Practical 2: Secure Software Platform Design and Implementation

Student: Maitham Al-rubaye
Supervisor: Atakan Aral
University of Vienna - 2024

1 Introduction

In Practical 1, we dove into the core tech and methods behind streaming data across multiple locations, with a special look at a simulation that had electric cars (EVs) and edge computing points. Our first research shone a light on how key it is to handle data on the fly and the power of being able to guess what's next if you want stuff to run when things keep changing. The hands-on work gave us a real eye-opener about the hurdles when dealing with fast-moving data flows and how critical it is that our models are both on point and up-to-date in systems that are spread out.
Expanding on the groundwork established in Practical 1, Practical 2 unfolds into two linked segments that deepen our grasp and skills in the world of safe effective data flow within IoT settings. The initial part zeroes in on setting up secure data transfer. This part includes launching four IoT gadgets that keep an eye on the weather, guaranteeing data safety via HTTPS protocols, a private network, and robust authentication using keys and IDs. To top it off, we're tracking system performance and use with Grafana, which lets us see all the data and resource use in real time. In Practical 2's second section, we compare hardware accelerators focusing on Google Coral USB against other stand-alone devices, to see how well they support future AI tasks like spotting unusual activity and identifying security risks. We aim to test how these tools perform when running complex AI models that can find anomalies and also take fitting actions in response.
The expected results of Practical 2 to boost our grasp of how to handle data and how AI can make system security and performance better. This step fits well with our current studies and prepares us for deeper dives into how to blend AI into real-world tools for things like the Internet of Things (figure 1).

Keywords:
Secure Data Transmission, IoT Security, Real-Time Data Processing, HTTPS, Private Networks, Data Visualization, Grafana, Hardware Accelerators, Google Coral USB, Anomaly Detection, AI in Security

2 Part 1: Ensuring Safe Data Sharing

2.1 Insight into IoT Arrangement and Security Measures

In this section, we concentrate on building a strong setup to ensure the safe sharing of data through IoT gadgets tailored for keeping track of the environment.
Our arrangement makes use of sensors controlled by microcontrollers (DHT11, DHT22 DHT20) paired with ESP8266 Mini D1 Pro boards, with a sharp focus on securing data and processing it.

Our technical setup incorporates:
- Sensors: DHT11, DHT22, and DHT20 for diverse environmental sensing.
- Controllers: ESP8266 Mini D1 Pro, providing reliable and efficient control over the sensors.
- Communication Protocol: Secure HTTPS, ensuring encrypted data transmissions.
- Security Features: Unique device identifiers and secret keys for enhanced data security.

Pseudo code for Setting Things Up:

```
INITIALIZE IoT Devices with specifications:
  - Devices: DHT11, DHT22, DHT20
  - Controllers: ESP8266 Mini D1 Pro
  - Communication Protocol: HTTPS
  - Security: Unique Device ID, Secret Key

SETUP Network:
  - Configure private network settings
  - Establish secure HTTPS connections for
     data transmission
```

2.2 Gathering and Sending Data

IoT gadgets gather data, add the current time to it, and send it safely using HTTPS. This keeps the info secret and unaltered. The utilization of secret keys during data transmission provides an additional layer of security, effectively preventing unauthorized access (the data of unauthorized devices will be dropped automatically)(figure 2).

Pseudo code Data Collection and Transmission:

```
FUNCTION Collect_Data:
  - READ temperature and humidity from DHT
     sensors
  - GET current timestamp from NTP client
  - FORMAT data as JSON:
    {
      "temperature": temperature_value,
      "humidity": humidity_value,
      "timestamp": current_time,
      "deviceID": device_id,
      "secretKey": secret_key
    }
  - SEND JSON data to server via HTTPS POST
     request
```

