

A COMPUTER-IMPLEMENTED METHOD, AND A SYSTEM

Technical Field

The invention relates generally to the field of traffic classification. One aspect of the invention relates to a computer-implemented method. Another aspect of the invention relates to a system. Another aspect of the invention relates to a traffic classification device.

Background

Digital network use cases become a key to enabling smart cities through applications like smart drones, autonomous vehicles, and the tactile Internet. Networks must adapt dynamically to cater to various real-time applications on a shared network infrastructure. For instance, tactile Internet requires ultra-low latency, drone surveillance demands high bandwidth, and the metaverse combines both latency and bandwidth needs. These dynamic, application-specific requirements in next-gen networks (NGNs) require the network engineers to rethink the network device configurations to ensure Quality of Service (QoS) and Quality of Experience (QoE). Real-time traffic flow understanding and categorization remain challenging but are crucial for optimizing flow management and guaranteeing end-to-end (E2E) performance for each protocol type.

Flow classification is essential for optimizing E2E performance and responsiveness. Fidelity in categorization is vital for resource allocation based on application/protocol-specific QoS requirements. For critical applications like video surveillance and VoIP in infrastructure, low-latency and high-fidelity are crucial for real-time threat detection and mitigation. High-fidelity classification helps networks prioritize time-sensitive data, improving overall QoS. Current traffic filtering methods may be slow and resource-intensive. Therefore, rethinking architectures to meet real-time speed/latency needs is desirable.

For example, in CN110896381 an approach is disclosed using a Convolutional Neural Network (CNN) model for network traffic classification. However, the hyperparameters of the CNN model are manually optimised.

It may be desirable to provide a more effective or efficient solution in some applications.

Summary

Embodiments may be implemented as set out in any of the independent claims. Some optional features are defined in the dependent claims.

Implementation of the techniques disclosed herein may provide significant technical advantages. Advantages of one or more aspects may include one or more of the following:

- The technical solution of faster traffic management based on the technical problem of sub optimal traffic management.
- The technical solution of higher accuracy traffic management based on the technical problem of sub optimal traffic management.
- The technical solution of higher security traffic management based on the technical problem of sub optimal traffic management.
- The technical solution of more efficient traffic management based on the technical problem of sub optimal traffic management.
- The technical solution of lower latency traffic management based on the technical problem of sub optimal traffic management.
- The technical solution of reduced data centre energy requirements based on the technical problem of sub optimal traffic management.
- The technical solution of reduced greenhouse emissions based on the technical problem of sub optimal traffic management. Because QoS and other network parameters can be maintained it allows less refreshing of pages or repeated unsuccessful requests for the same data.
- The technical solution of generating an optimal network traffic flow classification models using Reinforcement Learning (RL) techniques that can dynamically adapt to changing network conditions.
- The technical solution of prioritizing specific traffic types and enforce regulatory compliance.
- The technical solution of enhancing network management by optimizing resource allocation for specific traffic types.
- The technical solution of improved security for traffic types identified.

- The technical solution of reducing model convergence time in a large workload, complex and dynamic network environments such as 5G and beyond.
- The technical solution of automating the hyperparameter tuning using RL for a given network environment based on the technical problem of frequent deoptimization of the hyperparameters for a dynamic network environment.

In an exemplary implementation, the functionality of the techniques disclosed herein may be implemented in software running on a server (such as a one or more servers, one or more virtual machines, one or more processors or a cloud computing platform), or a dedicated hardware device, which communicates with the applications running on user terminals. The software which implements the functionality of the techniques disclosed herein may be contained in a computer program, or computer program product. The server manages communication channels with a large number of user terminals, according to an optimised traffic management algorithm.

Brief Description of the Drawings

The invention will now be described, by way of example only, and with reference to the accompanying drawings in which:

Figure 1 is a physical block diagram illustrating an exemplary communication system.

Figure 2 is a logical block diagram illustrating an exemplary communications network for the system in Figure 1.

Figure 3 is a schematic diagram of a server from Figure 1.

Figure 4 is a flow diagram of a proposed optimised admin parameters algorithm.

Figure 5 is a schematic diagram of the training, and evaluation of the CNN model, and training of the DQN agent.

Figure 6(a) is a flow diagram of a proposed optimised traffic management algorithm.

Figure 6(b) is a flow diagram of a proposed data pre-processing steps for model training.

Figures 7(a)-(d) are graphs of the results of the accuracy of the CNN by varying the reward for different values of β .

Figures 8(a)-(c) are graphs of the results of the performance of the CNN with optimised values of δ and β .

Figure 9 is a schematic diagram of the extraction of training data from a 5G network infrastructure.

Detailed Description

One or more embodiments in the following description regarding network traffic classification, is divided as follows.

- Example use cases for training the CNN architecture.
- Design procedure for a CNN architecture for a given network application.
- Formulate a CNN architecture design as a RL based hyperparameters optimization problem to adapt for the diverse range of applications enabled by current NGNs.
- Design procedure for a reward function for fast convergence of RL agent to generate a CNN architecture with high-fidelity accuracy and reward measures.
- Evaluate the architecture using the CMU-SynTraffic-2022 dataset.
- Example inference mode implementation of the trained CNN architecture for high-fidelity network traffic flow classification.

USE CASES

Network traffic flow classification is an important part in managing a digital communications infrastructure. In a sophisticated telecommunications network, network operators/owners faces challenges related to bandwidth allocation, congestion management and ensuring equitable access to digital services. Through traffic flow classification, prioritization of critical applications, optimization of network resources and maintaining seamless connectivity for businesses, residents and government agencies can be achieved. This proactive approach helps alleviate congestion, enhance network performance and ensure reliable access to digital services. With a large volume of financial transactions, sensitive information and corporate communications traversing its networks, network operators/owners must safeguard against cyber threats, data breaches and malicious activities. Network traffic classification enables organizations to identify and mitigate potential risks, enforce access controls and adhere to regulatory compliance standards such as the Personal Data Protection Act (PDPA) and the Cybersecurity Act, or their equivalents in each country.

Investments in a network traffic flow classification technology that is capable of effectively classifying network traffic in high volume ensures optimal Quality of Service

(QoS) and enhance the overall quality of experience for its users. This cutting-edge technology will play a pivotal role in bolstering cybersecurity defences, safeguarding critical infrastructure and maintaining trust in its digital ecosystem.

Traffic classification has very broad applications. Classification is the ability to identify the type of traffic/applications, e.g., Gaming, Whatsapp, Youtube, Spotify, Netflix (from typical consumer perspectives), Email, SAP, Oracle, Zoom, Web meeting etc (from typical business perspectives). Example applications include:

1. Traffic management - When traffic can be properly classified, we can assign priorities and give more resources/bandwidth to important applications, to enhance user experience. E.g., for anyone that watches Netflix, the experience will be bad if the video is constantly interrupted by bufferings or lags. For anyone attending a business meeting on Zoom, the users will be highly sensitive to video/sound quality. Therefore, important traffic can be assigned more network resources over less important traffic, e.g., email (email that arrives 1 min late do not matter).
2. Traffic engineering - Ability to block undesirable applications. E.g., block Google, Whatsapp, Facebook, etc, by countries such as China. In businesses, prevent employees from using applications deemed unproductive.
3. Network forensics - Troubleshoot network problems, identify why certain applications do not work as intended.
4. Lawful investigations - Ability by law enforcements to investigate the communication profiles of person of interest.
5. Cybersecurity - Identify and block malicious traffic at ingress/egress points.

