Towards Efficient and Reproducible Natural Language Processing

Jesse Dodge

CMU-LTI-20-004

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
www.lti.cs.cmu.edu

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
In Language and Information Technologies

Contents

Acknowledgements

I've been supremely fortunate to be have interacted with some fantastic people throughout my PhD. In research we all stand on the shoulders of giants; I've worked with some directly.

Special thanks to Noah Smith, my advisor, without whom this wouldn't have been possible. My committee of Pradeep Ravikumar, Barnabás Póczos, and especially Kevin Jamieson gave me valuable feedback on the work in this thesis. Thanks to my fellow Noah's ARK members, including Maarten Sap, Roy Schwartz, Dani Yogatama, Dallas Card, Sam Thomson, Swabha Swayamdipta, Ana Marasović, Brendan O'Connor, David Bamman, Waleed Ammar, Yanchuan Sim, Lingpeng Kong, Jeffrey Flanigan, Sofia Serrano, Nathan Schneider, Ofir Press, Hao Peng, Phoebe Mulcaire, Lucy Lin, Elizabeth Clark, Emily Gade, Yangfeng Ji, Chenhao Tan, Suchin Gururangan, and Rohan Ramanath.

I had many stellar colleagues at CMU, including Chris Dyer, Bill McDowell, Naomi Saphra, Xinlei Chen, Colin White, Kirthevasan Kandasamy, Anthony Platanios, Calvin Murdock, Willie Neiswanger, Micol Marchetti-Bowick, Nicole Rafidi, and Manaal Faruqui. A callout is deserved to the various people that sat in UW's AI lab while it existed between 2011 and 2019, including Tony Fader, Lydia Chilton, Morgan Dixon, Abe Friesen, Chloé Kiddon, Sam Ainsworth, and Nelson Liu. UW also brought me Kenton Lee, Luke Zettlemoyer, Justine Sherry Martins, Dan Garrette, Alan Ritter, Mark Yatskar, Mike Lewis, Nicholas FitzGerald, Eunsol Choi, Cynthia Matuszek, Marc Deisenroth, Yejin Choi, Yoav Artzi, Gabriel Ilharco, Gabriel Stanovsky, Mitchell Wortsman, Sarah Pratt, Vivek Ramanujan, Peter Guttorp, and June Morita. Some other brilliant colleagues I've had along the way include David Lopez-Paz, Emily Denton, Felix Hill, Robert Nishihara, Jason Weston, Antoine Bordes, Martín Arjovsky, Elad Eban, Karl Sratos, Jacob Andreas, Hal Daume III, Oren Etzioni, and Meg Mitchell.

Most of all, my friends and family have unfailing supported me throughout this journey, including Clea Hersperger, Clayton Hibbert, Jenny Abrahamson, Alex Ford, Nici Bissonnette, Liz Wallace, and Craig Rice. Luke and Caitlyn, I coludn't have done it without you.

This one's for you, Mom and Dad.

Chapter 1

Introduction

Machine learning has reported remarkable progress on a broad range of tasks, including machine translation, object recognition, and game playing (Shoham et al., 2018). Much of this progress has been achieved by increasingly large and computationally-intensive deep learning models. Figure 1.1, reproduced from Amodei and Hernandez (2018), plots training cost increase over time for state-of-the-art deep learning models starting with AlexNet in 2012 (Krizhevsky et al., 2012) to AlphaZero in 2017 (Silver et al., 2017a). The chart shows an overall increase of 300,000x, with training cost doubling every few months. An even sharper trend can be observed in NLP word embedding approaches by looking at ELMo (Peters et al., 2018) followed by BERT (Devlin et al., 2019), openGPT-2 (Radford et al., 2019), XLNet (Yang et al., 2019), Megatron-LM (Shoeybi et al., 2019), and T5 (Raffel et al., 2019).

This trend is driven by the strong focus of the AI community on obtaining state-of-the-art results, as exemplified by the popularity of leaderboards (Wang et al., 2019b,a) which typically report performance[1] but omit any mention of cost or efficiency. Despite the clear benefits of improving model performance, the focus on one single metric ignores the economic, environmental, and social cost of reaching the reported results.

This increase in computational expense has led to some types of research being prohibitively expensive, raising barriers to participation. In addition, recent research estimates a significant carbon footprint for NLP experiments (Strubell et al., 2019). This thesis advocates for increasing research activity in Green AI—AI research that is more efficient, inclusive, and environmentally friendly. We emphasize that Red AI has been yielding valuable scientific contributions, but it has been overly dominant in driving the direction of research in the field; we want to shift the balance towards the Green AI *option*. Specifically, the work in this thesis makes efficiency a evaluation criterion alongside accuracy and related measures.

AI research can be computationally expensive in a number of ways, but each provides

[1]In this thesis "performance" refers to accuracy, precision, and related measures, and not runtime or other metrics of computational efficiency (for which we use other terms).

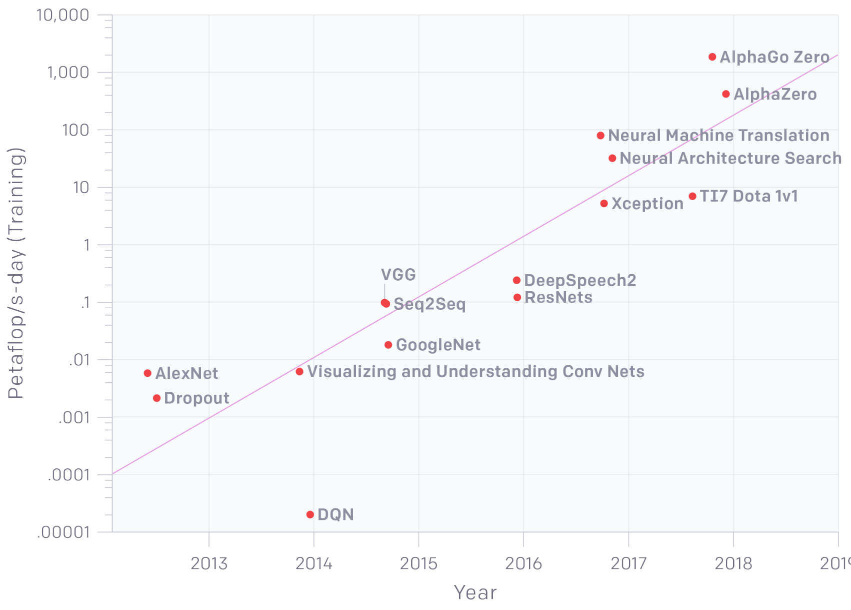


Figure 1.1: The amount of compute used to train deep learning models has increased 300,000x in 6 years. Figure taken from Amodei and Hernandez (2018).

opportunities for efficient improvements; for example, plotting performance as a function of training set size enables future work to compare performance even with small training budgets. Reporting the computational price tag of developing, training, and running models is a key Green AI practice. In addition to providing transparency, price tags are baselines that other researchers could improve on.

Our empirical analysis in Schwartz et al. (2019b) suggests that the AI research community has paid relatively little attention to computational efficiency. The computational cost of high-budget research continues to increase at a pace that exceeds Moore's Law (Moore, 1965). This holds despite the well-known diminishing returns from increasing the model size (Tan and Le, 2019) or the amount of data (Mahajan et al., 2018; Sun et al., 2017).

1.1 The Green AI Equation

To better understand the different ways in which AI research can be Red AI, consider an AI result reported in a scientific paper. This result typically characterizes a model trained on a training dataset and evaluated on a test dataset, and the process of developing that model often involves multiple experiments to tune its hyperparameters. We thus consider three dimensions which capture much of the computational cost of obtaining such a result: the cost of executing the model on a single (E)xample (either during training or at inference time); the size of the training (D)ataset, which controls the number of times the model is executed during training, and the number of (H)yperparameter experiments, which controls how many times the model is trained during model development. The total cost of producing a (R)esult in machine learning increases linearly with each of these quantities. This cost can be estimated as follows:

$$Cost(R) \propto E \cdot D \cdot H \tag{1.1}$$

We call this the Green AI equation, where the cost of an AI (R)esult grows linearly with the cost of processing a single (E)xample, the size of the training (D)ataset and the number of (H)yperparameter experiments. This thesis is structured around the factors therein; each chapter will introduce work which shows performance–efficiency tradeoffs for at least one of these terms. First, we will consider each factor in turn, outlining its trend in the broader community.

Expensive processing of one example, E: Our focus is on neural models, where it is common for each training step to require inference, so we discuss training and inference cost together as "processing" an example. Some works have used increasingly large models in terms of, e.g., model parameters, and as a result, in these models, performing inference can require a lot of computation, and training even more so. For instance, Google's BERT-large (Devlin et al., 2019) contains roughly 350 million parameters. OpenAI's openGPT2-XL model (Radford et al., 2019) contains 1.5 billion parameters. AI2 released Grover (Zellers et al., 2019), also containing 1.5 billion parameters. NVIDIA recently released Megatron-LM (Shoeybi et al., 2019), containing over 8 billion parameters. Google's T5-11B (Raffel et al., 2019) contains 11 billion parameters. Microsoft's Turing-NLG (Rosset, 2019) has 17 billion parameters. In the computer vision community, a similar trend is observed (Tan and Le, 2019).

Such large models have high costs for processing each example, which leads to large training costs. BERT-large was trained on 64 TPU chips for 4 days at an estimated cost of $7,000. Grover was trained on 256 TPU chips for two weeks, at an estimated cost of $25,000. XLNet had a similar architecture to BERT-large, but used a more expensive objective function (in addition to an order of magnitude more data), and was trained on 512 TPU chips for 2.5 days, costing more than $60,000.[2] It is impossible to reproduce the best BERT-large results or XLNet results using a single GPU,[3] and models such as openGPT2 are too large to be used in production.[4] Specialized models can have even more extreme costs, such as AlphaGo, the best version of which required 1,920 CPUs and 280 GPUs to play a single game of Go (Silver et al., 2016), with an estimated cost to reproduce this experiment of $35,000,000.[5,6]

When examining variants of a single model (e.g., BERT-small and BERT-large) larger models can have stronger performance, which is a valuable scientific contribution. However, this implies the financial and environmental cost of increasingly large AI models will not decrease soon, as

[2] `https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/`

[3] See `https://github.com/google-research/bert` and `https://github.com/zihangdai/xlnet`.

[4] `https://towardsdatascience.com/too-big-to-deploy-how-gpt-2-is-breaking-production-63ab29f0897c`

[5] `https://www.yuzeh.com/data/agz-cost.html`

[6] Recent versions of AlphaGo are far more efficient (Silver et al., 2017b).

the pace of model growth far exceeds the resulting increase in model performance (Howard et al., 2017). As a result, more and more resources are going to be required to keep improving AI models by simply making them larger.

In some cases the price of processing one example might be different at training and test time. For instance, some methods target efficient inference by learning a smaller model based on the large trained model. These models often do not lead to more efficient training, as the cost of E is only reduced at inference time. Models used in production typically have computational costs dominated by inference rather than training, but in research training is typically much more frequent, so studying methods for efficient processing of one example in both training and inference is important.

Processing many examples, D: Increased amounts of training data have also contributed to progress in state-of-the-art performance in AI. BERT-large (Devlin et al., 2019) had top performance in 2018 across many NLP tasks after training on 3 billion word-pieces. XLNet (Yang et al., 2019) recently outperformed BERT after training on 32 billion word-pieces, including part of Common Crawl; openGPT-2-XL (Radford et al., 2019) trained on 40 billion words; FAIR's RoBERTa (Liu et al., 2019b) was trained on 160GB of text, roughly 40 billion word-pieces, requiring around 25,000 GPU hours to train. T5-11B (Raffel et al., 2019) was trained on 1 trillion tokens, 300 times more than BERT-large. In computer vision, researchers from Facebook (Mahajan et al., 2018) pretrained an image classification model on 3.5 billion images from Instagram, three orders of magnitude larger than existing labelled image datasets such as Open Images.[7]

The use of massive data creates barriers for many researchers to reproducing the results of these models, and to training their own models on the same setup (especially as training for multiple epochs is standard). For example, the July 2019 Common Crawl contains 242 TB of uncompressed data,[8] so even storing the data is expensive. Finally, as in the case of model size, relying on more data to improve performance is notoriously expensive because of the diminishing returns of adding more data (Sun et al., 2017). For instance, Mahajan et al. (2018) show a logarithmic relation between the object recognition top-1 accuracy and the number of training examples.

Massive number of experiments, H: Some projects have poured large amounts of computation into tuning hyperparameters or searching over neural architectures, well beyond the reach of most researchers. For instance, researchers from Google (Zoph and Le, 2017) trained over 12,800 neural networks in their neural architecture search to improve performance on object detection and language modeling. With a fixed architecture, researchers from DeepMind (Melis et al., 2018) evaluated 1,500 hyperparameter assignments to demonstrate that an LSTM

[7]`https://opensource.google.com/projects/open-images-dataset`
[8]`http://commoncrawl.org/2019/07/`

language model (Hochreiter and Schmidhuber, 1997) can reach state-of-the-art perplexity results. Despite the value of this result in showing that the performance of an LSTM does not plateau after only a few hyperparameter trials, fully exploring the potential of other competitive models for a fair comparison is prohibitively expensive.

The value of massively increasing the number of experiments is not as well studied as the first two discussed above. In fact, outside of work developing new hyperparameter optimization methods, the number of experiments performed during model construction is often underreported. In Chapters 2 and 3 we provide evidence for a logarithmic relation exists here as well.

1.2 Structure of this thesis

The main contents of this thesis are organized around the Green AI equation. The chapters first address H, the set of experiments that go into getting a result, then E, the cost to process a single example during training or inference, and finally D, the amount of data. Each chapter presents performance–efficiency tradeoffs. Much of the experimental work in this thesis is on language data, though the motivations and conclusions from the results are more broadly applicable across AI and machine learning.

- Chapter 2 analyzes H, the number of experiments run for hyperparameter optimization. Here we focus on improved reporting of the results from the optimization procedure, which are rarely reported. We demonstrate that test-set performance scores alone are insufficient for drawing accurate conclusions about which model performs best. We argue for reporting additional details, especially performance on validation data obtained during model development. We present a novel technique for doing so: *expected validation performance* of the best-found model as a function of computation budget (i.e., the number of hyperparameter search trials or the overall training time). Using our approach, we find multiple recent model comparisons where authors would have reached a different conclusion if they had used more (or less) computation. Our approach also allows us to estimate the amount of computation required to obtain a given accuracy; applying it to several recently published results yields massive variation across papers, from hours to weeks. We conclude with a set of best practices for reporting experimental results which allow for robust future comparisons.

- Chapter 3 continues to analyze H, developing a technique for more efficient hyperparameter optimization. Specifically, here we study *open loop* hyperparameter optimization search methods: sequences that are predetermined and can be generated before a single configuration is evaluated. Examples include grid search, uniform random search, low discrepancy sequences, and other sampling distributions. In particular, we propose the use of k-determinantal point processes in hyperparameter optimization via random search.

Compared to conventional uniform random search where hyperparameter settings are sampled independently, a k-DPP promotes diversity. We describe an approach that transforms hyperparameter search spaces for efficient use with a k-DPP. In addition, we introduce a novel Metropolis-Hastings algorithm which can sample from k-DPPs defined over any space from which uniform samples can be drawn, including spaces with a mixture of discrete and continuous dimensions or tree structure. Our experiments show significant benefits in realistic scenarios with a limited budget for training supervised learners, whether in serial or parallel.

- Chapter 4 examines E, the computational requirements to process a single example, by building parameter-efficient neural models. Neural models for NLP typically use large numbers of parameters to reach state-of-the-art performance, which can lead to excessive memory usage and increased runtime. We present a structure learning method for learning sparse, parameter-efficient NLP models. Our method applies group lasso to rational RNNs (Peng et al., 2018), a family of models that is closely connected to weighted finite-state automata (WFSAs). We take advantage of rational RNNs' natural grouping of the weights, so the group lasso penalty directly removes WFSA states, substantially reducing the number of parameters in the model. Our experiments on a number of sentiment analysis datasets, using both GloVe and BERT embeddings, show that our approach learns neural structures which have fewer parameters without sacrificing performance relative to parameter-rich baselines. Our method also highlights the interpretable properties of rational RNNs. We show that sparsifying such models makes them easier to visualize, and we present models that rely exclusively on as few as three WFSAs after pruning more than 90% of the weights.

- Chapter 5 discusses D, the amount of data processed during a single experiment, and includes an example use of an early stopping algorithm. We study fine-tuning pretrained contextual word embedding models to supervised downstream tasks, which has become commonplace in natural language processing. This process, however, is often brittle: even with the same hyperparameter values, distinct random seeds can lead to substantially different results. To better understand this phenomenon, we experiment with four datasets from the GLUE benchmark, fine-tuning BERT hundreds of times on each while varying only the random seeds. We find substantial performance increases compared to previously reported results, and we quantify how the performance of the best-found model varies as a function of the number of fine-tuning trials. Further, we examine two factors influenced by the choice of random seed: weight initialization and training data order. We find that both contribute comparably to the variance of out-of-sample performance, and that some weight initializations perform well across all tasks explored. On small datasets, we observe that many fine-tuning trials diverge part of the way through training, and we offer best practices for practitioners to stop training less promising runs early.

1.3 Thesis statement

In this thesis I advocate for a broad adoption of efficiency as a primary evaluation criterion. Expensive experiments are driving the direction of AI, and such work is increasingly coming only from those research labs which have the largest budgets. The budgets used are often not described in detail; I argue that **reproducible conclusions as to which approach performs best cannot be made without accounting for the computational budget**. The first step to address these issues is to improve *reporting* of efficiency. Once we have appropriate measurement tools and measurements themselves, we can then *optimize* performance–efficiency tradeoffs.

Through the lens of the Green AI equation, I've outlined three areas for improvements. Together, the cost of processing of a single (E)xample, the amount of (D)ata, and the number of (H)yperparameter tuning experiments capture most of the increases in computational expense. This thesis illustrates how describing where research falls on each of these three dimensions facilitates more fair model comparisons and, thus, scientific advancement in machine learning. In addition, the work herein provides examples of how improved performance–efficiency tradeoffs can lead to performance improvements overall while also providing opportunities for lower-budget comparsions in future research, leading to a more equitable research community.

Chapter 2

Improved Reporting of Hyperparameter Optimization Results

In machine learning and natural language processing, improved performance on held-out test data is typically used as an indication of the superiority of one method over others. But, as the field grows, there is an increasing gap between the large computational budgets used for some high-profile experiments and the budgets used in most other work (Schwartz et al., 2019b). When the only reported performance is on test data, the computational budget and number of experiments is often not reported at all. This hinders meaningful comparison between experiments, as improvements in performance can, in some cases, be obtained purely through more intensive hyperparameter tuning (Melis et al., 2018; Lipton and Steinhardt, 2018). The work in this chapter extends Dodge et al. (2019a), and we publicly release code.[1]

Recent investigations into “state-of-the-art” claims have found competing methods to only be comparable, without clear superiority, even against baselines (Reimers and Gurevych, 2017; Lucic et al., 2018; Li and Talwalkar, 2019); this has exposed the need for reporting more than a single point estimate of performance. In this chapter, we demonstrate that test-set performance scores alone are insufficient for drawing accurate conclusions about which model performs best. We argue for reporting additional details, especially performance on validation data obtained during model development.

Echoing calls for more rigorous scientific practice in machine learning (Lipton and Steinhardt, 2018; Sculley et al., 2018), we draw attention to the weaknesses in current reporting practices and propose solutions which would allow for fairer comparisons and improved reproducibility.

Our primary technical contribution in this chapter is the introduction of a tool for reporting

[1] `https://github.com/allenai/show-your-work`

validation results in an easily interpretable way: *expected validation performance* of the best model under a given computational budget (we use the term *performance* as a general evaluation measure, e.g., accuracy, F_1, etc.). That is, given a budget sufficient for training and evaluating n models, we calculate the expected performance of the best of these models on validation data. Note that this differs from the *best observed* value after n evaluations. Because the expectation can be estimated from the distribution of N validation performance values, with $N \geq n$, and these are obtained during model development,[2] our method **does not require additional computation** beyond hyperparameter search or optimization. We encourage researchers to report expected validation performance as a curve, across values of $n \in \{1, \ldots, N\}$.

As we show in §2.3.3, our approach makes clear that the expected-best performing model is a function of the computational budget. In §2.3.4 we show how our approach can be used to estimate the budget that went into obtaining previous results; in one example, we see a too-small budget for baselines, while in another we estimate a budget of about 18 GPU days was used (but not reported). Previous work on reporting validation performance used the bootstrap to approximate the mean and variance of the best performing model (Lucic et al., 2018); in §2.2.2 we show that our approach computes these values with strictly less error than the bootstrap. Using our approach, we find multiple recent model comparisons where authors would have reached a different conclusion if they had used more (or less) computation.

We conclude by presenting a set of recommendations for researchers that will improve scientific reporting over current practice. We emphasize this work is about *reporting*, not about running additional experiments (which undoubtedly can improve evidence in comparisons among models). Our reporting recommendations aim at reproducibility and improved understanding of sensitivity to hyperparameters and random initializations. Some of our recommendations may seem obvious; however, our empirical analysis shows that out of fifty EMNLP 2018 papers chosen at random, none report all items we suggest.

2.1 Background

Reproducibility Reproducibility in machine learning is often defined as the ability to produce the *exact* same results as reported by the developers of the model. In this work, we follow Gundersen and Kjensmo (2018) and use an extended notion of this concept: when comparing two methods, two research groups with different implementations should follow an experimental procedure which leads to the same conclusion about which performs better. As illustrated in Fig. 2.1, this conclusion often depends on the amount of computation applied. Thus, to make a *reproducible* claim about which model performs best, we must also take into account the budget used (e.g., the number of hyperparameter trials).

[2]We leave forecasting performance with larger budgets $n > N$ to future work.

Notation We use the term *model family* to refer to an approach subject to comparison and to hyperparameter selection (examples of which include different architectures, but also ablations of the same architecture. Each model family $\mathcal{M}$ requires its own hyperparameter selection, in terms of a set of k hypermarameters, each of which defines a range of possible values. A *hyperparameter value* (denoted h) is a k-tuple of specific values for each hyperparameter. We call the set of all possible hyperparameter values $\mathcal{H}_{\mathcal{M}}$.The hyperparameter value space can also include the random seed used to initialize the model, and some specifications such as the size of the hidden layers in a neural network, in addition to commonly tuned values such as learning rate. Given $\mathcal{H}_{\mathcal{M}}$ and a computational budget sufficient for training B models, the set of hyperparameter values is $\{h_1, \ldots, h_B\}$, $h_i \in \mathcal{H}_{\mathcal{M}}$. We let $m_i \in \mathcal{M}$ denote the model trained with hyperparameter value h_i.

Hyperparameter value selection There are many ways of selecting hyperparameter values, h_i. Grid search and uniform sampling are popular systematic methods; the latter has been shown to be superior for most search spaces (Bergstra and Bengio, 2012). Adaptive search strategies such as Bayesian optimization select h_i after evaluating $h_1, \ldots, h_{i-1}$. While these strategies may find better results quickly, they are generally less reproducible and harder to parallelize (Li et al., 2018). Manual search, where practitioners use knowledge derived from previous experience to adjust hyperparameters after each experiment, is a type of adaptive search that is the least reproducible, as different practitioners make different decisions. We have further discussion of different hyperparameter search strategies in Chapter 3, where we introduce an approach that is fully parallel but has better coverage than uniform sampling. Regardless of the strategy adopted, we advocate for detailed reporting of the method used for hyperparmeter value selection and the budget (§2.4). We next introduce a technique to visualize results of samples which are drawn i.i.d. (e.g., random initializations or uniformly sampled hyperparameter values).

2.2 Expected Validation Performance Given Budget

After selecting the best hyperparameter values h_{i^*} from among $\{h_1, \ldots, h_B\}$ with actual budget B, NLP researchers typically evaluate the associated model m_{i^*} on the test set and report its performance as an estimate of the family $\mathcal{M}$'s ability to generalize to new data. We propose to make better use of the intermediately-trained models $m_1, \ldots, m_B$.

For any set of n hyperparmeter values, denote the validation performance of the best model as

$$v_n^* = \max\nolimits_{h \in \{h_1, \ldots, h_n\}} \mathcal{A}(\mathcal{M}, h, \mathcal{D}_T, \mathcal{D}_V), \tag{2.1}$$

where $\mathcal{A}$ denotes an algorithm that returns the performance on validation data $\mathcal{D}_V$ after training a model from family $\mathcal{M}$ with hyperparameter values h on training data $\mathcal{D}_T$.[3] We view evaluations of $\mathcal{A}$ as the elementary unit of experimental cost.[4]

Though not often done in practice, procedure equation 2.1 could be repeated many times with different hyperparameter values, yielding a *distribution* of values for random variable V_n^*. This would allow us to estimate the *expected* performance, $\mathbb{E}[V_n^* \mid n]$ (given n hyperparameter configurations). The key insight used below is that, if we use random search for hyperparameter selection, then the effort that goes into a single round of random search (Eq. 2.1) suffices to construct a useful estimate of expected validation performance, without requiring *any further experimentation.*

Under random search, the n hyperparameter values $h_1, \ldots, h_n$ are drawn uniformly at random from $\mathcal{H}_\mathcal{M}$, so the values of $\mathcal{A}(\mathcal{M}, h_i, \mathcal{D}_T, \mathcal{D}_V)$ are i.i.d. As a result, the maximum among these is itself a random variable. We introduce a diagnostic that captures information about the computation used to generate a result: the expectation of maximum performance, *conditioned* on n, the amount of computation used in the maximization over hyperparameters and random initializations:

$$\mathbb{E}\left[\max_{h \in \{h_1, \ldots, h_n\}} \mathcal{A}(\mathcal{M}, h, \mathcal{D}_T, \mathcal{D}_V) \mid n\right]. \tag{2.2}$$

Reporting this expectation as we vary $n \in \{1, 2, \ldots, B\}$ gives more information than the maximum v_B^* (Eq. 2.1 with $n = B$); future researchers who use this model will know more about the computation budget required to achieve a given performance. We turn to calculating this expectation, then we compare it to the bootstrap (§2.2.2), and discuss estimating variance (§2.2.3).

2.2.1 Expected Maximum

We describe how to estimate the expected maximum validation performance (Eq. 2.2) given a budget of n hyperparameter values. Conversion to alternate formulations of budget, such as GPU hours or cloud-machine rental cost in dollars, is straightforward in most cases.

