Sinhgad Institute of Technology and Science, Narhe

Department of Information Technology

414447: Lab Practice IV

Deep Learning

Deep Learning Laboratory Manual

Savitribai Phule Pune University, Pune Final Year Information Technology (2019 Course)		
Teaching Scheme:	**Credit Scheme:**	**Examination Scheme:**
Practical (PR):02 hrs/week	**01 credits**	**PR: 25 Marks** **TW: 25 Marks**

Savitribai Phule Pune University, Pune

Final Year Information Technology (2019 Course)

414447: Lab Practice IV Deep Learning

Teaching Scheme:	Credit Scheme:	Examination Scheme:
Practical (PR) : 2 hrs/week	0Credits	PR: 25 Marks TW: 25 Marks

Prerequisites:

1. Python programming language

Course Objectives:

The objective of the course is

1. To be able to formulate deep learning problems corresponding to different applications.
2. To be able to apply deep learning algorithms to solve problems of moderate complexity.
3. To apply the algorithms to a real-world problem, optimize the models learned and report on the expected accuracy that can be achieved by applying the models.

Course Outcomes:

On completion of the course, students will be able to–

CO1. Learn and Use various Deep Learning tools and packages.

CO2. Build and train a deep Neural Network modelsfor use in various applications.

CO3. Apply Deep Learning techniques like CNN, RNN Auto encoders to solve real word Problems.

CO4. Evaluate the performance of the model build using Deep Learning.

Guidelines for Instructor's Manual

1. The faculty member should prepare the laboratory manual for all the experiments and it should be made available to students and laboratory instructor/Assistant.

Guidelines for Student's Lab Journal

1. Student should submit term work in the form of handwritten journal based on specified list of assignments.

2. Practical Examination will be based on the term work.

3. Candidate is expected to know the theory involved in the experiment.

4. The practical examination should be conducted if and only if the journal of the candidate is complete in all aspects.

Guidelines for Lab /TW Assessment

1. Examiners will assess the term work based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of handwritten write-up along with results of implemented assignment, attendance etc.

2. Examiners will judge the understanding of the practical performed in the examination by asking some questions related to the theory & implementation of the experiments he/she has carried out.

3. Appropriate knowledge of usage of software and hardware related to respective laboratory should be checked by the concerned faculty member.

Guidelines for Laboratory Conduction

As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers of the program in journal may be avoided. There must be hand-written write-ups for every assignment in the journal. The DVD/CD containing student's programs should be attached to the journal by every student and same to be maintained by department/lab In-charge is highly encouraged. For reference one or two journals may be maintained with program prints at

Laboratory.

List of Laboratory Assignments

Mapping of course outcomes for Group A assignments: CO1, CO2,CO3,CO4

1. Study of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch. Document the distinct features and functionality of the packages.

Note: Use a suitable dataset for the implementation of following assignments.

2. Implementing Feed forward neural networks with Keras and TensorFlow

a. Import the necessary packages

b. Load the training and testing data (MNIST/CIFAR10)

c. Define the network architecture using Keras

d. Train the model using SGD

e. Evaluate the network

f. Plot the training loss and accuracy

3. Build the Image classification model by dividing the model into following 4 stages:

a. Loading and pre-processing the image data

b. Defining the model's architecture

c. Training the model

d. Estimating the model's performance

4. Use Autoencoder to implement anomaly detection. Build the model by using:

a. Import required libraries

b. Upload / access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation Metrics

5. Implement the Continuous Bag of Words (CBOW) Model. Stages can be:

a. Data preparation

b. Generate training data

c. Train model

d. Output

6. Object detection using Transfer Learning of CNN architectures

a. Load in a pre-trained CNN model trained on a large dataset

b. Freeze parameters(weights) in model'slower convolutional layers

c. Add custom classifier with several layers of trainable parameters to model

d. Train classifier layers on training data available for task

e. Fine-tune hyper parameters and unfreeze more layers as needed

Reference Books:

1. Hands-On Deep Learning Algorithms with Python: Master Deep Learning Algorithms with Extensive Math by Implementing Them Using TensorFlow

2. Python Deep Learning, 2nd Edition by Ivan Vasilev , Daniel Slater, GianmarioSpacagna,, Peter Roelants, Valentino Zocca

3. Natural Language Processing with Python Quick Start Guide by Mirant Kasliwal

Virtual Laboratory:

SPIT's Virtual Labs for AI and Deep Learning: https://vlab.spit.ac.in/ai/

Assignment No.: 01

Study of Deep Learning Packages: TensorFlow, Keras, Theano, and PyTorch

Field

Deep Learning, Neural Networks, Machine Learning Frameworks

Aim

To study and understand the distinct features and functionalities of major deep learning packages: TensorFlow, Keras, Theano, and PyTorch.

Objective

- Explore the architecture and design philosophy of each package.
- Compare usability, flexibility, and ecosystem support.
- Identify package-specific features like computation graphs, GPU support, and debugging facilities.
- Gain hands-on familiarity through simple experimentation.
- Understand which package suits different types of deep learning tasks and research.

Theory

Deep learning frameworks provide the tools and APIs to define, train, and deploy neural networks. TensorFlow, developed by Google, is a comprehensive end-to-end platform supporting computation graphs and distributed computing. Keras is a high-level API, now part of TensorFlow, known for user-friendly model building. Theano was one of the first symbolic math libraries focused on deep learning but is now largely deprecated. PyTorch, developed by Facebook, offers dynamic computation graphs and strong community adoption, favored in research.

Key differences include graph construction (static vs dynamic), ease of model prototyping, scalability, and available pre-trained models.

Procedure

1. Install all four packages in separate or isolated Python environments.
2. Explore their API documentation and example code.
3. Build a simple neural network or load example models in each.

4. Compare:
    - Syntax and ease of coding
    - Model compilation and training steps
    - Debugging and visualization tools
    - GPU and CPU utilization
    - Community and library ecosystem
5. Document observations emphasizing distinct features and ideal usage contexts.

Observations

- TensorFlow offers robust deployment and visualization (TensorBoard) options.
- Keras enables quick model building with clean, concise code.
- Theano requires more manual graph management; less popular now.
- PyTorch's dynamic graphs simplify debugging and have strong community support.
- Performance variations observed on different hardware.
- API maturity and documentation vary.

Result

An informed comparative understanding of TensorFlow, Keras, Theano, and PyTorch was achieved, facilitating wise framework selection for specific deep learning projects.

Conclusion

Each deep learning package has unique strengths. TensorFlow and PyTorch dominate current usage due to scalability and flexibility. Keras's simplicity suits beginners and rapid prototyping. Knowledge of these frameworks empowers effective model development.

Pre-Lab

- Basic understanding of neural networks.
- Familiarity with Python programming.
- Install Python and package managers (pip or conda).

Post-Lab

- Try building advanced neural networks with chosen frameworks.
- Explore integration with other ML tools.
- Follow updates and community projects.

Assignment No.: 02

Implementing Feed forward Neural Networks with Keras and TensorFlow

Field

Deep Learning, Neural Networks, Keras, TensorFlow

Aim

To implement a feedforward neural network using Keras and TensorFlow on a suitable dataset like MNIST or CIFAR-10.

Objective

- Import and utilize key deep learning packages.
- Load and preprocess standard datasets.
- Define a multilayer feedforward network architecture.
- Train the model using Stochastic Gradient Descent (SGD).
- Evaluate model performance quantitatively.
- Visualize training loss and accuracy trends.

Theory

Feedforward neural networks consist of input, hidden, and output layers with neurons connected unidirectionally. They learn complex patterns via backpropagation. Keras and TensorFlow provide layered APIs to define, compile, and optimize these networks. SGD is an iterative optimization method essential for training.

Procedure

1. Import Keras and TensorFlow libraries.
2. Load dataset (MNIST or CIFAR-10) with standard train-test splits.
3. Normalize and preprocess data.
4. Define neural network: input layer, one or more hidden layers with activation functions (ReLU), and output layer with appropriate activation (softmax).
5. Compile the model with SGD optimizer, specifying loss and metrics.
6. Train the model for several epochs on training data.
7. Evaluate accuracy and loss on test data.

8. Plot training curves for loss and accuracy.

Observations

- Model learns progressively; training loss decreases.
- Accuracy improves over epochs on train and test sets.
- Potential overfitting or underfitting visible in curve behavior.
- Training times depend on model complexity and hardware.

Result

Feed forward neural network constructed and trained successfully, demonstrating ability to classify images.

Conclusion

Keras and TensorFlow simplify deep learning workflows, enabling rapid prototyping and effective training of feedforward models.

Pre-Lab

- Understand structure of feedforward neural networks.
- Basic familiarity with dataset formats (image data).
- Installation of required deep learning libraries.

Post-Lab

- Experiment with different network depths and parameters.
- Apply regularization to reduce overfitting.
- Extend to convolutional neural networks for image tasks.

Assignment No.: 03

Build the Image Classification Model in Four Stages

Field

Computer Vision, Deep Learning, Image Classification

Aim

To build an image classification model by systematically implementing data loading, model architecture, training, and performance estimation.

Objective

- Load and preprocess image datasets effectively.
- Define a suitable neural network architecture for classification.
- Train the network on prepared datasets.
- Evaluate and estimate the model's classification accuracy and errors.

Theory

Image classification assigns labels to images based on learned features. Preprocessing involves normalization and augmentation. Neural network design influences learning capacity and generalization. Training updates model parameters to minimize classification error. Performance assessment through accuracy metrics guides model improvement.

