A Hyperparameter Optimization Toolkit for Neural Machine Translation Research

Xuan Zhang and **Kevin Duh** and **Paul McNamee**
Johns Hopkins University
Baltimore, Maryland, USA
xuanzhang@jhu.edu; kevinduh@cs.jhu.edu; mcnamee@jhu.edu

Abstract

Hyperparameter optimization is an important but often overlooked process in the research of deep learning technologies. To obtain a good model, one must carefully tune hyperparameters that determine the architecture and training algorithm. Insufficient tuning may result in poor results, while inequitable tuning may lead to exaggerated differences between models. We present a hyperparameter optimization toolkit for neural machine translation (NMT) to help researchers focus their time on the creative rather than the mundane. The toolkit is implemented as a wrapper on top of the open-source Sockeye NMT software. Using the Asynchronous Successive Halving Algorithm (ASHA), we demonstrate that it is possible to discover near-optimal models under a computational budget with little effort.[1]

1 Introduction

Deep learning models are difficult to train. Although they achieve impressive results on many tasks, non-trivial amounts of effort are required for selecting appropriate hyperparameters, such as the number of layers, vocabulary size, embedding dimension, and optimization algorithm. This trial-and-error process is necessary for each task, domain, or language. Further, the rapid development of new neural network architectures implies that this hyperparameter optimization process will only become more expensive.

Currently, hyperparameter optimization tends to be performed manually by researchers in an ad hoc fashion, using scripts put together independently. The lack of open-source support tools means that the level of rigor in hyperparameter optimization may vary widely. This poses two risks:

1. **Insufficient exploration** of the hyperparameter space may lead to poor results, killing an otherwise promising research idea.

2. **Inequitable allocation** of compute resources for hyperparameter optimization of one model over another may lead to exaggerated results differences and misleading conclusions.

The importance of documenting the hyperparameter optimization process in research has already been widely recognized and is included as an item under the "Responsible NLP Checklist"[2] required for paper submissions in the field. To support these efforts, we believe it will be beneficial to develop open-source tools to improve the hyperparameter optimization process itself.

This paper presents a *hyperparameter optimization toolkit* for NMT research. It enables researchers to easily explore the hyperparameter space of various NMT models based on the PyTorch codebase of AWS Sockeye framework (Hieber et al., 2022). One simply specifies (1) the desired set of hyperparameter options to search, (2) the compute resource constraints, and (3) the training data paths, then the toolkit will plan and execute an automatic hyperparameter optimization and return the best model discovered. The toolkit implements the Asynchronous Successive Halving Algorithm (ASHA) (Li et al., 2020), which is well-suited for commodity off-the-shelf distributed grids.

In the following, we first give an overview of the toolkit (Section 2) and hyperparameter optimization algorithm (Section 3). Then, the case study in Section 4 illustrates how the toolkit can help a researcher search over thousands of hyperparameter configurations with ease. Finally, Section 5 discusses our design choices, hopefully serving as a

[1] https://github.com/kevinduh/sockeye-recipes3 (code), https://cs.jhu.edu/~kevinduh/j/demo.mp4 (video demo)

[2] https://aclrollingreview.org/responsibleNLPresearch/

Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics
Volume 3: System Demonstrations, pages 161–168

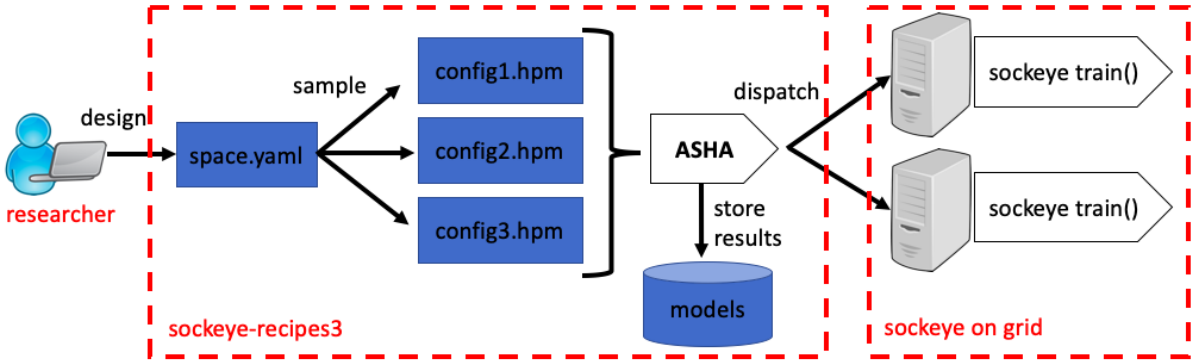

Figure 1: An overview of the `sockeye-recipes3` hyperparameter optimization toolkit

reference for those who want to implement similar toolkits for different NLP software.

