AdaptAE: Adaptive Autoencoder Training Framework for Edge Devices with Variable Memory Constraints

Nikhil Sengupta

University of
St Andrews

Supervised by
Dr. Blesson Varghese

January 5, 2024

Abstract

Autoencoders are an artificial neural network designed to learn a compressed representation of data to reconstruct the original input data as closely as possible. They have gained widespread adoption in edge computing due to their flexibility and wide variety of applications, including anomaly detection, feature learning and image classification. However, the autoencoder training process is computationally expensive and time-consuming due to its intensive hyperparameter tuning, iterative learning process and complex architecture. Present state-of-the-art techniques, namely Extreme Learning Machine Autoencoder (ELM-AE), have significantly reduced training time for autoencoders but still need to be thoroughly evaluated regarding reconstruction loss and memory consumption, a crucial factor for deployment on resource-constrained edge devices. This gap in the literature led to the development of AdaptAE, an ELM-based framework that allows devices with limited computing power to train autoencoders efficiently. The proposed solution, validated over five datasets (MNIST, Fashion-MNIST, CIFAR-10, CIFAR-100, and Tiny ImageNet), outperforms the traditional autoencoder by reducing the memory consumption by up to 66.3% the training time by up to 62.9% and outperforms ELM-AE by reducing the memory consumption by 89.6%. Additionally, AdaptAE maintains a comparable reconstruction loss and anomaly detection accuracies to both traditional autoencoders and ELM-AE on a sized-down Tiny ImageNet dataset, achieving a 0.0103 MSE reconstruction loss and 96.01% accuracy on anomaly detection, compared to the traditional autoencoder's 0.0113 MSE loss and 100% accuracy and ELM-AE's 0.0221 MSE Loss.

Keywords: Autoencoders, Edge Computing, Extreme Learning Machine, On-Device Training, Training Optimization

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 14,155 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Signed: Nikhil Sengupta
Date: January 5, 2024

Acknowledgements

I want to thank Dr Blesson Varghese for his guidance throughout this project. His willingness to provide weekly feedback and support, along with his expertise in edge computing and machine learning, was invaluable in shaping the direction and success of this research. I want to extend my thanks to Dr Varghese's doctoral student, Dhananjay Saikumar, for his practical knowledge of advanced machine-learning techniques. Finally, I would like to thank my family - my mother, father and brother - and my friends Eva Gunn and Aryan Singh for their unwavering encouragement and support, providing insightful feedback and suggestions that helped refine my work and maintain my focus for the duration of the project.

Contents

1 Introduction

Machine learning, particularly in the context of neural network optimization, has been the subject of extensive research from the early days of simple perceptrons [1] to the current era of deep learning [2] [3]. There is a constant pursuit of optimizing a model to process large amounts of data efficiently and extract meaningful insights from it. With the rise of edge computing, a paradigm that leverages devices with limited compute power near the network's edge to process data close to its source, [4] this pursuit now takes on the unique challenge of optimizing a neural network model to run on these devices. One such neural network model that has gained popularity in recent years due to its flexibility and wide range of applications is the autoencoder.

Autoencoders are a self-supervised neural network that aims to reconstruct its input data by learning a compressed data representation [5]. The vanilla autoencoder consists of an encoder and decoder, serving as the basis for all other autoencoder architectures. The encoder learns a compressed representation of the input data, known as the latent space, and the decoder attempts to reconstruct the original input data from the latent space. By learning to reduce the data to lower-dimensional space and attempting to reconstruct it back to its original form, autoencoders can provide valuable insights by capturing the underlying patterns and features in the data [6].

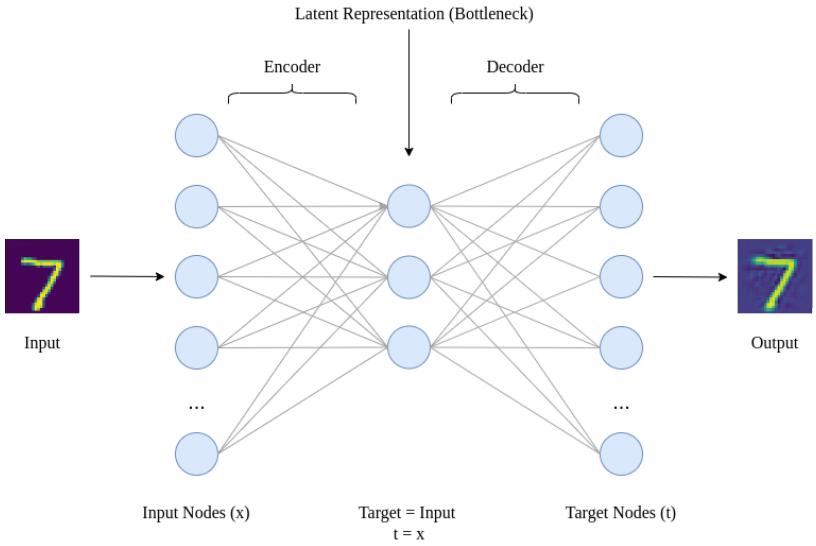

Figure 1: A vanilla autoencoder architecture with a single hidden layer

An autoencoder's ability to learn standard data patterns makes them particularly useful for tasks such as image denoising, anomaly detection, and classification [6]. In image denoising, autoencoders can be trained to remove noise from an image. This has beneficial medical applications, where image denoising is an essential pre-processing step in medical image analysis to enhance the quality and clarity of MRI and CT scans [7]. They have also proven more robust to perturbations and adversarial input data than other popular neural networks, such as Convolutional Neural Networks [8]. They are also prevalent for anomaly detection tasks, such as network flow intrusion detection [9] and credit card fraud detection [10] due to their proficiency in learning standard data patterns

and identifying deviations from them.

Edge devices, such as drones and wearables, can effectively leverage the inherent strengths of autoencoder for tasks like real-time image classification, identifying anomalies in sensor data streams and enhancing image quality in surveillance. Using an autoencoder, edge devices can pre-process the data and reduce its dimensionality on-site, enabling more efficient transmissions of processed data to the cloud and faster response actions for end devices. In addition, having an autoencoder run on edge devices can also prove to be a more cost-effective and efficient alternative to hosting machine learning models on the cloud. High latencies, which are especially prevalent in cloud computing that edge computing mitigates, can negatively affect the performance of the autoencoder, especially for real-time applications [11].

1.1 Problem Statement

Despite the suitability of autoencoders for edge computing, the training of autoencoders is computationally expensive, a significant barrier to their adoption in edge devices with limited compute power [12]. This is due to a variety of factors, including tuning the hyperparameters of an autoencoder (such as learning rate, batch size, and neurons per layer), the architecture of the autoencoder, and adjusting the autoencoder's weights with backpropagation, a popular machine-learning technique aims to improve the model iteratively. These factors can substantially impact both the time taken and the memory needed to train an autoencoder.

This project aims to create an algorithm that significantly reduces the training time required for an autoencoder while retaining a low reconstruction error and peak memory usage. The ultimate goal is to allow autoencoders to be trained on devices with different memory budgets.

1.2 Research Objectives

The main aim of this project is to develop an optimization technique which can **reduce** the training time and peak memory usage of a regular autoencoder whilst maintaining a reasonable level of reconstruction loss. This technique can be used to train on edge devices with a low memory budget to perform tasks such as anomaly detection in real-time sensor data, image and video processing for surveillance systems and decision-making on edge devices.

The following objectives have been identified to achieve this aim:

1. **Review state-of-the-art optimization techniques used to train autoencoders to understand their limitations.** Initiate a thorough literature review of traditional methods of training autoencoders, their applications for edge computing and the benefits and drawbacks of training autoencoders on edge devices. This review will also cover some of the latest methods designed to overcome the challenges of optimizing autoencoder training and analyze their potential limitations.

2. **Design and implement a novel autoencoder training optimization technique.** Based on the findings of limitations of both traditional and state-of-the-art methods of training autoencoders, develop a framework that aims to overcome the challenges found in these methods. The framework should be suitable to run on edge devices with limited computational resources. It should also be scalable and flexible, allowing it to be run on various edge devices with different computational constraints.

3. **Evaluate the performance of the proposed technique.** Following the implementation of the framework, evaluate its performance based on standard metrics used to evaluate the performance of autoencoder in an edge computing context, such as reconstruction loss, peak memory footprint, and training time. These metrics should be evaluated on common autoencoder tasks, such as feature learning through reconstruction and anomaly detection.

1.3 Accomplishments

The following accomplishments have been achieved over the course of this project:

1. **Identifying Limitations:** Over 40 academic papers and sources were reviewed on edge computing, autoencoders, the benefits and challenges of training autoencoders for edge devices and state-of-the-art methods that aim to address these challenges. The challenges of traditional autoencoder training methods included a significantly high training time due to both hyperparameter tuning and backpropagation, along with a high computational load that arises during hyperparameter tuning. State-of-the-art methods identified (ELM-AE and OS-ELM) mitigated the high training times of autoencoders but showed a lack of coverage on computational load. This review set the basis for the project and provided a comprehensive understanding of the current landscape in this field.

2. **Design and Implementation:** The design and implementation of AdaptAE, a modification of the OS-ELM (Online Sequential Extreme Learning Machine) algorithm. This flexible framework allows users to analytically place importance on either low memory (for edge devices with low compute power) or high performance whilst guaranteeing a significantly lower training time than a traditional autoencoder with the same architecture and maintaining a reasonably low level of reconstruction loss.

3. **Evaluation:** The performance of AdaptAE was evaluated against baseline models, including a vanilla autoencoder and ELM-AE for both reconstruction tasks for feature learning and anomaly detection tasks. Extensive experimentation and analysis were conducted by varying the hyperparameters of each model to observe their effect on reconstruction loss, peak memory usage and training time. This experimentation resulted in a comprehensive understanding of each hyperparameter to minimize memory consumption or maximize performance with precision based on the user's goals.

1.4 Organization of the Report

The remainder of this report is organized in the following manner:

Chapter 2: Background & Literature Review Explores existing literature and background on the fundamentals of training neural networks, autoencoders and their application in edge computing, followed by challenges that arise when training autoencoders and ending with current state-of-the-art methods that aim to mitigate these challenges.

Chapter 3: Requirement Specification & Ethics Outlines the project's requirements, including the primary and secondary objectives and the ethical considerations for the project, including the secondary datasets utilized.

Chapter 4: AdaptAE: Proposed Solution Introduces the proposed solution, AdaptAE, including the motivation behind the solution, followed by an overview of the method and ending with an explanation of the model

Chapter 5: Evaluation Evaluates the performance of the proposed solution against that of a regular autoencoder, including the experimental setup, the results and the analysis of the results.

Chapter 6: Conclusion and Future Work Concludes the report with a summary of the project, the contributions made and the future work that can be done to improve the proposed solution.

2 Background & Literature Review

This section delves into the background and a detailed literature review essential for understanding how autoencoders can be optimized to be trained on resource-constrained devices. The background will cover the fundamentals of edge computing, edge intelligence, autoencoder architecture and training techniques. Current challenges of training autoencoders for the edge will also be discussed, along with optimization techniques to mitigate these challenges. Finally, this section will summarize some key limitations identified from the literature review of traditional autoencoders and the state-of-the-art techniques to optimize them for edge environments.

2.1 Edge Computing

Edge computing can first be traced back to 1999 when Akamai launched its Content Delivery Network (CDN), where nodes that were geographically closer to the end user were introduced and were responsible for the delivery of cached content such as images and videos [13]. Since then, there have been multiple definitions for edge computing [14]. However, a common thread amongst these definitions is the focus on processing data sources closer to data sources to bring faster insights, processing larger pieces of data at faster speeds and better bandwidth availability [15] [16]. The nodes in the original CDN were known as "caching servers", but since its conception, the concept of an edge device has extended beyond just servers to include other types of devices, including routers, firewalls, robots and IoT devices [17].

In the edge computing paradigm, there are three main components [18]. They are as follows:

1. **End device (front-end)** End devices collect data from the environment, providing users with the best responsiveness and interactivity. They are also responsible for receiving edge device data and performing the corresponding tasks [14]. Examples of these kinds of devices include sensors, cameras, and actuators.

2. **Edge device (near-end)** Edge devices are responsible for processing the data collected by the edge devices. They sit between the cloud and the end devices but are closer to the end devices to leverage low latency. Edge devices typically have more computing power than end devices but are still significantly less powerful than cloud servers. These devices include network gateways, routers, computing units and drones.

3. **Core cloud (far-end)** Cloud servers are the most powerful of the three types of devices, with the highest compute power, storage capacity and memory. They are responsible for storing and processing the data collected from edge devices. Given their powerful capabilities, they are typically ideal for handling large-scale and complex processing tasks that go beyond what edge devices can handle.

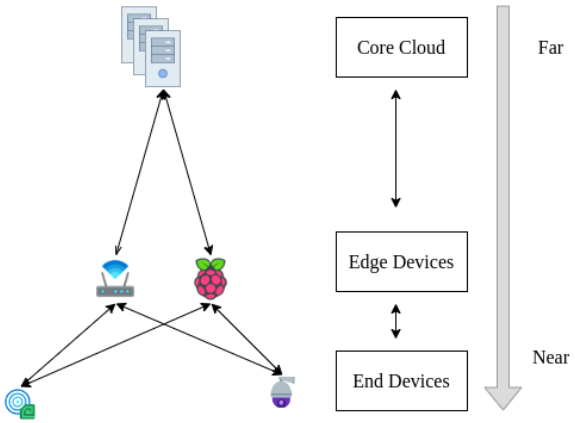

Figure 2: Edge computing paradigm

Cloud servers have been the dominant paradigm for data processing for the past decade. They provide a few significant benefits, including offloading low-level administration to cloud providers (such as system infrastructure and backup policy), reducing upfront costs of electricity, hardware and software, and more effective resource utilization [19]. However, despite these advantages, edge computing aims to resolve a few limitations of relying solely on cloud servers for data processing. They are as follows:

1. **Lower Latency** Edge computing reduces data processing latency as edge devices are closer to the end devices than cloud servers. Therefore, it is much more equipped to process large amounts of data quickly and efficiently compared to that of the cloud [20]. This is especially important for applications that require real-time processing, such as autonomous vehicles and robotics.

2. **Lower Bandwidth Usage** When a significant portion of data is congregated and set to the cloud for processing, it can cause the network bandwidth to be congested. This can lead to a sharp increase in latency and a waste of network resources [20]. However, edge computing allows edge devices to process the data locally, reducing the amount of data that needs to be sent to the cloud. Additionally, edge devices utilize the underlying hardware of the device itself rather than the cloud, which means that they use all the available resources without needing extra ones [21]. This is especially important for applications that require high bandwidth, such as image and video streaming.

3. **Robust Privacy** Edge computing can prove to be a more robust solution for privacy when compared to pure cloud computing. In the cloud computing paradigm, a few issues relating to privacy and security have been acknowledged, including areas on confidentiality, data security, phishing, and multi-tenancy [22]. One primary reason for this is that cloud computing transfers all the data out of the user's hands, where individuals have to rely on the security measures of their cloud provider. While edge computing does not aim to resolve these issues fully, it significantly mitigates them as data can now be processed at the nearest edge without sending sensitive data to remote servers.

From these advantages, edge computing harnesses the power of end devices, edge devices and the cloud to create a more scalable, efficient and private ecosystem for handling large and complex datasets.

2.2 Machine Learning on the Edge

Building upon the previously outlined benefits of edge computing over traditional cloud computing, edge computing lays the foundation for integrating machine learning (ML) directly into edge environments. With the rise of Mobile Edge Computing (MEC), a paradigm which allows mobile and cloud computing services within proximity of the user [23], the concept of Mobile Edge Learning (MEL) was defined, combining MEC with AI such that the learning model, parameters and data are distributed across multiple edge servers, and an AI model is trained from the distributed data [24]. Since MEL was introduced, the concept of combining edge computing with machine learning has been explored extensively, from assisting healthcare professionals in identifying new strains of diseases [25], fertilizing plants and driving automobiles [23]. AI can be especially beneficial for edge devices as it empowers them to adapt to new situations and learn to execute identical tasks [26]. With this newfound research emerges the concept of Edge AI.

