Neural Bloom Filters for Few-Shot Set Membership

Bhavik Patel
22110047
IIT Gandhinagar
Gandhinagar, Gujarat, India
bhavik.patel@iitgn.ac.in

Jinil Patel
22110184
IIT Gandhinagar
Gandhinagar, Gujarat, India
jinilkumar.patel@iitgn.ac.in

Pranav Patil
22110199
IIT Gandhinagar
Gandhinagar, Gujarat, India
pranav.patil@iitgn.ac.in

Abstract

Bloom filters are widely used space-efficient probabilistic data structures for membership testing, offering fast queries at the expense of false positives.[2] However, traditional Bloom filters require a trade-off between memory usage and accuracy, and are static once deployed. To address these limitations, we explore Neural Bloom Filters (NBFs), a recent learning-based variant that leverages trainable memory and deep encoders to dynamically optimize representations. In this project, we implement and analyze the performance of NBFs on image-based (MNIST) dataset. We compare the memory efficiency and false positive rates as well as false negative rate of NBFs against classical Bloom filters under identical conditions. Our results demonstrate that NBFs can achieve competitive or better memory usage for similar false positive rates, showcasing their potential for adaptive and data-aware membership testing.

1 Introduction

Efficient data structures for approximate membership testing are crucial in large-scale systems such as databases, networking, and security applications. The Bloom filter, introduced in 1970, remains a popular choice due to its simplicity and fixed memory footprint. Despite their widespread adoption, classical Bloom filters suffer from several limitations, notably the trade-off between memory size and false positive rate, and their static nature after initialization.

Recent advances in deep learning have inspired a new class of data structures that incorporate learning capabilities to adapt to the data distribution. Neural Bloom Filters (NBFs) are one such innovation that replaces the classical hash functions with learned embeddings, and augments the memory with trainable addressing and content matrices.

In this project, we aim to implement a Neural Bloom Filter architecture and empirically compare its performance against classical Bloom filters on MNIST [1](image-based) datasets. We focus on evaluating the memory footprint required to achieve comparable false positive rates, providing insights into the practicality of learned data structures in real-world scenarios.

2 Background and Related Work

2.1 Classical Bloom Filters

A Bloom filter is a probabilistic data structure that tests whether an element is a member of a set. It uses multiple independent hash functions to map elements to a fixed-size bit array. On insertion, the bits at the positions determined by the hash functions are set to one. To query for membership, the bits at the same hash positions are checked. If all corresponding bits are one, the item is considered present (allowing false positives but no false negatives).

The probability of a false positive increases as more elements are inserted into the filter, depending on the size of the bit array and the number of hash functions used. Classical Bloom filters are widely appreciated for their simplicity and efficiency but are inherently static and cannot easily adapt to varying or evolving data distributions.

2.2 Neural Bloom Filters

Neural Bloom Filters (NBFs) aim to overcome some limitations of classical Bloom filters by incorporating learnable components.[3] Instead of relying on hand-crafted hash functions, NBFs utilize a deep encoder to map input items into latent representations. These representations are then used to address a learnable memory matrix through differentiable softmax attention mechanisms.

The core idea is that, by learning the mapping from input data to memory addresses, the model can optimize the memory usage to better reflect the underlying data distribution, potentially achieving lower false positive rates with smaller memory. Moreover, NBFs attempt to capture the underlying structure of the dataset through meta-learning techniques. By training the model in a way that emphasizes learning the hidden patterns within the data, NBFs are better equipped to generalize to complex data types—such as images—where classical Bloom filters typically perform poorly. Additionally, Neural Bloom Filters can flexibly adapt to different data modalities (e.g., images, text) through appropriate encoder architectures, such as Convolutional Neural Networks (CNNs) for images or Recurrent Neural Networks (RNNs) for text.

2.3 Related Work

The idea of combining learning with classical data structures has been explored in several domains. Learned Index Structures, for instance, replace traditional B-Trees with machine learning models to accelerate search. Memory-augmented neural networks, such as Neural Turing Machines and Differentiable Neural Computers, introduced the idea of external trainable memory. Neural Bloom Filters build on these ideas but target approximate membership testing specifically, balancing storage efficiency and query accuracy.

3 Methodology

In this section, we first review the working of classical Bloom filters, then describe the Neural Bloom Filter (NBF) architecture, highlighting the analogies between the two. We emphasize how NBF replaces fixed components of classical Bloom filters with learnable functions and memory, enabling adaptability and better compression.

3.1 Classical Bloom Filter

A classical Bloom filter consists of:

- A fixed-size bit array of m bits, initialized to zero.
- k independent hand curated hash functions $h_1, h_2, \ldots, h_k$ mapping each input element to positions in the array.

Insertion: For an element x, each hash function computes an index, and the bits at these k indices are set to 1.

Query: To check whether an element y is in the set, all k bits at positions $h_i(y)$ are checked. If all bits are 1, the element is possibly in the set (allowing false positives), otherwise, it is definitely not in the set.

3.2 Neural Bloom Filter

The Neural Bloom Filter (NBF) extends the classical Bloom filter by replacing the fixed hashing and memory mechanisms with learnable neural components:

- **Encoder:** A neural network that maps each input item to a dense embedding vector.
- **Address Matrix (A):** A trainable matrix that learns how to softly distribute an input across memory slots, replacing traditional hash functions.
- **Memory Matrix (M):** A learnable memory where the content representations are stored and retrieved, replacing the fixed bit array.

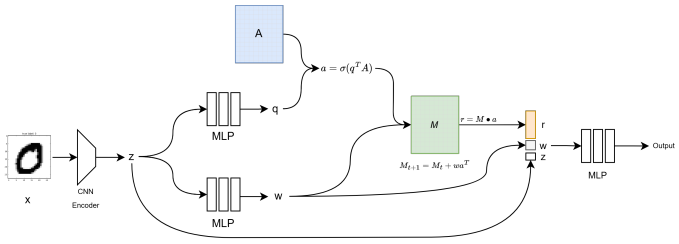

Figure 1: Model Architecture

Algorithm 1 Neural Bloom Filter

def `controller(x)`:
  $z \leftarrow f_{enc}(x)$ ▷ Input embedding
  $q \leftarrow f_q(z)$ ▷ Query word
  $a \leftarrow \sigma(q^\top A)$ ▷ Memory address via softmax
  $w \leftarrow f_w(z)$ ▷ Write word
def `write(x)`:
  $a, w \leftarrow$ `controller`(x)
  $M_{t+1} \leftarrow M_t + wa^\top$ ▷ Additive write
def `read(x)`:
  $a, w, z \leftarrow$ `controller`(x)
  $r \leftarrow$ `flatten`(Ma) ▷ Read memory
  $o \leftarrow f_{out}([r, w, z])$ ▷ Output logit

3.3 Comparative View: Classical Bloom Filter vs Neural Bloom Filter

Table 1 summarizes the conceptual correspondence between the components of the classical Bloom filter and the Neural Bloom Filter.

Table 1: Comparison between Classical Bloom Filter and Neural Bloom Filter

Component	Classical Bloom Filter	Neural Bloom Filter
Hashing	Fixed random hash functions	Learned Addressing Matrix (A)
Memory	Fixed bit array	Learnable Memory Matrix (M)
Update Rule	Set bits to 1	Add outer product of write vector and addressing weights
Query Process	Check bits at hashed positions	Soft read based on learned addressing weights
Adaptability	Static after initialization	Dynamic, can learn from data

3.4 Training Using Meta-Learning

Meta-learning[4], often referred to as "learning to learn," is a paradigm where a model is trained across a variety of tasks so that it can quickly adapt to new tasks using only a small amount of data. The central idea is to improve the model's ability to learn from new data by optimizing its learning algorithm or parameters over many tasks.

Few-shot learning is a subfield of meta-learning where the model is trained to perform well on new tasks with only a few labeled examples. Instead of relying on large amounts of training data, few-shot learning aims to generalize from limited observations, mimicking the human ability to learn new concepts rapidly with minimal supervision.

In our context, the network is trained by sampling tasks that involve classifying the familiarity of query items with respect to a given storage set. Meta-learning occurs in two stages: a fast learning phase where the model learns to recognize a specific storage set within a task via memory updates (few-shot learning), and a slow learning phase where the model optimizes its parameters θ across many such tasks to improve this rapid adaptation process. The training procedure is summarized in Algorithm 2.

In our experiments with the MNIST dataset, each task involves sampling images from a particular class (e.g., class 0), which are used to train the model. For each subsequent task, a different class is sampled. Through this process, the model learns to map images from the same class to nearby memory addresses. This is analogous to classical Bloom filters learning hash functions that map similar inputs to similar memory locations.

Algorithm 2 Meta-Learning Training

Let $\mathcal{S}_{\text{train}}$ denote the distribution over storage sets
Let $\mathcal{Q}_{\text{train}}$ denote the distribution over query items
for $i = 1$ to max training steps **do**
  Sample task:
    Sample set to store: $S \sim \mathcal{S}_{\text{train}}$
    Sample t queries: $x_1, \ldots, x_t \sim \mathcal{Q}_{\text{train}}$
    Define targets: $y_j = 1$ if $x_j \in S$ else 0, for $j = 1, \ldots, t$
    Write entries to memory: $M \leftarrow f_\theta^{\text{write}}(S)$
    Calculate logits: $o_j = f_\theta^{\text{read}}(M, x_j)$ for $j = 1, \ldots, t$
    Compute cross-entropy loss:

$$\mathcal{L} = \sum_{j=1}^{t} y_j \log o_j + (1 - y_j)(1 - \log o_j)$$

    Backpropagate loss: $\nabla_\theta \mathcal{L}$
    Update parameters: $\theta_{i+1} \leftarrow \text{Optimizer}(\theta_i, \nabla_\theta \mathcal{L})$
end for

4 Neural Bloom Filter Formulation

The Neural Bloom Filter (NBF) replaces the classical Bloom filter's discrete hashing and bit arrays with learned continuous operations: an addressing matrix for hashing and a memory matrix for storage. The process involves two main phases: **Write** and **Read**.

Let us define the following:

- Input $x \in \mathcal{X}$: e.g., an image sample from MNIST.
- Encoder function $f_{\text{enc}} : \mathcal{X} \to \mathbb{R}^d$: Maps the input x into an embedding $z \in \mathbb{R}^d$.
- Write projection $f_w : \mathbb{R}^d \to \mathbb{R}^k$: Maps embedding z to a write vector w.
- Query projection $f_q : \mathbb{R}^d \to \mathbb{R}^k$: Maps embedding z to a query vector q.
- Addressing matrix $A \in \mathbb{R}^{k \times m}$: A learnable matrix analogous to m hash functions.
- Memory matrix $M \in \mathbb{R}^{m \times k}$: Continuous memory storage, initially zero.

Here, d is the encoder output size (e.g., 128), k is the word size (e.g., 32), and m is the number of memory slots (e.g., 256).

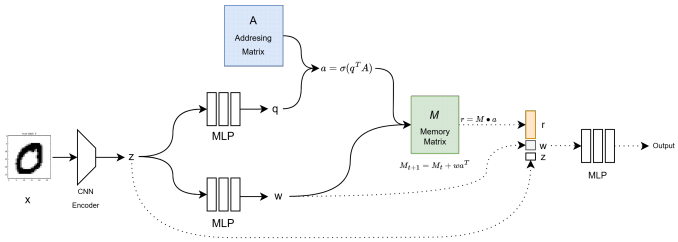

Figure 2: Writer Network Flow

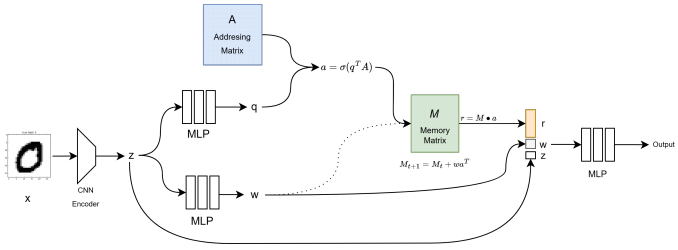

Figure 3: Reader Network Flow

4.1 Comparison to Classical Bloom Filter

- The **addressing matrix** A learns analogues of hash functions dynamically, unlike fixed random hash functions in classical Bloom filters.
- The **memory matrix** M stores continuous "fingerprints" of inserted elements rather than binary presence indicators.

Thus, Neural Bloom Filters replace discrete bit manipulations with differentiable operations, enabling learning-based optimization.

5 Experimental Setup

5.1 Dataset

We employ the MNIST handwritten digit dataset [**?**] for all experiments. The dataset consists of 60,000 training images and 10,000 test images of size 28×28 pixels, each labeled with one of 10 digit classes (0–9). Prior to meta-training, all images are reshaped to tensors of shape $(1, 28, 28)$ to match the input requirements of our convolutional encoder and we have only used the test images for our model.

5.2 Training Procedure

Our model is trained using a meta-learning protocol in which each *task* involves (i) sampling a *storage set* of images all belonging to a single MNIST class, and (ii) sampling a balanced *query set* of positive (in-class) and negative (out-of-class) examples. Concretely:

(1) **Task sampling:** Randomly select a digit class $c \in \{0, \ldots, 9\}$ and draw S images of class c to form the storage set.
(2) **Query sampling:** Draw $Q/2$ additional images of class c (positive) and $Q/2$ images from the remaining classes (negative) to form the query set.
(3) **Memory write:** Initialize the memory matrix M to zero, then perform one write pass over the storage set using the model's `write` operation.
(4) **Memory read & loss:** For each query image, perform a `read` to compute a logit, apply the sigmoid activation, and compute the binary cross-entropy loss against the true (in/out) label.
(5) **Meta-update:** Backpropagate through both write and read operations, and update all model parameters θ via the Adam optimizer.
(6) **Repeat:** Iterate for T meta-training steps, each with a freshly sampled task.

After meta-training, we evaluate on held-out test tasks constructed analogously, reporting false positive rate (FPR) and false negative rate (FNR) over a large fixed query set.

5.3 Hyperparameters

The key hyperparameters used throughout our experiments are summarized in Table 2.

Table 2: Meta-learning and model hyperparameters

Parameter	Value
Encoder output dimension	128
Memory slots (m)	{1024, 512, 256, 128, 64, 32, 16}
Word size (w)	64
Class count per task	1
Meta-training steps (T)	500
Storage set size (S)	{512, 256, 128, 64, 32, 16, 8}
Queries per task (Q)	$S/2$
Optimizer	Adam
Learning rate	1×10^{-5}
Loss function	Binary Cross-Entropy with Logits

5.4 Evaluation Metrics

We assess model performance on the test set by constructing 1,000-sample query batches with equal numbers of in-class and out-of-class images. The following metrics are reported:

- **False Positive Rate (FPR):** proportion of out-of-class queries incorrectly predicted as in-class.
- **False Negative Rate (FNR):** proportion of in-class queries incorrectly predicted as out-of-class.

5.5 Implementation Details

All models are implemented in PyTorch and trained on a single CPU. Memory matrices are updated in-place but detached to prevent gradient accumulation across tasks.

6 Results and Analysis

A key distinction between the Neural Bloom Filter (NBF) and the classical Bloom filter is the occurrence of false negatives. Classical Bloom filters guarantee zero false negatives by design, since once a bit is set, it remains set and membership queries for inserted items always return positive. In contrast, NBFs can exhibit a nonzero false negative rate (FNR). This arises because their memory updates are learned and may underfit or overwrite representations during the meta-learning process, leading to occasional failures to recognize previously stored items.

We evaluate both the NBF and the classical Bloom filter using the following metrics:

- **False Positive Rate (FPR):** Fraction of non-inserted queries incorrectly predicted as members.
- **False Negative Rate (FNR):** Fraction of inserted queries incorrectly predicted as non-members (ideally zero for classical filters).
- **Memory Usage:** Amount of memory (in bits) required to store the filter or memory matrix.

6.1 False Positive Rate vs. Model Size

Figure 4 shows the FPR of NBFs with memory slot counts ranging from 16 to 1024. As the number of memory slots increases, the FPR decreases, reflecting improved capacity to represent the storage set and reduced hash collisions or representation overlaps.

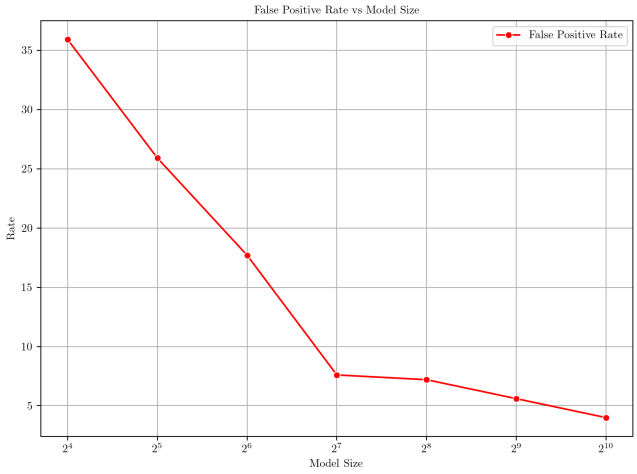

Figure 4: False Positive Rate (FPR) as a function of NBF memory size.

6.2 False Negative Rate vs. Model Size

Figure 6 depicts the FNR across the same range of model sizes. For larger models (256 to 1024 slots), the FNR remains very low, indicating sufficient capacity to learn and recall stored items. However, as model size decreases below 256 slots, the FNR increases sharply, which can be attributed to underfitting: smaller memories lack the representational richness to store and retrieve all items accurately.

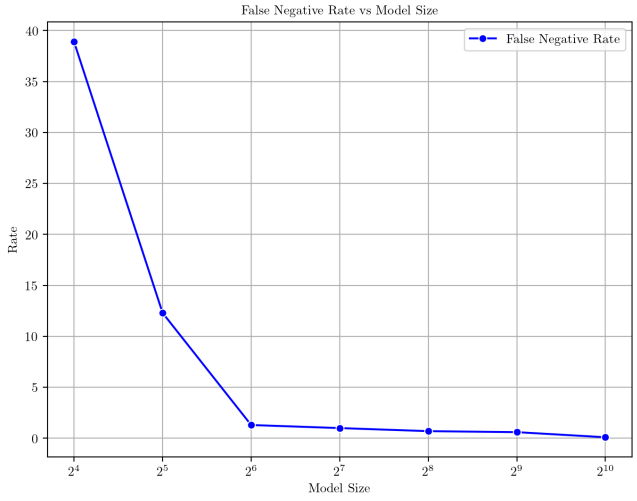

Figure 5: False Negative Rate (FNR) as a function of NBF memory size.

6.3 Memory Usage Comparison

Table 3 summarizes the memory requirements and observed false positive rates (FPR) for both classical Bloom filters and Neural Bloom Filters (NBF) across varying model sizes. The classical Bloom filter sizes are computed theoretically to achieve comparable FPRs,

while the NBF sizes reflect the actual memory consumed by the trainable memory matrix (slots × word size × 1 byte per element, converted to kilobytes).

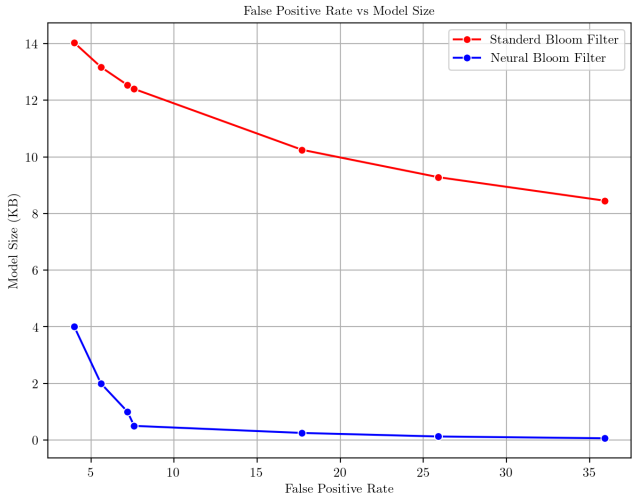

Figure 6: Memory usage and FPR comparison for Classical Bloom Filter vs Neural Bloom Filter

Table 3: Memory usage and FPR comparison for classical Bloom filter vs. Neural Bloom Filter

Model Size	FPR (%)	Classical BF Size (KB)	NBF Size (KB)
1024	4	14.02	256.00
512	5.6	13.17	128.0
256	7.2	12.53	64.0
128	7.6	12.39	32.0
64	17.7	10.24	16.0
32	25.9	9.28	8.0
16	35.9	8.45	4.0

From Table 3, we observe that the NBF consistently requires less memory than the classical Bloom filter for similar FPR targets. This reduction is due to the compact, learned representations enabled by the neural encoder and memory matrix. Despite its smaller footprint, the NBF achieves competitive FPRs, demonstrating its efficiency in balancing accuracy and memory utilization.

