5 Broader impact

This submission focuses on foundational and exploratory work, with application to general machine learning techniques. We propose methods to extend the range of operations that can be used in end-to-end differentiable systems, by allowing to include clustering as a differentiable operation in machine learning models. We do not foresee societal consequences that are specifically related to these methods, beyond those that are associated with the field in general.

6 Reproducibility and Licensing

Our experiments use data sets that are already open-sourced and cited in the references. All the code written for this project and used to implement our experiments is available at:

https://github.com/LawrenceMMStewart/DiffClust_NeurIPS2023,

and distributed under a Apache 2.0. license.

7 Limitations

At present, our implementation of Kruskal's algorithm is incompatible with processing very large batch sizes at train time. However, as our method can use mini-batches, it is suitable for training deep learning models on large data sets. For example, we use a batch size of 64 (similar to that for standard classification models) for all experiments detailed in our submission. Other methods that focus on cluster assignment, rather than pairwise clustering, often have an $n \times k$ parametrization, whose efficiency will depend on the comparison between n (batch size) and k (number of clusters).

At inference time this is not the case, since gradients need not be back-propagated hence, any implementation of Kruskal's algorithm can be used such as the union-find implementation. Faster GPU compatible implementations of Kruskal's algorithm for training are certainly possible ; improvements could be made using a binary heap or a GPU parallelized merge sort. In many learning experiments the batch sizes are typically of values $\{32, 64, 128, 256, 512\}$, so our current implementation is sufficient, however, for some specific applications (such as bio-informatics experiments requiring tens of thousands of points in a batch), it would be necessary to improve upon the current implementation. This is mainly an engineering task relating to data-structures and GPU programming, and is beyond the scope of this paper.

Finally, our clustering approach solves a linear program whose argmax solution is a matroid problem with a greedy single-linkage criterion. Whilst we saw that this method is effective for differentiable clustering in both the supervised and semi-supervised setting, there may well be other linkage criteria which also take the form of LP's but whom are more robust to outliers and noise in the train dataset. This line of work is outside the scope of the paper, but is closely related and could help improve our differentiable spanning forests framework.

8 Proofs of Technical Results

Proof of Proposition 1. We show these properties successively

1) We have by Definition 8

$$\bar{L}_{\mathsf{FY}}(\theta, p) = \min_{y \in \mathcal{C}(p)} L_{\mathsf{FY}}(\theta; y)\,.$$

We expand this

$$\bar{L}_{\mathsf{FY}}(\theta, p) = \min_{y \in \mathcal{C}(p)} \big\{F(\theta) - \langle y, \theta\rangle\big\} = F(\theta) - \max_{y \in \mathcal{C}(p)} \langle y, \theta\rangle\,.$$

As required, this implies, following the definitions given

$$\bar{L}_{\mathsf{FY}}(\theta, p) = F(\theta) - F(\theta; p)\,, \quad \text{where} \quad F(\theta; p) = \max_{y \in \mathcal{C}(p)} \langle y, \theta\rangle,$$

2) By linearity of derivatives and 1) above, we have

$$\nabla_\theta \bar{L}_{\mathsf{FY}}(\theta, p) = \nabla_\theta F(\theta) - \nabla_\theta F(\theta; p) = y^*(\theta) - y^*(\theta; p)\,, \quad \text{where} \quad y^*(\theta; p) = \operatorname*{argmax}_{y \in \mathcal{C}(p)} \langle y, \theta\rangle,$$

Since the argmax of a constrained linear optimization problem is the gradient of its value. We note that this property holds almost everywhere (when the argmax is unique), and almost surely for costs with positive, continuous density, which we always assume (e.g. see the following).

3) By linearity of expectation and 1) above, we have

$$\nabla_\theta \bar{L}_{\mathsf{FY},\varepsilon}(\theta, p) = \nabla_\theta \mathbf{E}[F(\theta + \varepsilon Z) - F(\theta + \varepsilon Z; p)] = \nabla_\theta F_\varepsilon(\theta) - \nabla_\theta F_\varepsilon(\theta; p) = y^*_\varepsilon(\theta) - y^*_\varepsilon(\theta; p)\,,$$

using the definition

$$y^*_\varepsilon(\theta; p) = \mathbf{E}[\operatorname*{argmax}_{y \in \mathcal{C}(p)} \langle y, \theta + \varepsilon Z\rangle]\,.$$

□

Proof of Proposition 2. By Jensen's inequality and the definition of the Fenchel-Young loss:

$$\begin{aligned}
\bar{L}_{\mathsf{FY},\varepsilon}(\theta; p) &= \mathbf{E}[\min_{y \in \mathcal{C}(p)} L_{\mathsf{FY}}(\theta + \varepsilon Z; y)] \\
&\le \min_{y \in \mathcal{C}(p)} \mathbf{E}[L_{\mathsf{FY}}(\theta + \varepsilon Z; y)] \\
&= \min_{y \in \mathcal{C}(p)} F_\varepsilon(y) - \langle \theta, y\rangle \\
&\le \min_{y \in \mathcal{C}(p)} L_{\mathsf{FY},\varepsilon}(\theta; y)\,.
\end{aligned}$$