Edge AI combines edge computing and AI, where computations are performed closer to the data source than at a centralized cloud centre [27]. This concept can be partitioned into two categories: 1) *AI for edge*, where AI can be used to endow edge devices with more intelligence by providing a better solution to constrained optimization problems, and 2) *AI on edge*, where AI models be trained on and run on resource-constrained edge devices [28]. The main goal of AI on edge applications is to allow complex AI models to conserve energy, reduce latency and improve privacy when running on edge devices. By performing AI reasoning tasks and training directly on edge, data can efficiently be processed in real-time rather than uploading massive quantities of data to the cloud for analysis [14].

2.3 Autoencoder

Autoencoders are an artificial neural network designed to learn a compressed representation of data to reconstruct the original input data as closely as possible. Its architecture comprises 1) an encoder, 2) a latent feature representation, and 3) a decoder. The encoder is responsible for encoding the input data to learn a compressed and meaningful representation of the data, referred to as the *latent feature representation*. The latent representation preserves the most important characteristics of the data. For instance, the latent features of hand-drawn digits could represent the number of lines required to write each number and how each line connects [5]. Finally, the decoder aims to reconstruct the original input from the latent representation whilst minimizing the reconstruction loss. This process can be described formally as

$$\arg\min_{f,g} E[\Delta(x, f(g(x))] \tag{1}$$

where the encoder is a function $g : \mathbb{R}^n \to \mathbb{R}^q$, the decoder is a function $f : \mathbb{R}^q \to \mathbb{R}^n$, E is the expectation over the distribution of x and Δ is the reconstruction loss function, measuring the distance between the output of the decoder and the input of the encoder [6]. To prevent an autoencoder from learning to reconstruct the output perfectly (thus learning the identify function), a *bottleneck* layer that contains fewer neurons than both the encoder and decoder is added to force the autoencoder to learn a compact and informative representation.

2.4 Autoencoders in Edge Computing

Autoencoders have recently gained traction, particularly in edge devices, due to their exceptional performance in anomaly detection, image classification and denoising tasks. Their effective performance can be attributed to a few main characteristics. Firstly, their ability to efficiently reduce the dimensionality of the input data is beneficial for anomaly detection. Once a latent representation of the input data is obtained, the reconstruction loss between new data and the data the autoencoder is trained can determine which samples are standard and which are anomalous [29]. They have also been shown to have a low computational cost compared to other dimensionality reduction methods such as kernel PCA [29]. These qualities make autoencoders an attractive option for resource-constrained devices like edge devices.

In the realm of edge computing, anomaly detection is increasingly prevalent in various scenarios, including detecting anomalies in video surveillance cameras, wearable health devices and autonomous vehicles [30] [31] [32]. Autoencoders have been shown to be particularly effective in anomaly detection tasks as they can learn a compressed representation of the input data and reconstruct the input data with minimal reconstruction loss. Another prevalent task for edge devices with poor-quality sensors, such as handheld ultrasound devices [33] and smartphones [34], is denoising noisy input data. A type of autoencoder known as a denoising autoencoder has proven to be particularly useful for denoising applications as they have shown better performance than traditional linear sparse coding algorithms in denoising images corrupted by additive white Gaussian noise [35].

2.5 Evaluating Autoencoder Performance in Edge Computing

Understanding the effectiveness of an autoencoder for edge devices involves looking at a few key factors, given that edge devices typically have limited power and computing ability. Here are four important metrics to consider:

Training Time Training time is the time it takes to train the autoencoder. It is used to evaluate how efficient and practical the model is for deployment on edge devices in a real-world edge computing scenario, where rapid deployment is crucial. Additionally, edge devices are often limited in energy resources, with devices that may be battery-powered or have other energy constraints. Because training time is shown to have a non-linear relationship with energy consumption [36], it is crucial to minimize the training time to maximize the model's energy efficiency.

Reconstruction Loss Reconstruction loss is the difference between the input and output data of the autoencoder. It is used to evaluate how well the autoencoder model can reconstruct the input data, and, in turn, how well the autoencoder can learn the underlying structure of the input data. A popular loss function to attain this reconstruction error is the Mean Squared Error (MSE) loss function, which is used in numerous applications [35] [37] to evaluate an autoencoder's performance.

Peak Memory Footprint Peak memory footprint is the maximum amount of memory the autoencoder uses during training. It is used to evaluate how efficient the autoencoder is in terms of memory usage. This is an especially important metric as it directly affects how feasible it is for resource-constrained edge devices to run the model [38].

Latent Space Visualization Latent space visualization can be used to determine how well the autoencoder can learn the underlying structure of the input data. t-SNE (t-distributed Stochastic

Neighbor Embedding) is an effective technique for visualizing high dimensional data by giving each data point of the latent representation a location on a two-dimensional map [39]. Suppose t-SNE reveals distinct clustering in the latent space. In that case, the autoencoder has effectively learned to differentiate between different classes in the input data, leading to better performance on tasks such as anomaly detection and feature extraction.

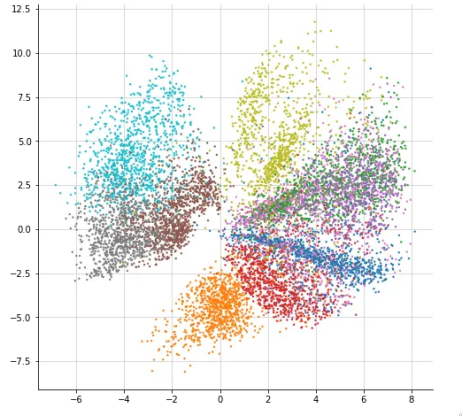

Figure 3: Example of a latent space visualization with good clustering

2.6 Challenges in Training Autoencoders on Edge Devices

Training autoencoders for real-world applications on edge devices, which often have limited computational resources, presents unique challenges. This section will discuss these challenges in detail.

Hyperparameter Tuning Hyperparameter tuning is the process of finding the optimal hyperparameters for a machine-learning model. When training autoencoders, numerous essential hyperparameters can be challenging to set. This includes the learning rate, weight cost, dropout fraction, batch size, number of epochs, number of nodes in each encoding and decoding layer, number of nodes in the bottleneck layer, the optimization algorithm and the kind of activation function [40]. Tuning this multitude of hyperparameters can significantly increase the computational demands of training an autoencoder. As such, an exhaustive or manual search of the parameter search space would be highly undesirable in terms of time and computational resources [41].

Autoencoder Architecture The architecture of an autoencoder is another important factor that can affect both the training time and the effectiveness of an autoencoder. A vanilla autoencoder often consists of a single-layer encoder and a single-layer decoder. However, increasing the depth of the autoencoder (adding more layers to the encoder and decoder) can offer many advantages. Because both the encoder and decoder parts of the autoencoder are feed-forward neural networks, the advantages of depth for feed-forward neural networks also apply to these components. These advantages include being able to enforce arbitrary constraints, such as making the latent representation sparse to approximate any mapping from input to code well, reducing the computational cost of representing some functions and decreasing the amount of training data needed to learn some functions [42].

However, having a deep architecture for autoencoders can increase the computational load for training an edge device if many hidden layers are used [43]. In addition, adding an unnecessary amount of neurons or layers might lead to overfitting, where the autoencoder could not generalize well with new data [43]. Therefore, it is essential to balance having enough depth to reap the benefits associated with depth in feed-forward neural networks whilst ensuring the computational load is the same as what an edge device with limited compute power can operate with.

Adjusting Weights with Backpropagation Backpropagation is a widely used training algorithm in autoencoders that uses gradient descent to adjust the weights of the autoencoder. In the forward pass of backpropagation, input data is processed from the input layer to the output layer to make a prediction. Once the prediction error is calculated from the target value, a backward pass from the output layer back to the input layer is made to adjust the weights of each neuron to minimize the prediction error. This process is repeated for multiple iterations until the network converges to a solution with a minimized error. In the case of large datasets, which autoencoders often require for effective training, or complex network architecture, such as deep autoencoders, these numerous iterations can significantly increase the training time [11]. In some cases, the network may not converge to a global minimum because many local minima exist, an issue that autoencoders with large initial weights are especially prone to [44]. Finally, backpropagation is also shown to consume a large amount of memory and power, which can pose practical limitations in the context of edge devices with resource constraints [45]

2.7 Optimization Techniques for Autoencoders

In order to overcome the challenges in training autoencoders, several strategies to reduce training time have been proposed in the literature. This section will cover some of the prominent strategies that are currently employed.

2.7.1 ELM-AE

ELM-AE (Extreme Learning Machine Auto-encoder) is a type of autoencoder that uses extreme machine learning (ELM). ELM is an algorithm for single-hidden layer feed-forward neural networks (SLFNs) such as vanilla autoencoders. The algorithm randomly chooses the input weights and analytically determines the output weights of SLFNs [46]. According to Huang et al, SFLNs with arbitrarily assigned input weights and hidden layer biases, along with any non-zero activation function, can universally approximate any continuous functions on any compact input sets, implying that the weights need not be adjusted [47].

The ELM algorithm uses the following equation to determine the output weights of the SLFNs:

$$f_{\bar{N}}(x_j) = \sum_{i=1}^{N} \beta_i G(a_i, b_i, x_j) = t_j, \quad j = 1, ..., N \tag{2}$$

where $G(a_i, b_i, x_j)$ is the activation function, a_i and b_i are the input weights and hidden layer biases respectively, t_j is the target output and $\bar{N}$ is the number of hidden nodes. This can be represented compactly as

$$H\beta = T$$

where

$$H = \begin{bmatrix} G(a_1, b_1, x_1) & \dots & G(a_{\bar{N}}, b_{\bar{N}}, x_1) \\ \vdots & \ddots & \vdots \\ G(a_1, b_1, x_N) & \dots & G(a_{\bar{N}}, b_{\bar{N}}, x_N) \end{bmatrix}_{N \times \bar{N}} \tag{3}$$

$$\tag{4}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_{\bar{N}^T} \end{bmatrix}_{\bar{N} \times m} \qquad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_{\bar{N}^T} \end{bmatrix}_{N \times m} \tag{5}$$

where H is the hidden layer output matrix, β is the output weight matrix and T is the target output matrix [48]. When the number of training samples *equals* the number of hidden nodes, the output weights β can be determined analytically by inverting H and realizing zero training error. When the number of training samples *exceeds* the number of hidden nodes, the pseudoinverse of H (using the Moore-Penrose generalised inverse) can be used to calculate the output weights while realizing a small nonzero training error [48]. This eliminates the large inefficiency of multiple iterations in the network, a process present in traditional backpropagation and other gradient descent methods.

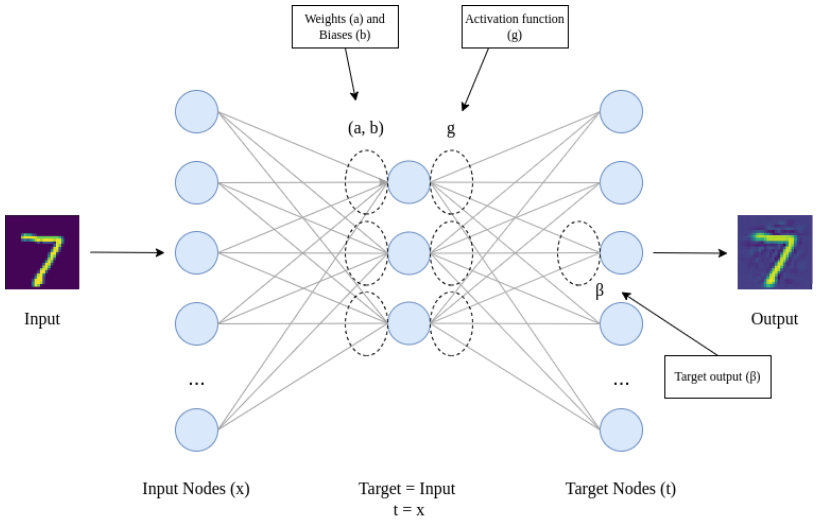

Figure 4: ELM-AE architecture, where the target output is the same as the input x and the hidden node parameters (a_i, b_i) are made orthogonal [49]

The architecture of ELM-AE is very similar to the original ELM algorithm, where the target output is the same as the input x and the weights of the hidden node parameters are orthogonal to each other to improve the generalization performance [49]. By combining ELM-AE with multilayer ELM (ML-ELM) to perform layer-by-layer unsupervised learning, Huang et al was able to achieve significantly faster learning than other deep networks, such as deep belief networks, stacked autoencoders and stacked denoising autoencoders [49]

2.7.2 OS-ELM

Variants of ELM have also been proposed to improve the training time of autencoders further. Some popular variants include OS-ELM (Online Sequential ELM).

OS-ELM Unlike ELM, which assumes that all the training data is available for training, OS-ELM is a variant of ELM where training data may arrive chunk-by-chunk or one-by-one, allowing for use in real-time applications [48]. Given a incoming chunk of data $\{x_i \in R^{k_i \times n}, t_i \in R^{k_i \times m}\}$ of batch size k, the output weight matrix β must now minimize

$$\left\| \begin{bmatrix} H_0 \\ \vdots \\ H_i \end{bmatrix} \beta_i - \begin{bmatrix} T_0 \\ \vdots \\ T_i \end{bmatrix} \right\| \tag{6}$$

where H_0 and t_0 are the hidden layer output matrix and target output matrix of the initial training chunk respectively and H_i and t_i are the hidden layer output matrix and target output matrix of the incoming chunk of data. Assuming

$$K_i \equiv \begin{bmatrix} H_0 \\ \vdots \\ H_i \end{bmatrix}^T \begin{bmatrix} H_0 \\ \vdots \\ H_i \end{bmatrix} \quad (i \geq 0), \quad P_i \equiv K_i^{-1} \tag{7}$$

the optimal output weight matrix can be obtained through the following equations:

$$P_i = P_{i-1} - P_{i-1} H_i^T (I + H_i P_{i-1} H_i^T)^{-1} H_i P_{i-1} \tag{8}$$

$$\beta_i = \beta_{i-1} + P_i H_i^T (t_i - H_i \beta_{i-1}) \tag{9}$$

When the training data is received one-by-one (i.e. $k_i = 1$), the output weight matrix β can be updated as follows:

$$P_i = P_{i-1} - \frac{P_{i-1} h_i h_i^T P_{i-1}}{1 + h_i^T P_{i-1} h_i} \tag{10}$$

$$\beta_i = \beta_{i-1} + P_i h_i (t_i^T - h_i^T \beta_{i-1}) \tag{11}$$

where $h_i = \big[G(a_1, b_1, x_i) \dots G(a_{\bar{N}}, b_{\bar{N}}, x_i)\big]$, thus avoiding the pseudoinverse operation $(I + H_i P_{i-1} H_i^T)^{-1}$ which is a major bottleneck in this algorithm [50].

E^2LM E^2LM is another variant of ELM that is designed for the distributed training of SLFNs. It introduces intermediate training results of the ELM algorithm that can be computed by and shared amongst multiple machines to integrate into their individual models [51]. As mentioned in the previous section, the output weight matrix β can be computed by taking the pseudoinverse of the hidden layer output matrix H:

$$\hat{\beta} = H^{\dagger} t \tag{12}$$

$$\Rightarrow \hat{\beta} = (H^T H)^{-1} H^T t \tag{13}$$

The intermediate training results can split Equation 13 into two parts, where $U = H^T H$ and $V = H^T t$. Thus, Equation 13 can be simplified as follows:

$$\hat{\beta} = U^{-1} V \tag{14}$$

When a new training sample x_i is added, the intermediate training results of the new sample are denoted as

$$U_i = H_i^T H_i \tag{15}$$

$$V_i = H_i^T t_i \tag{16}$$

To update the intermediate training results, the following equations are used:

$$U' = U_{i-1} + U_i \tag{17}$$

$$V' = V_{i-1} + V_i \tag{18}$$

Hence, the output weight matrix β can can be updated as follows [51]:

1. Compute U and V for the first batch of training data
2. Compute U_i and V_i for the new batch of training data using Equations 15 and 16
3. Update U' and V' using Equations 17 and 18
4. Compute β using Equation 14

Ito et al. propose combining elements of OS-ELM and E^2LM in order to train autoencoders on edge devices. Their method is made up of three phases: 1) Sequentially train the autoencoders (using OS-ELM), 2) Compute the intermediate results U and V and exchange them via a server (using E^2LM), and 3) Update individual models by integrating intermediate results from other edge devices [50].