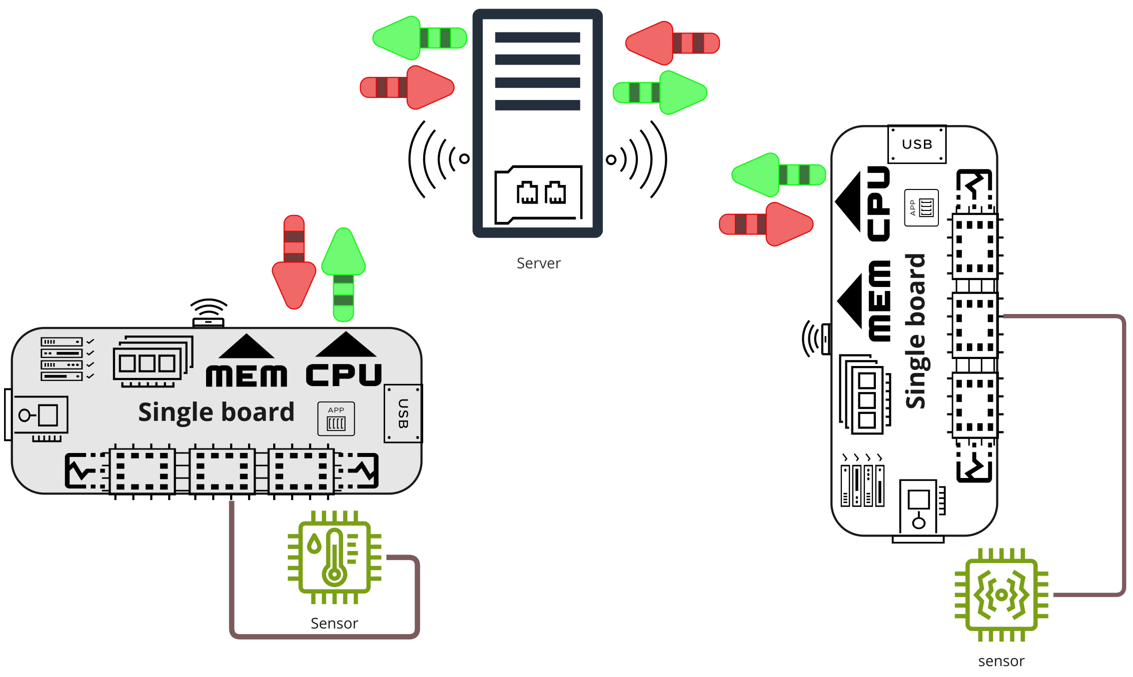


Figure 1: Setup of couple of EDGE NODES and main SERVER

2.3 Server-Side Data Handling

A Flask application on the server side receives the data, checks the secret key, and once confirmed, it is stored in InfluxDB for persistent storage and forwarded to Kafka for further processing. This dual handling ensures that the data remains secure and also ready for subsequent analytical steps.

Pseudo code for Server-Side Data Handling:

```
FUNCTION Handle_Incoming_Data (HTTP POST
    endpoint):
  - RECEIVE JSON data
  - VALIDATE secretKey against stored value
  IF valid:
    - EXTRACT temperature, humidity,
      deviceID, timestamp from JSON
    - CALL Store_In_Database
    - CALL Produce_To_Kafka
  ELSE:
    - RETURN "Unauthorized Access"
```

2.4 Keeping Eyes on System Performance with Grafana

For real-time system monitoring, Grafana plays a critical role by presenting data trends and the status of various devices. It enables immediate understanding of the environment and how well the system is functioning (figure 3).

