SoftSort: A Continuous Relaxation for the argsort Operator

Sebastian Prillo *1 Julian Martin Eisenschlos *2

Abstract
While sorting is an important procedure in computer science, the argsort operator - which takes as input a vector and returns its sorting permutation - has a discrete image and thus zero gradients almost everywhere. This prohibits end-to-end, gradient-based learning of models that rely on the argsort operator. A natural way to overcome this problem is to replace the argsort operator with a continuous relaxation. Recent work has shown a number of ways to do this, but the relaxations proposed so far are computationally complex. In this work we propose a simple continuous relaxation for the argsort operator which has the following qualities: it can be implemented in three lines of code, achieves state-of-the-art performance, is easy to reason about mathematically - substantially simplifying proofs - and is faster than competing approaches. We open source the code to reproduce all of the experiments and results.

1. Introduction
Gradient-based optimization lies at the core of the success of deep learning. However, many common operators have discrete images and thus exhibit zero gradients almost everywhere, making them unsuitable for gradient-based optimization. Examples include the Heaviside step function, the argmax operator, and - the central object of this work - the argsort operator. To overcome this limitation, continuous relaxations for these operators can be constructed. For example, the sigmoid function serves as a continuous relaxation for the Heaviside step function, and the softmax operator serves as a continuous relaxation for the argmax. These continuous relaxations have the crucial property that they provide meaningful gradients that can drive optimization. Because of this, operators such as the softmax are ubiquitous in deep learning. In this work we are concerned with continuous relaxations for the argsort operator.

Formally, we define the argsort operator as the mapping \( \text{argsort} : \mathbb{R}^n \rightarrow \mathcal{S}_n \) from \( n \)-dimensional real vectors \( s \in \mathbb{R}^n \) to the set of permutations over \( n \) elements \( \mathcal{S}_n \subseteq \{1, 2, \ldots, n\}^n \), where argsort \( (s) \) returns the permutation that sorts \( s \) in decreasing order. For example, for the input vector \( s = [9, 1, 5, 2]^T \) we have argsort \( (s) = [1, 3, 4, 2]^T \). If we let \( \mathcal{P}_n \subseteq \{0, 1\}^{n \times n} \subset \mathbb{R}^{n \times n} \) denote the set of permutation matrices of dimension \( n \), following (Grover et al., 2019) we can define, for a permutation \( \pi \in \mathcal{S}_n \), the permutation matrix \( P_\pi \in \mathcal{P}_n \) as:

\[
P_\pi[i, j] = \begin{cases} 1 & \text{if } j = \pi_i, \\ 0 & \text{otherwise} \end{cases}
\]

This is simply the one-hot representation of \( \pi \). Note that with these definitions, the mapping \( \text{sort} : \mathbb{R}^n \rightarrow \mathbb{R}^n \) that sorts \( s \) in decreasing order is \( \text{sort}(s) = P_{\text{argsort}(s)}s \). Also, if we let \( \mathbb{I}_n = [1, 2, \ldots, n]^T \), then the argsort operator can be recovered from \( P_{\text{argsort}(\cdot)} : \mathbb{R}^n \rightarrow \mathcal{P}_n \) by a simple matrix multiplication via

\[
\text{argsort}(s) = P_{\text{argsort}(s)}\mathbb{I}_n
\]

Because of this reduction from the argsort operator to the \( P_{\text{argsort}(\cdot)} \) operator, in this work, as in previous works (Mena et al., 2018; Grover et al., 2019; Cuturi et al., 2019), our strategy to derive a continuous relaxation for the argsort operator is to instead derive a continuous relaxation for the \( P_{\text{argsort}(\cdot)} \) operator. This is analogous to the way that the softmax operator relaxes the argmax operator.

The main contribution of this paper is the proposal of a family of simple continuous relaxations for the \( P_{\text{argsort}(\cdot)} \) operator, which we call SoftSort, and define as follows:

\[
\text{SoftSort}(\cdot)(s) = \text{softmax} \left( \frac{-d \left( \text{sort}(s) \mathbb{I}_n^T, \mathbb{I}_n s^T \right)}{\tau} \right)
\]

where the softmax operator is applied row-wise, \( d \) is any differentiable almost everywhere, semi–metric function of

---

* : Equal contribution 1University of California, Berkeley, California, USA 2Google Research, Zurich, Switzerland. Correspondence to: Sebastian Prillo <sprillo@berkeley.edu>, Julian Martin Eisenschlos <eisenjulian@google.com>.


---

*This is called the sort operator in (Grover et al., 2019). We adopt the more conventional naming.
Throughout this work we will predominantly use vectors to match the decreasing order of the scores \( s_i \) of vectors \( v_i \). To better illustrate the usefulness of the art in differentiable sorting by significantly simplifying the proof of these properties.

- \textbf{SoftSort} achieves state-of-the-art performance on multiple benchmarks that involve reordering elements.

- \textbf{SoftSort} shares the same desirable properties as the \textit{NeuralSort} operator \cite{grover2019neuralsort}. Namely, it is row-stochastic, converges to \( P_{\text{argsort}}(\cdot) \) in the limit of the temperature, and can be projected onto a permutation matrix using the row-wise argmax operator.

Therefore, the \textbf{SoftSort} operator advances the state of the art in differentiable sorting by significantly simplifying previous approaches. To better illustrate the usefulness of the mapping defined by \textbf{SoftSort}, we show in Figure 1 the result of applying the operator to soft-sort a sequence of vectors \( v_i \) \((1 \leq i \leq n)\) according to the order given by respective scores \( s_i \in \mathbb{R} \). Soft-sorting the \( v_i \) is achieved by multiplying to the left by \textbf{SoftSort}(\( s \)).

The code and experiments are available at \url{https://github.com/sprillo/softsort}

2. Related Work

Relaxed operators for sorting procedures were first proposed in the context of Learning to Rank with the end goal of enabling direct optimization of Information Retrieval (IR) metrics. Many IR metrics, such as the Normalized Discounted Cumulative Gain (NDCG) \cite{jarvelin2002impact}, are defined in terms of ranks. Formally, the rank operator is defined as the function \( \text{rank} : \mathbb{R}^n \rightarrow S_n \) that returns the inverse of the \textit{argsort} permutation: \( \text{rank}(s) = \text{argsort}(s)^{-1} \), or equivalently \( P_{\text{rank}}(s) = P_{\text{argsort}}^T(s) \). The rank operator is thus intimately related to the \textit{argsort} operator; in fact, a relaxation for the \( P_{\text{rank}}(\cdot) \) operator can be obtained by transposing a relaxation for the \( P_{\text{argsort}}(\cdot) \) operator, and vice-versa; this duality is apparent in the work of \cite{cuturi2019relaxed}.

We begin by discussing previous work on relaxed rank operators in section 2.1. Next, we discuss more recent work, which deals with relaxations for the \( P_{\text{argsort}}(\cdot) \) operator.

2.1. Relaxed Rank Operators

The first work to propose a relaxed rank operator is that of \cite{taylor2008soft}. The authors introduce the relaxation \( \text{SoftRank}_r : \mathbb{R}^n \rightarrow \mathbb{R}^n \) given by \( \text{SoftRank}_r(z) = \mathbb{E}[\text{rank}(s + z)] \) where \( z \sim \mathcal{N}(0, \tau I_n) \), and show that this relaxation, as well as its gradients, can be computed exactly in time \( \mathcal{O}(n^2) \). Note that as \( \tau \rightarrow 0 \), \( \text{SoftRank}_r(z) \rightarrow \text{rank}(z) \). This relaxation is used in turn to define a surrogate for NDCG which can be optimized directly.

\*\*\*Except when there are ties, which we assume is not the case. Ties are merely a technical nuisance and do not represent a problem for any of the methods (ours or others') discussed in this paper.
In (Qin et al. 2010), another relaxation for the rank operator \( \pi_\tau : \mathbb{R}^n \to \mathbb{R}^n \) is proposed, defined as:

\[
\hat{\pi}_\tau(s)[i] = 1 + \sum_{j \neq i} \sigma \left( \frac{s_i - s_j}{\tau} \right)
\]

where \( \sigma(x) = (1 + \exp(-x))^{-1} \) is the sigmoid function. Again, \( \hat{\pi}_\tau(s) \to \text{rank}(s) \) as \( \tau \to 0 \). This operator can be computed in time \( O(n^2) \), which is faster than the \( O(n^3) \) approach of (Taylor et al. 2008).

Note that the above two approaches cannot be used to define a relaxation for the \( \text{argsort} \) operator. Indeed, \( \text{SoftRank}_\tau(s) \) and \( \hat{\pi}_\tau(s) \) are not relaxations for the \( P_{\text{rank}}(\cdot) \) operator. Instead, they directly relax the \( \text{rank} \) operator, and there is no easy way to obtain a relaxed \( \text{argsort} \) or \( P_{\text{argsort}}(\cdot) \) operator from them.

### 2.2. Sorting via Bipartite Maximum Matchings

The work of (Mena et al. 2018) draws an analogy between the \( \text{argmax} \) operator and the \( \text{argsort} \) operator by means of bipartite maximum matchings: the \( \text{argmax} \) operator applied to an \( n \)-dimensional vector \( s \) can be viewed as a maximum matching on a bipartite graph with \( n \) vertices in one component and 1 vertex in the other component, the edge weights equal to the given vector \( s \); similarly, a permutation matrix can be seen as a maximum matching on a bipartite graph between two groups of \( n \) vertices with edge weights given by a matrix \( X \in \mathbb{R}^{n \times n} \). This induces a mapping \( M \) (for ‘matching’) from the set of square matrices \( X \in \mathbb{R}^{n \times n} \) to the set \( \mathcal{P}_n \). Note that this mapping has arity \( M : \mathbb{R}^{n \times n} \to \mathcal{P}_n \), unlike the \( P_{\text{argsort}}(\cdot) \) operator which has arity \( P_{\text{argsort}}(\cdot) : \mathbb{R}^n \to \mathcal{P}_n \).

Like the \( P_{\text{argsort}}(\cdot) \) operator, the \( M \) operator has discrete image \( \mathcal{P}_n \), so to enable end-to-end gradient-based optimization, (Mena et al. 2018) propose to relax the matching operator \( M(X) \) by means of the Sinkhorn operator \( S(X/\tau) \); \( \tau \) is a temperature parameter that controls the degree of the approximation; as \( \tau \to 0 \) they show that \( S(X/\tau) \to M(X) \). The Sinkhorn operator \( S \) maps the square matrix \( X/\tau \) to the Birkhoff polytope \( \mathcal{B}_n \), which is defined as the set of doubly stochastic matrices (i.e. rows and columns summing to 1).

The computational complexity of the (Mena et al. 2018) approach to differentiable sorting is thus \( O(Ln^2) \) where \( L \) is the number of Sinkhorn iterates used to approximate \( S(X/\tau) \); the authors use \( L = 20 \) for all experiments.

