

## 645 Appendix

### 646 A Software Framework: The PerturbBench Library

647 The PerturbBench library is designed to encapsulate the essential elements of perturbation modeling,  
648 prioritizing reusability and flexibility for researchers. It integrates seamlessly with leading Python-  
649 based machine learning frameworks including PyTorch, PyTorch Lightning, and Hydra, as well  
650 as cutting-edge single-cell analysis libraries such as Scanpy and AnnData. Our design choices  
651 streamline the training of innovative model architectures and the assessment of both existing and  
652 novel techniques across a comprehensive range of benchmarks and datasets. The library is structured  
653 into three core modules: `data`, `model`, and `analysis`. These modules are engineered to work  
654 together to support a variety of applications, from complete model development to modular use for  
655 integration with other tools and analytical assessments. Subsequent sections will detail the primary  
656 abstractions each module offers, illustrating their practicality and adaptability for diverse research  
657 tasks.

#### 658 A.1 Foundational Frameworks

659 PerturbBench leverages contemporary machine learning and single-cell analysis libraries that are  
660 prevalent within their respective research communities. This strategic choice is intended to lower  
661 the adoption barrier for the proposed benchmark. Additionally, these libraries offer comprehensive  
662 guidelines on usage patterns and best practices, which serve to inform the organizational structure of  
663 the code.

664 **Pytorch:** is one the most widely spread neural network libraries [40]. Its core functionality is to  
665 build computational graphs with support for efficient auto differentiation. Using this autodiff engine,  
666 Pytorch then provides abstractions to build and optimize various neural networks and ML algorithms.  
667 In addition, it provides utilities to load and serve data under different training regimes. These concepts  
668 are captured within the `torch.utils.data.Dataset` and `torch.utils.data.DataLoader` ab-  
669 stractions. We use these to implement our `perturbbench.data` module.

670 **Pytorch Lightning:** While Pytorch provides most of the functionality to train any neural network  
671 model, it could still be a challenging task to write training and evaluation code that can be portable  
672 across different platforms, have minimal boiler plate code, and be easy to read and understand.  
673 Pytorch Lightning is a library that builds on top Pytorch Lightning to provide [17]: 1) hardware agnostic  
674 model implementations, 2) clear easy to read codebase with minimal boiler plate code, 3) reproducible  
675 experiments, and 4) integration with popular machine learning tools. We wrap our models and data  
676 into Lightning’s `LightningModule` and `LightningDataModule` to abstract away most of the code  
677 for managing model training and serving data. Then we leverage Lightning’s `Trainer` that abstracts  
678 the various training loops to write generic train and evaluation scripts. Furthermore, Lightning’s  
679 `Callback` to integrate various logging libraries such as TensorBoard.

680 **Hydra:** A complex benchmarking suite needs to configure its large number of components and  
681 to provide a simple summary for reproducing any experiment. In `perturbbench`, we utilize Hydra  
682 [59] for managing configuration files. Hydra provides a hierarchical configuration system that can  
683 be composed based on the components of the system being configured. In addition, it provides  
684 convenient tools such as a command line interface (cli) with auto-completion, support for HPO via  
685 `optuna`, and basic type checking.

686 **AnnData** We use AnnData as our primary format for storing and interacting with single cell  
687 RNA-seq datasets [57]. Each AnnData object contains a single cell gene expression matrix with  
688 associated cell level metadata such as perturbation and covariates, as well as gene level metadata such  
689 as gene name and ID. Our data module expects single cell datasets stored as AnnData h5ad files and  
690 our analysis module expects model predictions in the form of AnnData objects.

## 691 A.2 Data Abstractions

692 The library is built around the `Example` abstraction given in Listing 1 that represents a single datum  
 693 and its batched version. This data structure contains the necessary fields required for the training and  
 694 evaluation of perturbation prediction models and serves to unify the model/data API. Each example  
 695 has two required fields: a `gene_expression` 1D tensor that contains the gene expression levels,  
 696 and the perturbations that has the list perturbation names that has been applied to this cell. In  
 697 addition, the example contains some optional fields that support more complex functionality like  
 698 using pre-computed embeddings in the `extra` field, or control matching via `controls`. An ordered  
 699 list of gene names can be provided in `gene_names` such that it is ordered according to the provided  
 700 gene counts in `gene_expression`.

```
701 | class Example(NamedTuple):
702 |     """Single Cell Expression Example."""
703 |
704 |     gene_expression: Tensor
705 |     perturbations: list[str]
706 |     covariates: dict[str, str] | None = None
707 |     controls: Tensor | None = None
708 |     gene_names: list[str]
709 |     extra: dict[str, Any]
```

Listing 1: Data structure representing a single example.

710 For training, we provide two types of pytorch datasets. The `SingleCellPerturbation` class repre-  
 711 sents a single cell RNA-seq dataset (Listing 2) and the `SingleCellPerturbationWithControls`  
 712 class adds control matching functionality, sampling a matched control cell for every perturbed cell  
 713 (Listing 3).

```
714 | class SingleCellPerturbation(Dataset):
715 |     """Single Cell Perturbation Dataset."""
716 |
717 |     gene_expression: Tensor
718 |     perturbations: list[list[str]]
719 |     covariates: dict[str, list[str]] | None = None
720 |     cell_ids: list[str] | None = None
721 |     gene_names: list[str] | None = None
722 |     transforms: Callable | None = None
723 |     extra: dict[str, Any]
724 |
725 |     # factory method
726 |     @staticmethod
727 |     def from_anndata(
728 |         adata: ad.AnnData,
729 |         perturbation_key: str,
730 |         perturbation_combination_delimiter: str | None,
731 |         covariate_keys: list[str] | None = None,
732 |         perturbation_control_value: str | None = None,
733 |         embedding_key: str | None = None,
734 |     ) -> tuple[SingleCellPerturbation, dict[str, Any]]:
735 |         ...
```

Listing 2: Pytorch dataset classes for training.

```

736 1 class SingleCellPerturbationWithControls(SingleCellPerturbation):
737 2     """Single Cell Perturbation Dataset with matched controls."""
738 3
739 4     control_ids: Sequence[str] | None = None
740 5     control_indexes: Map(CovariateDict, list[int])
741 6     control_expression: Tensor
742 7
743 8     # factory method
744 9     @staticmethod
745 10     def from_anndata(
746 11         adata: ad.AnnData,
747 12         perturbation_key: str,
748 13         perturbation_combination_delimiter: str | None,
749 14         covariate_keys: list[str] | None = None,
750 15         perturbation_control_value: str | None = None,
751 16         embedding_key: str | None = None,
752 17     ) -> tuple[SingleCellPerturbation, dict[str, Any]]:
753 18         ...

```

Listing 3: Pytorch dataset classes for training.

For inference, we provide two additional types of pytorch datasets. The Counterfactual dataset represents a desired set of counterfactual predictions. Since these counterfactual predictions are applied to unperturbed control cells, we only need to store control cell expression values. A single item of this dataset is a counterfactual perturbation applied to set of control cells with will return a Batch of data with the control cell expression, control covariates, and desired perturbation.