Procedure

1. Load dataset (e.g., CIFAR-10), resize images and normalize pixel values.
2. Define model architecture: input layer, convolutional or dense layers, activation functions, output layer for classification.
3. Compile the model with optimizer and loss function.
4. Train on training set for predefined epochs with batch processing.
5. Validate on separate testing data.
6. Measure accuracy, precision, recall.
7. Document model performance and observed behaviors.

Observations

- Model learns to distinguish image classes gradually.
- Confusion matrix highlights classification strengths and weaknesses.
- Overfitting signs may appear with large training epochs.

Result

Image classifier built in modular stages, achieving meaningful classification accuracy.

Conclusion

Structuring image classification into stages improves development clarity and modularity, enhancing performance tuning.

Pre-Lab

- Understand image data formats and preprocessing necessities.
- Review basics of CNNs and feedforward networks.

Post-Lab

- Implement data augmentation.
- Experiment with transfer learning to boost accuracy.

Assignment No.: 04

Autoencoder for Anomaly Detection

Field

Unsupervised Learning, Neural Networks, Anomaly Detection

Aim

To use autoencoders for detecting anomalies by compressing and reconstructing input data.

Objective

- Import essential libraries.
- Upload or access appropriate datasets.
- Build encoder and decoder networks forming an autoencoder.
- Compile models with optimizers and evaluation metrics.
- Detect anomalies via reconstruction errors.

Theory

Autoencoders learn compressed representations (latent space) of input data. The encoder maps original data to latent vectors, and the decoder reconstructs data from these vectors. Anomalies are identified by high reconstruction error, indicating input divergence from learned normal patterns.

Procedure

1. Import deep learning and data handling libraries.
2. Load dataset (e.g., MNIST, sensor data).
3. Build encoder network compressing input to lower dimension.
4. Build decoder network reconstructing input from latent representation.
5. Compile autoencoder with optimizer (e.g., Adam), loss (MSE), and metrics.
6. Train on normal data subset.
7. Compute reconstruction error on test data.
8. Flag data points with high error as anomalies.

Observations

- Autoencoder converges reducing reconstruction loss.
- Normal data reconstructed accurately.
- Anomalous data reconstruction error spikes.

Result

Autoencoder successfully implemented and utilized for anomaly detection.

Conclusion

Autoencoders are effective unsupervised tools for identifying outliers in complex datasets.

Pre-Lab

- Understand neural network basics and autoencoder principles.
- Explore datasets suitable for anomaly detection.

Post-Lab

- Tune architecture for better precision.
- Apply to real-world anomalies like fraud or faults.

Assignment No.: 05

Continuous Bag of Words (CBOW) Model

Field

Natural Language Processing, Word Embeddings

Aim

To implement the CBOW model for word embedding learning in a text corpus.

Objective

- Prepare textual data for modeling.
- Generate training contexts and target data.
- Train CBOW model to predict words from context.
- Analyze and output trained word vectors.

Theory

CBOW predicts a center word given surrounding context words. It captures semantic relationships by embedding words into a continuous vector space. Training uses a shallow neural network optimizing prediction accuracy.

Procedure

1. Preprocess corpus: tokenize and remove stopwords.
2. Generate input-output pairs (context-target).
3. Define and train the CBOW model using embeddings.
4. Extract embeddings for evaluation or downstream tasks.
5. Output vector representations.

Observations

- Training loss decreases over epochs.
- Vectors show meaningful semantic similarity.
- Vocabulary learned adequately from dataset.

Result

CBOW model built and trained to produce word embeddings.

Conclusion

CBOW effectively learns word representations useful in language modeling and NLP applications.

Pre-Lab

- Review NLP concepts and embedding techniques.
- Prepare a text dataset.

Post-Lab

- Compare CBOW to Skip-Gram model.
- Use embeddings in text classification or clustering.

Assignment No.: 06

Object Detection Using Transfer Learning of CNN Architectures

Field

Computer Vision, Transfer Learning, Deep Learning

Aim

To perform object detection by leveraging pre-trained CNN models and transfer learning methodologies.

Objective

- Load and adapt a pre-trained CNN model.
- Freeze lower convolutional layers to retain learned features.
- Add and train custom classifier layers for target task.
- Fine-tune parameters and unfreeze layers progressively.
- Optimize model performance on new dataset.

Theory

Transfer learning utilizes models pre-trained on large datasets (e.g., ImageNet) to jumpstart learning on new tasks, saving time and improving accuracy. Freezing early layers prevents overfitting by retaining generic feature detectors. Custom classifier layers adapt the model to specific object detection tasks.

Procedure

1. Import pre-trained CNN model without top layers.
2. Freeze early convolutional blocks.
3. Add new fully connected layers tailored to object classes.
4. Compile the model with suitable loss and optimizer.
5. Train classifier layers on task data.
6. Gradually unfreeze layers and fine-tune with adjusted learning rates.
7. Evaluate model accuracy and detection metrics.

Observations

- Training converges faster due to transferred features.
- Freezing layers reduces overfitting initially.
- Fine-tuning improves accuracy on target dataset.
- Detection precision and recall improve with iterative tuning.

Result

Object detection model built effectively using transfer learning techniques.

Conclusion

Transfer learning accelerates model development and boosts performance in computer vision tasks.

Pre-Lab

- Understand CNN architectures and transfer learning concepts.
- Prepare labeled images for detection tasks.

Post-Lab

- Explore advanced architectures (Faster-RCNN, YOLO).
- Implement data augmentation and hyperparameter optimization.

This lab manual outlines key experiments in deep learning, NLP, and computer vision covering foundational package usage, neural network implementation, practical modeling workflows, and advanced learning techniques. Completion of these tasks will develop strong theoretical understanding and practical skills in contemporary AI and machine learning.

Annexure

Experiment 1 – Comparative Study of Deep-Learning Packages (TensorFlow 2, Keras, PyTorch, Theano)

Pre-Lab

1. Install Anaconda / Miniconda.
2. Create environments:

```
conda create -n tf2 python=3.10 tensorflow
conda create -n torch python=3.10 pytorch torchvision torchaudio -c pytorch
```

3. Read the documentation "TensorFlow Guide → Eager Execution" and "PyTorch Autograd".

Objectives

- Compare the syntax, computational-graph philosophy, and deployment options of four DL frameworks.

Theory

- **Static graphs** (TensorFlow 1, Theano) compile once, run many times—good for optimisation but less intuitive.
- **Dynamic graphs** (PyTorch, TF 2 eager) build the graph on the fly, making debugging easier.
- Keras is now an *interface* inside TF 2 providing concise APIs (`Sequential`, Functional).

Deep-learning libraries supply the computational graph, automatic-differentiation engine, GPU/TPU back-ends, and higher-level APIs that let us build, train, and deploy neural networks. Although they ultimately perform the same tensor calculus, their philosophies—static vs. dynamic graphs, deployment scope, and usability—differ significantly.

Functional Highlights

1. **TensorFlow 2.x**
    - Combines eager execution (interactive, NumPy-like) with `@tf.function` graph tracing for production-level performance.
    - Integrated **`tf.data`** for scalable data pipelines, **`tf.distribute`** for multi-GPU/TPU strategy.
    - `SavedModel` format preserves computation graph + variables + signatures, enabling seamless serving.

2. **Keras (tf.keras)**

    - High-level, declarative syntax—`model = Sequential([...])`—ideal for rapid iteration.
    - Functional API supports DAG-style architectures (multi-input/output).
    - Extensive callbacks (`EarlyStopping`, `ReduceLROnPlateau`, `TensorBoard`) simplify training loops.
    - From TF 2.0 onward, Keras is tightly integrated: one install, one backend.

3. **PyTorch**

    - *Define-by-run*: the graph is built on each forward pass, allowing native Python loops, recursion, and conditional logic.
    - `torch.nn.Module` subclasses encapsulate parameters and forward computation; parameters registered automatically.
    - `torch.jit` (TorchScript) traces or scripts models for static graphs, enabling C++ deployment.
    - Strong community support for cutting-edge research (e.g., Hugging Face Transformers, Diffusers).

4. **Theano**

    - Pioneered symbolic tensors and GPU acceleration in Python.
    - Performs graph optimisation (fusion, constant folding) during compile.
    - Now archived (2017); important historically, but modern projects have largely migrated to TF or PyTorch.

