Bsc Computer Science

Final Project

"Deep learning for malware classification"

September 2023

1. Introduction

This is a project proposal for the “Machine Learning and Neural Networks” template. The chosen topic is “Deep Learning on Public Dataset".

Malware, or malicious software, is a pervasive threat in today's interconnected world. Its purpose ranges from illicit control over computers to theft of valuable information, posing significant security risks to individuals and organizations alike. Given the society's increasing dependence on computer networks, maintaining robust security has become a critical priority.

The motivation for this project stems from the dual challenges of combating the persistent threat of malware and harnessing the power of artificial intelligence (AI) in security endeavors. The potential misuse of AI in the development of new, sophisticated malware strains is a growing concern.

Malware analysis is typically categorized into static methods, where the malware binary samples are examined and their features are extracted and then used to detect and classify malicious code, and dynamic methods, where the malware is observed in sandbox environment to analyze its behavior. In an everyday setting, malware has to be detected quickly by an antivirus tool in order to prevent damage to the system. As such, malware detection is mainly a static analysis domain problem.

However, with an increasing rate of malware attacks and creation of new malware strains, as well as utilization of code obfuscation techniques by threat actors, traditional machine learning methods that require manual feature engineering, which often very time-consuming requires specialized expertize, is quickly becoming insufficient.

Several studies have demonstrated the viability of using deep learning, where the main advantage is automatic feature extraction, for malware classification. Nataraj et al, 2011 [8] had proposed a method for visualizing malware binary samples as images, which allows to utilize computer vision techniques for malware classification. This approach is promising for malware detection, as it was shown to be robust to popular code obfuscation techniques [1]. Studies such as Rafique et al in 2019 [2] and Ahmed Bensaoud et al in 2020 [10], successfully demonstrate the application of Convolutional Neural Network (CNN) architectures to malware classification problem, and Seneviratne al in 2022 [3] shows how Vision Transformers (ViT’s) can be applied in this context as well.

The study by Seneviratne et al [3] shows a comparison between their custom ViT SHERLOCK with several state-of-the-art CNNs. There are a few drawbacks in the study, however, such as a lack of clearly outlined model training methodology and no direct comparison between the proposed self-supervision enabled model’s performance against a “vanilla” ViT architecture, despite the authors mentioning using a ViT-B/16, which is a ViT variant pre-trained on a ImageNet-21K dataset (Dosovitskiy et al 2021 “An image is worth 16x16 words”). There is evidence that suggests that transfer learning does not provide significant benefits in settings where there is high semantic disparity between the dataset used in pre-training and fine-tuning [1], which the authors cite themselves.

This leads to the following research questions:

- Do ViT models fine-tuned from foundational models pre-trained on large non-domain specific datasets provide performance benefits in the specific domain of malware classification?
- How well do ViT models perform compared to state-of-the-art CNN models without pre-training?
- Does self-supervision approach enhance performance compared to supervised training, and how significantly?

Additionally, due to imbalanced nature of most publicly available malware datasets, another question I intend to investigate is: does training a model on a dataset composed of samples from one operating system generalize to same tasks on other operating systems?

The project aims to address these questions by presenting an experiment running several models that are considered state-of-the art on a transparent training pipeline, with performance investigated in both pre-trained and non-pre-trained model states. The performance of these models is then reported and compared using the macro F1-score, macro-precision and macro-recall, as well as their relative training costs.
The goal of this project is to investigate a number of approaches to malware detection and classification using of-the-art deep learning techniques in order to determine the one that is most suited to address the stakeholder requirements.

2. Literature review

2.1. "Classifying Malware Images with Convolutional Neural Network Models" by Ahmed Bensaoud, Nawaf Abudawaood, and Jugal Kalita.

The study in this paper presents a method for application of Convolutional Neural Networks. The authors cite the increasing time and effort costs associated with manual extraction of features that is required for traditional machine learning methods such as Naive Bayes, k-nearest neighbors etc as the motivation for shift to automatic static analysis techniques with deep learning. The authors compare several then state-of-the-art CNN models with models demonstrated in 2017 research paper “ Towards building an intelligent anti-malware system: A deep learning approach using support vector machine (SVM) for malware classification” by Agarap and Pepito.

The researchers train 6 models in total. Although they achieve impressive accuracy results, particularly with Inception V3 model, there are some issues could have had an impact on the final results:

1. The researchers mention that there are several datasets available for research, but do not explicitly state why they chose MalImg. They also do not state how or if they account for imbalances in the MalImg dataset.
2. The images used in research are grayscale, which, while providing a simpler and more direct conversion of binary files, contain less nuanced representation and thus may lead to decreased performance of a model, which showed especially on VCG16 and ResNet50, which were not designed for grayscale input.
3. The researchers do not explicitly state any hyperparameters which they used to train the models, which makes it difficult to recreate the results.

I believe that clearly stating the parameters used for each model in addition to their architecture would have facilitated the recreation of the results achieved in the paper. Using an RGB image dataset like CICAndMal2017 may have yielded more significant results in terms of comparing performance of models, as only there were only two models that were not designed for RGB images. Introducing micro F1 score alongside accuracy could have also

2.2. “MalNet: A Large-Scale Image Database of Malicious Software”

This paper presents a new dataset designed for the purposes of malware classification research called “MalNet”, which contains 1 262 024 byte images belonging to total of 696 classes. This Represents a 24-time increase over the second-largest public Virus-MNIST and 133-times increase over third-largest Malimg, with number of classes increasing 70 and 28-times respectively. At the

time of this project’s submission, it is the largest publicly available dataset containing byte image data.
The authors of the paper state that due to increased adoption of code obfuscation techniques, such as code polymorphism, the efficiency of signature-based detection approach suffers. As such, the industry has turned to computer-vision based approaches which are more robust against this type of threat. However, they note, most of the research ongoing in the field is either small scale or uses non-public data, which hinders the overall its progress.
The dataset was constructed by collecting Android APK (package files) from the AndroZoo [12] repository filtered by presence of labels obtained from Euphony [13]. Using this approach the researchers collected 1 262 024 packages that were already labeled by various antivirus vendors. The images themselves were created by extracting the bytecode (DEX) of each application and converting it into a vector of 8-bit unsigned integers, where each integer is in range [0,255] with 0 representing a completely black pixel and 255 representing white pixel. Colors are assigned to each pixel depending on the position of the associated byte in the source DEX file.
The researchers demonstrated various applications in which the dataset can be utilized. They’ve run seven experiments on different types of convolution neural network models, detailing the parameters, the training setup and the results in significant detail. They demonstrated how they accounted for dataset imbalances, and how their approach produced improvements in underrepresented classes.
However, despite the exceptional contribution of this paper, I would argue that constructing the dataset exclusively from Android binaries may have a negative impact on the ability of models that use this data to train to recognize and classify malware targeted for other platforms, the most notable example being Microsoft Windows, which was the target for 95% of new malware strains in 2022 [11].

2.3. “An image is worth 16 x 16 words: transformers for image recognition at scale”

This paper introduces a novel application of Transformer architecture for image classification tasks. The authors cite that, despite the success of transformer-based models in NLP field, image classification remains dominated by convolutional neural networks with ResNet considered state-of-the-art, despite the attempts to combine their architecture with self-attention mechanisms characteristic of transformers.
The authors propose to apply standard transformer architecture, with minimal modifications ,directly to images. To achieve this, the image is split into a number of fixed-size patches, which are then converted to embeddings as the sum of their image and positional embedding vectors, and passed to the transformer encoder block.
The authors found that, although this approach works, the model performs worse than a similarly sized ResNet model when trained on a medium-sized dataset (ImageNet-1K). The reason is that, unlike CNNs, transformers do have inductive biases characteristic of CNNs, and therefore perform less efficiently when not trained on large amounts of data. In order to address this problem, the authors train the Vision Transformer on large-scale datasets, such as ImageNet-21K (14M samples) and JFT-300M (300M samples). This allowed them to match or beat the performance of state-of-the-art CNN models.
Overall, this work presents a novel and promising approach to computer vision tasks, particularly image classification, which my project falls under. The authors make note that Vision Transformer performs best on a downstream task when pre-trained on a large dataset, which I intend to investigate as a part of this project, namely how well does pre-training on a general dataset translates to improved performance for malware classification tasks compared to training the classifier on target data from scratch.