To evaluate counterfactual predictions, we also provide the CounterfactualWithReference class which inherits from the Counterfactual class. In addition to providing a Batch of control cells with covariates and the desired perturbation, this class also provides an AnnData object with the gene expression values for the perturbed cells corresponding to the covariates and desired perturbations. This enables us to use our suite of benchmarking metrics to compare the model predictions with the observed data. We provide the classes in Listing 4.

```

765 1 class Counterfactual(Dataset):
766 2     """Counterfactual Dataset."""
767 3     # Desired counterfactual perturbations
768 4     perturbations: Sequence[list[str]]
769 5     covariates: dict[str, Sequence[str]]
770 6     control_expression: SparseMatrix
771 7     control_indexes: FrozenDictKeyMap
772 8     gene_names: Sequence[str] | None = None
773 9     transforms: InitVar[Callable | Sequence[Callable] | None] = field(
774     default=None)
775 10     info: dict[str, Any] | None = None
776 11     control_embeddings: np.ndarray | None = None
777 12
778 13 class CounterfactualWithReference(Counterfactual):
779 14     """Counterfactual Dataset with matched Reference Data."""
780 15     # A map from a unique perturbation and set of covariates to
781     indexes
782 16     # in the reference_adata (i.e. all indexes that contain k562 cells
783 17     # with AGR2 knocked down)
784 18     reference_indexes: dict[str, FrozenDictKeyMap] | None = None
785 19     # An AnnData object containing the observed perturbational dataset
786 20     # matching the desired counterfactual predictions
787 21     reference_adata: ad.AnnData | None = None

```

Listing 4: Pytorch dataset classes for inference.

### 788 A.3 Data Splitting

789 We implement a datasplitter class that can generate three types of datasplits:

- 790 1. Cross covariate splits that ask a model to predict a perturbation's effect in covariate(s) that  
791 were not in the training split. The model will have seen other perturbation in the covariate(s).
- 792 2. Combinatorial splits that ask a model to predict the effect of multiple perturbations. The  
793 model will have seen the individual perturbations and some other combinations.
- 794 3. Inverse combinatorial splits that ask a model to predict the effect of a single perturbation  
795 when it has seen a dual perturbation and the other single perturbation.

796 We design a data splitter with two parameters that allow us to curate the splits: (1) The maximum  
797 number,  $m$ , of cell types (covariates) to hold out. We randomly hold out between one and  $m$  cell  
798 types (sampled uniformly). The more cell types held out, the more challenging the task becomes due  
799 to fewer training cell types. (2) The total fraction of perturbations held out per cell type,  $f$ . A larger  
800 fraction makes it more difficult for the model to generate accurate predictions. The datasplitter can  
801 also read in custom splits from disk as a csv file.

### 802 A.4 Model Abstraction: Model Base Class

803 We implement a base model class, `PerturbationModel`, that abstracts away common model com-  
804 ponents that we describe in Listing 5. This class specifically contains:

- 805 • A default optimizer
- 806 • A training record that contains the data transforms and other key metadata needed for  
807 training and inference
- 808 • Methods for generating and evaluating counterfactual predictions

```
809 1 class PerturbationModel(L.LightningModule, ABC):  
810 2     """A base model class for perturbation prediction models."""  
811 3     training_record: dict = {  
812 4         'transform': None,  
813 5         'train_context': None,  
814 6         'n_total_covs': None,  
815 7     }  
816 8     evaluation_config: DictConfig | None = None  
817 9     summary_metrics: pd.DataFrame | None = None  
818 10    prediction_output_path: str | None = None  
819 11  
820 12    def configure_optimizers(self):  
821 13        """Base optimizer for lightning Trainer."""  
822 14  
823 15    def predict_step(  
824 16        self,  
825 17        data_tuple: tuple[Batch, pd.DataFrame],  
826 18        batch_idx: int,  
827 19    ) -> ad.AnnData | None:  
828 20        """Given a batch of data, predict the counterfactual perturbed  
829 21        """  
830 22  
831 23    def test_step(  
832 24        self,  
833 25        data_tuple: tuple[Batch, pd.DataFrame, ad.AnnData],  
834 26        batch_idx: int,  
835 27    ):  
836 28        """Run evaluation on a Batch of counterfactual predictions and  
837 29        matched observed predictions."""  
838 30  
839 31    def on_test_end(self) -> None:  
840 32        """Run rank evaluations (if specified) and summarize  
841 33        benchmarking
```

```

842 2         metrics."""
843 3
844 4     @abstractmethod
845 5     def predict(self, counterfactual_batch: Batch) -> torch.Tensor:
846 6         """Given a counterfactual_batch of data,
847 7             predict the counterfactual perturbed expression.
848 8         """

```

Listing 5: Pytorch dataset classes for inference.

## 849 A.5 Evaluation

850 All models that inherit from the base `PerturbationModel` class will be able to run evaluation using  
851 the Pytorch Lightning trainer test step. These evaluations can be configured via Hydra if using our  
852 `train.py` script and evaluations can be run automatically after training completes. For users who  
853 only want to use our evaluation metrics, we offer a kaggle style evaluation API that takes as input  
854 model predictions as an `AnnData` object, with an example in Listing 6.

```

855 1 from perturbench.analysis.benchmarks.evaluator import Evaluator
856 2
857 3 # List available tasks
858 4 print(Evaluator.list_tasks())
859 5
860 6 # Select an evaluation task
861 7 evaluator = Evaluator(
862 8     task = "sciplex3-transfer",
863 9 )
864 10 # The input format of the Evaluator class is a
865 11 # dictionary of model predictions stored as AnnData objects
866 12 input_dict = {"CPA_pred": cpa_pred} # cpa_pred is an AnnData object
867 13 result_df = evaluator.evaluate(input_dict)
868 14 print(result_df) # Summary dataframe with evaluation metrics

```

Listing 6: Evaluation API usage example.

## 869 **B Additional Modeling Background**

### 870 **B.1 Perturbation embeddings**

871 **Drug Embeddings** It can be beneficial to use pre-trained embeddings to enable or enhance predic-  
872 tive performance of perturbation models, for instance, ESM embeddings for gene expression [46].  
873 The performance of these models in predicting unseen perturbations is dependent on the quality of  
874 the perturbation representation, which is itself a complex task [22, 48, 46] and outside the scope  
875 of this study. GEARS uses gene co-expression to build a gene to gene graph [47], PerturbNet uses  
876 a perturbation encoder network to encode perturbations into a lower dimensional embedding [60].  
877 For drug perturbations, PerturbNet uses a structure encoder and for genetic perturbations, it models  
878 the gene as a multi-hot vector over all gene ontology annotations. The authors of CPA include a  
879 variation to their original model that embeds drugs into a lower dimensional space using molecular  
880 features [35]. scFoundation leverages GEARS but instead of constructing the graph using static gene  
881 coexpression, it uses the gene embeddings for a given cell to create a gene-gene graph [21]. The  
882 performance of these models in predicting unseen perturbations is dependent on the quality of the  
883 perturbation representation, which is itself a complex task [22, 48, 46] and outside the scope of this  
884 study.

