Statistical Methods in AI - Monsoon 2025

Assignment 3 [100 Marks]

Deadline : 14th October 2025 11:59 P.M.

Instructors: Prof Ravi Kiran Sarvadevabhatla, Prof Saikiran Bulusu

General Instructions

- Your assignment must be implemented in Python.
- Clearly label and organize your code, including comments that explain the purpose of each section and key steps in your implementation. Ensure that your files are well-structured, with headings, subheadings, and explanations as necessary.
- Make sure to test your code thoroughly before submission to avoid any runtime errors or unexpected behavior.
- Your assignment will be evaluated not only based on correctness but also on the quality of code, the clarity of explanations, and the extent to which you've understood and applied the concepts covered in the course.
- We are aware of the possibility of submitting the assignment late in GitHub Classroom using various hacks. Note that we will have measures in place accordingly and anyone caught attempting to do the same will be given a straight zero in the assignment.

AI Tools Usage Instructions (Mandatory if Applicable)

We are aware how easy it is to write code and solve questions with the help of LLM services, but we strongly encourage you to figure out the answers yourself. If you use any AI tools such as ChatGPT, Gemini, Claude, etc., to assist in solving any part of this assignment:

- **If you are unable to explain any part of the solution/code during evaluations, that solution/code will be considered plagiarized and you will be penalized.** You must also be able to briefly explain how you modified or verified the AI-generated content.
- You must include a **shareable link** to the AI Tool chat history or a **screenshot** of the relevant conversation. **Do NOT share any private or sensitive personal information** in the AI Tool conversations you include.

Submission Instructions

- Submit a self-contained **Jupyter notebook** containing all code, plots, and printed tables.
- Report all the analysis, comparison and any metrics in the notebook or a separate report that is part of the submission itself. No external links(Except for video for Q.4) to cloud storage files, wandb logs or any other alternate will be accepted as part of your submission. Only the values and visualizations as part of your commits will be graded.
- All plots must include your **email username** in the title or filename

```
plt.text(
    0.95, 0.95, "username",
    ha='right', va='top',
    transform=plt.gca().transAxes,
    fontsize=10, color='gray', alpha=0.7
)
```

Submission Policy (GitHub Classroom Assignments)

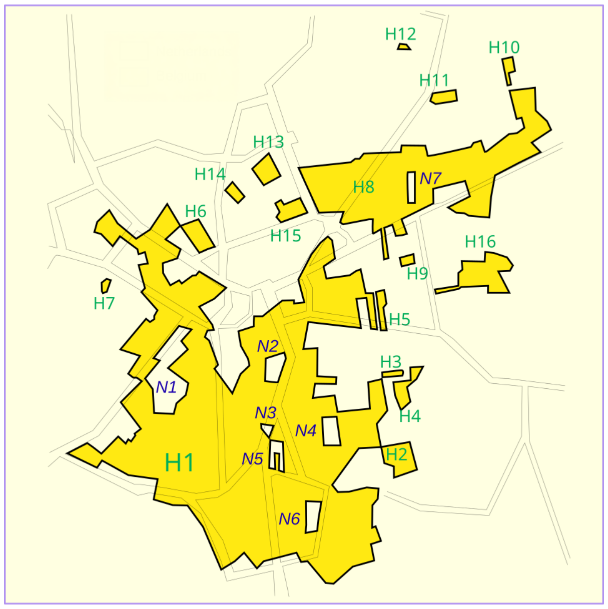
To encourage consistent progress and discourage last-minute submissions, the following policy applies:

- **Minimum Progress Requirement:** You must push at least **two meaningful commits on different days** prior to the deadline.
- **Commit Timing Check:** If over 80% of commits are made within the last 24 hours before the deadline, a **10% penalty** will be applied.
- **Commit Quality:** Commits must reflect actual progress. Non-informative or placeholder commits will not count.
- **Final Submission:** Your final grade is based on the latest commit before the deadline, but commit history will be reviewed.

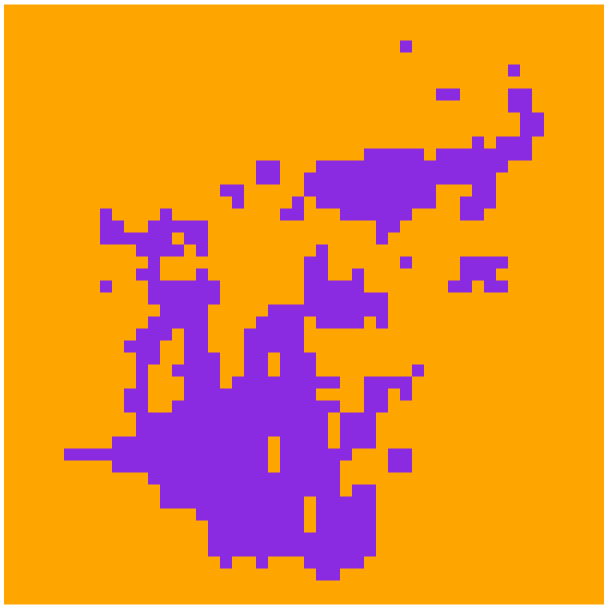
Guidelines for Implementation and Code Design

1. Use object-oriented programming.
2. Dataset for the complete assignment can be found **here**
3. 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**.
4. Keep visualization and computation logic in **separate functions**.
5. Add `docstrings` to methods explaining what they do.

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.

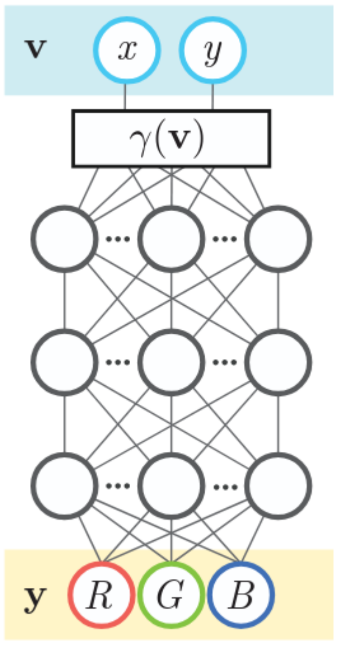


Figure 2: Illustration of an MLP with Fourier feature mapping applied to 2D input coordinates (x, y).

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.