Bayesian Uncertainty Quantification via Machine Learning

Student: Lindsey Schneider[1]
Mentors: Vincenzo Cirigliano[1] and Yukari Yamauchi[1]
[1]Institute for Nuclear Theory, University of Washington, Seattle, WA 98195, USA

September 6, 2024

Abstract

Uncertainty quantification (UQ) is an important step in physics research. In order to bridge the gap between experimental data and theoretical predictions, it is essential to calibrate model parameters according to their uncertainty. The model parameters are represented by a posterior distribution determined by Bayesian analysis. This means that calculating the uncertainty requires sampling from from a high dimensional probability distribution, so it is essential to use a sampling method which is efficient. Using a normalizing flow constructed by neural network training instead of Markov Chain Monte Carlo sampling is a useful way to save time and produce accurate results.

1 Project Introduction

One of the most important tasks in physics research is to build a connection between experiment and theory. One way this can be done is through predictions made by Bayes' Theorem:

$$P(\omega \boldsymbol{j} D) = \frac{P(D \boldsymbol{j} \omega) P(\omega)}{P(D)} \ , \tag{1}$$

where ω is the parameters in the theoretical model we are calibrating, and D represents experimental data. The posterior distribution $P(\omega \boldsymbol{j} D)$ is a product of the likelihood of observing the experimental data $P(D \boldsymbol{j} \omega)$ and the prior beliefs based on theory $P(\omega)$. The posterior gives us insight into how experiment aligns with previous theoretical predictions. The process of calibrating theoretical predictions based on experimental data is called Uncertainty Quantification (UQ) [1].

Rather than using a single number for the outcome of a model parameter, Bayesian analysis constructs a probability distribution. So, you need to be able to sample from it in order to quantify the uncertainty of your model parameters and calibrate theory and experiment accordingly. This is an ongoing process since new experiments will be conducted and calibration will be repeated. Thus, it is important that we can sample from a high dimensional distribution effectively. There are two approaches to perform this sampling; the traditional approach is to use Markov Chain Monte Carlo (MCMC) sampling, and the other is to sample from a normalizing flow. However, the traditional approach has several downsides which the new approach will fix.

1.1 Markov Chain Monte Carlo (MCMC) Sampling

MCMC, also called the Metropolis-Hastings algorithm, is a sampling method in which you start from a random point, take a random walk, and decide whether or not you are going to take that sample. There are two main issues with this method. The first issue is that this method takes a long time. It is inherently sequential since all samples are dependent on the previous sample, so there is no optimization that can be made by using multiple threads to take samples. The second issue is that samples are not independent. As stated above, all samples are dependent on the last sample, therefore the samples will be highly correlated [2].

1.2 Normalizing Flows

A normalizing flow is a map from a Gaussian distribution to a non-trivial distribution. A normalizing flow f is the change of variables:

$$dx \ p(x) = dy \ \det(\frac{\partial f(y)}{\partial y}) \ p(f(y)) = dy \ g(y) \tag{2}$$

where $x = f(y)$ [1]. To sample from a normalizing flow, you need to sample from the Gaussian distribution g and then apply the map to get samples from the goal distribution p.

1.3 MCMC vs Normalizing Flows

The advantage of using a normalizng flow is that it fixes the two issues with MCMC sampling. The first issue that it takes a long time is fixed because we can utilize using multiple threads since we are sampling from a Gaussian distribution. The second issue that samples are not independent is also fixed through sampling from a Gaussian distribution.

2 Project Progress

This summer I have been working on constructing normalizing flows for the two dimensional Rosenbrock-banana distribution

$$p_b(\alpha_1, \alpha_2) = e^{\quad (\ \ 1\ \ 1)^2\ \ 2(\ \ \frac{2}{1}\quad\ \ 2)^2}\ . \tag{3}$$

