Data Preprocessing

Contents

I. Introduction

In the paradigm of neural network-based classification, the structural integrity of input data serves as a primary determinant of model efficacy. Raw datasets are inherently prone to redundancies and latent noise, which frequently impede the convergence of the learning process. Consequently, **data preprocessing** is employed as a foundational pipeline to filter irrelevant information, thereby accelerating stochastic gradient descent and mitigating the computational ambiguity induced by data artifacts. While contemporary literature occasionally conflates data preprocessing with **feature engineering**, a rigorous conceptual distinction is essential for methodological clarity:

Data preprocessing focuses on the remediation of raw, imperfect data to ensure structural consistency and interpretability.

Feature engineering is defined as an iterative refinement process aimed at synthesizing optimal feature representations to maximize predictive throughput.

II. Data Feeding Mechanisms: Data Loader, Batch Size, and Epoch

To efficiently train a neural network with thousands of parameters, such as the 25,000-parameter model previously discussed, we must systematically manage how data is presented to the algorithm. This is handled through three core concepts:

1. **Batch Size (Hyperparameter):** Due to computational memory limits, we cannot feed the entire dataset into the model at once. Instead, the data is divided into smaller subsets called **batches**. The Batch Size determines how many samples the model processes before updating its internal parameters (weights and biases). A smaller batch size provides a more stochastic (random) gradient update, which can help escape local minima, while a larger batch size offers more stable gradient estimates but requires more memory.
2. **Epoch:** An Epoch is defined as one complete pass of the entire training dataset through the neural network. Training for multiple epochs allows the model to iteratively refine its parameters. However, the number of epochs is a critical hyperparameter: too few may lead to **underfitting** (the model hasn't learned enough), while too many will likely cause **overfitting** as the model starts memorizing the training noise (as seen in Figure 4-8).
3. **Data Loader:** The Data Loader is the programmatic engine that manages the flow of data. Its primary responsibilities include:
   - **Batching:** Grouping individual samples into batches based on the specified Batch Size.
   - **Shuffling:** Randomizing the order of data in each epoch to ensure the model does not learn the sequence of the dataset, which improves generalization.
   - **Multiprocessing:** Pre-fetching data in parallel to ensure the GPU/CPU is never idling, thus optimizing training speed.

III. Normalization and Standardization

Definition of Normalization and Normalization techniques

Normalization is formally defined as the process of adjusting measured values from disparate scales into a common standardized range, typically $[0, 1]$ or $[-1, 1]$. In the context of deep learning and neural network architectures, it serves as a critical data preprocessing step designed to rescale numerical features without distorting the underlying distribution or losing intrinsic information. Fundamentally, techniques such as Min-Max Scaling apply a linear transformation to the dataset. Because the transformation is strictly linear, the proportional distances between data points are preserved, ensuring that no latent patterns or structural configurations are compromised during the preprocessing phase.

The necessity of normalization primarily stems from the algorithmic vulnerability to **magnitude dominance**. Many machine learning algorithms—particularly distance-based models and gradient-based optimization frameworks such as Neural Networks—are highly scale-sensitive. If one feature spans thousands of units while another ranges strictly between 0 and 1, the algorithm implicitly assigns disproportionate computational weight to the larger-scale feature. Normalization forces all features onto an equal mathematical footing, compelling the model to evaluate feature importance based on actual statistical variance rather than arbitrary units of measurement.

To demonstrate the practical impact, consider a hypothetical housing price prediction model utilizing two distinct features: property area (ranging from 50 to 200 square meters) and the number of bedrooms (ranging from 1 to 5). Without scaling, a marginal change of 10 square meters will generate a significantly larger gradient update than the addition of a single bedroom, despite the latter potentially holding greater predictive value. By applying Min-Max Scaling, the data is projected onto a $[0, 1]$ interval using the following mathematical formulation:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

For a specific instance of a house with an area of 125 square meters and 3 bedrooms, the normalized area is calculated as $\frac{125-50}{200-50} = 0.5$, and the normalized bedroom count is $\frac{3-1}{5-1} = 0.5$. Post-normalization, both

features yield a value of 0.5, representing the exact statistical midpoint of their respective original ranges. This intervention ensures the neural architecture processes these features equitably, effectively neutralizing the computational bias previously introduced by their disparate measurement scales.

Beyond the preliminary preprocessing of input data, modern deep learning architectures frequently incorporate internal normalization layers to stabilize hidden state dynamics and accelerate optimization. While input normalization standardizes the raw dataset, these internal mechanisms operate directly on the intermediate feature maps within the network's topology:

Batch Normalization : Computes mean and variance across the mini-batch dimension. This intervention smooths the loss landscape and mitigates internal covariate shift, fundamentally stabilizing the training trajectory.

