Spiking Neural Networks Through the Lens of Streaming Algorithms

Yael Hitron
Weizmann Institute of Science, Rehovot, Israel
yael.hitron@weizmann.ac.il

Cameron Musco
University of Massachusetts, Amherst, MA, USA
cmusco@cs.umass.edu

Merav Parter
Weizmann Institute of Science, Rehovot, Israel
merav.parter@weizmann.ac.il

Abstract

We initiate the study of biologically-inspired *spiking neural networks* from the perspective of streaming algorithms. Like computers, human brains face memory limitations, which pose a significant obstacle when processing large scale and dynamically changing data. In computer science, these challenges are captured by the well-known streaming model, which can be traced back to Munro and Paterson '78 and has had significant impact in theory and beyond. In the classical streaming setting, one must compute a function f of a stream of updates $\mathcal{S} = \{u_1, \ldots, u_m\}$, given restricted single-pass access to the stream. The primary complexity measure is the space used by the algorithm.

In contrast to the large body of work on streaming algorithms, relatively little is known about the computational aspects of data processing in spiking neural networks. In this work, we seek to connect these two models, leveraging techniques developed for streaming algorithms to better understand neural computation. Our primary goal is to design networks for various computational tasks using as few auxiliary (non-input or output) neurons as possible. The number of auxiliary neurons can be thought of as the "space" required by the network.

Previous algorithmic work in spiking neural networks has many similarities with streaming algorithms. However, the connection between these two space-limited models has not been formally addressed. We take the first steps towards understanding this connection. On the upper bound side, we design neural algorithms based on known streaming algorithms for fundamental tasks, including distinct elements, approximate median, and heavy hitters. The number of neurons in our solutions almost match the space bounds of the corresponding streaming algorithms. As a general algorithmic primitive, we show how to implement the important streaming technique of linear sketching efficiently in spiking neural networks. On the lower bound side, we give a generic reduction, showing that any space-efficient spiking neural network can be simulated by a space-efficient streaming algorithm. This reduction lets us translate streaming-space lower bounds into nearly matching neural-space lower bounds, establishing a close connection between the two models.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases Biological distributed algorithms, Spiking neural networks, Streaming algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.10

Related Version `https://arxiv.org/abs/2010.01423`

Funding Supported by the BSF-NSF-Computer Science grant no. 713043.

Acknowledgements We are very grateful to Eylon Yogev for various discussions on pseudorandom generators, and for pointing out to us Lemma 27.

1 Introduction

In this work, we seek to understand the role of *memory constraints* in neural data processing. We consider data-stream tasks, in which a long stream of inputs is presented over time and a neural network must evaluate some function f of this stream. Examples include identifying frequent input patterns (items) or estimating summary statistics, such as the number of distinct items presented. The network cannot store the full stream and so must maintain some form of compressed representation in its working memory, which allows the eventual computation of f. The primary objective is to compute f with as few auxiliary (non-input or output) neurons as possible. The number of auxiliary neurons can be thought of as the "space" required by the network.

In computer science, data processing under space limitations is extensively studied in the area of streaming algorithms [37, 38]. We leverage this body of work to further our understanding of space-efficient neural networks. We start by designing neural networks for a large class of data-stream tasks, building off fundamental streaming algorithms and techniques, such as linear sketching. We also establish general connections between these models, showing that streaming-space lower bounds can be translated to neural-space lower bounds. We hope that these connections are a first step in extending work on streaming computation to better understand neural processing of massive and dynamically changing data under memory constraints.

The spiking neural network (SNN) model [32, 33]. A spiking network is represented by a directed weighted graph over n input neurons, r output neurons, and s auxiliary neurons. The edges of the graph represent synapses of different strengths connecting the neurons. The network evolves in discrete, synchronous rounds as a Markov chain where each neuron u acts as a (possibly probabilistic) threshold gate that either fires (spikes) or is silent in each round. In round t, the firing status of u depends on the firing status of its incoming neighbors in the preceding round $t-1$, and the strength of the connections from these neighbors. In *randomized* SNNs, there are two sources of randomness: the spiking behavior of the neurons and the selection of random edge weights in the network. In *deterministic* SNNs, the neurons are deterministic threshold gates and the edge weights are deterministically chosen. Aside from their relevance in modeling biological computation, SNNs have received significant attention as more energy efficient alternatives to traditional artificial neural networks [25, 42].

A recent series of works in the emerging area of *algorithmic SNNs* [33, 34, 11, 29, 28, 43, 8, 26, 41, 35, 40, 16] focuses on network design tasks. In this framework, given a target function $f : \{0,1\}^n \to \{0,1\}^r$, one seeks to design a space-efficient SNN (with few auxiliary neurons) that converges rapidly to an output spiking pattern matching $f(x)$ when the input spiking pattern matches x. Space-efficient SNNs have been devised for the winner-takes-all problem [28, 41], similarity testing and compression [31, 40], clustering [14, 26], approximate counting, and time estimation [30, 15]. Interestingly, many of these works borrow ideas from related streaming algorithms. However, despite the flow of ideas from streaming to neural algorithms, the connection between these models has not been studied formally.

The streaming model [37, 38]. A data-stream is a sequence of updates $\mathcal{S} = \{u_1, \ldots, u_m\}$. A streaming algorithm $\mathcal{A}$ computes some function of $\mathcal{S}$, given restricted access to the stream. In the standard single-pass model, the algorithm can only read the updates in $\mathcal{S}$ once, in the order they are presented.

Most commonly, and throughout this work, each update u_i represents the insertion or deletion of an item x_i belonging to a universe U with $|U| = n$. Without loss of generality, we will always consider U to be the set of integers $[n] = 1, \ldots, n$, and f is a function of the frequency vector $\bar{z} \in \mathbb{R}^n$, which tracks the total frequency of each item in the stream (the number of insertions minus the number of deletions). In the *insertion-only* setting, only insertions are allowed – i.e., each update increments some entry of $\bar{z}$. In the general *turnstile* (dynamic) setting, there are both insertions and deletions – i.e., increments and decrements to entries in $\bar{z}$. The primary complexity measure of a streaming algorithm is the *space* (measured in number of bits) required to maintain the evaluation of f on the data-stream.

Neural networks from a streaming perspective. Our primary goal is to devise space-efficient spiking neural networks that solve natural data-stream tasks, which mirror data processing tasks solved in real biological networks. In light of the large collection of space-efficient streaming algorithms that have been designed for various problems, we start by asking:

▶ **Question 1.** Is it possible to translate a space-efficient streaming algorithm for a given task into a space-efficient SNN algorithm for that task? Do generic reductions from SNNs to streaming exist?

The streaming literature is also rich with space lower bounds. For many classical data-stream problems, these lower bounds are nearly tight. To obtain space lower bounds for SNNs, we ask if reductions in the reverse direction exist:

▶ **Question 2.** Is it possible to translate a space-efficient SNN for a given task into a space-efficient streaming algorithm for that task?

An affirmative answer to both of these questions would imply that the streaming and SNN models are, roughly speaking, computationally equivalent. A priori, it is unclear if this is the case. On the one hand, streaming algorithms have the potential to be more space-efficient than SNNs. For example, a space-efficient algorithm may still have a lengthy description, which is not taken into account in its space complexity. In the SNN setting, where the algorithm description and memory are both encoded by the auxiliary neurons in the network and their connections, a lengthy description may lead to a large, and hence not space-efficient network.

On the other hand, SNNs have the potential to be more space-efficient than streaming algorithms. For example, a randomized SNN with a large number of input neurons but a small number of auxiliary neurons may have a large number of random bits encoded in random connections between its inputs and auxiliary neurons. These bits are not counted as part of its space complexity. In contrast, a streaming algorithm that requires persistent access to many random bits must store these bits, possibly leading to large space complexity.

