UNIVERSITY OF TORONTO

FCBII, Assignment 2

Date:	01/06/2025
Course:	MMG1344H
Name:	Maria Eleni Fafouti
Student number:	1010799094

Part 1: Dimensionality Reduction

0. Data Preparation

I decided to use the scanpy package in Python, to analyse the scRNA-seq data. After reading in the matrix and cell types text files, I ensured that the columns in expression data match the row order in the cell labels data.

Then, I transposed the expression matrix, such that the cells (observations) are the rows and the genes (variables) are the columns. This allowed me to create an Anndata object. To include the cell type information, I added the labels dataframe to the observations slot in the object, such that each cell has its corresponding label.

Before running PCA, I normalized the total counts per cell (i.e each column in the expression matrix), such that cells are comparable with each other. I also log-transformed the data, which allows comparison of genes that have really high expression values with ones that are lowly expressed.

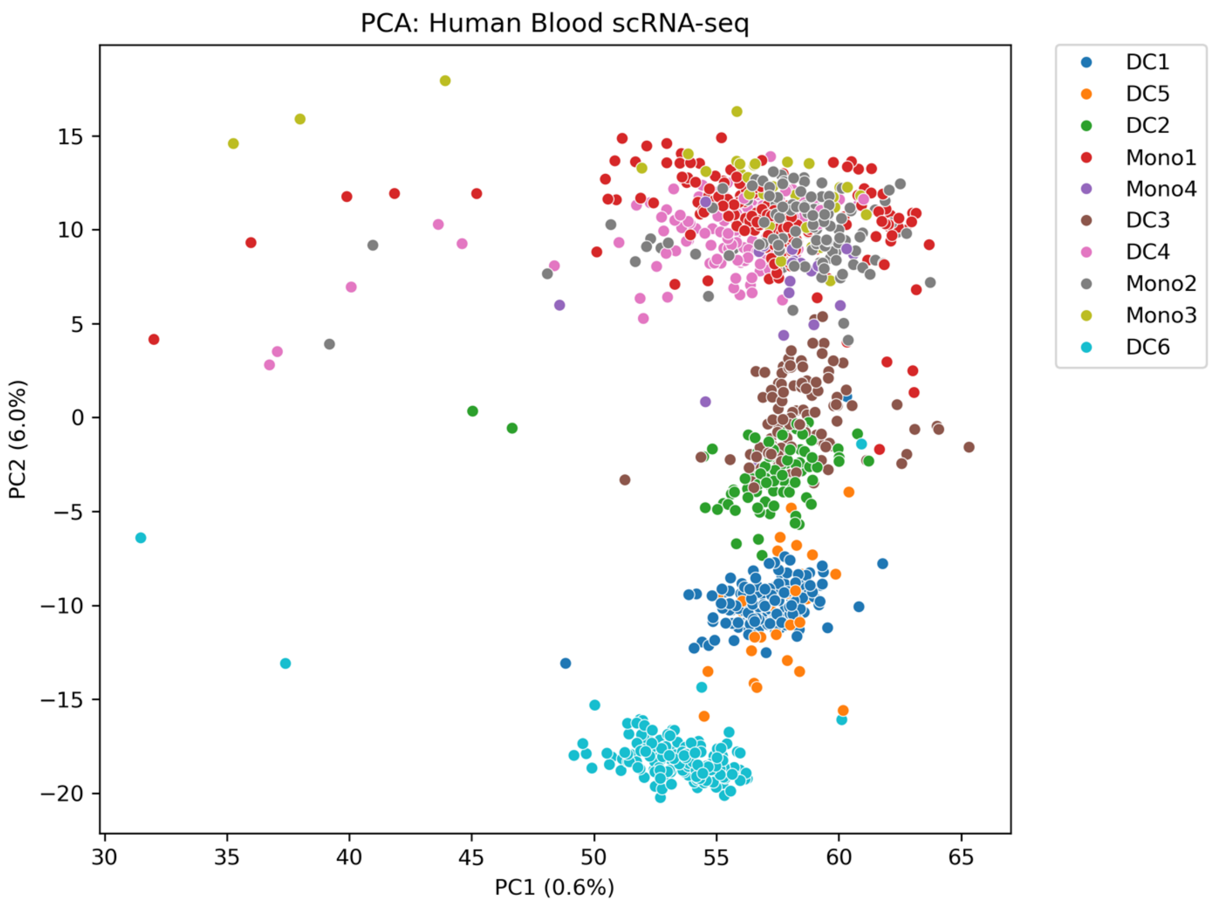
1. Principal component Analysis

PCA plot

I ran PCA through scanpy, which saves the outputs to different locations in the anndata object:

- **obsm**['X_pca'] = actual PCA-transformed data (cells in PC space). Used for plotting and clustering.
- **uns**['pca']['variance_ratio'] = Array of floats where each value is the proportion of total variance explained by a PC.
- **varm**['PCs'] = The PCA loadings, i.e., the contribution of each gene to each principal component. Useful for interpreting what each PC means biologically.

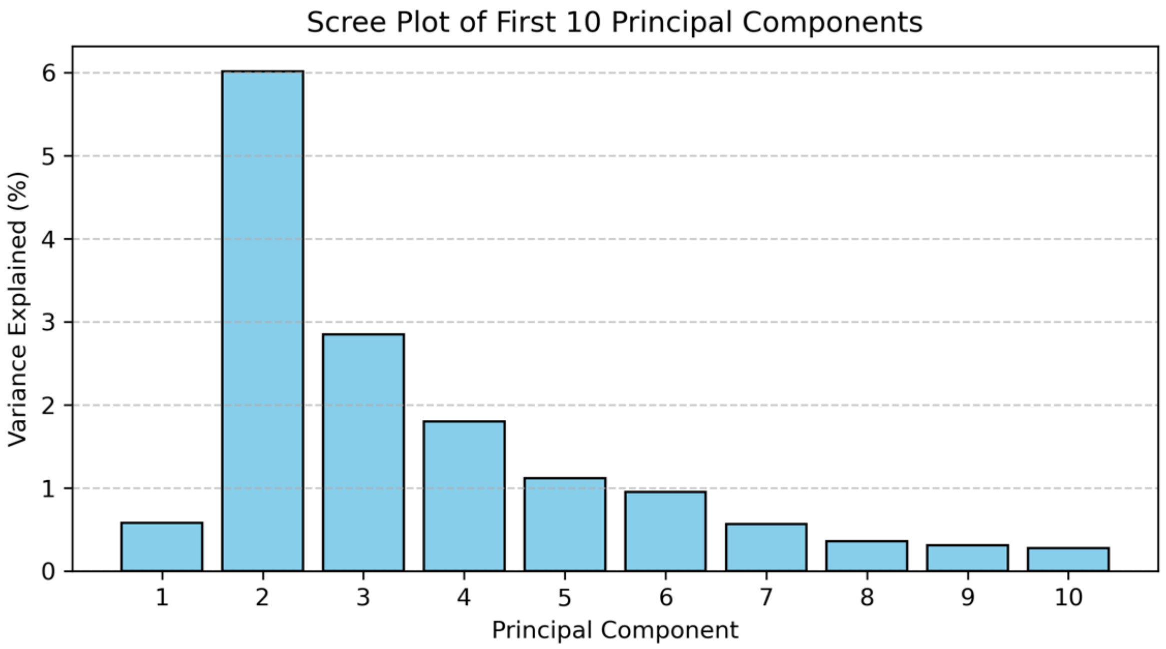
I extracted the variance explained for the first two PCs from uns, as well as the PCA coordinates from obsm. I then plotted the first two PCs:



This plot shows 4 major clusters of cells, 1 for monocytes and 3 for dendritic cells. This shows that there is greater variation in DCs compared to monocytes. We can see a great separation for dendritic cell (DC) type 6, which seems to be quite different from all other cells, while the remaining DC cells have some overlapping points. The remaining dendritic cell types are not as isolated, but they tend to overlap with other dendritic cell types. Generally, all dendritic cells are found close to each other. However, we see a few cells appearing at completely random locations, away from the other cells of their type. Monocytes appear more similar, but they still have some cells that are much farther from the bulk of the data. The variance explained by the PCs is quite low (0.6% and 6.0%) indicating that while PCA reveals some biologically meaningful structure, it may not capture the full complexity of the data. This suggests that other dimensionality reduction methods like t-SNE or UMAP, which can better preserve local neighborhood relationships, might provide improved separation of the cell subtypes in two dimensions.