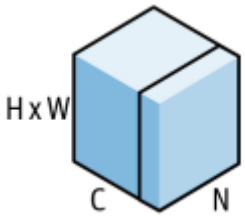

Figure 1: Batch Normalization explained

Before exploring Batch Normalization, it is essential to define a **mini-batch**. When training neural networks, computing weight updates using the entire dataset at once is computationally prohibitive. Conversely, updating weights after every single data sample introduces excessive instability.

A **mini-batch** is a small data subset (e.g., 32 samples) that optimally balances parallel processing efficiency with the statistical noise required to escape local minima.

During training, deep networks suffer from **internal covariate shift**—akin to a tower of blocks collapsing when its base shifts. As lower layers update their weights, their output distributions constantly fluctuate. This forces upper layers to perpetually adapt to shifting inputs rather than learning the target task, severely bottlenecking convergence.

To counteract this, Batch Normalization stabilizes the network by standardizing intermediate inputs at every hidden layer. For each mini-batch, the architecture performs three streamlined operations:

1. **Interception:** Captures the raw outputs (logits) immediately before they pass through the non-linear activation function.
2. **Standardization:** Normalizes these values across the current mini-batch by subtracting the batch mean and dividing by the standard deviation.
3. **Affine Transformation:** Restores the network's capacity to learn complex patterns by applying a linear transformation to the normalized inputs $\hat{x}$, using two trainable parameters: scale (γ) and shift (β).

Ultimately, by applying the operation $y = \gamma\hat{x} + \beta$, Batch Normalization effectively neutralizes internal distribution shifts. It acts as the structural "mortar" that secures the shifting blocks of the neural network, enabling faster and highly stable convergence. This information can be found in [1, page 137]

The following Python script demonstrates the practical application of Batch Normalization using the PyTorch framework:

```
import torch
import torch.nn as nn
def demo():
    torch.manual_seed(0)
    dummy_input = torch.randn(5, 3) * 10 + 5
    print(dummy_input)
    print("Before Batch Norm:")
    print("Mean:\n", dummy_input.mean(dim=0))
    print("Variance:\n", dummy_input.var(dim=0, unbiased=False))
    batch_norm = nn.BatchNorm1d(num_features=3)
    output = batch_norm(dummy_input)
    print("After Batch Norm:")
    print("Mean:\n", output.mean(dim=0).detach())
    print("Variance:\n", output.var(dim=0,
unbiased=False).detach())
if __name__ == '__main__':
    demo()
```

To ground the mathematical operations in a real-world context, we can interpret the input matrix X as a snapshot of a credit scoring system. In

this scenario, each row represents an individual loan applicant, while each column corresponds to a specific financial metric.

$$X = \begin{pmatrix} 20.4100 & 2.0657 & -16.7879 \\ 10.6843 & -5.8452 & -8.9860 \\ 9.0335 & 13.3803 & -2.1926 \\ 0.9666 & -0.9664 & 6.8204 \\ -3.5667 & 16.0060 & -5.7119 \end{pmatrix}$$

Before Batch Normalizing:

Mean: $(7.5055 \quad 4.9281 \quad -5.3716)$

Variance: $(68.8634 \quad 70.6324 \quad 60.4523)$

After Batch Normalizing:

Mean: $(3.5763e{-}08 \quad -1.1921e{-}08 \quad -4.7684e{-}08)$

Variance: $(1.0000 \quad 1.0000 \quad 1.0000)$

Before normalization, the disparate means (7.50 to -5.37) and high variances ($\approx$70) represent a "chaotic" state where features like Income dominate due to their larger scale, biasing the model's learning process. After applying Batch Normalization, the means are re-centered to ≈ 0 and variances scaled to 1.0. This standardization ensures all financial metrics contribute equitably to the weight updates, transforming inconsistent raw data into a stable format that facilitates rapid model convergence.

However, Batch Normalization still has some drawbacks. Specifically, if the batch size is large, computations will consume a large amount of memory; or if the batch size is really small, the performance will greatly decrease as the estimates of mean and variance are no longer accurate. Because of that, different normalization methods have been introduced as alternatives to Batch Normalization, especially Group Normalization.

