A modified multilayer perceptron for feature selection in single-cell RNA sequencing

Alexander Epstein
Ph.D. Candidate, Cao Laboratory
The Rockefeller University
aepstein@rockefeller.edu

January 15, 2024

Abstract

In single-cell RNA sequencing, the number of detected RNA molecules in each cell from each of several thousand gene features is used to separates cells by type using based on the expression level of but is limited by poor recovery of RNA sequences. Many captured RNAs come from "housekeeping genes" expressed at nearly equal levels in all cell types, and are therefore useless for classifying cells. Recovery of a subset of useful marker RNAs could be increased by targeting them for sequencing with gene-specific DNA primers, but the unsupervised method used to cluster cells is not sufficiently interpretable to select gene features for targeting. I propose to use a modified multilayer perceptron with a feature selection layer to both predict cell types from scRNA-seq data, and select a set of 100-200 gene features that are of greatest importance for the prediction. These genes could then be used in a future targeted sequencing experiment to profile brain cells with greatly increased efficiency.

1 Introduction

The DNA genome of a living organism serves as a "recipe book" for its growth and functioning. Each gene within the genome is a recipe for an RNA transcript molecule, which in turn usually codes for a protein. Proteins serve as the molecular machines and building blocks of the cell, and carry out key functions such as catalyzing chemical reactions and transporting nutrients. Every type of cell in the human body, from a skin cell on the sole of the foot to a neuron in the brain, has the same set of genes in its DNA genome. These distinct cell types are different in large measure because they produce different numbers of RNA transcripts corresponding to each gene. A neuron will make RNAs needed to produce proteins for its specific functions, such as electrochemical receptors for synaptic transmission. By contrast, a skin cell will make RNAs that code for its distinctive proteins, such as the collagen fibres that give our skin its strength and elasticity.

Single-cell RNA sequencing (scRNA-seq) is a recently developed method to sequence the RNA transcripts in a tissue sample, and match each transcript sequence to its corresponding cell and gene. The number of RNAs corresponding to each gene is treated as a feature for clustering cells. After data preprocessing and curation, a set of 2000 highly variable gene features is chosen. These features are reduced to 20-30 principal components after singular value decomposition. An unsupervised method such as Leiden clustering [Traag et al., 2019] is then used to divide cells into several clusters, each of which is thought to represent a specific cell type or state. Clusters are annotated using gene features previously shown to be characteristic of a particular cell type, and are then visualized using a non-linear dimensionality reduction method such as UMAP (Figure 1).

Preprint. Under review.

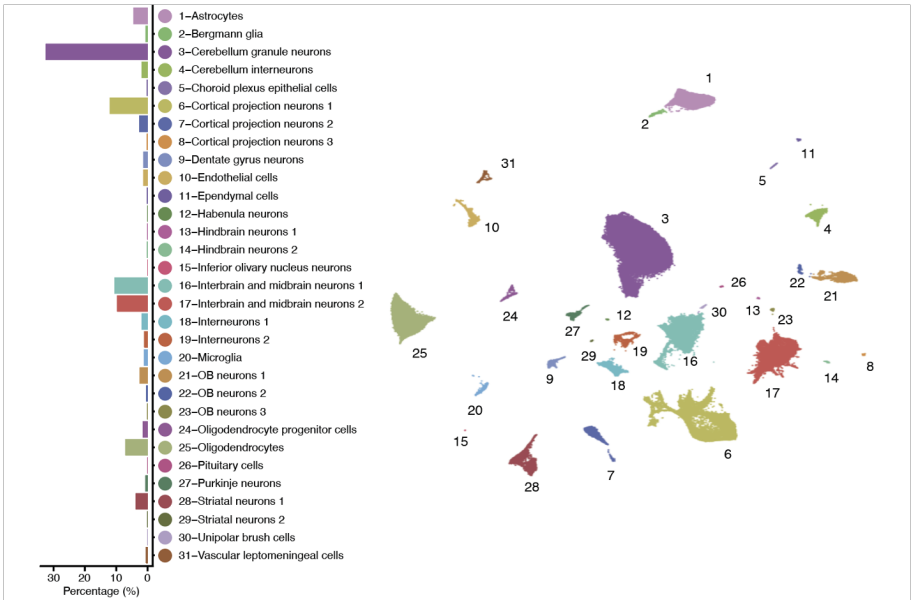

Figure 1: UMAP plot of 1.47 million mouse brain cells, colored by cell type cluster.

The throughput and quality of single-cell RNA sequencing is limited by low capture efficiency: only a small fraction of the RNAs in each cell are recovered. Many of the RNAs that are sequenced come from "housekeeping genes": features that are roughly equally abundant in all cell types, and therefore are not useful for clustering. These problems can be addressed by targeting informative genes during the sequencing process, using primers (short fragments of DNA) that match the known sequences of those genes. A substantially greater fraction of these RNAs can then be recovered from each cell, yielding a much more informative dataset for cell classification and for other downstream analysis.

In order to implement targeted sequencing successfully, it is necessary to 1) select a set of targeted genes of high classification power, and 2) use these genes to clusters cells by type. As "classificatoin power" is not a fixed metric, ideally the same method would be used both to select features and carry out predictions. However, the principal components used for unsupervised clustering are not very interpretable: they have nonzero weights for thousands of genes, and therefore feature selection poses a challenge.

One approach to address this problem is to replace principal component analysis with a dimensionality reduction method that has sparse loadings. Single-cell projective non-negative matrix factorization (scPNMF) obtains sparse loadings by requires all features to have non-negative weights in each principal component (PC), much like non-negative matrix factorization (NMF). PCs generated by PNMF often consist of small sets of closely related genes. PCs corresponding to unwanted sources of variation (e.g. RNA number, cell cycle state) are removed, and the top remaining PCs could form a gene set to be targeted [Song et al., 2021].

The other approach is to use the soft labels arising from unsupervised clustering to train a supervised learning algorithm to predict the cluster assignments using a limited subset of genes. Another recent algorithm, NS-Forest, trains a random forest model to predict cell types. To select genes/features for targeting, features are ranked by Gini index (i.e. how well they correspond to a specific cell type). [Aevermann et al., 2021]

Both approaches are designed for small single-cell RNA sequencing datasets containing at most tens of thousands of cells. scRNA-seq dataset generated with new combinatorial indexing-based methods are much larger, with hundreds of thousands to hundreds of millions of cells [Cao et al., 2019]. These approaches are too computationally inefficient to be run on such a large dataset, requiring operations such as densifying the sparse gene count matrix.

Barraza *et al*. recently proposed a deep neural network (DNN) architecture with embedded feature selection. Between the input layer and first hidden layer of a multilayer perceptron (MLP), they

add a "feature selection layer" which encourages the MLP to rely on a small subset of features for classification. I propose to select gene features in a single-cell RNA sequencing dataset using a modified form of this architecture, which I will call SelectorMLP. I conduct supervised training of SelectorMLP using the the soft labels previously assigned to each cell through unsupervised clustering. I can then rank gene features by importance and thereby select a subset of gene features which could theoretically be used for targeted single-cell RNA sequencing. As a baseline, I compare the results to those obtained with standard MLP and with a linear support vector machine (SVM).

