TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF PSEUDO/SOURCE CODE

INTISARI

PERBANDINGAN ALGORITMA PEMBELAJARAN MESIN INKREMENTAL PADA DETEKSI MANUSIA (IUP)

M. G. Maulana
18/423112/PA/18195

Kondisi deteksi manusia saat ini diselidiki dalam penelitian ini, termasuk tantangan dalam mendeteksi manusia, berbagai metode yang saat ini tersedia, dan beragam aplikasinya. Metode yang diteliti dalam penelitian ini meliputi teknik pembelajaran mesin, khususnya mesin vektor pendukung dan jaringan saraf, serta varian tambahannya. Aplikasi utama yang dibahas dalam penelitian ini adalah pendeteksian manusia dalam lingkungan pengawasan video, meskipun teknik-teknik ini juga dapat diterapkan pada aplikasi lain.

Fokus dari penelitian ini adalah untuk membandingkan tiga metode pembelajaran mesin inkremental - Incremental Support Vector Machines, Unified Classifier via Rebalancing, dan Dynamically Expandable Representation - berdasarkan akurasi, waktu berjalan, dan konsumsi memori.

Hasil dari penelitian ini menunjukkan bahwa DER memiliki performa terbaik dalam hal akurasi dan penggunaan memori puncak, sementara UCIR memiliki waktu berjalan terpendek namun memiliki akurasi yang lebih rendah dan penggunaan memori yang lebih tinggi dibandingkan dengan ISVM

Kata kunci: Pembelajaran Mesin Inkremental, Deteksi Manusia, Mesin Vektor Pendukung Tambahan, Pengklasifikasi Terpadu melalui Penyeimbangan Ulang, Representasi yang Dapat Diperluas Secara Dinamis

ABSTRACT

COMPARISON OF INCREMENTAL MACHINE LEARNING ALGORITHM ON HUMAN DETECTION (IUP)

M. G. Maulana
18/423112/PA/18195

The current state of human detection is investigated in this study, including the challenges of detecting humans, the various methods that are currently available, and their diverse applications. The methods examined in this study include machine learning techniques, particularly support vector machines and neural networks, as well as their incremental variants. The primary application discussed in this study is the detection of humans in a video surveillance setting, although it is noted that these techniques can also be applied to other applications.

The focus of this research is to compare three incremental machine learning methods - Incremental Support Vector Machines, Unified Classifier via Rebalancing, and Dynamically Expandable Representation - based on their accuracy, running time, and memory consumption.

The results of this study show that DER performed the best in terms of accuracy and peak memory usage, while UCIR had the shortest running time but lower accuracy and higher memory usage compared to ISVM.

Keywords: Incremental Machine Learning, Human Detection, Incremental Support Vector Machines, Unified Classifier via Rebalancing, Dynamically Expandable Representation

CHAPTER I
INTRODUCTION

1.1 Research Background

To date, multi-billion-dollar businesses apply several data processing strategies in order to gain valuable insight, predict certain trends, give better customer experience, etc. The amount of data collected by these companies increases by the day for example: 3.5 billion search queries are gathered by Google, 1 billion videos are being watched on YouTube, and 500 tweets sent by Twitter users a day (Real Time Statistics Project, 2014).

Data collection is not only used in business firms but also in fields of study, observation and entertainment such as space observatories, environmental monitoring facilities, competitive video game matchmaking, grading systems, customized learning programs, etc. The data collected are then utilized to be used to improve their models which are usually trained by machine learning methods to accurately detect meteors, abnormal plate movements, or match players according to their skill level, etc. These methods are also applied to video surveillance systems to detect humans.

Accurately detecting humans in a visual surveillance system is critical for a variety of applications such as abnormal event detection, human gait characterisation, congestion analysis, person identification, gender classification, and fall detection for the elderly (Paul et al., 2013), trespassing, whether at a country boundary or in a residential area. It is a severe threat to an individual's life, property, and security (Abhinava & Majumdar, 2017). Other practical applications would include: abnormal state detection in surveillance, warning of danger in advanced driver-assistance systems (ADAS), human interaction games, and interactive robot services (Hu et al., 2022).

Before human detection was done by machines it was done by a person whether on-site or behind a monitor watching cameras set up throughout the location. Other ways include laser sensors used in elevators to detect whether an

object was in the way, ultrasonic signals emitted from human footsteps, measuring the human body motion Doppler signature (Sabatier et al., 2007).

Nowadays most human detection is done with machine learning algorithms that take in the video feed from the cameras and the algorithms mainly do two processes: object detection and object classification. For object detection the target needs to be identified, Background subtraction is a common object detection approach that seeks to find moving objects by comparing the current frame to a background frame on a pixel-by-pixel or block-by-block basis, which the algorithm will class them as candidates to be classified human or non-human. The algorithm then classifies the candidates according to the features the candidates have for example: shape, appearance, texture, motion and/or a combination of features (Nguyen et al., 2016).

To enable the algorithm to classify what is human and what is not it has to train itself by feeding it an image dataset that contains both humans and non-humans which are usually tagged. The main difference between machine learning and incremental machine learning algorithm would be the ability for the incremental machine learning algorithm to continuously learn after the initial dataset was been learnt allowing it to adapt to individual customers and environments (Losing et al., 2018).

The aim of the study is to find the more efficient and accurate algorithm among three algorithms. To achieve this, the three algorithms would be trained and tested on the same image dataset which then will be validated on another dataset. The algorithms to be used are Unified Classifier Incrementally via Rebalancing (UCIR), Dynamically Expandable Representation (DER) and Incremental Support Vector Machines (Incremental SVM).

1.2 Research Problem

As technology evolves storage increases whilst hardware size scales down, there may be a need for these machines to train themselves within a specific situation such as in smart homes where every individual is different to one another,

traffic systems or security cameras in newly developed areas where there may not be an abundant amount of data to train from. These systems can benefit from incremental learning methods as it will be able to train on the fly according to the specific circumstances. Due to this, it is important to know which of the three algorithms would be suitable for this scenario. Furthermore, as there are few researches done in regards to the comparison of human detection using incremental learning methods it is then more imperative that a comparison is done.

1.3 Research Objectives

The objectives of this research are:

1. To use available ISVM, DER, UCIR and algorithms.
2. To train the models using human present datasets.
3. To test and compare the algorithms based on accuracy, memory consumption and running time.

1.4 Research Scope

Three algorithms will be designed, trained and tested, to be used to detect humans/pedestrians in a public area. The algorithm should be able to initially detect pedestrians after its first batch of training, once more data is added the algorithm must be able to train using the data and still be able to detect pedestrians.

1.5 Research Benefits

Currently there are few papers that compare different incremental learning methods on human detection and it is important to know if the method is suitable afterwards, the method that would work best in the aforementioned scenario would be found out. Additionally, this work can be expanded on to include additional algorithms and datasets.

1.6 Report Organization

This report consists of four parts, namely:

1. Chapter 1 describes the project introduction, problem statement, objectives, the scope of the project, and the significance of the project.
2. Chapter 2 includes a description of the study literature i.e. incremental machine learning, AI and human-detection and comparative studies of existing implementations.
3. Chapter 3 explains the theoretical basis of the research.
4. Chapter 4 explains the methodology of the research.
5. Chapter 5 depicts and explains the implementation of the methodology done in chapter 4.
6. Chapter 6 shows the results of the implementation done in Chapter 5 and the performance of the algorithms.
7. Chapter 7 concludes the report and a brief afterword of future works is given.

CHAPTER II
LITERATURE REVIEW

Generally, detection refers to the task of identifying whether or not an instance of a given object type exists in the scene. In this case the given object to be identified in the scene would be a person, but due to the nonrigid, articulate character of the human body, the work of human detection is very difficult. Human detection research also focuses on efficient learning algorithms and classification strategies that enable high detection accuracy while minimising false positives. Human detection algorithms often try to provide simply the location and scale of the humans in the scene. Recently, there has been an increased emphasis on retrieving the silhouette shape of each observed individual. Such algorithms often require training photos with the silhouettes explicitly marked in each image, however some approaches require either extremely minimal or no manual tagging of the training dataset. Because people have movable limbs, can appear in various stances, and may carry things (e.g., briefcase, backpack), obtaining the real silhouette form might give useful indications for gait and activity identification systems (Davis et al., 2009).

2.1 Study on Machine Learning on Human Detection

Robson et al. (2009) provided a human detection algorithm that used a wider descriptor set that included edge-based features, texture measurements, and colour information, and the results improved significantly. The augmentation of these features results in a very high dimensional space where previous machine learning approaches fail. The properties of the data made it a good situation for using Partial Least Squares (PLS) to produce a considerably lower dimensional subspace where simple and efficient classifiers were applied. They evaluated their methodology on a variety of datasets and found that it has high generalisation capabilities and outperforms state-of-the-art algorithms that incorporate extra cues at the time.

Kachouane et al. (2012) presented a fast humans detection method built with their own database that use histograms of oriented gradient (HOG) in conjunction

with the SVM classifier. Several experiments have been run to determine the best settings. The global detection rate is seen to be the best between the threshold values of 1.01 and 1.06, whereas false positive rates decline after a threshold of 1.04, based on detector application findings on two separate datasets. Processing time is enough for embedded applications; however, detector combinations can improve it.

Ivašić-Kos et al. (2019) attempts to recognise people in videos taken throughout the winter in various weather conditions at night and at various distances from the camera, ranging from 30m to 215m and wanted to see how popular deep learning approaches for object identification and recognition in RGB photos, such as the YOLO detector, performed in thermal images. The individuals were either sprinting or walking bent and attempting to keep out of sight. Even though thermal photos differ significantly from RGB images in appearance, they have assumed that the characteristics learnt by YOLO on a large COCO dataset of RGB images for the class Person will still give a suitable baseline for thermal images. For human detection in thermal images, the first YOLO model (bYOLO) obtained an average precision (AP) of just 7% which is substantially poorer than the results obtained by YOLO on visible spectrum photos, where the results vary depending on the circumstance. Unfortunately, this is owing to the disparity between visual and thermal images. As a result, they also trained the bYOLO model on thermal images from their custom dataset, and after training the model, tYOLO achieved significantly better results for person detection in different weather conditions and at different distances from the camera, with an AP of approximately 30%. The experiment demonstrated that further training on thermal datasets can greatly enhance the performance of the YOLO model on thermal imaging.

Ansari & Singh (2021) proposes a deep learning-based human detecting approaches for monitoring social separation in real-time. These strategies were created using a deep convoluted network and the sliding window notion as a region suggestion. They are also used in conjunction with the social distancing algorithm to assess people's distancing criteria. This analysed distance parameters determine if two persons are adhering to social distancing rules. Extensive trials were carried

out using CNN-based object detectors. Experiments show that CNN-based object detection algorithms outperform others in terms of accuracy. When dealing with real-time video sequences, it occasionally creates false positives. In the future, many current object detectors such as RCNN, Faster RCNN, SSD, RFCN, YOLO, and others may be used with the self-created dataset to improve detection accuracy and minimise false positives. Furthermore, a single viewpoint derived from a single camera cannot adequately depict the outcome. As a consequence, in the future, the suggested algorithm might be configured for varied viewpoints through several cameras to produce more accurate results.

2.2 Study on Incremental Machine Learning on Human Detection

Joshi and Fatih (2010) provides two ways to adapting generic training data to generate scene-specific detectors, one totally autonomous and one with minimum user control. The solutions mentioned solve the critical issue of rapid deployment in diverse places without needing costly data collecting activities at the spot. Using incremental learning, classifiers may mix the benefits of accessible generic and scene-specific data. Their strategy works by actively picking fresh cases for training and discarding old uninformative ones. The elimination of training examples allows them to keep consistent training sizes, allowing for efficient training on a set memory budget. They use a sliding window 75 pixels by 50 pixels wide with horizontal and vertical overlap of 50 and 30 pixels for each frame of test footage. For each window, HOG features are extracted, and the resulting vector is fed through the trained Support Vector Machine (SVM) classifier.

Xia et al. (2013) adopts the HOG based pedestrian detection as their baseline platform where the framework has two advantages. On one hand, if the data sets are collected continuingly, the system could rapidly and effectively train detectors using new data sets. On the other hand, if a data set is so large that memory cannot meet the demand, they could divide it into several parts and use them to train a detector. They propose a new incremental learning algorithm called Converged Passive-Aggressive algorithm (CPA) for pedestrian detection. Their results show on average, CPA has above 1.5% lotheyr miss rate than Passive-Aggressive and

above 0.8% lotheyr miss rate than Pegasos. And that on another dataset miss rate declines continually, which demonstrates that their framework has good ability for training better detectors.

Hanyu and Zhao (2017)'s explores the incremental strategy for enhancing the accuracy of the SVM-based human detector using new photos in the next section. They obtain a number of sub-images that do not include any person throughout the process of human detection with the human detector. These sub-images generate false positive errors and provide "near-miss" data that may be used to improve the existing detector. They just want to look at the trend of the incremental training procedure in this experiment. As a result, they created only 100 photos with only one person in each image and investigated how the detection rate changes when 24000 positive human images and 25000 background images were prepared. They believe the proposed approach can increase the accuracy of the generated SVM based on the experimental findings. They are able to generate a more accurate SVM utilising about the same amount of support vectors using the proposed strategy. Nonetheless, there are occasions where the backdrop image is identified as a human image. Because the majority of the mistakes happened in close proximity to humans, it appears that the results can be utilised for human detection with some suitable post-processing.