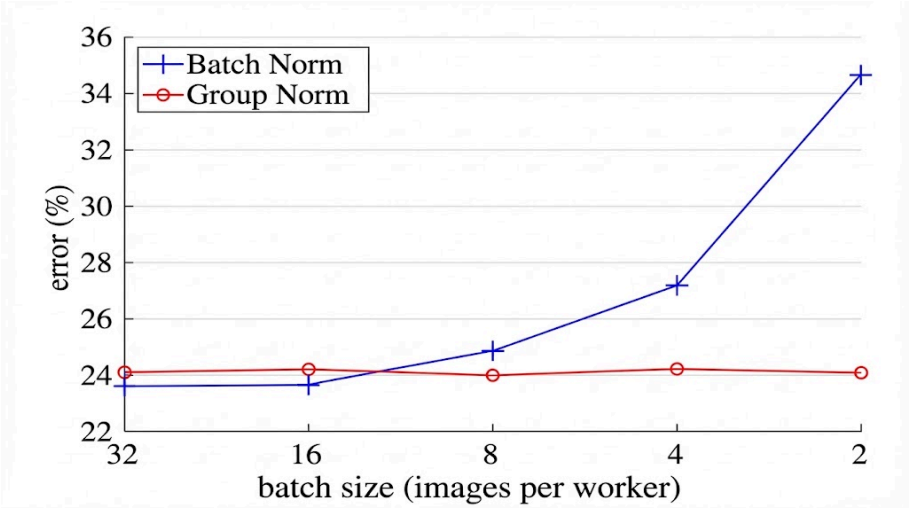

Figure 2: Batch Norm vs. Group Norm Error Rates

Group Normalization: Partitions feature channels into localized groups for statistical computation. This mechanism serves as a robust alternative to Batch Normalization in scenarios where hardware constraints dictate the use of strictly limited or micro-batch sizes.

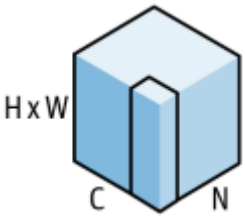

Figure 3: Group Norm explained

Group Normalization elegantly bypasses this limitation by shifting the normalization axis. Instead of normalizing across the batch dimension, it processes each data instance independently, making its stability entirely invariant to batch size.

For a single input image, Group Normalization performs three streamlined operations:

1. **Channel Grouping:** Divides the total number of feature channels into predefined, equal-sized groups.
2. **Standardization:** Computes the mean (μ) and variance (σ^2) independently within each specific group (across its spatial dimensions) and standardizes those local values:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Where,

$\hat{x}$ is the normalized value.
x is the original input value.
μ is the mean.
σ^2 is the variance.

1. **Affine Transformation:** Restores the network's representational capacity by applying a linear transformation to the normalized inputs $\hat{x}$, using channel-specific trainable parameters: scale (γ) and shift (β):

$$y = \gamma\hat{x} + \beta$$

Where,

y is the final output.
γ is the scale parameter.
$\hat{x}$ is the normalized input.
β is the shift/offset parameter

Ultimately, by applying the operation $y = \gamma\hat{x} + \beta$, Group Normalization acts as a robust, flexible stabilizer. It allows different sets of features to maintain distinct distributions, ensuring highly stable convergence even in memory-constrained environments like high-resolution image processing. [1, page 139]

```
import torch
import torch.nn as nn
def demo():
    sample = torch.tensor([
        [20.4100, 2.0657, -16.7879],
        [10.6843, -5.8452, -8.9860],
        [9.0335, 13.3803, -2.1926],
        [0.9666, -0.9664, 6.8204],
        [-3.5667, 16.0060, -5.7119],
    ], dtype=torch.float32)
    print(sample)
    print("Before Group Norm")
    mean_before = sample.mean(dim=1)
    var_before = sample.var(dim=1, unbiased=False)
    print("Mean:\n", mean_before)
    print("Variance:\n", var_before)
    gn = nn.GroupNorm(num_groups=1, num_channels=3)
    output = gn(sample)
```

```
    print(output.detach())
    print("After Group Norm")
    print("Mean:\n", output.mean(dim=1).detach())
    print("Variance:\n", output.var(dim=1,
unbiased=False).detach())
if __name__ == '__main__':
    demo()
```

Due to the independent, row-wise operation inherent to Group Normalization, the computation axis **dim** is shifted to 1. This ensures that the statistical moments are derived exclusively from the internal feature channels of each individual sample.

Before Group Normalizing:

Mean: $(1.8959 \quad -1.3823 \quad 6.7404 \quad 2.2735 \quad 2.2425)$

Variance: $(230.6284 \quad 74.4455 \quad 43.0484 \quad 10.9597 \quad 95.4844)$

After Group Normalizing:

Mean: $(0.0000e{+}00 \quad 5.9605e{-}08 \quad 0.0000e{+}00 \quad 0.0000e{+}00 \quad 0.0000e{+}00)$

Variance: $(1.0000 \quad 1.0000 \quad 1.0000 \quad 1.0000 \quad 1.0000)$

Collectively, these internal structural interventions ensure consistent gradient propagation and prevent the saturation of activation functions in deep networks.

Z-score Normalization (Standardization)

Z-score normalization transforms data so that it has a mean of 0 and a standard deviation of 1. This process adjusts data values based on how far they deviate from the mean, measured in units of standard deviation.This information can be found in [2]

$$Z = \frac{X - \mu}{\sigma}$$

Where,

Z is the Z-score.
X is the value of the data point.
μ is the mean of the dataset.
σ is the standard deviation of the dataset.