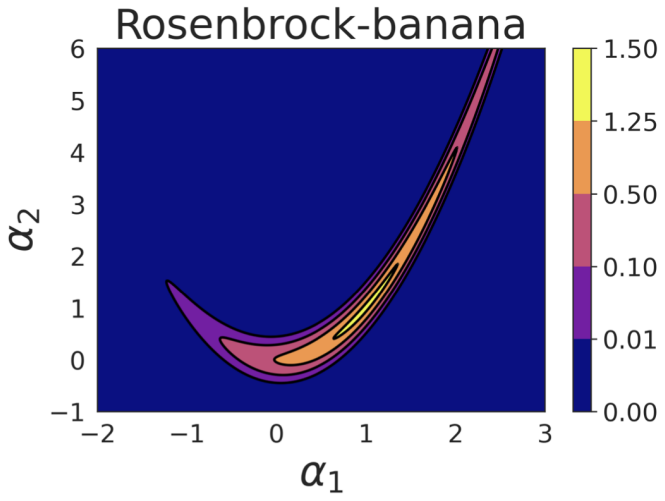

Figure 1: Rosenbrock-banana Distribution

After testing and verifying that this produces a feasible result, we will move on to testing higher dimensional distributions in order to use normalizing flows to sample from an 18 parameter neutrino model [3].

Machine learning techniques are used to construct normalizing flows. Using a neural network, we will train various parameters to create a map from the Gaussian distribution to our goal distribution. The first task is to figure out what classes we need to build the neural network. In the python code I developed, I created the following classes: MLP, Affine, SS_layer, RealNVP [4], Train.

2.1 Multi-Layer Perceptron (MLP)

An MLP object consists of alternating linear parameter layers and non-linear activation function layers. For an n-dimensional problem, we want n inputs and n outputs for each linear parameter layer since we want a mapping for each parameter from Gaussian space to our goal distribution. Each activation layer has one input and one output, and we apply the activation to each of the n inputs. The parameters we tune throughout the training process are contained in the linear parameter layers.

We include the activation layer in order to capture non-trivial aspects of the distribution. It was important when implementing this class to choose an effective activation function. One function that was tested was the CELU function.

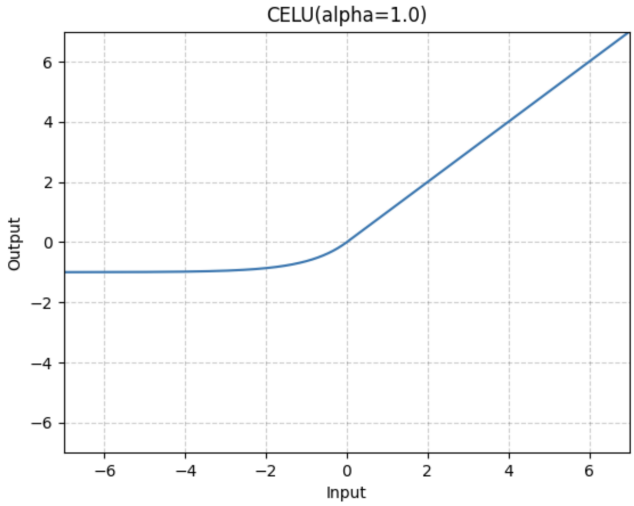

Figure 2: CELU Function [5]

When running training for this class, the loss became NaN (Not a Number) at certain steps. NaN occurs at very large or very small numbers dues to the computers ability to store such numbers. Since the CELU function linearly grows at large inputs, the loss function became very large, therefore it became NaN. This is not a good activation function to use since we must handle large inputs and produce a reasonable result for the loss function. The activation function that worked the best was the tangent hyperbolic function.

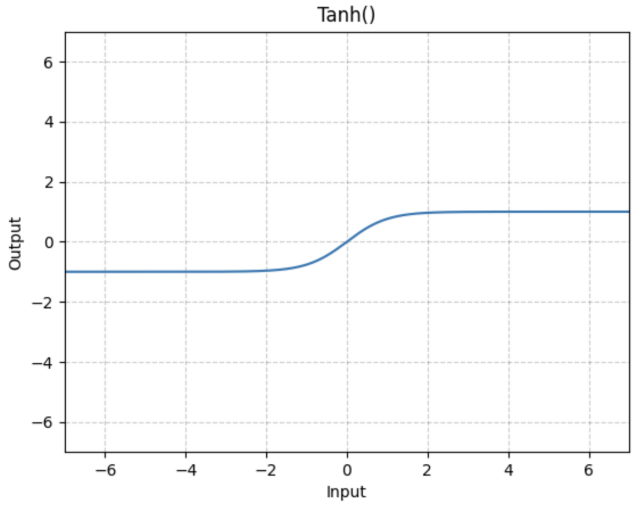

Figure 3: Tanh Function [6]

This is a good function to choose because there are bounds on the inputs to the function such that it will not result in an unreasonable output.

