Visualizing Bayesian Optimization for Teaching

William M. Temple*
University of Colorado Boulder

ABSTRACT

Effectively teaching machine learning, with its high cognitive load, demands exceptional teaching utilities. Often, students find abstract discussions or overly-formal mathematical discriptions either difficult to interpret and understand or insufficiently applicable. In this article, I explore the application of data visualization techniques to Bayesian Optimization (Gaussian Processes) and describe the tasks that students engaged in hypothetical study of this model would likely perform. I further develop some prototype user-interfaces and address technological challenges essential to the development of an interactive system for studying Bayesian Optimization.

1 INTRODUCTION

Educational technology systems present interesting design challenges when analyzed using a data-visualization lens. Students have different goals and desires and separate motivations from data analysts. For better or worse, much of the discourse about data visualization in our coursework and in the literature we have read for class has focused on the consumption of data by analysts—or, at least, by users (even untrained or novice users) *performing* an analytical role.

However, as my own research focuses on the development of educational tools for novice Computer Science and Programming students, I always try to consider the ways in which the data visualization techniques that we have discussed apply in the classroom to enhance the educational experience and provide for a more robust pedagogy.

Towards this goal, in this project I have focused on building a tool to analyze the state of a machine learning algorithm. In particular, the system shows the execution of a Bayesian Optimization using a Gaussian Process Regression. The system allows a curious user to examine the state of the model and shows not merely the *result* of the application of the model (which would be the primary interest of users in analytical roles), but also makes evident the choices that the model makes during execution and empowers the user to understand the mathematical reasoning behind those choices.

1.1 Why Gaussian Processes?

The Guassian Process is an advanced stochastic process which uses a suite of functions and strategies for optimizing those functions to create estimations of the mean and variance over a distribution of other functions. That is to say, it is relatively complicated.

However, a computer can execute a Gaussian Process in a small series of abstract steps:

1. Initialize the Algorithm by sampling a small number of random points and fitting an initial regression using a *kernel* function.

2. Use an *acquisition* function to choose a point which is likely to improve the estimated mean of the process.

3. Sample that point and add it to the regression.

4. Repeat until the model converges on an accurate estimation.

That is to say, the Gaussian Process isn't so complicated that it is immune to effective decomposition through visualization. It also has certain essential properties that I believe make it particularly suited to a visual approach to teaching its concepts. I discuss these properties further in Section 3.

2 RELATED WORK

In this section, I discuss some existing literature which informs my design choices and my thinking about how visualization and task modeling in particular relate to the goal of education. I also include some discussion of existing education literature which I feel aids my explanation of why visualization provides a prefererable environment for learning.

2.1 Teaching as Presentation

- Presentation tasks
  - Where does teaching fit in to presentation?

2.2 Task Modeling for Learning

- Students and teachers have diffeerent motivations
  - What tasks do students approach an educational vis with?
  - What specific tasks are we going to accomodate?

2.3 Knowledge Construction

- Constructivism
  - Exploratory design
  - User paced

3 DESIGN

3.1 User Experience

- Coordinated views
  - User can examine the result of each iteration
  - User sees the coordinated points, showing the alignment of true function, estimation, etc.

3.2 Technical Challenges

Early in the prototyping stages of this project, I realized that computational complexity would pose significant challenges to the interactivity of this software and its usability. The first Python prototypes of the software could only compute and render one iteration of the Optimization algorithm about every 25 seconds. Through some code optimization effort, I managed to reduce the runtime per iteration to 7 seconds, but it became clear during this process that further optimization would require a shift in development paradigm.

At this latency, the interactivity of the system is threatened, as users may press buttons that have no effect, and as they begin to use the software, it takes considerably less than seven seconds to digest the information divulged by a single iteration of the program. Rudimentary profiling of the code reveals that the program spends the vast majority (90%) of its time drawing the four coordinated plots, as each plot iterates over each pixel in the its image. Even drawing only one of the plots requires about two seconds using Python,

*e-mail: William.Temple@colorado.edu

where a single plot consists of ten thousand points (one hundred points in both the X and Y directions).

As a proof of concept, in order to solve this problem, I wrote GPU fragment shader programs using the GPU.js library [1]. This library dyanamically recompiles a restricted subset of JavaScript into GLSL shader code, which then runs on the SIMD processors in the host machine's Graphics Processing Unit, resulting in much faster execution. Furthermore, this shader code deposits resulting textures directly into an HTML `<canvas>` element, so that the browser can easily render it. I integrated these fragment shaders into my concept UI application, and measured that the same machine could render a function of the same complexity as those in my Python program at several times the resolution (2048 points in both the X and Y direction) in about 110 milliseconds [1].

4 Discussion

4.1 Prototype Artifacts

5 Conclusion

References

[1] GPU.js Contributors. GPU.js. `https://github.com/gpujs/gpu.js`, 2018.

[1] Test machine: 6-core Intel i7, 16GB RAM, NVIDIA GeForce GTX 960