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

A PREPRINT

Keira Wiechecki[1] and Jade Zaslavsky[2]

[1]Center for Genomics & Systems Biology, New York University , kaw504@nyu.edu
[2]Transcribbit

January 26, 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[4].

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

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

1.1 Notation

Details are in Appendix A.

2 Related Work

2.1 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[3]. 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[2]. This suggests the possibility of merging low level features or splitting high level features.

2.2 `noise2self`

For a large class of denoising functions, it is possible to find optimal parameters using only unlabeled noisy data[1].

See Appendix B.1.

2.3 Diffusion with weighted affinity kernels

`noise2self` is particularly useful for finding optimal parameters for generating a graph[5]. (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.

$$\hat{X} := \texttt{WAK}(G)X^\top \tag{2}$$

2.4 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.

$$\texttt{PWAK}(K) := \texttt{WAK}(K^\top K) \tag{3}$$

This lets us define a loss function

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

`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!

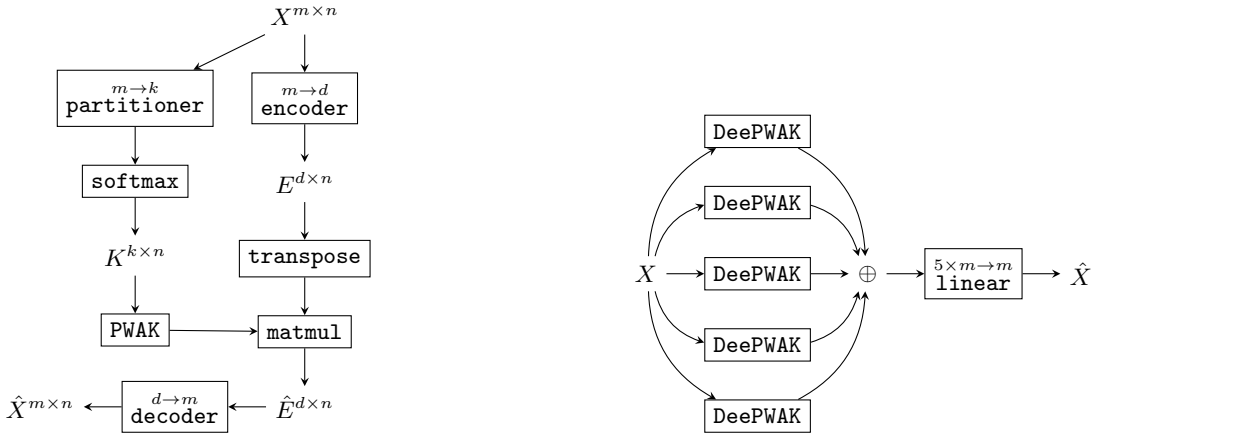

Figure 1: (a) Architecture of one DeePWAK head. (b) Architecture of one DeePWAK block with $h = 5$.

2.5 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 [7]. 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 θ and a decoder ϕ, our loss function is now

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

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. $\mathcal{L}_{\texttt{DeePWAK}}$ 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.

3 Methods

3.1 `DeePWAK`

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

3.2 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`.