Traffic classification can be done at any part of the network, from edge to distribution to core. The optimal location depends on the type and profile of traffic to be classified and / or the particular application. For example, traffic classification can be deployed in the following hardware devices: programmable routers/L3 switches, firewalls/Next-generation firewalls (NGFW), IPS (intrusion prevention systems), IDS (intrusion detection systems) and others. Traffic classification can also be used in analytical software to perform filtering, storing of filtered traffic to investigations.

Traditionally, traffic classification was done by using signatures identified in network packets. With the signature method, the speed of classification may not be fast or scalable enough for modern high-speed networks, because it depends on central processing unit (CPU) processing.

For example, traditional methods like Deep Packet Inspection (DPI) struggle with encrypted content and faced computational complexity challenges. Other traditional methods such as port-based approaches can't distinguish encrypted from regular traffic. These traditional methods were constrained by their inability to adapt to dynamic network conditions.

According to an example embodiment, RL with CNNs hyperparameter tuning can be used for traffic classification. Further RL/CNN can be deployed using graphics processing units (GPUs), which allows cheaper and/or faster processing compared to the CPU processing in the prior art.

An example communication system 100 is shown in Figure 1. A number of users each having a communications device 104, one or more network administrators having a user interface 106, on-prem servers and / or cloud servers in the core network 102 which is remotely managed by the network administrator using 106, and a central monitoring server 108 (or geographically distributed servers) are connected via a communication network. Each user device 104 may communicate with the server 102 to obtain data and gain access to the applications.

A communications provider may use the system 100 for the purpose of communication infrastructure. This may involve wireless communications such as 5G, 6G or other NGN networks, WiFi, or wired communication such as TCIP/IP based systems. Typically, the system 100 used by the communications provider will exchange data between the user devices 104, and the central server 102, a remote data source on the internet or other data sources depending on the application. The system 100 may then be used to simultaneously optimise the traffic between a large number of users using an optimised traffic management algorithm that is run on the central monitoring server 108, as will be described in more detail below.

Referring to Figure 2, further details of the components in the system of Figure 1 are now described. The communication apparatus 100 comprises the monitoring server 108, and it may include the user communication device 104, network administrator's device 106. These devices are connected to the high compute edge network 202, distribution network 204 and core network 206 through respective communication links 110, 112, and 114 implementing, for example, internet communication protocols. The communication devices 104, and 106 may be able to communicate through communication networks and/or protocols, including cellular communication networks, LAN, WAN, private data networks, VPN, fibre optic connections, laser communication, microwave communication, satellite communication, Bluetooth, Wifi, NFC, etc.

Monitoring server 108 is a notional entity providing traffic management functions by implementing an optimised traffic management algorithm according to one or more of the present embodiments. Server 108 may be provided in the high compute edge network 202 and / or the core network 206. Similarly, a virtualised optimised traffic management algorithm may be implemented in any or all of the high compute edge network 202 and / or the core network 206. The optimised traffic management algorithm may be implemented in a virtual server provided in the cloud or a discrete device such as a router or the high compute edge network (MEC) in 5G network context, or the core network function. If the MEC edge interface runs the classification model, this will feedback to the core network for near-real-time classification for high speed networks. When implemented as a cloud service³⁰², the cloud-based APIs can interact both with the MEC 304 and Core network functions 306, as shown in Figure 3.

In the example shown in Figure 2, the server 108 may comprise several individual components including, but not limited to, one or more microprocessors (CPU), one or more GPU, one or more memories (e.g., a volatile memory such as a RAM), and/or longer-term, non-volatile, or persistent, memory (e.g., SSD (Solid State Drive) or Hard disk drives (HDD)) for the loading of executable instructions, the executable instructions defining the functionality the server apparatus 108 carries out under the control of the CPU and/or the GPU. The communication server apparatus 108 also comprises one or more input/output modules (e.g. 10G ethernet controller) allowing the server to

communicate over the communication network. One or more admin interfaces are provided, for example, including computing peripheral devices such as display monitors, computer keyboards and the like. In a virtual implementation, the physical hardware components may be provided by a restricted share or pooled hardware.

In the examples above the CNN model training and inference mode classification may be processed primarily by the GPU. Depending on the application specific tasks within the overall procedure may be allocated to the CPU or GPU.

The server 108 may be a single server as illustrated schematically in Figure 2.

Alternatively, the functionality performed by the server apparatus 108 may be distributed across multiple physically or logically separate server components. In the context of the present specification, "a server", "the server" or "said server" is a computer program that is running on appropriate hardware and is capable of receiving requests (e.g., from electronic devices) over a network, and carrying out those requests, or causing those requests to be carried out. The hardware may be implemented as specialised hardware devices, one or more physical computers, one physical computer system, one or more virtual machines, or a cloud-based server network. In the present context, the use of the expression a "server" is not intended to mean that every task (e.g., received instructions or requests) or any particular task will have been received, carried out, or caused to be carried out, by the same physical server (i.e., the same software and/or hardware); it is intended to mean that any number of software elements or hardware devices may be involved in receiving/sending, carrying out or causing to be carried out any task or request, or the consequences of any task or request; and all of this software and hardware may be one server or multiple servers. The same approach is to be taken for references to other systems, software or hardware components e.g., processor, memory, storage etc. For example, the term processor may mean a physical CPU core, a physical GPU core, a virtual CPU core, a virtual GPU core, or any number of processing type functionalities used to carry out the firmware, OS, or software instructions.

The server apparatus 108 may also comprise one or more databases stored in volatile memory and/or non-volatile memory, for storing data. The data may be stored in a data structure according to the requirements of the application. The database may be

replicated, distributed, sharded or otherwise optimised according to the requirements of the application.

The user communication device 104 may comprise several individual components including, but not limited to, one or more microprocessors, a memory (e.g., a volatile memory such as RAM), and/or longer-term memory such as flash memory or SSD (Solid State drives) for the loading of executable instructions, the executable instructions defining the functionality the user communication device 104 carries out under the control of the microprocessor. The user communication device 104 also comprises an input/output module (e.g. 5G radio chipset) allowing the user communication device to communicate over the communication network. A user interface is provided for user control. If the user communication device 104 is, say, a smartphone or tablet device, the user interface will have a touch panel display as is prevalent in many smartphones and other handheld devices. Alternatively, if the user communication device 104 is, say, a desktop or laptop computer, the user interface may have, for example, computing peripheral devices such as display monitors, computer keyboards and the like.