Assume we draw $\{h_1, \ldots, h_n\}$ uniformly at random from hyperparameter space $\mathcal{H}_\mathcal{M}$. Each evaluation of $\mathcal{A}(\mathcal{M}, h, \mathcal{D}_T, \mathcal{D}_V)$ is therefore an i.i.d. draw of a random variable, denoted V_i, with observed value v_i for $h_i \sim \mathcal{H}_\mathcal{M}$. Let the maximum among n i.i.d. draws from an unknown distribution be

$$V_n^* = \max_{i \in \{1, \ldots, n\}} V_i \tag{2.3}$$

[3]$\mathcal{A}$ captures standard parameter estimation, as well as procedures that depend on validation data, like early stopping.

[4]Note that researchers do not always report validation, but rather *test* performance, a point we will return to in §2.4.

We seek the expected value of V_n^* given n:

$$\mathbb{E}[V_n^* \mid n] = \textstyle\sum_v v \cdot P(V_n^* = v \mid n) \tag{2.4}$$

where $P(V_n^* \mid n)$ is the probability mass function (PMF) for the max-random variable.[5]

For discrete random variables,

$$P(V_n^* = v \mid n) = P(V_n^* \leq v \mid n) - P(V_n^* < v \mid n), \tag{2.5}$$

Using the definition of "max", and the fact that the V_i are drawn i.i.d.,

$$\begin{aligned} P(V_n^* \leq v \mid n) &= P\left(\max_{i \in \{1,\ldots,n\}} V_i \leq v \mid n\right) \\ &= P(V_1 \leq v, V_2 \leq v, \ldots, V_n \leq v \mid n) \\ &= \textstyle\prod_{i=1}^n P(V_i \leq v) = P(V \leq v)^n, \end{aligned} \tag{2.6}$$

and similarly for $P(V_n^* < v \mid n)$.

$P(V \leq v)$ and $P(V < v)$ are cumulative distribution functions, which we can estimate using the empirical distribution, i.e.

$$\hat{P}(V \leq v) = \tfrac{1}{n} \textstyle\sum_{i=1}^n \mathbb{1}_{[V_i \leq v]} \tag{2.7}$$

and similarly for strict inequality.

Thus, our estimate of the expected maximum validation performance is

$$\hat{\mathbb{E}}[V_n^* \mid n] = \textstyle\sum_v v \cdot (\hat{P}(V_i \leq v)^n - \hat{P}(V_i < v)^n). \tag{2.8}$$

Discussion As we increase the amount of computation for evaluating hyperparameter values (n), the maximum among the samples will approach the observed maximum v_B^*. Hence the curve of $\mathbb{E}[V_n^* \mid n]$ as a function of n will appear to asymptote. Our focus here is not on estimating that value, and we do not make any claims about extrapolation of V^* beyond B, the number of hyperparameter values to which $\mathcal{A}$ is actually applied.

Two points follow immediately from our derivation. First, at $n = 1$, $\mathbb{E}[V_1^* \mid n = 1]$ is the mean of $v_1, \ldots, v_n$. Second, for all n, $\mathbb{E}[V_n^* \mid n] \leq v_n^* = \max_i v_i$, which means the curve is a lower bound on the selected model's validation performance.

2.2.2 Comparison with Bootstrap

Lucic et al. (2018) and Henderson et al. (2018) have advocated for using the bootstrap to

[5]For a finite validation set $\mathcal{D}_V$, most performance measures (e.g., accuracy) only take on a finite number of possible values, hence the use of a sum instead of an integral in Eq. 2.4.

estimate the mean and variance of the best validation performance. The bootstrap (Efron and Tibshirani, 1994) is a general method which can be used to estimate statistics that do not have a closed form. The bootstrap process is as follows: draw N i.i.d. samples (in our case, N model evaluations). From these N points, sample n points (with replacement), and compute the statistic of interest (e.g., the max). Do this K times (where K is large), and average the computed statistic. By the law of large numbers, as $K \to \infty$ this average converges to the sample expected value (Efron and Tibshirani, 1994).

The bootstrap has two sources of error: the error from the finite sample of N points, and the error introduced by resampling these points K times. Our approach has strictly less error than using the bootstrap: our calculation of the expected maximum performance in §2.2.1 provides a closed-form solution, and thus contains none of the resampling error (the finite sample error is the same).

2.2.3 Variance of V_n^*

Expected performance becomes more useful with an estimate of variation. When using the bootstrap, standard practice is to report the standard deviation of the estimates from the K resamples. As $K \to \infty$, this standard deviation approximates the sample standard error (Efron and Tibshirani, 1994). We instead calculate this from the distribution in Eq. 2.5 using the standard plug-in-estimator.

In most cases, we advocate for reporting a measure of variability such as the standard deviation or variance; however, in some cases it might cause confusion. For example, when the variance is large, plotting the expected value plus the variance can go outside of reasonable bounds, such as accuracy greater than any observed (even greater than 1). In such situations, we recommend shading only values within the observed range, such as in Fig. 2.4. Additionally, in situations where the variance is high and variance bands overlap between model families (e.g., Fig. 2.1), the mean is still the most informative statistic.

2.3 Case Studies

Here we show two clear use cases of our method. First, we can directly estimate, for a given budget, which approach has better performance. Second, we can estimate, given our experimental setup, the budget for which the reported validation performance (V^*) matches a desired performance level. We present three examples that demonstrate these use cases. First, we reproduce previous findings that compared different models for text classification. Second, we explore the time vs. performance tradeoff of models that use contextual word embeddings (Peters et al., 2018). Third, from two previously published papers, we examine the budget required for our expected performance to match their reported performance. We find these budget estimates vary drastically. Consistently, we see that the best model is a function of the budget. We publicly release the search space and training configurations used for each case study.[6]

Note that we do not report test performance in our experiments, as our purpose is not to establish a benchmark level for a model, but to demonstrate the utility of expected validation performance for model comparison and reproducibility.

2.3.1 Experimental Details

For each experiment, we document the hyperparameter search space, hardware, average runtime, number of samples, and links to model implementations. We use public implementations for all models in our experiments, primarily in AllenNLP (Gardner et al., 2018). We use Tune (Liaw et al., 2018) to run parallel evaluations of uniformly sampled hyperparameter values.

2.3.2 Validating Previous Findings

We start by applying our technique on a text classification task in order to confirm a well-established observation (Yogatama and Smith, 2015): logistic regression has reasonable performance with minimal hyperparameter tuning, but a well-tuned convolutional neural network (CNN) can perform better.

We experiment with the fine-grained Stanford Sentiment Treebank text classification dataset (Socher et al., 2013). For the CNN classifier, we embed the text with 50-dim GloVe vectors (Pennington et al., 2014), feed the vectors to a ConvNet encoder, and feed the output representation into a softmax classification layer. We use the *scikit-learn* implementation of logistic regression[7] with bag-of-word counts and a linear classification layer. The hyperparameter spaces $\mathcal{H}_{\text{CNN}}$ and $\mathcal{H}_{\text{LR}}$ are detailed in Appendix A.1. For logistic regression we used bounds suggested by Yogatama and Smith (2015), which include term weighting, n-grams, stopwords, and learning rate. For the CNN we follow the hyperparameter sensitivity analysis in Zhang and Wallace (2015).

We run 50 trials of random hyperparameter search for each classifier. Our results (Fig. 2.1) confirm previous findings (Zhang and Wallace, 2015): under a budget of fewer than 10 hyperparameter search trials, logistic regression achieves a higher expected validation accuracy than the CNN. As the budget increases, the CNN gradually improves to a higher overall expected validation accuracy. For all budgets, logistic regression has lower variance, so may be a more suitable approach for fast prototyping.

2.3.3 Contextual Representations

We next explore how computational budget affects the performance of contextual embedding models (Peters et al., 2018). Recently, Peters et al. (2019) compared two methods for using contextual representations for downstream tasks: *feature extraction*, where features are fixed after pretraining and passed into a task-specific model, or *fine-tuning*, where they are updated during task training. Peters et al. (2019) found that feature extraction is preferable to fine-tuning ELMo embeddings. Here we set to explore whether this conclusion depends on the experimental budget.

Closely following their experimental setup, in Fig. 2.2 we show the expected performance of the biattentive classification network (BCN; McCann et al., 2017) with three embedding approaches (GloVe only, GloVe + ELMo frozen, and GloVe + ELMo fine-tuned), on the binary Stanford Sentiment Treebank task. Peters et al. (2019) use a BCN with frozen embeddings and a BiLSTM BCN for fine-tuning. We conducted experiments with both a BCN and a BiLSTM with frozen and fine-tuned embeddings, and found our conclusions to be consistent. We report the full hyperparameter search space, which matched Peters et al. (2019) as closely as their reporting allowed, in Appendix A.2.

We use *time* for the budget by scaling the curves by the average observed training duration for each model. We observe that as the time budget increases, the expected best-performing model changes. In particular, we find that our experimental setup leads to the same conclusion as Peters et al. (2019) given a budget between approximately 6 hours and 1 day. For larger budgets (e.g., 10 days) fine-tuning outperforms feature extraction. Moreover, for smaller budgets ($<$ 2 hours), using GloVe embeddings is preferable to ELMo (frozen or fine-tuned).

2.3.4 Inferring Budgets in Previous Reports

Our method provides another appealing property: estimating the budget required for the expected performance to reach a particular level, which we can compare against previously reported results. We present two case studies, and show that the amount of computation required to match the reported results varies drastically.

We note that in the two examples that follow, the original papers only reported partial experimental information; we made sure to tune the hyperparameters they did list in addition to standard choices (such as the learning rate). In neither case do they report the method used to tune the hyperparameters, and we suspect they tuned them manually. Our experiments here are meant give an idea of the budget that would be required to reproduce their results or to apply their models to other datasets under random hyperparameter value selection.

[6] `https://github.com/allenai/show-your-work`
[7] `https://scikit-learn.org`

SciTail When introducing the SciTail textual entailment dataset, Khot et al. (2018) compared four models: an *n-gram* baseline, which measures word-overlap as an indicator of entailment, *ESIM* (Chen et al., 2017), a sequence-based entailment model, *DAM* (Parikh et al., 2016), a bag-of-words entailment model, and their proposed model, *DGEM* (Khot et al., 2018), a graph-based structured entailment model. Their conclusion was that DGEM outperforms the other models.

We use the same implementations of each of these models each with a hyperparameter search space detailed in Appendix A.3. The search space bounds we use are large neighborhoods around the hyperparameter assignments specified in the public implementations of these models. Note that these curves depend on the specific hyperparameter search space adopted; as the original paper does not report hyperparameter search or model selection details, we have chosen what we believe to be reasonable bounds, and acknowledge that different choices could result in better or worse expected performance. We use a budget based on trials instead of runtime so as to emphasize how these models behave when given a comparable number of hyperparameter configurations.

Our results (Fig. 2.3) show that the different models require different budgets to reach their reported performance in expectation, ranging from 2 (n-gram) to 20 (DGEM). Moreover, providing a large budget for each approach improves performance substantially over reported numbers. Finally, under different computation budgets, the top performing model changes (though the neural models are similar).

SQuAD Next, we turn our attention to SQuAD (Rajpurkar et al., 2016) and report performance of the commonly-used BiDAF model (Seo et al., 2017). The set of hyperparameters we tune covers those mentioned in addition to standard choices (details in Appendix A.3). We see in Fig. 2.4 that we require a budget of 18 GPU days in order for the expected maximum validation performance to match the value reported in the original paper. This suggests that some combination of prior intuition and extensive hyperparameter tuning were used by the original authors, though neither were reported.

2.4 Recommendations

Experimental results checklist The findings discussed in this chapter and other similar efforts highlight methodological problems in experimental machine learning and NLP. In this section we provide a checklist to encourage researchers to report more comprehensive experimentation results. Our list, shown in Text Box 1, builds on the reproducibility checklist that was introduced for the machine learning community during NeurIPS 2018 (which was required to be filled out for each NeurIPS 2019 and ICML 2020 submission; Pineau, 2019).

Our focus is on improved reporting of experimental results, thus we include relevant points from their list in addition to our own. Similar to other calls for improved reporting in machine learning (Mitchell et al., 2019; Gebru et al., 2018), we recommend pairing experimental results with the information from this checklist in a structured format (see examples provided in Appendix A.1).

Text Box 1 Experimental results checklist.

✓ **For all reported experimental results**

- ☐ Description of computing infrastructure
- ☐ Average runtime for each approach
- ☐ Details of train/validation/test splits
- ☐ Corresponding validation performance for each reported test result
- ☐ A link to implemented code

✓ **For experiments with hyperparameter search**

- ☐ Bounds for each hyperparameter
- ☐ Hyperparameter configurations for best-performing models
- ☐ Number of hyperparameter search trials
- ☐ The method of choosing hyperparameter values (e.g., uniform sampling, manual tuning, etc.) and the criterion used to select among them (e.g., accuracy)
- ☐ Expected validation performance, as introduced in §2.2.1, or another measure of the mean and variance as a function of the number of hyperparameter trials.

EMNLP 2018 checklist coverage. To estimate how commonly this information is reported in the NLP community, we sample fifty random EMNLP 2018 papers that include experimental results and evaluate how well they conform to our proposed reporting guidelines. We find that none of the papers reported all of the items in our checklist. However, every paper reported at least one item in the checklist, and each item is reported by at least one paper. Of the papers we analyzed, 74% reported at least some of the best hyperparameter assignments. By contrast, 10% or fewer papers reported hyperparameter search bounds, the number of hyperparameter evaluation trials, or measures of central tendency and variation. We include the full results of this analysis in Table 2.1.

Comparisons with different budgets. We have argued that claims about relative model performance should be qualified by computational expense. With varying amounts of computation, not all claims about superiority are valid. If two models have similar budgets, we can claim one outperforms the other (with that budget). Similarly, if a model with a small budget outperforms a model with a large budget, increasing the small budget will not change this conclusion. However, if a model with a large budget outperforms a model with a small budget, the difference might be due to the model or the budget (or both). As a concrete example, Melis et al. (2018) report the performance of an LSTM on language modeling the Penn Treebank after 1,500 rounds of Bayesian optimization; if we compare to a new $\mathcal{M}$ with a smaller budget, we

Checklist item	**Percentage of EMNLP 2018 papers**
Reports train/validation/test splits	92%
Reports best hyperparameter assignments	74%
Reports code	30%
Reports dev accuracy	24%
Reports computing infrastructure	18%
Reports empirical runtime	14%
Reports search strategy	14%
Reports score distribution	10%
Reports number of hyperparameter trials	10%
Reports hyperparameter search bounds	8%

Table 2.1: Presence of checklist items from §5 across 50 randomly sampled EMNLP 2018 papers that involved modeling experiments.

can only draw a conclusion if the new model outperforms the LSTM. [8]

In a larger sense, there may be no simple way to make a comparison "fair." For example, the two models in Fig. 2.1 have hyperparameter spaces that are different, so fixing the same number of hyperparameter trials for both models does not imply a fair comparison. In practice, it is often not possible to measure how much past human experience has contributed to reducing the hyperparameter bounds for popular models, and there might not be a way to account for the fact that better understood (or more common) models can have better spaces to optimize over. Further, the cost of one application of $\mathcal{A}$ might be quite different depending on the model family. Converting to runtime is one possible solution, but implementation effort could still affect comparisons at a fixed x-value. Because of these considerations, our focus is on reporting whatever experimental results exist.

2.5 Discussion: Reproducibility

In NLP, the use of standardized test sets and public leaderboards (which limit test evaluations) has helped to mitigate the so-called "replication crisis" happening in fields such as psychology and medicine (Ioannidis, 2005; Gelman and Loken, 2014). Unfortunately, leaderboards can create additional reproducibility issues (Rogers, 2019). First, leaderboards obscure the budget that was used to tune hyperparameters, and thus the amount of work required to apply a model to a new dataset. Second, comparing to a model on a leaderboard is difficult if they *only* report test scores. For example, on the GLUE benchmark (Wang et al., 2018), the differences in *test set* performance between the top performing models can be on the order of a tenth of a percent, while the difference between test and validation performance might be one percent or larger.

[8]This is similar to controlling for the amount of training data, which is an established norm in NLP research.

Verifying that a new implementation matches established performance requires submitting to the leaderboard, wasting test evaluations. Thus, we recommend leaderboards report validation performance for models evaluated on test sets.

As an example, consider Devlin et al. (2019), which introduced BERT and reported state-of-the-art results on the GLUE benchmark. The authors provide some details about the experimental setup, but do not report a specific budget. Subsequent work which extended BERT (Phang et al., 2018) included distributions of validation results, and we highlight this as a positive example of how to report experimental results. To achieve comparable test performance to Devlin et al. (2019), the authors report the best of twenty or one hundred random initializations. Their validation performance reporting not only illuminates the budget required to fine-tune BERT on such tasks, but also gives other practitioners results against which they can compare without submitting to the leaderboard.

2.6 Related Work

Lipton and Steinhardt (2018) address a number of problems with the practice of machine learning, including incorrectly attributing empirical gains to modeling choices when they came from other sources such as hyperparameter tuning. Sculley et al. (2018) list examples of similar evaluation issues, and suggest encouraging stronger standards for empirical evaluation. They recommend detailing experimental results found throughout the research process in a time-stamped document, as is done in other experimental science fields. Our work formalizes these issues and provides an actionable set of recommendations to address them.

Reproducibility issues relating to standard data splits (Schwartz et al., 2011; Gorman and Bedrick, 2019; Recht et al., 2019a,b) have surfaced in a number of areas. Shuffling standard training, validation, and test set splits led to a drop in performance, and in a number of cases the inability to reproduce rankings of models. Dror et al. (2017) studied reproducibility in the context of consistency among multiple comparisons.

Limited community standards exist for documenting datasets and models. To address this, Gebru et al. (2018) recommend pairing new datasets with a "datasheet" which includes information such as how the data was collected, how it was cleaned, and the motivation behind building the dataset. Similarly, Mitchell et al. (2019) advocate for including a "model card" with trained models which document training data, model assumptions, and intended use, among other things. Our recommendations in §2.4 are meant to document relevant information for experimental results.

In Chapter 5 we further examine the stability of contextual language models, and show that the results claimed in a number of other papers are brittle.

2.7 Conclusion

In this chapter we have shown how current practice in experimental NLP fails to support a simple standard of reproducibility, primarily due to failing to sufficiently report experimental information. We have shown that the computational budget, and how performance changes as a function of that budget, is necessary information when drawing reproducible conclusions about which approach performs best. Our technique for estimating the expected validation performance of a method as a function of the budget helps address this shortcoming. In addition, our recommendations for reporting experimental findings through a reproducibility checklist outline how the community can first measure then improve upon efficient methods in machine learning and NLP.

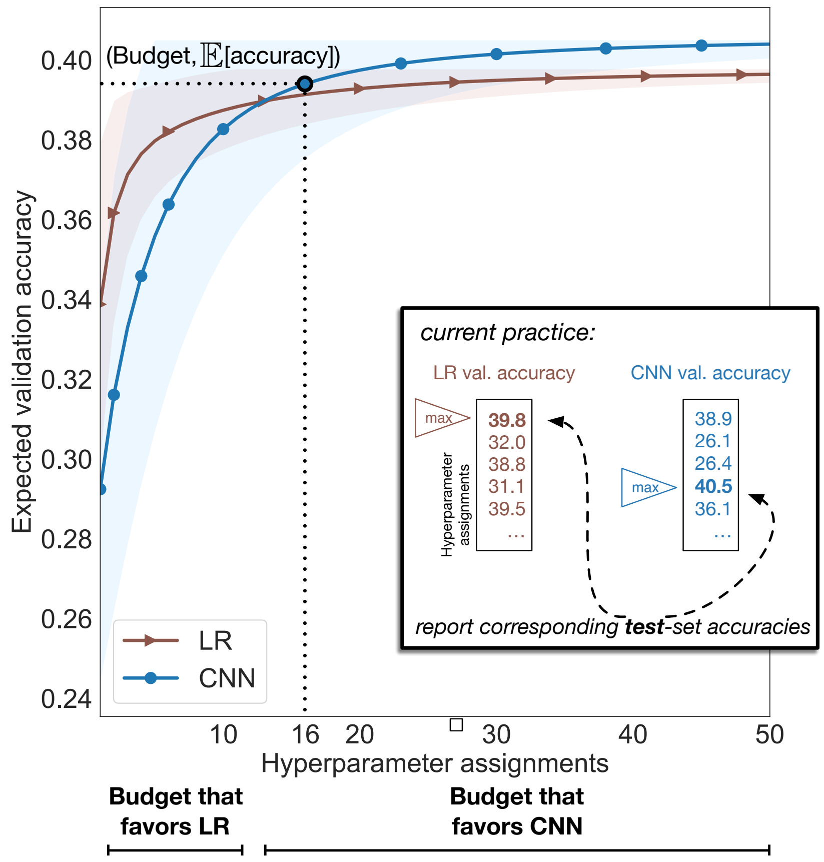


Figure 2.1: Current practice when comparing NLP models is to train multiple instantiations of each, choose the best model of each type based on validation performance, and compare their performance on test data (inner box). Under this setup, (assuming test-set results are similar to validation), one would conclude from the results above (hyperparameter search for two models on the 5-way SST classification task) that the CNN outperforms Logistic Regression (LR). In our proposed evaluation framework, we instead encourage practitioners to consider the expected validation accuracy (y-axis; shading shows ± 1 standard deviation), as a function of budget (x-axis). Each point on a curve is the *expected value* of the best validation accuracy obtained (y) after evaluating x random hyperparameter values. Note that (1) the better performing model depends on the computational budget; LR has higher expected performance for budgets up to 10 hyperparameter assignments, while the CNN is better for larger budgets. (2) Given a model and desired accuracy (e.g., 0.395 for CNN), we can estimate the expected budget required to reach it (16; dotted lines).

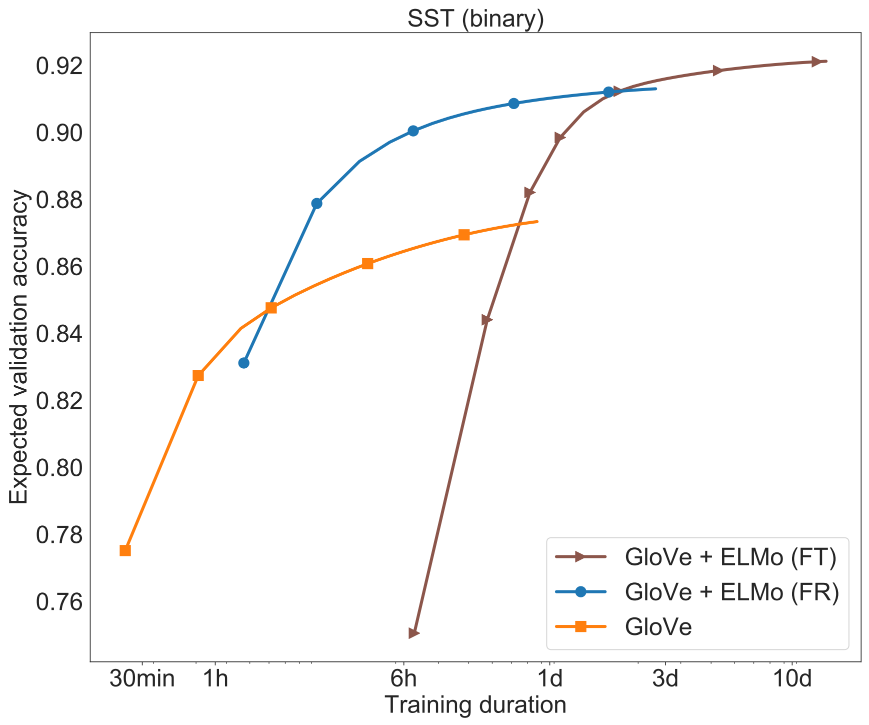


Figure 2.2: Expected maximum performance of a BCN classifier on SST. We compare three embedding approaches (GloVe embeddings, GloVe + frozen ELMo, and GloVe + fine-tuned ELMo). The x-axis is time, on a log scale. We omit the variance for visual clarity. For each of the three model families, we sampled 50 hyperparameter values, and plot the expected maximum performance with the x-axis values scaled by the average training duration. The plot shows that for each approach (GloVe, ELMo frozen, and ELMo fine-tuned), there exists a budget for which it is preferable.

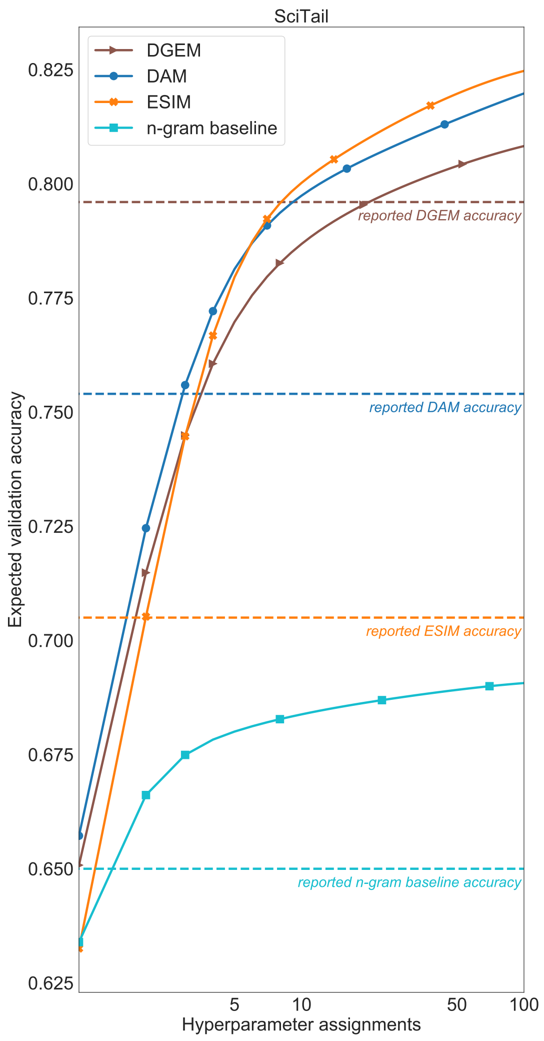


Figure 2.3: Comparing reported accuracies (dashed lines) on SciTail to expected validation performance under varying levels of compute (solid lines). The estimated budget required for expected performance to match the reported result differs substantially across models, and the relative ordering varies with budget. We omit variance for visual clarity.