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

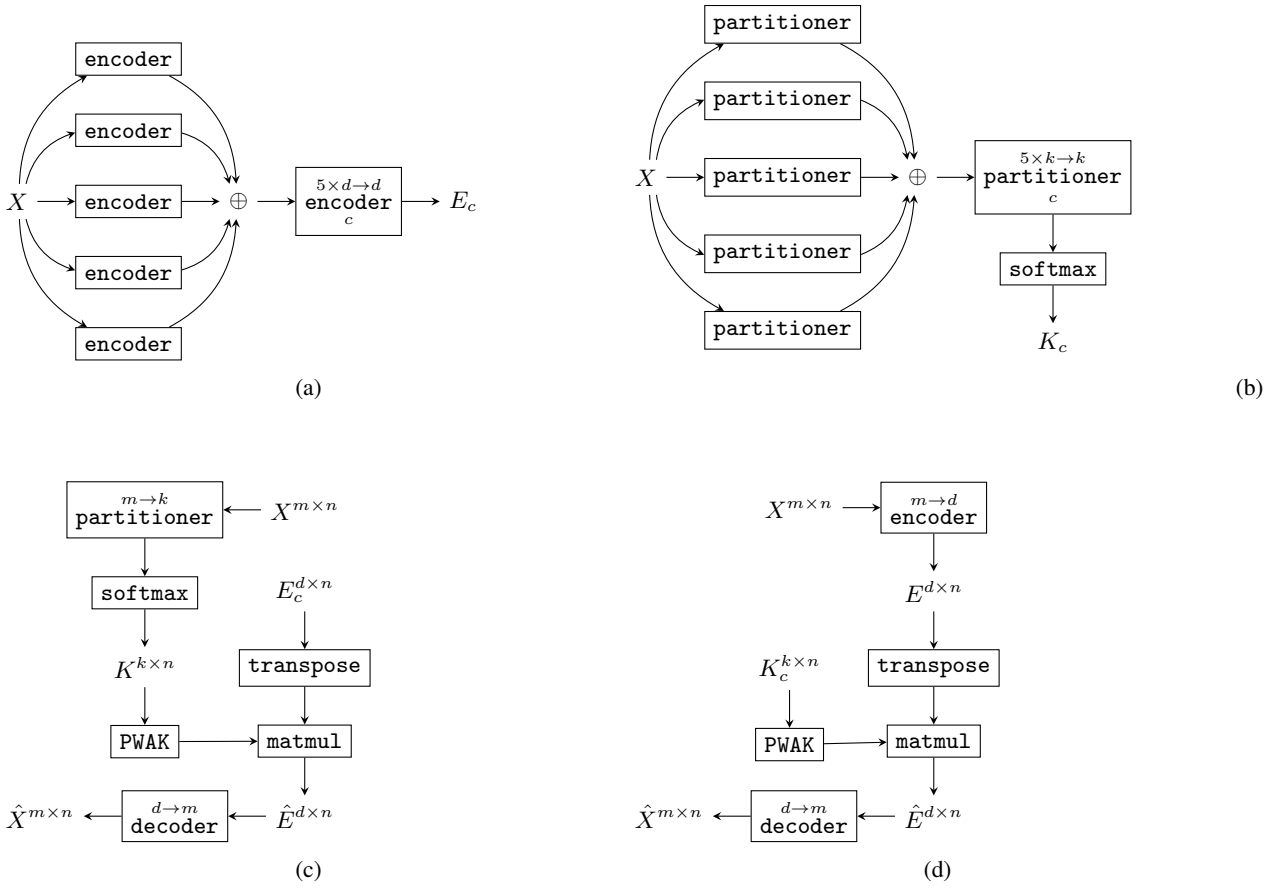

Figure 2: (a,b) Calculation of consensus E and C, respectively. (c,d) Drop-in of consensus E and C.

3.3 Recurrent DeePWAK

4 Results

4.1 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 sparcer.

It should be empasized 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.

5 Discussion

5.1 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.

