Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Cybernetics

Representation Learning and Adversarial Sample Generation for Strings

BACHELOR THESIS

Marek Galovič

Supervisor: doc. Mgr. Branislav Bošanský, Ph.D.
Study programme: Open Informatics
Specialization: Artificial Intelligence and Computer Science
Date: May 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Galovič Marek**	Personal ID number:	**466367**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Representation Learning and Adversarial Sample Generation for Strings

Bachelor’s thesis title in Czech:

Reprezentace a generování adversariálních řetězců

Guidelines:

In malware behavioral analysis, the list of accessed or created files is very often a strong feature in a classification problem whether the examined file is malicious or benign. However, malware authors are trying to avoid detection based on a simple filename comparison by generating random filenames and/or modifying the filenames (or filename generators) when a new version of malware is released. These changes represent real-world adversarial examples against the malware detection classifier. The goal of the thesis is to:
1. Analyze existing techniques for string / characters-sequence latent representation to be used for such a classifier.
2. Choose an appropriate representation that allows generating realistic adversarial samples of strings (filenames and their paths) using perturbations in the latent space.
3. Analyze the impact of the representation and/or adversarial training using generated samples on the accuracy of the classifier.

Bibliography / sources:

[1] A. Graves; Generating Sequences With Recurrent Neural Networks. arXiv:1308.0850 2013.
[2] X. Zhang, J. Zhao, Y. LeCun; Character-level Convolutional Networks for Text Classification. NeurIPS 2015.
[3] D. P Kingma, M. Welling; Auto-Encoding Variational Bayes. ICLR 2014.

Name and workplace of bachelor’s thesis supervisor:

doc. Mgr. Branislav Bošanský, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

doc. Mgr. Branislav Bošanský, Ph.D. Supervisor’s signature	prof. Ing. Tomáš Svoboda, Ph.D. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

Signature

Acknowledgement

I would like to thank my supervisor doc. Mgr. Branislav Bošanský, Ph.D. and advisor Mgr. Viliam Lisý, Ph.D. for their help and insights that helped me to complete this work, and write this thesis.

Abstract

In malware behavioral analysis, the list of accessed and created files is very often a strong predictive feature for classification whether the examined file is malicious or benign. However, malware authors are trying to avoid detection by generating random filenames, and/or modifying existing filenames with new versions of the malware. These changes represent real-world adversarial examples against the detection classifier. The goal of this work is to learn latent representations of character sequences, generate realistic adversarial examples, and improve the classifier's robustness against adversarial attacks. To obtain fixed-size vector representations of character sequences, we developed a recurrent autoencoder architecture that achieves high sample reconstruction accuracy. Using gradient-based adversarial attacks in the latent representation space, we were able to generate realistic adversarial examples in the input space, and use these adversarial examples to improve the classifier's robustness. Additionally, we showed that latent representations obtained using variational autoencoders improve adversarial robustness without the need for adversarial training.

Abstrakt

V analýze správania malwaru je zoznam vytvorených alebo otvorených súborov často silný príznak pre klasifikačný problém, v ktorom sa rozhoduje či je daný súbor bezpečný alebo nebezpečný. Autori malwaru sa snažia uniknúť odhaleniu s pomocou generovania náhodných názvov súborov, alebo modifikovaním existujúcich názvov súborov v nových verziách malwaru. Tieto zmeny predstavujú adversariálne útoky na detekčný klasifikátor. Cieľom tejto práce je učenie sa latentných reprezentácií znakových reťazcov, generovanie adversariálnych vstupov, a zlepšenie robustnosti klasifikátora voči adversariálnym útokom. Pre učenie sa vektorových reprezentácií znakových reťazcov sme vyvinuli rekurentnú autoenkóder architektúru, ktorá dosahuje vysokú rekonštrukčnú kvalitu. S použitím perturbácií latentných reprezentácií, ktoré sú založené na znalosti gradientu, sme potom boli schopní generovať adversariálne vstupy a použiť tieto adversariálne vstupy na zlepšenie robustnosti klasifikátora. Taktiež sme ukázali, že latentné reprezentácie získané pomocou variačných autoenkóderov zlepšujú adversariálnu robustnosť bez potreby adversariálneho učenia.

Contents

Chapter 1

Introduction

Automated malware detection is an integral part of today's commercial computer security software. Within this domain, the list of accessed and created files is often a strong feature for classifying whether an examined file is malicious or benign. Malware authors are aware of this fact and are trying to avoid detection by generating random filenames and/or modifying existing filenames with new versions of the malware. As these changes are designed to maximize the chance of classifying malicious code as benign, they represent real-world adversarial examples against the detection classifier. How successful attacks using these adversarial examples are depends primarily on how much knowledge the adversary has about the target classifier, and on the robustness of the classifier against adversarial examples. In this work, we focus on the latter part and aim to develop methods for training string classifiers that are robust against adversarial examples.

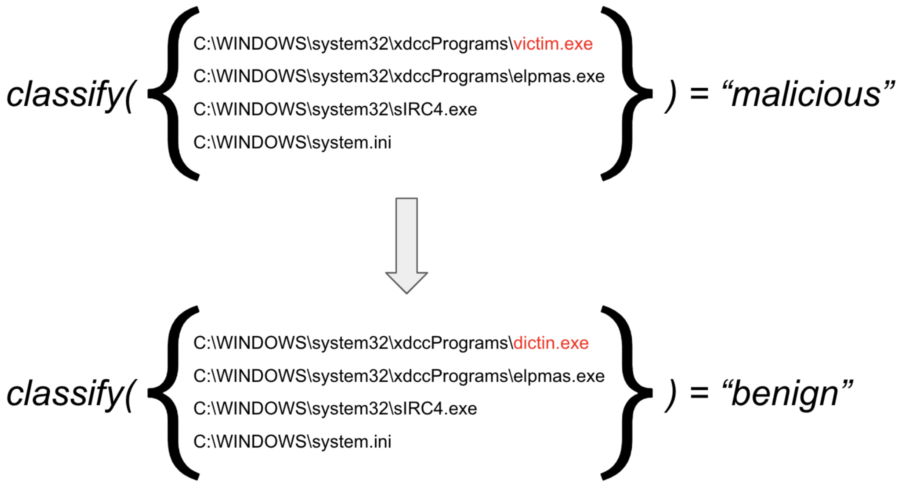

classify(
C:\WINDOWS\system32\xdccPrograms\victim.exe
C:\WINDOWS\system32\xdccPrograms\elpmas.exe
C:\WINDOWS\system32\sIRC4.exe
C:\WINDOWS\system.ini
) = “malicious”
classify(
C:\WINDOWS\system32\xdccPrograms\dictin.exe
C:\WINDOWS\system32\xdccPrograms\elpmas.exe
C:\WINDOWS\system32\sIRC4.exe
C:\WINDOWS\system.ini
) = “benign”

Figure 1.1: Desired adversarial perturbation

Adversarial perturbations were first introduced by Szegedy et al. [1] and later explored by Goodfellow et al. [2] in the domain of computer vision. They describe perturbations of input images that are imperceptible by humans but cause the image classifier to assign an incorrect label for the perturbed input. More importantly, different models trained on different subsets of data can misclassify the same adversarial example. This suggests that it is possible to generate adversarial examples using a surrogate model to which the attacker has full access, and then use these adversarial examples to attack an unknown model of interest. Computer vision is not the only domain in which adversarial examples exist. They were also explored, among others, in the field of natural language processing, which is relevant to ours because it operates on string inputs. Main approaches for adversarial attacks in the natural language processing domain can be categorized as sentence-level attacks [3, 4], word-level attacks [5, 6, 7], and character-level attacks [8, 9]. Sentence-level and word-level attacks seem to exploit high-level semantic information, and often make use of synonyms or sentences with the same meaning as adversarial perturbations. Arguably, file paths lack the same level of semantic richness, and therefore it makes more sense to focus on character-level adversarial perturbations. Another important consideration is the level of information an attacker has about the target model. Generally we can categorize adversarial attacks as *white-box*, *gray-box* and *black-box*. White-box attacks assume that the attacker has full access to model parameters, and can compute gradients of the output w.r.t. the input. Knowledge of gradients is very useful because it can be used to directly optimize the adversarial perturbation. Gray-box attack represents a more restrictive approach and assumes that only the model's decisions or output probabilities are known. While not as useful as gradients, model outputs can be used to train an approximation of the target model, which can then be used to attack using the white-box method. Black-box method is the most restrictive one and assumes no knowledge about the target model.

We are interested in a minimal sequence of perturbations that results in a given set of strings being misclassified (Figure 1.1). Unlike in the image domain, any change in the set of strings will be perceptible by humans, and therefore we are interested in perturbations that produce adversarial strings which are close to the original strings. Theoretically, we are guaranteed to find a minimal sequence of adversarial perturbations using combinatorial search algorithms, but these methods do not work well in practice due to the cardinality of the search space. Additionally, strings are discrete inputs and thus we cannot directly use the model's gradients to find an adversarial perturbation. To make adversarial sample generation practically feasible in the domain of character sequences, we instead focus on adversarial perturbations in the continuous latent representation space. As we have full access to the model, we can leverage its gradients to compute an adversarial perturbation of the latent representation which we can then use to directly generate an adversarial string. The ability to generate realistic adversarial examples very quickly compared to combinatorial search methods allows us to perform adversarial training of our classifier. We augment the training dataset with adversarial examples and use it to further train our detection classifier and improve its robustness against adversarial attacks.

Text classification is a well-studied problem in the field of natural language processing. Common tasks include sentiment classification, topic labeling, intent classification, language detection, etc. In this work, we are interested in classifying sets of strings, specifically file paths, as either malicious or benign. In order to use standard classification methods such as logistic regression, support vector machines, or neural networks on sets of strings, we first represent individual strings in the set as fixed-size vectors and then transform the set of string representations to a fixed-size vector. This fixed-size vector is then used as input features to one of the aforementioned classifiers. Traditional techniques for representing strings as vectors include one-hot encoding, n-gram counts, TF-IDF features, or more recently Word2Vec [10], recurrent neural networks [11], and transformers [12, 13, 14]. We also want to generate adversarial strings from perturbed latent representations which further constraints our choice to recurrent neural networks or transformers. While transformers achieve state-of-the-art results in the natural language domain, they are not very suitable for our use-case as their latent representation is a variable-length sequence of latent vectors, not a fixed-size vector. Therefore, we choose to use the recurrent neural network architecture to encode strings to fixed-size vectors and to decode fixed-size vectors to strings. Inputs to the recurrent neural network are character embeddings and outputs of the recurrent neural network are probability distributions over characters. Using character-level representation instead of token-level representation is beneficial because we aim to generate character-level adversarial perturbations, and because our domain contains a large number of unique tokens.

1.1 Structure of the Thesis

Chapter 2 presents an overview of existing algorithms and neural network architectures that are relevant to our work.

In Chapter 3 we describe the main contributions of this work. We start by presenting our solution to learning latent representations of character sequences in an unsupervised fashion (Section 3.1). Section 3.2 then describes methods for regularizing the latent representation space in order to generate new examples from it. In Section 3.3 we outline multiple instance classification and propose learnable bag aggregator, which is a novel instance aggregation function based on the transformer architecture. Finally, Section 3.4 proposes generating adversarial examples by perturbing latent representations, and Section 3.5 describes methods for training robust classifiers.

In Chapter 4 we provide details of our experimental setup and report achieved results.

Chapter 2

Background

This chapter provides an overview of relevant algorithms and architectures used throughout this work. Given the sequential nature of our data, we focus on using recurrent neural networks to encode strings as fixed-size vectors and to decode vector representations to strings. We learn both the encoder and the decoder functions in an unsupervised fashion by leveraging the autoencoder architecture. Additionally, we use 1D convolutional neural networks to reduce the temporal dimension of input sequences which substantially improves sample reconstruction quality. 1D convolution kernel also learns implicit n-gram representations which are advantageous over explicit n-gram input representation because our data contains a large number of unique n-grams. Using latent representations of strings as features, we train a multiple instance classifier to classify sets of strings as malicious or benign. Gradient-based adversarial attack methods are used to compute perturbations of latent representations, which are then used to generate adversarial strings and improve the classifier's robustness against adversarial examples. We also explore methods for regularizing the latent representation space in order to improve the quality of adversarial strings generated from perturbed latent representations.

