Bachelor's thesis

Information and Communications Technology

2025

Atte Mäki-Kerttula

A Workflow for Training Gas Classification Models from BME688 Sensor Data

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2025 | 28 pages

Atte Mäki-Kerttula

A Workflow for Training Gas Classification Models from BME688 Sensor Data

Digitalization in the brewing industry has increased the need for improved process monitoring and reliability. This thesis examines the use of artificial intelligence for detecting and classifying gases produced during fermentation, with the goal of developing a workflow for training machine learning models from sensor data and evaluating potential deployment platforms.

Data were collected using the Bosch BME688 gas sensor, and the outputs were preprocessed with the Bosch BME AI-Studio and converted into structured datasets through a Python-based pipeline. Model development was conducted with the PyTorch deep learning framework, incorporating data preprocessing, dataset creation, and neural network training using the One-Cycle learning rate strategy.

The resulting models successfully classified multiple fermentation-related gases and met the main requirements of flexibility and platform independence. The developed workflow forms a basis for future improvements, such as direct raw data processing and expanded hardware support and demonstrates that machine learning can be effectively applied to gas monitoring in smart brewery systems.

Keywords:

artificial intelligence, machine learning, neural network

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2025 | 28 sivua

Atte Mäki-Kerttula

Työnkulku BME688-anturidatasta koulutettaville kaasuluokittelumalleille

Digitalisaation lisääntyminen panimoteollisuudessa on lisännyt tarvetta parannetulle prosessien seurannalle ja luotettavuudelle. Tässä opinnäytetyössä tarkasteltiin tekoälyn käyttöä fermentaation aikana muodostuvien kaasujen havaitsemiseen ja luokitteluun. Opinnäytetyön tavoitteena oli kehittää työprosessi, joka mahdollistaa koneoppimismallien opettamisen anturidatasta sekä niiden soveltuvuuden arvioinnin eri käyttöympäristöissä.

Aineisto kerättiin Bosch BME688 -kaasuanturilla, ja se esikäsiteltiin Bosch BME AI-Studiolla ennen muuntamista rakenteiseksi tietoaineistoksi Python-pohjaisen käsittelyvaiheen avulla. Mallien kehitys toteutettiin PyTorch-syväoppimiskehyksellä. Työvaiheisiin sisältyivät datan esikäsittely, aineiston muodostaminen sekä neuroverkon koulutus One-Cycle-oppimisnopeusmenetelmää käyttäen.

Tuloksena syntyneet mallit luokittelivat onnistuneesti useita fermentaatioon liittyviä kaasuja ja täyttivät tärkeimmät joustavuuden ja alustariippumattomuuden vaatimukset. Kehitetty työprosessi luo perustan jatkokehitykselle, kuten raakadatan suoralle käsittelylle ja laajennetulle laitteistotuelle. Opinnäytetyö osoitti, että koneoppimista voidaan tehokkaasti hyödyntää kaasujen seurantaan älykkäissä panimojärjestelmissä.

Asiasanat:

tekoäly, koneoppiminen, hermoverkko

Contents

Figures

List of abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
EON	Edge Optimized Neural
ESP32	Espressif Systems 32-bit System-on-Chip Microcontroller
FID	Flame Ionization Detector
GPU	Graphics Processing Unit
HDF5 (.h5)	Hierarchical Data Format version 5
IoT	Internet of Things
ML	Machine Learning
MOS	Metal Oxide Semiconductor (sensor type)
PID	Photoionization Detector
Ppb	Parts per Billion
PyTorch	Python-based open-source deep learning framework
ReLU	Rectified Linear Unit activation function
SDK	Software Development Kit
SoC	System-on-Chip
ULP	Ultra-Low Power
VOCs	Volatile Organic Compounds
VSCs	Volatile Sulphur Compounds
Wi-Fi	Wireless Fidelity

1 Introduction

Smart brewery is a concept in which technology is used to further improve the effectiveness of the brewing process beyond what is possible using just the traditional methods and procedures. Smart brewery integrates automated, Internet of Things (IoT) devices and data analytics to control critical process parameters such as temperature and fermentation in real time. This higher level of control over the brewing process improves consistency, reduces human error and enables especially smaller breweries to operate much more reliably. The integration of more IoT devices to the brewing process can also lower energy and resource consumption and lessen the chances of failure in the brewing batches of individual products.

The aim of this thesis is to evaluate usage of different AI tools to make an AI model to be used in a smart brewery system. The models are used to evaluate the measurement results of the gases produced by the brewing process and monitor the progression of the brewing process.

The brewing process releases CO2 and different volatile organic compounds (VOC). In this thesis, these VOCs will be measured by a sensor, and the sensor data will be collected. This sensor data will be used to train different AI models with different AI tools and then the resulting AI models will be evaluated and compared to each other. The data collection will be conducted with a Bosch BME688 sensor and the AI tools used in the training of the AI models will be BME AI-Studio, Edge Impulse, and TensorFlow. With these AI tools three different AI models will be created and then compared to each other.

The main research question is threefold: Is it practical to use the ESP32 as a platform for the AI model or is it better to just use the ESP32 as a data collecting unit and send the data to another device to be processed? Or is a cloud computing solution a better choice? The use of ESP32 as the base for the AI model will put more limitations on the AI model itself than using the ESP32 only as a data collector and using a more powerful solution for running the AI model.

This thesis is structured as follows: the second chapter discusses the requirements of the thesis and reviews the hardware and software candidates for the completions of this task. The third chapter provides more details for each of the hardware and software candidates. In the fourth chapter the workflow created during this thesis is discussed. In the fifth chapter the results of the workflow will be presented and discussed. The sixth and final chapter is the conclusion and discussion of the future development of this workflow.

2 Requirements

The goal of this thesis work is to create a workflow that will use the collected data to teach an AI model to recognize certain gases or compounds produced by the brewing process. This workflow will use one of the methods listed as the possible candidates for completing this task. In the next chapter there is a more in-depth look into each of the possible options.

The current requirements for this workflow are as follows. The most important requirement is to be able to create a workflow that uses the data generated by the BME688 gas sensor to train an AI model to classify different gases and compounds. The workflow created should be implemented so that it can used for many different purposes and it should be able to classify as many different compounds as it is required by the task it is used for. Some less critical but still important requirements are that the workflow should not be too dependent on pre-existing products as this raises the risk of performance bottlenecks and unexpected breaking changes as you are not in control of these factors. Costs and unexpected charges are also a concern as products that provide free services are known to in time increase pricing, introduce usage caps of the free version or implement metered billing.

Considering the tools listed as candidates to be used to complete this workflow some of the options are easy to rule out. Based on the limitations imposed by some of the option considered in this thesis the BME AI-Studio and Edge Impulse can be ruled out immediately as both have limitation that are critical in the completions of the desired workflow.

The BME AI-Studio has severe limitation as it is only possible to train AI models to classify four different compounds at once. This is a very undesirable aspect of the AI model that this workflow is designed to train as the model will be needed at some point to classify more than four different classes. There is also the aspect of reliance on a pre-existing product, this is not in the interest of this workflow.

The Edge Impulse is also ruled out as it is also a pre-existing product, and its continuous use would require a paid plan to be used commercially and not just for research or education purposes.

3 Hardware and software

In this chapter the reasons and decisions of what hardware and software are to be used in this thesis is discussed. The most important part of this process is to have variety of tool to use in the creation of the AI models. As the selection of the sensor for this thesis its criteria are to be able to detect Volatile Organic Compounds in the air. These sensors include Photoionization Detectors (PIDs), Flame Ionization Detectors (FIDs) and Metal Oxide Semiconductor (MOS) Sensors [1]. In this thesis a Metal Oxide Semiconductor Sensor is used because the Bosch BME688 sensor has an official comprehensive BME AI-Studio software tool that can be used to make AI models with the data provided by the BME688 sensor [2].

3.1 Hardware

3.1.1 BME688-sensor

The Bosch BME688 is an environmental sensor that has artificial intelligence (AI) capabilities as well as high-linearity and high-accuracy gas, humidity, pressure, and temperature sensing integrated into a compact package. The sensor is housed in a sturdy 3.0 x 3.0 x 0.9 mm3 package and it is developed for mobile and connected applications where the sensors size and low power consumption are of high importance. The gas sensor is capable of detecting Volatile Organic Compounds (VOCs), volatile sulfuric compounds (VSCs) as well as other gases like carbon monoxide and hydrogen in part per billion (ppb) range. The BME688 sensor has a gas scanner function. In the sensors standard configuration, the presence of VSCs is detected as an indicator for example for bacterial growth. The gas scanner can be customized with different selectivity, sensitivity, data rate and power consumption. The BME AI-Studio tool enables individuals of companies to train the BME688 gas scanner for their specific applications that can include, for example home appliances, IoT products or Smart Home. [3]

3.1.2 ESP32

The ESP32 is a single chip with 2.4GHz Wi-Fi-and-Bluetooth combo that is designed with the TSMC low-power 40 nm technology. The chip is designed to achieve the best possible power and RF performance. It is robust, versatile and reliable in a wide range of applications and power scenarios. The ESP32 is a highly integrated, low-power system-on-chip (SoC) microcontroller and thus supports a wide range of Internet of Things (IoT), wearable, and embedded applications. It features a dual-core Xtensa® 32-bit LX6 microprocessor along with on-chip 448 BK ROM and 520 KB SRAM and QSPI supports external flash/SRAM chips. It also offers robust wireless connectivity via integrated 802.11 b/g/n 2.4GHz Wi-Fi up to 150Mbps and dual-mode Bluetooth with standard Bluetooth 4.2 and Bluetooth LE. The chip also has comprehensive peripheral interfaces such as ADCs, DACs, touch sensors, SPI, I²C, UART, motor PWM and more, that offers possibilities in hardware development. Its Ultra-Low-Power (ULP) coprocessor and multiple sleep modes make it well-suited for energy-efficient applications. [4]

3.2 Software

3.2.1 BME AI-Studio

BME AI-Studio is a software platform that brings together BME688 Development Kit with BSEC software. This software enables users to train and deploy machine learning models for gas detection using the BME688 environmental sensor. BME AI-Studio allows developers to use the BME688 Development Kit to record gas sensor data under different conditions and use that data to train an AI model to classify specific gas signatures, for example to classify coffee aroma or normal air. The platform supports data collection, preprocessing, model training, validation, and model deployment, making it a comprehensive solution for customized gas sensing applications. [2]

3.2.2 Edge Impulse

Edge Impulse is a pioneering development platform tailored for building and deploying machine learning (ML) models on embedded systems and edge devices. It provides a comprehensive toolchain that spans data acquisition, signal processing, model training, validation, and deployment, all optimized for low-power, resource-constrained hardware. [5] Edge Impulse has native Python integration with the Python SDK as well as extensive python and Node.js API bindings. It can also be optimized for on-device performance. Edge Impulse simplifies the deployment process by generating optimized C++ or EON (Edge Optimized Neural) models that can run on microcontrollers and embedded Linux devices without internet connectivity. [6]

3.2.3 TensorFlow

TensorFlow is a free and open-source software platform for building and deploying machine learning and deep learning models. It provides an ecosystem of tools and libraries that enable developers to design and deploy scalable machine learning applications. It can be used to develop across a variety of environments from mobile devices to large-scale distributed systems. TensorFlow supports flexible architecture with features such as Keras Functional API and Model Subclassing API. With robust support for model training, serving, and optimization, TensorFlow is a powerful tool for machine learning. [7]

3.2.4 PyTorch

PyTorch is an open-source deep learning framework designed to support both rapid research prototyping and scalable production deployment. It provides a flexible computational paradigm based on dynamic computation graphs,

enabling models to be constructed and modified at runtime in a manner consistent with idiomatic Python programming. The framework includes highly optimized tensor operations with support for hardware acceleration across GPUs and other performance-oriented backends. PyTorch is complemented by a broad ecosystem of domain specific libraries and tools that facilitate workflows in computer vision, natural language processing, reinforcement learning, and generative modelling. Together, these components provide a unified platform that supports the complete machine learning lifecycle, from model development and experimentation to large-scale training and deployment. [8]

4 Training and evaluating the AI models

The aim of the thesis work was to create a workflow that uses the data collected with Bosch BME688 sensor to train an AI model. This raw data from the sensor must be put through the BME AI-Studio to create the .bmespecimen files used in the workflow. The workflow then formats the data so that it can be used to train an AI model and train the model with the data. The trained AI model is then saved to be used with in classification of gasses. This workflow (Figure 1) was created with Python using the PyTorch framework. The workflow consists of four python script that work in in sequence to achieve the desired result.

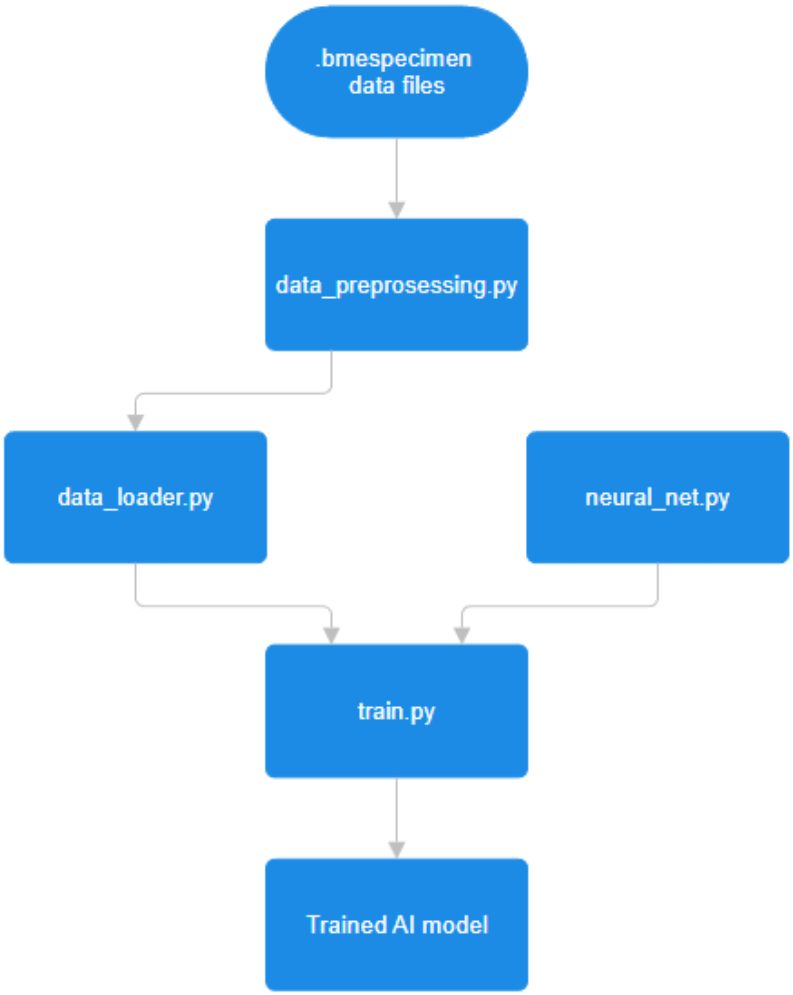

Figure 1. Flowchart of the workflow.

The first part of the workflow is the parsing of the .bmespecimen files. These data files are created with the Bosch AI-Studio and they are used in this workflow for their ease of use, the use of the raw data produced by the sensor was considered , but was eventually discarded as the raw data format is much harder to work with than the .bmespecimen files created from the raw data using the AI-Studio. To get the data in a format that is used to train the AI model the first script of the workflow, the data_preprosessing.py is used.

The data_preprosessing.py script reads the .bmespecimen files in a designated directory and parses the data into a more suitable format to be used in the training of the AI model. The script parses through the .bmespecimen data file and collects all the relevant data.

```python
class DataPreprocessor:
    def read_bmespecimen_file(self, file_path: str) -> Tuple[np.array, int]:

        try:
            root = data.get("data", {})

            # --- specimenData ---
            specimen_data = root.get("specimenData", {})
            specimen_info = {
                "id": specimen_data.get("id"),
                "label": specimen_data.get("label")
            }

            # --- heaterProfiles ---
            heater_profiles = []
            for hp in root.get("heaterProfiles", []):
                steps = hp.get("steps", [])
                temperatures = [s.get("temperature") for s in steps if "temperature" in s]
                heater_profiles.append({
                    "id": hp.get("id"),
                    "uid": hp.get("uid"),
                    "temperatures": temperatures
                })

            # --- sensorConfigs ---
            sensor_configs = []
            for sc in root.get("sensorConfigs", []):
                sensor_configs.append({
                    "id": sc.get("id"),
                    "sensorId": sc.get("sensorId"),
                    "heaterProfileId": sc.get("heaterProfileId")
                })

            # --- cycles ---
            cycles = []
            for cy in root.get("cycles", []):
                cycles.append({
                    "id": cy.get("id"),
                    "sensorId": cy.get("sensorId")
                })

            # --- dataColumns ---
            data_columns = []
            for dc in root.get("dataColumns", []):
                data_columns.append({
                    "name": dc.get("name"),
                    "key": dc.get("key")
                })
```

Figure 2. A part of code from the data_preprosessing.py showing some of the parsing process.

The structure of the .bmespecimen file is quite convoluted and requires quite a lot of linking of specific parts of structure and creating look up tables for easier access, so the parsing process can be completed without too much back and forth.

```
# --- Link specimenDataPoints to cycles by cycle_id ---
if cycles and mapped_data_points:
    # Create a lookup dictionary for fast access
    cycle_lookup = {c["id"]: c["sensorId"] for c in cycles if "id" in c and "sensorId" in c}

    for dp in mapped_data_points:
        cycle_id = dp.get("cycle_id")  # match key
        if cycle_id in cycle_lookup:
            dp["sensor_id"] = cycle_lookup[cycle_id]

# --- Link specimenDataPoints to sensor_configs by sensor_id ---
if sensor_configs and mapped_data_points:
    # Create a lookup dictionary for fast access
    sensor_configs_lookup = {c["sensorId"]: c["heaterProfileId"] for c in sensor_configs if "sensorId" in c and "heaterProfileId" in c}

    for dp in mapped_data_points:
        sensor_id = dp.get("sensor_id")  # match key
        if sensor_id in sensor_configs_lookup:
            dp["heater_profile_id"] = sensor_configs_lookup[sensor_id]

# --- Link specimenDataPoints to sensor_configs by cycle_step_index ---
if heater_profiles and mapped_data_points:
    # Create a lookup dictionary for fast access
    heater_profiles_lookup = {c["id"]: c["temperatures"] for c in heater_profiles if "id" in c and "temperatures" in c}

    for dp in mapped_data_points:
        index = dp.get("cycle_step_index")
        heater_profile_id = dp.get("heater_profile_id")  # match key
        if heater_profile_id in heater_profiles_lookup:
            dp["heater_temperature"] = heater_profiles_lookup[heater_profile_id][index]

# --- Link specimenDataPoints to specimen_info by sensor_id ---
for i in mapped_data_points:
        dp["specimen_id"] = specimen_info['id']
```

Figure 3. Code showing the linking and the creation of look up tables.

During this parsing process the relevant data points are collected and grouped in preparation for the training of the AI model. The BME688 sensor has a heating cycle that has ten steps and the parsing process groups these ten data points from the heating cycle into a thirteen data point long array along with temperature, humidity and pressure.

```
# --- build final structured output ---
keys_to_keep = ["resistance_gassensor","temperature","pressure","relative_humidity"]
extracted_data = [
    #"specimenData": specimen_info,
    #"heaterProfiles": heater_profiles,
    #"sensorConfigs": sensor_configs,
    #"cycles": cycles,
    #"dataColumns": data_columns,
    #"specimenDataPoints": mapped_data_points
    [d[k] for k in keys_to_keep if k in d] for d in mapped_data_points
]
```

Figure 4. Code of the final structure of the output.

The parsed data is used to create the training data and the validation data arrays. These two data arrays will be used to create the training and validation datasets. Finally, the script saves the training data and the validation data into a .h5 file format which is Hierarchical Data Format used to store large amounts of data for quick retrieval and analysis.

```
def prepare_training_data(self,
                          features: np.ndarray,
                          targets: np.ndarray,
                          train_split: float = 0.8,
                          random_seed: int = 42) -> Dict[str, np.ndarray]:
    # Set random seed
    np.random.seed(random_seed)

    # Split data
    indices = np.random.permutation(len(features))
    split_idx = int(len(features) * train_split)

    train_indices = indices[:split_idx]
    val_indices = indices[split_idx:]

    # --- FIX: Scale features ONLY using training set statistics ---

    # 1. Fit scaler on TRAINING features
    train_features_scaled = self.scaler.fit_transform(features[train_indices])

    # 2. Transform VALIDATION features using the *fitted* scaler
    val_features_scaled = self.scaler.transform(features[val_indices])

    # Create train and validation sets
    train_data = {
        'features': train_features_scaled, # Use already scaled training features
        'targets': targets[train_indices]
    }

    val_data = {
        'features': val_features_scaled, # Use already scaled validation features
        'targets': targets[val_indices]
    }

    return train_data, val_data
```

Figure 5. Code that creates training and validation data.

The second part of the workflow consists of the data_loader.py and the neural_net.py scripts. The data_loader.py handles the creation of the training and the validation datasets. These datasets are arrays created from the training data and validation data provided by the data_preprocessing.py. The data_loader.py script will also create the data loaders. The script defines a

custom PyTorch dataset that wraps the feature and target arrays and makes them compatible with the PyTorch framework.

```python
class BMESpecimenDataset(Dataset):
    def __init__(self, features: np.ndarray, targets: np.ndarray, transform=None):
        """
        Initialize the dataset.

        Args:
            features (np.ndarray): Feature array
            targets (np.ndarray): Target array
            transform: Optional transform to be applied to the data
        """
        self.features = torch.FloatTensor(features)

        # Ensure targets are 2D
        if len(targets.shape) == 1:
            targets = targets.reshape(-1, 1)
        self.targets = torch.LongTensor(targets)

        self.transform = transform

    def __len__(self) -> int:
        return len(self.features)

    def __getitem__(self, idx: int) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Get a data sample.

        Args:
            idx (int): Index of the sample to get

        Returns:
            tuple: (features, target)
        """
        features = self.features[idx]
        target = self.targets[idx]

        if self.transform:
            features = self.transform(features)

        return features, target
```

Figure 6. Code of the custom BMESpecimenDataset class.

The data_loader.py uses the imported methods from the data_preprosessing.py to load the data saved into the .h5 file and preparing it by splitting it to training and validation data according to the definable training split value. In this process a random seed is also given to the data, this ensures that any randomness in the data pipeline happens the same way every time the same data is used. By ensuring the randomness is the same every time helps with reproducibility. This means it is easier to compare results while using different models because the results are not affected by the differences in data order. Reproducibility also ensures that others running the code with the same data will get the same outputs using the same predetermined random seed.

The second script of this part of the workflow is the neural_net.py. This script defines the neural network either with the default values or with the values specified as options for the train.py script when the whole workflow is executed. The neural_net.py script defines a feedforward neural network that has a customizable number of hidden layers, uses ReLu activations, BatchNorm and Dropout. The class NeuralNetwork as seen in the figure below defines a standard multilayer perceptron (MPL). The number of hidden layers can be configured by giving an option to the workflow, this option defines the number of hidden layers as well as the individual number of neurons in each layer. Each hidden layer contains a linear layer that learns the weighted combination of inputs. ReLU activation is used to introduce non-linearity, and the Dropout is used to prevent overfitting by dropping a predefined number of connections between the neurons of each hidden layer.

```python
# ---------------------- NEURAL NETWORK DEFINITION ----------------------
class NeuralNetwork(nn.Module):
    def __init__(self, input_size: int, hidden_sizes: List[int], output_size: int):
        """
        Feedforward neural network for classification.
        """
        super(NeuralNetwork, self).__init__()

        layers = []
        prev_size = input_size

        # Hidden layers
        for hidden_size in hidden_sizes:
            layers.extend([
                nn.Linear(prev_size, hidden_size),
                nn.ReLU(),
                nn.BatchNorm1d(hidden_size),
                nn.Dropout(0.2)
            ])
            prev_size = hidden_size

        # Output layer (raw logits — CrossEntropyLoss applies softmax internally)
        layers.append(nn.Linear(prev_size, output_size))

        self.model = nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.model(x)
```

Figure 7. Part of the code showing the NeuralNetwork class.

The script also defines the ModelTrainer class which is used to train the created neural network. The ModelTrainer uses CrossEntropyLoss, Adam optimizer and

OneCycleLR learning rate scheduler. The CrossEntropyLoss is used for classification, Adam optimizer helps with faster convergence and OneCycleRL is a scheduler for the learning rate which starts with low learning rate then increases to the predetermined maximum learning rate and the decreases the learning rate again. This often produces faster and more accurate training.

```python
# ---------------------- MODEL TRAINER ----------------------
class ModelTrainer:
    def __init__(self, model: nn.Module, learning_rate: float = 0.001,
                 max_lr: float = 0.01, epochs: int = 20, steps_per_epoch: int = None):
        """
        Trainer class for classification using CrossEntropyLoss and OneCycleLR.
        """
        self.model = model
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.Adam(model.parameters(), lr=learning_rate)

        # Scheduler will be initialized after we know steps_per_epoch
        self.scheduler = None
        self.max_lr = max_lr
        self.epochs = epochs
        self.steps_per_epoch = steps_per_epoch

        self.train_losses, self.val_losses = [], []
        self.train_accuracies, self.val_accuracies = [], []
```

Figure 8. Code defining the ModelTrainer class.

The script plots training and validation loss and accuracy to visualize the model's convergence. Finally, the script saves the newly trained model to be used in classification of the data it was trained to classify.

The last part of the workflow is the train.py script. This is the top-level training script for the neural network. It uses the imported methods of the previous scripts executes them in order. First it processes the raw data provided or loads pre-processed data. Then it creates the PyTorch dataloaders, builds the neural network, and begins the training loop for the model. As the training loop progresses it plot the training and validation loss and accuracy. When the training is completed, the script generates and displays loss and accuracy curves as well as a confusion matrix to visualize the training and the results. Finally the trained model is saved.

5 Results

The code of the workflow created during this thesis can be found in GitHub: https://github.com/AtteMK/Thesis-work.

In this chapter the results are shown and discussed. The final product of this thesis is a Python based script collection that creates a workflow. This workflow reads the data file collected with Bosch BME688 gas sensor and uses it to train an AI model for classification. This workflow fulfils most of the requirements that were set at the beginning of this thesis work. The most important requirement was met as this is a Python based workflow created with PyTorch framework and it can use the data collected with BME688 sensor to train an AI model to classify different gases and compounds. The workflow can classify an unlimited number of different classes as there is no hard limit on the number of classes the AI model can be trained with. As this workflow is created purely with Python using the PyTorch framework, there is no dependencies to any pre-existing products in the functionality of the workflow. However, the data collected with the BME688 sensor must be processed using the BME AI-Studio to get the .bmespecimen files used in this workflow.

6 Conclusion and future work

The goal of the thesis was to create a flexible workflow that uses data collected with the BME688 gas sensor and to use that data to train an AI model for identifying gases and compounds produced during the brewing process. The objectives of the thesis included creating a scalable and adaptable system capable of handling a wide range of classification tasks while remaining usable for different applications. Another key objective was to have as little reliance on external tools to avoid performance limitations, unexpected changes, and potential cost increases in the future.

The workflow works and produces a trained AI model that can be deployed to be used for classification. Figure 9 illustrates the training process where the training loop prints training and validation loss and accuracy after every epoch.

```
Epoch 1/50: Train Loss=0.6850, Val Loss=0.1747, Train Acc=0.801, Val Acc=0.958, LR=0.000505
Epoch 2/50: Train Loss=0.1969, Val Loss=0.0768, Train Acc=0.941, Val Acc=0.975, LR=0.000815
Epoch 3/50: Train Loss=0.1257, Val Loss=0.0605, Train Acc=0.957, Val Acc=0.977, LR=0.001317
Epoch 4/50: Train Loss=0.1032, Val Loss=0.0571, Train Acc=0.963, Val Acc=0.976, LR=0.001988
Epoch 5/50: Train Loss=0.0961, Val Loss=0.0684, Train Acc=0.965, Val Acc=0.974, LR=0.002800
Epoch 6/50: Train Loss=0.1005, Val Loss=0.0704, Train Acc=0.963, Val Acc=0.971, LR=0.003717
Epoch 7/50: Train Loss=0.0984, Val Loss=0.0466, Train Acc=0.963, Val Acc=0.985, LR=0.004699
Epoch 8/50: Train Loss=0.0966, Val Loss=0.0601, Train Acc=0.965, Val Acc=0.978, LR=0.005702
Epoch 9/50: Train Loss=0.0964, Val Loss=0.0532, Train Acc=0.964, Val Acc=0.978, LR=0.006684
Epoch 10/50: Train Loss=0.0922, Val Loss=0.0560, Train Acc=0.966, Val Acc=0.982, LR=0.007601
Epoch 11/50: Train Loss=0.0895, Val Loss=0.0477, Train Acc=0.967, Val Acc=0.983, LR=0.008412
Epoch 12/50: Train Loss=0.0907, Val Loss=0.0630, Train Acc=0.966, Val Acc=0.974, LR=0.009084
Epoch 13/50: Train Loss=0.0884, Val Loss=0.0468, Train Acc=0.967, Val Acc=0.984, LR=0.009585
Epoch 14/50: Train Loss=0.0860, Val Loss=0.0544, Train Acc=0.968, Val Acc=0.978, LR=0.009895
Epoch 15/50: Train Loss=0.0812, Val Loss=0.0551, Train Acc=0.969, Val Acc=0.979, LR=0.010000
```

Figure 9. Prints in the console while the training is running.

After the training process is completed, the workflow will generate a chart that plots the training and validation loss and accuracy to visualize the whole training process (Figure 10).

Figure 10. The training and validation loss and accuracy chart generated by the workflow.

The workflow will also evaluate the model with confusion matrix. This confusion matrix is printed in the terminal (Figure 11), and the workflow also generate a better visualization for the confusion matrix (Figure 12).

```
INFO:__main__:Evaluating model with confusion matrix...
INFO:__main__:Confusion matrix:
[[44326     0     0     0    23     0]
 [    0 44575   853     1     0     0]
 [    5   921 44328     1     0     0]
 [    0    58     7 43565     0     0]
 [  160     0     2     0 44210     0]
 [    0     0     0     0     0 58841]]
INFO:__main__:Training completed. Model saved as 'trained_model.pth'
```

Figure 11. The confusion matrix printed in the terminal.

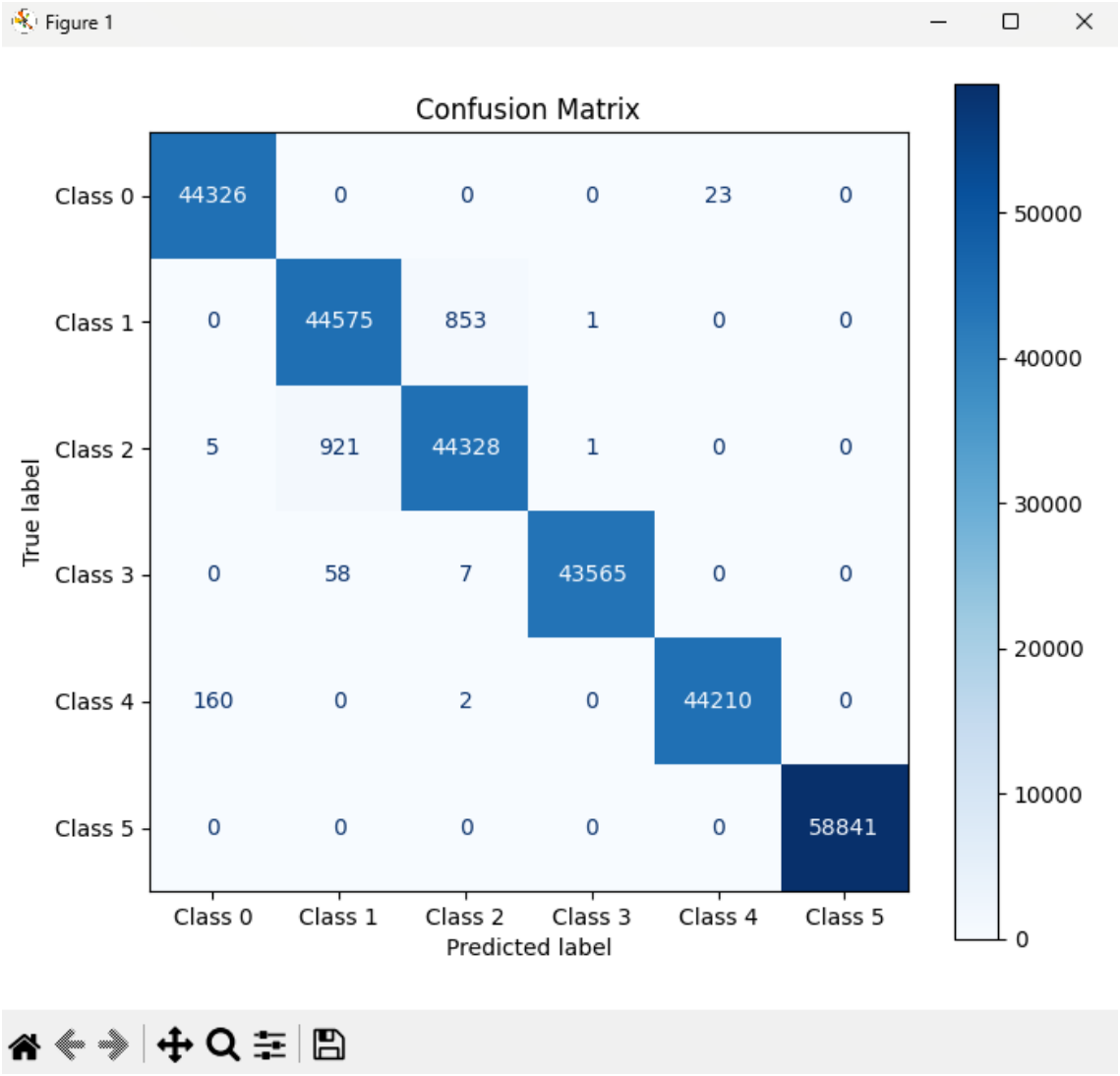

Figure 12. The better visualization of the confusion matrix generated by the workflow.

There are still some functionalities and other development idea for the workflow that were not implemented in this thesis. One such functionality would be the complete separation from the BME AI-Studio. This would be possible with a script that parses through the raw data files produced by the BME688 sensor. The parsing could generate a file that is in such a format that it can be used to generate the training and validation data for the AI model using parts of the workflow. The parsing could also generate a file that is structurally similar to the .bmespecimen file so that the workflow could use this new file generated from the raw data instead of .bmespecimen files. This would completely remove the

need to use the BME AI-Studio. Another future development idea would be the use of GPU acceleration in the training process. This functionality was not implemented as the datasets that were used in this thesis were not large enough to need to be processed with GPU acceleration. The GPU acceleration would still be a great addition to the workflow if the need to use much larger datasets rises in the future.

References

[1] The CMM Group, “Different Types of VOC Detectors Available,” The CMM Group, 2025. [Online]. Available: https://www.thecmmgroup.com/different-types-voc-detectors-available/. [Accessed 3 June 2025].

[2] Bosch Sensortec, “BME AI-Studio Documentation,” Bosch Sensortec, 2025. [Online]. Available: https://www.bosch-sensortec.com/software/bme/docs/. [Accessed 24 May 2025].

[3] Bosch Sensotec, “https://www.bosch-sensortec.com/,” February 2024. [Online]. Available: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme688-ds000.pdf. [Accessed 24 May 2025].

[4] Espressif Systems, “https://www.espressif.com/,” April 2025. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. [Accessed 24 May 2025].

[5] Edge Impulse, Inc., “Edge Impulse,” Edge Impulse, Inc., 2025. [Online]. Available: https://edgeimpulse.com/. [Accessed 24 May 2025].

[6] Edge Impulse, Inc., “Product,” Edge Impulse, Inc., 2025. [Online]. Available: https://edgeimpulse.com/product. [Accessed 24 May 2025].

[7] TensorFlow, “https://www.tensorflow.org/,” TensorFlow, 2025. [Online]. Available: https://www.tensorflow.org/about. [Accessed 24 May 2025].

[8] PyTorch, “PyTorch,” PyTorch, 2025. [Online]. Available: https://pytorch.org/projects/pytorch/. [Accessed 2 December 2025].