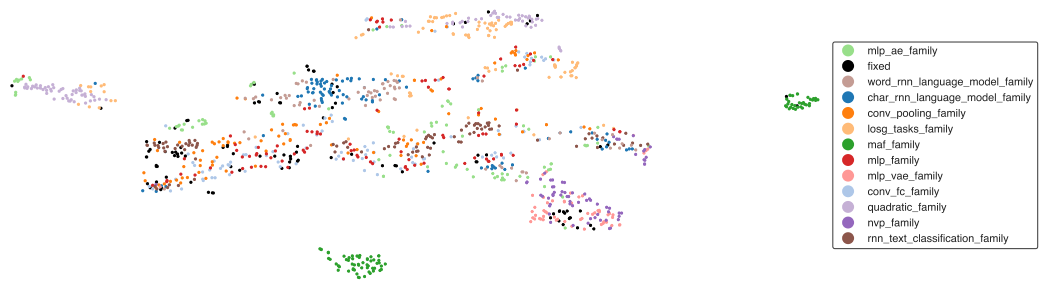

Figure S1: A 2D TSNE embedding of all 1162 tasks. This embedding is produced from a 1,000 dimensional feature vector consisting of task loss evaluated with many different hyperparameter configurations. We find similar tasks – e.g. masked auto regressive flow models, and character / word RNN models – cluster, suggesting similarity in the optimizers that perform well. See §**??** for more details.

A TaskSet Visualization

For a qualitative view, we constructed a feature space consisting of performance measurements for each task+optimizer pair (See §3.3). This forms a dense matrix of size number of tasks by number of optimizers. We then perform T-SNE [73, 115] to reduce the dimensionality to two and plot the results coloring by task family (Figure S1). Clusters in this space correspond to tasks that work well with similar optimizers. We find diversity of tasks with clusters occurring around similar families of tasks.

A.1 TSNE of TaskSet

B Additional Experiments

B.1 Generalization to different sized problems

Training learned algorithms on large models is often infeasible for computational reasons. As such, one form of generalization needed when building learned algorithms is the ability to transfer to different sized models. As shown in Figure 1 the tasks in this suite contain a wide range of parameter counts, and can thus be used to test this kind of generalization. We split the tasks into 8 groups – one group per order of magnitude in parameter count, and train hyperparameter lists on one range and test on the rest. In Figure S2 we plot the fraction of the training loss achieved by the test loss on the target parameter range. We find peak performance around the model sizes used for training, and smooth falloff as the testing tasks become more dissimilar as measured by parameter count. We note that our problems are not evenly distributed across these groups thus each group will contain a different percentage of the underlying tasks. While this potentially confounds these results, we believe a similar bias occurs in realistic workloads as well.

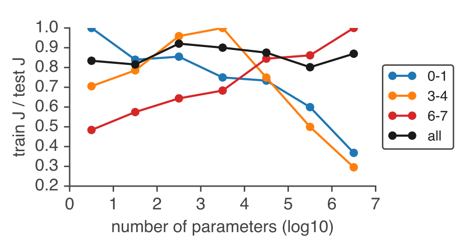

Figure S2: We show learned search space generalization, measured as a ratio of the loss achieved in training and testing, versus the number of task parameters used during search space training. Generalization falls off as one moves further away from the training regime. In black we show that a uniform mixture of the 7 parameter buckets does not fall off.

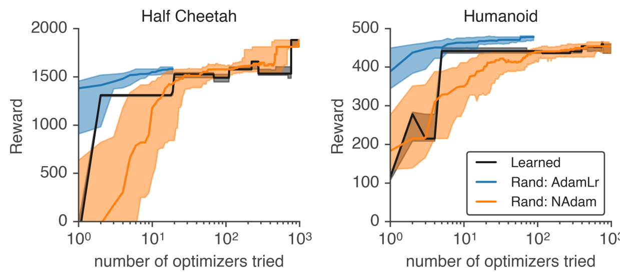

Figure S3: We find our learned hyperparameter lists performs about as well as random search on the NAdam search space, and worse than the random search on the learning rate tuned Adam search space.

B.2 Reinforcement Learning with PPO

We test the learned hyperparameter lists on two continuous control reinforcement learning environments, half cheetah and humanoid, from Gym's Mujoco environments[113, 20]. We use TF-Agents [45] with all non-optimizer hyperparameters set via searching a mixture of environments. In figure B.2 we find our learned hyperparameter lists achieves comparable to slightly worse performance does not out perform learning rate tuning of Adam in both efficiency nor final performance. To diagnose this behavior we ran all 1k optimizers for both problems and found the learned hyperparameter list performs comparable to random search in the underlying space. To probe further, we computed spearman correlation on the performance of each optimizer as compared to the rest of the tasks in the task suite. We found considerably worse correlations than where present for tasks in the TaskSet. This is not surprising as TaskSet contains no reinforcement learning problems.

B.3 LM1B targeting 20k iterations

We show a transformer on LM1B similar to that shown in §5 except run for only 20k iterations, a fith of the steps. Results in Figure S4. We find the learned hyperparameter lists are much more efficient than either of the baselines.

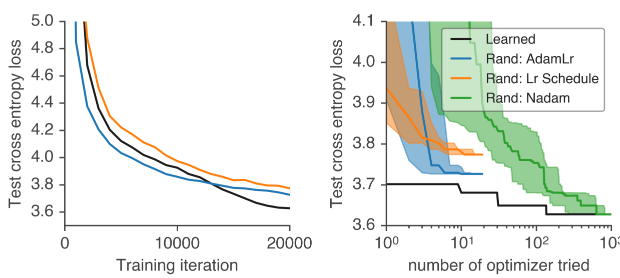

Figure S4: We find our learned hyperparameter lists out performs learning rate tuned Adam with both a constant, and a fixed learning rate schedule on a 53M parameter Transformer trained on LM1B. **Left:** Learning curves for the best of the optimizers. **Right:** Number of optimizers tried vs best test loss.

B.4 Probing short horizon

Often the goal when training a learned optimizers is to minimize performance after training some number of iterations. This is extremely computationally expensive and in practice approximations must be used. One common family of approximations is short horizon based methods. These methods rely upon somehow truncating training so that updates can be made to the learned optimizer more frequently. This is commonly done via truncated backprop [122, 123, 77, 128], or proxy objectives such as only training for a handful of epoch [136]. While this short horizon proxy is certainly

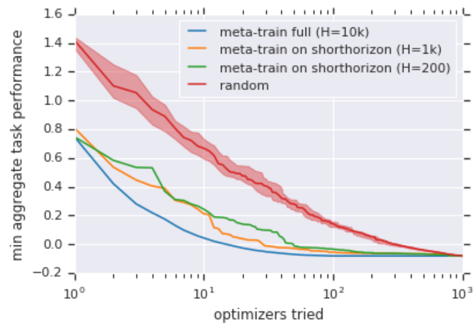

Figure S5: Hyperparameter lists trained on short horizon data generalize remarkably well. On the y-axis we show performance evaluated on the the full 10k training iterations for a given number of optimizers tried (x-axis). In color we show different number of steps used when evaluating task optimizer performance when training the hyperparameter list.

not optimal[128], the performance gains are immense and in practice is what makes meta-training optimizers feasible. In our task suite, we test this short horizon learning by training hyperparameter lists only using some finite amount of training iterations per task and testing in the full training regieme (10k steps). Results in figure S5. We find that even when learning the hyperparameter list on a mere 200 steps, our hyperparameter list continues to generalize to outperform random search on Adam8p. This is promising as this suggests that training the learned hyperparameter list can be done with 1/50th of the total compute. This result is surprising to us as prior work indicates the effect of this bias can be severe [128, 77]. We suspect it is due to the simplicity of the learned parameter space but leave a thorough analysis of this for future work.

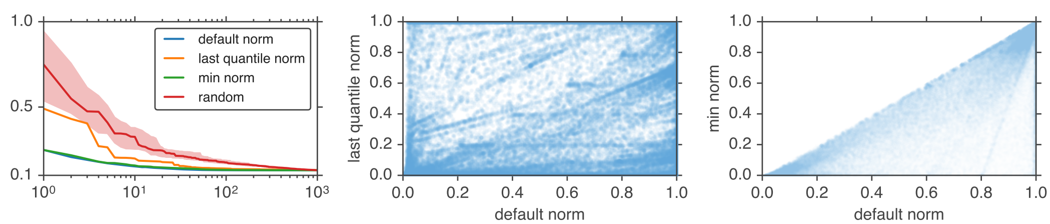

Figure S6: **Left:** Aggregate performance (y-axis) vs number of optimizer tried (x-axis) for different normalization and aggregation techniques. In each curve we train the hyperparameter list with a different normalization and aggregation strategy and test with the default normalization and aggregation technique described in 3.3. We find some some strategies are near identical in performance (e.g. min norm), while others perform significantly worse – e.g. last quantile norm. In both cases, however, we still perform better than the underlying random search. **Center:** Correlation between default normalization and the quantile based normalization strategy. Correlation is quite low – 0.193 Pearson's correlation. **Right:** Correlation between the default normalization using a mean to aggregate over validation over the course of training vs using a min over validation over the course training. We find a much higher correlation of 0.911.

B.5 Choice of normalization function

There is no easy way to define a single metric for optimizer performance over a mixture of tasks. This paper picks a single normalization strategy based on minimum validation loss and the validation loss at initialization presented in §3.3. In this section we show the impact of choosing a different normalization and or aggregation technique. First, instead of computing the mean over learning curves as described in §3.3 we compute a min. Second, instead of rescaling based on init and min, we linearly rescale based on the 95 percentile of validation loss and the min validation loss achieved at the end of training each task.In Figure S6 we show learned hyperparameter list training and testing performance as a function of number of optimizers tried when training with different normalization techniques. We find using the min instead of mean results in a negligible change, while using the percentile loss more significantly hurts performance. This difference can be explained by Figure S6b and S6c where we show correlations between the two losses. We find the percentile loss has a much weaker correlation to the default normalizer. We suspect this difference is due to the fact that many optimizers diverage on tasks. By using the 95 percentile we upweight optimizers that do not diverge.

B.6 Task families are diverse

To show the effects of diversity we train and test hyperparameter lists on each pair of task family. We additionally normalize each column from 0-1 to account for different mean losses across tasks. Results in Figure S7. While we do find some similarity in tasks – e.g. between MAF and NVP models, but no two tasks behave the same performance characteristics (no duplicate columns) suggesting that each task family is providing a different contribution to the space of all tasks. We also find when training on certain “far away” tasks, e.g. the quadratic family, we find poor performance on most other task families.

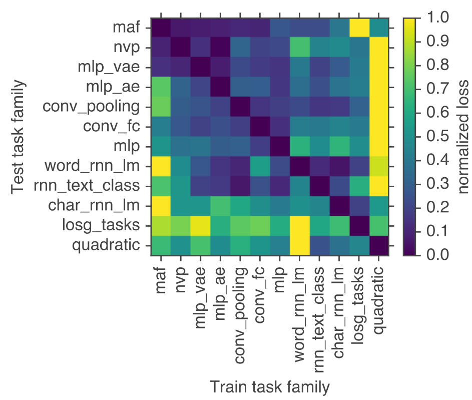

Figure S7: Learning hyperparameter lists using one task family and testing on the remainder of task families. We normalize each column from 0-1 to account for different mean losses across tasks. Lower loss means better performance. We find some groups of similar tasks, but in general no two task families behave identically.

B.7 Effects of the meta-training search space size

Our offline learning technique described in §3.4 hinges on a finite set of optimizers collected via random search. This set is denote by Θ in Eq.4. In this section we probe the impact of this size. We take different sized subsets of the the thousand Adam8p optimizer configurations and train and test search spaces on different iid splits of tasks. We then plot performance as a function of this number of optimizers in Figure S9. Moving left in this figure corresponds to increasing the compute needed to train the learned hyperparameter list. We find performance continues to improve as the size of Θ grows. Given the high dimension of our meta-parameters, 8, this is not a surprise as the number of evaluations needed to explore the space will grow exponentially. We find that the full thousand trials are needed to out perform learning rate tuned Adam when only given a single optimizer evaluation. We find around 100 optimizers (size of Θ) are needed in the case of 10 optimizer trials ($k = 10$).

Overall this sugjests that randomsearch might not be the most efficient learning method for creating hyperparameter lists. This is especially true as we work with optimizer families that have more hyperparameters. Other approximate learning methods should likely be explored such as truncated backprop through time as used by the learned optimizer community[77], and/or population based methods [7].

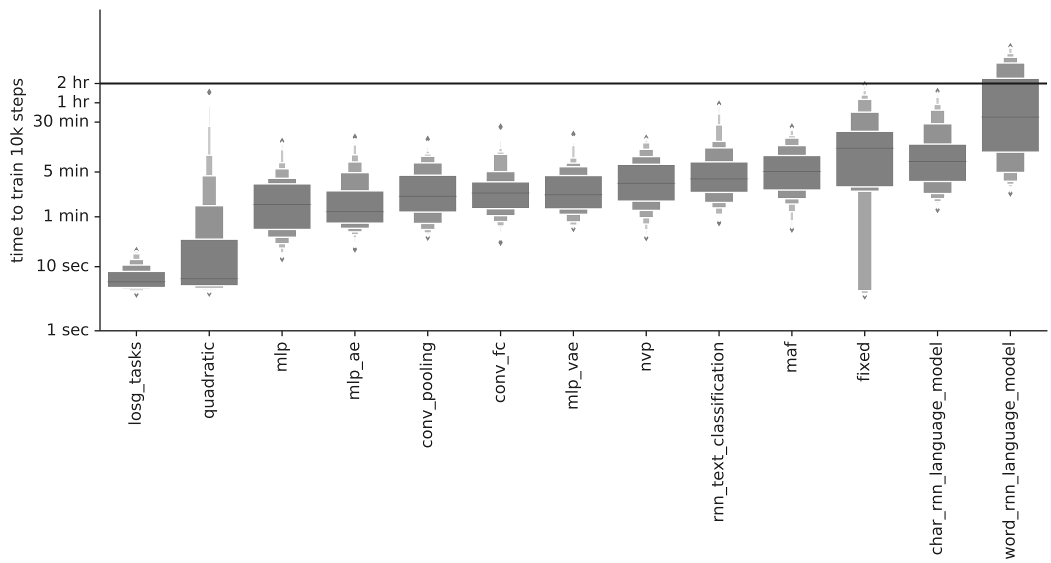

Figure S8: Timings computed for each task family. We find most task families have a narrow distribution of compute times.

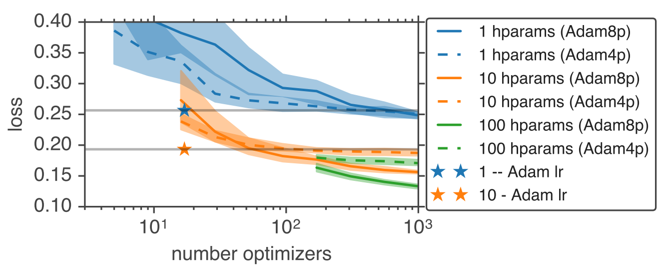

Figure S9: Performance continues to improve as more and more optimizers are used when training the search spaces. On the x-axis we show number of optimzers (size of Θ, the number of hyperparameter evaluations used in training the learned hyperparameter list) and y-axis we show test loss achieved when applying the learned search space for a given fixed length, e.g. different values of k shown in color). We plot median with 25-75 percentile shaded over different random optimizer samples and iid task splits. Stars (with horizontal guide lines) denote best search for the corresponding number of hyperparameters for learning rate tuned Adam in half orders of magnitude.

C Task timings

In Figure S8 we show box plots of training times for each problem. For each task we use the median step time recorded over a mixture of different physical devices and multipled by 10k to estimate a full training time. Future versions of this dataset of tasks will contain more variation within each task family.

D Optimizer family update equations

D.1 Adam8p update equations

The 8 meta-parameters are: the learning rate, α, first and second moment momentum, β_1, β_2, the numerical stability term, ϵ, ℓ_2 and ℓ_1 regularization strength, and learning rate schedule constants $\lambda_{\text{exp\_decay}}$ and $\lambda_{\text{linear\_decay}}$. For Adam6p, we set ℓ_1 and ℓ_2 to zero.

$$
\begin{aligned}
\phi^{(0)} =& \text{problem specified random initialization} && \text{(S1)}\\
m^{(0)} =& 0 && \text{(S2)}\\
v^{(0)} =& 0 && \text{(S3)}\\
g^{(t)} =& \frac{d}{d\phi^{(t)}}(f(x;\phi^{(t)}) + \ell_2||\phi^{(t)}||_2^2 + \ell_1||\phi^{(t)}||_1) && \text{(S4)}\\
m^{(t)} =& \beta_1 m^{(t-1)} + g^{(t)}(1-\beta_1) && \text{(S5)}\\
v^{(t)} =& \beta_2 v^{(t-1)} + (g^{(t)})^2(1-\beta_2) && \text{(S6)}\\
\hat{m}^{(t)} =& \frac{m^{(t)}}{1-\beta_1^{t+1}} && \text{(S7)}\\
\hat{v}^{(t)} =& \frac{v^{(t)}}{1-\beta_2^{t+1}} && \text{(S8)}\\
u^{(t)} =& \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} && \text{(S9)}\\
s_{\text{linear}}^{(t)} =& \max(1 - t\lambda_{\text{linear\_decay}}, 0) && \text{(S10)}\\
s_{\text{exp}}^{(t)} =& \exp(-t\lambda_{\text{exp\_decay}}) && \text{(S11)}\\
\phi^{(t+1)} =& \alpha s_{\text{linear}}^{(t)} s_{\text{exp}}^{(t)} u^{(t)} && \text{(S12)}
\end{aligned}
$$

D.2 NAdamW update equations

This optimizer family has 10 hyper parameters. The base learning rate, α_{base}, first and second moment momentum, β_1, β_2, the numerical stability term, ϵ, ℓ_{2WD} ℓ_2 regularization strength, ℓ_{2AdamW} AdamW style weight decay, and a boolean to switch between NAdam and Adam, $b_{\text{use nesterov}}$. The learning rate schedule is based off of a single cycle cosine decay with a warmup. It is controlled by 3 additional parameters – c_{warmup}, c_{constant}, and $c_{\text{min learning rate mult}}$.

The learning rate is defined by:

$$
\begin{aligned}
u =& c_{\text{warmup}} T > t && \text{(S13)}\\
\alpha_{\text{decay\&constant}} =& (\alpha_{base} - c_{\text{min learning rate mult}})(0.5 && \text{(S14)}\\
& \cos(t\pi/(T - c_{\text{constant}})) + 0.5) + && \text{(S15)}\\
& c_{\text{min learning rate mult}} && \text{(S16)}\\
\alpha_{\text{warmup}} =& \frac{t}{(Tc_{\text{warmup}})} && \text{(S17)}\\
\alpha =& (1-u)\alpha_{\text{decay\&constant}} + u\alpha_{\text{warm}} && \text{(S18)}
\end{aligned}
$$

The update equations of NAdamW are quite similar to that of Adam8p. For clarity we list the full update here.

$$\phi^{(0)} = \text{problem specified random initialization} \tag{S19}$$
$$m^{(0)} = 0 \tag{S20}$$
$$v^{(0)} = 0 \tag{S21}$$
$$g^{(t)} = \frac{d}{d\phi^{(t)}}(f(x;\phi^{(t)}) + \ell_{2wd}||\phi^{(t)}||_2^2 \tag{S22}$$
$$m^{(t)} = \beta_1 m^{(t-1)} + g^{(t)}(1-\beta_1) \tag{S23}$$
$$v^{(t)} = \beta_2 v^{(t-1)} + (g^{(t)})^2(1-\beta_2) \tag{S24}$$
$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^{t+1}} \tag{S25}$$
$$\hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^{t+1}} \tag{S26}$$
$$u^{(t)}_{\text{heavy ball}} = \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon} \tag{S27}$$
$$u^{(t)}_{\text{nesterov}} = \frac{\beta_1\hat{m}^{(t)} + (1-\beta_1)g^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon} \tag{S28}$$
$$\phi^{(t+1)} = \phi^{(t)} - (1-b_{\text{use nesterov}})\alpha u^{(t)}_{\text{heavy ball}} + \tag{S29}$$
$$b_{\text{use nesterov}}\alpha u^{(t)}_{\text{nesterov}} - \alpha\ell_{2AdamW}\phi^{(t)} \tag{S30}$$

E Optimizer family search spaces

E.1 Search Space Considerations

The performance of random search critically depends on the boundaries of the original search space. Without prior knowledge about the problems, however, picking a good search space is difficult. To explore this we additionally choose search spaces *after* collecting and looking at the data. We then use this search space to simulate random search within the constraints via rejection sampling. To find these search spaces we find the best hyper parameters for each task and construct new hyperparameter ranges with min and max values determined by the smallest and largest values of each hyperparameter which were the best hyperparameter for some task. This removes regions of the search space not used by any task. We also tested bounds based on the 5th and 95th percentile of best performing hyperparameters computed over all tasks. In the case of min and max, we find the optimal hyperparameters cover nearly all of the existing space, whereas the percentile based search spaces reduces the volume of the search hypercube by more than 90% leaving us with only $\sim$100 hyperparameter configurations. In Figure 3, we find, in all cases, learning the hyperparameter list is much more efficient.

E.2 Adam8p, Adam6p, Adam4p, AdamLr search spaces

For Adam1p, Adam4p, Adam6p, and Adam8p we sample learning rate logritmically between 1e-8 and 10, beta1 and beta2 we parametrize as $1-x$ and sample logrithmically between 1e-4 and 1 and 1e-6 and 1 respectively. For learning rate schedules we sample linear decay between 1e-7, 1e-4 logrithmically and exponential decay logrithmically between 1e-3, 1e-6. We sample both ℓ_1 and ℓ_2 logrithmcally between 1e-8, 1e1.

E.3 NAdamW search space

This search space was chosen heuristically in an effort to generalize to new problems. We would like to emphasize that it was not tuned. We used our insight from Adam based optimizer families and chose this. No iterations where done. We expect more iterations will improve not only in distribution performance, but alsos generalization performance.

The initial learning rate, α_{base} is sampled from log space between $1e-5$ and 1.0. $1-\beta_1$ is sampled logrithmically between $1e-3$, and 1.0. $1-\beta_2$ is sampled between $1e-5$, and 1.0. ϵ is sampled logarithmically between $1e-8$ and $1e4$. We sample using nesterov ($b_{\text{use nesterov}}$) 50% of the time. We sample ℓ_{2WD} and ℓ_{2AdamW} logrithmically between $1e-5$ and $1e-1$. Equal probabilities of a third we either use both terms, zero out ℓ_{2WD}, or zero out ℓ_{2AdamW}. With 50% probability we use a nonzero min learning rate multiplier sampled logrithmically between $1e-5$ and 1.0. With 50% probability we sample the warm up fraction, c_{warmup} between 1e-5 and 1e-1, otherwise it is set to zero. Finally, we uniformly sample the amount of time the learning rate is held constant(c_{constant}) between 0 and 1.

F Extended related work

F.1 Sets of tasks

Benchmarks consisting of multiple tasks are becoming an increasingly common technique for measuring improvement in algorithm design. Reinforcement learning has Atari [9], DMLab [8], gym [20], and dm_control [109]. Natural language processing has evaluation sets such as GLUE [120], Super GLUE [121], and the NLPDecathalon [75]. In computer vision there is [134] which studies transfer learning of image features. In black box optimization there is Nevergrad [93], COmparing Continuous Optimizers (COCO) [46] and a number of tasks to test Bayesian hyperparameter optimization presented in [29]. For first order gradient methods there are unit tests for stochastic optimization [96] which studies toy optimization functions, and DeepObs [99] which includes 20 neural network tasks. Hyperparameter tuning practices on these benchmarks vary between tuning on each task separately, to tuning one set of hyperparameters for all problems. In Atari [9], for example, it is common practice to tune hyperparameters on a subset of tasks and evaluate on the full set. This protocol can further be extended by leveraging unseen levels or games at test time as done in Obstacle Tower [55], ProcGen [28], CoinRun [27], and Sonic [82]. We believe generalization to unseen tasks is key for learned algorithms to be useful thus our learned search space experiments mirror this setting by making use of hold out tasks.

Existing meta-learning data sets share similar goals to our work but focus on different domains. In few shot learning there is MiniImageNet [119] which is built procedurally from the ImageNet dataset [95]. Meta-Dataset [114] takes this further and also focuses on generalization by constructing few shot learning tasks using images from a number of different domains for evaluation purposes. The automated machine learning community has OpenML [117] with a focus on selecting and tuning non-neural algorithms. For learning optimizers, the use of task suites has been limited and ad-hoc. Many works use a single or small number of standard machine learning tasks [4, 66, 71, 77]. Wichrowska et al. [123] uses a set of synthetic problems meant to emulate many different kinds of loss surfaces. While existing collections of tasks exist for optimizer evaluation, e.g. [99], they contain too small a number of tasks to act as a comprehensive training set for learning algorithms, and many of their tasks are additionally too computationally expensive to be useful during learning.

F.2 Hand designed and learned optimizers

Optimization is core to machine learning and thus the focus of extensive work. Methods such as Nesterov momentum [81], AdaGrad [34], RMSProp [111], and Adam [57] have all shown considerable improvements in both the speed of optimization and ease of use by exposing robust, and easier to tune hyperparameters than SGD [103]. Adaptive step size methods in particular have emerged at the forefront with many works building from it including AdamW [70], RAdam [69], Novograd [41], and NAdam [33]. Recently, there has been a focus on comparing optimizers either for best performance, or ease of use [124, 24, 99, 103]. This has proven difficult as performance is heavily dependent on the choice of search space for optimization hyperparameters [24].

Learned optimizers represent a parallel thread in the development of optimizers. By learning as opposed to hand-designing optimizers, researchers hope to not only increase performance but also ease of use (e.g. minimize the number of hyperparameters required or lower hyperparameter sensitivity) [11, 97, 53]. Recently, there has been renewed interest in parameterizating learning algorithms with neural networks and learning these optimizers on neural network based losses [4, 123, 66, 71, 77, 78]. Other approaches make learn symbolic parameterizations for new optimizers [10]. These various methods are all trained and evaluated on different distributions of tasks making comparison across papers challenging. The dataset of tasks presented here will hopefully aid in the ability to compare and evaluate progress in learned optimizer research.

In this work, we develop a much more minimal type of "learned optimizer" than previous work which developed new functional forms for the optimizer. Optimization involves not only the functional form of the optimizer, but also the rules for choosing hyperparameters and applying the optimizer. We focus on this second aspect of optimization and learn a hyperparameter search space to improve the performance of existing hand designed methods.

F.3 Hyperparameter search

Hyperparameter search is a key component in machine learning. Considerable improvements have been made in language [76], computer vision [104], and RL [23] simply by tuning better. Often no single hyperparameter configuration works well across all tasks for existing optimization methods. Most current hyperparameter search methods involve trying a very large number of hyperparameters for every new task, which is computationally infeasible for large tasks, and additionally can severely limit the number of hyperparameters that can be tuned. Many common techniques such as random search [12, 16], Bayesian optimization [104, 105], tree parzen estimators [13], or sequential halving [63] require setting a hyperparameter search space by hand which is not only difficult but often wildly inefficient.

Learning hyperparameters or search strategies by leveraging multiple tasks has been explored within the context of Bayesian optimization [107, 87, 88] as well as under the term meta-learning in Chen et al. [22] in which an LSTM is meta-trained to produce function locations to query.

The cost of hyperparameter search is often large as each evaluation requires training a model to completion. Often multi-fidelity based approaches are used which leverage "simpler" tasks and transfer the resulting hyperparameters [54]. Common approaches include training on partial function evaluations [108, 32, 67, 60, 37], or leveraging simplified data and models [89, 135, 19]. Our dataset of tasks serves as a: "simpler" set of tasks to train on; a large and diverse enough set of problems that optimization algorithms trained on it may be expected to generalize; and a framework to test transfer across different types of problems.

G List of NAdam HParams

Idx	Lr	warmup	constant	Min LR mult	beta1	beta2	epsilon	nesterov	l2 reg	l2 weight decay
0	1.24e-3	0.000	0.477	1.01e-3	0.94666	0.94067	8.114e-8	False	0.000e+00	7.258e-5
1	5.33e-3	0.000	0.172	0.0	0.96047	0.99922	8.665e-8	True	0.000e+00	5.563e-3
2	2.12e-4	0.000	0.210	1.39e-3	0.62297	0.97278	1.540e-7	False	0.000e+00	5.361e-2
3	4.06e-1	0.000	0.324	0.0	0.99724	0.98680	1.079e+02	True	0.000e+00	1.562e-2
4	2.05e-2	0.000	0.885	1.57e-5	0.35731	0.86043	8.874e-5	True	0.000e+00	7.217e-2
5	5.95e-4	0.008	0.378	0.0	0.89130	0.99983	1.483e-7	True	0.000e+00	4.087e-2
6	7.53e-3	0.000	0.422	9.55e-4	0.69192	0.98434	3.593e-8	False	0.000e+00	3.060e-4
7	4.69e-3	0.000	0.509	0.0	0.99639	0.98820	2.056e-5	False	0.000e+00	3.552e-2
8	2.95e-1	0.000	0.201	0.0	0.99678	0.99981	7.498e+00	False	3.792e-4	3.463e-4
9	2.04e-3	0.000	0.527	0.0	0.49995	0.99755	5.630e-8	True	0.000e+00	2.796e-2
10	7.39e-1	0.001	0.556	3.31e-3	0.99691	0.80639	2.900e+03	False	0.000e+00	7.851e-2
11	8.12e-3	0.000	0.207	0.0	0.17785	0.96033	7.971e-2	False	0.000e+00	1.489e-2
12	3.33e-2	0.000	0.369	0.0	0.69592	0.99997	5.510e-6	True	0.000e+00	1.362e-5
13	6.95e-3	0.000	0.014	0.0	0.99412	0.99305	4.352e-7	False	0.000e+00	3.142e-5
14	1.88e-1	0.000	0.205	1.08e-1	0.98597	0.56531	3.335e+00	True	1.265e-5	3.868e-3
15	9.47e-4	0.007	0.452	0.0	0.43977	0.09422	2.120e-7	False	0.000e+00	6.902e-3
16	3.75e-3	0.000	0.184	0.0	0.87756	0.96128	3.163e-3	True	7.468e-5	2.627e-3
17	7.25e-1	0.000	0.495	0.0	0.99800	0.99781	3.608e+00	True	1.656e-5	3.911e-2
18	4.58e-3	0.000	0.107	3.66e-1	0.42294	0.99963	4.174e-6	True	0.000e+00	4.446e-3
19	3.07e-4	0.007	0.518	0.0	0.57863	0.99625	9.881e-6	False	0.000e+00	5.521e-2
20	2.94e-5	0.000	0.830	8.27e-5	0.96916	0.99896	7.782e-7	True	3.364e-4	3.416e-3
21	1.65e-4	0.002	0.457	2.70e-1	0.95280	0.04565	2.832e-6	True	0.000e+00	1.141e-2
22	9.17e-1	0.010	0.897	2.67e-2	0.45061	0.99244	4.945e-1	False	1.253e-3	0.000e+00
23	2.36e-3	0.000	0.986	0.0	0.98560	0.99997	1.080e-8	True	0.000e+00	3.023e-3
24	2.14e-2	0.000	0.128	0.0	0.98741	0.99336	1.266e-4	False	0.000e+00	5.194e-4
25	5.91e-2	0.000	0.062	0.0	0.99794	0.99383	3.447e+02	True	0.000e+00	3.935e-2
26	1.57e-3	0.000	0.251	0.0	0.91820	0.99991	4.675e-5	False	0.000e+00	4.112e-5
27	4.43e-1	0.000	0.702	0.0	0.94375	0.93551	2.335e-8	True	0.000e+00	8.325e-5
28	2.98e-3	0.008	0.046	0.0	0.68612	0.94232	6.614e-2	False	6.489e-5	0.000e+00
29	1.65e-2	0.004	0.082	4.92e-4	0.95717	0.99789	3.068e+01	True	0.000e+00	8.920e-2
30	5.58e-3	0.000	0.538	0.0	0.97559	0.99990	3.238e-8	True	0.000e+00	4.896e-4
31	8.54e-1	0.000	0.229	0.0	0.93129	0.50200	2.051e-2	False	2.068e-4	2.801e-2
32	7.38e-3	0.000	0.722	8.78e-2	0.21456	0.99752	2.862e-2	False	0.000e+00	8.439e-2
33	4.26e-4	0.001	0.923	2.06e-1	0.47239	0.99974	8.221e-5	False	1.248e-5	0.000e+00
34	6.04e-3	0.000	0.698	0.0	0.97849	0.91449	1.806e+00	False	3.183e-3	1.762e-2
35	8.86e-3	0.000	0.104	1.66e-1	0.98967	0.99720	1.493e-2	True	0.000e+00	2.253e-2
36	1.51e-2	0.000	0.431	1.99e-3	0.80488	0.97878	2.538e-8	True	0.000e+00	2.269e-5
37	2.50e-3	0.000	0.009	0.0	0.98127	0.99988	1.799e-7	False	0.000e+00	1.303e-2
38	3.42e-4	0.000	0.827	6.38e-1	0.25217	0.96572	2.928e-7	True	0.000e+00	1.318e-3
39	6.94e-5	0.000	0.085	0.0	0.98674	0.42709	2.387e-7	False	0.000e+00	2.071e-4
40	3.03e-2	0.001	0.313	0.0	0.90610	0.99997	4.449e-3	True	0.000e+00	2.813e-5
41	4.64e-3	0.000	0.495	2.26e-5	0.64658	0.54108	3.528e-8	False	0.000e+00	2.996e-5
42	2.25e-3	0.000	0.722	0.0	0.97967	0.97518	1.488e-7	True	1.812e-5	2.180e-2
43	6.66e-4	0.000	0.632	2.79e-5	0.65968	0.99997	6.848e-6	True	0.000e+00	3.130e-3
44	3.31e-3	0.000	0.146	0.0	0.90447	0.99970	6.618e-6	True	0.000e+00	2.184e-2
45	7.84e-4	0.016	0.124	0.0	0.95065	0.99685	2.141e-2	False	0.000e+00	4.024e-5
46	6.16e-3	0.016	0.623	0.0	0.98823	0.98744	1.616e-6	False	0.000e+00	1.544e-2
47	3.26e-4	0.000	0.738	1.61e-4	0.78425	0.99998	3.468e-3	False	0.000e+00	4.709e-2
48	4.12e-3	0.001	0.205	0.0	0.99561	0.75382	2.390e-6	True	0.000e+00	3.631e-2
49	6.26e-1	0.000	0.932	2.52e-3	0.99401	0.83521	2.431e+00	True	0.000e+00	1.048e-2

Top 50 hyper parameters found using the NAdamW search space. We find diverse learning rates, with very little warmup used. We additionally find most good performing optimizers make use of AdamW style weight decay. Finally, matching insight from [24], we find large values of ϵ.

H Description of tasks in task suite

In this section we detail the task distribution used throughout this work. In addition to this text, a Tensorflow [2] implementation is also released at github.com/google-research/google-research/tree/master/task_set.

H.1 Sampled Tasks

H.1.1 Default sampled components

As many of the sampled tasks are neural networks. We define common sampling routines used by all the sampled tasks.

Activation functions: We define a distribution of activation functions which is sampled corresponding the following listing both name and weight. These are a mix of standard functions (relu, tanh) to less standard (cos).

- relu: 6
- tanh: 3
- cos: 1
- elu: 1
- sigmoid: 1
- swish [92]: 1
- leaky relu (with $\alpha = 0.4$): 1
- leaky relu (with $\alpha = 0.2$): 1
- leaky relu (with $\alpha = 0.1$): 1

Initializations: We sample initializers according to a weighted distribution. Each initialization sample also optionally samples hyperparameters (e.g. for random normal initializers we sample standard deviation of the underlying distribution).

- he normal [49]: 2
- he uniform [49]: 2
- glorot normal [42]: 2
- glorot uniform [42]: 2
- orthogonal: 1. We sample the "gain", or multiplication of the orthogonal matrix logarithmically between $[0.1, 10]$.
- random uniform 1.0: This is defined between $[-s, s]$ where s is sampled logarithmically between $[0.1, 10]$.
- random normal: 1.0: The std is sampled logarithmically between $(0.1, 10)$.
- truncated normal: 1.0: The std is sampled logarithmically between $(0.1, 10)$.
- variance scaling: 1.0: The scale is sampled logarithmically between $(0.1, 10)$.

RNN Cores: We define a distribution over different types of RNN cores used by the sequential tasks. With equal probability we sample either a vanilla RNN [36], GRU[26], or LSTM[52]. For each cell we either sample 1 shared initialization method or sample a different initialization method per parameter vector with a 4:1 ratio. We sample the core hidden dimension logarithmically between $[32, 128]$.

H.1.2 Sampled Datasets

Image Datasets: We sample uniformly from the following image datasets. Each dataset additionally has sampled parameters. For all datasets we make use of four data splits: train, valid-inner, valid-outer, test. Train is used to train models, valid-inner is used while training models to allow for modification of the training procedure (e.g. if validation loss doesn't increase, drop learning rate). Valid-outer is used to select meta-parameters. Test should not be used during meta-training.

For all datasets, we sample a switch with low probability (10% of the time) to only use training data and thus not test generalization. This ensures that our learned optimizers are capable of optimizing a loss as opposed to a mix of optimizing and generalizing.

Mnist: Batch size is sampled logarithmically between $[8, 512]$. We sample the number of training images logarithmically between $[1000, 55000]$ [64].

Fashion Mnist: Batch size is sampled logarithmically between $[8, 512]$. We sample the number of training images logarithmically between $[1000, 55000]$ [129].

Cifar10: Batch size is sampled logarithmically between $[8, 256]$. The number of training examples is sampled logarithmically $[1000, 50000]$ [61].

Cifar100: Batch size is sampled logarithmically between $[8, 256]$. The number of training examples is sampled logarithmically $[1000, 50000]$ [61].

{food101_32x32, coil100_32x32, deep_weeds_32x32, sun397_32x32}: These dataset take the original set of images and resize them to 32x32 using OpenCV's [18] cubic interpolation. We ignore aspect ratio for this resize. Batch size is sampled logarithmically between $[8, 256]$ [15, 80, 83, 130].

Imagenet32x32 / Imagenet16x16: The ImageNet 32x32 and 16x16 dataset as created by Chrabaszcz et al. [25]. Batch size is logrithmically sampled between $[8, 256]$.

H.1.3 Text classification:

IMDB sentiment classification: We use text from the IMDB movie reviews dataset[72] and tokenize using subwords using a vocab size of 8k[101]. We then take length s random slice from each example where s is sampled logarithmically between $[8, 64]$. These examples are then batched into a batch size logarithmically sampled between $[8, 512]$. We sample the number of training examples logarithmically between $[1000, 55000]$ and with 10% probability just use training data instead of valid / test to test pure optimization as opposed to generalization.

H.1.4 Character and Word language Modeling

For the character and word language modeling datasets we make use of the following data sources: **imdb movie reviews**[72], **amazon product reviews** [1] using the Books, Camera, Home, and Video subset each as separate datasets, LM1B[21], and **Wikipedia**[40] taken from the 20190301 dump using the zh, ru, ja, hab, and en language codes. We split each article by new lines and only keep resulting examples that contain more than 5 characters. For infrastructure reasons, we only use a million articles from each language and only 200k examples to build the tokenizer.

Byte encoding: We take length s random slices of each example where s is sampled logarithmically between $[10, 160]$. These examples are then batched into a batch size logarithmically sampled between $[8, 512]$. With probability 0.2 we restrict the number of training examples to a number logarithmically sampled between $[1000, 50000]$. Finally, with a 10% probability just use training data instead of valid / test to test pure optimization as opposed to generalization.

subword encoding: We encode the text as subwords with a vocabsize of 8k [101]. We then take length s random slices of each example where s is sampled logarithmically between $[10, 256]$. These examples are then batched into a batch size logarithmically sampled between $[8, 512]$. With probability 0.2 we restrict the number of training examples to a number logarithmically sampled between $[1000, 50000]$. Finally, with a 10% probability just use training data instead of valid / test to test pure optimization as opposed to generalization.

H.2 Sampled Tasks

H.2.1 MLP

This task family consists of a multi layer perceptron trained on flattened image data. The amount of layers is sampled uniformly from $[1, 6]$. Layer hidden unit sizes are sampled logarithmically between $[16, 128]$ with different number of hidden units per layer. One activation function is chosen for the whole network and is chosen as described in H.1.1. One shared initializer strategy is also sampled. The image dataset used is also sampled.

Two sampled configurations are shown below.


```
{
  "layer_sizes": [
    71
  ],
  "activation": "leaky_relu2",
  "w_init": [
    "he_normal",
    null
  ],
  "dataset": [
    "sun397_32x32",
    {
      "bs": 32,
      "just_train": false,
      "num_train": null
    },
    {
      "crop_amount": 0,
      "flip_left_right": false,
      "flip_up_down": true,
      "do_color_aug": false,
      "brightness": 0.002936489121851211,
      "saturation": 0.4308521744067503,
      "hue": 0.19648945965587863,
      "contrast": 0.036096320130911644
    }
  ],
  "center_data": false
}
```



```
{
  "layer_sizes": [
    68,
    37,
    78
  ],
  "activation": "relu",
  "w_init": [
    "glorot_normal",
    null
  ],
  "dataset": [
    "food101_32x32",
    {
      "bs": 117,
      "just_train": true,
      "num_train": null
    },
    null
  ],
  "center_data": true
}
```

H.2.2 MLP_ae

This task family consists of a multi layer perceptron trained with an auto encoding loss. The amount of layers is sampled uniformly from $[2, 7]$. Layer hidden unit sizes are sampled logarithmically between $[16, 128]$ with different number of hidden units per layer. The last layer always maps back to the input dimension. The output activation function is sampled with the following weights: tanh:2, sigmoid:1, linear_center:1, linear:1 where linear_center is an identity mapping. When using the linear_center and tanh activation we shift the ground truth image to $[-1, 1]$ before performing a comparison to the model's predictions. We sample the per dimension distance function used to compute loss with weights l2:2, l1:1, and the reduction function across dimensions to be either mean or sum with equal probability. A single activation function, and initializer is sampled. We train on image datasets which are also sampled.

A sample configurations is shown below.


```
{
  "hidden_units": [
    73,
    103,
    105,
    104,
    76
  ],
  "activation": "relu",
  "w_init": [
    "glorot_uniform",
    null
  ],
  "dataset": [
    "mnist",
    {
      "bs": 39,
      "num_train": 43753,
      "num_classes": 10,
      "just_train": false
    },
    null
  ],
  "output_type": "tanh",
  "loss_type": "l2",
  "reduction_type": "reduce_sum"
}
```


H.2.3 MLP VAE

This task has an encoder with sampled number of layers between $[1, 3]$. For each layer we sample the number of hidden units logarithmically between $[32, 128]$. For the decoder we sample the number of layers uniformly between $[1, 3]$. For each layer we sample the number of hidden units logarithmically between $[32, 128]$. We use a gaussian prior of dimensionality logarithmically sampled between $[32, 128]$. A single activation function and initialization is chosen for the whole network. The output of the encoder is projected to both a mean, and a log standard deviation which parameterizes the variational distribution, $q(z|x)$. The decoder maps samples from the latent space to a quantized gaussian distribution in which we compute data log likelihoods $\log p(x|z)$. The loss we optimize is the evidence lower bound (ELBO) which is computed by adding this likelihood to the kl divergence between our normal distribution prior and $q(z|x)$. We use the reparameterization trick to compute gradients. This model is trained on sampled image datasets.

A sample configuration is listsed below.


```
{
  "enc_hidden_units": [
```

```
    73
  ],
  "dec_hidden_units": [
    74
  ],
  "activation": "relu",
  "w_init": [
    "he_normal",
    null
  ],
  "dataset": [
    "food101_32x32",
    {
      "bs": 22,
      "just_train": true,
      "num_train": null
    },
    null
  ]
}
```

H.2.4 Conv Pooling

This task consists of small convolutional neural networks with pooling. We sample the number of layers uniformly between $[1, 5]$. We sample a stride pattern to be either all stride 2, repeating the stride pattern of 1,2,1,2... for the total number of layers, or 2,1,2,1... for the total number of layers. The hidden units are logarithmically sampled for each layer between $[8, 64]$. We sample one activation function and weight init for the entire network. Padding for the convolutions are sampled per layer to either be same or valid with equal probability. For the convnet we also sample whether or not to use a bias with equal probability. At the last layer of the convnet we do a reduction spatially using either the mean, max, or squared mean sampled uniformly. This reduced output is fed into a linear layer and a softmax cross entropy loss. These models are trained on a sampled image dataset.

A sample configuration is shown below.

```
{
  "strides": [
    [1, 1],
    [2, 2],
    [1, 1],
    [2, 2],
    [1, 1]
  ],
  "hidden_units": [
    46,
    48,
    47,
    29,
    18
  ],
  "activation": "leaky_relu4",
  "w_init": [
    "glorot_normal",
    null
  ],
  "padding": [
    "SAME",
    "SAME",
    "VALID",
    "SAME",
    "VALID"
```

```
  ],
  "pool_type": "squared_mean",
  "use_bias": true,
  "dataset": [
    "cifar100",
    {
      "bs": 10,
      "num_train": 5269,
      "just_train": true
    },
    null
  ],
  "center_data": false
}
```

H.2.5 Conv FC

This task consists of small convolutional neural networks, flattened, then run through a MLP. We sample the number of conv layers uniformly between $[1, 5]$. We sample a stride pattern to be either all stride 2, repeating the stride pattern of 1,2,1,2... for the total number of layers, or 2,1,2,1... for the total number of layers. The hidden units are logarithmically sampled for each layer between $[8, 64]$. Padding for the convolutions are sampled per layer to either be same or valid with equal probability.

The output is then flattened, and run through a MLP with hidden layers sampled uniformly from $[0, 4]$ and with sizes sampled logrithmically from $[32, 128]$. The loss is then computed via softmax cross entropy.

We sample one activation function and weight init for the entire network. For the convnet we also sample whether or not to use a bias with equal probability. These models are trained on a sampled image dataset.

An example configuration is shown below.

```
{
  "strides": [
    [2, 2],
    [2, 2],
    [2, 2],
    [2, 2]
  ],
  "hidden_units": [
    17,
    30,
    13,
    16
  ],
  "activation": "relu",
  "w_init": [
    "glorot_uniform",
    null
  ],
  "padding": [
    "VALID",
    "VALID",
    "VALID",
    "SAME"
  ],
  "fc_hidden_units": [],
  "use_bias": true,
  "dataset": [
    "coil100_32x32",
    {
```

```
      "bs": 49,
      "just_train": false,
      "num_train": null
    },
    null
  ],
  "center_data": true
}
```

H.2.6 character rnn language model

This task takes character embedded data, and embeds in a size s embedding vector where s is sampled logarithmically between $[8, 128]$ with random normal initializer with std 1.0. With 80% we use all 256 tokens, and with 20% chance we only consider a subset of tokens sampled logarithmically $[100, 256]$. We then pass this embedded vector to a RNN with teacher forcing with equal probability we use a trainable initializer or zeros. A linear projection is then applied to the number of vocab tokens. Losses are computed using a softmax cross entropy vector and mean across the sequence.

A sample configuration is shown below.

```
{
  "embed_dim": 30,
  "w_init": [
    "he_normal",
    null
  ],
  "vocab_size": 256,
  "core": [
    "gru",
    {
      "core_dim": 84,
      "wh": [
        "glorot_uniform",
        null
      ],
      "wz": [
        "random_normal",
        0.4022641748407826
      ],
      "wr": [
        "he_uniform",
        null
      ],
      "uh": [
        "he_normal",
        null
      ],
      "uz": [
        "glorot_normal",
        null
      ],
      "ur": [
        "glorot_uniform",
        null
      ]
    }
  ],
  "trainable_init": true,
  "dataset": [
    "lm1b/bytes",
    {
```

```
      "patch_length": 147,
      "batch_size": 63,
      "just_train": false,
      "num_train": null
    }
  ]
}
```


H.2.7 word rnn language model

This task takes word embedded data, and embeds in a size s embedding vector where s is sampled logarithmically between $[8, 128]$ with random normal initializer with std 1.0. A vocab size for this embedding table is sampled logarithmically between $[1000, 30000]$. We then pass this embedded vector to a RNN with teacher forcing with equal probability we use a trainable initializer or zeros. A linear projection is then applied to the number of vocab tokens. Losses are computed using a softmax cross entropy vector and mean across the sequence.

A sample configuration shown below.


```
{
  "embed_dim": 91,
  "w_init": [
    "glorot_uniform",
    null
  ],
  "vocab_size": 13494,
  "core": [
    "gru",
    {
      "core_dim": 96,
      "wh": [
        "he_normal",
        null
      ],
      "wz": [
        "he_normal",
        null
      ],
      "wr": [
        "he_normal",
        null
      ],
      "uh": [
        "he_normal",
        null
      ],
      "uz": [
        "he_normal",
        null
      ],
      "ur": [
        "he_normal",
        null
      ]
    }
  ],
  "trainable_init": true,
  "dataset": [
    "tokenized_amazon_reviews/Video_v1_00_subwords8k",
    {
      "patch_length": 14,
```

```
      "batch_size": 59,
      "just_train": false,
      "num_train": null
    }
  ]
}
```

H.2.8 LOSG Problems

These tasks consist of a mixture of many other tasks. We sample uniformly over the following types of problems. We brielfy describe them here but refer reader to the provided source for more information. In this work we took all the base problems from [123] but modified the sampling distributions to better cover the space as opposed to narrowly sampling particular problem families. Future work will consist of evaluating which sets of problems or which sampling decisions are required.

quadratic: n dimensional quadratic problems where n is sampled logarithmically between $[10, 1000]$. Noise is optionally added with probability 0.5 and of the scale s where s is sampled logarithmically between $[0.01, 10]$.

bowl: A 2d qaudratic bowl problem with a sampled condition number (logrithmically between $[0.01, 100]$). Noise is optionally added with probability 0.5 and of the scale s where s is sampled logarithmically between $[0.01, 10]$.

sparse_softmax_regression: A synthetic random sparse logistic regression task.

optimization_test_problems: A uniform sample over the following functions: Ackley, Beale, Branin, logsumexp, Matyas, Michalewicz, Rosenbrock, StyblinskiTang.

fully_connected: A sampled random fully connected classification neural network predicting 2 classes on synthetic data. Number of input features is sampled logrithmically between 1 and 16, with a random activation function, and a sampled number of layers uniformly sampled from 2-5.

norm: A problem that finds a minimum error in an arbitrary norm. Specifically: $(\sum(Wx - y)^p)^(\frac{1}{p})$ where $W \in \mathcal{R}^{NxN}$, $y \in \mathcal{R}^{Nx1}$. The dimentionality, N, is sampled logrithmically between 3, and 1000. The power, p, is sampled uniformly between 0.1 and 5.0. W, and y are drawn from a standard normal distribution.

dependency_chain: A synthetic problem where each parameter must be brought to zero sequentially. We sample dimensionality logrithmically between 3, 100.

outward_snake: This loss creates a winding path to infinity. Step size should remain constant across this path. We sample dimensionality logrithmically between 3 and 100.

min_max_well: A loss based on the sum of min and max over parameters: $\max x + 1/(\min x) - 2$. Note that the gradient is zero for all but 2 parameters. We sample dimentaionlity logrithmically between 10 and 1000. Noise is optionally added with probability 0.5 and of the scale s where s is sampled logarithmically between [0.01, 10].

sum_of_quadratics: A least squares loss of a dimentionality sampled logrithmically between 3 and 100 to a synthetic dataset.

projection_quadratic: A quadratic minimized by probing different directions. Dimentionality is sampled from 3 to 100 logrithmically.

In addition to these base tasks, we also provide a variety of transformations described bellow. The use of these transformations is also sampled.

sparse_problems: With probability 0.9 to 0.99 the gradient per parameter is set to zero. Additional noise is added with probability 0.5 sampled from a normal with std sampled logrithmically between $[0.01, 10.0]$.

rescale_problems: Rescales the loss value by 0.001 to 1000.0 sampled logrithmically.

log_objective: Takes the log of the objective value.

2 Sample configurations shown below.

```
[
  "fully_connected",
  {
    "n_features": 16,
    "n_classes": 2,
    "activation": "leaky_relu2",
    "bs": 7,
    "n_samples": 12,
    "hidden_sizes": [
      32,
      8,
      5,
      9,
      8
    ]
  },
  36641
]
```

```
[
  "outward_snake",
  {
    "dim": 9,
    "bs": 30,
    "n_samples": 249
  },
  79416
]
```

```
[
  "rescale_problems",
  {
    "base": [
      "sum_of_quadratics",
      {
        "dim": 36,
        "bs": 5,
        "n_samples": 1498
      }
    ],
    "scale": 227.86715292020605
  },
  89629
]
```

H.2.9 Masked Autoregressive Flows

Masked autoregressive flows are a family of tractable density generative models. See XX for more information. The MAF is defined by a sequence of bijectors. For one bijector samples a number of layers to either be 1 or 2 with equal probability, and a number of hidden layers sampled logarithmically between $[16, 128]$. We sample the number of bijector uniformly from $[1, 4]$ and use the same hidden layers across all bijector. We sample activation function, and initializer once for the whole model. In this task we model image datasets which are also sampled.

A sample configuration is shown below.

```
{
  "activation": "relu",
```

```
  "w_init": [
    "he_uniform",
    null
  ],
  "dataset": [
    "imagenet_resized/16x16",
    {
      "bs": 19,
      "just_train": true,
      "num_train": null
    },
    null
  ],
  "hidden_units": [
    44,
    24
  ],
  "num_bijectors": 3
}
```

H.2.10 Non volume preserving flows

NVP are a family of tractable density generative models. See [30] for more information. The NVP is defined by a sequence of bijectors. For one bijector samples a number of layers to either be 1 or 2 with equal probability, and a number of hidden layers sampled logarithmically between $[16, 128]$. We sample the number of bijector uniformly from $[1, 4]$ and use the same hidden layers across all bijector. We sample activation function, and initializer once for the whole model. In this task we model image datasets which are also sampled.

A sample configuration shown below.

```
{
  "activation": "cos",
  "w_init": [
    "glorot_normal",
    null
  ],
  "dataset": [
    "sun397_32x32",
    {
      "bs": 228,
      "just_train": false,
      "num_train": null
    },
    null
  ],
  "hidden_units": [
    21,
    121
  ],
  "num_bijectors": 4
}
```

H.2.11 Quadratic like problems

This task distribution defines a synthetic problem based on a non-linear modification to a quadratic. The dimensionality of the problem is sampled logarithmically between [2, 3000].

The loss for this task is described by:

$$\text{output\_fn}((AX - B)^2 + C) \tag{S31}$$

where $X = \text{param} * \text{weight\_rescale}$ and where param is initialized by initial_dist.sample() / weight_rescale.

The output_fn is sampled uniformly between identity, and $f(x) = \log(\max(0, x))$. The loss scale is sampled logarithmically between [10^{-5}, 10^3].

We define a distribution over matrices A as a sample from one of the following: normal: we sample a mean from a normal draw with a standard deviation of 0.05 and a std from a uniform [0, 0.05]. The elements of A are drawn from the resulting distribution. uniform: linspace_eigen: logspace_eigen:

We define a distribution over B to be either normal with mean and std sampled from N(0, 1), U(0, 2) respectively or uniform with min and range equal to U(-5, 2.5), U(0, 5) respectively.

With probability 50% we add noise from a distribution whose parameters are also sampled.

A sample configuration shown below.

```
{
  "A_dist": [
    "linspace_eigen",
    {
      "min": 32.09618575514275,
      "max": 122.78045861480965
    }
  ],
  "initial_dist": [
    "uniform",
    {
      "min": 2.3911997838130956,
      "max": 6.723940057771417
    }
  ],
  "output_fn": "log",
  "dims": 212,
  "seed": 68914,
  "loss_scale": 0.6030061302850566,
  "noise": null
}
```

H.2.12 RNN Text classification

This task consists of using an RNN to classify tokenized text. We first trim the vocab length to be of a size logarithmically sampled between $[100, 10000]$. The text is then embedded into a vocab size logarithmically sampled between $[8, 128]$. These embeddings get fed into a sampled config RNN. With equal probability the initial state of the rnn is either sampled, or zeros. With equal probability we either take the last RNN prediction, the mean over features, or the per feature max over the sequence. This batch of activations is then passed through a linear layer and a softmax cross entropy loss. The initialization for the linear projection is sampled.

An example configuration shown below. In this version of TaskSet the dataset sampling contains a bug. All data used is from the imdb_reviews/subwords8k dataset.

```
{
  "embed_dim": 111,
  "w_init": [
    "random_normal",
    0.1193048629073732
  ],
  "dataset": [
    "imdb_reviews/subwords8kimdb_reviews/bytes",
    {
      "bs": 43,
      "num_train": null,
```

```
      "max_token": 8185,
      "just_train": true,
      "patch_length": 20
    }
  ],
  "vocab_size": 3570,
  "core": [
    "vrnn",
    {
      "hidden_to_hidden": [
        "he_uniform",
        null
      ],
      "in_to_hidden": [
        "he_uniform",
        null
      ],
      "act_fn": "leaky_relu2",
      "core_dim": 35
    }
  ],
  "trainable_init": false,
  "loss_compute": "max"
}
```

H.3 Fixed Tasks

In addition to sampled tasks, we also define a set of hand designed and hand specified tasks. These tasks are either more typical of what researcher would do (e.g. using default initializations) or specific architecture features such as bottlenecks in autoencoders, normalization, or dropout.

In total there are 107 fixed tasks. Each task is labeled by name with some information about the underlying task. We list all tasks, discuss groups of tasks, but will not describe each task in detail. Please see the source for exact details.

Associative_GRU128_BS128_Pairs10_Tokens50
Associative_GRU256_BS128_Pairs20_Tokens50
Associative_LSTM128_BS128_Pairs10_Tokens50
Associative_LSTM128_BS128_Pairs20_Tokens50
Associative_LSTM128_BS128_Pairs5_Tokens20
Associative_LSTM256_BS128_Pairs20_Tokens50
Associative_LSTM256_BS128_Pairs40_Tokens100
Associative_VRNN128_BS128_Pairs10_Tokens50
Associative_VRNN256_BS128_Pairs20_Tokens50

These tasks use RNN's to perform an associative memory task. Given a vocab of tokens, and some number of pairs to store and a query the RNN's goal is to produce the desired value. For example given the input sequence A1B2C3?B_ the RNN should produce ________B.

This model embeds tokens, applies an RNN, and applies a linear layer to map back to the output space. Softmax cross entropy loss is used to compare outputs. A weight is also placed on the losses so that loss is incurred only when the RNN is supposed to predict. For RNN cells we use LSTM [52], GRU [26], and VRNN – a vanilla RNN. The previous tasks are defined with the corresponding RNN cell, number of units, batch size, sequence lengths, and number of possible tokens for the retrieval task.

Copy_GRU128_BS128_Length20_Tokens10
Copy_GRU256_BS128_Length40_Tokens50
Copy_LSTM128_BS128_Length20_Tokens10
Copy_LSTM128_BS128_Length20_Tokens20
Copy_LSTM128_BS128_Length50_Tokens5
Copy_LSTM128_BS128_Length5_Tokens10
Copy_LSTM256_BS128_Length40_Tokens50
Copy_VRNN128_BS128_Length20_Tokens10
Copy_VRNN256_BS128_Length40_Tokens50

These tasks use RNN's to perform a copy task. Given a vocab of tokens and some number of tokens the RNN's job is to read the tokens and to produce the corresponding outputs. For example an input might be: ABBC|____ and the RNN should output ____|ABBC. See the source for a complete description of the task. Each task in this set varies the RNN core, as well as the dataset structure.

This model embeds tokens, applies an RNN, and applies a linear layer to map back to the output space. Softmax crossentropy loss is used to compare outputs. A weight is also placed on the losses so that loss is incurred only when the RNN is supposed to predict. For RNN cells we use LSTM [52], GRU [26], and VRNN – a vanilla RNN. The previous tasks are defined with the corresponding RNN cell, number of units, batch size, sequence lengths, and number of possible tokens.

FixedImageConvAE_cifar10_32x32x32x32x32_bs128
FixedImageConvAE_cifar10_32x64x8x64x32_bs128
FixedImageConvAE_mnist_32x32x32x32x32_bs128
FixedImageConvAE_mnist_32x64x32x64x32_bs512
FixedImageConvAE_mnist_32x64x8x64x32_bs128

Convolutional autoencoders trained on different datasets and with different architectures (sizes of hidden units).

FixedImageConvVAE_cifar10_32x64x128x64x128x64x32_bs128
FixedImageConvVAE_cifar10_32x64x128x64x128x64x32_bs512
FixedImageConvVAE_cifar10_32x64x128x64x32_bs128
FixedImageConvVAE_cifar10_64x128x256x128x256x128x64_bs128
FixedImageConvVAE_mnist_32x32x32x32x32_bs128
FixedImageConvVAE_mnist_32x64x32x64x32_bs128
FixedImageConvVAE_mnist_64x128x128x128x64_bs128

Convolutional variational autoencoders trained on different datasets, batch sizes, and with different architectures.

FixedImageConv_cifar100_32x64x128_FC64x32_tanh_variance_scaling_bs64
FixedImageConv_cifar100_32x64x64_flatten_bs128
FixedImageConv_cifar100_bn_32x64x128x128_bs128
FixedImageConv_cifar10_32x64x128_flatten_FC64x32_tanh_he_bs8
FixedImageConv_cifar10_32x64x128_flatten_FC64x32_tanh_variance_scaling_bs64
FixedImageConv_cifar10_32x64x128_he_bs64
FixedImageConv_cifar10_32x64x128_largenormal_bs64
FixedImageConv_cifar10_32x64x128_normal_bs64
FixedImageConv_cifar10_32x64x128_smallnormal_bs64
FixedImageConv_cifar10_32x64x128x128x128_avg_he_bs64
FixedImageConv_cifar10_32x64x64_bs128
FixedImageConv_cifar10_32x64x64_fc_64_bs128
FixedImageConv_cifar10_32x64x64_flatten_bs128
FixedImageConv_cifar10_32x64x64_tanh_bs64
FixedImageConv_cifar10_batchnorm_32x32x32x64x64_bs128
FixedImageConv_cifar10_batchnorm_32x64x64_bs128
FixedImageConv_coil10032x32_bn_32x64x128x128_bs128
FixedImageConv_colorectalhistology32x32_32x64x64_flatten_bs128
FixedImageConv_food10164x64_Conv_32x64x64_flatten_bs64
FixedImageConv_food101_batchnorm_32x32x32x64x64_bs128
FixedImageConv_mnist_32x64x64_fc_64_bs128
FixedImageConv_sun39732x32_bn_32x64x128x128_bs128
Mnist_Conv_32x16x64_flatten_FC32_tanh_bs32

Convolutional neural networks doing supervised classification. These models vary in dataset, architecture, and initializations.

FixedLM_lm1b_patch128_GRU128_embed64_avg_bs128
FixedLM_lm1b_patch128_GRU256_embed64_avg_bs128
FixedLM_lm1b_patch128_GRU64_embed64_avg_bs128
FixedLM_lm1b_patch128_LSTM128_embed64_avg_bs128
FixedLM_lm1b_patch128_LSTM256_embed64_avg_bs128

Language modeling tasks on different RNN cell types and sizes.

FixedMAF_cifar10_3layer_bs64
FixedMAF_mnist_2layer_bs64
FixedMAF_mnist_3layer_thin_bs64

Masked auto regressive flows models with different architectures (number of layers and sizes).

FixedMLPAE_cifar10_128x32x128_bs128
FixedMLPAE_mnist_128x32x128_bs128

FixedMLPAE_mnist_32x32x32_bs128

Autoencoder models based on multi layer perceptron with different number of hidden layers and dataset.

FixedMLPVAE_cifar101_128x128x32x128x128_bs128
FixedMLPVAE_cifar101_128x32x128_bs128
FixedMLPVAE_food10132x32_128x64x32x64x128_bs64
FixedMLPVAE_mnist_128x128x8x128_bs128
FixedMLPVAE_mnist_128x64x32x64x128_bs64
FixedMLPVAE_mnist_128x8x128x128_bs128
Imagenet32x30_FC_VAE_128x64x32x64x128_relu_bs256

Variational autoencoder models built from multi layer perceptron with different datasets, batchsizes, and architectures.

FixedMLP_cifar10_BatchNorm_128x128x128_relu_bs128
FixedMLP_cifar10_BatchNorm_64x64x64x64x64_relu_bs128
FixedMLP_cifar10_Dropout02_128x128_relu_bs128
FixedMLP_cifar10_Dropout05_128x128_relu_bs128
FixedMLP_cifar10_Dropout08_128x128_relu_bs128
FixedMLP_cifar10_LayerNorm_128x128x128_relu_bs128
FixedMLP_cifar10_LayerNorm_128x128x128_tanh_bs128
FixedMLP_cifar10_ce_128x128x128_relu_bs128
FixedMLP_cifar10_mse_128x128x128_relu_bs128
FixedMLP_food10132x32_ce_128x128x128_relu_bs128
FixedMLP_food10132x32_mse_128x128x128_relu_bs128
FixedMLP_mnist_ce_128x128x128_relu_bs128
FixedMLP_mnist_mse_128x128x128_relu_bs128
FixedNVP_mnist_2layer_bs64

Image classification based on multi layer perceptron. We vary architecture, data, batchsize, normalization techniques, dropout, and loss type across problems.

FixedNVP_mnist_3layer_thin_bs64
FixedNVP_mnist_5layer_bs64
FixedNVP_mnist_5layer_thin_bs64
FixedNVP_mnist_9layer_thin_bs16

Non volume preserving flow models with different batchsizesm and architectures.

FixedTextRNNClassification_imdb_patch128_LSTM128_avg_bs64
FixedTextRNNClassification_imdb_patch128_LSTM128_bs64
FixedTextRNNClassification_imdb_patch128_LSTM128_embed128_bs64
FixedTextRNNClassification_imdb_patch32_GRU128_bs128
FixedTextRNNClassification_imdb_patch32_GRU64_avg_bs128
FixedTextRNNClassification_imdb_patch32_IRNN64_relu_avg_bs128
FixedTextRNNClassification_imdb_patch32_IRNN64_relu_last_bs128
FixedTextRNNClassification_imdb_patch32_LSTM128_E128_bs128
FixedTextRNNClassification_imdb_patch32_LSTM128_bs128
FixedTextRNNClassification_imdb_patch32_VRNN128_tanh_bs128
FixedTextRNNClassification_imdb_patch32_VRNN64_relu_avg_bs128
FixedTextRNNClassification_imdb_patch32_VRNN64_tanh_avg_bs128

RNN text classification problems with different RNN cell, sizes, embedding sizes, and batchsize.

TwoD_Bowl1
TwoD_Bowl10
TwoD_Bowl100
TwoD_Bowl1000

2D quadratic bowls with different condition numbers.

TwoD_Rosenbrock
TwoD_StyblinskiTang
TwoD_Ackley
TwoD_Beale

Toy 2D test functions.

I Old reviews:

Reviewer 1

Questions

1. Summary and contributions: Briefly summarize the paper and its contributions.

In this work the authors present a very large large meta-dataset for different hyperparameter settings of optimization algorithms for various neural networks trained on a variety of tasks and datasets. Furthermore, a simple algorithm to greedily generate a sequence of well-performing hyperparameter settings that can be applied to new tasks is proposed.

2. Strengths: Describe the strengths of the work. Typical criteria include: soundness of the claims (theoretical grounding, empirical evaluation), significance and novelty of the contribution, and relevance to the NeurIPS community.

The introduced meta-dataset could be of interest for AutoML research Simple yet efficient method that could be easily used by researchers with less expertise.

3. Weaknesses: Explain the limitations of this work along the same axes as above.

In my understanding, the proposed method is not novel. Lack of baselines does not allow for a fair validation of results.

4. Correctness: Are the claims and method correct? Is the empirical methodology correct? The authors need to clarify why their work is different to [126].

5. Clarity: Is the paper well written?

I think the paper would benefit a lot from some additional work. I want to share some things I considered painful when evaluating this work. It is written in a disruptive way and contains many disconnected "patches" of text blocks as if some of the text was added in hindsight. There is an extensive use of subsections, many of which have 7 lines or less. The paper heavily depends on the appendix and requires the user to lookup important knowledge there. Main discussions of experiments are oftentimes found in the captions rather than the text. 6. Relation to prior work: Is it clearly discussed how this work differs from previous contributions? Clear differentiation to [126] is missing. Related work in the main paper is missing. I suggest to replace that part with appendix G3.

7. Reproducibility: Are there enough details to reproduce the major results of this work?

Yes

8. Additional feedback, comments, suggestions for improvement and questions for the authors:

The authors claim that there work is similar to [126]. I fail to understand the difference of these two works. It looks the same as algorithm 1 in [126] to me. Given the strong similarity, the authors should discuss the differences.

Given the described setup, the proposed method would minimize Equation (1) after having used the best optimizers within the set of tasks the methods learns from. After reaching this step, no matter which optimizer will be chosen, the objective function will no longer change. Therefore, optimizers will be chosen at random. The authors should discuss whether this is a desired behavior or whether they believe this would not happen in practice.

This works lacks a useful baseline. Quasi-random or Latin Hypercube sampling would serve as a stronger baseline and would probably have been useful to generate tasks and optimizers as well. Beyond these very simple methods, the authors should also consider methods such as Bayesian optimization or Hyperband. More importantly, a metalearning baseline is missing. Given that the proposed method is very similar to [126], this could serve as one. Given T different tasks, the solution with least optimizers (<=T) to Equation (1) would be to use the optimizers that performed best on each task. In fact, using the best hyperparameter settings on the most similar dataset (with respect to some metafeatures) is a common idea in metalearning (e.g. [39]). How does the proposed method compare to this approach? Or how about randomly selecting from the top optimizer per task instead of all optimizers? It would be useful to add a line in all plots that indicate some sort of default optimizer (as done in Figure 4).

In my opinion, something like appendix G3 should replace the current related work.

Minor:

At one point the reference points to Figure 3 where it should point to 2.

"3.3. Scoring an optimizer by averaging over tasks" - this section describes only how task scoring works. How scoring for an optimizer works I can only infer by the subsection title. It would be useful to mention within the section that scoring an optimizer happens by averaging across all tasks.

What are the refined baselines in Figure 2?

Number of datasets unclear (i.e. all different splits considered). Which datasets are considered is only mentioned in appendix.

================= After Rebuttal =================

I would like to thank the authors for their clarifications in their answer and I think the proposed changes will be one good step towards a better version of this work. I think the value of the metadataset will have big impact but the paper requires some more work. I think it would be a good idea to survey different methods that would benefit from this data. In order to show the benefit empirically, the strongest hyperparameter optimization methods that are not able to use this data should be considered as baselines.

9. Please provide an "overall score" for this submission.

4: An okay submission, but not good enough; a reject.

10. Please provide a "confidence score" for your assessment of this submission.

5: You are absolutely certain about your assessment. You are very familiar with the related work.

11. Have the authors adequately addressed the broader impact of their work, including potential negative ethical and societal implications of their work?

Yes

Reviewer 2

Questions

1. Summary and contributions: Briefly summarize the paper and its contributions.

This paper proposes a hyper-parameter search algorithm via meta-learning which shows better sample efficiency. In addition, authors propose TaskSet which is a dataset including 1k diverse tasks (CNNs, RNNs, etc) for the study of hyper-parameter search. They also explore the generalization ability on ImageNet classification with Resnet50 and LM1B LM with transformers.

2. Strengths: Describe the strengths of the work. Typical criteria include: soundness of the claims (theoretical grounding, empirical evaluation), significance and novelty of the contribution, and relevance to the NeurIPS community.

This paper attempts to address hyper-parameter search which is a very important problem. It can reduce the effort of researchers on tuning hyper-parameters manually for a specific task. Authors promise to release code in multiple frameworks and the learned hyper-prparameter list is expected to be easily applied to any arbitrary models. In addition, the paper is well written and organized

3. Weaknesses: Explain the limitations of this work along the same axes as above.

1) My main concern is that tasks used for training and evaluation are realistic, but not large scale, which might limit the contribution of the work.

2) Compare with bayesian optimization algorithms (e.g., spearmint, DNGO [1], or more advanced) for hyper-parameter tuning?

3) Any chances to learn zero-shot hyper-parameter predictor in order to scale up?

[1] Scalable Bayesian Optimization Using Deep Neural Networks (http://proceedings.mlr.press/v37/snoek15.pdf)

4. Correctness: Are the claims and method correct? Is the empirical methodology correct?

yes

5. Clarity: Is the paper well written?

yes

6. Relation to prior work: Is it clearly discussed how this work differs from previous contributions? not enough

7. Reproducibility: Are there enough details to reproduce the major results of this work?

No

8. Additional feedback, comments, suggestions for improvement and questions for the authors: I was concerned about the baseline comparison. As other reviewers pointed out, the main contribution of this paper is dataset. I raise score after reading rebuttal but the work is not quite solid to be accepted as I expected. Hope authors can further improve it by incorporating feedback from reviewers.

9. Please provide an "overall score" for this submission.

5: Marginally below the acceptance threshold.

10. Please provide a "confidence score" for your assessment of this submission.

3: You are fairly confident in your assessment. It is possible that you did not understand some parts of the submission or that you are unfamiliar with some pieces of related work. Math/other details were not carefully checked.

11. Have the authors adequately addressed the broader impact of their work, including potential negative ethical and societal implications of their work?

Yes

Reviewer 3

Questions

1. Summary and contributions: Briefly summarize the paper and its contributions.

The paper presents a dataset of tasks (TaskSet) fro use in training and evaluating optimization algorithms and their hyperparameters. This task set mostly consists of neural network models. Most of the tasks are randomly generated and grouped into image models, languages models, quadratic, etc. TaskSet also includes 107 hand designed tasks, which consist of more common tasks that both improve the coverage beyond the sampled tasks.

The paper proposes a simple method for learning hyperparameter lists based on TaskSet. Those hyperparamter lists can be used as hyperparameter values when training models on different datasets,

The experimental results show that learning hyperparameter lists are more effective that random search, more tasks lead to better generalization. Also, learned optimizer list outperforms both learning rate tuned Adam and default training hyperparamater for ResNet50 and a Transformer model.

2. Strengths:

Describe the strengths of the work. Typical criteria include: soundness of the claims (theoretical grounding, empirical evaluation), significance and novelty of the contribution, and relevance to the NeurIPS community. Meta learning is an interesting and challenge topic.

The TaskSet created by the authors is a valuable resource for the research community.

3. Weaknesses: Explain the limitations of this work along the same axes as above.

The study is trying to provide optimizer/hyperparameter list for all types of models. But most likely, different type of models have different set of good hyperparameters. Why not produce optimizer suggestions based model types: image classification, language model, etc. ?

To use this TaskSet, the computing resource requirement is daunting.

4. Correctness: Are the claims and method correct? Is the empirical methodology correct?

I think so.

5. Clarity: Is the paper well written?

yes,

6. Relation to prior work: Is it clearly discussed how this work differs from previous contributions?

The paper reviews previous research.

7. Reproducibility: Are there enough details to reproduce the major results of this work?

Yes

8. Additional feedback, comments, suggestions for improvement and questions for the authors:

Thanks for the authors for answering my questions. It is no double that the Task set collected will be vert useful to the community. It is good to know that the total running time is not so significant as I expected. Maybe it is a good idea to stress that the contribution of this paper is not the performance of hyperparameter setting, but the TaskSet itself, and also do more competitive baseline comparison study.

9. Please provide an "overall score" for this submission.

6: Marginally above the acceptance threshold.

10. Please provide a "confidence score" for your assessment of this submission.

4: You are confident in your assessment, but not absolutely certain. It is unlikely, but not impossible, that you did not understand some parts of the submission or that you are unfamiliar with some pieces of related work.

11. Have the authors adequately addressed the broader impact of their work, including potential negative ethical and societal implications of their work?

Yes

Reviewer 4

Questions

1. Summary and contributions: Briefly summarize the paper and its contributions.

This paper proposes a dataset of many optimization problems to assist the research in learning to optimize. The main idea is that it is better to learn to optimize on a collection of tasks, so that the learned optimizer can be transferrable to other tasks.

2. Strengths: Describe the strengths of the work. Typical criteria include: soundness of the claims (theoretical grounding, empirical evaluation), significance and novelty of the contribution, and relevance to the NeurIPS community.

1. the idea is intuitive and dataset design process is clearly motivated 2. learned optimizers are applied on both image classification and language modeling 3. the problem addressed seems important

3. Weaknesses: Explain the limitations of this work along the same axes as above.

not much that i can spot

4. Correctness: Are the claims and method correct? Is the empirical methodology correct?

the methods seem correct to me

5. Clarity: Is the paper well written?

The paper is very well written.

6. Relation to prior work: Is it clearly discussed how this work differs from previous contributions?

The related work is clearly discussed, although I'm not an expert in the area so I might missed something.

7. Reproducibility: Are there enough details to reproduce the major results of this work?

Yes

9. Please provide an "overall score" for this submission.

6: Marginally above the acceptance threshold.

10. Please provide a "confidence score" for your assessment of this submission.

2: You are willing to defend your assessment, but it is quite likely that you did not understand central parts of the submission or that you are unfamiliar with some pieces of related work. Math/other details were not carefully checked.

11. Have the authors adequately addressed the broader impact of their work, including potential negative ethical and societal implications of their work? Yes

J What we did in response

This paper was always meant to be a paper about a dataset of tasks. We clarified our contributions as well as misc. edits to ensure that this comes through.