Scree plot

To generate a scree plot, I extracted the variance values from the unstructured slot in the object and converted it into a percentage. Then, I subset that to the first 10 PCs. This is how the scree plot looks like:



- How many principal components contribute more than 1% to the variance explained?
  ⇨ 4 PCs explain more than 1% of the variance.

- How much of the variance is explained by the first 10 PCs?
  ⇨ The first 10 PCs explain 14.83% of the total variance.

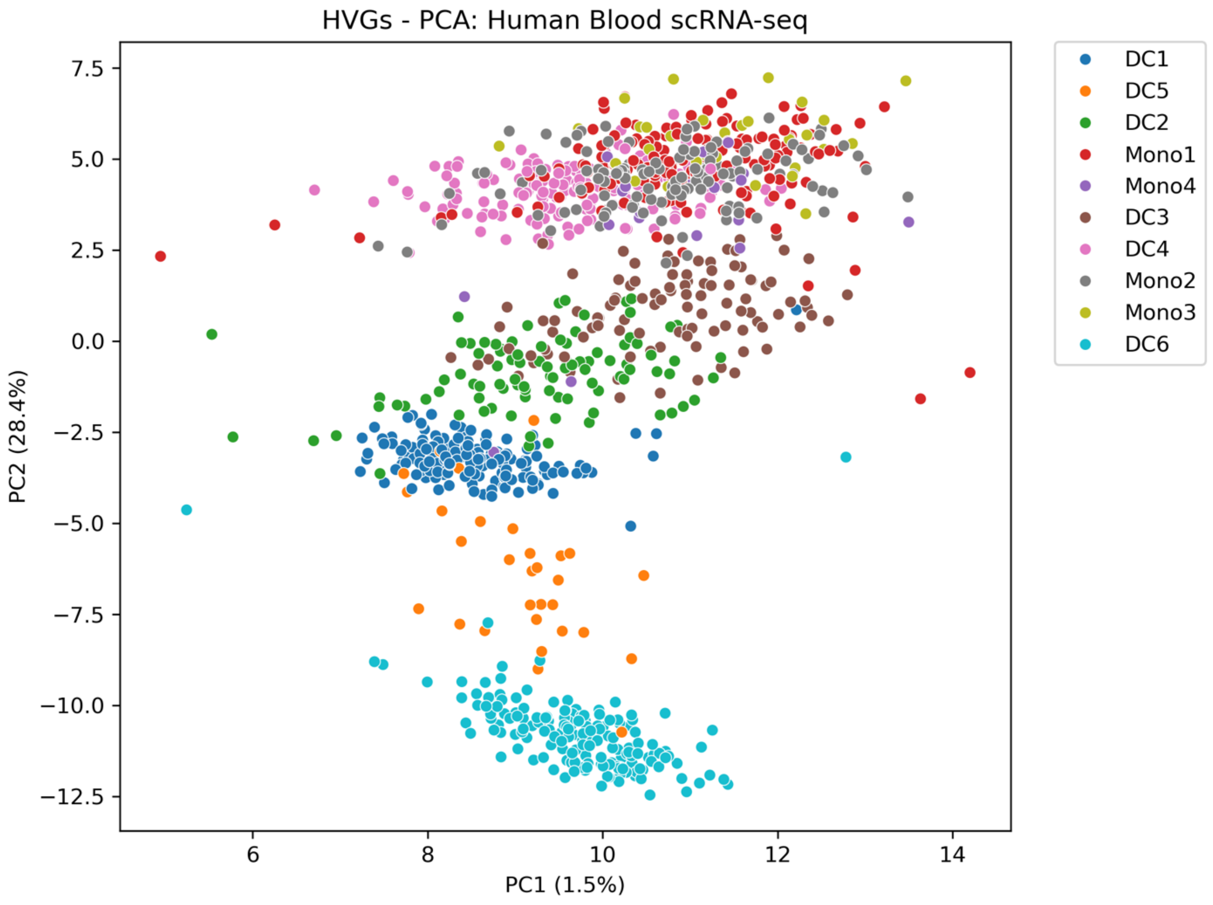
- Do you think the total variance explained by the first 10 PCs is high enough?
  ⇨ The first 10 PCs explain only 14.83% of the total variance, which is relatively low. This suggests that PCA alone may not fully capture the complex structure of the data, and that additional methods may be needed for clearer separation of cell types.

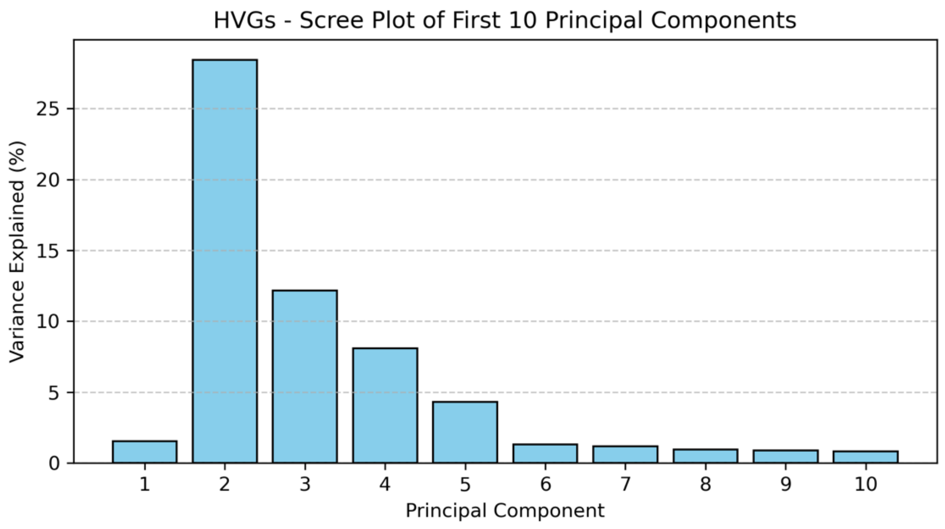
  - If you think the total variance explained is not high enough, what could be one reason causing the low explanatory power of the PCA?
    - The low variance explained is expected for scRNA-seq data due to its high dimensionality and biological heterogeneity. Many genes contribute small amounts of variation, and technical noise such as batch effects can also dilute meaningful signals. PCA aims to reduce dimensionality while preserving as much structure as possible but not to capture all variance. Hence, some loss of information is inevitable.

<u>Repeat using Highly Variable Genes (HVGs)</u>

To detect HVGs, I made a copy of the object and proceeded to use scanpy functionality to detect the top 500 genes, using the Seurat method. This method first bins genes based on their mean expression and computes the dispersion for each gene. That dispersion is standardized per bin (by computing a z-score). Finally, genes with standardized

dispersion values that surpass a certain threshold are selected. Those are the highly variable genes.
I then subset my anndata object to include only the highly variable genes and repeated the process of PCA. The plots for the first 2 PCs along with the scree plot can be found below:





The plot with the first 2 PCs already shows a great improvement in explaining the variance: the first 2 components explain ~30% of the variance. The clustering of the cells is similar the previous plot: certain groups of DCs cluster more distinctly, while others are more far apart. Different points from all cell types seem dispersed at random locations in the plot.

Regarding the scree plot, we can see that 7 PCs explain more than 1% of the variance, while the top 10 PCs explain 59.67% of the total variance, when using only highly variable genes.