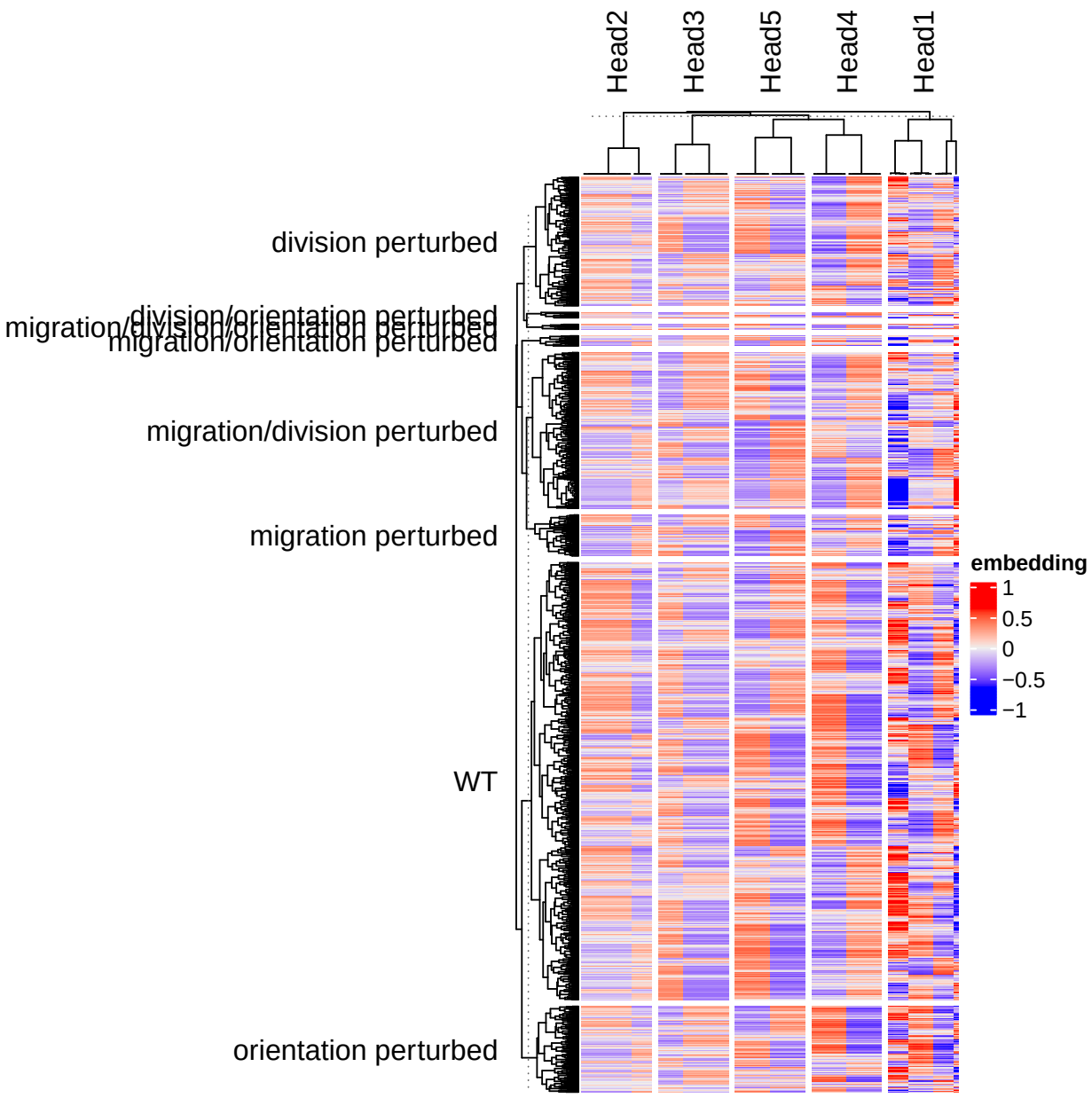

Figure 3: DeePWAK learns sparse embedding values.

