technische universität
dortmund

# Developing a General Neural Network Architecture for Solving Multi-Class Classification Problems in Imbalanced Multivariate Data

**Malyarchuk, Vladislav**

Masterarbeit

Studiengang: Master Informatik
Matrikelnummer: 222742
Erstgutachter: Prof. Dr. Christian Janiesch
Zweitgutachter: Dr. Philip Stahmann

Bearbeitungszeit: 06.11.2024 – 14.05.2025

Lehrstuhl für Enterprise Computing
Fakultät für Informatik

# Zusammenfassung

In vielen praxisnahen Geschäftsanwendungen sind Daten von Natur aus unausgewogen: Bestimmte Klassen treten wesentlich seltener auf als andere, was konventionelle Machine-Learning-Modelle vor erhebliche Herausforderungen stellt. Diese Modelle zeigen häufig eine schlechtere Leistung bei unterrepräsentierten Klassen, gerade jenen, die für Entscheidungen in kritischen Anwendungsgebieten wie Cybersicherheit, medizinischer Diagnostik, Fehlererkennung und anderen besonders entscheidend sein können.

Diese Arbeit befasst sich mit dem Problem der Multiklassenklassifikation auf unausgewogenen strukturierten Datensätzen, indem eine allgemeine neuronale Netzwerkarchitektur entwickelt wird, die ohne domänenspezifische Anpassung in unterschiedlichsten Anwendungsfällen einsetzbar ist. Die vorgeschlagene Architektur wird mit anderen gängigen, modernen Ansätzen verglichen und anhand von vier öffentlich verfügbaren Datensätzen (KDD-99, Darknet, Shuttle und Covertype) evaluiert, die jeweils eine einzigartige Kombination aus Dimensionalität, Umfang und Grad der Unausgewogenheit darstellen.

Die Ergebnisse zeigen, dass bestimmte architektonische Entscheidungen (darunter auch die in dieser Arbeit eingeführten "Random Mixing Layers") eine starke Leistung auf strukturierten unausgewogenen Datensätzen ermöglichen, während gleichzeitig die Recheneffizienz auf gängiger Verbraucher-Hardware gewahrt bleibt. Die endgültige Architektur (einschließlich der Zwischenschritte und der verwendeten Datensätze) wird als Open Source veröffentlicht, um die Reproduzierbarkeit und Anwendung in akademischen, industriellen und öffentlichen Domänen zu unterstützen.

# Abstract

In many real-world business cases, data is inherently imbalanced: certain classes occur far less frequently than others, posing significant challenges to conventional machine learning models. These models often underperform on minority classes, which can be the most critical for decision-making in domains such as cybersecurity, medical diagnostics, fault detection, and others.

This thesis addresses the problem of multi-class classification on imbalanced structured datasets by developing a general-purpose neural network architecture that can be applied across diverse use-cases without domain-specific tuning. The proposed architecture is also compared to other commonly used contemporary approaches and benchmarked on four publicly available datasets (KDD-99, Darknet, Shuttle, and Covertype), each representing a unique combination of dimensionality, size, and imbalance severity.

The results demonstrate that specific architectural choices, including Random Mixing Layers introduced in this work, can yield strong performance across structured imbalanced datasets while maintaining computational efficiency on common consumer-grade hardware. The final architecture (including the intermediary steps and the used datasets) is released as open-source to support reproducibility and application across academic, industrial, and public domains.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**ADASYN** Adaptive Synthetic Sampling

**AUC** Area Under the Receiver-Operating-Characteristic (ROC) Curve

**AUC-PR** Area Under the Precision-Recall Curve

**BA** Boosting Algorithm

**BGD** Batch Gradient Descent

**CIC** Canadian Institute for Cybersecurity

**CNN** Convolutional Neural Network

**CPU** Central Processing Unit

**DoS** Denial of Service

**FL** Focal Loss

**GB** Gigabyte

**GELU** Gaussian Error Linear Unit

**GHz** Gigahertz

**GPU** Graphics Processing Unit

**HAFL** Highly Adaptive Focal Loss

**IDS** Intrusion Detection Systems

**LCS** Linear Classifier System

**MAF1** Macro Average F1 Score

**MAP** Macro Average Precision

**MAR** Macro Average Recall

**MCC** Matthews Correlation Coefficient

**ML** Machine Learning

**MSE** Mean Squared Error

**NN** Neural Network

**RAM** Random Access Memory

**ReLU** Rectified Linear Unit

**SMOTE** Synthetic Minority Over-sampling Technique

**SMOTE-NC** SMOTE for Nominal and Continuous Features

**SVMSMOTE** Support Vector Machine SMOTE

**WAF1** Weighted Average F1 Score

**WAP** Weighted Average Precision

**WAR** Weighted Average Recall

**XAI** Explainable Artificial Intelligence

# 1 Introduction

In many real-world applications, data is inherently imbalanced, meaning that certain classes occur far less frequently than others. This phenomenon is particularly evident in the modern data-driven socioeconomic system, where private businesses and public organizations must make critical decisions based on skewed datasets. The presence of class imbalance poses significant challenges in machine learning, particularly in the domain of supervised learning, where classification models are often biased towards the majority class. This leads to models that fail to correctly identify minority class instances, thereby diminishing their practical utility.

There exist many examples where the minority classes–often the most important ones–can be easily overlooked by standard machine learning models. In the private sector, cybersecurity and network safety [WSL23] [BGJ20] are two of the most notable applications: each business and public institution is, of course, interested in detecting as many attacks on its infrastructure as possible. An undetected attack in this example could lead to severe financial and reputational damage, yet many intrusion detection models struggle due to the limited number of attack examples available in training data. And, since each attack is unlike the other, it is also important for machine learning models to be able to classify network accesses in not only "benign" and "malign" types, but also in different classes of attacks, like e.g. Denial of Service (DoS) or brute-forcing user passwords.

Similarly, the management of equipment health and fault diagnosis [Ren+23] is another field where it becomes ever more important to analyze large data structures for root causes and classify data points based on a few significant factors. Fault detection is critical in sectors such as aerospace, manufacturing, and healthcare, where an undetected failure could lead to catastrophic consequences.

Beyond these examples, biological taxonomy classification [SR20] [LQG21] and medical applications like diagnosis and screening of rare diseases [KCP11] [Liu+22] also highlight the importance of such approaches in public health. Many life-threatening conditions, such as rare cancers or genetic disorders, are naturally underrepresented in medical datasets. If machine learning models are not properly adjusted for class imbalance, they may fail to identify these diseases, leading to misdiagnosis and inadequate treatment.

There are even cases like oil spill [KHM98] and forest fire [Lai+22] detection, where not recognizing the underrepresented class, e.g., oil spills, is more detrimental than falsely recognizing normal data points as anomalies. If left undetected, oil spills and wildfires pose grave ecological and economic threats–at the same time, misidentifying normal surfaces as oil spills or mistakenly classifying harmless events as fires could lead to a waste of valuable first respondent resources.

Addressing this imbalance is crucial, as failing to do so can lead to biased predictions, poor generalization, suboptimal decision-making, and, ultimately, life-threatening situations. Neural Network (NN) architectures designed specifically for multi-class imbalanced datasets are therefore becoming an essential research focus,

aiming to improve model performance while ensuring fair and reliable predictions. While significant progress has been made in handling such datasets through resampling techniques, cost-sensitive learning, and ensemble methods, deep learning solutions for classification in imbalanced data remain relatively underdeveloped. Existing approaches, though valuable, often require manual tuning, careful preprocessing, or domain-specific adaptations, which limits their general applicability. More importantly, no single method has emerged as a definitive solution, as class imbalance remains a persistent challenge in real-world applications across numerous domains. This gap necessitates the development of novel NN architectures that can generalize well across various imbalanced structured (i.e. tabular) datasets.

This thesis, as such, focuses on building a universal NN architecture that can be applied to a variety of imbalanced structured datasets with high efficiency, regardless of the dataset's distribution, origin, and real-life application. The goal is to develop an adaptable model that mitigates the limitations of traditional approaches and tailors them to skewed datasets by incorporating thoroughly benchmarked architectural adjustments.

Contemporary research is often constrained to very specific use cases, developing architectures that are applied to one specific dataset and one specific application [KCP11] [LQG21] [Lai+22]. Often, however, especially in yet unknown applications, there is a large gap in research about possibly applicable methods. This problem is even more prevalent for businesses that find new, novel and yet unresearched use cases. The need for a generalizable solution is therefore pressing, as businesses and researchers require adaptable models that can be deployed across various domains with minimal reconfiguration.

In the recent few decades, there has indeed been considerable development of NNs for purposes of multi-class classification. Most of these, like the recent ResNeSt [Zha+20], RegNet [Xu+21], and ConvNeXt V2 [Woo+23], are, however, aimed at solving problems in the field of computer vision: classifying an object portrayed in an input image into a specific label. These NNs are notable for the fact that, during training, they usually receive balanced input data, due to the nature of benchmark datasets like CIFAR-100 [Kri09] and MNIST [LCB94]. However, in many real-world applications, structured datasets suffer from extreme class imbalances that require specialized strategies for achieving robust classification performance.

The research papers that look at convolutional NNs and modifications thereof not just as a way to classify images are, as of yet, few and far between. This creates a significant barrier to entry for practitioners and researchers who wish to build upon existing methodologies. The lack of standardized, openly accessible models limits the impact of research findings and hinders progress in both academic and industrial settings.

Another gap in contemporary research that motivates this thesis is thus the lack of high-quality open-source solutions stemming from contemporary research. Many papers that concern themselves with developing predictive models for imbalanced datasets do not provide their models and training code publicly [HLS22] [BGJ20] [SR20] [Liu+22]. Inevitably, this leads to modern research providing comparatively limited

value to the scientific, social, and economic progress of society. What's more, research results that cannot be replicated have been a heated topic in scientific research for almost a decade [Bak16]. The scientific community has recently started demanding a lot more diligence in such matters [HWS20], and with great results: research on the reuse of research artifacts shows an increase in availability and quality of research artifacts [Win+22] [Bal+23].

As such, a secondary goal of this thesis is to provide an open-source implementation of the developed universal NN architecture, thus further supporting the move towards replicability in public research and providing a benefit to economic and public actors that may find this research and its results to be of practical use.

In summary, the ultimate aim of the thesis reads as follows:

> *To build a universal open-source NN architecture that can be efficiently applied to a variety of imbalanced structured datasets, regardless of the dataset's distribution, origin, and real-life application.*

To this end, the thesis benchmarks 39 different iterations of a NN architecture, each with different approaches to mediating the imbalance of training data, on 4 different imbalanced datasets, each with distinct use cases, class distributions, and sizes, and in doing so determines the best universal architecture for solving multi-class classification problems in imbalanced multivariate data that can be run on almost any contemporary user-grade machine without a need for large investments into computational centers.

Additionally, to support the secondary goal of this thesis, both the datasets and the code used to generate the predictive models and plots will be uploaded to an online repository for others to reuse this research in their own projects or to run their own benchmarks and thus assert the correctness of the thesis' results.

# 2 Theoretical Background

The technologies used to classify the multi-class datasets described in this chapter can be divided into three main categories:

1. Learning Classifier System (LCS)
2. Boosting Algorithm (BA)
3. Neural Network (NN)

Each of them can be applied to a structured dataset with (usually) exclusively numerical data, where this data is organized into a table. Each row (i.e. data point) of such datasets has a set of values for the described features and a category for said set of values. The classifying technologies can then be trained using these datasets in order to learn the connections between the categories (also known as *labels* or *classes*) and the features' values. These trained models can then be applied to similarly-structured datasets and, for each new data point, "predict" which class it will be assigned to.

What follows in this chapter is a description of each of the aforementioned three different technologies. After that description, this chapter provides information on the intricacies of training NNs, and explains the datasets used to benchmark the aforementioned technologies. Finally, this chapter describes the used datasets, the methods used for resampling the datasets, and how the quality of different approaches is measured in this work.

## 2.1 Learning Classifier Systems

One of the more simple to understand multi-class classifiers is the LCS.

The foundational block of a LCS is a so-called *rule*, also known as a *classifier* [UM09]. A formal representation of how a rule looks like is shown in Figure 1.

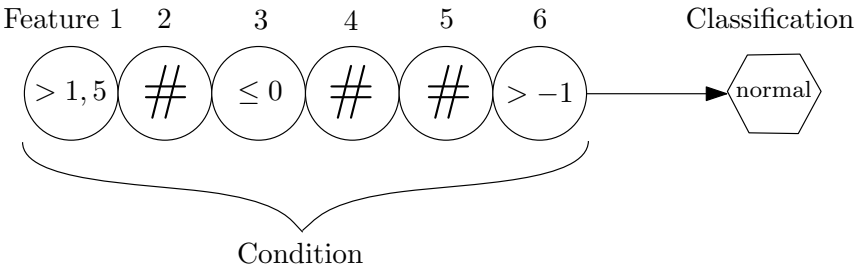


Figure 1: Rule in a LCS

A rule has two main parts: the condition and the classification. In the above case of the rule in Figure 1, the rule would classify a data point into the "normal" class if:

1. Its first feature's value was $> 1,5$

2. Its third feature's value was $\leq 0$
3. Its sixth feature's value was $> -1$

The second, fourth, and fifth features' values would not matter, as these are marked as *wildcards*, here portrayed using the "#" symbol.

A LCS uses a set of these rules to make its final decision. Each of these rules in a LCS has its own metrics such as numerosity, accuracy, fitness, and others that help the LCS to weight them. Through this, the LCS can define which rules have proven themselves to be more or less accurate during the training process.

This set of rules in a LCS is called the *rule-set*, and it can always be viewed by the user. This has made LCS models especially useful in the field of E (XAI), since decision makers can always visualize the reasoning of the model that made it classify a data point into a specific label.

During the training process, the LCS processes one data point after the other in an iterative manner. This instance is compared for matches among the rule-set: if there are rules that match the data point's condition, then these are placed into the *match-set*.

If there are no matches (for example during the first few training iterations), then the covering mechanism comes into play. During covering, the training instance is taken as the starting point of the new rule. Then the mechanism sets some attributes of the condition to wildcards, and saves this new rule in the rule-set. This mechanism is sometimes called *Smart Population Initialisation* [UGM14].

Then the match-set is divided into two parts based on the conditions of both the training data point and the rules of the match-set: those rules whose conditions match that of the data point's are moved into the *correct-set*, while the others are moved into the *incorrect-set*. Depending on which of the two sets the rules landed in, they receive either an improvement to their metrics (e.g. numerosity and accuracy), or a reduction of those.

After this come different steps that aim to increase the diversity of the rule-set. Some of the most common are:

- *Subsumption*: if the correct-set contains two rules such that one's condition is fully covered by that of the other rule (i.e. if the second rule has more wildcards, whereas the first rule specifies these wildcard attributes), then the second rule "subsumes" the first rule by absorbing its numerosity and adding it to itself. The more specific rule then gets deleted from the rule-set.
- *Genetic Algorithm*: two rules from the rule-set are selected as *parent rules*. The *child rule* gets created as a result of randomly selecting whether it gets to keep the condition from the first or the second parent rule for each of the condition's values.
- *Deletion*: if a rule is decided to be too uncommon and/or inaccurate based on its metrics, the LCS may decide to remove it from the rule-set.

After all instances are processed, the LCS outputs its final model that can be used to make predictions for unlabeled structured data.

## 2.2 Boosting Algorithms

Boosting is a machine learning technique designed to improve the accuracy and robustness of predictive models by combining multiple so-called *weak learners* to create a *strong learner* model. The key idea behind boosting is to train this model sequentially, where each subsequent iteration of the model corrects the mistakes of the previous ones.

Each model learns from the errors of its predecessor, placing greater emphasis on misclassified examples. This is achieved by granting a greater weight to misclassified instances, ensuring that challenging cases receive more attention in subsequent rounds. As a result, boosting produces a strong classifier that is highly accurate and capable of capturing complex patterns in data. This iterative process allows boosting algorithms to focus on difficult-to-classify instances, leading to improved generalization and performance.

*Weak learners* are the classification models that perform slightly better than random chance (an example of such a weak learner would be a decision tree with a small depth). Such small models are commonly used due to their simplicity and ability to capture non-linear relationships. Each weak learner contributes to the final prediction, and they are combined through a weighted voting or aggregation process, where more accurate learners receive higher influence. This adaptive weighting mechanism ensures that the overall predictive performance improves over time.

Despite the similarity between these approaches provided by BAs and LCS–both basing themselves on large sets of simple weighted classifiers–there are also many differences. A LCS is a rule-based system that evolves a population of classifiers using reinforcement learning and genetic algorithms: LCS models focus on discovering and maintaining a diverse set of rules for decision making, rather than combining weak classifiers to improve performance. In this way, a LCS is based on a more random approach than boosting, which iteratively adjusts weights to focus on misclassified examples.

A major advantage of boosting is therefore its ability to achieve high accuracy by focusing on difficult cases that other models might overlook. Thanks to this, it is widely used in many classification tasks and is particularly effective for handling imbalanced datasets.

Two of the most commonly used contemporary BAs are CatBoost and XGBoost.

**CatBoost** (*Cat*egorical *Boost*ing), developed by Yandex, is a BA specifically designed to handle categorical data efficiently while mitigating common issues such as overfitting and slow training [Pro+19].

CatBoost retains the fundamental principles of boosting while introducing some enhancements that make it more effective on datasets with categorical variables:

- *Oblivious Decision Trees*: Unlike traditional decision trees, an oblivious tree enforces a splitting condition on the same feature for all nodes in a row, resulting in a highly structured and symmetric tree architecture [Koh94] [FM16].

- *Ordered Boosting*: A well-known challenge in machine learning is target leakage, which occurs when the model learns from data that would not be available at the

time of prediction [Sas+23]. To address this, CatBoost employs a technique called ordered boosting, where each tree is trained using data that was available *before it* in the training sequence. By ensuring that a data point is never used to inform the predictions of earlier trees, ordered boosting reduces the risk of overfitting.

- *Encoding of Categorical Features*: Instead of transforming categorical features into high-dimensional sparse vectors (i.e. one-hot encoding), CatBoost replaces them with statistical representations based on the target variable [Pro+19].

- *Reduction of Hyperparameters*: CatBoost is designed to require minimal hyperparameter tuning, making it more accessible compared to other gradient boosting frameworks [Pro+19].

CatBoost thus represents an easily applicable tool for multi-class classification, particularly for datasets containing categorical variables where these variables have not yet been preprocessed using methods like one-hot encoding.

**XGBoost** (E*x*treme *G*radient *Boost*ing) is an earlier high-performance, scalable machine learning algorithm developed in 2016 [CG16]. It introduces several enhancements, including parallel computation and built-in regularization techniques which make XGBoost one of the most widely used algorithms for structured data tasks.

XGBoost builds upon the traditional gradient BA framework while incorporating novel optimizations to improve computational efficiency and predictive quality:

- *Weighted Quantile Sketch Algorithm*: When a decision tree tries to split data, it needs to find the best point to split a feature (e.g., if a feature is "age," it might decide to split at 30 years old vs. everything else). For small datasets, this is easy–one must only sort all values and try different split points. But for large datasets with millions of rows, sorting all values is too slow and memory-intensive. Instead, XGBoost groups numerical values into small buckets (e.g., instead of keeping every age, it groups them into ranges like 18-25, 26-35, etc.), assigns weights to these buckets based on how many data points fall into them, and selects the most representative points for splitting. This allows XGBoost to efficiently find the best split points for numerical features, improving scalability on large datasets [CG16].

- *Sparsity-aware Split Finding*: Real-world datasets often contain missing values or lots of zeros. Most decision trees struggle with missing data because they require filling in missing values before training (which can introduce bias). Instead of filling values in, XGBoost assigns missing values to whichever side of the tree split leads to a lower training loss (i.e. whichever path improves accuracy) [CG16].

- *Gradient Boosting with Second-Order Approximation*: In regular boosting approaches, decision trees are built by minimizing classification errors using gradients (first derivatives). However, this doesn't always provide the best guidance–sometimes, it's also necessary to know how fast the gradient is changing to make better updates. XGBoost thus also uses the curvature (i.e. second derivative, or Hessian) to

make better adjustments to its tree splits and to more accurately adjust the measure by which it changes the criteria inside of a tree node (i.e. adjust the step size). This in turn allows XGBoost to converge faster and to reach an optimal solution sooner [CG16].

- *Regularization to Reduce Overfitting*: Gradient boosting can create very deep, complex trees that perfectly fit the training data but fail to generalize to new data. To combat this, XGBoost adds a penalty when trees get too complex.

  The model makes predictions using the additive approach:

  $$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), \quad f_k \in \mathcal{F}$$

  Where:

  - $\hat{y}_i = \phi(x_i)$ is the predicted output for the $i$-th sample;
  - $f_k(x_i)$ represents the $k$-th decision tree;
  - $\mathcal{F}$ is the space of regression trees;
  - $K$ is the total number of trees in the model.

  The learning process thus minimizes the following objective function:

  $$L(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

  Where:

  - $l(y_i, \hat{y}_i)$ is the loss function measuring the difference between the predicted output $\hat{y}_i$ and the true value $y_i$;
  - $\Omega(f_k)$ is the regularization term that penalizes complex trees.

  The regularization function $\Omega(f)$ is designed to control tree complexity and prevent overfitting, and is defined as:

  $$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

  Where:

  - $T$ is the number of leaf nodes in the tree;
  - $w_j$ is the weight assigned to the $j$-th leaf;
  - $\gamma$ controls the penalty for having more leaves (encourages simpler trees);

  - $\lambda$ controls the L2 penalty on leaf weights (and thus prevents over-reliance on just a few leaves).

  Regularization thus discourages overly complex trees and smooths leaf weights, reducing overfitting.

- *Parallelized and Optimized Training*: Unlike traditional boosting methods that construct trees sequentially, XGBoost is designed for parallel computation, allowing it to train much faster on large datasets. XGBoost stores data in a compressed columnar format, allowing for efficient memory access and cache utilization, being able to process large datasets in batches, and enabling training on machines with limited Random Access Memory (RAM)–an important factor to consider when choosing between different BAs.

XGBoost thus represents a well-refined algorithm that ensures good generalization on large datasets, while also being optimized for speed and memory efficiency.

Many studies have been conducted on which of the two aforementioned BAs is the most optimal. The general consensus is that there is no universally "best" algorithm, and that the choice depends on the specific dataset characteristics and application requirements [BCM21]. The datasets being considered in the scope of this work, however, are all large (and, in some cases that will be explored later, exceptionally so), totaling well over 40000 entries each. For datasets like these, research has shown XGBoost to be the most optimal solution when compared to CatBoost, boasting an improvement of about 5% on average in absolute terms [Bol+23].

Moreover, the nature of the datasets being considered in this thesis (which will be introduced in a later chapter) demanded a unified common approach to encoding their data, and the categorical features were thus all converted into numerical ones using one-hot encoding.

Both of these factors combined ultimately made the argument for only considering XGBoost in the scope of this thesis, as it is an algorithm better suited for large datasets and for numerical and one-hot encoded features.

## 2.3 Neural Networks

NNs have become one of the most important tools in modern machine learning, enabling significant breakthroughs in areas ranging from image recognition to natural language processing. At their core, NNs are mathematical models designed to recognize patterns, learn from data, and make predictions. NNs have revolutionized artificial intelligence (and, in many ways, pioneered the term) because they can handle vast amounts of data and uncover patterns that traditional algorithms struggle to detect. Unlike traditional machine learning models, which often rely on explicitly programmed rules, NNs have the ability to extract complex relationships from data by adjusting internal parameters through an iterative learning process [GBC16].

The fundamental idea behind NNs is inspired by the human brain [Hay09]. The brain consists of billions of interconnected neurons that transmit signals to process information. While NNs are vastly simplified versions of this biological system, they follow a similar principle. At its core, a NN is a mathematical system designed to approximate complex functions. Given a set of inputs, it processes them through multiple transformations to produce an output that best fits the expected result.

The fundamental building block of a NN is the *neuron* (also called a node) [GBC16]. A neuron takes in one or more numerical inputs, applies a weight to each input, sums them up, and then passes the result through an *activation function*–a mathematical transformation that determines whether the neuron "activates" and passes information forward. This process mimics the commonly accepted way that biological neurons transmit signals in the brain, where only significant inputs trigger a response.

Neurons are organized into *layers*, forming the overall structure of a NN [GBC16]. These layers define how information flows from raw data to the final output:

- *Input Layer*: This is where the network receives data. Each neuron in the input layer corresponds to a single feature in the dataset (e.g. pixel values in an image, words in a sentence, or sensor readings in a machine). The input layer does not modify the data, but simply passes it forward.

- *Hidden Layers*: These are the core of the network where computation happens. Each neuron in a hidden layer takes weighted inputs from the previous layer, applies an activation function, and transmits the result forward. Generally, the more hidden layers a network has, the more complex relationships it can learn.

- *Output Layer*: This is the final layer, where the network produces its prediction. The number of neurons in this layer depends on the task–if it's a binary classification, there is typically one neuron (outputting a probability), while in multi-class classification–which is the use-case in this thesis–there are multiple neurons, each representing a class.

By tuning the weights and biases of the neurons, the NN learns the best way to map inputs to outputs.

Once a NN is created, it takes in new data and produces a prediction. This process, known as *forward propagation*, is how information flows through the network [RHW86].

An example of forward propagation would be using a NN to predict the price of a house based on features such as location, size, and age. Each of these features is an input neuron. These inputs are passed to the first hidden layer, where each neuron calculates a weighted sum of the inputs, applies an activation function, and passes the result forward. This process repeats through multiple hidden layers, each one of these further refining the information. Finally, the output layer produces a prediction–in this case, the estimated house price.

Mathematically, for each neuron, the output value $z$ is calculated as:

$$z = \sum_{1}^{n} w_i x_i + b$$

Where:

- $x_i$ are the inputs;
- $w_i$ are the input weights;
- $n$ is the amount of inputs in the neuron;
- $b$ is the bias (or correction factor).

This value is then passed through an activation function to introduce non-linearity and moves layer by layer until it reaches the output, which outputs the final prediction.

After making a prediction–which during initialization of the NN is most likely inaccurate–the NN needs to determine how wrong it was and to adjust accordingly. This is done using a process called *backpropagation*, which allows the network to learn from its mistakes [RHW86].