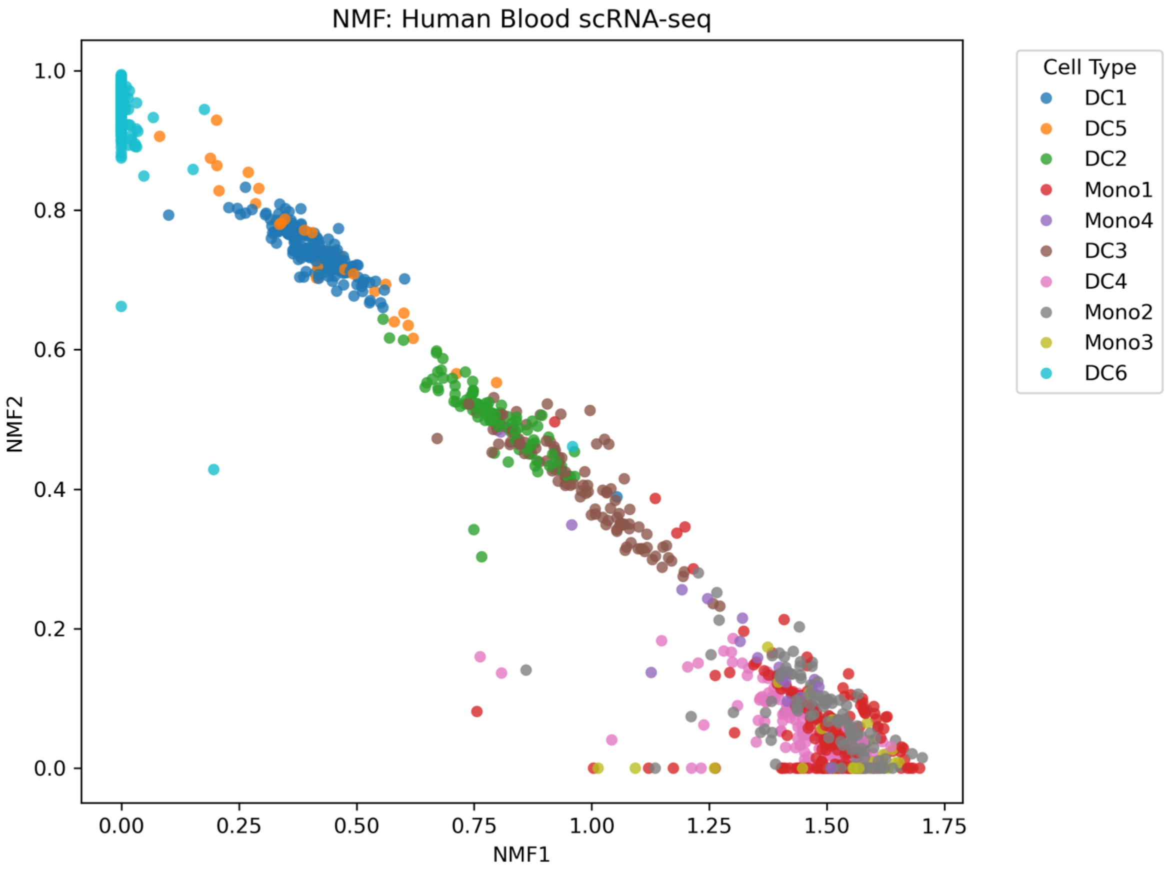
In conclusion, subsetting to highly variable genes did increase the variance explained by the PCs but did not drastically improve the biological signal i.e the separability of cell types in space. We can only see that cells are more dispersed in space, but cell types do not cluster distinctly, apart form DC6.

2. Non-negative Matrix Factorization (NMF)

To prepare the input for NMF, I am making sure to use the normalized and log-transformed matrix, which should contain non-negative values. Then, I converted my expression matrix to a dense numpy array (if the matrix was sparse, I also accounted for that). The result is an array compatible with NMF.

In the command where I initialize the NMF model, I specify 2 components, which correspond to the columns of the factor matrices (W and H). I also chose the 'nndsvda' (Nonnegative Double Singular Value Decomposition with Averages) initialization method, which is suitable for sparse or structured data such as scRNA-seq and helps NMF converge faster, avoiding local minima.

Below I am plotting the 2 components of NMF for all genes in the expression matrix (image in the next page). We can see that monocytes (bottom right) separate from dendritic cells, while the monocyte types 4 and 1 are the ones presenting more similarity to the dendritic cells. Dendritic cell types 2 and 3 have a few overlapping points, and are the closest to the monocyte cluster. A few monocyte (type 4) and DC (type 3) share similar locations. DC type 5 and type 1 also cluster together, farther from the DC2 & DC3, while DC6 form their own cluster farthest away from the monocytes. A few points appear scattered at random locations here (similarly to the PCA).



Why do we have to randomly hide data from the matrix rather than hold out entire cells or genes?

- ⇨ If we removed entire cells (i.e rows) we would not be able to evaluate the model on those cells at all. Similarly, if we removed entire genes (columns), we would also lose all the information about those genes, which would make it impossible for the model to learn them during training.
- ⇨ However, when we use a random mask to the 30% of individual data entries (i.e cell-gene combinations), it is highly unlikely to remove entire columns/rows. The model sees all genes and all cells, just not all possible combinations. Then, the model can be validated on the 30% hidden part, which we call validation set. This is done by comparing the model's predictions to the known values.

(based on given Figure) Which r was the best and why?

- ⇨ Generally, after running NMF we obtain two matrices (W and H) which are used to reconstruct the real matrix, using fewer dimensions.
- ⇨ On the figure in the assignment sheet, we can see two curves which correspond to two different values for Mean Squared Error (MSE). One MSE curve corresponds to the training process, evaluating the error between the reconstructucted matrix and the 70% of the data given to the model. The other MSE curve corresponds to the validation step, which is the error between the reconstructed matrix and the 30% of the data that was masked. Training MSE tends to decrease as r increases; more parameters mean that the fit is better.

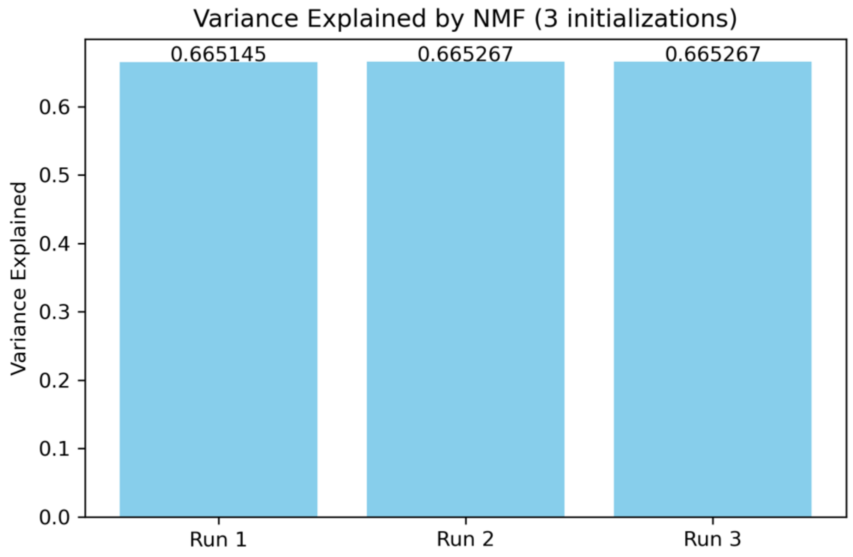
However, validation MSE decreases at first and from a certain value of r onwards it increases, leading to overfitting.

⇨ Given this information, the best factorization rank (r) = 10. This is the point where the training MSE is low, while the Validation MSE reaches the lowest value. From that point after, validation MSE increases which means that there is overfitting of the model. From this point onwards, we will use 10 components for our NMF.

Trying 3 random initializations

Although we will be using the same r (r=10) in this task, the result will differ once we change the random initialization, as the NMF model minimizes a non-convex function. This means that there will be multiple local minima, where the model will converge.