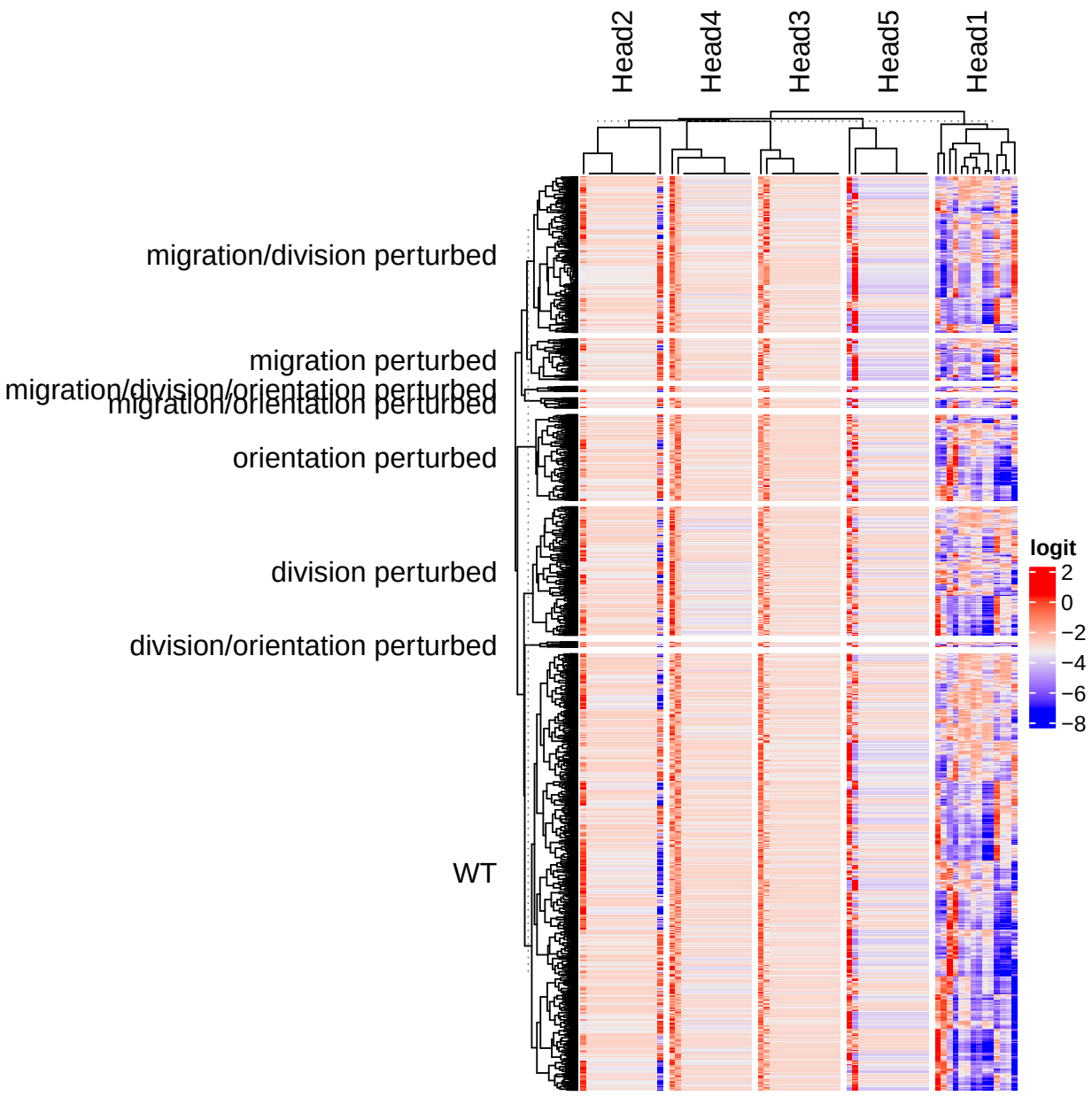

Figure 4: For most heads, two clusters dominate.

