Adapting to Data Drift in Encrypted Traffic Classification Using Deep Learning

by

Navid Malekghaini

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

Statement of Contributions

Content and images from the paper “Data Drift in DL: Lessons Learned from Encrypted Traffic Classification” [35] are borrowed in Chapters 1 through 5.

Abstract

Deep learning models have shown to achieve high performance in encrypted traffic classification. However, when it comes to production use, multiple factors challenge the performance of these models. The emergence of new protocols, especially at the application-layer, as well as updates to previous protocols affect the patterns in input data, making the model's previously learn patterns obsolete. Furthermore, proposed model architectures are usually tested on datasets collected in controlled settings, which makes the reported performances unreliable for production use. In this thesis, we start by studying how the performances of two high-performing state-of-the-art encrypted traffic classifiers change on multiple real-world datasets collected over the course of two years from a major ISP's network, Orange telecom. We investigate the changes in traffic data patterns highlighting the extent to which these changes, *a.k.a. data drift*, impact the performance of the two models in service-level and application-level classification. We propose best practices to manually adapt model architectures and improve their accuracy in the face of data drift. We show that our best practices are generalizable to other encryption protocols and different levels of labeling granularity. However, designing efficient model architectures and manual architectural adaptations is time-consuming and requires domain expertise. Neural architecture search (NAS) algorithms have been shown to automatically discover efficient models in other domains, such as image recognition and natural language processing. However, NAS's application is rather unexplored in Encrypted Traffic Classification. We propose AutoML4ETC, a tool to automatically design efficient and high-performing neural architectures for Encrypted Traffic Classification, given a target dataset and corresponding features. We define three powerful search spaces tailored specifically for the prominent categories of features in the Encrypted Traffic Classification state-of-the-art, i.e., packet raw bytes, flow time-series, and flow statistics. We show that a simple search strategy over AutoML4ETC's search spaces can generate model architectures that outperform the state-of-the-art Encrypted Traffic Classification models on several benchmark datasets, including real-world datasets of TLS and QUIC traffic collected from a major ISP network. In addition to being more accurate, the AutoML4ETC's architectures are significantly more efficient and lighter in terms of the number of parameters. We further showcase the potential of AutoML4ETC by experimenting with state-of-the-art NAS techniques and model ensembles generated from different search spaces. We also use AutoML4ETC to analyze the state of adoption of the QUIC protocol.

Acknowledgements

I want to express my gratitude to my supervisor, Professor Raouf Boutaba, for his tireless assistance, encouragement, and compassion in both this study and our other research endeavours.

I would also like to thank Elham Akbari Azirani, my dear friend and colleague who supported me throughout the hard times and assisted me with the related works of the research. Iman Akbari Azirani who helped me to understand and on-board the research project as easy as possible. Professor Noura Limam for her continuous help with publications and thoughtful comments. Professor Mohammad Ali Salahuddin for his suggestions and help writing the papers. Matheus Vrech Silveira Lima with assisting on the QUIC labeling procedure. Our colleagues in Orange Labs for providing the datasets used in this thesis. And all my friends that supported me throughout these years.

My kindest and deepest gratitude to my parents and sister for their help and affection, which helped me strive harder and accomplish more in both my personal life and my time in the University of Waterloo.

Dedication

To all the innocent lives that were taken during the Iran protests and Ukraine war. To anyone who helps other human beings unconditionally. In memory of #mahsaamini.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning (DL) models have shown superior performance in encrypted traffic classification [42, 4, 31]. However, when it comes to deploying a DL model in production, there is more to consider than model performance, which is dependent on the target dataset. In practice, the model performance on a given dataset is tightly coupled with the intrinsic properties of the dataset. The effect of the target dataset on model accuracy has been previously highlighted by comparing the performances of traffic classification models on datasets [4, 3].

The need for datasets with sample distributions that reflects real-world data is a known issue in traffic classification. The fact that network traffic datasets are often collected under controlled settings or generated synthetically is not due to a dismissal of this principle, but rather it is a reflection of the difficulty of labeling real-world network traffic. Even if perfectly collected and labeled data existed at some point, it is likely to become irrelevant just a few months later, due to the dynamic nature of network traffic. Over time, network traffic patterns are affected by the protocols, software, and devices that generate them. This pattern evolution is known as *data drift*, also known as *concept drift*, in the machine learning (ML) literature.

Data drift is a phenomenon in which the distribution of input data over classes changes with time. For example, a service may switch to another transport protocol leading to a different flow time-series (i.e., traffic shape). A flow time-series-based classifier is then likely to decay in identifying the new traffic. Hence, data drift refers to a change in the distribution of real-world data caused by the network's dynamic nature, which leads to model decay, significantly impacting model performance.

In this thesis, we study the effect of data drift on the performance of two state-of-the-art encrypted network traffic classifiers [4, 42]. Using several datasets of real-world network traffic collected from a major ISP's mobile network, we show that model performance degradation does indeed occur in a production setting, i.e., when a model trained on (i.e., seen) old data attempts to classify new data (i.e., unseen). We offer an explanation for the degradation, based on traffic input that the models struggle on. We also analyse the architecture of the models, offering guidelines for designing architectures that we empirically show are more robust to data drift. In practice, several factors in the data collection process affect the number of possible labeled samples, and the datasets on which the models train can be of various sizes. Therefore, we also study the effect of dataset size on model performance. The above findings clearly emphasise the need for re-training a traffic classifier over several, newer datasets in practice. More importantly, we suggest that for a classifier to remain effective over time, a domain expert must *manually* re-tune the hyper-parameters of the model, e.g., model architecture, which is time-consuming and primarily based on trial-and-error. Therefore, we also focuse on *automatically* finding an efficient model architecture for encrypted traffic classification, given a specific network traffic dataset.

The problem of choosing the suitable model hyper-parameters is not new to DL. It has been previously addressed under meta-learning [48, 21], where the use of a *supervisory* neural network was suggested to learn the hyper-parameters of a *subordinate* neural network. Recently, the related domain of Neural Architecture Search (NAS) [16, 63] was introduced as a sub-field of Automated Machine Learning (AutoML) [18], which addresses the problem of learning the best architecture for a neural network. The introduction of NAS was a response to the need for extensive model "architectural engineering" for each new image classification dataset, which closely aligns with the practical problem described above for traffic classification. NAS requires choosing building blocks for the target architecture, which constitutes the *search space* that requires domain knowledge. Therefore applying NAS for Encrypted Traffic Classification, which is a non-trivial task.

The main contributions of this thesis can be summarized as:

- We study the data drift phenomenon using five real-world TLS datasets collected over a course of more than two years from a major ISP's mobile network. To the best of our knowledge, we are the first to address the problem of data drift in real-world encrypted traffic classification.

- We provide insights into the type of data drift that happens in network traffic at different levels of labeling granularity, i.e., service-level and application-level classes. These insights are useful to practitioners working with traffic classification models in production.

- We perform an ablation study to analyze the impact of data drift on two state-of-the-art features for encrypted traffic classification: (i) TLS header bytes, and (ii) flow time-series information. We reason data drift on these features, and quantify the drift per service class and corresponding applications.

- We offer guidelines for designing models that are robust to a change of dataset, labeling granularity, and encryption protocol. Our guidelines have the distinction of being empirically tested on real-world data with different encryption protocols and for both service- and application-level classification.

- We propose AutoML4ETC, a tool to automatically design efficient and high-performing neural network architectures for Encrypted Traffic Classification, given a target dataset and corresponding features. We show that in addition to being more accurate, the AutoML4ETC's architectures are significantly more efficient and lighter in terms of the number of parameters than baseline state-of-the-art encrypted traffic classifiers, while the amount of resources needed for designing such architectures is reasonable.

- We define three powerful search spaces tailored specifically for the prominent categories of features in the Encrypted Traffic Classification state-of-the-art: (i) packet raw bytes, (ii) flow time-series, and (iii) flow statistics. We show that a simple search strategy over AutoML4ETC's search spaces can generate model architectures that outperform state-of-the-art Encrypted Traffic Classification models on several benchmark datasets, including real-world TLS and QUIC traffic collected from a major ISP network.

- We evaluate the potential of AutoML4ETC by experimenting with state-of-the-art neural architecture search techniques and model ensembles generated from the same or across search spaces.

- We provide traffic measurement estimations for TLS and QUIC protocols with Encrypted Traffic Classification to analyze the state of adoption of the QUIC protocol with our real-world datasets. Additionally, results on the publicly available QUIC dataset are also presented to show the generalizability of our approach to other (non-proprietary) datasets.

Chapter 2

Background & Related Works

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of ML that is influenced by behavioural psychology. . It pertains to how software/hardware *agents* should operate in a given *environment* to maximize a cumulative *reward*. The RL algorithm uses *states* to observe the *environment* and find the appropriate *action* for the *agent* at a given step. The process is iterative, i.e., the RL agent executes different *actions* and monitors the *states* and *rewards* to finally converge to a set of rules or *policy* for choosing the best *action* based on a given *state*.

Because of the iterative nature of RL algorithms, there is always a trade-off between choosing the optimal action from the policy (i.e., *exploit*) or try other actions (i.e., *explore*) and see the outcome at a given step. *REINFORCE* [61] is one of the oldest RL policy-based algorithms that addresses the trade-off with stochastic policies i.e., the action is chosen from a probability distribution. Another solution to balance the trade-off is to use an Upper Confidence Bound (UCB) with exploitation. In Equation 2.1, the term $argmax_a\tilde{R}(a)$ uses exploitation to choose the action that achieves the highest reward. Moreover, the addition of $\sqrt{\frac{2\ln n}{n_a}}$ results in higher exploration by incorporating the number of times a specific action a is chosen (i.e., n_a) out of the total chosen actions (i.e., n).

$$argmax_a\tilde{R}(a) + \sqrt{\frac{2\ln n}{n_a}} \tag{2.1}$$

2.2 Deep Learning Background

A Multi Layer Preceptron is simply a set of interconnected perceptrons (i.e., artificial neurons) stacked into dense layers. These dense layers can capture smaller patterns in the input data, so the full model would be able to solve more complex problems given a larger number of layers. An MLP network is used to consume input data that are independent. In other words, the data points in the training set can be reordered without affecting the result.

However, if the input data is of a sequential nature, then Recurrent Neural Networks (RNNs) must be used. The Long Short Term Memory (LSTM) model, an RNN, is capable of learning long term dependencies with a dataset. Due to this nature, LSTM is a popular component in neural networks for natural language processing. A popular use for LSTMs is sequence-to-sequence learning, which pretains to generating a sequence as output from a sequence of input (e.g., generating future model descriptions from current model descriptions).

Another useful neural network is Convolutional Neural Networks (CNNs). The advantage of CNNs is in capturing patterns spatial-invariant in the input data by using filters. Moreover, the use of filters and shifting it over the input with shared training parameters, reduces the number of training parameters significantly compared to MLP. This feature makes CNNs an ideal choice for network traffic raw bytes feature where information can be shifted in the input data.

Furthermore, Dropout layers in deep learning are useful to improve the generalizability of the model. The dropout layer sets the weights of the previous layer to zero with a given probability. This will force the model to try to learn from different parts of the information input from the previous layer thus, making it more generalizeable. Additionally, another hyperparameter to discuss is the Learning Rate. An optimization approach known as gradient descent is used to train deep-learning neural networks. When the model weights are changed, a hyperparameter called Learning Rate determines how much the model weights will be tweaked to account for the anticipated loss.

2.3 Encrypted Traffic Classification

In light of the obfuscation of previously reliable features by encryption, such as application-layer payload, the traffic classification literature turned to features (e.g., packet size, timestamp, direction and their statistics) that were difficult to tweak without affecting quality of service. Before the advent of DL, the performance of several traditional supervised ML models, such as Naïve Bayes, AdaBoost, and Support Vector Machine (SVM), was evaluated using these features for encrypted traffic classification (e.g., [28, 5]). Furthermore, semi-supervised approaches based on Gaussian Mixture Models, k-Means, k-Nearest Neighbour clustering, and Multi-Objective Genetic Algorithms were studied for real-time encrypted traffic classification (e.g., [12, 8, 9]). A survey of traditional ML approaches is available in [56, 13].

The capacity to automatically extract feature vectors from raw data in DL provided new opportunities for encrypted traffic classification. These opportunities were explored using various DL models including Multi-Layer Perceptrons, Stacked Autoencoders, Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) (e.g., [42, 31, 44, 25]). The models were primarily evaluated using public mixed-protocol datasets such as ISCXVPN2016 [30] and ISCXIDS2012 [51]. The work in [3] uses a proprietary dataset to evaluate numerous application-level classification methods that use DL models. A survey of DL models used for network traffic analysis is available in [2].

2.4 Data Drift

Several works in Website fingerprinting (WF) attacks against The Onion Router (Tor) observe the need for retraining ML models on fresh traffic traces to ensure attack effectiveness. Herrmann et al. [20] show that their Naïve Bayes approach is robust to data drift when the training and test data were collected within two days of one another. Their approach works on packet size and direction sequences of flows similar to the flow time-series feature used in this thesis. Rimmer et al. [45] evaluate the resilience of several state-of-the-art DL models to data drift on traffic periodically collected over two months. The authors show that different DL models age differently, with their accuracies dropping from around 95% to between 55% and 75% in the course of two months. A critical study of WF attacks [23] evaluates the effect of data staleness on WF by measuring an SVM-based classifier's accuracy on the data over the course of 90 days. The data is gathered by crawling Alexa Top 100 pages at different instants in time. The authors show that the classifier's accuracy drops from around 80% to around zero in less than 90 days when the number of sites (i.e., the number of classes) is 100, with the accuracy dropping below 50% in less than 10 days. To provide a solution to data staleness, the authors in [59] and [53] propose models that use less data to train, so that crawling the websites and collecting the necessary traces to re-train the model is feasible in the small window of time in which the model decays.

Andresini et al. [6] address robustness to data drift for intrusion detection in network traffic, a context in which data drift is especially important because of the continuously evolving nature of attacks. Their proposed approach consists of three phases: (i) initial training on historical data, (ii) incremental learning on unlabelled data facilitated by a learned oracle, and (iii) an explanation phase for how the model adapts to new attack categories. The authors use variable length time windows that span several minutes rather than fixed time splits to evaluate the model. Their approach is evaluated on a recently published version of the CICIDS2017 dataset [17], a dataset of benign and malware traffic traces spanning over 5 days. The time window they consider is much shorter than the time windows considered in this thesis. Furthermore, their domain is also different to ours (i.e., intrusion detection versus service/application detection).

Ma et al. [34] propose a framework to detect and adjust to data drift in an anomaly detection system. The authors define data drift as a sudden change in the distribution of the key performance indicator (KPI) stream. Since the number of KPIs in their base anomaly detector is large, they especially focus on automatic threshold setting for the data drift detection algorithm to free operators from manually tuning per-KPI parameters. Their data drift adaptation algorithm is based on linearly transforming the new concept to the old concept in each time window. Their work differs from ours, as they deal with a different domain where input data is in the form of a continuous stream, so applying standard data drift algorithms to their domain is rather straightforward.

Saurav et al. [47] consider the problem of an anomaly detection model losing its relevance when trained on historical data and used in a dynamically changing and non-stationary environment, where the definition of normal behavior changes. Their proposed model, a recurrent neural network (RNN) trained incrementally on a data stream, is used to make predictions while continuously adapting to new data when prediction errors increase. They show that their model is able to adapt to different types of data drift, e.g., sudden, gradual and incremental drift.

Taylor et al. [54] study the effect of training on one dataset and testing on another, building up on their previous work AppScanner, an automatic tool for fingerprinting smartphone apps from encrypted data. They collect five datasets of app generated traffic, four of which were collected six months after the first one and differ from the first one in a subset of three factors: (i) time of collection, (ii) app device, and (iii) app version. The authors test the effect of each factor on the accuracy of the model when trained on the base dataset and tested on the target dataset, and conclude that mere time passing has the least effect on the model's accuracy, whereas the model's accuracy drops from around 70% to 19% when tested on the dataset with new app versions and devices.

Although the work in [54] is based on traditional ML models, it relates to ours in

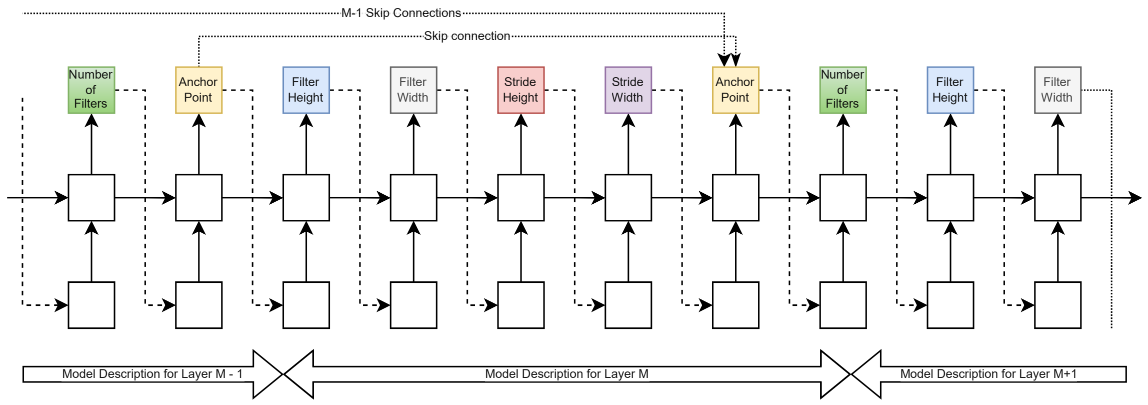


Figure 2.1: Generating deep model descriptions and skip connections for each layer with a controller RNN [63]

the recognition of the effect of ambiguous flows in confounding the classifier, as well as confirming the phenomenon of model decay in mobile app fingerprinting. As opposed to the synthetic datasets employed in [54], our work is based on real-world datasets.

2.5 Neural Architecture Search

NAS, first proposed in [63], leverages a RNN to generate a sequence that represents a neural network architecture, as depicted in Figure 2.1. For example, assuming that the architecture consists of convolutional layers only, each layer in the CNN is described by a sequence of tokens. Each token determines a separate characteristic of the convolutional layer, such as filter size and stride. The sequences denoting each convolutional layer follow. The possible combinations and permutations of the sequences determine the models that can potentially be generated, i.e., the search space. The controller RNN is trained by RL where actions are the choice of tokens and the reward signal is the validation accuracy of the model specified by the sequence. RL consists of a series of trials, where a child model is created by sampling the parameter values generated by the RNN at the end of each trial. As shown in Figure 2.2, the sampled model is trained and evaluated on a dataset per trial to compute the reward function, which makes NAS computationally expensive.

The RL algorithm described above operates within a space of possible sequences. This space is decided by the set of possible tokens that the controller RNN can generate at each time step. It is up to a domain expert to determine the set of possible tokens for RNN,

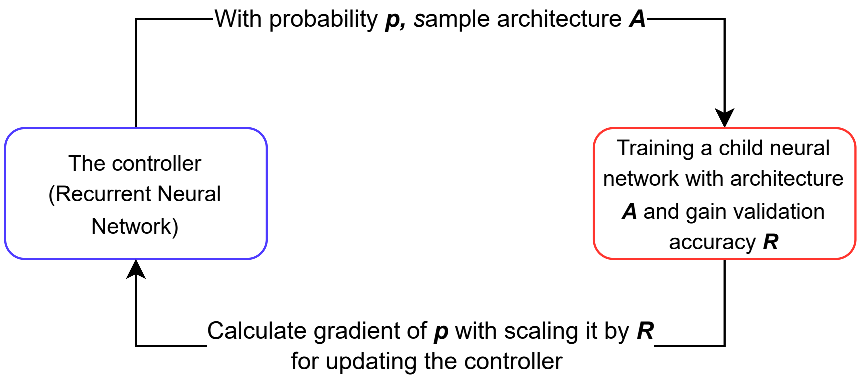

With probability ***p,*** sample architecture ***A***
The controller (Recurrent Neural Network)
Training a child neural network with architecture ***A*** and gain validation accuracy ***R***
Calculate gradient of ***p*** with scaling it by ***R*** for updating the controller

Figure 2.2: Overview of Neural Architecture Search with RL [63]

which is similar to the set of words in the dictionary of a language generator. Authors in [63] propose two different search spaces for creating both CNNs and RNNs. The authors further increase the complexity of the convolutional models by introducing *anchor points* into the search space, which determine the probability of skip connections existing between a layer and its previous layers, allowing the architecture to contain branching or skip connections similar to the ones in *ResNet* [19]. Their results show that the generated CNN models perform within a 1% error rate of the state-of-the-art image classifiers on the CIFAR-10 [27] dataset. However, this is achieved by training 12,800 architectures in total, using 800 GPUs for concurrently training 800 models, which makes such experiments resource intensive.

Authors in [64] enhance the search space or the set of tokens, i.e., architectural building blocks that the RNN generates in [63]. Their work is based on the observation that state-of-the-art image classifier architectures have repeated network motifs, i.e., small building blocks in the architecture's graph that are repeated. Their proposed search space consists of a sequence of *Normal cells* and *Reduction cells*, in which only the reduction cells reduce the size of the feature map. The Interior structure of the normal and reduction cells varies between different architectures in the search space. Each cell is made up of a constant number of network motifs, where each motif consists of two inputs fed into two blocks aggregated by a function. The types of blocks (e.g., separable convolution, identity, 1x1 convolution) and the aggregation function (e.g., add or concatenate) are determined by the controller RNN along with the connections between the motifs. Their method performs slightly better than the best record on CIFAR-10, with the added benefit of being transferable to the larger ImageNet dataset despite the computational complexity of NAS. The authors leverage a transfer learning approach to speed up child model training and ensure transferability.

Both previous approaches suffer from high computational complexity. To tackle this problem, [38] improve NAS's time efficiency by a factor of 1000 and ensure the best error rate within 0.3% of NAS, by introducing parameter sharing among all child models, which is inspired by the transfer learning approach [64] and multi-task learning [46]. The authors named their approach *Efficient Neural Architecture Search* (ENAS). ENAS uses a more restrictive search space in which the only child models considered are the ones that can be represented by a directed acyclic graph (DAG). Moreover, the authors propose a *micro search space* in which non-separable convolutional blocks are not considered. The results of the micro search space are then compared to those of the search space in NAS, i.e., the *macro search space*. Our proposed search space is inspired by the search space in ENAS.

The choice of the search algorithm has also been explored in NAS, where some works leverage RL while others resort to Evolutionary Algorithms (EAs). Outside the realm of NAS, [26] proposed Monte Carlo Tree Search (MCTS) which extends the well-known Multi-armed Bandit technique in RL to tree-structured search spaces. This inspired an interesting approach in [58] for improving the controller by using MCTS to find the best architecture hyper-parameters. Using MCTS with UCB is best known to balance the exploitation and exploration in the searching process to overcome possible sub-optimal solutions. The main idea is to use MCTS to find the model's hyper-parameters in a layer-by-layer fashion in the child model descriptions. Selection, Expansion, Playout with simulation, and Backpropagation are the main steps of MCTS. To estimate the search directions in MCTS, the child networks should be trained multiple times. The authors suggest using a simulation network to estimate the child network's accuracy to only train each child network once on the dataset as opposed to multiple times. The model's accuracy is then estimated by aggregating the training and simulation results.

EAs are an alternate to RL for searching the neural architecture search space [40, 39]. Authors in [40] evolve an initial *population* of strings representing neural architectures using a *tournament selection* algorithm, where after each pairwise comparison the worse individual dies and the better one mutates. The fitness of each string is determined by the respective architecture's validation accuracy after being trained on a dataset. More closely to [38], authors in [39] use an EA to search the *NASNet* search space. The authors use a tournament selection algorithm similar to [40] and introduce the concept of *aging* to individuals. Comparing their algorithm to the RL baseline, the authors show that their EA reaches higher accuracy faster than the RL-based method, however, methods converge to the same accuracy asymptote. The authors argue that the EA-based method may be of higher importance in larger search spaces where reaching the optimum can require more resources than available. We leverage and compare the EA algorithm in [39] to other search algorithms for our search space in Chapter 7.

2.6 Neural Architecture Search for Traffic Classification

The automatic generation of a network traffic classifier has been explored in a few works in the literature. Authors in [22] propose an AutoML framework where an ensemble classifier is automatically generated on a collected dataset for malware detection. The ensemble is created by picking the three best performing models out of 7 possible classical machine learning (ML) models. The models are then stacked, with each model repeated at most 10 times in the stack. Only one of the possible models is a neural network, which performs the worst on the given dataset. They make use of an open-source python package for automatically tuning the hyper-parameters of each model. Their contribution lies in the automatic creation and parameter tuning of an encrypted traffic classifier. Their work differs from ours in that they examine AutoML for classical ML algorithms. The number of hyper-parameters to be tuned in a model determines the number of variables of the AutoML problem and thus its complexity. Therefore, the space of possible solutions that an AutoML algorithm for classical ML models needs to search is orders of magnitude smaller than that of a AutoML algorithm. Hence, approaches such as NAS both restrict the search space and make use of RL or EA-based algorithms to search the space. In short, AutoML for deep networks is a fundamentally different problem from AutoML for classical ML models.

