Lab 1 - Introduction.

Google Colab

-> free cloud based coding environment by google.

Why?

write & run code in python in it.
Supports GPU & TPU
trains dl models faster.

Jupyter Notebook

- open source tool where we can write code, explain and get op.
- best for data analysis, visualization.
- runs on your PC not online.

TensorFlow

- dl framework developed by google.
- handles complex neural network building, training & deployment.
- supports both low & high leve API.

Keras

- high level API built on top of tensorflow.
- easier & faster than TF.
- Simple syntax.

PyTorch

- dl framework developed by Meta.
- flexible
- debug is made easier.

Numpy
- Handling arrays & mathematical operations.
- Base for most ML libraries.

Pandas
- Data Analysis & Manipulation

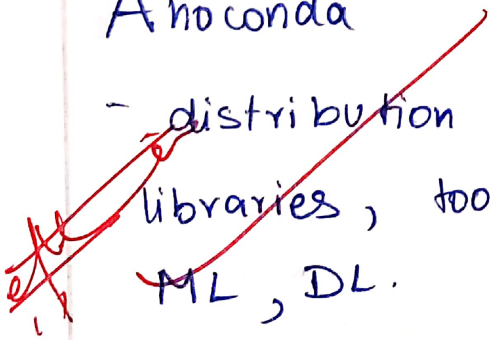
Matplotlib
- Plotting graphs & visualizing data.

Scikit-learn.
- Traditional ML model (SVM, DT) & preprocessing.

OpenCV.
- Computer vision task (image & video processing).

Hugging Face Transformers.
- NLP tasks & models like BERT, GPT

Anaconda
- distribution of Python bundled with many libraries, tools & environments used in ML, DL.



Lab 2: Implement A Classifier Using an Open Source Data Set.

Aim:

To implement a classification model using the SVM algorithm on the IRIS dataset and evaluate its performance using accuracy metrics.

Objectives:

1. To understand & apply the SVM algorithm for classification.
2. Open source dataset (Iris)
3. Preprocess & split into train & test tests.
4. To train SVM model & make predictions.
5. To evaluate the model's performance using metrics like (accuracy, precision, recall, F1-Score, confusion matrix).

Pseudo Code:

1. Import necessary libraries. (Sklearn)
2. Load the Iris data set
3. split the dataset into training & testing. (80% train, 20% test)
4. Initialize the SVM classifier.
5. Train the classifier using training data.
6. Predic the O/P. on test data.
7. Evaluate Performance.
8. Display the results.

About the dataset.

features ⇒ Sepal length, sepal width, petal length, petal width.

Target Label	Flower Species
0	Setosa
1	Versicolor
2	Virginica.

Data size

No. of samples - 150

No. of features - 4.

No. of classes - 3

Samples per class - 50 each.

Why SVM?

→ best for linearly seperable dataset. (Setosa)
RBF for versicolor & Virginica.

Observation.

Class	Precision	Recal	f1-score
Setosa	1.00	1.00	1.00
versicolor	1.00	0.93	0.97.
Virginica	0.93	1.00	0.97.

Feature Scaling & switching to an RBF Kernel

Accuracy: 97.78%.

Result: SVM Classifier was used on the dataset IRIS and was implemented successfully.

Hanishkaa01 Add files via upload d27b4cd · 7 minutes ago

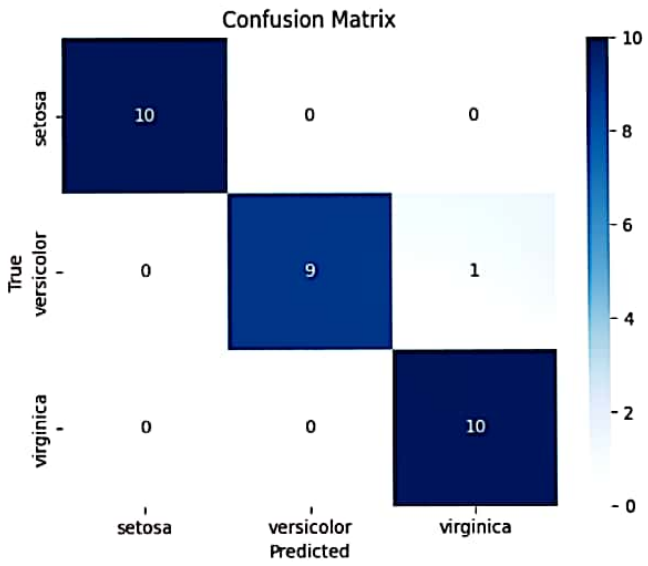
159 lines (159 loc) · 30 KB

Preview Code Blame Raw

In [14]:

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Standardize features
sc = StandardScaler()
X = sc.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Hyperparameter tuning for RBF SVM
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.01, 0.1, 1, 10],
    'kernel': ['rbf']
}
grid = GridSearchCV(SVC(), param_grid, cv=5)
grid.fit(X_train, y_train)
best_model = grid.best_estimator_

print("Best Parameters:", grid.best_params_)

# Predict on test set
y_pred = best_model.predict(X_test)

# Evaluation
acc = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {acc*100:.2f}%\n")

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, cmap="Blues", fmt="d",
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()

print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

```
Best Parameters: {'C': 1, 'gamma': 0.1, 'ke
rnel': 'rbf'}
Test Accuracy: 96.67%
```



Lab 3: Study of the Classifiers with Respect to Statistical Parameters.

Aim:

To study and compare the performance of 2 classification algorithms SVM & Logistic Regression on IRIS dataset using statistical evaluation metrics.

Objectives:

1. To implement SVM & Logistic Regression Classifiers
2. To apply both models on the same dataset.
3. To evaluate & compare their performance. Using Accuracy, Precision, Recall, f1-score.

Data Set Used:

- IRIS
- Sklearn.
- No. of samples : 150.
  No. of features : 4
  No. of Classes : 3.

Pseudo Code.

1. Import necessary libraries & load the Iris dataset.
2. Scale the data set using Standard Scaler.
3. Split the dataset into training & testing sets.
4. Initilize both SVM & Logistic Regression Classifiers.
5. Train both Classifiers on training data.

What is statistical Parameters?

- They are numerical values that describe characteristics of a dataset.
- In ml, it is referred to performance metrics Caccuracy, precision, recall, f1-score.

Accuracy

- no. of correctly predicted samples out of total
= $\frac{\text{No. of correct predictions}}{\text{Total no. of predictions.}}$

Precisions

Proportion of true positive out of all positive predictions made by model.

= $\frac{\text{True Positives}}{\text{True Positives + False Positives}}$

(High Prec
less. Fals

Recall / Sensitivity / True (+)ve Rate.

- proportions of true (+)ve out of all actu (+)ves.

= $\frac{\text{True Positives}}{\text{True Positive + False (Negative.}}$

F1-Score

harmonic mean of precision and recall.

$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall.}}$

6. Predict the test results using both models

7. Evaluate both models using
   - Accuracy
   - Confusion Matrix.
   - Classification Report.

8. Compare the results.

Observation:

Metric	SVM	Logistic Regression.
Accuracy	97.78 %.	97.78
Precision	0.98	0.93.
Recall	0.98	0.98.
F1-Score.	0.98	0.98.

Both Models performed equally well on the IRIS dataset.

Why logistic Regression?

→ Simple & fast.

→ training is fast on small datasets.

→ works well on linearly seperable dataset.

Result:

Both SVM & logistic Regression classifiers achieved high performance on IRIS dataset with accuracy 97.78 %.

Thus the experiment was successfully implemented.

main DEEP-LEARNING-TECHNIQUES-LAB-EXP / DLTLAB3.ipynb

Go to file

Hanishkaa01 Add files via upload

53eebc0 · 4 minutes ago

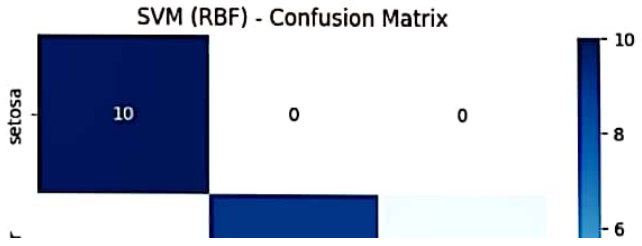
233 lines (233 loc) · 119 KB

Preview Code Blame Raw

In [2]:

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix,
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Standardize features
sc = StandardScaler()
X = sc.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Define classifiers
classifiers = {
    "SVM (RBF)": SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42),
    "Logistic Regression": LogisticRegression(max_iter=200, random_state=42),
    "KNN (k=5)": KNeighborsClassifier(n_neighbors=5),
    "Decision Tree": DecisionTreeClassifier(random_state=42)
}

# Compare classifiers
for name, model in classifiers.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, average='macro')
    rec = recall_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')

    print(f"\n{name} Statistics:")
    print(f"Accuracy : {acc*100:.2f}%")
    print(f"Precision: {prec*100:.2f}%")
    print(f"Recall   : {rec*100:.2f}%")
    print(f"F1-Score : {f1*100:.2f}%")

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, cmap="Blues", fmt="d",
                xticklabels=iris.target_names,
                yticklabels=iris.target_names)
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(f"{name} - Confusion Matrix")
    plt.show()

    # Detailed Classification Report
    print("Classification Report:\n", classification_report(y_test, y_pred, target_names=iris.target_n
```

```
SVM (RBF) Statistics:
Accuracy : 96.67%
Precision: 96.97%
Recall   : 96.67%
F1-Score : 96.66%
```



Ex:4 Build a Simple feed forward neural network to recognise handwritten characters.

Aim:

To design, implement and evaluate a Simple feedforward neural network for recognising handwritten characters.

Objectives:

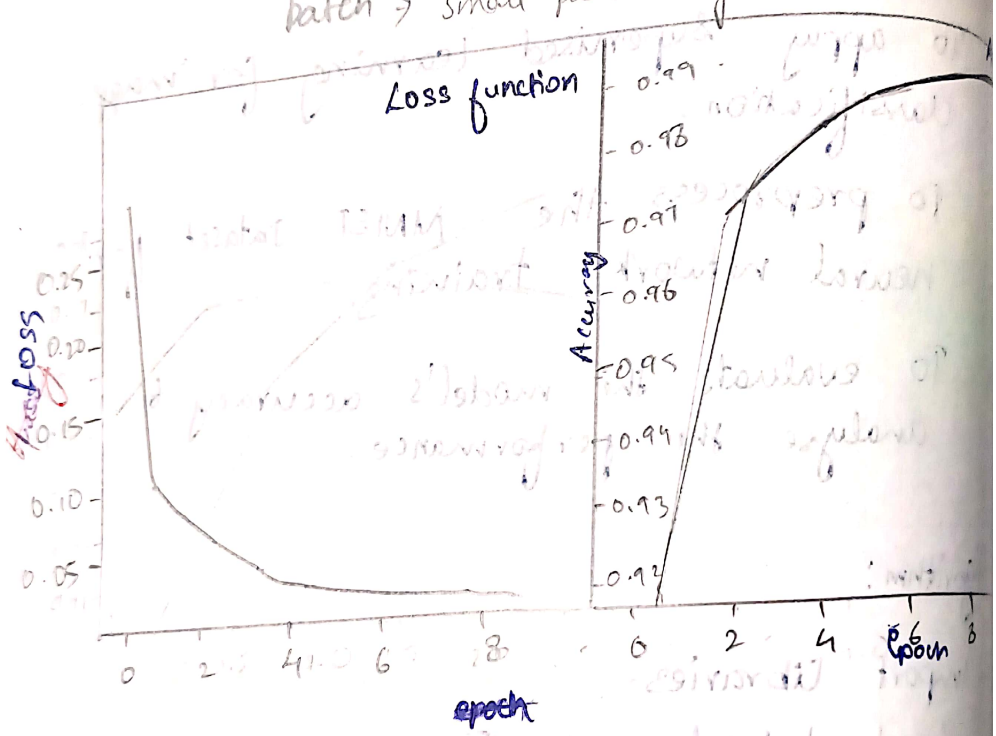
1. Understand the architecture of a feed forward neural network.
2. To apply supervised learning for image classification.
3. To preprocess the MNIST Dataset for neural network training.
4. To evaluate the model's accuracy & analyse the performance.

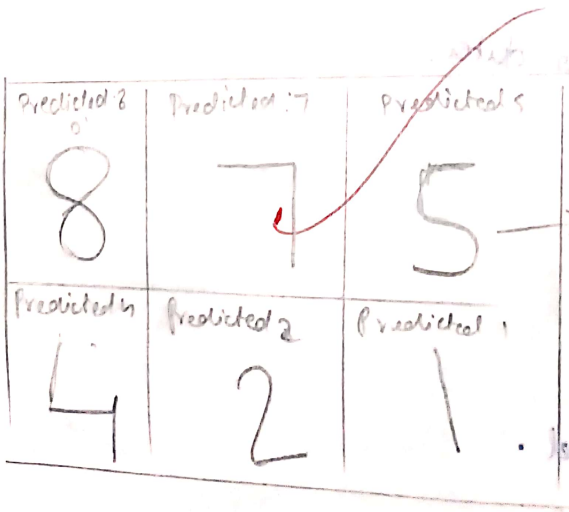
Algorithm:

1. Import libraries - pytorch, Keras, numpy.
2. Load dataset MNIST
3. Preprocess the data. normalise (0-1), flatten 28x28, convert labels
4. Define model. Input layer: 784 neurons
   Hidden layer: 128 neurons
   O/P layer: 10 neurons
5. Compile model.
6. Train model.
7. Evaluate model.
8. Predict & visualise.

Model: Simple feedforward NN.
Dataset: MNIST
Training: epoch 10
Batch size 64.
optimizer Adam
loss function Cross entropy loss.

Keywords:
⇒ feedforward → i/p → hidden → o/p no coming back (from
⇒ with cycles - RNN CNN
epochs → one complete cycle of all batch
batch → small packets of data.





Pseudocode:

START
Import requirell dibraries.
Load MNIST dataset
Normalise pixel values between 0 & 1
Flatten 28×28 images into vectors.
One-hot encode the labels.
Define a sequential Neural network.
    Input layer : size 784.
    Hidden layer : 128 neurons,
Compile mode with Adam optimiser softmax activation
Categorical cross entropy loss, accuracy metric.
Train model using training dataset for define epochs.
Evaluate model on test dataset.
Display accuracy & sample predictions
END.

Observations:

- achieved 97.96% accuracy on MNIST dataset with 5 epochs of training.
- Increasing hidden units can improve performance
- Model might Struggle with similar characters
- Error with centered, grayscale, similar Stroke thickness.
  invert colours.

Result:

The mode was successfully executed.

Hanishkaa01 Add files via upload 5407206 · 40 minutes ago

414 lines (414 loc) · 52.9 KB

Preview Code Blame Raw

In [1]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Load dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Define Feedforward Neural Network
class FeedForwardNN(nn.Module):
    def __init__(self):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = FeedForwardNN()

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training
epochs = 5
for epoch in range(epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {running_loss/len(trainloader):.4f}")

# Testing
correct, total = 0, 0
with torch.no_grad():
    for images, labels in testloader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy on test set: {100 * correct / total:.2f}%")
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 128MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 41.4MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 124MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 10.6MB/s]
Epoch 1, Loss: 0.4191
Epoch 2, Loss: 0.1967
Epoch 3, Loss: 0.1437
Epoch 4, Loss: 0.1151
Epoch 5, Loss: 0.1007
Accuracy on test set: 96.16%
```