The first step is calculating how far off the prediction is from the actual value. This is done using a loss function, which quantifies the error.

If one were to thus predict house prices, the Mean Squared Error (MSE) loss function might be used:

$$\text{Loss} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Where:

- $y_i$ is the actual price in the training dataset;
- $\hat{y}_i$ is the price predicted by the NN.

The larger the loss, the worse the prediction.

The network then needs to adjust its weights to reduce the calculated error. This is done using gradient descent, an optimization algorithm that finds the direction in which the weights should move to minimize the loss [Lec+98].

Mathematically, gradient descent updates the weights as follows:

$$w_{new} = w_{old} - \alpha \frac{\partial_{Loss}}{\partial_{w_{old}}}$$

Where:

- $w_{new}$ is the updated weight;
- $w_{old}$ is the current weight (i.e. the old value before updating);

- $\alpha$ is the learning rate (e.g. 0,01 or 0,001) that controls how big the adjustment is;
- $\frac{\partial_{Loss}}{\partial_{w_{old}}}$ is the gradient, i.e. the derivative of the loss function with respect to the weight $w_{old}$.

The NN then repeats this process however many times necessary, tweaking weights a little bit each time. Over time, through trial and error, the predictions become more accurate.

NNs, despite their complexity, offer several advantages when dealing with imbalanced datasets. Traditional machine learning models often favor the majority class while ignoring underrepresented categories. NNs, however, have several characteristics that make them well-suited for handling such imbalances.

Unlike traditional models that rely on predefined rules or simple statistical relationships, NNs can capture intricate patterns in data. NNs can also automatically learn feature representations that emphasize important characteristics of the data. This means that, even if the minority class has only a few samples, the NN can still extract meaningful information to distinguish it from the majority class [JK19].

In addition, NNs can be combined with other techniques, such as BAs or cost-sensitive learning, to further enhance performance [Jia+23]–something that will in fact be considered in the scope of this work. By leveraging multiple approaches, NNs can adapt to imbalanced datasets in ways that traditional models cannot.

## 2.4 Model Training

There exist many elements of NNs that can be tweaked for maximum efficiency on specific datasets and for specific use-cases. This section further highlights the most relevant ones that are used and tweaked in the scope of this work.

### 2.4.1 Batch Sizes

In the context of training NNs, the term *batch size* refers to the number of training examples utilized in one iteration of model training.

When training a NN, data is typically divided into smaller subsets called *batches*. This approach contrasts with using the entire dataset at once, known as *Batch Gradient Descent (BGD)*, or utilizing a single training example at each iteration, known as *stochastic gradient descent*.

The choice of batch size can significantly impact both the performance and speed of NN training. Smaller batch sizes often lead to more noisy gradients, which can help escape local gradient minima and potentially lead to better generalization on unseen data. However, they also result in longer training times due to more frequent weight updates and increased computational overhead. Conversely, larger batch sizes provide more stable gradient estimates but may converge to sharper minima in the loss landscape, which can negatively affect generalization [Kes+16].

Choosing an appropriate batch size involves balancing several factors:

- *Generalization vs. Convergence*: Smaller batches tend to improve generalization by introducing noise into the optimization process. In contrast, larger batches may converge faster, but risk overfitting.
- *Training Speed*: Larger batch sizes can take advantage of parallel processing capabilities of modern hardware (e.g., GPUs or TPUs), leading to faster computation per epoch. However, this advantage must be weighed against potential degradation in model accuracy.
- *Memory Constraints*: The available memory on hardware will limit how large a batch size can be used without causing errors during computation due to lack of operational memory.

To determine an optimal batch size for a specific task, users often employ empirical methods such as grid search or trial-and-error approaches while monitoring validation performance metrics [Smi17]. It is common practice to start with smaller batch sizes (e.g., 32 or 64) and gradually increase them based on observed performance until reaching a point where further increases do not yield significant improvements.

Selecting an appropriate batch size thus requires careful consideration of its effects on both model convergence speed and generalization ability and hardware limitations, which will also be observed in later chapters of this work.

### 2.4.2 Train, Test, and Validation Splits

In the process of training NNs, it is essential to divide the available dataset into distinct subsets: the *training set*, the *validation set*, and the *test set*. Each of these subsets serves a unique purpose in ensuring that the model is trained effectively and evaluated accurately.

- **Training Set**: This subset is used to train the NN. During training, the model learns patterns from this data by adjusting its weights based on the input features and corresponding target outputs. The more diverse and representative this dataset is of real-world scenarios, the better the model can generalize.
- **Validation Set**: The validation set can (yet doesn't have to) be utilized during training to fine-tune hyperparameters (e.g. learning rate and batch size) and make decisions about model architecture (e.g. number of layers). Moreover, it is also common to use the validation set as the actual "benchmark" dataset on which the trained model is applied in each epoch, in order to, for example, calculate the loss on the validation set and determine if the model should halt training after its improvement and classification quality plateaus.
- **Test Set**: Finally, after completing training and validation processes, the test set provides an unbiased evaluation of the final model's performance. It is crucial that this dataset remains untouched during both training and validation phases to

ensure that results reflect true predictive capability rather than memorization of specific examples.

A common practice in machine learning is to use an 80-20 split for dividing datasets into training and testing sets respectively; that is, to use 80% of data for training purposes and 20% for testing [Tan+21]. This ratio strikes a balance between having sufficient data to train models effectively while retaining enough samples for reliable performance assessment.

Moreover, some users also allocate a portion of their dataset (often around 10-15%) as a validation set within that 80% allocated for training [Koh95]. Thus a typical distribution might look as such:

- Training Set: 70-80%
- Validation Set: 10-15%
- Test Set: 10-20%

This division ensures robust evaluation metrics while maximizing information extraction from available data. Users, however, must be careful to make sure that each of the three datasets has samples from each class when considering multi-class classification problems (especially in imbalanced datasets), otherwise the model risks being unable to learn important patterns in the data.

### 2.4.3 Activation Functions

*Activation functions* are a crucial component in the architecture of NNs, serving as the mathematical gatekeepers that determine whether a neuron should be activated or not. They introduce non-linearity into the model, allowing NNs to learn complex patterns and relationships within the data. Without activation functions, a deep NN would essentially behave like a linear regression model, regardless of how many layers it has, thus limiting its capacity to capture intricate structures in datasets.

During the training of NNs, activation functions play several key roles:

- *Non-linearity*: They enable the network to combine inputs in non-linear ways, which is essential for learning from complex datasets.
- *Gradient Flow*: Activation functions affect how gradients propagate back through the network during the training process. This can significantly influence convergence speed and overall performance.
- *Output Range Control*: Different activation functions constrain outputs to specific ranges (e.g. between 0 and 1 or -1 and 1), which can be beneficial depending on the task at hand.

Several activation functions are widely used in practice, each with its own characteristics:

1. **Sigmoid Function**:
$$f(x) = \frac{1}{1 + e^{-x}}$$
The sigmoid function maps input values to an output range between 0 and 1. While historically popular for binary classification tasks, it suffers from issues such as vanishing gradients when inputs become too large or too small.

2. **Hyperbolic Tangent (tanh)**:
$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
The tanh function is similar to the sigmoid, but maps inputs to a range between -1 and 1. It generally performs better than sigmoid because it is zero-centered; however, it still faces vanishing gradient problems.

3. **Rectified Linear Unit (ReLU)**:
$$f(x) = \max(0, x)$$
ReLU is one of the most commonly used activation functions today due to its simplicity and effectiveness. It behaves much like the sigmoid function: ReLU outputs zero for negative inputs and passes positive inputs unchanged. This helps mitigate vanishing gradient issues, but can lead to the "dying ReLU" problem where neurons become inactive during training. Though it is sometimes accepted that the sigmoid function is better at imitating the neuron response in organic brain matter, ReLU is used more often as the more computationally efficient alternative that provides a similar enough performance.

4. **Leaky Rectified Linear Unit (Leaky ReLU)**:
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$
Where $\alpha$ is a near-zero propagation factor.

   Leaky ReLU addresses some limitations of standard ReLU by allowing a small, non-zero gradient when the input is negative. This prevents neurons from becoming inactive during training while still maintaining much of ReLU's benefits.

5. **Gaussian Error Linear Unit (GELU)**:
$$f(x) = x \cdot \Phi(x)$$

Where $\Phi(x)$ is defined as:

$$\Phi(x) = P(Z < x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

GELU combines properties of both linear units and probabilistic models by applying a Gaussian distribution scaling factor to its output ($f(x) = x \cdot \Phi(x)$, where $\Phi(x)$ is the cumulative distribution function of a standard normal distribution).

In terms of performance across different tasks and architectures, various studies have highlighted differences among these activation functions:

- A comparative analysis by Huang et al.demonstrated that ReLU outperforms traditional sigmoid and tanh activations in deep convolutional networks due to faster convergence times [Hua+17].
- However, recent work by Dumoulin and Visin indicates that while ReLU works well for many applications, GELU can lead to superior performance in transformer models due to its ability to provide more informative gradients throughout training [DV16].
- In Kumar et al., experiments showed that using GELU instead of ReLU resulted in improved accuracy on benchmark datasets across various architectures including ResNet and DenseNet [KGS22].

Recent studies thus suggest that GELU provides smoother gradients compared to ReLU and improves convergence rates in deeper networks. Unlike ReLU, it allows some small negative values to pass through, helping gradient flow. Therefore, within the scope of this work, GELU will be employed as the primary activation function due to its demonstrated efficacy in enhancing model performance during training–whenever any activation function is to be employed at all.

### 2.4.4 Loss Functions and Class Weights

Selecting an appropriate *loss function* is crucial for effective training, especially in scenarios involving imbalanced datasets. Common loss functions used in multi-class classification problems include:

1. **Categorical Cross-Entropy**:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- $y_i$ represents the true label for class $i$, which takes a value of 1 if the sample belongs to class $i$, and 0 otherwise;
- $\hat{y}_i$ is the predicted probability of the sample belonging to class $i$;
- $C$ is the number of classes.

This loss function is commonly used for multi-class classification tasks where each sample belongs to one class. It measures the dissimilarity between the predicted probability distribution and the true (one-hot encoded) distribution.

2. **Weighted Categorical Cross-Entropy**:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} w_i y_i \log(\hat{y}_i)$$

Where $w_i$ represents the weight of class $i$ given to the loss function.

This variant extends categorical cross-entropy by incorporating weights that can be assigned to each class. This approach helps address class imbalance directly by penalizing misclassifications of minority classes more heavily.

3. **Focal Loss (FL)**:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} \alpha(1 - \hat{y}_i)^{\gamma} y_i \log(\hat{y}_i)$$

Where:

- $\gamma$ is the focusing parameter to be tuned during training;
- $\alpha$ is a constant that balances the importance of samples, which adjusts how much emphasis is placed on different classes during training.

FL modifies categorical cross-entropy to place more focus on hard-to-classify examples while down-weighting easy-to-classify examples, in which it is similar to the weighted categorical cross-entropy. This characteristic makes it particularly useful for imbalanced datasets.

4. **Highly Adaptive Focal Loss (HAFL)**:

$$L(y_i, \hat{y}_i) = -\alpha(1 - p_t)^{\gamma^*_{\text{adaptation}}} \log(p_t)$$

Where:

- $p_t$ denotes the predicted probability for the true class.
  If $y_i = 1$, then $p_t = \hat{y}_i$; otherwise, if $y_i = 0$, it corresponds to the predicted probability for other classes;

- $\gamma^*_{\text{adaptation}}$ is the adaptive attenuation factor that changes dynamically based on model training progress.

The attentuation factor $\gamma^*_{\text{adaptation}}$ can be computed the following way:

$$\begin{aligned}\gamma^*_{\text{adaptation}} &= \gamma_{\text{start}}\gamma^*_{\text{epoch}}\gamma^*_{\text{balance}}\gamma^*_{\text{precision}} \\ &= \gamma_{\text{start}}\frac{1}{\sqrt{\beta}}\sqrt{\frac{nums_i}{\max(nums_j)}}(\log(\frac{precision_i}{\max(precision_j)}+1)+0,5)\end{aligned}$$

Where:

- $\beta$ is the number of iterations;
- $nums_i$ is the number of samples in the $i_{th}$ class;
- $\max(nums_j)$ is the number of samples in the class with the most samples;
- $precision_i$ is the average training accuracy of the $i_{th}$ category's samples;
- $\max(precision_j)$ is the average training accuracy of the best training category's samples.

HAFL is the most recent of the mentioned loss functions, developed by Jiang et al. [Jia+23], and promises great improvements to the classification accuracy by ensuring that the model maintains lasting attention to the minority samples.

In practice, modern frameworks like Tensorflow [Mar+15] also allow users to pass a list of *class weights* during training when using standard loss functions like categorical cros-entropy. By specifying weights corresponding to each class, users can therefore inform the NN about their importance relative to one another without having to define new weighted functions, if such aren't pre-implemented in the used frameworks.

This approach–using either tailored loss functions or class weights–enables models to more effectively handle imbalances present in real-world datasets.

### 2.4.5 Stopping Criteria

At some point, continuing to update the weights inside a NN either stops improving the model or even starts making it worse. This is when so-called *stopping criteria* come into play: these halt training after a certain condition is met.

There are several common stopping criteria:

1. *Convergence of the Loss Function*: If the loss stops decreasing significantly from one iteration to the next, the network is considered to have converged. A common example of this is the so-called *patience*: if the loss doesn't improved after a given number of epochs, then the NN stops training.

2. *Vanishing Gradients*: During backpropagation, if the gradients become too small, weight updates slow down, and learning practically stops. This halting can be sped up via a criterion that halts training when the gradient drops below a given value.

3. *Number of Epochs*: Since training NNs often takes a lot of time, the user can define a hard criterion of e.g. 100 epochs, after which the NN halts training.

In practice, it's also possible to combine multiple stopping criteria.

For example, a user can set a hard limit of 100 epochs, while also adding another criterion which halts training if the NN reaches a convergence of the loss function before reaching the 100th epoch.

## 2.5 Benchmark Datasets

In order to benchmark and thus determine how well the developed approaches perform over time, they would naturally have to be applied to different structured datasets, split in two subsets: the training subset, and the testing subset.

Since the stated problem focuses on imbalanced datasets, four datasets were chosen that were freely available online at the time of writing and that best fit this criterion.

The first dataset accessed was the **KDD Cup 1999 (KDD-99)** dataset [Sto+99]. It was originally created for the Third International Knowledge Discovery and Data Mining Tools Competition in 1999. It was designed specifically to benchmark Intrusion Detection Systems (IDS), focusing on identifying potential threats and anomalous activity in a network. As a result, the dataset is highly imbalanced, with the majority of data representing normal network traffic and several minority classes representing different types of network intrusions, such as DoS attacks, unauthorized access, and other forms of cyber attacks. The KDD-99 dataset is structured in tabular form, with each record representing a network connection and 41 associated features. These features include both continuous and categorical attributes, ranging from basic network attributes (e.g. protocol type, service, source bytes) to traffic content and traffic time-based attributes (e.g. number of connections to the same host). The original dataset contains over 4,8 million records in total, providing a substantial sample size for evaluating machine learning models on large-scale imbalanced data.

The second accessed dataset was **CIC-Darknet2020 (Darknet)** [LKR20]. This dataset was developed by the Canadian Institute for Cybersecurity (CIC) to support research on detecting malicious activities within the Darknet, which encompasses hidden online networks used for both legitimate and illicit activities. Unlike traditional network datasets, CIC-Darknet2020 focuses specifically on identifying behaviors and patterns associated with Darknet traffic, making it valuable for cybersecurity applications in areas like intrusion detection and anomaly detection. Given the nature of Darknet traffic, the dataset is also highly imbalanced, with benign connections representing the majority class and various types of Darknet connections making up the minority classes.

The dataset consists of records with more than 80 features per data point, capturing a wide range of network characteristics, such as packet lengths, header information, and time-based attributes. These features are designed to help machine learning models differentiate between normal and malicious behavior. With over 158 thousand entries, this dataset is the third largest and provides a good basis for training on varied data for each provided class.

The dataset that was accessed third was the **Covertype** dataset [Bla98]. This dataset was originally compiled by the U.S. Forest Service and researchers at Colorado State University to predict the type of forest cover, or vegetation, based on cartographic and environmental features. The Covertype dataset is moderately imbalanced, with the majority class (lodgepole pine) representing over 48% of the data and the remaining classes covering various forest types, such as fir, aspen, and cottonwood, each with considerably fewer samples. This distribution makes it suitable for testing approaches aimed at handling slightly less skewed imbalances than in the previous two datasets. The Covertype dataset consists of more than 581 thousand records, each with 54 features that capture a variety of factors, including soil type, elevation, slope, and climate characteristics. These features are a mix of continuous, ordinal, and categorical variables, providing a diverse set of training attributes.

The fourth accessed dataset was the **Statlog (Shuttle)** dataset [BM98]. Originally part of the Statlog Project, the Shuttle dataset was specifically created to classify shuttle data from NASA's space missions. This dataset is the third most imbalanced dataset in this work's line-up, with the majority class comprising over 78% of the samples. The dataset has nine numerical features, including measurements of positional, velocity, and telemetry data related to the shuttle's operations. Each record is classified into one of seven categories, with the minority classes representing unusual or abnormal conditions that may require additional attention. With approximately 58 thousand records, this dataset represents the smallest of all that will be benchmarked against, which makes it important insofar as measuring of the developed methods is concerned.

The four listed datasets were chosen due to their freedom of access and usage in scientific research. Unfortunately, most if not all imbalanced anomaly detection datasets that accurately reflect real business cases like card fraud and sensor data are not available for researchers outside of the concerned institutions (financial, logistical, industrial and otherwise) that have access to this data, especially not in a pre-labeled form.

Regardless, these four datasets provide a good and varied basis against which the developed approaches can be benchmarked: some being larger in both size and dimensionality, and some being less imbalanced than others.

## 2.6 Resampling Algorithms

Imbalanced datasets, like the ones introduced in the previous section, provide users with a plethora of challenges when it comes to achieving good accuracy on all classes. This imbalance, if approached improperly, can lead to biased models that favor the

majority class. While such models will still achieve a good accuracy when weighted for class sizes, the accuracy on underrepresented classes may still be close to zero.

Earlier sections looked at the different loss functions that may solve this issue.

Another solution is presented by *resampling algorithms*. Resampling algorithms mitigate this issue by modifying the original training dataset, most often by generating new synthetic instances from the original set.

Of such algorithms, the most notable ones (and the ones considered in the scope of this work) will be described below.

- **Synthetic Minority Over-sampling Technique (SMOTE)**:

  This method generates synthetic samples for the minority class by interpolating between existing instances. The primary idea behind SMOTE is to create new, synthetic examples rather than to simply duplicate existing ones, which helps in enhancing the decision boundary and improving model generalization [Bow+02].

  The process of generating synthetic samples involves several steps:

  1. For each instance $x_i$ in the minority class, randomly select $k$ nearest neighbors from the same class using a distance metric (e.g. the Euclidean distance).
  2. For each selected neighbor $x_i^j$, where $j = 1, 2, \ldots, k$, generate a synthetic sample $x_{\text{synthetic}}$ using the following formula:

     $$x_{\text{synthetic}} = x_i + \lambda(x_i^j - x_i)$$

     Where $\lambda$ is a random number uniformly drawn from the interval [0, 1].

     This equation essentially creates a point along the interval between $x_i$ and its neighbor $x_i^j$.
  3. Repeat the process in steps 1-2 until a desired level of oversampling is achieved.

  By employing this technique, SMOTE effectively increases the representation of minority classes in imbalanced datasets while preserving their underlying distribution characteristics. It mitigates overfitting associated with simple duplication, though it may also produce false new data points if the underlying distributions on some feature values don't follow the normal distribution, but e.g. a version of a bimodal distribution instead.

- **BorderlineSMOTE**:

  An extension of the original SMOTE algorithm is BorderlineSMOTE, which focuses on generating synthetic samples specifically for instances that lie near the decision boundary between classes. The rationale behind this approach is that instances close to the boundary are often more informative and critical for improving classifier performance [HWM05].

  The BorderlineSMOTE algorithm operates as follows:

1. Identify borderline instances in the data. An instance is classified as a borderline instance if at least half of its $k$ nearest neighbors belong to the majority class.
2. For each borderline instance $x_i$, identify its $k$ nearest neighbors and generate synthetic samples using the same approach as described for SMOTE.
3. Repeat the process in step 2 until a desired level of oversampling is achieved for all identified borderline instances.

By concentrating on borderline instances, BorderlineSMOTE aims to improve robustness by providing more relevant data points that can help refine decision boundaries. It slightly reduces the risk of generating noise by not randomly sampling from all minority instances, which may thus lead to better classification performance.

- **SMOTE for Nominal and Continuous Features (SMOTE-NC)**:

  While traditional SMOTE effectively handles datasets with continuous features, it struggles with nominal (categorical) features. To address this limitation, the SMOTE-NC algorithm was developed to generate synthetic samples in datasets that contain both types of features.

  When generating synthetic samples, SMOTE-NC distinguishes between continuous and categorical data:

  – For continuous features, synthetic values are generated using a similar approach to traditional SMOTE.
  – For nominal features, instead of interpolation, a random selection process is employed. Each nominal feature is given the value occurring in the majority of its $k$-nearest neighbors.

  The generated synthetic sample thus combines both the newly created continuous values and randomly chosen nominal values to form a complete new instance.

  By accommodating both categorical and numerical features, SMOTE-NC provides a more versatile solution for imbalanced datasets containing mixed data types– though the random selection of categorical features may again lead to creation of data points that fail to represent the truth.

- **KMeansSMOTE**:

  KMeansSMOTE incorporates clustering techniques to improve the generation of synthetic samples for minority classes [DBL18].

  The KMeansSMOTE algorithm operates as follows:

  1. The first step involves applying the K-means clustering algorithm to group instances from the dataset into $k$ clusters.

2. After this, the algorithm filters clusters to only consider those dominated by the minority classes.
3. For each cluster identified in the previous step, KMeansSMOTE generates synthetic samples using a modified version of the SMOTE approach:
   - For each instance $x_i$ within a cluster, its $k'$ nearest neighbors are determined (where $k'$ can be different from $k$ if specified so during initialization);
   - Synthetic samples are then created based on these neighbors using the aforementioned formula used for SMOTE.

The process then continues until a desired level of oversampling is achieved for all clusters.

By generating synthetic samples based on clusters, KMeansSMOTE aims to capture variations present in different sub-regions of feature space and to avoid the problem of oversampling over bimodal distributions. The primary advantage of KMeansSMOTE therefore lies in its focus on creating synthetic examples that reflect the *local* data distributions.

- **Support Vector Machine SMOTE (SVMSMOTE)**:

  SVMSMOTE is a variant of the SMOTE algorithm that leverages support vector machines (SVMs) to enhance the generation of synthetic samples for minority classes. The goal is to create more informative synthetic instances by considering the decision boundary defined by SVMs [NCK11].

  The algorithm operates as follows:

  1. Train a SVM classifier using the available data. The resulting classifier aims to find hyperplanes that best separate the classes in feature space.
  2. After training the SVM, the algorithm identifies support vectors. These lie closest to the decision boundary and are essential for defining its position, as they approximate the decision boundary needed for oversampling.
  3. With the support vectors identified, the algorithm generates new data instances along the lines joining each minority class support vector with a number of its nearest neighbors.

  SVMSMOTE thus aims to produce synthetic samples are strategically positioned near critical decision boundaries in the feature space.

- **Adaptive Synthetic Sampling (ADASYN)**:

  ADASYN is another resampling technique designed to address class imbalance by adaptively generating synthetic samples for the minority class. Unlike traditional methods that apply a uniform approach to oversampling, ADASYN focuses

on generating samples in a way that reflects the local distribution and density of minority instances [He+08].

This algorithm operates as follows:

1. Calculate the distribution of instances in both the minority and majority classes (in order to identify regions where the minority class is underrepresented and to determine how many synthetic samples need to be generated).
2. For each instance $x_i$ in the minority classes, ADASYN assesses its $k$ nearest neighbors. The algorithm then calculates the ratio of neighbors from the largest class to the total amount of neighbors. Instances with a higher ratio of majority neighbors are considered more difficult to classify.
3. Based on the difficulty level determined in the previous step, ADASYN assigns a weight $w_i$ to each minority instance $x_i$. This weight indicates how many synthetic samples should be generated for that instance.

   The number of synthetic samples $n_i$ for each such instance can be computed as follows:

   $$n_i = w_i N$$

   Where $N$ is the total number of desired new samples.
4. Similar to SMOTE, the algorithm creates new synthetic samples until the saturation condition is met.

ADASYN's stated goal is to provide more relevant data points and to thus shift the classifier decision boundary to be more focused on those difficult to learn examples [He+08].

It should be noted that *all* resampling algorithms can introduce considerable noise if not carefully applied. If the distributions of some features follow unusual distributions with multiple "peaks", the algorithms may have difficulty creating new data points that represent the true underlying distribution in the original data. Therefore, as will also be seen in this work, it is often necessary to combine those algorithms with other techniques to achieve the most optimal results–if these algorithms are to be used at all.

## 2.7 Evaluation Metrics

Different evaluation metrics provide insights into different aspects of classification performance, including precision, recall, and overall balance in prediction. Choosing the right evaluation metrics is imperative due to the imbalanced nature of the benchmarked datasets.

In standard classification problems, overall accuracy might be a sufficient measure of performance. However, in highly imbalanced datasets, accuracy alone can be misleading, as a model could achieve high accuracy by simply predicting the majority class

most of the time. Therefore, evaluation metrics must capture both the general predictive performance of the model and its effectiveness in correctly identifying minority classes. Metrics such as macro-averaged precision, recall, and F1 score provide a more balanced assessment, ensuring that the classifier's ability to predict underrepresented classes is also properly evaluated.

At the same time, weighted metrics cannot be excluded completely–in real-life cases, frequent misclassifications of overrepresented classes may lead to false decision-making and even to wasteful use of resources, like for example if too many normal water surfaces are classified as oil spills.