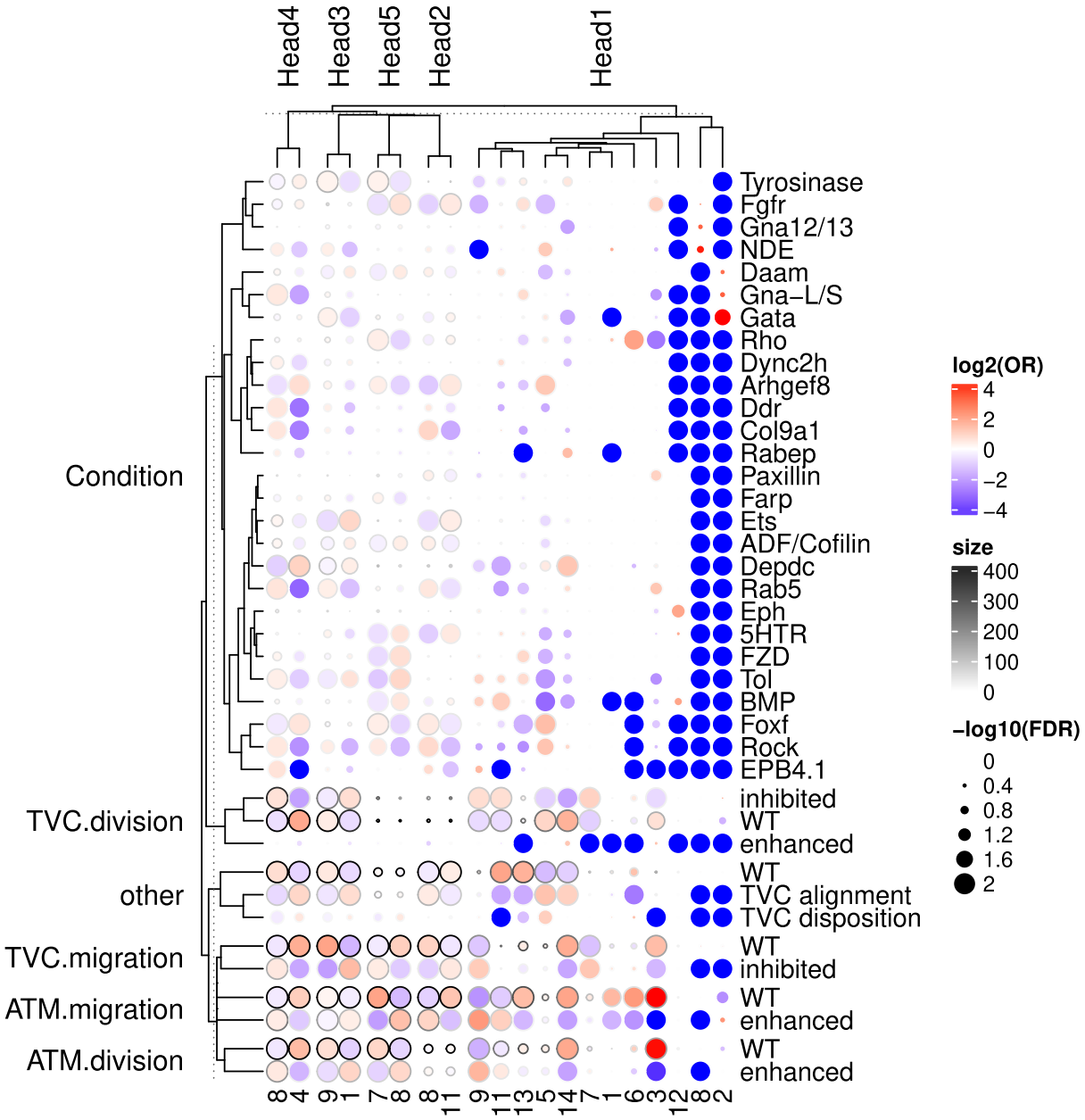

Figure 5: Hypergeometric test for enrichment of phenotype and treatment condition for each cluster in each head. Dot color indicates overrepresentation of each category in a cluster compared to a uniform prior. Dot size indicates statistical significance of the difference. Dot outline shade indicates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

5.2 The future of SAEs

We suspect combinatoric encoding of this sort is possible because `DeePWAK` performs computation on an entire mini-batch.

5.3 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`

6 Conclusion

References

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

[2] 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.

[3] Hoagy Cunningham et al. *Sparse Autoencoders Find Highly Interpretable Features in Language Models*. 2023. arXiv: `2309.08600 [cs.LG]`.

[4] 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`.

[5] 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`.

[6] 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`.

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

A 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.

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

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

$\texttt{List}(\texttt{Type}, n)$ 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.

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 $\theta \in \mathbf{\Theta}$ that depends on its complement X_{JC}

$$\hat{X}_J = \mathcal{F}(\theta)(X_{JC}) \tag{6}$$

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

$$\underset{\theta}{\overset{\mathbf{\Theta}}{\texttt{noise2self}}}[\mathcal{F}(\theta), X] := \underset{\theta}{\overset{\mathbf{\Theta}}{\texttt{argmin}}}[\sum_{J}^{\mathcal{J}} \mathbb{E}[X_J - \mathcal{F}(\theta)(X_{JC})]^2] \tag{7}$$

B.2 DEWAKSS

DEWAKSS [5] is a method of data imputation to address sparcity 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[6], 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
    $\mathbf{w} \leftarrow \overset{ij \to j}{\texttt{einsum}}(G_{-I})$ // sum over columns
    $W \leftarrow \overset{ij,i \to j}{\texttt{einsum}}(G_{-I}, \mathbf{w}^{-1})$ // scale columns to sum to 1
    return W
  end
end