In [5]:

```
from PIL import Image
import torchvision.transforms as transforms
import torch
from google.colab import files
import matplotlib.pyplot as plt

# Load trained model
model.eval()

# Preprocessing transform (same as training)
transform = transforms.Compose([
    transforms.Resize((28,28)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
```

1) Pumping Lemma

If a language L is Regular, pumping length P,
Any string $w \in L$ of length $|w| \geq P$
Consider $w = xyz$

where i) $|xy| \leq P$
ii) $|y| \geq 1$
iii) For All $i \geq 0$, $xy^iz \in L$

i) $L_1 = \{ w \in \{a,b\}^* \mid$ no. of a's in w = no of b's $\}$

Assume L_1 is regular

$L_1 = \{ ab, aabb, aaabbb, \cdots \}$

let $w = aabb$ and $P = 4$

$|w| = 4$, $\therefore |w| \geq P$

Consider $x = a$, $y = ab$, $z = \epsilon$

$|xy| \leq P$

$2 \leq 4$

$|y| \geq 1$

$2 \geq 1$

$xy^iz \in L_1$, for all $i \geq 0$

$i = 0 \rightarrow xz \rightarrow ab$

$i = 1 \rightarrow xyz \rightarrow aabb$

$i = 2 \rightarrow xy^2z \rightarrow aababb \in L_1$

Contradiction

L_1 is not Regular.

ii) $L_2 = \{w \text{ in } \{a,b\} \mid w \text{ is a Palindrome}\}$

Assume L_2 is Regular

$L_2 = \{aba, aabaa, aaabaaa, \ldots\}$

Let $w = aabaa$ & $p = 5$: $|w| \geq p$

Consider $x = aa$, $y = b$, $z = aa$

$$|xy| \leq p$$
$$4 \leq 5$$
$$1 \geq 1$$
$$|y| \geq 1$$
$$1 \geq 1$$

$\forall i \geq 0$, $xy^i z \in L_2$

$i = 0$, $xz = aaaa \notin L_2$

$i = 1$, $xyz = aabaa$

L_2 is not Regular

iii) $L_3 = \{a^k \mid k \text{ is Prime Number}\}$

Assume L_3 is Regular

$L_3 = \{aa, aaa, aaaaa, \ldots\}$

Let $w = aaa$ & $p = 3$

$|w| = 3 \quad \therefore |w| \geq p$

consider $x = a$, $y = a$, $z = a$

$$|xy| \leq p$$
$$2 \leq 3$$
$$|y| \geq 1$$
$$1 \geq 1$$

$\forall i \geq 0$, $xy^i z \in L_3$

$i = 0$, $yz = aa$

$i = 1$, $yyz = aaa$

$i = 2$, $yy^2 z = aaaa \notin L_3$

$\Rightarrow L_3$ is not Regular

iv) $L_4 = \{a^k \mid k$ is a perfect Square $\}$

Assume L_4 is Regular

$L_4 = \{a, aaaa, \ldots\}$

Let $w = aaaa$ & $P = 4$

$|w| = 4 \therefore |w| \geq P$

Consider $x = a$, $y = aa$, $z = a^1$

$|xy| \leq P$

$3 \leq 4$

$|y| \geq 1$

$2 \geq 1$

$\forall i \geq 0$, $xy^i z \in L_4$

$i = 0$, $yz = aa \notin L_4$

L_4 is not Regular

v) $L_5 = \{a^k \mid k$ is a factorial number $\}$

Assume L_5 is Regular

$L_5 = \{a, aa, aaaaaa, a^2, \ldots\}$

Let $w = aaaaaa$ and $P = 6$

$|w| = 6 \therefore |w| \geq P$

Consider $x = aa$, $y = aa$, $z = aa$

$|xy| \leq P$

$4 \leq 6$

$|y| \geq 1$

$|z| \geq 1$

$\forall i \geq 0$, $xy^i z \in L_5$

$i = 0$, $xz = aaaa \notin L_5$, Contradiction

$i = 1$, $xyz = aaaaaa \in L_5$

L_5 is not Regular.

2) If L and R are two Regular Languages

i) L concatenated with [L (intersection) R]

L . (L ∩ R)

given L and R are Regular

By Closure Properties

Intersection of Regular Languages is Regular

L ∩ R is Regular

Concatenation of Regular Languages is Regular

L . (L ∩ R) is Regular

∴ L (L ∩ R) is Regular

ii) Complement of R union (Reversal of L)

L & R are Regular

By Closure Properties.

- Complement of Regular lang is Regular (R^c is Regular)
- Reversal of Regular Language is Regular (L^R is Regular)
- Union of Regular Language is Regular ($R \cup L$ is Regular)

∴ $(R^c) \cup (L^R)$ is Regular.

iii) (Kleene Star of L) set difference (Kleene star of R) $(L^*) \setminus (R^*)$

given L & R are Regular.

- By Closure Properties
  - Kleene star of Regular language is Regular (L^* is Regular)
  - Complement of Regular Language is Regular (R^c is Regular)
  - Intersection of Regular Lang is Regular ($L \cap R$ is Regular)

$(L^*) \setminus (R^*) = L^* \cap (R^*)^c$

∴ $(L^*) \setminus (R^*)$ is Regular.

Lab5: Study Of Activation function and Their Role.

Aim:

To study the different activation functions in dl, understand their role & analyse their effects on training & performance of NN

Objectives:

- Understand the purpose of activation func in nl.
- implement various activation functions such as Sigmoid, Tanh, ReLU & Reaky ReLU.
- Analyse and compare the performance of a simple neural network using different Activation funct.
- Observe the effects of activation functions on learning dynamics.
- draw conclusion on best activation function.

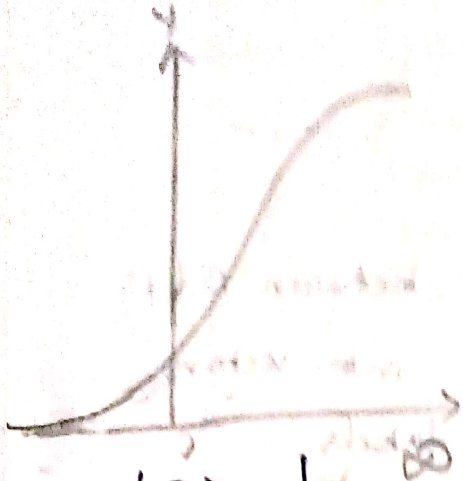
Common Activation functions:

- Sigmoid - Squashes input values to [0, 1]
- Tanh - Squashes input values [-1, 1]
- ReLU - Outputs 0 for negative inputs & linear to +ve inputs.
- Leaky ReLU - Allows a small, non zero gradients when i/p is -ve.

Algorithm:

1. Install required libraries (numpy, matplotlib)
2. Implement Python code to define and compute activation functions.
3. Generate input data using Numpy.
4. Apply activation function and Visualise their

Sigmoid

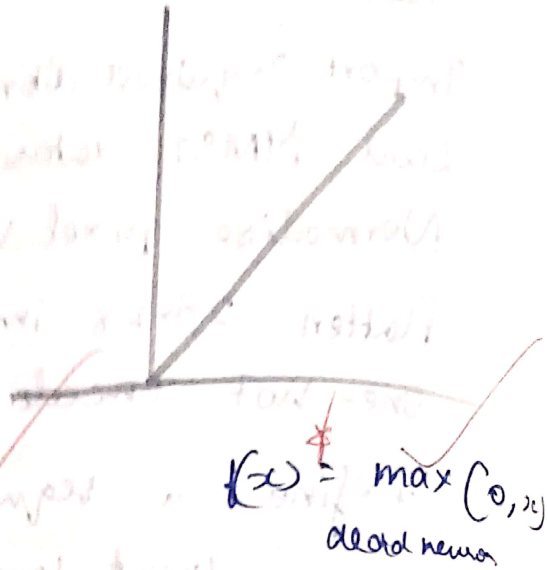


$$f(x) = \frac{1}{1+e^{-x}}$$

range ⇒ (0, 1)

V.G.

ReLU

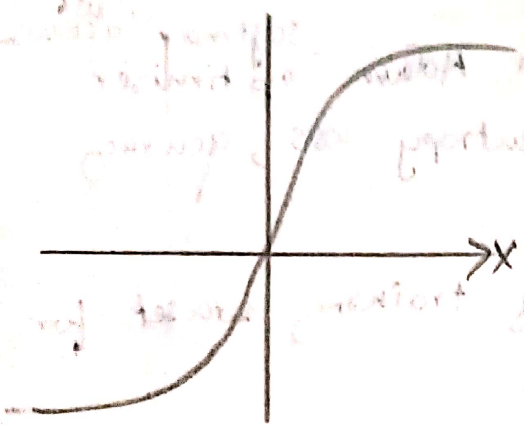


$f(x) = \max(0, x)$

dead neuron

$(0, \infty)$

tanh

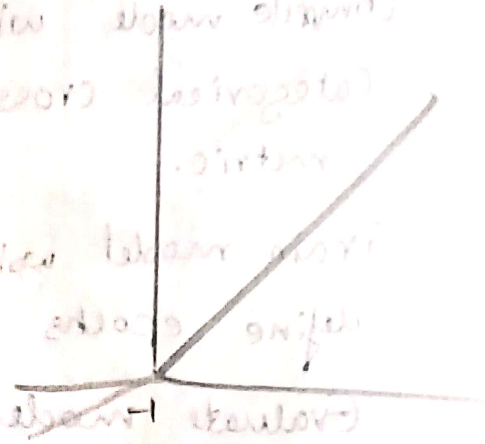


$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$(-1, 1)$

V.G. but slightly better...

Leaky ReLU



$f(x) = x, x > 0$

$0.1x$

$\max(0.1x, x)$

$(-\infty, \infty)$

outputs using Matplotlib.

5. Analyse the behaviour of each function.

Pseudo Code:

1. Initialize
   - Training dataset (X, Y)
   Initialize weights (W_1, W_2 ..) & biases (b_1, b_2 ...) randomly
   Choose activation function : ReLU / Sigmoid / Tanh
   Set Learning rate & number of epochs.

2. Training loop-
   for each epoch.
      for each training sample (x, y):
         $z1 = W1 * x + b1$
         $a1 = Activation(z1)$
         $z2 = W2 * a1 + b2$
         $a2 = Activation(z2)$
         $z3 = W3 * a2 + b3$
         y_pred = softmax (z3)
         loss = CrossEntropy (y, y_pred)
         Compute gradients of loss w.r.t W_1, W_2 W_3 & b_1, b_2, b_3.
         Update weights & biases using gradient Descent.
      END FOR
      Print current epoch & loss.

