DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Shahbad Daulatpur, Bawana Road, Delhi 110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CO328 :

DEEP LEARNING LAB

FILE

Submitted To:

Prof. Anil Singh Parihar
Department of Computer Science And Engineering

Submitted By:

Ayush Raghuvanshi
B. Tech. COE IIIrd Year (SEM - VI) 2K22/CO/132

Project Index

S. No	Title	Date	Sign
1	Data Visualization on MNIST Dataset.		
2	To build a deep learning model using Pytorch for predicting house prices.		
3	Implement and compare batch, mini-batch, and stochastic gradient descent in PyTorch.		
4	Exploring Optimizers (SGD, Adam, RMSProp) using PyTorch.		
5	Regularization (L1, L2, Dropout) using PyTorch.		
6	Batch Normalization & Data Augmentation in CNN using PyTorch.		
7	Implementing Early Stopping & Checkpointing using PyTorch.		
8	Handling Imbalanced Data.		
9	CNN Basics with PyTorch.		
10	Advanced CNN Architectures (VGG, ResNet).		
11	Visualizing Filters and Feature Maps.		
12	Object Detection with CNN Backbone.		
13	Transfer Learning on a Custom Dataset.		
14	Recurrent Neural Networks (RNN) for Time Series.		
15	Build an LSTM or GRU to classify sentiment or topic from text.		
16	Implement a basic Seq2Seq model (e.g., for translation or summarization) with attention.		
17	Transformer for Text Classification.		
18	Time-Series Forecasting with LSTM.		
19	Image Segmentation using UNet.		
20	Train a GAN to generate realistic images.		
21	CycleGAN for Image Style Transfer.		
22	Implement Variational Autoencoders (VAE).		
23	Implement Autoencoder for Noise Removal.		
24	Attention Mechanism in NLP.		
25	Implement Encoder-Decoder Transformer for Translation.		

Lab Work 1
Data Visualization on MNIST Dataset

1 Aim: To make a handwriting recognition model using the MNIST dataset and using the PyTorch library

2 Theory

An Artificial Neural Network (ANN) is employed for handwriting recognition on the MNIST dataset, which consists of 28×28 grayscale images of handwritten digits (0–9). The objective is to classify each image into one of 10 digit classes while visualizing the network's learned features to understand its decision-making process.

The ANN architecture is a feedforward multilayer perceptron (MLP) that processes flattened MNIST images $\mathbf{x} \in \mathbb{R}^{784}$. The model comprises:

- **Input Layer**: 784 neurons (28×28 pixels).
- **Hidden Layers**: Two layers with 512 and 256 neurons, respectively, using ReLU activations:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \quad \mathbf{h}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2),$$

  where $\mathbf{W}_1 \in \mathbb{R}^{512\times784}$, $\mathbf{W}_2 \in \mathbb{R}^{256\times512}$.
- **Output Layer**: 10 neurons with softmax activation to produce class probabilities:

$$\mathbf{y} = \text{softmax}(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3), \quad \mathbf{W}_3 \in \mathbb{R}^{10\times256}.$$

Visualization includes:

- **Weight Visualization**: The weights of the first hidden layer ($\mathbf{W}_1$) are reshaped into 28×28 images, revealing learned digit-like patterns (e.g., strokes resembling "3" or "8").
- **Feature Activations**: Hidden layer outputs ($\mathbf{h}_1$) for sample images are visualized to show which neurons activate for specific digits.
- **Confusion Matrix**: A 10×10 matrix displays classification performance, highlighting common errors (e.g., "7" misclassified as "1").

Training: The ANN is trained to minimize cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{10} t_i \log(y_i),$$

where t_i is the true label (one-hot) and y_i is the predicted probability. Training uses Adam optimizer ($\eta = 10^{-3}$) over 10 epochs on 60,000 training images, with dropout (0.2) for regularization.

3 Code

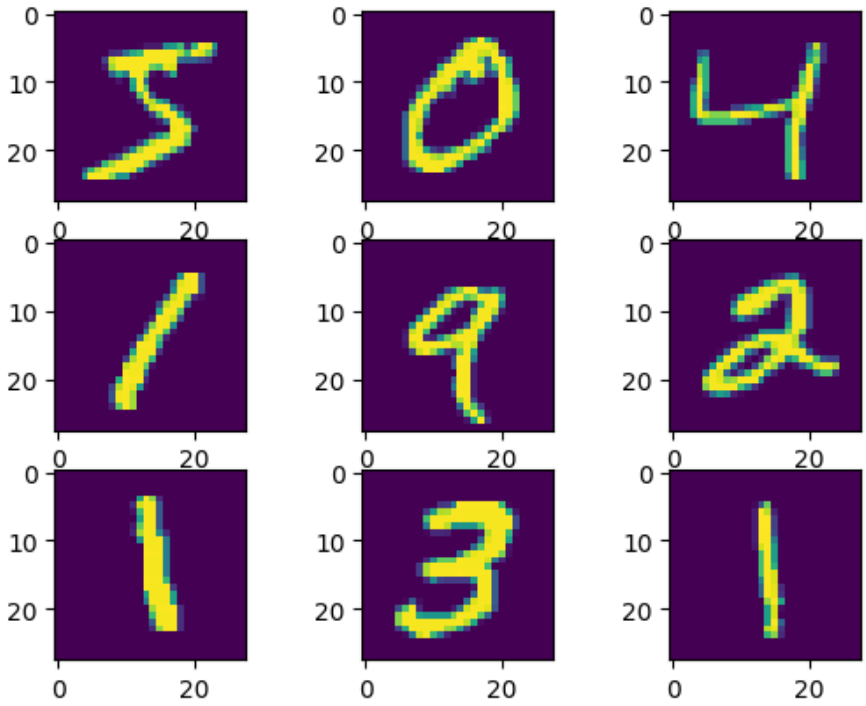
[1]:
```
# Doing all the necessary imports
import tensorflow as tf
from keras.datasets import mnist
from keras.models import Sequential
import matplotlib.pyplot as plt
import numpy as np
import cv2
import torch
import torchvision.datasets as datasets
```

[153]:
```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
```

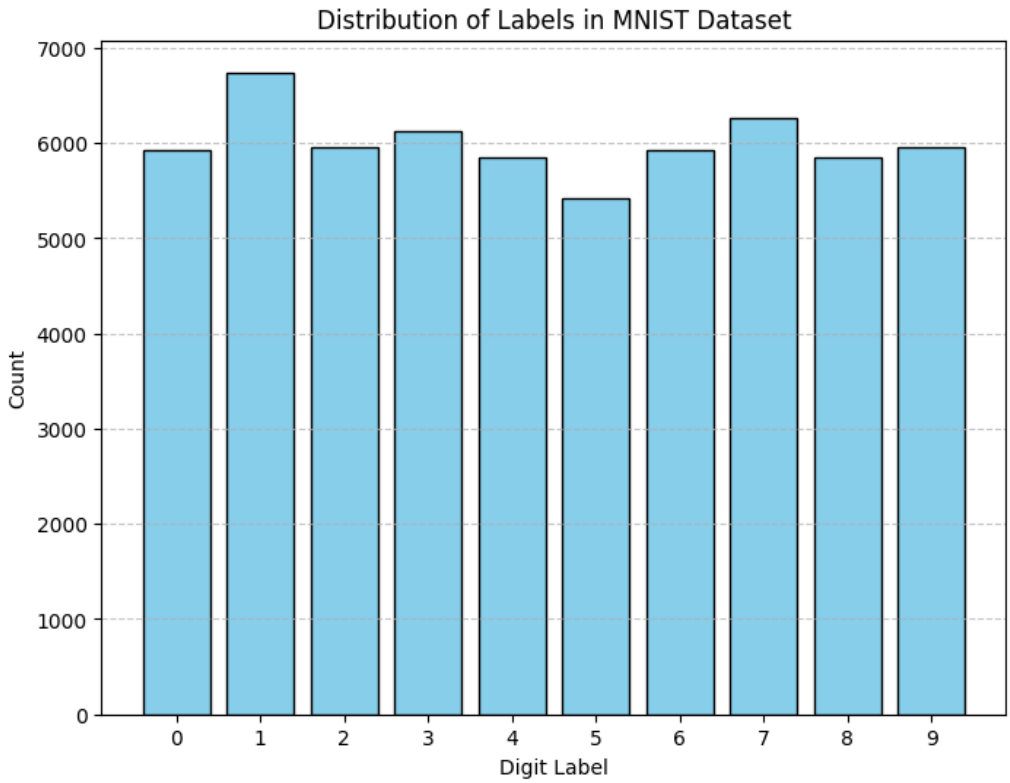
[154]:
```
print('The shape of the training inputs:', X_train.shape)
print('The shape of the training labels:',y_train.shape)
print('The shape of the testing inputs:',X_test.shape)
print('The shape of the testing labels:',y_test.shape)
```

```
The shape of the training inputs: (60000, 28, 28)
The shape of the training labels: (60000,)
The shape of the testing inputs: (10000, 28, 28)
The shape of the testing labels: (10000,)
```

[155]:
```
fig, axs = plt.subplots(3, 3)
cnt = 0
for i in range(3):
     for j in range(3):
         axs[i, j].imshow(X_train[cnt])
         cnt += 1
```



[156]:
```
label_counts = np.bincount(y_train)

# Create a bar plot
plt.figure(figsize=(8, 6))
plt.bar(range(10), label_counts, color='skyblue', edgecolor='black')
plt.xlabel('Digit Label')
plt.ylabel('Count')
plt.title('Distribution of Labels in MNIST Dataset')
plt.xticks(range(10))
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



[157]:
```
# Data Normalization
X_train = X_train / 255.0
X_test = X_test / 255.0
```

[158]:
```
# Let us import pytorch nn to make the ANN model
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
import numpy as np
```

[159]:
```
# Make the data into pytorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32).unsqueeze(1)  # Add␣
 ↪channel dimension
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).unsqueeze(1)    # Add␣
 ↪channel dimension
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
```

[160]:
```
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=100, shuffle=False)
```

[161]:
```
import torch.nn.functional as F

class WritingClassify(nn.Module):
    def __init__(self):
        super(WritingClassify, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128) # Batch normalization
        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.fc4 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = F.relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = self.fc4(x)
        return F.log_softmax(x, dim=1)

model = WritingClassify()
```

[162]:
```
criterion = nn.CrossEntropyLoss() # This is the established way of defining␣
↪things
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

[163]:
```
nn.init.xavier_uniform_(model.fc1.weight)
nn.init.xavier_uniform_(model.fc2.weight)
nn.init.xavier_uniform_(model.fc3.weight)
nn.init.xavier_uniform_(model.fc4.weight)
```

[163]:
```
Parameter containing:
tensor([[-0.0307,  0.1009, -0.1930,  0.2206,  0.2574, -0.0382, -0.2613,  0.0210,
          0.2056,  0.1312, -0.2624,  0.1811, -0.0369, -0.2792, -0.0184, -0.1848,
         -0.0943, -0.2553,  0.0217, -0.2544,  0.2619, -0.1688,  0.1915, -0.0400,
         -0.2697, -0.2482,  0.1325,  0.2419, -0.0333,  0.1951, -0.1164,  0.2136,
```

```
         -0.2612, -0.2268, -0.2801, -0.2738,  0.2216, -0.0701, -0.0630, -0.1373,
          0.0524, -0.0570,  0.1829, -0.2649, -0.1784, -0.0530, -0.2162,  0.0265,
          0.1574, -0.0545,  0.1672,  0.0640, -0.2025,  0.1642,  0.1831, -0.0079,
         -0.1753, -0.2307, -0.0825,  0.2369,  0.0689,  0.1558, -0.0751,
0.1453],
        [-0.0093, -0.2446,  0.2732,  0.1479,  0.2539,  0.0863, -0.2811,  0.0372,
         -0.1189,  0.2696,  0.2402,  0.1230, -0.2605, -0.0094, -0.0558,  0.2014,
          0.0972, -0.0704,  0.1293,  0.1371, -0.0321, -0.0969, -0.2561,  0.1035,
          0.0740,  0.0942, -0.0798, -0.0859, -0.0386,  0.1885,  0.2050,  0.0228,
          0.1922, -0.1824,  0.0035, -0.2005, -0.1566,  0.1824,  0.0354, -0.1404,
          0.0551, -0.0344, -0.0776, -0.0670, -0.1170,  0.0148, -0.1679,  0.2598,
         -0.2087,  0.0820, -0.0673, -0.2369, -0.2403,  0.1704,  0.1377, -0.1166,
          0.1189, -0.1752,  0.0495,  0.2648, -0.2701, -0.1052,  0.1403,
0.2648],
        [ 0.0173, -0.1397,  0.0996, -0.0608,  0.2216,  0.2535,  0.1784, -0.0058,
          0.0751,  0.1600,  0.2301,  0.2217,  0.0334, -0.0637, -0.1182, -0.2733,
          0.2118, -0.0733, -0.2420, -0.1606,  0.2236, -0.0131,  0.0461, -0.0164,
         -0.2737, -0.1412,  0.0303, -0.0740,  0.1228, -0.0519, -0.0689,  0.1255,
         -0.0112,  0.1137, -0.2238, -0.1399,  0.2606, -0.1042,  0.0088,  0.0241,
         -0.2014, -0.2414, -0.0807, -0.0251,  0.2137, -0.1855,  0.0082, -0.2328,
          0.0831,  0.2148,  0.2300,  0.0334, -0.1010,  0.0795, -0.1546, -0.0978,
          0.1141, -0.0739,  0.1750, -0.0003, -0.1396,  0.1722, -0.0531,
-0.2068],
        [-0.1434,  0.0192,  0.2335,  0.0901, -0.2372,  0.1186,  0.0225,  0.1023,
          0.0137, -0.0326,  0.0852, -0.1875,  0.1529,  0.2322,  0.1031,  0.0990,
          0.2188, -0.1288,  0.2564,  0.0182, -0.2524,  0.0074, -0.0298, -0.1379,
         -0.2359,  0.0138,  0.2252, -0.2541,  0.0569,  0.0827,  0.1030, -0.1354,
         -0.1798,  0.2769, -0.2035,  0.2161, -0.2539,  0.2542, -0.2396, -0.2314,
         -0.0345, -0.2415,  0.0288,  0.0275,  0.0672, -0.1671, -0.2503,  0.1248,
         -0.0507,  0.0468,  0.0749,  0.2354,  0.0926,  0.2084, -0.0096,  0.2635,
          0.0263, -0.1900,  0.1222,  0.1394, -0.2099,  0.2506,  0.1464,
-0.1333],
        [-0.0630,  0.2420,  0.2341,  0.2245,  0.1823,  0.1666, -0.0761, -0.1723,
         -0.1177, -0.0148, -0.2445, -0.2811, -0.2360, -0.1746, -0.1659, -0.1652,
         -0.0720,  0.0610, -0.1054, -0.2543, -0.0888, -0.2015, -0.2322, -0.1049,
         -0.2475,  0.0917, -0.0779,  0.1061,  0.0406,  0.1463, -0.0902,  0.2498,
          0.1618,  0.2636,  0.1925,  0.1178, -0.2367, -0.0236, -0.2434, -0.2533,
         -0.2208, -0.1021,  0.2717,  0.1283, -0.1673, -0.1347, -0.2757, -0.1253,
         -0.2705, -0.2318, -0.2317, -0.1130,  0.2253,  0.2699,  0.1832,  0.2504,
          0.2655,  0.2074, -0.0959, -0.0396,  0.2161, -0.0321, -0.2134,
0.2007],
        [ 0.0406,  0.0036,  0.2632,  0.2039, -0.0649,  0.1023,  0.1543,  0.1867,
          0.2199,  0.0769,  0.2769,  0.0751,  0.0316, -0.1103,  0.1570, -0.2807,
         -0.2556,  0.1553,  0.0370, -0.1691,  0.1652, -0.1559, -0.0699,  0.0716,
         -0.0876,  0.1539,  0.2230,  0.1737, -0.2486,  0.0848,  0.1444, -0.0434,
         -0.1646,  0.1655,  0.1179,  0.0801,  0.1081, -0.2138,  0.0783,  0.1086,
          0.1414,  0.1232,  0.1212, -0.0342, -0.0467, -0.2248,  0.0097, -0.1928,
```

```
          0.1190,  0.2193,  0.0976,  0.2615, -0.1409,  0.1905,  0.1411,  0.1511,
         -0.2727,  0.0549, -0.2510, -0.0750,  0.1567, -0.0141, -0.2320,
0.2329],
        [-0.0635,  0.0612, -0.1623, -0.1835,  0.2290, -0.1092,  0.2202, -0.1617,
          0.0070, -0.1082, -0.1286,  0.0317, -0.0036, -0.0887, -0.1597, -0.1223,
         -0.0452,  0.2013, -0.1742,  0.0379, -0.2517,  0.1414,  0.1831,  0.2546,
          0.2664,  0.1823, -0.0221, -0.2218,  0.0852, -0.0411, -0.0770, -0.1486,
          0.1023,  0.2143,  0.2611,  0.1525, -0.0320,  0.2563, -0.1932,  0.1056,
          0.0351, -0.2213,  0.1242,  0.2379, -0.0776,  0.1533, -0.0320,  0.1770,
          0.1486, -0.1405, -0.0536, -0.2398, -0.0904, -0.2381,  0.1709, -0.1902,
         -0.0074,  0.1647,  0.2412,  0.1695, -0.2454, -0.2583,  0.1508,
-0.0205],
        [ 0.0990,  0.2581, -0.2496, -0.1223, -0.1128, -0.2370, -0.2216, -0.1952,
         -0.0486,  0.0390, -0.0453, -0.0699,  0.0957,  0.2276, -0.2522,  0.0855,
          0.0584, -0.1388, -0.1608, -0.0590, -0.1408, -0.0112,  0.0138,  0.1204,
         -0.0163, -0.1572,  0.1058,  0.0323,  0.2524, -0.0876, -0.0230,  0.2571,
          0.2442, -0.0937,  0.2229, -0.0230, -0.1481,  0.0773,  0.1485, -0.0573,
          0.2494,  0.2708, -0.0033, -0.0543,  0.1924,  0.1016,  0.2010,  0.2625,
          0.1811,  0.0091, -0.2635, -0.2576, -0.1416, -0.2233, -0.2706,  0.0730,
         -0.0737,  0.2363, -0.1725,  0.1014,  0.0727, -0.1411,  0.2566,
0.2628],
        [ 0.1713, -0.2211,  0.1847,  0.2760, -0.2721,  0.1040,  0.0576, -0.0821,
          0.0129, -0.2699, -0.0799, -0.0369,  0.0536, -0.0813, -0.2206, -0.0435,
          0.1108, -0.0967,  0.0885,  0.0776,  0.0782,  0.2717, -0.1351,  0.1194,
         -0.0046,  0.0082, -0.1982, -0.0464,  0.2062, -0.2536, -0.1429, -0.2517,
         -0.1864, -0.1789, -0.2356, -0.2277, -0.0127,  0.1125, -0.1766,  0.1424,
          0.2321,  0.0004, -0.1000,  0.2648, -0.1492, -0.2264,  0.1455,  0.2663,
          0.0631,  0.1127,  0.0487, -0.1006, -0.1427,  0.0044,  0.1205,  0.0469,
          0.0032,  0.1910,  0.0108,  0.1810, -0.1195,  0.0417, -0.1734,
-0.0500],
        [ 0.2573, -0.0551,  0.2206, -0.0818, -0.0069,  0.2500, -0.2031,  0.2411,
          0.1152, -0.1928, -0.0674,  0.2790, -0.2380,  0.0938, -0.0881, -0.1620,
         -0.0879,  0.0430,  0.2017,  0.2360,  0.0096, -0.1020,  0.0867, -0.1532,
          0.1818, -0.0309,  0.0691, -0.0215, -0.1913, -0.0785, -0.2314, -0.1434,
         -0.1479,  0.2756,  0.0474, -0.1724, -0.2248,  0.1032,  0.1527, -0.2537,
          0.2841, -0.2204,  0.0914, -0.0653,  0.0783, -0.1823,  0.0993, -0.0870,
          0.0618, -0.0539,  0.0872, -0.2234, -0.2042,  0.1632,  0.1638, -0.0484,
          0.1134, -0.0319,  0.1530,  0.1380, -0.1821, -0.0233, -0.1559,
0.0180]],
       requires_grad=True)
```

[164]:
```
# Train the model

# This is how a model is trained in pytorch, in a very raw way by looping over␣
 ↪each epoch

epochs = 10
```

```
for epoch in range(epochs):
    model.train()
    running_loss = 0
    for image, labels in train_loader:
        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(image)
        loss = criterion(outputs, labels)

        # Backwards pass
        loss.backward()

        # Optimize
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.
↪4f}")
```

```
Epoch 1/10, Loss: 0.9972
Epoch 2/10, Loss: 0.4224
Epoch 3/10, Loss: 0.3064
Epoch 4/10, Loss: 0.2496
Epoch 5/10, Loss: 0.2155
Epoch 6/10, Loss: 0.1871
Epoch 7/10, Loss: 0.1664
Epoch 8/10, Loss: 0.1517
Epoch 9/10, Loss: 0.1423
Epoch 10/10, Loss: 0.1320
```

[167]:
```
model.eval()
correct = 0
total = 0

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

accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")
```

```
Test Accuracy: 97.75%
```

Lab Work 2
House Price Prediction using Deep Learning

1 Aim: To develope a regression model on House Price Prediction and use MSE and MAE and compare performance

1.1 Theory

Deep Learning, via an Artificial Neural Network (ANN), is applied to predict house prices using a dataset such as the California Housing dataset, which includes features like median income, house age, and rooms per household. The objective is to regress a continuous house price $\hat{y} \in \mathbb{R}$ from input features $\mathbf{x} \in \mathbb{R}^d$ (e.g., $d = 8$ features) while visualizing the network's learned representations and prediction performance to understand its behavior.

The ANN is a feedforward multilayer perceptron (MLP) with the following architecture:

- **Input Layer**: d neurons corresponding to normalized feature values.
- **Hidden Layers**: Two layers with 128 and 64 neurons, respectively, using ReLU activations to model non-linear relationships:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \quad \mathbf{h}_2 = \text{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2),$$

  where $\mathbf{W}_1 \in \mathbb{R}^{128\times d}$, $\mathbf{W}_2 \in \mathbb{R}^{64\times 128}$.
- **Output Layer**: A single neuron with linear activation to predict the house price:

$$\hat{y} = \mathbf{W}_3\mathbf{h}_2 + b_3, \quad \mathbf{W}_3 \in \mathbb{R}^{1\times 64}.$$

Visualization techniques enhance interpretability:

- **Feature Importance**: Weights of the first hidden layer ($\mathbf{W}_1$) are analyzed to quantify the influence of input features (e.g., median income vs. house age), often visualized as a bar plot.
- **Activation Maps**: Outputs of the first hidden layer ($\mathbf{h}_1$) for sample inputs are visualized to identify neurons sensitive to specific feature patterns.
- **Prediction vs. Actual Plot**: A scatter plot of predicted ($\hat{y}$) vs. actual (y) house prices, with a diagonal line indicating perfect predictions, highlights model accuracy and error patterns.

Training: The ANN minimizes the Mean Squared Error (MSE) loss to optimize predictions:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2,$$

where y_i is the true price and $\hat{y}_i$ is the predicted price. Training is conducted on a dataset split (e.g., 80% train, 20% test), typically using an optimizer like Adam with regularization (e.g., dropout or weight decay) to prevent overfitting.

2 Code

2.1 Import the libraries

[14]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torch.optim as optim
```

2.2 Load the dataset

[3]:
```
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
```

[4]:
```
housing
```

[4]:
```
{'data': array([[   8.3252    ,   41.        ,    6.98412698, ...,
2.55555556,
           37.88      , -122.23      ],
        [   8.3014    ,   21.        ,    6.23813708, ...,    2.10984183,
           37.86      , -122.22      ],
        [   7.2574    ,   52.        ,    8.28813559, ...,    2.80225989,
           37.85      , -122.24      ],
        ...,
        [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351 ,
           39.43      , -121.22      ],
        [   1.8672    ,   18.        ,    5.32951289, ...,    2.12320917,
           39.43      , -121.32      ],
        [   2.3886    ,   16.        ,    5.25471698, ...,    2.61698113,
           39.37      , -121.24      ]]),
 'target': array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894]),
 'frame': None,
 'target_names': ['MedHouseVal'],
 'feature_names': ['MedInc',
  'HouseAge',
  'AveRooms',
  'AveBedrms',
  'Population',
  'AveOccup',
  'Latitude',
  'Longitude'],
 'DESCR': '.. _california_housing_dataset:\n\nCalifornia Housing
```

```
dataset\n-------------------------\n\n**Data Set Characteristics:**\n\n:Number
of Instances: 20640\n\n:Number of Attributes: 8 numeric, predictive attributes
and the target\n\n:Attribute Information:\n    - MedInc        median income in
block group\n    - HouseAge      median house age in block group\n    - AveRooms
average number of rooms per household\n    - AveBedrms     average number of
bedrooms per household\n    - Population    block group population\n    -
AveOccup      average number of household members\n    - Latitude      block
group latitude\n    - Longitude     block group longitude\n\n:Missing Attribute
Values: None\n\nThis dataset was obtained from the StatLib
repository.\nhttps://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html\n\nThe
target variable is the median house value for California districts,\nexpressed
in hundreds of thousands of dollars ($100,000).\n\nThis dataset was derived from
the 1990 U.S. census, using one row per census\nblock group. A block group is
the smallest geographical unit for which the U.S.\nCensus Bureau publishes
sample data (a block group typically has a population\nof 600 to 3,000
people).\n\nA household is a group of people residing within a home. Since the
average\nnumber of rooms and bedrooms in this dataset are provided per
household, these\ncolumns may take surprisingly large values for block groups
with few households\nand many empty houses, such as vacation resorts.\n\nIt can
be downloaded/loaded using
the\n:func:`sklearn.datasets.fetch_california_housing` function.\n\n.. rubric::
References\n\n- Pace, R. Kelley and Ronald Barry, Sparse Spatial
Autoregressions,\n  Statistics and Probability Letters, 33 (1997) 291-297\n'}
```

2.3 Defining the dependent and the independent variables

[5]:
```
X = housing.data
y = housing.target
```

2.4 Train-test split

[6]:
```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
↪random_state=42)
```

2.5 Now let's normalize the dataset for better performance

[7]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

[8]:
```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32).view(-1, 1)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.float32).view(-1, 1)
```

```
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)
```

2.6 Let's define the model which is a simple linear regression model

[9]:
```
class LinearRegression(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_dim, 1)

    def forward(self, x):
        return self.linear(x)
```

[10]:
```
model = LinearRegression(X_train.shape[1])
```

2.7 For properly doing a comparitive analysis let's make a function which we can modify based on the loss function

[11]:
```
def train_model(model, criterion, optimizer, epochs=100):
    train_losses, val_losses = [], []
    for epoch in range(epochs):
        model.train()
        predictions = model(X_train_tensor)
        loss = criterion(predictions, y_train_tensor)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        model.eval()
        with torch.no_grad():
            val_predictions = model(X_val_tensor)
            val_loss = criterion(val_predictions, y_val_tensor)

        train_losses.append(loss.item())
        val_losses.append(val_loss.item())

        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Train Loss: {loss.item():.4f},␣
 ↪Val Loss: {val_loss.item():.4f}')

    return train_losses, val_losses
```

[12]:
```
# Train with MSE
criterion_mse = nn.MSELoss()
optimizer_mse = optim.Adam(model.parameters(), lr=0.01)
mse_train_losses, mse_val_losses = train_model(model, criterion_mse,␣
 ↪optimizer_mse)
```

```
# Train with MAE
criterion_mae = nn.L1Loss()
optimizer_mae = optim.Adam(model.parameters(), lr=0.01)
mae_train_losses, mae_val_losses = train_model(model, criterion_mae,␣
 ↪optimizer_mae)
```

```
Epoch [10/100], Train Loss: 5.8727, Val Loss: 5.6879
Epoch [20/100], Train Loss: 5.2404, Val Loss: 5.1389
Epoch [30/100], Train Loss: 4.7161, Val Loss: 4.6537
Epoch [40/100], Train Loss: 4.2513, Val Loss: 4.2070
Epoch [50/100], Train Loss: 3.8447, Val Loss: 3.8103
Epoch [60/100], Train Loss: 3.4868, Val Loss: 3.4614
Epoch [70/100], Train Loss: 3.1665, Val Loss: 3.1506
Epoch [80/100], Train Loss: 2.8774, Val Loss: 2.8695
Epoch [90/100], Train Loss: 2.6149, Val Loss: 2.6130
Epoch [100/100], Train Loss: 2.3766, Val Loss: 2.3790
Epoch [10/100], Train Loss: 1.2313, Val Loss: 1.2255
Epoch [20/100], Train Loss: 1.1324, Val Loss: 1.1257
Epoch [30/100], Train Loss: 1.0362, Val Loss: 1.0277
Epoch [40/100], Train Loss: 0.9428, Val Loss: 0.9354
Epoch [50/100], Train Loss: 0.8544, Val Loss: 0.8491
Epoch [60/100], Train Loss: 0.7743, Val Loss: 0.7703
Epoch [70/100], Train Loss: 0.7058, Val Loss: 0.7021
Epoch [80/100], Train Loss: 0.6500, Val Loss: 0.6454
Epoch [90/100], Train Loss: 0.6069, Val Loss: 0.6021
Epoch [100/100], Train Loss: 0.5753, Val Loss: 0.5711
```

2.8 Now we evaluate the model two times with the different loss function

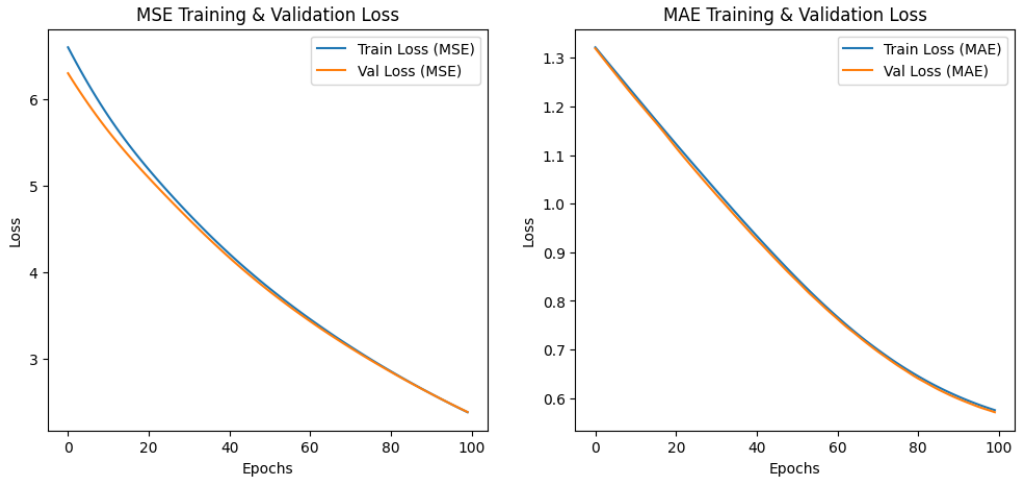
[15]:
```
model.eval()
with torch.no_grad():
    test_predictions = model(X_test_tensor)
    test_predictions = test_predictions.numpy().flatten()
    y_test_np = y_test_tensor.numpy().flatten()
    final_mse = mean_squared_error(y_test_np, test_predictions)
    final_mae = mean_absolute_error(y_test_np, test_predictions)
```

[16]:
```
print(f'Final Test MSE: {final_mse:.4f}')
print(f'Final Test MAE: {final_mae:.4f}')
```

