Python-Based Implementation of Machine Learning Pseudo-Random Noise Code Generation

Adam Moec
Purdue University

February 19, 2025

Abstract

Pseudo-Random Noise (PRN) codes transmitted by GNSS satellites play a critical role in satellite identification and communication delay estimation, both of which are central to accurate positioning. These aspects of GNSS rely on the PRN codes' low cross-correlation and low non-central mean auto-correlation. Currently, GPS C/A PRN codes use Gold Codes (GCs), which leverage a closed-form mathematical model for the generation of these unique, 1023-chip long codes. Their ability to be created using 10-stage Linear Feedback Shift Registers (LFSRs) also allowed for mass and cost savings for memory when the first generation of GPS was developed in the 1970s. Since memory constraints necessitating these designs have become negligible with modern technology, the door to using locally-stored memory codes has opened. Recent research has explored the use of genetic and evolutionary algorithms to generate new families of memory codes with improved correlation properties. This paper demonstrates a viable Python implementation of the evolutionary algorithm used by Mina and Gao, showing similar performance and only minor deviations with the published literature.

Nomenclature

x	=	Sampled value of a Gaussian distribution [-]
$\bar{x}$	=	Chip value (either 0 or 1) [-]
$\bar{\mu}$	=	Gaussian distribution mean [-]
σ	=	Gaussian distribution standard deviation [-]
ℓ	=	Code sequence length [-]
K	=	Number of codes per code family [-]
$\mathcal{I}$	=	Identity matrix
$\mathcal{N}$	=	Cardinality of the batch of samples [-]

I. Introduction

Global Navigation Satellite Systems (GNSS), such as the Global Positioning System (GPS), have become integral to modern technology, supporting both commercial and security interests ranging from navigation to mapping, Earth atmospheric observation, targeting, and telecommunications. The global dependence on GNSS technology amplifies the impact of any improvements, making it a critical area of research. A crucial component of GNSS functionality is the use of Pseudo-Random Noise (PRN) codes, which enable satellites to transmit unique signals identifiable to receivers on Earth. The generation and optimization of these PRN codes are essential for accurate positioning and reliable signal acquisition.

Traditionally, model-based methods such as the Kalman filter [1] and Newton-Raphson algorithms [2] have been employed to mitigate errors in GNSS positioning. These methods rely on mathematical models that assume certain statistical properties of signal noise, often simplifying its distribution as Gaussian or uniform [3]. While sufficient in many scenarios, these assumptions cannot account for complex errors observed in signal reflections and multipath effects common in urban environments. Mohanty and Gao [4] highlight these limitations while describing the potential for Machine Learning (ML) and Reinforcement Learning

(RL) techniques to overcome some of the constraints imposed by traditional mathematical models.

Advancements in ML and RL over the past decades offer promising alternatives to traditional methods. ML algorithms excel at modeling complex, non-linear relationships and can learn from data patterns without strict statistical assumptions [5]. Siemuri et al. [6] identify several areas within the GNSS domain where ML could enhance accuracy and efficiency, including signal acquisition, detection and classification, Earth observation and monitoring, navigation and precise positioning, and indoor navigation.

Despite these advancements, the application of ML techniques to PRN code generation has been relatively unexplored. Recognizing this gap, Mina and Gao introduced a novel approach using the Natural Evolution Strategies (NES) algorithm to generate PRN code families with improved non-central autocorrelation and cross-correlation properties compared to existing codes. Their work demonstrates the potential of evolutionary algorithms in optimizing PRN codes for enhanced GNSS performance.

This project aims to recreate the ML model using the NES algorithm as described by Mina and Gao, in order to validate and potentially extend their findings. By implementing their approach, we seek to generate PRN code families with better correlation properties, which are crucial for reducing interference and improving signal acquisition in GNSS receivers. This work not only replicates the original study but also provides insights into practical implementation challenges and potential areas for further enhancement.

The remainder of this paper is organized as follows. Section II details the implementation of the NES algorithm for PRN code generation. Section III presents the results of the models' training, comparing the generated codes with those obtained by Mina and Gao [7]. Section IV is dedicated to the analysis of these results, discussing their implications, limitations, and potential areas for improvement. Finally, Section V summarizes our conclusions and suggests directions for future research.

