EXPERIMENT-1

AIM: Classification using MLP on MNIST dataset for digit classification

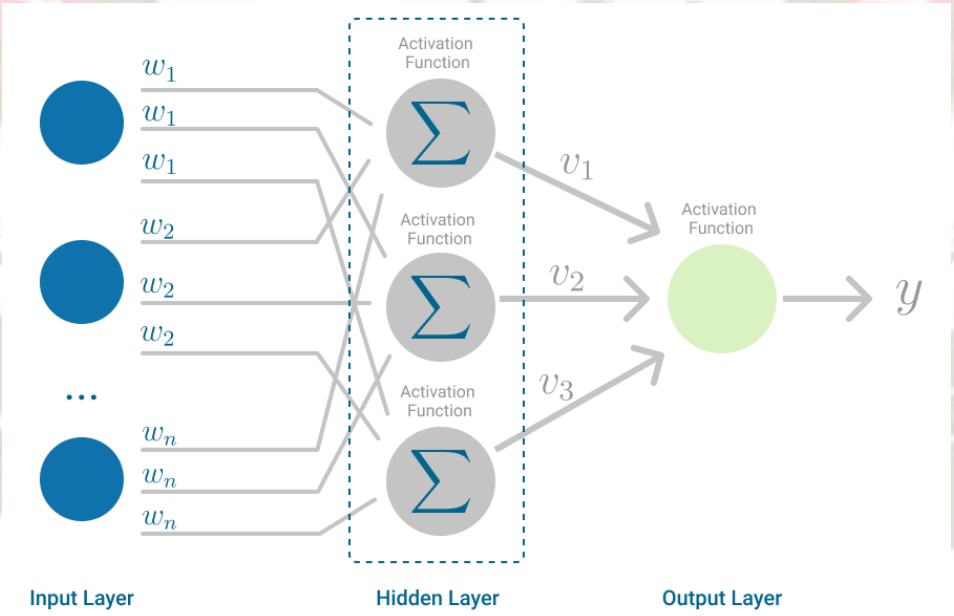
DESCRIPTION:

Introduction:

Multi-Layer Perceptron Learning:

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together sothat the outputs of some neurons are inputs of other neurons.

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any numberof hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



(reference: https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141)

Classification:

Classification is a process of categorizing data or objects into predefined classes or categories based on their features or attributes. Machine Learning classification is a type of

supervised learning technique where an algorithm is trained on a labeled dataset to predict the class or category of new, unseen data.

The main objective of classification machine learning is to build a model that can accurately assign a label or category to a new observation based on its features.

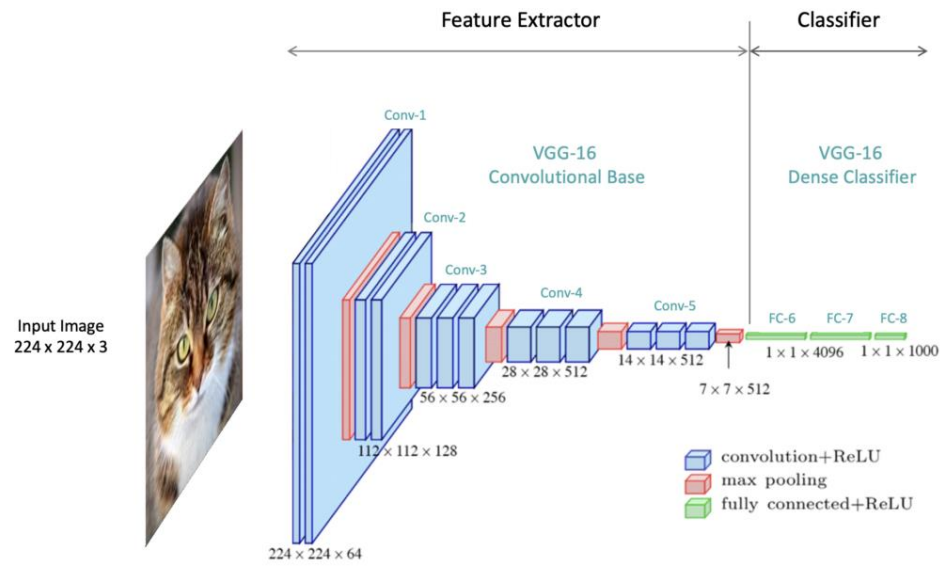
For example, a classification model might be trained on a dataset of images labeled as either dogs or cats and then used to predict the class of new, unseen images of dogs or cats based on their features such as color, texture, and shape.

About the MNIST dataset:

The MNIST (Modified National Institute of Standards and Technology) database is a large database of handwritten numbers or digits that are used for training various image processing systems. The dataset also widely used for training and testing in the field of machine learning. The set of images in the MNIST database are a combination of two of NIST's databases: Special Database 1 and Special Database 3.

The MNIST dataset has 60,000 training images and 10,000 testing images.

The MNIST dataset can be online, and it is essentially a database of various handwritten digits. The MNIST dataset has a large amount of data and is commonly used to demonstrate the real power of deep neural networks. Our brain and eyes work together to recognize any numbered image. Our mind is a potent tool, and it's capable of categorizing any image quickly. There are so many shapes of a number, and our mind can easily recognize these shapes and determine what number is it, but the same task is not simple for a computer to complete. There is only one way to do this, which is the use of deep neural network which allows us to train a computer to classify the handwritten digits effectively.



(reference: https://learnopencv.com/understanding-convolutional-neural-networks-cnn/)

So, we have only dealt with data which contains simple data points on a Cartesian coordinate system. From starting till now, we have distributed with binary class datasets. And when we use multiclass datasets, we will use the Softmax activation function is quite useful for classifying binary datasets. And it was quite effective in arranging values between 0 and 1. The sigmoid function is not effective for multicausal datasets, and for this purpose, we use the softmax activation function, which is capable of dealing with it.

Implementation of Classification with Multilayer Perceptron using Scikit-learn with MNIST Dataset

NumPy: NumPy stands for Numerical Python. It is one of the basic Python Library that is used for creating arrays, filling null values, statistical calculations and computations. Pandas is built on the top of the NumPy library.