```
Final Test MSE: 0.6742
Final Test MAE: 0.5556
```

2.9 We can now use matplotlib to visualise our results

[17]:
```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(mse_train_losses, label='Train Loss (MSE)')
plt.plot(mse_val_losses, label='Val Loss (MSE)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('MSE Training & Validation Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(mae_train_losses, label='Train Loss (MAE)')
plt.plot(mae_val_losses, label='Val Loss (MAE)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('MAE Training & Validation Loss')
plt.legend()

plt.show()
```



Lab Work 3
Comparing different forms of Gradient Descent

1 Aim: Do a comparitive analysis on Gradient Descent variations (Batch, Mini-Batch and Stochastic) and plot the loss differences

2 Theory

Gradient descent is an optimization algorithm used to minimize a loss function $\mathcal{L}(\theta)$ in machine learning, where $\theta \in \mathbb{R}^d$ represents model parameters. The goal is to iteratively update θ to find the optimal parameters that minimize $\mathcal{L}$. Three main variations—full-batch, stochastic, and mini-batch gradient descent—differ in how they compute gradients and update parameters, balancing computational efficiency, convergence speed, and stability.

Full-Batch Gradient Descent computes the gradient of the loss over the entire dataset of N samples:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell(\theta; \mathbf{x}_i, y_i),$$

where $\ell(\theta; \mathbf{x}_i, y_i)$ is the loss for sample i. The parameter update is:

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

with learning rate η. This method provides accurate gradient estimates, ensuring stable convergence toward the global minimum for convex losses. However, it is computationally expensive for large N, requiring significant memory and time per iteration.

Stochastic Gradient Descent (SGD) computes the gradient using a single randomly selected sample:

$$\nabla_\theta \mathcal{L}(\theta) \approx \nabla_\theta \ell(\theta; \mathbf{x}_i, y_i), \quad i \sim \text{Uniform}(1, N).$$

The update is applied after each sample, making iterations fast and memory-efficient. SGD introduces high variance in gradient estimates, leading to noisy updates that can escape local minima but may oscillate around the optimum, slowing convergence. It is well-suited for large datasets and online learning.

Mini-Batch Gradient Descent strikes a balance by computing the gradient over a small, randomly selected subset (mini-batch) of b samples ($1 < b < N$):

$$\nabla_\theta \mathcal{L}(\theta) \approx \frac{1}{b} \sum_{i \in B} \nabla_\theta \ell(\theta; \mathbf{x}_i, y_i),$$

where B is the mini-batch. Updates are performed after each mini-batch, offering a trade-off: lower variance than SGD (smoother convergence) and lower computational cost than full-batch (faster iterations). Typical batch sizes range from 32 to 256, depending on hardware and task complexity.

Training Dynamics: All variants aim to minimize $\mathcal{L}$ via iterative updates, but their convergence differs:

- Full-batch: Slow but stable, with fewer updates per epoch.

- SGD: Fast updates but noisy, requiring more epochs for convergence.
- Mini-batch: Balances speed and stability, widely used in deep learning.

Learning rate η is critical; adaptive optimizers (e.g., Adam) often enhance performance. Regularization (e.g., weight decay) mitigates overfitting across variants.

3 Code

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
```

[2]:
```
device = torch.device("mps")
```

[3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to [0, 1] range
])
```

[4]:
```
train_data = datasets.MNIST(root='./data', train=True, download=True,␣
↪transform=transform)
```

[5]:
```
test_data = datasets.MNIST(root='./data', train=False, download=True,␣
↪transform=transform)
```

[6]:
```
batch_size = 64
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)
```

[7]:
```
class GDVarTester(nn.Module):
    def __init__(self):
        super(GDVarTester, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # Input layer (28x28 flattened image␣
↪to 128 neurons)
        self.fc2 = nn.Linear(128, 64)       # Hidden layer
        self.fc3 = nn.Linear(64, 10)        # Output layer (10 classes for␣
↪digits 0-9)
        self.softmax = nn.Softmax(dim=1)    # Softmax for classification

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten the image
        x = torch.relu(self.fc1(x))  # First hidden layer with ReLU
        x = torch.relu(self.fc2(x))  # Second hidden layer with ReLU
        x = self.fc3(x)  # Output layer
```

```
        return x
```

[8]:
```
model = GDVarTester()
```

[9]:
```
criterion = nn.CrossEntropyLoss()

# First let us train using Stochastic Gradient Descent
optimizer = optim.SGD(model.parameters(), lr=0.0001)
```

[19]:
```
def train_model(optimizer_type='SGD', num_epochs=15):
    model = GDVarTester()
    model.to(device)
    criterion = nn.CrossEntropyLoss()  # Cross-entropy loss for classification
    optimizer = None

    # Choose optimizer based on gradient descent type
    if optimizer_type == 'SGD':
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
    elif optimizer_type == 'Batch GD':
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
    elif optimizer_type == 'Mini-batch GD':
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    train_losses = []
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            # Zero the gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)

            # Calculate loss
            loss = criterion(outputs, labels)

            # Backward pass and optimize
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
```

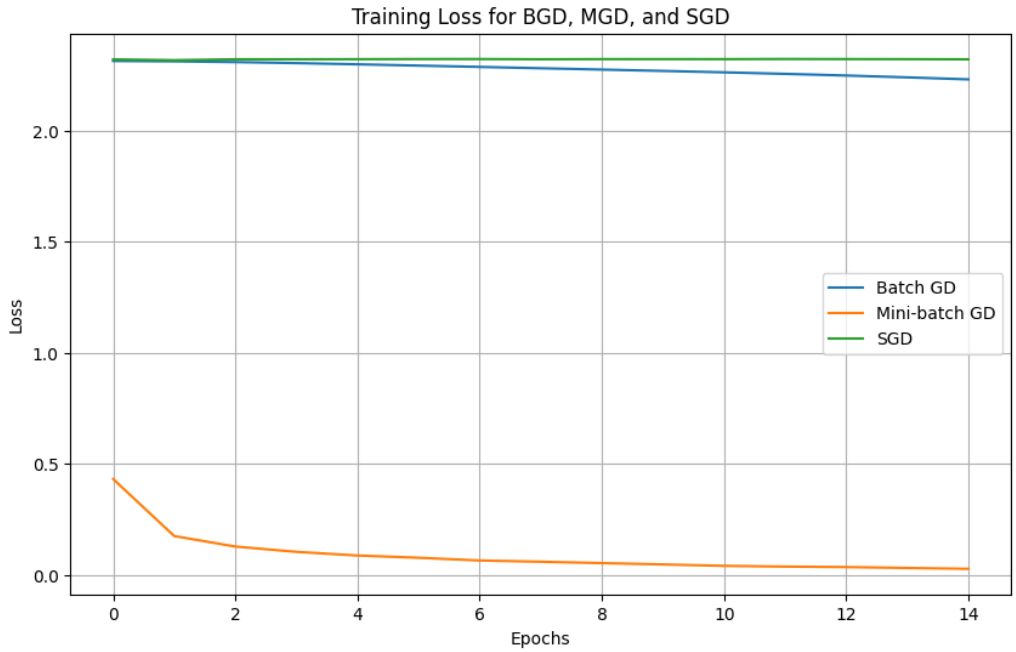
```
            # Calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        # Track the average loss and accuracy for the epoch
        avg_loss = running_loss / len(train_loader)
        train_accuracy = 100 * correct / total
        train_losses.append(avg_loss)

        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}, Accuracy:␣
↪{train_accuracy:.2f}%")

    return model, train_losses, train_accuracy

# Step 4: Train the models for BGD, MGD, and SGD
batch_sizes = [len(train_data), 64, 1]  # BGD: all samples, MGD: 64 samples, SGD:
↪ 1 sample

results = {}
accuracies = {}
for method, batch_size in zip(['Batch GD', 'Mini-batch GD', 'SGD'], batch_sizes):
    print(f"\nTraining with {method}...")
    if method == 'Batch GD':
        train_loader = DataLoader(train_data, batch_size=len(train_data),␣
↪shuffle=True)  # Full dataset in one batch
    elif method == 'Mini-batch GD':
        train_loader = DataLoader(train_data, batch_size=64, shuffle=True)  #␣
↪Mini-batches of size 64
    elif method == 'SGD':
        train_loader = DataLoader(train_data, batch_size=1, shuffle=True)  # One␣
↪sample at a time

    model, train_losses, train_accuracy = train_model(optimizer_type=method)
    results[method] = train_losses
    accuracies[method] = train_accuracy
```

```
Training with Batch GD...
Epoch [1/15], Loss: 2.3154, Accuracy: 8.75%
Epoch [2/15], Loss: 2.3136, Accuracy: 8.82%
Epoch [3/15], Loss: 2.3103, Accuracy: 8.92%
Epoch [4/15], Loss: 2.3057, Accuracy: 9.05%
Epoch [5/15], Loss: 2.3003, Accuracy: 9.19%
Epoch [6/15], Loss: 2.2945, Accuracy: 10.27%
Epoch [7/15], Loss: 2.2885, Accuracy: 15.50%
Epoch [8/15], Loss: 2.2826, Accuracy: 18.61%
```

```
Epoch [9/15], Loss: 2.2765, Accuracy: 19.43%
Epoch [10/15], Loss: 2.2704, Accuracy: 19.49%
Epoch [11/15], Loss: 2.2641, Accuracy: 19.62%
Epoch [12/15], Loss: 2.2573, Accuracy: 20.11%
Epoch [13/15], Loss: 2.2497, Accuracy: 21.17%
Epoch [14/15], Loss: 2.2414, Accuracy: 23.05%
Epoch [15/15], Loss: 2.2324, Accuracy: 25.48%

Training with Mini-batch GD...
Epoch [1/15], Loss: 0.4311, Accuracy: 87.01%
Epoch [2/15], Loss: 0.1741, Accuracy: 94.65%
Epoch [3/15], Loss: 0.1267, Accuracy: 96.11%
Epoch [4/15], Loss: 0.1028, Accuracy: 96.84%
Epoch [5/15], Loss: 0.0863, Accuracy: 97.31%
Epoch [6/15], Loss: 0.0761, Accuracy: 97.53%
Epoch [7/15], Loss: 0.0638, Accuracy: 98.00%
Epoch [8/15], Loss: 0.0583, Accuracy: 98.17%
Epoch [9/15], Loss: 0.0521, Accuracy: 98.33%
Epoch [10/15], Loss: 0.0461, Accuracy: 98.49%
Epoch [11/15], Loss: 0.0397, Accuracy: 98.68%
Epoch [12/15], Loss: 0.0363, Accuracy: 98.84%
Epoch [13/15], Loss: 0.0343, Accuracy: 98.84%
Epoch [14/15], Loss: 0.0302, Accuracy: 99.01%
Epoch [15/15], Loss: 0.0263, Accuracy: 99.12%

Training with SGD...
Epoch [1/15], Loss: 2.3226, Accuracy: 10.46%
Epoch [2/15], Loss: 2.3188, Accuracy: 10.68%
Epoch [3/15], Loss: 2.3230, Accuracy: 10.32%
Epoch [4/15], Loss: 2.3231, Accuracy: 10.21%
Epoch [5/15], Loss: 2.3234, Accuracy: 10.48%
Epoch [6/15], Loss: 2.3240, Accuracy: 10.05%
Epoch [7/15], Loss: 2.3240, Accuracy: 10.26%
Epoch [8/15], Loss: 2.3231, Accuracy: 10.18%
Epoch [9/15], Loss: 2.3237, Accuracy: 10.21%
Epoch [10/15], Loss: 2.3237, Accuracy: 10.24%
Epoch [11/15], Loss: 2.3236, Accuracy: 10.35%
Epoch [12/15], Loss: 2.3242, Accuracy: 10.18%
Epoch [13/15], Loss: 2.3238, Accuracy: 10.16%
Epoch [14/15], Loss: 2.3235, Accuracy: 10.31%
Epoch [15/15], Loss: 2.3229, Accuracy: 10.41%
```

[22]:
```python
# Step 5: Plot the training losses for comparison
plt.figure(figsize=(10,6))
for method, losses in results.items():
    plt.plot(losses, label=method)
```

```
plt.title("Training Loss for BGD, MGD, and SGD")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```



Lab Work 4
Exploring Optimizers (SGD, Adam, RMSProp) using PyTorch

1 Aim: To explore the differences in performance of a neural network when using different optimization algorithms like SGD, Adam, RMSProp

2 Theory

Optimization algorithms are critical for minimizing a loss function $\mathcal{L}(\theta)$ in deep learning, where $\theta \in \mathbb{R}^d$ represents model parameters. The objective is to iteratively update θ to find the optimal parameters that minimize $\mathcal{L}$. Three prominent optimizers—Stochastic Gradient Descent (SGD), Adam, and RMSprop—are compared, each offering distinct mechanisms to balance convergence speed, stability, and computational efficiency.

Stochastic Gradient Descent (SGD) updates parameters using the gradient of a mini-batch of b samples:

$$\mathbf{g}_t = \frac{1}{b} \sum_{i \in B} \nabla_\theta \ell(\theta_t; \mathbf{x}_i, y_i), \quad \theta_{t+1} = \theta_t - \eta \mathbf{g}_t,$$

where η is the learning rate. SGD is simple and memory-efficient but sensitive to η, often requiring careful tuning. Momentum variants (e.g., $\theta_{t+1} = \theta_t - \eta \mathbf{g}_t + \alpha \Delta\theta_{t-1}$) accelerate convergence by incorporating past gradients, reducing oscillations.

RMSprop (Root Mean Square Propagation) adapts the learning rate per parameter by normalizing gradients using a moving average of squared gradients:

$$E[\mathbf{g}^2]_t = \rho E[\mathbf{g}^2]_{t-1} + (1-\rho)\mathbf{g}_t^2, \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[\mathbf{g}^2]_t + \epsilon}} \mathbf{g}_t,$$

where ρ (e.g., 0.9) is the decay rate, and ϵ (e.g., 10^{-8}) prevents division by zero. RMSprop mitigates SGD's sensitivity to gradient scale, enabling faster convergence in non-convex settings, but may struggle with sharp minima.

Adam (Adaptive Moment Estimation) combines momentum and adaptive learning rates, maintaining moving averages of both gradients (first moment) and squared gradients (second moment):

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \quad \mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2,$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon},$$

where $\hat{\mathbf{m}}_t, \hat{\mathbf{v}}_t$ are bias-corrected moments, β_1 (e.g., 0.9), β_2 (e.g., 0.999), and ϵ (e.g., 10^{-8}). Adam is robust, converging quickly across diverse tasks, but may overgeneralize compared to SGD for some problems.

Training Dynamics: Each optimizer navigates the loss landscape differently:

- SGD: Slow but precise with momentum, excels in fine-tuning for sharp minima.

- RMSprop: Fast adaptation to gradient scales, effective for non-stationary objectives.
- Adam: Robust and fast, widely used but may require tuning for optimal generalization.

Learning rate η and hyperparameters (e.g., β_1, β lifespan is crucial for all optimizers, with Adam and RMSprop less sensitive to η due to adaptive scaling.

3 Code

3.1 Let's load the necessary packages

[27]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Subset
import torch.nn.functional as F
```

3.2 Checking and initialising GPU

[28]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Using device: {device}')
```

```
Using device: cuda
```

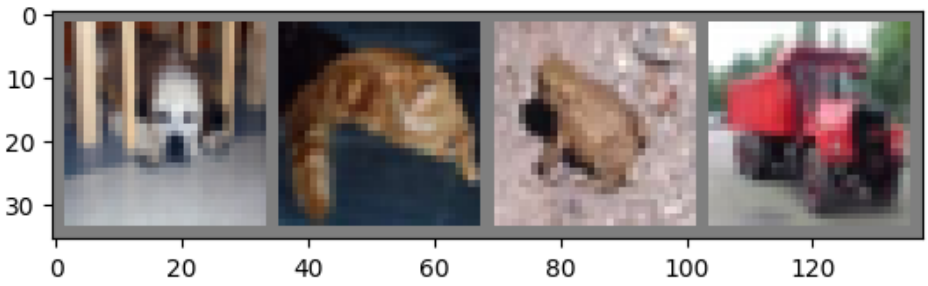
3.3 Let's now define the transformations to the dataset and the CIFAR-10 dataset iteself

[29]:
```
transform = transforms.Compose([
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

3.4 We can visualise some images to see our dataset

[30]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[31]:
```
def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```



```
dog   cat   frog  truck
```

3.5 Now for our classification task we need to define a Convolutional Neural Network (CNN)

[32]:
```python
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

3.6 Let's now define the training process that will be applicable to the model irespective of the optimizer

[33]:
```python
criterion = nn.CrossEntropyLoss()

def train_model(optimizer_name, model, criterion, optimizer, epochs=5):
    model.to(device)
    train_losses = []
    test_accuracies = []
    for epoch in range(epochs):
        running_loss = 0.0
        model.train()
        for images, labels in trainloader:
            images, labels = images.to(device), labels.to(device)
            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 = 0
        total = 0
        with torch.no_grad():
            for images, labels in testloader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        accuracy = 100 * correct / total
        test_accuracies.append(accuracy)

        print(f'[{optimizer_name}] Epoch {epoch+1}, Loss: {avg_loss:.4f},␣
↪Accuracy: {accuracy:.2f}%')
    return (train_losses, test_accuracies)
```

3.7 Now let's specifiy the 3 optimizers to compare

[34]:
```
optimizer_configs = {
    'SGD': lambda params: optim.SGD(params, lr=0.001),
    'Adam': lambda params: optim.Adam(params, lr=0.001),
    'RMSProp': lambda params: optim.RMSprop(params, lr=0.001)
}
```

3.8 Now we can run the model for each optimizer type and see the result

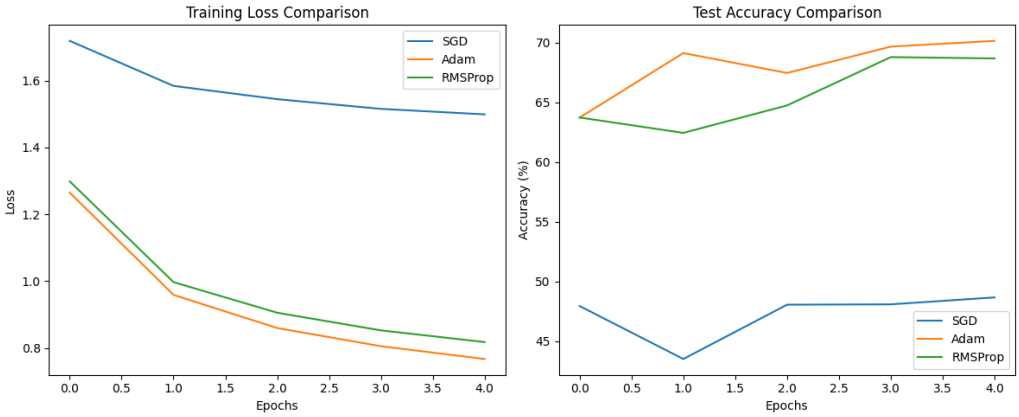
[36]:
```
# Initialize model and save initial state
model = CNN()
initial_state = model.state_dict()

results = {}
for opt_name, opt_class in optimizer_configs.items():
    model = CNN().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = opt_class(model.parameters())
    results[opt_name] = train_model(opt_name, model, criterion, optimizer)
```

```
[SGD] Epoch 1, Loss: 1.7188, Accuracy: 47.93%
[SGD] Epoch 2, Loss: 1.5841, Accuracy: 43.49%
[SGD] Epoch 3, Loss: 1.5444, Accuracy: 48.04%
[SGD] Epoch 4, Loss: 1.5153, Accuracy: 48.07%
```

```
[SGD] Epoch 5, Loss: 1.4988, Accuracy: 48.65%
[Adam] Epoch 1, Loss: 1.2646, Accuracy: 63.72%
[Adam] Epoch 2, Loss: 0.9587, Accuracy: 69.13%
[Adam] Epoch 3, Loss: 0.8594, Accuracy: 67.46%
[Adam] Epoch 4, Loss: 0.8050, Accuracy: 69.67%
[Adam] Epoch 5, Loss: 0.7668, Accuracy: 70.15%
[RMSProp] Epoch 1, Loss: 1.2983, Accuracy: 63.73%
[RMSProp] Epoch 2, Loss: 0.9969, Accuracy: 62.44%
[RMSProp] Epoch 3, Loss: 0.9052, Accuracy: 64.74%
[RMSProp] Epoch 4, Loss: 0.8524, Accuracy: 68.79%
[RMSProp] Epoch 5, Loss: 0.8174, Accuracy: 68.68%
```

3.9 Finally we plot the results to visualise the comparison

[37]:
```python
# Plot results
plt.figure(figsize=(12, 5))

# Plot Losses
plt.subplot(1, 2, 1)
for opt_name in results.keys():
    losses, _ = results[opt_name]
    plt.plot(losses, label=opt_name)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Comparison')
plt.legend()

# Plot Accuracies
plt.subplot(1, 2, 2)
for opt_name in results.keys():
    _, accuracies = results[opt_name]
    plt.plot(accuracies, label=opt_name)
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Test Accuracy Comparison')
plt.legend()

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


Training Loss Comparison
SGD
Adam
RMSProp
Loss
Epochs
Test Accuracy Comparison
Accuracy (%)
Epochs

Lab Work 5
Regularization (L1, L2, Dropout) using PyTorch.

1 Aim: To demonstrate and compare different types of Regularization techniques like L1, L2, Dropout

2 Theory

Regularization techniques—L1, L2, and Dropout—are employed in deep learning to prevent overfitting, ensuring models generalize well to unseen data. The objective is to constrain model complexity or introduce robustness during training, applied here in a PyTorch-based neural network context for tasks like classification or regression. These methods modify the loss function or training process to penalize overly complex models or mitigate reliance on specific parameters.

L1 Regularization (Lasso) adds the sum of absolute parameter values to the loss function:

$$\mathcal{L}_{\text{reg}} = \mathcal{L}(\theta) + \lambda \sum_i |\theta_i|,$$

where $\mathcal{L}(\theta)$ is the original loss (e.g., cross-entropy), θ_i are model parameters, and λ (e.g., 10^{-4}) controls the penalty strength. L1 promotes sparsity, driving some weights to exactly zero, which simplifies models and aids feature selection. However, it can lead to unstable gradients in non-convex settings.

L2 Regularization (Ridge) adds the sum of squared parameter values:

$$\mathcal{L}_{\text{reg}} = \mathcal{L}(\theta) + \lambda \sum_i \theta_i^2.$$

L2 penalizes large weights, encouraging smaller, distributed weights that reduce model sensitivity to individual features. It is computationally stable and widely used, but does not induce sparsity. In PyTorch, L2 is often implemented via weight decay in optimizers like Adam.

Dropout randomly deactivates a fraction p (e.g., 0.2) of neurons during training, effectively sampling different subnetworks:

$$\mathbf{h}_i = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b}) \cdot \mathbf{m}_i, \quad \mathbf{m}_i \sim \text{Bernoulli}(1-p).$$

At inference, all neurons are active, with weights scaled by $1-p$ to maintain expected outputs. Dropout reduces co-adaptation of neurons, promoting robustness and ensemble-like behavior, but increases training time due to stochasticity.

Training Dynamics: Regularization alters optimization:

- L1: Produces sparse models, potentially reducing test error but risking underfitting if λ is too large.
- L2: Smooths the loss landscape, improving generalization with less risk of underfitting.

- Dropout: Increases training loss variance but enhances robustness, especially for deep networks.

The choice of λ or p is critical, often tuned via validation. Combining methods (e.g., L2 + Dropout) can yield complementary benefits.

3 Code

3.1 Let's load the libraries

[43]:
```
import torch
import torch.nn as nn
import torchvision
import torch.optim as optim
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import torch.utils.data as data
```

3.2 Initialise my local GPU

[44]:
```
device = torch.device("cpu")
```

3.3 Initialise the downloaded titanic.csv as dataframe

[45]:
```
df = pd.read_csv("../data/titanic.csv")
```

[46]:
```
df
```

[46]:

	Survived	Pclass	Name	Sex	Age	Siblings/Spouses Aboard	Parents/Children Aboard	Fare
0	0	3	Mr. Owen Harris Braund	male	22.0	1	0	7.2500
1	1	1	Mrs. John Bradley (Florence Briggs Thayer) Cum...	female	38.0	1	0	71.2833
2	1	3	Miss. Laina Heikkinen					
3	1	1	Mrs. Jacques Heath (Lily May Peel) Futrelle					
4	0	3	Mr. William Henry Allen					
..	...	...	...					
882	0	2	Rev. Juozas Montvila					
883	1	1	Miss. Margaret Edith Graham					
884	0	3	Miss. Catherine Helen Johnston					
885	1	1	Mr. Karl Howell Behr					
886	0	3	Mr. Patrick Dooley					

```
2      female  26.0                      0                      0   7.9250
3      female  35.0                      1                      0  53.1000
4        male  35.0                      0                      0   8.0500
..        ...   ...                    ...                    ...      ...
882      male  27.0                      0                      0  13.0000
883    female  19.0                      0                      0  30.0000
884    female   7.0                      1                      2  23.4500
885      male  26.0                      0                      0  30.0000
886      male  32.0                      0                      0   7.7500

[887 rows x 8 columns]
```

3.4 Now we define the features of the dataset, handle the missing values and split the data into dependent and independent variables

[47]:
```
features = ['Pclass', 'Sex', 'Age', 'Siblings/Spouses Aboard', 'Parents/Children␣
 ↪Aboard', 'Fare']
target = 'Survived'

df['Age'].fillna(df['Age'].median(), inplace=True)
df['Sex'] = LabelEncoder().fit_transform(df['Sex'])

X = df[features].values
y = df[target].values

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

```
/var/folders/xc/v33xrpmn5hjfj49419jm6mqw0000gn/T/ipykernel_40658/1406114455.py:4
: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series
through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df['Age'].fillna(df['Age'].median(), inplace=True)
```

[48]:
```
df
```

[48]:
```
   Survived  Pclass                                               Name  Sex  \
0         0       3                            Mr. Owen Harris Braund    1
1         1       1  Mrs. John Bradley (Florence Briggs Thayer) Cum...    0
```

```
2           1       3                                Miss. Laina Heikkinen    0
3           1       1        Mrs. Jacques Heath (Lily May Peel) Futrelle    0
4           0       3                              Mr. William Henry Allen    1
..        ...     ...                                                 ...  ...
882         0       2                               Rev. Juozas Montvila    1
883         1       1                          Miss. Margaret Edith Graham    0
884         0       3                       Miss. Catherine Helen Johnston    0
885         1       1                               Mr. Karl Howell Behr    1
886         0       3                                 Mr. Patrick Dooley    1

      Age  Siblings/Spouses Aboard  Parents/Children Aboard     Fare
0    22.0                         1                        0   7.2500
1    38.0                         1                        0  71.2833
2    26.0                         0                        0   7.9250
3    35.0                         1                        0  53.1000
4    35.0                         0                        0   8.0500
..    ...                       ...                      ...      ...
882  27.0                         0                        0  13.0000
883  19.0                         0                        0  30.0000
884   7.0                         1                        2  23.4500
885  26.0                         0                        0  30.0000
886  32.0                         0                        0   7.7500

[887 rows x 8 columns]
```

[49]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[50]:
```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
```

[51]:
```
batch_size = 32
train_dataset = data.TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = data.TensorDataset(X_test_tensor, y_test_tensor)
train_loader = data.DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
test_loader = data.DataLoader(dataset=test_dataset, batch_size=batch_size,␣
 ↪shuffle=False)
```

3.5 Now that we have processed the data we can define our 4 models for the comparison

[52]:
```
class BaseModel(nn.Module):
    def __init__(self, input_size):
        super(BaseModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 2)

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

class L1Model(BaseModel):
    def __init__(self, input_size):
        super(L1Model, self).__init__(input_size)

    def l1_loss(self, lambda_l1=0.0005):
        l1_reg = sum(torch.norm(param, p=1) for param in self.parameters())
        return lambda_l1 * l1_reg

class L2Model(BaseModel):
    def __init__(self, input_size):
        super(L2Model, self).__init__(input_size)

class DropoutModel(nn.Module):
    def __init__(self, input_size, dropout_rate=0.3):
        super(DropoutModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_rate)
        self.fc2 = nn.Linear(64, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

3.6 Now we can write the general the train_model function to execute training

[53]:
```
def train_model(model, optimizer, lambda_l1=0, num_epochs=30):
    criterion = nn.CrossEntropyLoss()
    history = {'train_loss': [], 'val_loss': [], 'val_acc': []}

    for epoch in range(num_epochs):
        # Training phase
        model.train()
        train_loss = 0
        for features, labels in train_loader:
            features, labels = features.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(features)
            loss = criterion(outputs, labels)
            if isinstance(model, L1Model):  # Add L1 regularization
                loss += model.l1_loss(lambda_l1)

            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        # Validation phase
        model.eval()
        val_loss, correct, total = 0, 0, 0
        with torch.no_grad():
            for features, labels in test_loader:
                features, labels = features.to(device), labels.to(device)
                outputs = model(features)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        # Calculate averages
        train_loss = train_loss / len(train_loader)
        val_loss = val_loss / len(test_loader)
        val_acc = 100 * correct / total

        # Store metrics
        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
```

```
        print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f},␣
↪Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")

    return history
```

[54]:
```
input_size = X_train.shape[1]

# Initialize models
base_model = BaseModel(input_size).to(device)
l1_model = L1Model(input_size).to(device)
l2_model = L2Model(input_size).to(device)
dropout_model = DropoutModel(input_size).to(device)

# Optimizers
base_opt = optim.Adam(base_model.parameters(), lr=0.001)
l1_opt = optim.Adam(l1_model.parameters(), lr=0.001)
l2_opt = optim.Adam(l2_model.parameters(), lr=0.001, weight_decay=0.01)  # L2␣
↪regularization via weight_decay
dropout_opt = optim.Adam(dropout_model.parameters(), lr=0.001)
```

3.7 Let's train each model with its respective regularization technique

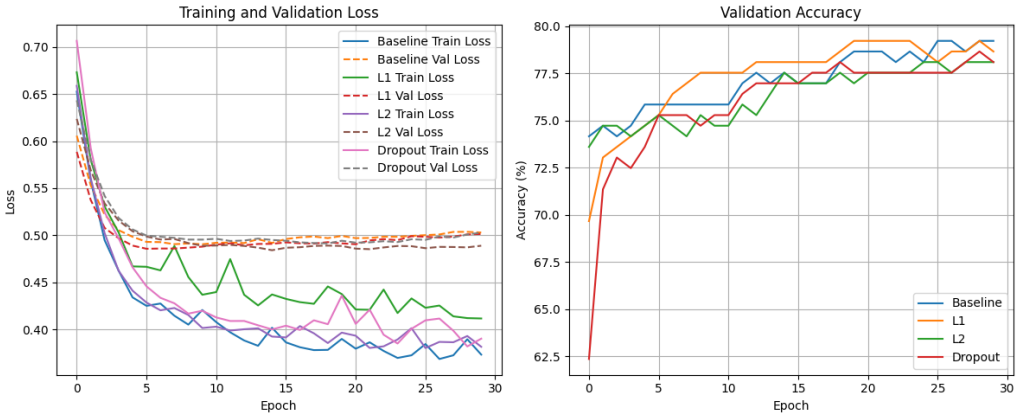
[55]:
```
hist_base = train_model(base_model, base_opt)
hist_l1 = train_model(l1_model, l1_opt, lambda_l1=0.0005)
hist_l2 = train_model(l2_model, l2_opt)
hist_dropout = train_model(dropout_model, dropout_opt)
```

```
Epoch [1/30], Train Loss: 0.6526, Val Loss: 0.6057, Val Acc: 74.16%
Epoch [2/30], Train Loss: 0.5580, Val Loss: 0.5531, Val Acc: 74.72%
Epoch [3/30], Train Loss: 0.4947, Val Loss: 0.5219, Val Acc: 74.16%
Epoch [4/30], Train Loss: 0.4625, Val Loss: 0.5054, Val Acc: 74.72%
Epoch [5/30], Train Loss: 0.4341, Val Loss: 0.4984, Val Acc: 75.84%
Epoch [6/30], Train Loss: 0.4251, Val Loss: 0.4930, Val Acc: 75.84%
Epoch [7/30], Train Loss: 0.4276, Val Loss: 0.4928, Val Acc: 75.84%
Epoch [8/30], Train Loss: 0.4146, Val Loss: 0.4906, Val Acc: 75.84%
Epoch [9/30], Train Loss: 0.4052, Val Loss: 0.4913, Val Acc: 75.84%
Epoch [10/30], Train Loss: 0.4207, Val Loss: 0.4905, Val Acc: 75.84%
Epoch [11/30], Train Loss: 0.4079, Val Loss: 0.4921, Val Acc: 75.84%
Epoch [12/30], Train Loss: 0.3971, Val Loss: 0.4923, Val Acc: 76.97%
Epoch [13/30], Train Loss: 0.3884, Val Loss: 0.4918, Val Acc: 77.53%
Epoch [14/30], Train Loss: 0.3827, Val Loss: 0.4959, Val Acc: 76.97%
Epoch [15/30], Train Loss: 0.4019, Val Loss: 0.4919, Val Acc: 77.53%
Epoch [16/30], Train Loss: 0.3863, Val Loss: 0.4960, Val Acc: 76.97%
Epoch [17/30], Train Loss: 0.3812, Val Loss: 0.4979, Val Acc: 76.97%
Epoch [18/30], Train Loss: 0.3781, Val Loss: 0.4987, Val Acc: 76.97%
Epoch [19/30], Train Loss: 0.3784, Val Loss: 0.4970, Val Acc: 78.09%
Epoch [20/30], Train Loss: 0.3900, Val Loss: 0.4993, Val Acc: 78.65%
```

```
Epoch [27/30], Train Loss: 0.4116, Val Loss: 0.4987, Val Acc: 77.53%
Epoch [28/30], Train Loss: 0.3987, Val Loss: 0.4977, Val Acc: 78.09%
Epoch [29/30], Train Loss: 0.3820, Val Loss: 0.5008, Val Acc: 78.65%
Epoch [30/30], Train Loss: 0.3903, Val Loss: 0.5029, Val Acc: 78.09%
```

3.8 Finally we can visualise our results using plots

[56]:
```python
def plot_results(histories, labels):
    plt.figure(figsize=(12, 5))

    # Loss plot
    plt.subplot(1, 2, 1)
    for hist, label in zip(histories, labels):
        plt.plot(hist['train_loss'], label=f"{label} Train Loss")
        plt.plot(hist['val_loss'], linestyle='--', label=f"{label} Val Loss")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title("Training and Validation Loss")
    plt.grid(True)

    # Accuracy plot
    plt.subplot(1, 2, 2)
    for hist, label in zip(histories, labels):
        plt.plot(hist['val_acc'], label=f"{label}")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.title("Validation Accuracy")
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Plot results
plot_results(
    [hist_base, hist_l1, hist_l2, hist_dropout],
    ["Baseline", "L1", "L2", "Dropout"]
)
```


Training and Validation Loss
Loss
Epoch
Baseline Train Loss
Baseline Val Loss
L1 Train Loss
L1 Val Loss
L2 Train Loss
L2 Val Loss
Dropout Train Loss
Dropout Val Loss
Validation Accuracy
Accuracy (%)
Epoch
Baseline
L1
L2
Dropout

Lab Work 6

Batch Normalization & Data Augmentation in CNN using PyTorch

1 Aim: To observe the improvement in a CNN model after implementing batch normalization layers and data augmentation

2 Theory

Batch Normalization (BN) and Data Augmentation are key techniques in Convolutional Neural Networks (CNNs) to enhance training stability, convergence, and generalization, applied here in a PyTorch context for tasks like image classification. The objective is to normalize activations and enrich the training data to improve model robustness and performance on datasets such as CIFAR-10 or ImageNet.