2.4. “Self-supervised Vision Transformers for malware detection" by Sachith Seneviratne, Ridwan Shariffdeen, Sanka Rasnayaka, and Nuran Kasthuriarachchi

This paper proposes a new Transformer-based computer-vision model called SHERLOCK, which is designed for classification of malware images into multiple classes. The authors use a novel (with regards to malware image classification) self-supervised learning approach, where the data is used as both features and labels. In order to implement it, the researchers use masked auto-encoding, where the next sequences of data are hidden (masked), and the masked image is then encoded into latent representation. The researchers show how the decoder component of the model reconstructs the image from the latent representation, which is compared to the actual image present in the dataset. This approach enabled them to reuse the same foundational representation in all classification tasks, which they cite as a significant advantage over supervised models, especially when more different tasks are considered.
The authors provide details about their hardware setup, outline the metrics by which they measure performance of the models which they compare in their experiments. They do not, however, release information about hyperparameters such as the number of epochs for training, learning rate etc., which makes reproducing their work significantly harder.
For benchmarking they referenced the performance of three popular CNN architecture models: ResNet, DenseNet and MobileV2 from [1]. The authors make a claim outperforming the state-of-the-art models with best accuracy scores for malware-type (83.7%) and malware-family (80.2%), however the values appear to be different in Table 2 where they present the results for three classification tasks.
Moreover, while the table does show SHERLOCK outperforming the three benchmark CNN models on type and family classification, on binary classification (detection) it gets one of the lowest F1 scores, and the lowest recall. The authors cite that transfer learning does not scale when there is significant semantic difference between foundational and downstream data, but do not make a comparison between their model, which appears to be fine-tuned from ViT-B/16, which itself is a ViT model pre-trained on an ImageNet dataset [4].
Overall, while the research does present a novel approach to malware classification tasks, it could have benefited from a more in-depth description of the training process and hyperparameters used for reproducability. The methodology could have also been further improved by demostrating the performance of the new training approach compared to a standard supervised one.

2.5. Training Vision Transformers from Scratch for Malware Classification by Ricky Xu

This is an example of a project which is similar to my own. The author outlines that his project was prepared for the 2021 iFLYTEK A.I. Developer Challenge competition.
Similarly to [3], they motivate their choice of architecture by the fact that Vision Transformers have attained results comparable to CNNs in computer vision tasks. They follow the Keras standard implementation of Vision Transformer model [14] and adapt it for use with their data. The details of hyperparameters and implementation are shown in their published code.
The data they used for competition comprised a total of 58 000 samples with assembly opcode frequency data and file size as features, which is sparse in comparison to MalNet dataset. The author does not account for imbalances in the dataset, which could have had an effect on models’ performance.
Although the author shows the plots of their model’s loss and accuracy over training and validation processess, they do not explicitly compare its’ performance against any benchmark models.
Overall I found that the project and the results could be improved using a richer dataset, and the opcode frequency may not provide sufficient semantic context for classification of malware samples at scale (although both points were the competition restrictions the author had to work with). In

addition, the author does mention that pre-trained CNN models good results and are faster for training, which would have been a good benchmark to test against.

3. Project design

Project template: Deep learning on a public dataset

3.1 Domain of the project and user requirements

The domain of the project is malware detection and prevention using malware images. The potential stakeholders of this project fall under the following categories:

- Regular users: people who use their digital devices to perform everyday tasks. The main requirement for this type of user is that the model, as a part of an antivirus solution, accurately and quickly determines whether any piece of software could potentially pose thread to their devices. However, since this category is very broad, there it poses a challenge to develop a model that is both robust and relatively lightweight such that it could be run on any type of consumer hardware;
- Organizations: companies are often targets for malicious actors due to being large repositories of valuable data. Companies that want to protect their digital assets may utilize deep learning models as part of their security program, for example, as a security control to inspect the incoming or internal traffic for malware markers. This category of stakeholders may vary is size greatly, and thus scalability of the architecture will be important.
- Cybersecurity organizations and researchers: the companies that research and develop detection and prevention solutions. This class of users would benefit from a model that has robust performance on type and family classification tasks, which would allow them to accelerate analysis of new malware samples. Because the new threats are constantly emerging, one of the key requirements that this category of users might impose is adaptability, meaning that the model should be relatively easy to train, so that it can incorporate new information quickly.

The main requirements of the project are therefore:

- Efficient architecture which would allow to utilize smaller number of layers while preserving accuracy;
- Modular design that is well suitable for parallelization;
- An architecture that allows to simplify the training and testing process.

3.2. Model selection and implementation

CNNs have long been a popular choice in image processing tasks due to their unique capacity to analyze local and global features from images through the use of convolutional layers. They are capable of reducing the dimensionality of data through their layered architecture, making them relatively efficient in terms of computational resources and suitable for various hardware configurations, which is especially important for regular users who may not have powerful hardware. The layered architecture of CNNs also inherently supports parallelization, which makes them highly suitable for large organizations that need to process large volumes of data simultaneously.

The Vision Transformers are a relatively new development in the field of deep learning, based on the successful application of Transformers in Natural Language Processing. They treat the image as a sequence of patches and apply self-attention mechanism to capture global dependencies between the patches, which could lead to better detection and classification results.

ViTs could be particularly useful for cyber security organizations and researchers. The global dependencies learned by ViTs may capture novel relationships in byte images that are not

immediately apparent or identifiable through local feature analysis, thus potentially aiding in the detection and classification of previously unseen or novel types of malware. ViTs have a training advantage: once trained, they can be fine-tuned on specific tasks or datasets relatively quickly and easily. Additionally, their self-attention mechanism makes them natively suitable for generation of attention maps. All of this makes them an excellent tool for cyber security researchers.
There is a significant body of research examining the performance of CNN architecture for malware detection and classification. Background research suggests that the most robust models for this task are ResNet [15] and MobileNet [16]. Their implementations, as well as recommended parameters are available in most deep learning frameworks and papers. As such, I will avoid rewriting them for the purpose of this experiment.
Vision Transformers have started to appear more often in recent research in the domain [3][5][17]. However, the literature I've examined so far haven't provided a fully described architecture and training methodology with application to malware detection. For this reason, I've elected to implement a Vision Transformer model manually, as well as to attempt to recreate a variation described in [3] to able to address the research questions presented.
Overall, the total of six architectures will be trained and compared to cover the scope of this project:

1. ViT-B/16 – the base model as presented in [4], fine-tuned on Malnet dataset;
2. ViT-SHERLOCK – a variant trained using self-supervised training as described in [3]. Although authors don't release the specific parameters of training process, they describe their approach sufficiently to recreate it and compare against a supervised model;
3. ViT-V – my own implementation of ViT, using just the parameter tuning;
4. ViT-VS – same as above, but using a masked auto-encoder from [3] for training;
5. ResNet18 – as a benchmark;
6. MobileNetV2 – same as above.

3.3. Data Collection and Preprocessing

For this project, I will utilize the MalNet dataset, which is a large-scale collection containing over 1.2 million byte-image samples. Although it is the most extensive publicly available dataset of its kind, it is limited to Android malware images [1]. Despite this limitation, I have elected to use it for model training because Android currently has the largest market share among mobile operating systems [18], and large-scale Windows PE datasets are not readily available or have limited use due complicated licensing associated with Windows binaries.
The dataset provided by the authors includes 1,262,024 RGB images in PNG format, with a resolution of 256x256. These images are organized into 47 folders named by binary type, each containing 696 sub-folders labeled by binary family. This organization enables easy loading of the files into a TensorFlow ImageGenerator object for further processing.
For data augmentation and regularization, I intend to follow the recommendations provided in the paper Steiner et al's 2022 "How to Train Your ViT?" [7]. As they suggest, for data augmentation I will utilize Mixup. Mixup works by taking pairs of images, forming a weighted average of each pair, and assigning a label that is also a weighted average of the corresponding labels. This process can improve the robustness of decision boundaries between classes and help prevent overfitting.
Instead of using RandAugment, the second technique proposed in [7]," I will apply Gaussian, Poisson, and Laplacian noise as described in Ozgur Catak et al.'s 2020 "Data Augmentation Based Malware Detection Using Convolutional Neural Networks" [19]. The reasoning for this is that due to byte images representing specific sequences of data, instead of spatial features like "normal" images, random spatial transformations like rotation, zoom, skew etc. are unlikely to provide performance benefits. Applying noise in this situation provides a way of masking certain bytes, which also may help prevent overfitting.

To address wether models trained in this project work well on other datasets, I will perform a single evaluation run of each model on a Virus MNIST dataset [6], which contains 51 880 byte plots of PE binaries (Windows) across 10 classes.

