Neural Bloom Filter with Few Shot Learning

CS328 - Data Science

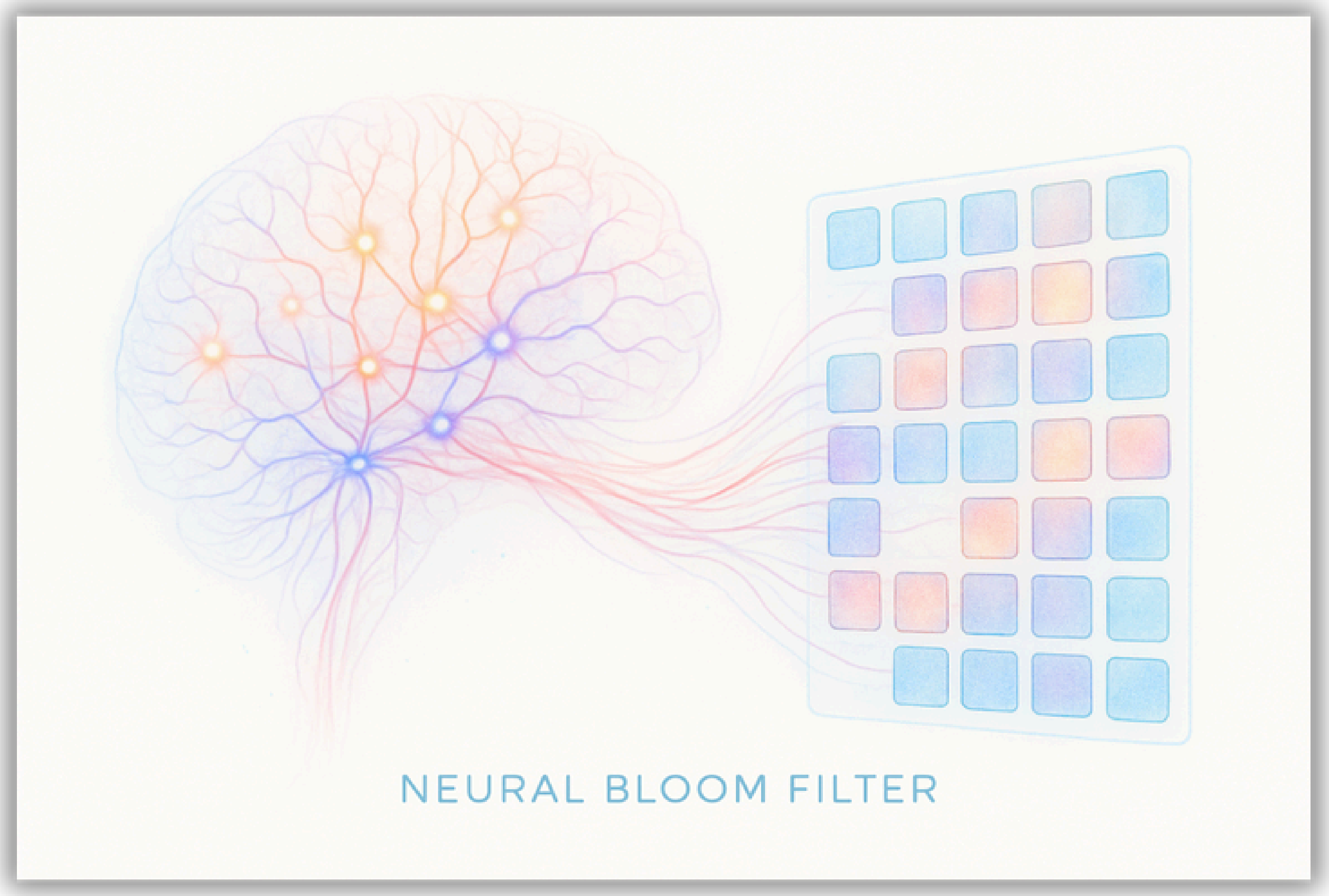
TEAM - 27

Bhavik Patel (22110047)

Jinil Patel (22110184)

Pranav Patil (22110199)

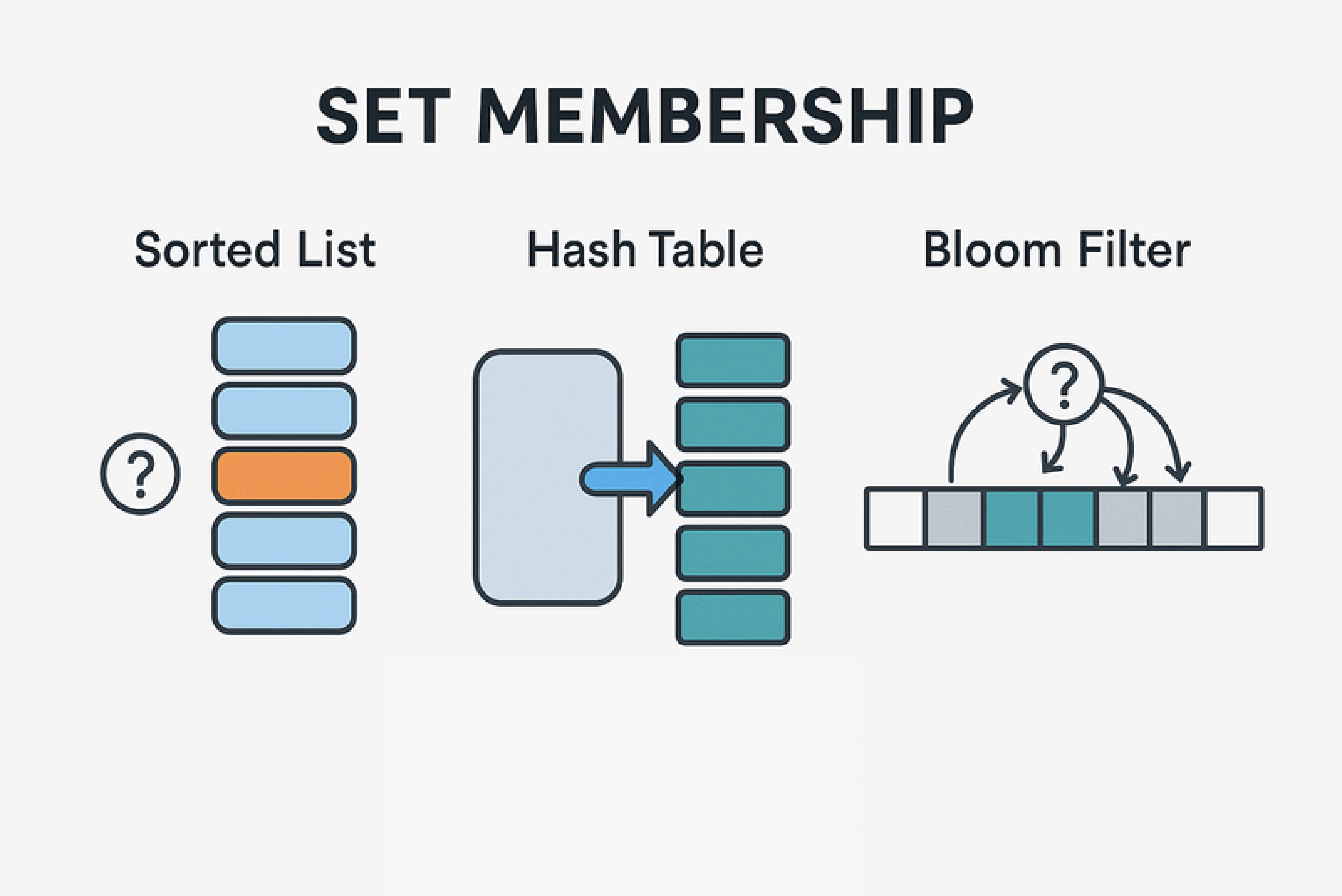
Indian Institue of Technology Gandhinagar
Palaj, Gujarat - 382355



Source Code : https://github.com/Zeenu03/CS-328-Neural-Bloom-Filter

Set Membership Query

- **Task:** "Is element x in set S?"
- Known Methods
  - Sorted List + Binary Search
  - Hash Table
  - Bloom Filter