Batch Normalization normalizes the activations of each layer across a mini-batch to maintain consistent mean and variance. For a layer's input $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$ (channels C, height H, width W), BN computes:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad \mathbf{y} = \gamma\hat{\mathbf{x}} + \beta,$$

where μ_B, σ_B^2 are the mini-batch mean and variance, ϵ (e.g., 10^{-5}) ensures numerical stability, and γ, β are learnable parameters. During inference, running averages of μ, σ^2 are used. BN reduces internal covariate shift, accelerates training, and allows higher learning rates, but adds computational overhead and may be less effective for small batches.

Data Augmentation artificially expands the training dataset by applying random transformations to input images (e.g., rotations, flips, color jitter). For an image $\mathbf{x}$, a transformation T (e.g., random crop, horizontal flip) generates a new sample $T(\mathbf{x})$. This increases dataset diversity, simulating real-world variations:

$$\mathcal{D}_{\text{aug}} = \{(\mathbf{x}_i, y_i), (T_1(\mathbf{x}_i), y_i), \ldots, (T_k(\mathbf{x}_i), y_i)\},$$

where $\mathcal{D}_{\text{aug}}$ is the augmented dataset. Augmentation improves generalization by exposing the model to varied inputs, but excessive transformations may distort task-relevant features.

Training Dynamics: Both techniques enhance CNN optimization:

- BN: Stabilizes gradients, reduces vanishing/exploding gradient issues, and accelerates convergence.
- Data Augmentation: Reduces overfitting by increasing effective dataset size, improving robustness to variations.

In PyTorch, BN is implemented as a layer (e.g., `nn.BatchNorm2d`), and augmentation is applied via `torchvision.transforms`. Hyperparameters like augmentation strength or BN's momentum (e.g., 0.1) require tuning.

3 Code

3.1 Loading the necessary libraries

[6]:
```
import torch
import torch.nn as nn
import torchvision
import torch.optim as optim
import torchvision.transforms as transforms
```

3.2 Loading my local GPU

[3]:
```
device = torch.device("mps")
```

3.3 Loading the dataset for the baseline and the imporved model

[7]:
```
transform_train_aug = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_train_base = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Loading the CIFAR-10 dataset
trainset_aug = torchvision.datasets.CIFAR10(root="./data", train=True,␣
 ↪download=True, transform=transform_train_aug)
trainloader_aug = torch.utils.data.DataLoader(trainset_aug, batch_size=64,␣
 ↪shuffle=True, num_workers=2)

trainset_base = torchvision.datasets.CIFAR10(root="./data", train=True,␣
 ↪download=True, transform=transform_train_base)
trainloader_base = torch.utils.data.DataLoader(trainset_base, batch_size=64,␣
 ↪shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False,␣
↪num_workers=2)
```

3.4 Defining the baseline and the improved models

[11]:
```
class BaselineCNN(nn.Module):
    def __init__(self):
        super(BaselineCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8) # Flatten
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class BatchNormCNN(nn.Module):
    def __init__(self):
        super(BatchNormCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 8 * 8, 128)
        self.bn_fc1 = nn.BatchNorm1d(128)      # Batch norm for fully connected␣
↪layer
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.pool(self.relu(self.bn2(self.conv2(x))))
        x = x.view(-1, 64 * 8 * 8)
        x = self.relu(self.bn_fc1(self.fc1(x)))
        x = self.fc2(x)
        return x
```

3.5 Defining the train_model function

[17]:
```
def train_model(model, trainloader, testloader, criterion, optimizer,␣
↪num_epochs=10):
    model.to(device)
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_loss = running_loss / len(trainloader)
        train_acc = 100 * correct / total
        print(f'Epoch {epoch+1}, Train Loss: {train_loss:.3f}, Train Acc:␣
↪{train_acc:.2f}%')

        # Evaluate on test set
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for data in testloader:
                images, labels = data
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        test_acc = 100 * correct / total
        print(f'Test Acc: {test_acc:.2f}%')
```

3.6 Actual training step

[18]:
```
baseline_model = BaselineCNN()
bn_model = BatchNormCNN()
criterion = nn.CrossEntropyLoss()
optimizer_base = optim.SGD(baseline_model.parameters(), lr=0.01, momentum=0.9)
optimizer_bn = optim.SGD(bn_model.parameters(), lr=0.01, momentum=0.9)

# Baseline model training
print("Training Baseline Model (No BatchNorm, No Augmentation)")
train_model(baseline_model, trainloader_base, testloader, criterion,␣
 ↪optimizer_base)

print("\nTraining Enhanced Model (With BatchNorm and Augmentation)")
train_model(bn_model, trainloader_aug, testloader, criterion, optimizer_bn)
```

```
Training Baseline Model (No BatchNorm, No Augmentation)
Epoch 1, Train Loss: 1.530, Train Acc: 44.43%
Test Acc: 58.57%
Epoch 2, Train Loss: 1.075, Train Acc: 61.75%
Test Acc: 63.68%
Epoch 3, Train Loss: 0.891, Train Acc: 68.30%
Test Acc: 69.25%
Epoch 4, Train Loss: 0.752, Train Acc: 73.45%
Test Acc: 71.56%
Epoch 5, Train Loss: 0.641, Train Acc: 77.42%
Test Acc: 71.63%
Epoch 6, Train Loss: 0.537, Train Acc: 81.14%
Test Acc: 71.50%
Epoch 7, Train Loss: 0.428, Train Acc: 85.15%
Test Acc: 72.94%
Epoch 8, Train Loss: 0.331, Train Acc: 88.41%
Test Acc: 72.50%
Epoch 9, Train Loss: 0.263, Train Acc: 90.74%
Test Acc: 72.60%
Epoch 10, Train Loss: 0.187, Train Acc: 93.45%
Test Acc: 72.87%

Training Enhanced Model (With BatchNorm and Augmentation)
Epoch 1, Train Loss: 1.373, Train Acc: 50.54%
Test Acc: 61.97%
Epoch 2, Train Loss: 1.083, Train Acc: 61.33%
Test Acc: 67.06%
Epoch 3, Train Loss: 0.974, Train Acc: 65.33%
Test Acc: 70.02%
Epoch 4, Train Loss: 0.905, Train Acc: 67.87%
Test Acc: 70.24%
Epoch 5, Train Loss: 0.859, Train Acc: 69.70%
```

```
Test Acc: 72.75%
Epoch 6, Train Loss: 0.814, Train Acc: 71.25%
Test Acc: 74.03%
Epoch 7, Train Loss: 0.784, Train Acc: 72.43%
Test Acc: 75.02%
Epoch 8, Train Loss: 0.757, Train Acc: 73.34%
Test Acc: 75.48%
Epoch 9, Train Loss: 0.731, Train Acc: 74.41%
Test Acc: 76.12%
Epoch 10, Train Loss: 0.715, Train Acc: 74.75%
Test Acc: 76.92%
```

The improved model has lower training accuracy because data augmentation and BatchNorm make it harder to overfit the training data, acting as regularizers that increase robustness and prevent memorization. This results in a higher test accuracy, demonstrating better generalization to unseen data. The Baseline Model, lacking these mechanisms, overfits the training set (high training accuracy, low test accuracy gap), making it less effective on the test set.

Lab Work 7
Implementing Early Stopping & Checkpointing using PyTorch.

1 Aim: To implement Early Stopping and Checkpointing using PyTorch

2 Theory

Early Stopping and Checkpointing are techniques used in deep learning to optimize training efficiency and model performance, applied here in a PyTorch context for tasks like classification or regression. The objective is to prevent overfitting and ensure the retention of the best model parameters during training, balancing computational resources and generalization.

Early Stopping monitors a validation metric (e.g., validation loss or accuracy) during training and halts the process if the metric ceases to improve after a specified number of epochs (patience). Formally, for a validation loss $\mathcal{L}_{\text{val}}(\theta_t)$ at epoch t, training stops if:

$$\mathcal{L}_{\text{val}}(\theta_{t+k}) \geq \mathcal{L}_{\text{val}}(\theta_t) \quad \text{for} \quad k = 1, \ldots, p,$$

where p is the patience parameter (e.g., 5). Early Stopping prevents overfitting by selecting the model parameters θ_t at the epoch with the best validation performance, avoiding unnecessary computation. However, overly aggressive stopping (small p) may lead to underfitting, while large p delays training termination.

Checkpointing saves the model's parameters θ_t and optimizer state at regular intervals or when a validation metric improves. The checkpoint includes:

$$\text{Checkpoint} = \{\theta\_t, \text{optimizer\_state}, \text{epoch}, \mathcal{L}\_\text{val}\},$$

allowing training to resume from the saved state or revert to the best model. In PyTorch, checkpoints are typically saved as `.pth` files. Checkpointing ensures robustness against interruptions (e.g., hardware failures) and enables selection of the optimal model without retraining, but increases storage requirements.

Training Dynamics: Both techniques enhance training:

- Early Stopping: Reduces overfitting by stopping when validation performance plateaus, preserving generalization.
- Checkpointing: Provides flexibility to recover or select the best model, improving reliability and efficiency.

In PyTorch, Early Stopping is implemented by tracking validation metrics, while Checkpointing uses `torch.save` and `torch.load`. Hyperparameters like patience and checkpoint frequency require tuning based on dataset size and task complexity.

3 Code

3.1 Loading the libraries

[12]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
import os
```

[13]:
```
device = torch.device("mps")
```

3.2 Doing all the pre-processing and loading the datasets

[15]:
```
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

dataset_path = "./data/PETImages"  # Path to dataset containing Cat/ and Dog/
dataset = torchvision.datasets.ImageFolder(root=dataset_path,␣
 ↪transform=transform)

train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

print(f"Class Mapping: {dataset.class_to_idx}")
```

```
Class Mapping: {'Cat': 0, 'Dog': 1}
```

3.3 Implementing the main CNN model

[16]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 64 * 64, 2)  # Adjust based on input size

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
```

```
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        return x
```

[17]:
```
model = CNN().to(device)
```

[18]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3.4 Implementing Early Stopping Class

[19]:
```
class EarlyStopping:
    def __init__(self, patience=5, checkpoint_path="best_model.pth"):
        self.patience = patience
        self.counter = 0
        self.best_loss = float("inf")
        self.checkpoint_path = checkpoint_path

    def __call__(self, val_loss, model):
        if val_loss < self.best_loss:
            self.best_loss = val_loss
            self.counter = 0
            torch.save(model.state_dict(), self.checkpoint_path)
            print(f"Checkpoint saved at validation loss: {val_loss:.4f}")
        else:
            self.counter += 1
            print(f"Early stopping counter: {self.counter}/{self.patience}")
            if self.counter >= self.patience:
                return True
        return False
```

[20]:
```
early_stopping = EarlyStopping()
```

3.5 Training the model while implementing early stopping if validation loss gets too much and saving the best model using checkpoint

[21]:
```
for epoch in range(20):  # Max epochs
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
        running_loss += loss.item()

    # Validation
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()

    val_loss /= len(val_loader)
    print(f"Epoch {epoch+1}, Validation Loss: {val_loss:.4f}")

    if early_stopping(val_loss, model):
        print("Early stopping triggered.")
        break
```

```
Epoch 1, Validation Loss: 0.5534
Checkpoint saved at validation loss: 0.5534
Epoch 2, Validation Loss: 0.5651
Early stopping counter: 1/5
Epoch 3, Validation Loss: 0.5434
Checkpoint saved at validation loss: 0.5434
Epoch 4, Validation Loss: 0.5869
Early stopping counter: 1/5
Epoch 5, Validation Loss: 0.6775
Early stopping counter: 2/5
Epoch 6, Validation Loss: 0.7045
Early stopping counter: 3/5
Epoch 7, Validation Loss: 0.7518
Early stopping counter: 4/5
Epoch 8, Validation Loss: 0.7873
Early stopping counter: 5/5
Early stopping triggered.
```

[22]:
```
model.load_state_dict(torch.load("best_model.pth"))
print("Best model loaded.")
```

```
Best model loaded.
```

[25]:
```
print(model)

total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```python
print(f"Total Parameters: {total_params}")
print(f"Trainable Parameters: {trainable_params}")
```

```
CNN(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=131072, out_features=2, bias=True)
)
Total Parameters: 263042
Trainable Parameters: 263042
```

Lab Work 8
Handling Imbalanced Data using PyTorch

1 Aim: To demonstrate class weighting, undersampling, oversampling and SMOTE

2 Theory

Handling imbalanced data is critical in machine learning when class distributions are skewed, such as in fraud detection or medical diagnosis, where the minority class (e.g., positive cases) is underrepresented. The objective is to mitigate bias toward the majority class, improving model performance on the minority class. The Synthetic Minority Oversampling Technique (SMOTE) is a popular method to address this by generating synthetic samples for the minority class, enhancing classifier generalization.

SMOTE operates by oversampling the minority class in the feature space. For a minority sample $\mathbf{x}_i \in \mathbb{R}^d$ (where d is the feature dimension), SMOTE selects k nearest neighbors (e.g., $k = 5$) from the same class using Euclidean distance. A synthetic sample $\mathbf{x}_{\text{new}}$ is generated by interpolating between $\mathbf{x}_i$ and a randomly chosen neighbor $\mathbf{x}_n$:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_i + \lambda(\mathbf{x}_n - \mathbf{x}_i), \quad \lambda \sim \text{Uniform}(0, 1).$$

This process is repeated until the minority class is sufficiently balanced with the majority class (e.g., equal class sizes). SMOTE creates diverse, realistic samples within the minority class's feature distribution, unlike simple oversampling, which duplicates existing samples and risks overfitting.

Advantages and Trade-offs: SMOTE improves minority class recall and F1-score by reducing classifier bias, particularly for algorithms like SVMs or neural networks. It preserves the original data's structure while increasing dataset size, aiding generalization. However, SMOTE may introduce noise if synthetic samples cross class boundaries, especially in complex datasets, and increases computational cost due to larger training sets. Variants like Borderline-SMOTE focus on boundary samples to mitigate this.

Training Dynamics: SMOTE alters the training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ to $\mathcal{D}_{\text{SMOTE}}$, balancing class distributions. Classifiers trained on $\mathcal{D}_{\text{SMOTE}}$ prioritize minority class patterns, improving performance metrics like precision, recall, and area under the ROC curve (AUC). Hyperparameters, such as k or the oversampling ratio, require tuning to avoid overgeneralization or noise.

3 Code

[7]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, WeightedRandomSampler, TensorDataset
import pandas as pd
import numpy as np
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report, roc_curve,␣
↪auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

3.1 Loading the dataset

[2]:
```
df = pd.read_csv("./data/creditcard.csv")

X = df.drop(columns=["Class"]).values  # Features
y = df["Class"].values  # Labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪stratify=y, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

3.2 Create PyTorch Tensors for compatibility

[4]:
```
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
```

3.3 In order to solve the class imbalance we first create a Sampler

[5]:
```
class_counts = np.bincount(y_train)
class_weights = 1.0 / torch.tensor(class_counts, dtype=torch.float32)
sample_weights = class_weights[y_train]
sampler = WeightedRandomSampler(weights=sample_weights,␣
↪num_samples=len(sample_weights), replacement=True)
```

[8]:
```
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, sampler=sampler)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

3.4 Now let's define the NN model

[9]:
```
class FraudDetector(nn.Module):
    def __init__(self):
        super(FraudDetector, self).__init__()
        self.fc1 = nn.Linear(X_train.shape[1], 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 2)
    def forward(self, x):
```

```
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

[12]:
```
model = FraudDetector()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

[14]:
```
for epoch in range(10):
    model.train()
    epoch_loss = 0
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    print(f"Epoch {epoch+1}/10, Loss: {epoch_loss/len(train_loader):.4f}")
```

```
Epoch 1/10, Loss: 0.0797
Epoch 2/10, Loss: 0.0223
Epoch 3/10, Loss: 0.0144
Epoch 4/10, Loss: 0.0115
Epoch 5/10, Loss: 0.0101
Epoch 6/10, Loss: 0.0090
Epoch 7/10, Loss: 0.0080
Epoch 8/10, Loss: 0.0075
Epoch 9/10, Loss: 0.0069
Epoch 10/10, Loss: 0.0058
```

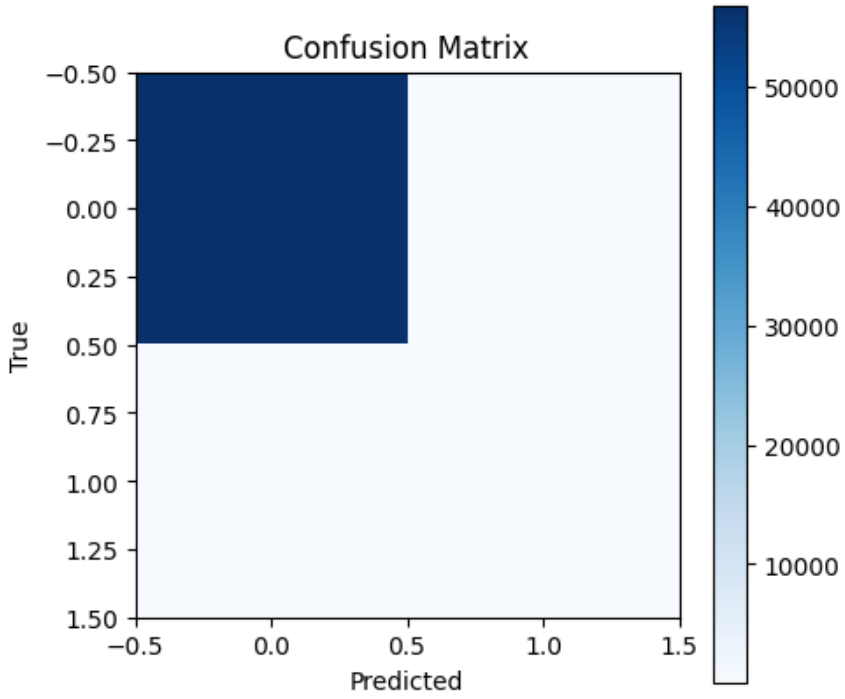
[16]:
```
model.eval()
y_pred, y_true, y_probs = [], [], []
with torch.no_grad():
    for X_batch, y_batch in test_loader:
        outputs = model(X_batch)
        probs = torch.softmax(outputs, dim=1)[:, 1]
        preds = torch.argmax(outputs, dim=1)
        y_pred.extend(preds.cpu().numpy())
        y_true.extend(y_batch.cpu().numpy())
        y_probs.extend(probs.cpu().numpy())
```

3.5 Let's compute the metrics and plot the Confusion Matrix

[17]:
```
print(classification_report(y_true, y_pred))
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5,5))
plt.imshow(cm, cmap='Blues', interpolation='nearest')
```

```
plt.colorbar()
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

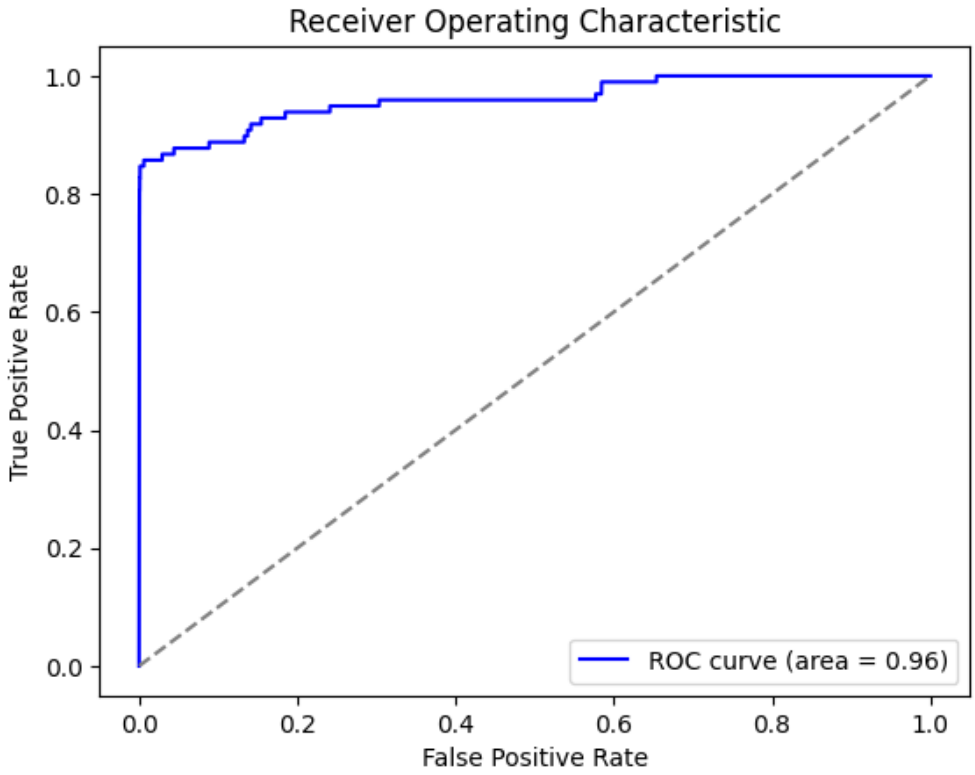
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.41      0.85      0.56        98

    accuracy                           1.00     56962
   macro avg       0.71      0.92      0.78     56962
weighted avg       1.00      1.00      1.00     56962
```



3.6 Finally we plot the ROC curve to see model performance

[18]:
```
fpr, tpr, _ = roc_curve(y_true, y_probs)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (area = {roc_auc:.2f})')
```

```
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend()
plt.show()
```



Lab Work 9
Implementing basics of CNN using PyTorch

1 Aim: To implement the basics of CNN using PyTorch

2 Theory

Convolutional Neural Networks (CNNs) are specialized neural networks designed for processing structured grid-like data, such as images, commonly used in tasks like image classification, object detection, and facial recognition. The objective is to automatically learn hierarchical feature representations from raw input data, leveraging spatial locality to achieve high performance with fewer parameters compared to fully connected networks.

A CNN consists of several key components:

- **Convolutional Layers**: Apply learnable filters to input data (e.g., an image $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$) to extract features like edges or textures. A filter $\mathbf{W} \in \mathbb{R}^{k \times k \times C}$ slides over $\mathbf{X}$, computing feature maps via:
$$\mathbf{Z}_{i,j} = \sum_{m,n,c} \mathbf{W}_{m,n,c} \mathbf{X}_{i+m,j+n,c} + b,$$
where b is a bias, and strides/padding control output size. ReLU activations (e.g., $\max(0, \mathbf{Z})$) introduce non-linearity.
- **Pooling Layers**: Reduce spatial dimensions (e.g., max-pooling) to downsample feature maps, preserving important features while reducing computational load:
$$\mathbf{P}_{i,j} = \max_{m,n \in \text{region}} \mathbf{Z}_{i \cdot s+m, j \cdot s+n},$$
where s is the stride. This enhances translation invariance.
- **Fully Connected Layers**: Aggregate learned features for tasks like classification, mapping high-level representations to output classes (e.g., 10 for digit recognition):
$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{h} + \mathbf{b}).$$

Advantages: CNNs exploit spatial hierarchies, reducing parameters through weight sharing and local connectivity, making them efficient for large inputs like images. They learn hierarchical features (e.g., edges in early layers, objects in deeper layers), outperforming traditional methods in vision tasks. However, CNNs require large datasets and computational resources, and their performance depends on careful hyperparameter tuning (e.g., filter size, number of layers).

Training Dynamics: CNNs are trained to minimize a loss function, such as cross-entropy for classification:
$$\mathcal{L} = -\sum_{i=1}^{C} t_i \log(y_i),$$
where t_i is the true label and y_i is the predicted probability. Training uses backpropagation with optimizers like Adam or SGD, often incorporating regularization (e.g., dropout) and data augmentation to prevent overfitting. The hierarchical structure enables CNNs to generalize well when trained on diverse datasets.

3 Code

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

[8]:
```
device = torch.device("mps")
```

[3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,␣
↪transform=transform)
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,␣
↪transform=transform)

train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)
```

[10]:
```
import matplotlib.pyplot as plt
import numpy as np

# Get a batch of images
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Display first 5 images
fig, axes = plt.subplots(1, 5, figsize=(10, 2))
for i in range(5):
    ax = axes[i]
    ax.imshow(images[i][0].numpy(), cmap='gray')  # MNIST images are grayscale
    ax.axis('off')
plt.show()
```

[6]:
```
class ClassicCNN(nn.Module):
    def __init__(self):
        super(ClassicCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3,␣
↪padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,␣
↪padding=1)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)  # Flatten before FC layers
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = ClassicCNN()
```

[11]:
```
criterion = nn.CrossEntropyLoss()  # For multi-class classification
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

[12]:
```
num_epochs = 10
model.to(device)

for epoch in range(num_epochs):
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader):.
↪4f}")

print("Training complete!")
```

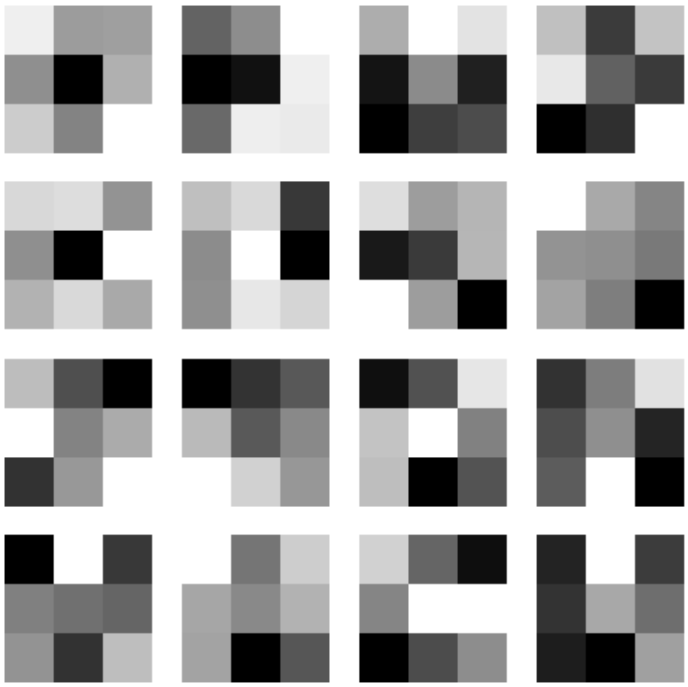
```
Epoch 1/10, Loss: 0.1572
Epoch 2/10, Loss: 0.0715
```

```
Epoch 3/10, Loss: 0.0677
Epoch 4/10, Loss: 0.0637
Epoch 5/10, Loss: 0.0640
Epoch 6/10, Loss: 0.0568
Epoch 7/10, Loss: 0.0639
Epoch 8/10, Loss: 0.0554
Epoch 9/10, Loss: 0.0581
Epoch 10/10, Loss: 0.0537
Training complete!
```

[14]:
```python
correct = 0
total = 0

with torch.no_grad():  # Disable gradient computation
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)  # Get class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Test Accuracy: 98.02%
```

[16]:
```python
def visualize_filters(layer):
    filters = layer.weight.data
    fig, axes = plt.subplots(4, 4, figsize=(5,5))
    for i, ax in enumerate(axes.flat):
        if i < filters.shape[0]:
            ax.imshow(filters[i, 0].cpu().numpy(), cmap="gray")
            ax.axis('off')
    plt.show()

visualize_filters(model.conv1)
```



Lab Work 10
Implementing Advanced CNN Architectures (VGG, ResNet)

1 Aim : To compare the performance of pre-trained image architecture with a simple CNN

2 Theory

Advanced Convolutional Neural Network (CNN) architectures, such as VGG and ResNet, are designed to achieve superior performance in image-related tasks like classification and object detection on datasets like ImageNet. The objective is to leverage deep architectures with innovative designs to extract complex hierarchical features while addressing challenges like vanishing gradients and computational efficiency.

VGG (Visual Geometry Group) employs a deep, uniform architecture with small 3×3 convolutional filters stacked in multiple layers (e.g., VGG16 with 16 layers). For an input image $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, VGG applies:

$$\mathbf{Z}_l = \text{ReLU}(\text{Conv}_{3\times3}(\mathbf{Z}_{l-1}) + \mathbf{b}_l),$$

followed by max-pooling to reduce spatial dimensions. Fully connected layers produce class probabilities:

$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{h} + \mathbf{b}).$$

VGG's uniformity simplifies design, and small filters increase depth while maintaining receptive fields, capturing intricate features. However, its high parameter count (e.g., 138M for VGG16) and computational cost pose challenges for resource-constrained settings.

ResNet (Residual Network) introduces residual connections to enable very deep networks (e.g., ResNet50 with 50 layers). Each residual block computes:

$$\mathbf{Z}_{l+2} = \mathbf{Z}_l + \text{ReLU}(\text{Conv}(\text{BN}(\text{Conv}(\text{BN}(\mathbf{Z}_l))))),$$

where $\mathbf{Z}_l + \cdot$ is a skip connection, and BN denotes Batch Normalization. Skip connections mitigate vanishing gradients by allowing gradients to flow directly, enabling training of networks with hundreds of layers. ResNet's efficiency and performance make it ideal for complex tasks, though it requires careful initialization and tuning.

Advantages and Trade-offs: VGG excels in feature extraction due to its depth but is computationally intensive and memory-heavy. ResNet supports deeper networks with better accuracy and faster convergence, but its complexity increases design and debugging efforts. Both outperform shallow CNNs by learning hierarchical features, from edges to objects.

Training Dynamics: VGG and ResNet minimize a loss function, typically cross-entropy for classification:

$$\mathcal{L} = -\sum_{i=1}^{C} t_i \log(y_i),$$

using optimizers like SGD with momentum or Adam. Techniques like Batch Normalization (in ResNet), dropout (in VGG), and data augmentation enhance generalization. Training on large datasets like ImageNet requires significant computational resources and hyperparameter tuning (e.g., learning rate, batch size).

3 Code

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
from torchvision import models
from torch.optim.lr_scheduler import StepLR
from tqdm.notebook import tqdm
```

[2]:
```
torch.manual_seed(42)
np.random.seed(42)
```

[3]:
```
device = torch.device("cuda:0")
print(f"Using device: {device}")
```

```
Using device: cuda:0
```

3.1 Loading the CIFAR-10 dataset

[4]:
```
batch_size = 128
mean = (0.4914, 0.4822, 0.4465)
std = (0.2470, 0.2435, 0.2616)

# Data transformations with data augmentation for training
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

```
# Load datasets
train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)

test_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)

# Create data loaders
train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)

test_loader = DataLoader(
    test_dataset, batch_size=batch_size, shuffle=False, num_workers=2)
```

```
100% 170M/170M [00:05<00:00, 29.2MB/s]
```

3.2 Let's first define the simple CNN architecture

[5]:
```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)

        x = F.relu(self.conv2(x))
        x = self.pool(x)

        x = F.relu(self.conv3(x))
        x = self.pool(x)

        x = x.view(-1, 128 * 4 * 4)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

[6]:
```python
def train_model(model, trainloader, criterion, optimizer, num_epochs=10):
    model.to(device)
    start_time = time.time()

    train_losses = []
    train_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(trainloader)
        epoch_acc = 100 * correct / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        print(f'Epoch {epoch+1}, Loss: {epoch_loss:.3f}, Accuracy: {epoch_acc:.
↪2f}%')

    training_time = time.time() - start_time
    return training_time, train_losses, train_accuracies
```

[7]:
```python
def evaluate_model(model, testloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
```

```
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy
```

3.3 Now let's initialize all the models

[8]:
```
simple_cnn = SimpleCNN()
criterion = nn.CrossEntropyLoss()
optimizer_simple = optim.SGD(simple_cnn.parameters(), lr=0.001, momentum=0.9)
```

[9]:
```
# VGG16
vgg16 = torchvision.models.vgg16(pretrained=False, num_classes=10)
optimizer_vgg = optim.SGD(vgg16.parameters(), lr=0.001, momentum=0.9)
```

```
/usr/local/lib/python3.11/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/python3.11/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=None`.
  warnings.warn(msg)
```

[10]:
```
resnet18 = torchvision.models.resnet18(pretrained=False, num_classes=10)
optimizer_resnet = optim.SGD(resnet18.parameters(), lr=0.001, momentum=0.9)
```

4 Now we train the models and then we will compare performance

[11]:
```
print("Training Simple CNN...")
simple_time, simple_losses, simple_accs = train_model(simple_cnn, train_loader,␣
 ↪criterion, optimizer_simple)

print("\nTraining VGG16...")
vgg_time, vgg_losses, vgg_accs = train_model(vgg16, train_loader, criterion,␣
 ↪optimizer_vgg)

print("\nTraining ResNet18...")
resnet_time, resnet_losses, resnet_accs = train_model(resnet18, train_loader,␣
 ↪criterion, optimizer_resnet)
```

```
Training Simple CNN...
Epoch 1, Loss: 2.279, Accuracy: 13.91%
Epoch 2, Loss: 2.059, Accuracy: 25.29%
Epoch 3, Loss: 1.907, Accuracy: 31.08%
```

```
Epoch 4, Loss: 1.774, Accuracy: 35.32%
Epoch 5, Loss: 1.657, Accuracy: 39.45%
Epoch 6, Loss: 1.583, Accuracy: 42.14%
Epoch 7, Loss: 1.525, Accuracy: 44.18%
Epoch 8, Loss: 1.485, Accuracy: 45.85%
Epoch 9, Loss: 1.447, Accuracy: 47.25%
Epoch 10, Loss: 1.411, Accuracy: 48.85%

Training VGG16...
Epoch 1, Loss: 2.286, Accuracy: 11.11%
Epoch 2, Loss: 2.012, Accuracy: 23.34%
Epoch 3, Loss: 1.796, Accuracy: 31.27%
Epoch 4, Loss: 1.656, Accuracy: 37.27%
Epoch 5, Loss: 1.543, Accuracy: 42.00%
Epoch 6, Loss: 1.428, Accuracy: 47.15%
Epoch 7, Loss: 1.328, Accuracy: 51.22%
Epoch 8, Loss: 1.236, Accuracy: 54.97%
Epoch 9, Loss: 1.155, Accuracy: 58.15%
Epoch 10, Loss: 1.080, Accuracy: 60.90%

Training ResNet18...
Epoch 1, Loss: 1.827, Accuracy: 32.93%
Epoch 2, Loss: 1.531, Accuracy: 43.87%
Epoch 3, Loss: 1.411, Accuracy: 48.74%
Epoch 4, Loss: 1.322, Accuracy: 52.02%
Epoch 5, Loss: 1.239, Accuracy: 55.27%
Epoch 6, Loss: 1.181, Accuracy: 57.71%
Epoch 7, Loss: 1.129, Accuracy: 59.28%
Epoch 8, Loss: 1.080, Accuracy: 61.18%
Epoch 9, Loss: 1.045, Accuracy: 62.57%
Epoch 10, Loss: 1.009, Accuracy: 63.91%
```

[15]:
```
simple_test_acc = evaluate_model(simple_cnn, test_loader)
vgg_test_acc = evaluate_model(vgg16, test_loader)
resnet_test_acc = evaluate_model(resnet18, test_loader)
```

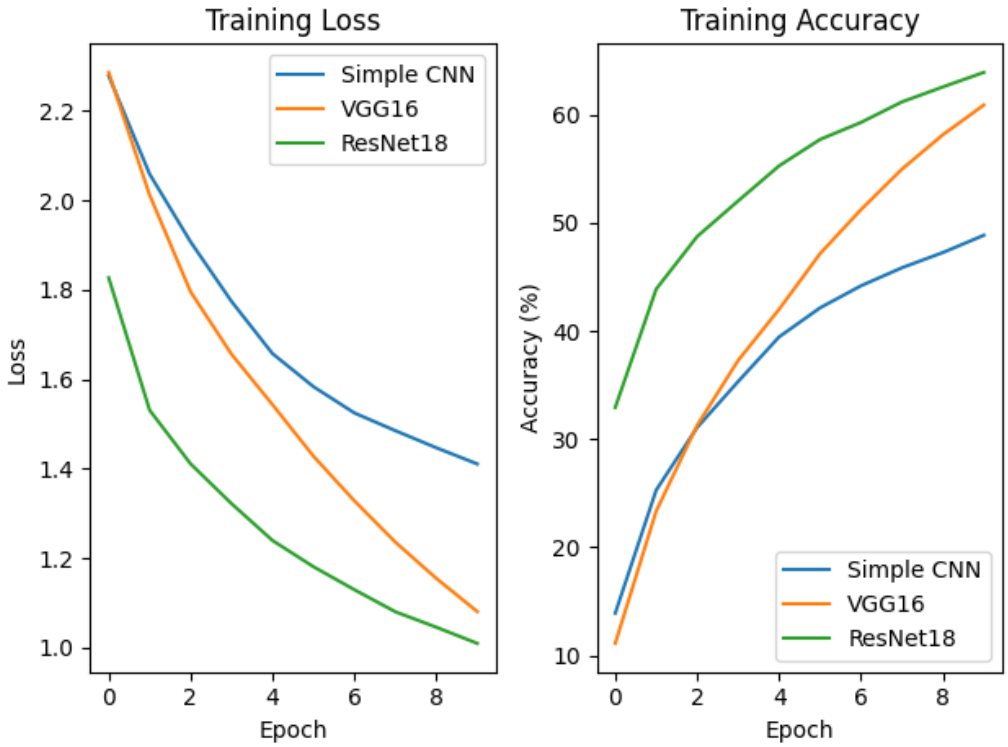
[12]:
```
plt.figure(figsize=(12, 4))
```

[12]:
```
<Figure size 1200x400 with 0 Axes>
```

```
<Figure size 1200x400 with 0 Axes>
```

[13]:
```
plt.subplot(1, 2, 1)
plt.plot(simple_losses, label='Simple CNN')
plt.plot(vgg_losses, label='VGG16')
plt.plot(resnet_losses, label='ResNet18')
plt.title('Training Loss')
plt.xlabel('Epoch')
```

```
plt.ylabel('Loss')
plt.legend()

# Training Accuracy
plt.subplot(1, 2, 2)
plt.plot(simple_accs, label='Simple CNN')
plt.plot(vgg_accs, label='VGG16')
plt.plot(resnet_accs, label='ResNet18')
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

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



[16]:
```
print("\nResults Summary:")
print(f"Simple CNN - Training Time: {simple_time:.2f}s, Test Accuracy:␣
↪{simple_test_acc:.2f}%")
```

```
print(f"VGG16 - Training Time: {vgg_time:.2f}s, Test Accuracy: {vgg_test_acc:.
↪2f}%")
print(f"ResNet18 - Training Time: {resnet_time:.2f}s, Test Accuracy:␣
↪{resnet_test_acc:.2f}%")
```

```
Results Summary:
Simple CNN - Training Time: 196.54s, Test Accuracy: 52.01%
VGG16 - Training Time: 360.30s, Test Accuracy: 63.73%
ResNet18 - Training Time: 201.19s, Test Accuracy: 64.74%
```

5 Architectural Discussion:

1. Simple CNN: - Basic architecture with 2 conv layers - Fast training but limited feature extraction capability - Prone to underfitting on complex datasets

2. VGG16: - Deep architecture with 16 layers - Uses small 3x3 filters stacked together - Increased depth improves feature extraction - More parameters lead to longer training time - Can suffer from vanishing gradients

3. ResNet18: - Uses skip connections to address vanishing gradient problem - Allows training of deeper networks - Residual learning: learns differences rather than direct mapping - Better gradient flow during backpropagation - Balances depth and training efficiency

Lab Work 11
Visualizing Filters and Feature Maps

1 Aim: To visualise filters and feature maps in a CNN architectures

2 Theory

Filters and feature maps are fundamental components of Convolutional Neural Networks (CNNs), designed to process grid-like data, such as images, for tasks like image classification, object detection, and segmentation. The objective is to extract hierarchical, spatially relevant features from raw inputs, enabling CNNs to learn patterns like edges, textures, or objects with high efficiency and robustness.

Filters (or kernels) are small, learnable matrices (e.g., $\mathbf{W} \in \mathbb{R}^{k \times k \times C}$, where k is the kernel size, typically 3 or 5, and C is the input channel count) applied to an input image or feature map $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$. The convolution operation slides the filter over $\mathbf{X}$, computing a weighted sum at each position:

$$\mathbf{Z}_{i,j,c'} = \sum_{m,n,c} \mathbf{W}_{m,n,c,c'} \mathbf{X}_{i+m,j+n,c} + b_{c'},$$

where $b_{c'}$ is a bias, and c' indexes the output channel. Filters detect specific patterns (e.g., edges, corners) by learning weights during training. Multiple filters (e.g., 64 per layer) produce diverse feature representations, with deeper layers capturing more complex patterns.

Feature Maps are the outputs of convolution, representing activations for each filter: $\mathbf{Z} \in \mathbb{R}^{H' \times W' \times C'}$, where H', W' depend on input size, stride, and padding, and C' is the number of filters. A non-linear activation (e.g., ReLU, $\max(0, \mathbf{Z})$) is typically applied to introduce sparsity and enhance feature detection:

$$\mathbf{A} = \text{ReLU}(\mathbf{Z}).$$

Feature maps encode spatial patterns detected by filters, with early layers capturing low-level features (e.g., edges) and deeper layers capturing high-level structures (e.g., object parts). Pooling layers (e.g., max-pooling) often follow to reduce feature map dimensions, enhancing computational efficiency and translation invariance.

Roles and Significance: Filters act as feature detectors, learning task-specific patterns through backpropagation. Feature maps provide a spatial representation of these detections, enabling hierarchical feature learning. Together, they reduce the need for manual feature engineering, making CNNs highly effective for vision tasks. However, the choice of filter size, number, and stride impacts model capacity and computational cost, requiring careful design.

Training Dynamics: Filters are optimized to minimize a loss function (e.g., cross-entropy for classification) via gradient-based methods. Feature maps evolve during training, reflecting increasingly abstract representations. Regularization (e.g., dropout) and normalization (e.g., BatchNorm) stabilize feature map distributions, improving generalization.

3 Code

3.1 Loading the libraries

[1]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

3.2 Let's load the CIFAR-10 dataset

[3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset = torchvision.datasets.CIFAR10(root='./dl_lab_datasets/', train=True,␣
 ↪download=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)

classes = trainset.classes
```

3.3 We define a simple CNN model to visualise the filters

[4]:
```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)   # Low-level
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)  # High-level
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

[5]:
```
net = SimpleCNN()
```

```
# Optional: Load trained weights
# net.load_state_dict(torch.load("model.pth"))
net.eval()
```

[5]:
```
SimpleCNN(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

[6]:
```
def imshow(img, title=None):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.figure(figsize=(2,2))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    if title:
        plt.title(title)
    plt.axis('off')
    plt.show()
```

3.4 Visualise the learned Kernels

def visualize_kernels(layer, title): kernels = layer.weight.data.clone() kernels = (kernels - kernels.min()) / (kernels.max() - kernels.min()) # Normalize to [0,1] fig, axes = plt.subplots(1, 6, figsize=(15, 3)) for idx, ax in enumerate(axes): npimg = kernels[idx].numpy() npimg = np.transpose(npimg, (1, 2, 0)) ax.imshow(npimg) ax.set_title(f'{title} #{idx}') ax.axis('off') plt.show()

visualize_kernels(net.conv1, 'Conv1 Kernel')

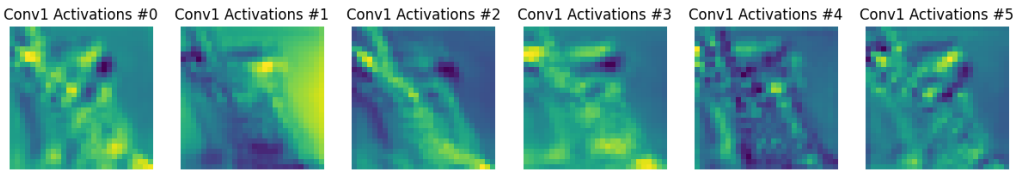
3.5 Intermediate Feature Maps

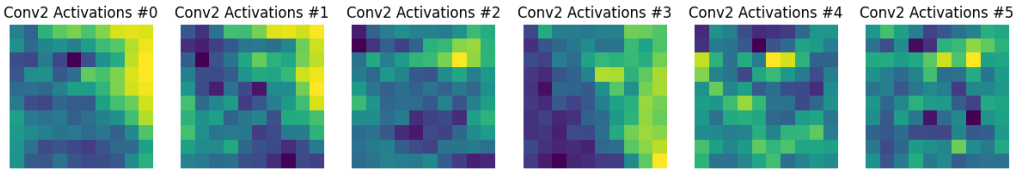
[8]:
```
def get_activations(net, image):
    activations = {}

    def hook(module, input, output):
        activations[module] = output.detach()

    net.conv1.register_forward_hook(hook)
    net.conv2.register_forward_hook(hook)

    _ = net(image.unsqueeze(0))  # Forward pass with batch size 1

    return activations
```

```
# Get a single image
dataiter = iter(trainloader)
images, labels = next(dataiter)
img = images[0]

imshow(img, f'Ground Truth: {classes[labels[0]]}')
activations = get_activations(net, img)
```

3.6 Visualising the Activations

[9]:
```
def plot_activations(acts, title, num_filters=6):
    act = acts.squeeze(0)
    fig, axes = plt.subplots(1, num_filters, figsize=(15, 3))
    for i, ax in enumerate(axes):
        ax.imshow(act[i].cpu(), cmap='viridis')
        ax.set_title(f'{title} #{i}')
        ax.axis('off')
    plt.show()

plot_activations(activations[net.conv1], "Conv1 Activations")
plot_activations(activations[net.conv2], "Conv2 Activations")
```




Conv2 Activations #0
Conv2 Activations #1
Conv2 Activations #2
Conv2 Activations #3
Conv2 Activations #4
Conv2 Activations #5

Lab Work 12
Object Detection with CNN Backbone

1 Aim: To implement objection with CNN Backbone and using bounding boxes

2 Theory

Convolutional Neural Networks (CNNs) are employed for object detection to locate and classify objects in images by predicting bounding boxes, widely used in tasks like autonomous driving, surveillance, and image analysis on datasets such as COCO or Pascal VOC. The objective is to identify objects' locations via bounding boxes and their corresponding class labels, leveraging CNNs' ability to extract spatial features.

Bounding Box Representation: A bounding box is defined by coordinates $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ or center-width-height format (x_c, y_c, w, h), where (x_c, y_c) is the box center, and w, h are width and height. For an image $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, a CNN predicts a set of bounding boxes $\mathcal{B} = \{(\mathbf{b}_i, c_i, s_i)\}_{i=1}^{N}$, where $\mathbf{b}_i$ is the box, c_i is the class label (e.g., "car", "person"), and s_i is a confidence score indicating object presence.

CNN Mechanisms for Detection: CNN-based object detectors, such as Faster R-CNN or YOLO, process images in stages:

- **Feature Extraction**: A CNN backbone (e.g., ResNet) generates feature maps $\mathbf{Z} \in \mathbb{R}^{H' \times W' \times C'}$ via convolutions:
$$\mathbf{Z} = \text{CNN}(\mathbf{X}), \quad \text{Conv}(\mathbf{X}) = \sum_{m,n,c} \mathbf{W}_{m,n,c} \mathbf{X}_{i+m,j+n,c} + b.$$
- **Region Proposals**: Methods like Region Proposal Networks (RPNs) predict candidate bounding boxes on feature maps, scoring potential object locations.
- **Box Regression and Classification**: Fully connected or convolutional heads refine box coordinates and predict class probabilities:
$$\mathbf{b}' = \mathbf{b} + \Delta\mathbf{b}, \quad \mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{z} + \mathbf{b}),$$
where $\Delta\mathbf{b}$ adjusts box coordinates, and $\mathbf{z}$ is a feature vector.

Loss Functions: Training minimizes a multi-task loss combining localization, classification, and confidence:
$$\mathcal{L} = \lambda_{\text{loc}} \sum_i \ell_{\text{loc}}(\mathbf{b}_i, \mathbf{b}_i^*) + \lambda_{\text{cls}} \sum_i \ell_{\text{cls}}(c_i, c_i^*) + \lambda_{\text{conf}} \sum_i \ell_{\text{conf}}(s_i, s_i^*),$$
where ℓ_{loc} (e.g., Smooth L1 loss) penalizes box coordinate errors, ℓ_{cls} (e.g., cross-entropy) penalizes class errors, ℓ_{conf} (e.g., binary cross-entropy) penalizes confidence errors, and λ terms balance contributions. Ground-truth boxes $\mathbf{b}_i^*$ are provided by annotated datasets.

Advantages and Trade-offs: Bounding box-based CNNs accurately localize and classify objects, leveraging hierarchical features. They handle multiple objects per image but require large annotated datasets and computational resources. Overlapping or small objects may challenge detection accuracy, necessitating advanced techniques like non-maximum suppression.

3 Code

3.1 Importing all the libraries

[13]:
```
import torch
from torchvision.models.detection import ssdlite320_mobilenet_v3_large
from torchvision.datasets import VOCDetection
from torchvision.transforms import functional as F
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import os
```

[14]:
```
device = torch.device('cuda:0') if torch.cuda.is_available() else torch.
↪device('cpu')
```

3.2 Loading the Pascal VOC dataset

[15]:
```
class VOCDataset(torch.utils.data.Dataset):
    def __init__(self, root, year="2007", image_set="train", transform=None):
        self.dataset = VOCDetection(root=root, year=year, image_set=image_set,␣
↪download=True)
        self.transform = transform

    def __getitem__(self, idx):
        img, target = self.dataset[idx]
        if self.transform:
            img = self.transform(img)

        objects = target['annotation']['object']
        if isinstance(objects, dict):
            objects = [objects]  # VOC sometimes wraps single object as dict

        boxes = []
        labels = []

        for obj in objects:
            bbox = obj['bndbox']
            xmin = float(bbox['xmin'])
            ymin = float(bbox['ymin'])
            xmax = float(bbox['xmax'])
            ymax = float(bbox['ymax'])
            boxes.append([xmin, ymin, xmax, ymax])
            labels.append(1)  # Just a placeholder label

        target = {
            'boxes': torch.tensor(boxes, dtype=torch.float32),
```

```
            'labels': torch.tensor(labels, dtype=torch.int64)
        }

        return img, target

    def __len__(self):
        return len(self.dataset)
```

[16]:
```
from torchvision.transforms import Resize, ToTensor, Compose

transform = Compose([
    Resize((320, 320)),
    ToTensor()
])

train_dataset = VOCDataset(root='./', year='2007', image_set='train',␣
↪transform=transform)

train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True,␣
↪collate_fn=lambda x: tuple(zip(*x)))
```

3.3 Now we load the model which is SSD with MobineNet Backbone

[ ]:
```
model = ssdlite320_mobilenet_v3_large(pretrained=True)
model.train()
model.to(device)
```

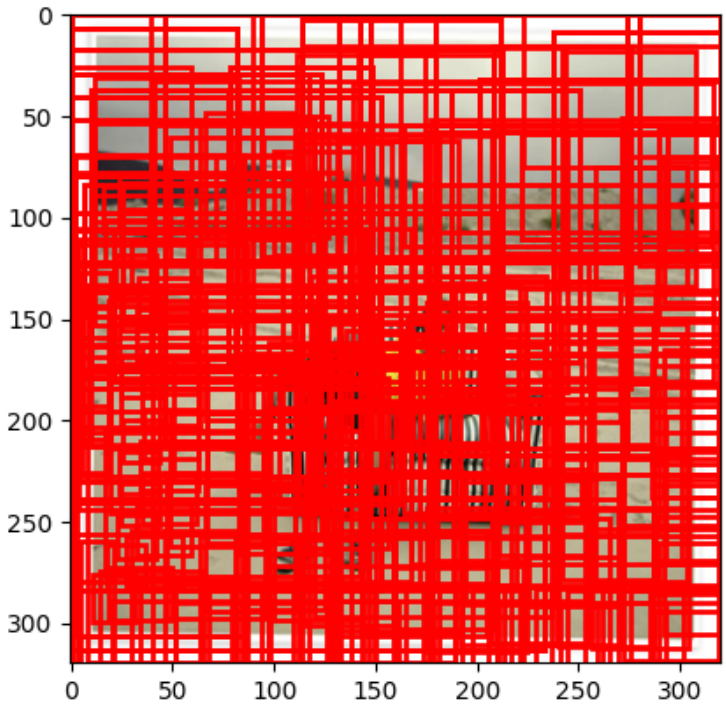
[18]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=0.005, momentum=0.9,␣
↪weight_decay=0.0005)

for epoch in range(1):
    model.train()
    for i, (images, targets) in enumerate(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]

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

            optimizer.zero_grad()
            losses.backward()
            optimizer.step()

            print(f"[Epoch {epoch+1}] Step {i+1} Loss: {losses.item():.4f}")
        except Exception as e:
            print(f"Skipping batch due to error: {e}")
```

[19]:
```
from torchvision.ops import box_iou

def compute_map(model, data_loader, iou_threshold=0.5):
    model.eval()
    aps = []

    with torch.no_grad():
        for images, targets in data_loader:
            images = list(img.cuda() for img in images)
            outputs = model(images)

            for output, target in zip(outputs, targets):
                pred_boxes = output['boxes'].cpu()
                gt_boxes = target['boxes'].cpu()
                iou = box_iou(pred_boxes, gt_boxes)

                if iou.numel() == 0:
                    continue
                max_iou = iou.max(dim=1)[0]
                correct = max_iou > iou_threshold
                ap = correct.sum().item() / len(gt_boxes)
                aps.append(ap)

    return sum(aps) / len(aps)

mAP = compute_map(model, train_loader)
print(f"mAP: {mAP:.4f}")
```

```
mAP: 1.1151
```

3.4 We can visualise the results from our model

[20]:
```
def visualize_prediction(img, boxes):
    img = img.permute(1, 2, 0).numpy()
    plt.imshow(img)
    for box in boxes:
        x1, y1, x2, y2 = box
        plt.gca().add_patch(plt.Rectangle((x1, y1), x2 - x1, y2 - y1,
                                          fill=False, edgecolor='red',␣
 ↪linewidth=2))
    plt.show()

model.eval()
images, targets = next(iter(train_loader))
with torch.no_grad():
    predictions = model([img.cuda() for img in images])
visualize_prediction(images[0], predictions[0]['boxes'].cpu())
```


0
50
100
150
200
250
300
0
50
100
150
200
250
300

Lab Work 13
Transfer Learning on a Custom Dataset

1 Aim: To practice freezeing and unfreezing of layers on a pre-trained CNN architecture

2 Theory

Transfer Learning leverages pre-trained models to enhance performance on new, often smaller, datasets for tasks like image classification, object detection, or natural language processing. The objective is to transfer knowledge learned from a large, general dataset (e.g., ImageNet for vision, BERT for text) to a target task with limited data, improving efficiency and generalization compared to training from scratch.

Mechanism: A pre-trained model, with parameters $\theta_{\text{pre}} \in \mathbb{R}^d$ optimized on a source dataset $\mathcal{D}_{\text{source}}$, provides a starting point. The model, typically a deep neural network (e.g., ResNet for images, BERT for text), consists of:

- **Feature Extractor**: Early layers capturing general features (e.g., edges in images, word embeddings in text), represented as $\mathbf{h} = f(\mathbf{x}; \theta_{\text{pre}}^{\text{early}})$.
- **Task-Specific Head**: Final layers tailored to the source task (e.g., classification), $\mathbf{y} = g(\mathbf{h}; \theta_{\text{pre}}^{\text{head}})$.

In Transfer Learning, the feature extractor is reused, and the head is adapted to the target task on dataset $\mathcal{D}_{\text{target}}$:

$$\mathbf{y}_{\text{target}} = g'(\mathbf{h}; \theta_{\text{new}}^{\text{head}}).$$

Approaches:

- **Feature Extraction**: Freeze $\theta_{\text{pre}}^{\text{early}}$, training only $\theta_{\text{new}}^{\text{head}}$ on $\mathcal{D}_{\text{target}}$. Suitable for small datasets.
- **Fine-Tuning**: Update both $\theta_{\text{pre}}^{\text{early}}$ and $\theta_{\text{new}}^{\text{head}}$, often with a small learning rate for early layers to preserve general features. Ideal for larger datasets or similar tasks.

Advantages and Trade-offs: Transfer Learning reduces training time and data requirements, leveraging robust features from large datasets to achieve high accuracy (e.g., 90% vs. 70% from scratch on small datasets). It mitigates overfitting in data-scarce scenarios but risks negative transfer if source and target domains differ significantly. Fine-tuning may overfit small datasets, while feature extraction may underfit if task-specific features are needed.

Training Dynamics: The model minimizes a task-specific loss (e.g., cross-entropy for classification) on $\mathcal{D}_{\text{target}}$:

$$\mathcal{L} = -\sum_{i=1}^{C} t_i \log(y_i).$$

Optimizers like Adam or SGD are used, with regularization (e.g., dropout) and learning rate schedules to balance adaptation and preservation of pre-trained knowledge. Transfer Learning converges faster than training from scratch, especially for feature extraction.

3 Code

3.1 Loading the libraries

[14]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from torchvision.datasets import ImageFolder
import time
import matplotlib.pyplot as plt
import os
from PIL import Image
```

[15]:
```
device = torch.device("mps")
print(f"Using device: {device}")
```

```
Using device: mps
```

3.2 Preparing the dataset

[17]:
```
data_dir = "./data/flowers/"


transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

dataset = ImageFolder(root=data_dir, transform=transform)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset,␣
↪[train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,␣
↪num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False,␣
↪num_workers=2)
```

3.3 Now we can train the model

[18]:
```
def train_model(model, train_loader, criterion, optimizer, num_epochs,␣
↪phase="frozen"):
    model.to(device)
    start_time = time.time()

    train_losses = []
    train_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        print(f'{phase} - Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.3f}, '
              f'Accuracy: {epoch_acc:.2f}%')

    training_time = time.time() - start_time
    return training_time, train_losses, train_accuracies
```

[19]:
```
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
```

```
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return accuracy
```

3.4 Let's load the ResNet18 model

[20]:
```
model = torchvision.models.resnet18(pretrained=True)
```

```
/Users/amishgogia/tensorflow_tutorial/venv/lib/python3.12/site-
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(
/Users/amishgogia/tensorflow_tutorial/venv/lib/python3.12/site-
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=ResNet18_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

3.5 We will modify the final layer to finally outpout 5 classes for 5 classes of flowers

[21]:
```
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 5)
```

3.6 Let's freeze all layers initially

[22]:
```
for param in model.parameters():
    param.requires_grad = False
```

[23]:
```
model.fc.weight.requires_grad = True
model.fc.bias.requires_grad = True
```

3.7 Phase 1 - Train with Frozen Layers

[24]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
```

[25]:
```
print("Phase 1: Training with frozen layers...")
frozen_time, frozen_losses, frozen_accs = train_model(model, train_loader,
                                                      criterion, optimizer,
```

```
                                                    num_epochs=5,
                                                    phase="Frozen")
```

```
Phase 1: Training with frozen layers...
Frozen - Epoch 1/5, Loss: 0.831, Accuracy: 71.68%
Frozen - Epoch 2/5, Loss: 0.468, Accuracy: 84.62%
Frozen - Epoch 3/5, Loss: 0.394, Accuracy: 86.94%
Frozen - Epoch 4/5, Loss: 0.362, Accuracy: 87.32%
Frozen - Epoch 5/5, Loss: 0.325, Accuracy: 88.94%
```

3.8 Phase 2 - Partial Unfreezing

[26]:
```
for param in model.layer4.parameters():
    param.requires_grad = True
for param in model.layer3.parameters():
    param.requires_grad = True
```

[27]:
```
optimizer = optim.Adam([
    {'params': model.fc.parameters(), 'lr': 0.001},
    {'params': model.layer4.parameters(), 'lr': 0.0001},
    {'params': model.layer3.parameters(), 'lr': 0.0001}
])
```

[28]:
```
print("\nPhase 2: Fine-tuning with partial unfreezing...")
finetune_time, finetune_losses, finetune_accs = train_model(model, train_loader,
                                                    criterion, optimizer,
                                                    num_epochs=5,
                                                    phase="Fine-tuned")
```

```
Phase 2: Fine-tuning with partial unfreezing...
Fine-tuned - Epoch 1/5, Loss: 0.304, Accuracy: 89.31%
Fine-tuned - Epoch 2/5, Loss: 0.062, Accuracy: 98.29%
Fine-tuned - Epoch 3/5, Loss: 0.023, Accuracy: 99.36%
Fine-tuned - Epoch 4/5, Loss: 0.013, Accuracy: 99.59%
Fine-tuned - Epoch 5/5, Loss: 0.016, Accuracy: 99.51%
```

3.9 Now let's evaluate both Phases

[29]:
```
frozen_acc = evaluate_model(model, test_loader)
print(f"\nTest Accuracy after frozen phase: {frozen_acc:.2f}%")

# Continue fine-tuning for evaluation
finetune_acc = evaluate_model(model, test_loader)
print(f"Test Accuracy after fine-tuning: {finetune_acc:.2f}%")
```

```
Test Accuracy after frozen phase: 92.01%
Test Accuracy after fine-tuning: 92.01%
```

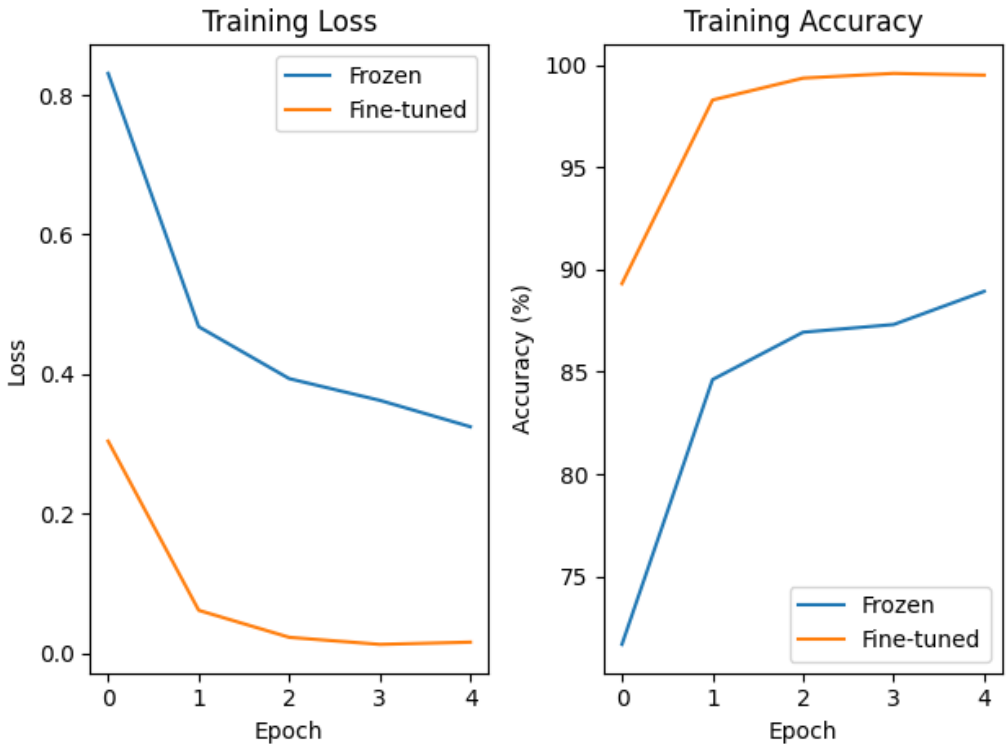
[30]:
```
plt.figure(figsize=(12, 4))
```

[30]: <Figure size 1200x400 with 0 Axes>

<Figure size 1200x400 with 0 Axes>

3.10 Finally we can plot all the results

[31]:
```
plt.subplot(1, 2, 1)
plt.plot(frozen_losses, label='Frozen')
plt.plot(finetune_losses, label='Fine-tuned')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(frozen_accs, label='Frozen')
plt.plot(finetune_accs, label='Fine-tuned')
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

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



[32]:
```
print("\nResults Summary:")
print(f"Frozen Phase - Training Time: {frozen_time:.2f}s, Test Accuracy:␣
↪{frozen_acc:.2f}%")
print(f"Fine-tuned Phase - Training Time: {finetune_time:.2f}s, Test Accuracy:␣
↪{finetune_acc:.2f}%")
```

```
Results Summary:
Frozen Phase - Training Time: 131.86s, Test Accuracy: 92.01%
Fine-tuned Phase - Training Time: 188.45s, Test Accuracy: 92.01%
```

Lab Work 14

Recurrent Neural Networks (RNN) for Time Series Analysis

1 Aim: To train Recurrent Neural Networks (RNNs) for Time Series forcasting

2 Theory

Recurrent Neural Networks (RNNs) are designed for sequential data processing, making them well-suited for time series analysis tasks such as stock price forecasting, weather prediction, or anomaly detection in sensor data. The objective is to model temporal dependencies in time series data $\mathbf{x} = (x_1, x_2, \ldots, x_T)$, where $x_t \in \mathbb{R}^d$ represents features at time t, to predict future values or classify sequences.