3.4. Model training

3.4.1 Hardware setup

All experiments are planned to be conducted on a system with following specifications: AMD Threadripper WX5950 CPU, 128 GB RAM, Nvidia RTX 4090 GPU. Due to deadline considerations, a GPU cloud such as lambdalabs.com may be utilized to offload some of the parameter validations runs, in which case their specifications will be listed in the final report.

3.4.2. Software libraries

The models will be implemented using Keras. Keras is a high-level API for TensorFlow, a popular machine learning framework. I chose Keras because it allows to abstract away the lower-level details of implementation and enables fast prototyping. It provides bindings in Python, a user-friendly language that is similarly well-suited for testing. The framework also provides access to a number of models out-of-the-box, including CNNs like ResNet50 and MobileNet.
To incorporate hyperparameter tuning into the training pipeline, I will use the KerasTuner library.
For training and evaluation visualization, the seaborn and matplotlib libraries will be used.
To provide an easy way to compute the performance metrics, I will utilize the sci-kit learn library, in particular its' confusion matrix and classification report functions. Pandas and numpy are included to provide backend conversion of data.
For development, I utilize Jupyter Lab IDE. Jupyter facilitates quick prototyping through its' use of cells, which enable modular out-of-order execution of code that is particularly suitable for making and testing changes to code without having to execute the entire program from the start.

3.4.3. Training

In order to achieve best possible performance for the Vision Transformer models, I will follow the recommendations from Andreas Steiner el al's 2022 "How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers" [7]. The paper specifically provides guideline boundaries for learning rate, weight decay and training epochs. Due to constrains of computational budget and timeline, however, the number of training epochs for each model will be limited to 100.
For ResNet [14] and MobileNetV2 models, the training parameters will be set as recommended in their respective papers. To account for significant class imbalances in the MalNet dataset, I will utilize class reweighing as proposed in [1]. For optimization, AdamW will be used, as proposed by Loshchilov and Hutter's 2017 "Decoupled Weight Decay Regularization" [9]. The training, validation and testing splits (70-10-20%) are provided by the creators of the MalNet dataset. Due to this, the hyperparameter tuning phase will be implemented as follows. Each model is trained on the training set using a number of parameter sets. For Vision Transformers, the parameters being tuned consist of the number of transformer blocks, model width, MLP size and the number of attention heads. For CNN-based models the parameters are the number of stages, model width, filter size, stride and pooling. Each parameter set is then evaluated on validation set. The best performing set of parameters is then used to train the model on the combined training and validation set, and the models performance is then evaluated on the testing set.

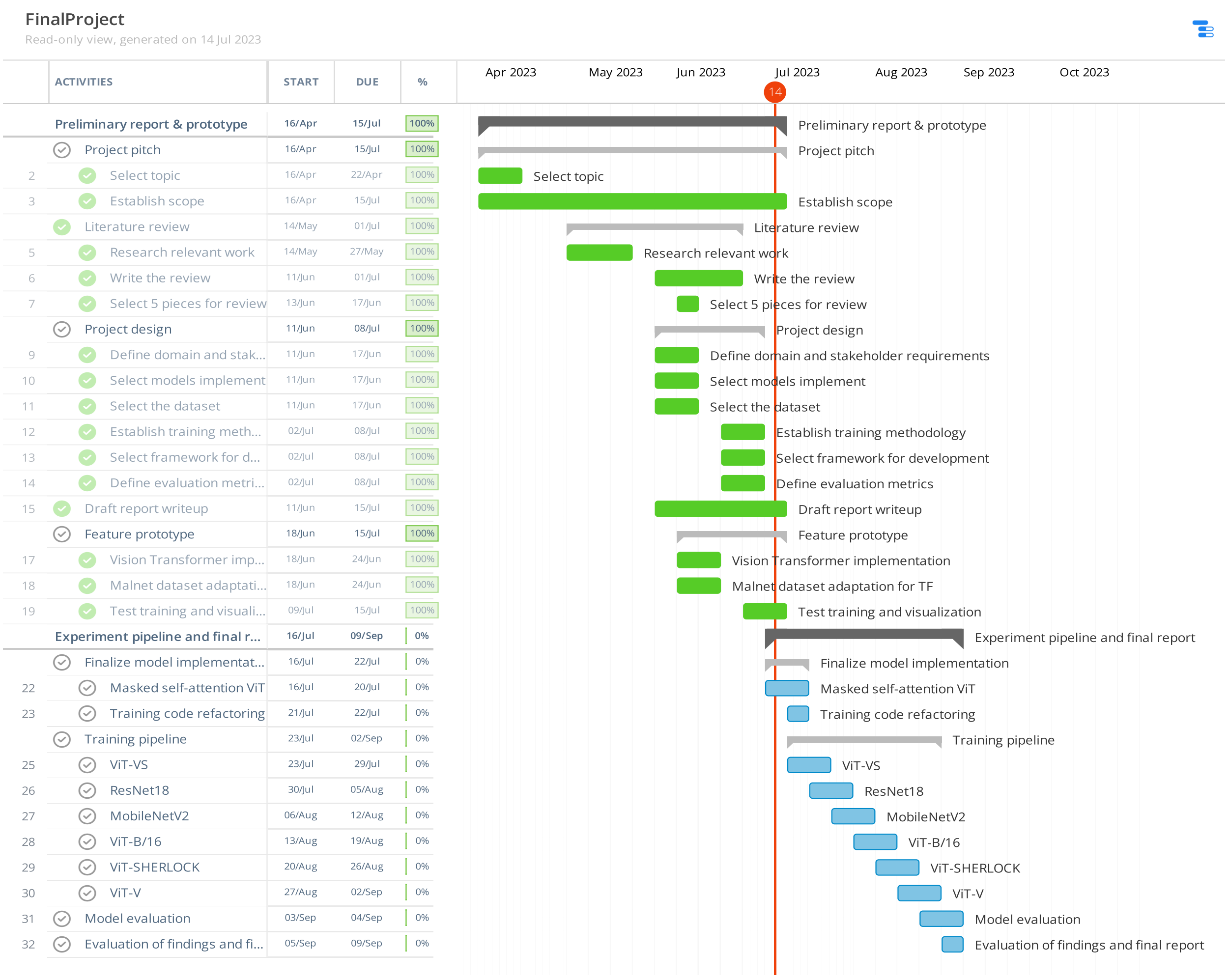
3.5. Evaluation plan

The evaluation of all models will be carried out using the following metrics: macro F1-Score, macro precision and macro recall. The choice of evaluation metrics is motivated by the fact that the dataset chosen for this project is highly imbalanced [1]. Macro-averaging assigns equal weight to all

classes, regardless of their frequency, which allows to better measure performance on rare classes. These metrics inform about the overall utility of the proposed model and are important to all user categories in the chosen domain.
In addition to the above metrics, an inference speed will also be measured. It is defined as time taken from input to produce a prediction, and will be measured in seconds. The motivation is that even if the model is accurate, it may not be practical to use if it is not able to classify malware in timely fashion.
The computational complexity of the forward pass, as measured in GFlops will also be taken into account, as it will be an important factor when considering deployment on edge devices. The computational complexity will be obtained using a TensorFlow profiler.

3.6 Work plan

Preliminary runs with the prototype Vision Transformer shown that a single pass through the training set of 61 201 samples took approximately 3 minutes. Based on these results, the estimated time cost of training a single Vision Transformer with current parameters on a full-sized dataset, using the specifications outlined in Section 3.5.1, appears to be around 72 hours for 100 training epochs. Following the completion of preliminary report and prototype, the training code will be refactored to facilitate a proper sequential training pipeline. There is a total of six models to be tested, and, with an above estimation, 7 to 8 weeks should be sufficient to complete the experiment, including the 1 to 2 weeks for code refactoring and visualization setups. Below is a visualization of the proposed training plan [Figure 1].