## C Further Results

### C.1 Generalizing from less complex to more complex biological systems

We also apply PerturBench to a highly relevant real-world task: predicting perturbation effects in more complex disease system using effects measured in less complex systems. The [Frangieh21](#) dataset contains 3 biological systems: primary melanoma cells cultured alone, or with  $\text{IFN}\gamma$ , or co-cultured with tumor infiltrating immune cells. We held out 70% of the perturbations in the co-culture system and used the remaining perturbations as well as all perturbations in the other systems as training.

The results are summarized in Table 5. Our baseline models such as LA and Decoder generally perform better on the rank metrics, whereas more sophisticated models such as SAMS-VAE\* (S) perform better on cosine LogFC and RMSE. Again, model simplification consistently results in performance gains, as in the case of CPA\* (noAdv) and SAMS-VAE\* (S). Overall, [Frangieh21](#) shows that biological heterogeneity of datasets has a major impact on model comparison, again highlighting the need to include diverse datasets for benchmark.

Aside from this note, we also observe that the vanilla SAMS-VAE\* suffers from mode collapse, as indicated by its near-random rank scores on cosine LogFC and RMSE. Indeed, we notice the model is generating generic expression vectors irrespective of the target perturbation, see details in Appendix C.3. Further dissecting SAMS-VAE\* performance, we find that model has learnt near-identical embedding vectors for any perturbations. This suggests that despite the theoretical advantage of sparse additive mechanism, effectively optimizing the model in practice remains a non-trivial question, and can lead to degenerate scenarios which fortunately can be uncovered by our rank metric.

Table 5: Results of a *covariate transfer* experiment generalizing from less complex biological systems to a more complex co-culture system in the [Frangieh21](#) dataset.

Model	Cosine LogFC	RMSE mean	Cosine LogFC rank	RMSE mean rank
CPA*	$0.10 \pm 7 \times 10^{-3}$	$0.027 \pm 1 \times 10^{-4}$	$0.30 \pm 3 \times 10^{-2}$	$0.28 \pm 1 \times 10^{-2}$
CPA* (noAdv)	$0.10 \pm 7 \times 10^{-3}$	$0.027 \pm 9 \times 10^{-5}$	<b><math>0.20 \pm 3 \times 10^{-3}</math></b>	$0.19 \pm 3 \times 10^{-2}$
CPA* (scGPT)	$0.07 \pm 4 \times 10^{-3}$	$0.029 \pm 2 \times 10^{-4}$	$0.26 \pm 5 \times 10^{-3}$	$0.26 \pm 1 \times 10^{-2}$
SAMS-VAE*	$0.15 \pm 2 \times 10^{-2}$	$0.026 \pm 2 \times 10^{-4}$	$0.48 \pm 2 \times 10^{-2}$	$0.46 \pm 2 \times 10^{-2}$
SAMS-VAE* (S)	<b><math>0.22 \pm 3 \times 10^{-3}</math></b>	<b><math>0.025 \pm 5 \times 10^{-5}</math></b>	$0.22 \pm 8 \times 10^{-3}$	$0.22 \pm 1 \times 10^{-2}$
BioLord*	$0.12 \pm 9 \times 10^{-3}$	$0.027 \pm 2 \times 10^{-4}$	$0.29 \pm 3 \times 10^{-2}$	$0.21 \pm 4 \times 10^{-2}$
LA	$0.17 \pm 6 \times 10^{-3}$	<b><math>0.024 \pm 4 \times 10^{-4}</math></b>	$0.26 \pm 1 \times 10^{-2}$	$0.21 \pm 2 \times 10^{-2}$
LA (scGPT)	$0.18 \pm 6 \times 10^{-3}$	<b><math>0.024 \pm 6 \times 10^{-5}</math></b>	$0.27 \pm 1 \times 10^{-2}$	$0.24 \pm 1 \times 10^{-2}$
Decoder	$0.10 \pm 2 \times 10^{-3}$	<b><math>0.025 \pm 4 \times 10^{-5}</math></b>	<b><math>0.21 \pm 5 \times 10^{-3}</math></b>	<b><math>0.15 \pm 4 \times 10^{-4}</math></b>
Linear	$0.01 \pm 4 \times 10^{-4}$	$0.043 \pm 7 \times 10^{-5}$	$0.24 \pm 9 \times 10^{-4}$	$0.30 \pm 2 \times 10^{-3}$

### C.2 Effect of Data Imbalance

An important consideration with perturbation response models is how robust they are to *imbalanced data* i.e. how evenly data is distributed across covariates. We quantify imbalance using normalized entropy as follows:

$$\text{Imbalance} := 1 - \frac{\sum_{i=1}^k \frac{n_i}{n} \log \frac{n_i}{n}}{\log k},$$

where  $n_i, i = 1, \dots, k$  denotes the number of samples in class  $i$  and  $n = \sum_{i=1}^n n_i$  the overall number of observations. The [Srivatsan20](#) data is perfectly balanced (Imbalance = 0), with every perturbation being observed in every cell type. However, *in-silico* machine learning perturbation models often aim to learn generalizable features by using data from multiple sources, which will invariably produce imbalanced datasets.

To test how different models' performance is affected by data imbalance we downsample perturbations per cell type from [Srivatsan20](#) to construct three sub-datasets with different levels of imbalance (Appendix D.4). The results are summarized in Figure C.1. We observe that when the data is highly

balanced, the linear model performs acceptably well, but this does not hold as imbalance increases. Imbalance may therefore be an important criteria for deciding the suitability of a linear model. CPA both with and without scGPT embeddings, is more robust to changes in data balance than the the linear or latent additive models. Interestingly, whilst the latent additive model is more markedly affected by data imbalance than other models, using scGPT embeddings seems to buffer performance by some extent. The extent to which performance is affected by data imbalance highlights the importance of curated datasets as well as potential mitigation strategies such as oversampling rare cell types [14].

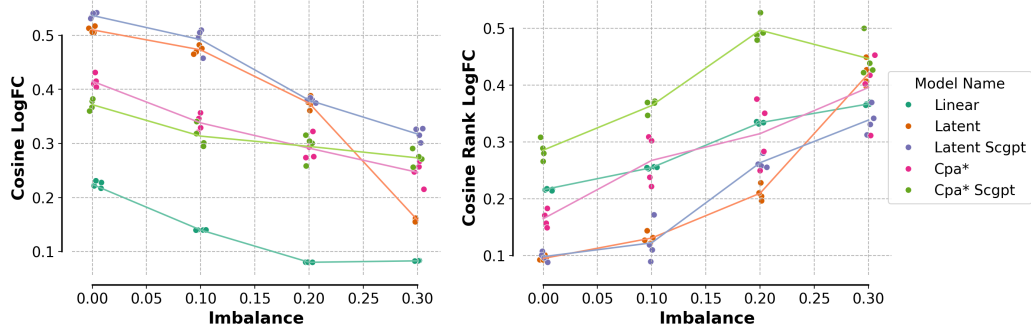


Figure C.1: Cosine similarity of log fold changes (left) and its rank (right) of the models as a function of data balance.