Here is a sample code:

```python
import numpy as np
def construct():
    data = np.array([70, 80, 90, 100, 110], dtype=float)
    mean = np.mean(data)
    std_dev = np.std(data)
    z_scores = (data - mean) / std_dev
    print("Original data:", data)
    print("Mean:", mean)
    print("Standard Deviation:", std_dev)
    print("Z-scores:", z_scores)
if __name__ == "__main__":
    construct()
```

The outputs are:

Mean: 90.0

Standard deviation: 14.142135623730951

Z-scores: $(-1.41421356 \;\; -0.70710678 \;\; 0 \;\; 0.70710678 \;\; 1.41421356)$

To establish a concrete understanding of this transformation, consider a practical dataset representing a feature with varying magnitudes—for instance, the resistance load (in kilograms) utilized across five consecutive sets in a strength training regimen: $X = [70, 80, 90, 100, 110]$.

When this input vector is processed through Z-score normalization, the resulting values articulate a precise statistical narrative rather than raw scalar metrics:

1. **The Baseline ($Z = 0$):** The data point 90 maps exactly to a Z-score of 0. This indicates that 90 is the exact mean ($mu = 90.0$) of the dataset, representing the central tendency with zero deviation.
2. **Positive Deviations ($Z > 0$):** The values 100 and 110 map to 0.70 and 1.41, respectively. A Z-score of 1.41 mathematically signifies that the maximum load (110) is strictly 1.41 standard deviations ($\sigma \approx 14.14$) above the dataset's average.
3. **Negative Deviations ($Z < 0$):** The values 70 and 80 yield negative Z-scores (-1.41 and -0.70). The negative sign provides a computational directive that these instances fall below the mean.

In the context of Neural Networks, feeding a raw value such as 110 alongside a probabilistically scaled feature (e.g., an activation threshold bounded strictly between 0 and 1) would inherently induce severe

magnitude dominance. By mapping 110 to 1.41, Z-score normalization strips away the original unit of measurement (kilograms), projecting the data into a dimensionless, standardized scale.

This intervention ensures that the optimization algorithm, such as Stochastic Gradient Descent, processes all features equitably. Consequently, it yields a more symmetric loss landscape, accelerates algorithmic convergence, and prevents features with naturally larger numeric ranges from disproportionately influencing the network's weight updates.

IV. Steps in Data Preprocessing

Data Cleaning

Data cleaning is a critical preprocessing phase dedicated to identifying and rectifying anomalies, errors, and inconsistencies within a dataset. The primary objectives of this phase encompass the treatment of missing values, the elimination of redundant records, the standardization of erroneous data, and the management of outliers. To achieve optimal data integrity, several analytical techniques are commonly employed:

1. **Mean Imputation**: Replaces missing values with the average of the attribute.
2. **Median Imputation**: Replaces missing values with the middle value, useful when outliers exist.
3. **Mode Imputation**: Replaces missing values with the most frequent value.
4. **Deletion Method**: Removes records that contain missing values.
5. **Interquartile Range (IQR)**: Detects outliers using the range between Q1 and Q3.
6. **Z-Score Method**: Identifies outliers based on standard deviation from the mean.
7. **Binning**: Smooths noisy data by grouping values into bins.
8. **Regression Smoothing**: Uses regression to predict and smooth noisy values.
9. **Duplicate Detection**: Identifies and removes repeated records.

Example: Replacing missing age values with the average age

Removing repeated rows in a dataset

Data Integration

Data Integration Data integration is the systemic process of consolidating information from disparate sources into a cohesive, unified dataset. This preprocessing phase addresses the inherent complexities arising from heterogeneous data formats, structural variations, and semantic discrepancies across different source systems.

The core objectives and operational contexts of this phase include:

Data Source Aggregation: This process is primarily executed when synthesizing data extracted from diverse repositories, including relational databases, flat files, and Application Programming Interfaces (APIs).

Redundancy Mitigation: It focuses on identifying and systematically eliminating overlapping or duplicated records across multiple datasets, thereby optimizing computational efficiency and preventing model bias.

Conflict Resolution: It involves the reconciliation of data value conflicts —instances where the same real-world entity is represented by discordant values across different sources—to ensure logical consistency and structural integrity within the final aggregated dataset.

To execute effective data integration, several robust analytical methodologies are employed:

1. **Schema Matching**: Aligns attributes from different data sources.
2. **Entity Resolution**: Identifies records that refer to the same real-world entity.
3. **Correlation Analysis**: Finds and removes redundant attributes.
4. **Data Conflict Resolution**: Resolves inconsistencies in units or data values.
5. **Duplicate Elimination**: Removes overlapping records after integration.

Example: Merging customer data from sales and marketing databases.

Data Transformation

Data transformation is a pivotal preprocessing stage designed to reconfigure datasets into an optimized structure, thereby enhancing the efficacy and accuracy of subsequent data mining and machine learning algorithms.

The primary objectives of this phase include:

Format Standardization: Systematically converting disparate and heterogeneous data structures into a unified, consistent mathematical format.

Efficiency Optimization: Streamlining the dataset to accelerate computational processing times and improve the overall performance of analytical models.

Algorithmic Suitability: Tailoring the data representations to meet the specific mathematical assumptions and structural prerequisites of advanced modeling techniques.

