Guidelines for Implementation and Code Design

1. Use object-oriented programming. For Q1, all sampling and analysis must use the original dataset stored in the `StudentDataset` object.
2. Use appropriate visualization libraries such as `matplotlib`, `seaborn`, or `plotly`. Each plot must include a **title, x-label, y-label, and legend (if applicable)**. Answers with missing labels or legends will receive **0 marks**.
3. Keep visualization and computation logic in **separate functions**.
4. Add `docstrings` to methods explaining what they do.
5. Use reproducible random sampling using the given `seed`.

Q1.0 Dataset Generation [6 marks]

Generate 10,000 student records with the following attributes:

- **gender:** Male (65%), Female (33%), Other (2%) [1]
- **major:** B.Tech (70%), MS (20%), PhD (10%) [1]
- **program:** distribution conditioned on major: [2]

Major	CSE	ECE	CHD	CND
B.Tech	40%	40%	10%	10%
MS	30%	30%	20%	20%
PhD	25%	25%	25%	25%

- **GPA:** Normally distributed by major, clipped to [4.0, 10.0] [2]

Major	GPA Distribution ($\mathcal{N}(\mu, \sigma)$)
B.Tech	$\mathcal{N}(7.0, 1.0)$
MS	$\mathcal{N}(8.0, 0.7)$
PhD	$\mathcal{N}(8.3, 0.5)$

You must implement a class as follows.

```
class StudentDataset:
    def __init__(self, num_students: int, seed: int):
    # Generates the full dataset during initialization using the
       specified number of students and seed.
    def get_full_dataframe(self) -> pd.DataFrame:
    # Do not regenerate the dataset in different methods or cells.
    # Use this method to access the full dataset consistently.
```

```
    def generate_gender(self) -> list[str]: ...
    def generate_major(self) -> list[str]: ...
    def generate_program(self, majors: list[str]) -> list[str]: ...
    def generate_gpa(self, majors: list[str]) -> list[float]: ...
    def assemble_dataframe(self) -> pd.DataFrame: ... # Assemble the
        full dataset from gender, major, program, and GPA.
```

Q1.1 Dataset Analysis

(a) Visualizations [15 marks]

Create suitable visualizations for the following distributions. The visualizations should convey meaningful information about the data.

- gender [1]
- major [1]
- program [1]
- GPA [1]
- program conditioned on major [1]
- GPA conditioned on major [1]
- GPA conditioned on program [1]
- GPA conditioned on program and major [2]
- gender, major, program and GPA of 100 randomly sampled students [3]
- Summary of entire dataset(e.g. pairplots) [3]

You may implement the following methods:

```
def plot_gender_distribution(self) -> None: ...
def plot_major_distribution(self) -> None: ...
def plot_program_distribution(self) -> None: ...
def plot_gpa_distribution(self, bins: int = 20) -> None: ...
def plot_program_by_major(self) -> None: ...
def plot_gpa_by_major(self) -> None: ...
def plot_gpa_by_program(self) -> None: ...
def plot_gpa_by_program_and_major(self) -> None: ...
def plot_sampled_dataset(self) -> None: ...
def plot_entire_dataset_summary(self) -> None: ...
```

(b) GPA Summary Statistics [1 mark]

Define a method to compute the mean and standard deviation of GPA:

```
def gpa_mean_std(self) -> tuple[float, float]: ...
```

Report the results and briefly comment on any observations.

(c) Program-Major Combinations [2 marks]

Define a method to count the number of students for each unique (program, major) pairs. Also write a method to visualize it with a heatmap.

```
def count_students_per_program_major_pair(self) -> pd.Dataframe: ...
def visualize_students_per_program_major_pair(self, counts_df : pd.
   Dataframe) -> None: ...
```

Report the counts and describe any patterns you observe.

Q1.2 Simple vs Stratified Sampling [5 marks]

- Sample 500 students uniformly at random. Repeat 50 times. Calculate the mean GPA and standard deviation for each sample. Report the average mean GPA and the standard deviation of the mean GPAs across the 50 repetitions. [2]
- Repeat using stratified sampling by major (divide population into strata by major based on their proportion in the dataset). Compare the results. [2]
- Which method has lower std deviation? Why? [1]

```
def get_gpa_mean_std_random(self, n: int = 500, repeats: int = 50)
   -> tuple[float, float]: ...
def get_gpa_mean_std_stratified(self, n: int = 500, repeats: int =
   50) -> tuple[float, float]: ...
```

Q1.3 Gender-Balanced Cohort [5 marks]

- Sample 300 students with exact same count across genders. Repeat 5 times. Report gender counts. [1]
- Consider the following **Sampling Strategy A**: Randomly pick a value from a discrete set of categories with equal probability (here, gender). Randomly pick a student from that category. Sample 300 students using this sampling strategy. Repeat 5 times. Report gender counts. [2]

- For each sample size (300, 600, 900, 1200, 1500), repeat the above sampling process 10 times. In each repeat, count the number of students in each gender, find the difference between the largest and smallest counts, and divide by the sample size (n) to get the maximum relative difference. Compute the average of these values over the 10 repeats for each sample size. Plot a histogram of average maximum relative difference (y-axis) versus sample size (x-axis). [2]

```
def get_gender_balanced_counts(self, n: int = 300, repeats: int = 5)
    -> list[dict[str, int]]: ...
def sample_gender_uniform_random(self, n: int = 300, repeats: int =
   5) -> list[dict[str, int]]:
def plot_avg_max_gender_diff_vs_sample_size(self, sample_sizes: list
   [int], repeats: int = 10) -> None: ...
```

Q1.4 GPA-Uniform Cohort [3 marks]

- Using Sampling Strategy A, select 100 students such that their GPA values are approximately uniformly distributed across 10 bins. [1]
- Plot GPA histogram and compare to original dataset's histogram. [1]
- Did you sample with or without replacement? Why? [1]

```
def sample_gpa_uniform(self, n: int = 100, bins: int = 10) -> pd.
   DataFrame: ...
def plot_gpa_histogram_comparison(self, sampled_df: pd.DataFrame) ->
    None: ...
```

Q1.5 Program-Major Balanced Cohort [3 marks]

- Using Sampling Strategy A, select 60 students such that all valid (program, major) combinations are represented approximately equally. [1]
- Show counts and heatmap. [1]
- Were any groups too small? How did you handle it? [1]

```
def sample_program_major_balanced(self, n: int) -> pd.DataFrame: ...
def show_program_major_counts_and_heatmap(self, sampled_df: pd.
   DataFrame) -> None: ...
```

Q2.0 k-Nearest Neighbors [30 marks]

Use k-NN (from sklearn) to predict `gender` based on student features. First, implement the following helper class for feature transformations.

```
class PerFeatureTransformer:
    def __init__(self):
        """Initializes memory for per-feature transformers."""
        ...
```

```
    def fit(self, df: pd.DataFrame, params: dict[str, str]) -> None:
        """Fits transformers for each feature based on the given
           type.
        Parameters:
            df: The dataframe containing features to be transformed.
            params: A dictionary mapping feature name to
               transformation type,
            e.g., {"GPA": "standard", "major": "ordinal", "program":
                "onehot"}.
        """
        ...

    def transform(self, df: pd.DataFrame) -> np.ndarray:
        """Applies the fitted transformers to the corresponding
           features and returns a NumPy array."""
        ...

    def fit_transform(self, df: pd.DataFrame, params: dict[str, str
       ]) -> np.ndarray:
        """Fits and transforms all features in one step using the
           given transformation parameters."""
        ...
```

Now, implement the following class for predicting gender using KNN.

```
class KNNGederPredictor:
    def __init__(self, student_df: pd.DataFrame, username: str):
        """Initializes the predictor with the full student dataset.
           Use the username for plots."""
        ...

    def train_val_test_split(self, test_size: float = 0.2, val_size:
        float = 0.2, seed: int = 42) -> tuple[pd.DataFrame, pd.
       DataFrame, pd.DataFrame]:
        ...
```

```
def get_feature_matrix_and_labels(self, df: pd.DataFrame,
    features: list[str]) -> tuple[np.ndarray, np.ndarray]:
    """
    Extract selected features and gender labels from the
        DataFrame.
    Applies encoding to categorical variables and normalizes
        numeric features. Do not fit encoders or scalers on test
        data. Only transform using previously fitted ones.
    """
```

```
def get_knn_accuracy_vs_k(self, k_values: list[int], distance:
    str = "euclidean") -> list[float]:
    """Calculates accuracy scores for various k values on the
        validation set."""
    ...

def plot_knn_accuracy_vs_k(self, k_values: list[int], distance:
    str = "euclidean") -> None:
    """Plots accuracy scores against k values on the validation
        set."""
    ...

def get_knn_f1_heatmap(self, k_values: list[int], distances:
    list[str]) -> pd.DataFrame:
    """Returns a dataframe with the f1-score for each
        combination on the validation set"""
    ...

def plot_knn_f1_heatmap(self, f1_scores_df: pd.DataFrame) ->
    None: ...

def get_knn_f1_single_feature_table(self, k_values: list[int],
    features: list[str], distance: str = "euclidean") -> pd.
    DataFrame:
    """Creates a table of F1 scores on the test set using only a
        single feature for prediction."""
    ...
```

Perform the following tasks.