The closest works to ours are [60, 33]. Authors in [60] designed a search space and used several EA strategies including multi-objective swarm optimization to perform NAS for the IDS2012 and ISCX VPN datasets [30]. Their search space consists of 4 types of CNNs connected via add and concatenate operations. Their generated classifiers operate on the first 160 bytes of payload from the first 10 packets of each flow (i.e., 1600 bytes in total). Their approach achieves above 99% precision and recall on both datasets, while a simple baseline k-NN is within 1 to 2% of the NAS-generated model. However, in our preliminary experiments, the performance for k-NN on our datasets was inferior. Classifying the mixed-protocol ISCX VPN dataset has been shown to be an easier classification task than classifying a full encrypted single-protocol TLS or QUIC dataset [4], similar to the datasets employed in this thesis. This may have contributed to the differences in challenges encountered and higher performance in [60] versus our work.

Authors in [33] proposed an approach for a network intrusion detection task over three datasets. Their search space consists of layers of stacked cells, with two types of cells called normal and reduction cells, where reduction cells halve the size of the input feature map. Each cell is a DAG of 5 connected nodes, with each node containing one of 12 operations. The operations are different convolution or pooling operations. For the search strategy, they use and compare three different multi-objective EAs. To increase the efficiency of their approach, the authors train several classical ML models to predict the performance of the generated models in each iteration of the EA. The models used for training are the initial population plus the selected best models added at the end of each round. Their generated model, NAS-NET, is evaluated at near 100% F1-score on all three datasets, with baseline models being within 1-2% range of NAS-NET in F1-score. Out of the three datasets used for evaluation, one is HTTP-based and unencrypted, whereas the other two are mixed-protocol CIC-DDoS2019 [50] and ISCXIDS2012 [52] datasets.

Contrary to [60, 33], our work focuses on fully encrypted datasets of TLS and QUIC traffic. We target the traffic classification task, as opposed to [33] which targets the intrusion detection task. In contrast to [60], we evaluate our work on datasets that were collected from 2019 onwards, and thus are more recent than the ISCX VPN dataset collected in 2016. We argue that our TLS dataset presents a more difficult classification task than the datasets presented in [60, 33], given the near 100% performance of all ML models, including classical ones, on their datasets. This is far from what we observed on our TLS dataset. Results on the publicly available QUIC dataset are presented to show the generalizability of our approach to other (non-proprietary) datasets.

Chapter 3

Methods & Models for the Manual Approach

3.1 Deep Learning models

3.1.1 UW Tripartite Model

The University of Waterloo Tripartite model (UW) is a DL model proposed in our previous work [4]. The UW model achieves an accuracy of over 90% on purely encrypted TLS network traffic. It is a three-part model, as depicted in Figure 3.1a, where each part is designed to operate on a different type of input data. Note that the orange and yellow boxes in the figure depict convolution and max-pooling layer kernels, respectively. Furthermore, each layer's output vector is depicted by a white box accompanied by its size.

Firstly, the model consists of a series of CNNs operating on header bytes from the first three packets of the TLS handshake. CNNs are useful for extracting shift-invariant information which makes them suitable for header bytes. Secondly, the model contains a series of LSTM layers operating on flow time-series data, which includes a three-dimensional array of packet sizes, packet directions, and packet inter-arrival times for each flow. LSTMs are renowned for relating useful information in a time-series data. The output of the LSTMs passes through a dropout layer before being concatenated to other parts' outputs. Lastly, a series of dense layers in the model is designed to work on statistical flow data, which includes 77 features. The statistical features are called auxiliary features in this thesis. Our experiments suggest that the auxiliary features have the least effect on the model's

Table 3.1: Service-level datasets properties

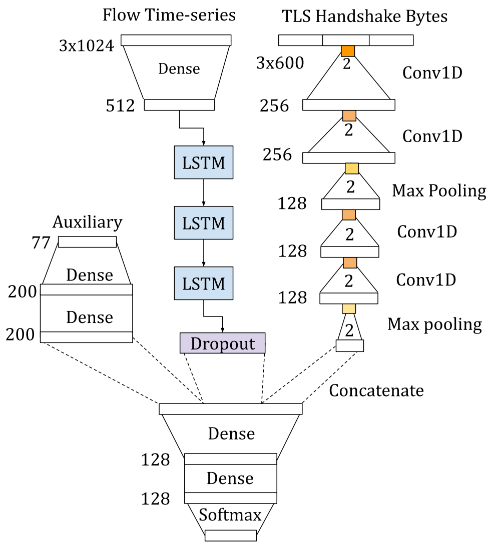
Protocol	Dataset	Total flows (K)	Labeled flows (K)	Labeled flows (%)
TLS	07-2019	762.7	119.8	15.7
	09-2020	411.7	89.9	21.8
	04-2021	284.8	42.3	14.8
	05-2021	124.0	17.5	14.1
	06-2021	261.2	51.2	19.6
QUIC	QUIC-05-2021	37.8	26.0	68.0

performance. The outputs of the three parts are then concatenated and passed through two dense layers and a softmax layer to obtain the final classification.

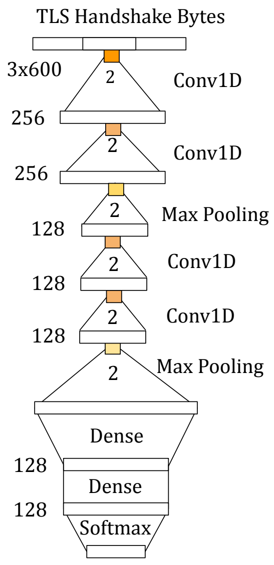
To the best of our knowledge, the UW model obtains the highest accuracy to date on a fully encrypted dataset, for service-level classification. In this thesis, we perform an ablation study on the different parts of the UW model. The decomposed parts of UW, i.e., for TLS header bytes (UW-H), flow time-series information (UW-F), and auxiliary features (UW-A), are depicted in Figure 3.1b, Figure 3.2a, and Figure 3.2b, respectively.

3.1.2 UCDavis CNN Model

The authors in [42] propose a CNN model for early classification of network traffic flows. Their CNN model operates on the first six packets of a flow, for each of which, the first 256 raw bytes from L3 and above are extracted and concatenated together to form the input feature vector. The model consists of convolutional, max-pooling and dense layers as shown in Figure 3.3. We leverage the UCDavis CNN model in this thesis, as it was shown in [4] that after the UW model, the UCDavis CNN obtains the best accuracy on their fully encrypted dataset among a number of evaluated models. This model was designed for performing on packet raw bytes. Therefore, it's only used in the comparisons performed on the packet raw bytes (e.g., not on the flow time-series information).

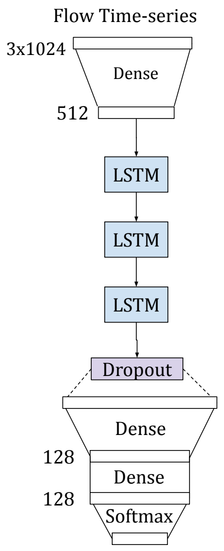


(a) UW model architecture [4]

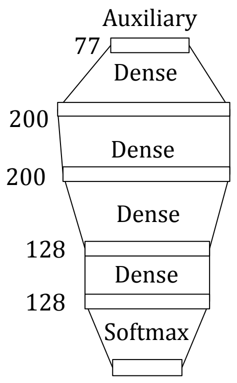


(b) UW-H, i.e., decomposed TLS header part of UW

Figure 3.1: UW Model and decomposed parts



(a) UW-F, i.e., decomp. flow time-series part of UW



(b) UW-A, i.e., decomposed statistical part of UW

Figure 3.2: UW Decomposed parts (cont.)

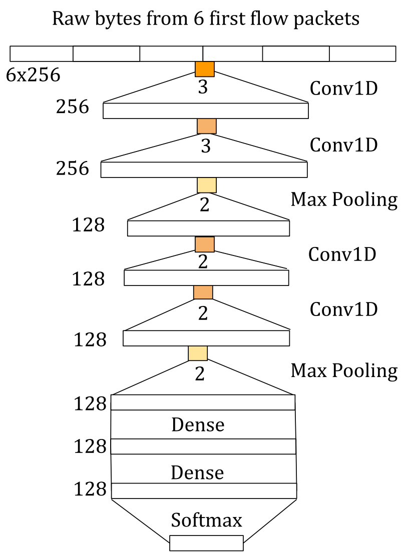


Figure 3.3: UCDavis CNN architecture [42]

3.2 Datasets description

We use a total of six datasets in this thesis which consist of TLS and QUIC traffic traces collected from a major ISP's mobile network. The source and destination IP addresses are obfuscated and the packets are truncated after 400 bytes, except for the TLS handshake packets. A flow is assumed to be a quintuple of source IP, destination IP, source port, destination port and protocol.

Preprocessing and labeling modules are used to turn the packet captures into labeled datasets of traffic flows. Both modules are implemented as in [4]. The preprocessed data includes raw TLS header bytes from the flows, as well as flow time-series information consisting of an array of packet sizes, packet inter-arrival times, and packet directions for each flow. Moreover, it consists of 77 auxiliary features for each flow, extracted using CICFlowMeter [30]. The auxiliary features include statistical information about flows, e.g., mean, median, minimum, and maximum of packet sizes in each direction.

The labeling module is used to label the flows according to the Server Name Indication (SNI) field. The flows are labeled at two levels of granularity: (i) service-level, and (ii) application-level. Service-level labels consist of 8 classes each representing a service category, namely, *chat*, *download*, *games*, *mail*, *search*, *social*, *streaming*, and *web*. For each service-level, there is a corresponding set of applications. For example, the *mail* class

Table 3.2: Application-level datasets properties

Protocol	Dataset	Total flows (K)	Labeled flows (K)	Labeled flows (%)
TLS	07-2019	762.7	83.1	10.9
	09-2020	411.7	59.8	14.52
	04-2021	284.8	26.3	9.2
	05-2021	124.0	11.1	9.0
	06-2021	261.2	34.6	13.2
QUIC	QUIC-05-2021	37.8	9.3	24.6

Table 3.3: Service-level and corresponding application-level classes for TLS datasets

Service-level class	Application-level classes			
chat	Facebook	Snapchat	Whatsapp	-
download	Apple	GooglePlay	-	-
mail	Gmail	Hotmail	Outlook	-
search	Google	-	-	-
social	Facebook	Instagram	Twitter	-
streaming	Facebook	Netflix	Snapchat	Youtube
web	Amazon	AppleLocalization	Microsoft	-
games	-	-	-	-

consists of *mailGmail*, *mailHotmail*, and *mailOutlook* applications. There are a total of 19 applications, which act as a finer level of labeling per service class. Note that not all the applications in a service class have enough flows to be categorized as an application class. Therefore, the number of labeled flows in the application-level is smaller than service-level. The service-level classes and corresponding applications are presented in Table 3.3. The *games* service class does not have corresponding application classes, as it consists of many applications with a very small number of flows. Nevertheless, these applications' flows together form the *games* class at the service-level.

The employed datasets can be categorized into two types based on encryption protocol, i.e., *TLS* and *QUIC*.

(i) TLS datasets: We leverage five datasets encrypted with the TLS protocol, each containing one to two hours of packet traces. The datasets are captured chronologically and named in the MM-YYYY format, i.e., 07-2019, 09-2020, 04-2021, 05-2021, and 06-2021,

respectively.

(ii) QUIC dataset: The QUIC dataset, QUIC-05-2021, is extracted from a packet trace of QUIC traffic captured at the same time as the TLS 05-2021 dataset. The TLS handshake bytes are tightly coupled to the TLS protocol and thus irrelevant to QUIC. Therefore, the QUIC dataset only consists of flow time-series information. Auxiliary data was not added to this dataset as the effect of flow statistics on model performance was negligible in our experiments. The QUIC dataset is used to show that our architecture adaptation best practices, which are centered around UW-F model, generalizes to non-TLS encrypted data (cf. Section 5).

Tables 3.1 and 3.2 show the total number of flows, labeled flows, and the percentage of labeled flows, for service-level and application-level datasets, respectively. The number of labeled flows depict the size of each dataset. The percentage of labeled flows in each dataset highlights the performance of the employed labeling module for the TLS flows. Evidently, the percentage of the labeled flows across the datasets are more or less inline with each other, asserting the suitability of the labeling module. Additionally, the labeled distribution of TLS flows for service and application classes are depicted in Figure 3.4 and Figure 3.5, respectively. There is insignificant difference in class distribution across the datasets. Therefore, we use the accuracy as the primary performance metric for evaluating the models across the datasets. To deal with class imbalance we adopt a weighting strategy, i.e., we up-sample classes with smaller number of flows.

There are several interesting takeaways when we compare the distribution of application classes. Notably, downloadApple has the highest number of flows. Furthermore, streamingNetflix has the lowest number of flows among all applications. This implies that although Netflix is an extremely popular application, not many users watch Netflix on their mobile devices when compared to Snapchat, Facebook, and Youtube. Note that the datasets were captured on the ISP's mobile network.

For labeling the QUIC dataset, we change the classes in the TLS dataset. Since QUIC is still not widely adopted by services across the web, not all classes from the TLS dataset have enough flows in the QUIC dataset. For instance, QUIC is known for enhanced security and faster connections, which makes it more suitable for time-sensitive applications, e.g., streaming services. Therefore, it makes sense that we did not see any flows labeled as the *download* class in the QUIC dataset. Hence, we keep the *games*, *social*, *streaming*, and *web* classes, while adding some new classes, i.e., *e-commerce* and *resources*. The *resources* class corresponds to the flows that are essentially shared among different websites that mostly deliver tools, such as JavaScript APIs or design content for websites. The new labeling module can label up to 68% of the flows, a large improvement over the less than

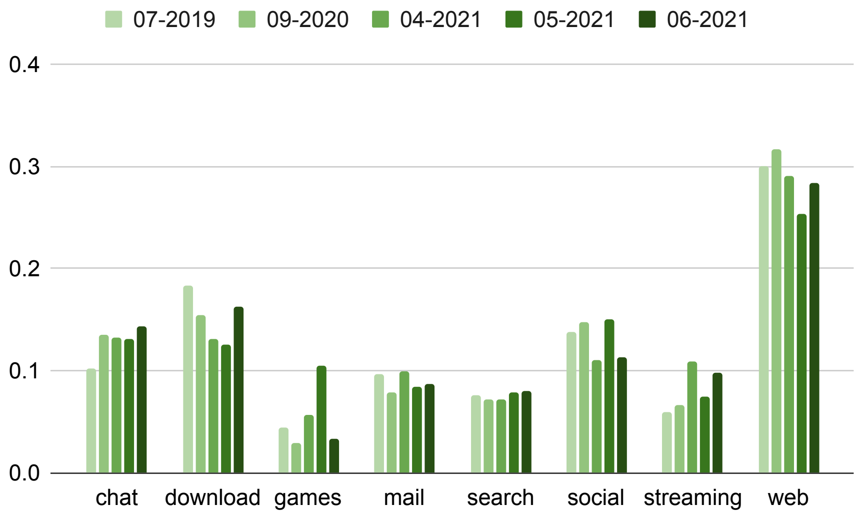


Figure 3.4: Service-level class distribution of the TLS datasets

20% labeling performance on the TLS datasets. We attribute this to fewer services using QUIC and most of them corresponding to the *resources* class. Therefore, the SNIs are not as varied in this dataset as they are in the TLS datasets.

3.3 Software stack and performance metrics

The software stack for data pre-processing, model training, and evaluation includes Tensorflow with Keras API, CUDA, PySpark, SCAPY, and TShark.[1] The use of these well-established technologies ensure that the code base is horizontally scalable and resilient. Training was conducted on 80% of each dataset, while the remaining 20% was used for validation. A multi-class classification problem can be seen as a set of many binary classification problems, one for each class. Each binary classification task may result into True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). The performance of each binary classifier can be measured in terms of:

$$Precision = \frac{TP}{TP + FP} \times 100$$

[1]The code repository: shorturl.at/iuvV1 also contains the comprehensive technical information on how the library should be used and how its environment should be configured on Linux platforms.

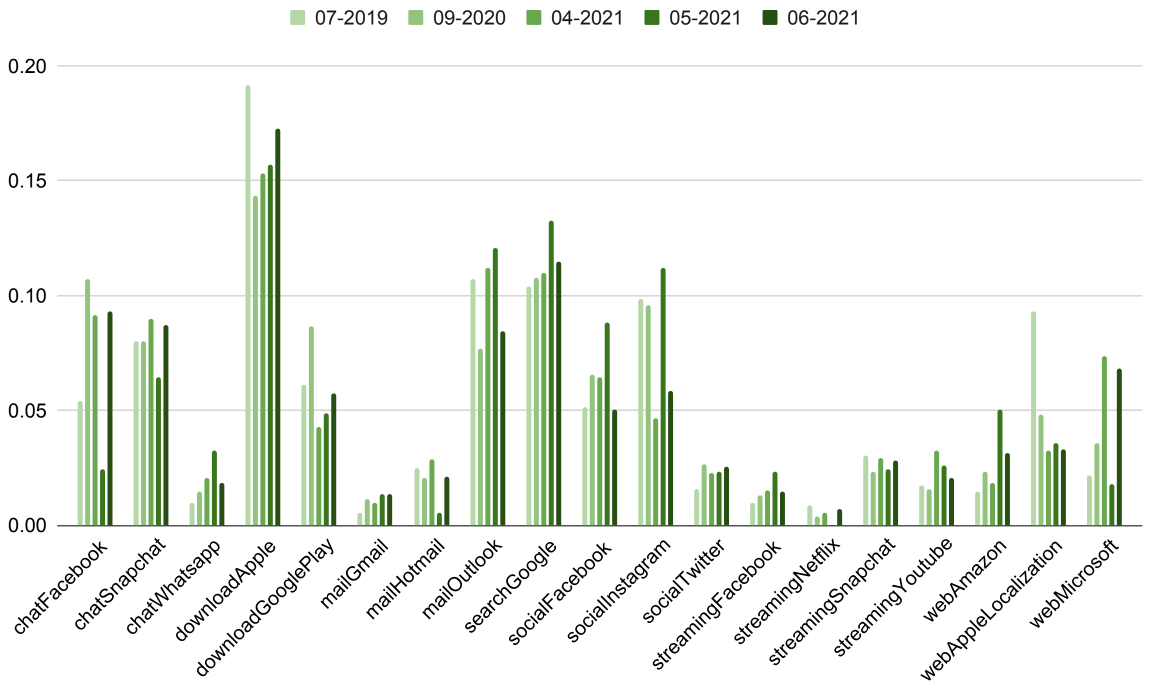


Figure 3.5: Application-level class distribution of the TLS datasets

$$Recall = \frac{TP}{TP + FN} \times 100,$$
$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100,$$
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100.$$

In this thesis, we measure the performance of the multi-class classifiers in terms of the most standard metrics in this domain such as accuracy, and weighted average F1-score, recall, and precision, where weighted average is the average of the corresponding metric across all classes weighted by the number of data points that we could label for each class.

Another metric that we use is top-k accuracy. This metric measures how often the model is able to predict the right class in the first k guesses. For instance, top-1 accuracy corresponds to the accuracy metric defined above. The logarithmic weighted mean of top-k accuracies is the weighted average of top-k accuracies, where the weights logarithmically decrease as k increases.

Chapter 4

Investigating Data Drift

In this chapter, we study the performance of UW model when trained on a baseline TLS dataset and tested on a different, target TLS dataset. We investigate model decay in time by leveraging the decomposed models and experimenting with different TLS datasets.

4.1 Baseline performance

We start by highlighting the performance of the UW model on the 07-2019 dataset, which is the oldest and biggest TLS dataset. To provide insight into the performance of each part of the UW model separately by conducting experiments on the decomposed models (i.e., UW-H, UW-F and UW-A). Table 4.1 shows the performance of these models in service-level classification.

We notice that, when we train the decomposed models on the 07-2019 dataset, the UW-H model shows the highest accuracy, which is 0.3% higher than the accuracy of the UW model on the same dataset. We attribute this to more data leakage in TLS headers at the time of the corresponding dataset collection (cf. Section 4.3). The UW-A model shows the lowest accuracy of 43.8%. With such a low accuracy, it is evident that the flow statistics are not helping, resulting in an even inferior performance to the UW-H model.

The results for application-level classification are depicted in Table 4.2. These results concur with the previous findings, with similar trends for UW and decomposed models with the exception of UW-H. The UW-H model achieves the highest accuracy of 96%, which is even better than the UW model, while the auxiliary input achieves the worst

Table 4.1: Model performance on the 07-2019 TLS dataset in service-level classification

Dataset	Accuracy (%)			
	UW	UW-H	UW-F	UW-A
07-2019	94.5	94.8	86.3	43.8

Table 4.2: Model performance on the 07-2019 TLS datasets in application-level classification

Dataset	Accuracy (%)			
	UW	UW-H	UW-F	UW-A
07-2019	95.7	96.0	84.2	29.1

performance (i.e., 29.1% accuracy). We will provide more details on the reasoning behind this phenomena for the UW-H in the latter chapters.

4.2 Robustness to performance decay

We study the performance of the UW model in service- level classification on different target (i.e., test) TLS datasets after training it on the baseline 07-2019 dataset. The target datasets, i.e., 09-2020, 04-2021, 05-2021, 06-2021, were collected at different points in time within two years from the 07-2019 dataset. As our experiments have shown that the performance of UW-A is inferior with little to no impact on UW model performance, we focus our study on the UW-H and UW-F models.

The results of the first set of experiments is shown in Figure 4.1. Evidently, the prediction ability of the model decays over time, which is quantified in Table 4.3. We see that the performance decay of the UW model is at its lowest on the 09-2020 dataset (i.e., 35.7%) and at its highest on the 06-2021 dataset (i.e., 41.1%). Note that the 07-2019 dataset and the 06-2021 dataset are about two years apart.

Model performance decay over time is an expected phenomenon. Nevertheless, we see that it does not have an equal impact on the UW-H and UW-F models. In fact, the performance of UW-H decays 7% more on average than the performance of UW-F

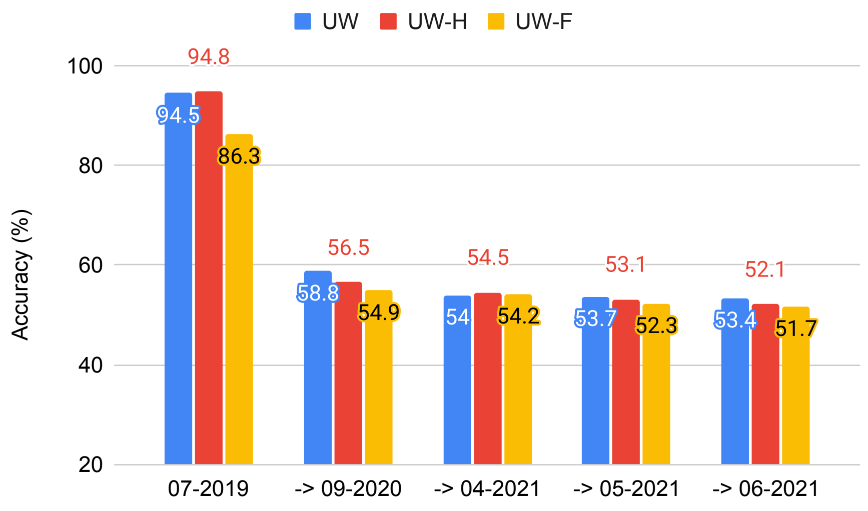


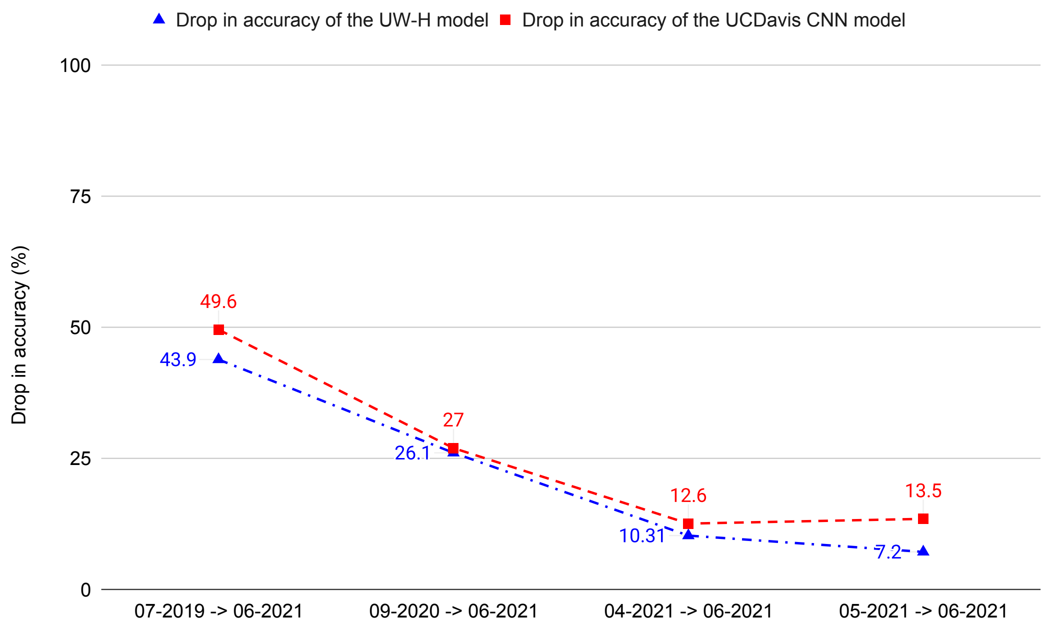
Figure 4.1: Model performance when trained on baseline 07-2019 dataset and tested (notation $\rightarrow$) on target datasets

(i.e., 40.75% compared to 33.02%). This suggests that using the traffic shape features, which is captured by the UW-F input, make the model more robust to decay over time. This also suggests that the TLS headers contribute more to the drop in accuracy over time for the UW model.