Algorithm 1: Weighted Affinity Kernel

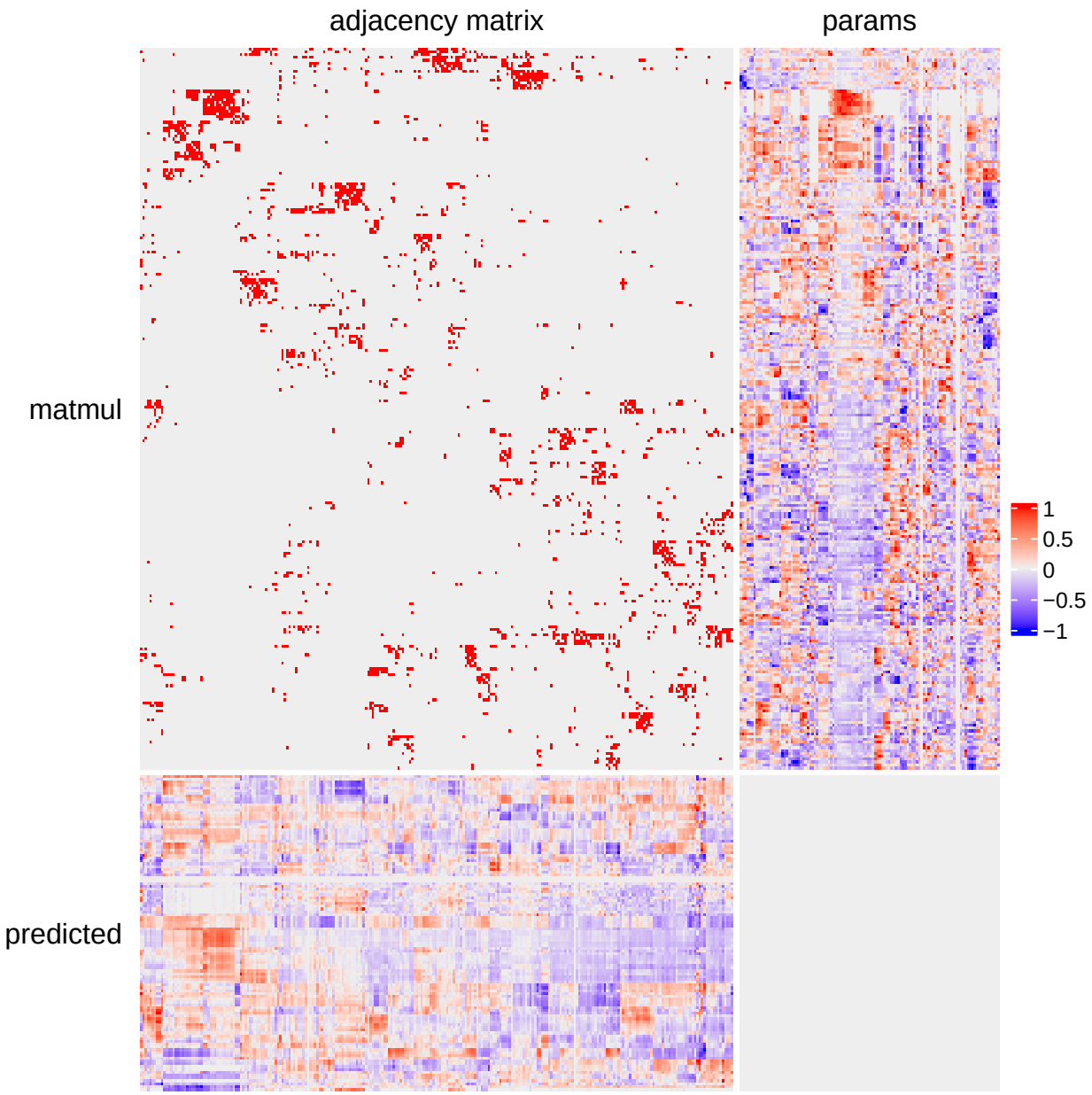

Figure 6: Illustration of diffusion with `DEWAKSS`. In this case the kernel is a 10-NN computed using the first 10 PCs of the data. Note that the denoised result is substantially smoother than the input data. While smoothing is often desirable for data imputation, for our purposes we would prefer to perserve 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
  $E \leftarrow \texttt{pca}(X)$ // 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
      $W \leftarrow \texttt{WAK}(G \odot D)$ // 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.

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: $\mathcal{K}$: `Arch(`m`,`k`)` // partitioner architecture
  Output: $\mathcal{P}$: $\texttt{Params}(\mathcal{K}) \rightarrow \texttt{Model}(m, k)$
end

Algorithm 3: Partitioner constructor

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

Algorithm 4: Partitioner application

Function $(\texttt{train}(\mathcal{P}))(\pi, X)$
  $\forall \mathcal{K} : \texttt{Arch}(m, k)$ // partitioner architecture
  $\forall \mathcal{P} : \texttt{Partitioner}(\mathcal{K})$ // partitioner
  Hyperparameters :
  $t, b : \mathbb{N}$ // epochs, batch size
  $\eta : \mathbb{R}$ // learning rate
  Input:
  $\pi : \texttt{Params}(\mathcal{K})$ // partitioner parameters
  $X : \mathbb{R}^m$ // input data
  Output:
  $\pi : \texttt{Params}(\mathcal{K})$
  begin
    for $\_ \in \{1 : t\}$ **do**
      $X \leftarrow \texttt{sample}(X[:,], b)$ // randomly sample data
      Function $\texttt{loss}(\kappa)$
        Input: $\kappa : \texttt{Params}(\mathcal{K})$ // parameters to update
        Output: $l : \mathbb{R}$ // mean squared error
        begin
          $K \leftarrow \mathcal{K}(\kappa, X)$ //
          $\hat{X} \leftarrow \texttt{PWAK}(K)X^\top$ //
          $l \leftarrow \texttt{MSE}(\hat{X}, X)$ //
          return l
        end
      end
      $\pi \leftarrow \pi - \eta \nabla \texttt{loss}(\pi)$ // gradient descent
    end
    return π
  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[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 $\texttt{DeePWAK}(\mathcal{E}, \mathcal{K}, \mathcal{D})$
  $\forall m, d, k : \mathbb{N}$ // input dimension, embedding dimension, max clusters
  Input:
  $\mathcal{E} : \texttt{Arch}(m,d)$ // encoder architecture
  $\mathcal{K} : \texttt{Arch}(m,k)$ // partitioner architecture
  $\mathcal{D} : \texttt{Arch}(d,m)$ // decoder architecture
  Output:
  $\mathcal{W} : \texttt{Params}(\mathcal{E}) \rightarrow \texttt{Params}(\mathcal{K}) \rightarrow \texttt{Params}(\mathcal{D}) \rightarrow \texttt{Model}(m, m)$
end