- Train/val/test split the dataset and apply the data transforms. [4]
- What value of k (odd values from 1 to 21) gave the highest accuracy on the validation set with Euclidean distance metric? Justify with a plot. [2]
- Repeat the above for distance metrics like Manhattan and Cosine Similarity. [4]
- Report the validation F-1 score vs k for all the three distance metrics. [4]

- Plot a heatmap: k × distance function vs F-1 score. [4]
- Which distance metric performs better? Why might that be? [2]
- Instead of using all student features, an alternative is to use a single feature for prediction. Create an F-1 score table where rows are various values of k, columns are the single features used. Report values for test set for all the distance metrics. [6]
- Which single feature performed the best? How does it compare with the result using all the features? Why? [4]

Q3.0 Linear Regression with Regularization [30 marks]

You will predict `GPA` using student features. Use a validation set to select the hyperparameters.
Start with a function of the following form:

```
def run_poly_regression(X_train, y_train,
                        X_val, y_val,
                        X_test, y_test,
                        degree=1,
                        regularizer=None,
                        reg_strength=0.0):
    """
    Fit a polynomial regression model with optional regularization.

    Parameters:
        degree (int): Degree of the polynomial to fit
        regularizer (str or None): 'l1', 'l2', or None
        reg_strength (float): Regularization coefficient (alpha)

    Returns:
        dict with train, val, and test MSEs, and learned
           coefficients
    """
```

Perform the following tasks.

- For three setups - no regularization, L1 and L2 regularization, repeat the below steps: [8×3=24]
  - Fit polynomial regression models across degrees 1 to 6 [2]
  - Plot polynomial degree vs MSE (on train and validation sets). Describe the trend you observe as degree increases. [3]
  - For each degree, use val MSE to choose the best regularization strength. [1]
  - Plot regularization strength (log scale) vs val MSE for best degree. [2]

- Comment on performance improvement (if any) from regularization. Which overall experimental setup (degree, regularizer) yielded the best test performance? [3]
- For the best setup using L1 regularization, which features had non-zero weights? List the most important predictors for GPA. Repeat the same with L2 regularization. Comment on the differences. [3]

2.0 K-Means Clusteing [10 marks]

2.1 Implement a K-Means Class [6 marks]

Implement the K-Means Clustering Class that replicates the core functionalities of the built-in K-Means library, including methods such as fit(), predict(), and getCost().

- The `fit()` function should: Train the K-Means model by using K clusters on the dataset.
- The `predict()` method should: Assign a cluster number to each data-point based on the centroids obtained from the fit() function.
- The `getCost()` method should: Return the cost of K-Means, i.e., the Within-Cluster Sum of Squares (WCSS).
- The number of clusters (k) must be specified when instantiating the class.
- Ensure it works with the given **dataset** after the required preprocessing is applied.

2.2 Determine the Optimal Number of Clusters [4 marks]

Use the following methods to determine the optimal number of clusters for the given dataset (you are allowed to use inbuilt-library functions):

- **Elbow Method (References)**
  - Vary the value of k and plot the Within-Cluster Sum of Squares (WCSS) against k.
  - Comment on the optimal value of k according to this method.
- **Silhouette Method (References)**
  - Plot the average silhouette score for each k.
  - Choose the value of k that maximizes the silhouette score.
  - What can you say about the clusters obtained at the optimal value?

3. Gaussian Mixture Models [10 marks]

3.1 Implement the GMM Class [6 marks]

- Write your own GMM class.
- Your GMM class should include methods like `fit()`, `getMembership()` and `getLikelihood()`.
- The `fit()` method implements the Expectation-Maximization (EM) algorithm on the dataset to determine the optimal parameters for the model.

- The `getMembership()` method returns the membership values Y_{ic} for each sample in the dataset. The membership value Y_{ic} for a sample x_i is the probability that the sample belongs to cluster c in the given GMM.
- The `getLikelihood()` method returns the overall likelihood of the entire dataset under the current model parameters.
- The `drawLikelihood()` draws the plot of likelihood vs iterations for the number of clusters used.
- Ensure it works with the given **dataset** after the required preprocessing is applied. The number of clusters (k) must be specified when instantiating the class. Plot the graph of overall likelihood vs iteration also (using `drawLikelihood()` function).

3.2 Determine the Optimal Number of Clusters [4 marks]

Use the following methods to determine the optimal number of clusters for the given dataset (you are allowed to use inbuilt-library functions):

- **BIC (Bayesian Information Criterion) (References)**
  - Vary the value of k and plot the BIC against k.
  - Comment on the optimal value of k according to this method.
- **Silhouette Method (References)**
  - Plot the average silhouette score for each k.
  - Comment on the optimal value of k according to this method.

4 Image Segmentation [20 marks]

In this question, you will use your custom-implemented Gaussian Mixture Model (GMM) from the previous question to perform color-based image segmentation on two satellite images **(satellite_1.png and satellite_2.png)**. The goal is to visualize the convergence of the Expectation-Maximization (EM) algorithm by creating a video that shows the segmentation process and the model's log-likelihood at each iteration.

4.1 Image Segmentation [8 marks]

- Load each of the two provided satellite images.
- Fit your GMM to the flattened dataset. Keep number of Gaussian components (k) as 3 to segment the image into distinct regions **(Land, Water and Vegetation cover)**.
- Choose colors that make the 3 distinct segments **clearly visible**. Ambiguous submissions will receive a score of zero.

4.2 Dynamic Visualization Video [12 marks]

- For each of the two satellite images, generate a video that visualizes the step-by-step fitting process of your GMM. You are free to use any library of your preference for video generation.
- Each frame in the video must correspond to a single iteration of the EM algorithm.
- The video frame must be composed of three panels displayed side-by-side:
  - Panel 1 (Left): The original, unchanged satellite image.
  - Panel 2 (Center): The segmented image at the current iteration. To generate this, replace each pixel's original color with the mean color of the Gaussian component it has been assigned to at that specific iteration.
  - Panel 3 (Right): A real-time graph of the log-likelihood versus the iteration number. The plot should update at each frame, showing the curve extend as the algorithm progresses towards convergence.
- Upload the videos to your OneDrive account and share a view-only link in the markdown.

5. PCA [10 marks]

In this question, you will be using the MNIST dataset (28x28 pixels), on which you will apply PCA from scratch.

5.1 Custom Implementation [4 marks]

- Write your own PCA class. Your PCA class should include methods like `fit()`, `transform()` and `checkPCA()`. The `fit()` method obtains the principal components of our dataset.
- The `transform()` method transforms the data using the principal components after we fit the data.
- The `checkPCA()` method should help you check if your class reduces the dimensions appropriately. It should return True if the class works and False otherwise.
- Specify the number of components (n_components) when instantiating the class to define how many dimensions the data should be reduced to.

5.2 PCA on MNIST [6 marks]

- Load the **MNIST dataset (28x28 pixels)** using any library of your preference and construct a **balanced subset of 1000 samples**, ensuring there are exactly 100 examples for each digit (0-9).

- Flatten each 28x28 pixel image in your subset into a single 784-dimensional vector.
- Using the PCA functions you implemented in section 4.1, project the 1000 sample dataset into the following target dimensions: **500, 300, 150, and 30**.
- Plot the graph of explained variance vs. the number of principal components.
- Select any 5 images from these samples. Plot them before dimensionality reduction, and after projecting them back to the original space (do this for each of the final dimension values). Write your observations.

6. PCA + Classification [20 marks]

In this question you will be investigating the effect of dimensionality reduction using Principal Component Analysis (PCA) on the performance of a K-Nearest Neighbors (KNN) classifier. You will use the **8x8 pixel images** MNIST digits dataset, apply your custom PCA implementation, classify the data using sklearn's KNN, and visualize the results.

6.1 Data Loading and Preparation [2 marks]

- Load the MNIST digits dataset, which consists of **8x8 pixel images** of handwritten digits. You can import it using `sklearn.datasets.load_digits()`.
- Split the dataset into a training set and a testing set. Use an 80-20 split and set random_state=42 to ensure your results are reproducible.