Pop et al. (2021) comprehensively illustrates alternative cross-modality learning methodologies of four systems based on Convolutional Neural Networks for pedestrian recognition: (1) specific cross-modality learning (PaCML); (2) separate cross-modality learning (SeCML); (3) correlated cross-modality learning (CoCML); and (4) incremental cross-modality learning (InCML). The particular cross-modality learning approach might be enhanced to include an automated annotation mechanism for new modality images. When there are not enough annotated images in each modality to enhance classification performance, incremental cross-modality learning might be applied. The efficacy of those strategies has been assessed using various performance indicators and statistical factors (Confidence Intervals, Correlation Coefficients, Structural Similarity

Index). The incremental cross-modality learning model is superior to both the independent and correlated cross-modality learning models. It also outperforms traditional learning of uni-modal CNNs via late-fusion on the Daimler data set in terms of classification performance. They consider the incremental technique to be the most promising cross-modality learning model. This cross-modality learning strategy is more adaptable than the others they studied since it can be employed with varied learning parameters tailored to each image modality. They presented a novel CNN architecture dubbed LeNet+ to increase its performance, and it outperforms the state-of-the-art pedestrian classifier for both non-occluded and partially-occluded pedestrian Daimler datasets.

2.3 Study on Comparisons of IML

Ammar et al. (2011) proposes a detection algorithm made up of a number of functions. The first phase is offline processing, which includes the creation of a person database, the HOG and SIFT extraction descriptors, and the training step utilising Gentle AdaBoost learning. The treatment of the retrieved video is done online; the sequence is separated into images, and a classifier is used to determine if the search window is person or nonperson. The tracking technique is then used to follow the moving individual using an incremental Principal Component Analysis (PCA). They state that the database they employed is based on the MIT Pedestrian Image Dataset and self-extracted human photos. As a consequence, while they cannot compare our results (recall or accuracy) to those of the literature, they do indicate the efficacy of the combination of SIFT and HOG descriptors. They also discovered that this method may be used to detect not only people but also objects in the area. They also assessed the performance of each AdaBoost method, calculating and comparing errors of Gentle, Modest, and Real AdaBoost to a set of images shown in a series of video sequences. Gentle AdaBoost appears to outperform the other methods in their situation.

Losing et al. (2018) investigates the most prevalent incremental learning methods at the time on a variety of datasets, both stationary and non-stationary, the number of IML methods being 8. It gives a quick overview of the key characteristics

of the different collection of techniques under consideration, facilitating the selection of an acceptable algorithm for a specific job. In terms of outcomes, SVMs often provide the best accuracy at the price of the most complicated model. Compared to the ISVM, the LASVM's approximate nature decreases training time and allows it to perform for bigger data sets. The ORF performs somewhat lower but has a far shorter training and running time. However, its model, like those of both SVMs, expands linearly with the amount of data and cannot be easily restricted. As a result, in contrast to the remaining approaches, which have either a constant or readily boundable complexity, these algorithms are not suitable for learning in infinite streams. The ILVQ is a more accurate and sparser alternative to SVMs. LPPCART is extremely versatile because the basis classifier may be chosen arbitrarily, however it may suffer due to its poor knowledge integration across chunks. Because of their compact representation and sub-linear run-time that is independent of the number of dimensions, tree-based models are particularly well-suited for high-dimensional data. However, the compact representation slows learning, therefore instance-based models converge faster and are better suited for learning tasks with only a few instances. Linear SGD and GNB's sparse models make them especially plausible alternatives for large-scale learning in high dimensional space on the one hand, but not complicated enough for low dimensional tasks on the other. GNB and tree-based approaches are the most practical, needing no or very minimal HPO. SVMs and ILVQs, on the other hand, demand the most precise settings.

Luo et al. (2020) evaluates incremental learning approaches which include 20 papers that use different methods that learn from various datasets. They discussed the fundamental ideas and major obstacles of incremental learning, as well as three incremental learning techniques for mitigating catastrophic forgetting: architecture strategy, regularisation strategy, rehearsal and pseudo-rehearsal strategy. They examined the present state of incremental learning research and forecasted future incremental learning research from the perspectives of application and theory through a discussion and comparison of relevant incremental learning approaches. Despite recent advancements, the realisation of flexible and durable

incremental learning that can adapt to many complicated settings remains a long way off. They discovered that the technique for coping with catastrophic forgetfulness has reached a roadblock via study. Researchers are more inclined to research incremental learning utilising a mix of current tactics, which is typically preferable than employing a single approach alone. Many existing incremental learning strategies are constrained by a lack of flexibility and practicality. The majority of approaches are task-based, but the real data flow is far more intricate. As a result, the progressive learning of task-free design should be given more consideration. They presume that unsupervised learning is one approach that may be taken. Due to the iterative update of parameters with the introduction of new data is a primary internal source of catastrophic forgetting, non-iterative training may be possible to overcome catastrophic forgetting at a lower level, which is also a promising avenue for future research.

Table 2.1 shows a rough comparison on the datasets, methods and results of papers mentioned in Sections 2.1 and 2.2.

Table 2.1 Comparison of Machine Learning on Human Detection

No	Reference	Dataset	Method	Result
1	(Robson et al., 2009)	INRIA Person Dataset, DaimlerChrysler Pedestrian Dataset, ETHZ Dataset.	Partial Least Squares	Demonstrated its good generalization capabilities and shown it to outperform state-of-the-art methods that use additional cues at the time.

No	Reference	Dataset	Method	Result
2	(Kachouane et al., 2012)	INRIA pedestrian database	SVM Classifier	The global detection rate achieved is 86%
3	(Ivašić-Kos et al., 2019)	COCO images Dataset, Custom-made dataset	YOLO	The Average Precision score achieved with tYOLO: 29%.
4	(Ansari & Singh, 2021)	INRIA Person Dataset (2018)	CNN-Based	Achieves an accuracy of 98.5%
5	(Joshi & Fatih, 2010)	INRIA pedestrian data, CAVIAR dataset	SVM Classifier with active learning and forgetting	As the number of background frames used increases, the number of false positives goes down.
6	(Xia et al., 2013)	INRIA and the NICTA pedestrian dataset	SVM and Converged Passive-Aggressive algorithm	CPA has above 1.1% lower miss rate than Passive-Aggressive and above 0.8% lower miss rate than Pegasos. With CPA having a miss rate of 14.21%

No	Reference	Dataset	Method	Result
7	(Hanyu & Zhao, 2017)	Custom-made Dataset	SVM where they incrementally added only needed data.	Achieves an accuracy rate of 87.6%
8	(Pop et al., 2021)	Daimler stereo vision dataset	Particular Cross-Modality	The highest accuracy is achieved by the Improved Incremental Cross-Modality learning method with an accuracy of 88.7%
			Separate Cross-Modality	
			Correlated Cross-Modality Learning	
			Incremental Cross-Modality	
			Improved Incremental Cross-Modality	

CHAPTER III
THEORETICAL BASIS

3.1 Human Detection

Human beings are capable of detecting humans by only using subtle clues. Automated algorithms are instead still far from matching or even just approaching this ability. This is partly due to the intrinsic difficulties associated with the human body and the environment in which it is located, the non-rigid nature of the human body produces numerous possible poses. It is also challenging to model simultaneously view and size variations arisen from the change of the position and direction of the camera. Unlike other types of objects, humans can be clothed with varying colours and texture, which adds another dimension of complexity. In addition, the environment can make humans less visually noticeable (Nguyen et al., 2016)

Human detection is a task of computer vision systems to locate all instances of human beings present in an image, and it has been most widely accomplished by searching all locations in the image, at all possible scales, and comparing a small area at each location with known templates or patterns of people (Davis et al., 2009)

An object recognition system must include the following components:

• Model database/dataset,

• Feature detector,

• Hypothesizer,

• Hypothesis verifier.

The model database includes all of the models that the system is aware of. The information in the model database is determined by the method used for recognition. It might range from a qualitative or functional description to detailed geometric surface data. Object models are often abstract feature vectors, as mentioned later in this section (Jain et al., 1995)

A feature is an object attribute that is useful in characterising and identifying the item in relation to other objects. Size, colour, and form are all common characteristics. The feature detector applies operators on images and detects feature locations that aid in the formation of object hypotheses. A system's characteristics are determined by the sorts of objects to be identified and the arrangement of the model database. Using the detected features in the image, the hypothesizer assigns likelihoods to objects present in the scene. This step is used to reduce the search space for the recognizer using certain features. To simplify the rejection of implausible object candidates from prospective consideration, the model-base is arranged using some form of indexing technique.

The verifier then utilises object models to validate the hypotheses and refines object likelihood. Based on all of the information, the algorithm then chooses the item with the highest chance as the correct object. Whether explicitly or indirectly, all object recognition systems incorporate models and feature detectors based on these object models. The importance of the hypothesis formation and verification components varies between approaches to object recognition. Some systems just develop hypotheses and then choose the item with the highest likelihood as the correct object.

Pattern categorization methods are an excellent illustration of this strategy. Many artificial intelligence systems, on the other hand, depend less on hypothesis creation and focus more on verification. For the processes outlined above, an object recognition system must pick acceptable tools and approaches.

Many types of features are used for object recognition. Most features are based on either regions or boundaries in an image. It is assumed that a region or a closed boundary corresponds to an entity that is either an object or a part of an object.

Object recognition is the set of procedures that must be taken after detecting acceptable characteristics. Not all object identification algorithms need rigorous

hypothesis generation and verification procedures. Most recognition systems have developed to incorporate differing proportions of these two processes.

The fundamental notion behind categorization is to distinguish items based on their attributes. This category includes pattern recognition algorithms, this category also includes neural net-based techniques. All approaches in this class assume that N features in images have been discovered and normalised such that they may be represented in the same metric space (Jain et al., 1995)

3.2 Incremental Machine Learning

Incremental learning is a machine learning paradigm where the learning process takes place whenever new example(s) emerge and adjusts what has been learned according to the new example(s). The most prominent difference of incremental learning from traditional machine learning is that it does not assume the availability of a sufficient training set before the learning process, but the training examples appear over time (Geng & Smith-Miles, 2009).

Incremental/on-line learning algorithms continuously integrate new information into the current model, which is relevant more now than ever due to the amount of data that needs to be processed, and so older methods that access all data at once to create new models from scratch are not preferable when in a time constraint (Losing et al., 2018).

The on-line learning scenario combines the training and testing stages, rather than learning from a training set and then testing on a test set. The obvious objective under this assumption is to learn a hypothesis with a modest anticipated loss or generalisation error. On-line learning, on the other hand, has no distributional assumptions and so has no concept of generalisation. Instead, the effectiveness of online learning algorithms is assessed using a mistake model and the concept of regret. Theoretical investigations in this paradigm are based on a worst-case or adversarial assumption to generate assurances. This allows it to process current data sets with several million or billion points, they are often much more efficient in both time and space, as well as more practical than batch

algorithms. They are also usually simple to implement. Furthermore, online algorithms make no distributional assumptions; their analysis is based on an adversarial scenario. As a result, they may be used in a range of situations where the sample points are not drawn independent and identically distributed (i.i.d) or in accordance with a predetermined distribution (Mohri et al., 2018)

T rounds are used in the general online context. At the tth round, the algorithm receives an instance $x_t \in X$ and makes a prediction $\hat{y}_t \in Y$. It then receives the true label $y_t \in Y$ and incurs a loss $L(\hat{y}_t, y_t)$, where $L: Y \times Y \rightarrow R_+$ is a loss function. In general, the algorithm's prediction domain could be $Y' \neq Y$, with the loss function defined over $Y' \times Y$. For classification issues, $Y = \{0, 1\}$ and $L(y, y') = |y' - y|$ is frequently used, but for regression, $Y \subseteq R$ and $L(y, y') = (y' - y)^2$ is commonly used. In the online scenario, the goal is to minimise the cumulative loss: $\sum_{t=1}^{T} L(\hat{y}_t, y_t)$ over T rounds (Mohri et al., 2018)

3.3 Incremental Support Vector Machine

To incrementally train an SVM the algorithm recursively establishes the solutions one point at a time, it does this by training the dataset, the support vector is then retained to be used to train the new training set with the new data. The Kuhn Tucker (KT) condition will still have to be maintained, which is done by partitioning the training data into 3 categories, changing the support vector coefficients incrementally - which keeps the training data in equilibrium- in order to add the new data adiabatically (Cauwenberghs & Poggio, 2001).

In order to make the SVM learning algorithm incremental, the dataset is partitioned in batches that fit into memory. Then, at each incremental step, the representation of the data seen so far is given by the set of support vectors describing the learned decision boundary (along with the corresponding weights). Such support vectors are incorporated with the new incoming batch of data to provide the training data for the next step, Since the design of SVMs allows the number of support vectors to be small compared to the total number of training examples, this scheme should provide a compact representation of the data set. It is

reasonable to expect that the model incrementally built won't be too far from the model built with the complete data set at once (batch mode). This is because, at each incremental step, the SVM remembers the essential class boundary information regarding the seen data, and this information contributes properly to generate the classifier at the successive iteration (Gunopulos & Domeniconi, 2001).

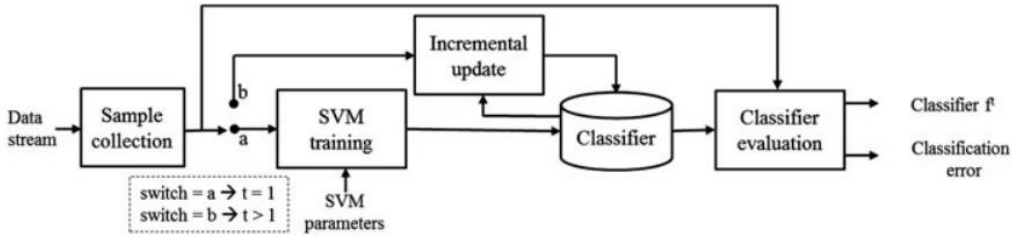

Figure 3.1: Incremental SVM learning procedure (Lawal, 2019)

When a fresh batch of samples $F^{t+1} = \{(\boldsymbol{x}_1^{t+1}, y_1^{t+1}), \dots\}$ is acquired at time $t + 1$, F^t may be updated to F^{t+1} utilising incremental learning without having to recompute the classifier from start as shown in Figure 1. Depending on the application, additional examples might appear in real time. While updating the classifier, the online techniques process the data stream one sample at a time and verify that the KKT criteria are maintained on all previously viewed samples (Lawal, 2019).