2 Network layout

Like the network employed by Barraza *et al*, the SelectorMLP network is a modified MLP with a feature selector layer (the "selector") inserted between the input layer and first linear hidden layer. The selector is densely connected to the following hidden layer, but each selector neuron receives input only from its corresponding feature in the input layer (Figure 1). This is equivalent to an element-wise multiplication of the input data $\mathbf{X}$ by the weight vector $\mathbf{w}$ of the selector:

$$\text{Selector}(X_{ij}) = w_j X_{ij} \tag{1}$$

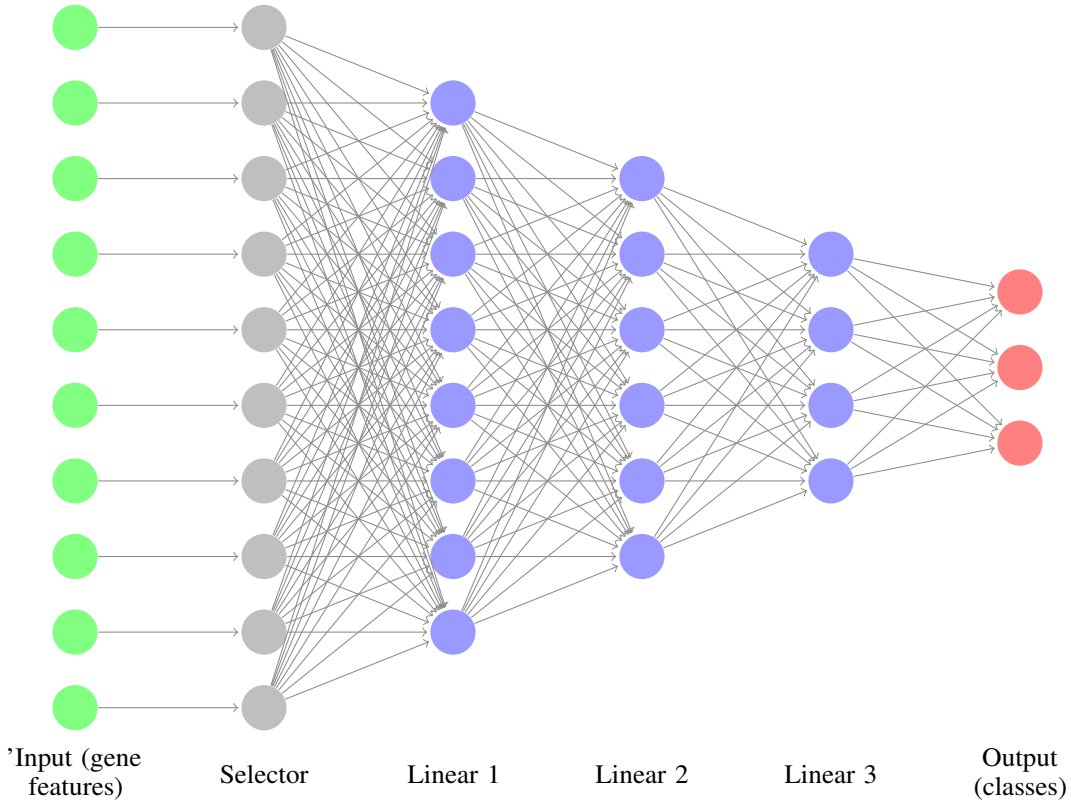

Figure 2: SelectorMLP example network layout. Each input feature has a corresponding selector node: the weight of the selector node regulates the input strength of the corresponding feature.

Each component $w^{(j}$ of the selector's weight vector $\mathbf{w}_{\text{Selector}}$ thereby regulates the input strength of the corresponding input feature $\mathbf{X}^{(j)}$. Selector weights can be made sparse during training using a regularization term, encouraging the model to rely on as few features as possible for supervised classification. In the original model proposed by Barraza *et al*, selector weights were clamped between 0 and 1, as a weight of 1 leaves the corresponding input feature unaltered, while a weight of 0 removes it entirely. In SelectorMLP, selector weights are not clamped, and the importance of each feature is determined by the absolute value of its corresponding weight.

Following the selector layer, the model is a fairly standard MLP, with a dropout layer and a leaky rectified linear unit (ReLU) activation function. The hidden layers gradually decrease in size (512, 128, 64); the output layer then has one node for each class in the dataset.

3 Preprocessing and Methods

All experiments were run on a sample dataset of 1.47 million mouse brain cells (Figure 1), which were profiled in the Cao Lab using a scRNA-seq method based on the principle of combinatorial indexing [Cao et al., 2019]. This dataset was curated for quality, and a set of 2000 genes with high variance and abundance was chosen for clustering using the Seurat R package; the remainder were discarded [Satija et al., 2015]. Unsupervised clustering was performed in Seurat, separating cells into 40 classes. The cell type corresponding to each class was identified using the abundance of known marker genes.

The data was then loaded into Python using Scanpy [Wolf et al., 2018]. Due to computational constrains, a subset of the data was used for training, validation, and testing. To minimize class imbalance, cells from each class were divided 5:1:1 between the training, validation, and test sets, up to a maximum of 5000 cells in the training set and 1000 cells in the validation and test sets. The resulting training set had 154401 cells, and the validation and test sets had 30875 each.

Deep network models were constructed in PyTorch [Paszke et al., 2017] and trained using the PyTorch Lightning wrapper (https://github.com/PyTorchLightning/pytorch-lightning) in Google Colab using a NVIDIA Tesla T4 GPU. Linear SVMs were imported from scikit-learn [Pedregosa et al., 2011]. See Appendices 1 and 2 for details of model parameters and Python code.

4 Results

4.1 Optimizing MLP cell classification

The SelectorMLP architecture depends on an underlying multilayer perceptron (MLP). This MLP was intiially trained with only weight decay for regularization, and displayed substantial overfitting: the training loss continued decreasing over 50 epochs, while the validation loss quickly stabilized and then began to increase. Adding random noise and dropout during training reduced overfitting, decreasing the training loss and accuracy but increasing the validation loss and accuracy (Figure 3). However, some degree of overfitting still remained. The final validation accuracy after regularization was 96.8%.

4.2 Effect of the selector layer on classification accuracy

The optimized MLP with noise and dropout was used as a base to construct a SelectorMLP model. I tested the effect of two parameters on the performance of the SelectorMLP:

- The regularization function used for the selector's weights ($\mathrm{R}(\mathbf{w})$), which is added to the training loss at each epoch.
- The use of a clamp $0 \leq w_j \leq 1$ on the selector weights, as in Barraza *et al*.

