A

committee

B

Malware Classification using DNNs under Continual Learning Settings

Master's Thesis submitted to the

SIT - Schaffhausen Institute of Technology

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Software Engineering

presented by

Soraya Reyhan

under the supervision of

Prof. Bertrand Meyer

co-supervised by

Prof. Adil Khan

June 2022

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Soraya Reyhan
Schaffhausen, 15 June 2022

To my family

…

Abstract

Today, with the increasing use of the internet by various software users, we are witnessing the emergence of new types of malware on a daily basis. Because these malwares are mutants of older types of malwares, nowadays the methods which are signature-based are less effective for the detection and classification of malwares. Fortunately, in recent years, much attention has been paid to the use of machine learning (ML) in this area. For example, deep learning (DL) can be called one of the most important and efficient ML algorithms that are used for malwares detection and classification. But, one of the factors that reduce the performance of DL is catastrophic forgetting (CF). In fact, catastrophic forgetting occurs when a model that was previously trained and tested on some of the malwares, now when trained and tested on new types of malwares, does not have the former accuracy in detection and classification of previous malwares. Our purpose in this study is multiple folds. Firstly, we show that even a highly accurate deep model for malware classification (MC) performs poorly in continual learning (CL) settings. Secondly, we implement and compare two state-of-the-art CL techniques for solving the problem, and show that even the state-of-the-art fails when tested in the single-head setup. Thirdly, we study the impact of class imbalance on the CL performance of our model for MC. Fourthly, we also study the impact of freezing the feature learning layers on the CL performance of our model. Finally, we also show that a hybrid approach consisting of rehearsal and the regularization methods is more accurate for single-head CL for MC.[1]

[1] Code available at https://github.com/SorayaReyhan/online-explicit-regularizer

Contents

Figures

Tables

Chapter 1

Introduction

Today, the use of software in all areas is common and increasing. It can even be said that in some fields, the use of these softwares is an integral part of working in those fields. On the side, these softwares are increasing work efficiency in some areas, but they also carry serious risks for them. These threats are called malware. In fact, malware, also known as malicious software, is a computer program with a malicious purpose and performs a malicious action.

On the other hand, since the use of the internet is now very common among software users, we are witnessing the detection of thousands of new malware every day. This is because malware is mutating on the internet very fast. This has made the detection and classification of malware one of the most important priorities in the field of computer security. Because traditional antivirus programs use a signature based approach to detect and classify malware, so for applying their strategy we need to have a local database to store the detected patterns of malware. But the problem is that by making simple changes in the malware, the signature of the malware could be changed. Therefore, with this method, many malware can easily avoid signature based detection by encrypting, obfuscating, or packing. Given all of the above, it can be said that the evolution and diversity of malware is a serious threat to modern systems.

ML algorithms are widely used today to analyze malwares. This approach is increasing because, as mentioned earlier, new malwares are generally a mutated version of the old malwares. But since these ML algorithms require feature representation, feature learning, and feature engineering, so they are very time consuming to use. Hence, by using advanced ML algorithms such as DL, the feature engineering process can be completely skipped.

For malware, we do not have a static scenario where we have some malwares, and then in the future will be just dealing with this malwares. Malware is a scenario or problem that every day new malwares come. Indeed, we don't have a specific number of malwares that in the future we need to deal with these malwares. So, a malware problem is a scenario in which malwares come on a daily bases, and therefore if we are using ML for malware detection and classification we need to keep our ML model updating based on new malwares and this shows us the importance of CL in malware detection and classification.

But this is not easy because ML models have a very big weakness and that is they forget. Indeed when we train them in new tasks they tend to forget the previous tasks, this is called CF and it impacts their ability to learn on a continuous base. In simple terms, when a ML model receives an input, some weights are applied to its parameters over training time to approximate the desired output. If our input is an image of a cat, for example, we want our model to classify it as a cat. If we change our inputs to the hamster, the weights must be recalculated to approximate the new output. Consider that we've successfully trained our model to recognize cat images 98% of the time. Then we begin a new training session, this time for hamster images. The model adjusts its weights to distinguish hamster but loses its ability to recognize cats as a result. The term CF refers to this phenomenon.

Thus, on the one side, it is very important to consider the CL aspect for malware detection and classification on the other side we should also consider how these two things could be combined. Since in the past, researchers didn't consider an CL scenario for MC, So, our first contribution is to combine these two things together. We will first study what happens to the performance of the ML model that we develop for MC as we train this model continuously. Second, we will analyze the result and we trying to understand how can we ensure that our model is going to be accurate as well as learns incrementally by applying a explicit quadratic regularizer. It is indeed very challenging with ML.

This study consists of the following chapters: chapter two will do a literature review regarding 1. State of the art ML techniques for MC, 2. State of the art techniques to overcome CF in CL and, 3. State of the art strategies for task splitting in CL. Chapter three consists of all the important research questions that this study wants to answer. In chapter four we have the methodology of our study that are: 1. Malware classifier which contains: a. Features that is a description of how the data is represented as our input for the model and what features are used, and, b. Classifier which describes how the classification is done and what is our classification model and why we choose this kind of model in this study. 2. CL setups (single head or multihead), 3. Methods for CL (sgd, ewc and explicit ewc, replay), 4. Dataset characteristics (balanced or imbalanced dataset). Then in chapter five, we will have experimental evaluations that consist of the following sections: 1. Description of our dataset, 2. The experimental result describes the setup of each experiment and connects it with our research questions, 3. Results and discussion that present the results of each experiment (as tables, figures, etc.) and discuss them. 4. Further analysis and improvement in the results which contains: a. Applying majority-minority classes to a certain task and, b. Applying rehearsal along with explicit ewc, ewc and sgd. 5. Important Findings which in there we list all important lessons that we have learned from the study and describe their significance for the field of ML for MC. Then we have the conclusion as our final chapter.

So in brief, in this study we implement and compare two state of the art CL techniques (ewc, explicit ewc) for ML based on CL and DL, in response to the demand for effective MC.

Chapter 2

Literature Review

2.1 State of the art ML techniques for MC

In this section we discuss the most recent works in MC. The features retrieved from executable files are used to train ML models for malware detection and classification. Control flow graphs, opcodes, API calls are examples of such features. Because extracting these features is costly in most cases, raw bytes-based techniques are favored if equivalent accuracy can be achieved. Byte n-grams, for example, have been successfully employed as features. Another example is that executable files can be treated as images and image analysis techniques applied to them.[Jain et al., 2011]So based on this reason Jain et al. [2011] visualizes malware samples as images, then classifies them using image analysis techniques such as CNN and Extreme Learning Machines (ELM).

Signature-based detection is previously the most widely used technology in commercial antivirus. This method can only detect malware after the malicious executable has already caused damage and if the malware has been properly documented. As a result, signature-based detection methods could not discover new malwares that come in the future. To overcome this issue Santos et al. [2013] proposed a new strategy for detecting unknown malware families by using opcodes that come from disassembling the associated exe files. His model is based on the frequency with which opcode sequences appear. Santos et al. [2013] also provides a method for determining the importance of each opcode and assessing the frequency of each opcode sequence. Despite the fact that he gets satisfactory results from his method, it has an important drawback, which is a time-consuming procedure of disassembling executable files for extracting opcodes.

Based on a study from Alsulami and Mancoridis [2108] static signature-based techniques employed by antivirus systems are less successful against newer polymorphic and metamorphic malware, hence behavioral malware detection tries to improve their effectiveness. Behavioral malware classification tries to go beyond malware detection by identifying a malware's family using a naming scheme similar to the ones used by antivirus providers. Behavioral MC approaches capture the behavioral characteristics of running programs using run-time features such as file system or network activities. Behavioral malware classifiers have hurdles from the growing volume of malware samples, the diversity of malware families, and the range of naming methods used by antivirus providers. So Alsulami and Mancoridis [2108] presents a behavioral classifier that uses a Convolutional Recurrent Neural Network as his model and Microsoft Windows Prefetch files as his data. Based on his results his proposal model is good in classifying malware samples into common and rare malware families, and it can handle the introduction of additional malware samples and families progressively.

According to Kim [2018] static analysis, which is a means of evaluating possibly malicious files, has been a popular strategy in malware detection. However, when malicious programs get more powerful and embrace the capability of obfuscating their binaries to execute the same malicious functions, static analysis becomes increasingly challenging for newer variants. Kim [2018] evaluated a method that is a novel dynamic malware analysis method that may generalize better to newer variants than static analysis. Based on recent successes in Natural Language Processing (NLP), widely used document classification approaches were evaluated in identifying malware by doing such analyses on system calls, which include useful information about a program's functioning as requests to the kernel. Features are retrieved from benign and malicious program system call traces, and the process of classifying these traces is addressed as a binary document classification task of system call traces. The parameters were removed from the system call traces, leaving only the system call function names. The features were divided into n-grams and weighted using the Term Frequency-Inverse Document Frequency method. Kim [2018] demonstrates that Linear Support Vector Machines (SVM) optimized by Stochastic Gradient Descent and conventional Coordinate Descent on the Wolfe Dual form of the SVM are both effective in this technique, with maximum accuracy.

Rafique et al. [2019] shows how to categorize malware families using a DL-based malware detection (DLMD) strategy based on static methods. The proposed DLMD approach classifies malware families by using both byte and ASM files for feature engineering. First, two distinct Deep Convolutional Neural Networks are used to extract features from byte file. Then, crucial and discriminative opcode features are chosen by using a wrapper-based method and the Support Vector Machine (SVM) works as a classifier. His plan is to create a hybrid feature space by integrating several feature spaces in order to overcome the shortcomings of each feature space and lower the chances of missing malware. Finally, Rafique et al. [2019] trained a Multilayer Perceptron on the hybrid feature space to classify all various malware families.

To get around state of the art detection technologies, modern malware develops several detection avoidance tactics. Combining image processing and ML approaches to classify and detect malware is an emerging trend to address this issue. Existing research on this subject, on the other hand, only uses simple image transformation methods, which limits the detection's accuracy. Vu et al. [2019] presents a novel way of classifying malware using a deep network on images transformed from binary samples. To turn binary executable into color images that express the binary data's semantics, Vu et al. [2019] offers a novel image transformation method which is a unique hybrid image modification method. Then he creates a Deep Convolutional Neural Networks that is used to train the images, which then classifies the test inputs into benign or malignant categories.

Huang et al. [2019] offered a neural network architecture consisting of an embedder, an encoder, and a filter for learning malware representations from characteristic execution sequences in order to classify malwares. The embedder captures relationships inside a single API request and between subsequent API calls in an execution trace using the state of the art embedding modules, BERT and Sent2Vec. The encoder uses gated recurrent units (GRU) to maintain the ordinal position of API calls and a self-attention method to compare intra-relations between distinct API call positions. To construct the malware representation, the filter detects representative API calls. To determine the impact of various framework components, he performs large-scale experiments. The results show that his proposed framework outperforms the baselines, and that using Sent2Vec to learn entire API call embeddings and GRU to explicitly preserve ordinal information yields more information and, as a result, considerable gains. In addition, the suggested method effectively classifies new harmful execution traces by comparing them to previously gathered families.

Using the function call graphs of x86 assembly instructions, Dalton et al. [2020] present a deep learning approach for identifying malware families. Although there has been previous work on static call graph analysis, very little of it has involved applying current, principled feature learning techniques to the problem. Dalton et al. [2020] offers a method that uses an executable's function call graph to get function representations via a recurrent neural network (RNN) auto-encoder that maps x86 instruction sequences into dense, latent vectors. These function embeddings are then represented in a network as vertices, with edges representing call dependencies. Capturing rich, node-level representations as well as global, topological aspects of an executable file increases malware family detection rates significantly and leads to a more principled approach to the problem that avoids arduous feature engineering and domain expertise.

According to Kwan [2020] malware detection at the processor microarchitecture level has lately emerged as a possible method for improving computer system security. ML algorithms are used in security mechanisms such as hardware-based malware detection to classify and detect malware using information from Hardware Performance Counters (HPCs). Microarchitectural data extracted from Hardware Performance Counters (HPCs), which contain behavioral data about a software application, is given to the ML classifiers. These HPCs are collected in real-time to mimic the behavior of the program. Because the number of HPCs per processor is restricted, many strategies use feature reduction to decrease the number of HPCs to the most important attributes.

2.2 State of the art techniques to overcome CF in CL

CL is a notion that allows us to learn a model for a large number of tasks in a sequential manner without losing what we learned from the previous tasks, even if the data from the previous tasks is no longer available when we're training new ones.

As models retrain with the data and data is changing over time so models also are changing always. Learning continually from a stream of data causes updating the model to maintain its performance. At this point, we have a problem with CF, which is when we forget what we've learned previously while learning new information. CF occurs in DNN mostly as a result of deeper layers adapting to new tasks. According to Zenke et al. [2017], solutions for CF have been addressed in three aspects:

1. Architectural techniques
2. Functional techniques
3. Structural techniques

In architectural techniques, we change the network's architecture to eliminate task interference without changing the objective function. In the architectural approach, we could see past solutions regarding the prevention of CF in these works: The most basic kind of architectural regularization is the freezing of particular network weights such that they remain constant [Razavian et al., 2014]. A more relaxed method lowers the learning rate for layers that are shared with the original task while fine-tuning to avoid drastic changes in the parameters [Donahue et al., 2013]. Furthermore, utilizing dropout to inject noise to sparsity gradients enhances performance [Goodfellow et al., 2013]. Rusu et al. [2016] advocated more radical architectural alterations, in which the whole network for the previous task is replicated and enhanced with new characteristics while solving a new task. in this paper, Progressive Neural Networks have been proposed as a simple method of increasing the number of tasks in a network. The goal is to create new layers that are related to the layers that have already been taught. The new layers can be trained on the new task, but the previous layers will not be changed. this solution completely eliminates the possibility of forgetting about previous tasks, although it does increase the architectural complexity as the number of tasks increases. Yoon et al. [2018] proposed a Dynamically Expandable Network. In fact, by exploiting task relatedness, the Dynamically Expandable Network retrains some of the network's parameters, while also increasing the number of parameters when necessary to account for new knowledge required from learning new tasks.

Another solution to counteract the CF effect is rehearsal strategies that rely on the storage of a subset of earlier data. According to Li et al. [2018], the network is updated with all data from the current task, as well as a selection of data from past tasks, in exemplar rehearsal approaches. It's still tough to come up with a general process for deciding which samples to keep and which to discard. The best option may be determined by the data or the learning algorithm. Rebuffi et al. [2017] proposed applying a distillation loss to exemplar predictions and a cross-entropy loss to new data predictions. Douillard et al. [2020] recommended using the distillation loss not only on the output but also on each network layer. In the single-class CL scenario, this modification looked to be quite efficient. End-to-end CL proposed by Castro et al. [2018] is similar to the idea proposed by Rebuffi et al. [2017] but Castro demonstrated that a classification layer could be used with a cross-entropy loss if we do training the network on each task as he did. Castro initially updates the network with all stored old data and all the new data, then does a second training phase with the old data and a fraction of the new data to ensure that old and new data are balanced. He also considered a small learning rate for this fine-tuning.