In an ISVM designed for 2-class problem is as follows: When a new data stream $(\boldsymbol{x}_1^{t+1}, y_1^{t+1})$ is received, the algorithm computes the value of $y_1^{t+1} f^t(\boldsymbol{x}_1^{t+1})$ to determine if the new sample has the potential to enhance the classifier. If $y_1^{t+1} f^t(\boldsymbol{x}_1^{t+1}) \leq 1$, it indicates that the samples are useful. Thus, for each new sample, the method initialises a coefficient α_1^{t+1} to 0 and then perturbs the SVM by progressively raising the value of the coefficient until the best SVM solution is reached (Cauwenberghs & Poggio, 2001).

The other $\alpha_i^t \forall i \in S_m^t$ and S_b^t are likewise altered during the SVM perturbation in order to maintain the KKT optimality requirements for all previously viewed samples. The incremental SVM learning is also reversible, which means that the patterns in the SVM solution may be unlearned one by one.

To unlearn a pattern, the relevant sample's coefficient is gradually decremented to zero, while the coefficients of the remaining samples are readjusted to preserve the KKT conditions (Lawal, 2019). Algorithm 1 describes the main steps of the algorithm.

Algorithm 1: The ISVM algorithm (Cauwenberghs & Poggio, 2001)

Input: $(\boldsymbol{x}_1^{t+1}, y_1^{t+1})$: new sample at time $t + 1$, f^t : decision function at t

Output: f^{t+1}: updated decision function at $t + 1$

Definitions: α_i^t : coefficient of the ith sample at t, S_m^t: margin support vector set at t, S_b^t: bounded support vector set at t, Q: kernel matrix

1: Begin:
2: Compute $z = y_1^{t+1} f^t(\boldsymbol{x}_1^{t+1})$,
3: If $z > 1$, then, $f^{t+1} \leftarrow f^t$, and go to step 10
4: Else,
5: Initialize $\alpha_1^{t+1} \leftarrow 0$, for $\boldsymbol{x}_1^{t+1}$
6: Compute Q_{i1} for $\forall x_i^t \in S_m^t$
7: Increment α_1^{t+1} to its largest value while maintaining the KKT optimality conditions on all previously seen samples
8: Check if one of the following conditions occurs:
   I. If $y_1^{t+1} f^t(\boldsymbol{x}_1^{t+1}) = 1$, then $S_m^t : \leftarrow \boldsymbol{x}_1^{t+1}$
   II. Else if $\alpha_1^{t+1} = C$, then $S_b^t \leftarrow \boldsymbol{x}_1^{t+1}$
   III. Else if any $\boldsymbol{x}_i^t \in S_m^t$ become part of S_b^t, due to the change in their corresponding α_i^t, then update S_m^t and S_b^taccordingly
9: $f^{t+1} \leftarrow f^t$, $S_m^{t+1} \leftarrow S_m^t$ and $S_b^{t+1} \leftarrow S_b^t$
10: End

3.4 Unified Classifier Incrementally via Rebalancing

UCIR fits under the distillation-based category. However, it varies from earlier efforts in one important way: rather than mixing different objective terms to balance old and new classes, the negative impacts of imbalance is extensively studied and a systematic method that addresses the problem from several viewpoints is suggested.

It includes three components in particular to mitigate the negative effects of the imbalance: (1) cosine normalisation, which enforces balanced magnitudes across all classes, including both old and new classes; (2) less-forget constraint,

which aims to preserve the geometric configuration of old classes; and (3) inter-class separation, which encourages a large margin to separate the old and new classes. The framework can more effectively conserve information obtained in earlier phases and decrease misunderstandings between old and new classes by rebalancing the training process with these strategies. (Hou et al., 2019)

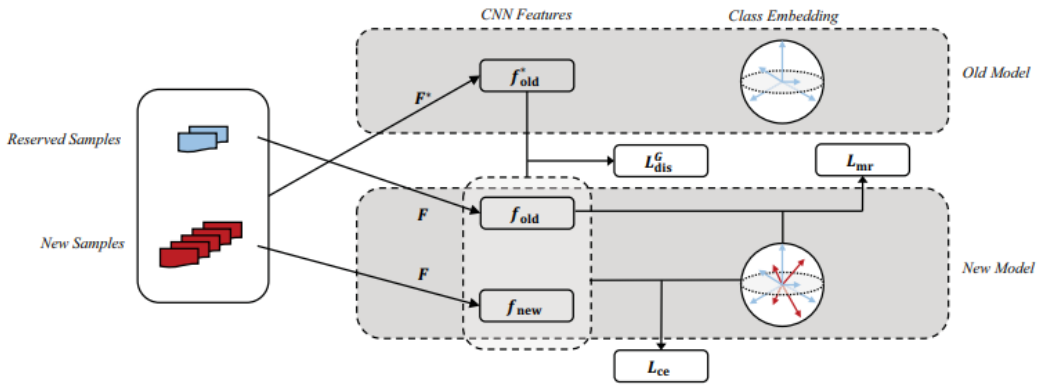

Figure 3.2: Unified Classifier Incrementally via Rebalancing (Hou et al., 2019)

Figure 1 illustrates the approach for the multi-class incremental learning, where f denotes the feature extractor and L denotes Loss.

To develop a unified classifier for both old classes C_O and new classes C_n, based on a new dataset $\mathcal{X} = \mathcal{X}_\mathrm{n} \cup \mathcal{X}'_\mathrm{o} \cdot \mathcal{X}_\mathrm{n}$, given a model trained on an old dataset X_O. X_n is a big dataset that solely includes the new classes C_n, whereas $\mathcal{X}'_\mathrm{o} \subset \mathcal{X}_\mathrm{o}$ only includes a small portion of previous samples. The primary difficulty is determining how to use the significantly unbalanced X and the original model to improve performance across all classes while avoiding catastrophic forgetting (Hou et al., 2019).

In a typical CNN, the predicted probability of a sample x is computed as follows:

$$p_i(x) = \frac{\exp(\theta_i^T f(x) + b_i)}{\Sigma_j \exp\,(\theta_j^T f(x) + b_j)} \tag{1}$$

where f is the feature extractor, θ and b are the weights and the bias vectors in the last layer. Due to the class imbalance, the magnitudes of both the embeddings and the biases for the new classes are significantly higher than those for the old classes. This results in the bias in the predictions that favour new classes. To address this issue, it is proposed to use cosine normalization in the last layer, as:

$$p_i(\mathrm{x}) = \frac{\exp\left(\eta\left(\bar{\theta}_i, \bar{f}(x)\right\rangle\right)}{\Sigma_{\mathrm{j}}\exp\left(\eta\left(\bar{\theta}_j, \bar{f}(x)\right\rangle\right)} \tag{2}$$

the learnable scalar η is introduced to regulate the peakiness of the SoftMax distribution because the range $\langle \bar{v}_1, \bar{v}_2 \rangle$ is constrained to [−1, 1] where $\bar{v} = {}^{v}/_{\|v\|_2}$ denotes the l2-normalized vector, and $\langle \bar{v}_1, \bar{v}_2 \rangle = \bar{v}_1^T \bar{v}_2$ measures the cosine similarity between two normalized vectors. Cosine normalisation is used to effectively remove the bias induced by the large magnitude difference (Hou et al., 2019).

Due to the scalar η in the original model and that in the current network differ, it is fair to imitate the scores before SoftMax rather than the probabilities after SoftMax for the distillation loss. It is also worth noting that, thanks to cosine normalisation, the scores before to SoftMax are all in the same range (i.e. [−1, 1]) and hence comparable. Formally, the distillation loss is updated as:

$$L_{dis}^{c}(x) = -\sum \left\| \left\langle \bar{\theta}_i, \bar{f}(x) \right\rangle - \left\langle \bar{\theta}_i^* \bar{f}^*(x) \right\rangle \right\| \tag{3}$$

where f^* and θ^* are the original model's feature extractors and class embeddings, and $|C_o|$ is the number of old classes. The normalised features and class embeddings are geometrically located on a high-dimensional sphere. L_{dis}^{c} promotes the geometric structures, as represented by the angles between features and the former class embeddings, to be roughly retained in the current network.

A model that has been updated to fresh data has a tendency to forget what it has previously learnt. As a result, one of the practical obstacles for incremental learning is minimising forgetting past information. A lessforget restriction is introduced via a new loss L_{dis}^{G}. L_{dis}^{G}, in particular, focuses on the local geometric structures, i.e., the angles between the normalised features and the old class

embeddings. This limitation does not prohibit the embeddings and features from being rotated completely. Fix the old class embeddings and computing a fresh distillation loss on the features to impose a stricter constraint on past knowledges:

$$L_{dis}^{G}(x) = 1 - \langle \bar{f}^{*}(x), \overline{f(x)} \rangle \tag{4}$$

where $\bar{f}^{*}(x)$ and $\bar{f}(x)$ are the normalised features recovered by the original and current models, respectively. L_{dis}^{G} promotes the new network's extracted features to be oriented similarly to those recovered by the old model. The loss is constrained ($L_{dis}^{G} \leq 2$) (Hou et al., 2019).

In practise, the degree of necessity to keep existing knowledge changes depending on the number of new classes introduced in each phase (e.g., 10 classes vs. 100 classes). In response, it is suggested that the weight of the loss L_{dis}^{G} (abbreviated λ) be set adaptively as follows:

$$\lambda = \lambda_{base} \sqrt{|C_n| \,/\, |C_0|} \tag{5}$$

where $|C_o|$ and $|C_n|$ represent the number of old and new classes in each phase, and λ_{base} is a constant for each dataset.

The stored samples for previous classes have been entirely utilised. The ground-truth old classes are distinguished from all the new classes by a margin for each reserved sample x, using x itself as an anchor. The embedding of the ground-truth class is regarded as beneficial. An online mining approach for locating hard negatives is proposed. As hard negative classes, new classes are chosen with the largest reactions to x and utilise their embeddings as negatives for the relevant anchor. As a result, the projected margin ranking loss is calculated as:

$$L_{mr}(x) = \sum_{k=1}^{k} \max\left(\mathrm{m} - \left\langle \bar{\theta}(\mathrm{x}), \bar{\mathrm{f}}(\mathrm{x}) \right\rangle + \langle \bar{\theta}^{k}, \bar{f}(x) \rangle, 0\right) \tag{6}$$

where m denotes the margin threshold, $\bar{\theta}(\mathrm{x})$ denotes the ground-truth class embedding of x and $\bar{\theta}^{k}$ denotes one of the top-K new class embeddings chosen as hard negatives for x (Hou et al., 2019).

This method solves the imbalance in multi-class incremental learning from many perspectives. When the losses mentioned above are added together, they arrive at a total loss composed of three components, denoted as:

$$L = \frac{1}{|N|} \sum_{x \in N} \left(L_{ce}(x) + \lambda L_{dis}^{G}(x) \right) + \frac{1}{|N_o|} \sum_{x \in N_o} L_{mr}(x) \tag{7}$$

where N is a training batch obtained from X and $N_o \subset N$ are the old samples reserved in N. λ is a loss weight that is determined by Eq. (5) (Hou et al., 2019).

3.5 Dynamically Expandable Representation

DER seeks to improve the trade-off between stability and plasticity by incrementally augmenting previously acquired representations with additional characteristics and employing a two-stage learning technique.

DER uses a structure-based technique, and RPSNet and CCGN are the most similar works to it. RPSNet cannot preserve the inherent structure of each old notion and tends to forget the taught concepts gradually by adding the previously learned features and the newly learnt features at each ConvNet stage. In CCGN, the learnt representation may deteriorate gradually over time since only the parameters of a subset of layers are locked. DER retains the previously learnt representation and augments it with unique features parameterized by a new feature extractor. This allows it to keep the inherent structure of old ideas in the previously learnt representation subspace and re-use the structure via the final classifier to reduce forgetting.

DER uses Class incremental learning, unlike task incremental learning, does not require task id during inference. Specifically, the model monitors a stream of class groups $\{Y_t\}$ and their related training data $\{D_t\}$ during class incremental learning. The entering dataset D_t at step t, in particular, takes the form $(\boldsymbol{x_i^t}, y_i^t)$ where $\boldsymbol{x_i^t}$ is the input image and $y_i^t \in Y_t$ is the label inside the label set Y_t. The

model's label space is all observed categories $\tilde{Y}_t = \cup_{i=1}^{t} Y_i$, and the model is intended to predict well on all classes in $\tilde{Y}_t$ (Yan et al., 2021).

The rehearsal technique used in the system, which preserves a portion of the data as the memory M_t for future training. it separates the learning process for step t into two consecutive steps, as shown below.

1) Representation Learning Stage: It corrects the prior feature representation and augment it with a new feature extractor trained on incoming and memory data to achieve a better compromise between stability and plasticity. It provides an additional loss for the novel extractor in order to encourage it to learn varied and discriminative characteristics. To increase model efficiency, it is provided a channel-level mask-based pruning strategy that dynamically expands the representation based on the complexity of incoming classes. Figure 1 depicts an overview of the suggested representation.