Matplotlib: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python

CODE:

```
# Importing modules
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

# Loading dataset
```

```
X, y = load_digits(return_X_y=True)
y = y.astype(int)
X /= 255.0  # Normalizing gray image to floats

# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
print(len(X_train), len(X_test), len(y_train), len(y_test))  # Output:
(1437, 360, 1437, 360)

# Training the model
m = MLPClassifier(hidden_layer_sizes=(64, 32, 16), max_iter=100)
m.fit(X_train, y_train)

# Making inferences
y_pred = m.predict(X_test)
print(len(y_pred))  # Output: 360

# Evaluating the model
conf_matrix = confusion_matrix(y_test, y_pred)
print(conf_matrix)

accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

Ouput:

98.5

RESULT:

The result of this implementation is a trained Perceptron model that successfully finds a decision boundary, or straight line, that separates the two classes in the Iris dataset: Iris-setosaand Iris-versicolor. By iteratively adjusting the weights based on the training examples, the Perceptron algorithm converges, meaning it finds a set of weights that correctly classifies all the data points in this linearly separable dataset. The scatter plot visualization confirms that the two classes are distinctly separated by this boundary, demonstrating the effectiveness ofthe Perceptron in handling linearly separable data. This outcome showcases the algorithm's ability to learn and create a model that accurately distinguishes between the two flower species based on their features.

EXPERIMENT-2

AIM:

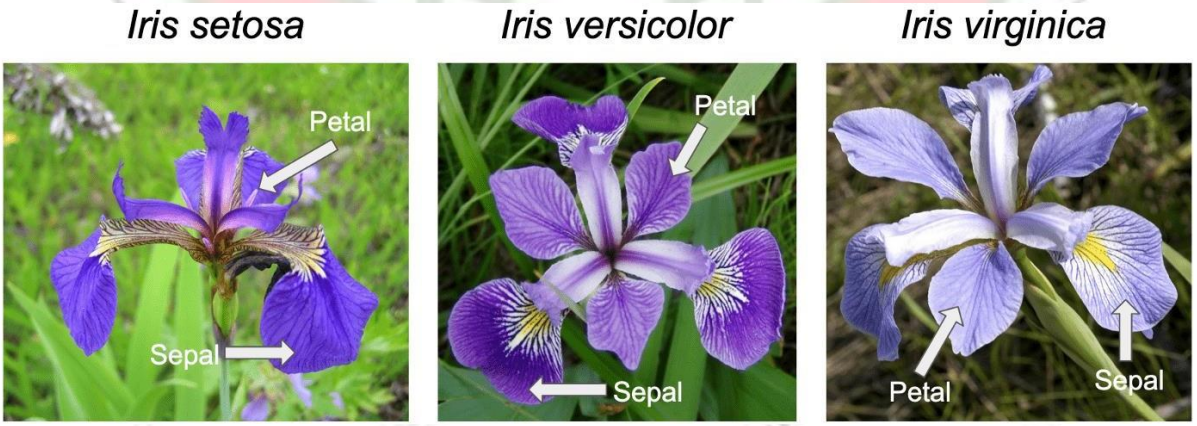
To implement and analyze the Perceptron learning algorithm for binary classification using a linearly separable subset of the Iris dataset, focusing on its ability to learn optimal weights and achieve accurate classification.

DESCRIPTION:

Introduction:

Perceptron Learning is one of the earliest and simplest forms of machine learning algorithms, designed for binary classification tasks. Developed by Frank Rosenblatt in 1958, the Perceptron is a foundational algorithm in the field of artificial intelligence, serving as the building block for more complex neural networks. The Perceptron works by learning a linear decision boundary, or hyperplane, that separates two classes in a dataset based on input features.

The Perceptron algorithm operates through an iterative process where it adjusts the weights assigned to each feature of the input data. Initially, the weights are set to zero or small random values. For each training example, the algorithm calculates a weighted sum of the input features and passes it through a step function to make a prediction. If the prediction matchesthe actual label, the weights remain unchanged. However, if the prediction is incorrect, the algorithm updates the weights by adding or subtracting a fraction of the input features based on the difference between the actual and predicted labels. This process continues for a predefined number of iterations or until the weights no longer change, indicating convergence.



(reference: https://towardsdatascience.com/the-iris-dataset-a-little-bit-of-history-and-biology-fb4812f5a7b5)

[dataset contains 4 features that describe the flower and classify them as belonging to one ofthe 3 classes.

We strip the last 50 rows of the dataset that belongs to the class 'Iris-virginica' and use only 2 classes 'Iris-setosa' and 'Iris-versicolor' because these classes are linearly separable and the algorithm converges to a local minimum by eventually finding the optimal weights.]

The Perceptron is particularly effective for linearly separable datasets, where it can successfully find a hyperplane that separates the classes. For example, in the classic Iris dataset, the Perceptron can be used to distinguish between the Iris-setosa and Iris-versicolorclasses based on features such as petal length and sepal length.

Importance:

The Perceptron Learning algorithm is significant not only as a historical milestone in AI but also as a practical tool for understanding the principles of linear classifiers. It forms the basis for more sophisticated models, such as multi-layer perceptrons and deep neural networks. By mastering the Perceptron, one gains insight into how more advanced machine learning models learn and adapt, making it an essential concept in the study of artificial intelligence and machine learning.

