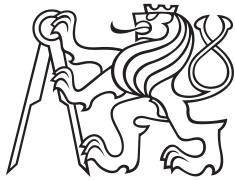


Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Deep Learning Based Malware Detection from Weakly Labeled URLs

Bc. Vít Zlámal

Supervisor: Ing. Jan Brabec
May 2020

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zlámal** Jméno: **Vit** Osobní číslo: **423305**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávající katedra/ústav: **Katedra počítačů**
Studijní program: **Otevřená informatika**
Specializace: **Kybernetická bezpečnost**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Detekce malwaru ze slabě označených URL pomocí metod hlubokého učení

Název diplomové práce anglicky:

Deep learning based malware detection from weakly labeled URLs

Pokyny pro vypracování:

The thesis addresses a problem of malicious communication detection from URLs extracted from network telemetry (proxy logs, enriched NetFlows). The main objectives are to create representation of URLs with corresponding neural network architecture and to utilize multiple sources of labels with varying degree of certainty for training.

The concrete goals are:

1. Learn about Deep Learning from textbook [1]. Review the prior art in classification of URLs (or similar problems) with neural networks and select applicable and relevant methods based on the review.
2. At first, focus on a fully supervised problem with a single source of labels. Use knowledge from the review to create a classifier for URLs and evaluate it's efficacy on a dataset of sufficient size originating from real network telemetry (dataset will be provided by supervisor).
3. Review the prior art in learning under weak supervision and select or modify methods that can be used in conjunction with the classifier created in step (2).
4. Design a scheme to combine multiple sources of ground truth (blacklists, results of other algorithms, ...) with varying confidence into weak labels and extend the classifier from step (2) to allow training in a weakly supervised manner.
5. Evaluate the results on a representative real-world dataset (will be provided by supervisor). Compare relevant alternatives and investigate the difference in efficacy between the fully-supervised and weakly-supervised approach.

Seznam doporučené literatury:

- [1] Aston Zhang, Zachary C. Lipton, Mu Li, & Alexander J. Smola (2020). Dive into Deep Learning. (<https://d2l.ai>)
- [2] Dehghani, M., Severyn, A., Rothe, S., & Kamps, J. (2017). Avoiding your teacher's mistakes: Training neural networks with controlled weak supervision. arXiv preprint arXiv:1711.00313.
- [3] Saxe, J., & Berlin, K. (2017). eXpose: A character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys. arXiv preprint arXiv:1702.08568.
- [4] Ishida, T., Niu, G., & Sugiyama, M. (2018). Binary classification from positive-confidence data. In Advances in Neural Information Processing Systems (pp. 5917-5928).
- [5] Franc, V., Sofka, M., & Bartos, K. (2015, September). Learning detector of malicious network traffic from weak labels. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases (pp. 85-99). Springer, Cham.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Brabec, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2020**

Termín odevzdání diplomové práce: **22.05.2020**

Platnost zadání diplomové práce: **30.09.2021**

Ing. Jan Brabec
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

Acknowledgements

I would like to thank the following people who have helped me undertake this research:

My supervisor Ing. Jan Brabec for his guidance, the advice he provided me during the last years and patience;

The whole Cognitive intelligence team and Cisco systems for the opportunity to work on this great project;

My family for providing me calm and study enabling environment since my childhood;

My loving girlfriend Zůza for her support.

And Martin R. with Karel B., who brought me to the cybersecurity field.

Declaration

I declare that I have developed the presented work independently and that I have listed all information sources used in accordance with the Methodical guidelines on maintaining ethical principles during the preparation of higher education theses.

In Prague, May 2020

.....

Abstract

In recent years, machine learning-based approaches are becoming a fundamental part of cybersecurity products to keep up with the growing number of cyber threats. In this thesis, we present the pipeline for large scale training and distributed evaluation of neural network models which is suitable for industrial use in Cisco Cognitive Intelligence production environment. We focused on the classification of URLs on the real world positive unlabeled dataset that originates in Cisco network telemetry with ratio 1 to 1500 between 25 positive classes and one unlabeled class. The whole model's life cycle can be managed by one task in the cloud service.

The second part of the thesis introduces a convolutional neural network architecture which uses information from untrusted sources as weak labels for identifying positive samples in the unlabeled part of the dataset and thus bringing valuable information into the training process.

Keywords: neural networks, convolution, imbalanced dataset, positive unlabeled, MXNet, weak labels, classification, malware

Supervisor: Ing. Jan Brabec
Cisco Systems, Karlovo nám. 10, 120 00
Nové Město

Abstrakt

Strojové učení se v posledních letech stalo nepostradatelným nástrojem v boji s rostoucí kyberkriminalitou. V rámci této diplomové práce jsme implementovali strukturu na trénování neuronových sítí s velkým množstvím dat a distribuovaný evaluačním systém, který je možné použít v produkčním prostředí produktu Cognitive intelligence od firmy Cisco. Zaměřili jsme se především na klasifikaci URL adres, které jsme získali ze síťové telemetrie společnosti Cisco. Tento dataset z reálné praxe se vyznačuje tím, že jedna jeho část je označena jako pozitivní, zatímco ta druhá obsahuje neoznačené záznamy, a také vysokou měrou imbalance v měřítku 1500 ku 1 mezi 25 pozitivními třídami a jednou neoznačenou třídou. Celý životní cyklus modelu může být obstarán pomocí jednoho příkazu v claudovém systému.

V druhé části práce představujeme architekturu konvoluční neuronové sítě, která využívá informace z neověřených zdrojů ve formě slabého označení našich vzorků. Toto označení se následně využívá při tréninku klasifikátoru k odhalení pozitivních vzorků v neoznačené části dat. Tento proces nám umožňuje vnést více informace do trénovacího procesu a tím zlepšit jeho efektivitu.

Klíčová slova: neuronové sítě, konvoluce, nevyvážený dataset, pozitivní a neoznačená data, MXNet, slabé signály, klasifikace, malware

Překlad názvu: Detekce malwaru ze slabě označených URL pomocí metod hlubokého učení

Contents

1 Introduction	1	7.1.2 TensorFlow	30
2 Supervised learning	3	7.1.3 MXNet	30
2.1 Classification	3	7.2 Python part	30
2.2 Positive unlabeled data	4	7.2.1 Data loading	31
2.3 Overparametrized models	5	7.2.2 Data providing	31
2.3.1 Double descent	5	7.2.3 Training loop	32
2.4 Evaluation metrics	6	7.3 Java part	32
2.4.1 Recall	7	7.3.1 Inference and Evaluation....	33
2.4.2 Specificity	7	8 Experiments	35
2.4.3 Accuracy	7	8.1 Dataset description	35
2.4.4 Precision	8	8.1.1 Malware classes	36
3 Neural nets	9	8.2 Fully supervised model	37
3.1 Classification with neural		8.2.1 Experiments with the number	
networks	10	of convolution filters	37
3.1.1 Softmax	10	8.2.2 Double descent experiments .	38
3.1.2 Cross-Entropy loss	11	8.2.3 Excluding the hostnames....	39
3.1.3 Adam	11	8.3 Weakly labelled model	39
3.2 Deep neural networks	12	8.3.1 Weighting of positive class ..	40
3.3 Convolution neural nets (CNN) .	14	8.3.2 Semi-supervised scenario	40
3.3.1 Convolution layer	15	9 Conclusion	45
3.3.2 Pooling	15	Bibliography	47
3.4 Regularization	16	A CD content	51
3.4.1 L2 regularization	16		
3.4.2 Dropout	16		
3.4.3 Implicit regularization of			
gradient descent	17		
3.4.4 Batch normalization	18		
3.5 Sequence models	19		
3.5.1 Recurrent neural networks ..	19		
3.5.2 Long Short Term Memory ...	20		
3.5.3 Transformer	20		
4 Classification of URLs	21		
4.1 Neural network models	21		
4.1.1 URLnet	22		
4.2 Other classification approaches .	22		
5 Fully supervised model	23		
5.1 Data prepossessing	23		
5.2 Architecture	23		
5.3 Hyperparameters	24		
6 Weakly labeled model	27		
6.1 Model with weak labels	28		
7 Infrastructure	29		
7.1 Frameworks	29		
7.1.1 PyTroch	29		

Figures

2.1 Double descent risk curve. Figure adopted from [5].	6
3.1 Linear neural network model with 4 inputs and 3 outputs.	11
3.2 Neural network model with one hidden layer.	14
3.3 Cross-correlation operation with input on the left, kernel in the middle and output on the right.	15
3.4 Max-pooling operation example.	15
3.5 Neural network before dropout is on the left and neural network after applying dropout is on the right.	17
5.1 Encoding of the URL into 95×100 one-hot representation.	24
5.2 The architecture of the neural network. Whit two convolutional layers one with the kernel width 4 and the second with kernel width 5. The number of kernels is discussed in Chapter 8. Convolution is followed by max-pooling which outputs $100 - kernel_width + 1$ values. Outputs from each max-pooling are concatenated and optionally dropout is applied. Follows two dense layers with ReLU nonlinearity which reduce the dimension to 300 and 100 respectively. Last output layer maps the input to our 26 classes one negative and 25 positives.	25
7.1 List of p2 instances on AWS from which we mostly used p2.8xlarge.	31
7.2 Diagram of infrastructure for training, testing and evaluation of models in AWS cloud.	34
8.1 Graphs from double descent experiments. We can not observe double descent in any graph. Experiment with L2 regularization and parameter $\lambda = 0.01$ resulted in accuracy and precision around 0.	39
8.2 Comparison of models with the raised weight of negative class. On the left is the model with negative class weight set to 100 on which we can not observe precision improvement. On the right, we see the model with negative class weighted by 120, which improves precision in later epochs.	40
8.3 Results of Model 3 with negative class weight set to 0.5 and positive classes weights set to 20 on testing dataset.	43

Tables

8.1 Datasets magnitudes.	36
8.2 Distribution of positive samples between classes in training and testing dataset.	37
8.3 In the table are results on our test dataset with different amounts of convolutional filters.	38
8.4 Results of experiment on data with excluded hostnames.	40
8.5 Results of base model with negative class weight set to 100. Left on the Figure 8.2	41
8.6 Results of base model with negative class weight set to 120. Right on the Figure 8.2	41
8.7 Results of semi-supervised training. Model 1 has negative and positive classes weights set to 1. In Model 2, the negative class weight is 1, and the positive classes have a weight set to 20. Model 3 has positive classes weights set to 20, and negative class weights are set to 0.5. Numbers in cells are amounts of weakly labelled samples that have been predicted as positive from total of 981 samples.	42



Chapter 1

Introduction

The cybernetic security field is growing due to the inevitable transfer of criminal activities from streets to the internet; the amount of attacks is growing so fast that it is impossible to keep up without automatization. Machine learning research in computer vision and natural language processing gives us a foothold for creating malware classifiers. Unfortunately, we can not adopt those algorithms fully while they are not counting with huge noise in real world data and imbalance between classes.

The most of malware is delivered to victims through malicious web sites; URLs leading to these sites are lurking in phishing emails, infected websites and more. We noticed that malicious campaigns often use similar URL patterns during the attack, which makes them a good match for convolutional neural network classifiers, that have proven ability to recognize patterns and generalize on them in computer vision classification tasks.

In this thesis, we aimed at three main goals:

- Implementation of pipeline that would handle the training, deploying and large scale inferencing of neural network models.
- Designing and implementing of the fully supervised model that would be usable in our production environment.
- Adding weak signal sources to our training procedure, so we are able to identify positive URLs in the unlabeled part of the dataset.

The next two Chapters 2 and 3 of this thesis are dedicated to explaining the classification with neural networks. We covered there the state of the art algorithms for creating neural network classification models as well as metrics to evaluate their performance.