The network administrator's device 106 may be, for example, a desktop or laptop computer with the same or a similar hardware architecture to that of the user communication device 104. Alternatively, the Administrator may access the server 108 directly using server 108 admin interface on the core network.

An example of the server 108 includes a GPU for the model to be trained on and to run in inference mode, such as a NVIDIA RTX A5000 which is well-suited for machine learning model development due to its robust specifications and architecture. On a large-scale, 108 can be GPU-only, GPU-CPU hybrid, CPU-only or a combination of these setups depending on the infrastructure implementation on the high-computing edge 202. The inclusion of 256 third-generation Tensor Cores significantly boosts performance in AI applications by accelerating matrix operations critical for deep learning algorithms such as CNN. Running a CNN under a GPU involves several differences in terms of speed, efficiency, and parallel processing capabilities. The differences can be categorized into the following: (i) Parallel Processing Power, (ii) Specialized Architecture, (iii) memory bandwidth, (iv) Specialized Libraries and APIs and (v) Scalability.

GPUs are highly parallelized processors designed to handle thousands of tasks simultaneously, whereas CPUs are optimized for sequential processing. CNN operations, such as convolutions and matrix multiplications, are highly parallelizable. GPUs can execute these operations much faster by distributing the workload across thousands of cores, each capable of performing simple computations simultaneously. On the other hand, CPUs have fewer cores optimized for handling complex tasks sequentially, making them less efficient for CNN computations.

GPUs are purpose-built for handling graphics rendering, which involves processing large amounts of data in parallel. This architecture, with its many cores and high memory bandwidth, is well-suited for deep learning tasks like training and inference with CNNs. On the other hand, CPUs are more general-purpose processors optimized for a wide range of tasks, including running operating systems and handling diverse workloads. While CPUs have made significant advancements in recent years, they are still less efficient than GPUs for deep learning computations due to their less parallelized architecture.

GPUs typically have higher memory bandwidth compared to CPUs. This means that data can be transferred between the GPU's memory and processing cores quicker, reducing the time spent waiting for data to be fetched during computations. CNNs often involve processing large volumes of data. The higher memory bandwidth of GPUs allows them to efficiently handle this large volume of data, further enhancing their performance compared to CPUs.

Another factor contributing to the speed advantage of GPUs in processing CNNs is the availability of optimized libraries and APIs (Application Programming Interfaces) specifically designed for GPU computing. Popular deep learning frameworks like TensorFlow, PyTorch, and CUDA (Compute Unified Device Architecture) provide support for GPU acceleration, allowing developers to leverage the parallel processing capabilities of GPUs seamlessly. These libraries and APIs optimize the execution of CNN operations on GPUs, further enhancing their performance compared to running on CPUs.

Designing CNN models for diverse network traffic classification applications presents new challenges aside from feature selection such as (i) hyperparameter optimization and (ii) model convergence for highly varying traffic patterns. The design of the CNN model's architecture and hyperparameters has a great influence on the performance of the model. Fine-tuning the hyperparameters of CNN and determining the optimal number of layers in the architecture even for a singular network application is an arduous task that demands manual effort and domain expertise. This labour-intensive process often involves iterative experimentation and adjustment to strike the right balance to ensure that the CNN aligns seamlessly with the unique requirements of the targeted application. As an example, for 5G networks that encompass a broad range of network applications, the process of designing the CNN to adapt to these applications becomes ever more complex due to the diverse and dynamic nature of the network environment.

According to an example embodiment Neural Architecture Search (NAS) may be used to automate neural network architecture design. However, Evolutionary algorithms (EAs) demand substantial computing resources, impractical for real-time traffic management. According to an example embodiment RL-based NAS, following RL principles, may be used to automates architecture design, compared to EA-based NAS.

CNNs have been traditionally used for image processing. However, a limitation of CNNs --- like other deep learning models, is the need for network engineers to engage in feature selection during the training and testing phases, which requires expertise, careful planning, manual feedback, and retraining.

Designing a CNN architecture can be categorized as a model selection problem. Fine-tuning the hyperparameters of CNN even for a singular network application is an arduous task that demands a labour-intensive process involving iterative experimentation and adjustment to strike the right balance and ensure the CNN aligns seamlessly with the unique requirements of the targeted application. For some networks such as 6G, the process of fine-tuning the hyperparameters of CNN to adapt to various applications becomes ever more complex due to its diverse and dynamic nature. Some of the hyperparameters of CNN include learning rates of the optimizer, batch sizes, and network architectures. Furthermore, the dynamic nature of network traffic within the network

introduces additional layers of complexity such as adjusting the number of convolutional layers and number of filters in CNN to capture both the subtle and complex patterns in network traffic. The continuous evolution of traffic patterns, combined with the need for real-time response, demands an agile and adaptable CNN architecture.

Reinforcement learning (RL) offers a promising approach to address the complex task of generating optimal CNN architectures, reducing the challenges associated with extensive hyperparameter tuning for diverse network applications. By framing CNN architecture design as an RL problem, an autonomous agent can iteratively explore and learn the best configuration based on defined rewards or objectives. This approach not only automates the process but also adapts to the dynamic nature of networks and their varied applications. Our focus here is on modelling a reward function for the RL agent to learn from performance feedback and continually enhance the CNN architecture's effectiveness across various network applications. This method offers a more automated and adaptive solution for tailoring CNN architectures.

Thus, it will be appreciated that Figures 1 to 3 and 9 and the foregoing description illustrate and describe a system 100 comprising:

- at least one memory configured to store an untrained CNN classifier model, and an action space of a plurality of combinations of hyperparameters, and

- at least one processor configured to execute instructions stored in one or more of the at least one memory to:

- for each of a plurality of episodes:

- initialise a plurality of episode variables,

- for each of a plurality of iterations:

- train the CNN classifier model using a state including the respective hyperparameters from the action space chosen by a reinforcement learning agent for that iteration,

- evaluate the accuracy of the trained CNN classifier model,

- evaluate the reward for that iteration using a quantized reward function with a negative reward for iterations with a lower accuracy compared to the previous iteration,

- store at least the reward and the state in replay memory, and
- determine a state for the next iteration based on predetermined criteria,
- increase the replay memory, and
- update the replay memory with the action space from the iterations for that episode,
- train the reinforcement learning agent based on the replay memory.

Further, it will be appreciated that Figures 4 to 6 illustrate and describe a computer implemented method performed in a device or virtual appliance configured for reinforcement learning, the method comprising, under control of at least one processor or at least one virtual processor:

- storing an untrained CNN classifier model for network traffic classification,
- storing an action space of a plurality of combinations of hyperparameters,
- for each of a plurality of episodes:
 - initialising a plurality of episode variables,
 - for each of a plurality of iterations:
 - training the CNN classifier model using a state including the respective hyperparameters from the action space chosen by a reinforcement learning agent for that iteration,
 - evaluating the accuracy of the trained CNN classifier model,
 - evaluating the reward for that iteration using a quantized reward function with a negative reward for iterations with a lower accuracy compared to the previous iteration,
 - storing at least the reward and the state in replay memory, and
 - determining a state for the next iteration based on predetermined criteria,
 - increasing the replay memory, and
 - updating the replay memory with the action space from the iterations
- for that episode,
- training the reinforcement learning agent based on the replay memory.