2 Usage Overview

Our hyperparameter optimization toolkit is named `sockeye-recipes3`, since it cooks up different models by training models with the AWS Sockeye NMT framework, version 3. An overview is shown in Figure 1. For concreteness, let us suppose the researcher in Figure 1 wants to run a rigorous hyperparameter optimization to obtain a strong Transformer baseline for a new dataset.

Step 1: The researcher designs a hyperparameter search space for his/her model. Table 1 shows some common hyperparameters for Transformers, but the toolkit is flexible to incorporate any user-defined hyperparameter. This hyperparameter space is expressed as a YAML file, e.g. `space.yaml`:

```
transformer_model_size: [256, 512, 1024]
transformer_attention_heads: 8
transformer_feed_forward_num_hidden: [1024, 2048]
...
```

The snippet above indicates that the researcher wishes to explore three choices for model size, one choice for attention head, and two choices for a feed-forward number of hidden units. The Cartesian product of all these choices forms the full hyperparameter space.

Step 2: `sockeye-recipes3` samples from the full hyperparameter space to generate a set of bash files called `hpm` files. Each `hpm` file represents a *specific* hyperparameter configuration and encapsulates all the information needed to train a model. This includes not only hyperparameter settings but also paths to training and validation data. For example, `config1.hpm` might train a model with:

```
transformer_model_size=256
transformer_attention_heads=8
transformer_feed_forward_num_hidden=1024
train_data=~/data/wmt.train.de-en.bitext
validation_data=~/data/wmt.dev.de-en.bitext
```

The set of `hpm` files represents all the hyperparameter configurations to be explored by the hyperparameter optimization algorithm. Rather than randomly sampling a subspace, one can also generate the full Cartesian product or manually edit some `hpm` files based on prior knowledge. Depending on the researcher's usage scenario, this set typically numbers from tens to thousands.

Step 3: Once the researcher is ready, he/she starts the ASHA program with resource specifications such as the number of concurrent GPUs to use and the number of checkpoints per training run. This Python code dispatches the training processes as standard Sockeye jobs to a distributed grid.[3] ASHA will attempt to efficiently train as many models as possible given the computational constraints. It is a bandit learning method that automatically learns when to stop a not-so-promising training run in order to allocate resources to other hyperparameter configurations. Details are in Section 3.

Step 4: The results of all Sockeye training runs dispatched by ASHA are stored on disk. Each `hpm` file will have a corresponding subdirectory with the output log of a Sockeye training process. This makes it easy to replicate or continue any training runs in the future, with or without the `sockeye-recipes3` toolkit. Ultimately, the researcher can pick out the best model from the set for further experimentation.

[3]The dispatch in `sockeye-recipes3` is currently implemented for the Univa Grid Engine (UGE) but is easily extendable to other similar grid management software like SLURM.

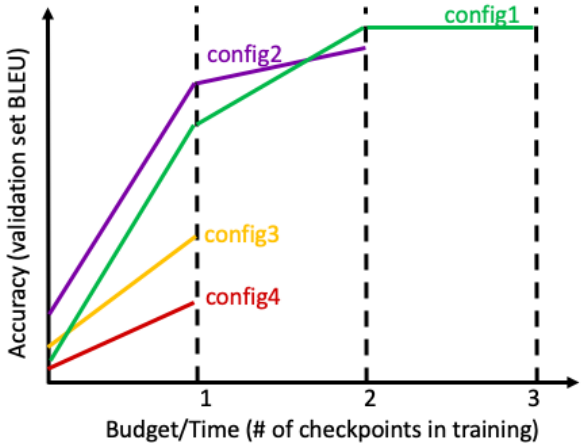

Figure 2: Illustration of Successive Halving

Additional features: (a) Metric: The toolkit's default is to find models with high BLEU on the validation set. This can be changed to any user-specified metric. Also, we have devised a multi-objective version of ASHA to enable joint optimization of accuracy and inference speed based on Pareto optimality (Marler and Arora, 2004).