1.1 Our Results

We take the first steps towards formally understanding the connections between streaming algorithms and spiking neural networks. The first part of the paper is devoted to studying upper bounds for SNNs, addressing Question 1. We design space-efficient neural networks for a wide class of streaming problems by simulating their respective streaming algorithms. These simulations must overcome several challenges in implementing traditional algorithms in neural networks. Most notably, in an SNN, the spiking status of the auxiliary neurons encodes the working memory of the algorithm, and their connections encode the algorithm

itself. A space-efficient network with few auxiliary neurons thus inherently has limited ability to express complex algorithms. In many data-stream algorithms, the target space complexity is only polylogarithmic in the input size, making this challenge significant. Additionally, unlike traditional algorithms, a neural network evolves continuously in response to its inputs. This leads to synchronization issues – for example, if an input is not presented for a sufficient number of rounds, the firing status of the network may not converge to a proper state before the next input is presented.

The second part of the paper focuses on lower-bound aspects, addressing Question 2. We show that any space-efficient neural network can be translated into a space-efficient streaming algorithm, while paying a small additive term (logarithmic in the stream length/universe size). For deterministic SNNs, such a reduction is not difficult. For randomized SNNs, the reduction is more involved, as it must account for the large number of random bits that may be implicitly stored in the random edge weights of the network. Throughout, we use the $\widetilde{O}()$ notation to hides factors that are poly-logarithmic in n, m and $1/\delta$, where n is the size of the domain, m is a bound on the stream length and δ is the error parameter.

1.1.1 Efficient Streaming Algorithms Yield Efficient SNNs

We consider data-stream tasks in which each update is an insertion or deletion of an integer item $x \in [n]$, and f is a function of the frequency vector $\bar{z} \in \mathbb{R}^n$ of these items. In the streaming setting, each update can be thought as an n-length vector with a single ± 1 entry, corresponding to an item insertion or deletion. In the SNN setting, each update may be encoded as the firing of one of n input neurons along with a sign neuron indicating if the update is an increment or a decrement. Or, the update may be encoded via $O(\log n)$ input neurons, indicating the item to be inserted or deleted. These different encodings correspond to different natural settings – the first corresponds to a network that collects firing statistics from a large set of inputs and the second to a network that records statistics on a large number of possible input patterns, encoded in the spiking patterns of a smaller number of input neurons.

In either case, each input is presented for some *persistence time*, a certain number of rounds in which the input is fixed to allow the network state to converge before the next input is presented.

Linear sketching. A linear sketching algorithm is a streaming algorithm in which the state of the algorithm is a linear function of the updates seen so far. In particular, the state can be represented as the multiplication of a sketching matrix $A \in \mathbb{R}^{r \times n}$ with the frequency vector $\bar{z} \in \mathbb{R}^n$. Such algorithms have many useful properties applicable in both the turnstile setting and in distributed settings. For example, the additive nature of these algorithms allows one to split the data-stream across multiple sites, which can process the data in an independent manner. Additionally, the obliviousness of linear sketching algorithms to the ordering of the stream yields an efficient generic derandomization scheme using the Nisan's PRG for space bounded computation [18]. Linear sketching algorithms constitute the state-of-the-art algorithms for essentially all problems in the turnstile model, including heavy-hitters, coresets for clustering problems [19], and ℓ_p estimation [9]. In fact, Li, Nguyen and Woodruff [27] present a general reduction from the streaming turnstile model to linear sketching. This reduction, and its caveats have been further studied in a recent work by Kallaugher and Price [21]. Given their ubiquity in turnstile streaming algorithms, an important step in designing space-efficient SNNs for data-stream problems is an efficient implementation of linear sketching in the neural setting. We give such an implementation:

▶ **Theorem 3** (Linear Sketch). *Let $\mathcal{A}$ be an algorithm approximating a function $f(\bar{x})$ in the turnstile model using a linear sketch with an integer matrix A of size $r \times n$. Let ℓ be a bound on the maximum entry in $|A\bar{x}|$ for every item $\bar{x}$. There exists a network $\mathcal{N}$ with $n+1$ input neurons, $r \cdot (\lceil \log \ell \rceil + 1)$ output neurons, $O(r \cdot \log \ell)$ auxiliary neurons which implements $\mathcal{A}$ in the following sense. The first n input neurons $x = (x_1, \ldots, x_n)$ represent the inserted item $[1, n]$, and the additional input neuron s indicates the sign of the update. Each input update has a persistence time of $O(\log \ell)$ rounds. The output neurons are divided into r vectors $\bar{y}_1, \ldots, \bar{y}_r$ each of length $\log \ell$, and r neurons $s_1, \ldots, s_r$. For every $i \in \{1, \ldots, r\}$, the decimal value of the binary vector $\bar{y}_i$ is equal to the absolute value of the i^{th} entry of $A \cdot \bar{z}$, and the sign neuron s_i indicates the sign, where $\bar{z}$ is the summations of all input items presented in the current stream.*

Theorem 3 applies to linear sketches using integer matrices, which are commonly used, see [27]. Via scaling, the construction can be extended to rational matrices as well. We note that the network of Theorem 3 does not implement the "decoding" step which estimates $f(\bar{z})$ from $A \cdot \bar{z}$. This step depends on the problem being solved, however it is often very simple and thus implementable via a space-efficient SNN. E.g., in ℓ_p norm estimation one might just have to compute the ℓ_p norm of $A \cdot \bar{z}$ [18]. In frequency estimation, one might have to compute an average of a subset of entries in $A \cdot \bar{z}$ [5].

Beyond our generic linear sketching reduction, we give neural solutions for two challenging problems in the insertion-only model, namely, distinct elements and median estimation. These simulation results are less general and provide several tools for bypassing critical obstacles that arise in streaming to SNN reductions.

Distinct elements. In the *distinct elements problem* one must approximate the number of distinct items appearing in a data-stream with repeated items. It is well known that an exact solution by a single-pass streaming algorithm requires linear space. In fact, as we discuss later on, one can also show that the exact computation requires linear space in the SNN setting. Therefore, we restrict our attention to $(1+\epsilon)$ approximation for the number of distinct elements for any $\epsilon \in (0, 1)$. This problem has been studied thoroughly in the streaming literature [6, 2, 12, 13, 22, 3, 20, 44, 1].

In this work, we provide an efficient neural implementation for the well-known LogLog streaming algorithm by [12, 13]. The LogLog and its improved variant the hyper-Loglog algorithms provide sub-optimal space bounds, but due to their simplicity they are commonly used in practice. As we will see, they are efficiently implementable in the neural setting. In addition, we provide a nearly matching space lower bound.

▶ **Theorem 4** (Neural Computation of Distinct Elements). *For every $n \in \mathbb{N}$, $\epsilon, \delta \in (0, 1)$, given n input neurons $\bar{x}$ representing the elements in $[n]$ there exists a network $\mathcal{N}$ with $\log n$ output neurons and $\widetilde{O}(1/\epsilon^2)$ auxiliary neurons that encode the logarithm of an $(1 \pm \epsilon)$ approximation of the number of distinct elements in the current stream, with probability $1 - \delta$. In addition, any SNN requires $\Omega(\log n + 1/\epsilon^2)$ neurons to compute an $(1 \pm \epsilon)$ approximation for the problem, with constant probability.*

The lower bound is obtained via a communication complexity reduction that mimics the corresponding streaming reduction. We note that this reduction works perfectly, i.e., without any asymptomatic loss in the space-bound (compared to the streaming bound).