2.1 Recurrent Neural Networks

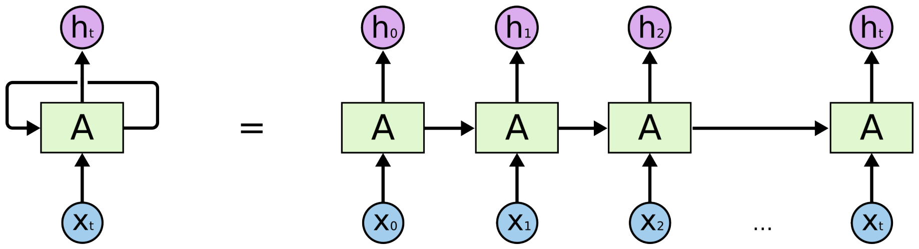


Figure 2.1: Unrolled Recurrent Neural Network [15]

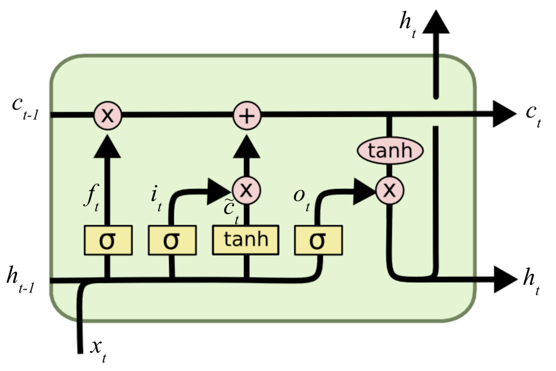
Recurrent neural networks (RNN) are a class of artificial neural networks that allow outputs at step t to be used as inputs at step $t+1$. They usually do this by maintaining a hidden state h that is passed between steps (Equation 2.1). The hidden state acts as a memory that summarizes inputs to the recurrent neural network up until step t. It is updated using a recurrence relation described by Equation 2.2, where h_{t-1} is the previous hidden state, x_t is the current input, W_h, W_x, b_h are trainable parameters and σ is a nonlinear activation function. The main advantage of recurrent networks is their ability to process inputs of variable length without changing the number of model parameters.

$$\hat{y_t},\ h_t = rnn(x_t, h_{t-1}) \tag{2.1}$$

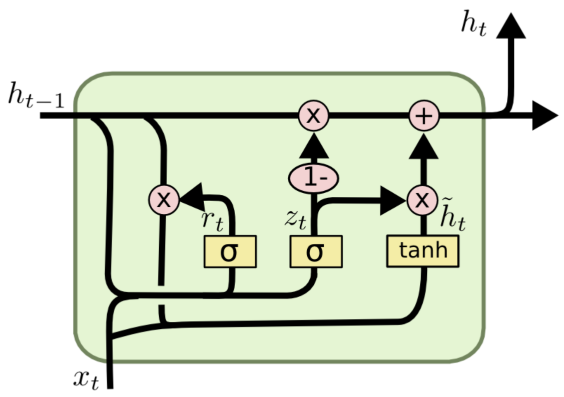
$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h) \tag{2.2}$$

$$\hat{y_t} = \sigma(W_y h_t + b_y) \tag{2.3}$$

However, there is a number of challenges with vanilla recurrent neural networks as described by equations (2.1) - (2.3), namely, forgetting and vanishing/exploding gradient. Because the information stored in the hidden state h_t can be easily changed, vanilla recurrent neural networks have issues accessing information from a long time ago. Optimization of such models is also difficult because backpropagating gradient through time can cause it to exponentially increase/decrease with respect to the input sequence length. Long short-term memory cell [16], and its later simplification Gated recurrent unit cell [17] were designed to alleviate these issues, and made recurrent neural networks usable in practice.



(a) LSTM



(b) GRU

Figure 2.2: Recurrent neural network cells [15]

2.1.1 Long Short-Term Memory Cell

Long short-term memory (LSTM) cell proposed by Hochreiter and Schmidhuber in [16] addresses forgetting and vanishing/exploding gradients by introducing a series of multiplicative gates f_t, i_t and o_t (Equation 2.4, 2.5, 2.8). The input gate i_t and the forget gate f_t are used to decide when to keep or override information stored in memory. Similarly, the output gate o_t is used to decide when information stored in memory should be accessed. Parameters of all gates are jointly optimized during the training of the network.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.4}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.5}$$

$$\tilde{c_t} = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.6}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c_t} \tag{2.7}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.8}$$

$$h_t = o_t \odot tanh(c_t) \tag{2.9}$$

Forget gate f_t, input gate i_t, and candidate cell state $\tilde{c_t}$ are computed as affine projections of previous hidden state h_{t-1} and current input x_t (Equation 2.4, 2.5, 2.6). Nonlinear activation functions σ and $tanh$ are applied to the affine projection to bound the output. New cell state c_t is then computed according to Equation 2.7, where $\odot$ is an element-wise multiplication. Output h_t is again a gated version of updated cell state c_t, where gating values o_t are computed in the same way as forget and input gates (Equation 2.8).

2.1.2 Gated Recurrent Unit Cell

Gated recurrent unit (GRU) cell proposed by Cho et al. [17] simplifies the LSTM cell by combining input and forget gates into a single update gate z_t (Equation 2.10). It also removes the cell state and uses only the hidden state to maintain information between steps. A reset gate r_t is used to selectively access information stored in the previous hidden state h_{t-1} (Equation 2.11).

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \tag{2.10}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \tag{2.11}$$

$$\tilde{h_t} = tanh(W_h \cdot [r_t \odot h_{t-1}, x_t]) \tag{2.12}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \tag{2.13}$$

Update gate z_t and reset gate r_t are computed as linear projections of previous hidden state h_{t-1} and current input x_t with nonlinear activation σ (Equation 2.10, 2.11). Candidate hidden state $\tilde{h_t}$ is then computed as a linear projection of gated previous hidden state h_{t-1} and current input x_t with nonlinear activation $tanh$ (Equation 2.12). Finally, cell output h_t is a linear interpolation between previous hidden state h_{t-1} and candidate hidden state $\tilde{h_t}$, where $\odot$ is an element-wise multiplication (Equation 2.13).

2.2 Convolutional Neural Networks

Convolutional neural networks (CNN) are primarily used in the field of computer vision as translation invariant feature extractors. Unlike fully connected neural networks which connect all input values to all output values, convolutional neural networks apply the same kernel to map parts of input to respective parts of output (Figure 2.3). Formally, this is written as Equation 2.14, where X is the input, W is the convolution kernel, and $*$ is the convolution operator.

$$Y = X * W \tag{2.14}$$

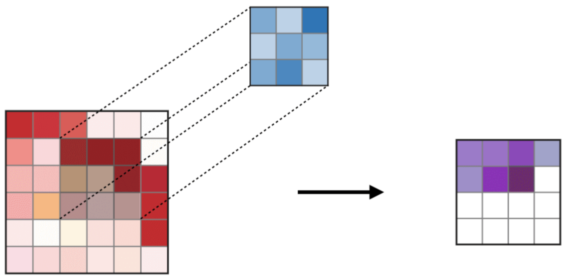


Figure 2.3: 2D Convolution [18]

2.2.1 1D Convolution

1D convolutions apply the kernel along only one of the input dimensions. Therefore, 1D convolutions are a suitable architecture for sequence modeling tasks. Some of their uses include text classification [19], neural machine translation [20], and audio generation [21]. In this work, we use 1D convolutions to learn position invariant n-gram representations.

2.3 Autoencoder

Autoencoder is a neural network architecture for unsupervised representation learning. Autoencoder architecture has generally two parts, namely an encoder and a decoder. The encoder F takes a high dimensional input $x \in \mathbb{R}^m$ and outputs a low dimensional representation $z \in \mathbb{R}^n$, where $n << m$. The decoder G then takes z as its input and attempts to reconstruct the original, high dimensional input x. This encoding-decoding scheme creates a bottleneck for the data and ensures that the low dimensional representation z contains only important information necessary to reconstruct the original input. Parameters of both the encoder and the decoder networks are jointly optimized by minimizing a reconstruction objective $\mathcal{L}(x, G(F(x))$ using gradient descent. A nice property of this framework is its generality. We can choose the encoder to be any differentiable function $F : \mathbb{R}^m \to \mathbb{R}^n$, and similarly the decoder can be any differentiable function $G : \mathbb{R}^n \to \mathbb{R}^m$.

A major drawback of this approach is the lack of regularity in the representation space. What this means is that samples that are close in the input domain can have their latent representations far apart. Vice versa, latent representations that are close can have their respective decoded samples far apart in the input domain. It is also not possible to randomly sample a point from the latent representation space, and expect the decoded output to come from the data distribution.

2.4 Variational Autoencoder

Similar to the autoencoder architecture described in Section 2.3, variational autoencoder [22] is a neural network architecture for unsupervised representation learning and content generation. It addresses the irregularity of the latent space by representing it as a distribution over latent vectors rather than a point-like estimate. This allows the regularization of the latent space to be expressed in terms of Kullback-Leibler divergence between the encoder distribution and a standard Gaussian distribution $\mathcal{N}(0, I)$. Kullback-Leibler divergence is then minimized jointly with the reconstruction objective.

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{2.15}$$

The optimization objective can be derived using a probabilistic framework. We assume a prior distribution $p(z) \equiv \mathcal{N}(0, I)$. The decoder can then be represented as $p(x|z)$, which describes a conditional distribution of data x given latent variable z. Similarly, the encoder can be represented as $p(z|x)$, which describes a conditional distribution of latent variable z given data x. From Bayes theorem we see that $p(z|x)$ is assumed to follow a prior distribution $p(z)$ (Equation 2.15).

$$q_x(z) \equiv \mathcal{N}(F_\mu^\theta(x), F_{\sigma^2}^\omega(x)). \tag{2.16}$$

Using variational inference [23], we can approximate the posterior distribution $p(z|x)$ with $q_x(z)$ defined in Equation 2.16. Function $F_\mu^\theta : \mathbb{R}^m \to \mathbb{R}^n$ is a mapping from input x to the mean of a Gaussian distribution μ parametrized by θ. Function $F_{\sigma^2}^\omega : \mathbb{R}^m \to \mathbb{R}^n$ is a mapping from input x to the diagonal of a Gaussian covariance matrix σ^2 parametrized by ω. Optimal parameters $\theta^* \in \Theta$ and $\omega^* \in \Omega$ are found according to Equation 2.17.

$$\begin{aligned}
(\theta^*, \omega^*) &= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} KL(q_x(z)||p(z|x)) \\
&= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} \int_z q_x(z) \log \frac{q_x(z)}{\frac{p(x|z)p(z)}{p(x)}} \\
&= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} \int_z q_x(z) \left[\log q_x(z) - \log \frac{p(x|z)p(z)}{p(x)}\right] \\
&= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} \int_z q_x(z) \left[\log q_x(z) - \log p(x|z) - \log p(z) + \log p(x)\right] \\
&= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} KL(q_x(z)||p(z)) + \int_z q_x(z) \left[-\log p(x|z) + \log p(x)\right] \\
&= \operatorname*{argmin}_{(\theta,\omega)\in\Theta\times\Omega} KL(q_x(z)||p(z)) - \mathbb{E}_{z\sim q_x(z)} \log p(x|z)
\end{aligned} \tag{2.17}$$

Hence, optimal parameters θ^* and ω^* minimize Kullback-Leibler divergence between the approximate posterior $q_x(z)$ and prior $p(z)$, and maximize the log likelihood of generating data x given a latent vector z.

2.5 Triplet Loss

Triplet loss is an optimization objective used for semi-supervised representation learning [24, 25]. It can be used to regularize the latent representation space, such that the distance between a pair related input examples is smaller than the distance between a pair of unrelated input examples. A hyperparameter $margin \in \mathbb{R}$ is used to introduce a gap between embeddings of related and unrelated examples. Given an encoder function $F : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and a triplet of input examples x_i^a (anchor), x_i^p (positive) and x_i^n (negative), we want Equation 2.18 to hold true $\forall(x_i^a, x_i^p, x_i^n) \in \mathcal{T}$.

$$||F(x_i^a) - F(x_i^p)||_2^2 + margin < ||F(x_i^a) - F(x_i^n)||_2^2 \tag{2.18}$$

This directly translates to an objective function $\mathcal{L}_{triplet}$ (Equation 2.19), which is jointly minimized during autoencoder training to regularize the latent representation space.

$$\mathcal{L}_{triplet} = \sum_{i}^{|\mathcal{T}|} max(||F(x_i^a) - F(x_i^p)||_2^2 - ||F(x_i^a) - F(x_i^n)||_2^2 + margin, 0) \tag{2.19}$$

2.6 Adversarial Examples

Adversarial examples [1] can be described as perturbed data examples $x_{adv} = x + \delta$, where $\delta \in \Delta$ is an adversarial perturbation, and Δ is a set of allowable perturbations. The goal of an adversary is to find a perturbation δ which changes the classifier's prediction from the correct label to an incorrect one.

Given a trained classifier f_θ parametrized by θ, input example $x_i \in \mathbb{R}^m$, ground truth label $y_i \in \mathbb{Z}_+$ and a loss function ℓ, we want to find an adversarial perturbation $\delta \in \Delta$ that maximizes $\ell(f_\theta(x_i + \delta), y_i)$.

$$\delta^* = \underset{\delta \in \Delta}{\operatorname{argmax}}\, \ell(f_\theta(x_i + \delta), y_i) \tag{2.20}$$

A commonly used set of allowable perturbations Δ is the ℓ_∞ ball (Equation 2.21), but other norms can be used as well.

$$\Delta = \{\delta : ||\delta||_\infty \leq \epsilon\} \tag{2.21}$$

The constrained optimization task (Equation 2.20) is an example of an untargeted adversarial attack which simply attempts to change classifier's prediction to an incorrect class. It is also possible to perform a targeted attack, where we force the classifier to change its prediction to a desired class y_{target} (Equation 2.22).

$$\delta^* = \underset{\delta \in \Delta}{\operatorname{argmax}}\, [\ell(f_\theta(x_i + \delta), y_i) - \ell(f_\theta(x_i + \delta), y_{target})] \tag{2.22}$$

2.6.1 Projected Gradient Descent

Projected gradient descent (PGD) is an iterative algorithm for solving the constrained optimization task defined in Equation 2.20. To find the adversarial perturbation δ, projected gradient descent algorithm iteratively updates δ according to Equation 2.23 for T steps, where $\delta_0 = 0$, α is the update step size, and $\mathcal{P}$ is a projection operator which ensures that $\delta_t \in \Delta; \forall t = 1...T$.

$$\delta_{t+1} = \mathcal{P}(\delta_t + \alpha \nabla_\delta \ell(f_\theta(x + \delta_t), y)) \tag{2.23}$$

For the case where the allowable set of perturbations is $\Delta = \{\delta : ||\delta||_\infty \leq \epsilon\}$, the projection operator $\mathcal{P}$ simply clips the updated estimate of δ component-wise, such that it lies within the range $[-\epsilon, \epsilon]$. Note that the algorithm is commonly known as projected gradient descent but it actually performs gradient ascent on δ to maximize the loss ℓ.