As such, this work uses eight different evaluation metrics for evaluating model quality:

1. **F2 Score**:

$$\text{F2} = (1 + 2^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{(2^2 \cdot \text{Precision}) + \text{Recall}}$$

   Where:

   - Precision measures the proportion of true positive predictions out of all positive predictions made by the model:

$$\text{Precision} = \frac{TP}{TP + FP}$$

   - Recall (or Sensitivity) is the proportion of actual positive instances that were correctly predicted by the model:

$$\text{Recall} = \frac{TP}{TP + FN}$$

   - $TP$ (True Positives) is the number of correctly predicted positive instances.
   - $FP$ (False Positives) is the number of instances incorrectly classified as positive.
   - $FN$ (False Negatives) is the number of actual positive instances incorrectly classified as negative.

   The F2 score is a weighted harmonic mean of precision and recall, placing more emphasis on recall.

   Due to its nature, it puts more emphasis on not missing positives. An intuitive interpretation of this can be found in the field of medicine: a doctor would rather diagnose healthy patients as needing a check-up than miss actually sick people.

   F2 ranges from 0 to 1. A score of 1 means perfect precision and recall.

2. **Matthews Correlation Coefficient (MCC)**:

$$MCC = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Where $TN$ (True Negatives) is the number of correctly predicted negative instances.

MCC is a balanced metric that takes into account true and false positives and negatives. Unlike accuracy, which can be misleading in imbalanced datasets, MCC ensures that the model is genuinely good at distinguishing both positive and negative cases.

MCC ranges from -1 (worst) to 1 (perfect), where 0 means the model is no better than random guessing.

3. **Macro Average Precision (MAP)**:

$$\text{Macro Precision} = \frac{1}{C} \sum_{i=1}^{C} \text{Precision}_i$$

Where:

- $C$ is the number of classes;
- $\text{Precision}_i$ is the precision for class $i$.

Macro Average Precision ranges from 0 to 1 (1 being the best), and calculates the mean precision across all classes without weighting by class frequency.

4. **Weighted Average Precision (WAP)**:

$$\text{Weighted Precision} = \sum_{i=1}^{C} w_i \cdot \text{Precision}_i$$

Where $w_i$ is the proportion of instances in class $i$.

Weighted Average Precision ranges from 0 to 1 (1 being the best), and accounts for class imbalance by weighting each class's precision by its support.

5. **Macro Average Recall (MAR) / Balanced Accuracy**:

$$\text{Macro Recall} = \frac{1}{C} \sum_{i=1}^{C} \text{Recall}_i$$

Where $\text{Recall}_i$ is the Recall for class $i$. Macro Average Recall ranges from 0 to 1 (1 being the best), and computes the unweighted mean recall across all classes.

6. **Weighted Average Recall (WAR) / Accuracy**:

$$\text{Weighted Recall} = \sum_{i=1}^{C} w_i \cdot \text{Recall}_i$$

Weighted Average Recall ranges from 0 to 1 (1 being the best), and is similar to Macro Recall, but weighted by class proportions.

7. **Macro Average F1 Score (MAF1)**:

$$\text{Macro F1} = \frac{1}{C} \sum_{i=1}^{C} F1_i$$

Where $F1_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$. The Macro Average F1 Score ranges from 0 to 1 (1 being the best), and is the mean of F1 scores computed for each class separately.

8. **Weighted Average F1 Score (WAF1)**:

$$\text{Weighted F1} = \sum_{i=1}^{C} w_i \cdot F1_i$$

The Weighted F1 Score ranges from 0 to 1 (1 being the best), and weights each class's F1 score by its class proportion.

# 3 Data Preparation

In order to benchmark the NN architectures against the four datasets presented in the previous section, they first needed to be preprocessed.

The preprocessing began on the smallest of the four datasets, which was the **Shuttle** dataset. Its original version published on the UCI Machine Learning Repository was already divided into a training and testing dataset, with the former accounting for 80% of the total uploaded volume and the latter for 20% of it. The original datasets already had no intersections, meaning a reduced risk of skewed benchmarking results during testing due to the model being applied to the same data as in the training dataset.

This commonly used 80-20 training-testing split is well-supported in machine learning research, as it typically offers a balance between having enough data to train a model while still providing a significant holdout set to evaluate its performance effectively [BCH24] [Tan+21]. In order to pursue a uniform approach to preprocessing the four datasets, it was decided to first combine the two datasets in order to perform normalization, and then to split the dataset back into the 80-20 training-testing form.

Since the original (now combined) dataset only included numerical features, it was sufficient to standardize these features $z$ by removing the mean and scaling to unit variance:

$$z = \frac{x - u}{s}$$

Where:

- $x$ is an original value of a given feature in a particular row;
- $u$ is the mean of all samples of the given feature;
- $s$ is the standard deviation of the training samples.

This centering and scaling all happen independently on each feature of the dataset.

Data normalization is a crucial preprocessing step in training NNs, as it significantly enhances the model's learning efficiency and overall performance. By normalizing data, typically with a mean of zero and standard deviation of one, NNs can train faster and achieve better generalization [SS97]. This effect occurs because normalization helps balance feature scales, ensuring that the optimization process in deep learning models doesn't become biased through features with larger ranges–the opposite often leading to unstable or poor convergence in most modern NN architectures [Sha+20].

After these two steps had been performed, the original dataset was transformed into two datasets, training and testing. The data was split in a stratified fashion, making sure that both datasets had the exact same distribution of class labels as the original combined dataset. The resulting distributions can be observed in Table 1. The amount of feature columns remained unchanged at a total of 9.

After the Shuttle dataset, the **Covertype** dataset was normalized. It had two kinds of features: numerical features, much like in the Shuttle dataset, and categorical features.

| Shuttle Train Set | | |
|---|---|---|
| **Label** | **%** | **Count** |
| 1 | 78,59 | 36469 |
| 4 | 15,35 | 7122 |
| 5 | 5,63 | 2614 |
| 3 | 0,30 | 137 |
| 2 | 0,09 | 40 |
| 7 | 0,02 | 10 |
| 6 | 0,02 | 8 |
| **Total** | **100%** | **46400** |

| Shuttle Test Set | | |
|---|---|---|
| **Label** | **%** | **Count** |
| 1 | 78,59 | 9117 |
| 4 | 15,35 | 1781 |
| 5 | 5,63 | 653 |
| 3 | 0,29 | 34 |
| 2 | 0,09 | 10 |
| 7 | 0,03 | 3 |
| 6 | 0,02 | 2 |
| **Total** | **100%** | **11600** |

Table 1: Shuttle Dataset Label Distributions

These latter features were already preprocessed using one-hot encoding, meaning that each "feature" like e.g. soil type was split into a number of different columns, where, for each particular row, only one of these columns' values was set to 1 (i.e. the row had that particular soil type), while all other columns in that row were set to 0. One-hot encoding is widely used because it allows NNs to interpret categorical variables as binary vectors, avoiding the misleading ordinal relationship that other encoding methods, like label encoding, could impose [PK23]. While this method may lead to a large expansion in the dataset's volume in some cases, as will be seen when preprocessing another dataset, it still remains indispensable for most NNs that work with numerical inputs [Fan+21], which made it the encoding of choice in this work as well.

In addition to passing over the one-hot encoded features to the resulting datasets, the numerical features were normalized in accordance with the aforementioned method. After this, the full dataset was split according to the 80-20 rule. The resulting distribution can be seen in Table 2. In total, the resulting datasets had 54 feature columns.

The **KDD-99** dataset provided a way larger challenge during preprocessing. The overwhelming majority of the dataset (78%) was taken up by just two classes which represented different kinds of online attacks: Smurf attacks (57%) and Neptune, or SYN Flood, attacks (21%). In contrast, connections of the normal kind amounted to just under 20%, with all other >20 classes all accounting for less than 0,4% of the entire volume each. While this distribution could be interesting in other development scenarios, the technical limitations made this original dataset too unwieldy to be processed on user-grade machines used in this work. As such, the first step was to remove all rows of the original dataset which represented these two most common attack types, leading to the "normal" class of values now being the overwhelming majority while also reducing the total volume of the dataset, which would speed up later benchmarking. Despite this cut in size, the KDD-99 dataset still remained the largest of all four.

The complete dataset was then preprocessed using the aforementioned steps. Nu-

**Covertype Train Set**

| Label | % | Count |
|---|---|---|
| 2 | 48,77 | 226640 |
| 1 | 36,46 | 169472 |
| 3 | 6,15 | 28603 |
| 7 | 3,53 | 16408 |
| 6 | 2,99 | 13894 |
| 5 | 1,63 | 7594 |
| 4 | 0,47 | 2198 |
| **Total** | **100%** | **464809** |

**Covertype Test Set**

| Label | % | Count |
|---|---|---|
| 2 | 48,77 | 56661 |
| 1 | 36,46 | 42368 |
| 3 | 6,15 | 7151 |
| 7 | 3,53 | 4102 |
| 6 | 2,99 | 3473 |
| 5 | 1,63 | 1899 |
| 4 | 0,47 | 549 |
| **Total** | **100%** | **116203** |

Table 2: Covertype Dataset Label Distributions

merical features were normalized, while the categorical features were one-hot encoded.

Due to the underrepresentation of some class labels, it was also found that a simple 80-20 split along the class labels did not assure that each dataset would have at least one representative of each class label. As such, for all classes that amounted to 3 or less total entries, a list of rare classes was defined. For these rare classes, half of the total volume was transferred to the training set, while the other half (i.e. 1 row) was transferred to the testing set. Due to the specifics of training NNs using Tensorflow [Mar+15], the library used in this work, it was necessary to make sure that each class had at least 2 representatives in the training dataset.

In total, the datasets resulting from these modifications had 126 feature columns, making them the largest in terms of both size (amount of data points) and width (amount of features).

Table 3 clearly demonstrates that these preprocessing steps, as much as they could, assured a representation of each class in both datasets.

Lastly, the **Darknet** dataset was preprocessed.

The original dataset included 2 columns for class labels: the first column indicated whether the connection type was "normal" (e.g. Non-Tor, Non-VPN etc.) or stemming from the Darknet, while the second column indicated which type of connection it was (VOIP, audio streaming, file transfer, peer-to-peer connections, etc.). Since the NNs being benchmarked are designed with the intention of predicting only one class label, it was necessary to combine these two columns into one. In the end, the normal connection types were all placed into one class named "Normal", while for the Darknet connection types the two label columns were combined, resulting in classes like "Darknet_Audio-Streaming" and the like.

In addition, the original dataset also included timestamps. While the NNs benchmarked in the course of this work are not designed to process time series data, focusing instead on numerical commonalities between different data points that indicate their

| KDD-99 Train Set | | | KDD-99 Test Set | | |
|---|---|---|---|---|---|
| **Label** | **%** | **Count** | **Label** | **%** | **Count** |
| normal. | 95,50 | 778224 | normal. | 95,50 | 194557 |
| satan. | 1,56 | 12714 | satan. | 1,56 | 3178 |
| ipsweep. | 1,23 | 9985 | ipsweep. | 1,23 | 2496 |
| portsweep. | 1,02 | 8330 | portsweep. | 1,02 | 2083 |
| nmap. | 0,23 | 1853 | nmap. | 0,23 | 463 |
| back. | 0,22 | 1762 | back. | 0,22 | 441 |
| warezclient. | 0,10 | 816 | warezclient. | 0,10 | 204 |
| teardrop. | 0,10 | 783 | teardrop. | 0,10 | 196 |
| pod. | 0,03 | 211 | pod. | 0,03 | 53 |
| guess _passwd. | 0,01 | 42 | guess _passwd. | 0,01 | 11 |
| buffer _overflow. | < 0,01 | 24 | buffer _overflow. | < 0,01 | 6 |
| land. | < 0,01 | 17 | land. | < 0,01 | 4 |
| warezmaster. | < 0,01 | 16 | warezmaster. | < 0,01 | 4 |
| imap. | < 0,01 | 10 | imap. | < 0,01 | 2 |
| rootkit. | < 0,01 | 8 | rootkit. | < 0,01 | 2 |
| loadmodule. | < 0,01 | 7 | loadmodule. | < 0,01 | 2 |
| multihop. | < 0,01 | 6 | multihop. | < 0,01 | 1 |
| ftp _write. | < 0,01 | 6 | ftp _write. | < 0,01 | 2 |
| phf. | < 0,01 | 3 | phf. | < 0,01 | 1 |
| perl. | < 0,01 | 2 | perl. | < 0,01 | 1 |
| spy. | < 0,01 | 2 | spy. | < 0,01 | 1 |
| **Total** | **100%** | **814821** | **Total** | **100%** | **203708** |

Table 3: KDD-99 Dataset Label Distributions

belonging to a particular class, this information can be useful if some classes have temporal seasonality in them (e.g. if, for example, "Darknet_Browsing" occurs mostly during the evenings). In order for models to be able to learn this seasonality and periodicity, it is important to break down timestamps into interpretable granular features [DT18], which is precisely what was done to this original feature of the Darknet dataset.

In addition, the usual preprocessing steps of one-hot encoding and normalization were also performed on the categorical and numerical features, respectively.

After one-hot encoding was performed initially, it was noticed that the size of the dataset exploded exponentially. This was due to the fact that some of the original features (namely "Flow ID", "Src IP", and "Dst IP") had a huge number of different values. Since these features had a large cardinality (i.e. there was a large variety of different

IP addresses) and were encoded as categorical using one-hot encoding, the number of columns of the resulting dataset would have been impossibly large, while also not providing enough useful information to the model. Since deep models (like the ones being benchmarked here) tend to underperform on such high-cardinal data [Sig23], it was decided to reduce the cardinality of these three features and only then one-hot encode them. 1% was taken as the threshold: if a certain value of a given feature appeared in less than 1% of the entire dataset, it would be combined into the new value "Other". As all three features have limited expressive capacity, serving more as unique device identifiers useful mostly for indicating suspicious repeat connections, it was decided that these features' cardinality could be reduced using the 1% rule without sacrificing model accuracy.

After all preprocessing steps, the resulting datasets then included 103 different encoded feature columns and one class label column. Table 4 shows the class distribution in the training and test datasets.

**Darknet Train Set**

| Label | % | Count |
|---|---|---|
| Normal | 84,67 | 107443 |
| DN _AS[1] | 8,37 | 10627 |
| DN _C[2] | 2,86 | 3633 |
| DN _FT[3] | 1,65 | 2088 |
| DN _VOIP[4] | 0,92 | 1172 |
| DN _VS[5] | 0,85 | 1077 |
| DN _EM[6] | 0,37 | 466 |
| DN _BR[7] | 0,17 | 210 |
| DN _P2P[8] | 0,14 | 176 |
| **Total** | **100%** | **126892** |

**Darknet Test Set**

| Label | % | Count |
|---|---|---|
| Normal | 84,67 | 26862 |
| DN _AS[1] | 8,37 | 2657 |
| DN _C[2] | 2,86 | 908 |
| DN _FT[3] | 1,65 | 522 |
| DN _VOIP[4] | 0,92 | 293 |
| DN _VS[5] | 0,85 | 269 |
| DN _EM[6] | 0,37 | 116 |
| DN _BR[7] | 0,17 | 53 |
| DN _P2P[8] | 0,14 | 44 |
| **Total** | **100%** | **31724** |

Table 4: Darknet Dataset Label Distributions
The label names were abbreviated to fit the page size. The full names are:
[1] Darknet_Audio-Streaming, [2] Darknet_Chat, [3] Darknet_File-Transfer, [4] Darknet_VOIP,
[5] Darknet_Video-Streaming, [6] Darknet_Email, [7] Darknet_Browsing, [8] Darknet_P2P

In the end, it was possible to convert all four datasets into a befitting form to be processed by the different models, while also achieving an imbalanced distribution of all data. It should also be noted that the imbalance varies from dataset to dataset. In some datasets, like KDD-99, the majority class takes up more than 95% of all data, while in others, like Covertype, that number is only 48%. Figure 6 and Figure 8 show the diversity of these four datasets in form of pie charts.

This allows to test the developed deep models on a varied set of imbalanced class distributions, thus helping develop a more robust architecture than if the benchmark datasets all had the same distribution.

# 4 Model Training

After the datasets were successfully preprocessed, attention was directed toward the conceptualization and training of a suitable NN architecture.

The foundational inspiration for the architecture was derived from the work presented in [Jia+23]. In their research, Jiang et al. proposed an innovative hybrid approach that combined NNs for feature extraction with BAs for subsequent classification tasks, leveraging convolutional layers to extract spatial features. However, this methodology was primarily designed for image-based datasets. For structured tabular datasets, such as those considered in this thesis, their proposed framework relied solely on boosting algorithms like XGBoost or CatBoost, bypassing NNs entirely.

Despite this limitation in their application to structured data, the idea of employing NNs as powerful feature extractors held significant promise. The ability of NNs to learn complex, non-linear relationships from raw data made them an attractive candidate for extending Jiang et al.'s approach beyond its original scope. It was hypothesized that incorporating a NN-based feature extraction pipeline into a model designed specifically for tabular data could lead to improved performance when dealing with highly imbalanced class distributions.

Thus, it was decided to adapt and extend this concept to structured datasets in order to explore its potential benefits in scenarios where class imbalance is prevalent. Unlike traditional Machine Learning (ML) models or standalone BAs–which often require extensive manual feature engineering–the use of NNs allows for automated representation learning.

Three different hardware configurations were used for training and benchmarking the resulting models:

1. Configuration 1 (Laptop):
    - Central Processing Unit (CPU): AMD® Ryzen™ 7 3750H (Base speed: 2,30 Gigahertz (GHz))
    - RAM: 16 Gigabyte (GB)
2. Configuration 2 (Laptop):
    - CPU: Intel® Core™ i9-9980HK (Base speed: 2,40 GHz)
    - RAM: 64 GB
3. Configuration 3 (Kaggle Server, specifications recorded at the time of benchmarking and writing):
    - CPU: Intel® Xeon® (Base speed: 2,20 GHz)
    - RAM: 32 GB

Graphics Processing Unit (GPU) acceleration was not used in the scope of this work in order to better replicate the lowest common denominator on which these architectures could be trained in real business applications–i.e. office computers and/or servers without advanced hardware.

Despite the seeming differences between the hardware combinations, these do not impact the quality of the trained models in any way, shape or form, with the only differences being the runtime needed to train the models, and the RAM available for loading the datasets and training the models.

The construction of the NN model was carried out using the Python programming language in version 3.12.3 [VD95]. The two main libraries used to develop the model architecture were `tensorflow` in version 2.18.0 [Mar+15] and `scikit-learn` in version 1.5.2 [Ped+11], with the other notable auxiliary libraries being `numpy` in version 1.26.4 [Har+20] and `pandas` in version 2.2.3 [McK+10]. The established nature and extensive support of Python and the used libraries made this ecosystem an ideal choice for implementing and experimenting with the NN architecture. Aside from offering a high-level framework (Keras) for defining the general architecture, this ecosystem also offers lower-level functionalities that enable granular control over every aspect of model development.

To begin the exploration phase, it was first decided to construct a rudimentary NN architecture. The primary objective was to investigate the feasibility of random feature extraction from structured datasets, without yet applying BAs for final classification. Unlike traditional approaches that rely on predefined or domain-specific feature engineering, this architecture aimed to leverage randomness as a mechanism for uncovering latent patterns within structured data.

The core innovation lay in the use of rudimentary custom layers, which in this work will be coined as *"Random Mixing Layers"*. These layers were designed to extract random combinations of features rather than adhering to the conventional sliding window approach commonly employed in image-based datasets. In image processing tasks, sliding windows operate on neighboring pixels due to their inherent spatial relationships: they extract the color values from these pixels and subsequently apply the required transformations to extract some numerical feature from them. However, structured datasets often exhibit correlations between non-adjacent columns–relationships that are not easily captured by such methods. By allowing for the combination of features located at different ends of a dataset, this architecture sought to exploit these potential dependencies and ensure better generalization.

The architecture consisted of multiple layers, each serving distinct roles:

1. *Input Layer*: As in every NN, the input layer was responsible for loading the data and feeding it to the next layer below it.

2. *Random Mixing Layers*: Next were two custom layers that generated random combinations of input features. Each layer randomly selected three features from the output produced by the previous layer and combined them into new representations. This process was repeated 20 times per layer, resulting in 20 unique feature

groups per iteration. These groups were then passed forward for further transformations.

3. *Aggregation Layer*: Following these initial stages of randomization came an aggregation step where the outputs from all 20 feature groups were concatenated into a single unified vector representation.

4. *Feature Refinement Layers*: To enhance representational power post-aggregation, three fully connected dense layers were introduced. These layers employed ReLU activation functions to refine extracted features further.

5. *Output Layer*: Finally came the output layer tasked with producing class predictions via softmax activation functions applied across $C$ distinct categories corresponding directly with benchmarked dataset labels.

In initial attempts at construction of the NN, dropout layers were also utilized: dropout randomly deactivates neurons during training iterations based on specified probabilities (set initially to 20%), thereby encouraging robustness. It was quickly found that, due to the imbalanced nature of the datasets, this approach only further limited the efficacy of the NN, tossing out valuable rare correlations that may have indicated belonging to one of the underrepresented classes.

Training proceeded using an 80-20 train-validation split performed upon the preprocessed training datasets. This facilitated real-time performance monitoring: validation loss was used as a stopping criterion, whereby the NN would halt training if the loss were to not improve over 10 training epochs.

The batch size was initially set to 32.

Due to the inherently randomized nature of training NNs, especially using the random mixing layers described above, each architecture was benchmarked multiple times. This approach was applied to each of the subsequent 39 architecture versions, which allowed to gain a more accurate evaluation of the performance metrics and more insight into the real efficacy of each model. The amount of such executions for each architecture version and each dataset can be further observed in Table 47.

Table 5 shows the values of the eight different performance metrics for the first architecture iteration, as well as the average values for each dataset and over all 32 resulting metrics.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9694 | 0.9170 | 0.9690 | 0.9699 | 0.9688 | 0.5686 | 0.5501 | 0.5289 | 0.8052 |
| Covertype | 0.4718 | 0.1506 | 0.4771 | 0.5338 | 0.4129 | 0.2434 | 0.1912 | 0.1617 | 0.3303 |
| KDD-99 | 0.9573 | 0.3463 | 0.9364 | 0.9640 | 0.9476 | 0.1046 | 0.0848 | 0.0851 | 0.5533 |
| Darknet | 0.8501 | 0.2384 | 0.7818 | 0.8731 | 0.8186 | 0.2611 | 0.2476 | 0.2354 | 0.5383 |
| Overall | | | | | | | | | 0.5568 |

Table 5: Performance Metrics for Architecture Version 1

The performance of the architecture showed to be promising, though also very much open for improvement. This improvement would come in later iterations.

Initially, it was found that the training time for this architecture was much too large– for faster comparisons between different architectures, the training approach had to be optimized. As was mentioned previously in subsubsection 2.4.1, the batch size can drastically impact the training speed. Smaller batch sizes, while generally better, can result in much longer training times. It was thus attempted to increase the batch sizes in order to strike the best balance between model performance and training speed.

In the second iteration, the batch size was increased to 256. Table 6 shows the performance of the model after this change.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9297 | 0.7963 | 0.9242 | 0.9338 | 0.9251 | 0.4900 | 0.4227 | 0.4400 | 0.7327 |
| Covertype | 0.4814 | 0.1630 | 0.4395 | 0.5393 | 0.4272 | 0.2649 | 0.2042 | 0.1770 | 0.3371 |
| KDD-99 | 0.9539 | 0.2695 | 0.9344 | 0.9609 | 0.9437 | 0.1176 | 0.0736 | 0.0763 | 0.5412 |
| Darknet | 0.8460 | 0.2680 | 0.7833 | 0.8656 | 0.8195 | 0.1839 | 0.1725 | 0.1720 | 0.5139 |
| Overall | | | | | | | | | 0.5312 |

Table 6: Performance Metrics for Architecture Version 2

It was found that the training took much less time, speeding up the process by a factor of approximately 6,5 (though mileage may vary on other machines). As can be seen in Table 6, the performance trade-off, while existent, was tolerable.

For the sake of completeness, however, other batch sizes were also tried. The third version of the architecture attempted to utilize BGD, i.e. to utilize the entire dataset in only one training batch. The results can be seen in Table 7.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.3524 | 0.1331 | 0.4671 | 0.3784 | 0.3385 | 0.2257 | 0.2220 | 0.1610 | 0.2848 |
| Covertype | 0.2802 | 0.0009 | 0.2873 | 0.3213 | 0.2462 | 0.1300 | 0.1358 | 0.0804 | 0.1853 |
| KDD-99 | 0.9443 | 0.0167 | 0.9127 | 0.9527 | 0.9322 | 0.0461 | 0.0515 | 0.0475 | 0.4880 |
| Darknet | 0.5732 | 0.0998 | 0.5972 | 0.5815 | 0.5745 | 0.1158 | 0.1292 | 0.1001 | 0.3464 |
| Overall | | | | | | | | | 0.3261 |

Table 7: Performance Metrics for Architecture Version 3

Additionally, it was attempted to instead make the batch size equal 1% of the entire dataset. This had the added benefit of dataset-agnosticism: the model would thus be able to automatically set the batch size by itself depending on the total amount of entries in the dataset. The motivation behind this idea was that e.g. a static batch size of 256 would perform better on a large dataset (where it would constitute but a small amount of training samples) than in a dataset of 1000 total entries (where it would constitute 25% of the total dataset). As observed, smaller batch sizes usually tend to lead to bet-

ter results [Kes+16]–the 1% approach would thus make sure that the batch size would always be relatively small. The results of this can be observed in Table 8.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9515 | 0.8673 | 0.9524 | 0.9538 | 0.9494 | 0.5147 | 0.4601 | 0.4701 | 0.7649 |
| Covertype | 0.5184 | 0.2114 | 0.4554 | 0.5685 | 0.4695 | 0.2899 | 0.2265 | 0.2038 | 0.3679 |
| KDD-99 | 0.9512 | 0.1373 | 0.9219 | 0.9590 | 0.9399 | 0.0616 | 0.0648 | 0.0623 | 0.5123 |
| Darknet | 0.8755 | 0.4421 | 0.8364 | 0.8910 | 0.8550 | 0.3138 | 0.2582 | 0.2638 | 0.5920 |
| Overall | | | | | | | | | 0.5593 |

Table 8: Performance Metrics for Architecture Version 4

Another concept that came up was the concept of *stratified batching*. The main idea behind stratified batching is to ensure that each batch of data used for training contains a representative distribution of the different classes present in the training dataset.

The 5th architecture iteration attempted to introduce this concept and evaluate its usefulness. Due to the nature of the original datasets, it was necessary to set the batch size to 10% of the original dataset–otherwise, the approach refused to work on such overwhelmingly imbalanced datasets as KDD-99. Table 9 shows the performance of the model after this concept was introduced.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.1906 | 0.0381 | 0.2820 | 0.2246 | 0.1745 | 0.0678 | 0.1432 | 0.0508 | 0.1464 |
| Covertype | 0.2714 | 0.0048 | 0.2482 | 0.2986 | 0.2524 | 0.0878 | 0.1491 | 0.0908 | 0.1754 |
| KDD-99 | 0.7167 | -0.0178 | 0.7273 | 0.7148 | 0.7201 | 0.0363 | 0.0452 | 0.0359 | 0.3723 |
| Darknet | 0.3119 | 0.0449 | 0.4574 | 0.3153 | 0.3337 | 0.0626 | 0.1119 | 0.0487 | 0.2108 |
| Overall | | | | | | | | | 0.2262 |

Table 9: Performance Metrics for Architecture Version 5

As can be seen above, the quality of the classification using the stratified batch approach dropped drastically. Indeed, as can be observed from the negative MCC for KDD-99, the classification even managed to sometimes be worse than simple random chance.

After the approach failed, it was attempted to test another percentage-based batch size. This time, the batch size was set to 5% of the dataset: the results of this decision can be observed in Table 10.

After running benchmarks for all these different batch sizes, it was clear that the smallest batch size, 32, was the best batch size out of all the "static" batch sizes. Generalizing over all metrics, however, it was clear that using a batch size of 1% was overall the best approach–both in terms of predictive quality and in terms of runtime. Therefore, the following iterations all use a batch size of 1%.

In the next few iterations, an attempt was made at combating the imbalanced nature of the datasets by applying rebalancing mechanisms. Though other resampling mech-

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8431 | 0.4495 | 0.8000 | 0.8649 | 0.8165 | 0.3845 | 0.3525 | 0.3545 | 0.6082 |
| Covertype | 0.3959 | 0.0602 | 0.3496 | 0.4712 | 0.3286 | 0.1538 | 0.1563 | 0.1146 | 0.2538 |
| KDD-99 | 0.9501 | 0.1473 | 0.9235 | 0.9581 | 0.9384 | 0.0836 | 0.0605 | 0.0625 | 0.5155 |
| Darknet | 0.8603 | 0.2846 | 0.7902 | 0.8814 | 0.8314 | 0.2487 | 0.2377 | 0.2350 | 0.5462 |
| Overall | | | | | | | | | 0.4809 |

Table 10: Performance Metrics for Architecture Version 6

anisms will yet be observed in later architecture versions, the next few ones dealt with the standard SMOTE mechanism.

A unique challenge was posed by the datasets benchmarked in this work. The imbalance between the different classes was so drastic that SMOTE and other similar approaches, if applied without limitations, all ran the risk of introducing too many artifacts into the dataset: after all, the most underrepresented class in the KDD-99 dataset constitutes only 2 entries, which is exactly 389112 times less than the largest class. If one were to apply SMOTE to the dataset without any limitations and/or downsampling of the largest class, an introduction of more than 700 thousand entries of such a class would inevitably lead to superfluous entries. What's more, such a lack of limitations on oversampling would blow up the dataset's size by more than a factor of 20 (in the case of KDD-99), which would in turn prohibit any kind of training on common user-grade machines. And, since the goal of this work is to develop a general approach that can be applied in any use-case regardless of the dataset's nature, this kind of edge case had to be kept in mind.

The 7th architecture version thus attempted to solve this problem via downsampling: the largest class would be shrunk by a factor of 2, and then SMOTE would be applied to oversample the other classes. The results of this can be seen in Table 11.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7198 | 0.6710 | 0.9400 | 0.7138 | 0.7422 | 0.5578 | 0.7252 | 0.5394 | 0.7011 |
| Covertype | 0.2765 | 0.1305 | 0.4562 | 0.2804 | 0.3022 | 0.2117 | 0.3385 | 0.1786 | 0.2718 |
| KDD-99 | 0.0095 | 0.0863 | 0.3870 | 0.0132 | 0.0100 | 0.0498 | 0.1453 | 0.0358 | 0.0921 |
| Darknet | 0.0934 | 0.1287 | 0.5423 | 0.0954 | 0.1239 | 0.1722 | 0.2912 | 0.1034 | 0.1938 |
| Overall | | | | | | | | | 0.3147 |

Table 11: Performance Metrics for Architecture Version 7

The performance was less than optimal. It was hypothesized that there was still a large issue with model hallucination and introduction of artifacts into the dataset via extensive oversampling. In version 8, therefore, a more refined approach was attempted: instead of halving the size of the largest class, this class would instead be downscaled to the size of the second largest one, after which SMOTE would be applied again. Table 12 shows the performance metrics calculated from testing this architecture on the

four used datasets.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8607 | 0.7208 | 0.9509 | 0.8465 | 0.8880 | 0.5977 | 0.8396 | 0.6287 | 0.7916 |
| Covertype | 0.2257 | 0.0631 | 0.3993 | 0.2438 | 0.2269 | 0.1913 | 0.2216 | 0.1238 | 0.2119 |
| KDD-99 | 0.0096 | 0.0175 | 0.3836 | 0.0131 | 0.0129 | 0.0260 | 0.0853 | 0.0063 | 0.0693 |
| Darknet | 0.1418 | 0.1450 | 0.5453 | 0.1461 | 0.1685 | 0.2162 | 0.4276 | 0.1790 | 0.2462 |
| Overall | | | | | | | | | 0.3298 |

Table 12: Performance Metrics for Architecture Version 8

A noticeable improvement could be observed. It was further hypothesized that, instead of downscaling the largest class, a limit could be placed on the upscaling of all other underrepresented classes. Here, the underrepresented classes would be upscaled (at most) by a factor of 10000: the upscaling of a class, like in SMOTE, would halt once the underrepresented class' size reaches that of the largest class. This 9th architecture's performance is documented in Table 13.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.6816 | 0.4884 | 0.8802 | 0.6738 | 0.7182 | 0.4777 | 0.7446 | 0.4918 | 0.6445 |
| Covertype | 0.1552 | 0.0575 | 0.5017 | 0.1488 | 0.1928 | 0.2027 | 0.2458 | 0.1069 | 0.2014 |
| KDD-99 | 0.0141 | 0.0822 | 0.3692 | 0.0171 | 0.0166 | 0.0616 | 0.1421 | 0.0388 | 0.0927 |
| Darknet | 0.3376 | 0.2375 | 0.7321 | 0.3314 | 0.3877 | 0.2764 | 0.4894 | 0.2447 | 0.3796 |
| Overall | | | | | | | | | 0.3296 |

Table 13: Performance Metrics for Architecture Version 9

The performance was promising. There was little to no drop in efficiency as compared to the previous architecture, while the runtime was noticeably shortened via this new limitation on upscaling. More experimentation followed: an even smaller factor of 5000 was tested in the 10th version, which was subsequently benchmarked with the results shown in Table 14.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8388 | 0.7159 | 0.9504 | 0.8232 | 0.8702 | 0.6000 | 0.8140 | 0.6123 | 0.7781 |
| Covertype | 0.1663 | 0.0576 | 0.3399 | 0.1715 | 0.1899 | 0.1589 | 0.2315 | 0.1063 | 0.1777 |
| KDD-99 | 0.1025 | 0.0720 | 0.5775 | 0.0948 | 0.1369 | 0.0824 | 0.1965 | 0.0689 | 0.1664 |
| Darknet | 0.2725 | 0.1380 | 0.6696 | 0.2827 | 0.2912 | 0.1591 | 0.3864 | 0.1255 | 0.2906 |
| Overall | | | | | | | | | 0.3532 |

Table 14: Performance Metrics for Architecture Version 10

Finally, as shown in Table 15, an attempt was made to further reduce the upscaling factor, this time to 2500.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.6441 | 0.4562 | 0.8738 | 0.6296 | 0.6803 | 0.4095 | 0.5935 | 0.4014 | 0.5860 |
| Covertype | 0.1889 | 0.0522 | 0.3572 | 0.2052 | 0.1967 | 0.1881 | 0.2307 | 0.1177 | 0.1921 |
| KDD-99 | 0.0039 | 0.0429 | 0.1972 | 0.0091 | 0.0040 | 0.0654 | 0.1323 | 0.0371 | 0.0615 |
| Darknet | 0.4155 | 0.2156 | 0.8697 | 0.3933 | 0.5002 | 0.2217 | 0.4628 | 0.1980 | 0.4096 |
| Overall | | | | | | | | | 0.3123 |

Table 15: Performance Metrics for Architecture Version 11

5000 ended up being the most optimal upscaling factor for SMOTE-like approaches, which would be returned to in later architecture versions.

In the 12th version, another rebalancing approach was considered that was described earlier in subsubsection 2.4.4. Instead of using a simple unweighted categorical cross-entropy loss, the architecture would compute the class weights based on their distribution in the dataset, and then pass them to the training function. This would automatically modify the cross-entropy loss and turn it into its weighted counterpart. In the next, 13th architecture iteration, the attempt was made to instead define this weighted categorical cross-entropy loss function by hand. A comparison of how these two approaches performed can be made upon observing Table 16 and Table 17.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8192 | 0.5858 | 0.8995 | 0.8130 | 0.8346 | 0.4632 | 0.7009 | 0.4773 | 0.6992 |
| Covertype | 0.1712 | 0.0647 | 0.3394 | 0.1997 | 0.1641 | 0.2046 | 0.2158 | 0.1201 | 0.1850 |
| KDD-99 | 0.0525 | 0.0257 | 0.3825 | 0.0485 | 0.0718 | 0.0533 | 0.1026 | 0.0308 | 0.0960 |
| Darknet | 0.4152 | 0.2286 | 0.8635 | 0.4096 | 0.4838 | 0.2035 | 0.3623 | 0.1795 | 0.3932 |
| Overall | | | | | | | | | 0.3433 |

Table 16: Performance Metrics for Architecture Version 12

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8272 | 0.6684 | 0.9451 | 0.8111 | 0.8607 | 0.4903 | 0.6652 | 0.4857 | 0.7192 |
| Covertype | 0.1032 | 0.0555 | 0.4852 | 0.1248 | 0.1125 | 0.2091 | 0.2213 | 0.0779 | 0.1737 |
| KDD-99 | 0.0468 | 0.0676 | 0.3808 | 0.0442 | 0.0636 | 0.0479 | 0.1268 | 0.0273 | 0.1006 |
| Darknet | 0.1885 | 0.1234 | 0.8568 | 0.1870 | 0.2362 | 0.1934 | 0.3736 | 0.1215 | 0.2851 |
| Overall | | | | | | | | | 0.3196 |

Table 17: Performance Metrics for Architecture Version 13

The weighted categorical cross-entropy loss has, on average, performed worse than SMOTE. Architecture version 14 utilized another loss function instead, namely FL. As explained earlier in subsubsection 2.4.4, it not only grants a larger weight to underrepresented classes, but in fact also down-weights easily classifiable classes. And, as can be

observed from the performance of this approach in Table 18, FL was indeed the better loss function.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8804 | 0.6456 | 0.8742 | 0.8919 | 0.8665 | 0.4017 | 0.3368 | 0.3477 | 0.6556 |
| Covertype | 0.4736 | 0.1437 | 0.4186 | 0.5339 | 0.4170 | 0.2132 | 0.1895 | 0.1596 | 0.3186 |
| KDD-99 | 0.9462 | 0.0036 | 0.9127 | 0.9551 | 0.9331 | 0.0479 | 0.0476 | 0.0465 | 0.4866 |
| Darknet | 0.8315 | 0.1260 | 0.7582 | 0.8572 | 0.7965 | 0.1971 | 0.1407 | 0.1405 | 0.4810 |
| Overall | | | | | | | | | 0.4854 |

Table 18: Performance Metrics for Architecture Version 14

After experimenting with different mechanisms for balancing out the classification, it was time to pivot the experimentation into another area. It was decided to first continue work on the architectures utilizing weighted categorical cross-entropy and FL.

Validation, while a useful mechanism in balanced datasets that promotes generalization and tests model quality along the way, has not yet been investigated properly in the scope of this work by the current point. The only validation split utilized in the previous architecture versions was a simple 80-20 one-time on the training set. There was, however, a large problem about this approach that became apparent at that point: the split was not stratified, meaning that the model was essentially under constant risk of training on incomplete data that did not include all classes. In other words, the training set may have had classes that were not represented in the validation set, and vice versa, which could have been reason for the middling performance of the previous architectures.

An alternative was to be developed. Instead of applying only a single 80-20 validation split, a decision was made to instead try *k-fold validation splitting*. For e.g. $k = 5$, this would mean splitting the dataset and training the model 5 separate times, while using a different 20% of the full dataset as the validation set in each of the 5 iterations. This way, the model would have the ability to train on the complete dataset, achieving higher robustness and predictive ability. Furthermore, the mechanism was expanded into a stratified 5-fold validation split, meaning that, in each of the 5 iterations, both the training and the validation sets were made to follow the class distribution of the full dataset being fed into the model.

Table 19 describes the quality of the resulting trained model after applying 5-fold validation splitting in combination with weighted cross-entropy categorical loss.

And Table 20 shows the results of the 5-fold validation split applied in conjunction with FL.

Naturally, training with a 5-fold validation split meant a way longer training time, since each model would, on average, have to train 5 times as long in order to perform validation on the entire training dataset. Following this, another hypothesis was made: due to the imbalanced nature of the benchmark datasets, even a stratified validation split was sometimes unable to place at least one sample of each class in both the training

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7872 | 0.5462 | 0.9056 | 0.7789 | 0.8094 | 0.4456 | 0.5599 | 0.4158 | 0.6561 |
| Covertype | 0.1765 | 0.0597 | 0.4768 | 0.1935 | 0.1855 | 0.2144 | 0.2360 | 0.1054 | 0.2060 |
| KDD-99 | 0.2971 | 0.0609 | 0.6992 | 0.2883 | 0.3278 | 0.0608 | 0.1094 | 0.0337 | 0.2346 |
| Darknet | 0.2726 | 0.1701 | 0.7931 | 0.2712 | 0.3149 | 0.1888 | 0.3530 | 0.1331 | 0.3121 |
| Overall | | | | | | | | | 0.3522 |

Table 19: Performance Metrics for Architecture Version 15

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9050 | 0.7147 | 0.8948 | 0.9126 | 0.8963 | 0.3935 | 0.3671 | 0.3732 | 0.6822 |
| Covertype | 0.4381 | 0.1028 | 0.4016 | 0.5083 | 0.3715 | 0.1915 | 0.1674 | 0.1334 | 0.2893 |
| KDD-99 | 0.9489 | 0.1051 | 0.9177 | 0.9571 | 0.9368 | 0.0622 | 0.0608 | 0.0602 | 0.5061 |
| Darknet | 0.8376 | 0.1888 | 0.7613 | 0.8615 | 0.8048 | 0.1635 | 0.1586 | 0.1535 | 0.4912 |
| Overall | | | | | | | | | 0.4922 |

Table 20: Performance Metrics for Architecture Version 16

and validation sets, leading to worse predictive quality. What if, instead of performing a validation split, one were to perform none at all, thus training the model on the entire dataset prepared for the process?

This hypothesis was tested in architecture versions 17 and 18 with weighted categorical cross-entropy loss and FL, respectively. Table 21 and Table 22 provide insights into the predictive quality of the resulting models.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7661 | 0.4689 | 0.8410 | 0.7597 | 0.7861 | 0.4567 | 0.5565 | 0.4315 | 0.6333 |
| Covertype | 0.2072 | 0.0691 | 0.3520 | 0.2353 | 0.2042 | 0.1768 | 0.2166 | 0.1045 | 0.1957 |
| KDD-99 | 0.0046 | 0.0574 | 0.5326 | 0.0121 | 0.0038 | 0.0571 | 0.1161 | 0.0314 | 0.1019 |
| Darknet | 0.1696 | 0.0905 | 0.8366 | 0.1707 | 0.2209 | 0.1539 | 0.3161 | 0.0742 | 0.2541 |
| Overall | | | | | | | | | 0.2962 |

Table 21: Performance Metrics for Architecture Version 17

These new architecture versions once again proved empirically that FL was the more suitable loss function, at least when used with a pure NN without subsequent BA classification, with performances improving by around 15% when considering the average over all metrics.

Interestingly, it was also found that a combination of FL with a lack of validation splitting provided greater performance than the same architecture *with* validation splitting. Together with the fact that, on average, training the model without validation splitting took 5 times less time than with validation splitting, it was concluded that the architecture was much better served being trained on the complete training dataset in one go. The performance improvement was more noticeable for the KDD-99 dataset,

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8808 | 0.6082 | 0.8605 | 0.8944 | 0.8645 | 0.4493 | 0.4037 | 0.4105 | 0.6715 |
| Covertype | 0.4232 | 0.0918 | 0.4658 | 0.5001 | 0.3523 | 0.2582 | 0.1548 | 0.1185 | 0.2956 |
| KDD-99 | 0.9533 | 0.2811 | 0.9274 | 0.9604 | 0.9429 | 0.0811 | 0.0751 | 0.0735 | 0.5369 |
| Darknet | 0.8454 | 0.2271 | 0.7725 | 0.8676 | 0.8148 | 0.1830 | 0.1661 | 0.1658 | 0.5053 |
| Overall | | | | | | | | | 0.5023 |

Table 22: Performance Metrics for Architecture Version 18

most likely due to the great imbalance between the classes that made even a stratified 80-20 split impossible, likely leading to missing classes in either the training or the validation set.

This proved the hypothesis that training on the full dataset prepared for training was the better option.

In order to check the other loss function considered in earlier versions, another version of the architecture, version 19, was created. This one was identical to the now best version, version 18, but used an unweighted categorical cross-entropy loss instead of FL. Its performance measurements can be seen in Table 23.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9295 | 0.7899 | 0.9292 | 0.9341 | 0.9245 | 0.6268 | 0.5597 | 0.5736 | 0.7834 |
| Covertype | 0.4343 | 0.0980 | 0.4467 | 0.4993 | 0.3720 | 0.1953 | 0.1660 | 0.1314 | 0.2929 |
| KDD-99 | 0.9501 | 0.1666 | 0.9195 | 0.9580 | 0.9384 | 0.0595 | 0.0633 | 0.0613 | 0.5146 |
| Darknet | 0.8621 | 0.3618 | 0.8163 | 0.8807 | 0.8377 | 0.3355 | 0.2308 | 0.2403 | 0.5706 |
| Overall | | | | | | | | | 0.5404 |

Table 23: Performance Metrics for Architecture Version 19

As can be seen, this architecture, contrary to earlier benchmarks, outperformed the last version that used FL on most metrics, with one exception being those measuring performance on the KDD-99 dataset. This could, however, be due to implementation difficulties: the hand-made FL function may have been not as well-optimized as those included into Tensorflow by default. Due to the lack of clarity as to which loss function was *the* best, the 2 loss functions–categorical cross-entropy and FL–will be compared in later iterations as well.

The next experiment would use yet another architecture in order to solidify some other hypotheses. Architecture version 20 would be identical to version 18, but with one tweak: it would use a batch size of 32 instead of 1%.

Table 24 shows the performance of this architecture.

This architecture once again proved that a utilization of smaller batch sizes leads to much better results: the average improvement of performance metrics constituted 4% in comparison to architecture version 18. Later architectures, due to time constraints, would still utilize the larger batch size of 1%, but this once again proves that, if provided

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9524 | 0.8602 | 0.9480 | 0.9554 | 0.9490 | 0.6407 | 0.5322 | 0.5656 | 0.8004 |
| Covertype | 0.4314 | 0.0867 | 0.3971 | 0.5012 | 0.3652 | 0.2195 | 0.1674 | 0.1384 | 0.2884 |
| KDD-99 | 0.9499 | 0.1730 | 0.9271 | 0.9577 | 0.9384 | 0.0925 | 0.0615 | 0.0610 | 0.5201 |
| Darknet | 0.8651 | 0.3955 | 0.8218 | 0.8812 | 0.8437 | 0.2611 | 0.2200 | 0.2212 | 0.5637 |
| Overall | | | | | | | | | 0.5432 |

Table 24: Performance Metrics for Architecture Version 20

with unlimited training time, the user would be better served utilizing smaller batch sizes.

For the sake of attaining a complete picture, architecture version 19 was modified to instead utilize 5-fold cross-validation, turning into architecture version 21 which can be analyzed using Table 25.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9075 | 0.7349 | 0.9063 | 0.9133 | 0.9011 | 0.4080 | 0.3705 | 0.3769 | 0.6898 |
| Covertype | 0.4360 | 0.0932 | 0.3983 | 0.5045 | 0.3712 | 0.1759 | 0.1672 | 0.1322 | 0.2848 |
| KDD-99 | 0.9494 | 0.1381 | 0.9197 | 0.9575 | 0.9378 | 0.0676 | 0.0647 | 0.0640 | 0.5123 |
| Darknet | 0.8312 | 0.1391 | 0.7495 | 0.8560 | 0.7972 | 0.1391 | 0.1359 | 0.1308 | 0.4724 |
| Overall | | | | | | | | | 0.4898 |

Table 25: Performance Metrics for Architecture Version 21

This benchmark once again discredited the k-fold validation approach. Not only did it lead to larger training times, but the improvement in the trained model's quality was questionable at best–and in some cases led to an unmistakable downgrade. The decrease in performance for the 21st iteration can be observed most for the Shuttle and Darknet datasets–yet not so much for the KDD-99 dataset, as its benchmarks were likely already way too low for a decrease to be noticeable.

The next steps were to benchmark the model trained with the help of resampling algorithms–SMOTE and the like. As seen in previous architecture versions, these algorithms perform best when limited in their ability to upscale the dataset: the best performance was achieved when the resamplers increased the size of underrepresented classes by at most a factor of 5000 (further limited by the size of the largest class).

First it was hypothesized that, as one of the more recent algorithms, ADASYN would perform the best on the kind of data observed in section 3. Yet, after attempting to rebalance the KDD-99 dataset, ADASYN output the following error message:

```
Not any neigbours belong to the majority class.
This case will induce a NaN case with a division by zero.
ADASYN is not suited for this specific dataset. Use SMOTE instead.
```

The above error was likely caused by the spread-out distribution of samples in underrepresented classes. This essentially discarded ADASYN from the list of potential resampling algorithms.

Since all categorical features were already converted into numerical features using one-hot encoding as described in section 3, SMOTE-NC was also not applicable.

The next algorithm was KMeansSMOTE: that algorithm was unable to find enough sufficient samples for many classes of the Shuttle dataset. Such that it could be run in the first place, the cluster balance threshold was set to 0–a dangerous precedent, since this would classify each point as its own cluster. Despite that, the resampling of the KDD-99 dataset still failed with the following error:

```
No clusters found with sufficient samples of class perl.
Try lowering the cluster_balance_threshold or increasing
the number of clusters.
```

Since the threshold was already minimal and there was no possibility to increase the amount of clusters any higher (as there were only 2 entries for some classes in the KDD-99 dataset), KMeansSMOTE was also discarded as a resampling algorithm.

Another resampling algorithm that initially showed large hopes was SVMSMOTE. It turned out to be prohibitively slow–yet that wasn't the largest issue.

The datasets Shuttle, Covertype and Darknet were indeed able to be resampled using this algorithm. When benchmarking a model architecture based on version 19 on these datasets, almost all performance metrics were worse than for the original metrics for version 19–the only exceptions were F2, WAR, and WAF1 of the Darknet dataset.

Yet the runtime of the algorithm must also be considered: resampling e.g. the Covertype dataset took over 12 hours. Though rebalancing the Shuttle and Darknet datasets took less time, these, still, underperformed on almost all metrics. In a business context, performance is one of the primal concerns of any use case: rebalancing approaches that take ten times as much space and require days to run (slower than the other algorithms by many factors) are anything but that.

Furthermore, SVMSMOTE, too, ran upon problems when resampling the KDD-99 dataset. The algorithm could not find any support vectors and considered all features to be noise-like, which lead to an abrupt stop to the execution with the following error message:

```
ValueError: All support vectors are considered as noise. SVM-
SMOTE is not adapted to your dataset. Try another SMOTE variant.
```

This left only 2 resampling algorithms introduced in subsection 2.6: SMOTE and BorderlineSMOTE.

Architecture version 19 was thus modified to resample the dataset using SMOTE. The resulting version 22 used SMOTE to resample the dataset, a NN with random mixing layers to extract features, and a categorical cross-entropy loss. Furthermore, for the

sake of experimentation, the architecture used a batch size of 0,5% in order to evaluate the effects of such a batch size on the runtime required to train the model. Version 22 also used no validation splitting, training instead on the complete train set. Table 26 shows the performance of this architecture.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7647 | 0.6062 | 0.9535 | 0.7407 | 0.8121 | 0.4620 | 0.7309 | 0.4534 | 0.6904 |
| Covertype | 0.1230 | 0.0551 | 0.3538 | 0.1348 | 0.1457 | 0.1773 | 0.2274 | 0.0918 | 0.1636 |
| KDD-99 | 0.2056 | 0.0758 | 0.4798 | 0.1979 | 0.2411 | 0.0679 | 0.1799 | 0.0494 | 0.1872 |
| Darknet | 0.2167 | 0.1623 | 0.8752 | 0.2140 | 0.2695 | 0.2232 | 0.4628 | 0.1582 | 0.3227 |
| Overall | | | | | | | | | 0.3410 |

Table 26: Performance Metrics for Architecture Version 22

Though validation splitting was discarded previously, it was hypothesized that, perhaps, a more prominent distribution of underrepresented classes resulting from this rebalancing would remedy the problems observed on imbalanced datasets in earlier architecture versions. Table 27 thus shows the performance of the same architecture, but with a 5-fold validation split.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8288 | 0.7018 | 0.9387 | 0.8200 | 0.8538 | 0.5910 | 0.8753 | 0.6339 | 0.7804 |
| Covertype | 0.1964 | 0.0835 | 0.4659 | 0.2126 | 0.2101 | 0.2198 | 0.2749 | 0.1343 | 0.2247 |
| KDD-99 | 0.2046 | 0.0755 | 0.4981 | 0.2008 | 0.2254 | 0.0694 | 0.1304 | 0.0441 | 0.1810 |
| Darknet | 0.2766 | 0.1740 | 0.8308 | 0.2671 | 0.3387 | 0.2190 | 0.4211 | 0.1652 | 0.3366 |
| Overall | | | | | | | | | 0.3807 |

Table 27: Performance Metrics for Architecture Version 23

The results were surprising: the performance of both architectures was worse than that of version 19 that used no resampling algorithms at all.

To finish benchmarking all resampling algorithms, it was still needed to evaluate the performance of the same architecture while using BorderlineSMOTE. Table 28 and Table 29 describe how the versions 24 and 25 using BorderlineSMOTE performed with and without 5-fold validation splitting, respectively.

The results were even more intriguing: the best performing architecture so far (of those utilizing a resampling algorithm) turned out to be architecture number 24, which used BorderlineSMOTE without validation splitting, yet it still underperformed version 19 that used no resampling techniques at all. One conclusion could be made, however: BorderlineSMOTE performed better than simple SMOTE.

The next 2 architectures would thus evaluate BorderlineSMOTE using FL instead of categorical cross-entropy loss. Their evaluation results are portrayed in Table 30 (without validation splitting) and in Table 31 (with 5-fold validation splitting).

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7286 | 0.5228 | 0.9204 | 0.7029 | 0.7795 | 0.4794 | 0.6009 | 0.4624 | 0.6496 |
| Covertype | 0.2521 | 0.0872 | 0.4414 | 0.2800 | 0.2492 | 0.2067 | 0.2844 | 0.1419 | 0.2429 |
| KDD-99 | 0.9447 | 0.1096 | 0.9209 | 0.9523 | 0.9341 | 0.0860 | 0.0712 | 0.0680 | 0.5109 |
| Darknet | 0.3864 | 0.0614 | 0.7856 | 0.3719 | 0.4310 | 0.1684 | 0.2414 | 0.1063 | 0.3191 |
| Overall | | | | | | | | | 0.4306 |

Table 28: Performance Metrics for Architecture Version 24

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.6877 | 0.4956 | 0.9088 | 0.6628 | 0.7388 | 0.4719 | 0.5951 | 0.4537 | 0.6268 |
| Covertype | 0.1349 | 0.0519 | 0.4250 | 0.1443 | 0.1542 | 0.1911 | 0.2455 | 0.1001 | 0.1809 |
| KDD-99 | 0.9517 | 0.2080 | 0.9245 | 0.9593 | 0.9405 | 0.0734 | 0.0668 | 0.0653 | 0.5237 |
| Darknet | 0.3013 | 0.0862 | 0.7451 | 0.2819 | 0.3602 | 0.1401 | 0.2542 | 0.0882 | 0.2822 |
| Overall | | | | | | | | | 0.4034 |

Table 29: Performance Metrics for Architecture Version 25

The best performing architecture version out of all those using SMOTE-like resampling algorithms, as such, was that which used BorderlineSMOTE in conjunction with FL without validation splitting. In fact, FL led to an increased performance in all cases where BorderlineSMOTE was concerned. This would be kept in mind for later, but another conclusion could be made in addition to this one: the architectures that utilized resampling algorithms all underperformed the original architecture that they were based on.

Consequently, the next few architecture versions would use no resampling algorithms, and would instead use categorical cross-entropy loss.

It was noticed that the architectures developed so far underperformed almost all conventional methods. A hypothesis was that this performance drawback was due to the fixed size of filters in each random mixing layer, which was thus far equal to 20– yet the KDD-99 dataset had 126 features in total. While these 20 convolutions (each combining 3 features) had a chance of discovering unusual feature patterns, even in the best case scenario they would only cover 60 features, which was less than half of the entire dataset. On the contrary, datasets like Shuttle had only 9 features, and 20 filters were likely increasing the model complexity without much improvement to the classification quality.

An attempt was made to use more filters, making the amount dependent on the input vector size (i.e. the amount of feature columns): the architecture would sample 3 random features from the previous layer $N$ times, with $N$ depending on the previous layer's size. The idea was that this would make the feature generator, and thus the classifier, more robust. However, the original (in many ways lackluster) implementation had noticeable drawbacks that made it less efficient than hoped, and consumed so much RAM that running this new model on the KDD-99 dataset was impossible on any of the

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.7889 | 0.6438 | 0.9559 | 0.7720 | 0.8254 | 0.5702 | 0.7022 | 0.5595 | 0.7272 |
| Covertype | 0.2599 | 0.0925 | 0.3869 | 0.2698 | 0.2756 | 0.1844 | 0.2777 | 0.1516 | 0.2373 |
| KDD-99 | 0.9498 | 0.1490 | 0.9229 | 0.9579 | 0.9381 | 0.0754 | 0.0614 | 0.0607 | 0.5144 |
| Darknet | 0.3951 | 0.1077 | 0.7262 | 0.3751 | 0.4446 | 0.1682 | 0.2552 | 0.1350 | 0.3259 |
| Overall | | | | | | | | | 0.4512 |

Table 30: Performance Metrics for Architecture Version 26

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8106 | 0.6303 | 0.9428 | 0.7914 | 0.8480 | 0.5485 | 0.7035 | 0.8480 | 0.7654 |
| Covertype | 0.1590 | 0.0628 | 0.3292 | 0.1830 | 0.1605 | 0.1745 | 0.2436 | 0.1032 | 0.1770 |
| KDD-99 | 0.9497 | 0.1428 | 0.9204 | 0.9577 | 0.9380 | 0.0647 | 0.0608 | 0.0603 | 0.5118 |
| Darknet | 0.2912 | 0.0857 | 0.8045 | 0.2712 | 0.3518 | 0.1646 | 0.2825 | 0.1051 | 0.2946 |
| Overall | | | | | | | | | 0.4372 |

Table 31: Performance Metrics for Architecture Version 27

3 machines used for this work.

Ultimately, the architecture involving random mixing layers had to be put to the sidelines until further notice. In the last architecture version, a highly modified version of it will still play a decisive role.

In the meantime, a new architectural approach was tested: one that took more inspiration from conventional NNs used for image classification.

First an attempt was made to adapt ConvNeXt to this work's use case, i.e. to structured datasets. ConvNeXt v2 is one of the most modern NNs built for image classification, promising some of the best performance out of all contemporary algorithms [Woo+23].

ConvNeXt, however, much like any other visual NN, is built exclusively for recognizing images: every already existing framework that implements this model also expects image data. Adapting the four datasets introduced in the scope of this work to ConvNeXt's expectations would mean modifying the data structure itself, either by padding the missing values with zeroes or by first extracting some kinds of numerical features that could then be plotted onto the three-dimensional width-height-color grid used in image data. The quality of this approach would largely depend on the nature of the dataset in question. Datasets like Shuttle (which has only 9 feature columns in total) would probably have to be padded in order to avoid introducing unnecessary artifacts. Others, like KDD-99 (with 126 features), were under a much lower risk of artifact introduction, but one that was still non-zero.

As such, the decision was made to instead improve on the original NN architecture by taking heavy inspiration from ConvNeXt and similar visual NNs. At the first layer, the commonly used spatial convolutional NN focuses on very simple details, like edges, corners, and basic textures: filters slide around the image, computing new features from

neighboring pixels. As the image moves through deeper layers, the NN starts to instead combine these simple patterns into more complex shapes. Each layer is often smaller than the previous one: convolutional filters are able to "combine" neighboring pixels into one value that encodes some kind of feature.

Another idea was to make the NN architecture "dynamically dependent" on the width of the structured dataset. The architecture would have $\max(2, \lfloor \frac{\#\text{ features}}{10} \rfloor)$ convolutional layers (with "# features" being the vector size), so the NN created for the Shuttle dataset would have just 2 layers, while the NN used to process the KDD-99 dataset would have 12.

The first convolutional layer would have $2(\#\text{features})$ filters, with each subsequent layer having twice as little, but no less than 16 filters.

To further replicate the architecture or ConvNeXt, each convolutional layer would have a batch normalization layer and a 0.2-dropout layer following it: the former would re-center and re-scale the resulting features over each batch, while the latter would drop 20% of all neurons at random.

Finally, the lowest layer of the NN would apply a Softmax activation function for the final classification. Softmax takes the raw values provided by the final convolutional layer and transforms them into class probabilities. This way, instead of getting arbitrary numbers, the NN outputs the likelihoods of the data point belonging to each class: the most probable class becomes the final prediction.

Table 32 shows how this architecture performed when benchmarked against the four datasets.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9616 | 0.9014 | 0.9662 | 0.9647 | 0.9590 | 0.5755 | 0.4941 | 0.5050 | 0.7909 |
| Covertype | 0.5535 | 0.2832 | 0.5131 | 0.6045 | 0.5068 | 0.2773 | 0.2849 | 0.2626 | 0.4107 |
| KDD-99 | 0.9993 | 0.9923 | 0.9993 | 0.9993 | 0.9993 | 0.7950 | 0.7441 | 0.7640 | 0.9116 |
| Darknet | 0.5724 | 0.3271 | 0.8482 | 0.5943 | 0.5647 | 0.3217 | 0.3012 | 0.2634 | 0.4741 |
| Overall | | | | | | | | | 0.6468 |

Table 32: Performance Metrics for Architecture Version 28

The result was very promising: the performance of this new NN architecture blew beyond that of all previous versions.

The 29th version attempted to simplify the NN even further. Instead of defining the amount of layers to be $\max(2, \lfloor \frac{\#\text{ features}}{10} \rfloor)$, it would instead be set to $\max(1, \lfloor \frac{\#\text{ features}}{20} \rfloor)$, i.e. two times less layers for each dataset. Table 33 provides insight into the performance of this version.

This version turned out to be even better on almost all datasets–except Covertype. It resulted in a NN that had just one layer for Shuttle, 2 layers for Covertype, 5 layers for Darknet and 6 layers for KDD-99: while decreasing the complexity for KDD-99 and especially Darknet provided an incredible boost in classification quality, the classification quality for Covertype decreased upon simplifying the NN. The performance on Shuttle,

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9749 | 0.9316 | 0.9731 | 0.9760 | 0.9737 | 0.5838 | 0.6022 | 0.5815 | 0.8246 |
| Covertype | 0.4869 | 0.1802 | 0.4875 | 0.5531 | 0.4252 | 0.2211 | 0.2210 | 0.1868 | 0.3452 |
| KDD-99 | 0.9995 | 0.9939 | 0.9995 | 0.9995 | 0.9995 | 0.9419 | 0.8611 | 0.8914 | 0.9608 |
| Darknet | 0.8297 | 0.6565 | 0.9040 | 0.8428 | 0.8202 | 0.4674 | 0.3378 | 0.3312 | 0.6487 |
| Overall | | | | | | | | | 0.6948 |

Table 33: Performance Metrics for Architecture Version 29

while improving, did not improve by much in comparison to the first two datasets.

Another hypothesis was made: that making the depth of a NN dependent on the amount of feature columns was counter-productive. Complex datasets would generate incredibly deep networks that took an incredibly long amount of time to train, while smaller datasets like Shuttle would have to do with 1 or 2 layers in total. What if, instead of utilizing complex formulas for layer generation, one were to simply define a set amount of convolutional layers? Table 34 shows the performance evaluation of such an architecture that uses 4 convolutional layers.

In addition, the new architecture didn't use dropout layers after each convolutional layer, and instead used them only in the final step after the dense layer.

The formula for the amount of filters stayed the same as in versions 28 and 29. In addition, instead of using ReLU as the activation function of choice, a switch was made to GELU due to research into the differences between the two functions showing that GELU often performs better than its rectified counterpart [DV16] [KGS22].

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9919 | 0.9793 | 0.9905 | 0.9926 | 0.9909 | 0.5367 | 0.4498 | 0.4573 | 0.7986 |
| Covertype | 0.7444 | 0.5962 | 0.7542 | 0.7510 | 0.7389 | 0.7008 | 0.5260 | 0.5441 | 0.6694 |
| KDD-99 | 0.8631 | 0.6098 | 0.9962 | 0.8415 | 0.9034 | 0.5455 | 0.5158 | 0.4661 | 0.7177 |
| Darknet | 0.9883 | 0.9579 | 0.9886 | 0.9886 | 0.9880 | 0.9133 | 0.8616 | 0.8805 | 0.9458 |
| Overall | | | | | | | | | 0.7829 |

Table 34: Performance Metrics for Architecture Version 30

As suspected, the classification quality on the Covertype dataset noticeably improved. The reduction in complexity also led to an improvement on Darknet's classification, and a worsening for KDD-99.

It was further hypothesized that using 5 convolutional layers instead of 4 would allow to strike the perfect middle ground between high complexity for simple datasets and low complexity for large datasets. Table 35 shows the classification results of the 31st architecture version resulting from this thought.

The 31st architecture's performance evaluation confirmed this hypothesis and led to the best average of all metrics out of all architecture versions considered thus far.

The architecture could now be combined with the rebalancing approaches and algo-

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9923 | 0.9801 | 0.9912 | 0.9929 | 0.9913 | 0.5599 | 0.4584 | 0.4701 | 0.8045 |
| Covertype | 0.7438 | 0.5968 | 0.7522 | 0.7505 | 0.7385 | 0.6818 | 0.5209 | 0.5370 | 0.6652 |
| KDD-99 | 0.9976 | 0.9745 | 0.9977 | 0.9978 | 0.9975 | 0.5974 | 0.5127 | 0.5374 | 0.8266 |
| Darknet | 0.9904 | 0.9652 | 0.9905 | 0.9905 | 0.9904 | 0.9143 | 0.8721 | 0.8897 | 0.9504 |
| Overall | | | | | | | | | 0.8117 |

Table 35: Performance Metrics for Architecture Version 31

rithms that proved their worth in previous versions and in scientific literature. Architecture version 32 was developed that based itself on the 31st one, with one distinction: the training datasets were to be preprocessed using the BorderlineSMOTE algorithm. A limitation was also set on how much the algorithm could upscale each class: each class could be increased in size by a factor of 5000, and could become no larger than 10% of the largest class (unless it was already larger than that before upscaling, in which case it would receive no new data points). This latter cap was introduced in addition to the former in order to further curb the occurrence of artifacts in upscaled data.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9920 | 0.9775 | 0.9926 | 0.9919 | 0.9921 | 0.7655 | 0.9473 | 0.8248 | 0.9355 |
| Covertype | 0.7801 | 0.6471 | 0.7848 | 0.7817 | 0.7800 | 0.6537 | 0.6823 | 0.6310 | 0.7176 |
| KDD-99 | 0.9961 | 0.9568 | 0.9966 | 0.9960 | 0.9962 | 0.7087 | 0.7289 | 0.7066 | 0.8857 |
| Darknet | 0.9908 | 0.9666 | 0.9921 | 0.9907 | 0.9911 | 0.8706 | 0.9457 | 0.9004 | 0.9560 |
| Overall | | | | | | | | | 0.8737 |

Table 36: Performance Metrics for Architecture Version 32

Table 36 shows that this approach paid off: upscaling classes led to an increase in classification quality for the entire architecture, proving that resampling algorithms do indeed have their use when applied in conjunction with properly designed NNs.

In the next versions, a different boosting approach was considered instead of BorderlineSMOTE. As opposed to using a Softmax activation function to classify the features extracted by the NN, a BA was to be employed. Due to the considerations mentioned above in subsection 2.2, the choice fell on XGBoost as the better performing algorithm when applied on large datasets [Bol+23]. The NN would thus only be used for extracting features from the dataset, while XGBoost would learn from these extracted features and perform the final classification. In addition, it was done away with dropout layers entirely: motivation being that it risked dropping valuable features that could help classify data points. Table 37 shows that this combination of a NN with XGBoost performed even better than all earlier architectures.

The 34th architecture version attempted something even further: instead of using XGBoost to classify data points based exclusively on the extracted features, the NN would instead output the *concatenation* of the original features with the extracted fea-

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9995 | 0.9986 | 0.9995 | 0.9995 | 0.9995 | 0.9927 | 0.9450 | 0.9622 | 0.9871 |
| Covertype | 0.8877 | 0.8195 | 0.8883 | 0.8881 | 0.8876 | 0.8940 | 0.8503 | 0.8697 | 0.8731 |
| KDD-99 | 0.9999 | 0.9984 | 0.9998 | 0.9999 | 0.9998 | 0.8700 | 0.7989 | 0.8257 | 0.9365 |
| Darknet | 0.9996 | 0.9984 | 0.9996 | 0.9996 | 0.9996 | 0.9894 | 0.9810 | 0.9849 | 0.9940 |
| Overall | | | | | | | | | 0.9477 |

Table 37: Performance Metrics for Architecture Version 33

tures, effectively extending the knowledge base available to XGBoost. After this, XGBoost would perform classification on the concatenated feature vector.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9998 | 0.9995 | 0.9998 | 0.9998 | 0.9998 | 0.9987 | 0.9270 | 0.9509 | 0.9844 |
| Covertype | 0.8963 | 0.8334 | 0.8968 | 0.8966 | 0.8962 | 0.9048 | 0.8660 | 0.8834 | 0.8842 |
| KDD-99 | 0.9999 | 0.9986 | 0.9999 | 0.9999 | 0.9999 | 0.9254 | 0.8301 | 0.8647 | 0.9523 |
| Darknet | 0.9996 | 0.9987 | 0.9996 | 0.9996 | 0.9996 | 0.9916 | 0.9831 | 0.9871 | 0.9949 |
| Overall | | | | | | | | | 0.9539 |

Table 38: Performance Metrics for Architecture Version 34

As can be seen in Table 38, this approach proved to be more successful, which made a lot of sense: more data is oftentimes better than less.

Finally, the next two versions–version 35 and version 36–used yet another rebalancing approach. They extended the architecture version 31 (the one using a 5-layer convolutional NN) with HAFL and FL, respectively, using the definitions of both loss functions as described in subsubsection 2.4.4.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9912 | 0.9773 | 0.9900 | 0.9900 | 0.9903 | 0.5497 | 0.4548 | 0.4658 | 0.8011 |
| Covertype | 0.7285 | 0.5812 | 0.7471 | 0.7360 | 0.7249 | 0.6884 | 0.5392 | 0.5612 | 0.6633 |
| KDD-99 | 0.9981 | 0.9795 | 0.9982 | 0.9982 | 0.9980 | 0.5991 | 0.5179 | 0.5447 | 0.8292 |
| Darknet | 0.9903 | 0.9646 | 0.9904 | 0.9904 | 0.9903 | 0.9103 | 0.8763 | 0.8911 | 0.9505 |
| Overall | | | | | | | | | 0.8110 |

Table 39: Performance Metrics for Architecture Version 35

As can be seen when comparing Table 35 with Table 39 and Table 40, only FL seemed to provide tangible improvement to the classification quality. It must, however, be admitted that Tensorflow provided an out-of-the-box implementation of FL. HAFL, on the other hand, had no such luxury. The original paper written by Xuezheng et al. provided no source code and no usable library that already implemented the function, so it had to be reverse-engineered based on the mathematical definitions provided in the paper [Jia+23]. It is therefore possible that the lackluster quality was a result of some

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9920 | 0.9790 | 0.9913 | 0.9926 | 0.9913 | 0.5981 | 0.4791 | 0.4999 | 0.8154 |
| Covertype | 0.7343 | 0.5776 | 0.7404 | 0.7401 | 0.7287 | 0.6907 | 0.4987 | 0.5187 | 0.6536 |
| KDD-99 | 0.9994 | 0.9935 | 0.9994 | 0.9994 | 0.9994 | 0.9125 | 0.8440 | 0.8636 | 0.9514 |
| Darknet | 0.9890 | 0.9601 | 0.9890 | 0.9892 | 0.9889 | 0.9143 | 0.8612 | 0.8841 | 0.9470 |
| Overall | | | | | | | | | 0.8419 |

Table 40: Performance Metrics for Architecture Version 36

issues and oversights that occurred during development. Due to time constraints, however, it was impossible to refine the implementation until one could be satisfied with the classification results.

By this point, it could be concluded that the 5-layer architecture was the most optimal of the "deep" architectures in regards to the trade-off between classification quality and runtime needed to train a model. It was also clear that, out of all resampling algorithms, BorderlineSMOTE stood out as the one most applicable to the datasets in question, and led to an improvement in classification quality. Furthermore, XGBoost showed promise as a substitute to simple Softmax classification, further improving the quality of the model. Lastly, it was obvious that FL also had a positive effect on classification quality; HAFL, unfortunately, performed worse, likely due to issues with the reverse-engineered implementation.

Further experiments combined these three rebalancing approaches together with the 5-layer NN. To start, architecture version 37 combined the 5-layer NN with BorderlineSMOTE and XGBoost to see how well it would fare. Table 41 provides the relevant performance metrics.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9994 | 0.9982 | 0.9995 | 0.9994 | 0.9994 | 0.9591 | 0.9820 | 0.9660 | 0.9879 |
| Covertype | 0.8923 | 0.8278 | 0.8936 | 0.8924 | 0.8925 | 0.8518 | 0.9039 | 0.8751 | 0.8787 |
| KDD-99 | 0.9999 | 0.9985 | 0.9998 | 0.9999 | 0.9999 | 0.8642 | 0.8080 | 0.8270 | 0.9372 |
| Darknet | 0.9995 | 0.9983 | 0.9995 | 0.9995 | 0.9995 | 0.9906 | 0.9853 | 0.9878 | 0.9950 |
| Overall | | | | | | | | | 0.9497 |

Table 41: Performance Metrics for Architecture Version 37

This architecture indeed performed better than simply applying BorderlineSMOTE without utilizing BAs to classify the extracted features. Despite that, it fell short of architecture version 34 (one that only used XGBoost with the 5-layer network) by 0,42% in absolute terms when considering the average over all metrics (see Table 38). While the metrics, on average, improved on almost all datasets, they fell in the case of Covertype.

The other combination to be tested was a combination of the following:

- BorderlineSMOTE for resampling the dataset before training;

- A 5-layer convolutional NN for feature extraction and concatenation;
- XGBoost to classify the data points;
- FL utilized as the loss function in the NN feature extractor and as the runtime evaluation metric in XGBoost.

When used as an evaluation metric, FL commonly does not utilize the weighting factor $\alpha$:

$$L(y, \hat{y}) = -\sum_{i=1}^{C}(1 - \hat{y}_i)^{\gamma} y_i \log(\hat{y}_i)$$

The above function is the one that was used as the evaluation metric as well.

It should be noted that the authors of XGBoost and the corresponding paper [CG16] did not implement the FL function in the original codebase. The documentation provided with the library, while useful for understanding how to define new loss functions (called evaluation metrics in the documentation), was nevertheless limited. Despite this, a version of FL was created for XGBoost and introduced into the NN architecture. Table 42 provides the performance metrics of this final architecture version.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9993 | 0.9982 | 0.9994 | 0.9994 | 0.9994 | 0.9526 | 0.9502 | 0.9433 | 0.9802 |
| Covertype | 0.8797 | 0.8074 | 0.8809 | 0.8798 | 0.8798 | 0.8343 | 0.8844 | 0.8566 | 0.8629 |
| KDD-99 | 0.9999 | 0.9985 | 0.9998 | 0.9999 | 0.9999 | 0.8157 | 0.7801 | 0.7934 | 0.9234 |
| Darknet | 0.9995 | 0.9981 | 0.9995 | 0.9995 | 0.9995 | 0.9902 | 0.9855 | 0.9878 | 0.9950 |
| Overall | | | | | | | | | 0.9404 |

Table 42: Performance Metrics for Architecture Version 38

Interestingly, the introduction of FL made the performance slightly worse.

The reasons could be manifold. The loss function, as mentioned previously, was implemented by hand based on the original documentation, which may have left space for optimization issues to sneak in. Another explanation could be that FL is, by its nature, not the best evaluation metric that can be used in conjunction with XGBoost.

All this knowledge about the different rebalancing approaches, loss functions and BAs was used to develop the next, and final, architecture version.

In the first couple dozen versions, feature extraction was conducted using a multi-layered random mixing architecture. This had multiple issues:

- Each layer had a set amount of filters, namely 20;
- Due to the multi-layered architecture, many original features were accidentally dropped out of evaluation, further decreasing classification quality;

- Finally, the subpar implementation left little room for adaptation.

To remedy the last point, a new type of layer was developed, now with the help of the Tensorflow development API, which increased performance by leveraging Tensorflow's embedded features instead of trying to replicate NN layer logic via means of default Python and Numpy functions, as was done in the first 27 architecture versions.

The first two points were both remedied with the help of knowledge gained from previous iterations. Knowing now how to modify the architecture depending on the vector size, this knowledge was applied to make the amount of random filters dependent on the amount of features in the original dataset. As in earlier iterations, a kernel size of 3 was chosen for these filters.

Since this newly developed random mixing layer would take random selections of the original dataset's features, it was necessary to assure that the layer would have a near 100% likelihood of covering all original features, as lack thereof was the main drawback of original approaches focusing on random mixing.

This is a classic problem related to the *Coupon Collector's Problem* in probability theory. The goal of this specific problem here is to find how many random selections of size 3 from a set of $X$ elements are required to have a near 100% chance of covering each element at least once. The formula for computing this necessary number of selections $n$ looks like this:

$$n = \left\lceil \frac{X \cdot (\ln X + \gamma)}{3} \cdot \ln\left(\frac{1}{1-p}\right) \right\rceil$$

Where:

- $X$ is the input vector size;
- $p$ is the sought after probability;
- $\gamma$ is the Euler-Mascheroni constant.

This formula (explained in better detail in the next chapter) was then used to compute the number of filters the random mixing layer would have. This layer replaced the previously used combination of concatenation, normalization and flattening layers, further simplifying the architecture.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9999 | 0.9996 | 0.9999 | 0.9999 | 0.9999 | 0.9996 | 0.9270 | 0.9513 | 0.9846 |
| Covertype | 0.9023 | 0.8431 | 0.9027 | 0.9026 | 0.9022 | 0.9103 | 0.8773 | 0.8922 | 0.8916 |
| KDD-99 | 0.9999 | 0.9986 | 0.9999 | 0.9999 | 0.9999 | 0.9087 | 0.8501 | 0.8672 | 0.9530 |
| Darknet | 0.9997 | 0.9988 | 0.9997 | 0.9997 | 0.9997 | 0.9956 | 0.9894 | 0.9924 | 0.9969 |
| Overall | | | | | | | | | 0.9565 |

Table 43: Performance Metrics for Architecture Version 39

The results of this approach can be seen in Table 43.

As can be observed, this new (rather simple) architecture outperformed all others: those that used an earlier rudimentary version of random mixing layers (and multiple such layers at that), and those that used a deep convolutional network–thus achieving the best classification performance on all datasets benchmarked in this work without having to resort to resampling algorithms or novel loss functions.

Due to time constraints, version 39 was also the final version of the NN architecture developed in this work.

# 5 Evaluation

In the end, two different architectures achieved the two best results when it came to benchmarking the 8 different metrics over the 4 datasets.

The first architecture was the more "traditional" deep NN resulting from the 34th version in the previous section, which was inspired by ConvNeXt and other NNs designed for image classification. This architecture combined 5 consequent layers of 3x1 spatial convolutions together with layer normalizations after each convolutional layer. After this, the resulting features were flattened into a one-dimensional vector and recombined into another, smaller set of features using a dense layer. Thereafter, the resulting feature vector was combined with the original feature vector, the concatenation of which was then fed into the XGBoost algorithm to produce a final classification decision.

The structure of this architecture, together with the amount of features computed in each layer, can be observed in Figure 2.

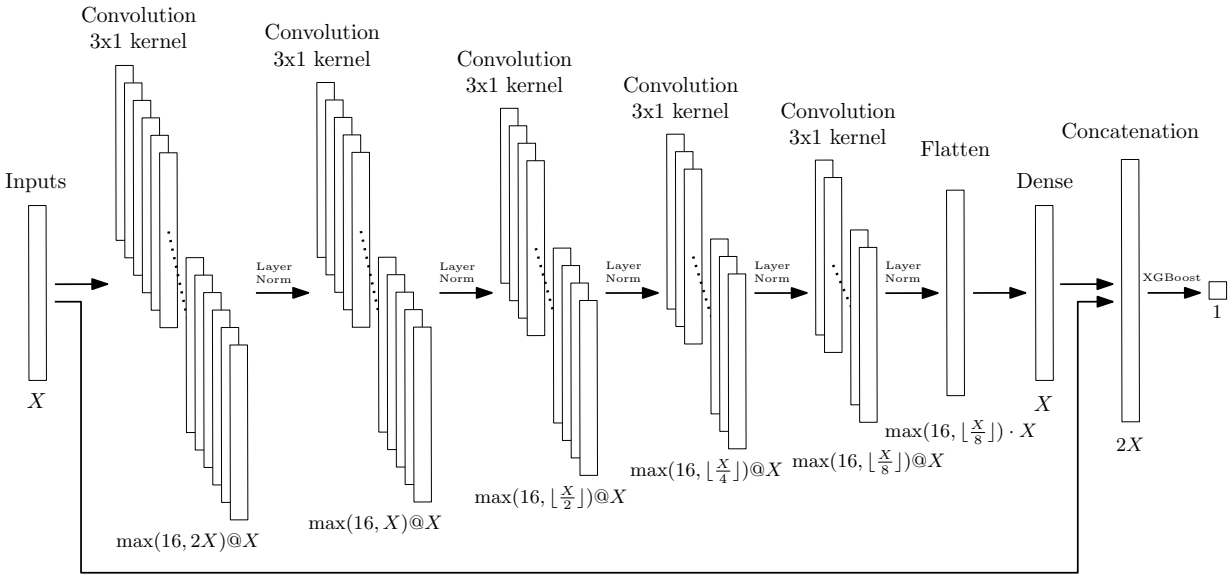


Figure 2: Version 34 Architecture

The other architecture, resulting from the 39th version, was much more simple and based on a more stochastic approach. It used a simple 1-layered architecture and leveraged XGBoost's classification quality with a random mixing layer for additional preliminary feature engineering.

This final architecture used the following elements, in the following order:

1. *Input Layer*: This layer reads the training data and reshapes it from a 2-dimensional tensor (batch_size $\times$ input_dim) into a 3-dimensional tensor by adding a new fictive dimension of size 1. This is necessary for further processing of the data by Tensorflow's convolutional layers. Further points assume that the input layer receives a vector $\mathbf{x}$ of size $X$.

2. *Random Mixing Layer*: This layer takes $n$ random selections of size 3 of the $X$ original features of the input dataset, following the same formula as outlined above:

$$n = \left\lceil \frac{X \cdot (\ln X + \gamma)}{3} \cdot \ln\left(\frac{1}{1-p}\right) \right\rceil$$

   Where:

   - $X$ is the input vector size (i.e. the width of the input dataset or the number of feature columns);
   - $p$ is the sought-after coverage probability (e.g. $p = 0.999$ for a sought after probability of 99.9% of covering each element at least once);
   - $\gamma$ is the Euler-Mascheroni constant ($\gamma \approx 0.5772156649$), which was set to 0.577 in the code in order to increase computational effectiveness.

   The random mixing layer then feeds the learned $n$ features to the next layer.

3. *Dense Layer*: This layer applies a linear transformation, mapping the extracted features back to the original input dimension. Mathematically, the transformation follows this formula:

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$$

   Where:

   - **x** is the input vector;
   - **W** is a weight matrix;
   - **b** is a bias vector.

   The dense layer thus learns which combinations of the computed artificial features provide the most insight into the data. This helps the model discard irrelevant details while keeping the essential information.

4. *Concatenation Layer*: As observed in this work, the model performs best when the extracted features are used for classification in conjunction with the original features from the training dataset. This layer thus concatenates the two sets of features of size $X$ (one from the original dataset and one from the dense layer) together into a vector of size $2X$ for each training instance. The model then passes these features onto the next layer.

5. *XGBoost*: For the final classification, XGBoost has proved itself to be more fitting than a simple loss-based classification. By integrating the above deep feature extractor with XGBoost, the architecture capitalizes on the advantages of both methods: the random mixing layer provides a robust mechanism for extracting data patterns, while the BA offers strong efficiency and improved generalization on tabular data.

The structure of this architecture is portrayed in Figure 3.

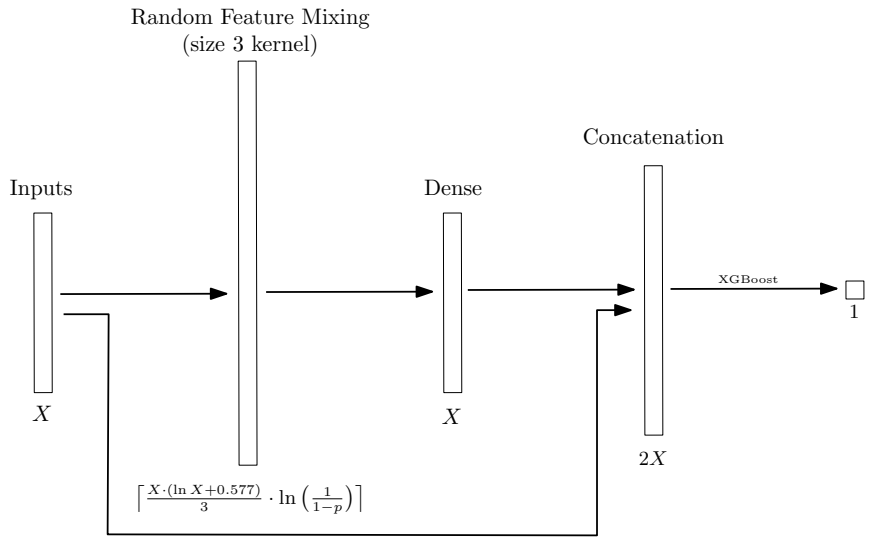


Figure 3: Version 39 Architecture

This architecture outperformed the deep convolution-based network in almost every single metric, with a small exception in form of MAP for the KDD-99 dataset, where the latter had slightly better performance. This shows, just as assumed initially, that feature extraction in structured data should not be limited to neighboring values, and should instead leverage the hidden correlations between non-neighboring features.

As a result, the version 39 architecture is the one produced in this work as the better of the two.

This final architecture's performance still needed to be compared to the most common approaches used in contemporary applications on the same metrics as used throughout this thesis. The goal was to cover a broad spectrum of different methods. Ultimately, three models were chosen for comparison: a LCS, an out-of-the-box NN, and XGBoost itself (without the preliminary feature extraction).

The principles of the first model, the LCS, were already explained in subsection 2.1. The model used for comparison with the developed NN architecture was trained using the `scikit-elcs` Python library created by Zhang et al. in version 1.2.4 [ZU20]. The performance of said model can be observed in Table 44.

The next model to be compared to the developed NN architecture was decided to be a readily available open-source NN-based framework. The decision was to choose TabNet as the benchmark for this type of model [AP21]. Unlike traditional NN models, which usually rely on fully connected layers (as is the case for the architectures developed in this work), TabNet uses an attention mechanism to selectively focus on relevant features for each decision. Furthermore, TabNet applies the concept of self-supervised learning to tabular data. This should, in theory, allow it to achieve good performance with a smaller amount of labeled data, which is precisely the use-case considered in the

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.8163 | 0.5102 | 0.7433 | 0.8451 | 0.7780 | 0.2885 | 0.2866 | 0.2768 | 0.5681 |
| Covertype | 0.6722 | 0.4706 | 0.6652 | 0.6833 | 0.6600 | 0.4571 | 0.3532 | 0.3594 | 0.5401 |
| KDD-99 | 0.9888 | 0.8760 | 0.9864 | 0.9898 | 0.9875 | 0.2967 | 0.2182 | 0.2357 | 0.6974 |
| Darknet | 0.9451 | 0.8045 | 0.9440 | 0.9500 | 0.9390 | 0.6752 | 0.4195 | 0.4494 | 0.7658 |
| Overall | | | | | | | | | 0.6429 |

Table 44: Performance Metrics for eLCS

scope of this work: classes (i.e. labels) that occur less often in the dataset would still be represented in the model during decision-making. Table 45 provides the performance metrics for TabNet on the four datasets considered in this work.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9971 | 0.9924 | 0.9969 | 0.9973 | 0.9968 | 0.7532 | 0.6416 | 0.6690 | 0.8805 |
| Covertype | 0.8593 | 0.7743 | 0.8606 | 0.8603 | 0.8587 | 0.8206 | 0.7323 | 0.7650 | 0.8164 |
| KDD-99 | 0.9969 | 0.9665 | 0.9968 | 0.9971 | 0.9966 | 0.4139 | 0.3901 | 0.3929 | 0.7689 |
| Darknet | 0.9969 | 0.9886 | 0.9969 | 0.9969 | 0.9969 | 0.9551 | 0.9525 | 0.9531 | 0.9796 |
| Overall | | | | | | | | | 0.8613 |

Table 45: Performance Metrics for TabNet

Finally, it was deemed necessary to compare the quality of the baseline XGBoost model trained without the preliminary feature generation steps. Just as during development of the NN architecture in section 4, this model was created with help of the `xgboost` Python library in version 2.1.2 [CG16]. For the sake of complete fairness, all hyperparameters were set to be the same as in the XGBoost code section of the developed NN architecture. Table 46 shows the performance of the baseline XGBoost model applied on the original datasets as preprocessed in section 3 without any newly generated features.

| | F2 | MCC | WAP | WAR | WAF1 | MAP | MAR | MAF1 | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Shuttle | 0.9997 | 0.9993 | 0.9997 | 0.9997 | 0.9997 | 0.9997 | 0.9243 | 0.9501 | 0.9840 |
| Covertype | 0.8692 | 0.7895 | 0.8697 | 0.8696 | 0.8690 | 0.8839 | 0.8319 | 0.8539 | 0.8546 |
| KDD-99 | 0.9998 | 0.9980 | 0.9998 | 0.9998 | 0.9998 | 0.8090 | 0.7351 | 0.7628 | 0.9130 |
| Darknet | 0.9997 | 0.9991 | 0.9997 | 0.9997 | 0.9997 | 0.9948 | 0.9890 | 0.9918 | 0.9967 |
| Overall | | | | | | | | | 0.9371 |

Table 46: Performance Metrics for XGBoost

As can be observed from the averages over all metrics and all datasets for each of the three benchmark models and for the developed final NN architecture, the latter model does indeed possess a better general performance than any of the other methods evaluated in this section.

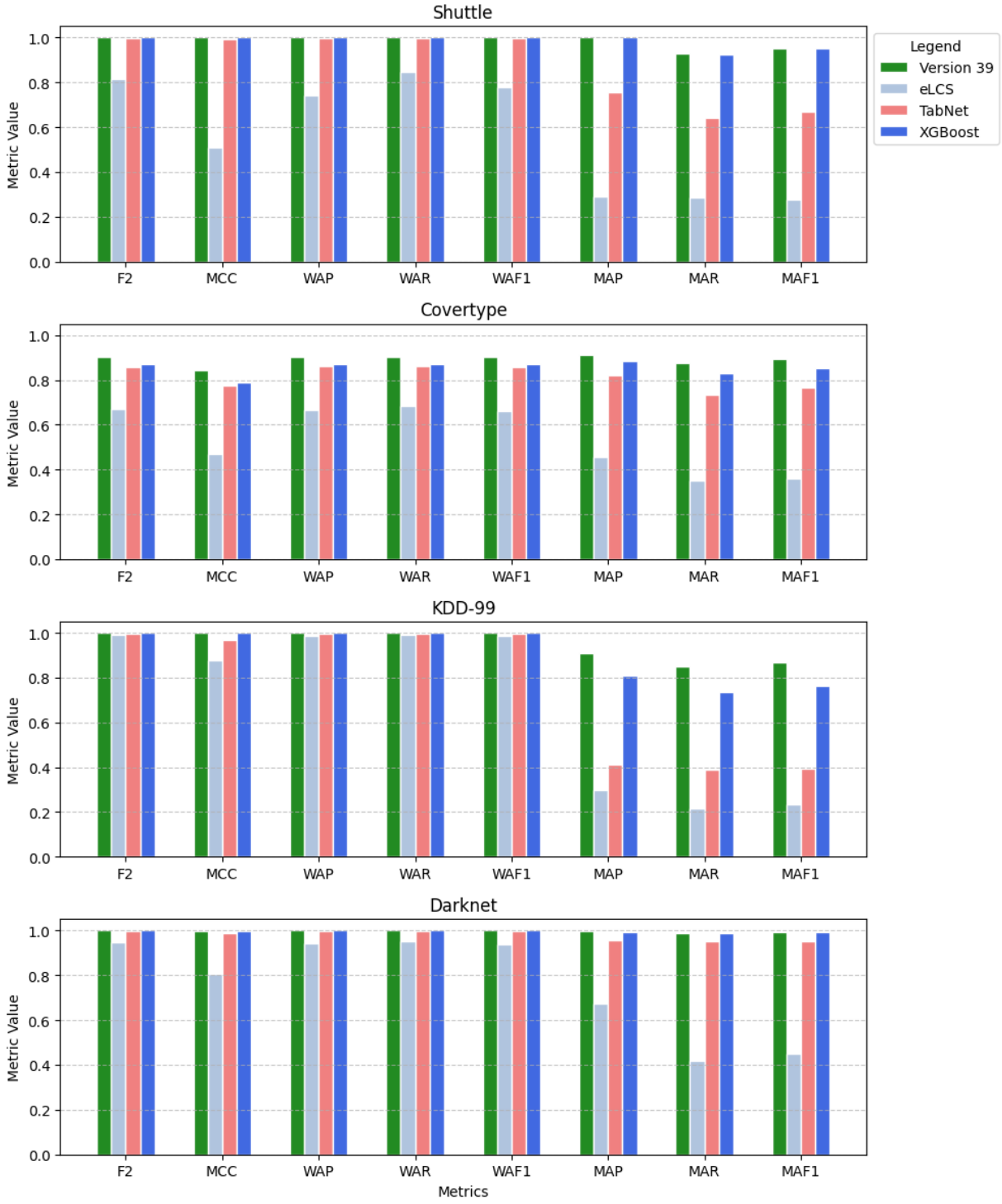


Figure 4: Comparison of Models Across Datasets and Performance Metrics

For easier comparison, it was decided to plot all of the calculated performance metrics using a bar chart in Figure 4. This would allow readers to intuitively and visually compare the quality of each approach as measured by each separate metric, and to determine if one of the benchmark approaches was indeed better than the one developed in this work when applied on a particular dataset. Additionally, readers can refer to Figure 11 for a comparison of the four different approaches using radar plots.

Both of these graphs openly demonstrate considerable performance differences between the different models. Furthermore, they clearly show the scale of performance improvement that the NN architecture developed in this work promises on imbalanced datasets. One of the strengths of the model lies therefore in its ability to maintain consistently high performance across multiple datasets.

The difference between the proposed architecture and the 3 other benchmark models, as well as the quality of the different architectural elements, will be further discussed in the next section.

# 6 Discussion

The original goal of this thesis was the following:

> *To build a universal open-source NN architecture that can be efficiently applied to a variety of imbalanced structured datasets, regardless of the dataset's distribution, origin, and real-life application.*

The developed approach would have thus had to generalize across different datasets while maintaining strong robustness and efficiency without the need for extensive manual adjustments and hyperparameter optimization to achieve an unfair advantage over other, similarly unoptimized comparison models.

Through iterative architectural improvements, this goal was successfully achieved and culminated in the selection of architecture version 39 as the optimal solution, whose layer structure can be observed in Figure 3. This architecture was iteratively refined based on extensive benchmarking of different model versions, each tested across four distinct imbalanced datasets: KDD-99, CIC-Darknet2020, Covertype, and Shuttle. These datasets were selected to ensure that the model's effectiveness could be demonstrated across different domains, including cybersecurity, environmental classification, and aerospace diagnostics.

The developed NN architecture demonstrated consistently high performance across all benchmark datasets. The model was able to achieve high classification accuracy while maintaining good precision and recall scores across all classes, including the minority classes, and performed better than both the LCS and the TabNet benchmark models in all metrics on all datasets.

However, it is important to note that the architecture was not the most optimal in every single case.

A pure XGBoost algorithm, without the preliminary feature generation via NNs, outperformed the proposed architecture in certain metrics. Specifically, XGBoost achieved slightly better results in the following cases:

- For the Shuttle dataset, XGBoost outperformed the proposed architecture in the following metrics:
  - MAP (by 0.01% in absolute terms).
- For the Darknet dataset, XGBoost outperformed the proposed architecture in all eight observed performance metrics by the following measures:
  - MCC (by 0.03% in absolute terms).

These outlined differences, however, arguably lay within margin of error. In all other metrics, the proposed architecture decisively outperformed all other other methods. This indicates that the creation of additional spatially independent features indeed helped leverage the strengths of XGBoost's boosting approach and better recognize what differentiated the minority classes of the tested datasets from the respective majority classes.

The proposed architecture thus boasts impressive classification quality. When computing the average over all metrics and all datasets, the proposed architecture outperforms the others by the following margins:

- LCS gets outperformed by 31.36% in absolute terms;
- TabNet gets outperformed by 9.52% in absolute terms;
- XGBoost gets outperformed by 1.94% in absolute terms.

The advantage over LCS can be explained by LCS' general failure to properly weight those rules that classify the data point as belonging to a certain minority class. Due to the nature of imbalanced datasets, the rule-set of any given LCS would be exceedingly dominated by rules classifying points into the majority class, ultimately leading to false decisions as the minority-classifying rules get outvoted by the former.

The main difference between the proposed architecture and TabNet consists in the latter's usage of attentive transformers (which take the processed feature representations from the feature transformer and apply a sparse attention mechanism) and the former's usage of gradient boosting to classify the learned feature values. This finding was all the more interesting when one considers the main use-case of transformers, which is to leverage global context in decision-making: one would naturally assume that this would prove beneficial for structured tabular datasets where correlations between different features don't always occur only between neighboring columns. Indeed, the performance of TabNet seems to be greater than that of the simple 5-layered Convolutional Neural Network (CNN) developed in version 31, as can be observed in Figure 5. However, the utilization of XGBoost and random mixing in the proposed architecture provided a noticeable boost to the architecture's performance. The improved classification quality is also in line with contemporary research, as scientific findings have already shown in the past that XGBoost tends to outperform most deep learning models, including TabNet [Fay+22]. Its utilization for final classification in the proposed architecture thus seemed to leverage the inherent advantages gradient boosting has over conventional NN models.

Notably, the proposed architecture also outperformed a version of XGBoost that did not use deep learning ensemble methods. This is also in line with research done over the previous years, which, too, showed ensemble models to perform better than "pure" BAs [Fay+22]. This can be explained very simply: the random feature generation layer developed in this work creates novel machine-interpretable features which, in conjunction with the original features of any given data point, provide XGBoost with far more data to make a well-founded classification decision.

As can be seen, the effectiveness of the proposed NN architecture was achieved by leveraging a number of design choices and enhancements tailored for imbalanced classification. These design choices did not come out of nowhere, but were instead the result of extensive iterative improvement over more than three dozen different architecture versions.

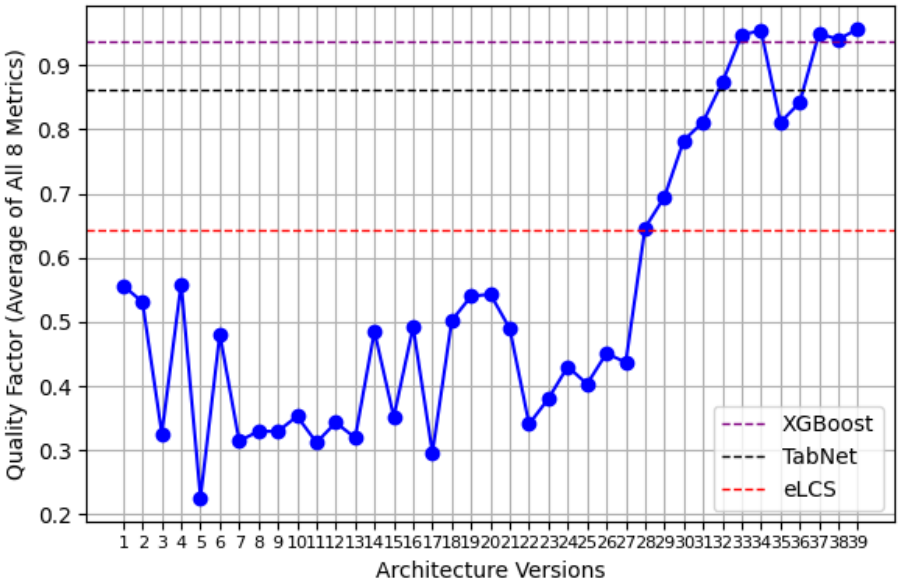


Figure 5: Average of All Metrics Across Architecture Versions for All Datasets

To observe the effect on performance of each given change, it was insufficient to evaluate all 32 different performance metrics (8 metrics for each of the 4 datasets) independently, as the amount of metrics would be prohibitively complex to make a well-founded conclusion. Instead, as mentioned previously, the performance was measured using an unweighted average of all 32 metrics: each of the different metrics was arguably as important as all others, highlighting different areas in which the particular architecture may excel. This provided a general singular metric, which was in line with the stated goal of building a *universal* NN architecture that can be efficiently applied to a variety of different use-cases. A visual representation of the development of this general metric over the 39 different architecture versions (compared also to the average over all metrics for eLCS, TabNet, and XGBoost) can be seen in Figure 5.

During the training process, multiple **batch sizes** were evaluated to determine their impact on model performance and training efficiency. The tested sizes were:

- *32*;
- *256*;
- *BGD* (i.e. the entire dataset);
- *1% of the dataset*;
- *5% of the dataset*.

A batch size of 32 provided a good classification quality. BGD led to substantial performance degradation: this is likely due to the reduced stochasticity in gradient updates. A dynamic batch size approach was investigated, wherein the batch size was

set to 1% (and in another iteration to 5%) of the total dataset. This method allowed for automatic batch size adjustments based on dataset size, thereby maintaining a balance between computational efficiency and generalization.

It was concluded that, while smaller batch sizes generally improve model generalization, a dataset-dependent dynamic batch size can offer superior flexibility and efficiency, particularly in scenarios where computational resources and training time are constrained. The dynamic batch size of 1% was thus ultimately selected as the optimal choice.

In addition, multiple **resampling algorithms** were evaluated as a method of addressing the class imbalance present in the benchmark datasets:

- *SMOTE*;
- *BorderlineSMOTE*;
- *SMOTE-NC*;
- *KMeansSMOTE*;
- *SVMSMOTE*;
- *ADASYN*.

While such resampling techniques remain useful in some scenarios, their application in this study was found to be suboptimal.

SMOTE-NC was discarded due to the nature of the datasets: all of them already had their categorical features converted using one-hot encoding, which made the algorithm obsolete in its main selling point. ADASYN, SVMSMOTE, and KMeansSMOTE were discarded only after attempting to resample the datasets: all three could not properly create synthetic samples for minority classes in at least one of the four datasets represented in this work.

A likely key reason for the limited success of the two resampling methods that *could* successfully resample the datasets (SMOTE and BorderlineSMOTE) was the introduction of artificial patterns into the training data. The generated synthetic samples, while possibly statistically similar to real data points, may not have aligned with the natural feature distribution of the concerned minority classes. The newly generated samples thus risked distorting the decision boundary rather than improving classification performance. Furthermore, the presence of such artifacts may have caused the model to learn redundant or misleading features, thereby reducing its ability to generalize effectively to unseen data. This conclusion is strengthened by the difference in performance between architecture versions 7 up to 11: the best performance could be achieved when the upscaling of minority classes was limited to a factor of 5000, which allowed both for an increased visibility of minority classes and a minimal risk of artifact occurrence. The distribution of the four datasets rebalanced under this paradigm can be observed in Figure 7.

Several **loss functions** were tested as a substitute for resampling algorithms:

- *Categorical cross-entropy (weighted and unweighted)*: while effective in standard multi-class classification tasks, it had considerable limitations when applied to imbalanced datasets in pure NN models. The unweighted version disproportionately concerned itself with the majority class, leading to poor classification of minority class instances. The weighted version provided some improvements by assigning higher loss values to minority class samples–however, it failed to sufficiently mitigate the imbalance, which led to suboptimal performance.

- *FL*: it emerged as the most effective loss function among those tested when applied to pure NN models. By applying an adaptive scaling factor to down-weight easy examples and focus more on difficult-to-classify instances, it improved the model's ability to learn from underrepresented classes.

  It should be noted that while FL proved beneficial for neural networks, its application to XGBoost was not as effective. XGBoost's tree-based structure relies on decision boundaries rather than backpropagation-based learning, meaning that the advantages of FL in improving gradient updates were not directly transferable. Consequently, traditional loss functions such as multiclass logarithmic loss or weighted cross-entropy remained more suitable for XGBoost. This further highlights the importance of model-specific loss function selection.

- *HAFL*: The implementation used in this work did not yield the expected benefits as promised in earlier works [Jia+23]. It was unfortunate that the original authors provided no original implementation of the function: the attempt at implementing HAFL undertaken within the framework of this thesis introduced noticeable challenges. These issues limited its practical usability in this study, which ultimately led to a discarding of HAFL as a potential aid during training.

Two **activation functions** were evaluated:

- *ReLU*: initially used in earlier architecture versions, it was ultimately discarded due to lower performance and also due to a phenomenon known as the "dying ReLU" problem: ReLU outputs zero for all negative inputs, leading to neurons that never activate during training [He+15], which can be a likely reason for its middling performance in some cases.

- *GELU*: models utilizing GELU exhibited slightly improved performance. In earlier research, GELU was found to facilitate better generalization by preserving more gradient information in deeper layers [Lee23]. Despite the increased computational complexity due to having to calculate exponential functions, GELU's superior performance proved it (in this research as well as in other [KGS22]) to perform better than the alternative.

Despite that, the final architecture used neither of the activation functions, relying instead on XGBoost for weighting the computed features.

The impact of **validation** on model performance was also examined during training.

It was observed that incorporating a validation split often led to adverse effects on classification quality, particularly for minority classes.

One of the key challenges introduced by *validation splitting* was the reduction in the amount of training data available to the model. Given the already highly imbalanced nature of the datasets, removing even a small percentage of data from the training set further limited the number of available minority class samples. This likely led to weaker decision boundaries and caused the model to be biased even further toward the majority class. Additionally, large fluctuations in validation loss were observed across different training iterations on the same architecture. In highly imbalanced datasets such as here, the validation loss may not always provide a reliable signal of overfitting: models can struggle to generalize to minority classes in the validation set, leading to early stopping before the model fully learns these patterns.

*K-fold validation splitting*, though promising initially, provided no noticeable benefits, instead only increasing the training times. *Stratified validation splitting*, as shown by evaluations of architecture versions 15 and 16, led to no improvement of classification quality as well.

An alternative approach was tested in which validation splitting was entirely omitted: training was conducted directly on the full training set (with the early stopping criterion now tracking the loss on the training set), with the independent test set being used for final evaluation. This adjustment led to several notable improvements. Classification performance remained comparable, and in certain cases, models trained without validation exhibited better generalization on the test set, particularly for underrepresented classes. The model may also train slightly longer for this one iteration thanks to having access to more information, potentially improving its ability to handle unseen data. Class weights or loss functions (e.g., weighted cross-entropy or FL) also become more effective when the training data is larger and better represents the minority classes.

It was thus concluded that, while validation splitting may be often considered beneficial for improving model performance, its impact in the context of highly imbalanced datasets may not always be positive. The loss of valuable minority class data outweighed the advantages of monitoring validation loss, making the decision to skip validation splitting a viable strategy for improving both efficiency and classification quality.

One of the other key aspects investigated in this study was the impact of different **architectural and layer configurations** on classification performance. Several structural variations of the proposed NN feature extractor were evaluated to determine the most effective design for handling multi-class classification in imbalanced datasets.

Some architecture versions (beginning with version 28 and ending with version 38) sought to optimize the depth and structural composition of the network. These utilized standard spatial convolutions, which used sliding windows moving across the dataset and applied certain filters on neighboring sets of feature values.

Initially, a dynamically determined architecture was explored in version 28, wherein

the number of layers was adjusted based on the complexity of the dataset. While this approach demonstrated some promise in tailoring the model to different datasets, the added computational complexity on wider datasets (i.e. those with larger input vectors) necessitated exploring other configurations.

Two fixed-depth architectures were then assessed: one with 4 layers (version 30) and another with 5 layers (version 31). It was observed that the 5-layer architecture consistently outperformed the 4-layer variant across all evaluation metrics. The presence of an extra layer likely allowed for better generalization to minority classes and contributed to increased non-linearity in the model, which aided in capturing complex relationships within the data: convolutional layers, unlike transformers, tend to better capture spatial relationships between neighboring features, so it was natural that an increased number of layers would allow for capturing a wider range of features in one output value.

It was further noted that the superior performance of the 5-layer architecture did not come at the cost of excessive overfitting, despite omitting certain regularization techniques like dropout. While an approach less fit for structured datasets than the final concept outlined below, this architecture proved to have comparably good performance if properly optimized.

The best-performing architecture, however, did away with deep convolutional networks completely, opting instead for a singular kind of layer that was coined in this work as a *"random mixing layer"*. This layer took random selections of the original datasets' features, making sure to make enough draws such that each feature had a 99.9% likelihood of being drawn by the feature generator at least once. A primary contributing factor to the underperformance of this layer's earlier iterations was the insufficient number of filters, which failed to adequately capture the complex feature relationships present in structured datasets, as well as other performance issues connected to its earlier implementations.

Through careful and lengthy optimization, it was possible to create a well-performing version of this layer that provided no runtime drawbacks in comparison to default Tensorflow implementations of convolutional layers. Furthermore, since this random mixing layer didn't depend on spatial relationships like convolutional layers, the model was able to learn more relationships between non-neighboring features, which is especially useful when processing structured (tabular) data where interdependencies are often hidden.

A key feature of the proposed model was also the integration of **gradient boosting**, represented here by XGBoost alongside the multi-class logarithmic loss evaluation metric after the feature generation and concatenation steps. By incorporating XGBoost as a supplementary learning mechanism, the model was able to mitigate biases that typically arise in class-imbalanced settings.

XGBoost contributed significantly to classification performance by addressing class imbalance through its iterative learning process. Unlike conventional NN training, which may struggle to adaptively adjust its focus toward underrepresented classes, XGBoost automatically reweights misclassified instances, thereby increasing attention on minority-class examples. This ensured that rare categories were not overshadowed by

dominant classes during optimization.

Additionally, the **concatenation of features** before applying XGBoost played a crucial role in refining classification quality. The feature extraction process within NNs inherently generates increasingly complex and abstract representations, which in this work were subsequently combined with raw original features before being passed into XGBoost. This hybrid approach capitalized on both feature learning (deep as in version 34 *and* stochastic as in version 39) and on gradient-boosted decision trees, allowing the model to maintain a high degree of flexibility when distinguishing between difficult class instances.

Experimental results indicated that this integration led to measurable improvements in classification performance.

The **trade-off** between computational **cost** and classification **performance** must be carefully considered, particularly in the context of large-scale datasets. Training the feature-generating NN in architecture version 34 proved to be a time-intensive process, often requiring several hours for large datasets. In contrast, training the entire model from architecture version 39 took less than 2 hours for the KDD-99 dataset, and training a pure XGBoost model took, on average, only a few minutes.

This discrepancy is primarily attributed to the complexity of deep learning training, which involves iterative backpropagation, weight updates, and regularization techniques. In contrast, pure XGBoost, despite employing an ensemble of decision trees, leverages gradient boosting with parallelization and optimized memory usage, resulting in comparatively faster training times.

This extended training duration raises some practical concerns regarding model deployment in environments where computational efficiency is a priority. While the benefits of feature learning are evident in improved classification performance, the associated costs in terms of training time and hardware requirements must be acknowledged.

For real-world applications, the decision to utilize the proposed ensemble architecture over XGBoost alone depends on the specific constraints of the problem domain. If training time and computational resources are limited, XGBoost may serve as a more practical alternative, offering competitive performance with significantly reduced training overhead. However, if classification performance–particularly in distinguishing minority classes–is of primary importance, the added cost of training the random mixing-based feature extraction NN may be justified.

While the current approach balances classification performance and computational efficiency, certain optimizations could further enhance both aspects. Potential refinements may include adjustments to the training process and modifications to the network structure. These possibilities, along with directions for future research, will be discussed in detail in the following chapter.

As also discussed in earlier sections and reiterated in the introduction, a major limitation of contemporary research is the frequent lack of publicly available implementations, which restricts the ability of others to validate findings and build upon existing work [Bak16] [HWS20]. With this motivation, all code used for building, training, and testing the models on the given datasets, as well as the datasets themselves and the

code used to preprocess them (as described in section 3), is available in a Zenodo repository under the DOI `10.5281/zenodo.15029115`. This ensures that all experimental results presented in this study can be fully reproduced and independently verified.

By making the entire experimental pipeline openly accessible, this study also aligns with best practices in scientific research, contributing to transparency and facilitating further advancements in the field of multi-class classification on imbalanced datasets.

# 7 Limitations and Future Work

Despite the progress achieved in developing and benchmarking a general NN architecture for multi-class classification on imbalanced datasets, several limitations have been encountered throughout this study. Constraints related to computational resources and time have restricted the scope of experimentation, preventing the evaluation of additional architectural variations and optimization techniques.

While multiple architectures have been tested across various datasets, it was not feasible to exhaustively explore all potential modifications, such as alternative activation functions, loss functions, and ensemble strategies. Furthermore, due to hardware limitations, certain models that may have provided valuable insights could not be included in the benchmark, particularly those requiring extensive computational power, large amounts of RAM, or longer training times (see earlier attempts at using lower batch sizes and the lack of hyperparameter optimization). These constraints highlight opportunities for future research, where more diverse configurations and larger-scale experiments can be conducted, ideally on better and more high-performing hardware.

A big number of potential improvement areas can thus be expanded upon in future research, some of which will be listed below.

- Subsequent research could further **fine-tune the neural network architecture** used for feature extraction, as additional modifications may yield improvements in classification performance.

  One possible approach involves adding extra layers to assess whether deeper architectures of random mixing layers enhance feature learning and lead to more accurate predictions, particularly for minority classes. While deeper networks have the potential to capture more complex representations, their benefits must be weighed against the risk of increased computational demands and feature dropout.

- Future work could also focus on more extensive **hyperparameter optimization for XGBoost**, as the current study primarily evaluated models using multiclass cross-entropy loss and FL, with the former proving to be the more effective evaluation metric. While these loss functions performed well on their own, additional evaluation metrics (such as Area Under the Receiver-Operating-Characteristic (ROC) Curve (AUC) or Area Under the Precision-Recall Curve (AUC-PR)) could be explored to optimize classification balance in imbalanced datasets.

  Other key hyperparameters, such as tree depth and step size shrinkage, were not exhaustively tuned beyond their default values of 6 and 0,3 respectively. Future studies could evaluate different values of these hyperparameters and others to fine-tune them and assess their impact on classification performance in imbalanced datasets.

- A potentially more effective approach for future research involves using **transformer-based architectures** (like those appearing in TabNet). While the

used feature generation layers (both spatial and random) offer certain advantages, transformers possess comparable and often even better performance as per earlier research [Zho+21]. Their ability to model long-range dependencies and dynamically weigh feature importance could address some of the challenges faced in learning from imbalanced datasets: transformers leverage self-attention mechanisms to capture correlations and relationships across all input features, making them highly adaptable to varying data distributions, much like BAs. Due to the constraints of this study, which focused on adapting NNs onto structured tabular data, transformer models were not integrated into the proposed architecture. It remains to be examined whether the latter's ability to focus on minority class patterns could improve classification performance beyond what was achieved in this study. Additionally, transformers' computational efficiency and scalability should be evaluated, as it may influence viability for real-world applications with limited resources and hardware constraints comparable to those observed in this work.

- A yet unexplored direction for improvement involves experimenting with different normalization techniques, such as **group normalization** or **batch normalization**.

  Batch normalization, for instance, normalizes activations across mini-batches, introducing a regularizing effect that can accelerate training and mitigate internal covariate shift.

  Group normalization, which divides feature channels into smaller groups before applying normalization, offers a compromise between batch and instance normalization, but may be less effective for large batch sizes where the impact on variance stabilization is reduced.

  Layer normalization, in contrast, operates independently for each training example by normalizing across all features within a layer. Since layer normalization was used in some intermediate parts of this study due to its independence from batch size, future research could explore whether these alternative normalization strategies lead to improved generalization, particularly when applied to imbalanced datasets with varying feature distributions.

- An important aspect that could be refined in future work (ideally conducted on better hardware) is the selection of smaller **batch sizes**. While the batch size strategy employed in this study was chosen to balance computational efficiency and classification performance, the varying sizes of mini-batches across different datasets may have introduced inconsistencies in optimization dynamics. Smaller batch sizes could help improve generalization by introducing additional stochasticity during training, potentially preventing the model from converging to sharp minima that may not generalize well to unseen data.

  However, smaller batches also come with trade-offs, such as increased training time and greater sensitivity to hyperparameter tuning, which may prove to be

problematic in real-life applications. Future research could also evaluate the impact of different batch size strategies on classification performance if a more efficient model architecture is developed.

- Finally, future research could also explore **hybrid approaches** to improve training efficiency and optimize computational resource usage.

  One potential strategy involves pre-training the neural network on smaller subsets of data before fine-tuning it on the full dataset, which could accelerate convergence and reduce the risk of overfitting, particularly in cases where data imbalance is severe.

  Additionally, transfer learning could be leveraged to initialize the model with pre-trained weights from a related task, potentially improving feature extraction without requiring extensive training from scratch.

  Such techniques could be particularly valuable in scenarios where computational resources are limited, as they allow deep-learning-based feature extraction models to become more computationally efficient while maintaining strong classification performance. Future work could also investigate the extent to which these optimizations can be applied to e.g. transformer-based architectures and how they can impact both training time and classification accuracy.

This study provides a foundation for future research by making both the code and datasets used in this thesis openly available. Researchers interested in building upon this work can access these resources under the DOI `10.5281/zenodo.15029115`. By fostering reproducibility and collaboration, this work aims to contribute to ongoing advancements in the field of deep learning for structured data, ensuring that future studies can build upon these findings to develop more efficient and effective solutions.

# References


[AP21] Sercan Ö. Arik and Tomas Pfister. "TabNet: Attentive Interpretable Tabular Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 8. 2021, pp. 6679–6687.

[Bak16] M. Baker. "1,500 scientists lift the lid on reproducibility". In: *Nature*. 533. 2016, pp. 452–454. DOI: 10.1038/533452a. URL: https://doi.org/10.1038/533452a.

[Bal+23] Maria Teresa Baldassarre et al. "(Re)Use of Research Results (Is Rampant)". In: *Commun. ACM* 66.2 (Jan. 2023), pp. 75–81. ISSN: 0001-0782. DOI: 10.1145/3554976. URL: https://doi.org/10.1145/3554976.

[BCH24] Houda Bichri, Adil Chergui, and Mustapha Hain. "Investigating the Impact of Train / Test Split Ratio on the Performance of Pre-Trained Models with Custom Datasets". In: *International Journal of Advanced Computer Science and Applications* 15.2 (2024). DOI: 10.14569/IJACSA.2024.0150235. URL: http://dx.doi.org/10.14569/IJACSA.2024.0150235.

[BCM21] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. "A Comparative Analysis of Gradient Boosting Algorithms". In: *Artificial Intelligence Review* 54.3 (2021), pp. 1937–1967. DOI: 10.1007/s10462-020-09896-5.

[BGJ20] Punam Bedi, Neha Gupta, and Vinita Jindal. "I-SiamIDS: an improved Siam-IDS for handling class imbalance in network-based intrusion detection systems". In: *Applied Intelligence* 51.2 (Sept. 2020), pp. 1133–1151. ISSN: 1573-7497. DOI: 10.1007/s10489-020-01886-y.

[Bla98] Jock Blackard. *Covertype*. UCI Machine Learning Repository. https://archive.ics.uci.edu/dataset/31/covertype, Accessed: 2024-11-11. 1998. DOI: 10.24432/C50K5N.

[BM98] C.L. Blake and C.J. Merz. *Statlog (Shuttle)*. UCI Machine Learning Repository. https://archive.ics.uci.edu/dataset/148/statlog+shuttle. 1998. DOI: 10.24432/C5WS31.

[Bol+23] Davide Boldini et al. "Practical Guidelines for the Use of Gradient Boosting for Molecular Property Prediction". In: *Journal of Cheminformatics* 15.1 (2023), p. 73. DOI: 10.1186/s13321-023-00743-7.

[Bow+02] Kevin W. Bowyer et al. "SMOTE: Synthetic Minority Over-sampling Technique". In: *CoRR* abs/1106.1813 (2002). arXiv: 1106.1813. URL: http://arxiv.org/abs/1106.1813.

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[DBL18] Georgios Douzas, Fernando Bacao, and Felix Last. "Improving imbalanced learning through a heuristic oversampling method based on k-means and SMOTE". In: *Information Sciences* 465 (2018), pp. 1–20. ISSN: 0020-0255. DOI: 10.1016/j.ins.2018.06.056. URL: http://dx.doi.org/10.1016/j.ins.2018.06.056.

[DT18] Suresh Dara and Priyanka Tumma. "Feature Extraction By Using Deep Learning: A Survey". In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. 2018, pp. 1795–1801. DOI: 10.1109/ICECA.2018.8474912.

[DV16] V. Dumoulin and F. Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016).

[Fan+21] Cheng Fan et al. "A Review on Data Preprocessing Techniques Toward Efficient and Reliable Knowledge Discovery From Building Operational Data". In: *Frontiers in Energy Research* 9 (2021). ISSN: 2296-598X. DOI: 10.3389/fenrg.2021.652801. URL: https://www.frontiersin.org/journals/energy-research/articles/10.3389/fenrg.2021.652801.

[Fay+22] Sheikh Amir Fayaz et al. "Is Deep Learning on Tabular Data Enough? An Assessment". In: *International Journal of Advanced Computer Science and Applications* 13.4 (2022). DOI: 10.14569/IJACSA.2022.0130454. URL: http://dx.doi.org/10.14569/IJACSA.2022.0130454.

[FM16] Michal Ferov and Marek Modrý. "Enhancing LambdaMART Using Oblivious Trees". In: *arXiv preprint arXiv:1609.05610* (2016).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[Hay09] Simon Haykin. *Neural Networks and Learning Machines*. 3rd. Pearson Education, 2009.

[He+08] Haibo He et al. "ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning". In: *Proceedings of the 2008 International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*. 2008, pp. 1322–1328. DOI: 10.1109/IJCNN.2008.4633969. URL: https://ieeexplore.ieee.org/document/4633969.

[He+15] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV]. URL: https://arxiv.org/abs/1502.01852.

[HLS22] Zhuo Huang, Yuang Liu, and Lewen Sun. "A Robust Multiple Network Attacks Detection Method Based on Artificial Neural Network". In: *2022 14th International Conference on Computer Research and Development (ICCRD)*. 2022, pp. 115–120. DOI: 10.1109/ICCRD54409.2022.9730311.

[Hua+17] G. Huang et al. "Densely Connected Convolutional Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 4700–4708.

[HWM05] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. "Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning". In: *Advances in Intelligent Computing*. Ed. by De-Shuang Huang, Xiao-Ping Zhang, and Guang-Bin Huang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 878–887. ISBN: 978-3-540-31902-3.

[HWS20] Ben Hermann, Stefan Winter, and Janet Siegmund. "Community Expectations for Research Artifacts and Evaluation Processes". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 469–480. ISBN: 9781450370431. DOI: 10.1145/3368089.3409767. URL: https://doi.org/10.1145/3368089.3409767.

[Jia+23] Xuezheng Jiang et al. "An adaptive multi-class imbalanced classification framework based on ensemble methods and deep network". In: *Neural Computing and Applications* 35 (2023), pp. 11141–11159. URL: https://link.springer.com/article/10.1007/s00521-023-08290-w.

[JK19] J.M. Johnson and T.M. Khoshgoftaar. "Survey on deep learning with class imbalance". In: *Journal of Big Data* 6.1 (2019), pp. 27–48. DOI: 10.1186/s40537-019-0192-5.

[KCP11] Mohammed Khalilia, Supriyo Chakraborty, and Mihail Popescu. "Predicting disease risks from highly imbalanced data using random forest". In: *BMC Medical Informatics and Decision Making* 11.1 (2011), p. 51. DOI: 10.1186/1472-6947-11-51. URL: https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/1472-6947-11-51.

[Kes+16] Nikita D. Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *arXiv preprint arXiv:1609.04836* (2016).

[KGS22] R. Kumar, P. Gupta, and S.P. Singh. "Comparative Analysis of Activation Functions in Deep Learning Models for Image Classification". In: *International Journal of Computer Applications* 182.24 (2022), pp. 1–5.

[KHM98] Miroslav Kubát, Robert C. Holte, and Stan Matwin. "Machine Learning for the Detection of Oil Spills in Satellite Radar Images". In: *Machine Learning* 30 (1998), pp. 195–215. URL: https://api.semanticscholar.org/CorpusID:1795112.

[Koh94] Ron Kohavi. "Bottom-Up Induction of Oblivious Read-Once Decision Graphs: Strengths and Limitations". In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press. 1994, pp. 613–618.

[Koh95] Ron Kohavi. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection". In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*. 1995.

[Kri09] Alex Krizhevsky. *CIFAR-100*. University of Toronto. https://www.cs.toronto.edu/~kriz/cifar.html, Accessed: 2024-11-17. 2009.

[Lai+22] Can Lai et al. "Forest Fire Prediction with Imbalanced Data Using a Deep Neural Network Method". In: *Forests* 13.7 (2022). ISSN: 1999-4907. DOI: 10.3390/f13071129. URL: https://www.mdpi.com/1999-4907/13/7/1129.

[LCB94] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. *CIFAR-100*. https://yann.lecun.com/exdb/mnist/, Accessed: 2024-11-17. 1994.

[Lec+98] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[Lee23] Minhyeok Lee. *GELU Activation Function in Deep Learning: A Comprehensive Mathematical Analysis and Performance*. 2023. arXiv: 2305.12073 [cs.LG]. URL: https://arxiv.org/abs/2305.12073.

[Liu+22] Yao Liu et al. "Solving the class imbalance problem using ensemble algorithm: application of screening for aortic dissection". In: *BMC Medical Informatics and Decision Making* 22 (2022). DOI: 10.1186/s12911-022-01821-w.

[LKR20] Arash Habibi Lashkari, Gurdip Kaur, and Abir Rahali. *DIDarknet: A Contemporary Approach to Detect and Characterize the Darknet Traffic using Deep Image Learning*. 10th International Conference on Communication and Network Security, Tokyo, Japan. https://www.unb.ca/cic/datasets/darknet2020.html, Accessed: 2024-11-14. 2020.

[LQG21] Yiran Liu, Xu Qiao, and Rui Gao. "Plankton Classification on Imbalanced Dataset via Hybrid Resample Method with LightBGM". In: *2021 6th International Conference on Image, Vision and Computing (ICIVC)*. 2021, pp. 191–195. DOI: 10.1109/ICIVC52351.2021.9526988.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[McK+10] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[NCK11] Hien M. Nguyen, Eric W. Cooper, and Katsuari Kamei. "Borderline oversampling for imbalanced data classification". In: *Int. J. Knowl. Eng. Soft Data Paradigm.* 3.1 (2011), pp. 4–21. ISSN: 1755-3210. DOI: 10.1504/IJKESDP.2011.039875. URL: https://doi.org/10.1504/IJKESDP.2011.039875.

[Ped+11] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[PK23] Ekaterina Poslavskaya and Alexey Korolev. *Encoding categorical data: Is there yet anything 'hotter' than one-hot encoding?* 2023. arXiv: 2312.16930 [cs.LG]. URL: https://arxiv.org/abs/2312.16930.

[Pro+19] Liudmila Prokhorenkova et al. "CatBoost: unbiased boosting with categorical features". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 31. 2019.

[Ren+23] Zhijun Ren et al. "A Systematic Review on Imbalanced Learning Methods in Intelligent Fault Diagnosis". In: *IEEE Transactions on Instrumentation and Measurement* 72 (2023), pp. 1–35. DOI: 10.1109/TIM.2023.3246470.

[RHW86] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[Sas+23] Leonard Sasse et al. "On Leakage in Machine Learning Pipelines". In: *arXiv preprint arXiv:2311.04179* (2023). URL: https://arxiv.org/abs/2311.04179.

[Sha+20] Jie Shao et al. "Is normalization indispensable for training deep neural networks?" In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[Sig23] Fabio Sigrist. *A Comparison of Machine Learning Methods for Data with High-Cardinality Categorical Variables*. 2023. arXiv: 2307.02071 [cs.LG]. URL: https://arxiv.org/abs/2307.02071.

[Smi17] Leslie N. Smith. "A Bayes-Optimal Approach to Batch Size Selection". In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2017.

[SR20] Fahad Sohrab and Jenni Raitoharju. "Boosting Rare Benthic Macroinvertebrates Taxa Identification With One-Class Classification". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, Dec. 2020, pp. 928–933. DOI: 10.1109/ssci47803.2020.9308359. URL: http://dx.doi.org/10.1109/SSCI47803.2020.9308359.

[SS97] J. Sola and J. Sevilla. "Importance of input data normalization for the application of neural networks to complex industrial problems". In: *IEEE Transactions on Nuclear Science* 44.3 (1997), pp. 1464–1468. DOI: 10.1109/23.589532.

[Sto+99] Sal Stolfo et al. *KDD Cup 1999 Data*. UCI KDD Archive. Irvine, CA: University of California, Department of Information and Computer Science. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, Accessed: 2024-11-14. 1999.

[Tan+21] Jimin Tan et al. *A critical look at the current train/test split in machine learning*. 2021. arXiv: 2106.04525 [cs.LG]. URL: https://arxiv.org/abs/2106.04525.

[UGM14] Ryan J. Urbanowicz, Delaney Granizo-Mackenzie, and Jason H. Moore. "An Extended Michigan-Style Learning Classifier System for Flexible Supervised Learning, Classification, and Data Mining". In: *Proceedings of the 13th International Conference on Parallel Problem Solving from Nature (PPSN XIII)* (2014), pp. 211–220. DOI: 10.1007/978-3-319-10762-2_21.

[UM09] Ryan J. Urbanowicz and Jason H. Moore. "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap". In: *International Journal of Artificial Intelligence and Machine Learning* 6.1 (2009), pp. 5–30. DOI: 10.1155/2009/736398.

[VD95] Guido Van Rossum and Fred L Drake Jr. *Python Reference Manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[Win+22] Stefan Winter et al. "A retrospective study of one decade of artifact evaluations". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 145–156. ISBN: 9781450394130. DOI: 10.1145/3540250.3549172. URL: https://doi.org/10.1145/3540250.3549172.

[Woo+23] Sanghyun Woo et al. *ConvNeXt V2: Co-designing and Scaling ConvNets with Masked Autoencoders*. 2023. arXiv: 2301.00808 [cs.CV]. URL: https://arxiv.org/abs/2301.00808.

[WSL23] Haizhou Wang, Anoop Singhal, and Peng Liu. "Tackling imbalanced data in cybersecurity with transfer learning: a case with ROP payload detection". In: *Cybersecurity* 6.1 (2023), p. 2. DOI: 10.1186/s42400-022-00135-8.

[Xu+21] Jing Xu et al. *RegNet: Self-Regulated Network for Image Classification*. 2021. arXiv: 2101.00590 [eess.IV]. URL: https://arxiv.org/abs/2101.00590.

[Zha+20] Hang Zhang et al. *ResNeSt: Split-Attention Networks*. 2020. arXiv: 2004.08955 [cs.CV]. URL: https://arxiv.org/abs/2004.08955.

[Zho+21] Hong-Yu Zhou et al. *ConvNets vs. Transformers: Whose Visual Representations are More Transferable?* 2021. arXiv: 2108.05305 [cs.CV]. URL: https://arxiv.org/abs/2108.05305.

[ZU20] Robert F. Zhang and Ryan J. Urbanowicz. "A scikit-learn compatible learning classifier system". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*. GECCO '20. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 1816–1823. ISBN: 9781450371278. DOI: 10.1145/3377929.3398097. URL: https://doi.org/10.1145/3377929.3398097.

# Appendices

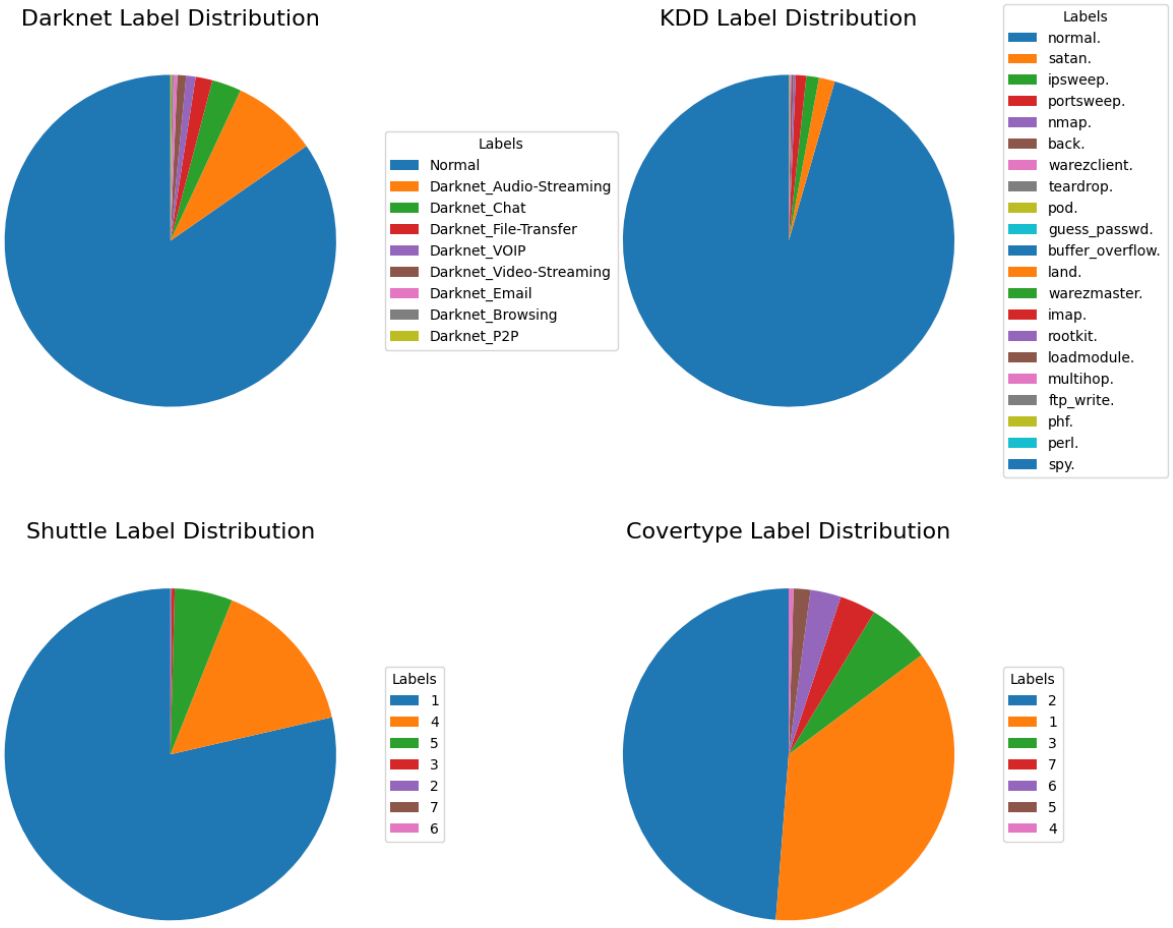


Figure 6: Training Dataset Label Distributions (Original)

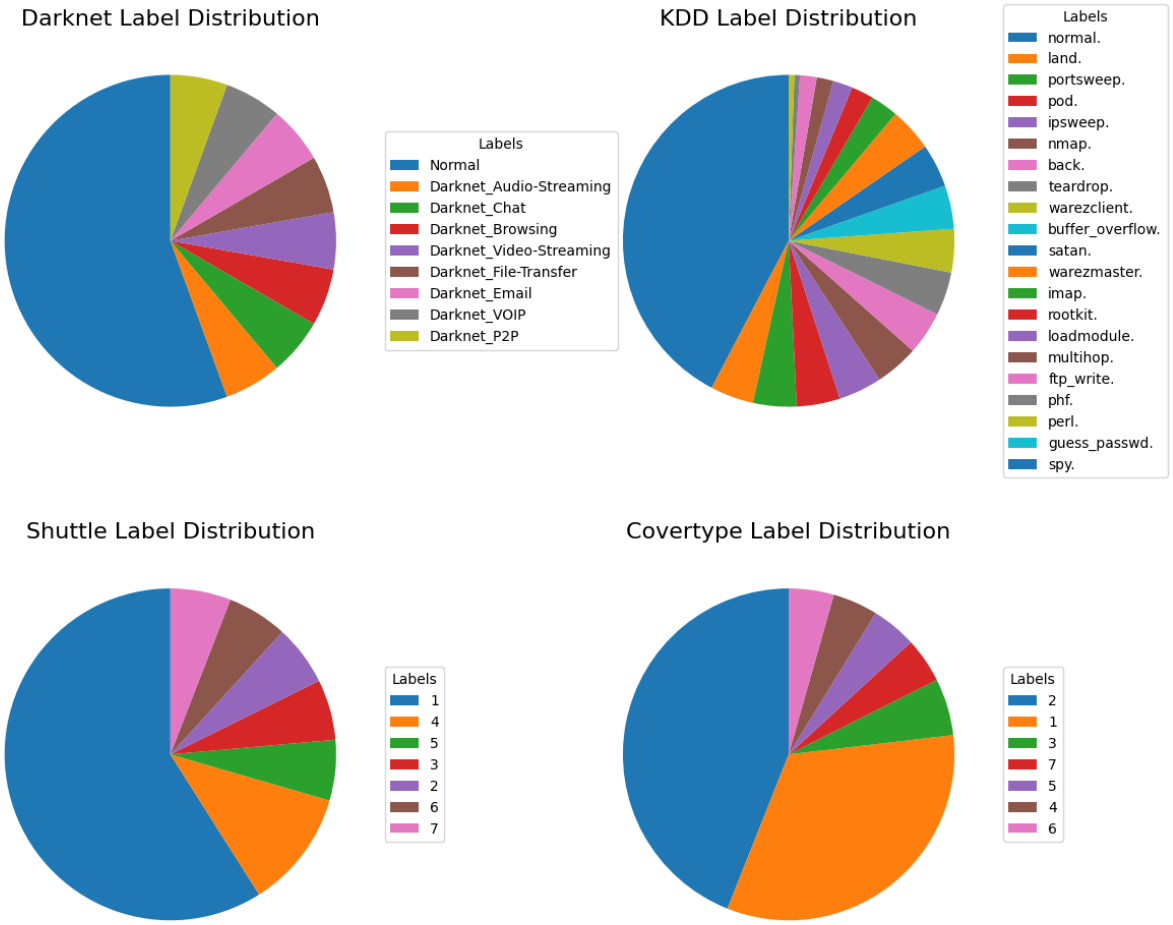


Figure 7: Training Dataset Label Distributions (BorderlineSMOTE)

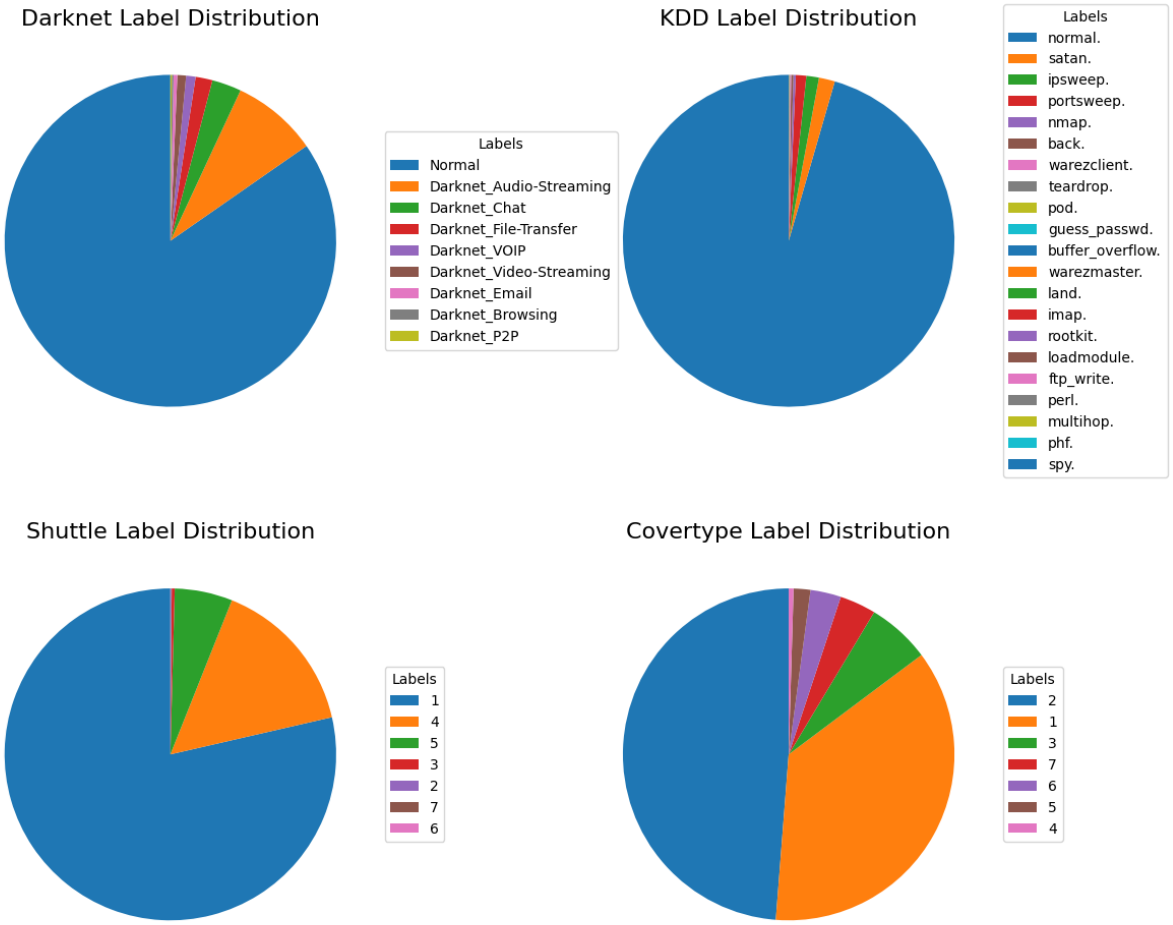


Figure 8: Test Dataset Label Distributions

| | Shuttle | Covertype | KDD-99 | Darknet |
|---|---|---|---|---|
| Version 1 | 5 | 5 | 5 | 5 |
| Version 2 | 5 | 5 | 5 | 5 |
| Version 3 | 5 | 5 | 5 | 5 |
| Version 4 | 5 | 5 | 5 | 5 |
| Version 5 | 5 | 5 | 5 | 5 |
| Version 6 | 5 | 5 | 5 | 5 |
| Version 7 | 5 | 5 | 5 | 5 |
| Version 8 | 5 | 5 | 5 | 5 |
| Version 9 | 5 | 5 | 5 | 5 |
| Version 10 | 5 | 5 | 5 | 5 |
| Version 11 | 5 | 5 | 5 | 5 |
| Version 12 | 5 | 5 | 5 | 5 |
| Version 13 | 5 | 5 | 5 | 5 |
| Version 14 | 5 | 5 | 5 | 5 |
| Version 15 | 5 | 5 | 5 | 5 |
| Version 16 | 5 | 5 | 5 | 5 |
| Version 17 | 5 | 5 | 5 | 5 |
| Version 18 | 5 | 5 | 5 | 5 |
| Version 19 | 5 | 5 | 5 | 5 |
| Version 20 | 5 | 5 | 5 | 5 |
| Version 21 | 5 | 5 | 5 | 5 |
| Version 22 | 10 | 10 | 10 | 10 |
| Version 23 | 5 | 5 | 5 | 5 |
| Version 24 | 10 | 10 | 10 | 10 |
| Version 25 | 5 | 5 | 5 | 5 |
| Version 26 | 10 | 10 | 10 | 10 |
| Version 27 | 5 | 5 | 5 | 5 |
| Version 28 | 10 | 5 | 2 | 5 |
| Version 29 | 10 | 10 | 2 | 10 |
| Version 30 | 50 | 10 | 2 | 9 |
| Version 31 | 50 | 10 | 3 | 9 |
| Version 32 | 50 | 4 | 3 | 5 |
| Version 33 | 50 | 23 | 8 | 31 |
| Version 34 | 50 | 27 | 7 | 31 |
| Version 35 | 50 | 8 | 3 | 15 |
| Version 36 | 50 | 14 | 2 | 12 |
| Version 37 | 50 | 24 | 4 | 19 |
| Version 38 | 50 | 33 | 3 | 18 |
| Version 39 | 50 | 20 | 8 | 20 |

Table 47: Amount of Runs per Dataset & per Architecture Version

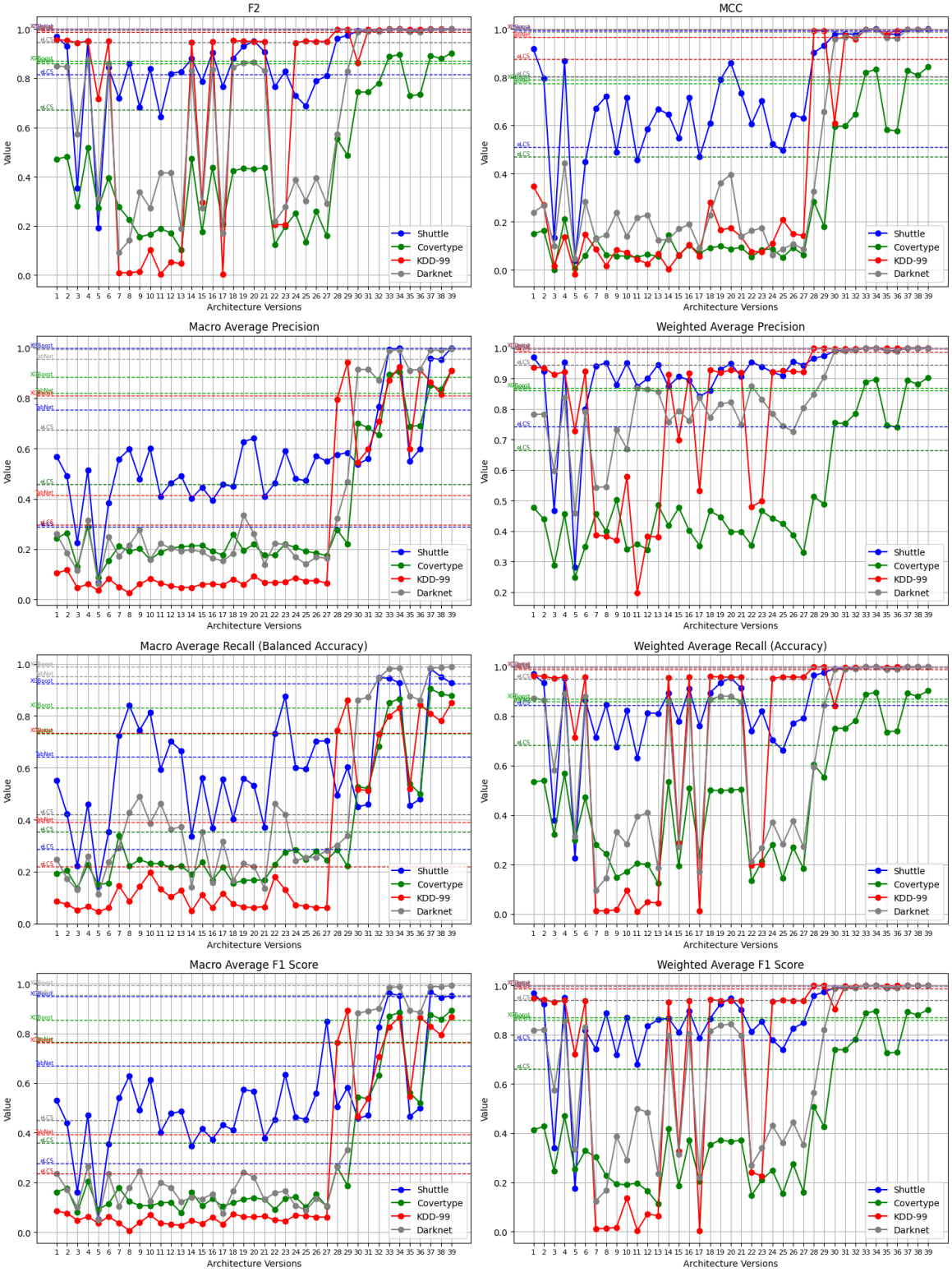


Figure 9: Performance Metric Values for Each Architecture Version (w. Comparison)

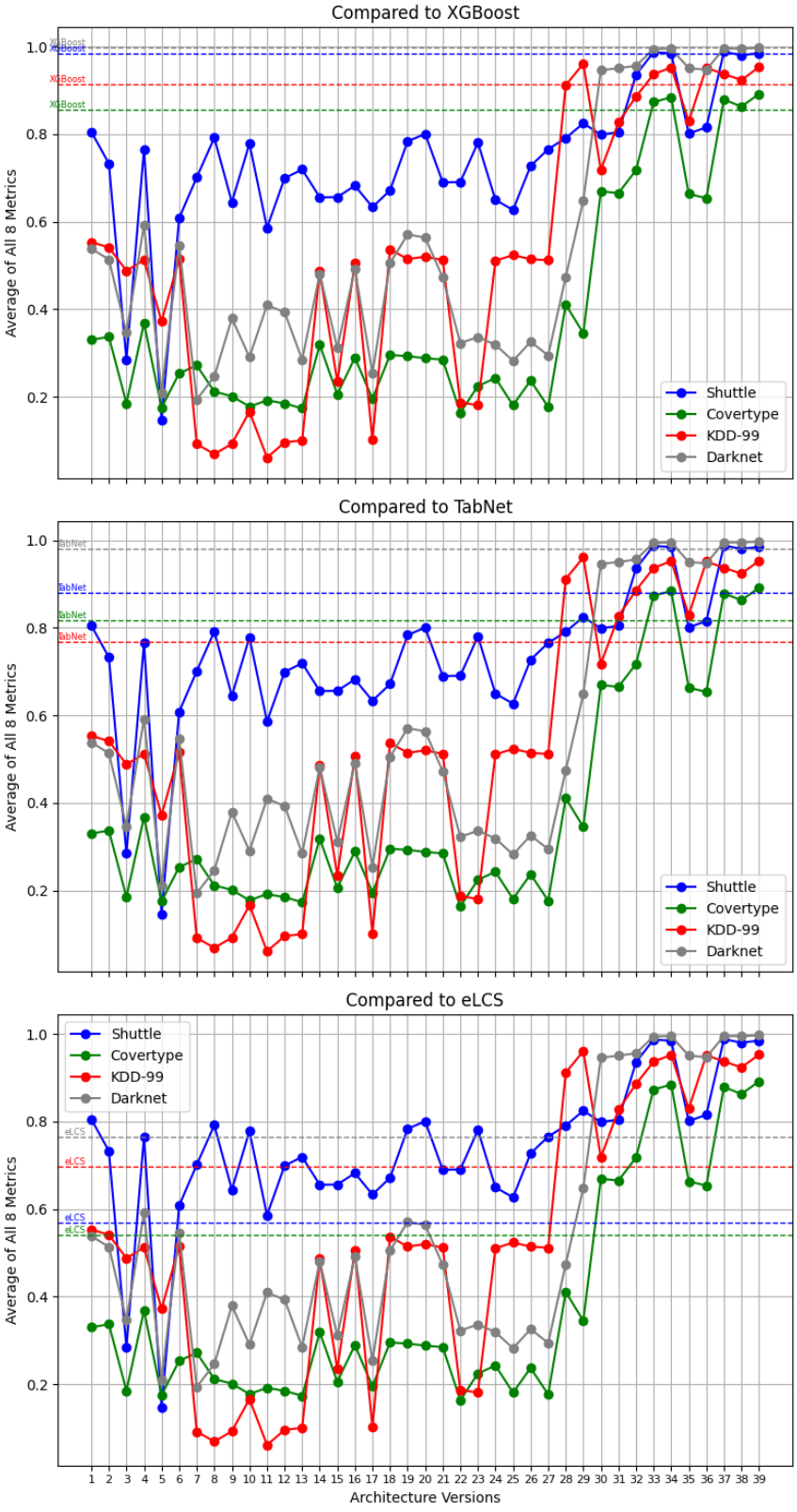


Figure 10: Average of 8 Metrics Across Architecture Versions for Each Dataset

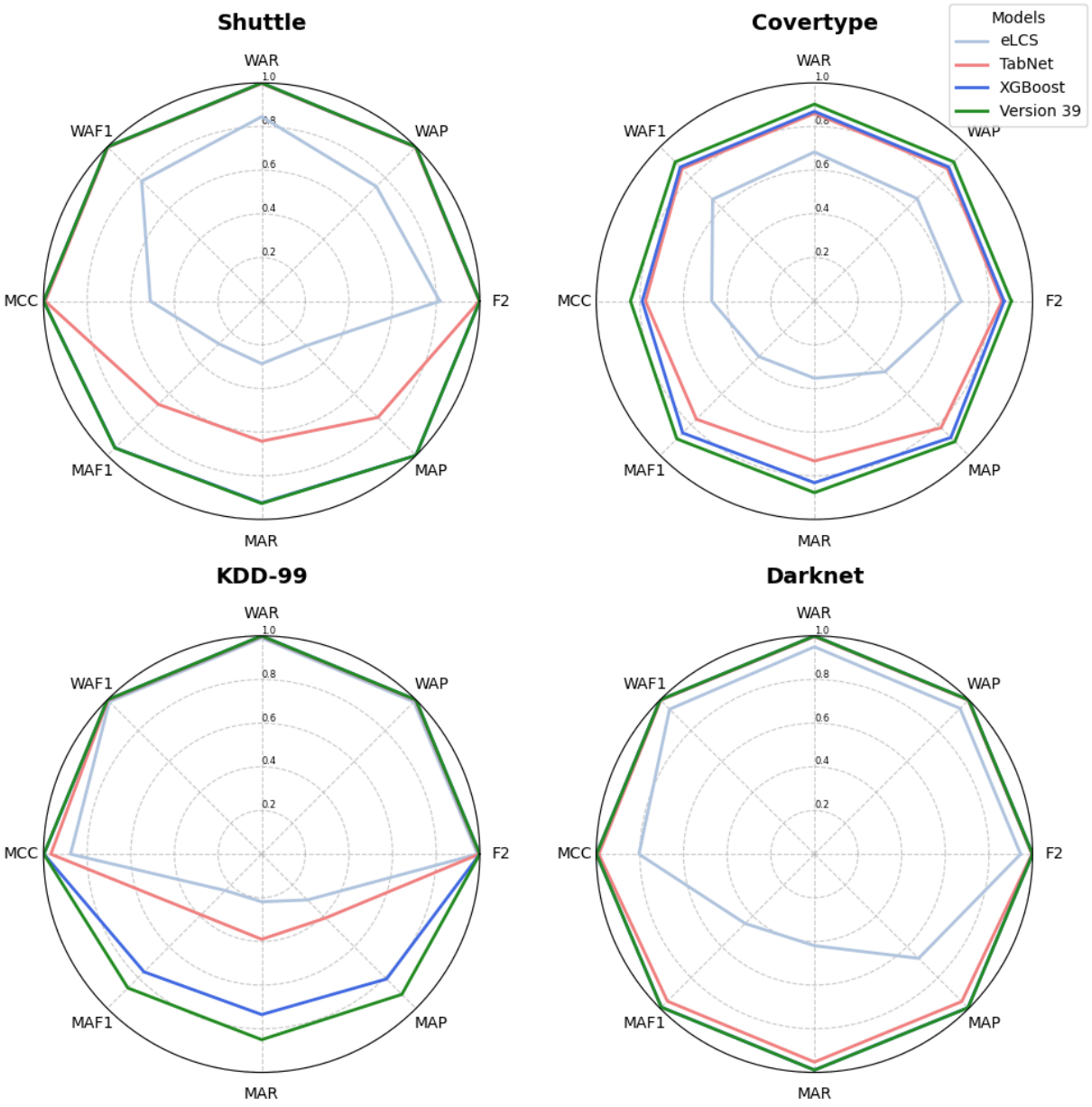


Figure 11: Model Comparison Using Radar Plots