□

9 Algorithms for Spanning Forests

As mentioned in Section 2, both $A^*_k(\Sigma)$ and $M^*_k(\Sigma)$ are calculated using Kruskal's algorithm (Kruskal, 1956). Our implementation of Kruskal's is tailored to our use: we first initialize both $A^*_k(\Sigma)$ and $M^*_k(\Sigma)$ as the identity matrix, and then sort the upper triangular entries of Σ. We build the maximum-weight spanning forest in the usual greedy manner, using $A^*_k(\Sigma)$ to keep track of edges in the forest and $M^*_k(\Sigma)$ to check if an edge can be added without creating a cycle, updating both matrices at each step of the algorithm. Once the forest has k connected components, the algorithm terminates. This is done by keeping track of the number of edges that have been added at any time.

We remark that our implementation takes the form as a single loop, with each step of the loop consisting only of matrix multiplications. For this reason it is fully compatible with auto-differentiation engines, such as JAX (Bradbury et al., 2018), Pytorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016), and suitable for GPU/TPU acceleration. Therefore, our implementation differs from that of the standard Kruskal's algorithm, which used a disjoint union-find data structure (and hence is not compatible with auto-differentiation frameworks).

9.1 Constrained Spanning Forests

As an heuristic way to solve the constrained problem detailed in Section 5, we make the modifications below to our implementation of Kruskal's, under the assumption that M_Ω represents *valid* clustering information (i.e. with no contradiction):

1. **Regularization** (Optional) : It is possible to bias the optimization problem over spanning forests to encourage or discourage edges between some of the nodes, according to the clustering information. Before performing the sort on the upper-triangular of Σ, we add a large value to all entries of Σ_{ij} where $(M_\Omega)_{ij} = 1$, and subtract this same value from all entries of Σ_{ij} where $(M_\Omega)_{ij} = 1$. Entries Σ_{ij} corresponding to where $(M_\Omega)_{ij} = \star$ are left unchanged. This biasing ensures that any edge between points that are constrained to be in the same cluster will always be processed before unconstrained edges. Similarly, any edge between points that are constrained to not be in the same cluster, will be processed after unconstrained edges. In most cases, i.e. when all clusters are represented in the partial information, such as when $\Omega = [n] \times [n]$ (full information), this is not required to solve the constrained linear program, but we have found that this regularization was helpful in practice.

2. **Constraint enforcement** : We ensure that adding an edge does not violate the constraint matrix. In other words, when considering adding the edge (i, j) to the existing forest, we check that none of the points in the connected component of i are forbidden from joining any of the points in the connected component of j. This is implemented using further matrix multiplications and done alongside the existing check for cycles. The exact implementation is detailed in our code base.

10 Existing Literature on Differentiable Clustering

As discussed in Section 1, there exists many approaches which use clustering during gradient based learning, but these approaches typically use clustering in an offline fashion in order to assign labels to points. The following methods aim to learn through a clustering step (i.e. gradients back-propagate through clustering):

Yang et al. (2017) use a bi-level optimization procedure (alternating between optimizing model weights and centroid clusters). They reported attaining 83% label-wise clustering accuracy on MNIST using a fully-connected deep network. Our method differs from this approach as it is allows for end-to-end online learning.

Genevay et al. (2019) cast k-means as an optimal transport problem, and uses entropic regularization for smoothing. Reported a 85% accuracy on MNIST and 25% accuracy on CIFAR-10 with a CNN.

11 Additional Experimental Information

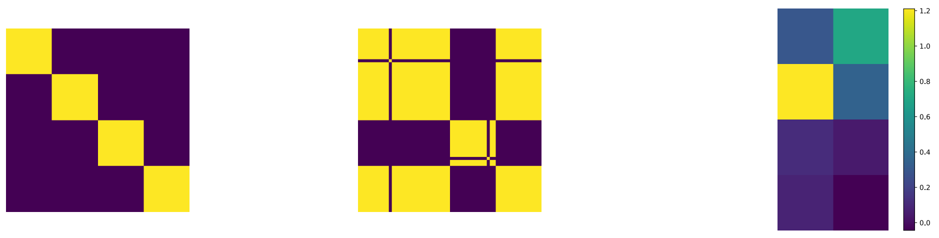

Figure 6: Leftmost figure: M_{signal}, Center Figure: $M_4^*(\Sigma)$, Rightmost Figure: θ after training.

