MIDDLE EAST TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS ENGINEERING DEPARTMENT

EE435 TERM PROJECT REPORT – PART 2

Signal Classification with Deep Learning

Student Names: Oğuzhan Oğuz – Arda Denizci

Student IDs: 2516631 - 2515963

1. Introduction

In modern communication systems, accurate classification of modulation types is very important. This facilitates efficient signal processing, spectrum monitoring, and interference detection. Traditional classification methods are often based on manual feature extraction and signal processing techniques. These techniques may be limited in handling complex and dynamic signal environments. With the rapid developments in deep learning, data-driven approaches have emerged as powerful tools for automating signal classification tasks. They have demonstrated superior performance in identifying various modulation schemes under challenging conditions. The aim of this project is to develop a deep learning-based classification system that can distinguish five different types of analog modulations. The motivation of this work is to explore and use the capabilities of deep learning, especially Residual Neural Networks, in signal classification to classify different types of modulations. Datasets simulating real-world conditions will be generated. Using these datasets, the ResNet model will be trained, and robust classification performance will be achieved. In this way, the effects of SNR and CFO on signal recognition will also be investigated. This report presents all these processes and experimental results in detail.

2. Literature Review

This project builds on the work of O'Shea et al. (2018) titled "Over-the-Air Deep Learning Based Radio Signal Classification" [1]. The work examines the role of deep learning in radio signal classification. CNNs and ResNets are used in this task. Traditional methods rely on manually designed features. These approaches require expertise and cannot adapt to changing conditions. Deep learning models automatically learn features from raw data. This increases the classification accuracy and robustness. It provides effective results even in difficult conditions such as CFO, SRO, and multi-path reflection. These distortions degrade the performance of traditional methods. O'Shea et al. presented a new dataset generation method that includes realistic channel effects. This method covers AWGN, frequency shifts, and multi-path reflection. They tested their model in both simulations and OTA experiments. ResNet architectures exhibited superior performance thanks to skip connections and deep layers. These findings demonstrate the importance of advanced models such as ResNet in analog modulation classification.

3. Signal Model for Generating Analog Signals

Based on the Over-the-Air Deep Learning Based Radio Signal Classification paper and common analog communication practice, the following parameters and assumptions can be used to model and generate synthetic analog modulated signals. While the O'Shea et al. paper frequently uses both digital and analog classes, this report outlines a reasonable setup for analog signals only and includes the type of randomness used in the datasets.

General Setup

- **Complex Baseband Representation:** Signals are often generated at complex baseband, and then up-converted to an intermediate frequency (IF) or carrier frequency for realism. The simulated received signal is then sampled at a certain rate to produce time-domain I/Q samples for the classifier.
- **Message Signal (m(t)):** For analog modulation, the message signal can be a band-limited random waveform. This could be realized as filtered Gaussian noise. For a Gaussian noise, apply a low-pass filter to ensure the message bandwidth is appropriate. For synthetic speech-like signals, random audio clips can be chosen from a dataset. Each generated signal instance should use different noise realizations to ensure variety and robustness in training.

Carrier and Modulation Parameters

- **Carrier Frequency (f_c):** Chosen as a fixed reference. Random frequency offsets around f_c may be introduced to represent realistic conditions.
- **Sampling Frequency (f_s):** High enough to capture both the carrier and modulation bandwidth and avoid aliasing.
- **Amplitude Modulation (DSB and SSB, with and without carrier):** Signal form is:

$$x_{AM-DSB-WC}(t) = [A + k_a m(t)]cos(2\pi f_c t)$$
$$x_{AM-DSB-SC}(t) = m(t)\, cos(2\pi f_c t)$$
$$x_{AM-SSB-WC}(t) = [A + m(t)]\, cos(2\pi f_c t) - m'(t)\, sin(2\pi f_c t)$$
$$x_{AM-SSB-SC}(t) = m(t)\, cos(2\pi f_c t) - m'(t)\, sin(2\pi f_c t)$$

  Carrier amplitude is chosen as a nominal value (e.g., $A = 1$) where $m'(t)$ is the Hilbert transform of the message signal. Some randomness can be introduced in amplitude scaling. The modulation index k_a can be adjusted in the code, and chosen as 0.8 in our applications.

- **Frequency Modulation (FM) :** Signal form is:

$$x_{FM}(t) = cos(2\pi f_c t + 2\pi k_f \int m(\tau) d\tau)$$

  The message signal can be approximated as a zero mean signal, the k_f value determines the frequency deviation. This value can be adjusted in the code, and chosen as 1 in our applications.

- **Incorporating Randomness:** Each generated signal introduces randomness through carrier frequency offset (CFO) and additive white Gaussian noise (AWGN) to simulate realistic transmission conditions. Each generated signal uses a separate segment of random band-limited noise to ensure that no two signals are the same. This randomness simulates different transmitted content. The additional white Gaussian noise (AWGN) is introduced at different SNR levels (e.g. SNR $\in$ [SNR_{min}, SNR_{max}]). This range can be simulated from noisy to less noisy environments. Also, slightly randomly shifted carrier frequency (CFO) is added to simulate oscillator mismatches (Δf_c ~ N(0,σ_{clk})). Reasonable values can be selected to be a few Hz, depending on the system scenario. These two factors are the sole sources of randomness in the signal-generation process. They are ensuring that the model remains both realistic and aligned with project requirements.

Channel and Impairments

- **Additive White Gaussian Noise (AWGN):** Noise is added to achieve a specific Signal-to-Noise Ratio (SNR) chosen from a distribution SNR ~ Uniform(SNR_{min}, SNR_{max}) . This ensures the model sees a range of channel conditions.
- **Frequency Offset (Δf):** A small frequency offset Δf_c ~ $N(0,\sigma_{clk})$ can simulate oscillator mismatch.

Normalization and Scaling

Before adding noise, the generated signals can be normalized to a standard power level. thus providing controlled SNR conditions and consistent input magnitude for training the neural network classifier.
By focusing on analog modulation methods such as AM and FM, and incorporating randomness in message signals, carrier offsets, and channel conditions, one can create a synthetic dataset similar in spirit to the one described by O'Shea et al. Although their work primarily addresses digital modulations, the methodology described (randomization, dataset size, over-the-air testing, and deep neural network architectures) can be directly applied to the analog domain. The signal model parameters and their distributions outlined above provide a reasonable starting point to generate and train deep learning models for robust analog signal classification.

4. Data Generation

Synthetic datasets were created using MATLAB as the simulation environment. The operations performed can be divided into the following steps:

Signal Parameterization: For each modulation type, basic parameters were defined. Message signals were selected from different kinds of audio signals. Variability and a realistic representation of different "messages" were provided.

Bandwidth Variation: To meet the requirement of having two bandwidth values per modulation type, the generation process was run twice for each modulation type, one at a base bandwidth of 4 kHz and one at twice the base bandwidth of 8 kHz. The appropriate bandwidth parameter was used each time.

Incorporating Randomness in Multiple Dimensions: Two other random parameters are introduced to mimic realistic imperfections and diversity in the dataset:

α. Carrier Frequency Offset (Δf_c): The carrier frequency offset is modelled as $N(0,\sigma_{clk})$, where σ_{clk} is selected as 0.1, providing a Gaussian-distributed CFO that reflects oscillator frequency drifts.

β. Additive White Gaussian Noise (AWGN): AWGN is introduced to simulate background noise and channel interference. Noise is added at various SNR levels with SNR$\in$ [-20, 30] dB to expose the model to different noise conditions, enhancing its robustness in noisy environments.

Incorporating SNR Variations: After generating the clean, normalized signal, additional white Gaussian noise (AWGN) is added to obtain a range of Signal-to-Noise Ratios (SNR). A random SNR value between -20 and 30 dB will be used. This range covers very low to moderately high SNR conditions. It allows the model to encounter both nearly corrupted signals and relatively clean signals. The logic behind these values is to test model robustness. Low SNR scenarios force the classifier to identify signals buried in noise. Higher SNR scenarios are more straightforward and help the model learn ideal patterns.

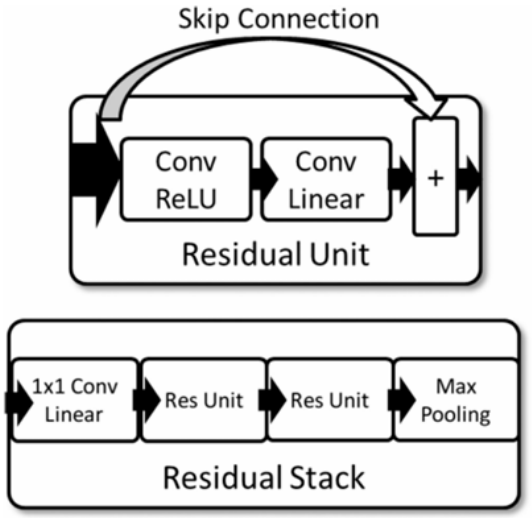
Applying Carrier Frequency Offset (CFO): A random CFO was added to each signal to simulate realistic receiver conditions since the local oscillator frequency might not match the transmitter frequency perfectly. For example, CFO values were drawn from a uniform distribution within N(0,0.1). This allows the classifier to learn to handle frequency mismatches that naturally occur due to hardware imperfections. The selected range represents typical oscillator mismatches in practical communication systems.

Ensuring Diversity and Realism: By introducing randomness in these specifications, the dataset simulates real-world conditions.

Signals were generated in accordance with the above-mentioned items. For each modulation type, message signals with two different bandwidths were modulated, and 500 samples were created for each case. The generated noisy signals were converted to baseband to obtain in-phase and quadrature components with low pass filtering. After this conversion process, 1024 samples starting from a random time index of I and Q components were selected from each signal to maintain realistic conditions. This was preferred because a 1024 sample size was used in the reviewed article. The obtained I/Q samples were grouped according to the modulation type and stored in HDF5 format. Each sample was recorded as two separate data sets (I and Q). Parameters such as CFO, SNR, and bandwidth were labelled for each sample. In this way, large data sets can be stored efficiently. Fast access to individual samples is provided, and data is organized hierarchically. Finally, the labelling of parameters is facilitated.

The MATLAB code created for data generation is added to the appendix section of the report.

ResNet Model



Layer	Output dimensions
Input	2×1024
Residual Stack	32×512
Residual Stack	32×256
Residual Stack	32×128
Residual Stack	32×64
Residual Stack	32×32
Residual Stack	32×16
FC/SeLU	128
FC/SeLU	128
FC/Softmax	24

Figure 1. ResNet model [1]

The architecture of the model is given in Figure 1. This ResNet-based model is designed for efficient feature extraction and classification of sequential data like signals. After obtaining the I-Q components with a size of 2x1024, the model takes them as input, and processes input data through a series of Residual Stacks, each containing 1×1 convolution, two residual units, and max pooling. Residual connections preserve gradient flow and features, while downsampling reduces dimensions and enriches feature representations. The architecture extracts features at various scales, with dimensions transitioning from 32×512 to 32×16.

After feature extraction, fully connected layers with SeLU activation (128 units each) and a softmax output layer classify the data into 5 modulation classes. This structure allows the model to learn complex signal patterns effectively.

The model is selected with various tests over different structures. The model selection criteria and methods are described in the following part.

5. ResNet Training Parameters

There are many parameters in the architecture of Residual Networks that determine the performance of the model. Depending on the depth and complexity of the model, ResNet architecture is defined with different layer numbers, residual block structures and filter numbers in these blocks. In this architecture, kernel sizes affect learning small or large-scale patterns. Pooling layers provide general features. Activation functions allow the model to learn non-linear relationships. Gradients provide effective propagation. In the training process, parameters such as batch size, learning rate, and optimizer are important. Techniques such as data augmentation and dropout are also used to increase generalization performance. The most important parameters include batch size, learning rate, optimizer type and activation functions. Tests were conducted to determine the most appropriate batch size, optimizer and activation function for the model.