Three different regularization functions were tested:

- No regularization ($\mathrm{R}(\mathbf{w}) = \mathbf{0}$). This should in theory be equivalent to a standard MLP (except with additional degrees of freedom) as there is no nonlinearity between the selector layer and the first hidden layer H_1.
- L1 regularization, computed as follows:

$$\mathrm{R}(\mathbf{w}) = \sum_j |w_j| \tag{2}$$

- L2 regularization:

$$\mathrm{R}(\mathbf{w}) = \sqrt{\sum_j w_j^2} \tag{3}$$

Figure 3: MLP training and validation loss (left) and accuracy (right) with and without the addition of noise and dropout during the training step. Both training and validation loss are calculated in evaluation mode (without noise and dropout) after each training epoch.

The goal of adding the selector and of regularizing its weights was to encourage the classifier to use a small number of features. Therefore, I did not expect these changes to benefit the network's classification accuracy on the full set of 2000 features. As expected, the MLP had the highest validation accuracy of all of the models (Figure 5). Other models had slightly lower validation accuracy, especially the selector with L1 regularization, which experienced an unexpected decrease in validation accuracy between epochs 0 and 10 (which was observed consistently across multiple runs). Clamping the selector weights between 0 and 1 seemed to help the validation accuracy slightly.

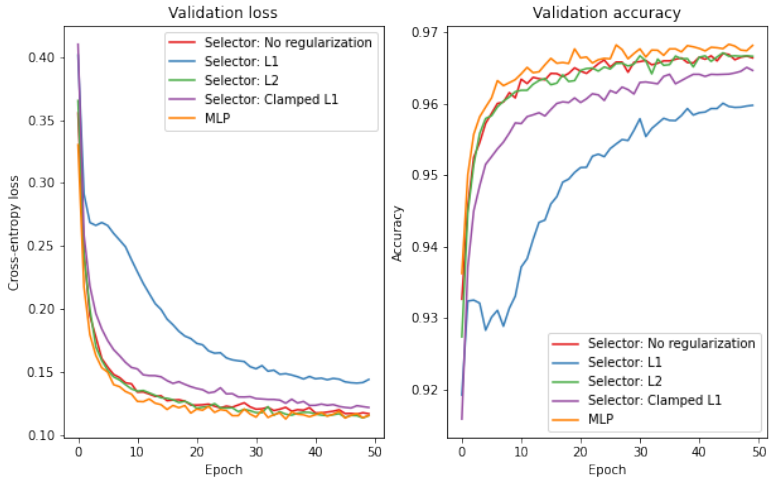

Figure 4: Validation loss (left) and accuracy (right) during training for the base MLP and several different SelectorMLP models

.

4.3 Effect of the selector layer when features are removed

Barraza *et al* tested the robustness of their selector-based model by masking features from the input data a few at a time. I applied the feature masking procedure to the trained SelectorMLP models. I ordered the features of each model by importance using the absolute values of their corresponding selector weights. I then masked the features fifty at a time in order of importance, beginning with the least important features (those whose selector weights had the smallest magnitudes). For each mask, I measured each model's classification accuracy on the validation set.

For comparison, I also performed feature masking on the standard trained MLP, and on two trained linear support vector machines: one with L1 weight regularization, and one using L2 regularization. For these models, there was no obvious way to rank features by importance. I chose to sum the absolute values of all of the weights corresponding to each input feature.

All of the models, including the linear SVMs, had similar classification accuracy on the full validation set. However, the MLP and SelectorMLPs performed much better than the linear SVMs when some features were masked (Figure 5; upper panel). Among the deep network-based models, the SelectorMLP with no weight regularization performed almost identically to the MLP, which was expected since they are essentially equivalent. The SelectorMLPs with weight regularization performed better: L1 regularization, which is known to promote sparse weights, proved optimal. Though clamping slightly improved the validation accuracy before feature masking, it substantially decreased validation accuracy when sufficient features were masked.

To determine whether the model weights accurately represented feature importance, I also tried removing features in random order for the L1-regularized SelectorMLP (Figure 5; upper panel). The resulting performance was even worse than that of the linear SVMs. As such, the feature ranking method for the MLP and SelectorMLP appears to accurately represent feature importance. However, it seems likely that the feature ranking procedure did not work well for the linear SVMs (though it is also possible that those models depended almost equally on each feature).

I obtained the 200 most important features for the best model in each class: the L1-regularized SelectorMLP, the MLP, and the L1-regularized SVM. Examining these, I found that 114 were shared between the MLP and L1-regularized SelectorMLP. By contrast, only 24 were shared between the L1-regularized SelectorMLP and the L1-regularized SVM. This is in line with the poor performance of the SVM in feature selection.

Once these 200 features were chosen, I created a cropped training, validation, and test datasets containing only these features for each model. I retrained each model on the features it had selected, re-ranked those features based on the new model weights, and plotted the classification accuracy on the validation set as a function of the number of masked features (Figure 5; lower panel). Before re-training, the MLP was less accurate using a set of 200 features than the SelectorMLP, but after re-training, this difference became negligible.

4.4 Model performance on the test set

IThe mean per-class accuracy on the test dataset of the L1-regularized SelectorMLP, the L1-regularized SVM, and the MLP was assessed under three conditions: 1) after training on the 2000-feature dataset, 2) after removing all but the 100 or 200 most important features, and 3) after re-training on these 100 or 200 features (Figure 6).

The results for 2000 or 200 features were in line with those observed on the validation set. All three models performed well when trained and tested on 2000 features. When trained on 2000 but tested on 200, the L1-regularized SelectorMLP continued to perform well, but the mean per-class accuracy of the MLP and SVM fell substantially. When each model was retrained on the top 200 features, the MLP regained its high classification accuracy, and performed similarly to the SelectorMLP; the SVM, however, fell short.

When trained with 100 features, however, some distinction between the SelectorMLP and the MLP persisted even after retraining. The SelectorMLP attained an accuracy of 93.1% after retraining, while the standard MLP attained only 89.6%. As such, it seems the SelectorMLP does in fact pick more relevant features than the standard MLP, though the difference is only apparent when a small number of final features is selected.