3. Testing
   input test data
   Perform forward pass to get predictions
   Compare predicted class vs actual label
   Compute Accuracy

END

Observations:

1. Sigmoid: Outputs are compressed b/w 0 & 1, suitable for probability based o/p but prone to vanishing gradients.
2. ReLU: o/p 0 for -ve i/p, promoting sparsity but may cause "dying ReLU", where neurons o/p 0 permanently.
3. Tanh: O/p range from -1 to 1, useful for centered data, but also suffers from vanishing gradient for large inputs.
4. Leaky ReLU: Allows small -ve o/p, mitigating the "dying ReLU" problem.

1.9.25

Result:

The code was successfully executed and verified.

In [2]:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt


transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, tra
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, tra

trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=1000, shuffle=False)


class SimpleNN(nn.Module):
    def __init__(self, activation_fn):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(28*28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.activation = activation_fn

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        return self.fc3(x)


def train_and_eval(activation_fn, epochs=5):
    model = SimpleNN(activation_fn)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_losses = []
    test_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for images, labels in trainloader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        avg_loss = running_loss / len(trainloader)
        train_losses.append(avg_loss)

        # Evaluate on test set
        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for images, labels in testloader:
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        acc = correct / total
        test_accuracies.append(acc)

        print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}, Test Acc: {acc*10

    return train_losses, test_accuracies


activations = {
    "Sigmoid": nn.Sigmoid(),
    "Tanh": nn.Tanh(),
    "ReLU": nn.ReLU(),
    "LeakyReLU": nn.LeakyReLU(0.1)
}

results = {}
for name, act in activations.items():
    print(f"\nTraining with {name} activation...")
    train_losses, test_accuracies = train_and_eval(act, epochs=5)
    results[name] = (train_losses, test_accuracies)


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


plt.subplot(1,2,1)
for name, (losses, accs) in results.items():
    plt.plot(losses, label=name)
plt.title("Training Loss vs Epochs")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()


plt.subplot(1,2,2)
for name, (losses, accs) in results.items():
```

Lab 6: Implement Gradient Descent And Backpropogation in Deep Neural Network.

Aim:

To implement and understand the working of gradient descent and backprogation in training a deep neural network for Supervised learning tasks.

Objectives:

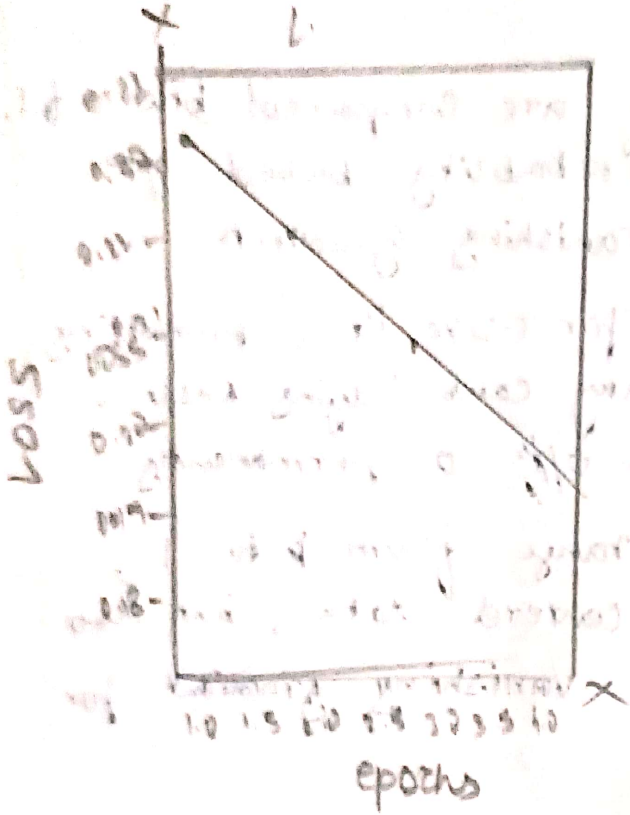
→ how weights & biases are updated using G.D.
→ forward pass & backward pass in DNN.
→ minimize loss function using optimisation.
→ learning rate & Iterations affect convergence.
→ trained network's predictions Compared to expected o/p.

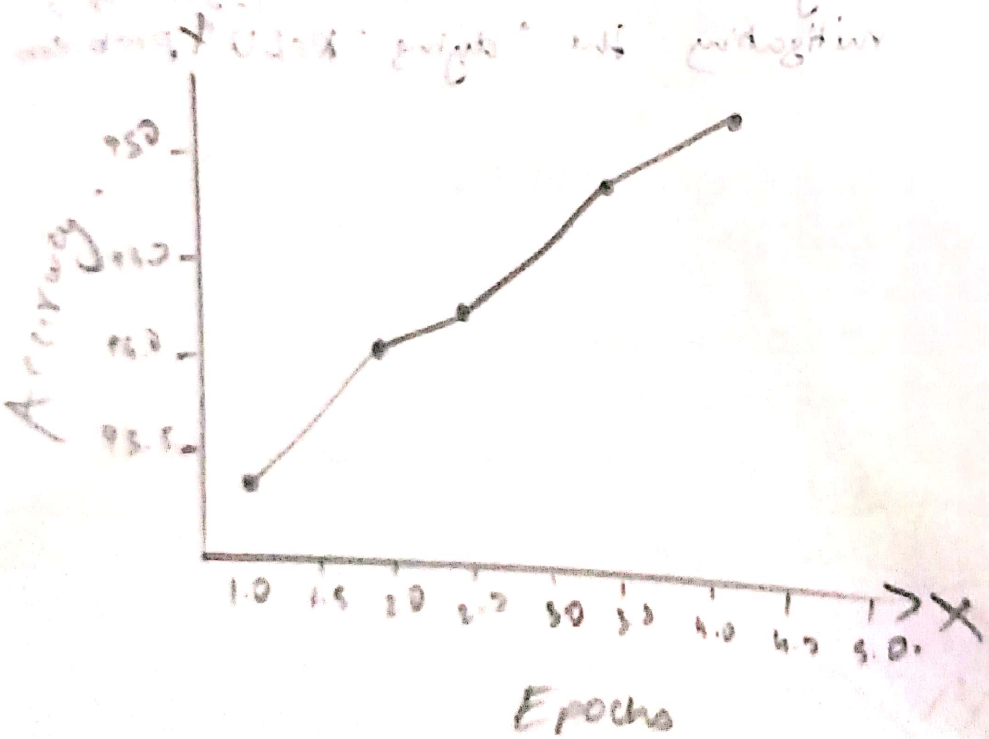
Algorithm:

1) Initilize network architecture (no. of i/p, hidden, o/p nodes)
2) Randomly initilize weights & biases.
3) Forward propagation,
   - input → hidden using weighted sum.
   - Apply activation func.
   - Computed o/p layer values.

Loss Calculation.
- Compute error using a loss function (MSE/CE)

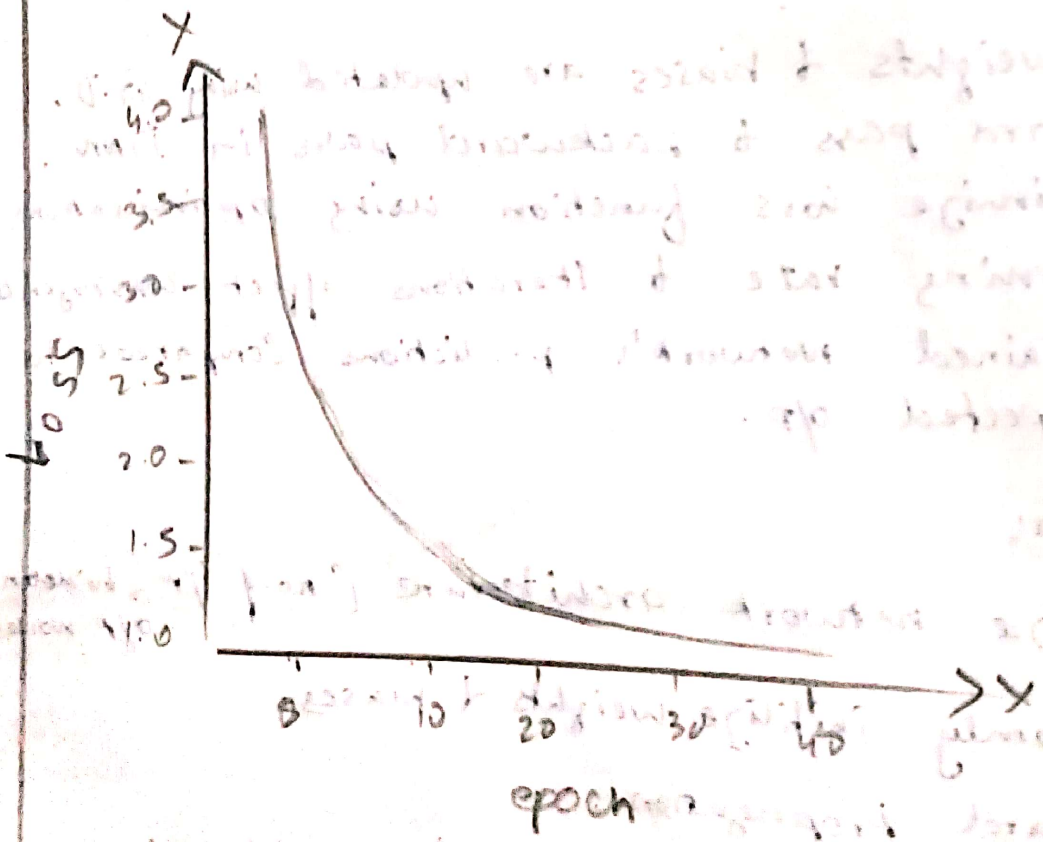
Backward Propagation,
- Compute gradient of loss with.r.to weights & biases.
- Update weights & biases using G.D rule.


Loss
epochs


Accuracy
Epochs

Epoch 1 :	Loss : 0.2449	Acc : 93.14 %
Epoch 2 :	Loss : 0.2246	Acc : 94.00 %
Epoch 3 :	Loss : 0.2059	Acc. 94.25 %
Epoch 4 :	Loss : 0.1893	Acc : 94.83 %
Epoch 5 :	Loss : 0.1750	Acc : 95.23 %

loss curve



$$W = W - \eta \frac{\partial L}{\partial W}$$

η → learning rate

$$b = b - \eta \frac{\partial L}{\partial b}$$

4) Repeat until convergence

5) O/P trained weights, loss trend

PseudoCode:

```
Initialize weights W, biases b, learning_rate, epochs
for each epoch:
    output = forward_pass(X, W, b)
    loss = compute_loss(y_true, output)
    gradients = backward_pass(output, y_true, W, b)
    W, b = update_parameters(W, b, gradients, learning_rate)
```

Observation:

- loss decreases as epochs increases
- Small L.R → slow convergence but stable.
- Large L.R → faster convergence but risk of divergence.
- Network accuracy improves with proper tuning of L.R & hidden layers.

Result:

The code was successfully executed & verified.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tqdm import trange


def sigmoid(x): return 1.0 / (1.0 + np.exp(-x))
def dsigmoid(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x): return np.tanh(x)
def dtanh(x):
    return 1.0 - np.tanh(x)**2

def relu(x): return np.maximum(0, x)
def drelu(x): return (x > 0).astype(float)

def leaky_relu(x, a=0.01):
    return np.where(x > 0, x, a * x)
def dleaky_relu(x, a=0.01):
    dx = np.ones_like(x)
    dx[x <= 0] = a
    return dx


def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True)
    expz = np.exp(z)
    return expz / np.sum(expz, axis=1, keepdims=True)

def cross_entropy_loss(probs, y_true_onehot):

    m = y_true_onehot.shape[0]
    p = np.clip(probs, 1e-12, 1.0)
    return -np.sum(y_true_onehot * np.log(p)) / m

class MLP:
    def __init__(self, input_dim, hidden_dims, output_dim, activation='relu', lr=0.1, weight_scale=0.0
        np.random.seed(seed)
        self.layers = [input_dim] + hidden_dims + [output_dim]
        self.L = len(self.layers) - 1
        self.lr = lr

        self.W = [None] * self.L
        self.b = [None] * self.L
        for i in range(self.L):
            self.W[i] = np.random.randn(self.layers[i], self.layers[i+1]) * weight_scale
            self.b[i] = np.zeros((1, self.layers[i+1]))

        if activation == 'sigmoid':
            self.act = sigmoid; self.dact = dsigmoid
        elif activation == 'tanh':
            self.act = tanh; self.dact = dtanh
        elif activation == 'relu':
            self.act = relu; self.dact = drelu
        elif activation == 'leaky':
            self.act = lambda x: leaky_relu(x, a=0.01)
            self.dact = lambda x: dleaky_relu(x, a=0.01)
        else:
            raise ValueError("activation must be: sigmoid, tanh, relu, leaky")

    def forward(self, X):
        """Forward pass. Stores pre-activations (Z) and activations (A)."""
        A = X
        self.As = [A]
        self.Zs = []
        for i in range(self.L - 1):
            Z = A.dot(self.W[i]) + self.b[i]
            A = self.act(Z)
            self.Zs.append(Z)
            self.As.append(A)

        Z = A.dot(self.W[-1]) + self.b[-1]
        self.Zs.append(Z)
        self.As.append(Z)
        probs = softmax(Z)
        return probs

    def backward(self, probs, y_onehot):
        """Backpropagation. Compute gradients for all W and b."""
        m = y_onehot.shape[0]
        grads_W = [None] * self.L
        grads_b = [None] * self.L


        dZ = (probs - y_onehot) / m
        grads_W[-1] = self.As[-2].T.dot(dZ)
        grads_b[-1] = np.sum(dZ, axis=0, keepdims=True)


        dA_prev = dZ.dot(self.W[-1].T)
        for l in range(self.L - 2, -1, -1):
            Z = self.Zs[l]
            dZ = dA_prev * self.dact(Z)
            grads_W[l] = self.As[l].T.dot(dZ)
            grads_b[l] = np.sum(dZ, axis=0, keepdims=True)
            if l > 0:
                dA_prev = dZ.dot(self.W[l].T)

        return grads_W, grads_b

    def step(self, grads_W, grads_b):
        """SGD parameter update (vanilla)."""
        for i in range(self.L):
            self.W[i] -= self.lr * grads_W[i]
            self.b[i] -= self.lr * grads_b[i]

    def predict(self, X):
        probs = self.forward(X)
```

```
    def predict(self, X):
        probs = self.forward(X)
        return np.argmax(probs, axis=1)


(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype(np.float32) / 255.0
x_test  = x_test.astype(np.float32) / 255.0
x_train = x_train.reshape(-1, 28*28)
x_test  = x_test.reshape(-1, 28*28)



y_train_oh = to_categorical(y_train, 10)
y_test_oh  = to_categorical(y_test, 10)


def train_experiment(activation='relu',
                     hidden_dims=[128, 64],
                     lr=0.1,
                     epochs=10,
                     batch_size=128,
                     weight_scale=0.05,
                     seed=123):
    model = MLP(input_dim=28*28, hidden_dims=hidden_dims, output_dim=10,
                activation=activation, lr=lr, weight_scale=weight_scale, seed=seed)
    history = {'train_loss': [], 'train_acc': [], 'test_loss': [], 'test_acc': []}
    n = x_train.shape[0]
    steps_per_epoch = n // batch_size

    for epoch in range(epochs):

        perm = np.random.permutation(n)
        x_shuf = x_train[perm]
        y_shuf = y_train_oh[perm]
        epoch_loss = 0.0

        for i in range(steps_per_epoch):
            start = i * batch_size
            end = start + batch_size
            Xb = x_shuf[start:end]
            yb = y_shuf[start:end]


            probs = model.forward(Xb)
            loss = cross_entropy_loss(probs, yb)
            epoch_loss += loss

            grads_W, grads_b = model.backward(probs, yb)


            model.step(grads_W, grads_b)


        avg_loss = epoch_loss / steps_per_epoch
        y_pred_train = model.predict(x_train[:2000])
        train_acc = (y_pred_train == y_train[:2000]).mean()

        probs_test = model.forward(x_test)
        test_loss = cross_entropy_loss(probs_test, y_test_oh)
        test_pred = np.argmax(probs_test, axis=1)
        test_acc = (test_pred == y_test).mean()

        history['train_loss'].append(avg_loss)
        history['train_acc'].append(train_acc)
        history['test_loss'].append(test_loss)
        history['test_acc'].append(test_acc)

        print(f"Epoch {epoch+1}/{epochs} | train_loss={avg_loss:.4f} train_acc={train_acc*100:.2f}% |

    return model, history


activations = ['sigmoid', 'tanh', 'relu', 'leaky']
results = {}


for act in activations:
    print("\n" + "="*50)
    print(f"Training with activation: {act}")
    model, hist = train_experiment(activation=act, hidden_dims=[256, 128], lr=0.1, epochs=8, batch_siz
    results[act] = hist


for act, hist in results.items():
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.plot(hist['train_loss'], marker='o')
    plt.title(f"{act} - Train Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")

    plt.subplot(1,2,2)
    plt.plot(hist['test_acc'], marker='o')
    plt.title(f"{act} - Test Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")

    plt.suptitle(f"Activation: {act}")
    plt.show()
```

```
Downloading data from https://storage.googl
eapis.com/tensorflow/tf-keras-datasets/mnis
t.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━
0s 0us/step
```

16/9/25

Lab 7: BUILD A CNN MODEL TO CLASSIFY CATS AND DOGS DATASET.

Aim:

To build and train a CNN model using transfer learning to classify images of cats & dogs with high accuracy.

Objective

- Understand and apply CNN for image classification
- Utilize transfer learning with pretrained model
- Implement data preprocessing and augementation
- Evaluate mode performance on unseen test data
- Test the trained model on new images.

Algorithm

1. Download & extract the cats & dogs dataset
2. Prepare data generators
   - Apply data Augementation & preprocessing
   - Split training data into training & Validation sets.
3. Load pretrained MobileNetV2 model without top layers.
4. Add custom classification head:
   - Global Avg Pool.
   - Dropout layer for regularisation.
   - Dense o/p layer with Sigmoid Activation for binary classification.

Layer (type)	Output shape	Param
conv2d-8 (Conv2D)	(None, 146, 148, 32)	896
Max pooling2d-8 (MaxPooling2D)	(None, 74, 74, 32)	0
Conv2d-9 (Conv2D)	(None, 72, 72, 64)	18,496
Max-pooling2d-9 (MaxPooling2D)	(None, 36, 36, 64)	0
Conv2d-10 (Conv2D)	(None, 34, 34, 128)	73,856
Max-Pooling2d-10 (MaxPooling2D)	(None, 17, 17, 128)	0

Epoch 1: acc 0.5092 loss: 0.7737 val.acc 0.49 Val loss: 0.6

Epoch 2: acc: 0.7188 loss: 0.6858 val_acc 0.6858 Vall 0.

Epoch 3: acc: 0.4794 loss: 0.6928 val_acc: 0.4688 Val_los: 0.6

Epoch 4: acc: 0.4062 loss: 0.6933 val_acc: 0.500 Val_loss: 0.690

Epoch 5: acc: 0.5055 loss: 0.6912 Val acc: 0.6250 Val_loss: 0.678

Epoch 6: acc: 0.6250 loss: 0.6724 Val_acc: 0.6042 Val_loss: 0.686

Epoch 7: acc: 0.5952 loss: 0.6875 Vall_acc: 0.6042 val loss: 0.66

Epoch 8: acc: 0.6250 loss: 0.6633 Val_acc: 0.6146 Val loss: 0.62

Epoch 9: acc: 0.6452 loss: 0.6038 val acc: 0.6346 val loss: 0.67

Epoch 10: acc: 0.7500 loss: 0.5812 val_acc: 0.6562 Val loss 0.62

Training loss
Validation loss
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6

Feature extraction
Classifier
Input
Convolution
ReLU
Pooling
Fully connected
output
Conv layer1
Pooling layer1
Conv layer2
Pooling layer2
Conv layer3
Pooling layer3
Fully connected layer

5. Compile a model with adam optimiser & binary Cross entropy loss.
6. Train the model on trainining data with Validation.
7. Unfreeze last few layers of base model for fine tuning.
8. Evaluate the model on test data
9. Save the trained Model.
10. Load saved model & preprocess new i/p images.

Observation :

- Pretrained model helped increase accuracy.
- Data Augmentation helped reduce overfitting.
- Final test acuracy reached upto 90-95%.
- Model could reliably predict on new unseen images with high confidence.

Result..

the Code was Verified successfully and executed

8. Build a LSTM

Aim:

To build a Long Term Memory model that Predicts future stock Prices based on historical data.

Objective.

1. Develop a deep learning model using LSTM
2. Understand how LSTM networks can capture long term dependencies in time series data.
3. Compare the predicted stock prices with actual prices to evaluate the model's accuracy.
4. Demonstrate that LSTM perform better than simple RNN in regression based sequence prediction Tasks.

Pseudocode:

1. Start
2. Import necessary libraries
   numpy, pandas, matplotlib, tensorflow
   sklearn, yfinance.
3. Load dataset.
   - download Google stock data using yfinance
   select the close price column.
4. Preprocess the data
   - Normalise data between 0 & 1 using MinMax Scaler
   - Split dataset into training(80) & testing 20%

~~Layer~~: Sequential Model

Layer (type)	Output Shape	Param #
LSTM	(None, 60, 50)	10,400
dropout (Dropout)	(None, 60, 50)	0
lstm_1 (LSTM)	(None, 50)	20,200
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 25)	1,275
dense_1 (Dense)	(None, 1)	26

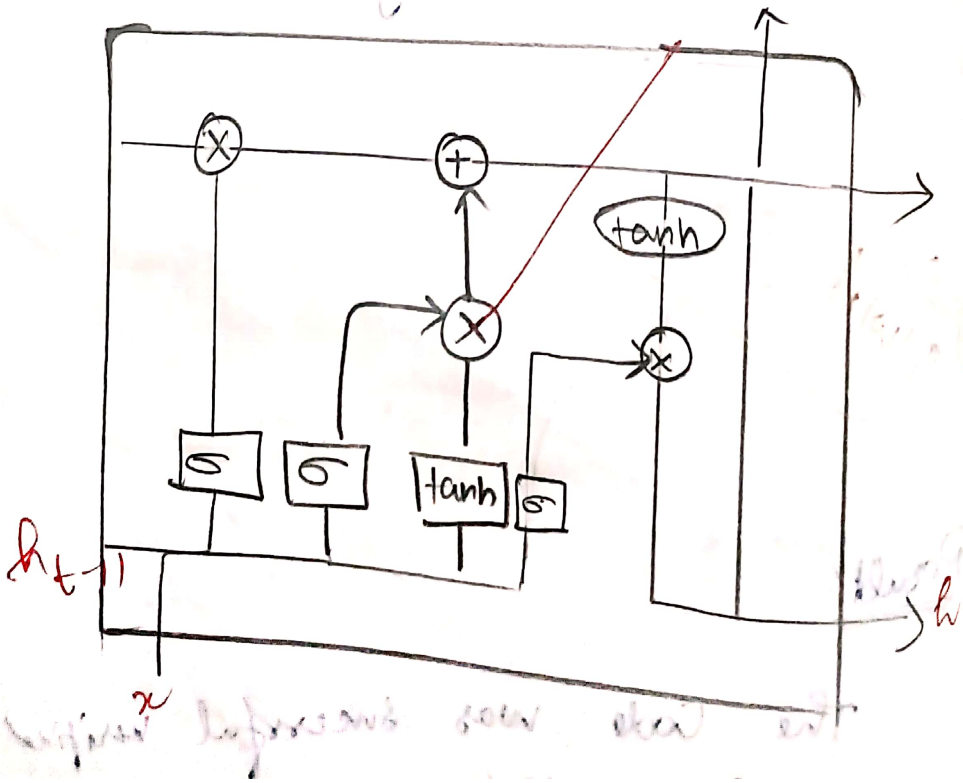
Total params: 31,901

Trainable params: 31,901

Non-Trainable params: 0.

Training data shape: (946, 60, 1)

LSTM Architecture



5. Create Time series sequences.
   for each epoch point in dataset
      Input (x) : Prev 60 closing prices.
      Output (y) : next closing price

6. Reshape Input data.

7. Buid LSTM model.

8. Train the model.

9. Evaluate the model.

10. Visualise the results.

11. Analyse performance

12. STOP.

Observation :

1. Google's stock prices for past 5 years was successfully loaded & visualised

2. Preprocessing such as normalisation (minmaxscaler)

3. During training both RNN & LSTM demonstrated decreasing training loss

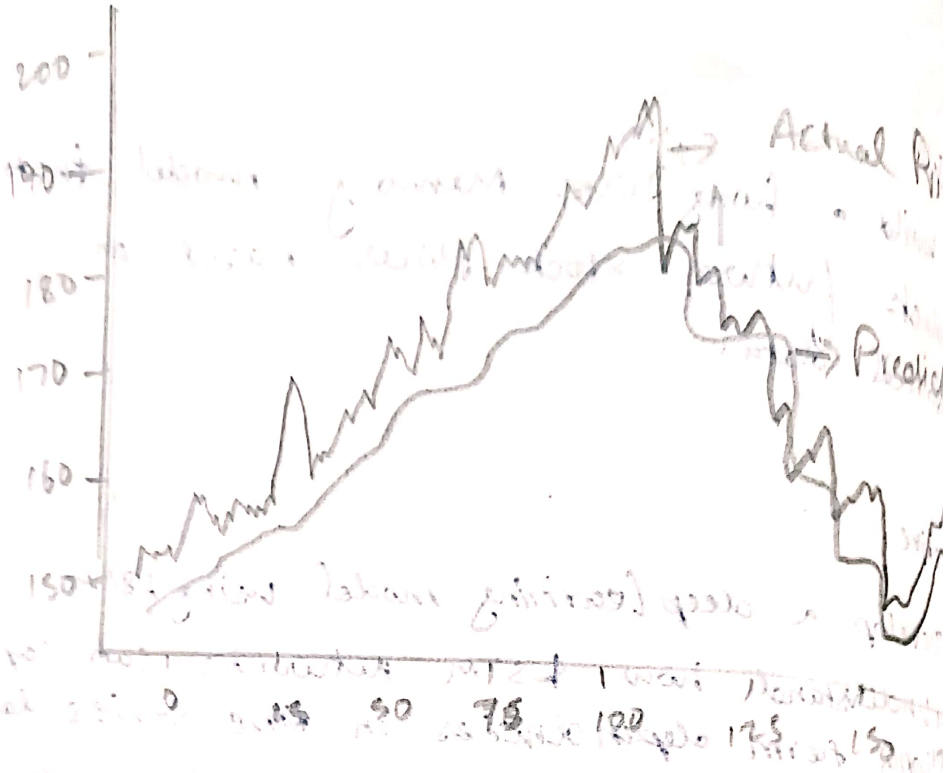
4. LSTM model showed smoother & more stable convergence.

5. MSE was lower than RNN.

9/10/25

Result :

The code was successfully executed & verified



epoch 1 : loss : 0.0489
epoch 2 : loss : 0.0041
epoch 3 : loss : 0.0025
epoch 4 : loss : 0.0025
epoch 5 : loss : 0.0026
epoch 6 : loss : 0.0027
epoch 7 : loss : 0.0019
epoch 8 : loss : 0.0018
epoch 9 : loss : 0.0017
epoch 10 : loss : 0.0015
epoch 11 : loss : 0.0016
epoch 12 : loss : 0.0016
epoch 13 : loss : 0.0013
epoch 14 : loss : 0.0013
epoch 15 : loss : 0.0014
epoch 16 : loss : 0.0013
epoch 17 : loss : 0.0012
epoch 18 : loss : 0.0014

9. Implementation of Recurrent Neural Network

Aim

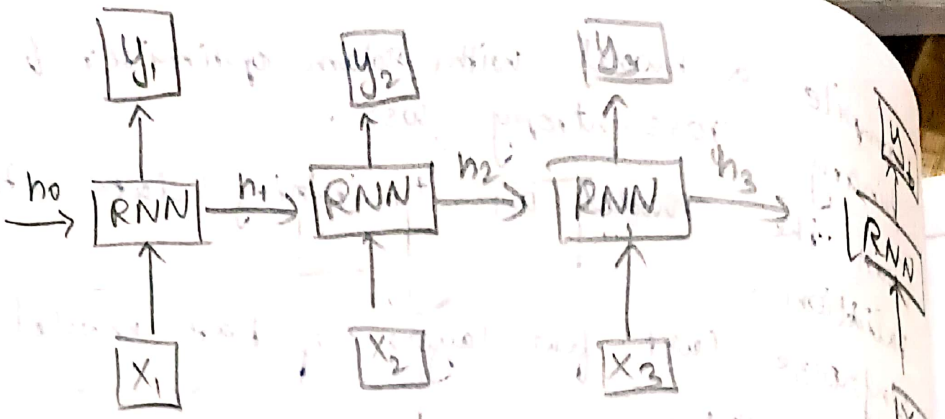
To design and implement a Recurrent Neural Network model for ~~text classification~~ Regression and evaluate its performance using accuracy & confusion matrix.

Objectives:

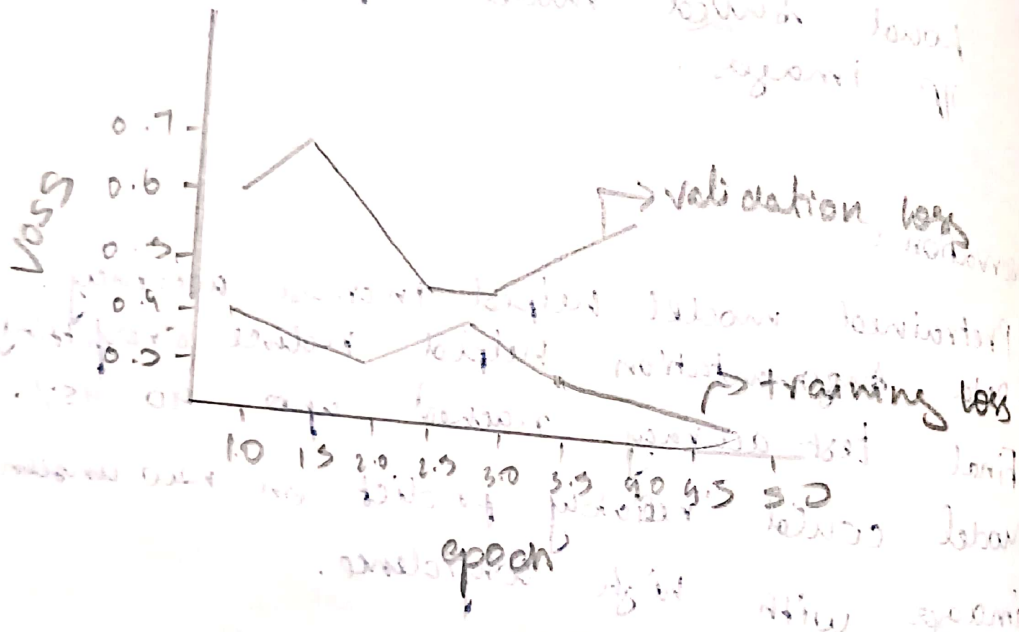
1. To process ~~textual~~ stock data into numerical sequence
2. To build RNN model for analysing sequencial data.
3. To train & validate the model for sentiment prediction.
4. To evaluate the model using accuracy, precision, recall, F1-score & confusion matrix.

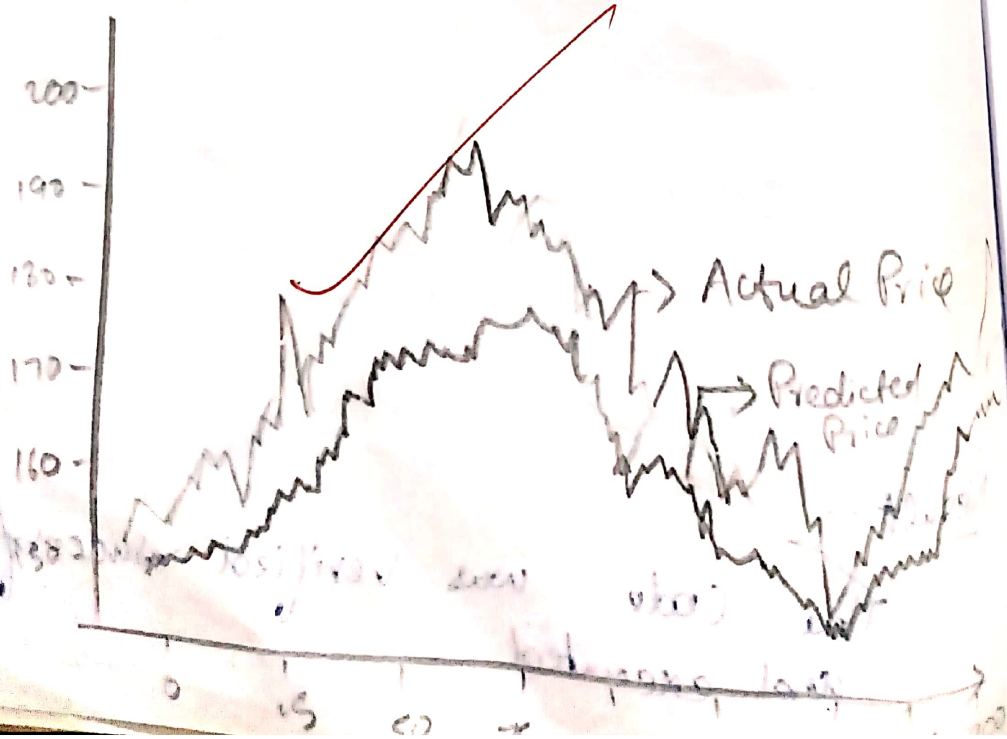
Algorithm:

1. Import Necessary libraries Tensorflow, Keras, Numpy etc..
2. Load & Preprocess the dataset. tokenisation, padding, train + test split
3. Build the RNN model using Keras Sequential API
4. Compile the model with an optimiser and loss function
5. Train the model on the training dataset
6. Evaluate the model using test data.



$$h^{(t)} = \tanh(h^{(t-1)} + W \cdot e^{(t)} + b$$





9. Implementati
Net

Aim

To design and
Neural Network m
and evaluate its pe
& confusion matrix.

Objectives:

1- To process
2. To build RNN
data.
3. To train
sentiment predi
4. To evaluate
precision, rec
matrix.

Algorithm:

1. Import Nece
2. Load &
tokenisation
3. Build the
sequential
4. Compile th
and loss
5. Train the
6. Evaluate

Epoch 1

loss : 0.5258 accuracy : 0.7373

val_loss 0.4582 - val_accuracy : 0.80

Epoch 2

loss : 0.3904 acc : 0.8317 val_loss : 0.4121 val

Epoch 3

loss 0.3168 acc 0.8692 val_loss : 0.3896 val

Epoch 4

loss 0.2657 acc 0.8923 val_loss 0.4064 val

Epoch 5

loss : 0.2263 acc : 0.9102 val_loss : 0.4458 val

Test loss : 0.4127

Test Acc : 0.2174

	precision	recall	f1-score	Suppo
	0.70	0.67	0.68	230
	0.73	0.75	0.74	270
macro avg	0.717	0.717	0.717	500
weighted avg	0.717	0.717	0.717	500

Mean squared Error : 59.76920735

Mean Absolute error : 6.567392102

7. Generate the Confusion matrix and Classification report.

8. Display accuracy & loss graphs.

Observation.

1. RNN has short term memory so it forget informations to retain it memories.
Sometimes in sequential data the continuous values like loss in stock prices is crucial to predict future prices
This is vanishing gradient Problem.
This is resolved by LSTM.

Result:

The Code was sucessful verified & executed.

Perform Compression on MNIST DATASET Using AUTO Encoder.

Aim:

To implement an autoencoder using pytorch for compressing and reconstructing images from the MNIST dataset, demonstrating unsupervised feature learning and dimensionality reduction.

Objectives:

1. To understand the working principle of an autoencoder and its encoder - decoder structure.
2. To build and train a fully connected autoconnected using pytorch on the mnist dataset.
3. To evaluate the models ability to compress and reconstruct images.
4. To visualise the original and reconstructed image to analyse reconstruction quality

Pseudocode

BEGIN

Import torch, torch.nn, torch.optim

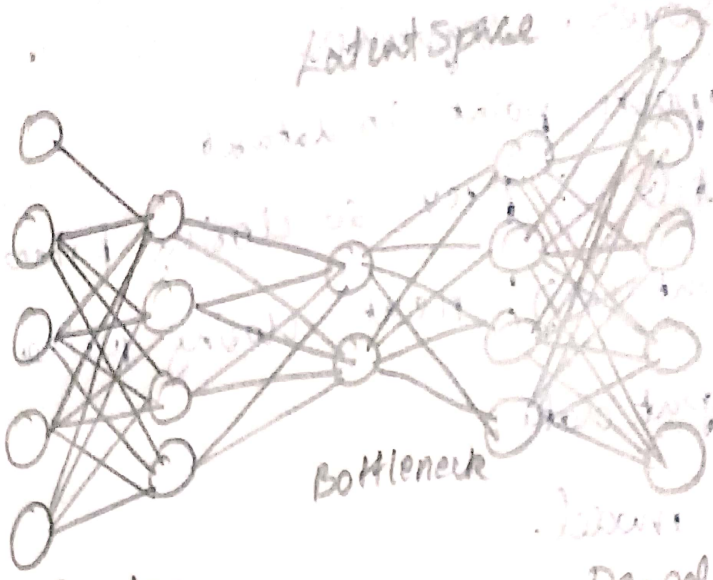
load MNIST dataset.

Normalise images to [0,1]

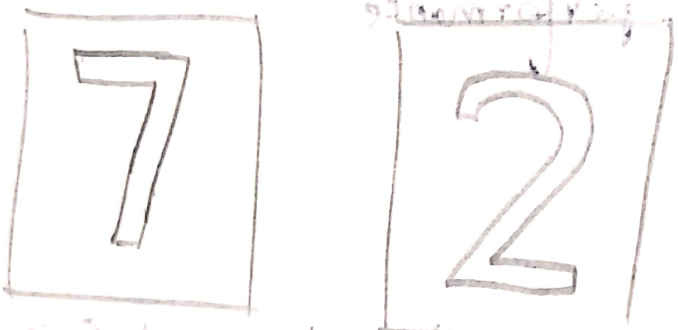
Define encoder:

Input → Dense (128, relu) → Dense (64, relu) →

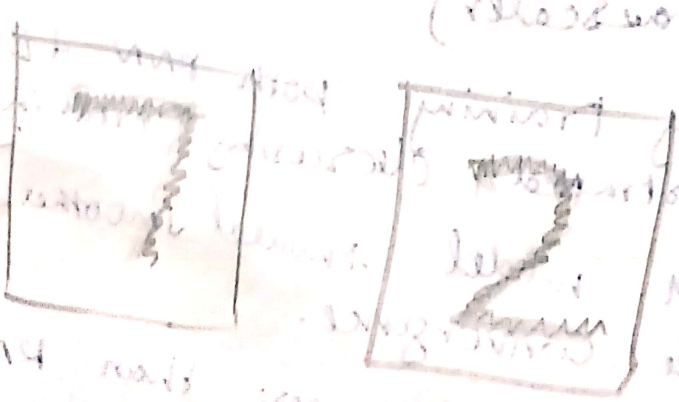
Dense (32, relu)



Original Image:



Reconstructed Image.



Define decoder

Dense (64, relu) → Dense(128, relu) → Dense(784, sigmoid)

Combine encoder and decoder → Autoencoder

Compile (autoencoder, optimizer = 'adam', loss='mse')

Train on MNIST images (input = output)

Visualise original vs reconstructed images

Observation

As epochs increases, reconstruction loss decreases, and the model learns compressed latent feature (32-dim representation).

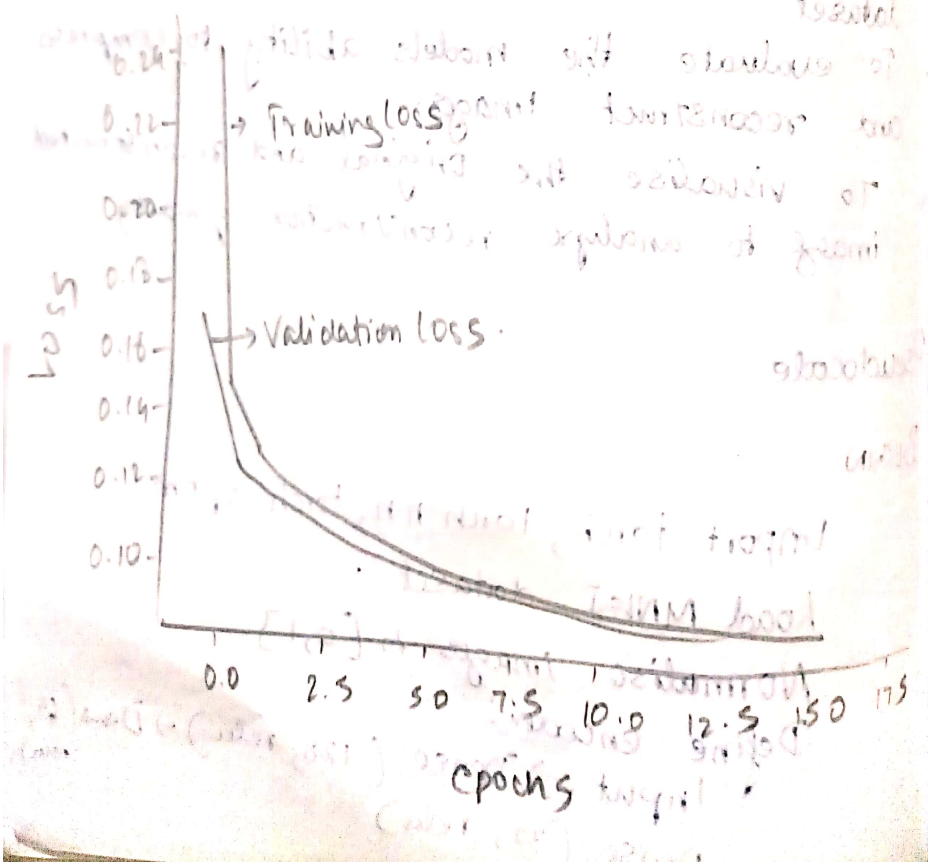
Result:

The Autoencoder successfully compressed and reconstructed MNIST images with minimal loss.

Training data shape (60000, 784)

Test data shape (10000, 784)

Epoch	loss	val_loss
1	0.3416	0.1647
2	0.1550	0.1229
3	0.1309	0.1153
4	0.1215	0.1103
5	0.1149	0.1073
6	0.1107	0.1043
7	0.1075	0.1025
8	0.1051	0.1004
9	0.1033	0.0988
10	0.1012	0.0975
11	0.0999	0.0967
12	0.0986	0.0957
13	0.0974	0.0948
14	0.0963	0.0941
15	0.0957	



Experiment Using Variational Autoencoder (VAE)

Aim:

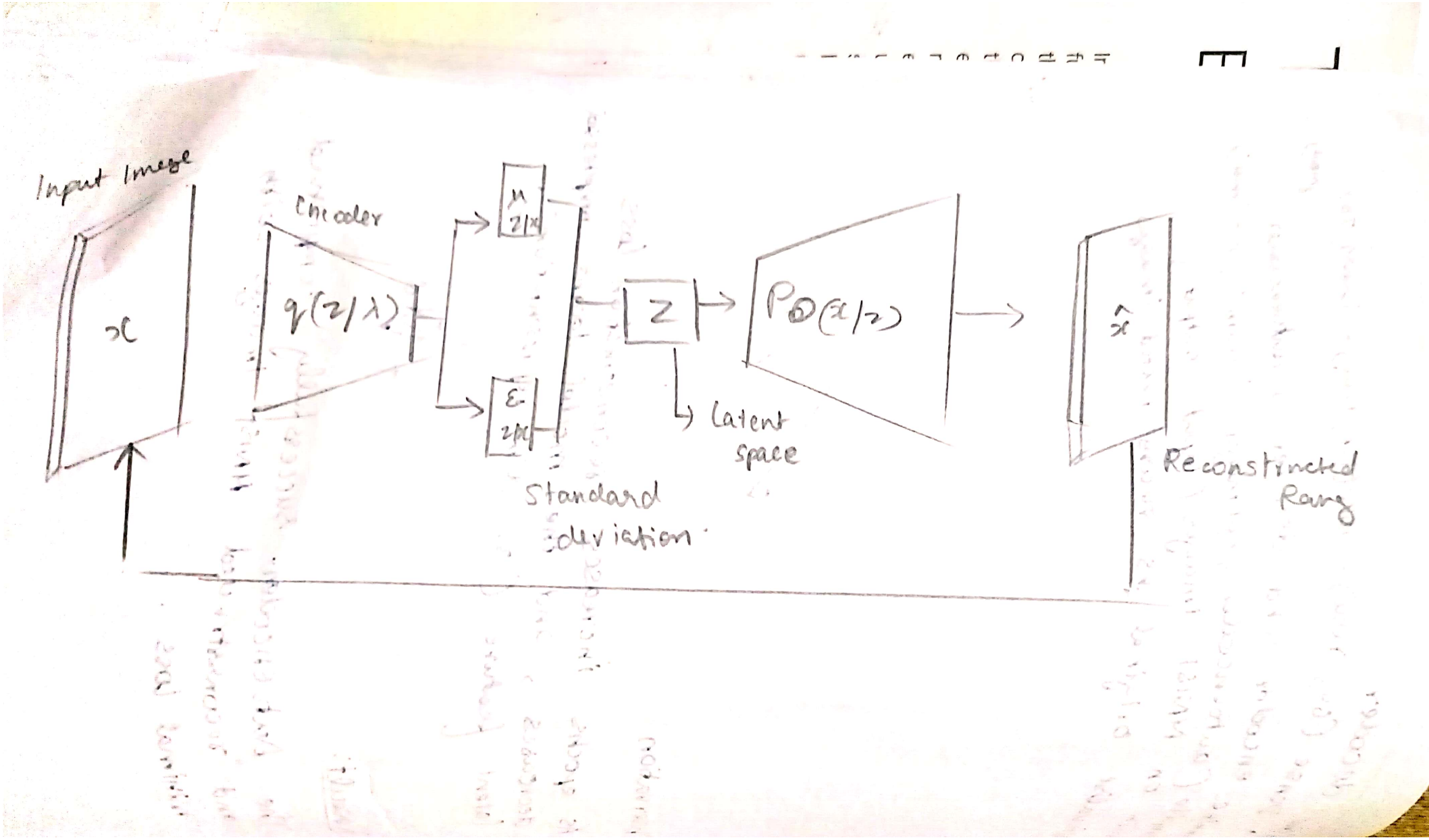
To implement a Variational Autoencoder using pytorch for learning probabilities latent representation and generating new handwritten digital images from MNIST dataset.

Objective

1. To understand the concept of VAE and how they differ from standard autoencoders.
2. To implement VAE model using pytorch that learn a latent distribution.
3. To reconstruct and generenate new images using the learned latent space.
4. To eraluate the generative Coobablity of the trained VAE.

Pseudocode:

```
BEGIN
    Import torch, torch nn, torch optim
    load MNIST data.
    Define VAE C
        Encoder
        Reparameterisation.
        Decoder
        Forward
    Define loss function
        Reconstruct loss.
```



Initialise model, optimiser(Adam)

For each epoch:

for epoch each batch in training data:

recon, μ, log σ^2 = model(input)

loss = BCE + KLD

Back propogation and update weights

Print Average epoch loss.

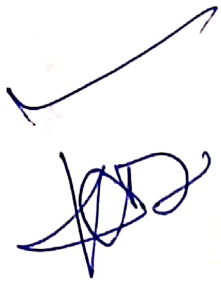
Test model

Display generated samples.

END.

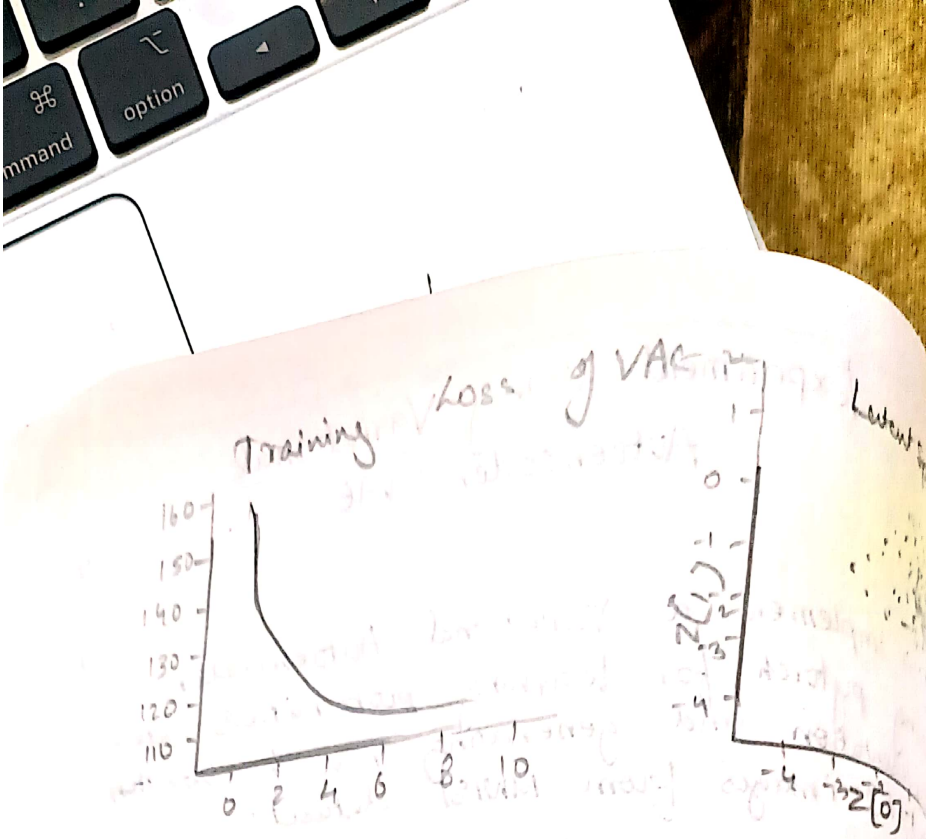
Observation:

Generated images resemble digits but with some variations due to sampling from latent space.

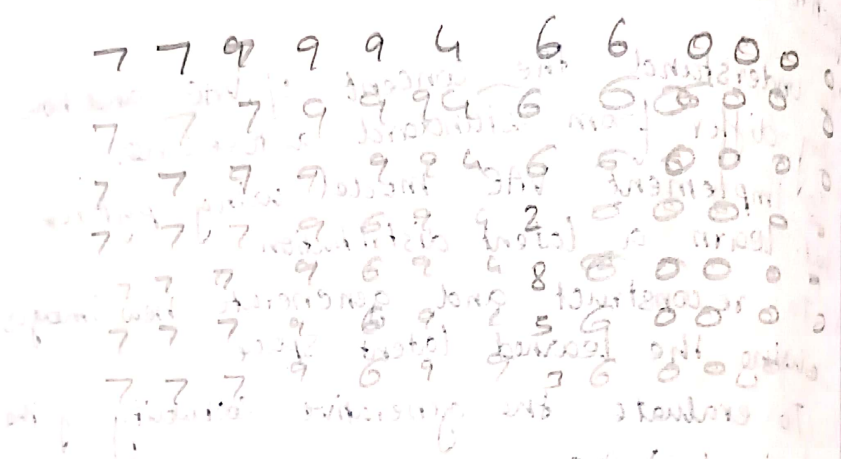


Result:

The VAE generated new digit images by learning probalistic latent distributions, Validating generative capability.



Training Loss of VAE



Epoch	KL-Loss	Reconstruction Loss	Val KL-Loss
1	10.0951	257.8873	4.0952
2	4.3199	170.6059	4.9377
3	4.8984	161.5327	5.0223
4	5.1112	157.6767	5.2313
5	5.3440	154.0489	5.4457
6	5.500-	151.4706	5.3396
7	5.5985	149.8503	5.5174
8	5.6778	148.5911	5.827
1	5.7714	147.2466	5.8609
⋮			
30	1.4627	136.0784	6.3085

13 Implement Deep Convolutional GAN to generate Complex Color images.

Aim:

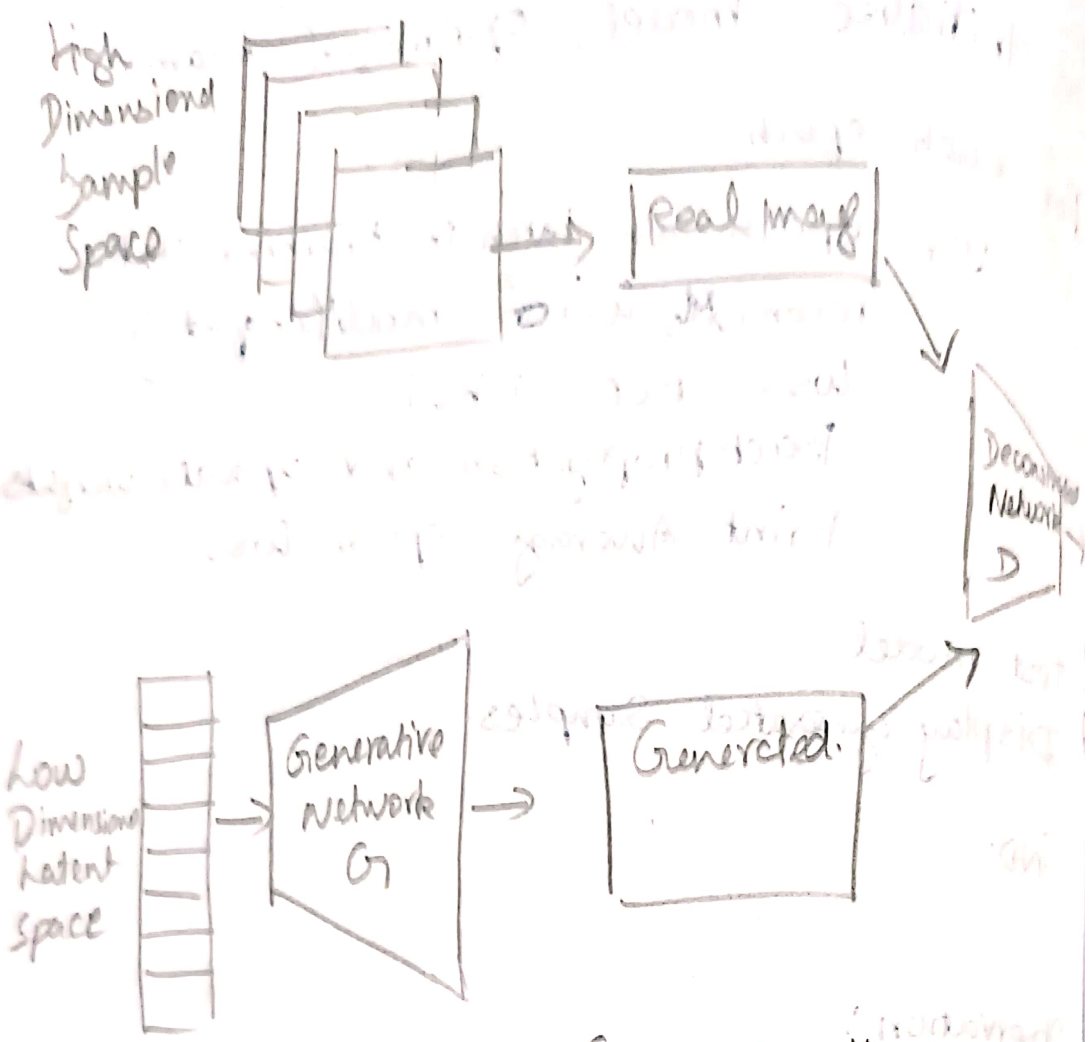
To implement a Deep Convolutional Generative Adversarial Network for generating Complex Color images

Objective

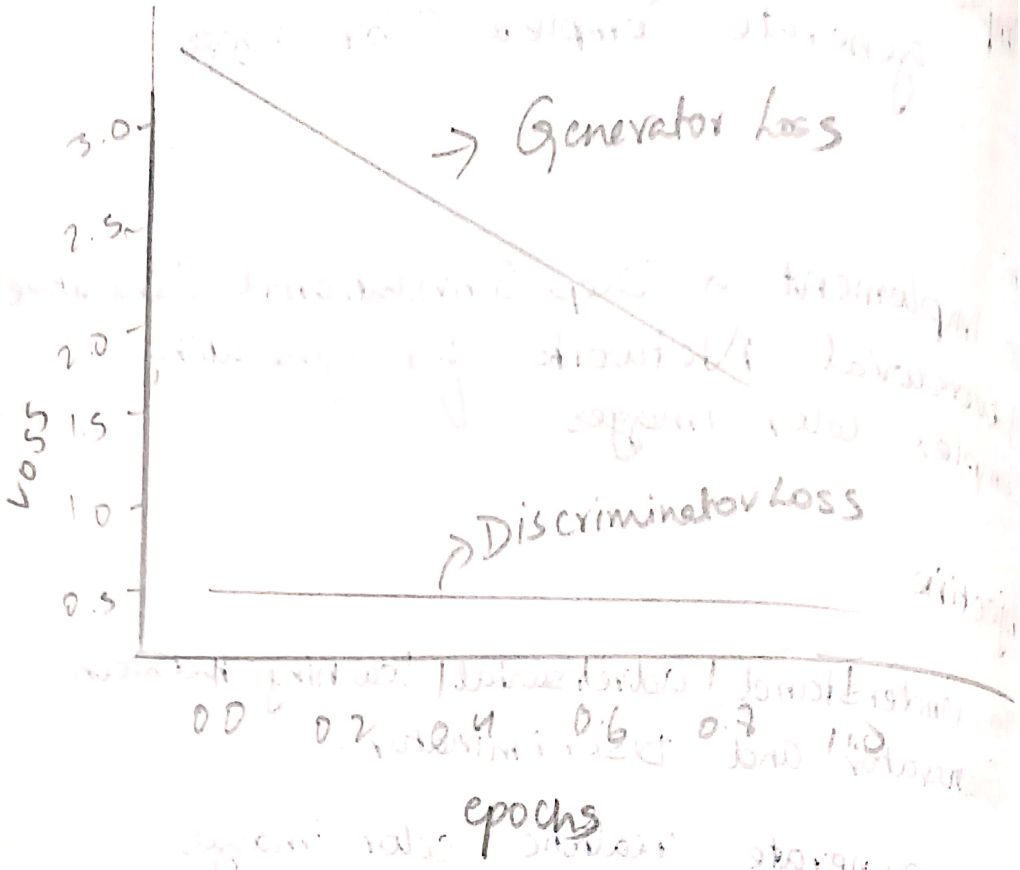
- To understand adversarial learning between Generator and Discriminator.
- To generate realistic color images.
- To observe the improvement of generated images over training epochs.
- To study latent space representation and image synthesis.

Pseudocode:

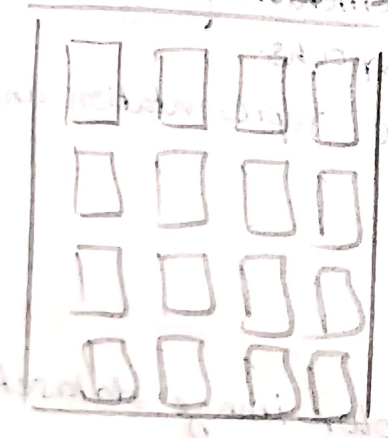
Load and normalise color image dataset
Define Generator:
Dense → Reshape → Conv2D Transpose layers (ReLU, Batch Norm) → Output(3-channels, tanh)
Define Discriminator:
Conv2D layers (Leaky ReLU, Dropout) → Dense(1, sigmoid)
Train adversarially
for each epoch:
Train Discriminator on real + fake images



GAN Architecture



Epoch 1 Gen loss : 3.1989 Disc Loss : 0.2433 Time



epoch 2/ Gen loss : 1.7013 | Disc loss : 0.2282 Time

Train Generator via combined model (Generator + frozen Discriminator)
Generate and visualise fake images.

Observation

Initially, images are noisy, after several epochs, generator starts producing realistic structures and colors.

Result

The code was executed & verified successfully.

Understanding the architecture of Pre-trained Model.

Aim:

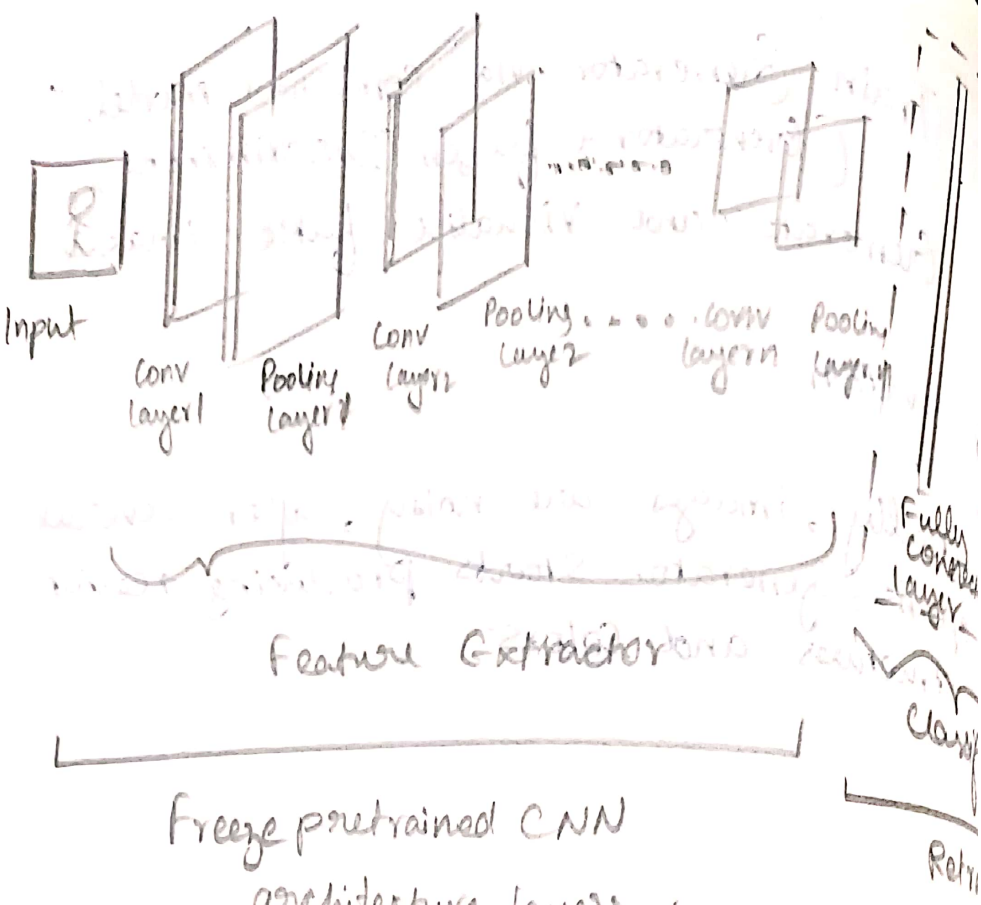
To study and analyse the architecture of pre-trained CNN models like ResNet and Inception.

Objective:

- To understand the layer-wise design of State of the art model.
- To visualise feature extraction process in deep CNNs.
- To analyse parameter distribution and feature map visualisation.
- To explore how pre trained models are used for transfer learning tasks.

Pseudo code

1. Import necessary libraries (Tensorflow/keras or pytorch)
2. Load a Pre-trained CNN model with imageNet weights.
3. Exclude top layers if needed.
4. Display and study the model summary to understand layers and parameters.
5. Load & preprocess a sample image.
6. Pass the image through the model to visualise feature maps.



	precision	recall	f1-score	support
airplane	0.33	0.66	0.44	103
automobile	0.41	0.36	0.38	89
bird	0.32	0.12	0.18	100
cat	0.60	0.03	0.06	103
deer	0.24	0.34	0.28	90
dog	0.30	0.03	0.06	86
frog	0.26	0.70	0.38	112
horse	0.45	0.25	0.33	102
ship	0.53	0.22	0.31	106
truck	0.43	0.54	0.48	109
accuracy			0.34	1000
macro avg	0.37	0.33	0.29	1000
weighted avg	0.39	0.34	0.29	1000

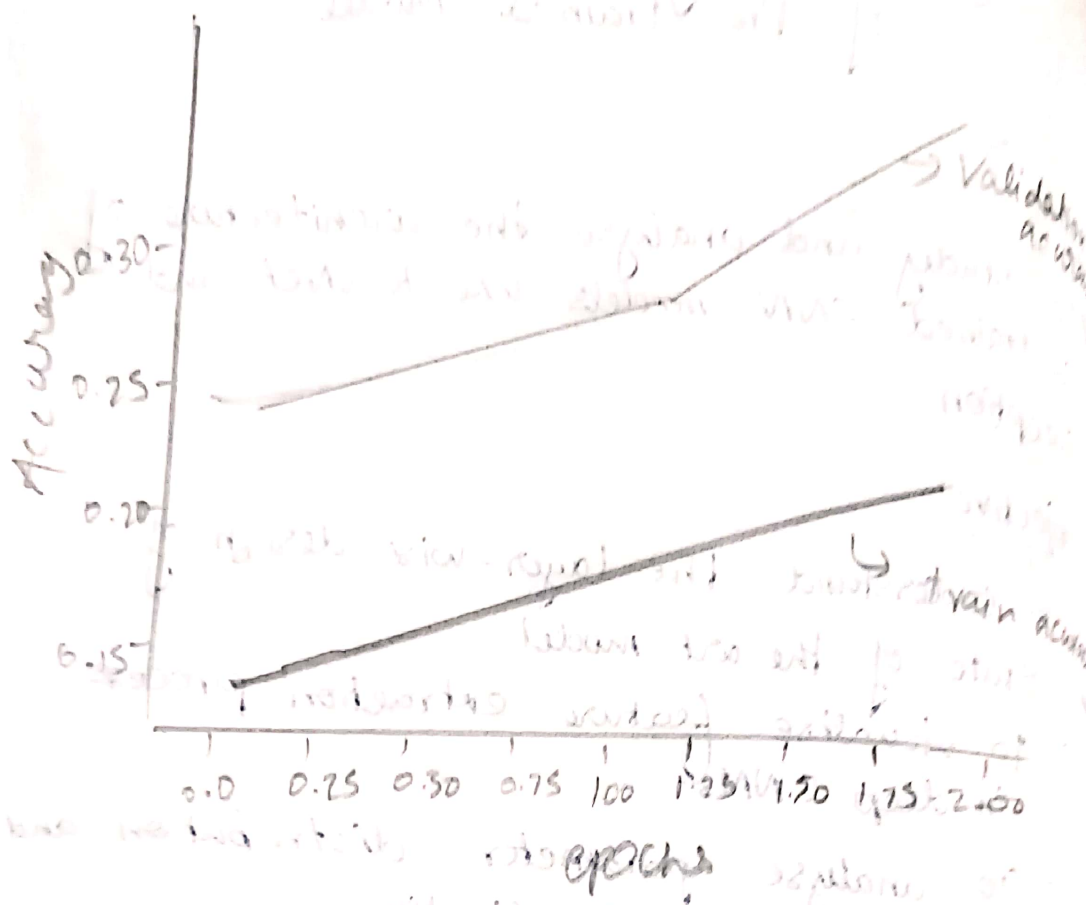
7. Analyse how different layers extract various levels of features

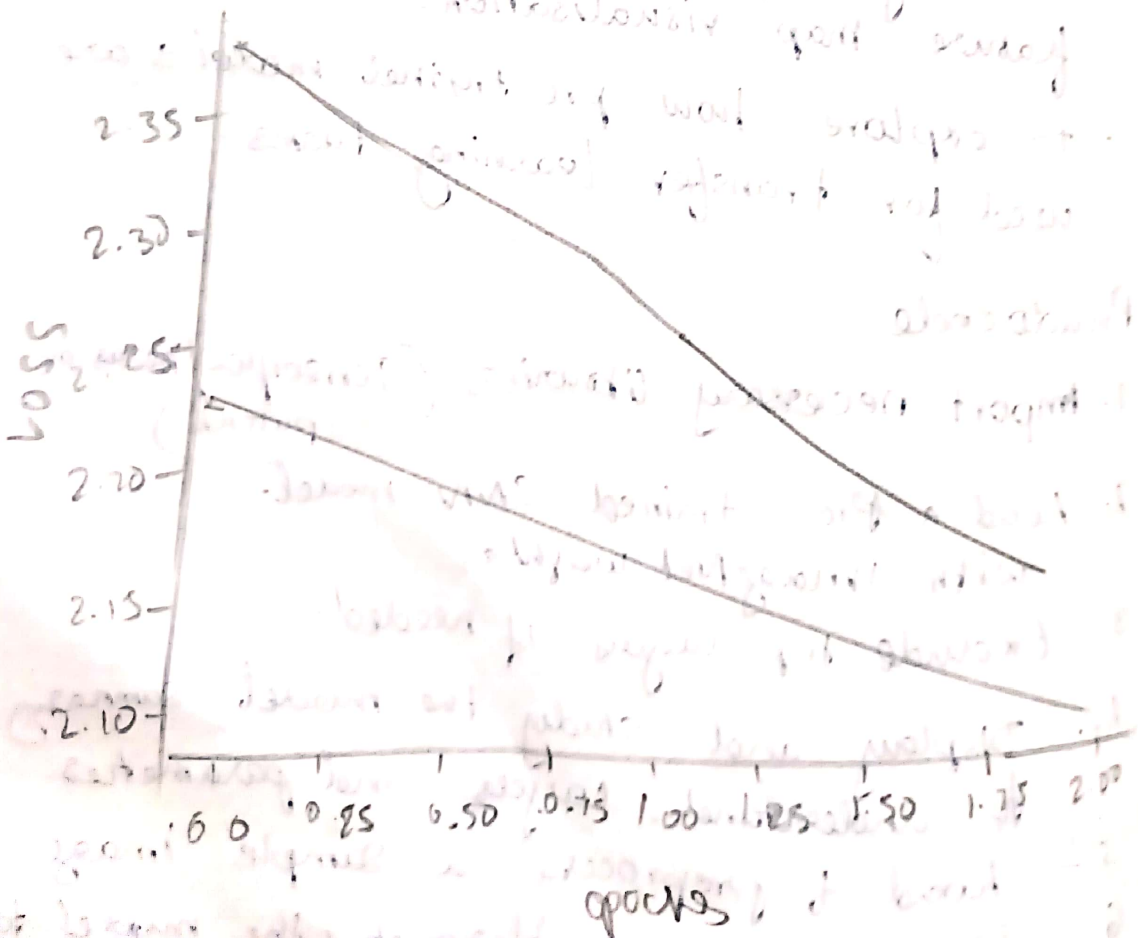
Observation

1. The model summary displayed multiple Convolutional, pooling, and fully connected layers arranged in a hierachial structure.
2. The early layers captured low-level features such as edges, textures, and simple patterns.
3. The deeper layers extracted high level abstract features like object parts and shapes
4. Feature map visualisation showed that each filter responds to specific image characteristics
5. Models like VGG16 had large no. of parameters with uniform layer depth, while ResNet used skip connection to improve learning.
6. The pre-trained model demonstrated efficient feature extraction, confirming its ability to generalise to various image types.

Result:

The code was successfully executed and verified.


Accuracy
0.30
0.25
0.20
0.15
0.0
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
epochs


Loss
2.35
2.30
2.25
2.20
2.15
2.10
0.0
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
epochs

Implement a Pre trained CNN Model as a feature Extractor Using Transfer learning.

Aim:

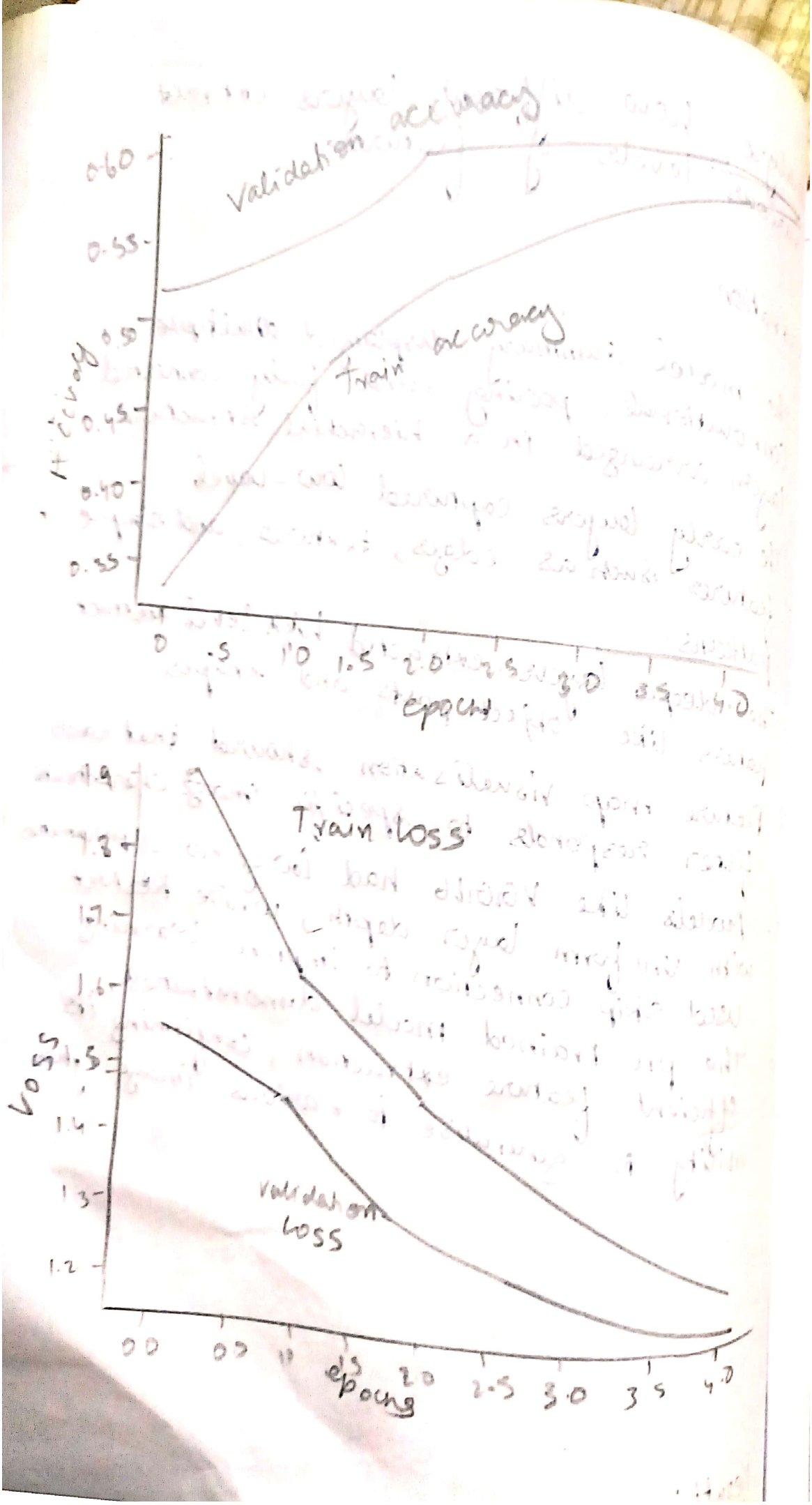
To Implement a pre-trained CNN model as a feature extractor using transfer learning for Image Classification.

Objectives:

1. To reuse pretrained CNN weights for a new dataset.
2. To extract high level image features from convolutional Layers.
3. To fine-tune the model for better performance on custom tasks.
4. To reduce training time and Computional Cost through transfer learning.

Pseudo Code:

1. Import required libraries and load a pre-trained CNN model
2. Freeze the convolutional base layers to preserve learned features.
3. Add custom layers (Flatten → Dense → Output layers with softmax)
4. Compile the model with an optimizer like Adam and suitable loss function.
5. Load and Preprocess the custom dataset.
6. Train the model on new data for limited epochs.


Validation accuracy
Train accuracy
0.60
0.55
0.50
0.45
0.40
0.35
0
.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch
Train loss
Validation loss
loss
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
epoch

epoch	accuracy	loss	Val'accuracy
1	0.2883	2.1613	0.5150
2	0.4471	1.5875	0.5440
3	0.5238	1.3993	0.6010
4	0.5606	1.2921	0.6060
5	0.6146	1.1758	0.6060

Total execute time : 96.27 seconds

7. Evaluate the model Accuracy and Visualise training performance.

Observation:

1. The model converged quickly with fewer epochs due to the pre trained feature extractor.
2. The lower convolutional layers captured general features, while the new dense layers learned task specific patterns.
3. Validation accuracy improved steadily without overfitting.
4. Training and loss curves showed stable learning behaviours.

Result:

The code was successfully verified & executed.

Implement a YOLO Model to Detect Objects.

Aim:

To implement a YOLO model for real time object detection in images / video streams.

Objectives:

1. To Understand the working of one stage object detection using YOLO architecture.
2. To detect and classify multiple objects within one image simultaneously.
3. To visualise bounding boxes & confidence scores for detected objects.
4. To evaluate the model's speed & accuracy in real time detection tasks.

PseudoCode:

1. Import required libraries (openCV, Tensorflow, Pytorch)
2. Load the pre-trained YOLO model and class labels.
3. Load the input image or video streams.
4. Preprocess the frame (resize, normalise, convert to tensor)
5. Perform detection using YOLO model.
6. Extract bounding box coordinates, class names, and confidence scores.

7. Draw bounding boxes and labels on the image.

8. Display or save the output image/ video with detected objects.

Observation:

1. The model successfully detected multiple object with bounding boxes and labels.
2. Detection speed was fast, suitable for real time application.
3. Confidence scores varied based on object clarity and lighting.
4. Bounding boxes accurately localised the detected objects in most cases.

Result:

The code was successfully executed and Verified.