To execute effective data transformation, various analytical and mathematical methodologies are employed:

1. **Min-Max Normalization**: Scales data into a fixed range, usually 0 to 1.
2. **Z-Score Normalization**: Transforms data using mean and standard deviation.
3. **Decimal Scaling**: Normalizes data by moving the decimal point.
4. **Log Transformation**: Reduces data skewness using logarithmic scaling.
5. **One-Hot Encoding**: Converts categories into binary columns.
6. **Label Encoding**: Assigns numeric labels to categorical values.
7. **Aggregation**: Combines detailed data into summarized form.

Example: Converting salary values into a fixed range (0–1)

Changing text labels like Male/Female into numeric values

Data Reduction

Data reduction is a strategic preprocessing mechanism aimed at compressing the overall volume of a dataset while rigorously preserving its core informational integrity and analytical value. This phase addresses the computational bottlenecks associated with high-dimensional datasets.

The primary objectives and benefits of this phase include:

Computational Acceleration: Significantly improving processing speed and minimizing model training times by reducing the data payload.

Storage Optimization: Decreasing the physical or cloud-based memory footprint required to archive the dataset.

Analytical Simplification: Streamlining the data topology, thereby facilitating more efficient and interpretable downstream modeling.

To execute effective data reduction, several advanced analytical and statistical methodologies are deployed:

1. **Principal Component Analysis (PCA)**: Reduces dimensions by projecting data onto principal components.
2. **Linear Discriminant Analysis (LDA)**: Reduces dimensions while maximizing class separation.
3. **Filter Methods**: Select features based on statistical measures.
4. **Wrapper Methods**: Select features using model performance.
5. **Embedded Methods**: Perform feature selection during model training.
6. **Simple Random Sampling**: Selects data points randomly from the dataset.
7. **Stratified Sampling**: Samples data proportionally from each class.

Benefits of Data Preprocessing: Improves data quality

Increases accuracy of mining results

Reduces errors in models

Makes data easier to understand

Advantages:

Improved Data Quality: Ensures data is clean, consistent, and reliable for analysis.

Better Model Performance: Reduces noise and irrelevant data, leading to more accurate predictions and insights.

Efficient Data Analysis: Streamlines data for faster and easier processing.

Enhanced Decision-Making: Provides clear and well-organized data for better business decisions.

Data Discretization

Data discretization is a specialized preprocessing technique designed to partition continuous numerical variables into a finite set of discrete, categorical intervals or bins. This transformation is pivotal for algorithms

that intrinsically require nominal or ordinal input spaces, such as Decision Trees or Naive Bayes classifiers, and it serves to reduce the cognitive load of continuous data structures.

Methodological Techniques:

1. **Equal-Width Binning**: A systematic partitioning method that divides the absolute range of a continuous variable into N intervals of uniform mathematical width.
2. **Equal-Frequency (Quantile) Binning**: An approach that segment boundaries such that each resulting bin contains an approximately identical number of discrete data points, effectively mitigating the distortion caused by highly skewed distributions.
3. **Entropy-Based Discretization**: A supervised, top-down algorithmic strategy that leverages specific class labels to evaluate and determine optimal split points. It achieves this by iteratively minimizing the informational entropy, thereby maximizing the information gain at each partition.

Illustrative Application: Transforming a continuous "Age" metric (e.g., 22, 45, 60) into distinct, ordinal brackets such as "Young Adult" (18-35), "Middle-Aged" (36-55), and "Senior" (>55) to enhance the interpretability of a demographic classification model. Information can be found in [3]

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA

data_main = {
    'CustomerID': [1, 2, 3, 4, 5],
    'Age': [25, 400, 22, 25, 45],
    'Income': [50000, 60000, -3, 50000, 80000]
}
df = pd.DataFrame(data_main)

print("Original Data")
print(df, "\n")

#Cleaning
df['Age'] = np.where((df['Age'] < 0) | (df['Age'] > 100), np.nan,
df['Age'])
```

```
df['Income'] = np.where(df['Income'] < 0, np.nan, df['Income'])
df['Age'] = df['Age'].fillna(df['Age'].mean())
df['Income'] = df['Income'].fillna(df['Income'].median())

print("After cleaning")
print(df, "\n")

#Integration
data_sales = {
    'CustomerID': [1, 2, 3, 4, 5],
    'Country': ['VN', 'US', 'JP', 'VN', 'JP'],
    'Total_Spent': [100, 250, 300, 150, 400]
}
df_sales = pd.DataFrame(data_sales)
df = pd.merge(df, df_sales, on='CustomerID', how='inner')

print("After integration (merge with sales data):")
print(df, "\n")

#Discretization
bins = [0, 18, 35, 55, 120]
labels = ['Child', 'Young Adult', 'Middle-Aged', 'Senior']
df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels)

print("After discretization (Age -> Age_Group):")
print(df[['Age', 'Age_Group']], "\n")