Procedure:

The provided program demonstrates the implementation of the Perceptron learning algorithm, a fundamental machine learning algorithm for binary classification. Here's abreakdown of what the program does:

1. Data Loading and Preprocessing:

The program starts by loading the Iris dataset from the UCI Machine Learning Repository. The dataset originally contains 150 samples with four features, describing three classes of iris flowers: Iris-setosa, Iris-versicolor, and Iris-virginica.

To simplify the problem and ensure that the data is linearly separable, the program strips thelast 50 rows corresponding to the Iris-virginica class. This leaves only two classes: Iris-setosa and Iris-versicolor.

The labels are then converted to binary values: 0 for Iris-setosa and 1 for Iris-versicolor.

2. Data Visualization:

A scatter plot is generated to visualize the dataset using two features: petal length and sepal length. The plot shows that the two classes can be clearly separated by a straight line, confirming that they are linearly separable.

3. Perceptron Algorithm Implementation:

The core of the program is the implementation of the Perceptron learning algorithm. The algorithm initializes the weights to zero and iteratively updates them based on the classification errors it encounters.

For each training example, the Perceptron calculates a weighted sum of the input features and predicts the class label based on whether the sum is greater than zero. If the prediction is incorrect, the weights are adjusted accordingly.

The algorithm runs for a specified number of iterations (`num_iter`), or until the weights stabilize, meaning the classification accuracy no longer improves.

4. Tracking Misclassifications:

The program keeps track of the number of misclassified examples in each iteration and plots this information to visualize the learning process. As the algorithm progresses, the number of misclassifications decreases, indicating that the Perceptron is learning the optimal weights to separate the two classes.

Important Notes:

The program highlights that a single-layer Perceptron can only solve problems where the datais linearly separable.

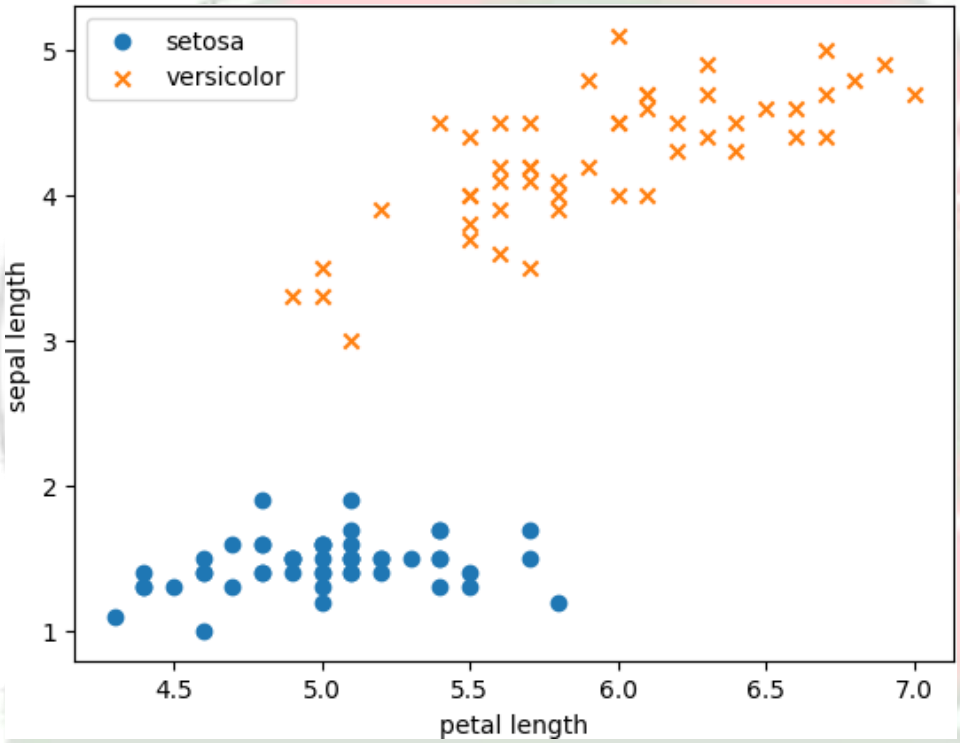
It also mentions that while the Perceptron is designed for binary classification, it can be extended to multiclass classification by using one Perceptron per class.

Overall, the program provides a clear demonstration of how the Perceptron algorithm works, from data preprocessing to visualizing the learning process, emphasizing its application in binary classification tasks.

CODE:

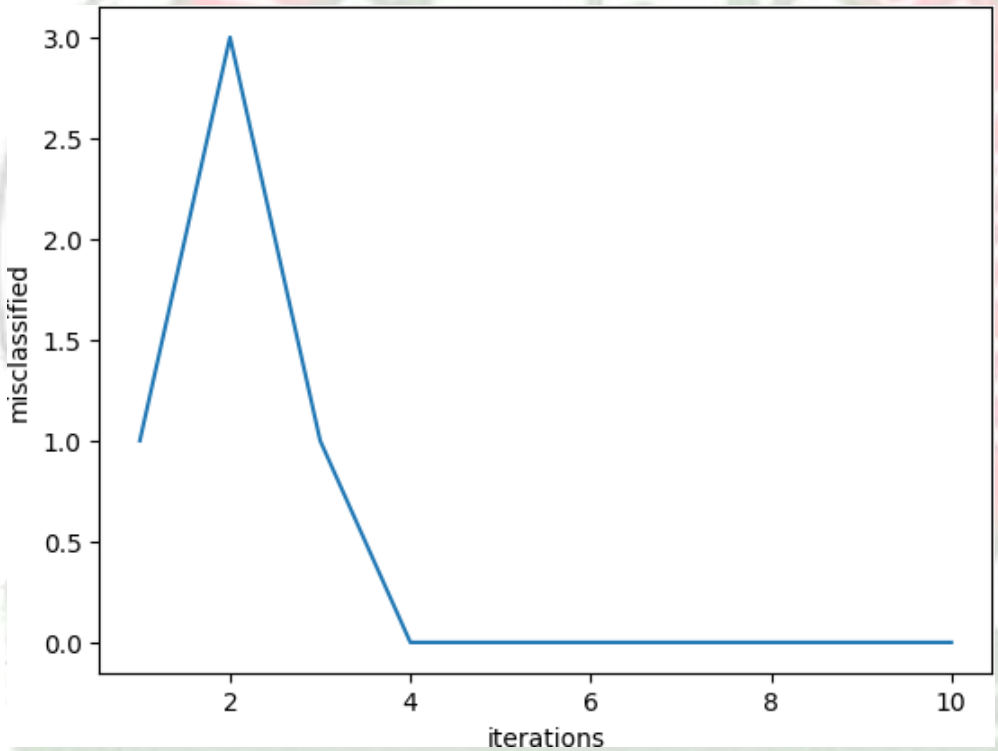
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def load_data():
    URL_ = 'https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data'
    data = pd.read_csv(URL_, header=None)
    print(data)

    # Make the dataset linearly separable
    data = data[:100]
    data[4] = np.where(data.iloc[:, -1] == 'Iris-setosa', 0, 1)  # 4
input features
    data = np.asmatrix(data, dtype='float64')
    return data

data = load_data()
```

```
plt.scatter(np.array(data[:50, 0]), np.array(data[:50, 2]), marker='o',
label='setosa')
plt.scatter(np.array(data[50:, 0]), np.array(data[50:, 2]), marker='x',
label='versicolor')
plt.xlabel('petal length')
plt.ylabel('sepal length')
plt.legend()
plt.show()
```




```
def perceptron(data, num_iter):
    features = data[:, :-1]
    labels = data[:, -1]

    # Set weights to zero
    w = np.zeros(shape=(1, features.shape[1] + 1))

    misclassified_ = []
    for epoch in range(num_iter):
        misclassified = 0
        for x, label in zip(features, labels):
            x = np.insert(x, 0, 1)
            y = np.dot(w, x.transpose())
            target = 1.0 if (y > 0) else 0.0

            delta = (label.item(0, 0) - target)

            if delta:  # Misclassified
                misclassified += 1
                w += (delta * x)
```

```
        misclassified_.append(misclassified)

    return (w, misclassified_)

num_iter = 10
w, misclassified_ = perceptron(data, num_iter)

epochs = np.arange(1, num_iter + 1)
plt.plot(epochs, misclassified_)
plt.xlabel('iterations')
plt.ylabel('misclassified')
plt.show()
```