### C.3 Collapse and Rank Metrics

*Mode or posterior collapse* is a common failure mode in generative models, notably in Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs). The problem arises when a generative model inadequately captures the diversity of training data, resulting in repetitive outputs that are effectively collapsing onto a set of limited modes in the data distribution. For VAEs, posterior collapse indicates the latent variables lack meaningful information and are ignored by the decoder. This effectively rids a VAE of any mechanism to control the properties of generated samples, which in practice also leads to mode collapse.

For perturbation response modelling, these issues are particularly concerning as the predicted responses are expected to be perturbation-specific, so that they are able to capture the nuances of perturbation effects across a diverse set of cell types, treatments and other conditions. This is above-all essential for inferring the distinct effect of perturbations on various biological processes, as well as ranking and identifying targets for the disease of interest.

However, as we will demonstrate, traditional measures such as RMSE, cannot distinguish between mode collapse or not, because models that simply predict the the average expression level for a particular cell type can still achieve relatively good performance. This motivates our rank metrics which can act as a safeguard against such degenerate cases, to some extent. Yet, we reveal its limitations through comparison with the visual assessment, and propose two new metrics that are more diagnostic of mode collapse.

Table 6: Numerical quantification of mode collapse for models presented in [Srivatsan20](#) dataset.

Model	RMSE mean	RMSE mean rank	RMSE mean transposed-rank	Matrix distance
DecoderOnly	0.026	0.488	0.482	49.906
DecoderOnly (+ perturbations)	0.021	0.116	0.232	40.645
CPA*	0.022	0.104	0.323	36.343
CPA* (noAdv)	0.021	0.098	0.337	32.238
SAMS-VAE* (S)	0.020	0.111	0.179	19.301



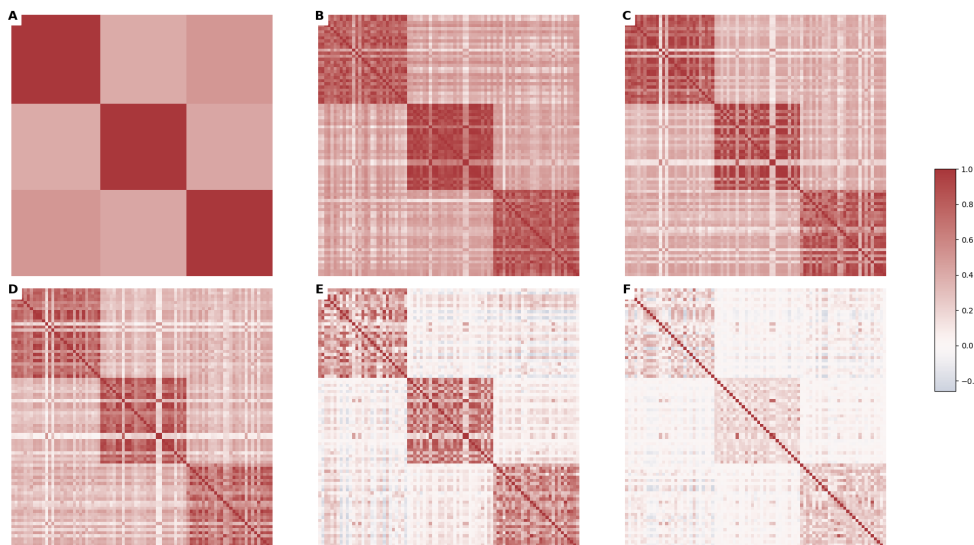


Figure C.2: Cosine similarity matrix based on log-fold changes predicted, between every pair of perturbation-covariate combination in [Srivatsan20](#) dataset. All models are hyperparameters optimised. **A)** DecoderOnly model with only covariates as input. **B)** DecoderOnly model with covariates and perturbations as input. **C)** CPA\*. **D)** CPA\* (noAdv). **E)** SAMS-VAE\* (S). **F)** true log-fold changes in the dataset. Diagonal blocks correspond to cell types: A549, K562, MCF-7.

### 944 C.3.1 Mode collapse in [Srivatsan20](#) dataset

945 Consider an example based on the [Srivatsan20](#) dataset which measures the perturbational effects of  
 946 small molecules applied to three cancer cell lines: A549, K562, MCF-7. We look at five models: 1)  
 947 our DecoderOnly with *only* cell type as input, 2) our DecoderOnly with cell type *and* perturbations  
 948 as input, 3) the vanilla CPA\*, 4) CPA\* with no adversarial components, and 5) the simplified version  
 949 of SAMS-VAE\*. In addition, we compare the model predictions to the experimental data.

950 Results are shown in Figure C.2. Of the 85 unique cell type and perturbation combinations in the  
 951 validation split, we measure the similarity between the predicted log-fold changes for every pair of  
 952 such combinations. In total, these similarities form a  $85 \times 85$  matrix which is shown as a heatmap.

953 The experimental data in **F)** shows the similarity of the log-fold changes between different pertur-  
 954 bations to be small within each cell type (three diagonal blocks correspond to the three cell lines  
 955 in [Srivatsan20](#) dataset), and even smaller between cell types. In comparison, the DecoderOnly  
 956 model with only cell type as input (panel **A)** shows its predictions are (unsurprisingly) identical  
 957 for any perturbation within a cell type, which is a prime example of mode collapse. Qualitatively,  
 958 from **A)** to **E)**, models have shown a general declining trend in mode collapse, with the simplified  
 959 SAMS-VAE\* (S) suffering the least mode collapse.

960 However, we point our readers to the RMSE metrics reported in Table 6, which are of roughly similar  
 961 order of magnitude for all models. This means judging by the RMSE alone, it is impossible to  
 962 tell which model suffers from mode collapse. On the other hand, our rank metric distinguishes the  
 963 degenerative case from the rest, but it is still not sensitive enough to discern the severity of mode  
 964 collapse.

### 965 C.3.2 Diagnostic metrics for mode collapse

966 To come up with metrics more indicative of mode collapse, we propose two other metrics which as  
 967 shown in Table 6, are more strongly correlated with the visual assessment. The first one is called

Table 7: Numerical quantification of mode collapse for models presented in [Frangieh21](#) dataset.