RNN Architecture: RNNs process sequences by maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^h$ that captures information from previous time steps. For an input x_t, the hidden state and output are computed as:

$$\mathbf{h}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x x_t + \mathbf{b}_h), \quad \mathbf{y}_t = \mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y,$$

where $\mathbf{W}_h \in \mathbb{R}^{h \times h}$, $\mathbf{W}_x \in \mathbb{R}^{h \times d}$, $\mathbf{W}_y \in \mathbb{R}^{o \times h}$ are weight matrices, $\mathbf{b}_h, \mathbf{b}_y$ are biases, σ is a non-linear activation (e.g., tanh), and $\mathbf{y}_t \in \mathbb{R}^o$ is the output (e.g., predicted value or class). The recurrent structure allows RNNs to model temporal dependencies by passing $\mathbf{h}_t$ to the next time step.

Variants: Standard RNNs suffer from vanishing/exploding gradients, limiting their ability to capture long-term dependencies. Variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) address this with gating mechanisms:

$$\text{LSTM} : \mathbf{h}_t, \mathbf{c}_t = \text{LSTM}(x_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}),$$

where $\mathbf{c}_t$ is a cell state preserving long-term information. These variants enhance RNNs' ability to model extended sequences.

Advantages and Trade-offs: RNNs excel at capturing sequential patterns, making them effective for time series with temporal correlations. They handle variable-length sequences and learn complex dynamics. However, they are computationally intensive, sensitive to hyperparameter tuning (e.g., learning rate, hidden size), and may struggle with very long sequences unless using LSTM/GRU. Compared to CNNs or Transformers, RNNs can be slower due to sequential processing.

Training Dynamics: RNNs minimize a task-specific loss, such as Mean Squared Error (MSE) for forecasting:

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2,$$

or cross-entropy for classification. Training uses backpropagation through time (BPTT), with optimizers like Adam or SGD. Regularization (e.g., dropout) and gradient clipping prevent overfitting and stabilize training. Data preprocessing (e.g., normalization, windowing) is crucial for effective learning.

3 Code

3.1 Importing all the libraries

[1]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import time
```

[2]:
```
device = torch.device("mps")
print(f"Using device: {device}")
```

```
Using device: mps
```

[14]:
```
data = pd.read_csv('./data/1_Daily_minimum_temps.csv',
                   parse_dates=['Date'],
                   index_col='Date',
                   date_format='%m/%d/%Y')
```

3.2 Since it is know this dataset contains some string values in temperature we will change those values to the mean values

[16]:
```
data['Temp'] = pd.to_numeric(data['Temp'], errors='coerce')

data['Temp'] = data['Temp'].fillna(data['Temp'].mean())
```

[17]:
```
data
```

[17]:
```
            Temp
Date
01/01/81  20.7
01/02/81  17.9
01/03/81  18.8
01/04/81  14.6
01/05/81  15.8
...        ...
12/27/90  14.0
12/28/90  13.6
12/29/90  13.5
12/30/90  15.7
12/31/90  13.0

[3650 rows x 1 columns]
```

[19]:
```
series = data['Temp'].values.reshape(-1, 1)
```

3.3 Normalizing the corrected data

[20]:
```
scaler = MinMaxScaler(feature_range=(-1, 1))
series_normalized = scaler.fit_transform(series)
```

3.4 Creating the sequences

[21]:
```
def create_sequences(data, seq_length):
    X, y = [], []
    for i in range(len(data) - seq_length):
        X.append(data[i:i + seq_length])
        y.append(data[i + seq_length])
    return np.array(X), np.array(y)
```

[22]:
```
seq_length = 10
X, y = create_sequences(series_normalized, seq_length)
```

[23]:
```
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

[24]:
```
X_train = torch.FloatTensor(X_train).to(device)
y_train = torch.FloatTensor(y_train).to(device)
X_test = torch.FloatTensor(X_test).to(device)
y_test = torch.FloatTensor(y_test).to(device)
```

3.5 Let's first define the RNN model

[25]:
```
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x shape: (batch_size, seq_length, input_size)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Take output from last time step
        return out
```

3.6 And now the LSTM model

[26]:
```
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=1):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Take output from last time step
        return out
```

[27]:
```
def train_model(model, X_train, y_train, num_epochs=100, lr=0.01):
    model.to(device)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    start_time = time.time()
    losses = []

    for epoch in range(num_epochs):
        model.train()
        outputs = model(X_train)
        optimizer.zero_grad()
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())
        if (epoch + 1) % 20 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

    training_time = time.time() - start_time
    return training_time, losses
```

[28]:
```
def evaluate_model(model, X_test, y_test):
    model.eval()
    with torch.no_grad():
        predictions = model(X_test)
        mse = nn.MSELoss()(predictions, y_test)
    return mse.item(), predictions.cpu().numpy()
```

[29]:
```
input_size = 1
hidden_size = 32
output_size = 1
num_layers = 1
```

[30]:
```
rnn_model = SimpleRNN(input_size, hidden_size, output_size, num_layers)
print("Training RNN...")
rnn_time, rnn_losses = train_model(rnn_model, X_train, y_train)
```

```
Training RNN...
Epoch [20/100], Loss: 0.0420
Epoch [40/100], Loss: 0.0358
Epoch [60/100], Loss: 0.0356
Epoch [80/100], Loss: 0.0354
Epoch [100/100], Loss: 0.0354
```

[31]:
```
lstm_model = LSTMModel(input_size, hidden_size, output_size, num_layers)
print("\nTraining LSTM...")
lstm_time, lstm_losses = train_model(lstm_model, X_train, y_train)
```

```
Training LSTM...
Epoch [20/100], Loss: 0.0466
Epoch [40/100], Loss: 0.0404
Epoch [60/100], Loss: 0.0384
Epoch [80/100], Loss: 0.0359
Epoch [100/100], Loss: 0.0354
```

3.7 Let's evaluate the performance of the two models

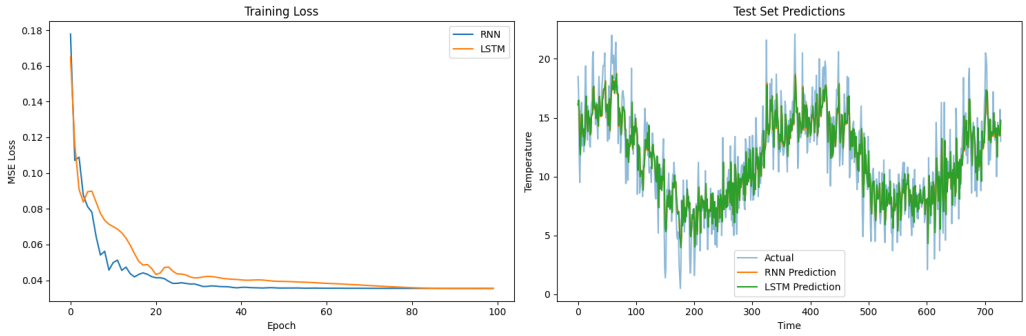
[32]:
```
rnn_mse, rnn_preds = evaluate_model(rnn_model, X_test, y_test)
lstm_mse, lstm_preds = evaluate_model(lstm_model, X_test, y_test)
```

[33]:
```
y_test_np = y_test.cpu().numpy()
rnn_preds_transformed = scaler.inverse_transform(rnn_preds)
lstm_preds_transformed = scaler.inverse_transform(lstm_preds)
y_test_transformed = scaler.inverse_transform(y_test_np)
```

3.8 Let's plot the results

[34]:
```
plt.figure(figsize=(15, 5))

# Loss Curves
plt.subplot(1, 2, 1)
plt.plot(rnn_losses, label='RNN')
plt.plot(lstm_losses, label='LSTM')
plt.title('Training Loss')
plt.xlabel('Epoch')
```

```
plt.ylabel('MSE Loss')
plt.legend()

# Predictions vs Actual
plt.subplot(1, 2, 2)
plt.plot(y_test_transformed, label='Actual', alpha=0.5)
plt.plot(rnn_preds_transformed, label='RNN Prediction')
plt.plot(lstm_preds_transformed, label='LSTM Prediction')
plt.title('Test Set Predictions')
plt.xlabel('Time')
plt.ylabel('Temperature')
plt.legend()

plt.tight_layout()
plt.show()

# Cell 10: Results Summary
print("\nResults Summary:")
print(f"RNN - Training Time: {rnn_time:.2f}s, Test MSE: {rnn_mse:.4f}")
print(f"LSTM - Training Time: {lstm_time:.2f}s, Test MSE: {lstm_mse:.4f}")
```




```
Results Summary:
RNN - Training Time: 2.50s, Test MSE: 0.0282
LSTM - Training Time: 1.83s, Test MSE: 0.0282
```

Lab Work 15

Comparison between LSTM and GRU for Sentiment analysis

1 Aim: To compare the performance of the LSTM and GRU by performing a sentiment analysis on the IMDb dataset

2 Theory

Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks are advanced recurrent neural networks (RNNs) designed to model sequential data, particularly suited for sentiment analysis, where the goal is to classify text sequences (e.g., reviews) as positive, negative, or neutral. The objective is to capture long-term dependencies in text to understand sentiment context, leveraging LSTM and GRU's gating mechanisms to overcome the vanishing gradient problem of standard RNNs.

LSTM Architecture: LSTMs process a sequence $\mathbf{x} = (x_1, \ldots, x_T)$, where $x_t \in \mathbb{R}^d$ is a word embedding, maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^h$ and cell state $\mathbf{c}_t \in \mathbb{R}^h$. The update at time t is:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_f), \quad \mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_i),$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_o), \quad \tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_c),$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \quad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t),$$

where $\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t$ are forget, input, and output gates, and $\odot$ denotes element-wise multiplication. The cell state $\mathbf{c}_t$ retains long-term information, enabling LSTMs to model extended dependencies.

GRU Architecture: GRUs simplify LSTMs by merging hidden and cell states into a single hidden state $\mathbf{h}_t$. The update is:

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_r), \quad \mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_z),$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}, x_t] + \mathbf{b}_h), \quad \mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t,$$

where $\mathbf{r}_t, \mathbf{z}_t$ are reset and update gates. GRUs are computationally lighter with fewer parameters, balancing efficiency and dependency modeling.

Advantages and Trade-offs: LSTMs excel at capturing long-term dependencies in complex sequences, ideal for lengthy reviews with nuanced sentiment, but are computationally intensive. GRUs are faster and require less memory, performing comparably for shorter sequences or simpler datasets, though they may struggle with very long dependencies. Both outperform standard RNNs but may be surpassed by Transformers for large-scale tasks.

Training Dynamics: LSTMs and GRUs minimize a loss function, typically cross-entropy for sentiment classification:

$$\mathcal{L} = -\sum_{i=1}^{C} t_i \log(y_i),$$

where t_i is the true label and y_i is the predicted probability from $\mathbf{h}_T$. Training uses backpropagation through time with optimizers like Adam, incorporating regularization (e.g., dropout) and pre-trained embeddings (e.g., GloVe) to enhance generalization.

3 Code

3.1 Importing the libraries

[5]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import nltk
from nltk.tokenize import word_tokenize
from collections import Counter
import time
import matplotlib.pyplot as plt
```

[20]:
```
nltk.download('punkt_tab')
```

```
[nltk_data] Downloading package punkt_tab to
[nltk_data]     /Users/amishgogia/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
```

[20]:
```
True
```

[8]:
```
device = torch.device("mps")
print(f"Using device: {device}")
```

```
Using device: mps
```

3.2 Load the dataset

[10]:
```
data = pd.read_csv('./dl_lab_datasets/IMDB Dataset.csv')
```

[11]:
```
data['sentiment'] = data['sentiment'].map({'negative': 0, 'positive': 1})
```

3.3 We need to tokenize this data, for that we are using nltk

[12]:
```
def tokenize(text):
    return word_tokenize(text.lower())
```

[13]:
```
def build_vocab(texts, max_size=10000):
    counter = Counter()
    for text in texts:
        counter.update(tokenize(text))
    # Reserve 0 for padding, 1 for unknown
    vocab = {word: i+2 for i, (word, _) in enumerate(counter.
 ↪most_common(max_size-2))}
    vocab['<PAD>'] = 0
    vocab['<UNK>'] = 1
```

```
    return vocab
```

[21]:
```
vocab = build_vocab(data['review'])
vocab_size = len(vocab)
```

3.4 Converting our text to sequences

[22]:
```
def text_to_sequence(text, vocab):
    return [vocab.get(token, vocab['<UNK>']) for token in tokenize(text)]

# Convert all reviews to sequences
sequences = [torch.tensor(text_to_sequence(text, vocab)) for text in␣
 ↪data['review']]
labels = torch.tensor(data['sentiment'].values, dtype=torch.long)
```

[23]:
```
max_len = 200  # Limit sequence length
padded_sequences = []
for seq in sequences:
    if len(seq) < max_len:
        padded_seq = torch.cat([seq, torch.zeros(max_len - len(seq), dtype=torch.
 ↪long)])
    else:
        padded_seq = seq[:max_len]
    padded_sequences.append(padded_seq)
padded_sequences = torch.stack(padded_sequences)
```

[24]:
```
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, labels,
                                                    test_size=0.2,␣
 ↪random_state=42)
```

[25]:
```
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
test_dataset = torch.utils.data.TensorDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

3.5 First we define the LSTM model

[26]:
```
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size,␣
 ↪num_layers=1):
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        embedded = self.embedding(x)  # (batch_size, seq_len, embedding_dim)
        output, (hidden, cell) = self.lstm(embedded)
        return self.fc(hidden[-1])  # Last hidden state
```

3.6 Now we define the GRU model

[27]:
```
class GRUClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size,␣
↪num_layers=1):
        super(GRUClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.gru = nn.GRU(embedding_dim, hidden_size, num_layers,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        embedded = self.embedding(x)  # (batch_size, seq_len, embedding_dim)
        output, hidden = self.gru(embedded)
        return self.fc(hidden[-1])  # Last hidden state
```

[28]:
```
def train_model(model, train_loader, criterion, optimizer, num_epochs=10):
    model.to(device)
    start_time = time.time()
    train_losses = []
    train_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
```

```
        epoch_acc = 100 * correct / total
        train_losses.append(epoch_loss)
        train_accuracies.append(epoch_acc)

        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, '
              f'Accuracy: {epoch_acc:.2f}%')

    training_time = time.time() - start_time
    return training_time, train_losses, train_accuracies
```

[29]:
```
def evaluate_model(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy
```

[31]:
```
embedding_dim = 100
hidden_size = 128
output_size = 2  # Binary classification
```

3.7 Let's train the models

[32]:
```
# LSTM
lstm_model = LSTMClassifier(vocab_size, embedding_dim, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer_lstm = torch.optim.Adam(lstm_model.parameters(), lr=0.001)
print("Training LSTM...")
lstm_time, lstm_losses, lstm_accs = train_model(lstm_model, train_loader,␣
↪criterion, optimizer_lstm)

# GRU
gru_model = GRUClassifier(vocab_size, embedding_dim, hidden_size, output_size)
optimizer_gru = torch.optim.Adam(gru_model.parameters(), lr=0.001)
print("\nTraining GRU...")
gru_time, gru_losses, gru_accs = train_model(gru_model, train_loader, criterion,␣
↪optimizer_gru)
```

```
Training LSTM...
Epoch 1/10, Loss: 0.6898, Accuracy: 52.81%
Epoch 2/10, Loss: 0.6288, Accuracy: 61.60%
```

```
Epoch 3/10, Loss: 0.3435, Accuracy: 85.42%
Epoch 4/10, Loss: 0.2602, Accuracy: 89.61%
Epoch 5/10, Loss: 0.2043, Accuracy: 92.13%
Epoch 6/10, Loss: 0.1521, Accuracy: 94.50%
Epoch 7/10, Loss: 0.1021, Accuracy: 96.61%
Epoch 8/10, Loss: 0.0661, Accuracy: 97.97%
Epoch 9/10, Loss: 0.0428, Accuracy: 98.86%
Epoch 10/10, Loss: 0.0313, Accuracy: 99.15%

Training GRU...
Epoch 1/10, Loss: 0.6875, Accuracy: 53.93%
Epoch 2/10, Loss: 0.4378, Accuracy: 79.35%
Epoch 3/10, Loss: 0.2724, Accuracy: 88.76%
Epoch 4/10, Loss: 0.1988, Accuracy: 92.35%
Epoch 5/10, Loss: 0.1372, Accuracy: 95.02%
Epoch 6/10, Loss: 0.0818, Accuracy: 97.28%
Epoch 7/10, Loss: 0.0479, Accuracy: 98.50%
Epoch 8/10, Loss: 0.0337, Accuracy: 99.00%
Epoch 9/10, Loss: 0.0259, Accuracy: 99.23%
Epoch 10/10, Loss: 0.0192, Accuracy: 99.42%
```

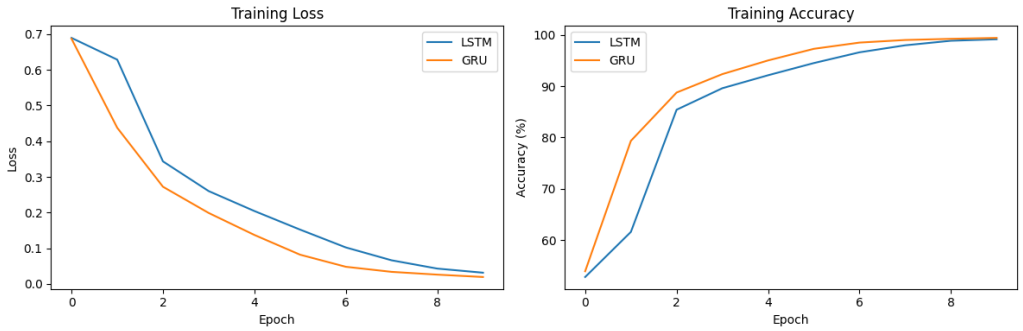
3.8 Let's evaluate the models

[33]:
```
lstm_test_acc = evaluate_model(lstm_model, test_loader)
gru_test_acc = evaluate_model(gru_model, test_loader)
```

3.9 We can also draw some plots

[38]:
```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(lstm_losses, label='LSTM')
plt.plot(gru_losses, label='GRU')
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(lstm_accs, label='LSTM')
plt.plot(gru_accs, label='GRU')
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend()

plt.tight_layout()
```

```
plt.show()
```



[36]:
```
print("\nResults Summary:")
print(f"LSTM - Training Time: {lstm_time:.2f}s, Test Accuracy: {lstm_test_acc:.
↪2f}%")
print(f"GRU - Training Time: {gru_time:.2f}s, Test Accuracy: {gru_test_acc:.
↪2f}%")
```

```
Results Summary:
LSTM - Training Time: 236.08s, Test Accuracy: 85.80%
GRU - Training Time: 2239.20s, Test Accuracy: 86.15%
```

Lab Work 16
Seq2Seq model with Attention mechanism

1 Aim: Implement a basic Seq2Seq model (e.g., for translation or summarization) with attention.

2 Theory

The Sequence-to-Sequence (Seq2Seq) model with an attention mechanism is a deep learning framework designed for mapping input sequences to output sequences, widely used in tasks like machine translation, text summarization, and chatbot responses. The objective is to model complex dependencies between variable-length input and output sequences, leveraging attention to focus on relevant input parts during decoding, improving performance over vanilla Seq2Seq models.

Seq2Seq Architecture: The model comprises an encoder and a decoder, typically implemented with Recurrent Neural Networks (RNNs) like LSTMs or GRUs. For an input sequence $\mathbf{x} = (x_1, \ldots, x_{T_x})$, the encoder processes each token to produce a sequence of hidden states:

$$\mathbf{h}_t = \mathrm{RNN}_{\mathrm{enc}}(x_t, \mathbf{h}_{t-1}), \quad \mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}],$$

where $\mathbf{h}_t \in \mathbb{R}^h$ is the hidden state. The decoder generates the output sequence $\mathbf{y} = (y_1, \ldots, y_{T_y})$ autoregressively:

$$\mathbf{s}_t = \mathrm{RNN}_{\mathrm{dec}}(y_{t-1}, \mathbf{s}_{t-1}, \mathbf{c}_t), \quad y_t = \mathrm{softmax}(\mathbf{W}_y \mathbf{s}_t + \mathbf{b}_y),$$

where $\mathbf{s}_t$ is the decoder state, and $\mathbf{c}_t$ is a context vector from the attention mechanism.

Attention Mechanism: Attention dynamically weights the encoder hidden states $\mathbf{H}$ to compute the context vector $\mathbf{c}_t$ for each decoder step:

$$\alpha_{t,i} = \frac{\exp(\mathrm{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_{j=1}^{T_x} \exp(\mathrm{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))}, \quad \mathbf{c}_t = \sum_{i=1}^{T_x} \alpha_{t,i} \mathbf{h}_i,$$

where $\mathrm{score}(\mathbf{s}_{t-1}, \mathbf{h}_i)$ is often a dot product or additive function (e.g., $\mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{t-1} + \mathbf{W}_h \mathbf{h}_i)$). The attention weights $\alpha_{t,i}$ indicate the importance of input token i for output token t, enabling the model to focus on relevant input parts (e.g., aligning words in translation).

Advantages and Trade-offs: Attention improves Seq2Seq performance by addressing the bottleneck of fixed-length context vectors, enhancing alignment in tasks like translation. It captures long-range dependencies and provides interpretable alignments. However, it increases computational complexity and may overfit small datasets without regularization (e.g., dropout). Compared to Transformers, Seq2Seq with attention is less parallelizable but simpler for certain tasks.

Training Dynamics: The model minimizes a loss function, typically cross-entropy for predicting output tokens:

$$\mathcal{L} = -\sum_{t=1}^{T_y} \log p(y_t | y_{1:t-1}, \mathbf{x}).$$

Training uses backpropagation through time with optimizers like Adam, often incorporating teacher forcing (feeding ground-truth inputs during decoding) and pre-trained embeddings (e.g., word2vec). Regularization and beam search decoding enhance generalization and output quality.

3 Code

3.1 Importing the necessary libraries

[2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd
from torch.nn.utils.rnn import pad_sequence
```

[3]:
```
device = torch.device("mps")
```

3.2 Load the dataset

[5]:
```
data = pd.read_csv("./dl_lab_datasets/eng_-french.csv")
data = data.sample(500)
# df['English words/sentences'].tolist(), df['French words/sentences'].tolist()
```

3.3 We will first have to build a vocabulary

[6]:
```
def build_vocab(sentences):
    vocab = {'<pad>': 0, '<sos>': 1, '<eos>': 2}
    for sentence in sentences:
        for word in sentence.lower().split():
            if word not in vocab:
                vocab[word] = len(vocab)
    return vocab
```

[7]:
```
src_vocab = build_vocab(data['English words/sentences'])
tgt_vocab = build_vocab(data['French words/sentences'])
inv_tgt_vocab = {idx: word for word, idx in tgt_vocab.items()}
```

3.4 Tokenizing the data

[8]:
```
def tokenize(sentence, vocab):
    return [vocab['<sos>']] + [vocab.get(word, vocab['<pad>']) for word in␣
 ↪sentence.lower().split()] + [vocab['<eos>']]
```

[9]:
```
train_src = [torch.tensor(tokenize(src, src_vocab)) for src in data['English␣
 ↪words/sentences']]
train_tgt = [torch.tensor(tokenize(tgt, tgt_vocab)) for tgt in data['French␣
 ↪words/sentences']]
train_src = pad_sequence(train_src, batch_first=True,␣
 ↪padding_value=src_vocab['<pad>']).to(device)
train_tgt = pad_sequence(train_tgt, batch_first=True,␣
 ↪padding_value=tgt_vocab['<pad>']).to(device)
```

3.5 First we have to define the encoder

[10]:
```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hid_dim, batch_first=True)

    def forward(self, src):
        embedded = self.embedding(src)
        outputs, (hidden, cell) = self.rnn(embedded)
        return outputs, hidden, cell
```

3.6 First let's do the decoder without attention

[11]:
```
class DecoderNoAttention(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim):
        super().__init__()
        self.output_dim = output_dim
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hid_dim, batch_first=True)
        self.fc = nn.Linear(hid_dim, output_dim)

    def forward(self, tgt, hidden, cell):
        tgt = tgt.unsqueeze(1)
        embedded = self.embedding(tgt)
        output, (hidden, cell) = self.rnn(embedded, (hidden, cell))
        prediction = self.fc(output.squeeze(1))
        return prediction, hidden, cell
```

3.7 And then with attention

[12]:
```
class DecoderWithAttention(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim):
        super().__init__()
        self.output_dim = output_dim
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim + hid_dim, hid_dim, batch_first=True)
        self.attention = nn.MultiheadAttention(hid_dim, num_heads=1)
        self.fc = nn.Linear(hid_dim, output_dim)

    def forward(self, tgt, hidden, cell, encoder_outputs):
        tgt = tgt.unsqueeze(1)
        embedded = self.embedding(tgt)

        attn_output, attn_weights = self.attention(
            hidden[-1].unsqueeze(0),
            encoder_outputs.transpose(0, 1),
```

```
            encoder_outputs.transpose(0, 1)
        )

        rnn_input = torch.cat((embedded, attn_output.transpose(0, 1)), dim=2)
        output, (hidden, cell) = self.rnn(rnn_input, (hidden, cell))
        prediction = self.fc(output.squeeze(1))
        return prediction, hidden, cell, attn_weights
```

3.8 Finally we combine the 2 to make the Seq2Seq model

[13]:
```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, use_attention=False):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.use_attention = use_attention

    def forward(self, src, tgt, teacher_forcing_ratio=0.5):
        batch_size = src.shape[0]
        tgt_len = tgt.shape[1]
        tgt_vocab_size = self.decoder.output_dim

        outputs = torch.zeros(batch_size, tgt_len, tgt_vocab_size).to(device)
        encoder_outputs, hidden, cell = self.encoder(src)

        input = tgt[:, 0]
        for t in range(1, tgt_len):
            if self.use_attention:
                output, hidden, cell, _ = self.decoder(input, hidden, cell,␣
 ↪encoder_outputs)
            else:
                output, hidden, cell = self.decoder(input, hidden, cell)

            outputs[:, t] = output
            teacher_force = np.random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            input = tgt[:, t] if teacher_force else top1

        return outputs
```

[14]:
```
def train(model, src, tgt, optimizer, criterion, clip=1):
    model.train()
    optimizer.zero_grad()

    output = model(src, tgt)

    output_dim = output.shape[-1]
```

```
    output = output[:, 1:].reshape(-1, output_dim)
    tgt = tgt[:, 1:].reshape(-1)

    loss = criterion(output, tgt)
    loss.backward()

    torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
    optimizer.step()

    return loss.item()
```

[24]:
```
INPUT_DIM = len(src_vocab)
OUTPUT_DIM = len(tgt_vocab)
EMB_DIM = 256
HID_DIM = 512
```

[25]:
```
# Without Attention
enc_no_attn = Encoder(INPUT_DIM, EMB_DIM, HID_DIM).to(device)
dec_no_attn = DecoderNoAttention(OUTPUT_DIM, EMB_DIM, HID_DIM).to(device)
model_no_attn = Seq2Seq(enc_no_attn, dec_no_attn, use_attention=False).to(device)
```

[26]:
```
# With Attention
enc_attn = Encoder(INPUT_DIM, EMB_DIM, HID_DIM).to(device)
dec_attn = DecoderWithAttention(OUTPUT_DIM, EMB_DIM, HID_DIM).to(device)
model_attn = Seq2Seq(enc_attn, dec_attn, use_attention=True).to(device)
```

[27]:
```
optimizer_no_attn = optim.Adam(model_no_attn.parameters())
optimizer_attn = optim.Adam(model_attn.parameters())
criterion = nn.CrossEntropyLoss(ignore_index=tgt_vocab['<pad>'])
```

3.9 Training the model

[28]:
```
N_EPOCHS = 150
print("Training Model without Attention:")
for epoch in range(N_EPOCHS):
    loss = train(model_no_attn, train_src, train_tgt, optimizer_no_attn,␣
 ↪criterion)
    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss:.3f}')

print("\nTraining Model with Attention:")
for epoch in range(N_EPOCHS):
    loss = train(model_attn, train_src, train_tgt, optimizer_attn, criterion)
    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss:.3f}')
```

```
Training Model without Attention:
Epoch 0, Loss: 7.139
```

```
Epoch 10, Loss: 5.463
Epoch 20, Loss: 5.133
Epoch 30, Loss: 4.892
Epoch 40, Loss: 4.684
Epoch 50, Loss: 4.631
Epoch 60, Loss: 4.406
Epoch 70, Loss: 4.319
Epoch 80, Loss: 3.678
Epoch 90, Loss: 3.546
Epoch 100, Loss: 3.044
Epoch 110, Loss: 3.607
Epoch 120, Loss: 3.061
Epoch 130, Loss: 2.131
Epoch 140, Loss: 2.575
Epoch 150, Loss: 1.585
Epoch 160, Loss: 2.571
Epoch 170, Loss: 1.009
Epoch 180, Loss: 1.933
Epoch 190, Loss: 1.921

Training Model with Attention:
Epoch 0, Loss: 7.143
Epoch 10, Loss: 5.460
Epoch 20, Loss: 5.239
Epoch 30, Loss: 5.079
Epoch 40, Loss: 4.965
Epoch 50, Loss: 4.905
Epoch 60, Loss: 4.726
Epoch 70, Loss: 4.637
Epoch 80, Loss: 4.517
Epoch 90, Loss: 4.132
Epoch 100, Loss: 3.918
Epoch 110, Loss: 3.745
Epoch 120, Loss: 3.565
Epoch 130, Loss: 3.270
Epoch 140, Loss: 3.266
Epoch 150, Loss: 3.033
Epoch 160, Loss: 3.092
Epoch 170, Loss: 3.034
Epoch 180, Loss: 1.692
Epoch 190, Loss: 1.715
```

3.10 Inference function

3.10.1 In machine learning, inference refers to the process of using a trained model to make predictions or generate outputs for new, unseen data.

[31]:
```
def translate_sentence(model, sentence, src_vocab, tgt_vocab, inv_tgt_vocab,␣
↪max_len=50):
    model.eval()
    tokens = tokenize(sentence, src_vocab)
    src = torch.LongTensor(tokens).unsqueeze(0).to(device)

    with torch.no_grad():
        encoder_outputs, hidden, cell = model.encoder(src)

        tgt_tokens = [tgt_vocab['<sos>']]
        for _ in range(max_len):
            input = torch.LongTensor([tgt_tokens[-1]]).to(device)
            if model.use_attention:
                output, hidden, cell, _ = model.decoder(input, hidden, cell,␣
↪encoder_outputs)
            else:
                output, hidden, cell = model.decoder(input, hidden, cell)

            pred_token = output.argmax(1).item()
            tgt_tokens.append(pred_token)
            if pred_token == tgt_vocab['<eos>']:
                break

    return ' '.join(inv_tgt_vocab[token] for token in tgt_tokens[1:-1])
```

[36]:
```
test_sentence = "Hello World!"
print("Original:", test_sentence)
print("No Attention:", translate_sentence(model_no_attn, test_sentence,␣
↪src_vocab, tgt_vocab, inv_tgt_vocab))
print("With Attention:", translate_sentence(model_attn, test_sentence,␣
↪src_vocab, tgt_vocab, inv_tgt_vocab))
```

```
Original: Hello World!
No Attention: je le visse par moi-même.
With Attention: je te voir.
```

Lab Work 17
Transformer for Text Classification

1 Aim: To implement a transformer for sentiment analysis and text classification

2 Theory

The Transformer model, originally designed for sequence-to-sequence tasks, is adapted for text classification to categorize text sequences (e.g., reviews, tweets) into classes like positive/negative sentiment or topic labels. The objective is to leverage the Transformer's self-attention mechanism to capture complex dependencies in text, achieving high accuracy on datasets like IMDB or SST-2 by modeling contextual relationships effectively.

Transformer Architecture: The model processes an input sequence $\mathbf{x} = (x_1, \ldots, x_T)$, where $x_t \in \mathbb{R}^d$ is a word embedding (e.g., from pre-trained GloVe or BERT). The Transformer consists of stacked layers, each with:

- **Multi-Head Self-Attention**: Computes attention scores across tokens to model relationships:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V},$$

  where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{T \times d_k}$ are query, key, and value matrices derived from $\mathbf{x}$, and $d_k = d/h$ for h heads. Multi-head attention concatenates outputs from multiple attention heads, capturing diverse dependencies.

- **Feed-Forward Network (FFN)**: Applies a position-wise non-linear transformation:

$$\text{FFN}(\mathbf{z}) = \text{ReLU}(\mathbf{W}_1\mathbf{z} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

- **Normalization and Residual Connections**: Layer normalization and skip connections stabilize training:

$$\mathbf{z}' = \text{LayerNorm}(\mathbf{z} + \text{Attention}(\mathbf{z})), \quad \mathbf{z}'' = \text{LayerNorm}(\mathbf{z}' + \text{FFN}(\mathbf{z}')).$$

For classification, a special token (e.g., [CLS]) or pooled output (e.g., mean of final hidden states $\mathbf{H} \in \mathbb{R}^{T \times d}$) is fed to a fully connected layer:

$$\mathbf{y} = \text{softmax}(\mathbf{W}_y\mathbf{h}_{\text{pool}} + \mathbf{b}_y).$$

Advantages and Trade-offs: Transformers excel at capturing long-range dependencies and contextual nuances, outperforming RNNs in text classification due to parallel processing and attention. They achieve high accuracy with pre-trained models (e.g., BERT), but require significant computational resources and large datasets to avoid overfitting. Fine-tuning pre-trained Transformers is more effective than training from scratch for small datasets.

Training Dynamics: The model minimizes a cross-entropy loss for classification:

$$\mathcal{L} = -\sum_{i=1}^{C} t_i \log(y_i),$$

where t_i is the true label and y_i is the predicted probability. Training uses optimizers like Adam with learning rate schedules, leveraging pre-trained embeddings or models. Regularization (e.g., dropout) and data augmentation (e.g., synonym replacement) enhance generalization. Positional encodings ensure the model accounts for token order.

3 Code

[3]:
```
import torch
from transformers import DistilBertTokenizerFast,␣
 ↪DistilBertForSequenceClassification, Trainer, TrainingArguments
from datasets import load_dataset
import numpy as np
from sklearn.metrics import accuracy_score
```

[4]:
```
device = torch.device('cuda:0')
print(f'Using device: {device}')
```

```
Using device: cuda:0
```

[5]:
```
dataset = load_dataset("imdb")
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
README.md:   0%|          | 0.00/7.81k [00:00<?, ?B/s]
train-00000-of-00001.parquet:   0%|          | 0.00/21.0M [00:00<?, ?B/s]
test-00000-of-00001.parquet:   0%|          | 0.00/20.5M [00:00<?, ?B/s]
unsupervised-00000-of-00001.parquet:   0%|          | 0.00/42.0M [00:00<?, ?B/s]
Generating train split:   0%|          | 0/25000 [00:00<?, ? examples/s]
Generating test split:   0%|          | 0/25000 [00:00<?, ? examples/s]
Generating unsupervised split:   0%|          | 0/50000 [00:00<?, ? examples/s]
```

[6]:
```
dataset
```

[6]:
```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 25000
    })
    unsupervised: Dataset({
        features: ['text', 'label'],
        num_rows: 50000
    })
})
```

3.1 We need to tokenize the data so we write a class for that

[7]:
```
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
```

```
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
config.json:   0%|          | 0.00/483 [00:00<?, ?B/s]
```

[8]:
```
def tokenize_function(examples):
    return tokenizer(examples['text'], padding="max_length", truncation=True,␣
↪max_length=512)
```

[9]:
```
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

```
Map:   0%|          | 0/25000 [00:00<?, ? examples/s]
Map:   0%|          | 0/25000 [00:00<?, ? examples/s]
Map:   0%|          | 0/50000 [00:00<?, ? examples/s]
```

[10]:
```
tokenized_datasets.set_format('torch', columns=['input_ids', 'attention_mask',␣
↪'label'])
```

3.2 Loading the datasets

[11]:
```
train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(10000))
test_dataset = tokenized_datasets["test"].select(range(2000))
```

3.3 Now we will load the pre-trained BERT model

[12]:
```
model = DistilBertForSequenceClassification.
↪from_pretrained("distilbert-base-uncased")
```

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed.
Falling back to regular HTTP download. For better performance, install the
package with: `pip install huggingface_hub[hf_xet]` or `pip install hf_xet`
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but
the 'hf_xet' package is not installed. Falling back to regular HTTP download.
For better performance, install the package with: `pip install
huggingface_hub[hf_xet]` or `pip install hf_xet`
```

```
model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
```

```
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-uncased and are newly initialized:
['classifier.bias', 'classifier.weight', 'pre_classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[15]:
```
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=2,
    logging_dir="./logs",
    logging_steps=50,
    load_best_model_at_end=True,
    report_to="none",  # turn off wandb if not needed
)
```

[16]:
```
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return {"accuracy": accuracy_score(labels, predictions)}
```

[17]:
```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    compute_metrics=compute_metrics,
)
```

[18]:
```
trainer.train()
```

```
<IPython.core.display.HTML object>
```

[18]:
```
TrainOutput(global_step=2500, training_loss=0.2633450273513794,
metrics={'train_runtime': 1030.4865, 'train_samples_per_second': 19.408,
'train_steps_per_second': 2.426, 'total_flos': 2649347973120000.0, 'train_loss':
0.2633450273513794, 'epoch': 2.0})
```

[19]:
```
trainer.evaluate()
```

```
<IPython.core.display.HTML object>
```

[19]:
```
{'eval_loss': 0.31402429938316345,
 'eval_accuracy': 0.9215,
 'eval_runtime': 30.2815,
 'eval_samples_per_second': 66.047,
 'eval_steps_per_second': 8.256,
 'epoch': 2.0}
```

Lab Work 18
Time-Series Forecasting with LSTM

1 Aim: To implement univariate time-series prediction using LSTM

2 Theory

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed for time series forecasting, where the goal is to predict future values of a sequence $\mathbf{x} = (x_1, \ldots, x_T)$, with $x_t \in \mathbb{R}^d$ representing features at time t (e.g., stock prices, temperature). The objective is to model long-term temporal dependencies in sequential data, enabling accurate predictions for tasks like financial forecasting or energy consumption prediction.

LSTM Architecture: LSTMs process sequences by maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^h$ and a cell state $\mathbf{c}_t \in \mathbb{R}^h$, which preserves long-term information. For an input x_t, the LSTM updates are:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_f), \quad \mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_i),$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_o), \quad \tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, x_t] + \mathbf{b}_c),$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \quad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t),$$

where $\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t$ are forget, input, and output gates, σ is the sigmoid function, $\odot$ denotes element-wise multiplication, and $\mathbf{W}, \mathbf{b}$ are learnable parameters. The cell state $\mathbf{c}_t$ mitigates vanishing gradients, enabling LSTMs to capture long-term dependencies.

Mechanism for Forecasting: For forecasting, an LSTM takes a window of past data $(x_{t-W}, \ldots, x_t)$ to predict future values $\hat{x}_{t+1}$ or a sequence $\hat{x}_{t+1}, \ldots, \hat{x}_{t+k}$. The final hidden state $\mathbf{h}_T$ or a sequence of outputs is fed to a fully connected layer:

$$\hat{x}_{t+1} = \mathbf{W}_y \mathbf{h}_T + \mathbf{b}_y.$$

LSTMs model complex temporal patterns, such as seasonality or trends, by selectively remembering or forgetting past information.

Advantages and Trade-offs: LSTMs excel at capturing long-term dependencies, outperforming standard RNNs and simpler models (e.g., ARIMA) for non-linear time series. They handle variable-length sequences and multivariate inputs but are computationally intensive and sensitive to hyperparameter tuning (e.g., hidden size, learning rate). Compared to Transformers, LSTMs are less parallelizable but effective for smaller datasets or sequential tasks.

Training Dynamics: LSTMs minimize a loss function, typically Mean Squared Error (MSE) for regression:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (x_{i+1} - \hat{x}_{i+1})^2,$$

using backpropagation through time with optimizers like Adam. Preprocessing (e.g., normalization, windowing) and regularization (e.g., dropout) are critical for generalization. Gradient clipping stabilizes training for long sequences.

3 Code

3.1 Importing the libraries

[1]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

3.2 Loading the dataset

[44]:
```
df = pd.read_csv('./dl_lab_datasets/aqi_data/city_day.csv')

# Filter for one city, say Delhi
df = df[df['City'] == 'Delhi']

# Convert date and sort
df['Date'] = pd.to_datetime(df['Date'])
df = df.sort_values('Date')

# Focus on PM2.5 (you can choose others like PM10, NO2, etc.)
df = df[['Date', 'PM2.5']].dropna()

# Reset index
df.reset_index(drop=True, inplace=True)
```

3.3 We will have to create Sequences to input into the LSTM model

[36]:
```
from sklearn.preprocessing import MinMaxScaler
import torch

scaler = MinMaxScaler()
pm_values = scaler.fit_transform(df[['PM2.5']].values)

# Sequence generation
def create_sequences(data, seq_len):
    xs, ys = [], []
    for i in range(len(data) - seq_len):
        x = data[i:(i + seq_len)]
        y = data[i + seq_len]
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)
```

```
SEQ_LEN = 90
X, y = create_sequences(data_scaled, SEQ_LEN)

# Train-test split (e.g., 80%)
train_size = int(0.8 * len(X))
X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]

# To PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
```

3.4 Define the LSTM and MLP model

[37]:
```
class LSTMModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=128, num_layers=2):  # was 32, 1
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = out[:, -1, :]
        return self.fc(out)
```

[38]:
```
class DenseModel(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten
        return self.fc(x)
```

[39]:
```
def train_model(model, X_train, y_train, X_test, y_test, epochs=50, lr=0.001):
    import torch.optim as optim

    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
```

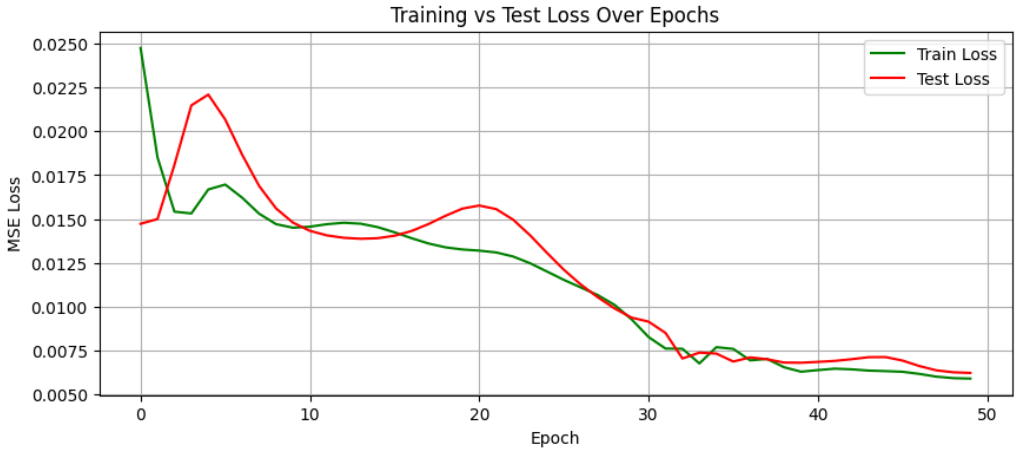
```
    train_losses, test_losses = [], []

    for epoch in range(epochs):
        model.train()
        optimizer.zero_grad()
        output = model(X_train)
        loss = criterion(output, y_train)
        loss.backward()
        optimizer.step()

        model.eval()
        with torch.no_grad():
            test_output = model(X_test)
            test_loss = criterion(test_output, y_test)

        train_losses.append(loss.item())
        test_losses.append(test_loss.item())

        print(f"Epoch {epoch+1}: Train Loss = {loss.item():.4f}, Test Loss =␣
↪{test_loss.item():.4f}")

    return train_losses, test_losses, test_output
```

[40]:
```
lstm_model = LSTMModel()
lstm_train_loss, lstm_test_loss, lstm_preds = train_model(
    lstm_model, X_train, y_train, X_test, y_test
)
```

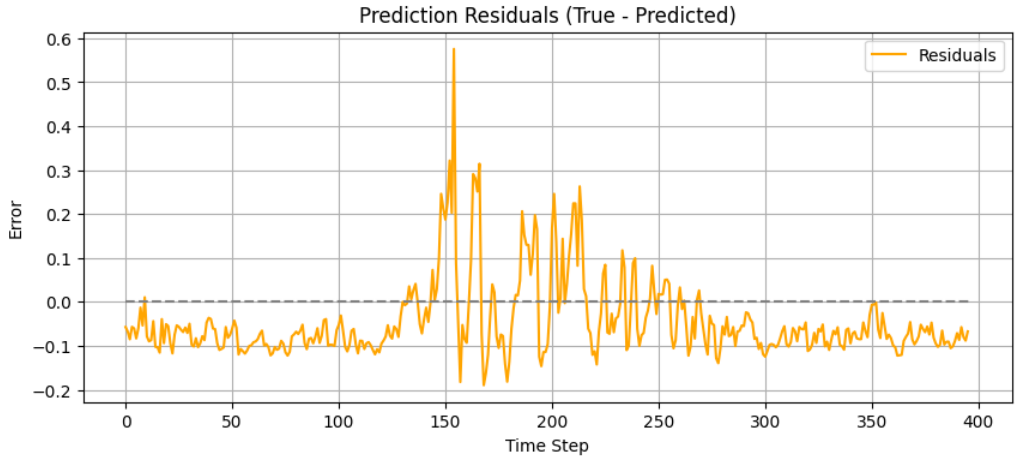
```
Epoch 1: Train Loss = 0.0248, Test Loss = 0.0147
Epoch 2: Train Loss = 0.0185, Test Loss = 0.0150
Epoch 3: Train Loss = 0.0154, Test Loss = 0.0181
Epoch 4: Train Loss = 0.0153, Test Loss = 0.0215
Epoch 5: Train Loss = 0.0167, Test Loss = 0.0221
Epoch 6: Train Loss = 0.0170, Test Loss = 0.0207
Epoch 7: Train Loss = 0.0162, Test Loss = 0.0187
Epoch 8: Train Loss = 0.0153, Test Loss = 0.0169
Epoch 9: Train Loss = 0.0147, Test Loss = 0.0156
Epoch 10: Train Loss = 0.0145, Test Loss = 0.0148
Epoch 11: Train Loss = 0.0146, Test Loss = 0.0143
Epoch 12: Train Loss = 0.0147, Test Loss = 0.0141
Epoch 13: Train Loss = 0.0148, Test Loss = 0.0139
Epoch 14: Train Loss = 0.0147, Test Loss = 0.0139
Epoch 15: Train Loss = 0.0145, Test Loss = 0.0139
Epoch 16: Train Loss = 0.0142, Test Loss = 0.0140
Epoch 17: Train Loss = 0.0139, Test Loss = 0.0143
Epoch 18: Train Loss = 0.0136, Test Loss = 0.0147
Epoch 19: Train Loss = 0.0134, Test Loss = 0.0152
Epoch 20: Train Loss = 0.0133, Test Loss = 0.0156
```

```
Epoch 21: Train Loss = 0.0132, Test Loss = 0.0158
Epoch 22: Train Loss = 0.0131, Test Loss = 0.0156
Epoch 23: Train Loss = 0.0129, Test Loss = 0.0150
Epoch 24: Train Loss = 0.0125, Test Loss = 0.0141
Epoch 25: Train Loss = 0.0120, Test Loss = 0.0131
Epoch 26: Train Loss = 0.0115, Test Loss = 0.0121
Epoch 27: Train Loss = 0.0111, Test Loss = 0.0113
Epoch 28: Train Loss = 0.0107, Test Loss = 0.0105
Epoch 29: Train Loss = 0.0101, Test Loss = 0.0099
Epoch 30: Train Loss = 0.0093, Test Loss = 0.0094
Epoch 31: Train Loss = 0.0083, Test Loss = 0.0091
Epoch 32: Train Loss = 0.0076, Test Loss = 0.0085
Epoch 33: Train Loss = 0.0076, Test Loss = 0.0070
Epoch 34: Train Loss = 0.0068, Test Loss = 0.0074
Epoch 35: Train Loss = 0.0077, Test Loss = 0.0073
Epoch 36: Train Loss = 0.0076, Test Loss = 0.0069
Epoch 37: Train Loss = 0.0069, Test Loss = 0.0071
Epoch 38: Train Loss = 0.0070, Test Loss = 0.0070
Epoch 39: Train Loss = 0.0065, Test Loss = 0.0068
Epoch 40: Train Loss = 0.0063, Test Loss = 0.0068
Epoch 41: Train Loss = 0.0064, Test Loss = 0.0069
Epoch 42: Train Loss = 0.0065, Test Loss = 0.0069
Epoch 43: Train Loss = 0.0064, Test Loss = 0.0070
Epoch 44: Train Loss = 0.0064, Test Loss = 0.0071
Epoch 45: Train Loss = 0.0063, Test Loss = 0.0071
Epoch 46: Train Loss = 0.0063, Test Loss = 0.0069
Epoch 47: Train Loss = 0.0062, Test Loss = 0.0066
Epoch 48: Train Loss = 0.0060, Test Loss = 0.0064
Epoch 49: Train Loss = 0.0059, Test Loss = 0.0063
Epoch 50: Train Loss = 0.0059, Test Loss = 0.0062
```

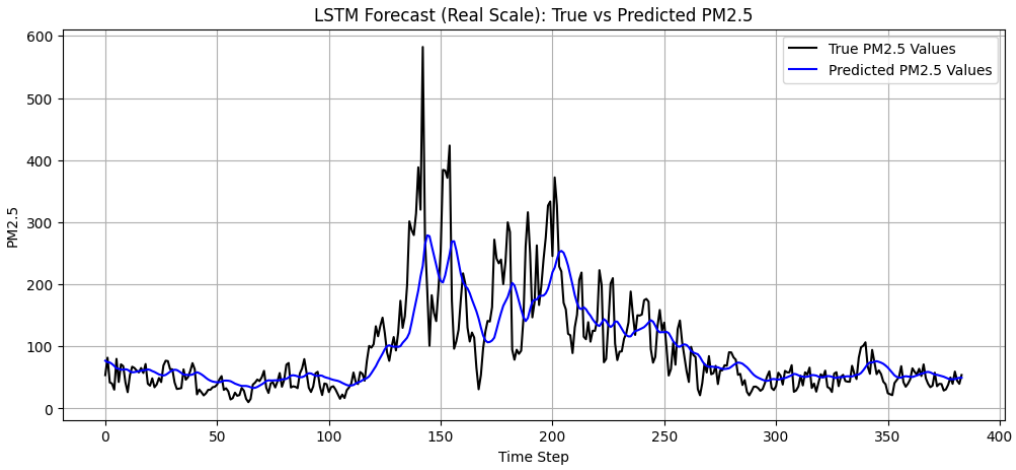
[41]:
```python
plt.figure(figsize=(10, 4))
plt.plot(lstm_train_loss, label="Train Loss", color="green")
plt.plot(lstm_test_loss, label="Test Loss", color="red")
plt.title("Training vs Test Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.legend()
plt.grid(True)
plt.show()
```



[42]:
```
residuals = true_vals - pred_vals

plt.figure(figsize=(10, 4))
plt.plot(residuals, label="Residuals", color="orange")
plt.hlines(0, 0, len(residuals), colors='gray', linestyles='dashed')
plt.title("Prediction Residuals (True - Predicted)")
plt.xlabel("Time Step")
plt.ylabel("Error")
plt.legend()
plt.grid(True)
plt.show()
```



[43]:
```
import matplotlib.pyplot as plt
true_vals = y_test.detach().cpu().numpy()
pred_vals = lstm_preds.detach().cpu().numpy()
pred_real_scale = scaler.inverse_transform(lstm_preds.reshape(-1, 1))
true_real_scale = scaler.inverse_transform(y_test.reshape(-1, 1))


plt.figure(figsize=(12, 5))
plt.plot(true_real_scale, label="True PM2.5 Values", color="black")
plt.plot(pred_real_scale, label="Predicted PM2.5 Values", color="blue")
plt.title("LSTM Forecast (Real Scale): True vs Predicted PM2.5")
plt.xlabel("Time Step")
plt.ylabel("PM2.5")
plt.legend()
plt.grid(True)
plt.show()
```



Lab Work 19
Image Segmentation using UNet

1 Aim: Perform image segmentation using UNet

2 Theory

U-Net is a Convolutional Neural Network (CNN) architecture designed for image segmentation, particularly effective in tasks like medical image segmentation (e.g., MRI scans) and semantic segmentation (e.g., autonomous driving). The objective is to assign a class label to each pixel in an input image $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$, producing a segmentation map $\mathbf{Y} \in \mathbb{R}^{H \times W \times K}$, where K is the number of classes (e.g., tumor vs. background), preserving spatial details for precise localization.

U-Net Architecture: U-Net features a symmetric encoder-decoder structure with skip connections. The encoder downsamples the input to extract hierarchical features:

$$\mathbf{Z}_l = \text{ReLU}(\text{Conv}_{3\times3}(\text{MaxPool}(\mathbf{Z}_{l-1}))),$$

where $\text{Conv}_{3\times3}$ denotes 3×3 convolutions, and MaxPool reduces spatial dimensions. The decoder upsamples features to recover spatial resolution:

$$\mathbf{Z}'_l = \text{ReLU}(\text{Conv}_{3\times3}(\text{UpConv}(\mathbf{Z}'_{l+1}) \oplus \mathbf{Z}_{l-\text{skip}})),$$

where UpConv is an upsampling operation (e.g., transposed convolution), and $\oplus$ denotes concatenation with skip connections from the encoder's corresponding layer $\mathbf{Z}_{l-\text{skip}}$. Skip connections preserve fine-grained details by combining low-level (spatial) and high-level (semantic) features. The final layer uses a 1×1 convolution with softmax or sigmoid to produce pixel-wise class probabilities:

$$\mathbf{Y} = \text{softmax}(\text{Conv}_{1\times1}(\mathbf{Z}'_1)).$$

Segmentation Mechanism: U-Net processes the input image through the encoder to capture context (e.g., organ shapes), then the decoder reconstructs a high-resolution segmentation map. Skip connections ensure precise localization by integrating detailed encoder features, critical for delineating boundaries (e.g., tumor edges). The model outputs a per-pixel probability map, assigning each pixel to a class based on the highest probability.

Advantages and Trade-offs: U-Net excels in segmentation with limited data, particularly in medical imaging, due to its efficient architecture and skip connections. It achieves high precision in boundary delineation but requires significant computational resources for deep networks. Compared to other models (e.g., FCNs), U-Net's symmetry and skip connections enhance performance on small datasets, though it may struggle with very high-resolution images without modifications.

Training Dynamics: U-Net minimizes a loss function, typically a combination of cross-entropy and Dice loss for segmentation:

$$\mathcal{L} = -\sum_{h,w,k} y_{h,w,k} \log(\hat{y}_{h,w,k}) + 1 - \frac{2\sum y_{h,w,k}\hat{y}_{h,w,k}}{\sum y_{h,w,k} + \sum \hat{y}_{h,w,k}},$$

where $y_{h,w,k}$ and $\hat{y}_{h,w,k}$ are ground-truth and predicted probabilities. Training uses optimizers like Adam, with data augmentation (e.g., rotations, flips) and regularization (e.g., dropout) to improve generalization. Pixel-wise annotations are required, often labor-intensive to obtain.

3 Code

3.1 Importing the libraries

[16]:
```
import os
import torch
import numpy as np
from PIL import Image
from torch.utils.data import Dataset, DataLoader
import albumentations as A
from albumentations.pytorch import ToTensorV2
import matplotlib.pyplot as plt
import segmentation_models_pytorch as smp
import torch.optim as optim
```

3.2 Let's define the dataset

[3]:
```
class WaterBodyDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.images = sorted(os.listdir(image_dir))  # Ensure matching order

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_dir, self.images[idx])
        mask_path = os.path.join(self.mask_dir, self.images[idx])  # Assuming␣
↪same name

        image = np.array(Image.open(img_path).convert("RGB"))
        mask = np.array(Image.open(mask_path).convert("L"))  # Grayscale

        mask = (mask > 127).astype("float32")  # Binarize mask

        if self.transform:
            augmented = self.transform(image=image, mask=mask)
            image = augmented["image"]
            mask = augmented["mask"]

        return image, mask.unsqueeze(0)  # Add channel to mask
```

[9]:
```
transform = A.Compose([
    A.Resize(256, 256),
    A.Normalize(mean=(0.5,), std=(0.5,)),
    ToTensorV2(),
```

```
])

train_dataset = WaterBodyDataset(
    image_dir='/content/Water Bodies Dataset/Images',
    mask_dir='/content/Water Bodies Dataset/Masks',
    transform=transform
)

train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
```

3.3 Now we define the UNet model

[10]:
```
model = smp.Unet(
    encoder_name="resnet34",
    encoder_weights="imagenet",
    in_channels=3,
    classes=1,
    activation=None
).cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet34-333f7ec4.pth" to
/root/.cache/torch/hub/checkpoints/resnet34-333f7ec4.pth
100%|| 83.3M/83.3M [00:00<00:00, 246MB/s]
```

3.4 Defining the Dice Loss and IoU metrics

[11]:
```
def dice_loss(pred, target, smooth=1.):
    pred = torch.sigmoid(pred)
    pred = pred.view(-1)
    target = target.view(-1)
    intersection = (pred * target).sum()
    return 1 - ((2. * intersection + smooth) / (pred.sum() + target.sum() +␣
 ↪smooth))

def iou_score(pred, target, threshold=0.5):
    pred = torch.sigmoid(pred) > threshold
    target = target > 0.5
    intersection = (pred & target).float().sum((1, 2, 3))
    union = (pred | target).float().sum((1, 2, 3))
    return (intersection + 1e-6) / (union + 1e-6)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
```

[12]:
```
def train_model(model, dataloader, epochs=10):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

```
    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for images, masks in dataloader:
            images, masks = images.to(device), masks.to(device)
            preds = model(images)
            loss = dice_loss(preds, masks)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        print(f"Epoch [{epoch+1}/{epochs}], Loss: {total_loss/len(dataloader):.
↪4f}")
```

3.5 Finally we visualise the results

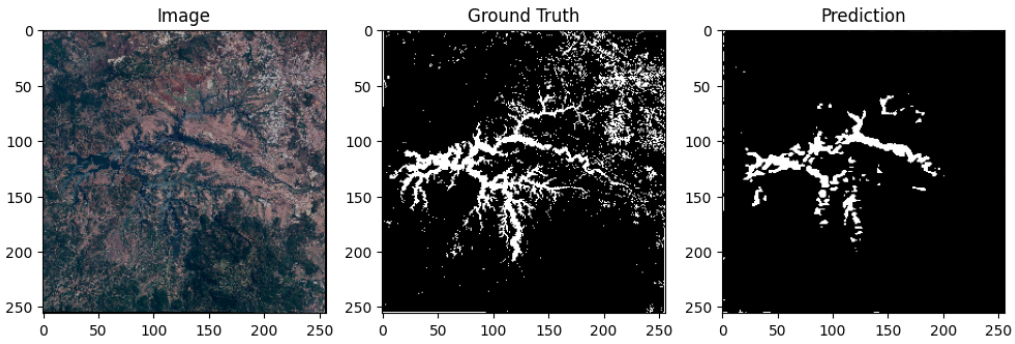
[19]:
```
def unnormalize(img):
    # img is a tensor with shape [3, H, W] and range [-1, 1]
    img = img * 0.5 + 0.5  # Scale from [-1,1] → [0,1]
    return img

def show_prediction(sample_image, sample_mask, prediction):
    sample_image = unnormalize(sample_image).permute(1, 2, 0).cpu().numpy()
    sample_mask = sample_mask.squeeze().cpu().numpy()
    prediction = torch.sigmoid(prediction).squeeze().cpu().numpy()

    fig, axs = plt.subplots(1, 3, figsize=(12, 4))
    axs[0].imshow(sample_image)
    axs[0].set_title("Image")
    axs[1].imshow(sample_mask, cmap="gray")
    axs[1].set_title("Ground Truth")
    axs[2].imshow(prediction > 0.5, cmap="gray")
    axs[2].set_title("Prediction")
    plt.show()
```

[21]:
```
train_model(model, train_loader)
```

```
Epoch [1/10], Loss: 0.1883
Epoch [2/10], Loss: 0.1920
Epoch [3/10], Loss: 0.1857
Epoch [4/10], Loss: 0.1832
Epoch [5/10], Loss: 0.1807
Epoch [6/10], Loss: 0.1802
Epoch [7/10], Loss: 0.1721
Epoch [8/10], Loss: 0.1683
Epoch [9/10], Loss: 0.1715
```

```
Epoch [10/10], Loss: 0.1719
```

[23]:
```
model.eval()

sample_img, sample_mask = train_dataset[0]
sample_img = sample_img.cuda().unsqueeze(0)

with torch.no_grad():
    pred_mask = model(sample_img)

# Visualize it
show_prediction(sample_img.squeeze(0).cpu(), sample_mask, pred_mask.squeeze(0).
↪cpu())
```



Lab Work 20
General Adversarial Networks for Image Generation

1 Aim: To train a GAN to generate realistic images

2 Theory

Generative Adversarial Networks (GANs) are a class of deep learning models designed to generate realistic images, used in applications like synthetic face generation, art creation, and data augmentation. The objective is to learn a generative model that produces images $\hat{\mathbf{X}} \in \mathbb{R}^{H \times W \times C}$ indistinguishable from real images $\mathbf{X}$ by training two competing networks: a generator and a discriminator, leveraging an adversarial process to approximate the true data distribution.

GAN Architecture: The generator G maps a random noise vector $\mathbf{z} \in \mathbb{R}^d$ (e.g., sampled from a Gaussian distribution) to a synthetic image:

$$\hat{\mathbf{X}} = G(\mathbf{z}; \theta_G),$$

where θ_G are generator parameters, typically implemented as a deep convolutional network with upsampling layers (e.g., transposed convolutions). The discriminator D evaluates whether an image is real or fake, outputting a probability:

$$p = D(\mathbf{X} \text{ or } \hat{\mathbf{X}}; \theta_D),$$

where θ_D are discriminator parameters, often a CNN with downsampling layers. The discriminator aims to maximize p for real images and minimize it for generated ones.

Training Mechanism: GANs are trained via a minimax game, optimizing a loss function:

$$\min_G \max_D \mathbb{E}_{\mathbf{X} \sim p_{\text{data}}}[\log D(\mathbf{X})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))],$$

where p_{data} is the real data distribution, and $p_{\mathbf{z}}$ is the noise distribution. The discriminator maximizes the probability of correctly classifying real vs. fake images, while the generator minimizes the probability that its outputs are classified as fake. In practice, the generator optimizes a non-saturating loss, $\mathbb{E}_{\mathbf{z}}[-\log D(G(\mathbf{z}))]$, for better gradient flow. Training alternates updates to G and D, using optimizers like Adam.

Advantages and Challenges: GANs generate high-quality, diverse images, capturing complex data distributions without explicit density estimation. They excel in tasks requiring realism (e.g., faces, landscapes) but face challenges like mode collapse (limited output diversity), training instability, and high computational cost. Variants like DCGANs or StyleGAN address these issues with architectural improvements (e.g., convolutional layers, progressive growing).

Training Dynamics: GANs require careful balancing of generator and discriminator performance to avoid one overpowering the other. Hyperparameters (e.g., learning rate, noise dimension) and regularization (e.g., label smoothing) stabilize training. Preprocessing (e.g., image normalization) and large datasets (e.g., CelebA) enhance output quality. Evaluation metrics like Fréchet Inception Distance (FID) quantify realism.

3 Code

3.1 Loading all the libraries

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import save_image
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from torchvision.datasets import FashionMNIST
import os
```

[2]:
```
device = torch.device("mps")
latent_dim = 100
hidden_dim = 128  # Reduced for smaller images
image_dim = 28 * 28 * 1  # 28x28 grayscale
num_epochs = 50
batch_size = 128
lr = 0.0002
beta1 = 0.5
```

[3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Grayscale: single channel
])
```

[4]:
```
dataset = FashionMNIST(root='./dl_lab_datasets/', train=True,␣
↪transform=transform, download=True)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
↪num_workers=2)
```

3.2 First create a Generator class

[5]:
```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim * 2),
            nn.ReLU(),
            nn.Linear(hidden_dim * 2, hidden_dim * 4),
            nn.ReLU(),
            nn.Linear(hidden_dim * 4, image_dim),
            nn.Tanh()  # Output range [-1, 1]
```