RESULT:

The result of this implementation is a trained Perceptron model that successfully finds a decision boundary, or straight line, that separates the two classes in the Iris dataset: Iris-setosaand Iris-versicolor. By iteratively adjusting the weights based on the training examples, the Perceptron algorithm converges, meaning it finds a set of weights that correctly classifies all the data points in this linearly separable dataset. The scatter plot visualization confirms that the two classes are distinctly separated by this boundary, demonstrating the effectiveness ofthe Perceptron in handling linearly separable data. This outcome showcases the algorithm's ability to learn and create a model that accurately distinguishes between the two flower species based on their features.

EXPERIMENT-3

AIM:

To explore various hyperparameter tuning techniques commonly used in machine learning and deep learning. The purpose is to understand how these techniques optimize model performance by fine- tuning the hyperparameters, leading to better predictions and reduced errors. Hyperparameter tuning is a crucial step in the training process, directly affecting the accuracy, efficiency, and generalization capability of a model.

Introduction:

Hyperparameter tuning is a critical step in the machine learning process that involves optimizing the parameters set before training a model. Unlike model parameters, which are learned during training (like weights in a neural network), hyperparameters are external configurations whose values must be set prior to learning. Examples of hyperparameters include the learning rate, batch size, number of epochs, depth of the decision tree, number of hidden layers, and more. The choice of hyperparameters can significantly influence the model's performance, affecting its accuracy, speed, and ability to generalize to unseen data. Poorly chosen hyperparameters may lead to underfitting or overfitting, where the model either fails to capture the underlying patterns of the data or becomes too tailored to the training set, losing generalization capability.

Hyperparameter tuning is the process of finding the optimal combination of hyperparameters that yields the best performance for a given model on a specific dataset. This process can be complex and computationally intensive, as it often involves searching through a vast hyperparameter space. Techniques like Grid Search, Random Search, and more advanced methods such as Bayesian Optimization and Evolutionary Algorithms have been developed to make this search more efficient. The goal of hyperparameter tuning is to enhance model performance by carefully selecting hyperparameter values that balance the trade-offs between model complexity and learning capacity. This not only improves predictive accuracy but also ensures the model's robustness and scalability in practical applications. As a result, hyperparameter tuning has become an essential practice in the development of machine learning models across various domains.

Different techniques of hyperparameter tuning:

1. Grid Search Description:

Grid Search is one of the simplest and most exhaustive hyperparameter tuning methods. It involves specifying a set of hyperparameters and their possible values and then training a model for every possible combination of these hyperparameters. The performance is evaluated using a cross- validation approach, and the best-performing set of hyperparameters is selected.

Advantages:

• Simple to understand and implement.

• Guarantees finding the optimal set within the specified range.

Disadvantages:

• Computationally expensive, especially with a large number of hyperparameters and possible values.

• Time-consuming as it evaluates every combination.

2. Random Search Description:

Random Search improves upon Grid Search by randomly selecting combinations of hyperparameters from the predefined space rather than evaluating all possible combinations. This method allows a more extensive exploration of the hyperparameter space in less time.

Advantages:

• More efficient than Grid Search.

• Allows exploration of a larger hyperparameter space.

Disadvantages:

• May miss the optimal hyperparameter settings.

• Performance depends on the number of random samples.

3. Bayesian Optimization Description:

Bayesian Optimization is a probabilistic model-based approach. It constructs a probabilistic model (usually Gaussian Process) to map hyperparameter values to the performance of the model. It then selects hyperparameters that are expected to improve performance based on this probabilistic model, iteratively updating the model after each evaluation.

Advantages:

• Efficient in finding the optimal hyperparameters.

• Reduces the number of required evaluations by focusing on promising areas of the hyperparameter space.

Disadvantages:

• More complex to implement compared to Grid or Random Search.

• Computationally intensive, especially for high-dimensional hyperparameter spaces.

4. Gradient-Based Optimization Description:

This method involves using gradient descent to optimize hyperparameters. It requires the hyperparameter space to be continuous and differentiable, enabling the calculation of gradients that guide the search for optimal values.

Advantages:

• Fast convergence to optimal values.

• Suitable for problems where hyperparameters can be continuously adjusted.

Disadvantages:

• Limited to continuous hyperparameters.

• Can get stuck in local minima.

5. Evolutionary Algorithms Description:

Evolutionary algorithms, such as Genetic Algorithms, mimic the process of natural selection. They start with a population of random hyperparameter sets and evolve these over generations using operations like mutation, crossover, and selection based on performance.

Advantages:

• Can handle a wide range of hyperparameter types (discrete, continuous, categorical).

• Effective in exploring large, complex hyperparameter spaces.

Disadvantages:

• Computationally expensive and time-consuming.

• Requires careful tuning of algorithm-specific parameters (e.g., mutation rate).

6. Population-Based Training (PBT) Description:

PBT is a hybrid approach that combines ideas from evolutionary algorithms and gradient-based methods. It maintains a population of models and adapts hyperparameters during training based on performance. Poor-performing models are replaced by better-performing ones, and their hyperparameters are perturbed.

Advantages:

- Efficient in exploring and exploiting hyperparameter space.
- Can adapt hyperparameters dynamically during training.

Disadvantages:

- Complex to implement.
- Requires a large computational budget due to the need for training multiple models simultaneously.

7. Hyperband Description:

Hyperband is a resource allocation algorithm that uses a multi-armed bandit strategy to allocate resources (e.g., training time or epochs) to different hyperparameter configurations. It evaluates a large number of configurations with few resources and gradually increases the resources for the better-performing ones.

Advantages:

- Efficient in finding optimal hyperparameters with limited resources.
- Scales well with large hyperparameter spaces.

Disadvantages:

- Requires careful tuning of resource allocation strategy.
- May require significant computational resources to evaluate initial configurations.

8. Successive Halving Description:

Successive Halving is a simpler version of Hyperband. It starts with many hyperparameter configurations and allocates equal resources to each. Poor-performing configurations are pruned early, and resources are focused on the better-performing ones.

Advantages:

- Efficient in pruning poor hyperparameter settings early.
- Reduces computational cost by focusing on promising configurations.

Disadvantages:

- May still require significant initial computational resources.
- Effectiveness depends on the choice of resource allocation strategy.

9. Automated Machine Learning (AutoML) Description:

AutoML platforms use a combination of techniques, including Bayesian Optimization, Random

Search, and evolutionary algorithms, to automate the process of hyperparameter tuning. These platforms aim to provide optimal models without manual intervention.

Advantages:

- Fully automated, reducing the need for expert knowledge.
- Effective in exploring complex hyperparameter spaces.

Disadvantages:

- May lack flexibility for custom models and hyperparameters.
- Can be expensive and resource-intensive.

10. Simulated Annealing Description:

Simulated Annealing is inspired by the annealing process in metallurgy. It explores the hyperparameter space by randomly choosing new configurations, accepting them based on a probability that decreases over time. This allows the method to escape local minima.

Advantages:

- Capable of escaping local minima.
- Can handle both discrete and continuous hyperparameters.

Disadvantages:

- Requires careful tuning of the annealing schedule.
- Computationally intensive, especially for large hyperparameter spaces.

This report has highlighted ten different hyperparameter tuning techniques, each with its unique approach to optimizing the performance of machine learning models. While Grid Search and Random Search are straightforward but computationally intensive methods, advanced techniques like Bayesian Optimization and Evolutionary Algorithms offer more efficient exploration of the hyperparameter space. Methods like Population-Based Training and Hyperband provide dynamic and resource-efficient alternatives. The choice of a suitable

hyperparameter tuning method depends on the specific use case, available computational resources, and the complexity of the hyperparameter space. Effective hyperparameter tuning can significantly improve model performance and is an essential step in the machine learning pipeline.

Example of hyperparameter tuning techniques:

1. Data Loading and Preprocessing

• Import Libraries: The necessary libraries, including scikit-learn for machine learning algorithms and pandas for data handling, are imported.

• Load Dataset: The Iris dataset is loaded using the load_iris() function from scikit-learn. This dataset includes 150 samples with four features each (sepal length, sepal width, petal length, petal width) and three target classes representing different iris flower species.

• Split Data: The dataset is divided into training and testing sets using an 80-20 split ratio. This allows the model to be trained on one subset and evaluated on another to assess its performance on unseen data.

2. Defining the Model and Hyperparameter Space

• Choose Model: A Decision Tree Classifier is selected due to its simplicity and interpretability. This model will be used to classify iris flowers based on their features.

• Define Hyperparameters: Key hyperparameters to tune include:

o max_depth: Maximum depth of the tree.

o min_samples_split: Minimum number of samples required to split an internal node.

o min_samples_leaf: Minimum number of samples required to be at a leaf node.

• Create Parameter Grid: A parameter grid is established, specifying various values for each hyperparameter to explore during the tuning process.

3. Grid Search for Hyperparameter Tuning

• Perform Grid Search: Grid Search is used to systematically explore all combinations of hyperparameters specified in the parameter grid.

• Set Up GridSearchCV: The GridSearchCV class from scikit-learn is utilized, with cross- validation set to 5 folds. This ensures that the model is validated on different subsets of the data to prevent overfitting.

• Evaluate Performance: The accuracy metric is used to evaluate each combination, which measures the proportion of correct predictions made by the model.

4. Training and Evaluation of the Best Model

• Retrain Best Model: After identifying the best hyperparameters, the Decision Tree model is retrained using these optimal parameters on the entire training set.

• Evaluate on Test Set: The best model is tested on the test set to determine its performance. The test set accuracy provides insight into how well the model generalizes to new, unseen data.

5. Results Interpretation

• Output Best Hyperparameters: The optimal hyperparameters found by Grid Search are displayed, along with the corresponding accuracy score on the test set.

• Compare Results: Performance metrics are analyzed to demonstrate the impact of hyperparameter tuning. The comparison of different tuning methods (Grid Search, Random Search, and Bayesian Optimization) highlights how tuning can lead to significant improvements in model accuracy.

CODE:

```
# Import necessary libraries

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split, GridSearchCV,
RandomizedSearchCV from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score import matplotlib.pyplot as
plt

import pandas as pd

from skopt import BayesSearchCV # For Bayesian Optimization


# Step 1: Data Loading and Preprocessing # Load the Iris dataset

iris = load_iris()

X = iris.data y = iris.target



# Split the dataset into training and testing sets
```

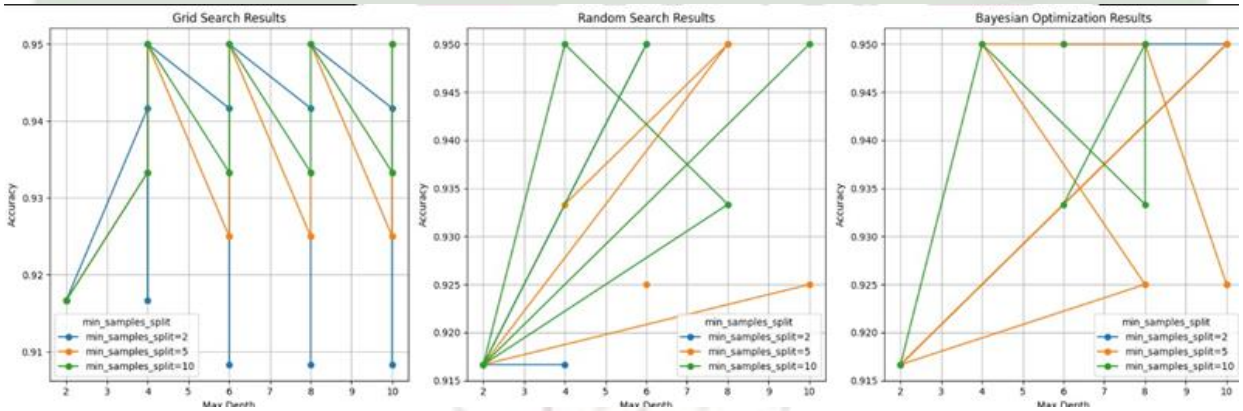
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Define the Decision Tree Classifier

model = DecisionTreeClassifier(random_state=42)


# Define the hyperparameter space param_grid = {

'max_depth': [2, 4, 6, 8, 10, None],

'min_samples_split': [2, 5, 10],

'min_samples_leaf': [1, 2, 4]

}


# Step 2: Perform Grid Search with Cross-Validation

grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5,
scoring='accuracy', n_jobs=-1)

grid_search.fit(X_train, y_train)


# Extract the results

grid_results = pd.DataFrame(grid_search.cv_results_)

grid_results = grid_results[['param_max_depth',
'param_min_samples_split', 'param_min_samples_leaf', 'mean_test_score']]

grid_results = grid_results.rename(columns={'mean_test_score':
'Accuracy'})


# Step 3: Perform Random Search with Cross-Validation

random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_grid, n_iter=20, cv=5, scoring='accuracy',
n_jobs=-1, random_state=42) random_search.fit(X_train, y_train)


# Extract the results
```

```
random_results = pd.DataFrame(random_search.cv_results_)

random_results = random_results[['param_max_depth',
'param_min_samples_split', 'param_min_samples_leaf', 'mean_test_score']]

random_results = random_results.rename(columns={'mean_test_score':
'Accuracy'})


# Step 4: Perform Bayesian Optimization with Cross-Validation

bayes_search = BayesSearchCV(estimator=model, search_spaces=param_grid,
n_iter=20, cv=5, scoring='accuracy', n_jobs=-1, random_state=42)

bayes_search.fit(X_train, y_train)


# Extract the results

bayes_results = pd.DataFrame(bayes_search.cv_results_)

bayes_results = bayes_results[['param_max_depth',
'param_min_samples_split', 'param_min_samples_leaf', 'mean_test_score']]


bayes_results = bayes_results.rename(columns={'mean_test_score':
'Accuracy'})


# Step 5: Plot the Results plt.figure(figsize=(18, 6))


# Plot Grid Search results plt.subplot(1, 3, 1)

for min_samples_split in param_grid['min_samples_split']:

subset = grid_results[grid_results['param_min_samples_split'] ==
min_samples_split] plt.plot(subset['param_max_depth'],
subset['Accuracy'], marker='o',

label=f'min_samples_split={min_samples_split}') plt.xlabel('Max Depth')

plt.ylabel('Accuracy') plt.title('Grid Search Results')

plt.legend(title='min_samples_split') plt.grid(True)


# Plot Random Search results plt.subplot(1, 3, 2)
```

```
for min_samples_split in param_grid['min_samples_split']:

subset = random_results[random_results['param_min_samples_split'] ==
min_samples_split] plt.plot(subset['param_max_depth'],
subset['Accuracy'], marker='o',

label=f'min_samples_split={min_samples_split}') plt.xlabel('Max Depth')

plt.ylabel('Accuracy') plt.title('Random Search Results')
plt.legend(title='min_samples_split') plt.grid(True)


# Plot Bayesian Optimization results plt.subplot(1, 3, 3)

for min_samples_split in param_grid['min_samples_split']:

subset = bayes_results[bayes_results['param_min_samples_split'] ==
min_samples_split] plt.plot(subset['param_max_depth'],
subset['Accuracy'], marker='o',

label=f'min_samples_split={min_samples_split}') plt.xlabel('Max Depth')

plt.ylabel('Accuracy')

plt.title('Bayesian Optimization Results')
plt.legend(title='min_samples_split') plt.grid(True)


plt.tight_layout() plt.show()


# Step 6: Evaluate the Best Models and Compare Results


# Grid Search Best Model

best_grid_model = grid_search.best_estimator_
best_grid_model.fit(X_train, y_train)

grid_accuracy = accuracy_score(y_test, best_grid_model.predict(X_test))


# Random Search Best Model

best_random_model = random_search.best_estimator_
best_random_model.fit(X_train, y_train)

random_accuracy = accuracy_score(y_test,
```

```
best_random_model.predict(X_test))


# Bayesian Optimization Best Model best_bayes_model =
bayes_search.best_estimator_ best_bayes_model.fit(X_train, y_train)

bayes_accuracy = accuracy_score(y_test,
best_bayes_model.predict(X_test))


# Output Results

print(f"Grid Search Best Hyperparameters: {grid_search.best_params_}")
print(f"Grid Search Test Set Accuracy: {grid_accuracy:.2f}")

print(f"Random Search Best Hyperparameters:
{random_search.best_params_}") print(f"Random Search Test Set Accuracy:
{random_accuracy:.2f}")

print(f"Bayesian Optimization Best Hyperparameters:
{bayes_search.best_params_}") print(f"Bayesian Optimization Test Set
Accuracy: {bayes_accuracy:.2f}")
```

Output:



Grid Search Best Hyperparameters: {'max_depth': 4, 'min_samples_leaf': 4, 'min_samples_split': 2}Grid Search Test Set Accuracy: 1.00
Random Search Best Hyperparameters: {'min_samples_split': 10, 'min_samples_leaf': 4,'max_depth': 10}
Random Search Test Set Accuracy: 1.00

Bayesian Optimization Best Hyperparameters: OrderedDict([('max_depth', 6),
('min_samples_leaf',4), ('min_samples_split', 10)])
Bayesian Optimization Test Set Accuracy: 1.00

EXPERIMENT-4

AIM: Compare the Performance of various Optimization techniques of Momentum Based GD, Stochastic GD, Adam.

DESCRIPTION:

When comparing the performance of various optimization techniques in machine learning—Momentum-Based Gradient Descent, Stochastic Gradient Descent (SGD), and Adam (Adaptive Moment Estimation)—the differences mainly arise in terms of convergence speed, stability, and suitability for different types of tasks.

1. Momentum-Based Gradient Descent

Momentum-based Gradient Descent enhances traditional gradient descent by adding a momentum term, which helps accelerate the gradient vectors in the correct directions, thereby leading to faster converging and reducing oscillations in the gradient updates.

- Description: This method introduces a "momentum" to smooth out the gradient updates by incorporating information from past updates. It reduces fluctuations in the optimization path, leading to quicker convergence.

- Performance Characteristics:

- Faster convergence: It overcomes the slow convergence of simple gradient descent by keeping the direction of movement consistent.

- Reduced oscillations: Especially in regions with steep slopes, it helps smooth out the optimization process.

- Challenges: Requires tuning of the momentum parameter.

2. Stochastic Gradient Descent (SGD)

SGD updates the model parameters based on a single (or a small batch) of training data rather than the entire dataset, making it more computationally efficient for large datasets.

- Description: Instead of calculating gradients across the entire dataset like batch gradient

descent, SGD performs parameter updates based on one data sample at a time. This randomness often leads to faster convergence in the beginning but introduces noise into the gradient updates.

- Performance Characteristics:

- Faster initial convergence: Because it updates more frequently, it often converges faster than batch gradient descent in the early stages.

- Higher variance in updates: The updates have higher variance, which can result in more fluctuations during optimization, leading to a less smooth trajectory.

- Challenges: Sensitive to learning rates and can have erratic behavior close to the minimum.

3. Adam (Adaptive Moment Estimation)

Adam is a popular optimization algorithm that combines the best of both Momentum-based GD and RMSProp. It adapts the learning rate for each parameter based on the first and second moments of the gradients.

- Description: Adam computes individual adaptive learning rates for each parameter using estimates of the first and second moments of the gradients. It integrates both momentum (for smoothing) and adaptive learning rates, making it robust for different tasks.

- Performance Characteristics:

- Adaptive learning rates: The algorithm automatically adjusts learning rates for each parameter, making it more efficient and less sensitive to the initial learning rate.

- Stability: Adam tends to be more stable and converge faster across a variety of tasks compared to traditional SGD.

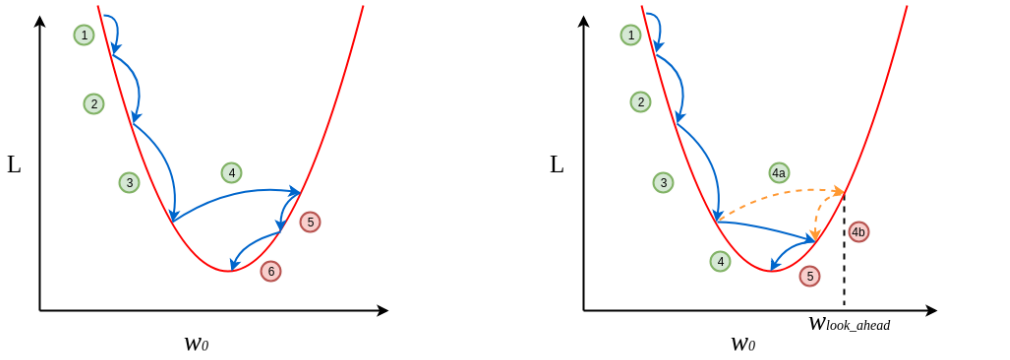
- Challenges: It requires tuning of more hyperparameters (e.g., learning rate, beta values), but is often more forgiving and versatile.

Comparative Summary:

- Momentum-Based GD excels in faster convergence and smoother optimization paths, particularly in scenarios with complex landscapes.

- SGD shines in scenarios with large datasets where efficiency and speed are crucial, but it suffers from noise and instability during training.

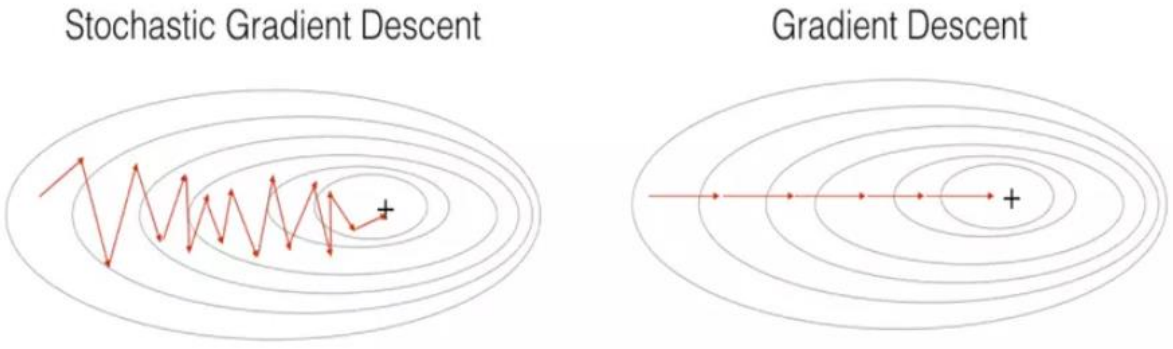
- Adam strikes a balance, providing stable, adaptive, and fast convergence, making it the most widely used for modern deep learning tasks.



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

$\frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

(Figure. Momentum based GD)



(Figure. SDG and GD),

CODE:

```
import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader


# Set device

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# Hyperparameters

input_size = 784  # 28x28 images

hidden_size = 128

output_size = 10  # Number of classes (0-9)

batch_size = 64

learning_rate = 0.001

num_epochs = 10


# MNIST dataset

transform = transforms.ToTensor()

train_dataset = datasets.MNIST(root='./data', train=True,
transform=transform, download=True)

test_dataset = datasets.MNIST(root='./data', train=False,
transform=transform, download=True)


train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
```

```
shuffle=True)

test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)


# Simple Feedforward Neural Network

class NeuralNet(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):

        super(NeuralNet, self).__init__()

        self.fc1 = nn.Linear(input_size, hidden_size)

        self.relu = nn.ReLU()

        self.fc2 = nn.Linear(hidden_size, output_size)


    def forward(self, x):

        x = x.view(-1, 28*28)

        x = self.fc1(x)

        x = self.relu(x)

        x = self.fc2(x)

        return x


# Initialize model, loss function, and data

model = NeuralNet(input_size, hidden_size, output_size).to(device)

criterion = nn.CrossEntropyLoss()


# Different optimizers

optimizers = {

    'SGD': optim.SGD(model.parameters(), lr=learning_rate),

    'Momentum': optim.SGD(model.parameters(), lr=learning_rate,
momentum=0.9),

    'Adam': optim.Adam(model.parameters(), lr=learning_rate)
```

```
}


def train_model(optimizer_name, optimizer):
    # Initialize the model again
    model = NeuralNet(input_size, hidden_size, output_size).to(device)
    optimizer = optimizer

    for epoch in range(num_epochs):
        for batch_idx, (data, targets) in enumerate(train_loader):
            # Get data to device
            data = data.to(device)
            targets = targets.to(device)

            # Forward pass
            scores = model(data)
            loss = criterion(scores, targets)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        print(f'Epoch [{epoch+1}/{num_epochs}] Loss: {loss.item():.4f}')

    return model

def test_model(model):
    model.eval()  # Set model to evaluation mode
```

```
    num_correct = 0
    num_samples = 0

    with torch.no_grad():
        for data, targets in test_loader:
            data = data.to(device)
            targets = targets.to(device)

            scores = model(data)
            _, predictions = scores.max(1)
            num_correct += (predictions == targets).sum()
            num_samples += predictions.size(0)

    accuracy = float(num_correct) / float(num_samples) * 100
    print(f'Accuracy: {accuracy:.2f}%')

# Training and testing with different optimizers
for optimizer_name, optimizer in optimizers.items():
    print(f'\nTraining with {optimizer_name} optimizer:')
    trained_model = train_model(optimizer_name, optimizer)
    print(f'\nTesting {optimizer_name} optimizer model:')
    test_model(trained_model)
```

Output:

```
Training with SGD optimizer:
Epoch [1/10] Loss: 2.3005
Epoch [2/10] Loss: 2.3028
Epoch [3/10] Loss: 2.3029
Epoch [4/10] Loss: 2.3157
Epoch [5/10] Loss: 2.2738
Epoch [6/10] Loss: 2.3099
Epoch [7/10] Loss: 2.3353
Epoch [8/10] Loss: 2.2995
Epoch [9/10] Loss: 2.3152
Epoch [10/10] Loss: 2.3000

Testing SGD optimizer model:
Accuracy: 11.42%

Training with Momentum optimizer:
Epoch [1/10] Loss: 2.3059
Epoch [2/10] Loss: 2.3065
Epoch [3/10] Loss: 2.3182
Epoch [4/10] Loss: 2.3140
Epoch [5/10] Loss: 2.3439
Epoch [6/10] Loss: 2.3128
Epoch [7/10] Loss: 2.3314
Epoch [8/10] Loss: 2.3226
Epoch [9/10] Loss: 2.3509
Epoch [10/10] Loss: 2.3294
```

