0.1 Introduction

As the hype of artificial intelligence and deep learning has spiked within academia and the tech industry, computer scientists have made great strides in implementing widely used software libraries for deep learning. Statisticians, however, have lagged behind, with only 1 widely used package on CRAN for deep learning – `h2o`, developed by **?**. While some statisticians are now using Python for scientific computing, many statisticians are still attached to R, prized for its familiarity, open-source nature, and dedicated use for statistical computing. The current standard for deep learning in R, `h2o`, is relatively easy to use, but heavily constrained to generic artificial neural networks, lacking functionality for state-of-the-art structures (convolutional, recurrent, etc.).

We look to develop a software package for R that provides an intuitive interface for statisticians, while allowing for deep flexibility with respect to network construction. If we can rely on a C-based engine that makes use of GPU computation, we will maintain ease-of-use through the scripting functionality of R, while providing computational feasibility. We also look to tie in Bayesian Optimization for hyperparameter tuning (see 0.1.1).

0.1.1 Bayesian Optimization

The goal of nonlinear optimization can be summarized as finding a global maximizer or minimizer for an objective function, where $f : \mathcal{X} \mapsto \mathbb{R}$ is some black box function taking input $\mathbf{x}$ from the domain $\mathcal{X}$ and returning a real number. Moreover, the Bayesian optimization framework allows f to take probabilistic characteristics, so there is no deterministic constraint.

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{X}}{\arg\min} \, f(\mathbf{x})$$

In most practical applications, the function is unknown or does not have a closed form solution to its gradient. Any optimization method must therefore consist of evaluating the function in some capacity. One of the special extremes of this problem is the case where each function evaluation is very expensive. Of course, this applies generally – in attempting to discover the global optimum, we would like to solve the problem with fewer, rather than more function evaluations. In the extremely expensive evaluation case, however, we must prioritize the efficiency of an optimization algorithm over other aspects (stability, etc.), as even a slightly smaller number of function evaluations could save hours or days in model development and release. These situations are where Bayesian optimization shines through, with its ability to discover a reasonable estimate of the optimizer within a constrained budget.

The general idea of Bayesian optimization is to use prior and posterior updating to iteratively optimize acquisition functions and augment our decision-making. Acquisition functions map potential candidate points for evaluation to their utility, usually by striking a balance between exploitation (choosing points we are certain to be near-optimal) and exploration (choosing points we know nothing about). The framework begins with a probabilistic surrogate model, which consists of a prior belief distribution on the behavior of our unknown objective function, and an observation model that embodies the data generation. Using a regret-based version of our acquisition functions, new points are chosen and then used to update the posterior distribution. For an extensive and formal explanation of the Bayesian optimization framework, as well as several applications and evaluations, please see the thorough work by **?**.

0.1.2 Technical Choices

Rcpp

We look to funnel the power and speed of C/C++ into an interface for R, noting that we must interface with external C/C++ libraries (rather than simply write C functions from scratch). We therefore cannot simply use the trivial `.C` calls from R, due to our need to interface with complicated library structures and header files. We use `Rcpp`, an R package developed by **?** for this exact purpose. It has been used to create thousands of R packages, and is well-documented with knowledgable authors.

Caffe

? released `Caffe` a few years ago, enabling researchers to make use of powerful and flexible deep learning framework. Under the umbrella of the Berkeley Vision and Learning Center, it is especially suited for image recognition and has strong support for convolutional neural networks. Due to the design choices, however, it is very easy to add other layers (as shown in Part 2, *Nonparametric LSTMs*), making it a strong candidate for our use. Its best selling point, however, is that it is self-contained and written in C++ , allowing us to link it using `Rcpp` (as opposed to embedding with Theano in Python, etc.).

Training in `Caffe` requires specifying two files: a network and a solver. These both come in the widely-used protocol buffer format, a data structure proposed and championed by Google. A network is composed of specified data, transformation, neuron, loss, and other layers that form a directed acylic graph. The solver file specifies other hyperparameters for training and testing, such as learning rate policies and training iterations. To create a new layer in C++ , one merely needs to specify a forward and backward computation, and the modularity of `Caffe` takes care of the rest. There is already, however, a multitude of layers implemented within Caffe, representing state-of-the-art advances in convolutional and recurrent neural networks. Therefore, using a Caffe-based system allows for researchers to experiment with the latest network types or even develop their own layers.

BayesOpt

Bayesian optimization has recently garnered lots of attention, but there are few software libraries with proper documentation. We choose `BayesOpt`, a C++ library developed by **?** with straightforward C and C++ APIs. To optimize a function in `BayesOpt`, one simply needs to provide a function signature fitting into the provided template, and specify parameters for a continuous or discrete domain.

0.2 Package Usage

As of the writing of this thesis, `deepbayes` is not yet a production-ready CRAN package. There are a variety of small improvements and touch-ups left, as we have laid the groundwork for finishing off the software. In this section, we will outline how the package is implemented, and how to use it. Please see the Github repository for source code.

0.2.1 Data

An easy example to follow along with is given in the file `max_iter_example.R`, which contains `max_iter_example()`, an R function that demonstrates how one would implement a deep network tuning procedure. First, we generate data and split it into folds with functions from `data_util.R`. For most cases, users will likely have their own dataset – they can follow the simulation methods and wrangle their data into HDF5 format (the easiest data format for interchange between R and `Caffe`). The `.h5` file can be located anywhere, but should be accompanied by a `.txt` file in the same directory and with the same filename (aside from the extension). The simulation and splitting methods in `data_util.R` write to `datasets/` within the package directory.

0.2.2 Creating Network and Solver Protocol Buffers

The next part requires the most user implementation. Depending on the hyperparameter tuning, net and solver `.prototxt` files should be generated programmatically. In our example, this is done in `create_protos_max_iter()`. It first pulls in a network template within the `nets/` folder in the package directory, and creates a new proto for each individual network. In this example, the hyperparameter to be tuned is specified in the solver file, so we only need to create k network protos. We then need to create kn solver files: for each of the n values of `max_iter` in consideration, we need to create k protos that point to each of the k folds of the dataset. Note that the paths in the network protocol buffers must point correctly to the data at hand. These protocol buffers themselves must be located within the installed package directory, as those created by `create_protos_max_iter()` do.

0.2.3 Optimizing

Finally, we simply call the `Rcpp`-exported `optimize()` function, providing the hyperparameter matrix, number of cross-validation folds, an evaluation budget, a prefix to retrieve the solvers, and some system paths. This function has been generalized to optimize cross-validation error over all hyperparameters provided by feeding `cv_error_function()` to the BayesOpt API.

0.2.4 Other Examples

In `max_iter_example()`, we tune a hyperparameter that is specified and contained within the solver file. Users may also wish to tune other meta-parameters such as the structure of the network itself. We outline such an example in `num_node_example()`, where all solver-based hyperparameters are held constant; we take a network with one hidden layer and tune the number of nodes within that layer. Note that because the number of training iterations is held constant, the optimum is somewhat artificially induced, as larger networks should require more training iterations before convergence.

We also provide a multi-dimensional tuning example, in `iter_layer_example()`, to demonstrate the capabilities of Bayesian optimization with respect to multiple variables, as well as to deal with the relation between model complexity and training iterations. This example simultaneously tunes the number of fully connected layers and the number of training iterations. Note that the protocol buffer file creation becomes complex when dealing with network structure rather than pure hyperparameters, so abstraction is ideal for future development (see Section 0.3.2).

0.3 Discussion and Future Work

In this section, we discuss improvements necessary for `deepbayes` to become published on CRAN, so that a researcher in the future has a documented start to finishing and refining the package.

0.3.1 External vs. Internal Libraries

Currently, the R package as developed functions assuming that the external C++ libraries `Caffe` and `BayesOpt` are installed on the user's machine. The general practice in R package development is to only make this assumption for very popular or widely-used external libraries, such as `boost`. While `Caffe` is gaining in popularity, it is not reasonable to assume that a given user has it installed – moreso for the lesser known `BayesOpt` library. It is therefore preferable – and likely necessary for CRAN-publication – that `Caffe` and `BayesOpt` are included internally with the source code of `deepbayes`. The `Makevars` file is configured with respect to the Princeton SMILE machine and its installations; requiring users to modify this source to enable installation of `deepbayes` is unwieldy and detrimental to ease of use.

0.3.2 Abstraction of Protocol Buffer Creation

While the optimization has been elegantly abstracted and generalized for the user, the creation of protocol buffer files for both the networks and solvers remains somewhat ungainly. Ideally, the user would like to avoid having to directly modifying files within R and changing network structure and hyperparameter values. If a `deepbayes` API could provide a generalized process for taking in user input for hyperparameter variation, and properly hiding the file creation, then the implementation required from the user drops drastically. In other words, the statistician could then spend more time experimenting and tuning their networks to their data, rather than implementing file manipulation methods.

0.3.3 Bayesian Optimization Parameters

Because it is highly desirable to hide the core C++ away from the R user, we also wish to have an R-side process for specifying parameters for the `BayesOpt` API call. Currently, the only mutable parameter is the budget, even though `BayesOpt` provides a wealth of options for priors, kernels, and much more. This is simply a matter of passing a dictionary-like object from R and parsing it into the `bayesopt::Parameters` object.

0.4 Conclusion

We have laid the groundwork for `deepbayes`, an R package that interfaces with the modular deep learning library `Caffe`, as well as `BayesOpt`, a Bayesian optimization library. `deepbayes` allows R users flexibility in developing and experimenting with their own network, while providing a framework for hyperparameter tuning. All source code for `deepbayes` is available on Github (`https://github.com/edz504/thesis/tree/master/project3`) as mentioned in the Preface, allowing for further development. Several additions and refinements are necessary to make `deepbayes` publishable on CRAN, which would allow for widespread adoption by the statistics community and allow for more statistical approaches to deep learning problems in a field dominated by computer scientists.