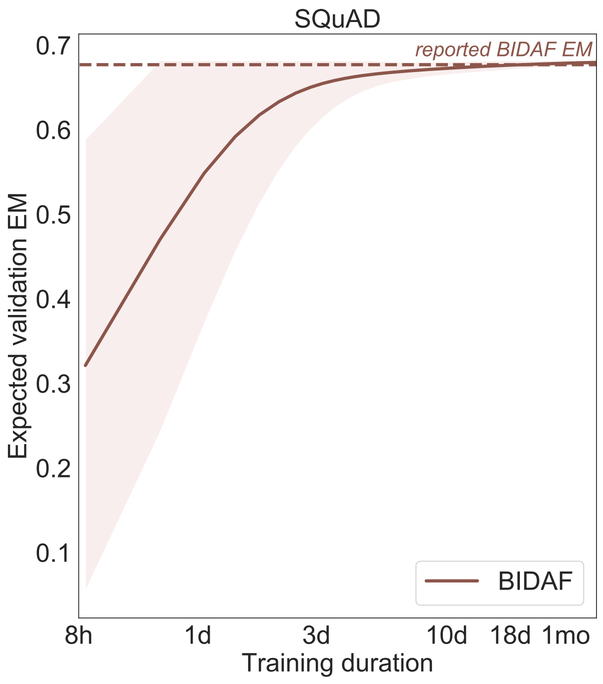


Figure 2.4: Comparing reported development exact-match score of BIDAF (dashed line) on SQuAD to expected performance of the best model with varying computational budgets (solid line). The shaded area represents the expected performance ± 1 standard deviation, within the observed range of values. for the expected performance to match the reported results.

Chapter 3

Efficient Hyperparameter Optimization

Driven by the need for parallelizable hyperparameter optimization methods, this chapter studies *open loop* search methods: sequences that are predetermined and can be generated before a single configuration is evaluated. Efficient hyperparameter search methods can reduce the computational cost of achieving a certain level of performance, or improve performance for a given budget. In Chapter 2 we developed a tool for reporting the results of uniform sampling for hyperparameter optimization, and here we introduce a new hyperparameter optimization algorithm which is more computationally efficient than uniform sampling. Our experiments show significant benefits in realistic scenarios with a limited budget for training supervised learners, whether in serial or parallel. This chapter extends Dodge et al. (2017).

Hyperparameter values—regularization strength, model family choices like depth of a neural network or which nonlinear functions to use, procedural elements like dropout rates, stochastic gradient descent step sizes, and data preprocessing choices—can make the difference between a successful application of machine learning and a wasted effort. To search among many hyperparameter values requires repeated execution of often-expensive learning algorithms, creating a major obstacle for practitioners and researchers alike.

In general, on iteration (evaluation) k, a hyperparameter searcher suggests a d-dimensional hyperparameter configuration $x_k \in \mathcal{X}$ (e.g., $\mathcal{X} = \mathbb{R}^d$ but could also include discrete dimensions), a worker trains a model using x_k, and returns a validation loss of $y_k \in \mathbb{R}$ computed on a hold out set. In this work we say a hyperparameter searcher is **open loop** if x_k depends only on $\{x_i\}_{i=1}^{k-1}$; examples include choosing x_k uniformly at random (Bergstra and Bengio, 2012), or x_k coming from a low-discrepancy sequence (c.f., (Iacò, 2015)). We say a searcher is **closed loop** if x_k depends on both the past configurations and validation losses $\{(x_i, y_i)\}_{i=1}^{k-1}$; examples include Bayesian optimization (Snoek et al., 2012) and reinforcement learning methods (Zoph and Le, 2017). Note that open loop methods can draw an infinite sequence of configurations before training a single model, whereas closed loop methods rely on validation loss feedback in order to make suggestions.

While sophisticated closed loop selection methods have been shown to empirically identify good hyperparameter configurations faster (i.e., with fewer iterations) than open loop methods like random search, **two trends have rekindled interest in embarrassingly parallel open loop methods**: 1) modern deep learning model are taking longer to train, sometimes up to days or weeks, and 2) the rise of cloud resources available to anyone that charge not by the number of machines, but by the number of CPU-hours used so that 10 machines for 100 hours costs the same as 1000 machines for 1 hour.

This chapter explores the landscape of open loop methods, identifying tradeoffs that are rarely considered, if at all acknowledged. While random search is arguably the most popular open loop method and chooses each x_k independently of $\{x_i\}_{i=1}^{k-1}$, it is by no means the only choice. In many ways uniform random search is the least interesting of the methods we will discuss because we will advocate for methods where x_k depends on $\{x_i\}_{i=1}^{k-1}$ to promote **diversity**. In particular, we will focus on drawing $\{x_i\}_{i=1}^{k}$ from a **k-determinantal point process (DPP)** (Kulesza et al., 2012). We introduce a sampling algorithm which allows DPPs to support real, integer, and categorical dimensions, any of which may have a tree structure, and we describe connections between DPPs and Gaussian processes (GPs).

In synthetic experiments, we find our diversity-promoting open-loop method outperforms other open loop methods. In practical hyperparameter optimization experiments, we find that it significantly outperforms other approaches in cases where the hyperparameter values have a large effect on performance. Finally, we compare against a closed loop Bayesian optimization method, and find that sequential Bayesian optimization takes, on average, more than ten times as long to find a good result, for a gain of only 0.15 percent accuracy on a particular hyperparameter optimization task.

3.1 Related Work

While this work focuses on open loop methods, the vast majority of recent work on hyperparameter tuning has been on closed loop methods, which we briefly review.

3.1.1 Closed Loop Methods

Much attention has been paid to sequential model-based optimization techniques such as Bayesian optimization (Bergstra et al., 2011; Snoek et al., 2012) which sample hyperparameter spaces adaptively. These techniques first choose a point in the space of hyperparameters, then train and evaluate a model with the hyperparameter values represented by that point, then sample another point based on how well previous point(s) performed. When evaluations are fast, inexpensive, and it's possible to evaluate a large number of points (e.g. $k = \Omega(2^d)$ for d hyperparameters) these approaches can be advantageous, but in the more common scenario where we have limited time or a limited evaluation budget, the sequential nature of closed loop methods can be cumbersome. In addition, it has been observed that many Bayesian optimization methods with a moderate number of hyperparameters, when run for k iterations, can be outperformed by sampling $2k$ points uniformly at random (Li et al., 2018), indicating that even simple open loop methods can be competitive.

Parallelizing Bayesian optimization methods has proven to be nontrivial, though many agree that it's vitally important. While many algorithms exist which can sample more than one point at each iteration (Contal et al., 2013; Desautels et al., 2014; González et al., 2016; Kandasamy et al., 2018), the sequential nature of Bayesian optimization methods prevent the full parallelization open loop methods can employ. Even running two iterations (with batches of size $k/2$) will take on average twice as long as fully parallelizing the evaluations, as you can do with open loop methods like grid search, sampling uniformly, or sampling according to a DPP.

One line of research has examined the use of k-DPPs for optimizing hyperparameters in the context of parallelizing Bayesian optimization (Kathuria et al., 2016; Wang et al., 2017). At each iteration within one trial of Bayesian optimization, instead of drawing a single new point to evaluate from the posterior, they define a k-DPP over a relevance region from which they sample a diverse set of points. They found their approach to beat state-of-the-art performance on a number of hyperparameter optimization tasks, and they proved that generating batches by sampling from a k-DPP has better regret bounds than a number of other approaches. They show that a previous batch sampling approach which selects a batch by sequentially choosing a point which has the highest posterior variance (Contal et al., 2013) is just approximating finding the maximum probability set from a k-DPP (an NP-hard problem (Kulesza et al., 2012)), and they prove that sampling (as opposed to maximization) has better regret bounds for this optimization task. We use the work of Kathuria et al. (2016) as a foundation for our exploration of fully-parallel optimization methods, and thus we focus on k-DPP sampling as opposed to maximization.

So-called configuration evaluation methods have been shown to perform well by adaptively allocating resources to different hyperparameter settings (Swersky et al., 2014; Li et al., 2018). They initially choose a set of hyperparameters to evaluate (often uniformly), then partially train a set of models for these hyperparameters. After some fixed training budget (e.g., time, or number of training examples observed), they compare the partially trained models against one another and allocate more resources to those which perform best. Eventually, these algorithms produce one (or a small number) of fully trained, high-quality models. In some sense, these approaches are orthogonal to open vs. closed loop methods, as the diversity-promoting approach we advocate can be used as a drop-in replacement to the method used to choose the initial hyperparameter assignments.

3.1.2 Sampling proportional to the posterior variance of a Gaussian process

GPs have long been lauded for their expressive power, and have been used extensively in the hyperparameter optimization literature. Hennig and Garnett (2016) show that drawing a sample from a k-DPP with kernel $\mathcal{K}$ is equivalent to sequentially sampling k times proportional to the (updated) posterior variance of a GP defined with covariance kernel $\mathcal{K}$. This sequential sampling is one of the oldest hyperparameter optimization algorithms, though our work is the first to perform an in-depth analysis. Additionally, this has a nice information theoretic justification: since the entropy of a Gaussian is proportional to the log determinant of the covariance matrix, points drawn from a DPP have probability proportional to exp(information gain), and the most probable set from the DPP is the set which maximizes the information gain. With our MCMC algorithm presented in Algorithm 2, we can draw samples with these appealing properties from any space for which we can draw uniform samples. The ability to draw k-DPP samples by sequentially sampling points proportional to the posterior variance grants us another boon: if one has a sample of size k and wants a sample of size $k+1$, only a single additional point needs to be drawn, unlike with the sampling algorithms presented in Kulesza et al. (2012). Using this approach, we can draw samples up to $k = 100$ in less than a second on a machine with 32 cores.

3.1.3 Open Loop Methods

As discussed above, recent trends have renewed interest in open loop methods. While there exist many different batch BO algorithms, analyzing these in the open loop regime (when there are no results from function evaluations) is often rather simple. As there is no information with which to update the posterior mean, function evaluations are hallucinated using the prior or points are drawn only using information about the posterior variance. For example, in the open loop regime, Kandasamy et al. (2018)'s approach without hallucinated observations is equivalent to uniform sampling, and their approach with hallucinated observations (where they use the prior mean in place of a function evaluation, then update the posterior mean and variance) is equivalent to sequentially sampling according to the posterior variance (which is the same as sampling from a DPP). Similarly, open loop optimization in SMAC (Hutter et al., 2012) is equivalent to first Latin hypercube sampling to make a large set of diverse candidate points, then sampling k uniformly among these points.

Recently, uniform sampling was shown to be competitive with sophisticated closed loop methods for modern hyperparameter optimization tasks like optimizing the hyperparameters of deep neural networks (Li et al., 2018), inspiring other works to explain the phenomenon (Ahmed et al., 2016). Bergstra and Bengio (2012) offer one of the most comprehensive studies of open loop methods to date, and focus attention on comparing random search and grid search. A main takeaway of the paper is that uniform random sampling is generally preferred to grid search[1] due to the frequent observation that some hyperparameters have little impact on performance, and random search promotes more diversity in the dimensions that matter. Essentially, if points are drawn uniformly at random in d dimensions but only $d' < d$ dimensions are relevant, those same points are uniformly distributed (and just as diverse) in d' dimensions. Grid search, on the other hand, distributes configurations aligned with the axes so if only $d' < d$ dimensions are relevant, many configurations are essentially duplicates.

However, grid search does have one favorable property that is clear in just one dimension. If k points are distributed on $[0,1]$ on a grid, the maximum spacing between points is equal to $\frac{1}{k-1}$. But if points are uniformly at random drawn on $[0,1]$, the expected largest gap between points scales as $\frac{1}{\sqrt{k}}$. If, by bad luck, the optimum islocated in this largest gap, this difference could be considerable; we attempt to quantify this idea in the next section.

3.2 Measures of spread

Quantifying the spread of a sequence $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ (or, similarly, how well $\mathbf{x}$ covers a space) is a well-studied concept. In this section we introduce discrepancy, a quantity used by previous work, and dispersion, which we argue is more appropriate for optimization problems.

3.2.1 Discrepancy

Perhaps the most popular way to quantify the spread of a sequence is star discrepancy. One can interpret the star discrepancy as a multidimensional version of the Kolmogorov-Smirnov statistic between the sequence $\mathbf{x}$ and the uniform measure; intuitively, when $\mathbf{x}$ contains points which are spread apart, star discrepancy is small. Star discrepancy is defined as

$$D_k(\mathbf{x}) = \sup_{u_1,\ldots,u_d \in [0,1]} \left| \mathcal{A}_k(\mathbf{x}, u_j) - \prod_{j=1}^{d} u_j \right|, \text{where}$$

$$\mathcal{A}_k(\mathbf{x}, u_j) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}\left\{ x_i \in \textstyle\prod_{j=1}^{d} [0, u_j) \right\}.$$

It is well-known that a sequence chosen uniformly at random from $[0,1]^d$ has an expected star discrepancy of at least $\sqrt{\frac{1}{k}}$ (and is no greater than $\sqrt{\frac{d \log(d)}{k}}$) (Shalev-Shwartz and Ben-David, 2014) whereas sequences are known to exist with star discrepancy less than $\frac{\log(k)^d}{k}$ (Sobol', 1967), where both bounds depend on absolute constants.

[1]Grid search uniformly grids $[0,1]^d$ such that $x_k = (\frac{i_1}{m}, \frac{i_2}{m}, \ldots, \frac{i_d}{m})$ is a point on the grid for $i_j = 0, 1, \ldots, m$ for all j, with a total number of grid points equal to $(m+1)^d$.

Comparing the star discrepancy of sampling uniformly and Sobol, the bounds suggest that as d grows large relative to k, Sobol starts to suffer. Indeed, Bardenet and Hardy (2016) notes that the Sobol rate is not even valid until $k = \Omega(2^d)$ which motivates them to study a formulation of a DPP that has a star discrepancy between Sobol and random and holds for all k, small and large. They primarily approached this problem from a theoretical perspective, and didn't include experimental results. Their work, in part, motivates us to look at DPPs as a solution for hyperparameter optimization.

Star discrepancy plays a prominent role in the numerical integration literature, as it provides a sharp bound on the numerical integration error through the the Koksma-Hlawka inequality (given in Section 3.2.1) (Hlawka, 1961). This has led to wide adoption of low discrepancy sequences, even outside of numerical integration problems. For example, Bergstra and Bengio (2012) analyzed a number of low discrepancy sequences for some optimization tasks and found improved optimization performance over uniform sampling and grid search. Additionally, low discrepancy sequences such as the Sobol sequence[2] are used as an initialization procedure for some Bayesian optimization schemes (Snoek et al., 2012).

Koksma-Hlawka inequality

Let $\mathcal{B}$ be the d-dimensional unit cube, and let f have bounded Hardy and Krause variation $Var_{HK}(f)$ on $\mathcal{B}$. Let $\mathbf{x} = (x_1, x_2, \ldots, x_k)$ be a set of points in $\mathcal{B}$ at which the function f will be evaluated to approximate an integral. The Koksma-Hlawka inequality bounds the numerical integration error by the product of the star discrepancy and the variation:

$$\left| \frac{1}{k} \sum_{i=1}^{k} f(x_i) - \int_{\mathcal{B}} f(u) du \right| \leq Var_{HK}(f) D_k(\mathbf{x}).$$

We can see that for a given f, finding $\mathbf{x}$ with low star discrepancy can improve numerical integration approximations.

3.2.2 Dispersion

Previous work on open loop hyperparameter optimization focused on low discrepancy sequences (Bergstra and Bengio, 2012; Bousquet et al., 2017), but optimization performance—how close a point in our sequence is to the true, fixed optimum—is our goal, not a sequence with low discrepancy. As discrepancy doesn't directly bound optimization error, we turn instead to dispersion

$$d_k(\mathbf{x}) = \sup_{x \in [0,1]^d} \min_{1 \leq i \leq k} \rho(x, x_i),$$

[2]Bergstra and Bengio (2012) found that the Niederreiter and Halton sequences performed similarly to the Sobol sequence, and that the Sobol sequence outperformed Latin hypercube sampling. Thus, our experiments include the Sobol sequence (with the Cranley-Patterson rotation) as a representative low-discrepancy sequence.

where ρ is a distance (in our experiments L_2 distance). Intuitively, the dispersion of a point set is the radius of the largest Euclidean ball containing no points; dispersion measures the worst a point set could be at finding the optimum of a space.

Following (Niederreiter, 1992), we can bound the optimization error as follows. Let f be the function we are aiming to optimize (maximize) with domain $\mathcal{B}$, $m(f) = \sup_{x \in \mathcal{B}} f(x)$ be the global optimum of the function, and $m_k(f; \mathbf{x}) = \sup_{1 \le i \le k} f(x_i)$ be the best-found optimum from the set $\mathbf{x}$. Assuming f is continuous (at least near the global optimum), the modulus of continuity is defined as

$$\omega(f; t) = \sup_{\substack{x,y \in \mathcal{B} \\ \rho(x,y) \le t}} |f(x) - f(y)|, \text{ for some } t \ge 0.$$

Theorem 1. *(Niederreiter, 1992) For any point set $\mathbf{x}$ with dispersion $d_k(\mathbf{x})$, the optimization error is bounded as*

$$m(f) - m_k(f; \mathbf{x}) \le \omega(f; d_k(\mathbf{x})).$$

Dispersion can be computed (somewhat) efficiently (unlike discrepancy, $D_k(\mathbf{x})$, which is NP-hard (Zhigljavsky and Zilinskas, 2007)). One algorithm is to find a (bounded) voronoi diagram over the search space for a point set X_k. Then, for each vertex in the voronoi diagram, find the closest point in X_k. The dispersion is the max over these distances. This is how it is computed in this section.

Discrepancy is a global measure which depends on all points, while dispersion only depends on points near the largest "hole". Dispersion is at least $\Omega(k^{-1/d})$, and while low discrepancy implies low dispersion ($d^{-1/2} d_k(\mathbf{x}) \le \frac{1}{2} D_k(\mathbf{x})^{1/d}$), the other direction does not hold. Therefore we know that the low-discrepancy sequences evaluated in previous work are also low-dispersion sequences in the big-O sense, but as we will see they may behave quite differently. Samples drawn uniformly are not low dispersion, as they have rate $(\ln(k)/k)^{1/d}$ (Zhigljavsky and Zilinskas, 2007).

Optimal dispersion in one dimension is found with an evenly spaced grid, but it's unknown how to get an optimal set in higher dimensions.[3] Finding a set of points with the optimal dispersion is as hard as solving the circle packing problem in geometry with k equal-sized circles which are as large as possible. Dispersion is bounded from below with $d_k(\mathbf{x}) \ge \left(\Gamma(d/2+1)\right)^{1/d} \pi^{-1/2} k^{-1/d}$, though it is unknown if this bound is sharp.

[3]In two dimensions a hexagonal tiling finds the optimal dispersion, but this is only valid when k is divisible by the number of columns and rows in the tiling.

3.2.3 Distance to the center and the origin

One natural surrogate of average optimization performance is to define a hyperparameter space on $[0,1]^d$ and measure the distance from a fixed point, say $\frac{1}{2}\mathbf{1} = (\frac{1}{2}, \ldots, \frac{1}{2})$, to the nearest point in the length k sequence in the Euclidean norm squared: $\min\limits_{i=1,\ldots,k} ||x_i - \frac{1}{2}\mathbf{1}||_2^2$. The Euclidean norm (squared) is motivated by a quadratic Taylor series approximation around the minimum of the hypothetical function we wish to minimize. In the first columns of Figure 3.1 we plot the smallest distance from the center $\frac{1}{2}\mathbf{1}$, as a function of the length of the sequence (in one dimension) for the Sobol sequence, uniform at random, and a DPP. We observe all methods appear comparable when it comes to distance to the center.

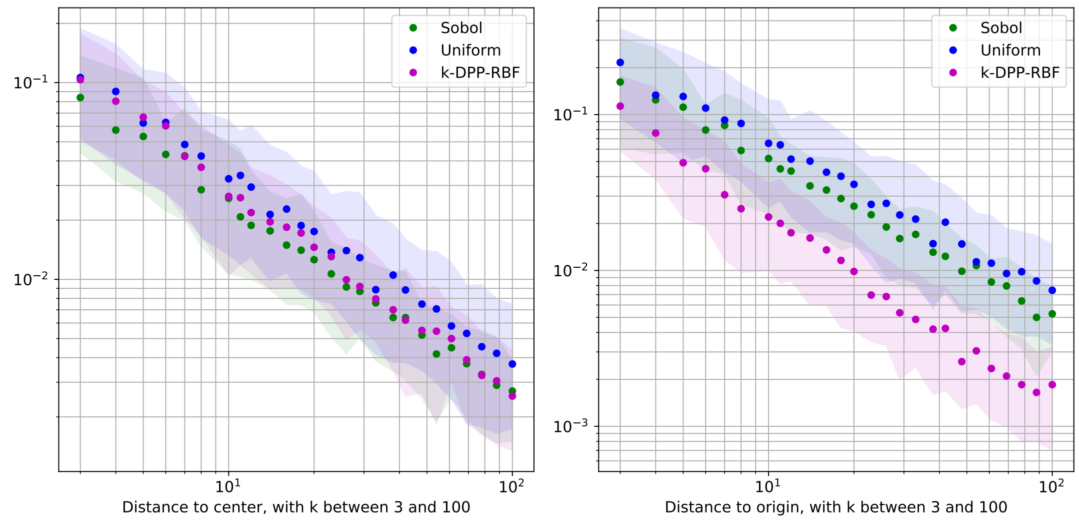


Figure 3.1: Comparison of the Sobol sequence, samples a from k-DPP, and uniform random for two metrics of interest. These log-log plots show uniform sampling and k-DPP-RBF performs comparably to the Sobol sequence in terms of distance to the center, but on another (distance to the origin) k-DPP-RBF samples outperform the Sobol sequence and uniform sampling.

Acknowledging the fact that practitioners define the search space themselves more often than not, we realize that if the search space bounds are too small, the optimal solution often is found on the edge, or in a corner of the hypercube (as the true global optima are outside the space). Thus, in some situations it makes sense to *bias* the sequence towards the edges and the corners, the very opposite of what low discrepancy sequences attempt to do. While Sobol and uniformly random sequences will not bias themselves towards the corners, a DPP does. This happens because points from a DPP are sampled according to how distant they are from the existing points; this tends to favor points in the corners. This same behavior of sampling in the corners is also very common for Bayesian optimization schemes, which is not surprise due to the known connections between sampling from a DPP and Gaussian processes (see Section 3.1.2). In the second column of Figure 3.1 we plot the distance to the origin which is just an arbitrarily chosen corner of hypercube. As expected, we observe that the DPP tends to outperform uniform at random and Sobol in this metric.

3.2.4 Comparison of Open Loop Methods

In Figure 3.2 we plot the dispersion of the Sobol sequence, samples drawn uniformly at random, and samples drawn from a k-DPP, in one and two dimensions. To generate the k-DPP samples, we sequentially drew samples proportional to the (updated) posterior variance (using an RBF kernel, with $\sigma = \sqrt{2}/k$), as described in Section 3.1.2. When $d = 1$, the regular structure of the Sobol sequence causes it to have increasingly large plateaus, as there are many "holes" of the same size.[4] For example, the Sobol sequence has the same dispersion for $42 \leq k \leq 61$, and $84 \leq k \leq 125$. Samples drawn from a k-DPP appear to have the same asymptotic rate as the Sobol sequence, but they don't suffer from the plateaus. When $d = 2$, the k-DPP samples have lower average dispersion and lower variance.

One other natural surrogate of average optimization performance is to measure the distance from a fixed point, say $\frac{1}{2}\mathbf{1} = (\frac{1}{2}, \ldots, \frac{1}{2})$ or from the origin, to the nearest point in the length k sequence. Our experiments (in Appendix 3.2.3) on these metrics show the k-DPP samples bias samples to the corners of the space, which can be beneficial when the practitioner defined the search space with bounds that are too small.

Note, the low-discrepancy sequences are usually defined only for the $[0, 1]^d$ hypecrube, so for hyperparameter search which involves conditional hyperparameters (i.e. those with tree structure) they are not appropriate. In what follows, we study the k-DPP in more depth and how it performs on real-world hyperparameter tuning problems.

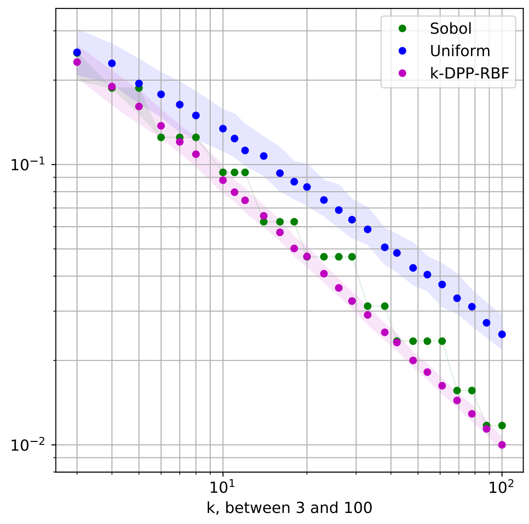
3.3 Method

We begin by reviewing DPPs and k-DPPs. Let $\mathcal{B}$ be a domain from which we would like to sample a finite subset. (In our use of DPPs, this is the set of hyperparameter assignments.) In general, $\mathcal{B}$ could be discrete or continuous; here we assume it is discrete with N values, and we define $\mathcal{Y} = \{1, \ldots, N\}$ to be a a set which indexes $\mathcal{B}$ (this index set will be particularly useful in Algorithm 1). In Section 3.3.2 we address when $\mathcal{B}$ has continuous dimensions. A DPP defines a probability distribution over $2^{\mathcal{Y}}$ (all subsets of $\mathcal{Y}$) with the property that two elements of $\mathcal{Y}$ are more (less) likely to both be chosen the more dissimilar (similar) they are. Let random variable $\boldsymbol{Y}$ range over finite subsets of $\mathcal{Y}$.

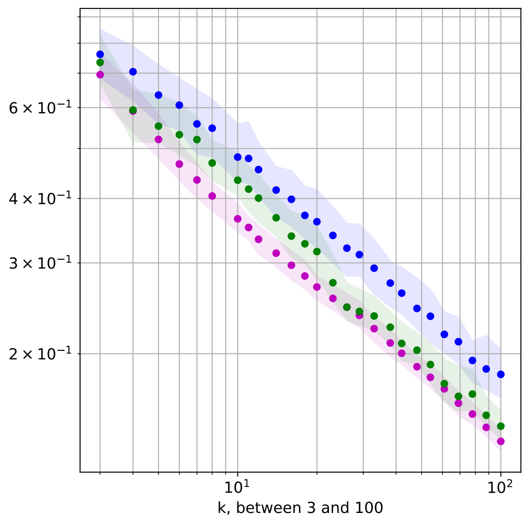
There are several ways to define the parameters of a DPP. We focus on $\mathbf{L}$-ensembles, which define the probability that a specific subset is drawn (i.e., $P(\boldsymbol{Y} = \mathcal{A})$ for some $\mathcal{A} \subset \mathcal{Y}$) as:

$$P(\boldsymbol{Y} = \mathcal{A}) = \frac{\det(\mathbf{L}_{\mathcal{A}})}{\det(\mathbf{L} + I)}. \tag{3.1}$$

[4]By construction, each individual dimension of the d-dimensional Sobol sequence has these same plateaus.



(a) Dispersion, with $d = 1$.



(b) Dispersion, with $d = 2$.

Figure 3.2: Dispersion of the Sobol sequence, samples from a k-DPP, and uniform random samples (lower is better). These log-log plots show when $d = 1$ that Sobol suffers from regular plateaus of increasing size, while when $d = 2$ the k-DPP samples have lower average dispersion and lower variance.

As shown in (Kulesza et al., 2012), this definition of $\mathbf{L}$ admits a decomposition to terms representing the *quality* and *diversity* of the elements of $\mathcal{Y}$. For any $y_i, y_j \in \mathcal{Y}$, let:

$$\mathbf{L}_{i,j} = q_i q_j \mathcal{K}(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j), \tag{3.2}$$

where $q_i > 0$ is the quality of y_i, $\boldsymbol{\phi}_i \in \mathbb{R}^d$ is a featurized representation of y_i, and $\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$ is a similarity kernel (e.g. cosine distance). (We will discuss how to featurize hyperparameter settings in Section 3.3.3.)

Here, we fix all $q_i = 1$; in future work, closed loop methods might make use of q_i to encode evidence about the quality of particular hyperparameter settings to adapt the DPP's distribution over time.

3.3.1 Sampling from a k-DPP

DPPs have support over all subsets of $\mathcal{Y}$, including $\emptyset$ and $\mathcal{Y}$ itself. In many practical settings, one may have a fixed budget that allows running the training algorithm k times, so we require precisely k elements of $\mathcal{Y}$ for evaluation. k-DPPs are distributions over subsets of $\mathcal{Y}$ of size k. Thus,

$$P(\boldsymbol{Y} = \mathcal{A} \mid |\boldsymbol{Y}| = k) = \frac{\det(\mathbf{L}_{\mathcal{A}})}{\sum_{\mathcal{A}' \subset \mathcal{Y}, |\mathcal{A}'| = k} \det(\mathbf{L}_{\mathcal{A}'})}. \tag{3.3}$$

Sampling from k-DPPs has been well-studied. When the base set $\mathcal{B}$ is a set of discrete items, exact sampling algorithms are known which run in $\mathcal{O}(Nk^3)$ (Kulesza et al., 2012). When the base set is a continuous hyperrectangle, a recent exact sampling algorithm was introduced, based on a connection with Gaussian processes (GPs), which runs in $\mathcal{O}(dk^2 + k^3)$ (Hennig and Garnett, 2016). We are unaware of previous work which allows for sampling from k-DPPs defined over any other base sets.

3.3.2 Sampling k-DPPs defined over arbitrary base sets

Anari et al. (2016) present a Metropolis-Hastings algorithm (included here as Algorithm 1) which is a simple and fast alternative to the exact sampling procedures described above. However, it is restricted to discrete domains. We propose a generalization of the MCMC algorithm which preserves relevant computations while allowing sampling from any base set from which we can draw uniform samples, including those with discrete dimensions, continuous dimensions, some continuous and some discrete dimensions, or even (conditional) tree structures (Algorithm 2). To the best of our knowledge, this is the first algorithm which allows for sampling from a k-DPP defined over any space other than strictly continuous or strictly discrete, and thus the first algorithm to utilize the expressive capabilities of the posterior variance of a GP in these regimes.

Algorithm 1 Drawing a sample from a discrete k-DPP (Anari et al., 2016)

```
Input: $\mathbf{L}$, a symmetric, $N \times N$ matrix where $\mathbf{L}_{i,j} = q_i q_j \mathcal{K}(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$ which defines a DPP over a
    finite base set of items $\mathcal{B}$, and $\mathcal{Y} = \{1, \ldots, N\}$, where $\mathcal{Y}_i$ indexes a row or column of $\mathbf{L}$
Output: $\mathcal{B}_\mathbf{Y}$ (the points in $\mathcal{B}$ indexed by $\mathbf{Y}$)
1: Initialize $\mathbf{Y}$ to $k$ elements sampled from $\mathcal{Y}$ uniformly
2: while not mixed do
3:     uniformly sample $u \in \mathbf{Y}, v \in \mathcal{Y} \setminus \mathbf{Y}$
4:     set $\mathbf{Y}' = \mathbf{Y} \cup \{v\} \setminus \{u\}$
5:     $p \leftarrow \frac{1}{2} min(1, \frac{\det(\mathbf{L}_{\mathbf{Y}'})}{\det(\mathbf{L}_\mathbf{Y})})$
6:     with probability $p$: $\mathbf{Y} = \mathbf{Y}'$
7: Return $\mathcal{B}_\mathbf{Y}$
```

Algorithm 1 proceeds as follows: First, initialize a set $\mathbf{Y}$ with k indices of $\mathbf{L}$, drawn uniformly. Then, at each iteration, sample two indices of $\mathbf{L}$ (one within and one outside of the set $\mathbf{Y}$), and with some probability replace the item in $\mathbf{Y}$ with the other.

When we have continuous dimensions in the base set, however, we can't define the matrix $\mathbf{L}$, so sampling indices from it is not possible. We propose Algorithm 2, which samples points directly from the base set $\mathcal{B}$ instead (assuming continuous dimensions are bounded), and computes only the principal minors of $\mathbf{L}$ needed for the relevant computations on the fly.

Even in the case where the dimensions of $\mathcal{B}$ are discrete, Algorithm 2 requires less computation and space than Algorithm 1 (assuming the quality and similarity scores are stored once computed, and retrieved when needed). Previous analyses claimed that Algorithm 1 should mix after $\mathcal{O}(N \log(N))$ steps. There are $\mathcal{O}(N^2)$ computations required to compute the full matrix L, and at each iteration we will compute at most $O(k)$ new elements of L, so even in the worst

Algorithm 2 Drawing a sample from a k-DPP defined over a space with continuous and discrete dimensions

```
Input: A base set $\mathcal{B}$ with some continuous and some discrete dimensions, a quality function
    $\mathbf{\Psi} : \mathbf{Y}_i \rightarrow q_i$, a feature function $\mathbf{\Phi} : \mathbf{Y}_i \rightarrow \boldsymbol{\phi}_i$
Output: $\boldsymbol{\beta}$, a set of $k$ points in $\mathcal{B}$
 1: Initialize $\boldsymbol{\beta}$ to $k$ points sampled from $\mathcal{B}$ uniformly
 2: while not mixed do
 3:     uniformly sample $u \in \boldsymbol{\beta}, v \in \mathcal{B} \setminus \boldsymbol{\beta}$
 4:     set $\boldsymbol{\beta}' = \boldsymbol{\beta} \cup \{v\} \setminus \{u\}$
 5:     compute the quality score for each item, $q_i = \mathbf{\Psi}(\boldsymbol{\beta}_i), \forall i$, and $q'_i = \mathbf{\Psi}(\boldsymbol{\beta}'_i), \forall i$
 6:     construct $\mathbf{L}_{\boldsymbol{\beta}} = [q_i q_j \mathcal{K}(\mathbf{\Phi}(\boldsymbol{\beta}_i), \mathbf{\Phi}(\boldsymbol{\beta}_j))], \forall i, j$
 7:     construct $\mathbf{L}_{\boldsymbol{\beta}'} = [q'_i q'_j \mathcal{K}(\mathbf{\Phi}(\boldsymbol{\beta}'_i), \mathbf{\Phi}(\boldsymbol{\beta}'_j))], \forall i, j$
 8:     $p \leftarrow \frac{1}{2} min(1, \frac{\det(\mathbf{L}_{\boldsymbol{\beta}'})}{\det(\mathbf{L}_{\boldsymbol{\beta}})})$
 9:     with probability $p$: $\boldsymbol{\beta} = \boldsymbol{\beta}'$
10: Return $\boldsymbol{\beta}$
```

case we will save space and computation whenever $k \log(N) < N$. In expectation, we will save significantly more.

3.3.3 Constructing L for hyperparameter optimization

Let $\boldsymbol{\phi}_i$ be a feature vector for $y_i \in \mathcal{Y}$, a modular encoding of the attribute-value mapping assigning values to different hyperparameters, in which fixed segments of the vector are assigned to each hyperparameter attribute (e.g., the dropout rate, the choice of nonlinearity, etc.). For a hyperparameter that takes a numerical value in range $[h_{\min}, h_{\max}]$, we encode value h using one dimension (j) of $\boldsymbol{\phi}$ and project into the range $[0, 1]$:

$$\boldsymbol{\phi}[j] = \frac{h - h_{\min}}{h_{\max} - h_{\min}} \tag{3.4}$$

This rescaling prevents hyperparameters with greater dynamic range from dominating the similarity calculations. A categorical-valued hyperparameter variable that takes m values is given m elements of $\boldsymbol{\phi}$ and a one-hot encoding. Ordinal-valued hyperparameters can be encoded using a unary encoding. (For example, an ordinal variable which can take three values would be encoded with [1,0,0], [1,1,0], and [1,1,1].) Additional information about the distance between the values can be incorporated, if it's available. In this work, we then compute similarity using an RBF kernel, $\mathcal{K} = \exp\left(-\frac{||\boldsymbol{\phi}_i - \boldsymbol{\phi}_j||^2}{2\sigma^2}\right)$, and hence label our approach k-DPP-RBF. Values for σ^2 lead to models with different properties; when σ^2 is small, points that are spread out interact little with one another, and when σ^2 is large, the increased repulsion between the points encourages them to be as far apart as possible.

3.3.4 Tree-structured hyperparameters

Many real-world hyperparameter search spaces are tree-structured. For example, the number of layers in a neural network is a hyperparameter, and each additional layer adds at least one new hyperparameter which ought to be tuned (the number of nodes in that layer). For a binary hyperparameter like whether or not to use regularization, we use a one-hot encoding. When this hyperparameter is "on," we set the associated regularization strength as above, and when it is "off" we set it to zero. Intuitively, with all other hyperparameter settings equal, this causes the off-setting to be closest to the least strong regularization. One can also treat higher-level design decisions as hyperparameters (Komer et al., 2014), such as whether to train a logistic regression classifier, a convolutional neural network, or a recurrent neural network. In this construction, the type of model would be a categorical variable (and thus get a one-hot encoding), and all child hyperparameters for an "off" model setting (such as the convergence tolerance for logistic regression, when training a recurrent neural network) would be set to zero.

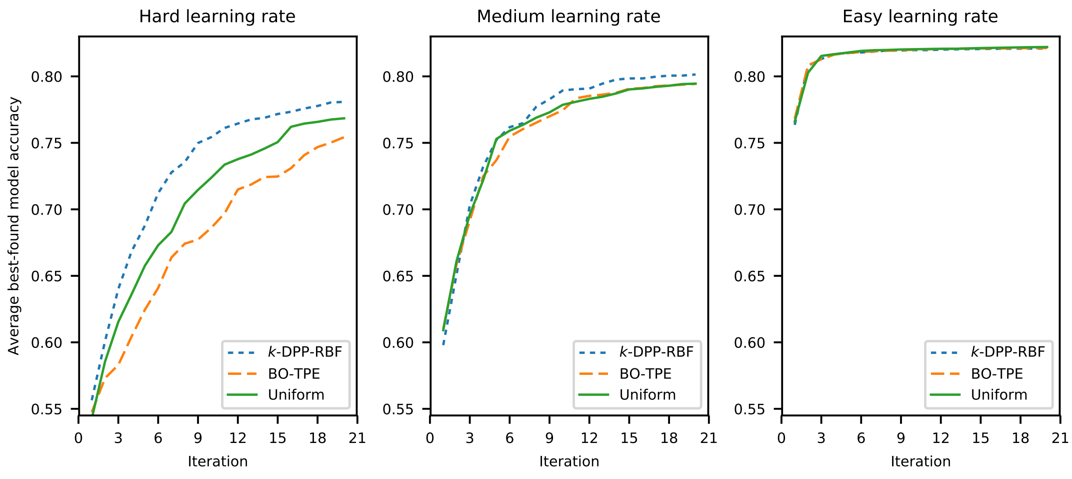


Figure 3.3: Average best-found model accuracy by iteration when training a convolutional neural network on three hyperparameter search spaces (defined in Section 3.4.1), averaged across 50 trials of hyperparameter optimization, with $k = 20$.

3.4 Hyperparameter Optimization Experiments

In this section we present our hyperparameter optimization experiments. Our experiments consider a setting where hyperparameters have a large effect on performance: a convolutional neural network for text classification (Kim, 2014). The task is binary sentiment analysis on the Stanford sentiment treebank (Socher et al., 2013). On this balanced dataset, random guessing leads to 50% accuracy. We use the CNN-non-static model from Kim (2014), with skip-gram (Mikolov et al., 2013) vectors. The model architecture consists of a convolutional layer, a max-over-time pooling layer, then a fully connected layer leading to a softmax. All k-DPP samples are drawn using Algorithm 2.

3.4.1 Simple tree-structured space

We begin with a search over three continuous hyperparameters and one binary hyperparameter, with a simple tree structure: the binary hyperparameter indicates whether or not the model will use L_2 regularization, and one of the continuous hyperparameters is the regularization strength. We assume a budget of $k = 20$ evaluations by training the convolutional neural net. L_2 regularization strengths in the range $[e^{-5}, e^{-1}]$ (or no regularization) and dropout rates in $[0.0, 0.7]$ are considered. We consider three increasingly "easy" ranges for the learning rate:

- Hard: $[e^{-5}, e^{5}]$, where the majority of the range leads to accuracy no better than chance.
- Medium: $[e^{-5}, e^{-1}]$, where half of the range leads to accuracy no better than chance.
- Easy: $[e^{-10}, e^{-3}]$, where the entire range leads to models that beat chance.

Figure 3.3 shows the accuracy (averaged over 50 runs) of the best model found after exploring 1, 2, . . . , k hyperparameter settings. We see that k-DPP-RBF finds better models with fewer iterations necessary than the other approaches, especially in the most difficult case. Figure 3.3 compares the sampling methods against a Bayesian optimization technique using a tree-structured Parzen estimator (BO-TPE; Bergstra et al., 2011). This technique evaluates points sequentially, allowing the model to choose the next point based on how well previous points performed (a closed loop approach). It is state-of-the-art on tree-structured search spaces (though its sequential nature limits parallelization). Surprisingly, we find it performs the worst, even though it takes advantage of additional information. We hypothesize that the exploration/exploitation tradeoff in BO-TPE causes it to commit to more local search before exploring the space fully, thus not finding hard-to-reach global optima.

Note that when considering points sampled uniformly or from a DPP, the order of the k hyperparameter settings in one trial is arbitrary (though this is not the case with BO-TPE as it is an iterative algorithm). In all cases the variance of the best of the k points is lower than when sampled uniformly, and the differences in the plots are all significant with $p < 0.01$.

3.4.2 Optimizing within ranges known to be good

Zhang and Wallace (2015) analyzed the stability of convolutional neural networks for sentence classification with respect to a large set of hyperparameters, and found a set of six which they claimed had the largest impact: the number of kernels, the difference in size between the kernels, the size of each kernel, dropout, regularization strength, and the number of filters. We optimized over their prescribed "Stable" ranges for three open loop methods and one closed loop method; average accuracies with 95 percent confidence intervals from 50 trials of hyperparameter

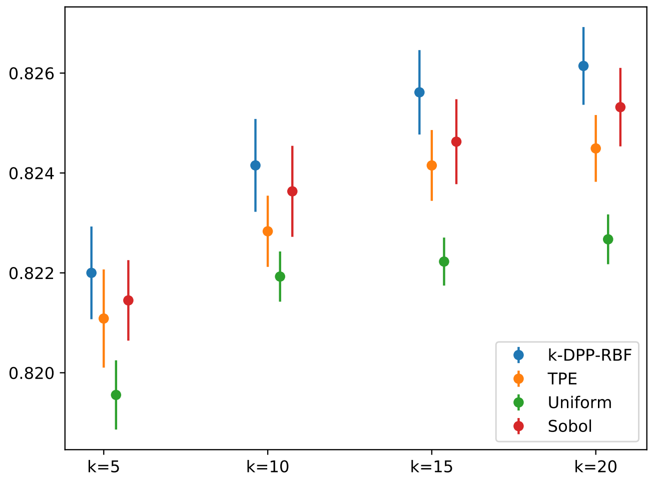


Figure 3.4: Average best-found model accuracy by iteration when training a convolutional neural network on the “Stable” search space (defined in Section 3.4.2), averaged across 50 trials of hyperparameter optimization, with $k = 5, 10, 15, 20$, with 95 percent confidence intervals. The k-DPP-RBF outperforms uniform sampling, TPE, and the Sobol sequence.

optimization are shown in Figure 3.4, across $k = 5, 10, 15, 20$ iterations. We find that even when optimizing over a space for which all values lead to good models, k-DPP-RBF outperforms the other methods.

Our experiments reveal that, while the hyperparameters proposed by Zhang and Wallace (2015), can have an effect, the learning rate, which they do not analyze, is at least as impactful.

3.4.3 Wall clock time comparison with Spearmint

Here we compare our approach against Spearmint (Snoek et al., 2012), perhaps the most popular Bayesian optimization package. Figure 3.5 shows wall clock time and accuracy for 25 runs on the “Stable” search space of four hyperparameter optimization approaches: k-DPP-RBF (with $k = 20$), batch Spearmint with 2 iterations of batch size 10, batch Spearmint with 10 iterations of batch size 2, and sequential Spearmint[5]. Each point in the plot is one hyperparameter assignment evaluation. The vertical lines represent how long, on average, it takes to find the best result in one run. We see that all evaluations for k-DPP-RBF finish quickly, while even the fastest batch method (2 batches of size 10) takes nearly twice as long on average to find a good result. The final average best-found accuracies are 82.61 for k-DPP-RBF, 82.65 for Spearmint with 2 batches of size 10, 82.7 for Spearmint with 10 batches of size 2, and 82.76 for sequential Spearmint. Thus, we find it takes on average more than ten times as long for sequential Spearmint to find its best solution, for a gain of only 0.15 percent accuracy.

[5]When in the fully parallel, open loop setting, Spearmint simply returns the Sobol sequence.

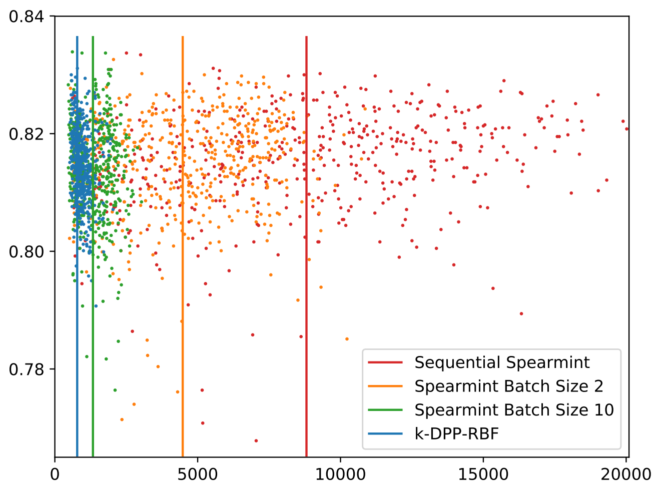


Figure 3.5: Wall clock time (in seconds, x-axis) for 25 hyperparameter trials of hyperparameter optimization (each with $k = 20$) on the "Stable" search space define in Section 3.4.2. The vertical lines represent the average time it takes too find the best hyperparameter assignment in a trial.

3.5 Conclusions

This chapter provides an approach for reducing the computational expense of H, hyperparameter tuning, in the Green AI equation. We explored open loop hyperparameter optimization built on sampling from a k-DPP. We described how to define a k-DPP over hyperparameter search spaces, and showed that k-DPPs retain the attractive parallelization capabilities of random search. In synthetic experiments, we showed k-DPP samples perform well on a number of important metrics, even for large values of k. In hyperprameter optimization experiments, we see k-DPP-RBF outperform other open loop methods. Additionally, we see that sequential methods, even when using more than ten times as much wall clock time, gain less than 0.16 percent accuracy on a particular hyperparameter optimization problem. An open-source implementation of our method is available.

Chapter 4

Structured Sparsity for Parameter Efficient Neural Models

State-of-the-art neural models for NLP are heavily parameterized, requiring hundreds of millions (Devlin et al., 2019) and even billions (Radford et al., 2019) of parameters. While over-parameterized models can be easier to train (Livni et al., 2014), they may also introduce memory problems on small devices, as well as contribute to overfitting. This chapter presents a structure learning method for learning sparse, parameter-efficient models. Thus, this addresses E in the Green AI equation, the cost of processing a single (E)xample. This chapter extends Dodge et al. (2019c).

In feature-based NLP, structured-sparse regularization, in particular the group lasso (Section 4.1.1, Yuan and Lin, 2006), has been proposed as a method to reduce model size while preserving performance (Martins et al., 2011). But, in neural NLP, some of the most widely used models—LSTMs (Hochreiter and Schmidhuber, 1997) and GRUs (Cho et al., 2014)—do not have an obvious, intuitive notion of "structure" in their parameters (other than, perhaps, division into layers), so the use of structured sparsity at first may appear incongruous.

In this chapter we show that group lasso can be successfully applied to neural NLP models. We focus on a family of neural models for which the hidden state exhibits a natural structure: rational RNNs (Section 4.1.2; Peng et al., 2018). In a rational RNN, the value of each hidden dimension is the score of a weighted finite-state automaton (WFSA) on (a prefix of) the input vector sequence. This property offers a natural grouping of the transition function parameters for each WFSA. As shown by Schwartz et al. (2018) and Peng et al. (2018), a variety of state-of-the-art neural architectures are rational (Lei et al., 2017; Bradbury et al., 2017; Foerster et al., 2017, *inter alia*), so learning parameter-efficient rational RNNs is of practical value. Rational

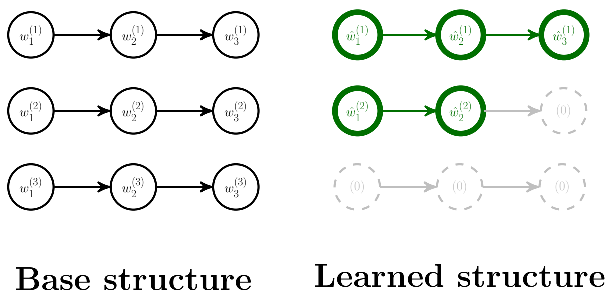


Figure 4.1: Our approach learns a sparse structure (right hand side) of a base rational RNN (left hand side) where each hidden unit corresponds to a WFSA (in this example, three hidden units, represented by the three rows). Grayed-out, dashed states are removed from the model, while retained states are marked in **bold green**.

RNNs also introduce a natural interpretation, in the form of "soft" patterns (Schwartz et al., 2018), which our method leverages.

We apply a group lasso penalty to the WFSA parameters of rational RNNs during training, where each group is comprised of the parameters associated with one state in one WFSA (Figure 4.1; Section 4.2). This penalty pushes the parameters in some groups to zero, effectively eliminating them, and making the WFSA smaller. When all of the states for a given WFSA are eliminated, the WFSA is removed entirely, so this approach can be viewed as learning the number of WFSAs (i.e., the RNN hidden dimension) as well as their size. We then retain the sparse structure, which results in a much smaller model in terms of parameters.

We experiment with four text classification benchmarks (Section 4.3), using both GloVe (Pennington et al., 2014) and contextual BERT (Devlin et al., 2019) embeddings. Our results show that as we vary the regularization strengths, we end up with smaller models. These models have a better tradeoff between the number of parameters and model performance compared to setting the number of WFSAs and their lengths by hand or using hyperparameter search. In almost all cases, our approach results in a model with fewer parameters and similar or better performance as our baseline models. In contrast to neural architecture search (Jozefowicz et al., 2015; Zoph and Le, 2017), which can take several GPU years to learn an appropriate neural architecture, our approach requires only two training runs: one to learn the structure, and the other to estimate its parameters.

Finally, our approach touches on another appealing property of rational RNNs—their interpretability. As shown by Schwartz et al. (2018), each WFSA captures a "soft" version of patterns like "such a great X", and can be visualized as such. By retaining a smaller number of WFSAs, our method allows to visualize the entire model succinctly. For example, we show in Section 4.4 that some of our sentiment analysis models rely exclusively on as few as *three* WFSAs (that is, a rational RNN with hidden size 3). We publicly release our implementation at https://github.com/dodgejesse/sparsifying_regularizers_for_RRNNs.

4.1 Background

We aim to learn parameter-efficient models using sparse regularization. Methods that encourage sparsity like group lasso (Yuan and Lin, 2006) assume that features can be meaningfully grouped. Such methods were popular when applied to linear models in NLP (Martins et al., 2011), where handcrafted features were grouped by the templates they instantiated. In neural NLP models, parameters are typically less interpretable, and hence it is not straightforward how to group them (Doshi-Velez, 2017).

In this work we apply sparse regularization to rational RNNs, which allows for learning meaningful sparse structures, by eliminating states or transitions, and even whole WFSAs. The main contribution of this chapter is applying group lasso to automate choices about which states or WFSAs to eliminate, thereby learning the neural structure itself. We briefly review group lasso and rational RNNs.

4.1.1 Group Lasso Penalty

This section reviews ℓ_1 regularizer and group lasso. When added to a learner's loss, the ℓ_1 norm of a parameter vector (i.e., sum of absolute values of its elements) acts as a convex, subdifferentiable approximation to the ℓ_0 penalty, known as the **lasso** (Tibshirani, 1996). Solving an ℓ_0 regularized problem has been shown to be NP-Hard (Candes and Tao, 2005). Like the more widely used squared ℓ_2 ("ridge") regularizer, the lasso is used to improve generalization, usually in linear models. It also induces sparsity, by driving some of the parameters to zero, which (in the linear setting) is equivalent to eliminating the corresponding features. Thus the lasso can promote parameter efficiency, and has also been argued to improve interpretability (Tibshirani, 1996). Like all regularizers, the lasso term is typically scaled by a regularization strength hyperparameter (denoted by λ), so that the regularized learning problem is:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) + \lambda \sum_i |w_i|, \tag{4.1}$$

where $\mathcal{L}$ is the training loss, and w_i is a scalar parameter in the model.

Group lasso (Yuan and Lin, 2006) generalizes the lasso to cases where a grouping of the parameters exists. Here sparsity works at the group level, driving all the parameters in a group towards zero. Denote the parameter vector for the g^{th} group by $\mathbf{w}_g$ (a subvector of $\mathbf{w}$). The group lasso-regularized learning problem with G groups is then:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) + \lambda \sum_{g=1}^{G} \sqrt{\dim(\mathbf{w}_g)}\, \|\mathbf{w}_g\|_2\,, \tag{4.2}$$

with dim(·) denoting the dimension of a vector. This weighting of groups, by the square root of their sizes, is one conventional approach, but others are possible. If every parameter is in its own group, the original lasso is recovered.

Lasso and group lasso were developed as tools for feature selection in linear models, but have recently been applied to neural networks (Scardapane et al., 2017; Gordon et al., 2018). They are both sparsifying regularizers that push parameters to zero, with different inductive biases: lasso regularizes individual parameters independently, while group lasso jointly regularizes groups of parameters. In its simplest form, the groups do not overlap; overlapping groups are possible but require more sophisticated optimization techniques; see Yuan et al. (2011) for a related discussion. Our groups are non-overlapping. We exploit the structure in rational RNNs for meaningful parameter grouping (i.e., each group contains parameters associated with a WFSA state), so that removing a group directly shrinks a WFSA and hence the overall architecture.

4.1.2 WFSAs and Rational RNNs

We define a weighted finite-state automaton (WFSA) as a tuple $\langle \Sigma, \mathcal{Q}, q_0, \mathcal{F}, \mathcal{T} \rangle$. Σ is a finite alphabet, and the WFSA assigns scores to strings in Σ^*. $\mathcal{Q}$ is a finite set of states, $q_0 \in \mathcal{Q}$ a designated start state, and $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states. $\mathcal{T}$ is a set of transitions, each a tuple of source state, target state, symbol from Σ, and weight. Start and final states can also be weighted. In this work we weight them by constant 1, and suppress the definition of start and final weight functions, simplifying notations. A path consists of a sequence of transitions, and its score is computed as the product of the transition weights. The score of an input string summarizes all the paths deriving it, either by taking the sum of all paths or the maximum-scoring path, typically calculated using dynamic programming (Baum and Petrie, 1966). For more discussion about WFSAs, see Kuich and Salomaa (1986).

Recently, Schwartz et al. (2018) and Peng et al. (2018) showed that several recently proposed recurrent architectures, including SRU (Lei et al., 2017) and quasi-RNN (Bradbury et al., 2017), are different implementations of specific WFSAs. They named this family of models rational recurrent neural networks (*rational RNNs*). [1]

Rational RNNs admit straightforward design of neural architectures. For instance, the one depicted in Figure 4.2 captures 4-gram patterns (Peng et al., 2018). Notably, unlike traditional WFSA-based models, rational RNNs compute their WFSA scores based on word vectors (rather than word symbols), and can thus be seen as capturing "soft" patterns (Schwartz et al., 2018).

An important property of rational RNNs is that, each WFSA state is parameterized by a separate set of parameters. We exploit this by treating a WFSA state's parameters as a group, and applying group lasso penalty to it. This results in dropping some states in some WFSAs (effectively making them smaller); those WFSAs with all of their regularized states dropped

[1] *Rational* follows the terminology *rational power series* (Berstel and Reutenauer, 1988),the mathematical counterpart of WFSAs.

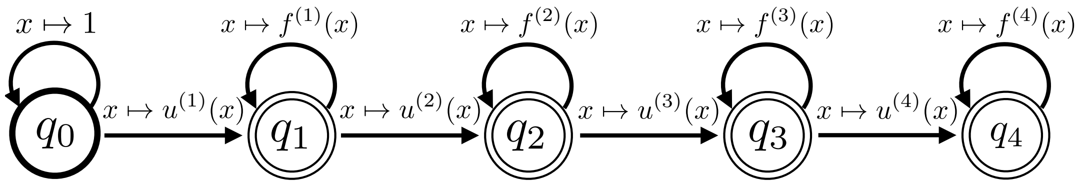


Figure 4.2: A 4-gram WFSA, from which we derive the rational RNN (Section 4.2). The rational RNN's hidden states corresponds to a set of WFSAs, each separately parameterized. We apply group lasso to each WFSA.

are removed entirely (effectively decreasing the RNN hidden dimension). We then extract this sparse structure, corresponding to a version of the same RNN with far fewer parameters. Our sentiment analysis experiments show that these thin RNNs perform on par with the original, parameter-rich RNN.

4.2 Method

We describe the proposed method. At a high level, we follow the standard practice for using ℓ_1 regularization for sparsification (Wen et al., 2016):

1. Fit a model on the training data, with the group lasso regularizer added to the loss during training (the parameters associated with one state comprise one group); see Eq. 4.2.
2. After convergence, eliminate the states whose parameters are zero.
3. Finetune the resulting, smaller model, by minimizing the unregularized loss with respect to its parameters.

In this work, we assume a single layer rational RNN, but our approach is equally applicable to multi-layer models. For clarity of the discussion, we start with a one-dimensional rational RNN (i.e., one based on a single WFSA only). We then generalize to the d-dimensional case (computing the scores of d WFSAs in parallel).

4.2.1 Rational Recurrent Network

Closely following Peng et al. (2018), we parameterize the transition functions of a 5-state WFSA with neural networks (Figure 4.2). A path starts at q_0; at least four tokens must be consumed to reach q_4, and in this sense it captures 4-gram "soft" patterns (Peng et al., 2018; Schwartz et al., 2018). In addition to q_4, we also use q_1, q_2, and q_3 as final states, allowing for the interpolation between patterns of different lengths. We found this to be more stable than using only q_4. The self-loop transitions over q_1, q_2, q_3, and q_4 aim to allow, but downweight, nonconsecutive patterns, as the self-loop transition functions are implemented to output values between 0 and 1 (using a sigmoid function).

The recurrent function is equivalent to applying the Forward dynamic programming algorithm (Baum and Petrie, 1966). To make the discussion self-contained, we concretely walk through the derivation below. Given input string $\mathbf{x} = x_1 \ldots x_n$, let $c_t^{(i)}$ (for any $t \leq n$) denote the sum of scores of all paths ending in state q_i after consuming prefix $x_1 \ldots x_t$. Denoting $u^{(i)}(x_t)$ by $u_t^{(i)}$, and $f^{(i)}(x_t)$ by $f_t^{(i)}$, we have

$$c_t^{(0)} = 1 \tag{4.3a}$$

$$c_t^{(i)} = c_{t-1}^{(i)} \cdot f_t^{(i)} + c_{t-1}^{(i-1)} \cdot u_t^{(i)}, \tag{4.3b}$$

for $i \in \{1, 2, 3, 4\}$.

The functions $f^{(i)}$ (representing self-loops) and $u^{(i)}$ (representing transitions) can be parameterized with neural networks. Letting $\mathbf{z}_t$ denote the embedding vector for token x_t,

$$f_t^{(i)} = \sigma\big(\mathbf{w}^{(i)\top}\mathbf{z}_t\big), \tag{4.4a}$$

$$u_t^{(i)} = (1 - f_t^{(i)}) \cdot \mathbf{v}^{(i)\top}\mathbf{z}_t, \tag{4.4b}$$

where $\mathbf{w}^{(i)}$ and $\mathbf{v}^{(i)}$ vectors are learned parameters. In the interest of notational clarity, we suppress the bias terms in the affine transformations.

c_t, the total score of the prefix string $x_1 \ldots x_t$, is calculated by summing the scores of the paths ending in each of the final states:

$$c_t = \sum_{i=1}^{4} c_t^{(i)}. \tag{4.5}$$

For a document of length n, c_n is then used in downstream computation, e.g., fed into an MLP classifier.

4.2.2 Promoting Sparsity with Group Lasso

We now apply group lasso to promote sparsity in a rational RNN, continuing with the same running example.

From the WFSA perspective, a smaller model is one with fewer states. This can be achieved by penalizing the parameters associated with a given state, specifically the parameters associated with *entering* that state, either by a transition from another state or a self-loop on that state. The parameters of the WFSA diagrammed in Figure 4.2 are assigned to four nonoverlapping groups, excluding the word embedding parameters, $\langle \mathbf{w}^{(i)}, \mathbf{v}^{(i)} \rangle$, for $i \in \{1, 2, 3, 4\}$.

During gradient-based training to solve Eq. 4.2, all parameters will be pushed toward zero, but some will converge very close to zero.There are specialized optimization methods to achieve "strong" sparsity known as proximal gradient descent methods (Parikh and Boyd, 2013), where some parameters are exactly set to zero during training. Recent work has shown these approaches can converge in the nonconvex settings (Reddi et al., 2016), but our experiments found them to be unstable. After convergence, we check the Euclidean norm of each group, and remove those that fall below $\epsilon = 0.1$. This threshold was lightly tuned in preliminary experiments and found to reliably remove those parameters which converged around zero without removing others. Note that, with our linear-structured WFSA, zeroing out the group associated with a state in the middle effectively makes later states inaccessible. While our approach offers no guarantee to remove states from the end first (thus leaving no unreachable states), it always does so in our experiments.

The resulting smaller model, along with its parameters, is then finetuned by continuing training on the data without the regularization term, i.e., setting $\lambda = 0$.

4.2.3 *d*-dimensional Case

The discussion so far centers on a one-dimensional model, i.e., a rational RNN with one WFSA. It is straightforward to construct a d-dimensional model: we stack d one-dimensional models. Each of them is separately parameterized, and recovers a single dimension of the d recurrent computation. Such elementwise recurrent computation is *not* a necessary condition for the model to be rational. We refer the readers to Section 4.3 of Peng et al. (2018) for related discussion.

Elementwise recurrent updates lead to desirable properties when trained with group lasso.

Let us consider a d-dimensional rational model derived from the WFSA in Figure 4.2. Its parameters are organized into $4d$ groups, four for each dimension. Since there is no direct interaction between different dimensions (e.g., through a affine transformation), group lasso sparsifies each dimension/WFSA independently. Hence the resulting rational RNN can consist of WFSAs of different sizes, the number of which could be smaller than d if any of the WFSAs have all states eliminated.

4.2.4 Discussion

One can treat the numbers and sizes of WFSAs as hyperparameters (Oncina et al., 1993; Ron et al., 1994; De la Higuera, 2010; Schwartz et al., 2018, *inter alia*). By eliminating states from WFSAs with group lasso, we learn the WFSA structure *while* estimating the models' parameters, reducing the number of training cycles by reducing the number of hyperparameters that have to be tuned.

4.3 Experiments

To evaluate our approach, we conduct experiments on sentiment analysis. We train the rational RNN models described in Section 4.2 with the group lasso regularizer, using increasingly large regularization strengths, resulting in increasingly compact models. As the goal of our experiments

	Training	**Dev.**	**Test**
kitchen	3,298	822	4,118
dvd	14,066	3,514	17,578
books	20,000	5,000	25,000
original_mix	20,000	5,000	25,000

Table 4.1: Text classification dataset sizes. Each dataset follows the same training/dev./test split ratio as the original mix.

is to demonstrate the ability of our approach for reducing the number of parameters, we only consider rational baselines: the same rational RNNs trained without group lasso. Rational RNNs have shown strong performance on the dataset we experiment with: a 2-layer rational model with between 100–300 hidden units obtained 92.7% classification accuracy, substantially outperforming an LSTM baseline (Peng et al., 2018). The results of our models, which are single-layered and capped at 24 hidden units, are not directly comparable to these baselines, but are still within two points of the best result from that paper. We manually tune the number and sizes of the baselines WFSAs, and then compare the tradeoff curve between model size and accuracy.

Data

We experiment with the Amazon reviews dataset (Blitzer et al., 2007), which is composed of 22 product categories. We examine the standard dataset (**original_mix**) comprised of a mixture of data from the different categories (Johnson and Zhang, 2015).[2] We also examine three of the largest individual categories as separate datasets (**kitchen**, **dvd**, and **books**), which we created following Johnson and Zhang (2015). Note that the three category datasets do *not* overlap with each other (though they do with **original_mix**), and are significantly different in sizes, so we can see how our approach behaves with different amounts of training data. See Table 4.1 for dataset statistics.

Pre-processing

As preprocessing for the data for each individual category, we tokenize using NLTK word tokenizer. We threw out reviews with text shorter than 5 tokens.

We binarize the review score using the standard procedure, assigning 1- and 2-star reviews as negative, and 4- and 5-star reviews as positive (discarding 3-star reviews). Then, if there were more than 25,000 negative reviews, we downsample to 25,000 (otherwise we keep them all), and then downsample the positive reviews to be the same number as negative, to have a balanced dataset. We match the train, development, and test set proportions of 4:1:5 from the original mixture.

[2]`riejohnson.com/cnn_data.html`

We generate the BERT embeddings using the sum of the last four hidden layers of the large uncased BERT model, so our embedding size is 1024. Summing the last four layers was the best performing approach in the ablation of Devlin et al., 2019 that had fewer than 4096 embedding size (which was too large to fit in memory). We embed each sentence individually (there can be multiple sentences within one example).

Implementation details

To classify text, we concatenate the scores computed by each WFSA, then feed this d-dimensional vector of scores into a linear binary classifier. We use log loss. We experiment with both type-level word embeddings (GloVe.6B.300d; Pennington et al., 2014) and contextual embeddings (BERT large; Devlin et al., 2019).[3] For GloVe, we train rational models with 24 5-state WFSAs, each corresponding to a 4-gram soft-pattern (Figure 4.2). For BERT, we train models with 12 WFSAs. BERT embeddings dimension is significantly larger than GloVe (1024 compared to 300), so to manage the increased computational cost we used a smaller number of WFSAs. As our results show, the BERT models still substantially outperform the GloVe ones. In both cases, we keep the embeddings fixed, so the vast majority of the learnable parameters are in the WFSAs.

Baselines

As baselines, we train five versions of the same rational architecture without group lasso, using the same number of WFSAs as our regularized models (24 for GloVe, 12 for BERT). Four of the baselines each use the same number of transitions for all WFSAs (1, 2, 3, and 4, corresponding to 2–5 states, and to 24, 48, 72, and 96 total transitions). The fifth baseline has an equal mix of all lengths (6 WFSAs of each size for GloVe, leading to 60 total transitions, and 3 WFSAs of each size for BERT, leading to 30 total transitions). As each transition is independently parameterized, the total number of transitions linearly controls the number of learnable parameters in the model. Nonetheless, the total number of parameters in the model relies also on the embedding layer, see discussion below.

Experimental setup

We evaluate the GloVe models on all datasets. Due to memory constraints we evaluate BERT only on the smallest dataset (**kitchen**).

For each model (regularized or baseline), we run random search to select our hyperparameters (evaluating 20 uniformly sampled hyperparameter configurations). For the hyperparameter configuration that leads to the best development result, we train the model again 5 times

[3]`https://github.com/huggingface/pytorch-pretrained-BERT`

with different random seeds, and report the mean and standard deviation of the models' test performance. We include hyperparameter search space in Table 4.2.

Type	**Range**
Learning Rate	$[7 * 10^{-3}, 0.5]$
Vertical dropout	[0, 0.5]
Recurrent dropout	[0, 0.5]
Embedding dropout	[0, 0.5]
ℓ_2 regularization	[0, 0.5]
Weight decay	$[10^{-5}, 10^{-7}]$

Table 4.2: Hyperparameter ranges considered in our experiments.

Parameters

The models are trained with Adam (Kingma and Ba, 2015). During training with group lasso we turn off the learning rate schedule (so the learning rate stays fixed), similarly to Gordon et al. (2018). This leads to improved stability in the learned structure for a given hyper-parameter assignment.

Following Peng et al., 2018 we sample 20 hyperparameters uniformly, for which we train and evaluate our models. Hyperparameter ranges are presented in Table 4.2. For the BERT experiments, we reduced both the upper and lower bound on the learning rate by two orders of magnitude.

Regularization Strength Search

We searched for model structures that were regularized down to close to 20, 40, 60, or 80 transitions (10, 20, 30, and 40 for BERT experiments). For a particular goal size, we uniformly sample 20 hyperparameter assignments from the ranges in Table 4.2, then sorted the samples by increasing learning rate. For each hyperparameter assignment, we trained a model with the current regularization strength. If the resulting learned structure was too large (small), we doubled (halved) the regularization strength, repeating until we were within 10 transitions of our goal (5 for BERT experiments). If the regularization strength became larger than 10^2 or smaller than 10^-9, we threw out the hyperparameter assignment and resampled (this happened when e.g. the learning rate was too small for any of the weights to actually make it to zero). Finally, we finetuned the appropriately-sized learned structure by continuing training without the regularizer, and computed the result on the development set. For the best model on the development set, we retrained (first with the regularizer to learn a structure, then finetuned) five times, and plot the mean and variance of the test accuracy and learned structure size.

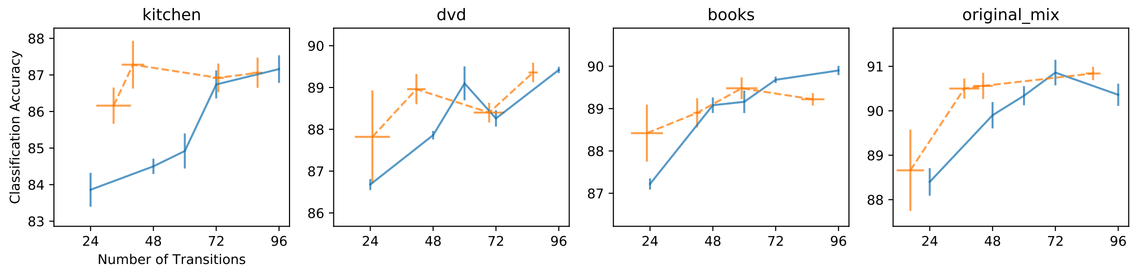


Figure 4.3: Text classification with GloVe embeddings: accuracy (y-axis) vs. number of transitions (x-axis). **Higher** and to the **left** is better. Our method (dashed orange line, varying regularization strength) provides a better tradeoff than the baseline (solid blue line, directly varying the number of transitions). Vertical lines encode one standard deviation for accuracy, while horizontal lines encode one standard deviation in the number of transitions (applicable only to our method).

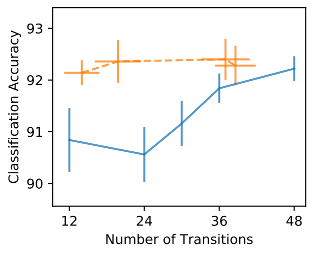


Figure 4.4: Text classification results using BERT embeddings on the **kitchen** dataset.

Regularization Strength Recommendation

If a practitioner wishes to learn a single small model, we recommend they start with λ such that the loss $\mathcal{L}(\mathbf{w})$ and the regularization term are equal. We found that having equal contribution led to eliminating approximately half of the states, though this varies with data set size, learning rate, and gradient clipping, among other variables.

Results

Figure 4.3 shows our test results on all four datasets when trained with GloVe embeddings. The figure shows the classification accuracy as a function of the total number of WFSA transitions in the model. The first thing we notice is that as expected, the performance of our unregularized baselines improves as models are trained with more transitions (i.e., more parameters).

Compared to the baselines, training with group lasso provides a better tradeoff between performance and number of transitions. In particular, our heavily regularized models perform substantially better than the unigram baselines, gaining between 1–2% absolute improvements in three out of four cases. As our regularization strength decreases, we naturally gain less compared to our baselines, although still similar or better than the best baselines in three out of four cases.

Turning to BERT embeddings on the **kitchen** dataset (Figure 4.4), we first see that all accuracies are substantially higher than with GloVe (Figure 4.3, first plot on the left), as expected. We also see the same gains using our method that we observed with GloVe embeddings: training with group lasso leads to smaller models that perform on par with the larger baseline models. In particular, our BERT model with only 14 transitions performs on par with the full baseline model with 3.4 times more transitions (48).

Discussion

One appealing property of our approach is that it can promote the adoption of state-of-the-art NLP models on memory-restricted devices such as smart phones. However, it only reduces the number of learnable RNN parameters, which in our experiments is typically smaller than 100K. The total number of parameters in our models is dominated by the embedding layer (which is untuned); the GloVe 300-dimension embedding with vocabulary size of 400K corresponds to 120M parameters; the BERT-large model uses even more parameters (roughly 340M). Thus, to make our approach applicable to small devices, reducing the number of parameters in the embedding layer is mandatory. We defer this research direction to future work.

4.4 Visualization

Schwartz et al. (2018) introduced a method for interpreting rational RNNs. They computed the score of each WFSA in the model's hidden states on every phrase in the training corpus. Then the top scoring phrases are selected for each WFSA, providing a prototype-like description of the captured pattern representing this WFSA. Their approach allows visualizing individual WFSAs (i.e., an RNN hidden unit). Yet their rational model consists of dozens of different WFSAs, making it indigestible for practitioners to visualize all WFSAs.

Our approach helps to alleviate this problem. By reducing the number of WFSAs, we are able visualize every hidden unit. Indeed, when we use a high regularization strength, the resulting sparse structures often contain only a handful of WFSAs.

Table 4.3 visualizes a sparse rational RNNs trained on **original_mix**. The test performance of this model is 88%, 0.6 absolute below the average of the five models reported in Figure 4.3. It contains only *three* WFSAs, with 8 main-path transitions in total. The table shows the top five scoring phrases for each WFSAs. As each WFSA score is used as a feature fed to a linear classifier (Section 4.3), negative scores are also meaningful. Hence we extract the five bottom scoring phrases as well. While the WFSA scores are the sum of all paths deriving a document (plus-times semiring), here we search for the max (or min) scoring one. Despite the mismatch, a WFSA scores every possible path, and thus the max/min scoring path selection is still valid. As our examples show, many of these extracted paths are meaningful.

The table shows a few interesting trends. First, looking only at the top scores of each WFSA, two of the patterns respectively capture the phrases "*not worth X </s>*" and "*miserable/returned*

		transition$_1$	**transition$_2$**	**transition$_3$**
Patt. 1	Top	not	worth	the time$_{SL}$ </s>
		not	worth	the 30$_{SL}$ </s>
		not	worth	it$_{SL}$ </s>
		not	worth	it$_{SL}$ </s>
		not	worth	it$_{SL}$ </s>
	Bottom	extremely	pleased	. . . $_{SL}$ </s>
		highly	pleased	. . . $_{SL}$ </s>
		extremely	pleased	. . . $_{SL}$ </s>
		extremely	pleased	. . . $_{SL}$ </s>
		extremely	pleased	. . . $_{SL}$ </s>
Patt. 2	Top	bad	. . . $_{SL}$ ltd	. . . $_{SL}$ buyer
		bad	. . . $_{SL}$ ltd	. . . $_{SL}$ buyer
		horrible	. . . $_{SL}$ hl4040cn	. . . $_{SL}$ expensive
		left	. . . $_{SL}$ ltd	. . . $_{SL}$ lens
		terrible	. . . $_{SL}$ labor	. . . $_{SL}$ panasonic
	Bottom	favorite	. . . $_{SL}$ ltd	. . . $_{SL}$ lens
		really	. . . $_{SL}$ ltd	. . . $_{SL}$ buyer
		really	. . . $_{SL}$ ltd	. . . $_{SL}$ buyer
		best	. . . $_{SL}$ hl4040cn	. . . $_{SL}$ expensive
		perfect	tool	. . . $_{SL}$ lens
Patt. 3	Top	miserable	</s>	
		miserable	. . . $_{SL}$ </s>	
		miserable	. . . $_{SL}$ </s>	
		returned	</s>	
		returned	</s>	
	Bottom	superb	</s>	
		superb	</s>	
		superb	</s>	
		superb	choice$_{SL}$ </s>	
		superb	. . . $_{SL}$ </s>	

Table 4.3: Visualization of a sparse rational RNN trained on **original_mix** containing only 3 WFSA. For each WFSA (i.e., pattern), we show the 5 top and bottom scoring phrases in the training corpus with this WFSA. Each column represents one main-path transition, plus potential self-loops preceding it (marked like this$_{SL}$). "... $_{SL}$" marks more than 2 self loops. "</s>" marks an end-of-document token.

X </s>". The third one is less coherent, but most examples *do* contain sentiment-bearing words such as *bad, horrible*, or *best.* This might be the result of the tuning process of the sparse rational structure (Section 4.2) learning to concentrate on earlier transitions, not making use of all the remaining transitions. As a result, this WFSA is treated as a unigram pattern rather than a trigram one.

We observe another interesting trend: two of the three patterns prefer expressions that appear near the end of the document. This could result from the nature of the datasets (e.g., many reviews end with a summary, containing important sentiment information), and/or our rational models' recency preference. More specifically, the first self loop has weight 1, and hence the model is not penalized for taking self loops before the match; in contrast, the weights of the last self loop take values in $(0, 1)$ due to the sigmoid, forcing a penalty for earlier phrase matches. Changing this behavior could be easily done by fixing the final self-loop to 1 as well.

Finally, comparing the top scores vs. the bottom scores, we see that each WFSA is learning (at least) two different patterns: one for phrases with high scores, and the other for ones with low (negative) scores. Still, given the small number of WFSAs used by the model, we are able to visualize all learned (soft) pattern in a single table.

Table 4.4 shows the same visualization for another sparse rational RNN containing only four WFSAs and 11 main-path transitions, trained with BERT embeddings on **kitchen**. It also shows

		transition$_1$	**transition**$_2$	**transition**$_3$
Patt. 1	Top	are	perfect	$\ldots_{SL}$ [CLS]
		definitely	recommend	$\ldots_{SL}$ [CLS]
		excellent	product	$\ldots_{SL}$ [CLS]
		highly	recommend	$\ldots_{SL}$ [CLS]
		great	orange	$\ldots_{SL}$ [CLS]
	Bottom	not	$\ldots_{SL}$ [SEP]	$\ldots_{SL}$ [CLS]
		very	disappointing	!$_{SL}$ [SEP]$_{SL}$ [CLS]
		was	defective	$\ldots_{SL}$ had
		would	not	$\ldots_{SL}$ [CLS]
		terrible	product	.$_{SL}$ [SEP]$_{SL}$ [CLS]
Patt. 2	Top	[CLS]	mine	broke
		[CLS]	it	$\ldots_{SL}$ heat
		[CLS]	thus	it
		[CLS]	it$_{SL}$ does	it$_{SL}$ heat
		[CLS]	$\ldots_{SL}$ is	$\ldots_{SL}$ winter
	Bottom	[CLS]	perfect	$\ldots_{SL}$ cold
		[CLS]	sturdy	$\ldots_{SL}$ cooks
		[CLS]	evenly	,$_{SL}$ withstand$_{SL}$ heat
		[CLS]	it	is
		[CLS]	it	works
Patt. 3	Top	‘	pops	’$_{SL}$ ’$_{SL}$ escape
		‘	gave	out
		that	had	escaped
		‘	non	-
		‘	they$_{SL}$ do	they$_{SL}$ not
	Bottom	simply	does	not
		[CLS]	useless	equipment$_{SL}$!
		unit	would	not
		[CLS]	poor	to$_{SL}$ no
		[CLS]	completely	worthless
Patt. 4	Top	[CLS]	after	
		[CLS]	our	
		mysteriously	jammed	
		mysteriously	jammed	
		[CLS]	we	
	Bottom	[CLS]	i	
		[CLS]	i	
		[CLS]	i	
		[CLS]	we	
		[CLS]	we	

Table 4.4: Visualization of a sparse rational RNN containing 4 WFSAs only, trained on **kitchen** using BERT.

a few clear patterns (e.g., Patt. 2). Interpretation here is more challenging though, as contextual embeddings make every token embedding depend on the entire context. Indeed, contextual embeddings raise problems for interpretation methods that work by targeting individual words, e.g., attention (Bahdanau et al., 2015), as these embeddings also depend on other words. Interpretation methods for contextual embeddings are an exciting direction for future work. A particular example of this is the excessive use of the start token ([CLS]), whose contextual embedding has been shown to capture the sentiment information at the sentence level (Devlin et al., 2019).

4.5 Related Work

Several approaches have been proposed for reducing the size of a neural model. Recent advances in neural architecture search approaches have reduced the computational expense from thousands of GPU hours (Zoph and Le, 2017) to tens of GPU hours (Pham et al., 2018).

DARTS (Liu et al., 2019a) uses differentiable representations of neural architectures, and choose a compression factor before training. Then during training DARTS only keeps the K strongest connections/operations, removing the rest. While scaling K is an easy way of changing the amount of reduction in the model, their neural architecture is not learning to perform well with a subset of the parameters as ours does, it simply removes the smallest parameters.

Our work is most similar to others which use an ℓ_1 regularizer during training to drive parameters to zero (Gordon et al., 2018; Scardapane et al., 2017; Wen et al., 2016, *inter alia*), though these typically just group parameters by layers (or don't group them at all), while we take advantage of the connections between rational RNNs and WFSAs to build our groups. Alternatively, instead of using the magnitude of the parameter itself, some works have estimated the importance of a neuron with the second derivative and pruned those with small values in the Hessian (LeCun et al., 1990; Lee et al., 2019).

Bayesian approaches have been used to promote sparsity by using sparsifying priors (Louizos et al., 2017) or using variational dropout to prune parameters (Molchanov et al., 2017). Stolcke (1994) took a Bayesian approach to learning the structure of FSAs for modeling natural language by using the posterior probability of the model itself to balance goodness-of-fit criteria with model simplicity, thus encouraging simpler models.

Approaches which promote sparsity have a long history throughout machine learning and NLP. Recent approaches include sparseMAP (Niculae et al., 2018), which allows for sparse structured inference, and sparsemax (Martins and Astudillo, 2016), which produces sparse attention mechanisms.

4.6 Conclusion

We presented a method for learning parameter-efficient RNNs. Our method applies group lasso regularization on rational RNNs, which are strongly connected to weighted finite-state automata, and thus amenable to learning with structured sparsity. Our experiments on four text classification datasets, using both GloVe and BERT embeddings, show that our sparse models provide a better performance/model size tradeoff.

The work in this chapter aims to reduce E from the Green AI equation, the cost of processing a single example. Inducing structured sparsity using our technique can lead to models which can process examples faster, with a lower computational requirement.

Chapter 5

Weight Initializations, Data Orders, and Early Stopping

The advent of large-scale self-supervised pretraining has contributed greatly to progress in natural language processing (Devlin et al., 2019; Liu et al., 2019b; Radford et al., 2019). In particular, BERT (Devlin et al., 2019) advanced accuracy on natural language understanding tasks in popular NLP benchmarks such as GLUE (Wang et al., 2019b) and SuperGLUE (Wang et al., 2019a), and variants of this model have since seen adoption in ever-wider applications (Schwartz et al., 2019a; Lu et al., 2019). Typically, these models are first pretrained on large corpora, then fine-tuned on downstream tasks by reusing the model's parameters as a starting point, while adding one task-specific layer trained from scratch. Despite its simplicity and ubiquity in modern NLP, this process has been shown to be brittle (Devlin et al., 2019; Phang et al., 2018; Zhu et al., 2019; Raffel et al., 2019), where fine-tuning performance can vary substantially across different training episodes, even with fixed hyperparameter values.

In this work, we investigate this variation by conducting a series of fine-tuning experiments on four tasks in the GLUE benchmark (Wang et al., 2019b). Changing only training data order and the weight initialization of the fine-tuning layer—which contains only 0.0006% of the total number of parameters in the model—we find substantial variance in performance across trials. This chapter primarily addresses D in the Green AI equation, the data. In addition to providing an analysis of the impact of random seed which controls the data order, Section 5.4 contains training curves and a simple early stopping algorithm which builds on them. In addition, this work shows that the random seed can be an impactful hyperparameter, so also addresses H in the Green AI equation, the amount of hyperparameter tuning. This chapter extends Dodge et al. (2019b); we publicly release our code[1] and data[2].

We explore how validation performance of the best found model varies with the number

[1] `https://github.com/dodgejesse/bert_on_stilts`
[2] `http://www.cs.cmu.edu/~jessed/data_hosting/random_seed_data.zip`

	MRPC	RTE	CoLA	SST
BERT (Phang et al., 2018)	90.7	70.0	62.1	92.5
BERT (Liu et al., 2019b)	88.0	70.4	60.6	93.2
BERT (ours)	**<u>91.4</u>**	**77.3**	**67.6**	**95.1**
STILTs (Phang et al., 2018)	90.9	83.4	62.1	93.2
XLNet (Yang et al., 2019)	89.2	83.8	63.6	95.6
RoBERTa (Liu et al., 2019b)	90.9	86.6	68.0	96.4
ALBERT (Lan et al., 2019)	90.9	<u>89.2</u>	<u>71.4</u>	<u>96.9</u>

Table 5.1: Fine-tuning BERT multiple times while varying only random seeds leads to substantial improvements over previously published validation results with the same model and experimental setup (top rows), on four tasks from the GLUE benchmark. On some tasks, BERT even becomes competitive with more modern models (bottom rows). Best results with standard BERT fine-tuning regime are indicated in bold, best overall results are underscored.

of fine-tuning experiments, finding that, even after hundreds of trials, performance has not fully converged. With the best found performance across all the conducted experiments of fine-tuning BERT, we observe substantial improvements compared to previous published work with the same model (Table 5.1). On MRPC (Dolan and Brockett, 2005), BERT performs better than more recent models such as XLNet (Yang et al., 2019), RoBERTa (Liu et al., 2019b) and ALBERT (Lan et al., 2019). Moreover, on RTE (Wang et al., 2019b) and CoLA (Warstadt et al., 2019), we observe a 7% (absolute) improvement over previous results with the same model. It is worth highlighting that in our experiments only random seeds are changed—never the fine-tuning regime, hyperparameter values, or pretrained weights. These results demonstrate how model comparisons that only take into account reported performance in a benchmark can be misleading, and serve as a reminder of the value of more rigorous reporting practices (Dodge et al., 2019a).

To better understand the high variance across fine-tuning episodes, we separate two factors that affect it: the weight initialization for the task-specific layer; and the training data order resulting from random shuffling. The contributions of each of these have previously been conflated or overlooked, even by works that recognize the importance of multiple trials or random initialization (Phang et al., 2018). By conducting experiments with multiple combinations of random seeds that control each of these factors, we quantify their contribution to the variance across runs. Moreover, we present evidence that some seeds are consistently better than others in a given dataset for both weight initializations and data orders. Surprisingly, we find that some weight initializations perform well across *all studied tasks*.

By frequently evaluating the models through training, we empirically observe that worse performing models can often be distinguished from better ones early in training, motivating investigations of early stopping strategies. We show that a simple early stopping algorithm (described in Section 5.4) is an effective strategy for reducing the computational resources needed to reach a given validation performance and include practical recommendations for a wide range of computational budgets.

To encourage further research in analyzing training dynamics during fine-tuning, we publicly release all of our experimental data. This includes, for each of the 2,100 fine-tuning episodes, the training loss at every weight update, and validation performance on at least 30 points in training.[3]

Our main contributions are:

- We show that running multiple trials with different random seeds can lead to substantial gains in performance on four datasets from the GLUE benchmark. Further, we present how the performance of the best-found model changes as a function of the number of trials.
- We investigate weight initialization and training data order as two sources of randomness in fine-tuning by varying random seeds that control them, finding that 1) they are comparable as sources of variance in performance; 2) in a given dataset, some data orders and weight initializations are consistently better than others; and 3) some weight initializations perform well across multiple different tasks.
- We demonstrate how a simple early stopping algorithm can effectively be used to improve expected performance using a given computational budget.
- We release all of our collected data of 2,100 fine-tuning episodes on four popular datasets from the GLUE benchmark to incentivize further analyses of fine-tuning dynamics.

5.1 Methodology

Our experiments consist of fine-tuning pretrained BERT to four downstream tasks from the GLUE benchmark. For a given task, we experiment multiple times with the same model using the same hyperparameter values, while modifying only the random seeds that control weight initialization (WI) of the final classification layer and training data order (DO). In this section we describe in detail the datasets and settings for our experiments.

5.1.1 Data

We examine four datasets from the GLUE benchmark, described below and summarized in Table 5.2. The data is publicly available and can be download from the repository jiant.[4] Three of our datasets are relatively small (MRPC, RTE, and CoLA), and one relatively large (SST). Since all datasets are framed as binary classification, the model structure for each is the same, as only a single classification layer with two output units is appended to the pretrained BERT.

[3]`http://www.cs.cmu.edu/~jessed/data_hosting/random_seed_data.zip`
[4]`https://github.com/nyu-mll/jiant`

Microsoft Research Paraphrase Corpus (MRPC; Dolan and Brockett, 2005) contains pairs of sentences, labeled as either nearly semantically equivalent, or not. The dataset is evaluated using the average of F_1 and accuracy.

Recognizing Textual Entailment (RTE; Wang et al., 2019b) combines data from a series of datasets (Dagan et al., 2005; Bar-Haim et al., 2006; Giampiccolo et al., 2007; Bentivogli et al., 2009). Each example in RTE is a pair of sentences, and the task is to predict whether the first (the premise) entails the second (the hypothesis).

Corpus of Linguistic Acceptability (CoLA; Warstadt et al., 2019) is comprised of English sentences labeled as either grammatical or ungrammatical. Models are evaluated on Matthews correlation (MCC; Matthews, 1975), which ranges between –1 and 1, with random guessing being 0.

Stanford Sentiment Treebank (SST; Socher et al., 2013) consists of sentences annotated as expressing *positive* or *negative* sentiment (we use the binary version of the annotation), collected from movie reviews.

We describe the number of training samples, the number of validation samples, the majority baseline, and the evaluation metric in Table 5.2.

	MRPC	RTE	CoLA	SST
evaluation metric	Acc./F_1	Acc.	MCC	Acc.
majority baseline	0.75	0.53	0.00	0.51
# training samples	3.7k	2.5k	8.6k	67k
# validation samples	409	277	1,043	873

Table 5.2: The datasets used in this work, which comprise four out of nine of the tasks in the GLUE benchmark (Wang et al., 2019b)

5.1.2 Fine-tuning

Following standard practice, we fine-tune BERT (BERT-large, uncased) for three epochs (Phang et al., 2018; Devlin et al., 2019). We fine-tune the entire model (340 million parameters), of which the vast majority start as pretrained weights and the final layer (2048 parameters) is randomly initialized. The weights in the final classification layer are initialized using the standard approach used when fine-tuning pretrained transformers like BERT, RoBERTa, and ALBERT (Devlin et al., 2019; Liu et al., 2019b; Lan et al., 2019): sampling from a normal distribution with mean 0 and standard deviation 0.02. All experiments were run on P100 GPUs with 16 GB of RAM. We train with a batch size of 16, a learning rate of 0.00002, and dropout of 0.1; the open source implementation, pretrained weights, and full hyperparameter values and experimental details can be found in the HuggingFace transformer library (Wolf et al., 2019).[5]

Each experiment is repeated N^2 times, with all possible combinations of N distinct random seeds for WI and N for DO.[6] For the datasets MRPC, RTE, and CoLA, we run a total of 625 experiments each (N=25). For the larger SST, we run 225 experiments (N=15).

5.2 The large impact of random seeds

Our large set of fine-tuning experiments evidences the sizable variance in performance across trials varying only random seeds. This effect is especially pronounced on the smaller datasets; the validation performance of the best-found model from multiple experiments is substantially higher than the expected performance of a single trial. In particular, in Table 5.1 we report the performance of the best model from all conducted experiments, which represents substantial gains compared to previous work that uses the same model and optimization procedure. On some datasets, we observe numbers competitive with more recent models which have improved pretraining regimes (Phang et al., 2018; Yang et al., 2019; Liu et al., 2019b; Lan et al., 2019); compared to BERT, these approaches pretrain on more data, and some utilize more sophisticated modeling or optimization strategies. We leave it to future work to analyze the variance from random seeds on these other models, and note that running analogous experiments would likely also lead to performance improvements.

In light of these overall gains and the computational burden of running a large number of experiments, we explore how the number of trials influences the expected validation performance.

5.2.1 Expected validation performance

To quantify the improvement found from running more experiments, we turn to expected validation performance as introduced in Chapter2. The standard machine learning experimental setup involves a practitioner training x models, evaluating each of them on validation data, then taking the model which has the best validation performance and evaluating it on test data. Intuitively, as the number of trained models x increases, the best of those x models will improve; expected validation performance calculates the expected value of the best validation performance as a function of x. A full derivation can be found in Section 2.2 of this thesis.

We plot expected validation curves for each dataset in Figure 5.1 with (plus or minus) the standard deviation shaded. We shade between the observed minimum and maximum. The leftmost point on each of these curves ($x = 1$) shows the expected performance for a budget of a single training run. For all datasets, Figure 5.1 shows, unsurprisingly, that expected validation

[5]`https://github.com/huggingface/transformers`

[6]Although any random numbers would have sufficed, for completeness: we use the numbers $\{1, \ldots, N\}$ as seeds.

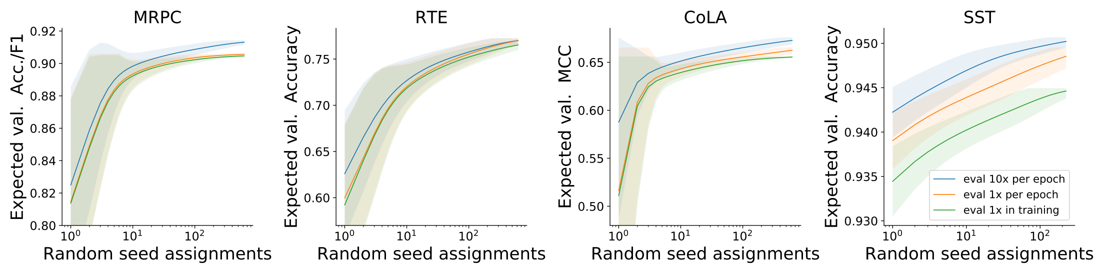


Figure 5.1: Expected validation performance (Dodge et al., 2019a), plus and minus one standard deviation, as the number of experiments increases. The x-axis represents the budget (e.g., $x = 10$ indicates a budget large enough to train 10 models). The y-axis is the expected performance of the best of the x models trained. Each plot shows three evaluation scenarios: in the first, the model is frequently evaluated on the validation set during training (blue); in the second, at the end of each epoch (orange); and in the third, only at the end training (green). As we increase the number of evaluations per run we see higher expected performance and smaller variances. Further, more frequently evaluating the model on validation data leads to higher expected validation values.

performance increases as more computational resources are used. This rising trend continues even up to our largest budget, suggesting even larger budgets could lead to improvements. On the three smaller datasets (MRPC, RTE, and CoLA) there is significant variance at smaller budgets, which indicates that individual runs can have widely varying performance.

In the most common setup for fine-tuning on these datasets, models are evaluated on the validation data after each epoch, or once after training for multiple epochs (Phang et al., 2018; Devlin et al., 2019). In Figure 5.1 we show expected performance as we vary the number of evaluations on validation data during training (all models trained for three epochs): once after training (green), after each of the three epochs (orange), and frequently throughout training (ten times per epoch, blue). Compared to training, evaluation is typically cheap, since the validation set is smaller than the training set and evaluation requires only a forward pass (so batch sizes can be much larger). Moreover, evaluating on the validation data can be done in parallel to training, and thus does not necessarily slow down training. Considering the benefits of more frequent evaluations as shown in Figure 5.1, we thus recommend this practice in similar scenarios.

5.3 Weight initialization and data order

To better understand the high variance in performance across trials, we analyze two source of randomness: the weight initialization of the final classification layer and the order the training data is presented to the model. While previous work on fine-tuning pretrained contextual representation models (Devlin et al., 2019; Phang et al., 2018) has generally used a single random seed to control these two factors, we analyze them separately.

Our experiments are conducted with every combination of a set of weight initialization seeds

	MRPC	RTE	CoLA	SST
Agg. over WI	.058	.066	.090	.0028
Agg. over DO	.059	.067	.095	.0024
Total	.061	.069	.101	.0028

Table 5.3: Expected (average) standard deviation in validation performance across runs. The expected standard deviation of given WI and DO random seeds are close in magnitude, and only slightly below the overall standard deviation.

(WI) and a set of data order (DO) seeds that control these factors. One data order can be viewed as one sample from the set of permutations of the training data. Similarly, one weight initialization can be viewed as a specific set of samples from the normal distribution from which we draw them.

An overview of the collected data is presented in Figure 5.2, where each colored cell represents the validation performance for a single experiment. In the plots, each row represents a single weight initialization and each column represents a single data order. We sort the rows and columns by their averages; the top row contains experiments with the WI with the highest average performance, and the rightmost column contains experiments with the DO with the highest average performance. Each cell represents an independent sample, so the rows and columns can be reordered.

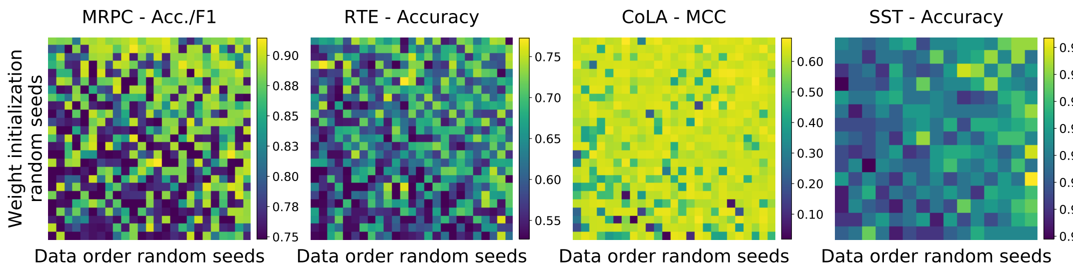


Figure 5.2: A visualization of validation performance for all experiments, where each colored cell represents the performance of a training run with a specific WI and DO seed. Rows and columns are sorted by their average, such that the best WI seed corresponds to the top row of each plot, and the best DO seed correspond to the right-most column. Especially on smaller datasets a large variance in performance is observed across different seed combinations, and on MRPC and RTE models frequently diverge, performing close to the majority baselines (listed in Table 5.2).

For MRPC, RTE, and CoLA, a fraction of the trained models diverge, yielding performance close to that of predicting the most frequent label (see Table 5.2). This partially explains the large variance found in the expected validation curves for those three datasets in Figure 5.1.

5.3.1 Decoupling

From Figure 5.2, it is clear that different random seed combinations can lead to substantially different validation performance. In this section, we investigate the sources of this variance, decoupling the distribution of performance based on each of the factors that control randomness.

For each dataset, we compute for each WI and each DO seed the standard deviation in validation performance across all trials with that seed. We then compute the expected (average) standard deviation, aggregated under all WI or all DO seeds, which are shown in Table 5.3. Although their magnitudes vary significantly between the datasets, the expected standard deviation from the WI and DO seeds is comparable, and are slightly below the overall standard deviation inside a given task.

We plot the distribution of standard deviations in final validation performance across multiple runs, aggregated under a fixed random seed, separately for weight initializations and data orders. The results are shown in Figure 5.3, indicating that the inter-seed aggregated variances are comparable in magnitude, considering aggregation over both WI and DO.

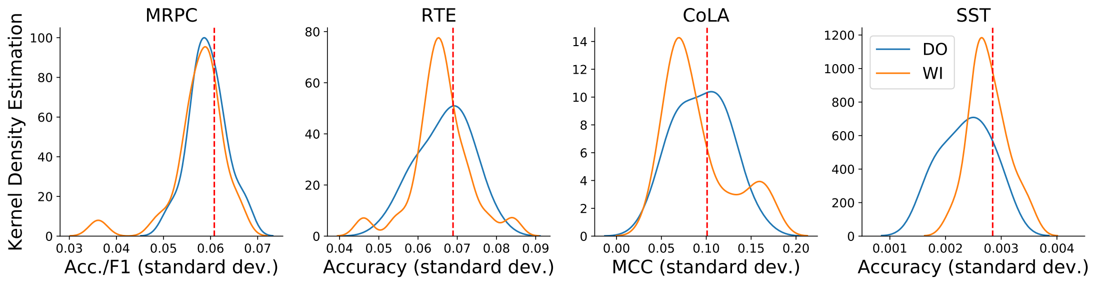


Figure 5.3: Kernel density estimation of the distribution of standard deviation in validation performance aggregated under fixed random seeds, either for weight initialization (blue) or data order (orange). The red dashed line shows the overall standard deviation for each dataset. The DO and WI curves have expected standard deviation values of similar magnitude, which are also comparable with the overall standard deviation.

5.3.2 Some random seeds are better than others

To investigate whether some WI or DO seeds are better than their counterparts, Figure 5.4 plots the random seeds with the best and worst average performance. The best and worst seeds exhibit quite different behavior: compared to the best, the worst seeds have an appreciably higher density on lower performance ranges, indicating that they are generally inferior. On MRPC, RTE, and CoLA the performance of the best and worst WIs are more dissimilar than the best and worst DOs, while on SST the opposite is true. This could be related to the size of the data; MRPC, RTE, and CoLA are smaller datasets, whereas SST is larger, so SST has more data to order and more weight updates to move away from the initialization.

Using ANOVA (Fisher, 1935) to test for statistical significance, we examine whether the performance of the best and worst DOs and WIs have distributions with different means. The results are shown in Table 5.4. For all datasets, we find the best and worst DOs and WIs are significantly different in their expected performance ($p < 0.05$).

ANOVA makes three assumptions: 1) independence of the samples, 2) homoscedasticity (roughly equal variance across groups), and 3) normally distributed data.

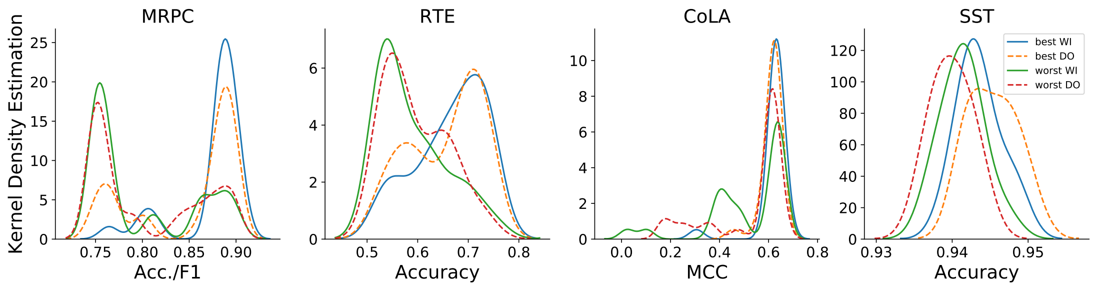


Figure 5.4: Some seeds are better then others. Plots show the kernel density estimation of the distribution of validation performance for best and worst WI and DO seeds. Curves for DO seeds are shown in dashed lines and for WI in solid lines. MRPC and RTE exhibit pronounced bimodal shapes, where one of the modes represents divergence; models trained with the worst WI and DO are more likely to diverge than learn to predict better than random guessing. Compared to the best seeds, the worst seeds are conspicuously more densely populated in the lower performing regions, for all datasets.

	MRPC	RTE	CoLA	SST
WI	$2.0{\times}10^{-6}$	$2.8{\times}10^{-4}$	$7.0{\times}10^{-3}$	$3.3{\times}10^{-2}$
DO	$8.3{\times}10^{-3}$	$3.2{\times}10^{-3}$	$1.1{\times}10^{-2}$	$1.3{\times}10^{-5}$

Table 5.4: p-values from ANOVA indicate that there is evidence to reject the null hypothesis that the performance of the best and worst WIs and DOs have distributions with the same means ($p < 0.05$).

ANOVA is not robust to violations of independence, but each DO and WI is an I.I.D. sample, and thus independent. ANOVA is generally robust to groups with somewhat differing variance if the groups are the same size, which is true in our experiments. ANOVA is more robust to non-normally distributed data for larger sample sizes; our SST experiments are quite close to normally distributed, and the distribution of performance on the smaller datasets is less like a normal distribution but we have larger sample sizes.

5.3.3 Globally good initializations

A natural question that follows is whether some random seeds are good *across datasets*. While the data order is dataset specific, the same weight initialization can be applied to multiple classifiers trained with different datasets: since all tasks studied are binary classification, models for all datasets share the same architecture, including the classification layer that is being randomly initialized and learned.

We compare the different weight initializations across datasets. We find that some initializations perform consistently well. For instance, WI seed 12 has the best performance on CoLA and RTE, the second best on MRPC, and third best on SST. This suggests that, perhaps surprisingly, some weight initializations perform well across tasks.

Studying the properties of good weight initializations and data orders is an important question that could lead to significant empirical gains and enhanced understanding of the fine-tuning process. We defer this question to future work, and release the results of our 2,100 fine-tuning experiments to facilitate further study of this question by the community.

5.4 Early stopping

Our analysis so far indicates a high variance in the fine-tuning performance of BERT when using different random seeds, where some models fail to converge. This was also observed by Phang et al. (2018), who showed that their proposed STILTs approach reduced the number of diverging models. In this section we show that better performance can be achieved with the same computational resources by using early stopping algorithms that stop the least promising trials early in training. We also include recommendations for practitioners for setting up experiments meeting a variety of computational budgets.

Early discovery of failed experiments

Figure 5.5 shows that performance divergence can often be recognized early in training. These plots show the performance values of 20 randomly chosen models at different times across training. In many of the curves, continuing training of lower performing models all the way through can be a waste of computation. In turn, this suggests the potential of early stopping least promising trials as a viable means of saving computation without large decreases in expected performance. For instance, after training halfway through the first epoch on CoLA the models which diverged could be stopped.

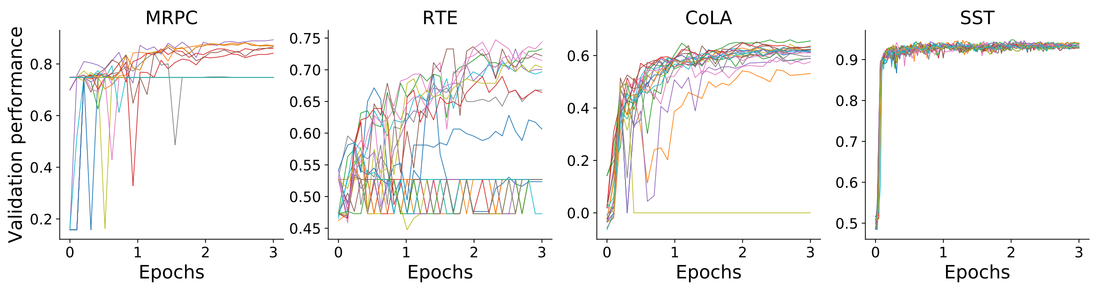


Figure 5.5: Some promising seeds can be distinguished early in training. The plots show training curves for 20 random WI and DO combinations for each dataset. Models are evaluated every 10th of an epoch (except SST, which was evaluated every 100 steps, equivalent to 42 times per epoch). For the smaller datasets, training is unstable, and a non-negligible portion of the models yields poor performance, which can be identified early on.

Correlation between points in training

We examine the correlation of validation performances at different points throughout training, shown in Figure 5.6. One point in one of these plots represents the Spearman's rank correlation between performance at iteration i and iteration j across trials. High rank correlation means that the ranking of the models is similar between the two evaluation points, and suggests we can stop the worst performing models early, as they would likely continue to underperform.

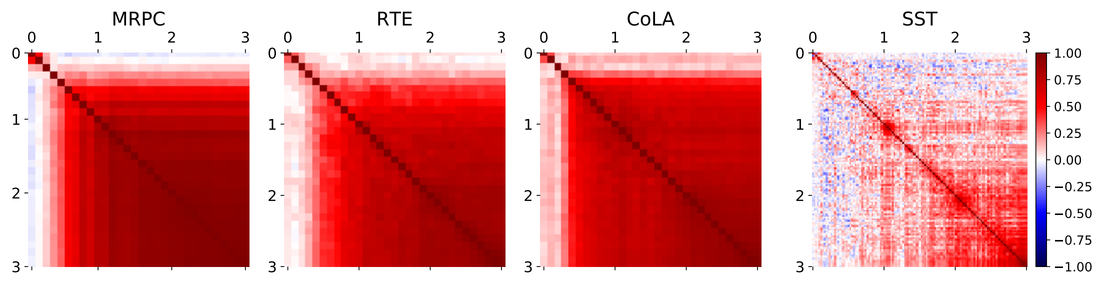


Figure 5.6: The rank of the models early in training is highly correlated with the rank late in training. Each figure shows the Spearman's rank correlation between the validation performance at different points in training; the axes represent epochs. A point at coordinates i and j in the plots indicates the correlation between the ranking of the models (ranked by validation performance) after i and after j evaluations. Note that the plots are symmetric.

On MRPC, RTE and CoLA, there exists a high correlation between the models' performance early on (part way through the first epoch) and their final performance. On the larger SST dataset, we see high correlation between the performance after training for two epochs and the final performance.

In Figure 5.7 we include the Pearson correlation between different points in training, the same data used for Figure 5.6. This captures relative magnitude of the differences in performance, instead of just the ranking of the models. One point in one of these plots represents the Pearson's correlation between performance at iteration i and iteration j across trials. Here we see a similar story, corroborating the results found in Figure 5.6. High correlation means that the performance of the models is similar between the two evaluation points.

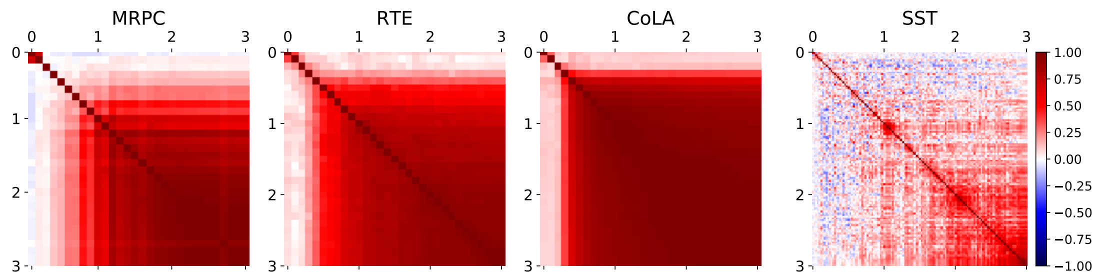


Figure 5.7: Performance early in training is highly correlated with performance late in training. Each figure shows the Pearson correlation between the validation performance at different points in training; the axes represent epochs. A point at coordinates i and j in the plots indicates the correlation between the best found performances after i and after j evaluations. Note that the plots are symmetric.

Early stopping

Considering the evidence from the training curves and correlation plots, we analyze a simple algorithm for early stopping. Our algorithm is inspired by existing approaches to making hyperparameter search more efficient by stopping some of the least promising experiments early

(Jamieson and Talwalkar, 2016; Li et al., 2018). “Early stopping” can also relate to stopping a single training run if the loss hasn’t decreased for a given number of epochs. Here we refer to the notion of stopping a subset of multiple trials. Here we apply an early stopping algorithm to select the best performing random seed. Our approach does not distinguish between DO and WI. While initial results suggest that this distinction could inspire more sophisticated early-stopping criteria, we defer this to future work. The algorithm has three parameters: t, f, and p. We start by training t trials, and partially through training (f, a fraction of the total number of epochs) evaluate all of them and only continue to fully train the p most promising ones, while discarding the rest. This algorithm takes a total of $(tf + p(1 - f))s$ steps, where s is the number of steps to fully train a model. In our experiments $s = 3$ epochs.

Start many, stop early, continue some

As shown earlier, the computational budget of running this algorithm can be computed directly from an assignment to the parameters t, f, and p. Note that there are different ways to assign these parameters that lead to the same computational budget, and those can lead to significantly distinct performance in expectation; to estimate the performance for each configuration we simulate this algorithm by sampling 50,000 times from from our full set of experiments. In Figure 5.8 we show the best observed assignment of these parameters for budgets between 3 and 90 total epochs of training, or the equivalent of 1 to 30 complete training trials. There are some surprisingly consistent trends across datasets and budgets – the number of trials started should be significantly higher than the number trained fully, and the number of trials to train fully should be around $x/2$. On three out of four datasets, stopping least promising trials after 20–30% of training (less than one epoch) yielded the best results—and on the fourth dataset this is still a strong strategy.

Early stopping works

We compare this algorithm with our baseline of running multiple experiments all the way through training, without any early stopping (f=1, t=p) and using the same amount of computation. Specifically, for a given computational budget equivalent to fully training t models, we measure improvement as the relative error reduction from using early stopping with the best found settings for that computational budget. Figure 5.9 shows the relative error reduction for each dataset as the computational budget varies, where we observe small but reasonably consistent improvements on all tasks.

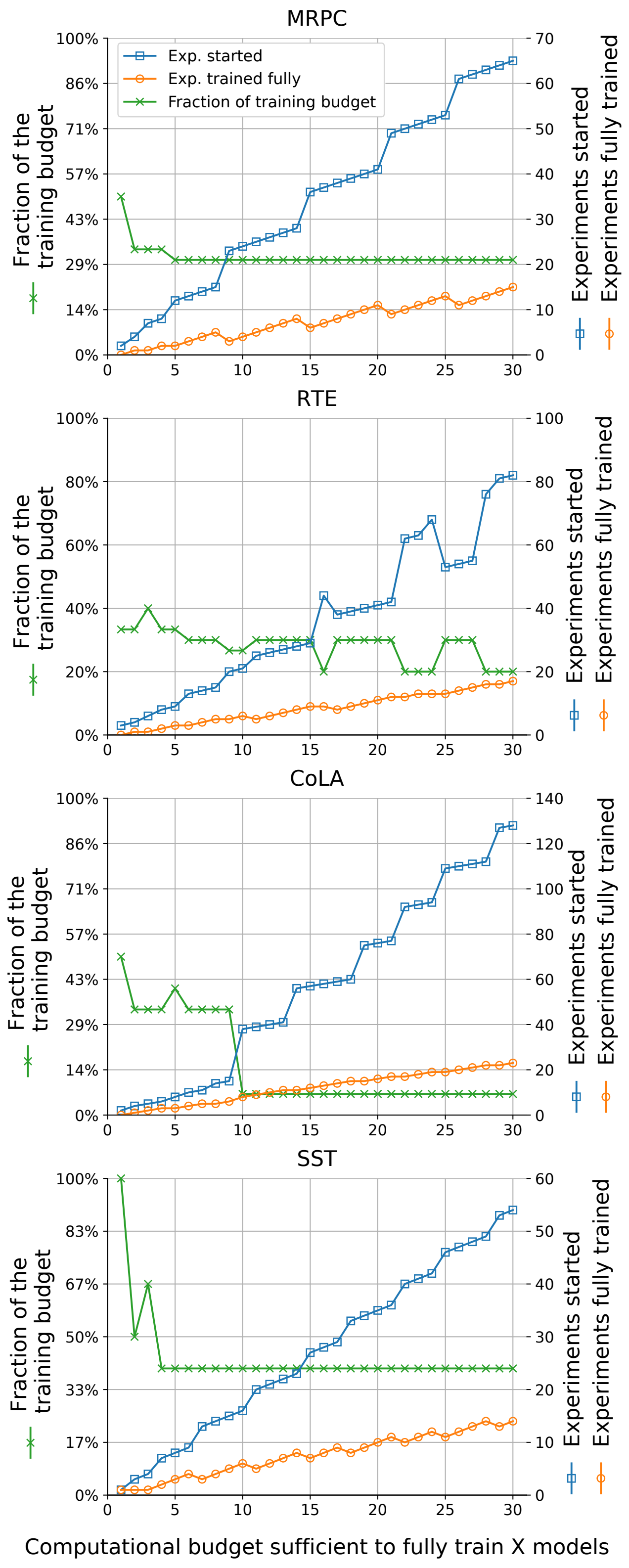


Figure 5.8: Best observed early stopping parameters on each dataset. For a given budget large enough to fully train x models (each trained for 3 epochs), this plot shows the optimal parameters for early stopping. For instance, in MRPC with a budget large enough for 20 trials, the best observed performance came by starting 41 trials (blue), then continuing only the 11 most promising trials (orange) after 30% of training (green).

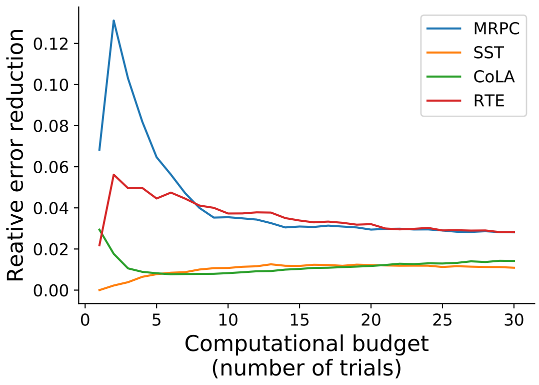


Figure 5.9: Relative error reduction from the early stopping approach in Figure 5.8, compared to the baseline of training x models on the full training budget. Performance on RTE and SST is measured using accuracy, on MRPC it is the average of accuracy and F1, and on CoLA it is MCC. "Error" here refers to one-minus-performance for each of these datasets. As the budget increases, the absolute performance on all four datasets increases, and the absolute improvement from early stopping is fairly consistent.

5.5 Related work

Most work on hyperparameter optimization tunes a number of impactful hyperparameters, such as the learning rate, the width of the layers in the model, and the strength of the regularization (Li et al., 2018; Bergstra et al., 2011). For modern machine learning models such tuning has proven to have a large impact on the performance; in this work we only examine two oft-overlooked choices that can be cast as hyperparameters and still find room for optimization.

Melis et al. (2018) heavily tuned the hyperpamareters of an LSTM language model, for some experiments running 1,500 rounds of Bayesian optimization (thus, training 1,500 models). They showed that an LSTM, when given such a large budget for hyperparameter tuning, can outperform more complicated neural models. While such work informs the community about the best performance found after expending very large budgets, it is difficult for future researchers to build on this without some measure of how the performance changes as a function of computational budget. Our work similarly presents the best-found performance using a large budget (Table 5.1), but also includes estimates of how performance changes as a function of budget (Figure 5.1).

A line of research has addressed the distribution from which initializations are drawn. The Xavier initialization (Glorot and Bengio, 2010) and Kaiming initialization (He et al., 2015) initialize weights by sampling from a uniform distribution or normal distribution with variance scaled so as to preserve gradient magnitudes through backpropagation. Similarly, orthogonal initializations (Saxe et al., 2014) aim to prevent exploding or vanishing gradients. In our work, we instead examine how different samples from an initialization distribution behave, and we hope future work which introduces new initialization schemes will provide a similar analysis.

Active learning techniques, which choose a data order using a criterion such as the model's uncertainty (Lewis and Gale, 1994), have a rich history. Recently, it has even been shown that that training on mini-batches which are diverse in terms of data or labels (Zhang et al., 2017) can be more sample efficient. The tools we present here can be used to evaluate different seeds for a stochastic active learning algorithm, or to compare different active learning algorithms.

5.6 Conclusion

This chapter focuses on improving D in the Green AI equation, data efficiency. We study the impact of random seeds on fine-tuning contextual embedding models, the currently dominant paradigm in NLP. We conduct a large set of experiments on four datasets from the GLUE benchmark and observe significant variance across these trials. Overall, these experiments lead to substantial performance gains on all tasks. By observing how the expected performance changes as we allocate more computational resources, we expect that further gains would come from an even larger set of trials. Moreover, we examine the two sources of variance across trials, weight initialization and training data order, finding that in expectation, they contribute comparably to the variance in performance. Perhaps surprisingly, we find that some data orders and initializations are better than others, and the latter can even be observed even across tasks. A simple early stopping strategy along with practical recommendations is included to alleviate the computational costs of running multiple trials. All of our experimental data containing thousands of fine-tuning episodes is publicly released.

Chapter 6

Conclusion and Future Work

6.1 Summary of contributions

As AI and its subfields of machine learning, natural language processing, and computer vision grow, we are observing a natural stratification of research by the size of the computational budget used for its supporting experiments. Strong community norms exist for comparing models which were trained on the same data, and the increased performance (with diminishing returns) observed with more training data is well-documented, so there is some guidance for models trained with varying amounts of data. While it is known that increasing model complexity and the total number of experiments run (for architecture search or other hyperparameter optimization) can lead to performance improvements, similar community-wide norms don't exist here, so new research is not held to a standard of accurately reporting the budget let alone making comparisons against similar-cost work. Drawing reproducible conclusions about which approach performs best is challenging when all relevant information is available, and, as we have shown, all but impossible when it is not.

The research in this thesis exists as examples of how to measure and improve performance and efficiency in three key areas: E, the computational expense to process a single example, D, the total number of examples processed, and H, the number of experiments. In Chapter 2 we introduce a method for reporting the results of a full hyperparameter search instead of reporting the results of the single best-found model. In Chapter 3 we introduce a novel hyperparameter optimization method which is more efficient than competing approaches. In Chapter 4 we introduce techniques for inducing structured sparsity in parameter-rich neural models which can lead to more efficient inference. In Chapter 5 we analyze training curves, and show the efficacy of early stopping even when consistent training procedures produce highly inconsistent results. The experiments in these chapters are primarily on language data, though the motivation and conclusions drawn from the results are applicable more generally in machine learning.

Producing research which exemplifies the ideals of fair model comparisons conditioned on the computational budget is, unfortunately, not enough to elicit community-wide change. Towards the broader goal of improving standards for new research this thesis also includes recommendations to the community for how to draw reproducible conclusions when comparing approaches against each other. These recommendations are centered around improved reporting of the budget and experimental information, especially that which is atypically included in published work. In addition, the applications in this thesis were selected to showcase how such improved reporting can lead to better performance-efficiency tradeoffs, and how the lack of reporting of details for even small changes such as varying the random seed can strongly hinder reproducibility.

This thesis is predicated on the idea that as the landscape of research changes, so to must the ways in which we evaluate our work. The nature of research is that new methods and applications are regularly introduced, and so our standards must continue to evolve. Even such norms as comparing models trained on the same data need to adapt; state-of-the-art results in natural language processing and computer vision are now driven by pretraining models on large unlabeled corpora then fine-tuning on labeled data. In this regime, the amount of pretraining data is not held to the same strict standards as the amount labeled data, so we see comparisons of models with wildly differing budgets. This example is one of the ample opportunities for future work outlined by the high-level themes in this thesis of efficiency and reproducibility, and in the next section highlights several more.

6.2 Future directions

6.2.1 E, Efficient Models

Structured sparsity for BERT

A natural extension of the work in Chapter 4 is to apply a similar technique to a more complicated class of models, such as transformers. Using structured sparsity would allow us to learn which parts of the model to remove during training, which will lead to more efficient models which can be put into production at scale, or run on resource-constrained devices. In addition, by examining which components are most important (i.e. not removed by the sparsifyer, even at high levels of sparisty) we can inform the development of new models.

It's possible that the learned structures will be different for different tasks, in which case we have evidence that the inductive bias in models is somewhat modular. It's also possible that the learned structures are task-agnostic, suggesting that we could then develop more general-purpose neural architectures.

Small models which can be act as prototypes for large models

Progress in research has been driven partly by increasingly large models being trained on massive amounts of data. An open question is whether or not we could use small versions of these massive models for fast prototyping, and have the results of that prototyping generalize to the large-budget setting. Is there some transformation that can be done on a set of large models (such as uniformly reducing the size of all of the weight matricies or reducing the depth of the models) which would lead to a set of models that a) use a small amount of computation for training and inference, and b) have the same *ranking*, in terms of performance, as the large models? If BERT-base, RoBERTa-base, and ELECTRA-small have the same ranking as BERT-large, RoBERTa-large, and ELECTRA-large, this would enable new research to compare against the small versions of these large models. On the other hand, if the ranking is different, this would be a surprising result.

6.2.2 D, Sample Efficiency

Better Data Orders for Pretraining BERT

As we show in Chapter 5, the order of the data can have a large impact on performance. Understanding why some random data orders lead to better performance is still an open question. It's likely that the diversity of the examples has a role to play here, where examples that are diverse in terms of labels or features lead to improved stability and performance, especially near the start of training. To promote diversity, we can use DPPs, as introduced in Chapter 3.

How little training is necessary to compare models?

Randomly initialized, untrained models have been shown to be surprisingly good feature extractors (Ramanujan et al., 2020). It is conjectured that the performance of untrained models reveals how appropriate the inductive bias within the neural architecture is for a given task. If we benchmark different neural architectures by training only the final classification layer (leaving the rest untrained), can we recreate the ranking we would find after fully training the models? Saxe et al. (2011) showed this was possible with simple, fully-connected networks; it may also be true with modern, expensive models. If so, this can be used for fast architecture search, only training the final layer to get a set of promising architectures, which could then be trained fully.

6.2.3 H, Efficient Hyperparameter Tuning

When to stop hyperparameter search

When sampling hyperparameter assignments uniformly at random, when should we stop? If we have evaluated n uniformly sampled hyperparameter assignments, with validation performance

$V_1, \dots, V_n$, and a max of $V_n^* = \max_{i \in \{1,\dots,n\}} V_i$, how much improvement should we expect to see after evaluating an additional m assignments, $V_{n+m}^* - V_n^*$? Is V_n^* close to the global max? Here we can turn to the literature on Extreme Value Theory (de Haan, 2005; Coles, 2001). The Fisher–Tippett–Gnedenko theorem states *if* a normalized maximum converges to a non-degenerate distribution, *then* the in the limit the distribution it converges to is in the Gumbel, Fréchet, or Weibull family.

Extreme Value Theory is used in many places, such as building bridges that can withstand a once-in-n-years flood, or estimating the effects of climate change on annual global temperatures. Maximum likelihood techniques are the most commonly used to fit the distributions, with samples found by either peak-over-threshold method or definiting blocks and using the max from each block.

New initialization method for pretrained models

Much attention has been paid to randomly initializing all of the learnable parameters in a neural model, but as we show in Chapter 5, randomly initializing just the classification layer of a pretrained model can have a large impact on performance. Many initialization schemes are designed so as to preserve the magnitude of gradients passed through each layer in a neural network (He et al., 2015). This idea translates to initializing the final layer of a pretrained model — keeping the pretrained weights fixed, and initializing the final layer so it preserves the gradients passed through the pretrained parts of the model could lead to more stable training, and better performance.

Bibliography

M. O. Ahmed, B. Shahriari, and M. Schmidt. Do we need "harmless" Bayesian optimization and "first-order" Bayesian optimization? In *Proc. of Advances in Neural Information Processing Systems (NeurIPS) Bayesian Optimization Workshop*, 2016.

D. Amodei and D. Hernandez. AI and compute, 2018. URL `https://openai.com/blog/ai-and-compute/`. Blog post.

N. Anari, S. O. Gharan, and A. Rezaei. Monte Carlo Markov chain algorithms for sampling strongly Rayleigh distributions and determinantal point processes. In *Proc. of Conference on Learning Theory (COLT)*, 2016.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*, 2015.

R. Bar-Haim, I. Dagan, B. Dolan, L. Ferro, D. Giampiccolo, B. Magnini, and I. Szpektor. The second pascal recognising textual entailment challenge. In *Proc. of the II PASCAL challenge*, 2006.

R. Bardenet and A. Hardy. Monte Carlo with determinantal point processes. In *arXiv:1605.00361*, 2016.

L. E. Baum and T. Petrie. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, 37(6), 1966. ISSN 0003-4851. doi: 10.1214/aoms/1177699147.

L. Bentivogli, P. Clark, I. Dagan, and D. Giampiccolo. The fifth pascal recognizing textual entailment challenge. In *TAC*, 2009.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. In *Proc. of Journal of Machine Learning Research (JMLR)*, 2012.

J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proc. of NeurIPS*, 2011.

J. Berstel, Jr. and C. Reutenauer. *Rational Series and Their Languages.* Springer-Verlag, Berlin, Heidelberg, 1988.

J. Blitzer, M. Dredze, and F. Pereira. Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proc. of ACL*, 2007. URL `http://aclweb.org/anthology/P07-1056`.

O. Bousquet, S. Gelly, K. Kurach, O. Teytaud, and D. Vincent. Critical hyper-parameters: No random, no cry. *arXiv preprint arXiv:1706.03200*, 2017.

J. Bradbury, S. Merity, C. Xiong, and R. Socher. Quasi-recurrent neural network. In *Proc. of ICLR*, 2017.

E. Candes and T. Tao. Decoding by linear programming. In *IEEE Transactions on Information Theory*, 2005.

Q. Chen, X.-D. Zhu, Z.-H. Ling, S. Wei, H. Jiang, and D. Inkpen. Enhanced LSTM for natural language inference. In *Proc. of ACL*, 2017. URL `https://arxiv.org/abs/1609.06038`.

K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proc. of EMNLP*, 2014. URL `http://www.aclweb.org/anthology/D14-1179`.

S. Coles. An introduction to statistical modeling of extreme values. *Springer Series in Statistics*, 2001.

E. Contal, D. Buffoni, A. Robicquet, and N. Vayatis. Parallel Gaussian process optimization with upper confidence bound and pure exploration. In *Proc. of Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013.

I. Dagan, O. Glickman, and B. Magnini. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, 2005.

L. de Haan. Extreme value theory. *Springer Series in Operations Research*, 2005.

C. De la Higuera. *Grammatical inference: learning automata and grammars.* Cambridge University Press, 2010.

T. Desautels, A. Krause, and J. W. Burdick. Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization. In *Proc. of Journal of Machine Learning Research (JMLR)*, 2014.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. of NAACL*, 2019. URL `https://arxiv.org/abs/1810.04805`.

J. Dodge, K. Jamieson, and N. A. Smith. Open loop hyperparameter optimization and determinantal point processes. In *Proc. of AutoML*, 2017.

J. Dodge, S. Gururangan, D. Card, R. Schwartz, and N. A. Smith. Show your work: Improved reporting of experimental results. In *Proc. of EMNLP*, 2019a.

J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. A. Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *ArXiv*, 2019b.

J. Dodge, R. Schwartz, H. Peng, and N. A. Smith. Rnn architecture learning with sparse regularization. In *Proc. of EMNLP*, 2019c.

B. Dolan and C. Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proc. of IWP*, 2005.

B. Doshi-Velez, Finale; Kim. Towards a rigorous science of interpretable machine learning, 2017. arXiv:1702.08608.

R. Dror, G. Baumer, M. Bogomolov, and R. Reichart. Replicability analysis for natural language processing: Testing significance with multiple datasets. *TACL*, 5:471–486, 2017. doi: 10.1162/tacl_a_00074. URL `https://www.aclweb.org/anthology/Q17-1033`.

B. Efron and R. Tibshirani. *An Introduction to the Bootstrap.* CRC Press, 1994.

R. A. Fisher. Statistical methods for research workers. *Oliver & Boyd (Edinburgh)*, 1935.

J. N. Foerster, J. Gilmer, J. Chorowski, J. Sohl-Dickstein, and D. Sussillo. Intelligible language modeling with input switched affine networks. In *Proc. of ICML*, 2017.

M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. E. Peters, M. Schmitz, and L. S. Zettlemoyer. AllenNLP: A deep semantic natural language processing platform. In *Proc. of NLP-OSS*, 2018. URL `https://arxiv.org/abs/1803.07640`.

T. Gebru, J. H. Morgenstern, B. Vecchione, J. W. Vaughan, H. M. Wallach, H. Daumé, and K. Crawford. Datasheets for datasets, 2018. URL `https://arxiv.org/abs/1803.09010`. arXiv:1803.09010.

A. Gelman and E. Loken. The statistical crisis in science. *American Scientist*, 102:460, 11 2014. URL `https://doi.org/10.1511/2014.111.460`.

D. Giampiccolo, B. Magnini, I. Dagan, and B. Dolan. The third pascal recognizing textual entailment challenge. In *Proc. of the ACL-PASCAL workshop on textual entailment and paraphrasing*, 2007.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. of AISTATS*, 2010.

J. González, Z. Dai, P. Hennig, and N. Lawrence. Batch Bayesian optimization via local penalization. In *Proc. of Artificial Intelligence and Statistics (AISTATS)*, 2016.

A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. MorphNet: Fast & simple resource-constrained structure learning of deep networks. In *Proc. of CVPR*, 2018.

K. Gorman and S. Bedrick. We need to talk about standard splits. In *Proc. of ACL*, 2019. URL `https://www.aclweb.org/anthology/P19-1267/`.

O. E. Gundersen and S. Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Proc. of AAAI*, 2018. URL `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/viewFile/17248/15864`.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proc. of ICCV*, 2015.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proc. of AAAI*, 2018. URL `https://arxiv.org/abs/1709.06560`.

P. Hennig and R. Garnett. Exact sampling from determinantal point processes. In *arxiv:1609.06840*, 2016.

E. Hlawka. Funktionen von beschrankter variation in der theorie der gleichverteilung. In *Ann. Math. Pura Appl.*, 1961.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8), 1997. doi: 10.1162/neco.1997.9.8.1735.

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017. arXiv:1704.04861.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Exact sampling from determinantal point processes. In *Proc. of the Learning and Intelligent OptimizatioN Conference (LION) 6*, 2012.

M. R. Iacò. Low discrepancy sequences: Theory and applications. In *arXiv:1502.04897*, 2015.

J. P. A. Ioannidis. Why most published research findings are false. *PLoS Med*, 2(8), 08 2005. URL `https://doi.org/10.1371/journal.pmed.0020124`.

K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Proc. of AISTATS*, 2016.

R. Johnson and T. Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proc. of NAACL*, 2015. doi: 10.3115/v1/N15-1011.

R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *Proc. of ICML*, 2015. URL `http://www.jmlr.org/proceedings/papers/v37/jozefowicz15.pdf`.

K. Kandasamy, A. Krishnamurthy, J. Schneider, and B. Poczos. Parallelised bayesian optimisation via thompson sampling. In *Proc. of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

T. Kathuria, A. Deshpande, and P. Kohli. Batched Gaussian process bandit optimization via determinantal point processes. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

T. Khot, A. Sabharwal, and P. Clark. SciTaiL: A textual entailment dataset from science question answering. In *Proc. of AAAI*, 2018. URL `http://ai2-website.s3.amazonaws.com/publications/scitail-aaai-2018_cameraready.pdf`.

Y. Kim. Convolutional neural networks for sentence classification. In *Proc. of Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. of ICLR*, 2015.

B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: automatic hyperparameter configuration for scikit-learn. In *Proc. of International Conference on Machine Learning (ICML) workshop on AutoML*, 2014.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. of NeurIPS*, 2012.

W. Kuich and A. Salomaa, editors. *Semirings, Automata, Languages.* Springer-Verlag, 1986.

A. Kulesza, B. Taskar, et al. Determinantal point processes for machine learning. In *Proc. of Foundations and Trends® in Machine Learning*, 2012.

Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv:1909.11942*, 2019.

Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Proc. of NeurIPS*, 1990.

N. Lee, T. Ajanthan, and P. H. Torr. SNIP: Single-shot network pruning based on connection sensitivity. In *Proc. of ICLR*, 2019.

T. Lei, Y. Zhang, and Y. Artzi. Training RNNs as fast as CNNs, 2017. arXiv:1709.02755.

D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *Proc. of SIGIR*, 1994.

L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. In *Proc. of UAI*, 2019. URL `https://arxiv.org/abs/1902.07638`.

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research (JMLR)*, 2018.

R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. In *Proc. of the ICML Workshop on AutoML*, 2018. URL `https://arxiv.org/abs/1807.05118`.

Z. C. Lipton and J. Steinhardt. Troubling trends in machine learning scholarship, 2018. URL `https://arxiv.org/abs/1807.03341`. arXiv:1807.03341.

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *Proc. of ICLR*, 2019a.

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv:1907.11692*, 2019b.

R. Livni, S. Shalev-Shwartz, and O. Shamir. On the computational efficiency of training neural networks. In *Proc. of NeurIPS*, 2014.

C. Louizos, K. Ullrich, and M. Welling. Bayesian compression for deep learning. In *Proc. of NeurIPS*, 2017.

J. Lu, D. Batra, D. Parikh, and S. Lee. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In *Proc. of NeurIPS*, 2019.

M. Lucic, K. Kurach, M. Michalski, O. Bousquet, and S. Gelly. Are GANs created equal? A large-scale study. In *Proc. of NeurIPS*, 2018. URL `https://arxiv.org/abs/1711.10337`.

D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten. Exploring the limits of weakly supervised pretraining, 2018. URL `https://arxiv.org/abs/1805.00932`. arXiv:1805.00932.

A. F. T. Martins and R. Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *Proc. of ICML*, 2016.

A. F. T. Martins, N. A. Smith, M. Figueiredo, and P. Aguiar. Structured sparsity in structured prediction. In *Proc. of EMNLP*, 2011. URL `http://aclweb.org/anthology/D11-1139`.

B. W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 1975.

B. McCann, J. Bradbury, C. Xiong, and R. Socher. Learned in translation: Contextualized word vectors. In *Proc. of NeurIPS*, 2017. URL `https://arxiv.org/abs/1708.00107`.

G. Melis, C. Dyer, and P. Blunsom. On the state of the art of evaluation in neural language models. In *Proc. of EMNLP*, 2018.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2013.

M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru. Model cards for model reporting. In *Proc. of FAT**, 2019. URL `https://arxiv.org/abs/1810.03993`.

D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. In *Proc. of ICML*, 2017.

G. E. Moore. Cramming more components onto integrated circuits, 1965.

V. Niculae, A. F. T. Martins, M. Blondel, and C. Cardie. SparseMAP: Differentiable sparse structured inference. In *Proc. of ICML*, 2018.

H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*, volume 63. Siam, 1992.

J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15:448–458, 1993.

A. P. Parikh, O. Täckström, D. Das, and J. Uszkoreit. A decomposable attention model for natural language inference. In *Proc. of EMNLP*, 2016. URL `https://arxiv.org/abs/1606.01933`.

N. Parikh and S. P. Boyd. *Proximal Algorithms.* Foundations and Trends in Optimization, 2013.

H. Peng, R. Schwartz, S. Thomson, and N. A. Smith. Rational recurrences. In *Proc. of EMNLP*, 2018. URL `http://aclweb.org/anthology/D18-1152`.

J. Pennington, R. Socher, and C. Manning. GloVe: Global vectors for word representation. In *Proc. of EMNLP*, 2014.

M. Peters, S. Ruder, and N. A. Smith. To tune or not to tune? Adapting pretrained representations to diverse tasks. In *Proc. of the RepL4NLP Workshop at ACL*, 2019. URL `https://arxiv.org/abs/1903.05987`.

M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. S. Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. In *Proc. of ICML*, 2018.

J. Phang, T. Févry, and S. R. Bowman. Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks. *arXiv:1811.01088*, 2018.

J. Pineau. Machine learning reproducibility checklist. `https://www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist.pdf`, 2019. Accessed: 2019-5-14.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2019. arXiv:1910.10683.

P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proc. of EMNLP*, 2016. doi: 10.18653/v1/D16-1264.

V. Ramanujan, M. Wortsman, A. Kembhavi, A. Farhadi, and M. Rastegari. What's hidden in a randomly weighted neural network? In *Proc. of CVPR*, 2020.

B. Recht, R. Roelofs, L. Schmidt, and V. Shankar. Do CIFAR-10 classifiers generalize to CIFAR-10?, 2019a. URL `https://arxiv.org/abs/1806.00451`. arXiv:1806.00451.

B. Recht, R. Roelofs, L. Schmidt, and V. Shankar. Do ImageNet classifiers generalize to ImageNet? In *Proc. of ICML*, 2019b. URL `https://arxiv.org/abs/1902.10811`.

S. J. Reddi, S. Sra, B. Póczos, and A. J. Smola. Proximal stochastic methods for nonsmooth nonconvex finite-sum optimization. In *Proc. of NeurIPS*, 2016.

N. Reimers and I. Gurevych. Reporting score distributions makes a difference: Performance study of LSTM-networks for sequence tagging. In *Proc. of EMNLP*, 2017. URL `https://arxiv.org/abs/1707.09861`.

A. Rogers. How the transformers broke NLP leaderboards. `https://hackingsemantics.xyz/2019/leaderboards/`, 2019. Accessed: 2019-8-29.

D. Ron, Y. Singer, and N. Tishby. Learning probabilistic automata with variable memory length. In *Proc. of COLT*, 1994.

C. Rosset. Turing-NLG: A 17-billion-parameter language model by microsoft. *Microsoft Blog*, 2019.

A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Y. Ng. On random weights and unsupervised feature learning. In *Proc. of ICML*, 2011.

A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *Proc. of ICLR*, 2014.

S. Scardapane, D. Comminiello, A. Hussain, and A. Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241, 2017.

D. Schwartz, M. Toneva, and L. Wehbe. Inducing brain-relevant bias in natural language processing models. In *Proc. of NeurIPS*, 2019a.

R. Schwartz, O. Abend, R. Reichart, and A. Rappoport. Neutralizing linguistically problematic annotations in unsupervised dependency parsing evaluation. In *Proc. of ACL*, 2011. URL `http://www.aclweb.org/anthology/P11-1067`.

R. Schwartz, S. Thomson, and N. A. Smith. SoPa: Bridging CNNs, RNNs, and weighted finite-state machines. In *Proc. of ACL*, 2018. URL `http://aclweb.org/anthology/P18-1028`.

R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni. Green AI, 2019b. URL `https://arxiv.org/abs/1907.10597`. arXiv:1907.10597.

D. Sculley, J. Snoek, A. Rahimi, and A. Wiltschko. Winner's curse? On pace, progress, and empirical rigor. In *Proc. of ICLR (Workshop Track)*, 2018. URL `https://openreview.net/references/pdf?id=HyT0zqkwG`.

M. J. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. In *Proc. of ICLR*, 2017. URL `https://arxiv.org/abs/1611.01603`.

S. Shalev-Shwartz and S. Ben-David. Understanding machine learning: From theory to algorithms. In *Proc. of Cambridge University Press*, 2014.

M. Shoeybi, , M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using GPU model parallelism, 2019. arXiv:1909.08053.

Y. Shoham, R. Perrault, E. Brynjolfsson, J. Clark, J. Manyika, J. C. Niebles, T. Lyons, J. Etchemendy, and Z. Bauer. The AI index 2018 annual report. *AI Index Steering Committee, Human-Centered AI Initiative, Stanford University. Available at `http://cdn.aiindex.org/2018/AI\%20Index\%202018\%20Annual\%20Report.pdf`*, 202018, 2018.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis.

Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017a. arXiv:1712.01815.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676): 354, 2017b.

J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. In *Proc. of Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 1967.

R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of EMNLP*, 2013.

A. Stolcke. *Bayesian learning of probabilistic language models.* PhD thesis, University of California, Berkeley, 1994.

E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in NLP. In *Proc. of ACL*, 2019.

C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proc. of ICCV*, 2017.

K. Swersky, J. Snoek, and R. P. Adams. Freeze-thaw Bayesian optimization. In *arXiv:1406.3896*, 2014.

M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proc. of ICML*, 2019. URL `https://arxiv.org/pdf/1905.11946v3.pdf`.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1), 1996.

A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proc. of ICLR*, 2018. URL `https://arxiv.org/abs/1804.07461`.

A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Proc. of NeuRIPS*, 2019a.

A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proc. of ICLR*, 2019b.

Z. Wang, C. Li, S. Jegelka, and P. Kohli. Batched high-dimensional bayesian optimization via structural kernel learning. In *Proc. of International Conference on Machine Learning (ICML)*, 2017.

A. Warstadt, A. Singh, and S. R. Bowman. Neural network acceptability judgments. *TACL*, 7: 625–641, 2019.

W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Proc. of NeurIPS*, 2016.

T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Proc. of NeuRIPS*, 2019.

D. Yogatama and N. A. Smith. Bayesian optimization of text representations. In *Proc. of EMNLP*, 2015. URL `https://arxiv.org/abs/1503.00693`.

L. Yuan, J. Liu, and J. Ye. Efficient methods for overlapping group lasso. In *Proc. of NeurIPS*, 2011.

M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1), 2006.

R. Zellers, A. Holtzman, H. Rashkin, Y. Bisk, A. Farhadi, F. Roesner, and Y. Choi. Defending against neural fake news, 2019. arXiv:1905.12616.

C. Zhang, H. Kjellström, and S. Mandt. Determinantal point processes for mini-batch diversification. In *Proc. of UAI*, 2017.

Y. Zhang and B. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification, 2015. URL `https://arxiv.org/abs/1510.03820`. arXiv:1510.03820.

A. Zhigljavsky and A. Zilinskas. *Stochastic global optimization*, volume 9. Springer Science & Business Media, 2007.

C. Zhu, Y. Cheng, Z. Gan, S. Sun, T. Goldstein, and J. Liu. Freelb: Enhanced adversarial training for language understanding. *arXiv:1909.11764*, 2019.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *Proc of ICLR*, 2017.

Appendix A

A.1 Hyperparameter Search Spaces for Section 2.3.2

Computing infrastructure	GeForce GTX 1080 GPU
Number of search trials	50
Search strategy	uniform sampling
Best validation accuracy	40.5
Training duration	39 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	50	50
patience	10	10
batch size	64	64
embedding	GloVe (50 dim)	GloVe (50 dim)
encoder	ConvNet	ConvNet
max filter size	*uniform-integer*[3, 6]	4
number of filters	*uniform-integer*[64, 512]	332
dropout	*uniform-float*[0, 0.5]	0.4
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	2 epochs	2 epochs
learning rate scheduler reduction factor	0.5	0.5
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.0008

Table A.1: SST (fine-grained) CNN classifier search space and best assignments.

Computing Infrastructure	3.1 GHz Intel Core i7 CPU
Number of search trials	50
Search strategy	uniform sampling
Best validation accuracy	39.8
Training duration	1.56 seconds
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
penalty	*choice*[L1, L2]	L2
no. of iter	100	100
solver	liblinear	liblinear
regularization	*uniform-float*[0, 1]	0.13
n-grams	*choice*[(1, 2), (1, 2, 3), (2, 3)]	[1, 2]
stopwords	*choice*[True, False]	True
weight	*choice*[tf, tf-idf, binary]	binary
tolerance	*loguniform-float*[10e-5, 10e-3]	0.00014

Table A.2: SST (fine-grained) logistic regression search space and best assignments.

A.2 Hyperparameter Search Spaces for Section 2.3.3

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	50
Search strategy	uniform sampling
Best validation accuracy	87.6
Training duration	1624 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	50	50
patience	10	10
batch size	64	64
gradient norm	*uniform-float*[5, 10]	9.0
embedding dropout	*uniform-float*[0, 0.5]	0.3
number of pre-encode feedforward layers	*choice*[1, 2, 3]	3
number of pre-encode feedforward hidden dims	*uniform-integer*[64, 512]	232
pre-encode feedforward activation	*choice*[relu, tanh]	tanh
pre-encode feedforward dropout	*uniform-float*[0, 0.5]	0.0
encoder hidden size	*uniform-integer*[64, 512]	424
number of encoder layers	*choice*[1, 2, 3]	2
integrator hidden size	*uniform-integer*[64, 512]	337
number of integrator layers	*choice*[1, 2, 3]	3
integrator dropout	*uniform-float*[0, 0.5]	0.1
number of output layers	*choice*[1, 2, 3]	3
output hidden size	*uniform-integer*[64, 512]	384
output dropout	*uniform-float*[0, 0.5]	0.2
output pool sizes	*uniform-integer*[3, 7]	6
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.0001
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	2 epochs	2 epochs
learning rate scheduler reduction factor	0.5	0.5

Table A.3: SST (binary) BCN GloVe search space and best assignments.

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	50
Search strategy	uniform sampling
Best validation accuracy	91.4
Training duration	6815 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	50	50
patience	10	10
batch size	64	64
gradient norm	*uniform-float*[5, 10]	9.0
freeze ELMo	True	True
embedding dropout	*uniform-float*[0, 0.5]	0.3
number of pre-encode feedforward layers	*choice*[1, 2, 3]	3
number of pre-encode feedforward hidden dims	*uniform-integer*[64, 512]	206
pre-encode feedforward activation	*choice*[relu, tanh]	relu
pre-encode feedforward dropout	*uniform-float*[0, 0.5]	0.3
encoder hidden size	*uniform-integer*[64, 512]	93
number of encoder layers	*choice*[1, 2, 3]	1
integrator hidden size	*uniform-integer*[64, 512]	159
number of integrator layers	*choice*[1, 2, 3]	3
integrator dropout	*uniform-float*[0, 0.5]	0.4
number of output layers	*choice*[1, 2, 3]	1
output hidden size	*uniform-integer*[64, 512]	399
output dropout	*uniform-float*[0, 0.5]	0.4
output pool sizes	*uniform-integer*[3, 7]	6
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.0008
use integrator output ELMo	*choice*[True, False]	True
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	2 epochs	2 epochs
learning rate scheduler reduction factor	0.5	0.5

Table A.4: SST (binary) BCN GLoVe + ELMo (frozen) search space and best assignments.

Computing Infrastructure	NVIDIA Titan Xp GPU
Number of search trials	50
Search strategy	uniform sampling
Best validation accuracy	92.2
Training duration	16071 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	50	50
patience	10	10
batch size	64	64
gradient norm	*uniform-float*[5, 10]	7.0
freeze ELMo	False	False
embedding dropout	*uniform-float*[0, 0.5]	0.1
number of pre-encode feedforward layers	*choice*[1, 2, 3]	3
number of pre-encode feedforward hidden dims	*uniform-integer*[64, 512]	285
pre-encode feedforward activation	*choice*[relu, tanh]	relu
pre-encode feedforward dropout	*uniform-float*[0, 0.5]	0.3
encoder hidden size	*uniform-integer*[64, 512]	368
number of encoder layers	*choice*[1, 2, 3]	2
integrator hidden size	*uniform-integer*[64, 512]	475
number of integrator layers	*choice*[1, 2, 3]	3
integrator dropout	*uniform-float*[0, 0.5]	0.4
number of output layers	*choice*[1, 2, 3]	3
output hidden size	*uniform-integer*[64, 512]	362
output dropout	*uniform-float*[0, 0.5]	0.4
output pool sizes	*uniform-integer*[3, 7]	5
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	2.1e-5
use integrator output ELMo	*choice*[True, False]	True
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	2 epochs	2 epochs
learning rate scheduler reduction factor	0.5	0.5

Table A.5: SST (binary) BCN GloVe + ELMo (fine-tuned) search space and best assignments.

A.3 Hyperparameter Search Spaces for Section 2.3.4

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	100
Search strategy	uniform sampling
Best validation accuracy	82.7
Training duration	339 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	140	140
patience	20	20
batch size	64	64
gradient clip	*uniform-float*[5, 10]	5.28
embedding projection dim	*uniform-integer*[64, 300]	78
number of attend feedforward layers	*choice*[1, 2, 3]	1
attend feedforward hidden dims	*uniform-integer*[64, 512]	336
attend feedforward activation	*choice*[relu, tanh]	tanh
attend feedforward dropout	*uniform-float*[0, 0.5]	0.1
number of compare feedforward layers	*choice*[1, 2, 3]	1
compare feedforward hidden dims	*uniform-integer*[64, 512]	370
compare feedforward activation	*choice*[relu, tanh]	relu
compare feedforward dropout	*uniform-float*[0, 0.5]	0.2
number of aggregate feedforward layers	*choice*[1, 2, 3]	2
aggregate feedforward hidden dims	*uniform-integer*[64, 512]	370
aggregate feedforward activation	*choice*[relu, tanh]	relu
aggregate feedforward dropout	*uniform-float*[0, 0.5]	0.1
learning rate optimizer	Adagrad	Adagrad
learning rate	*loguniform-float*[1e-6, 1e-1]	0.009

Table A.6: SciTail DAM search space and best assignments.

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	100
Search strategy	uniform sampling
Best validation accuracy	82.8
Training duration	372 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	75	75
patience	5	5
batch size	64	64
encoder hidden size	*uniform-integer*[64, 512]	253
dropout	*uniform-float*[0, 0.5]	0.28
number of encoder layers	*choice*[1, 2, 3]	1
number of projection feedforward layers	*choice*[1, 2, 3]	2
projection feedforward hidden dims	*uniform-integer*[64, 512]	85
projection feedforward activation	*choice*[relu, tanh]	relu
number of inference encoder layers	*choice*[1, 2, 3]	1
number of output feedforward layers	*choice*[1, 2, 3]	2
output feedforward hidden dims	*uniform-integer*[64, 512]	432
output feedforward activation	*choice*[relu, tanh]	tanh
output feedforward dropout	*uniform-float*[0, 0.5]	0.03
gradient norm	*uniform-float*[5, 10]	7.9
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.0004
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	0 epochs	0 epochs
learning rate scheduler reduction factor	0.5	0.5
learning rate scheduler mode	max	max

Table A.7: SciTail ESIM search space and best assignments.

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	100
Search strategy	uniform sampling
Best validation accuracy	81.2
Training duration	137 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	140	140
patience	20	20
batch size	64	64
dropout	*uniform-float*[0, 0.5]	0.2
hidden size	*uniform-integer*[64, 512]	167
activation	*choice*[relu, tanh]	tanh
number of layers	*choice*[1, 2, 3]	3
gradient norm	*uniform-float*[5, 10]	6.8
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.01
learning rate scheduler	exponential	exponential
learning rate scheduler gamma	0.5	0.5

Table A.8: SciTail n-gram baseline search space and best assignments.

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	100
Search strategy	uniform sampling
Best validation accuracy	81.2
Training duration	1015 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	140	140
patience	20	20
batch size	16	16
embedding projection dim	*uniform-integer*[64, 300]	100
edge embedding size	*uniform-integer*[64, 512]	204
premise encoder hidden size	*uniform-integer*[64, 512]	234
number of premise encoder layers	*choice*[1, 2, 3]	2
premise encoder is bidirectional	*choice*[True, False]	True
number of phrase probability layers	*choice*[1, 2, 3]	2
phrase probability hidden dims	*uniform-integer*[64, 512]	268
phrase probability dropout	*uniform-float*[0, 0.5]	0.2
phrase probability activation	*choice*[tanh, relu]	tanh
number of edge probability layers	*choice*[1, 2, 3]	1
edge probability dropout	*uniform-float*[0, 0.5]	0.2
edge probability activation	*choice*[tanh, relu]	tanh
gradient norm	*uniform-float*[5, 10]	7.0
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.0006
learning rate scheduler	exponential	exponential
learning rate scheduler gamma	0.5	0.5

Table A.9: SciTail DGEM search space and best assignments.

Computing Infrastructure	GeForce GTX 1080 GPU
Number of search trials	128
Search strategy	uniform sampling
Best validation EM	68.2
Training duration	31617 sec
Model implementation	`http://github.com/allenai/show-your-work`

Hyperparameter	**Search space**	**Best assignment**
number of epochs	20	20
patience	10	10
batch size	16	16
token embedding	GloVe (100 dim)	GloVe (100 dim)
gradient norm	*uniform-float*[5, 10]	6.5
dropout	*uniform-float*[0, 0.5]	0.46
character embedding dim	*uniform-integer*[16, 64]	43
max character filter size	*uniform-integer*[3, 6]	3
number of character filters	*uniform-integer*[64, 512]	33
character embedding dropout	*uniform-float*[0, 0.5]	0.15
number of highway layers	*choice*[1, 2, 3]	3
phrase layer hidden size	*uniform-integer*[64, 512]	122
number of phrase layers	*choice*[1, 2, 3]	1
phrase layer dropout	*uniform-float*[0, 0.5]	0.46
modeling layer hidden size	*uniform-integer*[64, 512]	423
number of modeling layers	*choice*[1, 2, 3]	3
modeling layer dropout	*uniform-float*[0, 0.5]	0.32
span end encoder hidden size	*uniform-integer*[64, 512]	138
span end encoder number of layers	*choice*[1, 2, 3]	1
span end encoder dropout	*uniform-float*[0, 0.5]	0.03
learning rate optimizer	Adam	Adam
learning rate	*loguniform-float*[1e-6, 1e-1]	0.00056
Adam β_1	*uniform-float*[0.9, 1.0]	0.95
Adam β_2	*uniform-float*[0.9, 1.0]	0.93
learning rate scheduler	reduce on plateau	reduce on plateau
learning rate scheduler patience	2 epochs	2 epochs
learning rate scheduler reduction factor	0.5	0.5
learning rate scheduler mode	max	max

Table A.10: SQuAD BiDAF search space and best assignments.