```
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 1, 28, 28)  # Reshape to (batch, C, H, W)
        return img
```

3.3 Now the discriminator class

[6]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(image_dim, hidden_dim * 4),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 4, hidden_dim * 2),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()  # Output probability
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)  # Flatten to (batch, image_dim)
        validity = self.model(img_flat)
        return validity
```

[7]:
```
generator = Generator().to(device)
discriminator = Discriminator().to(device)
```

3.4 Defining the loss function

[9]:
```
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, 0.999))
```

[10]:
```
d_losses = []
g_losses = []
```

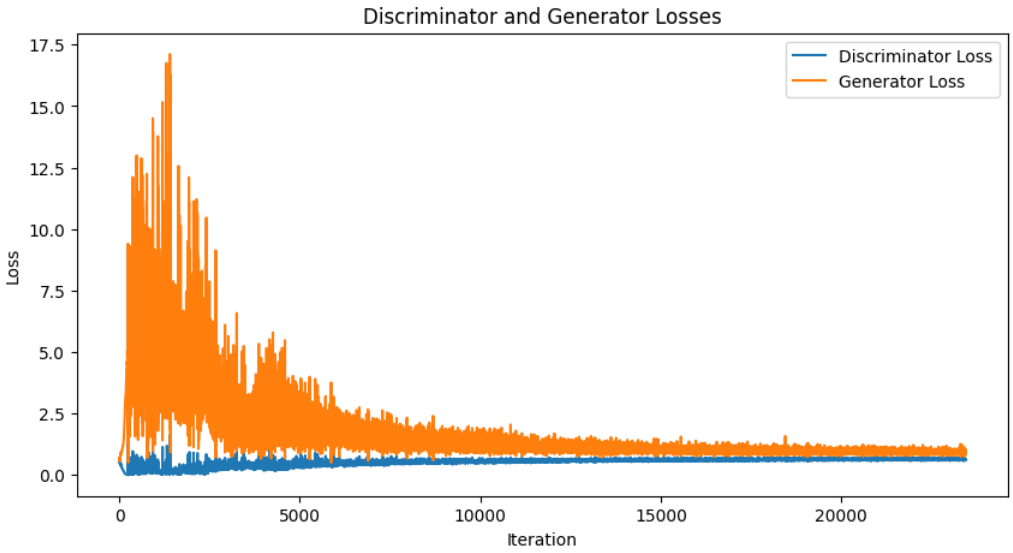
[11]:
```
os.makedirs("images", exist_ok=True)

for epoch in range(num_epochs):
    for i, (imgs, _) in enumerate(dataloader):
        batch_size = imgs.size(0)
        real_label = torch.ones(batch_size, 1).to(device)
        fake_label = torch.zeros(batch_size, 1).to(device)
```

```
        # Train Discriminator
        optimizer_D.zero_grad()
        real_imgs = imgs.to(device)
        real_loss = adversarial_loss(discriminator(real_imgs), real_label)

        z = torch.randn(batch_size, latent_dim).to(device)
        fake_imgs = generator(z)
        fake_loss = adversarial_loss(discriminator(fake_imgs.detach()),␣
↪fake_label)

        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
        g_loss = adversarial_loss(discriminator(fake_imgs), real_label)  # Trick␣
↪discriminator
        g_loss.backward()
        optimizer_G.step()

        # Store losses
        d_losses.append(d_loss.item())
        g_losses.append(g_loss.item())

        if i % 100 == 0:
            print(f"[Epoch {epoch}/{num_epochs}] [Batch {i}/{len(dataloader)}] "
                  f"D Loss: {d_loss.item():.4f}, G Loss: {g_loss.item():.4f}")

    # Save generated images every 10 epochs
    if epoch % 10 == 0:
        with torch.no_grad():
            fake_imgs = generator(torch.randn(16, latent_dim).to(device))
            fake_imgs = (fake_imgs + 1) / 2  # Rescale to [0, 1]
            save_image(fake_imgs, f"images/epoch_{epoch}.png", nrow=4,␣
↪normalize=True)
```

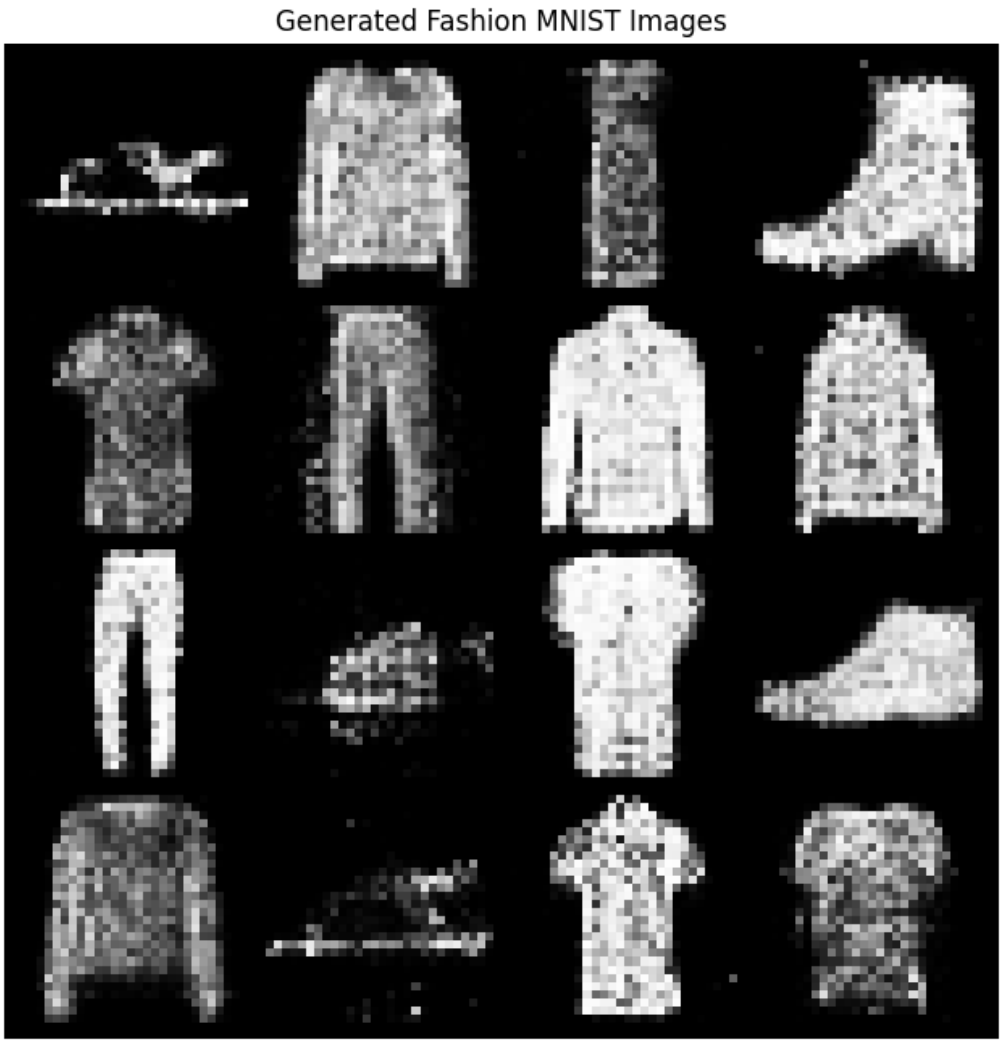
3.5 Let's plot the results

[12]:
```
plt.figure(figsize=(10, 5))
plt.plot(d_losses, label="Discriminator Loss")
plt.plot(g_losses, label="Generator Loss")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.title("Discriminator and Generator Losses")
```

```
plt.savefig("losses.png")
plt.show()
```



[13]:
```
with torch.no_grad():
    fake_imgs = generator(torch.randn(16, latent_dim).to(device))
    fake_imgs = (fake_imgs + 1) / 2
    grid = torchvision.utils.make_grid(fake_imgs, nrow=4, normalize=True)
    plt.figure(figsize=(8, 8))
    plt.imshow(grid.permute(1, 2, 0).cpu().numpy(), cmap="gray")
    plt.axis("off")
    plt.title("Generated Fashion MNIST Images")
    plt.savefig("final_images.png")
    plt.show()
```


Generated Fashion MNIST Images

Lab Work 21
CycleGAN for Image Generation

1 Aim: To translate image styles using GANs

2 Theory

Cycle-Consistent Generative Adversarial Networks (CycleGANs) are designed for unpaired image-to-image transfer, enabling transformation between two domains (e.g., horses to zebras, summer to winter landscapes) without paired training data. The objective is to learn mappings between domains $\mathcal{X}$ and $\mathcal{Y}$, producing images $\hat{\mathbf{Y}} \in \mathcal{Y}$ from $\mathbf{X} \in \mathcal{X}$ and vice versa, while preserving content and aligning styles across domains.

CycleGAN Architecture: CycleGAN employs two generators and two discriminators. Generator $G : \mathcal{X} \to \mathcal{Y}$ transforms images from domain $\mathcal{X}$ to $\mathcal{Y}$ (e.g., horse to zebra), and $F : \mathcal{Y} \to \mathcal{X}$ performs the reverse mapping:

$$\hat{\mathbf{Y}} = G(\mathbf{X}), \quad \hat{\mathbf{X}} = F(\mathbf{Y}).$$

Each generator is typically a deep convolutional network with encoder-decoder structure (e.g., ResNet-based). Discriminators D_X and D_Y distinguish real images $(\mathbf{X}, \mathbf{Y})$ from fake ones $(\hat{\mathbf{X}}, \hat{\mathbf{Y}})$:

$$p_X = D_X(\mathbf{X} \text{ or } \hat{\mathbf{X}}), \quad p_Y = D_Y(\mathbf{Y} \text{ or } \hat{\mathbf{Y}}).$$

The key innovation is cycle consistency, ensuring that mappings are reversible: $F(G(\mathbf{X})) \approx \mathbf{X}$ and $G(F(\mathbf{Y})) \approx \mathbf{Y}$.

Training Mechanism: CycleGAN optimizes a combined loss:

$$\mathcal{L} = \mathcal{L}_{\text{GAN}}(G, D_Y, \mathcal{X}, \mathcal{Y}) + \mathcal{L}_{\text{GAN}}(F, D_X, \mathcal{Y}, \mathcal{X}) + \lambda \mathcal{L}_{\text{cyc}}(G, F),$$

where the GAN loss is:

$$\mathcal{L}_{\text{GAN}}(G, D_Y, \mathcal{X}, \mathcal{Y}) = \mathbb{E}_{\mathbf{Y} \sim \mathcal{Y}}[\log D_Y(\mathbf{Y})] + \mathbb{E}_{\mathbf{X} \sim \mathcal{X}}[\log(1 - D_Y(G(\mathbf{X})))],$$

and the cycle-consistency loss enforces reversibility:

$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{\mathbf{X} \sim \mathcal{X}}[\|\mathbf{X} - F(G(\mathbf{X}))\|_1] + \mathbb{E}_{\mathbf{Y} \sim \mathcal{Y}}[\|\mathbf{Y} - G(F(\mathbf{Y}))\|_1],$$

with λ (e.g., 10) balancing terms. An optional identity loss, $\mathcal{L}_{\text{id}}(G, \mathbf{Y}) = \|\mathbf{Y} - G(\mathbf{Y})\|_1$, stabilizes training. Training alternates updates to generators and discriminators using optimizers like Adam.

Advantages and Challenges: CycleGAN enables unpaired image transfer, making it versatile for tasks where paired data is scarce (e.g., art style transfer). It preserves content while adapting style, but training is unstable, prone to mode collapse, and computationally intensive. The quality of transferred images depends on domain similarity and dataset size. Variants like StarGAN extend CycleGAN to multi-domain transfer.

Training Dynamics: CycleGAN requires large, diverse datasets (e.g., horse and zebra images) and careful hyperparameter tuning (e.g., learning rate, λ). Preprocessing (e.g., normalization, resizing) ensures consistent input formats. Evaluation uses metrics like Fréchet Inception Distance (FID) or visual inspection, as ground-truth pairs are unavailable.

3 Code

3.1 Importing all the libraries

[1]:
```
import os
import itertools
from PIL import Image
import torch
from torch import nn
import torchvision.transforms as T
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
```

[15]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving horse2zebra.zip to horse2zebra.zip
```

[17]:
```
import zipfile
import os

zip_filename = next(iter(uploaded))  # Automatically picks uploaded file
with zipfile.ZipFile(zip_filename, 'r') as zip_ref:
    zip_ref.extractall('/content/data')

# Check if structure is correct (i.e., data/horse2zebra/trainA/)
os.listdir('/content/data')
```

[17]:
```
['__MACOSX', 'horse2zebra']
```

[18]:
```
device = torch.device("cuda")
```

3.2 We will have to create the Horse2Zebra dataset loader

[19]:
```
class ImageDataset(Dataset):
    def __init__(self, root, transforms=None, mode='train'):
        self.transform = transforms
        self.files_A = sorted(os.listdir(os.path.join(root, f'{mode}A')))
        self.files_B = sorted(os.listdir(os.path.join(root, f'{mode}B')))
        self.root = root
        self.mode = mode

    def __getitem__(self, index):
        path_A = os.path.join(self.root, f'{self.mode}A', self.files_A[index %␣
 ↪len(self.files_A)])
```

```
        path_B = os.path.join(self.root, f'{self.mode}B', self.files_B[index %␣
↪len(self.files_B)])

        img_A = Image.open(path_A).convert("RGB")
        img_B = Image.open(path_B).convert("RGB")

        if self.transform:
            img_A = self.transform(img_A)
            img_B = self.transform(img_B)

        return {"A": img_A, "B": img_B}

    def __len__(self):
        return max(len(self.files_A), len(self.files_B))
```

[20]:
```
transform = T.Compose([
    T.Resize(256),
    T.CenterCrop(256),
    T.ToTensor(),
    T.Normalize((0.5,), (0.5,))
])
```

3.3 Defining the Generator

[8]:
```
class ResnetBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.block = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(dim, dim, 3),
            nn.InstanceNorm2d(dim),
            nn.ReLU(True),
            nn.ReflectionPad2d(1),
            nn.Conv2d(dim, dim, 3),
            nn.InstanceNorm2d(dim),
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):
    def __init__(self, input_nc, output_nc, n_residual_blocks=9):
        super().__init__()
        model = [
            nn.ReflectionPad2d(3),
            nn.Conv2d(input_nc, 64, 7),
            nn.InstanceNorm2d(64),
```

```
            nn.ReLU(True)
        ]

        # Downsampling
        in_features = 64
        out_features = in_features * 2
        for _ in range(2):
            model += [
                nn.Conv2d(in_features, out_features, 3, stride=2, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(True)
            ]
            in_features = out_features
            out_features *= 2

        # Residual blocks
        for _ in range(n_residual_blocks):
            model += [ResnetBlock(in_features)]

        # Upsampling
        out_features = in_features // 2
        for _ in range(2):
            model += [
                nn.ConvTranspose2d(in_features, out_features, 3, stride=2,␣
↪padding=1, output_padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(True)
            ]
            in_features = out_features
            out_features = out_features // 2

        model += [nn.ReflectionPad2d(3), nn.Conv2d(64, output_nc, 7), nn.Tanh()]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

3.4 Defining the discriminator

[9]:
```
class Discriminator(nn.Module):
    def __init__(self, input_nc):
        super().__init__()
        model = [
            nn.Conv2d(input_nc, 64, 4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True)
        ]
```

```
        in_features = 64
        out_features = in_features * 2
        for _ in range(3):
            model += [
                nn.Conv2d(in_features, out_features, 4, stride=2, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.LeakyReLU(0.2, inplace=True)
            ]
            in_features = out_features
            out_features *= 2

        model += [nn.Conv2d(in_features, 1, 4, padding=1)]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

[21]:
```
def init_weights(net):
    for m in net.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
            nn.init.normal_(m.weight.data, 0.0, 0.02)

netG_A2B = Generator(3, 3).to(device)
netG_B2A = Generator(3, 3).to(device)
netD_A = Discriminator(3).to(device)
netD_B = Discriminator(3).to(device)

init_weights(netG_A2B)
init_weights(netG_B2A)
init_weights(netD_A)
init_weights(netD_B)

criterion_GAN = nn.MSELoss()
criterion_cycle = nn.L1Loss()
criterion_identity = nn.L1Loss()

optimizer_G = torch.optim.Adam(itertools.chain(netG_A2B.parameters(), netG_B2A.
 ↪parameters()), lr=0.0002, betas=(0.5, 0.999))
optimizer_D_A = torch.optim.Adam(netD_A.parameters(), lr=0.0002, betas=(0.5, 0.
 ↪999))
optimizer_D_B = torch.optim.Adam(netD_B.parameters(), lr=0.0002, betas=(0.5, 0.
 ↪999))
```

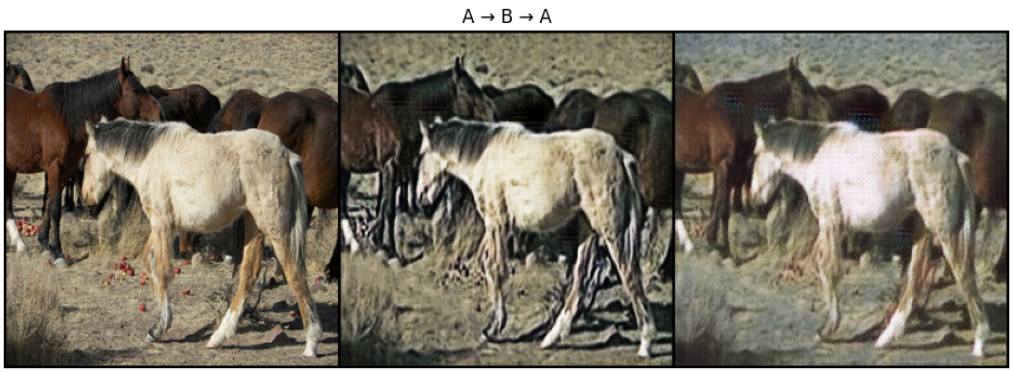
[23]:
```
from tqdm import tqdm

real_label = 1.0
fake_label = 0.0
```

```
dataloader = DataLoader(ImageDataset("/content/data/horse2zebra", transform,␣
↪mode="train"), batch_size=1, shuffle=True)

for epoch in range(5):
    for i, batch in enumerate(tqdm(dataloader)):

        real_A = batch['A'].to(device)
        real_B = batch['B'].to(device)

        ###### Train Generators ######
        optimizer_G.zero_grad()

        fake_B = netG_A2B(real_A)
        fake_A = netG_B2A(real_B)

        rec_A = netG_B2A(fake_B)
        rec_B = netG_A2B(fake_A)

        idt_A = netG_B2A(real_A)
        idt_B = netG_A2B(real_B)

        loss_idt = criterion_identity(idt_A, real_A) + criterion_identity(idt_B,␣
↪real_B)
        loss_GAN_A2B = criterion_GAN(netD_B(fake_B), torch.
↪ones_like(netD_B(fake_B)))
        loss_GAN_B2A = criterion_GAN(netD_A(fake_A), torch.
↪ones_like(netD_A(fake_A)))
        loss_cycle = criterion_cycle(rec_A, real_A) + criterion_cycle(rec_B,␣
↪real_B)

        loss_G = loss_GAN_A2B + loss_GAN_B2A + 10 * loss_cycle + 5 * loss_idt
        loss_G.backward()
        optimizer_G.step()

        ###### Train Discriminator A ######
        optimizer_D_A.zero_grad()
        loss_D_A = (
            criterion_GAN(netD_A(real_A), torch.ones_like(netD_A(real_A))) +
            criterion_GAN(netD_A(fake_A.detach()), torch.
↪zeros_like(netD_A(fake_A)))
        ) * 0.5
        loss_D_A.backward()
        optimizer_D_A.step()

        ###### Train Discriminator B ######
```

```
        optimizer_D_B.zero_grad()
        loss_D_B = (
            criterion_GAN(netD_B(real_B), torch.ones_like(netD_B(real_B))) +
            criterion_GAN(netD_B(fake_B.detach()), torch.
↪zeros_like(netD_B(fake_B)))
        ) * 0.5
        loss_D_B.backward()
        optimizer_D_B.step()

        if i % 200 == 0:
            print(f"[Epoch {epoch}/{5}] [Batch {i}]")
```

3.5 Let's visualise some images that have been generated

[47]:
```
def show_images(imgs, title=''):
    imgs = [img.cpu().detach() for img in imgs]
    grid = make_grid(imgs, nrow=len(imgs), normalize=True)
    plt.figure(figsize=(12, 6))
    plt.title(title)
    plt.imshow(grid.permute(1, 2, 0))
    plt.axis("off")
    plt.show()

sample = next(iter(dataloader))
real_A = sample["A"].to(device)
fake_B = netG_A2B(real_A)
rec_A = netG_B2A(fake_B)

show_images([real_A[0], fake_B[0], rec_A[0]], title="A → B → A")
```



Lab Work 22
Variational Auto-encoders (VAEs)

1 Aim: To learn about the data distribution in latent space using a Variational Autoencoder (VAE)

2 Theory

Variational Autoencoders (VAEs) are generative models that combine neural networks with Bayesian inference to generate new data samples, such as images or text, applicable in tasks like image synthesis, data denoising, or representation learning. The objective is to learn a latent representation of the data distribution $p(\mathbf{x})$ and generate samples by modeling the joint distribution $p(\mathbf{x}, \mathbf{z})$ of observed data $\mathbf{x} \in \mathbb{R}^d$ and latent variables $\mathbf{z} \in \mathbb{R}^m$.

VAE Architecture: A VAE consists of an encoder and a decoder, both typically implemented as neural networks. The encoder maps input data $\mathbf{x}$ to a latent distribution, parameterized by mean μ and variance σ^2:

$$q(\mathbf{z}|\mathbf{x}; \phi) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}), \text{diag}(\sigma^2(\mathbf{x}))),$$

where $\mu(\mathbf{x}), \sigma^2(\mathbf{x})$ are outputs of the encoder network with parameters ϕ. The decoder reconstructs data from a latent sample $\mathbf{z}$:

$$p(\mathbf{x}|\mathbf{z}; \theta) = \mathcal{N}(\mathbf{x}; \hat{\mathbf{x}}(\mathbf{z}), \mathbf{I}),$$

where $\hat{\mathbf{x}}(\mathbf{z})$ is the decoder output with parameters θ. For images, convolutional layers are used; for text, fully connected or recurrent layers may apply. The reparameterization trick samples $\mathbf{z}$ as:

$$\mathbf{z} = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}),$$

enabling differentiable training.

Training Mechanism: VAEs optimize the Evidence Lower Bound (ELBO) to approximate the log-likelihood $\log p(\mathbf{x})$:

$$\mathcal{L}_{\text{ELBO}} = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x}|\mathbf{z})] - D_{\text{KL}}(q(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})),$$

where $p(\mathbf{z}) = \mathcal{N}(0, \mathbf{I})$ is the prior, and D_{KL} is the Kullback-Leibler divergence, regularizing the latent distribution. The first term (reconstruction loss) encourages accurate data reconstruction, often using mean squared error:

$$\mathbb{E}[\log p(\mathbf{x}|\mathbf{z})] \approx -\frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2.$$

The KL term ensures $q(\mathbf{z}|\mathbf{x})$ aligns with the prior, promoting a structured latent space. Training uses optimizers like Adam, with mini-batches for scalability.

Advantages and Challenges: VAEs generate diverse samples and learn interpretable latent spaces, ideal for tasks requiring continuous representations (e.g., image interpolation). Unlike GANs, VAEs provide a principled probabilistic framework, avoiding mode collapse and offering

stable training. However, generated samples may be blurrier than GAN outputs due to the reconstruction loss, and the KL term can overly constrain the latent space, reducing expressiveness. Variants like β-VAEs adjust the KL weight to balance reconstruction and regularization.

Training Dynamics: VAEs require large, diverse datasets (e.g., MNIST, CelebA) to learn robust distributions. Preprocessing (e.g., normalization) and architectural choices (e.g., latent dimension m) impact performance. Hyperparameter tuning, particularly the balance between reconstruction and KL terms, is critical for quality. Techniques like annealing the KL term during early training improve convergence.

3 Code

3.1 Loading all the libraries

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
device = torch.device("cuda:0")
```

3.2 Hyperparameters for encoder and decoder

[3]:
```
input_dim = 784  # 28x28 MNIST images
hidden_dim = 400
latent_dim = 2  # 2D latent space for visualization
batch_size = 128
epochs = 20
lr = 1e-3
```

[5]:
```
!unzip MNIST.zip

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
↪5,), (0.5,))])
train_dataset = torchvision.datasets.MNIST(root='/content/MNIST', train=True,␣
↪download=False, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

```
Archive:  MNIST.zip
   creating: MNIST/
  inflating: __MACOSX/._MNIST
   creating: MNIST/MNIST/
  inflating: __MACOSX/MNIST/._MNIST
  inflating: MNIST/.DS_Store
```

```
  inflating: __MACOSX/MNIST/._.DS_Store
   creating: MNIST/raw/
  inflating: __MACOSX/MNIST/._raw
   creating: MNIST/MNIST/raw/
  inflating: __MACOSX/MNIST/MNIST/._raw
  inflating: MNIST/raw/t10k-images-idx3-ubyte
  inflating: __MACOSX/MNIST/raw/._t10k-images-idx3-ubyte
  inflating: MNIST/raw/t10k-labels-idx1-ubyte
  inflating: __MACOSX/MNIST/raw/._t10k-labels-idx1-ubyte
  inflating: MNIST/raw/train-images-idx3-ubyte
  inflating: __MACOSX/MNIST/raw/._train-images-idx3-ubyte
  inflating: MNIST/raw/t10k-images-idx3-ubyte.gz
  inflating: __MACOSX/MNIST/raw/._t10k-images-idx3-ubyte.gz
  inflating: MNIST/raw/train-images-idx3-ubyte.gz
  inflating: __MACOSX/MNIST/raw/._train-images-idx3-ubyte.gz
  inflating: MNIST/raw/train-labels-idx1-ubyte.gz
  inflating: __MACOSX/MNIST/raw/._train-labels-idx1-ubyte.gz
  inflating: MNIST/raw/train-labels-idx1-ubyte
  inflating: __MACOSX/MNIST/raw/._train-labels-idx1-ubyte
  inflating: MNIST/raw/t10k-labels-idx1-ubyte.gz
  inflating: __MACOSX/MNIST/raw/._t10k-labels-idx1-ubyte.gz
  inflating: MNIST/MNIST/raw/t10k-images-idx3-ubyte
  inflating: __MACOSX/MNIST/MNIST/raw/._t10k-images-idx3-ubyte
  inflating: MNIST/MNIST/raw/t10k-labels-idx1-ubyte
  inflating: __MACOSX/MNIST/MNIST/raw/._t10k-labels-idx1-ubyte
  inflating: MNIST/MNIST/raw/train-images-idx3-ubyte
  inflating: __MACOSX/MNIST/MNIST/raw/._train-images-idx3-ubyte
  inflating: MNIST/MNIST/raw/t10k-images-idx3-ubyte.gz
  inflating: __MACOSX/MNIST/MNIST/raw/._t10k-images-idx3-ubyte.gz
  inflating: MNIST/MNIST/raw/train-images-idx3-ubyte.gz
  inflating: __MACOSX/MNIST/MNIST/raw/._train-images-idx3-ubyte.gz
  inflating: MNIST/MNIST/raw/train-labels-idx1-ubyte.gz
  inflating: __MACOSX/MNIST/MNIST/raw/._train-labels-idx1-ubyte.gz
  inflating: MNIST/MNIST/raw/train-labels-idx1-ubyte
  inflating: __MACOSX/MNIST/MNIST/raw/._train-labels-idx1-ubyte
  inflating: MNIST/MNIST/raw/t10k-labels-idx1-ubyte.gz
  inflating: __MACOSX/MNIST/MNIST/raw/._t10k-labels-idx1-ubyte.gz
```

3.3 We define the VAE model

[7]:
```python
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
```

```
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU()
        )
        self.fc_mu = nn.Linear(hidden_dim // 2, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim // 2, latent_dim)
        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()  # Output in [0,1]
        )

    def encode(self, x):
        h = self.encoder(x)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        return self.decoder(z)

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, input_dim))
        z = self.reparameterize(mu, logvar)
        recon_x = self.decode(z)
        return recon_x, mu, logvar
```

[8]:
```
def loss_function(recon_x, x, mu, logvar):
    BCE = nn.functional.binary_cross_entropy(recon_x, x.view(-1, input_dim),␣
↪reduction="sum")
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

[9]:
```
model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
```

[10]:
```
def train():
    model.train()
    for epoch in range(epochs):
```

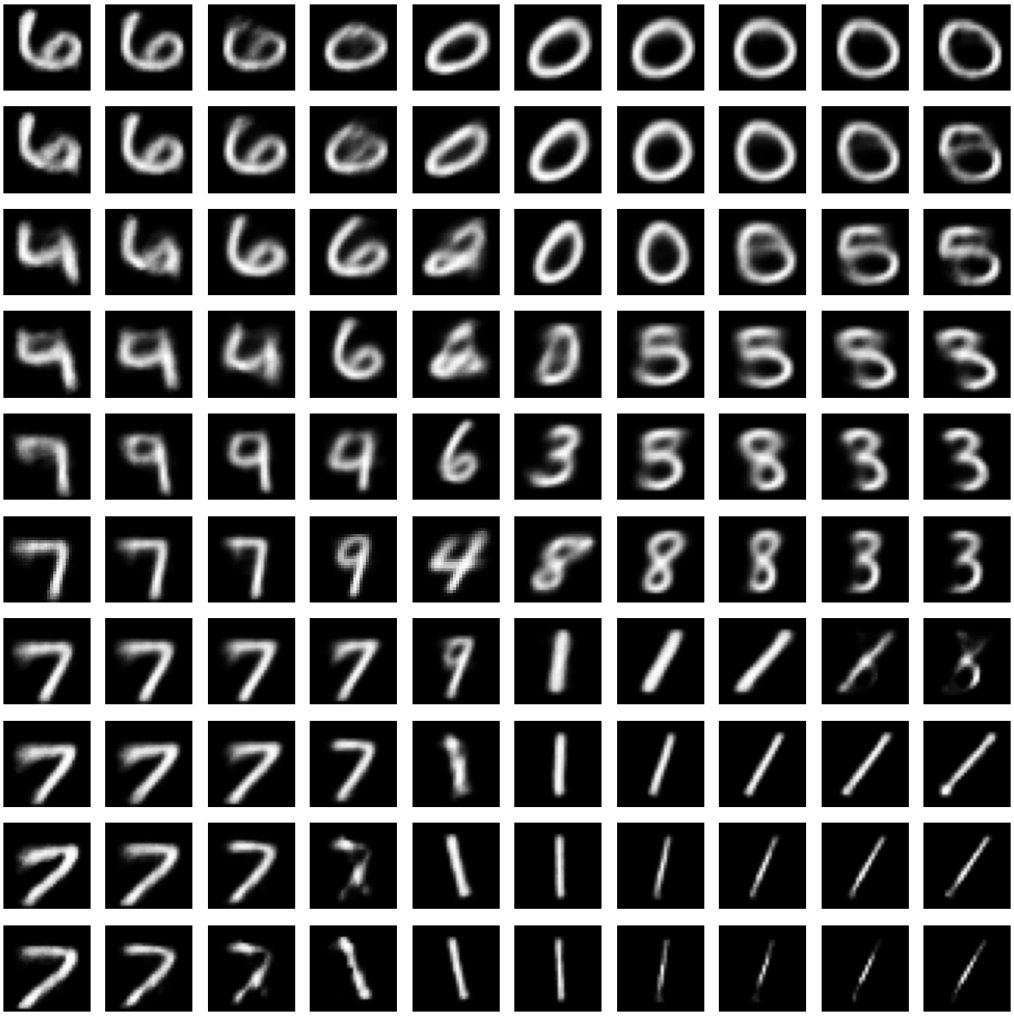
```
        train_loss = 0
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(data)
            loss = loss_function(recon_batch, data, mu, logvar)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()
        print(f"Epoch {epoch+1}, Loss: {train_loss / len(train_loader.dataset):.
↪4f}")
```

[11]:
```
def latent_space_traversal(model, n_steps=10, z_range=3):
    model.eval()
    fig, axes = plt.subplots(n_steps, n_steps, figsize=(10, 10))
    with torch.no_grad():
        z1 = np.linspace(-z_range, z_range, n_steps)
        z2 = np.linspace(-z_range, z_range, n_steps)
        for i, z1_val in enumerate(z1):
            for j, z2_val in enumerate(z2):
                z_sample = torch.tensor([[z1_val, z2_val]], dtype=torch.float32).
↪to(device)
                x_decoded = model.decode(z_sample)
                x_decoded = x_decoded.view(28, 28).cpu().numpy()
                axes[i, j].imshow(x_decoded, cmap="gray")
                axes[i, j].axis("off")
    plt.tight_layout()
    plt.show()
```

[12]:
```
# Run training and visualization
train()
latent_space_traversal(model)
```

```
Epoch 1, Loss: 184.8657
Epoch 2, Loss: 160.8631
Epoch 3, Loss: 155.2046
Epoch 4, Loss: 151.6053
Epoch 5, Loss: 149.0926
Epoch 6, Loss: 147.3473
Epoch 7, Loss: 146.2296
Epoch 8, Loss: 145.0989
Epoch 9, Loss: 144.2798
Epoch 10, Loss: 143.5598
Epoch 11, Loss: 142.8858
Epoch 12, Loss: 142.3469
Epoch 13, Loss: 141.9465
Epoch 14, Loss: 141.4069
Epoch 15, Loss: 140.8824
```

```
Epoch 16, Loss: 140.5154
Epoch 17, Loss: 140.0887
Epoch 18, Loss: 139.7374
Epoch 19, Loss: 139.5347
Epoch 20, Loss: 139.2701
```



Lab Work 23
Auto-encoders for Noise-Removal

1 Aim: To implement autoencoders for noise removal

2 Theory

Denoising Autoencoders (DAEs) are a variant of autoencoders designed to reconstruct clean data from noisy inputs, used in tasks like image denoising, data preprocessing, and robust feature learning. The objective is to learn a robust latent representation that captures the underlying structure of data $\mathbf{x} \in \mathbb{R}^d$ by training a neural network to recover $\mathbf{x}$ from a corrupted version $\tilde{\mathbf{x}}$, enhancing generalization and resilience to noise.