Also, the use of generative models to create artificial samples of previous data to update a network on a new task, a process known as pseudo-rehearsal, is a promising research avenue for CL. Generative Adversarial Networks (GANs) is a popular generative model for images, and they've been utilized for CL of image datasets for a long time. The goal is to train a GAN and a classification network at the same time. The GAN is used to produce labeled samples of old data that will be added to the fresh data when given a new task to learn. According to Shin et al. [2017] the developers of Deep Generative Replay (DGR) and Parameter Generation and Model Adaptation (PGMA) suggested methods for updating a GAN in real-time, however, their results were confined to simple datasets like MNIST, SVHN, and CIFAR-10. In a recent study, Chaudhry et al. [2019] show how to fine-tune samples from previous tasks while learning new ones using memory buffers.

Mi et al. [2021] developed Memory-based Hebbian Parameter Adaptation (Hebb) as a unified framework to address the two key concerns (CF and sample efficiency). Hebb augments a normal neural classifier with a continuously updated memory module to store representations of earlier data to prevent CF. Mi et al. [2021] introduced a parameter adaption approach based on the well-known Hebbian theory Hebb [1949], which wires the output network's parameters directly with similar representations recovered from memory to increase sample efficiency.

Regarding replay-based methods, which maintain and retrain on a limited buffer of data selected across tasks. Tiwari et al. [2021] proposed Gradient Coreset Replay (GCR) which is a revolutionary technique for replay buffer selection and updating based on a well-developed optimization criterion. He chose and kept a 'Coreset' that roughly approximates the gradient of all the data so far in terms of current model parameters. Han et al. [2022] introduced a novel framework called Bilevel Online Deep Learning (BODL), which combines bilevel optimization and an online ensemble classifier. He employed an ensemble classifier in the BODL approach, which builds several base classifiers from the output of different hidden layers in a DNN. The important weights of the base classifiers are updated in real-time using the exponential gradient descent method. In addition, he used a similar constraint to solve the online ensemble framework's convergence problem. Then, to monitor the change in the data probability distribution, an effective concept drift detection system based on the classifier's error rate is created. When concept drift is discovered, the BODL technique can use bilevel optimization to adaptively update the model parameters.

Hong and Pavlic [2022] developed a new neural network design that was inspired by neuromodulation in biological nervous systems. Neuromodulation is a biological mechanism that modulates and fine-tunes synaptic dynamics to match the behavioral context in real-time. Hong and Pavlic [2022] developed a single hidden-layer network that learns only a short context vector for each task (task-specific parameters) that neuromodulates unchanging, randomized weights (meta parameters) that convert the input. When task boundaries are given, he demonstrated that this approach may completely avoid CF while dramatically lowering the number of learnable parameters when compared to other context-vector-based approaches. Furthermore, he proved that the model can be adapted into a framework to execute CL without knowledge of task boundaries by integrating it with a simple meta-learning strategy for inferring task identity.

We add a regularization term to the objective in functional techniques to penalize changes in the neural network's input-output function. In the functional approach, we could see past solutions regarding the prevention of CF in these works: Li and Hoiem [2106] use a sort of knowledge distillation to encourage the predictions of the prior task's network and the current network to be similar when applied to data from the new task. Also, Jung et al. [2016] alternatively regularize the L2 distance between the final hidden activations instead of the knowledge distillation penalty. According to Zenke et al. [2017] both of these regularization approaches try to keep features of the old task's input-output mapping by storing or computing additional activations using the old task's parameters. This makes the functional approach to CF computationally expensive since it necessitates a forward pass over the old task's network for each new data piece.

Fini et al. [2020] proposed a new issue called Memory-Constrained Online Continual Learning (MC-OCL), in which strict memory limitations are imposed during the training process. Specifically, we want to keep the network's functionality while minimizing memory overhead. This means that the network discards all irrelevant data when making inferences. Fini et al. [2020] proposed that a memory-efficient OCL technique should meet the following two requirements: (1) No information other than the network itself should be sent from a generic time step t to time step t + 1; (2) no memory should be allocated for network extensions or devoted as auxiliary networks. Note that constraint (1) doesn't only means that each batch is treated separately; it also implies that information isn't passed from one task to the next. The proposed limits are especially important for application scenarios when the network is deployed on devices with limited memory or where prior images cannot be saved due to privacy concerns. So, Fini developed Batch-level Distillation (BLD) as a solution to the MC-OCL problem, in which distillation information is re-generated at each time step without breaking constraint (1). His strategy is divided into two distinct parts. The data from the current batch is used in the first stage, the warm-up, to execute a first gradient descent step, minimizing the cross-entropy loss over the new task classifier. The old task classifiers' predictions are saved in a probability bank that is used in the second stage, known as joint training. Both the distillation and the new task learning are carried out in this stage, using a dynamic weighting method based on the gradient norm obtained in the warm-up stage.

Penalties on the parameters are used in structural regularization to urge them to stay near to the parameters for the previous task. In the structural approach, we could see past solutions regarding the prevention of CF in these works: Kirkpatrick et al. [2017] suggested elastic weight consolidation (EWC), which is a quadratic penalty on the difference between the new and old task parameters. Assume we have a network that is currently trained on task A, and we want to train it on task B. The objective is to identify weights that are crucial to task A and then penalize updating to those weights more than non-important weights. Its means that changes in parameters that are critical for preserving performance on previous tasks should be penalized. Zenke et al. [2017] over the old parameters on the old task, Kirkpatrick employed a diagonal weighting proportional to the diagonal of the Fisher information metrics. The exact computation of the Fisher diagonal necessitates summing over all possible output labels, and so has a complexity proportional to the number of outputs. This restricts the use of this method in low-dimensional output spaces. Based on this reason a similar approach was proposed by Zenke et al. [2017], although their method computes per-synapse consolidation online and incorporates the complete learning trajectory rather than the final parameter value.

The subject of CF in DNNs has been addressed in a number of previous papers. In this part, we concentrate on quadratic regularization-based methods because they are more relevant to our study.

We have four quadratic regularization-based methods. The importance of parameters is defined on them as follows:

1. Elastic Weight Consolidation (EWC): It was created by Kirkpatrick et al. [2017] as a quadratic regularization approach that uses the diagonal of the Fisher information matrix to quantify the relevance of model parameters. Indeed it uses the Fisher information matrix to estimate the rigidity or flexibility of each parameter and applies its relevance to the prior tasks to update the model.

2. Synaptic Intelligence (SI): It is a concept introduced by Zenke et al. [2017], which defines the relevance of a parameter as its contribution to the reduction in the loss for past tasks. Indeed, it determines the importance of each parameter by calculating its contribution to the change in total loss from time 0 to T.

3. Memory Aware Synapses (MAS): It is described by Aljundi et al. [2017], in which the relevance of a parameter is defined as the sensitivity of model output to changes in that parameter. Indeed, it suggests relying on the last layer's sensitivity.

4. Reimannian Walk (RWalk): According to Chaudhry et al. [2018], the relevance of a parameter is defined as the sum of the diagonal of the Fisher information matrix and the contribution of that parameter to loss reduction for previous tasks.

According to Lubana et al. [2021] regarding quadratic regularization-based methods for mitigating CF, we could say that changes in model parameters that are critical for sustaining performance on previously learned tasks are penalized by these methods. They avoid forgetting by interpolating current values of model parameters with their values at the end of the preceding task's training. So, the effective learning rate of parameters that are more critical for maintaining performance on previously learned tasks is reduced by this interpolation technique. Indeed, they alter the parameter's "effective" learning rate. Parameters that are more important to previously acquired tasks have their learning rate reduced, making it impossible for them to change; on the other hand, parameters that are less important are permitted to change relatively easily, allowing the model to accommodate new tasks. Since the weights used in parameter interpolation are determined by the product of the learning rate (η), the regularization constant (λ), and the parameter's importance ($\alpha^{(k)}$), the efficacy of a quadratic regularizer is largely determined by these three factors.

2.3 State of the art strategies for task splitting in CL

To develop incrementally learning systems that learn about more and more concepts over time from a stream of data, we must simulate incrementally training strategies, which means that only the training data for a small number of classes should be present at the same time, and new classes can be added progressively.

Based on these papers [Zenke et al., 2017], [Goodfellow et al., 2013], [Yoon et al., 2018], [Chaudhry et al., 2019], [Aljundi et al., 2017], [Lubana et al., 2021], [Pham et al., 2022] that are related to overcoming CF in CL, in the experimental setup, they defined some tasks by dividing their datasets somehow (randomly or sequentially). Indeed each task contains some classes. They are training their model on each task one by one and sequentially. In some cases, these tasks are completely different, and in some cases, they are overlapped.

For example in a study on Continual Attentive Fusion for Incremental Learning in Semantic Segmentation by Yang et al. [2022], they consider two datasets in their experiments. The training and validation sets of the PASCAL-VOC 2012 dataset contain 10,582 and 1,449 samples, respectively. Pixels can be assigned to one of twenty-one different classes (20 plus the background). They distinguish between two types of experimental setups: disjunct and overlapping. The disjoint setup requires that the new set is distinct from the preceding ones, i.e. $(\bigcup_{j=0,\ldots,k-1} D_j \bigcap D_k = \emptyset)$. In the overlapping mode, each training step includes all images with at least one pixel of a novel class, regardless of whether or not other classes are present. It's vital to note that in this scenario, training images might contain pixels from classes that are unseen (thus labeled as background). This is a more realistic setting because it does not limit the objects that can be seen in the images. They conduct three experiments involving the addition of one class (19-1), five classes at once (15-5), and five classes added in alphabetical order one by one (15-1).

Also regarding the CL in the area of malware detection and classification, we list these two papers' experimental strategies in more detail:

First, In the paper regarding Support Vector Machine Based on CL for Malware Detection by Zhuang et al. [2015], The 20,000 samples are separated into 9 sections to apply the SVMIL method for CL: $D = D_0, D_1, D_2, ..., D_8$. Using D_0 as the beginning data set, train D_0 first to obtain the initial classification model, and then train the remaining 8 sample sets one by one through the CL approach.

And second, In the paper regarding CL for MC in Small Datasets by Li et al. [2020], The experimental data are from VirusShare (`https://virusshare.com/`) and VX Heavens (`https://83.133.184.251/virensimulation.org/index.html`). They chose four data subsets randomly from VX Heavens, naming them S1 to S4, and two data subsets from VirusShare, naming them S5 and S6. Each data subset contains 20 malware classes, with 600 malware samples in each class. There is no overlap between any two data subsets. They divided 600 malware samples into 500 training samples and 100 testing samples. They used all training samples from the first 5 classes to train a source model and the other classes as new data from new classes, and then the first 50 samples from all classes to train a source model and the other samples as new data from old classes to train CL.

In a recent study on Few-Shot Class-Incremental Learning by Sampling Multi-Phase Tasks by Zhou et al. [2022], nonoverlapping classes arrive sequentially, and it needs to build a classifier for all the classes incrementally. In the base task, there are enough training images to generate the initial model and limited images in the following tasks. After the learning process of each task, the model is evaluated among all seen classes. They use CIFAR100 as one of their experimental datasets. There are 100 classes in CIFAR100, comprising 60,000 images. The dataset splits of, 60 classes are used as base classes, and the rest 40 classes are organized as incoming new classes. They then divide these classes into eight incremental sessions; each contains a 5-way 5-shot incremental task.

Chapter 3

Goals of Thesis

In this study, we examine the performance of DL, which is one of the advanced algorithms of ML in the field of MC, and since our input data is image, we use CNN as our model. Our first and most important question is will DL works in dynamic situations with the same performance as it already worked in static situations for MC or not? Can our model have the previous accuracy by observing new samples and classes, or will it suffer CF?

Then we will implement and compare two state of the art CL techniques (ewc, explict ewc) to improve the performance of our CNN model in dynamic situations. We will apply explicit ewc as a regularizer which is based on a quadratic regularizer (ewc). In order to clarify the performance of this solution, at each step, we will compare the results of the training with explicit ewc (effective regularizer) with the results of the training with ewc (base regularizer) and the results of the training with sgd (without regularizer). Next, we will check whether the distribution of data in our dataset (balanced or imbalanced) has an effect on the performance of explicit ewc or not.

Since today's one of the two types of single head setup or multihead setup is mainly considered in these kind of studies, our next step will be to examine whether the explicit ewc can have the same performance in single head setup as it had in multihead setup. As the real conditions in the dynamic scenarios are closer to single head setup, the continuation of our studies will be focus mostly in this setup.

Then we will run an experiment to freeze the weights of a certain convolutional layer in the CNN to check the influence of that layer on CF. We specifically freeze the layers from the lowest layer which are the closest layers to the input, to a specified layer after training in task0, and train only the remaining layers in subsequent tasks.

In the next step, we will examine the impact of assigning majority classes and minority classes to one or two specific tasks on the performance of explicit ewc, ewc and sgd. To check what happens in the performance of each of them when a particular task takes a majority or minority class compared to when that class does not take a majority or minority class.

Then, we will go further in detail and examine why the explicit ewc performs much better in the multihead setup compared to the single head setup. Could this be due to the fact that in single head setup, explicit ewc could not do the interpolation between the current and previous values of the model parameters very well, or there may be another reason for this? Finally, we will check whether applying replay along with explicit ewc will improve the performance of explicit ewc in single head setup and if its performance improves, what could be the reason.

We will also check the effect of applying replay along with ewc and sgd in this experiment.

Chapter 4

Methodology

4.1 Malware classifier

4.1.1 Features

Because we implemented an effective state of the art method for classifying malware using quadratic regularization in an CL scenario on a CNN that uses image processing techniques, for our dataset, we used malware binaries that are visualized as grayscale images, with the observation that the images belonging to the same malware family appear very similar in layout and texture for all malware families. In fact, the malware executable binary files are converted into images by considering the binaries sequence as vectors of bytes to represent grayscale PNG image pixels.

