UNIVERSITY OF CALIFORNIA,
IRVINE

On Hyperparameter Optimization for Deep Learning

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Statistics

by

Lars Hertel

Dissertation Committee:
Professor Pierre Baldi, Chair
Professor Daniel L. Gillen
Professor Michele Guindani

2020

DEDICATION

To my parents, Stephanie, and Penelope.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to gratefully acknowledge funding from NIH/NIA grant number 5R01AG053555.

Furthermore I would like to acknowledge my co-authors for Chapter 3 Jianming Bian, Lingge Li, as well as my co-authors for Chapter 4 Julian Collado, Peter Sadowski, and Jordan Ott.

I would like to thank Jianming Bian for a collaboration that has allowed me to gain experience in applying machine learning in the real world and has been an important part of my PhD. I also want to thank the NOvA reconstruction group which has been very welcoming and supportive throughout.

I would like to thank Dan Gillen for the invaluable help he provided in the writing and development of Chapter 5 and Chapter 6.

Finally, I would like to thank my advisor for his mentorship and support throughout the program.

VITA

Lars Hertel

EDUCATION

Doctor of Philosophy in Statistics **2020**
University of California, Irvine *Irvine, California*

Masters of Science in Statistics **2014**
Imperial College London *London, United Kingdom*

Bachelor of Science in Physics **2012**
Queen Mary University of London *London, United Kingdom*

RESEARCH EXPERIENCE

Graduate Student Researcher **2019–2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2014–2019**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Improved energy reconstruction in NOvA with regression convolutional neural networks **2019**
Physical Review D

REFEREED CONFERENCE PUBLICATIONS

Sherpa: hyperparameter optimization for machine learning models **Dec 2018**
NIPS 2018 MLOSS Workshop

Approximate inference for deep latent gaussian mixtures **Dec 2016**
NIPS Bayesian Deep Learning Workshop

SOFTWARE

Sherpa `http://github.com/sherpa-ai/sherpa`
Hyperparameter optimization that enables researchers to experiment, visualize, and scale quickly.

ABSTRACT OF THE DISSERTATION

On Hyperparameter Optimization for Deep Learning

By

Lars Hertel

Doctor of Philosophy in Statistics

University of California, Irvine, 2020

Professor Pierre Baldi, Chair

Deep learning has recently achieved many breakthroughs. Neural networks - the models behind deep learning - have a large number of hyperparameters whose correct settings are crucial to obtain optimal performance. As the need for hyperparameter optimization has grown, much research has been produced on this topic. Existing hyperparameter optimization methods often assume a noiseless objective function. In this thesis we explore the effect of noise in the objective function, that is the neural network training, on the hyperparameter optimization. The thesis begins by motivating hyperparameter search through an applied machine learning problem in the domain of neutrino physics. With the help of hyperparameter optimization we develop an energy estimator for observations from the NOvA experiment at Fermilab. This energy estimator is able to outperform prior methods in terms of prediction accuracy by using a deep neural network to directly predict the target energy from the detector response. We then introduce hyperparameter optimization software, *Sherpa*, that has been developed as part of this thesis. After that we focus on methods for the selection of optimal hyperparameter settings when observations are noisy. This is solved through a group sequential testing framework that results in an equivalence class of hyperparameter configurations. The method is empirically validated on three machine learning tasks and is shown to return the optimal hyperparameter setting at a higher rate than choosing the best observed. It furthermore increases reproducibility by reducing variance in the outcome of the

search. Lastly, we focus on finding optimal trade-offs between repeated evaluation of hyperparameter settings and exploration of the space. In particular, we benchmark a number of popular hyperparameter search methods on machine learning tasks with high noise between runs. Empirical results show that standard Gaussian process based Bayesian optimization without repetition tends to deliver competitive results at even small computational budgets in high noise hyperparameter optimization. The thesis concludes with suggestions for future work.

Chapter 1

Introduction

Machine learning and particularly deep learning has demonstrated many recent successes in the domains of supervised learning, unsupervised learning, and reinforcement learning. *Deep learning* refers to methods based on neural networks with multiple hidden layers. Deep neural networks come with a number of hyperparameters - parameters that are not learned as part of the training procedure but that are chosen by the user. Optimization of these hyperparameters is crucial to obtain optimal performance. Tuning of hyperparameters typically involves picking different settings, training a model with each setting, evaluating one's models on the target objective (for example, out-of-sample prediction error), and choosing the setting with the best objective value. A large body of research has been dedicated to the task of picking good candidate settings. Much work has also been done on speeding up the optimization by probing hyperparameter settings before fully evaluating them. In this thesis we explore an aspect of hyperparameter optimization that has so far received less attention: hyperparameter optimization when observations are noisy.

In many situations training of the machine learning algorithm can involve stochasticity. This can range from random weight initialization in neural networks, to feature selection

in random forests, to training on dynamic and random environments as in reinforcement learning. If the noise introduced by this stochasticity is significant, regular hyperparameter optimization methods break down in two ways. Firstly, given a set of hyperparameter configurations and their associated performance, it may be non-trivial to identify which one is best on average. Secondly, the hyperparameter optimization algorithms which suggest candidate settings may rely on observations of hyperparameter settings and their corresponding objective values and may thus be affected by noise in these observations.

We begin the thesis by providing background on the field of hyperparameter optimization in Section 2.1. Section 2.2 goes on to provide an introduction to group sequential testing as applied in Chapter 5. In Chapter 3 we illustrate our first contribution, an application of deep convolutional neural networks to neutrino physics. This application serves as a motivating example for the importance of hyperparameter optimization. Specifically, we reconstruct the energy of electron neutrinos and electrons in the NOvA experiment. The developed method learns to predict neutrino and electron energies based on raw detector outputs and is able to outperform prior energy reconstruction in terms of reconstruction error. For hyperparameter optimization in this work we use a Python hyperparameter optimization software Sherpa which we have developed as part of this thesis. Sherpa - described in Chapter 4 - is a hyperparameter optimization library for machine learning models. It is specifically designed for problems with computationally expensive, iterative function evaluations, such as the hyperparameter tuning of deep neural networks. Its source code and documentation are available at `https://github.com/sherpa-ai/sherpa`. With the ability to run hyperparameter optimization algorithms at large computational scales we study noisy hyperparameter optimization in the following chapters.

Motivated by our work in Chapters 3 and 4, Chapter 5 focuses on how given a pool of candidate settings and their performance, one should choose a hyperparameter setting in the presence of stochasticity given a limited number of observations. In the solution of this

problem we particularly concentrate on how this issue relates to the reproducibility of the optimization. Reproducibility has been a key issue in machine learning over recent years. We illustrate how regular hyperparameter search methods can lead to a large variance in outcomes due to non-deterministic model training during the hyperparameter optimization. To remedy this issue we define the average error across model training runs as the objective for the hyperparameter search and show that common methods perform poorly on this metric. We then propose a hypothesis testing procedure that takes variation within hyperparameter settings into account while controlling family-wise type one error rates across the parameter search space. We further embed this procedure into a group sequential testing framework to increase efficiency in terms of the average number of training replicates required. Empirical results on machine learning benchmarks show that the proposed method reduces the variation in hyperparameter search outcomes by up to 90 percent. Moreover, the sequential testing framework successfully reduces computation while preserving performance of the method.

In Chapter 6, we consider how existing hyperparameter optimization algorithms are affected by noise in finding optimal hyperparameter settings. While Chapter 5 focused on reducing the variance between optimization runs, this chapter focuses on achieving the best average performance. We particularly explore whether repeated evaluation of the same hyperparameter settings is necessary and optimal. For this we benchmark a number of model-free and model-based methods using single and repeated evaluations per hyperparameter setting. Experiments are conducted on recent high-noise machine learning tasks from the domains of image generation and reinforcement learning. We demonstrate that model-based optimizations where the model can estimate the amount of variation within hyperparameter settings can obtain superior results using just one sample per setting. We conclude the thesis in Chapter 7 with suggestions for future work.

Chapter 2

Background

2.1 Hyperparameter Optimization

2.1.1 Motivation

Hyperparameters are tuning parameters of machine learning models that cannot be learned as part of the training procedure. Hyperparameter optimization refers to the process of tuning hyperparameters to optimize performance. This is crucial for a machine learning model to achieve optimal performance. Hyperparameter optimization can often be a manual process of trial and error in which the researcher tries configurations and comes up with new configurations based on observed results and experience. The field of automated hyperparameter optimization, so-called *auto ML*, or simply *hyperparameter optimization* attempts to automate this process. Automation is important for three reasons. The first reason is to free up researcher time. The manual process of trial and error also known as *grad student descent*[1] can require significant amounts of human effort. The second reason is to improve

[1] https://sciencedryad.wordpress.com/2014/01/25/grad-student-descent/

performance. Hyperparameter optimization can often find better settings than even experienced researchers as shown by a number of works [Snoek et al., 2012, 2015, Melis et al., 2017]. The third reason that automation is important is to improve reproducibility. Clearly, it is easier to reproduce an automated search and assure equal allocation of efforts between searches than it is when done manually by a researcher.

2.1.2 Problem Statement

Assume a prediction problem in which the available data set is split into $\mathcal{D}^{\text{train}}$, $\mathcal{D}^{\text{valid}}$, and $\mathcal{D}^{\text{test}}$ and considered fixed thereafter. Training algorithm $\mathcal{A}$ is used to obtain a function f by minimizing the in-sample prediction error $\mathcal{A}(\mathcal{D}^{\text{train}}) = f$. The function f represents the prediction model. For example, f may be a trained neural network. $\mathcal{A}$ typically involves optimization of f's parameters θ. Commonly $\mathcal{A}$ also has parameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ called hyperparameters, so $\mathcal{A}_{\boldsymbol{\lambda}}(\mathcal{D}^{\text{train}}) = f_{\boldsymbol{\lambda}}$. Here, $\boldsymbol{\Lambda} = \Lambda_1 \times \Lambda_2 \times \ldots \Lambda_M$ denotes the hyperparameter space where Λ_m can be continuous, discrete valued, or categorical.

To choose $\boldsymbol{\lambda}$ based on the hold-out validation set and a loss function $\mathcal{L}$ we seek to minimize the expected prediction error (EPE):

$$\overset{*}{\boldsymbol{\lambda}} = \arg\min_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \mathbb{E}_{(\mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}}) \sim \mathcal{D}} \; \Psi(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, \mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}}) \tag{2.1}$$

where $\mathcal{D}$ is the data generating process. The validation loss Ψ is given by

$$\Psi(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, \mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}}) = \frac{1}{\mid \mathcal{D}^{\text{valid}} \mid} \sum_{(x,y) \in \mathcal{D}^{\text{valid}}} \mathcal{L}(y, f_{\boldsymbol{\lambda}}(x)) \tag{2.2}$$

such that $f_{\boldsymbol{\lambda}} = \mathcal{A}_{\boldsymbol{\lambda}}(\mathcal{D}^{\text{train}})$. Since the data generating process $\mathcal{D}$ is generally unavailable, different strategies exist to approximate $\mathbb{E}_{(\mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}}) \sim \mathcal{D}} \; \Psi(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, \mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}})$. Given a fixed dataset $D \sim \mathcal{D}$ the expectation can be approximated by averaging Ψ across different

training-validation splits of D. For very large datasets, it is often sufficient to use just one training-validation split. The optimization will therefore minimize the validation loss Ψ over the hyperparameter space $\mathbf{\Lambda}$ and we will refer to $\Psi(\mathcal{L}, \mathcal{A}_{\boldsymbol{\lambda}}, \mathcal{D}^{\text{train}}, \mathcal{D}^{\text{valid}})$ as $\Psi(\boldsymbol{\lambda})$ since $\mathcal{L}$, $\mathcal{A}$, $\mathcal{D}^{\text{train}}$, and $\mathcal{D}^{\text{valid}}$ are fixed during the optimization. Note that while this notation assumes a supervised learning problem this setup equally holds for unsupervised or reinforcement learning scenarios so long as an objective value can be calculated. Chapter 6 extends this notation to tasks other than supervised learning.

2.1.3 Optimization Algorithms

Model-free Algorithms

The most basic hyperparameter optimization algorithm is grid search. In this case the user specifies choices for each hyperparameter and the algorithm evaluates the full factorial design, that is every combination of choices. Grid search has the obvious disadvantage that it scales poorly with the number of hyperparameters. An alternative is therefore to use random search [Bergstra and Bengio, 2012]. Random search samples configurations by sampling hyperparameter values uniformly at random from specified ranges. This has several advantages. Given a total of N explored $\boldsymbol{\lambda}$ values and M hyperparameter dimensions, grid search is only able to explore $N^{1/M}$ settings per hyperparameter dimension. Random search, however, can explore N unique settings for each hyperparameter dimension [Hutter et al., 2019]. In scenarios where some hyperparameters are more important than others this allows random search to explore more settings for the important hyperparameter and therefore often results in better performance. It is also easier to parallelize random search, since evaluations are independent of one another. Finally, a random search can be stopped early or continued for longer than planned while still yielding a valid random search. Chapter 4 implements grid search and random search as optimization algorithms for Sherpa. Random search is also

used as the hyperparameter suggestion algorithm in Chapters 3 and 5 and as a benchmark algorithm in Chapter 6.

Bayesian Optimization

Bergstra et al. [2011] and Snoek et al. [2012] demonstrated the feasibility of using Bayesian optimization for hyperparameter optimization. Bayesian optimization (BO) is a model-based approach to global derivative-free optimization of blackbox functions. Given existing observations of configurations and associated objective values, BO fits a surrogate model from the search space to the objective. This model is usually faster to evaluate than the objective function itself. For this reason, a large number of predictions of objective values can be made. An *acquisition function* describes the utility of each prediction. By maximizing the acquisition function one can find the optimal next point to evaluate for the true objective function [Hutter et al., 2019].

As a surrogate model for the function from the search space $\boldsymbol{\lambda}$ to the objective Ψ, it is common to use a Gaussian process $\mathcal{GP}(m(\boldsymbol{\lambda}), k(\boldsymbol{\lambda}, \boldsymbol{\lambda}))$. To allow for sufficient structure in the surrogate, a Mátern 5/2 kernel is commonly used. The hyperparameters of the kernel can be found by maximizing the marginal likelihood [Rasmussen, 2003] or via sampling for a fully Bayesian approach [Snoek et al., 2012]. In the noiseless case, posterior predictions for the mean and variance take the form:

$$\mu(\boldsymbol{\lambda}) = \mathbf{k}_* \mathbf{K}^{-1} \mathbf{y}, \ \sigma^2(\boldsymbol{\lambda}) = k(\boldsymbol{\lambda}, \boldsymbol{\lambda}) - \mathbf{k}_*^T \mathbf{K}^{-1} \mathbf{k}_*. \tag{2.3}$$

Here, $\mathbf{k}_*$ represents the covariance between $\boldsymbol{\lambda}$ and all previously sampled points and $\mathbf{K}$ represents the covariance matrix between all previously sampled points.

For the acquisition function the most popular choice is the *expected improvement* (EI) [Jones

et al., 1998] (for minimization):

$$EI(\boldsymbol{\lambda}) = \mathbb{E}[\max(\Psi_{min} - \mu(\boldsymbol{\lambda}), 0)]. \tag{2.4}$$

where the expectation is taken over $\Psi(\boldsymbol{\lambda})$ and Ψ_{min} is the best observed validation loss so far. The EI is convenient because it can be evaluated analytically for the Gaussian process model:

$$EI(\boldsymbol{\lambda}) = (\Psi_{min} - \mu(\boldsymbol{\lambda}))\Phi\left(\frac{\Psi_{min} - \mu(\boldsymbol{\lambda})}{\sigma(\boldsymbol{\lambda})}\right)\sigma\phi\left(\frac{\Psi_{min} - \mu(\boldsymbol{\lambda})}{\sigma(\boldsymbol{\lambda})}\right). \tag{2.5}$$

While the combination of Gaussian Process surrogate model and the expected improvement acquisition function is the most popular choice, a number of other models and acquisition functions have been proposed. Bergstra et al. [2011] propose a Tree of Parzen estimators (TPE) as a model. The TPE approach models $p(\boldsymbol{\lambda}\,|\Psi)$ and $p(\Psi)$ (as opposed to $p(\Psi|\,\boldsymbol{\lambda})$). In particular, $p(\boldsymbol{\lambda}\,|\Psi)$ is modeled as two non-parametric densities. One density is chosen for $\Psi < \Psi^*$ and another for $\Psi \geq \Psi^*$ where Ψ^* is a user-specified quantile of Ψ. An advantage of the TPE approach is its scalability to large numbers of observations, as well as its natural application to conditional hyperparameter spaces.

In other works, Hutter et al. [2011] employ a random forest (RF) to model $p(\Psi|\,\boldsymbol{\lambda})$. The RF has a smaller training complexity of $\mathcal{O}(n\log(n))$, as opposed to $\mathcal{O}(n^3)$ for the GP. Here n is the number of observed hyperparameter settings and corresponding objective values. This makes it a good choice when a large number observations is feasible. Furthermore, it can naturally incorporate categorical and conditional hyperparameters. However, the RF is not a probabilistic model and therefore does not provide uncertainty estimates. As a substitute [Hutter et al., 2011] use empirical estimates of the variance across trees in the ensemble. Unfortunately, these estimates are not necessarily well calibrated. For example, the variance estimates may not increase when extrapolating away from observed data as described in

[Shahriari et al., 2015].

Finally, Snoek et al. [2015] and Springenberg et al. [2016] utilize Bayesian neural networks to model $p(\Psi|\,\boldsymbol{\lambda})$. As explained in [Hutter et al., 2019], Bayesian neural networks can outperform GP based approaches for ~ 250 function evaluations or more.

We implement Bayesian optimization in Chapter 4 using the GPyOpt [authors, 2016] package. Chapter 6 dives deeper into the application of Bayesian optimization in settings with high observation noise.

Multi-fidelity Optimization

The complexity of machine learning models has increased dramatically over the last years with modern algorithms taking on the order of days or weeks to train. This makes hyperparameter optimization difficult since an evaluation of the objective function typically requires the full training and evaluation of the machine learning model. While more sophisticated searches such as Bayesian optimization do provide increased efficiency over random search, they still require at least $M+1$ observations to start making suggestions and usually many more to converge. An alternative approach is therefore to probe hyperparameter configurations using subsets of the training dataset, subsets of features, or a reduced number of training iterations. These approaches fall under the class of multi-fidelity optimization. Domhan et al. [2015] approach this concept by using Bayesian neural networks to model learning curves. Learning curves in this case correspond to the objective function value as a function of training iterations. By predicting the final objective value for a hyperparameter configuration, bad performers can be stopped early. Which hyperparameter settings to evaluate is in this case based on Bayesian optimization. Empirically this results in an approximately two-fold speed up of the hyperparameter search. Klein et al. [2016b] extend this methodology by predicting a weighted combination of parametric functions and conditioning

on the hyperparameter space. Golovin et al. [2017] follow a similar approach but using a GP with a specially designed kernel for learning curve modeling. Swersky et al. [2014] also use a GP model to predict learning curves, but incorporate this into the Bayesian optimization via an information-theoretic framework. Finally, Klein et al. [2016a] use a GP to jointly model loss and training time as a function of dataset size. Using this approach the authors report speed-ups of 10 to 100 times compared to standard Bayesian optimization.

Recently, approaches based on multi-armed bandits have also been applied to multi-fidelity optimization. Jamieson and Talwalkar [2016] applied the successive halving algorithm (SHA) to allocate increasing budgets to smaller and smaller subsets of hyperparameter configurations. In particular, given a number of candidate configurations, SHA:

1. Uniformly allocates a fixed budget to all candidates.
2. Evaluates the performance of all candidates.
3. Promotes the top half of candidates to the next rung.
4. Doubles the budget per configuration in the next rung.
5. Repeats until one configuration remains.

The SHA algorithm has the downside that the user has to decide on the number of configurations and the size of the initial budget. Li et al. [2016] approach this issue with the Hyperband algorithm by doing a grid-search over these values, covering scenarios of more configurations and smaller budgets as well as less configurations and larger budgets. Each combination then calls SHA as a sub-routine. The authors report speed-ups of over an order of magnitude compared to comparison methods such as standard Bayesian optimization. In addition, Li et al. [2018] fully parallelized the SHA algorithm.

Finally, Falkner et al. [2018] combined Hyperband with TPE-based Bayesian optimization.

The authors show that Hyperband yields large speed-ups for small budgets. However, given a large enough budget Bayesian optimization can catch-up and outperform Hyperband due to its model-based sampling. Combining Hyperband with Bayesian optimization provides up to 20-fold speed-ups for a small budget and up to 55-fold speed-ups at large budgets over standard random search according to an experiment by the authors[2].

Multi-fidelity optimization is implemented in Chapter 4 of this thesis via the asynchronous SHA algorithm. This implementation is then used as a benchmark algorithm for noisy hyperparameter optimization in Chapter 6.

Large Scale Approaches

A number of population based methods have been applied to the problem of hyperparameter optimization. These approaches include genetic algorithms, evolutionary algorithms, and particle swarm optimization. Most of these methods maintain a population of hyperparameter configurations and apply local perturbations (mutations) and combination of population members (cross-over) to obtain a new, improved generation. These methods are conceptually simple and easily parallelizable. While some approaches such as Population-based Training [Jaderberg et al., 2017] have been designed specifically with neural network training in mind and are feasible for smaller-scale hyperparameter optimizations, standard evolutionary methods often require a large number of function evaluations. For example, Real et al. [2017] demonstrate the competitiveness of evolutionary algorithms in designing neural network architectures, however, based on a computational budget that is feasible for few practitioners.

Similarly, deep reinforcement learning has recently been applied to optimizing neural network architectures [Zoph and Le, 2016]. While such approaches can achieve close to a new state-

[2] `https://www.automl.org/blog_bohb/`

of-the-art, these also suffer from immense computation requirements. Zoph and Le [2016] trained 12800 neural network architectures for one optimization run requiring 32,400-43,200 GPU-hours which is far more than is usually feasible.

In this thesis we implement the Population-based Training algorithm in Chapter 4. Due to the large amount of computation required and the relatively narrow application field, reinforcement learning and evolutionary approaches for hyperparameter suggestion are not in implemented.

2.2 Group Sequential Testing

In Chapter 5 we use group sequential testing methodology to efficiently choose the optimal hyperparameter setting among a pool of candidates in the presence of noise. This section aims to provide a general background on this methodology. Group sequential testing allows to stop a hypothesis test before the full sample size is reached if observational data suggests to. The methodology was initially developed for clinical trials where there is an ethical issue of allocating a new treatment to human subjects. The resulting framework can be applied in any case where data accumulates and efficiency in terms of the number of observed samples is a priority. In the following we illustrate group sequential testing methodology via a test for equality of two continuous random variables. In particular, we describe three approaches to the design of group sequential stopping rules and explain their different characteristics.

2.2.1 Motivational Example and Notation

We consider the case of two groups A and B with corresponding normal response variables X_A and X_B. The random variables X_A and X_B have means μ_A and μ_B, respectively, and

common known variance σ^2. Of interest is the test of hypotheses:

$$H_0 : \mu_A - \mu_B = 0 \quad \text{vs.} \quad H_1 : \mu_A - \mu_B \neq 0. \tag{2.6}$$

We consider the case where the observations from each group accumulate one after another and where we test after every $2n$ accrued observations (n in each group) for a total of K tests. Let the tests be indexed by $j = 1, \ldots, K$ and define the statistic

$$S_j = \sum_{i=1}^{j} \frac{\bar{X}_{Ai} - \bar{X}_{Bi}}{\sqrt{2\sigma^2}}. \tag{2.7}$$

where $\bar{X}_{Ai}$ refers to the sample mean of the ith batch of n observations. Then S_j is a partial sum of independent identically distributed normal random variables with variance 1 and mean $\sqrt{n}\frac{\mu_A - \mu_B}{\sqrt{2\sigma^2}}$.

2.2.2 False Positive Rate under the Fixed Sample Approach

Under the fixed sample approach to this scenario H_0 gets rejected if

$$|S_j| > z_{1-\frac{\alpha}{2}}\sqrt{j}. \tag{2.8}$$

The multiplier $z_{1-\frac{\alpha}{2}}$ is given by $\Phi^{-1}(1 - \frac{\alpha}{2})$ for a theoretical type I error of α. Here, Φ is the cumulative density function of the standard normal distribution. A simulation of this scenario using $\mu_A - \mu_B = 0$, $\alpha = 0.05$, and 100,000 simulation runs provides an empirical false positive rate of approximately 12.6%. This is considerably higher than the theoretical 5%.

To further illustrate this, Figure 2.1 shows a draw of the sample mean under the null hypothesis at different sample sizes. The bars indicate intermediate analyses under a fixed sample

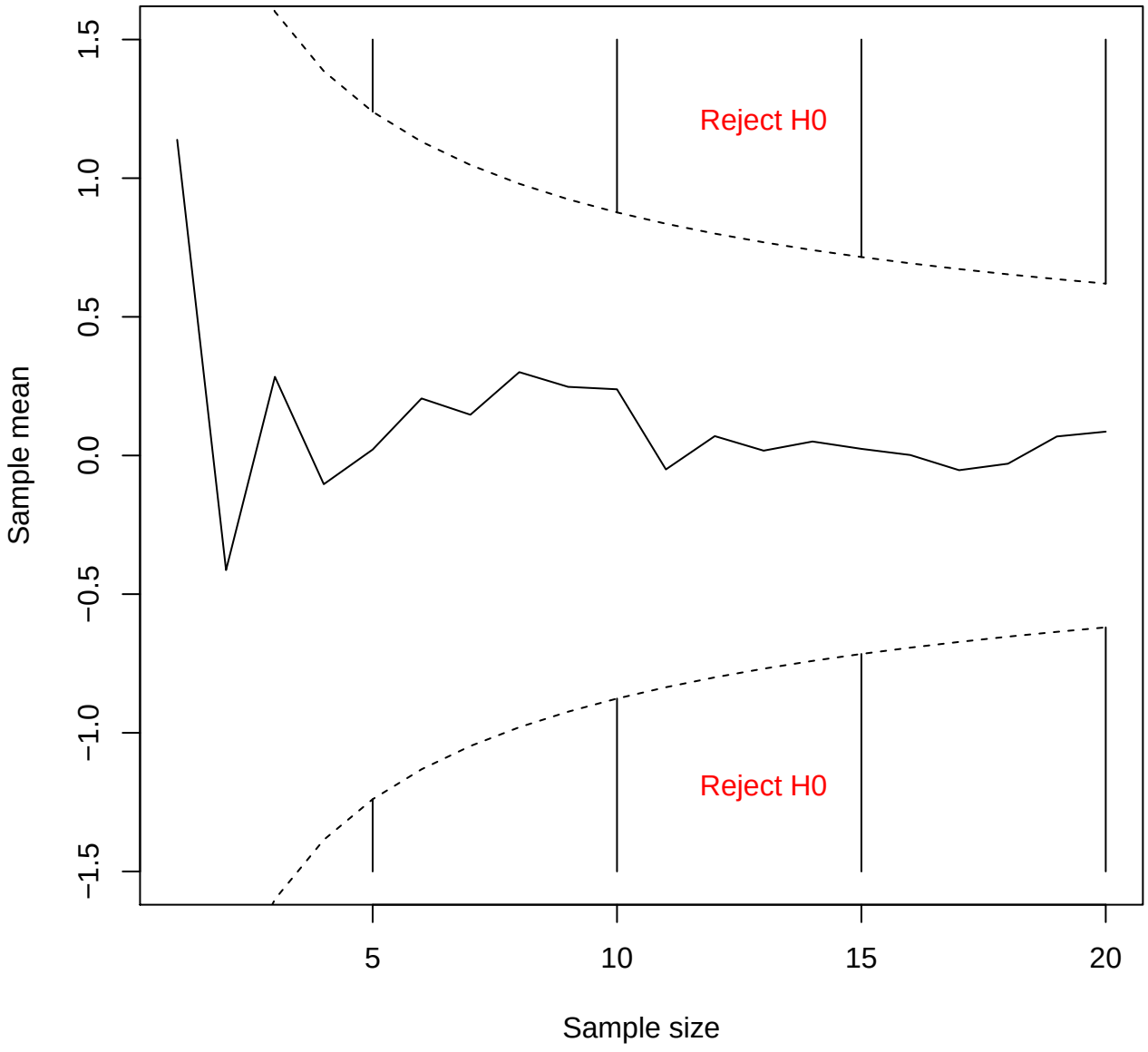


Figure 2.1: Plot of one draw of the sample mean against the number of observations.

stopping bound for a type I error of 5%. Simulating such a path over 1000 runs we obtain rejection rates of 0.042, 0.050, 0.049, and 0.049 for the individual intermediate analyses. As expected these match the specified type I error. The observed rate of stopping at any one analysis we find as 0.121. This means the overall type I error is not the intended 5%.

Armitage et al. [1969] illustrated this increase in false positive rate and derived the sampling distribution of the group sequential test statistic. Given continuation sets $\mathcal{C}_j$, the group sequential test statistic is given by (M, S) where $M = \min\{1 \leq j \leq K : S_j \notin \mathcal{C}_j\}$ and $S = S_M$. The sampling density for $(M = m, S = s)$ is then given by

$$p(m,s) = \begin{cases} f(m,s) & s \notin \mathcal{C}_m \\ 0, & \text{otherwise.} \end{cases}$$

where

$$`f(1,s) = \frac{1}{\sqrt{j}}\phi\left(\frac{s}{\sqrt{j}}\right) \tag{2.9}$$

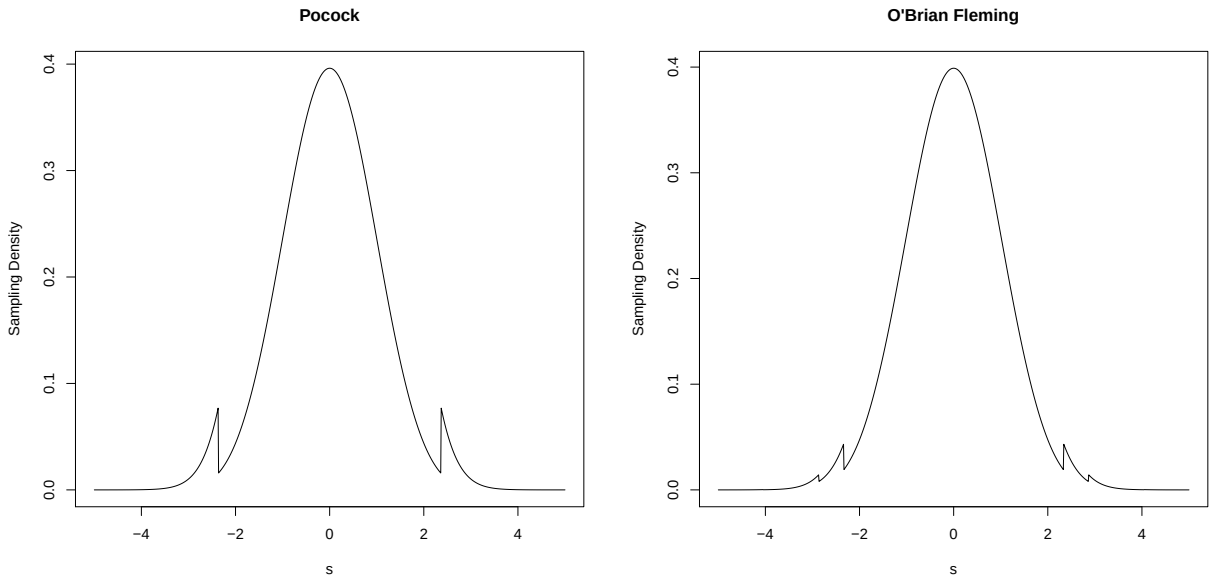
and

$$f(j,s) = \int_{\mathcal{C}_{j-1}} \frac{1}{\sqrt{j}}\phi\left(\frac{s-u}{\sqrt{j}}\right) f(j-1,u)du. \tag{2.10}$$

Here $\phi(x) = e^{-x^2/2}/\sqrt{2\pi}$ is the standard normal density.

2.2.3 Sequential Testing Boundaries

Using the density provided by Armitage et al. [1969] one naturally arrives at the Pocock boundary [Pocock, 1977]. For the Pocock stopping boundary one simply manipulates the multiplier of $\sqrt{j}$ so that it provides the correct type I error under the sequential density.



(a) (b)

Figure 2.2: Sampling densities of the normalized Z-score at the final analysis. The left plot shows the sampling density using a Pocock stopping boundary, the right plot is based on an O'Brien-Fleming boundary.

Particularly, the multiplier c is found such that

$$P_K = 1 - \int_{-c\sqrt{j}}^{c\sqrt{j}} f(K, s)ds \tag{2.11}$$

is equal to the intended type I error α. The repeated testing is stopped at any analysis if $S_j \notin [-c\sqrt{j}, c\sqrt{j}]$. Since $S_j/\sqrt{j}$ corresponds to the Z-score and c is constant across interim analyses, the Pocock boundary is characterized by the fact that it is constant on the scale of the Z-statistic. Figure 2.2a shows the density of the normalized Z-statistic with $K = 4$ and $n = 5$. The density has spikes at the values of $\pm c$ due to the discontinuities at these points.

Another choice of stopping boundary was proposed by O'Brien and Fleming [1979]. In this case, the stopping boundary is constant on the scale of the partial sums S_j. Specifically, testing is stopped and the null hypothesis is rejected at analysis j if $|S_j| > d$. The value of d is found via simulation so that the type I error given by Equation 2.11 is as desired.

This choice of stopping boundary is more conservative at earlier analyses. Figure 2.2b shows the sampling density of the normalized Z-statistic at $K = 4$ using the O'Brien-Fleming stopping boundary. Due to the fact, that the O'Brien-Fleming boundary is not constant on the Z-scale, individual spikes can be seen for each analysis.

Table 2.1: Average sample sizes needed to reject the null hypothesis under Pocock and O'Brien-Fleming boundaries.

$\mu_A - \mu_B$	Pocock ($\Delta = 0.5$)	$\Delta = 0.25$	O'Brien-Fleming ($\Delta = 0$)
0	19.54	19.77	19.87
0.2	19.43	19.68	19.82
0.4	19.09	19.41	19.62
0.6	18.47	18.89	19.24
0.8	17.53	18.09	18.62
1	16.28	17.00	17.74
1.2	14.78	15.68	16.65
1.4	13.18	14.24	15.42
1.6	11.61	12.83	14.19
1.8	10.20	11.53	13.04
2	9.00	10.40	12.04

Wang and Tsiatis [1987] recognized that these stopping boundaries can be interpolated. Using the parameterization

$$|S_j| \geq \Gamma(\alpha, K, \Delta) j^{\Delta} \tag{2.12}$$

the Pocock boundary is defined as $\Delta = 0.5$ and the O'Brien-Fleming boundary by $\Delta = 0$. The scaling factor $\Gamma(\alpha, K, \Delta)$ is found through numeric search. This gives a choice of infinitely many boundaries to achieve desired operating characteristics.

2.2.4 Evaluation of Stopping Boundaries

We now turn to evaluating the presented stopping boundaries in terms of the average sample size needed to reject H_0 for different values of $\mu_A - \mu_B$. Table 2.1 shows average sample

sizes until stopping for different effect sizes and the Pocock, O'Brien-Fleming, and $\Delta = 0.25$ stopping boundaries. The table shows that a Pocock boundary requires on average the smallest sample size to stop. As expected, the boundary defined by $\Delta = 0.25$ is between the Pocock and O'Brien-Fleming boundaries in terms of average sample size. Figure 2.3 shows a plot of the observed difference in means required to stop at each interim analysis. The Pocock boundary is the tightest boundary at the first two analyses while the O'Brien-Fleming boundary is the tightest boundary at the last analysis.

We note here that these are operating characteristics and one may desire different characteristics for different applications. In clinical trials one often does not want to stop too early. As we apply these methods in Chapter 5 to the problem of hyperparameter optimization we want to minimize the average number of samples needed to detect a certain effect size. We will therefore, as explained later, focus on a Pocock stopping boundary.

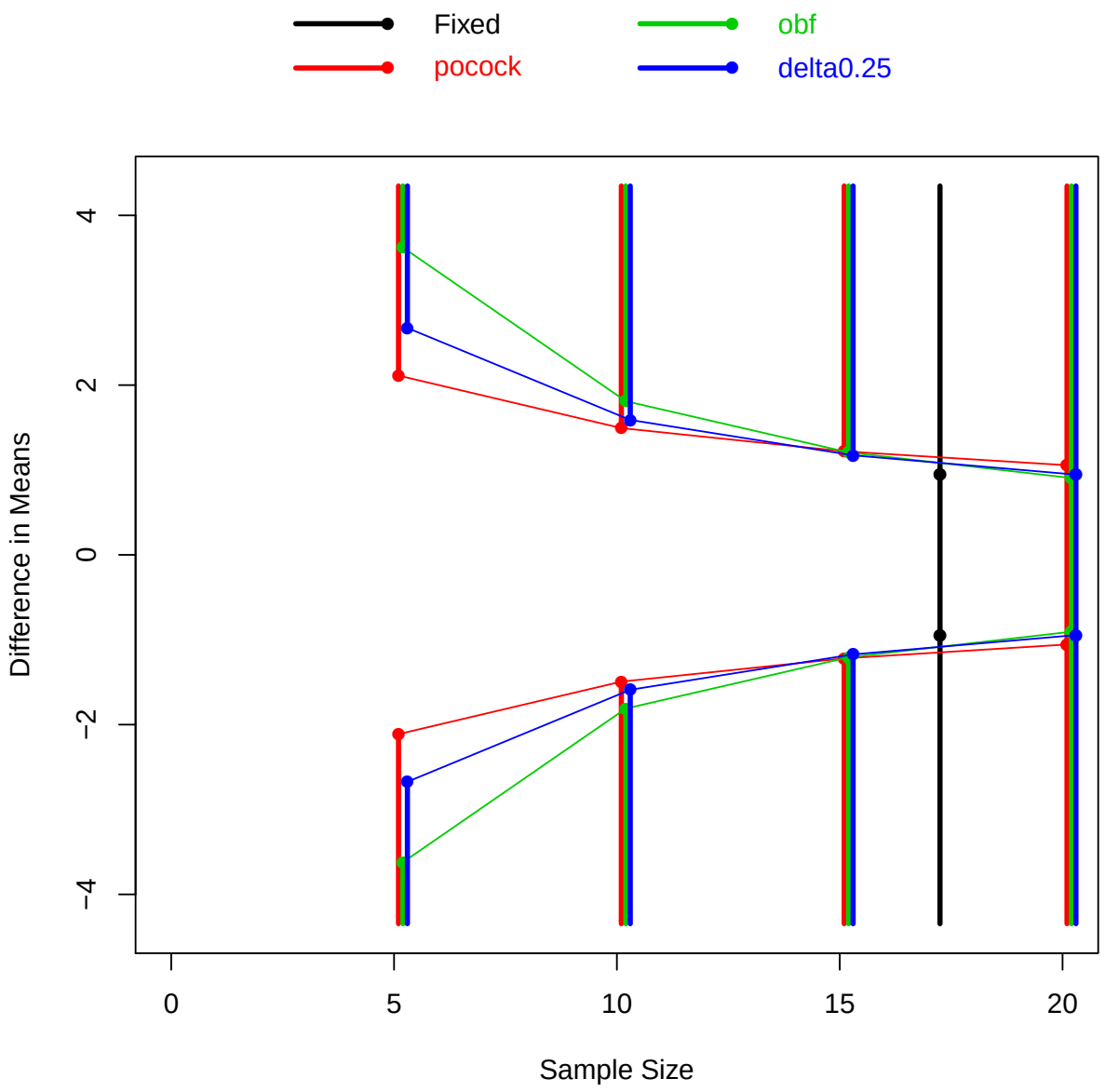


Figure 2.3: Pocock (poc) and O'Brien-Fleming (obf) stopping boundaries in terms of the observed difference in means required for testing to stop at each interim analysis.

Chapter 3

Improved Energy Reconstruction in NOvA with Regression Convolutional Neural Networks

[1]

3.1 Introduction

Energy reconstruction plays a key role in High Energy Physics (HEP), as it converts detector unit readout into kinematics of interactions. In a HEP analysis, event and particle types are usually identified first and then reconstructed energies are assigned to individual final state particles and the overall event via the energy reconstruction process. Based on these energies, physical phenomena can be studied as functions of overall energy and internal kinematics of an event. In neutrino physics, energy reconstruction is essential and challenging for the

[1]

neutrino oscillation studies. Traditionally, particle energies are reconstructed by adding up or fitting to the hits on detector readout units, and event energy is reconstructed as a function of particle energies in the event. In this paper, a deep-learning based method for energy reconstruction of neutrino oscillations will be discussed. The method directly uses detector hits as inputs without intermediate steps.

Neutrino oscillations are so far the only experimental observation beyond the standard model since its development about 30 years ago. Neutrinos are very elusive since they only interact via the weak nuclear force. They have three active flavor states ν_e, ν_μ, and ν_τ. Each is a different superposition of three mass states ν_1, ν_2, and ν_3. Neutrinos can oscillate between flavor states. The relationship between flavor and mass states, and the oscillation between flavors are commonly described by the Pontecorvo−Maki−Nakagawa−Sakata (PMNS) matrix [Pontecorvo, 1968].

Two fundamental questions of interest are remaining to be determined by studying neutrino oscillations. First, what is the CP phase δ? The CP phase δ relates to the difference in oscillation behavior between neutrinos and anti-neutrinos. CP violation in the lepton sector holds implications for matter-antimatter asymmetry in the Universe through leptogenesis. Second, what is the mass ordering ($m_3 > m_{1,2}$ or $m_{1,2} > m_3$) between neutrinos? The mass hierarchy provides key information for future searches of the neutrino-less double beta decay. Observing the neutrino-less double beta decay would imply that the neutrino is a Majorana particle meaning it is its own anti-particle. The mass hierarchy will also constrain the so far undetermined absolute neutrino masses.

Aiming to solve these two questions, current and future neutrino oscillation experiments focus on electron neutrino appearance ($\nu_\mu \rightarrow \nu_e$). Neutrino oscillation can be measured by sending a beam of neutrinos of one flavor through a detector. Oscillated neutrinos arriving in the detector will be of a different flavor than the one generated from the beam. Observed neutrino interactions need to be tagged by their flavors. Importantly, the energy of the

incoming neutrino needs to be well reconstructed as the $\nu_\mu \rightarrow \nu_e$ oscillation probability changes as a function of neutrino energy.

One of the challenges is thus a good estimation of electron neutrino energy. The accuracy of neutrino energy reconstruction influences how precisely neutrino oscillation parameters can be estimated. An electron neutrino can only be identified in charged current (CC) interactions where the electron neutrino converts into an electron. The $\nu_e - CC$ events are characterized by an electron along with other potential activity produced by hadrons. Traditionally, the reconstructed neutrino energy is calculated as a function of electron and hadron visible energy deposits. However, the estimation of the energy with the kinematics based method is complicated by missing energy in dead material, non-linear detector energy responses, invisible energy and identities (mass) of hadrons, and overlaps between electron and hadron showers.

To address these issues, the neutrino energy can be predicted directly from images of interactions. These images provide additional information on interaction details such as trajectories and energy deposit patterns of electrons and hadrons. Deep learning and deep convolutional neural network methods are a natural choice for processing data produced by complex detectors in high energy physics [Shimmin et al., 2017, Sadowski et al., 2017, Baldi et al., 2016, 2014] and have demonstrated success in **classification** problems in collider and neutrino experiments. NOvA has pioneered this deep learning technique in flavor tagging problems and has used it to produce oscillation physics results [Aurisano et al., 2016, Adamson et al., 2017]. CNN based event identification and reconstruction have also been investigated in other neutrino experiments [Racah et al., 2016, Acciarri et al., 2017, Renner et al., 2017, Ghosh, 2018, Delaquis et al., 2018]. In this work, we propose to develop a **regression** CNN based method to precisely reconstruct electron neutrino energy and electron energy in the NOvA neutrino experiment. Neutrino interactions at NOvA typically involve multiple final state particles with complicated kinematics. The convolutional filters in CNNs can extract

a richer set of features of these events than the sum of energy deposits. For example, two neutrinos could deposit almost exactly the same amount of energy in the detector and convolutional neural network features learned from event topologies could be used to make a more refined prediction on the true neutrino energy than the kinematics based method.

Reconstruction of individual final state particle energy in an interaction is a basic task of energy reconstruction. Specifically, these particle energies are used to study the kinematics, such as the momentum and energy transfer, in neutrino interactions. To demonstrate that the regression CNN can also reconstruct single particle energy, a similar method to the electron neutrino energy estimator is used to estimate electron energy. This estimator can be used for cross-section measurements and shower reconstruction study.

3.2 The NOvA Experiment

NOvA uses an intense neutrino beam and sends it through sensitive, fine-grained detectors for long periods of time. The NuMI (Neutrinos at the Main Injector) muon neutrino beam is produced at Fermilab, Illinois. Aimed at 3.3 degrees downward, the beam travels 810 km through the earth to the 14 kilotons far detector (FD) in Ash River, Minnesota [**?**]. The far detector measures electron neutrinos oscillated from muon neutrinos in the beam. The NOvA experiment has the longest beam-detector distance in the world which maximizes the matter effect and allows a measurement of the neutrino mass ordering. Additionally, a 330 ton functionally identical near detector at Fermilab measures unoscillated beam neutrinos and estimates backgrounds and signals at the far detector. Both detectors are located 14 milli-radians off the centerline of the neutrino beam. This allows the detectors to capture a narrow energy spectrum of neutrinos at approximately 2 GeV. This is the energy at which the oscillation probability from a muon neutrino to an electron neutrino is expected to be at its peak.

The NOvA detectors are constructed in layers of alternating vertical and horizontal PVC cells; activities in the cells are recorded in a top view and a side view. There are 344,064 cells in the far detector and 18,000 cells in the near detector. In the far detector, each cell is 3.9 cm wide, 6.0 cm deep and 15.6 m long. The cells are made of highly reflective plastic filled with liquid scintillator. The scintillation light produced by neutrino interactions in the detectors is collected by a wavelength shifting fiber connected to an avalanche photodiode installed on one end of each cell. The readouts from these photodiodes are converted to calorimetric energy for physics analyses.

3.3 Methods

3.3.1 Simulated Data Sample

Simulation

The standard NOvA simulation is used to generate training and validation samples for the regression CNN. The simulation of NuMI neutrino beam is described in Ref. [Acero et al., 2018]. The beam is simulated by GEANT4 [Agostinelli et al., 2003] and corrected according to external thin-target hadroproduction data with the PPFX tool [Aliaga et al., 2016]. The flux shape of the NuMI neutrino beam at NOvA is referred to as the regular flux in this paper. Since NOvA is an off-axis experiment, the neutrino spectrum at the NOvA far detector from the NuMI flux peaks at about 2 GeV, close to the $\nu_\mu \to \nu_e$ oscillation maximum. Since there are few low energy neutrino (< 1 GeV) events in the NOvA FD Monte Carlo sample with the regular NuMI flux, the regression CNN ν_e energy trained with it has a significant true energy dependence (see Section 3.4). To minimize the dependence of estimated neutrino energy on true neutrino energy in the ν_e energy training, a flat neutrino flux shape is used

to generate the far detector ν_e CC Monte Carlo sample to train the regression CNN for ν_e energy reconstruction. In the case of electron shower energy estimation, electrons from the regular flux ν_e CC far detector Monte Carlo sample are used in the electron energy regression CNN training and its validation.

At NOvA, interactions of neutrinos on nuclei are simulated by GENIE [Andreopoulos et al., 2010], and detector responses are then simulated by GEANT4. The customized NOvA detector simulation chain is described in [Aurisano et al., 2015].

To study the ν_e interactions in the far detector Monte Carlo, we generate ν_e interactions with energies taken from the ν_μ flux distribution. After that, no neutrino oscillations are applied to the training samples. This is equivalent to assuming all muon neutrinos oscillate to electron neutrinos in the far detector. To study realistic energy reconstruction performances from different energy estimators, the real ν_e appearance signal in the far detector can be obtained by applying realistic oscillation weights to this sample.

The simulation produces image pairs of the entire detector. As explained in Section 3.2 the two images correspond to cells in the top view (X-view) planes and side view (Y-view) planes of the detector. The images have a size 896×384 (horizontal × vertical), where each pixel corresponds to the energy deposited in the corresponding detector cell. The horizontal coordinate (0-895) of a pixel represents the plane index of the detector cell and the vertical coordinate (0-383) represents the cell index in that plane. Since X-view planes and Y-view planes are assembled alternatively, all pixels with odd (even) plane indices are set to zero for X-View (Y-View). The neutrino flavor and the type of interaction are tagged by the true neutrino interaction information in GENIE.

Reconstruction

The overall reconstruction process at NOvA is described in [Baird et al., 2015]. First, different neutrino interactions captured in the same pair of detector views are separated [Baird, 2015]. Cell hits are clustered by space and time. This separates neutrino interactions caused by beam neutrinos from cosmic ray neutrinos in a time window. The procedure collects cell hits from a single neutrino interaction (slice). The slices then serve as the foundation for all later reconstruction stages. We will refer to one slice as neutrino interaction from here on.

For each neutrino interaction, the vertex is then identified. The vertex is where the neutrino interacts with the detector material. All particles created in the interaction originate at the vertex. In order to reconstruct the vertex position, a modified Hough transform is used to fit straight-lines to cell hits. Then the lines are tuned in an iterative procedure until they converge to the image's reconstructed vertex [Ester et al., 1996, Fernandes and Oliveira, 2008, Gyulassy and Harlander, 1991, Krishnapuram and Keller, 1993, Niner, 2015]. The cell closest to the reconstructed interaction vertex in each view is chosen as the reference cell in our pixel maps. We will refer to the reference cell as the reconstructed vertex from here on.

Since the size of the neutrino interaction is much smaller than the entire detector, the image can be cropped. An image can be cropped by considering the number of pixels that are occupied in all four directions from the reconstructed vertex. To determine a window size the distribution of electron neutrino interactions is inspected. The cropped image contains 30 pixels to the left and 120 pixels to the right of the vertex. In the vertical-direction, 70 pixels above and below the vertex are included. This produces images of 151×141 pixels in each view. On average, 99.5% of the hits are contained in a cropped image.

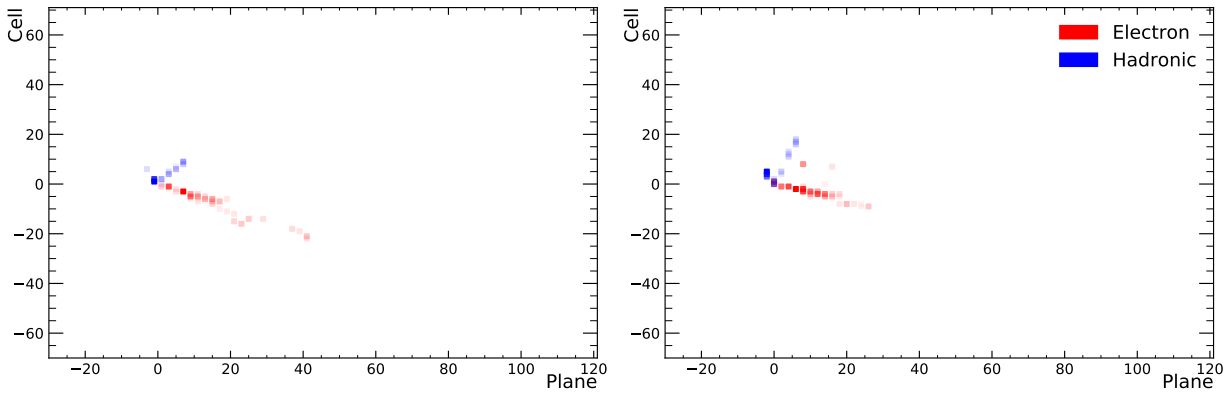


Figure 3.1: Electron neutrino image pair example.

The Electron Neutrino Dataset

True neutrino information is used to select ν_e CC events from all neutrino interactions. To to speed up the processing time, a loose pre-selection is applied to remove events with long prongs or too many hits. The pre-selection requires the number of occupied cells in the neutrino interaction to be less than 200. Additionally, the length of the longest prong is required to be less than 500 cm. Prongs are collections of cell hits with a start point and direction, which are reconstructed based on distances from hits to the lines associated with each of the particles that paths emanating from the reconstructed vertex [Ester et al., 1996, Fernandes and Oliveira, 2008, Gyulassy and Harlander, 1991, Krishnapuram and Keller, 1993, Niner, 2015]. The pre-selection keeps most of the electron-neutrino appearance signal while rejecting a large fraction of background events with long muon tracks. No requirements are applied to calorimetric energy or reconstructed neutrino interaction identities.

We use 0.98 million simulated samples of electron neutrino interactions as the electron neutrino dataset. Each sample consists of a pair of images from the two detector views, the reconstructed vertex and the simulated truth of electron neutrino energy. We split the dataset into a training sample with 0.75 million events and a validation sample with 0.23 million events. One example pair of images from the training dataset is shown in Figure 3.1.

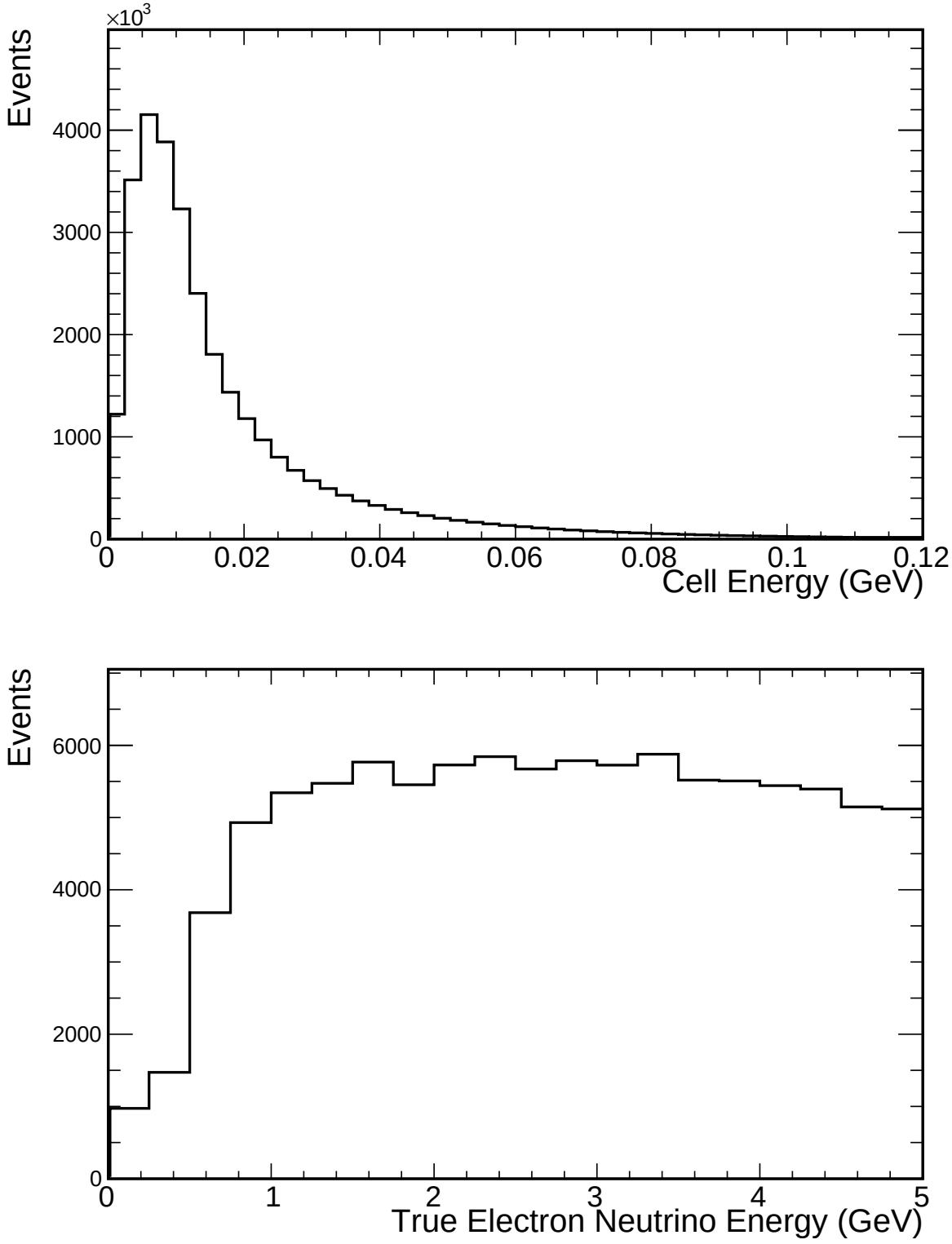


Figure 3.2: Distributions of input cell energies (top) and true electron neutrino energies (bottom) from a subset of the flat flux training sample.

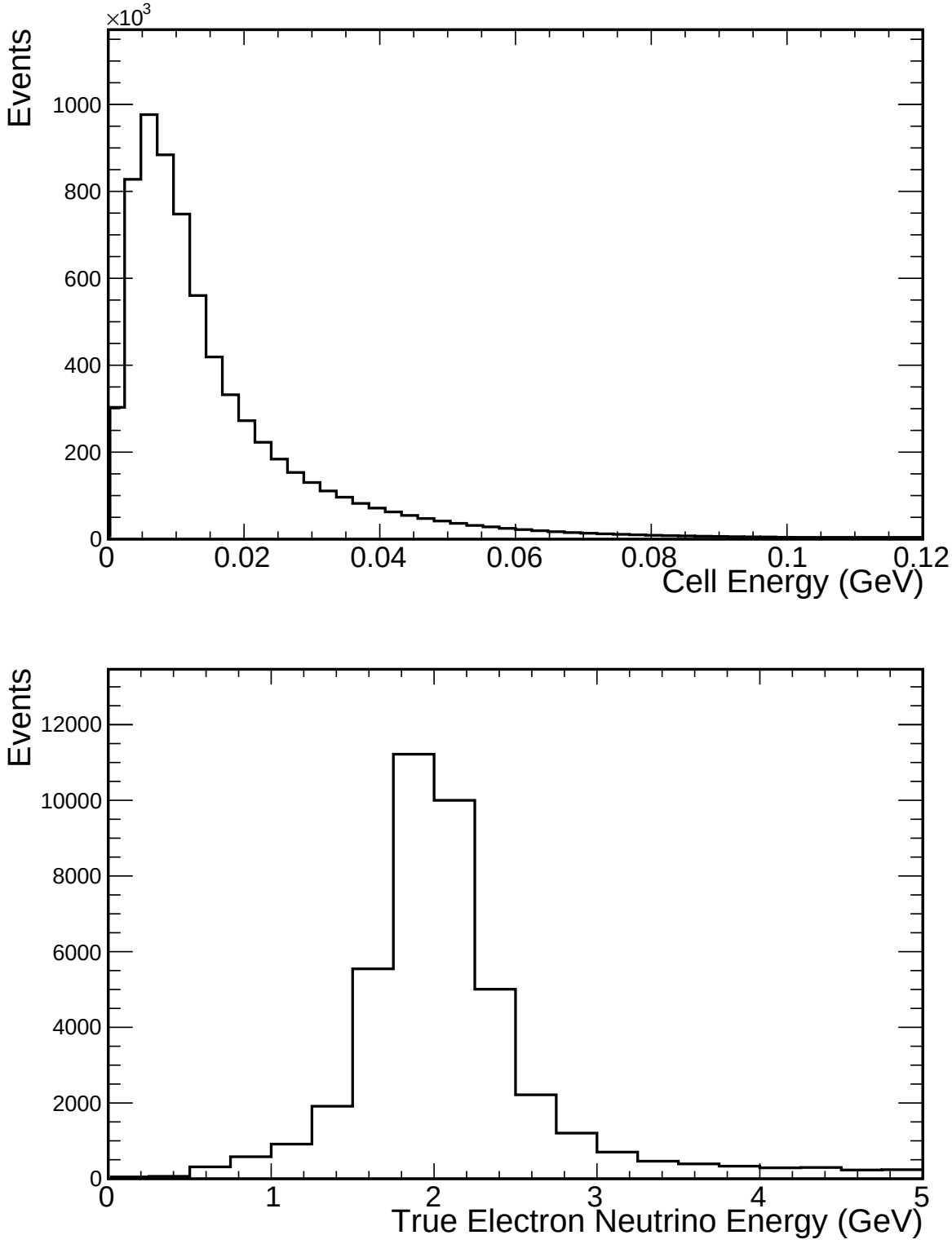


Figure 3.3: Distributions of input cell energies (top) and true electron neutrino energies (bottom) from a subset of the regular flux training sample.

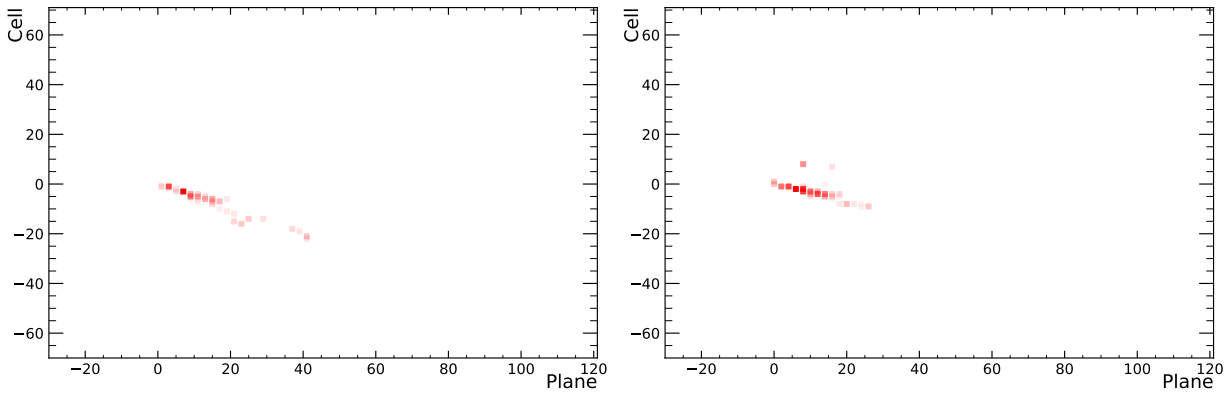


Figure 3.4: Electron shower image pair example.

In Figure 3.2 (left) we show the spectrum of cell energy deposits in cell hits from a subset of the flat flux training sample. Figure 3.2 (right) shows the spectrum of true ν_e energy from the subset of the flat flux training sample. One can find that there are enough events in the low ν_e energy region (< 1 GeV) for training. As a comparison, in Figure 3.3 the cell hit energy deposits and $\nu_e$energy from the regular flux ν_e FD Monte Carlo sample are shown. Since NOvA is an off-axis experiment, the ν_e energy in the FD is bell-shaped peaking around at 2 GeV, and there are few events below 1 GeV for training.

The Electron Shower Dataset

Electron shower images are created from electron neutrino interactions by reconstructing the pixels corresponding to the electron shower and setting cell energy values for all other pixels to zero. We select electron showers from ν_e CC FD Monte Carlo events by matching the reconstructed shower direction to the true particle direction.

The creation of electron shower pixel maps starts with prongs. First, the shower core is defined based on the prong direction provided by the prong cluster. Then signal hits are collected in a column around this core. The electron deposits energy through ionization in the first few planes before it starts multiple scattering. In order to capture all deposits from

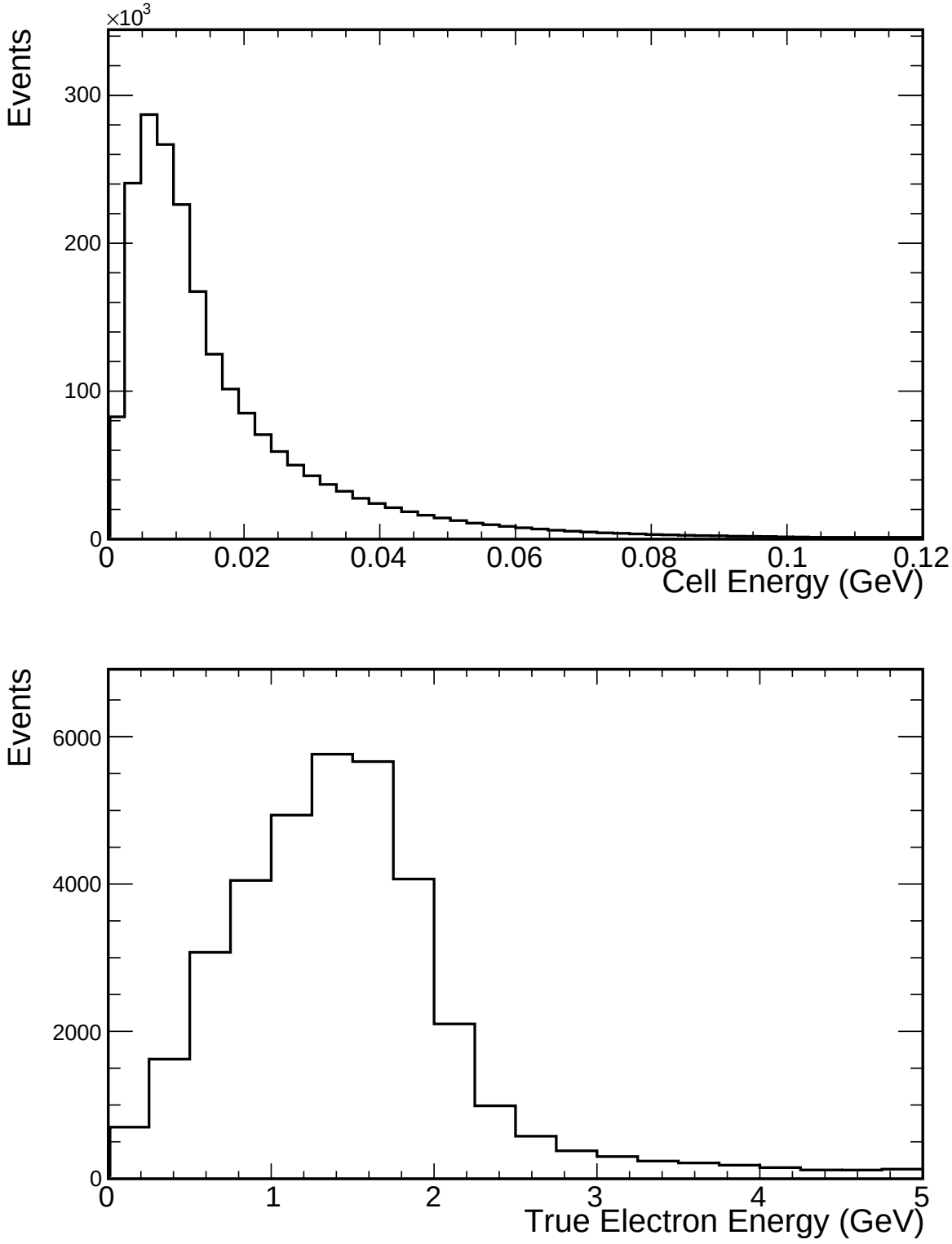


Figure 3.5: Distributions of input cell energies (top) and true electron energies (bottom) from a subset of the regular flux training sample.

the electron shower, the region corresponding to multiple scattering is enlarged. We require the radius to be twice the cell width for the first 8 planes from the start point of the shower. For the following planes, we require the radius to be 20 times the cell width [Bian, 2015].

Electrons from the regular flux ν_e CC FD Monte Carlo sample are used for training. One example pair of images from the training dataset is shown in Figure 3.4. We use 660k simulated samples of electron showers as the electron shower dataset. Each sample consists of a pair of images, the reconstructed vertex and the true electron energy. We split the dataset into 610,000 training samples and 50,000 validation samples. In Figure 3.5 (left) we show the spectrum of cell energy deposits from a subset of the regular flux training sample. Figure 3.5 (right) shows the spectrum of true electron shower energies from the subset of training sample.

3.3.2 Neural Network Architecture

The electron neutrino energy model and electron shower energy model are equal in architecture but weights are not shared across the two predictors. The neural network input consists of two matrices of shape $(151, 141)$ which represent the pixel values of the images and the cell indices for each view given by the reconstructed vertex. The output is one positive real-valued number. Inputs and outputs for each model are illustrated in Figure 3.6.

The neural network architecture is modified from the architecture used by NOvA's CVN event classifier [Aurisano et al., 2016]. This architecture is optimized from the convolutional neural network GoogLeNet [Szegedy et al., 2015] developed for image recognition. In addition, NOvA's CVN utilizes a siamese network structure [**?**]. The siamese network structure consists of two identical sub-networks whose outputs are merged to produce the final output. Each sub-network processes an image from one view. Weights are not shared between the sub-networks to provide independent information aggregation in each view. The

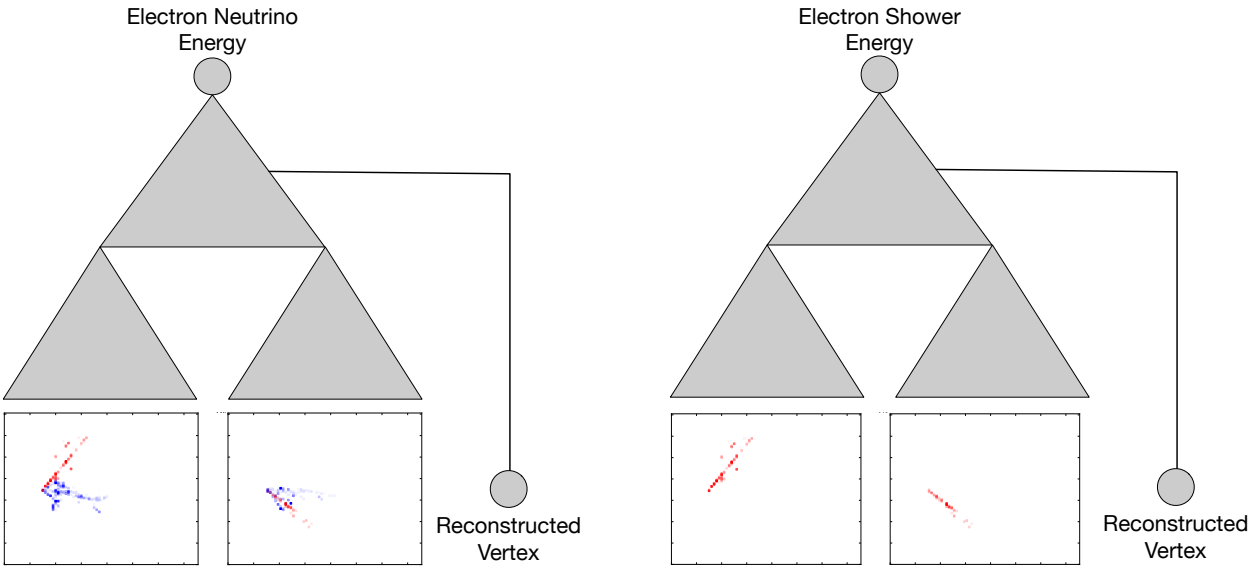


Figure 3.6: Diagram of electron neutrino energy predictor (left) and electron shower energy predictor (right). Triangles represent neural networks.

sub-networks are constructed from convolutional layers and pooling layers.

Convolutional layers apply a weight matrix in a sliding window fashion to the input image. This allows computing the same feature at different locations in the input image producing an output referred to as a feature map. Using multiple weight matrices allows learning a variety of features from the input images. Stacking these convolutional layers makes it possible to learn higher level features with each additional layer. Pooling layers take a feature map and reduce its dimensionality. This is done by tiling the feature map and reducing each tile to its maximum or average. Pooling layers are normally used between convolution layers. The convolutional layers in our network are based on the Inception module introduced by Ref. [Szegedy et al., 2015] as part of GoogLeNet.

GoogLeNet is the winner of the ImageNet Large Scale Visual Recognition Competition 2014 [Russakovsky et al., 2015] and state-of-the-art for convolutional neural networks with pixelmap inputs. Therefore, it is a good choice for our task. In particular, GoogLeNet uses the Inception module to efficiently extract features of different sizes from the input. This increases its modeling capacity without a significant increase in the computational cost.

Each sub-network here is constructed from a sequence of `Conv-MaxPool-Conv-Conv-MaxPool` layers followed by two Inception modules [Szegedy et al., 2015]. The Inception module is a specific configuration of convolutional layers built to simultaneously extract features of different dimensions. The features are then concatenated and pooled. Each convolutional layer and Inception module has 32 filters in the networks used. In the experiments, using additional Inception modules does not improve model performance. This is expected because the images here are sparse relative to natural images.

In the NOvA far detector, the scintillation light produced by neutrino interactions in each 15.5m-long detector cell is collected by the wavelength shifting fiber and read from the avalanche photodiode installed on one end of the cell. The attenuation of the scintillation light signal is a function of the distance from the interaction point to the readout photodiode, so the number of photoelectrons on the photodiode depends on the location of the interaction point. The readout threshold for each cell is a fixed number of photoelectrons, so the distribution of the cell energy deposit in each cell after the readout threshold cut is impacted by the position of the interaction with respect to the readout. This position dependence cannot be recovered by the cosmic attenuation calibration, which corrects the position dependence of the average number of photoelectrons for each cell using the minimum ionizing peak (MIP) position of cosmic muon hits. To consider this position effect in cell energies, we use the reconstructed vertex positions in the two views as neural network inputs. In our neural network architecture, after the inception modules and an average pooling layer, the output is flattened and concatenated with the reconstructed vertex position. A linear regression follows which produces the network output, the electron neutrino energy.

3.3.3 Neural Network Training

Neural networks for supervised learning are trained by defining a differentiable loss function L between neural network outputs $f_{\mathbf{W}}(\mathbf{x}_i)$ and target values y_i. Here, $\mathbf{W}$ represents the weights of the neural network and $\mathbf{x}_i$ is the neural network input. The loss function represents the metric by which the neural network accuracy is assessed. During training the neural network weights $\mathbf{W}$ are iteratively updated to minimize L using its gradient and a step size α.

Typically these updates are computed over mini-batches of size n which make up a partition of the total training dataset. Iteration over all mini-batches constitutes one epoch. Training parameters such as the step size α and the batch size n are referred to as hyperparameters must be selected before training and possibly tuned using the loss function on the validation dataset. We utilize the hyperparameter optimization software SHERPA [Hertel et al., 2018] which implements a number of hyperparameter optimization strategies and visualization tools. By automating the task, this software significantly speeds up the computationally expensive of finding optimal hyperparameters.

Unlike classification problems such as image recognition and particle identification, the target value y_i for the output (energy) of our regression neural network is a continuous variable varying over events. This requires the definition of an appropriate loss function for the task of the regression neural network. The goal is to minimize the standard deviation of a Gaussian fit to the peak of the histogram given by the energy resolution $\frac{E_{reco}-E_{true}}{E_{true}}$ on the test set. While this quantity cannot be directly optimized the absolute scaled error loss provides an appropriate surrogate for the task. The training loss function is then given by:

$$L(\mathbf{W}, \{\mathbf{x}_i, y_i\}_{i=1}^n) = \frac{1}{n}\sum_{i=1}^{n} |\frac{f_{\mathbf{W}}(\mathbf{x}_i) - y_i}{y_i}|. \tag{3.1}$$

Traditional loss functions for regression problems are the mean squared error $\frac{1}{n}\sum_{i=1}^{n}(f_{\mathbf{W}}(\mathbf{x}_i)-$

$y_i)^2$ or the mean absolute error $\frac{1}{n}\sum_{i=1}^{n}|f_{\mathbf{W}}(\mathbf{x}_i) - y_i|$. The former is often used due to its relationship to the log-likelihood when the data distribution is assumed to be Normal, its strict convexity, the fact it can be decomposed into variance and bias, and many other desirable properties. In our case, however, the mean squared error is suboptimal, because its derivative with respect to $\mathbf{W}$ is $\frac{1}{n}\sum_{i=1}^{n} 2(f_{\mathbf{W}}(\mathbf{x}_i) - y_i)\frac{\delta f_{\mathbf{W}}(\mathbf{x}_i)}{\delta \mathbf{W}}$. This increases proportionally with the distance of the predicted value from the truth. In other words, outliers will have increased impacts during gradient descent. In the training for neutrino energy, the events with large invisible energy due to dead material and hadronic interactions shouldn't have much larger impacts than those whose visible energy is close to the true energy, so we choose the absolute error instead of the squared error in the loss function. Furthermore, the original CVN/GoogleNet are designed and trained for classification tasks, we optimized training hyperparameters for the regression task.

Input image pixels are typically normalized to increase numerical stability and gradient quality. Here most image pixels are zero and the non-zero ones tend to be small. We apply three normalization methods: mean zero unit variance standardization, log transformation, and constant scaling. The three methods produce similar results. Therefore, a constant scaling factor of 100 is chosen after visual inspection of the input spectrum for ν_e (Figure 3.3 (right)) and electron (Figure 3.5 (right)).

The models are trained with stochastic gradient descent. The hyperparameter search yields as best hyperparameters: (1) a batch size of $n = 32$; (2) an initial learning rate of 5×10^{-4} with an exponential learning rate decay per batch of 1×10^{-5}; and (3) a momentum of 0.7. Models are trained for 100 epochs, or until the validation loss does not increase by at least 0.001 for 5 epochs. The weights from the epoch with the best validation loss are kept.

Regularization techniques are also explored using random search implemented in SHERPA. Regularization refers to a set of methods that reduce modeling capacity to prevent fitting to noise in the training set (over-fitting). Here, the model training is optionally regularized

with L2-penalty on all convolutional layer weights and on the fully connected layer weights. L2-penalty also referred to as weight-decay which adds the term $\lambda||\mathbf{W}||_2^2$ to the loss function $L(\mathbf{W}, \{\mathbf{x}_i, y_i\}_{i=1}^n)$. The added term prevents weights from getting too large and thus reduces modeling capacity. Random search was applied to the L2-penalty multiplier λ over a range of 1×10^{-5} to 1×10^{-7}. To increase the robustness of learned features, dropout [Srivastava et al., 2014, Baldi and Sadowski, 2014] was also applied to the fully-connected layer. Dropout is a technique that randomly sets hidden layer units to zero with a given dropout-probability during the training. In the hyperparameter search, we let the dropout-probabilities range from 0 to 0.4. The best performing model found from random search had $\lambda = 0$ and zero dropout-probability. While dropout and L2-penalty tend to be useful for classification there is an intuitive explanation as to why those methods decrease performance in our regression problem. In the case of classification, outputs do not directly depend on the magnitude of the neural network outputs since the outputs are normalized by the sum of all outputs. In regression, the output of the neural network has to exactly match the target value, which can take a wide range of values. If hidden layer units are randomly dropped as in dropout, this estimate may significantly change depending on what units are dropped. Similarly, L2-penalty may prevent weights from adopting the magnitude required by the scale of the targets of the prediction.

For the training and validation of the neural network, all models are implemented in Keras [Chollet et al., 2015] with Tensorflow backend.

3.4 Results

We use a simulated test data sample independent of the training and validation samples in Section 3.3 to test the physics performance of the trained neural networks. The test ν_e CC sample is produced with the same simulation and reconstruction method as the training

sample. Simulation and reconstruction at NOvA are described in Section 3.3.1. The test sample has a regular flux. In order to mimic the real neutrino energy spectrum in the NOvA FD, we apply the neutrino oscillations to each FD MC sample by event weighting. To mimic overall energy resolution of the ν_e oscillation signal while keeping independent from specific CP and mass order choices, oscillation probabilities are calculated from the first-order terms in the full oscillation formula. The values of oscillation parameters are chosen to be $\sin^2\theta_{23} = 0.5$, $\sin^2 2\theta_{23} = 1$, $\Delta m^2_{32} = +2.35 \times 10^{-3}\text{eV}^2$ and $\sin^2 2\theta_{13} = 0.1$.

3.4.1 ν_e-CC neutrino energy

The proposed regression CNN energy estimator is compared with two methods used in previous NOvA ν_e analyses: calorimetric energy estimation and kinematics-based energy estimation. Used as the ν_e CC energy in NOvA's first ν_e oscillation analysis in 2016 [Adamson et al., 2016], the calorimetric energy estimator takes the sum of the calibrated calorimetric energy in each cell for an event and multiplies the sum by a scale factor. The scale factor corrects for the dead material in NOvA detectors and missing energy taken by undetected particles. It is estimated via simulated neutrino events. The kinematics-based energy estimator is based on the method used in NOvA's ν_e analysis in 2017 [Psihas, 2017] (Kinematic Energy). This estimator is based on a quadratic function of the reconstructed electromagnetic and hadronic energy. The electromagnetic energy component is estimated by the sum of calorimetric energies from the electron and photons. The hadronic energy is estimated via the sum of calorimetric energies from hadrons such as pions, kaons, and protons. The electron, photons, and hadrons are identified by a deep-learning based particle identification algorithm called prong CVN [Psihas, 2018]. Parameters of the quadratic function are also determined using simulated data.

The true and reconstructed ν_e CC energy in the FD, weighted by the oscillation probabilities,

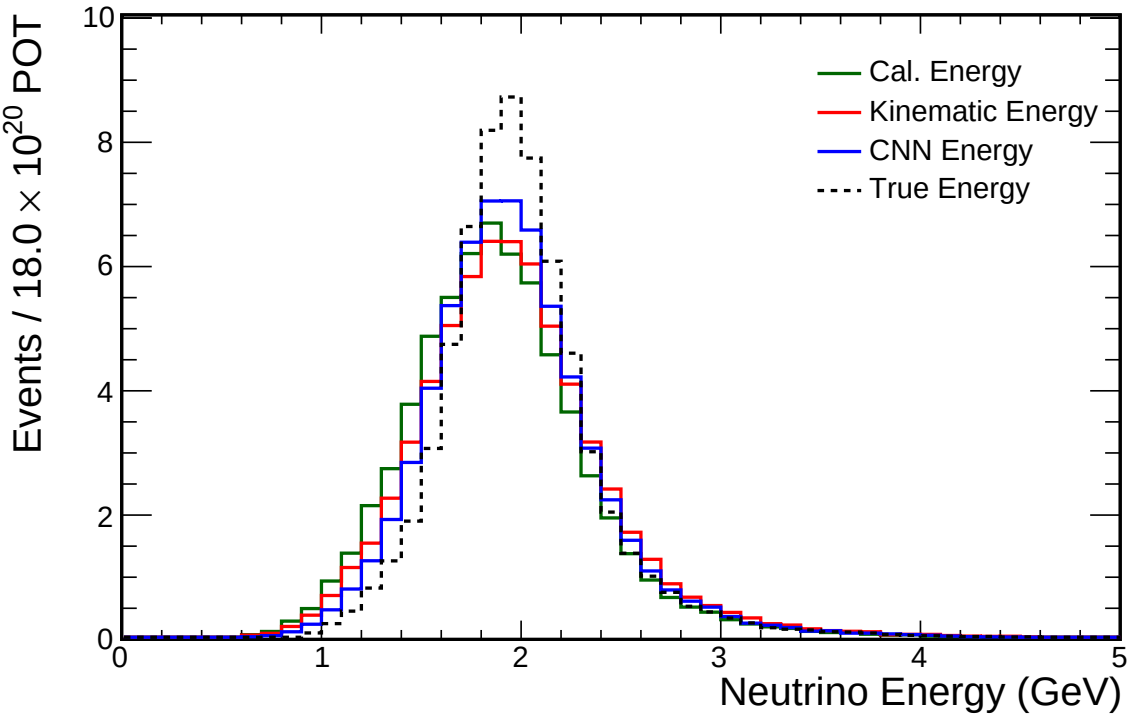


Figure 3.7: Monte Carlo distributions of ν_e CC energy reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green), overlapping with true neutrino energy (dashed). The neutrino oscillations are applied.

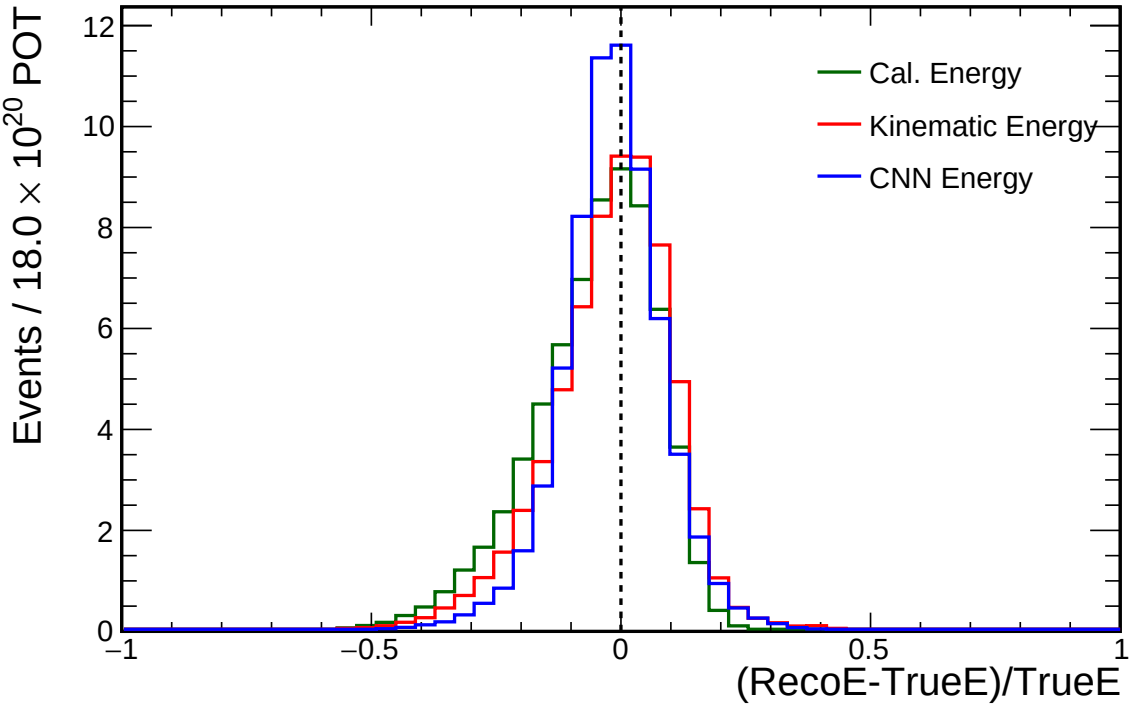


Figure 3.8: Monte Carlo distributions of ratios of differences between reconstructed and true ν_e CC energies to true ν_e CC energy in the calorimetric energy range of 0 to 5 GeV. Neutrino energy is reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

are shown in Figure 3.7. The off-axis spectrum convoluted with the oscillation probability makes the ν_e-CC energy spectrum peak at around 2 GeV.

The overall performance is illustrated in Figure 3.8. Shown are histograms of $(E_{reco} - E_{true})/E_{true}$, the ratio of the difference between reconstructed and true ν_e CC energy over true neutrino energy in the calorimetric energy range of 0 to 5 GeV. The neural network energy is the one with the best resolution. Gaussian fits to $(E_{reco} - E_{true})/E_{true}$ distributions provide relative resolutions of 8.9% (CNN Energy), 10.1% (Kinematic Energy) and 10.2% (Calorimetric Energy), respectively. The relative resolution is defined as the ratio of the standard deviation to the peak value in a Gaussian fit. Relative RMSs (the ratio of RMS to Mean) are 11.1% (CNN Energy), 13.2% (Kinematic Energy) and 13.6% (Calorimetric

Energy).

Figure 3.9 shows means and RMSs of $(E_{reco} - E_{true})/E_{true}$ in each 1-GeV-wide true energy bin. Both Kinematic Energy and CNN energy estimators are determined from the NOvA FD ν_e CC signal sample with oscillated energy spectrum, peaking around 2 GeV. As shown in Figure 3.9 (left), the energy scale of the three estimators shows no significant biases with respect to the true neutrino energy, and the regression CNN has better energy resolutions.

The standard training sample of the described regression CNN ν_e energy estimator uses a flat flux. We also train the regression CNN using the regular flux with the peak around 2 GeV to understand the effect of the training energy spectrum on the linearity of the energy scale. The flat flux sample and the regular flux sample are defined in Section 3.3.3.

Energy scales for neutrino energy based on the flat flux training and regular flux training are shown in Figure 3.10. One can find that the energy scale from the flat flux training has less biases over true neutrino energy. The flat flux training, therefore, represents the preferred training mode to generate the regression CNN for the neutrino energy reconstruction.

Figure 3.11 shows estimator performance by interaction mode. ν_e CC interactions can be classified as quasi-elastic (QE), resonant (RES), and deep-inelastic scattering (DIS) modes. In a QE event, the nucleon (p or n) recoils quasi-elastically from the scattering electron, and the electron, because of its small mass, takes the majority of the incident neutrino energy. Hadronic energy portions and hadron multiplicities vary in these three modes. For the RES mode, the nucleon is excited into baryonic resonances and decays to hadrons, so more neutrino energy is transferred into the hadronic system. In DIS events, the nucleon is smashed into several hadrons, requiring even larger neutrino energy transfer to the hadronic system. Figure 3.11 shows $(E_{reco} - E_{true})/E_{true}$ in these categories individually. The CNN Energy scale shows a better resolution and consistency among the interaction modes.

Systematic uncertainties in the energy reconstruction from the simulation of neutrino inter-

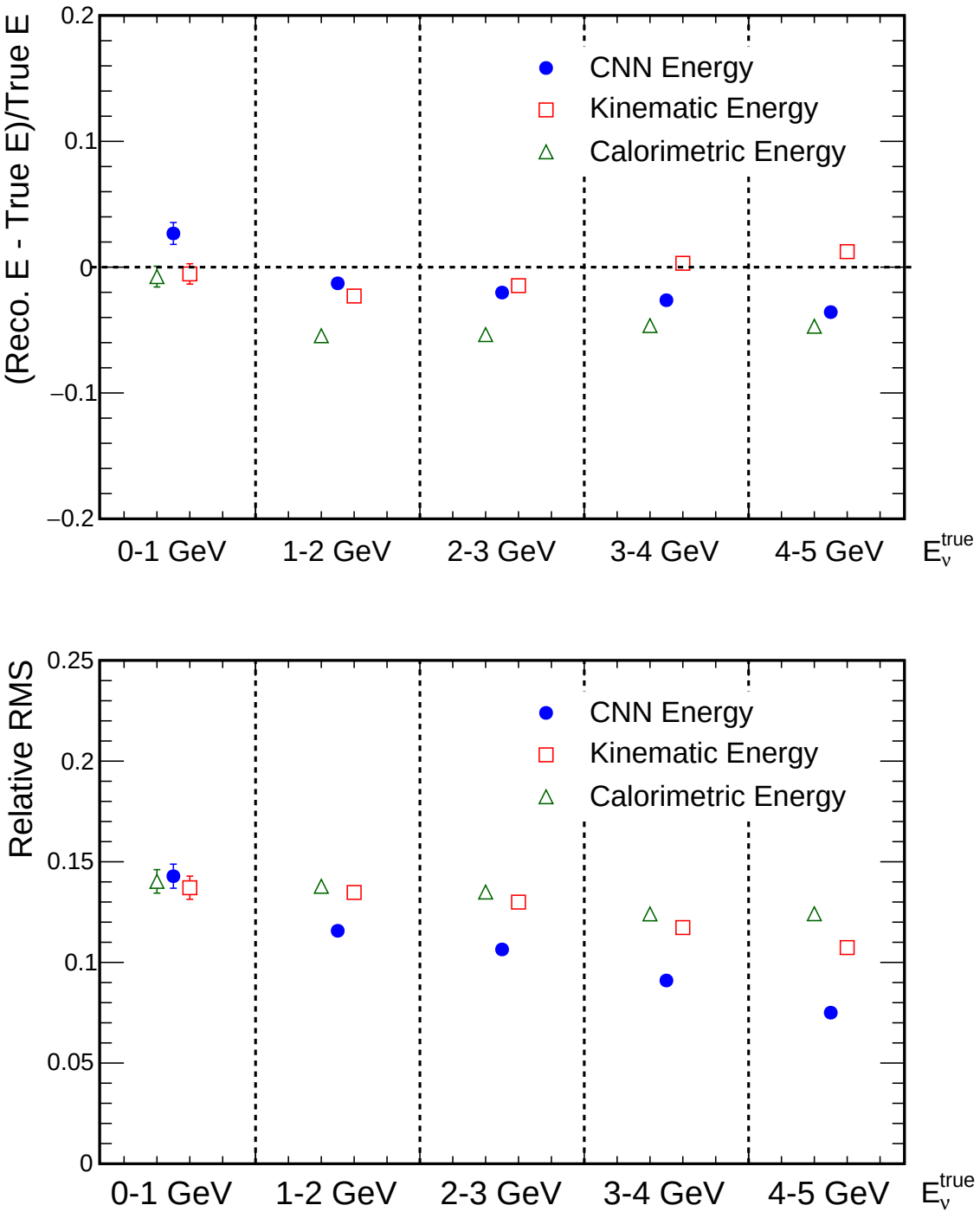


Figure 3.9: Means (top) and relative RMS (bottom, the ratio of RMS to the mean value of energy) of the Monte Carlo distributions of the ratios of differences between reconstructed and true ν_e CC energies to true ν_e CC energy for different true neutrino energy bins ranging from 0 to 5 GeV. Neutrino energy is reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

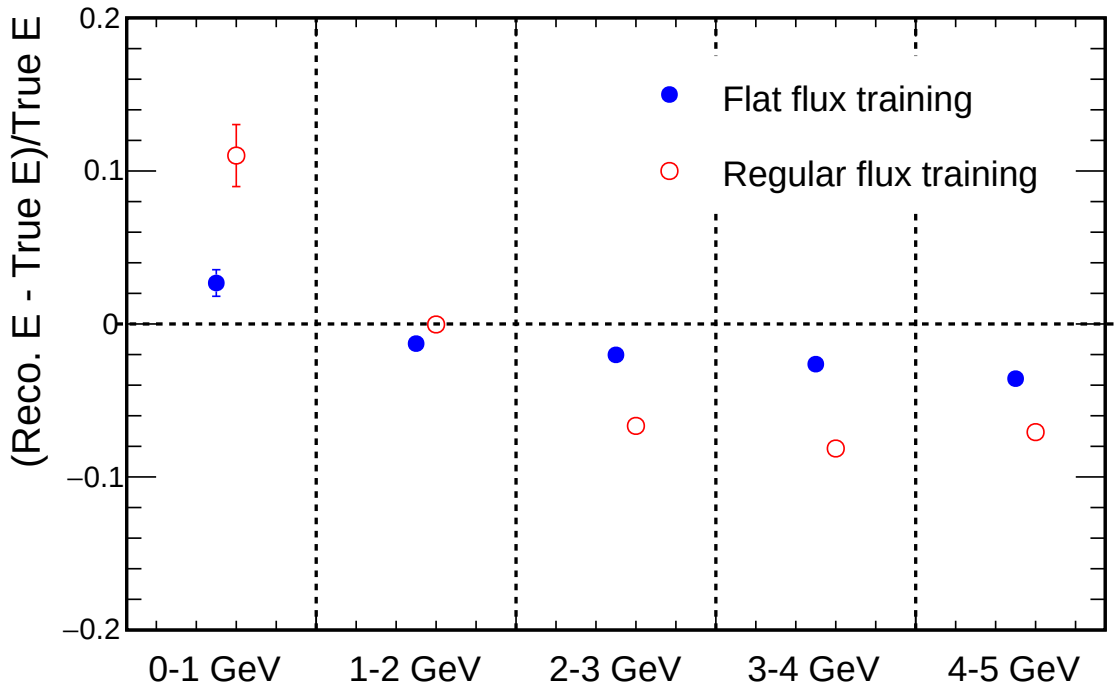


Figure 3.10: Means of the Monte Carlo distributions of ratios of differences between reconstructed and true ν_e CC energies to true ν_e CC energy for different true neutrino energy bins ranging from 0 to 5 GeV. Neutrino energy is reconstructed by CNN trained with flat flux (blue) and regular flux (red), the neutrino oscillations are applied.

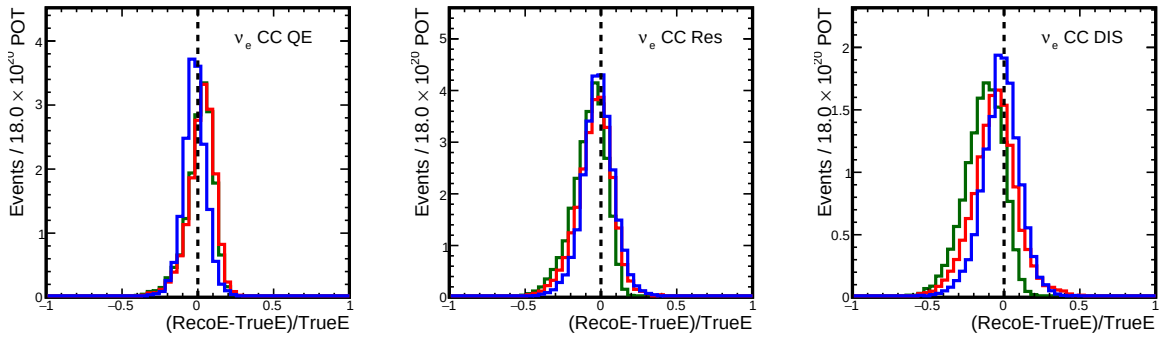


Figure 3.11: Monte Carlo distributions of ratios of the difference between reconstructed and true ν_e CC energy to true neutrino energy for QE, RES, and DIS modes. Neutrino energy is reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

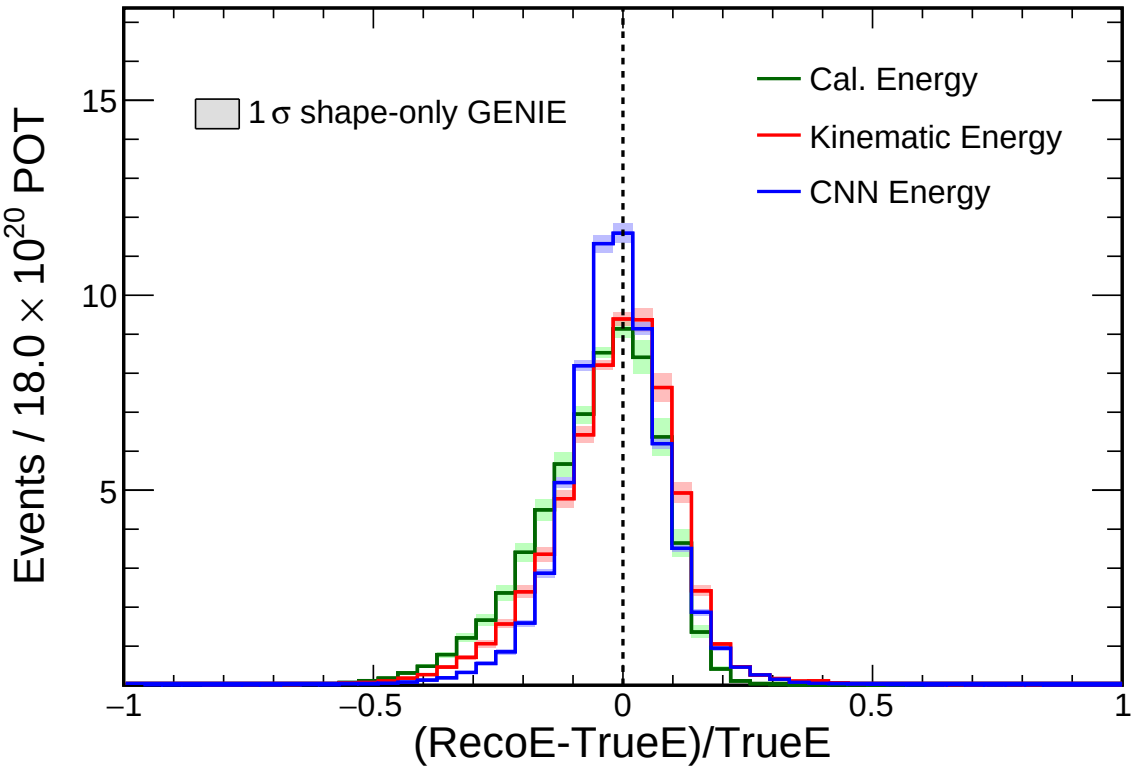


Figure 3.12: Monte Carlo distributions of ratios of differences between reconstructed and true ν_e CC energies to true ν_e CC energy in the calorimetric energy range of 0 to 5 GeV. Error bands represent systematic uncertainties evaluated by GENIE reweighting. Neutrino energy is reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

actions are evaluated by using the reweighting knobs built into GENIE [Andreopoulos et al., 2015]. Each reweighting knob computes a weighting factor that can be applied to MC events to vary normalization and/or shape of a specific type of interaction. In general, these GENIE reweighting knobs deal with systematic uncertainties from modeling of cross-sections, the hadronization, and final state interactions. The reweighting knobs used in this GENIE uncertainty study are similar to NOvA's oscillation analysis Ref [Acero et al., 2018]. We vary each reweighting knob by +1 σ and -1 σ, where the size of the systematic variation σ is the recommendation from the GENIE and NOvA authors, based on surveys of interaction models and existing experimental results. Both the background yield in the signal region before the background correction and the background correction factor determined by the Data-MC difference in the sideband are re-determined in the reweighted background MC.

The overall performance with GENIE systematic shifts is illustrated in Figure 3.12. Shown are histograms of $(E_{reco} - E_{true})/E_{true}$, the ratio of the difference between reconstructed and true ν_e CC energy over true neutrino energy in the calorimetric energy range of 0 to 5 GeV. The systematic errors of $(E_{reco} - E_{true})/E_{true}$ are 0.2% (CNN Energy), 0.6% (Kinematic Energy) and 0.9% (Calorimetric Energy), respectively. The systematic errors of the relative RMSs are 0.3% (CNN Energy), 0.4% (Kinematic Energy) and 0.4% (Calorimetric Energy). Systematic errors of mean and RMS in each energy bin are shown in 3.13. The regression CNN shows smallest systematic uncertainties from the simulation of neutrino interactions.

3.4.2 Electron Shower Energy

The reconstructed electron shower energy given by the regression CNN (CNN Energy) is compared to the sum of the calibrated calorimetric energies (Calorimetric Energy) in electron showers. In a simulated ν_e CC event, the most energetic shower matched to a true electron is chosen as the electron shower sample. The neutrino oscillation weights are applied to the

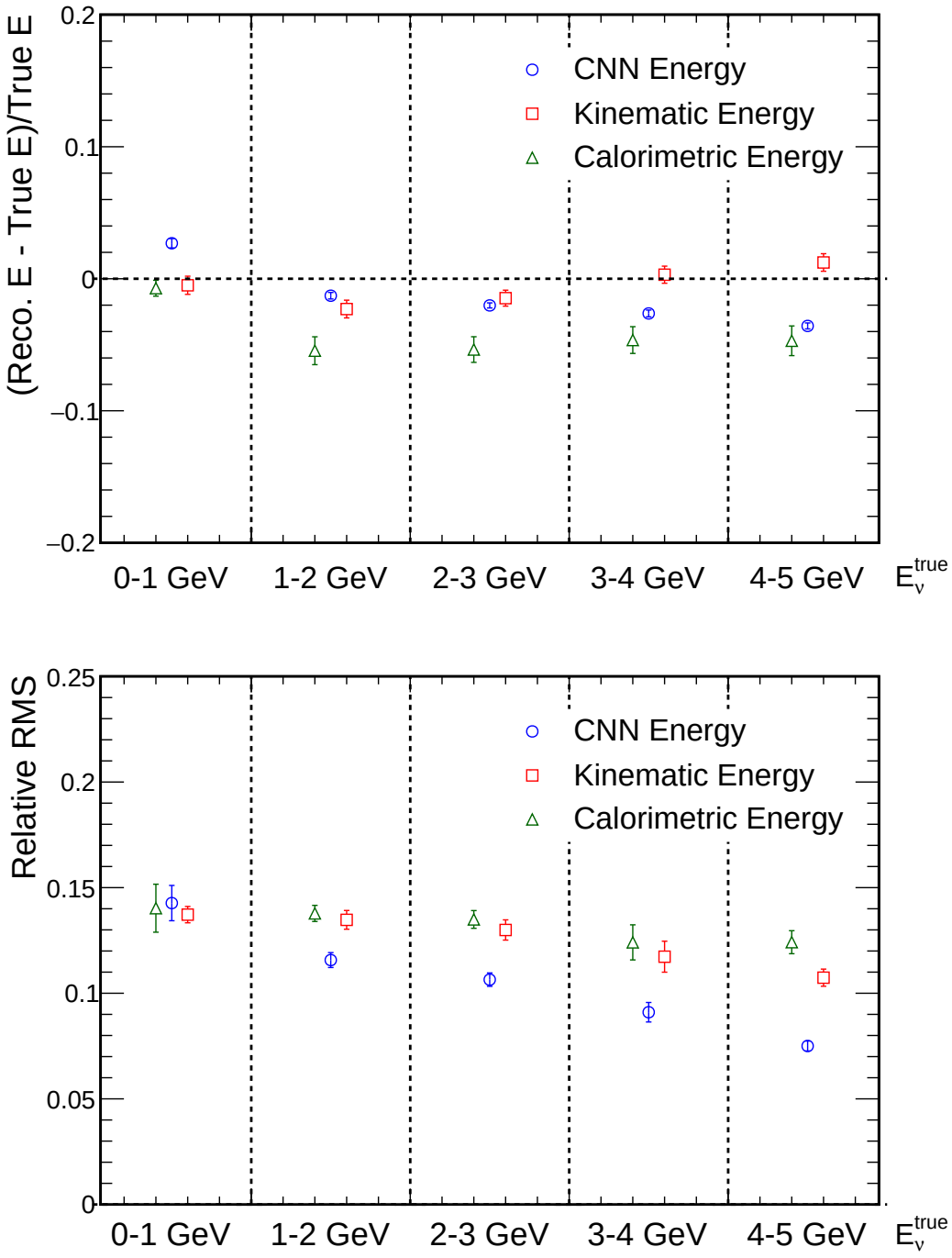


Figure 3.13: Means (top) and relative RMS (bottom) of the Monte Carlo distributions of the ratios of differences between reconstructed and true ν_e CC energies to true ν_e CC energy for different true neutrino energy bins ranging from 0 to 5 GeV. Error bars represent systematic uncertainties evaluated by GENIE reweighting. Neutrino energy is reconstructed by the regression CNN (CNN Energy, blue), particle kinematic information (Kinematic Energy, red), and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

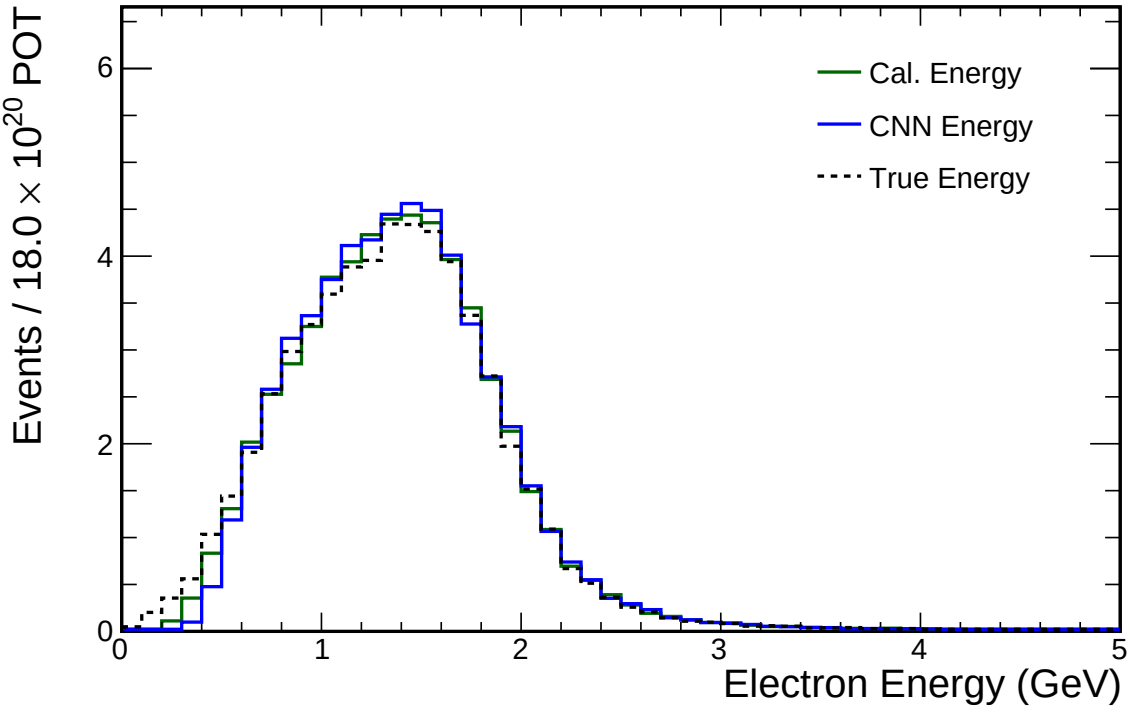


Figure 3.14: Monte Carlo distributions of reconstructed electron shower energy and true electron energy (dashed) in ν_e CC events. The shower energy is reconstructed by CNN (CNN Energy, blue) and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

ν_e CC events. True electron energy and reconstructed electron shower energy distributions in the FD are shown in Figure 3.14. The CNN electron shower energy is closer to the true electron energy than Calorimetric Energy. Overall $(E_{reco} - E_{true})/E_{true}$ for electron showers are shown in Figure 3.15, with relative Gaussian resolutions of 8.2% (CNN Energy) and 9.6% (Calorimetric Energy) and relative RMS of 13.4% (CNN Energy) and 15.2% (Calorimetric Energy) . Histograms of $(E_{reco} - E_{true})/E_{true}$ in different shower calorimetric energy bins and different interaction modes are shown in Figure 3.15 and 3.16. One can find that the CNN Energy has better resolutions. The bias in CNN electron energy at low energies is caused by the small proportion of low energy electrons in the regular flux sample used for training.

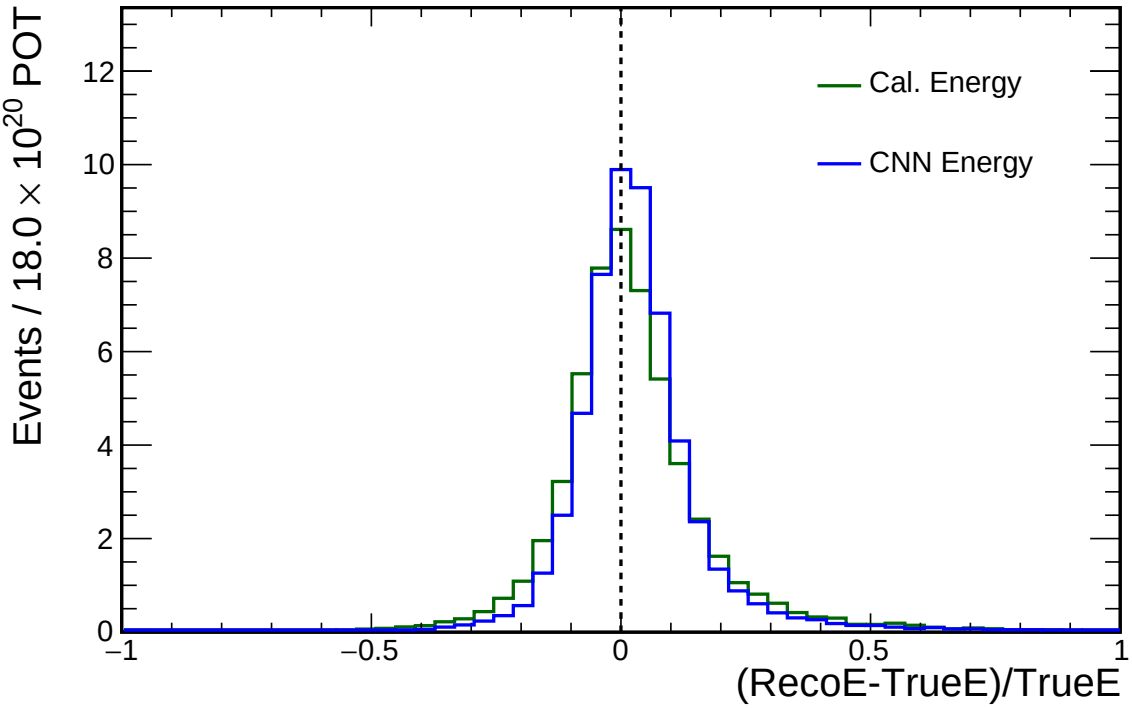


Figure 3.15: Monte Carlo distributions of ratios of reconstructed electron shower energy to true electron energy in ν_e CC events in the shower calorimetric energy range of 0 to 5 GeV. Shower energy is reconstructed by CNN (CNN Energy, blue) and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

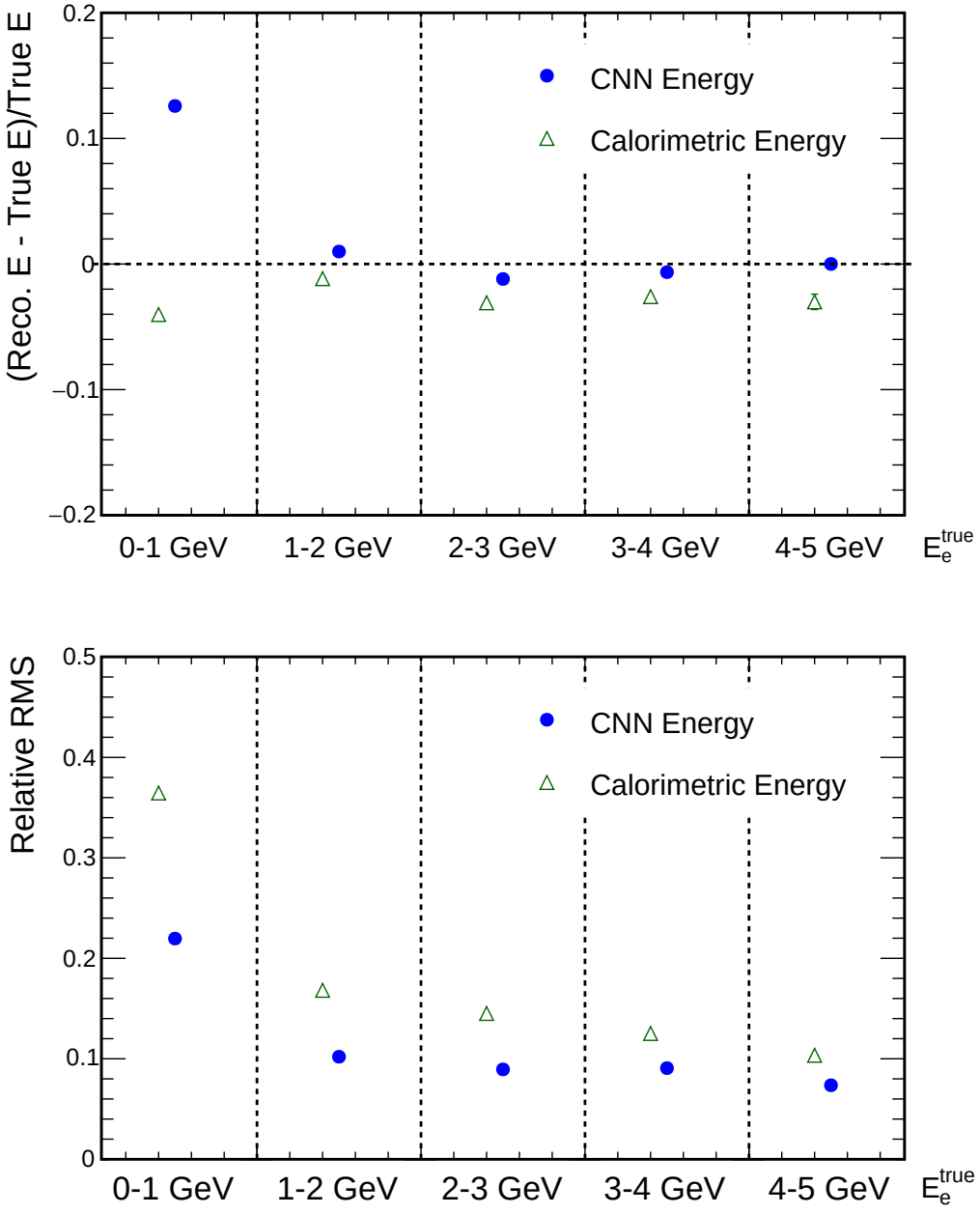


Figure 3.16: Means (top) and RMSs (bottom) of the Monte Carlo distributions of the ratios of reconstructed electron shower energy to true electron energy in ν_e CC events for different shower calorimetric energy bins ranging from 0 to 5 GeV. Shower energy is reconstructed by CNN (CNN Energy, blue) and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

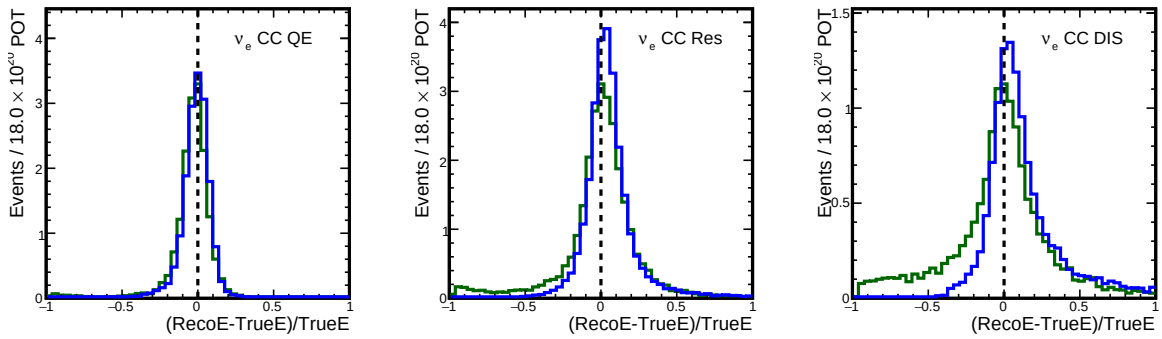


Figure 3.17: Monte Carlo distributions of ratios of reconstructed electron shower energy to true electron energy in ν_e CC events for QE, RES, and DIS modes. Shower energy is reconstructed by CNN (CNN Energy, blue) and summing the calibrated calorimetric energy in each cell (Calorimetric Energy, green). The neutrino oscillations are applied.

3.5 Summary

We developed regression CNNs with direct pixel-level inputs for electron neutrino energy and electron shower energy reconstruction. This is an early effort and proof-of-concept of using CNNs to solve regression problems such as energy reconstruction, vertex reconstruction, and track parameter determination in HEP. It was found that the absolute scaled error provides a useful neural network loss function when the goal is to optimize energy resolution histograms as commonly used in HEP. We also describe the best training parameters found from hyperparameter search for this regression task. This work demonstrates that energy reconstruction tasks can be simplified without elaborate energy scale calibration and model-dependent fits. The performance of the CNN energy reconstruction was verified for different energy bins, different hadronic energy fractions and different interaction modes. In all cases, the regression CNN energy achieves superior performance compared with kinematics based energy reconstruction methods. The regression CNN also shows smaller systematic uncertainties from the simulation of neutrino interactions.

Chapter 4

Sherpa: A Python Hyperparameter Optimization Framework

4.1 Motivation and significance

Hyperparameters are tuning parameters of machine learning models. Hyperparameter optimization refers to the process of choosing optimal hyperparameters for a machine learning model. This optimization is crucial to obtain optimal performance from the machine learning model. Since hyperparameters cannot be directly learned from the training data, their optimization is often a process of trial and error conducted manually by the researcher. There are two problems with the trial and error approach. Firstly, it is time consuming and can take days or even weeks of the researcher's attention. Secondly, it is dependent on the researcher's ability to interpret results and choose good hyperparameter settings. These limitations lead to a large need to automate this process. Sherpa is a software that addresses this need.

Existing hyperparameter optimization software can be divided into bayesian optimization

software, bandit and evolutionary algorithm software, framework specific software, and all-round software. Software that implements bayesian optimization started with SMAC [Hutter et al., 2011], Spearmint [Snoek et al., 2012], and HyperOpt [Bergstra et al., 2013]. More recent software in this regime has been GPyOpt [authors, 2016], RoBo [Klein et al., 2017], DragonFly [Kandasamy et al., 2019], Cornell-MOE [Wu and Frazier, 2016, Wu et al., 2017], and mlrMBO [Bischl et al., 2017]. These software packages have high quality, stand-alone bayesian optimization implementations, often with unique twists. However, most of these do not provide infrastructure for parallel training.

As an alternative to bayesian optimization, multi-armed bandits and evolutionary algorithms have recently become popular. HpBandSter implements Hyperband [**?**] and BOHB [Falkner et al., 2018], Pbt implements Population Based Training [Jaderberg et al., 2017], PyCMA implements CMA-ES [Igel et al., 2006], and TPot [Olson et al., 2016b,a] provides hyperparameter search via genetic programming.

A number of framework specific libraries have also been proposed. Auto-Weka [Kotthoff et al., 2017] and Auto-Sklearn [Feurer et al., 2015] focus on WEKA [Holmes et al., 1994] and Scikit-learn [Pedregosa et al., 2011], respectively. Furthermore, a number of packages have been proposed for the machine learning framework Keras [Chollet et al., 2015]. Hyperas, Auto-Keras [Jin et al., 2019], Talos, Kopt, and HORD each provide hyperparameter optimization specifically for Keras. These libraries make it easy to get started due to their tight integration with the machine learning framework. However, researchers will inevitably run into limitations when a different machine learning framework is needed.

Lastly, a number of implementations aim at being framework agnostic and also support multiple optimization algorithms. Table 4.1 shows a detailed comparison of these "all-round" packages to Sherpa. Note that we excluded Google Vizier [Golovin et al., 2017] and similar frameworks from other cloud computing providers since these are not free to use.

Software	Distributed	Visualizations	Bayesian-Optimization	Evolutionary	Bandit/Early-stopping
Sherpa	Yes	Yes	Yes	Yes	Yes
Advisor	Yes	No	Yes	Yes	Yes
Chocolate	Yes	No	Yes	Yes	No
Test-Tube[Falcon, 2017]	Yes	No	No	No	No
Ray-Tune[Liaw et al., 2018]	Yes	No	No	Yes	Yes
Optuna[Akiba et al., 2019]	Yes	Yes	Yes	No	Yes
BTB [Gustafson, 2018]	No	No	Yes	No	Yes

Table 4.1: Feature comparison of hyperparameter optimization frameworks. *Bayesian optimization*, *evolutionary*, and *bandit/early-stopping* refer to the support of hyperparameter optimization algorithms based on these methods.

Sherpa is already being used in a wide variety of applications such as machine learning methods Sadowski and Baldi [2018], solid state physics Cao et al. [2019], particle physics Baldi et al. [2019], medical image analysis Ritter et al. [2019], and cyber securityLangford et al. [2019]. Due to the fact that the number of machine learning applications is growing rapidly we can expect there to be a growing need for hyperparameter optimization software such as Sherpa.

4.2 Software Description

4.2.1 Hyperparameter Optimization

We begin by laying out the components of a hyperparameter optimization. Consider the training of a machine learning model. A user has a *model* that is being trained with *data*. Before training there are hyperparameters that need to be set. At the end of the training we obtain an *objective* value.

This workflow can be illustrated via the training of a neural network. The *model* is a neural network. The *data* are images that the neural network is trained on. The *hyperparameter setting* is the number of hidden layers of the neural network. The *objective* is the prediction

accuracy on a hold-out dataset obtained at the end of training.

For automated hyperparameter optimization we also need hyperparameter *ranges*, a *results* table, and a hyperparameter optimization *algorithm*. The hyperparameter ranges define what values each hyperparameter is allowed to take. The results store hyperparameter settings and their associated objective value. Finally, the algorithm takes results and ranges and produces a new suggestion for a hyperparameter setting. We refer to this suggestion as a *trial*.

For the neural network example the hyperparameter range might be 1, 2, 3, or 4 hidden layers. We might have previous results that 1 corresponds to 80% accuracy and 3 to 90% accuracy. The algorithm might then produce a new trial with 4 hidden layers. After training the neural network with 4 hidden layers we find it achieves 88% accuracy and add this to the results. Then the next trial is suggested.

4.2.2 Components

We now describe how Sherpa implements the components described in Section 4.2.1. Sherpa implements hyperparameter ranges as `sherpa.Parameter` objects. The *algorithm* is implemented as a `sherpa.algorithms.Algorithm` object. A list of hyperparameter ranges and an algorithm are combined to create a `sherpa.Study` (Figure 4.1). The study stores the *results*. Trials are implemented as `sherpa.Trial` objects.

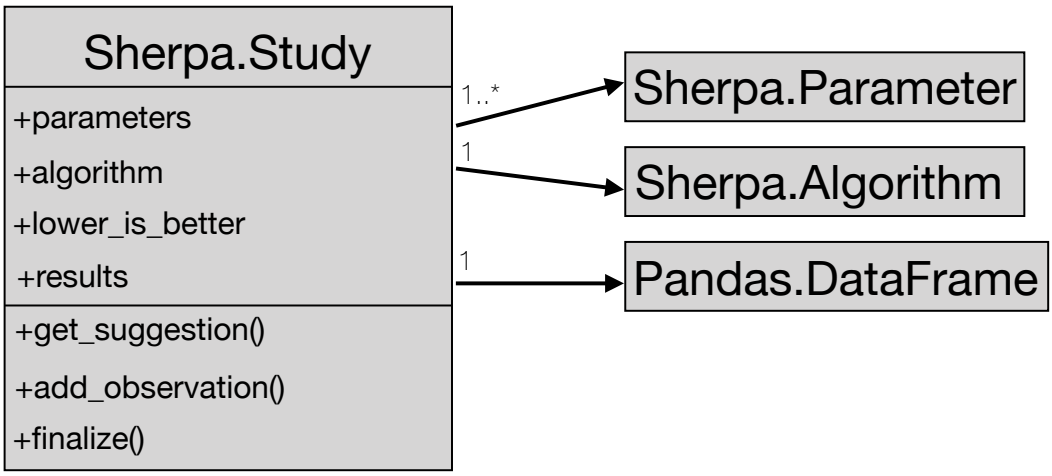


Figure 4.1: Diagram showing Sherpa's Study class.

Sherpa implements two user interfaces. We will refer to the two interfaces as *API mode* and *parallel mode*.

4.2.3 API Mode

In API mode the user interacts with the `Study` object. Given a study s:

1. A new trial of name `t` is obtained by calling `s.get_suggestion()` or by iterating over the study (e.g. `for t in s`).

2. First, `t.parameters` is used to initialize and train a machine learning model. Then `s.add_observation(t, objective=o)` is called to add objective `o` for trial `t`. Invalid observations are automatically excluded from the results.

3. Finally, `s.finalize(t)` informs Sherpa that the model training is finished.

Interacting with the Study class is easy. It also requires minimal setup. The limitation in API mode is that it cannot evaluate trials in parallel.

4.2.4 Parallel Mode

In *parallel-mode* multiple trials can be evaluated in parallel. The user provides two scripts: a *server script* and a *machine learning (ML) script*. The *server script* defines the hyperparameter ranges, the algorithm, the job scheduler, and the command to execute the machine learning script. The optimization starts by calling `sherpa.optimize`.
In the *machine learning script* the user trains the machine learning model given some hyperparameters and adds the resulting objective value to Sherpa. Using a `sherpa.Client` called `c` a trial `t` is obtained by calling `c.get_trial()`. To add observations `c.send_metrics(trial=t, objective=o)` is used.
Internally, `sherpa.optimize` runs a loop that uses the Study class. Figure 4.2 illustrates the *parallel-mode* architecture.

1. The loop submits new trials if resources are available by submitting a job to the scheduler. Furthermore, the new trials are added to a database. From there they can be retrieved by the client.

2. The loop updates results by querying the database for new results.

3. Finally, the loop checks whether jobs have finished. This means resources are free again. In addition, the corresponding trials can be finalized.

If the user's machine learning script does not submit an objective value such as when it crashed, Sherpa continues with the next trial.

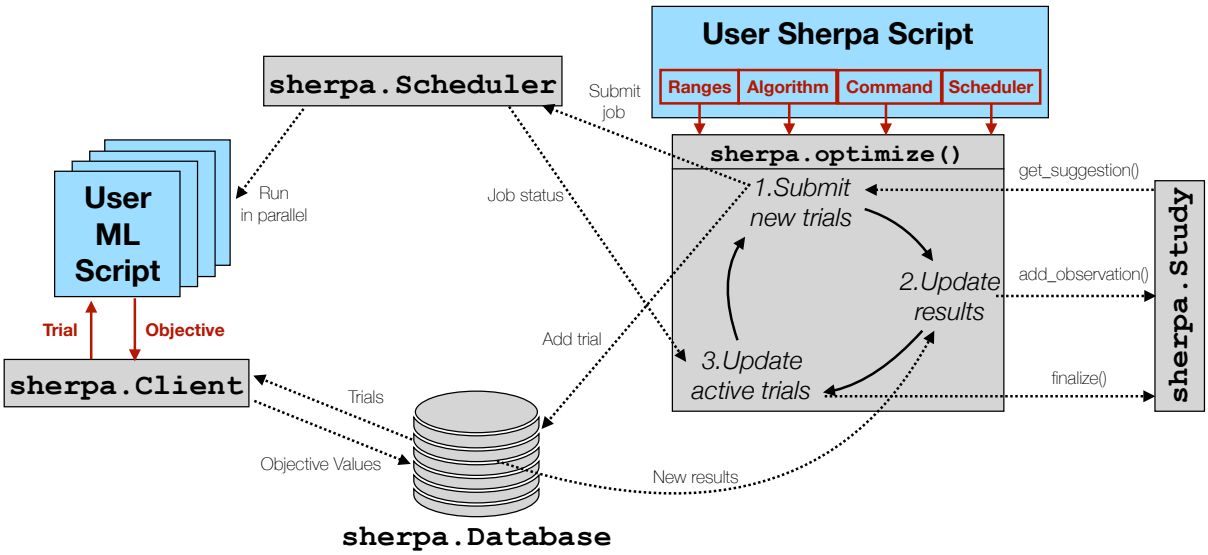


Figure 4.2: Architecture diagram for parallel hyperparameter optimization in Sherpa. The user only interacts with Sherpa via the solid red arrows, everything else happens internally.

4.3 Software Functionalities

4.3.1 Available Hyperparameter Types

Sherpa supports four hyperparameter types:

- `sherpa.Continuous`
- `sherpa.Discrete`
- `sherpa.Choice`
- `sherpa.Ordinal`.

These correspond to a range of floats, a range of integers, an unordered categorical variable, and an ordered categorical variable, respectively. Each parameter has `name` and `range`

arguments. The range expects a list defining lower and upper bound for continuous and discrete variables. For choice and ordinal variables the range expects the categories.

4.3.2 Diversity of Algorithms

Sherpa aims to help researchers at various stages in their model development. For this reason, it provides a choice of hyperparameter tuning algorithms. The following optimization algorithms are currently supported.

- `sherpa.algorithms.RandomSearch`:
  Random Search Bergstra and Bengio [2012] samples hyperparameter settings uniformly from the specified ranges. It is a robust algorithm because it explores the space uniformly. Furthermore, with the dashboard the user can make their own inference on the results.

- `sherpa.algorithms.GridSearch`:
  Grid Search follows a grid over the hyperparameter space and evaluates all combinations. It is useful to systematically explore one or two hyperparameters. It is not recommended for more than two hyperparameters.

- `sherpa.algorithms.bayesian_optimization.GPyOpt`:
  Bayesian optimization is a model-based search. For each trial it picks the most promising hyperparameter setting based on prior results. Sherpa's implementation wraps the package GPyOpt [authors, 2016].

- `sherpa.algorithms.successive_halving.SuccessiveHalving`:
  Asynchronous Successive Halving (ASHA) [Li et al., 2018] is a hyperparameter optimization algorithm based on multi-armed bandits. It allows the efficient exploration

of a large hyperparameter space. This is accomplished by the early stopping of unpromising trials.

- `sherpa.algorithms.PopulationBasedTraining`:
  Population-based Training (PBT) [Jaderberg et al., 2017] is an evolutionary algorithm. The algorithm jointly optimizes a population of models and their hyperparameters. This is achieved by adjusting hyperparameters during training. It is particularly suited for neural network training hyperparameters such as learning rate, weight decay, or batch size.

- `sherpa.algorithms.LocalSearch`:
  Local Search is a heuristic algorithm. It starts with a seed hyperparameter setting. During optimization it randomly perturbs one hyperparameter at a time. If a setting improves on the seed then it becomes the new seed. This algorithm is particularly useful if the user already has a well performing hyperparameter setting.

All implemented algorithms allow parallel evaluation and can be used with all available parameter types. An empirical comparison of the algorithms can be found in the documentation[1].

4.3.3 Accounting for Random Variation

Sherpa can account for variation via the `Repeat` algorithm. The objective value of a model may vary between training runs. Reasons for this can be random initialization or stochastic training. The `Repeat` algorithm runs each hyperparameter setting multiple times. Thus variation can be taken into account when analyzing results.

[1] `https://parameter-sherpa.readthedocs.io/en/latest/algorithms/algorithms.html`

4.3.4 Visualization Dashboard

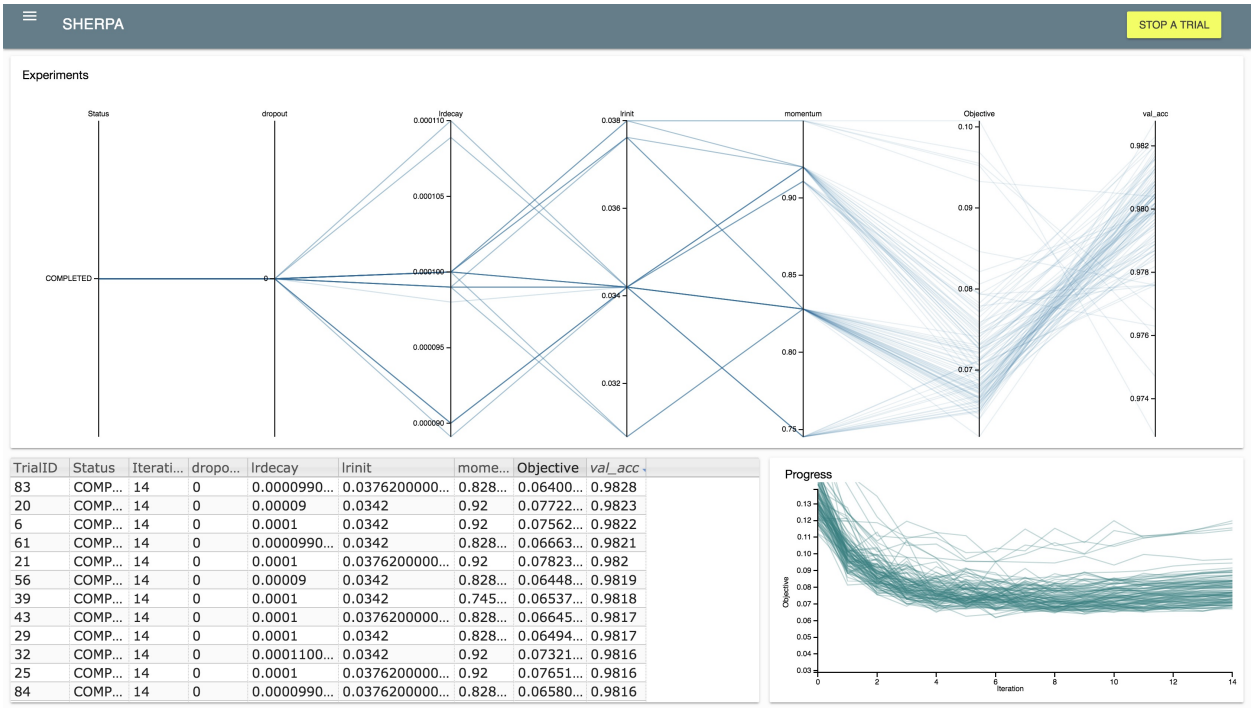
Sherpa provides an interactive web-based dashboard. It allows the user to monitor progress of the hyperparameter optimization in real time. Figure 4.3 shows a screenshot of the dashboard.

At the top of the dashboard is a parallel coordinates plot [Inselberg and Dimsdale, 1987, Hauser et al., 2002]. It allows exploration of relationships between hyperparameter settings and objective values (Figure 4.3 top). Each vertical axis corresponds to a hyperparameter or the objective. The axes can be brushed over to select subsets of trials. The plot is implemented using the D3.js parallel-coordinates library by Chang [2019]. At the bottom right is a line chart. It shows objective values against training iteration (Figure 4.3 bottom right). This chart allows to monitor training progress of each trial. It is also useful to analyze whether a trial's training converged. At the bottom left is a table of all completed trials (Figure 4.3 bottom left). Hovering over trials in the table highlights the corresponding lines in the plots. Finally, the dashboard has a stopping button (Figure 4.3 top right corner). This allows the user to cancel the training for unpromising trials.

The dashboard runs automatically during a hyperparameter optimization. It can be accessed in a web-browser via a link provided by Sherpa. The dashboard is useful to quickly evaluate questions such as:

- Are the selected hyperparameter ranges appropriate?
- Is training unstable for some hyperparameter settings?
- Does a particular hyperparameter have little impact on the performance of the machine learning algorithm?
- Are the best observed hyperparameter settings consistent?

Based on these observations the user can refine the hyperparameter ranges or choose a different algorithm, if appropriate.



TrialID	Status	Iterati...	dropo...	lrdecay	lrinit	mome...	Objective	val_acc
83	COMP...	14	0	0.0000990...	0.0376200000...	0.828...	0.06400...	0.9828
20	COMP...	14	0	0.00009	0.0342	0.92	0.07722...	0.9823
6	COMP...	14	0	0.0001	0.0342	0.92	0.07562...	0.9822
61	COMP...	14	0	0.0000990...	0.0342	0.828...	0.06663...	0.9821
21	COMP...	14	0	0.0001	0.0376200000...	0.92	0.07823...	0.982
56	COMP...	14	0	0.00009	0.0342	0.828...	0.06448...	0.9819
39	COMP...	14	0	0.0001	0.0342	0.745...	0.06537...	0.9818
43	COMP...	14	0	0.0001	0.0376200000...	0.828...	0.06645...	0.9817
29	COMP...	14	0	0.0001	0.0342	0.828...	0.06494...	0.9817
32	COMP...	14	0	0.0001100...	0.0342	0.92	0.07321...	0.9816
25	COMP...	14	0	0.0001	0.0376200000...	0.92	0.07651...	0.9816
84	COMP...	14	0	0.0000990...	0.0376200000...	0.828...	0.06580...	0.9816

Figure 4.3: The dashboard provides a parallel coordinates plot (top) and a table of finished trials (bottom left). Trials in progress are shown via a progress line chart (bottom right). Figure recommended to be viewed as PDF and via zooming in.

4.3.5 Scaling up with a Cluster

In *parallel mode* Sherpa can run parallel evaluations. A job scheduler is responsible for running the user's machine learning script. The following job schedulers are implemented.

- The `LocalScheduler` evaluates parallel trials on the same computation node. This scheduler is useful for running on multiple local CPU cores or GPUs. It has a simple resource handler for GPU allocation (see Figure 4.6 for an example).
- The `SGEScheduler` uses Sun Grid Engine (SGE) [Gentzsch, 2001]. Submission arguments and an environment profile can be specified via arguments to the scheduler.

- The `SLURMScheduler` is based on SLURM [Yoo et al., 2003]. Its interface is similar to the `SGEScheduler`.

Concurrency between workers is handled via MongoDB, a NoSQL database program. Parallel mode expects that MongoDB is installed on the system.

4.4 Illustrative Examples

4.4.1 Handwritten Digits Classification with a Neural Network

The following is an example of a Sherpa hyperparameter optimization. It uses the MNIST handwritten digits dataset Deng [2012]. A Keras neural network is used to classify the digits. The neural network has one hidden layer and a softmax output. The hyperparameters are the learning rate of the Adam Kingma and Ba [2014] optimizer, the number of hidden units, and the hidden layer activation function. The search is first conducted using Sherpa's API mode. After that we show the same example using Sherpa's parallel mode.

API Mode

Figure 4.4 shows the hyperparameter optimization in Sherpa's API mode. The script starts with imports and loading of the MNIST dataset. Next, the hyperparameters *learning_rate*, *num_units*, and *activation* are defined. These refer to the Adam learning rate, number of hidden layer units, and hidden layer activation function, respectively. As optimization algorithm the *GPyOpt* algorithm is chosen. Hyperparameter ranges and algorithm are combined via the *Study*. The `lower_is_better` flag indicates that lower objective values are not better. This is because we will be maximizing the classification accuracy. After that a for-loop iterates over the study. The for-loop yields a trial at each iteration. A Keras model is

instantiated using the hyperparameter settings. The Keras model is iteratively trained and evaluated via an inner for-loop. We add an observation for each iteration and use `finalize` after the training is finished. Note that we pass the loss as context to `add_observation`. The context accepts a dictionary with any additional metrics that the user wants to record. Code to replicate this example is available as a Jupyter notebook[2] and on Google Colab[3]. A video tutorial is also available on YouTube[4]. Tutorials using the Successive Halving and Population Based Training algorithms are also available[5][6].

[2] https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp.ipynb

[3] https://colab.research.google.com/drive/1I19R1GfKPjlgNdHlxJwNC4PitvySsdon

[4] https://youtu.be/-exnF3uv0Ws

[5] https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp_successive_halving.ipynb

[6] https://github.com/sherpa-ai/sherpa/blob/master/examples/keras_mnist_mlp_population_based_training.ipynb

```
import sherpa
import sherpa.algorithms.bayesian_optimization as
   bayesian_optimization
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
epochs = 15
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
# Sherpa setup
parameters = [sherpa.Continuous('learning_rate', [1e-4, 1e-2]),
              sherpa.Discrete('num_units', [32, 128]),
              sherpa.Choice('activation',
                            ['relu', 'tanh', 'sigmoid'])]
algorithm = bayesian_optimization.GPyOpt(max_num_trials=50)
study = sherpa.Study(parameters=parameters,
                     algorithm=algorithm,
                     lower_is_better=False)
for trial in study:
    lr = trial.parameters['learning_rate']
    num_units = trial.parameters['num_units']
    act = trial.parameters['activation']
    # Create model
    model = Sequential([Flatten(input_shape=(28, 28)),
                        Dense(num_units, activation=act),
                        Dense(10, activation='softmax')])
    optimizer = Adam(lr=lr)
    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])
    # Train model
    for i in range(epochs):
        model.fit(x_train, y_train)
        loss, accuracy = model.evaluate(x_test, y_test)
        study.add_observation(trial=trial, iteration=i,
                              objective=accuracy,
                              context={'loss': loss})
    study.finalize(trial=trial)
```

Figure 4.4: An example showing how to tune the hyperparameters of a neural network on the MNIST dataset using Sherpa in API mode.

Parallel Mode

We now show the same hyperparameter optimization using Sherpa's parallel mode. Figure 4.6 (top) shows the server script. First, the hyperparameters and search algorithm are defined. This time we also define a `LocalScheduler` instance. Hyperparameters, algorithm, and scheduler are passed to the `sherpa.optimize` function. We also pass a command "python trial.py". The command indicates how to execute the user's machine learning script. Furthermore, the argument `max_concurrent=2` indicates that two evaluations will be running at a time. Figure 4.6 (bottom) shows the machine learning script. First, we set environment variables for GPU configuration. Next we create a *Client*. To obtain hyperparameters we call the client's `get_trial` method. Furthermore, during training we call the client's `send_metrics` method. This replaces `add_observation` in parallel mode. Also, in parallel mode no *finalize* call is needed.

```
import sherpa
import sherpa.algorithms.bayesian_optimization as
   bayesian_optimization
from sherpa.schedulers import LocalScheduler
params = [sherpa.Continuous('learning_rate', [1e-4, 1e-2]),
          sherpa.Discrete('num_units', [32, 128]),
          sherpa.Choice('activation',
                        ['relu', 'tanh', 'sigmoid'])]
alg = bayesian_optimization.GPyOpt(max_num_trials=50)
sched = LocalScheduler(resources=[0,1])
sherpa.optimize(parameters=params,
                algorithm=alg,
                scheduler=sched,
                lower_is_better=False,
                command='python trial.py',
                max_concurrent=2)
```

Figure 4.5: Server-script for using Sherpa in parallel mode to tune the hyperparameters of a neural network trained on the handwritten digits dataset MNIST.

```
import sherpa, os
GPU_ID = os.environ['SHERPA_RESOURCE']
os.environ['CUDA_VISIBLE_DEVICES'] = GPU_ID
import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.datasets import mnist
from keras.optimizers import Adam
epochs = 15
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
# Sherpa client
client = sherpa.Client()
trial = client.get_trial()
lr = trial.parameters['learning_rate']
num_units = trial.parameters['num_units']
act = trial.parameters['activation']
# Create model
model = Sequential([Flatten(input_shape=(28, 28)),
                    Dense(num_units, activation=act),
                    Dense(10, activation='softmax')])
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(lr=lr),
              metrics=['accuracy'])
# Train model
for i in range(epochs):
    model.fit(x_train, y_train)
    loss, accuracy = model.evaluate(x_test, y_test)
    client.send_metrics(trial=trial, iteration=i,
                        objective=accuracy,
                        context={'loss': loss})
```

Figure 4.6: Trial-script for using Sherpa in parallel mode to tune the hyperparameters of a neural network trained on the handwritten digits dataset MNIST.

4.4.2 Deep learning for Cloud Resolving Models

Introduction

The following illustrates an example of a Sherpa hyperparameter optimization in the field of climate modeling, specifically cloud resolving models (CRM). We apply Sherpa to optimize the deep neural network (DNN) of Rasp et al. [2018].

The input to the model is a 94-dimensional vector. Features include temperature, humidity, meridional wind, surface pressure, incoming solar radiation, sensible heat flux, and latent heat flux. The output of the DNN is a 65-dimensional vector. It is composed of the sum of the CRM and radiative heating rates, the CRM moistening rate, the net radiative fluxes at the top of the atmosphere and surface of the earth, and the observed precipitation.

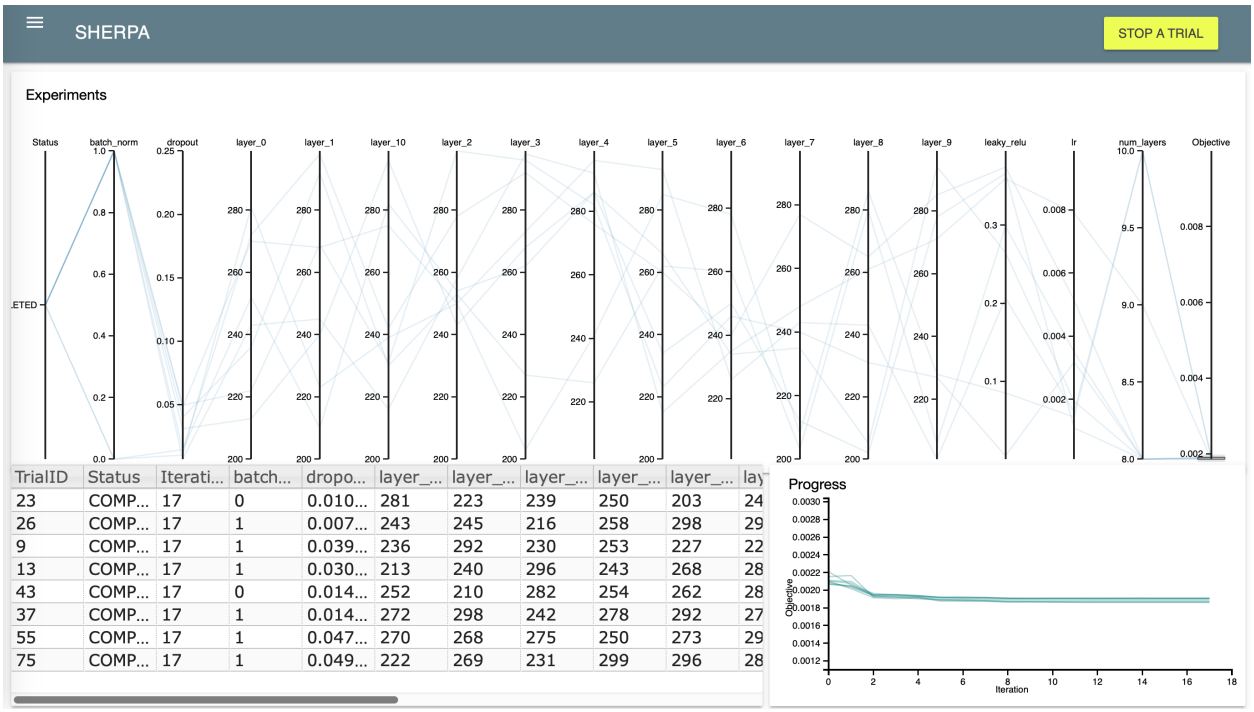
Table 4.2: DNN Hyperparameter Search Space.

Name	Options	Parameter Type
Batch Normalization	[yes, no]	Choice
Dropout	[0, 0.25]	Continuous
Leaky ReLU coefficient	[0 - 0.4]	Continuous
Learning Rate	[0.0001 - 0.01]	Continuous (log)
Nodes per Layer	[200 - 300]	Discrete
Number of layers	[8 - 10]	Discrete

General Hyperparameter Optimization

Initially a random search was conducted on the following hyperparameters: batch normalization [Ioffe and Szegedy, 2015], dropout [Srivastava et al., 2014, Baldi and Sadowski, 2013], Leaky ReLU coefficient [Agostinelli et al., 2014], learning rate, nodes per hidden layer, number of hidden layers. The parameter ranges were chosen to encompass the parameters spec-

ified in Rasp et al. [2018] and are shown in Table 4.2. On the dashboard we select the best trials (Figure 4.7) and identify that the best performing configurations have low dropout, leaky ReLU coefficients mostly around 0.3 or larger, and learning rates mostly near 0.002. The majority of good models have 8 layers and batch normalization. However, the number of units does not seem to have a large impact.



TrialID	Status	Iterati...	batch...	dropo...	layer_...	layer_...	layer_...	layer_...	layer_...	lay
23	COMP...	17	0	0.010...	281	223	239	250	203	24
26	COMP...	17	1	0.007...	243	245	216	258	298	29
9	COMP...	17	1	0.039...	236	292	230	253	227	22
13	COMP...	17	1	0.030...	213	240	296	243	268	28
43	COMP...	17	0	0.014...	252	210	282	254	262	28
37	COMP...	17	1	0.014...	272	298	242	278	292	27
55	COMP...	17	1	0.047...	270	268	275	250	273	29
75	COMP...	17	1	0.049...	222	269	231	299	296	28

Figure 4.7: Screenshot of the dashboard at the end of the initial random search. The 8 best trials were selected by brushing of the *Objective* axis in the parallel coordinates plot.

Optimization of the Learning Rate Schedule

An additional search was conducted to fine-tune the DNN training hyperparameters. Specifically, the initial learning rate and the learning rate decay were optimized. The range of initial learning rate values was $\pm 10^{-4}$ of the best value from Section 4.4.2. The range of learning rate decay factors was 0.5 to 1. The learning rate gets multiplied by this factor after every epoch to produce a new learning rate. In comparison, the model in Rasp et al. [2018] uses a decay factor of approximately 0.58. The remaining hyperparameters were set to the best

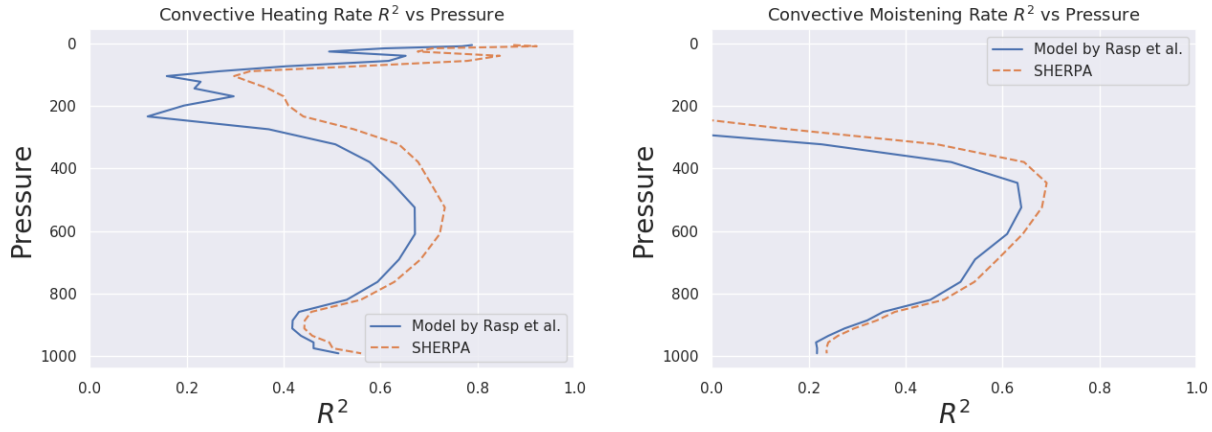
configuration from Section 4.4.2. A total of 50 trials were evaluated via random search. The best learning rate was found to be 0.001196. The best decay value was found as 0.843784. The overall optimal hyperparameter setting is shown in Table 4.3.

Table 4.3: Best hyperparameter configuration found by Sherpa.

Batch Normalization	No
Dropout	0.0
Leaky ReLU coefficient	0.3957
Learning Rate	0.001301
Learning Rate Decay	0.843784
Nodes per Layer	[299, 269, 248, 293, 251, 281, 258, 277, 209, 270]
Number of layers	10

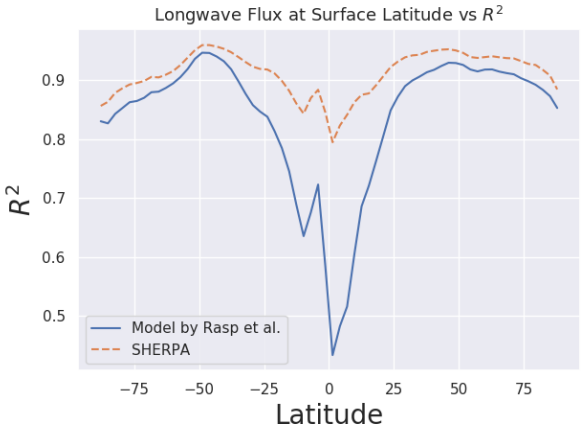
Results

We compare the model found by Sherpa to the model from Rasp et al. [2018] via R^2 plots (Figure 4.8). The R^2 plots show the coefficient of determination at different pressures and latitudes. We find that the Sherpa model consistently outperforms the comparison model. In particular, it is able to perform for latitudes for which the prior model fails. Figure 4.8f shows that the Sherpa model's loss reduces further after the Rasp et al. [2018] model has converged. This is the result of the learning rate fine-tuning from Section 4.4.2.

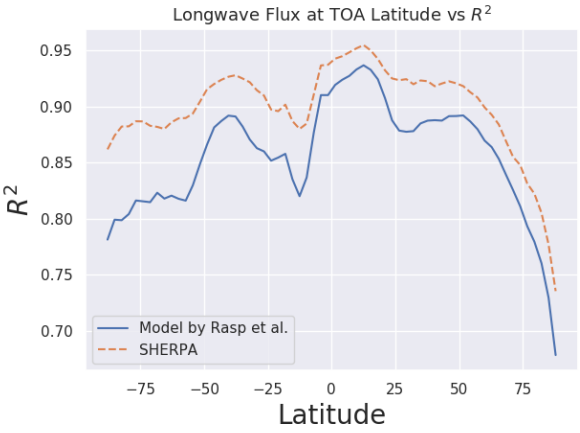


(a)

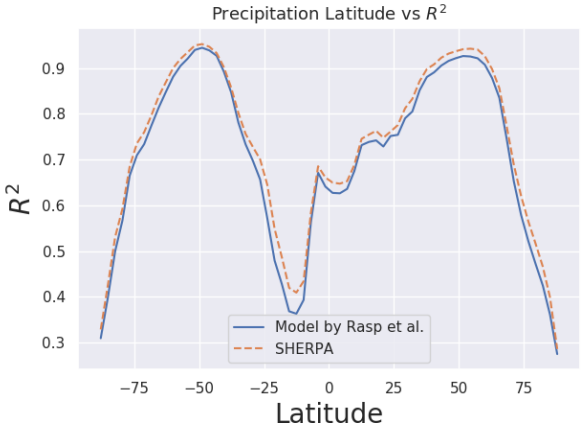
(b)



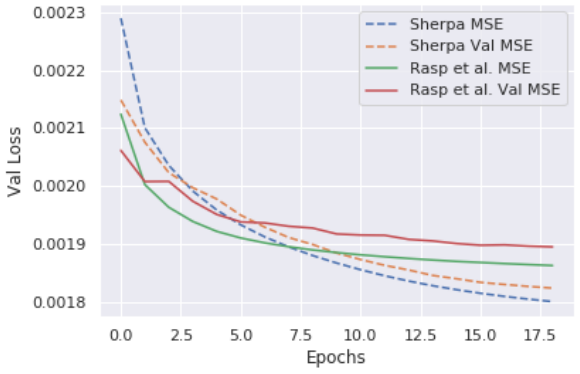
(c)



(d)



(e)



(f)

Figure 4.8: Case study results for an optimized deep neural network applied to cloud resolving models. Figures 4.8a and 4.8b show the coefficient of determination R^2 vs. pressure for convective heating rate and convective moistening rate, respectively. Figures 4.8c, 4.8d, and 4.8e show R^2 values against latitude, and 4.8f shows loss trajectories. All figures compare the optimized Sherpa model against the model developed by Rasp et al. [2018].

4.5 Impact

Machine learning is used to ever larger extends in the scientific community. Nearly every machine learning application can benefit from hyperparameter optimization. The issue is that researchers often do not have a practical tool at hand. Therefore, they usually resort to manually tuning parameters. Sherpa aims to be this tool. Its goal is to require minimal learning from the user to get started. It also aims to support the user as their needs for parallel evaluation or exotic optimization algorithms grow. As shown by references in Section 4.1, Sherpa is already being used by researchers to achieve improvements in a variety of domains. In addition to that, the software has been downloaded more than 6000 times from the PyPi Python package manager[7]. It also has over 160 stars on the software hosting website GitHub. A GitHub star means that another user has added the software to a personal list for later reference.

4.6 Conclusions

Sherpa is a flexible open-source software for robust hyperparameter optimization of machine learning models. It provides the user with several interchangeable hyperparameter optimization algorithms, each of which may be useful at different stages of model development. Its interactive dashboard allows the user to monitor and analyze the results of multiple hyperparameter optimization runs in real-time. It also allows the user to see patterns in the performance of hyperparameters to judge the robustness of individual settings. Sherpa can be used on a laptop or in a distributed fashion on a cluster. In summary, rather than a black-box that spits out one hyperparameter setting, Sherpa provides the tools that a researcher needs when doing hyperparameter exploration and optimization for the development of machine learning models.

[7] `https://pepy.tech/project/parameter-sherpa`

Chapter 5

Reproducible Hyperparameter Optimization

5.1 Introduction

In machine learning, hyperparameters are distinct from the model parameters and need to be set before model training. Hyperparameters are often unintuitive yet they can have large impact on model performance. Unlike model parameters, hyperparameters are usually non-differentiable. Therefore, the optimization of hyperparameters typically requires an empirical search evaluating model performance on training and validation samples. Optimization can be done manually with the researcher picking candidate values based on their intuition and experience or automatically with an algorithm suggesting candidates [Goodfellow et al., 2016, Feurer and Hutter, 2018].

Training algorithms are often subject to randomness. Examples of this are the random initialization of model parameters before optimization, shuffling of the training data before using stochastic gradient descent, stochastic regularization such as dropout [Srivastava et al.,

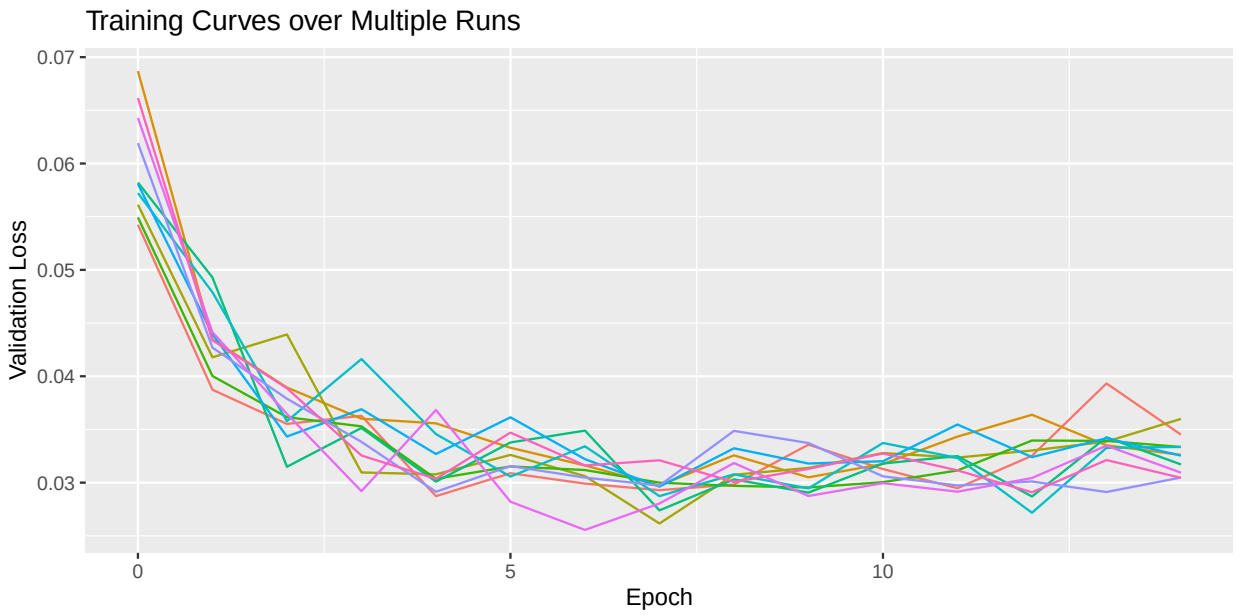


Figure 5.1: Validation loss against epochs for different training runs with the same hyperparameters. Different colors represent different runs.

2014, Baldi and Sadowski, 2013], and data augmentation through random transformations. The implication of stochasticity in training algorithms is that two equal algorithms may differ in their out-of-sample prediction error when trained with different seeds for the random number generator. As an example, Figure 5.1 shows training runs for a neural network with the same hyperparameters. Clearly, hyperparameter optimization may be impacted by variation between runs of the same hyperparameters. The end result of failing to account for such variation is lower reproducibility of experimental results and decreased generalizability of predictive performance at a specified hyperparameter choice.

Machine learning faces a reproducibility crisis [Hutson, 2018, Lipton and Steinhardt, 2018, Sculley et al., 2018]. Reproducibility issues appear in two ways. First, the reproducibility of experimental results themselves are crucial. In particular, Henderson et al. [2018] have shown that oftentimes it may be impossible to reproduce presented results. Hyperparameter optimization plays an important role in machine learning research. As part of the scientific process we would like the hyperparameter optimization itself to be reproducible, that is to

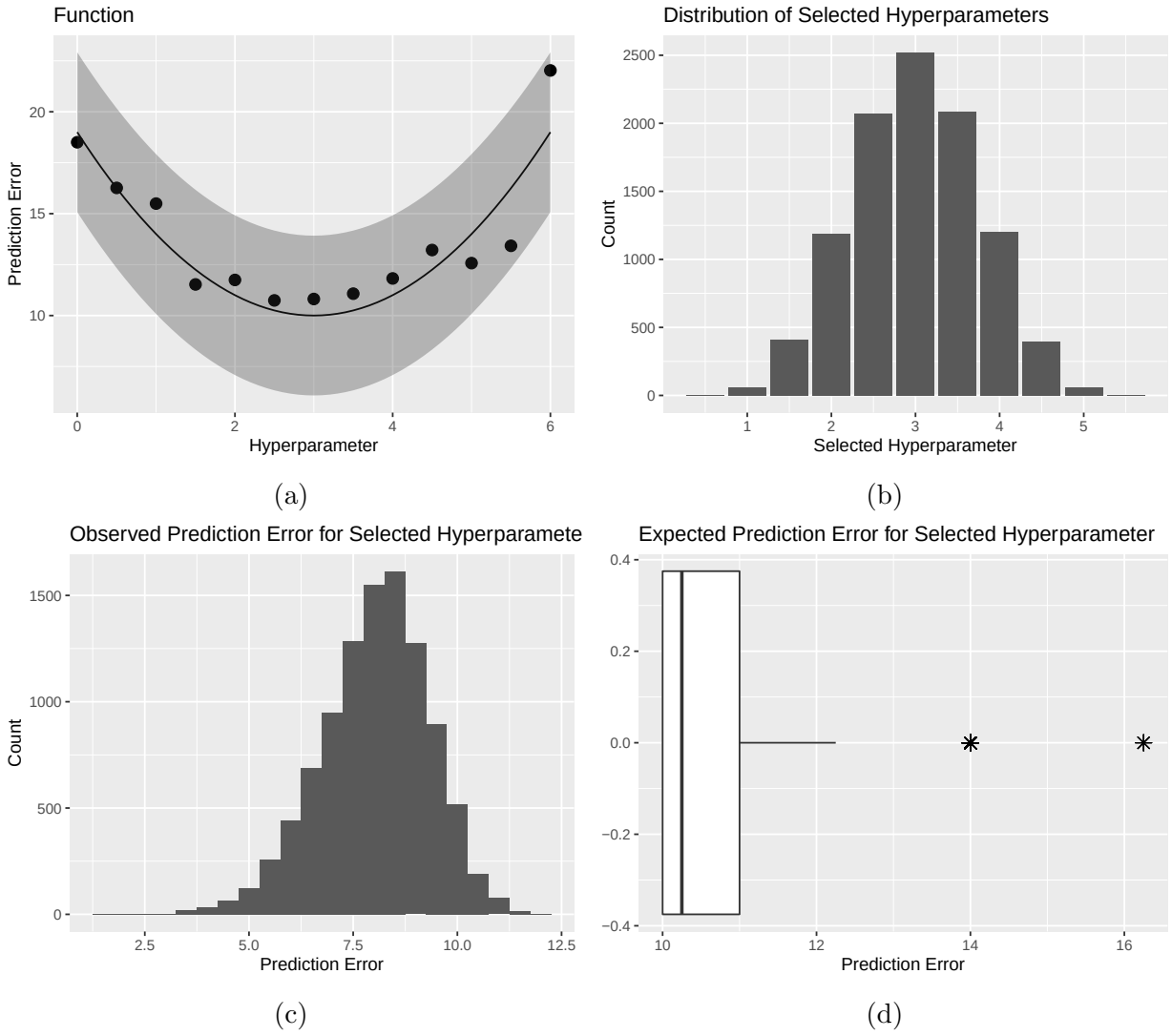


Figure 5.2: Illustration of the varying results obtained from hyperparameter optimization when the prediction error is a noisy function of the hyperparameter value. Panel (a) shows a simple example of expected prediction error as a function of the hyperparameter (solid line). The shaded area represents two standard deviations from the mean. The function is evaluated at hyperparameter values from 0 to 6 in steps of 0.5. The dots represent one such evaluation. In this case the selected hyperparameter would be 2.5 since it corresponds to the minimum observed prediction error. Panel (b) shows the distribution of selected hyperparameters over 10000 such evaluations. Panel (c) shows the observed prediction errors for the selected hyperparameters. Panel (d) shows the expected prediction errors corresponding to the selected hyperparameters.

produce similar results every time. This is not the case when training is non-deterministic. Figure 5.2 demonstrates this via a toy-example that shows a noisy U-shaped curve of prediction error against a hyperparameter as often observed in machine learning. Different draws from the curve result in different observed optimal hyperparameters. Figure 5.2(b) shows the distribution of selected hyperparameters over 10000 hyperparameter searches. Ideally this distribution would be peaked at the optimal value, here 3. Figure 5.2(c) shows the observed prediction errors for the selected hyperparameters. Lastly, Figure 5.2(d) shows the expected prediction errors corresponding to the selected hyperparameters. For an ideal hyperparameter search this box plot would be flat at 10, the best obtainable value.

The second issue that arises associated to reproducibility is related to the comparison of different methods. Failure to reproduce improvements on baselines has been shown in a number of domains such as sequence learning [Greff et al., 2017, Melis et al., 2017] , reinforcement learning [Islam et al., 2017, Nagarajan et al., 2018, Henderson et al., 2018], and generative adversarial networks [Lucic et al., 2018]. It is crucial to compare two machine learning methods at their optimal hyperparameters. However, if non-determinism is not taken into account during hyperparameter optimization, the overall comparison between methods may be invalid. Consider for example a comparison of machine learning algorithm A vs. machine learning algorithm B that have equal expected prediction errors across runs at their optimal hyperparameter setting. During a hyperparameter optimization that does not account for variation between runs we may select hyperparameters for A and B such that their resulting expected prediction error across runs is significantly different. We may then declare a difference in the two methods, despite the fact that at their optimal hyperparameters they would have been equal.

Generalization of hyperparameter settings is often assumed in practice. This is partly attributable to the fact that hyperparameter optimization is often computationally expensive. If the computational resources for hyperparameter optimization are unavailable, it is com-

mon to use hyperparameter settings provided in prior publications. There are two popular variations of this: re-using hyperparameters associated with a specific algorithm and re-using a complete model with all of its hyperparameters. As an example of re-using default parameters of an algorithm, consider adaptive learning rate algorithms for deep learning. Chollet et al. [2018] recommend for their widely used machine learning software Keras not to change the default values of many of these algorithms [Keras, 2019]. It is also common to re-use a complete model with all of its associated hyperparameters. The Stanford class Convolutional Neural Networks for Visual Recognition [Karpathy, 2017] recommends using the hyperparameters of the current state-of-art as baseline when solving computer vision problems. This practice can also be seen in research by Jean et al. [2016] and Lakhani et al. [2018] who use neural network hyperparameters from other publications for applications in social science and medical imaging, respectively.

Failure to take non-deterministic training into account during hyperparameter optimization may result in non-optimal hyperparameters. While this non-optimality may have small consequences when only applied to one research study, the publication and reuse of such hyperparameters can mean that such non-optimality may proliferate to other studies.

Hyperparameter optimization for machine learning models is a widely researched topic. In particular, a number of methods that suggest hyperparameter settings have been proposed. The most popular methods are Bayesian optimization [Shahriari et al., 2015], neural architecture search [Zoph and Le, 2016], and evolutionary algorithms [Real et al., 2017]. In Bayesian optimization, the relationship between hyperparameters and prediction error is modelled by a Gaussian Process or a Random Forest. An acquisition function is used to determine the next point to explore based on the model. Neural architecture search models the relationship between hyperparameters and prediction error as a reinforcement learning problem. Lastly, evolutionary algorithms rely on mutation operators to turn an initial hyperparameter setting into an optimal one over many trials. All of the above methods are

concerned with finding the optimal hyperparameter setting. We consider the problem discussed orthogonal to the problem of hyperparameter suggestion. In fact it is straightforward to incorporate such methodology with what we propose and the combination with different suggestion algorithms is an interesting future extension.

Another important research direction in hyperparameter search is to increase efficiency through early stopping of unpromising trials. Swersky et al. [2014], Golovin et al. [2017], and Klein et al. [2016b] model the prediction error after full training given the prediction error after training on a subset of the data using Bayesian non-parametric modeling. Krueger et al. [2015] evaluates all candidate hyperparameter settings on increasingly larger subsets of the data. Sequential testing is performed on whether a configuration is a top or flop configuration. Lastly, Li et al. [2016] rank trials by their performance on a limited budget of resources and choose a certain number of configurations from the top to continue. The focus of all of these methods is to speed up regular hyperparameter search. None of these methods focus on quantification of uncertainty within hyperparameter settings and are therefore again orthogonal to what is discussed here.

The current manuscript focuses on hyperparameter optimization that accounts for stochasticity in model training. Our proposed approach is based on the repetition of hyperparameter settings over multiple trials in order to obtain estimates of the expected prediction error for a hyperparameter setting. We formalize this procedure and propose a hypothesis test to obtain a set of hyperparameter settings that is not distinguishable in performance. Using the resulting equivalence class, we further propose to use a sequential testing framework to reduce the number of repeated trials expended on unpromising hyperparameter configurations while maintaining high sensitivity and specificity for detecting optimal settings.

5.2 Methods

5.2.1 Hyperparameter Optimization

Assume a prediction problem in which the available data set is split into $\mathcal{D}^{\text{train}}$, $\mathcal{D}^{\text{valid}}$, and $\mathcal{D}^{\text{test}}$ and considered fixed thereafter. Training algorithm $\mathcal{A}$ is used to obtain a function f by minimizing the in-sample prediction error $\mathcal{A}(\mathcal{D}^{\text{train}}) = f$. The function f represents the prediction model. For example, f may be a trained neural network. $\mathcal{A}$ typically involves optimization of f's parameters θ. Commonly $\mathcal{A}$ also has parameters λ called hyperparameters, so $\mathcal{A}_\lambda(\mathcal{D}^{\text{train}}) = f_\lambda$.

To choose λ one seeks to minimize the expected out of sample prediction error $\text{EPE}(f_\lambda) = \mathbb{E}_{X,Y}[\mathcal{L}(Y; f_\lambda(X))]$:

$$\lambda^* = \arg\min_{\lambda \in \Lambda} \text{EPE}(f_\lambda). \tag{5.1}$$

Here, $\mathcal{L}$ is a loss function defining the prediction error. Since the expected prediction error is unavailable it can be estimated with the error on the hold out validation data set $\mathcal{D}^{\text{valid}}$. In addition, the hyperparameter space is approximated by a discrete set of values, $\tilde{\Lambda} = \{\lambda_1, \ldots, \lambda_K\}$, and hyperparameter optimization proceeds by minimization over $\tilde{\Lambda}$:

$$\lambda^* \approx \arg\min_{\lambda \in \tilde{\Lambda}} \Psi(f_\lambda) \tag{5.2}$$

where the empirical loss is

$$\Psi(f_\lambda) = \frac{1}{\mid \mathcal{D}^{\text{valid}} \mid} \sum_{(x,y) \in \mathcal{D}^{\text{valid}}} \mathcal{L}(y, f_\lambda(x)). \tag{5.3}$$

As previously mentioned many methods incorporate stochasticity into $\mathcal{A}$. In this case pre-

dictions resulting from f_λ will be random depending on $\mathcal{A}_\lambda$ and $\mathcal{D}^{\text{train}}$. This means $\Psi(f_\lambda)$ are random variables, despite a fixed $\mathcal{D}^{\text{valid}}$, and hence

$$\arg\min_{\lambda \in \tilde{\Lambda}} \Psi(f_\lambda) \neq \arg\min_{\lambda \in \tilde{\Lambda}} \mathbb{E}_{\mathcal{A}_\lambda}[\Psi(f_\lambda)] \tag{5.4}$$

for a given $\mathcal{D}^{\text{train}}$ and $\mathcal{D}^{\text{valid}}$, in general. We refer to $\Psi(f_\lambda)$ as $\Psi(\lambda)$ hereafter.

5.2.2 Hyperparameter Optimization for Non-deterministic Training

Objective and Testing Procedure

We modify the objective of the hyperparameter optimization to find the hyperparameter setting λ^* that minimizes the average prediction error across model training runs. In particular, we aim to identify a subset of the explored hyperparameter settings such that no significant difference can be found in average performance for any hyperparameter configuration in the subset. More precisely, given hyperparameter settings $\tilde{\Lambda} = \{\lambda_1, \ldots, \lambda_K\}$ we collect n_λ evaluations of $\mathcal{A}_{\lambda_i}(\mathcal{D}^{\text{train}})$ for each λ_i and calculate $\hat{\tau}_{\lambda_i} = \frac{1}{n_\lambda} \sum_{j=1}^{n_\lambda} \Psi^j(\lambda_i)$ the mean observed validation loss for each hyperparameter setting across n_λ training runs. We order $\tilde{\Lambda}$ in increasing order by $\hat{\tau}_{\lambda_i}$ and test the hypotheses:

$$H_{0,k} : \tau_{\lambda_1} = \ldots = \tau_{\lambda_k} \quad \text{vs.} \quad H_1 : \text{at least one different.} \tag{5.5}$$

We start with $k = K$ that is all hyperparameter settings in $\tilde{\Lambda}$. If $H_{0,K}$ is rejected we reduce k and repeat the test using the testing algorithm described below. Testing is stopped once we have found $\tilde{k}$, the largest k such that we fail to reject $H_{0,k}$. The subset $\{\lambda_1, \ldots, \lambda_{\tilde{k}}\}$ represents an equivalence class of hyperparameter settings that cannot be distinguished in their average performance. To choose an estimate of λ^* from $\{\lambda_1, \ldots, \lambda_{\tilde{k}}\}$ one can either

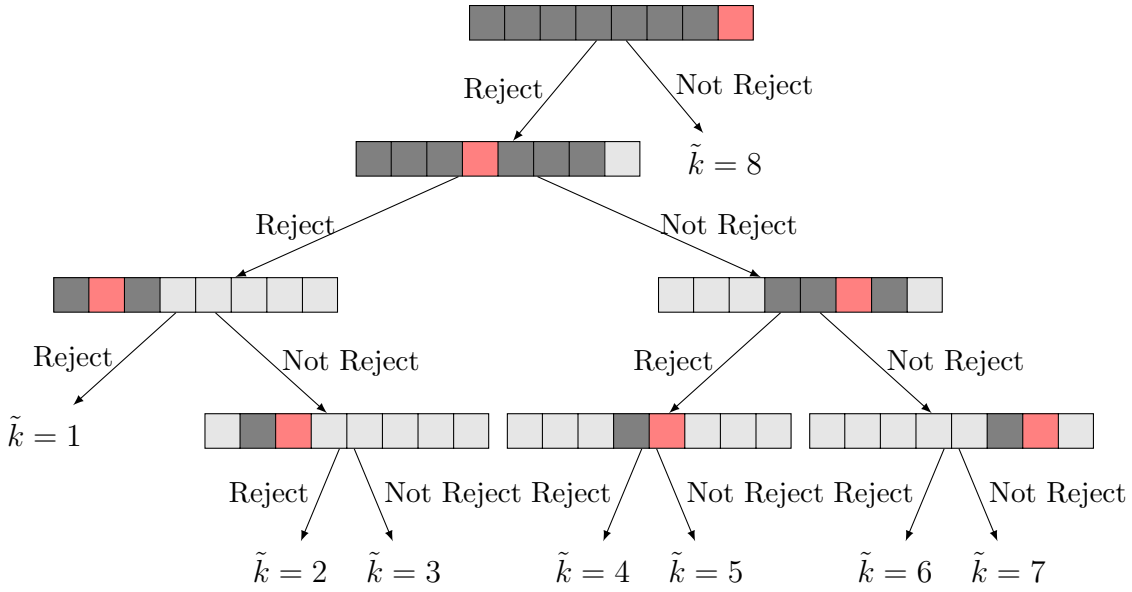


Figure 5.3: Binary tree illustrating the procedure of finding $\tilde{k}$, the maximum k for which the null hypothesis in Equation 5.5 is not rejected. Each node in the tree represents a test of this null hypothesis. The array represents the hyperparameter settings ordered by mean prediction error and the red fill the test point k. Therefore, if the fourth square is red this indicates that $k = 4$. The dark grey fill indicates the remaining possible values for $\tilde{k}$.

pick the best observed setting or sample one setting at random. We next describe how to find $\tilde{k}$. Since $\tilde{\Lambda}$ is sorted this can be done efficiently via a binary search.

Hierarchical Testing Algorithm

The algorithm to find $\tilde{k}$ is shown in Algorithm 1 and illustrated in Figure 5.3 for $K = 8$. Let l be the lowest possible value for $\tilde{k}$, u the highest possible value, and k the current test value in $H_{0,k}$. The procedure starts with $l = 1$, $u = K$, and $k = K$ and the first test is of $H_{0,K}$. The target value $\tilde{k}$ is defined as the largest possible value for which $H_{0,K}$ is not rejected. For this reason, when $H_{0,k}$ is rejected, the largest possible value for $\tilde{k}$ must be $u = k - 1$. If $H_{0,k}$ is not rejected we do not learn anything about u. However, if $H_{0,k}$ is not rejected this means that the lowest possible value for $\tilde{k}$ must be $l = k$ causing us to explore values to the right of k. This defines the recursive pattern needed to find $\tilde{k}$. The maximum number of tests is $\lceil \log_2 K \rceil + 1$.

Algorithm 1 Hierarchical ANOVA

input List of lists Ψ, type I error level α
  $l \leftarrow 1$; $u \leftarrow \text{length}(\Psi)$; $k \leftarrow u$
  while $l < u$ **do**
    p-value $\leftarrow$ ANOVA(Ψ[1:k])
    reject $\leftarrow$ p-value $< \alpha$
    if reject **then**
      $u \leftarrow k - 1$
    else
      $l \leftarrow k$
    end if
    $k \leftarrow \lceil \frac{l+u}{2} \rceil$
  end while
output l

Test for Equal Means

In order to test the hypotheses in Equation 5.5, consider an ANOVA F statistic of the form:

$$F_k = \frac{\frac{n_\lambda}{k-1} \sum_{i=1}^{k} [\hat{\tau}_{\lambda_i} - \hat{\hat{\tau}}]^2}{\frac{1}{k(n_\lambda - 1)} \sum_{i=1}^{k} \sum_{j=1}^{n_\lambda} [\Psi^j(\lambda_i) - \hat{\tau}_{\lambda_i}]^2}. \tag{5.6}$$

Here $\hat{\hat{\tau}} = \frac{1}{k} \sum_{i=1}^{k} \hat{\tau}_{\lambda_i}$ is the overall sample mean of validation losses. Assuming normality in $\Psi^j(\lambda_i)$, independence between observations, and equal variances within each λ_i, F_k follows an F distribution with parameters $k-1$ and $k(n_\lambda - 1)$ under the null hypothesis of equal means. Thus a p-value for the test statistic F_k can be obtained by comparing it to $F(k-1, k(n_\lambda - 1))$.

Closed Testing

The hierarchical nature of the above testing algorithm results in a closed testing procedure [Marcus et al., 1976, Banken et al.] so that the type I error level α for each test is simply that of the overall procedure.

To illustrate this, we first consider the scenario in which we were to find $\tilde{k}$ by iteratively testing $H_{0,k}$ for $k = K, K-1, \ldots, 1$ until we do not reject $H_{0,k}$. The closed testing procedure states we can reject hypothesis $H_{0,k}$ at level α if we reject at level α all hypotheses that intersect with $H_{0,k}$. For the purpose of illustration we use the notation that $(1, \ldots, k)$ represents $H_{0,k} : \tau_{\lambda_1} = \ldots = \tau_{\lambda_k}$. Then consider the example of $K = 4$. To reject the hypothesis $(1, 2, 3, 4)$ we only need to reject $(1, 2, 3, 4)$. To reject $(1, 2, 3)$, we need to reject $(1, 2, 3, 4)$ and $(1, 2, 3)$. To reject $(1, 2)$ we need to reject all of its intersections, that is, $(1, 2)$, $(1, 2, 3)$, $(1, 2, 3, 4)$, $(1, 2, 4)$, as well as $(1, 2), (3, 4)$. In our case the rejection of $(1, 2, 3)$ implies the rejection of $(1, 2, 4)$ due to the fact that we order by the observed means. In addition, $(1, 2), (3, 4)$ gets rejected since $(1, 2)$ is rejected. We now generalize this to any k and K.

Proposition 1. *Given K ordered means and a test of the hypothesis $H_{0,k} : \tau_{\lambda_1} = \ldots = \tau_{\lambda_k}$, $H_{0,k}$ can be rejected at α if the hypotheses $H_{0,k}$ to $H_{0,K}$ are all rejected.*

Proof: We first consider the case where $k = K$. The set of elementary hypotheses is $\{(1, 2), (1, 2, 3), \ldots, (1, \ldots, k)\}$. Therefore, the only elementary hypothesis that $(1, \ldots, k)$ intersects with is itself. Thus, testing $H_{0,k}$ is sufficient to reject $H_{0,k}$.
For the second case, we assume that $(1, \ldots, k+1)$ can be rejected. Then $(1, \ldots, k)$ can be rejected if $(1, \ldots, k)$ gets rejected. This is because the collection of hypotheses that $(1, \ldots, k)$ intersects with I_k consists of $(1, \ldots, k) \cup I_{k+1}$. Having rejected all intersections of $H_{0,k+1}$ (by assumption), rejecting the single hypothesis $(1, \ldots, k)$ implies that we have rejected all intersections with $H_{0,k}$. Using the closed testing principle we can therefore reject $(1, \ldots, k)$. By induction this result generalizes to all k and K.

In the hierarchical testing algorithm described in Section 5.2.2 we may not test all hypotheses between k and K. For example, we may start by testing $(1, 2, 3, 4)$. Upon rejection of that hypothesis we continue by testing $(1, 2)$. We now illustrate why under mild assumptions we

are able to reject $(1,2)$ at α after only rejecting $(1,2,3,4)$ and $(1,2)$ (and skipping $(1,2,3)$). We assume that $\hat{\tau}_{\lambda_1} - \hat{\tau}_{\lambda_2} = \hat{\delta}$ and $\hat{\tau}_{\lambda_2} - \hat{\tau}_{\lambda_3} = \hat{\delta}/m$. Given

$$F_2 = \frac{\frac{n_\lambda}{2-1}\sum_{i=1}^{2}[\hat{\tau}_{\lambda_i} - \frac{\hat{\tau}_{\lambda_1}+\hat{\tau}_{\lambda_2}}{2}]^2}{\frac{1}{2(n_\lambda-1)}\sum_{i=1}^{2}\sum_{j=1}^{n_\lambda}[\Psi^j(\lambda_i) - \hat{\tau}_{\lambda_i}]^2} \tag{5.7}$$

we assume that $\sum_{j=1}^{n_\lambda}[\Psi^j(\lambda_1) - \hat{\tau}_{\lambda_1}]^2 \approx \sum_{j=1}^{n_\lambda}[\Psi^j(\lambda_2) - \hat{\tau}_{\lambda_2}]^2 = SS$. We thus obtain an expression for $\hat{\delta}$ as

$$\hat{\delta} = 2\frac{q_{F(1-\alpha,1,2(n_\lambda-1))}SS}{n_\lambda(n_\lambda-1)}. \tag{5.8}$$

Similarly, given

$$F_3 = \frac{\frac{n_\lambda}{3-1}\sum_{i=1}^{3}[\hat{\tau}_{\lambda_i} - \frac{\hat{\tau}_{\lambda_1}+\hat{\tau}_{\lambda_2}+\hat{\tau}_{\lambda_3}}{3}]^2}{\frac{1}{3(n_\lambda-1)}\sum_{i=1}^{3}\sum_{j=1}^{n_\lambda}[\Psi^j(\lambda_i) - \hat{\tau}_{\lambda_i}]^2}. \tag{5.9}$$

and assuming $\sum_{j=1}^{n_\lambda}[\Psi^j(\lambda_3) - \hat{\tau}_{\lambda_3}]^2 \approx SS$ as well as using $\hat{\tau}_{\lambda_2} - \hat{\tau}_{\lambda_3} = \hat{\delta}/m$ we can obtain an expression for F_3 as

$$F_3 = \frac{1}{9}\left(6 + \frac{6}{m} + \frac{6}{m^2}\right) q_{F(1-\alpha,1,2(n_\lambda-1))}. \tag{5.10}$$

By comparing F_3 to $q_{F(1-\alpha,2,3(n_\lambda-1))}$ we can evaluate whether $H_{0,3}$ would be rejected or not. Table 5.1 evaluates this for different values of n_λ and m and a fixed $\alpha = 0.05$. Here "T" indicates rejection of $H_{0,3}$ and "F" indicates the test would not have been rejected. The table shows that for $n_\lambda = 3$, $\hat{\tau}_{\lambda_2}$ and $\hat{\tau}_{\lambda_3}$ can almost be equal and $H_{0,3}$ would still be rejected. For $n_\lambda = 10$ there needs to be a difference of at least $\hat{\delta}/9$ between $\hat{\tau}_{\lambda_2}$ and $\hat{\tau}_{\lambda_3}$ to reject $H_{0,3}$ given the rejection of $H_{0,2}$. We note that given $\alpha = 0.01$ all fields in the table evaluate as "T".

While omitting a formal proof, it is easy to see that this result extends to the general case where rejection of the hypotheses $H_{0,k}$ and $H_{0,2k}$ implies the rejection of the hypotheses

Table 5.1: Table showing whether $H_{0,3}$ would be rejected given the rejection of $H_{0,2}$, $\hat{\tau}_{\lambda_1} - \hat{\tau}_{\lambda_2} = \hat{\delta}$, $\hat{\tau}_{\lambda_2} - \hat{\tau}_{\lambda_3} = \hat{\delta}/m$, and $\alpha = 0.05$. "T" indicates rejection of $H_{0,3}$ and "F" indicates that the test would not be rejected.

$n_\lambda \backslash m$	1	2	3	4	5	6	7	8	9	10	100	1000	10000
3	T	T	T	T	T	T	T	T	T	T	T	T	F
5	T	T	T	T	T	T	T	T	T	T	F	F	F
10	T	T	T	T	T	T	T	T	F	F	F	F	F

$H_{0,k+1}, \ldots, H_{0,2k-1}$. Therefore, by Proposition 1 the testing procedure described in Section 5.2.2 is a closed testing procedure under mild assumptions on the observed effect sizes.

Sequential Testing

We propose to use group sequential testing [Jennison and Turnbull, 1999] in order to address efficiency issues when sampling multiple trials for each hyperparameter setting. Instead of conducting the testing procedure in Section 5.2.2 once, it is done multiple times at different fractions of the total sample size.

Previously, we discussed the FWER for a single hierarchical test. Now we embed this procedure in a sequential testing framework. The new type I error rate corresponding to the t-th hierarchical test is α_t. We begin by specifying the total number of analyses to be performed, T, and the samples sizes available at each analysis: $n_\lambda^1, \ldots, n_\lambda^T$.

The testing procedure starts with $\tilde{\Lambda}^1 = \{\lambda_1, \ldots, \lambda_K\}$ where n_λ^1 samples are collected for all λ_i. $\tilde{\Lambda}^1$ is sorted by the mean validation loss and the hierarchical testing produces $\tilde{k}^1$ at FWER α^1. For the second analysis $\tilde{\Lambda}^2 = \{\lambda_1, \ldots, \lambda_{\tilde{k}^1}\}$. We collect $n_\lambda^2 - n_\lambda^1$ additional samples for all $\lambda_i \in \tilde{\Lambda}^2$ and sort by the observed mean validation loss. Hierarchical testing produces $\tilde{k}^2$ at FWER α^2. This process is repeated until we obtain $\tilde{k}^T$ and the corresponding $\tilde{\Lambda}^T = \{\lambda_1, \ldots, \lambda_{\tilde{k}^T}\}$.

In order to avoid inflation of the family-wise error rate due to repeatedly testing accruing data, sequential testing theory based upon the results of [Armitage et al., 1969] offers adjusted FWERs α^t for each round of tests.

Sequential Testing Boundaries

We consider one-sided sequential testing boundaries with early stopping only in the case of rejection of the null hypothesis. The hierarchical test at the tth analysis is then based on the observations available at that time. In order to apply this methodology to the proposed hierarchical test we obtain the critical value for the hypothesis test at each interim analysis by numerically integrating the group sequential density [Armitage et al., 1969] in order to maintain the overall FWER. These critical values form the stopping boundaries for the sequential testing procedure. As noted in Emerson et al. [2007], an infinite number of possible boundaries can be formed to maintain the FWER, each displaying varying amounts of early conservatism for rejecting the null hypothesis, though all are defined as a function of the fraction of maximal statistical information to be obtained at the final planned analysis. Here we utilize the unified family of group sequential stopping boundary [Kittelson and Emerson, 1999]. Let Π_t be the accumulated information at analysis t which is n_t/n_T. The unified family can be expressed as:

$$g(\Pi; A, P, R, G) = (A + \Pi^{-P}(1 - \Pi)^R G). \tag{5.11}$$

Here, A, P, and R are used to design the shape of the boundary. The parameter G is found via computational search to satisfy desired operating characteristics such as the type I error (cf. Emerson et al. [2007]). This formulation of a sequential testing boundary is expressed on the scale of the maximum likelihood estimator. For our purposes, we can translate the boundary to the Z-scale under mean zero and unit variance by multiplying by $\sqrt{n_t}$. The type I error

level for the tth hierarchical test is then given by $1 - \Phi(\sqrt{n_t} g(\Pi_t; A, P, R, G))$ where Φ is the cumulative density function of the standard Normal distribution. Popular sequential testing designs such as the boundary introduced by Pocock [1977] and the boundary introduced by O'Brien and Fleming [1979] are given by the settings $A = 0$, $P = 0.5$, and $R = 0$ and $A = 0$, $P = 1$, and $R = 0$, respectively. It is worth noting that the Pocock design is constant on the Z-scale. With three total analyses equally spaced in information time, the resulting critical value of a one-sided test is approximately 1.99. This will be utilized in the examples provided in Section 3.

Hyperparameter Search Algorithm

If training is stochastic, a hyperparameter search using K hyperparameter settings from the space Λ can be augmented to use the proposed methodology. Testing with overall type I error level α can be conducted at partial sample sizes n_λ to eliminate unpromising hyperparameter settings as early as possible. Hyperparameter settings can be suggested as desired since no assumptions have been made about the hyperparameter settings $\tilde{\Lambda}$. Examples of possible algorithms are random search, grid search, or Bayesian optimization.

Algorithm 2 provides pseudo-code for an implementation of the proposed procedure. The sequential testing design is specified by the parameter P. For the pseudo-code in Algorithm 2 the function `sequential_boundary` implements the unified family of sequential testing designs described in Section 5.2.2 and allows for a flexible choice of stopping boundaries. As mentioned, the implementation can be used with any algorithm that suggests hyperparameter settings which is represented by the function `generate_hp`. In this case `generate_hp` takes arguments $\tilde{\Lambda}[1:k]$, $\Psi[:, 1:k]$, and Λ indicating that it can depend on the hyperparameter configurations explored so far, their objective values, and the hyperparameter space, respectively. In Algorithm 2 the function `train_and_eval` represents the training and evaluation of the machine learning algorithm and returns the validation loss.

Finally, `hierarchical_test` implements the hierarchical testing procedure described in Section 5.2.2. The procedure returns the set of equivalent hyperparameters as described in Section 5.2.2.

Algorithm 2 Sequential Hierarchical Test

input Number of hyperparameter settings K, array of cumulative sample size at each interim analysis n_λ, hyperparameter space Λ, overall type I error level α, sequential testing boundary parameter P

```
α* ← sequential_boundary(α, n_λ, P)
T ←length(n_λ)
Ψ ← nans((K, n_λ[T]))
k̃ ← K
Λ̃ ←list()
for t = 1 to T do
  count←n_λ[t − 1] if t > 1 else 0
  for k = 1 to k̃ do
    if t == 1 then
      Λ̃[k] = generate_hp(Λ̃[1 : k], Ψ[:, 1 : k], Λ)
    end if
    for j =count to n_λ[t] do
      Ψ[j, k]←train_and_eval(Λ̃[k])
    end for
  end for
  τ ←colmeans(Ψ[:, 1:k̃], na.rm=True)
  idxs ← argsort(τ)
  τ ← τ[idxs]
  Λ̃ ← Λ̃[idxs]
  Ψ ← Ψ[:, idxs]
  k̃ ← hierarchical_test(Ψ, τ, α*[t])
end for
```

output $\tilde{\Lambda}[1{:}\tilde{k}]$

5.3 Results

5.3.1 Type I Error Simulation

We verify via simulation the bound on the theoretical type I error for the hierarchical testing method introduced in Section 5.2.2 and the same procedure embedded in the sequential testing framework described in Section 5.2.2. For the simulation we let $K = 100$ and $\Psi^j(\lambda_k) \sim N(0,1)$ for $k = 1,\ldots,K$ and $j = 1,\ldots,n_\lambda$. In the case of the fixed sample hierarchical test we let $n_\lambda = 10$. For the hierarchical test embedded in the sequential testing procedure we let $n_\lambda = (3,6,9)$. The type I error level is set to $\alpha = 0.05$ and each simulation is repeated 1000 times. The observed rate of type I errors for the fixed sample hierarchical test is 4.4% in the simulation. For the sequential hierarchical test the observed type I error level is 5.7%.

5.3.2 Motivating Example

We revisit the motivating example introduced in Figure 5.2. This example had shown that for a simple noisy function from hyperparameter to prediction error a regular hyperparameter optimization would have a high variance in outcomes as opposed to just the one optimal value. The curve shown in Figure 5.2a is simulated as $Y = (X-3)^2 + 10 + \epsilon$ where $\epsilon \sim N(0,\sigma^2)$. We let $\sigma^2 = 4$. In this case X represents the hyperparameter and Y represents the obtained prediction error. The expected prediction error for a specific hyperparameter across multiple runs is given by $\mathbb{E}(Y|X) = (X-3)^2 + 10$. We simulate hyperparameter searches in this scenario by evaluating X on a grid of $\{0,1,2,3,4,5,6\}$ which includes the optimal value 3. The hyperparameter search simulation is repeated 10000 times to obtain results. Figure 5.4 shows the selected hyperparameter from a hyperparameter search with $n_\lambda = 1$ against the proposed method with $n_\lambda = (3,6,9)$. Here $n_\lambda = (3,6,9)$ means that we

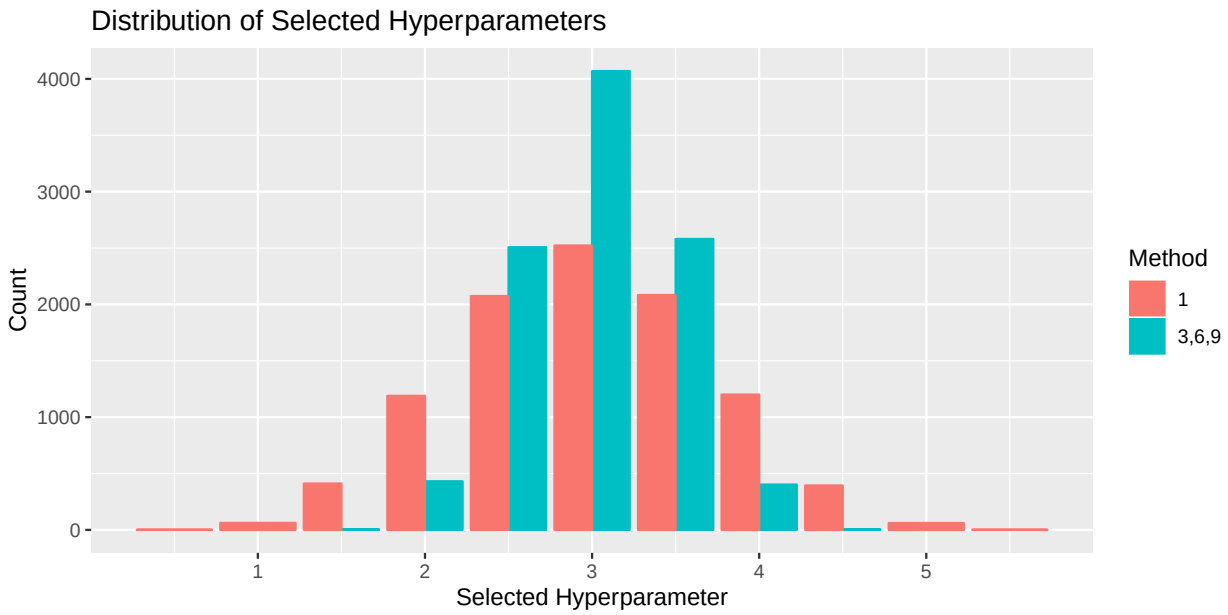


Figure 5.4: Bar chart showing counts out of 10000 runs that each hyperparameter value is selected by methods with one trial per hyperparameter setting ($n_\lambda = 1$) and the proposed procedure with intermediate sample sizes of $n_\lambda = (3, 6, 9)$. The proposed method selects the optimal value of 3 in a larger proportion of cases than the method with $n_\lambda = 1$. In addition to that, the overall distribution is more peaked around the optimal value (variance of 0.21 vs 0.57).

first make three draws for all values of X, then reduce the set of candidate X values according to the test in Section 5.2.2 and draw another three evaluations, and then repeat this process once more. It can be seen that the bar-chart for $n_\lambda = (3, 6, 9)$ is more peaked around the optimal value of $X = 3$ as compared to the bar-chart for $n_\lambda = 1$. This means that the optimal value gets selected in a larger proportion of hyperparameter searches. In particular, the mean and variance of selected hyperparameters for $n_\lambda = 1$ are given by 2.97 and 0.53, respectively. For $n_\lambda = (3, 6, 9)$ the corresponding mean and variance are 2.99 and 0.21, respectively, a 60% decrease in variation. Figure 5.5 shows the expected prediction errors $\mathbb{E}(Y|X)$ corresponding to the selected hyperparameters in Figure 5.4. It can be seen that the distribution for the regular method with $n_\lambda = 1$ is much wider (mean=10.53, variance=0.44) as compared to that for $n_\lambda = (3, 6, 9)$ (mean=10.21, variance=0.069).

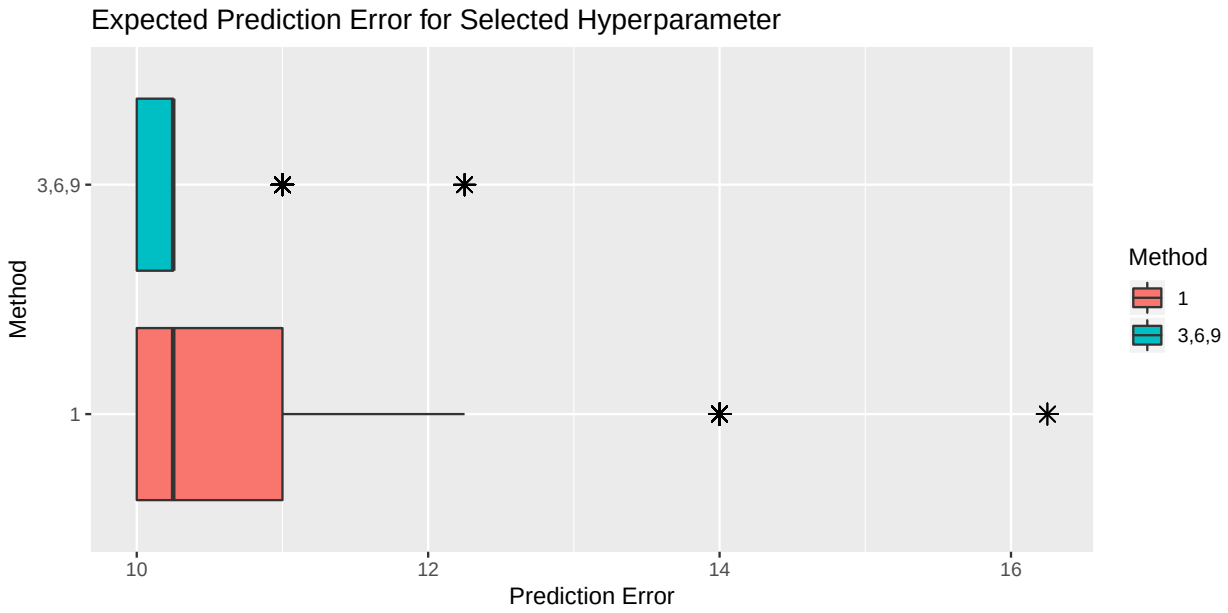


Figure 5.5: The figure shows a box plot that corresponds to the expected prediction error across multiple runs given selected hyperparameter values from Figure 5.4. Using an oracle the value of the expected prediction error would be 10 with zero variance. As shown by their corresponding boxes, the proposed method ($n_\lambda = (3, 6, 9)$) is closer to the optimum (mean=10.21, variance=0.075) when compared to the traditional method of $n_\lambda = 1$ (mean=10.57, variance=0.55).

5.3.3 Evaluation Procedure

We consider experiments with $n_\lambda = 1$, $n_\lambda = 5$, $n_\lambda = 10$, or $n_\lambda = (3, 6, 9)$ runs per hyperparameter setting. In the case of $n_\lambda = 1$, $\tilde{k} = 1$ meaning the hyperparameter setting with minimum validation error is returned as is currently standard practice. The cases of $n_\lambda = 5$ or 10 use the hierarchical test from Section 5.2.2 with a fixed sample size. The sequential testing case with $n_\lambda = (3, 6, 9)$ uses the sequential testing procedure from Section 5.2.2 indicating that interim analyses are conducted after 3, 6, and 9 runs per hyperparameter setting. Experiments use a Pocock boundary ($P = 0.5$) unless otherwise indicated. The following quantities are estimated for each n_λ:

1. $\hat{\Pr}(k^* \in [1, \tilde{k}])$, the empirical probability that the hyperparameter search returns the on average optimal setting among the explored hyperparameter settings, where k^* refers to the index of the optimal setting $\lambda^* \in \tilde{\Lambda}$.

2. Mean($\tilde{k}$), the number of hyperparameter settings returned on average by the procedure. Lower is better in this case.

3. $\hat{\Pr}(k \in [\tilde{k} + 1, K] | k \neq k^*)$, the power to reject non-optimal hyperparameter settings.

4. Average number of evaluations of $\mathcal{A}$ across the entire hyperparameter optimization.

5.3.4 Evaluation Data Sets and Experimental Setup

The above quantities are estimated via simulation on precomputed results. Specifically, hyperparameter optimization is run with 600 to 1000 randomly sampled hyperparameter configurations each repeated across 25 runs. For one simulation, a subset of K hyperparameter configurations is sampled from these data. A Monte Carlo estimate of the optimal hyperparameter setting λ^* is obtained. Then the proposed method is run, sampling n_λ trials

Table 5.2: Hyperparameter space Λ for MNIST CNN benchmark.

Hyperparameter	Type	Range	Scale
Learning Rate	Float	[0.001, 0.1]	Log
Learning Rate Decay	Float	[1e-4, 1e-9]	Log
Dropout Lower Layer	Float	[0.0001, 0.7]	Linear
Dropout Higher Layer	Float	[0.0001, 0.7]	Linear

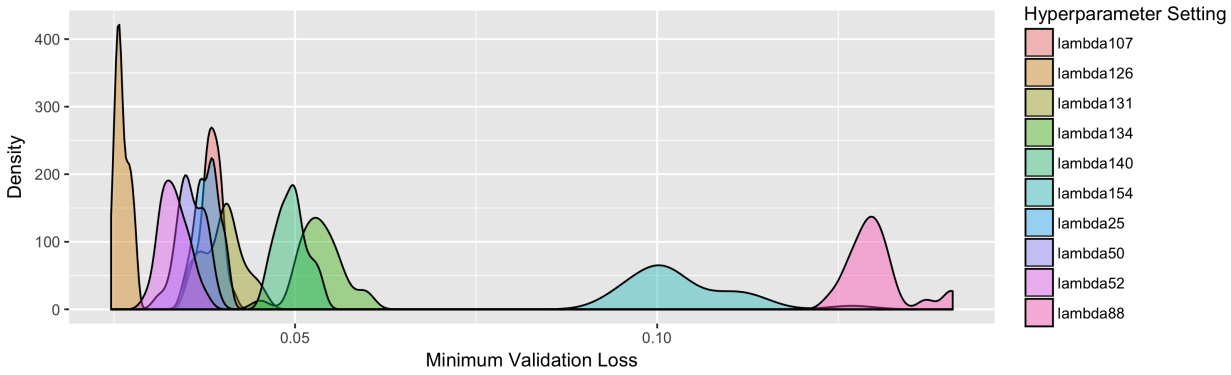


Figure 5.6: Distribution of validation loss across 25 runs for a sample of 10 hyperparameter settings in the MNIST CNN experiment.

from the given 25 trials for each of the K hyperparameter configurations in the subset and obtaining the $\tilde{k}$ best configurations. Then a new subset of size K is sampled and the process repeats. The quantities in Section 5.3.3 are estimated via 1000 independent simulations of this type.

All hyperparameter optimizations are run using the software Sherpa developed by Hertel et al. [2018] and its random search algorithm. The proposed sequential testing method is implemented in R [Team et al., 2013] using Jupyter Notebooks [Kluyver et al., 2016]. The MNIST convolutional neural network and the IMDB LSTM are implemented in Keras and loosely based on the respective examples provided by Keras `https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py` and `https://github.com/keras-team/keras/blob/master/examples/imdb_lstm.py`). The gradient boosting regressor is implemented in Scikit-Learn developed by Pedregosa et al. [2011].

Table 5.3: Results for the MNIST Convolutional Neural Network hyperparameter optimization across 1000 independent simulations for each value of K.

n_λ	1	5	10	3,6,9
$\underline{K = 50}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.6	1	1	1
Mean($\tilde{k}$)	1	3.1	2.1	2.5
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	0.99	0.96	0.98	0.97
Avg Evaluations	50	250	500	180
$\underline{K = 100}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.48	1	1	1
Mean($\tilde{k}$)	1	4.4	3	3.4
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	0.99	0.97	0.98	0.98
Avg Evaluations	100	500	1000	340
$\underline{K = 150}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.43	1	1	1
Mean($\tilde{k}$)	1	5.7	3.7	4.3
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	1	0.97	0.98	0.98
Avg Evaluations	150	750	1500	500

5.3.5 MNIST Convolutional Neural Network

A convolutional neural network is trained on the MNIST data set to classify handwritten digits. The hyperparameter space and ranges are shown in Table 5.2. Results for the metrics described in Section 5.3.3 are shown in Table 5.3. For the $n_\lambda = 1$ case, the optimal hyperparameter setting is returned approximately 60% of the time ($\hat{\Pr}(k^* \in [1, \tilde{k}])$ for $K = 50$ candidate hyperparameter settings). For $n_\lambda = 10$ this increases to approximately 100% of the time, however at a 10-fold increase in computation. Using sequential testing with incremental sample sizes of $n_\lambda = (3, 6, 9)$ the optimal hyperparameter setting is still found approximately 100% of the time while computation increases approximately 3.6 times. At the same time the average number of returned hyperparameter settings Mean($\tilde{k}$) is not too large at 2.5. Similar results can be observed for the $K = 100$ and $K = 150$ cases.

Table 5.4: Hyperparameter space Λ for IMDB LSTM benchmark.

Hyperparameter	Type	Range	Scale
Learning Rate	Float	[5e-4, 5e-3]	Log
Learning Rate Decay	Float	[1e-5, 1e-10]	Log
RMSprop ρ	Float	[0.5, 0.99]	Linear
Batch Size	Int	[10, 150]	Log
Embedding L2-penalty	Float	[1e-12, 1e-6]	Log
Kernel L2-penalty	Float	[1e-8, 1e-0]	Log
Recurrent L2-penalty	Float	[1e-8, 1e-0]	Log
Embedding Dropout	Float	[0.0001, 0.5]	Linear
LSTM Dropout	Float	[0.0001, 0.5]	Linear
Embedding Features	Int	[15000, 40000]	Linear
Hidden Dimension	Int	[100, 500]	Log

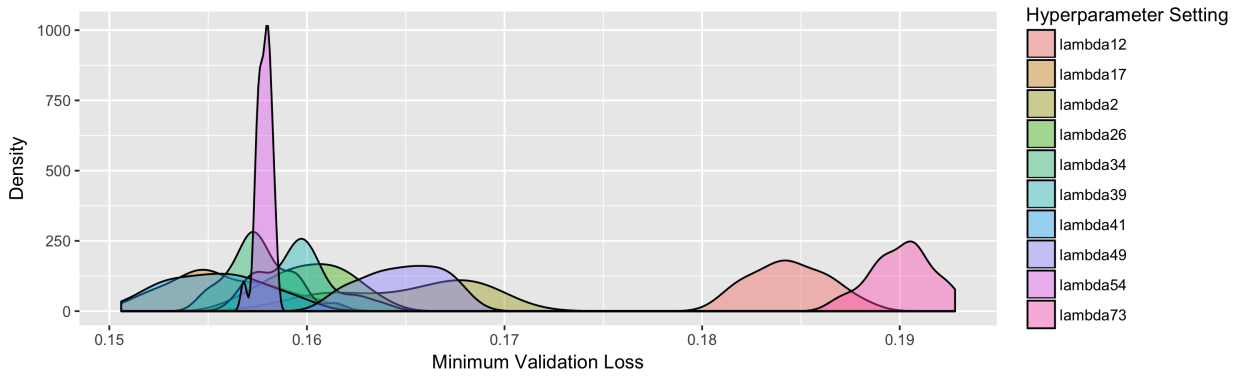


Figure 5.7: Distribution of validation loss across 25 runs for a sample of 10 hyperparameter settings in the IMDB LSTM experiment.

5.3.6 IMDB Movie Review LSTM

A long-short term memory recurrent neural network (LSTM) is trained to classify IMDB movie reviews with respect to positive/negative sentiment. The objective is the classification error. Hyperparameter ranges and model architecture are described in Table 5.4. The training data set consists of 25,124 examples; hyperparameter evaluation is done on a 12,376 sample validation set; the test set with 12,500 examples is reserved. Table 5.5 shows results of the proposed procedure compared to standard practices for $K = 50$, $K = 100$, and $K = 150$ hyperparameter settings.

Table 5.5: Results for the IMDB LSTM hyperparameter optimization across 1000 independent simulations for each value of K.

n_λ	1	5	10	3,6,9
$\underline{K = 50}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.61	1	1	1
Mean($\tilde{k}$)	1	3.5	2.3	2.5
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	0.99	0.95	0.97	0.97
Avg Evaluations	50	250	500	180
$\underline{K = 100}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.51	1	1	0.99
Mean($\tilde{k}$)	1	5	3.2	3.5
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	1	0.96	0.98	0.98
Avg Evaluations	100	500	1000	340
$\underline{K = 150}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.49	0.99	1	0.99
Mean($\tilde{k}$)	1	6.1	3.8	4.2
$\hat{\Pr}(k \in [\tilde{k}+1, K] \vert k \neq k^*)$	1	0.97	0.98	0.98
Avg Evaluations	150	750	1500	500

Similar to the results shown in Section 5.3.5 it can be seen that the optimal hyperparameter setting is only found 61%, 51%, and 49% of the time when $n_\lambda = 1$ and $K = 50$, $K = 100$, and $K = 150$ hyperparameter settings, respectively. This proportion increases to near 100% when any of the methods with repeated trials is used. Among those methods, $n_\lambda = (3, 6, 9)$ achieves the best trade-off between $\hat{\Pr}(k^* \in [1, \tilde{k}])$ and the average number of total evaluations, that is the computational cost.

5.3.7 Boston Housing Gradient Boosting Regressor

A gradient boosting regressor (GBR) is trained on the Boston Housing data set. Estimated are median housing prices for different locations. Predictors are 13 attributes of houses in different locations around the Boston suburbs in the late 1970s. The data set is split

Table 5.6: Hyperparameter space Λ for Boston Housing GBR benchmark.

Hyperparameter	Type	Range	Scale
Learning Rate	Float	[0.05, 0.15]	Linear
Number of Estimators	Int	[100, 400]	Linear
Max-depth	Int	[2, 10]	Linear
Subsample	Float	[0.5, 1.0]	Linear

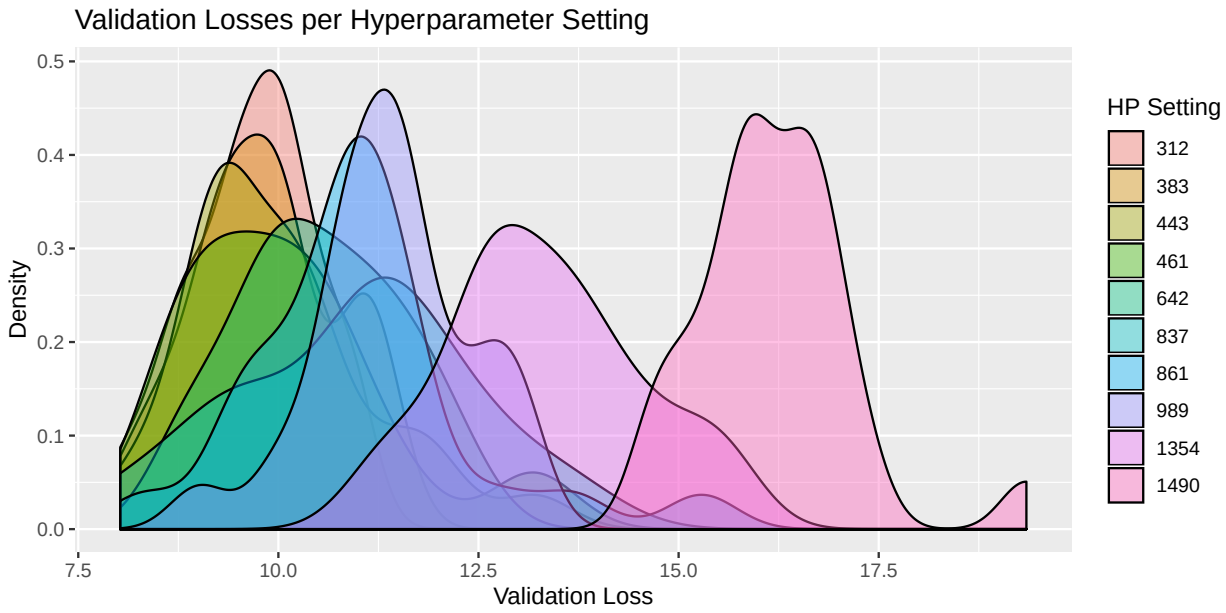


Figure 5.8: Distribution of validation loss across 25 runs for a sample of 10 hyperparameter settings in the Boston Housing GBR benchmark.

into a 303 example training set, 101 examples for validation, and 102 examples for testing. Tuning parameters include the learning rate, number of estimators, maximum depth, and subsampling proportion and are shown in Table 5.6. Figure 5.8 shows the validation loss distributions for a sample of 10 hyperparameter settings. It can be seen that there is a large amount of overlap between hyperparameter settings.

Table 5.7 shows previously introduced metrics evaluated across 1000 simulations with $K = 50, 100, 150$. As expected from the overlapping validation loss distributions shown in Figure 5.8 the ability to find the overall optimal hyperparameter setting is smaller as shown by

Table 5.7: Results for the Boston Housing gradient boosted regression tree hyperparameter optimization across 1000 independent simulations for each value of K.

n_λ	1	5	10	3,6,9
$\underline{K = 50}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.16	1	1	1
Mean($\tilde{k}$)	1	17	12	13
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.98	0.67	0.78	0.75
Avg Evaluations	50	250	500	280
$\underline{K = 100}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.071	1	1	1
Mean($\tilde{k}$)	1	32	23	24
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.99	0.69	0.78	0.76
Avg Evaluations	100	500	1000	530
$\underline{K = 150}$				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.07	1	1	1
Mean($\tilde{k}$)	1	46	33	35
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.99	0.7	0.78	0.77
Avg Evaluations	150	750	1500	770

$\hat{\Pr}(k^* \in [1, \tilde{k}])$ than in the previous examples. We discuss the case of $K = 50$. For $n_\lambda = 1$ only 16% of runs return the on average optimal hyperparameter setting out of the explored settings. The sequential testing procedure with $n_\lambda = (3, 6, 9)$ returns the optimal hyperparameter setting in 100% of the simulations. However, the average number of returned hyperparameter settings is 13. Furthermore, fewer hyperparameter settings are eliminated at $n_\lambda^1 = 3$ as indicated by the average number of evaluations.

5.3.8 Distribution of Hyperparameter Optimization Outcomes

Reproducible hyperparameter optimization should result in low average prediction error across model training runs and low variability across hyperparameter optimization runs. Let $\hat{\lambda^*}$ be an estimate of the optimal hyperparameter setting $\lambda^* \in \Lambda$. The estimated optimal

hyperparameter setting is then associated with $\tau_{\hat{\lambda}^*} = \mathbb{E}_{\mathcal{A}_\lambda}[\Psi(\hat{\lambda}^*)]$ the expectation across model training runs and for our purposes the outcome of the hyperparameter optimization. There are two ways of obtaining an optimal hyperparameter setting from the suggested method. One way is to pick the setting with the overall best observed average validation loss. The other way is to sample one configuration from the equivalence class returned by the method. In this section we consider the empirical distribution of $\tau_{\hat{\lambda}^*} = \mathbb{E}_{\mathcal{A}_\lambda}[\Psi(\hat{\lambda}^*)]$, the expected validation loss of the hyperparameter setting $\hat{\lambda}^*$ across different runs of the hyperparameter search via box-plots. We also provide summaries using the estimated mean $\hat{\mathbb{E}}\{\tau_{\hat{\lambda}^*}\}$ and variance $\hat{\text{Var}}\{\tau_{\hat{\lambda}^*}\}$.

Figure 5.9 shows empirical distributions of the expected prediction error $\tau_{\hat{\lambda}^*}$ across hyperparameter optimization runs. The method "1" corresponds to the standard method with one evaluation per hyperparameter setting. The method "3,6,9" corresponds to the proposed sequential testing procedure. The methods "3,6,9" and "3,6,9-s" correspond to picking the best observed and sampling from the equivalence class, respectively. We obtain Monte Carlo approximations of the expected prediction error across model training runs as the average across 25 evaluations. The figure rows correspond to the three experiments introduced previously. For each experiment prediction errors are shown on the validation data set and the test data set which is not used during the hyperparameter optimization. The methods "3,6,9" and "3,6,9-s" use $K = 150$ explored hyperparameter settings. To make the comparison fair, we choose K for the standard method $n_\lambda = 1$ so that its computational cost approximately matches that of the $n_\lambda = (3, 6, 9)$ method. That means that for $n_\lambda = 1$, $K = 496$ for the MNIST CNN experiment, $K = 501$ for the IMDB LSTM experiment, and $K = 775$ for the Boston GBR experiment. The results in Figure 5.9 indicate a number of trends. First the median values for the "3,6,9" method are equal to or lower than for the $n_\lambda = 1$ method in all plots. In the Boston GBR experiment the $n_\lambda = (3, 6, 9)$ method obtains a much lower median expected prediction error. We believe that this may be due to the large amount of variation within hyperparameter settings for this experiment. The "3,6,9-s" method indicates a higher

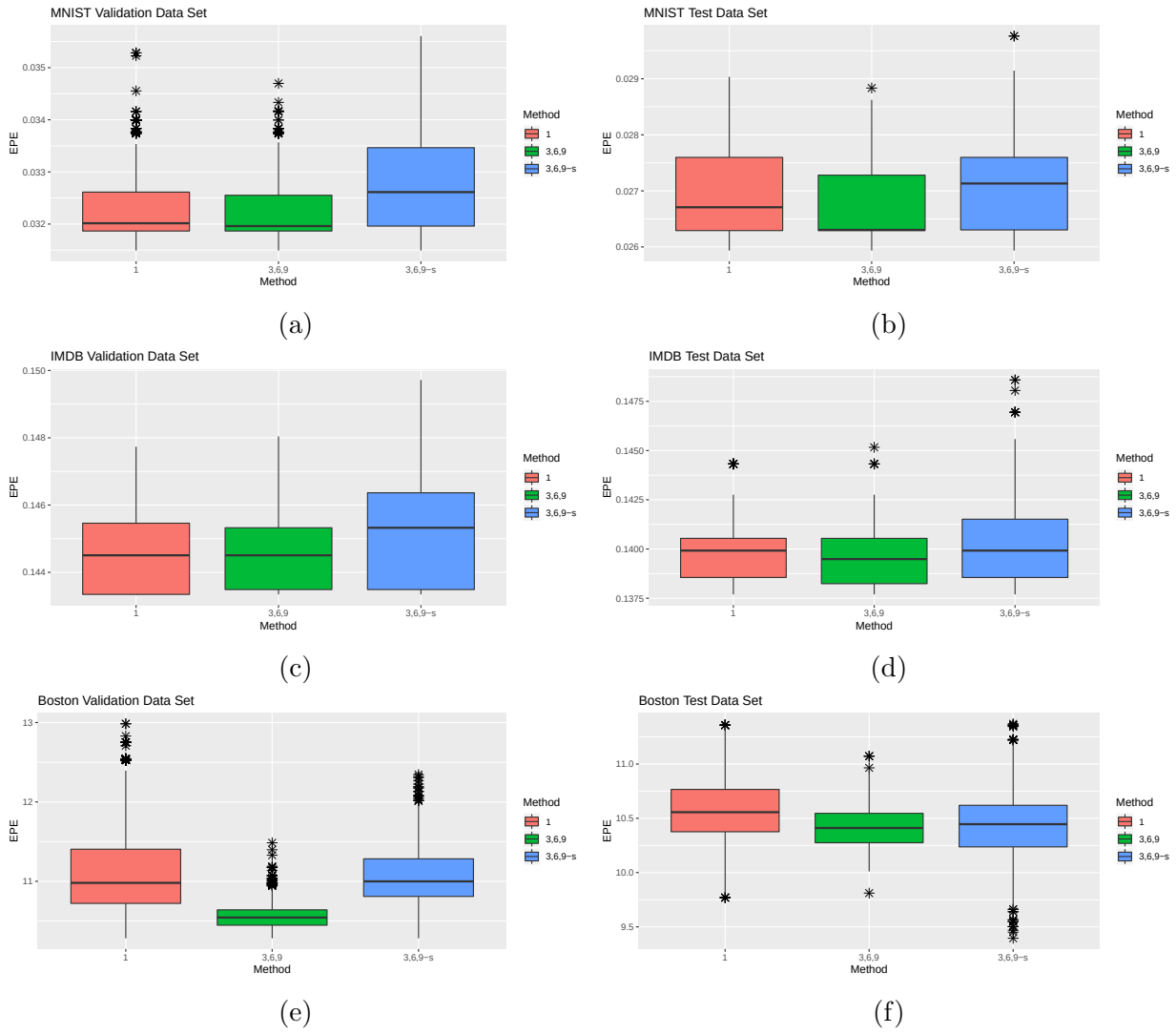


Figure 5.9: Boxplots showing distributions of the expected prediction error $\tau_{\hat{\lambda^*}}$ yielded by the hyperparameter search using methods with $n_\lambda = 1$ and $n_\lambda = (3, 6, 9)$ and $K = 150$ candidate settings. Here, "3,6,9" corresponds to choosing the best observed hyperparameter setting from the returned equivalence class while "3,6,9-s" corresponds to randomly sampling one from the equivalence class.

median for most of the plots. The spread given by the interquartile range of the box-plots is smaller or equal in all cases for the $n_\lambda = (3, 6, 9)$ method when compared to the $n_\lambda = 1$ method with the exception of Figure 5.9d (IMDB test data set). The $n_\lambda = (3, 6, 9)$ method with sampling indicates a spread equal to or larger than the $n_\lambda = 1$ method in all plots.

Table 5.8: Means (variances) of $\tau_{\hat{\lambda^*}}$ across hyperparameter optimization runs.

	$n_\lambda = 1$	$n_\lambda = (3, 6, 9)$	
		λ best	λ sampled
MNIST			
	$K = 177$	$K = 50$	
Valid	3.262e-02(7.981e-07)	3.311e-02(1.435e-06)	3.352e-02(2.114e-06)
Test	2.701e-02(6.170e-07)	2.728e-02(8.676e-07)	2.755e-02(1.217e-06)
	$K = 338$	$K = 100$	
Valid	3.245e-02(6.425e-07)	3.243e-02(5.951e-07)	3.298e-02(1.134e-06)
Test	2.690e-02(5.665e-07)	2.686e-02(4.888e-07)	2.720e-02(7.096e-07)
	$K = 496$	$K = 150$	
Valid	3.227e-02(4.899e-07)	3.217e-02(3.643e-07)	3.281e-02(8.175e-07)
Test	2.682e-02(5.481e-07)	2.667e-02(4.179e-07)	2.707e-02(5.712e-07)
IMDB			
	$K = 178$	$K = 50$	
Valid	1.451e-01(2.189e-06)	1.456e-01(2.250e-06)	1.461e-01(3.075e-06)
Test	1.402e-01(3.064e-06)	1.404e-01(3.599e-06)	1.410e-01(5.494e-06)
	$K = 340$	$K = 100$	
Valid	1.448e-01(1.539e-06)	1.449e-01(1.548e-06)	1.456e-01(2.663e-06)
Test	1.401e-01(1.920e-06)	1.399e-01(2.371e-06)	1.405e-01(3.951e-06)
	$K = 501$	$K = 150$	
Valid	1.445e-01(1.298e-06)	1.444e-01(1.105e-06)	1.452e-01(2.226e-06)
Test	1.399e-01(1.793e-06)	1.396e-01(1.851e-06)	1.402e-01(3.392e-06)
Boston			
	$K = 278$	$K = 50$	
Valid	1.111e+01(3.266e-01)	1.066e+01(3.886e-02)	1.115e+01(1.898e-01)
Test	1.052e+01(9.672e-02)	1.044e+01(3.261e-02)	1.040e+01(1.205e-01)
	$K = 528$	$K = 100$	
Valid	1.107e+01(2.840e-01)	1.058e+01(3.004e-02)	1.107e+01(1.364e-01)
Test	1.056e+01(8.644e-02)	1.042e+01(3.051e-02)	1.041e+01(1.134e-01)
	$K = 775$	$K = 150$	
Valid	1.109e+01(2.880e-01)	1.056e+01(2.934e-02)	1.106e+01(1.359e-01)
Test	1.059e+01(9.065e-02)	1.041e+01(3.040e-02)	1.042e+01(1.063e-01)

Table 5.8 shows means and variances corresponding to the distributions in Figure 5.9. The first column indicates results for the standard random search with $n_\lambda = 1$ trial per hyperparameter setting. The second column shows results for the proposed method with intermediate

sample sizes of $n_\lambda = (3, 6, 9)$ and choosing the best observed hyperparameter setting λ from the resulting equivalence class. The last column shows results for $n_\lambda = (3, 6, 9)$, but randomly sampling one hyperparameter setting from the equivalence class. The means and variances for $K = 150$ support the findings from the box-plots in Figure 5.9. For smaller values of K the observed variances of the proposed method increase (middle column) as compared to the standard method (left column). We believe that this may be due to the variation in the sampling of hyperparameter settings from the random search. This variation is expected to be more pronounced if less samples are taken (smaller values of K). We expect that model based sampling of hyperparameter settings such as in Bayesian optimization would alleviate this issue.

5.3.9 Sequential Testing Boundaries

Sequential testing offers an infinite choice of boundary functions when a unified family approach is considered as explained in Section 5.2.2. We consider varying the parameter P in the unified family. This parameter can be viewed as a measure of conservativeness at the first analysis. Lower values therefore correspond to tighter boundaries and higher rejection rates at the initial analysis. Table 5.9 shows previously introduced results for $K = 150$ using sequential testing boundaries corresponding to $P = 0.25$, 0.5, 0.75, and 1.. Considering the rate at which the optimal hyperparameter is returned $\hat{\Pr}(k^* \in [1, \tilde{k}])$ all values of P achieve a rate close to unity for all experiments. However, when the average number of returned configurations $\text{Mean}(\tilde{k})$ is considered it can be seen that larger values of P reduce this value. This is expected since the testing procedure has higher power at later analyses. As expected, the average number of evaluations increases for that reason when P is bigger. Finally, we would expect the mean and variance between successive runs of the hyperparameter optimization $\mathbb{E}\{\tau_{\hat{\lambda^*}}\}$ and $\text{Var}\{\tau_{\hat{\lambda^*}}\}$ to be lower for larger values of P. Table 5.9 does not clearly support this expectation. An explanation for this is that the optimal hyperparameter setting

Table 5.9: Results for 1000 simulations using different sequential testing boundaries parameterized by P for $K = 150$ candidate settings.

P	0.25	0.5	0.75	1.
MNIST				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.997	1	1	1
Mean($\tilde{k}$)	4.56	4.301	3.987	3.99
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.976	0.978	0.98	0.98
Avg Evaluations	493.689	496.512	500.787	510.423
$\hat{\mathbb{E}}\{\tau_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau_{\hat{\lambda}^*}\})$	0.032(3.594e-07)	0.032(3.778e-07)	0.032(3.811e-07)	0.032(3.550e-07)
$\hat{\mathbb{E}}\{\tau^T_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau^T_{\hat{\lambda}^*}\})$	0.027(4.259e-07)	0.027(4.301e-07)	0.027(4.232e-07)	0.027(4.171e-07)
IMDB				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.992	0.99	0.994	0.993
Mean($\tilde{k}$)	4.804	4.327	4.062	3.948
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.974	0.978	0.979	0.98
Avg Evaluations	499.194	501.741	506.433	515.751
$\hat{\mathbb{E}}\{\tau_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau_{\hat{\lambda}^*}\})$	0.144(1.064e-06)	0.144(1.051e-06)	0.144(1.062e-06)	0.144(1.108e-06)
$\hat{\mathbb{E}}\{\tau^T_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau^T_{\hat{\lambda}^*}\})$	0.14(1.838e-06)	0.14(1.830e-06)	0.14(1.872e-06)	0.14(1.902e-06)
Boston				
$\hat{\Pr}(k^* \in [1, \tilde{k}])$	0.999	1	1	1
Mean($\tilde{k}$)	36.753	34.988	34.52	33.962
$\hat{\Pr}(k \in [\tilde{k}+1, K] \mid k \neq k^*)$	0.76	0.772	0.775	0.779
Avg Evaluations	769.68	773.91	785.595	803.709
$\hat{\mathbb{E}}\{\tau_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau_{\hat{\lambda}^*}\})$	10.57(2.865e-02)	10.559(2.699e-02)	10.574(2.981e-02)	10.562(2.894e-02)
$\hat{\mathbb{E}}\{\tau^T_{\hat{\lambda}^*}\}(\hat{\mathrm{Var}}\{\tau^T_{\hat{\lambda}^*}\})$	10.412(2.866e-02)	10.411(2.819e-02)	10.419(2.947e-02)	10.417(2.627e-02)

is returned in almost 100% of cases for each value of P. For this reason the value of P has little effect on the mean and variation between hyperparameter optimization runs.

5.3.10 Normality Assumption

We now revisit the Normality-assumption made in Section 5.2.2. In particular, the procedure assumes normality in the residuals. In other words, it is assumed that the distribution of validation losses within each hyperparameter is normally distributed over runs. In practice, the ANOVA procedure is robust to slight violations to this assumption. Intuitively we do in fact expect stochasticity in model training to give rise to symmetric distributions

that are close to Normal. We check this assumption through quantile-quantile plots for the presented experiments shown in Figures 5.10a, 5.10b, and 5.10c and find that for the presented experiments sample quantiles are reasonably close to theoretical Normal quantiles.

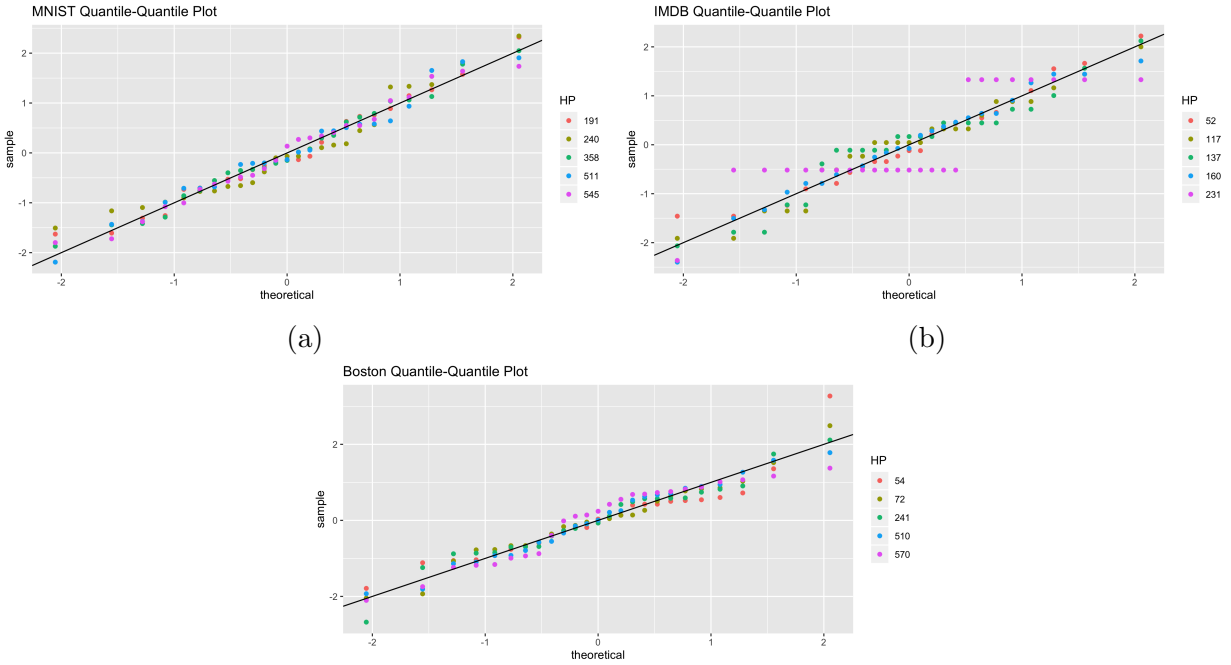


Figure 5.10: Normal quantile-quantile plot for validation losses across different runs of a sub-sample of 5 different hyperparameter settings in the MNIST CNN, IMDB LSTM, and Boston GBR experiments. In all cases the points are reasonably close to the straight line indicating that the empirical distributions are approximately Normal.

5.3.11 Supplementary Material

Code to generate the presented results is provided in R and available at `https://github.com/LarsHH/reproducible-hyperparameter-optimization-jcgs-2019/tree/submission`. Code to generate the precomputed hyperparameter searches is available in Python under the same code repository. Due to the computational cost of running the hyperparameter searches it is recommended to use the precomputed data sets included in the repository.

5.4 Discussion

The variation in model training during hyperparameter optimization poses a problem both for reproducibility of the hyperparameter search itself and comparisons of different methods, each optimized using hyperparameter search. In order to account for uncertainty in the training process replication is necessary. However, due to the computational complexity of most machine learning methods researchers may be less inclined to run large numbers of repetitions during hyperparameter optimization. To address this we have proposed a sequential testing and triaging procedure and show that for minimal increases in computation, optimal and reproducible hyperparameter values can be obtained. This is achieved by theoretically bounding type I error rates across the testing procedure.

In terms of applicability, the proposed methodology is unlikely to produce worse results than standard practices as demonstrated in the experiments. More importantly, we expect it to substantially improve results in scenarios where the variation within hyperparameter settings is close to variation between hyperparameter settings.

The presented method has the additional advantage that it is easy to estimate variation stemming from the sampling of the training data. Since one is already conducting multiple model training runs it is natural to re-sample the validation data set for each training run.

We acknowledge that our proposed approach may include increased computational burden associated with other optimization algorithms. Figure 5.9 demonstrates, however, that when the computational cost is matched with standard methods our proposed method achieves similar mean prediction errors and smaller variance. Therefore, as long as K is reasonably large, there is no actual increase in computation. Furthermore, while the algorithm in its current form does not have an anytime property a more sophisticated implementation could easily approach anytime performance. Future extensions of this work include the combination of the proposed method with model based hyperparameter suggestion such as

Bayesian optimization.

Chapter 6

Quantity vs. Quality: On Repetitions in Noisy Hyperparameter Optimization

6.1 Introduction

Many machine learning methods can show significant variation in performance between runs with different random seeds. Examples of this are the training of tree-based methods [Ganjisaffar et al., 2011], image generation algorithms [Lucic et al., 2018, Kurach et al., 2019], probabilistic unsupervised models [Locatello et al., 2018], as well as reinforcement learning algorithms [Henderson et al., 2018, Islam et al., 2017, Nagarajan et al., 2018, Colas et al., 2018]. Just like most machine learning algorithms, these methods have a number of hyperparameters that need to be configured for optimal performance.

Common hyperparameter optimization strategies typically assume noiseless observations or a negligible amount of noise. For example, most hyperparameter optimization algorithms

check different settings and then suggest the best observed. However, if observations vary drastically between seeds, the best observed configuration may not be the best configuration averaged across many random seeds. Most model-free hyperparameter optimization methods such as random search [Bergstra and Bengio, 2012], successive halving [Jamieson and Talwalkar, 2016, Li et al., 2016], and population-based training [Jaderberg et al., 2017] have no clear way of handling noisy observations other than evaluating multiple seeds and averaging. While model-based methods such as Bayesian optimization may be able to deal with noise in the model, they may still break down in the acquisition function [Picheny et al., 2013]. Using standard hyperparameter optimization methods for noisy optimization may therefore result in both suboptimal hyperparameter configurations and reduced reproducibility of the hyperparameter search.

While extensive research on noisy Bayesian optimization exists [Letham et al., 2019, Wu and Frazier, 2016], no clear evaluation has been done on the usefulness of these methods for noisy hyperparameter optimization. In particular, there are three important ways in which noisy hyperparameter search may differ from optimization on noisy synthetic benchmarks. First, objective functions in hyperparameter search are different and may be simpler than common optimization benchmarks. For example, the performance of an algorithm often marginally depends on a hyperparameter in a U-type way [Goodfellow et al., 2016]. Second, the noise may be heteroscedastic. In particular, there may be less noise near the optimum. Third, noise may be non-gaussian. The noise almost certainly has limited support, but in some cases may even be skewed or be close to uniform.

In order to deal with noise in hyperparameter optimization researchers have adopted the practice of averaging across seeds even during the search [Balandat et al., 2019, Falkner et al., 2018, Duan et al., 2016]. In this paper we want benchmark this practice against the use of common hyperparameter optimization techniques and techniques that have been developed specifically for noisy optimization. Our contributions are:

- A review of methods for hyperparameter optimization ranging from model-free multi-fidelity optimization (partial evaluations), to model-based optimization, to noisy optimization methods.
- An assessment of the performance of these methods on different noisy toy objective functions. In particular, we observe limitations of the different methods.
- A comparison of the presented methods on three noisy machine learning benchmarks across many hyperparameter optimization runs.
- Discussion of possible best practices and trade-offs for researchers when faced with hyperparameter optimization of noisy machine learning tasks.

6.2 Methods

6.2.1 Problem Statement

We define a learning algorithm by loosely relying on the framework introduced by Mitchell et al. [1997]. A machine learning algorithm $\mathcal{A}$ is given experience $\mathcal{E}$ and measured on a task $\mathcal{T}$ using a performance measure Ψ. Specifically we can say that $\mathcal{A}(\mathcal{E}) = g$ where g is a trained instance of the algorithm whose performance is measured as $\Psi(g, \mathcal{T})$. An example of this would be where the task $\mathcal{T}$ is to classify images of handwritten digits. Then $\mathcal{E}$ is an annotated dataset of images of handwritten digits, $\mathcal{A}$ might be the training of a neural network, g is the trained neural network (that is, the weights), and Ψ is the classification accuracy on a holdout dataset. This framework can equally be applied to cases where the training and evaluation are different from prediction tasks such as reinforcement learning or image generation. The learning algorithm $\mathcal{A}$ usually has associated hyperparameters $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$. This introduces dependencies of the trained model and the measured performance on these

hyperparameters, so $g_{\boldsymbol{\lambda}}$ and $\Psi(g_{\boldsymbol{\lambda}}, T)$, respectively. Hyperparameter optimization is then defined as the process of finding

$$\boldsymbol{\lambda}^* = \arg\max_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} \Psi(g_{\boldsymbol{\lambda}}, \mathcal{T}) \tag{6.1}$$

where $\mathcal{A}_{\boldsymbol{\lambda}}(\mathcal{E}) = g_{\boldsymbol{\lambda}}$. Now the learning algorithm $\mathcal{A}$ or the experience $\mathcal{E}$ may be stochastic. For example, $\mathcal{A}$ may be stochastic due to random weight initialization or $\mathcal{E}$ may be stochastic because it is a dynamic environment such as in reinforcement learning. In each case Ψ is random. We define the expectation across these factors $\mathbb{E}_{\mathcal{A},\mathcal{E}}[\Psi(g_{\boldsymbol{\lambda}}, T)] = f(\boldsymbol{\lambda})$ as the true objective function for the hyperparameter optimization. In practice, this would be the mean across infinitely many random seeds. The goal for the hyperparameter optimization is now to find

$$\boldsymbol{\lambda}^* = \arg\max_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} f(\boldsymbol{\lambda}). \tag{6.2}$$

Values of $\Psi(g_{\boldsymbol{\lambda}}, T)$ are noisy observations of $f(\boldsymbol{\lambda})$. We use $\Psi(\boldsymbol{\lambda})$ from this point on since the task $\mathcal{T}$ is usually fixed during the hyperparameter optimization. Equation 6.2 is more complicated so solve than the previous noiseless optimization of Equation 6.1 because $f(\boldsymbol{\lambda})$ is unavailable and and the distribution of $\Psi(\boldsymbol{\lambda})$ is generally unknown. In particular, this introduces two potential problems when applying regular hyperparameter optimization methods:

1. Choosing where to evaluate the next $\boldsymbol{\lambda}_i$ during the optimization may be impacted by observation noise.

2. Choosing $\boldsymbol{\lambda}^*$ at the end of the optimization may be impacted by noise.

6.2.2 Repeated Evaluation

A simple way to deal with observation noise in $\Psi(\boldsymbol{\lambda}_i)$ is to collect repeated evaluations and average the measurements $\bar{\Psi}(\boldsymbol{\lambda}_i) = \sum_{j=1}^{K} \Psi(\boldsymbol{\lambda}_i^j)$. The observed sample mean is consistent for $f(\boldsymbol{\lambda}_i)$ meaning that given a high enough number of samples K, $\bar{\Psi}(\boldsymbol{\lambda}_i) \approx f(\boldsymbol{\lambda}_i)$ and the optimization can be treated as noise-less. In practice, the number of repetitions is often constrained by a computational budget. For this reason, the question is then whether repetition is the optimal strategy or if there are more computationally efficient approaches. We now review a number of commonly used approaches to hyperparameter optimization and how these approaches are impacted by observation noise.

6.2.3 Model-free Optimization

Random search is a simple and popular model-free hyperparameter search algorithm [Bergstra and Bengio, 2012]. In particular, random search can often serve as a simple but robust baseline against more complex methods [Li and Talwalkar, 2019]. In random search hyperparameter configurations $\boldsymbol{\lambda}$ are sampled uniformly at random from the specified hyperparameter space $\boldsymbol{\Lambda}$. At the end of the optimization, $\boldsymbol{\lambda}^*$ is picked as the best observed configuration.

When applying random search in the noisy setting, the suggestion of hyperparameter configurations is not impacted. However, the choice of the final recommended configuration may be influenced. In the presence of noise the best observed setting may not be the on average best setting. Given the computational budget, this issue can be alleviated given repetition as described in Section 6.2.2. In experiments we denote random search as *Random Search*, *Random Search x3*, and *Random Search x5* for using one, three, and 5 evaluations, respectively.

Another more recent model-free approach to hyperparameter search is *Successive Halving*

(SHA) [Jamieson and Talwalkar, 2016], as well as its extensions *Hyperband* [Li et al., 2016] and *Asynchronous Successive Halving* (ASHA) [Li et al., 2018]. Hyperband has been shown to outperform random search and model-based search methods on certain benchmarks. Similar results are available for the ASHA algorithm. SHA operates by randomly sampling configurations and allocating increasingly larger budgets to well-performing configurations. In the beginning, a small budget is allocated to each configuration. All configurations are evaluated and the top $1/\eta$ is promoted. Then the budget for each promoted configuration is increased by a factor of η. This is repeated until the maximum per-configuration budget of R is reached. When tuning neural networks with SHA, the budget is often the number of training iterations. That way the algorithm can probe a lot of configurations without fully evaluating them. However, the budget can equally be the number of training examples or the number of random features.

We consider the application of SHA with the budget the number of training iterations in our noisy optimization problem. At intermediate stages, SHA prunes configurations based on their partial evaluations. Since we consider the case where full evaluations are noisy, partial evaluations are expected to be noisy, too. Hence, ranking and pruning may erroneously eliminate good configurations. SHA provides one or multiple completed configurations at the end of the optimization. Usually, the best observed is chosen from these. Just like for random search the best observed configuration may not necessarily the best configuration on average. In experiments we refer to SHA with training iterations as *ASHA* (we use the asynchronous version).

Alternatively, the SHA budget can be the number of repeated evaluations per configuration as demonstrated by Falkner et al. [2018]. Instead of deciding on a fixed number of repetitions per configuration as described in Section 6.2.2, SHA can start with one repetition per configuration and allocate more repetitions to well performing candidates. Given that the number of candidates described by $1/\eta$ is large enough at each stage, we would expect not

to eliminate the on average best configuration. In addition to that, at the end of the optimization we have multiple repetitions for each well performing configuration. This provides a better estimate of the performance on average and we are less likely to make a mistake by simply picking the best observed. In experiments we refer to SHA with repetitions as *ASHA-Repeat*.

Lastly, evolutionary algorithms have recently been successfully applied to hyperparameter optimization and architecture search for neural networks [Real et al., 2017]. One algorithm inspired by this paradigm is Population-Based Training (PBT) [Jaderberg et al., 2017]. In PBT a population of models is trained jointly with optimizing their hyperparameters. This happens in a two-stage approach where all population members are first trained for a number of training iterations. In the second stage the population evolves by replacing bad performing members with good ones and perturbing their hyperparameters. The two stages alternate until all population members are fully trained. Contrary to most other hyperparameter optimization approaches PBT allows schedules of hyperparameters via the perturbation mechanism. PBT has shown strong results in multiple domains including the training of reinforcement learning agents. Note however that there is a distinct difference in the goal of PBT compared to the other approaches discussed in this section. While most hyperparameter optimization methods aim to find the best performing hyperparameter setting, PBT aims to deliver an optimally trained model. Retraining that model is complicated because the schedule of hyperparameter settings needs to be traced back. Furthermore, if the training is noisy as in our case, the retrained model is not guaranteed to have similar performance. For these reasons we do not include PBT in our empirical evaluations.

6.2.4 Bayesian Optimization

Bergstra et al. [2011] and Snoek et al. [2012] demonstrated the feasibility of using Bayesian

optimization for hyperparameter optimization. Bayesian optimization (BO) is a model-based approach to global derivative-free optimization of blackbox functions. Given existing observations of configurations and associated objective values, BO fits a surrogate model from the search space to the objective. This model is usually faster to evaluate than the objective function itself. For this reason, a large number of predictions of objective values can be made. An *acquisition function* describes the utility of each prediction. By maximizing the acquisition function one can find the optimal next point to evaluate for the true objective function.

As described by Srinivas et al. [2009], it is common to use a Gaussian Process (GP) as a surrogate model for $f(\boldsymbol{\lambda})$. For the covariance kernel the Matern 5/2 kernel is a common choice. We assume that observations have normal noise, that is $\Psi(\boldsymbol{\lambda}) \sim \mathcal{N}(f(\boldsymbol{\lambda}), \tau^2)$. This offers a natural framework for noisy observations. Inference for the noise τ^2 can be made via maximization of the marginal likelihood. In particular, given a collection of observations $\mathcal{C} = \{(\boldsymbol{\lambda}_i, \Psi_i)\}_{i=1}^n$ and a GP prior, we can obtain the posterior distribution at hyperparameter setting $\boldsymbol{\lambda}$ as $f(\boldsymbol{\lambda})|\,\mathcal{C} \sim \mathcal{N}(\mu_n(\boldsymbol{\lambda}), \sigma_n^2(\boldsymbol{\lambda}))$. Assuming a zero prior, the posterior mean and variance are given by:

$$\mu_n(\boldsymbol{\lambda}) = \mathbf{k}(\boldsymbol{\lambda})^T(\mathbf{K} + \tau^2\mathbf{I})^{-1}$$
$$\sigma_n^2(\boldsymbol{\lambda}) = k(\boldsymbol{\lambda}, \boldsymbol{\lambda}) - \mathbf{k}(\boldsymbol{\lambda})^T(\mathbf{K} + \tau^2\mathbf{I})^{-1}\mathbf{k}(\boldsymbol{\lambda})$$

where $\mathbf{k}(\boldsymbol{\lambda})$ is the vector of covariances with all previous observations and $\mathbf{K}$ is the covariance matrix of the observations.

As mentioned before, Bayesian optimization chooses the next point to evaluate via the acquisition function. The most popular choice of acquisition function for hyperparameter

optimization is the expected improvement (EI) [Mockus et al., 1978]. The EI is given by

$$EI(\boldsymbol{\lambda}) = \mathbb{E}_{f(\boldsymbol{\lambda})|\mathcal{C}}[\mathrm{I}(\Psi_{min}, f(\boldsymbol{\lambda}))]. \tag{6.3}$$

where $\mathrm{I}(r, s) = \max(0, r - s)$ and $\Psi_{min} = \min_{1 \leq i \leq n} \Psi_i$ is the minimum observed validation loss. Part of why the EI is popular is that it can be computed analytically. However, in the noisy setting when $\mathrm{Var}(\Psi(\boldsymbol{\lambda}))$ is large, Ψ_{min} may just be low by chance and mislead the optimization. This can result in the over-exploitation around Ψ_{min} and the optimization getting "stuck" around this point. In experiments we refer to EI as *GPyOpt-EI* since we are using the GPyOpt [authors, 2016] implementation.

Letham et al. [2019] extended expected improvement to handle noisy observations by integrating over observations. Ignoring constraint handling also introduced by the authors, the noisy expected improvement (NEI) can be written as:

$$NEI(\boldsymbol{\lambda}) = \mathbb{E}_{f(\boldsymbol{\lambda}_{1:n})|\mathcal{C}}\{\mathbb{E}_{f(\boldsymbol{\lambda})|\mathcal{C}^*}[\mathrm{I}(\Psi^*_{min}, f(\boldsymbol{\lambda}))]\}. \tag{6.4}$$

where $\mathcal{C}^* = \{(\boldsymbol{\lambda}_i, \Psi^*(\boldsymbol{\lambda}_i)) : \Psi^*(\boldsymbol{\lambda}_i) \sim f(\boldsymbol{\lambda}_i)|\mathcal{C}\}_{i=1}^n$ is a fantasy dataset sampled from the posterior and Ψ^*_{min} is the minimum of the fantasy dataset. In practice, this approach fits a GP model with a residual variance term. Samples are drawn from the posterior at the observed points and new, noise-less GP models are fitted for each sample. For each noise-less fantasy model the EI is computed for a point $f(\boldsymbol{\lambda})$ under that model. The EI values for $\boldsymbol{\lambda}$ are then averaged across the fantasies to obtain the NEI. Balandat et al. [2019] simplify this approach by taking the expectation over the joint posterior:

$$qNEI(\boldsymbol{\lambda}) = \mathbb{E}_{f(\boldsymbol{\lambda}), f(\boldsymbol{\lambda}_{1:n})|\mathcal{C}}[\mathrm{I}(\min f(\boldsymbol{\lambda}_{1:n}), f(\boldsymbol{\lambda})]. \tag{6.5}$$

Letham et al. [2019] showed that NEI outperforms regular EI on objective functions with Gaussian noise and given the true variance. In our experiments we infer the noise term

τ^2. We further note that the true distribution of $\Psi(\boldsymbol{\lambda})$ may be non-gaussian which could impact the performance of NEI. We also perform experiments with the mean of repeated evaluations in which case a noise estimate will be provided and the distribution of the mean will be closer to normal due to the central limit theorem. In experiments we will use the qNEI implemented in BoTorch[1] to calculate the noisy expected improvement and refer to it as *BoTorch-qNEI*. The qNEI with repeated samples we will call *BoTorch-qNEI x3*.

An alternative to the EI criterion is the Lower(/Upper) Confidence Bound (LCB) criterion [Srinivas et al., 2009]. The LCB takes the form

$$LCB(\boldsymbol{\lambda}) = \mu_n(\boldsymbol{\lambda}) + \beta\sigma_n(\boldsymbol{\lambda}) \tag{6.6}$$

where β is a coefficient set by the user. This coefficient is sometimes termed the *exploration weight*. Tuning the exploration weight may improve the performance of LCB. However, since in practice the computational budget to tune this parameter is rarely available, we use a default value of $\beta = 2$. While LCB has been shown to be outperformed by EI on noise-less objective functions [Snoek et al., 2012], contrary to EI it is not impacted by noise. This is due to the fact that it only depends on the posterior mean and variance. In all experiments we refer to LCB as *GPyOpt-LCB*.

We note that there is a much larger choice of acquisition functions than the ones presented in this section. In particular, the knowledge gradient [Scott et al., 2011, Wu and Frazier, 2016] is an acquisition function that, while difficult to compute, can handle noise and has been shown to outperform the presented acquisition functions. Furthermore, entropy-search-based methods [Hennig and Schuler, 2012, Hernández-Lobato et al., 2014, Wang and Jegelka, 2017] also naturally handle noise. We leave the evaluation of these methods to future work.

While the choice of acquisition function determines how to choose the next point, one still

[1] `https://botorch.org/`

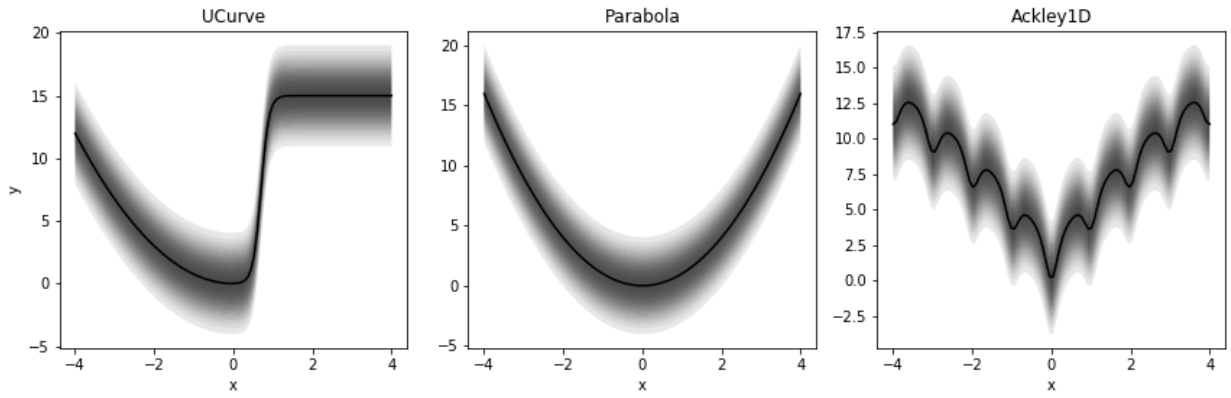


Figure 6.1: Toy objective functions. The U-curve and Parabola functions attempt to resemble marginal objective functions often observed for hyperparameters. The Ackley1D is a common optimization benchmark function. Each plot illustrates Gaussian noise with variance 3.

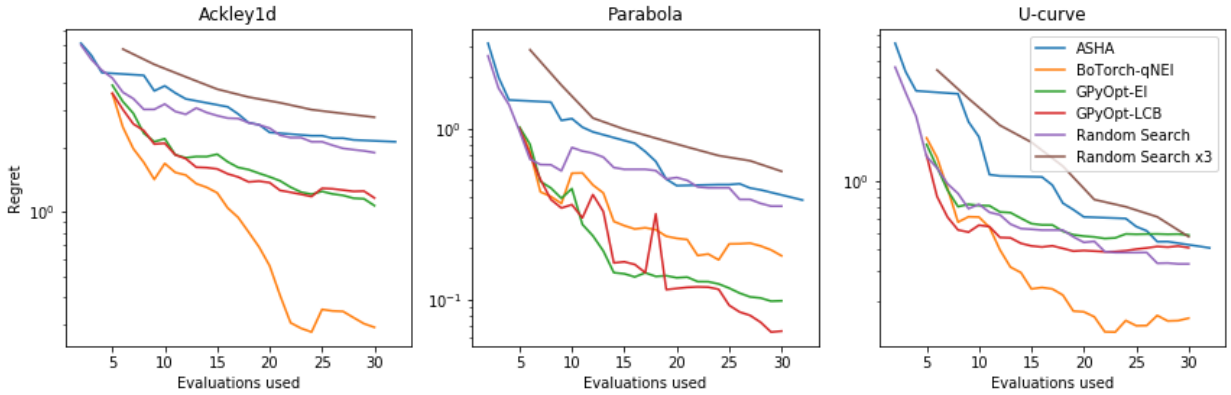


Figure 6.2: Best objective value against the number of evaluations used for each method.

needs to pick a best setting at the end of the optimization. In the noise-less case this is often done by picking the best observed $\boldsymbol{\lambda}_i$. Frazier [2018] suggests to rely on the surrogate model and pick the best predicted $\boldsymbol{\lambda}$ from that model. Since this approach is robust to noise we use the predicted best $\boldsymbol{\lambda}$ in all experiments. To make this feasible in multi-dimensional tasks we find the best predicted by optimizing an LCB acquisition with $\beta = 0$, thereby utilizing existing optimization tools in BoTorch and GPyOpt.

6.3 Experiments

6.3.1 Toy Functions

We first evaluate the methods in Section 6.2 on three toy objective functions: a parabola with a smooth step (Figure 6.1 left), a parabola (Figure 6.1 middle), and the Ackley 1-D function (Figure 6.1 right). To make the objective function noisy a Gaussian random variable is added with mean zero and variance 3. We compare three Bayesian optimization methods (BoTorch-qNEI, GPyOpt-EI, GPyOpt-LCB) as well as Random search, Random Search x3, and ASHA-Repeat that is successive halving applied to repetitions. Figure 6.2 shows the average regret for 100 optimization runs for each method. The regret is defined as the absolute difference between the best obtained value and the function minimum. Over a budget of 30 function evaluations it can be seen that for the U-curve and Ackley1D functions BoTorch-qNEI performs best on average. For the Parabola GPyOpt-EI and GPyOpt-LCB perform best. Variances between runs are shown in Figure 6.3 and exhibit similar trends. For Ackley1D and U-curve BoTorch-qNEI shows the smallest variance. For the parabola GPyOpt-EI and GPyOpt-LCB have the smallest variance across runs. We also conduct the same experiments with uniform and skewed noise distributions. While LCB and NEI still perform best, there is no significant difference in the performance of the two. We hypothesize that NEI relies heavily on the Gaussian noise assumption when integrating out the observations and may therefore be more susceptible to deviations from that assumption.

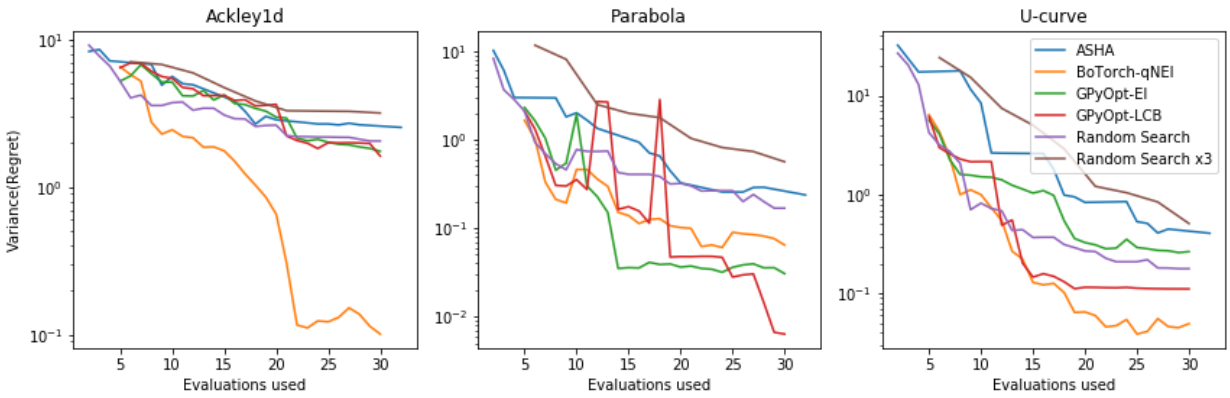


Figure 6.3: Variance of the regret across one hundred repeated optimizations against function evaluations used.

6.3.2 Reinforcement Learning

Cartpole Balancing

Next, we compare the introduced algorithms on a reinforcement learning task. The cartpole task [Barto et al., 1983] is a continuous control problem that requires to balance a pole on a friction-less cart. The cart can be moved left or right. A screenshot of the task is shown in Figure 6.4.

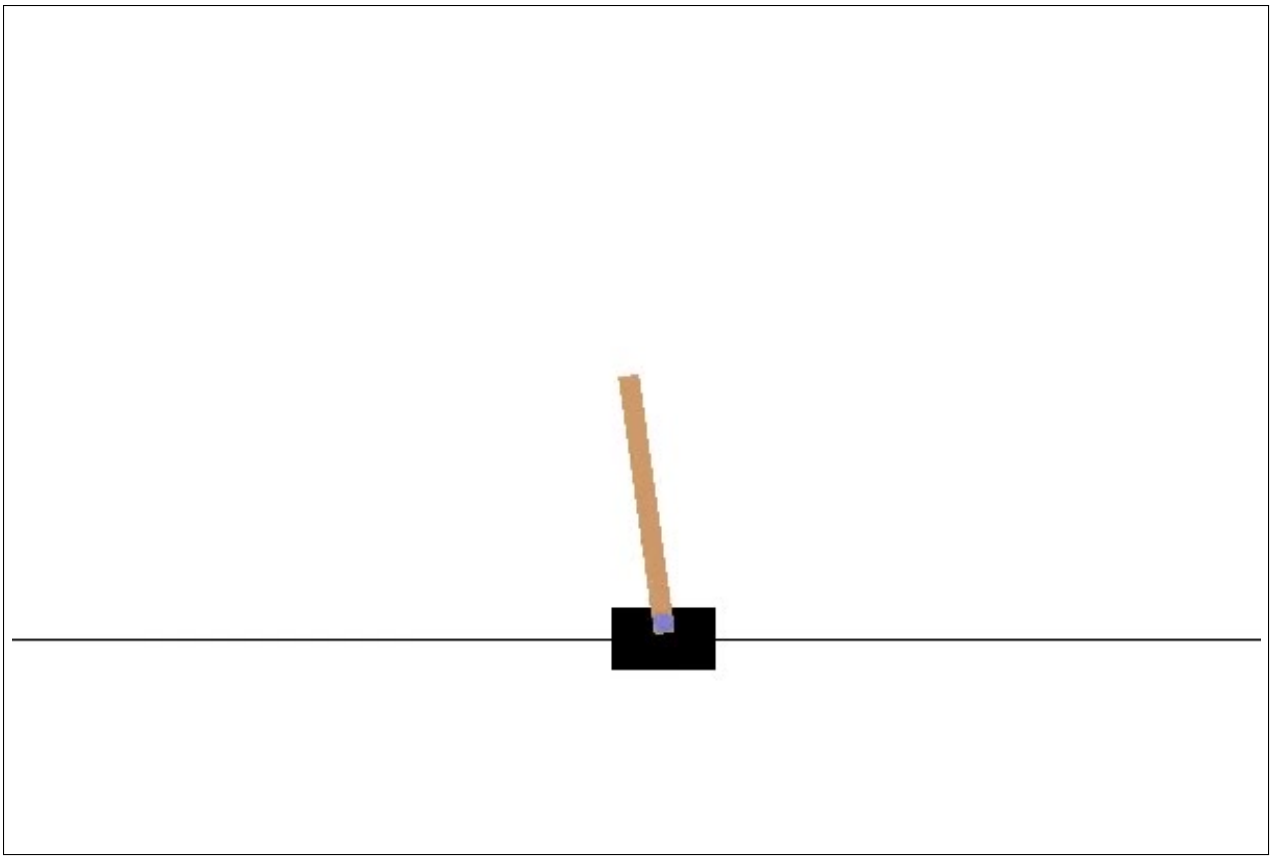


Figure 6.4: Screenshot of the cartpole task.

The cartpole task has been used as a benchmark for reinforcement learning (RL) algorithms [Duan et al., 2016] and most recently to benchmark hyperparameter optimization for reinforcement learning [Falkner et al., 2018, Balandat et al., 2019]. In RL it is common to use multiple observations for hyperparameter tuning [Duan et al., 2016, Falkner et al., 2018, Balandat et al., 2019]. This is due to the high noise that is inherent in the training of RL algorithms. In particular, training depends on the interaction with the environment. The RL agent may get off to a bad start and may not be able to recover from it. In this experiment we want to address whether the use of repeated samples is an optimal strategy. Here we tune the hyperparameters of the proximal policy optimization algorithm by Schulman et al. [2017] implemented in [Hill et al., 2018] for the Cartpole-v1 environment from [Brockman et al., 2016]. We tune the log learning rate (uniform $[-5, -1]$) as well as the entropy coeffi-

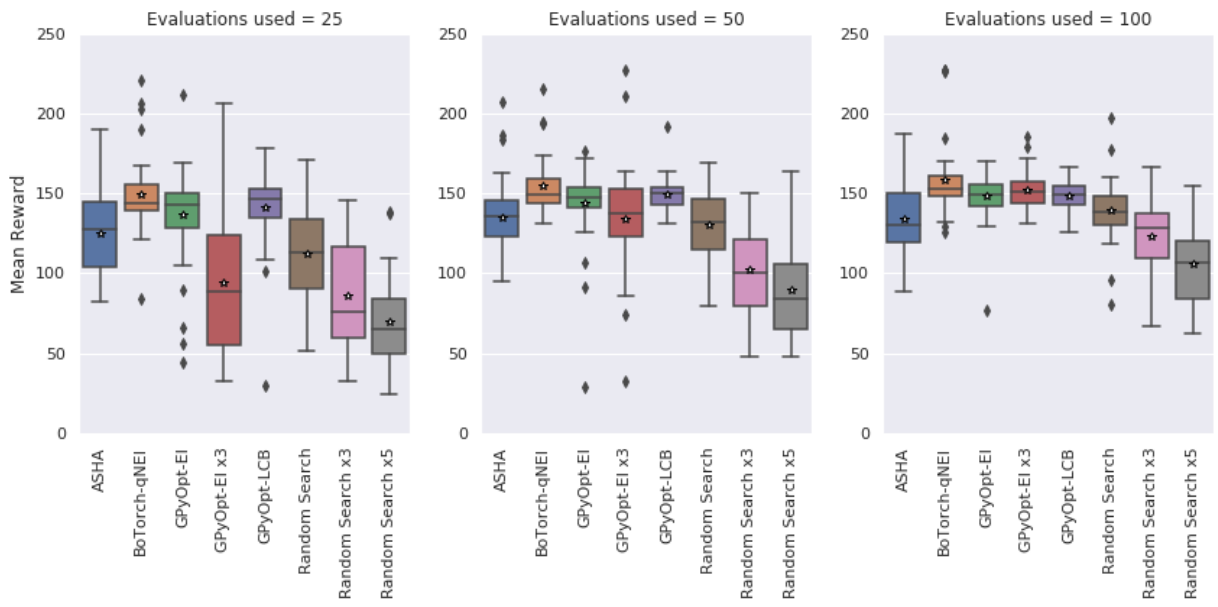


Figure 6.5: Boxplots showing the distribution of mean rewards across training over 40 optimization runs on the Cartpole environment using PPO2 (higher is better).

cient (uniform $[0, 1]$), discount factor (uniform $[0, 1]$), and likelihood ratio clipping (uniform $[0, 1]$). The RL agent is trained for 30000 steps. RL agents on the cartpole task are often compared in terms of the number of episodes until a mean reward of at least 195 is obtained over a one hundred episode run. However, this measure is not optimal for hyperparameter optimization since some agents may not converge.

Alternatively, RL agents are often compared via plotted reward curves. The qualitative comparison of two reward curves can be made quantitative by considering the area under the reward curve. For a constant number of steps the area under the curve is proportional to the mean reward across the training. We thus choose the mean reward across training as the hyperparameter optimization objective.

The hyperparameter optimization itself is given a budget of 100 function evaluations. After that the configuration returned by the optimization is trained 20 times and the mean across those runs is taken as the optimization outcome. The hyperparameter optimization itself is repeated 40 times. Results are shown in Figure 6.5 (right). The same evaluation procedure

Table 6.1: Mean rewards across training and across hyperparameter optimization runs for different budgets of function evaluations (higher is better). Cell values are averaged across 20 training runs of the best found agent and across 40 hyperparameter optimization runs.

Evaluations	25	50	100
ASHA	125.2	135.0	134.4
BoTorch-qNEI	**149.7**	**155.2**	**158.3**
GPyOpt-EI	136.5	143.6	148.5
GPyOpt-EI x3	93.7	133.5	151.9
GPyOpt-LCB	141.1	149.7	148.7
Random Search	112.0	130.3	139.1
Random Search x3	85.6	102.6	123.0
Random Search x5	69.6	89.1	105.7

is also done after 25 and 50 function evaluations (Figure 6.5 left and middle, respectively). Table 6.1 also shows means for the corresponding budgets and methods.

From Figure 6.5 and Table 6.1 it can be seen that *BoTorch-qNEI* has the best results across all budgets. However, it is notable that *GPyOpt-LCB* is only marginally worse. *GPyOpt-EI* catches up after 50 evaluations and *GPyOpt-EI x3* after 100 evaluations. This seems to indicate that the EI acquisition does perform worse due to noise. For EI with 3 replicates it seems that the computation allocated to replicates does not help, especially on a budget of 25 evaluations. One would expect that *Random Search* eventually catches up with the model-based methods. This is however not the case here. This may be due to the fact that *Random Search* still needs more evaluations. It may also be because *Random Search* picks the best observed configuration which may be inaccurate. *Random Search x3* and *Random Search x5* seem to lack a sufficient numbber of observations. Finally, ASHA, here applied to early stopping of configurations seems to stagnate after 50 evaluations. We believe that the pruning of configurations after partial training may be subject to too much noise to yield an accurate hyperparameter optimization. Finally, we also note that *BoTorch-qNEI* has some outliers with significantly better mean rewards. We emphasize here that these are not lucky training runs but lucky optimization runs. It seems therefore that there is still room for improvement on the benchmarked algorithms.

Inverted Pendulum Swing-up

As a second experiment from the domain of reinforcement learning, we consider the Inverted Pendulum Swing-up task. Just like the Cartpole balancing task from Section 6.3.2, the Inverted Pendulum Swing-up task is a classic problem from the control theory literature. A screenshot of the environment is shown in Figure 6.6. The friction-less pendulum (red) starts in a random position with random velocity and the goal is to keep it standing up. To achieve this the agent can apply negative or positive effort to the joint. The reward is a function of how upright and still the pendulum is.

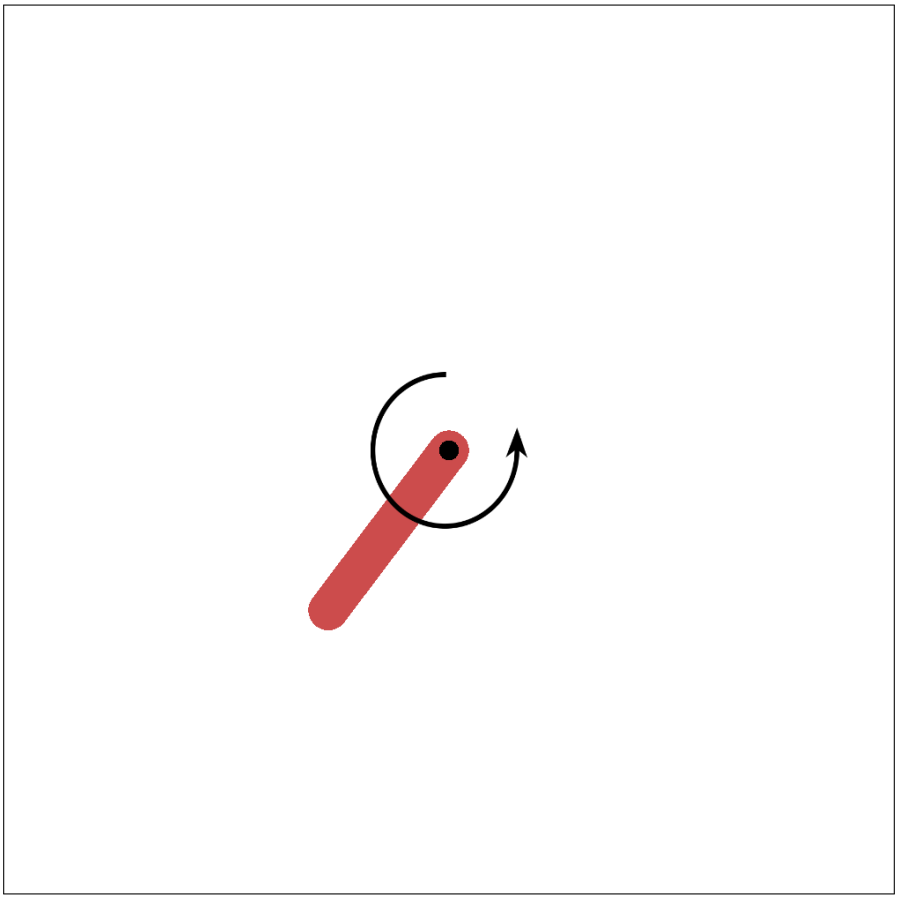


Figure 6.6: Screenshot of the Inverted Pendulum Swing-up task.

In comparison to the Cartpole balancing task, this problem is considered more difficult. This means it will generally take more training time-steps and algorithm tuning to achieve a high

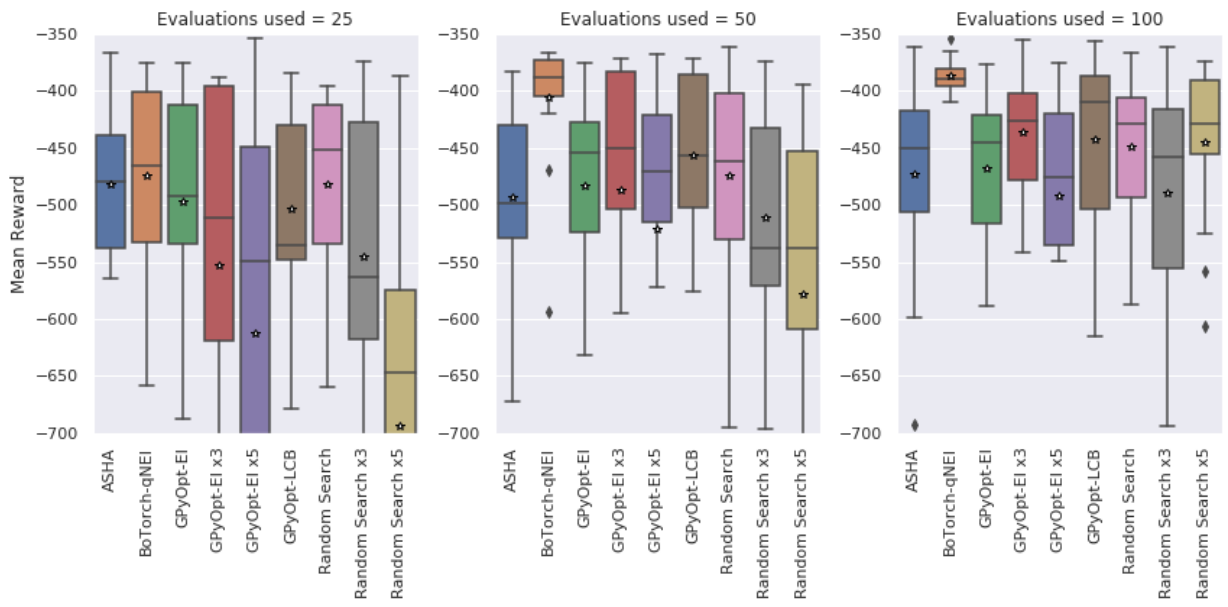


Figure 6.7: Boxplots showing the distribution of mean rewards across training over 15 hyperparameter optimization runs on the Inverted Pendulum Swing-up task using PPO2 (higher is better).

reward on this problem.

We again apply the proximal policy algorithm by Schulman et al. [2017]. This time the hyperparameter search-space consists of the log-learning-rate (uniform $[-5, -1]$) and the log-entropy-coefficient (uniform $[-5, 0]$). The agent is given 2×10^6 timesteps for training. The hyperparameter optimization objective is, as before, the mean reward across training.

Figure 6.7 shows boxplots of mean rewards across training over 15 hyperparameter optimization runs. It can be seen that after 25 evaluations all algorithms perform similarly except for algorithms with three or 5 repetitions which perform worse. After 50 evaluations *BoTorch-qNEI* performs significantly better than all other algorithms. This trend continues after 100 evaluations. These results are also reflected in the means shown in Table 6.2. The next best performers after *BoTorch-qNEI* are *GPyOpt-EI x3* and *GPyOpt-LCB* - the former better in terms of the mean and the latter better in terms of the median.

Table 6.2: Mean rewards for the Inverted Pendulum Swing-up task across training and across hyperparameter optimization runs for different budgets of function evaluations (higher is better). Cell values are averaged across 20 training runs of the best found agent and across 15 hyperparameter optimization runs.

Evaluations	25	50	100
ASHA	-482.0	-493.7	-472.6
BoTorch-qNEI	**-474.7**	**-406.2**	**-387.3**
GPyOpt-EI	-497.2	-483.5	-467.5
GPyOpt-EI x3	-553.1	-487.6	-436.9
GPyOpt-EI x5	-612.2	-520.9	-491.9
GPyOpt-LCB	-503.7	-456.2	-442.4
Random Search	-481.8	-474.2	-449.3
Random Search x3	-546.1	-511.4	-490.2
Random Search x5	-694.0	-578.2	-444.6

6.3.3 Image Generation

In this experiment we turn to the task of image generation using a Generative Adversarial Network (GAN). GANs are powerful generative models that are trained via a discriminator that classifies whether examples stem from the dataset or were sampled from the generative model. Both the discriminator and the generative model are trained via stochastic gradient descent. We use the non-saturating GAN introduced by Goodfellow et al. [2014] on the MNIST [Deng, 2012] dataset. Our objective is the Fréchet Inception Distance (FID) as introduced by Heusel et al. [2017]. While the authenticity of samples can only truly be judged by humans, different scores have been proposed for the purpose of automation and comparison across studies. For the FID, samples are first embedded into a continuous representation using a different neural network (usually the Inception model). Then the mean and covariance are estimated for generated and real data. The Fréchet distance between the two is the FID. We use the code implementation from the CompareGAN package[2] with 15625 training steps. For each training we compute the FID once at the end of the training. We tune the Adam [Kingma and Ba, 2014] learning rate over a range of $[1e-4, 1e-1]$

[2] `https://github.com/google/compare_gan`

on a logarithmic scale. This is similar to the "narrow" hyperparameter search conducted by Lucic et al. [2018]. We restrict the number of allowed evaluations - that is full training of the GAN - to 21 for each search algorithm. For the ASHA algorithm we use $R = 9$, $\eta = 3$, and let a resource of 1 correspond to 1200 training steps (resulting in a maximum of 15600 training steps). We also restrict ASHA to a maximum of 10 completed models. This corresponds to approximately 90 tried configurations at the bottom rung and matches the total computation used by the other methods. To estimate the mean across training runs for a specific configuration we use a surrogate GP that has been fitted on a large number of fully trained models across the hyperparameter range.

Figure 6.8 shows boxplots of the FID at the end of the optimization for three Bayesian optimization methods and ASHA over 16 hyperparameter optimization runs. ASHA, GPyOpt-LCB, and BoTorch-qNEI are all able to obtain the minimum for at least one run. Among these three, GPyOpt-LCB and BoTorch-qNEI tie on the median, but GPyOpt-LCB has a lower first quartile. ASHA has the highest variation overall but slightly lower median that GPyOpt-EI.

From exploratory runs we have found that the GAN optimization does have lower noise compared to previous examples. This justifies why ASHA is able to perform well.

6.4 Discussion

After evaluating a number of hyperparameter optimization methods on both synthetic functions and real machine learning hyperparameter optimization tasks we have found that model-based search in the form of GP-based Bayesian optimization performs best in terms of the average achieved across optimization runs. We believe that this is due to the fact that the GP estimates the mean function whose minimum is our optimization target. Among

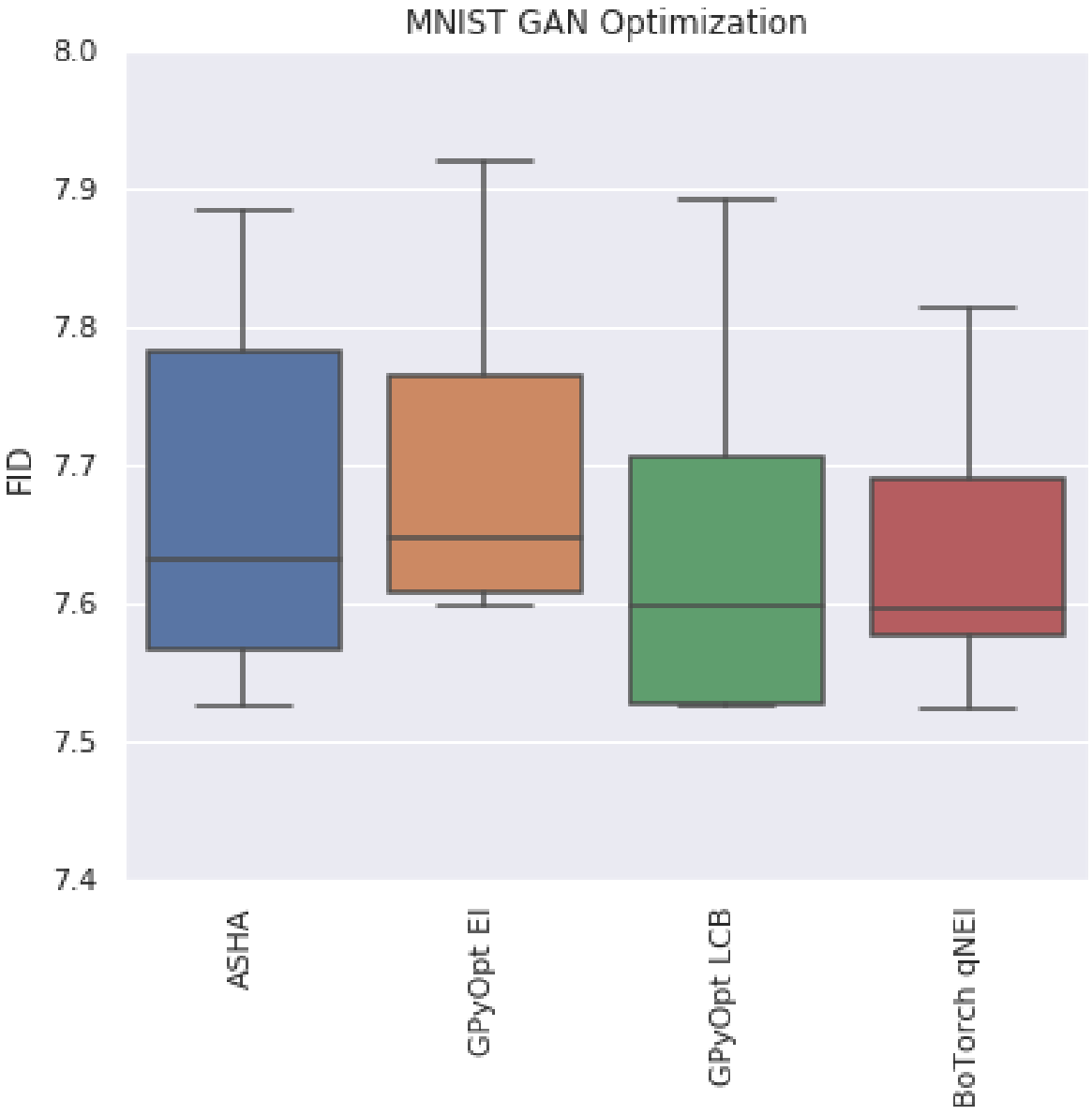


Figure 6.8: FID for learning rate optimization of the MNIST non-saturating GAN. Compared are three Bayesian optimization methods and the ASHA algorithm after 21 full evaluations of the GAN. Boxplots are over 16 runs of each hyperparameter optimization.

Bayesian optimization methods we find that the lower confidence bound as well as noisy expected improvement perform best. This is intuitive since these acquisition functions are able to handle noise better than the expected improvement acquisition function. Using replicates with expected improvement does not provide better results. We also find that successive halving applied to early stopping shows a large variation in outcomes and on-average does not perform as well as model-based search. For random search we also do not find that repeated evaluation of the same configuration helps when the computational budget is fixed. However, we believe that this may change if more function evaluations were allowed. Overall, our conclusion is that the best strategy for noisy hyperparameter optimization is to use a model-based search. The lower confidence bound performed well in experiments, is easy to implement, and is available in most Bayesian optimization packages. If available, more advanced criteria such as the noisy expected improvement may provide further improvements.

Chapter 7

Future Work

To conclude the thesis we illustrate directions of future work in the domains of deep learning applied to neutrino physics, the software Sherpa as developed in Chapter 4, and the problem of hyperparameter optimization when observations are noisy.

Future work in the domain of neutrino physics focuses around the next generation neutrino experiment Deep Underground Neutrino Experiment (DUNE). The goal for DUNE is to model energy reconstruction and additional tasks via deep learning. This includes the reconstruction of the direction of the electron shower as well as the reconstruction of the vertex - the point where the incoming neutrino interacted with the matter in the detector. Initial efforts towards this goal are described in Seong et al..

Future work also includes further development of Sherpa, the hyperparameter tuning software illustrated in Chapter 4. One of Sherpa's goals is to support a large variety of hyperparameter optimization algorithms. This inevitably requires the continuous updating, addition, or replacement of search algorithms in the software. Sherpa also aims to provide as much insight to the user as possible. For this reason, an important direction for future work is the extension of its visualizations. In particular, marginal plots may be useful to understand the

relative importance of individual hyperparameters. Finally, another direction for future work would be to increase user friendliness of Sherpa by turning it into an online service. In this case, the user would only provide the trial-script as defined in Chapter 4. The optimization itself would be defined via a web application. This would eliminate any need to deal with concurrency on the user-side and make it easier to set up Sherpa. Potential challenges would be how to finance servers running the web application and search algorithms.

Lastly, in the domain of hyperparameter optimization for noisy machine learning algorithms future work consists of further exploration of existing methods applied to this problem, as well as specific modeling for this task. Chapter 6 explored the application of successive halving to noisy hyperparameter optimization. Results showed that the model-free partial evaluation of hyperparameter settings may be too noisy to successfully prune under-performing settings. In the case of successive halving, the elimination of settings is based on quantiles of the objective. As described in Chapter 2, other works eliminate settings based on a model such as a Gaussian Process. In that case, the model may be able to delineate the noise from the mean trend just as we found is the case when modelling the hyperparameter space. It would be important to explore whether such methodology would perform well on benchmarks and allow speed-ups even for noisy objectives. Another direction for future work is to better model the observations of the objective, specifically with regards to the noise distribution. As mentioned in Chapter 6, NEI heavily depends on the assumption of Gaussian noise and its performance may deteriorate if the observation noise is significantly non-Gaussian. Observation noise from machine learning objectives may however be skewed or uniform. Therefore, it may be possible to improve the performance of NEI by relying less on the Gaussian assumption when suggesting the next hyperparameter setting.

Bibliography

R Acciarri, C Adams, R An, J Asaadi, M Auger, L Bagby, B Baller, G Barr, M Bass, F Bay, et al. Convolutional neural networks applied to neutrino events in a liquid argon time projection chamber. *Journal of instrumentation*, 12(3):P03011, 2017.

MA Acero, P Adamson, L Aliaga, T Alion, V Allakhverdian, N Anfimov, A Antoshkin, E Arrieta-Diaz, A Aurisano, A Back, et al. New constraints on oscillation parameters from ν e appearance and ν μ disappearance in the nova experiment. *Physical Review D*, 98(3):32012, 2018.

P Adamson, C Ader, M Andrews, N Anfimov, I Anghel, K Arms, E Arrieta-Diaz, A Aurisano, DS Ayres, C Backhouse, et al. First measurement of electron neutrino appearance in nova. *Physical review letters*, 116(15):151806, 2016.

P Adamson, L Aliaga, D Ambrose, Nikolay Anfimov, A Antoshkin, E Arrieta-Diaz, K Augsten, A Aurisano, C Backhouse, M Baird, et al. Constraints on oscillation parameters from ν e appearance and ν μ disappearance in nova. *Physical review letters*, 118(23):231801, 2017.

Forest Agostinelli, Matthew Hoffman, Peter Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014.

Sea Agostinelli, John Allison, K al Amako, John Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G 2 Barrand, et al. Geant4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631. ACM, 2019.

L Aliaga, M Kordosky, T Golan, O Altinok, L Bellantoni, A Bercellie, M Betancourt, A Bravar, H Budd, MF Carneiro, et al. Neutrino flux predictions for the numi beam. *Physical Review D*, 94(9):92005, 2016.

Costas Andreopoulos, A Bell, D Bhattacharya, F Cavanna, J Dobson, S Dytman, H Gallagher, P Guzowski, R Hatcher, P Kehayias, et al. The genie neutrino monte carlo gen-

erator. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 614(1):87–104, 2010.

Costas Andreopoulos, Christopher Barry, Steve Dytman, Hugh Gallagher, Tomasz Golan, Robert Hatcher, Gabriel Perdue, and Julia Yarba. The genie neutrino monte carlo generator: physics and user manual. *arXiv preprint arXiv:1510.05494*, 2015.

Peter Armitage, CK McPherson, and BC Rowe. Repeated significance tests on accumulating data. *Journal of the Royal Statistical Society: Series A (General)*, 132(2):235–244, 1969.

A Aurisano, C Backhouse, R Hatcher, N Mayer, J Musser, R Patterson, R Schroeter, and A Sousa. The nova simulation chain. In *Journal of Physics: Conference Series*, volume 664, page 72002. IOP Publishing, 2015.

A Aurisano, A Radovic, D Rocco, A Himmel, MD Messier, E Niner, G Pawloski, F Psihas, Alexandre Sousa, and P Vahle. A convolutional neural network neutrino event classifier. *Journal of Instrumentation*, 11(9):P09001, 2016.

The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

M Baird, Jianming Bian, M Messier, E Niner, D Rocco, and K Sachdev. Event reconstruction techniques in nova. In *Journal of Physics: Conference Series*, volume 664, page 72035. IOP Publishing, 2015.

Michael David Baird. An analysis of muon neutrino disappearance from the numi beam using an optimal track fitter. 2015.

Maximilian Balandat, Brian Karrer, Daniel R Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. Botorch: Programmable bayesian optimization in pytorch. *arXiv preprint arXiv:1910.06403*, 2019.

Pierre Baldi and Peter Sadowski. The dropout learning algorithm. *Artificial intelligence*, 210:78–122, 2014.

Pierre Baldi and Peter J Sadowski. Understanding dropout. In *Advances in neural information processing systems*, pages 2814–2822, 2013.

Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

Pierre Baldi, Kevin Bauer, Clara Eng, Peter Sadowski, and Daniel Whiteson. Jet substructure classification in high-energy physics with deep neural networks. *Physical Review D*, 93(9):94034, 2016.

Pierre Baldi, Jianming Bian, Lars Hertel, and Lingge Li. Improved energy reconstruction in nova with regression convolutional neural networks. *Physical Review D*, 99(1):12011, 2019.

L Banken, HU Burger, S Kristiansen, and M Pasquier. The closed test procedure.

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

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

James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20. Citeseer, 2013.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

Jianming Bian. First results of ν_e appearance analysis and electron neutrino identification at nova. *arXiv preprint arXiv:1510.05708*, 2015.

Bernd Bischl, Jakob Richter, Jakob Bossek, Daniel Horn, Janek Thomas, and Michel Lang. mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions. 2017. URL `http://arxiv.org/abs/1703.03373`.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Zhuo Cao, Yabo Dan, Zheng Xiong, Chengcheng Niu, Xiang Li, Songrong Qian, and Jianjun Hu. Convolutional neural networks for crystal material property prediction using hybrid orbital-field matrix and magpie descriptors. *Crystals*, 9(4):191, 2019.

Kai Chang. Parallel coordinates. `https://github.com/syntagmatic/parallel-coordinates`, 2019.

François Chollet et al. Keras. `https://keras.io`, 2015.

François Chollet et al. Keras: The python deep learning library. *Astrophysics Source Code Library*, 2018.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

S Delaquis, MJ Jewell, I Ostrovskiy, M Weber, T Ziegler, J Dalmasson, LJ Kaufman, T Richards, JB Albert, G Anton, et al. Deep neural networks for energy and position reconstruction in exo-200. *Journal of Instrumentation*, 13(8):P08023, 2018.

Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

Scott S Emerson, John M Kittelson, and Daniel L Gillen. Frequentist evaluation of group sequential clinical trial designs. *Statistics in medicine*, 26(28):5047–5080, 2007.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. proceedings of the second international conference on knowledge discovery and data mining, 1996.

W.A. Falcon. Test tube. `https://github.com/williamfalcon/test-tube`, 2017.

Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

Leandro AF Fernandes and Manuel M Oliveira. Real-time line detection through an improved hough transform voting scheme. *Pattern recognition*, 41(1):299–314, 2008.

Matthias Feurer and Frank Hutter. Hyperparameter optimization. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *AutoML: Methods, Sytems, Challenges*, chapter 1, pages 3–37. Springer, December 2018. To appear.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

Yasser Ganjisaffar, Rich Caruana, and Cristina Videira Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 85–94. ACM, 2011.

Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

Anushree Ghosh. Improved vertex finding in the minerva passive target region with convolutional neural networks and deep adversarial neural network. *PoS*, page 154, 2018.

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

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.

Laura Gustafson. Bayesian tuning and bandits: An extensible, open source library for automl. M. eng thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2018. URL `https://dai.lids.mit.edu/wp-content/uploads/2018/05/Laura_MEng_Final.pdf`.

Miklos Gyulassy and Magnus Harlander. Elastic tracking and neural network algorithms for complex pattern recognition. *Computer Physics Communications*, 66(1):31–46, 1991.

Helwig Hauser, Florian Ledermann, and Helmut Doleisch. Angular brushing of extended parallel coordinates. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 127–130. IEEE, 2002.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(Jun):1809–1837, 2012.

José Miguel Hernández-Lobato, Matthew W Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. In *Advances in neural information processing systems*, pages 918–926, 2014.

Lars Hertel, Julian Collado, Peter Sadowski, and Pierre Baldi. Sherpa: Hyperparameter optimization for machine learning models. 2018.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems*, pages 6626–6637, 2017.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. 1994.

Matthew Hutson. Artificial intelligence faces reproducibility crisis, 2018.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

Christian Igel, Thorsten Suttorp, and Nikolaus Hansen. A computational efficient covariance matrix update and a (1+ 1)-cma for evolution strategies. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 453–460. ACM, 2006.

Alfred Inselberg and Bernard Dimsdale. Parallel coordinates for visualizing multi-dimensional geometry. In *Computer Graphics 1987*, pages 25–44. Springer, 1987.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

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

Neal Jean, Marshall Burke, Michael Xie, W Matthew Davis, David B Lobell, and Stefano Ermon. Combining satellite imagery and machine learning to predict poverty. *Science*, 353(6301):790–794, 2016.

Christopher Jennison and Bruce W Turnbull. *Group sequential methods with applications to clinical trials*. Chapman and Hall/CRC, 1999.

Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956. ACM, 2019.

Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R. Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning Hyperparameters without Grad Students: Scalable and Robust Bayesian Optimisation with Dragonfly. *arXiv preprint arXiv:1903.06694*, 2019.

Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. 2016. *URL http://cs231n. github. io*, 2017.

Keras. Keras optimizers. `https://keras.io/optimizers/`, 2019. Accessed: 2019-03-27.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

John M Kittelson and Scott S Emerson. A unifying family of group sequential test designs. *Biometrics*, 55(3):874–882, 1999.

A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, December 2017.

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

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. 2016b.

Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

Raghuram Krishnapuram and James M Keller. A possibilistic approach to clustering. *IEEE transactions on fuzzy systems*, 1(2):98–110, 1993.

Tammo Krueger, Danny Panknin, and Mikio L Braun. Fast cross-validation via sequential testing. *Journal of Machine Learning Research*, 16(1):1103–55, 2015.

Karol Kurach, Mario Lučić, Xiaohua Zhai, Marcin Michalski, and Sylvain Gelly, editors. *A Large-Scale Study on Regularization and Normalization in GANs*, 2019. URL `https://arxiv.org/abs/1807.04720`.

Paras Lakhani, Daniel L Gray, Carl R Pett, Paul Nagy, and George Shih. Hello world deep learning in medical imaging. *Journal of digital imaging*, 31(3):283–289, 2018.

Zachary Langford, Logan Eisenbeiser, and Matthew Vondal. Robust signal classification using siamese networks. In *Proceedings of the ACM Workshop on Wireless Security and Machine Learning*, pages 1–5. ACM, 2019.

Benjamin Letham, Brian Karrer, Guilherme Ottoni, Eytan Bakshy, et al. Constrained bayesian optimization with noisy experiments. *Bayesian Analysis*, 14(2):495–519, 2019.

Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 18(1):6765–6816, 2016.

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

Zachary C Lipton and Jacob Steinhardt. Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*, 2018.

Francesco Locatello, Stefan Bauer, Mario Lucic, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem. Challenging common assumptions in the unsupervised learning of disentangled representations. *arXiv preprint arXiv:1811.12359*, 2018.

Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. In *Advances in neural information processing systems*, pages 700–709, 2018.

Ruth Marcus, Peritz Eric, and K Ruben Gabriel. On closed testing procedures with special reference to ordered analysis of variance. *Biometrika*, 63(3):655–660, 1976.

Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

Tom M Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

Prabhat Nagarajan, Garrett Warnell, and Peter Stone. The impact of nondeterminism on reproducibility in deep reinforcement learning. 2018.

Evan Niner. Observation of electron neutrino appearance in the numi beam with the nova experiment. 2015.

Peter C O'Brien and Thomas R Fleming. A multiple testing procedure for clinical trials. *Biometrics*, pages 549–556, 1979.

Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4206-3. doi: 10.1145/2908812.2908918. URL http://doi.acm.org/10.1145/2908812.2908918.

Randal S. Olson, Ryan J. Urbanowicz, Peter C. Andrews, Nicole A. Lavender, La Creis Kidd, and Jason H. Moore. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*, chapter Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pages 123–137. Springer International Publishing, 2016b. ISBN 978-3-319-31204-0. doi: 10.1007/978-3-319-31204-0_9. URL http://dx.doi.org/10.1007/978-3-319-31204-0_9.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct): 2825–2830, 2011.

Victor Picheny, Tobias Wagner, and David Ginsbourger. A benchmark of kriging-based infill criteria for noisy optimization. *Structural and Multidisciplinary Optimization*, 48(3): 607–626, 2013.

Stuart J Pocock. Group sequential methods in the design and analysis of clinical trials. *Biometrika*, 64(2):191–199, 1977.

Bruno Pontecorvo. Neutrino experiments and the problem of conservation of leptonic charge. *Sov. Phys. JETP*, 26(984-988):165, 1968.

Fernanda Psihas. Nova detectors response and energy estimation. *DPF*, 2017.

Fernanda Psihas. *Measurement of Long Baseline Neutrino Oscillations and Improvements from Deep Learning*. PhD thesis, Indiana University, 2018.

Evan Racah, Seyoon Ko, Peter Sadowski, Wahid Bhimji, Craig Tull, Sang-Yun Oh, Pierre Baldi, et al. Revealing fundamental physics from the daya bay neutrino experiment using deep neural networks. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 892–897. IEEE, 2016.

Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

Stephan Rasp, Michael S Pritchard, and Pierre Gentine. Deep learning to represent subgrid processes in climate models. *Proceedings of the National Academy of Sciences*, 115(39): 9684–9689, 2018.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In

Proceedings of the 34th International Conference on Machine Learning-Volume 70, pages 2902–2911. JMLR. org, 2017.

Joshua Renner, A Farbin, J Muñoz Vidal, JM Benlloch-Rodríguez, A Botas, Paola Ferrario, Juan José Gómez-Cadenas, Vicente Álvarez, CDR Azevedo, FIG Borges, et al. Background rejection in next using deep neural networks. *Journal of Instrumentation*, 12(1):T01004, 2017.

Christian Ritter, Thomas Wollmann, Patrick Bernhard, Manuel Gunkel, Delia M Braun, Ji-Young Lee, Jan Meiners, Ronald Simon, Guido Sauter, Holger Erfle, et al. Hyperparameter optimization for image analysis: application to prostate tissue images and live cell data of virus-infected cells. *International journal of computer assisted radiology and surgery*, pages 1–11, 2019.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

Peter Sadowski and Pierre Baldi. Neural network regression with beta, dirichlet, and dirichlet-multinomial outputs. 2018.

Peter Sadowski, Balint Radics, Yasunori Yamazaki, Pierre Baldi, et al. Efficient antihydrogen detection in antimatter physics by deep learning. *Journal of Physics Communications*, 1 (2):25001, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Warren Scott, Peter Frazier, and Warren Powell. The correlated knowledge gradient for simulation optimization of continuous parameters using gaussian process regression. *SIAM Journal on Optimization*, 21(3):996–1026, 2011.

D Sculley, Jasper Snoek, Alex Wiltschko, and Ali Rahimi. Winner's curse? on pace, progress, and empirical rigor. 2018.

Ilsoo Seong, Lars Hertel, Julian Collado, Lingge Li, Nitish Nayak, Jianming Bian, and Pierre Baldi. Convolutional neural networks for energy and vertex reconstruction in dune.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

Chase Shimmin, Peter Sadowski, Pierre Baldi, Edison Weik, Daniel Whiteson, Edward Goul, and Andreas Søgaard. Decorrelated jet substructure tagging using adversarial neural networks. *Physical Review D*, 96(7):74034, 2017.

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

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180, 2015.

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

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

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

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

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

R Core Team et al. R: A language and environment for statistical computing. 2013.

Samuel K Wang and Anastasios A Tsiatis. Approximately optimal one-parameter boundaries for group sequential trials. *Biometrics*, pages 193–199, 1987.

Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient bayesian optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3627–3635. JMLR. org, 2017.

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

Jian Wu, Matthias Poloczek, Andrew Gordon Wilson, and Peter I Frazier. Bayesian optimization with gradients. In *Advances in Neural Information Processing Systems*, pages 5267–5278, 2017.

Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.