Model	RMSE mean	RMSE mean rank	RMSE mean transposed-rank	Matrix distance
SAMS-VAE*	0.024	0.383	0.491	81.007
SAMS-VAE* (S)	0.023	0.160	0.416	54.481
CPA*	0.024	0.236	0.484	80.434
CPA* (noAdv)	0.024	0.156	0.480	80.166
LA	0.023	0.127	0.413	65.308

transposed-rank, defined slightly different than our vanilla rank metric:

$$\text{rank}_{\text{average}}^T := \frac{1}{p} \sum_{i=1}^p \text{rank}^T(\hat{x}_i), \quad \text{rank}^T(\hat{x}_i) := \frac{1}{p-1} \sum_{\substack{1 \leq j \leq p \\ j \neq i}} \mathbb{I}(\text{dist}(\hat{x}_i, x_j) \leq \text{dist}(\hat{x}_i, x_i)), \quad (3)$$

where  $p$  is the number of perturbations that are being modelled,  $\hat{x}_i, x_i$  are the predicted and observed (average) expression value of perturbation  $i$ , and  $\text{dist}$  is a generic distance.

Compared to Eq. 2, the new transposed-rank metric ranks the observed expression on a per prediction basis, whereas in the original rank metric, it is vice-versa. We hypothesize that ranking the observations on a per-prediction basis will make the test more challenging thus exposing the weakness of a mode-collapsed model, because in this case the observed expressions have more variance than the predictions.

Finally, we propose a matrix distance based metric which directly measures the difference between the two similarity matrices:

$$\text{distance}(\hat{S}_{\text{cosine}}, S_{\text{cosine}}) = \|\hat{S}_{\text{cosine}} - S_{\text{cosine}}\|_{\text{Frobenius}} = \sqrt{\sum_{i=1}^n \sum_{j=1}^n (\hat{s}_{ij} - s_{ij})^2}$$

where  $S_{\text{cosine}}$  is from model prediction and  $\hat{S}_{\text{cosine}}$  is from the experimental data.

Of all metrics reported in Table 6, matrix distance aligns the best with the visual assessment of the heatmaps, followed by the transposed-rank. Overall, this highlights one future direction to improve our metric, which includes symmetrizing our rank metrics and/or combining with the matrix distance measure, as a more robust way to pinpoint potential mode collapse issues.

### C.3.3 Mode collapse in [Frangieh21](#) dataset

So far, we have shown that all models suffer mode collapse in the [Srivatsan20](#) dataset, but to varying extent. Now, we move on to [Frangieh21](#), and demonstrate that similar observation still holds and in particular, the values of transposed-rank and matrix distance in diagnosing mode collapse.

Results are shown in Figure C.3 and Table 7. Models included for investigation are: 1) the vanilla SAMS-VAE\*, 2) the simplified SAMS-VAE\*, 3) the vanilla CPA\*, 4) CPA\* with no adversarial components, and 5) our LatentAdditive model. Once again, we compare the model predictions to the experimental data in [Frangieh21](#) which has only one cell type and 87 unique perturbations.

The vanilla SAMS-VAE\* has indeed mode-collapsed as it is observed in section C.1. CPA\* and CPA\* (noAdv) also suffer significant mode collapse, despite RMSE rank suggests otherwise. On the other hand, RMSE transposed-rank and matrix distance metrics clearly indicate mode collapse taking place in these models, which aligns with our visual assessment in Figure C.3. This indicates the transposed-rank and matrix distance are also better suited for identifying mode collapse in the [Frangieh21](#) dataset.

Overall, based on our observations in [Srivatsan20](#) and [Frangieh21](#) datasets, we establish that our simple baseline models such as DecoderOnly with perturbation and covariate inputs and LatentAdditive, as well as the simplified SAMS-VAE\* (S), are less prone to mode collapse.

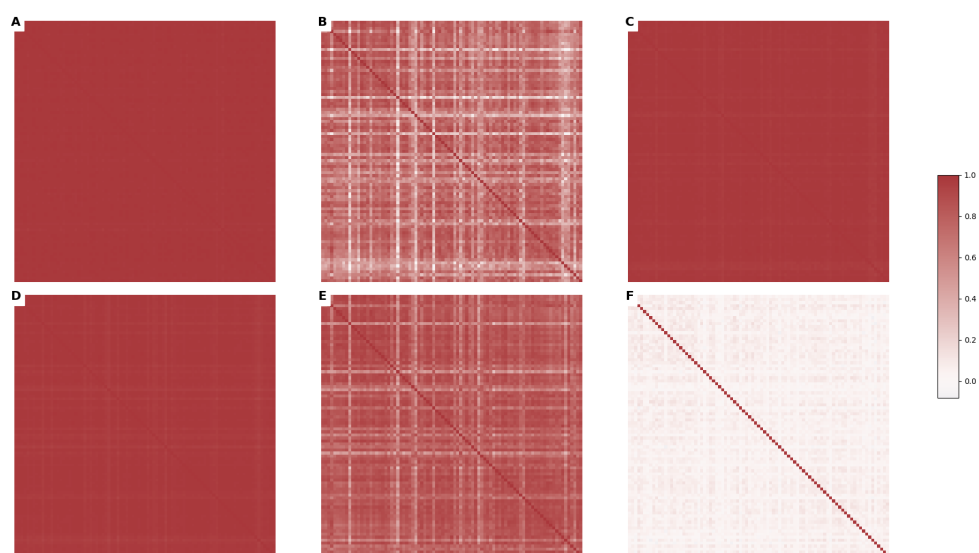


Figure C.3: Cosine similarity matrix based on log-fold changes predicted, between every pair of perturbation in [Frangieh21](#) dataset. All models are hyperparameters optimised **A**) SAMS-VAE\*. **B**) SAMS-VAE\* (S). **C**) CPA\*. **D**) CPA\* (noAdv). **E**) LatentAdditive. **F**) true log-fold changes in the dataset.

## D Implementation Details

### D.1 Dataset Summary

**Dataset 1 (Norman19)** This datasets [39] contains 287 gene overexpression perturbations with 131 containing multiple perturbations in K562 cells. We selected this dataset as it is the largest perturb-seq dataset with combinatorial perturbations so far. This dataset was also used in existing perturbation prediction studies including, e.g. CPA [35], scGPT [13], SAMS-VAE [6] and Biolord [42].

**Dataset 2 (Srivatsan20)** This dataset [51] includes 188 chemical perturbations across the K562, A549, and MCF-7 cell lines. The chemical perturbations were applied at 4 doses but for the purposes of this study, we subset to highest dose only since most of the models we are benchmarking do not have dose response modeling capacity. We selected this dataset to benchmark prediction of chemical perturbations. Additionally, this dataset was used as a benchmark in multiple perturbation prediction studies including CPA [35] and Biolord [42].

**Dataset 3 (Frangieh21)** This dataset [18] includes 248 genetic perturbations across 3 melanoma cell conditions that simulate interaction with immune cells. The conditions are: 1) melanoma cells cultured alone, 2) melanoma cells with  $IFN\gamma$ , and 3) melanoma cells co-cultured with tumor infiltrating immune cells. We selected this dataset to benchmark whether a perturbation response prediction model could predict perturbation effects in the more complex co-culture condition using training data from the simpler conditions.

**Dataset 4 (Jiang24)** This dataset [24] includes 219 genetic perturbations across 6 cell lines and 5 cytokine treatments (which can be seen as 30 unique biological states). We selected this dataset due to the large number of biological states and the fact that the perturbations were chosen because they had been reported to modulate cytokine signaling. This dataset has not been previously used to benchmark perturbation prediction models.

**Dataset 5 (McFalineFigueroa23)** This dataset [38] includes 525 genetic perturbations across 3 cell lines and 5 chemical treatments (which can be seen as 15 unique biological states). We selected this dataset due to the large number of perturbations and the fact that it contains multiple covariates (cell lines and chemical treatments). This dataset has not been previously used to benchmark perturbation prediction models.

### D.2 Dataset Curation

We download the gene expression counts matrices which are from the original sources of these datasets. Afterwards, we identify the key metadata columns (perturbation, cell line, chemical treatment) and standardize their naming and format across datasets.

### D.3 Dataset Preprocessing

It is a common practice to pre-process perturbation datasets before feeding them into a machine learning pipeline for training or inference. In this section, we describe the data processing that is used by our benchmark.

To ensure we are capturing the most biologically relevant features, we subset the expression vectors to highly variable or differentially expressed genes. Specifically, we keep the top 4000 variable genes using the scanpy `pp.highly_variable_genes` method with `flavor='seurat_v3'`. We also keep the top 25 top differentially expressed genes for every perturbation in every unique set of covariates, using scanpy's `tl.rank_genes_groups` method with default parameters. For datasets with genetic perturbations, we also ensure that the perturbed gene is included in the feature set as well.

For the models that require log-normalization, we apply the default scanpy [57] preprocessing pipeline. Specifically, we divide the counts by the total counts in each cell, multiply by a scaling factor of 10,000, and apply a log-transform with a pseudocount of 1, i.e.

$$x_{i,\text{normalized}} = \log \left( 1 + \frac{x_i}{\sum_j x_j} \cdot 10^4 \right).$$

Table 8: Number of perturbations in each cell type for downsampled subsets of [Srivatsan20](#) with different levels of data balance.

Balance	# Perts Cell Type 1	# Perts Cell Type 2	# Perts Cell Type 3
1	188	188	188
0.9	188	50	117
0.8	188	81	30
0.7	188	33	33

#### D.4 Data Splitting

**McFalineFigueroa23 splits** We manually generate the data scaling splits for the [McFalineFigueroa23](#) dataset by first selecting 3 covariates where certain perturbations will be held-out. Of the 3 cell type (a172, t98g, u87mg) and 5 treatments (control, nintedanib, zstk474, lapatinib, trametinib) in [McFalineFigueroa23](#), we have specifically selected the following 3 covariates: a172 with nintedanib, t98g with lapatinib, and u87mg with control. Within each of these "held-out covariates", we randomly hold out 70% of perturbations for validation and testing. Some perturbations may be held out across multiple covariates.

To build the small version of the dataset, we select 3 additional covariates that match the cell type and chemical treatment of the "held-out covariates" to add to the training split (a172 with control treatment, t98g with nintedanib, u87mg with lapatinib). To build the medium version of the dataset, we add the remaining 3 covariates that match cell type and chemical treatment to the training split. The large version contains the full dataset — all 15 covariates.

The attached codebase has a python notebook responsible for generating this split: `notebooks/neurips2025/build_data_scaling_splits.ipynb`.

**Jiang24 splits** We hold out 70% of perturbations in all 12 combinations of the following cytokines: IFNG, INS, TGFB and cell lines: k562, mcf7, ht29, hap1. The remaining perturbations are used for training.

The attached codebase has a python notebook responsible for generating this split: `notebooks/neurips2025/build_jiang24_frangieh21_splits.ipynb`.

**Frangieh21 splits** We hold out 70% of the perturbations in the co-culture condition and use the remaining perturbations for training.

The attached codebase has a python notebook responsible for generating this split: `notebooks/neurips2025/build_jiang24_frangieh21_splits.ipynb`.

**Srivatsan20 Data Imbalance splits** To generate the imbalanced [Srivatsan20](#) datasets for Figure C.1, we set three different desired level of imbalance, which we have quantified via normalized entropy based on the number of perturbations per cell type:

$$\text{Imbalance} := 1 - \frac{\sum_{i=1}^k \frac{n_i}{n} \log \frac{n_i}{n}}{\log k},$$

where  $n_i, i = 1, \dots, k$  denotes the number of samples in class  $i$  and  $n = \sum_{i=1}^k n_i$  the overall number of observations. The full [Srivatsan20](#) data is fully balanced with 188 perturbations seen in all three cell types. For the three subsequent imbalanced data sets, we fix the first cell type to always see all 188 perturbations, and then randomly choose the number of seen perturbations for the other two cell types that will result in the desired level of balance (distributions given in Table 8). Control cells are always seen in training for each cell type. We then randomly downsample each cell type to the desired number of perturbations, and use our datasplitter with default parameters to generate a cross cell type split. We set the minimum number of perturbations to 30 per cell type.

**Unseen perturbation splits** Some models such as scGPT, GEARS, and PerturbNet create an embedding over the perturbation space which enables prediction of the effect of perturbations that are never seen during training in any context. Since this task is very complex and most likely highly

1084 dependent on the quality of the perturbation embedding/representation, we choose not to address it  
1085 the scope of this study.

## 1086 D.5 Models

1087 **CPA** We implement a version of CPA using the published Theis lab model (forked 02/23). The  
1088 Theis lab [codebase](#) has been updated since publication, and we have incorporated the most important  
1089 changes into our implementation. However, some small differences remain, thus, we refer to our  
1090 implementation as CPA\*.

1091 To ensure that we have correctly implemented CPA, we verify that our implementation has achieved  
1092 similar or better performance on all metrics compared to the published versions. To this end, we have  
1093 trained a CPA model using the published Theis lab model (forked 02/23) and our implementation  
1094 using the exact same hyperparameters identified to be optimal by the authors, on the same data split  
1095 and assessed the performance. Our implementation of CPA has obtained comparable (and indeed,  
1096 slightly better) results than the original codebase. CPA alternates between training the generator and  
1097 discriminator, and in the ablated version: CPA\* (noAdv), we disregard the discriminator by setting  
1098 the adversarial loss to zero and training the generator exclusively.

1099 **SAMS-VAE** SAMS-VAE is available under a restrictive licence. For this reason, we imple-  
1100 ment a version of the model carefully following the authors’ description. Since, the model is  
1101 re-implementation, we refer to it as SAMS-VAE\*.

1102 To further understand the effect of sparse additive mechanism, we ablate the sparsity-inducing  
1103 component of SAMS-VAE by completely removing the binary mask and its associated global latent  
1104 variable. The other significant modification is to remove the global variable defined on the perturbation  
1105 embedding. To this end, we obtain a simplified version, i.e. SAMS-VAE\* (S), which does not contain  
1106 any global latent variables, and learns perturbation effects through ordinary embedding vectors  
1107 without any sparsity or distributional assumptions.