6.2 Dimensionality Reduction with PCA [5 marks]

- You will use the custom PCA class you implemented in previous questions (Q4).
- Define a list of dimensions to reduce the data to: **n_components_list = [2, 5, 10, 20, 30, 40, 50, 64].**
- For each value in n_components_list, fit your PCA model on the training data and then use it to transform both the training and testing data.

6.3 Classification with KNN [8 marks]

- For each of the transformed datasets from the previous step, you will **train and evaluate a KNN classifier**. You can use the KNeighborsClassifier from sklearn.neighbors.
- You need to test the classifier's performance for different values of k (the number of neighbors). Define a list of **k_values = [5, 25, 50, 100]**.
- For each combination of n_components and k, perform the following:
  - Initialize and train a KNeighborsClassifier model with the current k value on the transformed training data.

- Use the trained model to predict the labels for the transformed test data.
- **Calculate the accuracy** of the predictions and store it.

6.4 Visualization and Analysis [5 marks]

- Create a single plot to visualize the relationship between the **number of principal components and the classification accuracy** for each value of k.
- Based on your graph, briefly discuss your findings. How does the number of principal components affect the accuracy of the KNN classifier? How does the choice of k influence this relationship? Is there a point of diminishing returns where adding more components does not significantly improve accuracy?

7. Enhancing Clustering Accuracy with Data Transformation [20 marks]

You are provided with a modified version of the MNIST dataset containing 1,000 digit images and their labels **here**. Unlike the standard MNIST dataset, these images have been altered, making it challenging for clustering algorithms to correctly group them. Your task is to investigate this dataset, analyze the nature of the alterations, and design an appropriate transformation strategy. Use your custom implementation from previous questions for any algorithm that you use in this question.

7.1 Input

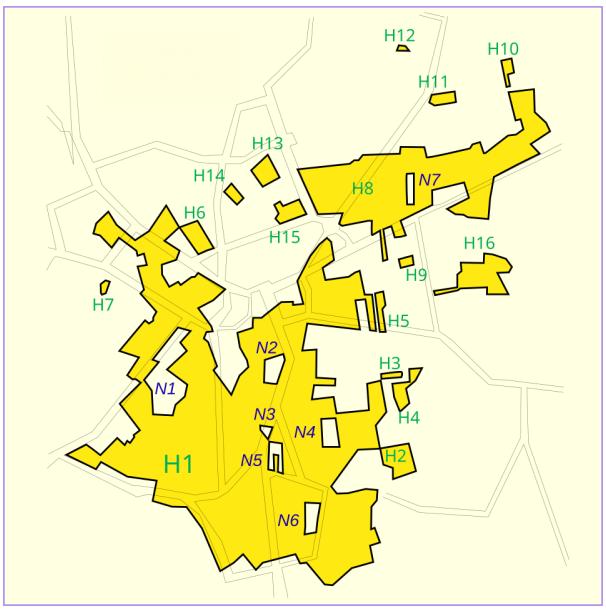
- `X_modified.npy`: Array of 1,000 altered MNIST digit images of shape (1000, 28, 28).
- `y_true.npy`: Ground truth digit labels of shape (1000,).

7.2 Expected Output

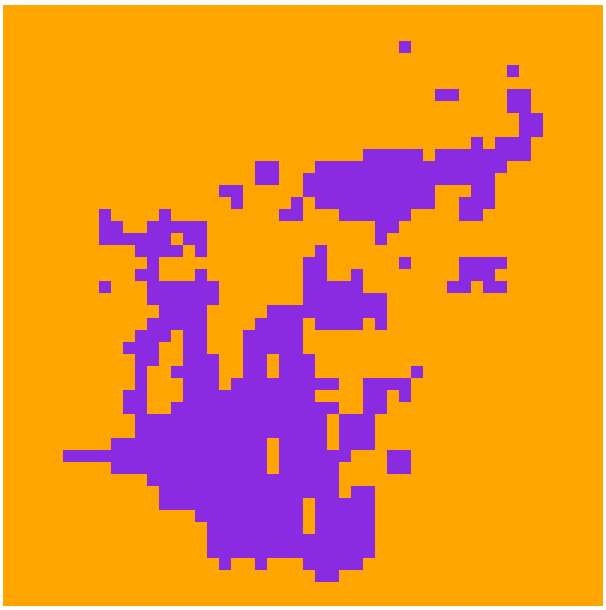
- A clear description of the observed differences between the modified dataset and standard MNIST. [4]
- A suitable transformation technique that addresses these differences, with justification. [6]
- A comparison of clustering performance before and after applying your transformation, using an evaluation metric of your choice. [5]
- A concise analysis explaining why the transformation improved clustering outcomes. [5]

Good luck with the assignment

1 Belgium Netherlands Border [40 Marks]



(a) The Baarle-Nassau border



(b) Pixelated input

1.1 Image Processing and Dataset Creation [5]

The border between Netherlands and Belgium at Baarle-Nassau is a complex one. Your job here is to best model the border using an MLP. Provided is a 50 by 50 pixelated version of the border image where pixels are colored to represent two countries: orange for Netherlands and purple for Belgium.

1. **Binary Mask Conversion:** Convert the input image into a 0-1 binary mask.
2. **Dataset Class:** Create a dataset class with a function that randomly returns all the pixels from the image in a shuffled order. The format for each returned sample should be: $((x, y), L)$, where:
   - (x, y) are the pixel coordinates, normalized to the range $[0, 1]$.
   - L is the corresponding label, where $L \in \{0, 1\}$.

1.2 Neural Network Implementation from Scratch [15]

Core Components

1. **Linear Layer Class:** Implement a 'Linear' layer class that takes input width, output width, and an activation function as arguments. It must store all data required for both forward and backward passes, including:
   - Forward pass data: weights, biases.

- Backward pass data: layer outputs, cumulative gradients.

2. **Activation Function Classes:** Implement each of the following activation functions as a separate class. Each class must contain methods for both the forward pass (calculating the activation) and the backward pass (calculating the derivative).

   - ReLU
   - Tanh
   - Sigmoid
   - Identity (no activation)

3. **Model Class:** Implement a 'Model' class that accepts a list of 'Linear' layers and a loss function type.

   - The model must implement 'forward' and 'backward' pass functions that sequentially execute the corresponding methods in each layer.
   - For now, implement the Mean Squared Error (MSE), and Binary Cross Entropy (BCE) loss function.

   Additionally, implement the following methods in the 'Model' class:

   - `train(x, y)`: Performs a forward pass on the input 'x' to get a prediction y_{pred}. It then computes the loss between y_{pred} and the true label 'y', backpropagates the loss, and adds the resulting gradients to the cumulative gradient stored in each layer. **Do not update the parameters in this step.**
   - `zero_grad()`: Resets the cumulative gradient in each layer to zero.
   - `update()`: Updates all model parameters using the values in the cumulative gradient, and then calls `zero_grad()`.
   - `predict(x)`: Performs a forward pass on the input 'x' and returns the prediction y_{pred}.
   - `save_to(path)` and `load_from(path)`: These methods should save/load all model parameters to/from a file in `.npz` format. Loading should throw an error if the architecture of the current model (i.e., the shapes of weights and biases arrays) does not match the shapes of the arrays in the saved file.

4. **Training Procedure:** Implement a training loop that takes `batch_size` and `grad_accumulation_steps` as parameters.

   - For each batch, call the `train()` method.
   - After `grad_accumulation_steps` have been completed, call the `update()` method to update the parameters.
   - Keep track of the loss at each iteration. Log the training progress (e.g., using `tqdm`).

- Upon completion of training, save both a plot of the training loss versus the number of samples seen and the final model parameters to a unique run folder (e.g., `runs/{timestamp}`).
- Save all the run details, include hyperparameters, losses, accuracies to WandB.
- **Note on Plotting:** When plotting the loss vs. samples graph, each data point a model trains on is considered a single sample. Ensure the x-axis is adjusted correctly by factoring in `batch_size` and `grad_accumulation_steps` to allow for consistent comparisons between runs with different hyperparameters.

5. **Early Stopping:** Implement an early stopping mechanism. As there is no separate validation set, training should stop when the loss drops by less than 1% over the last 10 epochs. Formally, stop if $L_t \geq 0.99 \cdot L_{t-10}$. The parameters `patience=10` and `relative_loss_threshold=0.01` should be configurable.

1.3 Sanity Check [5]

1.3.1 The XOR Problem [2]

Before applying the model to the map data, test its correctness on a simple dataset.

1. Create a dataset for the XOR function: $\{0,1\} \times \{0,1\} \to \{0,1\}$. The model should predict $x_1 \oplus x_2$.
2. Train and test the model with a variety of architectures. Vary the number of layers, the width of each layer, and use all implemented activation functions.
3. Ensure that the model converges to 100% accuracy for all training runs.