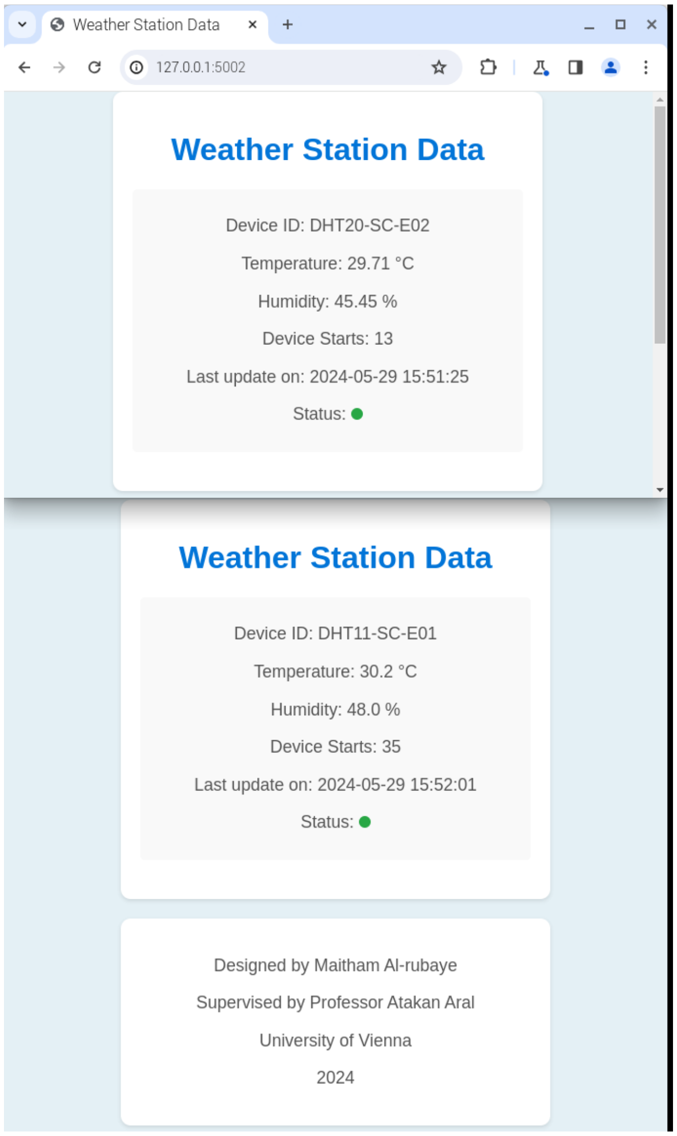
Pseudocode for Grafana Configuration:

```
CONFIGURE Grafana:
  - CONNECT to InfluxDB database
  - SET UP dashboards for real-time
    monitoring:
    - DISPLAY temperature and humidity
      trends
    - SHOW device operational status and
      data reception timestamps
```

2.5 Overview

This segment of Practical 2 showcases how we created a sturdy, dependable, and speedy IoT setup for environmental tracking. We've designed a data transmission framework that prioritizes speed and security, safeguarding the integrity and confidentiality of the data as it travels through the network.

Our system's efficacy is evident in its performance and also in its accessibility. All monitoring interfaces and operational scripts have been documented and are readily available in the Practical 2[1] repository on GitHub. This ensures that users can easily access tools for real-time performance monitoring and historical data analysis, providing a transparent view into the system's functionality and its data handling prowess.