Background and Related Work

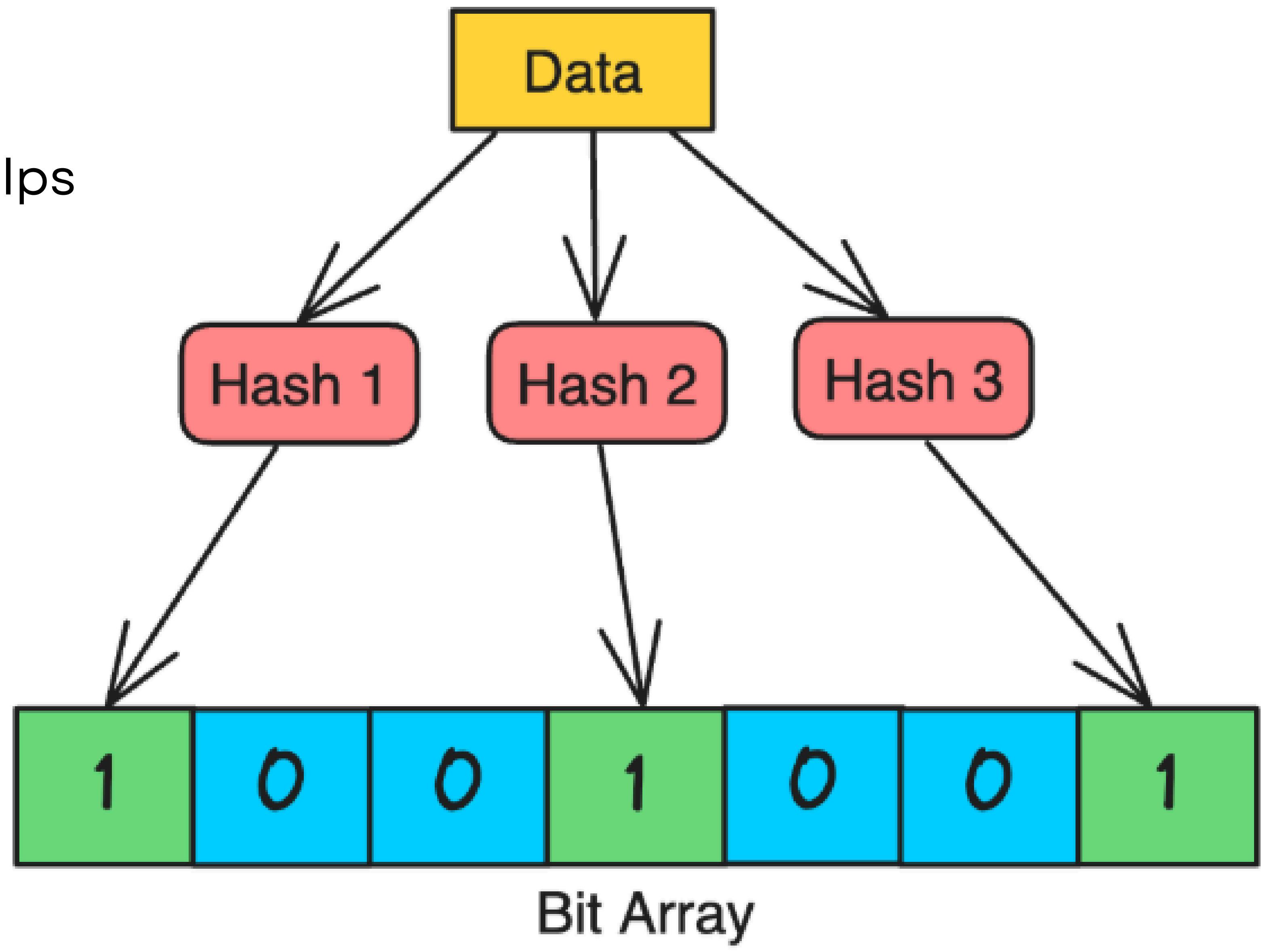
- **Classical Bloom Filters:**
  - Probabilistic data structures with no false negatives.
  - Trade-off between false positive rate and memory size.
  -
- **Memory-Augmented Neural Networks (MANNs)**:
  - Networks like DNC, Memory Networks that enhance traditional RNNs with external memory.
  -
- **Meta-Learning:**
  - Learning to learn across tasks rather than a single dataset.

- NBFs integrate ideas from Bloom Filters, MANNs, and Meta-Learning.

Classical Bloom Filter

- A Bloom Filter is a space-efficient data structure that helps us check if an element might be in a set, with:

  ✅ No false negatives
  ❌ Possible false positives



Why Neural Bloom Filter?

- Classical Bloom Filters are lightweight and fast but **non-adaptive**.
- **Fixed hash functions** in traditional Bloom Filters **cannot generalize**.
- Need for flexible, **trainable structures** that **adapt to datasets**.
- Neural Bloom Filters (NBFs) combine neural networks with Bloom Filter principles for better memory compression and task-specific learning.

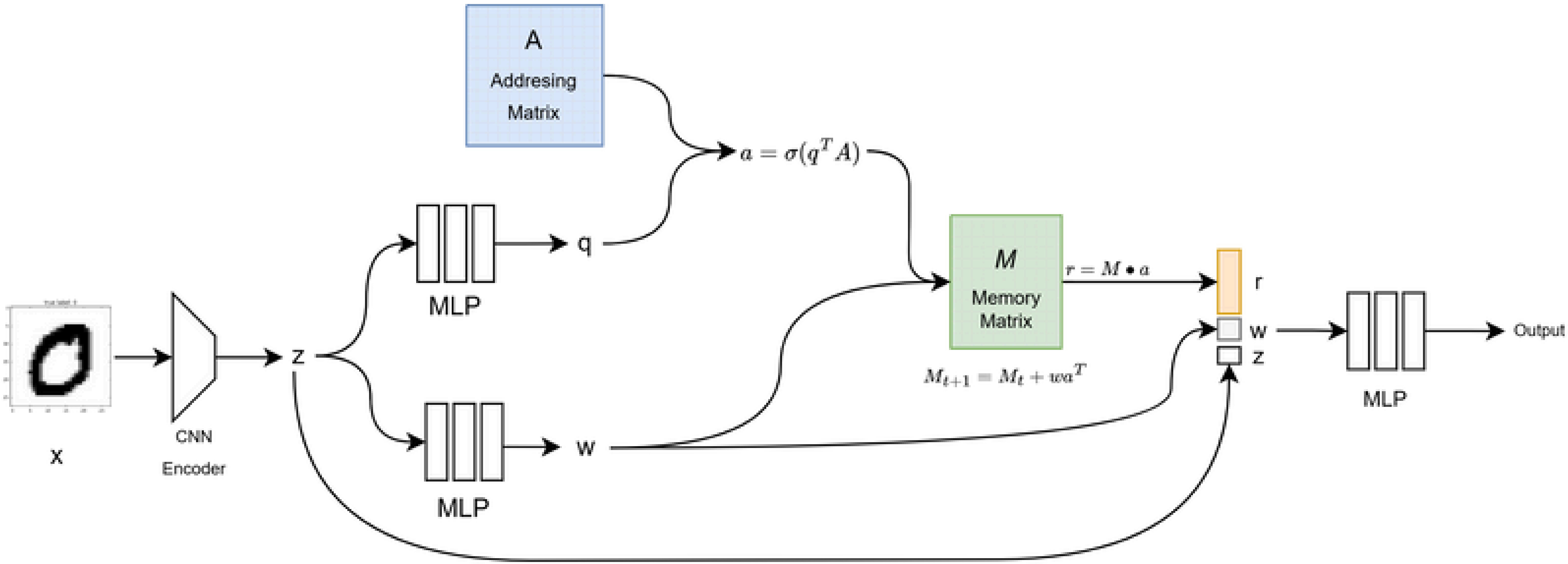
Neural Bloom Filter Formulation

Main Idea:
Replace hand-crafted hashing with learned addressing and distributed memory updates.

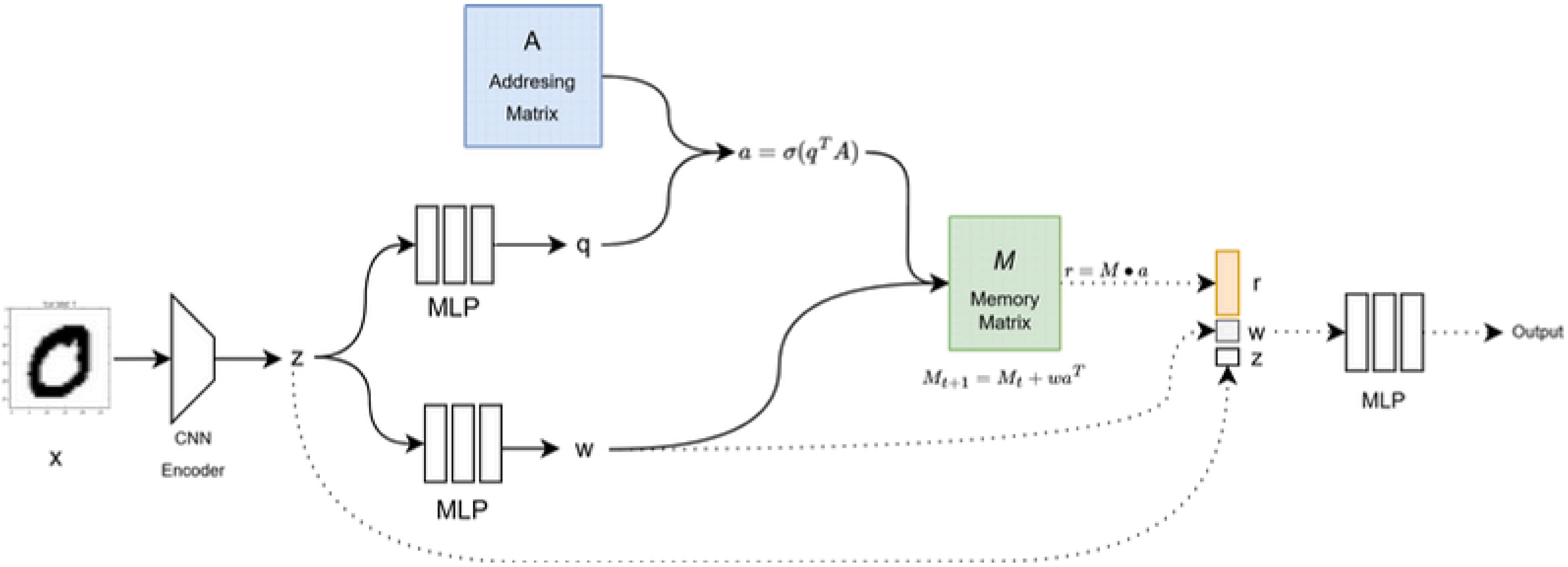
Key Components:

