UNIVERSITY OF ZILINA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY
Department of Multimedia and Information and Communication Technologies

MACHINE LEARNING FOR CLASSIFICATION OF RF SIGNALS

2026 Cesar Santiago Gonzalez Cuellar

UNIVERSITY OF ŽILINA
Faculty of Electrical Engineering
and Information Technology

Department of multimedia
and information and communication technology

Machine learning for classification of RF signals

Project IKT

Field of study: Informatics
Study program: Information and Communication Technologies
Consultant:: doc. Ing. Juraj Machaj
Department of the consultant: Department of Multimedia and Information and Communication Technologies

Žilina, 2026 Cesar Santiago Gonzalez Cuellar

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
AMC	Automatic Modulation Classification
ASK	Amplitude Shift Keying
BPSK	Binary Phase Shift Keying
CNN	Convolutional Neural Network
CNN1D	One-Dimensional Convolutional Neural Network
DL	Deep Learning
FSK	Frequency Shift Keying
GPU	Graphics Processing Unit
HDF5	Hierarchical Data Format version 5
I/Q	In-phase / Quadrature
LSTM	Long Short-Term Memory
ML	Machine Learning
OFDM	Orthogonal Frequency Division Multiplexing
OOK	On-Off Keying
PM	Phase Modulation
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RF	Radio Frequency
ResNet	Residual Network
ReLU	Rectified Linear Unit
SNR	Signal-to-Noise Ratio
YAML	YAML Ain't Markup Language

List of Symbols

Symbol	Unit	Meaning
I	[–]	In-phase component of a complex signal
Q	[–]	Quadrature component of a complex signal
N	[–]	Number of samples in the dataset
f_s	[Hz]	Sampling frequency
f_c	[Hz]	Carrier frequency of the signal
γ	[dB]	Signal-to-noise ratio
x	[–]	Input signal or input vector of the model
y	[–]	True modulation class
$\hat{y}$	[–]	Predicted modulation class
$\hat{p}_c$	[–]	Predicted probability of class c
c	[–]	Modulation class index
k	[–]	Group index or convolution kernel size, depending on context
L	[–]	Length of the input sequence
μ_B	[–]	Mini-batch mean in batch normalisation
σ_B^2	[–]	Mini-batch variance in batch normalisation
ε	[–]	Small constant for numerical stability
$\mathscr{L}$	[–]	Loss function value

Introduction

Automatic Modulation Classification (AMC) is an important task in wireless communication systems. Its goal is to identify the modulation type of a received radio-frequency (RF) signal before demodulation. This is useful in spectrum monitoring, cognitive radio, interference detection, and adaptive communication systems. [3, 11]

AMC is important because the RF spectrum is limited and expensive. Efficient use of spectrum requires reliable information about the signals occupying it. In adaptive communication systems, knowledge of the modulation type can support decisions such as selecting lower-order modulation under poor channel conditions or using higher-order modulation when the channel quality is good meaning it has low noisy or attenuation. AMC has been also relevant for interference and jamming detection, especially in Internet of Things (IoT) and cognitive radio environments. [3, 11]

Traditional AMC methods are commonly divided into likelihood-based and feature-based approaches. Likelihood-based methods require strong assumptions or prior knowledge about the signal and channel. Feature-based methods extract manually designed characteristics such as amplitude, phase, frequency, or higher-order statistics, and then use classifiers such as Support Vector Machines or Decision Trees. These methods can be computationally efficient, but their performance depends strongly on expert-designed features. [9, 3, 11]

Deep learning offers a different approach. Instead of manually defining features, neural networks can learn useful representations directly from raw I/Q samples. Convolutional Neural Networks (CNNs), recurrent networks such as LSTM, and multi-branch convolutional architectures have been increasingly used for AMC. This project experiments supervised multiclass classification using the RadioML 2018.01A dataset, which contains 2,555,904 labelled I/Q signal examples from 24 modulation classes and 26 SNR values from $-20\,\mathrm{dB}$ to $+30\,\mathrm{dB}$. [1, 11]

In this project, three deep learning architectures are implemented and compared: CLDNN, MCLDNN, and MCNet. CLDNN combines convolutional feature extraction with LSTM-based temporal modelling. MCLDNN extends this idea by using multiple branches to process joint and separate I/Q signal components before recurrent modelling. MCNet uses a deeper multi-branch convolutional structure without recurrent layers. The models are compared using the same dataset split, training configuration, and evaluation procedure.

The project is organised as follows. Chapter 1 presents the theoretical background covering modulation schemes, I/Q signal representation, the RadioML dataset, and relevant deep learning concepts. Chapter 2 reviews the current state of AMC methods. Chapter 3 defines the objectives of the work. Chapter 4 describes the methodology, dataset pipeline, model architectures, and training process. Chapter 5 presents and discusses the experimental results. Chapter 5.5 summarises the conclusions and possible future work. [11, 3]

1 Theoretical Background

1.1 Radio Frequency Modulation Schemes

Modulation is the process of encoding information into carrier signals by varying one or more of their properties: amplitude, frequency, or phase. In communication systems, modulation schemes can generally be divided into two categories: analogue modulation and digital modulation. Understanding their characteristics is essential for developing classifiers capable of distinguishing signals under noisy channel conditions [3, 11].

1.1.1 Analogue Modulation Schemes

Analogue modulation continuously varies a carrier signal according to the input message signal; in our case, analogue modulation schemes represent different types of architecture which can be labeled as the type of modulation classified. [6]

Amplitude Modulation (AM)

Amplitude Modulation (AM) encodes information by varying the amplitude of the carrier signal while keeping its frequency and phase constant.[11]

Frequency Modulation (FM)

Frequency Modulation (FM) conveys information by continuously varying the carrier frequency according to the input signal amplitude. FM provides improved noise immunity compared to AM and is widely used in radio broadcasting. [3, 9]

1.1.2 Digital Modulation Schemes

Digital modulation maps discrete symbol values into changes in amplitude, phase, or frequency of the carrier signal, but at the same time, for our project is a possible label to be classified by the models. [6]

Amplitude Digital Modulations

Amplitude Shift Keying (ASK) Amplitude Shift Keying (ASK) is a digital modulation technique in which binary or multi-level digital data modifies the amplitude of a high-frequency carrier waveform. Different symbol values are represented by discrete amplitude levels. From this modulation,**OOK** is derived, and other variations from ASK include **ASK4** and **ASK8**, which differ in the number of amplitude levels, as ASK uses only 2. Then ASK4 uses 4, and ASK8 uses 8, which improves efficiency in data transmission but worsens robustness towards noise.[11]

Amplitude Shift Keying (OOK) On-Off Keying (OOK) is defined as the simplest and most basic form of Amplitude Shift Keying (ASK). The fundamental process of OOK involves switching the carrier signal either completely önör öff"to represent and transmit binary data. [11]

Phase Digital Modulations

Phase Modulation (PM) represents information through continuous changes in the phase of the carrier signal. PM is closely related to FM, as both belong to the family of angle modulation techniques.[3]

Binary Phase Shift Keying (BPSK) Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying (PSK). It uses two distinct carrier phases, typically 0° and 180°, to represent binary values. Each symbol carries one bit of information. [7]

Quadrature Phase Shift Keying (QPSK) Quadrature Phase Shift Keying (QPSK) extends BPSK by using four distinct carrier phases. Since each symbol represents two bits, QPSK doubles the spectral efficiency compared to BPSK. [7]

Offset Quadrature Phase Shift Keying (OQPSK) Offset Quadrature Phase Shift Keying (OQPSK) is a variation of QPSK in which transitions between the in-phase and quadrature components are offset in time. This reduces abrupt phase changes and improves signal stability in practical communication systems. [7]

Analogue Quadrature Amplitude Modulation (QAM) In analogue QAM, two analogue message signals are transmitted together using two carrier waves that are out of phase by 90 degrees. This allows more than one analogue signal to be carried over the same channel. In this sense, analogue QAM is closely related to amplitude modulation, but it uses two quadrature carriers instead of only one. [7, 9]

Higher-Order PSK Higher-order Phase Shift Keying (PSK) schemes such as 8PSK and 16PSK increase spectral efficiency by using a larger number of distinct phase states to represent digital symbols. In these modulation techniques, each symbol carries multiple bits of information, allowing higher data transmission rates within the same bandwidth. However, as the number of phase states increases, the constellation points become closer together, making the modulation more sensitive to noise, phase errors, and signal distortion. Therefore, higher-order PSK schemes require better channel conditions and higher signal-to-noise ratios to maintain reliable communication. [7]

- **8PSK** is a modulation scheme that uses 8 different phase states to represent data, allowing the transmission of 3 bits per symbol. It provides better spectral efficiency than QPSK but is more sensitive to noise and phase distortion.
- **16PSK** uses 16 distinct phase states, enabling the transmission of 4 bits per symbol. It offers higher data rates and improved bandwidth efficiency, but it requires a significantly higher signal-to-noise ratio because the constellation points are more closely spaced.

Combined Amplitude-Phase-Based Digital Modulation

Quadrature Amplitude Modulation (QAM) In digital QAM, discrete digital symbols are represented by changes in both the amplitude and phase of the carrier signal. It can be understood as a combination of Amplitude Shift Keying (ASK) and Phase Shift Keying (PSK). Digital QAM is widely used in wireless communication systems such as cellular networks and Wi-Fi because it supports high data rates and efficient bandwidth utilization, [3, 11].

- **QAM16** is a modulation scheme with 16 constellation points, allowing the transmission of 4 bits per symbol. It provides good noise resistance and moderate spectral efficiency.
- **QAM32** uses 32 constellation points and transmits 5 bits per symbol. It offers higher data rates than QAM16 but requires a better signal-to-noise ratio.
- **QAM64** employs 64 constellation points, enabling the transmission of 6 bits per symbol. It is widely used in modern wireless systems due to its high spectral efficiency.