II. Method

This project builds on the research of Mina and Gao [7], who employed a Natural Evolutionary Strategy (NES) algorithm to optimize PRN code families with low cross-correlation and non-central autocorrelation properties. This method must therefore be carefully documented so as to maximize reproducibility and transparency in the process of the Python implementation. In this vein, this section begins with a comprehensive overview of the mathematical principles and network architecture employed by the authors of the original paper, as well as directly demonstrating the Python implementation of their principal functions. The model performance benchmarking strategy is also outlined, thus giving a complete overview as to the technical aspects of producing a ML model for PRN code generation.

A. Recreating the NES-Based ML Model for PRN Code Optimization

This author's approach re-implements this model to evaluate and generate optimized PRN code families by translating the high-level framework described in Mina and Gao's seminal paper into functional Python code. The translation of the principal mathematical functions used by Mina and Gao [7] to PyTorch-compatible functions are given below.

1. Model Description and NES Algorithm Overview

Mina and Gao's NES algorithm employs a proposal distribution over the design space of spreading codes. This proposal distribution allows the evaluation of randomly generated code sets using a stochastic performance evaluator, with results used to update network weights.

The recreated model follows the structure of Mina and Gao's NES framework given below:

- **Proposal Distribution Generation**: The model generates a proposal distribution from which individual PRN codes may be sampled. This proposal distribution can be defined as follows:

  $$p_\theta(x) \sim N(\mu(\theta), \sigma^2 \mathcal{I}^{(K\ell)}) \tag{1}$$

  Note that the parameter θ is what is actually optimized for during the NES algorithm backpropagation step described below. The standard deviation is kept constant at a value of 0.1 as described by Mina and Gao [7].

- **Gaussian Sampling and Chip Assignment**: Each chip value is sampled from an uncorrelated Gaussian distribution, which chip values assigned based on whether they exceed a threshold, ζ, determined by:

$$\zeta := \frac{\mu_L + \mu_U}{2} \tag{2}$$

  Where μ_L and μ_U are the minimal and maximal means for the Gaussian chip distributions respectively. If the i^{th} Gaussian distribution sampled value, x_i, is greater than or equal to this threshold value ζ then the corresponding i^{th} chip, $\bar{x}_i$, is assigned the value 1.

- **Objective Function Evaluation**: Following the sampling of several different code families from the proposal distribution function, the objective function has to be evaluated in order to determine the subsequent optimization step. Mina and Gao place equal weight in the cross-correlation and non-central mean autocorrelation; tuning their objective function to force the NES algorithm to optimize for both. The non-central mean autocorrelation function, f_{AC} is defined by Equation 3, while Equation 4 defines the cross-correlation function f_{CC}.

$$f_{AC} := \frac{1}{K\ell}\left(\sum_{k=1}^{K}\sum_{\delta=1}^{\ell-1}|R_k(\delta)|^2\right) \tag{3}$$

$$f_{CC} := \frac{1}{\binom{K}{2}\ell}\left(\sum_{k=1}^{K}\sum_{j=k}^{K}\sum_{\delta=0}^{\ell-1}\left|R_{k,j}(\delta)\right|^2\right) \tag{4}$$

  Note the binomial operator in the denominator of Equation 4 which is used to determine the average of the cross-correlation across all code pairs in a single code family and thus disregards autocorrelation. The objective function follows logically from the definition of the auto- and cross-correlation functions:

$$f := \max\left(f_{AC},\, f_{CC}\right) \tag{5}$$

  By selecting the maximum of f_{AC} and f_{CC} for a generated and sampled code family as the 'loss', this ensures that the NES algorithm focuses on reducing both types of correlation equally throughout the training process. Mina and Gao also note that this objective function could then easily be modified if lower cross-correlation was preferred for instance, by simply adding a multiplier to f_{CC} in the objective function.

- **Optimization Step**: Since the sampled code families are by definition discontinuous, gradient determination used in backpropagation cannot be automatically handled by Python ML libraries such as PyTorch. As such, a gradient estimate needs to be calculated before being employed in the optimization step. The gradient estimate used by Mina and Gao and first posited by Mohamed et al. [8] is described by Equation 6 below:

$$\nabla_\theta J(\theta) \approx \frac{1}{\mathcal{N}}\sum_{i=1}^{\mathcal{N}}\left(f(x^{(i)}) - b\right)\nabla_\theta \log(p_\theta(x^{(i)})) \tag{6}$$

  The parameter, b, serves as a baseline to approximate the traditional effect of normalization in ML. It is in effect the mean of the objective function across samples in the current batch, and given by the following equation:

$$b := \frac{1}{\mathcal{N}}\sum_{i=1}^{\mathcal{N}} f(x^{(i)}) \tag{7}$$

  As Mina and Gao point out, subtracting this parameter, b, does not introduce any bias in the gradient estimation and works to significantly reduce the variance. This in turn improves the training convergence as demonstrated in their paper [7].