Aspect	TensorFlow 2.x	Keras (tf.keras)	PyTorch	Theano (≤1.0.5, archived)
Year of Release	2015 (Google)	2015 (François Chollet; integrated into TF 2)	2016 (Facebook/Meta)	2010 (LISA lab, Univ. Montréal)
Core Graph Model	*Eager execution by default* + optional `tf.function` static graphs	High-level *interface* that builds on TensorFlow execution model	Pure **dynamic graph** (define-by-run)	**Static graph** compiled to C/CUDA
Primary Abstractions	`tf.Tensor`, `tf.data`, `tf.Module`, `SavedModel`	`Sequential`, Functional API, Subclassed `Model`	`torch.Tensor`, `nn.Module`, `autograd.Function`	Symbolic variables, *scan* loops
Auto-Diff Engine	Reverse-mode in C++, accessible via `GradientTape`	Inherits TF engine	Built-in autograd tracking on every op	Symbolic differentiation during graph compilation
Device Support	CPU, GPU, TPU	Same	CPU, GPU	CPU, GPU (CUDA)

Deployment & Serving	TF-Serving, TF-Lite (mobile), TF-JS, TF-Hub, TensorRT converter	Same	TorchScript, TorchServe, ONNX export, C++ frontend	Limited (no official serving stack)
Ecosystem Extras	tf.data pipelines, tf.summary/TensorBoard, tf.distribute for multi-device, tf.keras for high-level API	Integrated callbacks, AutoTuning, mixed precision	torchvision, torchaudio, torchtext, distributed, Profiler	Lasagne, Blocks (third-party, discontinued)
Distinct Strengths	- Production-grade deployment - Mixed eager/graph for speed - Multi-language (Python, C++, Java, Go)	- Minimal boilerplate - Rapid prototyping - Huge model-zoo on TF-Hub	- Pythonic control flow (loops, if statements inside forward pass) - Simple debugging with standard Python tools - Widely used in research	- First widely-adopted auto-diff lib - Numpy-like syntax; laid groundwork for later libraries
Limitations	Verbose low-level API; graph mode still has learning curve	Relies on TF runtime; less flexible than raw PyTorch for exotic research models	Fewer *enterprise* deployment tools out-of-box; mobile support relatively new	Project no longer under active development; static graph hampers interactive debugging

Equipment / Software

- OS: Ubuntu 22.04 / Windows 11
- Python 3.10, Conda, JupyterLab
- GPUs with CUDA 11.x (optional)

Procedure

1. Launch Jupyter, create a notebook **framework_demo.ipynb**.
2. Implement identical two-layer MLPs in all four libraries; measure lines of code and training time on MNIST subset.
3. Record RAM usage with `nvidia-smi` if GPU available.

Practical Considerations for This Lab

- **Dataset Choice:** CIFAR-10 or MNIST is recommended—they are included in both Keras and `torchvision` and keep training time low on classroom hardware.

- **Package Versions:**

```
pip install "tensorflow~=2.15" "torch>=2.2 torchvision torchaudio" theano keras
```

- **GPU Availability:** Verify with `nvidia-smi`. If GPUs aren't available, restrict epochs or use CPU-friendly subsets.
- **Notebook vs. Script:** Jupyter notebooks are encouraged for interactive exploration; save checkpoints (`.h5` for Keras, `.pt` for PyTorch) for later experiments (transfer learning, autoencoders).

Result / Observation

Record:

Framework	Lines of Code	Train Time (1 epoch)	Accuracy (%)
TensorFlow			
…			

Post-Lab Viva

1. Explain *eager* vs. *graph* execution.
2. How does PyTorch's `nn.Module` differ from Keras `Model` subclassing?

Summary

- **TensorFlow/Keras** excel in **production and cross-platform deployment** (TF-Lite, TF-JS).
- **PyTorch** dominates in **research prototyping** due to its intuitive imperative style.
- **Theano** is historically significant; many concepts (symbolic variables, graph optimisation) live on in TensorFlow's graph mode.

A clear grasp of the design philosophy and API surface of each framework enables you to select the most appropriate tool for subsequent assignments—feed-forward networks, CNN transfer learning, NLP with CBOW, and anomaly detection via autoencoders.

Conclusion

TensorFlow offers extensive deployment utilities, whereas PyTorch provides a more pythonic dynamic-graph experience. Keras remains the easiest high-level front-end.

Experiment 2 – Feed-Forward Neural Network (MNIST / CIFAR-10)

Pre-Lab

Revise soft-max, cross-entropy loss, stochastic gradient descent.

1. Verify installation:

```
python -c "import tensorflow as tf; print(tf.__version__)"
```

2. Review concepts: dense layers, activation functions (ReLU, soft-max), cross-entropy loss, SGD optimizer.

Objectives

Build, train, and evaluate a fully connected neural network classifier on MNIST (28×28 grayscale digits) or CIFAR-10 (32×32 RGB objects) using Keras with TensorFlow backend.

Theory

- **SGD** updates weights using mini-batch gradients:
  $ w_{t+1}=w_{t}-\eta\nabla L(w_t) $.
- **Soft-max** converts logits to probabilities.

1. Feed-Forward Neural Network (FF-NN) Architecture

A feed-forward network is the simplest form of an artificial neural network in which information moves strictly forward—from the input layer, through one or more hidden layers, to the output layer—without forming cycles or feedback loops.

```
Input → Hidden₁ → Hidden₂ → … → Output
```

- **Layers**
  - **Input layer**: receives raw features (e.g., 784 pixels for 28 × 28 MNIST; 3 × 32 × 32 = 3072 pixels for CIFAR-10).
  - **Hidden layers**: stacks of fully-connected (dense) neurons that learn hierarchical feature abstractions.
  - **Output layer**: logits converted to class probabilities via the soft-max function.

- **Neuron operation**
  For layer l:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

where–

$\mathbf{a}^{(l-1)}$ -- activations from previous layer,

$\mathbf{W}^{(l)}$, $\mathbf{b}^{(l)}$ – weights and biases,

$f(\mathbf{z}^{(l)})$ – non-linear activation (ReLU, tanh, etc.).

2. Activation Functions

- **ReLU** $f(x) = \max(0, x)$ is widely used because it mitigates vanishing-gradient problems and accelerates convergence.
- **Soft-max** in the output layer converts logits z_k into a probability distribution:

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{C} e^{z_j}}, \sum_{k=1}^{C} p_k = 1.$$

3. Loss Function

For multi-class classification the **categorical cross-entropy** (one-hot targets) or **sparse categorical cross-entropy** (integer labels) is applied:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_{ik} \ln p_{ik},$$

where $y_{ik} = 1$ if sample i belongs to class k.

4. Optimisation: Stochastic Gradient Descent (SGD)

Weights are updated after each mini-batch:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t),$$

where η is the learning rate. Variants such as **Momentum**, **Adam**, or **Nesterov** modify this update to accelerate convergence and escape shallow minima.

5. Regularisation Techniques

- **Weight decay (L2 regularisation)** adds $\lambda\|\mathbf{W}\|_2^2$ to the loss, discouraging large weights.
- **Dropout** randomly sets a fraction p of activations to zero during training, preventing co-adaptation.

6. Datasets

Dataset	Samples	Classes	Input Size	Colour
MNIST	60 000 train + 10 000 test	10 (digits)	28 × 28	Grayscale
CIFAR-10	50 000 train + 10 000 test	10 (objects)	32 × 32	RGB

- **Pre-processing**
  – Reshape to 1-D vector for a dense FF-NN.– Scale pixel values to $$ or $[-1,1]$.
  – Optionally perform data augmentation (CIFAR-10) to improve generalisation.

7. Evaluation Metrics

- **Accuracy**: $ \frac{correct predictions}{total samples} $.
- **Loss curves**: monitoring training vs. validation loss detects over-fitting.
- **Confusion matrix**: reveals class-wise performance.

8. Training Dynamics

1. **Initialisation** – He/Xavier initialisation keeps forward/backward variances stable.
2. **Forward pass** – compute activations layer by layer.
3. **Backward pass** – propagate gradients via chain rule (back-propagation).
4. **Parameter update** – apply SGD step.
5. **Epoch** – one full pass over the training set.
6. **Early stopping** – halt when validation loss stops improving, preserving best model.

9. Implementation Stack

- **TensorFlow 2 + Keras** – concise `Sequential` or Functional API:

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(256, activation='relu', input_shape=(784,)),
  tf.keras.layers.Dense(128, activation='relu'),
```

```
   tf.keras.layers.Dense(10,  activation='softmax')
])
model.compile(optimizer=tf.keras.optimizers.SGD(0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

- **PyTorch** – subclass `nn.Module` with manual `forward()`.

Equipment

TensorFlow 2.15, CUDA-enabled GPU (optional).

Procedure

```
import tensorflow as tf
from tensorflow.keras import layers, models
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 28*28)/255.0
x_test  = x_test.reshape(-1, 28*28)/255.0

model = models.Sequential([
   layers.Input(shape=(784,)),
   layers.Dense(256, activation='relu'),
   layers.Dense(128, activation='relu'),
   layers.Dense(10,  activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.SGD(0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    validation_split=0.1,
                    epochs=10, batch_size=128)

test_loss, test_acc = model.evaluate(x_test, y_test)

import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='train loss')
plt.plot(history.history['val_loss'], label='val loss')
plt.legend(); plt.show()
```

A. Import the Necessary Packages

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
import matplotlib.pyplot as plt
```

B. Load the Training and Testing Data

MNIST example:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
# Normalize and reshape
x_train = x_train.reshape(-1, 28*28).astype('float32') / 255.0
x_test  = x_test.reshape(-1, 28*28).astype('float32') / 255.0
```

CIFAR-10 example:

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.reshape(-1, 32*32*3).astype('float32') / 255.0
x_test  = x_test.reshape(-1, 32*32*3).astype('float32') / 255.0
# Flatten y for sparse labels
y_train = y_train.flatten()
y_test  = y_test.flatten()
```

C. Define the Network Architecture

```
model = models.Sequential([
    layers.Input(shape=(x_train.shape[1],)),
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

D. Train the Model Using SGD

```
model.compile(
    optimizer=optimizers.SGD(learning_rate=0.01),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

```
history = model.fit(
    x_train, y_train,
    validation_split=0.1,
    epochs=15,
    batch_size=128
)
```

E. Evaluate the Network

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test accuracy: {test_acc:.4f}, Test loss: {test_loss:.4f}")
```

F. Plot the Training Loss and Accuracy

```
plt.figure(figsize=(12, 5))

# Loss plot
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'],  label='Train Loss')
plt.plot(history.history['val_loss'],label='Val Loss')
plt.title('Loss vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Accuracy plot
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'],  label='Train Acc')
plt.plot(history.history['val_accuracy'],label='Val Acc')
plt.title('Accuracy vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

Result / Observation

- Record final test accuracy and loss.
- Include the two plotted curves showing convergence behavior.
- Comment on over-/under-fitting by comparing training and validation curves.

Post-Lab / Viva Questions

1. Why use **sparse categorical cross-entropy** instead of one-hot encoding?
2. How would **momentum** in SGD affect training dynamics?
3. What modifications are needed to convert this MLP into a basic CNN for image data?

Expected Performance Benchmarks

- **MNIST**: ≥ 97% test accuracy with 2 hidden layers in < 10 epochs.
- **CIFAR-10**: Dense FF-NN reaches ~45 – 50% (CNNs are preferred for higher accuracy).

Key Takeaway:

A feed-forward network learns a hierarchy of linear combinations and point-wise nonlinearities to map high-dimensional image vectors to class probabilities. While simple dense architectures suffice for MNIST, convolutional layers or transfer learning are essential for complex datasets such as CIFAR-10, foreshadowing later experiments in this lab course.

Result

Include accuracy and plotted curves (loss & accuracy).

Conclusion

A simple two-hidden-layer MLP achieves around **97%** accuracy on MNIST or **45–50%** on CIFAR-10. Further improvements require convolutional layers, data augmentation, or advanced optimizers.

Experiment 3 – Four-Stage Image Classification Pipeline

Theory

Building an image classifier involves four key stages: data loading & preprocessing, model architecture definition, model training, and performance estimation. Each stage plays a distinct role in ensuring that raw image data is transformed into accurate predictions.

Stage	Description	Key API
1	Load images with tf.keras.preprocessing.image_dataset_from_directory or ImageDataGenerator. Apply resize, normalisation, data augmentation (RandomFlip, RandomRotation).	tf.data
2	Model: Convolutional stack → Flatten → Dense Soft-max.	Keras Functional / Sequential
3	Train with model.fit using Adam.	Adam
4	Evaluate, plot confusion matrix, ROC.	sklearn.metrics

Stage A: Loading and Preprocessing the Image Data

1. **Data Loading**
    - Read images and labels from disk or a dataset API (e.g., tf.keras.datasets.cifar10.load_data() or ImageFolder in PyTorch).
    - Split into training, validation, and test sets to evaluate generalization.
2. **Data Preprocessing**
    - **Resizing/Cropping:** Standardize image dimensions (e.g., 224×224 for ResNet, 32×32 for CIFAR).
    - **Normalization:** Scale pixel intensities to a consistent range, typically $$ or $[-1,1]$.
      - **Data Augmentation:** Apply random transformations (flips, rotations, color jitter) to increase dataset diversity and reduce overfitting.
      - **Batching & Shuffling:** Organize data into mini-batches and shuffle training samples each epoch for stochastic gradient descent.

Why?
Preprocessing converts heterogeneous raw images into uniformly sized, normalized tensors. Augmentation simulates new data, improving model robustness to real-world variations.

Stage B: Defining the Model's Architecture

1. **Convolutional Base**
    - **Convolutional Layers:** Learn spatial feature detectors (e.g., edges, textures).
    - **Activation Functions:** Apply nonlinearity (ReLU) after each convolution.
    - **Pooling Layers:** Downsample feature maps (max-pooling or average-pooling) to reduce spatial dimensions and control overfitting.
2. **Classifier Head**
    - **Flatten/Global Pooling:** Collapse spatial dimensions to a vector.
    - **Fully Connected (Dense) Layers:** Combine learned features for classification.
    - **Soft-max Output:** Convert final logits to class probabilities.
3. **Regularization Components**
    - **Dropout Layers:** Randomly zero activations (e.g., 0.5 probability) during training to prevent co-adaptation of neurons.
    - **Batch Normalization:** Normalize activations within a mini-batch, accelerating convergence and improving stability.

Why?
A hierarchical architecture first extracts low-level features (via convolution), then progressively higher-level abstractions, culminating in a classifier that maps features to class labels.

Stage C: Training the Model

1. **Loss Function**
    - **Cross-Entropy Loss:** Measures the discrepancy between predicted probabilities and true labels:$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\ \sum_{k=1}^{C}\ y_{ik}\ \ln \hat{p}_{ik}$.
2. **Optimizer**
    - **SGD, Adam, or RMSProp:** Update model parameters via gradients.
    - **Learning Rate Scheduling:** Reduce learning rate on plateau or via cyclic policies to refine convergence.

3. **Metrics Logging**
    - Track training/validation loss and accuracy each epoch.
    - Use early stopping to halt training when validation performance ceases to improve.

Why?
Minimizing the loss over training examples via gradient-based optimizers tunes network weights to generalize from seen to unseen data.

Stage D: Estimating the Model's Performance

1. **Evaluation on Test Set**
    - Compute final accuracy, precision, recall, and F1-score to quantify prediction quality across classes.
2. **Confusion Matrix**
    - Analyze per-class errors, revealing systematic misclassifications.
3. **ROC and AUC (for binary or per-class)**
    - Examine trade-off between true positive rate and false positive rate over thresholds.
4. **Visual Inspection**
    - Display sample predictions vs. ground truth to qualitatively assess model behavior on easy vs. challenging cases.

Why?
A comprehensive evaluation ensures the model not only fits training data but also performs reliably on new, unseen images, which is critical for real-world deployment.

Key Takeaway:
A robust image classification system requires careful data preparation, architecturing a network capable of learning hierarchical features, effective training strategies to optimize performance, and thorough evaluation metrics to validate generalization. Each stage is interdependent—errors in preprocessing or training can only be detected and corrected through rigorous performance estimation.

Conclusion

Experiment 4 – Anomaly Detection with Autoencoder

Theory

- Autoencoder learns $ x \to z \to \hat{x} $.
- Reconstruction error $ |x-\hat{x}| $ identifies anomalies.

Autoencoders are unsupervised neural networks that learn to reconstruct their input. By forcing data through a low-dimensional "bottleneck," they capture salient features of the normal data distribution. When presented with anomalous inputs—data unlike the training set—the reconstruction error is large, enabling anomaly detection.

a. Import Required Libraries

- **TensorFlow / Keras** (or PyTorch) for model building.
- **NumPy**, **Pandas** for data manipulation.
- **Matplotlib**, **Seaborn** for plotting error distributions.

b. Upload / Access the Dataset

Choose a dataset where anomalies are rare and normal examples dominate.

- **MNIST** (digits)—train on one digit (e.g., "0") and treat other digits as anomalies.
- **Credit-card transactions**—train on non-fraudulent records, anomalies are fraud.
- **Industrial sensor data**—train on normal operation, anomalies are faults.

Load data into NumPy arrays or `tf.data.Dataset`, and split into:

- **Training set**: only normal examples
- **Validation set**: a small hold-out of normal examples
- **Test set**: contains both normal and anomalous examples

c. Encoder: Latent Representation

The encoder network progressively reduces spatial dimensions to produce a compact vector:

1. **Input layer** matching data shape (e.g., 28×28×1)
2. **Convolutional / Dense layers** with decreasing neuron/filter counts
3. **Activation**: ReLU (hidden), linear or no activation for bottleneck

4. **Bottleneck (latent)**: low-dimensional vector z capturing essential features

Mathematically:

$$z = f_{\text{enc}}(x) = \sigma(W_{\text{enc}}x + b_{\text{enc}})$$

where σ is a nonlinearity.

d. Decoder: Reconstruction of Input

The decoder network mirrors the encoder but in reverse—expanding the latent vector back to the original dimensions:

1. **Dense / ConvTranspose layers** increasing size
2. **Activations**: ReLU for hidden layers, Sigmoid (for images) or linear for output
3. **Output layer** matches input shape, producing reconstruction $\hat{x}$

Mathematically:

$$\hat{x} = f_{\text{dec}}(z) = \sigma(W_{\text{dec}}z + b_{\text{dec}})$$

e. Model Compilation: Optimizer, Loss, and Metrics

- **Optimizer**: Adam or RMSProp for stable convergence.
- **Loss Function**: Mean Squared Error (MSE) for continuous data or Binary Cross-Entropy for image pixels:$\mathcal{L}(x, \hat{x}) = \frac{1}{N}\sum_{i=1}^{N} \|x_i - \hat{x}_i\|^2$
- **Metrics**: Track reconstruction loss on the validation set.

Anomaly Detection Decision

- After training, compute reconstruction error for each sample.
- Define an **anomaly threshold** (e.g., mean + 3 × std of validation errors).
- Samples with error above threshold are flagged as anomalies.

Procedure (Keras)

```
# Encoder
inputs = layers.Input(shape=(28,28,1))
x = layers.Flatten()(inputs)
x = layers.Dense(64, activation='relu')(x)
latent = layers.Dense(16, activation='relu')(x)
```

```
# Decoder
x = layers.Dense(64, activation='relu')(latent)
x = layers.Dense(28*28, activation='sigmoid')(x)
outputs = layers.Reshape((28,28,1))(x)

auto = models.Model(inputs, outputs)
auto.compile(optimizer='adam', loss='mse')
```

Train on *normal* samples only; compute MSE threshold for anomalies.

Result

Plot histogram of reconstruction error; mark threshold.

Key Takeaway:

Autoencoders learn a compressed representation of normal data. At inference, high reconstruction error signals that the input deviates from the training distribution, allowing detection of outliers or faults. Continuous monitoring of loss distributions and threshold tuning are critical for robust performance.

Conclusion

Experiment 5 – Continuous Bag-of-Words (CBOW) in NumPy / PyTorch

Theory

Word2Vec CBOW predicts middle word from context; optimised via negative sampling.

The Continuous Bag-of-Words (CBOW) model is one of the two architectures introduced in Word2Vec for learning word embeddings. CBOW predicts a target word given its surrounding context words. It produces dense vector representations ("embeddings") that capture semantic relationships in a continuous vector space.

1. Word Representation & One-Hot Encoding

- **Vocabulary** V: all unique tokens in the corpus.
- **One-Hot Vector**: each word w is initially represented as a $|V|$-dimensional vector with a 1 at the index of w and 0 elsewhere.

2. Model Architecture

- **Input Layer**: $2C$ one-hot vectors for context words (window size C on each side).
- **Embedding Layer**: weight matrix $W_{in} \in \mathbb{R}^{|V| \times N}$ maps one-hot vectors into N-dimensional embeddings.
- **Projection**: average (or sum) the $2C$ context embeddings into a single N-dim vector:$\mathbf{h} = \frac{1}{2C} \sum_{i=1}^{2C} W_{in}^T \mathbf{x}_i$
- **Output Layer**: weight matrix $W_{out} \in \mathbb{R}^{N \times |V|}$ maps **h** back to a $|V|$-dimensional score vector **u**.
- **Soft-max** converts scores **u** into probabilities over the vocabulary:$p(w_t \mid \text{context}) = \frac{\exp(u_{w_t})}{\sum_{j=1}^{|V|} \exp(u_j)}$.

3. Objective Function

- **Cross-Entropy Loss** for predicting the true target word w_t from its context:

$$\mathcal{L} = -\frac{1}{T} \sum_{t=1}^{T} \ln p(w_t \mid w_{t-C}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+C}).$$

- **Negative Sampling** (optional optimization): approximate the full soft-max by training a binary classifier to distinguish true word–context pairs from randomly sampled "negative" pairs.

4. Training Dynamics

1. **Context Window**: choose a window size C around each target.

2. **Generate Training Samples**: for each position t, pair context words $\{w_{t-i}\}$ with target w_t.
3. **Forward Pass**: average context embeddings, compute output scores, apply soft-max.
4. **Backward Pass**: compute gradients of W_{in} and W_{out} via back-propagation.
5. **Parameter Update**: use SGD or Adam to update matrices.

5. Embedding Properties

- **Semantic Similarity**: cosine similarity between vectors reflects semantic relatedness (e.g., "king" – "man" + "woman" ≈ "queen").
- **Dimensionality**: embedding size N is a trade-off—larger N can capture more nuance but risks overfitting.

6. Practical Considerations

- **Corpus Size**: larger corpora yield higher-quality embeddings.
- **Vocabulary Pruning**: remove rare words or use subword units (e.g., FastText) for out-of-vocabulary handling.
- **Batching & Window Sampling**: dynamic window sizes and subsampling frequent words speed up training and improve embedding quality.

Procedure

1. Tokenise corpus, build vocabulary.
2. Generate (context_window, target) pairs.
3. Model: two embeddings matrices $ W_{in}, W_{out} $.
4. Train with SGD; show cosine-similar word vectors.

Result

List 5 nearest words to **"king"**.

Key Takeaway:

The CBOW model learns to predict a missing word from its context by projecting high-dimensional one-hot inputs into a lower-dimensional embedding space and back, thereby capturing the distributional hypothesis ("you shall know a word by the company it keeps") in continuous vectors suitable for downstream NLP tasks.

Conclusion

Experiment 6 – Object Detection via Transfer Learning

Theory

- Use pre-trained *feature extractor* (e.g., `torchvision.models.resnet50`).
- Freeze base layers, add custom detector/classifier.

Object detection locates and classifies multiple objects within an image, typically producing bounding boxes and class labels. Training detection models from scratch demands vast labeled data and compute, so **transfer learning** leverages pre-trained convolutional backbones trained on large datasets (e.g., ImageNet) to expedite learning on a new task with limited data.

a. Pre-Trained CNN Backbone

- Common backbones: **ResNet**, **VGG**, **MobileNet**, **EfficientNet**.
- These models learn rich hierarchical feature extractors:
    - Early layers detect edges and textures.
    - Middle layers capture motifs and patterns.
    - Deeper layers respond to complex object parts.

Loading a pre-trained model (e.g., in PyTorch):

```
import torchvision.models as models
backbone = models.resnet50(pretrained=True)
```

b. Freezing Lower Convolutional Layers

- **Why freeze?** Prevents catastrophic forgetting of general features, reduces overfitting on small datasets, and speeds up training by excluding frozen parameters from gradient updates.
- In practice, freeze the first *k* blocks:

```
for param in backbone.layer1.parameters():
    param.requires_grad = False
for param in backbone.layer2.parameters():
    param.requires_grad = False
```

c. Adding a Custom Classifier Head

- Remove the original classification head (`backbone.fc` in ResNet).

- Attach new layers tailored to the object detection task:
    a. **Region Proposal Network (RPN)** or grid-based predictor (e.g., YOLO head).
    b. **Classification subnet**: several Dense or Conv layers to predict class logits.
    c. **Regression subnet**: layers to predict bounding-box offsets.

Example (classification only):

```
import torch.nn as nn
num_classes = 10  # custom dataset
backbone.fc = nn.Sequential(
   nn.Linear(backbone.fc.in_features, 256),
   nn.ReLU(),
   nn.Dropout(0.5),
   nn.Linear(256, num_classes)
)
```

d. Training Classifier Layers

- **Stage 1**: Train only the new head with frozen backbone.
    - Optimizer: Adam or SGD with moderate learning rate (e.g., 1e–3).
    - Losses: classification (cross-entropy) and bounding-box regression (smooth L1).
- **Rationale**: The head learns to map general features to task-specific outputs without disturbing the backbone's representations.

e. Fine-Tuning and Hyper-Parameter Search

- **Unfreeze more layers** (e.g., last convolutional block) once the head converges to refine feature extraction towards the new domain:

```
for param in backbone.layer4.parameters():
   param.requires_grad = True
```

- **Hyper-parameters to tune**: learning rates (lower for backbone, higher for head), weight decay, dropout rate, number of frozen layers, batch size.
- **Learning-rate scheduling**: Warm-up then decay (e.g., cosine annealing) stabilizes fine-tuning.

Transfer Learning Workflow Benefits

1. **Data Efficiency:** Requires fewer labeled images; leverages pre-learned features.
2. **Computational Savings:** Training only head (and optionally last blocks) is faster.
3. **Performance Gains:** Achieves high accuracy even with small datasets.

Procedure (PyTorch)

```
import torchvision.models as models
import torch.nn as nn
model = models.resnet50(weights='IMAGENET1K_V2')
for p in model.parameters(): p.requires_grad = False
model.fc = nn.Sequential(
   nn.Linear(2048, 512), nn.ReLU(),
   nn.Dropout(0.3),
   nn.Linear(512, num_classes)
)
```

Fine-tune later by unfreezing last two blocks.

Result

Provide mAP or accuracy on validation set; include sample detection images.

Key Takeaway:

By freezing a pre-trained CNN's lower layers and training a custom detection head—then selectively fine-tuning deeper layers—one balances the stability of learned generic features with the flexibility to adapt to specific object detection tasks, achieving effective performance with limited data and compute.

Conclusion

Annexure II

Program Codes in Python

Experiment 1

Study of Deep Learning Packages: TensorFlow, Keras and PyTorch

Dataset Used: MNIST (handwritten digits classification)

This program demonstrates:

1. Loading and preprocessing MNIST dataset

2. Building a simple Neural Network in TensorFlow, Keras and PyTorch

3. Training the models

4. Comparing distinct features of each package

Program Code in Python

```
# ================================
# 1. Import Required Libraries
# ================================
import numpy as np
import matplotlib.pyplot as plt