Amplitude and Phase Shift Keying (APSK) Amplitude and Phase Shift Keying (APSK) is a digital modulation method that combines variations in both amplitude and phase to represent digital information. In APSK modulation, constellation points are arranged in multiple concentric rings, with each ring representing a different amplitude level and the symbols within each ring differing in phase. Modulation schemes such as APSK16, APSK32, and APSK64 use 16, 32, and 64 constellation symbols, respectively, enabling higher data transmission efficiency. Compared to conventional QAM, APSK provides improved performance in nonlinear communication channels and is therefore commonly used in satellite and broadcasting systems such as DVB-S2. [3]

- **APSK16** is a modulation technique with 16 symbols arranged in concentric rings, combining amplitude and phase variations to improve performance in nonlinear channels.
- **APSK32** uses 32 constellation symbols distributed across multiple amplitude rings, providing higher spectral efficiency and robustness for satellite communications.
- **APSK64** is a higher-order APSK modulation scheme with 64 symbols, capable of transmitting large amounts of data efficiently while requiring high signal quality.

Frequency-Based Digital Modulation

Gaussian Minimum Shift Keying (GMSK) Gaussian Minimum Shift Keying (GMSK) is a spectrally efficient variant of FSK that applies Gaussian filtering before frequency modulation. This reduces spectral sidelobes and minimizes interference between adjacent channels. GMSK is widely used in GSM communication systems. [7, 11]

1.2 I/Q Signal Representation

A bandpass signal can be represented in complex baseband form as:

$$s(t) = I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t), \tag{1.1}$$

where $I(t)$ is the in-phase component, $Q(t)$ is the quadrature component, and f_c is the carrier frequency. Together, (I, Q) form a two-dimensional representation of the signal. This is the representation used in the RadioML dataset. [7, 11]

1.3 The RadioML 2018.01A Dataset

RadioML 2018.01A is a benchmark dataset for AMC research. It was generated using GNU Radio and includes simulated channel impairments such as additive white Gaussian noise, multipath fading, carrier frequency offset, and clock offset. [6, 2].

Tab. 1.1: Summary of the RadioML 2018.01A dataset.

Property	**Value**
Total samples	2 555 904
Modulation classes	24
Samples per signal	1 024
Signal channels	2 (I and Q)
SNR range	-20 dB to $+30$ dB, step 2 dB
SNR levels	26
File format	HDF5
File size	approximately 25 GB

The dataset arrays are structured as follows:

- **X**: shape $(2{,}555{,}904 \times 1{,}024 \times 2)$ raw I/Q samples;
- **Y**: shape $(2{,}555{,}904 \times 24)$ one-hot modulation labels;
- **Z**: shape $(2{,}555{,}904)$ SNR values in dB.

The 24 modulation classes are OOK, 4ASK, 8ASK, BPSK, QPSK, 8PSK, 16PSK, 32PSK, 16APSK, 32APSK, 64APSK, 128APSK, 16QAM, 32QAM, 64QAM, 128QAM, 256QAM, AM-SSB-WC, AM-SSB-SC, AM-DSB-WC, AM-DSB-SC, FM, GMSK, and OQPSK.

1.4 Deep Learning Fundamentals

This section introduces the main deep learning concepts used in this project. The explanation follows the project workflow, which is dataset splitting, input representation, architecture-specific model designs, training objective, and evaluation metrics.

1.4.1 Supervised Learning

Supervised learning is a type of machine learning where an algorithm is trained using a "labeled"dataset. This means that for every piece of data you give the computer, you also provide the "correct answer"[10, 9, 1].

Dataset Splitting

Dataset splitting is the process of dividing the dataset frames or samples(information) into training where we modify the parameters and the model learns what to predict, then we have validation, where we stop modifiying the parameters, and we just let them run, but we do monitor the classification with the correct labels, and lastly we have testing, from where we display the final information from our models. [4]

Common Splitting Strategies

Different problem settings require different splitting strategies. **Random split.**The most common approach. Examples are randomly assigned to each subset, typically using a fixed random seed for reproducibility, as we did in our case by setting it to 42, since we wanted to compare three models using the same samples on each one.

Stratified split. Used in our project to split training, validation, and testing subsets with very similar class and SNR distributions across samples. Stratification ensures that each subset preserves the same class proportions as the original dataset, preventing any subset from being dominated by the majority class. **Time-series split.** Applied to temporally ordered data, where using future information to predict past events would constitute leakage. The data are split chronologically: earlier observations form the training set, and later observations form the test set.

***K*-fold cross-validation.** The dataset is partitioned into K equal folds. The model is trained K times; in each iteration, one fold serves as the validation set and the remaining $K-1$ folds serve as the training set. The final performance estimate is the average across all K runs. This technique makes efficient use of limited data and reduces the variance of the performance estimate.

Data Leakage

A critical failure mode in dataset splitting is *data leakage*, which occurs when information from the validation or test set inadvertently influences the training process.

In the context of this work, the dataset is split into training, validation, and test subsets before any preprocessing. Model selection and hyperparameter tuning are performed using the validation set, and the reported classification results are computed solely on the test set.

1.4.2 Convolutional Feature Extraction

All three models use 1D convolutional layers to extract local temporal features from the I/Q sequence. For a 1D signal x and a filter w of length k, the convolution at position n is:

$$(x * w)[n] = \sum_{m=0}^{k-1} x[n+m]w[m]. \tag{1.2}$$

In AMC, convolutional filters learn local changes in amplitude, phase, and frequency-related patterns that help distinguish modulation schemes.

1.4.3 Activation Functions

After convolutional layers, nonlinear activation functions are applied. In this project, the Rectified Linear Unit (ReLU) is used:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{1.3}$$

ReLU introduces nonlinearity into the model, allowing the network to learn more complex decision boundaries than a purely linear model.

1.4.4 Batch Normalisation

Batch normalisation is applied after convolutional layers to stabilise training. It normalises activations inside a mini-batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, \tag{1.4}$$

where μ_B and σ_B^2 are the mini-batch mean and variance, and ε is a small constant for numerical stability.

1.4.5 Pooling

Pooling layers reduce the temporal resolution of feature maps while preserving important information. Max pooling selects the strongest activation in a local

$$y[n] = \max_{m \in R_n} x[m]. \tag{1.5}$$

In this project, max-pooling is used to downsample intermediate feature maps. Adaptive average pooling is also used before the final classifier in the ResNet1D models to compress the temporal dimension into a fixed-size feature vector.

1.4.6 Dropout and Regularisation

Dropout is used to reduce overfitting by randomly setting a fraction of activations to zero during training. This prevents the model from relying too strongly on individual neurons and improves generalisation to unseen signals.

1.4.7 CNN1D Architecture

The CNN1D model is used as the simplest baseline architecture in this project. It applies one-dimensional convolutional layers directly to the I/Q signal, where the two input channels correspond to the in-phase and quadrature components. The convolutional layers extract local temporal patterns from the raw signal, such as changes in amplitude and phase that are useful for distinguishing modulation types.

The implemented CNN1D model contains several convolutional blocks with batch normalisation, ReLU activation, dropout, and max pooling. After convolutional feature extraction, adaptive average pooling compresses the temporal dimension into a fixed-size feature vector. This vector is then passed through fully connected layers to produce one output logit for each modulation class.

This architecture is useful as a baseline because it evaluates how well local convolutional features alone can classify modulation types without recurrent layers or residual connections.

1.4.8 CNN-LSTM Architecture

The CNN-LSTM model combines convolutional feature extraction with recurrent sequence modelling. First, one-dimensional convolutional layers extract local temporal features from the I/Q input signal and reduce the temporal length using pooling. The resulting feature map is then transposed into a sequence format and passed to an LSTM layer.

In this project, the CNN-LSTM model uses a bidirectional LSTM. This allows the model to process the extracted feature sequence in both forward and backward directions. Instead of using only the last LSTM output, the model averages the LSTM outputs across the sequence and then applies a final classification layer.

This architecture is more expressive than CNN1D because it combines local feature extraction with temporal dependency modelling. It is therefore suitable for evaluating whether recurrent sequence modelling improves modulation classification performance.

1.4.9 ResNet1D Architecture

The ResNet1D model adapts residual learning to one-dimensional I/Q signal. It begins with a convolutional stem that reduces the temporal resolution of the input signal and increases the number of feature channels. After the stem, the model applies several residual stages.

Each residual block contains two one-dimensional convolutional layers with batch normalisation, ReLU activation, and dropout. The output of these layers is added to a shortcut connection. When the number of channels or the temporal resolution changes, a projection shortcut is used to match the dimensions.

Residual connections help improve gradient flow and allow deeper convolutional models to be trained more stably. In this project, ResNet1D is used to evaluate whether deeper hierarchical convolutional feature extraction improves AMC performance compared with the simpler CNN1D and CNN-LSTM models.

1.4.10 Classification Layer and Softmax

The final layer of each model outputs 24 logits, one for each modulation class. The softmax function converts these logits into class probabilities:

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}, \tag{1.6}$$

where $C = 24$ is the number of classes and z_i is the logit for class i.

The predicted class is selected using the argmax decision rule:

$$\hat{y} = \arg\max_i z_i. \tag{1.7}$$

1.4.11 Loss Function and Optimisation

The models are trained using categorical cross-entropy loss. In PyTorch, this is implemented as cross-entropy loss applied directly to logits:

$$\mathscr{L} = -\sum_{i=1}^{C} y_i \log(p_i), \tag{1.8}$$

where y_i is the true one-hot label and p_i is the predicted probability for class i.

Although the raw dataset stores labels as one-hot vectors, the training pipeline converts them into integer class indices before applying cross-entropy loss.