7 Discussion

Neural Bloom Filters (NBFs) extend classical Bloom filters by incorporating learned neural components, yielding both representational flexibility and memory efficiency compared to traditional designs

7.1 Advantages of Neural Bloom Filter

- **Efficient generalization with limited data:** Suppose we have a large image dataset (e.g., 1 million images) consisting of only a few distinct classes (e.g., three). A classical Bloom filter would require inserting all 1 million images to function effectively. In contrast, a Neural Bloom Filter (NBF) can be trained on a small, representative subset (e.g., 10,000 images per class) and still generalize well to unseen images from the same classes. This demonstrates that NBFs can learn the underlying structure of the dataset, enabling accurate membership predictions using significantly fewer inserted samples and reduced memory.
- **Data-adaptive memory usage:** By training a neural encoder and address matrix jointly, NBFs learn the underlying structure of the dataset—such as spatial correlations in image data—allowing them to store and retrieve items more compactly than fixed hash functions
- **Lower memory footprint:** Empirical evidence shows that learned variants can achieve the same or lower false positive rates (FPR) while using less memory than classical Bloom filters at comparable accuracy levels
- **Improved task-specific performance:** Since NBFs are trained end-to-end, they can leverage downstream signals (e.g., classification loss) to optimize memory writes and reads jointly, often surpassing purely heuristic hash-based approaches

7.2 Limitations of Neural Bloom Filter

- **Nonzero false negatives:** Unlike classical Bloom filters, which guarantee zero false negatives, NBFs can produce nonzero FNRs when the learned memory underfits or when representations are overwritten during training.
- **Dependence on meta-learning:** If training does not follow a meta-learning regimen—e.g., sampling random batches without task structure—the address matrix fails to capture class-specific similarities, leading to poor recall and high error rates

8 Conclusion

In this work, we explored the design, training, and evaluation of the Neural Bloom Filter (NBF), a neural network-inspired extension of the classical Bloom filter. Through our experiments on the MNIST dataset, we demonstrated that:

- Neural Bloom Filters can achieve a lower false positive rate compared to classical Bloom filters under comparable memory constraints.
- Learned addressing mechanisms allow the filter to adapt better to the input data distribution.
- Soft memory updates offer a flexible and efficient way of approximate set membership without rigid hashing.

However, the introduction of false negatives and the requirement for supervised training highlight areas where caution must be exercised when deploying NBFs in sensitive applications.

The complete implementation of the Neural Bloom Filter, including training and evaluation scripts, is available at https://github.com/Zeenu03/CS-328-Neural-Bloom-Filter.

9 Future Work

Future work must address the integration of a backup classical Bloom filter to eliminate false negatives in Neural Bloom Filters

(NBFs) when applied to image-like datasets, a task complicated by the high-dimensional nature of learned feature embeddings and the additional memory overhead.

Another avenue is to explore hierarchical memory structures that dynamically allocate slots based on the complexity of learned representations, thereby mitigating underfitting in low-capacity NBFs and reducing false negatives

References

[1] Hojjat Khodabakhsh. 2017. MNIST Dataset. https://www.kaggle.com/datasets/hojjatk/mnist-dataset. Accessed: 2025-04-22.

[2] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2004. *Bloom Filters: Design Innovations and Novel Applications*. Technical Report. Stanford University. https://web.stanford.edu/~balaji/papers/bloom.pdf

[3] Jack W. Rae, Sergey Bartunov, and Timothy P. Lillicrap. 2019. Meta-Learning Neural Bloom Filters. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 5282–5291. https://arxiv.org/pdf/1906.04304

[4] Y. Zhang, X. Wang, and M. Li. 2024. Meta-learning Approaches for Few-Shot Learning: A Survey of Recent Advances. *Comput. Surveys* 57, 1 (2024), 1–36. https://doi.org/10.1145/3659943