The previous experiments also highlight that performance decay correlates with the time difference between the training and target datasets. Therefore, we run experiments to further investigate this observation. In particular, we train UW-H using different datasets, i.e., 07-2019, 09-2019, 04-2021 and 05-2021, and measure how much the performance of

Table 4.3: Drop in model's predictive accuracy when trained on the baseline 07-2019 dataset and tested on subsequent datasets

Model	Target datasets				Avg. accuracy drop (%)
	09-2020	04-2021	05-2021	06-2021	
UW	35.7	40.5	40.8	41.1	39.52
UW-H	38.3	40.3	41.7	42.7	40.75
UW-F	31.4	32.1	34.0	34.6	33.02



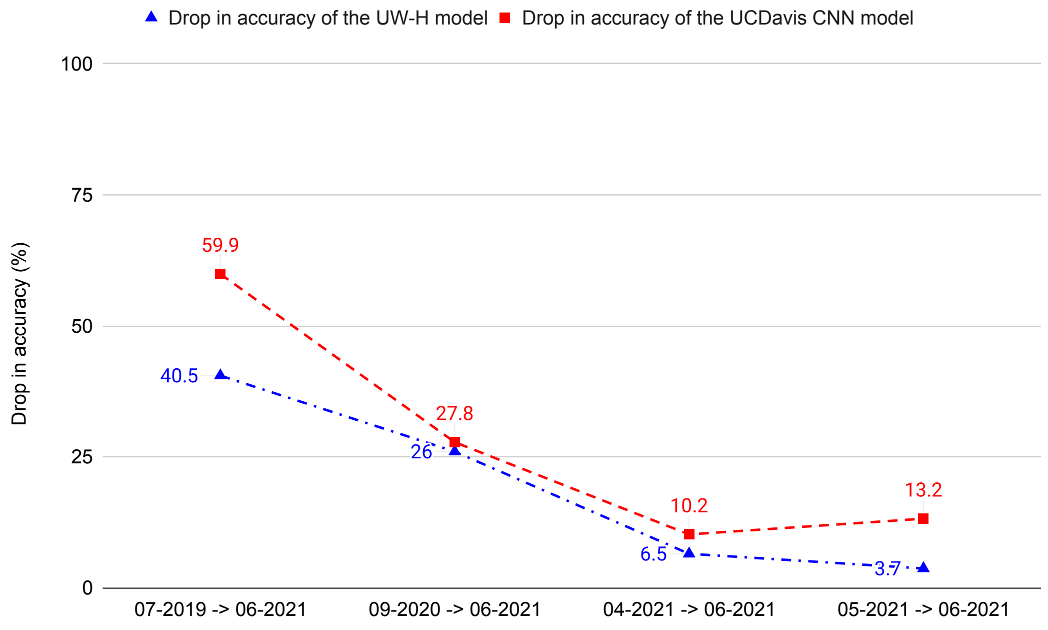


Figure 4.2: Performance decay of UW-H and UCDavis CNN in service-level classification *(up)* and application-level classification *(down)*

the trained model decays by 06-2021. We conduct the same set of experiments with the UCDavis CNN model and compare the performance of both models. Given that the size of training set, has an impact on the accuracy of a DL model, we down-sample the training datasets to the size of the smallest one (i.e., 05-2021 dataset) and average the results on all samples.

The performance decay in decreasing order of time span is shown in Figure 4.2. It is evident that the closer the datasets are in time of capture, the lower the performance decay of the UW-H model. For example, the accuracy of UW-H in service-level classification decays by 43.9%, 26.1%, 10.3%, and 7.2% roughly after 2 years, 1 year, 2 months, and 1 month, respectively. We attribute this to a discrepancy in data distribution between the training and target datasets, i.e., data drift, which we will investigate in the next subsection.

The same trend can be seen for the UCDavis CNN model up to 04-2021, although the performance decay is even more noticeable than on UW-H. For instance, when the training and target datasets are 2 years apart, the accuracy of the UCDavis CNN model decays by 49.6%, compared to 43.9% for UW-H in service-level classification. Two aspects of the UW-H model could be contributing to its comparatively higher robustness to data drift: (i) more regularization layers, which prevents the model from overfitting to the training dataset, and (ii) feature engineering, in which the TLS handshake header bytes are used as input as opposed to any header bytes, reducing the noise in the model's input. We note that the performance decay of the UCDavis CNN model in service-level classification is lower in the span of 2 months (i.e., between 04-2021 and 06-2021) compared to the span of 1 month (i.e., between 05-2021 and 06-2021), hence breaking the previous trend. This suggests that the model simply overfits the training dataset rather than naturally decay as data drifts over time. Furthermore, the UCDavis CNN model seems be less generalizable than the UW-H model.

Figure 4.2 also shows that the accuracy of UW-H in application-level classification decays by 40.5%, 26%, 6.5%, and 3.7% over the span of 2 years, 1 year, 2 months, and 1 month respectively, similar to service-level classification. The performance decay of the UCDavis CNN model in application-level classification also follows the same trend as in service-level classification. Specifically, the drop in accuracy decreases from 59.9% over the span of 2 years, to 27.8% over the span of 1 year, to 10.2% over the span of 2 months, and increases again to reach 13.2% when the datasets are 1 month apart. Indeed, the decay is much worse with the UCDavis CNN model than UW-H in application-level classification. This suggests that the UCDavis CNN model is even more susceptible to data drift and overfits to the training datasets in application-level classification.

Table 4.4: Per-service class accuracy of UW-H and UCDavis CNN on (a) the 04-2021 dataset, and (b) the 07-2019 dataset

(a)

Class	Accuracy (%)	
	UW-H	UCDavis CNN
chat	77	84
download	86	82
games	95	82
mail	83	89
search	87	83
social	82	82
streaming	88	82
web	86	79

(b)

Class	Accuracy (%)	
	UW-H	UCDavis CNN
chat	96	92
download	95	89
games	97	88
mail	97	95
search	99	95
social	96	93
streaming	93	82
web	92	91

In the following, we investigate which service classes are most affected by data drift. Furthermore, we investigate which service is confused with, as time passes. Finally, we investigate whether this confusion holds across different architectures or not. We strategically focus our study on two particular scenarios. The first is when the training and testing datasets are 2 years apart (i.e., the model is trained on the 07-2019 dataset and used to classify the 06-2021 dataset). That is when the datasets are the furthest apart and the effect of data drift is most noticeable on the overall performance of UW-H as well as UCDavis CNN, both in service-level and application-level classification. The second is when the datasets are only 2 months apart (i.e., the model is trained on the 04-2021 dataset and used to classify the 06-2021 dataset). This is when data drift affects the performance of UCDavis CNN the least. Tables 4.4 *(a)* and *(b)* present the baseline per-class accuracies, i.e., when the target dataset is the same as the training dataset

Figure 4.3 depicts the confusion matrix of UW-H in service-level classification, in each of the above scenarios. When the training and test datasets are 2 years apart, *streaming* and *download* are the two services UW-H misclassifies the most, achieving 22% and 16% accuracy on these classes, respectively, and hence a drop of roughly 70% and 80% in classification accuracy. In particular, the model misclassifies 53% of the *streaming* flows and 50% of the *download* flows as *web* flows. We note that *web* is the class most of the misclassified flows are confused with, e.g., 53% of the *streaming* flows, 50% of the *download* flows, 25% of the *games* flows, and 25% of the *mail* flows. *streaming* is the second most confused with, e.g., 27% of the *games* flows, 23% of the *chat* flows, and 17% of the *download* flows. We can associate the model's tendency to misclassify flows as *web* to the higher percentage of *web* flows in the training set, which creates a bias for this class. Therefore, *web* is the *default* label the model selects for a flow when its confidence in the true label is low.

When the training and target datasets are 2 months apart, however, we see a drastic increase in the model's ability to correctly classify all flows in general, in particular the *streaming* and *download* flows, i.e., 65% and 75% accuracy respectively. Some but fewer flows remain misclassified as *web* flows, e.g., 22% of the *games* flows, 16% of the *streaming* flows, and 10% of the *download* flows, compared to the previous 25%, 53%, and 50%, respectively. However, the model seems also to have a bias for the *games* class, as it now confuses more flows with *games* flows, e.g., 21% from the *chat* class, 15% from *search*, and 13% from *download*. In general, the *games* and *web* are the classes the model is most confused with and about, whether the training and the target datasets are close in time or far apart.

Figure 4.4 presents the confusion matrices of the UCDavis CNN model. Evidently, when the training and target datasets are two years apart, the UCDavis CNN model has

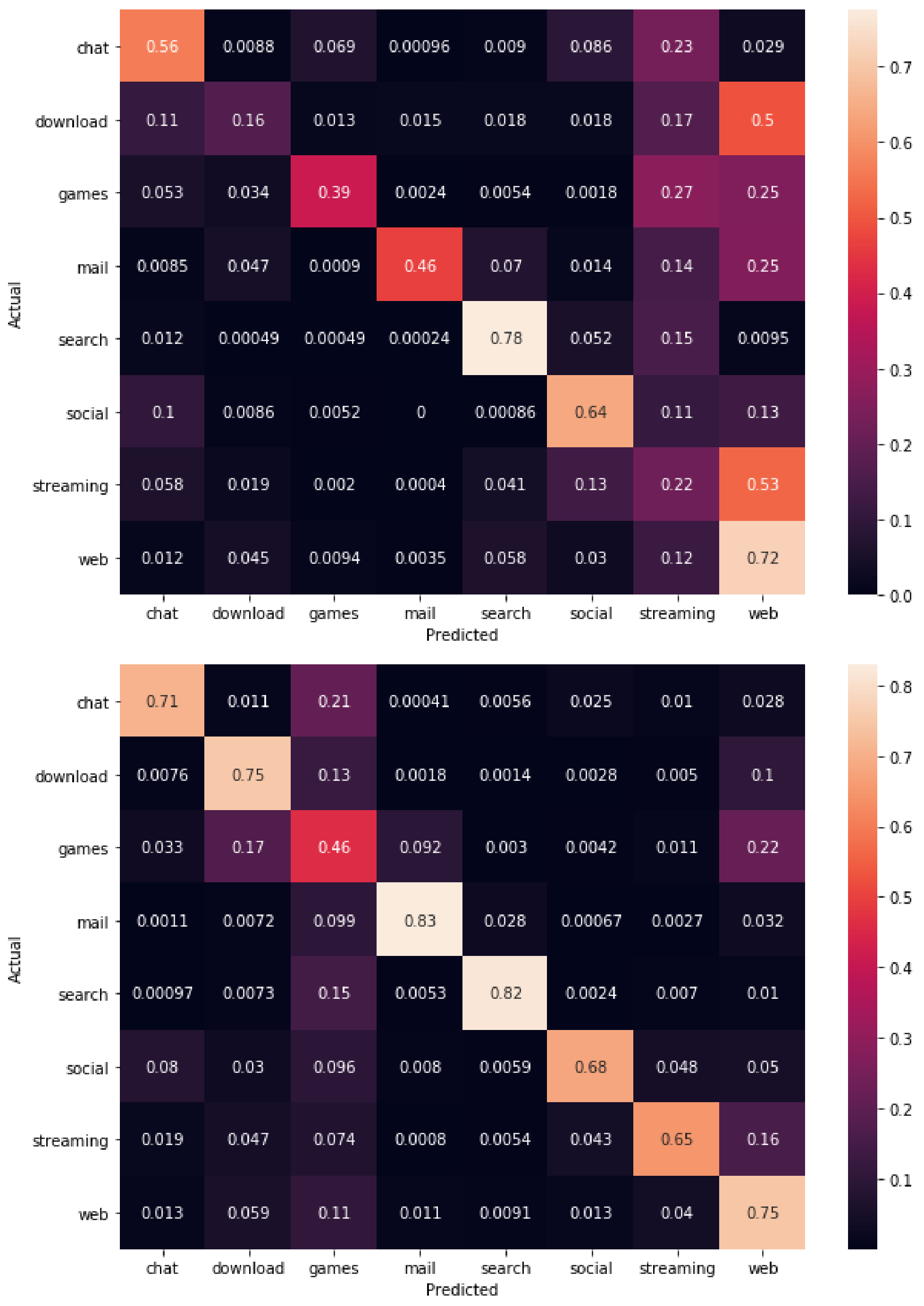
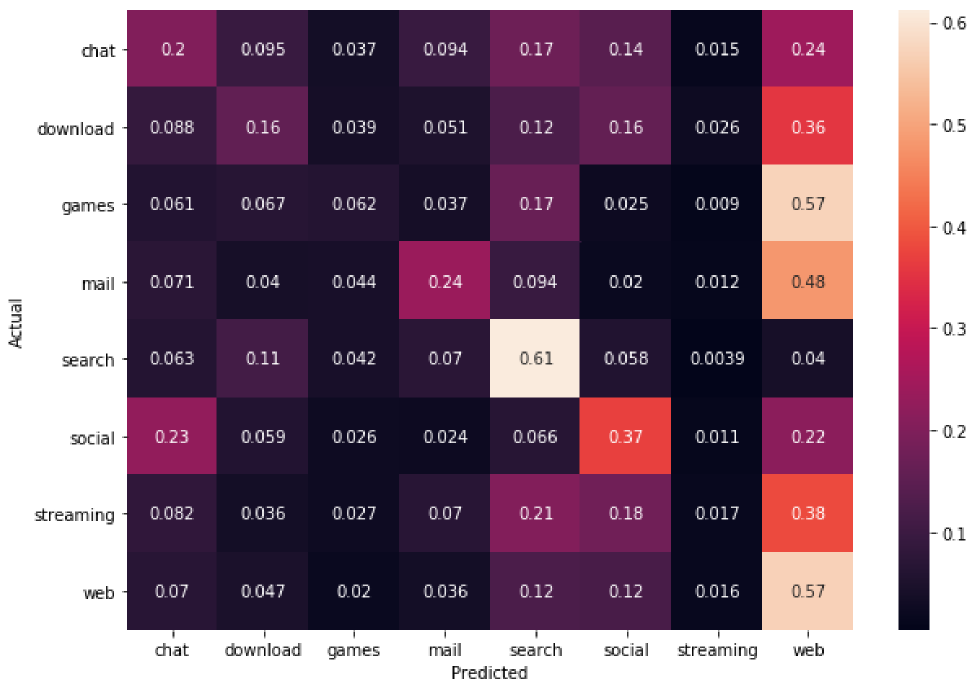


Figure 4.3: Confusion matrix of UW-H when training and target datasets are two years (*up*) and two months (*down*) apart



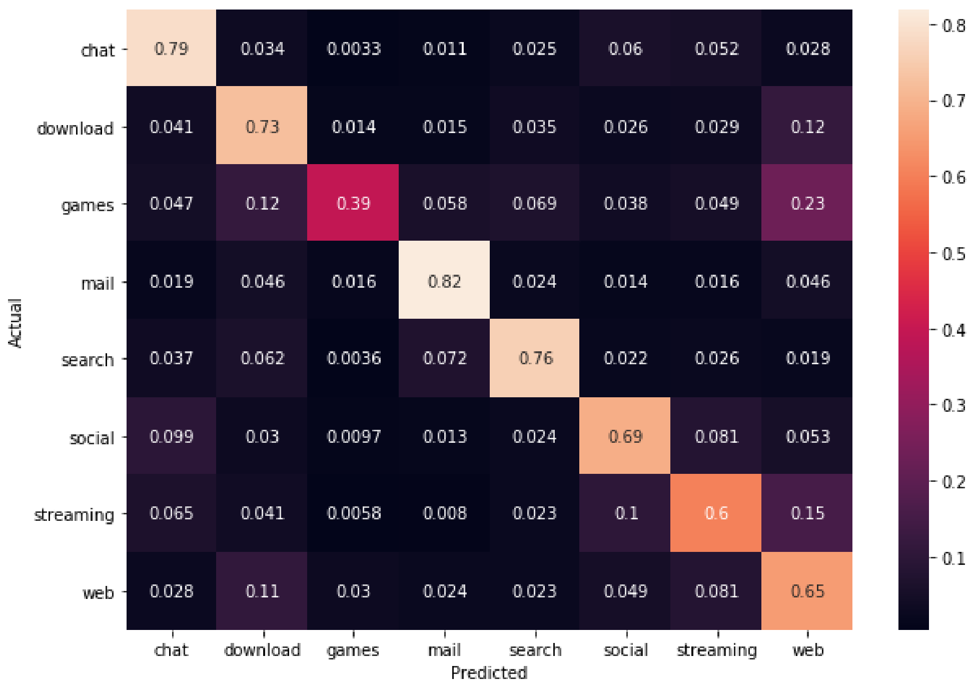


Figure 4.4: Confusion matrix of the UCDavis CNN model when training and target datasets are two years (*up*) and two months (*down*) apart

a much higher tendency to misclassify flows than UW-H. While *streaming* and *games* are the two classes with highest misclassification rates, i.e., the accuracy of the classifier not exceeding 1.7% and 6.2% respectively on these classes thus experiencing a drop of over 90% in per-class accuracy, the UCDavis CNN also misclassifies over 75% of the *chat*, *download*, and *mail* flows, confusing these with *web* traffic most of the time. Similarly to UW-H but at a larger extent, *web* is the class UCDavis CNN most confuses other classes with. UCDavis CNN also seems to be confused about more classes than UW-H. For instance, in addition to the *web*, UCDavis CNN equally misclassifies flows as *search* or *social*, e.g., 17% of *chat* flows are misclassified as *search* and 14% as *social*, 12% of *download* flows are misclassified as *search* and 16% as *social*, 21% of *streaming* flows are misclassified as *search* and 18% as *social*, and 12% of *web* flows are misclassified as *search* and 12% as *social*. We attribute this higher misclassification and confusion rate of the UCDavis CNN model to the noisy features (i.e., encrypted data) that are input to the model, as discussed earlier. We note that, the misclassification and confusion rates drop significantly when the datasets are 2 months apart, and UCDavis CNN exhibits similar behaviour to UW-H.

In the following, we attempt to study the extent to which classes are affected by data drift in traffic, leveraging the top-k accuracy measure. Some traffic classes may be impacted by data drift more than others such that it would take the model several more guesses to, eventually, classify them correctly.

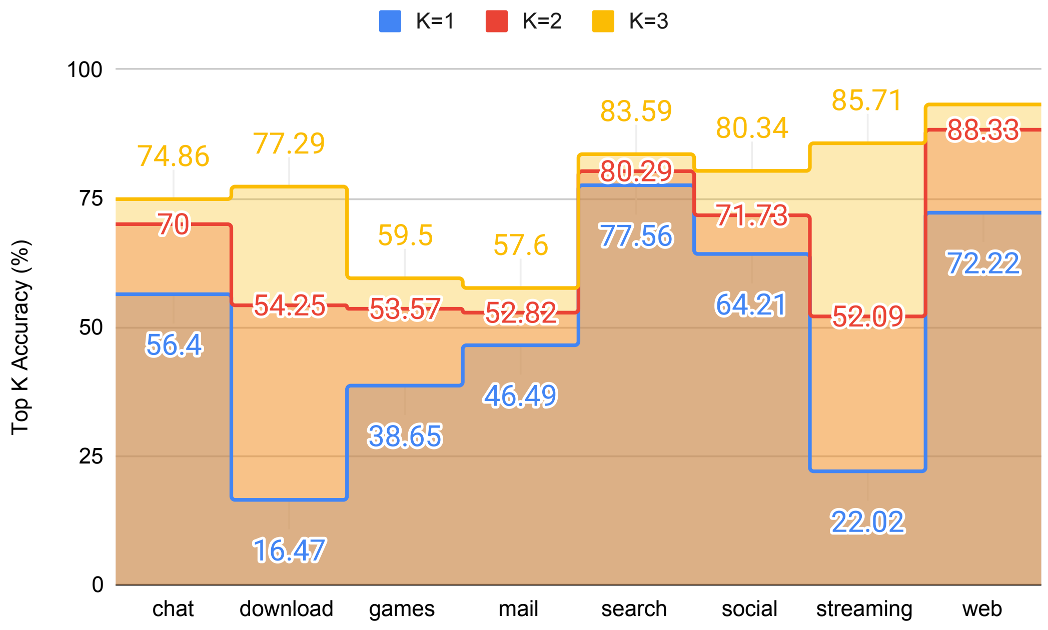
Figure 4.5, presents the top-k accuracy of UW-H when the target and training datasets are 2 years and 2 months apart, for k=1, 2 and 3. Evidently, considering the model's top-2 or top-3 guesses significantly increases the model's accuracy on particular classes, thus increasing the model's overall accuracy. For example, when the target and training datasets are 2 years apart, the accuracy of the model on the *download* class increases from 16.5% to 54.2% and 77.3% with k=2 and 3, respectively. On the contrary, the accuracy of the model on *mail* does not increase much when k is increased to 2 or 3, i.e., after the second guess UW-H still misclassifies 47% of *mail* flows and 42% after 3 guesses. This suggest that *mail* traffic data has shifted so much since 07-2019 (recall that the baseline accuracy of UW-H in classifying 07-2018 *mail* traffic is 97%) that it would take more than 3 guesses for the model to correctly classify 06-2021 *mail* flows. Indeed, we can see that the *download* and *mail* traffic drifted relatively less in the span of 2 months, as the model achieves relatively higher classification accuracy on these two classes, which also increases with k= 2 or 3. These findings are also inline with conclusions drawn earlier from the confusion matrices.

In Figure 4.6, we summarize the top-k accuracy plots by averaging the top-k accuracies (for k=1, 2, and 3) using a logarithmic weighted mean function. Logarithmically decreasing weights are applied to the top-k accuracies with increasing k, giving higher weights to top-1 accuracies. The goal is to study and compare, in a simplified way, the data drift in traffic data across service classes after 2 years versus 2 months.

We notice that *download* and *streaming* are the classes with the lowest average top-k accuracy when the training and test target sets are 2 years apart. Not only are they the most impacted by data drift but also this is where there is the most obvious correlation between data drift and time span between the training dataset and target dataset. The *games* class is also highly affected by the data drift due to the 2-year-time span between the training dataset and the 06-2021 datatset. However, interestingly, it is equally highly impacted by the 2-month-time span, experiencing roughly 50% drop in classification accuracy in both scenarios. On the contrary, *search*, *social*, and *web* are equally much less affected by data drift regardless of the time span between the training and test datasets; the average top-k accuracies being equally relatively high in both scenarios. Interestingly, the *games* class is the most diverse among all application classes. Several different SNIs are matched to the *games* class, and the class does not seem to be dominated by some major games. Thus, for the drop in accuracy to be this noticeable, it seems like online games and underlying protocols are in constant shift.

While we uncovered what traffic classes are most impacted by the data drift in network traffic data, at the service-level, it is worth investigating which applications within service classes are most susceptible to data drift. We conduct the 2-year-time span experiment at the application-level, and measure the impact of data drift across application classes using the logarithmic weighted mean top-k metric, as reported in Figure 4.7.