FinalProject
Read-only view, generated on 14 Jul 2023

	ACTIVITIES	START	DUE	%
	Preliminary report & prototype	16/Apr	15/Jul	100%
	Project pitch	16/Apr	15/Jul	100%
2	Select topic	16/Apr	22/Apr	100%
3	Establish scope	16/Apr	15/Jul	100%
	Literature review	14/May	01/Jul	100%
5	Research relevant work	14/May	27/May	100%
6	Write the review	11/Jun	01/Jul	100%
7	Select 5 pieces for review	13/Jun	17/Jun	100%
	Project design	11/Jun	08/Jul	100%
9	Define domain and stak...	11/Jun	17/Jun	100%
10	Select models implement	11/Jun	17/Jun	100%
11	Select the dataset	11/Jun	17/Jun	100%
12	Establish training meth...	02/Jul	08/Jul	100%
13	Select framework for d...	02/Jul	08/Jul	100%
14	Define evaluation metri...	02/Jul	08/Jul	100%
15	Draft report writeup	11/Jun	15/Jul	100%
	Feature prototype	18/Jun	15/Jul	100%
17	Vision Transformer imp...	18/Jun	24/Jun	100%
18	Malnet dataset adaptati...	18/Jun	24/Jun	100%
19	Test training and visuali...	09/Jul	15/Jul	100%
	Experiment pipeline and final r...	16/Jul	09/Sep	0%
	Finalize model implementat...	16/Jul	22/Jul	0%
22	Masked self-attention ViT	16/Jul	20/Jul	0%
23	Training code refactoring	21/Jul	22/Jul	0%
	Training pipeline	23/Jul	02/Sep	0%
25	ViT-VS	23/Jul	29/Jul	0%
26	ResNet18	30/Jul	05/Aug	0%
27	MobileNetV2	06/Aug	12/Aug	0%
28	ViT-B/16	13/Aug	19/Aug	0%
29	ViT-SHERLOCK	20/Aug	26/Aug	0%
30	ViT-V	27/Aug	02/Sep	0%
31	Model evaluation	03/Sep	04/Sep	0%
32	Evaluation of findings and fi...	05/Sep	09/Sep	0%

Apr 2023
May 2023
Jun 2023
Jul 2023
Aug 2023
Sep 2023
Oct 2023
14
Preliminary report & prototype
Project pitch
Select topic
Establish scope
Literature review
Research relevant work
Write the review
Select 5 pieces for review
Project design
Define domain and stakeholder requirements
Select models implement
Select the dataset
Establish training methodology
Select framework for development
Define evaluation metrics
Draft report writeup
Feature prototype
Vision Transformer implementation
Malnet dataset adaptation for TF
Test training and visualization
Experiment pipeline and final report
Finalize model implementation
Masked self-attention ViT
Training code refactoring
Training pipeline
ViT-VS
ResNet18
MobileNetV2
ViT-B/16
ViT-SHERLOCK
ViT-V
Model evaluation
Evaluation of findings and final report

Figure 1: Initial work plan

4. Implementation

4.1. Implementation overview

The model training pipeline is implemented in Jupyter Labs as a jupyter notebook as sequence of python-kernel cells which are executed in (or out of) sequence.
The notebook is structured in the following way:

1. Library imports
2. Data preprocessing and augmentation functions
3. Definition of model components: MLP, patches, MAE
4. Model constructors: ViT-V, ViT-B/16, ViT-SHERLOCK, ViT-VS, imports of pre-built models
5. Model building and validation, training utility functions
6. Experiment setup: training and evaluation routines

The final cells is used to run all the functions outlined above by selecting the name of the model to be run, setting the configuration for the dataset and training options, and saving the results.
The most important aspects of the code in each of the above sections are outlined above.

4.2. Libraries used

The software libraries used in the full implementation of the project are largely as described in 3.4.2. However, there are additional libraries that where included after the initial prototyping phase, namely 'scipy', which was used for its' implementation of interpolation function, and 'Classifiers' python library, which provided pre-built version of ResNet18, which is not included in Keras or TensorFlow.

4.3. Data preprocessing

The input size for all models is set to 224x224. This is done in order to achieve a more uniform comparison of the models from the referenced papers [3] and of pre-built CNN models (ResNet18, MobileNetV2), as those were originally designed for this input size.
This encompasses headings 'Prepare the data', 'Data augmentation' and 'Create image generators'. The source code for functions adapted from the companion repository to [1] at https://github.com/safreita1/malnet-image/tree/master.
Since the MalNet dataset is not included in standard libraries, the files were downloaded and stored locally. In order to use them for training they are loaded into TensorFlow generator objects. During the initial implementation phase, the Mixup data augmentation routine was implemented as a custom wrapper around ImageGenerator class. Mixup is a technique that works by creating new training examples through a linear interpolation between random pairs of existing training samples. It is formalized as follows:

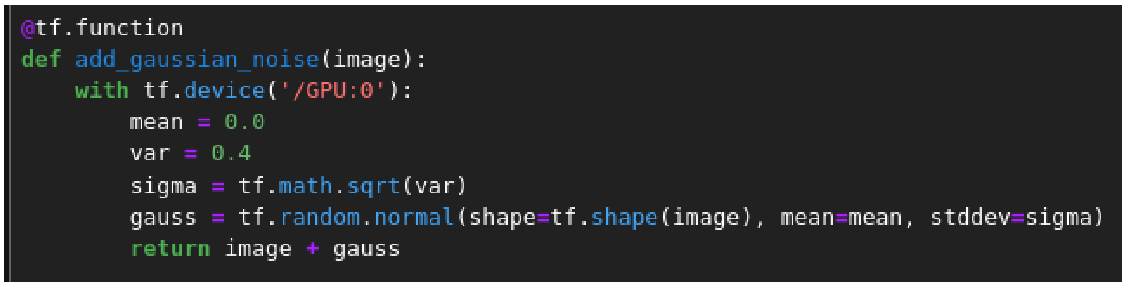
$$x_{mix} = \lambda \cdot x_1 + (1 - \lambda) \cdot x_2$$

$$y_{mix} = \lambda \cdot y_1 + (1 - \lambda) \cdot y_2$$

Where x_{mix} and y_{mix} are mixed respective inputs and labels, λ controls the contribution of each example to the mixed sample. If λ=1, then the mixed sample is identical to the first example; if λ=0, it is identical to the second example.
However, after a series of test I've found that samples created using class interpolation caused the model to learn erroneous features, which resulted in low model performance. This suggests that Mixup is an unsuitable choice of data augmentation technique for image-based malware classification.

Gaussian noise is another data augmentation technique that was tried during the testing phase. It is a type of noise that following the Gaussian distribution. The idea is to accustom the model to random permutations and distortions in the training data. The formal definition of probability density of Gaussian noise is defined by the following formula:

$f(x|\mu,\sigma^2)=1/(\sigma\sqrt{2\pi})e^{-(x-\mu)^2/(2\sigma^2)}$



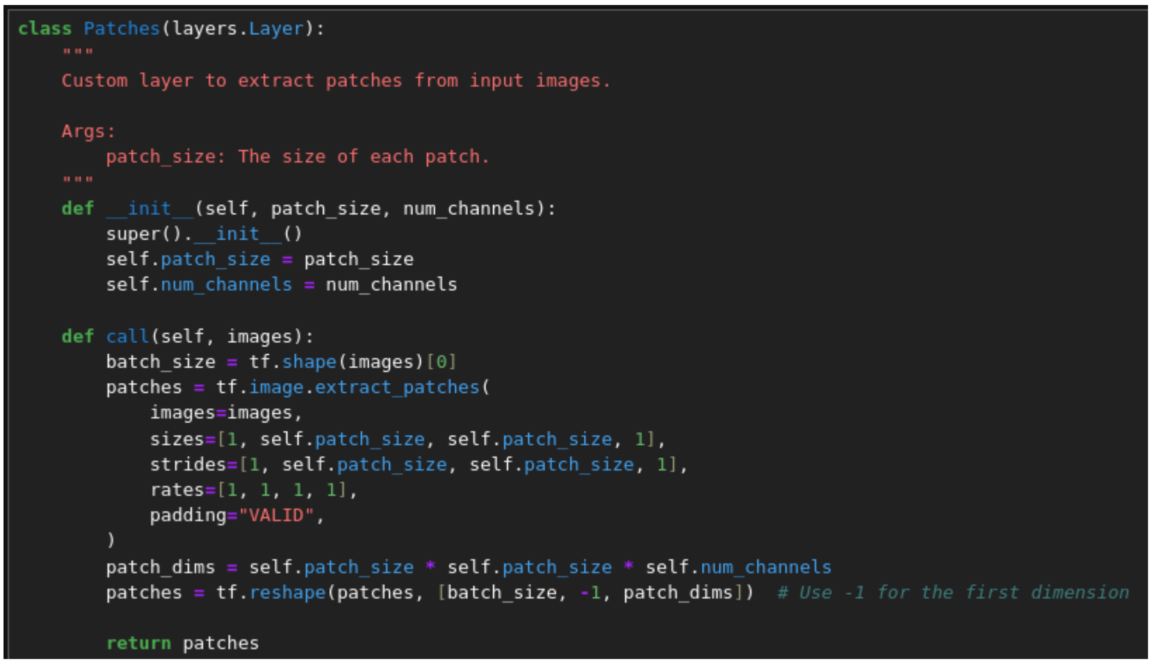
```python
@tf.function
def add_gaussian_noise(image):
    with tf.device('/GPU:0'):
        mean = 0.0
        var = 0.4
        sigma = tf.math.sqrt(var)
        gauss = tf.random.normal(shape=tf.shape(image), mean=mean, stddev=sigma)
        return image + gauss
```

Figure 2: Implementation of Gaussian noise

The implementation of Gaussian noise in code is shown in [Figure 2]. The variance parameter was chosen according to results in [19], since it was the value that provided the best results for the individual application of the Gaussian noise. The Laplacian and Poisson distributions were truncated due to using all of them in conjunction resulted in too much computational overhead to finish the project on deadlines. Performance testing with and without Gaussian noise has revealed no performance benefits for tasks presented in this project. Due to this, all models were trained without applying the Gaussian noise in order to reduce training time.