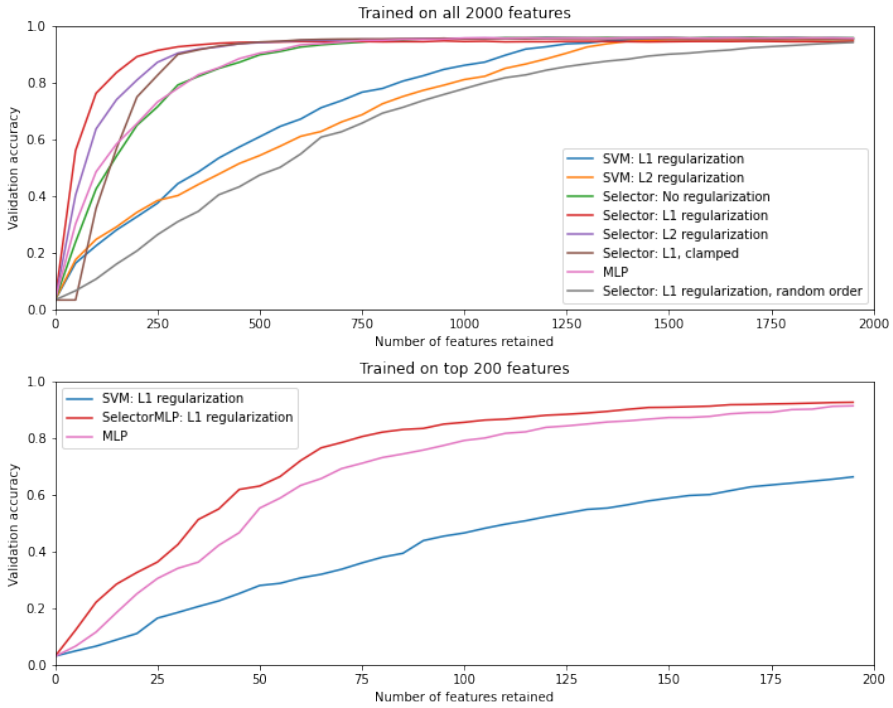

Figure 5: Top: validation accuracy as a function of the number of unmasked input features for SelectorMLP, MLP, and linear SVM models after training on the full dataset. Features were removed in order of ascending importance, except for the random case, for which feature masking was performed for five randomly chosen feature permutations and the results averaged. Bottom: validation accuracy as a function of number of features after re-training on the most important 200 features chosen by each model

.

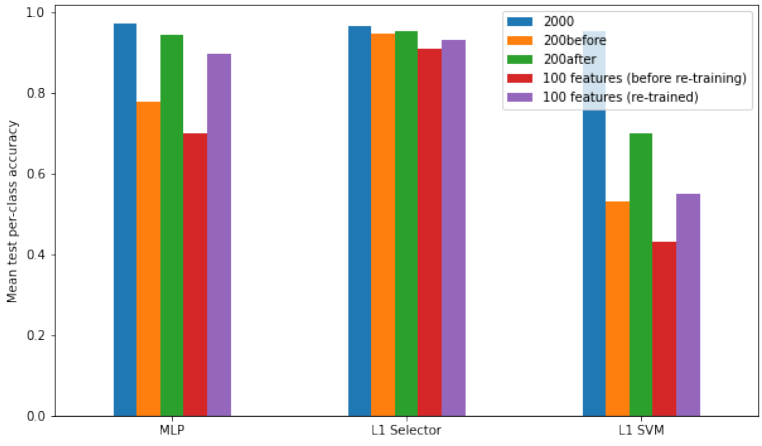

Figure 6: Mean per-class accuracy on the test dataset of three models under three conditions: 1) trained and tested on 2000 features, 2) trained on 2000 features and tested on 100 or 200, 3) retrained on 100 or 200 features and tested on 100 or 200

.

5 Conclusion and future work

The SelectorMLP model has the potential to be a useful tool for gene feature selection and cell type classification in single-cell RNA sequencing: it scales well to large datasets, requires little pre-processing, and outperforms a standard MLP and linear SVM in feature selection. One potential issue that remains to be addressed is whether a SelectorMLP model trained on a standard scRNA-seq dataset would be able to predict cell types in a targeted dataset, which would have much higher recovery rates for each of the targeted genes and a fundamentally different data structure. In the future, I hope to simulate targeted sequencing by imputing missing data in a standard scRNA-seq dataset, and determine whether the trained SelectorMLP model is robust to this imputation.

6 About Me

I am a Ph.D. student in the Cao Lab at Rockefeller University. The goal of my Ph.D. project is to develop new single-cell RNA sequencing methods targeting specific genes of interest. For some time, I have been interested in the problem of gene selection for targeted sequencing, and hope to try new supervised and unsupervised approaches to the problem over the coming year.

References

V. A. Traag, L. Waltman, and N. J. van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, Mar 2019. ISSN 2045-2322. doi: 10.1038/s41598-019-41695-z. URL https://doi.org/10.1038/s41598-019-41695-z.

Dongyuan Song, Kexin Li, Zachary Hemminger, Roy Wollman, and Jingyi Jessica Li. scPNMF: sparse gene encoding of single cells to facilitate gene selection for targeted gene profiling. *Bioinformatics*, 37(Supplement$_1$) : $i358 - -i366, 072021. ISSN1367 - 4803. doi : .$ URL https://doi.org/10.1093/bioinformatics/btab273.

B. Aevermann, Y. Zhang, M. Novotny, M. Keshk, T. Bakken, J. Miller, R. Hodge, B. Lelieveldt, E. Lein, and R. H. Scheuermann. A machine learning method for the discovery of minimum marker gene combinations for cell type identification from single-cell RNA sequencing. *Genome Res*, 31 (10):1767–1780, Oct 2021.

Junyue Cao, Malte Spielmann, Xiaojie Qiu, Xingfan Huang, Daniel M. Ibrahim, Andrew J. Hill, Fan Zhang, Stefan Mundlos, Lena Christiansen, Frank J. Steemers, Cole Trapnell, and Jay Shendure. The single-cell transcriptional landscape of mammalian organogenesis. *Nature*, 566(7745):496–502, Feb 2019. ISSN 1476-4687. 10.1038/s41586-019-0969-x. URL https://doi.org/10.1038/s41586-019-0969-x.

Rahul Satija, Jeffrey A Farrell, David Gennert, Alexander F Schier, and Aviv Regev. Spatial reconstruction of single-cell gene expression data. *Nature Biotechnology*, 33:495–502, 2015. 10.1038/nbt.3192. URL https://doi.org/10.1038/nbt.3192.

F. Alexander Wolf, Philipp Angerer, and Fabian J. Theis. Scanpy: large-scale single-cell gene expression data analysis. *Genome Biology*, 19(1):15, Feb 2018. ISSN 1474-760X. 10.1186/s13059-017-1382-0. URL https://doi.org/10.1186/s13059-017-1382-0.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

7 Appendix 1: Model parameters

The following parameters were used for deep learning models:

- Hidden layer 1 size: 512
- Hidden layer 2 size: 256
- Hidden layer 3 size: 64
- Output later size: 40 (as there are 40 classes)
- Learning rate: 0.0003
- Minibatch size: 256
- Dropout (when present): 0.3
- Weight decay: 0.3
- Noise standard deviation (when present): 0.2
- Regularization term coefficient (for all regularized selectors): 0.1