#Transformation
df = pd.get_dummies(df, columns=['Country', 'Age_Group'],
dtype=int)
scaler = MinMaxScaler()
df[['Income', 'Total_Spent']] = scaler.fit_transform(df[['Income',
'Total_Spent']])

print("After transformation (One-hot encoding và Min-Max
Scaling):")
print(df, "\n")

#Reduction
features = df.drop(columns=['CustomerID', 'Age'])
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(features)
df_final = pd.DataFrame(reduced_data, columns=['PC1', 'PC2'])
df_final['CustomerID'] = df['CustomerID']
```

```
print("After PCA reduction")
print(df_final)
```

Assume the original data input is:

Customer ID	Age	Income
1	25	50000
2	400	60000
3	22	−3
4	25	50000
5	45	80000

We define two strict constraints for data validity: age must not exceed 120, and income must be strictly greater than 0. Based on these conditions, the original dataset contains two invalid anomalies: an age of 400 and an income of −3.

To address these errors, the first step in the preprocessing pipeline is Data Cleaning. After applying statistical imputation techniques, the dataset is transformed into the following table:

Customer ID	Age	Income
1	25	50000
2	29.25	60000
3	22	55000
4	25	50000
5	45	80000

How are the new values (29.25 and 55000) computed?

The system computes these values by isolating the valid data points and applying specific statistical imputation methods:

1. **Computing Age (Mean Imputation):**
   The system identifies 400 as an invalid value and temporarily replaces it with a null value (`NaN`). To fill this missing gap, the algorithm calculates the arithmetic mean of the remaining valid ages (25, 22, 25, and 45). Mean Age $= \frac{25+22+25+45}{4} = 29.25$. Thus, the invalid age is seamlessly replaced with the mean value of 29.25.

2. **Computing Income (Median Imputation):**
   Similarly, the system flags -3 as an invalid value and replaces it with (`NaN`). For income data, the median (the exact middle value of a sorted dataset) is utilized instead of the mean to prevent extreme high or low values from skewing the distribution. First, the valid incomes are sorted in ascending order: 50000, 50000, 60000, and 80000. Since there is an even number of valid records (4), the median is calculated as the average of the two central numbers. Median Income $= \frac{50000+60000}{2} = 55000$ Consequently, the invalid income is replaced with the robust median value of 55000.

   The second step in the preprocessing pipeline is Data Integration. In real-world scenarios, data is often fragmented across multiple operational systems. Here, the cleaned demographic data (Age, Income) must be consolidated with transactional data originating from the sales department (Country, Total_Spent).

Assume the newly retrieved sales dataset contains the following records:

Customer ID	**Country**	**Total_Spent**
1	VN	100
2	US	250
3	JP	300
4	VN	150
5	JP	400

To synthesize this information, the system performs an **inner join** operation utilizing `CustomerID` as the common primary key. This relational

mapping ensures that each demographic profile is accurately matched and enriched with its corresponding purchasing behavior.

The resulting unified dataset seamlessly integrates all attributes without redundancy:

Customer ID	Age	Income	Country	Total_Spent
1	25.00	50000.0	VN	100
2	29.25	60000.0	US	250
3	22.00	55000.0	JP	300
4	25.00	50000.0	VN	150
5	45.00	80000.0	JP	400

The third phase of the preprocessing pipeline is Data Discretization. Continuous numerical variables can sometimes introduce unnecessary complexity or violate the assumptions of certain categorical algorithms. To mitigate this, the continuous `Age` metric is partitioned into discrete, ordinal intervals (bins).

Using the predefined boundaries `[0, 18, 35, 55, 120]`, the system maps the numerical ages to logical categorical labels: 'Child', 'Young Adult', 'Middle-Aged', and 'Senior'.

Age	Age_Group
25.00	Young Adult
29.25	Young Adult
22.00	Young Adult
25.00	Young Adult
45.00	Middle-Aged

Following discretization, the dataset undergoes Data Transformation to satisfy the mathematical prerequisites of machine learning algorithms. This step involves two primary techniques:

1. **One-Hot Encoding:** Machine learning models cannot natively process text. Therefore, categorical variables such as `Country` and `Age_Group` are binarized. The system creates orthogonal binary columns (0 or 1) for each category to prevent the algorithm from assuming false numerical hierarchies.
2. **Min-Max Scaling:** The numerical variables (`Income` and `Total_Spent`) inherently possess vastly different scales, which can bias distance-based algorithms. Min-Max scaling normalizes these values strictly within a $[0, 1]$ range.

The resulting transformed matrix expands to 11 columns to accommodate the newly encoded features:

Customer ID	**Age**	**...**	**Group_Middle-Aged**	**Group_Senior**
1	25.00	...	0	0
2	29.25	...	0	0
3	22.00	...	0	0
4	25.00	...	0	0
5	45.00	...	1	0

