

---

# DENOISING WITH A PARTITIONED AFFINITY KERNEL ENABLES CLUSTER-FEATURE DECOMPOSITION

---

A PREPRINT

Keira Wiechecki<sup>1</sup> and Jade Zaslavsky<sup>2</sup>

<sup>1</sup>Center for Genomics & Systems Biology, New York University, kaw504@nyu.edu

<sup>2</sup>Transcribbit

January 30, 2024

## ABSTRACT

We introduce a new method of deep clustering using denoising. A denoising prior allows for simultaneously learning a partition and an embedding of a data set. We find that without imposing any sparsity constraints, this method learns a sparse decomposition of the latent feature space and the latent cluster space.

## 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[5].

Compression is a generalization of supervised learning. We propose that

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 naively 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 essentially the goal of sparse dictionary learning, where  $C$  is the dictionary and  $K$  is the key.

**Clusters as attractor states** One of the most beautiful results from information theory is that *prediction is thermodynamic work*. The entropy of any dataset can be expressed in terms of transition probabilities. The entropy of a partition can be expressed as a mask applied to this transition matrix. An optimal clustering has maximal entropy[1]. This is equivalent to saying that the optimal partition is the one that *deletes* the most information from the data when adopted as a prior.

**Clustering as the optimal way of labeling a data set** Though clustering is normally phrased as an unsupervised learning problem, it may be informative to frame it as *the dual of supervised learning*. In other words, instead of finding the features that contain the most information given a labeling,

Code is available at <https://github.com/kewiechecki/DeePWAK>.

## 2 Background

### 2.1 The unreasonable effectiveness of transformers

### 2.2 The unreasonable effectiveness of MoE

### 2.3 Notation

We use lowercase Latin characters to denote scalars, boldface lowercase characters to denote vectors, and capital Latin characters to denote matrices. Subscripts indicate indices. Because we will mostly be working with matrices in  $\mathbb{R}$ , we abbreviate  $X : \mathbb{R}^{m \times n}$  as  $X^{m \times n}$ . We use a circumflex to indicate a reconstruction of an input by a predictor. Lowercase Greek characters indicate the parameters of a model.

Function names are in monospace.

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

Additional details are in Appendix A.

## 3 Related Work

### 3.1 Sparse autoencoders

Sparse dictionary learning has become a powerful tool for interpretation of neural networks. The activations of a neural network can be decomposed into a linear combination of dictionary features[4]. There are several limitations to this technique as currently applied. The number of features must be preselected. Interpretation of features still relies on manual inspection, but there is no structure or organization imposed on the dictionary. This results in a tradeoff between feature interpretability and feature search space. It would be desirable to be able to selectively decompose higher level features into lower level features. One of the most interesting properties of SAEs is that features seem to be organized into clusters[3]. This suggests the possibility of merging low level features or splitting high level features.

### 3.2 Denoising the data explains the data

For a large class of denoising functions, it is possible to find optimal parameters using only unlabeled noisy data[2].

See Appendix B.1.

### 3.3 Diffusion with weighted affinity kernels

`noise2self` is particularly useful for finding optimal parameters for generating a graph[6]. (see Appendix B.2) The adjacency matrix  $G$  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 (Algorithm 1).

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

$$\hat{X} := \text{WAK}(G)X^{\top} \quad (2)$$

### 3.4 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  $P \in \mathbb{R}^{n \times n}$  by what we'll call the *partitioned weighted affinity kernel* (PWAK) function.

$$\text{PWAK}(K) := \text{WAK}(K^\top K) \quad (3)$$

This lets us define a loss function

$$\mathcal{L}_{\text{PWAK}}(K, X) := \mathbb{E}[\text{PWAK}(K)X^\top - X]^2 \quad (4)$$

PWAK can be extended to soft cluster assignment, making it possible to learn  $K$  via SGD. 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. See Appendix C.1 for details.

We can now train a model to classify unlabeled data into an undefined number of clusters with no prior distribution in  $\mathcal{O}(n^2)$  time!

### 3.5 Diffusion in embedding space

A common problem in kernel diffusion is nonlinearity. We have no reason to expect features of the data are linearly separable. We can however use an autoencoder to find a linearly separable embedding [8]. Since we have no reason to expect a priori that a given embedding will be linearly separable, this generally involves applying a linear transformation to the embeddings and letting SGD figure it out.

We refer to this combination of an autoencoder with a `Partitioner` trained on a PWAK loss function as *denoising by deep learning of a partitioned weighted affinity kernel* (DeePWAK).

For an encoder  $\theta$  and a decoder  $\phi$ , our loss function is now

$$\mathcal{L}_{\text{DeePWAK}}(\theta, \phi, K, X) := \mathbb{E}[\phi(\text{PWAK}(K)\theta(X)^\top) - X]^2 \quad (5)$$

This is very similar to the SAE loss function, as both aim to impose linear separability in an embedding space. There is one subtle distinction that has wide-reaching ramifications.  $\mathcal{L}_{\text{DeePWAK}}$  is calculated for an entire minibatch rather than for each input value. Rather than directly learning features, DeePWAK attempts to recognize values that can be used to impute each other. This results in not only sparse features, but sparse clusters.