In Chapter 4, we discuss the state of the art solutions for URL classification with a focus on those using neural networks.

Chapter 5 covers the data preprocessing and architecture of the fully supervised model.

On the beginning of Chapter 6, we discuss the state of the art of semi-supervised methods using weak labels during classification and methods for obtaining weak labels. After that, we present our model enriched by the weighting mechanism for weak labels support.

In Chapter 7, we briefly introduce modern frameworks for neural networks development and our solution for large scale model training and deploying. We also define our best practices for training the classifiers, which are dealing with imbalanced datasets.

We show the results of our experiments and detailed description of our imbalanced positive unlabeled dataset in Chapter 8 followed by the conclusion and ideas for future work.

Chapter 2

Supervised learning

Supervised learning is a subdiscipline in the general discipline of pattern recognition, which solves a problem of predicting *targets* from *input* data. Predicted targets can be of several kinds according to tasks we are solving:

- **Classification**, where we are predicting class from a given set of classes. More about classification in Section 2.1.
- **Regression**, answers questions *How many* and *How much*. The output is a scalar value. An example can be predicting of patients stay in hospital in days from given diagnosis.
- **Tagging** refers to a problem where inputs do not fit nicely to a single class. A common example is tagging objects in pictures.
- **Ranking** problem is closely related to searching engines where most relevant items should be listed first. Also, personalized commercials and other recommender systems belong to this group.

More groups of supervised learning can be found, but it is out of the scope of this thesis to dive deep into these topics.

Tasks where we are dealing with data without prior knowledge of what, we are supposed to predict, e.g. data clustering, are part of unsupervised learning. We will not discuss these type of problems in this thesis.

2.1 Classification

In classification, we are predicting category of a given input. Examples are diagnosis from patients symptoms, recognizing handwritten digits or spam and ham email separation. Categories, often called *labels* or *classes*, are usually denoted by y , we will use these terms interchangeably. The input data regularly requires feature extraction, which is the process of obtaining a scalar representation of the given sample. The example can be the number of light and dark pixels on a CT image or more advanced like the ratio of light pixels on each side of a brain, which refers to the domain knowledge. (Light tumour in one brain hemisphere can shift the ratio from the healthy

brain CT.) Each input is then represented by a vector composed from these numbers called *feature vector* or *tensor* in a neural network context. All the input tensors lies in *feature space* $\mathbf{X} \in \mathbb{R}^d$ and are denoted by x . Pairs (input, label) are called an *examples*, *samples* or an *instances*. We can also address as examples inputs where labels are not known. Usual notation for a *dataset* which consists of n samples is $\{x_i, y_i\}_{i=1}^n$. Our *classifier* or *model* f_θ is a function that maps any given input x_i to a prediction $f_\theta(x_i)$. Where θ stands for chosen *hyperparameters* of the classifier in the opposite of trained values that we address as *parameters*

Datasets are split in *training datasets* and *testing datasets*. In the cyber-security context, it is a good habit to split datasets according to time and make the testing dataset from examples following the training one. For the sake of good performance of a classifier, training dataset should be a good representative of a testing dataset and reality. We assume that the training and testing inputs are i.i.d. Further, we expect that the distribution is not changing in time when we use the classifier in production. Reality shows us that these assumptions do not hold completely. Since we are dealing with URL addresses which very likely do change their distribution in time and random sampling rule is certainly broke due to the time split of our dataset on training and testing. Thus we must compensate these violations in our classifier design.

Training is mediated by a *loss function* which penalise models according to errors made during the training phase. Classification problems commonly use a cross-entropy loss function of some kind. We use the softmax cross-entropy loss defined as:

$$p = \text{softmax}(f_\theta(x))$$

$$L = - \sum_i \log p_i, y_i$$

Since we are focusing on adding information from weak labels to the classification problem, we define the final loss as a product of softmax cross-entropy loss and weight w .

$$L = w_j * (- \sum_i \log p_i, y_i)$$

2.2 Positive unlabeled data

Supervised learning algorithms are most successful on large labelled datasets such as image databases. However, there are a lot of real world tasks that requires a vast amount of time and effort for obtaining labels. Good examples can be found in medical diagnosing where some tests can be costly or as in our case where labelling requires time from security analyst.

Our dataset is *positive unlabeled*, which means that our positive class is checked by an expert and therefore is confirmed to be positive. On the other hand, the negative class is mostly negative, but still contains positive samples that have been missed during data preprocessing. A lot of recent works

is dealing with this kind of datasets [12, 2] and are covered by the term *PU-learning* which originates in [3, 4].

One idea of PU-learning from above works is to sample 15% [3] of positive data to the unlabeled dataset, we call them *spies*. These spies than should act like positive samples in our unlabeled dataset. We can then filter out or relabel samples that behave similarly like our spies. What we are left with are *real positives*.

We deal with this task in Chapter 6 where we are proposing an experiment that shares the idea that the neighbourhood of samples can define their label; on the other hand, we do not inject spies to our negative class, but instead, we are decreasing the importance of the samples about which we think might be positive. Thus the samples around with higher importance can bring those uncertain with them to the correct class. We approach like this while it is more natural for neural network models.

We also wanted our training pipeline to be fully automated and as simple as possible. Therefore we do not want to filter out or relabel any data from the training dataset by hand or do manual checks on different stages of data preprocessing.

2.3 Overparametrized models

The main goal of every classifier is to perform well on new unseen data. Performance on new data is called *generalization*. Since our data are not randomly sampled from a dataset, but rather they are time-dependent as in reality, good generalization is one of the most important aspects of our model. The usual approach is to configure *function capacity* denoted by $||\mathcal{H}||$ to fit *bias-variance trade-off*. U-shape curve on the left in the Figure 2.1 visualizes correlation of error and $||\mathcal{H}||$. Models, with too little parameters, tends to be *under-fitted*. On the other hand, if we introduce too many parameters, we are running into the risk of over-fitting and poor generalization [7] [8]. It is a widely accepted idea that this sweet spot between under-fitting and over-fitting is an optimal solution. Yet, many modern applications are overparametrized and perform very well on testing data. Moreover, many of them fit near perfectly on training data [6], which would be considered as over-fitted according to [8]. Novel research on neural networks and decision trees proposes an alternative to classical U-shape curve in the form of *double descent* curve [5].

2.3.1 Double descent

Neural networks are prone to over-fitting thanks to usually huge $||\mathcal{H}||$. As an example, we refer to a paper where a capacity of neural network models is shown by perfect interpolation of randomly labelled data [21]. One way to prevent over-fitting is by choosing a simpler model architecture, which reduces $||\mathcal{H}||$. Another is *regularization* (see Section 3.4), which refer to techniques preventing over-fitting (e.g. early stopping of a training). Researchers in [5]

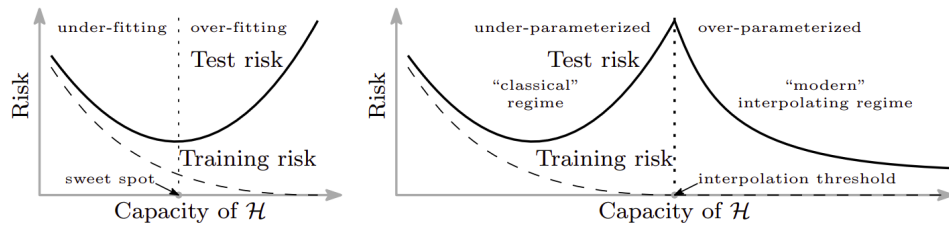


Figure 2.1: Double descent risk curve. Figure adopted from [5].

propose that with regularization techniques, bigger function capacity $||\mathcal{H}||$ leads to double descent curve, shown on right in the Figure 2.1. Functions with more parameters can interpolate data smoother. The smoother solutions are simpler than the rough ones. Thus by the Occam’s razor principle, they should be a better reflection of reality. We decided to test this hypothesis in our experiments.

2.4 Evaluation metrics

Earlier in this chapter, we determined our problem as a classification, described what datasets we are using and defined some essential properties we seek in classification models. But we did not specify any metrics that would objectively measure classifiers performance. Perhaps the most intuitive one is the *accuracy*, which is simply the number of correctly predicted samples divided by the amount of all samples. Since we are dealing with imbalanced datasets, accuracy can be misleading. Let us imagine the situation of a deadly disease like the plague (30% - 100% death ratio), which is fairly rare these days (A few thousands of cases per year). If it is diagnosed, the right treatment with the antibiotics usually saves a life. We could see something similar happening during the pandemic of COVID-19 in 2020 and the attempt of the Czech government to take a random sample from the population to estimate how many infected people there are in the country. The classifier which would predict every time that the patient is negative (healthy) would have very high accuracy. Still, somehow we feel that test like this is worthless despite its accuracy. Our situation in *network intrusion detection system* NIDS is identical; we have many negative samples and only a few positive.

On the way to improve evaluation, let us start with a definition of the confusion matrix on the binary problem with the positive and negative class. Our results in the confusion matrix can be of four types:

- **TP:** True positive. This is the number of correctly classified positive samples.
- **TN:** True negative. This is the number of correctly classified negative samples.
- **FN:** False negative: This is the number of incorrectly classified negative samples.

- **FP:** False positive: This is the number of incorrectly classified positive samples.

Now we can define four essential metrics that we will use to evaluate our models.

■ 2.4.1 Recall

Recall, *sensitivity* or *true positive rate* (TPR) gives us information about the ability of our classifier to predict positive class correctly. It is defined like this:

$$Recall = \frac{TP}{TP + FN}$$

Back in our example with plague and classifier that always predicts negative, the recall would be 0. Thus we get valuable information that the test is unable to detect positive cases. But if we flip the result of the test to predict always positive, we trivially achieved a 100% recall. Thus recall alone can also be misleading.

■ 2.4.2 Specificity

Specificity or *true negative rate* (TNR) gives us the same information about negative class as recall about positive class, which is how many percentages of negative samples are correctly predicted as negative. It is defined like this:

$$Specificity = \frac{TN}{TN + FP}$$

So the trivial plague test from our example has 100% specificity because all the negative patients are correctly predicted as negative.

■ 2.4.3 Accuracy

As we mentioned, earlier *accuracy* is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

It is also called classification error and gives us information about how much of the samples we misclassified. It works well on balanced datasets; for imbalanced problems, we can define balanced accuracy like this:

$$Balanced_Accuracy = \frac{TPR + TNR}{2}$$


In this thesis, we rather rely on recall and precision, which is our last metric.

■ 2.4.4 Precision

Precision or *positive predictive value* informs us about how many positive predictions are genuinely positive. Its definition is:

$$Precision = \frac{TP}{TP + FP}$$

It is the most important metric for us because it reflects the confidence of our classifier in predicting positive samples. Again we can illustrate this on our example. Precision is a percentage of treated patients that needs medication to survive. Hence antibiotics can be expensive; we want to treat only sick patients and not waste the resources on healthy ones. This corresponds with our use case, where system admins have only limited time. Thus we want to send them to fix only infected computers and not waste their time by reinstalling the healthy machines.



Chapter 3

Neural nets

Neural networks are a far broader topic than we can cover in this thesis. Further, in this chapter, we focus mainly on the classification problem and techniques that we used in our models.

As the name suggests, neural networks are inspired by real neurons. Artificial neuron mimics the real one like so:

- **Dendrites** which represent inputs
- **Nucleus** which simulates the computation unit
- **Axon** which is an output of nucleus computation (axon terminals which serve as a connection to other neurons)

Information x_i begins journey at dendrites; it can be received from another neuron or by an outer receptor, like hair cell (which is mechanical cell used to transmit sound in ears of all vertebrates). The signal is activated or inhibited in synapse $x_i w_i$, than the nucleus sums all the signals together $y = \sum x_i w_i + b$ and apply nonlinearity $\sigma(y)$. The final signal is sent to further processing by next neuron or serves its purpose in the final destination (e.g. neuromuscular junction).