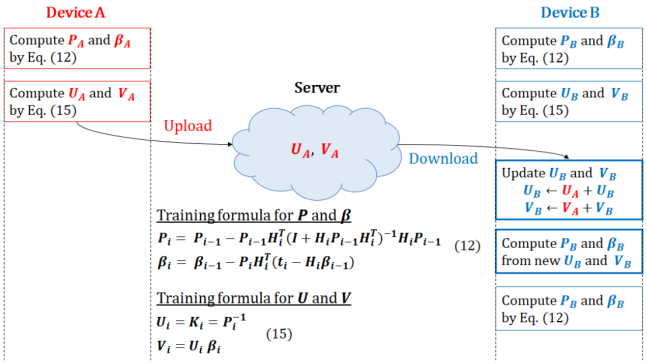

Figure 5: Flowchart of cooperative model update

Table 1: Specification of experimental machine

OS	Ubuntu 17.10
CPU	Intel Core i5 7500 3.4GHz
DRAM	8GB
Storage	SSD 480GB

Figure 5: OS-ELM and E^2LM for training autoencoders on edge devices [50]

Results obtained by utilizing this process for training autoencoders directly on the edge device outperformed traditional backpropagation-based autoencoder designed for federated learning in terms of both training and prediction latency [50]. In addition, by utilizing OS-ELM, this method is able to update the autoencoder's model in a *one-shot* manner, which beats backpropagation-based methods in terms of both computational and communication costs.

2.8 Towards a New Training Algorithm

From the above review, it is clear that traditional autoencoder training algorithms are not suitable for edge computing for the following reasons: 1) Traditional autoencoders require extensive hyperparameter tuning, which can be computationally demanding; 2) Complex autoencoder architectures can demand significant computational resources for training and 3) The backpropagation algorithm, central to traditional autoencoder training, involves numerous iterations through the network that cause substantial memory and power usage.

Some optimization algorithms, such as OS-ELM, ELM-AE, and E^2LM, have been proposed to address the shortcomings of traditional training algorithms. However, upon analyzing multiple papers on ELM-based solutions, there is a lack of documentation using these algorithms in an edge-computing context. Because of this lack of documentation, there is scant analysis regarding the memory and energy consumption these techniques require when deployed on edge devices.

In light of the factors impeding the deployment of traditional autoencoders on resource-constrained devices, along with the gap in the literature of state-of-the-art methods in terms of viable deployment on edge devices, the need for further exploration into the viability of using these techniques for edge devices, especially in terms of their computational efficiency, memory consumption and reconstruction performance is evident.

3 Requirements Specification & Ethics

This section will detail both the primary and secondary objectives of the project. The primary objectives represent the project's core goals, while the secondary objectives are additional goals that enhance the research's overall depth. The approach taken in exploring each objective and the degree of success achieved will be discussed for each objective.

3.1 Primary Objectives

The project's overarching goal is to address the challenges associated with deploying and training autoencoders on edge devices and optimize the training process such that an autoencoder can effectively function under the constraints of a given edge device. The following paragraphs detail the specific objectives pursued to achieve this goal.

Identify and address achievements and shortcomings in existing literature regarding reducing training and inference times of autoencoders. A literature review was conducted on autoencoders and their applications in edge computing. Over 40 research papers were read and analyzed, covering topics such as training neural networks, autoencoder architectures and applications with edge computing, challenges of traditional training techniques and current state-of-the-art methods to address these challenges. The review focused on identifying critical methodologies and results from these papers and understanding the gaps the project could aim to improve upon. This comprehensive literature review was successfully completed, providing a deep understanding of the field.

Develop an algorithm that reduces the time and memory consumption of training autoencoders while maintaining a high level of accuracy. After identifying the gaps in current methods of training autoencoders and researching some of the state-of-the-art methods aimed at addressing these gaps, implementing each of these methods, including OS-ELM and ELM-AE, was undertaken. Once implemented, each method was evaluated on various datasets, with their performance compared against each other. This process provided a deep understanding of the mechanisms driving these methods, which allowed for the development of further optimizations. These optimizations were implemented to be run on resource-constrained devices, leading to the conception of the AdaptAE framework.

Evaluate the algorithm's performance on its success in allowing autoencoders to be trained on a memory budget. Following the implementation of the AdaptAE framework, a thorough evaluation of the framework was conducted. This evaluation involved testing the framework on datasets with varying complexity and exploring the effects of each hyperparameter on key metrics, such as training time, peak memory usage and reconstruction loss. This exploration resulted in a deep understanding of the strengths and trade-offs between hyperparameters, allowing for an analytical default to be established for the AdaptAE framework that maintains a good balance between each of the metrics above.

3.2 Secondary Objectives

In addition to the primary objectives, this project also focuses on a set of secondary objectives that aim to explore and expand upon the applicability of the proposed solution in various contexts further. The following paragraphs detail these secondary objectives.

Benchmark the algorithm against suitable baselines that aim to reduce training times of autoencoders To benchmark AdaptAE, the solution was tested against a vanilla autoencoder as a traditional baseline model and ELM-AE as a state-of-the-art baseline model. Each model was trained on the same datasets (MNIST, Fashion-MNIST, CIFAR-10, CIFAR-100 and Tiny ImageNet) and evaluated on their performance on popular autoencoder tasks such as reconstruction and anomaly detection. Results obtained from these experiments indicated that AdaptAE achieved a significant reduction in training time and memory consumption while maintaining comparable accuracy, demonstrating its potential for deployment in edge computing environments.

Investigate the algorithm's performance on different memory budgets. When evaluating the AdaptAE framework, hyperparameters were varied to identify different modes suitable for devices with varying memory budgets. These modes include a sample mode optimized for devices with limited computing power and memory and a batch mode for devices with a more substantial memory budget. Despite being slower than the batch mode, the sample mode still outperforms a vanilla autoencoder in terms of training time and memory consumption. Furthermore, batch mode offers flexibility, making AdaptAE versatile across a wide range of edge devices with varying computing capabilities.

Generalize the algorithm to apply to other types of autoencoders, such as deep autoencoders. AdaptAE, rooted in the ELM framework, inherently operates on a single hidden layer. This architecture closely aligns with a vanilla autoencoder, allowing for a straightforward and comparable analysis of the key performance metrics. This approach highlights the potential for extending AdaptAE's principles to facilitate a deeper architecture, such as a deep autoencoder. An initial background exploration was conducted to understand the theory and feasibility of extending ELM-based solutions to incorporate additional hidden layers. However, due to the time constraints of this project, this exploration was only partially realized. Future research could extend AdaptAE to handle these deeper structures and investigate their impact on training time, memory consumption, and reconstruction loss.

3.3 Ethics

The project is purely a research and software development project; as such, there are no ethical considerations to be made, and a preliminary self-assessment form has been completed. This form can be found in the Appendix.

3.3.1 Secondary Datasets

This project will use a few secondary datasets to evaluate the algorithm's performance on different edge devices. These datasets are freely available to the research communities under relevant licenses and do not carry personal or sensitive information. The datasets are as follows:

- MNIST dataset
- Fashion-MNIST dataset
- CIFAR-10 and CIFAR-100 datasets
- Tiny ImageNet dataset

Details on these datasets can be found in Section 5.2.1

4 AdaptAE

AdaptAE is a framework that can be employed for real-time training of autoencoders on edge devices with limited memory capacities. It is designed to be flexible, allowing ML practitioners to analytically determine the best parameters based on the resource constraints of the desired edge device whilst guaranteeing a significantly faster training time than a traditional autoencoder. Researchers can use AdaptAE to address the following challenges:

1. **High energy consumption:** Low-powered edge devices can have a limited battery life, thus making it imperative to reduce the device's energy consumption. One major contributor to energy consumption when training neural networks on edge devices is the time to train the AI model [52]. AdaptAE can significantly reduce energy consumption through a markedly fast training speed, regardless of the memory budget of the device. Being an ELM-based solution, AdaptAE does not require iterative weight adjustments typical of a traditional autoencoder, nor does it require a search to find optimal parameters since this can be done analytically.

2. **Varying memory capacities:** Edge devices can vary significantly in terms of memory capacities. End devices, such as sensors, smart cameras, and smartwatches, typically have a few megabytes of memory. In contrast, edge servers are much more powerful and have more computational capabilities than end devices [53]. AdaptAE is an adaptive model that can run on memory-constrained end devices and more powerful edge servers. Similar to manipulating parameters to maximize training time, ML practitioners can *analytically* determine the suitable parameters to maximize the utilization of the device without exceeding its computational constraints.

3. **Real-time data processing:** Opposed to traditional autoencoders, which have to be trained on a cloud server before being deployed on edge devices due to their high computational requirements, AdaptAE allows for real-time data processing, adjusting its parameters from incoming sequential batches of data of varying sizes. This makes it suitable for dynamic environments and advantageous for scenarios in which data may not be readily available. Given that autoencoders are primarily used for tasks where unsupervised learning is key, AdaptAE's approach is particularly valuable for extracting meaningful information from incoming unlabelled data.

AdaptAE provides a significantly faster solution towards training autoencoders whilst providing both the flexibility to adapt the framework for devices with varying computational constraints and doing so in an analytical manner. This section will present some observations of traditional autoencoder training that motivated the concept of Adapt AE and provide an overview of the framework.

4.1 Motivation

As shown previously, traditional methods of training autoencoders can be unsuitable for on-device training due to their high computational requirements. With the need to first find a large number of optimal parameters through grid search and the need for iterative learning methods such as backpropagation to adjust the network's weights, training autoencoders can prove to be both time-consuming and computationally expensive. With current state-of-the-art methods, such as ELM and OS-ELM, a lack of analysis of the computational capability required for these models to be run on edge devices demonstrates the need for investigation. After extensive experimentation (as detailed in Section 5), these methods consumed a large amount of memory despite having a faster training time.

AdaptAE was developed to mitigate these issues with the following observations:

1. *Eliminating Iterative Learning Methods*: One of the main reasons traditional autoencoders can be both computationally expensive and time-consuming is the iterative learning methods used to adjust the network's weights. This is particularly true for backpropagation, which requires multiple iterations to find the optimal weights. ELM-based solutions such as AdaptAE, on the other hand, do not require iterative learning methods to adjust the network's weights. This is because the weights are randomly generated and must only be adjusted once, making AdaptAE significantly faster at training than traditional autoencoders.

2. *Analytical Parameter Tuning*: Another reason traditional autoencoders can be computationally expensive is the need to find optimal parameters through grid search. This is because the parameters of traditional autoencoders are not analytically determined. AdaptAE aims to resolve this issue by providing a framework where the parameters can be analytically chosen based on the application's goals.

4.2 Framework Overview

AdaptAE is a framework that leverages the batch learning capabilities of OS-ELM as its core component. This section will present an overview of the AdaptAE framework.

4.2.1 Phase I: Model Initialization

To initialize the AdaptAE model, the original OS-ELM algorithm is adapted to inherit the key characteristics of ELM-AE, allowing the architecture of the AdaptAE to have the same architecture as a traditional autoencoder. The first characteristic adopted from ELM-AE is that the number of input and output nodes is the *same*. In this way, AdaptAE can mimic the ELM-AE model's functionality, reconstructing any input data fed to it. When given the number of nodes for both the input and output layer of the model, the following parameters can be initialized:

Variable	Description	Shape
a	Connection weights between input and hidden layers	(n hidden nodes, n input nodes)
b	Additive bias for hidden layer nodes	(n hidden nodes, 1)
β	Weights between hidden and output layers	(n hidden nodes, n output nodes)
`p`	Intermediary matrix used for sequential learning	(n hidden nodes, n hidden nodes)

Table 1: AdaptAE Model Parameters

The second characteristic adopted from ELM-AE is that the weight and bias parameters are *orthogonalized*. This is done by initializing the weight matrix a and the bias vector b with random values and then applying an orthogonal initialization method to ensure that the parameters are orthogonal. This characteristic, as shown with ELM-AE, improves AdaptAE's ability to effectively adapt to new, unseen data, making it suitable to be deployed in real-time edge environments [49].

4.2.2 Phase II: Initial Learning

In this phase, the AdaptAE model is trained on an initial batch of data, which is processed in a single iteration. The following algorithm describes the process of initializing the AdaptAE model and learning on initial data:

Algorithm 1 AdaptAE Initial Learning

Data: Initial training data derived from the sequential proportion of entire training set $X_0 = 1 - seq\_prop * X$

Result: Initial prediction *init_pred*

$a \leftarrow \texttt{random()},\ b \leftarrow \texttt{random()}$

$a \leftarrow \texttt{orthogonalize}(a),\ b \leftarrow \texttt{orthogonalize}(b)$

$H_0 \leftarrow \texttt{g}(X_0 \cdot a + b)$

$p_0 \leftarrow \texttt{pinv}(H_0^T \cdot H_0)$

$\beta_0 \leftarrow p_0 \cdot H_0^T \cdot X_0$

$init\_pred \leftarrow H_0 \cdot \beta_0$ **return** *init_pred*

The model is first fed a *proportion* of the initial training data that the user determines. With this data, the hidden layer matrix is calculated by applying the activation function g to the dot product of the input data and the weight matrix a and then adding the bias vector b. Using the hidden layer matrix and the intermediary matrix p, the output matrix β can be calculated by applying the Moore-Penrose pseudoinverse to the dot product of the hidden layer matrix and the output data. The output matrix β is then used to calculate the initial prediction *init_pred* by applying the dot product of the hidden layer matrix and the output matrix β.

4.2.3 Phase III: Sequential Learning

In this phase, the AdaptAE model is now trained on batches of data fed to it sequentially. The following algorithm describes the process of sequential learning:

Algorithm 2 AdaptAE Sequential Learning

Data: Sequential training data derived from sequential proportion of entire training set $X_k = seq\_prop * X$

Result: Sequential prediction *seq_pred*

$H_{k+1} \leftarrow \texttt{g}(X_{k+1} \cdot a + b)$

if $k + 1\ batch\_size > 1$ **then**

  $p_{k+1} \leftarrow p_k - pk \cdot H_{k+1}^T \cdot \texttt{pinv}(I + H_{k+1} p_k \cdot H_{k+1}^T) \cdot H_{k+1} \cdot p_k$

  $\beta_{k+1} \leftarrow \beta_k + p_{k+1} \cdot H_{k+1}^T \cdot (X_{k+1} - H_{k+1} \cdot \beta_k)$

else

  $p_{k+1} \leftarrow p_k - \frac{p_k \cdot H_{k+1} \cdot H_{k+1}^T \cdot p_k}{1 + H_{k+1}^T \cdot p_k \cdot H_{k+1}}$

  $\beta_{k+1} \leftarrow \beta_k + p_{k+1} \cdot H_{k+1} \cdot (X_{k+1} - H_{k+1}^T \cdot \beta_k)$

end if

$seq\_pred \leftarrow H_{k+1} \cdot \beta_{k+1}$ **return** *seq_pred*

The proportion of the initial training data that is not used in the initial learning phase is used for sequential learning. As with the initial learning phase, the hidden layer matrix is calculated by applying the activation function g to the dot product of the input data and the weight matrix a and then adding the bias vector b. The intermediary matrix p and output matrix β are then calculated based on AdaptAE's mode. As shown in Algorithm 2, two modes determine the computations involved in this phase: **sample** mode, where the data is fed one sample at a time, and **batch** mode, where the data is fed in batches. Hence, if the batch size is set to 1, AdaptAE will operate under

sample mode, and if the batch size is greater than 1, AdaptAE will operate under *batch* mode. Once the output matrix has been calculated, it is then used to calculate the sequential prediction *_pred*.

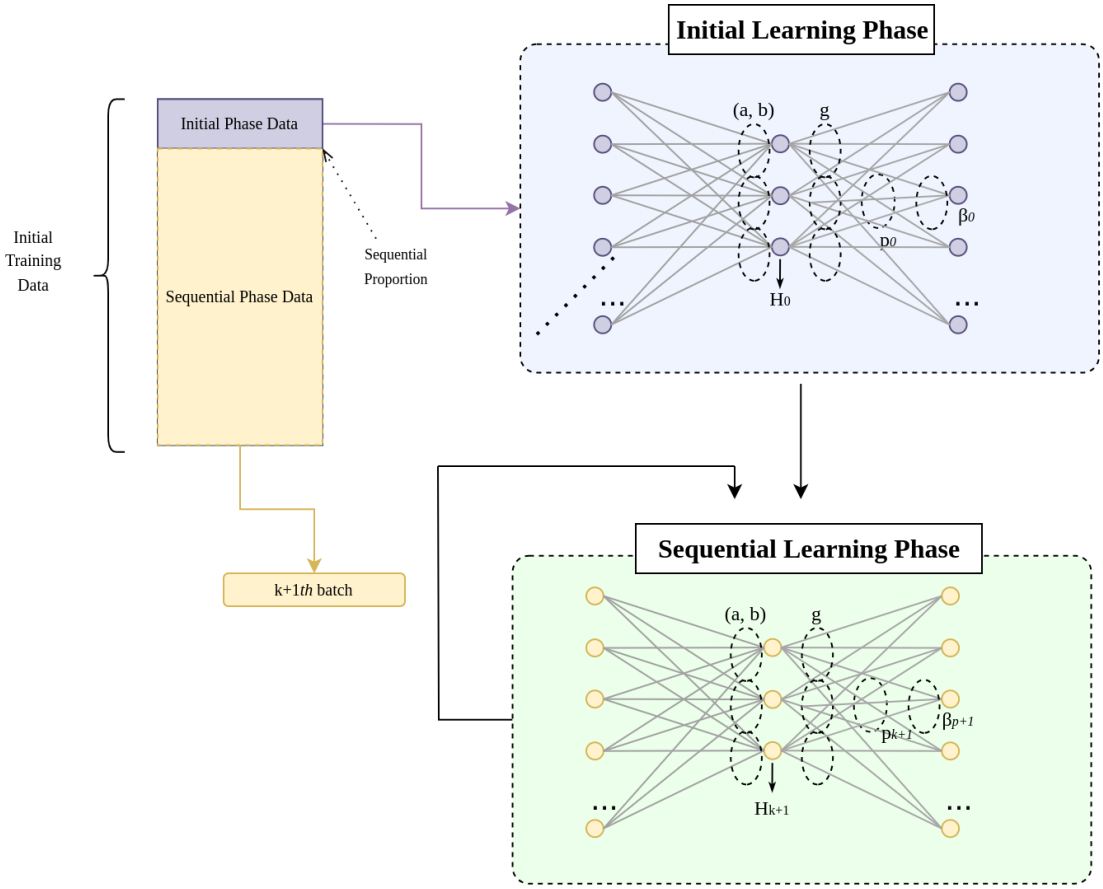

Figure 6: Training process of the AdaptAE Framework

4.2.4 Observations

In order to better understand the behaviour of AdaptAE, a series of experiments were conducted to understand the effect that each hyperparameter has on test loss, training time and peak memory usage. This section will explore critical findings that will allow users best to determine the optimal hyperparameters for their use case.

Observation 1: The amount of data used for sequential learning significantly impacts memory usage One significant difference between AdaptAE and OS-ELM is that AdaptAE does *not* use all of the initial training data available. In OS-ELM, the initialization phase uses *all* of the initial training data and the sequential phase is meant for continuous learning with incoming batches of data. AdaptAE, on the other hand, uses a *proportion* of the initial training data chosen by the user to initialize the model. In contrast, the rest of the initial data is used in the sequential

phase. This method can have a significant impact on the memory usage of the model.

Interestingly, the mode of AdaptAE (i.e. sample or batch) can prove to be a factor when determining the proportion of data to use for sequential learning, which can be shown in Figures 7 and 8:

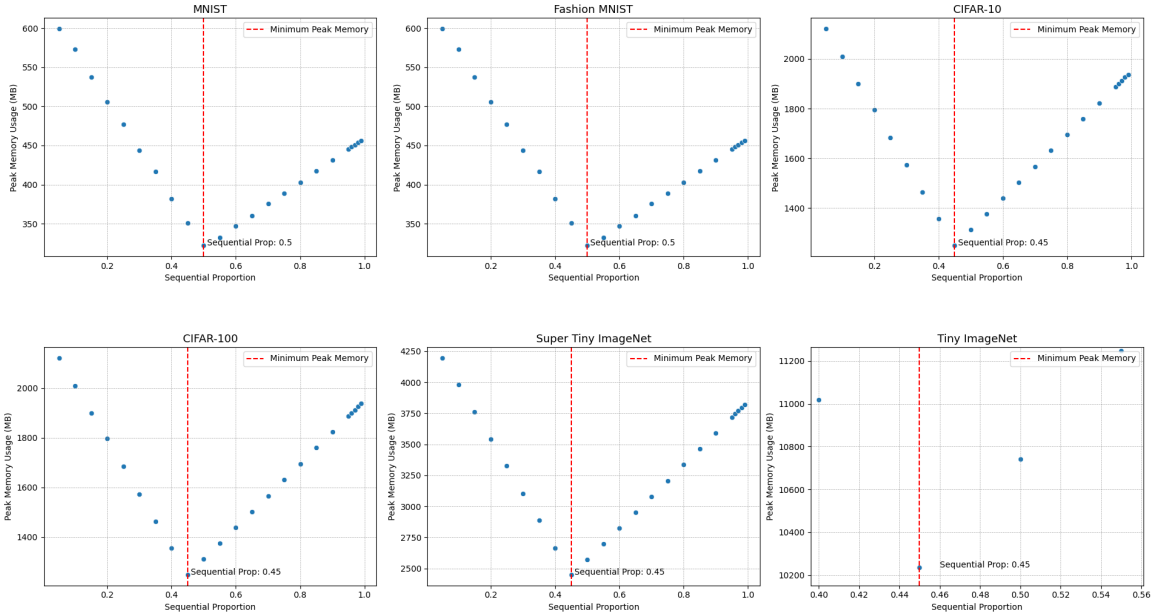

Figure 7: AdaptAE's relationship between proportions of data used for sequential learning phase and peak memory usage (batch size $\neq 1$)

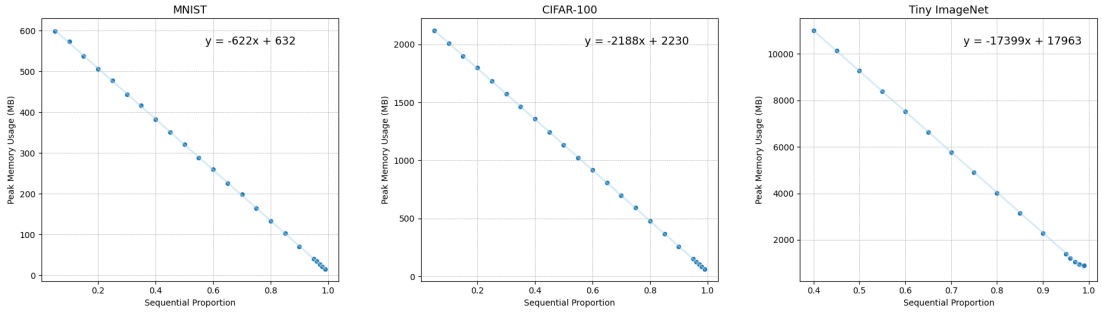

Figure 8: AdaptAE's relationship of proportions of data used for sequential learning phase and peak memory usage (batch size = 1)

AdaptAE on sample mode (batch size = 1) demonstrates a linear decrease in peak memory usage as the proportion of data used for sequential learning increases. In batch mode, however, the relationship between sequential proportion and peak memory usage exhibits a piecewise behaviour. As

with sample mode, the memory usage seems to decrease as the sequential proportion increases until a threshold, as indicated by the dashed vertical lines, where the memory starts to increase again. This threshold occurs when 55% of the initial training data is fed to the initial learning phase and 45% fed to the sequential phase.

From this observation, AdaptAE's default sequential proportion is set to **0.95** for sample mode and **0.45** for batch mode. This means that for sample mode, 97% of the initial training data is used for sequential learning, and 1% is used for initial learning. For batch mode, 45% of the initial training data is used for sequential learning and 55% is used for initial learning.

Observation 2: Memory usage increases as batch size increases The batch size used in the sequential learning phase is another factor that can impact the memory usage of AdaptAE. The most considerable difference in peak memory usage can be seen between the sample and batch mode, as can be seen in Figure 9.

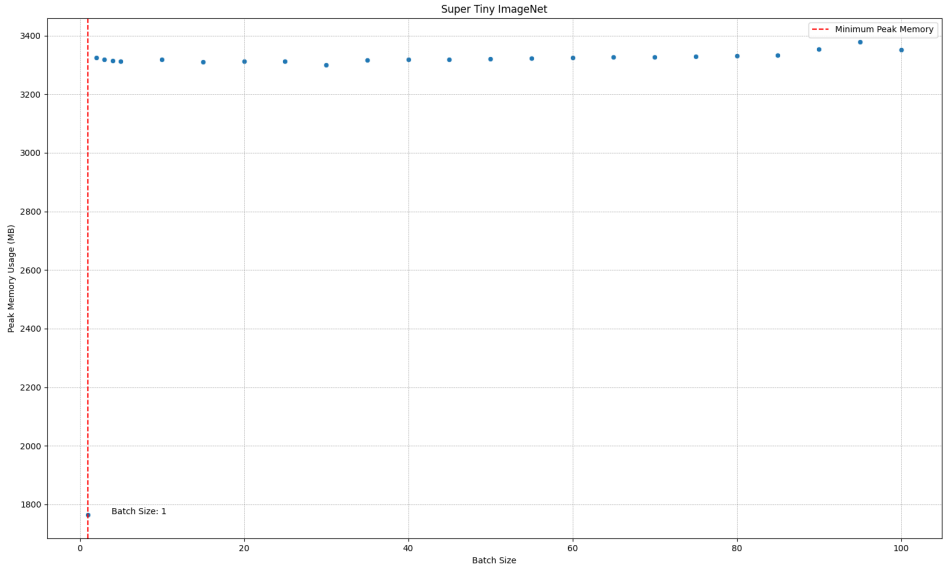

Figure 9: AdaptAE's relationship between batch size and peak memory usage

The peak memory usage of AdaptAE in sample mode is nearly two times lower than in batch mode. One potential cause is that the computations in sample mode do not include the Moore-Penrose inverse, which is used in batch mode. The Moore-Penrose inverse, which uses the SVD decomposition, is a computationally expensive operation that can be avoided in sample mode. One project motivation is to reduce peak memory usage; AdaptAE defaults to **sample** mode to avoid using the Moore-Penrose inverse and significantly conserve memory usage.

On batch mode, the peak memory usage of AdaptAE is also impacted by the batch size, which can be shown in Figure 10.

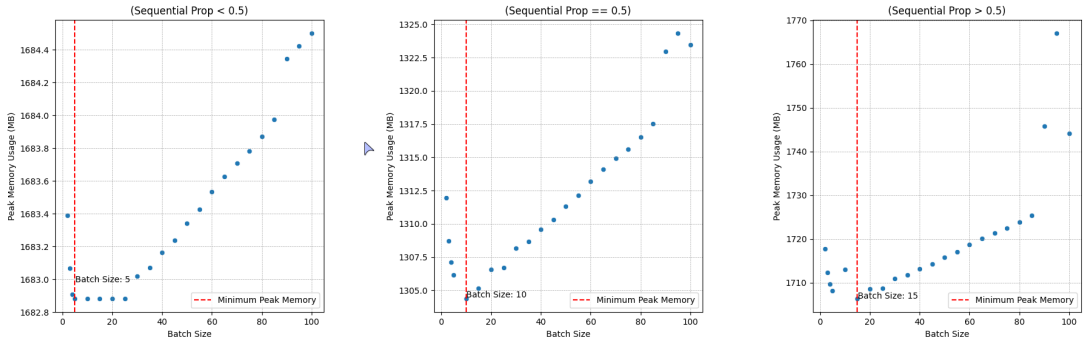

Figure 10: AdaptAE's relationship between batch size and peak memory usage (batch size $\neq 1$) using the CIFAR100 dataset. A sequential data proportion of 0.45 is recorded as the threshold, which is AdaptAE's sequential proportion to achieve the lowest peak memory usage for the CIFAR100 dataset (see Figure 7 for reference)

One interesting observation is that the peak memory usage of AdaptAE on batch mode can be determined by the sequential proportion used. The sequential proportion that minimizes peak memory usage, determined to be 0.45 in the previous observation, can serve as a key marker for memory use. Figure 10 shows the relationship between batch size and peak memory usage when using the minimum sequential proportion as a key marker when experimenting on multiple datasets of varying complexity (see Section 5.2.1 for details). Before reaching the minimum peak memory usage, there is a noticeable sharp decline in memory consumption. Beyond this point, however, the graphs in Figure 10 demonstrate a gradual linear increase continuing up to around a batch size of 80, after which the increase is relatively marginal.

Based on this observation, along with Observation 4, which shows an exponential decrease in training time as batch size increases, and Observation 6, which shows a weak relationship between batch size and test loss, and a general guideline for determining the recommended batch size based on the sequential proportion used is established in Table 2.

Sequential Proportion	**Recommended Batch Size**
$< min_{sq}$	$[5, 20]$
$= min_{sq}$	$[10, 30]$
$> min_{sq}$	$[15, 30]$

Table 2: Recommended batch sizes for different sequential proportions. min_{sq} is the minimum sequential proportion required to achieve the lowest peak memory usage for the given dataset. The lower limit optimizes for memory consumption. The upper limit is the value where the training time does not improve significantly with larger batch sizes, thus balancing training time and memory consumption. See Observation 4 for more details

Observation 3: Training time increases linearly as the quantity of data used for sequential learning increases There is an apparent linear increase in training time as more of the initial training data is used for the sequential phase. This is demonstrated in Figure 11.

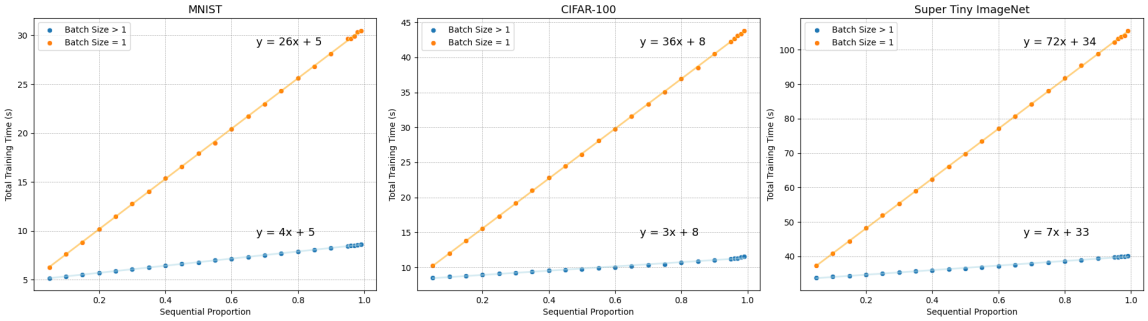

Figure 11: AdaptAE's relationship between proportions of data used for sequential learning phase and training time

As mentioned previously, AdaptAE operates on sample mode by default and has a default sequential proportion of 0.97. From this observation, it can be deduced that the default value places more importance on conserving memory usage than training time since it is imperative that the model can run on any edge device. Thus, users should note that the default values greatly minimize memory whilst sacrificing a higher training time.

Observation 4: Training time decreases exponentially as the batch size increases The batch size used in the sequential learning phase is another factor that can impact the training time of AdaptAE. As shown in Figure 12, the training time decreases exponentially as the batch size increases.

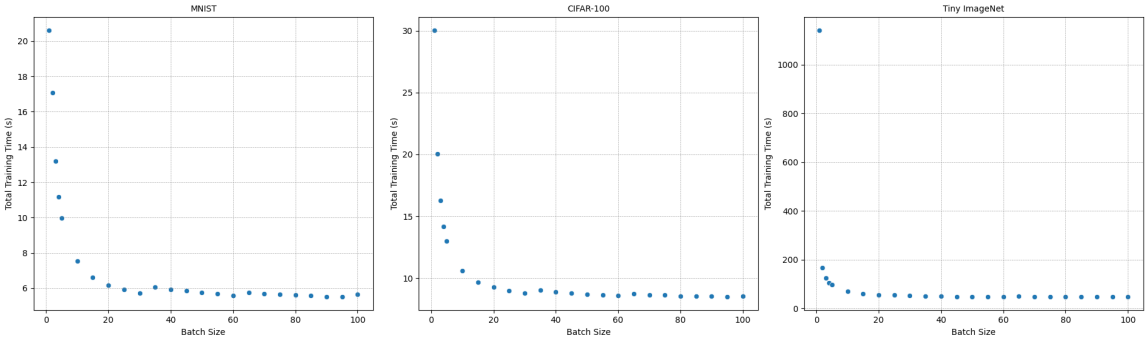

Figure 12: AdaptAE's relationship between batch size and training time

As mentioned in the previous observation, AdaptAE defaults to sample mode to conserve memory usage. However, if the user is willing to sacrifice memory usage for a faster training time, they can increase the batch size to achieve this.

Observation 5: Test loss increases exponentially as the sequential proportion increases The sequential proportion can significantly impact the test loss of AdaptAE. As shown in Figure 13, the test loss increases exponentially as the sequential proportion increases. After reaching a batch size of 30, further increases in batch size do not significantly improve training time.

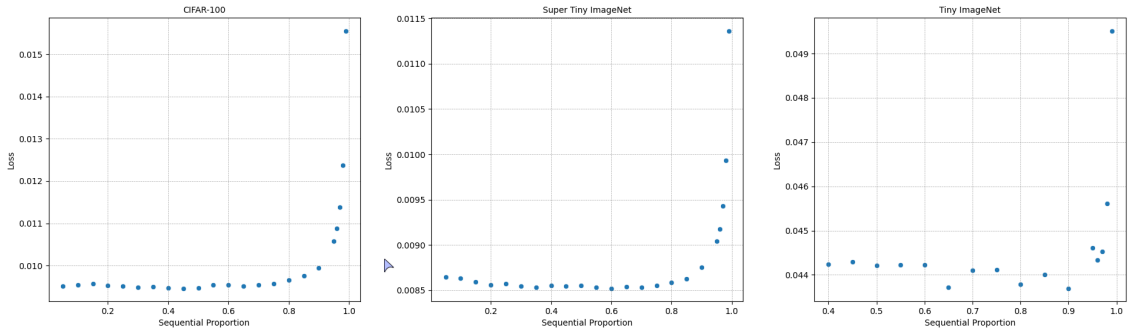

Figure 13: AdaptAE's relationship of proportions of data used for sequential learning phase and test loss

AdaptAE has a default sequential proportion of 0.97, which is the proportion that both minimizes peak memory usage whilst avoiding the substantial increase in loss that is observed at higher sequential proportions. However, when the memory constraint is especially high, the sequential proportion can be increased while sacrificing some reconstruction loss.

Observation 6: There a weak relationship between batch size and test loss The batch size has a weak relationship with the test loss of AdaptAE, suggesting that the batch size does not significantly impact the reconstruction loss of AdaptAE.

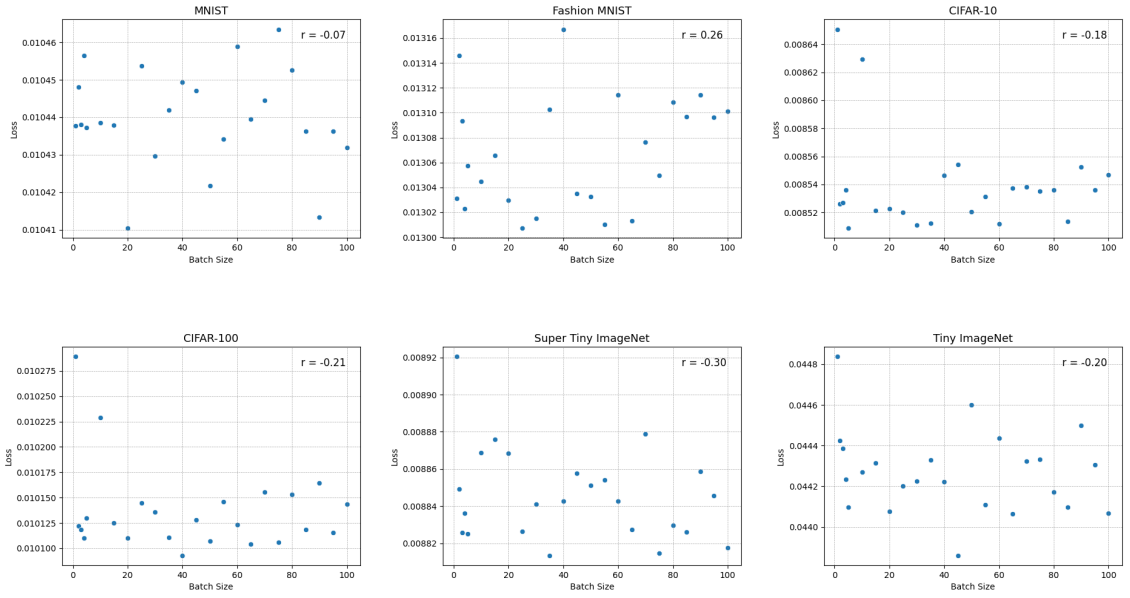

Figure 14: AdaptAE's relationship of batch size and test loss

As shown in Figure 14, the Pearson correlation coefficient between batch size and test loss is primarily negative, indicating a negative relationship between the two variables. However, the correlation coefficient with the highest absolute value is 0.3, showing that this relationship is very weak [54]. Therefore, the batch size is not a good indicator of the test loss of AdaptAE.

Summary The observations above demonstrate the trade-offs that must be made when using AdaptAE. These are the key takeaways from the observations:

- A sequential proportion of around 0.97 in sample and 0.45 in batch mode is identified as optimal for maintaining a good balance between reconstruction loss and memory usage.
- Maintaining the model in sample mode offers significant memory savings, which is crucial for edge devices often limited in computational resources.
- While increasing the batch size can speed up training time, it does so at the cost of more significant memory usage, highlighting a trade-off that users must consider based on their specific memory and time constraints.
- Finally, the weak relationship between batch size and reconstruction loss reinforces that batch size is not a significant indicator of AdaptAE's performance. Therefore, a careful balance between sequential proportion and batch size is necessary to minimize test loss without excessively increasing memory usage or training time.

5 Evaluation

This chapter will evaluate the proposed AdaptAE framework's performance and compare it to the baseline and state-of-the-art solutions. The structure of this chapter is as follows. Section 5.1 describes the tools and technologies used to develop and evaluate the AdaptAE, vanilla autoencoder and ELM-AE models. Section 5.2 details the experimental setup to assess and compare these models systematically. Section 5.3 justifies the key evaluation metrics used to evaluate each model. Finally, Section 5.4 reports the results of the baseline comparisons with AdaptAE.

5.1 Tools and Technologies

A significant factor in the project's success was ensuring that the tools and technologies utilized were appropriate and allowed for an in-depth understanding of the baseline, state-of-the-art, and proposed solutions. This section will detail the critical choices regarding programming languages, libraries' version control systems, and hardware support.

Programming Language Python was selected as the primary programming language for this project. It is a general-purpose language with extensive adoption in machine learning owing to its robust ecosystem of libraries and frameworks for machine learning and data visualization. Its popularity in machine learning and AI has fostered a large, active community that offers a wealth of knowledge through forums and discussions, such as the PyTorch Forums [55]. Community support and comprehensive documentation were invaluable resources for troubleshooting and extending the project's capabilities.

Machine Learning Libraries The project commenced with choosing between two prominent machine learning libraries: TensorFlow and PyTorch. Both are open-source and provide many tools for machine learning and deep learning applications.

The decision between TensorFlow and PyTorch was influenced by two primary factors. PyTorch's "Pythonic" approach offers intuitive use for those familiar with Python, as it is integrated closely with the language, unlike TensorFlow, built using C++. PyTorch's flexibility, resulting from its dynamic graph framework that allows the computational graph to be built as code executes, makes it a favoured choice among researchers requiring granular control over model architecture. On the other hand, TensorFlow's static graph framework requires pre-definition of the computational graph. The need for flexibility to facilitate adjustments during the training process of various state-of-the-art models made PyTorch the preferred library for this project.

Furthermore, PyTorch has CUDA support that allows for GPU acceleration, which is essential for training deep learning models. CUDA support is provided through the PyTorch CUDA Toolkit, a collection of libraries, tools, and extensions that enable GPU-accelerated computing using PyTorch [56]. Computations could, therefore, take advantage of PyTorch's ability to perform tensor operations on the GPU, enabling parallel processing and faster training.

Version Control Version control was managed with Git, using a private repository on GitHub. Regular commits and pushes were integral to maintaining consistency across different development environments, including laptops and GPU servers. Git also facilitated the review of the code's evolution, offering insights into the project's progression and critical developments at each stage.

GPU Support Development was conducted using a laptop with an NVIDIA GeForce RTX 3050 GPU, which was suitable for initial development and smaller dataset trials. However, to obtain results for the actual experiments of the project, the code was executed on an NVIDIA GeForce RTX 3060 GPU provided by the School of Computer Science. The specifications for the GPU are found in Table 3.

Memory	CUDA Cores	Tensor Cores	Tensor Performance	Power Consumption
12 GB GDDR6	3584	112	12.74 TFLOPS	170W

Table 3: Specifications of the NVIDIA GeForce RTX 3060 GPU used for experiments [57]

5.2 Experimental Setup

This section will detail the experimental setup for evaluating the AdaptAE framework. The section will describe the datasets used, how the data was processed, and how each model was constructed for a fair comparison. Additionally, the section will describe common autoencoder tasks used to evaluate each model.

5.2.1 Dataset Selection

Four datasets were selected for experimentation to assess the capability of image reconstruction across varying sizes and complexities. Each dataset varies in the number of images, image dimensions, and complexity, providing a comprehensive basis for evaluating model performance across different data types. The selected datasets are as follows:

MNIST The MNIST dataset is a set of handwritten digits consisting of 60,000 training images and 10,000 testing images, each being 28x28 pixels in size.

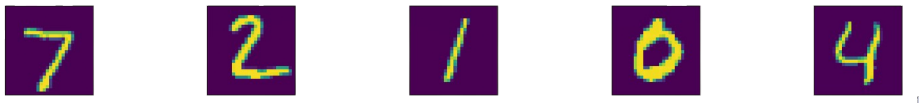

Figure 15: MNIST dataset

Fashion-MNIST The Fashion-MNIST dataset is similar to the MNIST dataset but consists of images of clothing items instead of handwritten digits. Like MNIST, it comprises 60,000 training images and 10,000 testing images, each being 28x28 pixels. Each image in this dataset represents products from 10 categories: trousers, shirts and dresses. Because each item has a different shape and texture, this dataset is slightly more complex than MNIST.

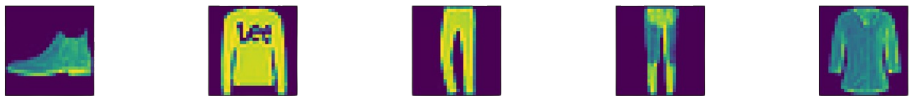

Figure 16: Fashion-MNIST dataset

CIFAR-10 The CIFAR-10 dataset ramps up the complexity by introducing colour images. It consists of 60,000 images of 10 different classes, each 32x32 pixels. Each image is also represented by three channels (RGB), which makes it more complex than the previous two datasets. The images are in 10 different classes and contain more intricate patterns than MNIST or Fashion-MNIST.

Figure 17: CIFAR-10 dataset

CIFAR-100 CIFAR-100 is similar to CIFAR-10 but contains 100 classes instead of 10. Like CIFAR-10, it contains 50,000 training and 10,000 testing images, with each image being 32x32 pixels and represented by three channels (RGB). This dataset adds to the challenge by significantly increasing the number of classes for each item.

Figure 18: CIFAR-100 dataset

Tiny ImageNet Tiny ImageNet is a scaled-down version of the ImageNet dataset. This dataset is significantly more complex than the previous ones, containing 100,000 training and 10,000 test images distributed across 200 classes. As with CIFAR-10 and CIFAR-100, each image is a colour image. The resolution, however, is also significantly higher than the previous datasets, with each image having a size of 64x64 pixels.

Figure 19: Tiny ImageNet dataset

5.2.2 Data Preprocessing

Each dataset underwent preprocessing to ensure compatibility with the typical architecture of the models under consideration. The only transformation performed for grayscale image datasets such as MNIST and Fashion-MNIST was converting images to PyTorch tensors, as detailed in the Implementation section. Data normalization was applied to standardize pixel values for more complex datasets, including CIFAR-10, CIFAR-100, and Tiny ImageNet. This normalization step is crucial for promoting faster model convergence, given that the sigmoid activation function employed by each model is susceptible to the scale of input data [58].

A new dataset named "Super Tiny ImageNet" was created by resizing the Tiny ImageNet dataset images to 32x32 pixels. This dataset was introduced due to the substantial increase in complexity from CIFAR-100 to Tiny ImageNet. The resizing of Tiny ImageNet provides an intermediate level of complexity, allowing models to be trained on a more challenging dataset without the computational

demands of the full-sized Tiny ImageNet. Moreover, the reduced size maintains sufficient image detail to present a challenge to each model effectively.

5.2.3 Model Selection

Given the array of variants for ELM and autoencoders, selecting relevant models for comparison with AdaptAE is essential. This section outlines the models implemented for experimentation and the reasons for their selection.

Autoencoder With various autoencoder types available, such as denoising autoencoders, convolutional autoencoders, and variational autoencoders, the decision was made to implement a vanilla autoencoder. Its architecture can be easily replicated using other ELM variants, as each ELM variant comprises a single hidden layer. This similarity enables a straightforward comparison between the vanilla autoencoder and ELM variants. Additionally, the vanilla autoencoder's simplicity makes the comparison more evident, particularly in understanding how well it learns and reconstructs data.

ELM-AE The ELM-AE model, a hybrid of a vanilla autoencoder and ELM was selected for implementation as it provides a baseline for comparison with the vanilla autoencoder and other ELM variants. This model is comparable to a vanilla autoencoder in terms of structural design and learning approach, making it particularly suitable for evaluating the impact of ELM principles on common autoencoder tasks.

5.2.4 Model Architecture

Ensuring direct comparability of the implemented models was essential, leading to the design of each model with an identical architecture. This design choice created a controlled environment for direct comparison of model performance. Consistency in architecture also meant that any observed differences in performance metrics were due to the model's design and not external factors.
The number of input and output nodes for each model is determined by the size of the input images and the number of colour channels in the images. The MNIST and Fashion-MNIST datasets contain grayscale images, meaning each image has a single colour channel. As such, since the size of each image is 28x28 pixels, the number of input and output nodes for these datasets is $28 * 28 * 1 = 784$. However, the CIFAR-10 and CIFAR-100 datasets contain three colour channels (red, green, blue) for each image and are 32x32 pixels in size. As such, the number of input and output nodes for these datasets is $32 * 32 * 3 = 3072$.

The number of input and output nodes determines the number of hidden nodes for each model. The number of hidden nodes for the MNIST and Fashion-MNIST datasets is $784/2 =$. The number of hidden nodes for the CIFAR-10 and CIFAR-100 datasets is $3072/2 = 1536$.

To summarize, the architecture of each model is as follows:

Dataset	Input & Output Nodes	Hidden Nodes
MNIST	784	128
Fashion-MNIST	784	128
CIFAR-10	3072	1024
CIFAR-100	3072	1024
Super Tiny ImageNet	3072	1024
Tiny ImageNet	12288	4096

Table 4: Model architecture

5.2.5 Hyperparameter Selection

This section will discuss the hyperparameter selection for the AdaptAE, autoencoder and ELM-AE models. For each model, the chosen parameters are intended to balance various aspects of the model's performance, including training time, peak memory usage and reconstruction loss, such that they can optimally run on resource-constrained devices. The methodology for choosing these parameters and the specific values are detailed below.

Autoencoder The autoencoder model, the most complex model under consideration due to its vast number of parameters, required extensive hyperparameter tuning to find the optimal parameters. The traditional method of hyperparameter tuning is a grid search, which does a complete search over a given subset of the hyperparameter space [59]. Utilizing grid search was essential for the autoencoder model as it is tested on various datasets, each with different characteristics. As such, the hyperparameters for the autoencoder model were tuned for each dataset individually. In this process, the learning rate and weight decay were selectively varied due to their significant influence on the learning dynamics [60], whilst other parameters were held constant. This approach allowed for a more balanced evaluation of the autoencoder's efficiency against other models. It is also important to note that because this process is required for training, the training time includes the time taken for hyperparameter tuning. The parameter space is detailed in Table 5 below.

Parameter	Description	Value
Learning Rate	Determines the step size at each iteration	[0.001, 0.01, 0.1]
Weight Decay	L2 Regularization parameter	[0.00001, 0.0001, 0.001]
Loss Function	Determines the loss function used for training	[MSE]
Activation Function	Determines the activation function used for training	[Tanh]
Optimizer	Determines the optimizer used for training	[Adam]
Batch Size	Determines the number of samples used per batch	[64]
Epochs	Determines the number of epochs to train for	[30]

Table 5: Autoencoder hyperparameter space for grid search

AdaptAE AdaptAE was configured with its default hyperparameters when comparing its performance to the other baseline models. These parameters were previously optimized to balance memory consumption, training time and reconstruction loss, as discussed in Chapter 4. Alongside the standard configuration, two additional settings were explored: a "performance" setting favouring training speed and reconstruction quality and a "low-memory" setting favouring low memory usage. The hyperparameter values for each setting are detailed in Table 6 below.

Parameter Setting	Batch Size	Sequential Proportion
Default	1	0.97
Low Memory	1	0.99
Performance	30	0.45

Table 6: AdaptAE hyperparameters for different parameter settings

The hyperparameters values for the performance setting were chosen based on two observations discussed in Chapter 4: Observation 1, which showed that AdaptAE on 'batch' mode has a minimum memory consumption when 50% of the initial training data is fed to the initial learning phase and 50% to the sequential learning phase, thus resulting in a sequential proportion of 0.45, and Observation 2, which provides the recommended batch size range when the sequential proportion bearing the lowest peak memory usage is chosen. The upper limit of this range was chosen to balance training time and memory consumption effectively. The same Mean Squared Error (MSE) loss function and hyperbolic tangent (Tanh) activation function were employed for consistency with the autoencoder.

ELM-AE ELM-AE training involves a single-shot learning approach, meaning the model is trained in a single iteration. As such, the only hyperparameter to be selected is the number of hidden nodes. This hyperparameter was selected based on the number of input and output nodes, as detailed in Section 5.2.4. As with the other models, the MSE loss function and Tanh activation function were employed.

5.2.6 Model Tasks

Two tasks were selected for experimentation: reconstruction and anomaly detection. These tasks were chosen as they critically reflect an autoencoder's ability to encode and decode information accurately. They are popular tasks in edge-computing contexts, where saving bandwidth through data compression and quick anomaly response is crucial. Each task was replicated five times for each model to gather enough data points for analysis. The following paragraphs detail the tasks and the rationale for their selection.

Reconstruction Reconstruction is a fundamental task for autoencoders, where the main aim is to learn a representation for a given set of data, typically for dimensionality reduction and feature learning, and reconstruct the original data as closely as possible. This task is crucial in autoencoders as it helps to understand and reveal the data's critical features while testing the model's capacity to reconstruct the input accurately. This makes it an essential measure for evaluating the model's performance.

Anomaly Detection Anomaly detection is another critical task where autoencoders are used to identify anomalies in data. For this task, autoencoders are evaluated based on their ability to learn the distribution of normal instances in the data. Anomalies are data points that deviate from this learned distribution and are thus reconstructed poorly. By evaluating the reconstruction error, anomalies can be detected effectively.

To create anomalies in the chosen datasets, random Gaussian noise was added to each image. The mean and standard deviation of the Gaussian noise were set to 0 and 0.5, respectively. This noise was added to the first 1500 images in the test set, leaving the training set unchanged. This approach was chosen to ensure that the model was not trained on any anomalous data, as this would have

resulted in the model learning to reconstruct the anomalies and thus failing to detect them.

Figure 20: Normal MNIST Image

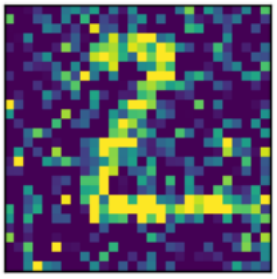

Figure 21: Corrupted MNIST Image

To determine if an image is anomalous, the reconstruction loss distributions for normal and anomalous samples were plotted and analyzed. A threshold for anomaly classification was established, calculated as the mean reconstruction loss of the normal samples plus three standard deviations. Samples with a reconstruction loss exceeding this threshold were labelled anomalous, while those below it were considered normal. This approach balances sensitivity to anomalies with the prevention of false positives.

One crucial note is that because ELM-AE operates in a single-shot learning manner, it is *impossible* to train the model on normal data and then test it on anomalous data. This is due to ELM-AE being unable to learn the distribution of normal data, a significant limitation that makes ELM-AE unsuitable for anomaly detection. As such, the model was excluded from the anomaly detection task.

5.3 Evaluation Metrics

In order to evaluate the performance of the models, several metrics were used to provide a comprehensive assessment of each model's performance. These metrics were chosen to offer insight into the practicality of each model for real-world edge computing scenarios, considering both resource constraints and performance. Table 7 outlines the consistent task-agnostic metrics that yield uniform results across both reconstruction and anomaly detection tasks. Table 8 details the criteria for the reconstruction task, where the metrics evaluate how well the model can learn and replicate the essential features of the input data. Finally, Table 9 outlines the metrics for the anomaly detection task, where the metrics determine how accurately each model can detect anomalies in the test data.

Performance Metric	Description
Training time	The time taken to train the model
Peak Memory Usage	The highest amount of memory occupied by the model during training

Table 7: Evaluation metrics used for the reconstruction and anomaly detection tasks

Performance Metric	Description
Test Loss	The loss value for each epoch during testing
Latent Space Visualization	A visual representation of the latent space
Reconstruction Images	A visual representation of the model's ability to reconstruct data

Table 8: Evaluation metrics for the reconstruction task

Performance Metric	Description
Accuracy	The proportion of anomalies and normal instances that were correctly identified
Precision	The proportion of anomalies identified that were actually anomalies
Recall	The proportion of anomalies that were correctly identified
F1 Score	The harmonic mean of precision and recall

Table 9: Evaluation metrics for the anomaly detection task

5.4 Results

The following section details the results of the reconstruction and anomaly detection tasks based on the evaluation metrics outlined in Tables 7, 8 and 9.

5.4.1 General Performance

Figures 22 and 23 depict the performance of AdaptAE, ELM-AE and the traditional autoencoder in terms of training time and peak memory usage on all the datasets involved in the study. Regarding training efficiency, ELM-AE emerged as the frontrunner, demonstrating the shortest training times, with AdaptAE following closely behind. The autoencoder, however, performed significantly slower than both models across all datasets. Results from the Super Tiny ImageNet dataset demonstrated the largest difference between the models, with AdaptAE performing 12.12 times faster than the autoencoder. Tiny ImageNet also showed a notable disparity between AdaptAE and autoencoders, with AdaptAE performing 2.67 times faster.

Despite ELM-AE's superior training efficiency, the model performed consistently worse than both AdaptAE and the autoencoder in terms of peak memory usage. Notably, in Figure 23, ELM-AE's

data point is absent for the Tiny ImageNet dataset, indicating the model's limitation despite using the relatively powerful school GPU. AdaptAE, on the other hand, significantly outperformed both models in terms of memory consumption across all datasets, except for the MNIST dataset, which consumed a mere 4.48 megabytes compared to the autoencoder. In the case of Tiny ImageNet, the autoencoder consumed over 2 gigabytes more memory than AdaptAE's. One interesting observation from the data shows that the memory consumption of the autoencoder for CFIAR-100 and Super Tiny ImageNet remained the same, whereas the memory consumption increased between these datasets with AdaptAE.

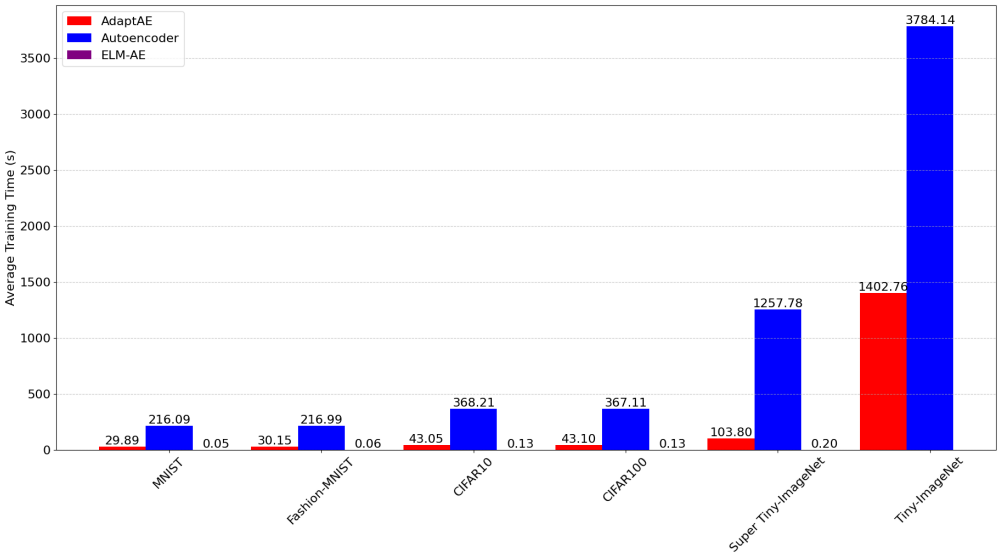

Figure 22: Comparison of training time for a vanilla autoencoder, ELM-AE and AdaptAE with default parameter settings on each dataset. ELM-AE produced no result for the Tiny ImageNet dataset as it exceeded memory constraints of the GPU

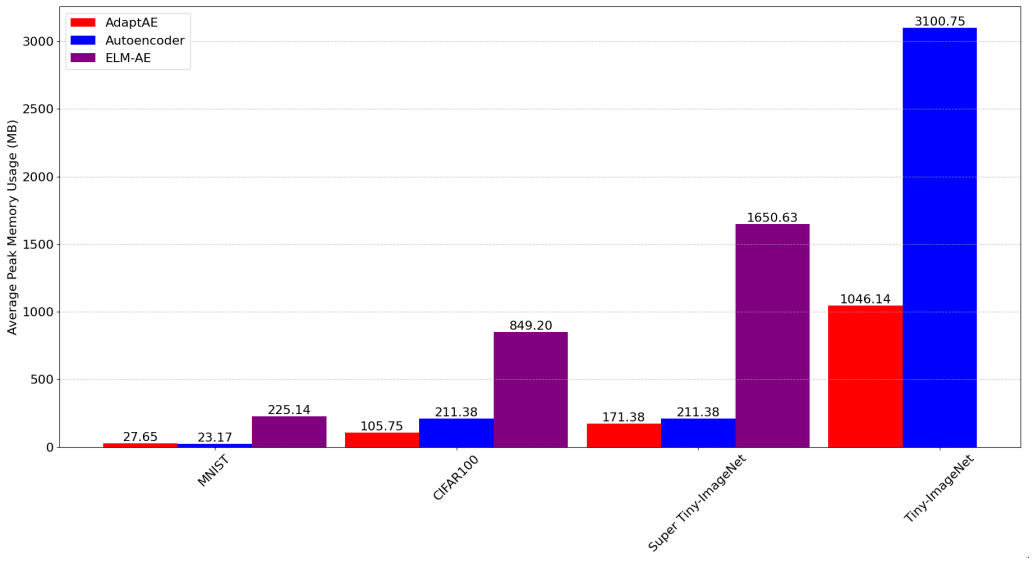

Figure 23: Comparison of peak memory usage for a vanilla autoencoder, ELM-AE and AdaptAE with default parameter settings on the MNIST, CIFAR-100, Super Tiny-ImageNet and Tiny ImageNet datasets. Fashion-MNIST and CIFAR-10 datasets were excluded as they provided the same results as MNIST and CIFAR-100 respectively. ELM-AE produced no result for the Tiny ImageNet dataset as it exceeded memory constraints of the GPU

5.4.2 Reconstruction

In the reconstruction task, Table 10 reports the results for each model on the Fashion-MNIST, CIFAR-100 and Super Tiny ImageNet to investigate whether a trend in RMSE scores exists whilst increasing the complexity of the dataset. Across all models, ELM-AE had the highest RMSE scores across each dataset, more than two times higher than the autoencoder and AdaptAE models. The next lowest-performing model was AdaptAE in low-memory mode, where the reconstruction error was consistently worse than other AdaptAE settings and the autoencoder. Whilst it has comparable performance to other AdaptAE settings for the Fashion-MNIST dataset, it does not perform well for the larger and more complex ones.

The autoencoder, AdaptAE with default settings and AdaptAE with performance settings all produce comparable results. Out of these three models, AdaptAE with the default settings had the highest reconstruction loss for both the Fashion-MNIST and CIFAR-100 datasets. Interestingly, the default setting seems to shine with the Super Tiny ImageNet dataset, where it performs slightly better (0.001 MSE) than the autoencoder. AdaptAE on the performance setting, though only matching the default AdaptAE with the Fashion-MNIST dataset, matches or outperforms the autoencoder model for the larger datasets.

Datasets	**Model**	**Average Test RMSE**
Fashion-MNIST	AdaptAE (Default)	0.0129
	AdaptAE (Low Memory)	0.0129
	AdaptAE (Performance)	0.0128
	Autoencoder	0.0091
	ELM-AE	0.0271
CIFAR-100	AdaptAE (Default)	0.0123
	AdaptAE (Low Memory)	0.0155
	AdaptAE (Performance)	0.0096
	Autoencoder	0.0096
	ELM-AE	0.0306
Super Tiny ImageNet	AdaptAE (Default)	0.0103
	AdaptAE (Low Memory)	0.0113
	AdaptAE (Performance)	0.0085
	Autoencoder	0.0103
	ELM-AE	0.0221

Table 10: Reconstruction performance of AdaptAE, the autoencoder and ELM-AE on the Fashion-MNIST, CIFAR-100 and Super Tiny ImageNet datasets

Figures 24, 25 and 26 depict the latent space representation of the MNIST dataset for AdaptAE, the autoencoder and ELM-AE, respectively. From a visual standpoint, the autoencoder and AdaptAE show a similar level of clustering, with well-defined clusters for each digit class. The autoencoder model shows a slightly better separation of clusters than AdaptAE, particularly for digits 4, 7 and 9. ELM-AE offers decent clustering, but the overlap between digits 4, 7 and 9 is much more apparent. Additionally, there seems to be less cohesion amongst the cluster, with more data points scattered around the map.

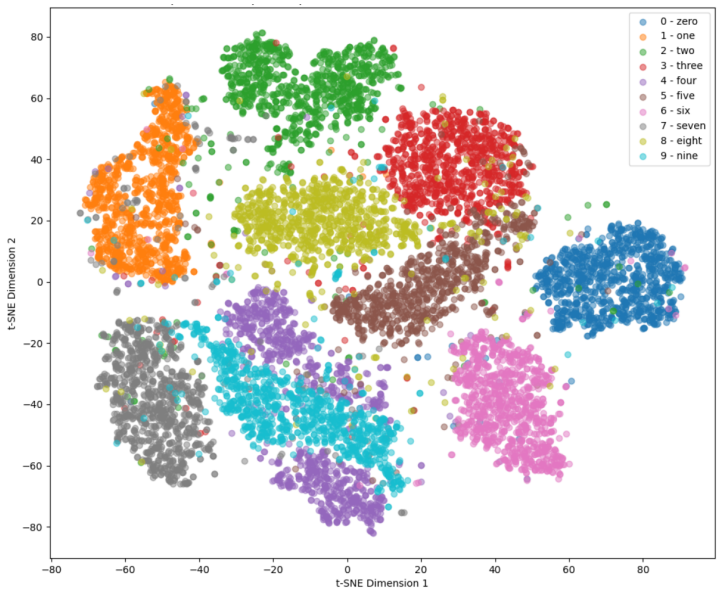

Figure 24: Latent space representation of the MNIST dataset for AdaptAE with default parameter settings

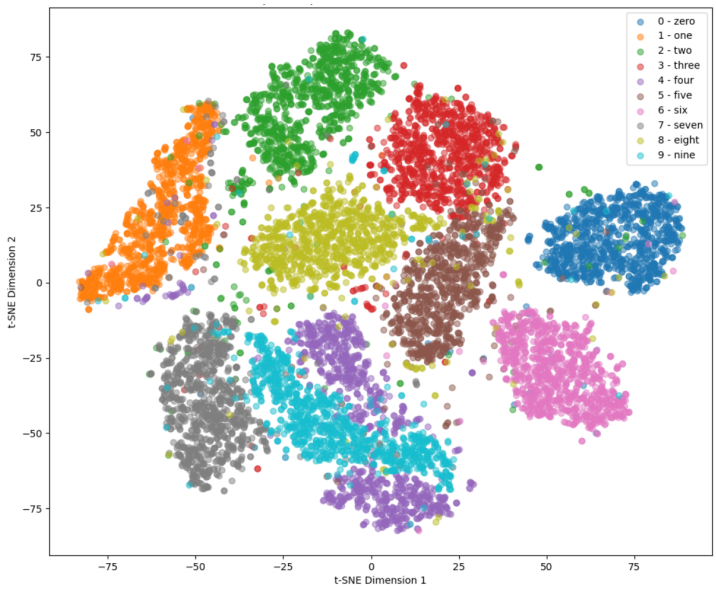

Figure 25: Latent space representation of the MNIST dataset for the autoencoder model

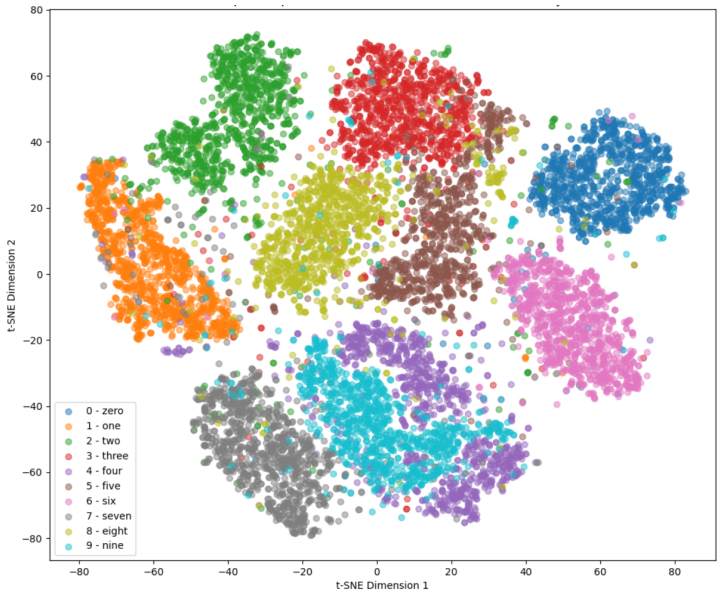

Figure 26: Latent space representation of the MNIST dataset for ELM-AE

Figures 27, 28 and 29 demonstrate the original images from the Super Tiny ImageNet dataset and the reconstructed images produced from each model. The ELM-AE model reconstructions seem to be blurry and lack much detail. The images also have a lower contrast, and some original image elements appear very faintly, such as the object in the first image. AdaptAE, with the default configuration, and the autoencoder have more comparable reconstructions with a higher image quality and colour preservation.

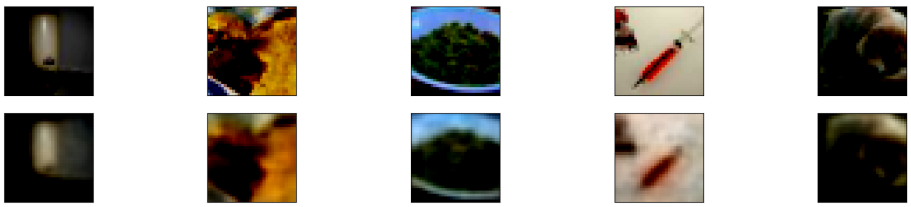

Figure 27: ELM-AE's reconstruction of five sample images from the Super Tiny ImageNet dataset for the reconstruction task

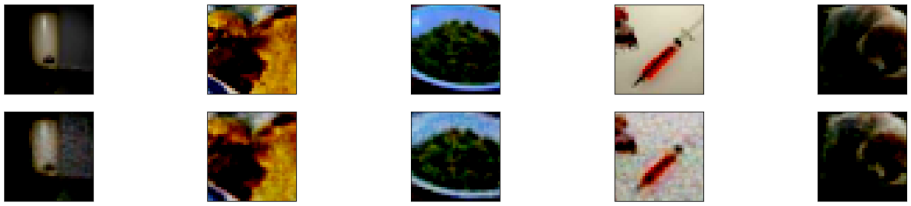

Figure 28: Autoencoder's reconstruction of five sample images from the Super Tiny ImageNet dataset for the reconstruction task

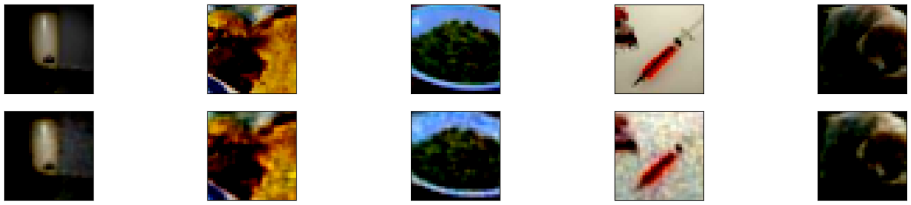

Figure 29: AdaptAE with default configurations' reconstruction of five sample images from the Super Tiny ImageNet dataset for the reconstruction task

5.4.3 Anomaly Detection

In the anomaly detection task, Table 11 reports the results for each model on the Fashion-MNIST, CIFAR-100, and Super Tiny ImageNet to investigate the capabilities of each model to identify anomalies as the datasets increase in complexity. For the Fashion-MNIST dataset, AdaptAE with the performance configuration stands out with the highest F1 score of 98.04% and precision of 96.1%. The default and low-memory settings of AdaptAE perform slightly worse (0.16% and 0.21%, respectively) in terms of the F1 score compared to the autoencoder model. Each model, however, achieves a 100% recall score.

For the CIFAR-100 dataset, the autoencoder model achieves perfect scores across all metrics, with AdaptAE's performance settings as a very close second (1.1% lower F1 score and 0.3% lower accuracy), despite matching the autoencoder's recall performance. AdaptAE's default and low-memory modes, however, have a 3.3% and 5.04% lower accuracy, respectively, than the traditional autoencoder. These modes also perform 11.15% lower and 18.19% lower in terms of F1 score.

For the Super Tiny ImageNet dataset, the autoencoder and AdaptAE with the performance configuration achieve perfect scores across all metrics. AdaptAE's default and low-memory settings show a reduction in both accuracy and F1 score compared to the autoencoder, with AdaptAE achieving a 96.01% accuracy and 90.01% F1 score with the default configurations and 96.51% accuracy and 88.13% F1 score with the low-memory settings. However, the results obtained for both models still indicated a high level of performance.

Datasets	**Model**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
Fashion-MNIST	AdaptAE (Default)	98.65%	91.7%	100%	95.67%
	AdaptAE (Low Memory)	98.61%	91.6%	100%	95.62%
	AdaptAE (Performance)	99.40%	96.1%	100%	98.04%
	Autoencoder	98.73%	92%	100%	95.83%
CIFAR-100	AdaptAE (Default)	96.70%	89.90%	87.87%	88.85%
	AdaptAE (Low Memory)	94.96%	89.28%	75.47%	81.81%
	AdaptAE (Performance)	99.7%	98.04%	100%	98.99%
	Autoencoder	100%	100%	100%	100%
Super Tiny ImageNet	AdaptAE (Default)	96.01%	89.30%	90.73%	90.01%
	AdaptAE (Low Memory)	96.51%	89.99%	86.33%	88.13%
	AdaptAE (Performance)	100%	100%	100%	100%
	Autoencoder	100%	100%	100%	100%

Table 11: Anomaly detection performance metrics for AdaptAE and the autoencoder model on the Fashion-MNIST, CIFAR-100 and Super Tiny ImageNet datasets

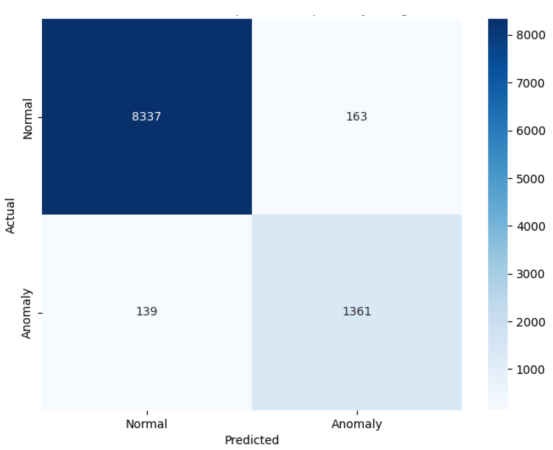

Figure 30: Confusion matrix for AdaptAE on the Super Tiny ImageNet dataset using the default configuration

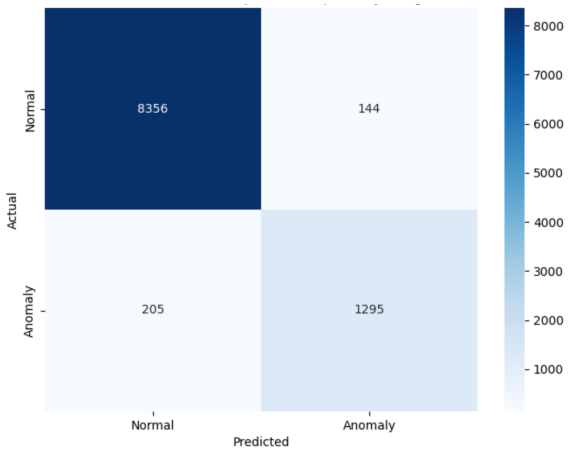

Figure 31: Confusion matrix for AdaptAE on the Super Tiny ImageNet dataset using the low-memory configuration

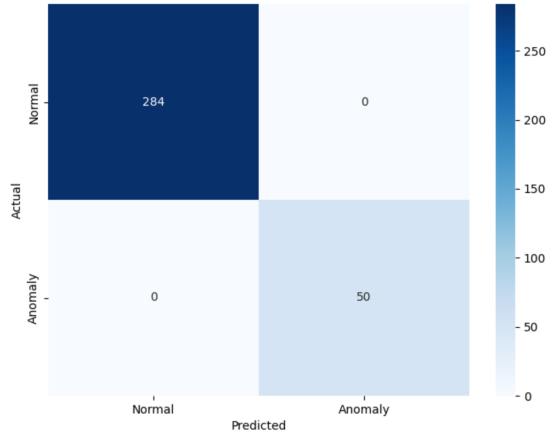

Figure 32: Confusion matrix for AdaptAE on the Super Tiny ImageNet dataset using the performance configuration

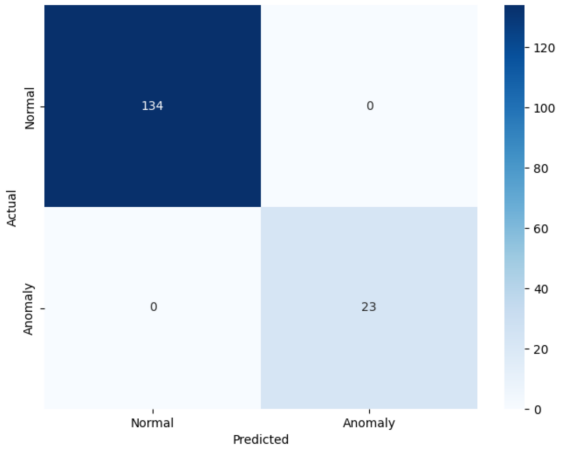

Figure 33: Confusion matrix for the autoencoder model on the Super Tiny ImageNet dataset

6 Conclusions and Future Work

Traditional autoencoder training is impractical for on-device training of resource-constrained devices due to the computational load that arises with hyperparameter tuning, iterative learning methods and complex architectures. Whilst techniques aim to reduce the computational load by improving training time, notable gaps in the literature regarding memory consumption and reconstruction loss can hinder its practicality for on-device training. This project presents AdaptAE, a framework that addresses these gaps by proposing a novel autoencoder training method that reduces the training time and memory consumption whilst achieving a reasonable reconstruction loss and anomaly detection accuracy. The results of this project demonstrate that AdaptAE can be used to train an autoencoder model on-device with minimal memory consumption and reconstruction loss. AdaptAE reduces the memory consumption by up to 66% for the traditional autoencoder and up to 89% for ELM-AE whilst also reducing the training time of the autoencoder by up to 62%.

6.1 Limitations & Future Work

Results from the reconstruction and anomaly detection tasks reveal that AdaptAE, with the performance settings, is able to match and sometimes even outperform traditional autoencoders in terms of reconstruction loss and anomaly detection accuracy, all whilst reducing the training time of the autoencoder by a significant margin. However, as shown in the observations of the framework, there is a significant increase in memory consumption when a batch size higher than one is used. Employing a batch size of 30 for the performance setting has a sharp increase from both AdaptAE with the default settings and the autoencoder. Future work could look into quantization, a technique that replaces full-precision weights and activations with lower-precision representations [61] as a means to reduce the memory consumption involved in the computations of batch mode, specifically the calculation of the Moore-Penrose inverse.

Additionally, one major limitation of the AdaptAE framework is that it is limited to a single hidden layer, much like other ELM-based solutions. This can prove inefficient compared to autoencoders with a deeper architecture, which have been shown to reduce the computational cost and decrease the amount of training data needed [42]. Future work can consider employing techniques for multiple AdaptAE components to be stacked together, mimicking a deeper architecture.

Finally, the current scope of the AdaptAE framework is limited to the training of traditional autoencoders, which, while beneficial, represents only a subset of applications for which autoencoders can be used. Future work can aim to expand on the framework's capabilities to encompass a wider range of autoencoder variants such as variational, denoising and convolutional autoencoder. By doing so, AdaptAE can be used as a more comprehensive tool for efficient on-device learning, allowing edge devices to perform a broader spectrum of tasks.

References

[1] A. G. Ivakhnenko. "Polynomial Theory of Complex Systems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-1.4 (1971), pp. 364–378. DOI: 10.1109/TSMC.1971.4308320.

[2] Shiliang Sun et al. "A Survey of Optimization Methods From a Machine Learning Perspective". In: *IEEE Transactions on Cybernetics* 50.8 (2020), pp. 3668–3681. DOI: 10.1109/TCYB.2019.2950779.

[3] Haohan Wang and Bhiksha Raj. "On the Origin of Deep Learning". In: (Feb. 2017).

[4] Jorge Pérez et al. "Edge computing". In: *Computing* 104.12 (Dec. 2022), pp. 2711–2747. ISSN: 1436-5057. DOI: 10.1007/s00607-022-01104-2. URL: https://doi.org/10.1007/s00607-022-01104-2.

[5] Umberto Michelucci. *An Introduction to Autoencoders*. 2022. arXiv: 2201.03898 [cs.LG].

[6] Dor Bank, Noam Koenigstein, and Raja Giryes. *Autoencoders*. 2021. arXiv: 2003.05991 [cs.LG].

[7] Lovedeep Gondara. "Medical Image Denoising Using Convolutional Denoising Autoencoders". In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016, pp. 241–246. DOI: 10.1109/ICDMW.2016.0041.

[8] Antoine Plumerault, Hervé Le Borgne, and Céline Hudelot. "AVAE: Adversarial Variational Auto Encoder". In: *CoRR* abs/2012.11551 (2020). arXiv: 2012.11551. URL: https://arxiv.org/abs/2012.11551.

[9] Sultan Zavrak and Murat İskefiyeli. "Anomaly-Based Intrusion Detection From Network Flow Features Using Variational Autoencoder". In: *IEEE Access* 8 (2020), pp. 108346–108358. DOI: 10.1109/ACCESS.2020.3001350.

[10] Mahdi Rezapour. "Anomaly Detection using Unsupervised Methods: Credit Card Fraud Case Study". In: *International Journal of Advanced Computer Science and Applications* 10.11 (2019). DOI: 10.14569/IJACSA.2019.0101101. URL: http://dx.doi.org/10.14569/IJACSA.2019.0101101.

[11] David Novoa-Paradela, Oscar Fontenla-Romero, and Bertha Guijarro-Berdiñas. "Fast deep autoencoder for federated learning". In: *Pattern Recognition* 143 (2023), p. 109805. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2023.109805. URL: https://www.sciencedirect.com/science/article/pii/S0031320323005034.

[12] B S Sharmila and Rohini Nagapadma. "Quantized autoencoder (QAE) intrusion detection system for anomaly detection in resource-constrained IOT devices using RT-IOT2022 dataset". In: *Cybersecurity* 6.1 (2023). DOI: 10.1186/s42400-023-00178-5.

[13] Nils Kuever. *Cloud and edge computing for IOT: A short history*. Feb. 2023. URL: https://blog.bosch-digital.com/cloud-and-edge-computing-for-iot-a-short-history/#:~:text=The%20origin%20of%20edge%20computing,such%20as%20images%20and%20videos..

[14] Haochen Hua et al. "Edge Computing with Artificial Intelligence: A Machine Learning Perspective". In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3555802. URL: https://doi.org/10.1145/3555802.

[15] *What is edge computing?* URL: https://www.accenture.com/gb-en/insights/cloud/edge-computing-index.

[16] *What is edge computing?* URL: https://www.ibm.com/topics/edge-computing.

[17] Mark Olding. *What are edge computing devices?* Sept. 2023. URL: https://www.exorint.com/exor-innovation-blog/what-are-edge-computing-devices.

[18] Ruizhe Yang et al. "Integrated Blockchain and Edge Computing Systems: A Survey, Some Research Issues and Challenges". In: *IEEE Communications Surveys Tutorials* 21.2 (2019), pp. 1508–1532. DOI: 10.1109/COMST.2019.2894727.

[19] Jafar Shayan et al. "Identifying Benefits and risks associated with utilizing cloud computing". In: *CoRR* abs/1401.5155 (2014). arXiv: 1401.5155. URL: http://arxiv.org/abs/1401.5155.

[20] Abhay Verma and Vinay Verma. "Comparative Study of Cloud Computing and Edge Computing: Three Level Architecture Models and Security Challenges". In: 9 (Aug. 2021).

[21] Bhaskar Prasad Rimal, Dung Pham Van, and Martin Maier. "Mobile-Edge Computing Versus Centralized Cloud Computing Over a Converged FiWi Access Network". In: *IEEE Trans. on Netw. and Serv. Manag.* 14.3 (Sept. 2017), pp. 498–513. ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2706085. URL: https://doi.org/10.1109/TNSM.2017.2706085.

[22] K. Lee. "Security threats in cloud computing environments". In: *International Journal of Security and its Applications* 6 (Jan. 2012), pp. 25–32.

[23] Raghubir Singh and Sukhpal Singh Gill. "Edge AI: A survey". In: *Internet of Things and Cyber-Physical Systems* 3 (2023), pp. 71–92. ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.02.004. URL: https://www.sciencedirect.com/science/article/pii/S2667345223000196.

[24] Shiqiang Wang et al. "Adaptive Federated Learning in Resource Constrained Edge Computing Systems". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), pp. 1205–1221. DOI: 10.1109/JSAC.2019.2904348.

[25] M. M. Kamruzzaman. "New Opportunities, Challenges, and Applications of Edge-AI for Connected Healthcare in Smart Cities". In: *2021 IEEE Globecom Workshops (GC Wkshps)*. 2021, pp. 1–6. DOI: 10.1109/GCWkshps52748.2021.9682055.

[26] Xiaofei Wang et al. "In-Edge AI: Intelligentizing Mobile Edge Computing, Caching and Communication by Federated Learning". In: *IEEE Network* 33.5 (2019), pp. 156–165. DOI: 10.1109/MNET.2019.1800286.

[27] *What is edge AI?* URL: https://www.redhat.com/en/topics/edge-computing/what-is-edge-ai.

[28] Shuiguang Deng et al. "Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence". In: *IEEE Internet of Things Journal* 7.8 (2020), pp. 7457–7469. DOI: 10.1109/JIOT.2020.2984887.

[29] Mayu Sakurada and Takehisa Yairi. "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. MLSDA'14. Gold Coast, Australia QLD, Australia: Association for Computing Machinery, 2014, pp. 4–11. ISBN: 9781450331593. DOI: 10.1145/2689746.2689747. URL: https://doi.org/10.1145/2689746.2689747.

[30] Adel Abusitta et al. "Deep learning-enabled anomaly detection for IoT systems". In: *Internet of Things* 21 (2023), p. 100656. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2022.100656. URL: https://www.sciencedirect.com/science/article/pii/S2542660522001378.

[31] Aleks Huč, Jakob Šalej, and Mira Trebar. "Analysis of Machine Learning Algorithms for Anomaly Detection on Edge Devices". In: *Sensors* 21.14 (2021). ISSN: 1424-8220.

DOI: 10.3390/s21144946. URL: https://www.mdpi.com/1424-8220/21/14/4946.

[32] Devashree R. Patrikar and Mayur Rajram Parate. *Anomaly Detection using Edge Computing in Video Surveillance System: Review*. 2021. arXiv: 2107.02778 [cs.CV].

[33] Yingnan Ma, Fei Yang, and Anup Basu. "Edge-guided CNN for Denoising Images from Portable Ultrasound Devices". In: *2020 25th International Conference on Pattern Recognition (ICPR)*. Jan. 2021, pp. 6826–6833. DOI: 10.1109/ICPR48806.2021.9412758.

[34] Sachin Mehta et al. "EVRNet: Efficient Video Restoration on Edge Devices". In: *Proceedings of the 29th ACM International Conference on Multimedia*. MM '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 983–992. ISBN: 9781450386517. DOI: 10.1145/3474085.3475477. URL: https://doi.org/10.1145/3474085.3475477.

[35] Shuangshuang Chen and Wei Guo. "Auto-Encoders in Deep Learning—A Review with New Perspectives". In: *Mathematics* 11.8 (2023). ISSN: 2227-7390. DOI: 10.3390/math11081777. URL: https://www.mdpi.com/2227-7390/11/8/1777.

[36] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. "Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 119–139. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/you.

[37] Jin Zheng and Lihui Peng. "An Autoencoder-Based Image Reconstruction for Electrical Capacitance Tomography". In: *IEEE Sensors Journal* 18.13 (2018), pp. 5464–5474. DOI: 10.1109/JSEN.2018.2836337.

[38] Olutosin Ajibola Ademola, Mairo Leier, and Eduard Petlenkov. "Evaluation of Deep Neural Network Compression Methods for Edge Devices Using Weighted Score-Based Ranking Scheme". In: *Sensors* 21.22 (2021). ISSN: 1424-8220. DOI: 10.3390/s21227529. URL: https://www.mdpi.com/1424-8220/21/22/7529.

[39] Laurens van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE". In: *Journal of Machine Learning Research* 9 (Nov. 2008), pp. 2579–2605.

[40] Meghanshu Vashista et al. "Autoencoder: Issues, Challenges and Future Prospect". In: *Recent innovations in mechanical engineering: Select proceedings of ICRITDME 2020*. Springer Singapore Pte. Limited, 2022, pp. 260–265.

[41] Arpan Biswas et al. "Optimizing training trajectories in variational autoencoders via latent Bayesian optimization approach*". In: *Machine Learning: Science and Technology* 4.1 (Feb. 2023), p. 015011. DOI: 10.1088/2632-2153/acb316. URL: https://dx.doi.org/10.1088/2632-2153/acb316.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Autoencoders". In: *Deep learning*. Alanna Maldonado, 2023.

[43] Muhammad Uzair and Noreen Jamil. "Effects of Hidden Layers on the Efficiency of Neural networks". In: *2020 IEEE 23rd International Multitopic Conference (INMIC)*. 2020, pp. 1–6. DOI: 10.1109/INMIC50486.2020.9318195.

[44] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647. eprint: https://www.science.org/doi/pdf/10.1126/science.1127647. URL: https://www.science.org/doi/abs/10.1126/science.1127647.

[45] Prasanna Date et al. "Training Deep Neural Networks with Constrained Learning Parameters". In: *2020 International Conference on Rebooting Computing (ICRC)*. 2020, pp. 107–115. DOI: 10.1109/ICRC2020.2020.00018.

[46] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. "Extreme learning machine: a new learning scheme of feedforward neural networks". In: *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*. Vol. 2. July 2004, 985–990 vol.2. DOI: 10.1109/IJCNN.2004.1380068.

[47] Guang-Bin Huang, Lei Chen, and Chee Siew. "Universal Approximation Using Incremental Constructive Feedforward Networks With Random Hidden Nodes". In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 17 (July 2006), pp. 879–92. DOI: 10.1109/TNN.2006.875977.

[48] Nan-ying Liang et al. "A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks". In: *IEEE Transactions on Neural Networks* 17.6 (Nov. 2006), pp. 1411–1423. ISSN: 1941-0093. DOI: 10.1109/TNN.2006.880583.

[49] Liyanaarachchi Kasun et al. "Representational Learning with ELMs for Big Data". In: *IEEE Intelligent Systems* 28 (Nov. 2013), pp. 31–34.

[50] Rei Ito, Mineto Tsukada, and Hiroki Matsutani. "An On-Device Federated Learning Approach for Cooperative Model Update Between Edge Devices". In: *IEEE Access* 9 (2021), pp. 92986–92998. DOI: 10.1109/access.2021.3093382. URL: https://doi.org/10.1109%2Faccess.2021.3093382.

[51] Junchang Xin et al. "Elastic extreme learning machine for big data classification". In: *Neurocomputing* 149 (2015). Advances in neural networks Advances in Extreme Learning Machines, pp. 464–471. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2013.09.075. URL: https://www.sciencedirect.com/science/article/pii/S0925231214011503.

[52] Mark Labbe. *Energy consumption of AI poses environmental problems*. Aug. 2021. URL: https://www.techtarget.com/searchenterpriseai/feature/Energy-consumption-of-AI-poses-environmental-problems.

[53] Henry Osborne. *Understanding edge computing devices: A comprehensive guide*. May 2023. URL: https://stlpartners.com/articles/edge-computing/edge-computing-devices/.

[54] *Strength of Correlation*. URL: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/strength-of-correlation.html.

[55] *PyTorch Forums*. URL: https://discuss.pytorch.org/.

[56] *torch.cuda Documentation*.

[57] *NVIDIA GeForce RTX 3060 12 GB*. Dec. 2023.

[58] Timo Stöttner. *Why data should be normalized before training a neural network*. May 2019. URL: https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d#:~:text=Normalizing%20the%20data%20generally%20speeds,seems%20to%20be%20strictly%20superior.

[59] Petro Liashchynskyi and Pavlo Liashchynskyi. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*. 2019. arXiv: 1912.06059 [cs.LG].

[60] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. arXiv: 1803.09820 [cs.LG].

[61] Xijie Huang et al. *Efficient Quantization-aware Training with Adaptive Coreset Selection*. 2023. arXiv: 2306.07215 [cs.LG].

A Usage Instructions

A.1 Installation Instructions

First, clone the GitHub repository:

```
git clone https://github.com/nsengupta5/AdaptAE.git
```

Then build the Docker image:

```
docker build -t adaptae .
```

Once the Docker image has been built, run the Docker container. If a NVIDIA GPU is not available, run the Docker container with the following command:

```
docker run -it adaptae
```

If a NVIDIA GPU is available, run the Docker container with the following command to allow the container for CUDA support:

```
docker run --gpus all -it adaptae
```

Once the Docker container is running, activate the Conda environment:

```
conda activate adaptae
```

A.2 Code Instructions

Run AdaptAE with the following command:

```
python3 adaptae/train-adapt-ae.py [-h] --mode MODE --dataset DATASET [--
    ↪ batch-size BATCH_SIZE] [--device DEVICE] [--seq-prop SEQ_PROP] [--
    ↪ generate-imgs] [--save-results] [--phased] [--result-strategy
    ↪ STRATEGY] [--num-images NUM_IMAGES] --task TASK
```

where:

- **-h, -help**: show help message and exit
- **-mode**: the mode of sequential training. AdaptAE uses 'sample' mode by default.
- **-dataset**: the dataset to use. Options are 'mnist', 'fashion-mnist', 'cifar10', 'cifar100', 'super-tiny-imagenet' and 'tiny-imagenet'.
- **-batch-size**: the batch size to use for training. Ignored when using 'sample' mode. Defaults to 10 if not provided
- **-device**: the device to use for training. Options are 'cpu', 'mps' and 'cuda'. Defaults to 'cuda' if not provided
- **-seq-prop**: the proportion of the dataset to use for sequential training. AdaptAE uses 97% of the training data for sequential training by default. However, other proportions can be tested to view the effect on performance metrics. Must be between 0.01 and 0.99 inclusive
- **-generate-imgs**: whether to generate images of the reconstructed data.
- **-save-results**: whether to save the results of the experiment to a CSV file

- **–phased**: Whether to monitor and save phased or total performance results
- **–result-strategy**: the independent variable to vary for the experiment. Options are 'batch-size', 'seq-prop', 'all-hyper', 'latent', 'all'
- **–num-images**: the number of images to generate. Defaults to 5 if not provided
- **–task**: the task to perform. Options are 'reconstruction' and 'anomaly-detection'

Run the autoencoder baseline model with the following command:

```
python3 autoencoder/train-autoencoder.py [-h] --dataset DATASET [--batch
    ↪ -size BATCH_SIZE] [--device DEVICE] [--generate-imgs] [--save-
    ↪ results] [--result-strategy STRATEGY] [--num-images NUM_IMAGES] --
    ↪ task TASK
```

where:

- **-h, –help**: show help message and exit
- **–dataset**: the dataset to use. Options are 'mnist', 'fashion-mnist', 'cifar10', 'cifar100', 'super-tiny-imagenet' and 'tiny-imagenet'.
- **–batch-size**: the batch size to use for training. Defaults to 64 if not provided
- **–device**: the device to use for training. Options are 'cpu', 'mps' and 'cuda'. Defaults to 'cuda' if not provided
- **–generate-imgs**: whether to generate images of the reconstructed data.
- **–save-results**: whether to save the results of the experiment to a CSV file
- **–result-strategy**: the independent variable to vary for the experiment. Options are 'batch-size', 'num-epochs', 'all-hyper', 'latent', 'all'
- **–num-images**: the number of images to generate. Defaults to 5 if not provided
- **–task**: the task to perform. Options are 'reconstruction' and 'anomaly-detection'

Run the ELM-AE baseline model with the following command:

```
python3 elmae/train-elm-ae.py [-h] --dataset DATASET [--device DEVICE]
    ↪ [--generate-imgs] [--save-results] [--result-strategy STRATEGY]
    ↪ [--num-images NUM_IMAGES] --task TASK
```

where:

- **-h, –help**: show help message and exit
- **–dataset**: the dataset to use. Options are 'mnist', 'fashion-mnist', 'cifar10', 'cifar100', 'super-tiny-imagenet' and 'tiny-imagenet'.
- **–device**: the device to use for training. Options are 'cpu', 'mps' and 'cuda'. Defaults to 'cuda' if not provided
- **–generate-imgs**: whether to generate images of the reconstructed data.

- –**save-results**: whether to save the results of the experiment to a CSV file
- –**result-strategy**: the independent variable to vary for the experiment. Options are 'latent', 'all'
- –**num-images**: the number of images to generate. Defaults to 5 if not provided
- –**task**: the task to perform. Options are 'reconstruction' and 'anomaly-detection'

B Ethical Approval Form

The preliminary self assessment form was filled out and submitted on 18th September 2023. Because this project did not require approval from the University of St Andrew's Teaching and Research Ethics Committee, it proceeded under the standard ethical guidelines for research projects.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
☑ **Undergraduate Project**

Title of project

Slicing Autoencoders for Efficient Image Processing

Name of researcher(s)

Nikhil Sengupta

Name of supervisor (for student research)

Blesson Varghese

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☑ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Print Name

Nikhil Sengupta

Date

September 15, 2023

Signature Lead Researcher or Supervisor

Print Name

Blesson Varghese

Date

September 15, 2023

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

The School Ethics Committee will be responsible for monitoring assessments.

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondary-data/ and https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ **NO** ☑

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ **NO** ☑

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☑

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☑

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☑

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/