1.3.2 Gradient Approximation [3]

When running the above, also verify the gradients are being computed correctly. Create a function to compute the per-weight gradient :

$$\nabla_{\theta_i} L(x;\theta) = \frac{L(x;\theta + \epsilon \cdot \hat{\theta}_i) - L(x;\theta - \epsilon \cdot \hat{\theta}_i)}{2\epsilon}$$

And verify that the obtained values converge to $\nabla_\theta L(x;\theta)$ obtained via backpropagation. (For small ϵ)

1.4 Map Prediction and Analysis [5]

Use the implemented neural network to predict whether each pixel in the map belongs to France or Belgium. The input is the 2D coordinate (x, y), and the label is $L \in \{0, 1\}$.

1. Train multiple model architectures (width of layer, number of layers, activation functions) until convergence (as determined by early stopping). Compare the final loss and accuracy of the predicted maps. Accuracy is defined as the proportion of correctly classified pixels.

2. For each run, save the following images side-by-side in a single plot within the run folder:
    - The ground-truth map.
    - The map generated from the model's predictions.
    - An error map showing misclassified pixels overlaid on the ground-truth image.
3. **Experimentation with Architecture:**
    - For a fixed layer width, vary the number of layers. Plot the final loss and accuracy as a function of depth.
    - For a fixed number of layers, vary the width of the layers. Plot the final loss and accuracy as a function of width.
4. **Experimentation with Hyperparameters:**
    - Train with a variety of `batch_size`, `grad_accumulation_steps`, and learning rates.
    - Provide a comparison of the time taken to converge and the total number of samples needed to converge for different hyperparameter settings.

1.5 Final Challenge [10]

Based on the insights from the previous experiments, attempt to achieve a target accuracy of 91% .

1. **Goal 1: Minimize Model Size.** Your first goal is to achieve the target accuracy with the minimum possible number of parameters. Comment on the process and reasoning that led you to your optimal architecture.
2. **Goal 2: Minimize Training Samples.** Your second goal is to achieve the target accuracy using the minimum number of training samples to converge. Comment on how you arrived at the optimal training parameters (learning rate, batch size, etc.).

Note: You may apply multiple training methods with checkpointing to achieve optimal results. Experiment and record your methodology.

2 Feature Mappings for Image Reconstruction [30 Marks]

This question is a continuation of the previous one where we attempted to reconstruct an image directly from raw inputs. Recent work Tancik et al., 2020 demonstrated that mapping inputs through **Fourier feature expansions** enables neural networks to approximate high-frequency functions much more effectively. In contrast, using raw inputs or low-order polynomial (Taylor) expansions provides different inductive biases and captures different aspects of the data.

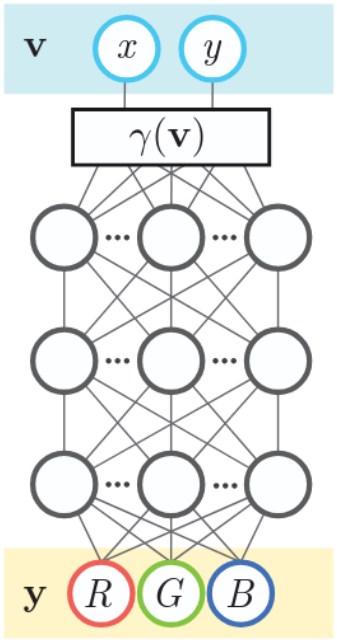


Figure 2: Illustration of an MLP with Fourier feature mapping applied to 2D input coordinates (x, y).

We will compare three different feature representations when training an MLP. The idea is to apply a transformation $\gamma(\mathbf{v})$ on the input pixel/coordinate vector $\mathbf{v} = (x, y)$ before feeding it into the network:

1. **Raw Pixels**: directly use pixel coordinates or intensities.
2. **Taylor Series Expansion**: approximate coordinates using polynomial basis functions.
3. **Fourier Feature Expansion**: map coordinates into a sinusoidal embedding.

2.1 Setup and Visualization [4]

- Load a grayscale image (`smiley.png`) from the following link, which is already 256×256: smiley.png

- Load an RGB image (`cat.jpg`) from the link below and resize it to 256×256: cat.jpg
- The task is to reconstruct these images using the three feature mappings: Raw, Polynomial, and Fourier.

2.2 Feature Expansions [9]

Raw Mapping The simplest approach is to directly use the 2D coordinates or pixel values as input features to the network. This mapping does not introduce any additional basis functions or transformations:

$$\gamma_{\text{raw}}(x, y) = (x, y)$$

```
def get_raw(coords):
    return coords
```

Polynomial Expansion (Taylor-inspired) Instead of using the full Taylor series, we construct a simple polynomial basis up to order k to approximate nonlinear variations in the input coordinates. This expansion allows the network to represent higher-order relationships between x and y without explicitly computing derivatives:

$$\gamma_{\text{poly}}(x, y) = \left[x, y, x^2, y^2, xy, \ldots, x^k, y^k\right]$$

```
def get_polynomial(coords, order=5):
    # Expands into [x, y, x^2, y^2, xy, ..., x^order, y^order]
```

This polynomial expansion is used to approximate a Taylor series and gives the network richer input features compared to raw coordinates.

Fourier Features Including all cross terms in a true 2D Fourier expansion i.e., all combinations of sine and cosine functions of x and y would drastically increase computation and memory usage. To avoid this combinatorial explosion, we use a practical trick: we compute sinusoidal embeddings for each coordinate independently and then stack them together. Concretely, we take

$$\gamma_{\text{fourier}}(x, y) = \begin{bmatrix} 1, \sin(2\pi f_1 x), \cos(2\pi f_1 x), \ldots, \sin(2\pi f_k x), \cos(2\pi f_k x), \\ \sin(2\pi f_1 y), \cos(2\pi f_1 y), \ldots, \sin(2\pi f_k y), \cos(2\pi f_k y) \end{bmatrix}$$

```
def get_fourier(coords, freq=10):
    # Practical Fourier feature approximation:
    # stack 1, sin/cos of x, sin/cos of y
    # Avoids cross-term combinatorial explosion
```

This approach keeps the number of features manageable while still capturing high-frequency variations along both axes.

2.3 Normalization and Modular MLP [3]

Each feature expansion produces values on very different scales, which can strongly affect training. To make fair comparisons across Raw, Polynomial, and Fourier mappings, proper normalization is essential.

- **Raw features:** scale pixel coordinates or intensities to $[0, 1]$.
- **Polynomial features:** higher-order terms (e.g., x^k, y^k) can grow very large; rescale these to avoid exploding values.
- **Fourier features:** sine and cosine outputs lie in $[-1, 1]$. Input coordinates should be scaled accordingly to roughly cover one full cycle of the sinusoidal functions - consider how this choice affects aliasing and high-frequency representation.

Define a modular data loader:

```
Modular_Dataloader(img_path, image_type, method, order, freq)
```

It accepts the following arguments:

- `img_path`: path to the image (RGB or grayscale)
- `image_type`: either `"RGB"` or `"Gray"`
- `method`: feature mapping to use (`"Raw"`, `"Polynomial"`, or `"Fourier"`)
- `order`: order of polynomial expansion (if using Polynomial)
- `freq`: number of Fourier frequencies (if using Fourier)

The loader produces the corresponding input features ready for the MLP. You can use the same MLP from the previous question and modify it slightly to accept this dataloader as input.

2.4 Training and Comparison [4]

We now use all the building blocks from previous sections to train and compare the different feature mappings. The training setup is as follows:

- Use the same MLP architecture for all methods. For example, a 3-layer MLP with hidden sizes $[64, 128, 128]$. Keep the architecture constant to allow a fair comparison.
- Use the same number of epochs for each image:
  - 50 epochs for `smiley.png`
  - 150 epochs for `cat.jpg`
- Use the same loss function as in the previous question (e.g., MSE between predicted and target pixel values).

Baseline: Raw Features : Train the MLP using raw pixel coordinates and save the predicted output at each epoch.

Polynomial Features : Train with different polynomial orders: $[5, 15, 25]$ and record the images for each of them .

Fourier Features : Similarly, train using Fourier feature expansions with different numbers of frequencies: $[5, 15, 25]$. Record the images

Tabulation of Results Create a table comparing:

- Method (Raw / Polynomial / Fourier)
- Final Inference Loss
- Epoch time
- Number of input parameters

Visualization with GIF : To better understand convergence and efficiency:

- Pick one working combination each of Polynomial and Fourier along with Raw features for the same architecture and number of epochs.
- Use the saved images for each epoch to create a 1×3 subplot GIF:
  - Left: Raw
  - Middle: Polynomial
  - Right: Fourier
- Include a legend in each subplot showing the loss curve.
- You can generate the GIF using the `PIL` library or any other preferred tool.