4.4. Model components

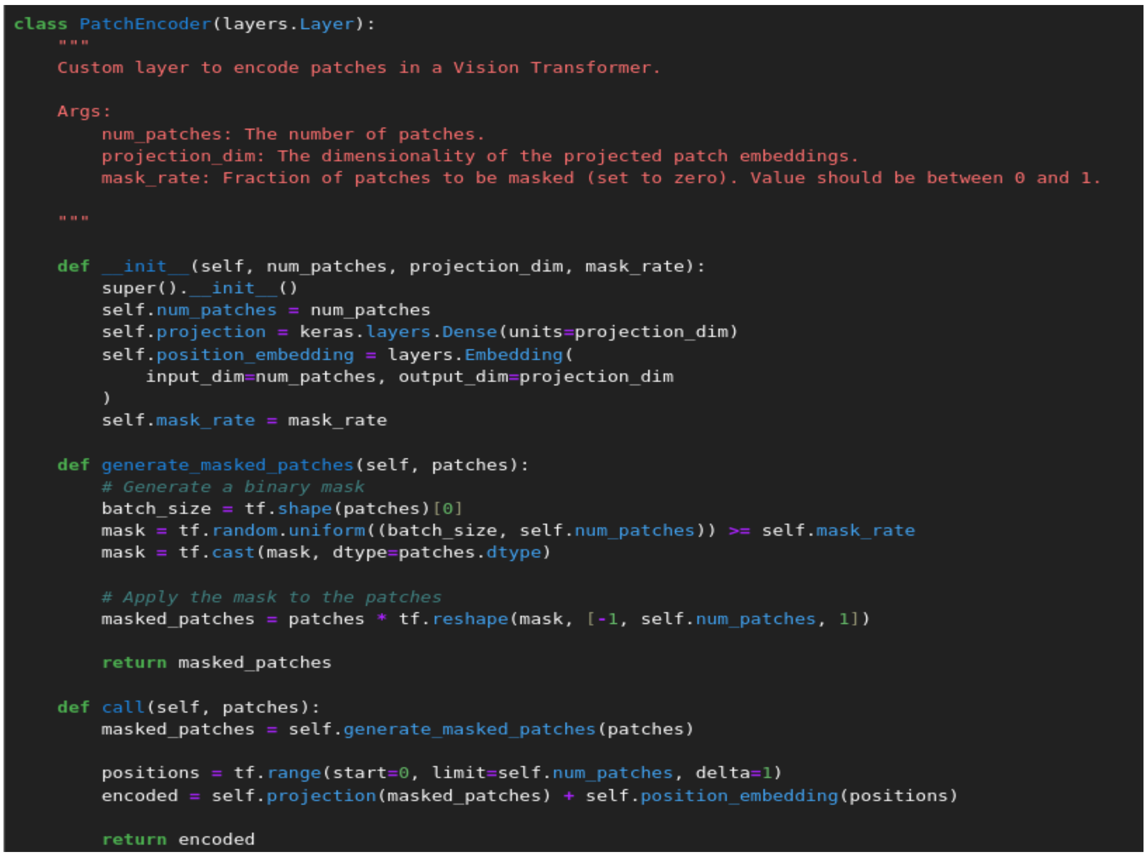
Since the project deals mostly with evaluation of behavior of Vision Transformer models in various conditions, the implementation of key components is done manually for the purposes of study. In Vision Transformers, the main processing element of the model is a standard transformer block used in conjunction with a special layer that converts the input image into a sequence of patches a predetermined size. The input processing layer splits the images in the input batch into patches using Tensorflow's built-in 'extract_patches' method as shown in [Figure 3].



```python
class Patches(layers.Layer):
    """
    Custom layer to extract patches from input images.

    Args:
        patch_size: The size of each patch.
    """
    def __init__(self, patch_size, num_channels):
        super().__init__()
        self.patch_size = patch_size
        self.num_channels = num_channels

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = self.patch_size * self.patch_size * self.num_channels
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])  # Use -1 for the first dimension

        return patches
```

Figure 3: Implementation of 'Patches' layer

Each patch of the input image is then vectorized using the fully connected ‘layers.Dense’ and assingned a relative position in the image using ‘layers.Embedding’. Their outputs are summed together to produce an encoding for individual patch.



```python
class PatchEncoder(layers.Layer):
    """
    Custom layer to encode patches in a Vision Transformer.

    Args:
        num_patches: The number of patches.
        projection_dim: The dimensionality of the projected patch embeddings.
        mask_rate: Fraction of patches to be masked (set to zero). Value should be between 0 and 1.

    """

    def __init__(self, num_patches, projection_dim, mask_rate):
        super().__init__()
        self.num_patches = num_patches
        self.projection = keras.layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )
        self.mask_rate = mask_rate

    def generate_masked_patches(self, patches):
        # Generate a binary mask
        batch_size = tf.shape(patches)[0]
        mask = tf.random.uniform((batch_size, self.num_patches)) >= self.mask_rate
        mask = tf.cast(mask, dtype=patches.dtype)

        # Apply the mask to the patches
        masked_patches = patches * tf.reshape(mask, [-1, self.num_patches, 1])

        return masked_patches

    def call(self, patches):
        masked_patches = self.generate_masked_patches(patches)

        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(masked_patches) + self.position_embedding(positions)

        return encoded
```

Figure 4: Implementation of 'PatchEncoder' layer

In order to reproduce the self-supervised training technique used in [3] for ViT-SHERLOCK, the ‘PatchesEncoder’ base from [14] was modified to incorporate masking of individual patches with ‘generate_masked_patches’ method.

The core all Vision Transformer models is the transformer block, which is visualized in [Figure 5] and implemented as a custom layer as shown in [Figure 6] The main component of the transformer block is multi-head attention layer. Each “head” independently takes the sequence of patches as input and calculates the attention scores for every patch. The results are concatenated and then linearly transformed into the projection of size ‘projection_dim’.

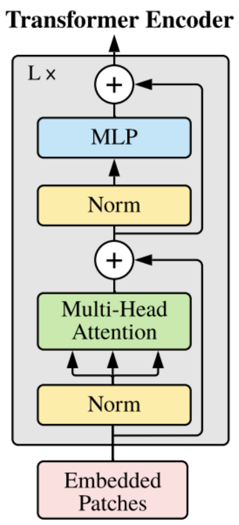
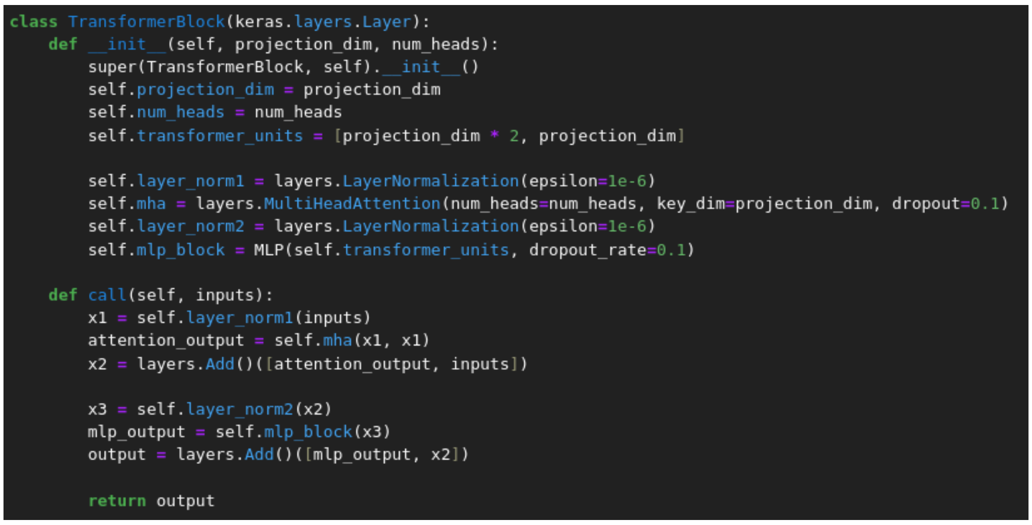


Figure 5: Transformer block from [4]



```python
class TransformerBlock(keras.layers.Layer):
    def __init__(self, projection_dim, num_heads):
        super(TransformerBlock, self).__init__()
        self.projection_dim = projection_dim
        self.num_heads = num_heads
        self.transformer_units = [projection_dim * 2, projection_dim]

        self.layer_norm1 = layers.LayerNormalization(epsilon=1e-6)
        self.mha = layers.MultiHeadAttention(num_heads=num_heads, key_dim=projection_dim, dropout=0.1)
        self.layer_norm2 = layers.LayerNormalization(epsilon=1e-6)
        self.mlp_block = MLP(self.transformer_units, dropout_rate=0.1)

    def call(self, inputs):
        x1 = self.layer_norm1(inputs)
        attention_output = self.mha(x1, x1)
        x2 = layers.Add()([attention_output, inputs])

        x3 = self.layer_norm2(x2)
        mlp_output = self.mlp_block(x3)
        output = layers.Add()([mlp_output, x2])

        return output
```

Figure 6: Transformer block implementation

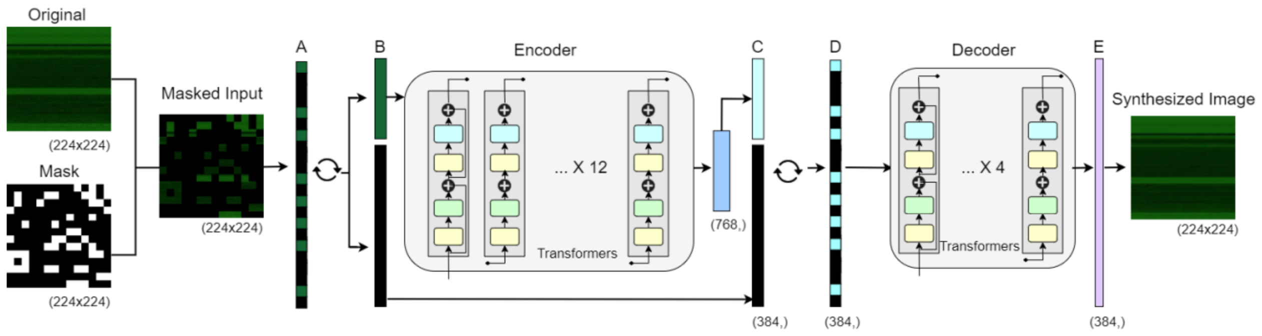
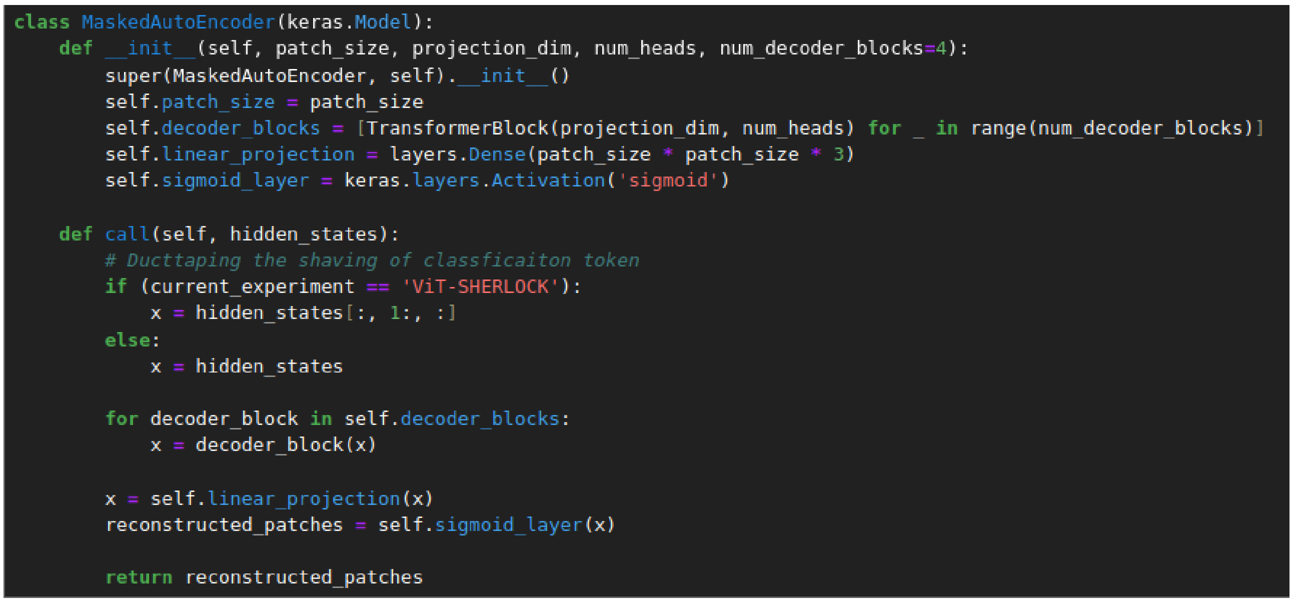


Figure 7: SHERLOCK model overview as shown in [3]

To complete the implementation of SHERLOCK [Figure 7] from [3], the 'MaskedAutoEncoder' layer was implemented as in [Figure 8].



```
class MaskedAutoEncoder(keras.Model):
    def __init__(self, patch_size, projection_dim, num_heads, num_decoder_blocks=4):
        super(MaskedAutoEncoder, self).__init__()
        self.patch_size = patch_size
        self.decoder_blocks = [TransformerBlock(projection_dim, num_heads) for _ in range(num_decoder_blocks)]
        self.linear_projection = layers.Dense(patch_size * patch_size * 3)
        self.sigmoid_layer = keras.layers.Activation('sigmoid')

    def call(self, hidden_states):
        # Ducttaping the shaving of classficaiton token
        if (current_experiment == 'ViT-SHERLOCK'):
            x = hidden_states[:, 1:, :]
        else:
            x = hidden_states

        for decoder_block in self.decoder_blocks:
            x = decoder_block(x)

        x = self.linear_projection(x)
        reconstructed_patches = self.sigmoid_layer(x)

        return reconstructed_patches
```

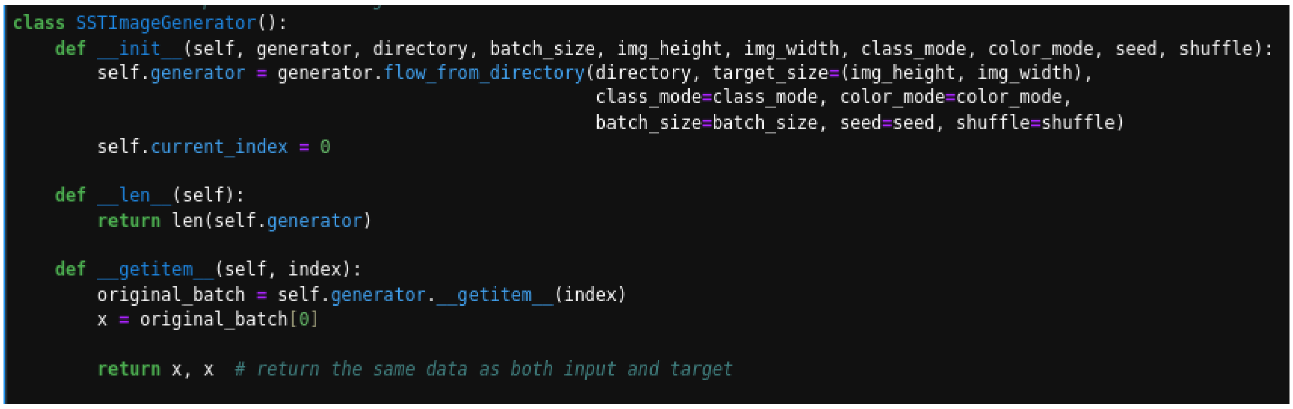
Figure 8: Masked Auto Encoder

The function of this layer is to reconstruct the patches from the embeddings created by the encoder part (which is also a series of transformer blocks) using a number (4 in this case as per [3]) of decoder blocks, which are transformer blocks themselves. The goal here is to calculate the difference in the form of mean error over images provided as ground truth and reconstructed images. As the error is back-propagated through the model, it forces the encoder part to learn efficient representations of features, which are then utilized by fine-tuning on specific classification task.

The above listed components are shared among the tested models, with Masked Auto Encoder applying to ViT-SHERLOCK and ViT-VS only. In addition to these, the ResNet18, MobileNetV2 and ViT/B-16 were imported from third-party libraries. ViT-SHERLOCK also uses weights imported from ViT/B-16.

Additionally, the implementation of self-supervised training for ViT-SHERLOCK and ViT-VS required a number of custom components, specifcially, the custom 'MSELoss' which calculates the mean error over pixels between original sample and sample synthesized by the masked auto encoder, 'SSTGenerator' which produces only X-side data due to measure of efficiency being closeness to the original input.

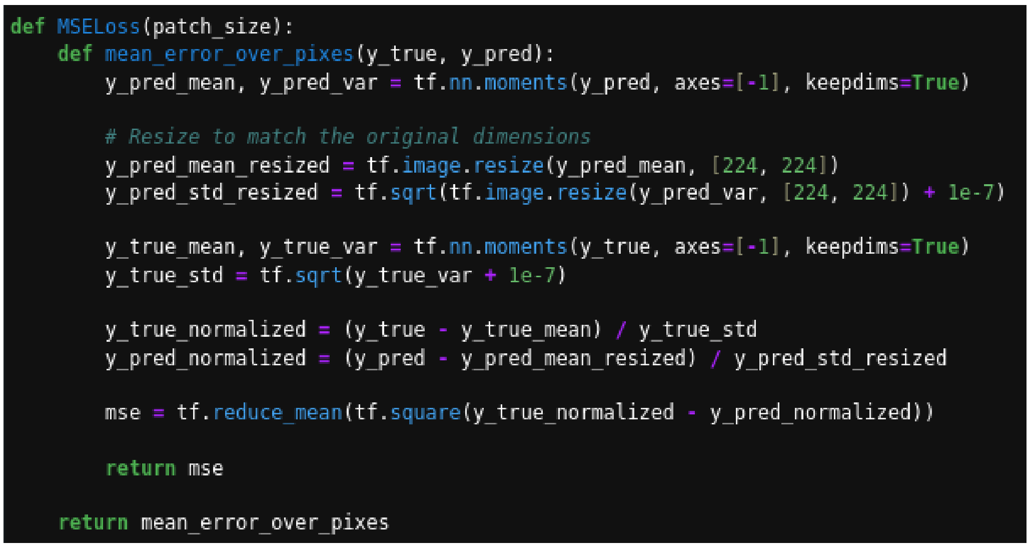
The main part of ‘SSTGenerator’ is show on [Figure 9]. The class creates a wrapper object for the standard Keras ImageGenerator, where the behavior of ‘__getitem__’ method is altered to return only the X side of the (x,y) pair.



```python
class SSTImageGenerator():
    def __init__(self, generator, directory, batch_size, img_height, img_width, class_mode, color_mode, seed, shuffle):
        self.generator = generator.flow_from_directory(directory, target_size=(img_height, img_width),
                                                       class_mode=class_mode, color_mode=color_mode,
                                                       batch_size=batch_size, seed=seed, shuffle=shuffle)
        self.current_index = 0

    def __len__(self):
        return len(self.generator)

    def __getitem__(self, index):
        original_batch = self.generator.__getitem__(index)
        x = original_batch[0]

        return x, x  # return the same data as both input and target
```

Figure 9: Part of the ImageGenerator wrapper class forself-supervised training

The ‘MSELoss’ class is a wrapper around a custom loss function ‘mean_error_over_pixels’, which is implemented following the description from [3] as shown in [Figure 10].



```python
def MSELoss(patch_size):
    def mean_error_over_pixes(y_true, y_pred):
        y_pred_mean, y_pred_var = tf.nn.moments(y_pred, axes=[-1], keepdims=True)

        # Resize to match the original dimensions
        y_pred_mean_resized = tf.image.resize(y_pred_mean, [224, 224])
        y_pred_std_resized = tf.sqrt(tf.image.resize(y_pred_var, [224, 224]) + 1e-7)

        y_true_mean, y_true_var = tf.nn.moments(y_true, axes=[-1], keepdims=True)
        y_true_std = tf.sqrt(y_true_var + 1e-7)

        y_true_normalized = (y_true - y_true_mean) / y_true_std
        y_pred_normalized = (y_pred - y_pred_mean_resized) / y_pred_std_resized

        mse = tf.reduce_mean(tf.square(y_true_normalized - y_pred_normalized))

        return mse

    return mean_error_over_pixes
```

Figure 10: Custom loss 'Mean error over pixels’

The ‘mean_error_over_pixels’ function computes the mean squared error image tensors after normalizing them. The mean and variance of ‘y_pred’ is calculated across its last axis, and then resized to dimensions [224, 224] for input size compliance. The standard deviation is derived from the variance with a small constant added to avoid division by zero. The mean and standard deviation of true labels are computed in similar manner. Both tensors are then normalized using their respective means and standard deviations. The function finally computes the error as the average of the squared differences between the normalized tensors.

5. Evaluation

5.1. Data processing evaluation

The initial testing phase has revealed several flaws in the original work plan. First, during workload estimation, the computational overhead of preprocessing each input image was not considered. The

plan initially included application of three distinct type of noise, in addition to using mixup sample generation. During initial runs, it became apparent that using three types of noise is unfeasible considering project constraints, as all three types of noise have increased the average batch processing time by 250% (from 20 minutes on average to 50 minutes). Additional inclusion of mixup further increased processing by 20%.
Manual testing of individual noise filtering revealed no significant effect on the targeted evaluation metrics (macro F-1 score, macro-recall and macro-precision), which led to decision to omit their use during final model runs.
Using Mixup yielded decrease in performance and quickly caused models to overfit. This suggests that using interpolation is not a suitable augmentation technique for the class of problems being studied, likely because the byte-images represent strict sequences of data, and partially replacing some sequences with others from a different sample may produce non-meaningful samples.
Adding Gaussian noise as an augmentation technique yielded no tangible performance benefits. A possible reason for that is the difference of RGB channel assignment schemes in [1] and [19] (for example, in [1] the authors assign channels depending on their position in the binary, while the researches in [19] assign red channel with decimal values of each byte, green channel with entropy values and leave blue as a zero channel).
As the dataset is heavily imbalanced, the class weights are computed based the effective number of samples as proposed in [1] and [3]. This had the most positive effect on per-class performance, as using the default class weights invariably led to models unable to correctly identify any classes with less than 10 000 samples.

5.2. Hyperparameter validation

In addition to the above mentioned investigation of preprocessing techniques, the feasibility of investigating a significant search space of hyperparameter values was limited by variance in computational cost of training of models. For the testing phase, only a single Vision Transformer model at ~24M parameters was implemented and tested only on ‘type’ classification task. However, this model is significantly smaller than ViT-B/16 at ~87M parameters.
Considering that the training cost per epochs scales linearly (using the setup as defined in section 3.4.1), I’ve decided to omit the validation stage in favor of manual targeted testing using specific recommendations in referenced material on a downsized version of the dataset.
To ensure that all models are investigated to address the main research questions outlined in the project, the training time for all models is set uniformly to 30 epochs per task. For models trained using the self-supervised approach, the epochs were capped at 60 for pre-training phase and 10 for each classification tasks. This allowed to infer whether obtaining the condensed features of byte-images would have any benefits resource-wise compared to training the models directly from scratch. The models were trained using a dynamic reduction of learning rate, which would decrease it by a factor of 10 every 5 epochs if validation loss stagnated, as well as an early stopping callback, which terminated training process if model did not improve for 10 consecutive epochs.
A full-scale hyperparameter search was feasible using a GPU cluster provider, however, due to niche use of the studied dataset, the logistics of moving the dataset for execution and processing in the cloud made it an unviable option for the timeline of the project.

5.3. Evaluation metrics

All metrics were kept as outlined in the initial design, with the exception of inference time, which was found to be inconsistent and dependent on variable factors such as CPU and GPU performance. Due to this, I’ve decided to omit it from the final evaluation as an inconsistent way to measure model performance compared to the number of FLOPS per forward pass.

5.4. Model evaluation

The results of training and testing six models outlined in 3.2. are summarized in the Table 1. The raw performance is not competitive with results achieved in [1], [3] and [4], which stems from a limited computational budget and possibly suboptimal hyperparameter choice. However, this does not prevent the direct comparison of architectures to address the research questions of the project.

Table 1: Model results per classification task

Model	**GFlops**	**Macro F-1**			**Macro Precision**			**Macro Recall**		
		Binary	Type	Family	Binary	Type	Family	Binary	Type	Family
MobileNetV2	19.2	0.76	0.26	0.14	**0.98**	0.27	0.13	0.69	**0.43**	**0.40**
ResNet18	935.6	0.78	0.13	0.19	**0.98**	0.21	0.17	0.71	0.17	0.38
ViT-V	175.8	**0.79**	**0.31**	0.09	0.96	**0.37**	0.10	**0.72**	0.42	0.30
ViT-B/16	469.4	0.48	0.20	**0.24**	0.47	0.27	**0.25**	0.50	0.22	0.28
ViT-VS	558.3	0.75	0.27	0.07	0.97	0.35	0.07	0.68	0.40	0.24
ViT-SHERLOCK	2254.7	0.48	0.24	0.16	0.47	**0.37**	0.21	0.5	0.28	0.16

The above results show that ViT-V generally outperforms other models on Macro-F1 scores on binary and type classification tasks, but underperforms on family classification. MobileNetV2 has similar performance and better family task metrics, while requiring less computational resources than any other model tested. ResNet18 offers slightly better performance on binary classification tasks, but is significantly worse than MobileNetV2 for type and family classification, in addition to requiring much more computational resources, however, this may be possible because it was imported from a third-party library.
ViT-B/16 and ViT-SHERLOCK have the worst overall performance, however, ViT/B-16 does better than any other model on family macro F-1 score and macro-precision. This may suggest that there is some marginal benefit in transfer learning from weights pre-trained on tasks with large number of classes (family task involves 696 classes, while type task is 49 classes). ViT/B-16 is also significantly larger than ViT-V, and training it was only possible with a batch size of 64. The same is the case for ViT-SHERLOCK, due to it using an encoder part similar to ViT/16, while also incorporating a decoder part with 4 additional layers, which resulted in reduction of number of attention heads to 6, compared to 12 in the original. The resources required for ViT-SHERLOCK are also much greater than any other model investigated.
ViT-VS offers performance that is generally on par with ViT-V on all classification tasks, while ViT-SHERLOCK replica model is similar to ViT-B/16. Since both ViT-SHERLOCK and ViT-B/16 used the same initial weights, and ViT-VS and ViT-V were trained from scratch, it is reasonable to assume that the performance drop is due to use of pre-trained weights.
The average duration of training for supervised models is 15-20 epochs per each task before termination on loss plateau. Self-supervised models were trained first for 34 epochs (ViT-VS) and 11 epochs (ViT-SHERLOCK), and then further fine-tuned for up to 10 epochs on each task. This result supports the idea presented in [3] that self-supervised training may be used as a way to optimize training time for models.

The results from Table 1, the Jupyter notebook containing the code, as well as extended reports on model training and architecture are available for the following repository: [Github]

6. Conclusions

Based on the results of experiments shown above, it appears that large-scale pre-training does not transfer well to malware classification problems. This is evident from poor performance of both ViT-B/16 and ViT-SHERLOCK, which use initial weights from pre-training on ImageNet-1K, compared to other models. This highlights the need for more large-scale datasets of malware byte-image samples in the public domain, as most datasets available have very few samples, especially in context of training transformer models, and can even become obsolete with time, as malware strains evolve constantly.

The ViT-V Vision Transformer outperforms MobileNetV2 on macro-F1 score for binary and type tasks, but the improvements are marginal. This suggests that CNN models are more still more suitable for use in consumer applications due to their lower computational requirements, while the use of ViT variants may find more utility in enterprise and research settings. The ResNet18 model is similar to MobileNetV2, however required much more FLOPs for forward pass, although this may have been due to specifics of the model implementation in the library used.

The results of testing the ViT-VS and ViT-SHERLOCK models suggest that there is no significant performance boost compared to training the models from scratch on each individual task. However, the fact that the encodings for each of these models were obtained from training for 34 and 11 epochs respectively and then fine-tuned for only up to 10 epochs for each task, does show that there is a benefit in terms of saving computational budget and training time, which makes this approach a viable option for security researches and organizations who can take a foundation model pre-trained on a larger scale dataset and adapt it for their specific tasks. This approach could be especially effective for smaller companies or individual researchers that may not have the resources required to train a large model.

The results shown in 5.4. are given from the evaluation of each model on the MalNet dataset. In the introduction to the project, I've also proposed an evaluation of these models on the Virus-MNIST dataset. However, due to the significant discrepancy between both the form and the semantic content of two datasets (Virus-MNIST contains byte-images of first 1024 bytes of the PE header), the evaluation was scrapped from the project as unlikely to have given any meaningful findings. This subject, however, could be a topic of further expansion of the project, for example, a study of differences between training process and efficiency of selected array of models on binaries from various operating systems. Another possible extension is the creation of mixed-platform dataset as tool to investigate wether it is possible to train models to learn more abstract features of malware across different operating systems.

While the overall performance of the models most likely have been impacted by the lack of full-scale parameter validation, the overall tendencies are consistent with what is shown in the prior work. As is, the project offers a direct comparison of transfer learning and training of Vision Transformer models from scratch in malware classification domain, which, although mentioned, was not explicitly performed in [1] and [3].

7. References

[1] Scott Freitas, Rahul Duggal, and Duen Horng Chau. 2022. MalNet: A large-scale image database of malicious software. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2102.01072

[2] Muhammad Furqan Rafique, Muhammad Ali, Aqsa Saeed Qureshi, Asifullah Khan, and Anwar Majid Mirza. 2020. Malware classification using Deep Learning based feature extraction and wrapper based feature selection technique. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/1910.10958

[3] Sachith Seneviratne, Ridwan Shariffdeen, Sanka Rasnayaka, and Nuran Kasthuriarachchi. 2022. Self-supervised Vision Transformers for malware detection. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2208.07049

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, et al. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2010.11929

[5] Ricky Xu. 2021. Training vision transformers from scratch for malware classification. Medium. Retrieved July 13, 2023 from https://medium.com/codex/training-vision-transformers-from-scratch-for-malware-classification-ccdae11d7236
[6] David Noever and Samantha E. Miller Noever. 2021. Virus-MNIST: A benchmark malware dataset. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2103.00602
[7] Andreas Steiner, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. 2022. How to train your vit? data, augmentation, and regularization in Vision Transformers. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2106.10270
[8] Nataraj, Lakshmanan & Karthikeyan, Shanmugavadivel & Jacob, Grégoire & Manjunath, B.. (2011). Malware Images: Visualization and Automatic Classification. 10.1145/2016904.2016908.
[9] Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/1711.05101
[10] Ahmed Bensaoud, Nawaf Abudawaood, and Jugal Kalita. 2020. Classifying malware images with convolutional neural network models. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2010.16108
[11] 95% of all new malware target windows in 2022 - atlas VPN. atlasVPN. Retrieved July 13, 2023 from https://atlasvpn.com/blog/over-95-of-all-new-malware-threats-discovered-in-2022-are-aimed-at-windows
[12] AndroZoo. https://androzoo.uni.lu/
[13] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE.
[14] Keras Team. Keras documentation: Image Classification with Vision Transformer. Keras: https://keras.io/examples/vision/image_classification_with_vision_transformer/
[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv.org*. Retrieved July 13, 2023 from https://arxiv.org/abs/1512.03385
[16] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted residuals and linear bottlenecks. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/1801.04381
[17] Kyoung-Won Park, Department of Artificial Intelligence, et al. 2022. A Vision Transformer enhanced with patch encoding for malware classification: Intelligent Data Engineering and automated learning – ideal 2022. Guide Proceedings. Retrieved July 13, 2023 from https://dl.acm.org/doi/abs/10.1007/978-3-031-21753-1_29
[18] Petroc Taylor. 2023. Global Mobile OS Market Share 2023. Statista. Retrieved July 13, 2023 from https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
[19] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, and Zahid Hussain Khand. 2020. Data augmentation based malware detection using Convolutional Neural Networks. arXiv.org. Retrieved July 13, 2023 from https://arxiv.org/abs/2010.01862