Batch size refers to the number of samples the model processes before updating its weights. Training speed has a direct impact on memory requirements and the generalization ability of the model. Small batch sizes require less memory. It can improve generalization due to noise in gradient updates. However, it can cause slower convergence. Large batch sizes provide faster training and produce more stable gradients. However, it requires more memory. Sometimes, it can negatively affect generalization performance. The optimizer (optimization algorithm) determines how to update the weights to minimize the model's loss function. It directly affects how efficiently the model converges during training. SGD offers simple and good generalization but converges more slowly. Adam adapts the learning rate for each parameter. It provides faster convergence. It is widely used in deep architectures such as ResNet. The choice of optimizer directly affects the training stability and overall performance of the model. Activation functions add nonlinearity to the model. In this way, it is possible to learn complex relationships in the data. Without an activation function, the network behaves like a linear model regardless of its depth. The most commonly used activation function for ResNet is ReLu. This function is computationally efficient and reduces the vanishing gradient problem by facilitating the propagation of gradients. [2]

Tests were performed to determine the best parameters for the created ResNet model. First, tests were performed for batch sizes 4, 8, 16, 32, 64, and 128, respectively (using ReLu and Adam optimizer):

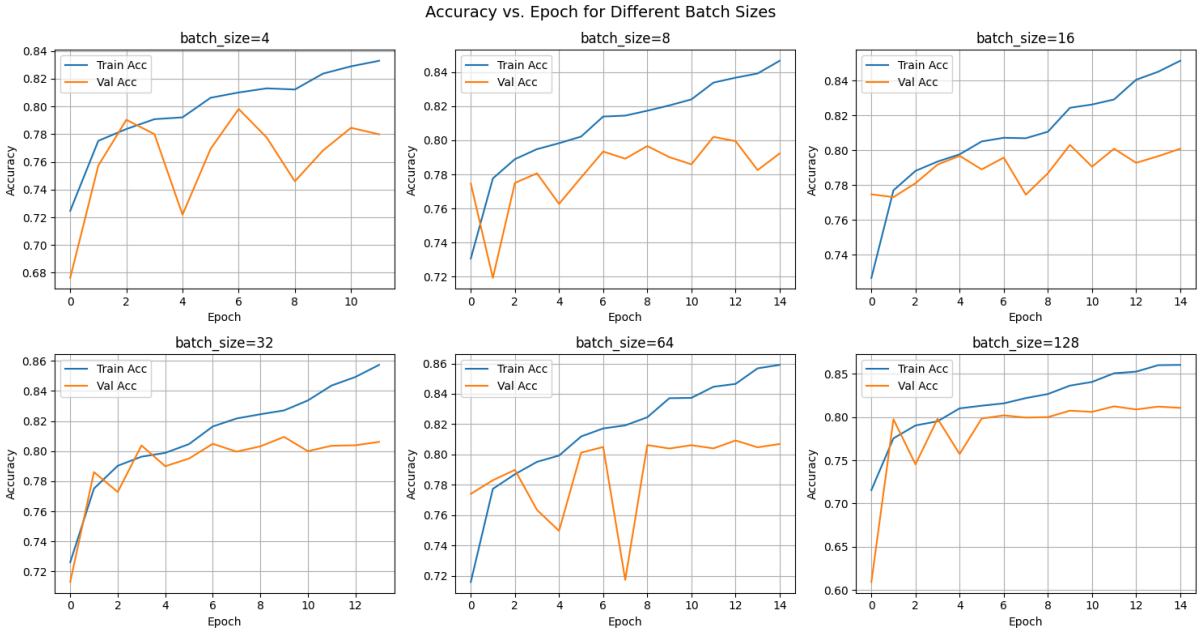


Figure 2. Accuracy plots for different batch sizes

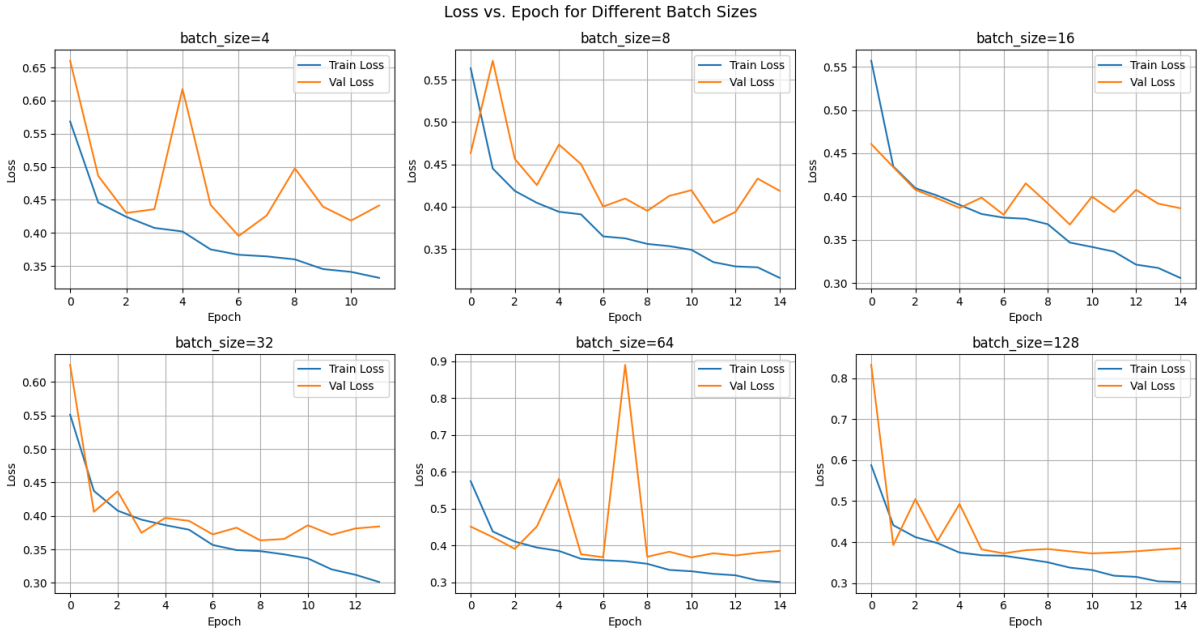


Figure 3. Figure 1. Loss plots for different batch sizes

Figure 4. Confusion matrices for different batch sizes

The obtained results can be seen in Figures 2, 3 and 4. The best performance comes for the values of 32, 64 and 128. Using these values, the Adam, SGD, RMSprop, AdamW and Adadgrad optimizers were tested. The best results were obtained with the Adam, AdamW and RMSprop optimizers. The results obtained for these three optimizers are shown in Figures 5 and 6. (For other results, see the appendix.)

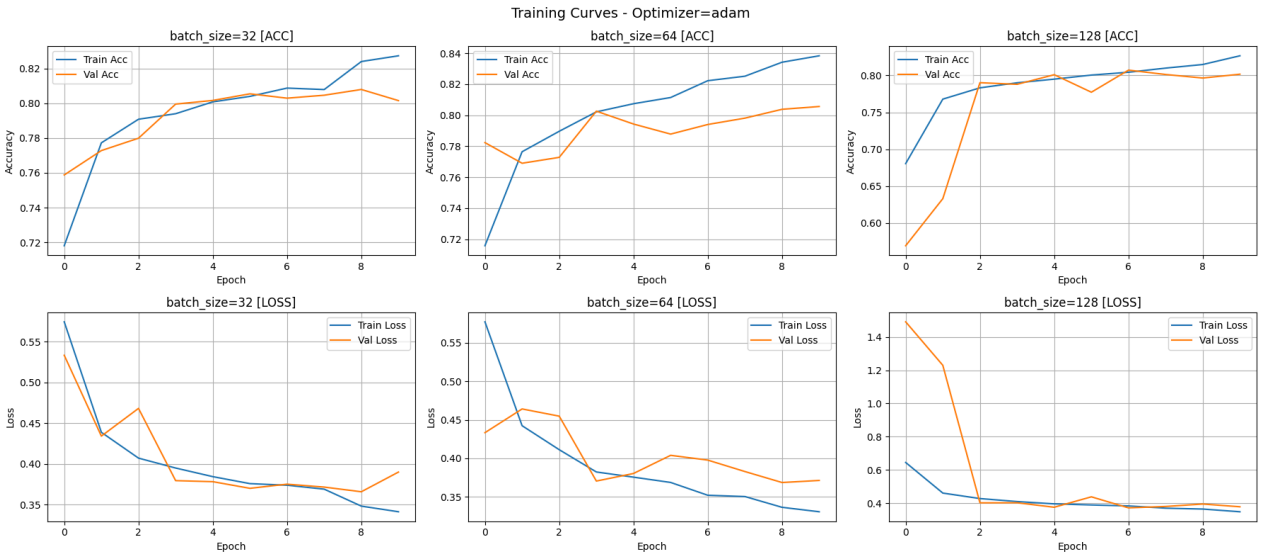


Figure 5. Accuracy and loss plots for Adam optimizer

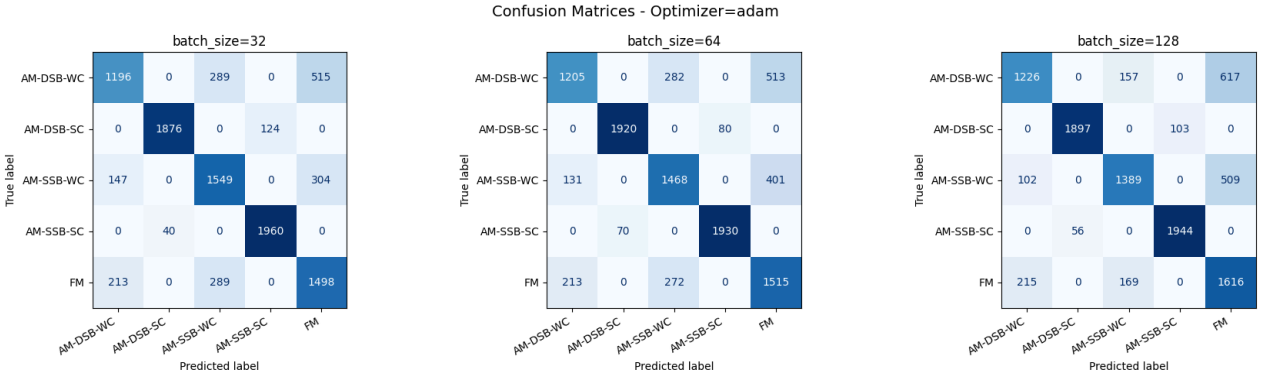


Figure 6. Confusion matrices for Adam optimizer

In order to obtain the most optimum Activision function, Relu, Leakyrelu, Swish, Gelu and Elu functions were tested with the combinations of 3 optimizers and 3 batch size values that gave the best results, respectively. The best results were obtained with the ReLu function. In Figures 7, 8 and 9, the accuracy, loss plots and confusion matrices obtained with the ReLu function can be seen. (For other results, see the appendix.)