I allowed the model to perform 1000 iterations, to ensure convergence. Here is the result after 3 random initializations:

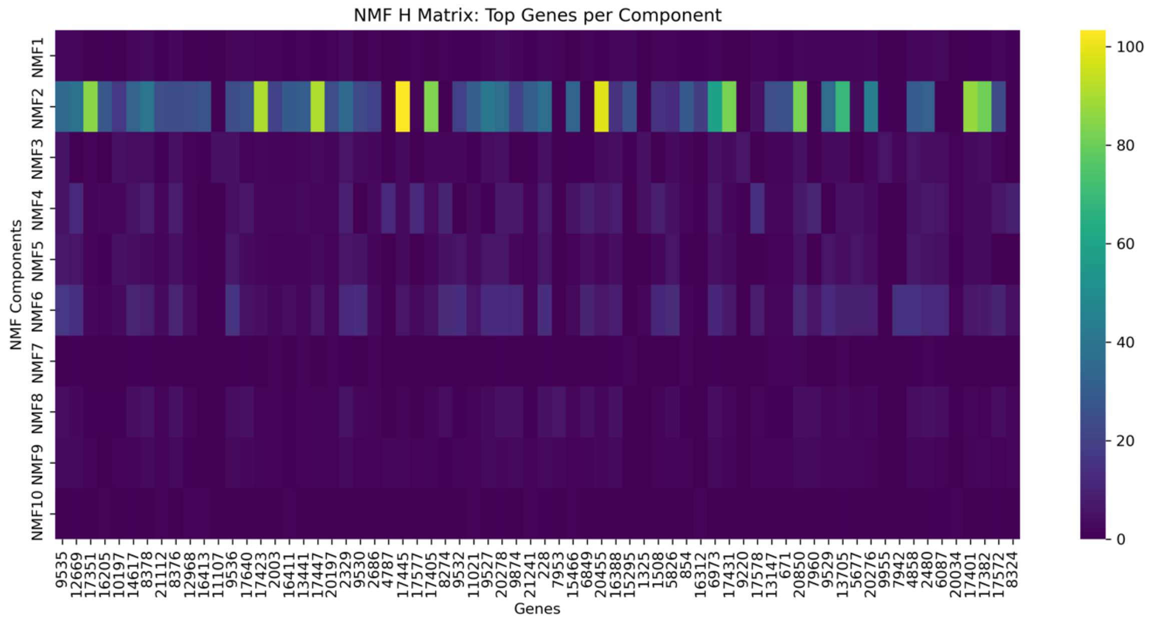


We can see that although each run results in about the same variance being explained, the exact values are not identical among runs. This indicates that the data has a stable structure and NMF is converging to similar local minima. The small variations are simply due to the different initialization.

Creating Heatmap to visualize the H matrix

The H matrix will be of size r=10 rows and 22430 rows (= the number of genes in the original matrix). To make the heatmap more easily interpretable and less computation-heavy, I selected the top 10 genes for each component. To do this, I sorted genes based on their weight for each component and selected the top 10 indices.

The result (next page) shows that component NMF2 captures very high gene weights compared to the other 9 components. This means that it captures a dominant source of variation in the data, which might represent the distinct cell types or, alternatively, a technical artifact or a major biological program such as the cell cycle or differentiation. In the context of our data, this is more likely to be differentiation into the dendritic and monocyte cell types we are studying. The component with the second highest gene weights is the 6th NMF component. This can be capturing more minor differences, such as the within cell type variation (i.e DC1 and DC2 subtypes within the dendritic cell type). In conclusion, a possible interpretation of this plot is that NMF 2 captures the major cell type variations and NMF 6 capture subclass variations, while other components capture smaller differences (i.e what makes certain subtypes more or less similar to each other). However, these statements can only be verified when we have real gene names in our expression matrix. Using those we can see if they are marker genes for the major and minor cell types in our dataset, by comparing them to relevant literature.

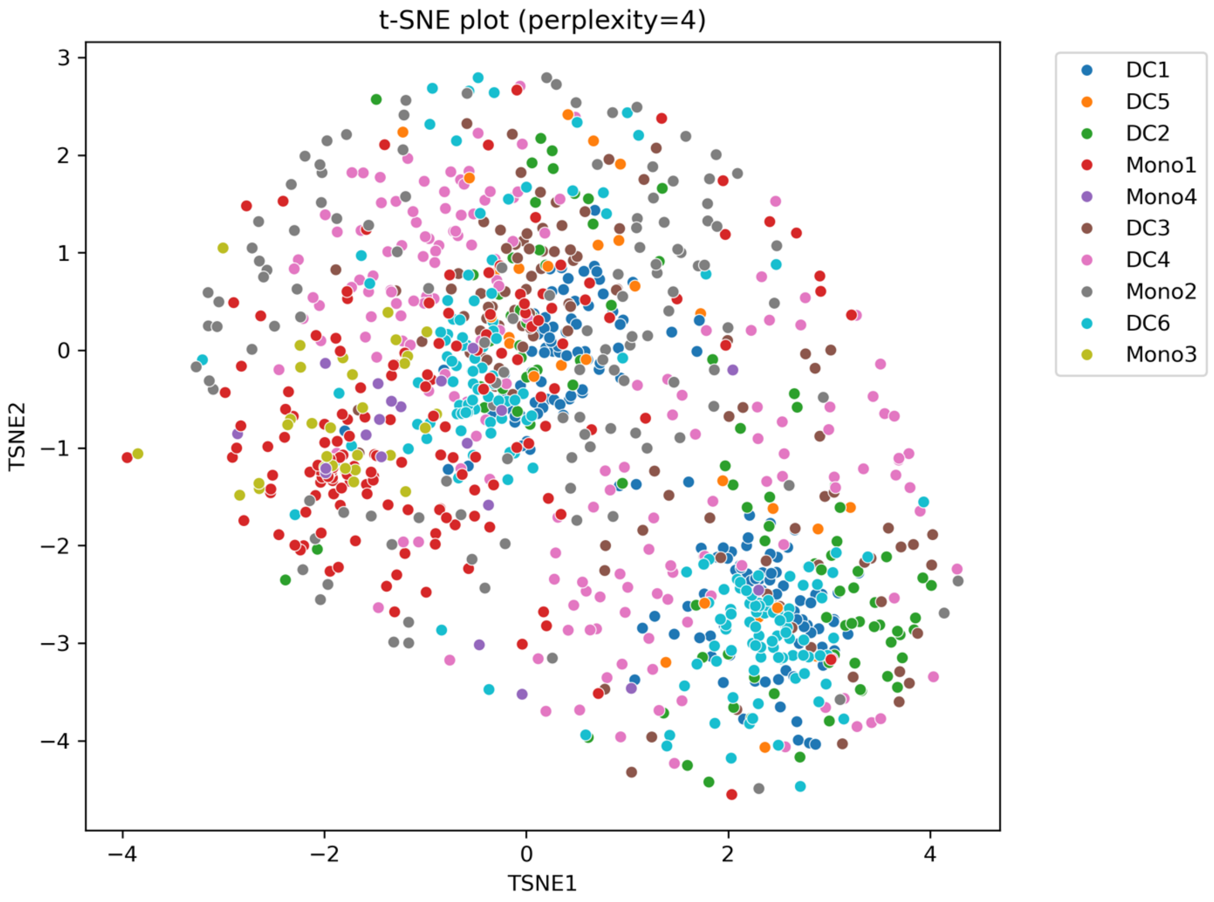


3. t-distributed Stochastic Neighbor Embedding

Usually, the PCA-reduced data is fed into the t-SNE to perform computations faster. However, since the purpose of this assignment is to compare the 3 dimensionality reduction methods, I will be using the raw expression data from the initial matrix.

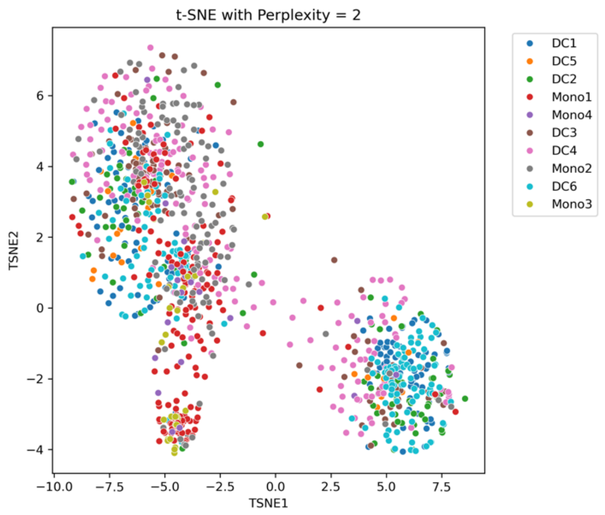
Applying t-SNE

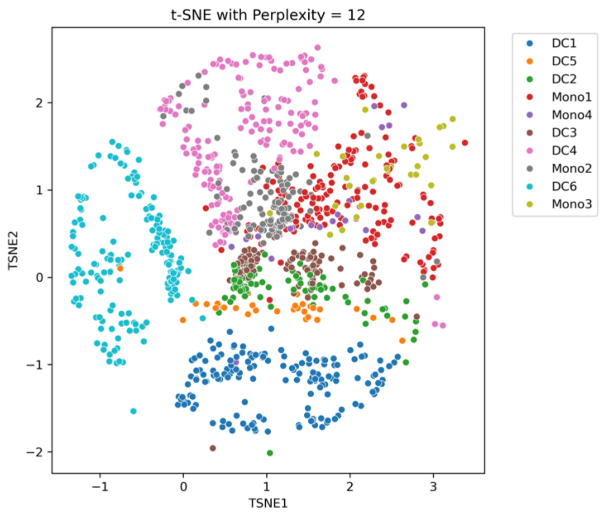
- ⇨ To begin with, I chose to set perplexity = 4 for my first run. After running t-SNE, I saved the result in my anndata object and then created a dataframe to plot the results. In the plot below, we can see that the data does not cluster distinctly based on cell type. 2 major clusters are formed which are composed of both dendritic cells and monocytes, while points appear quite dispersed in space, while it is not clear to which cluster certain points belong.
- ⇨ Perplexity is related to the number of nearest neighbors considered. A lower perplexity means les neighbors and hence emphasizes local structure and hence smaller differences, while a high perplexity means more neighbors and hence emphasizes broader, more global differences.