## 4 Methods

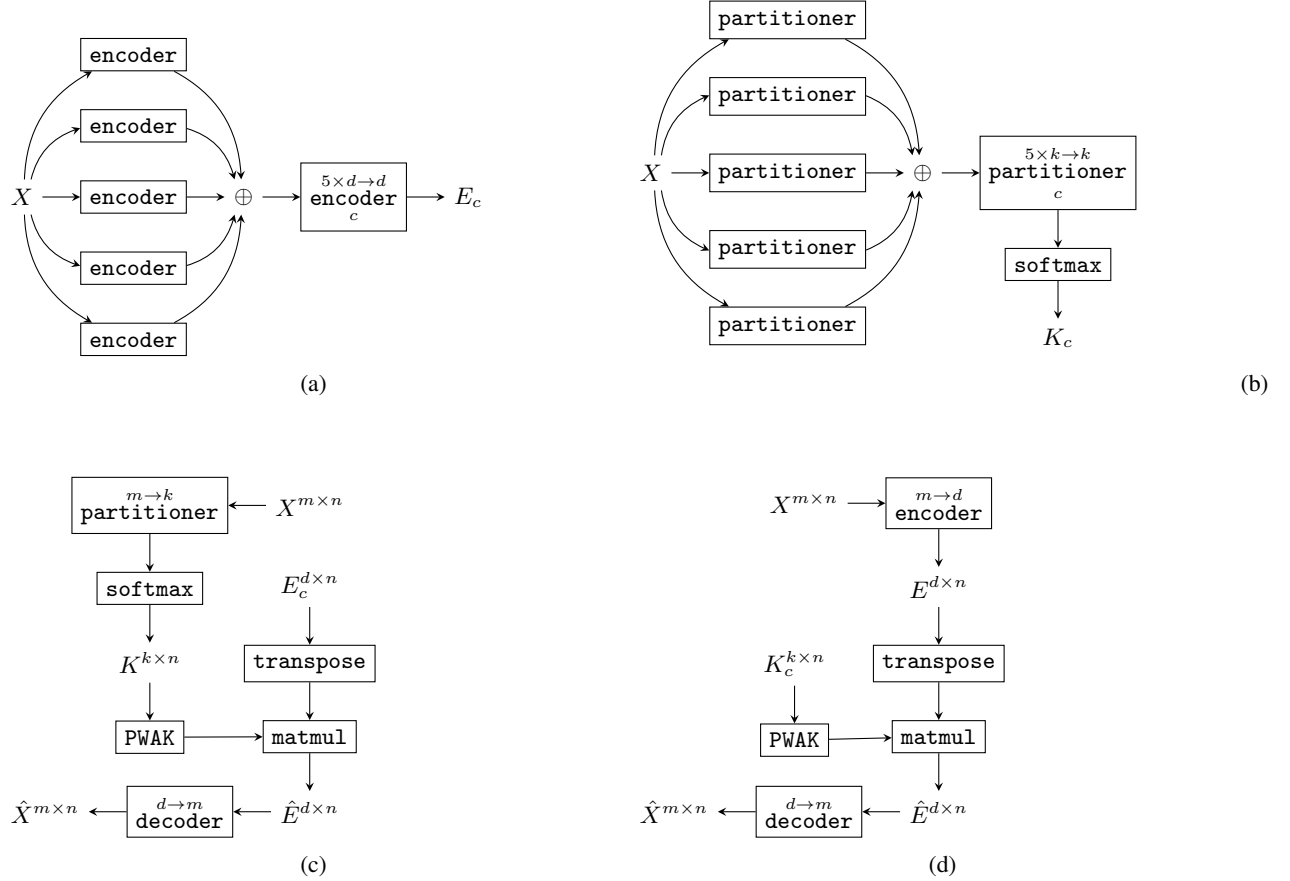
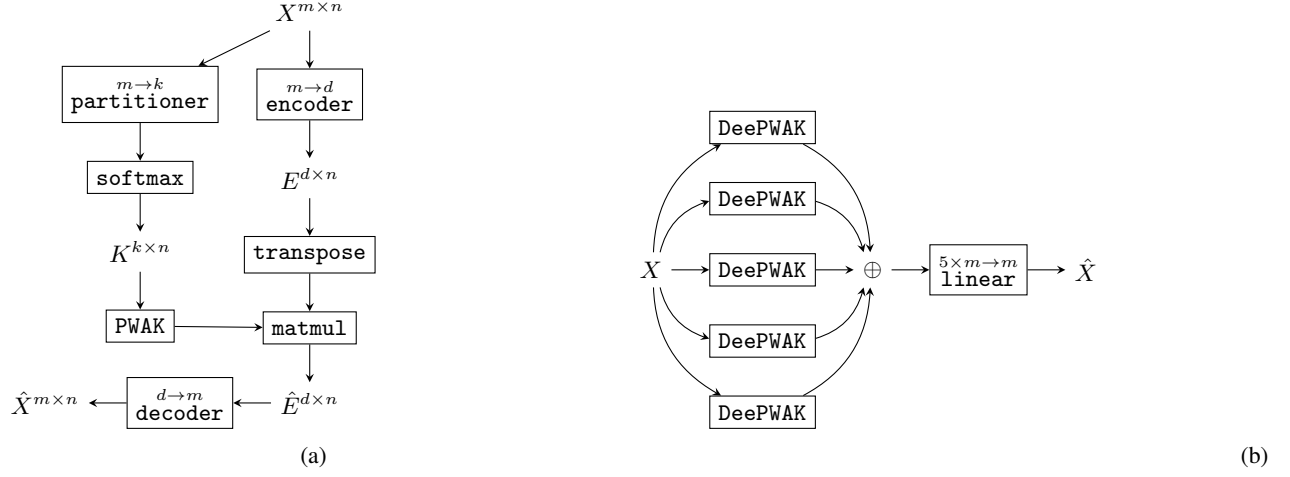
### 4.1 DeePWAK

Our basic unit performing clustering is the DeePWAK head. It is a deep learning architecture consisting of an encoder, partitioner, and decoder subnetworks.