Figure 4.7 depicts interesting findings. For instance, in the *chat* service class, the highest drop in accuracy is experienced by the *WhatsApp* application. In fact, the model fails to correctly classify all 06-2021 *WhatsApp* flows. Plus top-2 and top-3 classifications fail to boost the accuracy on this particular class. In the *download* service class, we can see that *GooglePlay* is more affected by data drift with a lower logarithmic weighted mean top-k accuracy compared to the *Apple* applications. In the *mail* service class *Gmail* is drastically affected by data drift, much more than the other mailing applications. More interestingly, in the *social* class, the biggest impact is experienced by the *Twitter* application, which makes *Facebook* and *Instagram* traffic seem more stable. Furthermore, almost all applications in the *streaming* service class experience roughly the same drop in accuracy and are affected almost equally by the data drift. Finally, for the *web* service class, the most stable traffic seem to belong to the *Microsoft* applications and the most affected traffic by the data drift is the one for the *AppleLocalization* application. The applications that are more affected by data drift (e.g., Whatsapp, Twitter, etc.) seem to be the most popular ones within their respective service classes. For instance, *Whatsapp* had more daily active users in France (i.e., the country of dataset collection) compared to *Facebook* according to



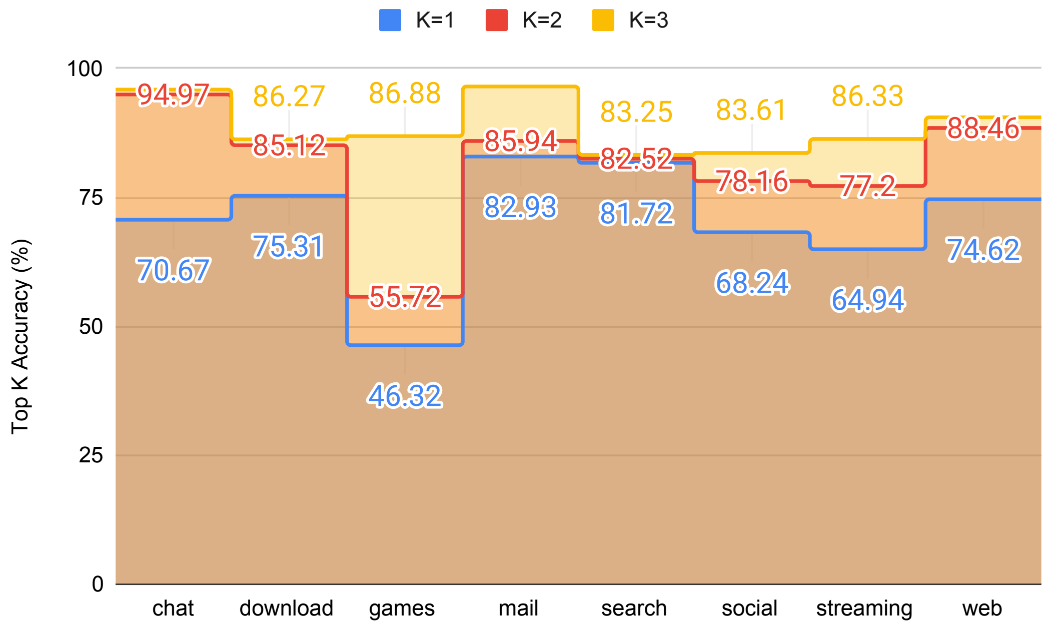


Figure 4.5: Top-k accuracy for the UW-H when training and target datasets are two years (*up*) and two months (*down*) apart

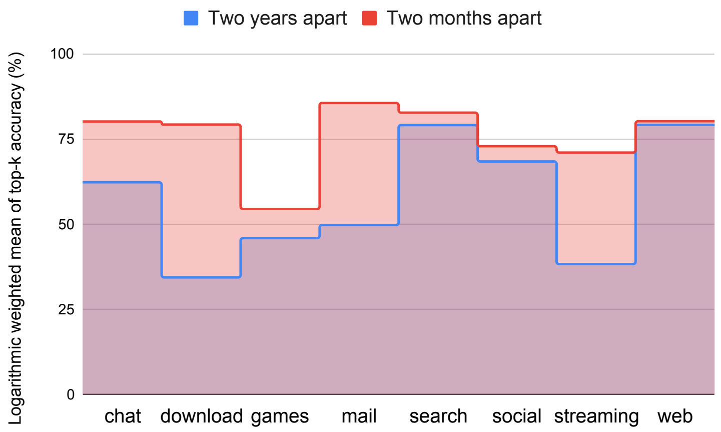


Figure 4.6: Logarithmic weighted mean of top-k (k = 1, 2, 3) accuracy with the service-level classes

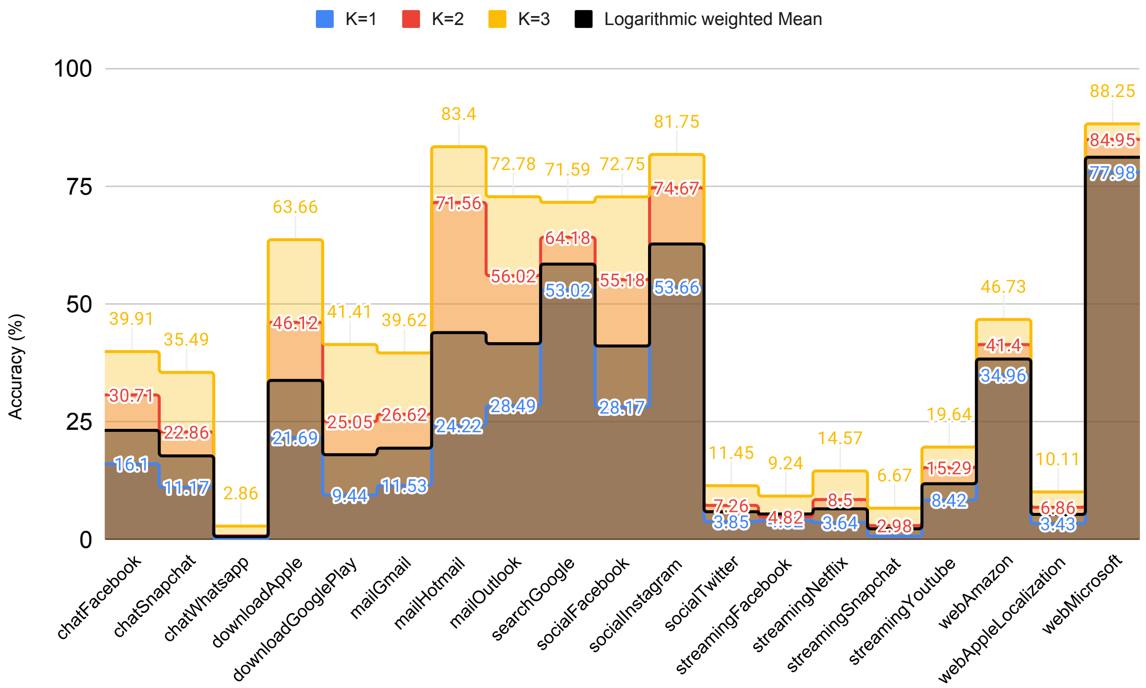


Figure 4.7: Per-application class top-k accuracy of the UW-H model when training and target datasets are two years apart

Table 4.5: Adoption of HTTP/2 and SPDY protocols over time

Protocol	Time		
	2018-2019	2019-2020	2019-2021
HTTP/2	+ 40.6%	+ 31.0%	+ 53.5%
SPDY	- 93.4%	- 66.6%	- 83.3%

July 2021 statistics [37].

4.3 Traffic data drift

The considerable drop in both model's performances when they are trained on 07-2019 and tested on 2021 datasets, as well as the fact that the performance drop correlates with the difference in time of capture between datasets, indicates that the data distributions the models are learning may be changing over time, thus making the learned patterns obsolete. To investigate this, we take a closer look at the L5 protocol distribution in the datasets, primarily looking for any time-related changes that we could identify. Note that since the data is encrypted, only some application-layer protocols are identifiable. Furthermore, as we filter on the application-layer protocols, we only perform the investigation for service-level classification.

Table 4.5 shows the adoption of HTTP/2 and SPDY protocols [57]. From 2018 to 2021, the adoption of HTTP/2 increased while the usage of SPDY, which is the predecessor to HTTP/2, drastically decreased. From 2019 to 2021, we can see that there is a 83.3% decrease in the usage of SPDY and a 53.5% rise in the adoption of HTTP/2. This may in part explain the drift in the datasets and, in particular, the different patterns in the raw header bytes, from 2019 to 2021. Unlike HTTP/2, SPDY uses a dynamic compression algorithm in the headers that makes it more vulnerable to chosen plain text attacks. Indeed, SPDY leads to more information leakage than HTTP/2 and is easier to classify. Moreover, we expect to see a change in the accuracy of UW-H model even when it is trained and tested on the newest datasets. We hypothesize better results on the 07-2019 datasets where there could be considerably more SPDY flows than the 2021 datasets.

We know that for DL models the dataset size has a direct impact on the overall classification accuracy. Therefore, in order to have a fair comparison, we reduced the number of flows in the 07-2019 dataset to the number of flows in the 04-2021 and 05-2021 datasets.

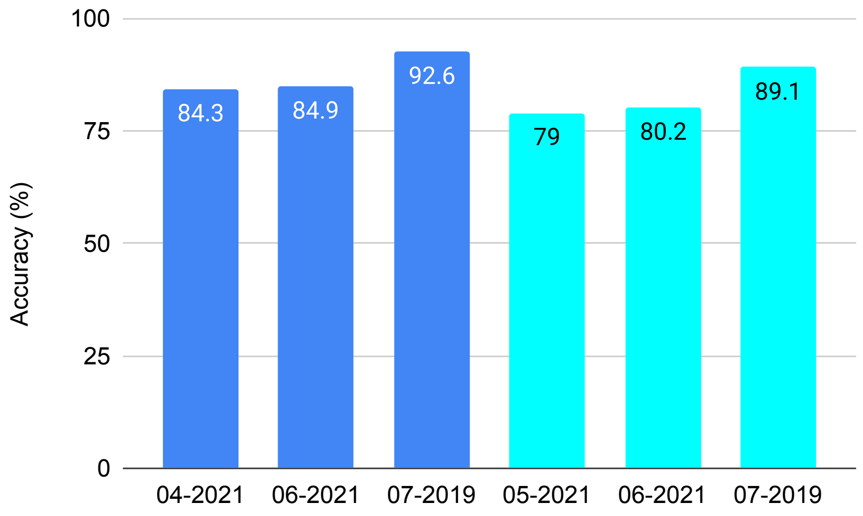


Figure 4.8: UW-H performance on datasets with similar sizes (dark blue: 04-2021 size, light blue: 05-2021 size)

Since we reduce the dataset using random sampling, we perform multiple experiments and report the average accuracy. The results are depicted in Figure 4.8. As can be seen, the accuracy of the model on the reduced 07-2019 datasets is still around 8% to 10% higher than on the other datasets. This suggests that the TLS headers in the 07-2019 dataset are easier to classify than the TLS headers in the newer datasets.

To confirm our hypothesis about the impact of the application-layer protocols, we conduct experiments based on the Application-Layer Protocol Negotiation (ALPN) header field of the TLS protocol. Table 4.6 shows the distribution of ALPN field values for different datasets. Note that all the 2021 datasets are merged. There are two main reasons for doing this: (i) 07-2019 and 09-2020 datasets consist of roughly 119K and 89K flows, respectively. In contrast, the 2021 datasets are considerably smaller and merging them results in 98.9K flows, which is comparable in size to the larger datasets; (ii) 2021 datasets are captured closer in time, which makes their data patterns rather similar as evident in Figure 4.2.

From Table 4.6, it is evident that between 62%-77% of flows in the considered datasets do not have an ALPN field value, i.e., Missing ALPN. Moreover, around 10%-20% of flows consist of HTTP/1 and HTTP/2 application-layer protocols, which are only a small portion of flows in each dataset. Therefore, we evaluate model performance in three different scenarios, where the flows in the datasets are either HTTP/1, HTTP/2, or unknown. It

Table 4.6: Distribution of ALPN field values for different datasets

ALPN filter	Dataset		
	07-2019	09-2020	Merged-2021
HTTP/2	0.12	0.09	0.09
HTTP/1	0.25	0.15	0.14
Missing ALPN	0.62	0.76	0.77

Table 4.7: UW-H performance based on ALPN

Dataset	Accuracy (%)		
	HTTP/2	HTTP/1	Missing ALPN
07-2019	93.5	97.5	93.2
09-2020	94.6	94.8	80.7
Merged-2021	91.6	96.9	81.1

is unknown for a flow with Missing ALPN, i.e., it may use HTTP/1, HTTP/2, or neither HTTP/1 nor HTTP/2. Table 4.7 illustrates the performance of UW-H on each dataset based on the ALPN field value. For HTTP/1 and HTTP/2, model performance across the datasets is more or less the same. However, the performance gap between the 07-2019 dataset and other datasets on flows with Missing ALPN is considerable. Specifically, UW-H achieves around 93.2% accuracy on the flows with Missing ALPN extracted from the 07-2019 dataset, while the performance is around 81% on the other datasets. This further substantiates that the TLS headers in the 07-2019 dataset are easier to classify, and the majority of this ease comes from flows with Missing ALPN.

By examining the ALPN of all the datasets, we found a few flows with application-layer protocols other than web protocols (e.g., Apple push-notification). Interestingly, the 07-2019 dataset is the only dataset that contains flows with the ALPN fields indicating the SPDY protocol. Recall from Table 4.5 that in the time frame corresponding to the 07-2019 dataset SPDY was still highly used, which we speculate as the reason for superior classification performance on the Missing ALPN portion of this dataset. Additionally, from Table 4.5 it can be seen that from 2019 to 2021 the adoption of HTTP/2 has increased by more than 83.3%, which substantiates previous findings.

For a fair comparison, we then reduce the number of HTTP/1, HTTP/2, and Missing

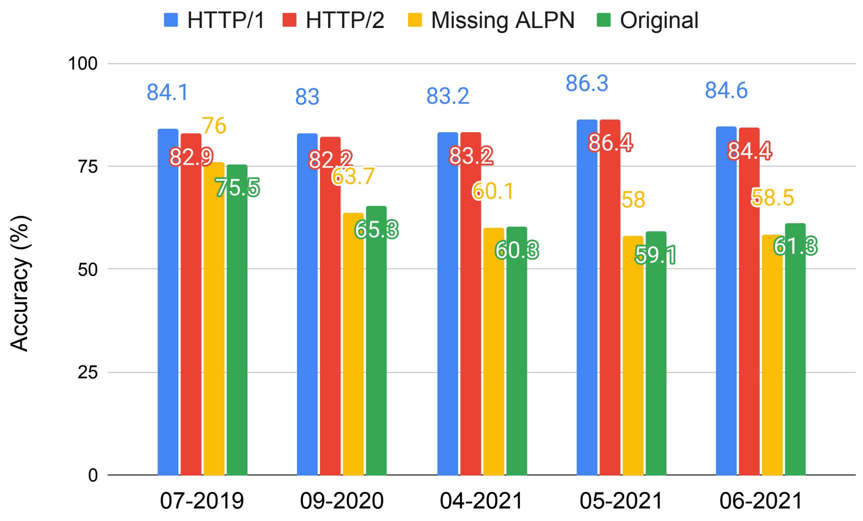


Figure 4.9: UW-H performance with ALPN filter on similar size datasets

ALPN flows in each dataset to the smallest across all the datasets (i.e., the number of HTTP/2 flows in the 05-2021 dataset). The results are depicted in Figure 4.9. It is evident that UW-H yields similar performance on HTTP/1 and HTTP/2 protocols. The accuracy is over 80% for all datasets on either HTTP/1 or HTTP/2 flows. However, the model shows inferior performance, i.e., around 60% average accuracy on the Missing ALPN portion of the datasets, except for the 07-2019 dataset which has a relatively higher accuracy of around 75%. Additionally, for the 09-2020 dataset, the performance of the model is lower than 07-2019 and higher than 04-2021 datasets. All of these results are inline with the increase in the adoption of HTTP/2 and decrease in SPDY usage over time in Table 4.5. This further substantiates our hypothesis that the Missing ALPN portion in the 07-2019 dataset is easier to classify. As the majority of the original flows (i.e., no filter on ALPN) are from the Missing ALPN portion, the performance of the model on the original flows is similar or slightly better than the Missing ALPN flows alone. It is better because of the small portion of HTTP/1 or HTTP/2 flows available in the original dataset compared to Missing ALPN flows.

We then investigate whether the model is biased on the ALPN field. Indeed, this could lead to better model performance when the ALPN field value is either HTTP/1 or HTTP/2. To investigate this, we obfuscate the ALPN field in the raw traffic bytes (e.g., replace with random bytes) and re-pre-process the data. We re-evaluate the UW-H model with the obfuscated ALPN field on HTTP/1 and HTTP/2 flows. Note that we leverage datasets

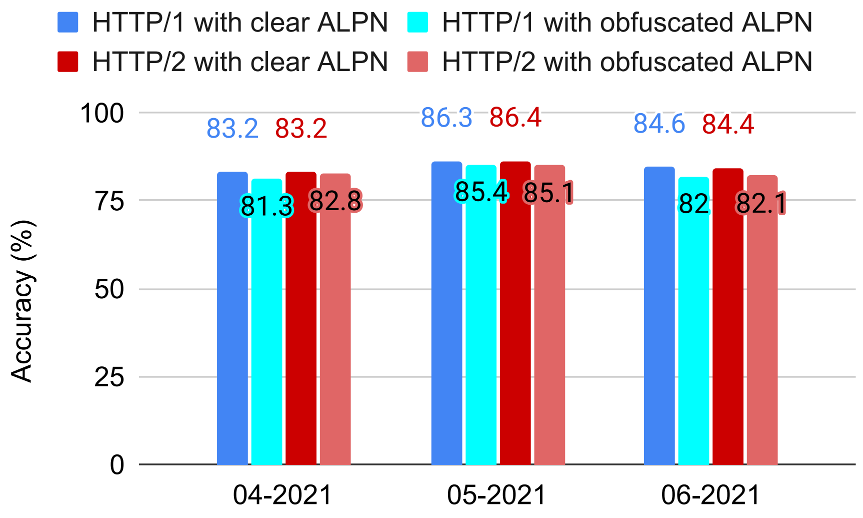


Figure 4.10: UW-H performance with ALPN obfuscation

with similar sizes as before and present average accuracy across multiple experiments. As shown in Figure 4.10, the ALPN field has an impact on classification performance, with lower performance when it is obfuscated. However, the performance degradation is only around 1%-2% in accuracy. For example, on the 04-2021 dataset, the model achieves 83.2% and 81.3% accuracy on HTTP/1 with clear ALPN and obfuscated ALPN, respectively. For HTTP/2, the accuracy is 83.25% versus 82.8%. Hence, a clear ALPN field is not the primary reason behind the model's performance gap between HTTP/1 and HTTP/2 flows with known ALPN, and other flows with Missing ALPN.

There are more protocols over TLS than HTTP/1 and HTTP/2 (e.g., Apple push-notification), and new and updated web protocols are likely to emerge over time. However, HTTP/1 and HTTP/2 are well established standard web protocols, and it is plausible that the model's performance over HTTP/1 and HTTP/2 protocols will remain rather consistent across different datasets in comparison to the unknown protocols. The results in Figure 4.9 support this claim with similar model performance for HTTP/1 and HTTP/2 web protocols. These results can be attributed to the existence of more information in flows that contain web traffic (e.g., HTTP/1 and HTTP/2) compared to other protocols (e.g., Apple push-notification). Therefore, web-related flows are easier to classify for the UW-H model.

Another hypothesis is that due to the negligible changes of the established protocols

Table 4.8: UW-H performance on datasets merged based on the ALPN filter

ALPN	HTTP/1 or HTTP/2	Missing
Accuracy (%)	95.2	83.0

over time, training the model on all historical HTTP/1 and HTTP/2 improves the model's accuracy, while training it on all unknown flows confuses the model despite the large number of samples in the dataset. To test this hypothesis, we merge all HTTP/1 and HTTP/2 flows of all datasets in one dataset, and all unknown flows of all datasets in another dataset. Table 4.8 illustrates the accuracy of the UW-H model on the HTTP/1 and HTTP/2 flows of all the datasets versus the merged unknown portion of all datasets. We see that the model shows an accuracy of 95.2% on the first dataset, compared to an accuracy of 83.04% on the second one. We also notice that the accuracy on the unknown portion is low, despite the large number of flows. Therefore, it seems that training the model on a merged dataset of HTTP/1 and HTTP/2 flows helps its performance, whereas training the model on more unknown flows seems to confuse the model, possibly because of the more varied patterns and protocols in that portion of the dataset.

Chapter 5

Manual Architecture Adaptation

In this chapter, we examine the performance of the UW model on the 2021 datasets. Observing a drop in model accuracy, we suggest updating the model architecture that improves accuracy on several datasets, thus making it more robust to data drift.

5.1 Ensuring model convergence

We start by training and testing the UW model and the decomposed models on datasets from 2021. Again, we skip the UW-A as its performance is negligible compared to the other models. The results of these experiments for service-level classification are shown in Table 5.1.

Although suffering a drop from the baseline 2019 dataset, the accuracy of the UW model is reasonable at 83.4% and 87.1% on 05-2021 and 06-2021 datasets, respectively.

Table 5.1: Model performance across the 2021 datasets in service-level classification

Model	Accuracy (%)		
	04-2021	05-2021	06-2021
UW	40.0	83.4	87.1
UW-F	11.0	81.0	85.9
UW-H	84.3	79.0	85.8

Table 5.2: Model performance across the 2021 datasets in application-level classification

Model	Accuracy (%)		
	04-2021	05-2021	06-2021
UW	85.2	79.6	88.9
UW-F	81.2	84.2	86.4
UW-H	85.9	74.1	86.5

However, the model has a rather peculiar accuracy of only 40% on the 04-2021 dataset, which is primarily attributed to UW-F, showing a mere accuracy of 11% (i.e., worse than a random classifier). On the other hand, the UW-H performs reasonably on the same dataset.

Before we delve into the reasons for the under performance of UW-F, we note that the lower performance of the model on 2021 datasets compared to 07-2019 dataset can be attributed to dataset size. 07-2019 dataset had 119K labeled flows, whereas 04-2021, 05-2021 and 06-2021 datasets have 42K, 17K and 51K flows, respectively. Therefore, given a much larger amount of training data, we expect the model to achieve a higher accuracy on 07-2019 dataset regardless of the architecture. However, the dismal accuracy on 04-2021 dataset cannot be simply explained by dataset size, and has to do with the model itself.

The results for the same experiment in application-level classification are summarized in Table 5.2. The overall classification results are better than for service-level classification, which is inline with the previous experiments (i.e., 2019 dataset). Furthermore, the same trends as service-level classification more or less hold in application-level classification. UW-H model performs relative to the dataset sizes, i.e., it achieves the highest and lowest accuracies on the biggest and smallest datasets, respectively. Moreover, the trends for UW-F are similar as for service-level classification but with more promising results. With application-level classification, there is no performance peculiarity for UW-F on the 04-2021 dataset. Nevertheless, UW-F performs the worst (i.e., 81.2% accuracy) on the 04-2021 dataset, albiet much better than the accuracy in service-level classification (i.e., 11%).

To troubleshoot the UW-F model performance on 04-2021 dataset, we examined the confusion matrix and accuracy of the model in the training phase, epoch by epoch. We found that the model does not converge, and the same class is predicted for all samples in each epoch. We tried two alterations to the model to alleviate this problem: (i) Learning Rate reduction—The learning rate for the optimizer was reduced from the default value of 0.001 [4] to 0.0001 (i.e., 10x reduction); (ii) Masking Layer addition—A masking layer

Table 5.3: UW-F adaptation best practices for service-level classification

Dataset	Adaptation	Training flows	Accuracy (%)
04-2021	Dropout + Learning Rate	33,900	89.4
	Dropout + Learning Rate + Masking Layer	33,900	90.1
05-2021	Dropout	14,024	87.3
	BLSTM + Dropout	14,024	88.2

Table 5.4: UW-F adaptation best practices for application-level classification

Dataset	Adaptation	Training flows	Accuracy (%)
04-2021	Dropout + Learning Rate	21,040	88.6
	Dropout + Learning Rate + Masking Layer	21,040	90.2
	BLSTM + Dropout + Learning Rate + Masking Layer	21,040	90.1
05-2021	Dropout	8,861	85.8
	BLSTM + Dropout	8,861	85.3
	CONV1D + Learning Rate + Masking Layer	8,861	85.9

was added at the beginning of the UW-F. The masking layer acts as a de-noising layer to filter out time-steps that do not have any information. Therefore, these time-steps can be skipped in the LSTM layer.

The above alterations boosted the accuracy of the UW-F model on the 04-2021 dataset from 11% to 88.3% in service-level classification. A smaller learning rate makes it more likely for the model to eventually converge to global optima, although it increases training time. A masking layer reduces data noise, while adding to the complexity of the model. Despite the downsides, evidently, in the case of the 04-2021 dataset, these alterations are necessary for the model to achieve reasonable performance in service-level classification.

5.2 Adjusting to dataset size

Given that dataset size can contribute to the model's drop is accuracy on the 2021 datasets, we suggest a number of best practices in designing a model architecture for smaller datasets, based on a number of experiments carried out on the two smallest datasets, i.e., 04-2021 and 05-2021.

Dropout rate reduction

The stacked LSTM in the UW-F model is followed by a dropout layer. The dropout layer randomly sets the units of LSTM output to zero based on the dropout rate, which is often used to avoid model overfitting. We found that in a smaller dataset, a high dropout rate does not help, as it sets units of valuable information to zero, thus leaving the final layers of the model with little information to work with. By reducing the dropout rate from 0.5 (i.e., default in [4]) to 0.3, we saw a boost in model accuracy on both 04-2021 and 05-2021 datasets, the two smallest datasets, as shown in Table 5.3 for service-level classification. With the same adaptation, a performance boost is also noticeable in application-level classification for the 04-2021 and 05-2021 datasets, as depicted in Table 5.4.

UW-F simplification

The stacked LSTM layer proposed in [4] is a complex UW-F model for flow time-series input with too many parameters for a small dataset. By reducing the number of LSTM layers by one, thus turning the stacked LSTM to a bidirectional LSTM (BLSTM), we were able to obtain better results on datasets with less than 20K flows, as shown for 05-2021 dataset in Table 5.3. We further found that on datasets smaller than 10K flows, even reducing the stacked LSTM layer to a 1D Convolution (CONV1D) layer helps UW-F performance in achieving comparable or better results with a lower number of parameters (i.e., a lighter and faster to train model), contrary to what was shown in [4] for large datasets.

The results for application-level classification with similar adaptations are shown in Table 5.4. For application-level classification the reduction of stacked LSTM layers to BLSTM results in a slightly lower model accuracy on the 05-2021 dataset (i.e., 85.3%). However, since this dataset has smaller than 10K flows, we leverage CONV1D layers masking layer addition and learning rate reduction). Evidently, with these adaptions, the UW-F model achieves the best performance in application-level classification with an accuracy of 85.9%.

Best practices

Table 5.5 summarizes our recommended best practices based on a given dataset's size. We suggest that when leveraging the UW model, UW-F should be adapted to the training dataset's size. When there are fewer than 50K training flows, reducing the dropout layer value (e.g., 0.3) is sufficient. If the number of samples are fewer than 20K, a simpler architecture such as BLSTM is preferred over stacked LTSM. In the UW model architecture

Table 5.5: UW-F architecture adaptation rules

Number of flows	Adaptation
<= 50K	Dropout reduction
<= 20K	BLSTM
<= 10K	1D Convolutions [4]

shown in Figure 3.1a, changing the stacked LSTM to a BLSTM would simply remove the last LSTM layer in the stack, as each LSTM works in reverse direction to the previous one. As an example, since 04-2021 dataset has 21K training flows in application-level classification, we leveraged the BLSTM architecture instead of stacked LSTM layers. This resulted in a similar accuracy of 90.1% which is only 0.1% lower than the accuracy of the more complex stacked LSTM architecture, as shown in Table 5.4. Finally, if the dataset has fewer than 10K flows, using simple 1D Convolutions (i.e., depicted in [4] appendix) is adequate and preferable over the LSTM layer.