RL-based optimization algorithm (ROA)

A Q-learning RL-based optimization algorithm (ROA) for CNN hyperparameter optimization is proposed according to an example embodiment, to address challenges in scalability and accessibility. ROA in this example, employs Q-learning to optimize the search process for NAS. A table of selected CNN hyperparameters is defined for the ROA to learn to select the optimal values from. ROA will select the value for each hyperparameter with the best accuracy and update the value in its Q-table.

However, traditional RL methods such as Q-learning struggle with state and action space dimensions and are inadequate for adapting NAS to design CNNs for the intricate patterns in NGN traffic. Deep Q-learning is a solution, utilizing deep neural networks to capture diverse network flows' intricate patterns. An RL-based optimization framework may apply deep Q networks (DQN) to search CNN models, showing better accuracy than other methods.

MODEL

Epsilon Greedy Algorithm

We use the Epsilon Greedy algorithm to address the exploration-exploitation trade-off for RL agents. During action selection, it balances exploration (random action) and exploitation (action with the highest Q-value) based on a probability parameter (p) and epsilon value (ϵ). When p exceeds ϵ , the RL agent chooses the action with the maximum Q-value, prioritizing exploitation; otherwise, it selects a random action. Initially, a higher ϵ value encourages the RL agent to explore extensively, facilitating diverse experiences and Q-values discovery with different state-action pairs. As the training progresses, ϵ decays --- gradually emphasizing exploitation. This adaptation ensures that the RL agent will leverage on the exploration to refine its Q-value estimates and converge towards an optimal policy. Our implementation designs the impact of the decay on ϵ to diminish linearly if the total number of episodes is increased. This linear decay per episode diminishes as the episode count rises, enabling the algorithm to balance exploring new possibilities and exploiting well-known high-reward actions.

Bellman Equation

The Bellman Equation (Eq (1)) is an optimality equation that describes the relationship between the value of a state-action pair and the expected future rewards. It is pivotal for RL decision-making and updates the Q-network's weight parameter to learn and make better predictions over time.

$$y_i = r_i + \gamma \cdot \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta) \quad (1)$$

where, y_i , i.e., target Q-value is calculated by adding the immediate reward to the discounted future rewards, considering the Q-network's weight parameter, θ . It utilizes the state s_{i+1} and action a_{i+1} in the next iteration to estimate future rewards, to calculate Q-value. During training, this target Q-value updates the Q-network, minimizing the disparity between predicted and target Q-values, enhancing Q-network's accuracy. We set the discount factor (γ) to 0.95 to optimize long-term gains.

Quantized Reward Function

The reward function is used in one example of RL for shaping the learning process and the agent's performance. Designing a reward function for deep learning model hyperparameter tuning requires optimizing the model's performance based on intricate objectives. Our proposed reward function, detailed in Eq (2), aims for faster RL convergence in generating a high-fidelity accuracy CNN architecture. Through empirical testing, we explored three scenarios for evaluating the performance of each generated CNN architecture in an iteration: (i) a high positive reward for an accuracy increase of at least δ compared to the previous iteration, (ii) a low-positive reward for slight accuracy improvements (within δ) over the previous iteration, and (iii) an exponential negative reward for lower accuracy compared to the previous iteration. We conducted tests with varying linear factor (β) and the difference in CNN accuracy between the current and previous iterations (δ) to assess the reward function's robustness. Eq (3) presents the cumulative iteration rewards for each episode, enabling monitoring of the learning agent's training progress.

$$\mathcal{R}_i = \begin{cases} \beta \times \tau_i & , (\tau_i - \tau_{i-1}) \geq \delta, \\ \left(\frac{\beta \times \tau_i}{2}\right) & , 0 \leq (\tau_i - \tau_{i-1}) < \delta, \\ -\left(\frac{\tau_i + \tau_{i-1}}{\tau_i}\right)^2 & , \tau_i < \tau_{i-1}. \end{cases} \quad (2)$$

$$\mathbf{R}_e = \sum_{i=1}^n \mathcal{R}_i \quad (3)$$

Deep Q-learning

A Deep Q-learning neural network employs a meta-controller based deep learning model, as the learning agent to explore a predefined hyperparameter search space for CNN. In each iteration, the agent selects an action to modify a hyperparameter value of the CNN architecture, followed by training on a dataset and evaluating performance on a testing dataset. This accuracy serves as a reward signal, guiding the agent's policy updates in the RL process. The objective is to enhance the policy for the learning agent to select actions yielding higher rewards using NAS. Our DQN algorithm implementation is shown in Algorithm 1. The epsilon greedy algorithm and experience replay makes it suitable for flow traffic classification using the DQN-NAS framework.

TABLE III: Mathematical Notations.

Notation	Definition
$\mathcal{R}_i$	Reward function for iteration i
$\mathbf{R}_e$	Reward function for episode e
τ_i	Test accuracy for iteration i
ϵ	Epsilon value
p	Generated probability for epsilon greedy algorithm
α	Learning rate for Bellman equation
γ	Discount factor for Bellman equation
$Q(s_i, a_i)$	Q-value for the state S and action A at iteration i
β	Linear factor for reward function
δ	Difference between the CNN's accuracy for iteration i and $i - 1$
θ	Q-network's weight parameter
B	Batch size for experience replay
D_n	Replay memory for experience replay
N	Capacity of replay memory for experience replay
y_i	Target Q value for iteration i
A	Action Space
S	Search Space

PROPOSED METHODOLOGY

Dataset and Design of CNN Architecture

We used the CMU-SynTraffic-2022 dataset, comprising 62 network traffic features, to simulate real-world network activity in diverse traffic streams. This dataset suits our traffic classification model for various NGN applications: audio streaming, web browsing, chat, file transfer, email, peer-to-peer, video streaming, and VoIP. Our CNN architecture is adapted from the model in Gurdip Kaur, Arash Habibi Lashkari, and Abir Rahali. "Intrusion traffic detection and characterization using deep image learning". In 2020, IEEE, Intl Conf on Dependable, Autonomic and Secure Computing, pages 55– 62. IEEE, 2020, which is incorporated herein by reference. This study transforms two network datasets, one with 64 features and the other with 68 features, into 9x9 gray images before inputting them to the CNN model. We maintain the same number of layers for our CNN architecture and use the RL agents to tune the hyperparameters in each layer aiming to enhance the performance specifically for the CMU-SynTraffic-2022 dataset. Table VI in the performance study section shows the detailed structure of the CNN architecture used by the study.