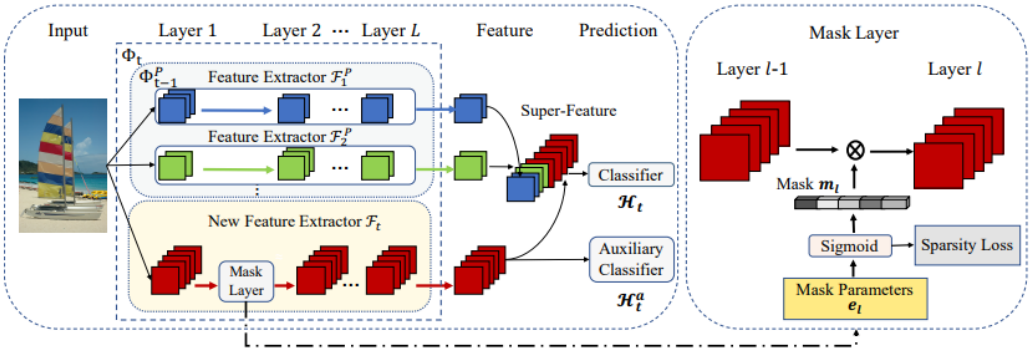

Figure 3.3: Dynamically Expandable Representation Learning (Yan et al., 2021)

2) Classifier Learning Stage: Following representation learning, the classifier is retrained with presently available data $\widetilde{\mathcal{D}}_t = \mathcal{D}_t \cup \mathcal{M}_t$ at step t to address the class imbalance problem using the balanced finetuning technique (Yan et al., 2021). The expandable representation is introduced as follows:

The model at step t consists of a super-feature extractor Φ_t and the classifier H_t. The super-feature extractor Φ_t is constructed by combining the feature extractor

$\Phi_t - 1$ with a freshly developed feature extractor F_t. Given an image $x \in \widetilde{D}_t$, the feature u extracted by Φ_t is derived by concatenation as follows:

$$\boldsymbol{u} = \Phi_t(\boldsymbol{x}) = [\Phi_{t-1}(\boldsymbol{x}), \mathcal{F}_t(\boldsymbol{x})] \tag{8}$$

In this case, it utilises the prior $F_1, \ldots, F_{t-1}$ and urge the new extractor F_t to learn just the innovative aspects of new classes. The feature u is then supplied into the classifier H_t to produce the following prediction:

$$p_{\mathcal{H}_t}(\boldsymbol{y} \mid \boldsymbol{x}) = \text{Softmax}\,(\mathcal{H}_t(\boldsymbol{u})) \tag{9}$$

The prediction $\hat{y} = \arg\, max p_{\mathcal{H}_t}(\boldsymbol{y} \mid \boldsymbol{x}), \hat{y} \in \widetilde{\mathcal{Y}}_t$ follows. For step t, the classifier is built to match its new input and output dimensions. To maintain past information, the parameters of H_t for the old characteristics are inherited from H_{t-1}, and its newly added parameters are randomly initialised (Yan et al., 2021).

Freeze the learnt function Φ_{t-1} at step t to prevent catastrophic forgetting since it captures the inherent structure of earlier data. In particular, the parameters of the last step super-feature extractor $\theta_{\Phi_{t-1}}$ and the Batch Normalization statistics are not updated. Then initialise F_t with F_{t-1} to utilise existing information for quick adaptation and forward transfer.

It learns the model with cross-entropy loss on memory and incoming data for Training Loss as follows:

$$\mathcal{L}_{\mathcal{H}_t} = -\frac{1}{|\widetilde{\mathcal{D}}_t|} \sum_{i=1}^{|\widetilde{\mathcal{D}}_t|} \log \left(p_{\mathcal{H}_t}(y = y_i \mid \boldsymbol{x}_i) \right) \tag{10}$$

where $\boldsymbol{x}_i$ is image and y_i is the corresponding label.

To enforce the network, it learns the diverse and discriminative features for novel concepts, an auxiliary loss is further developed operating on the novel feature $F_t(x)$. They provide an auxiliary classifier H_t^a that predicts the probability:

$$p_{\mathcal{H}_t^a}(y \mid x) = \text{Softmax}\,(\mathcal{H}_t^a(\mathcal{F}_t(x)) \tag{11}$$

To encourage the network to learn characteristics that distinguish between old and new concepts, H_t^a's label space is $|Y_t| + 1$, which includes the new category set Y_t

and the other class by considering all old concepts as one category. As a result, auxiliary loss is introduced and get the expandable representation loss as shown below:

$$\mathcal{L}_{\mathrm{ER}} = \mathcal{L}_{\mathcal{H}_t} + \lambda_a \mathcal{L}_{\mathcal{H}_t^a} \tag{12}$$

where λ_a is the hyper-parameter to control the effect of the auxiliary classifier. It is worth noting that $\lambda_a = 0$ for first step $t = 1$.

The super-feature is then dynamically increased based on the difficulty of novel concepts to eliminate model redundancy while maintaining a compact representation. To be more specific, a differentiable channel-level mask-based technique is used to prune filters of the extractor F_t, in which the masks are trained with the representation. Following mask learning, the mask is binarized and prune the feature extractor F_t to obtain the pruned network F_t^p (Yan et al., 2021).

The trimming is based on differentiable channel-level masks. The input feature map of convolutional layer l for a given image x is designated as f_l for the novel feature extractor F_t. To adjust the size of layer l, the channel mask is introduced as $\boldsymbol{m_l} \in \mathbb{R}^{c_l}$, where $m_l^i \in [0,1]$ and c_l is the number of channels in layer l. The mask is used to modify f_l as shown below:

$$f_l' = f_l \odot m_l \tag{13}$$

where f_l' is the masked feature map, $\odot$ means channel-level multiplication. To make the value of $\boldsymbol{m_l}$ fall into the interval [0, 1], the gating function is adopted as follows:

$$\boldsymbol{m_l} = \sigma(s\boldsymbol{e_l}) \tag{14}$$

where $\boldsymbol{e_l}$ denotes learnable mask parameters, the gating function $\sigma(\cdot)$ in this study employs the sigmoid function, and s is the scaling factor used to regulate the sharpness of the function. The super-feature $\tilde{u}$ of step t may be recast using such a mask method as:

$$\tilde{\boldsymbol{u}} = \Phi_t^P(\boldsymbol{x}) = [\mathcal{F}_1^P(\boldsymbol{x}), \mathcal{F}_2^P(\boldsymbol{x}), \dots, \phi_t(\boldsymbol{x})] \tag{15}$$

$\phi_t(\boldsymbol{x})$ is $F_t(\boldsymbol{x})$ with the soft masks during training. A big value is applied to s to binarize masks and obtain the pruned network $\mathcal{F}_t^P$, and $\phi_t(\boldsymbol{x}) = \mathcal{F}_t^P(\boldsymbol{x})$.

Mask Learning: During an epoch, a linear annealing schedule is applied for

$$s = \frac{1}{s_{max}} + \left(s_{max} - \frac{1}{s_{max}}\right)\frac{b-1}{B-1} \tag{16}$$

where b is the batch index, $s_{max} \gg 1$ is the scheduling hyper-parameter, and B is the number of batches in one epoch. The training era begins with all channels uniformly active. The mask is then binarized gradually as the batch index increases within an epoch (Yan et al., 2021).

The gradient of the sigmoid function is unstable due to the s scheduling, which is one of its drawbacks. To eliminate the impact of s, the gradient $\boldsymbol{g}_{\boldsymbol{e_l}}$ is adjusted with regard to $\boldsymbol{e_l}$:

$$\boldsymbol{g}'_{\boldsymbol{e_l}} = \frac{\sigma(\boldsymbol{e_l})[1-\sigma(\boldsymbol{e_l})]}{s\sigma(s\boldsymbol{e_l})[1-\sigma(s\boldsymbol{e_l})]}\boldsymbol{g}_{\boldsymbol{e_l}} \tag{17}$$

where $\boldsymbol{g}'_{\boldsymbol{e_l}}$ represents the corrected gradient.

Sparsity Loss: At each phase, the model is urged to minimise the number of parameters as much as possible while sacrificing as little performance as possible. As a result, sparsity loss is included based on the ratio of utilised weights in all accessible weights:

$$\mathcal{L}_S = \frac{\sum_{l=1}^{L} K_l \|\boldsymbol{m_{l-1}}\|_1 \|\boldsymbol{m_l}\|_1}{\sum_{l=1}^{L} K_l c_{l-1} c_l} \tag{18}$$

where L is the number of layers, K_l is the convolution layer l kernel size, layer $l = 0$ corresponds to the input image, and $\|m_0\|_1 = 3$. The final loss function is obtained after including the sparsity loss:

$$\mathcal{L}_{\text{DER}} = \mathcal{L}_{\mathcal{H}_t} + \lambda_a \mathcal{L}_{\mathcal{H}_t^a} + \lambda_s \mathcal{L}_S \tag{19}$$

where λ_s is the hyper-parameter to control the model size.

The classifier head is retrained during the representation learning stage to decrease the bias in the classifier weight induced by the unbalanced training. The classifier is specifically re-initialized with random weights before sampling a class-

balanced subset from the currently available data $\widetilde{D}_t$. In SoftMax, the classifier head is merely trained using the cross-entropy loss with a temperature δ. To enhance the margins between classes, the temperature influences the smoothness of the SoftMax function (Yan et al., 2021).

CHAPTER IV
RESEARCH METHODOLOGY

4.1 Research Description

The primary goal of this research is to compare three Incremental Machine Learning algorithms for detecting humans in images datasets that contain humans and otherwise. The algorithms are primarily: Incremental Support Vector Machines, Unified Classifier Incrementally via Rebalancing, and Dynamically Expanding Representation. If available, the algorithms will be compared primarily on their accuracy and running time.

4.2 Tools and Materials

4.2.1 Tools

The hardware and software to be used during the study is as follows:

1. Laptop with specification:
    - Laptop model : ASUS TUF Gaming FX504
    - CPU : Intel® Core™ i7-8750H CPU @ 2.20GHz
    - GPU : NVIDIA GeForce GTX 1050M
    - OS : Windows 10 Pro
    - RAM : 16 GB
2. Programming Language : Python 3.9
3. IDE : Microsoft Visual Studio Code

4.3 Research Steps

The steps that will be followed during the research to accomplish the research objectives are shown in Figure 4.1. The first step is to collect datasets and pre-process them. The second step is to modify the algorithms so that they can

accept the datasets as inputs and/or implement the algorithm. Finally, the algorithms are trained, tested, and tabulated.

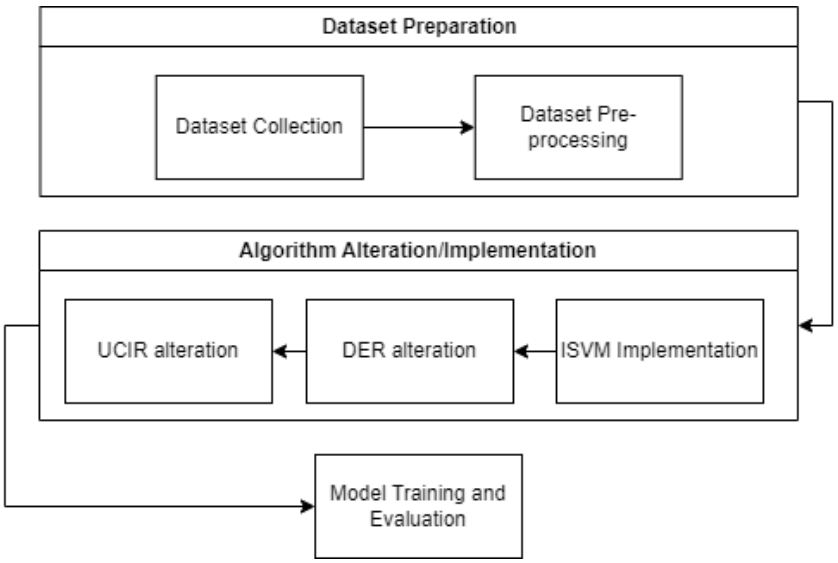

Figure 4.1: Research Steps

The datasets collected are required to have images with humans within them and images without humans, the datasets also have to include a substantial number of images in order for proper training. The datasets are then pre-processed to make sure they are in the format which the algorithms can take them as inputs. The algorithms collected are from their respective papers which have been downloaded from GitHub, afterwards their inputs and parameters are to be altered in a way for them to be able to train from the pre-processed dataset. After ensuring that the algorithms can take in the datasets, the algorithm is run and the models are then trained, tested and their results tabulated, the algorithms are run at least three times to get a good average in which to compare the three algorithms.

4.4 Dataset Preparation

4.4.1 Dataset Collection

The datasets collected from Kaggle must meet certain criteria, including the following: they must contain at least or close to 1000 images, half of which must include humans within the image, and the images must be labelled whether they

contain humans or not, or they will be pre-processed to be labelled in the appropriate way.

The datasets that match the requirements are namely:

4. Human Detection Dataset (Konstantin, 2022),
5. Human dataset latest (Veera, 2020),
6. Horses or Humans Dataset (Sani, 2018).

Table 4.1: Dataset Description

Dataset	Number of Images		
	With Humans	**Without humans**	**Total**
(Konstantin, 2022)	559	362	921
(Veera, 2020)	4377 + 340	5792 + 236	10745
(Sani, 2018)	527 + 128	500 + 128	1283

The datasets shown in Table 4.1 contain images with humans and images without humans in them, these images have been separated into two folders and will be further divided into two categories: The human latest dataset (Veera, 2020) will be used as the training dataset because it contains the most images, while the other two datasets will be used for testing. The (Veera, 2020) and (Sani, 2018) datasets have additional image folders that are validation images as shown in Table 4.1 as no. of training images + no. of validation images in the number of images column, these images will be combined to their respective image folders thus the 'train' folder will have a total of 10,475 images and the 'test' folder will have a total of 2,204 images which later may be cut down to balance the dataset. The following subchapters describe the combination and pre-processing of the three datasets.

4.4.2 Dataset Pre-processing

The image distribution within each dataset is depicted in Table 4.1, these image datasets do not have labels; however, the images are separated into folders, with all images containing humans in "1" and those containing no humans in "0".

```
start
set folder directory
for each filename in folder do
    check if filename is an image
        read the image
        convert the image format to rgb
        resize the image to 255 by 255
        save the resized image with .png extension
end
```

Pseudocode 4.1: Pre-processing the dataset

Since none of the datasets on the list are available in Pytorch, they must all be manually downloaded, pre-processed, and stored in an image directory. The process shown in Pseudocode 4.1 goes as follows: Since the images from the datasets have been separated into their respective folders in the first place there is no need to manually label them, instead the images are compiled into a training and testing folder accordingly and the images are split into their respective class folders, where they will then be pre-processed using a Python script to be converted into a .png image and have the dimensions of 255x255 pixels and will be renamed to "C_ (X)" where C is the class label of the image and X is the image number within the folder.