Count-Min sketch. A common tool used in many of the streaming algorithms is the Count-Min sketch data structure, which maintains frequency estimates for all items in a stream. Count-Min sketch is in fact a linear sketch, and thus can be implemented via

Theorem 3. However, it is not immediately clear how to implement certain important operations, like approximate frequency (count) queries via this reduction. We thus provide a direct implementation. Our implementation applies in the setting where there are $O(\log n)$ input neurons representing each insertion/deletion of an item $x \in [n]$. However, it can easily be extended to the setting in which there are n input neurons, one for each item.

▶ **Definition 5** (Count-Min Sketch [10]). *Given parameters $\epsilon, \delta > 0$, the Count-Min sketch is a probabilistic data structure that serves as a frequency table of items in a stream. It supports two operations: (i)* $\mathrm{inc}(x)$ *increases the frequency of x by one; (ii)* $\mathrm{count}(x)$ *returns an $(1+\epsilon)$ approximation of the frequency of x with probability $1-\delta$.*

For given parameters $\epsilon, \delta > 0$, the Count-Min sketch data structure contains $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$ each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise-independent hash function h_i. The $\mathrm{inc}(x)$ operation applies $T_i[h_i(x)] \leftarrow T_i[h_i(x)] + 1$ for every $i \in [\ell]$. The $\mathrm{count}(x)$ operation returns $\min_{i\in[\ell]} T_i[h_i(x)]$, which is shown to provide a good approximation for the frequency of x. The Count-Min data structure is used in many streaming algorithms including heavy-hitters, range queries, quantile estimation, and more. We provide an efficient implementation of a Count-Min sketch data structure, and show:

▶ **Theorem 6** (Neural Implementation of Count-Min Sketch). *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$ there exists a network $\mathcal{N}$ with $\log n$ input neurons, $O(1/\epsilon \cdot \mathrm{poly}(\log m, \log 1/\delta)))$ auxiliary neurons, and $\widetilde{O}(1)$ persistence time that implements a Count-Min sketch with approximation ration $(1+\epsilon)$ and success probability $1-\delta$, for an input stream of length at most m.*

Our neural implementation of the Count-Min sketch can immediately be used to give, e.g., a simple neural approximate heavy-hitters algorithm, which returns TRUE if a presented item has frequency $\geq m/k$ in a data-stream for some integer k, and FALSE if it has frequency $\leq (1-\epsilon)m/k$. Setting $\epsilon' = O(\epsilon/k)$, a $\mathrm{count}(x)$ query will return a frequency estimate $\geq m/k$ for any true heavy-hitter x and $\leq m/k$ for any x with frequency $\leq (1-\epsilon)m/k$. By keeping a counter for m using $O(\log m)$ neurons and performing a comparison operation with the output of $\mathrm{count}(x)$, we can thus solve the heavy hitters problem. Other applications of Count-Min sketch require more complex processing of the data structure's output. To illustrate how this processing can be implemented efficiently in an SNN, we detail one such application, to median approximation.

Approximate median. One of the most fundamental statistical measures of a data-stream is its quantile. The 1/2-quantile known as the median, attracts most attention in the streaming literature [37, 36, 5, 7]. Its non-linearity nature makes it considerably harder to maintain compared to its linear cousin, the mean. As in many other streaming problems, the exact computation of the median requires linear space both in the streaming and in the SNN setting (as will be discussed later on). This motivates the study of the relaxed $(1+\epsilon)$ approximation task. In the latter, the algorithm is allowed to output an item j provided that the total number of items with value at most j is in $[m/2 - \epsilon m, m/2 + \epsilon m]$.

Cormode and Muthukrishnan [10] presented an elegant streaming algorithm for this problem using a space of $\widetilde{O}(1/\epsilon)$ bits. The algorithm is based on the Count-Min sketch data structure, combined with a dyadic decomposition technique that is used in a number of other streaming algorithms. One of our key technical algorithmic contributions is in providing an efficient neural implementation of this algorithm.

▶ **Theorem 7** (Approximate Median). *For every $n, m \in \mathbb{N}$ and $\epsilon, \delta \in (0,1)$, there exists a neural network $\mathcal{N}_{n,m}$ solving the ϵ-approximate median problem using $O(1/\epsilon \cdot \mathrm{poly}(\log m, \log n, \log 1/\delta))$ auxiliary neurons and persistence time $\widetilde{O}(1)$ with probability $1-\delta$.*

1.1.2 Streaming Lower Bounds Yield SNN Lower Bounds

Our second contribution focuses on Question 2, showing that space-efficient SNNs can be translated into space-efficient streaming algorithms, and thus that lower bounds in the streaming model imply lower bounds in the neural setting. The underlying intuition for this transformation is based on the following observation.

▶ **Observation 8.** *A spiking neural network with deterministic edge weights and n input neurons and S non-input neurons can be simulated by a streaming algorithm using S bits of space.*

In the SNN model, the spiking behavior of neurons in a given round depends only on the firing states of their incoming neighbors in the previous round. Thus, to simulate the behavior of the network as one pass over the data-stream, it is sufficient to maintain the firing states of all non-input neurons in the network, thus storing S bits of information. When the edge weights of the network are randomly sampled such a small-space simulation becomes more involved. The explicit storage of all the edge weights might be too costly since there can be $\Omega(nS + S^2)$ edges in a network with n input neurons and S non-inputs. Nevertheless, we show that a small-space simulation is still possible using a pseudorandom number generator, if we pay an additive logarithmic overhead in the length of the stream and universe size.

▶ **Theorem 9.** *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and $\text{poly}(n)$ persistence time can be simulated over a data-stream of length m using a total space of $O(S + \log(nm))$. The success guarantee of the simulation is $1 - 1/\text{poly}(n, m)$.*

Theorem 9 is a powerful tool, since it lets us apply any streaming space lower bound (of which there are many) to give an SNN lower bound, with a loss of an $O(\log(nm))$ factor. In some cases, we can avoid this loss by more directly considering the lower-bound technique. This is obtained when the streaming lower bounds are derived via a reduction to communication complexity with shared randomness that can be applied in the SNN setting with no loss. For example, using this tighter approach we show that our neural network for the distinct elements problem is nearly space-optimal (see the full version for details).

1.2 Preliminaries

Spiking neural networks. A deterministic neuron u is modeled by a *deterministic* threshold gate. Letting $b(u)$ to be the threshold value of u, then u outputs 1 if the weighted sum of its incoming neighbors exceeds $b(u)$. A *spiking neuron* is modeled by a probabilistic threshold gate, which fires with a sigmoidal probability that depends on the difference between its weighted incoming sum and $b(u)$.

A *Neural Network* (NN) $\mathcal{N} = \langle X, Z, Y, w, b\rangle$ consists of n input neurons $X = \{x_1, \ldots, x_n\}$, m output neurons $Y = \{y_1, \ldots, y_m\}$, and k auxiliary neurons $Z = \{z_1, ..., z_k\}$. In spiking neural networks (SNN), the neurons can be either deterministic threshold gates or probabilistic threshold gates. The directed weighted synaptic connections between $V = X \cup Z \cup Y$ are described by the weight function $w : V \times V \to \mathbb{R}$. A weight $w(u, v) = 0$ indicates that a connection is not present between neurons u and v. Finally, for any neuron v, the value $b(v) \in \mathbb{R}$ is the bias value (activation threshold). Additionally, each neuron is either inhibitory or excitatory: if v is inhibitory, then $w(v, u) \leq 0$ and if v is excitatory, then $w(v, u) \geq 0$ for every u. This restriction arises from the biological structure of the neurons.