### 4.2 Ensemble DeePWAK

DeePWAK can be used for ensemble clustering by training multiple heads in parallel and linearly combining the results. We refer to this as a `DeePWAKBlock`. An intuitive interpretation is that each head learns to cluster a different “slice” of the data.

After training the `DeePWAKBlock`, we can attempt to learn pooling models that compute consensus clusters and embeddings (Fig. ?? Appendix ??). :


 Figure 2: (a,b) Calculation of consensus  $E$  and  $C$ , respectively. (c,d) Drop-in of consensus  $E$  and  $C$ .

### 4.3 Recurrent DeePWAK

## 5 Results

### 5.1 Multihead DeePWAK learns sparse representations

Multihead DeePWAK learns remarkably sparse features (Fig. 3) and clusters (Fig. 4). Even more remarkably, all but one head appear bisemantic. This is suggestive that the model may in fact be learning a tree structure to split the data along major features. These major features effectively split experimenter-labeled phenotypes (Fig. 5).

All of the more nuanced discrimination happens in **Head1**. It splits the data into almost but not quite the maximum number of clusters, indicating our choice of  $k$  is close to the latent concept space for (this projection of) the data. Fascinatingly, the embeddings are even sparser.

It should be emphasized how surprising it is to see this much structure emerge from this data set. Contrast Fig. 7. 1853 embryo halves is nothing by machine learning standards. Though the data represent rich 3D microscopy images, these images were very crudely summarized by extracting 114 ad hoc statistics with no consideration for redundancy or significance.

## 6 Discussion

### 6.1 Data are singular

This striking result seems likely to be due to learning an embedding and partition of the data simultaneously.

**MCMC interpretation** For each minibatch forward pass, the partitioner gives a prior on each sample being in the same cluster. During training it updates based on how well its classification does at predicting the data.

**GCN interpretation** A key insight of `noise2self` is the unique power of generative processes. Unlike most deep clustering methods, DeePWAK does not perform clustering on the embedding space.

**Can GANs be generalized as learning  $KE$  decomposition?**

### 6.2 Application to sparse dictionary learning

The embeddings appear to be representing the data as a sparse dictionary of features. These features appear to match observable phenotypes. Clusters are also sparse. However, *there are more clusters than features*. This is suggestive that data may be represented as *combinatoric* mixtures of features.

### 6.3 The future of SAEs

We suspect combinatoric encoding of this sort is possible because DeePWAK performs computation on an entire minibatch.