1108 **BioLord** For modelling the effect of perturbations, Biolord requires embeddings, either from  
1109 the GEARS GO graph for genes or RDKIT embeddings for small molecules. For the sake of fair  
1110 comparison, we have excluded the use of embeddings and therefore, implemented a slight variation of  
1111 Biolord, henceforth referred to as Biolord\*, where instead of neighbourhoods based on embeddings,  
1112 we use the same one hot representation for perturbations as for all other models.

1113 **GEARS** Because the GEARS model differs from other models in its use of GNNs to encode gene  
1114 expression values and perturbations, as well as the authors did not recommend applying it to the  
1115 *covariate transfer* task, we choose not to reimplement GEARS in our library. We instead write a  
1116 helper function and HPO script for training and evaluating GEARS using its publicly available code,  
1117 on the same [Norman19](#) split which we have used for other models.

1118 **scGPT Embeddings** To generate scGPT embeddings, we use the pretrained whole human [model](#)  
1119 and generate embeddings with no further fine-tuning on our processed datasets.

1120 **Linear** The simple linear baseline model uses the *control matching* approach. Given a perturbed  
1121 cell,  $x'$ , we sample a random control cell with *matched* covariates,  $x$ , and reconstruct  $x'$  by applying  
1122 one linear layer given the perturbation and covariates:

$$x' = x + f_{\text{linear}}(p_{\text{one\_hot}}, \text{cov}_{\text{one\_hot}}), \quad (4)$$

1123 where  $p_{\text{one\_hot}}$  denotes the one-hot encoding of the perturbation and  $\text{cov}_{\text{one\_hot}}$  denotes one-hot  
1124 encodings of covariates (i.e. cell type).

1125 **Latent Additive** We extended the linear model into a baseline latent additive model by encoding  
1126 expression values and perturbations into a latent space  $Z \subseteq \mathbb{R}^{d_z}$ , i.e.

$$z_{\text{ctrl}} = f_{\text{ctrl}}(x), \quad \text{and} \quad z_{\text{pert}} = f_{\text{pert}}(p_{\text{one\_hot}}),$$

1127 where  $p_{\text{one\_hot}}$  denotes the one-hot encoding of the perturbation. Subsequently, we reconstruct the  
1128 expression value by decoding the added latent space representation  $x' = f_{\text{dec}}(z_{\text{ctrl}} + z_{\text{pert}})$ .



**Decoder Only** As a further ablation study, we introduce a model class that aims to predict the transcriptome solely from covariates,  $cov_{one\_hot}$ , perturbation information,  $p_{one\_hot}$ , or a mix of both. This model takes as an input neither the transcriptome of a control cell nor the transcriptome of a perturbed cell. Consequently, prediction of the expression of a perturbed cell can be modelled as  $x' = f_{dec}(z)$  for  $z \in \{p_{one\_hot}\} \cup \{cov_{one\_hot}\} \cup \{(p_{one\_hot}, cov_{one\_hot})\}$  and we refer to them as *decoder only* models. This class of models provides a range of baselines:

- Firstly, a model decoding only from covariates provides a lower bound on the performance of acceptable models and a sense of what performance can be expected when a model collapses to its mode(s). For instance, if the covariates contain only the cell type, this model will only learn the average expression value for each cell type. Since no perturbation information is used, the model is completely collapsed for every class of covariates.
- Secondly, a model that decodes only from perturbations offers a baseline that illustrates the extent to which expression levels resulting from perturbations can be predicted, disregarding any information about cell type or expression levels in control cells.
- Thirdly, a model that decodes information from both cell type and perturbations provides a baseline for understanding the additional information that the transcriptome could offer, which is not already captured by the covariates or inherently present in the perturbation data.

## D.6 Hyperparameter Optimization

### D.6.1 Identifying a Hyperparameter Metric

In order to carry out HPO, we need to define a performance metric that can be taken as an objective function for `optuna`. The model loss calculated on the validation data can in many cases be unsuitable for such a task, as some hyperparameters are part of the loss itself and aim, for instance, to find a balancing factor between different loss terms. In such scenarios, the objective would induce `optuna` to simply set a scaling factor to 0. Hence, we require an alternative metric as an HPO objective function.

To define an objective functions we set out the following requirements:

- To make our models comparable and to avoid confounding issues, we compare all models based on the same metric for the purposes of HPO.
- Considering the results of Section C.3 hyperparameter optimization can not simply be carried out on one metric, such as RMSE, as we have established that this metric alone does not cover all aspects of model performance.

To identify suitable hyperparameter metrics, we carried out several HPO runs with linear combinations of cosine similarity and the respective rank metric, as well as RMSE and its respective rank metric. In a few pilot hpo runs we observed that

$$\mathcal{L}_{HPO} = \text{RMSE} + 0.1 \cdot \text{rank}_{\text{RMSE}}$$

results in models that perform well on both aspects, traditional model fit as well as ranking metrics.

### D.6.2 Hyperparameter Ranges

For hyperparameter optimization we used `optuna` [3]. Hence, we can define all hyperparameter ranges as `optuna` distributions, either in the form of `categorical`, `int` or `float`. We describe the seed and the specific `optuna` hyperparameter ranges as well as their distribution classes in Tables 9 to 13 and 15.

Table 9: CPA hyperparamter range.

Hyperparameter	Distribution
<i>Number of layers in the encoder part of the model:</i>	
n_layers_encoder	Int: 1 to 7, step=2
<i>Number of perturbation embedding layers:</i>	
n_layers_pert_emb	Int: 1 to 5, step=1
<i>Number of layers in the adversarial classifier:</i>	
adv_classifier_n_layers	Int: 1 to 5, step=1
<i>Hidden dimension size:</i>	
hidden_dim	Int: 256 to 5376, step=1024
<i>Hidden dimension size of the adversarial classifier:</i>	
adv_classifier_hidden_dim	Int: 128 to 1024, log=True
<i>Number of adversarial steps:</i>	
adv_steps	Categorical: [2, 3, 5, 7, 10, 20, 30]
<i>Number of latent variables:</i>	
n_latent	Categorical: [64, 128, 192, 256, 512]
<i>Learning rate:</i>	
lr	Float: 5e-6 to 1e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True
<i>Dropout rate:</i>	
dropout	Float: 0.0 to 0.8, step=0.1
<i>KL divergence weight:</i>	
kl_weight	Float: 0.1 to 20, log=True
<i>Adversarial weight:</i>	
adv_weight	Float: 0.1 to 20, log=True
<i>Penalty weight:</i>	
penalty_weight	Float: 0.1 to 20, log=True

Table 10: Latent additive model hyperparameter range.

Hyperparameter	Distribution
<i>Number of layers in the encoder part of the model:</i>	
n_layers	Int: 1 to 7, step=2
<i>Width of the encoder layers in the model:</i>	
encoder_width	Int: 256 to 5376, step=1024
<i>Dimensionality of the latent space:</i>	
latent_dim	Categorical: [64, 128, 192, 256, 512]
<i>Learning rate:</i>	
lr	Float: 5e-6 to 5e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True
<i>Dropout rate:</i>	
dropout	Float: 0.0 to 0.8, step=0.1



Table 11: Linear additive model hyperparameter range.