(b) Analysis: After an ASHA run, one may wish to see if there are certain trends in hyperparameters, e.g. are some more important than others? This introspection can be helpful in understanding the model or designing future hyperparameter spaces. We have included a tool for posthoc analysis using Explainable Boosting Machines (Deb et al., 2022).

3 Hyperparameter Opt. with ASHA

Problem: Suppose we have N hyperparameter configurations (`hpm` files) and a max compute budget of B, measured in terms of the total number of training checkpoints available. Let us select n configurations for actual training, where $n \leq N$. If each configuration is allocated the same budget, then each would be trained up to B/n checkpoints. When N is large, we have an untenable problem:

- If we choose n to be large (close to N), then B/n will be small, indicating that each configuration is only trained for a few checkpoints. Most models likely will not have converged.
- If we choose n to be small (despite N being large), then configurations that are chosen are trained well (large B/n) but the majority of configurations are not even trained at all.

The only solution is to allocate each configuration with a variable budget: i.e. train promising configurations for more checkpoints and terminate the not-so-promising ones prior to convergence. This is an intuitive idea that has probably been performed countless times by researchers by tracking learning curves in a manual fashion.

Successive Halving: The Successive Halving Algorithm (Jamieson and Talwalkar, 2016) implements this intuition algorithmically, and is illustrated in Figure 2. Suppose we choose $n = 4$ hyperparameter configurations to explore and the total budget is $B = 7$ checkpoints. We begin by first training each configuration up to checkpoint 1 and measuring their validation accuracy. The configurations with lower accuracies at this point (config3, config4) are deemed not-so-promising and are terminated. The remaining half (config1, config2) are trained longer, and validation accuracy is measured again at checkpoint 2. Again, half of the configurations are terminated and the other half is "promoted" to be trained longer; this is done successively until the total budget is reached.

The main assumption of Successive Halving is that learning curves of different configurations are comparable and that the relative ranking of validation accuracy at intermediate checkpoints correlates to that at convergence. This is an assumption that cannot be proved but is likely reasonable in most cases with the proper setting of checkpoint intervals.

ASHA: In practice, the Successive Halving Algorithm as described above has a bottleneck at each checkpoint: we need to wait for all configurations to return their validation score before deciding the best half to promote. The actual time that a configuration needs to reach a checkpoint depends on many factors such as GPU device type and model size. So we may end up waiting for the slowest training run, causing poor grid utilization.

To address this, an Asynchronous Successive Halving Algorithm (ASHA) is introduced (Li et al., 2020). The idea is to promote a configuration as soon as it is guaranteed to be in the top half, without waiting for all configurations to return with their checkpoints' validation accuracy. For example in Figure 2, suppose three configurations (e.g. config2, config3, config4) have already returned an accuracy for checkpoint 1. We are then safe to promote the best one out of the group (config2) without waiting for config1 to return since config2 will be among the top half regardless of config1's accuracy.

Name & Description	Settings
Architecture Hyperparameters	
transformer_model_size - size of model/embeddings	{256, 512, 1024}
transformer_attention_heads - # of heads	8
transformer_feed_forward_num_hidden - # units in feedforward layer	{1024, 2048}
num_layers - for "encoder:decoder"	{6:6, 8:4, 4:4, 6:2}
Data Pre-processing Hyperparameters	
bpe_symbols_src - # of BPE symbols on source side	{5k, 10k, 30k}
bpe_symbols_trg - # of BPE symbols on target side	{5k, 10k, 30k}
Training Hyperparameters	
optimized_metric	perplexity
initial_learning_rate: initial rate for ADAM optimizer	{0.0002, 0.001, 0.002}
embed_dropout - dropout rate for source:target embeddings	.0:.0
label_smoothing	0.1
seed - random initialization seed	{1, 2}
Hardware-related Hyperparameters	
batch_size - # of words in batch	4096
checkpoint_interval - #batches before saving checkpoint to disk	4000

Table 1: Hyperparameter space used in the case study. The settings in red font are searched over, while others are held fixed. In total, we will explore $3 \times 2 \times 4 \times 3 \times 3 \times 3 \times 2 = 1296$ configurations.