This visualization allows us to directly compare how each feature mapping affects convergence speed and reconstruction quality over training.

2.5 Reconstruction on Blurred Images [10]

For this exercise, we consider a dataset consisting of the same cat image blurred with Gaussian blur levels $\sigma = 0, 1, \ldots, 10$. The images are numbered from `blur_0.jpg` to `blur_10.jpg` and can be accessed here.

Train the coordinate MLP to reconstruct all blurred images using the following methods:

- **BASE:** Using raw input coordinates
- **FOURIER:** Using Fourier feature expansion with frequency $k = 5$

Training Instructions

1. Use early stopping as implemented previously to determine convergence. You may start with a maximum of 100 epochs for each image.

2. For each image and method, record:

    - The reconstructed image
    - The final reconstruction loss

Analysis and Visualization

1. Plot the final reconstruction loss across all images:

    - X-axis: image index or blur level (`blur_0` to `blur_10`)
    - Y-axis: reconstruction loss
    - Include both BASE and FOURIER results on the same plot

2. Plot the same comparison using a logarithmic scale for the Y-axis to highlight differences in low-loss regimes.

Discussion

- Observe how the reconstruction performance changes with increasing blur. Explain the observed behavior in terms of how high-frequency details are lost as blur increases, and verify your reasoning by inspecting the predicted images.

3 AutoEncoders [30 marks]

3.1 Part 1: Autoencoder for Image Reconstruction [12]

In this part, you will build and train a deep autoencoder from scratch to reconstruct images from the MNIST dataset. You should utilize the MLP components (Linear layers, Activations, Loss functions, Optimizers) that you have already implemented.

3.1.1 Autoencoder Implementation [6]

Create an **MLPAutoencoder** class using your existing MLP framework. This class will define the architecture of your autoencoder.

- **Encoder:** This part of the network will take an input image and compress it into a lower-dimensional latent representation (the "bottleneck").
- **Decoder:** This part will take the latent representation and attempt to reconstruct the original input image.

3.1.2 Training and Visualization on MNIST [6]

- **Training [3]:** Train your MLPAutoencoder on the MNIST training dataset. Your training loop should:
  1. Perform a forward pass to get the reconstructed output.
  2. Calculate the reconstruction loss using Mean Squared Error (MSE).
  3. Perform a backward pass to compute gradients for all parameters.
  4. Update the model's parameters using optimizer of your own choice.
- **Visualization [3]:** After training, visualize the performance of your autoencoder. For each digit class (0-9), display the original image alongside its reconstructed version from the test set.

3.2 Part 2: Anomaly Detection with Autoencoders [18]

Now, you will apply your autoencoder to an anomaly detection task on the **Labeled Faces in the Wild (LFW)** dataset. The core idea is to train the autoencoder to learn what the "normal" data looks like. When presented with "abnormal" or anomalous data, it should produce a significantly higher reconstruction error.

3.2.1 Anomaly Detection [8]

- **Training on Normal Data [3]:** For the "normal" class, we will use the class which has a significant number of images in the LFW dataset; this class is 'George_W_Bush'. This is to ensure sufficient learning of the autoencoder. Train your autoencoder exclusively on images from this normal class. The idea is that the autoencoder will learn

to reconstruct images of this class well but will have a higher reconstruction error for images from other classes ("anomalies").

- **Evaluation [5]:** Evaluate your trained autoencoder on the entire LFW test set. For each image, calculate the reconstruction error (MSE) and use it to classify images as normal or anomalous. You should:

    1. Determine a suitable threshold for the reconstruction error.
    2. Calculate and report the AUC Score, Precision, Recall, and F1-Score.

3.2.2 Analysis and Visualization [10]

- **Bottleneck Dimension Analysis [4]:** Investigate how the bottleneck dimension affects performance.

    1. Choose three different bottleneck dimensions.
    2. For each dimension, train a new autoencoder and evaluate its anomaly detection performance.
    3. Plot the ROC curves for all three models on a single graph and report their AUC scores.

- **Visualization [6]:** Provide a comprehensive visualization of your best model's results.

    1. Correct Classifications: Show an example of a correctly identified "normal" image (True Negative) and a correctly identified "anomaly" (True Positive), displaying the original, the reconstruction, and the reconstruction error for each.
    2. Misclassifications: Show an example of a misclassified "normal" image (False Positive) and a misclassified "anomaly" (False Negative), displaying the original, the reconstruction, and the reconstruction error for each.
    3. PR Curve: Plot the Precision-Recall (PR) curve for your best model.

Q1: Multi-Task CNN on Fashion-MNIST (75 Marks)

Objective

Convolutional Neural Networks (CNNs) are widely used for image-based tasks such as classification, regression, and object detection. In this assignment, you will implement a single **multi-task CNN** that jointly performs:

(a) **Classification** of Fashion-MNIST images into 10 clothing categories, and

(b) **Regression** of a continuous "ink" value corresponding to the normalized pixel intensity of each image.

The model will be trained using a joint loss:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{classification}} + \lambda_2 \mathcal{L}_{\text{regression}},$$

where $\mathcal{L}_{\text{classification}}$ is Cross-Entropy loss and $\mathcal{L}_{\text{regression}}$ is Mean Squared Error (MSE). You will explore the effect of λ_1 and λ_2 on the trade-off between classification and regression performance.

All experiments must be logged using Weights & Biases (wandb).

Dataset Description (Fashion-MNIST)

The Fashion-MNIST dataset (Zalando Research) contains 70,000 grayscale images (28×28) divided into 10 clothing classes:

```
0: T-shirt/top, 1: Trouser, 2: Pullover, 3: Dress, 4: Coat, 5: Sandal,
            6: Shirt, 7: Sneaker, 8: Bag, 9: Ankle boot.
```

Use the training split for model training and validation, and the provided test split for final evaluation.

Data Loading and Preprocessing (15 Marks)

1. **Dataset loading:** Implement a function `load_fashion_data()` that:
   - Loads the Fashion-MNIST dataset from Kaggle or torchvision,
   - Splits the original training set into `train` and `val` (e.g., 90/10 ratio),
   - Ensures no overlap or data leakage between splits.
2. **Custom Dataset class:** Implement `FashionMNISTDataset` that returns tuples `(image, class_label, ink_target)`:
   - **image:** normalized tensor using the mean and std of Fashion-MNIST,
   - **class_label:** the integer label $y \in \{0, \ldots, 9\}$,
   - **ink_target:** the normalized pixel intensity (average pixel value of the image), representing how much "ink" is used.

3. **Augmentations:** Use **light augmentations** (e.g., random crop, small rotation) for the training set only to encourage generalization. Keep validation and test sets unaltered.

Model Implementation: Multi-Task CNN (25 Marks)

1. **Architecture (7.5 Marks):** Implement a shared convolutional backbone with 2–3 Conv–BatchNorm–ReLU–Pool blocks and dropout. The model should then branch into two separate heads:
   - **Classification head:** outputs logits $[B, 10]$,
   - **Regression head:** outputs scalar values $[B, 1]$.
2. **Forward pass (7.5 Marks):** The `forward()` method should:
   - Pass inputs through shared convolutional layers,
   - Output both classification logits and regression predictions,
   - Optionally return intermediate feature maps for visualization.
3. **Joint loss (10 Marks):** Implement the total loss as:

$$\mathcal{L}_{\text{total}} = \lambda_1 \mathcal{L}_{\text{CE}} + \lambda_2 \mathcal{L}_{\text{MSE}}.$$

   Experiment with different values of λ_1 and λ_2 (e.g., 1:1, 2:1, 1:2, 0.5:1, etc.) to observe how they influence the model's performance on both tasks.

Hyperparameter Tuning and wandb Logging (20 Marks)

1. **Hyperparameters (7.5 Marks):** Explore key hyperparameters:
   - learning rate, dropout rate, optimizer choice (SGD, Adam, AdamW),
   - number of convolutional layers/filters, batch size, and
   - weighting factors λ_1 and λ_2.
2. **wandb Logging (Mandatory):** Each run must be logged on **Weights & Biases (wandb)**. Record:
   - Configuration (hyperparameters, λ_1, λ_2),
   - Training/validation loss curves (total, CE, MSE),
   - Validation metrics (accuracy, MAE, RMSE),
   - Final test performance and visualizations.
3. **Training runs (7.5 Marks):** Conduct at least **five distinct runs**, varying hyperparameters and λ_1, λ_2. Plot the training and validation losses for all runs (preferably via wandb line charts).

4. **Model selection (5 Marks):** Identify:

   - The model with the highest validation classification accuracy,
   - The model with the lowest validation regression RMSE.

   Report the corresponding test metrics and discuss how varying λ_1, λ_2 affected the trade-off between the two objectives.

Feature Map Visualization (15 Marks)

1. **Intermediate feature maps (7.5 Marks):** Modify `forward()` (or use hooks) to extract intermediate outputs after each convolutional block. Visualize feature maps from each layer for any three test images.

2. **Interpretation (7.5 Marks):** Describe the kinds of features captured at different layers (edges, textures, shapes, clothing regions) and how they help both classification and ink regression.

Deliverables

- **wandb Dashboard:** include a shareable project link in your report.
- **Report (PDF):** Describe your approach, hyperparameters, wandb plots, best models, and feature map visualizations. Discuss the observed trade-off between classification and regression performance as λ_1, λ_2 vary.
- **Metrics:**
  - Classification — Accuracy (and optionally F1-score)
  - Regression — MAE and RMSE

Q2: Image Colourization using CNNs (75 Marks)

Overview

In this task, you will implement and train a convolutional encoder–decoder network for **image colourization** on the CIFAR-10 dataset (32×32 RGB images). All code must be written in **PyTorch**, and you must log training, validation, and hyperparameter sweeps using **Weights & Biases (wandb)**.

Given a grayscale input image, your goal is to predict the colour of each pixel, classifying it into one of 24 colour clusters.

Colour Centroids and Data Preparation

To simplify the image colourization task, colour prediction is treated as a **classification** problem rather than regression. Each RGB pixel in CIFAR-10 is assigned to one of 24 representative clusters (colour centroids).

Provided: `color_centroids.npy` (shape: [24,3]) — 24 representative RGB cluster centroids computed by applying k-means ($k = 24$) over CIFAR-10 training pixels.

- Each pixel (x, y) is mapped to the nearest centroid $\mathbf{c}_i$:

$$\text{label}(x, y) = \arg \min_{i \in \{1,\ldots,24\}} \|\text{RGB}(x, y) - \mathbf{c}_i\|_2$$

- This produces a class map of size [32,32] with values in $\{0, \ldots, 23\}$.
- The model input is a grayscale image [1,32,32].
- Model output is logits per pixel [24,32,32].
- The loss is per-pixel cross-entropy.

Model Architecture: (20 pts)

In this architecture, the encoder is designed to **progressively reduce the spatial dimensions** of the input image through multiple convolutional and pooling layers. This deeper compression enables the network to capture more abstract and spatially invariant features. The decoder, on the other hand, reconstructs the colourized output image by performing **learned upsampling** using `nn.ConvTranspose2d` (transpose convolution) layers instead of simple interpolation. This approach allows the model to learn the most effective way to recover fine spatial details during reconstruction, encouraging students to understand how receptive fields expand in the encoder and how transpose convolutions can learn spatial mappings in the decoder.

Recommended block sequence

Use this architecture as the baseline. Experiments with kernel, NIC, NF.

```
Input [B, NIC, 32, 32]
Conv2d -> BN -> ReLU -> MaxPool2d   -> [B, NF, 16, 16]
Conv2d -> BN -> ReLU -> MaxPool2d   -> [B, 2*NF, 8, 8]
Conv2d -> BN -> ReLU -> MaxPool2d   -> [B, 4*NF, 4, 4]    # extra downsample
ConvTranspose2d -> BN -> ReLU       -> [B, 2*NF, 8, 8]   # learned upsample
ConvTranspose2d -> BN -> ReLU       -> [B, NF, 16, 16]
ConvTranspose2d -> BN -> ReLU       -> [B, NC, 32, 32]
Conv2d (classifier)                  -> [B, NC, 32, 32]
```

Architecture diagram

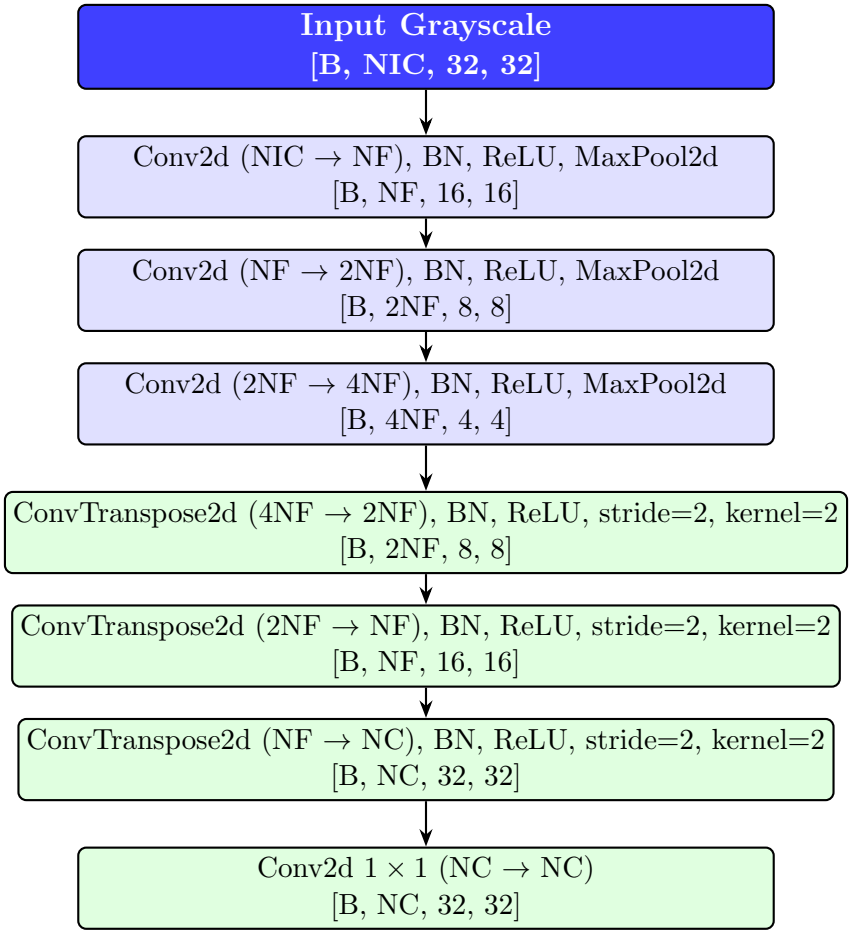


Figure 1: Model architecture

Training and Evaluation (20 pts)

- Dataset: CIFAR-10. Use supplied `color_centroids.npy` to map RGB images to 24-class labels; input is grayscale.

- Loss: per-pixel cross-entropy.
- Optimizer: SGD or Adam (log choice and hyperparameters in wandb).
- Train for at least 25 epochs; save and log the best checkpoint (lowest validation loss).
- At the end, produce:
    1. 10 example colourizations (input gray, predicted colourized, ground-truth colour).
    2. Training and validation loss curves.

Analytical Questions (10 pts)

For your `Model Architecture` compute (symbolically in terms of NIC, NF, NC):

1. Number of weights (ignore biases and BatchNorm parameters).
2. Number of outputs (total activation elements across all layers for a single input).
3. Number of connections (sum over layers of weight count times number of input activation elements each weight connects to).

Repeat the three quantities for input spatial size doubled (64×64). Show each step and state assumptions (padding/stride/kernels).

Hyperparameter Tuning (25 pts)

Perform a wandb sweep exploring at least 20 configurations. Tune at minimum:

- Learning rate: {1e-4, 3e-4, 1e-3, 3e-3}.
- Batch size: $\{32, 64, 128\}$.
- Number of filters (NF): $\{8, 16, 32\}$.
- Kernel size: $\{3, 5\}$ (adjust padding).
- Optimizer: SGD / Adam.

For each run log:

- Hyperparameters, train/val loss curves, best validation loss, example images, and best checkpoint.
- After the sweep, report the best run (hyperparameters and validation loss) and include the wandb run URL in your writeup.

Q3: Trees and Random Forests (75 Marks)

Overview

In this question, you will implement and analyze decision trees and random forests for a sentiment classification task. You will complete coding tasks to build a decision tree from scratch, use Scikit-Learn to perform hyperparameter tuning, and interpret the learned models.

Dataset: Bag-of-words for Amazon Product Reviews

We will use a text sentiment classification task where each example is a plain-text online review from amazon.com. Our goal is to predict whether the sentiment is positive or negative from the text. The data has been preprocessed into a bag-of-words representation using a fixed vocabulary of 7729 terms.

- **Feature vector** $x_n \in \mathbb{R}^F$: A binary vector of size $F = 7729$, indicating which vocabulary terms are present in the review.
- **Label** $y_n \in \{0, 1\}$: A binary label indicating negative ($y_n = 0$) or positive ($y_n = 1$) sentiment.

The starter code repository includes a training set (6346 documents), a validation set (792 documents), and a test set (793 documents).

Part 1: Decision Tree Implementation from Scratch (20 Marks)

In this problem, you'll complete a from-scratch implementation of a Decision Tree. The starter code for this assignment can be found in the repository provided.

- **Code Task A:** Implement `predict` for a `LeafNode`. See the `LeafNode` class within starter code file: `tree_utils.py`.
- **Code Task B:** Implement `predict` for an `InternalDecisionNode`. See the `InternalDecisionNode` class within starter code file: `tree_utils.py`.
- **Code Task C:** Implement function to select the best binary split. See the starter code file: `select_best_binary_split.py`.
- **Code Task D:** Implement function to train the tree. See the starter code file: `train_tree.py`.

Part 2: Decision Trees for Review Classification (25 Marks)

We'll now examine decision trees for our sentiment classification problem using Scikit-Learn.

2.1 Train a Simple Tree and Visualize (10 Marks)

- Train a `DecisionTreeClassifier` with `criterion='gini'`, `max_depth=3`, `min_samples_leaf=1`, `min_samples_split=2` and `random_state=101`.
- Show the ASCII-text representation of your trained tree by calling the helper function `pretty_print_sklearn`.
- Is there any internal node that has two child leaf nodes corresponding to the same sentiment class? Why would having two children predict the same class make sense?

2.2 Hyperparameter Tuning (15 Marks)

- Perform a grid search for the hyperparameters of `DecisionTreeClassifier` over the following settings. Use `sklearn.model_selection.GridSearchCV` with `scoring = 'balanced_accuracy'`, the provided validation split (`cv=my_splitter`), `return_train_score=True`, and `refit=False`.
  - `max_depth` in `[2, 8, 32, 128]`
  - `min_samples_leaf` in `[1, 3, 9]`
  - `random_state` in `[101]`
- What are the values of `max_depth` and `min_samples_leaf` for the best tree found by the grid search?

Part 3: Random Forests for Review Classification (30 Marks)

We'll now see if an ensemble of decision trees can improve performance.

3.1 Train a Simple Random Forest and Analyze Features (10 Marks)

- Train a `RandomForestClassifier` with `max_depth=3` and `n_estimators=100` (and other hyperparameters from the starter code).
- Access the `feature_importances_` attribute of your trained forest.
- Create a two-panel table. The left panel should list the top 10 vocabulary words with the highest feature importance. The right panel should list 10 randomly chosen terms with near-zero importance (importance ¡ 0.00001), and state how many such terms were eligible.

3.2 Hyperparameter Tuning for Random Forest (10 Marks)

- Perform a grid search for `RandomForestClassifier` over the following settings:
  - `max_features` in `[3, 10, 33, 100, 333]`
  - `max_depth` in `[16, 32]`

  - `min_samples_leaf` in `[1]`
  - `n_estimators` in `[100]`
  - `random_state` in `[101]`

- What is the value of `max_features` of your best forest?
- What is the maximum possible value for `max_features` for this dataset? Why is it beneficial to tune this hyperparameter?
- When fitting random forests, what is the primary tradeoff controlled by the `n_estimators` hyperparameter? Can you overfit by setting it to be too large? Why or why not?

3.3 Final Model Comparison (10 Marks)

- Create a table summarizing the balanced accuracy for all four models developed (Simple DT, Best DT, Simple RF, Best RF). The table should have columns for model type, number of trees, max depth, and balanced accuracy on the train, validation, and test sets.
- Summarize your conclusions.

Good luck with the assignment

1 Kernel Density Estimation for Foreground Detection [25 Marks]

Figure 1: Foreground Detection Using KDE

1.1 Image Pre-Processing [2 Marks]

You are provided with two images:

- A **background image** (without any foreground object)
- A **test image** (with a foreground object present)

Your tasks are as follows:

1. **Image Alignment and Resizing:** Convert both images to the same dimensions, balancing computational efficiency and image quality.
2. **Feature Extraction:** Extract a relevant feature set (e.g., RGB channels, grayscale intensity, or color histograms) to be used as input for the KDE model.

Provide a brief justification for your chosen preprocessing approach and feature representation.

1.2 Custom KDE Class Implementation [15 Marks]

You are required to implement a custom **Kernel Density Estimation (KDE)** model from scratch, using only the `numpy` library (no external statistical or machine learning libraries are permitted).

Class Methods

1. ***__init__***
   Initialize the class with the following parameters:

   - `kernel`: a string specifying the kernel type (e.g., "gaussian", "triangular", "uniform").
   - `bandwidth`: a positive float controlling the smoothness of the density estimate.
   - `data`: the training data (e.g., background image features).

2. ***fit(data)***
   Implement the fitting method to store or preprocess the background data. Include a *smart sampling strategy* (e.g., random subset selection or grid-based sampling) to reduce computational cost while maintaining the representational quality of the model.

3. ***predict(samples)***
   Implement a prediction method that accepts a set of sample points (e.g., pixels from the test image) and returns their estimated probability densities under the fitted model. The prediction should use the chosen kernel function and bandwidth parameter to compute:

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

   where n is the number of training samples and d is the feature dimensionality.

1.3 Foreground Detection [8 Marks]

Use your KDE implementation to perform foreground detection as follows:

1. Fit the KDE model on the **background image features**.
2. Predict the density values for each pixel (or feature vector) in the **test image**.
3. Classify pixels as foreground or background based on an appropriate probability density threshold.

Experiment with different values of **bandwidth** and **kernel type** to obtain the best segmentation results. Provide a brief explanation for why your chosen parameter combination achieves superior performance.

For each class, briefly describe the architecture, hyperparameters, and optimization strategy used. Report the validation performance of all models on identical train/validation splits.

2 Data-Driven Discovery of a Discrete-Time Recurrence [20 Marks]

2.1 Dataset and Problem Setup [3 Marks]

You are provided with multiple univariate sequences $\{x_k\}_{k=0}^{T}$ generated by an **unknown, time-invariant, autonomous, discrete-time mechanism**. The data are noisily observed but follow a fixed rule within each sequence. Your goals are to (i) build predictors of x_k from its history, (ii) *identify* a compact analytical recurrence consistent with the data (including its effective order), and (iii) analyze how well it fits to the data. You are suggested to use an RNN for this task. Dataset is at Dataset link.

Tasks

1. **Data splits:** Create chronological train/validation/test splits. Justify lengths.
2. **Supervised pairs:** Form $(\mathbf{h}_k \mapsto x_k)$ with history vectors of length p and explain how p is selected.
3. **Normalization:** Apply normalization based on the training set only; describe inverse transform.

Suggested Implementation Skeleton

- `data_prep.py`: load data, create time splits, and normalize.
- `model.py`: implement baseline predictors (e.g., Linear AR, MLP, or small RNN).
- `train.py`: train chosen models, tune p, and log results.
- `identify.py`: derive analytical recurrence F_θ consistent with learned mapping.
- `analyze.py`: plot residuals, and report test performance.

2.2 Sequence Prediction [6 Marks]

Train one or more predictors mapping a chosen history to the next value:

$$\hat{x}_k = g_\phi\big(x_{k-1}, x_{k-2}, \ldots, x_{k-p}\big),$$

and later use this to recover a *recurrent* analytical expression. (Hint : *recurrent*)

Suggested Deliverables

- Trained models and hyperparameter table.
- Prediction plots and validation curves.
- Short discussion on history length and model complexity.

2.3 Analytical Recurrence Identification [6 Marks]

Identify a closed-form recurrence

$$\hat{x}_k = F_\theta\big(x_{k-1}, x_{k-2}, \ldots, x_{k-\hat{p}}\big),$$

and compare residuals with black-box predictors.

2.4 Evaluation Criterion

- With MAE, MSE as your metrics, evaluate on single-step prediction with your DL model.
- Evaluate autoregressive generation, both with your DL model, and the analytical expression that you've fit to your data.
- Show variation in error with these models, with increasing forecasting length.

2.5 Parsimony and Stability [3 Marks]

Explore parameter count vs. performance. Select the simplest accurate model and present as to how well it fits to the data in comparison to other sweeps. What does this tell you about the dataset ? Is there any temporal relation you can extract from the dataset.

Report Checklist

- Model specifications, identified F_θ, parameter estimates.
- Complexity–accuracy trade-off figure.
- Stability plots and conclusions.

---

3 Time Series Forecasting: Cumulative GitHub Stars [20 Marks]

3.1 Dataset Description [3 Marks]

You are provided with M GitHub repositories, each represented by a time series of cumulative star counts:

$$y_t^{(i)} = \text{total stars for repository } i \text{ up to time } t.$$

The task is to forecast future growth trajectories and compare classical vs. deep learning methods. The dataset is present at Dataset Link.

Provided Files

- `stars_data.csv` — timestamps, repository IDs, and star counts.
- `repo_metadata.json` — optional meta-features (language, topic, etc.).

3.2 Single Repository Evaluation:

Choose 2 of the repositories present in the dataset, and perform all of the below tasks for the repositories you have chosen.

3.2.1 Preprocessing and Feature Engineering [4 Marks]

1. Clean and align time series; handle missing points or jumps.
2. Choose whether to work in cumulative or incremental domain:

$$\Delta y_t = y_t - y_{t-1}.$$

3. Look at scaling of your data.
4. Ensure temporal integrity—no leakage from future data.
5. Work in different time domains, visualize your data and look as to how the number of stars progress in the dataset. Consider the differences, creating your train-test split would make, based on how you split your forecasting data.

Suggested Implementation Skeleton

- `prep_stars.py`: data cleaning, feature creation, normalization.
- `split_repos.py`: chronological and repository-level splits.

3.2.2 Forecasting Models [7 Marks]

Fit and compare:

- **Classical:** ARMA.
- **Deep Learning:** small RNN and 1D CNN forecasters
- Note: Explain how youre preparing your data for each of these models. Justify your choice in loss function for this dataset.

Suggested Implementation Skeleton

- `classical.py`: wrapper around statsmodels ARMA/ARIMA.
- `dl_models.py`: simple PyTorch/TensorFlow RNN and CNN implementations.
- `train_models.py`:Training loop/calls for all models and hyperparameter search.

3.2.3 Evaluation Protocol [4 Marks]

Metrics and Evaluation

Evaluate model performance using standard error metrics such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

- For **single-timestep forecasting**, compute these metrics at each prediction horizon to quantify immediate predictive accuracy.
- For **multi-step (increasing-window) forecasting**, autoregressively generate future predictions over extended horizons (e.g., $h = 1, 3, 7, 14, \dots$), and plot the mean prediction error as a function of forecast length to analyze how accuracy varies with longer horizons.

Suggested Deliverables

- `evaluate.py`: backtesting, metric computation, and plotting.
- Summary table of metrics across models and horizons.
- Representative forecast and calibration plots.

3.3 Report and Reproducibility [[2 Marks]

Submit a concise report including:

- Data preparation summary and transformations.
- Model architectures and tuning results.
- Quantitative metrics, forecast visuals, and generalization results.

Ensure full reproducibility with all your scripts.

Marking Rubric (Summary)

- Data preprocessing and handling: 4
- Model implementation and comparison: 7
- Evaluation protocol and metrics: 4
- Clarity and reproducibility: 3
- Report polish and visualization quality: 2

4 Variational Autoencoder (VAE) [35 Marks]

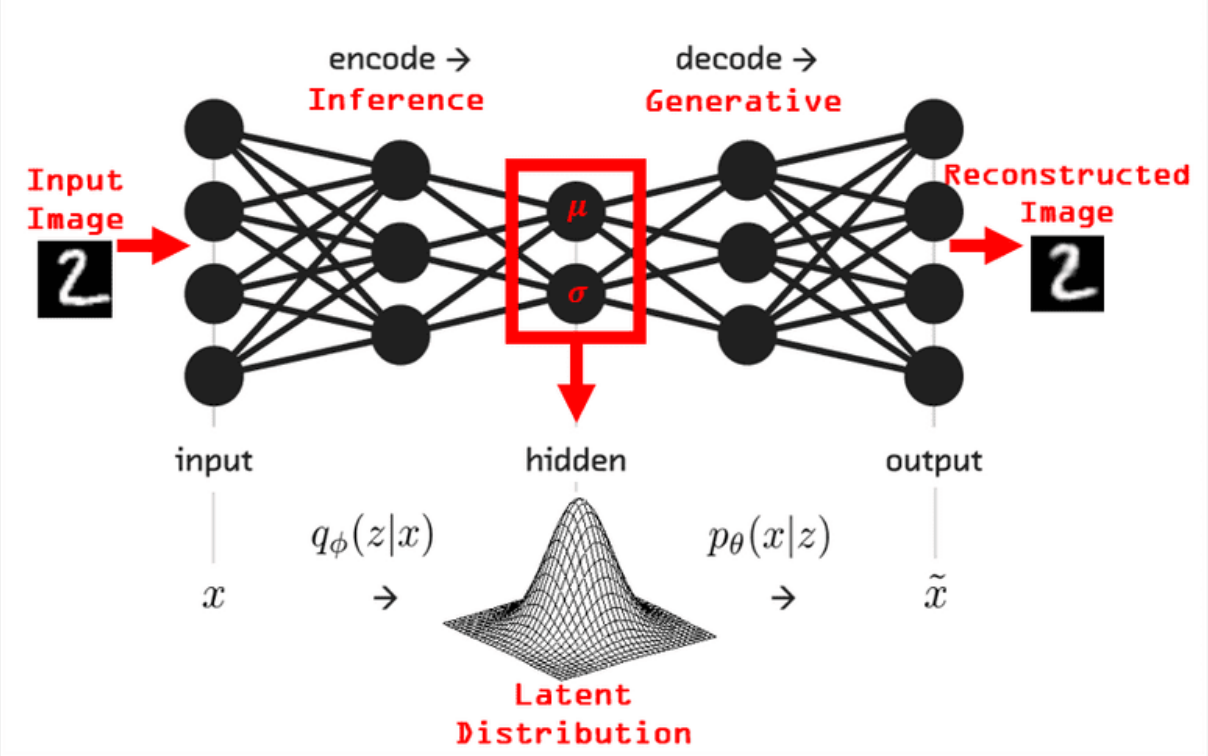


Figure 2: Variational Autoencoder Architecture

4.1 Introduction

A Variational Autoencoder (VAE) is a generative deep learning model that learns to encode data into a continuous latent space and decode it back to reconstruct the input. In this task, you will implement and analyze a VAE trained on the **Fashion-MNIST** dataset. You are allowed to use external libraries for this question.

4.2 Dataset Preparation [2 Marks]

Load and preprocess the **Fashion-MNIST** dataset using PyTorch utilities.

- Normalize the data and create DataLoader instances for both training and testing.
- Choose suitable batch size and normalization strategy.

4.3 Model Architecture [10 Marks]

Design and implement a Variational Autoencoder consisting of:

1. **Encoder:** A neural network mapping the input image to a latent representation defined by a mean vector μ and a log-variance vector $\log(\sigma^2)$.
2. **Reparameterization Trick:** Use $z = \mu + \epsilon \cdot \sigma$ with $\epsilon \sim \mathcal{N}(0, I)$ to enable stochastic sampling with gradient backpropagation.
3. **Decoder:** A neural network reconstructing images from latent samples.

Experiment with different latent dimensions and network depths, and discuss how these changes influence the smoothness of the latent space and reconstruction fidelity.

4.4 Loss Function [4 Marks]

The total loss for the VAE is a combination of:

- **Reconstruction Loss:** Measures pixel-wise similarity between input and reconstruction using Binary Cross-Entropy (BCE).
- **KL Divergence:** Encourages the latent distribution to approximate a standard normal prior.

Formally, the objective function is given by:

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - D_{KL}(q(z|x) \,||\, p(z))$$

Explain the role of each term in shaping the learned latent representation.

4.5 Training Procedure [2 Marks]

Train your model using an optimizer of your choice (e.g., Adam).

- Plot the total loss and its components over epochs.

4.6 Experimental Analysis [9 Marks]

In this section, you will conduct systematic experiments to understand the influence of key parameters on VAE performance.

- Vary the weighting between reconstruction loss and KL divergence using a β-parameter (as in β-VAE). Create a gif/video showing how the latent space evolves for values of $\beta = [0.1, 0.5, 1]$ for 3 classes of your choosing, make sure to color-code the data points in the latent space according to their class to observe clustering in the latent space (An example video has been attached in the dataset folder). Discuss how increasing β affects image sharpness/quality, Diversity, Cluster separation etc.
- Summarize your findings in a short table comparing quantitative metrics (e.g., reconstruction loss) and qualitative observations.

4.7 Evaluation and Visualization [3 Marks]

- Display a set of original and reconstructed test images side-by-side.
- Generate new samples by sampling from $\mathcal{N}(0, I)$ in the latent space and decoding them.
- Interpret the quality, diversity, and realism of generated samples, measure the Fréchet Inception Distance (FID) to evaluate goodness of newly generated samples.

4.8 Effect of Frozen Latent Parameters [5 Marks]

Investigate the influence of fixed latent distribution parameters on generation quality.

- Freeze the mean μ as 0 and try $\sigma = [0.1, 0.5, 1]$
- Generate new samples by decoding these frozen latent vectors.
- Compare the generated samples to those obtained from the standard stochastic sampling ($z = \mu + \epsilon \cdot \sigma$).
- Evaluate and discuss the effect of freezing μ and/or σ on image diversity, sharpness, and representational smoothness in the latent space.