2. Network Architecture

While most aspects of the NES algorithm GNN architecture were provided in Table 1 by Mina and Gao [7], some details such as the input layer dimensions and hidden layer activation functions remained unstated. As such a decision had to be made with regards to these final pieces of the network architecture in order to obtain a 'best guess' as to Mina and Gao's original choices. Due to its prolific use in machine learning applications, the rectified linear unit (ReLU) activation function was employed for all non-output layer neurons [9]. Additionally, the input layer was chosen to have the same size as that of the output layer, that is, $K \times \ell$. While not provided in Table 1, Mina and Gao do explicitly state the output layer to take the form $K \times \ell$, which of course makes sense considering the K codes, each with length ℓ required for each input vector [7].

Table 1 Design parameters of proposed NES machine learning method [7]

Parameter Description	Value
Gaussian proposal variance (σ^2)	0.1
Learning rate (for $\ell < 500$)	10^{-4}
Learning rate (for $\ell > 500$)	$5 \cdot 10^{-5}$
Hidden layer size	$2K\ell$
Number of hidden layers	2
Output activation	tanh
Network optimizer	Adam [10]
Batch size (N)	100
Number of iterations	10,000

It should be noted from the outset that this results in an extraordinarily large model for higher values of K and ℓ, as noted by Mina and Gao [7]. Since the size of the network is primarily driven by the $O(K^2 \times \ell^2)$ scaling, the models reach as high as 8.172×10^9 parameters for $K = 31$ and $\ell = 1031$. While the goal of this paper is to validate a Python implementation of the source material, research into a possible network redesign or parameter pruning would likely prove invaluable. However, given the current architecture, the upper limit to the model size necessitates the use of model parallelization across at least two GPUs from the DelftBlue supercomputer provided by Delft High Performance Computing Centre [11] for the larger model sizes.

3. Python Implementation

While Mina and Gao successfully implemented the NES algorithm in their research, the specific details of their code were not shared in the publication, leaving a gap in the practical reproducibility of their work. To bridge this gap, I developed a Python-based implementation of their NES framework, guided by their theoretical descriptions and mathematical models. My implementation involved translating key components, such as the proposal distribution, sampling process, and gradient estimation, into a functional codebase while ensuring fidelity to their outlined approach. This section provides a breakdown of the implementation through the key functions developed to achieve an operational NES-based PRN code optimizer. My implementation, along with all figures and data generated, is publically available on GitHub*.

- **Proposal Distribution Generation, Gaussian Sampling and Chip Assignment**: Listing 1 is the main function used in extracting samples from the proposal distribution, p_θ, by passing an input of random Gaussian noise (line 7) to the model. The vector of means for each Gaussian distribution assigned

*`https://github.com/amoec/prn-gen`

to each chip in the codes is the model output from the Gaussian noise distribution. The proposal distribution can then be defined as a vector of normal distributions using the aforementioned vector of means along with the constant standard deviation of 0.1 (line 10). Notably, it is at this stage already that part of the gradient estimation described in Equation 6 must be determined. $\nabla_\theta \log(p_\theta(x^{(i)}))$ is calculated in line 13 to be later used in the backpropagation step. The $\mathcal{N}$ samples of K codes, each of length ℓ are then extracted and discretized in line 16 to be only zeros or ones.

```
1  def get_samples(model, K, l, N, sigma=0.1, device='cuda'):
2      with torch.no_grad():
3          # Generate stochastic input noise
4          x = torch.randn(N, K * l, device=device)
5  
6          # Get model predictions
7          mean_vec = model(x)
8  
9          # Generate proposals
10         proposal = mean_vec + sigma * torch.randn_like(mean_vec)
11 
12         # Calculate gradients
13         p_theta = (proposal - mean_vec) / (sigma ** 2)
14 
15         # Generate samples with BPSK modulation
16         samples = torch.sign(proposal.view(N, K, l))
17     return samples, p_theta
18 
```

Listing 1 Function to generate samples and gradients for NES optimization

- **Objective Function Evaluation**: Listing 2 calculates both the cross-correlation and non-central mean autocorrelation, before determining the objective function in line 24 as described by Equation 5. This function proved to be the main bottleneck in the model training, especially for larger code families and code lengths given the $O(\mathcal{N} \cdot K^2 \cdot \ell \log \ell)$ time complexity. In all other areas, 'autocasting' is used for mixed-precision training in order to reduce GPU memory utilization. Equation 3 and Equation 4 are represented in parallel in lines 7-19, while Equation 5 can easily be recognized as line 24 of Listing 2.

```
1  def objective_function(samples):
2      N, K, l = samples.shape
3      samples = samples.float() # Ensure float32 for FFT
4  
5      with autocast(device_type=samples.device.type, enabled=False):
6          # Compute the FFT of all samples
7          fft_samples = fft(samples, dim=2)
8  
9          # Compute the correlation tensor
10         corr_matrix = torch.einsum('nkl,nml->nkml', fft_samples,
   torch.conj(fft_samples))
11         corr_matrix = ifft(corr_matrix, dim=3).real / l
12 
13     corr_matrix[:, torch.arange(K), torch.arange(K), 0] -= 1  # Remove
   central peak
14 
15     # Compute the objective function
16     corr_power = corr_matrix ** 2
17     corr_sum = torch.sum(corr_power, dim=3)
18     autocorr = torch.sum(torch.diagonal(corr_sum, dim1=1, dim2=2), dim=1)
19     crosscorr = torch.sum(corr_sum.view(N, -1), dim=1) - autocorr
20     f_ac = autocorr / (K * l)
21     # Number of pairs of codes
22     K_p = K * (K - 1)
23     f_cc = crosscorr / (K_p * l)
```

```
        obj = torch.max(f_ac, f_cc)
        return obj

```

Listing 2 Function to obtain the objective function value

- **Optimization Step**: Listing 3 works to both calculate the gradient estimate given in Equation 6 and backpropagate this estimate through the GNN. Line 5 corresponds to the baseline calculation described by Equation 7. Note that the maximum of the objective function values is recorded as the training loss, and is referred to as such subsequently.

```
1      def nes_gradient_step(model, optimizer, obj_values, log_prob, scaler):
2          N = obj_values.shape[0]
3
4          # Compute baseline value
5          b = torch.mean(obj_values)
6
7          # Compute the gradient estimate
8          weights = (obj_values - b).view(N, 1)
9          grad_estimate = torch.mean(weights * log_prob, dim=0)
10
11         # Update model parameters
12         optimizer.zero_grad()
13         x_dummy = torch.randn(1, grad_estimate.numel(),
       device=grad_estimate.device)
14
15         with autocast(device_type=grad_estimate.device.type):
16             y_dummy = model(x_dummy)
17
18         scaler.scale(y_dummy).backward(grad_estimate.view(1, -1))
19         scaler.step(optimizer)
20         scaler.update()
21
22         # Compute the maximum loss (for logging)
23         loss = torch.max(obj_values).item()
24         return loss
25
```

Listing 3 Function to obtain the gradient estimate and perform the backpropagation

B. Performance Benchmarking

In order to validate the equal implementation of the NES algorithm between this paper and that of Mina and Gao [7], the same performance benchmarks must be used. The PRN code generator models obtained via their training will therefore be compared to those obtained by Mina and Gao so as to validate the Python implementation of their algorithm. By digitizing the normalized correlation performance plots provided in their paper [7], a direct comparison can be drawn between the worst sampled performance from the trained models.

III. Results

This section contains the most pertinent results of both the model training and subsequent benchmarking against those of Mina and Gao [7], all other data and plots are available on GitHub[†]. Starting with a demonstration of the model convergence behavior across various K and ℓ configurations, and ending with a comparison of the models' performance versus that reported by Mina and Gao [7], this section demonstrates the Python implementation's ability to produce similar if not congruent results.

[†]`https://github.com/amoec/prn-gen`

A. Training Convergence

The training loss, that is, the result of the objective function defined by Equation 5, follows an almost identical path towards a stable, minimum value for all combinations of K and ℓ. By the 750th epoch, the model loss 'turns a corner' so to speak where additional epochs do little to improve its performance. For smaller models, these additional epochs serve to effectively 'dampen' the stochastic variation and expose the model to additional random samples.