We provide the details of the synthetic denoising experiment depicted in Figure 2 and described in Section 4.1. We create the signal data $X_{\text{signal}} \in \mathbb{R}^{60\times 2}$ by sampling iid. from four isotropic Gaussians (15 points coming from each of the Gaussians) each having a standard deviation of 0.2. We randomly sample the means of the four Gaussians; for the example seen in Section 4.1 the sampled means were:

$$\begin{pmatrix} 0.97627008 & 4.30378733 \\ 2.05526752 & 0.89766366 \\ -1.52690401 & 2.91788226 \\ -1.24825577 & 7.83546002 \end{pmatrix}.$$

Let Σ_{signal} be the pairwise euclidean similarity matrix corresponding to X_{signal}, and furthermore let $M_{\text{signal}} := M_4^*(\Sigma_{\text{signal}})$ be the equivalence matrix corresponding to the signal (M_{signal} will be the target equivalence matrix to learn).

We append an additional two ‘noise dimensions’ to X_{signal} in order to form the train data $X \in \mathbb{R}^{60\times 4}$, where the noise entries were sampled iid from a continuous unit uniform distribution. Similarly letting Σ be the pairwise euclidean similarity matrix corresponding to X, we calculate $M_4^*(\Sigma) \neq M_{\text{signal}}$. Both the matrices M_{signal} and $M_4^*(\Sigma)$ are depicted in Figure 6 ; we remark that adding the noise dimensions leads to most points being assigned one of two clusters, and two points being isolated alone in their own clusters. We also create a validation set of equal size (in exactly the same manner as the train set), to ensure the model has not over-fitted to the train set.

The goal of the experiment is to learn a linear transformation of the data that recovers M_{signal} i.e. a denoising, by minimizing the partial loss. There are multiple solutions to this problem, the most obvious being a transformation that removes the last two noise columns from X:

$$\theta^* = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad \text{for which} \quad X\theta^* = X_{\text{signal}}$$

For any $\theta \in \mathbb{R}^{4\times 2}$, we define Σ_θ to be the pairwise similarity matrix corresponding to $X\theta$, and $M_4^*(\Sigma_\theta)$ to its corresponding equivalence matrix. Then the problem can be summarized as:

$$\min_{\theta \in \mathbb{R}^{4\times 2}} \bar{L}_{\mathsf{FY},\varepsilon}(\Sigma_\theta, M_{\text{signal}}). \tag{1}$$

We initialized θ from a standard Normal distribution, and minimized the partial loss via stochastic gradient descent, with a learning rate of 0.01 and batch size 32.

For perturbations, we took $\varepsilon = 0.1$ and $B = 1000$, where ε denotes the noise amplitude in randomized smoothing and B denotes the number of samples in the Monte-Carlo estimate. The validation clustering error converged to zero after 25 gradient batches. We verify that the θ attained from training is indeed learning to remove the noise dimensions (see Figure 6).

11.1 Supervised Differentiable Clustering

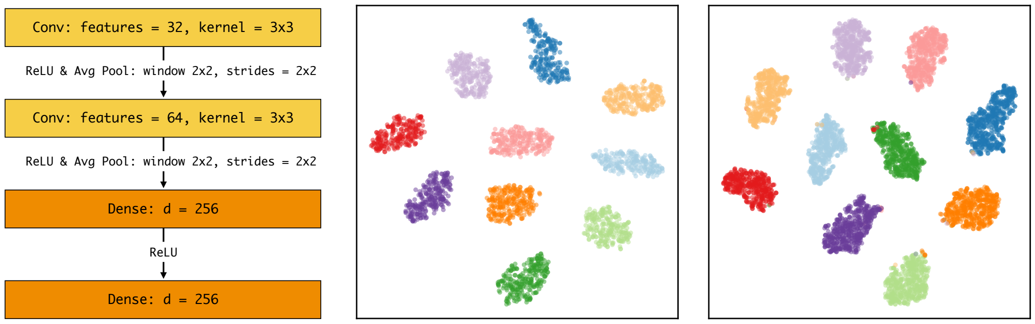

Figure 7: **(left)** Architecture of the CNN, **(middle)** t-SNE visualization of train data embeddings, **(right)** tSNE visualization of validation data embeddings.

As mentioned in Section 4.1, our method is able to cluster classical data sets such as MNIST and Fashion MNIST. We trained a CNN with the LeNet-5 architecture LeCun et al. (1998) using our proposed partial loss as the objective function. The exact details of the CNN architecture are depicted in Figure 7. For this experiment and all experiments described below, we trained on a single Nvidia V100 GPU ; training the CNN with our proposed pipeline took < 15 minutes.