Algorithm 6: DeePWAK constructor

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
    $\hat{E} \leftarrow (\texttt{PWAK}(K)E^\top)^\top$ // 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 $\texttt{DeePWAKBlock}(\mathcal{W}, h)$
  $\forall \mathcal{E} : \texttt{Arch}(m, d)$ // encoder template
  $\forall \mathcal{K} : \texttt{Arch}(m, k)$ // partitioner template
  $\forall \mathcal{D} : \texttt{Arch}(d, m)$ // decoder template
  Input:
  $\mathcal{W} : \texttt{DeePWAK}(\mathcal{E}, \mathcal{K}, \mathcal{D})$ // DeePWAK architecture
  $h : \mathbb{N}$ // number of heads
  Output:
  $\texttt{List}(\texttt{Params}(\mathcal{E}), h) \to \texttt{List}(\texttt{Params}(\mathcal{K}), h) \to \texttt{List}(\texttt{Params}(\mathcal{D}), h) \to \texttt{Params}(\texttt{Linear}(hm, m)) \to \texttt{Model}(m, m)$
end

Algorithm 8: DeePWAKBlock constructor

Function $(\texttt{DeePWAKBlock}(\mathcal{W}, h))(\Theta, \Pi, \Phi, \psi, X)$
  $\forall \mathcal{W} : \texttt{DeePWAK}(\mathcal{E}, \mathcal{K}, \mathcal{D})$ `// DeePWAK template`
  Input:
  $\Theta : \texttt{List}(\texttt{Params}(\mathcal{E}), h)$ `// encoders`
  $\Pi : \texttt{List}(\texttt{Params}(\mathcal{K}), h)$ `// partitioners`
  $\Phi : \texttt{List}(\texttt{Params}(\mathcal{D}), h)$ `// decoders`
  $\psi : \texttt{Params}(\texttt{Linear}(hm, m))$ `// 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`
    $f \leftarrow \texttt{Linear}(hm, m)(\psi)$ `// 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.