Trying 3 different perplexity values

⇨ Since I already plotted the result for perplexity = 4 above, I will now also compute the perplexity = 2 and perplexity = 12 results:

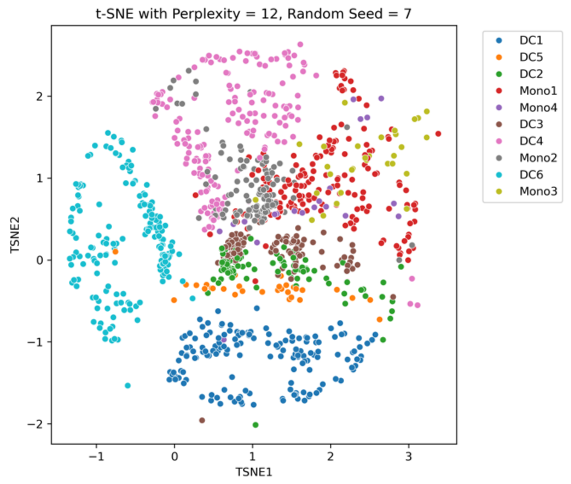


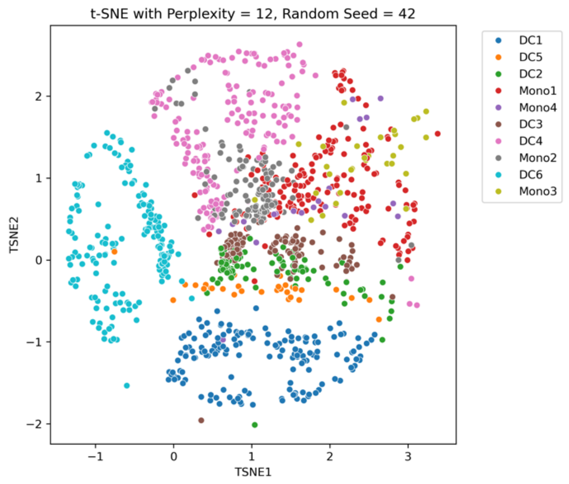


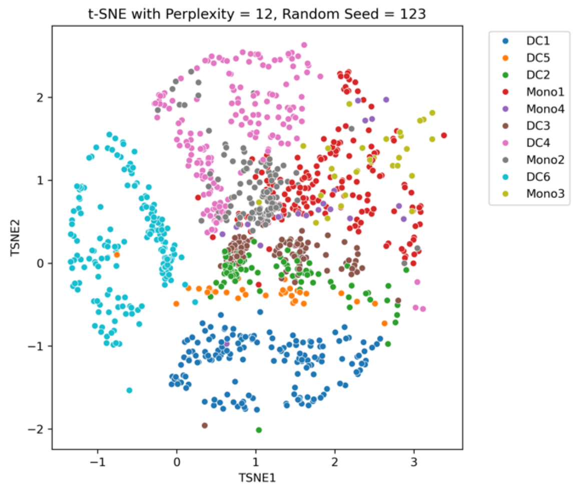
The perplexity = 12 does a much better job at separating the clusters; it is clearly visible that the cells of the same type appear much closer to each other. The division of cell types is still not perfect, meaning that some cell types appear in overlapping areas, while the separation of dendritic cells is much clearer than that of monocytes. Both values of lower perplexity (2,4) are using too few nearest neighbors and hence are unable to capture the cell type level differences that we have in this dataset.

<u>Trying 3 different initializations</u>

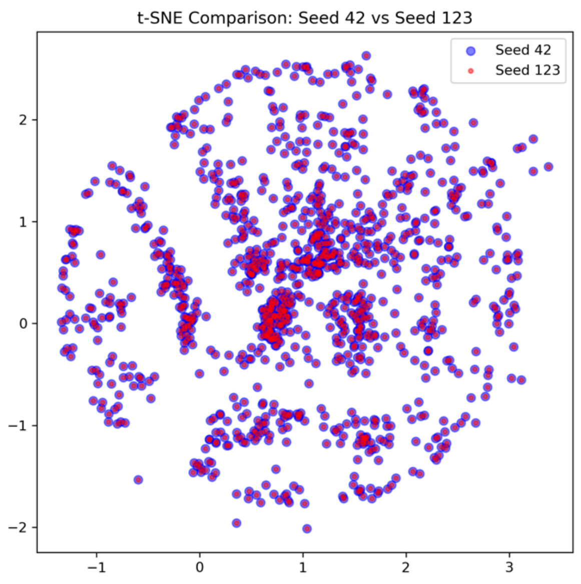
I used 3 different random seeds and obtained the following results:

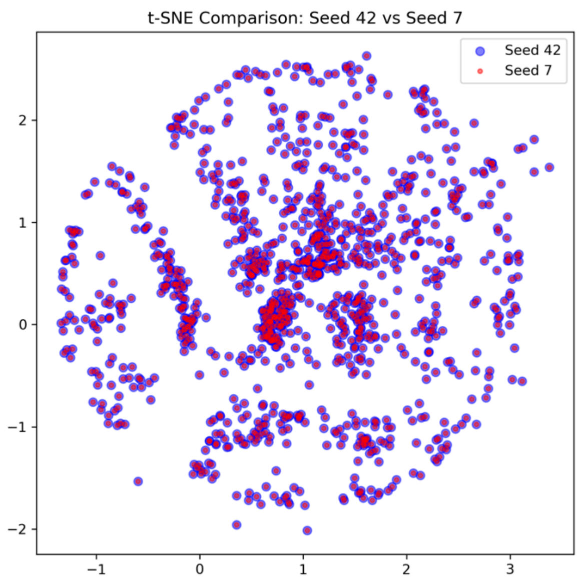






I noticed that the plots looked quite similar so I decided to compare the results of the seeds in pairs (plots in the next page), only to find out that the points actually overlap completely. This can be explained by the fact that t-SNE is a stochastic algorithm, but when the same data and perplexity are used, and the optimization converges similarly across runs, different random seeds can still yield nearly identical embeddings, especially when the data has a strong, well-separated structure that guides the solution toward a stable minimum regardless of initialization.





4. Compare and contrast

Each of the methods used above (PCA, NMF, t-SNE) offers different strengths depending on the data's characteristics. PCA is a linear technique that captures global variance in the data. It is fast and interpretable but often fails to separate biologically meaningful clusters in high-dimensional, sparse scRNA-seq data, where variance is dominated by technical noise or large-scale expression shifts. NMF is a parts-based linear method that can reveal biologically interpretable components by constraining the output to be non-negative. While it captures some structure, its performance can be limited when clusters are not linearly separable or when negative expression values (e.g after log-normalization) are present. t-SNE, a nonlinear method, excels at preserving local neighborhood structures, making it ideal for capturing subtle differences between closely related cell types. This makes it particularly suitable for high-dimensional, sparse, and non-linear datasets like single-cell transcriptomics. In PCA and NMF plots, there was significant overlap between certain DC and monocyte subtypes. The linear nature of these methods likely causes them to miss finer-grained, nonlinear relationships among the cell populations.

t-SNE showed the clearest separation between both DC and monocyte subtypes, preserving local structure and grouping transcriptionally similar cells effectively. Subtypes such as DC1, DC5, DC6, formed distinct, tight clusters that were not as easily resolved using PCA or NMF. Monocytes, while distinguished from dendritic cells did not form as distinct sub-clusters based on the subtypes. This is probably because there is less inter-class variability compared to the dendritic cell class.

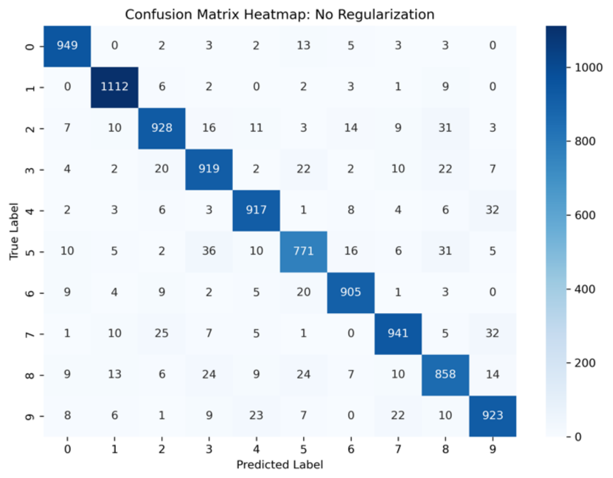
Part 2: Machine Learning

Question 1

We are using multinomial logistic regression to perform classification to more than one classes. Essentially, we are modelling the probabilities of each class and maximizing the likelihood of the correct labels. Applying regularization techniques prevents model overfitting by discouraging the model to learn overly complex patterns that will probably not be present in new (unseen) data. The result for each set is shown below:

=== No Regularization ===

Accuracy: 0.9224
Training time: 704.42 seconds
Confusion matrix:

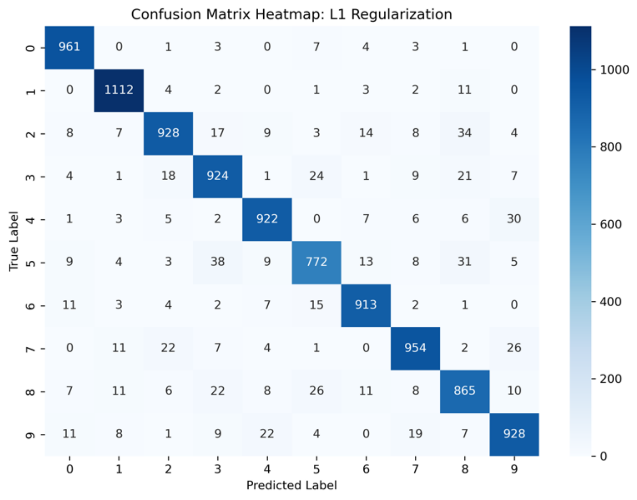


Confusion Matrix Heatmap: No Regularization

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	949	0	2	3	2	13	5	3	3	0
1	0	1112	6	2	0	2	3	1	9	0
2	7	10	928	16	11	3	14	9	31	3
3	4	2	20	919	2	22	2	10	22	7
4	2	3	6	3	917	1	8	4	6	32
5	10	5	2	36	10	771	16	6	31	5
6	9	4	9	2	5	20	905	1	3	0
7	1	10	25	7	5	1	0	941	5	32
8	9	13	6	24	9	24	7	10	858	14
9	8	6	1	9	23	7	0	22	10	923

=== L1 Regularization ===

Accuracy: 0.9280
Training time: 1222.27 seconds
Confusion matrix:

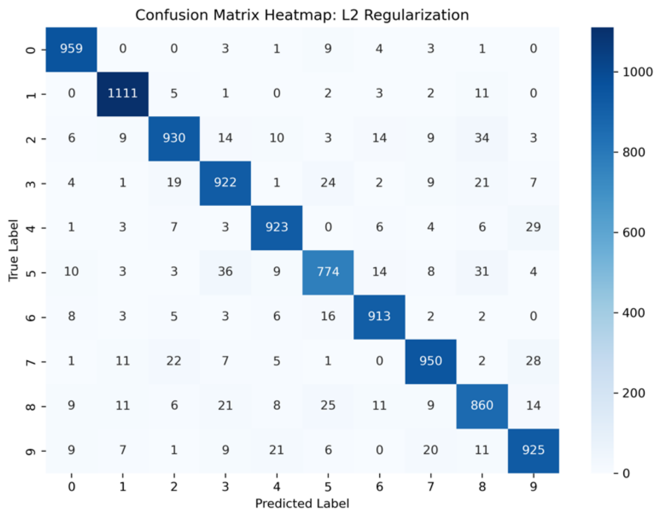


Confusion Matrix Heatmap: L1 Regularization

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	961	0	1	3	0	7	4	3	1	0
1	0	1112	4	2	0	1	3	2	11	0
2	8	7	928	17	9	3	14	8	34	4
3	4	1	18	924	1	24	1	9	21	7
4	1	3	5	2	922	0	7	6	6	30
5	9	4	3	38	9	772	13	8	31	5
6	11	3	4	2	7	15	913	2	1	0
7	0	11	22	7	4	1	0	954	2	26
8	7	11	6	22	8	26	11	8	865	10
9	11	8	1	9	22	4	0	19	7	928

=== L2 Regularization ===

Accuracy: 0.9268
Training time: 358.85 seconds
Confusion matrix:



To evaluate the different sets, we will first look at accuracy: all sets have a value above 90% meaning that they all perform well on our MNIST data. We can also notice that L1 slightly improves accuracy (0.9280), as it works by removing irrelevant weights (sparsity). L2 also shows an increase in accuracy (0.9268) compared to no regularization, suggesting that it helped generalizing the model by penalizing large weights. This is also reflected in the individual confusion matrices, where we can notice a dark blue diagonal forming, with high values inside the cells. This essentially means that the predictions match the true labels most of the time. Comparing the values within the diagonal with the surrounded values colored in light blue, we can see that numbers are being mistaken for similar-looking digits only a few times.

However, when we look at training time, we notice that L1 is the slowest one. This is because it introduces zero weights which do not have derivatives and hence the optimizer must "work harder" near those zero values. Further, since the goal is to make certain weights zero, the optimizer might need to take more steps to decide if a weight should be zero or not. L2 shows quite a high accuracy value, but it's much faster than L1 and takes half as much time as no regularization. Hence, if we wish to have a fast and accurate result L2 regularization is recommended for this dataset.

Question 2

Previously, we obtained probabilities for each number label. In this case, we will be using a Support Vector Machine to find a decision boundary among the different classes. That decision boundary can be linear (Linear SVM) or non-linear (RBF kernel) and detect more complex patterns. Below are the performance metrics for each case:

- **Accuracy** = % of correct predictions
- **Precision** = True positives / (True Positives + False Positives)
- **Recall** = True positives / (True Positives + False Negatives)

- **F1-score** = 2 *[(Precision * Recall / (Precision + Recall)] ,balances precision and recall.
- **Support** = the number of true instances per number label in the test dataset. This is the same for both SVM methods, as they operate on the same test dataset.

Linear Function Kernel

	precision	recall	f1-score	support
0	0.95	0.98	0.96	980
1	0.97	0.99	0.98	1135
2	0.93	0.94	0.93	1032
3	0.90	0.93	0.92	1010
4	0.93	0.96	0.95	982
5	0.92	0.91	0.91	892
6	0.95	0.95	0.95	958
7	0.95	0.93	0.94	1027
8	0.93	0.90	0.91	974
9	0.95	0.90	0.93	1009

Linear Kernel Accuracy: 0.9392939293929393

Training time (Linear Kernel): 137.7397 seconds

Radial basis Function Kernel (RBF)

	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	0.99	0.99	0.99	1135
2	0.97	0.98	0.97	1032
3	0.97	0.98	0.98	1010
4	0.98	0.98	0.98	982
5	0.99	0.98	0.98	892
6	0.98	0.98	0.98	958
7	0.97	0.97	0.97	1027
8	0.97	0.97	0.97	974
9	0.97	0.96	0.97	1009

RBF Kernel Accuracy: 0.9785978597859786

Training time (RBF Kernel): 133.5708 seconds

The SVM with RBF kernel performed better than the linear kernel, with higher overall accuracy and F1-scores, especially for classes that were not linearly separable (for example, numbers 8 and 9, shaded in light blue in both tables). This shows that the RBF kernel's ability to create nonlinear decision boundaries helped capture more complex patterns in the data.

Question 3

I applied the given NN architecture across different combinations: No regularization, L1, L2 and learning rates 0.001, 0.01, 0.1. Before training the model, I ensured that the pixel values were normalized as the activation functions, we are using (eg. ReLU) are quite sensitive input magnitudes. This normalization allows to scale the inputs in the range of 0-1 which makes training more stable and faster, while improving convergence.

All model trainings with the different parameters took only a few seconds to complete, which is the fastest across all methods used so far. The performance of the NN appeared better when no regularization was performed. Immediately after, the best performance was achieved by L2 regularization, while L1 had the poorest performance. For the no regularization set, the best learning rate appeared to be 0.01, while for the L2 and L1 sets, the best learning rate was 0.001. A learning rate determines the step size the optimizer takes in each iteration when trying to minimize a loss function. Hence, a small learning rate is more cautious, meaning it will update the weights by a small scalar, with the caveat of requiring more time to converge. Conversely, a higher learning rate is riskier, meaning that the weights are scaled by larger coefficients meaning that time to train might be less whilst compromising accuracy, as large steps might result in skipping over the minimum value.

Below are the performance metrics for the best set of parameters – ranking prioritizes accuracy and then time to train.

Regularization:	Learning Rate:	Accuracy:	Time:
none	0.01	0.9769	10.02s

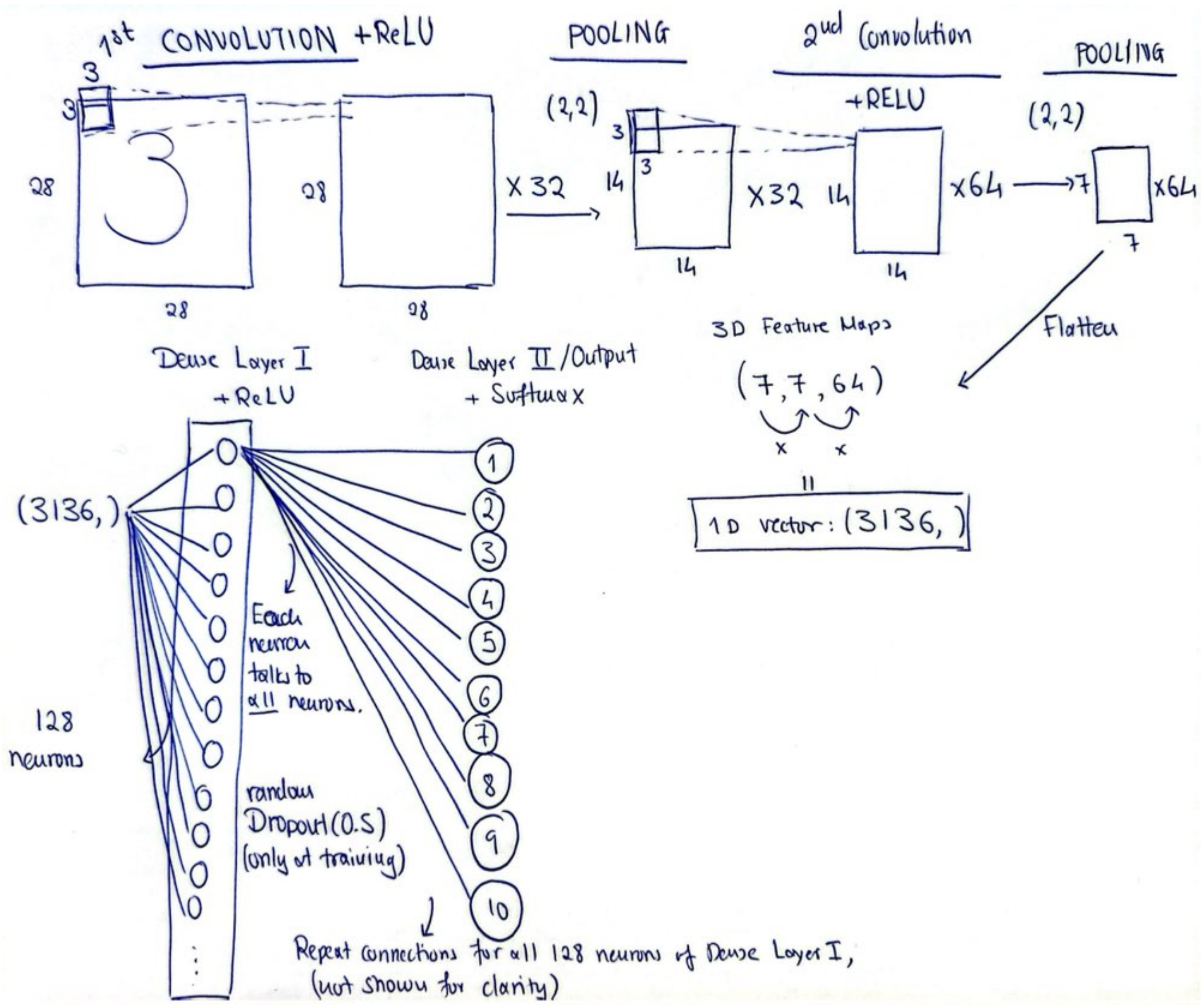
Question 4

<u>CNN structure</u>

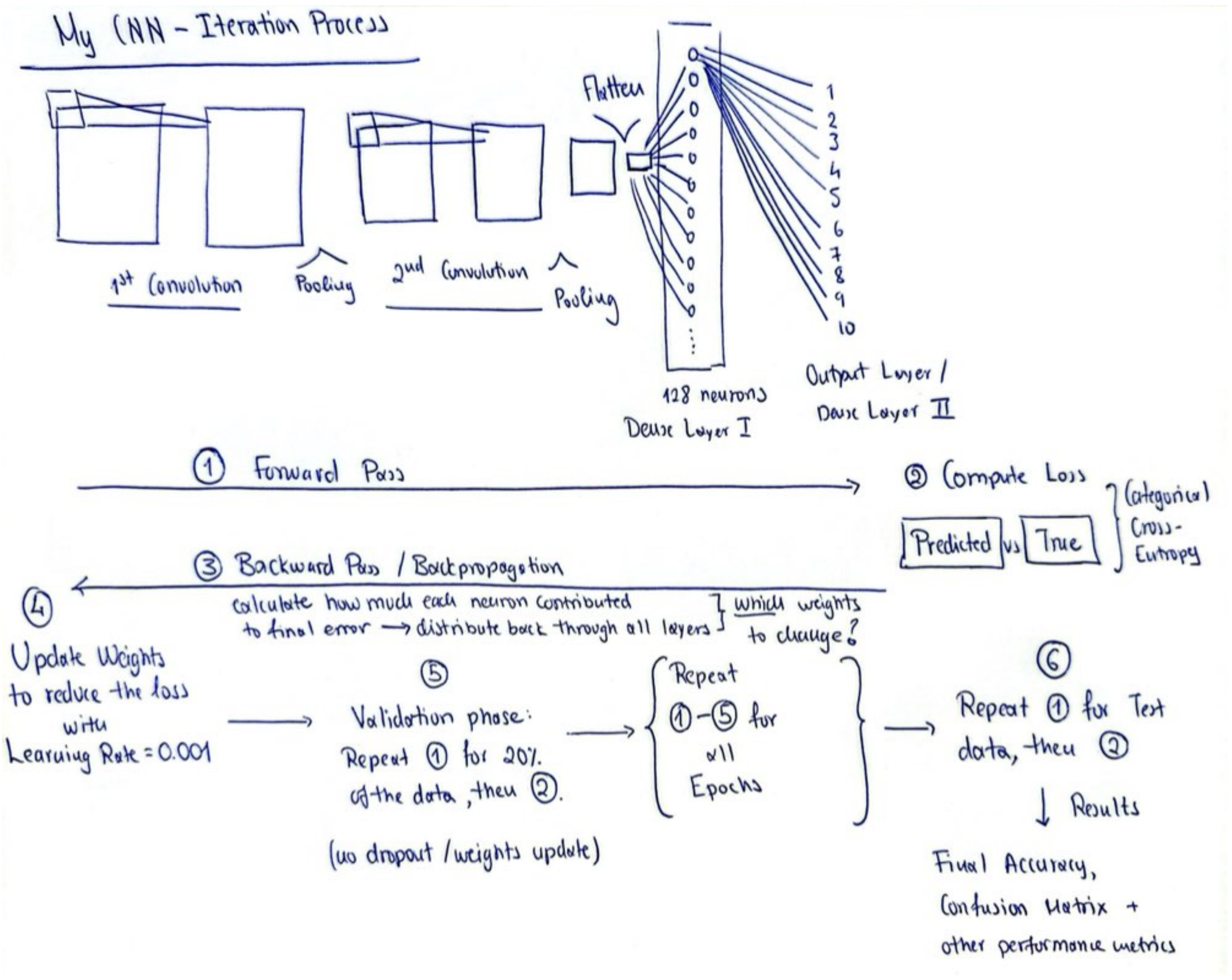
To build my CNN, I decided to use 2 convolutional layers, with pooling in-between, along with one dense hidden layer and finally an output layer. In the first convolutional layer, I am applying 32 3x3 filters to each input image (28x28 pixels), using stride=1 and padding to make the convoluted image the same dimensions as the input image. During pooling, both dimensions of the images are reduced in half. Pooling is crucial to help the model focus on strong signals, while it also reduces the number of pixels to speed up the training and also reduce overfitting (due to reduced parameters). In my 2nd layer, I am using 64 3x3 filters with stride=1 and padding to make the convoluted image the same dimensions as the input image. Pooling follows in the same way as before. Each convolutional layer is also followed by a ReLU activation function, which keeps only the positive weights and discards the negative ones. The goal of the convolutional steps is to detect local patterns, as the filters scan parts of the image.

After that, the data is flattened into a 1-dimensional input vector which will be the input for the fully connected, dense layer, made of 128 neurons. Each neuron takes this 1-dimensional vector and the goal here is to combine the local patterns learned in the previous layers to identify global patterns, i.e the identity of the number in each picture. The 128 neurons are connected with all neurons of the output layer, which is composed of 10 neurons, each corresponding to one number label. Between the hidden and the output layer, I have set random dropout to 50%, essentially applying regularization. It works by randomly dropping (sets to 0) a fraction of the neurons during training. This is crucial in making the model generalizable, ensuring that it does not memorize the training data. This step does not take place during testing or validation.

Below is a hand-drawn schematic of my model:


1st CONVOLUTION +ReLU
POOLING
2nd Convolution +RELU
POOLING
(2,2)
(2,2)
28
X 32
X32
x64
7
x64
14
3D Feature Maps
(7, 7, 64)
x x
=
1D vector: (3136,)
Flatten
Dense Layer I +ReLU
Dense Layer II /Output + Softmax
(3136,)
128 neurons
Each neuron talks to all neurons.
random Dropout (0.5) (only at training)
Repeat connections for all 128 neurons of Dense Layer I, (not shown for clarity)

The following schematic shows how the model iterates through the layers, computes loss and back-propagates and then updates the weights (in my case, using a learning rate of 0.001). This is the training process and is followed by validation using the 10% of the data and then loss is computed again. This process is repeated for all epochs (in my case, 15). Finally, a first pass is performed for the test data, to see how well the model is performing.

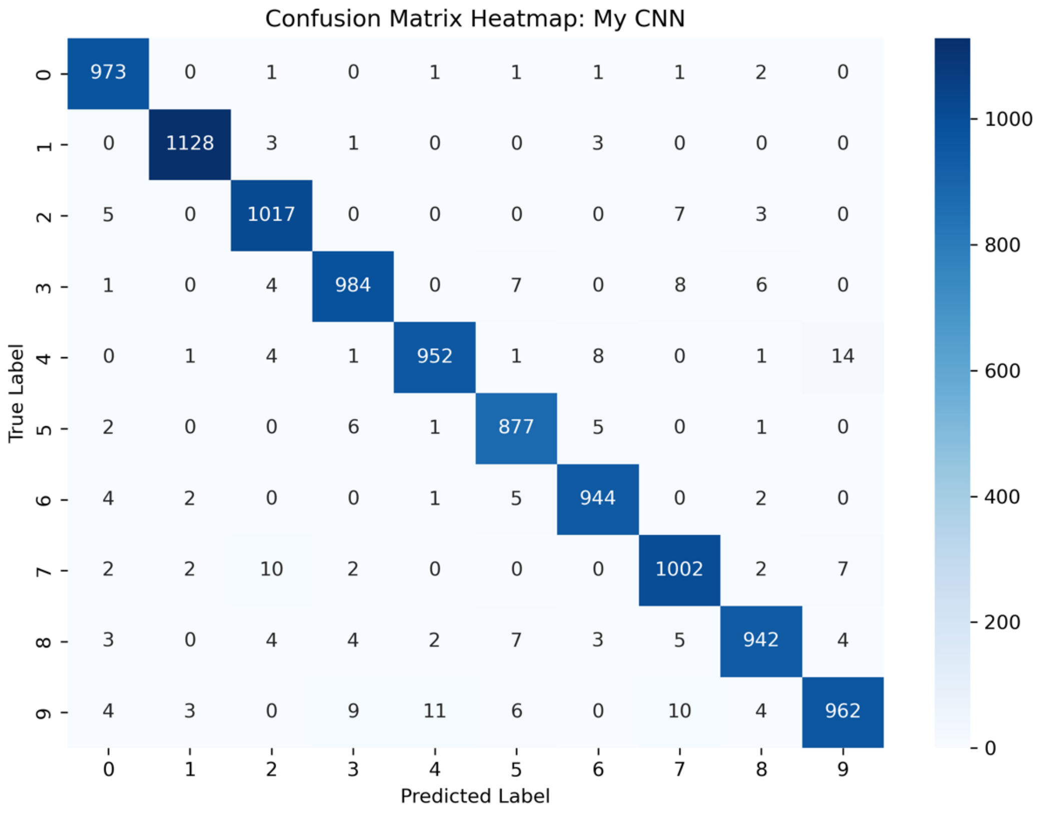

My CNN – Iteration Process
Flatten
1
2
3
4
5
6
7
8
9
10
1st Convolution
Pooling
2nd Convolution
Pooling
128 neurons
Dense Layer I
Output Layer / Dense Layer II
① Forward Pass
② Compute Loss
Predicted vs True
Categorical Cross-Entropy
③ Backward Pass / Backpropagation
calculate how much each neuron contributed to final error → distribute back through all layers
which weights to change?
④
Update Weights to reduce the loss with Learning Rate = 0.001
⑤
Validation phase: Repeat ① for 20% of the data, then ②.
(no dropout / weights update)
Repeat ①–⑤ for all Epochs
⑥
Repeat ① for Test data, then ②
Results
Final Accuracy, Confusion Matrix + other performance metrics

Here are the results of my CNN application:

Accuracy: 0.9818 - val_loss: 0.0715
Training time: 159.02 seconds

The 98.18% accuracy indicates strong generalization to unseen data. Although the accuracy is high, time to train is much higher than other methods, such as the fully connected neural network above, which also achieved marginally lower accuracy. The increase in training time indicates a trade-off when using CNNS: although they are generally better suited to image data, they require more computation due to their multiple convolution and pooling layers. Nonetheless, for tasks where training time or computational resources are limited, and a slight reduction in accuracy is acceptable, a simpler neural network might be a more efficient choice.

Below is a confusion matrix for my CNN model:


Confusion Matrix Heatmap: My CNN
True Label
Predicted Label
0: 973 0 1 0 1 1 1 1 2 0
1: 0 1128 3 1 0 0 3 0 0 0
2: 5 0 1017 0 0 0 0 7 3 0
3: 1 0 4 984 0 7 0 8 6 0
4: 0 1 4 1 952 1 8 0 1 14
5: 2 0 0 6 1 877 5 0 1 0
6: 4 2 0 0 1 5 944 0 2 0
7: 2 2 10 2 0 0 0 1002 2 7
8: 3 0 4 4 2 7 3 5 942 4
9: 4 3 0 9 11 6 0 10 4 962