### 6.4 Insight into the unreasonable effectiveness of transformers

**$QK, OV$  decomposition**

**Context matters even when there's no positional information**

**Attention is graph diffusion**

**Attention is `noise2self`**

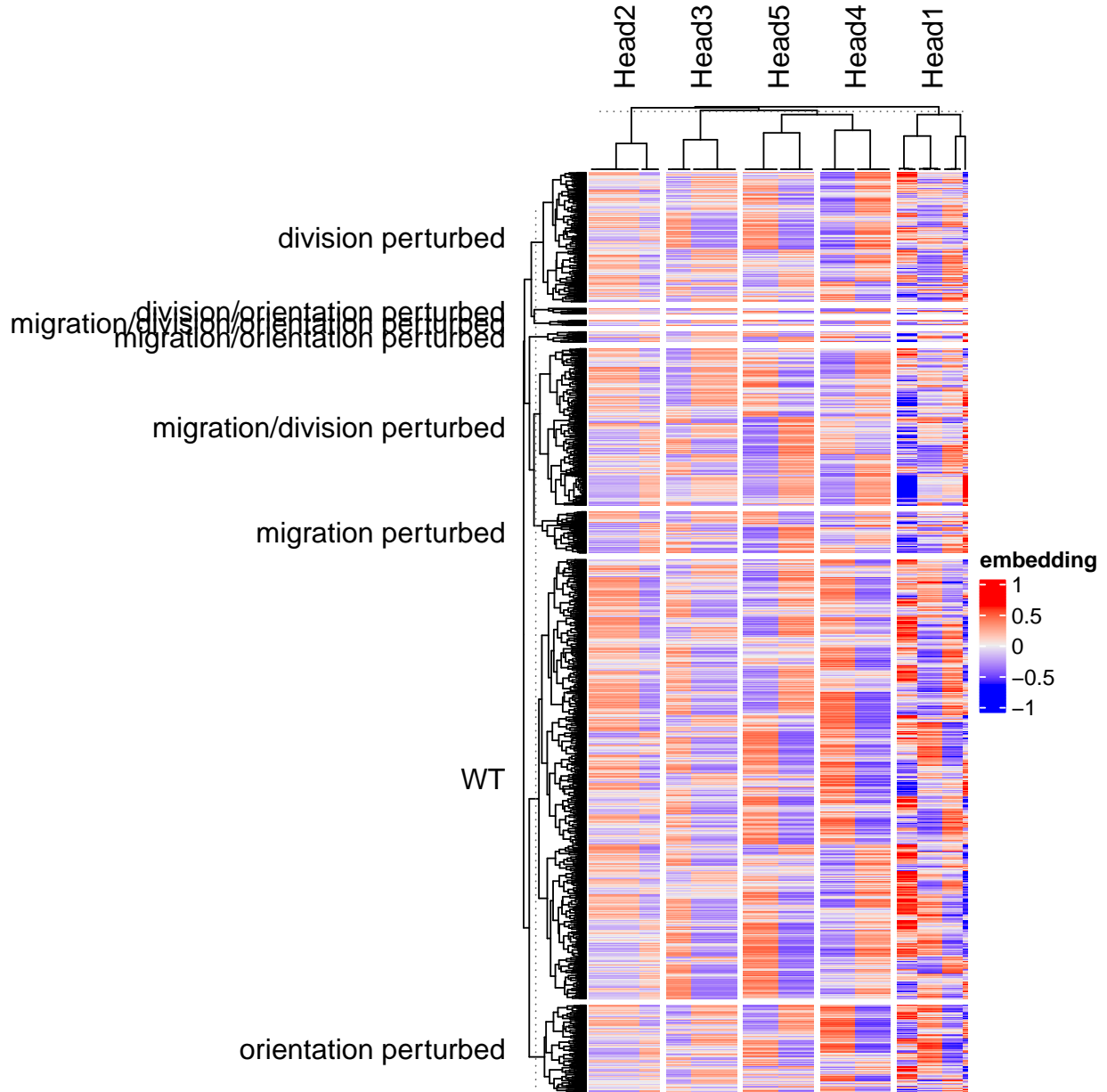


Figure 3: DeePWAK learns sparse embedding values.

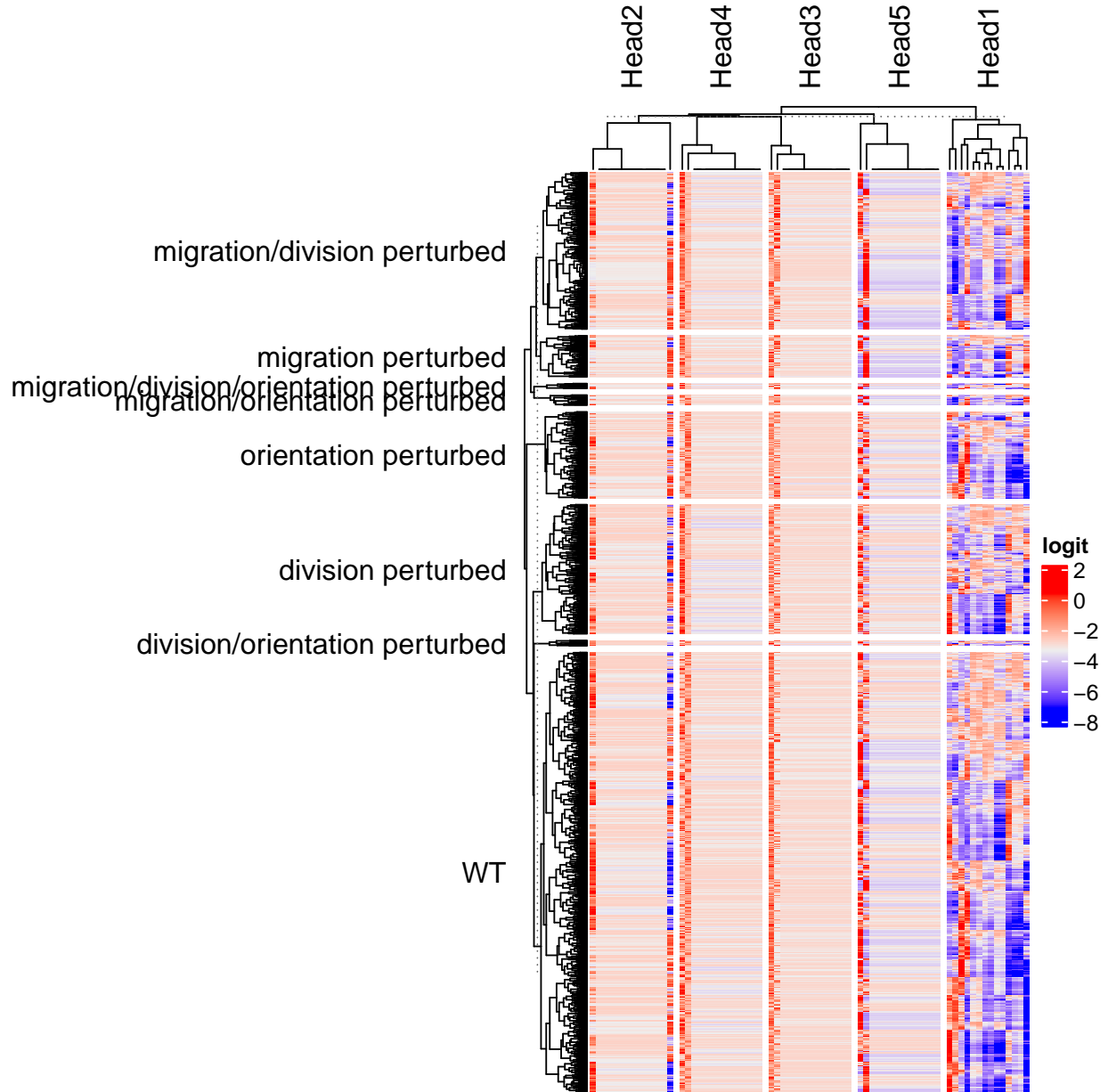


Figure 4: For most heads, two clusters dominate.

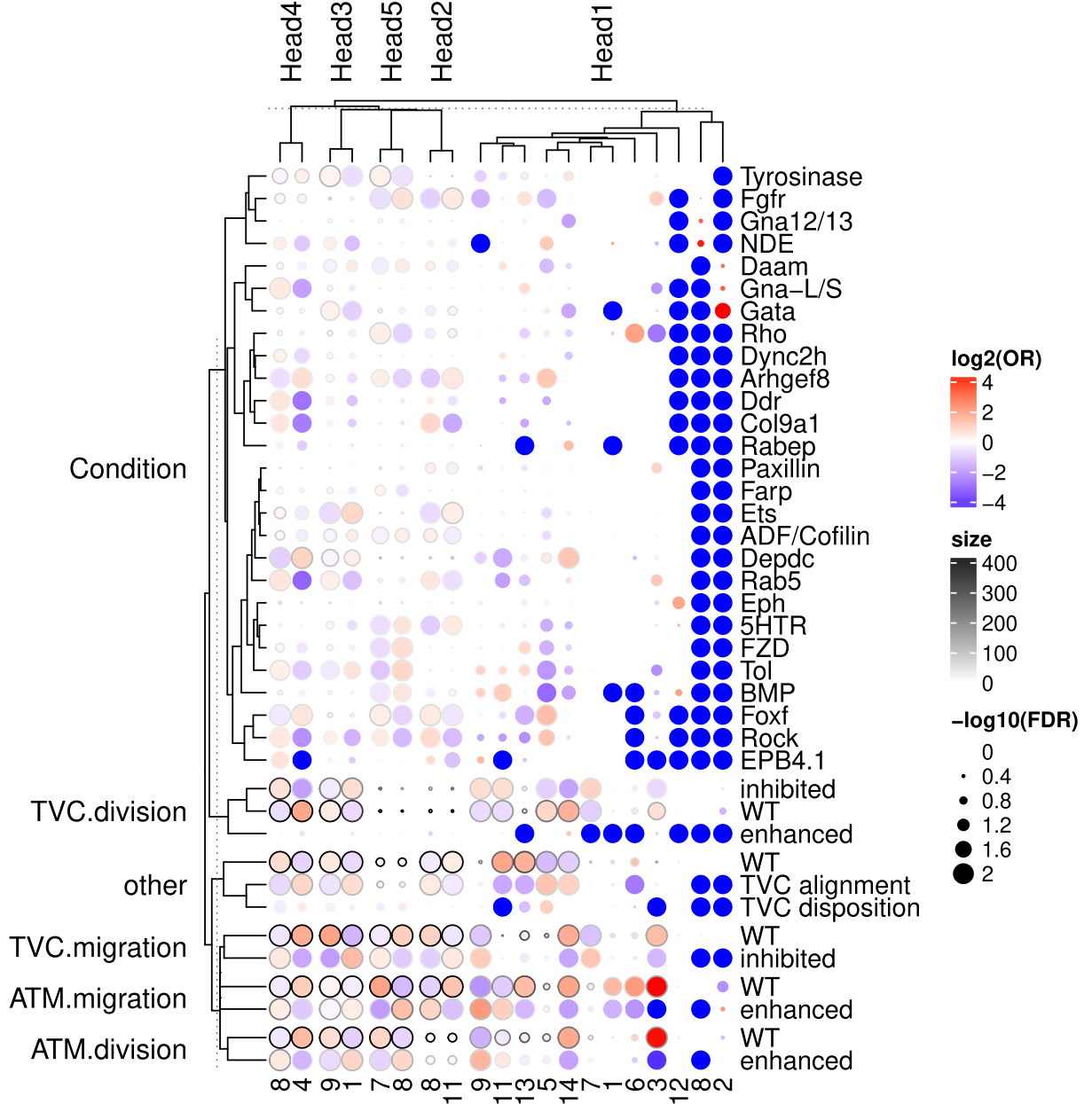


Figure 5: Hypergeometric test for enrichment of phenotype and treatment condition for each cluster in each head. Dot color indicates overrepresentation of each category in a cluster compared to a uniform prior. Dot size indicates statistical significance of the difference. Dot outline shade indicates number of embryos represented by a dot. Head 1 appears “polysemantic”. The others appear to distinguish single phenotypic categories.



## 7 Conclusion

## References

- [1] Edwin Aldana-Bobadilla and Angel Kuri-Morales. “A Clustering Method Based on the Maximum Entropy Principle”. In: *Entropy* 17.1 (2015), pp. 151–180. ISSN: 1099-4300. DOI: 10.3390/e17010151. URL: <https://www.mdpi.com/1099-4300/17/1/151>.
- [2] Joshua Batson and Loic Royer. *Noise2Self: Blind Denoising by Self-Supervision*. 2019. arXiv: 1901.11365 [cs.CV].
- [3] Trenton Bricken et al. “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning”. In: *Transformer Circuits Thread* (2023). <https://transformer-circuits.pub/2023/monosemantic-features/index.html>.
- [4] Hoagy Cunningham et al. *Sparse Autoencoders Find Highly Interpretable Features in Language Models*. 2023. arXiv: 2309.08600 [cs.LG].
- [5] 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.
- [6] Andreas Tjörnberg 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] Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. “From Louvain to Leiden: guaranteeing well-connected communities”. In: *CoRR* abs/1810.08473 (2018). arXiv: 1810.08473. URL: <http://arxiv.org/abs/1810.08473>.
- [8] Junyuan Xie, Ross Girshick, and Ali Farhadi. *Unsupervised Deep Embedding for Clustering Analysis*. 2016. arXiv: 1511.06335 [cs.LG].

## A Notation

Capital Greek letters indicate lists of parameters for multiple models. Boldface Greek letters indicate parameter spaces.

$\odot$  is the Hadamard product.

We believe mapping an expression over an input vector and concatenating the result deserves its own operator. We denote this with  $\bigoplus$ .

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

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

### A.1 Dependent Types

Because we are working with models composed of submodels, we would like to be able to impose constraints on which sort of submodels can be composed into a model of a given type. To accomplish this it is necessary to define the inputs and outputs of constituent models with typechecking.

We use capitalized monospace to indicate types. Dependent types are given as  $\text{Type}(x)$ .

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

### A.2 Models, Parameters, and Architectures

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

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

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

We use the notation  $\mathcal{F}(\theta)$  to indicate a model architecture  $\mathcal{F}$  parameterized by  $\theta$ .  $\mathcal{F}(\theta)(X)$  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.

### A.3 Pseudocode

For each model type, we define a constructor, an application rule, and a training function. Our definitions draw heavily on proofs as programs [empty citation]. We adapt typed lambda calculus notation from LACI [empty citation].

## B Additional Background

### B.1 noise2self

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  $\theta \in \Theta$  that depends on its complement  $X_{J^c}$

$$\hat{X}_J = \mathcal{F}(\theta)(X_{J^c}) \quad (6)$$

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

The optimal  $\theta$  is given by

$$\text{noise2self}_{\theta}[\mathcal{F}(\theta), X] := \underset{\theta}{\operatorname{argmin}}_{\Theta} \left[ \sum_J \mathbb{E}[X_J - \mathcal{F}(\theta)(X_{J^c})]^2 \right] \quad (7)$$

### B.2 DEWAKSS

DEWAKSS [6] is a method of data imputation to address sparsity in single cell RNA-seq experiments. It generates a  $k$ -NN graph from the top  $d$  principal components. By using a graph partitioning algorithm such as `leiden`[7], it can easily be turned into a clustering algorithm. Because these are nondifferentiable function, it has to grid search over all possible values. This severely limits the number of tunable parameters we can add, motivating finding an end-to-end differentiable alternative.

**Function** `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
     $\mathbf{w} \leftarrow \text{einsum}_{ij \rightarrow j}(G_{-I})$  // sum over columns
     $W \leftarrow \text{einsum}_{ij, i \rightarrow j}(G_{-I}, \mathbf{w}^{-1})$  // scale columns to sum to 1
    return  $W$ 
end
end

```

**Algorithm 1:** Weighted Affinity Kernel

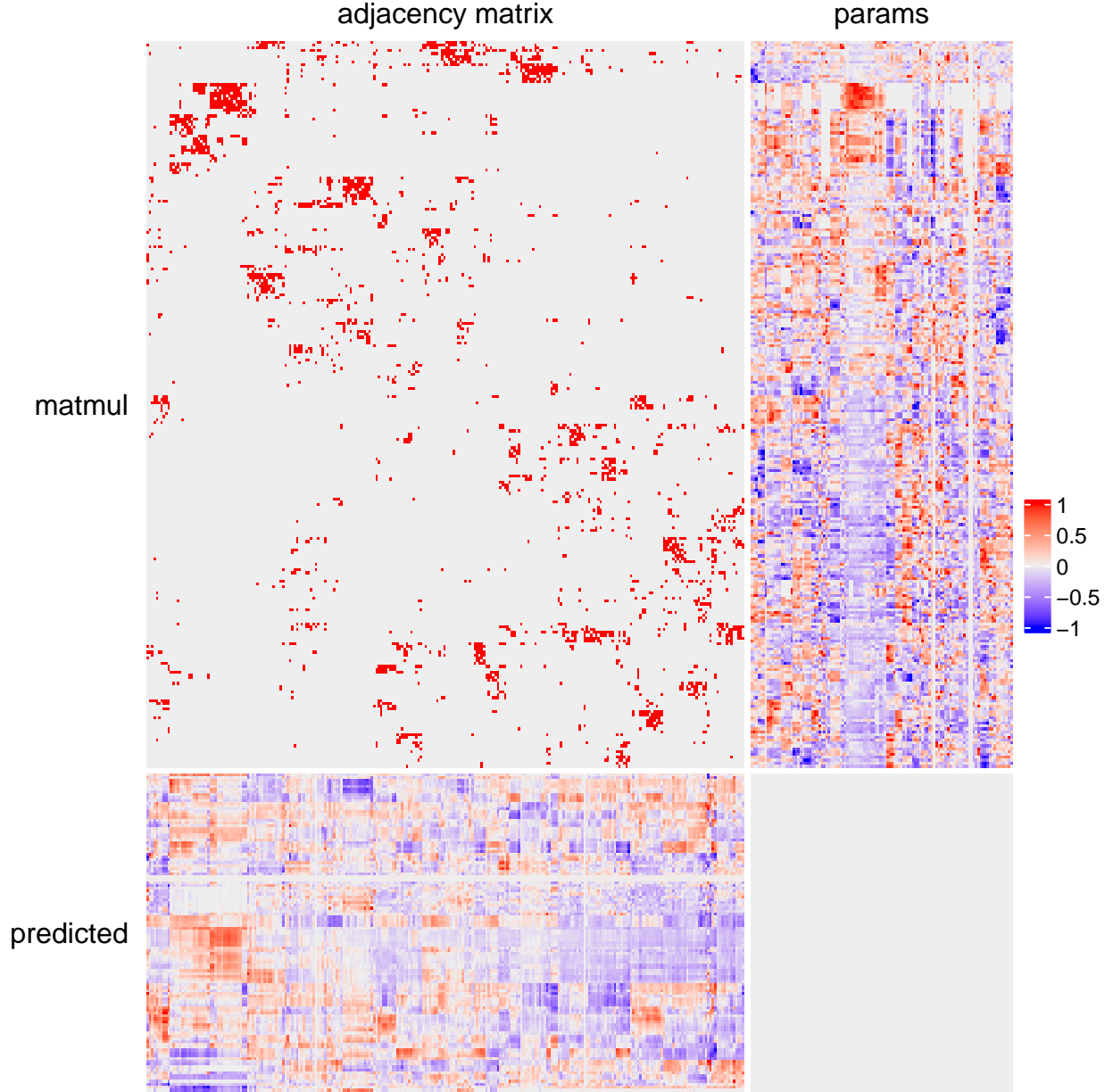


Figure 6: Illustration of diffusion with DEWAKSS. In this case the kernel is a 10-NN computed using the first 10 PCs of the data. Note that the denoised result is substantially smoother than the input data. While smoothing is often desirable for data imputation, for our purposes we would prefer to preserve variation.

```

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

**Algorithm 2:** Hyperparameter optimization with DEWAKSS.

## C Types

### C.1 Partitioner

The only hyperparameters are the maximum number of clusters, the neural net architecture, and the training hyperparameters. Because  $PX^\top$  is  $\mathcal{J}$ -invariant, this classifier will converge on a solution less than the maximum  $k$ . 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.

```

Data Partitioner( $\mathcal{K}$ )
     $\forall m, k : \mathbb{N}$  // input dimension, max clusters
    Input:  $\mathcal{K} : \text{Arch}(m, k)$  // partitioner architecture
    Output:  $\mathcal{P} : \text{Params}(\mathcal{K}) \rightarrow \text{Model}(m, k)$ 
end
    
```

**Algorithm 3:** Partitioner constructor

```

Function (Partitioner( $\mathcal{K}$ ))( $\pi, X$ )
     $\forall \mathcal{K} : \text{Arch}(m, k)$  // partitioner architecture
    Input:
         $\pi : \text{Params}(\mathcal{K})$  // 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
         $K \leftarrow \text{softmax}(K_0)$  //
    return  $K$ 
    end
end
    
```

**Algorithm 4:** Partitioner application

```

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

**Algorithm 5:** Partitioner training

## C.2 DeePWAK

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[8]. 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( $\mathcal{E}, \mathcal{K}, \mathcal{D}$ )
     $\forall m, d, k : \mathbb{N}$  // input dimension, embedding dimension, max clusters
    Input:
     $\mathcal{E} : \text{Arch}(m, d)$  // encoder architecture
     $\mathcal{K} : \text{Arch}(m, k)$  // partitioner architecture
     $\mathcal{D} : \text{Arch}(d, m)$  // decoder architecture
    Output:
     $\mathcal{W} : \text{Params}(\mathcal{E}) \rightarrow \text{Params}(\mathcal{K}) \rightarrow \text{Params}(\mathcal{D}) \rightarrow \text{Model}(m, m)$ 
    end
    
```

**Algorithm 6:** DeePWAK constructor

```

Function (DeePWAK( $\mathcal{E}, \mathcal{K}, \mathcal{D}$ ))( $\theta, \pi, \phi, X$ )
     $\forall n : \mathbb{N}$  // batch size
    Input:
     $\theta : \text{Params}(\mathcal{E})$  // encoder
     $\pi : \text{Params}(\mathcal{K})$  // partitioner
     $\phi : \text{Params}(\mathcal{D})$  // 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
         $f \leftarrow \text{Partitioner}(\mathcal{K})$  // partitioner function
         $K \leftarrow \bigoplus_{i=1}^n f(\pi, X[:, i])$  // cluster logit matrix
         $\hat{E} \leftarrow (\text{PWAK}(K)E^\top)^\top$  // 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

### C.3 DeePWAKBlock

```

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

**Algorithm 8:** DeePWAKBlock constructor

```

Function (DeePWAKBlock( $\mathcal{W}, h$ ))( $\Theta, \Pi, \Phi, \psi, X$ )
     $\forall \mathcal{W} : \text{DeePWAK}(\mathcal{E}, \mathcal{K}, \mathcal{D})$                                 // DeePWAK template
    Input:
     $\Theta : \text{List}(\text{Params}(\mathcal{E}), h)$                                 // encoders
     $\Pi : \text{List}(\text{Params}(\mathcal{K}), h)$                                 // partitioners
     $\Phi : \text{List}(\text{Params}(\mathcal{D}), h)$                                 // decoders
     $\psi : \text{Params}(\text{Linear}(hm, m))$                                 // 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
         $f \leftarrow \text{Linear}(hm, m)(\psi)$                             // pooling layer
         $\hat{X} \leftarrow f(\mathcal{X})$                                     // consensus denoised output
    return  $\hat{X}$ 
    end
end
    
```

**Algorithm 9:** DeePWAKBlock application

#### C.4 Consensus Clusters

### D Ensemble Clustering Example

#### D.1 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.

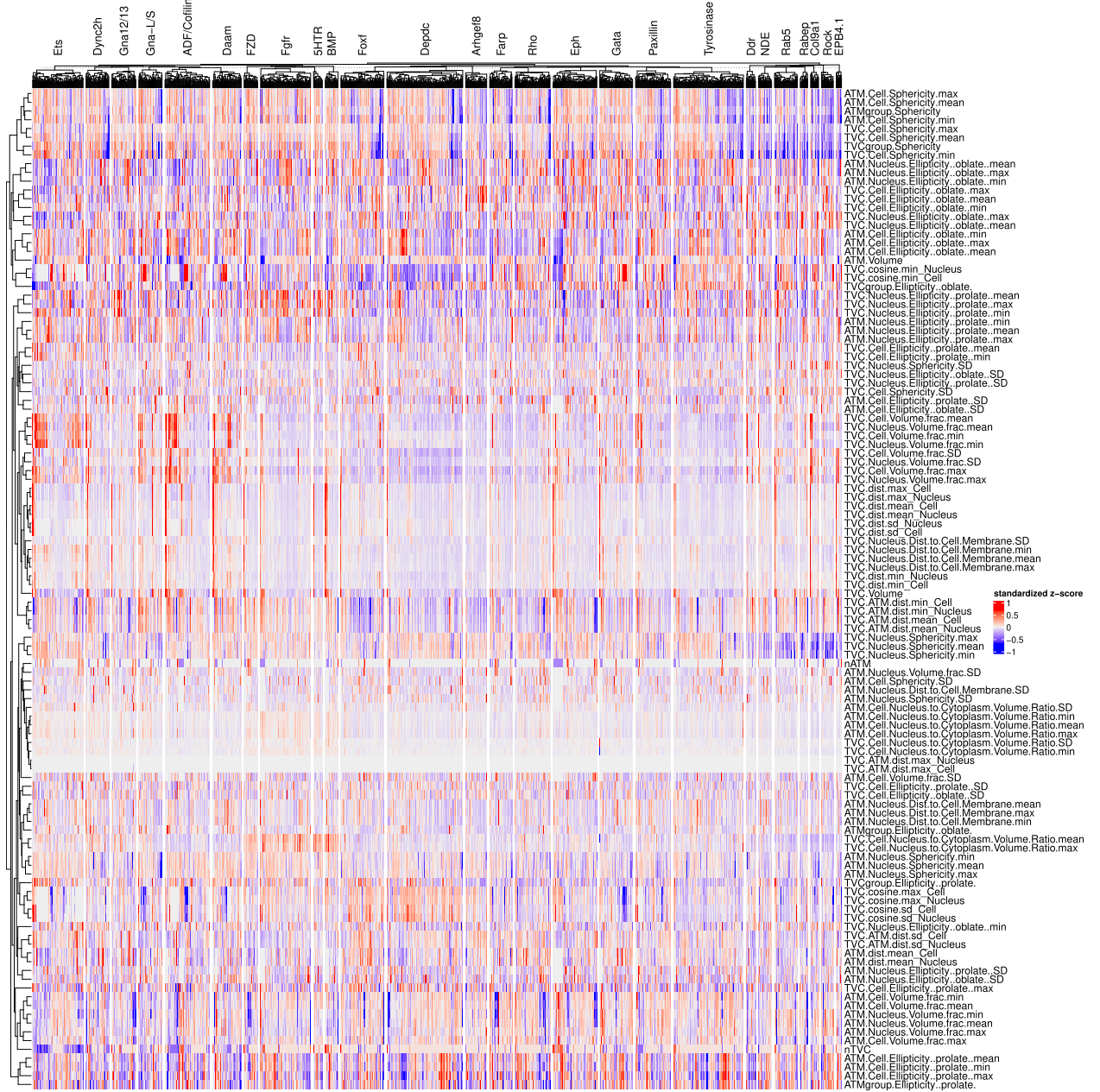


Figure 7: Preprocessed microscopy data.



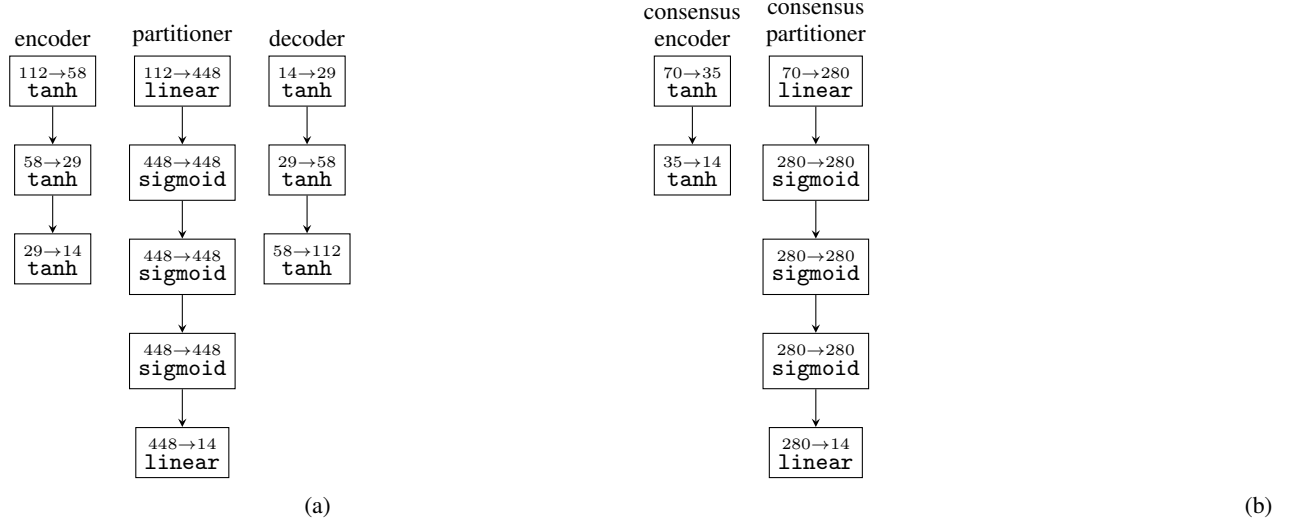


Figure 8: (a) Encoder, partitioner, and decoder submodels used for clustering. (b) Submodels used to derive consensus encoder and consensus partitioner.