4.5 Algorithm Alterations/Implementation

The algorithms to be used are Incremental Support Vector Machine, Dynamically Expandable Representation and Unified Classifier Incrementally via Rebalancing. These algorithms are obtained from the corresponding papers: (Hou et al., 2019), (Yan et al., 2021). As explained in Section 4.3.2 the datasets collected are not part of Pytorch and thus need to be pre-processed and added to the algorithms manually by adjusting the processes that take in the inputs that have been implemented by respective authors in order for the algorithms to be able to take in the datasets. As there are no ISVM algorithms the author will implement an ISVM algorithm.

4.5.1 Incremental Support Vector Machine Implementation

For the implementation of ISVM the main library used is Scikit-learn, as it provides various classification algorithms including a Stochastic Gradient Descent Classifier which will be used to approximately implement the ISVM as the SVM module within Scikit does not provide a way for the SVM to train partially/incrementally, for libraries such as LibSVM/Liblinear and Pegasos to be used the image pre-processing steps and dataset input steps would be different and are difficult for the author to implement due to the limitations of the author's knowledge to utilize the mentioned libraries appropriately.

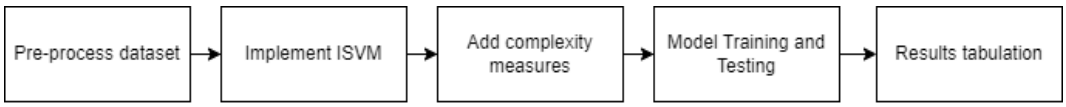

Figure 4.2: Steps to implement ISVM

The approximate implementation of ISVM is done according to the steps shown in Figure 4.2: Firstly, the pre-processed dataset is read and transformed accordingly; Secondly, the dataset read is converted to an array and transformed accordingly again; Thirdly, a Kernel approximation is applied to the data of the dataset, the SGD classifier would then be used to train the model incrementally over a certain number of epochs; Fourthly, complexity/performance measures are implemented before the model training; Finally, the model would be trained and tested the performance measures would be tabulated.

```
start
set path
for each file in files do
    if file has ".png" extension
        read image data
        resize the image to (32, 32)
        add the resized image to data array
        get label
        add the label to the label array
encode labels in the label array
convert data and label to numpy arrays using np.array
save data array as .npy file
```

```
13: save label array as .npy file
14: end
```

Pseudocode 4.2: Dataset pre-processing for ISVM

The pre-processed dataset would need to be read by the algorithm to make running the algorithm take less time the dataset will be transformed, converted into numpy array and saved. The transformation for the dataset is mainly to resize the images to 32x32, the data and labels are extracted from the images, the labels are then encoded, then both are saved into numpy arrays. Then all the main algorithm would need to do is read the numpy array to start training the model instead of going through the aforementioned process every time it is run. This process is shown in Pseudocode 4.2.

The kernel approximation used is the Radial Basis Function Kernel to further approximate the implementation of an ISVM using an SGD classifier. The parameters of the SGD classifier from sci-kit learn are: loss="hinge", penalty="l2" by using hinge loss it gives a linear SVM and using L2 penalty it uses the standard regularizer for linear SVMs. The models are trained using this classifier and applying a partial fit instead of a normal fit in order to allow the algorithm to train incrementally. The parameters that can be modified are the batch size and number of epochs, more detailed parameter changes are to be discussed in Chapter 5.

To assess the memory consumption of the algorithm, a memory profiler is necessary. The tracemalloc library is used to track the memory usage which is added to the start and end of the algorithm after its results have been outputted, which outputs the current and peak memory usage in Bytes during the run time of the training. The running time is also calculated by applying a function that subtracts the completion time of the algorithm's processing from the start time, yielding the running time of the algorithm. To run the algorithm simply run the python file.

4.5.2 Dynamically Expandable Representation

Before DER (Yan et al., 2021) is run a few steps are required to be followed for it to be able to train its model on the database mentioned in Section 4.3.1 and are as follows: Firstly, adjusting the inputs in the code to take in the new datasets, this includes the training set and validation set; Secondly, adding a function to enable memory usage monitoring and running time; Adjust the parameters to ensure the algorithm is able to train using the aforementioned dataset; Finally, train and test the model and tabulate the results. Figure 4.4 shows these steps and the steps are explained further in this section

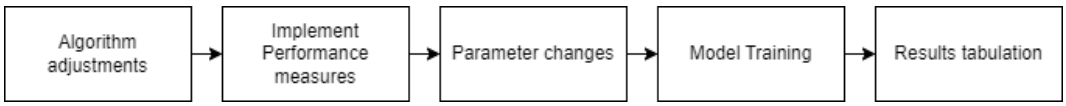

Figure 4.3: Steps to run DER

The main change within the algorithm is its input dataset and the way the algorithm reads it, as the algorithm uses CIFAR which the PyTorch library can directly take from without the need to manually download or tinker with.

Algorithm adjustments entail: preparing the dataset class for the algorithm, the dataset definitions, adjusting inputs and outputs as these would need to be similar to how the current algorithm takes in and reads the CIFAR dataset. The following steps to do this are: Read the data directory and take the folder name as its label, as the folder name is the class of the images; Read the images and transform them into 32x32 images, to match the dimensions of the images in CIFAR; Append data and label to their respective arrays and repeat for all the images in the directory; transform the target and label to ensure that it matches the format of CIFAR; Finally return images and labels according to the index specified by the algorithm when the object is called. These steps are shown in Pseudocode 4.3.

Another definition in dataset.py is created that follows the processes mentioned in order to make the dataset usable and the get_dataset definition is

modified to include the new dataset which is simply to add another elif function and naming it ‘humandetection’.

```
01: start
02: for each image name in directory do
03:     extract label from image name
04:     append label to label array
05:     im = concatenate image name and directory
06:     read image data using im
07:     resize image to 32 by 32
08:     append image to data array
09:     reshape data array to (-1, 3, 32, 32)
10:     transpose data array to (0, 2, 3, 1)
11: encode labels in label array into numerical values
12: get the length of data array
13: get the input index
14: get the image data from data array with index idx
15: get the target label from label array with index idx
16: return image and target
```

Pseudocode 4.3: Custom dataset class for DER and UCIR

Performance measures are implemented using: the tracemalloc library which is used to track the memory usage, used in the beginning and the end of the algorithm after its results have been outputted, this library outputs the current and peak memory usage in Bytes during the run time of the training; The algorithm's running time is determined by applying a function that subtracts the processing time for the algorithm from its start time.

The possible changes to the available parameters for the algorithm include: the epsilon, batch size, the number of classes and workers per increment set for the model which are set to 1e-8, 128, 10, 10 respectively; number of epochs, learning rate, learning rate decay, weight decay, to fine tune the last layer which is set to 30, 0.1, 0.1, and 5e-4 respectively; with the residual networks having a feature size of 64 and the algorithm allows the use of 18 and 32 layers.

The main parameter changes that will be taken for this study would be to the number of classes used to train the model, instead of 10 classes 2 would be used as the dataset contains only two classes of object being images with humans and

ones without, and possibly the number of epochs to be used in training, the learning rate scheduling of the model training and the number of GPU workers the algorithm will use, more detailed parameter changes are discussed in Chapter 5.

In order to run algorithm, the author has provided a shell script in their GitHub, their folder location being \codes\base\scripts, in order to run the algorithm, run run.sh.

4.5.3 Unified Classifier Incrementally via Rebalancing

For UCIR (Hou et al., 2019) similar steps are taken as DER, mentioned in Section 4.5: To begin, the algorithm is adjusted to accommodate the additional datasets, which comprise the training and validation sets. Second, include a function for measuring memory utilisation and running time. Change the parameters so that the algorithm can train on the aforementioned dataset. Finally, train the model and tabulate the results. These steps are as shown in Figure 4.6 and are further elaborated below.

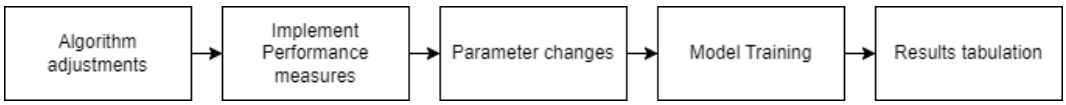

Figure 4.4: Steps to run UCIR

UCIR, like DER, employs CIFAR and PyTorch, so the algorithm must be adjusted to accommodate the new dataset. To accomplish this, Pseudocode 4.3 is implemented as a class and is initialized three times in the two main Python files and another being in the evaluation Python file having one initialization to be replaced. These three initializations being the training set, testing set and evaluation set which are in their base incremental learning file and their cosine normalisation incremental learning file, the last alteration being in the evaluation file which initializes the evaluation dataset. These replacements are done to exchange the pre-existing datasets with the new dataset.

The parameters that can be modified within the algorithm are as follows: classes per group, the number of classes in the first group, epochs, temperature for

distillation, beta for distillation, ratio for resample, the training, testing and evaluation batch size, learning rates, weight decay and the algorithm allows for the use of 18 and 32 layers. The parameters that will be modified include the: number of classes, classes per group, the number of classes in the first group which will be change to 2 to accommodate the number of classes for the new dataset. The number of epochs and batch size will be also be altered to match DER, more detailed parameter changes are discussed in Chapter 5.

The tracemalloc library is used to track memory consumption, which is utilized at the start and at the conclusion of the algorithm after the results have been outputted. This library produces the current and peak memory usage in Bytes during the training run time. The running time of the algorithm is calculated by subtracting the algorithm's processing time from its start time.

In order to run the algorithm, the author has provided a shell script.

4.6 Model Training and Evaluation

Each algorithm will be run 3 times, the results of the three runs and the average will be tabulated in order to compare the three algorithms, the three algorithms will be compared based on their accuracy, memory consumption and the running time it takes for the algorithms until it is finished with its processes.

Accuracy in machine learning is defined as the fraction of correct predictions among all predictions made. To measure the accuracy a simple formula is used:

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{CorrectPredictions}}{\boldsymbol{TotalPredictions}} \quad (1)$$

The higher percentage of accuracy the better the model is at predicting whether a human is in the image.

The complexity of the algorithms in this study includes space complexity and running time:

To compute the space complexity of the algorithms a memory profiler is used during the execution of the algorithm.

$$\boldsymbol{Space\ Complexity\ =\ Auxiliary\ Space\ +\ Input\ space} \quad (2)$$

The space complexity of an algorithm is the amount of memory required to run it in relation to the size of the input. Auxiliary space and space consumed by input both contribute to space complexity with Auxiliary Space being the extra space or the temporary space used by the algorithm during its execution. The memory used will be recorded in MegaBytes.

The running time of an algorithm is determined by how much time has passed during the execution of the algorithm, this is computed by:

$$\boldsymbol{Running\ Time = End\ Time - Start\ time} \quad (3)$$

The running time is measured in seconds. The lower the running time of an algorithm would mean that the algorithm takes less time to be executed.

CHAPTER V
IMPLEMENTATION

The purpose of this chapter is to demonstrate the basic processes needed to produce the datasets and train the three models as mentioned in the previous chapters. As a result, it focuses exclusively on the implementations associated with these sets of goals.

5.1 Dataset Preparation

Firstly, the datasets mentioned in Chapter 4 are downloaded from their respective sites, the datasets are then extracted into a main directory which will be the main root for pre-processing the images.

Before the images are transformed, Human Detection Dataset (Konstantin, 2022) and Horses or Humans Dataset (Sani, 2018) are separated into another folder to be used as the test and evaluation set whereas Human dataset latest (Veera, 2020) is used as the train set. The image names are then renamed into "C_ (X)" where C is the class number and X is the image number, this is done by selecting all the images and renaming them all at once by inputting 0_ in the rename field, this is done to easily do the next step when deleting the number of images and if the images are all in the same folder it would still be easy to extract the labels.

As the CIFAR dataset is a balanced dataset, meaning the number of images in all classes are the same, the same is done to also ensure that training is not influenced by class imbalance, this is done by removing 224 images from the human class in the test set and 1314 images from the nonhuman class in the train set to have a total of 1980 images in the test set with 990 images in each class and 9434 images in the train set with 4,717 images in each class.

The images are then transformed into images with 255 by 255 pixels in dimension and to PNG format if needed, this is done through the opencv2 library using the resize function and the cvtColor function respectively and

saving/renaming the image format to .png, Source code 5.1 shows the implementation of the aforementioned pre-processing.

```
folder = 'train\\0'
for filename in os.listdir(folder):
  # read the image file
  im = cv2.imread(os.path.join(folder, filename))
  # change format to rgb
  im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
  # resize the image
  im_resized = cv2.resize(im, (255, 255))
  # save the resized image
  cv2.imwrite(os.path.join('train\\resize_0\\' +
              filename + ".png"), im_resized)
```

Source code 5.1: Image pre-processing implementation

Firstly the folder is instantiated, this is to be manually changed into each of the image folders to be used; A loop to iterate through all images is then started; The image is then read using cv2.imread and is changed from BGR to RGB, if the image is already in RGB format it does not change due to how cv2.cvtColor works; the image is then resized to be in 255 by 255 pixels dimensionally by using cv2.resize and putting it into im_resized; Finally the resized image is saved to another folder named “resized_x”, where x is the class label, using the cv2.imwrite function; The images converted are separated to ensure that there are no confusions for the author and to not allow duplicate images.

5.2 Custom Dataset Implementation

The custom dataset implementation for DER and UCIR mimics the Pytorch torchvisions module for using a CIFAR10 and CIFAR100 dataset where it downloads the dataset, reads the labels and data, appends the labels and data, transforms the data by reshaping and transposing it and returns data and labels. This is similarly done for the custom dataset implementation, the main differences being: the images would be read directly instead of being from a binary format, the images would be further transformed to be 32 by 32 pixels in dimension, the labels would be extracted in the implementation, and a label encoder is applied to the label array

to enable the algorithm to properly use the label array, the class is then called whenever the algorithm requires it.

```
07: def get_label(img_path):
08:     # Extract the label from the filename
09:     filename = os.path.basename(img_path)
10:     label = filename.split("_")[0]
11:     return label
```

Source code 5.2: Get label definition

Source code 5.2 shows the implementation of the of the get_label definition that is utilized to extract the labels from the filename of the images into a string that will be used in the initialization of the custom dataset.