Weather Station Data

127.0.0.1:5002

Weather Station Data

Device ID: DHT20-SC-E02

Temperature: 29.71 °C

Humidity: 45.45 %

Device Starts: 13

Last update on: 2024-05-29 15:51:25

Status: ●

Weather Station Data

Device ID: DHT11-SC-E01

Temperature: 30.2 °C

Humidity: 48.0 %

Device Starts: 35

Last update on: 2024-05-29 15:52:01

Status: ●

Designed by Maitham Al-rubaye

Supervised by Professor Atakan Aral

University of Vienna

2024

Figure 2: GUI data gathering

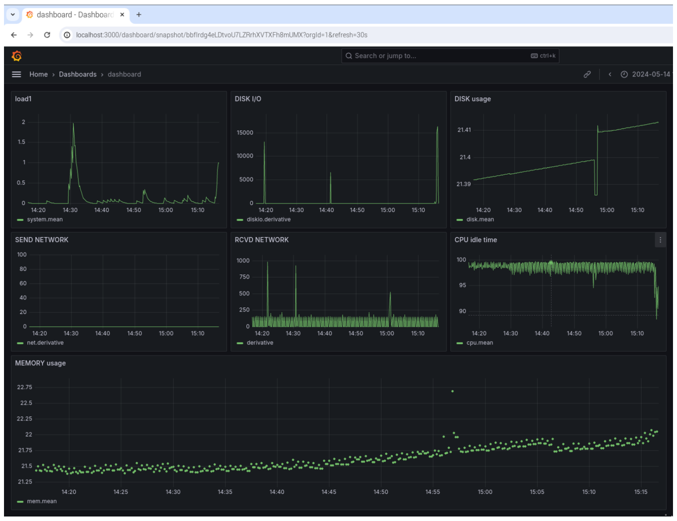


Figure 3: Resources usage

3 Part 2: Hardware Acceleration for AI Applications

3.1 Introduction

In continuation of Practical 1, which investigated the dynamics of data streaming across a network of smart cars, Practical 2 progresses into examining the real-time processing capabilities of edge devices. Our focus is particularly honed in on enhancing network security within EDGE environments through advanced anomaly detection. This examination was conducted using a Raspberry Pi 5 equipped with 8GB of RAM, exploring its compatibility with both standalone operations and integration with a Google Coral USB accelerator. The main goal of this experiment was to test if the Raspberry Pi could effectively manage and execute complex AI models designed to autonomously detect and respond to unusual network activities without compromising the network's overall performance. Initially, we ventured to deploy sophisticated AI models such as UNET, ICNet, and EfficientNet. While these models demonstrated high accuracy, they imposed substantial computational demands on the Raspberry Pi, resulting in significant latency issues that were not conducive to real-time anomaly detection. When we hit the challenge of balancing speed with accuracy, a decision emerged to make the models simpler. This adjustment achieved a faster operational tempo but at the cost of reduced precision. However, further complexities arose when attempting to deploy these streamlined models on the Google Coral TPU. We encountered compatibility issues with the latest versions of Python and AI libraries, necessitating a regression to older software versions facilitated by pyenv to accommodate the hardware constraints imposed by the Coral TPU. This part of the work showed us how current hardware like the Google Coral TPU isn't keeping up with the newest stuff in AI. This is a big deal because it gets in the way of using the freshest AI tech that needs the latest software tricks to run best. Our ultimate objective was to develop an AI model capable of predicting street elements from the Cityscapes dataset—a critical component in autonomous vehicle navigation. The model needed to operate under real-world conditions where swift data processing and immediate response capabilities are very important.

3.2 Experiment Setup - Hardware and Software Setup

In our quest to enhance the real-time processing capabilities of edge devices for security applications, we employed a Raspberry Pi 5 equipped with 8GB of RAM. This device was chosen for its robust performance in on-the-fly computational tasks. Our experiment was structured around two primary hardware setups:
Raspberry Pi on its Own:
The device worked on its own using just its main processing unit to handle data.
Raspberry Pi Plus Google Coral USB:
This arrangement had a Google Coral TPU booster that sped things up by taking on particular tasks the processor