This concept of learning and solving complex tasks by many "dummy" neurons stand on research in biology. Although nowadays, progress in neural networks is not much inspired by biology anymore but rather by mathematics.

3.1 Classification with neural networks

Classification in neural networks is the so-called soft classification. We are assigning a probability to each class instead of returning just predicted label. Let us demonstrate this on a classification problem of obtaining a diagnosis from CT image. For simplicity, we are considering only 2×2 CT image, and we want to predict if a patient is healthy, have tumour or infarction. Thus we have feature vector $x = (x_1, x_2, x_3, x_4)$ and one hot encoded classes $y = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ where:

- $(1, 0, 0) = \textit{healthy}$
- $(0, 1, 0) = \textit{tumor}$
- $(0, 0, 1) = \textit{infarction}$

Linear model classification needs 3 equations, one for every class. In this example, we need $3 * 4 = 12$ weights w and 3 biases b ; for each class we compute output like so:

$$o_1 = x_1w_{11} + x_2w_{12} + x_3w_{13} + x_4w_{14} + b_1$$

$$o_2 = x_1w_{21} + x_2w_{22} + x_3w_{23} + x_4w_{24} + b_2$$

$$o_3 = x_1w_{31} + x_2w_{32} + x_3w_{33} + x_4w_{34} + b_3$$

Above equations give us a neural net model (Figure 3.1) with one fully connected layer, also called a dense layer. We can express this model in a more compact way using linear algebra: $o = Wx + b$.

3.1.1 Softmax

We want our outputs to interpret probabilities of each class. We can not use outputs o directly because nothing is restricting those values to be non-negative or force them to sum up to 1, which violates fundamental rules of a probability distribution. Hence we use *softmax function*:

$$\hat{y} = \textit{softmax}(o) \text{ and } \hat{y}_i = \frac{\exp(o_i)}{\sum_j^n \exp(o_j)}$$

Values in \hat{y} are corresponding to the probability distribution while:

$$y_1 + y_2 + y_3 = 1 \text{ and } 0 \leq y_i \leq 1, \forall i$$

Predicted label is usually chosen as $\textit{argmax}(\hat{y})$, which is possible because softmax is preserving the ordering of values in \hat{y} .

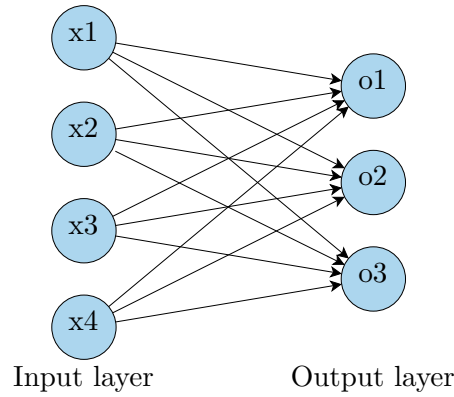


Figure 3.1: Linear neural network model with 4 inputs and 3 outputs.

3.1.2 Cross-Entropy loss

Vector \hat{y} gives us a conditional probability estimate of each class from input x , thus $\hat{y}_1 = \hat{P}(y = healthy|x)$. We can check our prediction with reality using log-likelihood:

$$P(Y|X) = \prod_{i=1}^n P(y^{(i)}|x^{(i)}) \rightarrow -\log P(Y|X) = \sum_{i=1}^n -\log P(y^{(i)}|x^{(i)})$$

Thus minimizing $-\log P(Y|X)$ coincide with predicting the right label. We can also derive loss function, which is called the cross-entropy loss from this relationship.

$$loss(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$$

At last we can chain softmax function with cross-entropy loss to obtain *softmax cross-entropy loss*.

$$loss(y, o) = -\sum_i y_i \log \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

3.1.3 Adam

Since we usually can not solve our high dimensional models analytically, we need to use some numeric method for solving them. Almost all optimizing techniques used in deep learning are some form of *gradient descent*. If our loss function surface is convex, it will eventually converge to a global minimum. On nonconvex surfaces, we hope it converges to a local minimum that will be good enough. In this thesis, we will cover only one advanced derivative of gradient descent, which is Adam optimizer, first described in [22].

Adam combines several optimization techniques; nevertheless, it is still reasonably robust. Thanks to its fast convergence, Adam becomes the best practise optimizer for deep neural networks. Thus we are using it for optimizing our models. Although it has been shown that Adam can diverge in some cases due to issues with variance control [23].

Adam algorithm

The goal of our algorithm is to update the parameter vector w in the direction of a local minimum. For understanding how Adam updates its values, we briefly recap *minibatch gradient descent* and *Leaky averages*. Minibatch gradient descent computes gradient for each sample from a small batch and averages them.

$$g_t = \partial_w \frac{1}{|B_t|} \sum_{i \in B_t} \text{loss}(x_i, w_t)$$

That has a positive side effect of decreasing variance, namely by a factor $|B_t|^{-\frac{1}{2}}$. Thus naively we should use as big batches as is the memory of the device we compute on, more on this in Chapter 7.

Leaky averages take advantage of this variance reduction one step further by introducing momentum v .

$$v_t = \beta v_{t-1} + g_{t,t-1}$$

Momentum takes account of past gradients and moves forward with respect to them.

Adam uses, in addition to momentum, the second moment both in exponential weighted form.

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$

β_1 and β_2 are positive parameters that are usually set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Corresponding normalized variables are defined like so:

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t}$$

$$\hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

Now with all prerequisites set we can introduce update equation

$$g'_t = \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

where η is learning rate and ϵ is constant usually $\epsilon = 10^{-6}$, which prevents us from dividing by zero. Parameters updates are then computed like this:

$$w_t = w_{t-1} - g'_t$$

3.2 Deep neural networks

On the beginning of this chapter, we defined a simple linear neural network model Figure 3.1. Furthermore, we discussed how to convert outputs to a probability distribution. We can also optimize models weights according to

the loss function. However, we are still able to solve only linear problems. Let us recall the formula for single-layer linear model:

$$o = Wx + b$$

We can now think of tasks like predicting if a patient will die based on body temperature. Patients with a body temperature above 36.6°C are running into higher risk with further temperature growth. On the other hand patients with body temperature under 36.6°C are getting better with raising temperature. Since linearity implies monotonicity and thus increase in the input must always increase or always decrease output value, a linear function can not fit those data well. However change in data representation can help us; for example, we can measure the distance from optimal temperature. In this easy example we can find out the correct data transformation. In the more complicated ones, we use hidden layers to learn the right representation in the training process.

The easiest way how to introduce hidden layers into a model is to stack several linear layers on top of each other. We can see the neural net model with the hidden layer in Figure 3.2. The output of this model is given as:

$$h = W_1x + b_1$$

$$o = W_2h + b_2$$

$$\hat{y} = \text{softmax}(o)$$

Unfortunately, (before we apply softmax) this is a linear function of linear functions, which is in the end linear function. To break the chains of linearity, we have to add nonlinear *activation function* σ . In our models, we use the rectified linear unit (ReLU) activation, but other functions exist such as sigmoid, tanh, etc.

$$\text{ReLU}(x) = \max(x, 0)$$

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

$$\text{tanh}(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

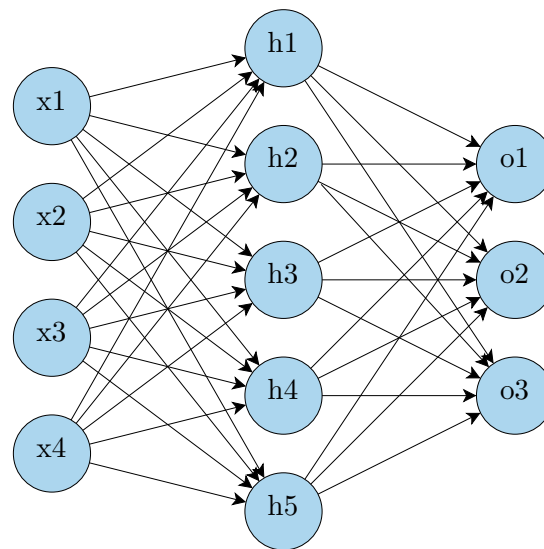
The modified equations for the model with the hidden layer and nonlinear activation function are:

$$h = \sigma(W_1x + b_1)$$

$$o = W_2h + b_2$$

$$\hat{y} = \text{softmax}(o)$$

Now we have everything to create deep models with multiple hidden layers that can learn complex interactions between inputs. It is widely known that even model with single hidden layer works as a universal approximator with certain choices of the activation function. Although it is not wise to use such architecture, because it is relatively hard to train it.



Input layer Hidden layer Output layer

Figure 3.2: Neural network model with one hidden layer.

3.3 Convolution neural nets (CNN)

Neural network models, composed of dense layers, are relevant for inputs that can be characterized as vectors of features, where we do not assume any structure or local interactions. Patients measurements like temperature and blood pressure that can be given in random order are an example of data that work well with models constructed from dense layers.

On the other hand, CT images where the position of each pixel matters are not suitable for this architectures. For example, if we would like to make a model with a hidden dense layer for a one-megapixel CT image that would reduce it to 1000 dimensions, this dense layer would have 10^9 weights to train. This is too much even for powerful GPU machines. For classification on CT images, we would usually use *convolution*. More specifically, convolution is relevant wherever:

- Respond to a pattern should be same without concern of position in an input (tumour can be anywhere on image).
- First layers of the neural network should analyze local regions without the influence of distant ones (detection of tumour in the top left corner of an image should not be subject to an infarction in the bottom right).

URL addresses and text, in general, are subject to the above criteria. Hence we decided to use CNN architecture for our classifier.

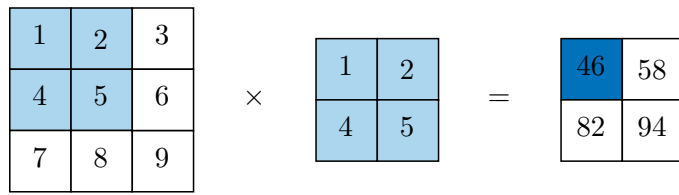


Figure 3.3: Cross-correlation operation with input on the left, kernel in the middle and output on the right.

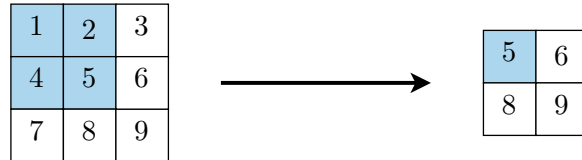


Figure 3.4: Max-pooling operation example.

3.3.1 Convolution layer

Convolution layers are more accurately cross-correlation layers, where we take input tensor and correlation *kernel* tensor then we apply sliding dot product operation to obtain cross-correlation. Let us illustrate it on an example with 3×3 input tensor and 2×2 kernel also called a *filter*.

The blue window on the input tensor in Figure 3.3 slides from left to right and top to bottom in each step dot product with kernel tensor is placed to output. After this process is done, we add bias.

To be complete, we have to mention hyperparameters that can change convolution behaviour.

- Padding is a technique where we pad input tensor, usually with zeros, around edges. Thus we do not shrink our output tensor in comparison to input.
- Stride is a parameter that defines the length of the slide. If we want to downsample our data, we can use higher strides.

3.3.2 Pooling

With convolution is very often introduced pooling, which reduces hidden dimension. Thus latter layers are sensitive to input as a whole. Another purpose of pooling is to reduce the importance of patterns position in the input.

Like in the convolution, pooling uses a sliding window of fixed size and traverses across input in the same manner. But unlike the convolution pooling is not learned operation, it usually extracts maximum or makes average from values in the window. Other operations are also possible. Thus the main difference lays in the lack of filters to be trained. Figure 3.4 demonstrate max-pooling, which we use in our architecture.