The final preprocessing step is Dimensionality Reduction. The transformation phase expanded the dataset to 11 dimensions, which can lead to computational inefficiencies and the "curse of dimensionality." To optimize the dataset, the system applies Principal Component Analysis (PCA).

First, redundant or non-predictive identifiers (such as `CustomerID` and the original `Age` column, which is now represented by `Age_Group`) are stripped from the mathematical computation. PCA then orthogonally transforms the remaining multi-dimensional feature space into two condensed components (PC1 and PC2) that capture the maximum variance of the original data.

The `CustomerID` is reattached post-computation to maintain record tracking. The ultimate, highly optimized dataset ready for predictive modeling is as follows:

Customer ID	PC1	PC2
1	−0.883649	−0.387280
2	−0.184276	0.970690
3	0.331353	0.101555
4	−0.819417	−0.365749
5	1.555989	−0.319216

V. Data Splitting: Training, Validation, and Testing Sets

The Complexity of Artificial Neural Networks

A fundamental challenge in utilizing Artificial Neural Networks (ANNs) is their inherent structural complexity and the massive number of parameters they require to function.

For instance, consider a basic feed-forward network designed to classify simple 28×28 pixel grayscale images from the MNIST dataset. Even with a highly modest architecture—consisting of two hidden layers (each containing 30 neurons) and a final 10-neuron softmax output layer—the model must calculate and optimize approximately 25,000 trainable parameters.

This sheer volume of parameters for such a small task can introduce significant computational and generalization issues. To clearly understand why an excessively high parameter count becomes problematic, we will examine the foundational toy example illustrated in Figure 4.

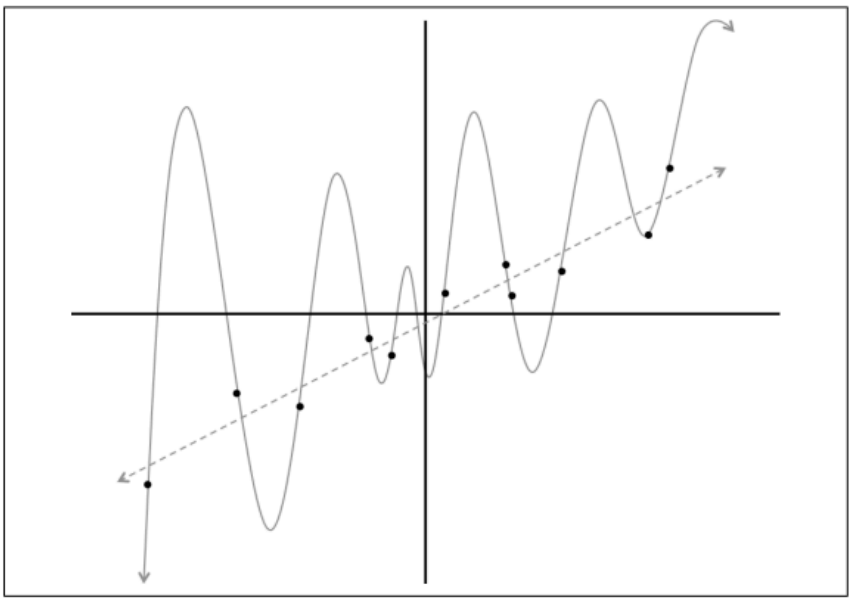

Figure 4: Two potential models that might describe our dataset: a linear model versus a degree 12 polynomial

Figure 4 visually illustrates the severe consequences of utilizing highly complex models (such as neural networks with vast parameter counts) on relatively simple datasets. This phenomenon is known as **Over-fitting**.

The graph displays three key components:

1. **Black Dots (Training Data):** These represent the underlying dataset, exhibiting a general positive linear trend along with minor natural variance (noise).
2. **Dashed Line (Simple Model):** This represents a low-complexity model (e.g., linear regression). While it does not intersect every specific data point perfectly, it successfully captures the fundamental, underlying trend of the data. This model will likely generalize well to new, unseen data.
3. **Solid Oscillating Curve (Complex Model):** This represents a highly parameterized neural network. Because the model has an excessive amount of parameters, it possesses the flexibility to perfectly memorize the training data, passing through every single dot.

The Danger of Complexity: The erratic, high-frequency oscillations of the solid curve demonstrate the core issue. By forcing itself to intersect every point, the complex model memorizes the random noise rather than learning the actual signal. Consequently, if a new data point is introduced between the existing ones, the complex model's prediction will wildly fluctuate, resulting in a catastrophic failure in generalization. This visually

explains why an excessive parameter count can severely degrade a model's predictive reliability in real-world applications. [1, page 65, 66]

How To Prevent Over-fitting: Data Splitting

The most fundamental strategy to prevent a model from over-fitting (like the oscillating curve in Figure 4-8) is to stop it from seeing all the data at once. By dividing the dataset into three distinct, isolated subsets, we can force the model to learn the actual trend rather than just memorizing the noise.

1. **Training Set (The Study Material):** This partition typically constitutes the largest segment of the dataset (e.g., 70% to 80%) and is the only data the algorithm directly "sees" and processes to learn. During this phase, the model takes inputs, outputs predictions, and calculates the error using a predefined Loss Function (such as Mean Squared Error or Cross-Entropy Loss). Through optimization algorithms like Gradient Descent and Backpropagation, the neural network iteratively adjusts its internal **parameters** (weights and biases) to minimize this error and discover underlying patterns. However, an over-reliance on this set without external monitoring inevitably leads to over-fitting.
2. **Validation Set (The Mock Exam):** Comprising roughly 10% to 15% of the data, this subset is kept hidden during the direct learning process but acts as an essential, independent feedback loop. Crucially, the model does **not** use this data to update its internal weights. Instead, developers evaluate the model on this set iteratively to tune **hyper-parameters**—the high-level architectural settings defined before training begins (e.g., learning rate or number of hidden layers). Furthermore, if the model's accuracy on the training set keeps rising but drops on the validation set, it is a clear signal of over-fitting (memorizing). Tracking this allows developers to pinpoint the exact epoch where over-fitting begins and halt training immediately (a technique called **Early Stopping**).