8 Appendix 2: Python code

8.1 Loading data

```
def createDataset(adata, y_column, sparse=True, sparse_type='CSR'):
  if sparse:
    if sparse_type == 'CSR':
      X = scipy.sparse.csr_matrix(adata.X)
      X = torch.sparse_csr_tensor(X.indptr, X.indices, X.data, X.shape
                                  , requires_grad=True)
    elif sparse_type == 'COO':
      X = scipy.sparse.coo_matrix(adata.X)
      indices = np.vstack((X.row, X.col))
      X = torch.sparse_coo_tensor(indices, X.data, X.shape,
                                  requires_grad=True)
    else:
      raise ValueError("Valid sparse_type is 'CSR' or 'COO'.")
  else:
    X = torch.from_numpy(adata.X.toarray())
  y_col = adata.obs[y_column]
  if y_col.dtype.name == 'category': # categorical data, e.g. clusters
    y = torch.from_numpy(y_col.cat.codes.values).long()
  else:
    y = torch.from_numpy(y_col.values).long()
  return TensorDataset(X, y)

 # Balancing classes
def evenClusters(adata, col, max_train=5000, max_val=1000, max_test=
                                    1000):
  train_cells = []
  val_cells = []
  test_cells = []
  max_total_cells = max_train + max_val + max_test
  for cluster, num in adata_full.obs[col].value_counts().iteritems():
    cluster_cells = adata.obs.index[adata.obs[col]==cluster].values
    num_cells = cluster_cells.shape[0]
    cell_multiplier = min(1, num_cells/max_total_cells)
    train = int(max_train*cell_multiplier)
    val = int(max_val*cell_multiplier)
    test = int(max_test*cell_multiplier)

    train_cells.append(cluster_cells[:train])
    val_cells.append(cluster_cells[train:train+val])
```

```
    test_cells.append(cluster_cells[train+val:train+val+test])

  adata_train = adata[np.concatenate(train_cells), :]
  adata_val = adata[np.concatenate(val_cells), :]
  adata_test = adata[np.concatenate(test_cells), :]

  return adata_train, adata_val, adata_test


adata_full = sc.read_h5ad("data/adata_variable.h5ad")

col = 'Main_cluster_name'
sparse=False
adata_train, adata_val, adata_test = evenClusters(adata_full, col)
train_set = createDataset(adata_train, col, sparse)
val_set = createDataset(adata_val, col, sparse)
test_set = createDataset(adata_test, col, sparse)
```

8.2 Defining and training deep neural networks

MLP and SelectorMLP models:

```
class Selector(nn.Module):
    def __init__(self, features, std=1, device=None, dtype=None):
        super().__init__()
        self.features = features
        self.weight = nn.Parameter(torch.empty(features))
        self.register_parameter('bias', None) # Not sure if this is
                                              necessary yet
        self.reset_parameters(std)

    def reset_parameters(self, std):
        if std is None:
            std=np.sqrt(self.features)
        nn.init.normal_(self.weight, mean=0, std=std)

    def forward(self, x):
        return torch.mul(x, self.weight) # Element-wise multiplication

class ClampedSelector(nn.Module):
    def __init__(self, features: int, std=1, device=None, dtype=None):

        #factory_kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.features = features
        #self.weight = nn.Parameter(torch.empty(features), **
                                              factory_kwargs)
        self.weight = nn.Parameter(torch.empty(features))
        self.register_parameter('bias', None) # Not sure if this is
                                              necessary yet
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.normal_(self.weight, mean=0, std=0.2)
        self.weight.data = torch.abs(self.weight.data)
    def forward(self, x):
        self.weight.data = torch.clamp(self.weight.data, min=0, max=1)
                                               # Result will be between 0
                                               and 1
        return torch.mul(x, self.weight) # Element-wise multiplication

class SelectorMLP(nn.Module):
    def __init__(self, num_features, num_classes, Selector=None,
                                          batch_norm=True, reg_type='L1/
```

```
                                        L2', L1_size=512, L2_size=128,
                                        L3_size=64, leak_angle=0.2,
                                        dropout=0.3):
    super().__init__()

    if Selector is None:
      self.selector = None
    else:
      self.selector = Selector(num_features, std=1)

    self.reg_type = reg_type

    if batch_norm:
      self.batch1 = nn.BatchNorm1d(L1_size)
      self.batch2 = nn.BatchNorm1d(L2_size)
      self.batch3 = nn.BatchNorm1d(L3_size)
    else:
      self.batch1 = nn.Identity()
      self.batch2 = nn.Identity()
      self.batch3 = nn.Identity()

    self.h1 = nn.Linear(num_features, L1_size)
    self.h2 = nn.Linear(L1_size, L2_size)
    self.h3 = nn.Linear(L2_size, L3_size)

    self.out = nn.Linear(L3_size, num_classes)
    self.relu = nn.LeakyReLU(leak_angle)
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    if self.selector is not None:
      x = self.selector(x)

    x = self.dropout(self.relu(self.batch1(self.h1(x))))
    x = self.dropout(self.relu(self.batch2(self.h2(x))))
    x = self.dropout(self.relu(self.batch3(self.h3(x))))
    return self.out(x)

  def regularize(self):
    if self.selector is None or self.reg_type is None:
      return 0
    elif self.reg_type == 'L1':
      L1 = torch.abs(self.selector.weight).sum()
      return L1
    elif self.reg_type == 'L2':
      L2 = torch.sqrt((self.selector.weight**2).sum())
      return L2
    elif self.reg_type == 'L1/L2':
      L1 = torch.abs(self.selector.weight).sum()
      L2 = torch.sqrt((self.selector.weight**2).sum())
      return L1/L2
```

Lightning trainer:

```
class MultiClassifier(pl.LightningModule):

  def __init__(self, Net,  num_features, num_classes, lossFunc=nn.
                                    CrossEntropyLoss(), lr=0.001,
                                    alpha=0.01, noise_std=0,
                                    optimizer=optim.AdamW, betas=(0.9
                                    , 0.999), weight_decay=0.01, **
                                    net_args):
    super().__init__()
    self.save_hyperparameters(ignore=['Model', 'lossFunc', 'optimizer'
                                      ])
```

```
    self.net = Net(num_features, num_classes, **net_args)
    self.optimizer = optimizer
    self.lossFunc = lossFunc
    self.accFunc = torchmetrics.Accuracy()

  def forward(self, data): # Final predictions, including softmax
    return torch.softmax(self.net(data), dim=1)

  def training_step(self, batch, batch_idx):
    X, y = batch
    y_pred = self.net(X + self.hparams.noise_std*torch.randn(X.shape,
                                        device=self.device))
    reg_term = self.net.regularize()
    loss_term = self.lossFunc(y_pred, y)
    loss = loss_term + self.hparams.alpha * reg_term # Custom
                                        regularization term from net
    self.log("train_batch_loss", loss, on_step=True, on_epoch=False)
    self.log('reg_term', reg_term, on_step=True, on_epoch=False)
    self.log("train_batch_loss_term", loss_term, on_step=True,
                                        on_epoch=False)
    return loss

  # Dataloader idx 0: validation. Dataloader idx 1: training (optional
                                       )
  def validation_step(self, batch, batch_idx, dataloader_idx=0):
    X, y = batch
    y_pred = self.net(X)
    loss = self.lossFunc(y_pred, y)
    acc = self.accFunc(torch.softmax(y_pred, dim=1), y)
    if dataloader_idx == 0:
      self.log('val_eval_loss', loss, on_step=False, on_epoch=True)
      self.log('val_accuracy', acc, on_step=False, on_epoch=True)
    elif dataloader_idx == 1:
      self.log('train_eval_loss', loss, on_step=False, on_epoch=True)
      self.log('train_accuracy', acc, on_step=False, on_epoch=True)

  def configure_optimizers(self):
    return self.optimizer(self.net.parameters(), lr=self.hparams.lr,
                                        betas=self.hparams.betas,
                                        weight_decay=self.hparams.
                                        weight_decay)
```