3.4 Regularization

In Chapter 2, we touched the topic of regularization. Let us quickly remind that we stated that neural networks are prone to overfitting and have huge variance. We also stated that our models should generalize well to obtain good results in a production environment; we also suggest that using simple models can improve generalization. Now we introduce it in the context of neural networks, and we define several regularization techniques that we have tested.

First, we have to mention that the size of dataset matters. Probably the best way of generalization is to collect enough data to train a complex model that will not overfit to them, the downside of this is the cost of obtaining data and the cost of training. Another straightforward technique is to stop training when we reach the sweet spot in bias-variance trade-off. Further, we will assume that we did our best in data collecting and that we will pick our best model from training epochs.

3.4.1 L2 regularization

L2 regularization or *weight decay* is motivated by the assumption that the function $f = 0$ is the simplest one. Thus models that are closer to f are better. How to measure this proximity between some function h and f is an open question. One of the possible answers can be some norm, which we then use as a penalization during minimization of h . Most common realization is adding $\frac{\lambda}{2} \|w\|^2$ to the *loss* function. There λ is a hyperparameter determining penalty strength, and w is a vector of weights of a model. Thus we are adding λw to the computed gradient g . As a result, we are not taking a step in direction $-\eta g_t$ but rather $-\eta(g_t + \lambda w)$ this effectively decreases w by $\eta \lambda w$ at each backpropagation run of the learning process. Recall that η denotes learning rate scalar. In other words, we are pushing the classifier to use more and smaller weights, instead of depending on a few superior ones. From there comes the name weight decay.

This is not the usual technique of regularization of complex neural net models, but since it is proposed in [5] we did tests with it. Results of our experiments are described in chapter 8.

3.4.2 Dropout

In the case of *dropout*, we examine smoothness as a measure of functions simplicity. We can also interpret this as robustness against small changes in the input (e.g. noise in the image). Christopher Bishop proved that training with noise is equivalent to Tikhonov regularization [20], which is designed to improve efficiency in parameter estimation in exchange for bias in problems without unique solutions (ill-posed problems). Dropout, as stated in [18, 17], is a way how to inject noise to the hidden layers in neural networks. We borrow biological motivation for dropout, as presented in [18], which comes

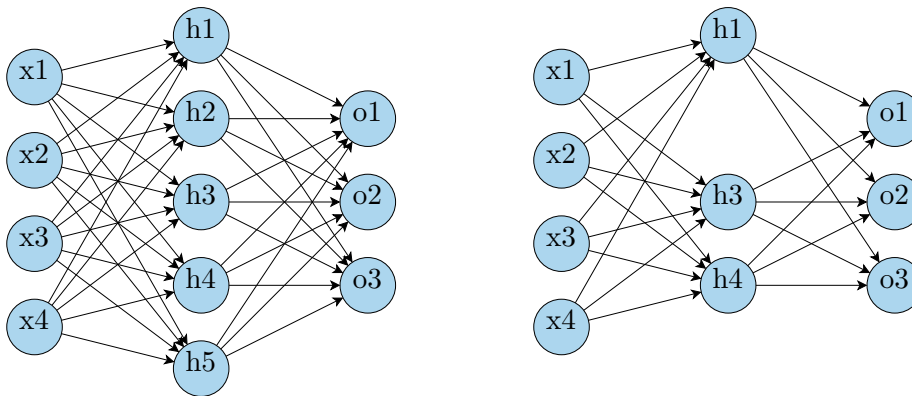


Figure 3.5: Neural network before dropout is on the left and neural network after applying dropout is on the right.

from the evolutionary role of sex, described in [19].

For sexual reproduction, we usually take the first half of genes from one parent and the second from another one, final offspring is a combination of both of them with some minor mutations. Asexual reproduction skips the combining part and produces offspring as a copy of a parent with minor mutations. At first glance, it might seem that asexual strategy is better for individuals fitness while optimized genes already can work together in the form of co-adaption. Sexual reproduction would destroy all these fine-tuned co-adaptions that evolved in the past. Nevertheless, sexual reproduction is the one we see in the most advanced organisms. Proposed explanation of this phenomena is that the ability of genes to work with other not co-adapted ones is maybe more important. This enables the spreading of useful genes across the population; it is also easier to pop up when they are not blocked by a chance of breaking some sharply bounded co-adapted gene complex. Thus, in the end, it is easier to improve individuals fitness, also when the environment changes, organisms can adapt without breaking those co-adaptions. Hidden layers in neural networks should follow the same principle and do not overfit to exact patterns in the previous layer. Dropout prevents this by randomly setting a value of a hidden node to 0 with a given probability (usually 0.5). We predestine that dropout regularization works better for our setup; the results of our experiments are in Chapter 8.

■ 3.4.3 Implicit regularization of gradient descent

Recent papers [24, 25] are also talking about implicit regularization of gradient descent algorithm. We are not going into depth in this topic, but it is good to keep in mind that the optimization algorithm by itself can favour simple solutions above others. Both papers are dealing with matrix completion problem, where we have been given some entries $X_{i,j} : (i, j) \in \Omega$ from the matrix X . Our task is to recover missing values from the given entries. This can be viewed as a regression problem where training points are given values

from X , and the model is matrix W . The optimization of W can be done by:

$$loss = \sum_{(i,j) \in \Omega} (W_{i,j} - X_{i,j})^2$$

We can say that our model generalizes well when X is similar to M in unobserved regions. Since loss function has multiple optimal solutions that we can not compare, we have to add an assumption that the matrices with lower ranks are preferable. Gunasekar et al. [24] stated that with low learning rate and near-zero initialization linear neural networks of depth 2 find the solution with the minimal nuclear norm. In [25] is proposed that deeper linear neural networks have even better solutions. It is thanks to the gradient descent tendency to improve singular values by little each step until a certain threshold is reached after that singular values rise rapidly. Furthermore, the rise of singular values is getting steeper with the growing depth of the linear neural network. Thus gradient descent prefers solutions with lower ranks and implicitly regularize the model.

■ 3.4.4 Batch normalization

It is known that the data preprocessing can impact models performance profoundly. Often we have features of different scales. For example, one feature can be expressed in percentages and take values from 0 to 1 another one can be real value ranges from 0 to 1000. It is a good idea to standardize the inputs, so they have the same mean and variance. This helps the optimizer to converge in the right direction based equally on all features in the input since the small difference in the magnitude of values does not make the gradient to act hectically. Thanks to smaller but more accurate gradients, we can use higher learning rates and converge faster.

The motivation for batch normalization comes from the fact that nothing restricts hidden layers from taking on values of varying magnitudes. Thus it makes sense to standardize them as well as inputs. When we apply batch normalization on a layer we first compute activations as usual, then we normalize them on each node. By normalizing, we mean subtracting its mean and dividing it by standard deviation, which we both obtain from the current minibatch.

$$\hat{\mu}_B = \frac{1}{|B|} \sum_{x \in B} x \text{ and } \hat{\sigma}_B^2 = \frac{1}{|B|} \sum_{x \in B} (x - \mu_B)^2 + \epsilon$$

The ϵ constant is added for ensuring that we never divide by zero. Now we can formally define batch normalization like so:

$$Batch_normalization = \frac{1}{|B|} \sum_{x \in B} \gamma \odot \frac{x - \hat{\mu}}{\hat{\sigma}} + \beta$$

Here γ is scaling coefficient and β is offset, together they ensure that layer will not diverge, because we are actively centring and rescaling μ with σ to

given values. By using an estimated $\hat{\mu}$ and $\hat{\sigma}$ we bring in the noise, which can be beneficial for robustness as we described in the dropout section.

Once training is complete, we compute mean and variance from the whole dataset and use them for the inference. Like this, we will have consistent predictions during inference, not depending on batch.

3.5 Sequence models

Neural networks can have many more specialized layers and optimizations than we described in this chapter. Sequence models are one of those that are used in URL classification. In this section, we will briefly introduce popular architectures for handling sequential data. We will not go deep into implementations and math since it is out of the scope of this thesis.

Previous models expected independent data from the same distribution. This assumption is violated in many real-world tasks. Namely, natural language processing (NLP) is one of those tasks. State of the art in NLP is driven by sequential models that deal with dependencies between inputs. URL addresses are indeed texts of varying length and can be processed by NLP classifiers.

3.5.1 Recurrent neural networks

Recurrent neural networks (RNN) introduce hidden-state h , which acts as a memory of previous data. We can say that h stores sequence information. To obtain h_t we use information from current input x_t and previous hidden state h_{t-1} in our activation function f .

$$h_t = f(x_t, h_{t-1})$$

For the better understanding let us assume the input sequence $X_t \in \mathbb{R}^{n \times d}$ where $t \in T$ denotes position of input in the sequence. First, we need to compute $H_t \in \mathbb{R}^{n \times h}$, which stands for the hidden state for input t from the sequence. For that, we also need state $H_{t-1} \in \mathbb{R}^{n \times h}$ from the previous timestep. Unlike from the dense layer, we use two parameter matrices $W_{xh} \in \mathbb{R}^{d \times h}$, which serves the same purpose as in the dense layer, and $W_{hh} \in \mathbb{R}^{h \times h}$ which is used to determine how to handle the previous hidden state.

$$H_t = \sigma(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

After obtaining H_t the output is given by:

$$O_t = H_t W_{hq} + b_q$$

Here, $W_{hq} \in \mathbb{R}^{h \times q}$ contains the weights of the output layer and b_q with b_h are corresponding biases. RNN uses the same parameters for all the timesteps in T ; hence the number of parameters stays the same for the sequence of arbitrary length.

■ 3.5.2 Long Short Term Memory

Long Short Term Memory (LSTM) architectures belong to recurrent models family. These type of neural networks are designed to preserve long-term information and skip short-term input. One of the first publication on this topic is [27]. The motivation for those models can be found in logic gates. Namely, we need a representation of the output gate, the input gate and forgot gate. Gates together create a memory cell. This mechanism serves the purpose of deciding when to ignore the input and when to remember it. We skip the realization of the memory cell and gates while it is out of the scope of this thesis.

■ 3.5.3 Transformer

In 2017 Google researchers introduced the new architecture for processing the sequential data and called it *Transformer* [26]. It improves the state of the art models of encoder-decoder that used two RNNs connected through hidden dense layers. Transformer proposes a multi-head attention mechanism that uses the input sequence and the so far obtained output sequence together for predicting the next output in the sequence; for capturing the position in the sequence, positional embedding is used. After each multi-head attention layer is placed normalization, which helps with the training process. At the end of the Transformer lays a linear layer which outputs the same number of outputs as is possible outcomes after that softmax is applied, and the maximal argument is selected as a result. The attention formula from [26] is:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The Q stands for the query, the K stands for keys, and the V contains values. Multi-head attention layer, in which most of the mapping from one sequence to another happens, takes the K and V from the input sequence and Q from the so far build output sequence, d_k is a dimension of keys and queries from the input. Thus we use keys and values from our input to obtain the output which we query by so far obtained tokens in sequence.

The whole process can be parallelized in several places (e.g. multi-head attention can happen parallelly); thus, the whole training process is faster than the RNN solution. According to [26] Transformers are becoming the current state of the art solutions for language translation.

Chapter 4

Classification of URLs

In recent years deep learning has experienced a boom. That can be beside other credited to a number of public datasets [41] that allowed researchers to compete, benchmark and innovate under the same conditions. Unlike in computer vision or natural language processing, cybersecurity is lagging behind in this regard. Public datasets are of poor quality, mostly due to the preservation of privacy; moreover, we did not find anyone who would deal with the multiclass classification of URLs. Therefore it is tough to compare the state of the art solutions.

4.1 Neural network models

There are two general approaches to handle URLs with neural network classifiers. The first one is to use a sequential model such as a recurrent neural network or long short term memory model both described in Chapter 3. In [31] are researchers testing several sequential models on binary URL classification problem with promising results of accuracy precision and recall above 95%. Unfortunately, their datasets are rather small (tens of thousands) and balanced. Vinayakumar et al. [30] made a similar comparison on an even smaller dataset. To obtain metrics for the distribution in our data, we would have to rescale them according to our imbalance ratio for example by methods described in [39], which would lead to a drop in precision.

The second approach is to use the CNN model. Data entering the convolution layer must be of uniform dimension, which unfortunately rises a requirement to threshold on maximum URL length. Luckily the majority of URLs have no more than tens maximally hundreds of characters. Thus we are able to fit URLs to tensors that can be convoluted in a reasonable time.

Compared to sequential models CNN are much faster and less prone to vanishing gradient. CNN also showed that they are capable of handling sequential data [32]. Joshua Saxe and Konstantin Berlin [29] proposed a method for classification file path, registry keys and URLs based on CNN over matrix with embedded characters. According to [31] their solution performed better than sequential models. Their architecture was followed by URLNet project to which we dedicate the next section.

4.1.1 URLnet

Perhaps closest to our task are researchers from Singapore Management University with their URLNet [28]. They solve the binary classification problem on URLs with the usage of convolution layers. Unlike us, they are embedding characters and words into matrices. The unique word dictionary is made from whole training dataset before training the model. Because the dictionary can grow with each new URL in the dataset paper also proposes character-level word embedding, which saves memory at the expense of computational complexity. Convolution is than made over those matrices. Overall, URLNet is more complicated, and thus we expect it to be slower than our model.

The most significant difference lies in the dataset on which URLNet is trained. Malicious samples were obtained from VirusTotal. In [28] is written: *"Given an input URL, VirusTotal scans through 64 different blacklists (e.g. CyberCrime, FraudSense, BitDefender, Google Safebrowsing, etc.), and reports how many of these blacklists contain the input URL."* URL that appeared in more than 4 blacklists was declared malicious. Benign URLs were those that appeared in none of the blacklists, rest of the URLs were discarded. This is a significant difference from our positive unlabeled dataset. Also, the ratio of the positive to negative samples is different URLNet has roughly 15 times more negative samples, while we have 1500 times more unlabeled samples than positive.

4.2 Other classification approaches

We discussed neural net classifiers while they are related to our work, but there are many more options on how to approach the URL classification task. Namely, we can make future extraction and use some standard algorithm e.g. SVM. Or we could use other machine learning approaches like random forests in [33]. Further research of those alternatives is out of the scope of this thesis.

Chapter 5

Fully supervised model

Malicious campaigns often use patterns in URLs that distinguish them from legitimate traffic. We designed our fully supervised model to find these patterns and generalize on them. We use one negative label which covers unlabeled part of the dataset and 25 positive labels for malware classes. More about labels and our dataset can be found in Chapter 8.

5.1 Data preprocessing

Before we push our data into the first layer, we limit each URL to a maximum length of 100 characters; then we apply one-hot encoding. Individual characters are encoded by number 1..94 which covers all allowed symbols in URL; 0 is used as a padding for URLs shorter than 100 characters. Thus we have 95×100 tensor representation of URL that enters the first layer (Figure 5.1).

In some experiments, we removed the hostnames from URLs to prevent over-fitting to them. The side effect of cutting out hostnames is that we can end with the same examples in the positive and the negative class. While precision is critical for us, we decided to place those samples only in the negative class.

5.2 Architecture

Our pattern recognition mechanism is built on two convolutional layers with different kernel sizes followed by 1D max-pooling. They can be seen as a form of n-gram pattern-finding layers. Outputs from pooling are concatenated into a single tensor, and optionally dropout is applied here. At last, two dense layers, one with 300 hidden neurons and second with 100 hidden neurons followed by output layer are connected to the model. Both dense layers use ReLU nonlinearity. The whole architecture is shown in Figure 5.2. We did not use batch normalization because our convergence during epochs is fast enough, and we were adding new epochs mostly due to the introduction of new negative samples. But we plan to experiment with batch normalization in the future because it seems like it can only improve the model's performance.

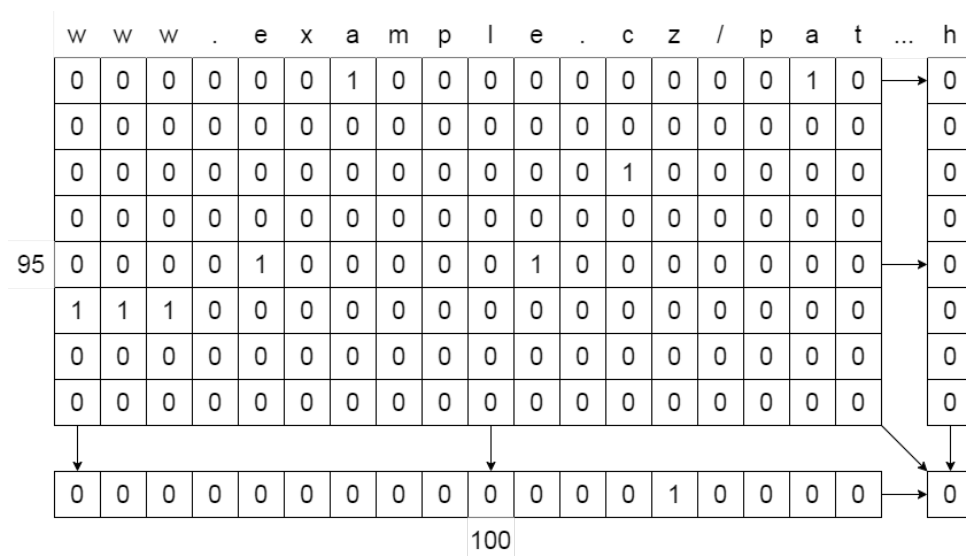


Figure 5.1: Encoding of the URL into 95×100 one-hot representation.

5.3 Hyperparameters

In our experiments, we did not modify the parameters of the Adam optimizer as the best practice is to use predefined ones in [22]. We also did not evaluate deeply batch sizes and stick to best practise from [40], although theory says it is optimal to have batches of the same size as the memory on the computing device (see Section 3.1.3 to find out how bigger batches increase variance).

We did investigate sizes of convolutional kernels as they are a crucial component of our pattern recognizing mechanism, and find out that small numbers around 5 work best for us. We attribute the lack of difference in behaviour between similarly sized kernels to the fact that small or zero values in convolution filters act in the same manner like a choice of a narrower filter; also dependency between two filters can result in recognition of n-gram wider than the filter width. We also tuned the number of kernels in the convolution. We aimed for the lowest amount that does not hurt the performance of the model because convolution is the most time complex part of the model. We found out that higher tens of kernels are the sweet spot where the performance of the model does not improve with further kernel increase.

We were also investigating different architecture setups from the optimal number of dense layers with the number of hidden neurons in them to convolution layers stacking setup. In this thesis, we present the final most successful architecture that we come up with.

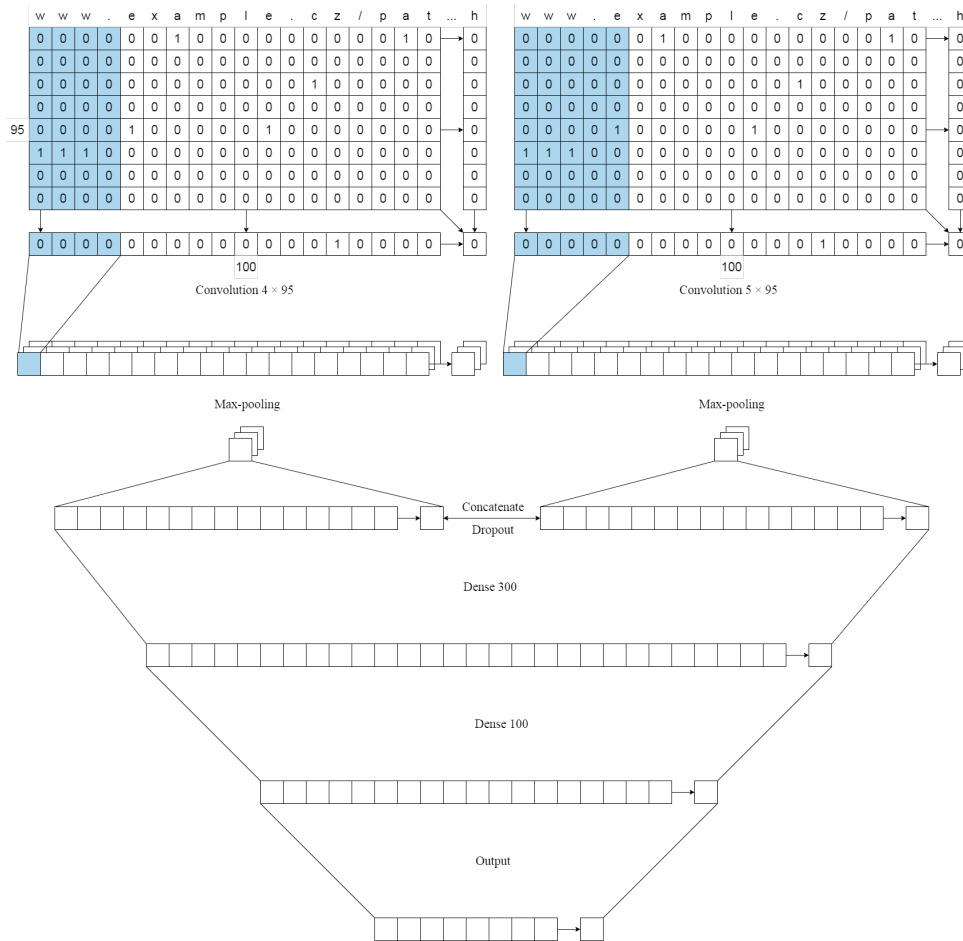


Figure 5.2: The architecture of the neural network. With two convolutional layers one with the kernel width 4 and the second with kernel width 5. The number of kernels is discussed in Chapter 8. Convolution is followed by max-pooling which outputs $100 - kernel_width + 1$ values. Outputs from each max-pooling are concatenated and optionally dropout is applied. Follows two dense layers with ReLU nonlinearity which reduce the dimension to 300 and 100 respectively. Last output layer maps the input to our 26 classes one negative and 25 positives.

Chapter 6

Weakly labeled model

Modern supervised learning methods depend on large labelled training datasets. Labelling the data is often a key bottleneck of the machine learning pipeline. In cybersecurity is this task especially expensive due to the need of domain experts for correct labelling; that does not scale with continually evolving malware. With just a restricted amount of labelled data, weak signals can significantly increase information gained from the training dataset. Hence semi-supervised models are arising.

The basic semi-supervised approach is to pre-train model on weakly labelled data and then make final tuning on the data with true labels [16]. According to [10] this is not an optimal approach due to lack of control how much information from each source we gain since some sources can be more reliable than others. One possible answer to this problem can be found in [36, 37] where noisy label sources are combined together. Significant progress in this field has been made by researchers from Stanford University who designed a framework that can combine multiple weak labelled sources with contradicting overlaps and assign the correct label to each sample based on weak label source reliability [34, 35].

In the URL classification task, we can obtain weak labels as outputs of different classifiers or from publicly available blacklists, which from practice rarely overlap. At the time of writing this thesis, popular rule based network intrusion detection system Snort [15] has on its official sites listed nine sources from Cisco and fifteen third party sources. There is also a semi-supervised classifier on proxy logs from V. Franc et al. [13], that uses domain blacklists from [14] and multiple instance learning.