2.6.2 Fast Gradient Sign Method

Fast gradient sign method (FGSM) is a one-step algorithm for solving the constrained optimization task described in Equation 2.20 proposed by Goodfellow et al. [2]. It assumes that the set of allowable perturbations Δ is ℓ_∞ ball as defined in Equation 2.21, and that the response of classifier is linear.

$$\delta^* = \epsilon sgn(\nabla_\delta \ell(f_\theta(x + \delta), y)) \tag{2.24}$$

Optimal adversarial perturbation δ^* is then computed as ϵ-multiple of the sign function applied to gradient of ℓ w.r.t. δ (Equation 2.24).

$$\begin{aligned} \nabla_\delta \ell(f_\theta(x + \delta), y) &= \nabla_\delta \theta_y^T(\delta) \\ &= \nabla_\delta \theta_y^T x + \nabla_\delta \theta_y^T \delta \\ &= \theta_y \end{aligned} \tag{2.25}$$

The linear response assumption can be written as $\ell(f_\theta(x + \delta), y) = \theta_y^T(x + \delta)$ with the gradient w.r.t δ derived in Equation 2.25. Given the constraint $||\delta||_\infty \leq \epsilon$, the classifier's response to the adversarial example is maximized by setting $\delta = \epsilon sgn(\theta_y) = \epsilon sgn(\nabla_\delta \ell(f_\theta(x + \delta), y))$.

2.7 Adversarial Training

The goal of adversarial training is to train a classifier that is robust to adversarial attacks. It modifies the optimization objective from a simple minimization of the classification loss function $\ell(f_\theta(x), y)$ to a minimax objective (Equation 2.26).

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \max_{\delta \in \Delta} \ell(f_\theta(x + \delta), y) \tag{2.26}$$

The adversary tries to maximize the classifier's loss on the perturbed input $x + \delta$, while the outer minimization tries to minimize it. We assume that the inner maximization has full access to the classifier, and that it can leverage its gradients to generate adversarial perturbations.

2.8 Multiple Instance Learning

Multiple Instance Learning (MIL) formalism assumes that each sample is represented as a bag of instances, where labels are available for the whole bag but not necessarily for individual instances in the bag. Bags $b \in \mathcal{B}$ can be thought of as unordered sets that contain arbitrarily many, possibly duplicated, instances $x \in \mathcal{X}$ (Equation 2.27). Instances are usually represented by feature vectors which are outputs of some instance feature extractor.

$$b = \{x_i \in \mathcal{X} | i \in \{1...|b|\}\}; b \in \mathcal{B} \tag{2.27}$$

Prior work describes three main paradigms for solving multiple instance learning problems, namely, *instance-space paradigm*, *bag-space paradigm* and *embedding-space paradigm*. Approaches following *instance-space paradigm* [26, 27] assume that a classifier $f : \mathcal{X} \rightarrow \{-1, 1\}$ is trained on individual instances $x \in \mathcal{X}$, and the label of the bag b is then determined as $\hat{y} = \max_{x \in b} f(x)$. More general approaches do not assume that the instance level classifier is available, and only consider labels on the bag level. *Bag-space paradigm* [28, 29] defines a measure of distance between bags, and *embedding-space paradigm* [30, 31] defines a transformation function from bags to fixed-size vectors.

Chapter 3

Methods

In this chapter, we describe in detail methods used to generate string adversarial samples using perturbations in the latent representation space. We first describe our approach for representing strings as fixed-size vectors, where we leverage recurrent autoencoders with 1D convolutions. Next, we explore triplet loss and variational loss which we use to regularize the latent space of the autoencoder in order to improve its content generation capability. Because our input samples are bags of strings, we also describe our approach to multiple instance classification and propose a novel attention-based bag aggregation function. Finally, we use the classifier's gradients to compute perturbations of latent representations and generate adversarial strings from these perturbed latent representations.

3.1 Representation Learning

We aim to represent variable-length ASCII-encoded sequences of characters as fixed-size vectors in $\mathbb{R}^d$. While there exist many approaches for obtaining such representations [19, 12], we focus specifically on using recurrent neural networks. Because we want to generate adversarial strings from perturbed latent representations, we are also interested in reconstructing the input sequence given its vector representation. Therefore, we use the autoencoder architecture described in Section 2.3 to both encode the input sequence to its latent representation, and subsequently decode this representation to a sequence of characters.

3.1.1 Character Embedding

ASCII characters can be encoded as 256-dimensional one-hot vectors in a straightforward way [19]. While simple, one-hot encoding of characters has several issues. First, it results in unnecessarily large embeddings and second, the pair-wise distance between character embeddings is the same for all characters. For example, we would like to embed alphabetical characters closer to other alphabetical characters than to numerical characters. To address these issues, we choose to represent ASCII characters as real-valued vectors in $\mathbb{R}^e$, where $e = 32$ is the embedding size [32]. The mapping from one-hot vectors to real-valued vectors is done via an embedding function $E_\Theta : \{0, ..., 255\} \rightarrow \mathbb{R}^e$, where Θ is a learnable embedding matrix $\Theta \in \mathbb{R}^{256 \times e}$. Using this formulation, the embedding for i-th character is then computed as $x = e_i \cdot \Theta$, where $e_i \in \mathbb{R}^{256}$ is the i-th canonical basis vector.

3.1.2 Recurrent Autoencoder

Recurrent autoencoders (Figure 3.1) are a standard method for unsupervised representation learning of variable-length sequential data. The encoder $F : \mathbb{R}^{T \times e} \rightarrow \mathbb{R}^d$ is a recurrent neural network that takes a sequence of embedded characters with length T as its input, and produces a sequence of outputs $o_t; t = 1...T$ and cell states $h_t; t = 1...T$. We are particularly interested in the last cell state h_T, as it is the latent representation of the whole input sequence. The decoder $G : \mathbb{R}^d \rightarrow \mathbb{R}^{T \times d}$ is also a recurrent neural network. However, unlike the encoder, the decoder does not have access to the input sequence, but instead uses its outputs at time $t-1$ as inputs at time t. Hence, it autoregressively decodes the output sequence $o_t; t = 1...T$ using only information encoded in the last encoder cell state h_T which is used as its initial cell state.

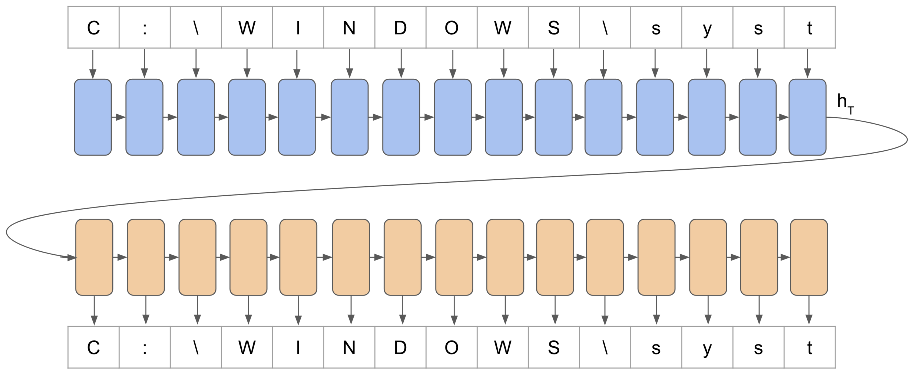


Figure 3.1: Recurrent sequence-to-sequence autoencoder

We project each decoder output $o_t \in \mathbb{R}^d$ to $\tilde{o}_t \in \Delta^{256}$, where Δ^{256} is a 256-dimensional unit simplex. Every $\tilde{o}_t$ then represents a categorical probability distribution over ASCII characters.

$$\mathcal{L}^i_{reconstruction} = -\sum_{t=1}^{T} y^i_t \cdot \log(\tilde{o}_t{}^i) \tag{3.1}$$

Considering the setup described above, we can jointly train both the encoder and the decoder by minimizing piecewise cross-entropy between one-hot encoded inputs $y_t \in \{0, ..., 255\}$ and outputs $\tilde{o}_t$. Minimizing Equation 3.1 in turn maximizes the reconstruction quality of the autoencoder.

The main issue with vanilla sequence-to-sequence recurrent autoencoders is that their reconstruction quality suffers for long sequences. This can be primarily attributed to vanishing and exploding gradients when backpropagating through time during training, and forgetting information in the cell state during sequence generation. Previous work [33] tries to address this issue by introducing an attention mechanism. The attention mechanism allows the decoder to selectively access all encoder outputs at each decoding step, and therefore alleviates the need to rely solely on the information encoded in the last encoder state. However, we require the decoder to only access the last encoder state in order to generate adversarial samples from its perturbed version, and therefore using the attention mechanism does not work for us.

3.1.3 Convolutional Recurrent Autoencoder

Because we cannot introduce additional ways for information to flow from the encoder to the decoder, we instead aim to reduce the length of the input to the recurrent neural network. There exist multiple approaches to achieve this, namely, changing the input representation from characters to words or n-grams, using only top-k most common words/n-grams, or applying 1D convolutions. We experimented with word level and n-gram level representations and realized that this approach does not work for us. Unlike natural language, where word level or n-gram level representations work well, file paths contain a lot of randomly generated strings which are still valid as directory names or file names. Representing such strings on the word level requires a very tall embedding matrix with entries for words that may only occur a small number of times in the training set. It also increases the size of decoder's output probability distribution to the same size as the number of rows in the embedding matrix which makes the optimization more difficult. Another option is to use n-gram representation which requires precisely 256^n entries in the embedding matrix, and increases the size of decoder's output probability distribution to 256^n. A common technique to reduce the number of entries in the embedding matrix is to encode only the top-k most common words/n-grams in the training set. In our data, we have very few common entries and a lot of unique entries which we still want to encode and decode, and therefore reducing the number of entries does not work for us.

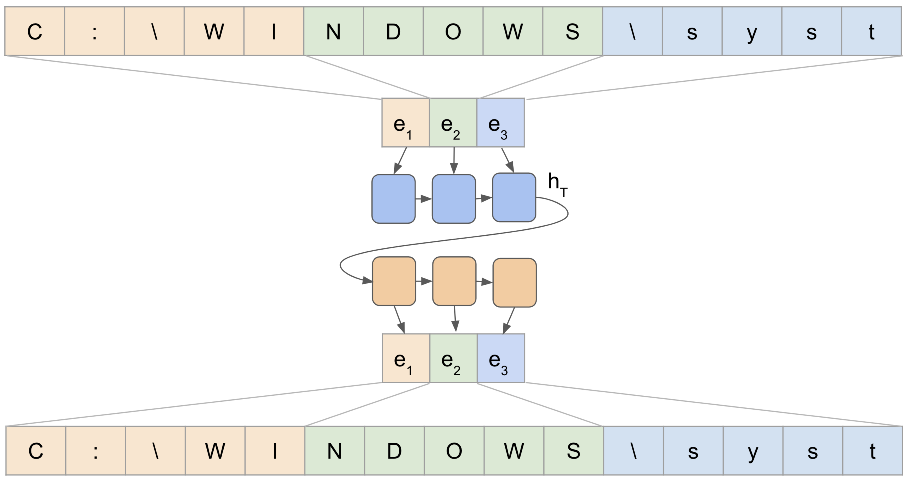


Figure 3.2: Convolutional Recurrent Sequence-to-Sequence Autoencoder

Instead of changing the input representation from characters to words or n-grams, we apply a 1D convolution layer with $kernelWidth = 5$ and $stride = 5$ to the sequence of character embeddings. Values of $kernelWidth$ and $stride$ were chosen empirically. The convolution kernel learns an implicit representation of 5-grams, and $stride = 5$ effectively reduces the sequence length by a factor of 5. Latent representations of 5-grams are then used as inputs to a recurrent autoencoder which has the same architecture as described in Section 3.1.2 and outputs raw decoder vectors, not categorical probability distributions. The recurrent decoder also decodes reduced-length sequences which need to be upsampled by a factor of 5, so that they have the same length as input. The upsampling is done using a 1D deconvolution layer with the same $kernelWidth$ and $stride$ as the aforementioned convolution. Finally, we project the deconvolution layer outputs to categorical probability distributions over ASCII characters, and minimize the piecewise cross-entropy (Equation 3.1).

3.2 Latent Space Regularization

In order to generate adversarial examples from perturbed latent representations, we need the latent representation space to be regular. Specifically, we want to preserve metric properties between the input domain and the latent domain for some input domain distance measure d_i, and some latent domain distance measure d_l. For example, given strings s_1, s_2, s_3 for which there holds $d_i(s_1, s_2) \leq d_i(s_1, s_3)$, we want their respective latent representations z_1, z_2, z_3 to satisfy $d_l(z_1, z_2) \leq d_l(z_1, z_3)$. Vice versa, given latent representations z_1, z_2, z_3 for which there holds $d_l(z_1, z_2) \leq d_l(z_1, z_3)$, we want their respective decoded strings $\hat{s_1}, \hat{s_2}, \hat{s_3}$ to satisfy $d_i(\hat{s_1}, \hat{s_2}) \leq d_i(\hat{s_1}, \hat{s_3})$.

Minimizing only the reconstruction objective of the autoencoder does not provide any guarantees on the regularity of the latent space. Therefore, we explore using additional regularization objectives on the latent space to approximate the desired metric preserving properties described above.

3.2.1 Triplet Loss

We experiment with a synthetically generated dataset, where triplets are constructed in the following way: first we generate a random path (anchor), and then using one of the perturbations described below we obtain positive and negative examples.

1. **Change extension 1**
    - Positive: change extension within family
    - Negative: change extension across families
2. **Change extension 2**
    - Positive: different filename but same extension as original
    - Negative: same filename but different extension (family) as original
3. **Change path 1**
    - Positive: mutate path at depth $< d, N >$
    - Negative: mutate path at depth $< 0, d)$

File extensions are generated by first choosing an extension family at random, and then choosing an extension from this family. We use extension families to approximate the fact that filenames with extensions from the same family, e.g. *picture.jpg* and *picture.png*, are more similar than filenames with extensions from different families, e.g. *picture.jpg* and *picture.exe*. Extension families are randomly generated at the beginning, and remain fixed during the course of training and testing. Mutation depth d is randomly sampled from the range $[1, depth(path) - 1]$, where $depth(path)$ is the number of directories in the path $+1$ for the filename.

3.2.2 Variational Autoencoder

We use an affine transformation to obtain parameters μ and σ^2 of the encoder latent distribution $q_x(z)$. Given a latent representation $\tilde{z}_i = F_\theta(x_i)$, where F_θ is a deterministic encoder, we define $\mu_i = \theta_\mu^T \cdot \tilde{z}_i + b_\mu$ and $\sigma_i^2 = \exp(\theta_\sigma^T \cdot \tilde{z}_i + b_\sigma)$. Decoder input z_i is then sampled from a multivariate Gaussian distribution parametrized by μ_i and σ_i^2 (Equation 3.2).

$$z_i \sim \mathcal{N}(\mu_i, diag(\sigma_i^2)) \tag{3.2}$$

Because sampling from a distribution is not differentiable, we use the reparametrization trick to obtain decoder input z_i from μ_i and σ_i^2. First, we sample $q_i \sim \mathcal{N}(0, I)$, and then we reparametrize $\mathcal{N}(0, I)$ as $z_i = \mu_i + q_i \odot \sqrt{\sigma_i^2}$, where $\odot$ is element-wise multiplication.

$$\begin{aligned}
KL(q_{x_i}(z)||p(z)) &= \int_z q_{x_i}(z) \log \frac{q_{x_i}(z)}{p(z)} \\
&= \int_z q_{x_i}(z) \log \left[\frac{\frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma_i|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(z-\mu_i)^T \Sigma_i^{-1} (z-\mu_i)\right)}{\frac{1}{(2\pi)^{\frac{n}{2}}} \exp\left(-\frac{1}{2} z^T z\right)} \right] \\
&= \mathbb{E}_{z \sim q_{x_i}(z)} \left[-\frac{1}{2} \log |\Sigma_i| - \frac{1}{2}(z-\mu_i)^T \Sigma_i^{-1}(z-\mu_i) + \frac{1}{2} z^T z \right] \\
&= -\frac{1}{2} \log |\Sigma_i| - \frac{1}{2} \mathbb{E}_{z \sim q_{x_i}(z)} \left[tr((z-\mu_i)(z-\mu_i)^T \Sigma_i^{-1}) \right] + \frac{1}{2} \mathbb{E}_{z \sim q_{x_i}(z)} \left[z^T z \right] \\
&= -\frac{1}{2} \log |\Sigma_i| - \frac{1}{2} + \frac{1}{2} \mathbb{E}_{z \sim q_{x_i}(z)} \left[(z-\mu_i+\mu_i)^T (z-\mu_i+\mu_i) \right] \\
&= -\frac{1}{2} \log |\Sigma_i| - \frac{1}{2} + \frac{1}{2} \mathbb{E}_{z \sim q_{x_i}(z)} \left[(z-\mu_i)^T (z-\mu_i) + 2(z-\mu_i)^T \mu_i + \mu_i^T \mu_i \right] \\
&= -\frac{1}{2} \log |\Sigma_i| - \frac{1}{2} + \frac{1}{2} \mathbb{E}_{z \sim q_{x_i}(z)} \left[(z-\mu_i)^T (z-\mu_i) + \mu_i^T \mu_i \right] \\
&= -\frac{1}{2} \log |\Sigma_i| - \frac{1}{2} + \frac{1}{2} \left(tr(\Sigma_i) + \mu_i^T \mu_i \right) \\
&= -\frac{1}{2} \left(1 + \log |\Sigma_i| - tr(\Sigma_i) - \mu_i^T \mu_i \right) \\
&= -\frac{1}{2} \left(1 + \sum_{k=1}^{n} \log \sigma_{i,k}^2 - \sigma_{i,k}^2 - \mu_{i,k}^2 \right) = \mathcal{L}_{KL}^i
\end{aligned} \tag{3.3}$$

As derived in Section 2.4, the regularization objective that is minimized is the Kullback-Leibler divergence $KL(q_{x_i}(z)||p(z))$. It can be written in a closed form [34] as Equation 3.3, where $\Sigma_i = diag(\sigma_i^2)$. The objective function that is minimized during training is then a sum of the reconstruction loss $\mathcal{L}_{reconstruction}$ and the Kullback-Leibler regularization term $\mathcal{L}_{KL}$ (Equation 3.4).

$$\mathcal{L}_{VAE} = \frac{1}{|\mathcal{T}|} \sum_{i=1}^{|\mathcal{T}|} \left[- \sum_{t=1}^{T} y_t^i \cdot \log(\tilde{o_t}^i) - \frac{1}{2} \left(1 + \sum_{k=1}^{n} \log \sigma_{i,k}^2 - \sigma_{i,k}^2 - \mu_{i,k}^2 \right) \right] \tag{3.4}$$

3.2.3 Beta-VAE

β-VAE [35] is an extension of the variational autoencoder architecture. It allows for more control over the amount of regularization by introducing a hyperparameter β (Equation 3.5). Because the prior distribution of latent variables $p(z)$ is a standard multivariate Gaussian, setting $\beta > 1$ will produce more disentangled representations. Disentangled representations are particularly nice because each component of the latent vector represents a singular feature of the input space and is invariant to other features.

$$\mathcal{L}_{\beta-VAE} = \mathcal{L}_{reconstruction} + \beta \mathcal{L}_{KL} \tag{3.5}$$

3.3 Multiple Instance Classification

Programs may access or create multiple files during their runtime, hence the data we obtain from the analyzer contain sets (bags) of strings (instances), where each string represents path of a file that the program accessed or created. Our approach follows the *embedding-space paradigm*, and assumes that the labels are only available on the bag level. We use a transformation function $\mathcal{A} : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^d$, where n is the number of instances in the bag, m is the size of vectors that represent instances, and d is the output size. The aggregation function $\mathcal{A}$ should be permutation invariant, differentiable, should accept bags with variable number of instances, and should output fixed-size vectors. Inputs to $\mathcal{A}$ are latent representations of strings obtained using one of the methods described in previous sections. Output of $\mathcal{A}$ is then used as features for a downstream classifier. Suitable choices of $\mathcal{A}$ are for example: mean, max, mean + max or learnable bag aggregator which is described in the next section.

3.3.1 Learnable Bag Aggregator

Learnable bag aggregator (LBA) is a differentiable, permutation invariant function that accepts an arbitrary number of input vectors and outputs a single vector $o \in \mathbb{R}^k$. LBA input is represented as a matrix $E \in \mathbb{R}^{n \times m}$, where n is the number of instances and m is the size of instance vectors. LBA has three trainable matrices: key projection matrix $W_K \in \mathbb{R}^{m \times d}$, value projection matrix $W_V \in \mathbb{R}^{m \times d}$, and a query matrix $W_Q \in \mathbb{R}^{h \times d}$, where h is the number of attention heads [12] and d is the hidden size.

$$K := EW_K \in \mathbb{R}^{n \times d} \tag{3.6}$$

$$V := EW_V \in \mathbb{R}^{n \times d} \tag{3.7}$$

$$W := W_Q K^T \in \mathbb{R}^{h \times n} \tag{3.8}$$

$$w_i = \frac{exp(w_i)}{\sum_{j=1}^{n} exp(w_j)}; i = 1...n \tag{3.9}$$

Inputs E are first projected to keys and values as described by Equation 3.6 and Equation 3.7. Multiplying keys with the query matrix W_Q (Equation 3.8) gives us a matrix of attention weights whose columns are then softmax normalized to unit sum (Equation 3.9). Finally, multiplying the normalized weights matrix W with the values matrix V produces a matrix of convex combinations of value vectors for each head. This matrix is then flattened to a vector of fixed dimension $\mathbb{R}^{hd}$ (Equation 3.10), which represents the aggregated output.

$$o := vec(WV) \in \mathbb{R}^{hd} \tag{3.10}$$

3.3.2 Classifier

We use a simple linear classifier to classify aggregated sets of vectors as benign or malicious. The classifier can be written as Equation 3.11, where $o_i \in \mathbb{R}^d$ is the bag vector and $\theta \in \mathbb{R}^{d \times 2}$, $b \in \mathbb{R}^2$ are classifier parameters.

$$\hat{y}_i = f_{\theta,b}(o_i) = softmax(\theta^T o_i + b) \tag{3.11}$$

Classification loss is then the cross-entropy between one-hot encoded ground truth labels $y_i \in \{0, 1\}$ and classifier predictions $\hat{y}_i$ (Equation 3.12).

$$\mathcal{L}_{CLF} = -\frac{1}{|\mathcal{T}|} \sum_{i=1}^{|\mathcal{T}|} y_i \cdot \log(\hat{y}_i) \tag{3.12}$$

3.4 Adversarial Sample Generation

As described in Section 2.6, the goal of adversarial sample generation is to find a perturbation $\delta \in \Delta$, where Δ is the set of allowable perturbations, such that the perturbed input $x + \delta$ is misclassified. In this work we focus on adversarial examples for character sequences, hence the set of allowable perturbations includes adding, removing or changing individual characters. Finding a minimal sequence of perturbations that changes the classification of a sample x is a combinatorial optimization problem, and can be solved exactly using algorithms for combinatorial optimization. Unfortunately, exact solution is often not feasible in practice as the size of the search space is $\approx 256^l$, where l is the length of the string. It is also difficult do design a heuristic that would reduce the search space enough to be practically feasible.

Instead of searching the space of perturbations in the input domain, we focus on perturbations of latent representations, and attempt to generate adversarial examples using the decoder part of an autoencoder. Given a trained encoder F, decoder G and a multiple instance classifier C, we generate adversarial samples in the following way. First, using the encoder F we encode all instances $S = \{s_1, ..., s_n\}$ in the bag to their respective latent representations as $E = \{e_i = F(s_i);\ \forall e_i \in \mathbb{R}^d,\ i \in \{1, ..., n\}\};\ E \in \mathbb{R}^{n\times d}$, where n is the number of instances and d is the latent representation size. Then, we use the multiple instance classifier C to obtain a perturbation of these latent representations as $\delta^* = \text{argmax}_{\delta\in\Delta}\, \ell(C(E+\delta), y);\ \delta^* \in \mathbb{R}^{n\times d}$. Finally, we generate a set of adversarial instances in the original (string) domain as $S_{adv} = \{s_i^{adv} = G(E_i + \delta_i^*);\ i \in \{1, ..., n\}\}$, where $E_i \in \mathbb{R}^d$ and $\delta_i^* \in \mathbb{R}^d$.

3.4.1 Projected Gradient Descent

Our implementation of the projected gradient descent algorithm (Algorithm 1) is a slight modification of the method described in Section 2.6.1. The modifications are due to the fact that the encoder F and the decoder G are not a perfect inverse of each other, i.e. it does not necessarily hold true that $F(G(z)) = z$ for some latent representation $z \in \mathbb{R}^d$. Because of this, projected gradient descent may find a perturbation δ^* that is adversarial in the latent representation space, i.e. $C(z + \delta^*) \neq y$, but the bag generated from the perturbed state $z + \delta$ is not adversarial, i.e. $C(F(G(z + \delta^*))) = y$. Therefore, we modify the projected gradient descent algorithm to generate an adversarial bag after each gradient step, encode this generated bag to its respective latent representation, classify the latent representation, and check if the label is different from the true label. Only misclassified bags generated from perturbed latent states are considered to be successful adversarial examples.

3.4.2 Normalized Gradient Ascent

Magnitude of the update $\delta_i = \delta_{i-1} + \alpha\nabla_\delta$ is highly dependent on the norm of the gradient ∇_δ. This is an issue because for $||\nabla_\delta||_2 \approx 0$ the algorithm makes very little progress, and for $||\nabla_\delta||_2 >> 0$ the algorithm takes a big step towards the boundary of the allowable set of perturbations Δ. To address this issue, we normalize the gradient to unit norm and use the normalized gradient to update δ. We also add $\gamma \approx 10^{-12}$ to the denominator for numerical stability. This modification of the update rule ensures that δ is updated with a constant step size, and makes it much easier to select hyper-parameters α and T.

3.4.3 Fast Gradient Sign Method

Modifications described in Section 3.4.1 are also used for the fast gradient sign method algorithm (Algorithm 2). We additionally extend the one-step algorithm described in Section 2.6.2 with an iterative search over the perturbation strength ϵ. The iterative approach is beneficial because the perturbation strength ϵ required to generate an adversarial bag is not constant, and we do not want to use an unnecessarily large ϵ which would generate adversarial bags with too many perturbations.

Algorithm 1: Projected Gradient Descent

```
Input: $paths, y, \alpha, \mathcal{P}, T$
Output: $adversarialPaths$
$state \leftarrow encode(paths)$;
$\delta_0 \leftarrow zerosLike(state)$;
$\hat{y} \leftarrow classify(state)$;
if $\hat{y} \neq y$ then
    return $paths$;
end
for $i$ = $1...T$ do
    $\nabla_\delta \leftarrow \nabla_\delta \ell(classify(state + \delta_{i-1}), y)$;
    $\delta_i \leftarrow \mathcal{P}(\delta_{i-1} + \alpha \frac{\nabla_\delta}{||\nabla_\delta||_2 + \gamma})$;
    $adversarialPaths \leftarrow decode(state + \delta_i)$;
    $adversarialState \leftarrow encode(adversarialPaths)$;
    $\hat{y} \leftarrow classify(adversarialState)$;
    if $\hat{y} \neq y$ then
        return $adversarialPaths$;
    end
end
```

Algorithm 2: Fast Gradient Sign Method

```
Input: $paths, y, \epsilon, \epsilon_{max}, \delta_\epsilon$
Output: $adversarialPaths$
$state \leftarrow encode(paths)$;
$\hat{y} \leftarrow classify(state)$;
if $\hat{y} \neq y$ then
    return $paths$;
end
$\nabla_{state} \leftarrow \nabla_{state} \ell(classify(state), y)$;
while $\epsilon \leq \epsilon_{max}$ do
    $adversarialPaths \leftarrow decode(state + \epsilon sgn(\nabla_{state}))$;
    $adversarialState \leftarrow encode(adversarialPaths)$;
    $\hat{y} \leftarrow classify(adversarialState)$;
    if $\hat{y} \neq y$ then
        return $adversarialPaths$;
    end
    $\epsilon \leftarrow \epsilon + \delta_\epsilon$;
end
```

3.5 Robust Classifier Training

To train a classifier that is robust to adversarial examples, we augment the training dataset with adversarial examples generated using methods described in the previous section (Section 3.4). Given a classifier function f_θ with parameters θ, we modify the training procedure from a simple minimization of the classification loss $\ell(f_\theta(x), y)$ to minimax optimization (Equation 3.13). The outer minimization tries to minimize the classification loss on the training set, while the inner maximization tries to maximize it by adversarially perturbing the input.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \frac{1}{|\mathcal{T}|} \sum_{i=1}^{|\mathcal{T}|} \max_{\delta \in \Delta} \ell(f_\theta(x_i + \delta), y_i) \tag{3.13}$$

The update rule for classifier's parameters θ is then defined as Equation 3.14.

$$\theta := \theta - \alpha \nabla_\theta \max_{\delta \in \Delta} \ell(f_\theta(x + \delta), y) \tag{3.14}$$

To compute gradient of the inner maximization we make use of the Danskin's Theorem [36], which states that the gradient of the *max* operator is equal to the gradient of the inner function evaluated at the maximum point (Equation 3.15).

$$\nabla_\theta \max_{\delta \in \Delta} \ell(f_\theta(x + \delta), y) = \nabla_\theta \ell(f_\theta(x + \delta^*), y) \tag{3.15}$$

However, this result only applies for the case where we can compute the maximum exactly and that maximum is unique, neither of which can be guaranteed for our case as we are only able to solve the maximization approximately. Nevertheless, the stronger the adversarial attack is, the better approximation of the true maximum we get, and thus we also get a better approximation of the gradient.

Algorithm 3: Adversarial Training

```
for $\mathcal{B} \in \mathcal{T}$ do
    $\nabla_\theta \leftarrow \mathbf{0}$;
    for $(x, y) \in \mathcal{B}$ do
        $\nabla_\theta \leftarrow \nabla_\theta + \nabla_\theta \ell(f_\theta(x), y)$;
        if $f_\theta(x) = y$ then
            $\delta^* \leftarrow \operatorname{argmax}_{\delta \in \Delta} \ell(f_\theta(x + \delta), y)$;
            $\nabla_\theta \leftarrow \nabla_\theta + \nabla_\theta \ell(f_\theta(x + \delta^*), y)$;
        end
    end
    $\theta \leftarrow \theta - \frac{\alpha}{|\mathcal{B}|} \nabla_\theta$;
end
```

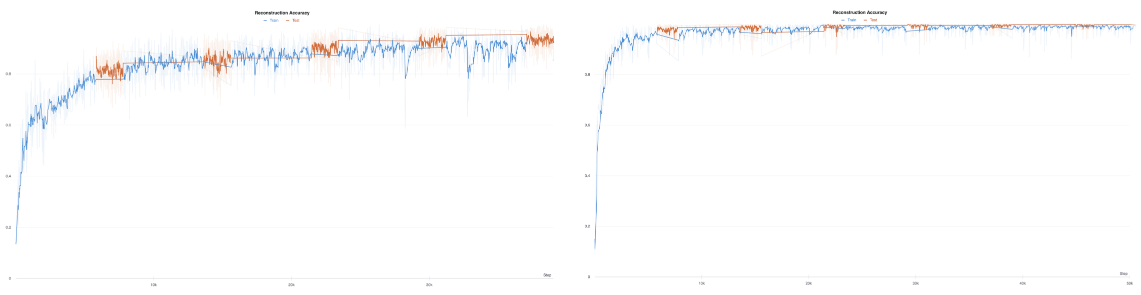
Chapter 4

Experiments

This chapter provides details of the experimental setup and results obtained using methods described in Chapter 3. We split the model training into two distinct parts. First, we train an autoencoder to represent character sequences as fixed-size vectors in an unsupervised way. Then we freeze the autoencoder's parameters and use its encoder as an instance feature extractor to a train multiple instance classifier. Given the trained autoencoder and classifier, we use the classifier's gradients to compute perturbations of latent representations and generate adversarial strings using the autoencoder's decoder. We also experiment with adversarial classifier training using both latent perturbations only, as well as character perturbations.

4.1 Representation Learning

To obtain vector representations of variable-length character sequences, we train an autoencoder to encode the input sequence to a latent vector representation, and subsequently decode this vector representation to a sequence of characters. Parameters of the autoencoder are obtained by minimizing the position-wise cross-entropy between the input sequence and the decoded sequence.



(a) Vanilla RNN (b) Convolutional RNN

Figure 4.1: Reconstruction accuracy

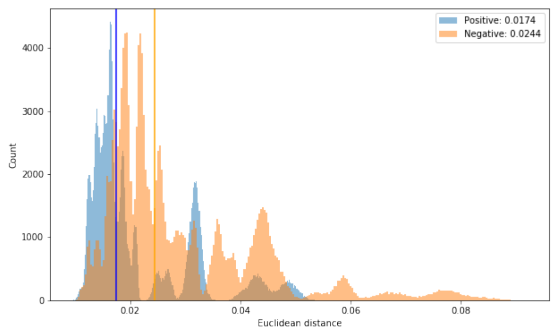
Figure 4.1a depicts a learning curve of the vanilla recurrent autoencoder. As described in Section 3.1.2, the reconstruction quality of this architecture decreases for longer sequences. Overall, it achieves **96.3%** reconstruction accuracy on the test set. By introducing convolution and deconvolution layers as described in Section 3.1.3, we reduce the length of the input to the recurrent layer, and greatly improve the reconstruction accuracy. The learning curve of the convolutional recurrent autoencoder is depicted in Figure 4.1b. Overall, it achieves **99.36%** reconstruction accuracy on the same test set.

Note that the learning curves show test reconstruction accuracy (orange) to be higher than train reconstruction accuracy (blue). This is a result of splitting the dataset into training and testing subsets based on time, rather than randomly. The reason behind the temporal split is that malware authors keep developing new versions of their malware, and thus, the temporal split is a better approximation of the data distribution at the time of deployment. Additionally, the temporal split allows us to compare our approach with existing methods developed for our dataset.

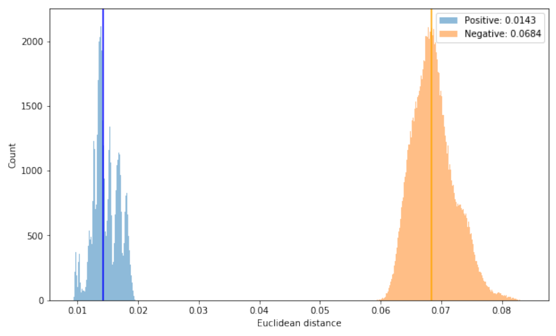
4.2 Triplet Loss

We train a regular autoencoder, and an autoencoder with the triplet loss on a synthetically generated paths dataset. Positive and negative inputs are derived from the randomly generated path using rules described in Section 3.2.1. Figures (4.2) - (4.4) show histograms of the euclidean distance between latent representations of anchor and positive inputs (blue), and between the latent representations of anchor and negative inputs (orange).

For the *Change extension 1* mutation, median of the autoencoder's anchor-positive distance is ≈ 0.0174 and median of the anchor-negative distance is ≈ 0.0244 (Figure 4.2a). For the autoencoder trained with the triplet loss (Figure 4.2b), median of the anchor-positive distance is ≈ 0.0143 and median of the anchor-negative distance is ≈ 0.0684. The Jensen-Shannon divergence between anchor-positive distance and anchor-negative distance distributions is ≈ 0.1987 for the autoencoder, and ≈ 0.6931 for the autoencoder with the triplet loss.



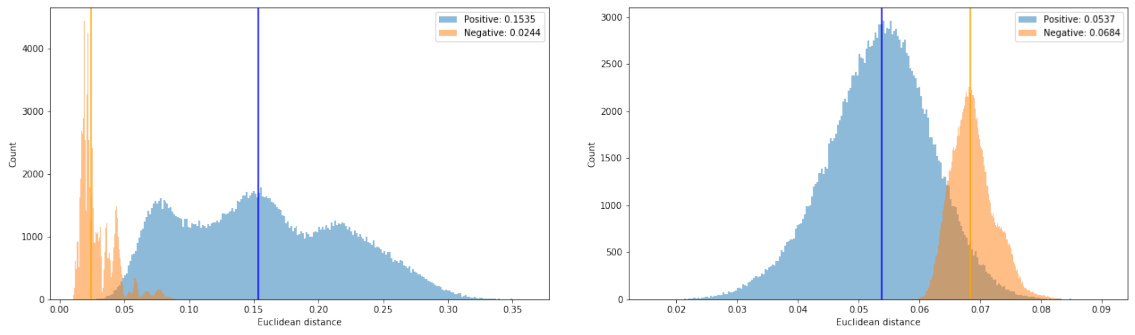
(a) Autoencoder



(b) Autoencoder + Triplet Loss

Figure 4.2: Mutation: Change extension 1

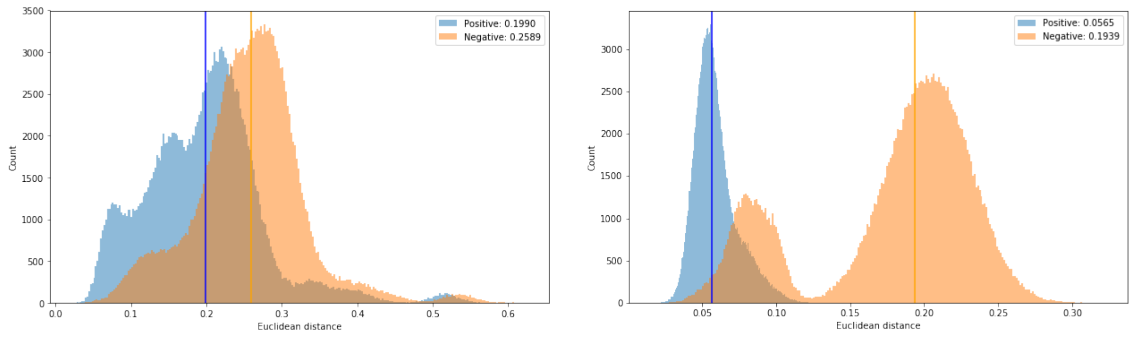
For the *Change extension 2* mutation, median of the autoencoder's anchor-positive distance is ≈ 0.1535 and median of the anchor-negative distance is ≈ 0.0244 (Figure 4.3a). For the autoencoder trained with the triplet loss (Figure 4.3b), median of the anchor-positive distance is ≈ 0.0537 and median of the anchor-negative distance is ≈ 0.0684. The Jensen-Shannon divergence between anchor-positive distance and anchor-negative distance distributions is ≈ 0.6103 for the autoencoder, and ≈ 0.4848 for the autoencoder with the triplet loss. In this case, the Jensen-Shannon divergence is greater for the regular autoencoder, however, it also encodes negative instances closer than positive instances.



(a) Autoencoder (b) Autoencoder + Triplet Loss

Figure 4.3: Mutation: Change extension 2

For the *Change path 1* mutation, median of the autoencoder's anchor-positive distance is ≈ 0.1990 and median of the anchor-negative distance is ≈ 0.2589 (Figure 4.4a). For the autoencoder trained with the triplet loss (Figure 4.4b), median of the anchor-positive distance is ≈ 0.0565 and median of the anchor-negative distance is ≈ 0.1939. The Jensen-Shannon divergence between anchor-positive distance and anchor-negative distance distributions is ≈ 0.1120 for the autoencoder, and ≈ 0.4975 for the autoencoder with the triplet loss.



(a) Autoencoder (b) Autoencoder + Triplet Loss

Figure 4.4: Mutation: Change path 1

Greater distance between latent representations implies that the autoencoder trained with the triplet loss learns to preserve an arbitrary distance metric defined in the input space.

4.3 Variational Autoencoder

Regularizing the latent representation space using the Kullback-Leibler objective does not explicitly preserve any distance measure defined in the input space. Instead, it maps inputs to a distribution over latent vectors which makes it possible to generate new examples by sampling from this distribution. Paths generated from randomly sampled latent vectors can be seen in Appendix A.

Additionally, using variational autoencoder representations improves the classifier's robustness against adversarial examples. We conjecture that this is due to the fact that variational autoencoders attempt to learn disentangled representations (Section 3.2.3), which are more strongly correlated with the label [37]. We empirically verify this claim by training multiple variational autoencoders with increasing regularization strength β and observing the adversarial attack success rate. Table 4.1 shows average standard accuracy and average attack success rate for different values of β, where the attack method used is projected gradient descent with ℓ_∞ projection. Increasing the regularization strength β has little effect on standard accuracy, but significantly reduces adversarial attack success rate.

	Accuracy	Attack Success Rate
$\beta = 0$	93.45 ± 0.04	61.28 ± 4.62
$\beta = 1$	93.22 ± 0.20	46.10 ± 1.40
$\beta = 2$	93.22 ± 0.14	44.96 ± 5.86

Table 4.1: Adversarial robustness of β variational autoencoders

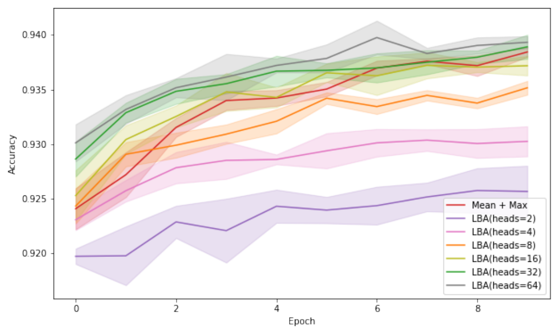
4.4 Classifier

We experiment with mean+max and learnable bag aggregator aggregation functions for multiple instance classification. Using representations from both autoencoder and variational autoencoder, we train 5 classifiers for each embedding model and aggregation method. We additionally experiment with setting the number of attention heads to $2, 4, 8, 16, 32, 64$. Average classification accuracy and standard deviation are reported in Table 4.2.

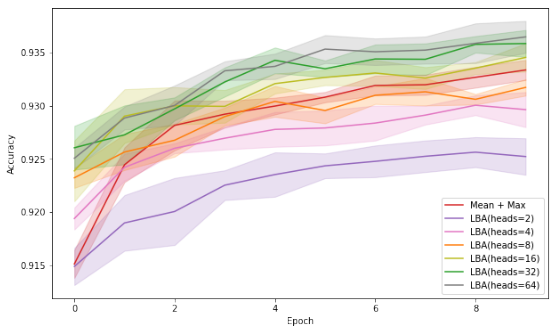
	Autoencoder		Variational Autoencoder	
	Train	Test	Train	Test
Mean + Max	91.44% ± 0.05%	93.84% ± 0.05%	90.43% ± 0.04%	93.38% ± 0.04%
LBA(heads=2)	89.45% ± 0.04%	92.65% ± 0.04%	89.11% ± 0.10%	92.65% ± 0.10%
LBA(heads=4)	90.24% ± 0.03%	93.14% ± 0.03%	89.72% ± 0.08%	93.08% ± 0.08%
LBA(heads=8)	90.94% ± 0.06%	93.52% ± 0.06%	90.30% ± 0.05%	93.23% ± 0.05%
LBA(heads=16)	91.50% ± 0.07%	93.85% ± 0.07%	90.73% ± 0.04%	93.48% ± 0.04%
LBA(heads=32)	92.04% ± 0.05%	93.92% ± 0.05%	91.13% ± 0.04%	93.65% ± 0.04%
LBA(heads=64)	92.51% ± 0.03%	94.04% ± 0.03%	91.48% ± 0.06%	93.71% ± 0.06%

Table 4.2: Comparison of bag aggregation methods

Learnable bag aggregator with 64 heads achieves the best test classification accuracy of **94.04%** using autoencoder representations, and **93.71%** using variational autoencoder representations. Test accuracy is higher than train accuracy due to the same reason as described in Section 4.1.



(a) Autoencoder



(b) Variational Autoencoder

Figure 4.5: Test accuracy curves

4.5 Adversarial Attacks

Now that we have a trained autoencoder and a trained classifier, we can generate adversarial samples using methods described in Section 3.4. Text 4.1 shows examples of successfully generated adversarial paths. Additional examples can be found in Appendix B. We also train the classifier adversarially using latent representations from the vanilla autoencoder as features (Section 3.5).

```
Input:
C:\Documents and Settings\Administrator\Application Data\Yandex\ui
Adversarial input:
C:\Documents and Settings\Administrator\Application Data\Yandex\ote.exi

Input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_uk.dll
Adversarial input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_rk.dll

Input:
C:\Program Files\GUMA36C.tmp\goopdateres_en-GB.dll
Adversarial input:
C:\Program Files\EUMA36C.tmp\goopdateres_en-d1.ddl.exe

Input:
C:\WINDOWS\Temp\7F4987FB1A6E43d69E3E94B29EB75926\seed.txt
Adversarial input:
C:\WINDOWS\Temp\7F4987FB1A6E43d69E3E6BB29EB72926\poog.e9et.ele
```

Text 4.1: Examples of adversarial perturbations

Table 4.3 shows standard classification accuracy on the test set for each of the models used in this section. Latent space adversarial training only searches for adversarial perturbations in the latent space and uses perturbed latent vectors as adversarial inputs to the classifier. String space adversarial training first generates adversarial strings from perturbed latent representations, then encodes them using the encoder part of the autoencoder, and finally uses the encoded latent representation as adversarial inputs to the classifier. The motivation behind string space adversarial training is explained in Section 3.4.1.

Model	Classification accuracy
Autoencoder	**93.49%**
Autoencoder + latent space adversarial training	91.51%
Autoencoder + string space adversarial training	91.13%
Variational autoencoder ($\beta = 1$)	93.34%

Table 4.3: Classification accuracy

Tables (4.4) - (4.7) show adversarial attack success rate and frequency of adversarial classes for all models and attack methods.

Attack Method	Success Rate	Class 0	Class 1
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	3.77%	38.20%	61.80%
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	57.35%	37.04%	62.96%
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	58.57%	39.30%	60.70%
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	**85.09%**	56.33%	43.67%
PGD(alpha: 1.00, eps: 10.00, projection: l2)	54.37%	43.45%	56.55%
PGD(alpha: 1.00, eps: 10.00, projection: linf)	70.25%	48.98%	51.02%

Table 4.4: Autoencoder

Attack Method	Success Rate	Class 0	Class 1
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	1.81%	57.73%	42.27%
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	18.97%	54.20%	45.80%
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	30.16%	64.07%	35.93%
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	**98.57%**	63.05%	36.95%
PGD(alpha: 1.00, eps: 10.00, projection: l2)	31.56%	53.50%	46.50%
PGD(alpha: 1.00, eps: 10.00, projection: linf)	51.11%	47.69%	52.31%

Table 4.5: Autoencoder + latent space adversarial training

Attack Method	Success Rate	Class 0	Class 1
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	1.13%	46.38%	53.62%
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	16.89%	54.86%	45.14%
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	13.14%	56.62%	43.38%
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	**87.34%**	59.95%	40.05%
PGD(alpha: 1.00, eps: 10.00, projection: l2)	9.43%	39.92%	60.08%
PGD(alpha: 1.00, eps: 10.00, projection: linf)	16.69%	48.74%	51.26%

Table 4.6: Autoencoder + string space adversarial training

Attack Method	Success Rate	Class 0	Class 1
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	5.46%	32.03%	67.97%
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	76.39%	51.98%	48.02%
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	30.83%	47.11%	52.89%
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	**98.19%**	62.56%	37.44%
PGD(alpha: 1.00, eps: 10.00, projection: l2)	20.00%	32.57%	67.43%
PGD(alpha: 1.00, eps: 10.00, projection: linf)	40.75%	30.10%	69.90%

Table 4.7: Variational autoencoder ($\beta = 1$)

4.5.1 Adversarial Sample Similarity

Figures (4.6) - (4.9) show how similar generated adversarial examples are to their respective input examples. The x-axis shows the maximum Levenshtein distance, and the y-axis shows the percentage of examples that have their Levenshtein distance between the original input and the adversarial input less than or equal to the threshold. Tables (4.4) - (4.7) show that *FGSM(delta: 0.10, max_eps: 10.00, sign: True)* is the most successful adversarial attack method, but it also generates adversarial inputs that are least similar to the original input.

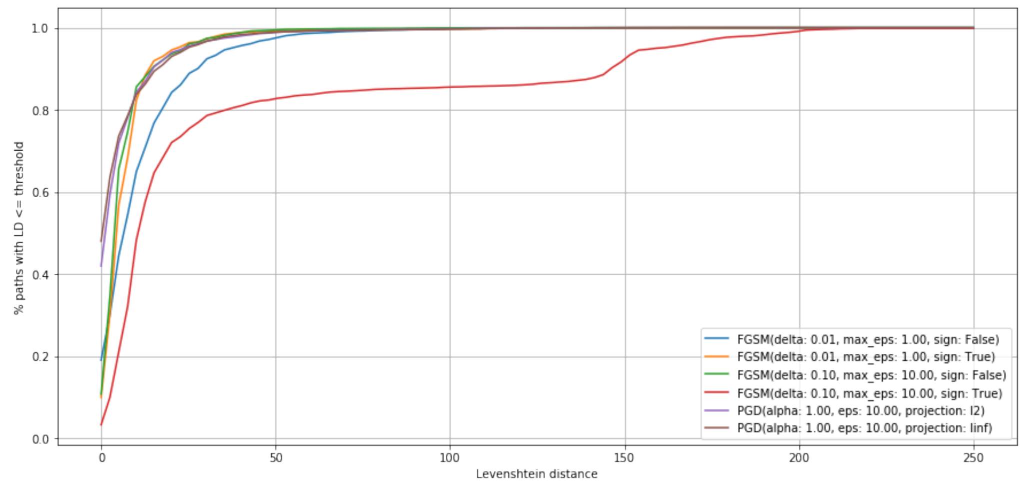


Figure 4.6: Autoencoder

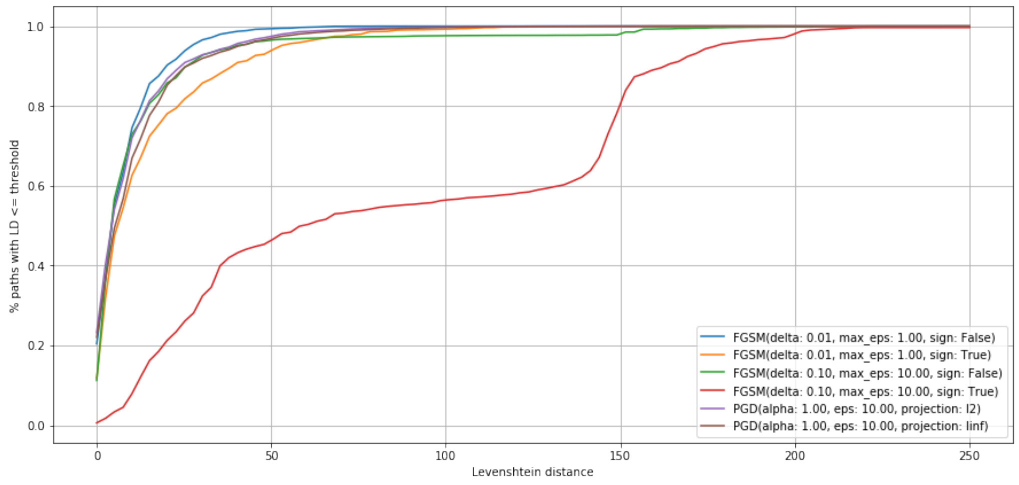


Figure 4.7: Autoencoder + latent space adversarial training

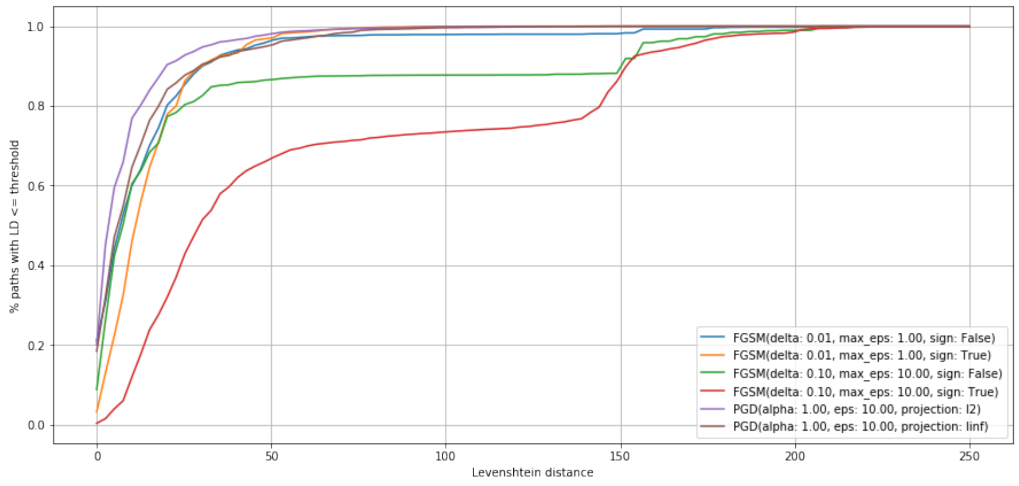


Figure 4.8: Autoencoder + string space adversarial training

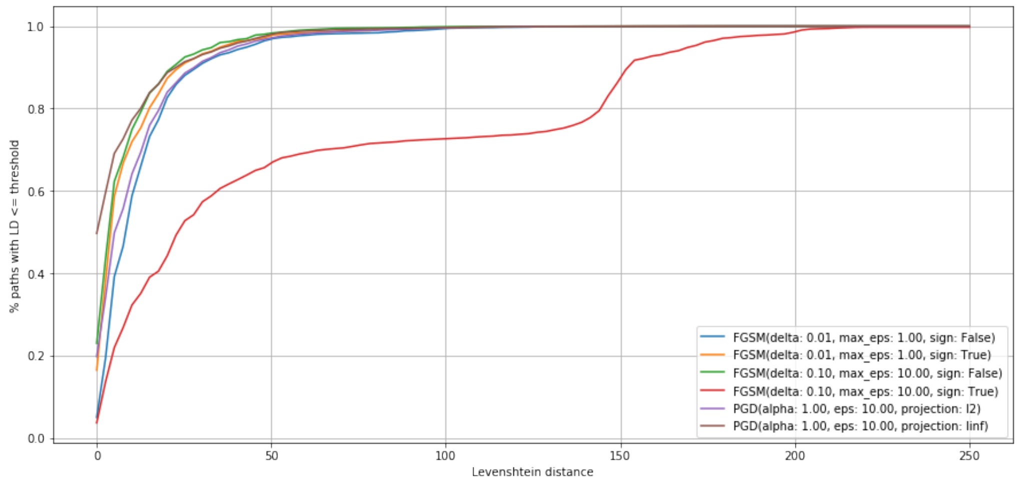


Figure 4.9: Variational autoencoder ($\beta = 1$)

4.5.2 Reconstructed Sample Similarity

Figures (4.10) - (4.13) show how similar paths generated from unperturbed latent representations are to their respective input examples. Similar to Section 4.5.1, the x-axis shows the maximum Levenshtein distance, and the y-axis shows the percentage of examples that have their Levenshtein distance between the original input and the reconstructed input less than or equal to the threshold. Additionally, we only consider examples for which the adversarial search found an adversarial input.

Note that *FGSM(delta: 0.01, max_eps: 1.00, sign: False)* is the least successful adversarial attack method overall and that its reconstruction quality is the lowest among all attack methods. This fact suggests that when this method finds an adversarial example, this example is already from a low-density region of the latent space, and thus it is easily perturbed and misclassified.

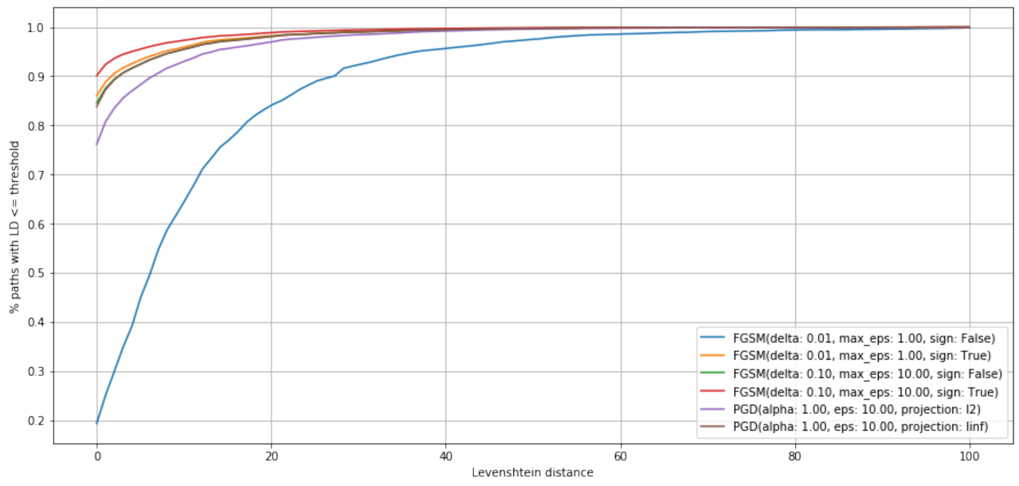


Figure 4.10: Autoencoder

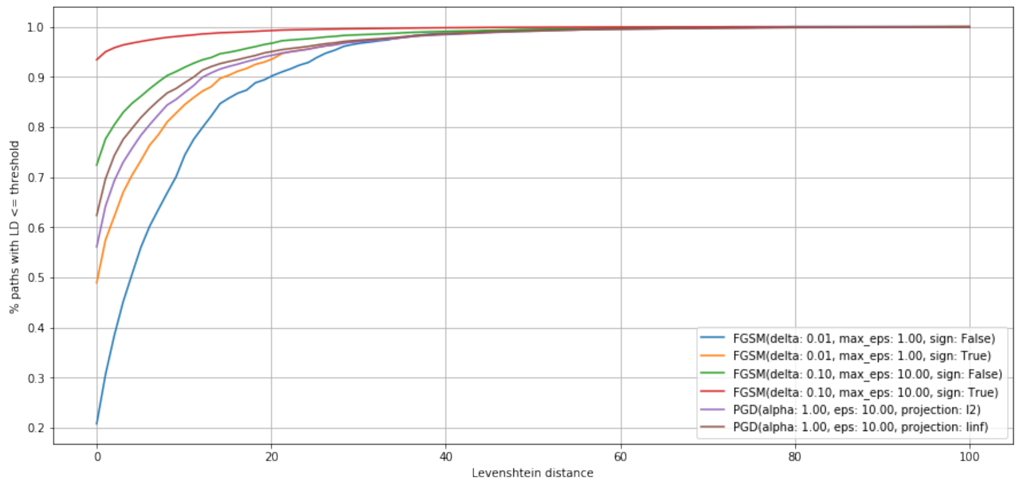


Figure 4.11: Autoencoder + latent space adversarial training

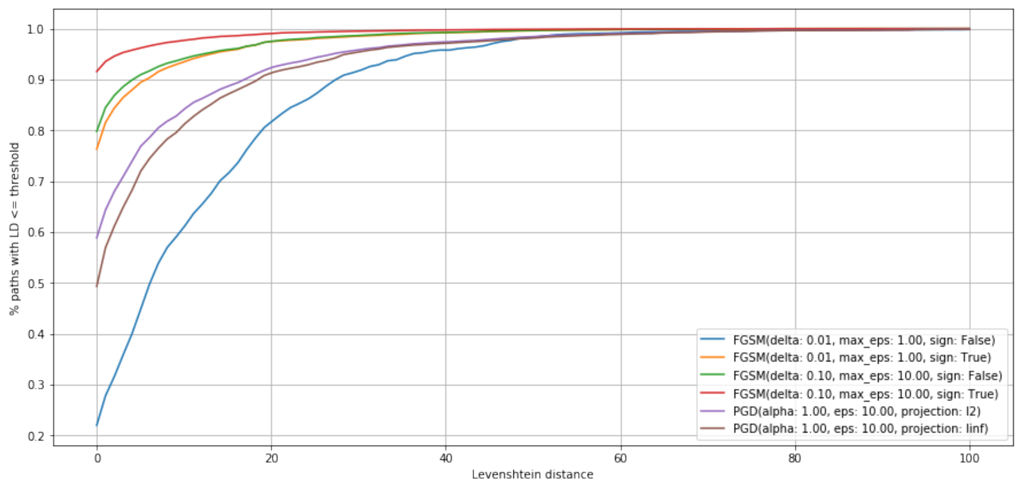


Figure 4.12: Autoencoder + string space adversarial training

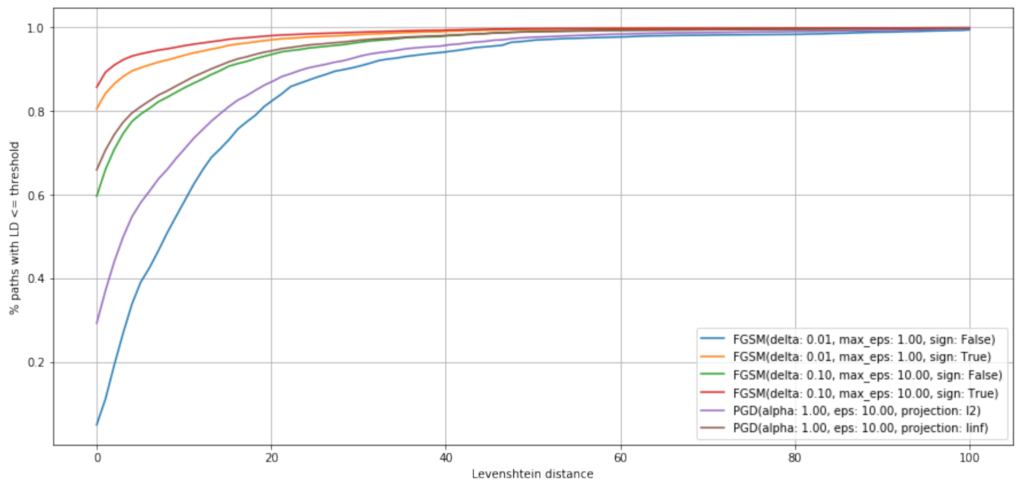


Figure 4.13: Variational autoencoder ($\beta = 1$)

4.5.3 Cross Attack

Finally, we look at how successful adversarial examples generated with one model-method combination are at attacking other models. Each of the figures (4.14) - (4.17) represents a target model, where columns are attack models and rows are attack methods. All models were trained on the same training set, and adversarial examples were generated from the same testing set.

Overall, models are most successful at attacking themselves, which is expected. A more interesting result is that adversarial inputs generated using the vanilla autoencoder seem to be the least successful at attacking other models, especially using the strongest attack method *FGSM(delta: 0.10, max_eps: 10.00, sign: True)*. This is likely caused by the fact that the vanilla autoencoder is not explicitly optimized to be robust against adversarial examples, and therefore adversarial examples generated using this model are not very strong.

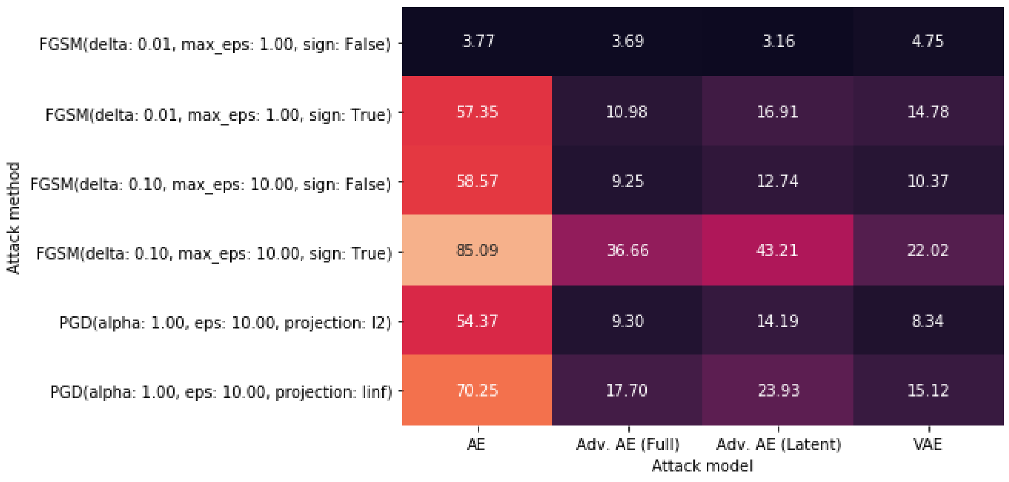


Figure 4.14: Target model: Autoencoder

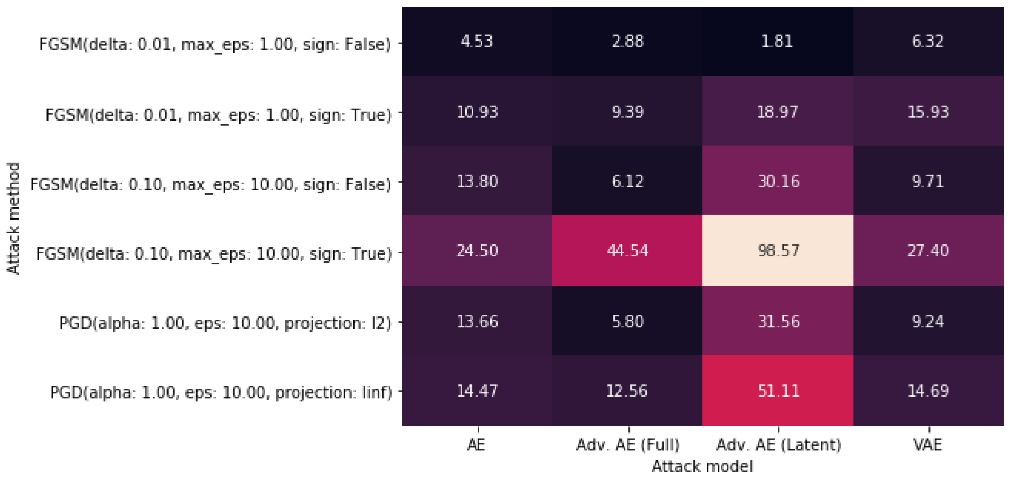


Figure 4.15: Target model: Autoencoder + latent space adversarial training

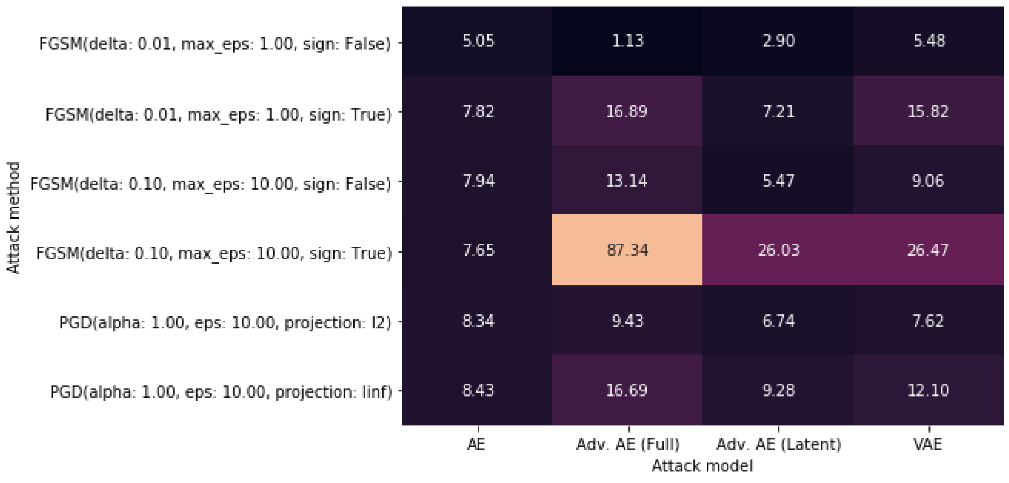


Figure 4.16: Target model: Autoencoder + string space adversarial training

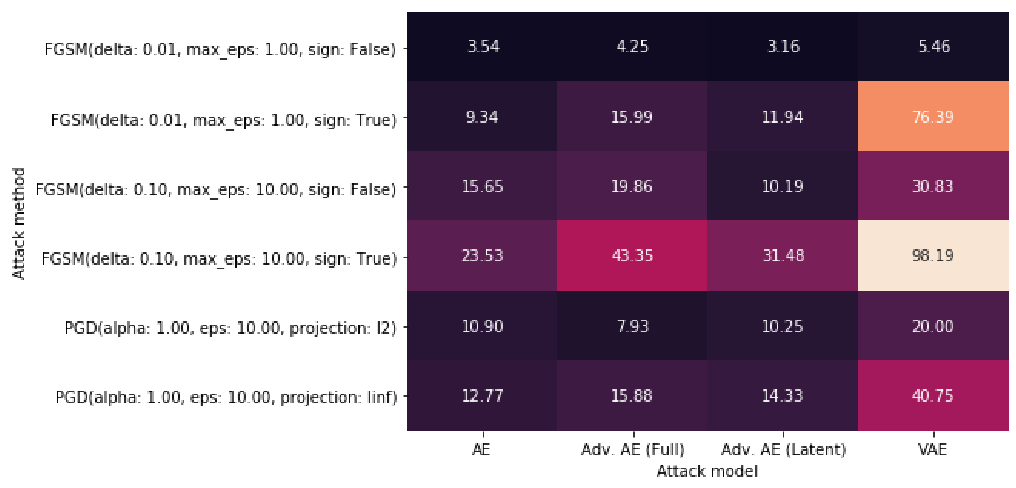


Figure 4.17: Target model: Variational autoencoder ($\beta = 1$)

Table 4.8 shows the most successful attack model for each target model (column) and each attack method (row), excluding the target model itself.

	AE	VAE
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	VAE	Adv. AE (Full)
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	Adv. AE (Latent)	Adv. AE (Full)
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	Adv. AE (Latent)	Adv. AE (Full)
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	Adv. AE (Latent)	Adv. AE (Full)
PGD(alpha: 1.00, eps: 10.00, projection: l2)	Adv. AE (Latent)	AE
PGD(alpha: 1.00, eps: 10.00, projection: linf)	Adv. AE (Latent)	Adv. AE (Full)

	Adv. AE (Latent)	Adv. AE (Full)
FGSM(delta: 0.01, max_eps: 1.00, sign: False)	VAE	VAE
FGSM(delta: 0.01, max_eps: 1.00, sign: True)	VAE	VAE
FGSM(delta: 0.10, max_eps: 10.00, sign: False)	AE	VAE
FGSM(delta: 0.10, max_eps: 10.00, sign: True)	Adv. AE (Full)	VAE
PGD(alpha: 1.00, eps: 10.00, projection: l2)	AE	AE
PGD(alpha: 1.00, eps: 10.00, projection: linf)	VAE	VAE

Table 4.8: Most successful cross attack models

Chapter 5

Summary

This work focused on three main topics: learning latent representations of strings, generating adversarial strings using perturbations in the latent space, and improving classifier's robustness by training on adversarial examples.

We used autoencoders with recurrent encoder-decoder layers to learn latent representations of strings. By introducing a convolutional layer that learns implicit n-gram representations, we significantly improved the reconstruction quality of the autoencoder, especially on long and highly irregular/random strings. Additionally, we explored methods for learning latent representations that preserve metric properties between the space of strings, and the space of latent representations.

Section 3.3.1 introduced learnable bag aggregator, an attention-based bag aggregation function which we used as a replacement for the traditionally used mean+max function. Learnable bag aggregator improves classification accuracy for representations produced by both autoencoders and variational autoencoders.

We showed that it is possible to generate realistic adversarial examples using gradient-based perturbations of latent representations. This required us to develop modified versions of projected gradient descent and fast gradient sign method algorithms which account for the fact that encoder and decoder are not a perfect inverse of each other.

Training classifiers on adversarial examples improved their robustness against adversarial attacks. We experimented with adversarial training using both latent perturbations only, as well as string perturbations. Training on string perturbations produced the most robust classifier across multiple attack methods. However, increased adversarial robustness decreased standard classification accuracy on the temporally split test set. This fact suggests that training on generated realistic adversarial examples is not sufficient to obtain a model that is robust against future real-world adversarial inputs. Additionally, we showed that training the classifier on latent representations obtained using the variational autoencoder improved its robustness without any additional adversarial training. We proposed a possible explanation for this fact and provided an empirical verification of our conjecture.

Bibliography

[1] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2014.

[2] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[3] Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples, 2018.

[4] Yicheng Wang and Mohit Bansal. Robust machine comprehension models via adversarial training, 2018.

[5] Suranjana Samanta and Sameep Mehta. Towards crafting text adversarial samples, 2017.

[6] Motoki Sato, Jun Suzuki, Hiroyuki Shindo, and Yuji Matsumoto. Interpretable adversarial perturbation in input embedding space for text, 2018.

[7] Huangzhao Zhang, Hao Zhou, Ning Miao, and Lei Li. Generating fluent adversarial examples for natural languages, 2020.

[8] Javid Ebrahimi, Daniel Lowd, and Dejing Dou. On adversarial examples for character-level neural machine translation. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 653–663, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[9] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56, 2018.

[10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[11] Alex Graves. Generating sequences with recurrent neural networks, 2014.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[14] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[15] Christopher Olah. Understanding lstm networks.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[17] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[18] Afshine Amidi and Shervine Amidi. Convolutional neural networks cheatsheet.

[19] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification, 2016.

[20] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time, 2017.

[21] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

[22] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[23] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, Apr 2017.

[24] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015.

[25] Ajay Kumar Tanwani, Pierre Sermanet, Andy Yan, Raghav Anand, Mariano Phielipp, and Ken Goldberg. Motion2vec: Semi-supervised representation learning from surgical videos, 2020.

[26] S. Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. Support vector machines for multiple-instance learning. In *NIPS*, 2002.

[27] Paul Viola, John Platt, and Cha Zhang. Multiple instance boosting for object detection. volume 18, 01 2005.

[28] Thomas Gärtner, Peter Flach, Adam Kowalczyk, and Alex Smola. Multi-instance kernels. pages 179–186, 01 2002.

[29] Krikamol Muandet, Kenji Fukumizu, Francesco Dinuzzo, and Bernhard Schölkopf. Learning from distributions via support measure machines, 2013.

[30] Yixin Chen, Jinbo Bi, and J. Z. Wang. Miles: Multiple-instance learning via embedded instance selection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1931–1947, 2006.

[31] Veronika Cheplygina, David M.J. Tax, and Marco Loog. Multiple instance learning with bag dissimilarities. *Pattern Recognition*, 48(1):264–275, Jan 2015.

[32] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models, 2015.

[33] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[34] Stephen Odaibo. Tutorial: Deriving the standard variational autoencoder (vae) loss function, 2019.

[35] I. Higgins, Loïc Matthey, A. Pal, C. Burgess, Xavier Glorot, M. Botvinick, S. Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

[36] Wikipedia contributors. Danskin's theorem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 16-March-2021].

[37] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy, 2019.

Appendix A

Generated Paths

```
C:\Program Files\GoMgleE.tmp\poopdgtetatepridmpgd.tsi
c:\arkgr.exe
C:\WINDOwS\system32\gIegyeT.exp
C:\aikguoseXy
c:\4psfe.exe
C:\ProDram\Filps\MUCA368.t1p\2oudam_da-d_l1.d.l
C:\WINdOWS\Tystem\VEXE
C:\Program\Filps\Goog74\Temp\GUMSE47.tmp\psmschin4.dll
c:\5cmaki.exe
C:\WINDOWS\Temp\is-5GMIE.tmp\Pmas\ets\\amesep.en.
C:\arogrwm feme\fsybAAB..ttmpnntorele-dittou.rube.lu-httubu.embe.tui
c:\0rthrl.tem
C:\Windows\System\yznqvaf.exe
C:\WINDOWS\Temp\ISP896~1MpmaHuse.Re
C:\WINDOWS\Temp\isi47FF6.tmp
C:\WIndows\System\WVWOJqj.exe
c:\3akomwc.exe
C:\WINDOWS\Temp\iNiO4S6P.tmp\ieiril
D:\gsndew.eSystemQ2byReteeT.exuIeckgmasee.enfe.esfe.exfe.esie.esf
C:\WINDOWS\Temp\inH17749233692ilgsherenoaheole
C:\WINDOWS\Temp\j\WEWEORQ.gxe
C:\Program Files\Google\Temp\GUMA287.tmp\goopdateres_ee.dlu
C:\WINDOWS\Temp\is-6QADM.tmb\wmpmmuct_statld.dml
c:\drkgmo.exe
C:\WINDOWS\systeG3D MoSSSreMSteoe er.exe
C:\WINDOWS\Temp\Wr__rmc16OmO\LENEE2exer.e
c:\nxfnhnf.exe
C:\Windows\System\flwITZe.exe
C:\WINDOWS\Temp\Windojs XP leIcs.exe
C:\WINDOWS\Syspem\2hZOnRj.exe
```

Appendix B

Adversarial Inputs

```
Input:
C:\Documents and Settings\Administrator\Application Data\Yandex\ui
Adversarial input:
C:\Documents and Settings\Administrator\Application Data\Yandex\ote.exi

Input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_uk.dll
Adversarial input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_rk.dll

Input:
C:\Program Files\GUMA36C.tmp\goopdateres_en-GB.dll
Adversarial input:
C:\Program Files\EUMA36C.tmp\goopdateres_en-d1.ddl.exe

Input:
C:\WINDOWS\Temp\oCFVhbs.ini
Adversarial input:
C:\WINDOWS\Temp\UCBstos.eni

Input:
C:\WINDOWS\Temp\nsb7261.tmp\UAC.dll
Adversarial input:
C:\WINDOWS\Temp\nsc6531.tmp\UAC.dll

Input:
C:\WINDOWS\Temp\BgRh53b.ini
Adversarial input:
C:\WINDOWS\Temp\1OnDoebF.bae
```

```
Input:
C:\WINDOWS\Temp\nsk6C58.tmp\UAC.dll
Adversarial input:
C:\WINDOWS\Temp\nsk6C58.tmp

Input:
C:\WINDOWS\Temp\i4j_nlog_2
Adversarial input:
C:\WINDOWS\Temp\i4j_nlogdllgodb.g.dxe

Input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_zh-TW.dll
Adversarial input:
C:\WINDOWS\Temp\GUM896.tmp\goopdateres_zh-TR.dll

Input:
C:\WINDOWS\Temp\lkpYxWW.ini
Adversarial input:
C:\WINDOWS\Temp\wLpYIzC.eni

Input:
C:\WINDOWS\Temp\nsg28B0.tmp\System.dll
Adversarial input:
C:\WINDOWS\Temp\nsg68F0.tmp\System.dll

Input:
C:\WINDOWS\Temp\7F4987FB1A6E43d69E3E94B29EB75926\seed.txt
Adversarial input:
C:\WINDOWS\Temp\7F4987FB1A6E43d69E3E6BB29EB72926\poog.e9et.ele

Input:
C:\WINDOWS\Temp\nsb9DBA.tmp\modern-header.bmp
Adversarial input:
C:\WINDOWS\Temp\nsu48FD.tmp\modern-header.bmp

Input:
C:\WINDOWS\Temp\Opera Installer\elpmas.exe
Adversarial input:
C:\WINDOWS\Temp\GiertoInstailereenomas.txe
```