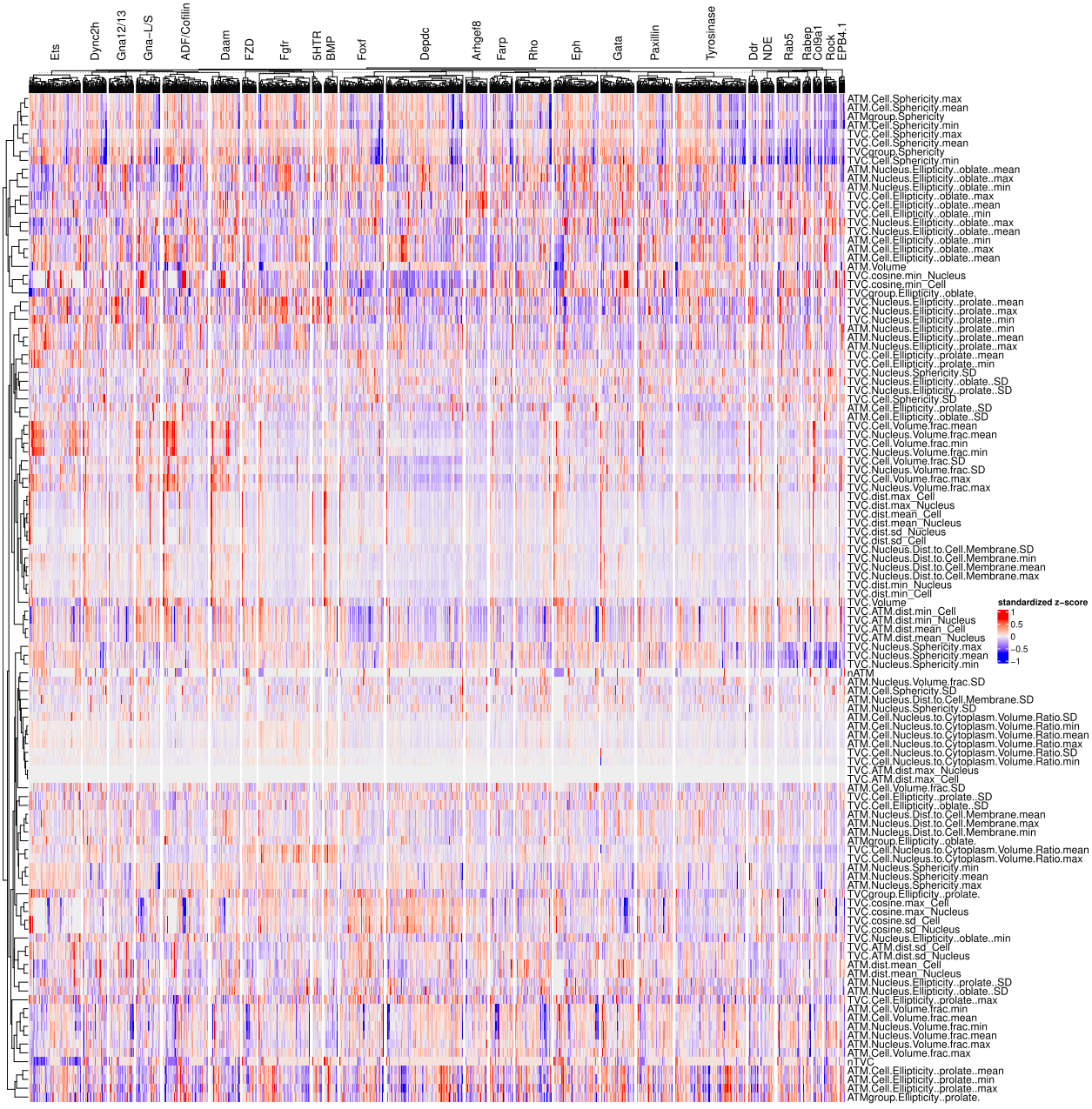

Figure 7: Preprocessed microscopy data.

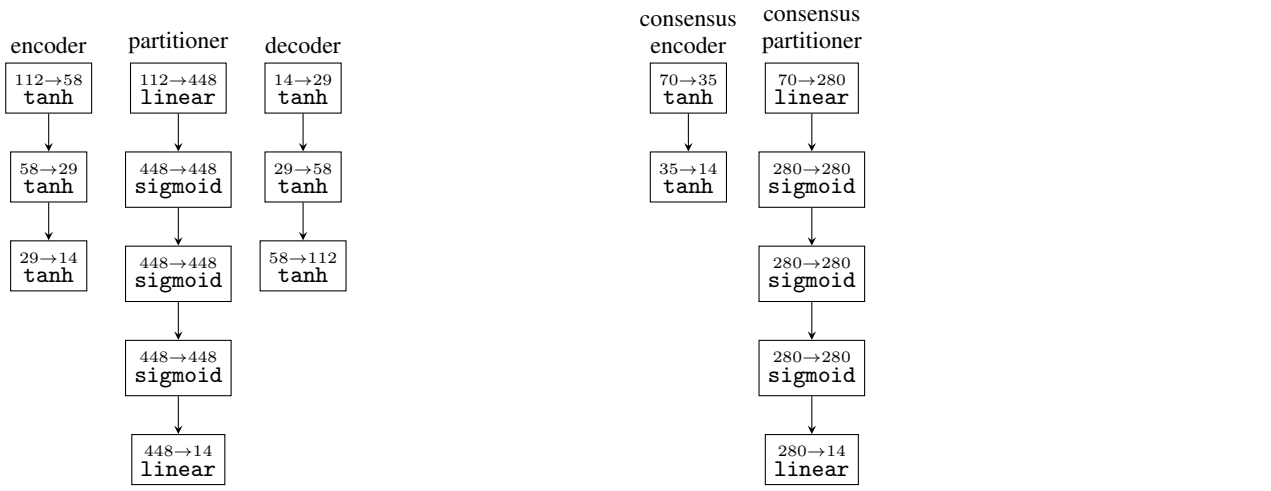

Figure 8: (a) Encoder, partitioner, and decoder submodels used for clustering. (b) Submodels used to derive consensus encoder and consensus partitioner.