Training model (example provided: L1 SelectorMLP. Other models followed the same procedure).

```
save_dir = 'logs_selector_conditions'
name = 'Selector_L1'
num_features = 2000
selector_models = {}
dropout=0.3
batch_norm=False
noise=0.2

train_loader = DataLoader(train_set, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    True)
train_eval_loader = DataLoader(train_set, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    False)
val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=
                                    num_workers, shuffle=False)

selector = MultiClassifier(SelectorMLP, num_features, num_classes, lr=
                                    lr, batch_norm=batch_norm,
                                    noise_std=noise, reg_type='L1',
```

```
                                    Selector=Selector, weight_decay=0.3
                                    , dropout=dropout)
logger1 = pl.loggers.TensorBoardLogger(save_dir=save_dir, name=name)
logger2 = pl.loggers.CSVLogger(save_dir=save_dir, name=name)
trainer = pl.Trainer(max_epochs=50, accelerator='gpu', logger=[logger1
                                    , logger2], default_root_dir=
                                    save_dir)
trainer.fit(selector, train_loader, [val_loader, train_eval_loader])
torch.save(selector.net, os.path.join(save_dir, 'L1_selector.pkl')
```

Training several conditions for comparison:

```
# Testing MLP conditions
save_dir = 'logs_MLP_conditions'
conditions = [{'name':'MLP_final', 'dropout':0.3,'batch_norm':True, '
                                    noise':0.2},
              {'name':'MLP_no_dropout', 'dropout':0,'batch_norm':True,
                                     'noise':0.2},
              {'name':'MLP_no_batchnorm', 'dropout':0.3,'batch_norm':
                                    False, 'noise':0.2},
              {'name':'MLP_no_noise', 'dropout':0.3,'batch_norm':True,
                                     'noise':0},
              {'name':'MLP_nothing', 'dropout':0,'batch_norm':False, '
                                    noise':0}
]

for cond in conditions:
  print("Testing condition %s" % cond['name'])
  train_loader = DataLoader(train_set, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    True)
  train_eval_loader = DataLoader(train_set, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    False)
  val_loader = DataLoader(val_set, batch_size=batch_size, num_workers=
                                    num_workers, shuffle=False)

  mlp_model = MultiClassifier(SelectorMLP, num_features, num_classes,
                                    lr=lr, batch_norm=cond['
                                    batch_norm'], noise_std=cond['
                                    noise'], Selector=None,
                                    weight_decay=0.3, dropout=cond['
                                    dropout'])
  logger1 = pl.loggers.TensorBoardLogger(save_dir=save_dir, name=cond[
                                    'name'])
  logger2 = pl.loggers.CSVLogger(save_dir=save_dir, name=cond['name'])
  trainer = pl.Trainer(max_epochs=50, accelerator='gpu', logger=[
                                    logger1, logger2],
                                    default_root_dir=save_dir)
  trainer.fit(mlp_model, train_loader, [val_loader, train_eval_loader]
                                    )
```

8.3 Plotting logged data from training

Loading logs:

```
def getLogs(folder, find_version=True):

  if find_version:
    versions = [f for f in os.listdir(folder) if f.startswith('version
                                    ')]
    versions.sort()
    folder = os.path.join(folder, versions[-1])
```

```
  print("Reading logs from %s..." % folder)

  logs = pd.read_csv(os.path.join(folder, "metrics.csv"))
  logs.columns = [col.split('/')[0] for col in logs.columns]

  logs_step = logs[logs['train_batch_loss'].notnull()].dropna(axis=1)
  logs_step.set_index('step', inplace=True)
  logs_epoch = logs[logs['val_eval_loss'].notnull()].dropna(axis=1)
  logs_epoch.set_index('epoch', inplace=True)

  return logs_step, logs_epoch

logs_step_MLP = {}
logs_epoch_MLP = {}
for cond in conditions:
  logs_step_MLP[cond['name']], logs_epoch_MLP[cond['name']] = getLogs(
                                    os.path.join("logs_MLP_conditions
                                    ", cond['name']))
```

Plotting training and validation loss/accuracy (example provided: MLP):

```
fig, axs = plt.subplots(1, 2, figsize=(10, 6))
colors = plt.cm.get_cmap('Set1').colors
conds = ['MLP_nothing', 'MLP_no_batchnorm']
names = ['Weight decay only', 'Noise and dropout']
for cond, name, color in zip(conds, names, colors):
  axs[0].plot(logs_epoch[cond]['train_eval_loss'], '--', color=color,
                                    label='%s: training loss' % name)
  axs[0].plot(logs_epoch[cond]['val_eval_loss'], color=color, label='%
                                    s: validation loss' % name)
  axs[1].plot(logs_epoch[cond]['train_accuracy'], '--', color=color,
                                    label='%s: training accuracy' %
                                    name)
  axs[1].plot(logs_epoch[cond]['val_accuracy'], color=color, label='%s
                                    : validation accuracy' % name)

for ax in axs:
  ax.legend()
  ax.set_xlabel('Epoch')
axs[0].set_ylabel('Cross-entropy loss')
axs[0].set_title('Training and validation loss')
axs[1].set_ylabel('Accuracy')
axs[1].set_title('Training and validation accuracy')
fig.savefig('figures/MLPLoss.png')
```

8.4 Training SVMs

:

```
from sklearn.svm import LinearSVC

X_train, y_train = [tensor.numpy() for tensor in train_set.tensors]
l2_model = LinearSVC(penalty='l2', loss='squared_hinge', dual=False)
l1_model = LinearSVC(penalty='l1', loss='squared_hinge', dual=False)

l1_model.fit(X_train, y_train)
l2_model.fit(X_train, y_train)
```

8.5 Ranking features and performing masking:

For DNNs:

```
def selectFeatures(feat_nums, indices, net, X, y, lossFunc=nn.
                                    CrossEntropyLoss(), accFunc=
                                    torchmetrics.Accuracy()):
  with torch.no_grad():
    losses = []
    accs = []
    filter = np.zeros(indices.shape[0])
    for num in tqdm(feat_nums):
      filter[indices[:num]] = 1
      X_filtered = torch.mul(X, torch.from_numpy(filter).float())
      y_pred = net(X_filtered)
      losses.append(lossFunc(y_pred, y))
      accs.append(accFunc(y_pred, y))

    losses = torch.stack(losses).cpu().detach().numpy()
    accs = torch.stack(accs).cpu().detach().numpy()
    return losses, accs

def testFeatureSelection(net, X, y, group_size=50, random_reps=5, **
                                    kwargs):
  with torch.no_grad():
    if net.selector is None: # For an MLP, sum up all of the weights
                                        corresponding to each feature
      weight_abs = np.sum(np.abs(net.h1.weight.cpu().detach().numpy())
                                         , axis=0)
    else:
      weight_abs = np.abs(net.selector.weight.cpu().detach().numpy())

    feat_nums = np.arange(0, weight_abs.shape[0], group_size)

    # Get sorted weights
    idx_sorted = np.argsort(weight_abs)[::-1]
    losses_sorted, accs_sorted = selectFeatures(feat_nums, idx_sorted,
                                        net, X, y, **kwargs)

    # Get random weight results
    losses_random = np.zeros(feat_nums.shape[0])
    accs_random = np.zeros(feat_nums.shape[0])

    for i in range(random_reps):
      idx_random = np.random.permutation(weight_abs.shape[0])
      losses_tmp, accs_tmp = selectFeatures(feat_nums, idx_random, net
                                         , X, y, **kwargs)
      losses_random += losses_tmp
      accs_random += accs_tmp

    losses_random /= random_reps
    accs_random /= random_reps
    return losses_sorted, accs_sorted, losses_random, accs_random,
                                        feat_nums
```

