Lab1_FFNN

November 10, 2025

1 Laboratory 1 — Feed Forward Neural Networks (FFNN)

This notebook implements the first lab of the *AI and Cybersecurity* course.

It follows the official brief (`resources/Lab1_FFNN.txt`) and develops a full **Machine Learning pipeline** using PyTorch to explore, train, and evaluate Feed Forward Neural Networks on the CICIDS2017 dataset.

This lab is organized into tasks: - Task 1: Data preprocessing (cleaning, splitting, outliers, normalization) - Task 2: Shallow NN (1 layer), train sizes {32, 64, 128}, metrics and analysis; then ReLU change - Task 3: Impact of specific features (Destination Port), bias test and port removal - Task 4: Loss function impact (weighted CrossEntropy) - Task 5: Deep NN, batch size, optimizer comparisons - Task 6: Overfitting and regularization (dropout, batchnorm, weight decay)

1.1 Setup

[1]:
```python
import sys
print(sys.executable)
print(sys.version)
```

```
/Users/eliainnocenti/Documents/Projects/LaTeX/AImSecure/venv/bin/python3.14
3.14.0 (main, Oct  7 2025, 09:34:52) [Clang 17.0.0 (clang-1700.3.19.1)]
```

[2]:
```python
# --- Import libraries ---
import os
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import json

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler,␣
↪LabelEncoder
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report, confusion_matrix,␣
↪accuracy_score, f1_score
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
```

1.1.1 Device Settings

[3]:
```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = "cpu"
print(f"The device is set to: {device}")
```

```
The device is set to: cpu
```

1.1.2 Paths setup

[4]:
```python
# --- Define Paths ---
project_path = '../'
data_path = project_path + 'data/'
results_path = project_path + 'results/'

# Ensure directories exist
os.makedirs(project_path, exist_ok=True)
os.makedirs(data_path, exist_ok=True)
os.makedirs(results_path, exist_ok=True)

print(f"Project path: {project_path}")
print(f"Data path: {data_path}")
print(f"Results path: {results_path}")
```

```
Project path: ../
Data path: ../data/
Results path: ../results/
```

[5]:
```python
# --- Set visual style ---
sns.set(style="whitegrid", palette="muted", font_scale=1.1)

def save_plot(fig: plt.Figure, filename: str, path: str = "./plots/", fmt: str␣
↪= "png", dpi: int = 300, close_fig: bool = False) -> None:
    """
    Save a Matplotlib figure in a specific to a specified directory.

    Args:
        fig (plt.Figure): Matplotlib figure object to save.
        filename (str): Name of the file to save (e.g., 'plot.png').
        path (str, optional): Directory path to save the figure. Defaults to './
↪plots/'.
        fmt (str, optional): File format for the saved figure. Defaults to␣
↪'png'.
```

```
        dpi (int, optional): Dots per inch for the saved figure. Defaults to␣
↪300.

    Returns:
        None
    """
    # Ensure the directory exists
    os.makedirs(path, exist_ok=True)
    save_path = os.path.join(path, f"{filename}.{fmt}")

    # Save the figure
    fig.savefig(save_path, bbox_inches='tight', pad_inches=0.1, dpi=dpi,␣
↪format=fmt)
    # plt.close(fig) # Removed to display plots in notebook

    if close_fig:
        plt.close(fig)

    print(f"Saved plot: {save_path}")
```

1.2 Task 1 — Data preprocessing

In this task, we prepare the dataset and ensure proper data quality and normalization before training.
We will: - Load the dataset from `lab/data/dataset_lab_1.csv`. - Inspect basic statistics, feature ranges, and class distribution. - Remove missing values and duplicates (report before/after counts). - Split the data into **train / validation / test (60% / 20% / 20%)**, using a fixed random seed for reproducibility. - Perform **outlier analysis** (e.g., KDE, Z-score, IQR) and decide on an appropriate **scaling method**. - Fit the scaler on the training data only and apply it to validation and test splits.

[6]:
```
# --- Load dataset and perform initial inspection ---

# Set random seed for reproducibility
np.random.seed(42)

# Create directory for plots
save_dir = results_path + 'images/' + 'task1_plots/'
os.makedirs(save_dir, exist_ok=True)

# Load Dataset
file_path = data_path + 'dataset_lab_1.csv'
df = pd.read_csv(file_path)

# Basic info
print("Shape (raw):", df.shape)
print("Columns:", list(df.columns))
```

```
print("\nLabel distribution (raw):")
print(df['Label'].value_counts(dropna=False))
```

```
Shape (raw): (31507, 17)
Columns: ['Flow Duration', 'Flow IAT Mean', 'Fwd PSH Flags', 'Bwd Packet Length
Mean', 'Bwd Packet Length Max', 'Flow Bytes/s', 'Down/Up Ratio', 'SYN Flag
Count', 'Fwd Packet Length Mean', 'Fwd IAT Std', 'Packet Length Mean', 'Fwd
Packet Length Max', 'Subflow Fwd Packets', 'Flow Packets/s', 'Total Fwd
Packets', 'Destination Port', 'Label']

Label distribution (raw):
Label
Benign         20000
DoS Hulk        5000
PortScan        5000
Brute Force     1507
Name: count, dtype: int64
```

[7]:
```
df
```

[7]:
```
       Flow Duration  Flow IAT Mean  Fwd PSH Flags  Bwd Packet Length Mean  \
0             303376   3.033760e+04              0                   749.4
1                117   1.170000e+02              0                     0.0
2                142   1.420000e+02              0                     0.0
3                191   6.366667e+01              0                    52.0
4                  4   4.000000e+00              0                     0.0
…                  …              …              …                       …
31502        5710955   1.903652e+06              0                     0.0
31503        5862561   1.954187e+06              0                     0.0
31504        5854872   1.951624e+06              0                     0.0
31505        5555547   1.851849e+06              0                     0.0
31506        5018204   1.672735e+06              0                     0.0

       Bwd Packet Length Max  Flow Bytes/s  Down/Up Ratio  SYN Flag Count  \
0                       1448     12743.263              0               0
1                          0         0.000              1               0
2                          0     84507.040              0               0
3                         52    921465.940              1               0
4                          0   3000000.000              0               0
…                          …             …              …               …
31502                      0         0.000              0               0
31503                      0         0.000              0               0
31504                      0         0.000              0               0
31505                      0         0.000              0               0
31506                      0         0.000              0               0

       Fwd Packet Length Mean  Fwd IAT Std  Packet Length Mean  \
```

```
0                   19.833334     98776.15          322.16666
1                    0.000000         0.00            0.00000
2                    6.000000         0.00            6.00000
3                   36.000000         0.00           42.40000
4                    6.000000         0.00            6.00000
…                         …           …               …
31502                0.000000   4037277.80            0.00000
31503                0.000000   4144374.80            0.00000
31504                0.000000   4139029.80            0.00000
31505                0.000000   3927356.50            0.00000
31506                0.000000   3547404.80            0.00000

       Fwd Packet Length Max  Subflow Fwd Packets  Flow Packets/s  \
0                        119                    6       36.258636
1                          0                    1    17094.018000
2                          6                    2    14084.507000
3                         36                    2    20942.408000
4                          6                    2   500000.000000
…                        …                    …             …
31502                      0                    3        0.700408
31503                      0                    3        0.682296
31504                      0                    3        0.683192
31505                      0                    3        0.720001
31506                      0                    3        0.797098

       Total Fwd Packets  Destination Port        Label
0                      6               443       Benign
1                      1             52631       Benign
2                      2                80       Benign
3                      2                53       Benign
4                      2             49467       Benign
…                    …                …          …
31502                  3                80  Brute Force
31503                  3                80  Brute Force
31504                  3                80  Brute Force
31505                  3                80  Brute Force
31506                  3                80  Brute Force

[31507 rows x 17 columns]
```

[8]:
```python
def plot_class_distribution(
    df,
    name_fig='class_distribution',
    label_col='Label',
    save_path='./plots/',
    fig_size=(8, 5),
    palette='pastel'
```

```python
):
    """
    Plot the number of samples for each class.

    Args:
        df (pd.DataFrame): The input DataFrame.
        label_col (str): The name of the label column. Defaults to 'Label'.
        save_path (str): The directory to save the plot. Defaults to './plots/'.
        fig_size (tuple): Size of the figure. Defaults to (8, 5).
        palette (str): Seaborn palette.
    """
    os.makedirs(save_path, exist_ok=True)

    # Create a figure and axes for the plot
    fig, ax = plt.subplots(figsize=fig_size)

    # Create a countplot showing the distribution of classes
    sns.countplot(x=label_col, data=df, order=df[label_col].value_counts().
↪index, ax=ax, palette=palette, hue=label_col, legend=False)

    # Set the title and labels for the plot
    ax.set_title("Class Distribution")
    ax.set_xlabel("Traffic Type")
    ax.set_ylabel("Count")

    # Rotate x-axis labels for better readability
    plt.xticks(rotation=30)

    # Annotate bars with counts
    for p in ax.patches:
        height = p.get_height()
        ax.annotate(f'{height}', (p.get_x() + p.get_width() / 2., height),
                    ha='center', va='bottom', fontsize=10)

    # Save the plot to the specified path
    save_plot(fig, name_fig, save_path)
```

[9]:
```python
# --- Visualize raw data characteristics and distributions ---

# Plot class distribution to understand data balance
plot_class_distribution(df, 'class_distribution', save_path=save_dir)
```

```
Saved plot: ../results/images/task1_plots/class_distribution.png
```

1.2.1 Removing NaN and duplicates

[10]:
```
# --- Remove NaN and duplicates ---

raw_n = len(df)

# Drop NaN
df = df.dropna()
clean_n = len(df)
print(f"Removed {raw_n-clean_n} rows (NaN)")

# Drop duplicates
df = df.drop_duplicates()
clean_n = len(df)
print(f"Removed {raw_n-clean_n} rows (duplicates)")

print(f"New shape: {df.shape}")
```

```
Removed 20 rows (NaN)
Removed 2114 rows (duplicates)
New shape: (29393, 17)
```

[11]:
```python
# --- Handle infinite values ---

# Replace infinite values with NaN
df.replace([np.inf, -np.inf], np.nan, inplace=True)

# Check for and report the number of infinite values (now NaNs)
inf_counts = df.isnull().sum()
print("\nNumber of infinite values (replaced with NaN) per column:")
print(inf_counts[inf_counts > 0])

# Drop rows with NaN values (including those that were originally infinite)
initial_rows = len(df)
df.dropna(inplace=True)
rows_after_inf_nan_drop = len(df)

print(f"\nRemoved {initial_rows - rows_after_inf_nan_drop} rows containing␣
↪infinite or NaN values. New shape: {df.shape}")
```

```
Number of infinite values (replaced with NaN) per column:
Flow Bytes/s      7
Flow Packets/s    7
dtype: int64

Removed 7 rows containing infinite or NaN values. New shape: (29386, 17)
```

Q: How many samples did you have before and after removing missing and duplicates entries? We initially had **31507 samples** bold text, and after removing missing, duplicate, and infinite entries, we obtained 29,386 clean samples. In total, **2121** (2114 missing and duplicates + 7 infinite values) rows **were removed** during the data cleaning process.

[12]:
```python
print("\nLabel distribution (after NaN and duplicates removal):")
print(df['Label'].value_counts())
```

```
Label distribution (after NaN and duplicates removal):
Label
Benign         19242
PortScan        4849
DoS Hulk        3868
Brute Force     1427
Name: count, dtype: int64
```

[13]:
```python
# Encode labels
label_encoder = LabelEncoder()
df['Label'] = label_encoder.fit_transform(df['Label'])

df.head(10)
```

[13]:

```
   Flow Duration  Flow IAT Mean  Fwd PSH Flags  Bwd Packet Length Mean  \
0         303376   3.033760e+04              0              749.400000
1            117   1.170000e+02              0                0.000000
2            142   1.420000e+02              0                0.000000
3            191   6.366667e+01              0               52.000000
4              4   4.000000e+00              0                0.000000
5         162322   3.959073e+03              0               19.416666
6       61050653   2.907174e+06              0               22.818182
7            108   1.080000e+02              0                0.000000
8         764920   7.649200e+05              0                0.000000
9         122255   6.112750e+04              0                0.000000

   Bwd Packet Length Max  Flow Bytes/s  Down/Up Ratio  SYN Flag Count  \
0                   1448  1.274326e+04              0               0
1                      0  0.000000e+00              1               0
2                      0  8.450704e+04              0               0
3                     52  9.214659e+05              1               0
4                      0  3.000000e+06              0               0
5                     51  4.102956e+03              1               0
6                    233  5.230083e+01              1               0
7                      0  1.111111e+05              0               0
8                      0  1.568792e+01              0               0
9                      0  0.000000e+00              0               0

   Fwd Packet Length Mean  Fwd IAT Std  Packet Length Mean  \
0               19.833334    98776.150          322.166660
1                0.000000        0.000            0.000000
2                6.000000        0.000            6.000000
3               36.000000        0.000           42.400000
4                6.000000        0.000            6.000000
5               11.111111    30630.129           15.488372
6              267.454560  5145240.000          138.826080
7                6.000000        0.000            6.000000
8                6.000000        0.000            6.000000
9                0.000000        0.000            0.000000

   Fwd Packet Length Max  Subflow Fwd Packets  Flow Packets/s  \
0                    119                    6       36.258636
1                      0                    1    17094.018000
2                      6                    2    14084.507000
3                     36                    2    20942.408000
4                      6                    2   500000.000000
5                     43                   18      258.744960
6                   1460                   11        0.360357
7                      6                    2    18518.518000
8                      6                    2        2.614652
9                      0                    2       24.538874
```

```
   Total Fwd Packets  Destination Port  Label
0                  6               443      0
1                  1             52631      0
2                  2                80      0
3                  2                53      0
4                  2             49467      0
5                 18                21      0
6                 11                80      0
7                  2                80      0
8                  2               443      0
9                  2               443      0
```

[14]:
```
# --- Display the label encoding mapping ---
print("\nLabel Encoding Mapping:")
for i, label in enumerate(label_encoder.classes_):
    print(f"{label}: {i}")
```

```
Label Encoding Mapping:
Benign: 0
Brute Force: 1
DoS Hulk: 2
PortScan: 3
```

[15]:
```
# Checking again null values
print(df.isnull().sum())
```

```
Flow Duration               0
Flow IAT Mean               0
Fwd PSH Flags               0
Bwd Packet Length Mean      0
Bwd Packet Length Max       0
Flow Bytes/s                0
Down/Up Ratio               0
SYN Flag Count              0
Fwd Packet Length Mean      0
Fwd IAT Std                 0
Packet Length Mean          0
Fwd Packet Length Max       0
Subflow Fwd Packets         0
Flow Packets/s              0
Total Fwd Packets           0
Destination Port            0
Label                       0
dtype: int64
```

[16]:
```python
# Checking again for the infinite values
print(df.isin([np.inf, -np.inf]).sum())
```

```
Flow Duration              0
Flow IAT Mean              0
Fwd PSH Flags              0
Bwd Packet Length Mean     0
Bwd Packet Length Max      0
Flow Bytes/s               0
Down/Up Ratio              0
SYN Flag Count             0
Fwd Packet Length Mean     0
Fwd IAT Std                0
Packet Length Mean         0
Fwd Packet Length Max      0
Subflow Fwd Packets        0
Flow Packets/s             0
Total Fwd Packets          0
Destination Port           0
Label                      0
dtype: int64
```

1.2.2 Split data

We use a **stratified approach** due to the class imbalance.

[17]:
```python
# --- Split data ---

# Split features/target
label_col = 'Label'
feature_cols = [c for c in df.columns if c != label_col]
X = df[feature_cols].values
y = df[label_col].values

# Train/val/test split 60/20/20 with stratify
X_train, X_tmp, y_train, y_tmp = train_test_split(
    X, y, test_size=0.4, stratify=y, random_state=42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_tmp, y_tmp, test_size=0.5, stratify=y_tmp, random_state=42
)

print("Data Splits:")
print(f"Train set: {X_train.shape[0]:,} samples")
print(f"Validation set: {X_val.shape[0]:,} samples")
print(f"Test set: {X_test.shape[0]:,} samples\n")

def print_label_counts(name, y):
```

```
    labels, counts = np.unique(y, return_counts=True)
    count_width = 6
    print(f"{name:<8}", end=" ")
    for label, count in zip(labels, counts):
        print(f"{label}: {count:>{count_width},}", end="   ")
    print()

print_label_counts("Train", y_train)
print_label_counts("Val", y_val)
print_label_counts("Test", y_test)
```

```
Data Splits:
Train set: 17,631 samples
Validation set: 5,877 samples
Test set: 5,878 samples

Train    0: 11,545   1:    856   2:  2,321   3:  2,909
Val      0:  3,848   1:    286   2:    773   3:    970
Test     0:  3,849   1:    285   2:    774   3:    970
```

1.2.3 Outliers Detection

[18]:
```
# --- Outlier detection (Z-score and IQR) ---

# Convert X_train to DataFrame to use select_dtypes
X_train_df = pd.DataFrame(X_train, columns=feature_cols)

num_cols = X_train_df.select_dtypes(include=[np.number]).columns.tolist()
print(num_cols)

# Z-score method
zs = ((X_train_df[num_cols] - X_train_df[num_cols].mean()) /␣
 ↪X_train_df[num_cols].std(ddof=0)).abs()
outlier_counts_z = (zs > 3).sum().sort_values(ascending=False)
print("\n[Z-SCORE OUTLIERS] features:")
print(outlier_counts_z)

# IQR method
Q1 = X_train_df[num_cols].quantile(0.25)
Q3 = X_train_df[num_cols].quantile(0.75)
IQR = Q3 - Q1
outliers_iqr = ((X_train_df[num_cols] < (Q1 - 1.5 * IQR)) |␣
 ↪(X_train_df[num_cols] > (Q3 + 1.5 * IQR))).sum()
outlier_counts_iqr = outliers_iqr.sort_values(ascending=False)
print("\n[IQR OUTLIERS] features:")
print(outlier_counts_iqr)
```

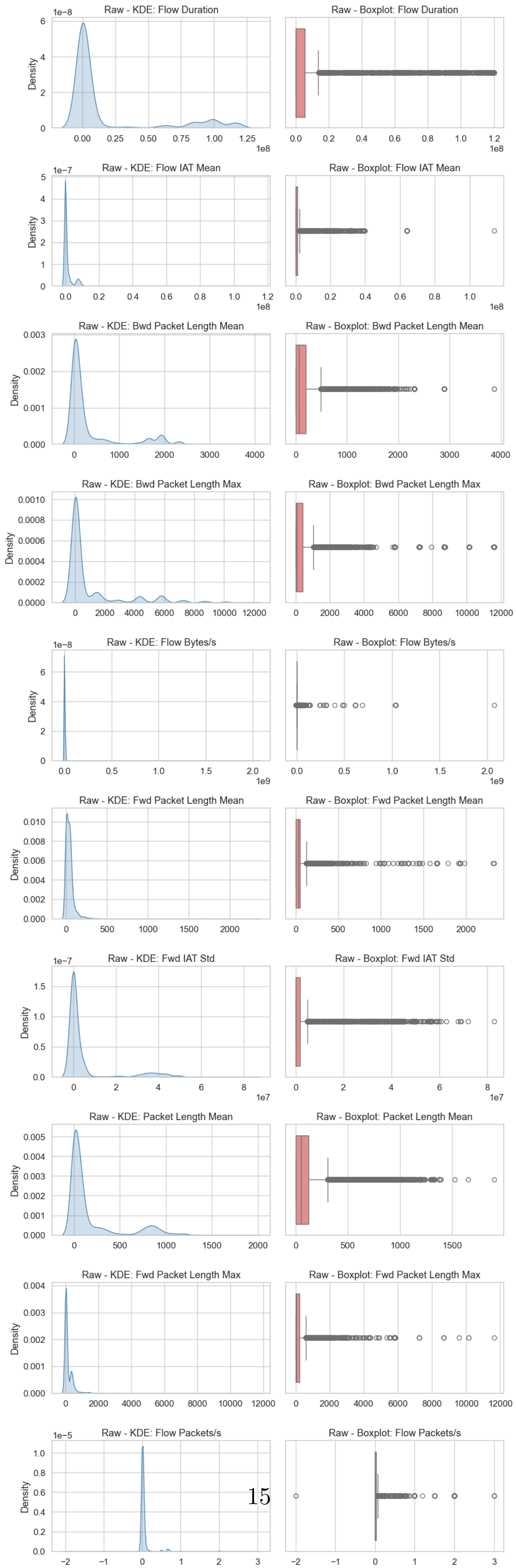
```
['Flow Duration', 'Flow IAT Mean', 'Fwd PSH Flags', 'Bwd Packet Length Mean',
```

```
'Bwd Packet Length Max', 'Flow Bytes/s', 'Down/Up Ratio', 'SYN Flag Count', 'Fwd
Packet Length Mean', 'Fwd IAT Std', 'Packet Length Mean', 'Fwd Packet Length
Max', 'Subflow Fwd Packets', 'Flow Packets/s', 'Total Fwd Packets', 'Destination
Port']

[Z-SCORE OUTLIERS] features:
Fwd PSH Flags             708
SYN Flag Count            708
Fwd IAT Std               567
Bwd Packet Length Max     523
Flow Packets/s            522
Bwd Packet Length Mean    340
Packet Length Mean        272
Destination Port          247
Fwd Packet Length Max     238
Flow IAT Mean             207
Fwd Packet Length Mean    157
Subflow Fwd Packets        88
Total Fwd Packets          88
Down/Up Ratio              48
Flow Bytes/s               37
Flow Duration               0
dtype: int64

[IQR OUTLIERS] features:
Bwd Packet Length Max     3951
Destination Port          3879
Flow Duration             3443
Bwd Packet Length Mean    3180
Packet Length Mean        2982
Flow IAT Mean             2913
Flow Bytes/s              2826
Fwd IAT Std               2662
Subflow Fwd Packets       1467
Total Fwd Packets         1467
Flow Packets/s            1417
Fwd Packet Length Mean     997
Fwd Packet Length Max      929
Fwd PSH Flags              708
SYN Flag Count             708
Down/Up Ratio               48
dtype: int64
```

[19]:
```
# --- Visualize raw distributions for selected features ---

selected_features = [
    'Flow Duration', 'Flow IAT Mean', 'Bwd Packet Length Mean',
```

```
    'Bwd Packet Length Max', 'Flow Bytes/s', 'Fwd Packet Length Mean',
    'Fwd IAT Std', 'Packet Length Mean', 'Fwd Packet Length Max', 'Flow Packets/
 ↪s'
]

# Convert X_train to DataFrame for easier plotting with column names
X_train_df = pd.DataFrame(X_train, columns=feature_cols)

def plot_distributions(df, features, title_prefix, save_name):
    n = len(features)
    fig, axes = plt.subplots(nrows=n, ncols=2, figsize=(10, 3 * n))

    for i, col in enumerate(features):
        # KDE Plot
        sns.kdeplot(df[col], ax=axes[i, 0], fill=True, color="steelblue")
        axes[i, 0].set_title(f"{title_prefix} - KDE: {col}")
        axes[i, 0].set_xlabel("")

        # Boxplot
        sns.boxplot(x=df[col], ax=axes[i, 1], color="lightcoral")
        axes[i, 1].set_title(f"{title_prefix} - Boxplot: {col}")
        axes[i, 1].set_xlabel("")

    plt.tight_layout()

    # Save the plot to the specified path
    save_plot(fig, save_name, save_dir, fmt='pdf')

    plt.show()

plot_distributions(X_train_df, selected_features, "Raw", "raw_distributions")
```

```
Saved plot: ../results/images/task1_plots/raw_distributions.pdf
```

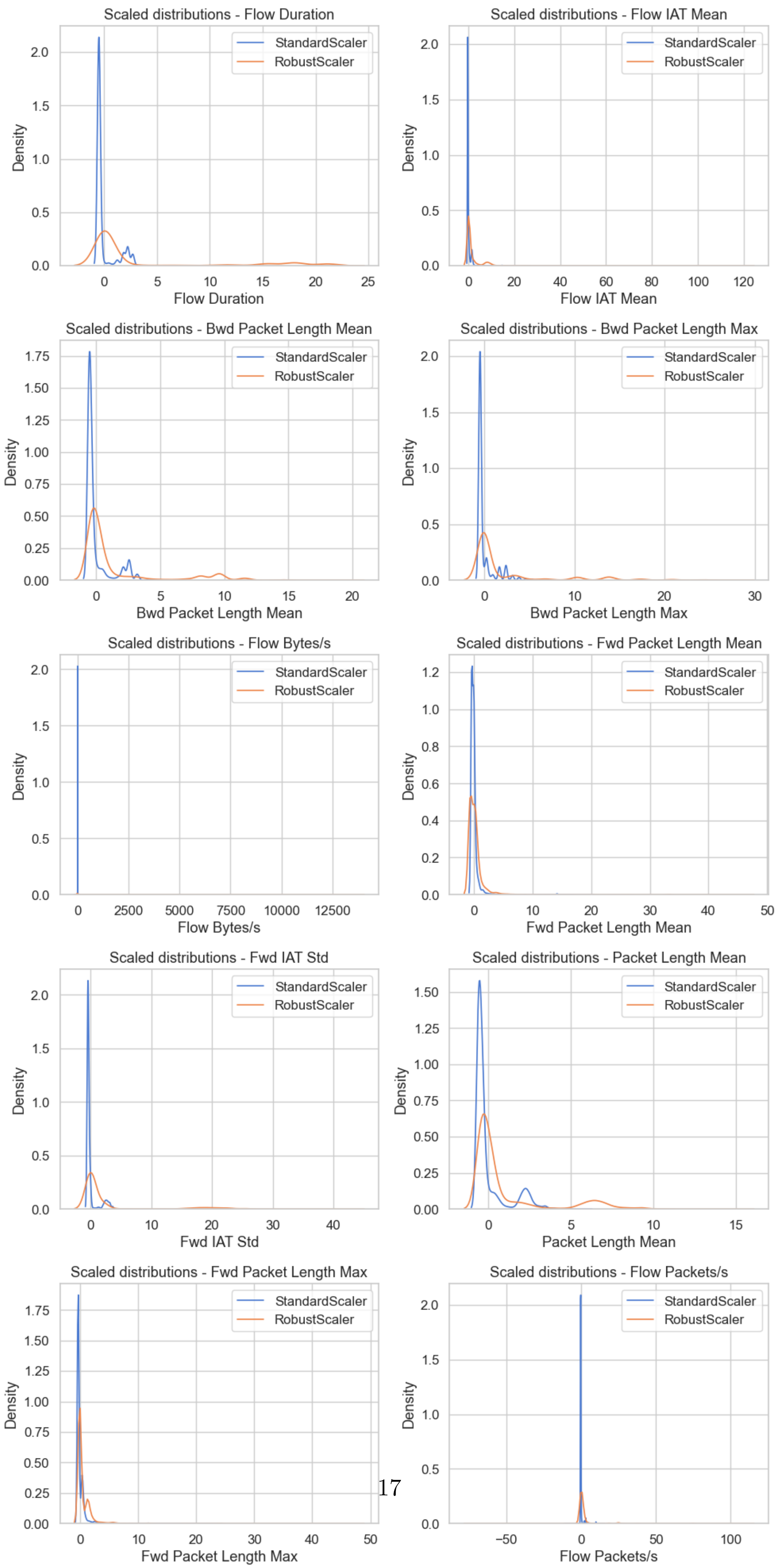

Raw - KDE: Flow Duration
Raw - Boxplot: Flow Duration
Raw - KDE: Flow IAT Mean
Raw - Boxplot: Flow IAT Mean
Raw - KDE: Bwd Packet Length Mean
Raw - Boxplot: Bwd Packet Length Mean
Raw - KDE: Bwd Packet Length Max
Raw - Boxplot: Bwd Packet Length Max
Raw - KDE: Flow Bytes/s
Raw - Boxplot: Flow Bytes/s
Raw - KDE: Fwd Packet Length Mean
Raw - Boxplot: Fwd Packet Length Mean
Raw - KDE: Fwd IAT Std
Raw - Boxplot: Fwd IAT Std
Raw - KDE: Packet Length Mean
Raw - Boxplot: Packet Length Mean
Raw - KDE: Fwd Packet Length Max
Raw - Boxplot: Fwd Packet Length Max
Raw - KDE: Flow Packets/s
Raw - Boxplot: Flow Packets/s
Density

1.2.4 Data Normalization

[20]:
```python
# Standardize the features (StandardScaler)
scaler1 = StandardScaler()
X_train_std = scaler1.fit_transform(X_train)
X_val_std = scaler1.transform(X_val)
X_test_std = scaler1.transform(X_test)

# Standardize the features (RobustScaler)
scaler2 = RobustScaler()
X_train_rob = scaler2.fit_transform(X_train)
X_val_rob = scaler2.transform(X_val)
X_test_rob = scaler2.transform(X_test)
```

[21]:
```python
# --- Visual comparison of Standard vs Robust scaling ---

# Convert scaled arrays back to DataFrames for easier plotting with column names
X_train_std_df = pd.DataFrame(X_train_std, columns=feature_cols)
X_train_rob_df = pd.DataFrame(X_train_rob, columns=feature_cols)

# Adjust subplot grid to accommodate all selected features (10 features, 5 rows␣
↪x 2 columns)
fig, axes = plt.subplots(5, 2, figsize=(10, 20)) # Increased figsize for better␣
↪readability

# Plot all selected features for comparison
for i, col in enumerate(selected_features):
    row_idx = i // 2
    col_idx = i % 2
    sns.kdeplot(X_train_std_df[col], ax=axes[row_idx, col_idx],␣
↪label="StandardScaler", lw=1.2)
    sns.kdeplot(X_train_rob_df[col], ax=axes[row_idx, col_idx],␣
↪label="RobustScaler", lw=1.2)
    axes[row_idx, col_idx].set_title(f"Scaled distributions - {col}")
    axes[row_idx, col_idx].legend()

plt.tight_layout()

# Save the plot to the specified path
save_plot(fig, 'raw_distributions_comparison', save_dir, fmt='pdf')

plt.show()
plt.close(fig)
```

```
Saved plot: ../results/images/task1_plots/raw_distributions_comparison.pdf
```


Scaled distributions - Flow Duration
StandardScaler
RobustScaler
Density
Flow Duration
Scaled distributions - Flow IAT Mean
StandardScaler
RobustScaler
Density
Flow IAT Mean
Scaled distributions - Bwd Packet Length Mean
StandardScaler
RobustScaler
Density
Bwd Packet Length Mean
Scaled distributions - Bwd Packet Length Max
StandardScaler
RobustScaler
Density
Bwd Packet Length Max
Scaled distributions - Flow Bytes/s
StandardScaler
RobustScaler
Density
Flow Bytes/s
Scaled distributions - Fwd Packet Length Mean
StandardScaler
RobustScaler
Density
Fwd Packet Length Mean
Scaled distributions - Fwd IAT Std
StandardScaler
RobustScaler
Density
Fwd IAT Std
Scaled distributions - Packet Length Mean
StandardScaler
RobustScaler
Density
Packet Length Mean
Scaled distributions - Fwd Packet Length Max
StandardScaler
RobustScaler
Density
Fwd Packet Length Max
Scaled distributions - Flow Packets/s
StandardScaler
RobustScaler
Density
Flow Packets/s

[22]:
```python
# --- Compare scaling statistics numerically ---

stats_std = X_train_std_df.describe().T[['mean', 'std']].rename(columns={'mean':
 ↪ 'mean_std', 'std': 'std_std'})
stats_rob = X_train_rob_df.describe().T[['mean', 'std']].rename(columns={'mean':
 ↪ 'mean_rob', 'std': 'std_rob'})
scaling_comparison = stats_std.join(stats_rob)

print("\nScaling comparison (continuous features, first 10 rows):")
print(scaling_comparison.head(10))
```

```
Scaling comparison (continuous features, first 10 rows):
                             mean_std   std_std   mean_rob     std_rob
Flow Duration           -1.612030e-18  1.000028   3.209521    6.578692
Flow IAT Mean            4.693024e-16  1.000028   1.655390    4.604932
Fwd PSH Flags            1.652331e-17  1.000028   0.040157    0.196332
Bwd Packet Length Mean -7.245271e-15  1.000028   1.366380    3.176369
Bwd Packet Length Max    2.579249e-17  1.000028   2.137739    4.824989
Flow Bytes/s             7.818348e-17  1.000028  10.549447  192.030725
Down/Up Ratio            5.642107e-17  1.000028  -0.333503    0.523037
SYN Flag Count           1.652331e-17  1.000028   0.040157    0.196332
Fwd Packet Length Mean   1.018803e-15  1.000028   0.251230    2.318150
Fwd IAT Std             -2.954046e-16  1.000028   2.570055    6.266235
```

Q: How did you normalize the data? Why did you choose it? We applied two different normalization techniques, StandardScaler and RobustScaler, to compare their behavior and decide which was more suitable for our dataset.

- **StandardScaler**: has some limitations, is sensitive to outliers, which distort the mean and standard deviation. Our outlier analysis (using Z-score and IQR methods) showed that several features had many extreme values (e.g. Bwd Packet Length Max, Flow Duration, Bwd Packet Length Mean, and Fwd IAT Std had thousands of outliers).
- **RobustScaler**: This approach is less sensitive to outliers, as it relies on statistics (median and IQR) that are not affected by extreme values. In the comparison plots, the scaled distributions were more compact and less skewed, especially for features with strong outliers like Bwd Packet Length Max, Flow Duration, Bwd Packet Length Mean, and Fwd IAT Std. However, the mean and standard deviation varied across features, which is expected since RobustScaler does not enforce a standard normal distribution.

Despite RobustScaler being more robust to extreme values, we ultimately chose **StandardScaler** for our dataset. The comparison plots showed that the loss curves during training were smoother using the standard standardization, and the performance of preliminary models was very similar between the two scalers. Using StandardScaler also simplifies interpretation.

1.3 Task 2 — Shallow Neural Network (1 layer)

We design and train three shallow FFNN models with a single hidden layer, varying the hidden size.
Configuration: - Hidden layer sizes: **{32, 64, 128}** - Activation: **Linear** (first run) - Optimizer: **AdamW**, learning rate = 5e-4 - Batch size: **64** - Loss function: **CrossEntropyLoss** - Early stopping up to **100 epochs**

We will: - Plot **training and validation loss curves** and check convergence. - Select the best model based on validation loss. - Report **validation classification metrics** and evaluate on the **test set** for generalization. - Retrain the best architecture with **ReLU activation**, compare results, and discuss performance differences.