### 2.3. Sorting via Optimal Transport

The recent work of (Cuturi et al. 2019) also makes use of the Sinkhorn operator to derive a continuous relaxation for the \( P_{\text{argsort}}(\cdot) \) operator. This time, the authors are motivated by the observation that a sorting permutation for \( s \in \mathbb{R}^n \) can be recovered from an optimal transport plan between two discrete measures defined on the real line, one of which is supported on the elements of \( s \) and the other of which is supported on arbitrary values \( y_1 < \cdots < y_n \). Indeed, the optimal transport plan between the probability measures \( \frac{1}{n} \sum_{i=1}^n \delta_{s_i} \) and \( \frac{1}{n} \sum_{i=1}^n \delta_{y_i} \) (where \( \delta_x \) is the Dirac delta at \( x \)) is given by the matrix \( P_{\text{argsort}}(\cdot) \).

Notably, a variant of the optimal transport problem with entropy regularization yields instead a continuous relaxation \( P_{\text{argsort}}(\cdot) \) mapping \( s \) to the Birkhoff polytope \( \mathcal{B}_n \); \( \epsilon \) plays a role similar to the temperature in (Mena et al. 2018), with \( P_{\text{argsort}}(\cdot) \to P_{\text{argsort}}(\cdot) \) as \( \epsilon \to 0 \). This relaxation can be computed via Sinkhorn iterates, and enables the authors to relax \( P_{\text{argsort}}(\cdot) \) by means of \( P_{\text{argsort}}(\cdot) \). Gradients can be computed by backpropagating through the Sinkhorn operator as in (Mena et al. 2018).

The computational complexity of this approach is again \( O(Ln^2) \). However, the authors show that a generalization of their method can be used to compute relaxed quantiles in time \( O(Ln) \), which is interesting in its own right.

### 2.4. Sorting via a sum-of-top-\( k \) elements identity

Finally, a more computationally efficient approach to differentiable sorting is proposed in (Grover et al. 2019). The authors rely on an identity that expresses the sum of the top \( k \) elements of a vector \( s \in \mathbb{R}^n \) as a symmetric function of \( s_1, \ldots, s_n \) that only involves max and min operations (Ogryczak & Tamir 2003) Lemma 1). Based on this identity, and denoting by \( A_s \) the matrix of absolute pairwise differences of elements of \( s \), namely \( A_s[i,j] = |s_i - s_j| \), the authors prove the identity:

\[
P_{\text{argsort}}(s)[i,j] = \begin{cases} 1 & \text{if } j = \text{argmax}(c_i), \\ 0 & \text{otherwise} \end{cases}
\]

where \( c_i = (n + 1 - 2i)s - A_s1 \), and \( 1 \) denotes the column vector of all ones.

Therefore, by replacing the \( \text{argmax} \) operator in Eq. 2 by a row-wise \( \text{softmax} \), the authors arrive at the following continuous relaxation for the \( P_{\text{argsort}}(\cdot) \) operator, which they call \( \text{NeuralSort} \):

\[
\text{NeuralSort}_\tau(s)[i,\cdot] = \text{softmax} \left( \frac{c_i}{\tau} \right)
\]

Again, \( \tau > 0 \) is a temperature parameter that controls the degree of the approximation; as \( \tau \to 0 \) they show that \( \text{NeuralSort}_\tau(s) \to P_{\text{argsort}}(s) \). Notably, the relaxation proposed by (Grover et al. 2019) can be computed in time \( O(n^2) \), making it much faster than the competing approaches of (Mena et al. 2018; Cuturi et al. 2019).
SoftSort: A Continuous Relaxation for the argsort Operator

\[
\text{NeuralSort}_τ(s) = g_τ \begin{pmatrix}
0 & s_2 - s_1 & 3s_3 - s_1 - 2s_2 & 5s_4 - s_1 - 2s_2 - 2s_3 \\
0 & s_2 - s_1 & s_3 - s_2 & 3s_4 - s_2 - 2s_3 \\
2s_2 + 3s_3 - 3s_1 & s_3 - s_2 & s_4 - s_3 \\
2s_2 + 3s_3 + s_4 - 3s_1 & 2s_3 + s_4 - 3s_2 & 0
\end{pmatrix}
\]

Figure 2. The 4-dimensional NeuralSort operator on the region of space where \( s_1 \geq s_2 \geq s_3 \geq s_4 \). We define \( g_τ(X) \) as a row-wise \( \text{softmax} \) with temperature: \( \text{softmax}(X/τ) \). As the most closely related work, this is the main baseline for our experiments.

\[
\text{SoftSort}_τ^d(s) = g_τ \begin{pmatrix}
0 & s_2 - s_1 & s_3 - s_1 & s_4 - s_1 \\
s_2 - s_1 & 0 & s_4 - s_3 \\
s_3 - s_1 & s_4 - s_3 \\
s_4 - s_1 & s_4 - s_3 & 0
\end{pmatrix}
\]

Figure 3. The 4-dimensional SoftSort operator on the region of space where \( s_1 \geq s_2 \geq s_3 \geq s_4 \) using \( d = |\cdot| \). We define \( g_τ(X) \) as a row-wise \( \text{softmax} \) with temperature: \( \text{softmax}(X/τ) \). Our formulation is much simpler than previous approaches.

3. **SoftSort**: A simple relaxed sorting operator

In this paper we propose SoftSort, a simple continuous relaxation for the \( P_{\text{argsort}(i)} \) operator. We define SoftSort as follows:

\[
\text{SoftSort}_τ^d(s) = \text{softmax}\left( -\frac{d(\text{sort}(s) 1^T, 1 s^T)}{τ} \right)
\]

where \( τ > 0 \) is a temperature parameter that controls the degree of the approximation and \( d \) is semi–metric function applied pointwise that is differentiable almost everywhere. Recall that a semi–metric has all the properties of a metric except the triangle inequality, which is not required. Examples of semi–metrics in \( \mathbb{R} \) include any positive power of the absolute value. The SoftSort operator has similar desirable properties to those of the NeuralSort operator, while being significantly simpler. Here we state and prove these properties. We start with the definition of *Unimodal Row Stochastic Matrices* ([Grover et al., 2019]), which summarizes the properties of our relaxed operator:

**Definition 1** (*Unimodal Row Stochastic Matrices*). An \( n \times n \) matrix is Unimodal Row Stochastic (URS) if it satisfies the following conditions:

1. **Non-negativity**: \( U[i, j] \geq 0 \) \( \forall i, j \in \{1, 2, \ldots, n\} \).
2. **Row Affinity**: \( \sum_{j=1}^{n} U[i, j] = 1 \) \( \forall i \in \{1, 2, \ldots, n\} \).
3. **Argmax Permutation**: Let \( u_\cdot \) denote a vector of size \( n \) such that \( u_\cdot = \arg \max_{j} U[i, j] \) \( \forall i \in \{1, 2, \ldots, n\} \). Then, \( u \in S_n \), i.e., it is a valid permutation.

While NeuralSort and SoftSort yield URS matrices (we will prove this shortly), the approaches of ([Mena et al., 2018] [Cuturi et al., 2019]) yield bistochastic matrices. It is natural to ask whether URS matrices should be preferred over bistochastic matrices for relaxing the \( P_{\text{argsort}(i)} \) operator. Note that URS matrices are not comparable to bistochastic matrices: they drop the column-stochasticity condition, but require that each row have a distinct \( \arg\max \), which is not true of bistochastic matrices. This means that URS matrices can be trivially projected onto the probability simplex, which is useful for e.g. straight-through gradient optimization, or whenever hard permutation matrices are required, such as at test time. The one property URS matrices lack is column-stochasticity, but this is not central to soft sorting. Instead, this property arises in the work of ([Mena et al., 2018]) because their goal is to relax the bipartite matching operator (rather than the \( \text{argsort} \) operator), and in this context bistochastic matrices are the natural choice. Similarly, ([Cuturi et al., 2019]) yields bistochastic matrices because they are the solutions to optimal transport problems (this does, however, allow them to simultaneously relax the \( \text{argsort} \) and rank operators). Since our only goal (as in the NeuralSort paper) is to relax the \( \text{argsort} \) operator, column-stochasticity can be dropped, and URS matrices are the more natural choice.

Now on to the main Theorem, which shows that SoftSort has the same desirable properties as NeuralSort. These are ([Grover et al., 2019] Theorem 4):

**Theorem 1** The SoftSort operator has the following properties:

1. **Unimodality**: \( \forall τ > 0, \text{SoftSort}_τ^d(s) \) is a unimodal row stochastic matrix. Further, let \( u \) denote the permutation obtained by applying \( \arg \max \) row-wise to \( \text{SoftSort}_τ^d(s) \). Then, \( u = \text{argsort}(s) \).
2. Limiting behavior: If no elements of \( s \) coincide, then:

\[
\lim_{\tau \to 0^+} \text{SoftSort}^d_{\tau}(s) = P_{\text{argsort}}(s)
\]

In particular, this limit holds almost surely if the entries of \( s \) are drawn from a distribution that is absolutely continuous w.r.t. the Lebesgue measure on \( \mathbb{R} \).

Proof.

1. Non-negativity and row affinity follow from the fact that every row in \( \text{SoftSort}^d_{\tau}(s) \) is the result of a softmax operation. For the third property, we use that softmax preserves maxima and that \( d(\cdot, x) \) has a unique minimum at \( x \) for every \( x \in \mathbb{R} \). Formally, let \( \text{sort}(s) = [s_1, \ldots, s_n]^T \), i.e. \( s_1 \geq \cdots \geq s_n \) are the decreasing order statistics of \( s \). Then:

\[
\begin{align*}
  u_i &= \arg \max_j \text{SoftSort}^d_{\tau}(s)[i, j] \\
  &= \arg \max_j \left( \text{softmax}(-d(s[i], s_j)/\tau) \right) \\
  &= \arg \min_j \left( d(s[i], s_j) \right) \\
  &= \text{argsort}(s)[i]
\end{align*}
\]

as desired.

2. It suffices to show that the \( i \)-th row of \( \text{SoftSort}^d_{\tau}(s) \) converges to the one-hot representation \( h \) of \( \text{argsort}(s)[i] \). But by part 1, the \( i \)-th row of \( \text{SoftSort}^d_{\tau}(s) \) is the softmax of \( v/\tau \) where \( v \) is a vector whose unique argmax is \( \text{argsort}(s)[i] \). Since it is a well-known property of the softmax that \( \lim_{\tau \to 0^+} \text{softmax}(v/\tau) = h \) [Elfadel & Wyatt, 1994], we are done.

Note that the proof of unimodality of the \text{SoftSort} operator is straightforward, unlike the proof for the \text{NeuralSort} operator, which requires proving a more technical Lemma and Corollary [Grover et al., 2019, Lemma 2, Corollary 3).

The row-stochastic property can be loosely interpreted as follows: row \( r \) of \text{SoftSort} and \text{NeuralSort} encodes a distribution over the value of the rank \( r \) element, more precisely, the probability of it being equal to \( s_j \) for each \( j \). In particular, note that the first row of the \text{SoftSort} operator is precisely the softmax of the input vector. In general, the \( r \)-th row of the \text{SoftSort} operator is the softmax of the negative distances to the \( r \)-th largest element.

Finally, regarding the choice of \( d \) in \text{SoftSort}, even though a large family of semi–metrics could be considered, in this work we experimented with the absolute value as well as the square distance and found the absolute value to be marginally better during experimentation. With this in consideration, in what follows we fix \( d = | \cdot | \) the absolute value function, unless stated otherwise. We leave for future work learning the metric \( d \) or exploring a larger family of such functions.

4. Comparing \text{SoftSort} to \text{NeuralSort}

4.1. Mathematical Simplicity

The difference between \text{SoftSort} and \text{NeuralSort} becomes apparent once we write down what the actual operators look like; the equations defining them (Eq. 3, Eq. 4) are compact but do not offer much insight. Note that even though the work of [Grover et al., 2019] shows that the \text{NeuralSort} operator has the desirable properties of Theorem 1, the paper never gives a concrete example of what the operator instantiates to in practice, which keeps some of its complexity hidden.

Let \( g_{\tau} : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n} \) be the function defined as \( g_{\tau}(X) = \text{softmax}(X/\tau) \), where the softmax is applied row-wise. Suppose that \( n = 4 \) and that \( s \) is sorted in decreasing order \( s_1 \geq s_2 \geq s_3 \geq s_4 \). Then the \text{NeuralSort} operator is given in Figure 2 and the \text{SoftSort} operator is given in Figure 3. Note that the diagonal of the logit matrix has been 0-centered by subtracting a constant value from each row; this does not change the softmax and simplifies the expressions. The \text{SoftSort} operator is straightforward, with the \( i,j \)-th entry of the logit matrix given by \( -|s_i - s_j| \). In contrast, the \( i,j \)-th entry of the \text{NeuralSort} operator depends on all intermediate values \( s_1, s_{i+1}, \ldots, s_j \). This is a consequence of the coupling between the \text{NeuralSort} operator and the complex identity used to derive it. As we show in this paper, this complexity can be completely avoided, and results in further benefits beyond aesthetic simplicity such as flexibility, speed and mathematical simplicity.

Note that for an arbitrary region of space other than \( s_1 \geq s_2 \geq s_3 \geq s_4 \), the \text{NeuralSort} and \text{SoftSort} operators look just like Figures 2 and 3 respectively except for relabelling of the \( s_i \) and column permutations. Indeed, we have:

**Proposition 1** For both \( f = \text{SoftSort}^d \) and \( f = \text{NeuralSort} \), the following identity holds:

\[
f(s) = f(\text{sort}(s))P_{\text{argsort}}(s)
\]

We defer the proof to appendix B. This proposition is interesting because it implies that the behaviour of the \text{SoftSort} and \text{NeuralSort} operators can be completely characterized by their functional form on the region of space where \( s_1 \geq s_2 \geq \cdots \geq s_n \). Indeed, for any other value of \( s \), we can compute the value of...
SoftSort: A Continuous Relaxation for the argsort Operator

def neural_sort(s, tau):
    n = tf.shape(s)[1]
    one = tf.ones((n, 1), dtype=tf.float32)
    A_s = tf.abs(s - tf.transpose(s, perm=[0, 2, 1]))
    B = tf.matmul(A_s, tf.matmul(one, tf.transpose(one)))
    scaling = tf.cast(n + 1 - 2 * (tf.range(n) + 1), dtype=tf.float32)
    C = tf.matmul(s, tf.expand_dims(scaling, 0))
    P_max = tf.transpose(C-B, perm=[0, 2, 1])
    P_hat = tf.nn.softmax(P_max / tau, -1)
    return P_hat

Figure 4. Implementation of NeuralSort in TensorFlow as given in (Grover et al., 2019)

def soft_sort(s, tau):
    s_sorted = tf.sort(s, direction='DESCENDING', axis=1)
    pairwise_distances = -tf.abs(tf.transpose(s, perm=[0, 2, 1]) - s_sorted)
    P_hat = tf.nn.softmax(pairwise_distances / tau, -1)
    return P_hat

Figure 5. Implementation of SoftSort in TensorFlow as proposed by us, with $d = |\cdot|$

SoftSort(s) or NeuralSort(s) by first sorting s, then applying SoftSort or NeuralSort, and finally sorting the columns of the resulting matrix with the inverse permutation that sorts s. In particular, to our point, the proposition shows that Figures 2 and 3 are valid for all s up to relabeling of the $s_i$ (by $s_{[i]}$) and column permutations (by the inverse permutation that sorts s).

To further our comparison, in appendix D we show how the SoftSort and NeuralSort operators differ in terms of the size of their matrix entries.

4.2. Code Simplicity

In Figures 4 and 5 we show TensorFlow implementations of the NeuralSort and SoftSort operators respectively. SoftSort has a simpler implementation than NeuralSort, which we shall see is reflected in its faster speed. (See section 5.5)

Note that our implementation of SoftSort is based directly off Eq. [4] and we rely on the sort operator. We would like to remark that there is nothing wrong with using the sort operator in a stochastic computation graph. Indeed, the sort function is continuous, almost everywhere differentiable (with non-zero gradients) and piecewise linear, just like the max, min or ReLU functions.