The Adam optimiser is used with a learning rate of 0.001. Adam adapts the learning rate for each parameter using estimates of the first and second moments of the gradients, making it effective for training deep neural networks.

1.4.12 Model Complexity and Comparison

The three architectures differ in modelling capacity and inductive bias. The CNN1D model is the simplest and focuses mainly on local temporal features.

The CNN-LSTM model adds recurrent sequence modelling, making it better suited for capturing longer-term dependencies in the I/Q sequence.

The ResNet1D model increases depth using residual connections, allowing it to learn more complex hierarchical features while maintaining stable gradient flow.

1.4.13 Evaluation Metrics

Overall accuracy is used as the main performance metric, but it is not sufficient by itself for a 24-class AMC problem. Therefore, additional metrics are used, including per-class accuracy, precision, recall, F1-score, and confusion matrices.

The confusion matrix shows which modulation classes are the most inaccurate predictions from the model. This is especially important in AMC because similar modulation types, such as QPSK and OQPSK or high-order QAM schemes, may have similar signal structures, which makes sense for the model to mislabel on these.

Performance is also evaluated as a function of SNR. Low SNR values make classification more difficult because noise hides useful signal patterns, while higher SNR values usually allow the models to classify modulations more accurately. Therefore, SNR-conditioned accuracy is essential for understanding model behaviour across different channel conditions.

2 State of the Art

2.1 Background of AMC Methods

First, I state that the focus of this project is Automatic Modulation Classification (AMC), an essential component of RF wireless communication receivers. The core challenge is to compare advanced machine deep learning architectures.

There are several reasons why AMC is a significant problem in wireless communication. Firstly, we know that the RF spectrum is expensive, as we are unable to increase the bandwidth. Even if that were possible, it would also increase the noise of the channel, which would significantly decrease the channel's effectiveness.

Additionally, we have problems in different fields like spectrum management, where, depending on channel quality, we may need to set lower order modulation and a higher coding rate for better BER (bit error rate) or higher-order modulation with a lower coding rate when we have better channel quality.

Another important field is interference detection, where jamming signals may have a different modulation type than those authorized transmission signals, which are very common attacks in the Internet of Things (IoT). The timeline of the approaches used so far starts from likelihood-based extraction, after which they perform featured-based methods which improved this process because they didnt require high computational complexity, however, featured-based methods do have other tradeoffs like the limited learning capacity, these because ML classification methods like Support Vector Machines or Decision Trees contain fixed architectures, meaning that the parameters they take into account for classification are fixed input data.

This is a big change in deep learning models like Convolutional Neural Networks, where the different values used to set and configure the architecture of the model change with biases and weights; this is when the model uses backpropagation.

2.1.1 Classical Machine Learning (ML)

Classical machine learning covers two principal methods:

- Extraction: Here we will work with cumulants or high-order statistic values like skewness or kurtosis.
- Classification: Here, we expect to input this information to the ML supervised learning type model. In our case, we plan to use Support Vector Machines (SVMs) and Decision Trees (DT).

2.2 Deep Learning-Based AMC Methods

Deep learning methods for automatic modulation classification learn features directly from signal data. Instead of relying on manually designed statistical features, these models process raw I/Q samples and learn representations that are useful for distinguishing modulation classes. This is especially effective when large labelled datasets such as RadioML 2018.01A are available.[3, 11]

Convolutional neural networks are commonly used in AMC because they can extract local temporal patterns from I/Q sequences. In this project, CNN1D is used as a baseline convolutional model to evaluate how well local temporal features alone can classify modulation types. [5, 4]

Recurrent layers, especially LSTM layers, can be combined with convolutional layers to model longer temporal dependencies in the signal. The CNN-LSTM model used in this project follows this approach: convolutional layers first extract local signal features, and the LSTM layer then models the resulting feature sequence.[6]

Residual neural networks are another important deep learning approach. They use shortcut connections to support deeper architectures and improve gradient flow during training. In this project, ResNet1D is used to evaluate whether a deeper one-dimensional convolutional architecture improves AMC performance compared with CNN1D and CNN-LSTM. [5, 3]

Recent AMC research also investigates attention mechanisms and Transformer-based models. These approaches can model long-range dependencies more directly, but they usually require more computational resources. In this project, the focus is placed on CNN1D, CNN-LSTM, and ResNet1D because they represent three practical deep learning approaches: local convolutional feature extraction, recurrent sequence modelling, and deeper residual convolutional feature extraction.[8]

2.2.1 Advanced Machine Learning (ML)

Advanced machine learning methods for automatic modulation classification are primarily deep learning models. Unlike classical approaches, which often depend on manually designed features, deep learning models can learn useful representations directly from raw I/Q signal samples.[11, 4, 5]

In convolutional neural networks, filters are learned during training and are used to extract local patterns from the input signal. Recurrent layers, such as LSTM layers, can then be used to model temporal dependencies in the extracted feature sequence. During training, the model parameters are updated by backpropagation, so both feature extraction and classification are optimized together for the modulation classification task. [11, 4, 5]

2.3 Position of this Project

This project focuses on three deep learning architectures for automatic modulation classification:

- **CNN1D**, used as a convolutional baseline for direct classification of I/Q samples;
- **CNN-LSTM**, used to combine local convolutional feature extraction with recurrent temporal modelling;
- **ResNet1D**, used to evaluate the benefit of residual connections in a deeper one-dimensional convolutional architecture.

The goal is not to propose a new state-of-the-art architecture, but to implement and compare representative deep learning models for AMC under the same dataset split, training configuration, and evaluation procedure.

3 Objectives of the Project

The main objective of this project is to implement, train, evaluate, and compare deep learning models for automatic modulation classification of radio frequency signals using the RadioML 2018.01A dataset.

The partial objectives are:

- to review the theoretical background of modulation schemes, I/Q signal representation, and deep learning architectures for AMC;
- to analyse the structure of the RadioML 2018.01A dataset;
- to design a memory-efficient data loading pipeline based on lazy HDF5 reading and predefined train/validation/test split indices;
- to implement the CNN1D, CNN-LSTM, and ResNet1D architectures in PyTorch;
- to train and validate the models using the same dataset split and controlled experimental configuration;
- to evaluate model performance using accuracy, confusion matrices, precision, recall, F1-score, false positive rate, and per-SNR accuracy;
- to compare the models and discuss their suitability for automatic modulation classification.

4 Methodology

4.1 Dataset Pipeline

The RadioML 2018.01A dataset is stored as an HDF5 file of approximately 25 GB. Loading the complete signal array into RAM is not practical on standard hardware. Therefore, this project uses a two-stage data pipeline. First, split indices are created and stored. Second, samples are loaded from the HDF5 file only when needed during training or evaluation.

4.1.1 Dataset Splitting

Dataset splitting is the process of dividing a single dataset into two or more disjoint subsets to train and evaluate a machine learning model. Its primary goal is to ensure that a model can *generalise*, in other words, perform well on new or unseen data rather than simply memorising the specific examples it was trained on [9].

The Three Standard Subsets

In a supervised learning workflow, the dataset is typically partitioned into three parts, each serving a distinct purpose in the development pipeline.

Training set In our project (60 % of the data). This is the subset the model directly learns from. During training, the algorithm iterates over these examples, adjusting its internal parameters such as the weights of a neural network to minimise the chosen loss function.

Validation set (20 %). This subset is used during the development phase to tune *hyperparameters* (e.g. learning rate, number of layers, regularisation strength) and to monitor for overfitting. It acts as a proxy test that guides model selection without influencing the learned weights directly.

Test set (20 %). This subset is withheld entirely until training and model selection are complete. It provides a single, unbiased estimate of the model's performance on genuinely unseen data, representing the expected behaviour in a real deployment scenario. The test set must never be used to make any modelling decisions; otherwise the evaluation becomes optimistic and unreliable.

4.1.2 Stratified Train/Validation/Test Split

Stratification is performed over class and SNR combinations, resulting in **24 classes $\times$ 26 SNR values = 624 stratification keys**. This ensures that the training, validation, and test sets contain proportional samples from every modulation class at every SNR level. Table 4.1 shows the number of samples assigned to each subset. This is also thanks to the balanced RadioML2018.A01 dataset, since it has the same number of samples (frames) for each group of class-SNR.

A fixed random seed of 42 is used when shuffling the indices inside each stratification group. This ensures reproducibility of the split and allows all models to be trained and evaluated on the same samples.

Tab. 4.1: Baseline number of samples in the stratified RadioML split.

Split	**Samples per class-SNR group**	**Total samples**
Training	2458	1 533 792
Validation	819	511 056
Testing	819	511 056
Total	4096	2 555 904

Another important feature is that the split is stratified using a key that represents each unique (c, γ) combination, where c is the modulation class and γ is the SNR value. Samples with the same stratification key belong to the same modulation SNR group. Within each group, the indices are shuffled using a fixed random seed of 42, which makes the split reproducible. The shuffled indices are then divided according to the selected train, validation, and test ratios.

A stratify key is computed as:

$$k = c \cdot N_{\text{SNR}} + \text{snr\_code}, \tag{4.1}$$

where c is the modulation class ID, $N_{\text{SNR}} = 26$ is the number of SNR levels, and snr_code is an integer identifier assigned to each unique SNR value in dB.

Tab. 4.2: Example stratification key values using $N_{\text{SNR}} = 26$.

Modulation	$\mathbf{c_i}$	**SNR**	$\mathbf{snr_c}$	$\mathbf{k_i = c_i \cdot 26 + snr_c}$
OOK	0	-20 dB	0	0
OOK	0	-18 dB	1	1
BPSK	3	-20 dB	0	78
BPSK	3	20 dB	20	98

4.1.3 PyTorch Dataset Class

The `RadioMLDataset` class is a custom PyTorch dataset used to load samples from the RadioML 2018.01A HDF5 file. It extends `torch.utils.data.Dataset`, which allows it to be used directly with a PyTorch `DataLoader`. The purpose of this class is to provide memory-efficient access to the dataset without loading the full HDF5 file into RAM.

Instead of storing all signal samples in memory, the class stores only:

- the path to the HDF5 dataset file;
- the selected row indices for the current split;
- a flag indicating whether sample-level normalization should be applied.

This design is important because the RadioML dataset is very large. Loading the complete signal array at once would require a large amount of memory. Therefore, the dataset uses lazy loading: each sample is read from the HDF5 file only when it is requested during training, validation, or testing.

Initialization

The constructor `__init__` receives the HDF5 file path, the indices belonging to a specific split, and an optional normalization flag:

- `hdf5_path` specifies where the RadioML HDF5 file is stored;
- `indices` contains the row numbers assigned to the current split;
- `sample_normalize` controls whether each signal sample should be normalized individually.

The class does not immediately open the HDF5 file during initialization. Instead, the file handle is set to `None` and opened later only when data is first accessed.

Dataset Length

The `__len__` method returns the number of samples available in the current dataset split:

$$\mathbb{L} = \texttt{len(indices)} \tag{4.2}$$

For example, if the object is created using the training indices, then `__len__` returns the number of training samples.

Lazy HDF5 File Access

The `file` property is responsible for opening the HDF5 file lazily. If the file has not been opened yet, it creates an HDF5 file handle using `h5py.File`. If the file is already open, the existing handle is reused.

This approach avoids repeatedly opening and closing the HDF5 file for every sample. It is also suitable for PyTorch data loading, where each worker process can create its own file handle when needed.

Reading One Sample

The `__getitem__` method defines how one sample is retrieved. It receives a local dataset index and maps it to the corresponding row in the original HDF5 file:

1. The local index is converted to the real HDF5 row index using `self.indices[idx]`.
2. The signal sample `X[sample_idx]` is read from the HDF5 file.
3. The signal is converted to `float32`.
4. The signal is transposed from shape $(1024, 2)$ to $(2, 1024)$.
5. If enabled, sample-level normalization is applied.
6. The one-hot encoded label `Y[sample_idx]` is read.
7. The modulation class is obtained using `argmax`.
8. The SNR value is read from `Z[sample_idx]`.
9. The signal, class label, and SNR value are returned as PyTorch tensors.

The transposition step is necessary because PyTorch `Conv1d` layers expect input in the format:

$$(\text{channels}, \text{sequence length}) \tag{4.3}$$

In this project, the two channels correspond to the in-phase and quadrature components, while the sequence length is 1024. Therefore, each sample is returned with shape $(2, 1024)$.

Optional Sample Normalization

If `sample_normalize` is enabled, the dataset normalizes each signal independently:

$$x_{\text{norm}} = \frac{x - \mu}{\sigma} \tag{4.4}$$

where μ is the mean of the sample and σ is its standard deviation. This can help make the input scale more consistent. The implementation checks if the standard deviation is greater than zero before applying normalization, preventing division by zero.

Returned Values

Each call to `__getitem__` returns a tuple:

$$(x, y, \text{snr}) \tag{4.5}$$

where:

- `x` is a PyTorch tensor containing the I/Q signal with shape $(2, 1024)$;
- `y` is a `long` tensor containing the integer modulation class;
- `snr` is a `float32` tensor containing the signal-to-noise ratio.

Returning the SNR together with the signal and label is useful because it allows later evaluation of model performance at different noise levels.

Closing the File

The `close` method closes the HDF5 file handle if it is open. The destructor `__del__` also calls `close` when the dataset object is destroyed. This helps release system resources and prevents open file handles from remaining after training or evaluation finishes.

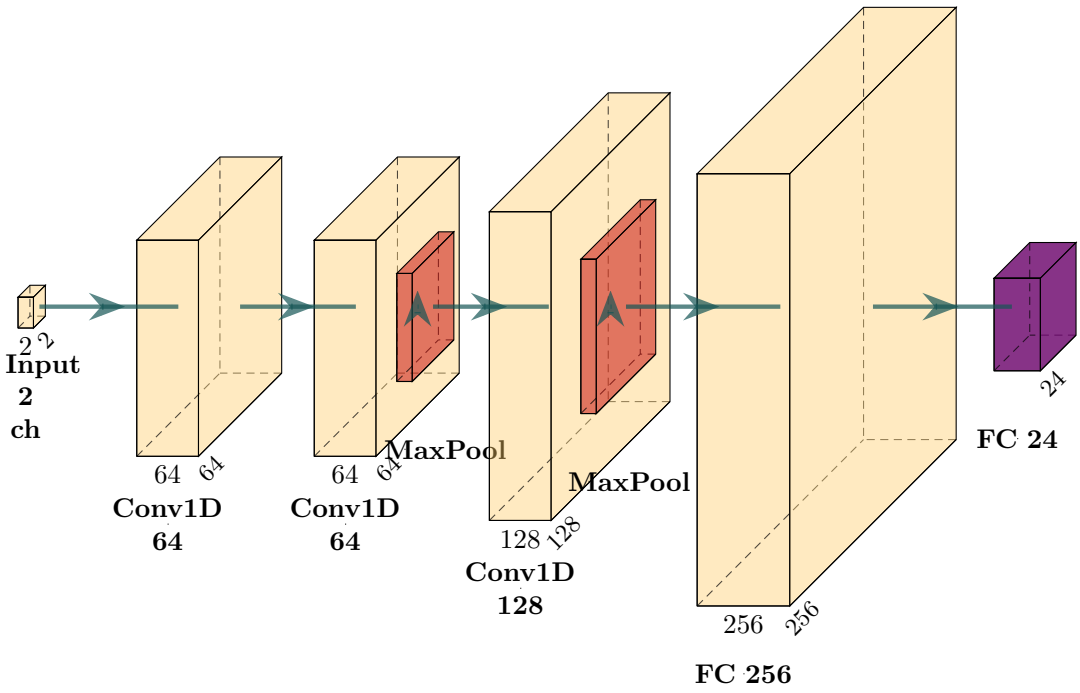
Overall, the `RadioMLDataset` class provides an efficient bridge between the large HDF5-based RadioML dataset and the PyTorch training pipeline. It keeps memory usage low, supports predefined train/validation/test splits, formats the data correctly for one-dimensional convolutional models, and returns both labels and SNR values for classification and per-SNR evaluation.

4.2 Model Architectures

4.2.1 CNN1D Architecture

The CNN1D model is used as the simplest baseline architecture in this project. It applies one-dimensional convolutional layers directly to the I/Q input signal, where the two input channels correspond to the in-phase and quadrature components. The convolutional layers extract local temporal patterns from the raw signal, while max pooling reduces the temporal resolution of the feature maps. [4, 9]

After convolutional feature extraction, adaptive average pooling compresses the temporal dimension into a fixed-size feature vector. This vector is passed through fully connected layers to produce one output logit for each of the 24 modulation classes. The CNN1D model is useful as a baseline because it evaluates how well local convolutional features alone can classify modulation types without recurrent layers or residual connections. [4, 9]

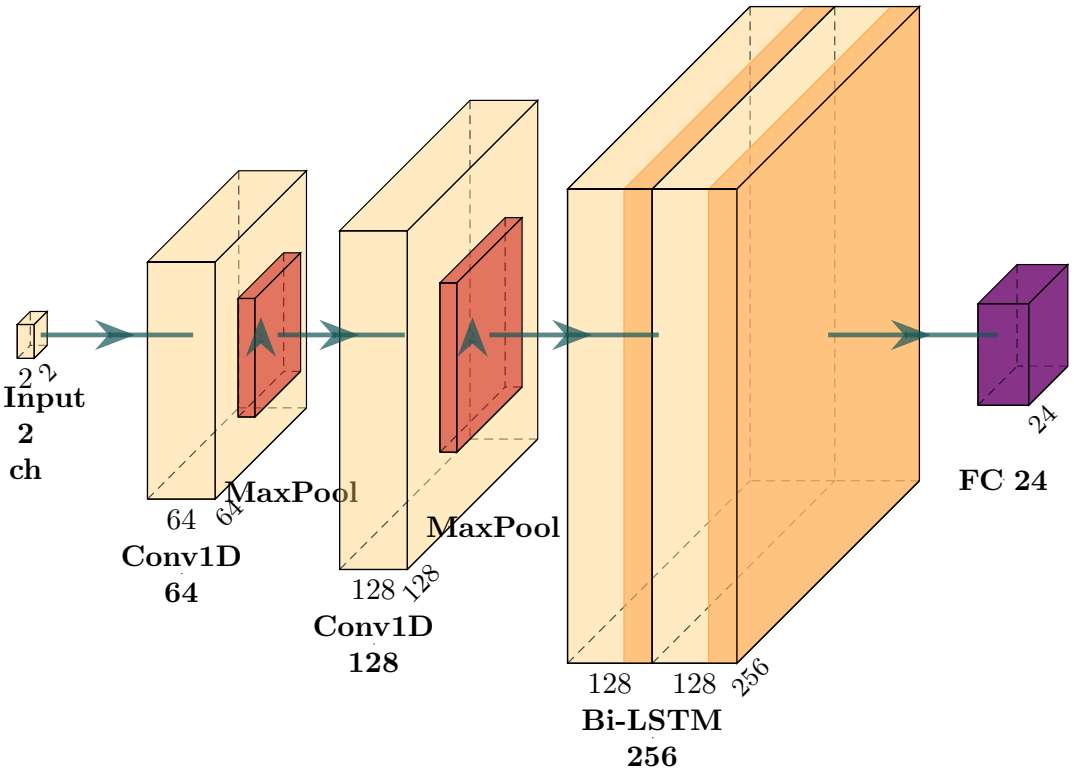


Obr. 4.1: Architecture of the CNN1D model.

4.2.2 CNN-LSTM Architecture

The CNN-LSTM model combines convolutional feature extraction with recurrent sequence modelling. First, one-dimensional convolutional layers extract local features from the I/Q signal and reduce the temporal length using pooling. The resulting feature map is then transposed into a sequence representation and passed to an LSTM layer.

In this project, the CNN-LSTM model uses a bidirectional LSTM, allowing the model to process the extracted feature sequence in both forward and backward directions. The LSTM outputs are averaged across the sequence and passed through a final classification layer. This architecture evaluates whether recurrent temporal modelling improves performance compared with a purely convolutional baseline.

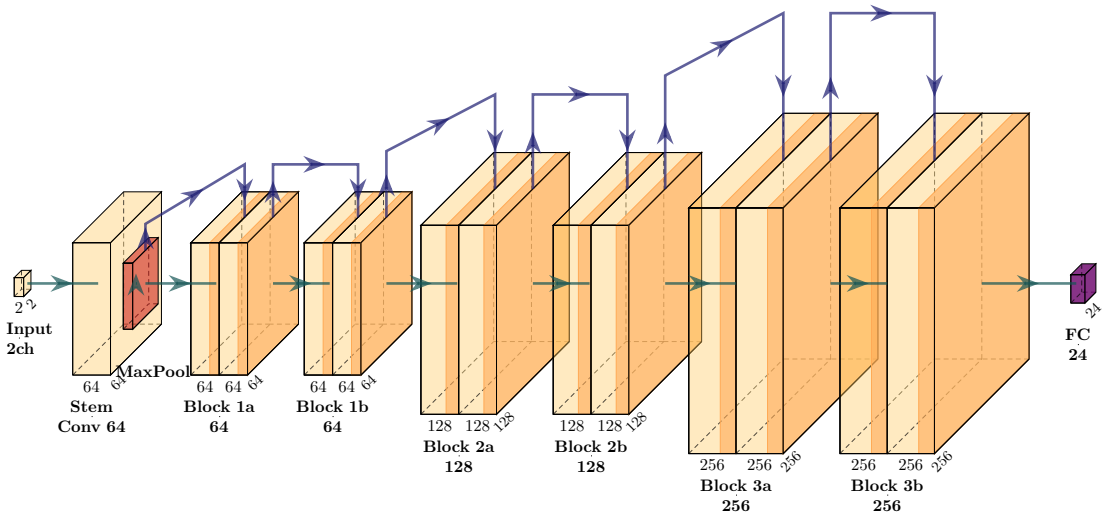


Obr. 4.2: Architecture of the CNN-LSTM model.

4.2.3 ResNet1D Architecture

The ResNet1D model adapts residual learning to one-dimensional I/Q signal classification. It begins with a convolutional stem that reduces the temporal resolution of the input signal and increases the number of feature channels. After the stem, the model applies several residual stages composed of residual blocks.

Each residual block contains two one-dimensional convolutional layers with batch normalisation, ReLU activation, and dropout. The output of these layers is added to a shortcut connection. When the number of channels or the temporal resolution changes, a projection shortcut is used to match the dimensions. These residual connections help improve gradient flow and allow a deeper convolutional network to be trained more stably.



Obr. 4.3: Architecture of the ResNet1D model.

4.3 Training Pipeline

4.3.1 Configuration-Driven Experiments

All experiments are controlled through YAML configuration files. These files define the model architecture, training hyperparameters, dataset paths, split file, random seed, and output directory. This avoids hardcoded values in the source code and makes experiments easier to reproduce and modify.

At the beginning of training, the selected YAML file is loaded, and optional command-line overrides can be applied, such as changing the number of epochs, batch size, early stopping patience, or the maximum number of samples used for quick experiments. The final configuration used for the run is saved as `effective_config.json` in the experiment directory.

For the baseline experiments, CNN1D, CNN-LSTM, and ResNet1D use the same main training settings and the same stratified train/validation/test split. This ensures that the models are compared under consistent data and optimization conditions.

Tab. 4.3: Shared baseline training hyperparameters.

Hyperparameter	Value
Batch size	512
Maximum epochs	50
Learning rate	0.001
Optimizer	Adam
Weight decay	0.0
Dropout	0.4
Early stopping patience	10 epochs
Random seed	42
Sample normalization	Disabled

Although the main training settings are shared, each model still has architecture-specific parameters, such as convolutional filters, kernel sizes, LSTM hidden units, and residual block structure. These parameters are also defined in the YAML configuration files and can be modified without changing the source code.

4.3.2 Training Procedure

The training script first builds the dataloaders from the saved split file. The training dataloader shuffles samples during training, while the validation and test dataloaders keep a fixed order. The model is then created using a model factory, which selects the correct architecture based on the `type` field in the configuration file.

During each epoch, the model is trained on the training set and then evaluated on the validation set. For training batches, gradients are enabled, the loss is backpropagated, and the optimizer updates the model parameters. For validation batches, gradients are disabled, so the model is only evaluated. For both phases, the average loss and classification accuracy are recorded.

4.3.3 Loss Function and Optimisation

Cross-entropy loss is used for all experiments:

$$\mathscr{L} = -\sum_{c=1}^{24} y_c \log \hat{p}_c, \tag{4.6}$$

where y_c is the ground-truth class indicator and $\hat{p}_c$ is the predicted probability for class c. The Adam optimizer is used with an initial learning rate of 10^{-3} and no weight decay.

4.3.4 Early Stopping and Checkpointing

Validation accuracy is monitored after every epoch. If the validation accuracy improves, the current model is saved as the best checkpoint. The checkpoint contains the epoch number, model weights, optimizer state, best validation accuracy, best validation loss, and the configuration used for the run.

If validation accuracy does not improve for the number of epochs specified by the early stopping patience, training stops early. In addition, the training history is saved after each epoch in `history.csv`, and a summary of the run is saved in `train_report.json`.

4.3.5 Reproducibility

The same random seed is used for Python, NumPy, and PyTorch. In addition, all baseline models reuse the same saved stratified split file. Therefore, the baseline experiments use the same training, validation, and test samples, allowing the models to be compared fairly. The test set is not used during training; it is reserved for final evaluation after training is complete.

4.4 Performance Metrics

The models are evaluated on the test set using:

- overall accuracy;
- per-class accuracy;
- per-SNR accuracy;
- confusion matrix;
- precision, recall, and F1-score.

4.5 Software Implementation

The project is implemented in Python using PyTorch. The main project files are shown in Table 4.4.

Tab. 4.4: Key project files and their responsibilities.

File	Responsibility
`src/dataset.py`	HDF5 reading, split management, DataLoader creation
`src/train.py`	Training loop, early stopping, checkpoint saving
`src/evaluate.py`	Test inference, metrics, report generation
`src/models/cnn1d.py`	CNN1D architecture
`src/models/cnn_lstm.py`	CNN-LSTM architecture
`src/models/resnet1d.py`	ResNet1D architecture
`src/models/factory.py`	Model creation from config
`src/utils.py`	Config loading, seeding, device selection
`configs/*.yaml`	Experiment hyperparameter definitions

5 Results and Discussion

5.1 Experimental Setup

Experiments are performed using the stratified 60/20/20 split with random seed 42. The training set contains 1 533 792 samples, the validation set contains 511 056 samples, and the test set contains 511 056 samples. The split is stratified by both modulation class and SNR value as described earlier in Section 4.1.2.

All three models are trained using the same experimental setup: **batch size 512, maximum 50 epochs, Adam optimizer, learning rate 0.001, dropout 0.4, and early stopping patience of 10 epochs.** The final reported metrics are computed on the test set.

5.1.1 Metrics Graphical Representation

The MATLAB visualization script is used after model evaluation to generate graphical summaries of the experiment results. It reads the files produced by the Python training and evaluation pipeline.

The script uses the following experiment outputs:

- `history.csv`, which contains training and validation loss and accuracy for each epoch;
- `train_report.json`, which stores the best epoch and validation performance;
- `test_report.json`, which contains the test accuracy, precision, recall, F1-score, per-class metrics, and per-SNR accuracy;
- `confusion_matrix.csv`, which stores the final confusion matrix on the test set.

Using these files, the script generates learning curves, test-set summary metrics, accuracy versus SNR plots, per-class performance charts, confusion matrices, prediction bias plots, and SNR regime comparisons. The figures are exported as PNG files and saved in a visualization directory for use in the final report.

5.2 Classification Performance

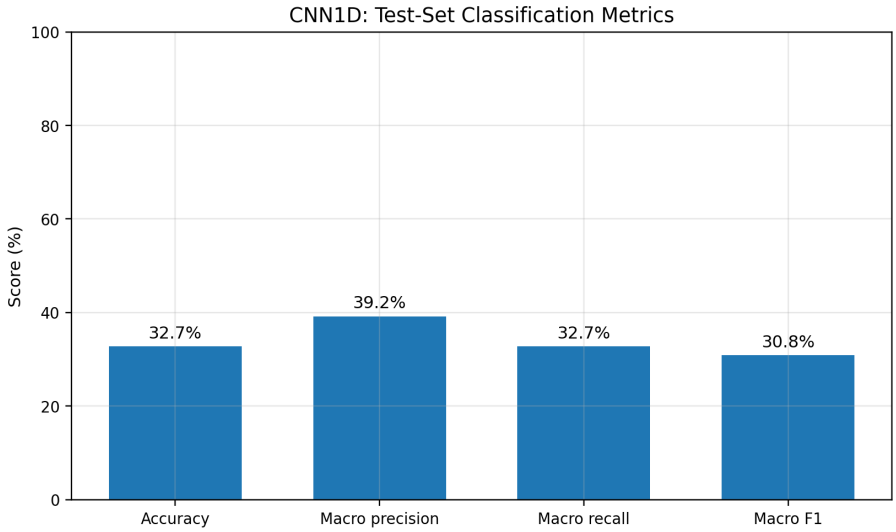
Table 5.1 shows the test-set performance achieved by each architecture. The comparison is performed on the same test split, which contains 511 056 samples.

Tab. 5.1: Overall test performance for each model.

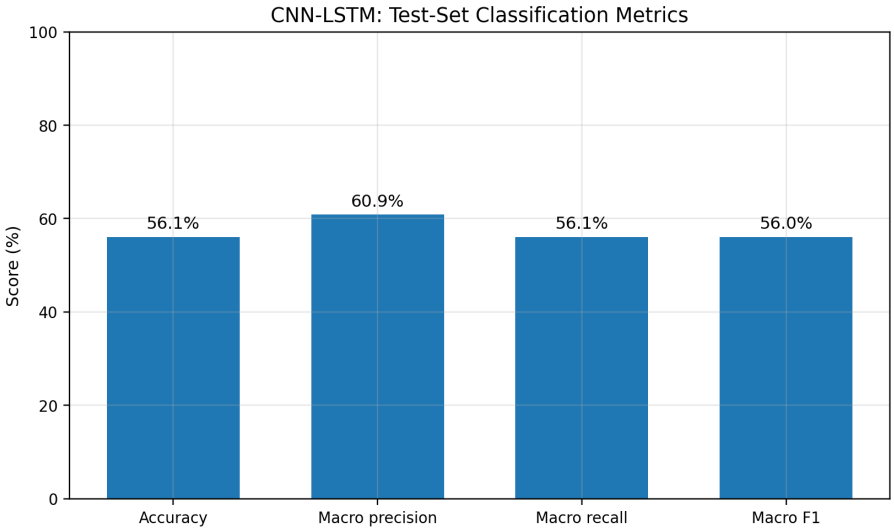
Model	Parameters	Accuracy	Precision	Recall	F1-score
CNN1D	77 208	32.73%	39.17%	32.73%	30.85%
CNN-LSTM	313 048	56.06%	60.85%	56.06%	56.04%
ResNet1D	961 816	58.98%	66.71%	58.98%	59.42%

5.2.1 Test-set Classification Metrics

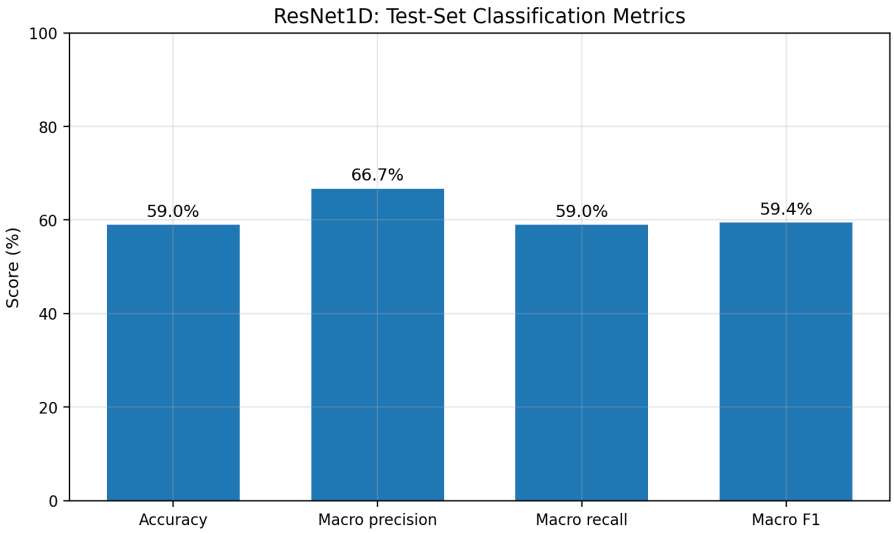
For the figures 5.1, 5.2, and 5.3, we stay they are macro-metrics, which means taking the average of each-class measurement; this term is used because we are applying multiclass classification.



Obr. 5.1: Summary test-set metrics for CNN1D.



Obr. 5.2: Summary test-set metrics for CNN-LSTM.



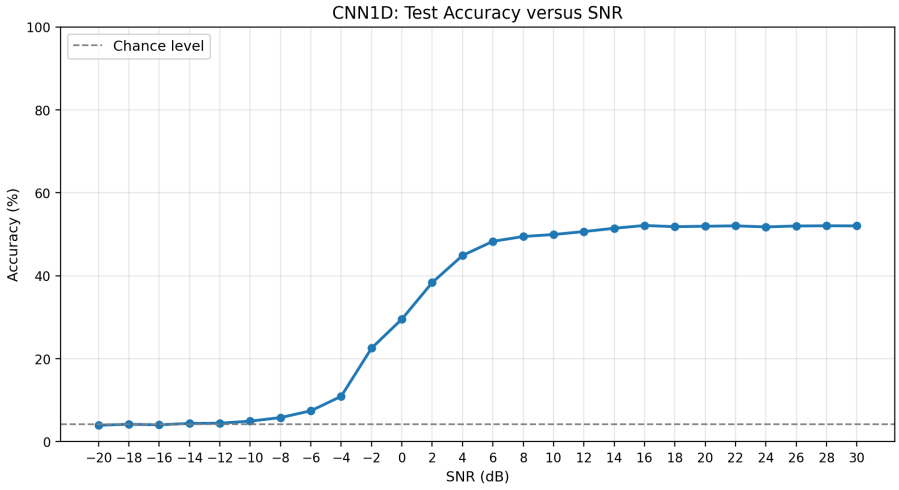
Obr. 5.3: Summary test-set metrics for ResNet1D.

5.3 Per-SNR Accuracy

Per-SNR accuracy is used to analyse how classification performance changes with channel quality. At low SNR values, especially below 0 dB, the signal is strongly affected by noise and classification becomes difficult. As SNR increases, the signal structure becomes clearer, and classification accuracy is expected to improve. Figures 5.4,5.5,5.6 shows the test accuracy as a function of SNR for the selected experiment. The curve confirms that classification performance is strongly dependent on channel quality.

5.3.1 CNN1D accuracy with respect to SNR.

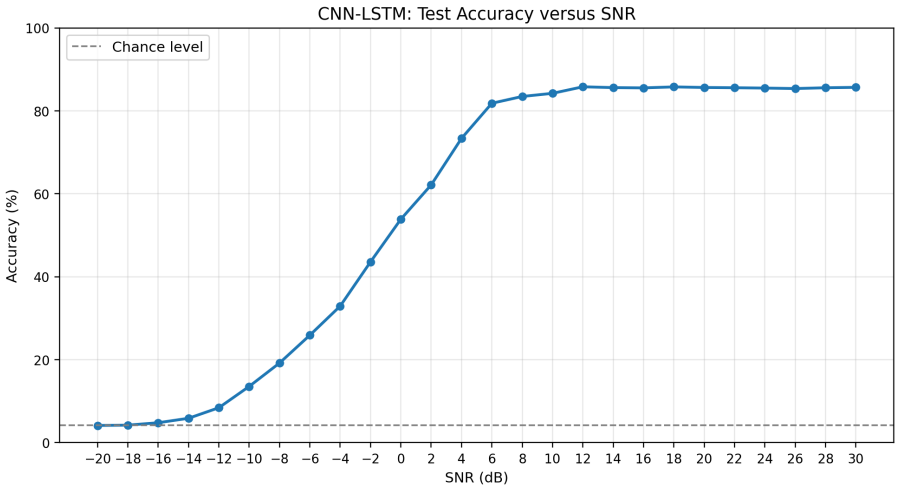
Figure 5.4 shows that CNN1D performs poorly at very low SNR values, where the accuracy is close to the chance level. At -20 dB the accuracy is approximately 3.96%, and at -10 dB it remains low at about 4.94%. The performance starts to improve around 0 dB, reaching 29.54%. At higher SNR values the curve increases further, but it saturates at around 50-52%, showing that the simple CNN1D model is limited even when the signal quality is high.



Obr. 5.4: Per-SNR test accuracy for CNN1D.

5.3.2 CNN-LSTM accuracy with respect to SNR.

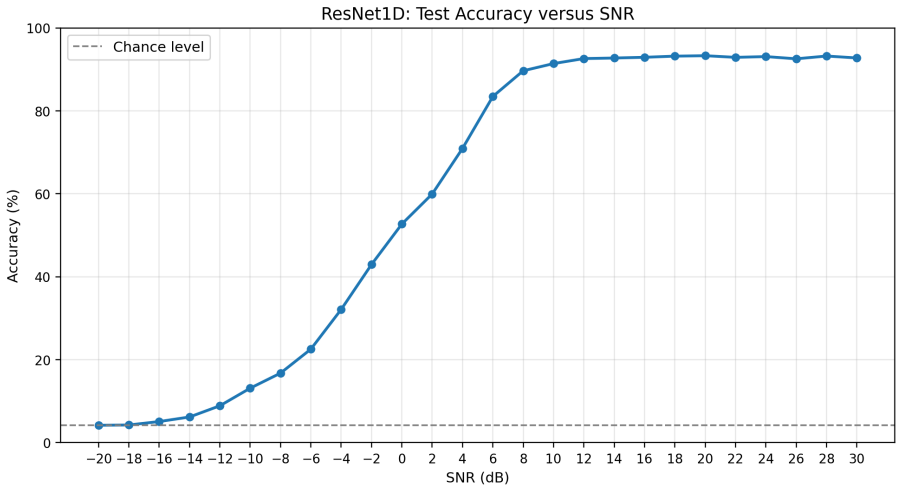
Figure 5.5 shows a stronger dependence on SNR. At very low SNR values, the model still performs close to chance level, with 4.14% accuracy at -20 dB. However, the accuracy increases more rapidly than for CNN1D as the SNR improves. At 0 dB, CNN-LSTM reaches 53.87%, and at 10 dB it reaches 84.22%. The accuracy then stabilizes around 85–86% at high SNR values. This indicates that the LSTM component helps the model use temporal information more effectively when the signal structure becomes clearer.



Obr. 5.5: Per-SNR test accuracy for CNN-LSTM.

5.3.3 ResNet1D accuracy with respect to SNR.

Figure 5.6 shows that ResNet1D reaches the best high-SNR performance among the three models. At -20 dB, the accuracy is approximately 4.19%, because the signal is strongly dominated by noise, which was explained in . The accuracy improves significantly as SNR increases, reaching 52.71% at 0 dB and 91.39% at 10 dB. At high SNR values, the model achieves around 93% accuracy. This confirms that the deeper residual architecture is able to extract more discriminative features when the signal quality is sufficient.



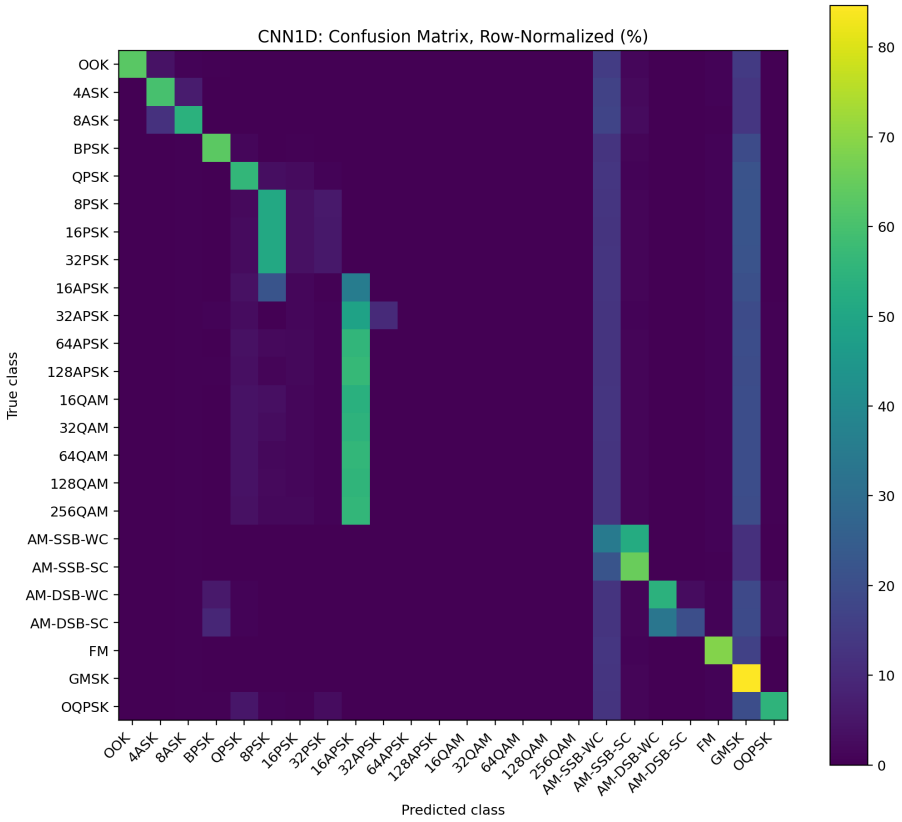
Obr. 5.6: Per-SNR test accuracy for ResNet1D.

5.4 Confusion Matrix Analysis

The confusion matrix shows which modulation classes are most frequently confused. Common confusion pairs are between higher-order QAM variants, higher-order PSK variants, and analogue modulation types with similar signal characteristics. Figures 5.7, 5.8, 5.9 show the row-normalized confusion matrices for CNN1D, CNN-LSTM, and ResNet1D. Row normalization makes the matrices easier to compare because each row represents the prediction distribution for one true modulation class. The row-normalized confusion matrices show that the models do not make errors uniformly across all modulation classes. Some modulation types are classified more reliably, while others are frequently confused with related classes.

5.4.1 Confusion Matrix - CNN1D

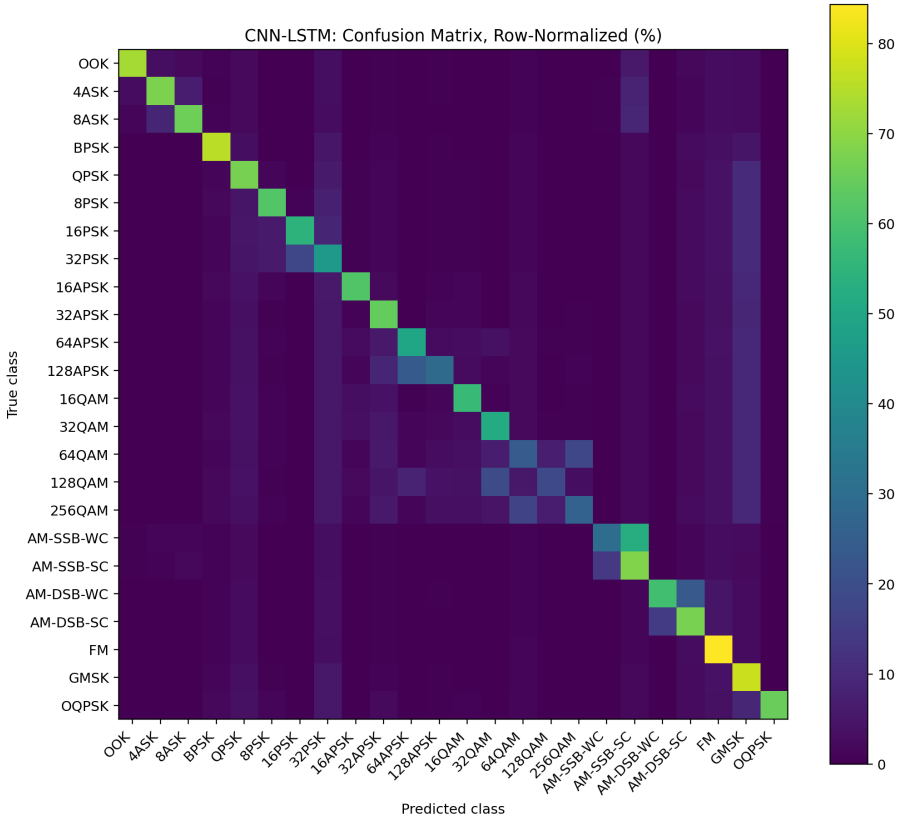
The best detected classes are GMSK, FM, AM-SSB-SC, BPSK, and OOK. However, the model performs very poorly on several high-order QAM classes, including 16QAM, 32QAM, 64QAM, 128QAM, and 256QAM. These classes are often confused with APSK-like classes, which indicates that the simple CNN1D model is not strong enough to separate more complex constellation structures.



Obr. 5.7: Row-normalized confusion matrix for CNN1D on the test set.

5.4.2 Confusion Matrix - CNN-LSTM

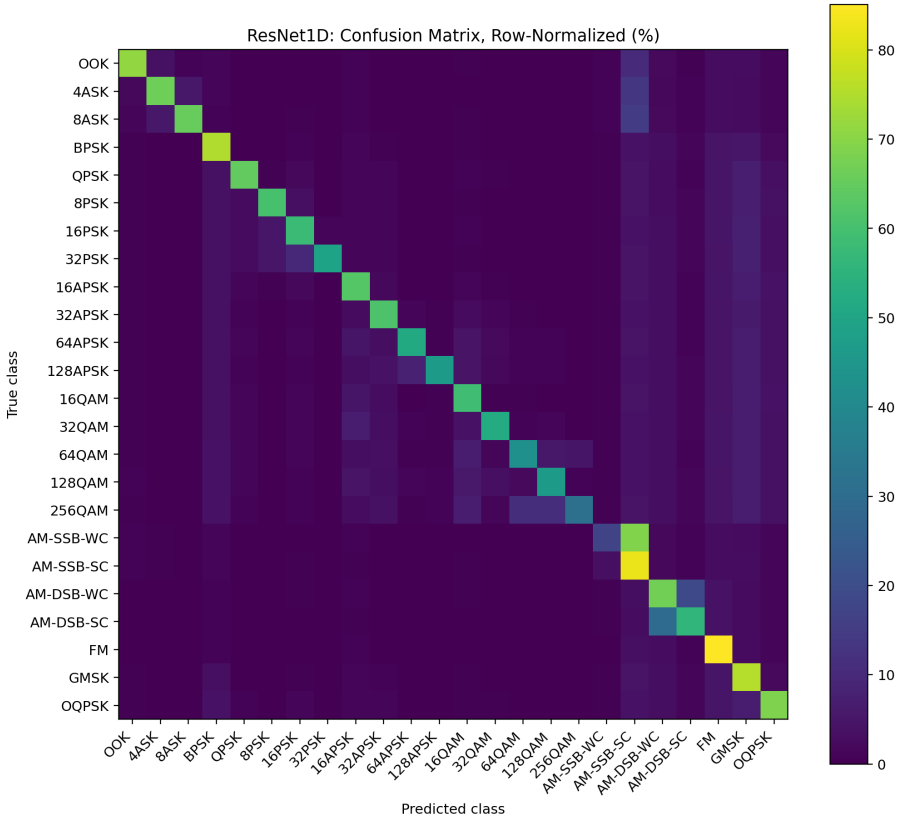
For CNN-LSTM, the best detected classes are FM, GMSK, BPSK, OOK, and AM-SSB-SC. Compared with CNN1D, the CNN-LSTM model improves classification of many digital modulation classes. However, it still struggles with high-order QAM and APSK variants, especially 128QAM, 64QAM, 256QAM, and 128APSK.



Obr. 5.8: Row-normalized confusion matrix for CNN-LSTM on the test set.

5.4.3 Confusion Matrix - Resnet

For ResNet1D, the best detected classes are FM, AM-SSB-SC, GMSK, BPSK, and OOK. This model gives the best overall performance among the three architectures. Nevertheless, it still shows confusion between analogue modulation variants, especially AM-SSB-WC and AM-SSB-SC, and between high-order QAM classes such as 256QAM, 128QAM, and 64QAM.



Obr. 5.9: Row-normalized confusion matrix for ResNet1D on the test set.

5.5 Discussion

The results show a clear performance difference between the three evaluated architectures. CNN1D achieved the lowest overall test accuracy, with 32.73% accuracy and a macro F1-score of 30.85%. This indicates that local convolutional features alone are not sufficient for robust classification of all 24 modulation classes in the RadioML 2018.01A dataset. CNN-LSTM improved the performance substantially, reaching 56.06% accuracy and a macro F1-score of 56.04%. This improvement suggests that temporal modelling is useful for AMC, because modulation patterns are not only local but also depend on how the I/Q sequence evolves. The bidirectional LSTM layer allows the model to use information from both directions of the extracted feature sequence.

ResNet1D achieved the best overall result, with 58.98% accuracy and a macro. F1-score of 59.42%. Although it has the highest number of trainable parameters, its residual connections allow a deeper convolutional network to be trained more stably. This suggests that deeper hierarchical feature extraction is beneficial for this classification task.

The per-class results show that some modulation classes are easier to detect than others. Across the models, FM, GMSK, BPSK, OOK, and AM-SSB-SC are among the best detected classes. In contrast, high-order QAM classes such as 64QAM, 128QAM, and 256QAM are more difficult, especially for CNN1D and CNN-LSTM. These modulations have more complex constellation structures and are more sensitive to noise, which makes them harder to separate.

The confusion matrices also show repeated confusion between modulation types with similar characteristics. For example, AM-SSB-WC is frequently confused with AM-SSB-SC, and high-order QAM classes are confused with each other. This confirms that errors are mainly concentrated between modulation families with similar signal structures rather than being randomly distributed across all classes.

Finally, the per-SNR results confirm that SNR is one of the most important factors affecting AMC performance. At low SNR values, the signal is dominated by noise and classification becomes difficult for all models. At higher SNR values, the modulation-specific structure becomes clearer, and the models achieve substantially better classification accuracy.

Annexes

List of annexes

1. Project directory structure
2. Key source code listings

Annex A – Project Directory Structure

Listing 1: Project directory structure used for the AMC experiments.

```
projekt-z-IKT/
|-- src/
|   |-- train.py
|   |-- evaluate.py
|   |-- compare_models.py
|   |-- dataset.py
|   |-- utils.py
|   `-- models/
|       |-- factory.py
|       |-- cnn1d.py
|       |-- cnn_lstm.py
|       `-- resnet1d.py
|-- configs/
|   |-- cnn_baseline.yaml
|   |-- cnn_lstm_baseline.yaml
|   `-- resnet1d_baseline.yaml
|-- data/
|   |-- raw/
|   |   `-- radioml2018/
|   |       `-- GOLD_XYZ_OSC.0001_1024.hdf5
|   `-- splits/
|       `-- radioml2018a_seed42_60_20_20.npz
|-- experiments/
|   |-- cnn1d_baseline_20260511_180658-BS512/
|   |-- cnn_lstm_baseline_20260512_062159-BS512/
|   |-- resnet1d_baseline_20260513_004424-BS512/
|   |-- model_kpis_all_runs.csv
|   |-- model_kpis_all_runs.md
|   |-- model_kpis_best_per_model.csv
|   `-- model_kpis_best_per_model.md
|-- visualizations/
|   |-- current_experiment/
|   |   |-- 01_learning_curves.png
|   |   |-- 02_summary_metrics.png
|   |   |-- 03_accuracy_vs_snr.png
|   |   |-- 04_per_class_metrics.png
|   |   |-- 05_confusion_matrix_counts.png
|   |   |-- 06_confusion_matrix_normalized.png
|   |   |-- 07_prediction_bias.png
|   |   `-- 08_snr_regime_accuracy.png
|   |-- models/
```

```
|   |   |-- cnn1d.pdf
|   |   |-- cnnlstm.pdf
|   |   `-- resnet1d.pdf
|   `-- plot_current_experiment.m
|-- requirements.txt
|-- Dockerfile
|-- docker-compose.yml
`-- README.md
```

Annex B – Key Source Code Excerpts

Listing 2: RadioMLDataset.__getitem__ – lazy HDF5 reading.

```
def __getitem__(self, idx: int) -> tuple[torch.Tensor, torch.Tensor, torch.T
    sample_idx = int(self.indices[idx])

    x = self.file["X"][sample_idx].astype(np.float32).T
    if self.sample_normalize:
        std = x.std()
        if std > 0:
            x = (x - x.mean()) / std

    y = int(np.argmax(self.file["Y"][sample_idx]))
    snr = float(np.asarray(self.file["Z"][sample_idx]).reshape(-1)[0])

    return (
        torch.from_numpy(x),
        torch.tensor(y, dtype=torch.long),tensor(snr, dtype=torch.float32),
    )
```

Listing 3: BasicBlock1D – residual block used in ResNet1D.

```
class BasicBlock1D(nn.Module):
    expansion = 1

    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        stride: int = 1,
        dropout: float = 0.1,
    ) -> None:
        super().__init__()

        self.conv1 = nn.Conv1d(
            in_channels,
            out_channels,
            kernel_size=3,
            stride=stride,
            padding=1,
            bias=False,
        )
        self.bn1 = nn.BatchNorm1d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(dropout)
```

```
        self.conv2 = nn.Conv1d(
            out_channels,
            out_channels,
            kernel_size=3,
            padding=1,
            bias=False,
        )
        self.bn2 = nn.BatchNorm1d(out_channels)

        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv1d(
                    in_channels,
                    out_channels,
                    kernel_size=1,
                    stride=stride,
                    bias=False,
                ),
                nn.BatchNorm1d(out_channels),
            )
        else:
            self.shortcut = nn.Identity()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        identity = self.shortcut(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.conv2(out)
        out = self.bn2(out)

        out = out + identity
        return self.relu(out)
```

Conclusion

This project presented a deep learning framework for Automatic Modulation Classification of RF signals using the RadioML 2018.01A dataset. Three neural network architectures, CNN1D, CNN-LSTM, and ResNet1D, were designed, implemented, and evaluated using a reproducible experimental pipeline.

The main contributions of this work are:

- a memory-efficient dataset pipeline based on lazy HDF5 reading;
- stratified train/validation/test splitting by modulation class and SNR;
- implementation of three deep learning architectures for AMC;
- configuration-driven training and evaluation;
- evaluation using accuracy, confusion matrices, and per-SNR metrics.

The results show that deep learning models can learn useful representations directly from raw I/Q samples. However, performance remains strongly dependent on SNR. Future work could include attention-based architectures, transformer models, improved signal normalisation, data augmentation, and model compression for deployment on embedded radio hardware.

References

[1] Rakshit Chennagiri, Sahil Sehgal, and Yerram Ravinder. A survey on automatic modulation classification techniques. In *2024 Intelligent Systems and Machine Learning Conference (ISML)*, pages 94–100, 2024. doi: 10.1109/ISML60050.2024.11007395.

[2] Xiao Hu, Mingju Chen, Xingyue Zhang, Jie Rao, Senyuan Li, and Xiaofei Song. Mfca-transformer: Modulation signal recognition based on multidimensional feature fusion. *Sensors*, 25(16):5061, 2025. doi: 10.3390/s25165061.

[3] Thien Huynh-The, Quoc-Viet Pham, Toan-Van Nguyen, Thanh T. Nguyen, Rukhsana Ruby, Ming Zeng, and Dong-Seong Kim. Automatic modulation classification: A deep architecture survey. *IEEE Access*, 9:142950–142971, 2021. doi: 10.1109/ACCESS.2021.3120419.

[4] Arun Kumar, Nishant Gaur, Sumit Chakravarty, Mohammed H. Alsharif, Peerapong Uthansakul, and Monthippa Uthansakul. Analysis of spectrum sensing using deep learning algorithms: Cnns and rnns. *Ain Shams Engineering Journal*, 15(3):102505, 2024. doi: 10.1016/j.asej.2023.102505.

[5] Xiaoyu Liu, Diyu Yang, and Aly El Gamal. Deep neural network architectures for modulation classification. *arXiv preprint arXiv:1712.00443*, 2017.

[6] Timothy J. O'Shea, Tamoghna Roy, and T. Charles Clancy. Over-the-air deep learning based radio signal classification. *IEEE Journal of Selected Topics in Signal Processing*, 12(1): 168–179, 2018. doi: 10.1109/JSTSP.2018.2797022.

[7] John G. Proakis and Masoud Salehi. *Digital Communications*. McGraw-Hill Higher Education, New York, 5th edition, 2008. ISBN 978-0072957167.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[9] Mengtao Wang, Youchen Fan, Shengliang Fang, Tianshu Cui, and Donghang Cheng. A joint automatic modulation classification scheme in spatial cognitive communication. *Sensors*, 22 (17):6500, 2022. doi: 10.3390/s22176500.

[10] Jiawei Zhang, Tiantian Wang, Zhixi Feng, and Shuyuan Yang. Amc-net: An effective network for automatic modulation classification. *arXiv preprint arXiv:2304.00445*, 2023. doi: 10.48550/arXiv.2304.00445.

[11] Qinghe Zheng, Xinyu Tian, Lisu Yu, Abdussalam Elhanashi, and Sergio Saponara. Recent advances in automatic modulation classification technology: Methods, results, and prospects. *International Journal of Intelligent Systems*, 2025:4067323, 2025. doi: 10.1155/int/4067323.