and Y. LeCun. Scaling learning algorithms towards ai. *Large-Scale Kernel Machines*, 34, 2007.

[6] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems 19 (NIPS)*, Cambridge, MA, 2007. MIT Press.

[7] H. Braun and J. Weisbrod. Evolving neural feedforward networks. In *Artificial Neural Nets and Genetic Algorithms*, pages 25–32. Springer, 1993.

[8] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

[9] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *Advances in neural information processing systems*, pages 1647–1655, 2011.

[10] D. Dasgupta and D. McGregor. Designing application-specific neural networks using the structured genetic algorithm. In *Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, pages 87–96, 1992.

[11] Y. Dauphin, R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *ArXiv e-prints*, abs/1406.2572, 2014.

[12] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62, 2008.

[13] D. B. Fogel. *Blondie24: Playing at the Edge of AI.* 2001.

[14] L. G. Fonseca, A. C. Lemonge, and H. J. Barbosa. A study on fitness inheritance for enhanced efficiency in real-coded genetic algorithms. In *Proceedings of the 2012 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2012.

[15] J. Gauci and K. O. Stanley. Autonomous evolution of topographic regularities in artificial neural networks. *Neural Computation*, 22(7):1860–1898, 2010.

[16] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[17] C. Goerick and T. Rodemann. Evolution strategies: An alternative to gradient-based learning. In *Proceedings of the International Conference on Engineering Applications of Neural Networks*, volume 1, pages 479–482. Citeseer, 1996.

[18] D. E. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. In L. Davis, editor, *Genetic algorithms and simulated annealing*, pages 74–88, London, 1987. Pitman.

[19] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum, 1987.

[20] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5: 317–342, 1997.

[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[22] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[23] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] G.-B. Huang, P. Saratchandran, and N. Sundararajan. A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation. *Neural Networks, IEEE Transactions on*, 16(1):57–67, 2005.

[25] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation*, pages 2588–2595, Piscataway, NJ, 2003. IEEE Press.

[26] P. Larranaga and J. A. Lozano. *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media, 2002.

[27] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning (ICML-2012)*, 2012.

[28] Y. LeCun and C. Cortes. The MNIST database of handwritten digits, 1998.

[29] J. Lehman and K. O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223.

[30] N.-Y. Liang, G.-B. Huang, P. Saratchandran, and N. Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *Neural Networks, IEEE Transactions on*, 17 (6):1411–1423, 2006.

[31] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014.

[32] M. Mandischer. A comparison of evolution strategies and backpropagation for neural network training. *Neurocomputing*, 42(1):87–117, 2002.

[33] R. Marc'Aurelio, L. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. In *Advances in Neural Information Processing Systems 20 (NIPS)*, pages 1185–1192, Cambridge, MA, 2007. MIT Press.

[34] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. pages 762–767, 1989.

[35] D. E. Moriarty and R. Miikkulainen. Forming neural networks through efficient and adaptive co-evolution. *Evolutionary Computation*, 5:373–399, 1997.

[36] J.-B. Mouret and S. Doncieux. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 20(1):91–133, 2012.

[37] H. Mühlenbein. Limitations of multi-layer perceptron networks – steps towards genetic neural networks. *Parallel Computing*, 14(3):249–260, 1990.

[38] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[39] V. W. Porto, D. B. Fogel, and L. J. Fogel. Alternative neural network training methods. *IEEE Intelligent Systems*, (3):16–22, 1995.

[40] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, pages 318–362. 1986.

[41] R. E. Smith, B. A. Dike, and S. A. Stegmann. Fitness inheritance in genetic algorithms. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, SAC '95, 1995.

[42] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[43] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *ArXiv e-prints*, abs/1505.00387, 2015.

[44] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

[45] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[46] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research (JAIR)*, 21: 63–100, 2004.

[47] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 15(2): 185–212, 2009.

[48] T. Tieleman and G. Hinton. Lecture 6.5-RMSprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[49] D. Whitley. Fundamental principles of deception. *Foundations of Genetic Algorithms (FOGA 1)*, 1:221, 1991.

[50] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14: 347–361, 1990.

[51] W. Wienholt. Minimizing the system error in feedforward neural networks with evolution strategy. In *ICANN 93*, pages 490–493. Springer, 1993.

[52] D. R. Wilson and T. R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

[53] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.