Network dynamics. The network evolves in discrete, synchronous rounds as a Markov chain. The firing status of every neuron u in round τ denoted as $\sigma_\tau(u)$, depends on the firing status of its neighbors in round $\tau - 1$, via a standard sigmoid function. For each neuron u, and each round $\tau \geq 0$, let $\sigma_\tau(u) = 1$ if u fires (i.e., generates a spike) in round τ. For every neuron u and every round $\tau \geq 1$, let $\text{pot}(u, \tau) = \sum_{v \in V} w(v, u) \cdot \sigma_{\tau-1}(v) - b(u)$ denote the membrane potential at round τ. A deterministic threshold gate u fires in round τ iff $\text{pot}(u, \tau) \geq 0$. A probabilistic threshold gate fires with a probability that depends on $\text{pot}(u, \tau)$. All our network constructions in this work use deterministic threshold-gates, and the randomness of the network comes from the randomized selection of the edge weights.

Neural networks for data-stream problems. A data-stream problem is defined by a relation $P_n \subset \mathbb{Z}^n \times \mathbb{Z}$. The length of the stream is upper bounded by some integer m. Each data-item is represented by a binary vector of length n. A value $i \in [1, n]$ is represented by having the i^{th} input neuron fire while all other input neurons are idle. Each input is presented for some persistence time, at the end of which the output neurons of the network encode (in binary) the evaluation of a given relation over the current stream. To avoid cumbersome notation, we may assume that m and n are powers of 2.

1.3 Basic Tools

Our constructions are based on the following neural network modules.

Neural timers and counters. For a given time parameter t, a neural timer $\mathcal{NT}_t$ is an SNN network that consists of an input neuron x, an output neuron y, and additional auxiliary neurons. The network satisfies that in every round τ, y fires in round τ iff x fires at some round τ' for $\tau' \in [\tau - t, \tau]$. It is fairly trivial to design a neural timer network with $O(t)$ auxiliary neurons. [15] presented a construction of a considerably more succinct network $\mathcal{NT}_t$ with only $O(\log t)$ neurons. In the related setting of neural counting, the network is required to encode the *number* of firing events of its input neuron within a given time window. Specifically, given time parameter t, a *neural counter network* $\mathcal{NC}_t$ has a single input neuron x, and $\lceil \log t \rceil$ output neurons that encode the number of firing events of x within a span of t rounds.

▶ **Fact 10.** *[30, 15] For every integer parameter t, there exist (i) a neural timer network $\mathcal{NT}_t$ with $O(\log t)$ neurons, and (ii) a neural counter network $\mathcal{NC}_t$ with $O(\log t)$ auxiliary neurons, such that for every round i, the output neurons encode f_i by round $i + O(\log t)$ where f_i is the number of firing events up to round i. Both networks $\mathcal{NT}_t$ and $\mathcal{NC}_t$ are deterministic.*

We next describe *new* tools introduced in this work which will be heavily used in our constructions. Missing proofs are deferred to the full version of the paper.

Potential encoding. Our SNN constructions are based on a module that encodes the potential p of a given neuron x by its binary representation using $\log p$ neurons. We will use this modules in the constructions of Theorem 3 and Lemma 14.

▶ **Lemma 11.** *Let x be a deterministic neuron such that $\text{pot}(x, t') \leq 2^\ell$ for every $t' \in [t, t + O(\ell)]$ for some integer $\ell \in \mathbb{N}_{>0}$. There exists a deterministic network $\text{POT}_\ell(x)$ which uses ℓ identical copies of x (the same input and bias), 2ℓ auxiliary neurons, and ℓ output neurons $y_0 \dots y_{\ell-1}$ that encodes $\text{pot}(x, t)$ in a binary form within $O(\ell)$ rounds.*

Implementing pairwise-independent hash functions. Many streaming algorithms in the insertion only model are based on the notion of pairwise independent hash functions.

▶ **Definition 12** (Pairwise Independence Hash Functions). *A family of functions* $\mathcal{H} : [a] \to [b]$ *is* pairwise independent *if for every* $x_1 \neq x_2 \in [a]$ *and* $y_1, y_2 \in [b]$*, we have:* $\Pr[h(x_1) = y_1 \text{ and } h(x_2) = y_2] = 1/b^2$.

For ease of notation, assume that a, b are powers of 2.

▶ **Definition 13** (Pairwise Independence Hash SNN). *Given two integers* a, b*, a* pairwise independent hash network $\mathcal{N}_{a,b}$ *is an SNN with an input layer of* $\log a$ *neurons, an output layer of* $\log b$ *neurons, and a set of* s *auxiliary spiking neurons. For every input value* x *presented at round* t*, let* $\mathcal{N}(x)$ *be the value of the output layer after* $\tau_{a,b}$ *rounds. Then, for every* $x \neq x' \in [a]$*, it holds that* $\Pr[\mathcal{N}(x) = \mathcal{N}(x')] = 1/b$.

We show a neural network implementation of a pairwise independent hash function using the construction of pairwise hash function by [4].

▶ **Lemma 14** (Neural Implementation of Pairwise Indep. Hash Function). *There exists a pairwise independent hash network* $\mathcal{N}_{a,b}$ *with* $s = O(\log b \cdot \log\log a)$ *auxiliary neurons that computes the output value of each input within* $O(\log\log a)$ *rounds (persistence of the input neurons).*

2 Linear Sketching

A linear sketching algorithm is a streaming algorithm in which the state of the algorithm at time t is a linear function of the updates seen up to time t. We start with a formal definition.

▶ **Definition 15** (Linear Sketching Algorithm, [23]). *A linear sketching algorithm* $\mathcal{L}$ *gives a method for processing a vector* $\bar{x} \in \mathbb{R}^n$*. The algorithm is characterized by a (typically randomized) sketch matrix* $A \in \mathbb{R}^{r \times n}$*, and by a possibly randomized decoding function* $f : \mathbb{R}^r \to O$ *where* O *is some output domain. Algorithm* $\mathcal{L}$ *is executed by first computing* $A \cdot \bar{x}$ *and then outputting* $f(A \cdot \bar{x})$*. Note that* f *only takes* $A \cdot \bar{x}$ *as input,* f *cannot depend on* A *in any other way, e.g. it cannot share randomness with* A.

Linear sketching algorithms provide the state-of-the-art space bounds for a large collection of problems in the turnstile model.

The challenge and our approach. Throughout we assume the sketching matrix is integral, i.e., $A \in \mathbb{Z}^{r \times n}$, which captures most of the classic implementations in the turnstile model. We start by describing a straw man approach for computing the value $A\bar{x}$ in the neural setting: Take a single-layer neural network with an input layer of length $n+1$ and an output layer of length r. Specifically, the input layer contains n neurons $x_1, \ldots, x_n$ that represent the absolute value of the update, and an additional *sign* neuron that indicates the sign of the update. For example, an update vector $[0, 0, -1, 0]$ is represented by letting $x_3 = 1$, $s = 1$ and $x_1, x_2, x_4 = 0$. The output layer is defined by r output neurons $y_1, \ldots, y_r$. The edge weights are specified by the matrix A where $w(x_j, y_i) = A_{i,j}$. It is then easy to verify that the weighted sum of the incoming neighbors of each neuron y_j (i.e., its potential) is the value of the j^{th} bit in $A\bar{x}$.

This naive description fails for various reasons. First, from a biological perspective, each input neuron can be either inhibitory or excitatory. This implies that the sign of the outgoing edge weights of a given neuron must be either a plus (excitatory) or a minus (inhibitory).

Mathematically, this requires the sketch matrix A to be sign-consistent (i.e., the sign of all entries in a given raw are either a plus or a minus). However, in general, the given sketch matrix might not be sign-consistent. The second technicality is that the neurons $y_1, \ldots, y_n$ have a *binary* output (either firing or not) rather then an *integer* value. The third aspect to take into account is concerned with the update mechanism. Specifically, given a stream of data items, one should make sure that each data item would be processed exactly *once* by the network. This requires a more delicate update mechanism.

In the high-level, we handle the sign-consistency challenge by dividing the sketch matrix A into a non-negative matrix A^+ and a non-positive matrix A^- where $A = A^+ - A^-$. Then, given a new update $(\bar{x}, s)$, the network computes $A\bar{x}$ and $-A\bar{x}$ using $A^+\bar{x}$ and $A^-\bar{x}$. The final output $A\bar{x}$ is computed by using these values combined with the sign neuron s. To handle the second challenge, we use the module of Lemma 11 to translate the *potential* of each output neuron y_j (corresponding to the j'th bit in the sketch) into its binary representation. The output layer consists of $O(r \log n)$ output neurons that encode the value of the current r-length sketch. The complete implementation details are given to the full version.

Due the space consideration, the neural implementation for the *distinct-element problem* (Proof of Theorem 4) is deferred to the full version of the paper. In this proof we also demonstrate how to translate the streaming lower bound into a matching space lower bound for the neural setting.

3 Median Approximation

Before presenting the neural computation of the approximate median, we describe the neural implementation of the Count-Min Sketch and prove Theorem 6.

3.1 A Neural Implementation of Count-Min Sketch

We follow the streaming implementation of Count-Min by [10] described as follows. The algorithm maintains a data structure that consists of $\ell = O(\log 1/\delta)$ hash tables $T_1, \ldots T_\ell$, each with $b = O(1/\epsilon)$ bins, and each table T_i is indexed using a different pairwise-independent hash function h_i (i.e., the output domain of h_i is $\{0,1\}^{\log b}$). The operation inc(x) increases the value in each bin $T_i[h_i(x)]$ for every $i \in [\ell]$. The count(x) operation returns the value $\min_{i\in[\ell]} T_i[h_i(x)]$.

▶ **Fact 16** ([10]). $\Pr[\text{count}(x) \notin (f(x), f(x) + O(m/b))] \leq 1/2^{\Omega(\ell)}$ *where* $f(x)$ *is actual frequency of* x *in the stream of length* m.

▶ **Definition 17** (Neural Count-Min Sketch). *Given parameters* $\epsilon, \delta > 0$, *a neural Count-Min sketch network* $\mathcal{N}_{\epsilon,\delta}$ *has an input layer of* $\log n + 1$ *neurons denoted as* $a, x_1, \ldots x_{\log n}$, *an output layer of* $\log m$ *neurons* $y_1, \ldots y_{\log m}$, *and a set of* s *auxiliary neurons. The neurons* $x_1, \ldots x_{\log n}$ *encode the binary representation of an element* $x \in [n]$ *and the neuron* a *indicates whether this is an* inc *or* count *operation, where* $a = 1$ *indicates an* inc *operation. For every fixed input value* $x = (x_1, \ldots x_{\log n})$ *presented at round* t *and* $a = 0$ *(i.e., a* count *operation), let* $\mathcal{N}_{\delta,\epsilon}(x)$ *be the value encoded in binary by the output layer* $y_1, \ldots y_{\log m}$ *in round* $t + \tau_{n,m}$. *It holds that* $\Pr[\mathcal{N}_{\delta,\epsilon}(x) \notin (f(x), f(x) + O(\epsilon m'))] \leq \delta$, *where* $m' \leq m$ *is the stream length by round* t *and* $f(x)$ *is the current frequency of* x.

We first describe the network construction to support the inc(x) operation. Then we explain the remaining network details for implementing a count(x) operation.

Supporting inc(x) operation. The network contains $\ell = O(\log 1/\delta)$ sub-networks $\mathcal{H}^1_{n,b}, \ldots, \mathcal{H}^\ell_{n,b}$ each implements a pairwise independent hash function $h_i : \{0,1\}^{\log n} \to \{0,1\}^{\log b}$ using Lemma 14. The output vector of each network $\mathcal{H}^i_{\log n,b}$ is denoted by $\bar{h}_i$ for every $i \in \{1, \ldots, \ell\}$. Every $\bar{h}_i$ has an inhibitory copy $\bar{h}'_i$.

For each sub-networks $\mathcal{H}^i_{\log n,b}$, and for every value $j \in \{1, \ldots, b\}$, the network contains a counter sub-network that counts the number of data-items x in the stream that satisfies $h_i(x) = j$. Every counter network is implemented by a neural-counter network from Fact 10 with time parameter $t = m$. Let $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,b}$ be the neural counter networks corresponding to the i^{th} hash network $\mathcal{H}^i_{\log n,b}$. The counter $\mathcal{C}_{i,j}$ is updated based on the values of the output neurons $\bar{h}_i$ as follows. For every counter $\mathcal{C}_{i,j}$ the network contains an index neuron $c_{i,j}$ with input from $\bar{h}_i$ and $\overline{h'}_i$ which fires only if[1] $\text{dec}(\bar{h}_i) = j$. The input to the counter $\mathcal{C}_{i,j}$ denoted as $e_{i,j}$ is an AND gate between the input neuron a and the index neuron $c_{i,j}$, firing in inc(x) operations where $h_i(x) = j$. To make sure the counter is incremented once per inc(x) operation, the network contains an inhibitory neuron denoted as $e'_{i,j}$ which has the same incoming edges and weights as $e_{i,j}$, that inhibits the neurons $e_{i,j}$, $c_{i,j}$ and a. This guarantees that $e_{i,j}$ would be active for exactly *one* round per inc(x) operation.

Supporting count(x) operation. To support a count(x) operation, for each counter $\mathcal{C}_{i,j}$, the network includes $\log m$ neurons $\bar{s}_{i,j} = s^1_{i,j}, \ldots s^{\log m}_{i,j}$ which hold the value stored in the counter $\mathcal{C}_{i,j}$ such that $h_i(x) = j$. Each neuron $s^k_{i,j}$ is an AND gate of the index neuron $c_{i,j}$ and the j^{th} output neuron of $\mathcal{C}_{i,j}$. In addition, for every $i \in \{1, \ldots, \ell\}$ there are $\log m$ neurons $\bar{g}_i = g_{i,1}, \ldots g_{i,\log m}$ where the j^{th} neuron $g_{i,j}$ is an OR gate of all the j^{th} neurons of the vectors $\bar{s}_{i,1}, \ldots, \bar{s}_{i,b}$. As a result, $\bar{g}_i$ encodes the value stored in $h_i(x)$. Finally, the output value is set to be the *minimum value* of $\text{dec}(\bar{g}_1), \ldots, \text{dec}(\bar{g}_\ell)$ using the minimum computation network of [34]. The correctness analysis is deferred to the full version of the paper.

3.2 Neural Computation of the Approximate Median

In this section, we present our main technically involved algorithmic result for computing an estimate for the median of the data-stream.

▶ **Definition 18** (Approximate Median). *Given $\epsilon, \delta \in (0,1)$ and a stream $\mathcal{S} = \{x_1, x_2, \ldots x_m\}$ with each $x_i \in [n]$, in the* approximate median *problem, it is required to output an element $x_j \in \mathcal{S}$ whose rank is $m/2 \pm \epsilon m$ with probability at least $1 - \delta$.*

For ease of notation, assume that n is power of 2. Our neural solution is based on the streaming algorithm of [10], that uses $\widetilde{O}(1/\epsilon)$ space. Up to the logarithmic terms, this space-bound is known to be optimal [24].

▶ **Fact 19** (Theorem 5 [10]). *For every $\epsilon, \delta \in (0,1)$, there exists a randomized streaming algorithm for computing the ϵ-approximate median with probability $1 - \delta$ and $\widetilde{O}(1/\epsilon)$ space.*

We start by providing a high-level exposition of this streaming algorithm, and then explain its implementation in the neural setting. The latter turns out to be quite involved, yet demonstrating the expressive power of SNN networks.

[1] For implementation reasons, verifying that $\text{dec}(\bar{h}_i) = j$ requires input from both $\bar{h}_i$ and $\overline{h'}_i$.

A high-level description of the streaming algorithm. The algorithm is based on applying a binary search over *range queries* which, roughly speaking, compute the frequency of the elements in a given range.

▶ **Definition 20** (Range Queries). *Given a data-stream of numbers $\mathcal{S} = \{x_1, \ldots, x_m\}$ with each $x_i \in [n]$, a range query receives a range of number $[a, b] \subseteq [1, n]$ and returns the frequency of the items $\{a, a+1, \ldots, b\}$ in the stream $\mathcal{S}$.*

To support range queries with small space, the algorithm maintains $\log n$ data structures of Count-Min sketch, for each of the $\log n$ *dyadic intervals* of $[n]$.

▶ **Definition 21** (Dyadic Intervals). *The* dyadic intervals *of the set $[n]$ are a collection of $\log n$ partitions of n, $\mathcal{I}_1 \ldots, \mathcal{I}_{\log n}$ such that*

$$\begin{aligned}
\mathcal{I}_0 &= \{\{1\}, \{2\}, \{3\}, \ldots, \{n\}\} \\
\mathcal{I}_1 &= \{\{1,2\}, \{3,4\}, \{5,6\}, \ldots, \{n-1, n\}\} \\
\mathcal{I}_2 &= \{\{1,2,3,4\}, \{5,6,7,8\}, \ldots, \{n-3, n-2, n-1, n\}\} \\
&\ldots \\
\mathcal{I}_{\log n} &= \{\{1, 2 \ldots n\}\}
\end{aligned}$$

Note that every range $[i, j] \subseteq [n]$ can be written as a union of at most $\log n$ sets from the dyadic intervals. Hence, by introducing $\log n$ Count-Min data structures with parameters $\delta' = \log(\log n/\delta)$ and $\epsilon' = \epsilon/\log n$ for dyadic-intervals of $[n]$, we can answer range queries within an additive error of $m \cdot \epsilon$ with probability $1 - \delta$. The approximated median is obtained by employing a Binary search over the range queries [2].

▶ **Definition 22** (SNN for the Approximate Median Problem). *Given two integers n, m and additional parameters $\epsilon, \delta \in (0, 1)$, an approximate-median network $\mathcal{N}_{n,m}$ has an input layer of $n+1$ neurons, an output layer of $\log n$ neurons and a set of s auxiliary neurons. The input neurons are denoted as $(a, x_1, \ldots, x_n)$ where the neuron a indicates whether this is a median query or an insertion operation. When the input layer represents a median query, the neuron a fires and the neurons $x_1, \ldots x_n$ are idle. For every round t, let $\mathcal{S}_t = \{a_1, a_2, \ldots a_t\}$ be the data-stream presented as input to the network by round t. For any median-query presented in round t, by round $t + \tau_{n,m}$ the output layer encodes an element $y \in \mathcal{S}_t$ whose rank in $\mathcal{S}_t$ is $t/2 \pm \epsilon t$ with probability at least $1 - \delta$.*

The challenge: The crux of the streaming algorithm is based on a binary search over range queries. A-priori, it is unclear how to implement such a search using a poly-logarithmic number of neurons. Specifically, the (implicit) decision tree that governs the binary search has a linear size. Since the neural network (unlike the streaming algorithm) has to hard-wire the algorithm description, the explicit encoding of the search tree leads to a linear space solution. Our key contribution is in showing a succinct network construction that simulates the binary search of the streaming algorithm using a nearly matching space bound.

We next provide a high-level description of the network. Recall that the type of the operation is represented by the input neuron a, where $a = 1$ represents a median query.

[2] The same algorithm can be applied for any quantile estimation.

Supporting an insertion operation. In the high level, the network contains 3 parts (1) a set of $\log n$ neurons that encode the inserted element in its binary form, (2) a neural counter that counts the length of the current stream, and (3) $\log n$ Count-Min sketch sub-networks that maintain the frequencies of the $\log n$ dyadic intervals of $[n]$.

1. The n-length input vector $\bar{x}$ is connected to $\log n$ neurons $\bar{x}' = (x'_1, \ldots, x'_{\log n})$ such that $\bar{x}'$ encodes the binary representation[3] of the element presented in the input neurons $\bar{x}$.
2. The network contains a counter sub-networks $\mathcal{NC}_m$ for counting the number of data-items inserted so far (i.e., the current length of the stream). The counter is implemented by a neural-counter network from Fact 10 with time parameter $t = m$. The input neuron to the $\mathcal{NC}_m$ sub-network denoted as a' is an OR gate of the input neurons $\bar{x}$. To make sure the counter is incremented once per insertion operation, the network contains an inhibitory copy of a' denoted as r', which inhibits a' and the neurons $\bar{x}$. As a result, the input neuron a' will be active for exactly *one* round per insertion operation.
3. The network contains $\log n$ sub-networks $\mathcal{C}_1, \ldots \mathcal{C}_{\log n}$ each implements a Count-Min sketch with parameters n, m and $\epsilon' = O(\epsilon/\log n)$, $\delta' = O(\delta/\log n)$ using Theorem 6. For each Count-Min sketch sub-networks $\mathcal{C}_i$, let $\bar{z}_i = (z_{i,1}, \ldots z_{i,\log n})$ and b_i be its input layer, where the neuron b_i indicates whether the operation is inc or count. The neuron b_i is an OR gate of the neurons in $\bar{x}$.
   The input neurons $\bar{z}_i$ are connected to the binary representation of the input $\bar{x}'$ in the following manner. For every $i \in \{1, \ldots, \log n\}$ and every $j \geq i$, the neuron x'_j is connected to the neuron $z_{i,j}$. In addition, for every $j < i$ the neuron $z_{i,j}$ serves as an OR gate between the neurons of $\bar{x}'$. The neurons b_i, $\bar{z}_i$ are equipped with self-loops. The Count-Min sketch sub-networks are then modified such that these neurons will be inhibited once the computation is complete (by the inhibitory neurons $e'_{i,j}$ of each sub-networks respectively).

Supporting a median query. Given a median query, the network computes the approximate median by employing at most $\log n$ steps of binary search. In every step[4] $i \in \{\log n, \ldots, 1\}$, the network obtains a current candidate for the median denoted by χ_i. Initially, $\chi_{\log n} = n/2$. Each χ_i would be provided as input for the i^{th} Count-Min sketch $\mathcal{C}_i$. The output neurons of $\mathcal{C}_i$ would then define the next candidate χ_{i-1}. Specifically, depending on the rank estimation of χ_i, the network defines the new search range. The width of the search range would be cut by a factor 2 in every step i. Consequently, the algorithm will be using the Count-Min sketch $\mathcal{C}_{i-1}$ which is defined over a partitioning $\mathcal{I}_{i-1}$ in which each set is smaller by factor 2 compared to $\mathcal{I}_i$. We now describe these steps in more details.

1. For every $i \in \{\log n, \ldots, 1\}$ the network contains an additional Count-Min sub-networks $\mathcal{C}'_i$ which counts the frequencies of the data-elements (similar to $\mathcal{C}_1$). This additional Count-Min sub-networks will be useful in a scenario where the median item j^* has a very large frequency. In such a case, the frequency of the range $[1, j^*]$ is too large and the frequency of $[1, j^* - 1]$ is too small. This special case would be handled using the $\mathcal{C}'_i$ sub-networks.
2. For every $i \in \{\log n, \ldots, 1\}$ the network contains three *comparison* neurons s_i, g_i, e_i (corresponding to *smaller, greater* or *equal*). These neurons receive their input from the output neurons of the counters $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_i$, and from the the output of the neural

[3] As discussed in the introduction our solution supports both types of input formats: $\log n$-bits of the binary representation or an n-length vector with one active entry.

[4] It is convenient to count the steps in a backward manner, as in the i^{th} step the network will access the i^{th} Min-Sketch module $\mathcal{C}_i$.

counter $\mathcal{NC}_m$. Let $\chi_i = \text{dec}(\bar{z}_i)$, this value would correspond to the median candidate at phase i of the binary-search. The firing states of the comparison neurons would be determined as follows. The neuron g_i would fire if the frequency estimation of $[1, \chi_i]$ is greater than $m'/2 + \epsilon/2m'$. The neuron s_i would fire if frequency estimation of $[1, \chi_i]$ is smaller than $m'/2 - \epsilon/2m'$. Finally, e_i would fire if the frequency estimation of $[1, \chi_i]$ is in the range $(m'/2 - \epsilon/2m', m'/2 + \epsilon/2)$.

3. For $i \in \{\log n, \ldots, 1\}$, every two consecutive sub-networks $\mathcal{C}_{i+1}$ and $\mathcal{C}_i$ are connected in a way that guarantees the following. Let χ_{i+1} be median candidate at phase $i+1$ of the binary search (i.e., that was fed as input to $\mathcal{C}_{i+1}$). Let $\text{freq}([x, y])$ be the estimated frequency of the range $[x, y]$ obtained by the Count-Min sketch networks $\mathcal{C}_{\log n}, \ldots, \mathcal{C}_{i+1}$. Then candidate χ_i is defined as:

$$\chi_i = \begin{cases} \chi_{i+1} - 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) > m'/2 + \epsilon/2m' \\ \chi_{i+1} + 2^{i-1}, & \text{if } \text{freq}([1, \chi_{i+1}]) < m'/2 - \epsilon/2m' \ . \end{cases}$$

In the remaining case where $\text{freq}([1, \chi_{i+1}]) \in [m'/2 \pm \epsilon/2m']$, the candidate χ_{i+1} is returned as the output result. Its value will be encoded by the output neurons of the network. The complete description and its analysis is deferred to the full version of the paper.

4 Streaming Lower Bounds Yield SNN Lower Bounds

We conclude by addressing Question 2, giving a generic reduction that lets us simulate a space-efficient SNN with a space-efficient neural network. This establishes a tight connection between the two models – any streaming space lower bound yields a near-matching neural-space lower bound. Missing proofs of this section appear in the full version of the paper.

Complexity classes in the SNN model. For integer parameters n, m, S, let $\mathcal{SNN}_{\text{det}}(n, m, S)$ be the set of all data-stream problems $P_{n,m}$ defined over universe $[n]$ and stream length at most m that are solvable by a deterministic SNN with (i) at most $O(S)$ non-input neurons (i.e., auxiliary and output neurons) and (ii) polynomially bounded edge weights (by n and m). Let $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S)$ be the class of all data-stream problems $P_{n,m}$ in $\mathcal{SNN}_{\text{det}}(n, m, S)$ whose network solution also have in addition a polynomial persistence time (in n and m). That is, the problems in $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S)$ are solvable in polynomial-time by a deterministic SNN that has properties (i,ii).

We also consider the class of data-stream problems that are solvable by a randomized SNN. Let $\mathcal{SNN}_{\text{rand}}(n, m, S, \delta)$ be the set of all data-stream problems $P_{n,m}$ that are solvable by a randomized SNN with: (i) at most $O(S)$ non-input neurons, (ii) polynomially bounded edge weights, and (iii) $\leq \delta$ failure probability on any input. The class $\mathcal{SNN}^{\text{poly}}_{\text{rand}}(n, m, S, \delta)$ is a sub-class of $\mathcal{SNN}_{\text{rand}}(n, m, S, \delta)$ that requires also a polynomial persistence time.

Complexity classes in the streaming model. Let $\mathcal{ST}_{\text{det}}(n, m, S)$ be the class of all data-stream problems for which there exists a single-pass deterministic streaming algorithm for the problem using space $O(S)$ (potentially with exponentially large update time). Also, let $\mathcal{ST}_{\text{rand}}(n, m, S, \delta)$ be the class of all data-stream problems for which there exists a single-pass randomized streaming algorithm that solves the problem with failure probability $\leq \delta$ using space $O(S)$. One can also define the classes $\mathcal{ST}^{\text{poly}}_{\text{det}}(n, m, S)$ and $\mathcal{ST}^{\text{poly}}_{\text{rand}}(n, m, S, \delta)$ which require polynomial update time.

We start by showing that any deterministic SNN with space S for a given data-stream problem $P_{n,m}$ yields an S-space deterministic streaming algorithm for the problem.

▶ **Lemma 23.** *For every n, m, S, we have:*
$\mathcal{SNN}_{\text{det}}(n, m, S) \subseteq \mathcal{ST}_{\text{det}}(n, m, S)$ *and* $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n, m, S)$.

Proof. Fix the parameters n, m, S, and consider a problem $\Pi \in \mathcal{SNN}_{\text{det}}(n, m, S)$. Let $\mathcal{N}$ be the SNN for the problem Π. Thus $\mathcal{N}$ has S auxiliary and output neurons. We now describe a streaming algorithm for Π that uses space S. The algorithm traverses the stream and feeds each item as an input to the network $\mathcal{N}$ (with sufficient large persistence time). Importantly, when considering the subsequent input item, the streaming algorithm only keeps the current firing states of the S auxiliary and output neurons. The correctness follows immediately by the correctness of the network $\mathcal{N}$. The space complexity is S bits corresponding to the firing states of the (non-input) neurons in $\mathcal{N}$. The proof that $\mathcal{SNN}^{\text{poly}}_{\text{det}}(n, m, S) \subseteq \mathcal{ST}^{\text{poly}}_{\text{det}}(n, m, S)$ is analogous since the update time of the streaming algorithm is polynomial in the network size and the persistence time of the network. ◀

Pseudorandomness for neural networks. Our next goal is to simulate space-efficient randomized SNNs for data-stream problems with small-efficient streaming algorithms. The main barrier arises in the case where the edge weights of the network $\mathcal{N}$ are chosen randomly according to some distribution. Since an S-space network with n input neurons might have $\Omega(Sn + S^2)$ edges, the explicit specification of the edge weights is too costly for our purposes.

To overcome this barrier, we will use Nisan-Wigderson type pseudorandom generators [39], which fool circuits of a given size, defined as follows:

▶ **Definition 24** (NW-type *PRG*s.)**.** *A function* $\text{NWPRG}\colon \{0,1\}^{d(n)} \to \{0,1\}^n$ *is an* NW-type *PRG* against circuits of size $t(n)$ *if it is (i) computable in time* $2^{O(d(n))}$ *and (ii) any circuit C of size at most $t(n)$ distinguishes $U \leftarrow \{0,1\}^n$ from* $\text{NWPRG}(s)$*, where $s \leftarrow \{0,1\}^{d(n)}$, with advantage at most $1/t(n)$.*

▶ **Theorem 25.** *[17] Assume there exists a function in $E = \text{DTIME}(2^{O(n)})$ with circuit complexity $2^{\Omega(n)}$. Then, for any polynomial $t(\cdot)$, there exists a NW-type generator* $\text{NWPRG}\colon \{0,1\}^{d(n)} \to \{0,1\}^n$ *against circuits of size $t(n)$, where $d(n) = O(\log n)$.*

▶ **Lemma 26.** *If one does not restrict the running time of the PRG (and allows it to be computable by a non-uniform circuit), then a version of Theorem 25 holds* unconditionally*.*

Since an SNN with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and polynomial persistence time can be computed in polynomial time, we have the following:

▶ **Lemma 27.** *Any SNN $\mathcal{N}$ with n input neurons, S non-input neurons for $S = \text{poly}(n)$, and persistence time* $\text{poly}(n)$ *in an m-length stream can be simulated using a total space of $O(S + \log(nm))$. The success guarantee of the simulation is* $1 - 1/\text{poly}(n, m)$*.*

Proof. Consider a (centralized, offline) algorithm that given an ordered stream of length $m' \leq m$ of elements in $[1, n]$ evaluates the output of the network $\mathcal{N}$ on that stream. This algorithm can be implemented in time $\text{poly}(n, m)$ and thus there exists a circuit of size $M = \text{poly}(n, m)$ that implements this algorithm. Our goal is to simulate this circuit using a random seed of length $O(\log(nm))$ while reducing the success guarantee by an additive term of $1/\text{poly}(n, m)$. To do that, we will use the PRG construction of Lemma 26 that given a random seed of size $O(\log(nm))$ fools the family of all circuits of size at most M with probability $1 - 1/\text{poly}(M)$.

We now describe how to simulate $\mathcal{N}$ using $O(\log(nm) + S)$ space. We store a seed of $O(\log(nm))$ random bits R and the current firing states of all non-input neurons in $\mathcal{N}$. Then, as we traverse the stream, for every data-item in the stream, the algorithm first applies

the NWPRG function on the random seed R and outputs $\text{poly}(n, m)$ coins that are used to define the edge weights of $\mathcal{N}$, as well as to simulate the spiking decisions of the S neurons in the i^{th} step. The evaluation time of this NWPRG function (i.e., outputting each coin) might be exponential. However, we will only store one pseudorandom coin at a time, when its value is required as part of an update. By knowing the states of the S non-input neurons from the previous step, and using the coins to determine the edge weights one at a time (requiring $O(\log n)$ space as the edge weights are polynomially bounded) one can simulate the firing pattern of the neurons in the given step. When the neurons are probabilistic threshold gates, the pseudorandom coins are also used to simulate the firing decisions of these neurons. While each firing probability is real-valued, it can be rounded to $1/\text{poly}(n, m)$ accuracy (and hence determined by $O(\log(nm))$ random coins) without changing the probability of any output configuration by more than $1/\text{poly}(n, m)$. The step ends with the computation of the firing states of all non-input neurons, which are stored for the next step. The success guarantee of using these coins rather than fresh random coins is decreased by an additive term of $1/\text{poly}(n, m)$. ◀

Lemma 27 implies that any randomized SNN with space S that solves a streaming problem $P_{n,m}$ with probability $1 - \delta$ in polynomial time translates into a randomized streaming algorithm for $P_{n,m}$ using space of $S + O(\log(nm))$. We therefore have:

▶ **Theorem 28.** $\mathcal{SNN}^{\text{poly}}_{\text{rand}}(n, m, S, \delta) \subseteq \mathcal{ST}_{\text{rand}}(n, m, S + O(\log(nm)), \delta + 1/\text{poly}(n, m))$.

A useful implication of Theorem 28 is that any space lower-bound in the streaming model immediately translates into space lower-bound for networks that have a polynomial persistence time on the input stream. See the full version for the precise formulation.

References

1 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

2 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, pages 1–10, 2002.

3 Jaroslaw Blasiok. Optimal streaming and tracking distinct elements with high probability. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2432–2448, 2018. doi:10.1137/1.9781611975031.156.

4 J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

5 Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

6 Philippe Chassaing and Lucas Gerin. Efficient estimation of the cardinality of large data sets. *arXiv preprint math/0701347*, 2007.

7 Zhiwei Chen and Aoqian Zhang. A survey of approximate quantile computation on large-scale data. *IEEE Access*, 8:34585–34597, 2020.

8 Chi-Ning Chou, Kai-Min Chung, and Chi-Jen Lu. On the algorithmic power of spiking neural networks. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

9 Graham Cormode, Mayur Datar, Piotr Indyk, and Shanmugavelayutham Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

10 Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

11 Sanjoy Dasgupta, Charles F Stevens, and Saket Navlakha. A neural algorithm for a fundamental computing problem. *Science*, 358(6364):793–796, 2017.

12 Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, pages 605–617, 2003.

13 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07)*, 2007. URL: `https://dmtcs.episciences.org/3545`.

14 Yael Hitron, Nancy A. Lynch, Cameron Musco, and Merav Parter. Random sketching, clustering, and short-term memory in spiking neural networks. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 23:1–23:31, 2020.

15 Yael Hitron and Merav Parter. Counting to ten with two fingers: Compressed counting with spiking neurons. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, pages 57:1–57:17, 2019.

16 Yael Hitron, Merav Parter, and Gur Perri. The computational cost of asynchronous neural communication. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, pages 48:1–48:47, 2020.

17 Russell Impagliazzo and Avi Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 220–229, 1997.

18 Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):307–323, 2006.

19 Piotr Indyk and Eric Price. K-median clustering, model-based compressive sensing, and sparse recovery for earth mover distance. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 627–636, 2011.

20 Piotr Indyk and David P. Woodruff. Tight lower bounds for the distinct elements problem. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 283–288, 2003.

21 John Kallaugher and Eric Price. Separations and equivalences between turnstile streaming and linear sketching. In *Symposium on Theory of Computing, STOC 2020*, 2020.

22 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52, 2010. `doi:10.1145/1807085.1807094`.

23 Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. Fast and space efficient spectral sparsification in dynamic streams. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1814–1833. SIAM, 2020.

24 Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78. IEEE, 2016.

25 Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

26 Robert A. Legenstein, Wolfgang Maass, Christos H. Papadimitriou, and Santosh S. Vempala. Long term memory and the densest k-subgraph problem. In *9th Innovations in Theoretical*

Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA, pages 57:1–57:15, 2018.

27 Yi Li, Huy L. Nguyen, and David P. Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 174–183, 2014. doi:10.1145/2591796.2591812.

28 Nancy Lynch, Cameron Musco, and Merav Parter. Computational tradeoffs in biological neural networks: Self-stabilizing winner-take-all networks. *Innovations in Theoretical Computer Science*, 2017.

29 Nancy Lynch, Cameron Musco, and Merav Parter. Spiking neural networks: An algorithmic perspective. In *5th Workshop on Biological Distributed Algorithms (BDA 2017)*, 2017.

30 Nancy Lynch and Mien Brabeeba Wang. Integrating temporal information to spatial information in a neural circuit. *arXiv preprint arXiv:1903.01217*, 2019.

31 Nancy A. Lynch, Cameron Musco, and Merav Parter. Neuro-ram unit with applications to similarity testing and compression in spiking neural networks. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 33:1–33:16, 2017.

32 Wolfgang Maass. On the computational power of noisy spiking neurons. In *Advances in neural information processing systems*, pages 211–217, 1996.

33 Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

34 Wolfgang Maass. On the computational power of winner-take-all. *Neural computation*, 12(11):2519–2535, 2000.

35 Wolfgang Maass, Christos H. Papadimitriou, Santosh S. Vempala, and Robert A. Legenstein. Brain computation: A computer science perspective. In *Computing and Software Science - State of the Art and Perspectives*, pages 184–199. Springer, 2019. doi:10.1007/978-3-319-91908-9_11.

36 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, 27(2):426–435, 1998.

37 J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

38 Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

39 Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

40 Christos H. Papadimitriou and Santosh S. Vempala. Random projection in the brain and computation with assemblies of neurons. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, pages 57:1–57:19, 2019.

41 Lili Su, Chia-Jung Chang, and Nancy Lynch. Spike-based winner-take-all computation: Fundamental limits and order-optimal circuits. *Neural Computation*, 31(12):2523–2561, 2019.

42 Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

43 Leslie G. Valiant. Capacity of neural networks for lifelong learning of composable tasks. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 367–378, 2017.

44 David P. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 167–175, 2004.