For SVMs:

```
from sklearn.metrics import log_loss
def selectFeaturesSVM(feat_nums, indices, model, X, y):
    accs = []
    filter = np.zeros(indices.shape[0])
    for num in tqdm(feat_nums):
      filter[indices[:num]] = 1
      X_filtered = np.multiply(X, filter)
      y_pred = model.predict(X_filtered)
      acc = (y_pred==y).sum() / len(y)
      accs.append(acc)

    accs = np.array(accs, dtype=float)
```

```
    return accs

def testFeatureSelectionSVM(model, X, y, group_size=50, random_reps=5,
                                    **kwargs):

    if type(X) == torch.Tensor:
      X = X.cpu().detach().numpy()
    if type(y) == torch.Tensor:
      y = y.cpu().detach().numpy()

    weight_abs = np.abs(model.coef_).sum(axis=0) # hope the sum makes
                                        sense
    feat_nums = np.arange(0, weight_abs.shape[0], group_size)

    # Get sorted weights
    idx_sorted = np.argsort(weight_abs)[::-1]
    accs_sorted = selectFeaturesSVM(feat_nums, idx_sorted, model, X, y
                                        , **kwargs)

    # Get random weight results
    accs_list = []
    accs_random = np.zeros(feat_nums.shape[0])
    for i in range(random_reps):
      idx_random = np.random.permutation(weight_abs.shape[0])
      accs_tmp = selectFeaturesSVM(feat_nums, idx_random, model, X, y,
                                          **kwargs)
      accs_random += accs_tmp

    accs_random /= random_reps
    return accs_sorted, accs_random, feat_nums

SVM_models = {}
with open("SVMs/l1_model.pkl", 'rb') as f:
  SVM_models['SVM_L1'] = pickle.load(f)
with open("SVMs/l2_model.pkl", 'rb') as f:
  SVM_models['SVM_L2'] = pickle.load(f)
MLP_models = {}
MLP_models['Selector_None'] = torch.load("logs_selector_conditions/
                                    None.pkl")
MLP_models['Selector_L1'] = torch.load("logs_selector_conditions/L1.
                                    pkl")
MLP_models['Selector_L2'] = torch.load("logs_selector_conditions/L2.
                                    pkl")
MLP_models['Selector_Clamped_L1'] = torch.load("
                                    logs_selector_conditions/
                                    selector_clamped.pkl")
MLP_models['MLP'] = torch.load('logs_MLP_conditions/MLP_best.pkl')

losses_sorted = {}
accs_sorted = {}
losses_random = {}
accs_random = {}
for name, model in SVM_models.items():
  accs_sorted[name], accs_random[name], feat_nums =
                                    testFeatureSelectionSVM(model,
                                    X_val, y_val)
for name, model in MLP_models.items():
  losses_sorted[name], accs_sorted[name], losses_random[name],
                                    accs_random[name], feat_nums =
                                    testFeatureSelection(model, X_val
                                    , y_val)
```

Creating plot:

```
fig, ax = plt.subplots(figsize=(10, 6))
```

```
labels = ['SVM: L1 regularization', 'SVM: L2 regularization', '
                                    Selector: No regularization', '
                                    Selector: L1 regularization', '
                                    Selector: L2 regularization', '
                                    Selector: L1, clamped', 'MLP']
for key, label in zip(accs_sorted.keys(), labels):
  ax.plot(feat_nums, accs_sorted[key], label=label)
ax.plot(feat_nums, accs_random['Selector_L1'], label='Selector: L1
                                    regularization, random order')
ax.legend()
ax.set_xlabel('Number of features retained')
ax.set_ylabel('Validation accuracy')
fig.savefig('figures/feature_selection_with_clamped.png')
```

8.6 Retraining models on 200 features

Identifying top 200 features for each model, and creating cropped model-specific datasets

```
def chooseFeatures(features, *datasets):
  out_list = []
  for dataset in datasets:
    X, y = dataset.tensors
    out_list.append(TensorDataset(X[:, features.copy()], y))
  return out_list

num_features=200
L1_selector_weights = torch.abs(L1_Selector.selector.weight).cpu().
                                    detach().numpy()
L1_selector_top_features = np.argsort(L1_selector_weights)[::-1][:
                                    num_features]
train_set_selector, val_set_selector, test_set_selector =
                                    chooseFeatures(
                                    L1_selector_top_features, train_set
                                    , test_set, val_set)

L1_SVM_weights = np.abs(L1_SVM.coef_).sum(axis=0)
L1_SVM_top_features = np.argsort(L1_SVM_weights)[::-1][:num_features]
train_set_SVM, val_set_SVM, test_set_SVM = chooseFeatures(
                                    L1_SVM_top_features, train_set,
                                    test_set, val_set)

MLP_weights = torch.abs(MLP.h1.weight).sum(axis=0).cpu().detach().
                                    numpy()
MLP_top_features = np.argsort(MLP_weights)[::-1][:num_features]
train_set_MLP, val_set_MLP, test_set_MLP = chooseFeatures(
                                    MLP_top_features, train_set,
                                    test_set, val_set)
```

Retraining models with 200 features:

```
# Retraining SVM
from sklearn.svm import LinearSVC
L1_SVM200 = LinearSVC(penalty='l1', dual=False)
X_train, y_train = train_set_SVM.tensors
L1_SVM200.fit(X_train.cpu().detach().numpy(), y_train.cpu().detach().
                                    numpy())

# Retraining selector with 200 features
save_dir = 'logs_selector_conditions'
name='L1_200'
num_features = train_set_selector.tensors[0].shape[1]
dropout=0.3
batch_norm=False
noise=0.2
```

```
train_loader = DataLoader(train_set_selector, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    True)
#train_eval_loader = DataLoader(train_set_selector, batch_size=
                                    batch_size, num_workers=num_workers
                                    , shuffle=False)
val_loader = DataLoader(val_set_selector, batch_size=batch_size,
                                    num_workers=num_workers, shuffle=
                                    False)

L1_Selector200 = MultiClassifier(SelectorMLP, num_features,
                                    num_classes, lr=lr, batch_norm=
                                    batch_norm, noise_std=noise,
                                    reg_type='L1', Selector=Selector,
                                    weight_decay=0.3, dropout=dropout)
logger1 = pl.loggers.TensorBoardLogger(save_dir=save_dir, name=name)
logger2 = pl.loggers.CSVLogger(save_dir=save_dir, name=name)
trainer = pl.Trainer(max_epochs=50, accelerator='gpu', logger=[logger1
                                    , logger2], default_root_dir=
                                    save_dir)
trainer.fit(L1_Selector200, train_loader, val_loader)
torch.save(L1_Selector200.net, os.path.join(save_dir,"L1_200.pkl"))
```

Making final accuracy plot:

```
accs_sorted_SVM200, _, feat_nums = testFeatureSelectionSVM(L1_SVM200,
                                    *val_set_SVM.tensors, group_size=5,
                                     random_reps=0)
_, accs_sorted_Selector200, _, _, feat_nums = testFeatureSelection(
                                    L1_Selector200.net, *
                                    val_set_selector.tensors,
                                    group_size=5, random_reps=0)
_, accs_sorted_MLP200, _, _, feat_nums = testFeatureSelection(MLP200.
                                    net, *val_set_MLP.tensors,
                                    group_size=5, random_reps=0)

feat_nums_2000 = np.arange(0, 2000, 50)
feat_nums_200 = np.arange(0, 200, 5)
fig, axs = plt.subplots(2, 1, figsize=(10, 8))
labels = ['SVM: L1 regularization', 'SVM: L2 regularization', '
                                    Selector: No regularization', '
                                    Selector: L1 regularization', '
                                    Selector: L2 regularization', '
                                    Selector: L1, clamped', 'MLP']
for key, label in zip(accs_sorted.keys(), labels):
  axs[0].plot(feat_nums_2000, accs_sorted[key], label=label)
axs[0].plot(feat_nums_2000, accs_random['Selector_L1'], label='
                                    Selector: L1 regularization, random
                                     order')
axs[0].legend()
axs[0].set_xlabel('Number of features retained')
axs[0].set_ylabel('Validation accuracy')
axs[0].set_title('Trained on all 2000 features')
axs[0].set_ylim(0, 1)
axs[0].set_xlim(0, 2000)
axs[1].plot(feat_nums_200, accs_sorted_SVM200, color='tab:blue', label
                                    ='SVM: L1 regularization')
axs[1].plot(feat_nums_200, accs_sorted_Selector200, color='tab:red',
                                    label='SelectorMLP: L1
                                    regularization')
axs[1].plot(feat_nums_200, accs_sorted_MLP200, color='tab:pink', label
                                    ='MLP')
axs[1].set_xlabel('Number of features retained')
axs[1].set_ylabel('Validation accuracy')
```

```
axs[1].set_title('Trained on top 200 features')
axs[1].legend()
axs[1].set_ylim(0, 1)
axs[1].set_xlim(0, 200)
fig.tight_layout()
fig.savefig('figures/feature_selection_2000_200.png')
```

8.7 Calculating test per-class accuracy

```
from sklearn.metrics import confusion_matrix
import sklearn

L1_selector_weights = torch.abs(L1_Selector.selector.weight).cpu().
                                    detach().numpy()
L1_selector_top_features = np.argsort(L1_selector_weights)[::-1][:
                                    num_features]
train_set_selector, val_set_selector, test_set_selector =
                                    chooseFeatures(
                                    L1_selector_top_features, train_set
                                    , test_set, val_set)

L1_SVM_weights = np.abs(L1_SVM.coef_).sum(axis=0)
L1_SVM_top_features = np.argsort(L1_SVM_weights)[::-1][:num_features]
train_set_SVM, val_set_SVM, test_set_SVM = chooseFeatures(
                                    L1_SVM_top_features, train_set,
                                    test_set, val_set)

MLP_weights = torch.abs(MLP.h1.weight).sum(axis=0).cpu().detach().
                                    numpy()
MLP_top_features = np.argsort(MLP_weights)[::-1][:num_features]
train_set_MLP, val_set_MLP, test_set_MLP = chooseFeatures(
                                    MLP_top_features, train_set,
                                    test_set, val_set)

# Calculating final test accuracies (per-class) for several models
# MLP (2000 features), L1 selector (2000 features), L1 selector (200
                                    features), L1 SVM (2000 features),
                                    L1 SVM (200 features)
def getTestAccuracy(model, dataset, features=None):
  X, y = dataset.tensors
  if features is not None:
    filter = torch.zeros(X.shape[1])
    filter[features.copy()] = 1
    X = torch.mul(X, filter)

  y = y.cpu().detach().numpy()

  if type(model) == sklearn.svm._classes.LinearSVC:
    X = X.cpu().detach().numpy()
    y_pred = model.predict(X)
  else:
    with torch.no_grad():
      model.eval()
      y_pred = torch.softmax(model(X), dim=1).cpu().detach().numpy().
                                        argmax(axis=1)
  accuracy = (y==y_pred).sum()/y.shape[0]
  per_class_accuracy = confusion_matrix(y_pred, y, normalize='true').
                                      diagonal()
  return accuracy, per_class_accuracy.mean()

with open("SVMs/l1_model.pkl", "rb") as f:
  L1_SVM = pickle.load(f)
L1_Selector = torch.load("logs_selector_conditions/L1.pkl")
```

```
MLP = torch.load('logs_MLP_conditions/MLP_best.pkl')

model_types = ['MLP','L1_Selector','L1_SVM']
all_models = {}
all_models['MLP'] = {'model_2000':MLP,'model_200':MLP200,'dataset_200'
                                    :test_set_MLP,'features_200':
                                    MLP_top_features}
all_models['L1_Selector'] = {'model_2000':L1_Selector,'model_200':
                                    L1_Selector200,'dataset_200':
                                    test_set_selector,'features_200':
                                    L1_selector_top_features}
all_models['L1_SVM'] = {'model_2000':L1_SVM,'model_200':L1_SVM200,'
                                    dataset_200':test_set_SVM,'
                                    features_200':L1_SVM_top_features}

mean_per_class_accuracies = pd.DataFrame(index=['2000 features','200
                                    features (before re-training)', '
                                    200 features (re-trained)'])
for model, info in all_models.items():
  _, acc_2000 = getTestAccuracy(info['model_2000'], test_set)
  _, acc_200before = getTestAccuracy(info['model_2000'], test_set,
                                      features=info['features_200'])
  _, acc_200after = getTestAccuracy(info['model_200'], info['
                                      dataset_200'])
  mean_per_class_accuracies[model] = [acc_2000, acc_200before,
                                      acc_200after]
```

Making plot:

```
fig, ax = plt.subplots(figsize=(10, 6))
mean_per_class_accuracies.T.plot.bar(ax=ax)
labels = ['MLP', 'L1 Selector', 'L1 SVM']
ax.set_xticklabels(labels, rotation=0)
ax.set_ylabel('Mean test per-class accuracy')
fig.savefig('figures/test_accuracy.png')
```