Finally, the unimodality property (Theorem 1) implies that any algorithm that builds a relaxed permutation matrix can be used to construct the true discrete permutation matrix. This means that any relaxed sorting algorithm (in particular, NeuralSort) is lower bounded by the complexity of sorting, which justifies relying on sorting as a subroutine. As we show later, SoftSort is faster than NeuralSort. Also, we believe that this modular approach is a net positive since sorting in CPU and GPU is a well studied problem (Singh et al., 2017) and any underlying improvements will benefit SoftSort’s speed as well. For instance, the current implementation in TensorFlow relies on radix sort and heap sort depending on input size.

5. Experiments

We first compare SoftSort to NeuralSort on the benchmarks from the NeuralSort paper (Grover et al., 2019), using the code kindly open-sourced by the authors. We show that SoftSort performs comparably to NeuralSort. Then, we devise a specific experiment to benchmark the speeds of the SoftSort and NeuralSort operators in isolation, and show that SoftSort is faster than NeuralSort while taking the same number of gradient steps to converge. This makes SoftSort not only the simplest, but also the fastest relaxed sorting operator proposed to date.

5.1. Models

For both SoftSort and NeuralSort we consider their deterministic and stochastic variants as in (Grover et al., 2019). The deterministic operators are those given by equations [3] and [4]. The stochastic variants are Plackett-Luce distributions reparameterized via Gumbel distributions (Grover et al., 2019, Section 4.1), where the $P_{\text{argsort}}(\cdot)$ operator that is applied to the samples is relaxed by means of the SoftSort or NeuralSort operator; this is analogous to the Gumbel-Softmax trick where the argmax operator that is applied to the samples is relaxed via the softmax operator (Jang et al., 2017; Maddison et al., 2017).
Table 1. Results for the sorting task averaged over 10 runs. We report the mean and standard deviation for the proportion of correct permutations. From \( n = 7 \) onward, the results are comparable with the state of the art.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( n = 3 )</th>
<th>( n = 5 )</th>
<th>( n = 7 )</th>
<th>( n = 9 )</th>
<th>( n = 15 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic NeuralSort</td>
<td>0.921 ± 0.006</td>
<td>0.797 ± 0.010</td>
<td>0.663 ± 0.016</td>
<td>0.547 ± 0.015</td>
<td>0.253 ± 0.021</td>
</tr>
<tr>
<td>Stochastic NeuralSort</td>
<td>0.918 ± 0.007</td>
<td>0.801 ± 0.010</td>
<td>0.665 ± 0.011</td>
<td>0.540 ± 0.019</td>
<td>0.250 ± 0.017</td>
</tr>
<tr>
<td>Stochastic SoftSort (Ours)</td>
<td>0.918 ± 0.005</td>
<td>0.796 ± 0.009</td>
<td>0.666 ± 0.016</td>
<td>0.544 ± 0.015</td>
<td>0.256 ± 0.023</td>
</tr>
<tr>
<td>Stochastic SoftSort (Ours)</td>
<td>0.918 ± 0.005</td>
<td>0.798 ± 0.005</td>
<td>0.669 ± 0.018</td>
<td>0.548 ± 0.019</td>
<td>0.250 ± 0.020</td>
</tr>
</tbody>
</table>

\textbf{Cuturi et al.} \cite{cuturi2019report} reported 0.928 \( \pm \) 0.111 \( \pm \) 0.656 \( \pm \) 0.497 \( \pm \) 0.126.

Table 2. Results for the quantile regression task. The first metric is the mean squared error (\( \times 10^{-4} \)) when predicting the median number. The second metric - in parenthesis - is Spearman’s \( R^2 \) for the predictions. Results are again comparable with the state of the art.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( n = 5 )</th>
<th>( n = 9 )</th>
<th>( n = 15 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic NeuralSort</td>
<td>21.52 (0.97)</td>
<td>15.00 (0.97)</td>
<td>18.81 (0.95)</td>
</tr>
<tr>
<td>Stochastic NeuralSort</td>
<td>24.78 (0.97)</td>
<td>17.79 (0.96)</td>
<td>18.10 (0.94)</td>
</tr>
<tr>
<td>Deterministic SoftSort (Ours)</td>
<td>23.44 (0.97)</td>
<td>19.26 (0.96)</td>
<td>\textbf{15.54 (0.95)}</td>
</tr>
<tr>
<td>Stochastic SoftSort (Ours)</td>
<td>26.17 (0.97)</td>
<td>19.06 (0.96)</td>
<td>20.65 (0.94)</td>
</tr>
</tbody>
</table>

5.2. Sorting Handwritten Numbers

The large-MNIST dataset of handwritten numbers is formed by concatenating 4 randomly selected MNIST digits. In this task, a neural network is presented with a sequence of \( n \) large-MNIST numbers and must learn the permutation that sorts these numbers. Supervision is provided only in the form of the ground-truth permutation. Performance on the task is measured by:

1. The proportion of permutations that are perfectly recovered.
2. The proportion of permutation elements that are correctly recovered.

Note that the first metric is always less than or equal to the second metric. We use the setup in \cite{cuturi2019report} to be able to compare against their Optimal-Transport-based method. They use 100 epochs to train all models.

The results for the first metric are shown in Table 1. We report the mean and standard deviation over 10 runs. We see that SoftSort and NeuralSort perform identically for all values of \( n \). Moreover, our results for NeuralSort are better than those reported in \cite{cuturi2019report}, to the extent that NeuralSort and SoftSort outperform the method of \cite{cuturi2019report} for \( n = 9, 15 \), unlike reported in said paper. We found that the hyperparameter values reported in \cite{grover2019} and used by \cite{cuturi2019report} for NeuralSort are far from ideal. \cite{grover2019} reports using a learning rate of \( 10^{-4} \) and temperature values from the set \{1, 2, 4, 8, 16\}. However, a higher learning rate dramatically improves NeuralSort’s results, and higher temperatures also help. Concretely, we used a learning rate of 0.005 for all the SoftSort and NeuralSort models, and a value of \( \tau = 1024 \) for \( n = 3, 5, 7 \) and \( \tau = 128 \) for \( n = 9, 15 \). The results for the second metric are reported in appendix \ref{app:metrics}. In the appendix we also include learning curves for SoftSort and NeuralSort, which show that they converge at the same speed.

5.3. Quantile Regression

As in the sorting task, a neural network is presented with a sequence of \( n \) large-MNIST numbers. The task is to predict the median element from the sequence, and this is the only available form of supervision. Performance on the task is measured by mean squared error and Spearman’s rank correlation. We used 100 iterations to train all models.

The results are shown in Table 2. We used a learning rate of \( 10^{-3} \) for all models - again, higher than that reported in \cite{grover2019} - and grid-searched the temperature on the set \{128, 256, 512, 1024, 2048, 4096\} - again, higher than that reported in \cite{grover2019}. We see that SoftSort and NeuralSort perform similarly, with NeuralSort sometimes outperforming SoftSort and vice versa. The results for NeuralSort are also much better than those reported in \cite{grover2019}, which we attribute to the better choice of hyperparameters, concretely, the higher learning rate. In the appendix we also include learning curves for SoftSort and NeuralSort, which show that they converge at the same speed.

5.4. Differentiable kNN

In this setup, we explored using the SoftSort operator to learn a differentiable \( k \)-nearest neighbours (kNN) classifier that is able to learn a representation function, used to
SoftSort: A Continuous Relaxation for the argsort Operator

Figure 6. Speed of the NeuralSort and SoftSort operators on (a) CPU and (b) GPU, as a function of \( n \) (the size of the vector to be sorted). Twenty vectors of size \( n \) are batched together during each epoch. The original NeuralSort implementation is up to 6 times slower on both CPU and GPU. After some performance improvements, NeuralSort is 80% slower on CPU and 40% slower on GPU.

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MNIST</th>
<th>FASHION-MNIST</th>
<th>CIFAR-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>kNN+Pixels</td>
<td>97.2%</td>
<td>85.8%</td>
<td>35.4%</td>
</tr>
<tr>
<td>kNN+PCA</td>
<td>97.6%</td>
<td>85.9%</td>
<td>40.9%</td>
</tr>
<tr>
<td>kNN+AE</td>
<td>97.6%</td>
<td>87.5%</td>
<td>44.2%</td>
</tr>
<tr>
<td>kNN + DETERMINISTIC NeuralSort</td>
<td><strong>99.5%</strong></td>
<td>93.5%</td>
<td>90.7%</td>
</tr>
<tr>
<td>kNN + STOCHASTIC NeuralSort</td>
<td>99.4%</td>
<td>93.4%</td>
<td>89.5%</td>
</tr>
<tr>
<td>kNN + DETERMINISTIC SoftSort (Ours)</td>
<td>99.37%</td>
<td>93.60%</td>
<td>92.03%</td>
</tr>
<tr>
<td>kNN + STOCHASTIC SoftSort (Ours)</td>
<td>99.42%</td>
<td><strong>93.67%</strong></td>
<td>90.64%</td>
</tr>
<tr>
<td>CNN (w/o kNN)</td>
<td>99.4%</td>
<td>93.4%</td>
<td>95.1%</td>
</tr>
</tbody>
</table>

Table 3. Average test classification accuracy comparing k-nearest neighbor models. The last row includes the results from non kNN classifier. The results are comparable with the state of the art, or above it by a small margin.

measure the distance between the candidates.

In a supervised training framework, we have a dataset that consists of pairs \((x, y)\) of a datapoint and a label. We are interested in learning a map \(\Phi\) to embed every \(x\) such that we can use a kNN classifier to identify the class of \(\hat{x}\) by looking at the class of its closest neighbours according to the distance \(\|\Phi(x) - \Phi(\hat{x})\|\). Such a classifier would be valuable by virtue of being interpretable and robust to both noise and unseen classes.

This is achieved by constructing episodes during training that consist of one pair \(\hat{x}, \hat{y}\) and \(n\) candidate pairs \((x_i, y_i)\) for \(i = 1 \ldots n\), arranged in two column vectors \(X\) and \(Y\). The probability \(P(\hat{y}|\hat{x}, X, Y)\) of class \(\hat{y}\) under a kNN classifier is the average of the first \(k\) entries in the vector

\[
P_{\text{argsort}}(\Phi(X) - \Phi(\hat{x}))^2 \mathbb{I}_{Y = \hat{y}}
\]

where \(\|\Phi(X) - \Phi(\hat{x})\|^2\) is the vector of squared distances from the candidate points and \(\mathbb{I}_{Y = \hat{y}}\) is the binary vector indicating which indexes have class \(\hat{y}\). Thus, if we replace \(P_{\text{argsort}}\) by the SoftSort operator we obtain a differentiable relaxation \(\hat{P}(\hat{y}|\hat{x}, X, Y)\). To compute the loss we follow [Grover et al., 2019] and use \(-\hat{P}(\hat{y}|\hat{x}, X, Y)\). We also experimented with the cross entropy loss, but the performance went down for both methods.

When \(k = 1\), our method is closely related to Matching Networks [Vinyals et al., 2016]. This follows from the following result: (See proof in Appendix C)

**Proposition 2** Let \(k = 1\) and \(\hat{P}\) be the differentiable kNN operator using SoftSort\(_2\). If we choose the embedding function \(\Phi\) to be of norm 1, then

\[
\hat{P}(\hat{y}|\hat{x}, X, Y) = \frac{\sum_{i:y_i = \hat{y}} e^{\Phi(\hat{x}) \cdot \Phi(x_i)}}{\sum_{i=1 \ldots n} e^{\Phi(\hat{x}) \cdot \Phi(x_i)}}
\]

This suggests that our method is a generalization of Matching Networks, since in our experiments larger values of \(k\) yielded better results consistently and we expect a kNN classifier to be more robust in a setup with noisy labels.
However, Matching Networks use contextual embedding functions, and different networks for \( \hat{x} \) and the candidates \( x_i \), both of which could be incorporated to our setup. A more comprehensive study comparing both methods on a few shot classification dataset such as Omniglot (Lake et al., 2011) is left for future work.

We applied this method to three benchmark datasets: MNIST, Fashion MNIST and CIFAR-10 with canonical splits. As baselines, we compare against NeuralSort as well as other kNN models with fixed representations coming from raw pixels, a PCA feature extractor and an auto-encoder. All the results are based on the ones reported in (Grover et al., 2019). We also included for comparison a standard classifier using a convolutional network.

Results are shown in Table 1. In every case, we achieve comparable accuracies with NeuralSort implementation, either slightly outperforming or underperforming NeuralSort. See hyperparameters used in appendix A.4.5.

5.5. Speed Comparison

We set up an experiment to compare the speed of the SoftSort and NeuralSort operators. We are interested in exercising both their forward and backward calls. To this end, we set up a dummy learning task where the goal is to perturb an \( n \)-dimensional input vector \( \theta \) to make it become sorted. We scale \( \theta \) to \([0, 1]\) and feed it through the SoftSort or NeuralSort operator to obtain \( \hat{P}(\theta) \), and place a loss on \( \hat{P}(\theta) \) that encourages it to become equal to the identity matrix, and thus encourages the input to become sorted.

Concretely, we place the cross-entropy loss between the true permutation matrix and the predicted soft URS matrix:

\[
L(\hat{P}) = -\frac{1}{n} \sum_{i=1}^{n} \log \hat{P}[i, i]
\]

This encourages the probability mass from each row of \( \hat{P} \) to concentrate on the diagonal, which drives \( \theta \) to sort itself. Note that this is a trivial task, since for example a pointwise ranking loss \( \frac{1}{n} \sum_{i=1}^{n} (\theta_i + i)^2 \) (Taylor et al., 2008) Section 2.2) leads the input to become sorted too, without any need for the SoftSort or NeuralSort operators. However, this task is a reasonable benchmark to measure the speed of the two operators in a realistic learning setting.

We benchmark \( n \) in the range 100 to 4000, and batch 20 inputs \( \theta \) together to exercise batching. Thus the input is a parameter tensor of shape \( 20 \times n \). Models are trained for 100 epochs, which we verified is enough for the parameter vectors to become perfectly sorted by the end of training (i.e., to succeed at the task).

In Figure 6 we show the results for the TensorFlow implementations of NeuralSort and SoftSort given in Figures 4 and 5 respectively. We see that on both CPU and GPU, SoftSort is faster than NeuralSort. For \( N = 4000 \), SoftSort is about 6 times faster than the NeuralSort implementation of (Grover et al., 2019) on both CPU and GPU. We tried to speed up the NeuralSort implementation of (Grover et al., 2019), and although we were able to improve it, NeuralSort was still slower than SoftSort, concretely: 80% slower on CPU and 40% slower on GPU. Details of our improvements to the speed of the NeuralSort operator are provided in appendix A.4.5.

The performance results for PyTorch are provided in the appendix and are similar to the TensorFlow results. In the appendix we also show that the learning curves of SoftSort with \( d = |\cdot|^2 \) and NeuralSort are almost identical; interestingly, we found that using \( d = |\cdot| \) converges more slowly on this synthetic task.

We also investigated if relying on a sorting routine could cause slower run times in worst-case scenarios. When using sequences sorted in the opposite order we did not note any significant slowdowns. Furthermore, in applications where this could be a concern, the effect can be avoided entirely by shuffling the inputs before applying our operator.

As a final note, given that the cost of sorting is sub-quadratic, and most of the computation is payed when building and applying the \( n \times n \) matrix, we also think that our algorithm could be made faster asymptotically by constructing sparse versions of the SoftSort operator. For applications like differentiable nearest neighbors, evidence suggests that processing longer sequences yields better results, which motivates improvements in the asymptotic complexity. We leave this topic for future work.

6. Conclusion

We have introduced SoftSort, a simple continuous relaxation for the argsort operator. The \( r \)-th row of the SoftSort operator is simply the softmax of the negative distances to the \( r \)-th largest element.

SoftSort has similar properties to those of the NeuralSort operator of (Grover et al., 2019). However, due to its simplicity, SoftSort is trivial to implement, more modular, faster than NeuralSort, and proofs regarding the SoftSort operator are effortless. We also empirically find that it is just as easy to optimize.

Fundamentally, SoftSort advances the state of the art in differentiable sorting by significantly simplifying previous approaches. Our code and experiments can be found at https://github.com/sprillo/softsort.
Acknowledgments

We thank Assistant Professor Jordan Boyd-Graber from University of Maryland and Visiting Researcher at Google, and Thomas Müller from Google Language Research at Zurich for their feedback and comments on earlier versions of the manuscript. We would also like to thank the anonymous reviewers for their feedback that helped improve this work.

References