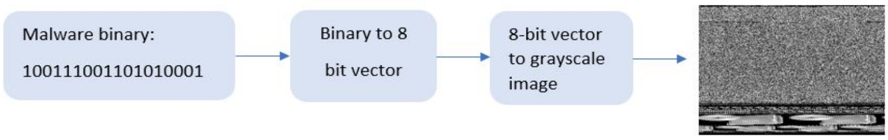

Figure 4.1. Malware binary to gray scale image

4.1.2 Classifier

According to Nataraj et al. [2011] traditional malware analysis methods involve extracting binary signatures from the malware, which serve as a fingerprint. Every year, the number of new signatures increases exponentially due to the rapid proliferation of malware. Static code analysis and dynamic code analysis are two basic methods for studying malware. The static analysis looks for malicious patterns by disassembling the code and studying the executable's control flow. Dynamic analysis works by running the code in a virtual environment and generating a behavioral report that characterizes the executable based on the execution trace. Regarding the advantages and disadvantages of both strategies, we can say Static analysis provides the most comprehensive coverage, but it is usually hampered by code obfuscation. Also in Static analysis, before the analysis, the executable must be unpacked and decrypted, and even then, difficulties of intractable complexity can stymie the process. On the other side, Dynamic analysis is more efficient since it does not require the unpacking or decryption of the executable. But it is, time and resource expensive, so it is posing scaling concerns. Furthermore, because the environment does not meet the triggering conditions, some malicious behaviors might be unobserved. A malware executable can be represented as a binary string of zeros and ones on a larger scale. This vector can be transformed into a matrix and viewed as an image. At this point, we could see a lot of visual similarities in image texture and layout between malwares from the same family. As a result, unlike Static or Dynamic analysis, using CNN as a classifier does not require disassembly or execution.

So based on the above clarification we think of MC as an image classification challenge by using these grayscale images. Figure 4.2 shows samples of malware belonging to three families of malware. As can be seen in the figure, the samples belonging to each family are similar to each other and each family is different from the other family in appearance.

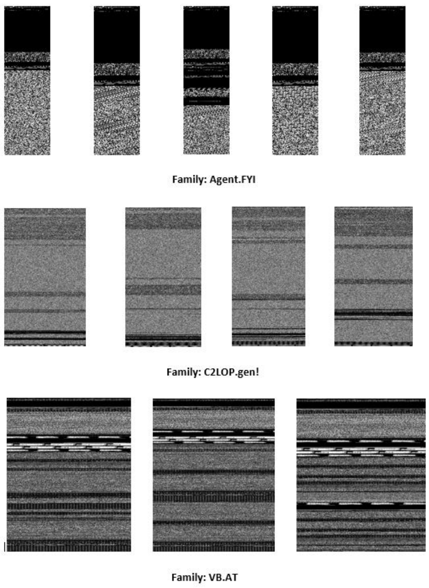

Figure 4.2. Visualization of some malware samples from three families

One of the most widely used DL algorithms is CNN. CNN is mostly used to extract features in order to do object recognition in images. It means we do not need to do manual feature extraction because it learns directly from images and could classify images based on that. In general to make a CNN model, we push three types of layers on top of each other. These layers are the convolutional layer, pooling layer, and fully connected layer.

In our CNN model, two convolutional layers are followed by three fully connected layers for the classification of malware represented as grayscale images. The input has a shape of [64 * 64 * 3] : [width * height * depth] and the malware sample's anticipated class is determined by the network's output. Our model will be based on the following structure:

- Convolutional Layer: 30 filters, (3 * 3) kernel size
- Max Pooling Layer : (2 * 2) pool size

- Convolutional Layer: 15 filters, (3 * 3) kernel size
- Max Pooling Layer : (2 * 2) pool size
- Flatten Layer
- Dense/Fully Connected Layer: 128 neurons, Relu activation function
- Dense/Fully Connected Layer: 50 neurons, Relu activation function
- Dense/Fully Connected Layer: number of class neurons, Softmax activation function

Below is a visualization of our CNN model architecture.

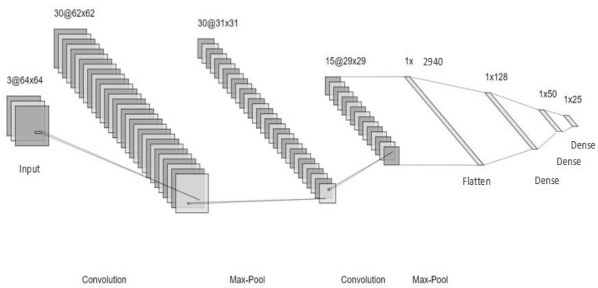

Figure 4.3. Model Architecture

After training and testing our model, we reach a final accuracy of 0.9486 in our imbalance dataset[1] and 0.9035 in our balance dataset[2](after removing the five classes of malware that have the most samples from the imbalance dataset).

4.2 Continual learning setup

We consider two mainly different setups for our experimental results in CL. First multihead classifier and second single head classifier. In a multihead setup, a multihead classifier can be used, with each task having its own output layer (head). The model uses the task label to select the proper head and compute the result during testing. In a Single head setup, our single head classifier does not partition the output space and the output neurons shared by all tasks.

[1]Code available at https://colab.research.google.com/drive/1F9Wfih-IjyPEidq8frhGQSI7Dy5ayhjk?usp=sharing
[2]Code available https://colab.research.google.com/drive/1lJEkG91wwqmFnhxU8u-5EjYkgoY1JKri?usp=sharing

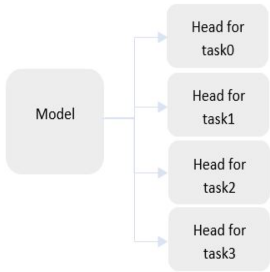

Figure 4.4. Multihead classifier setup

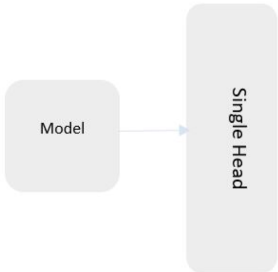

Figure 4.5. Single head classifier setup

4.3 Methods for continual learning

In this section, we will explain which of the methods we are using to overcome CF in CL.

4.3.1 SGD

The first one is the sgd which is the baseline where we are not trying to overcome CF. In sgd we used SGD (Stochastic Gradient Decent) as our optimizer and we train our model on our data without any regularization. Indeed the SGD is a simple version of the standard gradient descent method, which is a first-order optimization method for learning classifier weights. One of the drawbacks of standard gradient descent is that it takes a long time to reach the global minima. This is the problem that is solved by SGD, because in SGD the weight matrix W is updated for a smaller batch of data rather than updating at the end of each epoch.

4.3.2 EWC

The second one is ewc (Elastic Weight Consolidation) which is a quadratic regularizer and is our base regularizer. According to Kirkpatrick et al. [2017], ewc is a regularization strategy that attempts to control forgetting in CL by selectively constraining model parameters weights that are significant for prior tasks when training our model on new tasks. Indeed, it decreases the learning rate of parameters that are critical for previous tasks in order to maintain the model's performance on prior tasks, while allowing other parameters that aren't as relevant to them to adapt to new tasks.

4.3.3 EXPLICIT EWC

And third, explicit ewc which is the effective regularizer in this study. Indeed explicit ewc's strategy for overcoming the CF that happens in CL is based on a paper on quadratic regularization along with interpolation[Lubana et al., 2021]. In fact, quadratic regularizers decrease the learning rate of important parameters for previous tasks as new tasks are learned over time. As a result, it limits the change of critical parameters in order to maintain performance on previously acquired tasks and avoid CF. In this regard to limit the amount of change in model parameters at any given iteration, we interpolate between the current values of model parameters and their values at the end of the preceding task.
Based on the paper we have two strategies for applying this interpolation:

- Quadratic regularization with implicit interpolation
- Quadratic regularization with explicit interpolation

We will choose the second one for our implementation but to understand why we chose that one, first we also need to understand the first one.

Quadratic regularization with implicit interpolation:

First: We need to choose one of the quadratic regularizers (EWC, SI, MAS, Rwalk), and based on that we could calculate the importance score for regularizing the n^{th} task. This importance score was calculated during tasks 0 to $n-1$ and we show this with α_{n-1}. **Second**: We need to do interpolation to prevent CF at a given iteration so based on that we should update our model parameters based on the interpolation of current and previous values of them and also we need to consider the task-specific gradient.

$$\theta_n[i+1] = (1-\eta\lambda\alpha_{n-1}) \odot \theta_n[i] + (\eta\lambda\alpha_{n-1}) \odot \theta^*_{n-1} - \eta\nabla_{\theta_n[i]} L_{T_n}$$

Notations used in the above formula: θ_n denotes model parameters and L_{T_n} denotes a task-specific loss for the n^{th} task; θ^*_{n-1} denotes model parameters at the end of task $n-1$; $\theta_n[i]$ denotes model parameterization at the i^{th} iteration; η denotes the overall model learning rate, and λ denotes the regularization constant. We use $v^{(k)}$ to index a vector v at location k.

In quadratic regularizers generally, we assume importance scores for regularizing the n^{th} task (denoted α_{n-1}) are calculated during training of tasks 0 to $n-1$. These scores stay constant as the n^{th} task proceeds. Also, we know the total loss for learning the n^{th} task is as follows:

$$L = L_{T_n} + \frac{\lambda}{2} \sum \alpha^{(k)}_{n-1} (\theta^{(k)}_n - \theta^{*(k)}_{n-1})^2$$

In fact, in a CL, when learning a new task, the model parameters can be regularized to ensure arbitrary interpolations between current and previous values of parameters achieve low loss on both new and previously learned tasks.
According to Lubana et al. [2021], their analysis shows two primary pitfalls that limit the effectiveness of quadratic regularization with an implicit interpolation:

- Extrapolation, instead of interpolation, of parameters due to inappropriate hyperparameter when they did not consider enough choices for hyperparameter.

- Lower importance assignment to deeper layers. The standard importance definitions in quadratic regularizers can assign much lower importance to deeper layers of DNNs model. In fact, quadratic regularizers like EWC, MAS, SI, and RWALK will allow a model to change its deeper layers more readily than its recent layers because they assign to this deeper layer lower importance. As shown by Ramasesh et al. [2020], forgetting in DNNs is primarily caused by the adaptation of deeper layers to recent tasks. So, quadratic regularizers are unable to address the main source of CF in DNNs when they are considering lower importance scores for parameters in deeper layers.

So, based on the above explanation two primary conditions should be satisfied to ensure a quadratic regularizer performs well:

- The regularizer should function in the interpolation regime

- Importance scores be relatively constant across layers

These two conditions could be easily satisfied by breaking the quadratic regularizer update into two steps:

- $\theta_n^{'} = \theta_n[i] - \eta \nabla_{\theta_n[i]} L_{T_n}$

- $\theta_n[i+1] = (1 - R_j) \odot \theta_n^{'} + R_j \odot \theta_{n-1}^{*}$

And this is quadratic regularization with explicit interpolation.

Quadratic regularization with explicit interpolation:

While quadratic regularizers perform the task-specific update of parameters and the interpolation operation in a single step, the explicit interpolation version will break that update into two separate operations.

Implicit:

$$\theta_n[i+1] = (1 - \eta\lambda\alpha_{n-1}) \odot \theta_n[i] + (\eta\lambda\alpha_{n-1}) \odot \theta_{n-1}^{*} - \eta \nabla_{\theta_n[i]} L_{T_n}$$

Explicit:

$$\theta_n^{'} = \theta_n[i] - \eta \nabla_{\theta_n[i]} L_{T_n}$$
$$\theta_n[i+1] = (1 - R_j) \odot \theta_n^{'} + (R_j) \odot \theta_{n-1}^{*}$$

The vector R_j is defined to control the amount of interpolation at iteration j. To see why making interpolation in an explicit way is beneficial, consider that the amount of interpolation is dependent of the training hyperparameters like learning rate and regularization constant in implicit regularizer updates, in explicit regularizer updates but only the vector R_j controls the amount of interpolation. As a result, we can use this independence to define R_j in such a way that the desired requirements for effective quadratic regularization are always met. For this purpose, we define R_j as a ratio of a parameter's importance in past tasks to its importance in all tasks.

Assume that the n^{th} task is currently being learned. We used α_{n-1} to denote the importance of parameters according to tasks 0 to n-1 and α_{T_n} to denote the importance of parameters according to the n^{th} task, we define the relative importance of a parameter $\theta_n^{(k)}$ as follows:

$$R_j^{(k)} = \frac{\sqrt{\alpha_{n-1}^{(k)}}}{\sqrt{\alpha_{T_n}^{(k)}} + \sqrt{\alpha_{n-1}^{(k)}}}$$

Here, we could determine the importance of a parameter can be calculated by any quadratic regularizer. Based on this formulation, $R_j^{(k)}$ is in ranges from 0 to 1 for all parameters, so it will satisfy the first condition regarding preventing extrapolation (ensuring stable training). Because if $R_j^{(k)}$ is close to 1, then the k^{th} parameter is relatively more important for preserving performance on previous tasks than for the current task. In this case, the parameter's previous value is weighted more heavily to prevent its change. In the other scenario, if $R_j^{(k)}$ is close to 0, then the k^{th} parameter is relatively less important for previous tasks. In this case, the parameter's current value is weighted more heavily, allowing the parameter to adapt to a new task. Even if a parameter's absolute importance is low, still its relative importance can be high if it is unimportant to previous tasks. In this case, by using this definition for R_j, we can also avoid issues with assigning lower importance scores to deeper layers.

We used the below algorithm from [Lubana et al., 2021] to implement quadratic regularization with explicit interpolation steps in our code:

Input: parameterization θ_{n-1}^* and importance α_{n-1} after learning n-1 tasks, number of training interaction $\#iters$ and L_{Tn} for learning the n^{th} task.
Initialize:$\theta_n = \theta_{n-1}^*$; $\alpha_{T_n} = 0$.
for $j = 0$ **to** $\#iters - 1$ **do**
$\theta_n \rightarrow \theta_n - \eta \nabla_{\theta_n} L_{T_n}$ (Task-specific change)
Update α_{Tn}
Compute $R_j = \frac{\sqrt{\alpha_{n-1}}}{\sqrt{\alpha_{T_n}} + \sqrt{\alpha_{n-1}}}$ (Relative importance)
$\theta_n \rightarrow (1 - R_j) \odot \theta_n + (R_j) \odot \theta_{n-1}^*$ (Interpolation)
end for
Return:θ_n, α_n

Notation for the above algorithm:
θ_n denotes model parameters and L_{T_n} denotes a task-specific loss for the n^{th} task; θ_{n-1}^* denotes model parameters at the end of task $n-1$.

For the quadratic regularizer method, we choose EWC because it has the lowest average forgetting score based on the experiment that is shown in the paper by Lubana et al. [2021]. EWC regularizes the model parameter at each step with the model parameter at the previous step via the Fisher information matrix for the current task, which enables us to find a low loss for both tasks.

4.3.4 REPLAY

To overcome CF we have several approaches. Regularizers are used in the first category of techniques to limit the flexibility of the network during training on new tasks so that it remains stable on the prior tasks. The second method involves dynamically boosting the network's capacity to accept new tasks. Next one is replay based strategies which are replaying samples from the prior tasks while learning the new task . If these samples are real, we call this process rehearsal, Indeed rehearsal relies on the storage of a subset of earlier data to overcome CF. But if these samples are generated by generative mechanisms like GAN to be added to the fresh data we call this process synthesis. Indeed in synthesis we create artificial samples of previous data during the training of new data.

For example one strategy for applying rehearsal could be as follow: considering a buffer size of N, which means when we train on the second task we also have N samples from the first task during training time. in the next step when we train on the third task we have N/2 samples from the first task and N/2 samples from the second task during training time. So when we train on task k^{th} we will have N/k samples from each of the previous tasks.

4.4 Dataset characteristics

In training time it is not just the model or the classifier or the techniques that have impact on the accuracy of ML workflow. but the data itself also plays an important role. The simplest and the most important characteristic is whether the dataset is balanced or imbalance. And therefore we have decided to check performance of explicit ewc and ewc in both balanced and imbalanced datasets.

A balance dataset is a dataset in which the data is evenly distributed between classes. For example, suppose our database consists of two classes, A and B, and has a total of 1,000 samples. Now, if out of these 1000 samples, 460 samples belong to class A and 540 samples belong to class B, then we can say that the dataset is balanced. On the other hand, if the distribution of data in this dataset is such that 200 samples belong to class A and 800 samples belong to class B, we can say that this dataset is imbalanced. In fact, in an imbalanced database, the number of instances of one or two classes is significantly higher than in others.

Chapter 5

Experimental Evaluation

5.1 Dataset

Our imbalanced dataset was provided by Nataraj et al. [2011] and it consists of 9339 grayscale images of 25 different malware classes. Below diagram shows how the original dataset is distributed among all 25 malware classes. As you see this is an imbalanced dataset. So for our experiment, we consider two scenarios based on our dataset. First balance dataset that we earn it by removing 5 malware classes that have more samples (Allaple.L, Allaple.A, Yuner.A, VB.AT, Instantaccess). And second, is the imbalance dataset which is our original dataset.

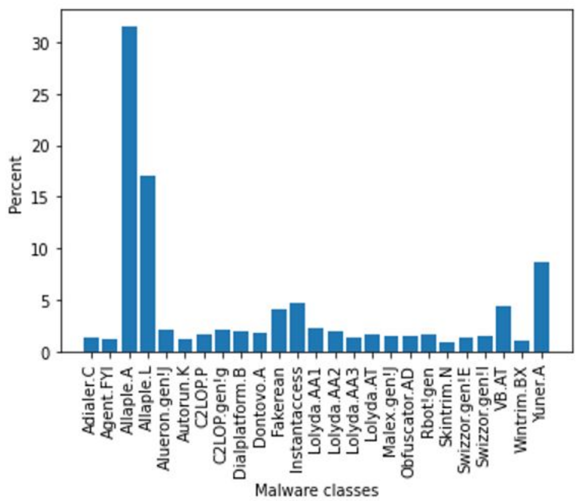

Figure 5.1. Dataset distribution

Table 5.1 gives more information.

Class Family	Malware Name
Worm	Allaple.L
Worm	Allaple.A
Worm	Yuner.A
Worm	VB.AT
Worm: AutoIT	Autorun.K
PWS	Lolyda.AA
PWS	Lolyda.AA 2
PWS	Lolyda.AA 3
PWS	Lolyda.AT
Trojan	C2Lop.P
Trojan	C2Lop.gen!g
Trojan	Skintrim.N
Trojan	Alueron.gen!J
Trojan	Malex.gen!J
TDownloader	Swizzot.gen!I
TDownloader	Swizzor.gen!E
Rogue	Fakerean
Dialer	Instantaccess
Dialer	Dialplatform.B
Dialer	Adialer.C
TDownloader	Wintrim.BX
TDownloader	Dontovo.A
TDownloader	Obfuscator.AD
Backdoor	Agent.FYI
Backdoor	Rbot!gen

Table 5.1. The name and family of malwares

Below you could see our confusion matrix for the original dataset. You can observe that although most of the malwares were well classified, but Autorun.K is always mistaken for Yuner.A. This is probably due to the fact that we have very few samples of Autorun.K in our training set. Moreover, Swizzor.gen!E is often mistaken with Swizzor.gen!l, which can be explained by the fact that they come from really close families and thus could have similarities in their code.

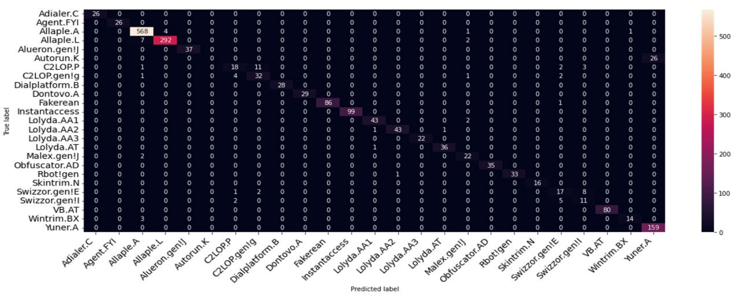

Figure 5.2. Confusion Matrix of the original dataset

5.2 Experimental setup

First, we consider two setups for all our experiments, one is a single head classifier that is more realistic and the other is a multihead classifier, to check whether the explicit ewc can have the same performance in single head setup as it has in multihead setup.

To check whether the distribution of data in our dataset (balanced or imbalanced) has an effect on the performance of explicit ewc or not, for each of our setups (single head and multihead classifier), we consider two scenarios for our dataset. The first is the balanced dataset which consists of 20 classes of malware and we make it by removing the 5 classes that have the most samples from the imbalance dataset. The second is the imbalance dataset which is our original dataset and has 25 classes of malware.

To simulate a dynamic situation and check whether applying explicit ewc could keep the accuracy of our model on the previous tasks when we train it on the new task or still it suffers from CF we train and test our model on some sequential tasks. Indeed in the balance dataset scenario, we define 4 tasks which each of them containing 5 classes of malware randomly[1]. And for the imbalance dataset, we consider 5 tasks that each of them containing 5 classes of malware randomly[2].

Our training setup is based on Lopez-Paz and Ranzato [2017]'s standard protocol, which states that for lifetime learning problems, the model should only observe a sample once. Indeed, for applying this online CL protocol over T tasks, the model only observes training samples selected from the current training task data at each step, and it must perform well on the testing data of all observed tasks so far. We do a hyperparameter on the first task to find the proper learning rate learning (it is 0.01)[3]. We train and test our model on the train split and test split of all tasks incrementally. The model will train with a batch size of 32 and the model is allowed to visit each sample only once.

In order to clarify the performance of explicit ewc which is our effective regularizer to overcome CF in CL, at each step, we will compare the results of the training with explicit ewc with the training with the other two methods: ewc and sgd. Indeed in our setup, we define three trainers. First sgd trainer that is our trainer without regularization (it just used SGD as an optimizer). Second, ewc trainer that is our trainer with base regularizer (EWC). And third, explicit ewc trainer that is our trainer with explicit quadratic regularizer.

In order to check the influence of freezing the weights of certain layers in our CNN model on CF, we define three training versions. So for each of our trainers (explicit ewc, ewc, sgd), we train and test our model in three versions. First version is normal fitting that is training our CNN model without any freezing. In the second version, we freeze the first convolutional layer and then we train and test our model. And for the third version, we freeze two convolutional layers in our model and then we train and test it. Here we freeze layers with the assumption that the features that have already been learned would generalize to new tasks as well.

In order to check what happens in the performance of explicit ewc when a particular task takes a majority or minority class compared to when that class does not take a majority or minority class, we choose two majority classes and assign them to task0. Then we choose two minority classes and assign them to task2 then we train and test the model in our imbalanced dataset. In the next step, we assign the two minority classes to task0 and the two majority classes to task2 to check what pattern we could find in the result of assigning majority-minority

[1] Code is available at https://colab.research.google.com/drive/13FGUj0xQIXOh08S7pVtB786QvCRmWYHk?usp=sharing

[2] Code is available at https://colab.research.google.com/drive/1A5nT8bMTLp0CeBnCfmaBmzkX5yzrh2GN?usp=sharing

[3] Code available at https://colab.research.google.com/drive/1lwRqzjylWOOLlnY9BzvMnYUv3Fm2LusfscrollTo=3xePq7BdIAG3

classes to a specific task (we do this experiment in both single head and multihead setups). In order to make sure that the probable pattern is not related to our dataset, then we make an artificial dataset from caltech101 that has 25 classes of images and is imbalanced and we repeat the previous experiment on that dataset (this experiment is only for single head setup).

Then, we go further in detail, and in order to check why the explicit ewc performs much better in the multihead setup compared to the single head setup, we calculate the average interpolation and average R for all tasks to see how R control the interpolation between the current and the previous value of model parameters in explicit ewc in single head setup.

Finally, in order to check whether applying Replay along with explicit ewc will improve the performance of explicit ewc in single head setup, we apply rehearsal along with explicit ewc. But for further investigation, we also apply rehearsal along with ewc and sgd to check how much effect it has on them. We also check how we affect the results by changing the buffer size in this experiment. In this experiment, we start with a buffer size of N, which means that during training time on task1, we also receive N samples from task0. We have N/2 samples from task0 and N/2 samples from task1 during training time in the next step when we train on task2. As a result, we'll get N/k samples from each of the prior tasks when we train on task(k). We get the results with three different numbers for N in this experiment (300, 600, 900).

5.3 Results and discussion

5.3.1 Results in multihead classifier setup

In this step, we have results of the first random balanced dataset (first iteration) for sgd (training without regularization - stochastic gradient descent is our optimizer), ewc(training with ewc regularizer - base quadratic regularizer), explicit ewc regularizer (we implement this regularizer base on the paper from Lubana et al. [2021]). For each of them, we have a table that contains average accuracy, average forgetting, and final accuracy.

In this experiment, we consider the epoch equal to 100, based on this we have these definitions for the below pieces of information in the tables.

- Average accuracy for each task is defined as the average test accuracy achieved on all epochs.

- Final accuracy for each task is defined as the test accuracy achieved on the last epoch.

- Average forgetting for each task is defined as the difference between the maximum and final accuracy achieved on it during all its epochs

These tables and graphs are related to the results of sdg, ewc, explicit ewc on one randomly balanced dataset splitting in one iteration:
Training with sgd (without regularizing):

sgd			
	Average accuracy	Average forgetting	Final accuracy
task0	0.7936	0.3684	0.6263
task1	0.7354	0.4702	0.5099
task2	0.9568	0.0938	0.9063
task3	0.9996	0	1

Table 5.2. Training with sgd - balanced dataset

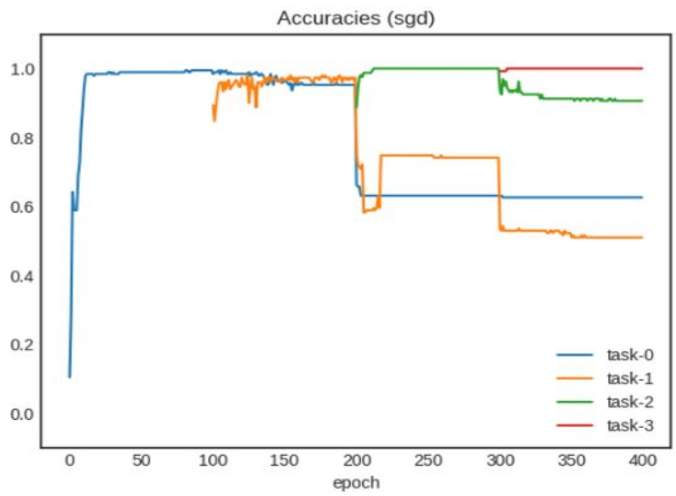

Figure 5.3. Training with sgd - balanced dataset

Regularizing with ewc:

ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.9320	0.0684	0.9263
task1	0.9302	0.0397	0.9404
task2	0.9818	0.0125	0.9875
task3	0.9929	0	1

Table 5.3. Training with ewc - balanced dataset

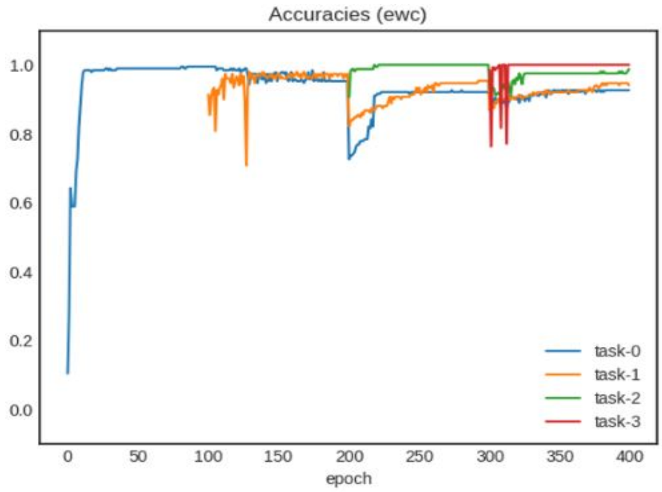

Figure 5.4. Training with ewc - balanced dataset

Regularizing with explicit ewc:

explicit ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.9822	0	0.9947
task1	0.9496	0.0265	0.9470
task2	0.9759	0.0063	0.9813
task3	0.9674	0	0.9924

Table 5.4. Training with explicit ewc - balanced dataset

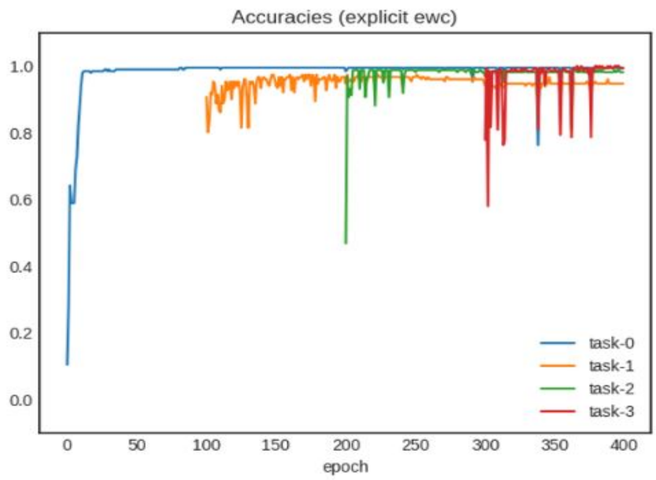

Figure 5.5. Training with explicit ewc - balanced dataset

These are the results of the first random imbalanced dataset (first iteration) for training with sgd, ewc, and explicit ewc regularizer in multihead classifier setup. For each of them, we have a table that contains average accuracy, average forgetting, and final accuracy.
Training with sgd (without regularizing):

sgd			
	Average accuracy	Average forgetting	Final accuracy
task0	0.9421	0.0656	0.9344
task1	0.9717	0.0518	0.9451
task2	0.7555	0.3707	0.6293
task3	0.8722	0.2970	0.6970
task4	0.9975	0.0000	1.0000