5.3 QUIC results

We also evaluate the performance of the UW model on real-world QUIC data, before and after employing the adaptation best practices proposed in the previous subsection. We show that these guidelines indeed improve model accuracy on a dataset consisting of QUIC flows, thus showing that our adaptation best practices generalize to encrypted protocols other than TLS.

UW-F was shown to achieve over 99% accuracy on a synthetic QUIC dataset [4]. However, on our real-world QUIC data, i.e., QUIC-05-2021, the model achieved 86.7% and 83% accuracy in service-level and application-level classification, respectively. Therefore, we chose the following architectural adaptations for the model: (i) decreasing the initial learning rate, (ii) adding a masking layer, and (iii) reducing the dropout rate to 0.4. The performance of UW-F before and after adaptions for service-level and application-level classification are shown in Figure 5.2 and Figure 5.3, respectively.

The model achieves an accuracy of 86.7% before adaptation, whereas the adapted model achieves 95.6% for service-level classification. Furthermore, the application-level classification accuracy is boosted from 83% to 91%. A similar trend is visible in other performance metrics, such as weighted average F1-score, precision, and recall, where the adapted model

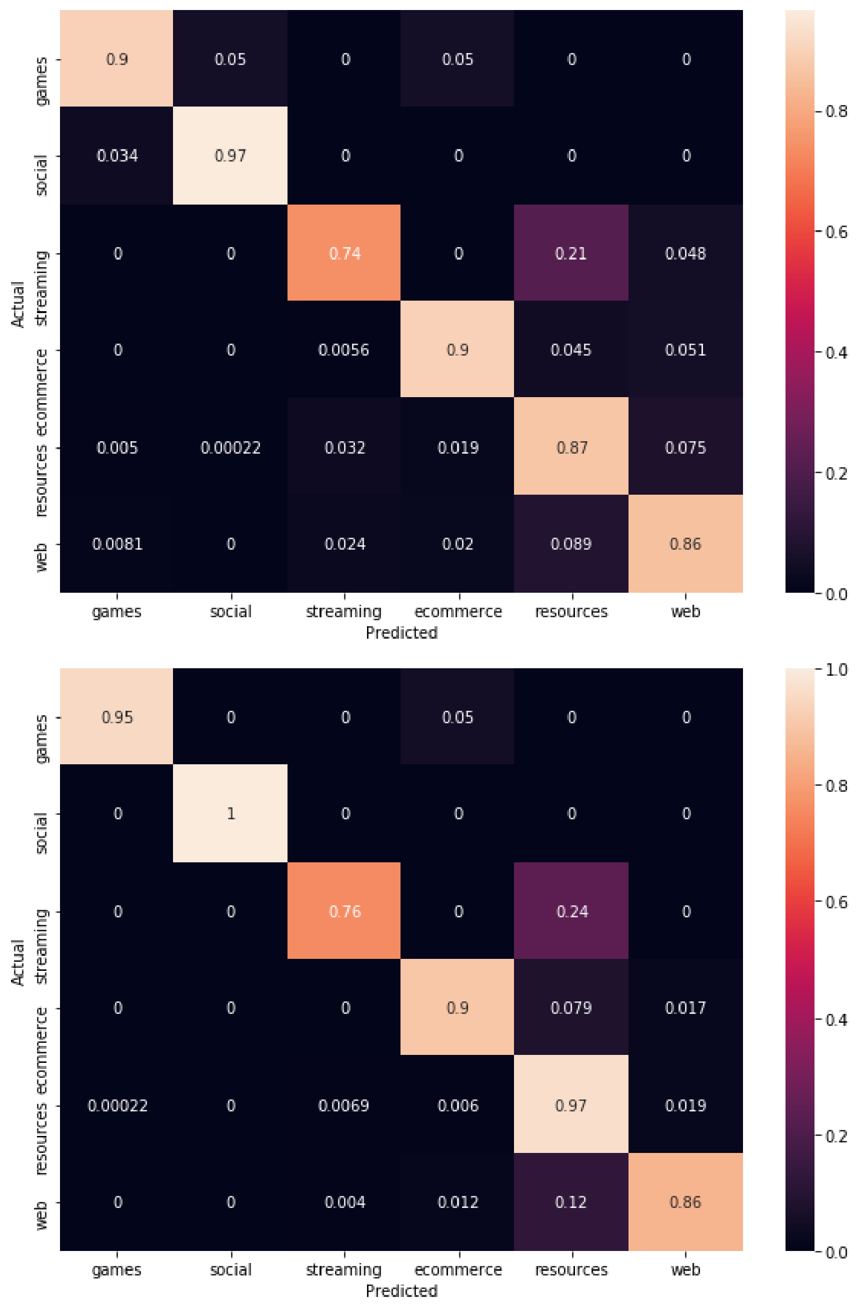


Figure 5.1: Confusion matrix for UW-F (*up*) vs. its adapted version (*down*) on the QUIC-05-2021 dataset in service-level classification

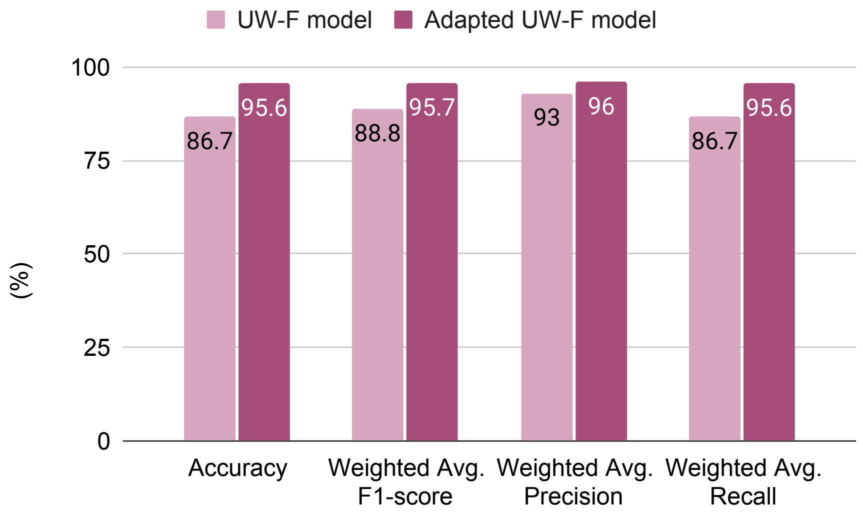


Figure 5.2: UW-F performance with and without adaptations on the QUIC-05-2021 dataset in service-level classification

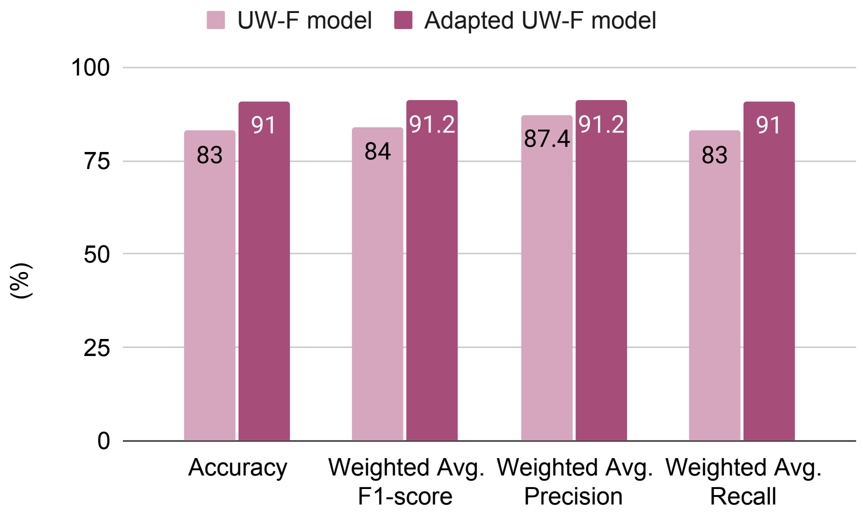


Figure 5.3: UW-F performance with and without adaptations on the QUIC-05-2021 dataset in application-level classification

outperforms the original UW-F model by 3% to 9% in service-level classification and by 3.8% to 8% in application-level classification. The precision for both models is quite high, however, the main advantage of the adapted model is correctly predicting a larger portion of flows for each class, which results in a 9% and 8% increase in recall for service-level and application-level classification, respectively. Figure 5.1 and Figure 5.4 show the confusion matrices of UW-F and its adapted version for service-level and application-level classification, respectively. The recall increase is visible in the confusion matrices, where the adapted model achieves a higher accuracy per class in service-level classification. Furthermore, for application-level classification the adapted UW-F model receives significantly higher accuracy across 75% of the application classes, especially for classes with the lowest accuracy without adaptation. Therefore, the adaptations allow the model to achieve a higher classification accuracy across classes. The most significant increase is for the *resources* class with 10% increase in accuracy for service-level classification. Similarly, the adaptation results in a 19% increase in accuracy for the *resourcePbstck* class in application-level classification.

For the ease of use, we created a software API for the full pipeline of data pre-processing to adaptation of models with just a few configurations and lines of code in Python. The software API is described in Appendix .4.

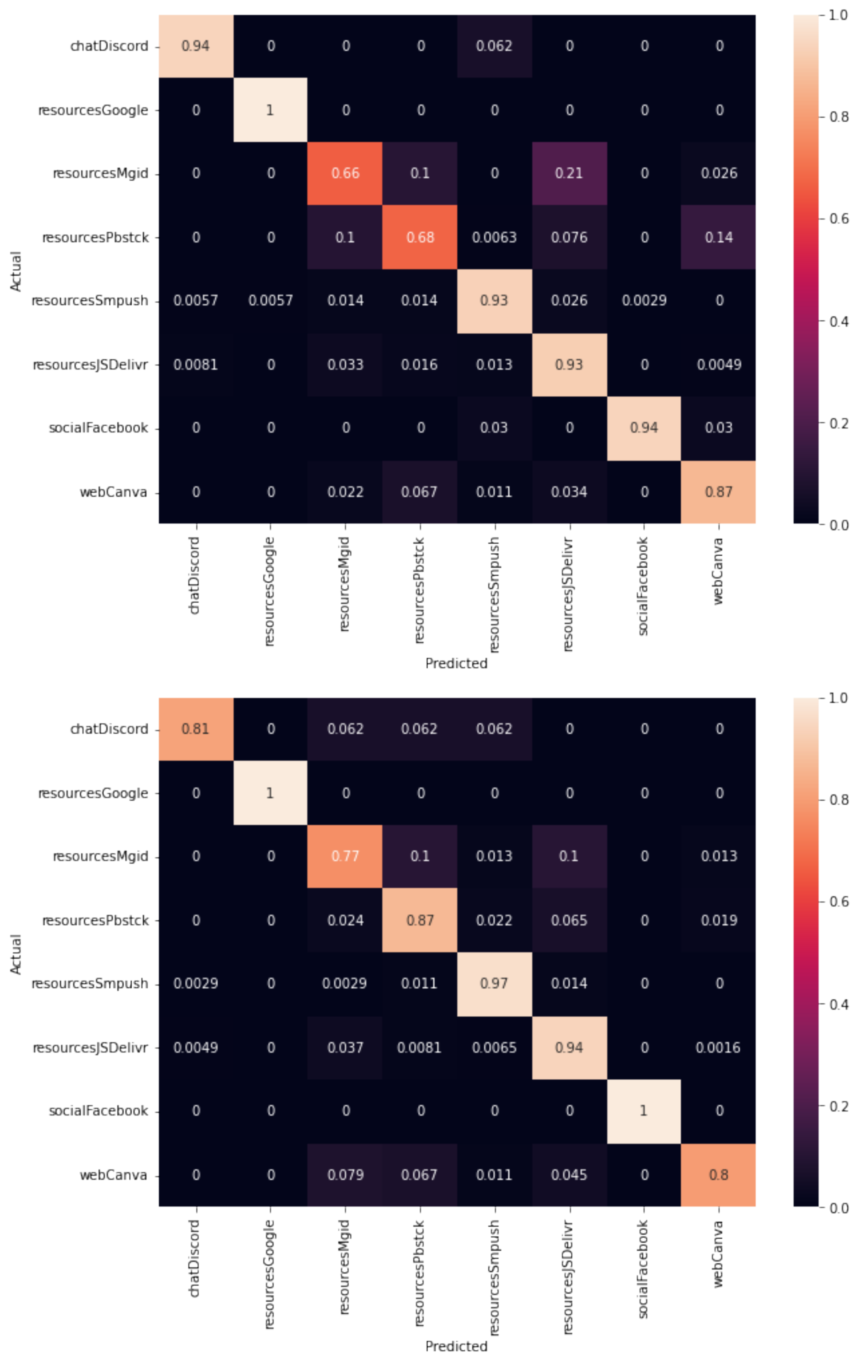


Figure 5.4: Confusion matrix of the UW-F (*up*) vs. its adapted version (*down*) on the QUIC-05-2021 dataset with application level-classification

Chapter 6

Methodology for Automatic Approach

In this Chapter, we discuss the search spaces designed for each feature type. We also briefly explain the state-of-the-art Encrypted Traffic Classification models used in this work to showcase the benefits of *AutoML4ETC* over the existing manual approaches.

6.1 Search Spaces

We design search spaces for: *(i) flow statistics*, *(ii) packet raw bytes*, and *(iii) flow time-series*. All search spaces should be connected to a *Softmax* layer to produce the final classification. With respect to the terminology used in Tables 6.1, 6.2, and 6.3, *Choice* corresponds to the search algorithm choosing a value from a range of *values* for the parameter; *Optional* means that the layer is optional; *Permutation* only organizes the best permutation of values; *Reduce factor* defines the sequential amount of reduction in dense units.

(i) Flow Statistics Search Spaces

The state-of-the-art ETC models typically resort to a simple architecture for consuming flow statistics (e.g., [4, 30]). Therefore, for the flow statistics search space, referred to as *MLP*, we use a sequence of dense layers and a permutation of dropout, batch norm, and activation layers between every two dense layers, as depicted in Figure 6.1. We repeat this

Table 6.1: MLP search space parameters, values, and types

Parameter	Dense units	Number of Dense layers	Reduce factor	Activation	Batch norm	Dropout	-
Values	[100, 200, 400]	[3, 4, 5]	[1, 0.7]	[relu, elu]	Boolean	(0.3,0.5)	[dropout, batch norm, activation]
Type	Choice	Choice	Choice	Choice	Optional	Real number	Permutation

Table 6.2: CNN search space parameters, values, and types

Parameter	CONV Block Repeat	Kernel size	Filter size	Dropout	-	Pooling layer	Activation	Batch norm
Values	[2,3,4,5,6]	[(1,1), (2,2)]	[32, 64]	(0,.05)	[dropout, activation, batch norm]	[MaxPool, AveragePool]	[relu, elu]	Boolean
Type	Choice	Choice	Choice	Real	Permutation	Choice	Choice	Optional

Dense Block several times, which is specified by the *Dense Block Repeat* parameter. These blocks are connected and the number of units in the dense layer is reduced in the next block by the *Reduce factor* parameter. The parameters of the flow statistics search space are summarized in Table 6.1.

(ii) Packet Raw Bytes Search Spaces

For *packet raw bytes*, we implement two different search spaces: *(i) CNN + MLP*, and *(ii) AutoML4ETC*.

CNN + MLP

The *CNN + MLP* search spaces are inspired by the ETC state-of-the-art [14, 4, 42, 43]. The overall structure for the CNN (i.e., 2-D CNNs or 1-D CNNs) search space is depicted in Figure 6.2. Each *CONV Pool Block* is a sequence of one or more *CONV Blocks* connected to a pooling layer. The number of CONV Pool Blocks in the sequence is determined by the *CONV Pool Block Repeat* parameter. Moreover, the inner CONV Block is also sequentially repeated *CONV Block Repeat* times, where the two repeat parameters are independent of one another. Additionally, for the first two CONV Pool Blocks, we set the CONV Block Repeat to two repetitions. Thereafter, the search algorithm can choose CONV Block Repeat from a range of 3 to 5 which is a typical number of convolutional

Table 6.3: AutoML4ETC flow time-series parallel SLSTM search space parameters

Parameter	Initial dense units	Number of LSTM layers	LSTM Direction	LSTM unit size	Activation	Dropout
Values	[256, 512]	[1, 2]	Boolean	[128, 256, 512]	[relu, elu]	(0.3,.05)
Type	Choice	Choice	Mandatory	Choice	Choice	Real

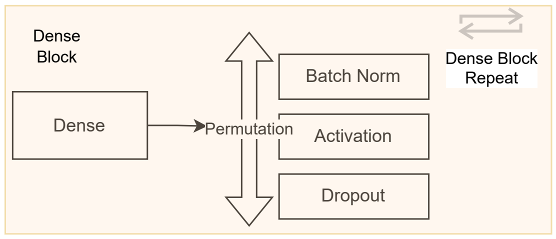


Figure 6.1: MLP search space overview

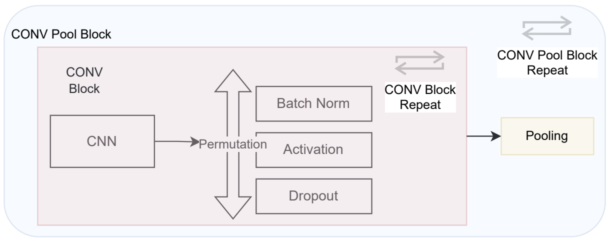


Figure 6.2: CNN search space overview

layers in ETC [4, 42]. Also, the number of filters is cut in half in the following repetitions of CONV Block, i.e., after the first two CONV Pool Blocks. The parameters for this search space are summarized in Table 6.2. The *CNN* search space is sequentially connected to the *MLP* search space to construct the *CNN + MLP* search space.

AutoML4ETC

The core of *AutoML4ETC* search space is derived from the *ENAS micro search space*, which offers maximum flexibility and parameter efficiency combined with *CNN + MLP* search space. More specifically, the search algorithm only generates a single *Normal cell* and a single *Reduction cell*. The incentive for this approach is to learn simpler and more generalizable models instead of complex sequential convolutional architectures, as depicted in Figure 6.5.

Each *cell* contains four nodes. Figure 6.3 shows an example of the inside of a cell with four nodes. For each node, the search algorithm makes the following decisions:

1. Choose *Input 1* and *Input 2* from the output of the previous nodes. If it is the first node, choose from the inputs of the *cell*.

2. Choose the operation for *Input 1* and *Input 2* from: (i) identity, (ii) separable convolution hyper-layer with kernel size 3 or 5, and (iii) average or max pooling with kernel size 3.

3. Add the output of the two operations and return this as the output of the node.

A *separable convolution hyper-layer* is based on [64], consisting of sequentially connected layers of Relu, separable convolution, batch normalization, and 0.4 dropout rate. We find that forcing a high-rate dropout layer during search results in a more generalizable model

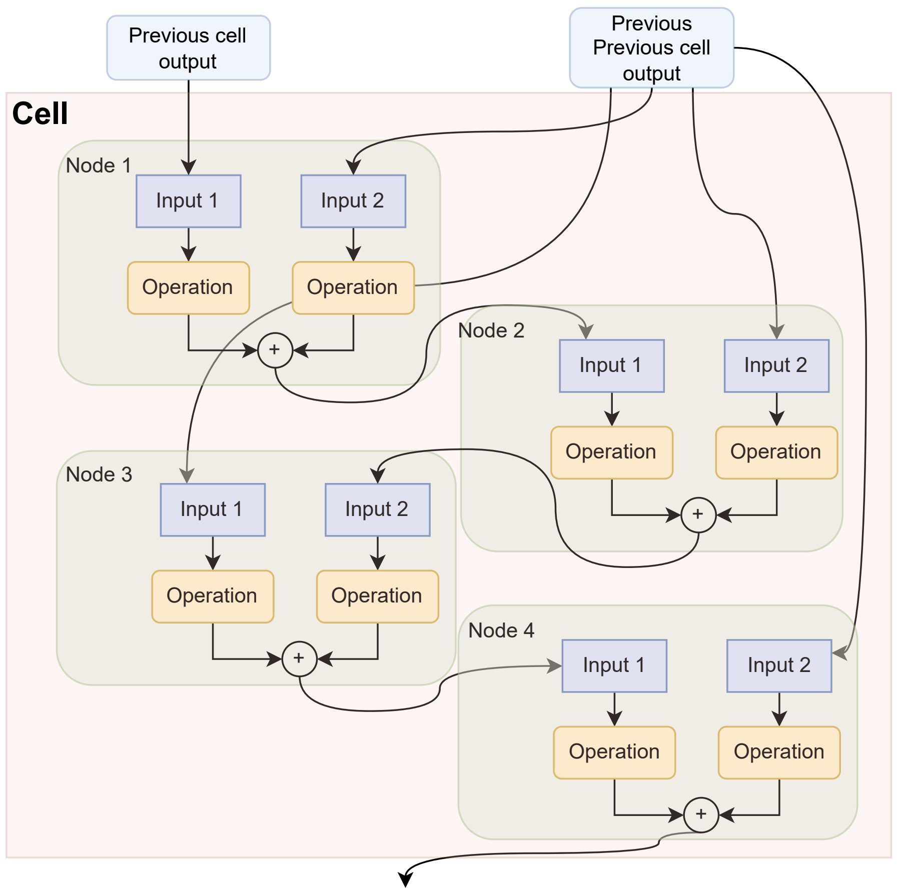


Figure 6.3: Example cell components

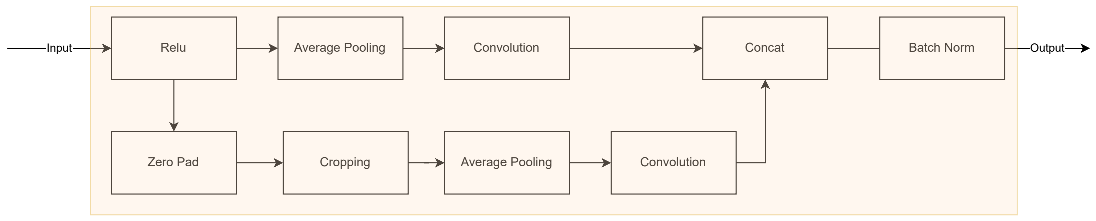


Figure 6.4: Factorized Reduction hyper-layer modules

with less overfitting over the training data. Furthermore, similar to [64], we also use two sequentially connected *separable convolution hyper-layers* every time the search algorithm chooses this operation.

As depicted in Figure 6.5, the input goes through a hyper-layer before entering a cell. This hyper-layer is either a *Filter Alignment* or a *Factorized Reduction* layer depending on the cell type.

- *Filter Alignment hyper-layer*: A Normal Cell is preceded by a Filter Alignment layer which consists of sequential Relu, convolution, and batch normalization. This layer ensures the existence of the number of filters that need to be at the beginning (i.e., 64 initial filters).
- *Factorized Reduction Hyper-layer:* A Reduction Cell is preceded by a Factorized Reduction layer with a structure depicted in Figure 6.4. This layer simply processes and reduces the input size to half.
- *Loose ends*: In some cases, the search algorithm may not choose the outputs of all the nodes inside a cell to be used by other nodes. We call these unused outputs *Loose ends*. Since Loose ends may also contain useful information, they are fed to the add operation at the output of the last node of the cell.

(iii) Flow Time-series Search Spaces

For the *flow time-series* we implement two different search spaces: *(i) AutoML4ETC*, and *(ii) SLSTM + MLP*.

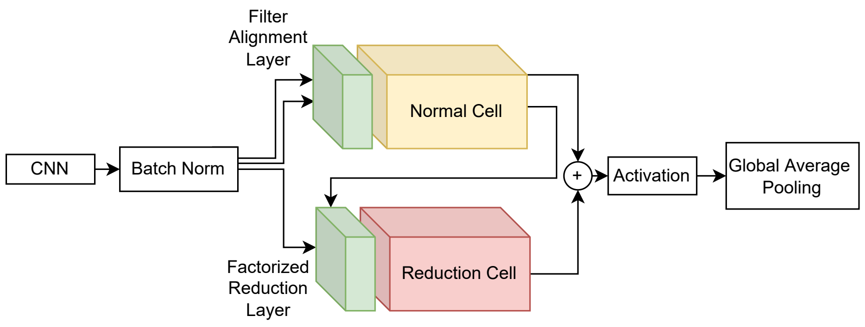


Figure 6.5: *AutoML4ETC* for the packet raw bytes search space

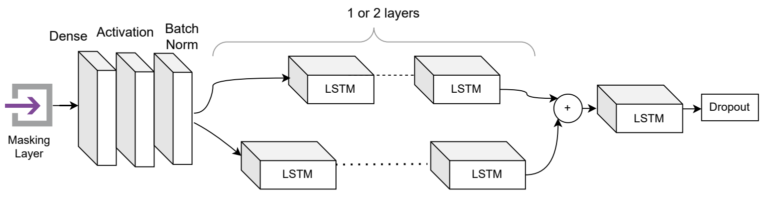


Figure 6.6: *AutoML4ETC* for the flow time-series search space

AutoML4ETC

Sequential LSTM layers [4] have shown to be the most effective neural architecture for time-series information in ETC. However, the core idea behind *AutoML4ETC* is to use the benefits of *ResNet*-inspired architectures combined with sequential LSTM layers. Specifically, instead of one sequential LSTM layer (i.e., SLSTM), the outputs of two parallel SLSTM layers are added to one another. This method enables the neural network to extract different useful time-series information in each parallel SLSTM layer. Figure 6.6 depicts this search space. The initial processing layers consist of a *Masking layer* followed by dense, activation, and batch normalization layers. The Masking layer simply skips the time-steps whose value is meaningless, e.g., zero-padded, to denoise the input. The output of the initial processing layers is fed to a parallel SLSTM structure. The search algorithm chooses the number of LSTM layers, directions, and units in each branch. The outputs of the two parallel SLSTM structures are added to one another and passed to a final SLSTM layer. Finally, there is a dropout layer to reduce overfitting. This search space is followed by the *MLP* search space and a final softmax layer to produce classification results. The parameters of this search space are summarized in Table 6.3.