Firstly, the directory is initialized into ‘filename’, then the label is made by splitting the filename with ‘_’ and taking the first character from it and returning the label value which is in string.

```
10: class CustomDataset(data.Dataset):
11:     @staticmethod
12:     def get_label(img_path):
13:         # Extract the label from the filename

18:     def __init__(self, root: str, transform: Optional[Callable] =
None, target_transform: Optional[Callable] = None) -> None:
19:         self.root = root
20:         self.transform = transform
21:         # self.train = train
22:         self.target_transform = target_transform
23:         self.data: Any = []
24:         self.targets = []
25:         # now load the picked numpy arrays
26:         for root, dirs, files in os.walk(root):
27:             for file in files:
28:                 if file.endswith(".png"):
29:                     img_path = os.path.join(root, file)
30:                     img = cv2.imread(img_path)
31:                     img = cv2.resize(img, (32, 32))
32:                     self.data.append(img)
33:                     self.targets.append(CustomDataset.get_label(img_
path))
35:         self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
```

```
36:         self.data = self.data.transpose((0, 2, 3, 1))  # convert to
HWC
37:         label_encoder = LabelEncoder()
38:         self.targets = label_encoder.fit_transform(self.targets)
```

Source code 5.3: Implementation of custom dataset class (1)

The class ‘CustomDataset’ for the custom dataset is defined Source code 5.3. The ‘for’ loop only takes into account files with a ‘.png’ extension, this makes sure that the file is a PNG image and iterates across all files in the specified directory using ‘os.walk()’. The label is extracted from the image file name using the get_label() function from Source code 5.2. The image files' root directory and any optional transformations for the input data and target labels are sent into the function method init(). The image data is loaded within the function using the 'cv2.imread()' method and transformed using 'cv2.resize()' to a fixed size of 32x32 pixels. Two distinct lists 'self.data' and 'self.targets', are used to hold the preprocessed image data and the corresponding labels. Then, using the 'np.vstack()' function, the image data is stacked into a 4D array with the dimensions (-1, 3, 32, 32), where -1 denotes the number of samples and transposed into HWC format which Pytorchs’ CIFAR dataset utilize. The label list is then encoded using the ‘LabelEncoder()’ method from Pytorch and with a fitting transform.

```
40:     def __len__(self):
41:         return len(self.data)
43:     def __getitem__(self, index) -> Tuple[Any, Any]:
44:         img, target = self.data[index], self.targets[index]
45:         img = Image.fromarray(img)
47:         if self.transform is not None:
48:             img = self.transform(img)
50:         if self.target_transform is not None:
51:             target = self.target_transform(target)
52:
53:         return img, target
```

Source code 5.4: Implementation of custom dataset class (2)

Source code 5.4 is the continuation of Source code 5.3 where len() and getitem() methods are made. The length of the 'self.data' list determines the number of samples in the dataset, which is returned by 'len()'. Given an index, the function

‘getitem()’ is used to obtain a specific sample from the collection. The 'self.data' and 'self.targets' lists are initially retrieved to obtain the picture data and associated label at the provided index. Then, using "Image.fromarray()," it converts the image data into a PIL Image object. The image data and label are changed using the supplied function if a data transformation function is specified in the function if "self.transform" or "self.target_transform" is not None. The label and altered image data are returned as a tuple.

```
File: data.py
158:         current_class_idx = 0
159:         train_root_dir =
'Z:\ProposalAlgos\humandetectiondataset\\train'
160:         train_dataset = dataset(root=train_root_dir)
161:         test_root_dir =
'Z:\ProposalAlgos\humandetectiondataset\\test'
162:         test_dataset = dataset(root=test_root_dir)
163:         self.train_dataset = train_dataset
164:         self.test_datasets = test_dataset
165:         self.n_tot_cls = 2  #number of classes in whole dataset
```

Source code 5.5: Calling of the custom dataset for DER

Source code 5.5 shows how the custom dataset class is utilized to initialize the dataset for DER. The root directory is first specified which is to be used by the custom dataset class to know what root directory to use, then the custom dataset class is called to create the dataset objects for training and testing respectively and the total number of classes is specified in n_tot_cls, which is 2, to be used by the algorithm.

```
File: class_incremental_cifar100.py
155: train_root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\train'
156: trainset = evd.CustomDataset(root=train_root_dir,
transform=transform_train)
157: test_root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\test'
158: testset = evd.CustomDataset(root=test_root_dir,
transform=transform_test)
159: root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\test'
160: evalset = evd.CustomDataset(root=root_dir,
transform=transform_test)

File: class_incremental_cosine_cifar100.py
```

```
187: train_root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\train'
188: trainset = evd.CustomDataset(train_root_dir,
transform=transform_train)
189: test_root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\test'
190: testset = evd.CustomDataset(test_root_dir,
transform=transform_test)
191: root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\test'
192: evalset = evd.CustomDataset(root_dir, transform=transform_test)

File: eval_cumul_acc.py
63: root_dir = 'Z:\ProposalAlgos\humandetectiondataset\\test'
64: evalset = evd.CustomDataset(root=root_dir, transform=transform_test)
```

Source code 5.6: Calling of the custom dataset for UCIR

Source code 5.6 shows mostly the same as Source code 5.5 except the custom dataset is utilized for the DER algorithm, the custom dataset class is called in three different Python scripts as it is how the original author initializes the CIFAR dataset. The only difference with the DER custom dataset call is that the evaluation set object is created using the test set.

The custom dataset implementation for ISVM similarly uses the steps for UCIR and DER but instead of reading the dataset directly the datasets are converted into a numpy array where train and test data and labels are extracted to be used by the model. The following procedure was discussed in the previous chapter, furthermore for a fair comparison the images are also transformed to be 32 by 32 pixels. Although a feature extractor is typically used before training the model it would not be used in this case as the feature extractors, such as HOG and patch extraction seem from sci-kit learn seem to cause issues with the algorithm. Source code 5.3 shows the implementation of the custom dataset.

```
13: # Load images
14: path = "Z:\ProposalAlgos\humandetectiondataset\\train"
15: total = 0
16: X_train, Y_train = [], []
17: for root, dirs, files in os.walk(path):
18:     for file in files:
19:         if file.endswith(".png"):
20:             total = total+1
21:             img_path = os.path.join(root, file)
```

```
22:             img = cv2.imread(img_path)
23:             img = cv2.resize(img, (32, 32))
24:             #fd, hog_image = hog(img, orientations=8,
pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=True,
multichannel=True)
25:             X_train.append(img)
26:             Y_train.append(get_label(img_path))
```

Source code 5.7: Custom dataset implementation for ISVM (1)

Source code 5.7 retrieves images from the given path and stores them in X_train and Y_train as numpy arrays. The method reads the images using the cv2 and resizes them to a shape of (32, 32). The get_label method from Source code 5.2 is used to acquire the label for each image.

```
27: # Convert to numpy array
28: label_encoder = LabelEncoder()
29: Y_train = label_encoder.fit_transform(Y_train)
30: X_train = np.array(X_train)
31: Y_train = np.array(Y_train)
32: np.save('Z:\ProposalAlgos\ISVM\ISVM\X_train.npy', X_train)
33: np.save('Z:\ProposalAlgos\ISVM\ISVM\Y_train.npy', Y_train)
```

Source code 5.8: Custom dataset implementation for ISVM (2)

The labels from the Y_train are transformed into numerical values using the ‘LabelEncoder’ method from sklearn. The X_train and Y_train numpy arrays are created, and then they are stored as .npy files in the designated locations. This process is then repeated with the test set, using the appropriate path and saving the arrays as X_test.npy and Y_test.npy. The saved numpy arrays are to be used as the custom dataset for the training of the ISVM algorithm.

5.3 ISVM Implementation

The approximate implementation of the ISVM goes as follows: Firstly, the dataset arrays are loaded namely the train data and labels and test data and labels; The train set are then shuffled; Both test and train data are reshaped into (-1, 2048) which come from the dimensions and the number of colours thus 32*32*3; The train and test labels are flattened.

```
14: start_time = time.time()
15: tracemalloc.start()
16: # Load Datasets
17: X_train = np.load('Z:\ProposalAlgos\ISVM\ISVM\X_train.npy')
18: Y_train = np.load('Z:\ProposalAlgos\ISVM\ISVM\Y_train.npy')
19: X_test = np.load('Z:\ProposalAlgos\ISVM\ISVM\X_test.npy')
20: Y_test = np.load('Z:\ProposalAlgos\ISVM\ISVM\Y_test.npy')
21: # Print the shapes of the arrays

27: # Shuffle the dataset
28: X_train, Y_train = shuffle(X_train, Y_train, random_state=0)
29: # Split the dataset
30: #Reshape sets
31: X_train, X_test = X_train.reshape(-1,32*32*3), X_test.reshape(-
1,32*32*3)
32: Y_train, Y_test = Y_train.flatten(), Y_test.flatten()
```

Source code 5.9: ISVM Implementation (1)

As shown in Source code 5.9 the performance measures, ‘start_time’ and ‘tracemalloc’ are initialized, the training and testing numpy arrays made from Source code 5.7 and 5.8 are then loaded using ‘np.load’ and the training set is shuffled. The image data arrays are then reshaped into the shape of (-1, 2048) where -1 is the index of the data and 2048 is the image data in numbers where it comes from 32 by 32 pixels with 3 colors, RGB whilst the train and test labels are flattened.

```
34: # Initialize the model
35: rbf_feature = RBFSampler(gamma=1, random_state=1)
36: X_train = rbf_feature.fit_transform(X_train)
37: X_test = rbf_feature.fit_transform(X_test)
38: clf = SGDClassifier(loss="hinge", penalty="l2")
39: # Load the existing model if it exists
40: try:
41:     clf = joblib.load('Z:\ProposalAlgos\ISVM\ISVM\ISVMmodel.joblib')
42:     print('Loaded existing model')
43: except:
44:     print('No existing model found')
```

Source code 5.10: ISVM Implementation (2)

The kernel approximation method is applied to the train and test data, the kernel approximation method being Radial Basis Function Kernel this function is taken from sci-kit learns kernel_approximation module RBFSampler; the classifier

is then initialized, the classifier being the SGDClassifier from sci-kit learn using hinge loss, L2 regularization and optimal learning rate, learning rate cannot be adjusted if partial_fit() is used; If an existing model is available it would be loaded into the classifier otherwise a new model will be made. The implementation of these steps is depicted in Source code 5.10.

```
45: #Train in Epochs since partial_fit doesn't use max_iter
46: n_epochs = 100
47: # Split the dataset into batches
48: batch_size = 128
49: n_batches = int(np.ceil(len(X_train) / batch_size))
50: # Train the model
51: for epoch in range(n_epochs):
52:     print("Training epoch:", epoch + 1)
53:     for i in range(n_batches):
54:         X_batch = X_train[i*batch_size:(i+1)*batch_size]
55:         Y_batch = Y_train[i*batch_size:(i+1)*batch_size]
56:         clf.partial_fit(X_batch, Y_batch,
classes=np.unique(Y_train))
57:     print("train accuracy : ", accuracy_score(Y_train,
clf.predict(X_train)))
58:     print("test accuracy : ", accuracy_score(Y_test,
clf.predict(X_test)))
59: joblib.dump(clf, 'Z:\ProposalAlgos\ISVM\ISVM\ISVMmodel.joblib')
60: print("saved model: ISVMmodel.joblib")
61: Y_test_preds = clf.predict(X_test)
62: Y_train_preds = clf.predict(X_train)
```

Source code 5.11: ISVM implementation (3)

The number of epochs and batch size are initialized whereas the number of batches are calculated to be the total number of images divided by batch size; the model is then train using the classifier and is partially fit, using sci-kit learns partial_fit() method, to incrementally train the model; After the model is trained the model is saved for future use, if needed; the performance measures are calculated and output. The implementation of these steps is shown in Source code 5.11.

```
63: #results
64: print("Train Accuracy: {}".format(accuracy_score(Y_train,
Y_train_preds)))
65: print("Test Accuracy: {}".format(accuracy_score(Y_test,
Y_test_preds)))
```

```
68: #perfomance measures
69: Running_time = time.time() - start_time
70: print("Running time: %.2f seconds" % Running_time)
71: # Get the peak memory usage
72: _ , mem_usage = tracemalloc.get_traced_memory()
73: print(tracemalloc.get_traced_memory())
74: print(f"Peak memory usage: {mem_usage / 10**6} MB")
75: tracemalloc.stop()
```

Source code 5.12: ISVM Implementation (4)

Source code 5.12 prints the performance measures, these being the train and test accuracy, which is calculated using sklearns' accuracy_score() and cl.predict methods, the running time is calculated using the current time and subtracting it from the start_time variable previously initialized in Source code 5.9, the memory tracing snapshot is taken, printed and stopped.

5.4 Parameter Adjustments

The parameter changes for DER and UCIR would be similar as the parameters they use are similar. The main parameters to change for this research are shown in Table 5.1, parameter changes that are unique to each algorithm will be discussed separately.

Table 5.1: Parameter adjustments for DER and UCIR

Parameters	Value
Increment/number of classes in first group	2
Epochs	100
Batch size	128
Learning rate	0.1
Scheduling	35, 70
Learning rate decay	0.1
Weight decay	0.0005
Memory size	2000

These parameters are changed in the config files for DER, specifically in the 2.yaml, and within the 2 main codes for UCIR, specifically class_incremental_cifar100.py and class_incremental_cosine_cifar100.py.

The Increment parameter is used in DER to specify the number of classes in the first group and is used in a calculation to determine the number of runs the algorithm goes through, the calculation is total number of classes divided by the number of first classes thus if the increment number is more than two the algorithm will not run. The number of classes in the first group parameter is also used in UCIR albeit only for that reason, the number of runs determined in UCIR is its own parameter and is set to 1 to match DER.