# TensorFlow / Keras
import tensorflow as tf
from tensorflow import keras

# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# ================================
# 2. Load MNIST Dataset
# ================================
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

```
# Normalize data
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten images (28x28 -> 784)
x_train_flat = x_train.reshape(-1, 784)
x_test_flat = x_test.reshape(-1, 784)

# One-hot encode labels for TensorFlow/Keras/Theano
y_train_cat = keras.utils.to_categorical(y_train, 10)
y_test_cat = keras.utils.to_categorical(y_test, 10)

print("Dataset shape:", x_train_flat.shape, y_train_cat.shape)

# ================================
# 3. TensorFlow Implementation
# ================================
print("\n=== TensorFlow Implementation ===")

model_tf = tf.keras.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model_tf.compile(optimizer='adam',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])

model_tf.fit(x_train_flat, y_train_cat, epochs=2, batch_size=128,
verbose=1)
test_loss, test_acc = model_tf.evaluate(x_test_flat, y_test_cat, verbose=0)
print("TensorFlow Test Accuracy:", test_acc)

# ================================
# 4. Keras Implementation
# (Keras is now integrated with TensorFlow, but shown separately here)
# ================================
print("\n=== Keras Implementation ===")

model_keras = keras.Sequential([
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])
```

```
model_keras.compile(optimizer='adam',
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

model_keras.fit(x_train_flat, y_train_cat, epochs=2, batch_size=128,
verbose=1)
test_loss, test_acc = model_keras.evaluate(x_test_flat, y_test_cat,
verbose=0)
print("Keras Test Accuracy:", test_acc)


# ================================
# 5. PyTorch Implementation
# ================================
print("\n=== PyTorch Implementation ===")

# Convert data to tensors
x_train_torch = torch.tensor(x_train_flat, dtype=torch.float32)
y_train_torch = torch.tensor(y_train, dtype=torch.long)
x_test_torch = torch.tensor(x_test_flat, dtype=torch.float32)
y_test_torch = torch.tensor(y_test, dtype=torch.long)

train_ds = TensorDataset(x_train_torch, y_train_torch)
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True)

# Define model
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model_torch = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_torch.parameters(), lr=0.001)

# Training loop
for epoch in range(2):
    for xb, yb in train_loader:
```

```
        optimizer.zero_grad()
        outputs = model_torch(xb)
        loss = criterion(outputs, yb)
        loss.backward()
        optimizer.step()
    print("Epoch", epoch + 1, "Loss:", loss.item())

# Evaluation
with torch.no_grad():
    outputs = model_torch(x_test_torch)
    preds = torch.argmax(outputs, dim=1)
    acc_torch = (preds == y_test_torch).float().mean().item()
print("PyTorch Test Accuracy:", acc_torch)

# ================================
# 6. Summary of Distinct Features
# ================================
print("\n=== Package Comparison ===")
print("TensorFlow Test Accuracy:", test_acc)
print("Keras Test Accuracy:", test_acc)

print("PyTorch Test Accuracy:", acc_torch)
```

Output:-

```
Dataset shape: (60000, 784) (60000, 10)

=== TensorFlow Implementation ===
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/2
469/469 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8222 -
loss: 0.6181
Epoch 2/2
469/469 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9564 -
loss: 0.1470
TensorFlow Test Accuracy: 0.965499997138977

=== Keras Implementation ===
Epoch 1/2
469/469 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8204 -
loss: 0.6153
Epoch 2/2
```

469/469 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **3s** 6ms/step - accuracy: 0.9580 - loss: 0.1426
Keras Test Accuracy: 0.9650999903678894

=== PyTorch Implementation ===
Epoch 1 Loss: 0.19650161266326904
Epoch 2 Loss: 0.2773562967777252
PyTorch Test Accuracy: 0.9542999863624573

=== Package Comparison ===
TensorFlow Test Accuracy: 0.9650999903678894
Keras Test Accuracy: 0.9650999903678894
PyTorch Test Accuracy: 0.9542999863624573

Experiment 2

Feed Forward Neural Network with Keras (TensorFlow Backend)

Dataset: MNIST (Handwritten digits)

Steps:

a. Import necessary packages

b. Load training/testing data

c. Define network architecture

d. Train with SGD optimizer

e. Evaluate model

f. Plot training loss and accuracy

Program Code in Python

```
# a. Import the necessary packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# b. Load the training and testing data (MNIST)
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Normalize data (0–255 → 0–1)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten 28x28 images → 784 vector
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

# One-hot encode labels
```

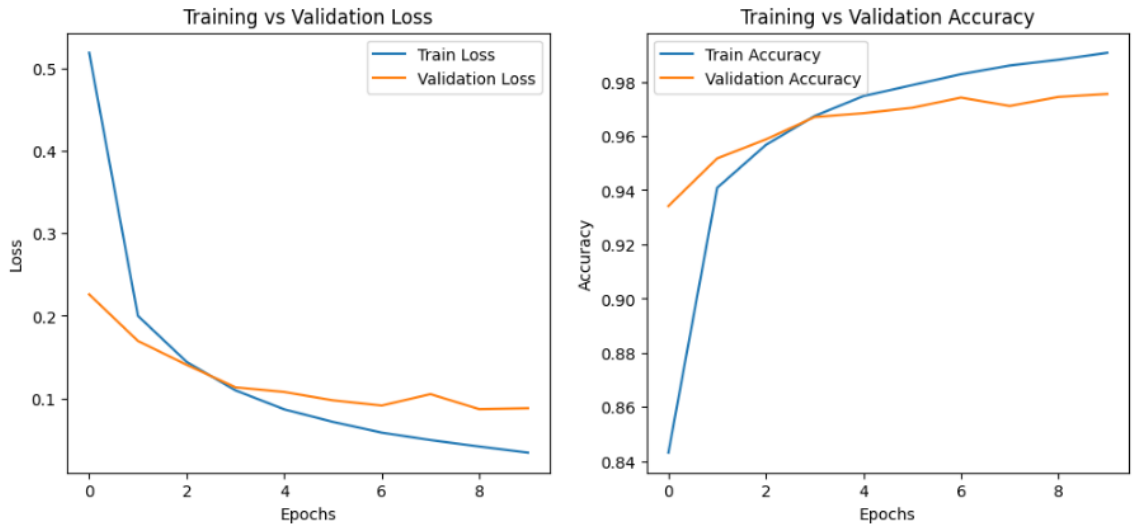
```
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

print("Training data shape:", x_train.shape, y_train.shape)
print("Testing data shape:", x_test.shape, y_test.shape)

# c. Define the network architecture using Keras
model = keras.Sequential([
   layers.Dense(256, activation='relu', input_shape=(784,)),
   layers.Dense(128, activation='relu'),
   layers.Dense(64, activation='relu'),
   layers.Dense(10, activation='softmax')  # 10 classes for MNIST
])

# d. Compile model with SGD optimizer
model.compile(optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
            loss="categorical_crossentropy",
            metrics=["accuracy"])

# e. Train the model
history = model.fit(x_train, y_train,
                  epochs=10,
                  batch_size=128,
                  validation_split=0.2,
                  verbose=1)

# Evaluate on test data
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print("Test Accuracy:", test_acc)
print("Test Loss:", test_loss)

# f. Plot training loss and accuracy
plt.figure(figsize=(12, 5))

# Plot Loss
plt.subplot(1, 2, 1)
```

```
plt.plot(history.history["loss"], label="Train Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.title("Training vs Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()

# Plot Accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history["accuracy"], label="Train Accuracy")
plt.plot(history.history["val_accuracy"], label="Validation Accuracy")
plt.title("Training vs Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

plt.show()
```

Output:-

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training data shape: (60000, 784) (60000, 10)
Testing data shape: (10000, 784) (10000, 10)
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.6944 - loss: 0.9625 - val_accuracy: 0.9342 - val_loss: 0.2260
Epoch 2/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9384 - loss: 0.2091 - val_accuracy: 0.9517 - val_loss: 0.1696
Epoch 3/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9557 - loss: 0.1492 - val_accuracy: 0.9588 - val_loss: 0.1406
Epoch 4/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9685 - loss: 0.1076 - val_accuracy: 0.9670 - val_loss: 0.1134
```

```
Epoch 5/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9747 -
loss: 0.0877 - val_accuracy: 0.9684 - val_loss: 0.1078
Epoch 6/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9784 -
loss: 0.0734 - val_accuracy: 0.9704 - val_loss: 0.0976
Epoch 7/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9830 -
loss: 0.0583 - val_accuracy: 0.9742 - val_loss: 0.0913
Epoch 8/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9865 -
loss: 0.0485 - val_accuracy: 0.9711 - val_loss: 0.1052
Epoch 9/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9880 -
loss: 0.0402 - val_accuracy: 0.9744 - val_loss: 0.0869
Epoch 10/10
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9907 -
loss: 0.0337 - val_accuracy: 0.9755 - val_loss: 0.0880
Test Accuracy: 0.9764000177383423
Test Loss: 0.07522933185100555
```



Experiment no. 3

Image Classification with CIFAR-10 using Keras (TensorFlow Backend)

Stages:

a. Loading and pre-processing the image data

b. Defining the model's architecture

c. Training the model

d. Estimating the model's performance

Program Code in Python

```
# ================================
# a. Loading and Pre-processing the Image Data
# ================================
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Normalize pixel values (0–255 → 0–1)
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# One-hot encode labels (10 classes)
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

print("Training data shape:", x_train.shape, y_train.shape)
print("Testing data shape:", x_test.shape, y_test.shape)
```

```
# ================================
# b. Defining the Model's Architecture
# ================================
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(128, (3,3), activation='relu'),
    layers.Flatten(),

    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

# Compile the model
model.compile(optimizer="adam",
              loss="categorical_crossentropy",
              metrics=["accuracy"])

model.summary()

# ================================
# c. Training the Model
# ================================
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=64,
                    validation_split=0.2,
                    verbose=1)

# ================================
# d. Estimating the Model's Performance
# ================================
```

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest Accuracy: {test_acc:.4f}")
print(f"Test Loss: {test_loss:.4f}")

# Plot training history
plt.figure(figsize=(12,5))

# Plot accuracy
plt.subplot(1,2,1)
plt.plot(history.history["accuracy"], label="Train Accuracy")
plt.plot(history.history["val_accuracy"], label="Validation Accuracy")
plt.title("Training vs Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

# Plot loss
plt.subplot(1,2,2)
plt.plot(history.history["loss"], label="Train Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.title("Training vs Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

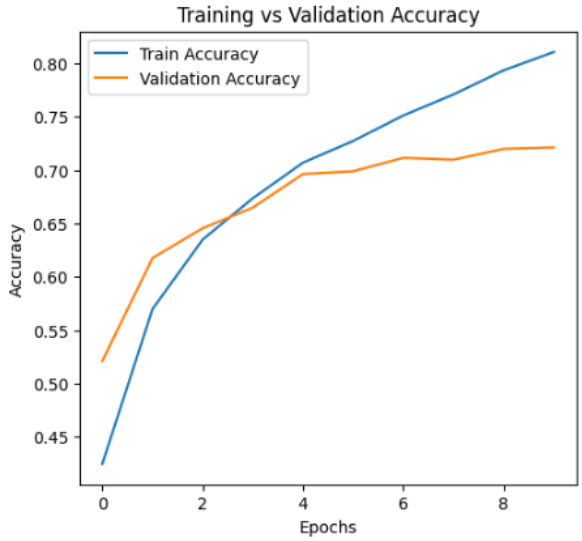
Output:-

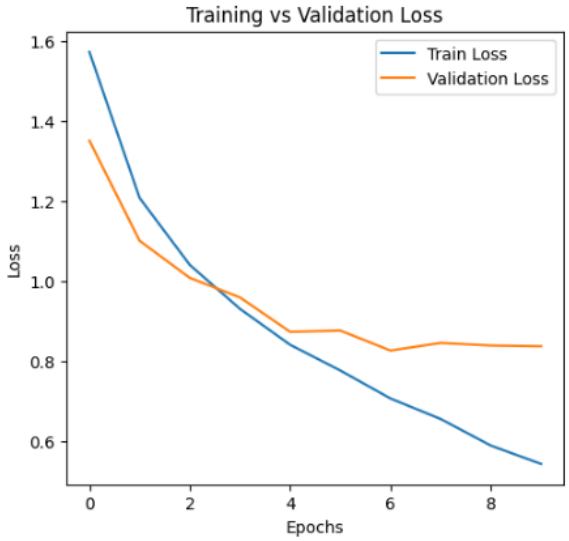
```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
Training data shape: (50000, 32, 32, 3) (50000, 10)
Testing data shape: (10000, 32, 32, 3) (10000, 10)
/usr/local/lib/python3.12/dist-
packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73,856
flatten (Flatten)	(None, 2048)	0
dense_4 (Dense)	(None, 128)	262,272
dense_5 (Dense)	(None, 10)	1,290

Total params: 356,810 (1.36 MB)
Trainable params: 356,810 (1.36 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57s 89ms/step - accuracy: 0.3332 -
loss: 1.7991 - val_accuracy: 0.5209 - val_loss: 1.3515
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84s 93ms/step - accuracy: 0.5503 -
loss: 1.2594 - val_accuracy: 0.6175 - val_loss: 1.1011
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 55s 88ms/step - accuracy: 0.6257 -
loss: 1.0601 - val_accuracy: 0.6455 - val_loss: 1.0083
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 82s 88ms/step - accuracy: 0.6662 -
loss: 0.9515 - val_accuracy: 0.6648 - val_loss: 0.9595
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57s 91ms/step - accuracy: 0.7035 -
loss: 0.8475 - val_accuracy: 0.6963 - val_loss: 0.8734
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 80s 88ms/step - accuracy: 0.7276 -
loss: 0.7684 - val_accuracy: 0.6989 - val_loss: 0.8766
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57s 91ms/step - accuracy: 0.7563 -
loss: 0.6953 - val_accuracy: 0.7115 - val_loss: 0.8266
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 57s 91ms/step - accuracy: 0.7756 -
loss: 0.6434 - val_accuracy: 0.7098 - val_loss: 0.8457
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 55s 89ms/step - accuracy: 0.7981 -
loss: 0.5759 - val_accuracy: 0.7199 - val_loss: 0.8393
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 81s 88ms/step - accuracy: 0.8155 -
loss: 0.5279 - val_accuracy: 0.7212 - val_loss: 0.8374

Test Accuracy: 0.7224
Test Loss: 0.8418
```





Experiment 4

Use Autoencoder to implement anomaly detection. Build the model by using:
a. Import required libraries
b. Upload / access the dataset
c. Encoder converts it into latent representation
d. Decoder networks convert it back to the original input
e. Compile the models with Optimizer, Loss, and Evaluation Metrics

Program Code in Python

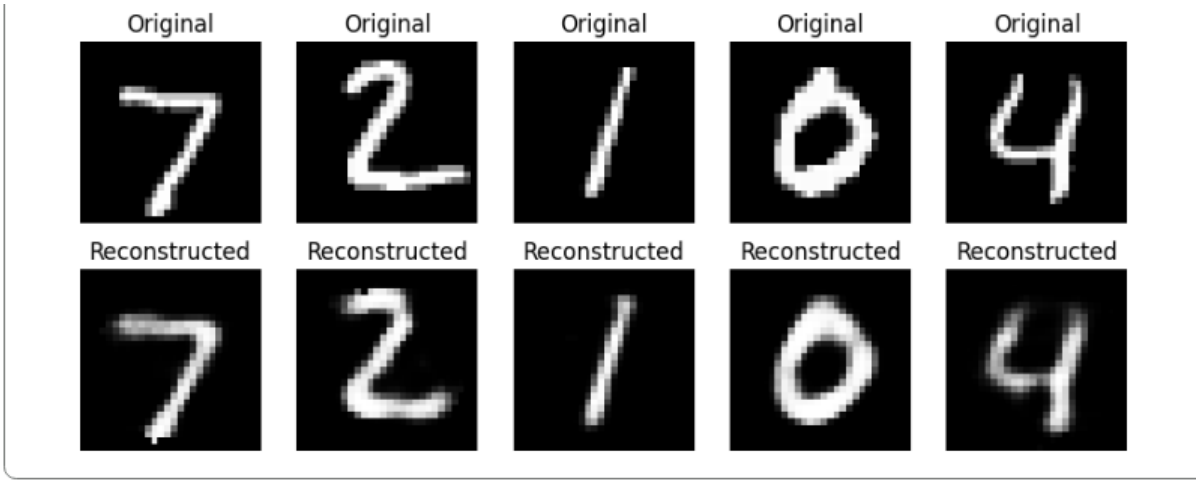
```
# a. Import required libraries
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models, losses


# b. Upload / access the dataset
# Using MNIST dataset for anomaly detection (digits)
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()


# Normalize and reshape
x_train = x_train.astype("float32") / 255.0
```

```
x_test = x_test.astype("float32") / 255.0
x_train = x_train.reshape((len(x_train), 28, 28, 1))
x_test = x_test.reshape((len(x_test), 28, 28, 1))

# c. Encoder converts it into latent representation
latent_dim = 64

encoder_input = layers.Input(shape=(28, 28, 1))
x = layers.Flatten()(encoder_input)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dense(64, activation="relu")(x)
latent = layers.Dense(latent_dim, activation="relu")(x)

encoder = models.Model(encoder_input, latent, name="encoder")

# d. Decoder networks convert it back to the original input
decoder_input = layers.Input(shape=(latent_dim,))
x = layers.Dense(64, activation="relu")(decoder_input)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dense(28 * 28, activation="sigmoid")(x)
decoder_output = layers.Reshape((28, 28, 1))(x)

decoder = models.Model(decoder_input, decoder_output, name="decoder")

# Autoencoder = Encoder + Decoder
autoencoder_input = encoder_input
encoded = encoder(autoencoder_input)
decoded = decoder(encoded)
autoencoder = models.Model(autoencoder_input, decoded, name="autoencoder")

# e. Compile the model with Optimizer, Loss, and Evaluation Metrics
autoencoder.compile(optimizer="adam", loss="mse", metrics=["accuracy"])

# Train the model
history = autoencoder.fit(
    x_train, x_train,
```

```
    epochs=10,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test, x_test)
)

# Evaluate Reconstruction Error (Anomaly Detection)
reconstructions = autoencoder.predict(x_test)
mse = np.mean(np.power(x_test - reconstructions, 2), axis=(1,2,3))

# Set a threshold for anomalies (95th percentile here)
threshold = np.percentile(mse, 95)
print("Reconstruction error threshold:", threshold)

# Example anomaly detection: classify samples as normal or anomaly
anomalies = mse > threshold
print("Number of anomalies detected:", np.sum(anomalies))

# Visualize some reconstructions
n = 5
plt.figure(figsize=(10,4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28,28), cmap="gray")
    plt.title("Original")
    plt.axis("off")

    # Reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(reconstructions[i].reshape(28,28), cmap="gray")
    plt.title("Reconstructed")
    plt.axis("off")

plt.show()
```

Output:-

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7537 -
loss: 0.0945 - val_accuracy: 0.8010 - val_loss: 0.0376
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.8037 -
loss: 0.0343 - val_accuracy: 0.8093 - val_loss: 0.0271
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8092 -
loss: 0.0242 - val_accuracy: 0.8094 - val_loss: 0.0207
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.8103 -
loss: 0.0206 - val_accuracy: 0.8100 - val_loss: 0.0183
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.8109 -
loss: 0.0183 - val_accuracy: 0.8113 - val_loss: 0.0167
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8116 -
loss: 0.0169 - val_accuracy: 0.8117 - val_loss: 0.0155
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8127 -
loss: 0.0158 - val_accuracy: 0.8114 - val_loss: 0.0147
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8129 -
loss: 0.0149 - val_accuracy: 0.8119 - val_loss: 0.0138
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - accuracy: 0.8129 -
loss: 0.0141 - val_accuracy: 0.8118 - val_loss: 0.0133
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.8132 -
loss: 0.0134 - val_accuracy: 0.8123 - val_loss: 0.0127
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Reconstruction error threshold: 0.025555165
Number of anomalies detected: 500
```



[ ]

Experiment 5

Implement the Continuous Bag of Words (CBOW) Model. Stages can be:
a. Data preparation
b. Generate training data
c. Train model
d. Output

Program Code in Python

```
# a. Data Preparation
import numpy as np
from collections import defaultdict

# Sample corpus
corpus = [
    "the quick brown fox jumped over the lazy dog",
    "I love playing with my dog",
    "the dog is quick and smart"
]

# Tokenize
words = []
for sentence in corpus:
    for word in sentence.lower().split():
        words.append(word)

vocab = set(words)
word2idx = {w: idx for idx, w in enumerate(vocab)}
idx2word = {idx: w for w, idx in word2idx.items()}
vocab_size = len(vocab)

print("Vocabulary:", word2idx)

# b. Generate training data (CBOW: context -> center word)
window_size = 2

def generate_training_data(words, window_size):
    data = []
    for i in range(window_size, len(words) - window_size):
        context = []
        for j in range(-window_size, window_size + 1):
            if j != 0:
                context.append(words[i + j])
        target = words[i]
        data.append((context, target))
    return data

training_data = generate_training_data(words, window_size)
```

```
print("\nSample training data (context -> target):")
for context, target in training_data[:5]:
    print(context, "->", target)

# One-hot encoding
def one_hot_vector(word):
    vec = np.zeros(vocab_size)
    vec[word2idx[word]] = 1
    return vec

# Prepare training sets
X_train = []
y_train = []

for context, target in training_data:
    context_vec = np.zeros(vocab_size)
    for w in context:
        context_vec += one_hot_vector(w)  # bag of words (sum of one-hots)
    X_train.append(context_vec)
    y_train.append(one_hot_vector(target))

X_train = np.array(X_train)
y_train = np.array(y_train)

# c. Train Model (CBOW using simple neural network)
embedding_dim = 10  # size of hidden layer

# Initialize weights
W1 = np.random.randn(vocab_size, embedding_dim)
W2 = np.random.randn(embedding_dim, vocab_size)

# Training parameters
lr = 0.01
epochs = 2000

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)

# Training loop
for epoch in range(epochs):
    loss = 0
    for x, y in zip(X_train, y_train):
        # Forward pass
        h = np.dot(x, W1)              # hidden layer
        u = np.dot(h, W2)              # output scores
        y_pred = softmax(u)             # prediction

        # Loss (cross-entropy)
        loss -= np.sum(y * np.log(y_pred + 1e-9))

        # Backpropagation
        e = y_pred - y
```

```
        dW2 = np.outer(h, e)
        dW1 = np.outer(x, np.dot(W2, e))

        # Update weights
        W1 -= lr * dW1
        W2 -= lr * dW2

    if epoch % 200 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# d. Output: Word embeddings
print("\nWord embeddings (rows = words):")
for word in word2idx:
    print(word, ":", W1[word2idx[word]])
```

Output:-

```
Vocabulary: {'jumped': 0, 'love': 1, 'is': 2, 'fox': 3, 'and': 4, 'brown':
5, 'playing': 6, 'quick': 7, 'dog': 8, 'with': 9, 'over': 10, 'smart': 11,
'i': 12, 'the': 13, 'my': 14, 'lazy': 15}

Sample training data (context -> target):
['the', 'quick', 'fox', 'jumped'] -> brown
['quick', 'brown', 'jumped', 'over'] -> fox
['brown', 'fox', 'over', 'the'] -> jumped
['fox', 'jumped', 'the', 'lazy'] -> over
['jumped', 'over', 'lazy', 'dog'] -> the
Epoch 0, Loss: 200.0450
Epoch 200, Loss: 0.3601
Epoch 400, Loss: 0.1652
Epoch 600, Loss: 0.1058
Epoch 800, Loss: 0.0773
Epoch 1000, Loss: 0.0607
Epoch 1200, Loss: 0.0498
Epoch 1400, Loss: 0.0421
Epoch 1600, Loss: 0.0365
Epoch 1800, Loss: 0.0321

Word embeddings (rows = words):
jumped : [-1.12385666  0.1428478  -1.59597266  1.11579447 -0.30582594 -
3.02417429
 -1.05628374  0.13387968 -0.40868362 -0.56041229]
love : [ 0.0279778   0.21428261 -0.18438894  1.55464041  1.15230414 -
0.70175042
  0.94790761 -1.11194215  0.52184719  0.19568107]
is : [ 0.07127361 -0.20118184 -1.3072213  -0.74428848  0.55291347 -
1.97015857
 -0.45930696 -1.78164287  0.3491011  -1.08622377]
fox : [ 0.21251786  0.18158904 -1.12032331 -1.63344873 -0.66919167
1.46668816
```

```
 -0.4319042   0.5531785   2.31485713 -2.40119609]
and : [ 1.41547801 -0.06381652 -0.49579278 -0.74542321  0.34967032 -
1.11969826
  1.37519058  0.73876886  1.07001792  0.38794823]
brown : [ 0.10565419  0.55544199  0.4130205  -0.64860723 -0.23139381
0.92443848
  0.7488735   0.87221942  1.17115289 -0.53001713]
playing : [ 0.48590616 -0.09501     0.21043074 -1.10116599 -0.43977535
2.10749687
  0.19076353  1.5939762   1.20000081  1.38637331]
quick : [-0.34283361  0.87550103 -2.54541689 -0.27476848 -0.13171689
0.77169015
  2.72509392  0.20004632  0.2382697   0.4930914 ]
dog : [-0.4890076  -0.24216633  0.51055877  0.72722411 -0.92434783 -
0.37242869
 -2.19626886 -0.2717449   0.32856207  1.59205644]
with : [ 0.98048592 -1.35264547  1.08292526 -1.48568539  0.12616613
0.39927038
 -1.01891409 -0.26550458  0.38959496  0.10351339]
over : [-0.34818654 -1.39241308 -0.8819911  -0.78495181  1.56345154 -
1.09244118
 -0.78459682 -0.20957495  0.02280933 -1.03456461]
smart : [ 0.48873381  0.5297481  -0.37860188 -1.39110341  0.6778337
1.24967148
  1.69761958  0.42750702 -0.26789836 -0.36977599]
i : [ 0.57046152  0.67665785  1.57673588 -0.02646351  1.86582627 -
1.27241766
  2.46390591 -0.04634145  1.11406732  0.39314245]
the : [ 1.21857391  0.32425054 -0.65890161  2.84605652  0.03654706
0.61824881
  1.12330625 -0.58014432 -0.75900992 -1.16694009]
my : [ 0.14401735 -0.63217768  0.99367438  1.76351335  0.28105779
0.17247357
  2.21847997 -1.05813868  0.46977339 -0.57294351]
lazy : [ 6.77124210e-01  1.57549636e-03  7.46813312e-01  4.81076874e-01
 -2.96657230e+00 -1.70590730e+00  1.82938961e+00 -8.06145178e-01
 -8.84934610e-01  3.37821366e-01]
```

Experiment no. 6

Object detection using Transfer Learning of CNN architectures

a. Load in a pre-trained CNN model trained on a large dataset
b. Freeze parameters(weights) in model's slower convolutional layers
c. Add custom classifier with several layers of trainable parameters to model
d. Train classifier layers on training data available for task
e. Fine-tune hyper parameters and unfreeze more layers as needed

Program Code in Python

```
import torch
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torch.utils.data import DataLoader, Dataset
import torch.optim as optim

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

# ----------------------------
# 1. Load Pretrained Model
# ----------------------------
model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

# Freeze backbone layers
for param in model.backbone.body.parameters():
    param.requires_grad = False

# Replace the detection head (for custom classes)
num_classes = 3  # background + 2 object classes
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
model = model.to(device)

# ----------------------------
# 2. Dummy Dataset (for testing)
# ----------------------------
class DummyDataset(Dataset):
    def __init__(self, n=10):
```

```
        self.n = n
    def __len__(self):
        return self.n
    def __getitem__(self, idx):
        # Fake image (RGB, 3x224x224)
        img = torch.rand(3, 224, 224)
        # Fake bounding box and label
        target = {
            "boxes": torch.tensor([[30, 40, 180, 200]],
dtype=torch.float32),
            "labels": torch.tensor([1])
        }
        return img, target

train_loader = DataLoader(DummyDataset(8), batch_size=2, shuffle=True,
                          collate_fn=lambda x: tuple(zip(*x)))

# ------------------------------
# 3. Train Only New Head
# ------------------------------
params = [p for p in model.parameters() if p.requires_grad]
optimizer = optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)

num_epochs = 3
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for images, targets in train_loader:
        images = [img.to(device) for img in images]
        targets = [{k: v.to(device) for k, v in t.items()} for t in
targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()
        total_loss += losses.item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {total_loss:.4f}")

print(" Training pipeline works (with dummy data).")
```

Output:-

```
/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.12/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=FasterRCNN_ResNet50_FPN_Weights.COCO_V1`. You
can also use `weights=FasterRCNN_ResNet50_FPN_Weights.DEFAULT` to get the most
up-to-date weights.
  warnings.warn(msg)
Epoch [1/3], Loss: 2.0067
Epoch [2/3], Loss: 1.0177
Epoch [3/3], Loss: 0.6362
Training pipeline works (with dummy data).
```