- **Encoder:**
  Maps input (e.g., image) to a dense embedding.
- **Write Network:**
  Encodes input into write vector and query vector.
- **Addressing Matrix (A):**
  Learns how to softly address memory slots (like learned hash functions).
- **Memory Matrix (M):**
  Stores distributed representations of inserted items.
- **MLP Decoder:**
  Predicts membership from read vector.

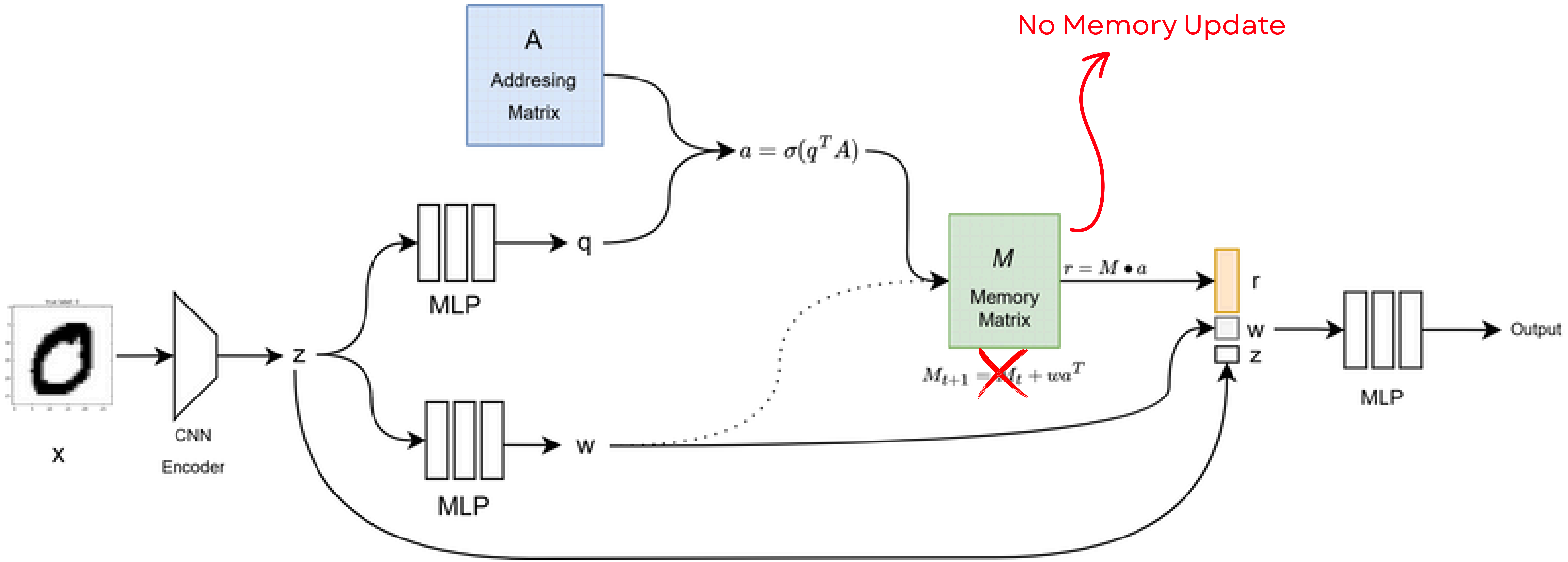
Model Architecture



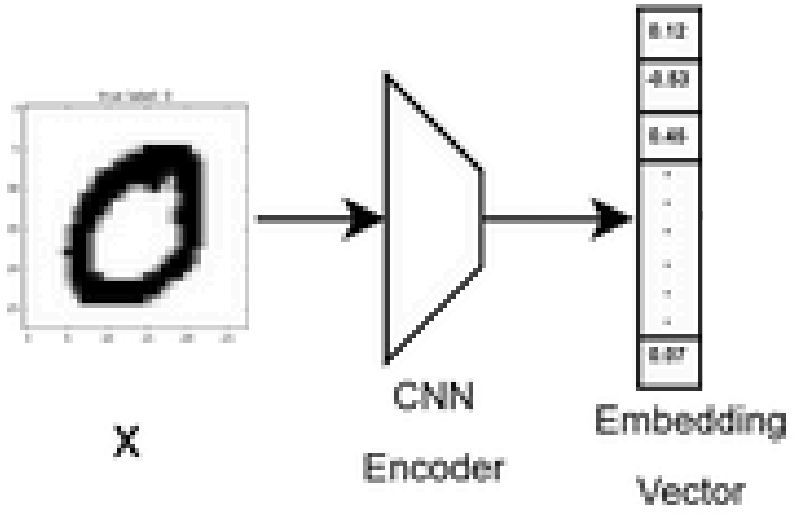
Writer Network



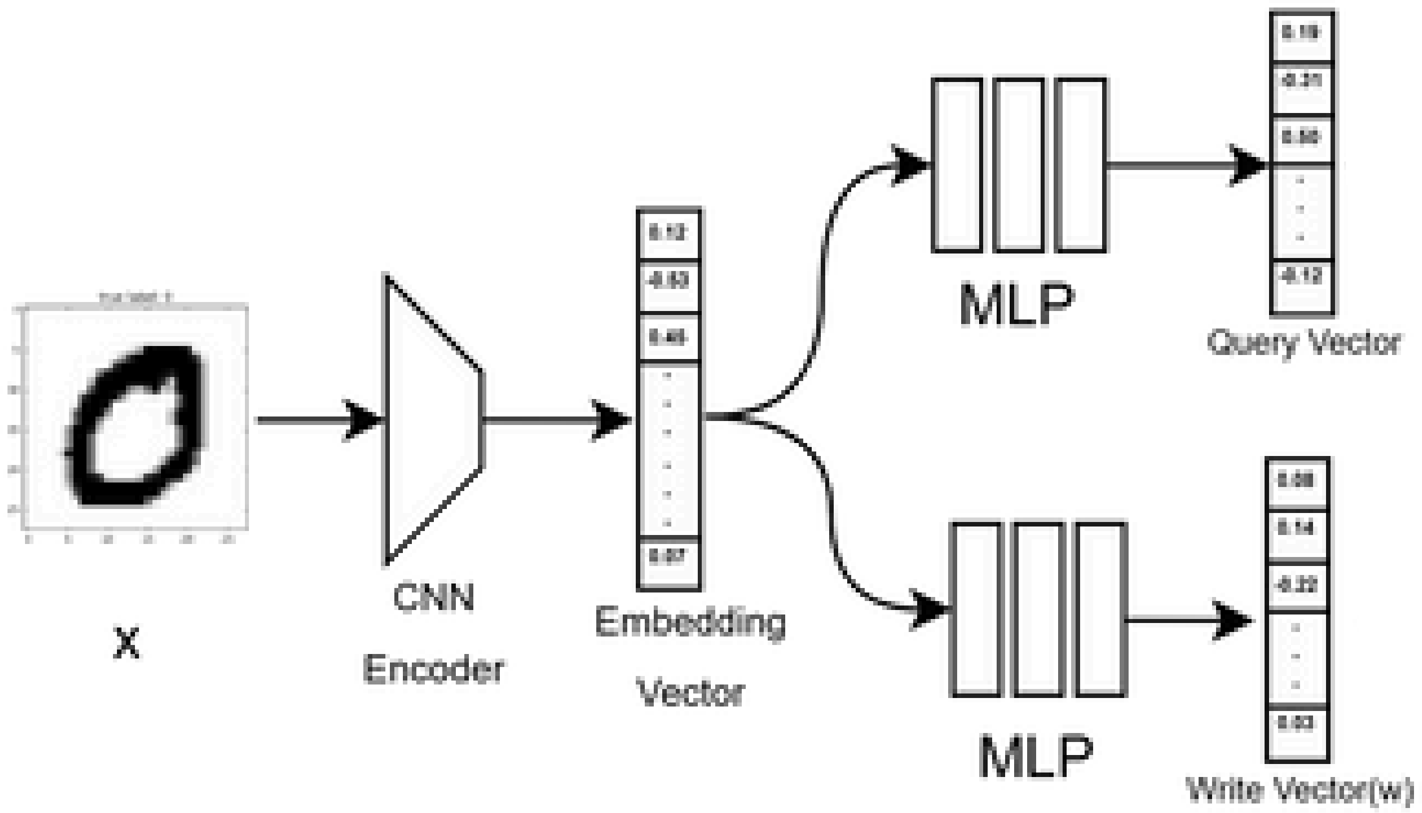
Read Network



Writer Flow

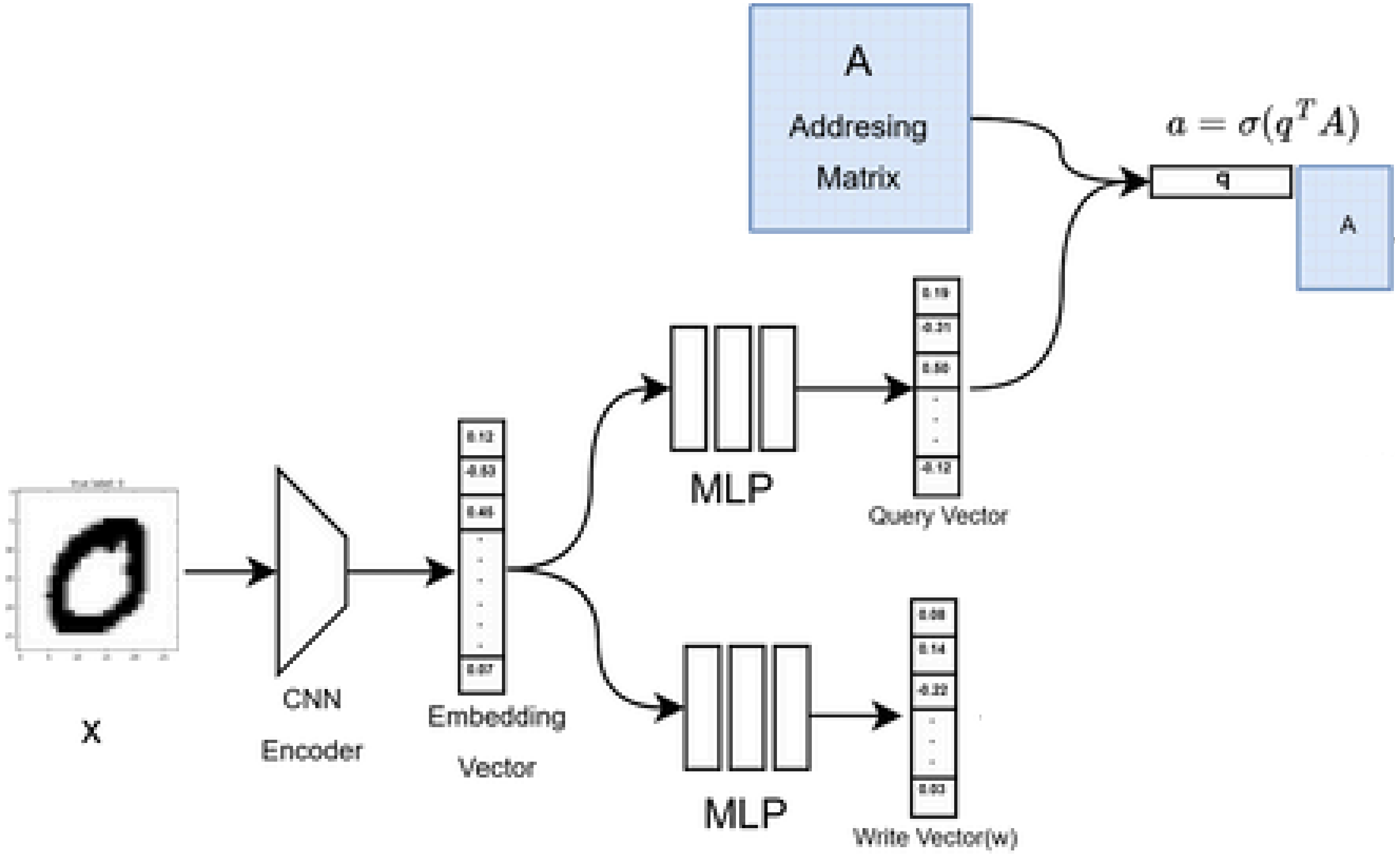


Writer Flow



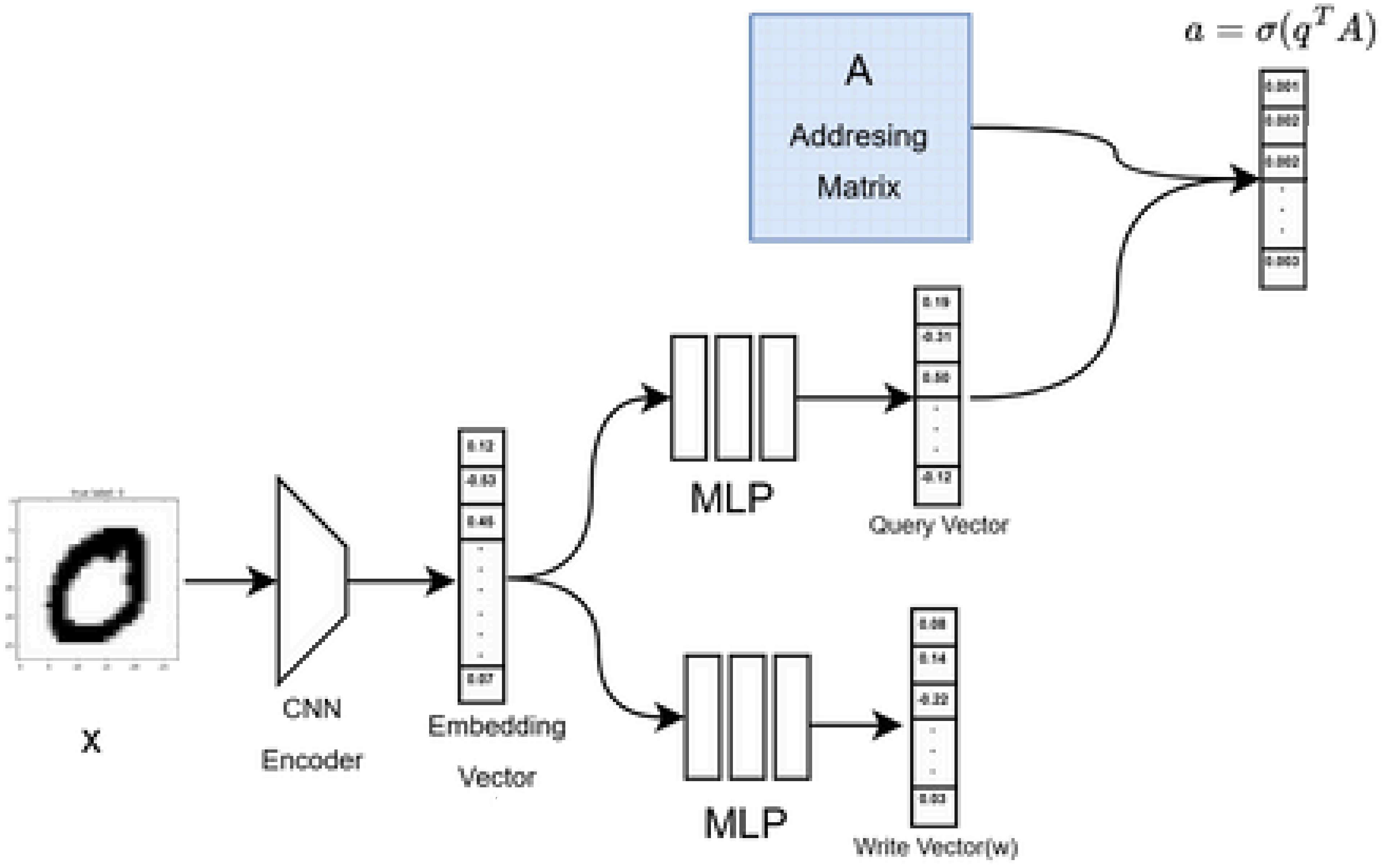
0.19
-0.21
0.50
-0.12
MLP
Query Vector
0.12
-0.53
0.45
0.07
X
CNN
Encoder
Embedding
Vector
0.08
0.14
-0.22
0.03
MLP
Write Vector(w)

Writer Flow



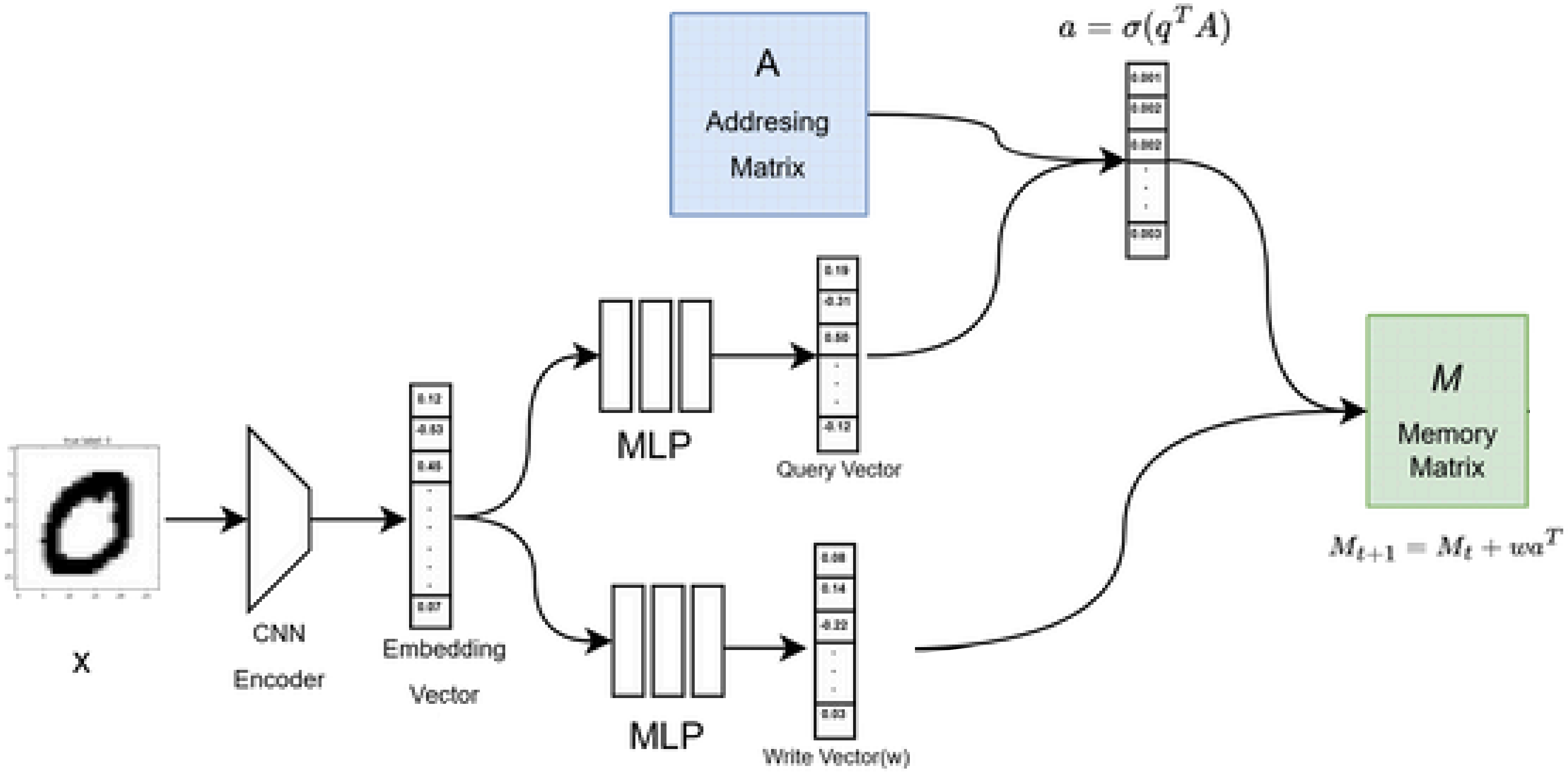
A
Addresing
Matrix
$a = \sigma(q^T A)$
q
A
0.19
-0.31
0.50
-0.12
Query Vector
MLP
0.12
-0.53
0.46
0.07
x
CNN
Encoder
Embedding
Vector
MLP
0.08
0.14
-0.22
0.03
Write Vector(w)

Writer Flow



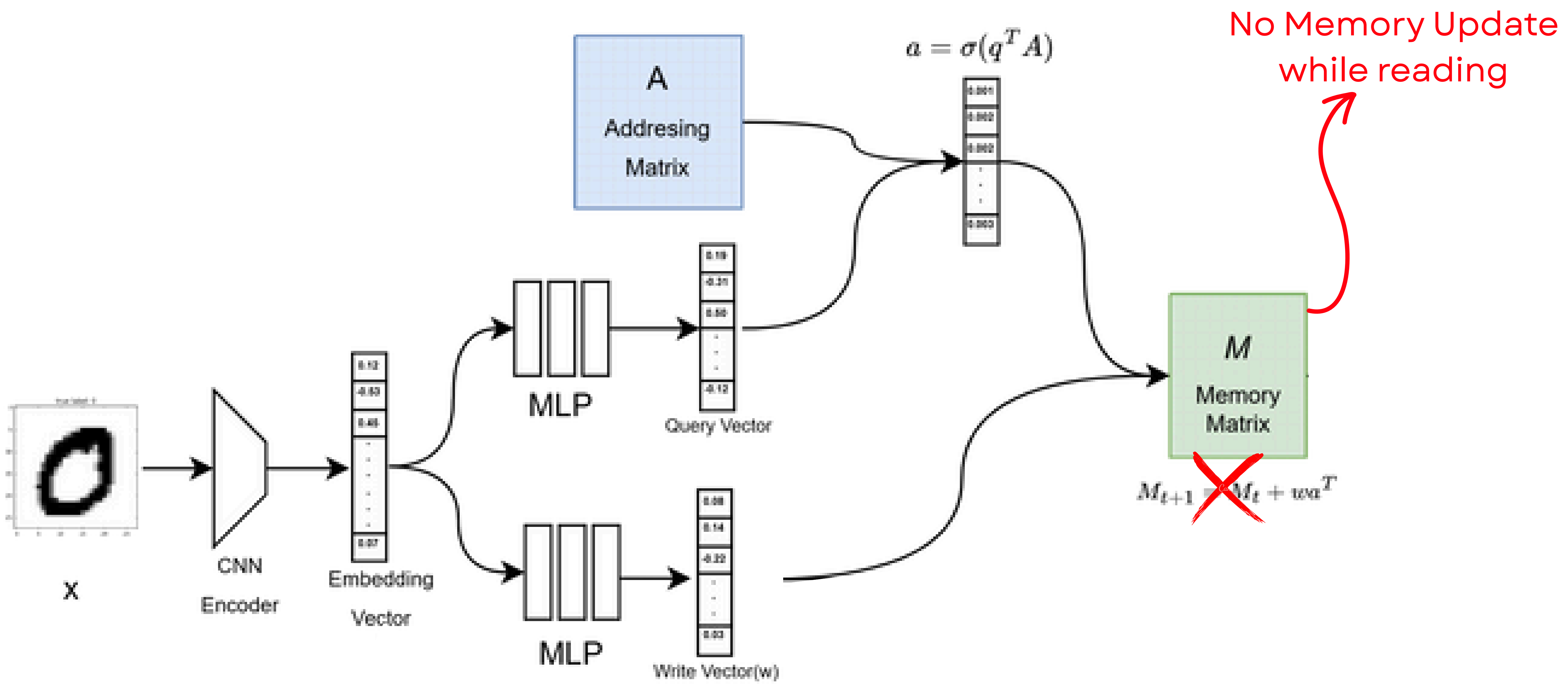
$a = \sigma(q^T A)$
A
Addresing
Matrix
MLP
Query Vector
MLP
Write Vector(w)
x
CNN
Encoder
Embedding
Vector

Writer Flow



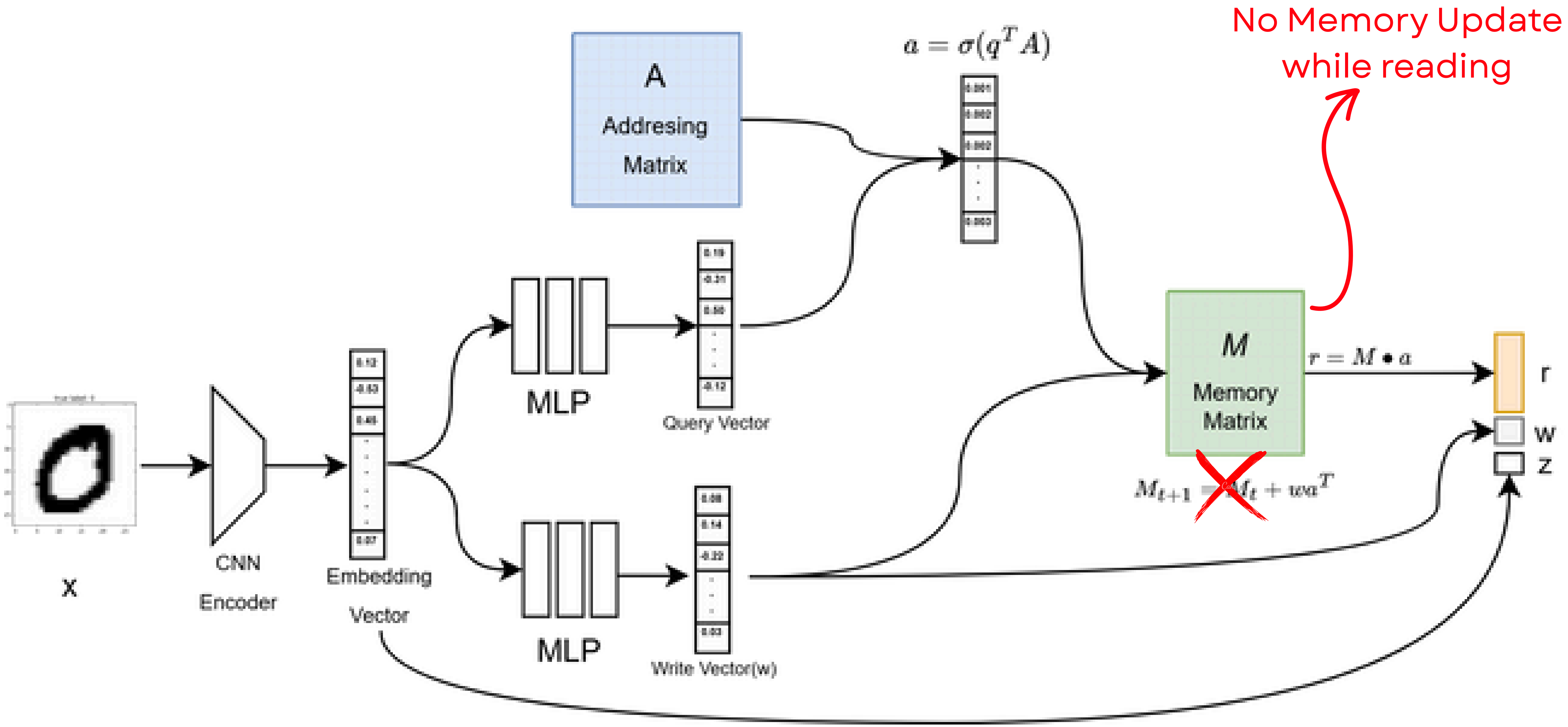
$a = \sigma(q^T A)$
A
Addresing
Matrix
MLP
Query Vector
M
Memory
Matrix
$M_{t+1} = M_t + wa^T$
X
CNN
Encoder
Embedding
Vector
MLP
Write Vector(w)

Reader Flow

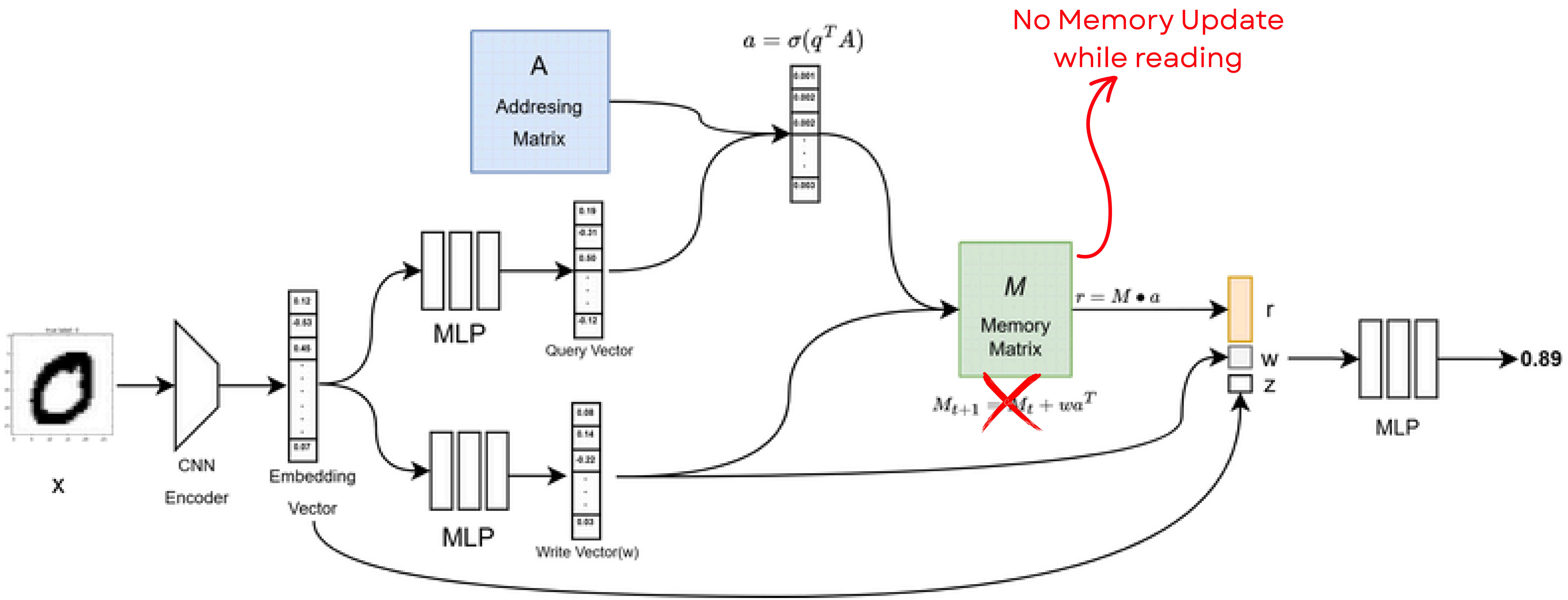


A
Addresing
Matrix
$a = \sigma(q^T A)$
No Memory Update
while reading
M
Memory
Matrix
$M_{t+1} = M_t + wa^T$
MLP
Query Vector
X
CNN
Encoder
Embedding
Vector
MLP
Write Vector(w)

Reader Flow


No Memory Update
while reading
A
Addresing
Matrix
$a = \sigma(q^T A)$
MLP
Query Vector
MLP
Write Vector(w)
X
CNN
Encoder
Embedding
Vector
M
Memory
Matrix
$r = M \bullet a$
$M_{t+1} = M_t + wa^T$
r
w
z

Reader Flow



No Memory Update
while reading
A
Addresing
Matrix
$a = \sigma(q^T A)$
MLP
Query Vector
M
Memory
Matrix
$r = M \bullet a$
$M_{t+1} = M_t + wa^T$
r
w
z
MLP
0.89
x
CNN
Encoder
Embedding
Vector
MLP
Write Vector(w)

Model Workflow

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)`:
  $a, w \leftarrow \texttt{controller}(x)$
  $M_{t+1} \leftarrow M_t + wa^\top$ ▷ Additive write
def `read(x)`:
  $a, w, z \leftarrow \texttt{controller}(x)$
  $r \leftarrow \texttt{flatten}(Ma)$ ▷ Read memory
  $o \leftarrow f_{out}([r, w, z])$ ▷ Output logit

Classical Bloom Filter Vs Neural Bloom Filter

Classical Bloom Filter	Neural Bloom Filter
Fixed Hashing Functions	Learned Addressing Matrix
Fixed Bit Array	Learned Memory Matrix

Training using Meta-Learning

Meta-learning trains the model across many tasks to quickly adapt to new ones with minimal data, improving the ability to learn new storage sets efficiently.

Algorithm 2 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

Neural Bloom Filter Implementation

```python
class SimpleEncoder(nn.Module):
    def __init__(self):
        super(SimpleEncoder, self).__init__()
        # A simple CNN encoder: input (batch_size,channels,height,width) -> output vector (batch_size,128)
        self.conv = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),  # 28x28 -> 14x14
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1), # 14x14 -> 7x7
            nn.ReLU(),
            nn.Flatten()
        )
        self.fc = nn.Linear(32 * 7 * 7, 128)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc(x)
        return x  # output shape: (batch_size, 128)
```

```python
class NeuralBloomFilter(nn.Module):
    def __init__(self, memory_slots=10, word_size=32, class_num=1):
        super().__init__()
        self.encoder = SimpleEncoder()
        self.fc_q    = nn.Linear(128, word_size)
        self.fc_w    = nn.Linear(128, word_size)
        self.A       = nn.Parameter(torch.randn(word_size, memory_slots))
        self.register_buffer('M', torch.zeros(memory_slots, word_size))
        inp_dim = memory_slots*word_size + word_size + 128
        self.mlp = nn.Sequential(
            nn.Linear(inp_dim, 128),
            nn.ReLU(),
            nn.Linear(128, class_num)
        )

    def controller(self, x):
        """
        x -> (B, 1, 28, 28): input image
        Runs the encoder, computes query q, write word w, and normalized address a.
        Returns (a, w, z).
        """
        z = self.encoder(x)                     # (B,128)
        q = self.fc_q(z)                        # (B,word_size)
        w = self.fc_w(z)                        # (B,word_size)
        a_logits = q @ self.A                   # (B,slots)
        a = F.softmax(a_logits, dim=1)          # (B,slots)
        return a, w, z
```

```python
    def write(self, x):
        """
        Write all x in one shot:
          M ← M + ∑_i w_i a_i^T
        """
        a, w, _ = self.controller(x)          # a:(B,slots), w:(B,word_size)
        # Outer product per sample: update_i[k,p] = a[i,k] * w[i,p]
        # Stack them and sum over batch:
        # shape: (B, slots, word_size)
        update = torch.einsum('bk,bp->bkp', a, w)
        # Add to memory and detach so writes don't backprop through time:
        self.M = self.M + update.sum(dim=0).detach()

    def read(self, x):
        """
        Read operation:
          r_i = flatten( M ⊙ a_i )  (componentwise)
          logits = f_out([r_i, w_i, z_i])
        """
        a, w, z = self.controller(x)          # (B,slots), (B,word_size), (B,128)
        # M ⊙ a_i: scale each row of M by a_i:
        # shape before flatten: (B, slots, word_size)
        r = (a.unsqueeze(2) * self.M.unsqueeze(0)).reshape(x.size(0), -1)  # (B, slots*word_size)
        # Concatenate r, w, z:
        concat = torch.cat([r, w, z], dim=1)                               # (B, inp_dim)
        logits = self.mlp(concat)                                           # (B, class_num)
        return logits,a
```

