AM205 Final Project: Learned Uncertainty - Aware (LUNA) Modifications with Numerical Methods

Victor Avram, Michael Butler, M. Elaine Cunha, Jack Scudder

Due: 2:00 PM, December 14, 2020

1 INTRODUCTION AND MOTIVATION

Recent years have seen an explosion in machine learning applications for healthcare, police work, autonomous vehicles, and the like – expanding their application beyond strictly research and scientific bases. Myriad examples of boons and banes of machine learning pour forth from a simple Google search. These machine learning applications range from improved natural language processing allowing to facial recognition systems incorrectly identifying suspects in criminal investigations [1]. In the latter example, an African-American male was falsely accused of robbery and apprehended based on the results of a facial recognition algorithm. Although released the next day, the damage cannot be undone. This case is not unique, and has lead researchers and policymakers alike to bolster the development of machine learning models that remove bias from their classifiers.

The primary issue at hand is dealing with uncertainty in models. As a short synopsis, there are two primary forms of uncertainty in statistical models: epistemic and aleatoric. Epistemic uncertainty results from insufficient data and can be combated by obtaining more observations. Aleatoric results from confounders in the data, variations in the data gathering methods, and other characteristics that cannot be easily assessed or explained by the model. A more detailed explanation with examples can be found at the link to Kendall's blog [3] in our References.

An increasingly popular solution to this problem of accounting for uncertainty involves taking a Bayesian approach to modeling neural networks. Rather than producing a single point estimate, Bayesian models result in a distribution of predictions for any given input. Users may then infer model confidence based on the variation in the estimates produced: low variability suggests greater confidence in predictions, while high variability indicates greater uncertainty. Ideally, predictive uncertainty reflects an input's proximity to the training data (i.e., data-scarce regions show greater uncertainty).

Producing Bayesian estimates from a neural network with the desired expression of uncertainty requires several layers of computation that can be both complex and resource expensive. In this paper, we review one model, Learned Uncertainty-Aware bases for linear regression (LUNA), and evaluate potential improvements to its computational implementation using alternative numerical methods. Particularly, we evaluate different optimization techniques, contrasting them with the default optimizer for LUNA, Adam (the Adaptive Moment Estimation) [4]. We also delve into the finite difference schema used in LUNA to add diversity to the model which promotes necessary uncertainty the further the predictions are from the given training data [7].

We have found that discerning between different optimization methods for neural network training is a complicated task. Given the dimensionality (i.e. number of parameters) of the objective function, finding optimal solutions is a non-trivial task. Simpler, gradient-based approaches provide substantial computational benefits over second derivative-based approaches. Furthermore, second derivative-based approaches do not necessarily produce better fits nor better measures of uncertainty in data scarce regions. Modifications to the finite difference schema did not significantly reduce computational time or improve log-likelihood calculations. Ultimately, we could not assess a clear trade-off across performance metrics.

2 BACKGROUND

2.1 Bayesian Models for Neural Networks

Traditionally, the weights of a neural network are trained for the best fit point estimates given a set of training data. At test time, the neural net produces a single prediction for a given input. Bayesian Neural Networks (BNNs)[5] instead apply a prior distribution over the weights and train for the best fit distribution given a set of training data, resulting in a distribution of predictions for a single input at test time. As described above, the variance in these predictions can then be used to evaluate a model's certainty in its predictions.

However, this approach comes with significant computational complexity and cost. Neural Linear Models (NLMs) have emerged as an alternative for producing Bayesian estimates from neural networks without the inherent complexity of BNNs.[6][8] NLMs fit a model in two stages: 1) train a standard neural network over all data, 2) replace the last layer with coefficients fit with a Bayesian linear regression; all weights outside of the last layer are held constant at point estimates.

In more technical terms, suppose we have training data that consist of features $X \in \mathbb{R}^{N \times D}$ and target $Y \in R^N$, where N is the number of observations, and D is the number of features. In the first stage of NLM training, we train a neural network with K hidden layers, each with width L_k using our training data. To train, we minimize the following objective function:

$$C = \frac{1}{N}||(Y - f_\Theta(X))||_2^2 + \gamma||\Theta||_2^2$$

In other words, we seek to find a neural network, f, with parameters Θ that minimizes the mean squared error between the predictions and the actual target data, Y, while penalizing large coefficients using the positive regularization term γ. Once trained, the f maps from: $\mathbb{R}^D \to \mathbb{R}$.

Next we chop off the final set of weights, producing a feature map that now predicts the final hidden layer: $\phi_\theta : \mathbb{R}^D \to \mathbb{R}^{L_K}$, where $\theta \subset \Theta$ contains the network weights from the $K-1$ layers, and L_K is the dimension of the last hidden layer. As a last step, we apply a prior to the last layer of weights, and run a Bayesian linear regression to predict our target data Y. In other words, the Bayesian linear regression takes the form of:

$$y \sim \mathcal{N}(\mathbf{\Phi}_\theta \mathbf{w}, \sigma^2 \mathbf{I}), \qquad w \sim \mathcal{N}(\mathbf{0}, \alpha \mathbf{I}) \tag{1}$$

where $\mathbf{\Phi}_\theta$ is a $N \times (L_K + 1)$ matrix that applies the feature map to X and augments the resulting $N \times L_K$ matrix with a row of ones for the bias; $\mathbf{w}$ represents the final layer of weights; and the covariance of y and the prior over w are identity matrices multiplied by a constant (σ^2 or α, respectively). This formula illustrates how the fitted weights of the neural network serve as the basis for the Bayesian linear regression.

NLMs notably improve upon BNNs in runtime but are susceptible to misleading expressions of uncertainty, especially when trained with regularization. To build some intuition on why that's the case, consider, in general, why practitioners add regularization to their models. As a simple example, the shape of a high degree polynomial fit to training data in Figure 1 would change drastically in the data poor region, depending on which samples we use to fit. Thus, if the practitioner built confidence intervals for this high degree polynomial model via bootstrap sampling, we would expect the confidence intervals to be extremely large in the missing data region. If we instead regularized the coefficients on the polynomial, we could push the coefficients to be smaller in magnitude, generally leading to tighter confidence intervals in the data gap (Tibshirani, 2013) [2]. In general, the practitioner tunes the regularization hyperparameter in order to maximize predictive performance on test data, typically by minimizing validation mean squared error through cross validation or via maximizing some sort Information criterion (like Bayes Information Criterion). However, as Thakur et al. (2020) notes, tuning a model to minimize test mean squared error doesn't necessarily produce reasonable uncertainty estimates for out of sample data. In other words, the model with the supposed best accuracy may return estimates that are too confident in data scarce regions [7].

As Thakur et al. (2020) point out, the magnitude of regularization doesn't affect the test log likelihood of an NLM either; thus, if a practitioner couldn't easily visualize the predictive distribution of an NLM, they may utilize a trained model that returns poor out of sample uncertainty estimates [7]. Like a regularized polynomial model, NLMs with more regularization actively select weights that form a basis with low functional diversity, which restricts the final distribution of model predictions. In other words, when we train the neural network in the first stage, the regularization term $||\Phi||^2$, incentivizes the optimizer to seek weights with smaller magnitudes. Like with a regularized polynomial model, the trained neural network will transform test input data to look quite similar in the final layer. Thus, after the final Bayesian linear transformation, the NLM's predictions will lack variation in both data both rich and poor regions alike.

To show that these poor uncertainty estimates are due to the feature map Φ, and not the final Bayesian linear regression, we plot the prior predictives created by an NLM in Figure 1, using the hyper parameters described in subsection 3.1. At a high level, a prior predictive predicts an output for a given test observation using the learned feature map and the prior (i.e. not updated) distribution of the final layer coefficients. More explicitly, we create the figure in the following way: consider test input data X which consists of many evenly spaced points from -5 to 5. Map X into $R^{N \times L_K}$ with Φ; sample a set of final layer coefficients from $w_{prior} \sim \mathcal{N}(\mathbf{0}, \alpha \mathbf{I})$; and map the final layer to output space $\hat{Y} = w_{prior}\Phi(X)$, creating a squiggly blue line.

Figure 1 shows that sampling several sets of final layer coefficients from the prior distribution returns a relatively diverse set of predictions in the data-scarce region when we have no regularization. Consequently, the posterior predictions of the unregularized model contain more uncertainty. But even with zero regularization, the uncertainty in this NLM isn't ideal– the 95% predictive interval should span above and below the true function. As Thakur et al. (2020) note, if we run an unregularized NLM's repeatedly with the same hyperparameters, the out of sample uncertainty interval will significantly vary merely due to the randomness involved in the optimization. For more evidence, we fit additional NLM models on a variety regularization values in Figure 13 and found inconsistent uncertainty estimates. Overall, NLMs cannot reliably produce reasonable uncertainty, regardless of the amount of regularization [7].