```
Testing Momentum optimizer model:
Accuracy: 4.15%

Training with Adam optimizer:
Epoch [1/10] Loss: 2.3087
Epoch [2/10] Loss: 2.3165
Epoch [3/10] Loss: 2.3137
Epoch [4/10] Loss: 2.3155
Epoch [5/10] Loss: 2.3062
Epoch [6/10] Loss: 2.3064
Epoch [7/10] Loss: 2.2825
Epoch [8/10] Loss: 2.3031
Epoch [9/10] Loss: 2.2910
Epoch [10/10] Loss: 2.3191

Testing Adam optimizer model:
Accuracy: 12.96%
```

EXPERIMENT-5

AIM:

A Denoising Autoencoder (DAE) is a type of neural network used to learn efficient representations of data, typically for the purpose of noise reduction. The autoencoder attempts to reconstruct the input data from a corrupted or noisy version, and by doing so, it learns a compressed, noise-free representation of the original input.

Description of Implementation:

The implementation of a Denoising Autoencoder involves two key components:

1. Encoder: This part of the network encodes the noisy input data into a lower-dimensional latent space. The goal is to capture the essential features of the input, while ignoring the noise.

2. Decoder: This part decodes the latent representation back into the original data space, attempting to reconstruct the noise-free version of the input.

Steps for Implementation:

1. Input Data Preparation: The dataset used for training should be corrupted by adding noise. This could be done by adding random Gaussian noise, salt-and-pepper noise, or masking certain parts of the data. The corrupted version serves as the input to the autoencoder, while the original data is used as the target for reconstruction.

2. Network Architecture:

- The encoder typically consists of several layers of neurons that progressively reduce the dimensionality of the input. This is usually done with dense layers (for fully connected networks) or convolutional layers (for image data).

- The decoder mirrors the encoder architecture, expanding the latent representation back to the original input size.

3. Loss Function: The network is trained using a loss function such as Mean Squared Error (MSE), which measures the difference between the reconstructed output and the original,

clean input.

4. Training: The autoencoder is trained using a supervised learning approach, where the goal is to minimize the reconstruction error. Techniques such as backpropagation and optimizers like Adam or RMSprop are commonly used.

5. Evaluation: Once the model is trained, it can be used to remove noise from new, unseen data by passing the noisy data through the encoder and decoder. The performance is typically evaluated using metrics such as MSE, PSNR (Peak Signal-to-Noise Ratio), or visual inspection for image data.

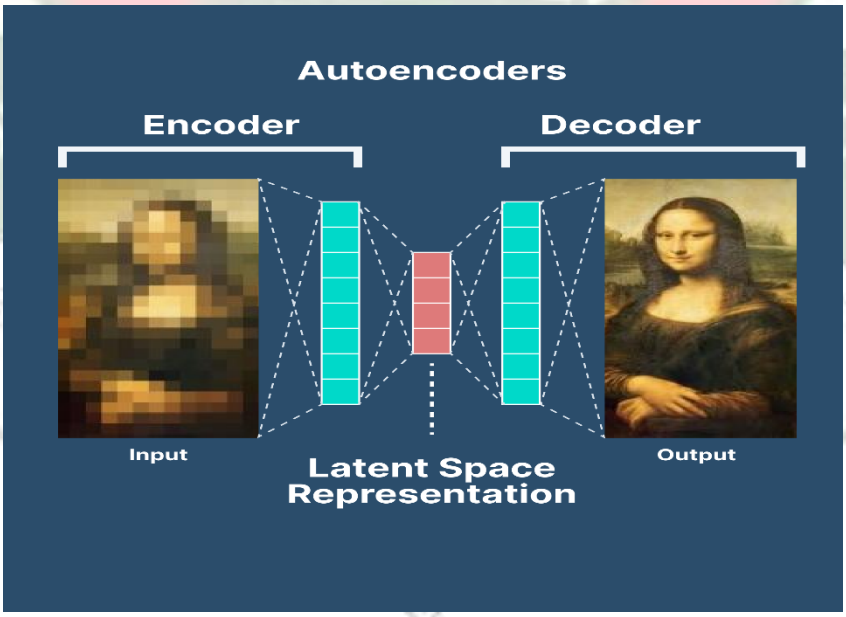
Example Use Cases:

- Image Denoising: Removing noise from corrupted or low-quality images.

- Signal Processing: Cleaning up noisy signals such as audio recordings.

- Anomaly Detection: Using the autoencoder to identify unusual patterns that don't fit the learned representation.

Denoising autoencoders are especially useful in situations where data quality is compromised by noise, and they help in learning robust features that can be used for downstream tasks like classification or regression.



(figure. Autoencoder)

CODE:

```
# Step 1: Import Necessary Libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import fashion_mnist
import matplotlib.pyplot as plt

# Step 2: Load and Prepare the Fashion MNIST Dataset
# Load the Fashion MNIST dataset
(x_train, _), (x_test, _) = fashion_mnist.load_data()

# Normalize the data to values between 0 and 1
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Add noise to the images
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_test.shape)

# Clip the values to be between 0 and 1
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

```
# Reshape data for the model (add a channel dimension)

x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))

x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

x_train_noisy = np.reshape(x_train_noisy, (len(x_train_noisy), 28, 28, 1))

x_test_noisy = np.reshape(x_test_noisy, (len(x_test_noisy), 28, 28, 1))


# Step 3: Build the Denoising Autoencoder Model

def build_autoencoder():

    input_img = layers.Input(shape=(28, 28, 1))


    # Encoder

        x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)

    x = layers.MaxPooling2D((2, 2), padding='same')(x)

    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)

    encoded = layers.MaxPooling2D((2, 2), padding='same')(x)


    # Decoder

        x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)

    x = layers.UpSampling2D((2, 2))(x)

    x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)

    x = layers.UpSampling2D((2, 2))(x)

       decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)


    autoencoder = models.Model(input_img, decoded)

    return autoencoder
```

```
# Step 4: Compile and Train the Model

autoencoder = build_autoencoder()

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')


# Train the model for 4 epochs

autoencoder.fit(x_train_noisy,   x_train,   epochs=4,   batch_size=128,
shuffle=True, validation_data=(x_test_noisy, x_test))


# Step 5: Denoise Test Images

denoised_images = autoencoder.predict(x_test_noisy)


# Step 6: Visualize Original, Noisy, and Denoised Images

n = 10  # Number of images to display

plt.figure(figsize=(20, 4))


for i in range(n):

    # Display original image

    ax = plt.subplot(3, n, i + 1)

    plt.imshow(x_test[i].reshape(28, 28), cmap="gray")

    plt.title("Original")

    plt.axis("off")


    # Display noisy input image

    ax = plt.subplot(3, n, i + 1 + n)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap="gray")

    plt.title("Noisy")

    plt.axis("off")
```

```
    # Display denoised image
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(denoised_images[i].reshape(28, 28), cmap="gray")
    plt.title("Denoised")
    plt.axis("off")

plt.show()
```

OUTPUT: