A. Further Model Details

A.1. Efficient addressing

We discuss some implementation tricks that could be employed for a production system.

Firstly the original model description defines the addressing matrix A to be trainable. This ties the number of parameters in the network to the memory size. It may be preferable to train the model at a given memory size and evaluate for larger memory sizes. One way to achieve this is by allowing the addressing matrix A to be non-trainable. We experiment with this, allowing $A \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ to be a fixed sample of Gaussian random variables. We can think of these as point on a sphere in high dimensional space, the controller network must learn to organize inputs into separate buckets across the surface of the sphere.

To make the addressing more efficient for larger memory sizes, we experiment with sparsification of the addressing softmax by preserving only the top k components. We denote this sparse softmax $\sigma_k(\cdot)$. When using a sparse address, we find the network can fixate on a subset of rows. This observation is common to prior sparse addressing work (Shazeer et al., 2017). We find sphering the query vector, often dubbed whitening, remedies this (see Appendix G for an ablation). The modified sparse architecture variant is illustrated in Algorithm 3.

Algorithm 3 Sparse Neural Bloom Filter

```
1:  def sparse_controller(x):
2:      z ← f_enc(x)
3:      s ← f_q(z)                         // Raw query word
4:      q ← moving_zca(q)                  // Spherical query
5:      a ← σ_k(q^T A)                     // Sparse address
6:      w ← f_w(z)

7:  def sparse_write(x):
8:      a, w ← sparse_controller(x)
9:      M_{t+1}[a_idx] ← M_t[a_idx] + w a_val^T

10: def sparse_read(x):
11:     a, w, z ← sparse_controller(x)
12:     r ← M[a_idx] ⊙ a_val
13:     o ← f_out([r, w, z])
```

One can avoid the linear-time distance computation $q^T A$ in the addressing operation $\sigma_k(q^T A)$ by using an approximate k-nearest neighbour index, such as locality-sensitive hashing (Datar et al., 2004), to extract the nearest neighbours from A in $\mathcal{O}(\log m)$ time. The use of an approximate nearest neighbour index has been empirically considered for scaling memory-augmented neural networks (Rae et al., 2016; Kaiser et al., 2017) however this was used for attention on M directly. As M is dynamic the knn requires frequent re-building as memories are stored or modified. This architecture is simpler — A is fixed and so the approximate knn can be built once.

To ensure the serialized size of the network (which can be shared across many memory instantiations) is independent of the number of slots in memory m we can avoid storing A. In the instance that it is not trainable, and is simply a fixed sample of random variables that are generated from a deterministic random number generator — we can instead store a set of integer seeds that can be used to re-generate the rows of A. We can let the i-th seed c_i, say represented as a 16-bit integer, correspond to the set of 16 rows with indices $16i, 16i + 1, \ldots, 16i + 15$. If these rows need to be accessed, they can be regenerated on-the-fly by c_i. The total memory cost of A is thus m bits, where m is the number of memory slots[3].

Putting these two together it is possible to query and write to a Neural Bloom Filter with m memory slots in $\mathcal{O}(\log m)$ time, where the network consumes $\mathcal{O}(1)$ space. It is worth noting, however, the Neural Bloom Filter's memory is often much smaller than the corresponding classical Bloom Filter's memory, and in many of our experiments is even smaller than the number of unique elements to store. Thus dense matrix multiplication can still be preferable - especially due to its acceleration on GPUs and TPUs (Jouppi et al., 2017) - and a dense representation of A is not inhibitory. As model optimization can become application-specific, we do not focus on these implementation details and use the model in its simplest setting with dense matrix operations.

A.2. Moving ZCA

The moving ZCA was computed by taking moving averages of the first and second moment, calculating the ZCA matrix and updating a moving average projection matrix θ_{zca}. This is only done during training, at evaluation time θ_{zca} is fixed. We describe the update below for completeness.

$$\text{Input: } s \leftarrow f_q(z) \tag{1}$$
$$\mu_{t+1} \leftarrow \gamma \mu_t + (1-\gamma)\bar{s} \quad \text{1st moment EMA} \tag{2}$$
$$\Sigma_{t+1} \leftarrow \gamma \Sigma_t + (1-\gamma)\; s^T s \quad \text{2nd moment EMA} \tag{3}$$
$$U, s, _ \leftarrow \texttt{svd}(\Sigma - \mu^2) \quad \text{Singular values} \tag{4}$$
$$W \leftarrow UU^T / \sqrt(s) \quad \text{ZCA matrix} \tag{5}$$
$$\theta_{zca} \leftarrow \eta \theta_{zca} + (1-\eta) W \quad \text{ZCA EMA} \tag{6}$$
$$q \leftarrow s\; \theta_{zca} \quad \text{Projected query} \tag{7}$$

In practice we do not compute the singular value decomposition at each time step to save computational resources, but

[3] One can replace 16 with 32 if there are more than one million slots

instead calculate it and update θ every T steps. We scale the discount in this case $\eta' = \eta/T$.

A.3. Relation to uniform hashing

We can think of the decorrelation of s, along with the sparse content-based attention with A, as a hash function that maps s to several indices in M. For moderate dimension sizes of s (256, say) we note that the Gaussian samples in A lie close to the surface of a sphere, uniformly scattered across it. If q, the decorrelated query, were to be Gaussian then the marginal distribution of nearest neighbours rows in A will be uniform. If we chose the number of nearest neighbours $k = 1$ then this implies the slots in M are selected independently with uniform probability. This is the exact hash function specification that Bloom Filters assume. Instead we use a continuous (as we choose $k > 1$) approximation (as we decorrelate $s \rightarrow q$ vs Gaussianize) to this uniform hashing scheme, so it is differentiable and the network can learn to shape query representations.

B. Space Comparison

For each task we compare the model's memory size, in bits, at a given false positive rate — usually chosen to be 1%. For our neural networks which output a probability $p = f(x)$ one could select an operating point τ_ϵ such that the false positive rate is ϵ. In all of our experiments the neural network outputs a memory (state) s which characterizes the storage set. Let us say SPACE(f, ϵ) is the minimum size of s, in bits, for the network to achieve an average false positive rate of ϵ. We could compare SPACE(f,ϵ) with SPACE(Bloom Filter,ϵ) directly, but this would not be a fair comparison as our network f can emit false negatives.

To remedy this, we employ the same scheme as Kraska et al. (2018) where we use a 'backup' Bloom Filter with false positive rate δ to store all false negatives. When $f(x) < \tau_\epsilon$ we query the backup Bloom Filter. Because the overall false positive rate is $\epsilon + (1-\epsilon)\delta$, to achieve a false positive rate of at most α (say 1%) we can set $\epsilon = \delta = \alpha/2$. The number of elements stored in the backup bloom filter is equal to the number of false negatives, denoted n_{fn}. Thus the total space can be calculated, TOTAL_SPACE(f,α) = SPACE(f,$\frac{\alpha}{2}$) + n_{fn} * SPACE(Bloom Filter,$\frac{\alpha}{2}$). We compare this quantity for different storage set sizes.

C. Model Size

For the MNIST experiments we used a 3-layer convolutional neural network with 64 filters followed by a two-layer feed-forward network with 64&128 hidden-layers respectively. The number of trainable parameters in the Neural Bloom Filter (including the encoder) is $243,437$ which amounts to 7.8Mb at 32-bit precision. We did not optimize the encoder architecture to be lean, as we consider it part of the library in a sense. For example, we do not count the size of the hashing library that an implemented Bloom Filter relies on, which may have a chain of dependencies, or the package size of TensorFlow used for our experiments. Nevertheless we can reason that when the Neural Bloom Filter is 4kb smaller than the classical, such as for the non-uniform instance-based familiarity in Figure 2b, we would expect to see a net gain if we have a collection of at least $1,950$ data-structures. We imagine this could be optimized quite significantly, by using 16-bit precision and perhaps using more convolution layers or smaller feed-forward linear operations.

For the database experiments we used an LSTM character encoder with 256 hidden units followed by another 256 feed-forward layer. The number of trainable parameters in the Neural Bloom Filter $419,339$ which amounts to 13Mb. One could imagine optimizing this by switching to a GRU or investigating temporal convolutions as encoders.

D. Hyper-Parameters

We swept over the following hyper-parameters, over the range of memory sizes displayed for each task. We computed the best model parameters by selecting those which resulted in a model consuming the least space as defined in Appendix B. This depends on model performance as well as state size. The Memory Networks memory size was fixed to equal the input size (as the model does not arbitrate what inputs to avoid writing).

Memory Size (DNC, NBF)	{2, 4, 8, 16, 32, 64}
Word Size (MemNets, DNC, NBF)	{2, 4, 6, 8, 10}
Hidden Size (LSTM)	{2, 4, 8, 16, 32, 64}
Sphering Decay η (NBF)	{0.9, 0.95, 0.99}
Learning Rate (all)	{1e-4, 5e-5}

Table 3. Hyper-parameters considered

E. Experiment Details

For the class-based familiarity task, and uniform sampling task, the model was trained on the training set and evaluated on the test set. For the class-based task sampling, a class is sampled at random and S is formed from a random subset of images from that class. The queries q are chosen uniformly from either S or from images of a different class.

For the non-uniform instance-based familiarity task we sampled images from an exponential distribution. Specifically we used a fix permutation of the training images, and from that ordering chose $p(i_{th} \text{ image}) \propto 0.999^i$ for the images to store. The query images were selected uniformly. We used

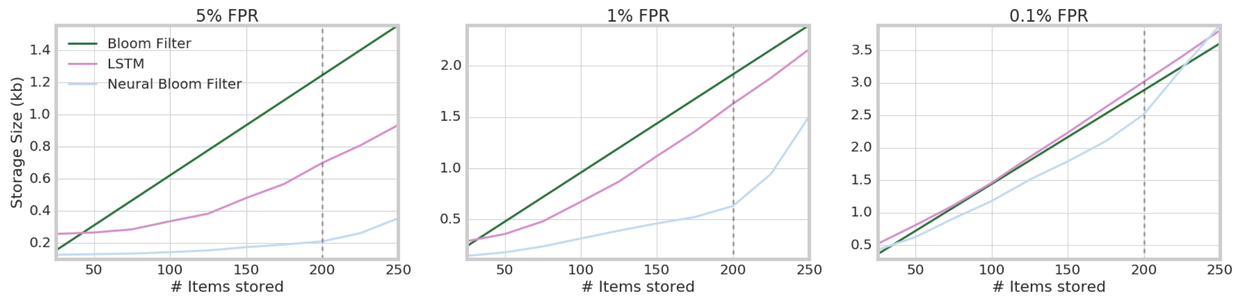

Figure 4. **Database extrapolation task**. Models are trained up to sets of size 200 (dashed line). We see extrapolation to larger set sizes on test set, but performance degrades. Neural architectures perform best for larger allowed false positive rates.

a fixed permutation (or shuffle) of the images to ensure most probability mass was not placed on images of a certain class. I.e. by the natural ordering of the dataset we would have otherwise almost always sampled 0 images. This would be confounding task non-uniformity for other latent structure to the sets. Because the network needed to relate the image to its frequency of occurence for task, the models were evaluated on the training set. This is reasonable as we are not wishing for the model to visually generalize to unseen elements in the setting of this exact-familiarity task. We specifically want the network weights to compress a map of image to probability of storage.

For the database task a universe of $2.5M$ unique tokens were extracted from GigaWord v5. We shuffled the tokens and placed 2.3M in a training set and 250K in a test set. These sets were then sorted alphabetically. A random subset, representing an SSTable, was sampled by choosing a random start index and selecting the next n elements, which form our set S. Queries are sampled uniformly at random from the universe set. Models are trained on the training set and evaluated on the test set.

F. Database Extrapolation Task

We investigate whether neural models are able to extrapolate to larger test sizes. Using the database task setup, where each set contains a contiguous set of sorted strings; we train both the Neural Bloom Filter and LSTM on sets of sizes 2 - 200. We then evaluate on sets up to 250, i.e. a 25% increase over what is observed during training. This is to emulate the scenario that we train on a selection of databse tablets, but during evaluation we may observe some tablets that are slightly larger than those in the training set. Both the LSTM and Neural Bloom Filter are able to solve the task, with the Neural Bloom Filter using significantly less space for the larger allowed false positive rate of 5% and 1%. We do see the models' error increase as it surpasses the maximum training set size, however it is not catastrophic. Another interesting trend is noticeable; the neural models have higher utility for larger allowed false positive rates. This may be because of the difficulty in training the models to an extremely low accuracy.

G. Effect of Sphering

We see the benefit of sphering in Figure 5 where the converged validation performance ends up at a higher state. Investigating the proportion of memory filled after all elements have been written in Figure 6, we see the model uses quite a small proportion of its memory slots. This is likely due to the network fixating on rows it has accessed with sparse addressing, and ignoring rows it has otherwise never touched — a phenomena noted in Shazeer et al. (2017). The model finds a local minima in continually storing and accessing the same rows in memory. The effect of sphering is that the query now appears to be Gaussian (up to the first two moments) and so the nearest neighbour in the address matrix A (which is initialized to Gaussian random variables) will be close to uniform. This results in a more uniform memory access (as seen in Figure 6) which significantly aids performance (as seen in Figure 5).

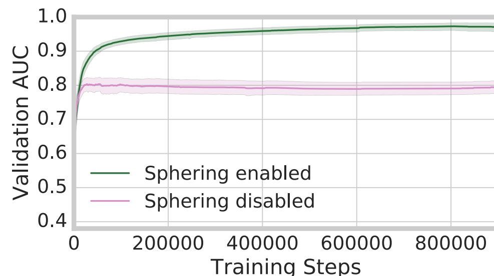

Figure 5. For sparse addresses, sphering enables the model to learn the task of set membership to high accuracy.

H. Timing Benchmark

We use the Neural Bloom Filter network architecture for the large database task (Table 1). The network uses an encoder LSTM with 256 hidden units over the characters, and feeds

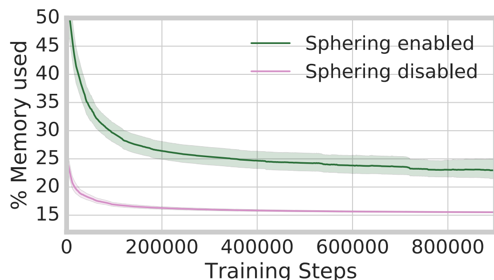

Figure 6. For sparse addresses, sphering the query vector leads to fewer collisions across memory slots and thus a higher utilization of memory.

this through a 256 fully connected layer to encode the input. A two-layer 256-hidden-unit MLP is used as the query architecture. The memory and word size is 8 and 4 respectively, and so the majority of the compute is spent in the encoder and query network. We compare this with an LSTM containing 32 hidden units. We benchmark the single-query latency of the network alongside the throughput of a batch of queries, and a batch of inserts. The Neural Bloom Filter and LSTM is implemented in TensorFlow without any custom kernels or specialized code. We benchmark it on the CPU (Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz) and a GPU (NVIDIA Quadro P6000). We compare to empirical timing results published in a query-optimized Bloom Filter variant (Chen et al., 2007).

It is worth noting, in several Bloom Filter applications, the actual query latency is not in the critical path of computation. For example, for a distributed database, the network latency and disk access latency for one tablet can be orders of magnitude greater than the in-memory latency of a Bloom Filter query. For this reason, we have not made run-time a point of focus in this study, and it is implicitly assumed that the neural network is trading off greater latency for less space. However it is worth checking whether run-time could be prohibitive.