Epoch, batch size, memory size and learning rate related parameters are set the way it is due to hardware limitations, while the epoch could be set higher it would take longer for the model to train. The batch size could be smaller to achieve better results but due to the reasons explained previously it is set to 128. The memory size is the number of exemplars set for one epoch in DERs case it is 2000 per epoch whereas for UCIR it is set to 1000 for each class thus 2000 per epoch. The worker parameter for DER is set to 0 due to the hardware unable to make use of NVIDIAs CUDA tools to utilize the GPU for training, although both algorithms use the GPU the currently used hardware is unable to use workers. The implementation of the parameter changes are shown in Source code 5.7 and 5.8 for DER and UCIR respectively.

```
27: #Optimization; Training related
28: task_max: 5
29: lr_min: 0.00005
30: lr: 0.1
31: weight_decay: 0.0005
32: dynamic_weight_decay: False
33: scheduler: 'multistep'
34: scheduling:
35:   - 35
36:   - 70
37: lr_decay: 0.1
38: epochs: 100
```

Source code 5.13: DER Parameter adjustments (1)

Source code 5.13 depicts the parameters used by the DER algorithm for training where: task max is the maximum number of tasks the algorithm will perform this is set to 5 by default, lr_min is the minimum learning rate set to 0.00005 by default, lr is the initial learning rate and is set to 0.1 by default, weight_decay is the weight decay of the training model which is set to 0.0005 by default and dynamic_weight_decay is set to false by default, the scheduler is set to multistep by default and is used to specify the kind of learning rate scheduling it uses, scheduling depicts when the learning rate is adjusted depending on what epoch the model is currently training on, this is set to 35 and 70 as it is 1/3rd and 2/3rds of the way from 100 which is the epoch set.

```
80: # Dataset Cfg
81: dataset: "human" #'imagenet100', 'cifar100'
82: trial: 2
83: increment: 2
84: batch_size: 128
85: workers: 0
86: validation: 0 # Validation split (0. <= x <= 1.)
87: random_classes: False #Randomize classes order of increment
88: start_class: 0 # number of tasks for the first step, start from 0.
89: start_task: 0

91: #Memory
92: coreset_strategy: "iCaRL"  # iCaRL, random
93: mem_size_mode: "uniform_fixed_total_mem" #uniform_fixed_per_cls,
uniform_fixed_total_mem
94: memory_size: 2000 # Max number of storable examplars
95: fixed_memory_per_cls: 5 # the fixed number of exemplars per cls
```

Source code 5.14: DER Parameter adjustments (2)

Source code 5.14 depicts the dataset parameters used in DER, dataset is set to “human” as it is the custom dataset that will be used, trial and increment is set to 2 to allow the algorithm to run and specifies the number of increments the algorithm will run, the reasons have been explained earlier in Chapter 5.4, batch size is set to 128 by default which specifies the number of images to be used to train in an epoch, workers has been set to 0 due to the current hardware unable to make use of workers which is only available for newer GPUs with Nvidia CUDA compatibility,

validation is set to 0 by default, random_classes is set to 0 by default, start_class and start_task depict the starting class to be used in training and at what task to start respectively, these has been set to 0 by default, the coreset strategy has been set to "iCarl" by default same with mem_size_mode which has been set to "uniform_fixed_total_mem", memory_size depicts the max number of exemplars that can be used in training, and fixed_memory_per_cls depicts the fixed number of exemplars per class which is set to 5 due to hardware constraints.

```
33: ######### Modifiable Settings ###########
34: parser = argparse.ArgumentParser()
35: parser.add_argument('--dataset', default='humandetection', type=str)
36: parser.add_argument('--num_classes', default=2, type=int)
37: parser.add_argument('--nb_cl_fg', default=2, type=int, \
38:     help='the number of classes in first group')
39: parser.add_argument('--nb_cl', default=2, type=int, \
40:     help='Classes per group')
41: parser.add_argument('--nb_protos', default=1000, type=int, \
42:     help='Number of prototypes per class at the end')
43: parser.add_argument('--nb_runs', default=1, type=int, \
44:     help='Number of runs (random ordering of classes at each run)')
45: parser.add_argument('--ckp_prefix',
default=os.path.basename(sys.argv[0])[:-3], type=str, \
46:     help='Checkpoint prefix')
47: parser.add_argument('--epochs', default=100, type=int, \
48:     help='Epochs')
49: parser.add_argument('--T', default=2, type=float, \
50:     help='Temperature for distialltion')
51: parser.add_argument('--beta', default=0.25, type=float, \
52:     help='Beta for distialltion')
53: parser.add_argument('--resume', action='store_true', \
54:     help='resume from checkpoint')
55: parser.add_argument('--fix_budget', action='store_true', \
56:     help='fix budget')
57: parser.add_argument('--rs_ratio', default=0, type=float, \
58:     help='The ratio for resample')
59: parser.add_argument('--random_seed', default=1993, type=int, \
60:     help='random seed')
61: args = parser.parse_args()
```

Source code 5.15: UCIR parameter adjustments (1)

Source code 5.15 depicts the parameters UCIR algorithm uses when training the model: firstly the dataset is set to “humandetection”, this is to set the naming of the checkpoints utilized by the algorithm; num_classes, nb_cl, nb_cl_fg is set to 2 as there are only 2 classes in the dataset, nb_protos is the number of exemplars/prototypes to be used in training per class, this is set to 1000 to match DERs’ 2000 memory_size as nb_protos calculates the number of total exemplars by multiplying this variable by the number of classes; nb_runs is set to 1 to match DER; epochs is set to 100; the rest are set by default and are explained in the Source code as comments.

```
63: ########################################
64: assert(args.nb_cl_fg % args.nb_cl == 0)
65: assert(args.nb_cl_fg >= args.nb_cl)
66: train_batch_size       = 128            # Batch size for train
67: test_batch_size        = 128            # Batch size for test
68: eval_batch_size        = 128            # Batch size for eval
69: base_lr                = 0.1            # Initial learning rate
70: lr_strat               = [35, 70]      # Epochs where learning rate
gets decreased
71: lr_factor              = 0.1            # Learning rate decrease
factor
72: custom_weight_decay    = 5e-4           # Weight Decay
73: custom_momentum        = 0.9            # Momentum
74:
75: np.random.seed(args.random_seed)        # Fix the random seed
76: print(args)
77: ########################################
105: dictionary_size     = 4717
```

Source code 5.16: Parameter adjustment for UCIR (2)

Source code 5.16 also depicts the parameters used by the UCIR algorithm for training its model: train, test, eval _batchsize is set to 128 by default, the same goes with base_lr, lr_factor, custom_weight_decay, custom_momentum, ckp_prefix and the random seed(args.random.seed) whereas lr_strat is changed to 35, 70 to match DER and the dictionary_size is set to the total number of images in one class in the train set.

The parameters set for ISVM are the same for DER and UCIR only for the applicable parameters these being epoch and batch size. The learning rate is set to optimal where the starting LR is 0.1 and is scheduled according to a heuristic proposed by Leon Bottou (Pedregosa, et al. 2011), the learning rate is unable to be altered due to partial_fit() being utilized in the implementation of ISVM. The parameters for ISVM are shown in Source code 5.6 line 38, 46 and 48.

5.5 Performance Measure

The performance measures implemented for the three algorithms are the model's accuracy, the running time of training and the memory used whilst training. For DER and UCIR the accuracy has been implemented by their respective authors whereas ISVM the accuracy is calculated by using sci-kit learns accuracy score module where it takes in the labels from the train/test set and predictions from the model and computes how many exact matches are made.

```
149: start_time = time.time()
150: # Start tracing memory allocations
151: tracemalloc.start()

442: #Print the results
443: Running_time = time.time() - start_time
444: print("Running time: %.2f seconds" % Running_time)
445: # Get the peak memory usage
446: _, peak = tracemalloc.get_traced_memory()
447: print(f"memory stats: ",tracemalloc.get_traced_memory())
448: print(f"Peak memory usage: {peak / 10**6} MB")
449: # Stop tracing memory allocations
450: tracemalloc.stop()
```

Source code 5.17: Sample Running time and peak memory (RAM) usage measure sample implementation

The running time is calculated using Pythons time module where the start time is set to the current time when the algorithm starts and an end time is set after the model has finished training, the end time is then subtracted by the start time to give the running time in seconds. This is implemented to all three algorithms.

The memory usage is implemented with Pythons tracemalloc module, it allows for tracing memory blocks allocated by Python, it is instantiated by tracemalloc.start(), the main usage of the module is to find the peak memory (RAM) usage during the runtime of the algorithm, this is done using tracemalloc.get_traced_memory() after the training has been finished, as it outputs current and peak memory usage only the peak will be tabulated. Source code 5.9 shows the implementation of the running time and peak memory usage performance measure for UCIR, it is similarly done for DER and ISVM.

CHAPTER IV
RESULTS AND DISCUSSION

6.1 Dataset Preparation

The results of dataset preparation for the three algorithms are shown following the implementation steps.

Table 6.1: Dataset Preparation Image count

Directory	Number of images
\train\	9,434
\test\	1,980
\train\0\	4.717
\test\0\	990

Figure 6.1 shows the number of images in the train and test set folder where the train set has 9,434 images with a class having 4,717 images and the test set has 1980 images with a class having 990 images.

When using an unbalanced image dataset, which was the way the initial dataset was without the pruning of images, the UCIR algorithm refuses to start training the model as the dictionary size is fixed as shown in Source code 5.16.

Table 6.2: Image pre-processing sample

Directory	Filename	Filetype	Width (p)	Height (p)
\train\0\	0_ (1)	PNG	255	255
\train\0\	0_ (57)	PNG	255	255
\train\0\	0_ (427)	PNG	255	255
\train\1\	1_ (1)	PNG	255	255
\train\1\	1_ (57)	PNG	255	255
\train\1\	1_ (427)	PNG	255	255

Table 6.2 shows the details of sample images from the train folder where the images have: the name format, PNG as their filetype and dimensions of 255 by 255 pixels.

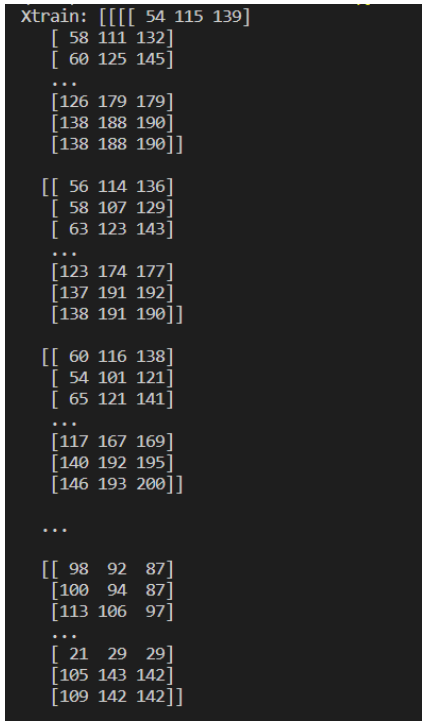

Figure 6.1: Train data arrays sample

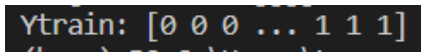

Figure 6.2: Label array sample

Figure 6.3 shows samples of data and label numpy arrays of the train set used for ISVM.

6.2 Custom Dataset Implementation results

The implementation results for the custom datasets are shown in Figure 6.4, 6.5 and 6.6 for DER, UCIR and ISVM respectively.

Table 6.3: DER dataset implementation results

Array Name	Length
nb(number of images)	9434

classes_order	2

Table 6.3 depicts the number of classes shown as classes_order [[0, 1]] showing that there are 2 classes and nb 9434 shows that there are 9,434 images used for training.

Table 6.4: UCIR dataset implementation results

Array Name	**Data point**			
	Length	**X-Axis**	**Y-Axis**	**Depth**
X_train_total	9434	32	32	3
Y_train_total	9434	-	-	-
X_test_total	1980	32	32	3
Y_test_total	1980	-	-	-
classes_order	2	-	-	-

Table 6.4 depicts the total number of images and labels shown, X_train_total.shape depicts the shape of the train data array and Y_train_total depicts the labels for the train set, The class_order depicts the total number of classes to be used in a specific run of training and in this case shows 2.

Table 6.5: ISVM dataset implementation results

Array Name	**Data point**			
	Length	**X-Axis**	**Y-Axis**	**Depth**
Xtrain	9434	32	32	3
Ytrain	9434	-	-	-
Xtest	1980	32	32	3
Ytest	1980	-	-	-

Table 6.5 depicts the total number of images, image data and label shape for the dataset array similar to the depiction shown for UCIR.

6.3 Parameter Adjustments results

The implementation results for the parameter adjustments are shown in Figure 6.3, 6.4 and 6.5 for DER, UCIR and ISVM respectively.

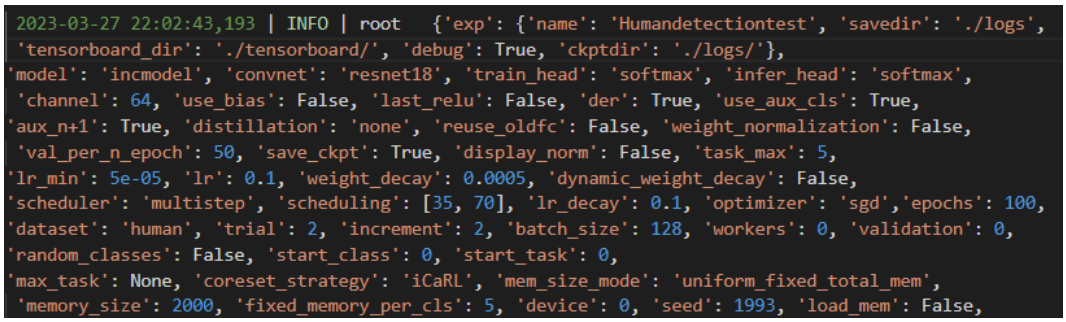

Figure 6.3: DER parameters output

Figure 6.3 shows the logs in which all the parameters set for the DER model are outputted before the model is trained, the parameter adjustments are therefore also shown in the logs. The logs have been broken down into smaller chunks to ensure visibility.

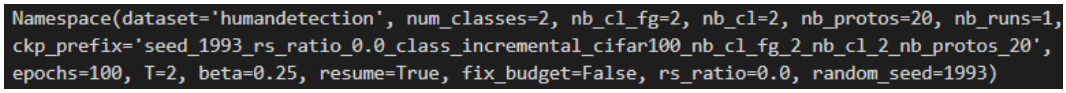

Figure 6.4: UCIR parameters output

Figure 6.4 similarly shows the logs for all the parameters set before training is done. The parameters changed are shown as implemented in the previous except for nb_protos as it is multiplied by 100 thus making it 2000, the parameter set for the algorithm is shown in Source code 5.8.

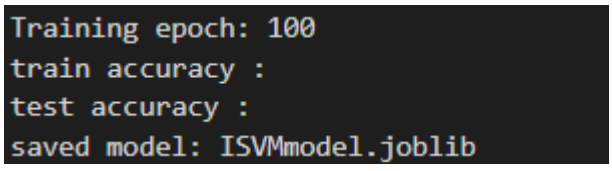

Figure 6.5: ISVM epoch output

Figure 6.5 depicts the total epochs done by the algorithm, this is the only result of the parameter adjustment printed, as the other parameter values do not get printed.

Performance results with varying parameters would have been preferable albeit the only parameter changes that can be made are to the epochs and batch sizes for training as it would be unfair to ISVM due to it only having these two parameters that can be changed. Furthermore, increasing the epoch may increase the accuracy and run time, same goes for decreasing the batch size, these parameter changes would not be very significant when compared to the other parameters that can be changed within DER and UCIR, additionally it would have put a heavier toll on the system which is undesirable.

6.4 Performance Measure results and comparison

The algorithms are run three times each, before each run the saved models are deleted to ensure that the model doesn't use a pretrained model, this is to make sure that there are no flukes when testing. Table 6.1 shows the results after the models have been trained using the custom dataset made.

Table 6.6: Performance Measure results of DER

Run	Test Accuracy (%)	Running Time (seconds)	Peak Memory (RAM) Usage (MB)
1	72.17	4955	162.498135
2	72.17	4840	162.498445
3	72.17	4827	162.497533

DER shows that across its three runs its accuracy is very consistent and is able to accurately classify the image nearly 3 out of 4 times, the time it takes to fully run the algorithm is consistently around 81 minutes on average while using at most 162.5 MB (RAM) per run.

Table 6.7: Performance Measure results of UCIR

Algorithm	Run	Test Accuracy (%)	Running Time (seconds)	Peak Memory (RAM) Usage (MB)
UCIR	1	49.04	2643	409.396074
	2	50.10	2601	409.396523
	3	50.81	2911	409.396410

Whereas UCIR also shows that its accuracy is consistent but is only accurate around half of the time, the time it takes to full run the algorithm is nearly half the time, on average when compared to DER, averaging at around 45 minutes per run but also uses up more than twice the memory at 409 MB.

Table 6.8: Performance Measure results of ISVM

Algorithm	Run	Test Accuracy (%)	Running Time (seconds)	Peak Memory (RAM) Usage (MB)
ISVM	1	52.02	11	277.122106
	2	52.32	10	277.122043
	3	52.32	11	277.122048

ISVM on the other hand is similar to UCIR in terms of accuracy with it being consistent and able to be correct in its classification ~50% of the time, the running time is massively faster averaging in 10.5 seconds across three runs while also using nearly half the amount of memory at most.

Table 6.9: Comparison of average Performance Measure results

Algorithm	Test Accuracy (%)	Running Time (seconds)	Peak Memory (RAM) Usage (MB)
DER	72.17	4874	162.498038

UCIR	49.98	2,718	409.396336
ISVM	52.22	10	277.122066

Table 6.9 shows the comparison of the average performance results of each algorithm where it can be seen that DER has the highest accuracy, ISVM takes the least amount of time and UCIR has the highest peak memory usage.

6.5 Analysis

From the results this research has taken, it shows that DER is a superior classification algorithm in terms of accuracy and peak memory (RAM) usage coming in at 72.17% and 162.5 MB respectively. Whereas UCIR falls behind ISVM minorly in accuracy averaging at 48.98% but it also falls behind both in terms of peak memory usage, 409 MB on average and running time of 45 minutes when compared to ISVM, the only advantage UCIR has, in this research, is its running time in comparison to DER.

The parameters could have been changed in order for the algorithms to learn more efficiently such as increasing the number of epochs to a suitable number, adjusting the batch sizes, number of prototypes, scheduling of learning rates, etc. Although this would allow the models to train for better accuracy it would also mean that it would take longer to train unless the hardware is sufficient and beefy enough to train using the more hardware intense parameters.

Furthermore, the results may show that the hardware is not suitable to run DER and/or UCIR optimally, this is attributed to DER using significantly less memory but nearly double the running time, it has been observed while training with DER, on the current hardware, utilizes 90-100% of the GPU thus using less memory (RAM), as it mostly relies on the GPU to train rather than the CPU, whereas for UCIR it utilizes at most 40% of the GPU, as it was programmed to partially use CPU and GPU, thus taking less time. For ISVM it does not use the GPU at all.

Peak GPU % usage for each algorithm is shown in Table 6.7 and is monitored using Windows Task manager and manually recording the peak GPU utilization during its run.

Table 6.10: Comparison of algorithms on peak GPU utilization

Algorithm	GPU Utilization (%)
DER	100
UCIR	49.7
ISVM	0

It should also be noted that DER and UCIR were generally developed for multi-class classification and may have been a reason as to why UCIR has less accuracy when compared to the approximate implementation of ISVM in this research, due to ISVM generally being a binary classification model.

Additionally, DER and UCIR were supposed to be able to run more than once whilst using a trained model/checkpoints to further train, this was not done as it is inapplicable to DER as it only allows 1 run due to the nature of how the increment part of the algorithm works, if the increment parameter was set to 1 it would only train on one class which would make the experiment unfair because UCIR and ISVM would still train on the two classes whilst running twice. The algorithms could have also been adjusted to train incrementally using quarters of the dataset instead of incrementally training it all at once to perhaps better train the model but the author does not have the knowledge to adjust the aforementioned algorithms to that level as of yet.

CHAPTER VII
CONCLUSION

7.1 Conclusion

The main purpose of the research was to compare incremental machine learning algorithms: the newer DER, the middle aged UCIR or the older ISVM, in terms of their accuracy, running time and memory usage. This was done to find out which algorithm would perhaps be suitable to be used to detect humans in ever-changing environments utilizing smaller hardware. The research was able find out which of the three algorithms was the most suitable for the given problem, the problem in this case would be a combination of three datasets which include human and non-human images.

The results of the research showed that DER performed the best in terms of accuracy and peak memory usage, while UCIR had the shortest running time but lower accuracy and higher memory usage compared to ISVM. Although the algorithms could potentially be adjusted for better efficiency, it would require more time and better hardware. Furthermore, the study found that the hardware used may not have been suitable for running DER and UCIR optimally, as DER relied heavily on the GPU, while UCIR partially used both the CPU and GPU, whereas ISVM solely relied on the CPU. It is important to note that DER and UCIR were designed for multi-class classification and may have performed worse in a binary classification setting.

The study emphasizes the significance of using the best classification method for the particular job and dataset. Although it was discovered that the DER method was the most accurate in this instance, it may not always be the best option for all classification issues. The research also emphasizes the necessity of having the right tools and resources to effectively train and evaluate machine learning models. Although the algorithms' parameters may be changed for greater efficacy, doing so might require more expensive hardware or longer training periods.

7.2 Future Works

During this research many problems cropped up that could be resolved in future papers: The glaring one being hardware, it is common sense but it should be emphasized that the more the data needs to be processed the better the hardware should be, especially so if the algorithm used does an unimaginable number of calculations per second. Whilst debugging DER and UCIR, an error cropped up due to the absence of Nvidia CUDA, which is a toolkit that allows the use of GPUs to accelerate general purpose processing, the current GPU was not able to make full use this toolkit, as it is not supported, although GPU was utilized for DER and UCIR it could only do so much with it.

When implementing ISVM, the use of Sci-kit learns SGDClassifier is not optimal as it is not an exact implementation of ISVM, it has been put to use in this research as neither LibSVM/LibLinear and Pegasos libraries were used due to the author not being knowledgeable enough to properly adjust the dataset to their requirements. For future works relating with ISVM LibSVM and/or LibLinear should be used.

Smaller issues such as using higher resolution images instead of 32 by 32 images in the final transformations of the dataset images could be implemented but this would also cause the algorithms to process more data thus beefier hardware would be preferable to run optimally for this case as well.

For future works it would be far more suitable to compare algorithms made for specific problems on those specific problems, where in this research multi-class classification was done for binary classification.

BIBLIOGRAPHY

Internet live stats - internet usage & social media statistics (2014) *Internet Live Stats - Internet Usage & Social Media Statistics*. Real Time Statistics Project. Available at: https://www.internetlivestats.com/ (Accessed: January 3, 2023).

Abhinava, B. & Majumdar, J., 2017. Automatic Detection of Human in Video and Human Tracking. *Interantional Journal of Engineering Research & Technology (IJERT)*, 6(10).

Ammar, B., Wali, A. & Alimi, A., 2011. *Incremental Learning Approach for Human Detection and Tracking.* Abu Dhabi, IEEE.

Ansari, M. A. & Singh, D. K., 2021. Monitoring social distancing through human detection for preventing/reducing COVID spread. *International Journal of Information Technology,* Volume 13, p. 1255–1264.

Cauwenberghs, G. & Poggio, T., 2001. *Incremental and Decremental Support Vector Machine Learning.* Cambridge, MIT Press.

Davis, J. W., Sharma, V., Tyagi, A. & Keck, M., 2009. Human Detection and Tracking. Dalam: A. J. Stan Z. Li, penyunt. *Encyclopedia of Biometrics.* New York: Springer New York, p. 708–712.

Geng, X. & Smith-Miles, K., 2009. *Incremental Learning.* Boston(MA): Springer US.

Gunopulos, D. & Domeniconi, C., 2001. *Incremental support vector machine construction.* California, IEEE, pp. 589-592.

Hanyu, T. & Zhao, Q., 2017. *Incremental Training of SVM-Based Human Detector.* Seoul, IEEE.

Hou, S. et al., 2019. *Learning a Unified Classifier Incrementally via Rebalancing.* Long Beach, IEEE, pp. 831-839.

Hu, X. et al., 2022. Online human action detection and anticipation in videos: A survey. *Neurocomputing,* Volume 491, pp. 395-413.

Ivašić-Kos, M., Krišto, M. & Pobar, M., 2019. *Human Detection in Thermal Imaging Using YOLO.* Istanbul, Association for Computing Machinery.

Jain, R., Kasturi, R. & Schunck, B. G., 1995. *Machine Vision.* s.l.:McGraw-Hill, Inc..

Joshi, A. J. & Fatih, P., 2010. *Scene-Adaptive Human Detection with Incremental Active Learning.* Istanbul, Istanbul.

Kachouane, M., Sahki, S., Lakrouf, M. & Ouadah, N., 2012. *HOG Based fast Human Detection.* Algiers, IEEE.

Konstantin, V., 2022. *Human Detection Dataset.* [Online]
Available at: https://www.kaggle.com/datasets/constantinwerner/human-detection-dataset
[Accessed 14 November 2022].

Laskov, P., Gehl, C., Krueger, S. & Müller, K.-R., 2006. Incremental Support Vector Learning: Analysis, Implementation and Applications. *Journal of Machine Learning Research (JMLR),* Volume 7, p. 1909–1936.

Lawal, I. A., 2019. Incremental SVM Learning: Review. Dalam: *Learning from Data Streams in Evolving Environments.* s.l.:Springer, Cham, p. 279–296.

Losing, V., Hammer, B. & Wersing, H., 2018. Incremental on-line learning: A review and comparison of state of the art algorithms. *Neurocomputing,* Volume 275, pp. 1261-1274.

Luo, Y., Yin, L., Bai, W. & Mao, K., 2020. An Appraisal of Incremental Learning Methods. *Entropy,* Volume 22, p. 1190.

Mohri, M., Rostamizadeh, A. & Talwalkar, A., 2018. *Foundations of Machine Learning.* 2nd penyunt. London: The MIT Press.

Nguyen, D. T., Li, W. & Philip O. Ogunbona, 2016. Human detection from images and videos: A survey,. *Pattern Recognition,,* Volume 51, pp. 148-175.

Paul, M., Haque, S. M. E. & Chakraborty, S., 2013. Human detection in surveillance videos and its applications - a review. *EURASIP Journal on Advances in Signal Processing,* p. 176.

Pedregosa, F. et al., 2011. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12(85), pp. 2825-2830.

Pop, D. O., Rogozan, A., Nashashibi, F. & Bensrhair, A., 2021. Pedestrian Recognition Using Cross-Modality Learning in Convolutional Neural Networks. *IEEE Intelligent Transportation Systems Magazine,* 13(1), pp. 210-224.

Robson, S. W., Aniruddha, K., David, H. & S., D. L., 2009. *Human detection using partial least squares analysis.* Kyoto, IEEE.

Sabatier, J. M., Ekimov, A. E. & Frederickson, C. K., 2007. *Methods for detecting humans.* United States of America, Paten No. EP2010935A2.

Sani, K., 2018. *Horses Or Humans Dataset.* [Online]
Available at: https://www.kaggle.com/datasets/sanikamal/horses-or-humans-dataset
[Accessed 14 November 2022].

Xia, Y., Huang, Y., Wang, L. & Geng, X., 2013. Pedestrian Detection Based on Incremental Learning. *Intelligence Science and Big Data Engineering,* Volume 8261, p. 603–610.

Yan, S., Xie, J. & He, X., 2021. *DER: Dynamically Expandable Representation for Class Incremental Learning.* Nashville, IEEE, pp. 3013-3022.