Table 5.5. Training with sgd - imbalanced dataset

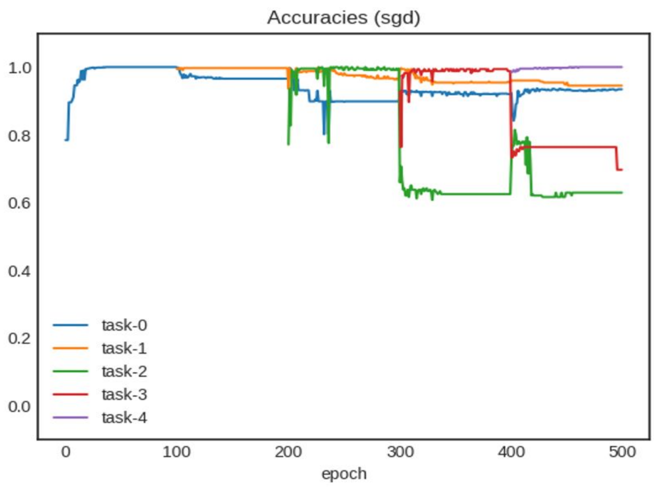

Figure 5.6. Training with sgd - imbalanced dataset

Regularizing with ewc:

ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.9567	0.0128	0.9872
task1	0.9644	0.0366	0.9604
task2	0.9487	0.0905	0.9095
task3	0.8797	0.1818	0.8182
task4	0.9969	0.0000	1.0000

Table 5.6. Training with ewc - imbalanced dataset

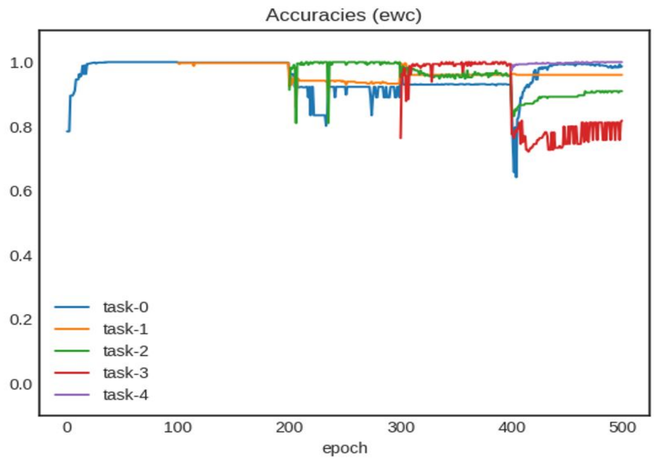

Figure 5.7. Training with ewc - imbalanced dataset

Regularizing with explicit ewc:

explicit ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.9918	0.0014	0.9986
task1	0.9519	0.0854	0.9116
task2	0.9738	0.0086	0.9914
task3	0.9631	0.0061	0.9879
task4	0.9912	0.0000	0.9932

Table 5.7. Training with explicit ewc - imbalanced dataset

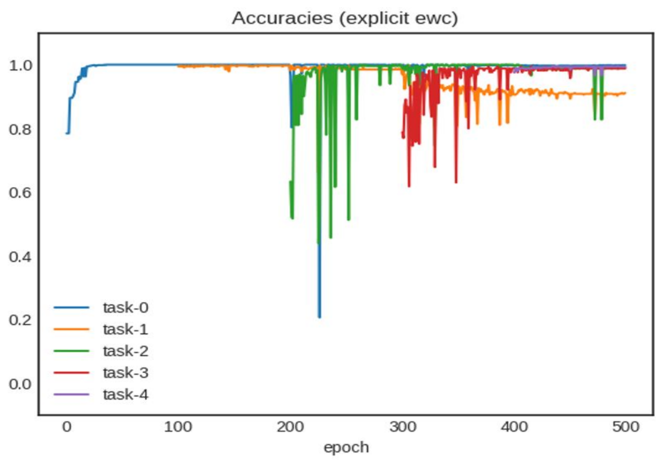

Figure 5.8. Training with explicit ewc - imbalanced dataset

Here we have a summary of average accuracy for all tasks in balanced and imbalanced dataset in three situation: normal fitting (training of our model without freezing any convolutional layer), V1 (training of our model when we freeze the first two convolutional layers), and V2 (training of our model when we freeze the first convolutional layer). We did each of these experiments six times on six randomly datasets so these results is average of those experiments. These are excel links regarding details of all the results for training and testing of our model on balanced[4] and imbalanced[5] datasets in this setup.
Here average accuracy means average accuracy of all tasks. And for improvement accuracy we apply this formula: [(explicit ewc average accuracy - sgd average accuracy)/sgd average accuracy]

multihead classifier-normal fitting-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.9788	0.934	0.7769
imbalance	0.9723	0.9186	0.8663

Table 5.8. Average accuracy in normal fitting-multihead classifier

multihead classifier-normal fitting	
improvement in avg acc(sgd to explicit ewc)	
balance	0.26
imbalance	0.12

Table 5.9. Improvement in average accuracy in normal fitting-multihead classifier

multihead classifier-V1-freezing the first 2 con layers-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.9555	0.9369	0.8074
imbalance	0.9795	0.9085	0.9092

Table 5.10. Average accuracy in V1-multihead classifier

multihead classifier-freezing the first 2 con layers	
improvement in avg acc(sgd to explicit ewc)	
balance	0.18
imbalance	0.08

Table 5.11. Improvement in average accuracy in V1-multihead classifier

4 Details at: https://docs.google.com/spreadsheets/d/102OaGHVeMC10YX5GMWNtcy6gujWRTgRqohk$_J ko498g/edit?usp = sharing$

5 Details at: https://docs.google.com/spreadsheets/d/1hd2liYhzWeEJMJBn6nGMQi$_R XacGDuKa6NFcvzMNlns/edit?usp = sharing$

multihead classifier-V2-freezing the first con layer-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.9646	0.9555	0.7787
imbalance	0.9558	0.9214	0.8889

Table 5.12. Average accuracy in V2-multihead classifier

multihead classifier-freezing the first con layer	
improvement in avg acc(sgd to explicit ewc)	
balance	0.24
imbalance	0.08

Table 5.13. Improvement in average accuracy in V2-multihead classifier

All the above results show us that for balanced and imbalance dataset always explicit ewc works better than ewc and ewc works better than sgd in all three situations. And this happens because in explicit ewc we interpolate between the recent and previous values of model parameters explicitly, to reduce CF in CL. Also we notice that the average accuracy of all tasks in training with sgd for the imbalance dataset is higher than the balanced dataset. But the average accuracy of all tasks in training with explicit ewc for the imbalance dataset is almost the same as the balanced dataset.

Results show us that freezing our convolutional layers in our CNN along we sgd training increases the average accuracy of all tasks in balanced and imbalance datasets. But we couldn't see any specific pattern in the results of ewc or explicit ewc along with freezing convolutional layers, maybe this happened because we apply regularization on them (because of lack of time we didn't do further analysis in this issue). Even when we freeze one or two of our convolutional layers, still applying explicit ewc always increase the average accuracy of all tasks in balanced and imbalance datasets.

5.3.2 Results in single head classifier setup

We will do exactly all the experiments that we did before in the multihead classifier, this time in the single head classifier. Here in below we have results of one time training with sgd, ewc, and explicit ewc in balanced dataset.

Training with sgd (without regularizing):

sgd			
	Average accuracy	Average forgetting	Final accuracy
task0	0.7770	0.5463	0.4537
task1	0.5599	0.8076	0.1924
task2	0.7681	0.4555	0.5445
task3	0.9999	0.0000	1.0000

Table 5.14. Training with sgd - balanced dataset

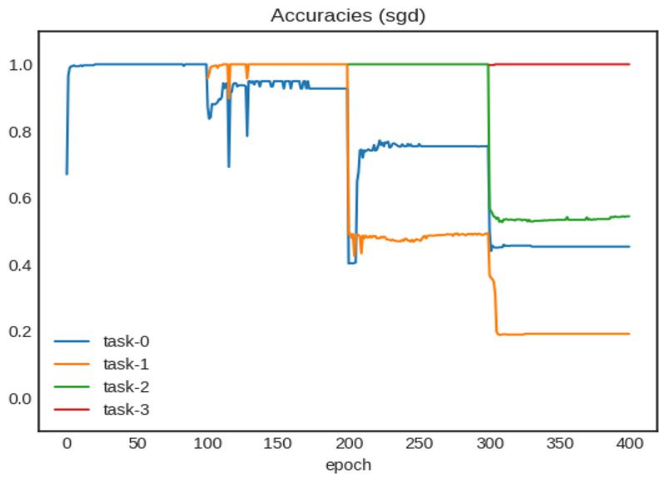

Figure 5.9. Training with sgd - balanced dataset

Regularizing with ewc:

ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.7838	0.4179	0.5821
task1	0.5817	0.6804	0.3196
task2	0.7402	0.5259	0.4741
task3	0.9998	0.0000	1.0000

Table 5.15. Training with ewc - balanced dataset

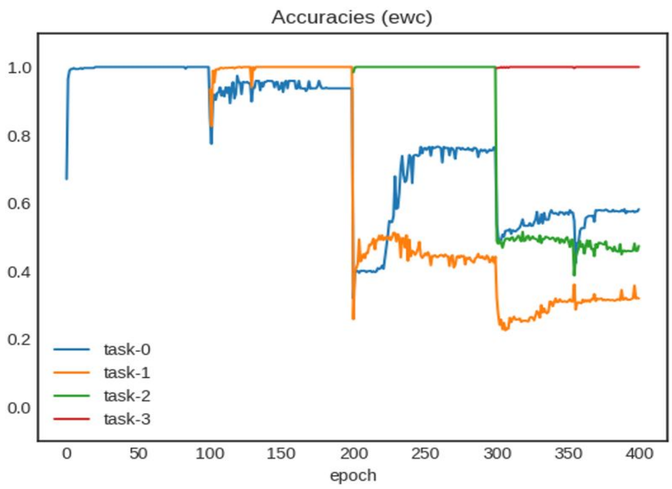

Figure 5.10. Training with ewc - balanced dataset

Regularizing with explicit ewc:

explicit ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.7185	0.5652	0.4348
task1	0.6046	0.8141	0.1859
task2	0.7413	0.5580	0.4420
task3	0.9852	0.0000	1.0000

Table 5.16. Training with explicit ewc - balanced dataset

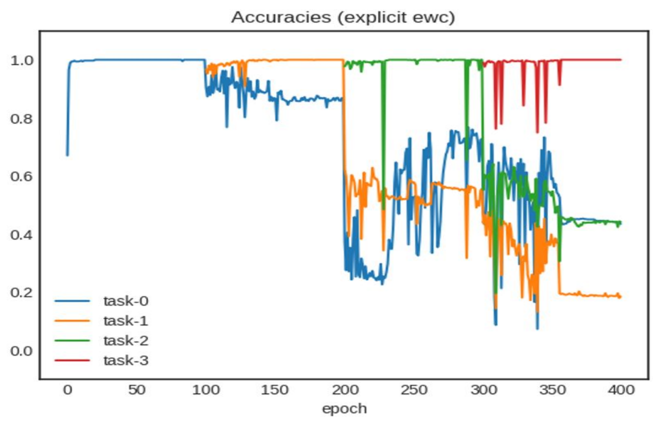

Figure 5.11. Training with explicit ewc - balanced dataset

Here these are the results of the first random imbalanced dataset (first iteration) for training with sgd, ewc, and explicit ewc regularizer in single head classifier setup.
Training with sgd (without regularizing):

sgd			
	Average accuracy	Average forgetting	Final accuracy
task0	0.2489	0.8947	0.1039
task1	0.3260	1.0000	0.0000
task2	0.5293	0.7708	0.2292
task3	0.6118	0.7844	0.2096
task4	0.9971	0.0000	1.0000

Table 5.17. Training with sgd - imbalanced dataset

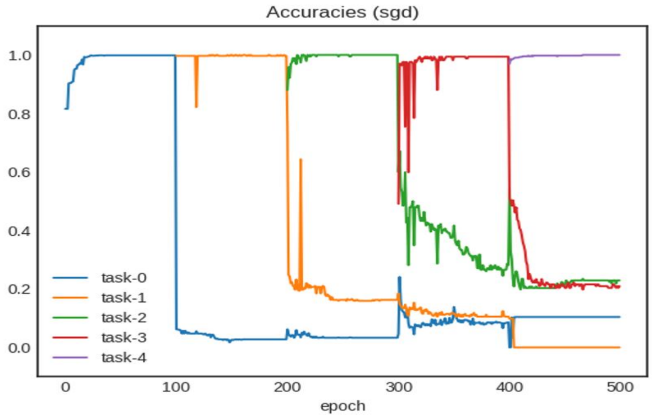

Figure 5.12. Training with sgd - imbalanced dataset

Regularizing with ewc:

ewc			
	Average accuracy	Average forgetting	Final accuracy
task0	0.2336	0.9697	0.0289
task1	0.3616	0.9013	0.0987
task2	0.5774	0.6510	0.3490
task3	0.6628	0.6587	0.3353
task4	0.9948	0.0000	0.9954

Table 5.18. Training with ewc - imbalanced dataset

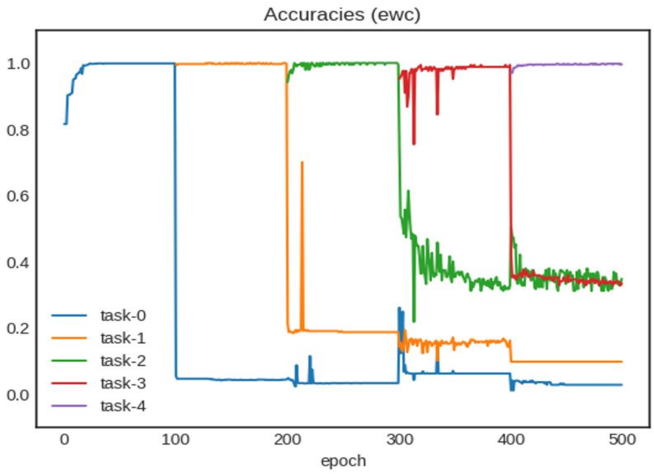

Figure 5.13. Training with ewc - imbalanced dataset

Regularizing with explicit ewc:

explicit ewc			
	Average accuracy	Average forgetting	Final accuracy
Task0	0.2763	0.8947	0.1039
task1	0.3863	0.9586	0.0414
task2	0.4727	0.8594	0.1406
task3	0.6823	0.5629	0.4192
task4	0.9736	0.0000	0.9931

Table 5.19. Training with explicit ewc - imbalanced dataset

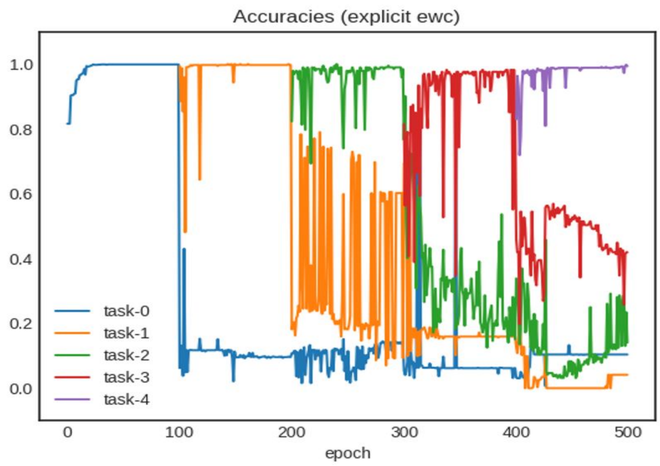

Figure 5.14. Training with explicit ewc - imbalanced dataset

And here in below we summarize all the results regarding the normal fitting, V1, and V2 for balanced and imbalanced dataset in single head classifier setup. These results are average of six times training a random splitting of our balanced and imbalance datasets on our model. These are excel links regarding all the results for training and testing our model on balanced dataset[6] and imbalanced dataset[7] in this setup.

single head classifier-normal fitting-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.5229	0.7211	0.5619
imbalance	0.3315	0.3383	0.3092

Table 5.20. Average accuracy in normal fitting-single head classifier

[6]Details at: https://docs.google.com/spreadsheets/d/12hntKdhMuVKFY6Zc7Y1Efo0iMCWLZkkzrTQe6fWh17c/edit?usp=sharing
[7]Details at: https://docs.google.com/spreadsheets/d/1MDpkWIKASK3PN3FitWnkCexqPkTopiN1zZ5MTMyuNpk/edit?usp=sharing

single head classifier-normal fitting	
improvement in avg acc(sgd to explicit ewc)	
balance	-0.07
imbalance	0.07

Table 5.21. Improvement in average accuracy in normal fitting-single head classifier

single head classifier-V1-freezing the first 2 con layers-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.5315	0.603	0.5664
imbalance	0.3329	0.3224	0.2998

Table 5.22. Average accuracy in V1-single head classifier

single head classifier-freezing the first 2 con layers	
improvement in avg acc(sgd to explicit ewc)	
balance	-0.06
imbalance	0.11

Table 5.23. Improvement in average accuracy in V1-single head classifier

single head classifier-V2-freezing the first con layer-avg acc			
dataset	explicit ewc	ewc	sgd
balance	0.5715	0.5754	0.5458
imbalance	0.346	0.3373	0.2991

Table 5.24. Average accuracy in V2-single head classifier

single head classifier-freezing the first con layer	
improvement in avg acc(sgd to explicit ewc)	
balance	0.05
imbalance	0.16

Table 5.25. Improvement in average accuracy in V2-single head classifier

By checking the results in normal fitting, we notice that in single head classifier setup, we could see that in balance dataset the accuracy of all tasks decreases 7% and in imbalance dataset the accuracy of all tasks increases 7% when we apply explicit ewc regularizer compare to training with sgd (training without regularizing). When we comparing results of ewc and explicit ewc in normal fitting, we see that for balance dataset ewc works much better than explicit ewc but they works same in imbalance dataset. We will discuss this issue in the next session in details.

Also results regarding the freezing first two convolutional layers shows us that, the performance of sgd almost doesn't change compare to normal fitting in balanced dataset and imbalanced datasets in single head setup. But when we check the result regarding the freezing first convolutional layer, we could see that the average accuracy of all tasks decreases in balanced dataset but it is almost the same as normal fitting in imbalanced dataset. Another point that we find in the results related to freezing first covolutional layer is that explicit ewc has higher average accuracy for all tasks compare to sgd in balanced and imbalance datasets in single head setup. But we could not see the same behavior from explicit ewc when we freeze first two convolutional layer. Finding the reason for the difference between the results related to these two setup needs more analysis and because of lack of time we will do it in the future.

5.3.3 Results in multihead VS single head classifier setup

By comparing the results we could see that a difference between results of explicit ewc in single head classifier and multihead classifie. Indeed in multihead classifier setup, average accuracy of all tasks for balance and imbalance dataset is almost the same (0.9788 for balance dataset and 0.9723 for imbalance dataset). We have 26% increase in average accuracy of all tasks in balance dataset and 12% increase in average accuracy of all tasks in imbalance dataset. But in single head classifier we could see that average accuracy of all tasks in balance dataset is 19% higher than imbalance dataset in training with explicit ewc. These results shows us that in multihead classifier explicit ewc works very well for balance and imbalance dataset. But in single head classifier explicit ewc doesn't work in balance dataset (we have 7% decrease in average accuracy of all tasks), and it also improve a little average accuracy of all tasks in imbalance dataset(7%).

By checking the results obtained from the balanced dataset and imbalance dataset in the single head classifier setup, we find that in the imbalance dataset, the achievement average accuracy of all tasks in all three types of training (sgd, ewc, explicit ewc) are very close to each other. Therefore, we conclude that explicit ewc has no effect on overcoming CF in normal fitting.

But the results from the balanced dataset show that explicit ewc performed worse than ewc and even sgd (training without regularization). A comparison of the results obtained from ewc and sgd shows that the results in ewc have improved by 28% compared to sgd. We know that in ewc interpolation between current and previous values of model parameters is performed in an implicit way.

$$\theta_n[i+1] = (1-\eta\lambda\alpha_{n-1}) \odot \theta_n[i] + (\eta\lambda\alpha_{n-1}) \odot \theta^*_{n-1} - \eta\nabla_{\theta_n[i]} L_{T_n}$$

On the other hand, we can see in the results related to explicit ewc and ewc, that the results in explicit ewc have decreased compared to ewc by 27%. This indicates that explicit ewc could not perform the interpolation well between the current and previous values of the model parameters in an explicit way to obtain a low loss for both the current and previous tasks.

As you know in interpolation in explicit format, we do task-specific gradient and interpolation between current and previous values separately and in two steps.

- $\theta'_n = \theta_n[i] - \eta\nabla_{\theta_n[i]} L_{T_n}$ (Task-specific gradient)
- $\theta_n[i+1] = (1-R_j) \odot \theta'_n + (R_j) \odot \theta^*_{n-1}$ (Interpolation)

Indeed, we define a vector R_j and by using this vector we control the amount of interpolation in each iteration j. This R_j is defined as a ratio of a parameter's importance in past tasks to its importance in all tasks.

$$R_j^{(k)} = \frac{\sqrt{\alpha_{n-1}^{(k)}}}{\sqrt{\alpha_{T_n}^{(k)}}+\sqrt{\alpha_{n-1}^{(k)}}}$$

For all parameters, $R_j^{(k)}$ is in the range of 0 to 1, to prevent extrapolation. Because if the k^{th} parameter is more important for preserving performance on prior tasks than for the current task so $R_j^{(k)}$ is close to 1. So, the previous value of the parameter is weighted more highly in this scenario to prevent it from changing. Furthermore, if $R_j^{(k)}$ is close to 0, the k^{th} parameter is less important for previous tasks. So, the current value of the parameter is weighted more heavily in this situation, allowing the parameter to adjust to the new task. Even if a parameter's absolute value is low, if it is unimportant to previous tasks, its relative importance can be high. We can avoid problems with assigning lower importance scores to deeper layers by using this definition for R_j.

Our hypothesis is that R_j could not control the interpolation well. To validate the hypothesis we calculate average of R and average of interpolation for each task (except task0 because task0 is the first task and we do not need to do interpolation on it). We calculate average R and average interpolation in explicit ewc trainer for three randomly splitting of our dataset in multihead setup and single head setup. Below is the results of this experiment:[8]

	Avaerage of R		Average of interpolation	
task number	single head	multihead	single head	multihead
task1	0.6280	0.6130	0.0012	0.0005
task2	0.7919	0.5884	0.0012	0.0010
task3	0.5805	0.4434	0.0016	0.0022
task4	0.3762	0.2273	0.0024	0.0027
average for all tasks	0.5942	0.4680	0.0016	0.0016

Table 5.26. Training with explicit ewc - average R & average interpolation - version 1

	Avaerage of R		Average of interpolation	
task number	single head	multihead	single head	multihead
task1	0.7322	0.6615	0.0027	0.0012
task2	0.7896	0.5652	0.0025	0.0016
task3	0.6947	0.426	0.0027	0.0027
task4	0.5866	0.4018	0.0031	0.0031
average for all tasks	0.7008	0.5136	0.0028	0.0022

Table 5.27. Training with explicit ewc - average R & average interpolation - version 2

[8]Details available at: https: //docs.google.com/spreadsheets/d/1Q-DDUbIIjUss9HRP2etiCHsXW4DENszKc-vX8MoDNDs/edit?usp=sharing

	Avaerage of R		Average of interpolation	
task number	single head	multihead	single head	multihead
task1	0.5736	0.6525	0.0031	0.0012
task2	0.8325	0.4563	0.0029	0.0016
task3	0.5686	0.3482	0.0034	0.0027
task4	0.3975	0.2809	0.0034	0.0031
average for all tasks	0.5931	0.4345	0.0032	0.0022

Table 5.28. Training with explicit ewc - average R & average interpolation - version 3

As the results show us, the value of R in all three versions of our dataset is greater for all tasks in a single head than in a multihead. Calculations show that in the first version there is a 0.12, in the second version there is a 0.18, and in the third version there is a 0.15 difference in the average of R for all tasks between single head and multihead. Regarding interpolation, it can be said that the average interpolation of all tasks in single head is almost close to each other. But when we look at the average interpolation of tasks in the multihead, we find that the more beginning tasks have less average interpolation than the more recent tasks.

Since our hypothesis is that R could not do interpolation well in single head setup, by comparing the results obtained from the average R and the average interpolation of each of the tasks in single head and multihead, it can be concluded that a greater R in single head, compared to multihead in each of the tasks, caused the amount of interpolation of all tasks in single head to be uniform (close to each other). In fact, in this case, all tasks (both beginning and recent) have been treated the same way. In our opinion, this has caused the explicit ewc to not be able to perform well in the single head.

But still, we don't know why is R different in single head and multihead setup which is could be the subject of further research which we will do in the future.

5.4 Further analysis & improvement in the results

5.4.1 More investigation in the imbalance dataset

We start to do more investigation on the performance of our trainers (sgd, ewc, explicit ewc) in our imbalance dataset in single head classifier setup and multihead classifier setup. Indeed we want to check what will happen if a certain task gets two of the majority classes (classes that have the most samples in the imbalance dataset) and another certain task gets two of the minority classes (classes that have the fewer samples in the imbalance dataset). So for this purpose, we choose Allaple.L and Allaple.A classes of malware as our majority classes and assign them to task2. Then we choose Skintrim.N and Wintrim.BX classes of malware as our minority classes and assign them to task0.

As we know for the imbalance dataset we have five tasks and each of them contains 5 classes of malware. So except for these four classes that are assigned manually to task0 and task2, other classes are assigned randomly to each task. In this experiment, we train and test three different scenarios (training with sgd, training with ewc, and training with explicit ewc) on our tasks incrementally. To get a more realistic result, we did this experiment three times and calculated all tasks' average accuracy.

Then we bring a change in our experiment and assign two minority classes to task2 and two

majority classes to task0 and again we train and test our three scenarios(sgd, ewc, explicit ewc) on our data incrementally.
Here are the results regarding multihead setup:

sgd _ avg of accuracies-multihead setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.7415**	**0.7187**
task1	0.4065	0.7877
task2	**0.7573**	**0.7936**
task3	0.7354	0.8203
task4	0.9976	0.9931
average accuracy	0.7277	0.8227

Table 5.29. Training with sgd - majorityminority - malware dataset - multihead setup

ewc _ avg of accuracies-multihead setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.8270**	**0.9194**
task1	0.6907	0.9769
task2	**0.6953**	**0.8955**
task3	0.5857	0.9232
task4	0.9976	0.9963
average accuracy	0.7593	0.9423

Table 5.30. Training with ewc - majorityminority - malware dataset - multihead setup

explicit ewc _ avg of accuracies-multihead setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.9241**	**0.9863**
task1	0.5689	0.8957
task2	**0.9715**	**0.9815**
task3	0.9847	0.9939
task4	0.9884	0.9912
average accuracy	0.8875	0.9697

Table 5.31. Training with explicit ewc-majorityminority-malware dataset - multihead setup

Here are the results regarding single head setup:

sgd _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.3326**	**0.0898**
task1	0.4884	0.2558
task2	**0.0397**	**0.3587**
task3	0.4027	0.4071
task4	0.9976	0.9940
average accuracy	0.4522	0.4211

Table 5.32. Training with sgd - majorityminority - malware dataset - single head setup

ewc _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.3293**	**0.0978**
task1	0.5171	0.2548
task2	**0.1060**	**0.4107**
task3	0.3535	0.4033
task4	0.9964	0.9950
average accuracy	0.4605	0.4323

Table 5.33. Training with ewc - majorityminority - malware dataset - single head setup

explicit ewc _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.3925**	**0.2542**
task1	0.3403	0.1496
task2	**0.1995**	**0.4459**
task3	0.4493	0.3285
task4	0.9914	0.9774
average accuracy	0.4746	0.4311

Table 5.34. Training with explicit ewc-majorityminority-malware dataset-single head setup

Examination of the results obtained in the single head setup shows that in three types of training (sgd, ewc, explicit ewc), the accuracy for a task that contains 2 minority classes is higher than the task that contains 2 majority classes. By comparing the average of all tasks, we find that in all three types of training (sgd, ewc, explicit ewc), the average accuracy of all tasks in the scenario where task0 contains 2 minority classes is higher than when the task0 contains 2 majority classes. From the results related to single head setup, it can be concluded that assigning minority-containing classes in more beginning tasks increases the overall average accuracy score in the single head setup.

By checking the results obtained from the multihead setup, no specific pattern can be found. It can only be said that the average accuracy of all tasks in all three types of training (sgd, ewc, explicit ewc) in the case where task0 contains 2 majority classes is higher than when the task0 contains 2 minority classes. From the results related to multihead setup, it can be concluded that assigning majority-containing classes in more beginning tasks increases the overall average accuracy score in the multihead setup.[9]

In this step, we simulate a new imbalanced database containing 25 classes to check whether the previous results in single head setup are dependent on our dataset or are related to our neural network architecture. And we do the scenario done in the single head setup as before on this new dataset to see if we see the same pattern as before in the results or not.

For this purpose, we use Caltech101 dataset and select 25 classes in such a way that our dataset is imbalanced. Below you can see how the data is distributed in this dataset.

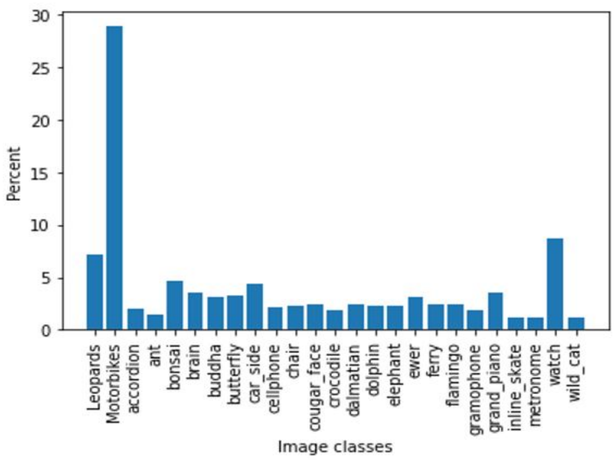

Figure 5.15. Artificial dataset distribution

In this experience, we select 'Motorbikes' and 'Watch' classes as our majority classes and 'Wild cat' and 'Metronome' as our minority classes. The results of this experience are as follows.

[9]Details of all results available at: https://docs.google.com/spreadsheets/d/1Z6RBgwZtp7fdYsLwCELbb1DP5Q_6pzC2V3O9adLLUyI/edit?usp=sharing

sgd _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.1988**	**0.2971**
task1	0.1690	0.2398
task2	**0.4054**	**0.2566**
task3	0.2584	0.3427
task4	0.6222	0.6807
average accuracy	0.3308	0.3634

Table 5.35. Training with sgd - majorityminority - artificial dataset - single head setup

ewc _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	**0.1372**	**0.3988**
task1	0.2535	0.2249
task2	**0.5350**	**0.2020**
task3	0.2340	0.2456
task4	0.2978	0.6982
average accuracy	0.2915	0.3539

Table 5.36. Training with ewc - majorityminority - artificial dataset - single head setup

explicit ewc _ avg of accuracies-single head setup		
task number	majority to task_2 and minority to task_0	majority to task_0 and minority to task_2
task0	0.3634	0.5739
task1	0.1964	0.3128
task2	0.3924	0.2802
task3	0.3893	0.2451
task4	0.3281	0.4436
average accuracy	0.3339	0.3711

Table 5.37. Training with explicit ewc-majorityminority-artificial dataset-single head setup

Results for this dataset are opposite of the results that we got in our dataset. As you see here tasks that contain majority classes get higher average accuracy. And also we could see that when we assign the majority classes to earlier tasks the average accuracy of all tasks is higher than when we assign the majority classes to the later tasks[10]. Therefore, based on the results obtained in this section, it can not yet be concluded with certainty that the assigning of majority

[10]Details of all results at: https://docs.google.com/spreadsheets/d/1jHKdJJaw0AuHCmkTBO9wHn92HlKIR$_3 MaVCJwGAWQyw/edit?usp = sharing$

or minority classes to one or two specific tasks in the single head setup provides a pattern in the results.

5.4.2 Applying rehearsal along with explicit ewc, ewc & sgd

Here in this part, we want to apply rehearsal strategy along with explicit ewc trainer in single head classifier setup to check whether the replay method improves the average accuracy of all tasks or not. We also check how we affect the results by changing the size of our buffer samples in this experiment. We also use rehearsal in conjunction with ewc and sgd to see how big of an impact it has on them.

First, we start with a buffer size of N, which means when we train on task1 we also have N samples from task0 during training time. in the next step when we train on task2 we have N/2 samples from task0 and N/2 samples from task1 during training time. So when we train on task(k) we will have N/k samples from each of the previous tasks. In this experiment, we try three different values for N(300 , 600 , 900) and for each of these sample sizes we do 3 times our experience, so the results are average of them. Below we list the results related to each method along with replay for balanced[11] and imbalanced[12] datasets in single head setup.
sgd along with replay:

sgd _ avg accuracy of all tasks			
	N=300	N=600	N=900
balance	0.9636	0.9730	0.9677
imbalance	0.9512	0.9512	0.9504

Table 5.38. Sgd along with replay

ewc along with replay:

ewc _ avg accuracy of all tasks			
	N=300	N=600	N=900
balance	0.3995	0.4002	0.4011
imbalance	0.2389	0.2389	0.2389

Table 5.39. Ewc along with replay

[11] Detail of results at: https://docs.google.com/spreadsheets/d/1C03bdSYBw10W2p2tEN7d$_{A}qvrGB5MxSG - DnbTB2bijM/edit?usp = sharing$

[12] Details of results at: https://docs.google.com/spreadsheets/d/1rVpaKUhVT32x6XhOU9iWd72t0SPVXhFMHpNuW4FBYzk/edit?usp=sharing

explicit ewc along with replay:

explicit ewc _ avg accuracy of all tasks			
	N=300	N=600	N=900
balance	0.8080	0.7986	0.8428
imbalance	0.5336	0.6782	0.6889

Table 5.40. Explicit ewc along with replay

To compare the results of each method (sgd, ewc, explicit ewc) along with replay versus their results without replay, bellow we have a summary of both scenarios side by side along with how much they change. To calculate the improvement in average accuracy from without replay to with replay we used the following formula: (improvements in avg acc = without replay - with replay).

sgd:

sgd _ avg accuracy of all tasks		
	without Replay	with Replay
balance	0.5619	0.9677
imbalance	0.3092	0.9504

Table 5.41. Sgd-without Replay & with Replay(N=900)

sgd	
improvements in avg acc (without replay - with replay)	
balance	0.4058
imbalance	0.6412

Table 5.42. Sgd-improvements in avg acc by using replay

ewc:

ewc _ avg accuracy of all tasks		
	without Replay	with Replay
balance	0.7211	0.4011
imbalance	0.3383	0.2389

Table 5.43. Ewc-without Replay & with Replay(N=900)

ewc	
improvements in avg acc (without replay - with replay)	
balance	-0.3200
imbalance	-0.0994

Table 5.44. Ewc-improvements in avg acc by using replay

explicit ewc:

explicit ewc _ avg accuracy of all tasks		
	without Replay	with Replay
balance	0.5229	0.8428
imbalance	0.3315	0.6889

Table 5.45. Explicit ewc-without Replay & with Replay(N=900)

explicit ewc	
improvements in avg acc (without replay - with replay)	
balance	0.3199
imbalance	0.3574

Table 5.46. Explicit ewc-improvements in avg acc by using replay

The results show that applying a replay along with sgd and explicit has improved their performance. As can be seen in the balanced dataset case, the average accuracy of all tasks in sgd has increased by 0.4 units and in explicit ewc this value has increased by 0.31 units. And in the imbalanced dataset case, the average accuracy of all tasks in sgd has increased by 0.64 units and in explicit ewc this value has increased by 0.35 units compared to before.

On the other hand, we see that the results obtained in ewc are completely different with sgd and explicit ewc. The results show that applying the replay along with it not only did not improve its performance but also reduced its performance in the balanced dataset by 0.32 units and in the imbalance dataset by 0.099 units.

Regarding the performance of sgd and explicit ewc, it can be said that accessing part of the data of previous tasks during the training of the new task will make our model better update its parameters. In fact, in this case the model parameters could update in such a way that they still perform relatively well on previous tasks.

Regarding the decrease in performance in ewc, after further examination of the results, we found that gradient explosion occurred during the training. Gradient explosion problem occurs when the gradients used to update the weights of neural networks rise rapidly during training time. By printing the amount of loss for all epochs of each task, we realized that from task2 onwards, the amount of loss for all the epochs of the task is equal to NaN. The figures associated with applying replay along with sgd, ewc, and explicit ewc show this issue in ewc very well.

sgd along with replay:

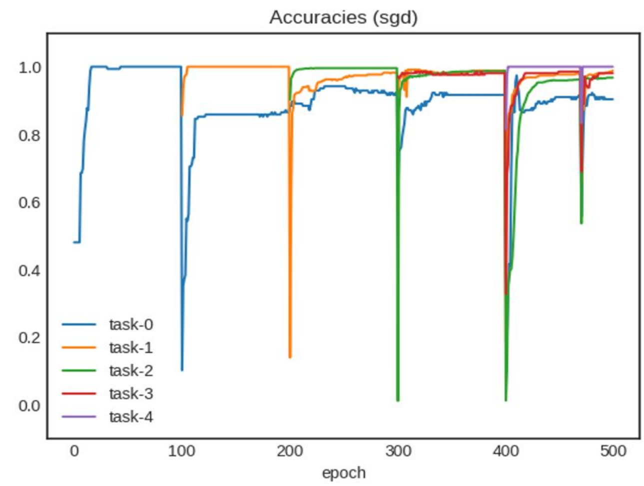

Figure 5.16. Sgd along with replay-accuracy

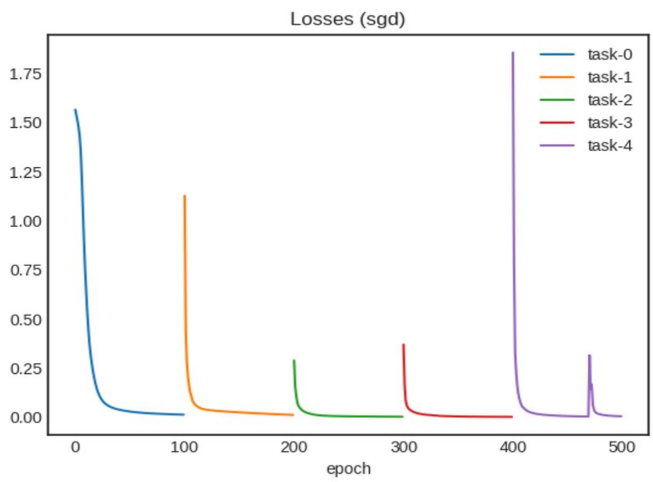

Figure 5.17. Sgd along with replay-loss

ewc along with replay:

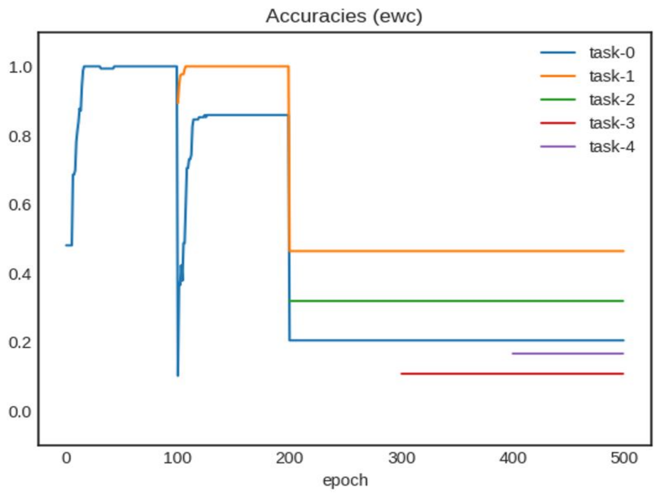

Figure 5.18. Ewc along with replay-accuracy

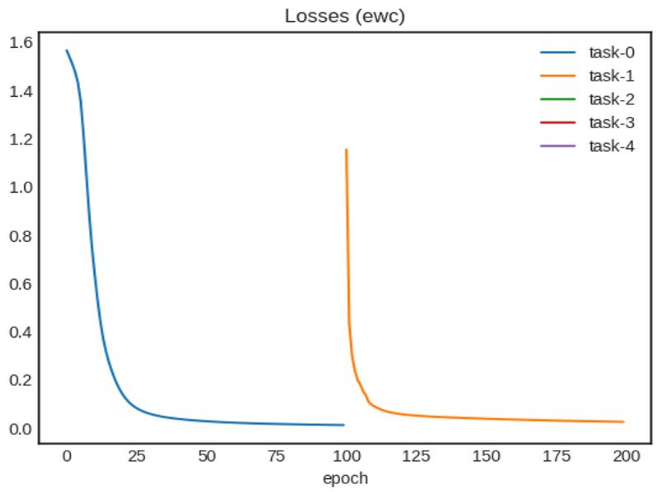

Figure 5.19. Ewc along with replay-loss

explicit ewc along with replay:

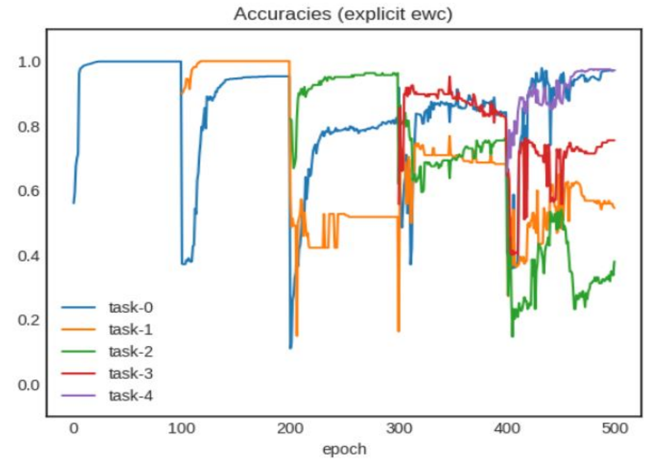

Figure 5.20. Explicit ewc along with replay-accuracy

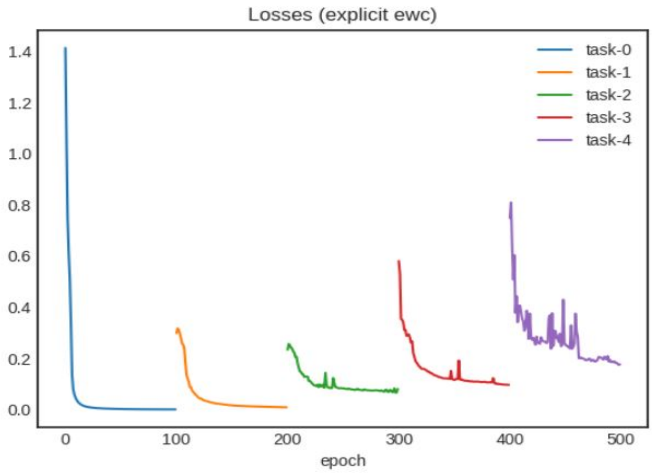

Figure 5.21. Explicit ewc along with replay-loss

It seems that the reason that gradient explosion occurred in ewc, but it did not occur in explicit ewc, is that in ewc the interpolation between the current and previous values of our model parameters is done implicitly. But as we have already mentioned, in explicit ewc this interpolation between the previous and current values of our model parameters is explicit. In fact, in explicit ewc on the contrary to ewc, the probability of unstable training is zero. This is because the interpolation value is always controlled by R which is the ratio of a parameter's importance in past tasks to its importance in all tasks. And this value is always between zero to one. So in explicit ewc we will never see gradient explosion. But still, this is not confirmed and it will be confirmed after careful analysis which is what we will do in the future.

That being said we saw instability when we training with ewc but still, we don't know the exact cause. It might be the dataset has issues or certain other things might be influencing it, discovering this also is one of the future steps[13]

5.5 Important findings

Preliminary results on the performance of CNN as a classifier show well that CNN has been successful in MC in balanced and imbalanced datasets in static conditions. Also since by using CNN in MC we do not require disassembly or execution, and only require to convert the binary code of each malware to a grayscale image, so choosing CNN as a method for classifying malware is a smart choice.

In the next step, the results show that DL in dynamic conditions has not been able to maintain the performance that it previously had in MC in static conditions. Indeed, it can be said that the reason for this is CF. In fact, by training on the new task, our model updates its parameters according to the data that comes from the new task. It means that it will lose what it had learned during the training in the previous tasks. So it does not have the previous performance in the previous tasks after training on the new task.

We also see that the impact of freezing convolutional layers in our CNN model is different in single head classifier and multihead classifier setup. In multihead setup we could see in the results that freezing always increases the average accuracy of all tasks in sgd. But we can not see always the same pattern in the results of ewc and explicit ewc (because of lack of time we didn't do further analysis in this issue). Another point that we find in the results is that applying explicit ewc improve the average accuracy of all tasks compare to sgd even when we freeze some layers in our model in balanced and imbalance datasets.

On the other side results related to freezing the convolutional layers in single head setup, show us that freezing first two convolutional layers has not any impact in results of sgd in balanced and imbalance datasets. And also we could not find any specific pattern in the results when we apply explicit ewc compare to sgd in this case. But freezing only first convolutional layer decreases the average accuracy of all tasks in balances dataset and still keeps results almost the same as normal fitting in imbalanced dataset. Also applying explicit ewc improve the results compare to sgd in balanced and imbalance datasets.

From the results, it can be concluded that explicit ewc, our implemented regularizer to overcome the CF in the MC in the dynamic scenario, has worked well in the multihead classifier and it can improve the average accuracy of all tasks compared to sgd (our setup without regularizing) in balanced and imbalanced datasets. This indicates that the explicit strategy in

[13]Details of results at: https://docs.google.com/spreadsheets/d/1hNgsQMoklvBCt9AM5BIps0pNixyPYxLVLaJ7nxGobMA/edit?usp=sharing

explicit ewc, which is the proper controlling of the interpolation between the current and previous values of our model parameters when our model is training on the new task, is well done by R.

Indeed R is a ratio of a parameter's importance in past tasks to its importance in all tasks. So based on this definition R is always in ranges from 0 to 1 for all parameters, so it will prevent extrapolation (ensuring stable training). On the other side, we also could avoid concerns with assigning lower importance scores to deeper layers in deep neural networks by adopting this definition for R because even though a parameter's absolute value is low, it can nevertheless have high relative importance if it is unimportant to past tasks.

In fact, when our model was training on new families of malware, it was able to resist the change in the important parameters of our model for each of the previous malware families that they are already trained and tested. So this model continuously performs well in detecting and classifying previous malwares after it trains on the new malwares. On the other hand, by examining the results, we will find that explicit ewc alone can not perform well in single head setup because in this case, R could not do a proper interpolation between the current and previous values of our model parameters during training of our model on the new task.

The results of the study of explicit ewc during the training of our model on the balanced dataset and imbalanced dataset show that in both of these types of datasets, explicit ewc in the multihead setup had almost the same performance. This indicates that even in situations where the dataset is imbalanced, which is of course closer to the real situation in malware datasets, by applying explicit ewc regularizer we can be effective in detecting and classifying malwares in the dynamic scenario.

Another point that can be deduced from the results is that the presence of majority or minority classes in the beginning or middle tasks does not provide any pattern in the results obtained from them. In fact, this shows that when training our model on a sequence of malwares, it does not matter if the majority or minority classes of malwares are at the beginning of the training queue or in the middle of the queue.

Finally, the results from our hybrid approach consisting of rehearsal and the regularization methods in single head setup show us that rehearsal works very well along with explicit ewc. But we could not say something precisely about its effect along with ewc because we saw unstable training while we did rehearsal along with ewc and we still do not know the cause.

Chapter 6

Conclusion

First, we saw that in CL situations, even a highly accurate deep model like CNN along with SGD optimizer performs poorly for MC in multihead classifier and single head classifier setup. Another observation in this experiment is that the amount of forgetting of previous tasks when we train on new tasks in multihead classifier setup is less than single head classifier setup.

As a result, applying ewc as a quadratic regularization with explicit interpolation of recent and previous values of our model's parameters which is named explicit ewc, improves the average accuracy of all tasks in a multihead classifier setup in balance and imbalance datasets in comparison to ewc. We got almost the same results for average accuracy of all tasks when we train our model along with explicit ewc as our regularizer on balanced and imbalance datasets (0.9788 for the balanced dataset and 0.9723 for the imbalance dataset). Indeed we improve the average accuracy of all tasks by 0.26% in the balanced dataset and 0.12% in the imbalance dataset by applying explicit ewc instead of ewc.

The results obtained from the two regularizers (ewc and explicit ewc) reveal that they do not work properly on our imbalance dataset in the single head classifier setup. On the other hand even ewc works better than explicit ewc on our balanced dataset in single head classifier setup. Another observation is that when we compare the results of explicit ewc with the sgd (training without regularization), its result in the average accuracy of all tasks is less than sgd on balanced dataset in single head classifier setup.

The results regarding the study of the impact of freezing the feature learning layers on the CL performance of our model show us that freezing of feature layers along with SGD optimizer decreases to some extent forgetting in CL when we train our CNN model on balanced and imbalance datasets.

Therefore, the results of the hybrid approach show that combining rehearsal with explicit ewc regularizer has improved its performance in single head setup on both balanced and imbalanced datasets but we saw unstable training when we combine rehearsal with ewc. In conclusion, although explicit ewc works very well in multihead setup in balance and imbalance datasets still a large gap exists in applying explicit ewc in single head setup in CL scenarios, which requires new approaches and techniques.

Glossary

ML: Machine Learning
DL: Deep Leaning
DNN: Deep Neural Network
CF: Catastrophic Forgetting
MC: Malware Classification
CL: Continual Learning
CNN: Convolutional Neural Network
SGD: Stochastic Gradient Decent
EWC: Elastic Weight Consolidation

Bibliography

R. Aljundi, F. Babiloni, M. Elhoseiny, M. Rohrbach, and T. Tuytelaars. Memory aware synapses: Learning what (not) to forget. pages 139–154, 2017. doi: https://doi.org/10.48550/arXiv.1711.09601.

B. Alsulami and S. Mancoridis. Behavioral malware classification using convolutional recurrent neural networks. pages 103–111, 2108. doi: https://doi.org/10.48550/arXiv.1811.07842.

F. Castro, M. Marín-Jiménez, N. Guil, C. Schmid, and K. Alahari. End-to-end incremental learning. pages 233–248, 2018. doi: https://doi.org/10.48550/arXiv.1807.09536.

A. Chaudhry, P. Dokania, T. Ajanthan, and P. Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. pages 532–547, 2018. doi: https://doi.org/10.48550/arXiv.1801.10112.

A. Chaudhry, M. Rohrbach, M. Elhoseiny, T. Ajanthan, P. Dokania, P. Torr, and M. Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019. doi: https://doi.org/10.48550/arXiv.1902.10486.

T. Dalton, M. Schmidtler, and A. Khodabakhshi. Classifying malware using function representations in a static call graph. pages 243–254, 2020. doi: https://doi.org/10.48550/arXiv.2012.01939.

J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. pages 647–655, 2013. doi: https://doi.org/10.48550/arXiv.1310.1531.

A. Douillard, M. Cord, C. Ollion, T. Robert, and E. Valle. Podnet: Pooled outputs distillation for small-tasks incremental learning. pages 86–102, 2020. doi: https://doi.org/10.48550/arXiv.2004.13513.

E. Fini, S. Lathuilière, E. Sangineto, M. Nabi, and E. Ricci. Online continual learning under extreme memory constraints. pages 720–735, 2020. doi: https://doi.org/10.48550/arXiv.2008.01510.

I. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgeting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013. doi: https://doi.org/10.48550/arXiv.1312.6211.

Y. Han, J. Liu, B. Xiao, X. Wang, and X. Luo. Bilevel online deep learning in non-stationary environment. pages 347–358, 2022. doi: https://doi.org/10.48550/arXiv.2201.10236.

J. Hong and T. Pavlic. Learning to modulate random weights can induce task-specific contexts for economical meta and continual learning. *arXiv preprint arXiv:2204.04297*, 2022. doi: https://doi.org/10.48550/arXiv.2204.04297.

Y. Huang, T. Chen, Y. Sun, and M. Chen. Learning malware representation based on execution sequences. *arXiv preprint arXiv:1912.07250*, 2019. doi: https://doi.org/10.48550/arXiv.1912.07250.

M. Jain, W. Andreopoulos, and M. Stamp. Cnn vs elm for image-based malware classification. *arXiv preprint arXiv:2103.13820*, 2011. doi: https://doi.org/10.48550/arXiv.2103.13820.

H. Jung, J. Ju, M. Jung, and J. Kim. Less-forgetting learning in deep neural networks. *arXiv preprint arXiv:1607.00122*, 2016. doi: https://doi.org/10.48550/arXiv.1607.00122.

C. Kim. A machine learning approach to malware detection using native api system calls. *arXiv preprint arXiv:1802.05412*, 2018. doi: https://doi.org/10.48550/arXiv.1802.05412.

J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017. doi: https://doi.org/10.1073/pnas.1611835114.

A. Kwan. Malware detection at the microarchitecture level using machine learning techniques. *arXiv preprint arXiv:2005.12019*, 2020. doi: https://doi.org/10.48550/arXiv.2005.12019.

J. Li, D. Xue, W. Wu, and J. Wang. Incremental learning for malware classification in small datasets. *Security and Communication Networks*, 2020. doi: https://doi.org/10.1155/2020/6309243Showcitation.

Y. Li, Z. Li, L. Ding, Y. Hu, W. Chen, and X. Gao. Supportnet: a novel incremental learning framework through deep learning and support data. 2018. doi: https://doi.org/10.1101/317578.

Z. Li and D. Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2106. doi: https://doi.org/10.48550/arXiv.1606.09282.

D. Lopez-Paz and M. Ranzato. Gradient episodic memory for continual learning. *Advances in neural information processing systems*, 30, 2017. doi: https://doi.org/10.48550/arXiv.1706.08840.

E. Lubana, P. Trivedi, D. Koutra, and R. Dick. How do quadratic regularizers prevent catastrophic forgetting: The role of interpolation. *arXiv preprint arXiv:2102.02805*, 2021. doi: https://doi.org/10.48550/arXiv.2102.02805.

F. Mi, T. Lin, and B. Faltings. Representation memorization for fast learning new knowledge without forgetting. *arXiv preprint arXiv:2108.12596*, 2021. doi: https://doi.org/10.48550/arXiv.2108.12596.

L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. pages 1–7, 2011. doi: https://doi.org/10.1145/2016904.2016908.

Q. Pham, C. Liu, and S. Hoi. Continual normalization: Rethinking batch normalization for online continual learning. *arXiv preprint arXiv:2203.16102*, 2022. doi: https://doi.org/10.48550/arXiv.2203.16102.

M. Rafique, M. Ali, A. Qureshi, A. Khan, and A. Mirza. Malware classification using deep learning based feature extraction and wrapper based feature selection technique. *arXiv preprint arXiv:1910.10958*, 2019. doi: https://doi.org/10.48550/arXiv.1910.10958.

V. Ramasesh, E. Dyer, and M. Raghu. Anatomy of catastrophic forgetting: Hidden representations and task semantics. *arXiv preprint arXiv:2007.07400*, 2020. doi: https://doi.org/10.48550/arXiv.2007.07400.

A. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. pages 806–813, 2014. doi: https://doi.org/10.48550/arXiv.1403.6382.

S. Rebuffi, A. Kolesnikov, G. Sperl, and C. Lampert. icarl: Incremental classifier and representation learning. pages 2001–2010, 2017. doi: https://doi.org/10.48550/arXiv.1611.07725.

A. Rusu, N. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. doi: https://doi.org/10.48550/arXiv.1606.04671.

I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231: 64–82, 2013. doi: https://doi.org/10.1016/j.ins.2011.08.020.

H. Shin, J. Lee, J. Kim, and J. Kim. Continual learning with deep generative replay. *Advances in neural information processing systems*, 30, 2017. doi: https://doi.org/10.48550/arXiv.1705.08690.

R. Tiwari, K. Killamsetty, R. Iyer, and P. Shenoy. Gcr: Gradient coreset based replay buffer selection for continual learning. *arXiv preprint arXiv:2111.11210*, 2021. doi: https://doi.org/10.48550/arXiv.2111.11210.

D. Vu, T. Nguyen, T. Nguyen, T.Nguyen, F. Massacci, and P. Phung. A convolutional transformation network for malware classification. pages 234–239, 2019. doi: https://doi.org/10.1109/NICS48868.2019.9023876.

G. Yang, E. Fini, D. Xu, P. Rota, M. Ding, H. Tang, X. Alameda-Pineda, and E. Ricci. Continual attentive fusion for incremental learning in semantic segmentation. *IEEE Transactions on Multimedia*, 2022. doi: https://doi.org/10.48550/arXiv.2202.00432.

J. Yoon, E. Yang, J. Lee, and S. Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2018. doi: https://doi.org/10.48550/arXiv.1708.01547.

F. Zenke, B. Poole, and S. Ganguli. Continual learning through synaptic intelligence. 70:3987–3995, 2017. doi: https://doi.org/10.48550/arXiv.1703.04200.

D. Zhou, H. Ye, and D. Zhan. Few-shot class-incremental learning by sampling multi-phase tasks. *arXiv preprint arXiv:2203.17030*, 2022. doi: https://doi.org/10.48550/arXiv.2203.17030.

W. Zhuang, L. Xiao, J. Cui, and W. Zhuang. Support vector machine based on incremental learning for malware detection. pages 204–207, 2015.