DAE Architecture: A DAE consists of an encoder and a decoder. The encoder maps a noisy input $\tilde{\mathbf{x}}$ to a latent representation $\mathbf{z}$:

$$\mathbf{z} = f_{\text{enc}}(\tilde{\mathbf{x}}; \theta_{\text{enc}}) = \sigma(\mathbf{W}_{\text{enc}}\tilde{\mathbf{x}} + \mathbf{b}_{\text{enc}}),$$

where θ_{enc} are encoder parameters, σ is a non-linear activation (e.g., ReLU), and $\mathbf{z} \in \mathbb{R}^m$ (with $m < d$ for compression). The decoder reconstructs the clean data:

$$\hat{\mathbf{x}} = f_{\text{dec}}(\mathbf{z}; \theta_{\text{dec}}) = \sigma(\mathbf{W}_{\text{dec}}\mathbf{z} + \mathbf{b}_{\text{dec}}),$$

where θ_{dec} are decoder parameters. For images, convolutional layers replace fully connected ones to preserve spatial structure. Noise is introduced to the input via a corruption process, such as Gaussian noise $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2\mathbf{I})$, or dropout (randomly masking elements).

Training Mechanism: DAEs minimize a reconstruction loss between the clean input $\mathbf{x}$ and the reconstructed output $\hat{\mathbf{x}}$, typically mean squared error:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|_2^2,$$

where $\hat{\mathbf{x}}_i = f_{\text{dec}}(f_{\text{enc}}(\tilde{\mathbf{x}}_i))$. The model learns to denoise by capturing robust features that generalize beyond the noise. Training uses optimizers like Adam, with the encoder and decoder jointly optimized via backpropagation. The noise level (e.g., σ^2 or dropout rate) is a critical hyperparameter, balancing robustness and reconstruction fidelity.

Advantages and Challenges: DAEs learn robust, generalizable representations, making them effective for denoising (e.g., removing noise from medical images) and pre-training for downstream tasks (e.g., classification). They outperform standard autoencoders by preventing trivial identity mappings and are simpler than generative models like VAEs or GANs. However, excessive noise can degrade performance, and DAEs may struggle with complex, high-dimensional data without deep architectures. Overfitting to the training noise distribution is a risk, requiring careful regularization.

Training Dynamics: DAEs require large, diverse datasets (e.g., MNIST, CIFAR-10) to learn meaningful representations. Preprocessing (e.g., normalization) ensures consistent input scales. Architectural choices, such as latent dimension m or convolutional depth, impact capacity. Regularization (e.g., dropout, weight decay) and noise scheduling (e.g., varying noise levels) enhance robustness. DAEs can be stacked or integrated with other models (e.g., for semi-supervised learning) to improve performance.

3 Code

3.1 Loading all the libraries

[45]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import numpy as np
import matplotlib.pyplot as plt
from math import log10
```

3.2 We will add Gaussian noise to the data

[44]:
```
class AddGaussianNoise:
    def __init__(self, mean=0., std=0.5):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

transform = transforms.Compose([
    transforms.ToTensor(),
    AddGaussianNoise(0., 0.5),
    transforms.Lambda(lambda x: torch.clamp(x, 0., 1.))  # clip to valid range
])

clean_transform = transforms.ToTensor()
```

[54]:
```
train_dataset_noisy = datasets.MNIST(root='./data', train=True,␣
↪transform=transform, download=True)
train_dataset_clean = datasets.MNIST(root='./data', train=True,␣
↪transform=clean_transform, download=True)

test_dataset_noisy = datasets.MNIST(root='./data', train=False,␣
↪transform=transform, download=True)
test_dataset_clean = datasets.MNIST(root='./data', train=False,␣
↪transform=clean_transform, download=True)

train_loader = DataLoader(list(zip(train_dataset_noisy, train_dataset_clean)),␣
↪batch_size=128, shuffle=True)
test_loader = DataLoader(list(zip(test_dataset_noisy, test_dataset_clean)),␣
↪batch_size=128, shuffle=False)
```

3.3 Now we define the autoencoder

[47]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),  # -> (16, 14, 14)
            nn.ReLU(True),
            nn.Conv2d(16, 8, 3, stride=2, padding=1),  # -> (8, 7, 7)
            nn.ReLU(True)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2, padding=1, output_padding=1),␣
↪ # -> (16, 14, 14)
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),␣
↪ # -> (1, 28, 28)
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

[48]:
```
model = Autoencoder().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

[55]:
```
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    running_loss = 0.0
    for (noisy_data, clean_data) in train_loader:
        noisy_imgs = noisy_data[0].to(device)
        clean_imgs = clean_data[0].to(device)

        optimizer.zero_grad()
        outputs = model(noisy_imgs)
        loss = criterion(outputs, clean_imgs)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f'Epoch [{epoch+1}/{n_epochs}], Loss: {running_loss/len(train_loader):.
↪4f}')
```

```
Epoch [1/10], Loss: 0.0465
Epoch [2/10], Loss: 0.0168
```

```
Epoch [3/10], Loss: 0.0156
Epoch [4/10], Loss: 0.0151
Epoch [5/10], Loss: 0.0149
Epoch [6/10], Loss: 0.0147
Epoch [7/10], Loss: 0.0145
Epoch [8/10], Loss: 0.0144
Epoch [9/10], Loss: 0.0143
Epoch [10/10], Loss: 0.0142
```

[56]:
```python
def compute_psnr(img1, img2):
    mse = ((img1 - img2) ** 2).mean().item()
    if mse == 0:
        return 100
    return 20 * log10(1.0 / np.sqrt(mse))
```

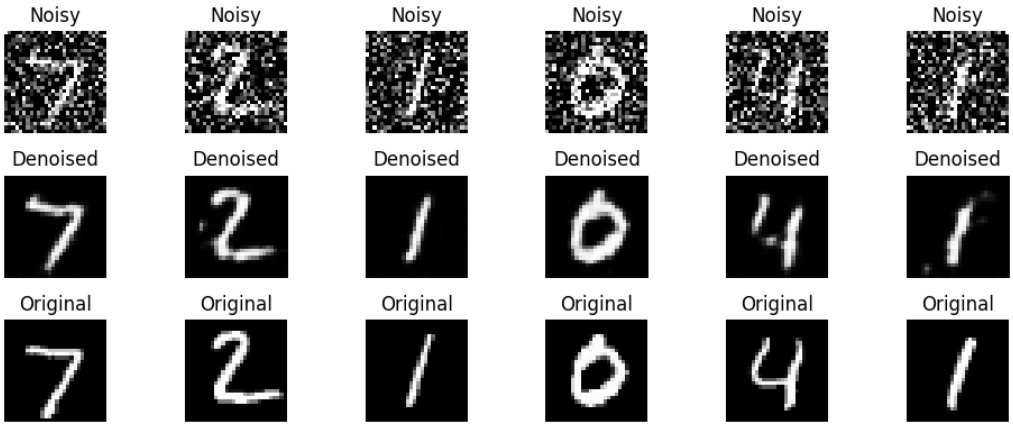
[58]:
```python
model.eval()
psnr_total = 0
with torch.no_grad():
    for noisy_imgs, clean_imgs in test_loader:
        noisy_imgs = noisy_imgs[0].to(device)
        clean_imgs = clean_imgs[0].to(device)
        outputs = model(noisy_imgs)

        psnr_total += compute_psnr(outputs.cpu(), clean_imgs.cpu())

avg_psnr = psnr_total / len(test_loader)
print(f'\nAverage PSNR on test set: {avg_psnr:.2f} dB')
```

```
Average PSNR on test set: 18.57 dB
```

3.4 We visualise the set of clean, noisy and denoised images

[62]:
```python
def imshow(tensor, title=""):
    img = tensor.detach().squeeze().numpy() # Detach the tensor from the␣
 ↪computation graph
    plt.imshow(img, cmap='gray')
    plt.title(title)
    plt.axis('off')

dataiter = iter(test_loader)
(noisy_data, clean_data) = next(dataiter)
noisy_imgs = noisy_data[0].to(device)
clean_imgs = clean_data[0].to(device)
output = model(noisy_imgs)

# Plot
plt.figure(figsize=(10, 4))
```

```
for i in range(6):
    plt.subplot(3, 6, i+1)
    imshow(noisy_imgs[i].cpu(), "Noisy")
    plt.subplot(3, 6, i+7)
    imshow(output[i].cpu(), "Denoised")
    plt.subplot(3, 6, i+13)
    imshow(clean_imgs[i].cpu(), "Original")
plt.tight_layout()
plt.show()
```



Lab Work 24
Attention Mechanism in NLP

1 Aim: To implement scaled dot-product attention in NLP

2 Theory

Attention mechanisms have transformed Natural Language Processing (NLP) by enabling models to focus on relevant parts of input sequences, significantly improving performance in tasks like machine translation, sentiment analysis, and question answering. The objective is to capture contextual dependencies in text sequences $\mathbf{x} = (x_1, \ldots, x_T)$, where $x_t \in \mathbb{R}^d$ represents a word embedding, by dynamically weighting input tokens based on their relevance to the task, unlike standard Recurrent Neural Networks (RNNs), which process sequences sequentially.

Attention Mechanism: Attention computes a weighted sum of input representations to produce a context vector for each output. For a sequence of hidden states $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_T] \in \mathbb{R}^{T \times h}$ (e.g., from an RNN or Transformer), attention is defined as:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_{t,i} \mathbf{h}_i, \quad \alpha_{t,i} = \frac{\exp(\text{score}(\mathbf{s}_t, \mathbf{h}_i))}{\sum_{j=1}^{T} \exp(\text{score}(\mathbf{s}_t, \mathbf{h}_j))},$$

where $\mathbf{s}_t$ is the decoder state or query, $\text{score}(\mathbf{s}_t, \mathbf{h}_i)$ is a compatibility function (e.g., dot product: $\mathbf{s}_t^\top \mathbf{h}_i$ or additive: $\mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_t + \mathbf{W}_h \mathbf{h}_i)$), and $\alpha_{t,i}$ are attention weights. In NLP, attention enables models to prioritize relevant tokens (e.g., aligning "cat" in English to "chat" in French translation). Variants like multi-head attention (used in Transformers) compute multiple attention contexts for richer representations.

Role in NLP: Attention is integral to models like Transformers and enhanced Seq2Seq architectures. It addresses limitations of fixed-length context vectors in vanilla Seq2Seq RNNs by dynamically focusing on input tokens, improving alignment and capturing long-range dependencies. Self-attention, where queries, keys, and values are from the same sequence, enables parallel processing and is central to models like BERT for tasks requiring contextual understanding.

Comparison with RNNs: Standard RNNs process sequences sequentially, updating a hidden state $\mathbf{h}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x x_t + \mathbf{b}_h)$. This sequential nature limits parallelization and struggles with long-term dependencies due to vanishing/exploding gradients. RNNs encode entire sequences into a single context vector, losing fine-grained information. Attention mechanisms, conversely:

- **Capture Long-Range Dependencies**: Attention directly connects any pair of tokens, unlike RNNs' sequential propagation.
- **Enable Parallelization**: Self-attention processes all tokens simultaneously, speeding up training compared to RNNs' sequential computation.
- **Provide Interpretability**: Attention weights reveal token importance (e.g., key words in sentiment), while RNN hidden states are less interpretable.

However, attention requires more memory (quadratic in sequence length) and lacks inherent sequential modeling, necessitating positional encodings in Transformers. RNNs are simpler and effective

for short sequences or resource-constrained settings but are outperformed by attention-based models in complex NLP tasks.

Advantages and Challenges: Attention enhances accuracy and scalability in NLP, enabling state-of-the-art performance in tasks like translation (e.g., Transformer-based models). It is computationally intensive and requires large datasets to avoid overfitting. RNNs, while less resource-heavy, are slower to train and less effective for long sequences, making attention the preferred choice for modern NLP.

Training Dynamics: Attention-based models minimize task-specific losses (e.g., cross-entropy for classification) using optimizers like Adam. Pre-trained embeddings (e.g., GloVe) and regularization (e.g., dropout) improve generalization. RNNs rely on backpropagation through time, which is prone to gradient issues, while attention benefits from parallel gradient computation, though it demands careful hyperparameter tuning (e.g., number of heads, layer depth).

3 Code

3.1 Loading all the libraries

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

3.2 Let's take a dummy dataset

[2]:
```
sentences = [
    ("I love machine learning", "ML is amazing"),
    ("Transformers are powerful", "They replaced RNNs"),
    ("Attention is all you need", "That's the paper's title"),
]

# Tokenizer: simple word to index
vocab = set(word for s in sentences for pair in s for word in pair.lower().
 ↪split())
word2idx = {word: idx for idx, word in enumerate(vocab)}
idx2word = {idx: word for word, idx in word2idx.items()}
vocab_size = len(vocab)

# Encode: convert words to indices
def encode(sentence, max_len=5):
    tokens = [word2idx[w] for w in sentence.lower().split()]
    tokens = tokens[:max_len] + [0] * (max_len - len(tokens))
    return torch.tensor(tokens)

data = [(encode(s1), encode(s2)) for s1, s2 in sentences]
```

3.3 We implement the scaled dot-product attention

[3]:
```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, d_k):
        super().__init__()
        self.scale = torch.sqrt(torch.tensor(d_k, dtype=torch.float32))

    def forward(self, Q, K, V):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
        attn_weights = F.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_weights, V)
        return output, attn_weights
```

3.4 And now the attention model

[4]:
```
class SimpleAttentionModel(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.q_linear = nn.Linear(embed_dim, embed_dim)
        self.k_linear = nn.Linear(embed_dim, embed_dim)
        self.v_linear = nn.Linear(embed_dim, embed_dim)
        self.attention = ScaledDotProductAttention(embed_dim)

    def forward(self, src, tgt):
        src_embed = self.embedding(src)  # (B, L, D)
        tgt_embed = self.embedding(tgt)

        Q = self.q_linear(tgt_embed)  # query from target
        K = self.k_linear(src_embed)  # key from source
        V = self.v_linear(src_embed)  # value from source

        attended, weights = self.attention(Q, K, V)
        return attended, weights
```

[5]:
```
embed_dim = 8
model = SimpleAttentionModel(vocab_size, embed_dim)
```

3.5 We now implement the RNN for comparison

[13]:
```
class RNNEncoder(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_dim, batch_first=True)

    def forward(self, src):
```

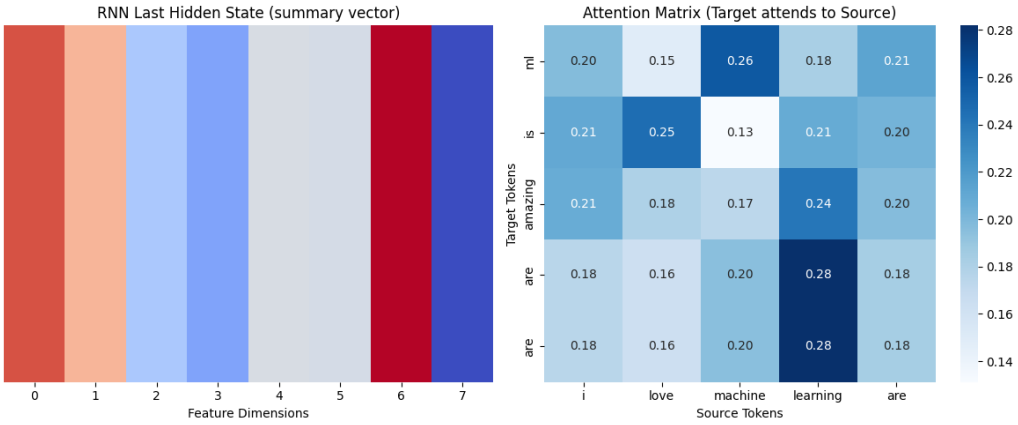
```
        embedded = self.embedding(src)
        output, h_n = self.rnn(embedded)
        return output, h_n
```

[10]:
```
src, tgt = data[0]  # ("I love machine learning", "ML is amazing")
src = src.unsqueeze(0)  # Shape: (1, seq_len)
tgt = tgt.unsqueeze(0)
```

[12]:
```
model.eval()
with torch.no_grad():
    attended_output, attn_weights = model(src, tgt)
```

[14]:
```
rnn_model = RNNEncoder(vocab_size, embed_dim, embed_dim)  # use same dim for␣
↪fair comparison
rnn_model.eval()
with torch.no_grad():
    rnn_output, rnn_hidden = rnn_model(src)
```

[15]:
```
import seaborn as sns
import numpy as np

# Convert attention weights to numpy
attn_matrix = attn_weights.squeeze(0).numpy()

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

# RNN summary vector
plt.subplot(1, 2, 1)
plt.title("RNN Last Hidden State (summary vector)")
sns.heatmap(rnn_hidden.squeeze(0).numpy().reshape(1, -1), cmap='coolwarm',␣
↪cbar=False)
plt.yticks([])
plt.xlabel("Feature Dimensions")

# Attention heatmap
plt.subplot(1, 2, 2)
plt.title("Attention Matrix (Target attends to Source)")
sns.heatmap(attn_matrix, cmap='Blues', annot=True, fmt=".2f",
            xticklabels=[idx2word[i.item()] for i in src.squeeze()],
            yticklabels=[idx2word[i.item()] for i in tgt.squeeze()])
plt.xlabel("Source Tokens")
plt.ylabel("Target Tokens")

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


RNN Last Hidden State (summary vector)
0
1
2
3
4
5
6
7
Feature Dimensions
Attention Matrix (Target attends to Source)
Target Tokens
ml
is
amazing
are
are
0.20 0.15 0.26 0.18 0.21
0.21 0.25 0.13 0.21 0.20
0.21 0.18 0.17 0.24 0.20
0.18 0.16 0.20 0.28 0.18
0.18 0.16 0.20 0.28 0.18
i
love
machine
learning
are
Source Tokens
0.28
0.26
0.24
0.22
0.20
0.18
0.16
0.14

Lab Work 25
Implement Encoder-Decoder Transformer for Translation

1 Aim: To implement a Encoder-Decoder Transformer for translation

2 Theory

The Encoder-Decoder Transformer is a powerful deep learning model designed for machine translation, converting a source language sequence $\mathbf{x} = (x_1, \ldots, x_{T_x})$ (e.g., English sentence) into a target language sequence $\mathbf{y} = (y_1, \ldots, y_{T_y})$ (e.g., French sentence). The objective is to learn a mapping that captures semantic and syntactic relationships across languages, leveraging self-attention to model long-range dependencies and achieve high translation quality on datasets like WMT.

Transformer Architecture: The Transformer comprises an encoder and a decoder, each with stacked layers of multi-head self-attention and feed-forward networks. The encoder processes the source sequence:

$$\mathbf{H} = \text{Encoder}(\mathbf{x}; \theta_{\text{enc}}),$$

where $\mathbf{H} \in \mathbb{R}^{T_x \times d}$ is a sequence of contextualized representations, and θ_{enc} are encoder parameters. Each encoder layer applies:

- **Multi-Head Self-Attention**: Computes attention across all source tokens:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V},$$

  where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{T_x \times d_k}$ are derived from input embeddings with positional encodings, and $d_k = d/h$ for h heads.

- **Feed-Forward Network (FFN)**: Applies position-wise transformations:

$$\text{FFN}(\mathbf{z}) = \text{ReLU}(\mathbf{W}_1\mathbf{z} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

- **Normalization and Residuals**: Layer normalization and skip connections stabilize training.

The decoder generates the target sequence autoregressively, incorporating encoder outputs:

$$\mathbf{y}_t = \text{Decoder}(\mathbf{y}_{1:t-1}, \mathbf{H}; \theta_{\text{dec}}),$$

with layers applying masked self-attention (to prevent attending to future tokens), cross-attention (to attend to $\mathbf{H}$), and FFNs. Cross-attention uses encoder outputs as keys and values, aligning source and target tokens.

Translation Mechanism: The encoder captures source sentence context, producing rich representations $\mathbf{H}$. The decoder uses these to generate target tokens, with cross-attention aligning words (e.g., "dog" to "chien"). Positional encodings ensure order awareness, and the final layer outputs token probabilities:

$$p(y_t) = \text{softmax}(\mathbf{W}_y\mathbf{s}_t + \mathbf{b}_y).$$

Beam search or greedy decoding produces fluent translations.

Advantages and Challenges: The Transformer excels in capturing long-range dependencies and parallelizing computation, outperforming RNN-based Seq2Seq models in speed and accuracy (e.g., higher BLEU scores). It handles variable-length sequences and benefits from pre-training (e.g., BERT encoder). However, it requires substantial computational resources, large datasets, and careful hyperparameter tuning to avoid overfitting. Attention's quadratic complexity with sequence length can be mitigated by variants like Performer.

Training Dynamics: The Transformer minimizes cross-entropy loss over target tokens:

$$\mathcal{L} = -\sum_{t=1}^{T_y} \log p(y_t | y_{1:t-1}, \mathbf{x}).$$

Training uses optimizers like Adam with learning rate schedules (e.g., warmup), leveraging pre-trained embeddings or models. Teacher forcing feeds ground-truth tokens during training, while regularization (e.g., dropout, label smoothing) and data augmentation (e.g., back-translation) enhance generalization. Large parallel corpora (e.g., WMT) are critical for robust performance.

3 Code

3.1 Importing all the libraries

[89]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import random
import nltk
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/amishgogia/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

[89]:
```
True
```

[90]:
```
def tokenize(text):
    return word_tokenize(text.lower())
```

3.2 We will have to define a dummy dataset due to resource constraints

[91]:
```
# Toy dataset
raw_data = [
    ("hello", "hallo"),
    ("how are you", "wie geht es dir"),
    ("i am fine", "mir geht es gut"),
    ("thank you", "danke"),
```

```
    ("good morning", "guten morgen")
]
```

3.3 We will have to built our own vocabulary

[92]:
```
def build_vocab(sentences, specials=["<pad>", "<sos>", "<eos>", "<unk>"]):
    vocab = set()
    for sent in sentences:
        vocab.update(sent.split())
    vocab = specials + sorted(list(vocab))
    stoi = {tok: i for i, tok in enumerate(vocab)}
    itos = {i: tok for tok, i in stoi.items()}
    return stoi, itos
```

[93]:
```
src_sentences = [pair[0] for pair in raw_data]
trg_sentences = [pair[1] for pair in raw_data]

src_vocab, idx2src = build_vocab(src_sentences)
trg_vocab, idx2trg = build_vocab(trg_sentences)
```

[94]:
```
def tokenize(sent, vocab):
    return [vocab.get(tok, vocab["<unk>"]) for tok in sent.split()]

def prepare_data(data, src_vocab, trg_vocab):
    processed = []
    for src, trg in data:
        src_tensor = torch.tensor(tokenize(src, src_vocab), dtype=torch.long)
        trg_tensor = torch.tensor(
            [trg_vocab["<sos>"]] + tokenize(trg, trg_vocab) +␣
 ↪[trg_vocab["<eos>"]],
            dtype=torch.long
        )
        processed.append((src_tensor, trg_tensor))
    return processed
```

[95]:
```
train_data = prepare_data(raw_data, src_vocab, trg_vocab)
test_data = train_data  # Using the same for testing BLEU
```

[96]:
```
pad_idx = trg_vocab["<pad>"]
```

3.4 Dataset and Dataloader

[97]:
```
class TranslationDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
```

```
        return len(self.data)

    def __getitem__(self, idx):
        src, trg = self.data[idx]
        src_tokens = ["<sos>"] + tokenize(src) + ["<eos>"]
        trg_tokens = ["<sos>"] + tokenize(trg) + ["<eos>"]
        src_ids = numericalize(src_tokens, SRC_VOCAB)
        trg_ids = numericalize(trg_tokens, TRG_VOCAB)
        return torch.tensor(src_ids), torch.tensor(trg_ids)

def collate_fn(batch):
    src_batch, trg_batch = zip(*batch)
    src_padded = nn.utils.rnn.pad_sequence(src_batch,␣
 ↪padding_value=SRC_VOCAB["<pad>"], batch_first=True)
    trg_padded = nn.utils.rnn.pad_sequence(trg_batch,␣
 ↪padding_value=TRG_VOCAB["<pad>"], batch_first=True)
    return src_padded, trg_padded

dataset = TranslationDataset(data)
loader = DataLoader(dataset, batch_size=2, collate_fn=collate_fn, shuffle=True)
```

3.5 Now we define the Transformer model

[98]:
```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-torch.
 ↪log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

[99]:
```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.
 ↪log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
```

```
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # shape: (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x


class Transformer(nn.Module):
    def __init__(self, src_vocab_size, trg_vocab_size, d_model=256, nhead=4,␣
 ↪num_layers=3, dropout=0.1, max_len=512):
        super().__init__()
        self.src_embed = nn.Embedding(src_vocab_size, d_model)
        self.trg_embed = nn.Embedding(trg_vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, max_len)
        self.pos_decoder = PositionalEncoding(d_model, max_len)

        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_layers,
            num_decoder_layers=num_layers,
            dropout=dropout,
            batch_first=True  # Important for (batch, seq, features) shape
        )

        self.fc_out = nn.Linear(d_model, trg_vocab_size)

    def forward(self, src, trg, src_pad_mask=None, trg_pad_mask=None):
        # Create target mask for causal (auto-regressive) decoding
        trg_seq_len = trg.size(1)
        tgt_mask = self.generate_square_subsequent_mask(trg_seq_len).to(trg.
 ↪device)

        src = self.src_embed(src)
        trg = self.trg_embed(trg)

        src = self.pos_encoder(src)
        trg = self.pos_decoder(trg)

        out = self.transformer(
            src, trg,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_pad_mask,
            tgt_key_padding_mask=trg_pad_mask,
        )
```

```
        return self.fc_out(out)

    def generate_square_subsequent_mask(self, sz):
        return torch.triu(torch.full((sz, sz), float('-inf')), diagonal=1)
```

[100]:
```
def create_pad_mask(matrix, pad_idx):
    return (matrix == pad_idx)
```

[101]:
```
def train(model, data, optimizer, criterion, pad_idx, device):
    model.train()
    total_loss = 0

    for src, trg in data:
        src = src.unsqueeze(0).to(device)  # (1, src_len)
        trg = trg.unsqueeze(0).to(device)  # (1, trg_len)

        optimizer.zero_grad()

        output = model(
            src, trg[:, :-1],
            src_pad_mask=create_pad_mask(src, pad_idx),
            trg_pad_mask=create_pad_mask(trg[:, :-1], pad_idx)
        )

        output = output.reshape(-1, output.shape[-1])
        target = trg[:, 1:].reshape(-1)

        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(data)
```

[102]:
```
def translate(model, src_tensor, trg_sos_idx, trg_eos_idx, max_len=50,␣
↪pad_idx=0):
    model.eval()
    src_tensor = src_tensor.unsqueeze(0).to(next(model.parameters()).device)
    src_pad_mask = create_pad_mask(src_tensor, pad_idx)

    trg_indexes = [trg_sos_idx]

    for _ in range(max_len):
        trg_tensor = torch.tensor([trg_indexes], dtype=torch.long).to(src_tensor.
↪device)
        trg_pad_mask = create_pad_mask(trg_tensor, pad_idx)
```

```
        output = model(
            src_tensor, trg_tensor,
            src_pad_mask=src_pad_mask,
            trg_pad_mask=trg_pad_mask
        )

        next_token = output[0, -1].argmax(-1).item()
        trg_indexes.append(next_token)

        if next_token == trg_eos_idx:
            break

    return trg_indexes[1:]  # Remove <sos>
```

[103]:
```
def evaluate_bleu(model, data, trg_vocab, idx2word, trg_sos_idx, trg_eos_idx,␣
↪pad_idx):
    smoothie = SmoothingFunction().method4
    scores = []

    for src, trg in data:
        pred_idxs = translate(model, src, trg_sos_idx, trg_eos_idx,␣
↪pad_idx=pad_idx)
        pred_tokens = [idx2word[idx] for idx in pred_idxs if idx != pad_idx]
        ref_tokens = [idx2word[idx.item()] for idx in trg[1:] if idx.item() not␣
↪in [trg_eos_idx, pad_idx]]

        score = sentence_bleu([ref_tokens], pred_tokens,␣
↪smoothing_function=smoothie)
        scores.append(score)

    avg_score = sum(scores) / len(scores)
    print(f"Average BLEU score: {avg_score:.4f}")
    return avg_score
```

[104]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = Transformer(
    src_vocab_size=len(src_vocab),
    trg_vocab_size=len(trg_vocab),
    d_model=256,
    nhead=4,
    num_layers=3
).to(device)

optimizer = optim.Adam(model.parameters(), lr=0.0005)
criterion = nn.CrossEntropyLoss(ignore_index=pad_idx)
```

```
EPOCHS = 50
for epoch in range(1, EPOCHS+1):
    loss = train(model, train_data, optimizer, criterion, pad_idx, device)
    print(f"Epoch {epoch}: Loss = {loss:.4f}")
```

```
Epoch 1: Loss = 3.2602
Epoch 2: Loss = 2.2789
Epoch 3: Loss = 2.0638
Epoch 4: Loss = 1.5304
Epoch 5: Loss = 0.9457
Epoch 6: Loss = 0.7261
Epoch 7: Loss = 0.6817
Epoch 8: Loss = 0.3919
Epoch 9: Loss = 0.2174
Epoch 10: Loss = 0.1944
Epoch 11: Loss = 0.0721
Epoch 12: Loss = 0.0581
Epoch 13: Loss = 0.0370
Epoch 14: Loss = 0.0301
Epoch 15: Loss = 0.0296
Epoch 16: Loss = 0.0223
Epoch 17: Loss = 0.0174
Epoch 18: Loss = 0.0166
Epoch 19: Loss = 0.0126
Epoch 20: Loss = 0.0102
Epoch 21: Loss = 0.0104
Epoch 22: Loss = 0.0083
Epoch 23: Loss = 0.0091
Epoch 24: Loss = 0.0079
Epoch 25: Loss = 0.0077
Epoch 26: Loss = 0.0073
Epoch 27: Loss = 0.0079
Epoch 28: Loss = 0.0068
Epoch 29: Loss = 0.0070
Epoch 30: Loss = 0.0066
Epoch 31: Loss = 0.0061
Epoch 32: Loss = 0.0059
Epoch 33: Loss = 0.0076
Epoch 34: Loss = 0.0058
Epoch 35: Loss = 0.0053
Epoch 36: Loss = 0.0059
Epoch 37: Loss = 0.0055
Epoch 38: Loss = 0.0049
Epoch 39: Loss = 0.0048
Epoch 40: Loss = 0.0053
Epoch 41: Loss = 0.0049
Epoch 42: Loss = 0.0045
```

```
Epoch 43: Loss = 0.0043
Epoch 44: Loss = 0.0045
Epoch 45: Loss = 0.0043
Epoch 46: Loss = 0.0042
Epoch 47: Loss = 0.0046
Epoch 48: Loss = 0.0040
Epoch 49: Loss = 0.0041
Epoch 50: Loss = 0.0036
```

[105]:
```
evaluate_bleu(
    model=model,
    data=test_data,
    trg_vocab=trg_vocab,
    idx2word=idx2trg,  # Dictionary: index → word
    trg_sos_idx=trg_vocab["<sos>"],
    trg_eos_idx=trg_vocab["<eos>"],
    pad_idx=trg_vocab["<pad>"]
)
```

```
Average BLEU score: 0.3369
```

[105]:
```
0.3368698148413915
```