The model was trained for 30k gradient steps on mini-batches of size 64. We used the Adam optimizer (Kingma and Ba, 2015) with learning rate $\eta = 3 \times 10^{-4}$, momentum parameters $(\beta_1, \beta_2) = (0.9, 0.999)$, and an ℓ_2 weight decay of 10^{-4}. We validated / tested the model using the zero-one error between the true equivalence matrix and the equivalence matrix corresponding to the output of the model. We used an early stopping of 10k steps (i.e. training was stopped if the validation clustering error did not improve over 10k steps). For efficiency (and parallelization), we also computed this clustering error batch-wise with batch-size 64. As stated in Section 4.1, we attained a batch-wise clustering precision of 0.99 for MNIST and 0.96 on Fashion MNIST.

The t-SNE visualizations of the embedding space of the model trained on MNIST for a collection of train and validation data points are depicted in Figure 7. It can be seen that the model has learnt a distinct cluster for each of the ten classes.

In similar fashion, we trained a ResNet (He et al., 2016) on the Cifar-10 data set. The exact model architecture is similar to that of ResNet-50, but with minor modifications to the input convolutions for compatibility with the dimensions of Cifar images, and is detailed in the code base.

The training procedure was identical to that of the CNN, except the model was trained for 75k steps (with early stopping), and used the standard data augmentation methods for Cifar-10, namely: a combination of four-pixel padding, random flip followed by a random crop. As mentioned in Section 4.1, the model achieved a batch-wise clustering test precision of 0.933.

11.2 Semi-Supervised Differentiable Clustering

As mentioned in Section 4.2, we show that our method is particularly useful in settings where labelled examples are scarce, even in the particularly challenging case of having no labelled examples for some classes. Our approach allows a model to leverage the semantic information of unlabeled examples when trying to predict a target equivalence matrix M_Ω ; this is owing to the fact that the prediction of a class for a single point depends on the values of all other points in the batch, which is in general not the case for more common problems such as classification and regression.

To demonstrate the performance of our approach, we assess our method on two tasks:

1. **Semi-Supervised Clustering:** Train a model to learn an embedding of the data which leads to a good clustering error. We can compare our methodology to that of a baseline model trained using the cross-entropy loss. This is to check that our model has leveraged information from the unlabeled data and that our partial loss is indeed leading to good clustering performance.
2. **Downstream Classification:** Assess the trained model's capacity to serve as a backbone in a downstream classification task (transfer learning), where its weights are frozen and a linear layer is trained on top of the backbone.

We describe our data processing for both of these tasks below.

11.2.1 Data Sets

In our semi-supervised learning experiments, we divided the standard MNIST and Cifar-10 train splits in the following manner:

- We create a balanced hold-out data set consisting of 1k images (100 images from each of the 10 classes). This hold-out data set will be used to assess the utility of the frozen clustering model on a downstream classification problem.
- From the remaining 59k images, we select a labeled train set of n_ℓ points (detailed in Section 4.2). Our experiments also vary $k_w \in \{0, 3, 6\}$, the number of digits to withhold all labels from. For example, if $k_w = 3$, then the labels for the images corresponding to digits $\{0, 1, 2\}$ will never appear in the labeled train data.

11.2.2 Semi-Supervised Clustering Task

For each of the choices of n_ℓ and k_w, we train the architectures described in Section 11.1 using the following two approaches:

1. **Ours:** The model is trained on mini-batches, where half the batch is labeled data and half the batch is unlabeled data (i.e. a semi-supervised learning regime), to minimize the partial loss.
2. **Baseline:** The baseline model shares the same architecture as that described in Section 11.1, but with an additional dense layer with output dimension 10 (the number of classes). We train the model using mini-batches consisting of labeled points, minimizing the cross-entropy loss. The training regime is fully-supervised learning (classification). The baseline backbone refers to all of the model, minus the dense output layer.

Both models were trained with mini-batches of size 64, with points sampled uniformly without replacement. All hyper-parameters and optimization metrics were identical to those detailed in Section 4.1. For MNIST, we repeated training for each model with five different random seeds $s \in [5]$ (and with three random seeds $s \in [3]$ for Cifar), in order to report population statistics on the clustering error.

11.2.3 Transfer-Learning: Downstream Classification

In this task both models are frozen, and their utility as a foundational backbone is assessed on a downstream classification task using the hold-out data set. We train a linear (a.k.a dense) layer on top of both models using the cross-entropy loss. We refer to this linear layer as the downstream head. Training this linear head is equivalent to performing multinomial logistic regression on the features of the model.

To optimize the weights of the linear head we used the SAGA optimizer (Defazio et al., 2014). The results are depicted in Figure 4. It can be seen that training a CNN backbone using our approach with just 250 labels leads to better downstream classification performance than the baseline trained with 5000 labels. It is worth remarking that the baseline backbone was trained on the same objective function (cross-entropy) and task (classification) as the downstream problem, which is not the case for the backbone corresponding to our approach. This highlights how learning 'cluster-able embeddings' and leveraging unlabeled data can be desirable for transfer learning.