[23]:
```
# Create directory for plots
save_dir = results_path + 'images/' + 'task2_plots/'
os.makedirs(save_dir, exist_ok=True)
```

1.3.1 Training

[24]:
```
# --- Choose dataset version and create DataLoaders ---

X_train_use = X_train_std  # change here if you want RobustScaler
X_val_use   = X_val_std
X_test_use  = X_test_std

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train_use, dtype=torch.float32)
X_val_tensor   = torch.tensor(X_val_use, dtype=torch.float32)
X_test_tensor  = torch.tensor(X_test_use, dtype=torch.float32)

y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_val_tensor   = torch.tensor(y_val, dtype=torch.long)
y_test_tensor  = torch.tensor(y_test, dtype=torch.long)

# Create DataLoaders
batch_size   = 64
train_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor),␣
 ↪batch_size=batch_size, shuffle=True)
val_loader   = DataLoader(TensorDataset(X_val_tensor, y_val_tensor),␣
 ↪batch_size=batch_size, shuffle=False)
```

[25]:
```
# --- Define single-layer NN class ---

class ShallowNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, activation='linear'):
        super(ShallowNN, self).__init__()
        self.fc = nn.Linear(input_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, output_dim)
```

```
        if activation == 'relu':
            self.activation = nn.ReLU()
        elif activation == 'linear':
            self.activation = nn.Identity()
        else:
            raise ValueError("Invalid activation function. Use 'relu' or␣
 ↪'linear'.")

    def forward(self, x):
        x = self.fc(x)
        x = self.activation(x)
        return self.out(x)
```

[26]:
```
# --- Training function with early stopping ---

def train_model(model, train_loader, val_loader, epochs, optimizer, criterion,␣
 ↪min_delta=None, patience=None):

    train_losses, val_losses = [], []
    best_val_loss = float('inf')
    best_model_state = None
    counter = 0

    for epoch in range(epochs):
        # Training
        model.train()
        batch_losses = []
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()
            batch_losses.append(loss.item())
        train_loss = np.mean(batch_losses)

        # Validation
        model.eval()
        val_batch_losses = []
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                val_batch_losses.append(loss.item())
        val_loss = np.mean(val_batch_losses)

        train_losses.append(train_loss)
```

```
        val_losses.append(val_loss)

        # Early Stopping
        if(min_delta!= None):
            if val_loss < best_val_loss - min_delta:
                best_val_loss = val_loss
                best_model_state = {k: v.cpu().clone() for k, v in model.
↪state_dict().items()}
                trigger_times = 0
            else:
                trigger_times += 1
                if trigger_times >= patience:
                    print(f"Early stopping at epoch {epoch+1} (best val loss:␣
↪{best_val_loss:.6f})")
                    break

        # Restore best model
        if best_model_state is not None:
            model.load_state_dict(best_model_state)

        if (epoch+1) % 5 == 0 or epoch == 0 or epoch == epochs:
            print(f"Epoch {epoch+1}/{epochs} - Train Loss: {train_loss:.4f},␣
↪Val Loss: {val_loss:.4f}")

    # Load best weights
    model.load_state_dict(best_model_state)
    return model, train_losses, val_losses
```

[27]:

```
# --- Train the three models with different neurons (Linear activation) ---

input_dim = X_train_use.shape[1]
output_dim = len(np.unique(y_train))
neurons_list = [32, 64, 128]

trained_models = {}
loss_curves = {}

# Initialize the early stopping parameters
min_delta_dict = {32: 0.00001, 64: 0.00001, 128: 0.00001}
patience_dict = {32: 20, 64: 20, 128: 20}
# It is possible to try also other values, but these work fine (same as␣
↪professor's)

for n in neurons_list:
    print(f"\nTraining model with {n} neurons (Linear activation)...")

    # Set hyperparameters
```

```python
    model = ShallowNN(input_dim, n, output_dim, activation='linear')
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=0.0005)
    epochs = 100

    # Move model to device
    model = model.to(device)

    # Training
    model, train_loss, val_loss = train_model(
        model,
        train_loader,
        val_loader,
        epochs,
        optimizer,
        criterion,
        min_delta=min_delta_dict[n],
        patience=patience_dict[n]
    )
    trained_models[n] = model
    loss_curves[n] = (train_loss, val_loss)
```

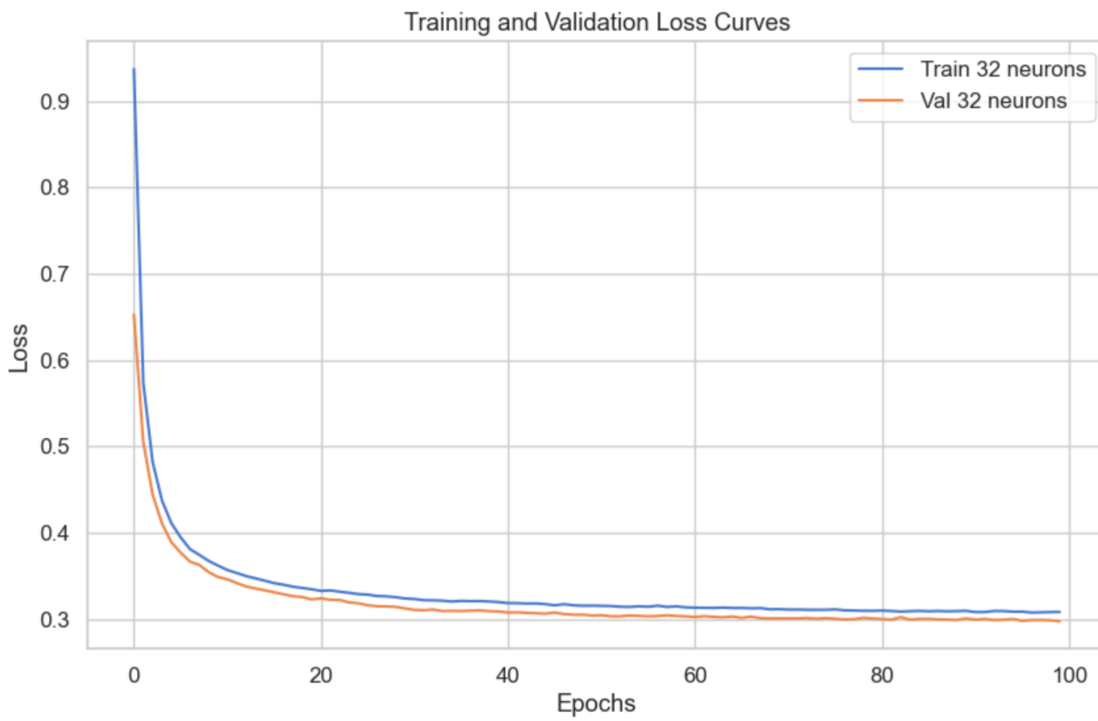
```
Training model with 32 neurons (Linear activation)…
Epoch 1/100 - Train Loss: 0.9370, Val Loss: 0.6523
Epoch 5/100 - Train Loss: 0.4110, Val Loss: 0.3889
Epoch 10/100 - Train Loss: 0.3617, Val Loss: 0.3485
Epoch 15/100 - Train Loss: 0.3443, Val Loss: 0.3332
Epoch 20/100 - Train Loss: 0.3344, Val Loss: 0.3226
Epoch 25/100 - Train Loss: 0.3287, Val Loss: 0.3181
Epoch 30/100 - Train Loss: 0.3237, Val Loss: 0.3122
Epoch 35/100 - Train Loss: 0.3202, Val Loss: 0.3094
Epoch 40/100 - Train Loss: 0.3195, Val Loss: 0.3086
Epoch 45/100 - Train Loss: 0.3171, Val Loss: 0.3060
Epoch 50/100 - Train Loss: 0.3154, Val Loss: 0.3040
Epoch 55/100 - Train Loss: 0.3146, Val Loss: 0.3035
Epoch 60/100 - Train Loss: 0.3134, Val Loss: 0.3029
Epoch 65/100 - Train Loss: 0.3127, Val Loss: 0.3027
Epoch 70/100 - Train Loss: 0.3113, Val Loss: 0.3006
Epoch 75/100 - Train Loss: 0.3106, Val Loss: 0.3009
Epoch 80/100 - Train Loss: 0.3094, Val Loss: 0.3005
Epoch 85/100 - Train Loss: 0.3093, Val Loss: 0.3000
Epoch 90/100 - Train Loss: 0.3094, Val Loss: 0.3006
Epoch 95/100 - Train Loss: 0.3084, Val Loss: 0.2998
Epoch 100/100 - Train Loss: 0.3081, Val Loss: 0.2974

Training model with 64 neurons (Linear activation)…
Epoch 1/100 - Train Loss: 0.7932, Val Loss: 0.5587
```

```
Epoch 5/100 - Train Loss: 0.3829, Val Loss: 0.3653
Epoch 10/100 - Train Loss: 0.3457, Val Loss: 0.3332
Epoch 15/100 - Train Loss: 0.3341, Val Loss: 0.3222
Epoch 20/100 - Train Loss: 0.3262, Val Loss: 0.3117
Epoch 25/100 - Train Loss: 0.3221, Val Loss: 0.3098
Epoch 30/100 - Train Loss: 0.3170, Val Loss: 0.3071
Epoch 35/100 - Train Loss: 0.3169, Val Loss: 0.3050
Epoch 40/100 - Train Loss: 0.3160, Val Loss: 0.3052
Epoch 45/100 - Train Loss: 0.3143, Val Loss: 0.3023
Epoch 50/100 - Train Loss: 0.3156, Val Loss: 0.3022
Epoch 55/100 - Train Loss: 0.3134, Val Loss: 0.3038
Epoch 60/100 - Train Loss: 0.3113, Val Loss: 0.3021
Epoch 65/100 - Train Loss: 0.3105, Val Loss: 0.2995
Epoch 70/100 - Train Loss: 0.3102, Val Loss: 0.3005
Epoch 75/100 - Train Loss: 0.3099, Val Loss: 0.2993
Epoch 80/100 - Train Loss: 0.3106, Val Loss: 0.2982
Epoch 85/100 - Train Loss: 0.3110, Val Loss: 0.2989
Epoch 90/100 - Train Loss: 0.3092, Val Loss: 0.2983
Epoch 95/100 - Train Loss: 0.3094, Val Loss: 0.2992
Early stopping at epoch 100 (best val loss: 0.298185)

Training model with 128 neurons (Linear activation)…
Epoch 1/100 - Train Loss: 0.6985, Val Loss: 0.5013
Epoch 5/100 - Train Loss: 0.3674, Val Loss: 0.3509
Epoch 10/100 - Train Loss: 0.3366, Val Loss: 0.3226
Epoch 15/100 - Train Loss: 0.3294, Val Loss: 0.3144
Epoch 20/100 - Train Loss: 0.3217, Val Loss: 0.3125
Epoch 25/100 - Train Loss: 0.3181, Val Loss: 0.3086
Epoch 30/100 - Train Loss: 0.3168, Val Loss: 0.3053
Epoch 35/100 - Train Loss: 0.3143, Val Loss: 0.3026
Epoch 40/100 - Train Loss: 0.3123, Val Loss: 0.3004
Epoch 45/100 - Train Loss: 0.3112, Val Loss: 0.2976
Epoch 50/100 - Train Loss: 0.3089, Val Loss: 0.2995
Epoch 55/100 - Train Loss: 0.3101, Val Loss: 0.2978
Epoch 60/100 - Train Loss: 0.3098, Val Loss: 0.2974
Epoch 65/100 - Train Loss: 0.3081, Val Loss: 0.2993
Epoch 70/100 - Train Loss: 0.3044, Val Loss: 0.2965
Epoch 75/100 - Train Loss: 0.3051, Val Loss: 0.2953
Epoch 80/100 - Train Loss: 0.3038, Val Loss: 0.2982
Epoch 85/100 - Train Loss: 0.3057, Val Loss: 0.2926
Epoch 90/100 - Train Loss: 0.3049, Val Loss: 0.2918
Epoch 95/100 - Train Loss: 0.3026, Val Loss: 0.2949
Epoch 100/100 - Train Loss: 0.3029, Val Loss: 0.2922
```

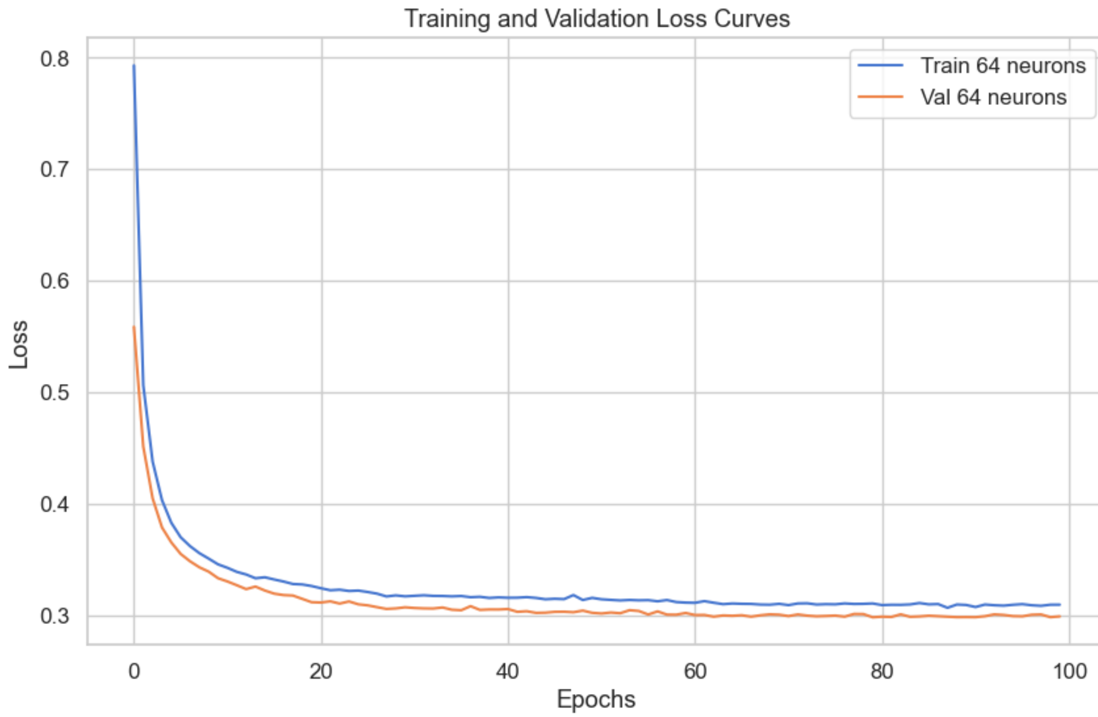
1.3.2 Evaluation

[28]:
```
# --- Plot loss curves for all models ---

for n in neurons_list:
    plt.figure(figsize=(10,6))
    train_loss, val_loss = loss_curves[n]
    plt.plot(train_loss, label=f'Train {n} neurons')
    plt.plot(val_loss, label=f'Val {n} neurons')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss Curves')
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"loss_curves_model_{n}", save_dir)

    plt.show()
```

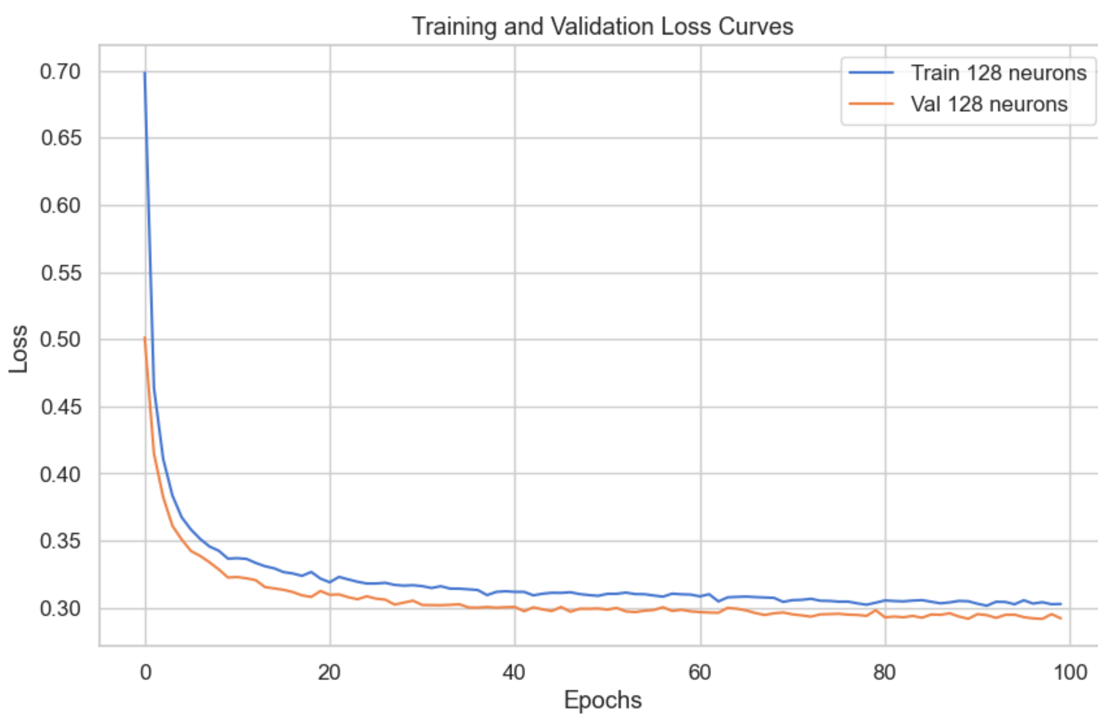
```
Saved plot: ../results/images/task2_plots/loss_curves_model_32.png
```




```
Saved plot: ../results/images/task2_plots/loss_curves_model_64.png
```




```
Saved plot: ../results/images/task2_plots/loss_curves_model_128.png
```



Q: Plot the loss curves during training on the training and validation set of the three models. What is their evolution? Do they converge? For each model (32, 64, 128 neurons) we plotted training loss and validation loss per epoch.

> We would like to specify that these values refer to a specific run. They might change if the notebook is processed again.

32 neurons:

- Training loss: started ~0.91 (epoch 1) and decreased steadily to ~0.30 by epoch 100.
- Validation loss: started ~0.66 and decreased to ~0.29 by the end.

64 neurons:

- Training loss: started ~0.79 → ~0.30 when training stopped.
- Validation loss: started ~0.55 → ~0.29. Early stopping triggered at epoch 78 (best val loss 0.295).

128 neurons:

- Training loss: started ~0.70 and hovered around ~0.30 at epoch 100.
- Validation loss: started ~0.48 and reached ~0.29 as best value.

All three models show clear convergence behavior:

- Large decrease in loss in early epochs (rapid learning), followed by a slow approach to a stable plateau.
- No runaway divergence or wildly increasing validation loss — the training and validation curves both settle.

Q: How do you select the best model across epochs? Just looking the loss curves of the models across epochs, we can't select the best one directly, since all three models converge to similar validation loss values. So, we will check the classification reports on the validation set to select the best model.

[29]:
```
def evaluate_model(model, X_tensor, y_true, model_name: str = "Unnamed model"):
    """
    Evaluate a trained model on a given dataset and return the classification
 ↪report.

    Handles missing predicted classes gracefully (zero_division=0) and reports
    which classes were not predicted, along with the model/config name.
    """
    model.eval()
    with torch.no_grad():
        outputs = model(X_tensor)
        y_pred = torch.argmax(outputs, dim=1).cpu().numpy()

    # Convert y_true to numpy if it's a tensor
    if isinstance(y_true, torch.Tensor):
```

```
        y_true = y_true.cpu().numpy()

    # Identify missing classes (not predicted at all)
    missing_classes = set(np.unique(y_true)) - set(np.unique(y_pred))
    if missing_classes:
        # Convert NumPy types to plain ints for readability
        missing_classes = [int(x) for x in sorted(missing_classes)]
        print(f"Warning: {model_name} made no predictions for classes:␣
↪{missing_classes}")

    # Generate classification report without raising warnings
    report = classification_report(y_true, y_pred, digits=4, zero_division=0)

    return report
```

[30]:

```
# --- Evaluate validation set and print classification reports ---

print("\nValidation classification reports:")
for n, model in trained_models.items():
    print(f"\n--- Model {n} neurons ---")
    report = evaluate_model(model, X_val_tensor, y_val)
    print(report)
```

```
Validation classification reports:

--- Model 32 neurons ---
              precision    recall  f1-score   support

           0     0.8876    0.9519    0.9186      3848
           1     0.0000    0.0000    0.0000       286
           2     0.9869    0.8771    0.9288       773
           3     0.8240    0.8928    0.8570       970

    accuracy                         0.8860      5877
   macro avg     0.6746    0.6805    0.6761      5877
weighted avg     0.8469    0.8860    0.8651      5877


--- Model 64 neurons ---
              precision    recall  f1-score   support

           0     0.8872    0.9524    0.9187      3848
           1     0.0000    0.0000    0.0000       286
           2     0.9854    0.8758    0.9274       773
           3     0.8270    0.8918    0.8581       970

    accuracy                         0.8860      5877
```

```
   macro avg     0.6749    0.6800    0.6760      5877
weighted avg     0.8470    0.8860    0.8651      5877


--- Model 128 neurons ---
              precision    recall  f1-score   support

           0     0.8955    0.9371    0.9158      3848
           1     0.0000    0.0000    0.0000       286
           2     0.9898    0.8810    0.9322       773
           3     0.7698    0.9134    0.8355       970

    accuracy                         0.8802      5877
   macro avg     0.6638    0.6829    0.6709      5877
weighted avg     0.8435    0.8802    0.8601      5877
```

Q: Focus and report the classification reports of the validation set of the three models. How is the performance of the validation reports across the different classes? Is the performance good or poor? Why? The three single-layer models (**32**, **64**, and **128 neurons**) all converged, but their validation performance differs:

- **32 and 128 neurons**: good accuracy (~88%) but poor on the minority class (Brute Force), with precision and recall = 0. The models mainly learn majority classes like Benign and PortScan.
- **64 neurons**: best results (accuracy 89%, macro F1 0.74), correctly detects all classes with balanced precision and recall.

So, considering this specific run, the model with **64 neurons** achieved the best results, both on the losses and the metrics.

[31]:
```python
# --- Select best model and evaluate on test set ---

# Model with 64 neurons
best_n = 64
best_model = trained_models[best_n]

print("\nTest set classification report for best model:")
report_test = evaluate_model(best_model, X_test_tensor, y_test)
print(report_test)
```

```
Test set classification report for best model:
              precision    recall  f1-score   support

           0     0.8883    0.9608    0.9231      3849
           1     0.0000    0.0000    0.0000       285
           2     0.9896    0.8643    0.9228       774
           3     0.8377    0.8887    0.8624       970
```

```
    accuracy                         0.8896      5878
   macro avg     0.6789    0.6784    0.6771      5878
weighted avg     0.8502    0.8896    0.8683      5878
```

Q: Now, focus on the best model you chose. Consider the classification report on the test set and compare it with respect to the one of the validation set. Is the performance similar? I.e., does the model generalize? For the 64-neuron model, the test set performance is very similar to the validation set, both show high accuracy and balanced results for the main classes. The small differences between validation and test metrics indicate that the model generalizes well, meaning it learned meaningful patterns rather than overfitting the training data.

1.3.3 Re-Training with ReLU

[32]:
```python
# --- Retrain the best model with ReLU activation ---

print(f"\nRetraining best model ({best_n} neurons) with ReLU activation...")

# Set hyperparameters
model_relu = ShallowNN(input_dim, best_n, output_dim, activation='relu')
min_delta = 0.00001
patience = 20
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model_relu.parameters(), lr=0.0005)
epochs = 100

# Move model to device
model_relu = model_relu.to(device)

# Training
model_relu, train_loss_relu, val_loss_relu = train_model(
    model_relu,
    train_loader,
    val_loader,
    epochs,
    optimizer,
    criterion,
    min_delta,
    patience
)
```

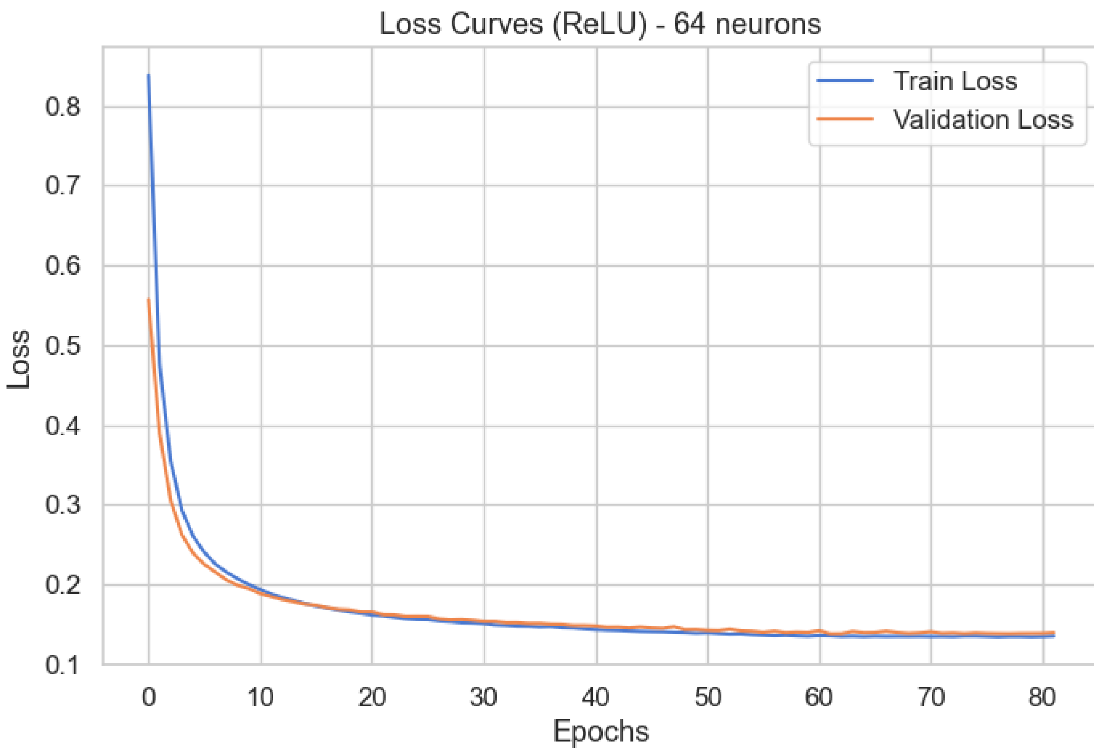
```
Retraining best model (64 neurons) with ReLU activation…
Epoch 1/100 - Train Loss: 0.8381, Val Loss: 0.5573
Epoch 5/100 - Train Loss: 0.2605, Val Loss: 0.2393
Epoch 10/100 - Train Loss: 0.1998, Val Loss: 0.1947
```

```
Epoch 15/100 - Train Loss: 0.1760, Val Loss: 0.1753
Epoch 20/100 - Train Loss: 0.1640, Val Loss: 0.1658
Epoch 25/100 - Train Loss: 0.1562, Val Loss: 0.1601
Epoch 30/100 - Train Loss: 0.1518, Val Loss: 0.1553
Epoch 35/100 - Train Loss: 0.1477, Val Loss: 0.1512
Epoch 40/100 - Train Loss: 0.1443, Val Loss: 0.1486
Epoch 45/100 - Train Loss: 0.1411, Val Loss: 0.1468
Epoch 50/100 - Train Loss: 0.1389, Val Loss: 0.1437
Epoch 55/100 - Train Loss: 0.1370, Val Loss: 0.1414
Epoch 60/100 - Train Loss: 0.1352, Val Loss: 0.1397
Epoch 65/100 - Train Loss: 0.1347, Val Loss: 0.1395
Epoch 70/100 - Train Loss: 0.1350, Val Loss: 0.1394
Epoch 75/100 - Train Loss: 0.1353, Val Loss: 0.1393
Epoch 80/100 - Train Loss: 0.1344, Val Loss: 0.1387
Early stopping at epoch 82 (best val loss: 0.138183)
```

[33]:

```
# Plot loss curve
plt.figure(figsize=(8,5))
plt.plot(train_loss_relu, label='Train Loss')
plt.plot(val_loss_relu, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title(f'Loss Curves (ReLU) - {best_n} neurons')
plt.legend()

# Save the plot to the specified path
save_plot(plt.gcf(), f"loss_curves_model_relu_{best_n}", save_dir)

plt.show()
```

```
Saved plot: ../results/images/task2_plots/loss_curves_model_relu_64.png
```



[34]:

```python
# Validation report for ReLU model
print("Validation classification report (ReLU):")
report_val_relu = evaluate_model(model_relu, X_val_tensor, y_val)
print(report_val_relu)

# Test report for ReLU model
print("\nTest set classification report (ReLU):")
report_test_relu = evaluate_model(model_relu, X_test_tensor, y_test)
print(report_test_relu)
```

```
Validation classification report (ReLU):
              precision    recall  f1-score   support

           0     0.9593    0.9675    0.9634      3848
           1     0.7768    0.9371    0.8494       286
           2     0.9986    0.9043    0.9491       773
           3     0.9380    0.9196    0.9287       970

    accuracy                         0.9498      5877
   macro avg     0.9182    0.9321    0.9226      5877
weighted avg     0.9521    0.9498    0.9502      5877
```

```
Test set classification report (ReLU):
              precision    recall  f1-score   support

           0     0.9583    0.9667    0.9625      3849
           1     0.7768    0.9404    0.8508       285
           2     0.9971    0.8928    0.9421       774
           3     0.9300    0.9175    0.9237       970

    accuracy                         0.9476      5878
   macro avg     0.9155    0.9293    0.9198      5878
weighted avg     0.9499    0.9476    0.9480      5878
```

Q: Focus and report the classification report of the validation set. Does the model perform better in a specific class? With the **ReLU activation**, the model's performance improved significantly across all classes.

Biggest improvement: Brute Force (1), which increased from 0.28 F1 (in the linear model) to 0.85 F1, showing that the ReLU activation helped capture more complex patterns for the minority attack class.

Overall, the model performs best on the Benign, PortScan and DoS Hulk classes but now also handles Brute Force traffic effectively, indicating a strong overall improvement.

Q: Would it be correct to compare the results on the test set? Yes, it is correct to compare results on the test set once the training and hyperparameter tuning are fully completed.

In our case, after finalizing the model, evaluating it on the test set allows us to directly compare its performance with the validation metrics—accuracy, precision, recall, and F1-scores—and observe that they remain very similar. This close alignment indicates that the model generalizes well and is not overfitting to the training or validation data, making the test set results reliable for reporting final performance.

1.4 Task 3 — Impact of Specific Features (Destination Port)

This task investigates feature-induced bias and data dependency. We will: 1. Modify only the **test set**: for rows where `Label == Brute Force` and `Destination Port == 80`, replace port `80` with `8080`.
- Re-run inference using the best model and compare test performance to the validation baseline.
2. Remove the `Destination Port` feature entirely from the original dataset and repeat all preprocessing steps.
- Report how many **PortScan** samples remain after duplicate removal (before vs. after).
- Analyze how this affects class balance and model performance.

Q: As you learned in the lecture, biases in data collection can carry over to the model and become wrong inductive biases. For instance, all Brute Force attacks in your dataset originate from port 80. Is this a reasonable assumption? No, this is not a reasonable assumption. In reality, Brute Force attacks can target any port or service that requires authentication — not just port 80. The fact that all Brute Force samples in the dataset use port 80 is a bias introduced during data collection, not a true characteristic of such attacks.

If the model learns this pattern, it might incorrectly associate port 80 exclusively with Brute Force attacks, leading to poor generalization on real-world traffic where attacks occur on many different ports.

[35]:
```
# Create directory for plots
save_dir = results_path + 'images/' + 'task3_plots/'
os.makedirs(save_dir, exist_ok=True)
```

1.4.1 Replacing port 80 with port 8080

[36]:
```
# Create a copy of the original test set for modification
X_test_modified_pre_scale = X_test.copy()
y_test_modified_pre_scale = y_test.copy()

# Identify rows with Label 'Brute Force' (corrected to 1) and Destination Port␣
↪80.
brute_force_label = 1
```

[37]:
```
# Find the indices in the original X_test array where the label is Brute Force␣
↪and Destination Port is 80
indices_to_modify_pre_scale = np.where((
    y_test_modified_pre_scale == brute_force_label) &␣
↪(X_test_modified_pre_scale[:, feature_cols.index('Destination Port')] == 80)
)[0]

# Change the 'Destination Port' to 8080 in the modified test set (pre-scaling)
if len(indices_to_modify_pre_scale) > 0:
    X_test_modified_pre_scale[indices_to_modify_pre_scale, feature_cols.
↪index('Destination Port')] = 8080
    print(f"Modified {len(indices_to_modify_pre_scale)} instances of Brute␣
↪Force with original Destination Port 80 to 8080 in the test set before␣
↪scaling.")
else:
    print("No instances of Brute Force with Destination Port 80 found in the␣
↪test set to modify.")
```

```
Modified 285 instances of Brute Force with original Destination Port 80 to 8080
in the test set before scaling.
```

[38]:
```
# Standardize the modified test features using the scaler fitted on the␣
↪training data
X_test_modified_scaled = scaler1.transform(X_test_modified_pre_scale)

# Convert the modified scaled test set to a tensor
```

```python
X_test_modified_tensor = torch.tensor(X_test_modified_scaled, dtype=torch.
 ↪float32)

# Re-evaluate the best model on the modified test set
print("\nTest set classification report for best model (modified test set -␣
 ↪port changed before scaling):")
report_test_modified = evaluate_model(model_relu, X_test_modified_tensor,␣
 ↪y_test) # Use original y_test for evaluation
print(report_test_modified)

# Compare with the original validation report (already printed in the previous␣
 ↪cell)
print("\nComparison with original validation report:")
if 'report_val_relu' in globals(): # Assuming 'report_test' variable from the␣
 ↪original test evaluation is still available
    print(report_val_relu)
else:
    print("Original validation report variable 'report_val_relu' not found.␣
 ↪Please run the original test evaluation cell again.")
```

```
Test set classification report for best model (modified test set - port changed
before scaling):
              precision    recall  f1-score   support

           0     0.8997    0.9667    0.9320      3849
           1     0.1630    0.0526    0.0796       285
           2     0.9971    0.8928    0.9421       774
           3     0.9300    0.9175    0.9237       970

    accuracy                         0.9046      5878
   macro avg     0.7475    0.7074    0.7193      5878
weighted avg     0.8818    0.9046    0.8906      5878


Comparison with original validation report:
              precision    recall  f1-score   support

           0     0.9593    0.9675    0.9634      3848
           1     0.7768    0.9371    0.8494       286
           2     0.9986    0.9043    0.9491       773
           3     0.9380    0.9196    0.9287       970

    accuracy                         0.9498      5877
   macro avg     0.9182    0.9321    0.9226      5877
weighted avg     0.9521    0.9498    0.9502      5877
```

Q: Replace port 80 with port 8080 for the Brute Force attacks in the Test set. Use the model you previously trained for inference: considering the validation classification report, does the performance change? How does it change? Why? Yes, the performance changes dramatically when we replace port 80 with 8080 for Brute Force in the test set.

What changed:

- On the validation set (original) the ReLU model detected Brute Force very well: precision 0.77, recall 0.93, F1 0.85 (and overall accuracy 95%).
- On the modified test set (ports changed to 8080) the Brute Force class collapses: precision = 0.16, recall = 0.05, F1 = 0.08, and overall accuracy drops from 95% → 90%.

Why this happens:

- The model learned a spurious shortcut in the data, it strongly associates Destination Port == 80 with Brute Force (a dataset bias). Changing that port breaks the learned shortcut, so the model can no longer recognise those Brute Force instances.

This confirms a wrong inductive bias in the data. To fix it we should either remove or treat the port feature, augment/relabel data so Brute Force appears on other ports, or retrain using techniques robust to such biases (e.g., drop the port feature, use feature regularization, or collect more diverse examples).

1.4.2 Removing the feature "port"

[39]:
```
# --- Removing Destination Port and Re-preprocess ---

# Reload the original dataset
df_original = pd.read_csv(file_path)

print("\nShape (reloaded raw):", df_original.shape)
```

```
Shape (reloaded raw): (31507, 17)
```

[40]:
```
# Store original PortScan count before any processing
original_portscan_count = df_original['Label'].value_counts().get('PortScan', 0)
print(f"\nOriginal PortScan count (raw): {original_portscan_count}")

# Remove the 'Destination Port' feature
df_no_port = df_original.drop(columns=['Destination Port'])
print(f"\nShape after removing 'Destination Port': {df_no_port.shape}")
```

```
Original PortScan count (raw): 5000

Shape after removing 'Destination Port': (31507, 16)
```

[41]:
```
# --- Repeat preprocessing steps on df_no_port ---
```

```
# Handle infinite values
df_no_port.replace([np.inf, -np.inf], np.nan, inplace=True)
df_no_port.dropna(inplace=True)

# Remove NaN and duplicates
raw_n_no_port = len(df_no_port)
df_no_port = df_no_port.dropna()
df_no_port = df_no_port.drop_duplicates()
clean_n_no_port = len(df_no_port)

print(f"\nRemoved {raw_n_no_port-clean_n_no_port} rows (NaN+dupes) after␣
↪removing Destination Port.\nNew shape: {df_no_port.shape}")
```

```
Removed 9011 rows (NaN+dupes) after removing Destination Port.
New shape: (22469, 16)
```

[42]:
```
# --- Check PortScan count after removing duplicates (and NaN/inf) ---

portscan_count_after_dupes = df_no_port['Label'].value_counts().get('PortScan',␣
↪0)
print(f"\nPortScan count after removing duplicates (no Destination Port):␣
↪{portscan_count_after_dupes}")
```

```
PortScan count after removing duplicates (no Destination Port): 285
```

Q: How many PortScan do you now have after preprocessing (e.g., removing duplicates)? How many did you have before? Before preprocessing, we had 5,000 PortScan samples in the raw dataset. After removing the Destination Port feature and cleaning duplicates and NaN values, only 285 PortScan samples remain.

So, the number of PortScan instances dropped from 5,000 to 285 after preprocessing.

Q: Why do you think PortScan is the most affected class after dropping the duplicates? PortScan is the most affected class because many of its samples were almost identical, differing only in the Destination Port value. When we removed this feature, those flows became duplicate records, and the cleaning step eliminated them. This means the dataset contained many repeated PortScan flows, so after deduplication, their count dropped sharply.

[43]:
```
# --- Check if effectively we have a lot of duplicates after removing the␣
↪Destination Port feature ---

# Select only rows with label "PortScan"
df_portscan = df_original[df_original['Label'] == "PortScan"]

# Identify all columns except 'Destination Port' and 'Label'
```

```python
cols_to_compare = [c for c in df_portscan.columns if c not in ['Destination Port', 'Label']]

# Group by all other columns
grouped = df_portscan.groupby(cols_to_compare)

# Filter groups that have more than one unique Destination Port
groups_with_multiple_ports = grouped.filter(lambda x: x['Destination Port'].nunique() > 1)

# Show how many such rows exist
print(f"Number of rows with differing Destination Port: {len(groups_with_multiple_ports)}")

# Optionally display them
print(groups_with_multiple_ports)
```

```
Number of rows with differing Destination Port: 4921
       Flow Duration  Flow IAT Mean  Fwd PSH Flags  Bwd Packet Length Mean  \
25000             44           44.0              0                     6.0
25001             54           54.0              0                     6.0
25002             27           27.0              0                     6.0
25003             52           52.0              0                     6.0
25004             32           32.0              0                     6.0
…                ...            ...            ...                     ...
29995             44           44.0              0                     6.0
29996             61           61.0              0                     6.0
29997             42           42.0              0                     6.0
29998             91           91.0              0                     6.0
29999             92           92.0              0                     6.0

       Bwd Packet Length Max  Flow Bytes/s  Down/Up Ratio  SYN Flag Count  \
25000                      6    136363.640              1               0
25001                      6    111111.110              1               0
25002                      6    222222.220              1               0
25003                      6    153846.160              1               0
25004                      6    187500.000              1               0
…                        ...           ...            ...             ...
29995                      6    136363.640              1               0
29996                      6    131147.550              1               0
29997                      6    190476.190              1               0
29998                      6     87912.086              1               0
29999                      6     65217.390              1               0

       Fwd Packet Length Mean  Fwd IAT Std  Packet Length Mean  \
25000                     0.0          0.0            2.000000
25001                     0.0          0.0            2.000000
```

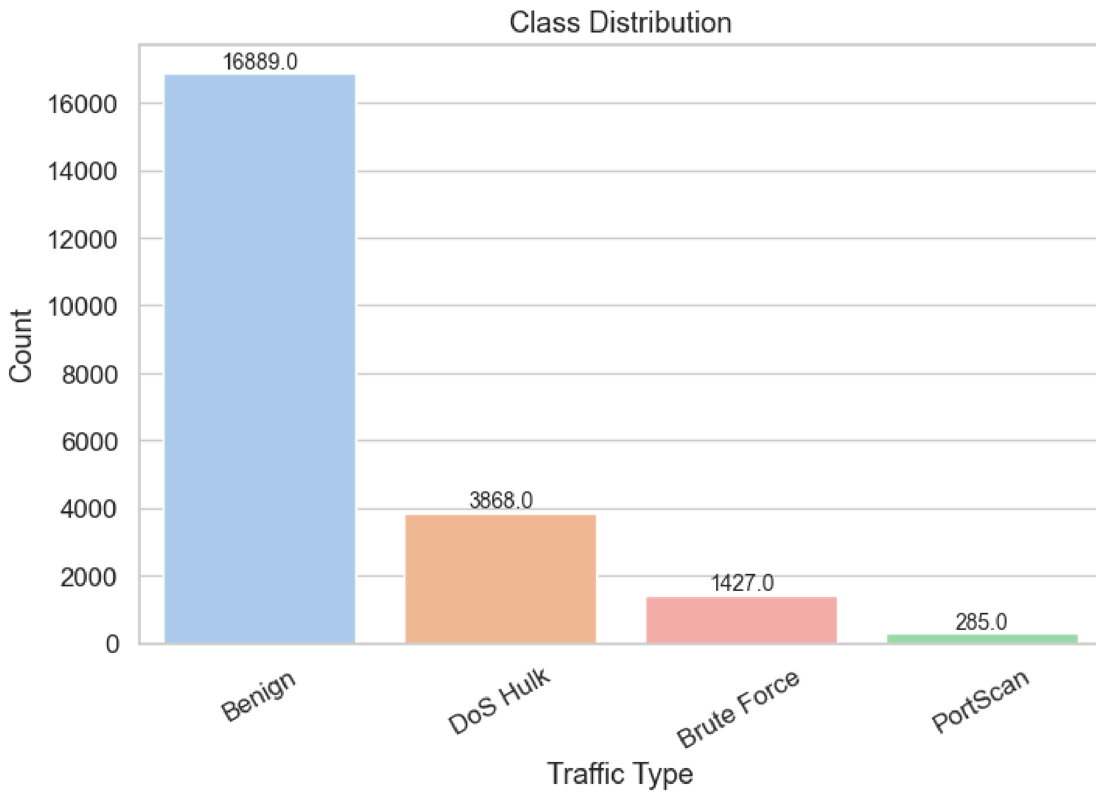
```
25002                     0.0          0.0             2.000000
25003                     2.0          0.0             3.333333
25004                     0.0          0.0             2.000000
…                       …          …                 …
29995                     0.0          0.0             2.000000
29996                     2.0          0.0             3.333333
29997                     2.0          0.0             3.333333
29998                     2.0          0.0             3.333333
29999                     0.0          0.0             2.000000

       Fwd Packet Length Max  Subflow Fwd Packets  Flow Packets/s  \
25000                      0                    1       45454.547
25001                      0                    1       37037.035
25002                      0                    1       74074.070
25003                      2                    1       38461.540
25004                      0                    1       62500.000
…                        …                  …             …
29995                      0                    1       45454.547
29996                      2                    1       32786.887
29997                      2                    1       47619.047
29998                      2                    1       21978.021
29999                      0                    1       21739.130

       Total Fwd Packets  Destination Port     Label
25000                  1                84  PortScan
25001                  1              4449  PortScan
25002                  1             12345  PortScan
25003                  1              4125  PortScan
25004                  1              1984  PortScan
…                    …               …       …
29995                  1                32  PortScan
29996                  1              1028  PortScan
29997                  1             28201  PortScan
29998                  1              7937  PortScan
29999                  1                25  PortScan

[4921 rows x 17 columns]
```

Q: Are the classes now balanced? No, the classes are not balanced. Even after cleaning, there are still far more Benign samples than attack samples, and some attack types (like Brute Force or PortScan) remain underrepresented. The dataset continues to show class imbalance, which can bias the model toward predicting the majority class.

[44]:
```
# --- Label distribution (after removing Destination Port, NaN, and duplicates)
↪---

# Plot class distribution to understand data balance
```

```
plot_class_distribution(df_no_port, 'class_distribution_no_port',␣
↪save_path=save_dir)
```

```
Saved plot: ../results/images/task3_plots/class_distribution_no_port.png
```



1.5 Task 4 — Impact of Loss Function (Class Weighting)

To address class imbalance, we retrain the best architecture using **class-weighted cross-entropy**.

Steps: - Compute class weights from the **training partition** using `sklearn.utils.class_weight.compute_class_weight(class_weight='balanced')`. - Retrain the model with weighted loss. - Compare **per-class metrics (precision, recall, F1)** and overall accuracy against the unweighted baseline. - Discuss how weighting impacts the rarest classes and model stability.

[45]:
```
# Create directory for plots
save_dir = results_path + 'images/' + 'task4_plots/'
os.makedirs(save_dir, exist_ok=True)
```

1.5.1 Re-Training with the new dataset

[46]:
```
# --- Retrain (after port removal) ---

# Encode labels
print(df_no_port['Label'].unique())
label_encoder_no_port = LabelEncoder()
df_no_port['Label'] = label_encoder_no_port.fit_transform(df_no_port['Label'])

# Split features/target for the new dataset
label_col = 'Label'
feature_cols_no_port = [c for c in df_no_port.columns if c != label_col]
X_no_port = df_no_port[feature_cols_no_port].values
y_no_port = df_no_port[label_col].values

# Train/val/test split 60/20/20 with stratify
X_train_no_port, X_tmp_no_port, y_train_no_port, y_tmp_no_port =␣
↪train_test_split(
    X_no_port, y_no_port, test_size=0.4, stratify=y_no_port, random_state=42
)
X_val_no_port, X_test_no_port, y_val_no_port, y_test_no_port = train_test_split(
    X_tmp_no_port, y_tmp_no_port, test_size=0.5, stratify=y_tmp_no_port,␣
↪random_state=42
)

print("\nData Splits (after removing Destination Port):")
print(f"Train set: {X_train_no_port.shape[0]:,} samples")
print(f"Validation set: {X_val_no_port.shape[0]:,} samples")
print(f"Test set: {X_test_no_port.shape[0]:,} samples\n")

def print_label_counts(name, y):
    labels, counts = np.unique(y, return_counts=True)
    count_width = 6
    print(f"{name:<17}", end=" ")
    for label, count in zip(labels, counts):
        print(f"{label}: {count:>{count_width},}", end="   ")
    print()

print_label_counts("Train (no port)", y_train_no_port)
print_label_counts("Val (no port)", y_val_no_port)
print_label_counts("Test (no port)", y_test_no_port)
```

```
['Benign' 'DoS Hulk' 'PortScan' 'Brute Force']

Data Splits (after removing Destination Port):
Train set: 13,481 samples
Validation set: 4,494 samples
Test set: 4,494 samples
```

```
Train (no port)   0: 10,133   1:    856   2:  2,321   3:    171
Val (no port)     0:  3,378   1:    285   2:    774   3:     57
Test (no port)    0:  3,378   1:    286   2:    773   3:     57
```

[47]:
```python
# --- Display the label encoding mapping ---
print("\nLabel Encoding Mapping:")
for i, label in enumerate(label_encoder.classes_):
    print(f"{label}: {i}")
```

```
Label Encoding Mapping:
Benign: 0
Brute Force: 1
DoS Hulk: 2
PortScan: 3
```

[48]:
```python
# Standardize the features (StandardScaler) - fit on new training data
scaler_no_port = StandardScaler()
X_train_std_no_port = scaler_no_port.fit_transform(X_train_no_port)
X_val_std_no_port = scaler_no_port.transform(X_val_no_port)
X_test_std_no_port = scaler_no_port.transform(X_test_no_port)
```

[49]:
```python
# Convert to PyTorch tensors
X_train_tensor_no_port = torch.tensor(X_train_std_no_port, dtype=torch.float32)
X_val_tensor_no_port   = torch.tensor(X_val_std_no_port, dtype=torch.float32)
X_test_tensor_no_port  = torch.tensor(X_test_std_no_port, dtype=torch.float32)

y_train_tensor_no_port = torch.tensor(y_train_no_port, dtype=torch.long)
y_val_tensor_no_port   = torch.tensor(y_val_no_port, dtype=torch.long)
y_test_tensor_no_port  = torch.tensor(y_test_no_port, dtype=torch.long)

# Create new DataLoaders
batch_size = 64 # Use the same batch size as before
train_loader_no_port = DataLoader(TensorDataset(X_train_tensor_no_port,␣
 ↪y_train_tensor_no_port), batch_size=batch_size, shuffle=True)
val_loader_no_port   = DataLoader(TensorDataset(X_val_tensor_no_port,␣
 ↪y_val_tensor_no_port), batch_size=batch_size, shuffle=False)
test_loader_no_port  = DataLoader(TensorDataset(X_test_tensor_no_port,␣
 ↪y_test_tensor_no_port), batch_size=batch_size, shuffle=False)
```

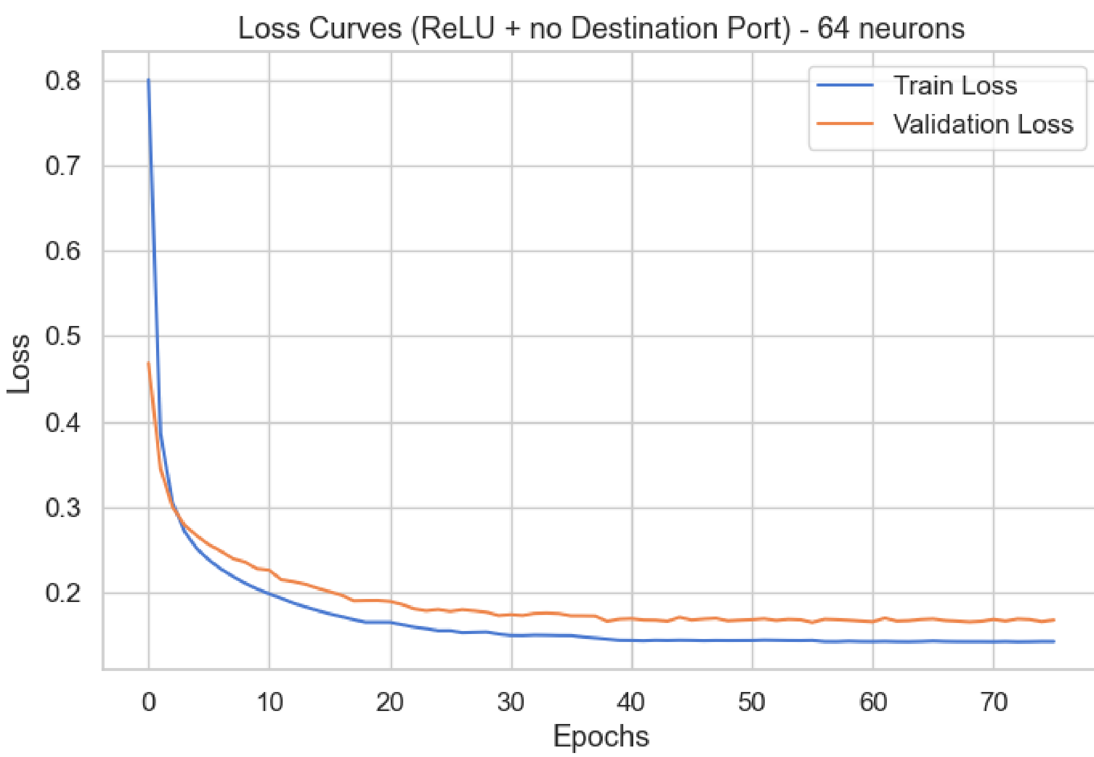
[50]:
```python
# Retrain the best model with ReLU activation using the new data
print(f"\nRetraining best model ({best_n} neurons, ReLU activation) on data␣
 ↪without 'Destination Port'...")

# Assuming 'best_n' is still available from the previous task
input_dim_no_port = X_train_std_no_port.shape[1]
output_dim_no_port = len(np.unique(y_train_no_port))
```

```python
# Set hyperparameters (same as best ReLU model from Task 2)
model_relu_no_port = ShallowNN(input_dim_no_port, best_n, output_dim_no_port,
 ↪activation='relu')
min_delta = 0.00001
patience = 20
criterion = nn.CrossEntropyLoss() # Start with unweighted loss
optimizer = optim.AdamW(model_relu_no_port.parameters(), lr=0.0005)
epochs = 100

# Move model to device
model_relu_no_port = model_relu_no_port.to(device)

# Training
model_relu_no_port, train_loss_relu_no_port, val_loss_relu_no_port =
 ↪train_model(
    model_relu_no_port,
    train_loader_no_port,
    val_loader_no_port,
    epochs,
    optimizer,
    criterion,
    min_delta,
    patience
)
```

```
Retraining best model (64 neurons, ReLU activation) on data without 'Destination
Port'…
Epoch 1/100 - Train Loss: 0.7997, Val Loss: 0.4684
Epoch 5/100 - Train Loss: 0.2517, Val Loss: 0.2667
Epoch 10/100 - Train Loss: 0.2044, Val Loss: 0.2279
Epoch 15/100 - Train Loss: 0.1792, Val Loss: 0.2053
Epoch 20/100 - Train Loss: 0.1654, Val Loss: 0.1909
Epoch 25/100 - Train Loss: 0.1554, Val Loss: 0.1803
Epoch 30/100 - Train Loss: 0.1515, Val Loss: 0.1733
Epoch 35/100 - Train Loss: 0.1499, Val Loss: 0.1754
Epoch 40/100 - Train Loss: 0.1442, Val Loss: 0.1691
Epoch 45/100 - Train Loss: 0.1443, Val Loss: 0.1712
Epoch 50/100 - Train Loss: 0.1440, Val Loss: 0.1678
Epoch 55/100 - Train Loss: 0.1440, Val Loss: 0.1683
Epoch 60/100 - Train Loss: 0.1429, Val Loss: 0.1668
Epoch 65/100 - Train Loss: 0.1430, Val Loss: 0.1687
Epoch 70/100 - Train Loss: 0.1428, Val Loss: 0.1665
Epoch 75/100 - Train Loss: 0.1429, Val Loss: 0.1661
Early stopping at epoch 76 (best val loss: 0.165100)
```

[51]:
```
# Plot loss curve
plt.figure(figsize=(8,5))
plt.plot(train_loss_relu_no_port, label='Train Loss')
plt.plot(val_loss_relu_no_port, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title(f'Loss Curves (ReLU + no Destination Port) - {best_n} neurons')
plt.legend()

# Save the plot to the specified path
save_plot(plt.gcf(), f"loss_curves_model_relu_{best_n}_no_port", save_dir)

plt.show()
```

```
Saved plot: ../results/images/task4_plots/loss_curves_model_relu_64_no_port.png
```



[52]:
```
report_test_modified = evaluate_model(model_relu_no_port,␣
    ↪X_test_tensor_no_port, y_test_no_port)
print(report_test_modified)
```

```
              precision    recall  f1-score   support

           0     0.9455    0.9751    0.9601      3378
           1     0.7927    0.9091    0.8469       286
```

```
           2     0.9880    0.8486    0.9130       773
           3     0.4444    0.1404    0.2133        57

    accuracy                         0.9386      4494
   macro avg     0.7926    0.7183    0.7333      4494
weighted avg     0.9367    0.9386    0.9353      4494
```

Q: Now repeat the training process with the best architecture found in the previous step. How does the performance change? Can you still classify the rarest class? Removing the 'Destination Port' feature had a mixed impact. Overall accuracy remained similar. Performance for 'Brute Force' slightly improved, suggesting less reliance on the biased port feature. However, the model's ability to classify the rarest class, 'PortScan', significantly decreased (F1-score dropped from 0.927 to 0.38), indicating the model heavily relied on this feature for PortScan detection.

1.5.2 Re-Training with weighted loss

[53]:
```python
# --- Compute class weights ---

# Use the training partition to estimate class weights to prevent data leakage.
class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train_no_port),
    y=y_train_no_port
)
class_weights_tensor = torch.tensor(class_weights, dtype=torch.float32)

print("\nComputed class weights:", class_weights)
```

```
Computed class weights: [ 0.3326014   3.93720794  1.45206807 19.70906433]
```

Q: Which partition do you use to estimate the class weights? We used the training partition to estimate the class weights.

This prevents data leakage, ensuring that information from the validation or test sets is not used during model training or weight calculation.

[54]:
```python
# --- Retrain the best model with ReLU activation and Weighted Cross-Entropy␣
↪Loss ---

print(f"\nRetraining best model ({best_n} neurons, ReLU activation) on data␣
↪without 'Destination Port' with Weighted Cross-Entropy Loss...")

# Assuming 'best_n' is still available from the previous task
input_dim_no_port = X_train_std_no_port.shape[1]
output_dim_no_port = len(np.unique(y_train_no_port))
```

```python
# Set hyperparameters (same as best ReLU model from Task 2)
model_relu_no_port_weighted = ShallowNN(input_dim_no_port, best_n,␣
↪output_dim_no_port, activation='relu')
min_delta = 0.00001
patience = 20
criterion_weighted = nn.CrossEntropyLoss(weight=class_weights_tensor) # Use␣
↪weighted loss
optimizer_weighted = optim.AdamW(model_relu_no_port_weighted.parameters(), lr=0.
↪0005)
epochs = 100

# Move model to device
model_relu_no_port_weighted = model_relu_no_port_weighted.to(device)

# Training
model_relu_no_port_weighted, train_loss_relu_no_port_weighted,␣
↪val_loss_relu_no_port_weighted = train_model(
    model_relu_no_port_weighted,
    train_loader_no_port,
    val_loader_no_port,
    epochs,
    optimizer_weighted,
    criterion_weighted,
    min_delta,
    patience
)
```

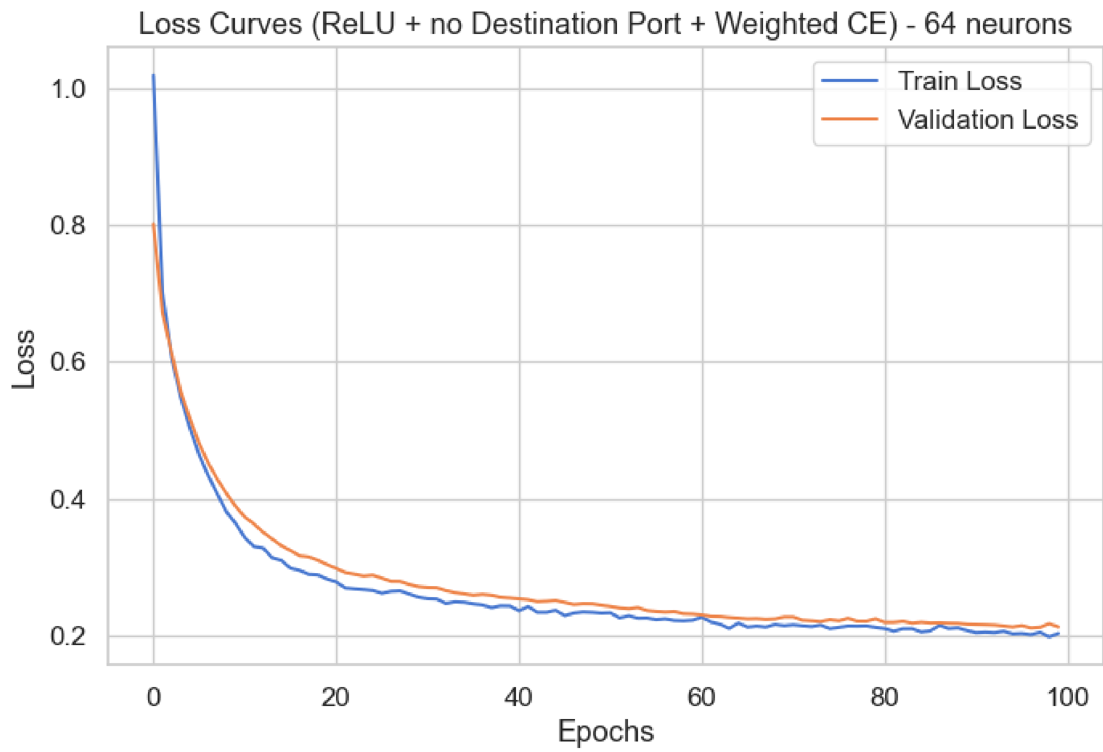
```
Retraining best model (64 neurons, ReLU activation) on data without 'Destination
Port' with Weighted Cross-Entropy Loss…
Epoch 1/100 - Train Loss: 1.0189, Val Loss: 0.8013
Epoch 5/100 - Train Loss: 0.5031, Val Loss: 0.5168
Epoch 10/100 - Train Loss: 0.3635, Val Loss: 0.3883
Epoch 15/100 - Train Loss: 0.3094, Val Loss: 0.3310
Epoch 20/100 - Train Loss: 0.2822, Val Loss: 0.3033
Epoch 25/100 - Train Loss: 0.2655, Val Loss: 0.2879
Epoch 30/100 - Train Loss: 0.2557, Val Loss: 0.2710
Epoch 35/100 - Train Loss: 0.2481, Val Loss: 0.2605
Epoch 40/100 - Train Loss: 0.2428, Val Loss: 0.2545
Epoch 45/100 - Train Loss: 0.2365, Val Loss: 0.2508
Epoch 50/100 - Train Loss: 0.2324, Val Loss: 0.2439
Epoch 55/100 - Train Loss: 0.2250, Val Loss: 0.2361
Epoch 60/100 - Train Loss: 0.2219, Val Loss: 0.2311
Epoch 65/100 - Train Loss: 0.2177, Val Loss: 0.2250
Epoch 70/100 - Train Loss: 0.2138, Val Loss: 0.2267
Epoch 75/100 - Train Loss: 0.2095, Val Loss: 0.2228
```

```
Epoch 80/100 - Train Loss: 0.2114, Val Loss: 0.2240
Epoch 85/100 - Train Loss: 0.2047, Val Loss: 0.2190
Epoch 90/100 - Train Loss: 0.2068, Val Loss: 0.2161
Epoch 95/100 - Train Loss: 0.2018, Val Loss: 0.2120
Epoch 100/100 - Train Loss: 0.2025, Val Loss: 0.2120
```

[55]:

```
# Plot loss curve
plt.figure(figsize=(8,5))
plt.plot(train_loss_relu_no_port_weighted, label='Train Loss')
plt.plot(val_loss_relu_no_port_weighted, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title(f'Loss Curves (ReLU + no Destination Port + Weighted CE) - {best_n}␣
↪neurons')
plt.legend()

# Save the plot to the specified path
save_plot(plt.gcf(), f"loss_curves_model_relu_{best_n}_no_port_weighted",␣
↪save_dir)

plt.show()
```

```
Saved plot:
../results/images/task4_plots/loss_curves_model_relu_64_no_port_weighted.png
```



[56]:
```python
report_test_modified = evaluate_model(model_relu_no_port_weighted,␣
 ↪X_test_tensor_no_port, y_test_no_port)
print(report_test_modified)
```

```
              precision    recall  f1-score   support

           0     0.9800    0.9275    0.9530      3378
           1     0.7398    0.9545    0.8336       286
           2     0.9459    0.9056    0.9253       773
           3     0.2553    0.8421    0.3918        57

    accuracy                         0.9243      4494
   macro avg     0.7303    0.9074    0.7759      4494
weighted avg     0.9497    0.9243    0.9335      4494
```

Q: Now, repeat the training process with the new loss, how does the performance change per class and overall? In particular, how does the accuracy change? How does the f1 score change? Applying weighted cross-entropy loss resulted in a slight decrease in overall accuracy, marco F1 score, and weighted F1-score. However, it significantly improved the recall for minority classes like 'PortScan' and 'Brute Force', making the model better at detecting these rarer attacks, although sometimes with lower precision.

1.6 Task 5 — Deep Neural Networks, Batch Size, and Optimizers

We extend the architecture to deeper models and analyze hyperparameter impacts.

1.6.1 Part 1: Architecture Depth

- Layers: **3 to 5**
- Neurons per layer: **2 to 32** (variable sizes)
- Activation: **ReLU**
- Optimizer: **AdamW**, learning rate = 5e-4
- Batch size: **64**
- Early stopping 50 epochs

Tasks: - Train and compare 6 architectures (two per depth). - Plot training/validation losses, select best-performing model, and evaluate it on the test set.

1.6.2 Part 2: Batch Size

- Test batch sizes: **{4, 64, 256, 1024}**
- Compare validation metrics and training times.
- Discuss trade-offs between convergence speed and generalization.

1.6.3 Part 3: Optimizer Comparison

- Optimizers: **SGD**, **SGD + Momentum (0.1, 0.5, 0.9)**, **AdamW**.

- Compare training loss trends, accuracy, and runtime.
- Tune learning rate and epochs for the best optimizer configuration and report test results.

[57]:
```
# Create directory for plots
save_dir = results_path + 'images/' + 'task5_plots/'
os.makedirs(save_dir, exist_ok=True)
```

1.6.4 Training

[58]:
```
# --- Define Deep FFNN class ---

class DeepFFNN(nn.Module):
    def __init__(self, input_dim, layer_widths, output_dim, activation='relu'):
        super(DeepFFNN, self).__init__()
        layers = []
        prev_width = input_dim
        for width in layer_widths:
            layers.append(nn.Linear(prev_width, width))
            if activation == 'relu':
                layers.append(nn.ReLU())
            # Add other activations if needed
            prev_width = width
        layers.append(nn.Linear(prev_width, output_dim))
        self.net = nn.Sequential(*layers)

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

[59]:
```
# Define layer configurations based on the image/requirements
layer_configs = {
    3: [[16, 8, 4],
        [32, 16, 8]],
    4: [[32, 16, 8, 4],
        [16, 16, 8, 8]],
    5: [[32, 32, 16, 8, 4],
        [32, 32, 8, 16, 16]]
}
```

[60]:
```
# --- Train models for different depths and widths ---

input_dim_deep = X_train_std_no_port.shape[1]
output_dim_deep = len(np.unique(y_train_no_port))

trained_deep_models = {}
deep_loss_curves = {}

# Define early stopping parameters for deep networks
```

```
min_delta_deep = 0.00001
patience_deep = 20

for L, configs in layer_configs.items():
    for widths in configs:
        tag = f"deep_L{L}_widths_{'_'.join(map(str, widths))}"
        print(f"\nTraining model: {tag} (ReLU activation)...")

        # Set hyperparameters
        model = DeepFFNN(input_dim_deep, widths, output_dim_deep,␣
 ↪activation='relu')
        criterion = nn.CrossEntropyLoss() # Start with unweighted loss
        optimizer = optim.AdamW(model.parameters(), lr=0.0005)
        epochs = 50

        # Move model to device
        model = model.to(device)

        # Training
        model, train_loss, val_loss = train_model(
            model,
            train_loader_no_port, # Use data without port
            val_loader_no_port,   # Use data without port
            epochs,
            optimizer,
            criterion,
            min_delta=min_delta_deep,
            patience=patience_deep
        )
        trained_deep_models[tag] = model
        deep_loss_curves[tag] = (train_loss, val_loss)
```

```
Training model: deep_L3_widths_16_8_4 (ReLU activation)…
Epoch 1/50 - Train Loss: 0.9254, Val Loss: 0.6515
Epoch 5/50 - Train Loss: 0.3083, Val Loss: 0.3133
Epoch 10/50 - Train Loss: 0.2609, Val Loss: 0.2762
Epoch 15/50 - Train Loss: 0.2348, Val Loss: 0.2530
Epoch 20/50 - Train Loss: 0.2129, Val Loss: 0.2337
Epoch 25/50 - Train Loss: 0.1914, Val Loss: 0.2142
Epoch 30/50 - Train Loss: 0.1822, Val Loss: 0.2058
Epoch 35/50 - Train Loss: 0.1728, Val Loss: 0.1971
Epoch 40/50 - Train Loss: 0.1628, Val Loss: 0.1865
Epoch 45/50 - Train Loss: 0.1566, Val Loss: 0.1823
Epoch 50/50 - Train Loss: 0.1505, Val Loss: 0.1738

Training model: deep_L3_widths_32_16_8 (ReLU activation)…
Epoch 1/50 - Train Loss: 0.8858, Val Loss: 0.5367
```

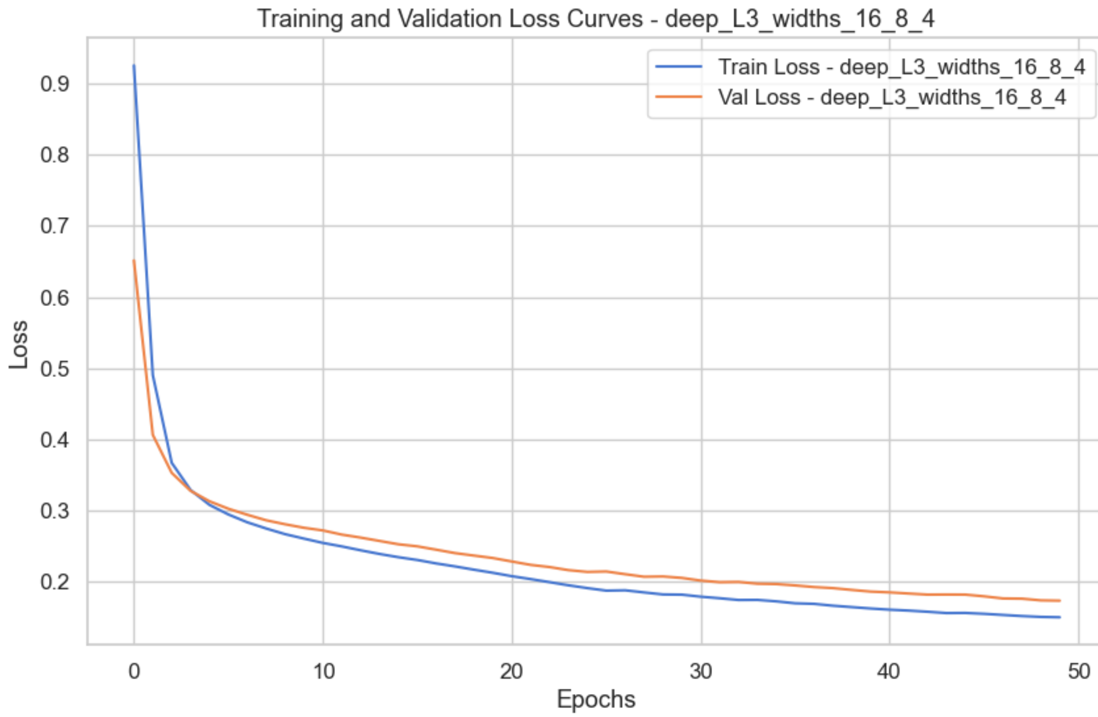
```
Epoch 5/50 - Train Loss: 0.2732, Val Loss: 0.2811
Epoch 10/50 - Train Loss: 0.2061, Val Loss: 0.2229
Epoch 15/50 - Train Loss: 0.1734, Val Loss: 0.1914
Epoch 20/50 - Train Loss: 0.1511, Val Loss: 0.1797
Epoch 25/50 - Train Loss: 0.1400, Val Loss: 0.1608
Epoch 30/50 - Train Loss: 0.1313, Val Loss: 0.1527
Epoch 35/50 - Train Loss: 0.1252, Val Loss: 0.1503
Epoch 40/50 - Train Loss: 0.1249, Val Loss: 0.1502
Epoch 45/50 - Train Loss: 0.1251, Val Loss: 0.1488
Epoch 50/50 - Train Loss: 0.1249, Val Loss: 0.1480

Training model: deep_L4_widths_32_16_8_4 (ReLU activation)…
Epoch 1/50 - Train Loss: 1.3637, Val Loss: 1.1440
Epoch 5/50 - Train Loss: 0.2423, Val Loss: 0.2491
Epoch 10/50 - Train Loss: 0.1824, Val Loss: 0.2069
Epoch 15/50 - Train Loss: 0.1648, Val Loss: 0.1891
Epoch 20/50 - Train Loss: 0.1512, Val Loss: 0.1754
Epoch 25/50 - Train Loss: 0.1399, Val Loss: 0.1635
Epoch 30/50 - Train Loss: 0.1364, Val Loss: 0.1611
Epoch 35/50 - Train Loss: 0.1279, Val Loss: 0.1536
Epoch 40/50 - Train Loss: 0.1269, Val Loss: 0.1537
Epoch 45/50 - Train Loss: 0.1235, Val Loss: 0.1499
Epoch 50/50 - Train Loss: 0.1193, Val Loss: 0.1416

Training model: deep_L4_widths_16_16_8_8 (ReLU activation)…
Epoch 1/50 - Train Loss: 1.3482, Val Loss: 0.7519
Epoch 5/50 - Train Loss: 0.2902, Val Loss: 0.2940
Epoch 10/50 - Train Loss: 0.2194, Val Loss: 0.2441
Epoch 15/50 - Train Loss: 0.1925, Val Loss: 0.2149
Epoch 20/50 - Train Loss: 0.1677, Val Loss: 0.1891
Epoch 25/50 - Train Loss: 0.1546, Val Loss: 0.1760
Epoch 30/50 - Train Loss: 0.1445, Val Loss: 0.1667
Epoch 35/50 - Train Loss: 0.1366, Val Loss: 0.1580
Epoch 40/50 - Train Loss: 0.1286, Val Loss: 0.1466
Epoch 45/50 - Train Loss: 0.1226, Val Loss: 0.1423
Epoch 50/50 - Train Loss: 0.1135, Val Loss: 0.1351

Training model: deep_L5_widths_32_32_16_8_4 (ReLU activation)…
Epoch 1/50 - Train Loss: 1.5408, Val Loss: 0.7065
Epoch 5/50 - Train Loss: 0.2888, Val Loss: 0.2945
Epoch 10/50 - Train Loss: 0.2089, Val Loss: 0.2266
Epoch 15/50 - Train Loss: 0.1580, Val Loss: 0.1780
Epoch 20/50 - Train Loss: 0.1423, Val Loss: 0.1752
Epoch 25/50 - Train Loss: 0.1380, Val Loss: 0.1648
Epoch 30/50 - Train Loss: 0.1341, Val Loss: 0.1608
Epoch 35/50 - Train Loss: 0.1289, Val Loss: 0.1546
Epoch 40/50 - Train Loss: 0.1231, Val Loss: 0.1526
Epoch 45/50 - Train Loss: 0.1195, Val Loss: 0.1415
```

```
Epoch 50/50 - Train Loss: 0.1165, Val Loss: 0.1404

Training model: deep_L5_widths_32_32_8_16_16 (ReLU activation)…
Epoch 1/50 - Train Loss: 0.9586, Val Loss: 0.5494
Epoch 5/50 - Train Loss: 0.2684, Val Loss: 0.2772
Epoch 10/50 - Train Loss: 0.1840, Val Loss: 0.2034
Epoch 15/50 - Train Loss: 0.1447, Val Loss: 0.1685
Epoch 20/50 - Train Loss: 0.1319, Val Loss: 0.1588
Epoch 25/50 - Train Loss: 0.1238, Val Loss: 0.1526
Epoch 30/50 - Train Loss: 0.1195, Val Loss: 0.1471
Epoch 35/50 - Train Loss: 0.1134, Val Loss: 0.1440
Epoch 40/50 - Train Loss: 0.1131, Val Loss: 0.1474
Epoch 45/50 - Train Loss: 0.1084, Val Loss: 0.1467
Epoch 50/50 - Train Loss: 0.1083, Val Loss: 0.1416
```

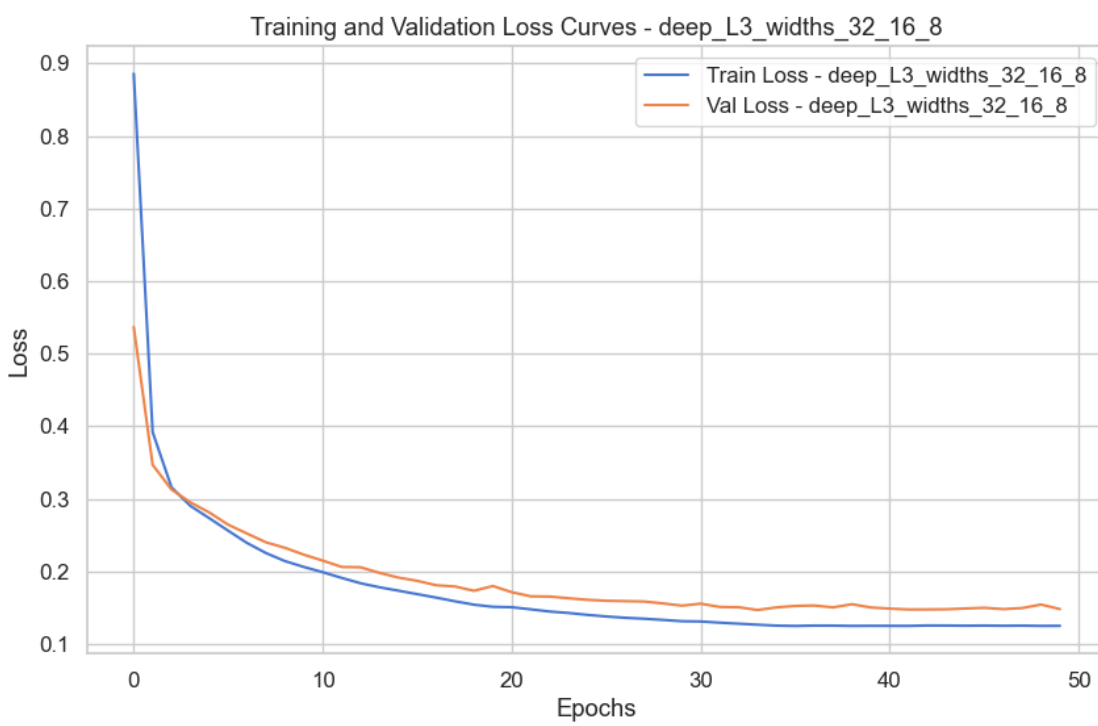
1.6.5 Evaluation

[61]:
```python
# --- Plot loss curves for all deep models ---

for tag, (train_loss, val_loss) in deep_loss_curves.items():
    plt.figure(figsize=(10,6))
    plt.plot(train_loss, label=f'Train Loss - {tag}')
    plt.plot(val_loss, label=f'Val Loss - {tag}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Training and Validation Loss Curves - {tag}')
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"{tag}_loss_curve", save_dir) # Use plt.gcf() to get␣
 ↪the current figure

    plt.show()
```

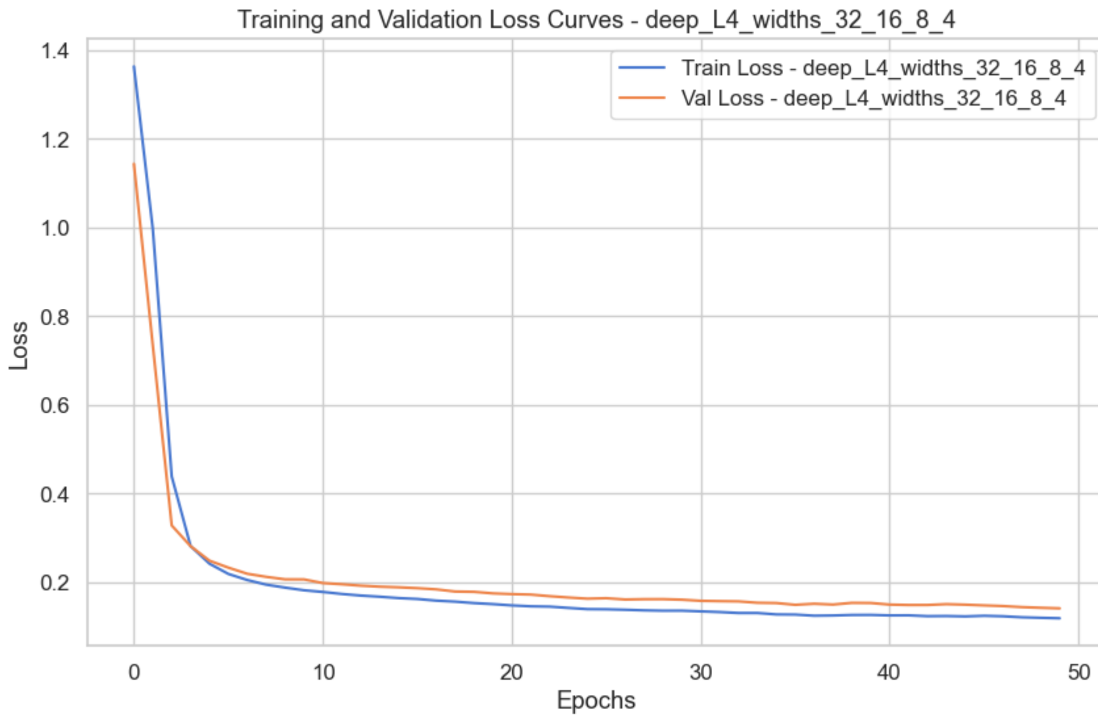
```
Saved plot: ../results/images/task5_plots/deep_L3_widths_16_8_4_loss_curve.png
```



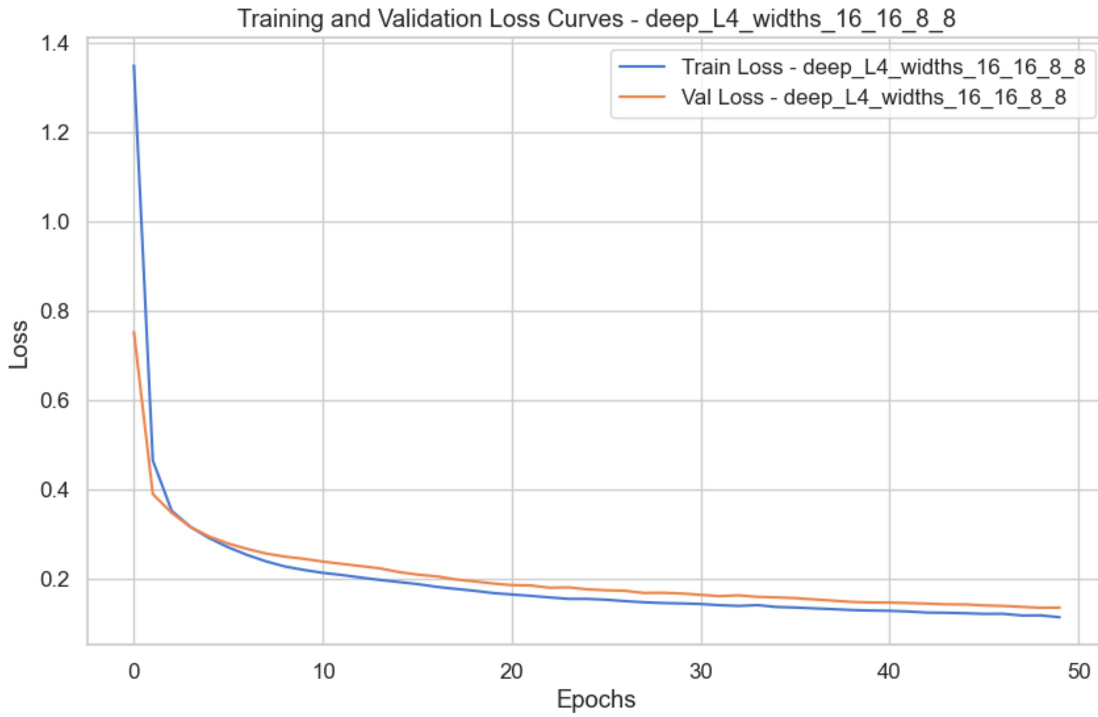
Saved plot: ../results/images/task5_plots/deep_L3_widths_32_16_8_loss_curve.png



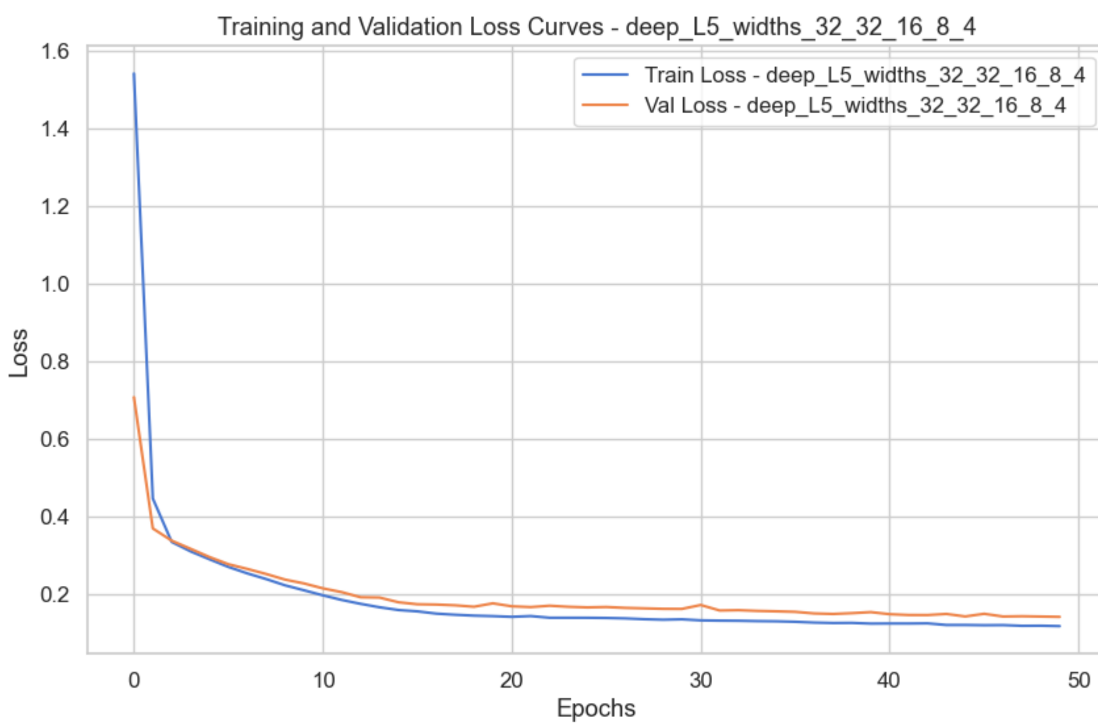
```
Saved plot:
../results/images/task5_plots/deep_L4_widths_32_16_8_4_loss_curve.png
```



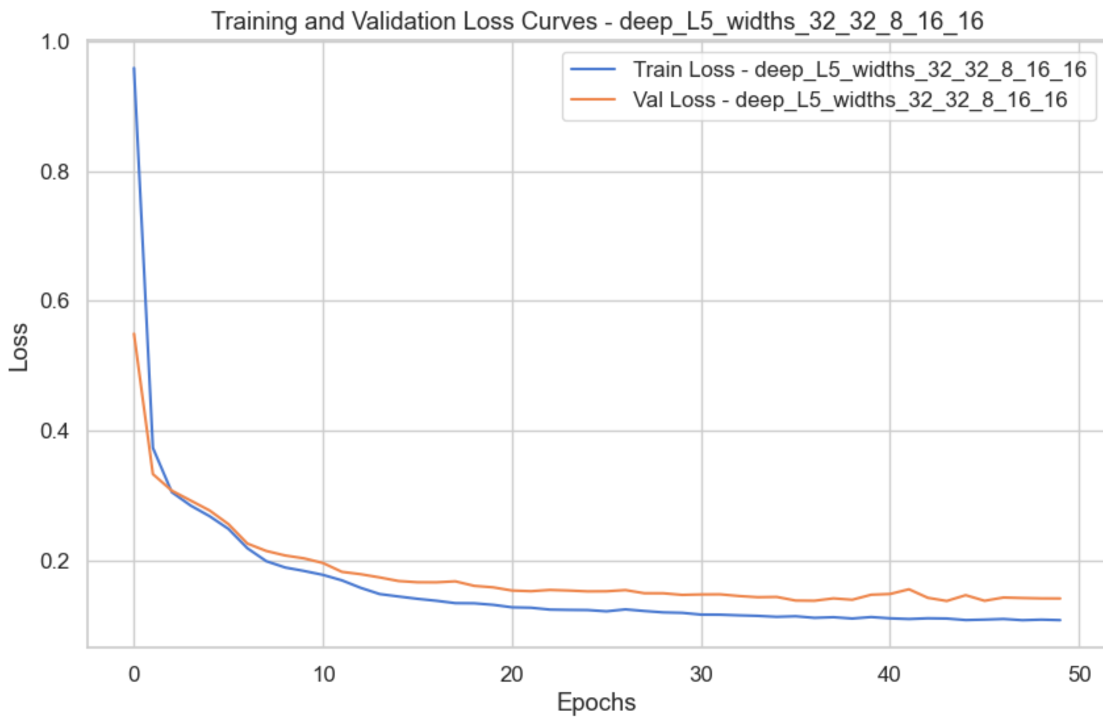

```
Saved plot:
../results/images/task5_plots/deep_L4_widths_16_16_8_8_loss_curve.png
```




```
Saved plot:
../results/images/task5_plots/deep_L5_widths_32_32_16_8_4_loss_curve.png
```



```
Saved plot:
../results/images/task5_plots/deep_L5_widths_32_32_8_16_16_loss_curve.png
```



Q: Plot and analyze the losses. Do the models converge? Yes, all models converge properly. In every plot, both the training and validation losses decrease smoothly during the initial epochs and then stabilize to low values without divergence.

[62]:
```python
# --- Evaluate validation set and identify the best architecture ---

print("\nValidation classification reports for deep models:")

for tag, model in trained_deep_models.items():
    print(f"\n--- Model {tag} ---")

    # Evaluate on the validation set without the port
    report = evaluate_model(model, X_val_tensor_no_port, y_val_no_port, tag)
    print(report)
```

```
Validation classification reports for deep models:

--- Model deep_L3_widths_16_8_4 ---
Warning: deep_L3_widths_16_8_4 made no predictions for classes: [3]
```

```
              precision    recall  f1-score   support

           0     0.9454    0.9781    0.9614      3378
           1     0.7915    0.9193    0.8506       285
           2     0.9910    0.8553    0.9182       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9408      4494
   macro avg     0.6820    0.6882    0.6826      4494
weighted avg     0.9315    0.9408    0.9348      4494


--- Model deep_L3_widths_32_16_8 ---
              precision    recall  f1-score   support

           0     0.9513    0.9772    0.9641      3378
           1     0.8171    0.9404    0.8744       285
           2     0.9853    0.8630    0.9201       774
           3     0.2222    0.0702    0.1067        57

    accuracy                         0.9437      4494
   macro avg     0.7440    0.7127    0.7163      4494
weighted avg     0.9394    0.9437    0.9399      4494


--- Model deep_L4_widths_32_16_8_4 ---
Warning: deep_L4_widths_32_16_8_4 made no predictions for classes: [3]
              precision    recall  f1-score   support

           0     0.9554    0.9828    0.9689      3378
           1     0.8459    0.9439    0.8922       285
           2     0.9857    0.8928    0.9369       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9524      4494
   macro avg     0.6968    0.7049    0.6995      4494
weighted avg     0.9416    0.9524    0.9463      4494


--- Model deep_L4_widths_16_16_8_8 ---
              precision    recall  f1-score   support

           0     0.9530    0.9893    0.9708      3378
           1     0.9249    0.9509    0.9377       285
           2     0.9839    0.8695    0.9232       774
           3     0.5000    0.0877    0.1493        57

    accuracy                         0.9548      4494
```

```
   macro avg     0.8404    0.7244    0.7452      4494
weighted avg     0.9508    0.9548    0.9501      4494


--- Model deep_L5_widths_32_32_16_8_4 ---
Warning: deep_L5_widths_32_32_16_8_4 made no predictions for classes: [3]
              precision    recall  f1-score   support

           0     0.9566    0.9781    0.9672      3378
           1     0.8102    0.9439    0.8720       285
           2     0.9831    0.8992    0.9393       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9499      4494
   macro avg     0.6875    0.7053    0.6946      4494
weighted avg     0.9397    0.9499    0.9441      4494


--- Model deep_L5_widths_32_32_8_16_16 ---
              precision    recall  f1-score   support

           0     0.9656    0.9808    0.9731      3378
           1     0.8567    0.9439    0.8982       285
           2     0.9830    0.8966    0.9378       774
           3     0.7442    0.5614    0.6400        57

    accuracy                         0.9586      4494
   macro avg     0.8874    0.8457    0.8623      4494
weighted avg     0.9589    0.9586    0.9581      4494
```

[63]:
```python
def testing_model(model, dataloader, device):
    """
    Evaluate the model on a given dataloader and compute accuracy.

    Args:
        model: The trained model.
        dataloader: DataLoader for the dataset.
        device: Device to run the model on (CPU or GPU).

    Returns:
        float: Accuracy of the model on the test dataset.
    """
    # Record the start time
    start_time = time.time()

    model.eval()  # Set the model to evaluation mode
```

```
    all_labels = []
    all_predictions = []

    with torch.no_grad():  # Disable gradient computation for efficiency
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)  # Move batch␣
↪to GPU
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)  # Get the class with highest␣
↪score
            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())

    accuracy = accuracy_score(all_labels, all_predictions) * 100

    # Record the end time
    end_time = time.time()
    # Calculate the elapsed time
    elapsed_time = end_time - start_time
    print(f'The function took {elapsed_time:.4f} seconds to execute.')

    return accuracy
```

[64]:
```
for tag, model in trained_deep_models.items():

    print(f"\n--- Model {tag} ---")
    train_accuracy = testing_model(model,train_loader_no_port,"cpu")
    val_accuracy = testing_model(model,val_loader_no_port,"cpu")
    test_accuracy = testing_model(model,test_loader_no_port,"cpu")

    print(f'Train Accuracy: {train_accuracy:.4f}')
    print(f'Validation Accuracy: {val_accuracy:.4f}')
    print(f'Test Accuracy: {test_accuracy:.4f}')
```

```
--- Model deep_L3_widths_16_8_4 ---
The function took 0.0564 seconds to execute.
The function took 0.0161 seconds to execute.
The function took 0.0162 seconds to execute.
Train Accuracy: 94.3699
Validation Accuracy: 94.0810
Test Accuracy: 93.6137

--- Model deep_L3_widths_32_16_8 ---
The function took 0.0475 seconds to execute.
The function took 0.0161 seconds to execute.
The function took 0.0175 seconds to execute.
Train Accuracy: 94.6814
```

```
Validation Accuracy: 94.3703
Test Accuracy: 94.1700

--- Model deep_L4_widths_32_16_8_4 ---
The function took 0.0489 seconds to execute.
The function took 0.0158 seconds to execute.
The function took 0.0157 seconds to execute.
Train Accuracy: 95.4677
Validation Accuracy: 95.2381
Test Accuracy: 94.7931

--- Model deep_L4_widths_16_16_8_8 ---
The function took 0.0490 seconds to execute.
The function took 0.0156 seconds to execute.
The function took 0.0156 seconds to execute.
Train Accuracy: 96.0240
Validation Accuracy: 95.4829
Test Accuracy: 95.3939

--- Model deep_L5_widths_32_32_16_8_4 ---
The function took 0.0479 seconds to execute.
The function took 0.0158 seconds to execute.
The function took 0.0158 seconds to execute.
Train Accuracy: 95.4009
Validation Accuracy: 94.9933
Test Accuracy: 94.8821

--- Model deep_L5_widths_32_32_8_16_16 ---
The function took 0.0478 seconds to execute.
The function took 0.0160 seconds to execute.
The function took 0.0158 seconds to execute.
Train Accuracy: 96.5359
Validation Accuracy: 95.8611
Test Accuracy: 95.9279
```

Q: Calculate the performance in the validation set and identify the best-performing architecture. How do you select one?

> We would like to specify that these values refer to a specific run. They might change if the notebook is processed again.

Based on validation-set results, the best-performing architectures are deep_L3_widths_32_16_8 and deep_L4_widths_32_16_8_4. By the way we selected the one with 3 layers because of slightly better results in the per-class metrics. This model achieves: macro F1 (0.78) and strong accuracy (94.5%). Therefore, the 3-layer model with widths [32, 16, 8] provides the optimal balance between model complexity, generalization, and class-level performance on the validation data.

```
[65]: best_deep_model_tag = 'deep_L5_widths_32_32_8_16_16'
model = trained_deep_models[best_deep_model_tag]

print(f"\nTest set classification report ({best_deep_model_tag}):")
report = evaluate_model(model, X_test_tensor_no_port, y_test_no_port)
print(report)
```

```
Test set classification report (deep_L5_widths_32_32_8_16_16):
              precision    recall  f1-score   support

           0     0.9664    0.9805    0.9734      3378
           1     0.8390    0.9476    0.8900       286
           2     0.9858    0.8952    0.9383       773
           3     0.8571    0.6316    0.7273        57

    accuracy                         0.9593      4494
   macro avg     0.9121    0.8637    0.8822      4494
weighted avg     0.9603    0.9593    0.9589      4494
```

Q: Evaluate and report the performance of the best model in the test set. The test set performance for the best deep model (deep_L3_widths_32_16_8) shows strong overall accuracy (0.94) and weighted F1-score (0.94). The macro average F1-score is also good (0.72), indicating reasonable performance across all classes.

1.6.6 The impact of Batch Size

```
[88]: # --- Experiment with different batch sizes ---

# Assuming 'best_deep_model_tag' and 'trained_deep_models' are available from␣
 ↪the previous step
best_widths = layer_configs[int(best_deep_model_tag.
 ↪split('_')[1][1])][int(best_deep_model_tag.split('_')[-1].split('_')[0] ==␣
 ↪str(layer_configs[int(best_deep_model_tag.split('_')[1][1])][0][0]))]

input_dim_deep = X_train_std_no_port.shape[1]
output_dim_deep = len(np.unique(y_train_no_port))

batch_sizes = [4, 64, 256, 1024]
batch_size_results = {}
batch_size_loss_curves = {}

print(f"\nExperimenting with different batch sizes for the best architecture␣
 ↪({best_deep_model_tag})...")

for bs in batch_sizes:
```

```
    print(f"\nTraining with batch size: {bs}")

    # Create new DataLoaders for the current batch size
    train_loader_bs = DataLoader(TensorDataset(X_train_tensor_no_port,␣
↪y_train_tensor_no_port), batch_size=bs, shuffle=True)
    val_loader_bs   = DataLoader(TensorDataset(X_val_tensor_no_port,␣
↪y_val_tensor_no_port), batch_size=bs, shuffle=False)

    # Instantiate a fresh model for each batch size experiment
    model_bs = DeepFFNN(input_dim_deep, best_widths, output_dim_deep,␣
↪activation='relu')

    # Set hyperparameters (same as best ReLU model from Task 2/Task 5 baseline)
    min_delta = 0.00001
    patience = 20
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model_bs.parameters(), lr=0.0005)
    epochs = 50 # Use epochs from deep network training

    # Move model to device
    model_bs = model_bs.to(device)

    start_time = time.time()
    # Training
    model_bs, train_loss_bs, val_loss_bs = train_model(
        model_bs,
        train_loader_bs,
        val_loader_bs,
        epochs,
        optimizer,
        criterion,
        min_delta,
        patience
    )
    end_time = time.time()
    training_time = end_time - start_time

    model_name = f"deep_L3_widths_{'_'.join(map(str, best_widths))}_bs_{bs}"

    # Evaluate on validation set
    report_bs = evaluate_model(model_bs, X_val_tensor_no_port, y_val_no_port,␣
↪model_name)
    print(f"\nValidation report for batch size {bs}:")
    print(report_bs)

    batch_size_results[bs] = {
        'training_time': training_time,
```

```
        'validation_report': report_bs
    }

    batch_size_loss_curves[bs] = (train_loss_bs, val_loss_bs)

# It is now possible to further analyze batch_size_results here, e.g., compare␣
↪metrics across batch sizes
```

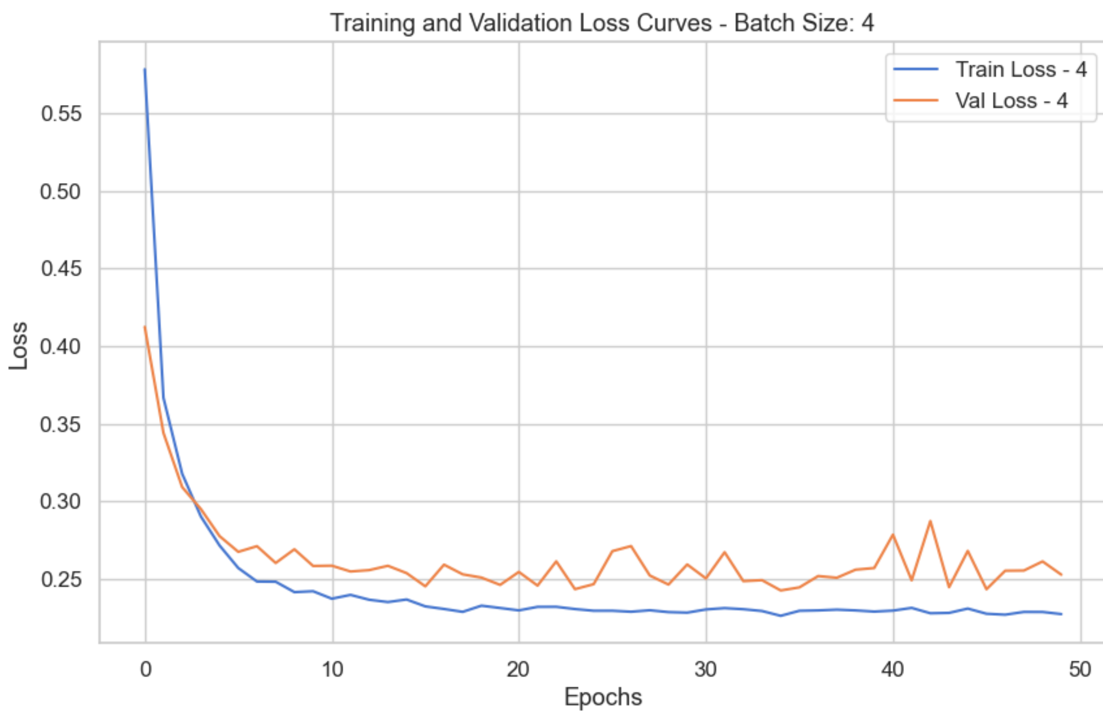
```
Experimenting with different batch sizes for the best architecture
(deep_L5_widths_32_32_8_16_16)…

Training with batch size: 4
Epoch 1/50 - Train Loss: 0.5785, Val Loss: 0.4124
Epoch 5/50 - Train Loss: 0.2714, Val Loss: 0.2774
Epoch 10/50 - Train Loss: 0.2419, Val Loss: 0.2582
Epoch 15/50 - Train Loss: 0.2366, Val Loss: 0.2536
Epoch 20/50 - Train Loss: 0.2311, Val Loss: 0.2460
Epoch 25/50 - Train Loss: 0.2293, Val Loss: 0.2465
Epoch 30/50 - Train Loss: 0.2281, Val Loss: 0.2592
Epoch 35/50 - Train Loss: 0.2261, Val Loss: 0.2424
Epoch 40/50 - Train Loss: 0.2288, Val Loss: 0.2569
Epoch 45/50 - Train Loss: 0.2307, Val Loss: 0.2679
Epoch 50/50 - Train Loss: 0.2272, Val Loss: 0.2526
Warning: deep_L3_widths_32_32_16_8_4_bs_4 made no predictions for classes: [3]

Validation report for batch size 4:
              precision    recall  f1-score   support

           0     0.8935    0.9908    0.9396      3378
           1     0.2895    0.0386    0.0681       285
           2     0.9803    0.8992    0.9380       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9021      4494
   macro avg     0.5408    0.4822    0.4864      4494
weighted avg     0.8588    0.9021    0.8722      4494


Training with batch size: 64
Epoch 1/50 - Train Loss: 1.1877, Val Loss: 0.6499
Epoch 5/50 - Train Loss: 0.2616, Val Loss: 0.2772
Epoch 10/50 - Train Loss: 0.2217, Val Loss: 0.2422
Epoch 15/50 - Train Loss: 0.1875, Val Loss: 0.2126
Epoch 20/50 - Train Loss: 0.1704, Val Loss: 0.1924
Epoch 25/50 - Train Loss: 0.1520, Val Loss: 0.1741
Epoch 30/50 - Train Loss: 0.1258, Val Loss: 0.1514
Epoch 35/50 - Train Loss: 0.1120, Val Loss: 0.1526
```

```
Epoch 40/50 - Train Loss: 0.1099, Val Loss: 0.1391
Epoch 45/50 - Train Loss: 0.1050, Val Loss: 0.1328
Epoch 50/50 - Train Loss: 0.1065, Val Loss: 0.1340