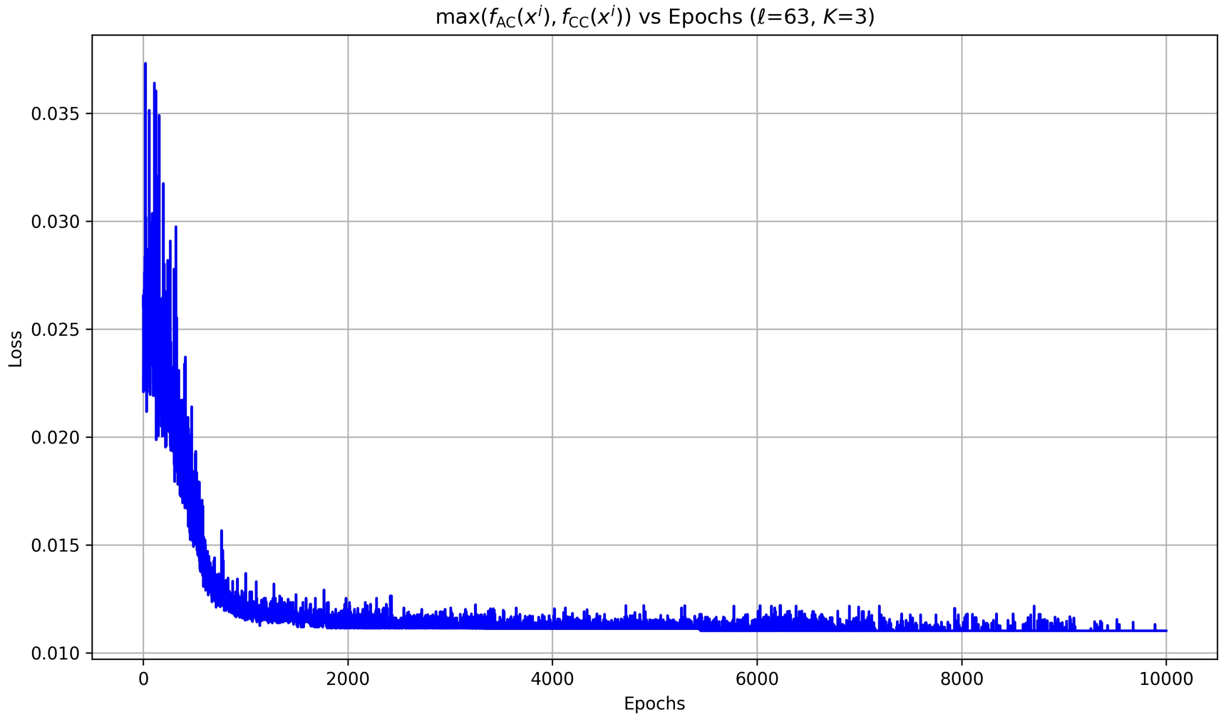


Fig. 1 63 chips in a code, 3 codes in a family

Since overfitting is not a concern here due to the random nature of the input vector and thus lack of training data, a large number of epochs can be used without fear of the model's inability to 'generalize' given its inapplicability in this context. While smaller code lengths tend to exhibit slightly more stochastic variation as the cross-correlation and non-central mean autocorrelation are both pushed down to a minimum, larger code lengths tend to eliminate much of the variation early on. This behavior can be seen on either extreme with Figure 1 above and Figure 2 below.

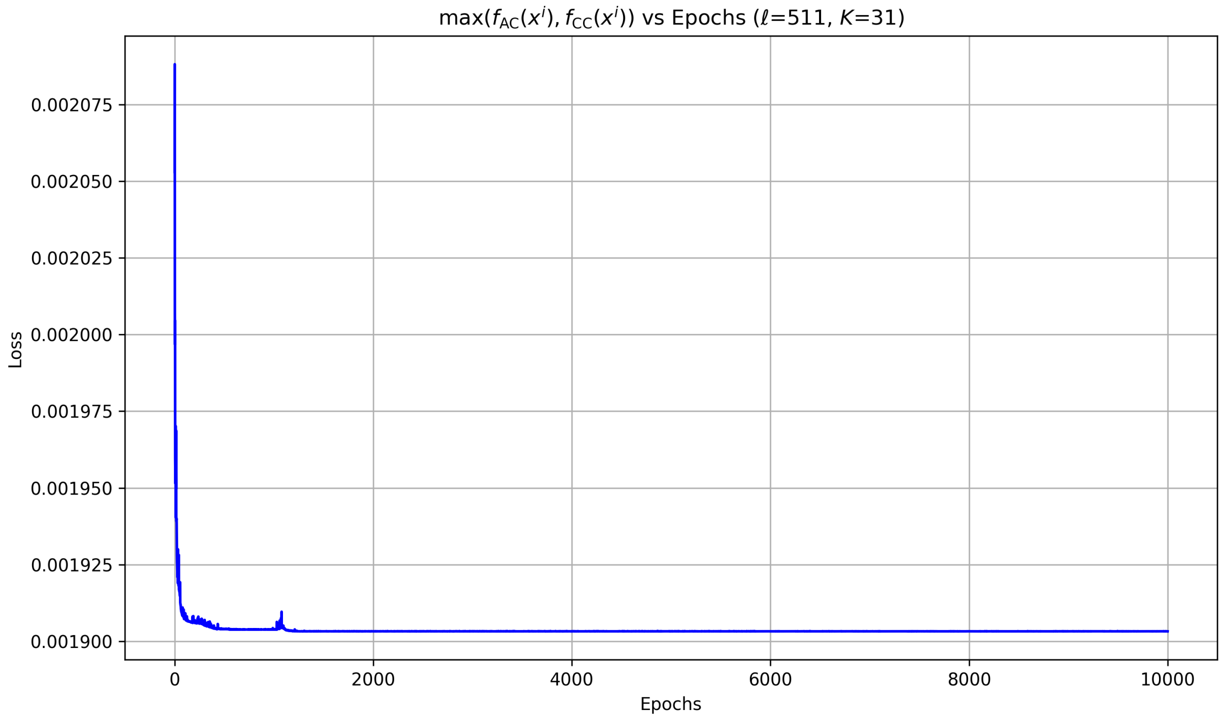


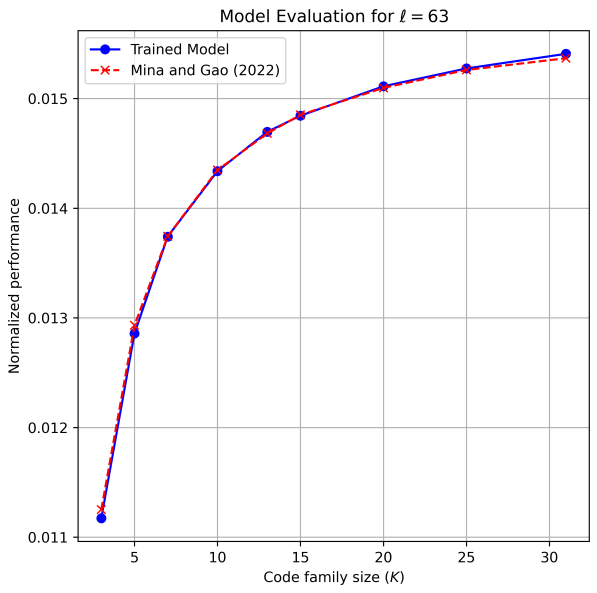
Fig. 2 511 chips in a code, 31 codes in a family

B. Performance Benchmarks

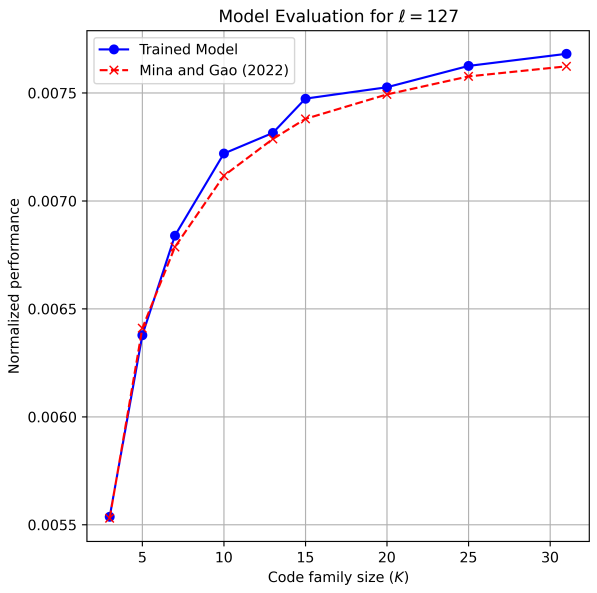
The models' performance versus those obtained by Mina and Gao [7] is provided in Figure 3 below. From the results obtained, a trend appears wherein smaller code lengths, such as $\ell = 63$ shown in Figure 3a matches Mina and Gao's performance more closely. The normalized performance for $\ell = 127$ once again more closely matches that of Mina and Gao's than for $\ell = 511$ (Figure 3b and Figure 3c respectively). Indeed, the maximum percentage difference between the trained models and established literature is 0.7%, 1.44%, and 2.9% for $\ell = 63$, 127 and 511 respectively. Aside from this discrepancy, the results do seem to align with those of Mina and Gao [7].

Interestingly, within the $\ell = 127$ and 511 performance comparisons with those of Mina and Gao, we see the maximum error occurring for medium code family sizes, that is, K between roughly 7 and 15 inclusive. For $K < 7$ and $K > 15$, the results from the trained models once again more closely match those of Mina and Gao [7]. While most of the discussion surrounding this observation will be reserved for Section IV, the trend of increasing discrepancy with increasing model size points to imprecision in floating-point operations. Since 'autocasting' was used in order to take advantage of mixed-precision training to reduce memory utilization, half-precision floating points were more likely to be employed in cases where memory utilization was already high, that is, large model sizes.

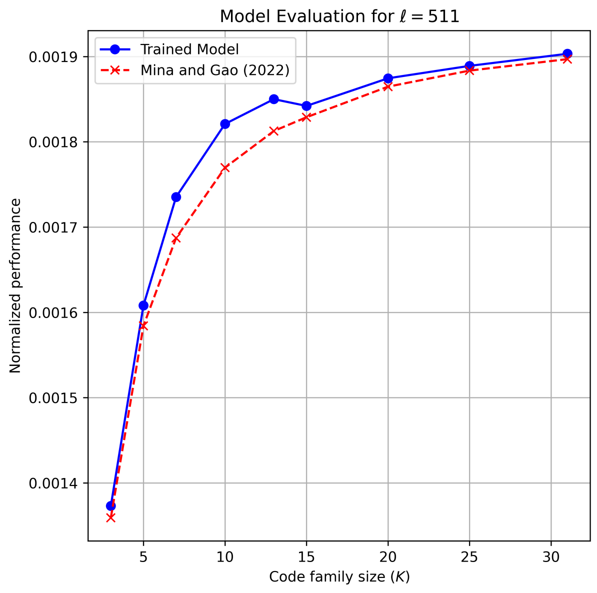
Generally, however, we see a very close match between the trained models and the data taken from Mina and Gao, especially for the smallest code-length training. While further investigation into the sources of the error and possible model retraining would be necessary to properly validate this Python implementation of Mina and Gao's NES algorithm, these results are certainly encouraging.



(a) 63 chips in a code



(b) 127 chips in a code



(c) 511 chips in a code

Fig. 3 Trained model performance versus that of Mina and Gao [7]

IV. Discussion

The training process revealed several interesting aspects of the NES algorithm's implementation as described by Mina and Gao [7]. The large model sizes for higher values of K and ℓ, as described in Section II, necessitates the use of so-called 'autocasting' enabling the use of mixed-precision training. While this was a great tool in reducing the GPU memory utilization associated with the model training, it naturally resulted in far higher round-off errors due to the use half-precision floating point operations.

While the exact cause of the trained models' imprecision with regards to established results is likely not solely due to floating-point precision, the increasing error with increasing model sizes is a clear indicator as to its negative impact on the model performance. Looking more closely at which combinations of K and ℓ lead to the highest discrepancies further supports this theory. For small values of K, the PyTorch autocasting functionality is more likely to default to the use of single-precision floating points due to the far lower model sizes, even for large code lengths. For high code lengths, the objective function values will naturally be far lower than for larger code lengths, meaning that high floating point precision becomes crucial to obtain reasonable results from the various Python function evaluations.

As the value of K continues to increase for the larger code lengths, the value of the objective function is naturally higher, lending itself to reduced round-off error. Indeed, we see that for the higher values of K, the discrepancy between the trained models' performance and that of Mina and Gao's once again becomes negligible. It is also worth mentioning that the remaining difference between obtained and provided results could stem from several independent factors such as the use of a plot digitizer to transcribe Mina and Gao's results to CSV format, or this project's independent selection of the hidden-layer activation functions.

Finally, it is difficult to understate the computational requirements of this network architecture. While offloading the model training to NVIDIA A100 Tensor Core GPUs provided as part of the DelftBlue supercomputer did result in significant improvements to training time, the largest models still required to be split across multiple GPUs [11, 12]. The sheer complexity of the network architecture therefore results in significant time and computational resources in order to obtain satisfactory model performance. Out-of-memory (OOM) errors plagued earlier attempts to run the model training locally thus forcing the switch to DelfBlue's vastly improved computational resources [11].

V. Conclusion

Overall, this research should be counted a success while also opening the door to several potential avenues of further refinement to improve the model performance. This Python implementation of Mina and Gao's NES algorithm [7] was successful in reproducing their recorded results with a maximum percentage difference in performance of just 2.90% for the largest trained model with a code length, ℓ, of 511 chips. By providing the code used to reproduce the NES algorithm in the form of an open-source GitHub repository, the author invites anyone interested in novel approaches to GPS C/A PRN code generation to refine its implementation.

While this project was largely a success, floating-point imprecision remained the likely culprit for slight deviation in the performance of trained models versus that of Mina and Gao's. While mixed-precision training proved invaluable to ensure training convergence without running the risk of experience OOM errors at the execution level, this was at the cost of model performance and future work should thus look into improving the memory utilization of the author's Python implementation, and investigate whether the NES network architecture can be simplified or pruned without negatively affecting the trained model performance.

The use of ML for GNSS PRN code generation remains relatively untested field. As such, work remains to determine whether entirely different ML approaches can achieve similar or better model performance. Convolutional neural networks (CNNs) or even reinforcement learning (RL) approaches such as soft actor-critic (SAC) or proximal-policy optimization (PPO) are untested in this domain and offer exciting avenues of research.

Acknowledgments

The author acknowledges the use of computational resources of the DelftBlue supercomputer, provided by Delft High Performance Computing Centre [11]. The author also acknowledges the incredible work of Mina and Gao on researching the implementation of machine learning techniques for GNSS applications [7, 13]. Finally, the author acknowledges the guidance and support of Professor James L. Garrison of Purdue University throughout this research project.

References

[1] Kalman, R. E., "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, Vol. 82, No. 1, 1960, pp. 35–45. doi:10.1115/1.3662552, URL https://asmedigitalcollection.asme.org/fluidsengineering/article/82/1/35/397706/A-New-Approach-to-Linear-Filtering-and-Prediction.

[2] Nadler, A., and Bar-Itzhack, I. Y., "An Efficient Algorithm for Attitude Determination Using GPS," 1998, pp. 1783–1789. URL http://www.ion.org/publications/abstract.cfm?jp=p&articleID=3116.

[3] McCallum, J. C., “Memory Prices 1957 - 2024,” , 2024. URL https://jcmit.net/memoryprice.htm, Accessed: 2024-09-07.

[4] Mohanty, A., and Gao, G., “A survey of machine learning techniques for improving Global Navigation Satellite Systems,” *EURASIP Journal on Advances in Signal Processing*, Vol. 2024, No. 1, 2024, p. 73. doi:10.1186/s13634-024-01167-7.

[5] Bishop, C. M., *Pattern recognition and machine learning*, Information science and statistics, Springer, New York, 2006.

[6] Siemuri, A., Kuusniemi, H., Elmusrati, M. S., Valisuo, P., and Shamsuzzoha, A., “Machine Learning Utilization in GNSS—Use Cases, Challenges and Future Applications,” *2021 International Conference on Localization and GNSS (ICL-GNSS)*, IEEE, Tampere, Finland, 2021, pp. 1–6. doi:10.1109/ICL-GNSS51451.2021.9452295, URL https://ieeexplore.ieee.org/document/9452295/.

[7] Mina, T. Y., and Gao, G. X., “Designing Low-Correlation GPS Spreading Codes with a Natural Evolution Strategy Machine-Learning Algorithm,” *Journal of the Institute of Navigation*, Vol. 69, No. 1, 2022. doi:10.33012/navi.506.

[8] Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A., “Monte Carlo Gradient Estimation in Machine Learning,” *Journal of Machine Learning Research*, , No. 21, 2020, pp. 1–62.

[9] Glorot, X., Bordes, A., and Bengio, Y., “Deep Sparse Rectifier Neural Networks,” *International Conference on Artificial Intelligence and Statistics*, 2011. URL https://api.semanticscholar.org/CorpusID:2239473.

[10] Kingma, D. P., and Ba, J., “Adam: A Method for Stochastic Optimization,” , Jan. 2017. doi:10.48550/arXiv.1412.6980, URL http://arxiv.org/abs/1412.6980, Accessed: 2024-12-03, arXiv:1412.6980 [cs].

[11] Delft High Performance Computing Centre (DHPC), “DelftBlue Supercomputer (Phase 2),” https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2 , 2024.

[12] “NVIDIA A100 GPUs Power the Modern Data Center,” , 2024. URL https://www.nvidia.com/en-us/data-center/a100/, Accessed: 2024-12-03.

[13] Mina, T. Y., and Gao, G. X., “Devising High-Performing Random Spreading Code Sequences Using a Multi-Objective Genetic Algorithm,” Miami, Florida, 2019, pp. 1076–1089. doi:10.33012/2019.17044.