Both the above works [13, 10] show the usefulness of using weak signals in the training of classifiers on noisy datasets; also thanks to blacklists and already working classifiers it is far cheaper to obtain weak labels than true labels. Thus our neural net classifier uses weak signals during training, which is recommended in [10], as weights for the loss function defined in Section 2.1 like this:

$$p = \text{softmax}(f_{\theta}(x))$$
$$L = w_j * \left(- \sum_i \log p_i, y_i\right)$$

We describe our semi-supervised experiment with weak labels in Chapter 8,

where is also the description of our dataset.

■ 6.1 Model with weak labels

Our model with weak labels is identical to the fully supervised one (Figure 5.2). Simple multiplication of loss function by weight scalar is sufficient for introducing weak labels. This favours neural nets models, for example before forest models, where weighting can be trickier.

We use the weights to decrease training loss of the samples from the unlabeled part of the dataset that might be malicious, but we do not have that information from a reliable source. Hence we effectively say to the neural network to take the sample less seriously, and thus the sample can be classified according to samples that contain similar patterns, and we are confident with their labels.

Chapter 7

Infrastructure

One of the biggest challenges of the whole project was building an infrastructure that would fit all our needs. Key aspects are speed, comparability with our current models and scaling in our production environment, which is running under Java. This brings up several challenges from which the most influencing is choosing the right framework. Almost all state of the art libraries uses Python as the language for training models, and only a few have support for Java. Thus transferring the trained model into the production environment can cause us a problem. In our first attempt, we implemented the whole inference part in Java by our selves while using a framework for training in Python. This solution was fast enough for production while it was optimized for the current architecture, but model retraining was hard to automatize. Every modification in the training part must have been replicated in the inference part, which is prone to mistakes and took twice as much time. So our next steps aimed for better infrastructure with easier maintenance.

7.1 Frameworks

It is just now when we start seeing mature frameworks for deep learning, that are meant to work on industrial level. They also take into account training on multiple GPUs, which speeds up the training process significantly. Further, we will focus on three main frameworks; each represented by one company giant. The competition between Facebook and Google brought us a lot of improvement and easy baselining of deep learning frameworks; also, excellent documentation helps the community to proliferate. Recently Amazon joined two giants, took the good things from both Google and Facebook and added multiple platform solution.

7.1.1 PyTorch

PyTorch is a pythonic deep learning framework mostly developed by Facebook. API is flexible and straightforward, which is good for fast learning with no restrictions on how complex models can be. Even with a high level of abstraction, one can build any neural net model from scratch and add unique features. Another defining aspect of PyTorch is a dynamic computational

graph representation of models. The computational graph is a representation of the model by a directed graph where edges show data flow. PyTorch allows modifications of this graph during runtime. The latest version also supports TorchScript, which is meant to be a production environment.

■ 7.1.2 TensorFlow

TensorFlow from Google provides multiple levels of abstraction as PyTorch, although the learning curve of PyTorch seems steeper to us, at least on the beginning. The user must understand more of how the TensorFlow works (sessions, placeholders, etc.) to start using it. Also, models are represented by static computational graphs; hence sequential data are harder to process since we must know their size in advance. TensorFlow is more production-ready than PyTorch; in the latest version, we can find unstable java API, which is a big step forward for plugging neural networks into production. It is also more prepared for distributed training [38]. Last but not least, TensorBoard is a brilliant tool that visualizes models in the browser, which we miss in other frameworks.

■ 7.1.3 MXNet

Finally, we introduce our choice MXNet from Amazon. MXNet is a framework build on Gluon API, which is a collection of machine learning algorithms written in C++. We prioritized MXNet before others mostly because of the strong support of languages like Scala and Java, which we use in our production environment, while we can still comfortably train our models in Python. Another advantage is native support of Amazon Web Services (AWS), which we use for training and where was recently transferred our production environment. MXNet is also capable of mixing symbolic and imperative programming, which allows easier debugging without sacrificing performance. We also can not omit excellent documentation and a book [9] with examples. All mentioned above makes MXNet most suitable choice for our projects. Further, we will look at two parts of our infrastructure developed with the use of MXNet. First one is a training part in Python followed by inference and evaluation written in Java/Scala.

■ 7.2 Python part

Python part can be divided into several sections.

- Data loading
- Data providing
- Training loop

In our training codebase, we use Python 3.6 with MXNet 1.5.1, which is not the latest one. On February 21, 2020 was released version 1.6.0 with focus

Name	GPUs	vCPUs	RAM (GiB)	Network Bandwidth	Price/Hour*	RI Price / Hour**
p2.xlarge	1	4	61	High	\$0.900	\$0.425
p2.8xlarge	8	32	488	10 Gbps	\$7.200	\$3.400
p2.16xlarge	16	64	732	20 Gbps	\$14.400	\$6.800

Figure 7.1: List of p2 instances on AWS from which we mostly used p2.8xlarge.

on NumPy compatibility (version 1.5.1 used NumPy 1.16.2 which was not the major version at the time of release) and also resolved some issues that we had to hotfix in our code. Whole training loop is running under *AWS Batch*, which is capable of starting and allocating resources for docker images with our experiments. We store our images in *Amazon Elastic Container Service* (ECS) and data on *Amazon simple storage service* (S3). Code is then physically running on one of the *p2 Amazon Elastic Compute Cloud* (EC2) instances (Figure 7.1). The whole pipeline of retraining model in production can be done by one job under AWS Batch, which was our desired idea when we were switching from our first implementation; trained in PyTorch and than manually transferred to hard-coded Java environment.

7.2.1 Data loading

Our data are stored on AWS S3 in Optimized Row Columnar (ORC) files format, which enables to work with the desired column without need of reading the whole row. We used *PyArrow* library to read ORC files, which sadly support ORC parsing futures only in the older release that ultimately depends on the older version of NumPy than we needed for MXNet. When we stitched all libraries together, something went wrong, and we were downloading whole rows instead of the desired column with URLs. Thus we made a Spark job to extract only labels with URLs in the form of tab-separated values (tsv) file, which as the name indicates is a simple text file where on each line is label and URL separated by the tabulator. During training, we are directly downloading these tsv files into memory. In future, we want to remove the extra step of converting ORC to tsv and optimize this process to run on CPUs while training will be in progress on GPUs.

7.2.2 Data providing

On the beginning of this chapter, we mentioned that we had to hotfix some imperfections of older MXNet release. Data loader was one of those. Since we have enough negative data samples, we decide to load a large amount of them and then randomly pick from these for a mini-batch instead of reusing them each epoch. This raised problem because C implementation had allocated 32-bit address space for items in one data loader. When we were optimizing the amount of data with the size of RAM on AWS instances, we overflow this

restriction. As a solution, we created class superior to the data loader, which stores an array of data loaders and iterates through them.

■ Epochs dataset composition

Our dataset is highly imbalanced in one month of traffic we observe tens of millions of negative samples. Yet, many positive classes do not exceed ten occurrences (the most "talkative" ones have tens of thousands of samples). The general approach that we know from many image recognition datasets, where we go once through each sample during one epoch is predestined to end up with model that always predicts negative and have almost 100% accuracy with zero recall. Thus before each epoch, we compose our data into equal bags for each class we have. This profoundly impacts data distribution and can hurt precision at the expense of recall. We use the number of epochs alongside with loss weighting to find the optimal model and thus compensate the distribution shift. As mentioned before we have much more negative samples than we can process, so we pick randomly corresponding amount for each epoch.

To recap; first, we make bags of equal size for each class. The bag has a size based on the biggest positive class (like that we use all our positive samples). Smaller positive classes have duplicates in them. After that, we add a bag of the desired size from the negative class, and we declare these bags to be dataset for the current epoch.

■ 7.2.3 Training loop

When we built epoch's dataset, our training loop is pretty standard. We create data loader from it and add weights in this part of the algorithm like this we can change weights between epochs. We also declare our mini-batches to be n times larger than we want them to be in the end, because when data loader provide us with mini-batch, we immediately split it by the number of GPUs that we have on the EC2 instance. Thus n denotes the number of available GPUs. We also followed the recommended size for mini-batches in [40] and set it to 32, although we can probably go higher, this is something to be tested in the future.

For optimization, we used the cross-entropy loss as described in Chapter 3 and Adam optimizer with recommended hyperparameters from [22], also described in Chapter 3.

■ 7.3 Java part

Our production environment is running under Java and Scala. Thus models must adapt to this environment. MXNet provides native support for inference in Java which is one of the reasons we are using it. Nevertheless, we encounter several interesting obstacles on the way to the production-ready pipeline.

Right on the beginning, we had to resolve maven dependencies based on the operating system. MXNet requires different dependencies for macOS, Linux CPU and Linux GPU. Maven profiles can partially resolve library differences, but they can not solve the situation when we need to build on our local machine that is running on macOS and then push the fat JAR to AWS where is running Linux. This situation must be specified by a parameter, so maven uses the right dependency.

■ 7.3.1 Inference and Evaluation

For inference and evaluation, we use java API, which right now lacks the support for training. MXNet provides NArray infrastructure for handling tensors alongside with standard Java float lists. We tried both and adopted the NArray one, while it performed better in our pipeline.

One particularly exciting task was making the whole process to work in a distributed way on Spark. When we created the model on the driver, Scala serialized the object and pushed it to workers. This raised segmentation fault exception deep down in C++ library because of C++ implementation have stored pointers to virtual memory on the driver, which was not present on workers. We solved this issue by creating the lazy implementation in Scala which initialize the predictor at the time of first use which is on workers, at the same time object can be serialized and pushed to individual workers. The only requirement is to copy model definitions to all workers. The last piece of code to implement was a wrapper that took the model and compared it with our other classifiers.

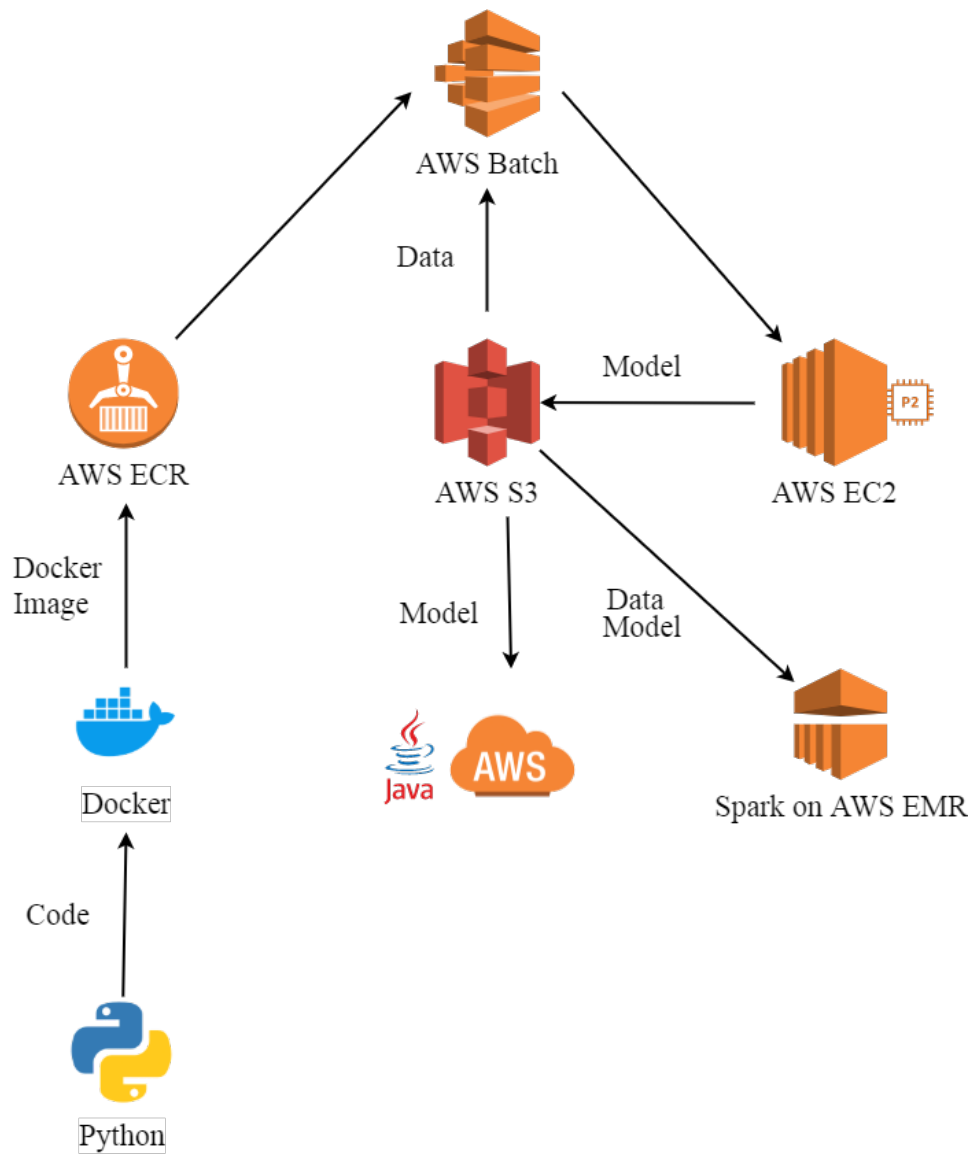


Figure 7.2: Diagram of infrastructure for training, testing and evaluation of models in AWS cloud.

Chapter 8

Experiments

We divide our experiments into two sections. In the first one, we aimed for the model that can run in our production environment. That mainly includes optimization of precision so that the users are not overwhelmed by false positive alarms. Although we expect that our negative data contains some percentage of positive samples; thus, 100% precision is also not desired outcome.

In the second section, we focus on weak labels that would help us extract positive samples from negative data. This model aims to enrich our malware class definitions, based on the similarity with already known ones.

8.1 Dataset description

Our training dataset is obtained from Cisco network telemetry. We use NetFlows that are enriched by information from the initial data packet (IDP), which contains the URLs in unencrypted traffic. All NetFlows go through anomaly layer which filters out 90% of all samples before we save them.

For our negative class, we use unique samples from one month. For the positive class, we doubled the time window to two months and obtained 25 different classes of malware, which we are using only to deliver information about the attack. Thus we are not particularly interested in misclassifications between malicious classes. Labels origins in nonpublic definitions of malware created by our colleagues focused on thread research.

Our testing dataset consists of one week of network telemetry, which follows the training period. This reflects our use-case where we can decide how much samples we use for training, but metrics are correctly computed on data with the real-world distribution. We did not filter or modified datasets in a way that would require human interaction. This creates a situation that we are not used to in the standard datasets; for example, some classes that we train on are not present in the testing dataset. The final ratio of samples is roughly 1500 negative samples to 1 positive.

In Table 8.1 we provide precise information about datasets magnitudes.

	Positive	negative
Train	45894	20844523
Test	3643	5460031
Total	49537	26304554

Table 8.1: Datasets magnitudes.

■ 8.1.1 Malware classes

Our positive samples are made up of all the high-risk malware that has appeared in two months period of our traffic data and have been confirmed by our threat researchers. By high-risk, we mean that all samples are of the same severity and requires fast reaction in the infected system. But their behaviour can be very different from one to another. In our dataset, we see this especially on the frequency of communication. Sality malware dominates in communication frequency. It is a malware distributor that attack windows machines and establish command-and-control through HTTP. Thus we see much communication. In the middle on the scale of communication frequency are trojans and click frauds, that use peer to peer communication for establishing command-and-control or mine bitcoins in the browser. Due to peer to peer communication, we see only hundreds of URLs. Last part of our spectrum is occupied by threats that do not need much of the communication like ransomware which gets into the computer and silently encrypts data and information stealers that do not communicate over the HTTP. More detailed description of malware classes is in Table 8.2.

	Train	Test
malware distribution 1	21818	1516
malware distribution 2	14739	962
trojan 1	7918	971
malware distribution 3	427	52
click fraud 1	199	32
information stealer 1	169	33
malware distribution 4	137	0
click fraud 2	124	10
information stealer 2	97	5
malware distribution 5	87	16
information stealer 3	30	15
banking trojan 1	26	0
trojan 2	18	0
information stealer 4	15	6
click fraud 3	14	3
click fraud 4	12	0
malware distribution 6	10	6
malicious content distribution 1	10	0
trojan 3	8	4
malicious content distribution 2	7	0
banking trojan 2	7	6
ransomware	6	3
information stealer 5	6	3
information stealer 6	6	0
information stealer 7	4	0
Total	45894	3643

Table 8.2: Distribution of positive samples between classes in training and testing dataset.

8.2 Fully supervised model

we present our experiments made with the fully supervised model without any weighting of the loss function. Usage of this model was meant for our production environment with the demand for acceptable precision and high speed.

8.2.1 Experiments with the number of convolution filters

After defining the architecture first hyperparameter to tune was the number of filters of convolution layers. We kept the same amount of filters for both layers for all our experiments. We iterate through 1, 50, 100 and 400 kernels in 250 epochs for each convolution layer. We aimed for identifying the model that has enough capacity for our data. Results of our tests are in Table 8.3.

Epoch 50	Precision	Accuracy	Recall
1_kernel	0.003	0.828	0.979
50_kernel	0.422	0.999	0.998
100_kernel	0.629	0.999	0.968
400_kernel	0.583	0.999	0.998
Epoch 100	Precision	Accuracy	Recall
1_kernel	0.004	0.865	0.975
50_kernel	0.565	0.999	0.993
100_kernel	0.637	0.999	0.970
400_kernel	0.673	0.999	0.997
Epoch 150	Precision	Accuracy	Recall
1_kernel	0.005	0.876	0.988
50_kernel	0.594	0.999	0.997
100_kernel	0.510	0.999	0.996
400_kernel	0.573	0.999	0.998
Epoch 200	Precision	Accuracy	Recall
1_kernel	0.005	0.888	0.980
50_kernel	0.601	0.999	0.996
100_kernel	0.437	0.999	0.990
400_kernel	0.509	0.999	0.997
Epoch 250	Precision	Accuracy	Recall
1_kernel	0.006	0.902	0.974
50_kernel	0.572	0.999	0.997
100_kernel	0.651	0.999	0.996
400_kernel	0.646	0.999	0.995

Table 8.3: In the table are results on our test dataset with different amounts of convolutional filters.

We decided that the model with 100 kernels have with reserve enough capacity and is reasonably fast. We kept it as our baseline for further testing.

8.2.2 Double descent experiments

In Chapter 2, we discussed double descent phenomena and decided to test it on our data. First, we tried L2 regularization with $\sigma = 0.05$ and $\sigma = 0.005$. Then we moved on dropout regularization. We also increased the number of epochs so we could observe double descent behaviour as stated in [42]. Witnessing double descent in under parameterized models (models where the number of parameters is smaller than the number of training samples) is harder [5]. To be sure that we achieve interpolation in our model, we would theoretically need as many parameters as the number of classes multiplied by the number of training samples, which is far more than our biggest model

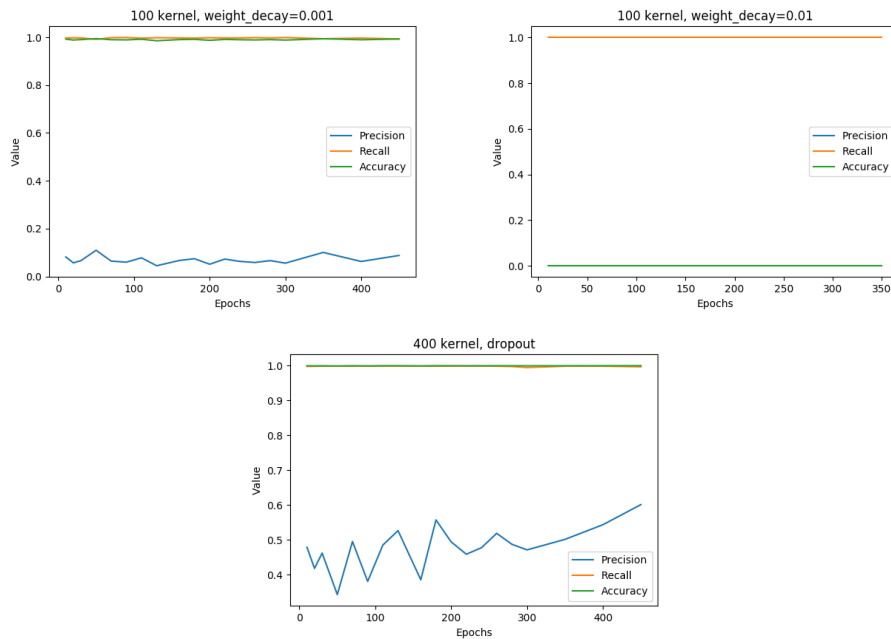


Figure 8.1: Graphs from double descent experiments. We can not observe double descent in any graph. Experiment with L2 regularization and parameter $\lambda = 0.01$ resulted in accuracy and precision around 0.

which consist of $(400 * 5 * 96) + (400 * 4 * 97) + 300 + 100 = 347600$ parameters. We did not observe double descent behaviour in any of the tests, but we also do not have relevant sources on how to proceed with highly imbalanced datasets like ours, where we achieve accuracy of 0.99 right after the start of training. However, we find out that L2 regularization mostly hurt our performance while dropout does not. Thus we kept dropout as a form of regularization in our model. Summarized results are in the Figure 8.1.

8.2.3 Excluding the hostnames

In the experiments in Table 8.4, we excluded hostnames from URL samples. We did these tests because we did not want the model to fit on hostnames but rather to find out patterns of malicious campaigns and prevent them in future when attackers change the server but leave the same attack pattern. Although the performance of our model gets worse, we believe that this approach has its use.

8.3 Weakly labelled model

Our semi-supervised model aims to identify more true positives from the unlabeled part of the dataset. We present experiments that bring more control over precision in imbalanced datasets and an artificially designed scenario on real-world data that shows the value of bringing weak signals into the

Epoch	Precision	Accuracy	Recall
Epoch 50	0.121	0.994	0.996
Epoch 100	0.175	0.996	0.996
Epoch 150	0.117	0.994	0.996
Epoch 200	0.263	0.997	0.995
Epoch 250	0.332	0.998	0.984

Table 8.4: Results of experiment on data with excluded hostnames.

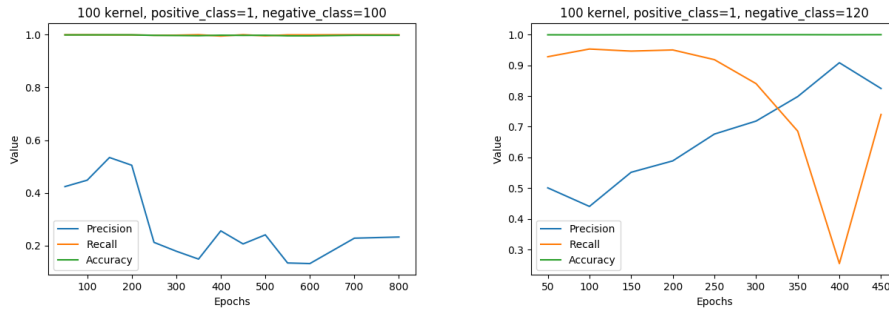


Figure 8.2: Comparison of models with the raised weight of negative class. On the left is the model with negative class weight set to 100 on which we can not observe precision improvement. On the right, we see the model with negative class weighted by 120, which improves precision in later epochs.

training process.

8.3.1 Weighting of positive class

In our fully supervised model, we lack the ability to control the amount of false positive alarms. Simple threshold on positive class did not work for us because the model’s confidence during all predictions achieves near 100%. Hence when we prepared weighting for our semi-supervised experiments, we first tried to reduce false positives by raising the weights of all negative samples in our dataset. With big enough values, we were able to control the boundary between false negative and false positive. In the experiment where we multiplied the weight of negative samples by 100, we can not see much difference from the unweighted one (Table 8.5). The situation changes when we set the weight to 120; in Table 8.6, we can see how precision improves in later epochs. The course of both experiments is shown in Figure 8.2.

8.3.2 Semi-supervised scenario

To prove the usefulness of weak labels, we created a regular expression that describes one of our malicious classes. The regular expression looks like this: `*/images/logo[a-z0-9]?\.`*gif*. Then we assigned a weak negative label to those URLs, which effectively decreased their weight during training to $\epsilon = 10^{-6}$. This simulates the situation when we get unverified information about some

Epoch	Precision	Accuracy	Recall
Epoch 50	0.423	0.999	0.999
Epoch 100	0.447	0.999	0.999
Epoch 150	0.533	0.999	0.999
Epoch 200	0.504	0.999	0.999
Epoch 250	0.212	0.998	0.997
Epoch 300	0.178	0.996	0.998
Epoch 350	0.148	0.996	1.0
Epoch 400	0.255	0.998	0.994
Epoch 450	0.206	0.997	1.0
Epoch 500	0.240	0.997	0.995
Epoch 550	0.133	0.995	0.999
Epoch 600	0.131	0.995	0.999
Epoch 700	0.227	0.997	1.0
Epoch 800	0.232	0.997	0.99972707

Table 8.5: Results of base model with negative class weight set to 100. Left on the Figure 8.2

Epoch	Precision	Accuracy	Recall
Epoch 50	0.500	0.999	0.927
Epoch 100	0.440	0.999	0.953
Epoch 150	0.551	0.999	0.946
Epoch 200	0.588	0.999	0.950
Epoch 250	0.676	0.999	0.918
Epoch 300	0.718	0.999	0.840
Epoch 350	0.798	0.999	0.685
Epoch 400	0.908	0.999	0.254
Epoch 450	0.824	0.999	0.739

Table 8.6: Results of base model with negative class weight set to 120. Right on the Figure 8.2

malware that we might have in unlabeled data. We decrease the weights of all matching records and let the training run. If the information was wrong and we do not have any malicious URLs that would match the pattern, we should observe no weakly labelled data with positive prediction, on the other hand, if the malicious pattern is present in our positive data, we should see that weakly labelled records has been soaked up by positive classes. Hence we were checking if the extracted data were classified as malicious or benign when we found the model that successfully predicted the weakly labelled samples as malicious, we tested the model on our test dataset. We also did similar experiments on a small (thousands of samples) subset of the data. In this pre-experiment, we split one of our malware classes and placed one part to unlabeled data. When we trained the model without the second part in

positive classes, weakly labelled data stayed in the negative class, and when we returned the second part to the positive dataset, weakly labelled records fall back to the positive class. In this small scale experiment, we achieved perfect results. Thus we know that theoretically, this approach should work.

In Table 8.7, we show how our models were predicting 981 samples that fit the regular expression above during training. The first model had positive and negative class weights set both to 1. In the second model, we raised weights for positive classes to 20. And in the third model, we additionally lower the negative class weights to 0.5.

Results of the third model on the testing dataset are shown in Figure 8.3. We achieved best results in epoch 200 where precision is 0.137, accuracy is 0.996 and recall is 0.971.

Epoch	Model 1	Model 2	Model 3
50	161	259	349
100	232	278	399
150	133	230	407
200	140	299	273
250	131	223	416

Table 8.7: Results of semi-supervised training. Model 1 has negative and positive classes weights set to 1. In Model 2, the negative class weight is 1, and the positive classes have a weight set to 20. Model 3 has positive classes weights set to 20, and negative class weights are set to 0.5. Numbers in cells are amounts of weakly labelled samples that have been predicted as positive from total of 981 samples.

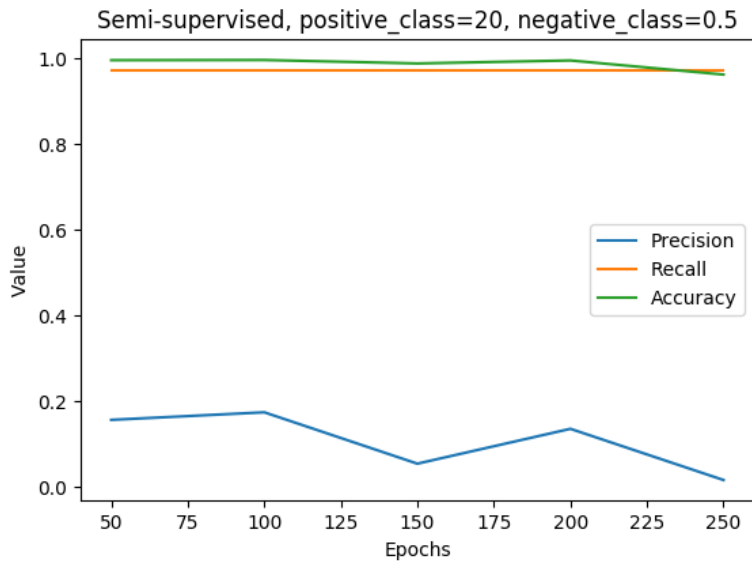


Figure 8.3: Results of Model 3 with negative class weight set to 0.5 and positive classes weights set to 20 on testing dataset.



Chapter 9

Conclusion

The cybersecurity field is still lagging behind the fields like computer vision in various aspects (e.g. good quality public datasets); This can be due to it is an inherently more difficult task and lack of interest; this is slowly changing due to the connection of critical systems to the internet e.g. cars or hospitals. Also, the neural network usually not outperform by huge margin significantly simpler models like random forests which are in contrast to neural networks very fast in inference and easy to scale with industry demands.

In this thesis, we presented a successful pipeline that can be used in the production environment for deploying large scale neural network models in a distributed manner, which partially solves the speed problem and also opens the space for future experiments with different architectures or on different data. We summarized the approach on how to prepare and mix data for training when dealing with an imbalanced dataset. We also implemented code for comparison between neural networks and already used random forest models, which give us the ability to benchmark us with the current state of the art solutions.

On top of the pipeline, we tested our classifier on a real-world positive unlabeled dataset with promising results, which in contrast to solutions on small binary datasets proves our model to be usable in industry. With the usage of class weighting, we can control the amount of false positive alarms, which is a critical aspect for users, so they do not become overwhelmed. We also examined basic generalization techniques and tested double descent hypothesis.

Last but not least, we indicated the possible direction in the use of weak signals during training, that thanks to many available untrusted sources bring valuable information into the training process. We also demonstrated the smooth and straight forward method on how to incorporate weighting to neural network models.

In future research, we would like to focus on testing different architectures of our classifier; namely sequential models, multiple instance learning architectures and batch normalization layers. We would also like to investigate how often is necessary to retrain our model on new data and schedule this retrain to be automatic.

It is also necessary to make further investigation in the are of weak labels

before we can use it in our production environment. We want to design a system for combining multiple weak sources that would give us weights for each sample, as outlined in Chapter 6.



Bibliography

- [1] LI, Xiao-Li; LIU, Bing. Learning from positive and unlabeled examples with different data distributions. In: *European conference on machine learning*. Springer, Berlin, Heidelberg, 2005. p. 218-229.
- [2] ELKAN, Charles; NOTO, Keith. Learning classifiers from only positive and unlabeled data. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008. p. 213-220.
- [3] LIU, Bing, et al. Partially supervised classification of text documents. In: *ICML. 2002*. p. 387-394.
- [4] LI, Xiao-Li; LIU, Bing; NG, See-Kiong. Negative training data can be harmful to text classification. In: *Proceedings of the 2010 conference on empirical methods in natural language processing*. Association for Computational Linguistics, 2010. p. 218-228.
- [5] Mikhail BELKIN, Mikhail, et al. Reconciling modern machine learning practice and the bias-variance trade-off. *arXiv preprint arXiv:1812.11118*, 2018.
- [6] BELKIN, Mikhail; MA, Siyuan; MANDAL, Soumik. To understand deep learning we need to understand kernel learning. *arXiv preprint arXiv:1802.01396*, 2018.
- [7] GEMAN, Stuart; BIENENSTOCK, Elie; DOURSAT, René. Neural networks and the bias/variance dilemma. *Neural computation*, 1992, 4.1: 1-58.
- [8] HASTIE, Trevor; TIBSHIRANI, Robert; FRIEDMAN, Jerome. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009. p. 130-132
- [9] Aston ZHANG, Zachary C. LIPTON, Mu Li, Alexander J. SMOLA. *Dive into Deep Learning, Unpublished draft*. Retrieved, 2020. <https://d2l.ai>
- [10] DEHGHANI, Mostafa, et al. Avoiding your teacher's mistakes: Training neural networks with controlled weak supervision. *arXiv preprint arXiv:1711.00313*, 2017.

- [26] VASWANI, Ashish, et al. Attention is all you need. In: *Advances in neural information processing systems*. 2017. p. 5998-6008.
- [27] HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. *Neural computation*, 1997, 9.8: 1735-1780.
- [28] HLE, Hung, et al. URLnet: Learning a URL representation with deep learning for malicious URL detection. *arXiv preprint arXiv:1802.03162*, 2018.
- [29] SAXE, Joshua; BERLIN, Konstantin. eXpose: A character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys. *arXiv preprint arXiv:1702.08568*, 2017.
- [30] VINAYAKUMAR, R.; SOMAN, K. P.; POORNACHANDRAN, Prabaharan. Evaluating deep learning approaches to characterize and classify malicious URL's. *Journal of Intelligent & Fuzzy Systems*, 2018, 34.3: 1333-1343.
- [31] VINAYAKUMAR R, SRIRAM S, SOMAN KP, MAMOUN Alazab *Malicious URL Detection using Deep Learning, techrxiv preprint 10.36227/techrxiv.11492622.v1*, 2020.
- [32] KALCHBRENNER, Nal, et al. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.
- [33] Jan BRABEC *Decision Forests in the Task of Semi-Supervised Learning*, Prague: CTU 2017. Diploma thesis, CTU Faculty of Electrical Engineering, Department of Cybernetics. <https://dspace.cvut.cz/handle/10467/66851>
- [34] RATNER, Alexander J., et al. Data programming: Creating large training sets, quickly. In: *Advances in neural information processing systems*. 2016. p. 3567-3575.
- [35] COHEN-WANG, Benjamin, et al. Interactive Programmatic Labeling for Weak Supervision. 2019.
- [36] VARMA, Paroma, et al. Learning dependency structures for weak supervision models. *arXiv preprint arXiv:1903.05844*, 2019.
- [37] BACH, Stephen H., et al. Learning the structure of generative models without labeled data. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017. p. 273-282.
- [38] MIRHOSEINI, Azalia, et al. Device placement optimization with reinforcement learning. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 2017. p. 2430-2439.
- [39] BRABEC, Jan, et al. On Model Evaluation under Non-constant Class Imbalance. *arXiv preprint arXiv:2001.05571*, 2020.

- [40] KESKAR, Nitish Shirish, et al. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.
- [41] ImageNet dataset. Available at <http://www.image-net.org/>.
- [42] NAKKIRAN, Preetum, et al. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292*, 2019.



Appendix A

CD content

The root directory of the CD contains:

- [Thesis.pdf], which is this thesis.
- [source code] folder which contains an implementation of our classifier with a training loop code.
- [Dockerfile] which prepares the environment for running the code.