Example Algorithm for DQN-NAS

Model	Architecture
Initial State	Conv2D(32, (5, 5), activation='relu', input_shape=(9, 9, 1), padding='same'), MaxPooling2D((2, 2), strides=2), Dropout(0.50), Conv2D(64, (3, 3), activation='relu', padding='same'), MaxPooling2D((2, 2), strides=2), Dropout(0.25), Flatten(), Dense(1024, activation='relu'), Dropout(0.40), Dense(8), Adam optimizer with learning rate of 0.0001

TABLE I: State Initialized at Start of Each Episode.

Variable	Initial Value	Range	Purpose
max_tries	0	[0, 10]	The maximum number of attempts an RL agent can make in selecting an action to generate a CNN model that has already been generated within a single episode. If the maximum number of attempts is reached, the RL agent exits the iteration loop and initiates a new episode.
prev_acc	0	[0, 1]	Accuracy for a CNN model from the previous iteration.

TABLE II: Variables Initialized at Start of Each Episode.

Below we present the various steps of the proposed algorithm for the generation of an efficient and high-accuracy CNN for the specific network application according to one embodiment. Algorithm 1 below, provides a summary of the steps.

Algorithm 1 DQN with Experience Replay, Epsilon-Greedy and our Reward Function

```
1: Initialize Q-network with random weights
2: Initialize replay memory  $D_n$ ,  $n \in \{1, 2, \dots, N\}$ 
3: for each episode  $e \in \{e_1, e_2, \dots, e_{50}\}$  do
4:    $s_i \leftarrow$  Initialize using Table I ( $\forall i \in e$ )
5:   for each iteration  $i$  in  $e$  do
6:     Generate, train, validate CNN based on  $s_i$ 
7:      $\tau_i \leftarrow \text{CNN.accuracy}(s_i)$ 
8:     Update  $\mathcal{R}_i$  using Eq (2)
9:     if  $p < \epsilon$  then
10:       $a_i \leftarrow \text{rand}(A)$  //Table V Action space
11:     else
12:       $a_i \leftarrow \text{argmax}_{a_i \in A} (Q\text{-Network})$ 
13:     end if
14:      $D_n \leftarrow (s_i, a_i, r_i, s_{i+1})$ 
15:      $s_i \leftarrow s_{i+1}$ 
16:      $\text{prev\_acc} \leftarrow \tau_i$ 
17:     if  $((\tau_i \geq 0.99) \vee (\text{max\_tries} \geq 10))$  then
18:       exit
19:     end if
20:   end for
21:    $B \leftarrow \text{UniformSample}(D_n) \forall n \in \{1, 2, \dots, N\}$ 
22:   for each tuple  $(s_i, a_i, r_i, s_{i+1}) \in B$  do
23:     Update  $y_i$  using Eq (1)
24:     Update Q-network's weight parameter  $\theta$ 
25:   end for
26: end for
```

As shown in Figure 5, state represents the CNN model's hyperparameters 502. The CNN model's hyper parameters 502 are used to generate a CNN model 504 and train using the training dataset 506. The trained CNN model is then evaluated with the testing dataset 508 to obtain the CNN model's accuracy 510 which is used by the reward function 512 to generate the reward. The reward, state, next state and action are stored in the replay memory of the DQN 514 and will be used to train the DQN once a sufficient number of samples are stored in the DQN. Based on the policy 516 of the RL agent, the next state can either be generated by the DQN or randomly to become the new hyperparameters for the CNN model to be trained and evaluated again. This process will repeat itself until the DQN converges.

The training phase or mode described in relation to Figure 5 earlier, includes both an Exploration and Exploitation phase by the DQN agent 514 that adapts based on reward function 512. In reinforcement learning (RL), unlike supervised learning, distinct training and testing phases are not defined. In Figure 6a the exploration phase is when the agent

tries out different actions 610 to gather information about the environment 612 whereas exploitation phase is when the DQN agent uses the knowledge it has gained to make the best decisions and maximize rewards 616 to produce the CNN model hyperparameters for network traffic classification.

Algorithm 1 will now be explained in more detail with reference to the method 600 in Figures 6a and 6b:

- 1) The data is first pre-processed 601 to gather the data in the appropriate format.
- 2) Create 602 the Action Space for the DQN. DQN will be using this Action Space to select an action to change a value in the state for the generation of a new CNN model in each iteration. See Table V for all available actions.
- 3) A variable `target_acc` is initialized 604 to store the target accuracy that a CNN model should achieve for the DQN agent to reach a terminal state within an episode during the exploitation phase. During the exploration phase, the target accuracy that a CNN model should achieve is set to a value of at least 0.99.
- 4) At the start of each episode, an initialised state shown in Table I is used to generate 606 the initial CNN model and the variables (`max_tries` and `prev_acc`) in Table II are initialized.
- 5) At the start of each iteration, a CNN model is generated based on the state for that iteration. The model is then trained and validated to obtain accuracy and / or loss. Accuracy being the primary metric.
- 6) The reward for this iteration is then computed 608 using the accuracy and Eq (2).
- 7) Select 610 the action for the generation of the next state. If the generated p is less than ϵ , a random action will be selected. Otherwise, the action that is predicted by DQN will be selected.

- 8) Check that the next state, which will result from taking this action, has not been generated previously within this episode. If yes, take the action and generate the next state. If no, increment the variable `max_tries` by one and repeat Step 6 until the maximum number of 10 attempts has been reached.
- 9) Store the experience consisting of the state, action, reward, and next state in the replay memory.
- 10) Update the value of the variable `prev_acc` with the accuracy achieved by the CNN in this iteration and set the state to the next state for the next iteration.
- 11) Repeat Steps 4 to 9 until the terminal state for the episode has been reached.
- 12) If the maximum accuracy achieved by the CNN in this episode is higher than the value of the variable `target_acc`, update the variable's value with the maximum accuracy achieved in this episode.
- 13) Sample a random batch of experiences from the replay memory and train the DQN using Eq (1).
- 14) Repeat Steps 3 to 12 until the maximum number of episodes has been reached.

Component in CNN	Hyperparameter	Range
Convolutional layer	Number of filters	[32, 64]
Convolutional layer	Filter size	[3, 5, 7]
Convolutional layer	Padding	["same"]
Pooling layer	Filter size	[2]
Pooling layer	Stride	[2]
Regularization layer	Dropout rate	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
Hidden layer	Number of neurons	[128, 256, 512, 1024]
Common	Activation function	["relu"]
Adam optimizer	Learning rate	[0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]

TABLE IV: Search Space.

