Appendix

A. Experimental Details

We use the code kindly open sourced by (Grover et al., 2019) to perform the sorting, quantile regression, and kNN experiments. As such, we are using the same setup as in (Grover et al., 2019). The work of (Cuturi et al., 2019) also uses this code for the sorting task, allowing for a fair comparison.

To make our work self-contained, in this section we recall the main experimental details from (Grover et al., 2019), and we also provide our hyperparameter settings, which crucially differ from those used in (Grover et al., 2019) by the use of higher learning rates and temperatures (leading to improved results).

A.1. Sorting Handwritten Numbers

A.1.1. Architecture

The convolutional neural network architecture used to map 112×28 large-MNIST images to scores is as follows:

Conv[Kernel: 5x5, Stride: 1, Output: 112x28x32, Activation: Relu]
→Pool[Stride: 2, Output: 56x14x32]
→Conv[Kernel: 5x5, Stride: 1, Output: 56x14x64, Activation: Relu]
→Pool[Stride: 2, Output: 28x7x64]
→FC[Units: 64, Activation: Relu]
→FC[Units: 1, Activation: None]

Recall that the large-MNIST dataset is formed by concatenating four 28×28 MNIST images, hence each large-MNIST image input is of size 112×28.

For a given input sequence x of large-MNIST images, using the above CNN we obtain a vector s of scores (one score per image). Feeding this score vector into `NeuralSort` or `SoftSort` yields the matrix $\widehat{P}(s)$ which is a relaxation for $P_{\texttt{argsort}(s)}$.

A.1.2. Loss Functions

To learn to sort the input sequence x of large-MNIST digits, (Grover et al., 2019) imposes a cross-entropy loss between the rows of the true permutation matrix P and the learnt relaxation $\widehat{P}(s)$, namely:

$$L = \frac{1}{n} \sum_{i,j=1}^{n} \mathbb{1}\{P[i,j] = 1\} \log \widehat{P}(s)[i,j]$$

This is the loss for one example (x, P) in the deterministic setup. For the stochastic setup with reparameterized Plackett-Luce distributions, the loss is instead:

$$L = \frac{1}{n} \sum_{i,j=1}^{n} \sum_{k=1}^{n_s} \mathbb{1}\{P[i,j] = 1\} \log \widehat{P}(s + z_k)[i,j]$$

where z_k $(1 \leq k \leq n_s)$ are samples from the Gumbel distribution.

A.1.3. Hyperparamaters

For this task we used an Adam optimizer with an initial learning rate of 0.005 and a batch size of 20. The temperature τ was selected from the set $\{1, 16, 128, 1024\}$ based on validation set accuracy on predicting entire permutations. As a results, we used a value of $\tau = 1024$ for $n = 3, 5, 7$ and $\tau = 128$ for $n = 9, 15$. 100 iterations were used to train all models. For the stochastic setting, $n_s = 5$ samples were used.

A.1.4. Learning Curves

In Figure 1 (a) we show the learning curves for deterministic `SoftSort` and `NeuralSort`, with $N = 15$. These are the average learning curves over all 10 runs. Both learning curves are almost identical, showing that in this task both operators can essentially be exchanged for one another.

A.2. Quantile Regression

A.2.1. Architecture

The same convolutional neural network as in the sorting task was used to map large-MNIST images to scores. A second neural network g_θ with the same architecture but different parameters is used to regress each image to its value. These two networks are then used by the loss functions below to learn the median element.

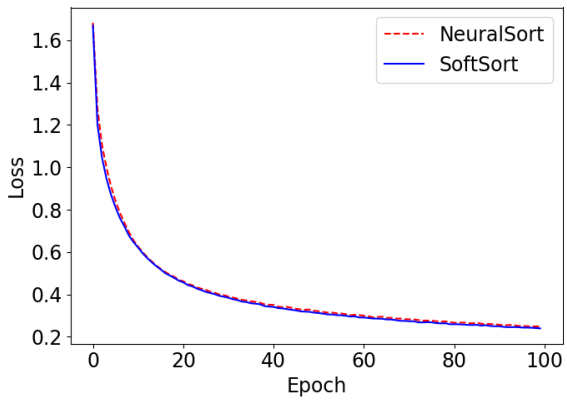
A.2.2. Loss Functions

To learn the median element of the input sequence x of large-MNIST digits, (Grover et al., 2019) first soft-sorts x via $\widehat{P}(s)x$ which allows extracting the candidate median image. This candidate median image is then mapped to its predicted value $\hat{y}$ via the CNN g_θ. The square loss between $\hat{y}$ and the true median value y is incurred. As in (Grover et al., 2019, Section E.2), the loss for a single example (x, y) can thus compactly be written as (in the deterministic case):

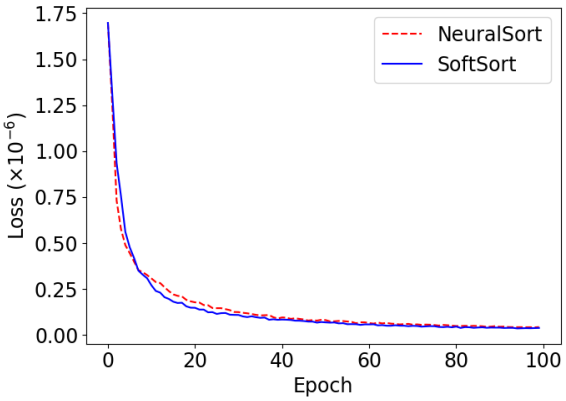
$$L = \|y - g_\theta(\widehat{P}(s)x)\|_2^2$$

For the stochastic case, the loss for a single example (x, y) is instead:

$$L = \sum_{k=1}^{n_s} \|y - g_\theta(\widehat{P}(s + z_k)x)\|_2^2$$



(a) Sorting handwritten numbers learning curves.



(b) Quantile regression learning curves

Figure 1. Learning curves for the 'sorting handwritten numbers' and 'quantile regression' tasks. The learning curves for `SoftSort` and `NeuralSort` are almost identical.

Table 1. Values of τ used for the quantile regression task.

ALGORITHM	$n = 5$	$n = 9$	$n = 15$
DETERMINISTIC NEURALSORT	1024	512	1024
STOCHASTIC NEURALSORT	2048	512	4096
DETERMINISTIC SOFTSORT	2048	2048	256
STOCHASTIC SOFTSORT	4096	2048	2048

where z_k $(1 \leq k \leq n_s)$ are samples from the Gumbel distribution.

A.2.3. HYPERPARAMATERS

We used an Adam optimizer with an initial learning rate of 0.001 and a batch size of 5. The value of τ was grid searched on the set $\{128, 256, 512, 1024, 2048, 4096\}$ based on validation set MSE. The final values of τ used to train the models and evaluate test set performance are given in Table 1. 100 iterations were used to train all models. For the stochastic setting, $n_s = 5$ samples were used.

A.2.4. LEARNING CURVES

In Figure 1 (b) we show the learning curves for deterministic `SoftSort` and `NeuralSort`, with $N = 15$. Both learning curves are almost identical, showing that in this task both operators can essentially be exchanged for one another.

A.3. Differentiable KNN

A.3.1. ARCHITECTURES

To embed the images before applying differentiable kNN, we used the following convolutional network architectures. For MNIST:

Conv[Kernel: 5x5, Stride: 1, Output: 24x24x20,
Activation: Relu]
→Pool[Stride: 2, Output: 12x12x20]
→Conv[Kernel: 5x5, Stride: 1, Output: 8x8x50,
Activation: Relu]
→Pool[Stride: 2, Output: 4x4x50]
→FC[Units: 50, Activation: Relu]

and for Fashion-MNIST and CIFAR-10 we used the *ResNet18* architecture (He et al., 2016) as defined in github.com/kuangliu/pytorch-cifar, but we keep the 512 dimensional output before the last classification layer.

For the baseline experiments in pixel distance with PCA and kNN, we report the results of (Grover et al., 2019), using *scikit-learn* implementations.

In the auto-encoder baselines, the embeddings were trained using the follow architectures. For MNIST and Fashion-

MNIST:

Encoder:
FC[Units: 500, Activation: Relu]
→FC[Units: 500, Activation: Relu]
→FC[Units: 50, Activation: Relu]
Decoder:
→FC[Units: 500, Activation: Relu]
→FC[Units: 500, Activation: Relu]
→FC[Units: 784, Activation: Sigmoid]

For CIFAR-10, we follow the architecture defined at github.com/shibuiwilliam/Keras_Autoencoder, with a bottleneck dimension of 256.

A.3.2. Loss Functions

For the models using `SoftSort` or `NeuralSort` we use the negative of the probability output from the kNN model as a loss function. For the auto-encoder baselines we use a per-pixel binary cross entropy loss.

A.3.3. Hyperparamaters

We perform a grid search for $k \in (1, 3, 5, 9)$, $\tau \in (1, 4, 16, 64, 128, 512)$, learning rates taking values in 10^{-3}, 10^{-4} and 10^{-5}. We train for 200 epochs and choose the model based on validation loss. The optimizer used is *SGD* with momentum of 0.9. Every batch has 100 episode, each containing 100 candidates.

A.4. Speed Comparison

A.4.1. Architecture

The input parameter vector θ of shape $20 \times n$ (20 being the batch size) is first normalized to $[0, 1]$ and then fed through the `NeuralSort` or `SoftSort` operator, producing an output tensor $\widehat{P}$ of shape $20 \times n \times n$.

A.4.2. Loss Function

We impose the following loss term over the batch:

$$L(\widehat{P}) = -\frac{1}{20}\sum_{i=1}^{20}\frac{1}{n}\sum_{j=1}^{n}\log \widehat{P}[i, j, j]$$

This loss term encourages the probability mass from each row of $\widehat{P}[i, :, :]$ to concentrate on the diagonal, i.e. encourages each row of θ to become sorted in decreasing order. We also add an $L2$ penalty term $\frac{1}{200}\|\theta\|_2^2$ which ensures that the entries of θ do not diverge during training.

A.4.3. Hyperparamaters

We used 100 epochs to train the models, with the first epoch used as burn-in to warm up the CPU or GPU (i.e. the first epoch is excluded from the time measurement). We used a temperature of $\tau = 100.0$ for `NeuralSort` and $\tau = 0.03, d = |\cdot|^2$ for `SoftSort`. The entries of θ are initialized uniformly at random in $[-1, 1]$. A momentum optimizer with learning rate 10 and momentum 0.5 was used. With these settings, 100 epochs are enough to sort each row of θ in decreasing order perfectly for $n = 4000$.

Note that since the goal is to benchmark the operator's speeds, performance on the Spearman rank correlation metric is anecdotal. However, we took the trouble of tuning the hyperparameters and the optimizer to make the learning setting as realistic as possible, and to ensure that the entries in θ are not diverging (which would negatively impact and confound the performance results). Finally, note that the learning problem is trivial, as a pointwise loss such as $\sum_{i=1}^{20}\sum_{j=1}^{n}(\theta_{ij} + j)^2$ sorts the rows of θ without need for the `NeuralSort` or `SoftSort` operator. However, this bare-bones task exposes the computational performance of the `NeuralSort` and `SoftSort` operators.

A.4.4. Learning Curves

In Figure 2 we show the learning curves for $N = 4000$; the Spearman correlation metric is plotted against epoch. We see that `SoftSort` with $d = |\cdot|^2$ and `NeuralSort` have almost identical learning curves. Interestingly, `SoftSort` with $d = |\cdot|$ converges more slowly.

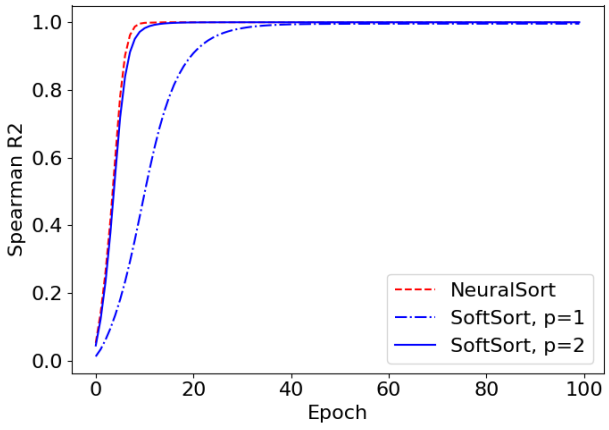


Figure 2. Learning curves for `SoftSort` with $d = |\cdot|^p$ for $p \in \{1, 2\}$, and `NeuralSort`, on the speed comparison task.

A.4.5. NEURALSORT PERFORMANCE IMPROVEMENT

We found that the `NeuralSort` implementations provided by (Grover et al., 2019) in both TensorFlow and PyTorch have complexity $\mathcal{O}(n^3)$. Indeed, in their TensorFlow implementation (Figure 4), the complexity of the following line is $\mathcal{O}(n^3)$:

```
B = tf.matmul(A_s, tf.matmul(one,
              tf.transpose(one)))
```

since the three matrices multiplied have sizes $n \times n$, $n \times 1$, and $1 \times n$ respectively. To obtain $\mathcal{O}(n^2)$ complexity we associate differently:

```
B = tf.matmul(tf.matmul(A_s, one),
              tf.transpose(one))
```

The same is true for their PyTorch implementation (Figure 5). This way, we were able to speed up the implementations provided by (Grover et al., 2019).

A.4.6. PYTORCH RESULTS

In Figure 3 we show the benchmarking results for the PyTorch framework (Paszke et al., 2017). These are analogous to the results presented in Figure 6) of the main text. The results are similar to those for the TensorFlow framework, except that for PyTorch, `NeuralSort` runs out of memory on CPU for $n = 3600$, on GPU for $n = 3900$, and `SoftSort` runs out of memory on CPU for $n = 3700$.

A.4.7. HARDWARE SPECIFICATION

We used a GPU V100 and an n1-highmem-2 (2 vCPUs, 13 GB memory) Google Cloud instance to perform the speed comparison experiment.

We were also able to closely reproduce the GPU results on an Amazon EC2 p2.xlarge instance (4 vCPUs, 61 GB memory) equipped with a GPU Tesla K80, and the CPU results on an Amazon EC2 c5.2xlarge instance (8 vCPUs, 16 GB memory).

B. Proof of Proposition 1

First we recall the Proposition:

Proposition. For both $f = \texttt{SoftSort}_\tau^d$ (with any d) and $f = \texttt{NeuralSort}_\tau$, the following identity holds:

$$f(s) = f(\texttt{sort}(s))P_{\texttt{argsort}(s)} \tag{1}$$

To prove the proposition, we will use the following two Lemmas:

Lemma 1 *Let $P \in \mathbb{R}^{n\times n}$ be a permutation matrix, and let $g : \mathbb{R}^k \to \mathbb{R}$ be any function. Let $G : \underbrace{\mathbb{R}^{n\times n} \times \cdots \times \mathbb{R}^{n\times n}}_{k \text{ times}} \to \mathbb{R}^{n\times n}$ be the pointwise application of g, that is:*

$$G(A_1, \ldots, A_k)_{i,j} = g((A_1)_{i,j}, \ldots, (A_k)_{i,j}) \tag{2}$$

Then the following identity holds for any $A_1, \ldots, A_k \in \mathbb{R}^{n\times n}$:

$$G(A_1, \ldots, A_k)P = G(A_1P, \ldots, A_kP) \tag{3}$$

Proof of Lemma 1. Since P is a permutation matrix, multiplication to the right by P permutes columns according to some permutation, i.e.

$$(AP)_{i,j} = A_{i,\pi(j)} \tag{4}$$

for some permutation π and any $A \in \mathbb{R}^{n\times n}$. Thus, for any fixed i, j:

$$\begin{aligned}
&(G(A_1, \ldots, A_k)P)_{i,j} \\
&\overset{(i)}{=} G(A_1, \ldots, A_k)_{i,\pi(j)} \\
&\overset{(ii)}{=} g((A_1)_{i,\pi(j)}, \ldots, (A_k)_{i,\pi(j)}) \\
&\overset{(iii)}{=} g((A_1P)_{i,j}, \ldots, (A_kP)_{i,j}) \\
&\overset{(iv)}{=} G(A_1P, \ldots, A_kP)_{i,j}
\end{aligned}$$

where $(i), (iii)$ follow from Eq. 4, and $(ii), (iv)$ follow from Eq. 2. This proves the Lemma. ■

Lemma 2 *Let $P \in \mathbb{R}^{n\times n}$ be a permutation matrix, and $\sigma = \texttt{softmax}$ denote the row-wise softmax, i.e.:*

$$\sigma(A)_{i,j} = \frac{\exp\{A_{i,j}\}}{\sum_k \exp\{A_{i,k}\}} \tag{5}$$

Then the following identity holds for any $A \in \mathbb{R}^{n\times n}$:

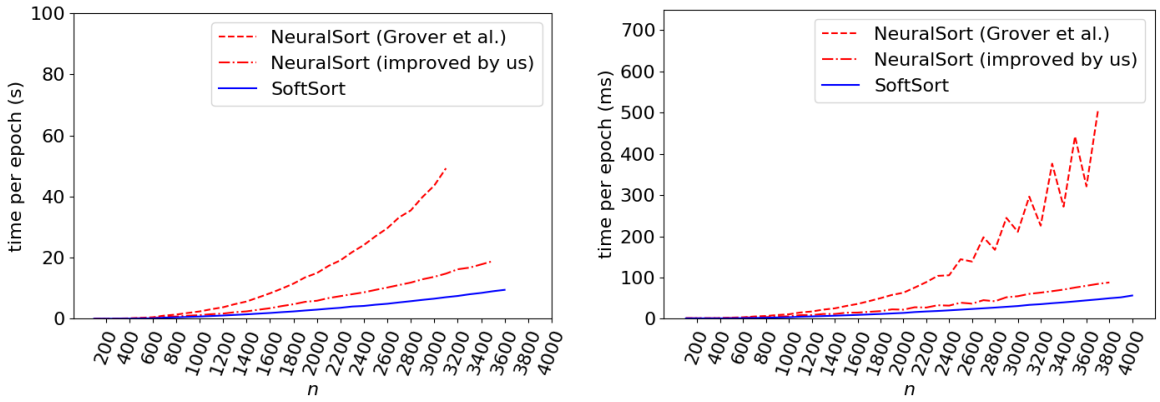
$$\sigma(A)P = \sigma(AP) \tag{6}$$

Proof of Lemma 2. As before, there exists a permutation π such that:

$$(BP)_{i,j} = B_{i,\pi(j)} \tag{7}$$

for any $B \in \mathbb{R}^{n\times n}$. Thus for any fixed i, j:

$$\begin{aligned}
&(\sigma(A)P)_{i,j} \\
&\overset{(i)}{=} \sigma(A)_{i,\pi(j)} \\
&\overset{(ii)}{=} \frac{\exp\{A_{i,\pi(j)}\}}{\sum_k \exp\{A_{i,\pi(k)}\}} \\
&\overset{(iii)}{=} \frac{\exp\{(AP)_{i,j}\}}{\sum_k \exp\{(AP)_{i,k}\}} \\
&\overset{(iv)}{=} \sigma(AP)_{i,j}
\end{aligned}$$



(a) CPU speed vs input dimension n

(b) GPU speed vs input dimension n

Figure 3. 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. Note that CPU plot y-axis is in seconds (s), while GPU plot y-axis is in milliseconds (ms). Implementation in PyTorch.

where $(i), (iii)$ follow from Eq. 7 and $(ii), (iv)$ follow from the definition of the row-wise softmax (Eq. 5). This proves the Lemma. ■

We now leverage the Lemmas to provide proofs of Proposition 1 for each operator. To unclutter equations, we will denote by $\sigma = \texttt{softmax}$ the row-wise softmax operator.

Proof of Proposition 1 for `SoftSort`. We have that:

$$\begin{aligned}
&\texttt{SoftSort}^d_\tau(\texttt{sort}(s))P_{\texttt{argsort}(s)} \\
&\overset{(i)}{=}\sigma\Big(\frac{-d(\texttt{sort}(\texttt{sort}(s))\mathbb{1}^T, \mathbb{1}\texttt{sort}(s)^T)}{\tau}\Big)P_{\texttt{argsort}(s)} \\
&\overset{(ii)}{=}\sigma\Big(\frac{-d(\texttt{sort}(s)\mathbb{1}^T, \mathbb{1}\texttt{sort}(s)^T)}{\tau}\Big)P_{\texttt{argsort}(s)}
\end{aligned}$$

where (i) follows from the definition of the `SoftSort` operator (Eq. 4) and (ii) follows from the idempotence of the `sort` operator, i.e. $\texttt{sort}(\texttt{sort}(s)) = \texttt{sort}(s)$. Invoking Lemma 2, we can push $P_{\texttt{argsort}(s)}$ into the softmax:

$$=\sigma\Big(\frac{-d(\texttt{sort}(s)\mathbb{1}^T, \mathbb{1}\texttt{sort}(s)^T)}{\tau}P_{\texttt{argsort}(s)}\Big)$$

Using Lemma 1 we can further push $P_{\texttt{argsort}(s)}$ into the pointwise d function:

$$=\sigma\Big(\frac{-d(\texttt{sort}(s)\mathbb{1}^T P_{\texttt{argsort}(s)}, \mathbb{1}\texttt{sort}(s)^T P_{\texttt{argsort}(s)})}{\tau}\Big)$$

Now note that $\mathbb{1}^T P_{\texttt{argsort}(s)} = \mathbb{1}^T$ since P is a permutation matrix and thus the columns of P add up to 1. Also, since $\texttt{sort}(s)^T = P_{\texttt{argsort}(s)}s$ then $\texttt{sort}(s)^T P_{\texttt{argsort}(s)} = s^T P^T_{\texttt{argsort}(s)} P_{\texttt{argsort}(s)} = s^T$ since $P^T_{\texttt{argsort}(s)} P_{\texttt{argsort}(s)} = I$ (because $P_{\texttt{argsort}(s)}$ is a permutation matrix). Hence we arrive at:

$$\begin{aligned}
&=\sigma\Big(\frac{-d(\texttt{sort}(s)\mathbb{1}^T, \mathbb{1}s^T)}{\tau}\Big) \\
&=\texttt{SoftSort}^d_\tau(s)
\end{aligned}$$

which proves the proposition for `SoftSort`. ■

Proof of Proposition 1 for `NeuralSort`. For any fixed i, inspecting row i we get:

$$\begin{aligned}
&(\texttt{NeuralSort}_\tau(\texttt{sort}(s))P_{\texttt{argsort}(s)})[i,:] \\
&\overset{(i)}{=}(\texttt{NeuralSort}_\tau(\texttt{sort}(s))[i,:])P_{\texttt{argsort}(s)} \\
&\overset{(ii)}{=}\sigma\Big(\frac{(n+1-2i)\texttt{sort}(s)^T - \mathbb{1}^T A^T_{\texttt{sort}(s)}}{\tau}\Big)P_{\texttt{argsort}(s)}
\end{aligned}$$

where (i) follows since row-indexing and column permutation trivially commute, i.e. $(BP)[i,:] = (B[i,:])P$ for any $B \in \mathbb{R}^{n\times n}$, and (ii) is just the definition of `NeuralSort` (Eq. 3, taken as a row vector).

Using Lemma 2 we can push $P_{\texttt{argsort}(s)}$ into the softmax, and so we get:

$$\begin{aligned}
&= \sigma\Big(((n+1-2i)\texttt{sort}(s)^T P_{\texttt{argsort}(s)} \\
&\qquad - \mathbb{1}^T A^T_{\texttt{sort}(s)} P_{\texttt{argsort}(s)})/\tau\Big)
\end{aligned} \tag{8}$$

Now note that $\texttt{sort}(s)^T P_{\texttt{argsort}(s)} = s^T$ (as we showed in the proof of the Proposition for `SoftSort`). As for the subtracted term, we have, by definition of $A_{\texttt{sort}(s)}$:

$$\begin{aligned}
&\mathbb{1}^T A^T_{\texttt{sort}(s)} P_{\texttt{argsort}(s)} \\
&=\mathbb{1}^T|\texttt{sort}(s)\mathbb{1}^T - \mathbb{1}\texttt{sort}(s)^T|P_{\texttt{argsort}(s)}
\end{aligned}$$

Applying Lemma 1 to the pointwise absolute value, we can push $P_{\texttt{argsort}(s)}$ into the absolute value:

$$= \mathbb{1}^T |\texttt{sort}(s)\mathbb{1}^T P_{\texttt{argsort}(s)} - \mathbb{1}\texttt{sort}(s)^T P_{\texttt{argsort}(s)}|$$

Again we can simplify $\texttt{sort}(s)^T P_{\texttt{argsort}(s)} = s^T$ and $\mathbb{1}^T P_{\texttt{argsort}(s)} = \mathbb{1}^T$ to get:

$$= \mathbb{1}^T |\texttt{sort}(s)\mathbb{1}^T - \mathbb{1}s^T| \tag{9}$$

We are almost done. Now just note that we can replace $\texttt{sort}(s)$ in Eq. 9 by s because multiplication to the left by $\mathbb{1}^T$ adds up over each column of $|\texttt{sort}(s)\mathbb{1}^T - \mathbb{1}s^T|$ and thus makes the `sort` irrelevant, hence we get:

$$\begin{aligned} &= \mathbb{1}^T |s\mathbb{1}^T - \mathbb{1}s^T| \\ &= \mathbb{1}^T A_s \end{aligned}$$

Thus, putting both pieces together into Eq. 8 we arrive at:

$$\begin{aligned} &= \sigma\Big(\frac{(n+1-2i)s - \mathbb{1}^T A_s}{\tau}\Big) \\ &= \texttt{NeuralSort}_\tau(s)[i,:] \end{aligned}$$

which proves Proposition 1 for `NeuralSort`. ■

C. Proof of Proposition 2

First, let us recall the proposition:

Proposition. Let $k = 1$ and $\widehat{P}$ be the differentiable kNN operator using $\texttt{SoftSort}_2^{|\cdot|}$. If we choose the embedding function Φ to be of norm 1, then

$$\widehat{P}(\hat{y}|\hat{x}, X, Y) = \sum_{i:y_i=\hat{y}} e^{\Phi(\hat{x})\cdot\Phi(x_i)} \Big/ \sum_{i=1\ldots n} e^{\Phi(\hat{x})\cdot\Phi(x_i)}$$

Proof. Since $k = 1$, only the first row of the `SoftSort` matrix is used in the result. Recall that the elements of the first row are the `softmax` over $-|s_i - s_{[1]}|$. Given that $s_{[1]} \geq s_i \ \forall i$, we can remove the negative absolute value terms. Because of the invariance of `softmax` for additive constants, the $s_{[1]}$ term can also be cancelled out.

Furthermore, since the embeddings are normalized, we have that $s_i = -\|\Phi(x_i) - \Phi(\hat{x})\|^2 = 2\,\Phi(x_i)\cdot\Phi(\hat{x}) - 2$. When we take the `softmax` with temperature 2, we are left with values proportional to $e^{\Phi(x_i)\cdot\Phi(\hat{x})}$. Finally, when the vector is multiplied by $\mathbb{I}_{Y=\hat{y}}$ we obtain the desired identity. ■

D. Magnitude of Matrix Entries

The outputs of the `NeuralSort` and `SoftSort` operators are $n \times n$ (unimodal) row-stochastic matrices, i.e. each of their rows add up to one. In section 4.1 we compared the mathematical complexity of equations 3 and 4 defining both operators, but how do these operators differ *numerically*? What can we say about the magnitude of the matrix entries?

For the $\texttt{SoftSort}^{|\cdot|}$ operator, we show that the values of a given row come from Laplace densities evaluated at the s_j. Concretely:

Proposition 1 *For any $s \in \mathbb{R}^n$, $\tau > 0$ and $1 \leq i \leq n$, it holds that $\texttt{SoftSort}_\tau^{|\cdot|}(s)[i,j] \ \propto_j \ \phi_{\textit{Laplace}(s_{[i]},\tau)}(s_j)$. Here $\phi_{\textit{Laplace}(\mu,b)}$ is the density of a Laplace distribution with location parameter $\mu \geq 0$ and scale parameter $b > 0$.*

Proof. This is trivial, since:

$$\texttt{SoftSort}_\tau^{|\cdot|}(s)[i,j] = \underbrace{\frac{1}{\sum_{k=1}^n \exp\{-|s_{[i]} - s_k|/\tau\}}}_{c_i} \underbrace{\exp\{-|s_{[i]} - s_j|/\tau\}}_{\phi_{\text{Laplace}(s_{[i]},\tau)}(s_j)}$$

where c_i a constant which does not depend on j (specifically, the normalizing constant for row i). □

In contrast, for the `NeuralSort` operator, we show that in the prototypical case when the values of s are equally spaced, the values of a given row of the `NeuralSort` operator come from *Gaussian* densities evaluated at the s_j. This is of course not true in general, but we believe that this case provides a meaningful insight into the `NeuralSort` operator. Without loss of generality, we will assume that the s_j are sorted in decreasing order (which we can, as argued in section 4.1); this conveniently simplifies the indexing. Our claim, concretely, is:

Proposition 2 *Let $a, b \in \mathbb{R}$ with $a > 0$, and assume that $s_k = b - ak \ \forall k$. Let also $\tau > 0$ and $i \in \{1, 2, \ldots, n\}$. Then $\texttt{NeuralSort}_\tau(s)[i,j] \propto_j \phi_{\mathcal{N}(s_i, a\tau)}(s_j)$. Here $\phi_{\mathcal{N}(\mu,\sigma^2)}$ is the density of a Gaussian distribution with mean $\mu \geq 0$ and variance $\sigma^2 > 0$.*

Proof. The i,j-th logit of the `NeuralSort` operator be-

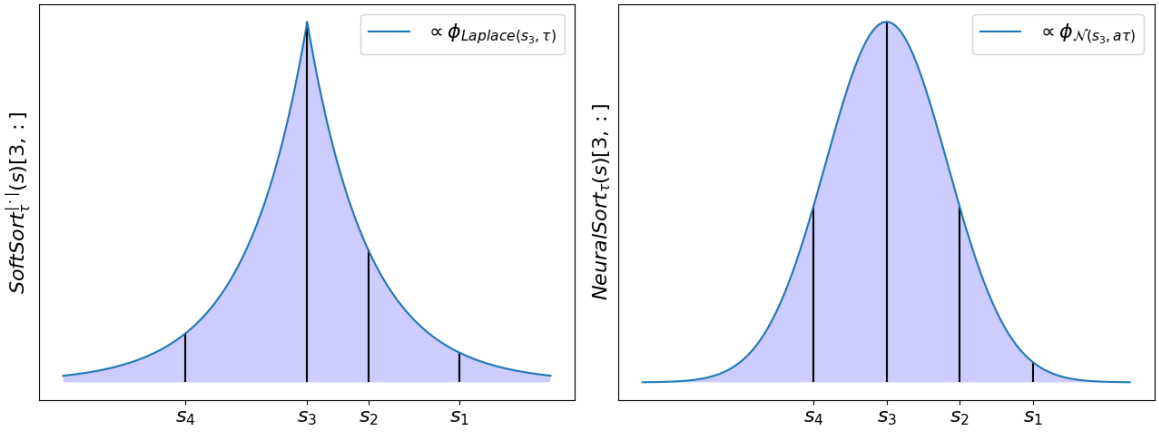


Figure 4. Rows of the $\texttt{SoftSort}^{|\cdot|}$ operator are proportional to Laplace densities evaluated at the s_j, while under the equal-spacing assumption, rows of the $\texttt{NeuralSort}$ operator are proportional to Gaussian densities evaluated at the s_j. Similarly, rows of the $\texttt{SoftSort}^{|\cdot|^2}$ operator are proportional to Gaussian densities evaluated at the s_j (plot not shown).

fore division by the temperature τ is (by Eq. 3):

$$\begin{aligned}
&(n+1-2i)s_j - \sum_{k=1}^{n} |s_k - s_j| \\
&= (n+1-2i)(b-aj) - \sum_{k=1}^{n} |b-ak-b+aj| \\
&= (n+1-2i)(b-aj) - a\sum_{k=1}^{n} |j-k| \\
&= (n+1-2i)(b-aj) - a\frac{j(j-1)}{2} \\
&\qquad\qquad - a\frac{(n-j)(n-j+1)}{2} \\
&= -a(i-j)^2 + a(i^2 - \frac{n^2}{2} - \frac{n}{2}) - b(2i-n-1) \\
&= -\frac{(s_i-s_j)^2}{a} + \underbrace{a(i^2 - \frac{n^2}{2} - \frac{n}{2}) - b(2i-n-1)}_{\Delta_i}
\end{aligned}$$

where Δ_i is a constant that does not depend on j. Thus, after dividing by τ and taking softmax on the i-th row, Δ_i/τ vanishes and we are left with:

$$\texttt{NeuralSort}_\tau[i,j] = \underbrace{\frac{1}{\sum_{k=1}^{n} \exp\{-(s_i-s_k)^2/(a\tau)\}}}_{c_i} \underbrace{\exp\{-(s_i-s_j)^2/(a\tau)\}}_{\phi_{\mathcal{N}(s_i,a\tau)}(s_j)}$$

where c_i a constant which does not depend on j (specifically, the normalizing constant for row i). □

Gaussian densities can be obtained for $\texttt{SoftSort}$ too by choosing $d = |\cdot|^2$. Indeed:

Proposition 3 *For any $s \in \mathbb{R}^n$, $\tau > 0$ and $1 \le i \le n$, it holds that $\texttt{SoftSort}_\tau^{|\cdot|^2}(s)[i,j] \propto_j \phi_{\mathcal{N}(s_{[i]},\tau)}(s_j)$.*

Proof. This is trivial, since:

$$\texttt{SoftSort}_\tau^{|\cdot|^2}(s)[i,j] = \underbrace{\frac{1}{\sum_{k=1}^{n} \exp\{-(s_{[i]}-s_k)^2/\tau\}}}_{c_i} \underbrace{\exp\{-(s_{[i]}-s_j)^2/\tau\}}_{\phi_{\mathcal{N}(s_{[i]},\tau)}(s_j)}$$

where c_i a constant which does not depend on j (specifically, the normalizing constant for row i). □

Figure 4 illustrates propositions 1 and 2. As far as we can tell, the Laplace-like and Gaussian-like nature of each operator is neither an advantage nor a disadvantage; as we show in the experimental section, both methods perform comparably on the benchmarks. Only on the speed comparison task does it seem like $\texttt{NeuralSort}$ and $\texttt{SoftSort}^{|\cdot|^2}$ outperform $\texttt{SoftSort}^{|\cdot|}$.

Finally, we would like to remark that $\texttt{SoftSort}^{|\cdot|^2}$ does not recover the $\texttt{NeuralSort}$ operator, not only because Proposition 2 only holds when the s_i are equally spaced, but also because even when they *are* equally spaced, the Gaussian in Proposition 2 has variance $a\tau$ whereas the Gaussian in Proposition 3 has variance τ. Concretely: we can only make the claim that $\texttt{SoftSort}_{a\tau}^{|\cdot|^2}(s) = \texttt{NeuralSort}_\tau(s)$ when s_i are equally spaced at distance a. As soon as the spacing between the s_i changes, we need to *change the temperature* of the $\texttt{SoftSort}^{|\cdot|^2}$ operator to match the $\texttt{NeuralSort}$ operator again. Also, the fact that the $\texttt{SoftSort}^{|\cdot|^2}$ and $\texttt{NeuralSort}$ operators agree in this prototypical case for some choice of τ does not mean

that their *gradients* agree. An interesting and under-explored avenue for future work might involve trying to understand how the *gradients* of the different continuous relaxations of the `argsort` operator proposed thus far compare, and whether some gradients are preferred over others. So far we only have empirical insights in terms of learning curves.

E. Sorting Task - Proportion of Individual Permutation Elements Correctly Identified

Table 2 shows the results for the second metric (the proportion of individual permutation elements correctly identified). Again, we report the mean and standard deviation over 10 runs. Note that this is a less stringent metric than the one reported in the main text. The results are analogous to those for the first metric, with `SoftSort` and `NeuralSort` performing identically for all n, and outperforming the method of (Cuturi et al., 2019) for $n = 9, 15$.

F. PyTorch Implementation

In Figure 6 we provide our PyTorch implementation for the $\texttt{SoftSort}^{|\cdot|}$ operator. Figure 5 shows the PyTorch implementation of the `NeuralSort` operator (Grover et al., 2019) for comparison, which is more complex.

Table 2. Results for the sorting task averaged over 10 runs. We report the mean and standard deviation for the *proportion of individual permutation elements correctly identified*.

ALGORITHM	$n = 3$	$n = 5$	$n = 7$	$n = 9$	$n = 15$
DETERMINISTIC NEURALSORT	0.946 ± 0.004	0.911 ± 0.005	$\mathbf{0.882 \pm 0.006}$	$\mathbf{0.862 \pm 0.006}$	$\mathbf{0.802 \pm 0.009}$
STOCHASTIC NEURALSORT	0.944 ± 0.004	0.912 ± 0.004	$\mathbf{0.883 \pm 0.005}$	$\mathbf{0.860 \pm 0.006}$	$\mathbf{0.803 \pm 0.009}$
DETERMINISTIC SOFTSORT	0.944 ± 0.004	0.910 ± 0.005	$\mathbf{0.883 \pm 0.007}$	$\mathbf{0.861 \pm 0.006}$	$\mathbf{0.805 \pm 0.007}$
STOCHASTIC SOFTSORT	0.944 ± 0.003	0.910 ± 0.002	$\mathbf{0.884 \pm 0.006}$	$\mathbf{0.862 \pm 0.008}$	$\mathbf{0.802 \pm 0.007}$
(CUTURI ET AL., 2019, REPORTED)	**0.950**	**0.917**	**0.882**	0.847	0.742

```
def neural_sort(s, tau):
    n = s.size()[1]
    one = torch.ones((n, 1), dtype = torch.float32)
    A_s = torch.abs(s - s.permute(0, 2, 1))
    B = torch.matmul(A_s, torch.matmul(one, torch.transpose(one, 0, 1)))
    scaling = (n + 1 - 2 * (torch.arange(n) + 1)).type(torch.float32)
    C = torch.matmul(s, scaling.unsqueeze(0))
    P_max = (C-B).permute(0, 2, 1)
    sm = torch.nn.Softmax(-1)
    P_hat = sm(P_max / tau)
    return P_hat
```

Figure 5. Implementation of `NeuralSort` in PyTorch as given in (Grover et al., 2019)

```
def soft_sort(s, tau):
    s_sorted = s.sort(descending=True, dim=1)[0]
    pairwise_distances = (s.transpose(1, 2) - s_sorted).abs().neg() / tau
    P_hat = pairwise_distances.softmax(-1)
    return P_hat
```

Figure 6. Implementation of `SoftSort` in PyTorch as proposed by us (with $d = |\cdot|$).

References

Cuturi, M., Teboul, O., and Vert, J.-P. Differentiable ranking and sorting using optimal transport. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 6858–6868. Curran Associates, Inc., 2019.

Grover, A., Wang, E., Zweig, A., and Ermon, S. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M. (eds.), *Computer Vision – ECCV 2016*, pp. 630–645, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., Devito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *Advances in Neural Information Processing Systems 30*, 2017.