Meta Training Implementation

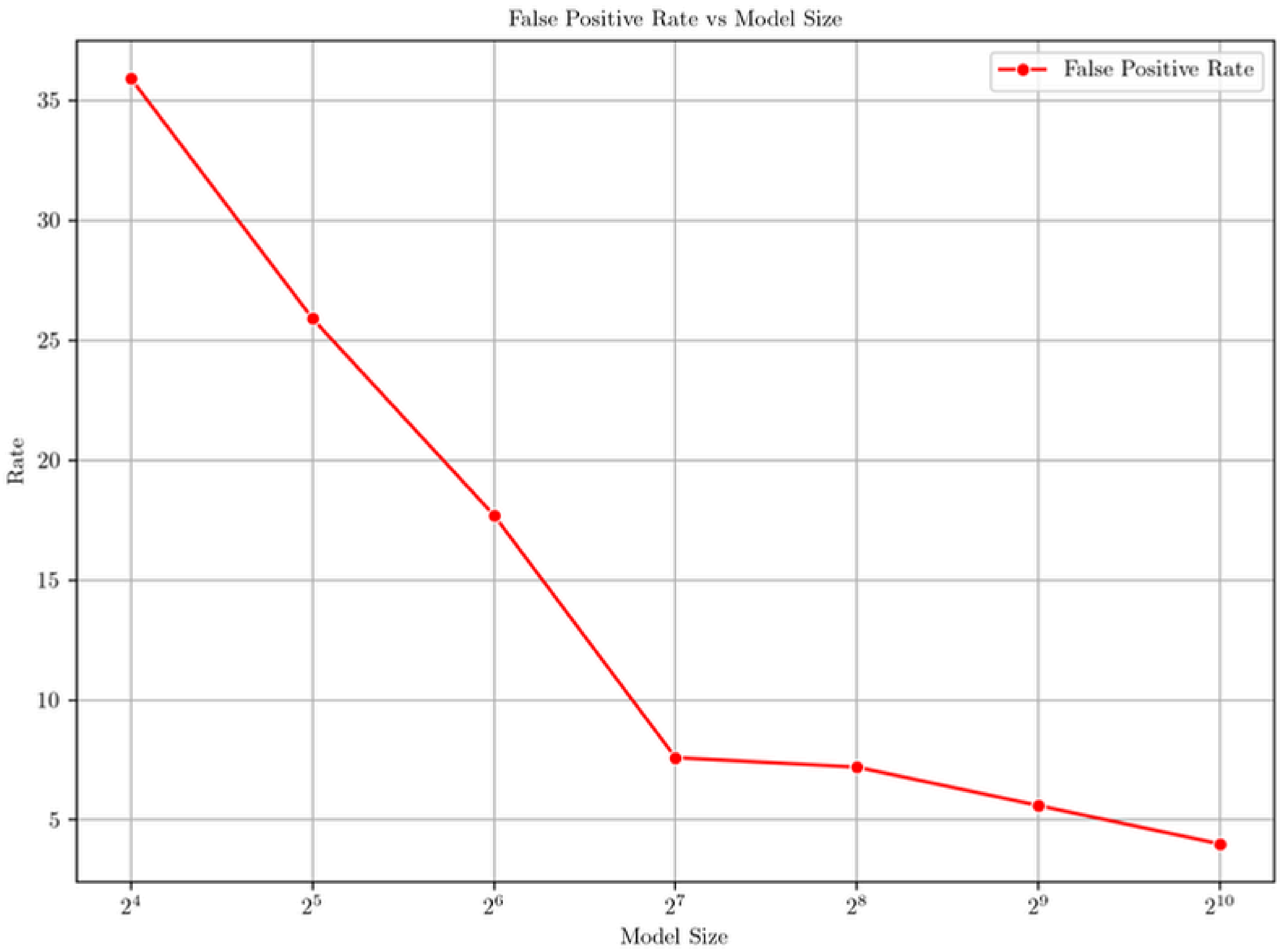
```python
def meta_train(model, dataset, labels, optimizer, criterion, device, meta_epochs=10, storage_set_size=60, num_queries=10, classes=[0,8,9,6]):
    model.train()
    total_loss = 0.0
    for epoch in range(meta_epochs):
        class_num = random.choice(classes)
        storage_indices, query_indices, targets= sample_task(labels, storage_set_size, num_queries, class_num)
        storage_images = dataset[storage_indices]
        storage_images = torch.tensor(storage_images, dtype=torch.float32).unsqueeze(1)
        query_images = dataset[query_indices]
        query_images = torch.tensor(query_images, dtype=torch.float32).unsqueeze(1)

        model.M.zero_()
        _ = model.write(storage_images)  # Expected shape: (storage_set_size, word_size)
        logits,_ = model.read(query_images)  # Expected shape: (num_queries, class_num)
        probs = torch.sigmoid(logits)
        # Convert probabilities to binary predictions
        predictions = (probs > 0.5).float()
        fnr = 0
        fpr = 0
        for i in range(len(predictions)):
            if predictions[i] == 0 and targets[i] == 1:
                fnr += 1
            elif predictions[i] == 1 and targets[i] == 0:
                fpr += 1
        loss = criterion(logits, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        if (epoch + 1) % 100 == 0:
            avg_loss = total_loss / (epoch + 1)
            print(f"Epoch [{epoch+1}/{meta_epochs}], Loss: {loss.item():.4f}, False Positive Rate: {fpr}, False Negative Rate: {fnr}")
    return total_loss / meta_epochs
```

```python
def sample_task(labels, storage_set_size, num_queries, class_num):
    class_to_indices = {}
    for idx, label in enumerate(labels):
        if label not in class_to_indices:
            class_to_indices[label] = []
        class_to_indices[label].append(idx)

    storage_indices = random.sample(class_to_indices[class_num], storage_set_size)

    num_in = num_queries // 2
    num_out = num_queries - num_in
    query_in_indices = random.sample(class_to_indices[class_num], num_in)
    other_classes = [c for c in class_to_indices if c != class_num]
    query_out_indices = []
    for _ in range(num_out):
        other_class = random.choice(other_classes)
        query_out_indices.append(random.choice(class_to_indices[other_class]))

    query_indices = query_in_indices + query_out_indices
    # define target as [1,0] for in-class and [0,1] for out-of-class
    targets = []
    for i in range(num_in):
        targets.append(1)
    for i in range(num_out):
        targets.append(0)

    targets = torch.tensor(targets, dtype=torch.float32).unsqueeze(1)
    return storage_indices, query_indices, targets
```