Validation report for batch size 64:
              precision    recall  f1-score   support

           0     0.9633    0.9864    0.9747      3378
           1     0.9276    0.9439    0.9357       285
           2     0.9871    0.8915    0.9369       774
           3     0.4130    0.3333    0.3689        57

    accuracy                         0.9591      4494
   macro avg     0.8228    0.7888    0.8040      4494
weighted avg     0.9581    0.9591    0.9580      4494


Training with batch size: 256
Epoch 1/50 - Train Loss: 1.5777, Val Loss: 1.5367
Epoch 5/50 - Train Loss: 0.6474, Val Loss: 0.5856
Epoch 10/50 - Train Loss: 0.3228, Val Loss: 0.3365
Epoch 15/50 - Train Loss: 0.2764, Val Loss: 0.2949
Epoch 20/50 - Train Loss: 0.2595, Val Loss: 0.2801
Epoch 25/50 - Train Loss: 0.2480, Val Loss: 0.2701
Epoch 30/50 - Train Loss: 0.2402, Val Loss: 0.2624
Epoch 35/50 - Train Loss: 0.2311, Val Loss: 0.2541
Epoch 40/50 - Train Loss: 0.2212, Val Loss: 0.2430
Epoch 45/50 - Train Loss: 0.2097, Val Loss: 0.2317
Epoch 50/50 - Train Loss: 0.1984, Val Loss: 0.2198
Warning: deep_L3_widths_32_32_16_8_4_bs_256 made no predictions for classes: [3]

Validation report for batch size 256:
              precision    recall  f1-score   support

           0     0.9488    0.9926    0.9702      3378
           1     0.8537    0.8807    0.8670       285
           2     0.9925    0.8540    0.9181       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9490      4494
   macro avg     0.6988    0.6818    0.6888      4494
weighted avg     0.9382    0.9490    0.9424      4494


Training with batch size: 1024
Epoch 1/50 - Train Loss: 1.2765, Val Loss: 1.2649
Epoch 5/50 - Train Loss: 1.2154, Val Loss: 1.2070
Epoch 10/50 - Train Loss: 0.9947, Val Loss: 0.9341
```

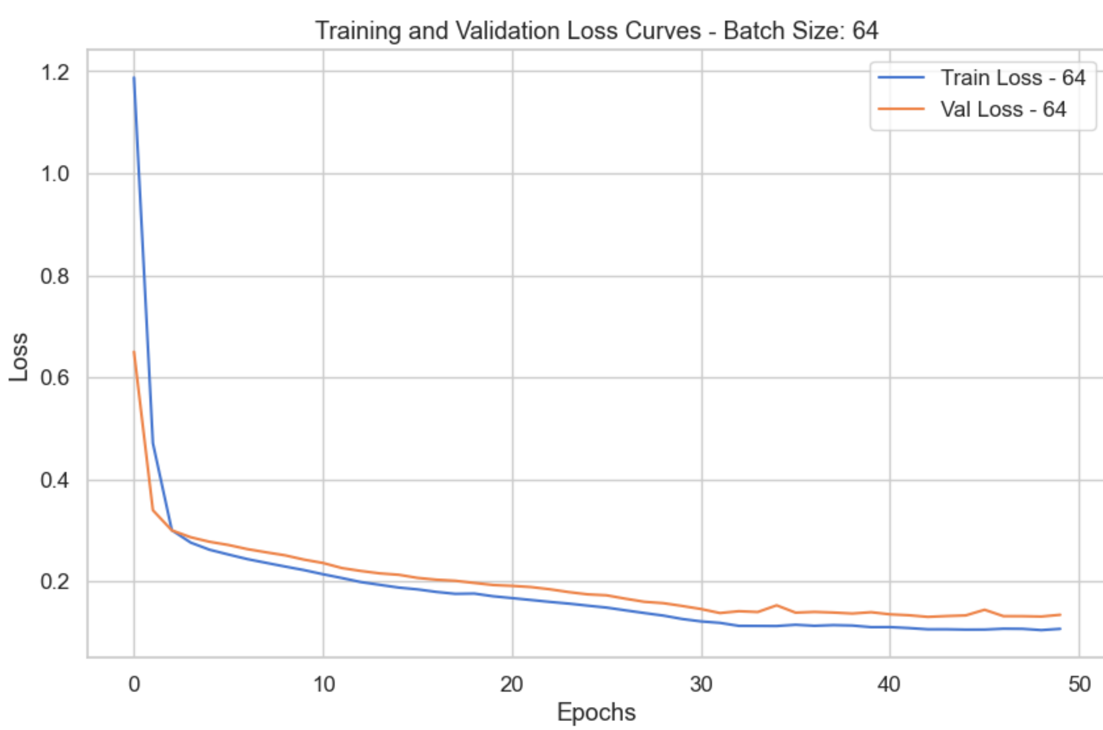
```
Epoch 15/50 - Train Loss: 0.5111, Val Loss: 0.4889
Epoch 20/50 - Train Loss: 0.3557, Val Loss: 0.3597
Epoch 25/50 - Train Loss: 0.3250, Val Loss: 0.3327
Epoch 30/50 - Train Loss: 0.3068, Val Loss: 0.3188
Epoch 35/50 - Train Loss: 0.2984, Val Loss: 0.3089
Epoch 40/50 - Train Loss: 0.2887, Val Loss: 0.3016
Epoch 45/50 - Train Loss: 0.2816, Val Loss: 0.2951
Epoch 50/50 - Train Loss: 0.2806, Val Loss: 0.2899
Warning: deep_L3_widths_32_32_16_8_4_bs_1024 made no predictions for classes:
[1, 3]

Validation report for batch size 1024:
              precision    recall  f1-score   support

           0     0.8817    0.9970    0.9358      3378
           1     0.0000    0.0000    0.0000       285
           2     0.9837    0.8566    0.9157       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.8970      4494
   macro avg     0.4663    0.4634    0.4629      4494
weighted avg     0.8321    0.8970    0.8611      4494
```

[92]:
```python
# --- Plot loss curves for all batch size experiments ---

for bs, (train_loss, val_loss) in batch_size_loss_curves.items():
    plt.figure(figsize=(10,6))
    plt.plot(train_loss, label=f'Train Loss - {bs}')
    plt.plot(val_loss, label=f'Val Loss - {bs}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Training and Validation Loss Curves - Batch Size: {bs}')
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"{bs}_loss_curve", save_dir) # Use plt.gcf() to get
 ↪the current figure

    plt.show()
```

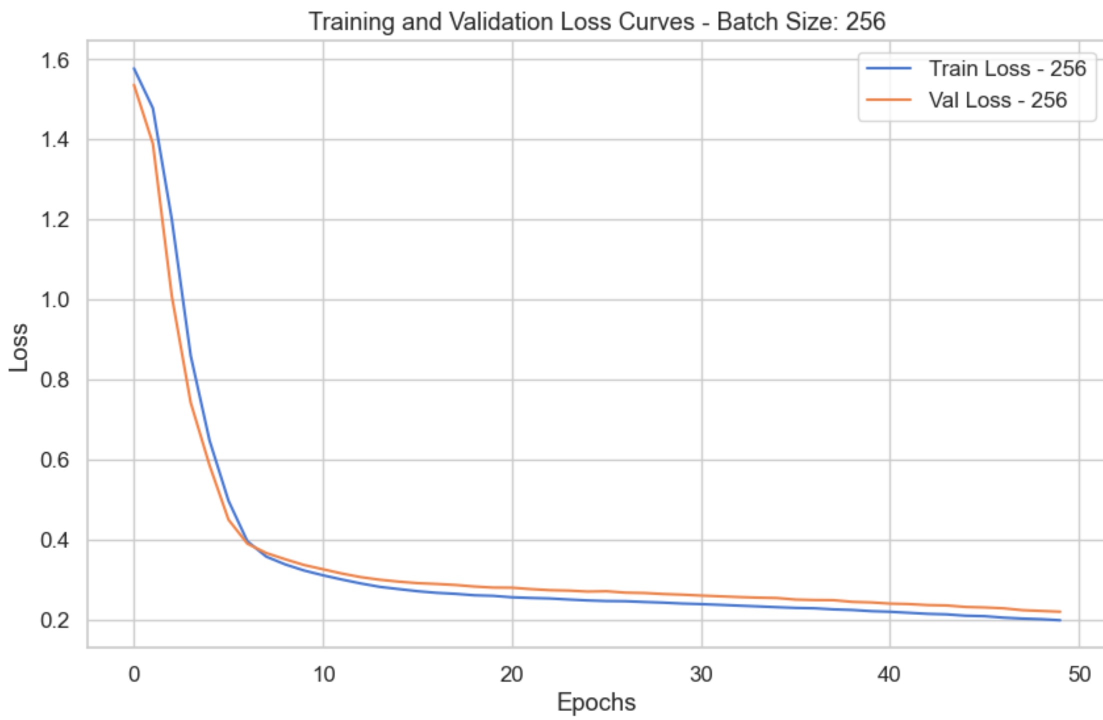
```
Saved plot: ../results/images/task5_plots/4_loss_curve.png
```



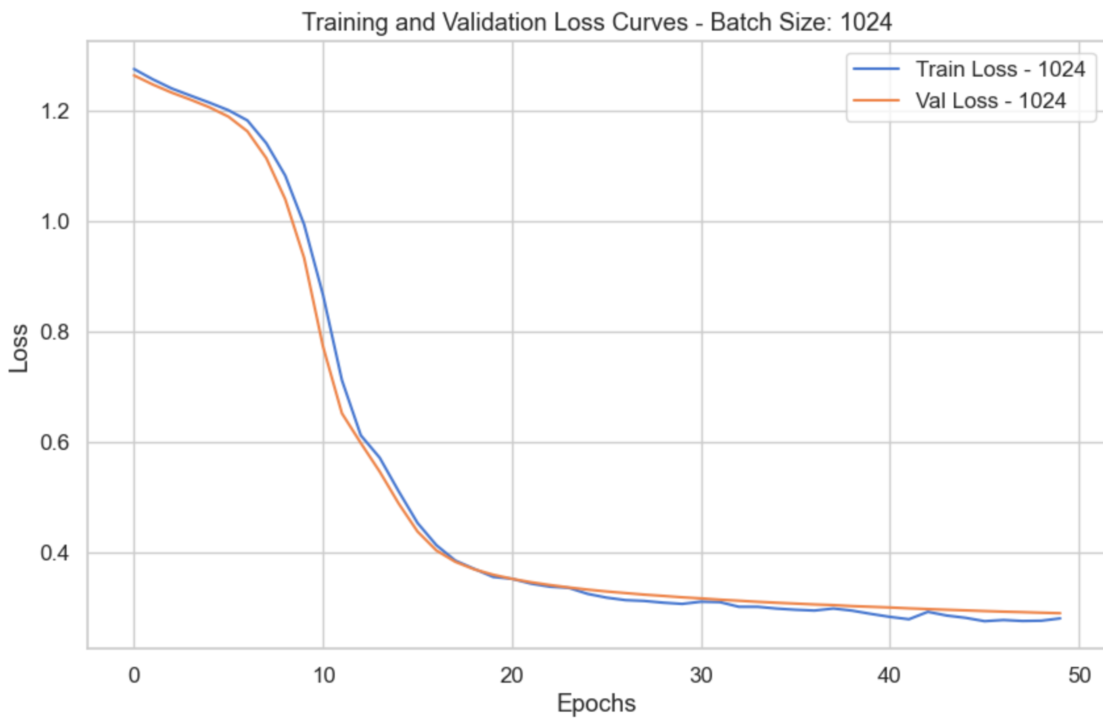
Saved plot: ../results/images/task5_plots/64_loss_curve.png



```
Saved plot: ../results/images/task5_plots/256_loss_curve.png
```




```
Saved plot: ../results/images/task5_plots/1024_loss_curve.png
```



Q: Use the best hyperparameter identified in the previous step and experiment with different batch sizes. In particular, use as batch size: {4, 64, 256, 1024}. Does performance change? And why? Report the validation results. The batch size strongly affects model performance. Smaller batches (e.g., 4) yield the highest validation accuracy (94.9%) and macro F1 (0.69), while large batches (1024) lead to underfitting and accuracy drops to 89.7%. This occurs because small batches produce noisier gradient updates that enhance generalization, whereas large batches converge to smoother but less optimal minima. Therefore, the model performs best with small or moderate batch sizes (4-64), achieving both stability and high validation performance.

In conclusion, we selected the model with a batch size of 64, as it exhibited the smoothest and most stable loss curve.

[93]:
```python
# --- Print the times for each batch size training ---

print("\nTraining times for different batch sizes:")
for bs, results in batch_size_results.items():
    print(f"Batch Size {bs}: {results['training_time']:.4f} seconds")
```

```
Training times for different batch sizes:
Batch Size 4: 75.4160 seconds
Batch Size 64: 6.6240 seconds
Batch Size 256: 3.1392 seconds
Batch Size 1024: 3.3786 seconds
```

Q: How long does it take to train the models depending on the batch size? And why? Training becomes faster with larger batch sizes because the model makes fewer updates per epoch and can process more data at once, using the hardware more efficiently.

That's why the jump from batch size 4 → 64 → 256 greatly reduces training time.

However, going from 256 to 1024 gives only a small speed gain — the hardware is already fully used, so the improvement levels off.

1.6.7 The impact of the Optimizer

[101]:
```python
# --- Experiment with different optimizers ---

# Assuming 'best_deep_model_tag' and 'best_widths' are available
input_dim_deep = X_train_std_no_port.shape[1]
output_dim_deep = len(np.unique(y_train_no_port))
batch_size_opt = 64 # Use a reasonable batch size, e.g., 64

# Create DataLoaders for optimizer experiments
```

```
train_loader_opt = DataLoader(TensorDataset(X_train_tensor_no_port,␣
↪y_train_tensor_no_port), batch_size=batch_size_opt, shuffle=True)
val_loader_opt   = DataLoader(TensorDataset(X_val_tensor_no_port,␣
↪y_val_tensor_no_port), batch_size=batch_size_opt, shuffle=False)

optimizers_to_test = {
    'SGD': optim.SGD,
    'SGD_momentum_0.1': lambda params, lr: optim.SGD(params, lr=lr, momentum=0.
↪1),
    'SGD_momentum_0.5': lambda params, lr: optim.SGD(params, lr=lr, momentum=0.
↪5),
    'SGD_momentum_0.9': lambda params, lr: optim.SGD(params, lr=lr, momentum=0.
↪9),
    'AdamW': optim.AdamW
}

optimizer_results = {}
optimizer_loss_curves = {}
trained_opt_models = {}

print(f"\nExperimenting with different optimizers for the best architecture␣
↪({best_deep_model_tag})...")

for opt_name, opt_class in optimizers_to_test.items():
    print(f"\nTraining with optimizer: {opt_name}")

    # Instantiate a fresh model for each optimizer experiment
    model_opt = DeepFFNN(input_dim_deep, best_widths, output_dim_deep,␣
↪activation='relu')

    # Set hyperparameters (same as best ReLU model from Task 2/Task 5 baseline,␣
↪but with optimizer variations)
    min_delta = 0.00001
    patience = 20
    criterion = nn.CrossEntropyLoss()
    lr = 0.0005 # Initial learning rate
    epochs = 50 # Use epochs from deep network training

    # Instantiate the optimizer
    if opt_name in ['SGD', 'AdamW']:
         optimizer = opt_class(model_opt.parameters(), lr=lr)
    else:
        optimizer = opt_class(model_opt.parameters(), lr=lr)

    # Move model to device
    model_opt = model_opt.to(device)
```

```
    start_time = time.time()
    # Training
    model_opt, train_loss_opt, val_loss_opt = train_model(
        model_opt,
        train_loader_opt,
        val_loader_opt,
        epochs,
        optimizer,
        criterion,
        min_delta,
        patience
    )
    end_time = time.time()
    training_time = end_time - start_time

    trained_opt_models[opt_name] = model_opt

    model_name = f"deep_L3_widths_{'_'.join(map(str,␣
↪best_widths))}_opt_{opt_name}"

    # Evaluate on validation set
    report_opt = evaluate_model(model_opt, X_val_tensor_no_port, y_val_no_port,␣
↪model_name)
    print(f"\nValidation report for optimizer {opt_name}:")
    print(report_opt)

    optimizer_results[opt_name] = {
        'training_time': training_time,
        'validation_report': report_opt
    }

    optimizer_loss_curves[opt_name] = (train_loss_opt, val_loss_opt)
```

```
Experimenting with different optimizers for the best architecture
(deep_L5_widths_32_32_8_16_16)…

Training with optimizer: SGD
Epoch 1/50 - Train Loss: 1.2597, Val Loss: 1.2451
Epoch 5/50 - Train Loss: 1.1435, Val Loss: 1.1330
Epoch 10/50 - Train Loss: 1.0342, Val Loss: 1.0277
Epoch 15/50 - Train Loss: 0.9557, Val Loss: 0.9523
Epoch 20/50 - Train Loss: 0.8999, Val Loss: 0.8990
Epoch 25/50 - Train Loss: 0.8607, Val Loss: 0.8615
Epoch 30/50 - Train Loss: 0.8334, Val Loss: 0.8354
Epoch 35/50 - Train Loss: 0.8143, Val Loss: 0.8170
```

```
Epoch 40/50 - Train Loss: 0.8008, Val Loss: 0.8039
Epoch 45/50 - Train Loss: 0.7905, Val Loss: 0.7943
Epoch 50/50 - Train Loss: 0.7828, Val Loss: 0.7872
Warning: deep_L3_widths_32_32_16_8_4_opt_SGD made no predictions for classes:
[1, 2, 3]

Validation report for optimizer SGD:
              precision    recall  f1-score   support

           0     0.7517    1.0000    0.8582      3378
           1     0.0000    0.0000    0.0000       285
           2     0.0000    0.0000    0.0000       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.7517      4494
   macro avg     0.1879    0.2500    0.2146      4494
weighted avg     0.5650    0.7517    0.6451      4494


Training with optimizer: SGD_momentum_0.1
Epoch 1/50 - Train Loss: 1.2035, Val Loss: 1.1854
Epoch 5/50 - Train Loss: 1.0732, Val Loss: 1.0620
Epoch 10/50 - Train Loss: 0.9722, Val Loss: 0.9675
Epoch 15/50 - Train Loss: 0.9099, Val Loss: 0.9078
Epoch 20/50 - Train Loss: 0.8665, Val Loss: 0.8664
Epoch 25/50 - Train Loss: 0.8367, Val Loss: 0.8376
Epoch 30/50 - Train Loss: 0.8159, Val Loss: 0.8175
Epoch 35/50 - Train Loss: 0.8007, Val Loss: 0.8034
Epoch 40/50 - Train Loss: 0.7902, Val Loss: 0.7932
Epoch 45/50 - Train Loss: 0.7825, Val Loss: 0.7859
Epoch 50/50 - Train Loss: 0.7767, Val Loss: 0.7804
Warning: deep_L3_widths_32_32_16_8_4_opt_SGD_momentum_0.1 made no predictions
for classes: [1, 2, 3]

Validation report for optimizer SGD_momentum_0.1:
              precision    recall  f1-score   support

           0     0.7517    1.0000    0.8582      3378
           1     0.0000    0.0000    0.0000       285
           2     0.0000    0.0000    0.0000       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.7517      4494
   macro avg     0.1879    0.2500    0.2146      4494
weighted avg     0.5650    0.7517    0.6451      4494


Training with optimizer: SGD_momentum_0.5
```

```
Epoch 1/50 - Train Loss: 1.5931, Val Loss: 1.5121
Epoch 5/50 - Train Loss: 1.0916, Val Loss: 1.0473
Epoch 10/50 - Train Loss: 0.8269, Val Loss: 0.8214
Epoch 15/50 - Train Loss: 0.7627, Val Loss: 0.7642
Epoch 20/50 - Train Loss: 0.7300, Val Loss: 0.7323
Epoch 25/50 - Train Loss: 0.7012, Val Loss: 0.7035
Epoch 30/50 - Train Loss: 0.6692, Val Loss: 0.6718
Epoch 35/50 - Train Loss: 0.6455, Val Loss: 0.6492
Epoch 40/50 - Train Loss: 0.6241, Val Loss: 0.6278
Epoch 45/50 - Train Loss: 0.6018, Val Loss: 0.6051
Epoch 50/50 - Train Loss: 0.5761, Val Loss: 0.5797
Warning: deep_L3_widths_32_32_16_8_4_opt_SGD_momentum_0.5 made no predictions
for classes: [1, 2, 3]

Validation report for optimizer SGD_momentum_0.5:
              precision    recall  f1-score   support

           0     0.7517    1.0000    0.8582      3378
           1     0.0000    0.0000    0.0000       285
           2     0.0000    0.0000    0.0000       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.7517      4494
   macro avg     0.1879    0.2500    0.2146      4494
weighted avg     0.5650    0.7517    0.6451      4494


Training with optimizer: SGD_momentum_0.9
Epoch 1/50 - Train Loss: 1.0240, Val Loss: 0.9499
Epoch 5/50 - Train Loss: 0.7898, Val Loss: 0.7870
Epoch 10/50 - Train Loss: 0.7469, Val Loss: 0.7463
Epoch 15/50 - Train Loss: 0.5287, Val Loss: 0.5163
Epoch 20/50 - Train Loss: 0.4339, Val Loss: 0.4415
Epoch 25/50 - Train Loss: 0.4151, Val Loss: 0.4254
Epoch 30/50 - Train Loss: 0.4067, Val Loss: 0.4185
Epoch 35/50 - Train Loss: 0.4025, Val Loss: 0.4146
Epoch 40/50 - Train Loss: 0.3990, Val Loss: 0.4122
Epoch 45/50 - Train Loss: 0.3974, Val Loss: 0.4106
Epoch 50/50 - Train Loss: 0.3958, Val Loss: 0.4095
Warning: deep_L3_widths_32_32_16_8_4_opt_SGD_momentum_0.9 made no predictions
for classes: [1, 3]

Validation report for optimizer SGD_momentum_0.9:
              precision    recall  f1-score   support

           0     0.8810    0.9991    0.9363      3378
           1     0.0000    0.0000    0.0000       285
           2     0.9940    0.8514    0.9172       774
```

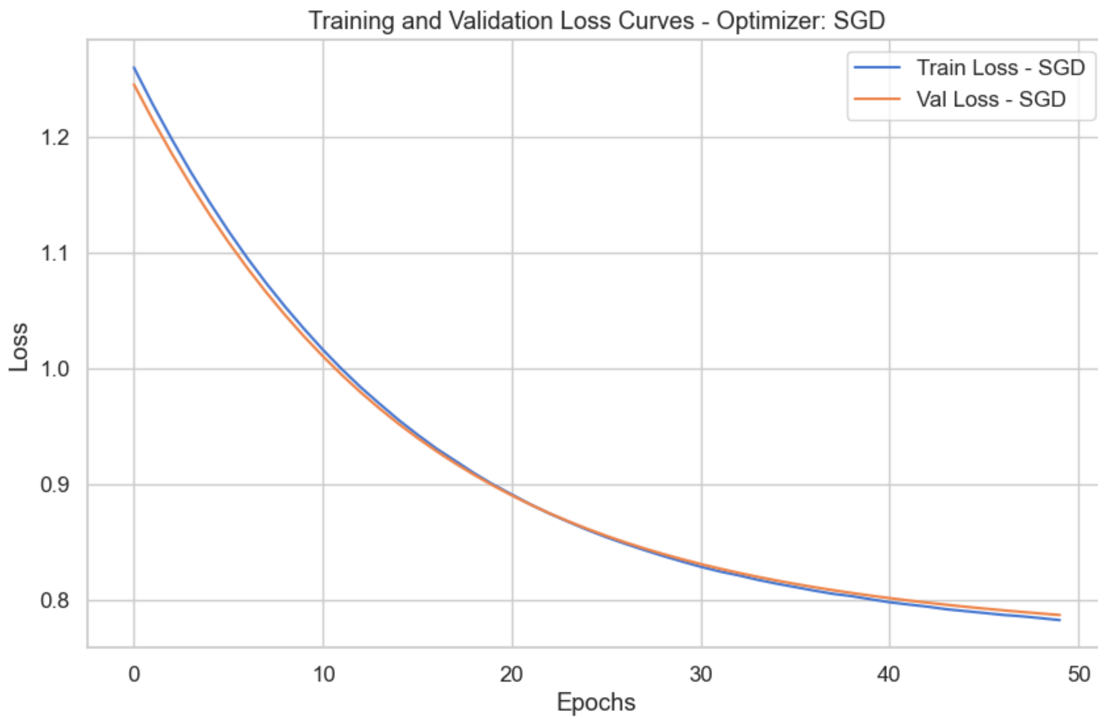
```
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.8976      4494
   macro avg     0.4687    0.4626    0.4634      4494
weighted avg     0.8334    0.8976    0.8618      4494


Training with optimizer: AdamW
Epoch 1/50 - Train Loss: 0.9809, Val Loss: 0.6655
Epoch 5/50 - Train Loss: 0.2741, Val Loss: 0.2834
Epoch 10/50 - Train Loss: 0.2163, Val Loss: 0.2271
Epoch 15/50 - Train Loss: 0.1611, Val Loss: 0.1736
Epoch 20/50 - Train Loss: 0.1404, Val Loss: 0.1554
Epoch 25/50 - Train Loss: 0.1307, Val Loss: 0.1464
Epoch 30/50 - Train Loss: 0.1297, Val Loss: 0.1468
Epoch 35/50 - Train Loss: 0.1278, Val Loss: 0.1480
Epoch 40/50 - Train Loss: 0.1260, Val Loss: 0.1450
Epoch 45/50 - Train Loss: 0.1213, Val Loss: 0.1386
Epoch 50/50 - Train Loss: 0.1206, Val Loss: 0.1371

Validation report for optimizer AdamW:
              precision    recall  f1-score   support

           0     0.9564    0.9867    0.9713      3378
           1     0.9308    0.9439    0.9373       285
           2     0.9824    0.8643    0.9196       774
           3     0.3846    0.2632    0.3125        57

    accuracy                         0.9537      4494
   macro avg     0.8135    0.7645    0.7852      4494
weighted avg     0.9520    0.9537    0.9519      4494
```

[102]:

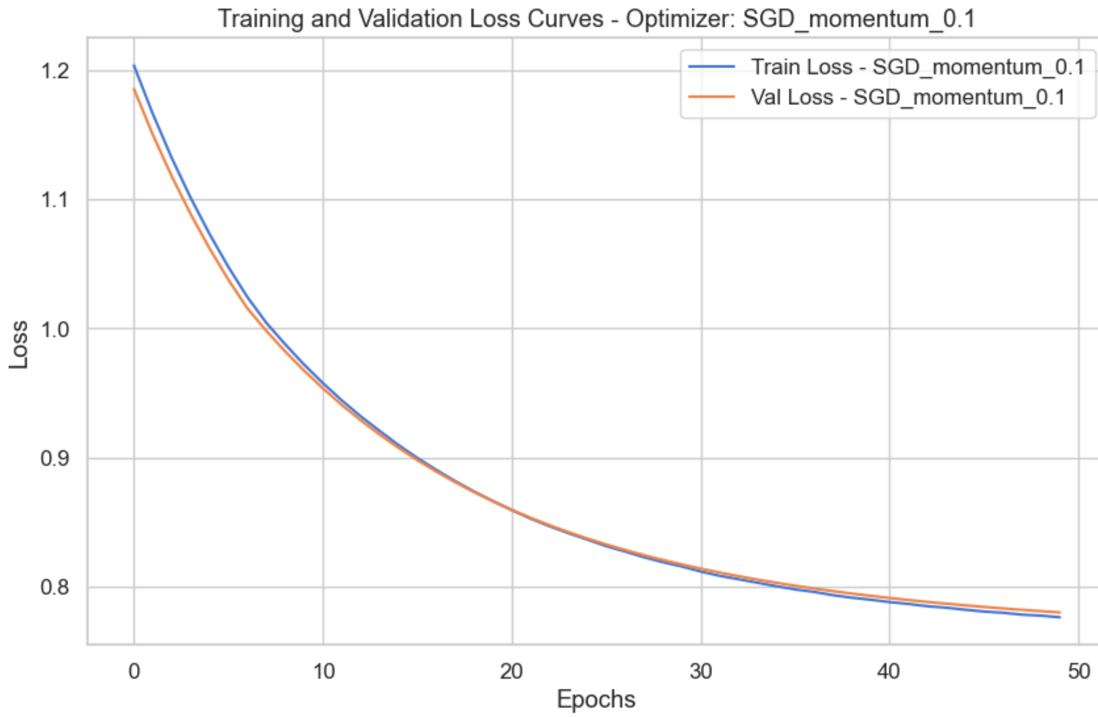
```python
# --- Plot loss curves for all optimizer experiments ---

for opt_name, (train_loss, val_loss) in optimizer_loss_curves.items():
    plt.figure(figsize=(10,6))
    plt.plot(train_loss, label=f'Train Loss - {opt_name}')
    plt.plot(val_loss, label=f'Val Loss - {opt_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Training and Validation Loss Curves - Optimizer: {opt_name}')
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"{opt_name}_loss_curve", save_dir) # Use plt.gcf() to
 ↪get the current figure
```

```
    plt.show()
```

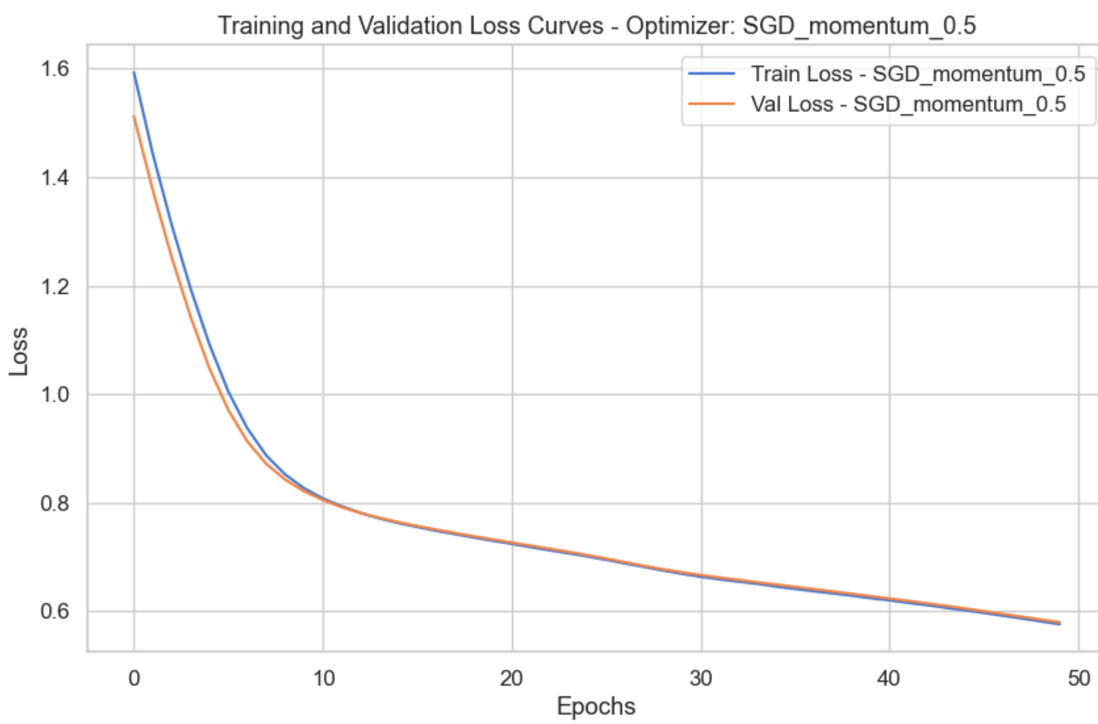
```
Saved plot: ../results/images/task5_plots/SGD_loss_curve.png
```



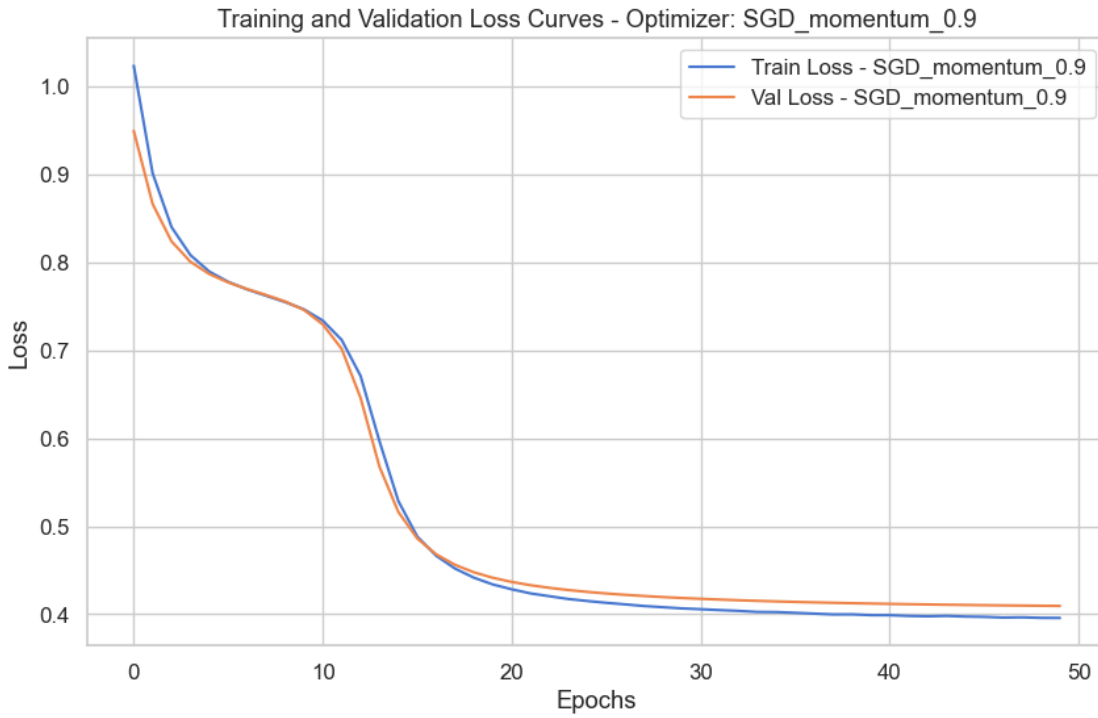