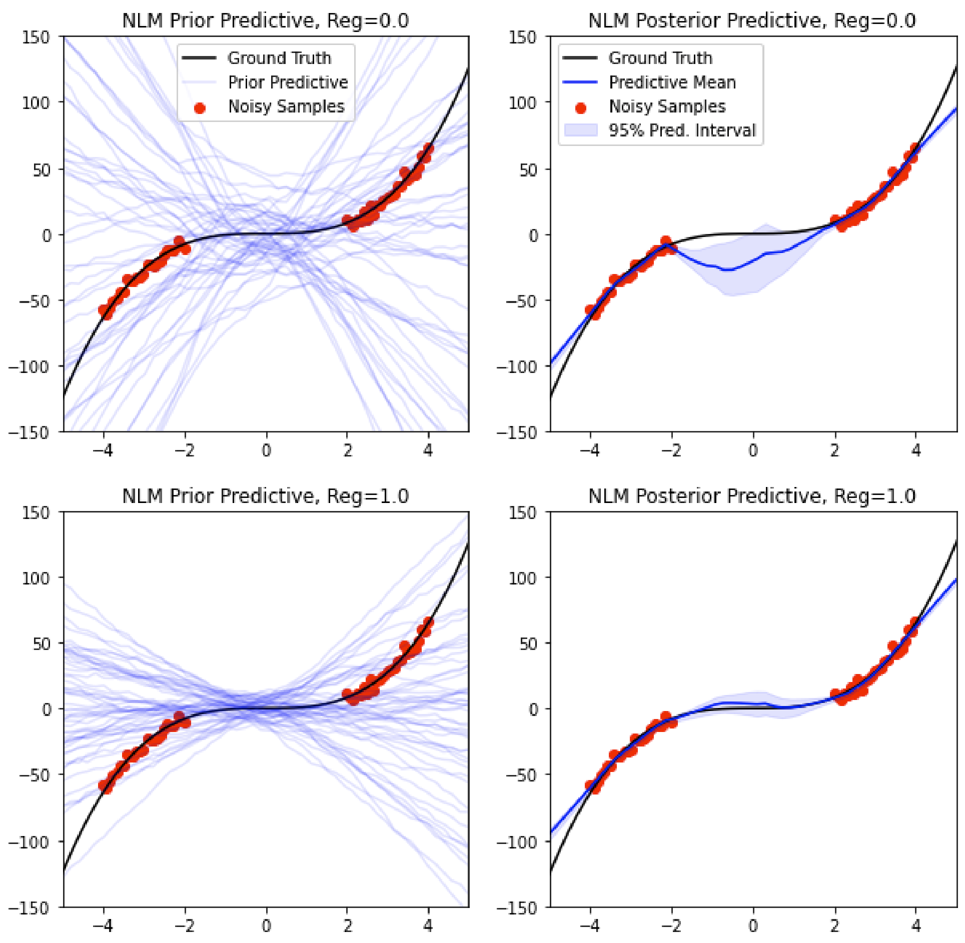

Figure 1: NLM predictive output with and without regularization. The top row (without regularization) reveals a diverse span of functions in the prior predictive, translating to a wider predictive interval in the posterior predictive. The bottom row (with regularization) is more constrained in both cases. Regardless of regularization, the test log likelihood hovers around -120. For information on the data and hyper parameters used, see subsection 3.1.

2.2 Learned Uncertainty-Aware (LUNA) Neural Linear Models

Thakur et al. (2020) introduces the Learned Uncertainty-Aware (LUNA) neural linear model which specifically trains for the neural network of an NLM to produce a basis with high functional diversity. This basis enables the Bayesian linear regression to consistently produce a wider distribution of model predictions, particularly in data-scarce regions. Figure 2 illustrates the difference in predictive uncertainty between the NLM and LUNA models. Furthermore, Figure 12 in the appendix shows that LUNA will produce reasonable uncertainty estimates across a variety of regularization values.

The LUNA model contains two key modifications to the neural network of an NLM. First, the network is trained to produce M auxiliary outputs that collectively contribute to the loss function. Figure 3 shows a graphical illustration of the LUNA architecture. Conceptually, for a given observation, each of the M auxiliary outputs predicts the output Y while using different final layer parameters.

Second, the objective function is augmented with a term explicitly encouraging diversity amongst the M functions that map input to each auxiliary output. Equation 2 breaks down the complete

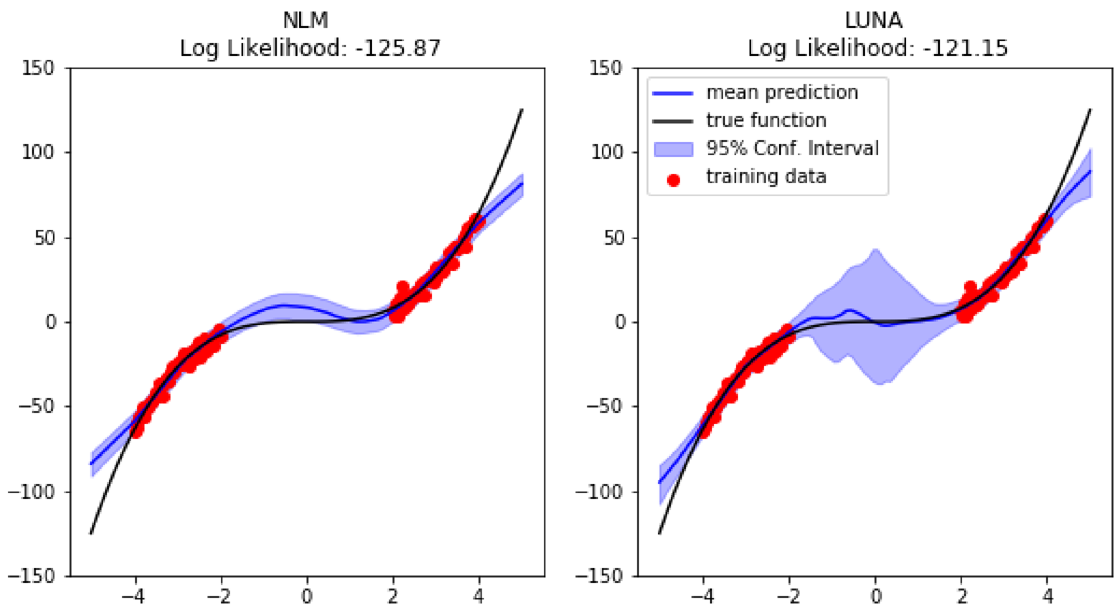

Figure 2: Visualization of predictive uncertainties for NLM and LUNA using our base hyper-parameters (subsection in 3.1). Note that NLM took 4 minutes to train while LUNA took 90 minutes.

loss function, $\mathcal{L}_{LUNA}$ into $\mathcal{L}_{fit}$ and $\mathcal{L}_{diverse}$ components:

$$\mathcal{L}_{LUNA}(\Psi) = \mathcal{L}_{fit}(\Psi) - \lambda \cdot \mathcal{L}_{diverse}(\Psi) \tag{2}$$

where Ψ consists of all weights used in training the neural network (including the final layer of weights that are retrained in the Bayesian regression phase); and where λ controls how much weight is given to the diversity metric. Notice that we seek the maximize this objective function over Ψ.

$\mathcal{L}_{fit}$ evaluates the model's fit to the training data using the average log likelihood across all of the auxiliary outputs with ℓ_2 regularization:

$$\mathcal{L}_{fit}(\Psi) = \frac{1}{M}\sum_{m=1}^{M} logN(y; f_m(X), \sigma^2 I) - \gamma\|\Psi\|_2^2 \tag{3}$$

Equation 4 defines $\mathcal{L}_{diverse}$, which measures diversity in the auxiliary functions outputs. It sums the squared cosine similarity between the gradient of every pairwise combination of auxiliary functions, thus rewarding the model when auxiliary functions have different shapes. In practice, in the gradients in diversity score are approximated with finite differences.

$$\mathcal{L}_{diverse}(\Psi) = \sum_{i=1}^{M}\sum_{j=i+1}^{M} \text{CosSim}^2\left(\nabla_x f_i(x_{train}), \nabla_x f_j(x_{train})\right) \tag{4}$$

By balancing model fit with diversity in the training phase, the resulting weights of the network necessarily take on diverse values that fit the data and form an expressive basis for regression.

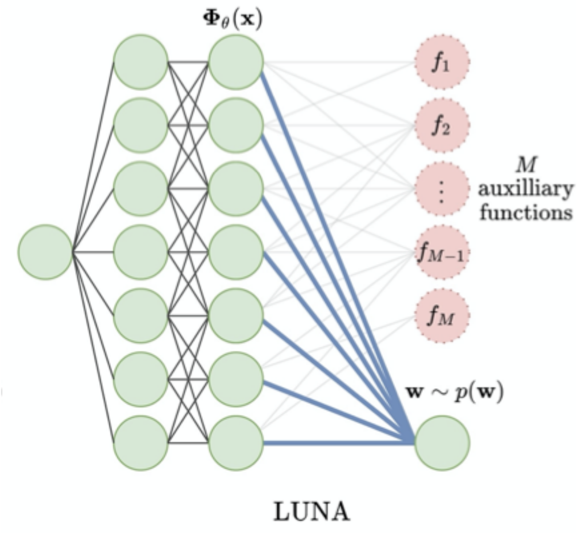

Figure 3: Example illustration of LUNA architecture. Pink circles represent M auxiliary outputs which are discarded after training. The single green node of the last layer constitutes the output of the Bayesian linear regression step. Source: Thakur et al. (2020), Figure 2.

In the Bayesian linear regression step, the auxiliary outputs are discarded, but the expressiveness of the weights trained for diversity remain.

While LUNA produces effective predictive uncertainty and avoids the complexity of a full BNN, it is computationally slower than an NLM because the diversity score is expensive to compute. In the next section, we explore potential alterations to the numerical implementation of LUNA in the hopes of improving runtime while maintaining the quality of the results.

3 METHODS AND RESULTS

Our analysis evaluates alternative numerical implementations of two areas of the LUNA model:

1. The **optimization technique** used to fit the neural network
2. The **finite difference** approximation for gradients in the cosine similarity calculation of Eq. 4

3.1 Base LUNA Implementation and Evaluation

We compare these alternative implementations to a base case example of how Thakur et al. (2020) fit and tuned LUNA. See Figure 2 for a visualization of the base implementation of LUNA and NLM. First, they generate 100 training points from the following function:

$$y = x^3 + \epsilon, \epsilon \sim N(0, 3)$$

we only sample points in $x \in [-4, -2] \bigcup [2, 4]$, creating a missing data region.

In the first stage of NLM and LUNA training, they fit a neural network with 2 hidden layers, a width of 50 for each layer, and with ReLu activation functions. Note that the neural network used in NLM has 1 output node while the one used in LUNA has 50 output nodes for the auxiliary functions.

To maximize the objective function in Equation 2, Thakur et al. use the Adam Optimizer and run for a maximum of 10000 iterations, with a step size of 0.01 (Kingma, 2015). To speed up training, our analysis used a maximum of 3500 iterations, though we do not believe this significantly affected performance.

For the Bayesian linear regression component of the NLM, where the model in Equation 1 is fit, Thakur et al. (2020) assumed the prior variance α of w is 1 and the variance σ^2 of y is 9.

To find a 'gold standard' implementation of regular NLM, Thakur et al. train several NLMs over a variety of regularization values, γ, and conclude that γ = 8.37 maximizes test log likelihood. In our base NLM implementation, we also use this value of gamma, though we show in Figure 1 and 13 that NLM performs inconsistently with respect to uncertainty quantification across a variety of regularization values.

To find a 'gold standard' implementation of LUNA, Thakur et al. train several LUNA models over a grid of regularization values γ and diversity score weights λ. They conclude that γ = 0.1 and λ = 1 maximize test log likelihood. We adopt these values in our analysis.

To assess the predictive performance of each model, we calculate the test log likelihood over the whole model. We calculate test log likelihood by first generating test points from the training data generating process (described above), and compute the joint probability of observations of the test data, given the posterior distribution of the weights. More explicitly, given a test set $\{(\mathbf{x}_n^*, \mathbf{y}_n^*)\}$, the log posterior predictive likelihood equals:

$$\log \prod_{n=1}^{N} p(\mathbf{y}_n^* | \mathbf{x}_n^*, \text{Training Data}) = \sum_{n=1}^{N} \log p(\mathbf{y}_n^* | \mathbf{x}_n^*, \text{Training Data})$$

where in a Bayesian linear regression model with posterior $\mathcal{N}(\mu, \Sigma)$:

$$p(y_n^* | x_n^*, \text{Data}) = \mathcal{N}(\mu^\top \mathbf{x}_n^*, \sigma^2 + (\mathbf{x}_n^*)^\top \Sigma \mathbf{x}_n^*)$$

3.2 Alternative Optimization Methods: Techniques

The implemented optimization methods include the steepest descent method, Newton's method, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, and the conjugate gradient method. A brief description of these methods are given below.

- Steepest Descent: A method that utilizes information from the first derivative of the given function in order to make updates.
- Newton's method: A method that uses information from the second derivative of the given function in order to make updates. The exact Hessian is derived for these updates.
- BFGS: A quasi-Newton method that uses an approximation of the Hessian in order to make updates.
- Conjugate Gradient: A quasi-Newton method that uses an approximation of the Hessian in order to make updates.

The stated methods were implemented with the aid of the Python package **autograd**, allowing automatic differentiation of arbitrarily complex functions. Specifically the **grad** and **hessian** methods were used in order to compute the gradient and hessian matrix (only for Newton's method) of the objective function used in training. We note that typically, Newton's method and the quasi-Newton methods allow for faster convergence than steepest descent, given that these methods incorporate information from the second derivative.

For each method, we developed custom implementations from scratch and evaluated our code on simple test functions with known minimums. At test time for the NLM and LUNA models, we found our implementations required significant computational resources to evaluate the more complex models. In order to obtain interpretable results, we relied on the **SciPy** package implementation of the BFGS and conjugate gradient methods for improved efficiency when modeling with LUNA.

3.3 Alternative Optimization Methods: Results

3.3.1 Custom Implementation of Optimization Methods on Test Functions

The implementation for each optimizer was tested on two test functions $f_1(x) = x^2 + 3x + 1$ and $f_2(x, y) = x^2 + y^2$, each with a single known minimum. f_1 has a global minimum at $x = -1.5$ and f_2 has a global minimum at $x = 0, y = 0$. Each algorithm was able to find the correct optimal solution.

Table 1: Performance of each optimizer on simple test functions

Optimizer	Function	Starting Point	Number of Iterations
Steepest Descent	f_1	$x = 10$	2
Newton's Method	f_1	$x = 10$	2
BFGS	f_1	$x = 10$	3
Conjugate Gradient	f_1	$x = 10$	2
Steepest Descent	f_2	$x = 12, y = 5$	2
Newton's Method	f_2	$x = 12, y = 5$	2
BFGS	f_2	$x = 12, y = 5$	3
Conjugate Gradient	f_2	$x = 12, y = 5$	2

3.3.2 Alternative Optimization Methods for NLMs

As previously stated, the first step in creating an NLM involves training a standard neural network to the given data. The optimal parameters (i.e. weights and bias terms) in a neural network are computed through an iterative optimization process of a specified objective function. Equation 3 above specifies the objective function for an NLM, which includes terms for mean squared error (MSE) and regularization.

Each optimizer was tested on a sequential neural net with 2 hidden layers, each 20 nodes wide. The training set consisted of 50 observations with one predictor and one response variable. Each optimizer, besides Newton's method, was able to find a solution (i.e. the optimal weights for the neural network). The regularization parameter was set to 8.37. Given that the objective function is a complex non-convex function, numerical optimization techniques are not guaranteed to find the optimal solution or even converge. Newton's method is not typically used as an optimization technique for neural network parameter tuning because it involves an analytically derived expression for the Hessian of the given objective function. Even when deriving the Hessian is feasible, the computational burden increases dramatically as the number of parameters n increases, given that the Hessian matrix scales by n^2. Newton's method was not able to find an optimal solutions when training the neural network with the specified parameters. The algorithm either could not converge or the Hessian matrix became non-invertable. Interestingly, Newton's method was able to find solutions for much simpler neural networks with fewer parameters. For these reasons, Newton's method was left out of subsequent analyses.

The performance of each optimizer was initially assessed by creating a NLM and fitting the given NLM to a training set of 50 datapoints. Performance was assessed by both the fit and computational requirements of the optimization method. The MSE as stated in equation **??** was used to assess the model fit and the number of iterations and computation time were used to assess the computational requirements. A table with each optimizer's performance is given in Table 2.

Table 2: Performance of each optimizer when training a NLM. The neural network consisted of 2 hidden layers, each with 20 nodes. The input dimension and output dimension were 1.

Optimizer	MSE	Number of Iterations	Elapsed Time (s)
Steepest Descent	500.7727	4	6.0527
BFGS	291.6327	3	172.9966
Conjugate Gradient	271.4967	3	145.2712

It can be seen that the quasi-Newton methods perform better than steepest descent given that the MSE scores for both quasi-Newton methods are considerably smaller than that for steepest descent. As well, fewer iterations are required to arrive at the optimal solution. However, the improved performance comes at a computational cost. The computational time required to find the optimal solution for both quasi-Newton methods is over an order of magnitude larger than that for steepest descent. The extra computational expenditure is likely due to the utilization of second derivative information for the quasi-Newton methods. The size of the Hessian approximation scales by n^2 given n parameters.

The mean of the predictive intervals for each method is given in Figure 4. All three models produce a mean prediction that fits the training data well, although the paths of the mean predictions in the data-scarce regions differ.

Of course, it may be necessary to change the architecture of the neural network, either by changing the number of nodes (width) of the hidden layers or by changing the number of hidden layers. The performance for each optimizer is given across different values for the width of the hidden layers in Figure 5.

BFGS consistently outperforms the other optimization methods across different values for the width of the neural network. However, as the width of the NN increase, the performance for BFGS and the performance for conjugate gradient seem to converge. However, the computation time for the steepest descent method does not scale as drastically with the width of the NN. As stated previously, the steepest descent method uses first derivative information and therefore is less computationally intensive.

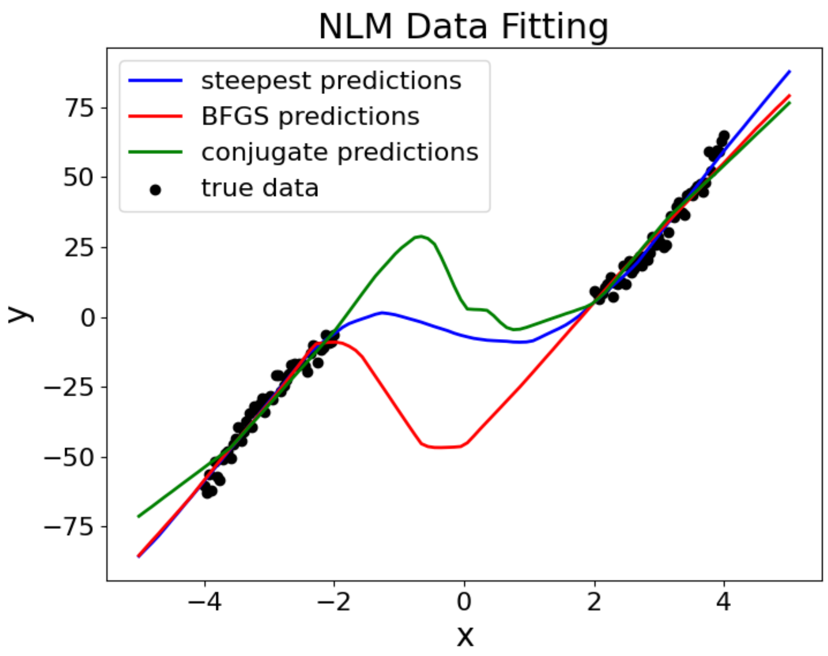

Figure 4: NLM predictive output for the steepest descent, BFGS, and conjugate gradient optimization methods. The mean prediction for each data point across 1000 predictions is plotted for each optimizer.

3.3.3 Alternative Optimization Methods for LUNA

The optimization methods used for NLMs were extended to the LUNA implementation. The objective function for LUNA is specified in Equation 2, with components specified in Equations 3 and 4. The complexity of the objective function increases, given the addition of the $\mathcal{L}_{diverse}(\Psi)$ term. Performance for each optimizer was first assessed by running 100 iterations of the LUNA algorithm. The same data used for training the NLM was also used for training the LUNA model. The underlying neural network consisted of 2 hidden layers, each with 50 nodes. The regularization parameter was set to 0.1. The results are given in Figures 6, 7, and 8. 100 iterations was used as a proxy to measure the rate of progress for each optimizer. Table 2 contains performance metrics for each optimizer. Specifically, it includes the log likelihood estimate and computation time for each optimizer.

Table 3: Performance of each optimizer when training with LUNA for 100 iterations. The neural network consisted of 2 hidden layers, each with 50 nodes. The input dimension was 1 and the output dimension was 50 in order to accommodate the auxiliary functions.

Optimizer	Log Likelihood	Elapsed Time (s)
Steepest Descent	-134.4904	131.082
BFGS	-129.4452	1067.979
Conjugate Gradient	-126.7051	1274.759

The LUNA algorithm was additionally run for 1000 iterations in order to produce more robust measures of uncertainty in the data scarce region (i.e. the gap between the two groups of data points). Each optimization method was assessed. The resulting fits are given in Figures 6, 7, and 8. Interestingly, after 1000 iterations, sufficient progress in terms of fit and predictive uncertainty are not made for all optimizers. Steepest Descent makes progress in terms of

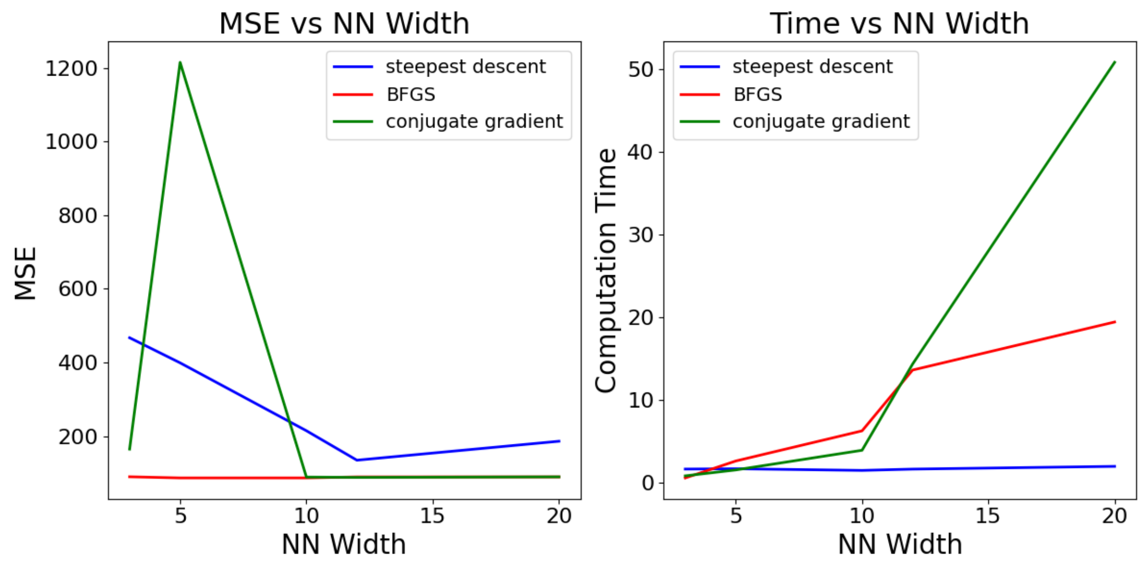

Figure 5: MSE and computation time vs the width of the hidden layers for the steepest descent, BFGS, and conjugate gradient optimization methods. The number of hidden layers was kept constant at 2.

fitting the true underlying function and indeed the predictive uncertainty in the data scarce region increases while maintaining low uncertainty in the data full regions. However, BFGS and the conjugate gradient method do not make robust progress in terms of fitting the true underlying function or increasing the predictive uncertainty in the data scarce region. This calls into question the utility of certain optimization methods when performing optimizations on complex non-convex functions. Steepest descent, which can be termed as a simpler method, had better performance and required less computation time. The log likelihood estimates and computation times for each optimizer are given in Table 3.

Table 4: Performance of each optimizer when training with LUNA for 1000 iterations. The neural network consisted of 2 hidden layers, each with 50 nodes. The input dimension was 1 and the output dimension was 50 in order to accommodate the auxiliary functions.

Optimizer	Log Likelihood	Elapsed Time (s)
Steepest Descent	-122.5896	682.474
BFGS	-121.8735	3022.393
Conjugate Gradient	-136.2256	1576.207

The log likelihood estimate increased with more iterations for all optimization methods except for the conjugate gradient method. These findings are indicative of the pitfalls of certain optimization methods and the difficulty of finding optimal solutions for complicated problems. Typically, more simple gradient descent methods such as stochastic gradient descent and gradient descent with momentum are used as optimization methods for neural networks. The extra computational cost of heavier methods such as quasi-Newton methods does not seem to receive merit. Steepest descent and BFGS both produce roughly the same log likelihood after 1000 iterations, while the computation time for steepest descent is roughly 1 order of magnitude lower than that for BFGS.

Our optimization methods were compared to the Adam Optimizer, serving as a benchmark in order to assess the utility of alternative methods in LUNA training. The LUNA algorithm was run for 3500 iterations using the Adam Optimizer with the same regularization parameter of

0.1. The resulting log likelihood estimate was approximately 120 and the computation time was approximately 90 minutes. Assuming linear scaling between computation time and number of iterations, the steepest descent, BFGS, and conjugate gradient methods would have arrived at a solution after approximately 40 minutes, 176 minutes, and 92 minutes, respectively. While there is a clear computational advantage to utilizing simple methods such as steepest descent, Adam likely outperforms all of the stated alternative methods. The stochastic nature of Adam allows for a more robust traversal of the parameter space, since Adam is less likely to get stuck in sub-optimal local minima. Additionally, is a first-order gradient algorithm and therefore relatively efficient, making it one of the gold standards for neural network training. The alternative methods all have exhibited difficulty in fitting the true underlying function, as well as robustly predicting uncertainty. Adam remains as the optimal method for optimization.

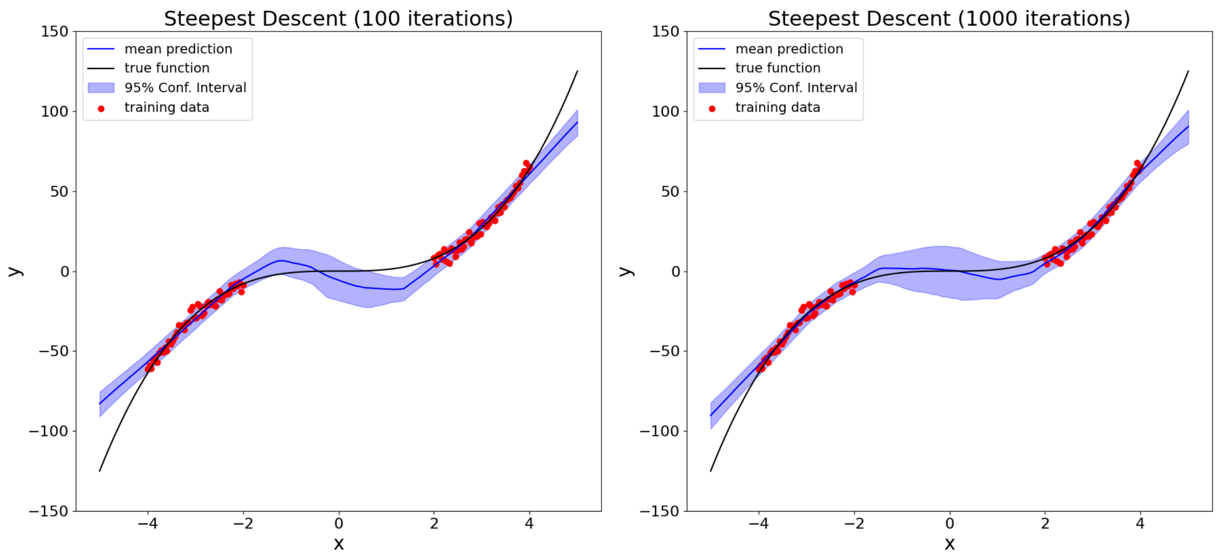

Figure 6: Results from the LUNA model using 100 iterations (left) and 1000 iterations (right). Steepest descent was used as the optimization algorithm. The shaded blue region represents the 95% CI.

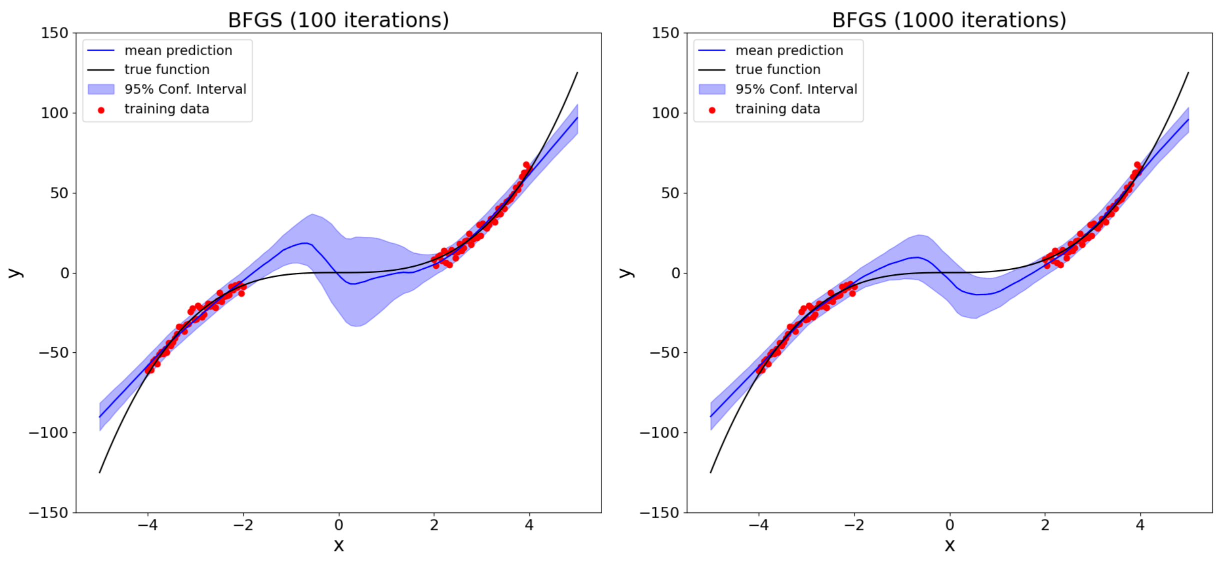

Figure 7: Results from the LUNA model using 100 iterations (left) and 1000 iterations (right). BFGS was used as the optimization algorithm. The shaded blue region represents the 95% CI.

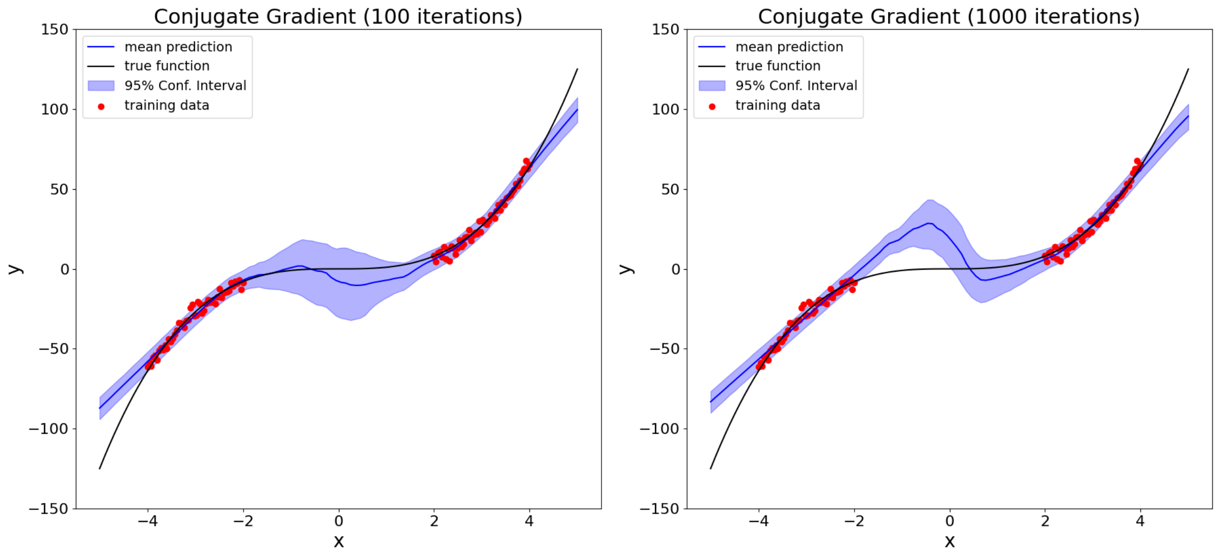

Figure 8: Results from the LUNA model using 100 iterations (left) and 1000 iterations (right). The conjugate gradient method was used as the optimization algorithm. The shaded blue region represents the 95% CI.

3.4 Modifying Finite Differences

3.4.1 Initial Approach

When examining the LUNA framework, we noted that the authors used finite differences to approximate gradients used in the cosine similarity function: their training objective. The equations 5 and 6 below are taken from Appendix B of Thakur et al. (2020) [7].

$$\text{CosSim}^2(\nabla_{\mathbf{x}} f_i(\mathbf{x}), \nabla_{\mathbf{x}} f_j(\mathbf{x}))^2 = \frac{(\nabla_{\mathbf{x}} f_i(\mathbf{x})^T \nabla_{\mathbf{x}} f_j(\mathbf{x}))^2}{(\nabla_{\mathbf{x}} f_i(\mathbf{x})^T \nabla_{\mathbf{x}} f_i(\mathbf{x}))(\nabla_{\mathbf{x}} f_j(\mathbf{x})^T \nabla_{\mathbf{x}} f_j(\mathbf{x}))} \tag{5}$$

$$\nabla_{\mathbf{x}} f_i(\mathbf{x}) \approx \frac{f_i(\mathbf{x}+\delta\mathbf{x}) - f_i(\mathbf{x})}{\delta\mathbf{x}} \tag{6}$$

We examined several ways to modify the calculations, ultimately deciding on the following:

1. Substituting forward finite difference for backward finite difference or central finite difference calculations
2. Using a constant value for $\delta\mathbf{x}$ in Eq. 6 which is originally sampled from a normal distribution $\delta\mathbf{x} \sim N(0, \sigma^2)$ where $\sigma = 0.1$
3. Instead of calculating the gradients of functions based on all observations, sampling a random selection of observations by their indices

Running a simple scenario with a test case with a NLM of 1 input dimension, 2 hidden layers, a width of 8, an output dimension of 5, and 1500 iterations, we determined that further analyzing the first method wasn't fruitful. The values were identical and the calculation times were not different, even with modified conditions. Despite the fact that central difference methods are second order accurate, it did not seem to impact the results, so we focused on the other two modifications.

3.4.2 Fixed $\delta\mathbf{x}$ in Finite Difference Calculations

Whereas the base LUNA model randomly samples over a Gaussian to obtain a matrix of step sizes to apply to each layer, we wanted to experiment with a matrix of a single identical step size applied to every observation in the layer. We wanted to see if this reduced computational time or resulted in better uncertainty estimates. I.e. Instead of randomly sampling step sizes that could potentially be large, why not always use a really small step size that doesn't compromise machine precision?

To experiment, we ran a modified LUNA using a fixed $\delta\mathbf{x}$ = 0.1, 0.001, and 0.0001 for 3500 iterations. These values were chosen because the original method sampled from a normal distribution using a mean of 0 with a standard deviation of 0.1. The results are shown in Figure 9. Although the time to calculate each scenario took less time than the base model of LUNA with the same parameters – roughly on order of 1 hour as opposed to 90 minutes shown in Figure 2 – the models had slightly worse log-likelihood.

Overall, the advantage of using a fixed value of $\delta\mathbf{x}$ was limited to its slightly reduced computational time, though more experiments are needed to verify that a fixed delta speeds training time. Unfortunately, the faster model training time may cause LUNA's predictive uncertainty to be less reliable. Looking at the plot of the confidence interval in 9, the 95% confidence region varies significantly depending on step size. We're cautious to draw too many conclusions about this picture because this variability could be due to the randomness involved in optimizing LUNA's objective function.

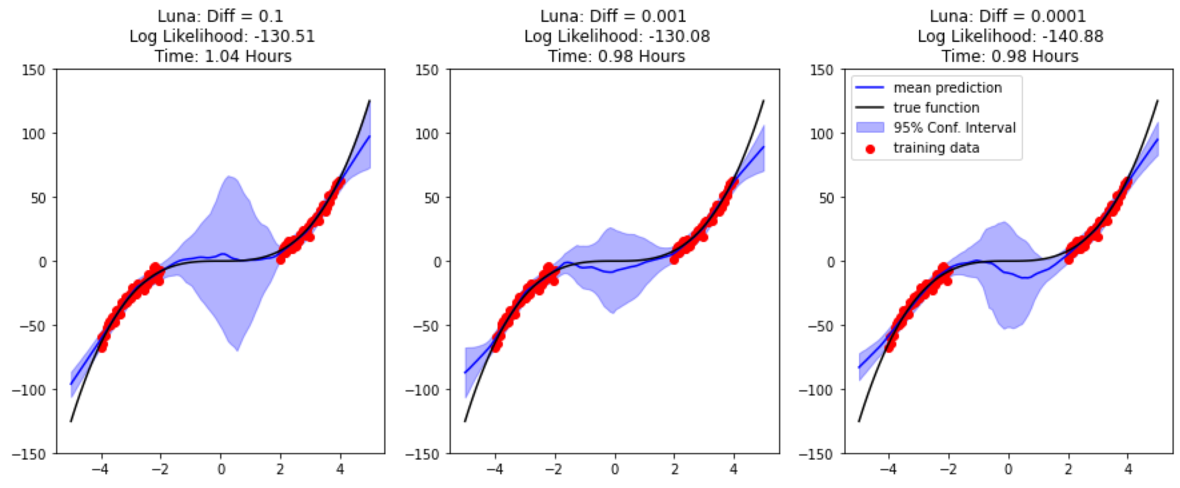

Figure 9: LUNA models given various fixed values of $\delta \mathbf{x}$ in the finite difference calculation for approximating the gradients.

Given the variation in mean prediction from the true function, hese results appeared to show that the smaller step sizes are more susceptible to the other randomness of the function (i.e. how the model function used a random ϵ sampling). This observation could be a starting point for future research on the topic of fixed step sizes.

3.4.3 Random Sampling of Indices for Gradient Calculations

In searching for further ways to improve LUNA's speed, we looked again at how the gradient was calculated for every observation in the data set in diversity score. We wanted to evaluate how reducing the number of gradients calculated would affect the speed and accuracy of LUNA. We assumed that calculating fewer finite differences would speed up the calculation.

To set up this experiment, we modified LUNA to compute the diversity score using a radnom subset of training observations. We trained 5 different LUNA models, each of which sampled a fixe proportion of training observations: 1%, 10%, 20%, 40%, 60%, and 80%. Using our standard configuration for hyperparameters, we ran 3500 iterations.

We plot the predictive uncertainties for each model in Figure 10 and compare the log-likelihood over the sample proportion in Figure 11. We were surprised that the results did not show a significant reduction when we sampled fewer observations. This series of 6 models only took approximately 50 minutes to run, which was much quicker than the 90 minutes required to run the base LUNA model (Figure 2). This discovery led us to consider that each group members' local systems may have impacted the run times.

Figure 11 is the best visual for this experiment. While it doesn't show anything unexpected, it does demonstrate that an increase in the number of observation indices sampled improves the log-likelihood until approximately the 40% mark. Again – we were limited by computational

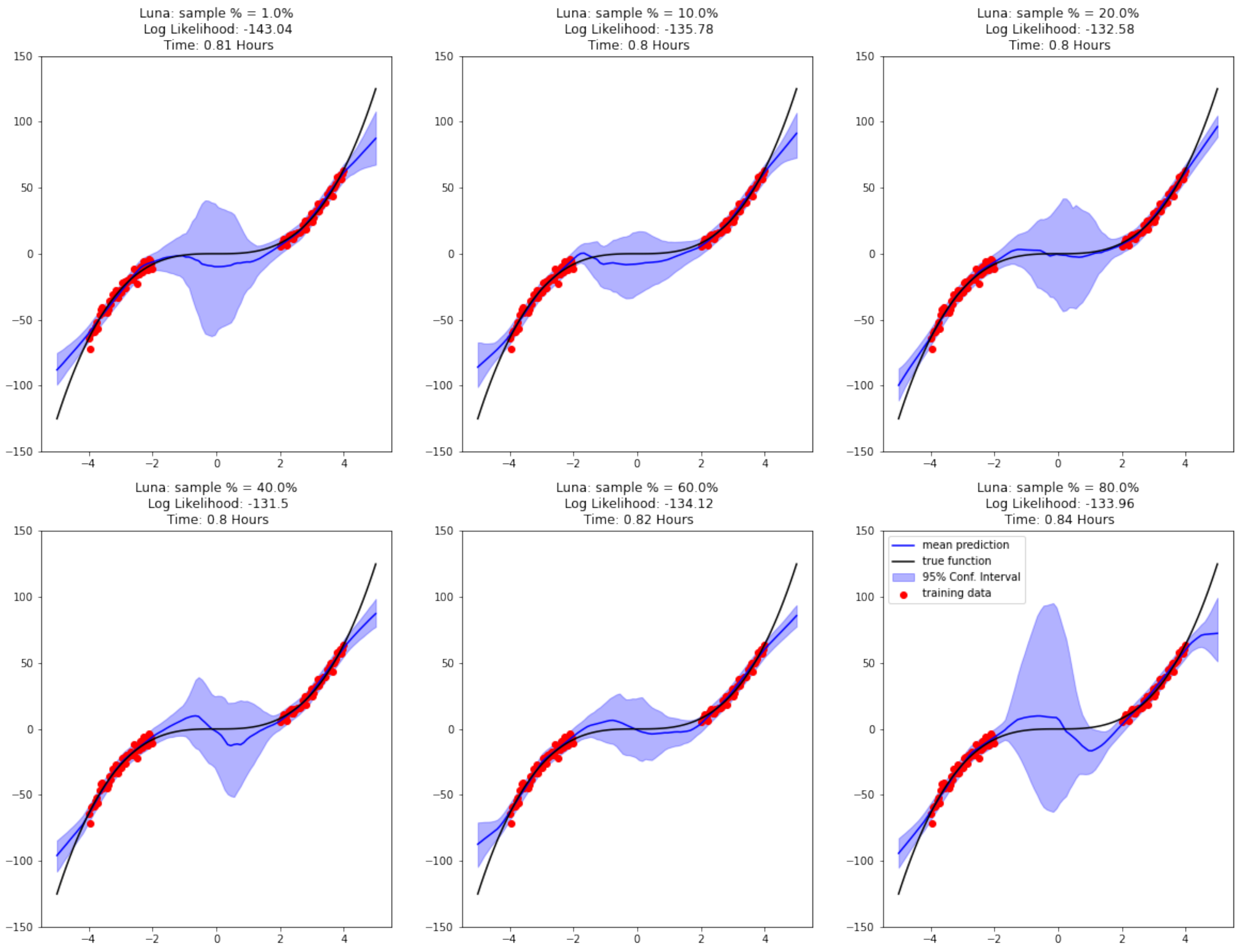

Figure 10: LUNA models given various percentages of random indices chosen for the finite difference calculation for approximating the gradients.

power, so the curve is very jagged, but the characteristics were interesting to see. Of note, if the base LUNA model's log-likelihood were included on the plot, the line would jump from -134 to -120, showing LUNA's superior accuracy when all observations are sampled for the gradient finite difference calculation.

To conclude, the sampling fewer observations didn't significantly affect run time. Uncertainty (measured by the 95% confidence interval shaded region), did seem to reduce over time, but at the 80% sample model, uncertainty increased again and the overall model fit looked more similar to the base LUNA model. We suspect that observation sampling could significantly affect run time if the training data set were larger.

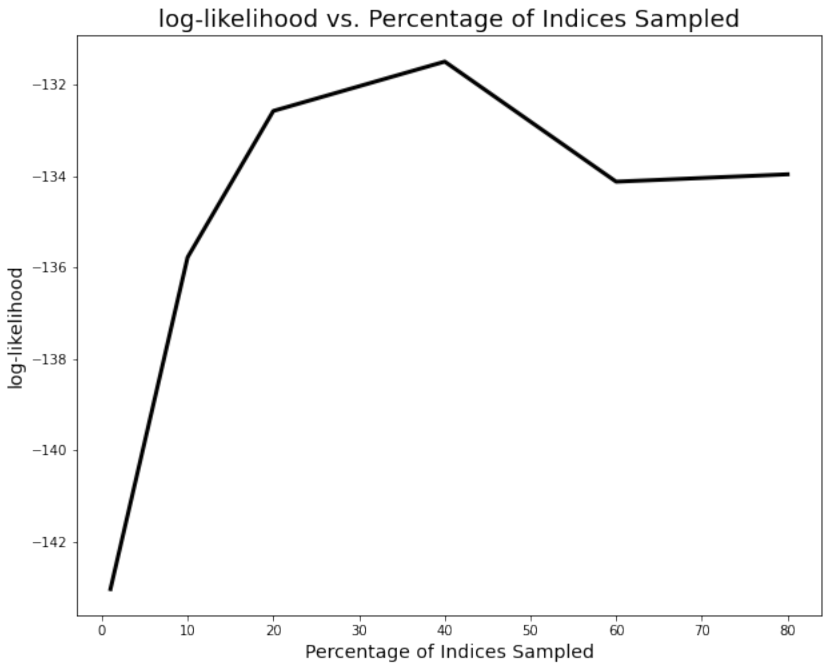

Figure 11: The log-likelihood of LUNA models based on 6 different percentages of random indices sampled: 1%, 10%, 20%, 40%, 60%, and 80%.

4 CONCLUSION AND FUTURE DIRECTION

In this project, we constructed the LUNA model (a class of Neural Linear Models) and experimented with alternative numerical implementations. We primarily focused on altering two areas of the model: 1) the specific optimization method used in training the neural network component and 2) the finite difference approach used for calculating the gradient in a key term of the objective function. Our results for the optimization method experiments suggest that simpler, first-derivative calculations are more efficient for fitting a neural network than methods that require calculation of the Hessian, which is an inherently computationally intensive exercise. Furthermore, heavier optimization techniques, either Newton or quasi-Newton methods, that typically exhibit better performance do not necessarily perform well in the context of neural network training. This observation highlights the importance to considering a function's structure in tandem with any search algorithm's properties when choosing an optimization method. We anticipated that finite difference modifications would not improve the accuracy of the models, which was confirmed. However, we thought that finite difference simplifications (fixed step and sampling fewer observations) would reduce the runtimes of training the models, and our results did not support that hypothesis.

Given that the LUNA model was only introduced within the last year, there are several avenues of exploration after our introductory evaluation. First, we felt that some of our results were inconclusive due to computational power. We would like to see the results of more iterations conducted on a cluster where the results would be standardized by the computational power of a single system, instead of spread across 4 graduate students' personal laptops. Perhaps this would show the reduced computational time we expected to see with simpler optimizers and fewer calculations of finite difference.

Third, we could examine further research done to improve the speed of large neural nets in general and apply those lessons learned. Although Thakur et al used cosine similarity as a penalty function on the last layer of auxiliary functions, we could explore another method that encourages diversity but is computationally faster [7].

Finally, we would like to observe how our LUNA model (with modifications) performs on real data. Thakur et. al's paper highlights the use of toy problems and common machine learning data sets from UC Irvine's repository. We would like to see how our recreated model performs in comparison.

5 APPENDIX

A link to our GitHub repo is below:

`https://github.com/vsavram/AM205-Project`

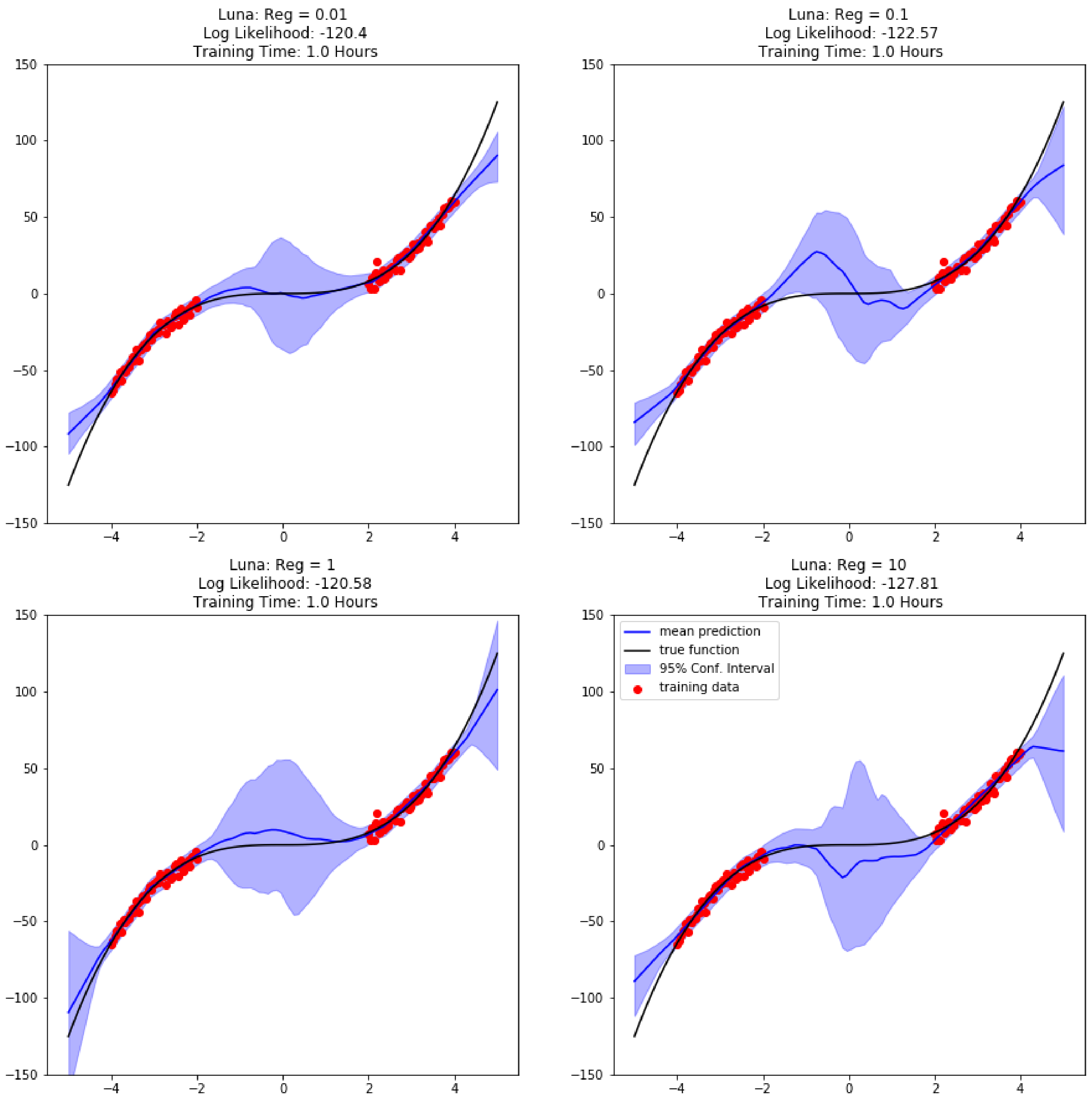

Figure 12: This Figure shows that LUNA models trained across a variety of regularization value maintain good uncertainty estimates

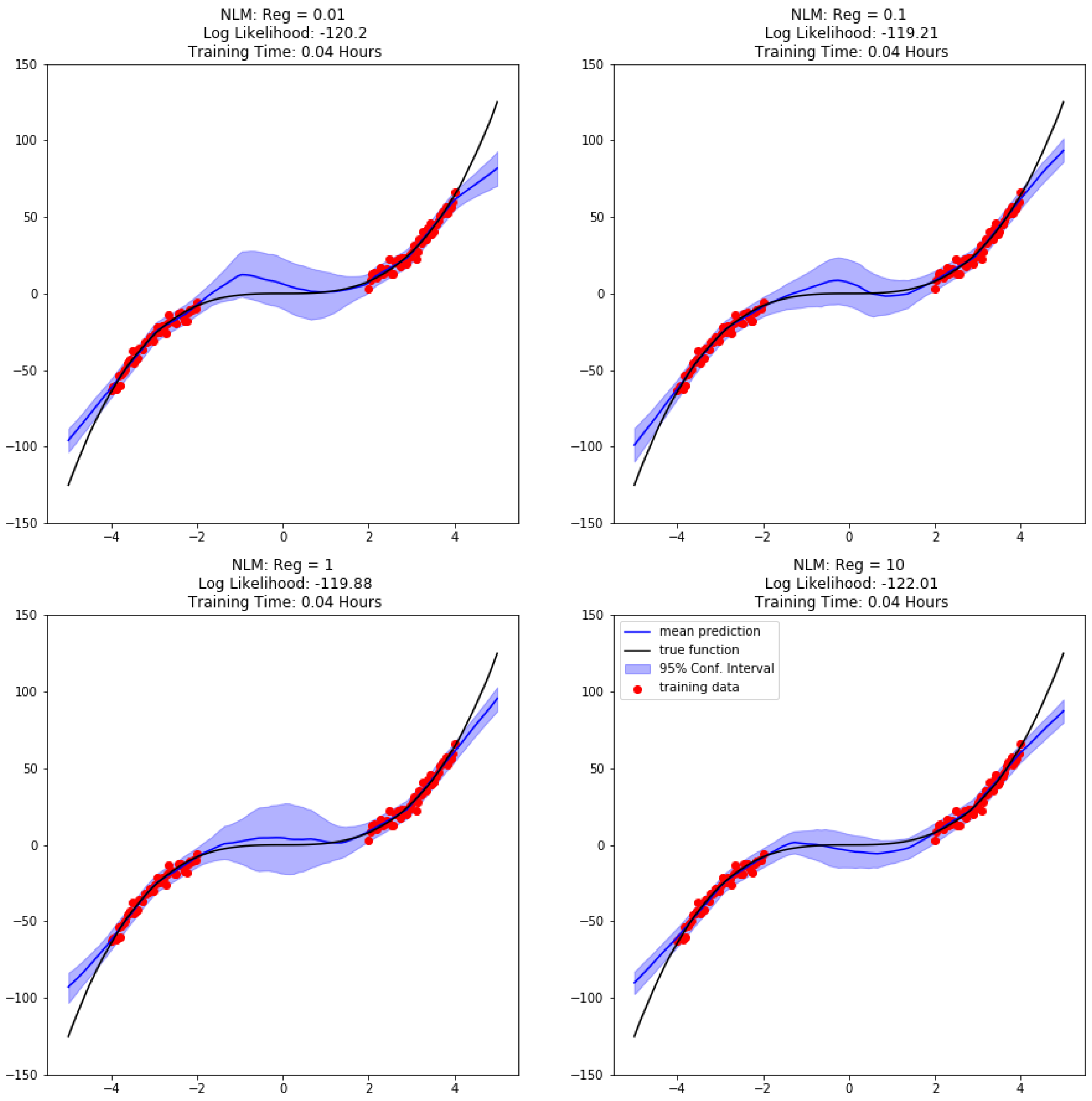

Figure 13: This Figure shows that NLM models trained across a variety of regularization values return finniky uncertainty estimates

REFERENCES

[1] Kashmir Hill. "Wrongfully Accused by an Algorithm". en-US. In: *The New York Times* (June 2020). ISSN: 0362-4331. URL: `https://www.nytimes.com/2020/06/24/technology/facial-recognition-arrest.html` (visited on 12/14/2020).

[2] Gareth James et al. *An Introduction to Statistical Learning*. en. Vol. 1. Springer Texts in Statistics. New York, NY: Springer New York, 2013. ISBN: 978-1-4614-7137-0 978-1-4614-7138-7. DOI: `10.1007/978-1-4614-7138-7`. URL: `http://link.springer.com/10.1007/978-1-4614-7138-7` (visited on 12/13/2020).

[3] Alex Kendall. *Deep Learning Is Not Good Enough, We Need Bayesian Deep Learning for Safe AI*. en. May 2017. URL: `//alexgkendall.com/computer_vision/bayesian_deep_learning_for_safe_ai/` (visited on 12/14/2020).

[4] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Jan. 2017). arXiv: 1412.6980. URL: `http://arxiv.org/abs/1412.6980` (visited on 11/05/2020).

[5] Radford M. Neal. *Bayesian Learning for Neural Networks*. en. Ed. by P. Bickel et al. Vol. 118. Lecture Notes in Statistics. New York, NY: Springer New York, 1996. ISBN: 978-0-387-94724-2 978-1-4612-0745-0. DOI: `10.1007/978-1-4612-0745-0`. URL: `http://link.springer.com/10.1007/978-1-4612-0745-0` (visited on 12/13/2020).

[6] Jasper Snoek et al. "Scalable Bayesian Optimization Using Deep Neural Networks". In: *arXiv:1502.05700 [stat]* (July 2015). arXiv: 1502.05700. URL: `http://arxiv.org/abs/1502.05700` (visited on 12/13/2020).

[7] Sujay Thakur et al. "Learned Uncertainty-Aware (LUNA) Bases for Bayesian Regression using Multi-Headed Auxiliary Networks". In: *arXiv:2006.11695 [cs, stat]* (July 2020). arXiv: 2006.11695. URL: `http://arxiv.org/abs/2006.11695` (visited on 11/01/2020).

[8] Weilin Zhou and Frederic Precioso. "Adaptive Bayesian Linear Regression for Automated Machine Learning". In: *arXiv:1904.00577 [cs, stat]* (Apr. 2019). arXiv: 1904.00577. URL: `http://arxiv.org/abs/1904.00577` (visited on 12/13/2020).