usually does.
When it came to setting up Google Coral TPU, which has limited compatibility with newer Python versions and libraries, we utilized pyenv to manage different Python environments. This allowed us to use older, Coral-compatible Python versions and libraries without disrupting the broader software ecosystem.
Model Development:
Making the Model Early trials to apply sophisticated models like UNET[4][5][6], ICNet[3], MobileNetV2, and EfficientNet[2] yielded high performance but faced slow processing times on the standalone Raspberry Pi. A high-spec laptop hosted these models for development and training tapping into its advanced processing abilities. This was followed by tweaks to make them suitable for use on edge devices. Yet, owing to inefficiencies in performance, attention turned to simpler models. These decreased accuracy but boosted the speed of analysis to meet the real-time processing needs of the application.
Model Adaptation for Hardware Acceleration:
Refitting Models for Increased Speed with Specialized Hardware Moving to Google Coral called for updates to make the models work with the older software platforms that the TPU operates on. This included:
- Making the model simpler to work with the TPU's operational limits.
- Using TensorFlow Lite to change and enhance the models making sure they were simple enough to run well on the TPU.
Dataset:
Dataset For training and testing the models, we chose the Cityscapes dataset, renowned for its rich visual information of urban scenes. This dataset is particularly suited for developing AI systems for autonomous driving, providing extensive data crucial for training models to identify and interpret road elements.
Performance Metrics:
We focused on these main metrics to see how the experiment did:
- Inference Time: We assess this to gauge each arrangement's capability to process data in real-time.
- Accuracy and Loss: We evaluate these to understand the balance between a simple model and its precision in forecasting results.
The arrangement we've created offers a detailed plan for investigating how well using high-end hardware accelerators such as Google Coral, with Raspberry Pi works for edge AI tasks.

3.3 Creating and Training the Model

Choosing and Crafting the Model
Complex models often stumble when they run in real-time settings. So, we went with a UNET design to strike a good balance between quickness and getting it right. We picked this model because it's great at sorting out different parts of a picture, which is just what we need for spotting things on the road in the Cityscapes collection of images. We tweaked the setup of the architecture to cut down reduce computational load while maintaining sufficient detail to perform the required task effectively.
Setting Up for Development:
At first, we created the models on a high-performance laptop that had strong computing power. This made it possible to test and improve the models fast. We chose TensorFlow as our primary machine learning framework due to its robust support for image recognition networks and compatibility with TensorFlow Lite, which is crucial for deploying models on more compact devices like the Google Coral.
How We Trained the Models?
Data Preparation: In this step we processed the images from the Cityscapes dataset by resizing, normalizing, and augmenting them to enhance the model's robustness against various lighting conditions and resolutions.
Model Development: To train the UNET model, we used both the Adam optimizer and the Sparse Categorical Crossentropy loss function. We ran several training epochs, we have implemented early stopping and checkpoints to save the best-performing models based on validation loss.
Size Reduction and Speed Increase (Post-training): We utilized TensorFlow Lite to compress the model, significantly reducing its size and increasing its processing speed without substantial losses in accuracy. This optimization was important for efficient deployment on the Google Coral device.
Dealing with Compatibility Issues
While transitioning to Google Coral, we ran into several hurdles due to discrepancies in Python and TensorFlow versions. We employed pyenv to manage different Python environments effectively, allowing us to integrate older, TPU-compatible libraries without disrupting our primary development environment[8].
Model Training on Colab
Google Colab proved to be a savior when we considered the Raspberry Pi's limited processing abilities and our need to speed up the training. It provided access to more powerful computational resources, such as GPUs and TPUs, which accelerated the training process and offered the flexibility to experiment with various model configurations. These tools helped us train our models quicker and let us play around with various model setups[7].

3.4 Inference Performance

Performance Evaluation Metrics
We looked at two main scores to measure how well the models worked:
Inference Time: This metric was key in determining the real-time applicability of the models. Quick inference times are essential for enabling smart cars to react promptly to dynamic road conditions (exactly as in threads detection which is the goal of our study).
Accuracy: Even though we made the model simpler to make it faster, it was still super important for it to pick

up on unusual things and recognize different parts of the road without messing up.

Comparison of Model Processing Speeds

We timed the model processing speeds in two different setups:

Standalone Raspberry Pi: Here, the model ran on the Raspberry Pi's CPU. This setup gave us a basic level of performance without any extra speed boosts from other hardware (Figure 4).