Hyperparameter	Distribution
<i>Learning rate:</i>	
lr	Float: 5e-6 to 5e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True

Table 12: Biolord hyperparameter range.

Hyperparameter	Distribution
<i>Weight of the penalty term in the loss function:</i>	
penalty_weight	Float: 1e1 to 1e5, log=True
<i>Number of layers in the encoder part of the model:</i>	
n_layers	Int: 1 to 7, step=2
<i>Width of the encoder layers in the model:</i>	
encoder_width	Int: 256 to 5376, step=1024
<i>Dimensionality of the latent space:</i>	
latent_dim	Categorical: [64, 128, 192, 256, 512]
<i>Learning rate:</i>	
lr	Float: 5e-6 to 5e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True
<i>Dropout rate:</i>	
dropout	Float: 0.0 to 0.8, step=0.1

Table 13: SamsVae hyperparameter range.

Hyperparameter	Distribution
<i>Number of layers in the encoder part of the model:</i>	
n_layers_encoder_x	Int: 1 to 7, step=2
<i>Number of layers in the encoder part of the model:</i>	
n_layers_encoder_e	Int: 1 to 7, step=2
<i>Number of layers in the decoder part of the model:</i>	
n_layers_decoder	Int: 1 to 7, step=2
<i>Width of the encoder layers in the model:</i>	
latent_dim	Categorical: [64, 128, 192, 256, 512]
<i>Hidden dimension for x:</i>	
hidden_dim_x	Int: 256 to 5376, step=1024
<i>Hidden dimension for the conditional input:</i>	
hidden_dim_cond	Int: 50 to 500, step=50
<i>Whether to use sparse additive mechanism:</i>	
sparse_additive_mechanism	Categorical: [True, False]
<i>Whether to use mean field encoding:</i>	
mean_field_encoding	Categorical: [True, False]
<i>Learning rate:</i>	
lr	Float: 5e-6 to 1e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True
<i>The target probability for the masks:</i>	
mask_prior_probability	Float: 1e-4 to 0.99, log=True
<i>Dropout rate:</i>	
dropout	Float: 0.0 to 0.8, step=0.1

Table 14: DecoderOnly hyperparameter range.

Hyperparameter	Distribution
<i>Number of layers in encoder/decoder:</i>	
n_layers	Int: 1 to 7, step=2
<i>Width of the encoder:</i>	
encoder_width	Int: 256 to 5376, step=1024
<i>Learning rate:</i>	
lr	Float: 5e-6 to 5e-3, log=True
<i>Weight decay:</i>	
wd	Float: 1e-8 to 1e-3, log=True
<i>Whether to apply a softplus activation to the output of the decoder to enforce non-negativity:</i>	
softplus_output	Categorical: [True, False]

Table 15: GEARS hyperparameter range.

Hyperparameter	Distribution
<i>Number of layers in perturbation GNN:</i> num_go_gnn_layers	Int: 1 to 3, , step=1
<i>Number of layers in gene GNN:</i> num_gene_gnn_layers	Int: 1 to 3, step=1
<i>Number of neighboring perturbations in GO graph:</i> num_similar_genes_go_graph	Int: 10 to 30, step=10
<i>Number of neighboring genes in gene co-expression graph:</i> num_similar_genes_co_express_graph	Int: 10 to 30, step=10
<i>Width of the encoder:</i> hidden_size	Int: 32 to 512, step=32
<i>Minimum coexpression threshold:</i> co_express_threshold_graph	Float: 0.2 to 0.5, step=0.1
<i>Learning rate:</i> lr	Float: 5e-6 to 5e-3, log=True
<i>Weight decay:</i> wd	Float: 1e-8 to 1e-3, log=True

## D.7 Compute Resources

For the [Norman19](#) and [Srivatsan20](#), and data imbalance tasks, we used nodes with one Nvidia A10G GPU each. We ran 60 hyperparameter optimization trials for each model, and assessed 10 models on the [Srivatsan20](#) task and 9 models on the [Norman19](#) task. We also ran 4 training runs with the best hyperparameters for stability analysis. We also ran an additional 5 models on the 4 different data imbalance splits, again with 4 runs for stability. For the HPO runs we used 813 hours for [Srivatsan20](#) and 399 hours for [Norman19](#). See details in Table 16.

For the [McFalineFigueroa23](#) data scaling task, we used nodes with Nvidia A10G GPUs for most of the combinations of models and subsets. We used A100G or H100G GPUs for all deep learning model for the biggest split, and for all datasets on CPA (which required the most GPU memory). We again used 60 hyperparameter optimization trials across 4 models with an additional 4 runs for stability. In total for this experiments we used 2517 hours of servers with GPUs, see details in Table 16.

Table 16: Total runtime of HPO for different models and datasets

dataset	model	runtime	A100
mcfaline23-full	cpa	171.97	yes
mcfaline23-full	decoder-only	136.91	yes
mcfaline23-full	latent-additive	150.36	yes
mcfaline23-full	linear-additive	321.08	
mcfaline23-medium	cpa	127.44	yes
mcfaline23-medium	decoder-only	225.24	
mcfaline23-medium	latent-additive	280.12	
mcfaline23-medium	linear-additive	359.33	
mcfaline23-small	cpa	105.12	yes
mcfaline23-small	decoder-only	135.38	
mcfaline23-small	latent-additive	186.14	
mcfaline23-small	linear-additive	317.91	
norman19	biolord	129.71	
norman19	cpa	42.98	
norman19	cpa-no-adversary	48.08	
norman19	cpa-scgpt	25.20	
norman19	decoder	20.48	
norman19	latent	32.69	
norman19	latent-scgpt	21.42	
norman19	linear	30.17	
norman19	sams	48.06	
sciplex3	biolord	312.49	
sciplex3	cpa	41.66	
sciplex3	cpa-no-adversary	51.83	
sciplex3	cpa-scgpt	38.54	
sciplex3	decoder	40.41	
sciplex3	decoder-cov	36.42	
sciplex3	latent	56.59	
sciplex3	latent-scgpt	76.25	
sciplex3	linear	70.48	
sciplex3	sams	88.76	

## D.8 Benchmarking metrics

A common approach is to report metrics that are associated with the “global” fit or the *accuracy* of the model. These metrics include RMSE and *cosine similarity*

$$S_{\text{cosine}}(x, y) = \frac{\sum_i^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}}$$

1185 between predicted and observed perturbations since the correlation metrics (whether in the flavour of  
1186 Spearman or Pearson) are invariant with respect to shifts in mean expression values.

1187 We have two different classes of *accuracy* related metrics. One class of metrics evaluates whether  
1188 predicted and observed aggregates have similar shapes (pearson, cosine). Another class evaluates  
1189 whether predicted and observed aggregates have similar values (RMSE, MAE, MSE, R2 score).