3. **Testing Set (The Final Exam):** This remaining 10% to 15% of the data is strictly locked away and quarantined until the entire training and hyper-parameter tuning process is finalized. Once the model is fully built, it is evaluated on this completely unseen data **exactly once**. If developers were to iteratively tweak the model based on testing results, information from the test set would inadvertently influence the model—

a critical flaw known as "Data Leakage." By keeping this set completely isolated, it provides an honest, objective, and unbiased estimate of how well the model will generalize in a real-world production environment.

Understanding Backpropagation

Backpropagation (short for "backward propagation of errors") is the fundamental algorithm used to compute the gradients of the loss function with respect to every weight and bias within an Artificial Neural Network.

Once the network makes a prediction (Forward Pass) and calculates its error, backpropagation applies the calculus **Chain Rule** to systematically propagate this error backward from the output layer down to the input layer. By calculating the partial derivatives, it precisely quantifies how much each individual parameter contributed to the total error. This crucial gradient information is then utilized by optimization algorithms, such as Gradient Descent, to update the weights and iteratively minimize the model's error. [4, page 171, 172]

Understanding Hyper-parameters

Most machine learning algorithms incorporate **hyper-parameters**, which function as external configuration settings that control the behavior and learning dynamics of the model. Unlike standard model parameters, hyper-parameters are not automatically learned or optimized by the training process itself; instead, they must be manually specified by the developer.

Key examples of hyper-parameters that dictate model performance include:

1. **Model Capacity:** In polynomial regression, the degree of the polynomial is a primary hyper-parameter. It directly determines the model's complexity—a higher degree allows for more intricate patterns but increases the risk of over-fitting.
2. **Regularization Strength (λ):** In weight decay methods, the λ value serves as a hyper-parameter to control the trade-off between minimizing training error and keeping weights small to prevent complexity.

While the algorithm cannot self-adjust these values during standard training, advanced workflows often employ nested learning loops where one optimization process is specifically designed to identify the most effective hyper-parameters for another.

VI. Conclusion

The transition from raw data to a high-performing neural network is not merely a matter of computational power, but a rigorous sequence of structural interventions. As explored in this report, **Data Preprocessing** serves as the essential bedrock, transforming "chaotic" datasets—characterized by disparate scales and noise—into standardized formats that gradient-based optimizers can effectively process. Techniques such as Normalization and Standardization (Z-score) ensure that no single feature disproportionately dominates the learning trajectory due to its numerical magnitude.

Furthermore, the strategic management of the data pipeline through **Data Loaders**, **Batching**, and **Data Splitting** (Train, Validation, and Test sets) provides the necessary framework to combat the "memorization" trap. While highly parameterized models, such as the 25,000-parameter MNIST network, possess the capacity to learn intricate patterns, they are inherently prone to over-fitting. By utilizing the Validation set to tune **Hyper-parameters** and monitoring performance benchmarks on the isolated Testing set, we ensure that the resulting model is not just a statistical artifact of its training environment, but a robust system capable of accurate generalization in real-world applications.

Ultimately, the synergy between meticulous data preparation and disciplined model evaluation defines the success of modern deep learning architectures.

VII. References

[1] N. Buduma, N. Buduma, and J. Papa, Fundamentals of Deep Learning. O'Reilly Media, 2022.

[2] GeeksforGeeks, "Z-Score Normalization: Definition and Examples." Accessed: July 23, 2025. [Online]. Available: https://www.geeksforgeeks.org/data-analysis/z-score-normalization-definition-and-examples/

[3] GeeksforGeeks, "Data Preprocessing in Data Mining." Accessed: Feb. 07, 2026. [Online]. Available: https://www.geeksforgeeks.org/data-science/data-preprocessing-in-data-mining/

[4] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning, translated by Nguyen Dinh Quy and Nguyen Khanh Chi. 2015.