Raspberry Pi with Google Coral: By bringing in the Google Coral TPU, we aimed to cut down the processing times a lot. The TPU takes over heavy-duty tasks so we expected things to go faster (figure 5).

From what we saw, adding Google Coral sped things up, though we couldn't test every model because some weren't compatible. But the UNET model worked faster with the TPU showing us that these dedicated accelerators have a lot of promise for portable computing situations.

Considering Precision

In efforts to speed up the processing time, the model was simplified, which resulted in some give-and-take regarding precision. We measured the model's precision by comparing it to the Cityscapes dataset and using clear-cut metrics for image segmentation like Intersection over Union (IoU) and pixel accuracy.

Discussion on Results

The inference tests clearly demonstrated that while hardware accelerators like the Google Coral can significantly enhance performance, they also introduce complexities related to software compatibility. The experiments highlighted the importance of carefully choosing model architectures that balance computational power with the need for rapid processing. This balance is crucial in deploying AI-driven applications in real-world settings where both speed and accuracy are pivotal.

3.5 Results and Discussion - Summary of Findings

Our experiments offered valuable understanding about the balance among model complexity how quick a model can infer, and its precision in edge computing settings that include hardware accelerators:

Inference Speed: When we added the Google Coral USB accelerator to the setup with the Raspberry Pi, the time it took for the simplified UNET model to analyse data saw a big drop. This cut in time shows the good things that can come from using extra hardware to help out if fast answers are needed like in anomalies detection.

Accuracy Trade-offs: But when we had to make the model simpler so the hardware could handle it, we did see the accuracy go down. We looked at this by using scores like IoU and pixel accuracy, which showed us how hard it is to get the balance right between being fast and being correct when creating models.

Resource Utilization: Monitoring of resources showed the Coral TPU took over computations from the Raspberry Pi's CPU. However, the system still used a lot of memory and energy.

Faced Obstacles

Compatibility Problems: Our main struggle centered around getting Google Coral to work with the newest Python libraries and AI technologies. We had to stick to older software versions, which made development trickier and limited our access to cutting-edge modeling tools.

Model Streamlining: To fit with Google Coral, we had to trim down our model's complexity. This cutback meant we had to give up some high-level features, which lessened the model's overall performance.

Possible Fixes and Upcoming Tasks

Investigating Newer Accelerators: Upcoming studies should focus on different hardware accelerators that work with modern software setups. These options could lead to higher efficiency without simplifying the model.

Refining Models: Digging deeper into ways to condense and fine-tune models could result in them running fast and fitting well with today's hardware accelerators.

Mixing Model Types: Using a combined system of both intricate and straightforward models may strike the right balance. Use detailed models for the first round of learning and easier models for making quick decisions.

3.6 Code Implementation and Pseudo code

This section provides a concise pseudo code representation of the key components involved in the development and deployment of the models discussed in the experiment. The actual code and detailed documentation are available in the official Practical 2 repository, ensuring that the methodologies are accessible and reproducible.

Pseudo code for Model Training and Inference:

```
FUNCTION TrainModel
    INPUT: Dataset, ModelArchitecture
    OUTPUT: TrainedModel

    Initialize Model based on Architecture
    FOR each Epoch in TrainingCycles
        FOR each Batch in Dataset
            Perform Forward Pass
            Compute Loss
            Perform Backward Pass to Update
                Model Weights
        END FOR
        Evaluate Model on Validation Set
        IF Performance Improvement is
            Minimal
            Terminate Training Early
        END IF
    END FOR
    RETURN TrainedModel
END FUNCTION
```

Pseudo code for Model Quantization:

```
FUNCTION QuantizeModel
    INPUT: TrainedModel
    OUTPUT: QuantizedModel

    DEFINE QuantizationParameters
    APPLY Quantization to TrainedModel using
         Parameters
    RETURN QuantizedModel
END FUNCTION
```