Please refer to the original papers on ASHA, Successive Halving, and a variant called Hyperband (Li et al., 2016) for more detailed analyses. We focus on ASHA in `sockeye-recipes3`.

4 Case Study

Goal: To illustrate how `sockeye-recipes3` works in practice, we show a case study on building a strong Transformer baseline for a new Telugu-to-English dataset. Our initial training set consists of 900k lines of bitext obtained from public sources via the OPUS portal (Tiedemann, 2012). This is augmented with 7 million lines of back-translated data obtained by running a reverse system (English-to-Telugu NMT trained on 900k) on web-scraped news from the Leipzig corpus (Goldhahn et al., 2012). 3000 lines are held out from the initial training set to serve as the validation set.

Given this setup, our goal is to run hyperparameter optimization on a standard Transformer architecture to obtain the best possible model according to validation BLEU. This model can serve as a strong baseline for any future NMT experiment based on the same dataset. Since this is a low-resource language pair that is relatively unexplored in the research community, we opt to search a large hyperparameter space.

Hyperparameter space: Our `space.yaml` file is defined according to the options listed in Table 1. While any user-defined hyperparameter is possible, `sockeye-recipes3` exposes the most common options. We explore a total of 1296 configurations.

ASHA run: We run ASHA using the resource settings in Table 2. The reduction rate decides the fraction of configurations that are promoted each time: a factor p=2 reduction rate corresponds to “halving", but in practice, one can choose to be more or less aggressive. We also specify the number of GPUs that can be used concurrently by ASHA: here, it will dispatch jobs asynchronously up to that limit of G=40.

Finally, the settings for a min, max, and per-rung checkpoints are NMT-specific modifications we found useful for ASHA. In Figure 2, halving is performed at each checkpoint, or at each “rung" in ASHA terminology. It is convenient to give NMT researchers the flexibility to choose the exact schedule: here, we decide that each configuration is trained for at least r=5 checkpoints (corresponding to 5×4000 batches due to the `checkpoint_interval` in Table 1) before we perform successive halving at the first rung. Thereafter, each configuration is trained for u=2 checkpoints before successive halving is performed. Finally, no configurations will be trained with more than R=25

Reduction rate. Top 1/p promoted	p=2
# of GPUs available	G=40
min checkpoints per model	r=5
#checkpoints per config per rung	u=2
max checkpoints per model	R=25

Table 2: ASHA settings for case study

checkpoints regardless of other ASHA settings; this small number of maximum checkpoints will probably not obtain state-of-the-art results but is suitable for the purpose of discovering several good configurations. The researcher may first inspect the ASHA results to identify several promising configurations, then manually train them for longer.[4]

Figure 3 samples a few learning curves (out of the 1296 configurations in total) to demonstrate how ASHA works in practice. The top figure is analogous to Successive Halving in Figure 2, while the bottom figure shows how the asynchronous dispatch occurs over time.

Comparison with grid search: To confirm whether ASHA finds good models, we also run a grid search on the same 1296 configurations, training each with up to 25 checkpoints. This corresponds to a total cost of $25 \times 1296 = 32,400$. In comparison, the ASHA run in our case study costs 60% less at 9066 checkpoints in total.

Table 3 confirms that ASHA can find good models that are found by an exhaustive grid search. For example, the maximum BLEU score by grid search is 20.3, and while this model is terminated at rung 4, the final model discovered by ASHA has a competitive BLEU score of 20.1. In our experience, ASHA is effective at finding a set of reasonable models at a fraction of the computational cost; if we desire the best possible model, nothing can replace the manual effort of an experienced researcher.

rung	ckpt	config	budget	med	max
0	5	1296	6480	0.3	20.3
1	7	648	7776	17.2	20.3
2	9	324	8424	18.9	20.3
3	11	162	8748	19.4	20.3
4	13	81	8910	19.7	20.3
5	15	40	8990	19.7	20.1
6	17	20	9030	19.7	20.1
7	19	10	9050	19.8	20.1
8	21	5	9060	19.8	20.1
9	23	2	9064	20.0	20.1
10	25	1	9066	20.1	20.1

Table 3: ASHA vs. Grid search: Each row lists the # of configurations explored in each rung, # of checkpoints (ckpt) trained so far per configuration, and accumulated budget (total checkpoints). The med/max columns are median/max BLEU scores among the configurations explored if they were trained to completion in a grid search. For example, in rung 2, 324 configurations were explored by ASHA and trained up to 9 checkpoints. If they were trained up to the full 25 checkpoints and their BLEU scores were collected, the median would be 18.9 and the max would be 20.3. ASHA preserves many of the top configurations that would be found by grid search.

5 Design

`sockeye-recipes3` is designed with two principles: (1) All NMT codes, such as a researcher's proposed extension of the Sockeye framework, are encapsulated in separate conda environments. (2) All hyperparameters and data paths (for baseline and proposed methods) are explicitly specified in `hpm` files, and stored together with each sockeye training run. This means that it is easy to replicate or continue any training run by referring to (1) and (2). ASHA dispatches will run Sockeye training for u checkpoints at a time, so a job will automatically return the GPU resource at the end of each rung.

The ASHA implementation is a Python script that sits on a single server and regularly checks the status of Sockeye training runs on the distributed grid setup. The pseudocode is shown in Algorithm 1. The script keeps track of configurations that are training or paused at a checkpoint. When there is an idle GPU, it will decide whether to explore a new `hpm` or promote an existing one. The dispatch is a job submission command that starts a Sockeye train process on a GPU node. It depends only on the conda-environment provided, so it is easy to optimize different NMT implementations by exchanging the environment while keeping similar `space.yaml`, leading to equitable tuning.

6 Related Work

ASHA and variants can be viewed as bandit algorithms that balance exploration (trying new configurations) with exploitation (training the current configurations for longer). They obtain efficiency

[4]The best model discovered has 8 encoder layers, 4 decoder layers, 1024 model size, 2048 feedforward size, 10k source subwords, 30k target subwords, and achieves 35.6 spBLEU on the FLORES101 devtest (Goyal et al., 2022).

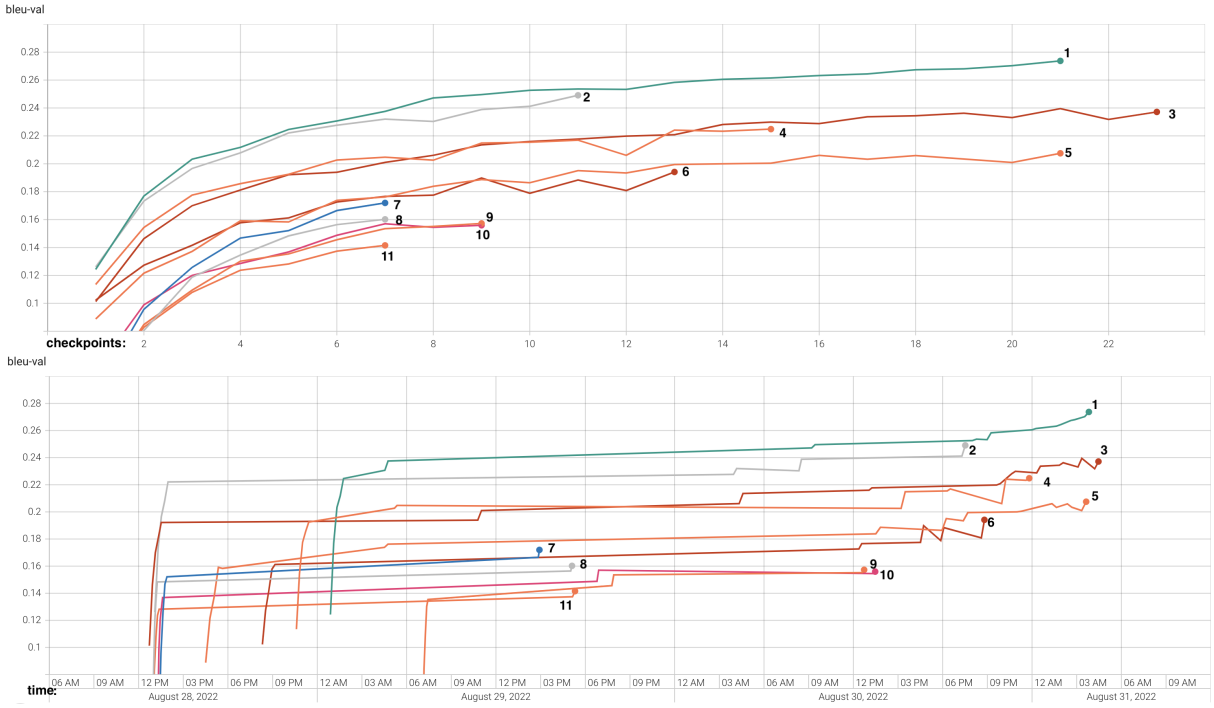

Figure 3: Learning curves for a random sample of configurations in ASHA. The y-axis is the validation BLEU score. The top figure, where the x-axis represents # of checkpoints, is analogous to Figure 2 and shows which configurations are promoted. The bottom figure represents the same configurations plotted against wallclock time on the x-axis; this illustrates the asynchronous nature of ASHA. Observe that configurations are not started in sync, and long plateaus indicate when ASHA decided to pause the configuration at a checkpoint to allocate GPUs for other ones.

Algorithm 1 ASHA pseudocode

```
while budget remains do
    for all c ∈ configs do
        s = check_state(c)   ▷ Still training or at checkpoint?
    end for
    for all g ∈ idle GPU do
        h = get_hpm(configs)   ▷ Explore new or promote?
        dispatch(h, g, conda-env)   ▷ Sockeye train()
    end for
    pause for m minutes
end while
```

by early stopping. Another class of methods are "blackbox" optimizers, e.g. Bayesian Optimization and Evolutionary Methods (Feurer and Hutter, 2019): they treat hyperparameters as input features, observed accuracy as output targets, and train a proxy model to predict new hyperparameters that are worth sampling. These two classes of methods can be combined (Falkner et al., 2018); this is potentially future work. Several benchmarks provide comparisons of state-of-the-art (Zhang and Duh, 2020; Zöller and Huber, 2021).

Neural Architecture Search (Elsken et al., 2019) is related to hyperparameter optimization but focuses more on fine-grained choices (e.g. changing skip connections at different layers). This is an active area of research, but out-of-scope for our purpose of improving NMT experimentation.

There are some existing toolkits like Vizier (Song et al., 2022) and Ray Tune (Liaw et al., 2018), which are suitable for those wanting general rather than application-specific solutions.

7 Conclusions

There is a progression of toolkit development that enables researchers to do better work. Deep learning toolkits like PyTorch and Tensorflow made it easy to exploit GPU hardware. Application-specific toolkits like Sockeye and Fairseq build on top of that, and enabled researchers to quickly prototype new ideas. Further on top, we believe that hyperparameter optimization toolkits and experiment management toolkits in general will further help advance the speed and rigor of research.

We presented `sockeye-recipes3`, an open-

source hyperparameter optimization toolkit for NMT research. Our hope is this will relieve some of the mundane aspects of manual hyperparameter tuning so that researchers can focus on more creative activities. A rigorous and automated hyperparameter optimization process will also lead to more trustworthy experiment results.

Limitations

Scope of support: The `sockeye-recipes3` toolkit only supports the AWS Sockeye NMT framework. It is suitable for researchers who plan to implement and test out different NMT models in PyTorch using Sockeye's codebase. It is not meant to be extensible to hyperparameter optimization methods for other frameworks in NLP. The reason is that each toolkit has its own nuanced error messages and hyperparameter definitions, so it is easier to do design a focused toolkit.

No guarantees: In general, hyperparameter optimization methods give no theoretical guarantees; there is always an aspect of uncertainty. For example, there is no guarantee that ASHA will keep the top configurations if the learning curves do not follow our assumptions. One may be more conservative by setting more checkpoints per rung in ASHA, but this decreases the potential for efficiency.

Manual design: `sockeye-recipes3` does not fully automate the entire model-building process. The researcher still needs to design the hyperparameter space for each task. This search space is critical for the success of ASHA that follows. One may imagine a transfer learning (or meta-learning) approach where hyperparameter spaces from similar tasks are borrowed, but this is currently an open problem.

Ethics Statement

Automated hyperparameter optimization can lead to efficiencies in model building, but we need to be cognizant that there is also a risk of excessive optimization. The user needs to design what is a reasonable search space: for example, would it be worthwhile to optimize over many different random initialization seeds or over small differences between model sizes?

Excessive optimization poses three risks: First, one may select models that "overfit", though this can be ameliorated by proper choices of validation sets. Second, hyperparameter optimization gives an advantage to research teams with large compute resources; ASHA and similar methods are not useful on grids with less than e.g. 10 GPUs.

Third and perhaps more important, the computation may be wasteful. "Green AI" is an important call-to-arms for the research community: hyperparameter optimization is a double-edged sword in that proper usage leads to efficiency while excessive usage leads to wastefulness.

For example, to quantify the CO_2e emissions in our case study, we estimate that ASHA and grid search spent a total of 3050 hours on GPU compute node. Our grid contains a mix of NVIDIA TITAN RTX, GeForce RTX 2080 Ti, and Tesla V100. In future versions of `sockeye-recipes3`, we plan to track power use individually for all jobs but let us assume an average power consumption of 250 watts, for a total of 0.762MWh. If we assume carbon efficiency[5] is at 432 kg CO_2e per MWh, data center power usage effectiveness (PUE) is 1.5, and there are no additional offsets for renewable energy, we end up with:

$$\frac{0.762\ \text{MWh}}{1} \times \frac{432\ \text{kg}}{\text{MWh}} \times \frac{1.5}{1} = 494\ \text{kg}\ CO_2\text{e} \quad (1)$$

This corresponds to the CO_2e of driving a car for 2000km or burning 247kg of coal. Ideally, we will eventually reach an understanding as a community of what amount of use is appropriate or excessive.

References

Kiron Deb, Xuan Zhang, and Kevin Duh. 2022. Post-hoc interpretation of transformer hyperparameters with explainable boosting machines. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 51–61, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21.

Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*.

Matthias Feurer and Frank Hutter. 2019. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer.

Dirk Goldhahn, Thomas Eckart, and Uwe Quasthoff. 2012. Building large monolingual dictionaries at the

[5] https://mlco2.github.io/impact/

Leipzig corpora collection: From 100 to 200 languages. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 759–765, Istanbul, Turkey. European Language Resources Association (ELRA).

Naman Goyal, Cynthia Gao, Vishrav Chaudhary, Peng-Jen Chen, Guillaume Wenzek, Da Ju, Sanjana Krishnan, Marc'Aurelio Ranzato, Francisco Guzmán, and Angela Fan. 2022. The Flores-101 evaluation benchmark for low-resource and multilingual machine translation. *Transactions of the Association for Computational Linguistics*, 10:522–538.

Felix Hieber, Michael Denkowski, Tobias Domhan, Barbara Darques Barros, Celina Dong Ye, Xing Niu, Cuong Hoang, Ke Tran, Benjamin Hsu, Maria Nadejde, Surafel Lakew, Prashant Mathur, Anna Currey, and Marcello Federico. 2022. Sockeye 3: Fast neural machine translation with pytorch.

Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*, pages 240–248. PMLR.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A system for massively parallel hyperparameter tuning. In *Proceedings of Machine Learning and Systems*, volume 2, pages 230–246.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet S. Talwalkar. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*.

R. Marler and Jasbir Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26:369–395.

Xingyou Song, Sagi Perel, Chan Lee Lee, Greg Kochanski, and Daniel Golovin. 2022. Open source vizier: Distributed infrastructure and api for reliable and flexible blackbox optimization. *ArXiv*, abs/2207.13676.

Jörg Tiedemann. 2012. Parallel data, tools and interfaces in OPUS. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 2214–2218, Istanbul, Turkey. European Language Resources Association (ELRA).

Xuan Zhang and Kevin Duh. 2020. Reproducible and efficient benchmarks for hyperparameter optimization of neural machine translation systems. *Transactions of the Association for Computational Linguistics*, 8:393–408.

Marc-André Zöller and Marco F. Huber. 2021. Benchmark and survey of automated machine learning frameworks. *J. Artif. Int. Res.*, 70:409–472.