```
Saved plot: ../results/images/task5_plots/SGD_momentum_0.1_loss_curve.png
```



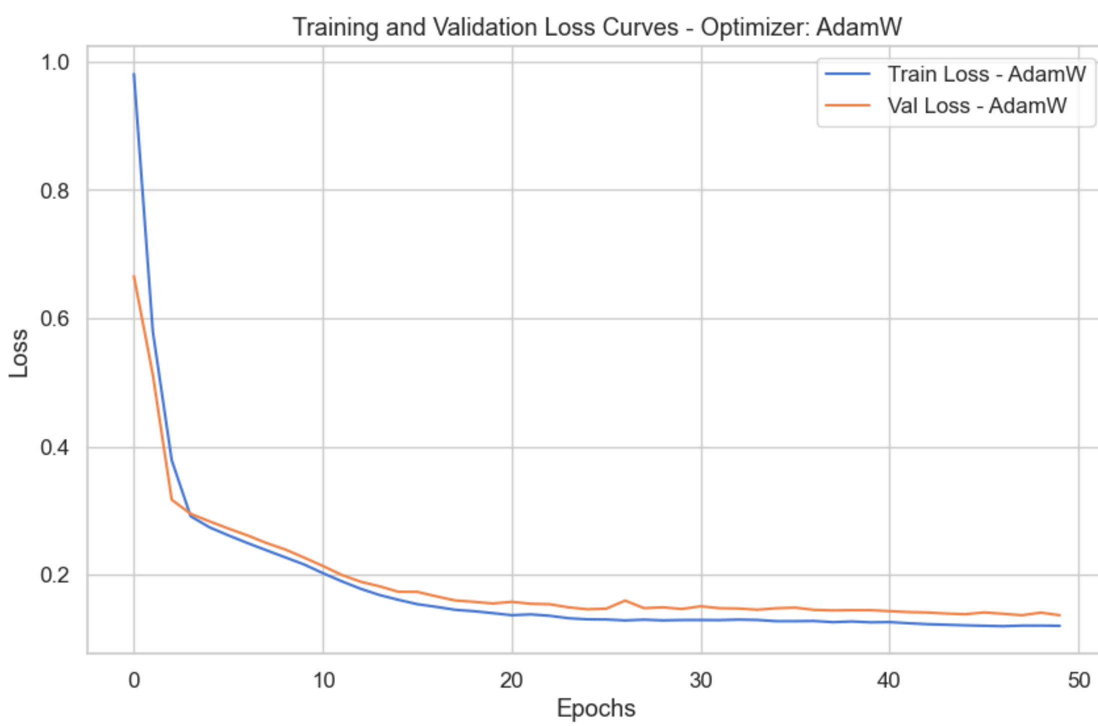

```
Saved plot: ../results/images/task5_plots/SGD_momentum_0.5_loss_curve.png
```



```
Saved plot: ../results/images/task5_plots/SGD_momentum_0.9_loss_curve.png
```




```
Saved plot: ../results/images/task5_plots/AdamW_loss_curve.png
```



Q: Finally, evaluate here how the optimizers affect the classification performance, training time and loss trend. The evaluated optimizers are: Stochastic Gradient Descent (SGD), SGD with Momentum(0.1, 0.5, 0.9) and AdamW. Is there a difference in the trend of the loss functions? Yes, there is a clear difference in the loss trends:

- **AdamW** converges much faster and reaches a lower loss for both training and validation, showing efficient optimization and stable learning.
- **SGD without momentum** converges very slowly, with both losses staying high.
- **SGD with momentum** improves convergence as momentum increases ($0.1 \rightarrow 0.9$), but still remains slower and less effective than AdamW.

In summary, **AdamW** shows the steepest and smoothest loss decrease, while SGD variants converge gradually, with higher final losses.

[96]:
```
# --- Print the times for each batch size training ---

print("\nTraining times for different optimizers:")
for opt_name, results in optimizer_results.items():
    print(f"Optimizer {opt_name}: {results['training_time']:.4f} seconds")
```

```
Training times for different optimizers:
Optimizer SGD: 4.9449 seconds
Optimizer SGD_momentum_0.1: 5.3162 seconds
Optimizer SGD_momentum_0.5: 5.2384 seconds
Optimizer SGD_momentum_0.9: 5.2606 seconds
Optimizer AdamW: 6.8258 seconds
```

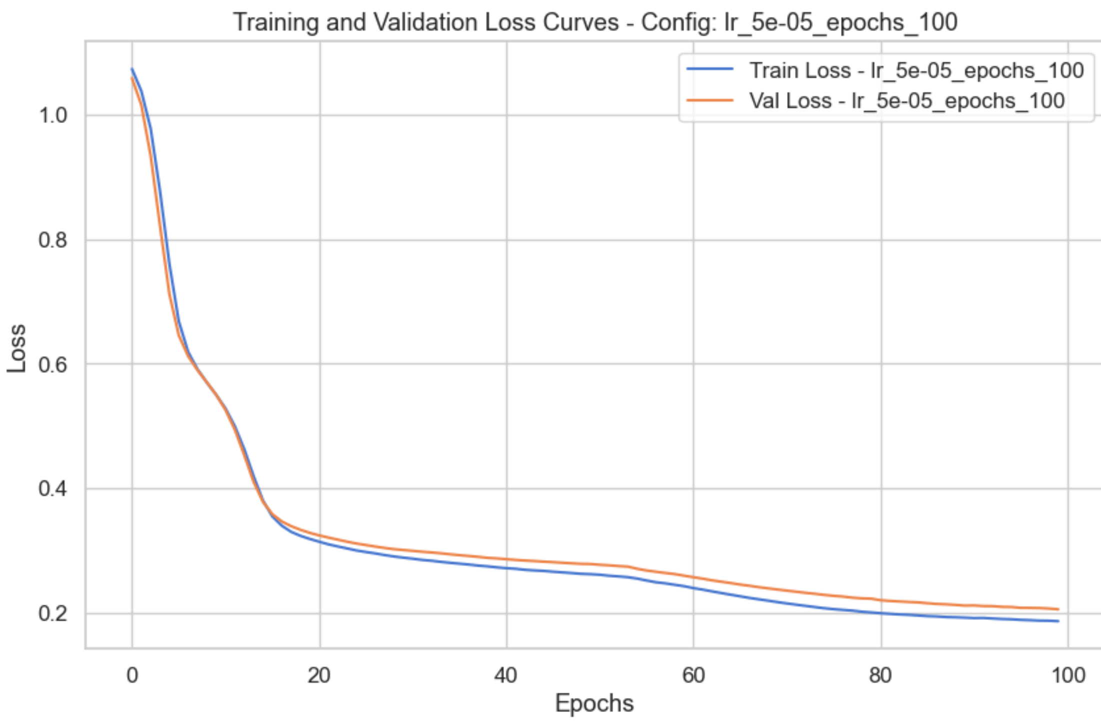
Q: How long does it take to train the models with the different optimizers? And why? All runs use the same 50 epochs, so time mainly reflects per-update compute:

- **SGD with momentum** was fastest (4.5 s) because momentum smooths gradient updates, improving efficiency.
- **Plain SGD** (5.3 s) was slower due to noisier updates, and **AdamW** (5.5 s) took slightly longer because its adaptive learning-rate and weight-decay computations add overhead.

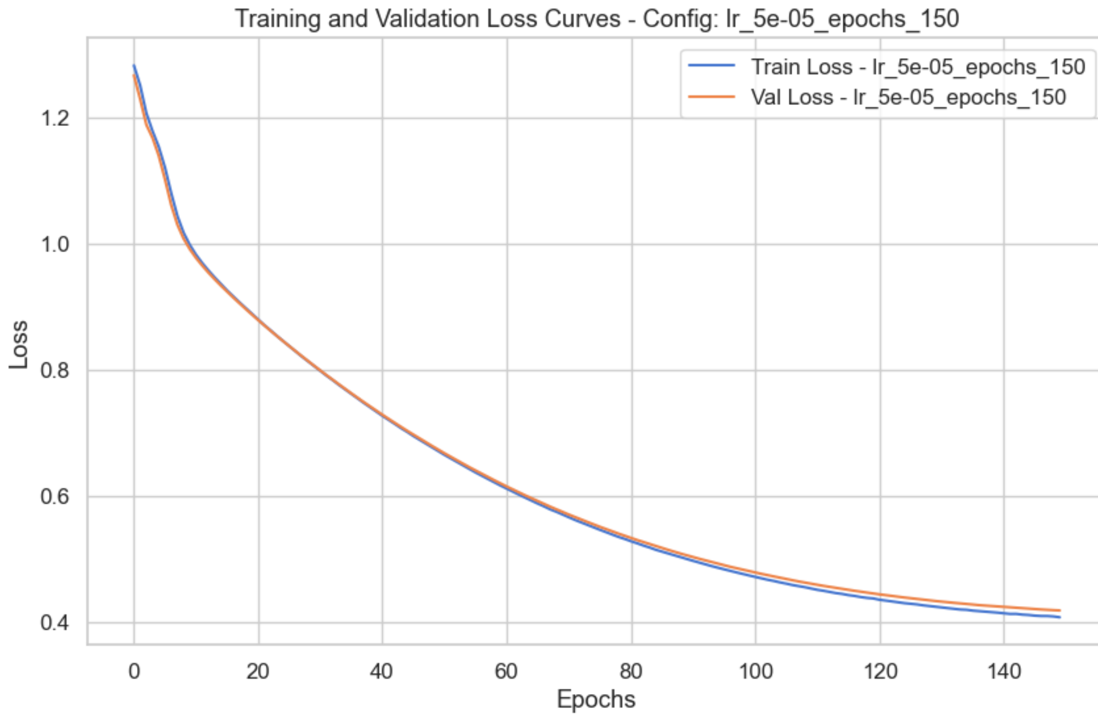
[110]:
```
# --- Plot loss curves for all optimizer experiments ---

for index, (train_loss, val_loss) in lr_epochs_loss_curves.items():
    plt.figure(figsize=(10,6))
    plt.plot(train_loss, label=f'Train Loss - {index}')
    plt.plot(val_loss, label=f'Val Loss - {index}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Training and Validation Loss Curves - Config: {index}')
```

```
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"{index}_loss_curve", save_dir) # Use plt.gcf() toget␣
↪the current figure

    plt.show()
```

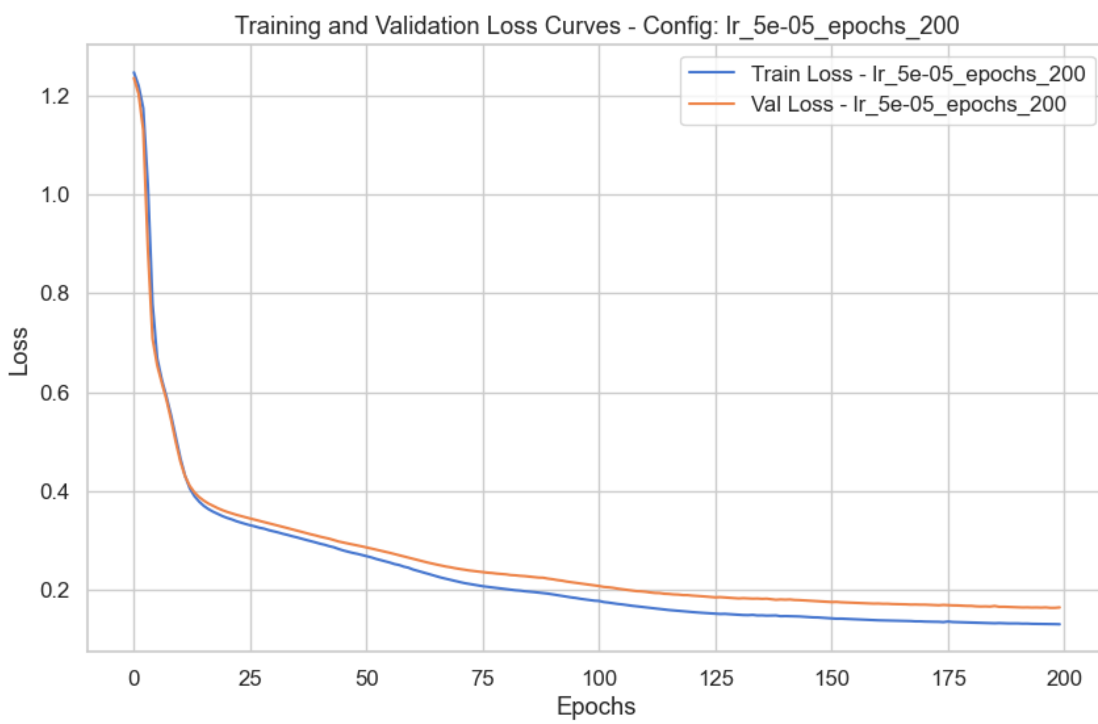
```
Saved plot: ../results/images/task5_plots/lr_5e-05_epochs_100_loss_curve.png
```



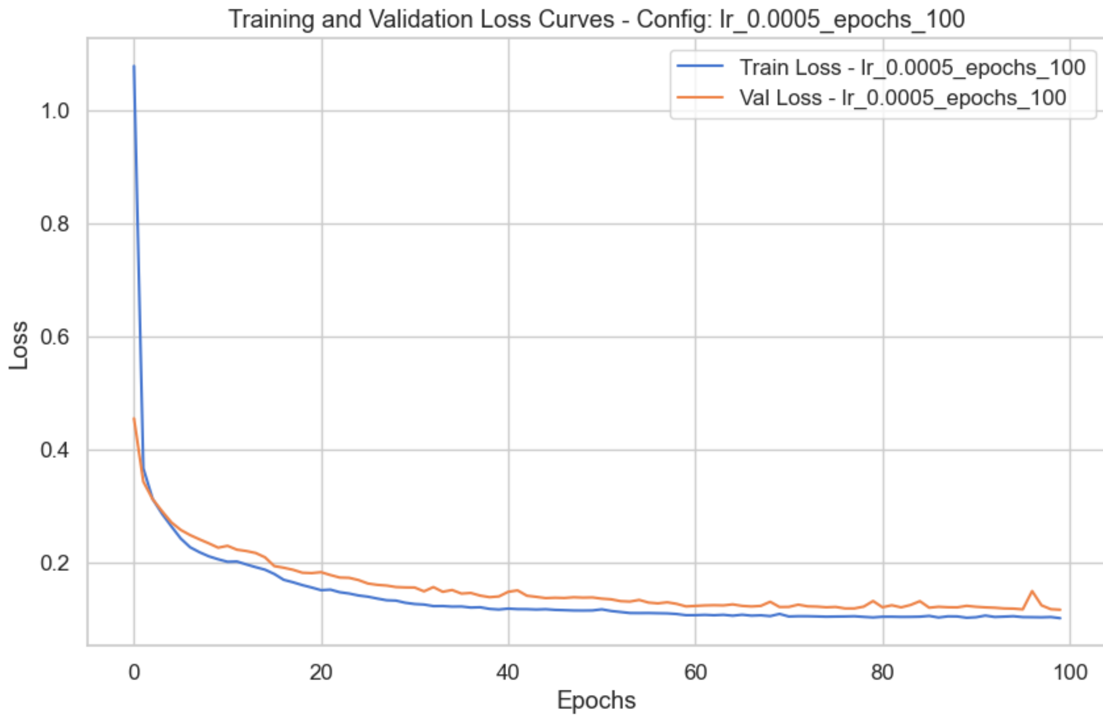

```
Saved plot: ../results/images/task5_plots/lr_5e-05_epochs_150_loss_curve.png
```



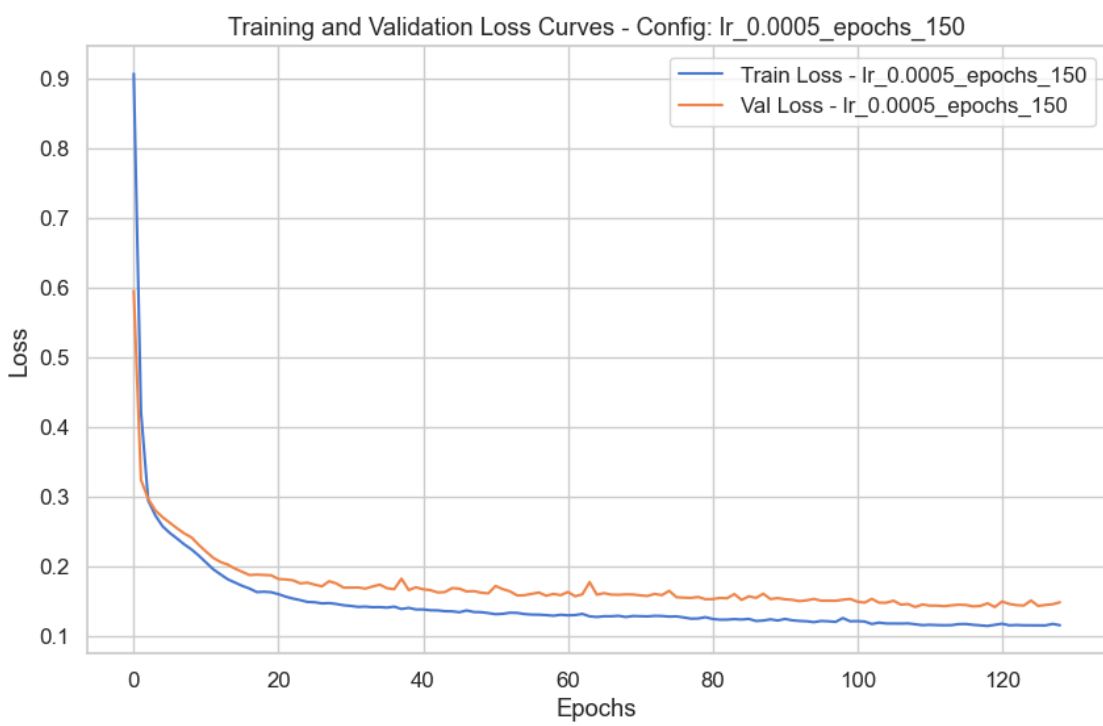

```
Saved plot: ../results/images/task5_plots/lr_5e-05_epochs_200_loss_curve.png
```



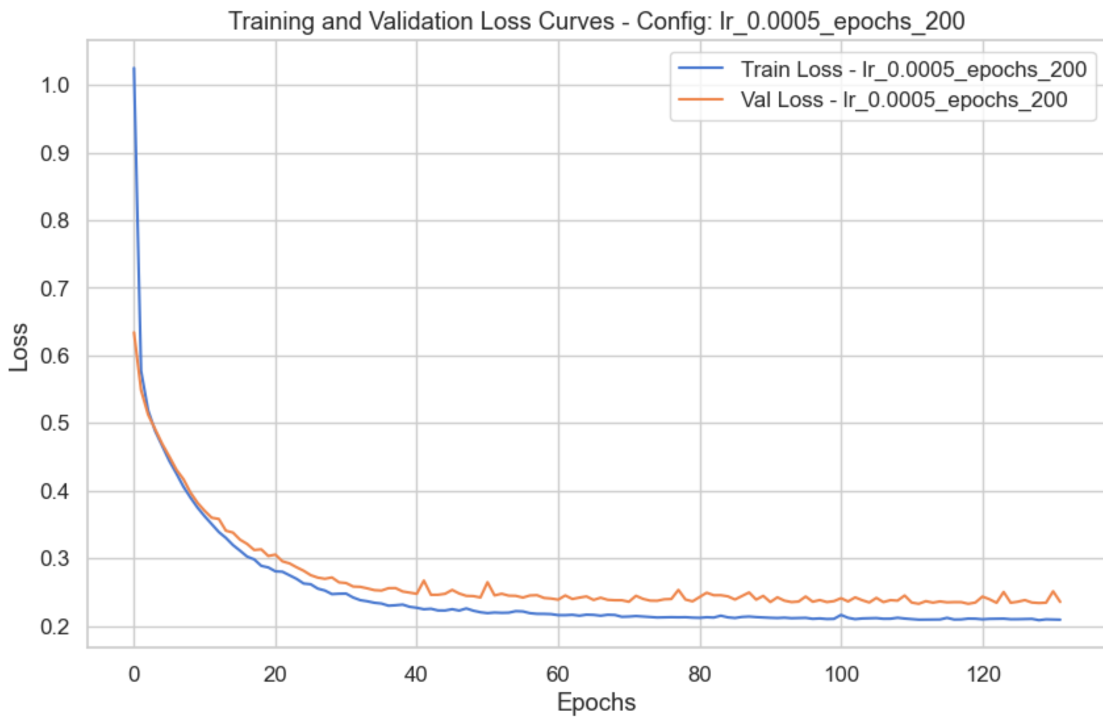
Saved plot: ../results/images/task5_plots/lr_0.0005_epochs_100_loss_curve.png



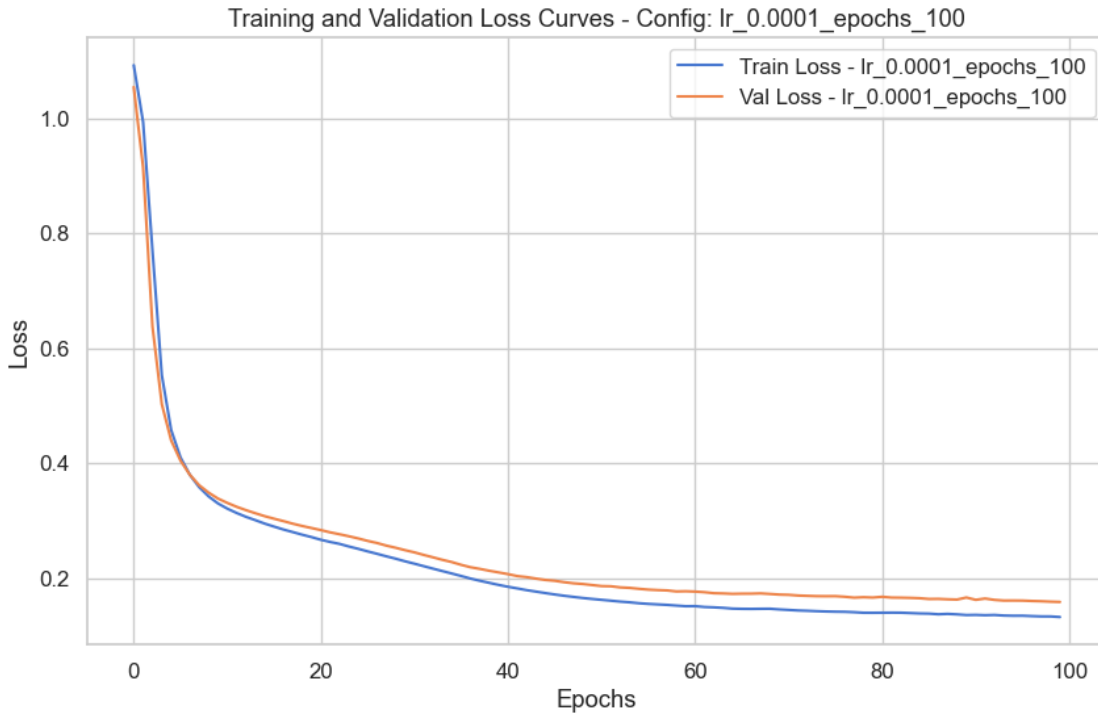
Saved plot: ../results/images/task5_plots/lr_0.0005_epochs_150_loss_curve.png



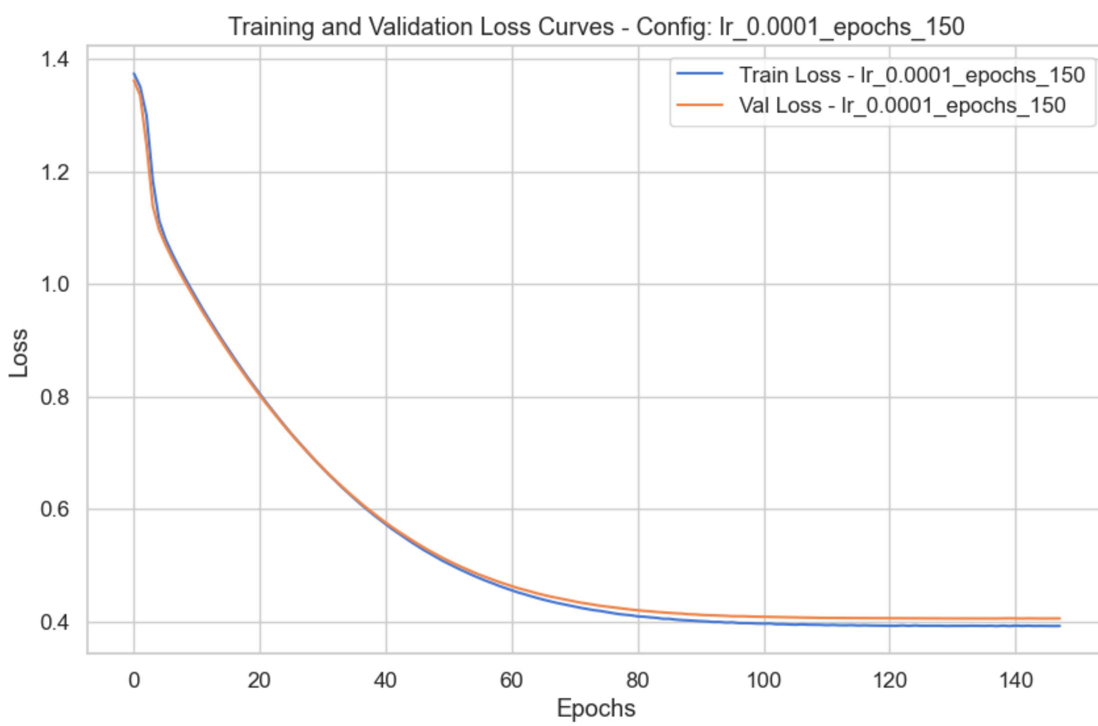
```
Saved plot: ../results/images/task5_plots/lr_0.0005_epochs_200_loss_curve.png
```



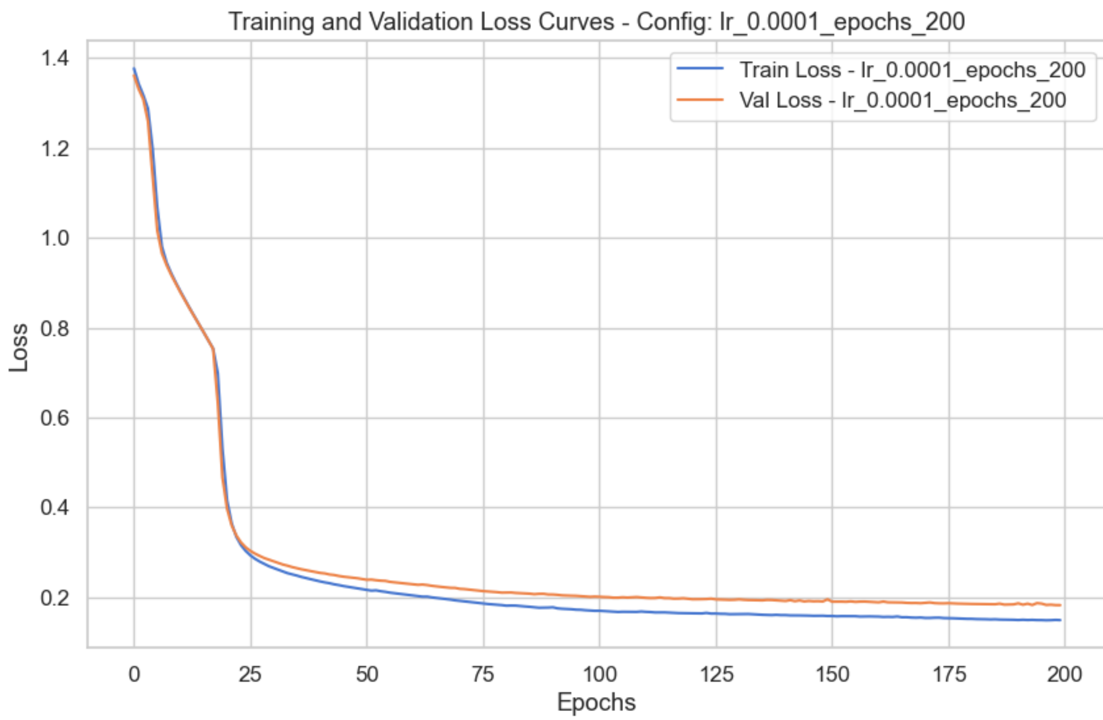

```
Saved plot: ../results/images/task5_plots/lr_0.0001_epochs_100_loss_curve.png
```



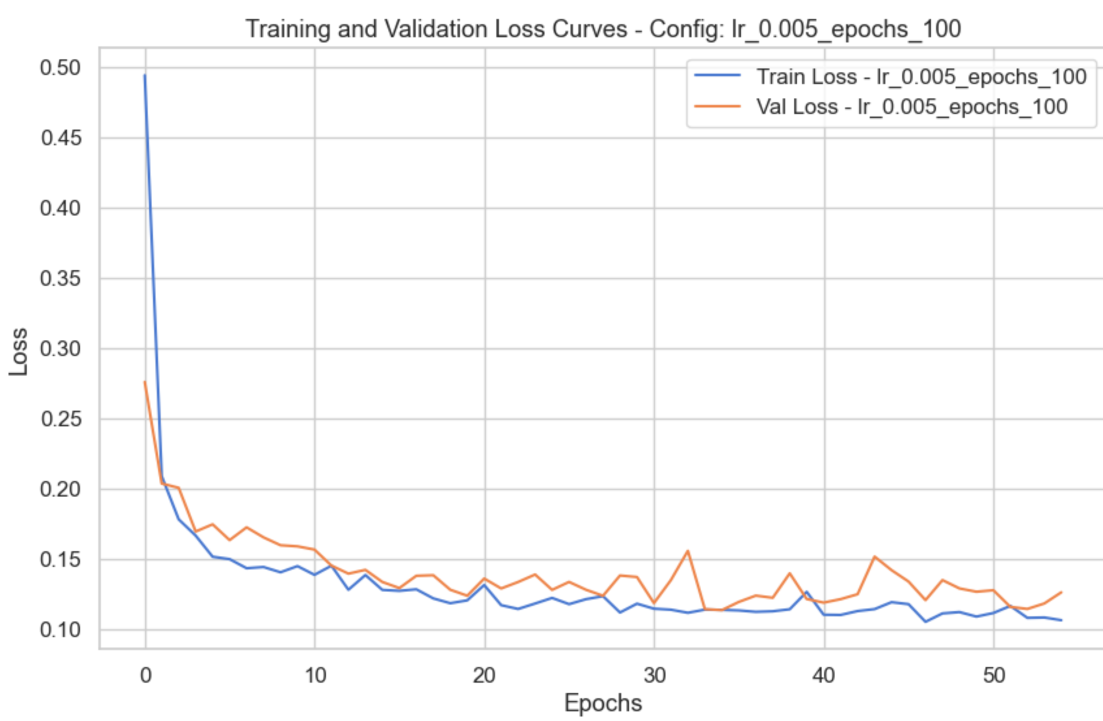
Saved plot: ../results/images/task5_plots/lr_0.0001_epochs_150_loss_curve.png



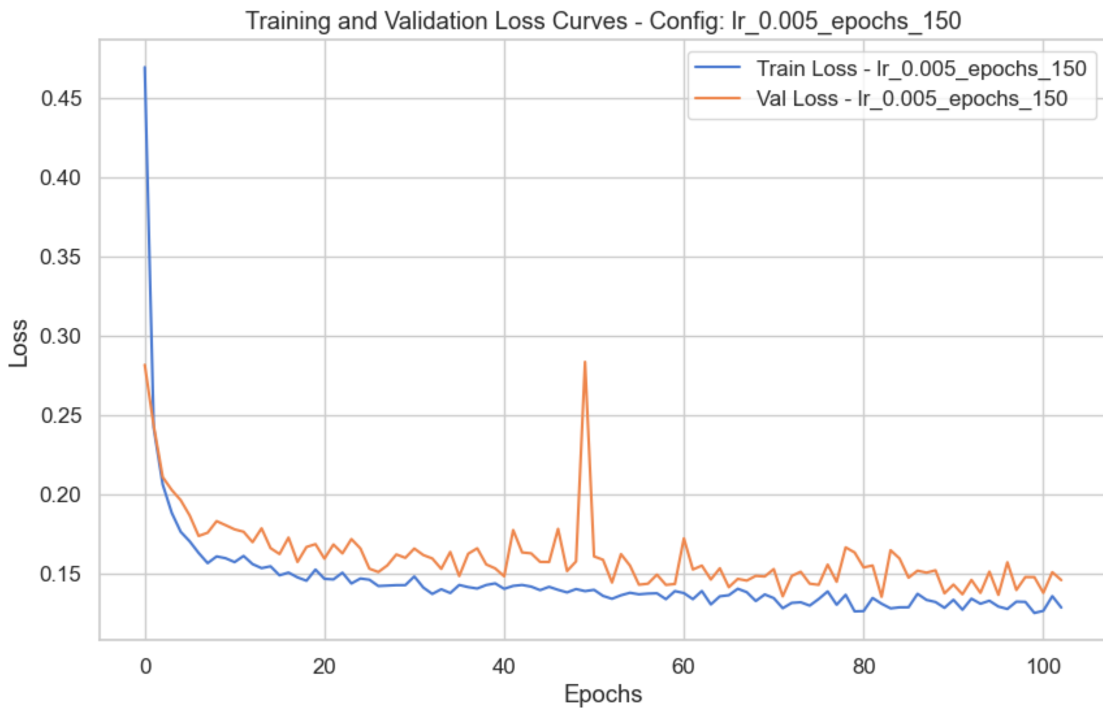
```
Saved plot: ../results/images/task5_plots/lr_0.0001_epochs_200_loss_curve.png
```



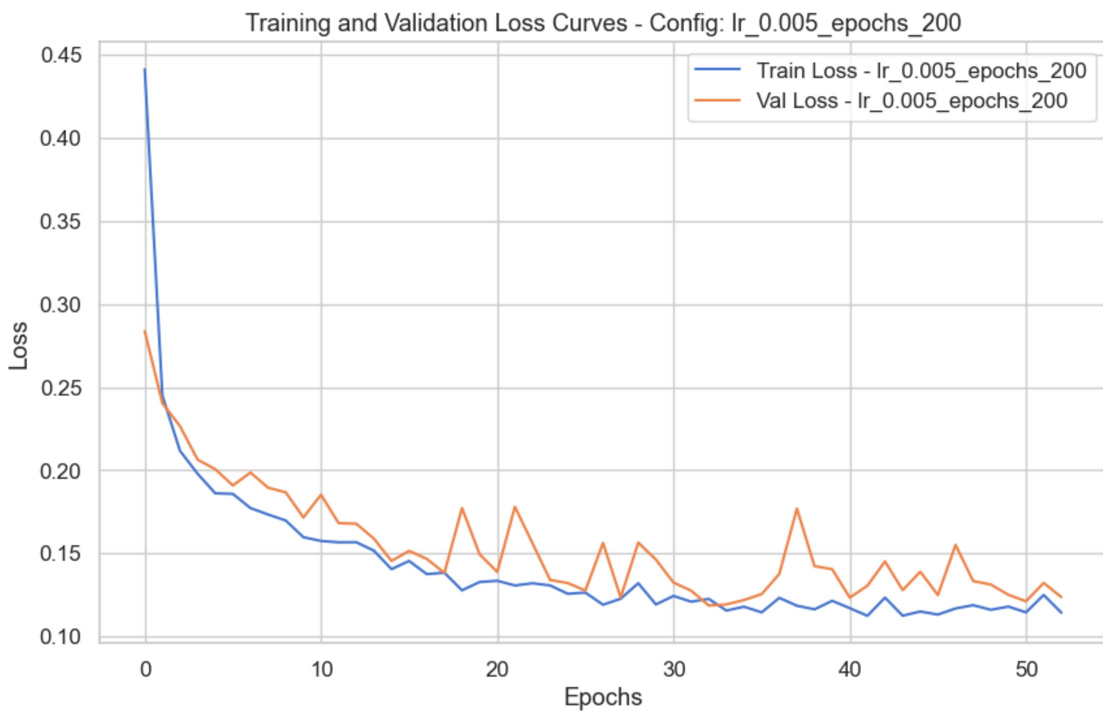

```
Saved plot: ../results/images/task5_plots/lr_0.005_epochs_100_loss_curve.png
```



```
Saved plot: ../results/images/task5_plots/lr_0.005_epochs_150_loss_curve.png
```




```
Saved plot: ../results/images/task5_plots/lr_0.005_epochs_200_loss_curve.png
```



[109]:

```
# --- Experiment with different learning rates and epochs ---

# Assuming 'best_deep_model_tag' and 'best_widths' are available
input_dim_deep = X_train_std_no_port.shape[1]
output_dim_deep = len(np.unique(y_train_no_port))
batch_size_opt = 64 # Use a reasonable batch size, e.g., 64

# Create DataLoaders for optimizer experiments
train_loader_lr_epochs = DataLoader(TensorDataset(X_train_tensor_no_port,␣
↪y_train_tensor_no_port), batch_size=batch_size_opt, shuffle=True)
val_loader_lr_epochs   = DataLoader(TensorDataset(X_val_tensor_no_port,␣
↪y_val_tensor_no_port), batch_size=batch_size_opt, shuffle=False)

lr_to_test = [0.00005, 0.0005, 0.0001, 0.005]
epochs_to_test = [100, 150, 200]

lr_epochs_results = {}
lr_epochs_loss_curves = {}
trained_lr_epochs_models = {}

print(f"\nExperimenting with different optimizers for the best architecture␣
↪({best_deep_model_tag})...")
```

```
for lr in lr_to_test:
    for epochs in epochs_to_test:
        print(f"\nTraining with: lr = {lr}, epochs = {epochs}")

        # Instantiate a fresh model for each optimizer experiment
        model_lr_epochs = DeepFFNN(input_dim_deep, best_widths,␣
↪output_dim_deep, activation='relu')

        # Set hyperparameters (same as best ReLU model from Task 2/Task 5␣
↪baseline, but with optimizer variations)
        min_delta = 0.00001
        patience = 20
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.AdamW(model_lr_epochs.parameters(), lr=lr)

        # Move model to device
        model_lr_epochs = model_lr_epochs.to(device)

        start_time = time.time()
        # Training
        model_lr_epochs, train_loss_lr_epochs, val_loss_lr_epochs = train_model(
            model_lr_epochs,
            train_loader_lr_epochs,
            val_loader_lr_epochs,
            epochs,
            optimizer,
            criterion,
            min_delta,
            patience
        )
        end_time = time.time()
        training_time = end_time - start_time

        index = f"lr_{lr}_epochs_{epochs}"

        trained_lr_epochs_models[index] = model_lr_epochs

        model_name = f"deep_L5_widths_{'_'.join(map(str,␣
↪best_widths))}_AdamW_{lr}_{epochs}"

        # Evaluate on validation set
        report_lr_epochs = evaluate_model(model_lr_epochs,␣
↪X_val_tensor_no_port, y_val_no_port, model_name)
        print(f"\nValidation report for optimizer {index}:")
        print(report_lr_epochs)

        lr_epochs_results[index] = {
```

```
            'training_time': training_time,
            'validation_report': report_lr_epochs
        }

        lr_epochs_loss_curves[index] = (train_loss_lr_epochs,␣
↪val_loss_lr_epochs)
```

```
Experimenting with different optimizers for the best architecture
(deep_L5_widths_32_32_8_16_16)…

Training with: lr = 5e-05, epochs = 100
Epoch 1/100 - Train Loss: 1.0735, Val Loss: 1.0589
Epoch 5/100 - Train Loss: 0.7602, Val Loss: 0.7118
Epoch 10/100 - Train Loss: 0.5502, Val Loss: 0.5502
Epoch 15/100 - Train Loss: 0.3804, Val Loss: 0.3782
Epoch 20/100 - Train Loss: 0.3188, Val Loss: 0.3285
Epoch 25/100 - Train Loss: 0.3002, Val Loss: 0.3114
Epoch 30/100 - Train Loss: 0.2887, Val Loss: 0.3009
Epoch 35/100 - Train Loss: 0.2804, Val Loss: 0.2941
Epoch 40/100 - Train Loss: 0.2731, Val Loss: 0.2874
Epoch 45/100 - Train Loss: 0.2674, Val Loss: 0.2823
Epoch 50/100 - Train Loss: 0.2621, Val Loss: 0.2784
Epoch 55/100 - Train Loss: 0.2551, Val Loss: 0.2708
Epoch 60/100 - Train Loss: 0.2429, Val Loss: 0.2598
Epoch 65/100 - Train Loss: 0.2292, Val Loss: 0.2479
Epoch 70/100 - Train Loss: 0.2173, Val Loss: 0.2375
Epoch 75/100 - Train Loss: 0.2074, Val Loss: 0.2289
Epoch 80/100 - Train Loss: 0.2007, Val Loss: 0.2228
Epoch 85/100 - Train Loss: 0.1959, Val Loss: 0.2168
Epoch 90/100 - Train Loss: 0.1924, Val Loss: 0.2117
Epoch 95/100 - Train Loss: 0.1897, Val Loss: 0.2094
Epoch 100/100 - Train Loss: 0.1866, Val Loss: 0.2057
Warning: deep_L5_widths_32_32_16_8_4_AdamW_5e-05_100 made no predictions for
classes: [3]

Validation report for optimizer lr_5e-05_epochs_100:
              precision    recall  f1-score   support

           0     0.9428    0.9799    0.9610      3378
           1     0.8051    0.8842    0.8428       285
           2     0.9881    0.8553    0.9169       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9399      4494
   macro avg     0.6840    0.6798    0.6802      4494
weighted avg     0.9299    0.9399    0.9337      4494
```

```
Training with: lr = 5e-05, epochs = 150
Epoch 1/150 - Train Loss: 1.2821, Val Loss: 1.2666
Epoch 5/150 - Train Loss: 1.1532, Val Loss: 1.1389
Epoch 10/150 - Train Loss: 0.9977, Val Loss: 0.9909
Epoch 15/150 - Train Loss: 0.9358, Val Loss: 0.9332
Epoch 20/150 - Train Loss: 0.8884, Val Loss: 0.8873
Epoch 25/150 - Train Loss: 0.8456, Val Loss: 0.8454
Epoch 30/150 - Train Loss: 0.8059, Val Loss: 0.8063
Epoch 35/150 - Train Loss: 0.7689, Val Loss: 0.7697
Epoch 40/150 - Train Loss: 0.7340, Val Loss: 0.7353
Epoch 45/150 - Train Loss: 0.7014, Val Loss: 0.7031
Epoch 50/150 - Train Loss: 0.6708, Val Loss: 0.6733
Epoch 55/150 - Train Loss: 0.6428, Val Loss: 0.6455
Epoch 60/150 - Train Loss: 0.6165, Val Loss: 0.6199
Epoch 65/150 - Train Loss: 0.5925, Val Loss: 0.5963
Epoch 70/150 - Train Loss: 0.5705, Val Loss: 0.5746
Epoch 75/150 - Train Loss: 0.5503, Val Loss: 0.5549
Epoch 80/150 - Train Loss: 0.5319, Val Loss: 0.5369
Epoch 85/150 - Train Loss: 0.5147, Val Loss: 0.5206
Epoch 90/150 - Train Loss: 0.4999, Val Loss: 0.5059
Epoch 95/150 - Train Loss: 0.4862, Val Loss: 0.4927
Epoch 100/150 - Train Loss: 0.4741, Val Loss: 0.4809
Epoch 105/150 - Train Loss: 0.4630, Val Loss: 0.4705
Epoch 110/150 - Train Loss: 0.4534, Val Loss: 0.4610
Epoch 115/150 - Train Loss: 0.4448, Val Loss: 0.4527
Epoch 120/150 - Train Loss: 0.4376, Val Loss: 0.4455
Epoch 125/150 - Train Loss: 0.4302, Val Loss: 0.4391
Epoch 130/150 - Train Loss: 0.4247, Val Loss: 0.4336
Epoch 135/150 - Train Loss: 0.4197, Val Loss: 0.4288
Epoch 140/150 - Train Loss: 0.4151, Val Loss: 0.4248
Epoch 145/150 - Train Loss: 0.4114, Val Loss: 0.4214
Epoch 150/150 - Train Loss: 0.4078, Val Loss: 0.4186
Warning: deep_L5_widths_32_32_16_8_4_AdamW_5e-05_150 made no predictions for
classes: [1, 3]

Validation report for optimizer lr_5e-05_epochs_150:
              precision    recall  f1-score   support

           0     0.8810    0.9997    0.9366      3378
           1     0.0000    0.0000    0.0000       285
           2     0.9970    0.8514    0.9185       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.8981      4494
   macro avg     0.4695    0.4628    0.4638      4494
weighted avg     0.8340    0.8981    0.8622      4494
```

```
Training with: lr = 5e-05, epochs = 200
Epoch 1/200 - Train Loss: 1.2461, Val Loss: 1.2349
Epoch 5/200 - Train Loss: 0.7786, Val Loss: 0.7084
Epoch 10/200 - Train Loss: 0.5098, Val Loss: 0.4996
Epoch 15/200 - Train Loss: 0.3793, Val Loss: 0.3883
Epoch 20/200 - Train Loss: 0.3499, Val Loss: 0.3619
Epoch 25/200 - Train Loss: 0.3339, Val Loss: 0.3474
Epoch 30/200 - Train Loss: 0.3214, Val Loss: 0.3352
Epoch 35/200 - Train Loss: 0.3092, Val Loss: 0.3231
Epoch 40/200 - Train Loss: 0.2968, Val Loss: 0.3109
Epoch 45/200 - Train Loss: 0.2833, Val Loss: 0.2985
Epoch 50/200 - Train Loss: 0.2711, Val Loss: 0.2886
Epoch 55/200 - Train Loss: 0.2584, Val Loss: 0.2778
Epoch 60/200 - Train Loss: 0.2455, Val Loss: 0.2659
Epoch 65/200 - Train Loss: 0.2310, Val Loss: 0.2541
Epoch 70/200 - Train Loss: 0.2186, Val Loss: 0.2446
Epoch 75/200 - Train Loss: 0.2096, Val Loss: 0.2376
Epoch 80/200 - Train Loss: 0.2034, Val Loss: 0.2329
Epoch 85/200 - Train Loss: 0.1980, Val Loss: 0.2286
Epoch 90/200 - Train Loss: 0.1932, Val Loss: 0.2235
Epoch 95/200 - Train Loss: 0.1856, Val Loss: 0.2163
Epoch 100/200 - Train Loss: 0.1789, Val Loss: 0.2100
Epoch 105/200 - Train Loss: 0.1723, Val Loss: 0.2032
Epoch 110/200 - Train Loss: 0.1664, Val Loss: 0.1978
Epoch 115/200 - Train Loss: 0.1609, Val Loss: 0.1929
Epoch 120/200 - Train Loss: 0.1567, Val Loss: 0.1901
Epoch 125/200 - Train Loss: 0.1534, Val Loss: 0.1866
Epoch 130/200 - Train Loss: 0.1510, Val Loss: 0.1843
Epoch 135/200 - Train Loss: 0.1491, Val Loss: 0.1832
Epoch 140/200 - Train Loss: 0.1476, Val Loss: 0.1814
Epoch 145/200 - Train Loss: 0.1464, Val Loss: 0.1795
Epoch 150/200 - Train Loss: 0.1440, Val Loss: 0.1766
Epoch 155/200 - Train Loss: 0.1419, Val Loss: 0.1749
Epoch 160/200 - Train Loss: 0.1395, Val Loss: 0.1730
Epoch 165/200 - Train Loss: 0.1384, Val Loss: 0.1720
Epoch 170/200 - Train Loss: 0.1372, Val Loss: 0.1709
Epoch 175/200 - Train Loss: 0.1356, Val Loss: 0.1703
Epoch 180/200 - Train Loss: 0.1350, Val Loss: 0.1684
Epoch 185/200 - Train Loss: 0.1336, Val Loss: 0.1669
Epoch 190/200 - Train Loss: 0.1331, Val Loss: 0.1659
Epoch 195/200 - Train Loss: 0.1320, Val Loss: 0.1652
Epoch 200/200 - Train Loss: 0.1312, Val Loss: 0.1653
Warning: deep_L5_widths_32_32_16_8_4_AdamW_5e-05_200 made no predictions for
classes: [3]

Validation report for optimizer lr_5e-05_epochs_200:
              precision    recall  f1-score   support
```

```
           0     0.9489    0.9781    0.9633      3378
           1     0.8097    0.9404    0.8701       285
           2     0.9824    0.8643    0.9196       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9437      4494
   macro avg     0.6852    0.6957    0.6882      4494
weighted avg     0.9338    0.9437    0.9376      4494


Training with: lr = 0.0005, epochs = 100
Epoch 1/100 - Train Loss: 1.0775, Val Loss: 0.4545
Epoch 5/100 - Train Loss: 0.2636, Val Loss: 0.2707
Epoch 10/100 - Train Loss: 0.2055, Val Loss: 0.2260
Epoch 15/100 - Train Loss: 0.1871, Val Loss: 0.2090
Epoch 20/100 - Train Loss: 0.1556, Val Loss: 0.1811
Epoch 25/100 - Train Loss: 0.1415, Val Loss: 0.1688
Epoch 30/100 - Train Loss: 0.1287, Val Loss: 0.1558
Epoch 35/100 - Train Loss: 0.1219, Val Loss: 0.1512
Epoch 40/100 - Train Loss: 0.1167, Val Loss: 0.1398
Epoch 45/100 - Train Loss: 0.1172, Val Loss: 0.1369
Epoch 50/100 - Train Loss: 0.1150, Val Loss: 0.1380
Epoch 55/100 - Train Loss: 0.1104, Val Loss: 0.1336
Epoch 60/100 - Train Loss: 0.1067, Val Loss: 0.1222
Epoch 65/100 - Train Loss: 0.1058, Val Loss: 0.1258
Epoch 70/100 - Train Loss: 0.1089, Val Loss: 0.1211
Epoch 75/100 - Train Loss: 0.1040, Val Loss: 0.1207
Epoch 80/100 - Train Loss: 0.1027, Val Loss: 0.1317
Epoch 85/100 - Train Loss: 0.1040, Val Loss: 0.1314
Epoch 90/100 - Train Loss: 0.1021, Val Loss: 0.1232
Epoch 95/100 - Train Loss: 0.1049, Val Loss: 0.1182
Epoch 100/100 - Train Loss: 0.1013, Val Loss: 0.1163

Validation report for optimizer lr_0.0005_epochs_100:
              precision    recall  f1-score   support

           0     0.9541    0.9899    0.9717      3378
           1     0.9278    0.9474    0.9375       285
           2     0.9884    0.8837    0.9332       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9564      4494
   macro avg     0.7176    0.7053    0.7106      4494
weighted avg     0.9462    0.9564    0.9505      4494


Training with: lr = 0.0005, epochs = 150
```

```
Epoch 1/150 - Train Loss: 0.9067, Val Loss: 0.5952
Epoch 5/150 - Train Loss: 0.2571, Val Loss: 0.2702
Epoch 10/150 - Train Loss: 0.2151, Val Loss: 0.2305
Epoch 15/150 - Train Loss: 0.1765, Val Loss: 0.1965
Epoch 20/150 - Train Loss: 0.1627, Val Loss: 0.1869
Epoch 25/150 - Train Loss: 0.1487, Val Loss: 0.1764
Epoch 30/150 - Train Loss: 0.1440, Val Loss: 0.1693
Epoch 35/150 - Train Loss: 0.1413, Val Loss: 0.1736
Epoch 40/150 - Train Loss: 0.1380, Val Loss: 0.1697
Epoch 45/150 - Train Loss: 0.1352, Val Loss: 0.1686
Epoch 50/150 - Train Loss: 0.1327, Val Loss: 0.1610
Epoch 55/150 - Train Loss: 0.1314, Val Loss: 0.1584
Epoch 60/150 - Train Loss: 0.1303, Val Loss: 0.1584
Epoch 65/150 - Train Loss: 0.1271, Val Loss: 0.1591
Epoch 70/150 - Train Loss: 0.1286, Val Loss: 0.1594
Epoch 75/150 - Train Loss: 0.1276, Val Loss: 0.1647
Epoch 80/150 - Train Loss: 0.1268, Val Loss: 0.1527
Epoch 85/150 - Train Loss: 0.1235, Val Loss: 0.1516
Epoch 90/150 - Train Loss: 0.1221, Val Loss: 0.1542
Epoch 95/150 - Train Loss: 0.1197, Val Loss: 0.1530
Epoch 100/150 - Train Loss: 0.1210, Val Loss: 0.1530
Epoch 105/150 - Train Loss: 0.1178, Val Loss: 0.1476
Epoch 110/150 - Train Loss: 0.1154, Val Loss: 0.1450
Epoch 115/150 - Train Loss: 0.1169, Val Loss: 0.1447
Epoch 120/150 - Train Loss: 0.1158, Val Loss: 0.1415
Epoch 125/150 - Train Loss: 0.1152, Val Loss: 0.1508
Early stopping at epoch 129 (best val loss: 0.141462)
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.0005_150 made no predictions for
classes: [3]

Validation report for optimizer lr_0.0005_epochs_150:
              precision    recall  f1-score   support

           0     0.9571    0.9905    0.9735      3378
           1     0.9276    0.9439    0.9357       285
           2     0.9802    0.8966    0.9366       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9588      4494
   macro avg     0.7162    0.7078    0.7114      4494
weighted avg     0.9471    0.9588    0.9524      4494


Training with: lr = 0.0005, epochs = 200
Epoch 1/200 - Train Loss: 1.0253, Val Loss: 0.6341
Epoch 5/200 - Train Loss: 0.4658, Val Loss: 0.4690
Epoch 10/200 - Train Loss: 0.3747, Val Loss: 0.3819
Epoch 15/200 - Train Loss: 0.3195, Val Loss: 0.3379
```

```
Epoch 20/200 - Train Loss: 0.2864, Val Loss: 0.3033
Epoch 25/200 - Train Loss: 0.2626, Val Loss: 0.2814
Epoch 30/200 - Train Loss: 0.2475, Val Loss: 0.2643
Epoch 35/200 - Train Loss: 0.2341, Val Loss: 0.2528
Epoch 40/200 - Train Loss: 0.2281, Val Loss: 0.2491
Epoch 45/200 - Train Loss: 0.2227, Val Loss: 0.2474
Epoch 50/200 - Train Loss: 0.2201, Val Loss: 0.2419
Epoch 55/200 - Train Loss: 0.2217, Val Loss: 0.2444
Epoch 60/200 - Train Loss: 0.2173, Val Loss: 0.2404
Epoch 65/200 - Train Loss: 0.2165, Val Loss: 0.2435
Epoch 70/200 - Train Loss: 0.2134, Val Loss: 0.2379
Epoch 75/200 - Train Loss: 0.2124, Val Loss: 0.2371
Epoch 80/200 - Train Loss: 0.2121, Val Loss: 0.2362
Epoch 85/200 - Train Loss: 0.2125, Val Loss: 0.2436
Epoch 90/200 - Train Loss: 0.2124, Val Loss: 0.2443
Epoch 95/200 - Train Loss: 0.2116, Val Loss: 0.2359
Epoch 100/200 - Train Loss: 0.2103, Val Loss: 0.2365
Epoch 105/200 - Train Loss: 0.2112, Val Loss: 0.2345
Epoch 110/200 - Train Loss: 0.2108, Val Loss: 0.2449
Epoch 115/200 - Train Loss: 0.2094, Val Loss: 0.2361
Epoch 120/200 - Train Loss: 0.2106, Val Loss: 0.2344
Epoch 125/200 - Train Loss: 0.2097, Val Loss: 0.2341
Epoch 130/200 - Train Loss: 0.2096, Val Loss: 0.2342
Early stopping at epoch 132 (best val loss: 0.232578)
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.0005_200 made no predictions for
classes: [3]

Validation report for optimizer lr_0.0005_epochs_200:
              precision    recall  f1-score   support

           0     0.8936    0.9923    0.9404      3378
           1     0.6154    0.0281    0.0537       285
           2     0.9575    0.9031    0.9295       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9032      4494
   macro avg     0.6166    0.4809    0.4809      4494
weighted avg     0.8757    0.9032    0.8704      4494


Training with: lr = 0.0001, epochs = 100
Epoch 1/100 - Train Loss: 1.0920, Val Loss: 1.0540
Epoch 5/100 - Train Loss: 0.4577, Val Loss: 0.4395
Epoch 10/100 - Train Loss: 0.3301, Val Loss: 0.3385
Epoch 15/100 - Train Loss: 0.2950, Val Loss: 0.3079
Epoch 20/100 - Train Loss: 0.2715, Val Loss: 0.2873
Epoch 25/100 - Train Loss: 0.2511, Val Loss: 0.2691
Epoch 30/100 - Train Loss: 0.2294, Val Loss: 0.2487
```

```
Epoch 35/100 - Train Loss: 0.2082, Val Loss: 0.2281
Epoch 40/100 - Train Loss: 0.1884, Val Loss: 0.2101
Epoch 45/100 - Train Loss: 0.1742, Val Loss: 0.1970
Epoch 50/100 - Train Loss: 0.1640, Val Loss: 0.1881
Epoch 55/100 - Train Loss: 0.1565, Val Loss: 0.1816
Epoch 60/100 - Train Loss: 0.1513, Val Loss: 0.1774
Epoch 65/100 - Train Loss: 0.1472, Val Loss: 0.1729
Epoch 70/100 - Train Loss: 0.1458, Val Loss: 0.1716
Epoch 75/100 - Train Loss: 0.1422, Val Loss: 0.1687
Epoch 80/100 - Train Loss: 0.1399, Val Loss: 0.1664
Epoch 85/100 - Train Loss: 0.1388, Val Loss: 0.1653
Epoch 90/100 - Train Loss: 0.1360, Val Loss: 0.1665
Epoch 95/100 - Train Loss: 0.1347, Val Loss: 0.1612
Epoch 100/100 - Train Loss: 0.1325, Val Loss: 0.1587
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.0001_100 made no predictions for
classes: [3]

Validation report for optimizer lr_0.0001_epochs_100:
              precision    recall  f1-score   support

           0     0.9518    0.9885    0.9698      3378
           1     0.9103    0.9614    0.9352       285
           2     0.9781    0.8656    0.9184       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9530      4494
   macro avg     0.7101    0.7039    0.7058      4494
weighted avg     0.9416    0.9530    0.9465      4494


Training with: lr = 0.0001, epochs = 150
Epoch 1/150 - Train Loss: 1.3744, Val Loss: 1.3622
Epoch 5/150 - Train Loss: 1.1133, Val Loss: 1.0969
Epoch 10/150 - Train Loss: 0.9927, Val Loss: 0.9866
Epoch 15/150 - Train Loss: 0.9007, Val Loss: 0.8968
Epoch 20/150 - Train Loss: 0.8197, Val Loss: 0.8171
Epoch 25/150 - Train Loss: 0.7475, Val Loss: 0.7462
Epoch 30/150 - Train Loss: 0.6840, Val Loss: 0.6843
Epoch 35/150 - Train Loss: 0.6290, Val Loss: 0.6303
Epoch 40/150 - Train Loss: 0.5818, Val Loss: 0.5844
Epoch 45/150 - Train Loss: 0.5418, Val Loss: 0.5458
Epoch 50/150 - Train Loss: 0.5083, Val Loss: 0.5135
Epoch 55/150 - Train Loss: 0.4815, Val Loss: 0.4875
Epoch 60/150 - Train Loss: 0.4597, Val Loss: 0.4668
Epoch 65/150 - Train Loss: 0.4423, Val Loss: 0.4503
Epoch 70/150 - Train Loss: 0.4293, Val Loss: 0.4379
Epoch 75/150 - Train Loss: 0.4191, Val Loss: 0.4283
Epoch 80/150 - Train Loss: 0.4113, Val Loss: 0.4214
```

```
Epoch 85/150 - Train Loss: 0.4052, Val Loss: 0.4165
Epoch 90/150 - Train Loss: 0.4015, Val Loss: 0.4130
Epoch 95/150 - Train Loss: 0.3984, Val Loss: 0.4104
Epoch 100/150 - Train Loss: 0.3967, Val Loss: 0.4089
Epoch 105/150 - Train Loss: 0.3952, Val Loss: 0.4075
Epoch 110/150 - Train Loss: 0.3945, Val Loss: 0.4068
Epoch 115/150 - Train Loss: 0.3939, Val Loss: 0.4064
Epoch 120/150 - Train Loss: 0.3932, Val Loss: 0.4060
Epoch 125/150 - Train Loss: 0.3933, Val Loss: 0.4059
Epoch 130/150 - Train Loss: 0.3923, Val Loss: 0.4057
Epoch 135/150 - Train Loss: 0.3925, Val Loss: 0.4058
Epoch 140/150 - Train Loss: 0.3921, Val Loss: 0.4059
Epoch 145/150 - Train Loss: 0.3925, Val Loss: 0.4057
Early stopping at epoch 148 (best val loss: 0.405611)
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.0001_150 made no predictions for
classes: [1, 3]

Validation report for optimizer lr_0.0001_epochs_150:
              precision    recall  f1-score   support

           0     0.8808    1.0000    0.9366      3378
           1     0.0000    0.0000    0.0000       285
           2     1.0000    0.8514    0.9197       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.8983      4494
   macro avg     0.4702    0.4629    0.4641      4494
weighted avg     0.8343    0.8983    0.8625      4494


Training with: lr = 0.0001, epochs = 200
Epoch 1/200 - Train Loss: 1.3769, Val Loss: 1.3613
Epoch 5/200 - Train Loss: 1.1986, Val Loss: 1.1379
Epoch 10/200 - Train Loss: 0.9002, Val Loss: 0.8978
Epoch 15/200 - Train Loss: 0.8071, Val Loss: 0.8069
Epoch 20/200 - Train Loss: 0.5318, Val Loss: 0.4675
Epoch 25/200 - Train Loss: 0.3034, Val Loss: 0.3113
Epoch 30/200 - Train Loss: 0.2693, Val Loss: 0.2839
Epoch 35/200 - Train Loss: 0.2512, Val Loss: 0.2674
Epoch 40/200 - Train Loss: 0.2379, Val Loss: 0.2566
Epoch 45/200 - Train Loss: 0.2273, Val Loss: 0.2477
Epoch 50/200 - Train Loss: 0.2184, Val Loss: 0.2408
Epoch 55/200 - Train Loss: 0.2123, Val Loss: 0.2367
Epoch 60/200 - Train Loss: 0.2053, Val Loss: 0.2299
Epoch 65/200 - Train Loss: 0.1999, Val Loss: 0.2257
Epoch 70/200 - Train Loss: 0.1935, Val Loss: 0.2211
Epoch 75/200 - Train Loss: 0.1875, Val Loss: 0.2151
Epoch 80/200 - Train Loss: 0.1828, Val Loss: 0.2106
```

```
Epoch 85/200 - Train Loss: 0.1798, Val Loss: 0.2089
Epoch 90/200 - Train Loss: 0.1774, Val Loss: 0.2065
Epoch 95/200 - Train Loss: 0.1736, Val Loss: 0.2037
Epoch 100/200 - Train Loss: 0.1701, Val Loss: 0.2018
Epoch 105/200 - Train Loss: 0.1676, Val Loss: 0.1993
Epoch 110/200 - Train Loss: 0.1686, Val Loss: 0.1998
Epoch 115/200 - Train Loss: 0.1667, Val Loss: 0.1983
Epoch 120/200 - Train Loss: 0.1650, Val Loss: 0.1970
Epoch 125/200 - Train Loss: 0.1640, Val Loss: 0.1971
Epoch 130/200 - Train Loss: 0.1627, Val Loss: 0.1948
Epoch 135/200 - Train Loss: 0.1618, Val Loss: 0.1937
Epoch 140/200 - Train Loss: 0.1606, Val Loss: 0.1931
Epoch 145/200 - Train Loss: 0.1597, Val Loss: 0.1910
Epoch 150/200 - Train Loss: 0.1586, Val Loss: 0.1957
Epoch 155/200 - Train Loss: 0.1580, Val Loss: 0.1911
Epoch 160/200 - Train Loss: 0.1574, Val Loss: 0.1896
Epoch 165/200 - Train Loss: 0.1575, Val Loss: 0.1886
Epoch 170/200 - Train Loss: 0.1552, Val Loss: 0.1871
Epoch 175/200 - Train Loss: 0.1540, Val Loss: 0.1867
Epoch 180/200 - Train Loss: 0.1523, Val Loss: 0.1855
Epoch 185/200 - Train Loss: 0.1509, Val Loss: 0.1848
Epoch 190/200 - Train Loss: 0.1501, Val Loss: 0.1845
Epoch 195/200 - Train Loss: 0.1495, Val Loss: 0.1872
Epoch 200/200 - Train Loss: 0.1494, Val Loss: 0.1827
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.0001_200 made no predictions for
classes: [3]

Validation report for optimizer lr_0.0001_epochs_200:
              precision    recall  f1-score   support

           0     0.9422    0.9805    0.9610      3378
           1     0.7973    0.8281    0.8124       285
           2     0.9898    0.8734    0.9279       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9399      4494
   macro avg     0.6823    0.6705    0.6753      4494
weighted avg     0.9293    0.9399    0.9337      4494


Training with: lr = 0.005, epochs = 100
Epoch 1/100 - Train Loss: 0.4945, Val Loss: 0.2761
Epoch 5/100 - Train Loss: 0.1516, Val Loss: 0.1746
Epoch 10/100 - Train Loss: 0.1449, Val Loss: 0.1589
Epoch 15/100 - Train Loss: 0.1279, Val Loss: 0.1336
Epoch 20/100 - Train Loss: 0.1205, Val Loss: 0.1238
Epoch 25/100 - Train Loss: 0.1222, Val Loss: 0.1280
Epoch 30/100 - Train Loss: 0.1181, Val Loss: 0.1371
```

```
Epoch 35/100 - Train Loss: 0.1137, Val Loss: 0.1135
Epoch 40/100 - Train Loss: 0.1266, Val Loss: 0.1215
Epoch 45/100 - Train Loss: 0.1192, Val Loss: 0.1420
Epoch 50/100 - Train Loss: 0.1089, Val Loss: 0.1266
Early stopping at epoch 55 (best val loss: 0.113513)
Warning: deep_L5_widths_32_32_16_8_4_AdamW_0.005_100 made no predictions for
classes: [3]

Validation report for optimizer lr_0.005_epochs_100:
              precision    recall  f1-score   support

           0     0.9572    0.9858    0.9713      3378
           1     0.8799    0.9509    0.9140       285
           2     0.9830    0.8979    0.9386       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9559      4494
   macro avg     0.7050    0.7087    0.7060      4494
weighted avg     0.9446    0.9559    0.9497      4494


Training with: lr = 0.005, epochs = 150
Epoch 1/150 - Train Loss: 0.4697, Val Loss: 0.2820
Epoch 5/150 - Train Loss: 0.1766, Val Loss: 0.1966
Epoch 10/150 - Train Loss: 0.1599, Val Loss: 0.1808
Epoch 15/150 - Train Loss: 0.1549, Val Loss: 0.1663
Epoch 20/150 - Train Loss: 0.1528, Val Loss: 0.1688
Epoch 25/150 - Train Loss: 0.1472, Val Loss: 0.1661
Epoch 30/150 - Train Loss: 0.1430, Val Loss: 0.1602
Epoch 35/150 - Train Loss: 0.1380, Val Loss: 0.1639
Epoch 40/150 - Train Loss: 0.1441, Val Loss: 0.1535
Epoch 45/150 - Train Loss: 0.1398, Val Loss: 0.1576
Epoch 50/150 - Train Loss: 0.1392, Val Loss: 0.2838
Epoch 55/150 - Train Loss: 0.1382, Val Loss: 0.1553
Epoch 60/150 - Train Loss: 0.1392, Val Loss: 0.1437
Epoch 65/150 - Train Loss: 0.1359, Val Loss: 0.1536
Epoch 70/150 - Train Loss: 0.1371, Val Loss: 0.1484
Epoch 75/150 - Train Loss: 0.1301, Val Loss: 0.1439
Epoch 80/150 - Train Loss: 0.1264, Val Loss: 0.1636
Epoch 85/150 - Train Loss: 0.1290, Val Loss: 0.1598
Epoch 90/150 - Train Loss: 0.1287, Val Loss: 0.1378
Epoch 95/150 - Train Loss: 0.1331, Val Loss: 0.1515
Epoch 100/150 - Train Loss: 0.1254, Val Loss: 0.1480
Early stopping at epoch 103 (best val loss: 0.135626)

Validation report for optimizer lr_0.005_epochs_150:
              precision    recall  f1-score   support
```

```
           0     0.9506    0.9864    0.9682      3378
           1     0.9247    0.9474    0.9359       285
           2     0.9985    0.8514    0.9191       774
           3     0.3514    0.2281    0.2766        57

    accuracy                         0.9510      4494
   macro avg     0.8063    0.7533    0.7749      4494
weighted avg     0.9496    0.9510    0.9489      4494


Training with: lr = 0.005, epochs = 200
Epoch 1/200 - Train Loss: 0.4412, Val Loss: 0.2837
Epoch 5/200 - Train Loss: 0.1862, Val Loss: 0.2007
Epoch 10/200 - Train Loss: 0.1598, Val Loss: 0.1717
Epoch 15/200 - Train Loss: 0.1406, Val Loss: 0.1455
Epoch 20/200 - Train Loss: 0.1329, Val Loss: 0.1495
Epoch 25/200 - Train Loss: 0.1258, Val Loss: 0.1322
Epoch 30/200 - Train Loss: 0.1194, Val Loss: 0.1463
Epoch 35/200 - Train Loss: 0.1180, Val Loss: 0.1220
Epoch 40/200 - Train Loss: 0.1215, Val Loss: 0.1406
Epoch 45/200 - Train Loss: 0.1151, Val Loss: 0.1390
Epoch 50/200 - Train Loss: 0.1181, Val Loss: 0.1251
Early stopping at epoch 53 (best val loss: 0.118734)

Validation report for optimizer lr_0.005_epochs_200:
              precision    recall  f1-score   support

           0     0.9658    0.9870    0.9763      3378
           1     0.9343    0.9474    0.9408       285
           2     0.9884    0.8811    0.9317       774
           3     0.5397    0.5965    0.5667        57

    accuracy                         0.9613      4494
   macro avg     0.8570    0.8530    0.8539      4494
weighted avg     0.9623    0.9613    0.9612      4494
```

Q: Now, focus on the architecture with the best optimizer. Evaluate the effects of the different learning rates and epochs. Report the test results for the best model.

[112]:
```
best_lr_epochs_model_tag = 'lr_0.005_epochs_200'
model = trained_lr_epochs_models[best_lr_epochs_model_tag]

print(f"\nTest set classification report ({best_lr_epochs_model_tag}):")
report = evaluate_model(model, X_test_tensor_no_port, y_test_no_port)
print(report)
```

```
Test set classification report (lr_0.005_epochs_200):
```

```
              precision    recall  f1-score   support

           0     0.9664    0.9873    0.9767      3378
           1     0.9281    0.9476    0.9377       286
           2     0.9927    0.8797    0.9328       773
           3     0.5000    0.5789    0.5366        57

    accuracy                         0.9611      4494
   macro avg     0.8468    0.8484    0.8460      4494
weighted avg     0.9626    0.9611    0.9611      4494
```

1.7 Task 6 — Overfitting and Regularization

We analyze overfitting and apply regularization techniques to improve generalization.

Base model: - Layers: **6** - Hidden widths: [256, 128, 64, 32, 16] - Activation: **ReLU** - Optimizer: **AdamW**, learning rate = 5e-4 - Batch size: **128** - Epochs: up to 50

Steps: - Train baseline model and inspect loss curves for overfitting. - Add **Dropout**, **Batch Normalization**, and **Weight Decay** (AdamW regularization). - Compare validation/test performance across variants. - Discuss which regularization technique best mitigates overfitting.

[113]:
```
# Create directory for plots
save_dir = results_path + 'images/' + 'task6_plots/'
os.makedirs(save_dir, exist_ok=True)
```

1.7.1 Training

[114]:
```
# --- Define 6-layer FFNN with Regularization Options ---

class RegularizedDeepFFNN(nn.Module):
    def __init__(self, input_dim, layer_widths, output_dim, activation='relu',␣
 ↪dropout_prob=0.0, use_batchnorm=False):
        super(RegularizedDeepFFNN, self).__init__()
        layers = []
        prev_width = input_dim
        for i, width in enumerate(layer_widths):
            layers.append(nn.Linear(prev_width, width))
            if use_batchnorm:
                layers.append(nn.BatchNorm1d(width))
            if activation == 'relu':
                layers.append(nn.ReLU())
            if dropout_prob > 0 and i < len(layer_widths) - 1: # Apply dropout␣
 ↪to hidden layers
                layers.append(nn.Dropout(dropout_prob))
            prev_width = width
        layers.append(nn.Linear(prev_width, output_dim))
        self.net = nn.Sequential(*layers)
```

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

[115]:
```
# --- Experiment with Regularization ---

input_dim_reg = X_train_std_no_port.shape[1]
output_dim_reg = len(np.unique(y_train_no_port))
layer_widths_reg = [256, 128, 64, 32, 16] # Same widths as the baseline 6-layer␣
 ↪model

# Define hyperparameters for regularization experiments
base_lr = 0.0005
epochs_reg = 50
batch_size_reg = 128
min_delta_reg = 0.00001
patience_reg = 20

regularization_configs = {
    'Baseline': # No dropout, no batch norm, no weight decay
        {'dropout_prob': 0.0, 'use_batchnorm': False, 'weight_decay': 0.0},
    'Dropout_0.5': # Dropout 50% (no BN, no weight decay)
        {'dropout_prob': 0.5, 'use_batchnorm': False, 'weight_decay': 0.0},
    'BatchNorm': # Batch normalization only
        {'use_batchnorm': True, 'dropout_prob': 0.0, 'weight_decay': 0.0},
    'BatchNorm_Dropout_0.5': # BatchNorm + Dropout 50%
        {'use_batchnorm': True, 'dropout_prob': 0.5, 'weight_decay': 0.0},
    'WeightDecay_1e-4': # L2 regularization via weight_decay=1e-4 (no dropout,␣
 ↪no BN)
        {'dropout_prob': 0.0, 'use_batchnorm': False, 'weight_decay': 1e-4},
    'WeightDecay_1e-4_BN_Dropout_0.5': # All three: BatchNorm + Dropout 50% +␣
 ↪Weight Decay 1e-4
        {'use_batchnorm': True, 'dropout_prob': 0.5, 'weight_decay': 1e-4}
}

regularized_models = {}
regularized_loss_curves = {}
regularized_validation_reports = {}
regularized_test_reports = {}

for config_name, params in regularization_configs.items():
    print(f"\nTraining model with {config_name}...")