2.2 Affine

An Affine object consists of a scaling MLP, a shifting MLP, and a masking vector. These objects are used as follows:

$$x = m \quad y + ([1,1] \quad m) \quad (y \quad e^{\mathsf{S}(m \;\; y)} + T(m \quad y)) \;, \tag{4}$$

where y is a sample from the Gaussian distribution, x is the output in the goal distribution space, and denotes element-wise multiplication. The two MLP networks are then tuned throughout the training process. The purpose of the masking vector m is to isolate a few variables at a time, and only those variables will see a change. For example, if you have a two dimensional distribution and you use the masking vector $m = [1, 0]$, then your output will be:

$$\begin{aligned} x_1 &= y_1 \\ x_2 &= y_2 \quad (e^{\mathsf{S}([y_1;0])_1}) + T([y_1, 0])_1 \ . \end{aligned}$$

Using multiple Affine layers with various masking to train a normalizing flow is useful since masking different variables at each layer will ensure the tuning of parameters for one variable does not affect other variables.

2.3 SS_Layer

The purpose of a final scaling and shifting linear layer is to capture the overall size and spread of parameters. The prior distribution $P(\omega)$ tells us where the posterior distribution is contained, therefore we want our normalizing flow to be in this region. At the final step of training, a fixed scaling/shifting factor will be applied based on the prior.

2.4 RealNVP

The RealNVP class [4] consists of Affine layers with varying masking vectors and a final scaling and shifting layer (SS_Layer). The code I developed for two dimensional distributions consists of 12 Affine layers alternating between $[1, 0]$ and $[0, 1]$ masking.

2.5 Train

The training class contains a RealNVP object to be tuned, and a callable function of the posterior distribution. It is assumed that the posterior distribution is known and we can get the value of it at a given value of ω. At every step of training there must be a way to determine how much the constructed normalizing flow differs from the goal distribution. This is called the loss function, and it will be used to tune the parameters in the normalizing flow. At every step of training, the loss function should approach zero since the goal is to get as close to the goal distribution as possible. The loss function used in my code is based on Jeffrey's Divergence [7]:

$$D_J(P, \pi) = \int d\omega \ (\tilde{P}(\omega) log(\frac{P(\omega)}{\pi(\omega)}) + \tilde{\pi}(\omega) log(\frac{\pi(\omega)}{P(\omega)})) \ , \tag{5}$$

where π is the normalizing flow, P is the goal distribution, and $\tilde{\pi}$ and $\tilde{P}$ are normalized distributions of the normalizing flow and goal distribution respectively. The second term, $\tilde{\pi}(\omega) log(\frac{(!)}{\mathsf{P}(!)})$, is the reverse-KL Divergence. It is often called the unsupervised learning term since it samples from the normalizing flow π. We can do this sampling since a normalizing flow can be sampled using a Gaussian distribution. However, to sample from the first term we will need to use reweighting. The first term, $\tilde{P}(\omega) log(\frac{\mathsf{P}(!)}{(!)})$, is the KL Divergence. It is often called the supervised learning term since it samples from the goal distribution P. We cannot easily do this sampling since sampling from the goal distribution using MCMC is what we are trying to avoid by using normalizing flows.

So to resolve this, we will perform a reweighting on this term. The KL Divergence is estimated as

$$D_{KL}(P,\pi) = \frac{\int d\omega\ P(\omega)\log\frac{P(!)}{(!)}}{\int d\omega\ P(\omega)} = \frac{\frac{\int d!\quad (!)\frac{P(\omega)}{\pi(\omega)}\log\frac{P(\omega)}{\pi(\omega)}}{\int d!\quad (!)}}{\frac{\int d!\quad (!)\frac{P(\omega)}{\pi(\omega)}}{\int d!\quad (!)}} = \frac{h\frac{P(!)}{(!)}\log\frac{P(!)}{(!)}i}{h\frac{P(!)}{(!)}i} \tag{6}$$

$$\frac{\frac{1}{N_s}\sum_{i=1;2}^{N_s}\quad\frac{P(!_i)}{(!_i)}\log\frac{P(!_i)}{(!_i)}}{\frac{1}{N_s}\sum_{i=1;2}^{N_s}\quad\frac{P(!_i)}{(!_i)}} = \frac{\frac{1}{N_s}\sum_{i=1;2G}^{N_s}\frac{P(f(y_i))}{\det\left(\frac{\partial y_i}{\partial\omega_i}\right)G(y_i)}\log\frac{P(f(y_i))}{\det\left(\frac{\partial y_i}{\partial\omega_i}\right)G(y_i)}}{\frac{1}{N_s}\sum_{i=1;2G}^{N_s}\frac{P(f(y_i))}{\det\left(\frac{\partial y_i}{\partial\omega_i}\right)G(y_i)}}\ .$$

At every step of training, the derivative of the loss with respect to each parameter will be calculated in order to tune those parameters. The parameters will be tuned by this derivative multiplied by a fixed learning rate. In the code I developed, I used a learning rate of 1e-3. It is important to not over-train, so the learning rate cannot be too large.

2.6 Results

After training a normalizing flow using python code, I graphed 10,000 normalizing flow samples against 10,000 MCMC samples.

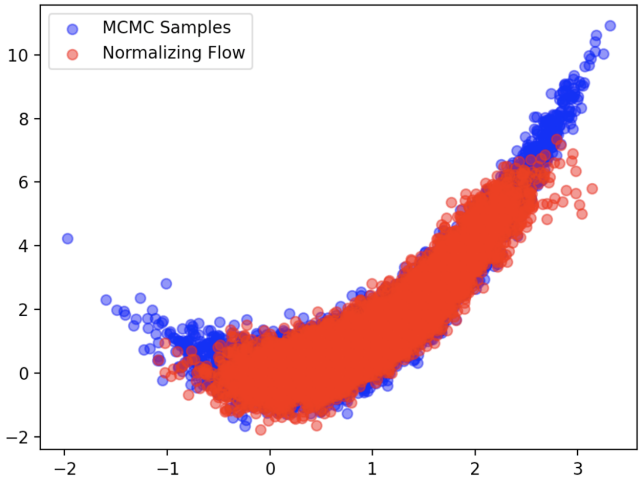

Figure 4: MCMC vs Normalizing Flow

This gives us an idea of how well MCMC sampling and normalizing flows agree with each other. Visually, they appear to agree. However, we need a more concrete way of determining whether this result is accurate to the true distribution. To do this, we must calculate a mean value and standard deviation for the samples. Using this, we will see whether it agrees with the expected true mean. The expected value for the x_0 parameter is 1.0, and the expected value for the x_1 parameter is 1.5. After taking 10,000 samples 100 times and calculating the mean and standard deviation, the result is $x_0 = 0.989 \quad 0.005$, and $x_1 = 1.33 \quad 0.01$. This does not agree with the expected value. To correct this, we will reweight the final results. Reweighting needs to be performed for the final result in order for the error bars to account for how much the mean value differs from the true

value. As mentioned in the training section, we use reweighting to estimate sampling from the goal distribution. Since we have access to the goal distribution and its value at a given point, we can use this to determine how much the calculated mean differs from the true goal distribution. The parameters are tuned as follows:

$$x_0 = \frac{\sum x_0 \quad KL}{\sum KL} \tag{7}$$

$$x_1 = \frac{\sum x_1 \quad KL}{\sum KL} \tag{8}$$

Where KL is the ratio P/π which appears in the integrand of equation (5).

Below is a graph of each parameter before reweighting and after reweighting at every 500 training steps.