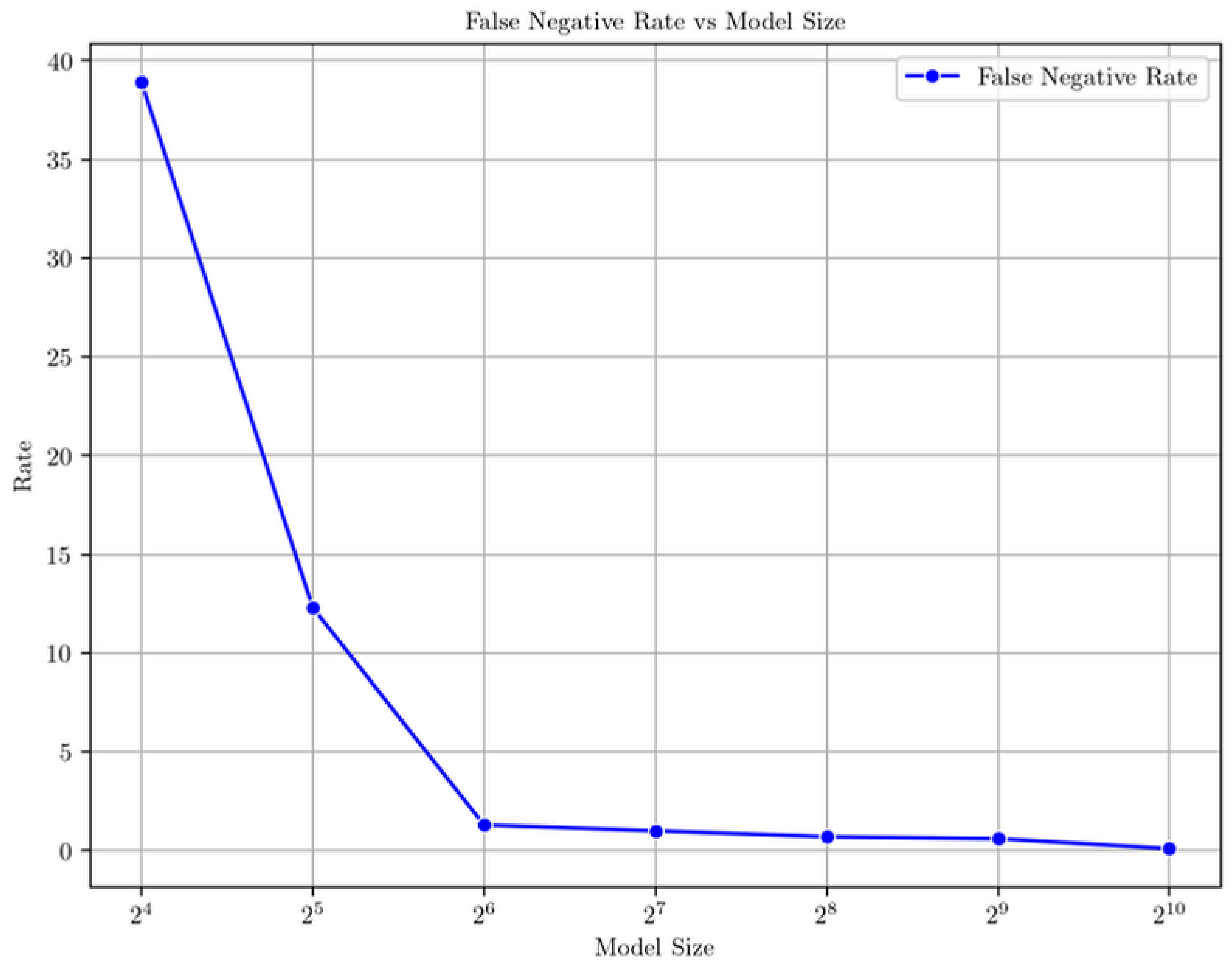
Mode Hyperparameters used for training

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

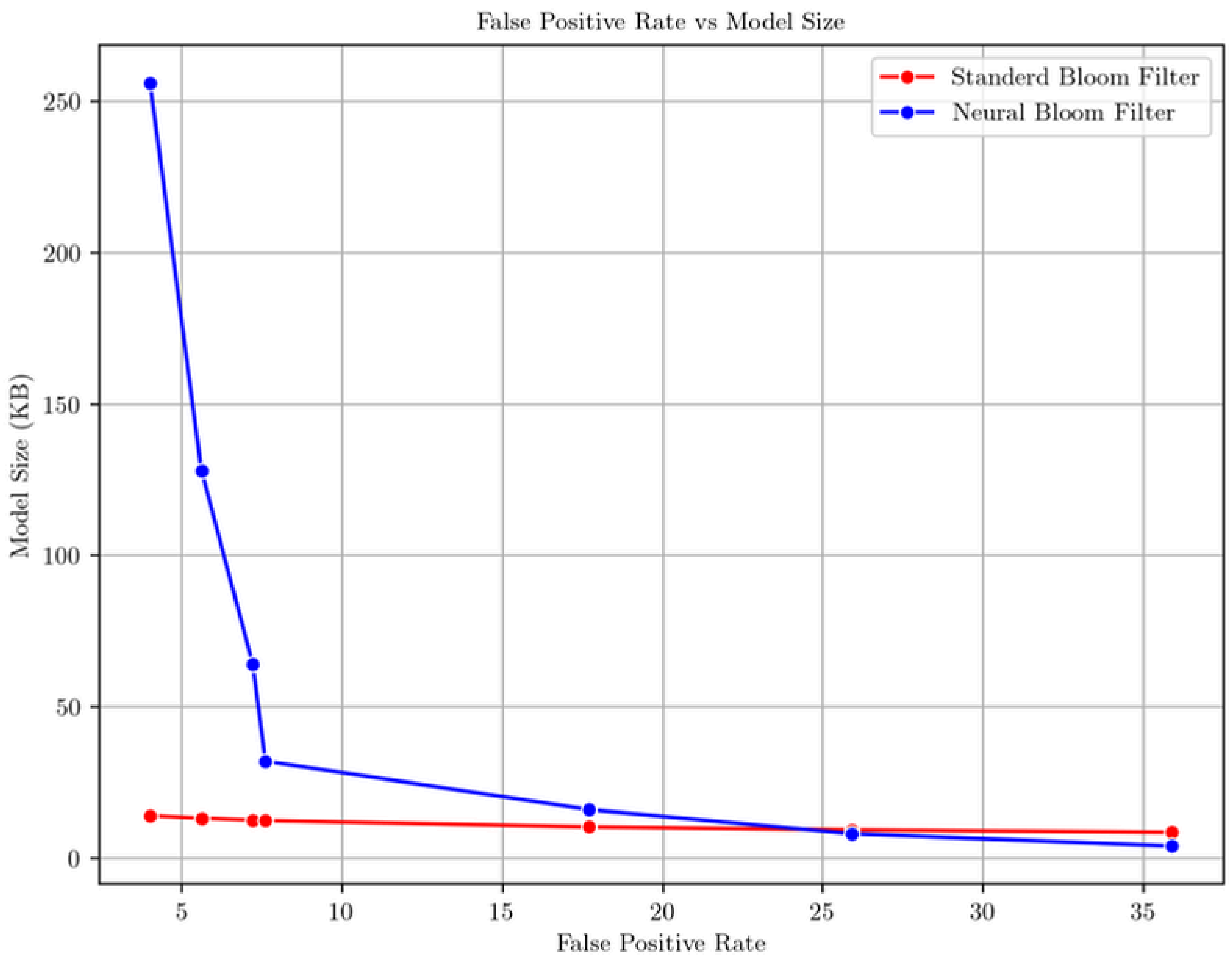
Results


False Positive Rate vs Model Size
False Positive Rate
Rate
35
30
25
20
15
10
5
2^4
2^5
2^6
2^7
2^8
2^9
2^{10}
Model Size

Results


False Negative Rate vs Model Size
False Negative Rate
Rate
40
35
30
25
20
15
10
5
0
2^4
2^5
2^6
2^7
2^8
2^9
2^{10}
Model Size

Results


False Positive Rate vs Model Size
Standerd Bloom Filter
Neural Bloom Filter
Model Size (KB)
0
50
100
150
200
250
5
10
15
20
25
30
35
False Positive Rate

Results

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

Advantages of NBF

Data-Adaptive Memory:

- Learns dataset structure (e.g., spatial patterns)
- Compact and efficient storage
- Performs well on image-like datasets

Improved Task-Specific Performance

- End-to-end training with downstream signals (e.g., classification loss)
- Optimizes both memory writes and reads.
- Outperforms heuristic hash functions.

Disadvantages of NBF

Nonzero False Negatives

- Unlike classical Bloom filters (zero FNR)
- Risk of missing true positives if memory underfits or overwrites

Dependence on Meta-Learning

- Requires structured task sampling
- Random batch training ➔ poor address learning
- Leads to degraded accuracy

Conclusion

Neural Bloom Filter (NBF) Summary
Extends classical Bloom Filters with neural network concepts
Trained and evaluated on MNIST dataset

Key Achievements
🔵 Lower false positive rate under memory constraints
🔵 Data-adaptive addressing through learned mechanisms
🔵 Soft memory updates for flexible set membership

Challenges
⚠ Introduction of false negatives
⚠ Supervised training requirement

Future Works

Backup Bloom Filters

- Integrate classical backup to reduce false negatives

Quantization & Compression

- Minimize memory requirements (floating-point optimization)

Training on Diverse Datasets

- Move beyond MNIST:
  - URL datasets
  - Genomic sequences

End-to-End Learned Hashing

- Joint optimization of hashing and addressing matrices

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

THANK YOU

TEAM - 27

Bhavik Patel (bhavik.patel@iitgn.ac.in)

Jinil Patel (jinil.patel@iitgn.ac.in)

Pranav Patil (pranav.patil@iitgn.ac.in)

Indian Institute of Technology Gandhinagar
Palaj, Gujarat - 382355