Index	Component in CNN	Action
1	Convolutional_layer_1	Conv2D(32, (3, 3), activation='relu', padding='same')
2	Convolutional_layer_1	Conv2D(32, (5, 5), activation='relu', padding='same')
3	Convolutional_layer_1	Conv2D(32, (7, 7), activation='relu', padding='same')
4	Convolutional_layer_2	Conv2D(32, (3, 3), activation='relu', padding='same')
5	Convolutional_layer_2	Conv2D(32, (5, 5), activation='relu', padding='same')
6	Convolutional_layer_2	Conv2D(32, (7, 7), activation='relu', padding='same')
⋮	⋮	⋮
44	Optimizer	Adam with learning rate of 0.0005
45	Optimizer	Adam with learning rate of 0.0001

TABLE V: Action Space.

Data preprocessing 601 is shown in more detail in Figure 6b. It involves the following steps. Initially, packets are extracted 620 from pcap files. After extracting the packets, the dataset undergoes manual inspection 622 to remove missing or corrupt packets, ensuring data integrity. Feature extraction 624 is then performed using CICFlowmeter, capturing important network traffic information such as flow ID, source IP, destination IP, source port, destination port. Subsequently, the dataset is normalized 626 to standardize feature values to prevent features from dominating the training process due to differences in magnitude. Min-max scaling method is used to normalize the dataset, ensuring that feature values are uniformly distributed to a predefined range, typically [0.0, 1.0]. Finally, the dataset is split into training and validation sets 628 using an 80:20 ratio to evaluate the CNN model's performance, ensuring it can generalize well to unseen data. These data preprocessing steps collectively prepare the data from pcap files for CNN training, enabling the model to learn and classify network traffic patterns effectively.

Search Space

The CNN architecture comprises of five components: i) convolutional layer, ii) pooling layer, iii) regularization layer, iv) flatten layer, and v) dense layer. Each component has its hyperparameters that need careful tuning. Different functions exist within specific layers,

e.g., Conv2D, Conv2DTranspose, SeparableConv2D, and DepthwiseConv2D for the convolutional layer. We've detailed these hyperparameters as our search space in Table IV for action generation in our action space. Some CNN components, like pooling layer and convolutional layer padding, activation function, have restricted hyperparameter choices to prevent negative image dimensions and errors. This precaution ensures successful CNN model generation in each iteration, avoiding premature termination of the DQN agent's training.

Search Strategy and Action Space

We implement a simple Q-network in our proposed methodology for action selection within the action space. The action space comprises 45 actions (Table V) with each action altering one hyperparameter for a specific CNN layer. For instance, the first action changes the filter size in the initial convolutional layer to 3, while the second action sets it to 5. We excluded actions that modify hyperparameters for the pooling layer and convolutional layer's padding type to reduce the complexity and prevent issues like negative image dimensions.

PERFORMANCE STUDY

For evaluating the heuristics, we run the simulation on the CPU of ROG Strix G16CH, Intel(R) Core (TM) i7-13700F CPU @ 2.1GHz. The simulation can be run on a GPU-only, GPU-CPU hybrid, CPU-only or a combination of these setups depending on the specific requirements and constraints of the infrastructure. Running the simulation on a CPU, particularly on a consumer-grade CPU like the Intel(R) Core (TM) i7-13700F, was primarily for initial testing and debugging purposes to ensure the correctness of the implementation. However, to fully demonstrate the performance advantages of our proposed system, it would utilize a scaled GPU infrastructure. The proposed system has been implemented in Python using Keras and TensorFlow by utilizing Scikit-Learn. The DQN agent's training involved 50 episodes, with each episode comprising 50 iterations. Episodes could end prematurely if the RL agent reached a terminal state (e.g., target accuracy achieved). The CMU-SynTraffic-2022 dataset was split into 80% training and 20% testing sets for CNN model validation, undergoing 5 training epochs before testing. Initially, epsilon was set to 1, decaying linearly over episodes, gradually shifting from exploration to exploitation. We experimented with different β (2-10) and δ (0.05 and 0.1)

values to optimize the reward function and compare against an existing CNN model as a baseline using 5-fold cross-validation.

We use the following performance metrics to evaluate the proposed algorithm.

- **DQN Reward function:** The reward represents the cumulative reward received by the DQN agent for each episode as shown in Eq (3).
- **CNN Accuracy:** The accuracy represents the accuracy of the CNN model generated in the final iteration of each episode. The accuracy of the CNN model is obtained using the testing set after the CNN model has been trained.
- **CNN Loss function:** The loss represents the loss of the CNN model generated in the final iteration of each episode obtained using the training set.

Model	Architecture	Average accuracy
Baseline	Conv2D(32, (3, 3), activation='relu', input_shape=(9, 9, 1), padding='valid'), MaxPooling2D((2, 2), strides=2), Dropout(0.25), Conv2D(64, (3, 3), activation='relu', padding='same'), MaxPooling2D((2, 2), strides=2), Dropout(0.25), Flatten(), Dense(256, activation='relu'), Dropout(0.30), Dense(8), Adam optimizer with learning rate of 0.0001	82.21%
DQN	Conv2D(64, (7, 7), activation='relu', input_shape=(9, 9, 1), padding='same'), MaxPooling2D((2, 2), strides=2), Dropout(0.50), Conv2D(64, (7, 7), activation='relu', padding='same'), MaxPooling2D((2, 2), strides=2), Dropout(0.25), Flatten(), Dense(1024, activation='relu'), Dropout(0.40), Dense(8), Adam optimizer with learning rate of 0.001	95.47%

TABLE VI: Comparison of Performance for CNN Architectures

- **Baseline in Table VI:** Baseline refers to the CNN architecture used in Fatma M Talaat and Samah A Gamel, “RI based hyper-parameters optimization algorithm (roa) for convolutional neural network”, Journal of Ambient Intelligence and Humanized Computing, 14(10):13349–13359, 2023, the contents of which are incorporated herein by reference.
- **DQN in Table VI:** DQN refers to the CNN architecture that is generated by the DQN agent in the final iteration during the exploitation phase after training of the DQN agent has reached convergence. The parameters used in the reward function are based on the optimal settings.

Figure 7a illustrates that the DQN agent converges within 30 episodes for most of β as the sum of rewards for each episode converges. The accuracy of the CNN in Figure 7b reaches

the maximum after the sum of the rewards for their corresponding episodes in Figure 7a stabilizes. This demonstrates the reliability (and hence the high-fidelity) of the proposed DQN-NAS framework in tuning hyperparameters for CNN. There are also β values that did not perform well such as $\beta=7$ and $\beta=6, 8$ in Figure 7b and Figure 7d, respectively. The potential reason for the performance disparity could be due to model complexity as we have selected a simple deep learning model to reduce computational complexity. While the DQN agent could not converge for $\beta=7$ when $\delta=0.05$, it did converge when $\delta=0.1$ for the same β . The reward function and hence the accuracy for the DQN depends on both β and δ values. Upon experimental analysis, it is observed from Figure 7b that consistently better CNN generated by the DQN agent during the exploitation phase with $\delta=0.05$ achieves an target accuracy of 96.3%. The corresponding reward function uses $\beta=6$. For Figure 6d, the consistently better CNN achieves an accuracy of 91.1% and the associated $\beta=8$ and $\delta=0.1$. Therefore, upon empirical analysis, we chose the optimal settings of $\beta=6$ and $\delta=0.05$ in our reward function to analyse the CNN model generated in Figure 8 and Table VI.