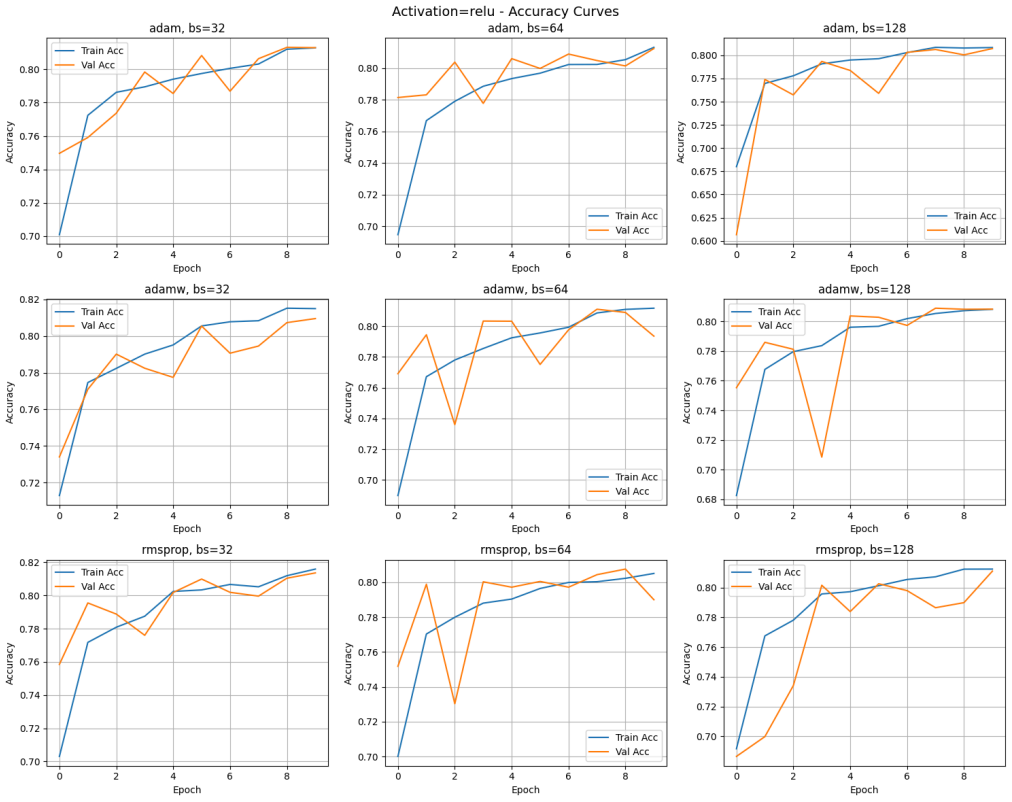


Figure 7. Accuracy plots for relu function

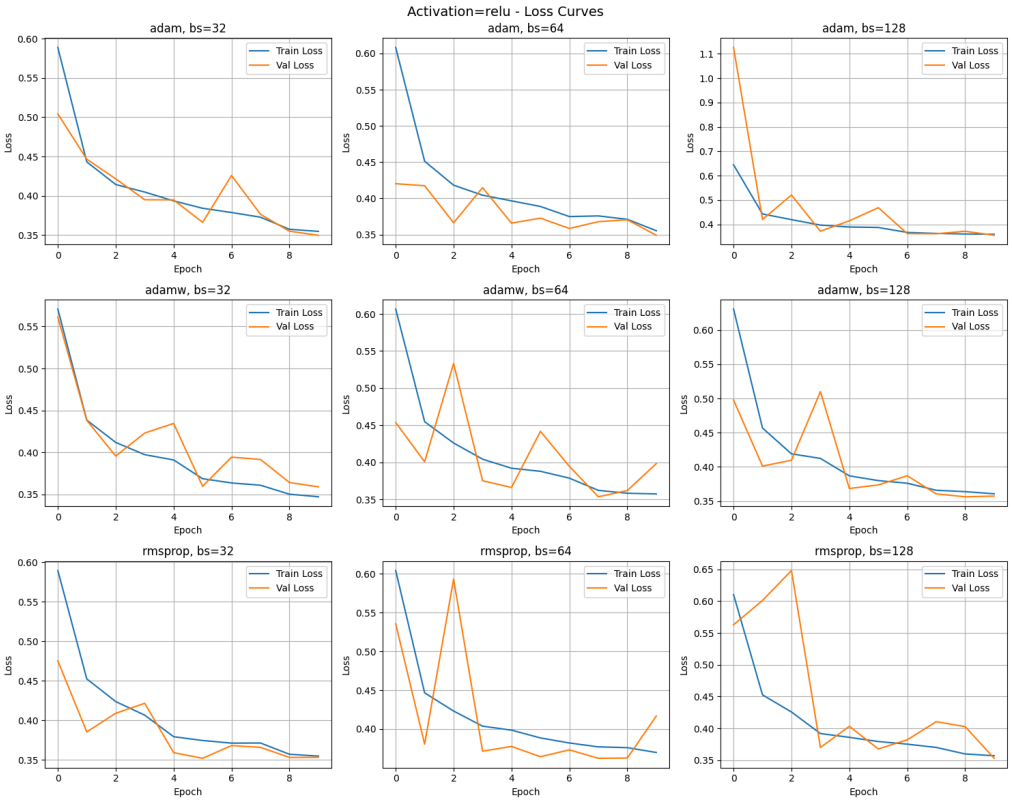


Figure 8. Loss plots for relu function

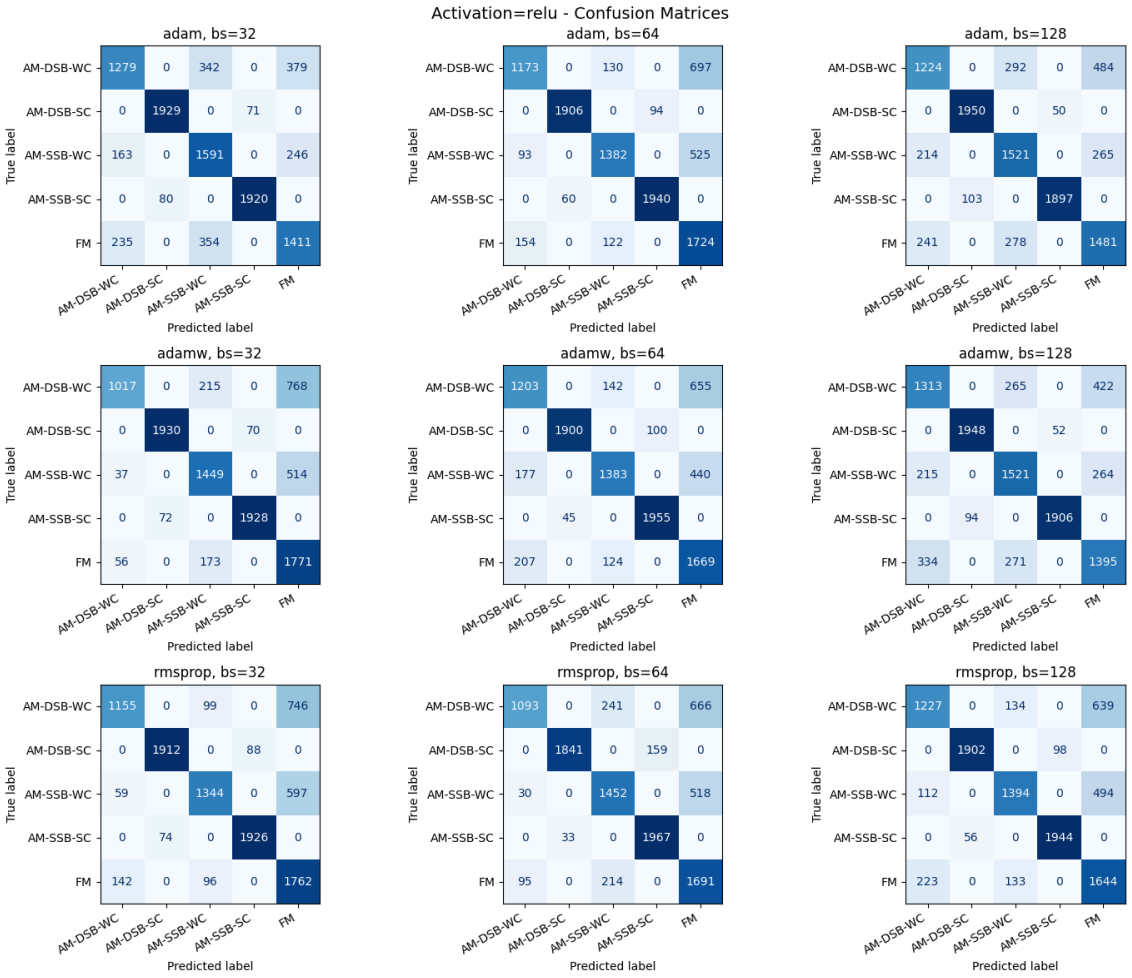


Figure 9. Confusion matrices for relu function

The best combination determined as a result of the tests is the combination of relu, adam optimized and batch size = 64. In this way, the model was tested using different data. In some cases, the combination of relu + selu gives better results. This is because the selu function enables the network to gain self-normalization by keeping the mean of the layer outputs close to zero and the variance close to one. This allows the activations and gradients to remain stable and the network to learn faster. Therefore, both relu and selu functions were used while training some datasets.

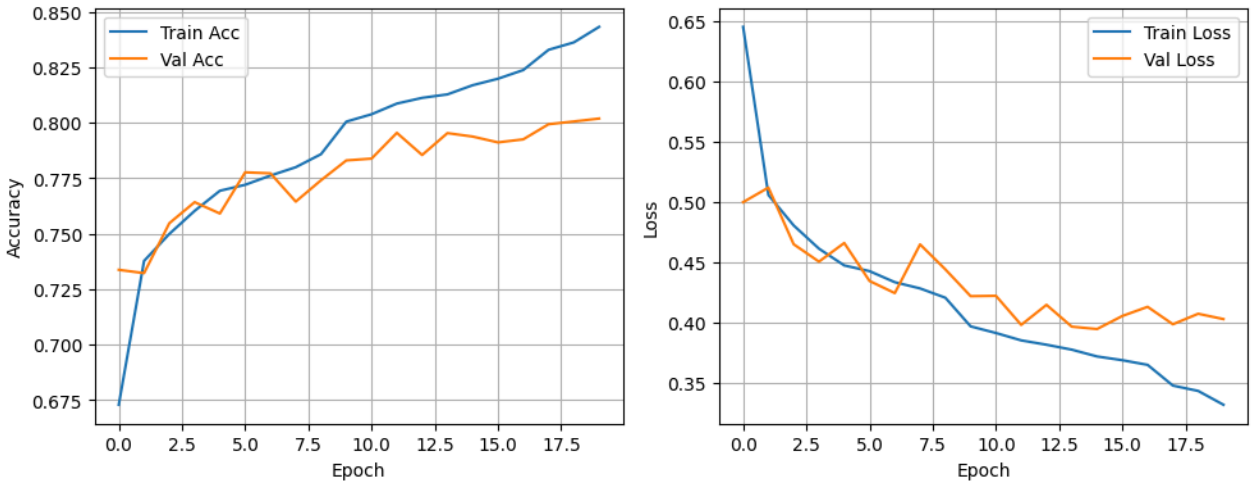


Figure 10. Accuracy and loss plots of trained model

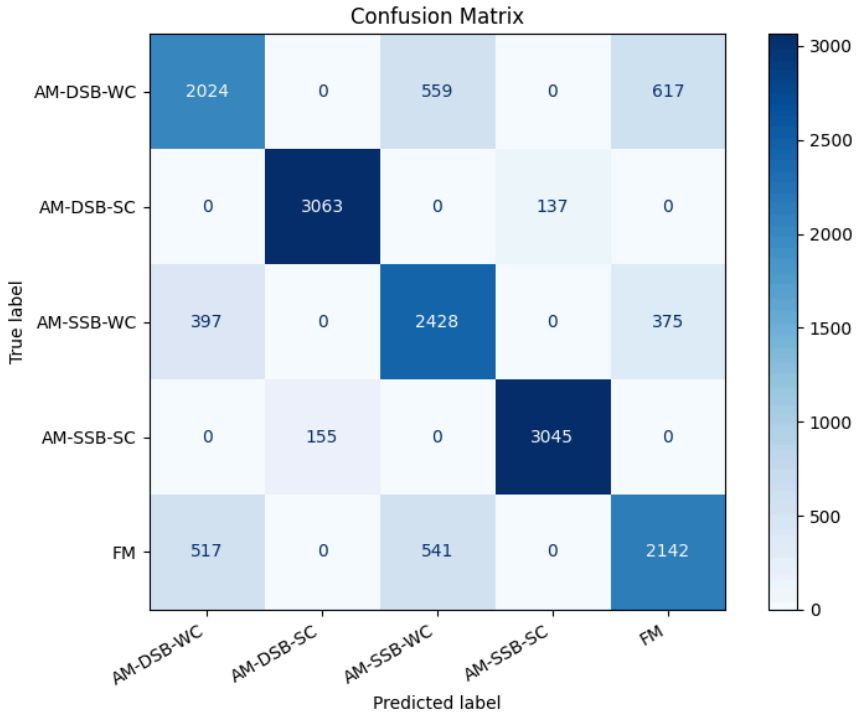


Figure 11. Confusion matrix of trained model

The test results of the trained model can be seen in Figures 10 and 11. The training accuracy reaches up to 85%. The model also fits the training data very well. It can be seen that the errors are gradually decreasing. The trained model has 80% accuracy. It shows lower performance than the training data but provides a generalizable solution. The variation of the trained model for different SNR levels was tested. During the testing, a dataset was used in training, and another dataset was not used in training.

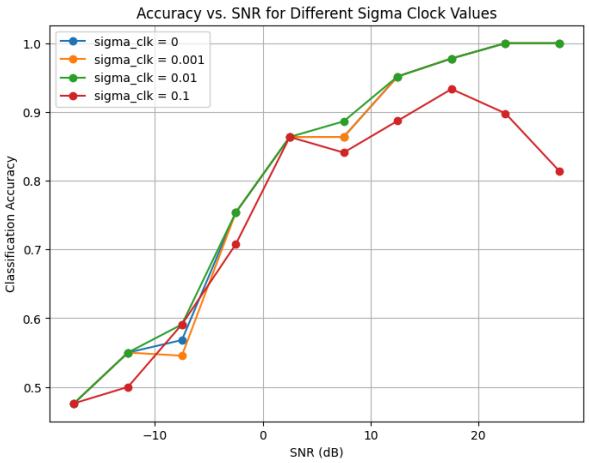


Figure 12. Accuracy vs SNR plot for dataset used in train

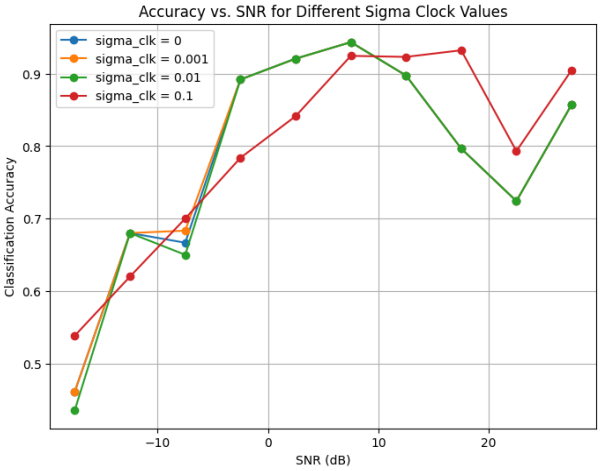


Figure 13. Accuracy vs SNR plot for dataset not used in train

The test result of the data set used in training can be seen in Figure 12. The test result of the data set not used in training can be seen in Figure 13. In the test conducted with the dataset used in training, it is seen that the model's accuracy increases directly proportional to SNR. At low SNR levels, the accuracy rate is 60%, while at high SNR levels, it reaches 90%. This situation shows that the model performs better in conditions where the noise level is low and the signal is more distinct. In addition, as the CFO variance increases, accuracy decreases, especially for low SNR levels. This effect decreases when SNR levels increase. In other words, jitter effects are felt more clearly at low SNR levels. In the test conducted with the dataset not used in training, it is seen that the model's success rate decreases. This is due to the limitations in the model's generalization capacity. However, apart from this, the trends in the plot are consistent with the features mentioned above.

6. Conclusion

In the first phase of the project, synthetic data sets required for a deep learning-based modulation classification system were created. The data generation process was designed and implemented in MATLAB environment for the classification of analog modulation types. Data sets were generated at different SNR levels, considering CFO effects. Randomness sources such as CFO and AWGN were integrated into the data generation process. In this way, distortion and noise that signals may encounter in real-world conditions were simulated. Moreover, a diverse and realistic data set was provided for the classification model training. In the second phase of the project, a model was trained to understand the five given modulation modulation types. For this, the ResNet model was first built. This model was trained with the obtained synthetic data. Parameters such as batch size, optimizer and activation function that would give the best results in model training were tested and selected. Model training was completed with these parameters. The performance of the trained model was examined against different data sets. The performance changes at different SNR and CFO levels were observed. It was determined that increasing SNR levels increased performance. It was concluded that increasing CFO levels negatively affected performance, especially at low SNR levels.

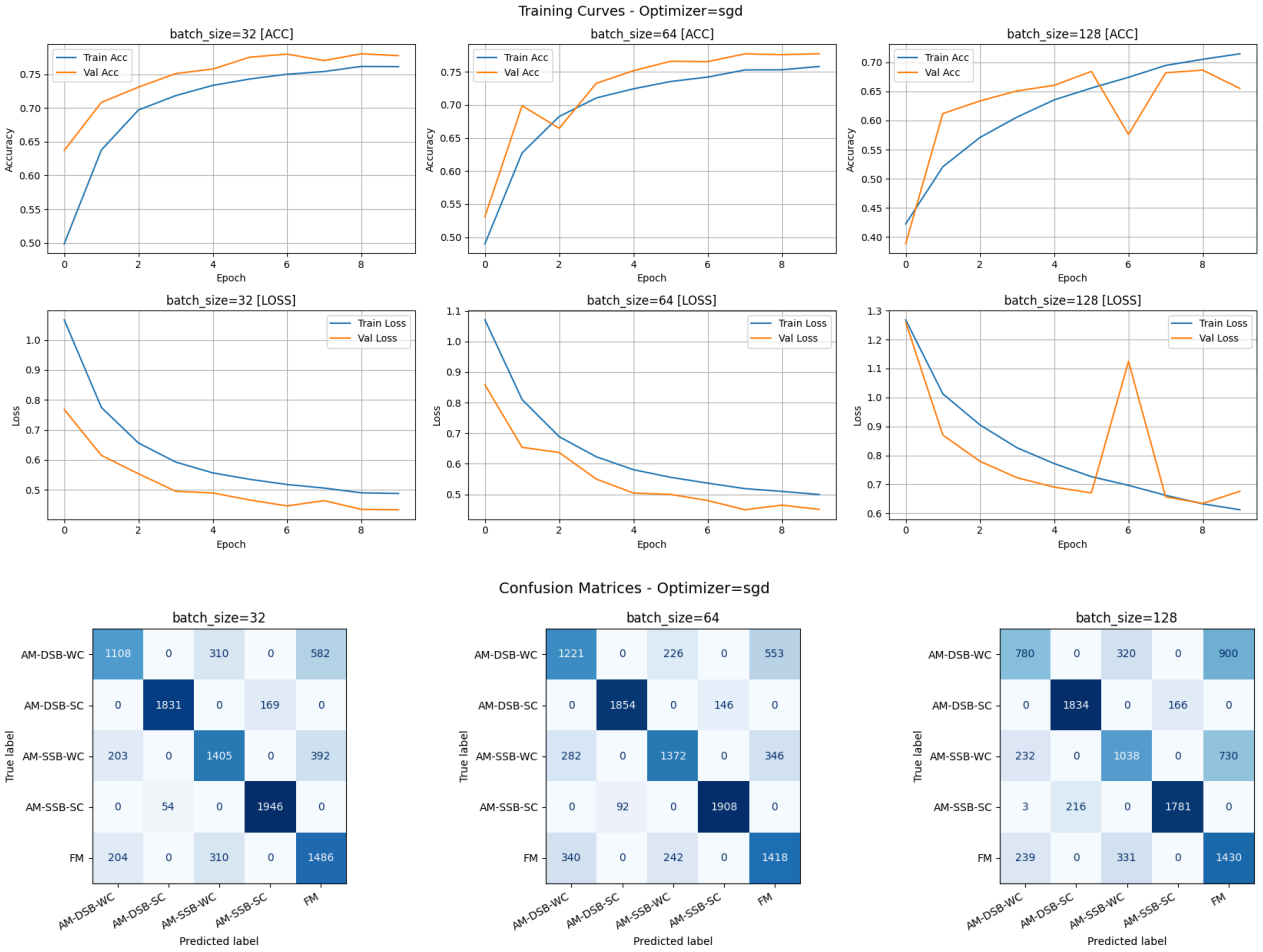
References

[1] T. J. O'Shea, T. Roy and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," in IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168-179, Feb. 2018.

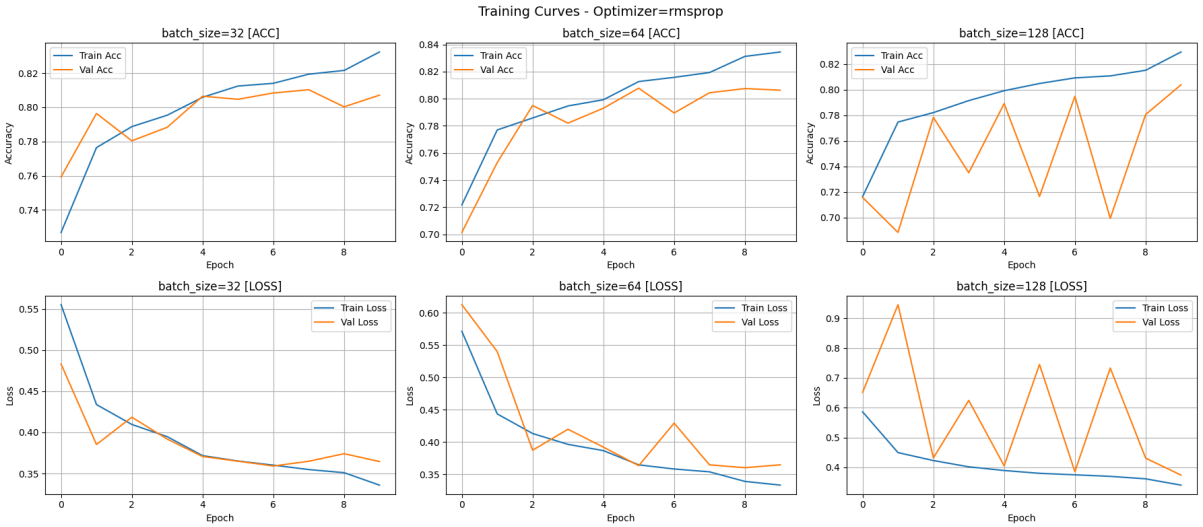
[2] Bello, I., Fedus, W., Du, X., Cubuk, E. D., Srinivas, A., Lin, T. Y., ... & Zoph, B. (2021). Revisiting resnets: Improved training and scaling strategies. *Advances in Neural Information Processing Systems*, *34*, 22614-22627.

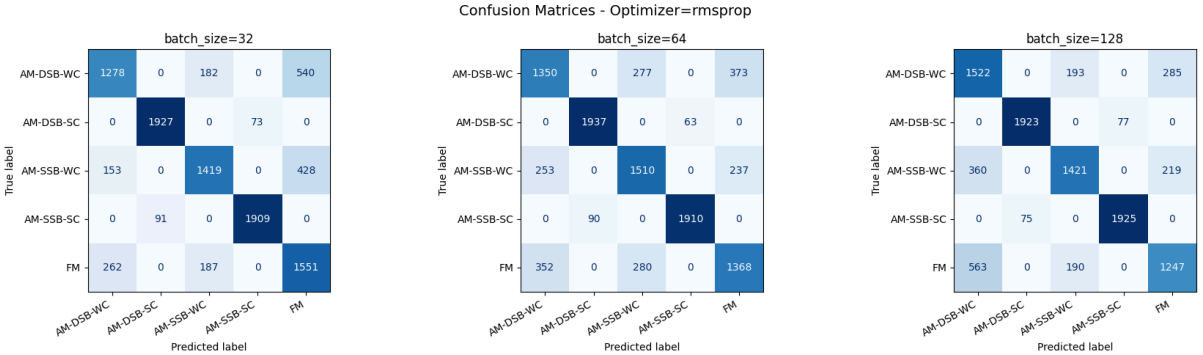
APPENDIX

Training Results for SGD, RMSprop, AdamW and Adadgrad Optimizers

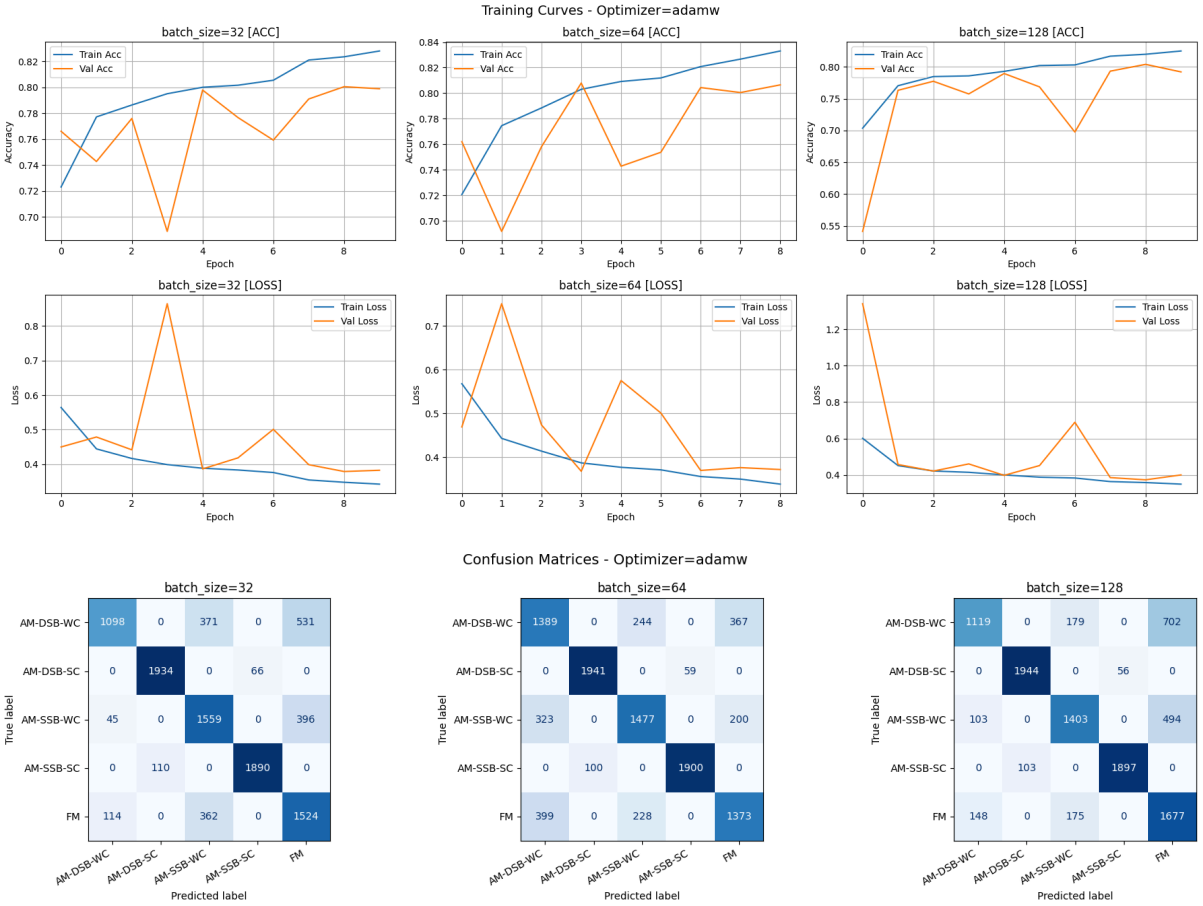


Appendix 1. Accuracy and loss plots and confusion matrix for SGD

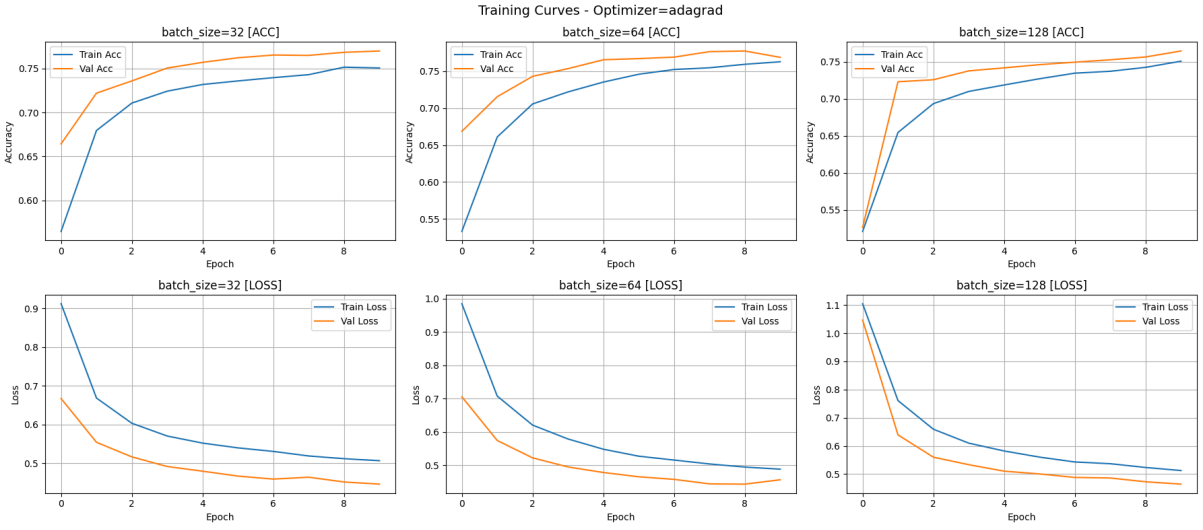


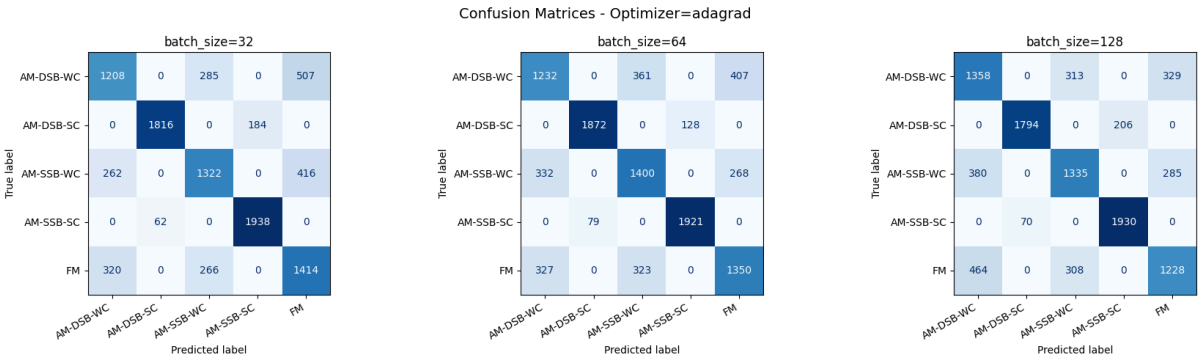


Appendix 2. Accuracy and loss plots and confusion matrix for RMSprop



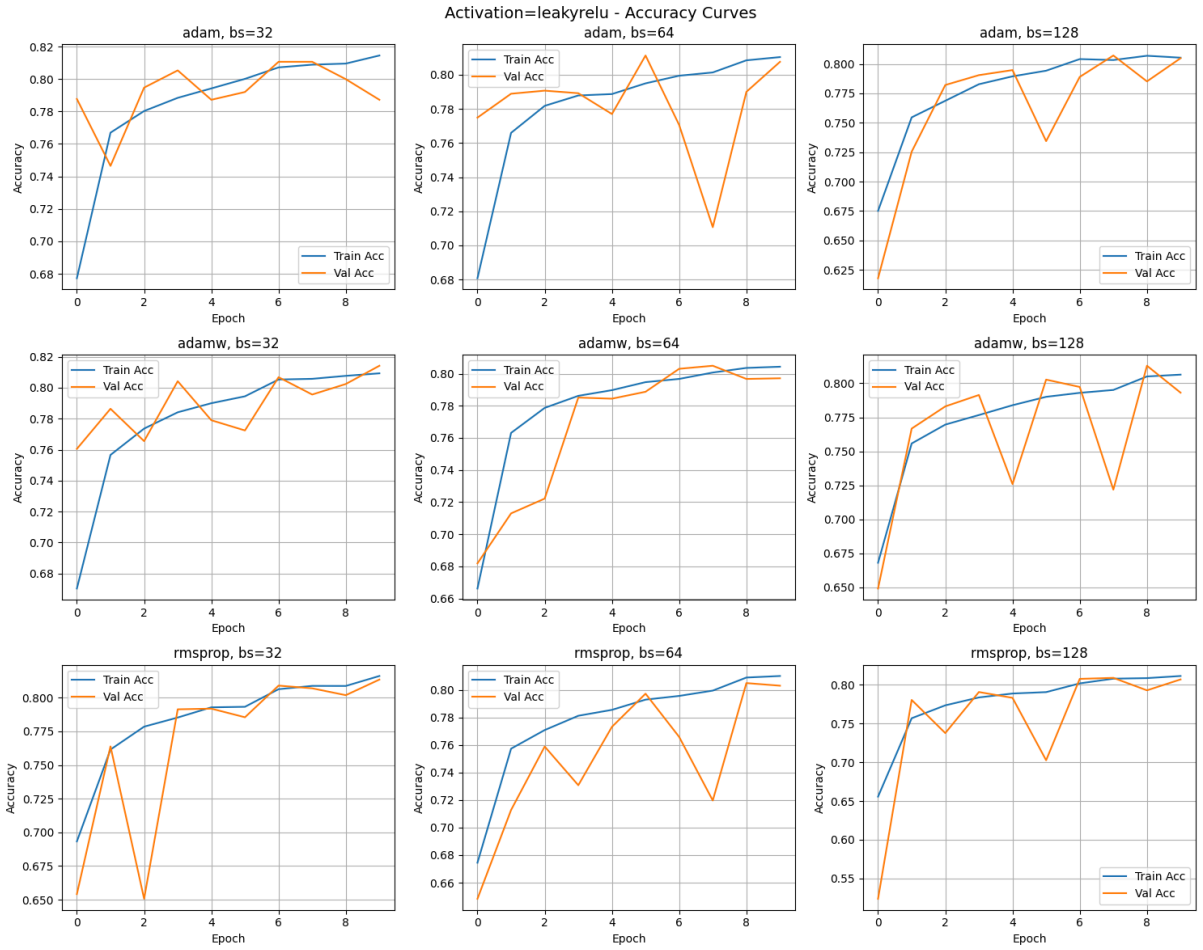
Appendix 3. Accuracy and loss plots and confusion matrix for AdamW

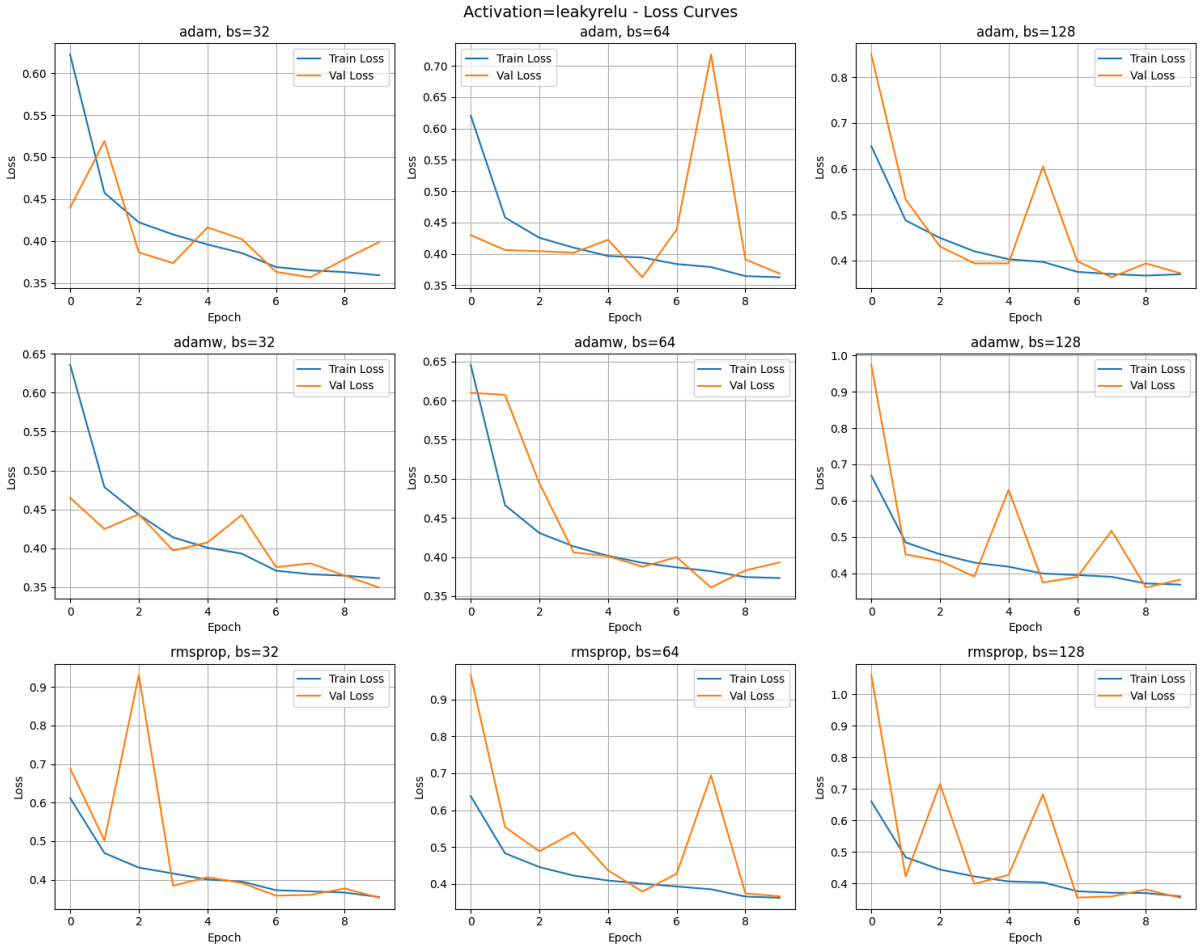


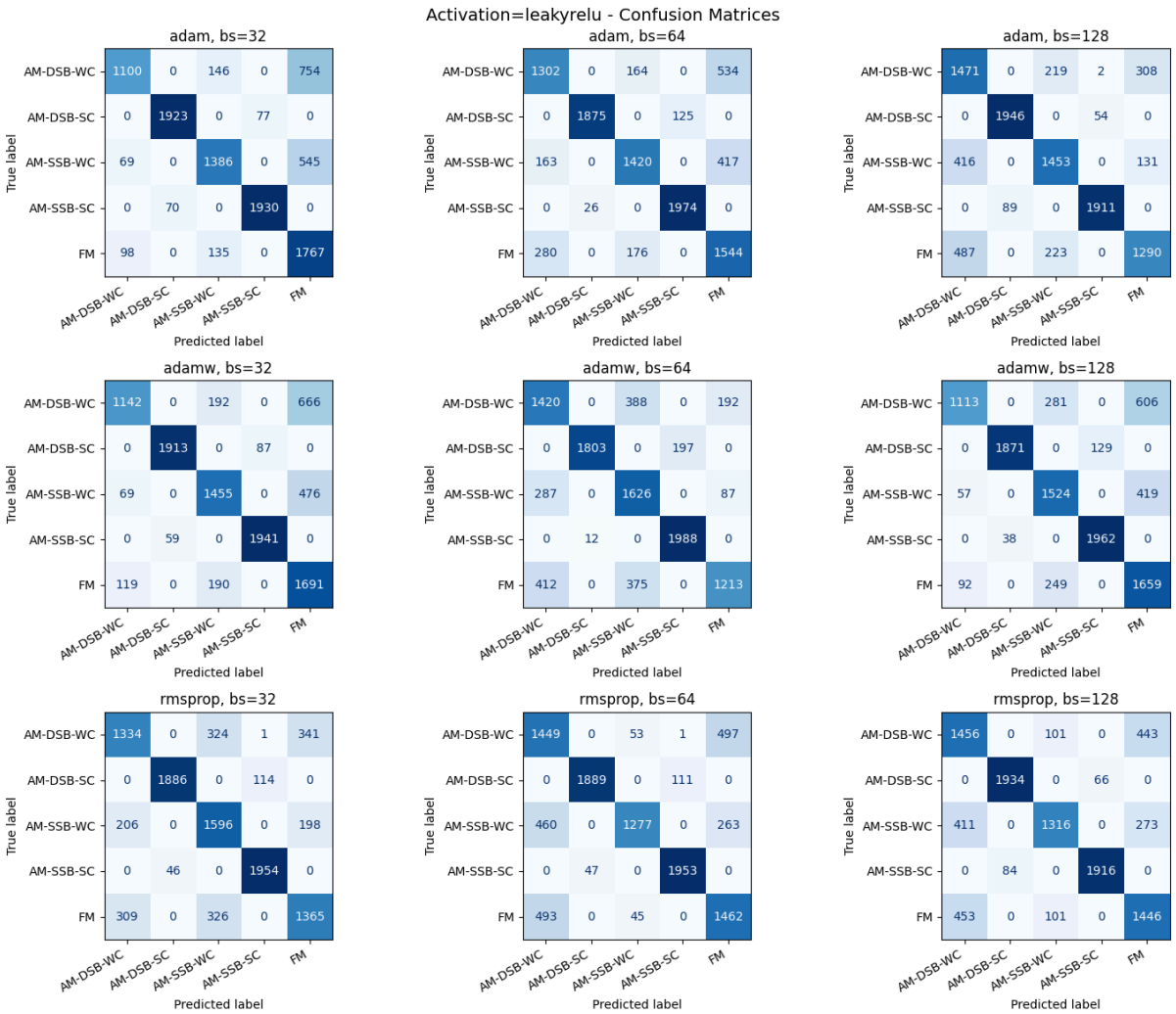


Appendix 4. Accuracy and loss plots and confusion matrix for Adadgrad

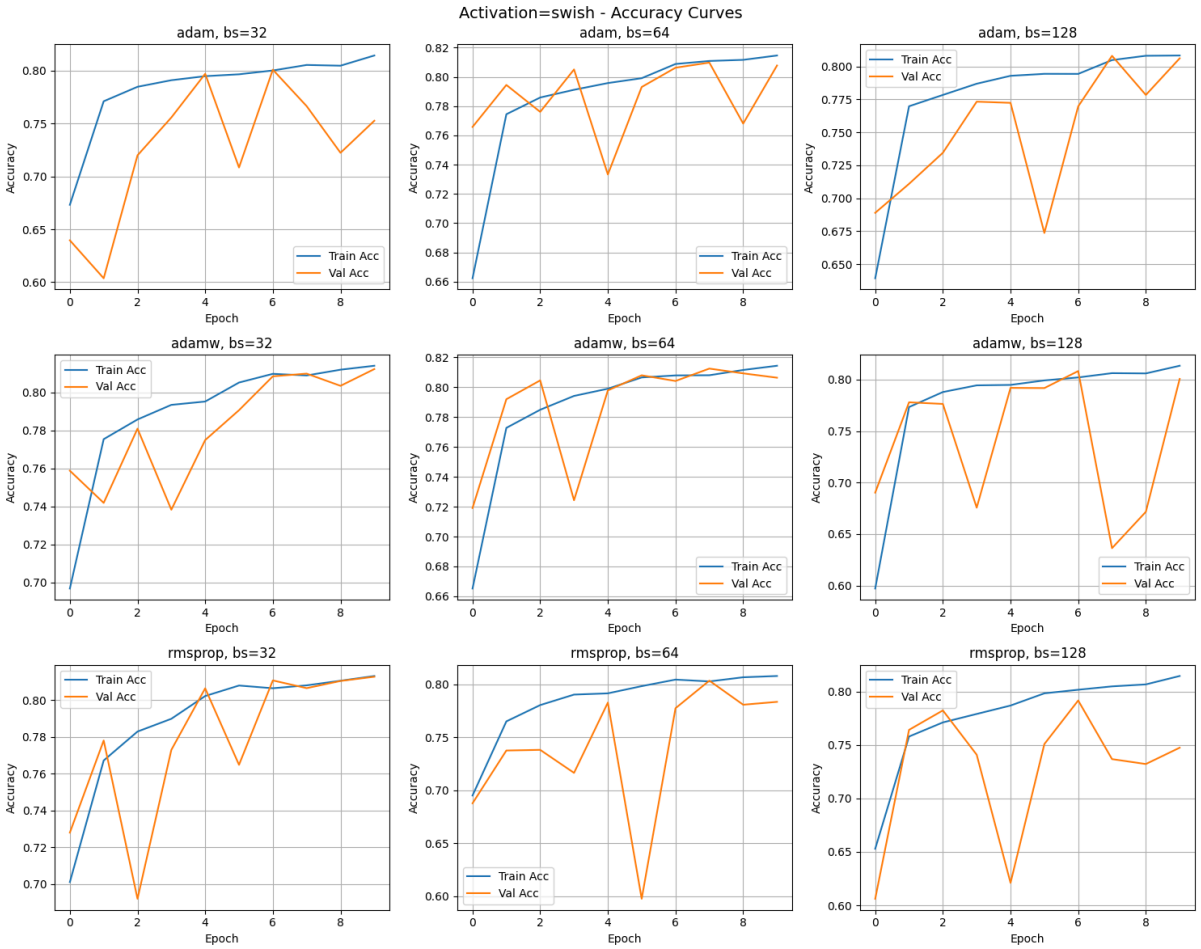
Training Results for Leakyrelu, Swish, Gelu and Elu Activation Functions

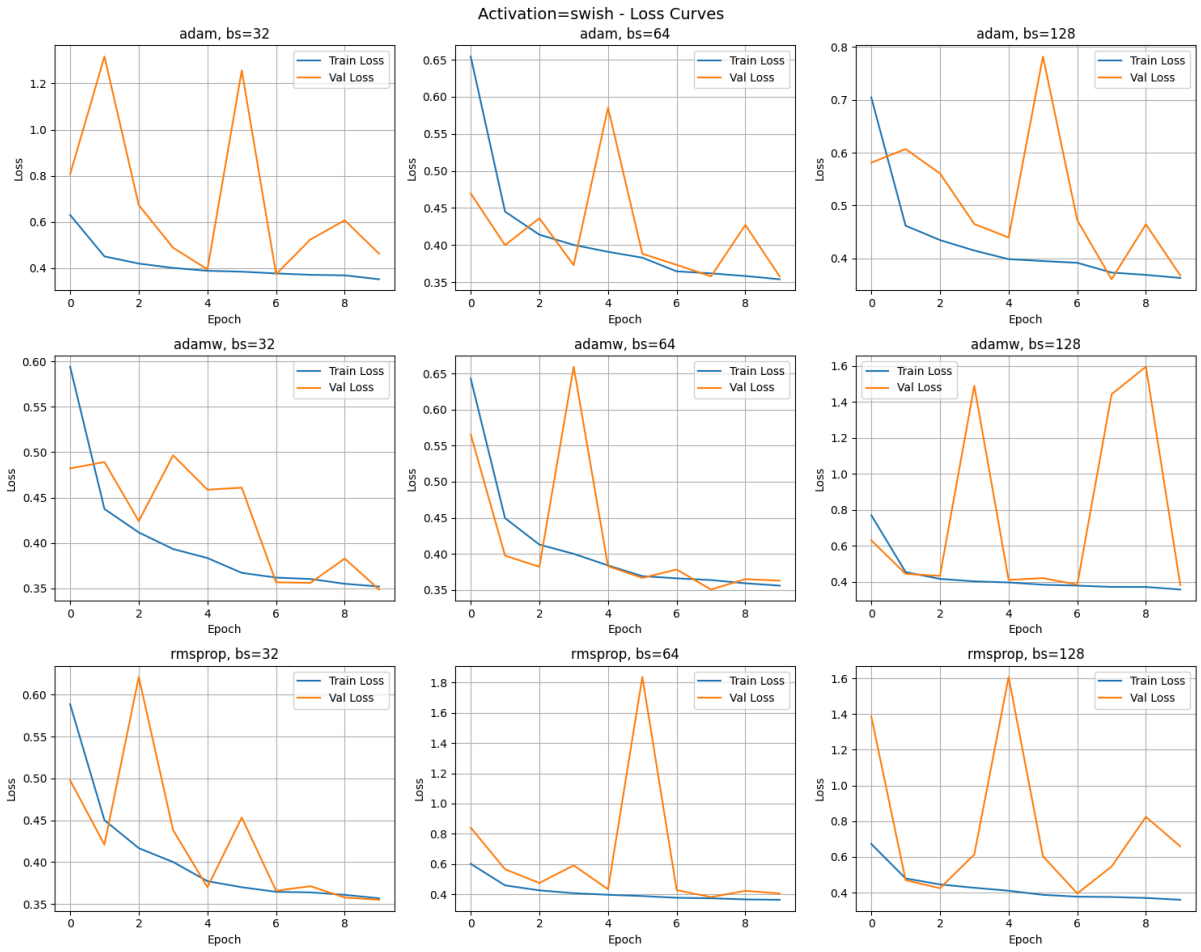


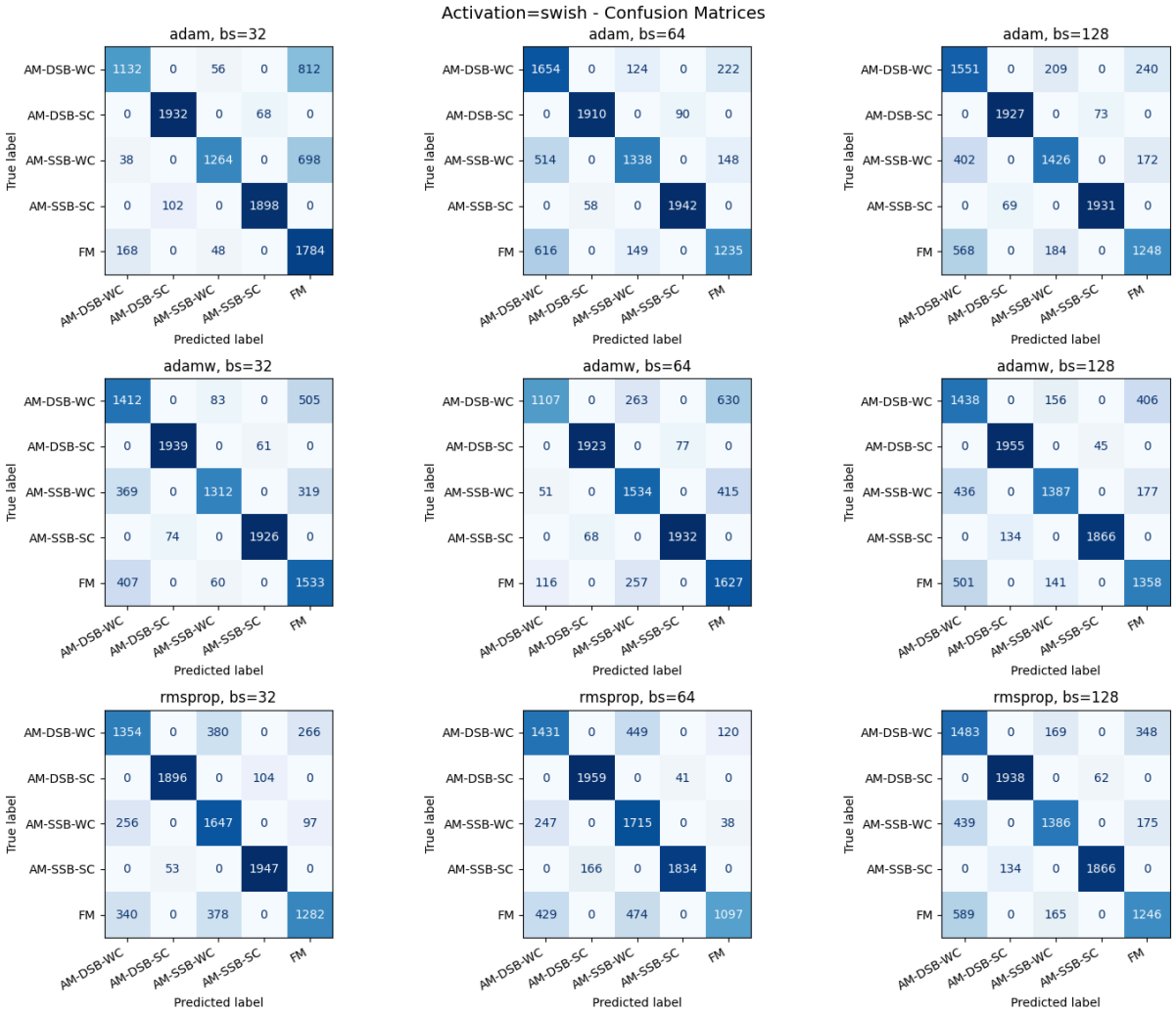

Activation=leakyrelu - Loss Curves
adam, bs=32
adam, bs=64
adam, bs=128
adamw, bs=32
adamw, bs=64
adamw, bs=128
rmsprop, bs=32
rmsprop, bs=64
rmsprop, bs=128
Train Loss
Val Loss
Loss
Epoch


Activation=leakyrelu - Confusion Matrices
adam, bs=32
1100 0 146 0 754
0 1923 0 77 0
69 0 1386 0 545
0 70 0 1930 0
98 0 135 0 1767
adam, bs=64
1302 0 164 0 534
0 1875 0 125 0
163 0 1420 0 417
0 26 0 1974 0
280 0 176 0 1544
adam, bs=128
1471 0 219 2 308
0 1946 0 54 0
416 0 1453 0 131
0 89 0 1911 0
487 0 223 0 1290
adamw, bs=32
1142 0 192 0 666
0 1913 0 87 0
69 0 1455 0 476
0 59 0 1941 0
119 0 190 0 1691
adamw, bs=64
1420 0 388 0 192
0 1803 0 197 0
287 0 1626 0 87
0 12 0 1988 0
412 0 375 0 1213
adamw, bs=128
1113 0 281 0 606
0 1871 0 129 0
57 0 1524 0 419
0 38 0 1962 0
92 0 249 0 1659
rmsprop, bs=32
1334 0 324 1 341
0 1886 0 114 0
206 0 1596 0 198
0 46 0 1954 0
309 0 326 0 1365
rmsprop, bs=64
1449 0 53 1 497
0 1889 0 111 0
460 0 1277 0 263
0 47 0 1953 0
493 0 45 0 1462
rmsprop, bs=128
1456 0 101 0 443
0 1934 0 66 0
411 0 1316 0 273
0 84 0 1916 0
453 0 101 0 1446
AM-DSB-WC
AM-DSB-SC
AM-SSB-WC
AM-SSB-SC
FM
True label
Predicted label

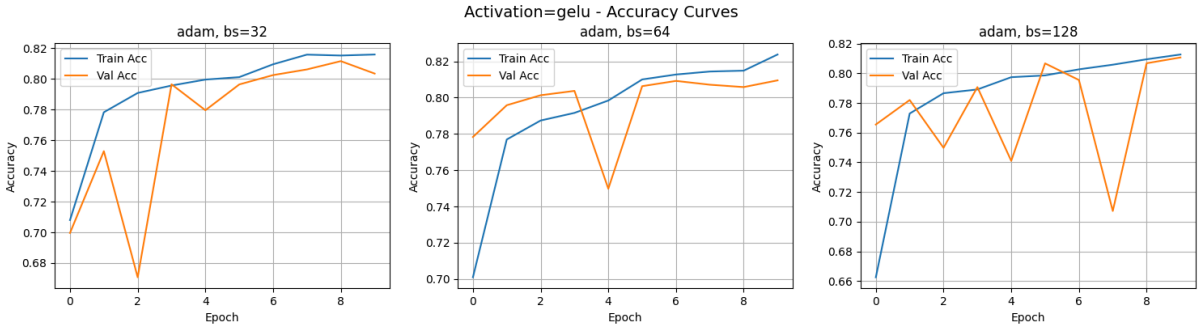
Appendix 5. Accuracy and loss plots and confusion matrix for Leakyrelu

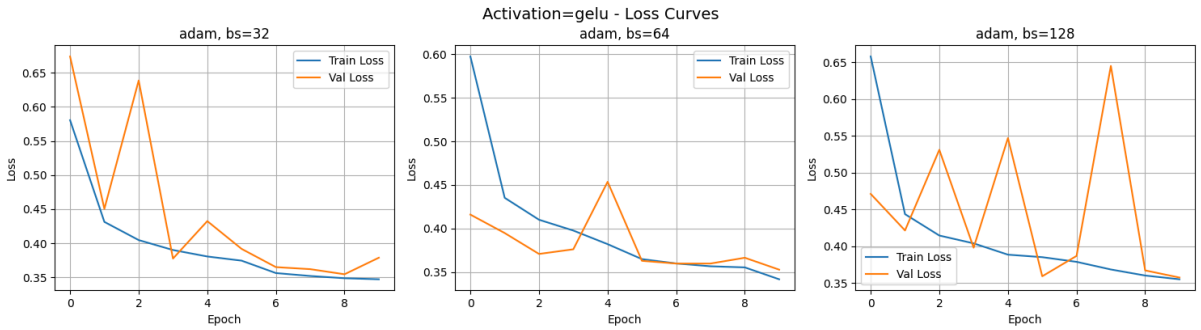


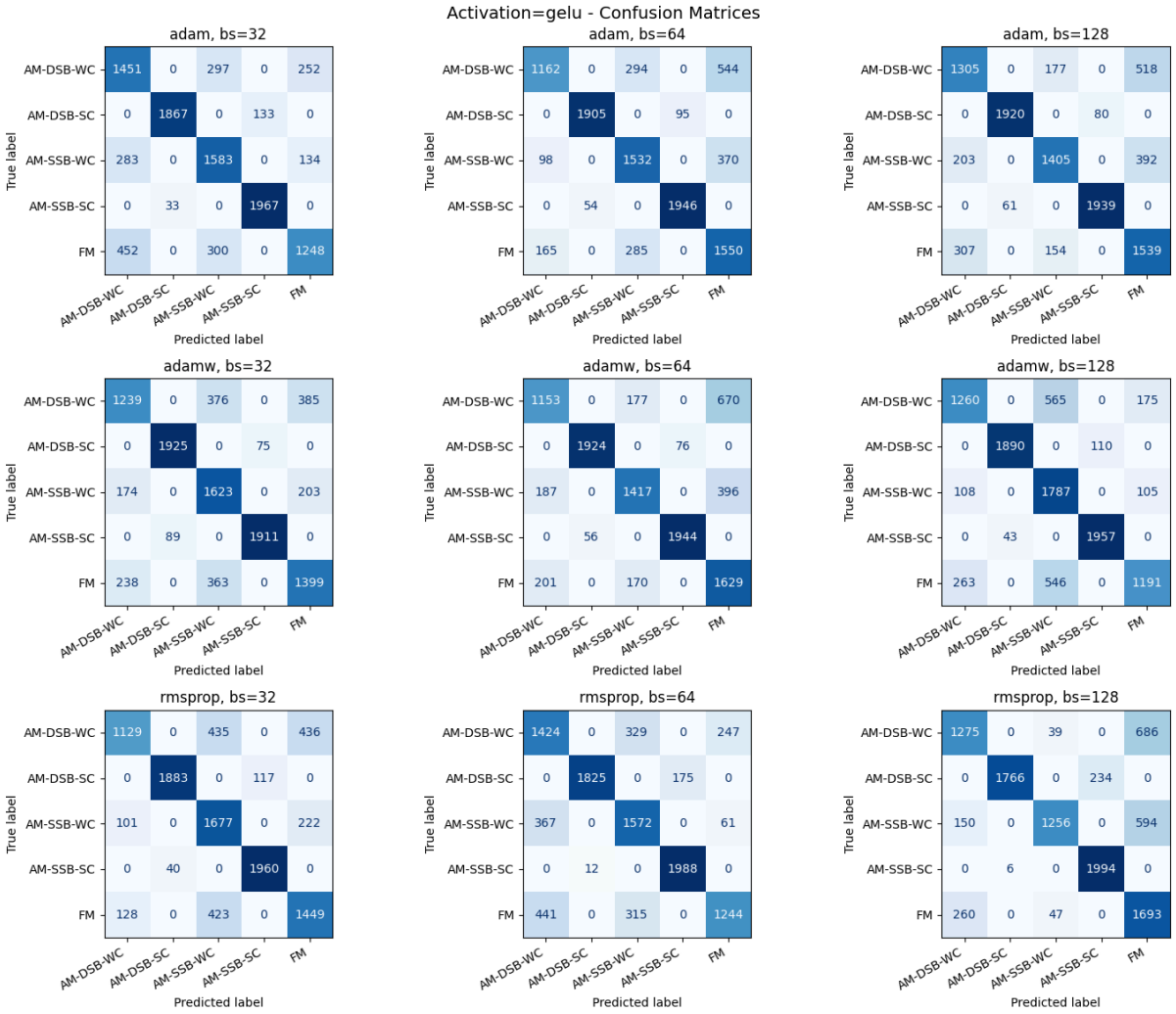




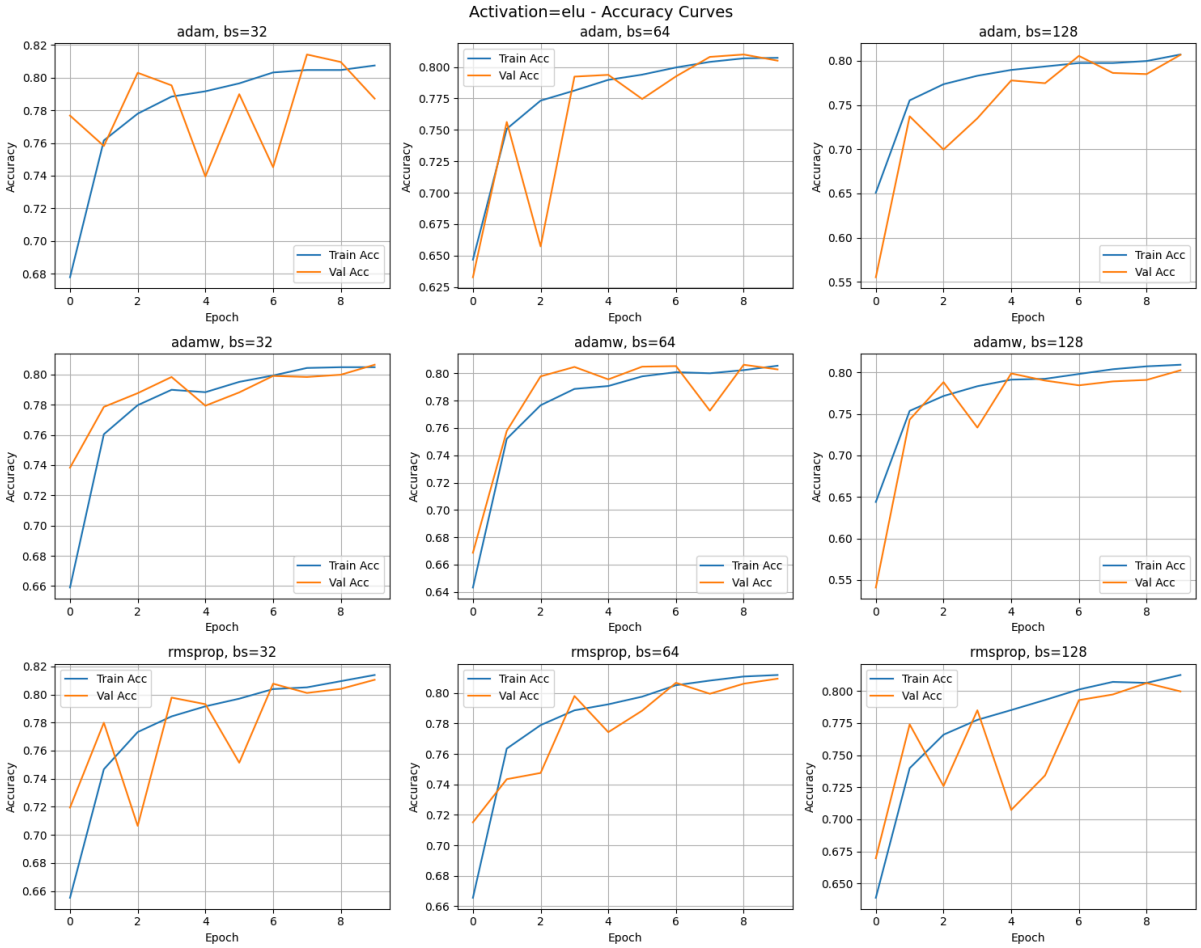
Appendix 6. Accuracy and loss plots and confusion matrix for Swiss

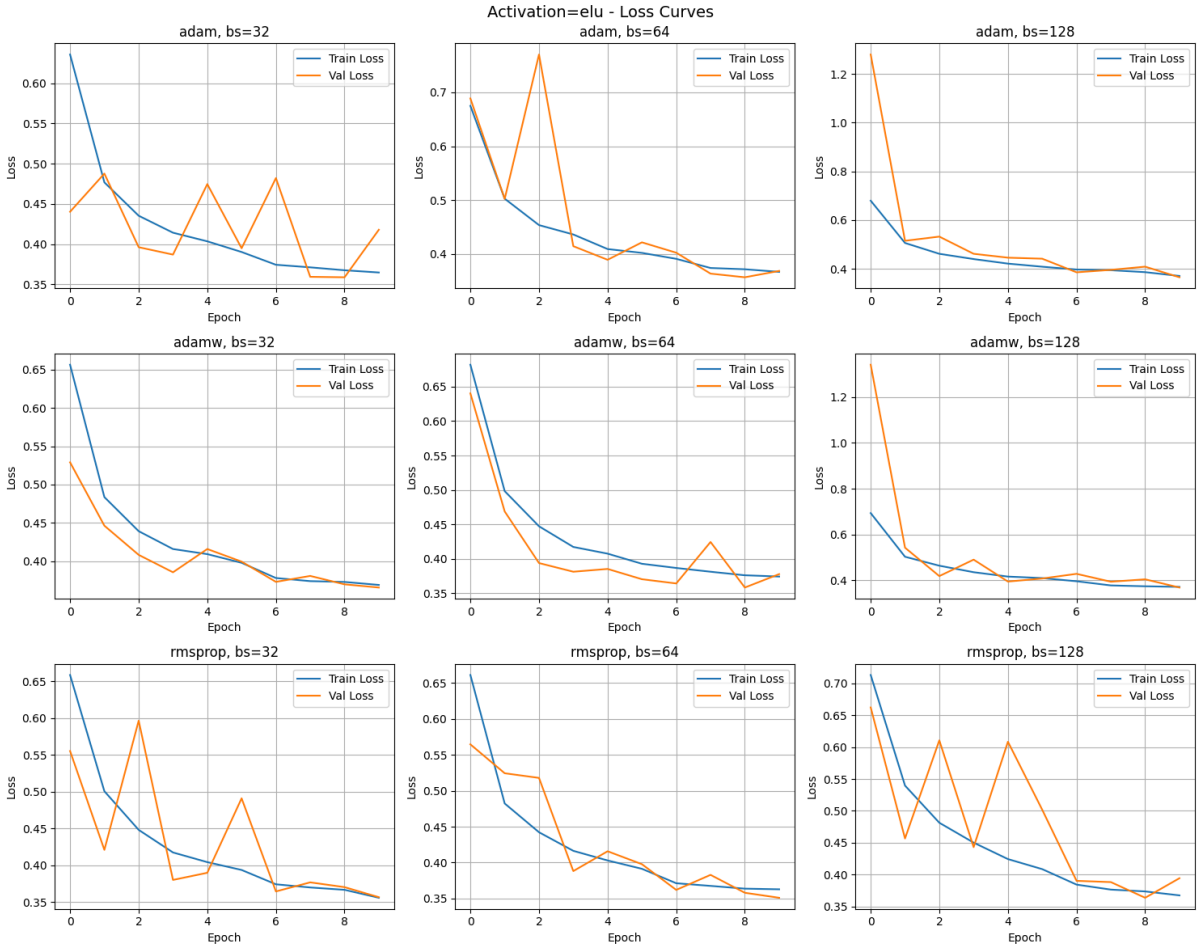

Activation=gelu - Accuracy Curves
adam, bs=32
adam, bs=64
adam, bs=128
adamw, bs=32
adamw, bs=64
adamw, bs=128
rmsprop, bs=32
rmsprop, bs=64
rmsprop, bs=128
Train Acc
Val Acc
Accuracy
Epoch

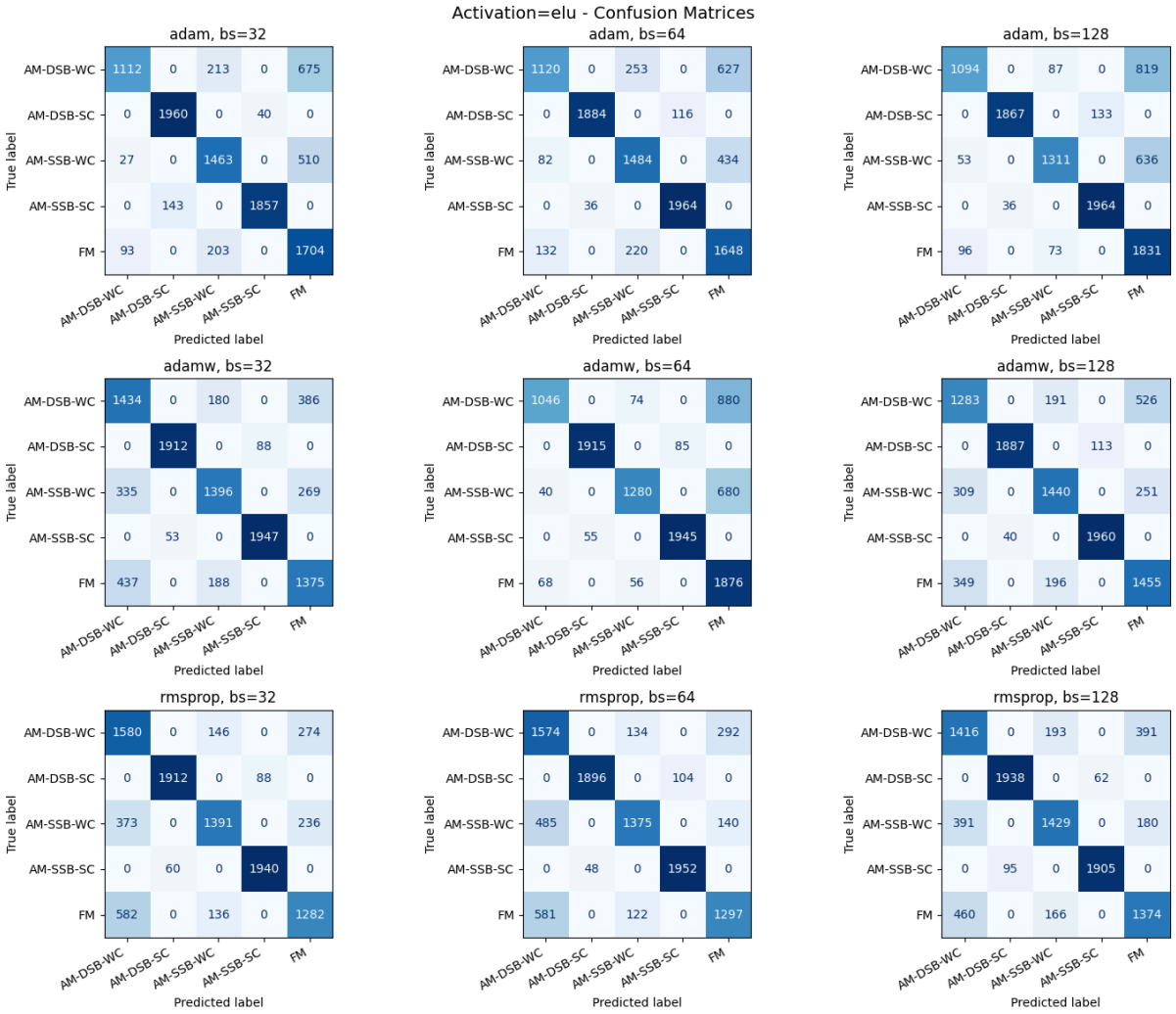

Activation=gelu - Loss Curves
adam, bs=32
adam, bs=64
adam, bs=128
adamw, bs=32
adamw, bs=64
adamw, bs=128
rmsprop, bs=32
rmsprop, bs=64
rmsprop, bs=128
Train Loss
Val Loss
Loss
Epoch



Appendix 7. Accuracy and loss plots and confusion matrix for Gelu


Activation=elu - Accuracy Curves
adam, bs=32
adam, bs=64
adam, bs=128
adamw, bs=32
adamw, bs=64
adamw, bs=128
rmsprop, bs=32
rmsprop, bs=64
rmsprop, bs=128
Train Acc
Val Acc
Accuracy
Epoch


Activation=elu - Loss Curves
adam, bs=32
adam, bs=64
adam, bs=128
adamw, bs=32
adamw, bs=64
adamw, bs=128
rmsprop, bs=32
rmsprop, bs=64
rmsprop, bs=128
Train Loss
Val Loss
Loss
Epoch



Appendix 8. Accuracy and loss plots and confusion matrix for Elu

Data Generation Code

```
close all;
clear all;
clc;
fs = 48e3;
T = 5;
N = 50;
t = (0:(1/fs):T-(1/fs)).';
B = 4e3;
BW_factors = [1,2];
mod_types = {'AM-DSB-WC','AM-DSB-SC','AM-SSB-WC','AM-SSB-SC','FM'};
sigma_clk = 0.1;
snr_high = 30;
snr_low = -20;
f_c = 10e3;
messages =
{'LabT_G_fs_48kHz_messageNumber_1.wav','LabT_G_fs_48kHz_messageNumber_2.wav',...

'LabT_G_fs_48kHz_messageNumber_3.wav','LabT_G_fs_48kHz_messageNumber_4.wav',...

'LabT_G_fs_48kHz_messageNumber_5.wav','LabT_G_fs_48kHz_messageNumber_6.wav',...

'LabT_G_fs_48kHz_messageNumber_7.wav','LabT_G_fs_48kHz_messageNumber_8.wav',...

'LabT_G_fs_48kHz_messageNumber_9.wav','LabT_G_fs_48kHz_messageNumber_10.wav'};
hdf5_filename = 'synthetic_dataset_message_rand.h5';
% Genel bilgi yazma
if isfile(hdf5_filename)
   delete(hdf5_filename);
end
for mt_idx = 1:length(mod_types)
   mod_type = mod_types{mt_idx};
   for m_i = 1:length(messages)
       label = messages{m_i};
       [m,Fs] = audioread(label);
       m = m(:,1);
       m = m./max(m);
       if mod(m_i,2) == 0
           bw_factor = BW_factors(2);
       else
           bw_factor = BW_factors(1);
       end
       for n_i = 1:N
           current_B = B * bw_factor;
           Wn = 2*current_B/fs;
           if Wn > 1
               error('Requested bandwidth is too large compared to fs.');
           end
           filter_order = 128;
           b = fir1(filter_order, Wn, 'low');
           Delta_fc = sigma_clk*randn;
           snr = (snr_high - snr_low)*rand(1,1) + snr_low;
           x_base = generate_modulated_signal(mod_type, f_c + Delta_fc, fs, m,
t);
           %x_channel = apply_rayleigh_channel(x_base, fs);
           x_channel = x_base;
           x_norm = (rms(x_channel))^2;
           noise_power = x_norm/(10^(snr/10));
           noise = randn(size(x_channel))*sqrt(noise_power);
           x_noisy = x_channel + noise;
           I_demod = x_noisy .* cos(2*pi*f_c*t);
           Q_demod = x_noisy .* sin(2*pi*f_c*t);
           I_baseband = filter(b, 1, I_demod);
           Q_baseband = filter(b, 1, Q_demod);
```

```
            x_noisy = I_baseband + 1i.*Q_baseband;
            num_rand = randi((size(x_noisy,1)-1024),1,1);
            x_noisy = x_noisy(num_rand:(num_rand + 1023));
            % HDF5 writing for each signal
            group_name = sprintf('/data/%s/%s/sample_%d', mod_type, label, n_i);
            h5create(hdf5_filename, strcat(group_name, '/signal_real'),
size(real(x_noisy)));
            h5write(hdf5_filename, strcat(group_name, '/signal_real'),
real(x_noisy));
            h5create(hdf5_filename, strcat(group_name, '/signal_imag'),
size(imag(x_noisy)));
            h5write(hdf5_filename, strcat(group_name, '/signal_imag'),
imag(x_noisy));
            h5create(hdf5_filename, strcat(group_name, '/bandwidth'), 1);
            h5write(hdf5_filename, strcat(group_name, '/bandwidth'), current_B);
            h5create(hdf5_filename, strcat(group_name, '/SNR'), 1);
            h5write(hdf5_filename, strcat(group_name, '/SNR'), snr);
            h5create(hdf5_filename, strcat(group_name, '/Delta_fc'), 1);
            h5write(hdf5_filename, strcat(group_name, '/Delta_fc'), Delta_fc);
        end
    end
end
disp('Data generation complete. "synthetic_dataset_message_rand.h5" has been
created.');
function x_base = generate_modulated_signal(mod_type, f_c, fs, m, t)
switch mod_type
    case 'AM-DSB-WC'
        modulation_ind = 0.8;
        x_base = (1 + modulation_ind.*m).*cos(2*pi*f_c*t);
    case 'AM-DSB-SC'
        x_base = m.*cos(2*pi*f_c*t);
    case 'AM-SSB-WC'
        m_hilbert = hilbert(m);
        x_base = (1 + m).*cos(2*pi*f_c*t) - imag(m_hilbert).*sin(2*pi*f_c*t);
    case 'AM-SSB-SC'
        m_hilbert = hilbert(m);
        x_base = m.*cos(2*pi*f_c*t) - imag(m_hilbert).*sin(2*pi*f_c*t);
    case 'FM'
        phi = 2*pi*f_c*t + 2*pi*cumsum(m)/fs;
        x_base = cos(phi);
    otherwise
        error('Unknown modulation type: %s', mod_type);
end
end
function x_out = apply_rayleigh_channel(x_in, fs)
    path_delays = 0.5;
    H = comm.RayleighChannel('SampleRate', fs,'PathDelays', path_delays);
    x_out = H(x_in);
end
```

Model Training Code

```
from google.colab import drive
drive.mount('/content/drive')

import os
os.chdir('/content/drive/MyDrive/Colab_Notebooks/435project')

import h5py
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

def load_hdf5_data(hdf5_filepath, mod_types, num_samples_per_class=20000):
    x_data = []
    y_data = []

    with h5py.File(hdf5_filepath, 'r') as hdf5_file:
        for label_idx, mod_type in enumerate(mod_types):
            data_group = f'/data/{mod_type}'
            if data_group not in hdf5_file['/data']:
                continue

            for message in hdf5_file[data_group].keys():
                msg_group = f"/data/{mod_type}/{message}"
                for sample_key in hdf5_file[msg_group].keys():
                    group_name = f"/data/{mod_type}/{message}/{sample_key}"
                    signal_real = hdf5_file[f"{group_name}/signal_real"][:]
                    signal_imag = hdf5_file[f"{group_name}/signal_imag"][:]

                    signal_real = signal_real.flatten()
                    signal_imag = signal_imag.flatten()

                    signal = np.stack((signal_real, signal_imag), axis=-1)

                    x_data.append(signal)
                    y_data.append(label_idx)

                    if len(x_data) >= num_samples_per_class * len(mod_types):
                        break
                if len(x_data) >= num_samples_per_class * len(mod_types):
                    break

    x_data = np.array(x_data)
    y_data = tf.keras.utils.to_categorical(y_data, num_classes=len(mod_types))

    return x_data, y_data

def normalize_data(x):
    mean_val = np.mean(x, axis=(0,1), keepdims=True)
    std_val = np.std(x, axis=(0,1), keepdims=True) + 1e-9
    x_normed = (x - mean_val) / std_val
    return x_normed

def residual_block(x, filters, kernel_size=3, stride=1):
    shortcut = x
    x = layers.Conv1D(filters, kernel_size, strides=stride, padding='same',
activation='relu')(x)
    x = layers.BatchNormalization()(x)
    x = layers.Conv1D(filters, kernel_size, padding='same')(x)
    x = layers.BatchNormalization()(x)
    if stride != 1 or shortcut.shape[-1] != filters:
        shortcut = layers.Conv1D(filters, kernel_size=1, strides=stride,
padding='same')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)
    x = layers.Add()([x, shortcut])
    x = layers.ReLU()(x)
    return x
```

```
def build_resnet(input_shape=(1024, 2), num_classes=5):
    inputs = layers.Input(shape=input_shape)
    x = layers.Conv1D(64, kernel_size=3, strides=2, padding='same',
activation='relu')(inputs)
    x = layers.BatchNormalization()(x)
    filters = 32
    for i in range(4):
        stride = 2 if i > 0 else 1
        x = residual_block(x, filters, kernel_size=3, stride=stride)
    x = layers.GlobalAveragePooling1D()(x)
    x = layers.Dense(128, activation='selu')(x)
    x = layers.Dropout(0.2)(x)
    x = layers.Dense(64, activation='selu')(x)
    x = layers.Dropout(0.2)(x)
    outputs = layers.Dense(num_classes, activation='softmax')(x)
    model = models.Model(inputs, outputs)
    return model

if __name__ == "__main__":
    mod_types = ['AM-DSB-WC', 'AM-DSB-SC', 'AM-SSB-WC', 'AM-SSB-SC', 'FM']
    hdf5_filepath =
'/content/drive/MyDrive/Colab_Notebooks/435project/synthetic_dataset1000python.h5'
    x_data, y_data = load_hdf5_data(hdf5_filepath, mod_types,
num_samples_per_class=20000)
    x_data = normalize_data(x_data)
    x_train, x_val, y_train, y_val = train_test_split(
        x_data, y_data,
        test_size=0.2,
        random_state=42,
        stratify=y_data.argmax(axis=1)
    )
    resnet_model = build_resnet(input_shape=(1024, 2), num_classes=len(mod_types))
    resnet_model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )
    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=2,
        verbose=1
    )
    history = resnet_model.fit(
        x_train, y_train,
        validation_data=(x_val, y_val),
        epochs=50,
        batch_size=64,
        callbacks=[early_stopping, reduce_lr],
        verbose=1
    )
    test_loss, test_accuracy = resnet_model.evaluate(x_val, y_val, verbose=0)
    print(f"Validation Loss: {test_loss:.4f}, Validation Accuracy:
{test_accuracy:.4f}")
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='Train Acc')
    plt.plot(history.history['val_accuracy'], label='Val Acc')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
```

```
plt.grid(True)
plt.tight_layout()
plt.show()
y_pred = resnet_model.predict(x_val)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_val, axis=1)
cm = confusion_matrix(y_true_classes, y_pred_classes)
plt.figure(figsize=(6,5))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mod_types)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title("Confusion Matrix")
plt.show()
fig, ax = plt.subplots(figsize=(8,6))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mod_types)
disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
plt.setp(ax.get_xticklabels(), rotation=30, ha="right")
plt.title("Confusion Matrix")
plt.tight_layout()
plt.show()
resnet_model.save("my_model.h5")
from google.colab import files
files.download("my_model.h5")
```