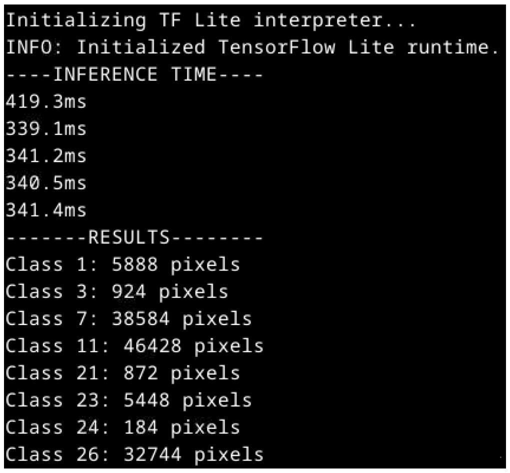
Pseudo code for Running Inference on Edge Devices:

```
FUNCTION PerformInference
    INPUT: Model, InputData
    OUTPUT: InferenceResults

    Load Model into Device Memory
    FOR each Input in InputData
        Preprocess Input to Match Model
            Requirements
        Run Model on Input
        Postprocess Output for Desired
            Format
        Store
        Inference Results
    END FOR
    RETURN InferenceResults
END FUNCTION
```

Pseudo code for Resource Monitoring

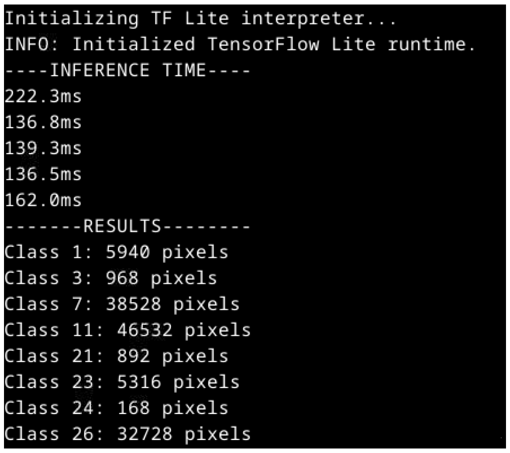
```
FUNCTION MonitorResources
    INPUT: None
    OUTPUT: ResourceUsageData

    WHILE Model is Running
        Check CPU Usage
        Check Memory Usage
        Check Disk IO
        Log Resource Usage
    END WHILE
    RETURN ResourceUsageData
END FUNCTION
```

Discussion on Code Implementation
The pseudo code outlined above represents a generalized flow of the processes involved in training, quantizing, and deploying models on hardware accelerators like Google Coral. These procedures are critical for understanding the interaction between software and hardware in the context of AI-driven applications on edge devices.
By maintaining simplicity and clarity in the pseudo code, the aim is to make the underlying algorithms and their implementation accessible to developers and researchers. This approach facilitates a deeper understanding of the practical challenges and considerations involved in deploying AI models in resource-constrained environments.



```
Initializing TF Lite interpreter...
INFO: Initialized TensorFlow Lite runtime.
----INFERENCE TIME----
419.3ms
339.1ms
341.2ms
340.5ms
341.4ms
-------RESULTS--------
Class 1: 5888 pixels
Class 3: 924 pixels
Class 7: 38584 pixels
Class 11: 46428 pixels
Class 21: 872 pixels
Class 23: 5448 pixels
Class 24: 184 pixels
Class 26: 32744 pixels
```

Figure 4: Performance on EDGE CPU



```
Initializing TF Lite interpreter...
INFO: Initialized TensorFlow Lite runtime.
----INFERENCE TIME----
222.3ms
136.8ms
139.3ms
136.5ms
162.0ms
-------RESULTS--------
Class 1: 5940 pixels
Class 3: 968 pixels
Class 7: 38528 pixels
Class 11: 46532 pixels
Class 21: 892 pixels
Class 23: 5316 pixels
Class 24: 168 pixels
Class 26: 32728 pixels
```

Figure 5: Performance on EDGE TPU

4 Conclusion

Practical 2 extends the foundational research begun in Practical 1, delving into distributed data streaming and the application of edge computing in smart automotive systems and IoT devices. The project was divided into two distinct but interconnected parts, each focusing on different aspects of edge computing and hardware acceleration.