Figure 8a-8c illustrates the average and maximum metrics of the CNN that is generated by the DQN agent using the optimal settings for the parameters of our reward function. The results further support that the training process of the DQN agent converges after 28 episodes. In Figure 8b, the provided target accuracy of 96.3% represents the goal that the CNN model, generated by the DQN agent in each iteration, should attain during the exploitation phase to reach the terminal state.

Table VI compares the performance of the DQN-NAS-generated CNN with a baseline CNN using 5-fold cross-validation on the CMU-SynTraffic-2022 dataset. The baseline achieved an average accuracy of 82.21% whereas the DQN-NAS-generated CNN achieved a better accuracy of 96.3%.

INFERENCE MODE

As shown in Figure 9, using one or more embodiments of traffic classification, the admin can set up a port mirroring at the core network to tap 902 the N2/N3 interface network traffic and generate PCAP/PCAPNG files which will be fed to the CICflowmeter. CICflowmeter is then applied on the PCAP/PCAPNG files to extract out the features as

part of the preprocessing before they are classified by the trained CNN model using the hardware/software use cases previously described.

The network administrator's device 106 may be used to monitor and control traffic management, according to one or more of the examples of traffic management given earlier.

β and δ (given in Equation 2 and 3) can change depending on the network application. Currently, these two parameters are tested for various NGN applications (audio streaming, web browsing, chat, file transfer, email, peer-to-peer, video streaming, and VoIP). We experimented with different β (2–10) and δ (0.01, 0.05 and 0.1) values to obtain the optimized β and δ values based on the QoS required.

Before deploying the framework to the network setting, the network traffic is evaluated and categorized into one of the use case applications (such as high speed, delay sensitive, IoT), and the corresponding β , δ and efficiency level will be optimized at the factory level and set by the operator admin.

Figure 4 shows an example 400 of how to compute the optimum values for β and δ . A range of values is suggested 402 for both β and δ . The mid values 404 of β and δ are obtained. The optimum value for δ is determined 406 first by evaluating the performance of the RL agent using the mid value for β . The lower and upper limit of the range of values for δ are also evaluated to determine if the mid value for δ needs to be adjusted 408 to be closer to the lower or upper limit to reach the optimum value. During the computation of the optimum value for δ , the mid value of β is selected for β . After the optimum value for δ is obtained using 50 episodes or until the DQN agent rewards converge (whichever is less), this process is repeated 410 to obtain the optimum value for β . During the computation of the optimum value for β , the optimum value of δ is selected for δ . The performance of the RL agent is now evaluated with a fixed value of δ (which is the optimum value from previous step 410) and the lower and upper limits of the range of values for β in the next evaluation of 50 episodes or until the DQN agent rewards converge (whichever is less). Similar steps are repeated to determine the optimal value of β for the fixed optimal value of δ .

The hyperparameters are tuned periodically or based on event/threshold. The event will be generated by the DPI/NGFW. If the event is detected by the DPI or NGFW for more than 3 times or periodically (whichever is earlier), the hyperparameters will need to be retuned. The period is determined based on the frequency of significant changes observed in the network dataset and the criticality of maintaining optimal performance. Generally, a period is set to one week, but this can be adjusted according to the specific needs and variability of the network environment. This is measured by rerunning the exploration and exploitation phase for the current network state.

The hyperparameters of the production CNN model will be tuned by the RL agent based on the target accuracy obtained during the exploration phase of the RL agent. If the CNN model has never reached the target accuracy for a certain time threshold, the RL agent and the CNN model has to be retrained with a new dataset from the current network environment.

Models that can be used for network traffic classification, depending on the requirements of the application, include CNN, Gaussian Naïve Bayes, Classification and Regression Decision Tree, Logistic Regression, C-Support Vector Machine, Light Gradient Boosting, AlexNet and LSTM model.

The contents of the following documents are incorporated herein by reference:

1. Kaur, G., Lashkari, A. H., & Rahali, A. (2020, August). Intrusion traffic detection and characterization using deep image learning. In 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCoM/CyberSciTech) (pp. 55-62). IEEE.
2. Talaei Khoei, T., & Kaabouch, N. (2023). A comparative analysis of supervised and unsupervised models for detecting attacks on the intrusion detection systems. *Information*, 14(2), 103.
3. Lim, H. K., Kim, J. B., Kim, K., Hong, Y. G., & Han, Y. H. (2019). Payload-based traffic classification using multi-layer lstm in software defined networks. *Applied Sciences*, 9(12), 2550.

Claims

1. A computer implemented method performed in a device or virtual appliance configured for reinforcement learning, the method comprising, under control of at least one processor or at least one virtual processor:
 - storing an untrained CNN classifier model for network traffic classification,
 - storing an action space of a plurality of combinations of hyperparameters,
 - for each of a plurality of episodes:
 - initialising a plurality of episode variables,
 - for each of a plurality of iterations:
 - training the CNN classifier model using a state including the respective hyperparameters from the action space chosen by a reinforcement learning agent for that iteration, evaluating the accuracy of the trained CNN classifier model,
 - evaluating the reward for that iteration using a quantized reward function with a negative reward for iterations with a lower accuracy compared to the previous iteration,
 - storing at least the reward and the state in replay memory, and
 - determining a state for the next iteration based on predetermined criteria,
 - increasing the replay memory, and
 - updating the replay memory with the action space from the iterations
 - for that episode,
 - training the reinforcement learning agent based on the replay memory.
2. The method according to claim 1, wherein the negative reward for iterations with a lower accuracy compared to the previous iteration is exponentially negative.
3. The method according to claim 2, wherein the quantized reward function has a positive reward for an accuracy improvement below a threshold compared to the previous iteration.

4. The method according to claim 3, wherein the quantized reward function has a higher positive reward for an accuracy improvement above the threshold, compared to the positive reward for the accuracy improvement below the threshold.
5. The method according to any preceding claim, wherein the quantized reward function is based on a linear factor (β) representing how much to reward, and when (δ) to provide the higher reward fraction of positive reward.
6. The method according to claim 5, further comprising determining an optimal δ .
7. The method according to claim 6, further comprising determining an optimal β based on the highest accuracy, using the optimal δ .
8. The method according to any preceding claim, wherein the quantized reward function is based on

$$\mathcal{R}_i = \begin{cases} \beta \times \tau_i & , (\tau_i - \tau_{i-1}) \geq \delta, \\ \left(\frac{\beta \times \tau_i}{2}\right) & , 0 \leq (\tau_i - \tau_{i-1}) < \delta, \\ -\left(\frac{\tau_i + \tau_{i-1}}{\tau_i}\right)^2 & , \tau_i < \tau_{i-1}. \end{cases} .$$

9. The method according to any preceding claim, wherein determining a state for the next iteration based on predetermined criteria is done by computing an epsilon value using an Epsilon Greedy Algorithm for the current iteration.
10. The method according to any preceding claim, wherein training the reinforcement learning agent is done using a Bellman Equation.
11. The method according to claim 10, wherein the Bellman Equation is

$$y_i = r_i + \gamma \cdot \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta)$$

12. The method according to claim 10, wherein the discount factor (γ) is about 0.95.

13. The method according to any preceding claim, wherein the hyperparameters that are optimised in the action space include those shown in the below table:

Layer	Parameters	Value Range of the parameters
Convolution Layers		
Conv2D	Number of Filters (Not for DepthwiseConv2D) Kernel Size Strides Padding	$\in \{16, 32, 64, 96, 128, 160, 192, 224, 256\}$ $\in \{3, 5, 7, 9, 11\}$ $\in \{1, 2, 3\}$ $\in \{\text{'same'}, \text{'valid'}\}$
Conv2DTranspose		
SeparableConv2D		
DepthwiseConv2D		
Pooling Layers		
MaxPooling2D	Pool size	$\in \{2, 3, 4, 5\}$
AveragePooling2D	Strides	$\in \{2, 3, 4, 5\}$
GlobalMaxPooling2D	Padding (Not for GlobalAveragePooling2D and GlobalMaxPooling2D)	$\in \{\text{'same'}, \text{'valid'}\}$
GlobalAveragePooling 2D		
Regularization Layers		
Dropout	Dropout rate	$\in (0, 1)$
Flatten Layer		
Flatten	(No parameter values to be changed)	(No parameter values to be changed)
Dense Layer		
Dense	Number of neurons Activation function	$\in \{8, 16, 32, 64, 128, 256, 512, 1024\}$ $\in \{\text{'sigmoid'}, \text{'tanh'}, \text{'relu'}, \text{'elu'}, \text{'selu'}, \text{'swish'}\}$
Optimizer		
Optimizer	Type Learning rate	$\in \{\text{'Adam'}, \text{'SGD'}, \text{'RMSprop'}\}$ $\in (0, 1)$

14. The method according to any preceding claim, wherein the hyperparameters that are set with predetermined values include those shown in the below table:

Layer	Parameters	Values of the parameters preset
Convolution Layers		
Conv2D	Number of Filters	$\in \{32, 64\}$
	Kernel Size	$\in \{3, 5, 7\}$
	Strides	$\in \{1\}$
	Padding	$\in \{\text{'same'}\}$
Pooling Layers		
MaxPooling2D	Pool size	$\in \{2\}$
	Strides	$\in \{2\}$
	Padding	$\in \{\text{'same'}\}$
Regularization Layers		
Dropout	Dropout rate	$\in \{0.25, 0.40, 0.50\}$
Flatten Layer		
Flatten	(No parameter values to be changed)	(No parameter values to be changed)
Dense Layer		
Dense	Number of neurons	$\in \{1024\}$
	Activation function	$\in \{\text{'relu'}\}$
Optimizer		
Optimizer	Type	$\in \{\text{'Adam'}\}$
	Learning rate	$\in \{0.0001\}$

15. The method according to claim 12, wherein the search space is defined by combinations of hyperparameters that are to be optimised and not the hyperparameters that are set.

16. The method according to any preceding claim, wherein the search space is not narrowed during iterations.

17. The method according to any of claims 5 to 8, wherein β is between 2 to 10 and δ is between 0.05 and 0.1.
18. The method according to any of claims 5 to 8, wherein β is about 6 and δ is about 0.05.
19. The method according to any preceding claim, further comprising using the trained CNN classifier model to classify network traffic in inference mode, based on the optimised hyperparameters from the reinforcement learning agent.
20. The method according to claim 19, further comprising periodically tuning the hyperparameters in response to a change in the network dataset.
21. A computer implemented method performed in a device or virtual appliance configured for traffic classification, the method comprising, under control of a processor of the communication server apparatus: using a CNN classifier trained with hyperparameters optimised according to any of claims 1 to 18 to classify traffic on a data network.
22. The method according to claim 21, further comprising periodically tuning the hyperparameters in response to a change in a dataset of the data network.
23. A computer program or computer program product comprising instructions for implementing the method according to any preceding claims.
24. A computer program carrier carrying the computer program or computer program product according to claim 23, wherein the computer program or computer program product is one of an electronic signal, optical signal, radio signal or computer-readable storage medium.
25. A non-transitory tangible computer-readable storage medium storing the computer program or computer program product according to claim 23.

26. A device or virtual appliance configured for traffic classification, comprising:
at least one memory configured to store a trained CNN classifier model, and
at least one processor configured to execute instructions stored in one or more of the at least one memory to classify traffic on a data network, using the trained CNN classifier, based on a combination of hyperparameters optimised according to the method of any of claims 1 to 18.
27. A user device, comprising:
an input/output module configured to receive classified traffic from a data network, wherein the traffic is classified using a trained CNN classifier, based on a combination of hyperparameters optimised according to the method of any of claims 1 to 18, and
at least one user interface configured to display information according to the classified traffic data received from the data network.
28. A device or virtual appliance configured for reinforcement learning, comprising:
at least one memory configured to store an untrained CNN classifier model, and
an action space of a plurality of combinations of hyperparameters, and
at least one processor configured to execute instructions stored in one or more of the at least one memory to:
for each of a plurality of episodes:
initialise a plurality of episode variables,
for each of a plurality of iterations:
train the CNN classifier model using a state including the respective hyperparameters from the action space chosen by a reinforcement learning agent for that iteration,
evaluate the accuracy of the trained CNN classifier model,
evaluate the reward for that iteration using a quantized reward function with a negative reward for iterations with a lower accuracy compared to the previous iteration,
store at least the reward and the state in replay memory, and

determine a state for the next iteration based on
predetermined criteria,
increase the replay memory, and
update the replay memory with the action space from the iterations for
that episode,
train the reinforcement learning agent based on the replay memory.

29. A device or virtual appliance configured for traffic classification, comprising:
at least one memory configured to store a trained CNN classifier model, and
at least one processor configured to execute instructions stored in one or more
of the at least one memory to:

use the trained CNN classifier to classify traffic on a data network, based
on an optimised combination of hyperparameters,
periodically tune the hyperparameters in response to a change in the
network dataset for each of a plurality of episodes wherein periodically tuning
includes:

train the CNN classifier model using a state including the
respective hyperparameters from the action space chosen by a
reinforcement learning agent,
evaluate the accuracy of the trained CNN classifier model,
evaluate the reward using a quantized reward function with a
negative reward for lower accuracy,
store at least the reward and the state in replay memory,
increase the replay memory,
update the replay memory with the action space from the
iterations for that episode, and
train the reinforcement learning agent based on the replay
memory.

30. The device or virtual appliance in claim 29 further comprising an admin interface
configured for setting operating parameters for the reinforcement learning agent.