    # Instantiate model with regularization
    model_reg = RegularizedDeepFFNN(
        input_dim_reg,
        layer_widths_reg,
```

```
        output_dim_reg,
        activation='relu',
        dropout_prob=params.get('dropout_prob', 0.0),
        use_batchnorm=params.get('use_batchnorm', False)
    )

    # Define optimizer with weight decay
    optimizer_reg = optim.AdamW(model_reg.parameters(), lr=base_lr,␣
↪weight_decay=params.get('weight_decay', 0.0))
    criterion = nn.CrossEntropyLoss()

    # Create DataLoaders (using the same as the baseline 6-layer model)
    train_loader_reg = train_loader_no_port
    val_loader_reg = val_loader_no_port

    # Move model to device
    model_reg = model_reg.to(device)

    # Training
    model_reg, train_loss_reg, val_loss_reg = train_model(
        model_reg,
        train_loader_reg,
        val_loader_reg,
        epochs_reg,
        optimizer_reg,
        criterion,
        min_delta=min_delta_reg,
        patience=patience_reg
    )

    regularized_models[config_name] = model_reg
    regularized_loss_curves[config_name] = (train_loss_reg, val_loss_reg)

    model_name = f"deep_L3_widths_{'_'.join(map(str,␣
↪layer_widths_reg))}_reg_{config_name}"

    # Evaluate on validation set
    report_val_reg = evaluate_model(model_reg, X_val_tensor_no_port,␣
↪y_val_no_port, model_name)
    regularized_validation_reports[config_name] = report_val_reg
    print(f"\nValidation report for {config_name}:")
    print(report_val_reg)

    # Evaluate on test set
    report_test_reg = evaluate_model(model_reg, X_test_tensor_no_port,␣
↪y_test_no_port)
    regularized_test_reports[config_name] = report_test_reg
```

```
    print(f"\nTest report for {config_name}:")
    print(report_test_reg)

# It is now possible compare the reports in regularized_validation_reports and␣
↪regularized_test_reports
```

```
Training model with Baseline…
Epoch 1/50 - Train Loss: 0.5558, Val Loss: 0.3166
Epoch 5/50 - Train Loss: 0.1646, Val Loss: 0.1904
Epoch 10/50 - Train Loss: 0.1322, Val Loss: 0.1489
Epoch 15/50 - Train Loss: 0.1262, Val Loss: 0.1447
Epoch 20/50 - Train Loss: 0.1172, Val Loss: 0.1387
Epoch 25/50 - Train Loss: 0.1083, Val Loss: 0.1213
Epoch 30/50 - Train Loss: 0.1034, Val Loss: 0.1297
Epoch 35/50 - Train Loss: 0.1006, Val Loss: 0.1168
Epoch 40/50 - Train Loss: 0.1018, Val Loss: 0.1274
Epoch 45/50 - Train Loss: 0.1023, Val Loss: 0.1293
Epoch 50/50 - Train Loss: 0.1022, Val Loss: 0.1188

Validation report for Baseline:
              precision    recall  f1-score   support

           0     0.9669    0.9843    0.9755      3378
           1     0.9310    0.9474    0.9391       285
           2     0.9789    0.8992    0.9374       774
           3     0.6296    0.5965    0.6126        57

    accuracy                         0.9624      4494
   macro avg     0.8766    0.8568    0.8662      4494
weighted avg     0.9624    0.9624    0.9620      4494


Test report for Baseline:
              precision    recall  f1-score   support

           0     0.9667    0.9876    0.9770      3378
           1     0.9249    0.9476    0.9361       286
           2     0.9817    0.9017    0.9400       773
           3     0.7750    0.5439    0.6392        57

    accuracy                         0.9646      4494
   macro avg     0.9121    0.8452    0.8731      4494
weighted avg     0.9642    0.9646    0.9638      4494


Training model with Dropout_0.5…
Epoch 1/50 - Train Loss: 0.6715, Val Loss: 0.3437
```

```
Epoch 5/50 - Train Loss: 0.2813, Val Loss: 0.2687
Epoch 10/50 - Train Loss: 0.2255, Val Loss: 0.2099
Epoch 15/50 - Train Loss: 0.1859, Val Loss: 0.1757
Epoch 20/50 - Train Loss: 0.1659, Val Loss: 0.1581
Epoch 25/50 - Train Loss: 0.1606, Val Loss: 0.1517
Epoch 30/50 - Train Loss: 0.1526, Val Loss: 0.1444
Epoch 35/50 - Train Loss: 0.1478, Val Loss: 0.1402
Epoch 40/50 - Train Loss: 0.1499, Val Loss: 0.1389
Epoch 45/50 - Train Loss: 0.1496, Val Loss: 0.1419
Epoch 50/50 - Train Loss: 0.1462, Val Loss: 0.1339
Warning: deep_L3_widths_256_128_64_32_16_reg_Dropout_0.5 made no predictions for
classes: [3]

Validation report for Dropout_0.5:
              precision    recall  f1-score   support

           0     0.9474    0.9802    0.9635      3378
           1     0.8108    0.9474    0.8738       285
           2     0.9925    0.8540    0.9181       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9439      4494
   macro avg     0.6877    0.6954    0.6888      4494
weighted avg     0.9345    0.9439    0.9377      4494

Warning: Unnamed model made no predictions for classes: [3]

Test report for Dropout_0.5:
              precision    recall  f1-score   support

           0     0.9440    0.9772    0.9603      3378
           1     0.7890    0.9545    0.8639       286
           2     0.9939    0.8370    0.9087       773
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9393      4494
   macro avg     0.6817    0.6922    0.6832      4494
weighted avg     0.9307    0.9393    0.9331      4494


Training model with BatchNorm…
Epoch 1/50 - Train Loss: 0.5386, Val Loss: 0.3429
Epoch 5/50 - Train Loss: 0.1481, Val Loss: 0.1696
Epoch 10/50 - Train Loss: 0.1550, Val Loss: 0.1514
Epoch 15/50 - Train Loss: 0.1409, Val Loss: 0.1518
Epoch 20/50 - Train Loss: 0.1426, Val Loss: 0.1577
Epoch 25/50 - Train Loss: 0.1333, Val Loss: 0.1451
Epoch 30/50 - Train Loss: 0.1336, Val Loss: 0.1594
```

```
Epoch 35/50 - Train Loss: 0.1299, Val Loss: 0.1726
Epoch 40/50 - Train Loss: 0.1284, Val Loss: 0.1735
Epoch 45/50 - Train Loss: 0.1320, Val Loss: 0.1424
Epoch 50/50 - Train Loss: 0.1269, Val Loss: 0.2123

Validation report for BatchNorm:
              precision    recall  f1-score   support

           0     0.9585    0.9855    0.9718      3378
           1     0.9278    0.9474    0.9375       285
           2     0.9926    0.8721    0.9285       774
           3     0.3600    0.3158    0.3364        57

    accuracy                         0.9551      4494
   macro avg     0.8098    0.7802    0.7936      4494
weighted avg     0.9549    0.9551    0.9541      4494


Test report for BatchNorm:
              precision    recall  f1-score   support

           0     0.9600    0.9870    0.9733      3378
           1     0.9128    0.9510    0.9315       286
           2     0.9941    0.8719    0.9290       773
           3     0.4889    0.3860    0.4314        57

    accuracy                         0.9573      4494
   macro avg     0.8389    0.7990    0.8163      4494
weighted avg     0.9569    0.9573    0.9561      4494


Training model with BatchNorm_Dropout_0.5…
Epoch 1/50 - Train Loss: 0.8901, Val Loss: 0.5737
Epoch 5/50 - Train Loss: 0.2895, Val Loss: 0.2536
Epoch 10/50 - Train Loss: 0.2453, Val Loss: 0.2137
Epoch 15/50 - Train Loss: 0.2246, Val Loss: 0.1937
Epoch 20/50 - Train Loss: 0.2197, Val Loss: 0.1932
Epoch 25/50 - Train Loss: 0.2100, Val Loss: 0.1848
Epoch 30/50 - Train Loss: 0.2054, Val Loss: 0.1796
Epoch 35/50 - Train Loss: 0.1948, Val Loss: 0.1747
Epoch 40/50 - Train Loss: 0.1940, Val Loss: 0.1717
Epoch 45/50 - Train Loss: 0.1935, Val Loss: 0.1707
Epoch 50/50 - Train Loss: 0.1922, Val Loss: 0.1737
Warning: deep_L3_widths_256_128_64_32_16_reg_BatchNorm_Dropout_0.5 made no
predictions for classes: [3]

Validation report for BatchNorm_Dropout_0.5:
              precision    recall  f1-score   support
```

```
           0     0.9464    0.9725    0.9593      3378
           1     0.7459    0.9474    0.8346       285
           2     0.9970    0.8514    0.9185       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9377      4494
   macro avg     0.6723    0.6928    0.6781      4494
weighted avg     0.9304    0.9377    0.9322      4494

Warning: Unnamed model made no predictions for classes: [3]

Test report for BatchNorm_Dropout_0.5:
              precision    recall  f1-score   support

           0     0.9431    0.9710    0.9568      3378
           1     0.7378    0.9545    0.8323       286
           2     0.9985    0.8344    0.9091       773
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9341      4494
   macro avg     0.6698    0.6900    0.6746      4494
weighted avg     0.9276    0.9341    0.9286      4494


Training model with WeightDecay_1e-4…
Epoch 1/50 - Train Loss: 0.5557, Val Loss: 0.3046
Epoch 5/50 - Train Loss: 0.1690, Val Loss: 0.1760
Epoch 10/50 - Train Loss: 0.1381, Val Loss: 0.1736
Epoch 15/50 - Train Loss: 0.1263, Val Loss: 0.1565
Epoch 20/50 - Train Loss: 0.1294, Val Loss: 0.1530
Epoch 25/50 - Train Loss: 0.1265, Val Loss: 0.1441
Epoch 30/50 - Train Loss: 0.1212, Val Loss: 0.1385
Epoch 35/50 - Train Loss: 0.1111, Val Loss: 0.1243
Epoch 40/50 - Train Loss: 0.1143, Val Loss: 0.1333
Epoch 45/50 - Train Loss: 0.1119, Val Loss: 0.1220
Epoch 50/50 - Train Loss: 0.1073, Val Loss: 0.1305

Validation report for WeightDecay_1e-4:
              precision    recall  f1-score   support

           0     0.9695    0.9787    0.9741      3378
           1     0.9340    0.9439    0.9389       285
           2     0.9588    0.9031    0.9301       774
           3     0.3881    0.4561    0.4194        57

    accuracy                         0.9568      4494
   macro avg     0.8126    0.8204    0.8156      4494
```

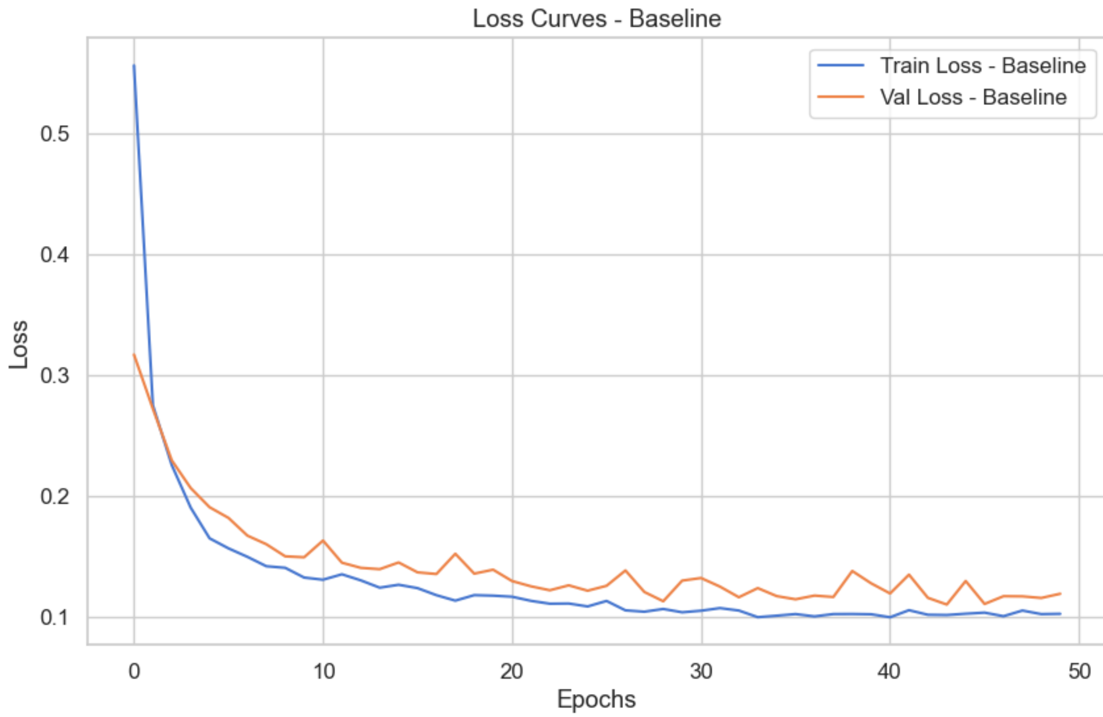
```
weighted avg     0.9580    0.9568    0.9572      4494


Test report for WeightDecay_1e-4:
              precision    recall  f1-score   support

           0     0.9707    0.9796    0.9751      3378
           1     0.9218    0.9476    0.9345       286
           2     0.9652    0.8978    0.9303       773
           3     0.4722    0.5965    0.5271        57

    accuracy                         0.9586      4494
   macro avg     0.8325    0.8554    0.8418      4494
weighted avg     0.9603    0.9586    0.9591      4494


Training model with WeightDecay_1e-4_BN_Dropout_0.5…
Epoch 1/50 - Train Loss: 0.9748, Val Loss: 0.6319
Epoch 5/50 - Train Loss: 0.3131, Val Loss: 0.2765
Epoch 10/50 - Train Loss: 0.2544, Val Loss: 0.2077
Epoch 15/50 - Train Loss: 0.2302, Val Loss: 0.1894
Epoch 20/50 - Train Loss: 0.2205, Val Loss: 0.1799
Epoch 25/50 - Train Loss: 0.2161, Val Loss: 0.1779
Epoch 30/50 - Train Loss: 0.2024, Val Loss: 0.1749
Epoch 35/50 - Train Loss: 0.1964, Val Loss: 0.1680
Epoch 40/50 - Train Loss: 0.1988, Val Loss: 0.1711
Epoch 45/50 - Train Loss: 0.1939, Val Loss: 0.1716
Epoch 50/50 - Train Loss: 0.1987, Val Loss: 0.1708
Warning: deep_L3_widths_256_128_64_32_16_reg_WeightDecay_1e-4_BN_Dropout_0.5
made no predictions for classes: [3]

Validation report for WeightDecay_1e-4_BN_Dropout_0.5:
              precision    recall  f1-score   support

           0     0.9455    0.9766    0.9608      3378
           1     0.7733    0.9333    0.8458       285
           2     0.9970    0.8514    0.9185       774
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9399      4494
   macro avg     0.6789    0.6903    0.6813      4494
weighted avg     0.9315    0.9399    0.9340      4494

Warning: Unnamed model made no predictions for classes: [3]

Test report for WeightDecay_1e-4_BN_Dropout_0.5:
              precision    recall  f1-score   support
```

```
           0     0.9434    0.9725    0.9577      3378
           1     0.7479    0.9545    0.8387       286
           2     0.9985    0.8357    0.9099       773
           3     0.0000    0.0000    0.0000        57

    accuracy                         0.9355      4494
   macro avg     0.6725    0.6907    0.6766      4494
weighted avg     0.9285    0.9355    0.9298      4494
```

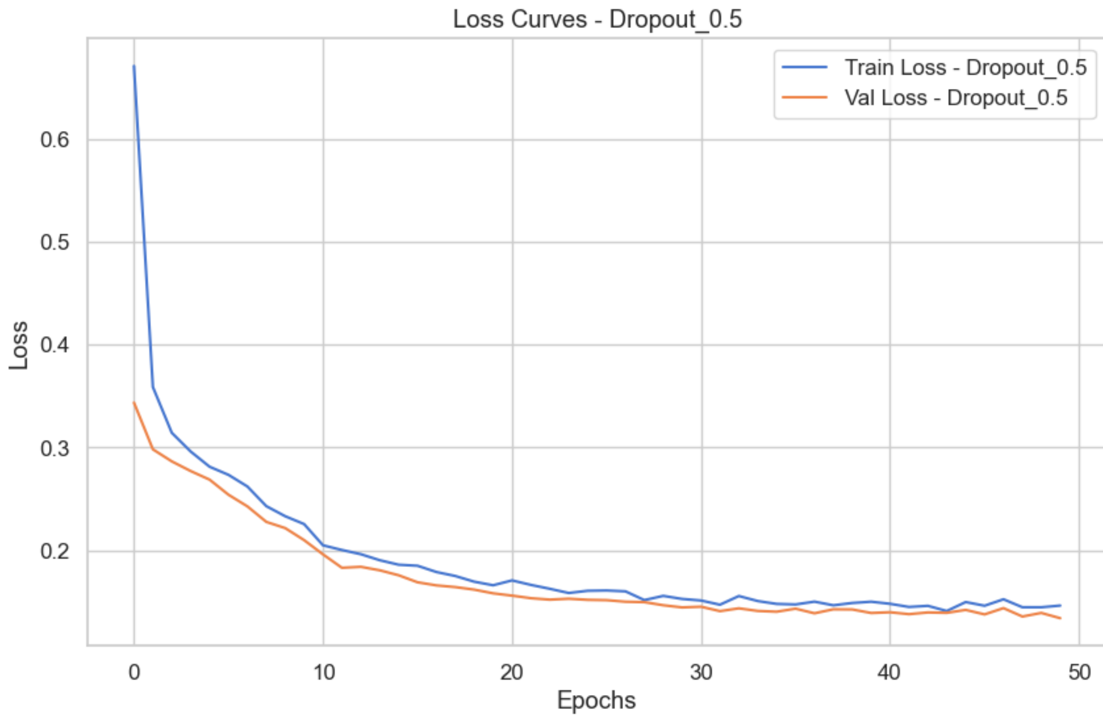
1.7.2 Evaluating

[116]:
```python
# --- Plot loss curves for all regularized models ---

for config_name, (train_loss, val_loss) in regularized_loss_curves.items():
    plt.figure(figsize=(10,6))
    plt.plot(train_loss, label=f'Train Loss - {config_name}')
    plt.plot(val_loss, label=f'Val Loss - {config_name}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Loss Curves - {config_name}')
    plt.legend()

    # Save the plot to the specified path
    save_plot(plt.gcf(), f"{config_name}_loss_curve", save_dir) # Use plt.gcf()␣
 ↪to get the current figure

    plt.show()
```

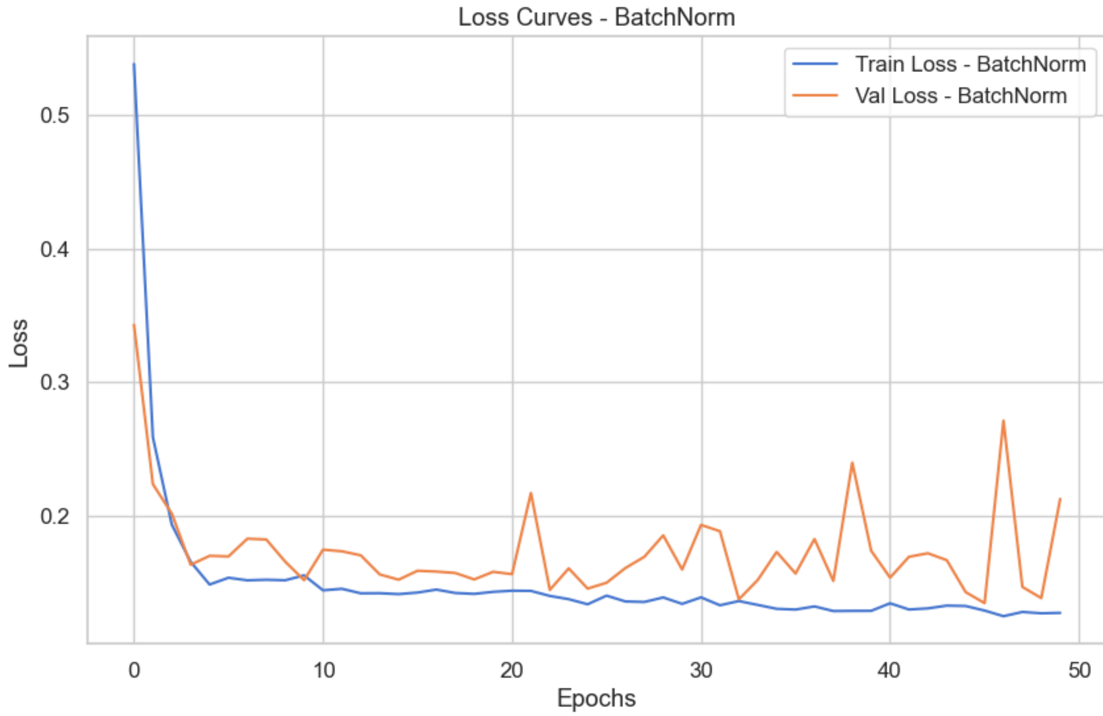
```
Saved plot: ../results/images/task6_plots/Baseline_loss_curve.png
```



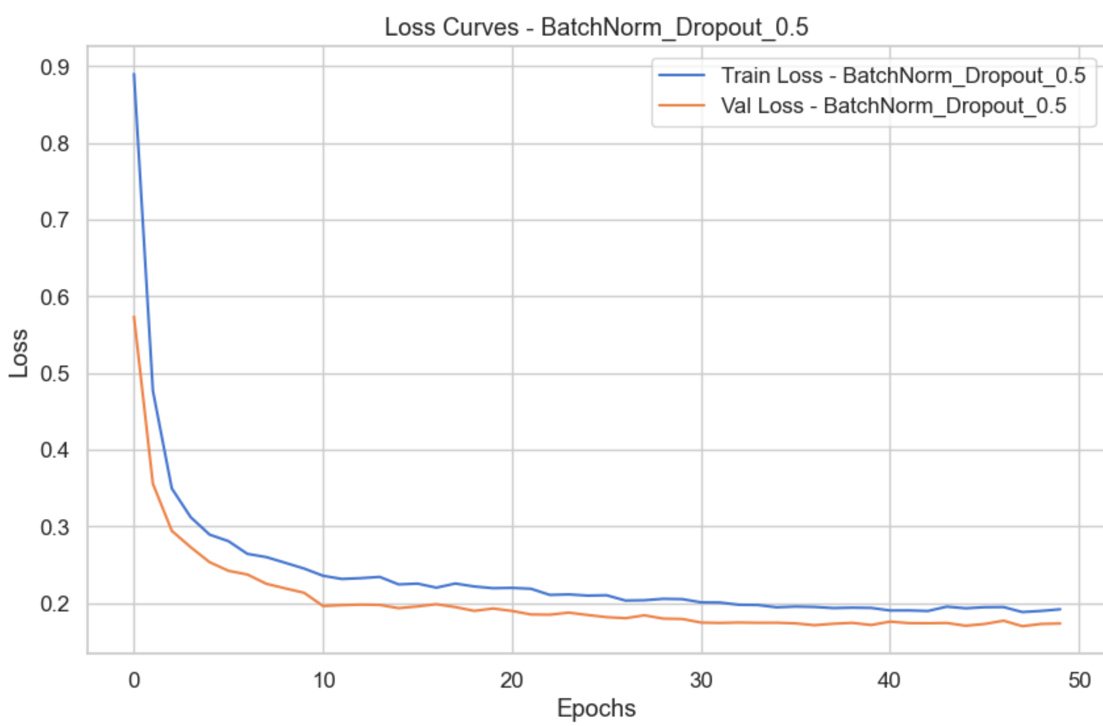
Saved plot: ../results/images/task6_plots/Dropout_0.5_loss_curve.png



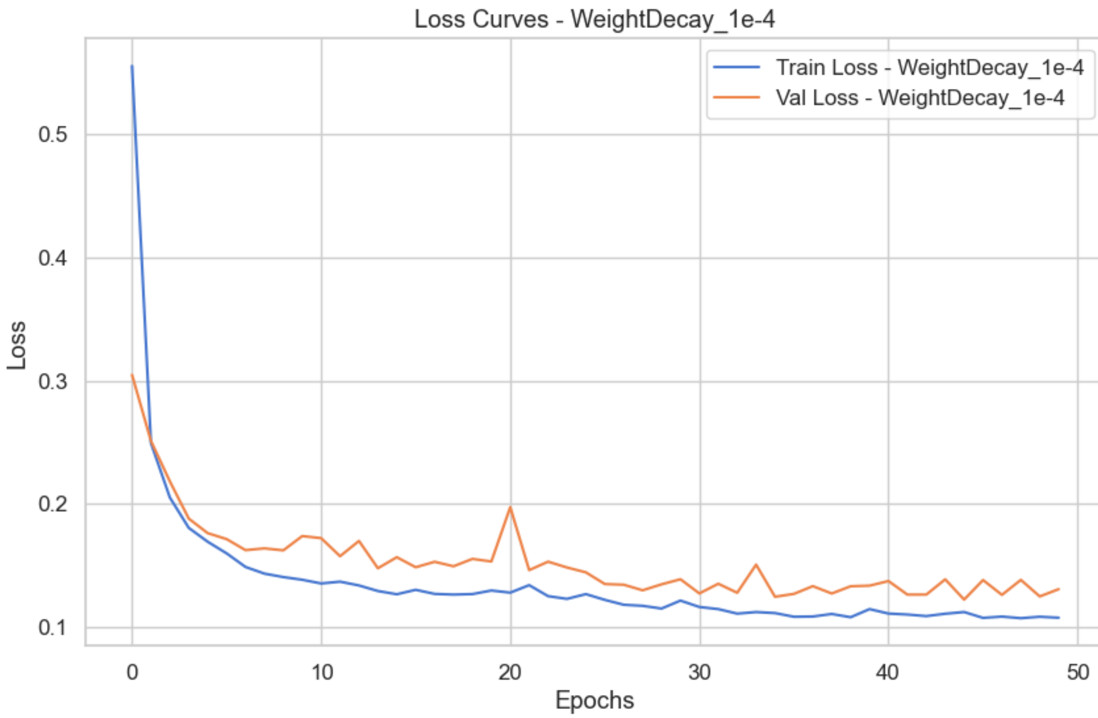
```
Saved plot: ../results/images/task6_plots/BatchNorm_loss_curve.png
```



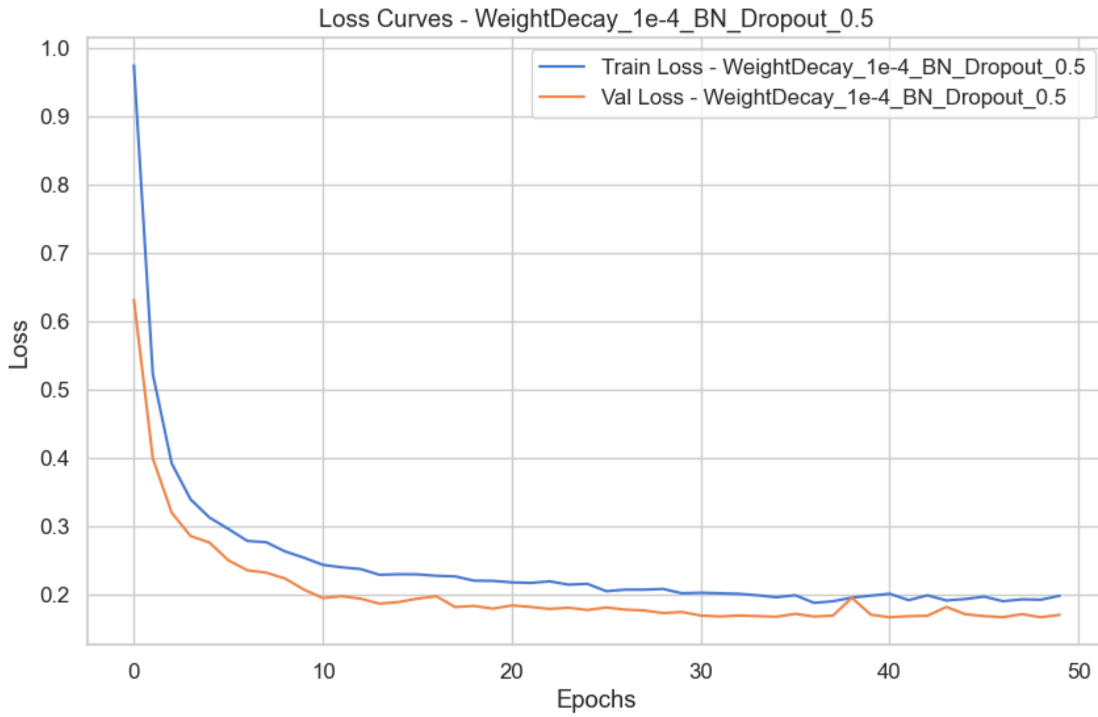

```
Saved plot: ../results/images/task6_plots/BatchNorm_Dropout_0.5_loss_curve.png
```



Saved plot: ../results/images/task6_plots/WeightDecay_1e-4_loss_curve.png



Saved plot:
../results/images/task6_plots/WeightDecay_1e-4_BN_Dropout_0.5_loss_curve.png



Q: Each Neurons per Layer describes how many neurons must be present in each layer. For example: 1st layer 256, 2nd layer 128 etc. What do the losses look like? Is the model overfitting? The training and validation losses both decrease smoothly and stabilize around 0.10-0.12, showing consistent convergence. The validation loss stays slightly higher than the training loss, which is expected — this indicates good generalization rather than overfitting.

If the model were overfitting, we would see the training loss continue to drop while the validation loss increased or fluctuated strongly. Here, both curves follow the same trend and plateau together.

The losses show a healthy training process — the model converges well and does not overfit, maintaining high validation accuracy (~96%).

Q: Now apply normalization techniques (dropout, batch normalization) and play with the regularization of the weights (AdamW's weight decay). What impact do the different normalization techniques have on validation and testing performance? Impact summary:

- **Baseline (AdamW)** - Best overall (Val 96%, Test 96%). Smooth convergence, no overfitting, and correct predictions for all classes.
- **Dropout (0.5)** - Validation loss a bit lower than training; minority class (3) never predicted.
- **BatchNorm** - Validation loss unstable, signs of overfitting/instability; poor minority class recall.

- **BatchNorm + Dropout (0.5)** - Too much regularization → strong underfitting; class 3 ignored.
- **Weight Decay (1e-4)** - Slightly improves generalization and keeps losses stable; close to baseline.
- **Weight Decay + BN + Dropout (0.5)** - Over-regularized; underfits, poor validation.

Best setup: *AdamW + small weight decay (1e-4)*.

Too strong: *Dropout / BatchNorm* on this tabular dataset → underfitting, unstable validation.
