DENOISING WITH A PARTITIONED AFFINITY KERNEL ALLOWS EFFICIENT END-TO-END DIFFERENTIABLE CLUSTERING

A PREPRINT

Keira Wiechecki **Jade Zaslavsky** **Margaux Failla** **Lionel Christiaen**

January 25, 2024

ABSTRACT

Here we introduce Denoising by Deep learning of a Partitoned Weighted Affinity Kernel (DeeP-WAK).

1 Introduction

Fundamentally, unsupervised learning is a problem of finding the latent space of a data set. Denoising, compression, and clustering all ultimately attempt to solve this problem, but approach it from different directions. Denoising aims to remove spurious dimensions whereas compression looks to select informative dimensions. The connection to clustering is less obvious, but no less foundational[3].

At its heart, clustering is an optimization problem of how to partition a data set given some loss function.

There are a few ways to conceptualize this isomorphism.

Clustering as principal components in latent space Data are sparse in latent space. We could think of a cluster as a vector along which data are relatively dense. Though it might naïvely be expected that enforcing orthogonality would be a problem, it's important to keep in mind that latent space is *really high dimensional*. In practice most vectors can be treated as "almost orthogonal".

Clustering as (lossy) compression A cluster can be thought of as an injection of a subset of the data onto some representative value/vector. This is essentially a decomposition of the data into a 1-hot incidence matrix of cluster assignments and a matrix of central cases. As an illustration, consider data $X \in \mathbb{R}^{m \times n}$ where n is the number of data points and m is the dimensionality. We cluster X into k clusters. We obtain a matrix of central cases for each cluster $C \in \mathbb{R}^{k \times m}$. The incidence matrix is given by $K \in \mathbb{B}^{n \times k}$. We now have a lossy approximation of X with the matrix decomposition

$$X \approx KC \tag{1}$$

Clustering as denoising Compression loss isn't always a problem. Identifying informative latent features means throwing out uninformative latent features.

Clustering as sparse dictionary learning Optimizing (1) is essentally the goal of sparse dictionary learning, where C is the dictionary and K is the key.

Clustering as the optimal way of labeling a data set I propose

Clusters as attractor states One of the most beautiful results from information theory is that *prediction is thermodynamic work*. Consider

Though applications of deep learning to classification is well established, self-supervised classification has been much less thoroughly explored. Deep clustering is an active area of research[4]. Similarly to DeepCluE[2], we use an ensemble clustering method. But rather than creating a single consensus partition, we aim to maximise independence between submodels.

Like ClAM[5] it is end-to-end differentiable. Unlike ClAM,

1.1 Notation

Capital letters indicate matrices. Subscripts indicate indices. Superscripts indicate dimensionality. A circumflex indicates a reconstruction of data by a predictor. Lowercase Greek letters indicate tunable parameters. Capital Greek letters indicate lists of parameters for multiple models. Boldface Greek letters indicate parameter spaces.

$\overset{n\to m}{\texttt{function}}$ indicates a layer with input dimension n, output dimension m, and activation function `function`.

$\odot$ is the Hadamard product.

$\bigoplus_{i=k}^{n} \texttt{expression}$ indicates mapping `expression` over $i \in \{k : n\}$.

$\bigoplus_{x,y}^{X,Y} \texttt{expression}$ indicates mapping `expression` over $(x \in X, y \in Y)$.

$\texttt{List}(\texttt{Type}, n)$ indicates a list of n elements of `Type`.

We will sometimes find it necessary to distinguish between a model, a model's architecture, and a model's parameters. $f : \texttt{Model}(n, m)$ indicates a model $f : \mathbb{R}^n \to \mathbb{R}^m$.

$\mathcal{F} : \texttt{Arch}(n, m)$ indicates an architecture for a model $\texttt{Model}(n, m)$.

$\theta : \texttt{Params}(\mathcal{F})$ indicates the parameters for $\mathcal{F}$.

We use the notation $\mathcal{F}(\theta)$ to indicate a model architecture $\mathcal{F}$ parameterized by θ. $\mathcal{F}(\theta)(X)$ indicates passing data X to the model $\mathcal{F}(\theta)$. We write this as a curried function to emphasize that $\mathcal{F}(\theta)$ is stateful.

1.2 `noise2self`

Batson & Royer[1] identify a class of denoising functions which can be optimised using only unlabeled noisy data. Moreover, this denoising

Let $J \in \mathcal{J}$ be independent partitions of noisy data X. Let $\mathcal{F}(\theta)$ be a family of predictors of X_J with tunable parameters $\theta \in \mathbf{\Theta}$ that depends on its complement X_{J^C}

$$\hat{X}_J = \mathcal{F}(\theta)(X_{J^C}) \tag{2}$$

In other words, $\mathcal{F}$ predicts each data point X_J from some subset of the data excluding X_J.

The optimal θ is given by

$$\underset{\theta}{\overset{\mathbf{\Theta}}{\texttt{noise2self}}}[\mathcal{F}(\theta), X] := \underset{\theta}{\overset{\mathbf{\Theta}}{\texttt{argmin}}}[\sum_{J}^{\mathcal{J}} \mathbb{E}[X_J - \mathcal{F}(\theta)(X_{J^C})]^2] \tag{3}$$

1.3 Diffusion with weighted affinity kernels

Our choice of $\mathcal{F}$ is adapted from DEWAKSS[6]. The parameters we want to tune generate a graph G from embeddings E. The adjacency matrix of any graph can be treated as a transition matrix (or weighted affinity kernel) by setting the diagonal to 0 and normalizing columns to sum to 1. We call this the `WAK` function.

Function $\texttt{WAK}(G)$
  Input: $G : \mathbb{R}^{n \times n}$
  Output: $W : \mathbb{R}^{n \times n}$
  begin
    $G_{-I} \leftarrow G \odot (\mathbf{1} - I)$ // set diagonal to 0
    $\mathbf{w} \leftarrow \bigoplus_{j=1}^{n} \sum_{i=1}^{n} G_{-I}[i, j]$ // sum over columns
    $W \leftarrow G_{-I} \odot \mathbf{w}^{-1}$ // scale columns to sum to 1
    return W
  end
end

Algorithm 1: Weighted Affinity Kernel

For each embedding, an estimate is calculated based on its neighbors in the graph. This can be expressed as matrix multiplication.

$$\hat{E} := \texttt{WAK}(G)E^{\top} \tag{4}$$

Hyperparameters :
$d_{max} : \mathbb{N}$ // maximum number of PCs
$k_{max} : \mathbb{N}$ // maximum number of neighbors
Input:
$X : \mathbb{R}^{m \times n}$ // input data
Output:
$d : \mathbb{N}$ // optimal number of PCs
$k : \mathbb{N}$ // optimal number of neighbors
begin
  $E \leftarrow \texttt{pca}(X)$ // principal components
  $L \leftarrow [:]$ // initialize loss
  for $d \in \{1 : d_{max}\}$ **do**
    $D \leftarrow \texttt{euclidean}(E[1 : d])$ // euclidean distance
    for $k \in \{1 : k_{max}\}$ **do**
      $G \leftarrow \texttt{knn}(D, k)$ // kNN adjacency matrix
      $W \leftarrow \texttt{WAK}(G)$ // weighted affinity kernel
      $\hat{X} \leftarrow WX^{\top}$ // denoised data
      $L[d, k] \leftarrow \texttt{MSE}(X\hat{X})$ // mean squared error
    end
  end
  $d, k \leftarrow \texttt{argmin}(L)$ // optimal d,k
  return d, k
end

Algorithm 2: Hyperparameter optimization with DEWAKSS.

1.4 Partitioned weighted affinity kernels

Though DEWAKSS uses a k-NN graph, any adjacency matrix will do. A clustering can be expressed as a graph where points within a cluster are completely connected and clusters are disconnected.

Let $K^{k \times n}$ be a matrix representing a clustering of n points into k clusters. Let each column be a 1-hot encoding of a cluster assignment for each point. We can obtain a partition matrix $P^{n \times n}$ by

$$P := \texttt{WAK}(K^{\top}K) \tag{5}$$

This method can be extended to soft cluster assignment, making it possible to learn K via SGD.

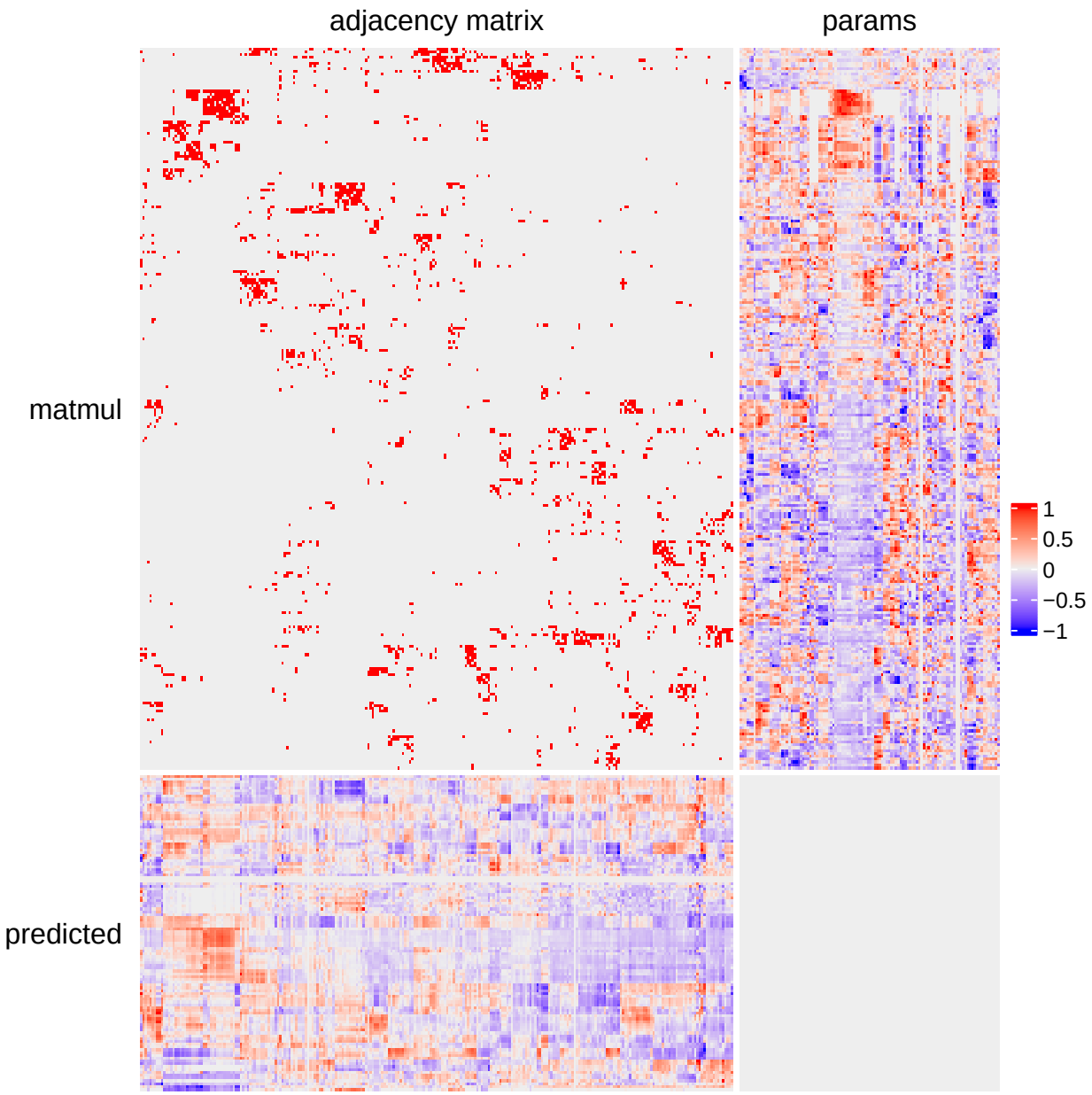

Figure 1: Illustration of diffusion with DEWAKSS. In this case the kernel is a 10-NN computed using the first 10 PCs of the data. Note that the denoised result is substantially smoother than the input data. While smoothing is often desirable for data imputation, for our purposes we would prefer to perserve variation.

Data `Partitioner(`$\mathcal{P}$`)`
- $\forall m : \mathbb{N}$ `// input parameter dimension`
- $\forall k : \mathbb{N}$ `// maximum number of clusters`
- **Input:**
- $\mathcal{P}$: `Arch(`*m,k*`)` `// partitioner architecture`
- **Output:**
- `Params(`$\mathcal{P}$`)` $\rightarrow$ `Model(`m, k`)`

end

Algorithm 3: Partitioner constructor

Function (`Partitioner(`$\mathcal{K}$`)`)(π, X)
- $\forall \mathcal{K}$: `Arch(`m, k`)` `// partitioner architecture`
- **Input:**
- π : `Params(`$\mathcal{K}$`)` `// partitioner parameters`
- $X : \mathbb{R}^m$ `// input data`
- **Output:**
- $K : \mathbb{R}^k$ `// cluster logits`
- **begin**
  - $K_0 \leftarrow \mathcal{K}(\pi)(X)$ `// cluster logit matrix`
  - $K \leftarrow$ `softmax(`K_0`)` `//`
  - **return** K
- **end**

end

Algorithm 4: Partitioner application

Function (`train(`$\mathcal{P}$`)`)(π, X)
- $\forall \mathcal{K}$: `Arch(`m, k`)` `// partitioner architecture`
- $\forall \mathcal{P}$: `Partitioner(`$\mathcal{K}$`)` `// partitioner`
- **Hyperparameters :**
- $t : \mathbb{N}$ `// number of epochs`
- $b : \mathbb{N}$ `// batch size`
- $\eta : \mathbb{R}$ `// learning rate`
- **Input:**
- π : `Params(`$\mathcal{K}$`)` `// partitioner parameters`
- $X : \mathbb{R}^m$ `// input data`
- **Output:**
- π : `Params(`$\mathcal{K}$`)`
- **begin**
  - **for** $\_ \in \{1 : t\}$ **do**
    - $X \leftarrow$ `sample(`$X[:,], b$`)` `// randomly sample data`
    - **Function** `loss(`κ`)`
      - **Input:** κ : `Params(`$\mathcal{K}$`)` `// parameters to update`
      - **Output:** $l : \mathbb{R}$ `// mean squared error`
      - **begin**
        - $K \leftarrow \mathcal{K}(\kappa, X)$ `//`
        - $P \leftarrow$ `WAK(`$K^\top K$`)` `//`
        - $\hat{X} \leftarrow PX^\top$ `//`
        - $l \leftarrow$ `MSE(`$\hat{X}, X$`)` `//`
        - **return** l
      - **end**
    - **end**
    - $\pi \leftarrow \pi - \eta \nabla$`loss(`$\pi$`)` `// gradient descent`
  - **end**
  - **return** π
- **end**

end

Algorithm 5: Partitioner training

We can now train a classifier with unlabeled data, no prior distribution, and no predefined number of clusters in $\mathcal{O}(n^2)$ time! The only hyperparameters are the maximum number of clusters, the neural net architecture, and the training hyperparameters. Because $PE^\top$ is $\mathcal{J}$-invariant, this classifier will converge on a solution less than the maximum k. We will refer to a model of this sort as a `Partitioner` to emphasize that while it returns logits corresponding to classifications, there are no labels on the training data.

Intuitions from transformers may be helpful in visualizing why this works. Informally, P can be equated to position-independent attention with data points as tokens and the batch size as the context window. Attentive readers may make a connection between masking the diagonal and BERT.

1.5 Diffusion in embedding space

Optimizing K using diffusion assumes

2 Methods

2.1 DeePWAK

Our basic unit performing clustering is the DeePWAK head. It is a deep learning architecture consisting of an encoder, classifier, and decoder subnetworks. We refer to the classifier subnetwork as a "partitioner" to emphasize that the data are unlabeled. There is no accuracy measure separate from the decoder loss. The partitioner simply tries to find the best P for minimizing loss of the decoded output. It can be considered a form of deep embedding clustering[7]. However there are several key differences from the canonical DEC implementation. Because we use a loss function based on `noise2self`, we don't need to carefully choose a kernel

The DeePWAK algorithm is inspired by dot product attention, but has a few key differences. Because it is position-independent, Q and K are simply transposes of each other. Instead of a shared MLP layer, each head has a separate encoder and decoder. This makes it easier to reason about components in isolation.

Data `DeePWAK(`$\mathcal{E}, \mathcal{K}, \mathcal{D}$`)`
- $\forall m : \mathbb{N}$ `// input parameter dimension`
- $\forall d : \mathbb{N}$ `// embedding dimension`
- $\forall k : \mathbb{N}$ `// maximum number of clusters`
- **Input:**
- $\mathcal{E}$: `Arch(`*m,d*`)` `// encoder architecture`
- $\mathcal{K}$: `Arch(`*m,k*`)` `// partitioner architecture`
- $\mathcal{D}$: `Arch(`*d,m*`)` `// decoder architecture`
- **Output:**
- `Params(`$\mathcal{E}$`)` $\rightarrow$ `Params(`$\mathcal{K}$`)` $\rightarrow$ `Params(`$\mathcal{D}$`)` $\rightarrow$ `Model(`m, m`)`

end

Algorithm 6: DeePWAK constructor

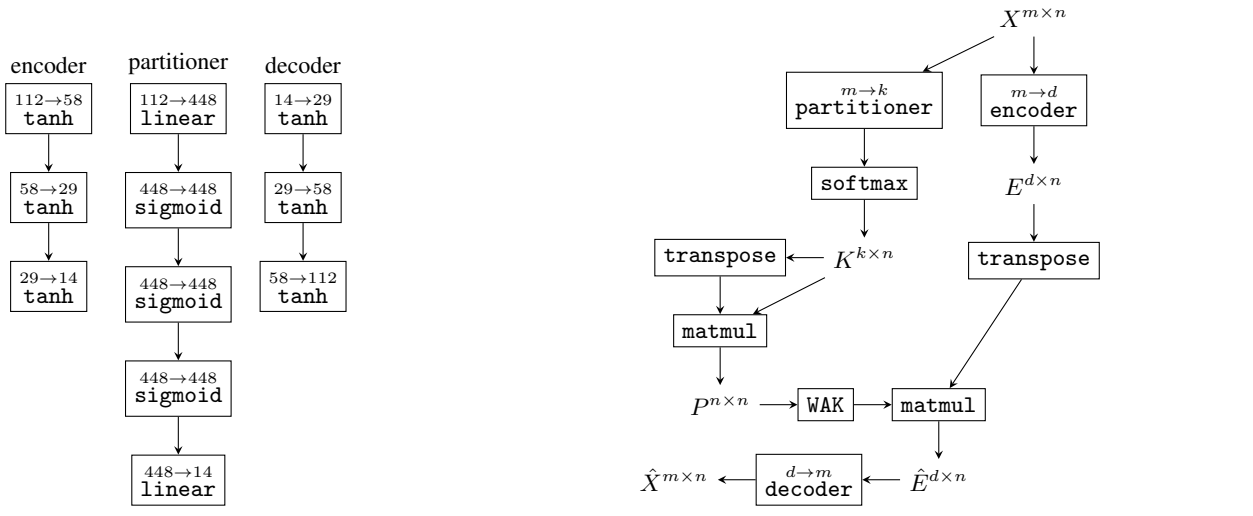

Figure 2: (a) Example architectures of encoder, partitioner, and decoder submodels. These are the architectures used in Section 3, but the algorithm generalizes to any $f, g, h : \mathbb{R}^* \to \mathbb{R}^*$ with appropriate input and output dimensions. In this case $m = 114$, $d = 14$,and $k = 14$. (b) Architecture of one DeePWAK head.

Function $(\texttt{DeePWAK}(\mathcal{E}, \mathcal{K}, \mathcal{D}))(\theta, \pi, \phi, X)$
- $\forall n : \mathbb{N}$ // batch size
- **Input:**
- $\theta : \texttt{Params}(\mathcal{E})$ // encoder
- $\pi : \texttt{Params}(\mathcal{K})$ // partitioner
- $\phi : \texttt{Params}(\mathcal{D})$ // decoder
- $X : \mathbb{R}^{m\times n}$ // input data
- **Output:**
- $\hat{X} : \mathbb{R}^{m\times n}$ // denoised data
- **begin**
  - $E \leftarrow \bigoplus_{i=1}^{n} \mathcal{E}(\theta)(X[:, i])$ // embedding matrix
  - $f \leftarrow \texttt{Partitioner}(\mathcal{K})$ // partitioner function
  - $K \leftarrow \bigoplus_{i=1}^{n} f(\pi, X[:, i])$ // cluster logit matrix
  - $P \leftarrow \texttt{WAK}(K^\top K)$ // partition matrix
  - $\hat{E} \leftarrow (PE^\top)^\top$ // denoised embedding
  - $\hat{X} \leftarrow \bigoplus_{i=1}^{n} \mathcal{D}(\phi)(\hat{E}[:, i])$ // denoised data
  - **return** $\hat{X}$
- **end**

end

Algorithm 7: DeePWAK application

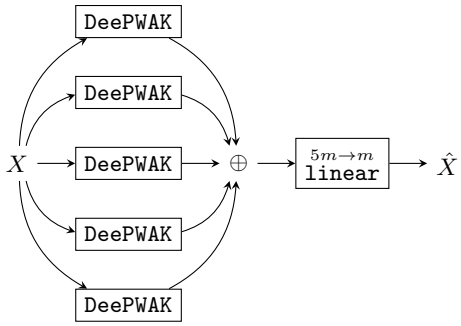

Figure 3: Architecture of one DeePWAK block with $h = 5$.

2.2 Ensemble DeePWAK

Data `DeePWAKBlock`$(\mathcal{W}, h)$
- $\forall \mathcal{E}$: `Arch`(m, d) // encoder template
- $\forall \mathcal{K}$: `Arch`(m, k) // partitioner template
- $\forall \mathcal{D}$: `Arch`(d, m) // decoder template
- **Input:**
- $\mathcal{W}$: `DeePWAK`$(\mathcal{E}, \mathcal{K}, \mathcal{D})$ // DeePWAK architecture
- h : $\mathbb{N}$ // number of heads
- **Output:**
- `List`$($`Params`$(\mathcal{E}), h) \to$ `List`$($`Params`$(\mathcal{K}), h) \to$ `List`$($`Params`$(\mathcal{D}), h) \to$ `Params`$($`Linear`$(hm, m)) \to$ `Model`(m, m)

end

Algorithm 8: DeePWAKBlock constructor

Function $($`DeePWAKBlock`$(\mathcal{W}, h))(\Theta, \Pi, \Phi, \psi, X)$
- $\forall \mathcal{W}$: `DeePWAK`$(\mathcal{E}, \mathcal{K}, \mathcal{D})$ // DeePWAK template
- **Input:**
- Θ : `List(Params(`$\mathcal{E}$`),`h`)` // encoders
- Π : `List(Params(`$\mathcal{K}$`),`h`)` // partitioners
- Φ : `List(Params(`$\mathcal{D}$`),`h`)` // decoders
- ψ : `Params(Linear(`hm, m`))` // pooling layer params
- X : $\mathbb{R}^{m \times n}$ // input data
- **Output:**
- $\hat{X}$: $\mathbb{R}^{m \times n}$ // denoised data
- **begin**
  - $\mathcal{X} \leftarrow \bigoplus_{\theta,\pi,\phi}^{\Theta,\Pi,\Phi} \mathcal{W}(\theta, \pi, \phi, X)$ // apply DeePWAK in parallel
  - $f \leftarrow$ `Linear`$(hm, m)(\psi)$ // pooling layer
  - $\hat{X} \leftarrow f(\mathcal{X})$ // consensus denoised output
  - **return** $\hat{X}$
- **end**

end

Algorithm 9: DeePWAKBlock application

After training the `DeePWAKBlock`, we can attempt to learn pooling models that compute consensus clusters and embeddings.

2.3 Example Data

We tested DeePWAK on preprocessed data from 3D microscopy. This data set was obtained from 1853 *Ciona robusta* embryos.

PCA-based methods perform poorly on this data set.

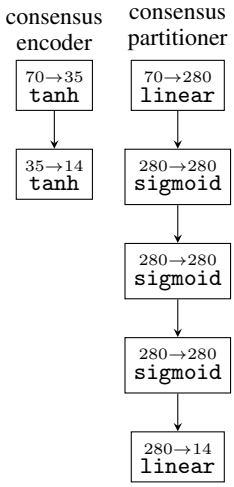

Figure 4: Example architectures of pooled encoder and pooled clusterer.

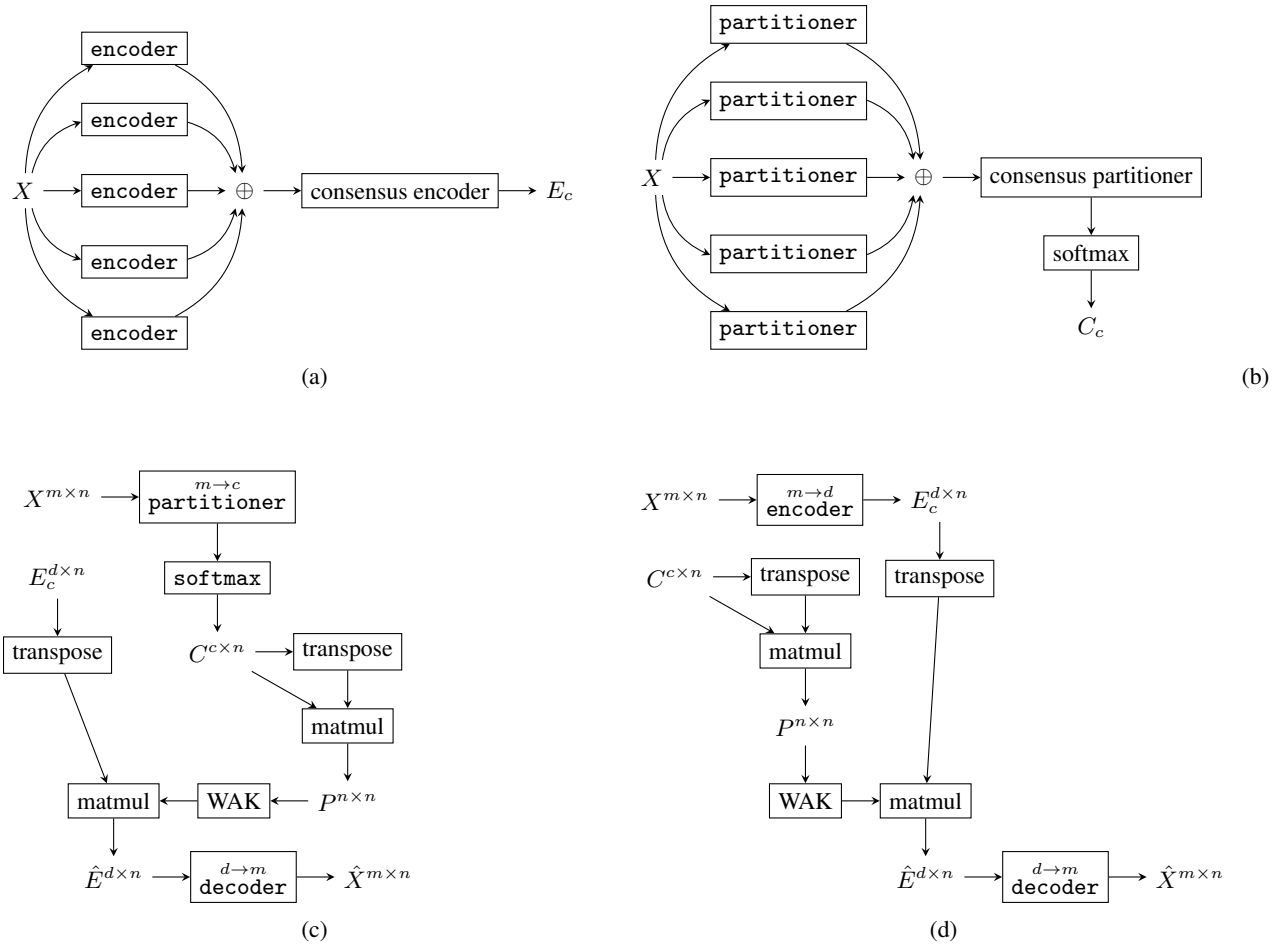

Figure 5: (a,b) Calculation of consensus E and C, respectively. (c,d) Drop-in of consensus E and C.

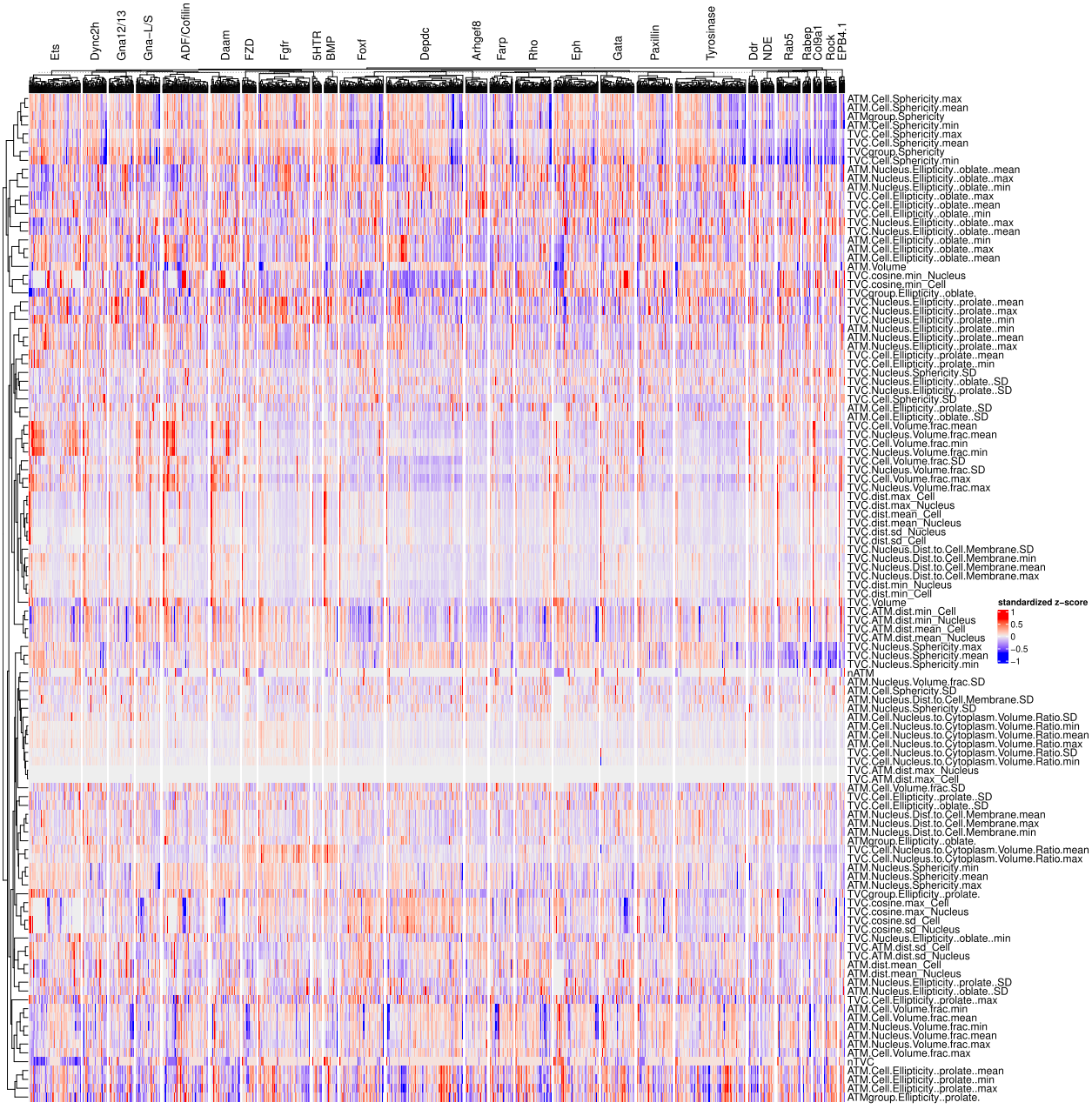

Figure 6: Microscopy data after preprocessing.

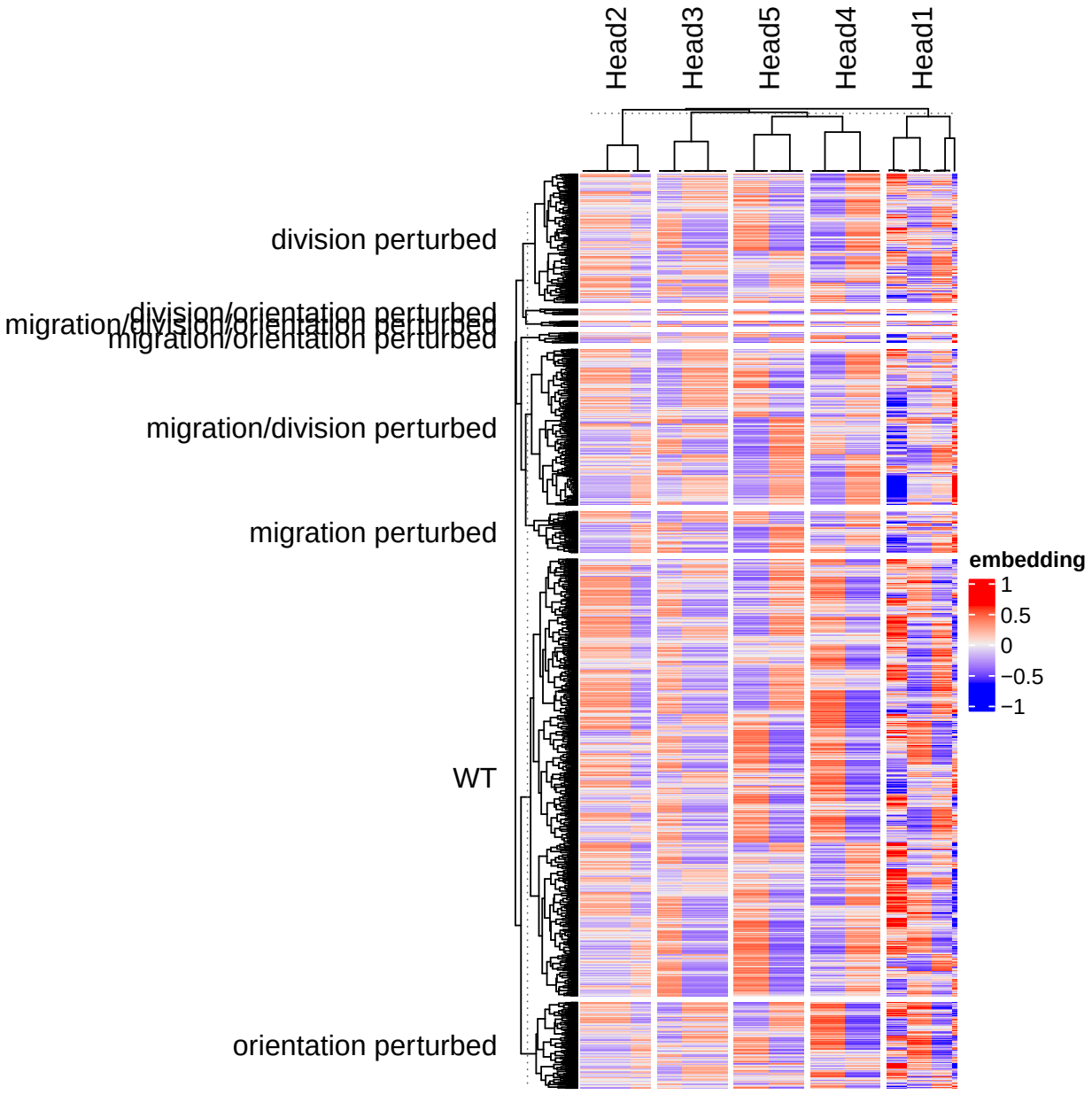

Figure 7: DeePWAK learns sparse embedding values.

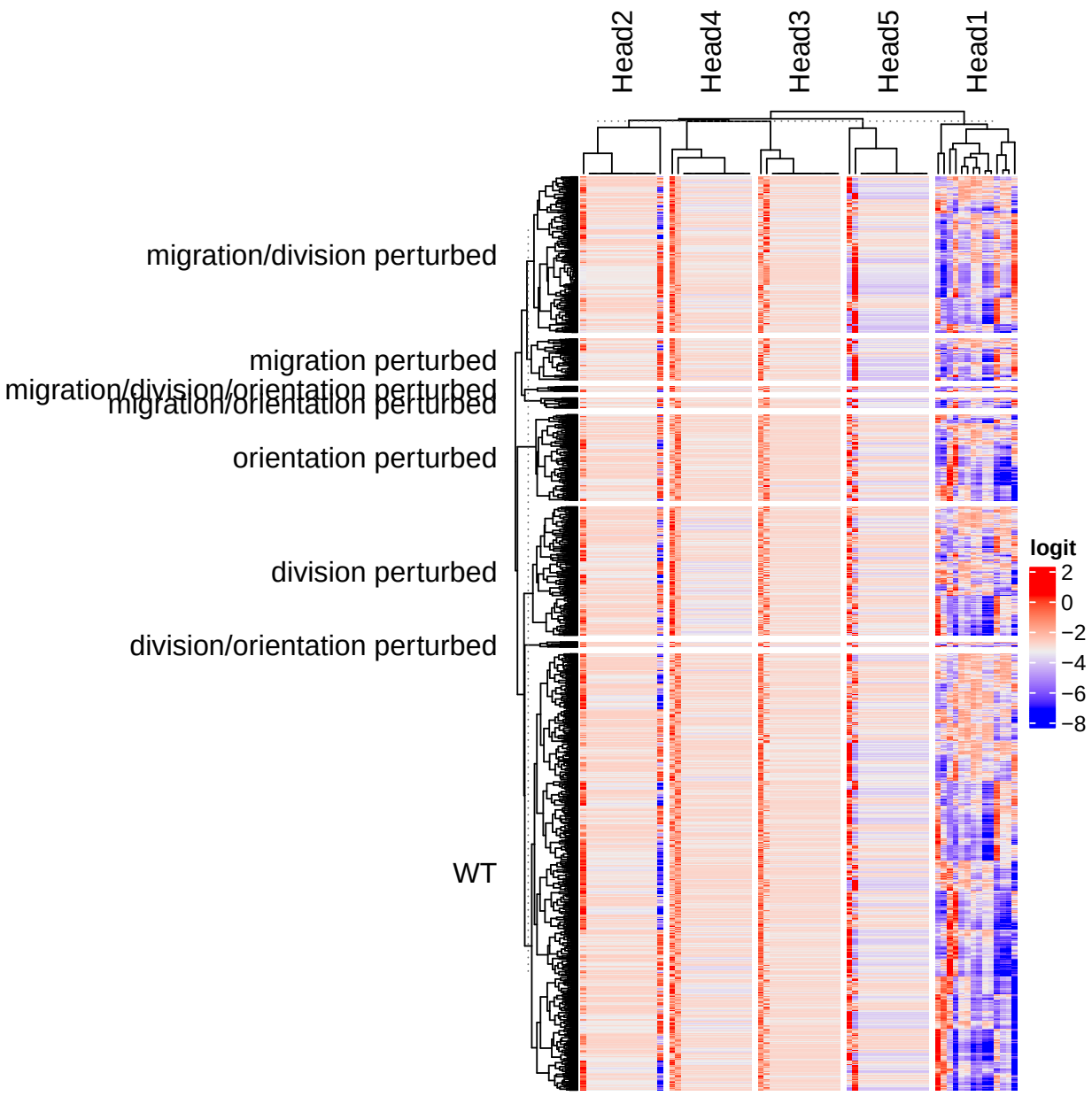

Figure 8: For most heads, two clusters dominate.

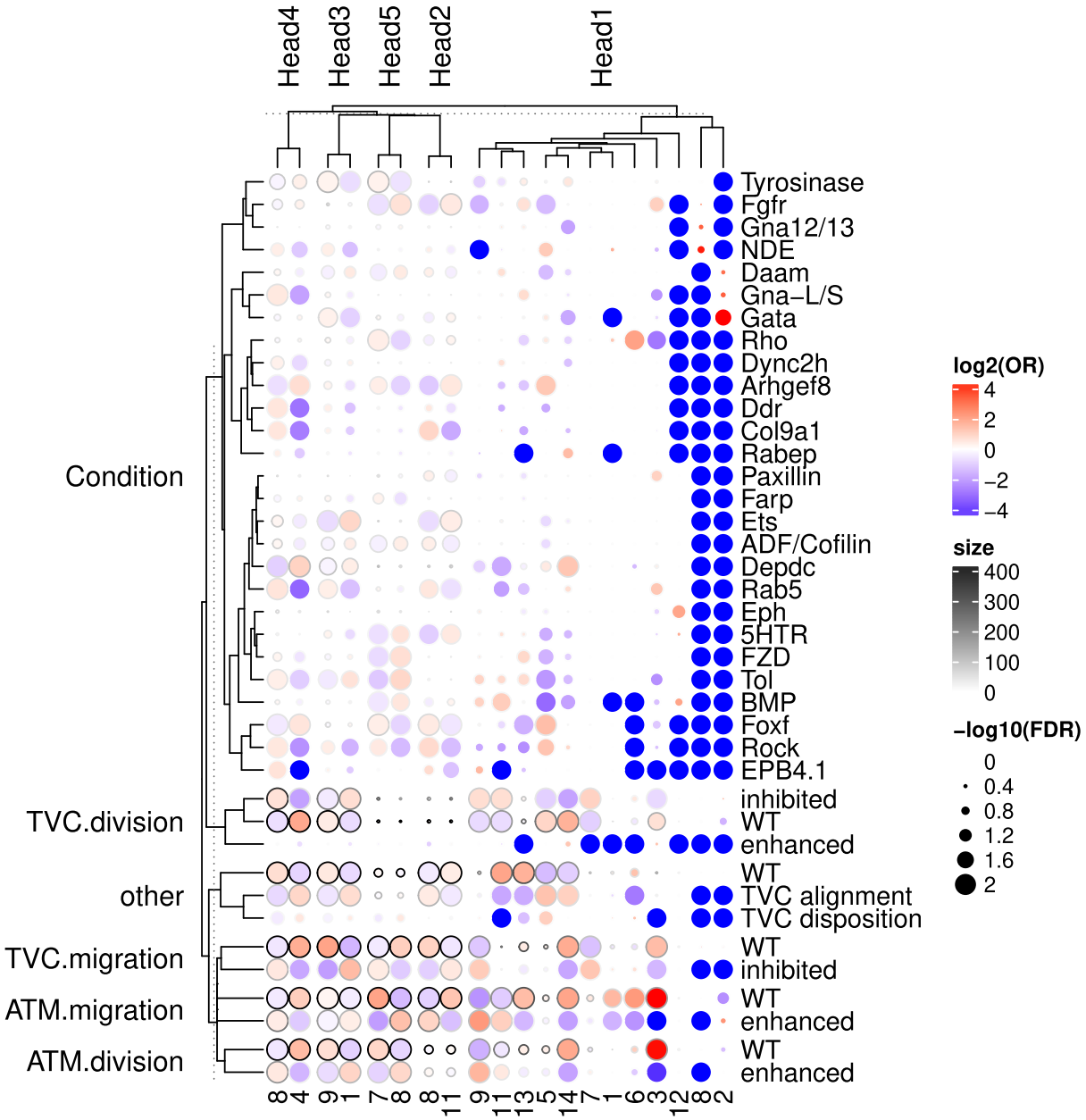

Figure 9: Hypergeometric test for enrichment of phenotype and treatment condition for each cluster in each head. Dot color indicates overrepresentation of each category in a cluster compared to a uniform prior. Dot size indicates statistical significance of the difference. Dot outline shade indicates number of embryos represented by a dot. Head 1 appears "polysemantic". The others appear to distinguish single phenotypic categories.

3 Results

3.1 Multihead DeePWAK learns sparse representations

4 Discussion

4.1 Application to Sparse Dictionary Learning

4.2 Recurrent DeePWAK

5 Conclusion

References

[1] Joshua Batson and Loic Royer. *Noise2Self: Blind Denoising by Self-Supervision*. 2019. arXiv: 1901.11365 [cs.CV].

[2] Dong Huang et al. *DeepCluE: Enhanced Image Clustering via Multi-layer Ensembles in Deep Neural Networks*. 2023. arXiv: 2206.00359 [cs.CV].

[3] Liang Li et al. "Local Sample-Weighted Multiple Kernel Clustering With Consensus Discriminative Graph". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022), pp. 1–14. DOI: 10.1109/TNNLS.2022.3184970.

[4] Yazhou Ren et al. *Deep Clustering: A Comprehensive Survey*. 2022. arXiv: 2210.04142 [cs.LG].

[5] Bishwajit Saha et al. *End-to-end Differentiable Clustering with Associative Memories*. 2023. arXiv: 2306.03209 [cs.LG].

[6] Andreas Tjärnberg et al. "Optimal tuning of weighted kNN- and diffusion-based methods for denoising single cell genomics data". In: *PLOS Computational Biology* 17.1 (Jan. 2021), pp. 1–22. DOI: 10.1371/journal.pcbi.1008569. URL: https://doi.org/10.1371/journal.pcbi.1008569.

[7] Junyuan Xie, Ross Girshick, and Ali Farhadi. *Unsupervised Deep Embedding for Clustering Analysis*. 2016. arXiv: 1511.06335 [cs.LG].