SLSTM + MLP

The *SLSTM + MLP* search space is similar to AutoML4ETC, except it contains sequential SLSTM layers instead of parallel SLSTM addition layers. This approach is based on the state-of-the-art ETC methods for time-series information (e.g., [4, 32]). The motivation for having this search space is to show the advantage of a parallel over a sequential LSTM structure.

Chapter 7

Evaluation of the Automatic Approach

In this chapter, we start by describing the datasets and the data pre-processing pipeline in Section 7.1. The software and hardware stack are described in Appendix .5. We evaluate *AutoML4ETC*'s packet raw bytes and flow time-series search spaces in Section 7.2 and Section 7.3, respectively; we compare them against state-of-the-art search spaces, study the performance of different search algorithms on the search spaces, investigate child-model performance estimation through partial training, and compare *AutoML4ETC*-generated architectures against state-of-the-art models using different datasets. In Section 7.4, we experiment with combining and ensembling *AutoML4ETC*-generated models within and across search spaces. Finally we provide insights in Section 7.5 about traffic measurements for TLS and QUIC real-word datasets. All of the reported metrics here are validated on the testing part of each dataset.

7.1 Datasets

Our approach is evaluated on five real-world datasets of traffic traces captured on a major ISP network, as well as a public synthetically-generated and pre-processed QUIC dataset. Three of the real-world datasets consist of TLS traffic and the other two consist of QUIC traffic. The real-world datasets were created by pre-processing and labeling the collected traffic traces and named after their year and month of capture. A brief description of the datasets is available in Table 7.1. In each dataset, we reserve 80% of the labeled data samples for training and 20% for validation.

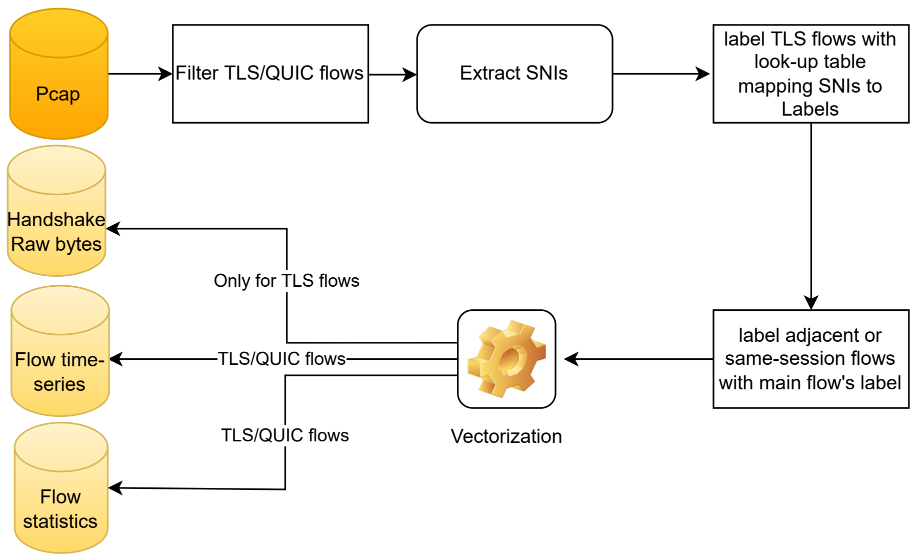


Figure 7.1: Overview of the pre-processing procedure for real-world datasets

The overall procedure for pre-processing and labeling raw packet captures (i.e., PCAP files) into ML-usable datasets is shown in Figure 7.1. As the first pre-processing step, packet payloads beyond the TLS or QUIC header were removed and the IP addresses masked to preserve user privacy. The resulting pcap files were broken into *flows*, where each flow consists of packets close in time that share source IP, destination IP, source port, destination port, and protocol. From each TLS flow, three types of data were extracted:

- Flow time-series is a sequence of packet inter-arrival times and signed packet sizes, where packet direction determines the sign.
- TLS raw bytes include headers of the first three TLS handshake packets, in which the TLS Server Name Indication (SNI) and TLS cipher information are obfuscated.
- Flow statistics are extracted using CICFlow-meter [30] on the datasets during the pre-processing step. These statistics include standard deviation, mean, minimum, maximum of packet sizes, inter-arrival times, and TCP flags, amongst others.

We only use Flow time-series and Flow statistics for QUIC datasets as it has been shown in [4, 43] that they are effective for ETC.

Table 7.1: Dataset properties

Protocol	Type	Dataset name	Percentage of labeled flows (%)	Number of labeled flows (thousand)	Dataset classes
TLS	Real-world	July 2019	16	119.8	chat, download, games, mail, search, social, streaming, Web
		April 2021	15	42.3	
		May 2021	41	51.2	
QUIC		QUIC - April 2021	72	37.5	web, social, streaming, ecommerce, resources, games
		QUIC - May 2021	68	26.0	
	Synthetic	QUIC - UCDavis [43]	100	3.63	Google Docs, Google Drive, Google Music, Google Search, YouTube

The real-world datasets were labeled based on the SNI field in each flow, which is one reason why we obfuscate the SNI value in preprocessing. Not all flows contain a readable SNI value. Moreover, the utilization of clear SNI is likely to decline in favor of the proposed Encrypted SNI (ESNI) extension [41]. Hence, we need traffic classifiers that learn the intrinsic characteristics of the flows rather than a trivial mapping from SNIs to classes.

We use an approximate labeling function to extract labels from SNIs. We developed a look-up table by visiting top websites in each service class and extracting regular expressions from their domain names that are matched with the SNI value to map each SNI to a label. Because not all flows contain an SNI value, we also label adjacent flows (i.e., flows with the same TLS session-id or close-enough starting time) based on the main flow that has an SNI to increase the number of labeled flows.

The labeling module was initially designed based on the TLS datasets and then adapted to the QUIC datasets. We observed from the *SNI*s in the QUIC dataset, that some of the TLS dataset classes did not have any instances in the QUIC datasets. However, some other classes such as *resources, ecommerce* appeared that did not fit in the TLS categories.

Resources class consists of flows that contain materials for the page content, such as *JS APIs*, that are often offered by some major providers (e.g., Cloudflare). Additionally, *ecommerce* is mostly referred to the marketing and commercial-related services.

There are two main reasons for having slightly different classes for QUIC and TLS: *(i)* QUIC is still a relatively new protocol and is not yet as widely adopted as TLS, and *(ii)* QUIC offers a higher connection speed than HTTP over TLS; therefore, many of the less time-sensitive applications (e.g., mail, download services) switched to QUIC. On the other hand, more time-sensitive services such as *resources* have adopted QUIC which has an impact on the loading time of websites.

7.2 Packet Raw Bytes-oriented NAS

In this section, we focus on evaluating our proposed *packet raw bytes*-oriented neural architecture search method. We leverage various TLS datasets for this part as it has been shown that TLS handshake raw bytes can achieve high classification accuracy [42, 4].

Our method has three major components: (i) search space, (ii) search algorithm, and, (iii) the training strategy for child models. We start by evaluating and discussing our proposed *packet raw bytes*-oriented search space. Then, we evaluate and compare different state-of-the-art neural architecture search algorithms on our search space. We also evaluate and compare different child model training strategies. For all of the above experiments, we use the May 2021 TLS dataset which is the smallest of our TLS datasets. Finally, we compare our *AutoML4ETC*-generated architecture to the state-of-the-art ETC models across all the real-world TLS datasets to show the effectiveness of our approach.

Evaluation of the search space

In this section, we focus on the search space design for the packet raw bytes features. The end goal for a packet raw bytes-based traffic classifier is to achieve high performance in terms of classification accuracy, preferably with low complexity in terms of the number of parameters of the neural architecture for faster predictions. These two characteristics combined are desirable for the early classification of TLS flows [4, 42].

We use the same baseline search algorithm, i.e., Random Search (RS), across all the studied neural architecture search spaces as the focus here is on the impact of the search space on the performance of the best possible child model. Table 7.2 summarizes our findings on the *May 2021* TLS dataset when the child model is trained for 40 epochs. We will experiment with different numbers of epochs and other datasets in the following sections.

The best child model generated from the *CNN-2D + MLP* search space after 200 trials achieves a 77.55% accuracy with almost 22 million parameters. This search-space represents a naive approach towards using NAS for ETC where the input is transformed into an image format and NAS is applied the same way as in image classification (i.e., [49] with NAS). However, turning raw bytes into 1-D vectors and using 1-D CNNs can boost the performance of the model. This is because images have 2 dimensions (i.e., pixels) and both dimensions of them have a meaning. However, raw bytes are not actual images to have a meaningful 2 dimensional representation. Evidently, on the *CNN-1D + MLP* search

Table 7.2: *Packet raw bytes* search spaces comparison

Search Space	Search Algorithm	Trials	Accuracy (%)	Parameters (Thousand)
AutoML4ETC	**RS**	**100**	**82.86**	**111.5**
CNN-2D + MLP	RS	200	77.55	21,940.5
CNN-1D + MLP	RS	200	78.75	12,116.1
ENAS micro	RS	200	80.4	120.6

space, it is possible to obtain a child model that achieves 78.75% accuracy with almost half the number of parameters after 200 trials.

RS on the *ENAS micro* search space can find a much lighter model (i.e., 120.6 thousand parameters) that achieves higher accuracy (i.e., 80.4%), after 200 trials as well. However, with the *AutoML4ETC* search space, that combines ideas from the above spaces, it is possible to generate a classifier that is not only more accurate (i.e., 82.86% classification accuracy) but also lighter (i.e., 111.5 thousand parameters) than any of the above after only half the number of trials (i.e., 100 trials). Hence, our *AutoML4ETC* search space outperforms state-of-the-art search spaces both in terms of accuracy and complexity of the best child model on this dataset.

Comparison of Search Algorithms

In this section, we explore different neural architecture search algorithms and study their impact on the performance and complexity of the best child model. More precisely, we experiment with *RL* [38], *MCTS* [58], and *EA* [39], and compare them against the baseline *RS* algorithm. These search algorithms (excluding *RS*) were previously developed and evaluated based on the *NASNet* or *ENAS Micro* search space from which we derived our *packet raw bytes* search space. Therefore, comparing them against each other is particularly relevant here.

For a fair comparison, we fix the other parameters of our *AutoML4ETC* method as follows. We set the number of child model training epochs to 10 for faster training and set the total number of trials to 100. We will discuss the effectiveness of the 10 epochs of child model training in the next section.

We compare the mean accuracy of the top-N child models found by each of the search algorithms for N=1, 5, 10, 20, and 30. This is because we are not only interested in comparing the performance of the global best child models, but also we want to compare the overall ability of each search algorithm to find reasonably good (i.e., reasonably accurate) child models throughout the search process.

Performance evaluation results are depicted in Figure 7.2. Several interesting observations can be made here. First, concerning the global best child models (i.e., top-1 child models), it is evident that all of the search algorithms lead to equally well-performing best child models; the standard deviation of the accuracy distribution across the best child models is only 0.25%. This result highlights in particular the power of our search space, where the simplest search algorithm (i.e., RS) can perform as well as much more complex search algorithms at the cost of a more parameters (i.e., 206.53, 231.62, 242.12, and 258.24 thousand for the *RL, MCTS, EA*, and *RS*, respectively).

Indeed, as pointed out earlier, many of these algorithms were previously validated in a setting that is not available to any ordinary user, in terms of resources (more than 400 GPUs [39, 63, 64]) and time complexity (e.g., up to 50,000 trials [64], 310 epochs for architecture search [38], etc.). This may suggest that in a more prevalent cost-effective setting, these algorithms may not converge to their best results. From another perspective, even with massive resources, *RS* remains competitive, achieving comparable results as the other more complex search algorithms. In [58], the authors compared a variation of *EA* and *MCTS* to *RS* and the spectrum of the measured accuracy ranges between 94.1 and 94.2%. The same applies to [63, 38, 39] showing that the performance of *RS* is within 1% of the performance of the other algorithms.

Another interesting observation is that as N increases, the gap between the mean accuracies of the top-N child models grows. While the *MCTS* algorithm is the top performer for N=5,..,30, *RL* scores the worst performance. Moreover, in this same range, the *EA* and *RS* algorithms perform almost identically and score in between *MCTS* and *RL*. This suggests that *MCTS* can build more top-performing models with a few trials and child model training epochs.

All of these findings and observations suggest that designing a good search space is more important than the search algorithm itself when using fewer trials and epochs for child model training. A good search space is when most of the architecture combinations result in reasonably good accuracy, and this applies to our search space. Therefore, in our next experiments, for the sake of simplicity and efficiency, to use *RS* as our search algorithm. However, the search spaces in *AutoML4ETC* can be combined with any other search algorithm to realize any other desiderata.

Evaluation of Child Model Training Strategies

The time complexity of our NAS approach is the total time spent in each of the following steps: (i) search algorithm, (ii) child architecture composition, and (iii) training of child

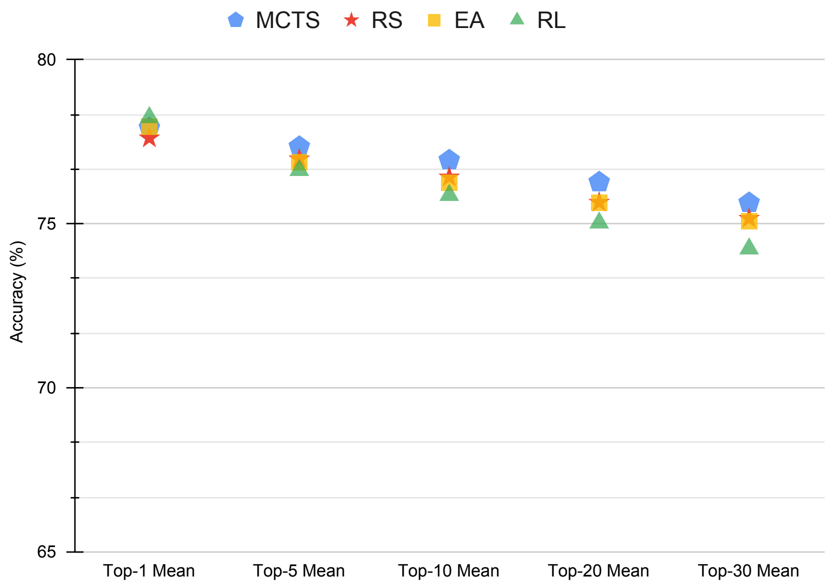


Figure 7.2: Comparison of average accuracy of top-N child models using different search algorithms

models. In our experiments the first and second steps take 0.1 seconds long on average using the RS algorithm. This means that the most time-consuming step of our NAS approach is the third step and the other parts have a negligible performance impact. Every time a child architecture is composed, it is trained over several epochs and then tested to measure its performance. Therefore, our goal is to reduce the child model training time (i.e., number of epochs) as much as possible without compromising the performance of the NAS.

In this section, we investigate the impact of different child model training strategies, i.e., *full training* versus *partial training*. Full training implies training child models on as many epochs as needed to generate the best possible architecture. With *partial training*, we train child models on a smaller number of epochs.

For this, we start by searching for an upper bound on the number of training epochs needed to find the best child model. To find this number, we conduct a set of experiments where we increase the number of training epochs starting from 10 and validate every 10 epochs on the testing dataset. Figure 7.3 is the mean validation accuracy of the top-10 child models for a varying number of training epochs. We can see that the average accuracy of the top-10 child models (i.e., black line curve) flattens beyond 40 epochs. Therefore, 40 epochs can be set as our upper bound for the full training of child models.

Table 7.3: Partial training of child models (10 epochs) vs full training (40 epochs) during the searching time in the packet raw bytes search space

Child model training epochs	Child model accuracy (%)	Full train accuracy (%)	Total parameters
10 epochs	77.61	79.72	263,368
40 epochs	82.86	-	111,560

From another perspective, if we partially train the child models over 10 epochs only, we can save approximately 75% of the total NAS time. With this method we just use 10 epochs for training child models, then extract the top child model and train it for extra 30 epochs. However, the best child model resulting from partial training is an estimate of the global best child model.

Table 7.3 further shows the trade-off between the accuracy and complexity of the top child model. We can see that with a 10-epoch partial training strategy, the top child model achieves 79.71% accuracy. However, with the full training strategy, i.e., training over 40 epochs, the top child model can achieve a higher accuracy of 82.86%. Additionally, the number of parameters of the top child model resulting from the full training approach is less than half of its counterpart. Therefore, the trade-off can be diluted down to a loss of ~3% in accuracy with twice as many parameters for a ~75% lower time complexity (i.e., search time).

Since the *AutoML4ETC* for *packet raw bytes* models are lighter in general (i.e., number of parameters) and the difference in accuracy is noticeable, a 40 epoch full training strategy seems to be a better option over this particular search space.

AutoML4ETC versus State-of-the-art

In the previous sections, we concluded that using *RS* as the search algorithm and 40 epochs for child model training would be our choices for the *packet raw bytes* search space. In this section, we compare the AutoML4ETC-generated model to other state-of-the-art architectures.

The first state-of-the-art model is UC Davis CNN [42] and the second is UW-H [4]. We use the same batching size for all models and also use the [4] input features (i.e., TLS handshake header) for our *AutoML4ETC* method.

Table 7.4 presents the performance of the *AutoML4ETC* and other state-of-the-art ETC models. It is evident that the *AutoML4ETC* approach outperforms the state-of-the-art models across all the datasets. In fact, the *AutoML4ETC*-generated model is ∼1 to 5% more accurate than state-of-the-art. Moreover, it is simpler and lighter, with over 50 times fewer parameters than state-of-the-art models on average.

Another insight is that the number of parameters of the *AutoML4ETC* model decreases as the dataset size decreases. This means that *AutoML4ETC* detects that a simpler model (in terms of parameters) is more appropriate for smaller datasets to achieve better results. The reason would be that with a complex model and small dataset, the model would not have enough data to learn the best parameter values.

To have a better understanding of the effect of the number of parameters, we report the computing power of each of these models across different datasets in Table 7.5. The number of FLOPS (in millions) for *AutoML4ETC* models is 1.46 to 2.53 times less than the state-of-the-art models. Specifically, the range of FLOPS for *AutoML4ETC* models is 263.01 to 389.04 whereas, the range for the state-of-the-art models is 568.26 to 666.1. Additionally, we compare the two major operations (in terms of computing) in these neural architectures, which are convolution and matrix multiplication (e.g., dense layers in neural networks). We can see that the comparison between the total number of FLOPS for the *AutoML4ETC* models and state-of-the-art models roughly holds for the convolution operations as well. However, the matrix multiplication for the *AutoML4ETC* models are on average 6,815 times less than the state-of-the-art models. We also compared the prediction time of the *AutoML4ETC* models and the state-of-the-art models in terms of average milliseconds per batch. We can see that GPU processing time for the *AutoML4ETC* models is roughly half of the state-of-the-art models, which makes them faster in terms of total prediction time. Additionally, the GPU processing time is almost the same regardless of the dataset for the *AutoML4ETC* model as opposed to the state-of-the-art models. The reason is that the *AutoML4ETC* tunes the child model's parameters based on the size and the inherent characteristics of the dataset, which makes it more stable. For these experiments, we used a computer with A40 GPU (see Appendix .5). Moreover, the GPU processing time is averaged across multiple runs.

7.3 Flow Time-series-oriented NAS

In this section, we evaluate the search space and the resulting model generated by AutoML4ETC for flow time-series data. We evaluate the search spaces by evaluating the best model generated when searching each search space by the same search strategy. We also compare the best models generated when partial versus full training is employed as child model training strategy.

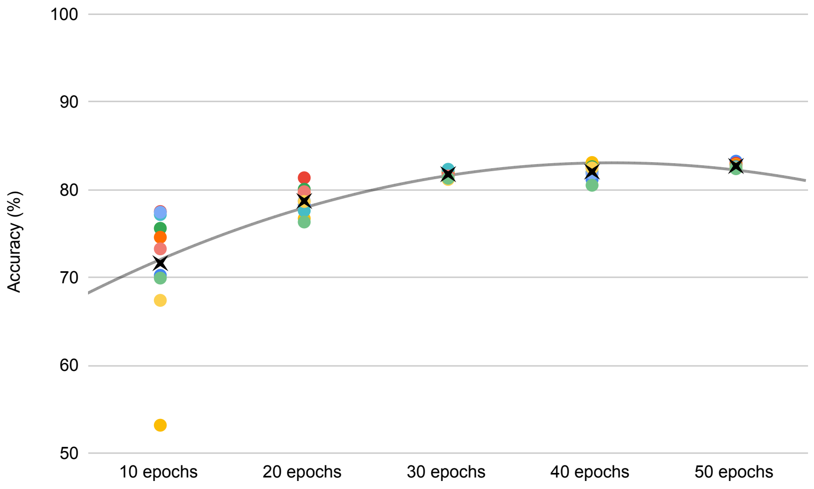


Figure 7.3: Accuracy of top 10 child models with different training epochs; the × mark is the average for each epoch.

Search Space Evaluation

Contrary to the search space for packet raw bytes, we only use model accuracy to evaluate the search space for flow time-series data. Flow time-series features summarize each packet in the flow in two numbers, signed (i.e., +/- for the incoming/outgoing direction) packet size and inter-arrival time. Therefore, they are not informative if the flow is cut to the first few packets and fast detection is not an evaluation metric.

Table 7.6 compares the AutoML4ETC flow time-series search spaces by comparing the accuracies of their best models. To obtain these accuracies, we fixed the number of trials to 200, the search algorithm to baseline *RS*, and used partial training as the child model training strategy (as discussed in the next section). The models are evaluated on the *May 2021* dataset.

Partial versus Full training of Child Models

Training the child models in the AutoML4ETC flow time-series search space is computationally expensive because they are LSTM-based. LSTM-based neural networks are generally more time-consuming to train than training MLP or CNN-based networks since they have a significantly higher number of parameters. AutoML4ETC time-series search

Table 7.4: AutoML4ETC versus UC Davis CNN [42] and UW-H [4] for the *packet raw bytes search space*

Dataset	Model	Accuracy (%)	W Avg. F-1 score (%)	W Avg. recall (%)	W Avg. precision (%)	Total parameters	Trainable parameters
July 2019	**AutoML4ETC**	**95.99**	**95.98**	**95.99**	**96**	**182,984**	**179,656**
	UW-H	94.87	94.87	94.87	94.93	7,588,360	7,588,360
	UC Davis CNN	90.95	90.92	90.95	90.93	6,507,016	6,507,016
April 2021	**AutoML4ETC**	**86.21**	**86.89**	**86.21**	**89.4**	**121,544**	**118,984**
	UW-H	84.59	86.62	84.59	90.95	-	-
	UC Davis CNN	82.17	82.3	82.17	82.87	-	-
May 2021	**AutoML4ETC**	**82.86**	**84.03**	**82.85**	**88.2**	**111,560**	**109,256**
	UW-H	79	80	79	83.86	-	-
	UC Davis CNN	79.29	79.38	79.29	79.56	-	-

Table 7.5: AutoML4ETC vs state-of-the-art [4, 42] in terms of computing operations for the packet raw bytes search space

Dataset	Model	Total params	MATmul FLOPS (million)	Convolution FLOPS (million)	Total FLOPS (million)	ms/batch (GPU)
July 2019	**AutoML4ETC**	**179,656**	**0.002**	**370.71**	**389.04**	**8**
	UW-H	7,588,360	14.71	649.67	666.1	15
	UC Davis CNN	6,507,016	12.55	554.24	568.26	14
April 2021	**AutoML4ETC**	**118,984**	**0.002**	**282.24**	**296.53**	**8**
	UW-H	-	-	-	-	14
	UC Davis CNN	-	-	-	-	19
May 2021	**AutoML4ETC**	**109,256**	**0.002**	**252.75**	**263.01**	**7**
	UW-H	-	-	-	-	5
	UC Davis CNN	-	-	-	-	19

space contains models with parallel Stacked LSTM layers, therefore, the corresponding models are even more time consuming to train than simple LSTMs.

We found that 30 epochs were enough for the full training of the child models on the AutoML4ETC flow time-series search space. The number of epochs is lower than that of the AutoML4ETC packet raw bytes search space because LSTM-based models are more susceptible to overfitting than CNN or MLP-based models. Aiming for a lower computation time for the AutoML4ETC time-series search space, we took a partial training strategy in which the child models are only trained for 10 epochs, and only the best model is trained for 20 more epochs for accuracy evaluation. We call 10-epoch training sessions partial training and 30-epoch sessions full training. Table 7.7 shows the accuracy of the best model when all child models are fully trained versus when the partial training strategy was employed. We see in partial training that the accuracy of the resulting model is only 0.09% less than in full training while computation time is reduced by threefold.

Table 7.6: *Flow time-series* search spaces comparison

Search Space	Search Algorithm	Trials	Child model Accuracy (%)	Full train Accuracy (%)
AutoML4ETC	**RS**	**200**	**85.25**	**88.27**
ENAS micro	RS	200	77.4	83.02
AutoML4ETC [packet raw bytes]	RS	200	52.34	64.04
SLSTM + MLP	RS	200	82.99	86.9

Table 7.7: Partial training of child models (10 epochs) vs full training (30 epochs) during the searching time in the flow time-series search space

Child model training epochs	Child model accuracy (%)	Full train accuracy (%)
10 epochs	85.25	88.27
30 epochs	88.36	-

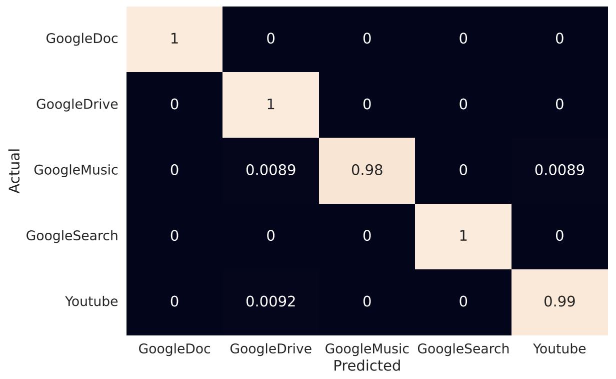
AutoML4ETC versus State-of-the-art

In this section, we compare the performance of the model generated by *AutoML4ETC* to the state-of-the-art UW-F model [4] on the *flow time-series* data. The UW-F model [4] achieves higher accuracy than other notable works [43, 42]. For these comparisons, we use 200 trials, the child models partial training strategy, and *RS* as the search algorithm.

Table 7.8 reveals the significantly higher performance of the *AutoML4ETC* models compared to the UW-F across all datasets and metrics. Interestingly, over some datasets, the difference in performance exceeds 10%.

In more detail, the UW-F model does not converge on the *April 2021* dataset and achieves only 0.05% accuracy, whereas the *AutoML4ETC* achieves 90.52% accuracy. Moreover, on the *QUIC May 2021* dataset UW-F results in 44.9% accuracy, whereas the *AutoML4ETC* finds a model with 89.7% accuracy. For the other datasets the gap between the accuracy of the *AutoML4ETC* UW-F models is in the range of ∼3-10%. This significant difference and the fact that the state-of-the-art model may not even converge on some datasets, promotes the significance of the *AutoML4ETC* automatic method.

Lastly, we also compare the performance of the *AutoML4ETC* to the UW-F model on the *QUIC - UCDavis* dataset. It is evident that although the UW-F model achieves high accuracy of 99.4% with 20 epochs, the *AutoML4ETC* can achieve an even higher accuracy of 99.7% with just 10 epochs, which is 0.5% higher than UW-F. Note that, the best model



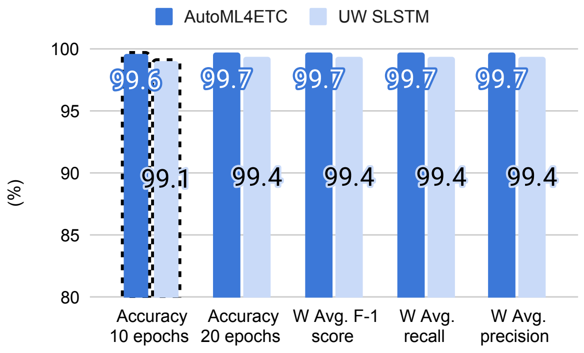


Figure 7.4: Confusion matrix for the *AutoML4ETC* (left) and the performance metrics for *AutoML4ETC* vs UW-F [4] (right) on the *QUIC - UCDavis* dataset. The dotted border is only for 10 epochs, whereas the rest is for 20.

Table 7.8: AutoML4ETC vs UW-F [4] for the *flow time-series search space*

Dataset	Model	Accuracy (%)	W Avg. F-1 score (%)	W Avg. recall (%)	W Avg. precision (%)
July 2019	**AutoML4ETC**	**89.31**	**89.35**	**89.31**	**89.46**
	UW-F	86.32	86.4	86.32	86.64
April 2021	**AutoML4ETC**	**90.52**	**90.54**	**90.52**	**90.63**
	UW-F	0.05	0.005	0.052	0.002
May 2021	**AutoML4ETC**	**88.27**	**88.31**	**88.27**	**88.58**
	UW-F	81	81.05	81	81.8
QUIC May 2021	**AutoML4ETC**	**96.25**	**96.3**	**96.25**	**96.36**
	UW-F	86.79	88.84	86.79	93
QUIC April 2021	**AutoML4ETC**	**89.7**	**91.87**	**89.7**	**94.98**
	UW-F	44.99	58.34	44.99	88.82

and method for *QUIC - UCDavis* dataset in the original paper [43] can achieve the highest accuracy of 98%. Detailed results of the performance metrics and the confusion matrix for this dataset are shown in Figure 7.4.

7.4 Combining AutoML4ETC Models

Across Search Spaces

AutoML4ETC can find the best model for both *flow time-series* and *packet raw bytes*. In this part, we want to combine these two powerful models to increase the overall classification accuracy (i.e., build a hybrid model). The intuition behind this is that for some samples and classes where the *packet raw bytes* are not doing well (i.e., have low accuracy), the *flow time-series* may do better and help boost the classification accuracy.

For this, we take the best *flow time-series* and *packet raw bytes* models and freeze the learned weights in the neural network. Each of these models produces a probability per class in their last layer (i.e., Softmax layer). Therefore, we only need to add a simple method such as *logistic regression* to learn how to best combine these probabilities. In our approach, we use a thin layer of MLP. We concatenate the output probabilities of the *AutoML4ETC* models, then add a single layer of MLP with 11 units (i.e., 2/3 number of inputs) followed by ReLU activation layer. The output is connected to a final softmax layer and trained for up to 20 epochs with early stopping and patience of 3.

Figure 7.5a depicts the performance difference of the combined *AutoML4ETC* approach versus the UW Tripartite model [4]. Both of the models use the same input features for the *flow time-series* and *packet raw bytes*. Additionally, UW Tripartite model also uses *flow statistical* features as the third input feature. It is evident that combining with only two feature sets is outperforming the UW Tripartite model over all of the TLS datasets with a difference in accuracy ranging from 0.27-49.81%. This significant difference shows the effectiveness of the simple combining method over the more complicated UW Tripartite model dense layers [4].

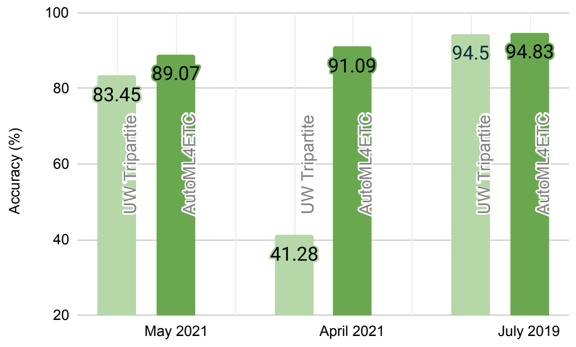
As an example, the combined model of discovered neural architectures by *AutoML4ETC* along with its details is shown in Appendix Figure 7.6 for the *May 2021* dataset.

Within a Search Space

In ML it is often believed that instead of using the most powerful classifier (i.e., with the highest accuracy), an ensemble of weaker classifiers can result in higher classification accuracy. For instance, Random Forest algorithm is an ensemble of Decision Trees (DT).

The same is also applicable to DNNs. However, the ensembling approach is often overlooked in DNNs since training a single DNN is a more time-consuming task than a classical ML algorithm, such as DT. With the *AutoML4ETC* we already have multiple child models trained on our target dataset.

We use the *packet raw bytes* models as we already have executed the full training of child models strategy during searching. Furthermore, we use the last 4 child models that have shown increasing validation accuracy and combine these models with the previous section method (i.e., single layer MLP). The use of last 4 increasing instead of top 4 child models is to simulate the weaker classifier condition of ensembling approaches. Meaning



(a) Hybrid *AutoML4ETC* models' performance vs the *UW Tripartite* model [4]

Dataset	Ensembled	Accuracy (%)	W Avg. F-1 score (%)	Total parameters
May 2021	×	82.86	84.03	111,560
	✓	**83.85**	**85.05**	**505,733**
April 2021	×	86.21	86.89	121,544
	✓	**86.86**	**87.53**	**620,933**

(b) Ensembling approach for the *AutoML4ETC* models in increasing accuracy order

Figure 7.5: Combining AutoML4ETC models results

that the first three classifiers have lower accuracy than the fourth one which is the best child model.

Figure 7.5b shows the results for this ensembling approach. It is evident that the ensembling approach indeed helps to boost the performance from ~ 0.6 to 1% accuracy. Additionally, the number of parameters increases by around ~ 4.5 to 5 times more than the best child model. Therefore, there is a trade-off between achieving higher accuracy and the number of parameters that could be increased by using more than 4 child models. We omitted the *July 2019* dataset as *AutoML4ETC* found the best child model on the first trial.

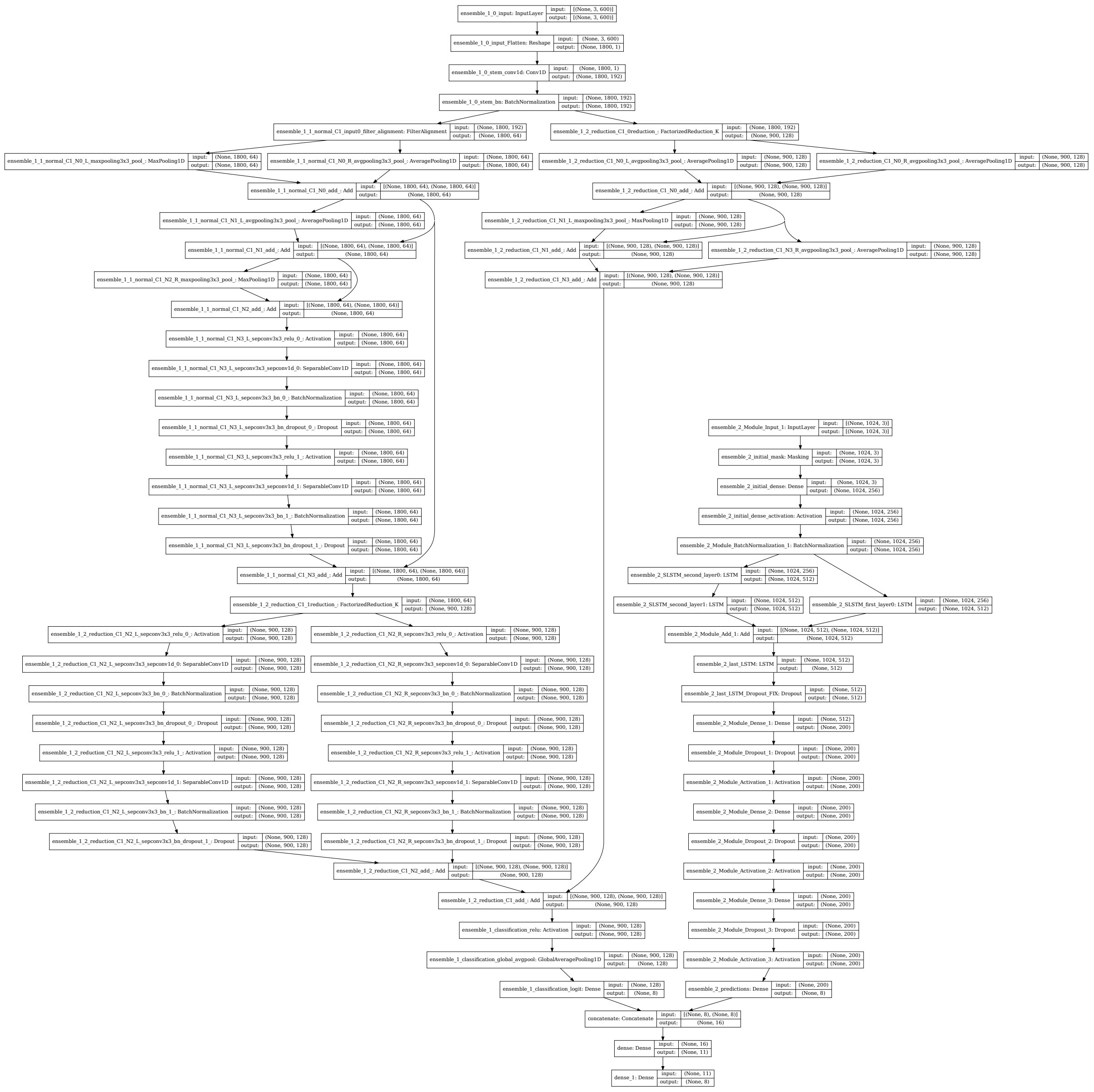

ensemble_1_0_input: InputLayer | input: [(None, 3, 600)] | output: [(None, 3, 600)]
ensemble_1_0_input_Flatten: Reshape | input: (None, 3, 600) | output: (None, 1800, 1)
ensemble_1_0_stem_conv1d: Conv1D | input: (None, 1800, 1) | output: (None, 1800, 192)
ensemble_1_0_stem_bn: BatchNormalization | input: (None, 1800, 192) | output: (None, 1800, 192)
ensemble_1_1_normal_C1_input0_filter_alignment: FilterAlignment | input: (None, 1800, 192) | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_0reduction_: FactorizedReduction_K | input: (None, 1800, 192) | output: (None, 900, 128)
ensemble_1_1_normal_C1_N0_L_maxpooling3x3_pool_: MaxPooling1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N0_R_avgpooling3x3_pool_: AveragePooling1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_N0_L_avgpooling3x3_pool_: AveragePooling1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N0_R_avgpooling3x3_pool_: AveragePooling1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_1_normal_C1_N0_add_: Add | input: [(None, 1800, 64), (None, 1800, 64)] | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_N0_add_: Add | input: [(None, 900, 128), (None, 900, 128)] | output: (None, 900, 128)
ensemble_1_1_normal_C1_N1_L_avgpooling3x3_pool_: AveragePooling1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_N1_L_maxpooling3x3_pool_: MaxPooling1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_1_normal_C1_N1_add_: Add | input: [(None, 1800, 64), (None, 1800, 64)] | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_N1_add_: Add | input: [(None, 900, 128), (None, 900, 128)] | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N3_R_avgpooling3x3_pool_: AveragePooling1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_1_normal_C1_N2_R_maxpooling3x3_pool_: MaxPooling1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_N3_add_: Add | input: [(None, 900, 128), (None, 900, 128)] | output: (None, 900, 128)
ensemble_1_1_normal_C1_N2_add_: Add | input: [(None, 1800, 64), (None, 1800, 64)] | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_relu_0_: Activation | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_sepconv1d_0: SeparableConv1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_bn_0_: BatchNormalization | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_bn_dropout_0_: Dropout | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_relu_1_: Activation | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_sepconv1d_1: SeparableConv1D | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_bn_1_: BatchNormalization | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_L_sepconv3x3_bn_dropout_1_: Dropout | input: (None, 1800, 64) | output: (None, 1800, 64)
ensemble_1_1_normal_C1_N3_add_: Add | input: [(None, 1800, 64), (None, 1800, 64)] | output: (None, 1800, 64)
ensemble_1_2_reduction_C1_1reduction_: FactorizedReduction_K | input: (None, 1800, 64) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_relu_0_: Activation | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_relu_0_: Activation | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_sepconv1d_0: SeparableConv1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_sepconv1d_0: SeparableConv1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_bn_0_: BatchNormalization | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_bn_0_: BatchNormalization | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_bn_dropout_0_: Dropout | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_bn_dropout_0_: Dropout | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_relu_1_: Activation | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_relu_1_: Activation | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_sepconv1d_1: SeparableConv1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_sepconv1d_1: SeparableConv1D | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_bn_1_: BatchNormalization | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_bn_1_: BatchNormalization | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_L_sepconv3x3_bn_dropout_1_: Dropout | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_R_sepconv3x3_bn_dropout_1_: Dropout | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_2_reduction_C1_N2_add_: Add | input: [(None, 900, 128), (None, 900, 128)] | output: (None, 900, 128)
ensemble_1_2_reduction_C1_add_: Add | input: [(None, 900, 128), (None, 900, 128)] | output: (None, 900, 128)
ensemble_1_classification_relu: Activation | input: (None, 900, 128) | output: (None, 900, 128)
ensemble_1_classification_global_avgpool: GlobalAveragePooling1D | input: (None, 900, 128) | output: (None, 128)
ensemble_1_classification_logit: Dense | input: (None, 128) | output: (None, 8)
ensemble_2_Module_Input_1: InputLayer | input: [(None, 1024, 3)] | output: [(None, 1024, 3)]
ensemble_2_initial_mask: Masking | input: (None, 1024, 3) | output: (None, 1024, 3)
ensemble_2_initial_dense: Dense | input: (None, 1024, 3) | output: (None, 1024, 256)
ensemble_2_initial_dense_activation: Activation | input: (None, 1024, 256) | output: (None, 1024, 256)
ensemble_2_Module_BatchNormalization_1: BatchNormalization | input: (None, 1024, 256) | output: (None, 1024, 256)
ensemble_2_SLSTM_second_layer0: LSTM | input: (None, 1024, 256) | output: (None, 1024, 512)
ensemble_2_SLSTM_second_layer1: LSTM | input: (None, 1024, 512) | output: (None, 1024, 512)
ensemble_2_SLSTM_first_layer0: LSTM | input: (None, 1024, 256) | output: (None, 1024, 512)
ensemble_2_Module_Add_1: Add | input: [(None, 1024, 512), (None, 1024, 512)] | output: (None, 1024, 512)
ensemble_2_last_LSTM: LSTM | input: (None, 1024, 512) | output: (None, 512)
ensemble_2_last_LSTM_Dropout_FIX: Dropout | input: (None, 512) | output: (None, 512)
ensemble_2_Module_Dense_1: Dense | input: (None, 512) | output: (None, 200)
ensemble_2_Module_Dropout_1: Dropout | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Activation_1: Activation | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Dense_2: Dense | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Dropout_2: Dropout | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Activation_2: Activation | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Dense_3: Dense | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Dropout_3: Dropout | input: (None, 200) | output: (None, 200)
ensemble_2_Module_Activation_3: Activation | input: (None, 200) | output: (None, 200)
ensemble_2_predictions: Dense | input: (None, 200) | output: (None, 8)
concatenate: Concatenate | input: [(None, 8), (None, 8)] | output: (None, 16)
dense: Dense | input: (None, 16) | output: (None, 11)
dense_1: Dense | input: (None, 11) | output: (None, 8)

Figure 7.6: Example of hybrid model from combining the *packet raw bytes* and *flow time-series* search spaces, generated by *AutoML4ETC* on the *May 2021* dataset

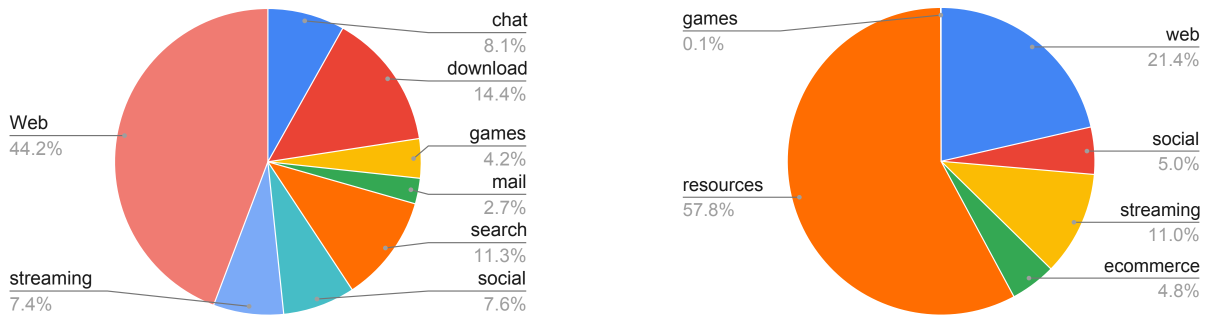


Figure 7.7: Measurement estimation on the *unknown* part of the TLS (left) and QUIC (right) datasets captured in May 2021

7.5 Traffic Measurement

One of the end goals of having an Encrypted Traffic Classifier is to estimate the volume and distribution of *unknown* traffic in the network. In other words, we build an approximate look-up table to label a portion of the real-world collected dataset. This labeling module is just capable of labeling a part of a dataset (i.e., *SNI* to service mapping is easy for humans); hence, a major portion of encrypted data is still unlabeled and referred to as *unknown*.

Figure 7.7 shows the flow distribution of the *unknown* traffic for *May 2021* (TLS) and *QUIC - May 2021* datasets. For this experiment we used the best models found by the *AutoML4ETC*. For the TLS dataset, we can see that nearly half of the flows (i.e., 44.2%) are related to the web class. However, the *web* class contains around half of this number for the QUIC dataset (i.e., 21.4%). Furthermore, we can see that nearly half of the flows for the QUIC dataset (i.e., 57.8%) are related to the *resources* class. Interestingly, the *social, streaming* classes have a rather similar percentage for both the TLS (i.e., 7.6, 7.4%) and QUIC datasets (i.e., 5.0, 11%). Overally, by comparing the traffic measurements between TLS and QUIC datasets captured at the same time, it can be inferred that still QUIC has a long way to be adopted for majority of services other than *resource* delivery.

Chapter 8

Conclusion & Future Work

In this thesis, we investigated the effect of data drift on two state-of-the-art deep encrypted traffic classification models. We examined the robustness of these models to data drift, providing insights about the type of drift that occurs in network traffic data. We showed that a model that operates on the traffic shape is more resilient to data drift than one that operates on TLS headers. Also, we examined the impact of model architecture and feature engineering on model robustness by comparing the two models over the same datasets.

We investigated how model architectures are affected by data drift using confusion matrices for both UW-H and UCDavis CNN models. Furthermore, we presented the top-k accuracy and its logarithmic weighted mean to measure the amount of confusion due to data drift amid service classes when training and test datasets are close (i.e., 2 months) and far (i.e., 2 years) apart. We also analysed the contribution of each application towards performance drop over a long period among different service classes. We examined the impact of the application-layer protocols on model robustness, demonstrating that the model performance improves by selecting more stable protocols (e.g., HTTP/1, HTTP/2) for the model to train on, regardless of dataset collection time.

To warrant the need for architectural adaptations, we showcased the performance and convergence issues that arise when a state-of-the-art model is trained on different datasets with no adaptations. We performed an ablation study and examined the performance of decomposed models, as well as the effect of changing structural parameters, to propose best practices for designing an architecture that performs well on unseen and possibly newer datasets. We showed results for application-level and service-level classification to highlight generalizability of proposed adaptions at different levels of labeling granularity. We also showed the generalizability of our proposed guidelines to different encryption protocols by testing the adapted architecture on a dataset of QUIC traffic for service- and application-level classification, which resulted in up to 9% higher classification accuracy than the default model without adaptations.

The adaptation approaches proposed in the first part of the thesis are manual. To automate this approach we introduced *AutoML4ETC*, a fully automated and efficient method for NAS in Encrypted Traffic Classification. We designed multiple search spaces tailored specifically for the well-known feature sets in Encrypted Traffic Classification based on both Encrypted Traffic Classification literature and other domains such as image classification. Furthermore, we evaluated the effectiveness of our search space with a simple *RS* search algorithm. We showed that the found model outperforms the expert-designed state-of-the-art Encrypted Traffic Classification models in terms of accuracy while having fewer parameters. We showed that *AutoML4ETC* models' performance can be boosted by using simple ensembling techniques. Lastly, we measured the adoption of the QUIC protocol in comparison with TLS on our real-world datasets using the Encrypted Traffic Classification model.

There are three main future directions for this work. The first is to focus on improving the search spaces. As an example, Attention Networks [55] can be incorporated into the flow time-series search space with LSTM or CNN layers as recently they shown promising results in the NLP domain. The second direction is to make the overall process efficient, for example by using one-shot approaches [11] or transfer learning [10]. The last direction is to improve the generalizability of the classifier by using incremental learning methods that leverage previously learned knowledge, both to reduce training time and increase performance on new datasets.

References

[1] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (ntma): a survey. *Computer Communications*, 170:19–41, 2021.

[3] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019.

[4] Iman Akbari, Mohammad A. Salahuddin, Leni Ven, Noura Limam, Raouf Boutaba, Bertrand Mathieu, Stephanie Moteau, and Stephane Tuffin. A look behind the curtain: Traffic classification in an increasingly encrypted web. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(1), 2021.

[5] Riyad Alshammari and A Nur Zincir-Heywood. Can encrypted traffic be identified without port numbers, ip addresses and payload inspection? *Computer networks*, 55(6):1326–1350, 2011.

[6] Giuseppina Andresini, Feargus Pendlebury, Fabio Pierazzi, Corrado Loglisci, Annalisa Appice, and Lorenzo Cavallaro. Insomnia: Towards concept-drift robustness in network intrusion detection. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 111–122, 2021.

[7] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[8] Carlos Bacquet, A Nur Zincir-Heywood, and Malcolm I Heywood. Genetic optimization and hierarchical clustering applied to encrypted traffic identification. In *2011 IEEE symposium on computational intelligence in cyber security (CICS)*, pages 194–201. IEEE, 2011.

[9] Roni Bar-Yanai, Michael Langberg, David Peleg, and Liam Roditty. Realtime classification for encrypted traffic. In *International Symposium on Experimental Algorithms*, pages 373–385. Springer, 2010.

[10] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine learning*, 79(1-2):151–175, 2010.

[11] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

[12] Laurent Bernaille and Renata Teixeira. Early recognition of encrypted applications. In *International Conference on Passive and Active Network Measurement*, pages 165–175. Springer, 2007.

[13] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

[14] Zhitang Chen, Ke He, Jian Li, and Yanhui Geng. Seq2img: A sequence-to-image based approach towards ip traffic classification using convolutional neural networks. In *IEEE International conference on big data (big data)*, pages 1271–1276, 2017.

[15] François Chollet et al. Keras. https://keras.io, 2015.

[16] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[17] Gints Engelen, Vera Rimmer, and Wouter Joosen. Troubleshooting an intrusion detection dataset: the cicids2017 case study. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 7–12. IEEE, 2021.

[18] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42, 2009.

[21] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International conference on artificial neural networks*, pages 87–94. Springer, 2001.

[22] Didier Frank Isingizwe, Meng Wang, Wenmao Liu, Dongsheng Wang, Tiejun Wu, and Jun Li. Analyzing learning-based encrypted malware traffic classification with automl. In *IEEE International Conference on Communication Technology (ICCT)*, pages 313–322, 2021.

[23] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 263–274, 2014.

[24] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, 2017.

[25] Amirhossein Khajehpour, Farid Zandi, Navid Malekghaini, Mahdi Hemmatyar, Naeimeh Omidvar, and Mahdi Jafari Siavoshani. Deep inside tor: Exploring website fingerprinting attacks on tor traffic in realistic settings. In *2022 12th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 148–156, 2022.

[26] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[27] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[28] Yuichi Kumano, Shingo Ata, Nobuyuki Nakamura, Yoshihiro Nakahira, and Ikuo Oka. Towards real-time processing for application identification of encrypted traffic. In *2014*

International Conference on Computing, Networking and Communications (ICNC), pages 136–140. IEEE, 2014.

[29] Arash Habibi Lashkari. CICFlowMeter. https://github.com/ahlashkari/CICFlowMeter, 2022. [Online; GitHub Repository].

[30] Arash Habibi Lashkari, Gerard Draper-Gil, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of tor traffic using time based features. In *ICISSp*, pages 253–262, 2017.

[31] Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. Deep packet: A novel approach for encrypted traffic classification using deep learning. *Soft Computing*, 24(3):1999–2012, 2020.

[32] Bei Lu, Nurbol Luktarhan, Chao Ding, and Wenhui Zhang. Iclstm: Encrypted traffic service identification based on inception-lstm neural network. *Symmetry*, 13:1080, 06 2021.

[33] Renjian Lyu, Mingshu He, Yu Zhang, Lei Jin, and Xinlei Wang. Network intrusion detection based on an efficient neural architecture search. *Symmetry*, 13(8):1453, 2021.

[34] Minghua Ma, Shenglin Zhang, Dan Pei, Xin Huang, and Hongwei Dai. Robust and rapid adaption for concept drift in software system anomaly detection. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 13–24. IEEE, 2018.

[35] Navid Malekghaini, Elham Akbari, Mohammad A. Salahuddin, Noura Limam, Raouf Boutaba, Bertrand Mathieu, Stephanie Moteau, and Stephane Tuffin. Data drift in dl: Lessons learned from encrypted traffic classification. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[36] Navid Malekghaini and Iman Akbari. DeepTrafficV2. shorturl.at/cfglD, 2022. [Online; Private GitHub Repository].

[37] M. Mehner. Whatsapp, wechat and meta messenger apps - global usage of messaging apps, penetration and statistics. Accessed Oct. 14, 2022.

[38] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[39] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[40] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017.

[41] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-14, Internet Engineering Task Force, February 2022. Work in Progress.

[42] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. Large-scale mobile app identification using deep learning. *IEEE Access*, 8:348–362, 2020.

[43] Shahbaz Rezaei and Xin Liu. How to achieve high classification accuracy with just a few labels: A semi-supervised approach using sampled packets. 2018.

[44] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *arXiv preprint arXiv:1708.06376*, 2017.

[45] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *arXiv preprint arXiv:1708.06376*, 2017.

[46] Sebastian Ruder. An overview of multi-task learning in deep neural networks, 2017.

[47] Sakti Saurav, Pankaj Malhotra, Vishnu TV, Narendhar Gugulothu, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. Online anomaly detection with concept drift adaptation using recurrent neural networks. In *Proceedings of the acm india joint international conference on data science and management of data*, pages 78–87, 2018.

[48] Tom Schaul and Jürgen Schmidhuber. Metalearning. *Scholarpedia*, 5(6):4650, 2010.

[49] Tal Shapira and Yuval Shavitt. Flowpic: Encrypted internet traffic classification is as easy as image recognition. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 680–687, 2019.

[50] Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A. Ghorbani. Developing realistic distributed denial of service (ddos) attack dataset and taxonomy.

In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8, 2019.

[51] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374, 2012.

[52] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers Security*, 31(3):357–374, 2012.

[53] Payap Sirinam, Nate Mathews, Mohammad Saidur Rahman, and Matthew Wright. Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1148, 2019.

[54] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust smartphone app identification via encrypted network traffic analysis. *IEEE Transactions on Information Forensics and Security*, 13(1):63–78, 2017.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[56] Petr Velan, Milan Čermák, Pavel Čeleda, and Martin Drašar. A survey of methods for encrypted traffic classification and analysis. *International Journal of Network Management*, 25(5):355–374, 2015.

[57] W3Techs. Historical yearly trends in the usage statistics of site elements for websites. Accessed Feb. 2022.

[58] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. Neural architecture search using deep neural networks and monte carlo tree search. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 9983–9991, 2020.

[59] Tao Wang and Ian Goldberg. On realistically attacking tor with website fingerprinting. *Proc. Priv. Enhancing Technol.*, 2016(4):21–36, 2016.

[60] Xiaojuan Wang, Xinlei Wang, Lei Jin, Renjian Lv, Bingying Dai, Mingshu He, and Tianqi Lv. Evolutionary algorithm-based and network architecture search-enabled multiobjective traffic classification. *IEEE Access*, 9:52310–52325, 2021.

[61] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[62] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[63] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[64] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

APPENDICES

.1 Flow-based statistics from CICFlowMeter [29]

- Duration of the flow in Microsecond
- Duration of the flow in Microsecond
- Total packets in the forward direction
- Total packets in the backward direction
- Total size of packet in forward direction
- Total size of packet in backward direction
- Minimum size of packet in forward direction
- Maximum size of packet in forward direction
- Mean size of packet in forward direction
- Standard deviation size of packet in forward direction
- Minimum size of packet in backward direction
- Maximum size of packet in backward direction
- Mean size of packet in backward direction
- Standard deviation size of packet in backward direction
- Number of flow packets per second
- Number of flow bytes per second
- Mean time between two packets sent in the flow
- Standard deviation time between two packets sent in the flow
- Maximum time between two packets sent in the flow
- Minimum time between two packets sent in the flow
- Minimum time between two packets sent in the forward direction
- Maximum time between two packets sent in the forward direction
- Mean time between two packets sent in the forward direction
- Standard deviation time between two packets sent in the forward direction

- Total time between two packets sent in the forward direction
- Minimum time between two packets sent in the backward direction
- Maximum time between two packets sent in the backward direction
- Mean time between two packets sent in the backward direction
- Standard deviation time between two packets sent in the backward direction
- Total time between two packets sent in the backward direction
- Number of times the PSH flag was set in packets travelling in the forward direction (0 for UDP)
- Number of times the PSH flag was set in packets travelling in the backward direction (0 for UDP)
- Number of times the URG flag was set in packets travelling in the forward direction (0 for UDP)
- Number of times the URG flag was set in packets travelling in the backward direction (0 for UDP)
- Total bytes used for headers in the forward direction
- Total bytes used for headers in the backward direction
- Number of forward packets per second
- Number of backward packets per second
- Minimum length of a packet
- Maximum length of a packet
- Mean length of a packet
- Standard deviation length of a packet
- Variance length of a packet
- Number of packets with FIN
- Number of packets with SYN
- Number of packets with RST
- Number of packets with PUSH
- Number of packets with ACK
- Number of packets with URG
- Number of packets with CWE
- Number of packets with ECE
- Download and upload ratio
- Average size of packet
- Average size observed in the forward direction
- Average number of bytes bulk rate in the forward direction
- Length of header for forward packet
- Average number of bytes bulk rate in the forward direction
- Average number of packets bulk rate in the forward direction
- Average number of bulk rate in the forward direction
- Average number of bytes bulk rate in the backward direction

- Average number of packets bulk rate in the backward direction
- Average number of bulk rate in the backward direction
- The average number of packets in a sub flow in the forward direction
- The average number of bytes in a sub flow in the forward direction
- The average number of packets in a sub flow in the backward direction
- The average number of bytes in a sub flow in the backward direction
- The total number of bytes sent in initial window in the forward direction
- The total number of bytes sent in initial window in the backward direction
- Count of packets with at least 1 byte of TCP data payload in the forward direction
- Minimum segment size observed in the forward direction
- Minimum time a flow was active before becoming idle
- Mean time a flow was active before becoming idle
- Maximum time a flow was active before becoming idle
- Standard deviation time a flow was active before becoming idle
- Minimum time a flow was idle before becoming active
- Mean time a flow was idle before becoming active
- Maximum time a flow was idle before becoming active
- Standard deviation time a flow was idle before becoming active
- Total packets in the forward direction
- Total packets in the backward direction
- Total size of packet in forward direction
- Total size of packet in backward direction
- Minimum size of packet in forward direction
- Minimum size of packet in backward direction
- Maximum size of packet in forward direction
- Maximum size of packet in backward direction
- Mean size of packet in forward direction
- Mean size of packet in backward direction
- Standard deviation size of packet in forward direction
- Standard deviation size of packet in backward direction
- Total time between two packets sent in the forward direction
- Total time between two packets sent in the backward direction
- Minimum time between two packets sent in the forward direction
- Minimum time between two packets sent in the backward direction

- Maximum time between two packets sent in the forward direction
- Maximum time between two packets sent in the backward direction
- Mean time between two packets sent in the forward direction
- Mean time between two packets sent in the backward direction
- Standard deviation time between two packets sent in the forward direction
- Standard deviation time between two packets sent in the backward direction
- Number of times the PSH flag was set in packets travelling in the forward direction (0 for UDP)
- Number of times the PSH flag was set in packets travelling in the backward direction (0 for UDP)
- Number of times the URG flag was set in packets travelling in the forward direction (0 for UDP)
- Number of times the URG flag was set in packets travelling in the backward direction (0 for UDP)
- Total bytes used for headers in the forward direction
- Total bytes used for headers in the backward direction
- Number of forward packets per second
- Number of backward packets per second
- Number of flow packets per second
- Number of flow bytes per second
- Minimum length of a flow
- Maximum length of a flow
- Mean length of a flow
- Standard deviation length of a flow
- Minimum inter-arrival time of packet
- Maximum inter-arrival time of packet
- Mean inter-arrival time of packet
- Standard deviation inter-arrival time of packet
- Number of packets with FIN
- Number of packets with SYN
- Number of packets with RST
- Number of packets with PUSH
- Number of packets with ACK
- Number of packets with URG
- Number of packets with CWE
- Number of packets with ECE
- Download and upload ratio
- Average size of packet
- Average size observed in the forward direction
- Average number of bytes bulk rate in the forward direction
- Average number of packets bulk rate in the forward direction
- Average number of bulk rate in the forward direction

- Average size observed in the backward direction
- Average number of bytes bulk rate in the backward direction
- Average number of packets bulk rate in the backward direction
- Average number of bulk rate in the backward direction
- The average number of packets in a sub flow in the forward direction
- The average number of bytes in a sub flow in the forward direction
- The average number of packets in a sub flow in the backward direction
- The average number of bytes in a sub flow in the backward direction
- Minimum time a flow was active before becoming idle
- Mean time a flow was active before becoming idle
- Maximum time a flow was active before becoming idle
- Standard deviation time a flow was active before becoming idle
- Minimum time a flow was idle before becoming active
- Mean time a flow was idle before becoming active
- Maximum time a flow was idle before becoming active
- Standard deviation time a flow was idle before becoming active
- The total number of bytes sent in initial window in the forward direction
- The total number of bytes sent in initial window in the backward direction
- Count of packets with at least 1 byte of TCP data payload in the forward direction
- Minimum segment size observed in the forward direction

.2 UW Tripartite Model Details

Table 1: Architecture of the UW tripartite model with 1-D convolutions in the flow side from the [4].

Type	Shape	Connection
In	$(3, 600)$	Header Input
Reshape	$(1800, 1)$	
Convolution1D	$(1799, 256)$	
ReLU		
Convolution1D	$(1799, 256)$	
ReLU		
MaxPooling1D	$(899, 256)$	
Convolution1D	$(898, 128)$	
ReLU		
Convolution1D	$(897, 128)$	
ReLU		
MaxPooling1D	$(448, 128)$	
Flatten	(57344)	
In	(61)	Flow Meter Input
Dense	(200)	
BatchNorm		
LeakyReLU		
Dropout(0.2)		
Dense	(200)	
BatchNorm		
LeakyReLU		
Dropout(0.5)		
In	$(1024, 3)$	Flow Input
Convolution1D	$(1024, 128)$	
BatchNorm		
ELU		
Convolution1D	$(1024, 128)$	
BatchNorm		
ELU		
MaxPooling1D	$(512, 128)$	
Convolution1D	$(512, 64)$	
BatchNorm		
ELU		
Convolution1D	$(512, 64)$	
BatchNorm		
ELU		
MaxPooling1D	$(256, 64)$	
Flatten	(16384)	
Concatenate	(73928)	←
Dense	(128)	
LeakyReLU		
Dropout(0.5)		
Dense	(128)	
LeakyReLU		
Dropout(0.5)		
Dense	(8)	
Softmax		

Table 2: Architecture of the UW tripartite model with stacked LSTM's in the flow side from the [4].

Type	Shape	Connection
In	$(3, 600)$	Header Input
Reshape	$(1800, 1)$	
Convolution1D	$(1799, 256)$	
ReLU		
Convolution1D	$(1799, 256)$	
ReLU		
MaxPooling1D	$(899, 256)$	
Convolution1D	$(898, 128)$	
ReLU		
Convolution1D	$(897, 128)$	
ReLU		
MaxPooling1D	$(448, 128)$	
Flatten	(57344)	
In	(61)	Flow Meter Input
Dense	(200)	
BatchNorm		
LeakyReLU		
Dropout(0.2)		
Dense	(200)	
BatchNorm		
LeakyReLU		
Dropout(0.5)		
In	$(1024, 3)$	Flow Input
Dense(Distributed)	$(1024, 512)$	
BatchNorm		
LeakyReLU		
LSTM(Forward)	$(1024, 256)$	
LSTM(Backward)	$(1024, 256)$	
LSTM(Forward)	(256)	
Dropout(0.5)		
Concatenate	(57800)	←
Dense	(128)	
LeakyReLU		
Dropout(0.5)		
Dense	(128)	
LeakyReLU		
Dropout(0.5)		
Dense	(8)	
Softmax		

.3 Acronyms

Acronym	Meaning
ALPN	Application-Layer Protocol Negotiation
CNN	Convolutional Neural Network
DL	Deep Learning
DNS	Domain Name System
ISP	Internet Service Provider
LSTM	Long Short-term Memory
BLSTM	Biderctional LSTM
SLSTM	Stacked LSTM
ML	Machine Learning
MLP	Multi-layer Perceptron
WF	Website Fingerprinting
KPI	Key Performance Index
RNN	Recurrent Neural Network
UW	University of Waterloo Tripartite model
UW-H	TLS header part of UW
UW-F	flow time-series part of UW
UW-A	statistical part of UW
IP	Internet Protocol
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
PCAP	Packet Capture
LR	Learning Rate
CONV1D	1-D Convolutional Layer
HTTP	Hypertext Transfer Protocol
QUIC	Quick UDP Internet Connections
RF	Random Forest
SAE	Stacked Auto-encoder
SNI	Server Name Indication
SVM	Support Vector Machine
TC	Traffic Classification
ETC	Encrypted Traffic Classification
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

.4 DeepTraffic Software API Code Examples

DeepTrafficV2 is the name of the software API created as a result of the research presented in this thesis. The Deep Traffic GitHub [36] repository has the complete documentation for the code base. Here, we offer several code samples of the library's application for pre-processing network traces, and training models. The model modifications based on this thesis are included in the latest version of DeepTraffic. Additionally, it offers the classification of the QUIC protocol at the service level and application level, together with all of the relevant modules like the look-up table for labeling datasets.

Our library's implementation is based on Tensorflow 2.1, CUDA 10.1, TensorRT 6.0.1.5, and CuDNN 7.6.5. By creating automatic installation scripts, we have made it easier for DeepTraffic's environment to be set up on Ubuntu 18.04 LTS and later. Listing 1 shows how these scripts are used for QUIC traffic.

```
# Installs dependencies and activate the python virtual environment
./install-deps-QUIC.sh
./install-py-QUIC.sh
. activate-env-38-QUIC.sh

```

Listing 1: Set-up the environment for DeepTrafficV2

.4.1 Pre-processing

Packet traces (PCAP files) are used as the input for the pre-processing, which outputs binary files in a specific output directory in a customised folder structure.

A single command, as displayed in Listing 2, may readily be used to run the complete pipeline.

.4.2 Usage

DeepTraffic's Python library, which can handle all the difficult chores associated with loading the data, doing training on GPUs or CPUs, assessing the results, loading and

```
# Pre-process the PCAP file at pcap-file and store the resulting in the output-dir
# and keep the cache and YAF output as well
./pcap_to_dataset_quic.sh /path/to/pcap-file -o /path/to/output-dir --keep-cache
```

Listing 2: Converting an input PCAP file with QUIC protocol to pre-processed dataset

saving models, etc., with just a few lines of code, is where the program's essential features are implemented. Additionally, it is intended to function with zero configurations.

The loading of the dataset supplied by the command (cf. Listing 2), which is undoubtedly the initial step in any data pipeline in DeepTrafficV2, is shown as a complete ML pipeline in Python in Figure Listing 3. The training is then carried out by the *Training* object, which is created together with the model. The same dataset is used to conduct volumetry on the model once it has been visually verified.

```python
import deeptraffic.train
import deeptraffic.model
import deeptraffic.dataset

# load data-set with service-level labeling
dataset = deeptraffic.dataset.DataSet("/path/to/dataset", labeling="service")

# create the UW model with two feature categories (Flow time-series and TLS headers)
model = deeptraffic.model.Sigmetrics2021ModelSpec(dataset, features="fh")

# train the model
training = deeptraffic.train.Training(model, dataset)
training.train_model("/path/to/output.model")

# model validation (with PyPlot visualization of confusion matrix)
training.validate(plot=True)
```

Listing 3: A full ML pipeline from loading the dataset to computing the volumetry in DeepTrafficV2

Configuration

DeepTrafficV2 was created with ease of use and simplicity in mind. However, if required, the user may adjust and fine-tune all of the model's training and application-specific features. The possible settings are displayed in Table 3 . The configurations are made using a YAML file named `conf.yml`.

Configuration Name	Default	Description
`cores`	64	number of Spark executors
`spark.driver.memory`	8g	Spark driver memory
`spark.executor.memory`	4g	Spark executors memory
`preproc.cfm.replace_nan`	0	whether should replace NaN in CFM results and if so, with what value
`preproc.cfm.replace_pos_inf`	2	whether should replace +inf in CFM results and if so, with what value
`preproc.cfm.replace_neg_inf`	-2	whether should replace -inf in CFM results and if so, with what value
`model.packet_cutoff`	600	cut-off threshold for TLS headers in bytes
`model.max_flow_size`	1024	maximum flow packets included in training
`model.num_headers`	3	maximum handshake packets included in training
`model.activation`	relu	neuron activation function
`model.sigmetrics.header_channel_size`	256	no. channels in each layer of the model in the side processing raw traffic
`model.sigmetrics.flow_channel_size`	512	no. channels in each layer of the model in the side processing traffic shape time-series
`model.ifip.flow_dropout_size`	0.5	The dopout rate after the SLSTM layers

`model.ifip.use_maskinglayer`	True	accepted values: ["True", "False"] whether to use masking layer at the begining of flow time-series or not
`model.ifip.slstm_layers`	SLSTM	accepted values: ["SLSTM", "BLSTM", "CONV1D"] Use of architecture for Flow time-series side
`model.sigmetrics.aux_channel_size`	200	no. channels in each layer of the model in the side processing statistical features
`model.sigmetrics.dense_channel_size`	128	no. channels in each layer of the model in the part after concatenation of the three parts (fully-connected layers)
`model.sigmetrics.recurrent_units`	256	no. channels in the model's SLSTM part
`training.dataset.alpn_filter`	None	accepted values: ["None","h1", "h2", "h1h2", "noh1h2"] ALPN filtering
`training.analysis.use_top_k`	False	accepted values: ["1", "2", "3", "4", "False"] whether to use top-k accuracy analysis
`training.validation_set_ratio`	5	ratio of validation set to training set
`training.models_dir`	./Models	directory for storing model weights backup at checkpoints
`training.epochs`	10	how many epochs to train for
`training.learning_rate_checkpoint`	3	how many epochs between decimations of the learning rate
`training.learning_rate_discount`	0.1	co-efficient for discounting learning-rate at checkpoints

`training.initial_learning_rate`	0.001	initial learning rate in training
`labeling.domains_path`		path to directory of domains data used for labeling relative to the execution path

Table 3: The available configurations for the DeepTrafficV2 library, their default value and their description.

.5 Software and Hardware Stack for the Automatic Approach

The tests were run on multiple computers. One NVIDIA Tesla P40 GPU with 24GB of RAM, 56 Intel(R) Xeon(R) Gold 5120 2.20GHz CPU, and 376 GB of RAM. Three computers with NVIDIA Tesla A100 GPU with 40GB of RAM, 2x AMD EPYC 7302 16-Core CPUs, and 512GB of RAM. Two computers with NVIDIA Tesla A40 GPU with 48GB of RAM, 2x AMD EPYC 7272 12-Core CPUs, and 512GB of RAM. Tensorflow 2.2 [1] with Keras [15], PySpark 2.4.4 [62] make up the neural architecture design and pre-processing software stack.

.6 Using Stack of Cells

Through experiments with the *AutoML4ETC* on the *packet raw bytes* features, we saw that adding a stack of *Normal Cells* and *Reduction Cells* (e.g., *"NRNR"*) will not only lead to less generalization because of the perfect fitting over the training dataset but also has a negligible effect on the classification accuracy while increasing the number of parameters significantly.

.7 AutoML4ETC for Flow Statistics

To show the effectiveness of the *flow statistics* search space, we use the simple setting of the *AutoML4ETC* method on the two smallest QUIC and TLS real-world datasets. We use only partial training of child models strategy to find the best model and further train the

Table 4: *AutoML4ETC* vs UW MLP [4] for *flow statistics* features

Dataset	Model	Accuracy (%)	W Avg. F-1 score (%)	W Avg. recall (%)	W Avg. precision (%)
May 2021	**AutoML4ETC**	**64.37**	**64.16**	**64.37**	**66.54**
	UW MLP	45	43.03	45	49.6
QUIC May 2021	**AutoML4ETC**	**50.45**	**61.32**	**50.45**	**89.32**
	UW MLP	24.07	32.92	24.07	87.9

best model for additional 30 epochs. Table 4 compares the performance of *AutoML4ETC* to the UW MLP part of their Tripartite model [4]. It is evident that the *AutoML4ETC* gains significantly higher accuracy with a 19.37 and 26.38% difference on the *May 2021* and *QUIC - May 2021* datasets, respectively. However, these features alone show inferior classification performance compared to the *packet raw bytes* and *flow time-series* features as they contain summarized information of a flow. The inferior performance of the *flow statistics* was highlighted at beginning of the thesis when evaluating the UW-A model on the baseline dataset.

.8 Training Child Networks

Each child network is trained with an initial learning rate of 0.001 for *packet raw bytes* search space and 0.0001 for *flow time-series*. Furthermore, we cut the learning rate in half every 10 epochs through the training of child models. This method suggests better convergence and a finer resolution searching for the gradient descent algorithm. Moreover, we use the *Adam* optimizer and ”sparse categorical crossentropy” loss function to train the child networks. The *Factorized Reduction* and *Filter Alignment* hyper layers for *ENAS micro* search space were used along with the searching algorithms implementations. Tensorflow [1] FLOPS reporter was used to produce results in Table 7.5.

.9 Search Algorithms

RL

The *RL* algorithm is based on the RNN controller in [38] that uses *REINFORCE* with baseline and *adam* as the optimizer. The RNN controller is a single-layer LSTM with 100 Tanh constant 1.5 applied to the controller logits. Baseline decay is set to 0.999. The norm of gradients was clipped at 5.0. The learning rate for *adam* optimizer is 1e-3.

MCTS

The *MCTS* algorithm is based on the [58] that uses *UCT* [7]. The maximum node expansion is set to 10. When rolling out, the sample size for the simulation network to assess potential pathways is set to 10. The meta-learner is *LightGBM* [24] with default parameters.

EA

The *EA* algorithm is based on the [39]. The population size is set to 20. Furthermore, the number of parent candidates selected per evolution cycle is set to 5.

RS

Makes random decisions at each step and does not have any state to update itself from previous decisions.