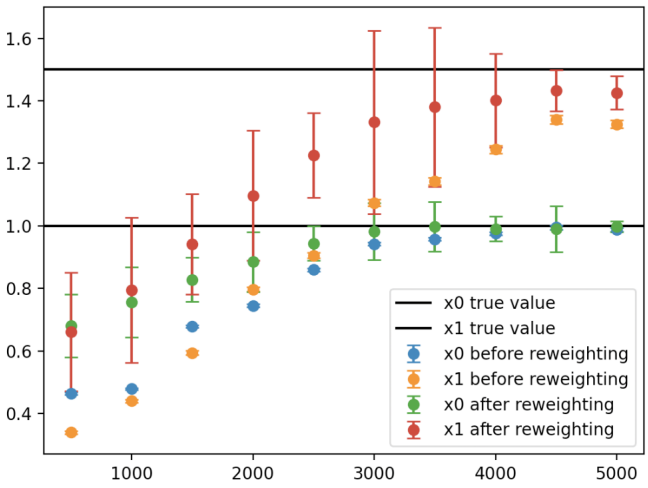

Figure 5: Mean Reweighted Samples

Starting at step 3,000 the reweighted averages for both parameters agree with the expectation value. After step 4,500 the reweighted averages do not agree with the expectation values. This is due to over-training. One potential way to resolve this is to reduce the learning rate after a certain step.

2.7 Benefits of Using NF over MCMC

Based on the graph above, we can get a value from normalizing flows that agrees with the expected value. The benefit of using normalizing flows over MCMC is that it saves a lot of time. The graph above took around 10 minutes to generate. This includes sampling at each training step, and sampling for the reweighted averages. On the other hand, the MCMC samples took an hour to generate. This is a huge advantage of using normalizing flows.

3 Future Directions

Now that we have tested the two dimensional distribution, we can move on to testing higher dimensional distributions. To do this we will use a distribution which we can calculate the mean

value of the model parameters by hand and see whether the normalizing flow agrees with the values, such as we did with the Rosenbrock-banana distribution.

Another important step in this project is to make sure the python code we developed is compatible with other programming languages. If we have a probability distribution which was coded in another programming language, we want to make sure that we can still use our python code to construct a normalizing flow for that distribution. The goal in developing our code is for researchers to use it, so it is essential that it can be used given any distribution regardless of its implementation.

Once we can use our code with any distribution, we will train a normalizing flow for the Beyond the Standard Model with 3 right-handed neutrinos [3]. We want to perform Bayesian analysis for this model in order to make predictions for neutrinoless double beta decay. Since this model has 18 parameters, performing MCMC sampling will take a very long time. We will show that using normalizing flows achieves the physics goal faster that the traditional method.

Finally, we would like to create an online platform to share the resources we have developed.

Acknowledgement

I would like to thank the INT for their support of this research through the U.S. Department of Energy grant No. DE-FG02-00ER41132. I would also like to thank the N3AS for their support through the National Science Foundation award No. 2020275.

References

[1] Yukari Yamauchi et al. *Normalizing Flows for Bayesian Posteriors: Reproducibility and Deployment*. Oct. 2023. arXiv: 2310.04635 [nucl-th].

[2] Christof Gattringer and Christian B. Lang. *Quantum chromodynamics on the lattice*. Vol. 788. Berlin: Springer, 2010. ISBN: 978-3-642-01849-7, 978-3-642-01850-3. DOI: 10.1007/978-3-642-01850-3.

[3] Marcin Chrzaszcz et al. "A Frequentist analysis of three right-handed neutrinos with GAMBIT". In: *The European Physical Journal C* 80.6 (June 2020). ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-020-8073-9. URL: http://dx.doi.org/10.1140/epjc/s10052-020-8073-9.

[4] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. *Density estimation using Real NVP*. 2017. arXiv: 1605.08803 [cs.LG]. URL: https://arxiv.org/abs/1605.08803.

[5] PyTorch Contributors. *CELU*. 2023. URL: https://pytorch.org/docs/stable/generated/torch.nn.CELU.html (visited on 08/27/2024).

[6] PyTorch Contributors. *Tanh*. 2023. URL: https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html (visited on 08/27/2024).

[7] Harold Jeffreys. "An invariant form for the prior probability in estimation problems". In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 186.1007 (1946), pp. 453–461. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1946.0056.