Part 1: Secure Data Transmission
In the first part of the project, the focus was on establishing secure data transmission protocols within a network of IoT devices used for weather monitoring. This involved setting up a robust infrastructure using HTTPS protocols, private networking, and secure authentication mechanisms to ensure data integrity and security. The system's performance and resource usage were monitored using Grafana, which provided real-time insights into system operations. The outcome demonstrated the effectiveness of the secu-

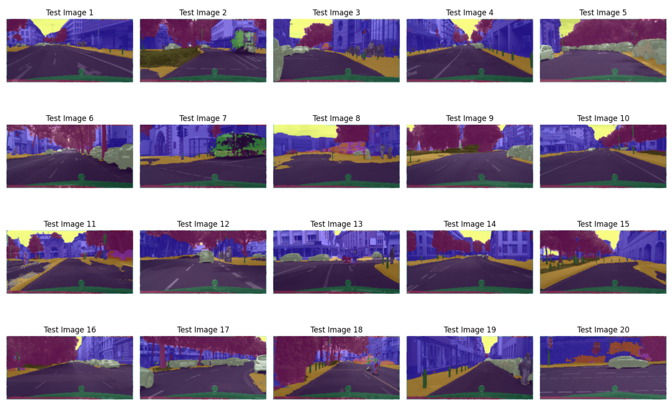


Figure 6: Server output

rity measures implemented, highlighting their importance in maintaining data confidentiality and integrity in IoT environments.

Part 2: Hardware Acceleration for AI Applications

The second part explored the use of hardware accelerators, specifically Google Coral, to enhance the processing capabilities of edge devices like the Raspberry Pi in smart car networks. The experiments focused on evaluating the impact of Google Coral on the efficiency and speed of AI models designed for detecting road anomalies and managing network security. Although the hardware accelerator significantly improved inference times, compatibility issues with newer software versions posed challenges, leading to a reliance on older technology stacks. The experiments underscored the trade-offs between model complexity and real-time processing needs, emphasizing the need for balance in choosing suitable model architectures and hardware configurations.

Practical 2 successfully demonstrated the critical role of secure data transmission and efficient data processing in IoT and smart automotive systems. The project not only highlighted the technical challenges associated with implementing advanced AI models on edge devices but also showcased potential solutions through hardware acceleration.

Looking forward, the project suggests a continued exploration of alternative hardware accelerators and advanced model optimization techniques. Such endeavors could potentially overcome current limitations and pave the way for more sophisticated implementations. The possibility of hybrid architectural approaches, combining both local and centralized processing, might also offer a promising solution to the challenges faced in deploying real-time AI applications efficiently.

Overall, Practical 2 builds on the insights from Practical 1 and pushes the envelope on integrating edge computing with advanced AI capabilities in real-world applications, setting the stage for future innovations in smart technology.

References

[1] Main Repository: `https://github.com/Maitham16/practical_2`

[2] Cityscapes ENet Model: `https://github.com/Maitham16/cityscapes_ENet`

[3] Cityscapes ICNet Model: `https://github.com/Maitham16/cityscapes_ICNet`

[4] Cityscapes UNet Model: `https://github.com/Maitham16/cityscapes_unet`

[5] Cityscapes UNet Accurate Model: `https://github.com/Maitham16/cityscapes_unet_accurate`

[6] Cityscapes UNet Enhanced Model: `https://github.com/Maitham16/cityscapes_unet_enhanced`

[7] Compiler version: `https://coral.ai/docs/edgetpu/compiler/#compiler-and-runtime-versions`

[8] Supported layers: `https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview`