F3

Faculty of Electrical Engineering
Department of computers

Master thesis

Min-max optimization methods in adversarial learning

Training robust neural network classifiers, including S4

Bc. Tomáš Kasl
Open informatics / Artificial intelligence

2024

`https://gitlab.fel.cvut.cz/kasltoma/diploma-thesis`

Supervisor: doc. Ing. Tomáš Kroupa, Ph.D.

Acknowledgement / Declaration

Firstly, I would like to thank my supervisor, doc. Tomáš Kroupa, for being patient with me. I would also like to thank my other teachers for preparing me for this personal milestone. Also, I would like to express my thanks towards Mgr. Dmytro Mishkin, Ph.D., prof. Ing. Václav Šmídl, Ph.D. and RNDr. Milan Straka, Ph.D. for providing me with a consultation. Most importantly, however, I would like to express my gratitude towards my friends, family and fellow CTU students, for they have been a great support not only along making this project but also for the whole study period.

I, Tomáš Kasl, declare that this master thesis is my work and that I have listed all sources of information used within in accordance with methodical instructions for observing ethical principles in the preparation of university theses.

Prague, May 24, 2024

Já, Tomáš Kasl, prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 24. 5. 2024

Abstrakt / Abstract

Tato diplomová práce se soustředí na zvýšení odolnosti klasifikačních neuronových sítí proti datům, která byla záměrně upravena (především gradientními metodami) za účelem obelstění klasifikátoru. Toho se snaží docílit úpravami procesu učení neuronových sítí modifikacemi, které vychází z přístupu dvouhráčové teorie her. Skrze experimenty, kromě úspěšnosti metody robustního učení, zhodnotí také její složitost na implementaci a zvýšení výpočetní náročnosti.

Klíčová slova: robustní klasifikace; robustní učení; teorie her; neuronové sítě; adversariální data; diplomová práce;

This thesis focuses on enhancing the resilience of neural network classifiers against adversarially modified data (mainly by gradient-based methods), so that they are misclassified. It tries to achieve this by employing a two-player game-theoretical approach, modifying the training process of neural networks. Through experimentation, the study evaluates the effectiveness of this approach in improving classifier resistance to adversarial samples. Additionally, it explores the difficulty of its implementation as well as an increase in the computational complexity.

Keywords: robust classification; robust training; game theory; neural networks; adversarial data; master thesis;

Contents /

Chapter 1
Introductions

1.1 Motivation

In recent years, artificial neural networks have been increasingly used for various kinds of tasks, very often pushing out alternative machine learning approaches, and in some sense nowadays basically dominating the field of AI.

While generative models are gaining momentum in the scientific race in recent months, it is the classification models, which are the best understood and are being deployed into production in various crucial day-to-day live scenarios. Some very prevalent examples known for being targeted by artificial data can be scanning license plates on highways by police cameras, when checking whether the drivers are adhering to the traffic rules[1], or unlocking a smartphone by face recognition.

Other noteworthy examples, which will be used as motivation for this thesis, are these:

The task of classification of the road signs on the highways by cameras of autonomous vehicles, so that they understand, what traffic rules applies[2].

Another one is deploying sound classifiers with microphones in the streets of my hometown to automatically notify the police department when dangerous sounds appear, such as gunshots or someone screaming [3].

Many more examples come to mind, the point is that the domain of using neural networks as classifiers is large, and growing further still.

However, as these neural networks are increasingly deployed into real-world scenarios, the question of their exploitability, and the possibility to make them misclassify (either accidentally, or purposefully with malicious intents) is also gaining importance. The latter option is the main topic of this thesis.

Even when the trained classifiers can have almost perfect accuracy, at least when it comes to naturally occurring data, we must consider the presence of attackers, which are motivated to fool it. It does not necessarily matter, whether the person's motivation is some personal gain (monetary or other), or to cause harm, or something else entirely. Any mistake made by the classifier might be costly, and it is reasonable to expect the presence of people (called *attackers*) seeking to purposefully cause those mistakes.

Imagine putting, for example, a sticker on a STOP sign on a highway, so that it is understood as some irrelevant sign by the autonomous vehicle, which could lead to

[1] https://platerecognizer.com/alpr-research/

[2] https://www.tesla.com/ownersmanual/modely/en_eu/GUID-A701F7DC-875C-4491-BC84-605A77EA152C.html

[3] https://nasregion.cz/bezpecnost-v-plzenskych-ulicich-budou-hlidat-specialni-detektory-zvuku-308092

multiple unnecessary accidents. Or altering the sound of nigthly gunshots so that is not recognized by the police's sound detection system, making their arrival late.

This thesis focuses on the topic of training the neural network classifier with these adversaries in mind, making the classifiers resistant against such attacks, and increasing the accuracy even for artificially conjured samples.

The strife for the robustness is a complex field, without an explicit solution or algorithm, and without an obvious direction of how it should be solved. Multiple various approaches have been proposed, and in this thesis, I attempt to increase the neural network-based classifier's resistance against the *adversarial* samples by modeling the situation as a two-player game.

Following mainly the paper "Solving a class of non-convex min-max games using iterative first order methods [1]", I will try to make the classification models robust against such attacks, using the game-theoretical min-max approach.

The methods outlined there could possibly be upscaled to be used also in the large generative models, as they are already targeted by data perturbations, for example artists modifying the pixels of their art imagery, such that the difference is unnoticed by a human eye, yet they make the models' learning of their art style much more difficult[4].

While there is a valid argument to be made, that every specific model is vulnerable against a different specific perturbation of the classified data, since the models of neural networks used ultimately end up quite often similar, one attack can actually fool a set of models [5].

The main questions to be explored, through several straightforward experiments, are:

Mainly, how well do they, when trained this way, actually perform, meaning how their accuracy changed towards general data, and how much they make the models resistant against the modified data. Then the difficulty of implementing such robust learning techniques and the computational demands of their implementation. Multiple architectures are tested to see, whether the effects of the robust learning are shared across the whole neural network domain, or if there are some conditions to be met.

Also, we will have a look at how well are they in symbiosis with other commonly used methods of so-called regularizations applied to the training process.

1.2 The thesis outline

Firstly, (in Chapter 2), I will describe the exact problem definition. Following this, I will outline the main ideas, as well as the motivations, behind the game theoretical min-max model of the situation and just briefly compare this approach to alternative ones.

After that (in Chapter 3) follows an enumeration of attack methods and regularization techniques commonly used to mitigate the phenomenon of overfitting.

[4] https://glaze.cs.uchicago.edu/what-is-glaze.html
[5] https://platform.openai.com/docs/models/overview

Then (in Chapter 4), I will enumerate neural architectures considered in the experiments. Also, a brief explanation of the code implementation used for the experiments, as well as an explanation of the experiments themselves.

Finally (in Chapter 5), there will be the outcomes gathered from running the experiments, what do they mean, and what that means for us.

Everything is then concluded (in Chapter 6) by a discussion, about how much is implementing this approach then worth it, considering its performance and cost.

Chapter 2
Two-player game, robust learning

2.1 The problem definition

The main goal for us, the designers of the classification models, remains the maximization of its accuracy on the data presented to it. But this time also with the presence of an adversarial agent, who generates purposefully confusing data samples, in mind. Importantly, we expect the change, that is being done to the specific samples, to be in some sense *small*, focusing on misclassification caused by perturbation of some realistic, naturally existing data. To increase clearness, in this thesis, I will refer to *natural*, *adversarial*, *real*, *train* and *test* data samples:

By *natural* samples are meant samples created the natural way, e.g. images made by taking a photograph, etc. For these examples, we expect the data samples to be generated randomly and independently from some (unknown) distribution.

By the *adversarial* samples are meant the samples generated by the attacker, seeking to fool our classifier. These samples are based on *natural* examples, which have been artificially modified, based on the knowledge of the classifier. Thus, they are not expected to be generated randomly, but they remain *near* the original distribution of the original sample.

By *real* samples are meant samples, that the classifier can be expected to face, whenever it is deployed into the real world, and so it consists of both the *natural* and the *adversarial* samples.

The *train* and *test* samples are the datasets used for training and validating the neural network model. They are possibly modified by some random change, or modified to be the *adversarial* samples, which can be understood from the context.

The task of classification of the *i.i.d.* generated *natural* data is the classical domain of training the neural networks, where the models try to learn/estimate the data classes' distributions. The training algorithm is usually based on *stochastic gradient descend* (SGD) over a differentiable loss function (for classification usually cross-entropy), derived from the theory *Maximum Likelihood Estimate* (MLE). The basics are well explained in a little textbook [2].

Since the amount of the training data is limited, and the training algorithm is iterative and stochastic, the learned estimates of the distributions usually end up being imperfect, albeit often very close to the real distributions.

Mainly, regarding the topic of this thesis, there is a tendency of (local) overfitting, where the learned parameters only *barely* classify specific training samples correctly, and the classification border (well visible in Figure 2.1.) is too close. And because of the nearby classification border, even a small change to the specific sample can make the model to misclassify.

The task of classification robustness of the *adversarial* data is not yet definitely solved, as is usually the case when it comes to the presence of some *enemy*. We would like to come up with a modification of the usual training procedure, so that the accuracy on *natural* data remains as high, but the model is robust towards the artificially created *adversarial* samples, too. There are multiple, diverse approaches being explored.

Clearly, the data labeled as *adversarial* can be generated randomly, from their respective distributions, too. They are, however, in some sense outliers, and thus not very common, and following the classical *SGD*, not really that impactful.

One natural approach to increasing the classifiers' robustness, well explained in the work of Theoretically Principled Trade-off between Robustness and Accuracy [3], is to directly mitigate the overfitting tendency. The idea is to purposefully push the borders of classification areas further from the *training* samples by forcing the ϵ-neighborhood of a train sample to be classified correspondingly. That idea follows the approaches used in other classical machine learning methods, where it usually improves the accuracy on the *real* data. Importantly, a small change to a data sample is less likely to make the model misclassify, making the *attacker*'s task harder.

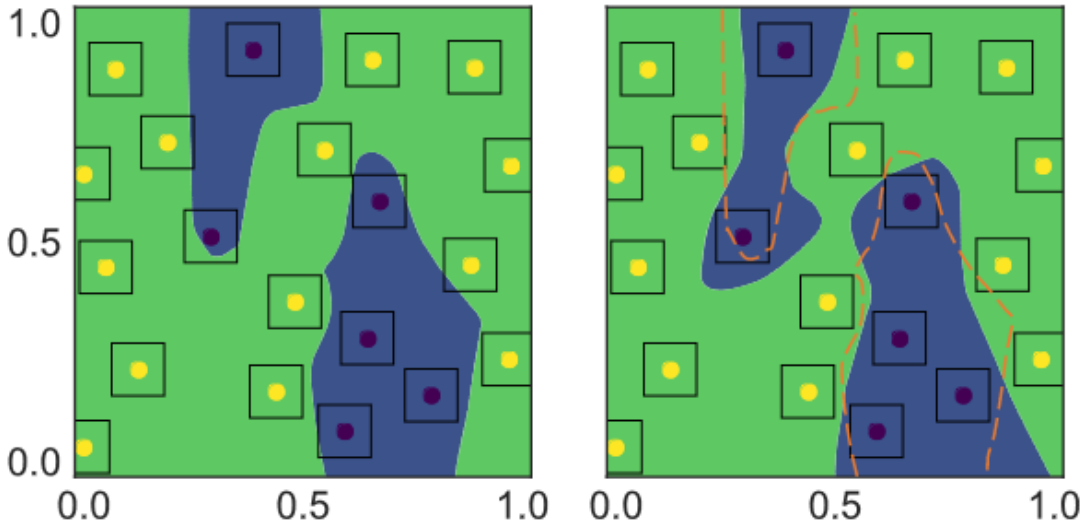


Figure 2.1. Left figure: decision boundary learned by natural training method. Right figure: decision boundary learned by their adversarial training method. Both methods achieve zero natural training error. Picture taken from a source [3].

Another approach is to model the situation in the min-max game-theoretic framework, which is being explored in this work. We expect the presence of attackers willing to purposefully exploit the models' imperfections, meaning that *adversarial* modification to the data shifts it in some *specific direction*, not just in the general area of the *training* samples. The challenge is to retain accuracy against the *natural* data high, while we want the models to be resistant against exactly against the carefully prepared *adversarial* samples.

That leads to the notion of a game of two players, the classifier and the attacker, which is explained in the following section.

2.2 Game theory

2.2.1 Notation and intuition

In this section, I try to outline a plausible motivation, why modeling the task of the robust classifier under the framework of two-player Game Theory makes sense, and following this, how can it be used to design an algorithm.

Let's have a neural network, used as some classifier, denoted as f. Given data sample x and its learned weights $\mathbf{w}$, f returns its prediction:

$$\hat{y} = f(x; \mathbf{w})$$

The classifier is *SGD* trained using a differentiable (in our case the categorical cross-entropy) loss function (comparing the predicted class $\hat{y}$ with the real class y):

$$L(\hat{y}, y)$$

By completing the training process, we want to create a classification model minimizing the value of the loss function by learning the *right* values of the models' weights $\mathbf{w}$, that is:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \; L(f(x_i; \mathbf{w}), y_i)$$

As outlined in the paper Non-convex Min-Max Optimization: Applications, Challenges, and Recent Theoretical Advances [4], we expect the presence of an attacker, whose goal is to force the model to misclassify by making an ϵ-*small* change to the data sample.

More specifically, we define ϵ ($\epsilon \in R, \; \epsilon > 0$), which limits the size of the vector δ representing the data perturbation. Given a trained model, the attack is performed, in our case, by maximizing the neural network's loss function for the specific presented data samples, that is:

$$\delta_i^* = \arg\max_{\delta_i : ||\delta_i|| \leq \epsilon} \; L(f(x_i + \delta_i; \mathbf{w}^*), y_i) = \arg\max_{\delta_i : ||\delta_i|| \leq \epsilon} \; \min_{\mathbf{w}} \; L(f(x_i + \delta_i; \mathbf{w}), y_i)$$

Notice that δ_i is a different vector for each data sample, attacks on different samples are therefore independent of each other. Also notice that it does not prescribe a specific norm $||\cdot||$, and thus any norm can be used.

An example of an *adversarial* data sample is then

$$x_i^{adversarial} = x_i + \delta_i \qquad s.t. \;\; ||\delta_i|| \leq \epsilon$$

It is well known that even for very small values of ϵ (making the perturbation *small*, sometimes possibly even humanly indistinguishable), the data modification can have a huge effect on the classification outcome.

When we expect the model to be interacted with by such an attacker, it is a natural idea to swap the *min* and *max* (thus creating a classifier maximizing the accuracy against the fraudulent data) and to define the training task as

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \max_{\delta_i : ||\delta_i|| \leq \epsilon} L(f(x_i + \delta_i; \mathbf{w}), y_i)$$

which is the standard *min-max* definition of the robust learning task.

The issue is that this optimization is generally non-convex in the minimization, and non-concave in the maximization, making it problematic to solve. This will be addressed later.

The expectation of this definition is, given a classification model f, the weights $\mathbf{w}^*$ provide the best accuracy against the *worst* type of data samples. Such a model can therefore be seen as the best possible model against the (hypothetical) most successful attacker, even though its accuracy might be non-optimal when presented with another type of data.

That is a one-sided model of the situation, with the intuition being: let there be the most successful attacker possible, and so let's train the classifier in such a way, that it has the best performance against this attacker. Let's consider the attacker's point of view, too. His motivation is to gain some advantage by making the model misclassify. He is concerned with the details of the classifier only to the extent, that lets him exploit it; he is indifferent, whether the model has been trained with his presence considered. As such, the min-max definition of the problem remains meaningful.

Consider, however, that many models of neural networks put into production are often very much alike. If a fast and reliable model for image classification is required, quite commonly, the whole problem is solved by downloading the EfficientNetV2 (presented in the paper EfficientNetV2: Smaller Models and Faster Training [5]), or some other widely used model. Similarly, all the *large language* models are based on models with self-attention transformer neurons. The attacker can therefore theoretically use one specific classification model for experimentation, and then prepare the *adversarial* samples against the whole class of existing classifiers.

This is the motivation of the zero-sum two-player game. The designer of some (broadly used) classifier *is aware* of the presence of the attacker, and the attacker *is aware* of the fact the models are trained against his struggles. He might be, therefore, motivated to create the optimal attack against the robustly-trained classifiers, too.

Under the assumption of a zero-sum two-player game, the value of min-max solution is equal to the value of max-min (that would lead to game solution in the so-called Nash equilibrium), whenever the loss function is convex in *min* and concave in *max*. But since that is not the case, here, the methods for finding such equilibrium cannot be used.

Thus, we must take another approach to solve the optimization task, employing a relaxed definition of the equilibrium, as presented in the next section.

2.2.2 Foundations and the Outline

Let's consider a general two-player zero-sum game. The players (p_1 and p_2) decide to take some actions (from their respective action spaces), which lead to some outcome of the game. For the outcome, both players are rewarded some with score, in our case penalized by the loss. Naturally, both players try to minimize their own loss received from playing the game. The whole dynamic of the two-player games is well explained in the textbook Multiagent Systems [6].

Because of the zero-sum nature of the game, the received loss of one player is always equal to the gain of his opponent. That allows us to view the two players as the minimizing player, and the maximizing player.

Let's consider the game of these two players:

- The minimizing player (the *classifier*) tries to minimize the error rate of the neural network by modifying its parameters against data prepared by the attacker.
- The maximizing player (the *attacker*) tries to maximize the error rate of the neural network classifier by modifying the data samples.

They both strife for the best strategy against the other player.

A continuous two-player zero-sum game in defined as:

$$G = (\Theta, \mathrm{A}, L)$$

where:

- Θ is the action space for p_1, $\Theta \subseteq R^m$, where m is the size of the weights $\mathbf{w}$
- A is the action space for p_2, $\mathrm{A} \subseteq R^n$, where n is the size of the data samples δ_i
- $L: \Theta \times \mathrm{A} \to R$, is the loss function giving the first player's cost for the chosen actions.

Notes:

- Θ might the same as A, but it is not a requirement
- It is enough to model only one utility function L, since $L_1(\theta_i, \alpha_j) = -L_2(\theta_i, \alpha_j)$ (where L_i is the loss function of the respective player)

Because, in the zero-sum game scenario, the players' loss functions are linked like that (minimizing own loss is the same as increasing the opponents loss), we can model the whole situation of finding the optimal strategies against each other, given the loss function L, as:

$$\min_{\theta \in \Theta} \max_{\alpha \in \mathrm{A}} L(\theta, \alpha)$$

These conditions are expected to hold for the rest of the thesis:

- Θ and A are both convex sets
- L is a continuously differentiable function

2.2.3 Nash Equilibrium

A Nash Equilibrium [7] of the zero-sum game is a pair of strategies (of the two players), such that none of the players has an incentive to change their strategies; any one player is unable to get a better outcome by just changing his own strategy. That means it is a stable pair of strategies, generally considered as the *solution* of the game.

In our setting, this corresponds to a pair of strategies, so that the classifier cannot reach better accuracy on the data (including the *adversarial*), and the attacker cannot make more effective *adversarial* samples when facing such classifiers.

The standard definition of Nash Equilibrium is a pair:

$$(\theta^*, \alpha^*) \in \Theta \times \mathrm{A}$$

ensuring the equilibrium in the players' costs, that is:

$$L(\theta^*, \alpha) \leq L(\theta^*, \alpha^*) \leq L(\theta, \alpha^*) \qquad \forall \theta \in \Theta, \forall \alpha \in \mathrm{A}$$

For cases, where the action spaces are finite sets, some Nash Equilibria always exist, and can be found by solving a linear program. In cases where $L(\theta, \alpha)$ is convex in θ and concave in α, some Nash Equilibria always exist, but finding them is not a straightforward task..

For cases, where L is neither convex (in min) nor concave (in max), as explained in [1], global Nash Equilibria might not exist at all, and even finding local Nash Equilibria is generally NP-hard.

Because the loss function of a neural network usually is not convex nor concave, a relaxed definition of the equilibrium is used, called the **First Order Nash Equilibrium (FNE)** (see Definition 2.1 in [1]), which is defined as a pair:

$$(\theta^*, \alpha^*) \in \Theta \times \mathrm{A}$$

s.t.:

$$\langle \nabla_\theta L(\theta^*, \alpha^*), \theta - \theta^* \rangle \geq 0 \qquad \forall \theta \in \Theta$$
$$\langle \nabla_\alpha L(\theta^*, \alpha^*), \alpha - \alpha^* \rangle \leq 0 \qquad \forall \alpha \in \mathrm{A}$$

The reason why these conditions are more constraining than the usual definition of critical points

$$(\theta^*, \alpha^*)\ s.t.\ \ \nabla_\theta L(\theta^*, \alpha^*) = \nabla_\alpha L(\delta^*, \alpha^*) = 0$$

is that we are searching only for critical points specifically corresponding to the min-max relation.

In the work [1] it is shown, if Θ and A are both non-empty, compact and convex, and if L is twice continuously differentiable, then there exists a pair (θ^*, α^*), which is a **FNE** of the game. However, its existence is not sufficient for its practical localization, since the second continuous derivation is often out of the scope, and thus iterative algorithms are usually used for the optimization tasks in practice.

Therefore, it is necessary to also define even less restrictive term, the ϵ-**First Order Nash Equilibrium** (ϵ-**FNE**), which is again a pair (θ^*, α^*) which satisfies:

$$\mathrm{X}(\theta^*, \alpha^*) \leq \epsilon \quad and \quad \Upsilon(\theta^*, \alpha^*) \leq \epsilon,$$

where

$$\mathrm{X}(\theta^*, \alpha^*) := -\min_{\theta} \langle \nabla_\theta L(\theta^*, \alpha^*), \theta - \theta^* \rangle \;\; s.t\; \theta \in \Theta,\; ||\theta - \theta^*|| \leq 1$$

$$\Upsilon(\theta^*, \alpha^*) := \max_{\alpha} \langle \nabla_\alpha L(\theta^*, \alpha^*), \alpha - \alpha^* \rangle \;\; s.t\; \alpha \in \mathrm{A},\; ||\alpha - \alpha^*|| \leq 1$$

This definition of (ϵ-**FNE**) guarantees that neither player can improve their outcome by more than ϵ when using first-order information (by modifying only own strategy)

The paper [1] proposes an algorithm, which is proven to find such ϵ-**FNE** for the optimization task, if the inner *max* problem is concave. It is also shown, how this can be achieved for the broader task of classification, and possibly even even for the use of neural networks.

2.2.4 Quick problem recapitulation

We want to define a min-max optimization problem (interpretable as a two-player game), where the attacker maximizes the NN's error rate over the data (achieved by intentionally modifying the data), and the NN's trainer strifes for the minimization of the error rate against such opponent:

$$\min_{\mathbf{w}} \sum_{i=1}^{N} \max_{\delta_i \,:\, ||\delta_i||_\infty \leq \epsilon} L(f(x_i + \delta_i; \mathbf{w}), y_i)$$

where:

- x_i is the specific data sample presented to the neural network
- δ_i is the perturbation of the data
- $f(x_i; \mathbf{w})$ is the prediction of the neural network given its weights $\mathbf{w}$
- $L(\hat{y_i}, y_i)$ is the neural network's loss function given the sample i, $i \in 1, ..., N$
- N is the size of the used dataset
- ϵ is the maximal change allowed for the data change

We require the inner maximization to be *concave* in δ_i.

2.2.5 Multi-Step Projected Gradient Step Solution

Following the approach defined in [1], we can transform the problem into:

$$\min_{\mathbf{w}} \sum_{i=1}^{N} \max\{L(f(\hat{x}_{i0}(\mathbf{w}); \mathbf{w}), y_i), ..., L(f(\hat{x}_{iC}(\mathbf{w}); \mathbf{w}), y_i)\}$$

where each $\hat{x}_{ij}(\mathbf{w})$ is the result of a targeted attack on sample x_i aiming at changing the output of the network to label j (C is the number of considered data classes). The

function $f(x, \mathbf{w})$ is the output of the final fully-connected layer of the neural network, where each neuron corresponds to a classification class. $L(f(x, \mathbf{w}), y)$ is then the output of the usual softmax and categorical cross-entropy loss function.

Let's denote the output of the final neuron (before the softmax) corresponding to the class j as $Z_j(x, \mathbf{w})$.

The values $\hat{x}_{ij}(\mathbf{w})$ are obtained iteratively as:

$$x_{ij}^{k+1} = Proj_{B(x,\epsilon)}\left[x_{ij}^{k} + \alpha \nabla_x \left(Z_j(x_{ij}^{k}, \mathbf{w}), -Z_{y_i}(x_{ij}^{k}, \mathbf{w})\right)\right]$$

for all of the data samples x_i in the training dataset. The initial value is naturally set as:

$$x_{ij}^{0} := \quad x_{ij}$$

However, this is still an optimization problem, which is non-convex in $\mathbf{w}$ and non-concave in x. We can approximate this finite max problem with an alternative concave problem over a probability simplex T:

$$\min_{\mathbf{w}} \sum_{i=1}^{N} \max_{\mathbf{t} \in \mathbf{T}} \sum_{j=0}^{M} \mathbf{t_j} \left(L(f(x_{ij}^{K}, \mathbf{w}), y_i)\right.$$

so that t is a probability distribution over C elements, that is:

$$\mathbf{T} = \{\mathbf{t} \in R^C \mid \mathbf{t} \geq \mathbf{0},\ ||\mathbf{t}||_{\mathbf{1}} = \mathbf{1}\}$$

which is non-convex in $\mathbf{w}$, but concave in $\mathbf{t}$. It therefore fulfills the conditions required for the robust learning algorithm, which is utilized in this thesis (see formula (70) in [1]).

Instead of coming up with a completely new *SGD*-based algorithm for training the neural network, authors of the paper propose a way how to incorporate the iterative solution of the inner concave maximization problem into the outer minimization problem, which is already being solved by the classical *SGD-backpropagation*, mainly by modifying the computation of the *loss* function to take all these possible *adversarial* samples into account apriori.

The outcome of such training process should then be a classifier with the maximal accuracy against the *adversarial* data samples (generated by gradient-based attacks).

The pseudocode of the modification of the training algorithm is following:

```
Multi-Step Projected Gradient, algorithm 1:
Extension of the categorical cross-entropy loss
```

inputs: *batch* of data x_i, y_i number ϵ, number *stepsize*, number K
output: the robust loss L_{batch}

(1) duplicate every x_i and y_i in the *batch* C-times

(2) set $x_{ij}^0 := x_{ij} \quad \forall i \in 1, ..., N, \forall j \in 1, ..., C$ in the *batch*

(3) for $k = 1, ..., K$:

(a) compute the gradient $\mathbf{G}_{batch} = \nabla_x \left(\text{mean} \left(Z_j(x_{ij}^{k-1}, \mathbf{w}), -Z_{y_i}(x_{ij}^{k-1}, \mathbf{w}) \right) \right)$

(b) compute $x_{ij}^{temp} := x_{ij}^{k-1} + \text{sign}(\mathbf{G}_{batch}) \times stepsize$

(c) compute $x_{ij}^k := \min_{element-wise} \left(\epsilon \, , \max_{element-wise} \left(-\epsilon \, , x_{ij}^{temp} - x_{ij}^{k-1} \right) \right)$

(4) compute logits $P_{ijc} \quad \forall i \in 1, ..., N, \forall j \in 1, ..., C, \forall c \in 1, ..., C$ in the *batch*

(5) compute the softmax $\sigma_{ijc} := \frac{exp(P_{ijc})}{\sum_{j=1}^{C} exp(P_{ijc})} \quad \forall i, j, c$ in the *batch*

(6) update $\mathbf{w}$ by passing $L_{batch} := -\text{mean}_i \left(\max_{jc} (\sigma_{ijc}) \right)$ to the backward pass

2.2.6 Additional notes

Firstly, notice that the approximation of the optimal *adversarial* samples is done iteratively in K steps. Is is, therefore, possible to generate increasingly more successful samples by lowering the *stepsize* and increasing the value of K, but with considerable computational cost.

While for the backward backpropagation we require one complete gradient computation (done via the differential chain-rule over all the layers of the neural network), this algorithm requires K more of them.

Also, let there be b data samples in the processed *batch*. In addition to the increased computational complexity, the algorithm must keep $b + b \cdot C$ sample copies in its memory at all times. If this lowers the size of processable batch, it will have as an effect that each epoch must be learned for higher number of smaller batches, further increasing the training time.

Also notice that the algorithm uses max-norm $|| \cdot ||_\infty$ as the the ϵ-limit for the modification of the data.

A possible issue with the presented line of reasoning and the proposed algorithm is that it puts too much focus on the *adversarial* samples, as only those are now considered for the loss computation. Is it possible the unknown data distributions have so much overlay, so that the algorithm (ignoring the unmodified data) actually considerably decreases the accuracy towards the *natural* data? I propose a modification of the algorithm, so that it considers both the worst-case *adversarial* data as well as the unchanged data.

By computing the loss function both for the worst case *adversarial* data sample and the unmodified samples, we can intuitively reduce the possible decrease of accuracy on the *natural* data, even when we possibly break the math behind making the inner optimization concave. Thus, this modification is also worth experimenting with (see, e.g. Figure 5.20.). The change is only in the step (4).

```
Multi-Step Projected Gradient 2, algorithm 2
Extension of the categorical cross-entropy loss
```

inputs: *batch* of data x_i, y_i number ϵ, number *stepsize*, number K
output: the robust loss L_{batch}

(1) duplicate every x_i and y_i in the *batch* C-times

(2) set $x_{ij}^0 := x_{ij} \quad \forall i \in 1, ..., N, \forall j \in 1, ..., C$ in the *batch*

(3) for $k = 1, ..., K$:

(a) compute the gradient $\mathbf{G}_{batch} = \nabla_x \left(\text{mean} \left(Z_j(x_{ij}^{k-1}, \mathbf{w}), -Z_{y_i}(x_{ij}^{k-1}, \mathbf{w}) \right) \right)$

(b) compute $x_{ij}^{temp} := x_{ij}^{k-1} + \text{sign}(\mathbf{G}_{batch}) \times stepsize$

(c) compute $x_{ij}^k := \min_{element-wise} \left(\epsilon \, , \max_{element-wise} (-\epsilon \, , x_{ij}^{temp} - x_{ij}^{k-1}) \right)$

(4) compute logits $P_{ijc} \quad \forall i \in 1, ..., N, \forall j \in 1, ..., C, \forall c \in 0, ..., C,$
where $c = 0$ is the set of unmodified samples, in the *batch*

(5) compute the softmax $\sigma_{ijc} := \frac{exp(P_{ijc})}{\sum_{j=1}^{C} exp(P_{ijc})} \quad \forall i, j, c$ in the *batch*

(6) update $\mathbf{w}$ by passing $L_{batch} := -\text{mean}_i \left(\max_{jc} (\sigma_{ijc}) \right)$ to the backward pass

Chapter 3
Attacks on Neural Networks, Regularizations

3.1 Attack introduction

There are multiple ways to undermine the neural network models' performance. For example, one can infect the *training* data of the learning phase, so that the model becomes predisposed to end up biased in a desired way. Such a scenario can also be modeled in the way of a two-player min-max problem, the main drawback of this approach is clear, thought. There is a requirement for direct access to the training data pool even before the model training process starts, and that is generally not a possibility.

The methods considered in this thesis, however, are the ones, which only require access to an already trained model. These attack methods then consequently learn from the models' responses in order to modify the data in malicious ways.

3.2 Approaches - gradient-based attacks

A common attack approach is to modify the data slightly before it is presented to the neural network, in the direction of its gradient (over the classifier's parameters), so that it is misclassified, either arbitrarily, or as some desired class.

We consider:

- x as the specific data presented to the neural network
- $f(x, \mathbf{w})$ as the prediction of the neural network given its weights $\mathbf{w}$
- $L(\hat{y}, y)$ as the neural network's loss function
- ϵ as the maximal change allowed for the data change
- δ is a bias in the direction of the gradient

The outcome of the gradient-based attacks is a data sample $x_{adversarial}$ generally created as:

$$x_{adversarial} = x + \delta \quad s.t. \quad |\delta_i| \leq \epsilon \quad \forall \delta_i \in \delta$$

meaning every observation (e.g. a pixel) in a data sample (e.g. in a picture) can be modified by at most $\pm\epsilon$.

We need to specify some small value of ϵ so the change of the data remains *small*, which is advantageous for multiple reasons. Firstly, it ensures the perturbation of the data does not become too large to get out of the attacker's scope, being too difficult or costly to actually perform. It also reduces the chance the data is easily identified as

adversarial by some other observer, as the *adversarial* data sample becomes too much of an outlier of the class distribution.

These attacks are known to be especially successful in laboratory conditions, however with a drawback of their own. They require the access to the specific neural network model, against which the attack is performed. This can be achieved in the scenario of preparing the *adversarial* data samples against a widely reused model, such as the pre-trained EfficientNETV2.

The interesting characteristic of such attacks is that, by limiting the change by the value of ϵ, the sample $x_{adversarial}$ is basically indistinguishable from the original x, as the difference between the two seems to be just a random noise of small values.

3.2.1 FGSM - Fast Gradient Sign Method

It is a straightforward modification of the data in the direction of the gradient. It has been proposed in the paper Explaining and Harnessing Adversarial Examples [8] in 2014 by Google, very soon after the beginning of the 2010s period of neural networks' renaissance.

The main message was that even well-trained classifiers with high confidence in the data classification can be confused into misclassification by just a tiny modification of the data sample.

Doing that, the authors presented an innate vulnerability of the neural networks used as the classifier. Besides that, they also outlined an initial countermeasure, trying to incorporate the FGSM-attacked samples into the classical SGD learning process.

The formula used for creating the modified data in this thesis, following the previously set notation is:

$$x_{adversarial} = x + \epsilon \cdot sign\Big(\nabla_x \, L(f(x, \mathbf{w}); y)\Big)$$

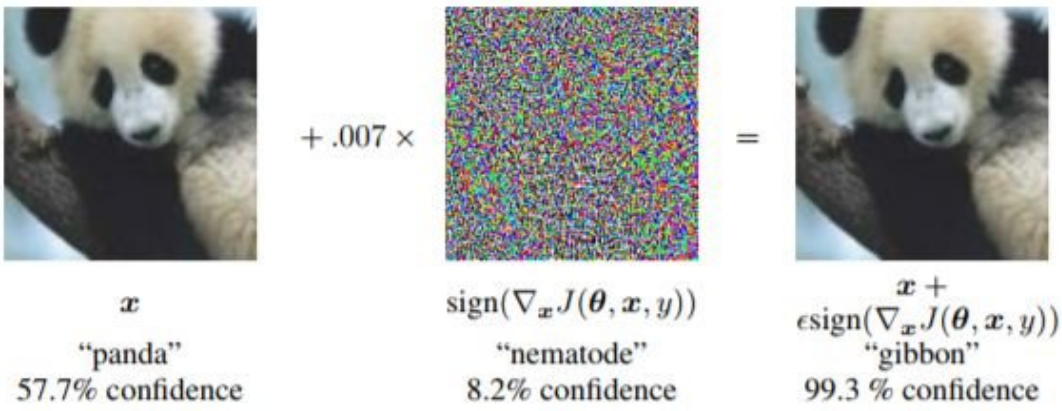


Figure 3.1. The infamous picture with pandas taken from Explaining and Harnessing Adversarial Examples [8]

As seen in the picture, even such a naive attack can have a huge impact on the classifier's performance. Notice, also, how the modified image can appear to be identical to the original one, at least to a human eye. In the case of data, which has some noise present just by its nature (like imperfections of a camera), it might be difficult to even distinguish, which samples have been modified, beforehand.

3.2.2 PGD - Projected Gradient Descent

This is a natural extension of the previous method, originally designed again by Google researches, in multiple papers [9] simultaneously (there, it is not called PGD, but 'Basic Iterative Method').

For a specified number K iterations, we change the data samples in smaller steps. The implication is that this attack method still makes the total change to the samples relatively small but with potentially even higher success of fooling the model. It also requires considerably more computational power.

The formula for the modified data is:

$$x^{k+1}_{adversarial} = Clip_{x^0,\epsilon}\Big(x^k_{adversarial} + \alpha \times sign\Big(\nabla_x L(f(x(\mathbf{w}); y)\Big)\Big)$$

for k in $0, ..., K$

where the function $Clip_{x^0,\epsilon}$ again limits the final change by the value of ϵ, meaning, for every number in the data sample, every new value is at most ϵ-different from the original value. See the step (3.c) of Algorithm 1.

Interestingly enough, the same authors (Alexey Kurakin, Ian J. Goodfellow and Samy Bengio) in the paper Adversarial Machine Learning at Scale [10] try (besides other things) to solve the task of robust learning by adding the PGD-modified samples to the training process of a neural network, which are created on the fly during the forward pass. That is really not that different idea from the one explored in this thesis.

They did not, though, find much success with such an approach. Specifically, they say: 'We also tried to use iterative adversarial examples during training, however we were unable to gain any benefits out of it. It is computationally costly and we were not able to obtain robustness to adversarial examples or to prevent the procedure from reducing the accuracy on clean examples significantly.'

3.3 Approaches - Foolbox

For the task of creating *advesarial* data samples (mainly focused on images) to achieve the misclassification, there is a framework called 'Foolbox - Fast adversarial attacks to benchmark the robustness of machine learning models [11]'. It contains code for many various already prepared attacks. Generally, these attack methods try to modify the picture in some ways, observing the model's classification decision and confidence, and then providing you with the most likely one to be misclassified.

The great thing about the framework is the implementation of an interface for plugging into the main libraries for developing the neural networks, Keras, Torch and JAX for Python. Instead of reimplementing the whole attack method, you can only point the Foolbox's API to the $classify(x)$ method of your classifier.

3.3.1 Spatial attack

This is arguably the most likely non-*adversarial*, naturally appearing reason of misclassification, as it rotates and translates the picture in different directions and angles, something you can expect to encounter when, for example classifying, the road signs. A sign can be rotated on its stand in such a way, that it confuses the trained classification model. Interestingly, it also works quite well as a targeted attack method.

The clear drawback of the Foolbox attacks is that they are aimed mainly at the image classification, and so their use is limited.

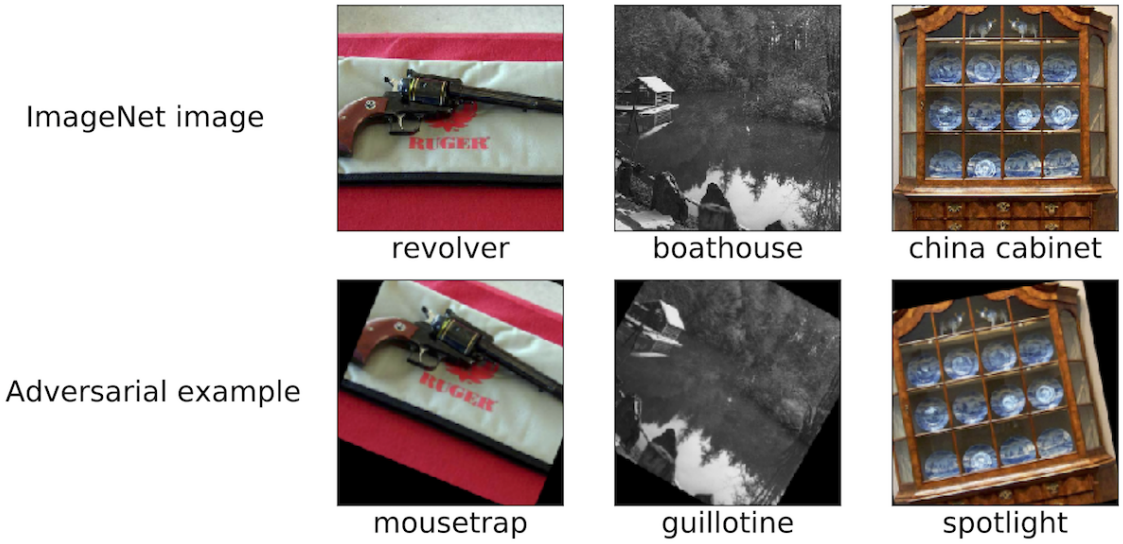


Figure 3.2. Examples of the data samples by a spatial attack, picture taken from Exploring the Landscape of Spatial Robustness [12].

While it can be argued this type of attack is also a *small* change to the original data sample, it does not comply with the condition of modifying each pixel of the data sample by at most ϵ.

This makes it more of an exercise in curiosity of how well (and if even at all) does the robust learning process (aimed against gradient-based attacks) generalizes onto completely different kinds of attacks.

3.4 Regularization

The term regularization is generally used as a large collection of varying approaches commonly used to reduce the tendency of the model's overfitting. The idea is to (hopefully) increase accuracy on *test* and later *real* data by making the neural network's training process more $difficult$.

The crucial characteristic all the methods share is that they are disabled upon finishing the training period.

Finally, the algorithm of the robust training can be seen as a form of regularization, too. It clearly is a modification of the SGD learning process, which is completely inactive after the training period is over, while it is expected to increase accuracy on *real* data.

3.4.1 Training data modification - augmentation

A common approach is to directly modify the training dataset by adding specific, for example even the *adversarially* crafted samples, into the training set. The trained model, being trained also on the inconvenient examples, should be, intuitively, much less likely to be successfully attacked.

Generally, however, the samples are (randomly) modified at the moment they are loaded in a batch to process the training step. Doing so means the random modifications to a single sample are different for each epoch without the requirement of a larger memory or dataset.

For the image modification, there are quite obvious adepts to be used. In this work, these specific random alterations (augmentations) are present under the process of training with regularization:

- Random noise added to the pixels
- Random resize (enlargement only)
- Random crop to receive the original size again
- Random rotation (up to 30 degrees)

Adding the random noise can be seen as some poor, random attack within the framework of the ϵ-gradient-based attack. The other transformations do not fulfill the condition to only modify the data by $\pm\epsilon$, yet they might be able to increase the robustness, too. They can be seen simply as a means of enlarging the training dataset, so the neural network is trained on a larger number of *unique* samples, making it potentially more successful against the spatial attacks.

There are, of course, other commonly used augmentations, but their usage is not implicitly beneficial. Take, for example, a horizontal flip. That is a common regularization method, basically doubling the size of the training dataset. Let's have an image of a horse with a head on the left side of the image, then the flipped image with the head on the right side of the image is again a valid, different image of a horse. A horizontally flipped image of the MNIST number 3, though, is not necessarily a valid data sample anymore.

For the modification of sound samples (explained later), similar things can be constructed, one must only remember that the data no longer represent 2D objects (e.g. rotating the sample does not have the same meaning anymore). In this work, these specific random alterations are present under the framework of training with regularization:

- Random noise added to the sample
- Randomly masking sections with zeros
- Resampling (decreasing the sample rate)

While these procedures generally increase the loss during the training period (as every training sample presented to the neural network has never been seen by it before), it makes the model better estimate the classes' sample distributions.

3.4.2 Dropout

It is another classic regularization method, but this one is not meant to tamper with the data samples themselves. The basic idea is that during the learning phase, every neuron has a specified shared probability p to be independently disqualified for a learning step. The motivation behind this idea is to force every single neuron into learning and providing some specific information, about the seen data, on its own. Without this, usually, multiple unique information pieces are gathered by a combination of multiple neurons, instead of these information pieces being identified independently. This process then often leads to higher accuracy on the test data [13].

A known issue is that while dropout is very powerful in feed-forward deep neural networks, in recurrent neural networks, it has to be executed carefully. Ignoring some processed data during the pass through the network, in recurrent models, not only corresponds to hiding some part of the data, but also tampering with the current inner state of the network.

It is naturally expected, for this regularization method, to be the most noticeable with the classical, non-recurrent, neural network models.

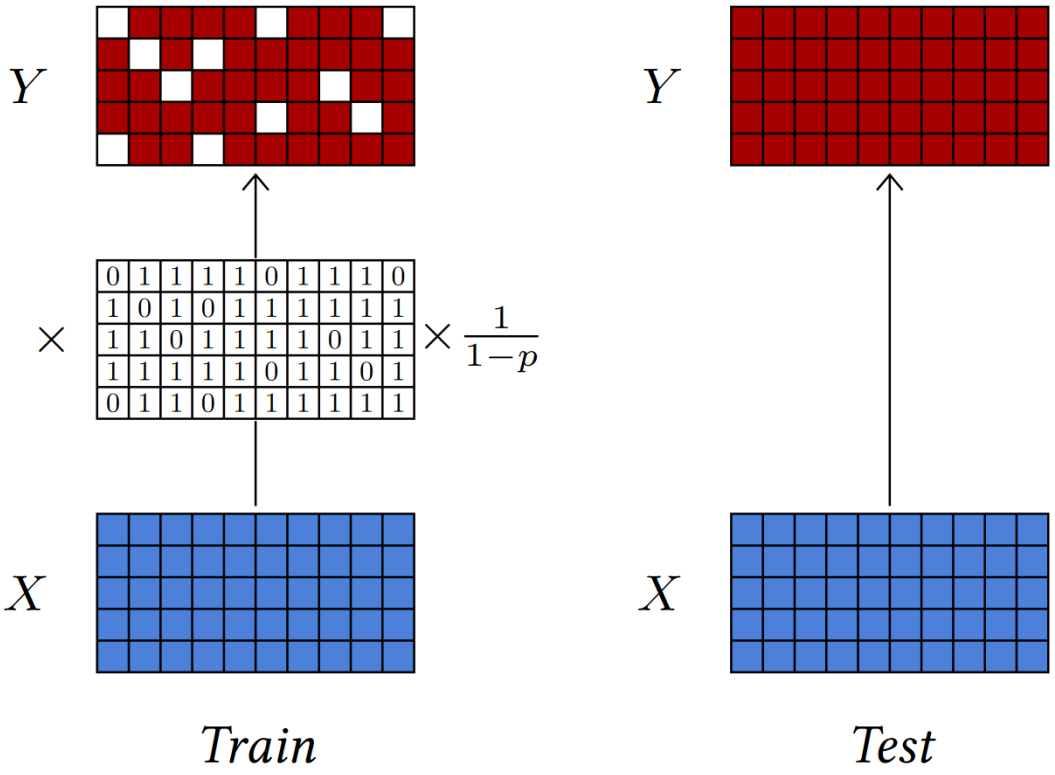
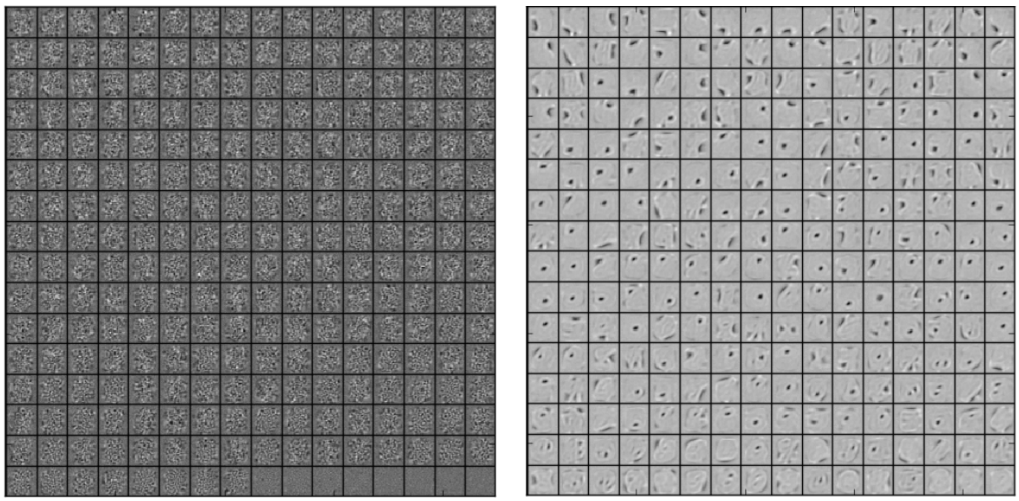


Figure 3.3. During training (left), it sets activations at random to zero with probability p and applies a multiplying factor to keep the expected values unchanged. During the test/validation phase (right), it keeps all the activations unchanged. Image taken from [2].

In Figure 3.3, there is a simple schematic, of how the dropout can be easily implemented - zeroing out values passed to/from the specific layer of the neural network. Generally, this can be done to any layer within the network, but with varying success to improve the accuracy. It is taken from the textbook for neural networks, Little Book of Deep Learning [2].

7.1 Effect on Features



(a) Without dropout

(b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

Figure 3.4. Picture taken from the paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting [13]

Figure 3.4. shows the general consequence of using dropout on the learned weights of the neurons. Instead of chaotically identifying patterns as elaborate linear combinations of neurons, each one of them is trained to search for one specific information.

The intuition, of why this should increase the robustness of the trained model, is, therefore, simple: the noise-like small, ϵ-limited perturbation to the data samples, should have much lesser impact on the well-defined neurons seen on image (b).

Chapter 4
Neural network models

4.1 Implementation, hardware

For the code implementation, I have taken and expanded the codebase provided by the work of Annotated S4[1]. For convenience and possible validation of my results, I have included the file `requirements.txt` with all libraries (and their versions) used. The Python version used is Python 3.10.12 [GCC 11.4.0].

The codebase is written in Python, leveraging multiple libraries for machine learning projects. The most crucial one is the framework JAX[2]. It is quite a recent framework for Python, developed by Google, which provides multiple great features, such as:

- Gradient (and hessian) computation of arbitrary real-valued functions
- Custom implementation of NumPy, which is a superset of the original library
- Convenient access to GPU/TPU hardware
- Precompilation of code onto the GPU/TPU hardware for computation efficiency
- Extensive language for explicitly defining&using vectorized operations

The JAX framework consists of a whole ecosystem of libraries, a noteworthy example being Flax, which provides code intended for the development of neural networks. For example, it provides the AdamW algorithm for *SGD* training and the LSTM/GRU neural cell implementation.

Because of the just-in-time (JIT) compilation onto the GPU/TPU hardware, the runtime efficiency is beyond its competition. Many empirical experiments commonly propose JAX as *faster* than Keras and Torch (both when JIT compiled and completely interpreted [3]) by a few percent. Importantly, the efficiency of the computation of gradients of real-valued functions is incomparable to the other libraries, as seen in the plot. [4]

[1] `https://srush.github.io/annotated-s4/`
[2] `https://jax.readthedocs.io/en/latest/index.html`
[3] `https://github.com/hengyuan-hu/jax-vs-pytorch`
[4] `https://towardsdatascience.com/jax-vs-pytorch-automatic-differentiation-for-xgboost-10222e1404ec`

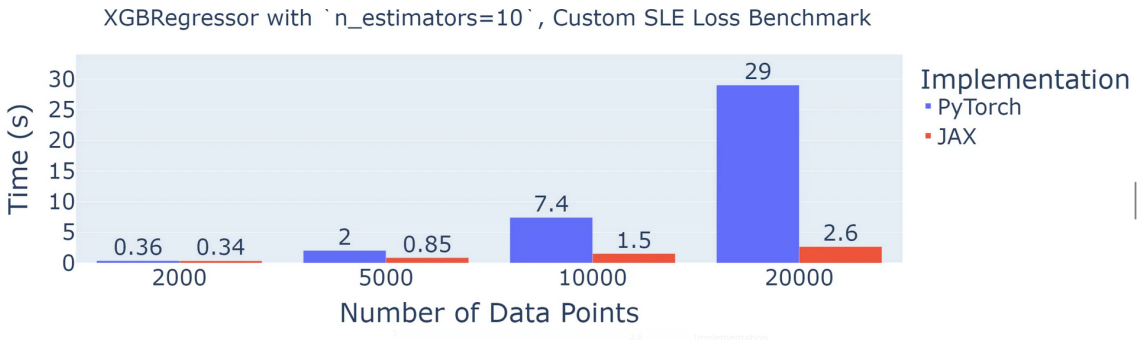


Figure 4.1. Comparison of the algorithm (including a gradient computation) run times written in JAX and in PyTorch.

That is a crucial reason to implement the experiments of this thesis in JAX instead of PyTorch, as the algorithm for robust learning is based on gradient computation. As shown later, even on (close to) current high-end hardware and JAX implementation, a single epoch of training can take hours, even for small neural networks.

Provided with the means of simple gradient computation, implementation of Algorithm 1 was not terribly difficult. It is enough to overload the implementation of the broader loss function implementation, the gradient in it is analogous to the one used for backpropagation. One must only be careful with the larger number of the tensors' axes.

The biggest drawback, however, is the fact JAX has basically no inter-compatibility with other Python frameworks for neural network development, such as the aforementioned Keras and Torch.

An important note is that, while the JAX ecosystem is multiplatform, much of the advanced functionality (like direct access to GPU/TPU) is only accessible on Linux machines. This makes the implementation codes of this thesis (at least for the time being) non-executable on computers with other operating systems.

The whole codebase was designed for use with data series processing by neural networks, like the naive SSM and S4 architectures. To perform the image classification, to simulate the dataseries processing, the images are fed into the neural networks pixel by pixel, which in reality is a somewhat uncommon and sub-optimal approach for this task, and is thus just an artificially created exercise.

In the code, this is achieved using the JAX's VMAP (vectorizing map) feature, mapping each pixel onto the neural network's input, one by one, in conjunction with the library Functools.Partial, allowing for currying (from the functional programming paradigm) the shared or static values.

The dataset loading process is done naturally, as is the standard in the field, by extending the Torch classes of `Dataset` and `DataLoader`. The transformations (both for the conveniences and regularizations) are done employing the functionality provided by `TorchVision.transformations` and `TorchAudio.transformations`.

I use the code for the S4 architecture definition basically unchanged, most of the code (such as the model training process) is greatly modified to incorporate the robust learning of S4 as well as possibly other models. Other parts of the code, such as loading different datasets, loading/saving the models to disk, plotting, and of course the attacks are a complete addition.

The *SGD* implementation of the neural network's training is quite classical AdamW with weight decay, computing the gradient over a batch of the data samples, with either the cosine learning rate schedule, starting at value 0.1, or constant one. All of the data samples have been apriori shifted into the range $< -1, 1 >$ to keep the domain small, and more importantly, shared across the datasets.

Generally, all the metaparameters (size of the neural network as well as metaparameters regarding the training process) are specified and loaded from the `config.yaml` file.

For the computation itself, both a personal computer:

- CPU: AMD Ryzen 5, 2600 (6cores/12threads)
- RAM: 32GB DDR4
- GPU: Nvidia GTX 1070, 8GB VRAM

as well as CTU's (Czech Technical University, alma mater) RCI supercomputer cluster is used. The cluster node is:

- CPU: Intel Xeon Scalable Gold 6146/6150 (4cores)
- RAM: 40GB DDR4
- GPU: Nvidia Tesla V100, 32GB VRAM or Nvidia Tesla A100, 40GB VRAM

As noted before, the robust training algorithm is quite VRAM demanding (the computation is of course being done by GPU), and so the RCI's Teslas with much more of VRAM well surpass the capabilities of any reasonable personal computer.

4.2 Basic feed-forward, dense model

A classic network built from fully connected layers, where the output layer consists of C neurons, each corresponding to the predicted likelihood of its class. Their outputs are put into softmax to select the predicted class.

This is the only non-recurrent model of neural network used. Therefore, the whole data sample is always presented to its input layer at once, not in the serial manner. This is controlled by the parameter `series` when creating the dataloader in file `data.py`. The model's presence in this thesis is mainly for validation of the outcomes and perhaps some kind of reality-check, since it is the most basic and commonly used model in this kind of experiments.

The neural network consists of the input layer, which has the input size equal to the size of the data samples, and has the output size of d. It is followed by the hidden layer, with both input and output size equal to d, and finally the output layer, with the output size equal to C. The inequality used is ReLU.

4.3 Model S4

The architecture called S4 is another step in the evolution of State Space Models (SSM). These models have large state matrices of learned parameters to keep the trace of the inner state of the models, making them good in dealing with data series. SSMs are achieving high performance in the series processing by different approach than widely used recurrent models like LSTM's or Transformers.

As is the case with other architectures for dataseries processing, the SSM's can be used also for data generation tasks, and not only for prediction or classification tasks. That is, however, outside the scope of this thesis.

Main drawbacks of SSMs are the necessity of learning the huge amount of parameters in their state matrices, and especially the numerical instability. A basic implementation of SSM is by far outclassed by a basic implementation of the LSTM architecture.

These are the motivation of the S4 architecture, which strifes to replace the huge amount of values in the state matrices by learning functions generating the numbers expected to be in the matrices instead.

By solving the clear issues of SSMs, the S4 (and its similar architectures) model is an interesting competitor to the *attention*-module based Transformer architectures, which are currently dominating the market (being the backbone of, for example, the famous GPTs).

4.4 Recurrent - LSTM

The LSTM (Long short-term memory) architecture was the first advanced successful model of recurrent neural networks, being in development since the mid 1990's. A neuron of the LSTM model has an input, output and memory gate. Until the end of the decade, it has been successfully improved by, for example, being extended by the *forget gate*, and computational optimizations; overall, it holds 8 matrices of learned parameters.

Since then, it has been the general go-to architecture for processing data series, while it witnessed barely any additional improvements.

In this work, the model is given a time-record one by one from the data sample, recurrently updating its inner state, and finally sent to the output (fully connected) layer after the last time tick, on which the softmax is computed. No complex architecture, such as a bothdirectional implementation, is present, as optimizing an architecture towards a specific task is not the focus of the thesis.

4.5 Recurrent - GRU

The Gated Recurrent Unit (GRU) has been developed much later as a computationally more efficient alternative to the LSTM. The inner state of the cell is represented only using one tensor (instead of 2 in LSTM), and retains only 6 matrices of learned weights.

The strange relation between the GRU and the LSTM architecture is that they empirically perform, on average, the same. Sometimes, on specific use-cases, however, one or the other excells and far surpasses the other one. The decision regarding which one should be used for a specific task is usually decided by trying both and selecting the one, which has better performance.

In this thesis, the high-level GRU model is the same as the LSTM's.

4.6 Datasets - overview

- MNIST numbers - dataset of grayscale images of (handwritten) digits has become the most used dataset for experiments connected to training neural networks. It consists of simple data (only one color channel). Importantly, it has been used in both the paper presenting the robust learning algorithm, as well as the paper presenting the S4 neural network architecture, so it is natural for it to be present in this thesis as well, for the sake of consistency, even though the image classification is not a natural fit for dataseries-oriented neural networks.
  - Presented by The MNIST Database of Handwritten Digit Images for Machine Learning Research [14]
- GTSRB (German Traffic Sign Recognition Benchmark) - dataset is a collection of pictures of road signs (of 43 types/classes) photographed under various conditions, distances and angles. The main difference between the GTSRB and the MNIST dataset is the GTSRB images are RGB (meaning 3 channels per pixel instead of 1). It is used only for a quick experiment of how well the dataseries models, attack methods and the robust training process generalize from single-channel to tripple-channel data (as shown leter - poorly), and how is the accuracy affected for *training* data under the robust training.
  - GTSRB (German Traffic Sign Recognition Benchmark) [5]
- Speech Commands - dataset of voice recordings (samples of one-second audio) of 35 commands for controlling e.g. a vehicle. Data series of this type clearly suits the S4's intended purpose better. Being a standard Torch dataset, it allows for simple data transformations, such as convolution or resampling, or the transformations used for the regularization
  - Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition [15]
- UrbanSound8K - dataset of sound recording common to hear in a downtown (10 classes, e.g. "a drill"), up to 4 seconds long. While the nature of the data is the same as with the Speech commands, there are few key differences. Firtly, these are not speech recording, but sounds with high periodicity instead. They are also recorded with higher numnber of different devices and under varying conditions. Lastly, it is not a Torch dataset, meaning it requires another means of data processing before it can be put into the neural network.
  - UrbanSound8K - Urban Sound Datasets [6]

[5] `https://benchmark.ini.rub.de/`
[6] `https://urbansounddataset.weebly.com/urbansound8k.html`

Chapter 5
Experiments and outcomes

5.1 MNIST numbers

5.1.1 Outline of the experiments

The first dataset used is the classic MNIST dataset of rasterized hand-drawn digits. In this thesis, this dataset consists of data samples of the shape [784,1], meaning 784 (28*28) pixels of a single color channel, only 784 values per one image. The individual values are not binary {black, white}, while it may seem like that from the ilustration, but are spanned over the whole greyscale interval.

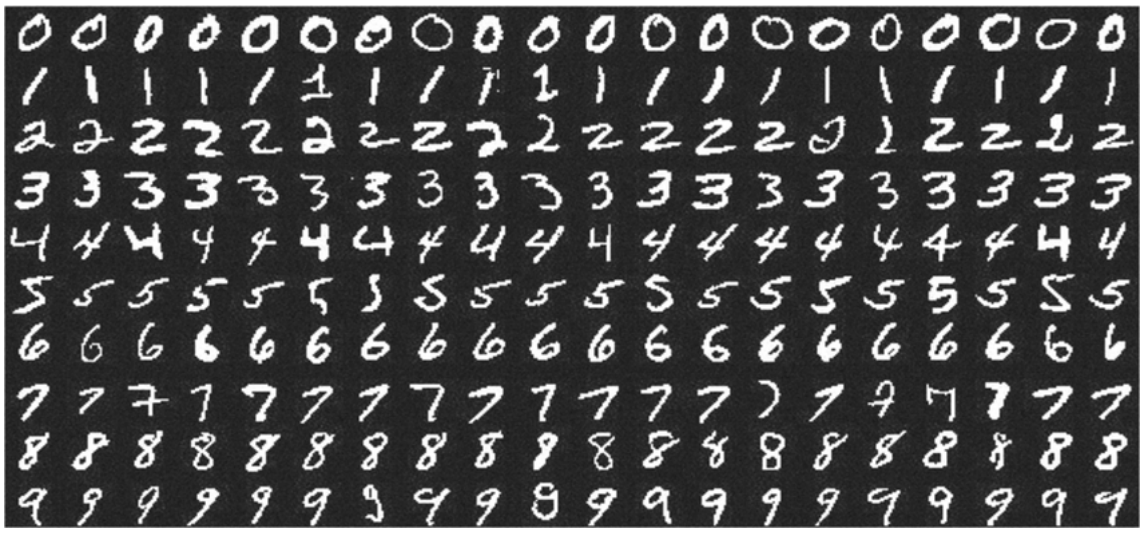


Figure 5.1. An ilustrative subset of the MNIST dataset, picture taken from a paper [14]

Firstly, let's have a look on examples of the *adversarial* data samples, which are created using the attack methods presented above, created by my implementation. In each of the following figures, there are always 3 images, the original one, the adversarially modified one, and their difference (which has been somewhat pronounced for a better visibility).

The gradient-based attack methods were constraind by $\epsilon := 0.1$, for the PGD attack, the number of iterations is $K := 40$ and $stepsize := 0.01$.

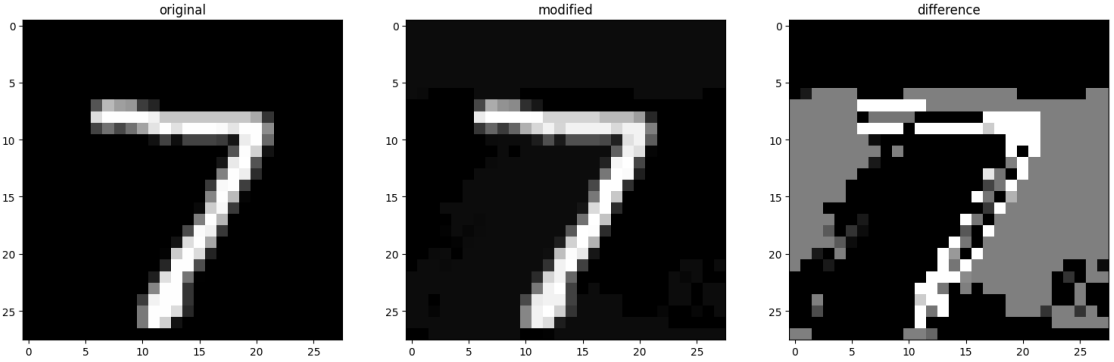


Figure 5.2. The example of PGD attack on MNIST

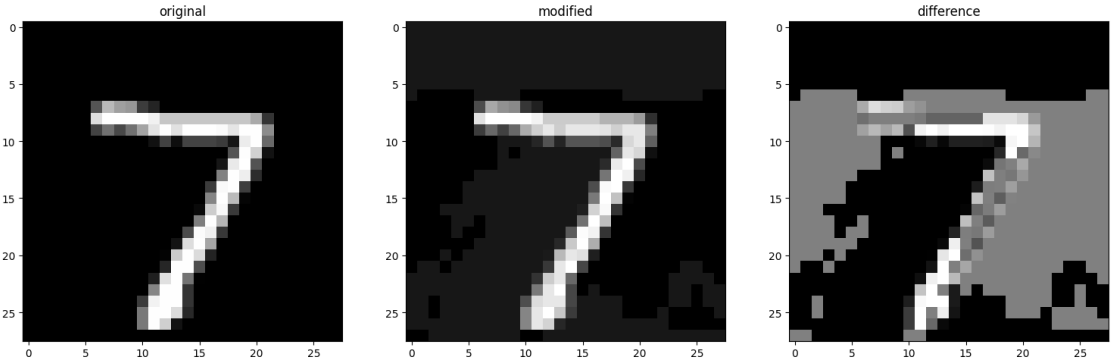


Figure 5.3. The example of FGSM attack on MNIST

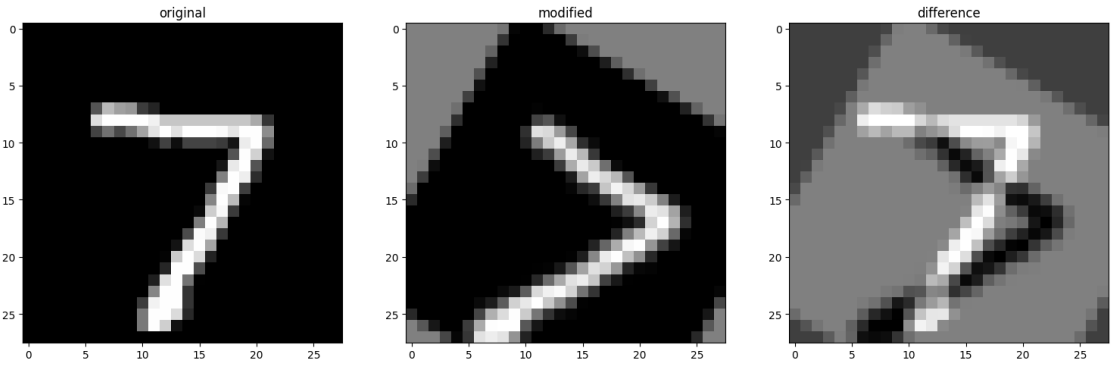


Figure 5.4. The example of spatial attack on MNIST

All three of the samples were succesfull in fooling the non-robustly trained S4 architecture, and the samples were misclassified. Notice how the FGSM and PGD attack methods produce quite similar attack vectors δ.

For the experiment, all four mentioned architectures of neural networks are trained on the dataset in four scenarios:

- Basic, non-robust AdamW training
- Robust training, using Algorithm 1
- Basic, non-robust with regularizations enabled
- Robust (Algorithm 1) training with regularizations enabled

After finishing the training process, we are interested in:

- Accuracy on the *natural* data samples
- Accuracy on the *adversarial* data samples
- Symbiosis between the robust training and the regularizations
- Influence of these on reached values of loss function on *test* data
- Increase in the computational cost of the robust training

For all of the 4 architectures and 4 scenarios, the parameters used in the experiment are in the config file:

- Size of each layer (number of neurons per layer): 128
- Size of the recurrent hidden state: 64
- Batch size: 20
- Epochs: 10
- Learning rate: 0.001
- Weight decay: 0.001
- Iterations of the robust learning algorithm: 8
- Stepsize of the robust learning algorithm: 0.15

With these metaparameters, the models have these numbers of learnable weights (that is, the minimization domain of the player - $classifier$:

- Feed-forward - 151562 parameters
- GRU - 133642 parameters
- LSTM - 166410 parameters
- S4 - 100618 parameters

The player - *attacker* is maximizing over creating the attack vector of size 784 per data sample.

As stated previously, the S4 architecture, as well as the recurrent ones, is meant to be used for time-series data processing, not for image classification. This task is therefore somewhat artificially defined, and it is achieved by feeding the neural models images pixel-by-pixel.

The exception to this is the plain feed-forward dense (FF) neural network, which is used for verification of the observed outcomes, as its implementation is simple and its learning process fast. This FF model receives the images as a 1D flattened array of pixels in one go.

These chosen parameters represent the upper limit of what can be computed on a personal computer. I use the CUDA computing cores (accessed by the JAX library) on my GTX 1070 with 8GB of VRAM. As seen on the following screenshot, the robust learning process of the S4 architecture, with these parameters, barely fits on the hardware, as seen on the following figure 5.5.

```
grifon@grifon-MS-7C02: $ nvidia-smi
Sat Apr 20 17:47:18 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.14              Driver Version: 550.54.14      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce GTX 1070        Off |   00000000:1C:00.0  On |                  N/A |
| 54%   82C    P2             128W /  151W |    8045MiB /   8192MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      1726      G   /usr/lib/xorg/Xorg                            149MiB |
|    0   N/A  N/A      1901      G   /usr/bin/gnome-shell                           32MiB |
|    0   N/A  N/A      3210      G   gnome-control-center                           12MiB |
|    0   N/A  N/A     46084      G   ...erProcess --variations-seed-version        127MiB |
|    0   N/A  N/A    158342      G   /opt/teamviewer/tv_bin/TeamViewer              12MiB |
|    0   N/A  N/A    162031      C   /bin/python3                                 6268MiB |
|    0   N/A  N/A    184476      C   python3                                      1434MiB |
+-----------------------------------------------------------------------------------------+
```

Figure 5.5. Graphics card utilization

5.1.2 Experiment outcomes

The experiments have been run multiple times, and the outcomes generally stay very similar. These are taken from a specific experiment run. Since the MNIST dataset is so simple, it is well handled by Algorithm 1.

Firstly, let's observe the table of reached accuracies:

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3	Spatial
FF	95.6	71.3	3.9	44.5	27.0	95.5
FF robust	96.5	90.7	54.5	86.4	27.0	96.2
FF regul.	86.6	25.2	0.0	6.6	0.1	82.4
FF robust+reg.	90.9	64.2	22.8	50.9	5.6	8.4
GRU	95.7	36.8	0.1	63.2	8.7	0.0
GRU robust	95.6	94.4	92.0	94.8	93.7	0.0
GRU regul.	94.8	46.0	0.4	53.6	5.1	0.0
GRU rob.+reg.	94.5	91.8	88.3	92.6	90.1	0.0
LSTM	91.7	19.1	0.9	23.4	7.7	0.0
LSTM robust	93.9	91.0	86.6	92.4	90.5	0.0
LSTM regul.	83.4	35.5	15.9	40.2	25.6	0.0
LSTM rob.+reg.	95.6	94.0	90.7	94.6	93.4	0.0
S4	98.2	93.3	34.5	90.4	36.2	5.0
S4 robust	97.9	96.5	91.4	96.4	92.2	2.8
S4 regul.	96.1	87.5	24.6	80.0	18.8	0.0
S4 robust+reg.	97.0	95.2	89.5	95.2	90.3	0.2

Table 5.1. Accuracy of the different scenarios reached for the MNIST dataset, Algorithm 1

Several things are directly clear. The basic feed-forward dense model is clearly more resistant against the spatial attacks, probably because the spatial patterns remain similar after the attack. Since the shift in pixels corresponds into large time-difference observed by the other models, they are affected more. The regularization methods do not appear to increase the resistance, though. A possible explanation is that random resize and crop may create difficult samples. Take for example cutting-out the upper part of the number 7, resulting in a sample basically identical to the number 1

Because of that, I have decided to re-run the experiment with smaller set of regularizations, employing only the random noise, rotation and dropout. This seems to produce much more convenient numbers. The outcomes are:

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3	Spatial
FF regul.	86.6	25.2	0.0	6.6	0.1	82.4
FF robust+reg.	90.9	64.2	22.8	50.9	5.6	8.4
GRU regul.	94.8	46.0	0.4	53.6	5.1	0.0
GRU rob.+reg.	94.5	91.8	88.3	92.6	90.1	0.0
LSTM regul.	83.4	35.5	15.9	40.2	25.6	0.0
LSTM rob.+reg.	95.6	94.0	90.7	94.6	93.4	0.0
S4 regul.	96.1	87.5	24.6	80.0	18.8	0.0
S4 robust+reg.	97.0	95.2	89.5	95.2	90.3	0.2

Table 5.2. Accuracy of the different scenarios reached for MNIST dataset, reduced set of regularizations used, Algorithm 1

Notice that the S4 model appears to reach higher accuracies than the other models, suggesting it is a good alternative for the dataseries processing. The LSTM model has generally been somewhat outclassed by the GRU model on this dataset, which is also computationally less demanding.

Because the relation of between the robust learning and the other regularizations is not clear, I only put the plots of accuracies without the regularizations, here. The rest of them can be seen in the Appendix A.

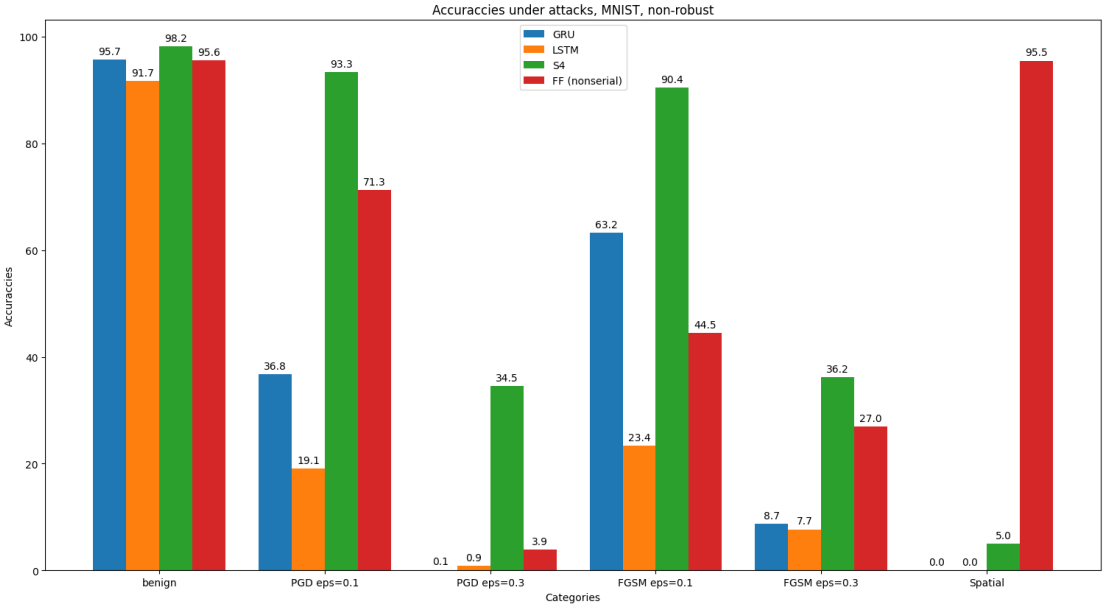


Figure 5.6. Accuracies reached by the non-robust training on MNIST

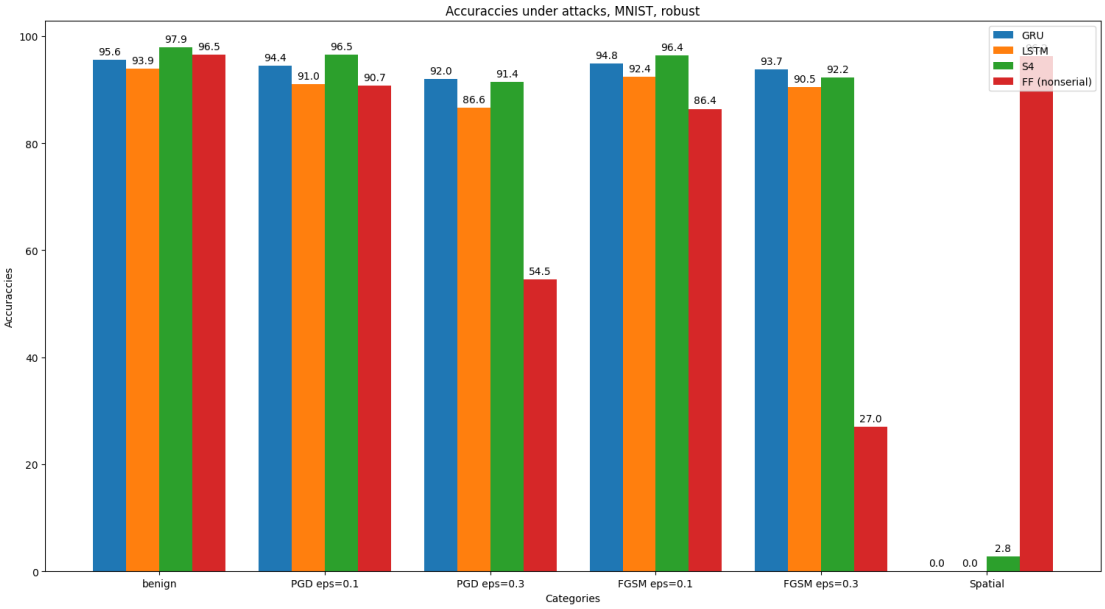


Figure 5.7. Accuracies reached by non-robust training on MNIST

The robust training method proved advantageous, generally even slightly increasing the accuracy on the *test* dataset. It seems, however, it provides no advantage against the spatial attack. With the exception the feed-forward model, the models retained accuracy close to 90% even with the PGD attack with the sample change of $\epsilon := 0.3$.

Even though the resistance against the gradient-based attacks is not absolute, it is quite high in this example, and the algorithm is thus worth consideration when training a model into real-world usecase.

The increase in computation demand is, however, a clear drawback. While one epoch of the basic, non-robust learning takes around 35 seconds for the feed-forward model, one epoch of the robust learning takes roughly 1300 seconds (a bit over 20 minutes), making the even the short 10epoch-long experiment take several hours. For the S4, the time has increased from 75 to 1400 seconds, for GRU, it changed from 260 to 2300 seconds, and for LSTM, it went from 220 to 2100 seconds. The time required by the training process is roughly 65times longer.

Following are the plots of the accuracy progress during the epochs of the training phase. This has been measured on a model with lower number of neurons for better ilustration, but similar, less pronounced fenomena can be seen, too. All of the actual experiments' outputs and data can be viewed on inside the `outcomes` folder.

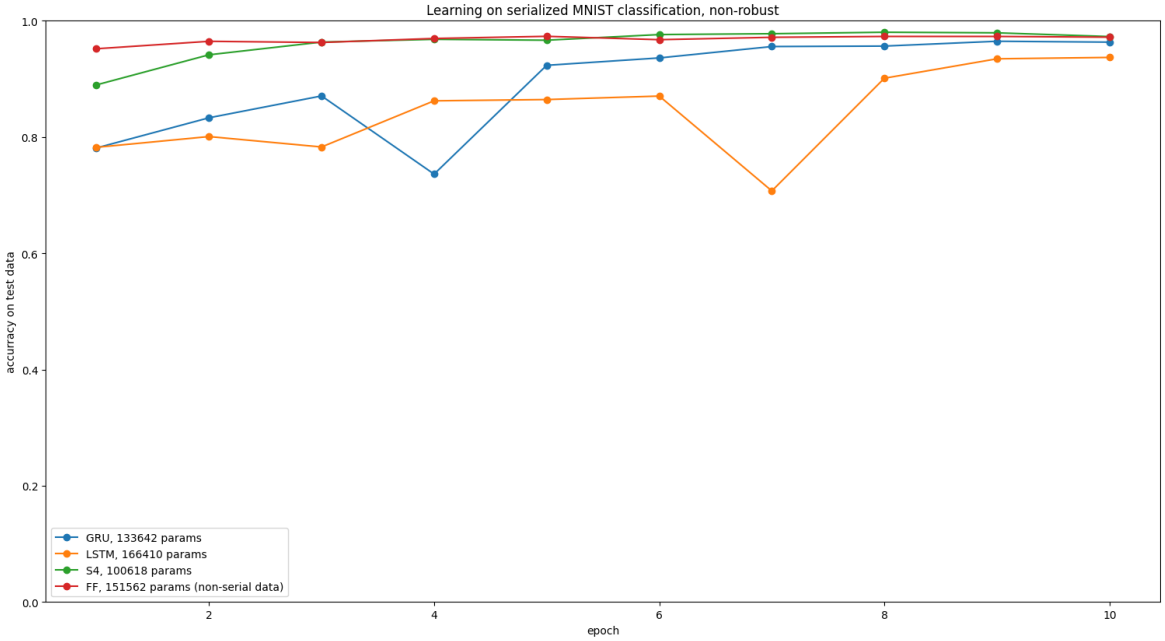


Figure 5.8. Accuracy progress on the MNIST dataset, non-robust training

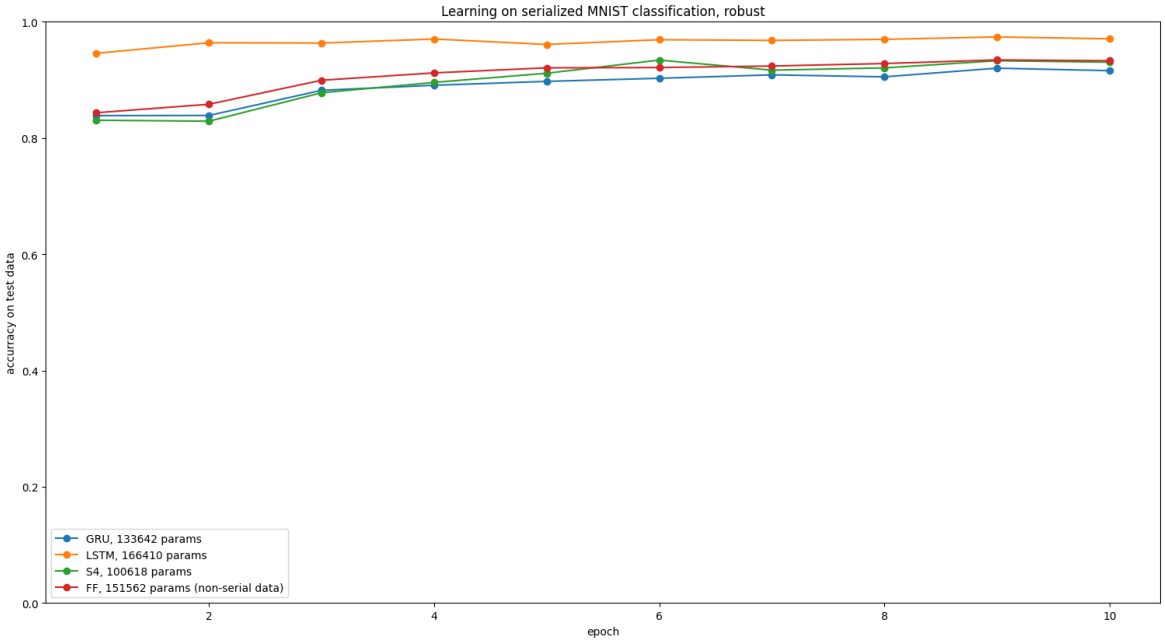


Figure 5.9. Accuracy progress on the MNIST dataset, robust training

The plots on figures 5.8. and 5.9. support the idea of understanding the robust training algorithm as another form of regularization, as it reduces the tendency of overfitting, which consequently lead to the accuracy drops in certain epochs visible on the figure 5.6.. Instead, the models are pushed into generating more general distributions of the data samples, pushing the accuracy up even for the *bening* data samples.

Overall, the approach of the robust training proved to be very succesfull on this dataset, with the only drawback being the increase in computation cost. The experiments thus confirm the findings presented in the original paper.

5.2 German road signs

The GTSRB (German Traffic Sign Recognition Benchmark) dataset is a collection of pictures of road signs photographed under various conditions, from various angles, etc..

It is thus another dataset used for the image classification, intended to possibly support the outcomes achieved on MNIST.

The data samples are RGB images, which means 3 channels per pixel, corresponding to 43 types (classes). That means there are now 3072 (3*32*32) values per each sample. In this thesis, for the serial-data NN models, these samples are streams of the 3 color channels in parallel, that is, with the shape of [1024, 3].

Figure 5.10. An ilustrative subset of the GTSRB dataset, picture taken from a paper [1]

This time, the data are much more complex, having and having much more classification classes makes the task further more difficult. For that reason, I have ran the experiment on the CTU's RCI cluster, running on NVidia's Tesla V100, providing considerably more computing power.

In order to contain the higher complexity of the data samples, I have increased the size of the models, this time, the metaparameters are:

- Size of each layer (number of neurons per layer): 224
- Size of the recurrent hidden state: 324
- Batch size: 24
- Epochs: 20
- Learning rate: 0.001
- Weight decay: 0.001
- Iterations of the robust learning algorithm: 8
- Stepsize of the robust learning algorithm: 0.15

With these metaparameters, the numbers of trainable weights of the neural network models are:

- S4 - 807775 parameters
- FF - 662623 parameters

Again, let's start by having a look on the examples of the *adversarial* data samples:

[1] `https://paperswithcode.com/dataset/gtsrb`

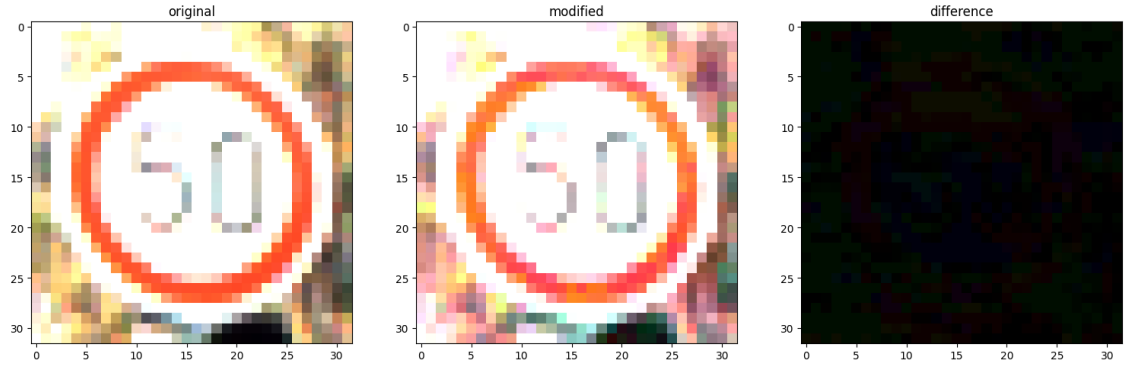


Figure 5.11. PGD attack on GTSRB

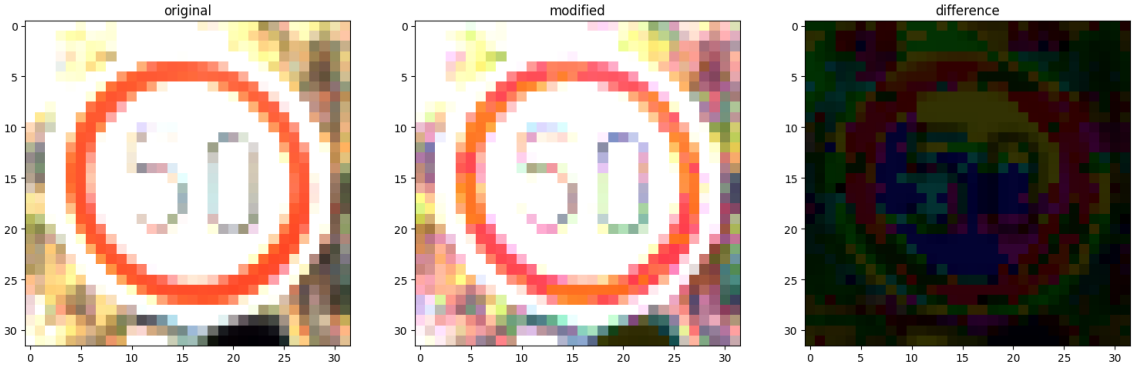


Figure 5.12. FGSM attack on GTSRB

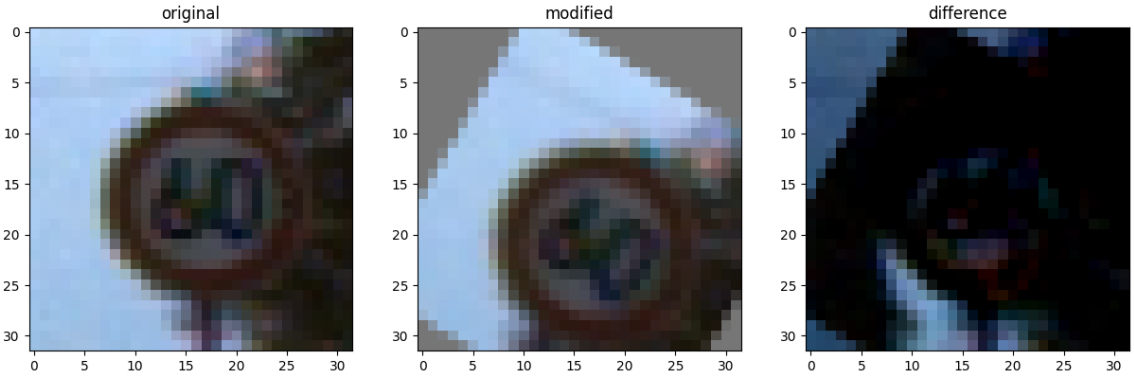


Figure 5.13. Spatial attack on GTSRB

Notice, that the attack vector of the PGD attack is so tiny, it cannot even be printed properly. In the online version, the vector is actually (barely) visible. Yet, even such a small change to the image was enough to make the non-robustly trained S4 architecture misclassify.

Let's have a look on the table of reached accuracies:

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3	Spatial
FF	81.7	13.3	3.9	25.5	8.7	0.0
FF robust	47.4	11.3	0.9	16.4	6.7	0.0
FF regul.	76.0	11.6	4.3	23.3	9.1	0.0
FF rob.+reg.	43.6	9.2	1.0	13.4	5.5	0.0
S4	64.9	0.0	0.0	5.7	6.4	0.0
S4 robust	35.9	3.0	0.7	4.8	3.0	0.0
S4 regul.	62.0	0.5	0.2	5.7	3.4	0.0
S4 rob.+reg.	33.0	2.9	0.1	4.3	3.0	0.0

Table 5.3. Accuracy reached on GTSRB, Algorithm 1

On the *natural* data, models trained with the vanilla training process have by far the best accuracy (remember there are 43 classes considered, much more than the previous MNIST), which is not that surprising. Interestingly, the robust training appears to have no meaningful impact on the models' performance on the *adversarial* samples, either.

It appears, though, this dataset is beyond the capabilities of used neural network models. Even the S4, generally outperforming the GRU and LSTM architecture, struggles to gain any reasonable performance, here. Clearly, convolutional neural networks designed specifically for image recognition would be a better fit for this task. It does not make sense to dive deeper into the image classification.

Regarding the computation time increase, for the dense model, time per epoch went from 27 seconds to 41 seconds, barely changing at all, but for the S4 model, the increased from 41 seconds to roughly 650 seconds (11 minutes per epoch).

The conclusion then seems to be that the robust training is worth only when the size of the classifier allows for containing the whole complexity of the dataset. Because other wise, not only is the increase in computational demands not worth it, it does not appear to grant any benefit at all.

For clearance, let's also add the plot of the accuracy and loss (on the *test* dataset) progression during the epochs, this time for the dense model, without regularizations.

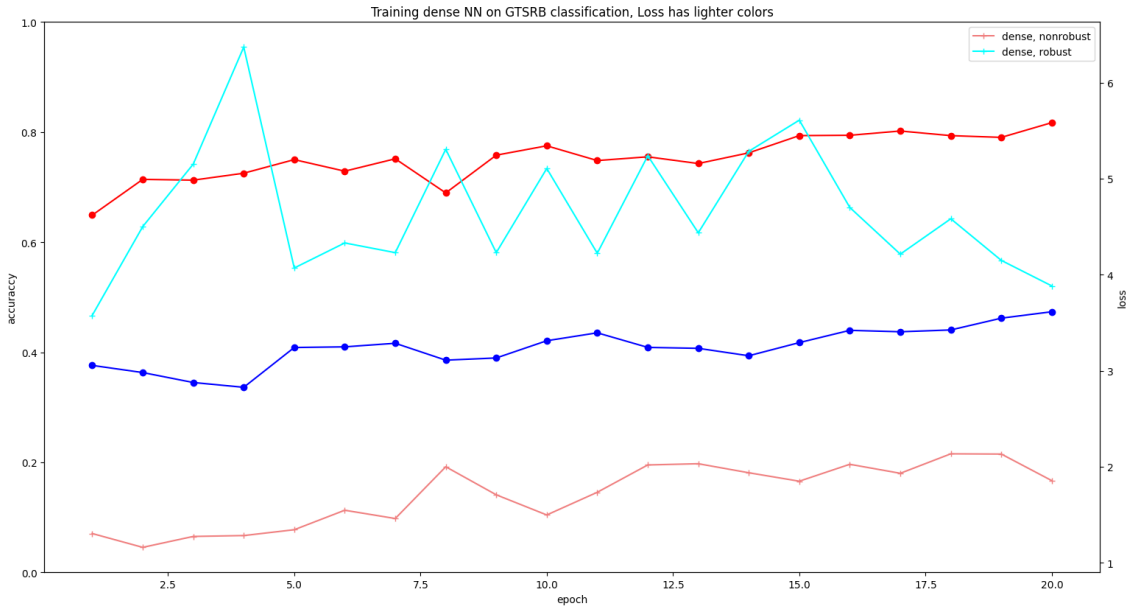


Figure 5.14. Plot for the feed-forward model. The robust (blue lines) training keeps the Loss (right axis) much higher, while keeping the accuracy (left axis) lower.

As the image classification is not a good fit for these neural networks, let's not explore this settings further.

5.3 Speech Commands

5.3.1 Outline of the experiments - direct sound processing

A reasonable alternative classification task could then be the task of classifying sounds. The SpeechCommands dataset contains very small sound recordings (around 1 second long) of 35 voice commands (105829 samples in total), all of them already resampled to the same 16Khz sample rate. The dataset is also provided via the Torch ecosystem, and thus easy to load and use.

The initial idea is to process the sound recording just as it is captured by the microphone, a time series of audio samples/values. This makes the data samples quite long (and thus difficult to train on, as for the recurrent neural network, the gradient must be computed for the whole chain of recurrent entries), and not really used in reality. But it supports the idea of simple input modification to force the classifier into misclassification.

Each *training* data sample is then represented in shape [16000, 1], far surpassing the previous datasets in size per sample. Computing the adversarial gradients and their memory demands proved to be an issue. Remember, that the memory consumption of the algorithms increases together with the number of considered label classes.

To tackle these issues, I had to cut down the number of classes, specifically, I only took the commands `zero`, `one`,...,`nine`, and I also employed one 1D convolution before filling the neural network with the training samples.

For the chosen classes, there are still 38908 samples in the dataset, more than enough for this experiment. They are randomly split into the *train* (80%) and *test* (20%) datasets.

Again, this experiment has been run on the CTU's RCI cluster to help with the time and computational demands.

The metaparameters chosen for the experiment runs are:

- Size of each layer (number of neurons per layer): 64
- Size of the recurrent hidden state: 128
- Batch size: 16
- Epochs: 16
- Learning rate: 0.001
- Weight decay: 0.001
- Iterations of the robust learning algorithm: 10
- Stepsize of the robust learning algorithm: 0.05

We are testing the same scenarios with the same models as with the MNIST dataset, just without the spatial attack (as it obviously does not make the same intuitive sense, these sounds are not 2D images, and rotation by an angle is not defined).

Here, you can see a section of a recording of command `four`, and how the different attacks changed it.

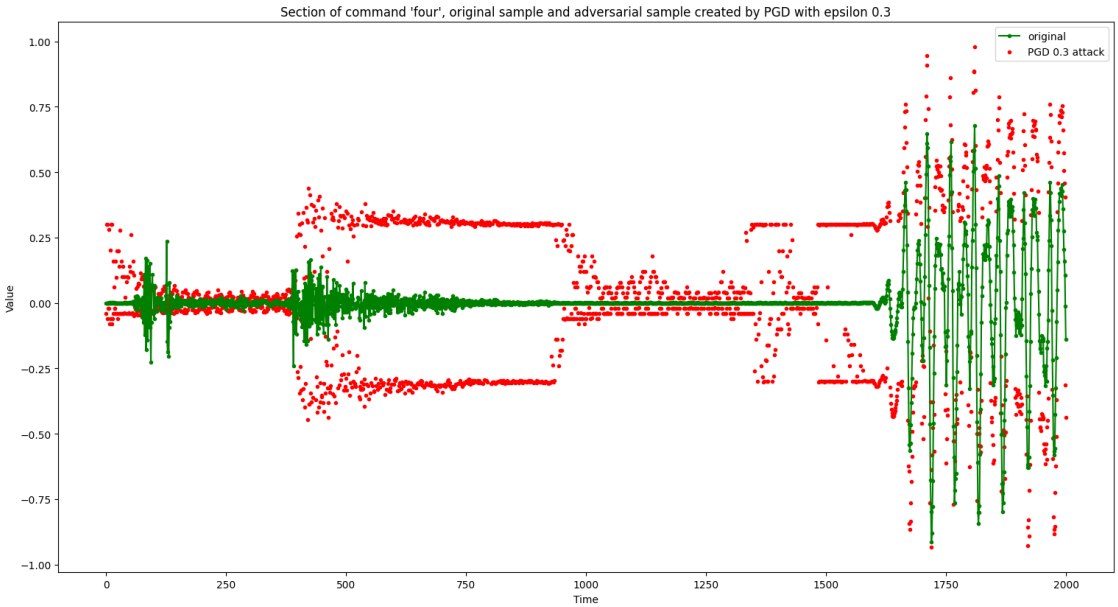


Figure 5.15. PGD ($\epsilon = 0.3$) attack on the command `four`.

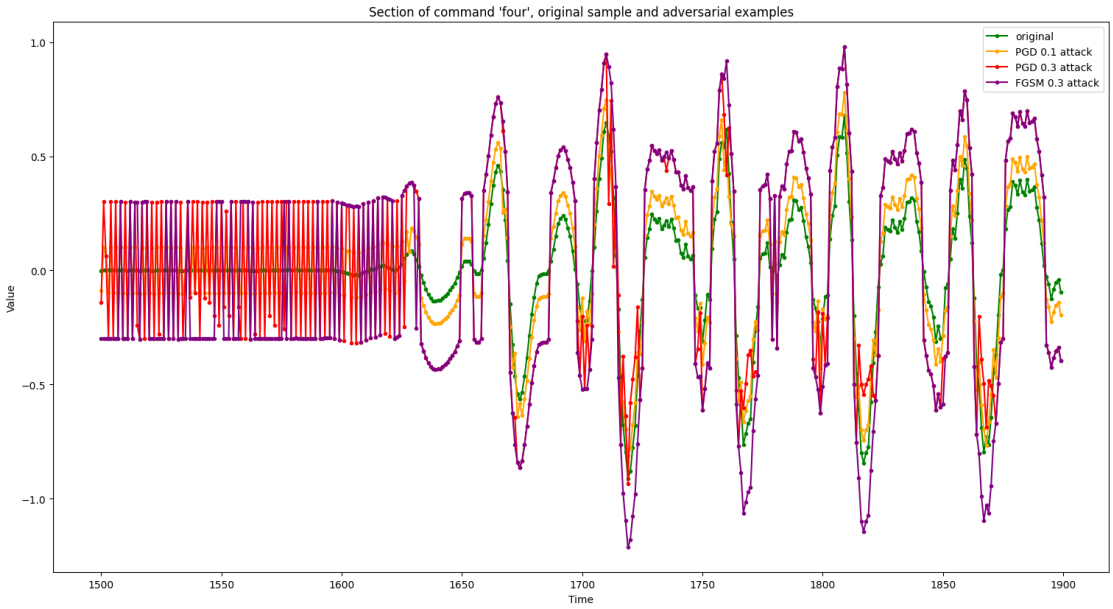


Figure 5.16. Gradient-based attacks on the command `four`.

As shown, the gradient-based attacks have the tendency to modify the original sample differently in different sections of the recordings. Somewhere, just amplifying the values is the most advantageous change possible. Other times, the *adversarial* sample chaotically jumps around the original sample in order to exploit the learned weights of the neural network.

5.3.2 Outline of the experiments - spectrogram processing

Because, as shown in the following section, the performance appears quite poor, I have tried the classification also with the much more common way of processing the sound signal. That is using the so-called Mel spectrogram [16][2]

Using this transformation, based on Fourier transforms over short time periods, the data samples become much more *dense* in some sense, consisting of much fewer individual values. These can be computed in real-time, as the sound is recorded, but the drawback is that it is much more difficult to generate the original sound back from it. Thus, the idea of filtering, or directly changing real-world sounds in some way, is now not as straightforward. Thorough gradient can possibly be computed nonetheless [3].

Out of curiosity, and to increase the dataset variance, I have tried the experiments with 2 distinc cases of this transform, firstly using just the Mel spectrogram (with nfft=1024 parameter)[4], and secondly following the spectrogram by AmplitudeToDB[5] transformation to simplify the data samples even further, as shown in the following images:

[2] `https://www.cs.brandeis.edu/~cs136a/CS136a_docs/KishorePrahallad_CMU_mfcc.pdf`
[3] `https://github.com/SarthakYadav/audax`
[4] `https://pytorch.org/audio/main/generated/torchaudio.transforms.Spectrogram.html`
[5] `https://pytorch.org/audio/main/generated/torchaudio.transforms.AmplitudeToDB.html`

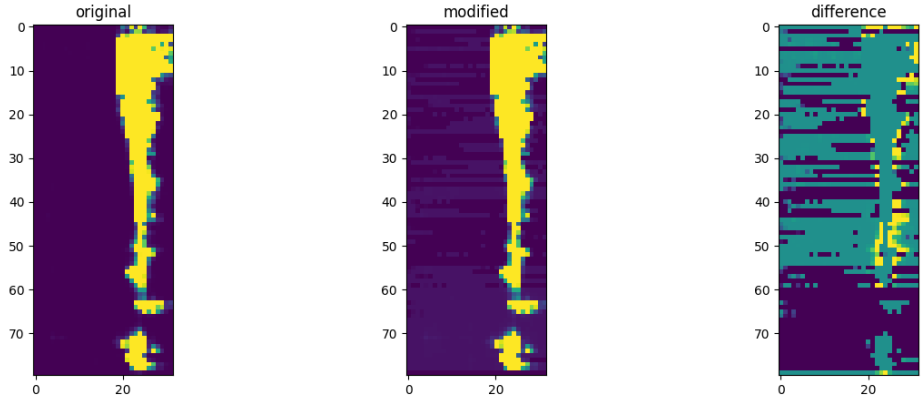


Figure 5.17. PGD attack ($\epsilon = 0.1$) on the spectrogram on the command `four`.

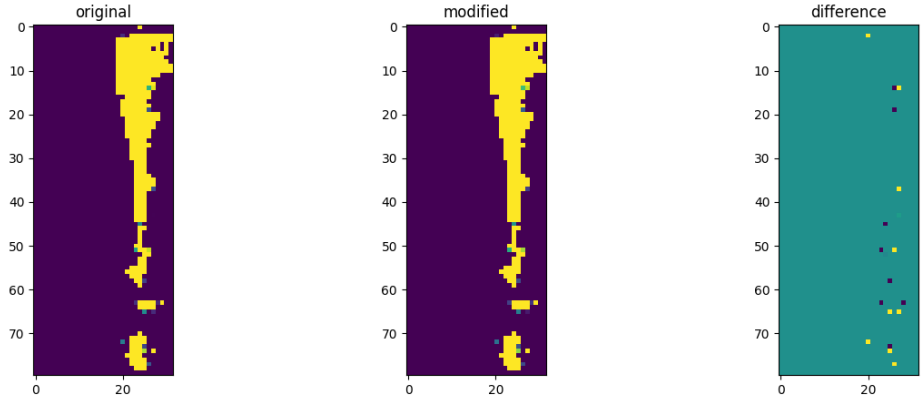


Figure 5.18. PGD attack ($\epsilon = 0.1$) on the spectrogram, followed by AmplitudeToDB, on the command `four`.

The metaparameters chosen for the experiment runs are:

- Size of each layer (number of neurons per layer): 120
- Size of the recurrent hidden state: 196
- Batch size: 32
- Epochs: 16
- Learning rate: 0.001
- Weight decay: 0.001
- Iterations of the robust learning algorithm: 10
- Stepsize of the robust learning algorithm: 0.05

5.3.3 Experiment outcomes

Firstly, let's observe the table of reached accuracies when using Algorithm 1 for training, on the first, direct sound recording classification approach:

Model	Plain	PGD0.1	PGD0.2	FGSM0.1	FGSM0.2
FF	10.8	0.0	0.0	0.0	0.0
FF robust	10.8	0.0	0.0	0.0	0.0
FF regula.	10.3	0.0	0.0	0.0	0.0
GRU	83.7	0.0	0.0	14.1	9.4
GRU robust	47.4	0.0	0.0	19.3	20.8
GRU regul.	14.7	0.2	0.0	15.3	11.2
LSTM	55.5	0.0	0.0	18.0	10.8
LSTM robust	32.2	0.2	0.2	25.3	16.2
LSTM regul.	10.8	0.0	0.0	0.0	0.0
S4	81.0	0.0	0.0	9.2	9.1
S4 robust	55.2	0.0	0.0	20.3	20.8
S4 regul.	12.5	0.0	0.0	0.8	10.3

Table 5.4. Accuracy reached for the SpeechCommands dataset, direct sound processing, Algorithm 1

Clearly, the performance is far from great, both for the robust and the nonrobust training. Interestingly, while the robust training by Algorithm 1 seems to somewhat increase the resistance to FGSM attacks, robustness against PGD attacks remains nonexistent, and accuracy on the *natural* data also decreases dramatically. Since the data samples are quite long, the training process takes a considerable amount of time, making the robust training process of each model roughly 20 hours long.

For the LSTM architecture, computation time went up from 8 minutes to 76 minutes, for GRU it went up from 12 minutes to 47 minutes, and for S4 from 0.75 minutes to 21 minutes.

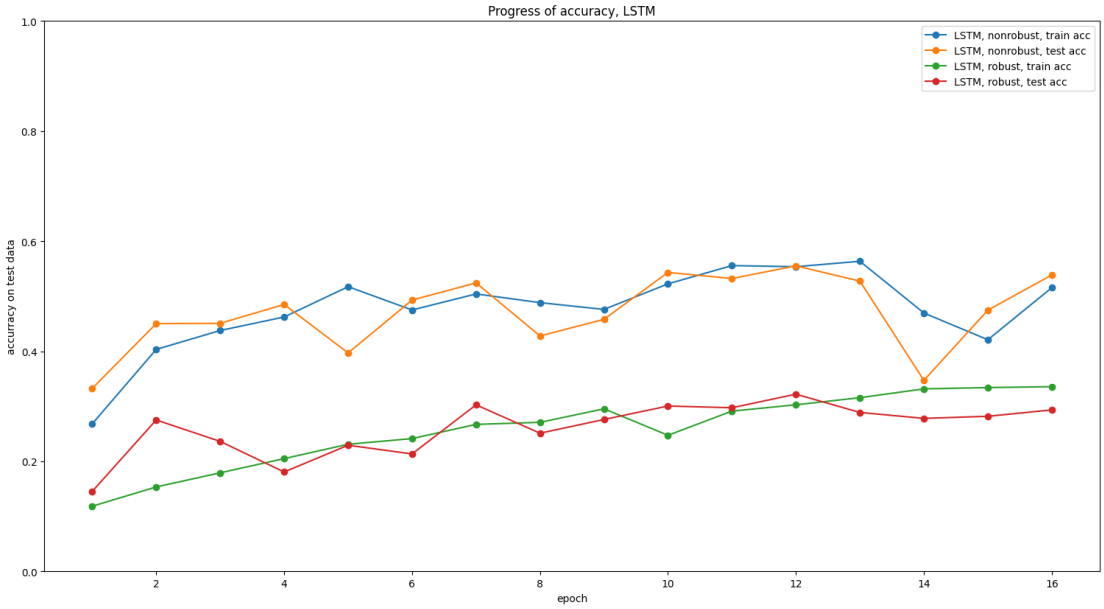


Figure 5.19. Accuracy progression of the LSTM network on the SpeechCommands dataset, direct sound processing, Algorithm 1

As Figure 5.19. suggests, that training the models, which already took many hours, probably won't increase the accuracy significantly more. Moreover, Algorithm 1 did not provide any meaningful benefit at all.

For comparison, let's have a look at the accuracy reached when using the Spectrogram (followed by AmplitudeToDB) approach. The models were trained using Algorithm 1:

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF	70.3	9.6	9.1	68.2	63.8
FF robust	70.5	11.6	10.8	62.8	59.2
FF regul.	11.9	10.0	9.9	10.1	8.3
FF robust+reg.	12.9	9.3	8.0	10.2	9.0
GRU	92.1	47.0	44.6	88.7	85.4
GRU robust	21.2	88.7	86.4	14.1	12.8
GRU regul.	14.3	11.0	10.7	9.0	7.2
GRU robust+reg.	12.5	10.1	10.0	11.4	10.4
LSTM	92.0	50.3	47.6	89.7	87.0
LSTM robust	25.6	87.9	86.5	15.3	13.8
LSTM regul.	13.2	9.5	9.1	8.3	6.9
LSTM robust+reg.	14.2	11.0	10.8	10.3	9.7
S4	93.1	62.2	59.8	89.9	87.0
S4 robust	21.7	88.8	87.6	18.5	15.4
S4 regul.	13.4	8.5	7.6	7.5	6.2
S4 robust+reg.	12.9	11.0	10.6	10.8	9.8

Table 5.5. Accuracy reached for the SpeechCommands dataset, spectrogram+AmplitudeToDB processing, Algorithm 1. See Figure A.3..

With the exception of the feed-forward dense model, the classifiers trained without both robust reached an accuracy of over 90%. They also have quite high resistance against the attack methods. The robust training process greatly increased the accuracy against the *adversarial* data samples created by PGD, but decreased the accuracy on the *natural* data samples almost to zero. It looks like it is too focused on the worst-case samples, and completely trades the potential robustness for accuracy on the unchanged data.

Motivated by that, I have decided to rerun the experiments with Algorithm 2, with my proposal to also include the unmodified data in the robust training process. Also, since it appears like augmentation of the original sound recording does not help with the training process, I have tried augmenting the spectrogram representations instead. Other parameters remained the same. These were the results:

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF	70.3	9.6	9.1	68.2	63.8
FF robust	67.3	9.9	9.5	58.2	54.3
FF regul.	59.8	14.9	14.2	52.2	47.9
FF robust+reg.	60.8	72.1	70.3	54.1	55.2
GRU	92.1	47.0	44.6	88.7	85.4
GRU robust	90.0	87.4	85.1	87.0	83.6
GRU regul.	92.5	72.9	69.6	88.7	85.7
GRU robust+reg.	93.2	65.8	91.9	87.8	84.4
LSTM	92.0	50.3	47.6	89.7	87.0
LSTM robust	90.3	88.0	86.1	81.9	78.3
LSTM regul.	91.2	63.5	60.9	84.1	81.4
LSTM robust+reg.	93.3	68.0	65.4	91.1	89.0
S4	93.1	62.2	59.8	89.9	87.0
S4 robust	90.4	88.3	86.5	81.9	78.3
S4 regul.	91.1	75.9	73.3	88.0	85.5
S4 robust+reg.	94.0	75.3	72.8	91.5	89.1

Table 5.6. Accuracy reached for the Speech Commands dataset, spectrogram+AmplitudeToDB processing, Algorithm 2

Overall, the accuracies reached are quite high, given the relatively small dataset. It shows the correct data transformation can enable much more precise classification.

Surprisingly, the best accuracy was generally (with the exception of the dense model) reached by Algorithm 2 proposed in the previous chapter in conjunction with the random regularizations, even on the *natural* data samples.

That again suggests that there are use cases, for which the robust training approach is a welcomed addition to the training process.

Algorithm 2 does not have the best performance on neither the *adversarial* nor *natural* data samples, on average, however, provides much higher accuracy than the Algorithm 1 proposed in the cited paper [1]. That is well visible in the following Figure 5.20.

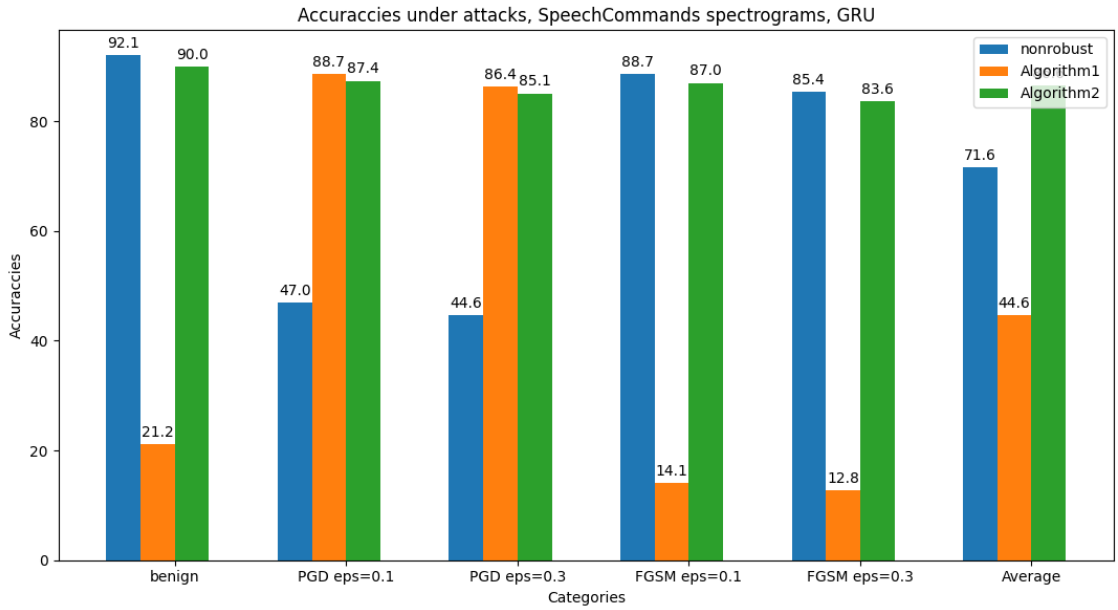


Figure 5.20. Accuracies reached by the two robust algorithms, GRU, SpeechCommands dataset, spectrogram+AmplitudeToDB processing

For the sample representation using only the Mel Spectrogram (without the AmplitudeToDB transform), the outcomes were (not as good as in the previous experiment):

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF	39.4	1.5	0.0	21.1	12.4
FF robust	37.4	5.2	0.8	24.5	14.6
FF regul.	38.5	0.9	0.0	23.1	13.1
FF robust+reg.	39.1	2.5	0.0	25.1	15.5
GRU	85.0	4.1	0.1	48.9	32.4
GRU robust	80.9	31.6	11.5	73.1	60.9
GRU regul.	83.0	16.5	0.3	58.5	33.4
GRU robust+reg.	82.1	25.7	9.9	75.2	62.2
LSTM	82.1	8.0	0.3	44.6	26.1
LSTM robust	79.6	29.1	10.8	71.3	59.3
LSTM regul.	82.3	16.5	0.5	57.7	31.1
LSTM robust+reg.	80.9	26.6	9.2	74.1	60.9
S4	81.4	13.5	1.0	33.4	18.6
S4 robust	75.9	28.3	10.9	64.8	44.6
S4 regul.	77.3	15.4	0.6	35.1	13.3
S4 robust+reg.	76.2	29.4	6.5	62.4	39.8

Table 5.7. Accuracy reached for the Speech Commands dataset, spectrogram only processing, Algorithm 2

5.4 UrbanSound8K

5.4.1 Outline of the experiments - direct sound processing

The second dataset used for the task of sound classification. It consists of 8732 (hence the name) audio files of urban sounds (see the description at the beginning of the chapter) in WAV format. The sampling rate, bit depth, and number of channels may vary from file to file, as it has been recorded by different devices under various conditions.

For the transformation of a sound file into a NumPy array, so that it can be further processed and put into the neural network, I have used the Python library Wave. Afterwards, when all the samples are transformed, a classic Torch dataset and dataloader can be created.

In general, the recordings last 2-4 seconds. That makes the data samples even larger, in general roughly with the shape [40000, 1]. This requires setting batch size even lower.

The data samples consist of these 10 classes:

- air conditioner
- car horn
- children playing
- dog bark
- drilling
- engine idling
- gun shot
- jackhammer
- siren
- street music

To again tackle the issues with the VRAM demands of the gradient computation, and because a few hundred examples per class can be considered too low of a training set, I have re-labeled the samples into two classes: the **dangerous** class consists of dog bark, gun shot, siren, and the **benign** class consists of the rest. Such a grouping is completely artificial, and in practice would anyway be dependent on the customer seeking the trained classifier.

Otherwise, the setup and layout of the experiments remains the same as with the SpeechCommands dataset, that is:

- Size of each layer (number of neurons per layer): 196
- Size of the recurrent hidden state: 240
- Batch size: 16
- Epochs: 16
- Learning rate: 0.001
- Weight decay: 0.001
- Iterations of the robust learning algorithm: 10
- Stepsize of the robust learning algorithm: 0.05

This time, the numbers of learnable weights are:

- Feed-forward - 2574722 parameters
- GRU - 353154 parameters
- LSTM - 438610 parameters
- S4 - 493922 parameters

Here, you can again see a section (with 2 zooms) of a recording of the sound of `children playing` (the *benign* class), and how the different attack methods change it.

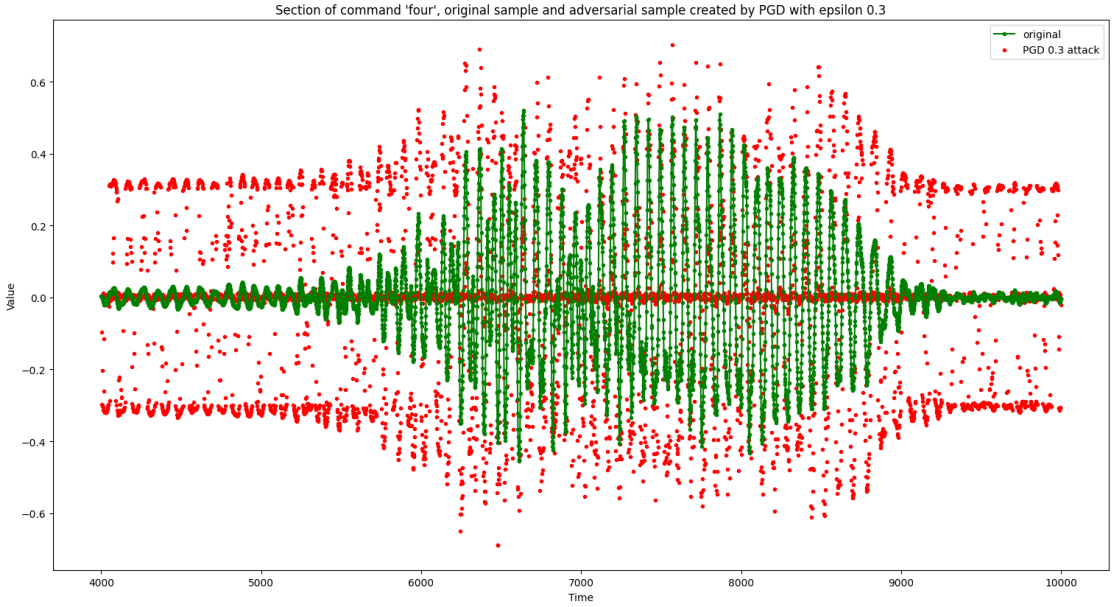


Figure 5.21. Example of PGD ($\epsilon = 0.3$) attack on the sound of `children playing`

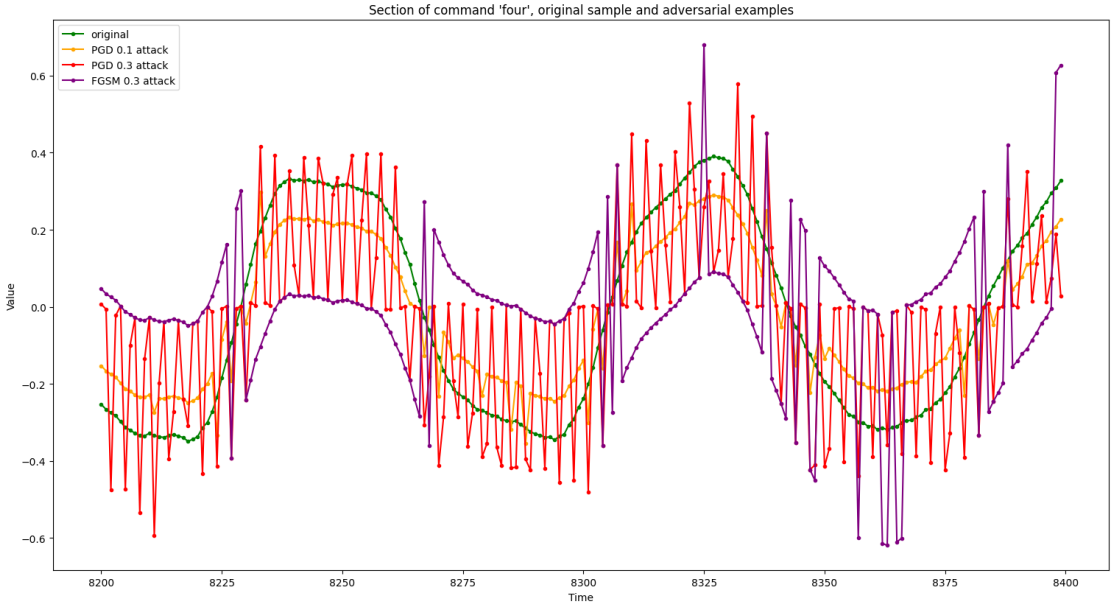


Figure 5.22. Example of PGD ($\epsilon = 0.3$) attack on the sound of `children playing`

Following are the attack examples on the sound of a `dog barking` (the *dangerous* class):

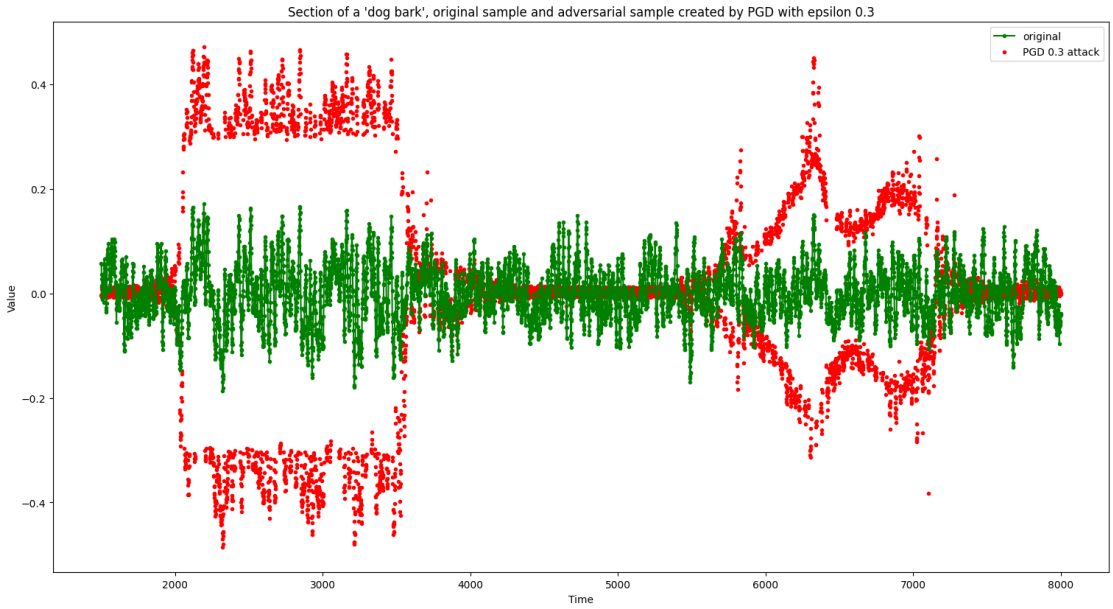


Figure 5.23. Example of PGD ($\epsilon = 0.3$) attack on the sound of a `dog barking`

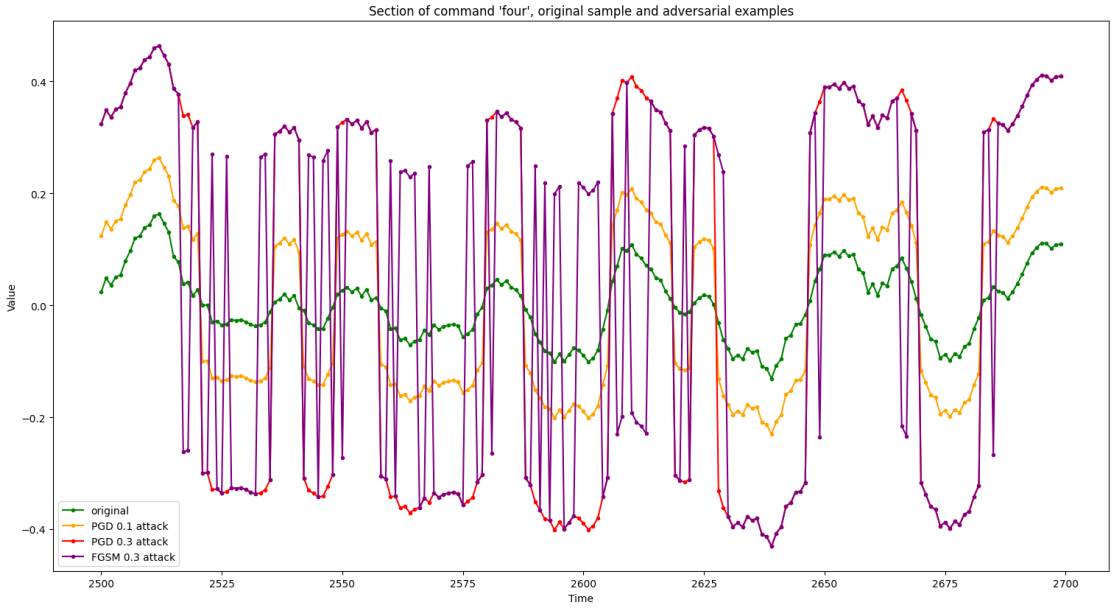


Figure 5.24. Example of PGD ($\epsilon = 0.3$) attack on the sound of a `dog barking`

Again, the gradient-based attacks move the values of the original sample to different directions based on the section of the recording, the adversarial shift is not the same everywhere.

5.4.2 Outline of the experiments - spectrogram processing

Similarly as with the dataset of speech commands, there are experiments with classifying the sounds, after they have been transformed into the Mel spectrogram. Otherwise, the setup remains the same.

For the sake of consistency, both Mel spectrogram and Mel spectrogram+AplitudeToDB transformations are tested, and both are classified by both Algorithm 1 and Algorithm 2.

Some interesting outcomes are presented in the next section, the rest of them are in Appendix A. Interestingly, this time, the difference between the accuracies in the different tasks is not even remotely close.

5.4.3 Experiment outcomes

Firstly, let's again observe the table of reached accuracies, with the direct sound processing approach, tackled by Algorithm 1:

Model	Plain	PGD0.1	PGD0.2	PGD0.3	FGSM0.1	FGSM0.3
FF	71.8	71.4	71.4	71.4	71.4	71.4
FF robust	71.8	71.4	71.4	71.4	71.4	71.4
GRU	77.7	89.3	87.9	66.9	87.1	73.9
GRU robust	74.3	44.6	43.2	42.8	71.4	71.4
LSTM	78.0	72.3	71.8	71.4	71.9	70.9
LSTM robust	74.2	11.7	10.3	10.1	70.0	69.9
S4	78.0	72.3	71.8	71.4	71.9	70.9
S4 robust	77.6	20.8	16.7	15.6	72.2	71.4

Table 5.8. Accuracy reached for the UrbanSound8K dataset, direct sound sample processing, Algorithm 1

In this situation, Algorithm 1 somehow decreased the accuracy across the board, both on the *natural* data, but mainly on the *adversarial* samples. I do not have a good explanation for that.

The time increase was again noticeable, for LSTM, it went up from 275 seconds (around 5 minutes) per epoch to roughly 3800 seconds (a bit over an hour). For the S4 model, it went up from 51 seconds to 1000 seconds when employing Algorithm 1. For the GRU model, it went up from 440 to 4200 seconds. The train time for the dense FF architecture was 0 seconds in both cases, as it did not have to unwind the recurrent gradient.

The remaining are only the numbers from trying the classification based on spectrogram+AmplitudeToDB processing instead. There are both the {Algorithm 1, Algorithm 2}

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF	67.9	12.8	12.2	65.9	61.3
FF robust	70.5	11.6	10.8	62.8	59.2
FF regul.	11.9	10.0	9.9	10.1	8.3
FF robust+reg.	12.9	9.3	8.0	10.2	9.0
GRU	80.1	68.9	58.0	74.1	64.8
GRU robust	78.9	67.1	61.0	75.8	72.9
GRU regul.	78.8	68.9	60.1	74.5	65.5
GRU robust+reg.	77.8	58.1	48.5	62.2	50.6
LSTM	79.5	67.5	51.8	70.2	61.7
LSTM robust	77.3	65.3	55.0	72.6	66.7
LSTM regul.	79.7	57.8	14.8	62.4	55.5
LSTM robust+reg.	79.3	60.7	52.7	72.5	63.6
S4	79.8	65.1	43.6	70.7	65.5
S4 robust	79.6	71.4	70.3	73.4	67.6
S4 regul.	77.5	24.3	14.6	67.4	61.1
S4 robust+reg.	77.3	71.4	69.2	73.0	63.9

Table 5.9. Accuracy reached for the UrbanSound8K dataset, spectrogram processing, Algorithm 1

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF robust	75.8	64.2	52.8	71.9	60.2
FF robust+reg.	78.2	63.4	25.1	68.9	60.0
GRU robust	79.9	63.0	55.4	75.4	67.0
GRU robust+reg.	80.5	59.6	39.2	57.7	45.2
LSTM robust	77.3	66.6	58.5	75.7	72.0
LSTM robust+reg.	80.5	60.5	54.7	63.4	57.1
S4 robust	78.7	70.6	60.5	71.7	63.0
S4 robust+reg.	79.2	70.6	57.2	68.2	55.9

Table 5.10. Accuracy reached for the UrbanSound8K dataset, spectrogram processing, Algorithm 2

On this dataset, Algorithm 1 appears to successfully increase the resistance against the gradient-based attacks, basically without affecting the accuracy on *natural* data samples. Sadly, the increase is generally not that big.

As expected, the performance of Algorithm 2 is somewhere between non-robust training and Algorithm 1.

Overall, there seems to be no clear pattern, and from this experiment, it is difficult to support the time invested in either Algorithm 1 or Algorithm 2, when it comes to this specific classification task.

5.5 Training Time increase

Let's have a look at the figure of times required for the training epoch. These are various recorded time increases on different architectures, sizes of the neural network and other parameters, but it still provides a quick idea, of what the relation is.

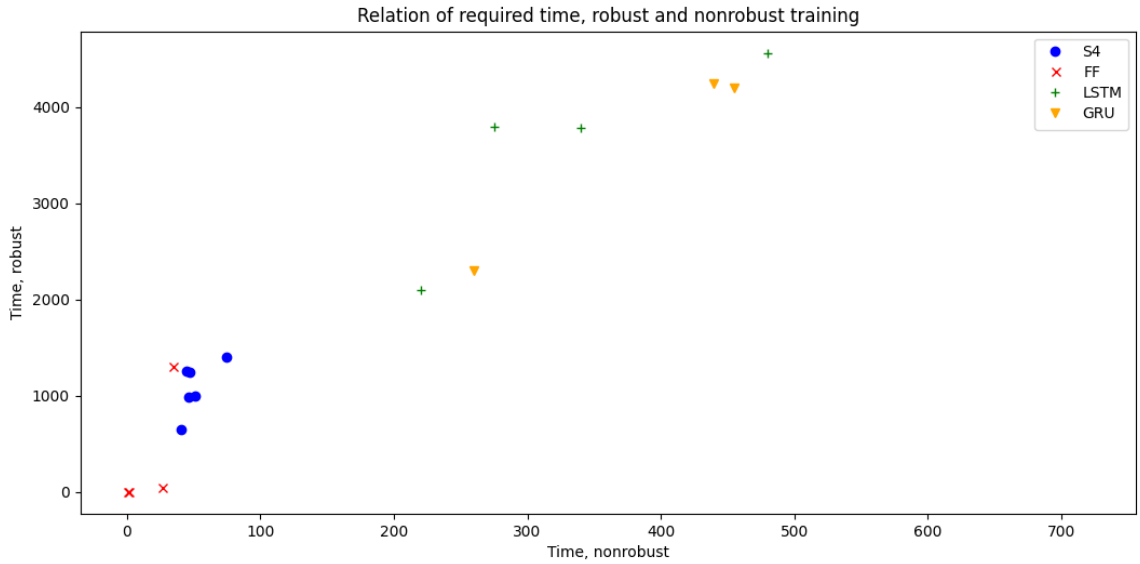


Figure 5.25. Plot of the trend of the increase of the required time by the robust training process

Even though the specific points in Figure 5.25. were taken from various architectures classifying various datasets, the trend appears to be linear, where the robust training done by Algorithm 1 takes roughly one order of magnitude more time. Run time of Algorithm 2 is almost identical to the time of Algorithm 1.

Chapter 6
Conclusion and discussion

6.1 The effect of robust learning

As confirmed in this thesis, the game-theoretical approach and the derived Algorithm 1 is very successful on the MNIST dataset. Not only does it provide high resistance against the gradient-based modified, adversarially crafted samples using the Fast Gradient Sign Method and the Projected Gradient Descend Method. It also acts as a form of regularization of the training process, with the ability to increase the accuracy even on *natural* data samples.

The general outcome, however, is not so conclusive, and hugely depends on the specific situation. There are instances, as seen in Table 5.4., where it possibly only decreases the accuracy on *natural* data samples, without granting any resistance against the attacks, providing only disadvantages

As seen in Table 5.5, the robust training can provide high resistance against the gradient-based attacks, yet decimate the accuracy on the *natural* data at the same time; that can be possibly resolved by a slight modification of the approach (Algorithm 2), as seen in the Table 5.6.

The robust training, however, does not appear to significantly increase the models' performance against other types of attacks based on creating the *adversarial* data samples.

Generally speaking, it seems Algorithm 1 (or its modifications) can be advantageous only in those cases when the dataset is well containable by the neural network and is thus prone to overfitting.

6.2 Synergy with regularizations

In all instances, the random noise (and possibly other regularization methods) decreased the provided robustness against the gradient-based attacks. An explanation could be that the added noise just shifts the optimal gradient-based attack in another direction.

The resistance against the attacks usually remains much higher than without the robust training, and there are some cases, where it increases accuracy even on the *natural* data, it is thus a beneficial combination.

No definitive answer can be said, though.

6.3 Computational price of the robust learning

The increase in computation time demand is not negligible. Even with the JAX implementation running on dedicated TPUs, which can be considered the best scenario, the run time can easily reach several hours per epoch. Generally said, as shown in Figure 5.25, the required time for robust training is an order of magnitude longer.

Also, if the batch size must be reduced because the algorithm must keep a modified data sample copy of each considered class (as explained in Chapter 2) in the memory, and thus requiring considerably more of it, the run time of the training phase can be increased even further.

6.4 Conclusions

Provided with a comfortable gradient computation, implementation of the algorithm of the robust training is not that difficult. It has the opportunity to make the neural network robust against *adversarial* data samples, making the accuracy almost as high as on the *natural* data samples. It possibly even increases the performance on *natural* data samples, acting as some kind of regularization method. Unfortunately, it does not seem to increase performance against non-gradient attack methods.

It appears that the algorithm's performance does not depend on a specific architecture, as the effect on all the tested models is very similar. When considering the usage of robust training, the specific neural network used is not important, which is good. Just changing the specific neural network in a specific use case, when retaining the robust training, will have predictable outcomes.

Its performance seems to be the best when the classifier tends to overfit. In such use cases, however, other classification algorithms, rather than a neural network, might be considered. Still, a similar robustness approach can be constructed for those classifiers, too. If the computational cost is not an issue, or if it is well justified, extending the training algorithm for robustness might be worthy of the investment. Motivated by the fact we can expect some neural networks to be commonly re-deployed with the pre-trained weights, the robust training only can make its performance potentially higher. It is not, however, a simple case, and not always does the implementation of this robust training algorithm even provide any benefit. It is not the case of "just use the algorithm, it always provides additional resistance against adversarial".

The decision then depends on answering the question, whether the potential benefits outweigh the cost of the long training process, even if it possibly does not provide any.

References

[1] Maher Nouiehed, Maziar Sanjabi, Tianjian Huang, Jason D. Lee, and Meisam Razaviyayn. *Solving a Class of Non-Convex Min-Max Games Using Iterative First Order Methods*. 2019.
`https://arxiv.org/pdf/1902.08297`. Advances in Neural Information Processing Systems 32, 2019, pages 14 905-14 916..

[2] François Fleuret. *The Little Book of Deep Learning*. Alanna Maldonado, 2023. ISBN 9789732346495.

[3] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P. Xing, Laurent El Ghaoui, and Michael I. Jordan. *Theoretically Principled Trade-off between Robustness and Accuracy*. 2019.
`https://arxiv.org/pdf/1901.08573`. NeurIPS 2018 Adversarial Vision Challenge.

[4] Meisam Razaviyayn, Tianjian Huang, Songtao Lu, Maher Nouiehed, Maziar Sanjabi, and Mingyi Hong. *Non-convex Min-Max Optimization: Applications, Challenges, and Recent Theoretical Advances*. 2020.
`https://arxiv.org/pdf/2006.08141`. EEE Signal Processing Magazine, 2020, 37.5, pages 55-66..

[5] Mingxing Tan, and Quoc V. Le. *EfficientNetV2: Smaller Models and Faster Training*. 2021.
`https://arxiv.org/pdf/2104.00298`. https://pytorch.org/vision/main/models/efficientnetv2.html.

[6] Yoav Shoham, and Kevin Leyton-Brown. *Multiagent Systems*. Cambridge University Press, 2008. ISBN 9780521899437.

[7] John Nash. *Non-Cooperative Games*. September 1951.
`https://www.cs.vu.nl/~eliens/download/paper-Nash51.pdf`. In: Annals of Mathematics, Second Series, Vol. 54, N^o. 2 , pages 286-295 .

[8] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015.
`https://arxiv.org/pdf/1412.6572`. ICLR 2015: San Diego, CA, USA.

[9] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. *Adversarial examples in the physical world*. 2017.
`https://arxiv.org/pdf/1607.02533`. International Conference on Learning Representations, 2016.

[10] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. *Adversarial Machine Learning at Scale*. 2017.
`https://arxiv.org/pdf/1611.01236`. International Conference on Learning Representations, 2017.

[11] Jonas Rauber, Wieland Brendel, and Matthias Bethge. *Foolbox: A Python toolbox to benchmark the robustness of machine learning models*. In: *Reliable Machine*

Learning in the Wild Workshop, 34th International Conference on Machine Learning. 2017.
https://arxiv.org/pdf/1611.01236. https://github.com/bethgelab/foolbox/tree/master.

[12] Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. *Exploring the Landscape of Spatial Robustness*. 2017.
https://arxiv.org/pdf/1712.02779. Proceedings of the 36th International Conference on Machine Learning, June 2019.

[13] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014.
https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf. Journal of Machine Learning Research 15, (2014) 1929-1958.

[14] Li Deng. *The MNIST Database of Handwritten Digit Images for Machine Learning Research*. 2012.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6296535. IEEE Signal Processing Magazine, Volume: 29, Issue: 6, November 2012, Pages 141 - 142 .

[15] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018.
https://arxiv.org/pdf/1804.03209.

[16] Boyang Zhang, Jared Leitner, and Samuel Thornton. *Audio Recognition using Mel Spectrograms and Convolution Neural Networks*. 2019.
noiselab.ucsd.edu/ECE228_2019/Reports/Report38.pdf.

Appendix A
Additional results

A.1 MNIST results

Firstly, here is the effect of the robust training parameters, going from 8 to 10 iterations (k), but decreasing the *stepsize* from 0.15 to 0.05., making the worst-case *adversarial* modification more precise. Because more gradients had to be computed, the computational time increased again, roughly by 10 percent (for GRU, time increased from 2300 to 2540 seconds).

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF robust	96.2	91.9	69.7	89.3	51.9
GRU robust	95.6	94.2	92.1	94.6	93.2
LSTM robust	92.7	89.6	85.5	91.3	90.0
S4 robust	98.4	96.9	93.5	97.0	94.2

Table A.1. Accuracy of the different scenarios reached for MNIST dataset, smaller *stepsize*, Algorithm 2

These scores are almost identical to the initial experiment, which implies that the algorithm's performance is not that sensitive to the chosen parameters.

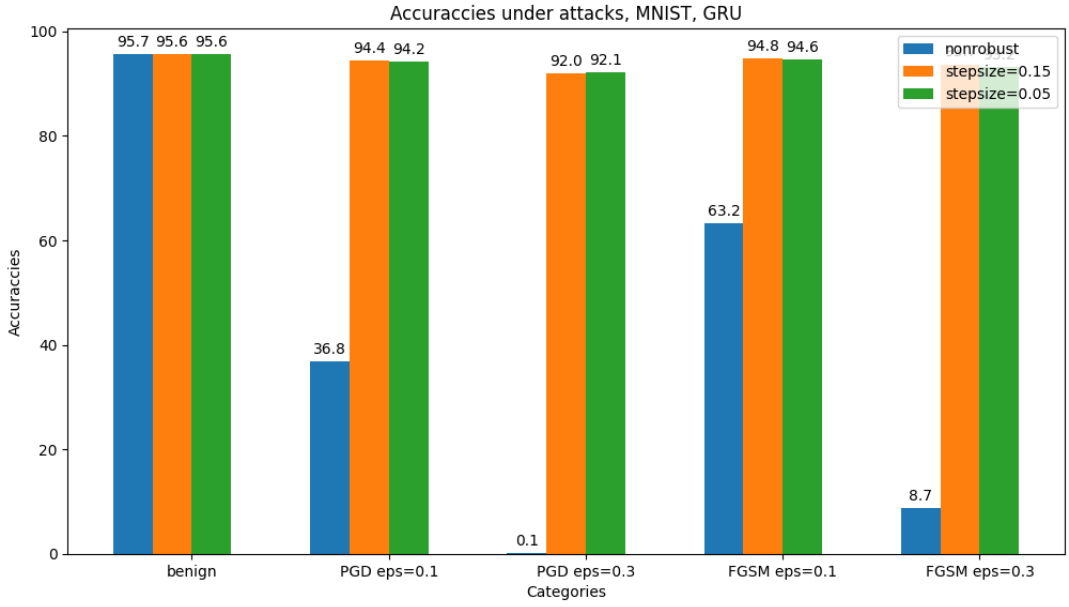


Figure A.1. Plot of the effect of the different values of stepsize of the algorithm

A.2 GTSRB results

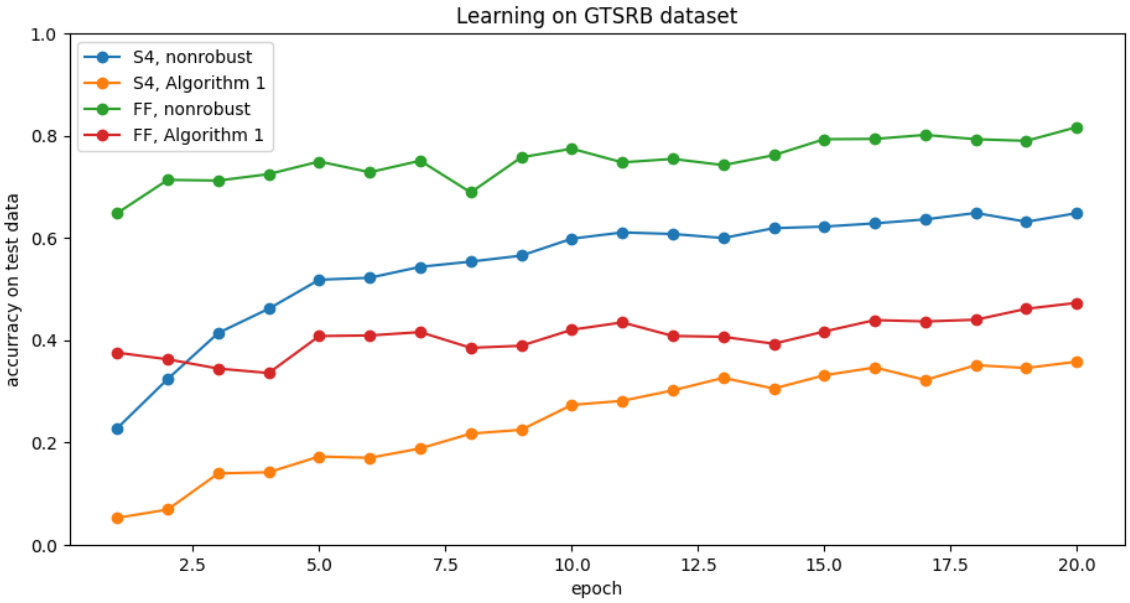


Figure A.2. Progress of the accuracy reached on the GTSRB dataset

When training the neural networks using Algorithm 1, there is a clear tendency to learn slower. Nothing, however, indicates the accuracy should become considerably better with further epochs.

To support this claim, though, I have also tried to train the dense FeedForward neural network for 50 epochs, and under Algorithm 1, it never surpassed 53.3%. You can see the output of this specific experiment, as well as the others, in folder `outputs` on the associated GIT repository.

A.3 SpeechCommands results

This is the plot of accuracies under the attack methods, corresponding to Table 5.5., for simpler comprehension. The robust training makes the neural network more robust against the PGD attacks but does not provide a real advantage overall.

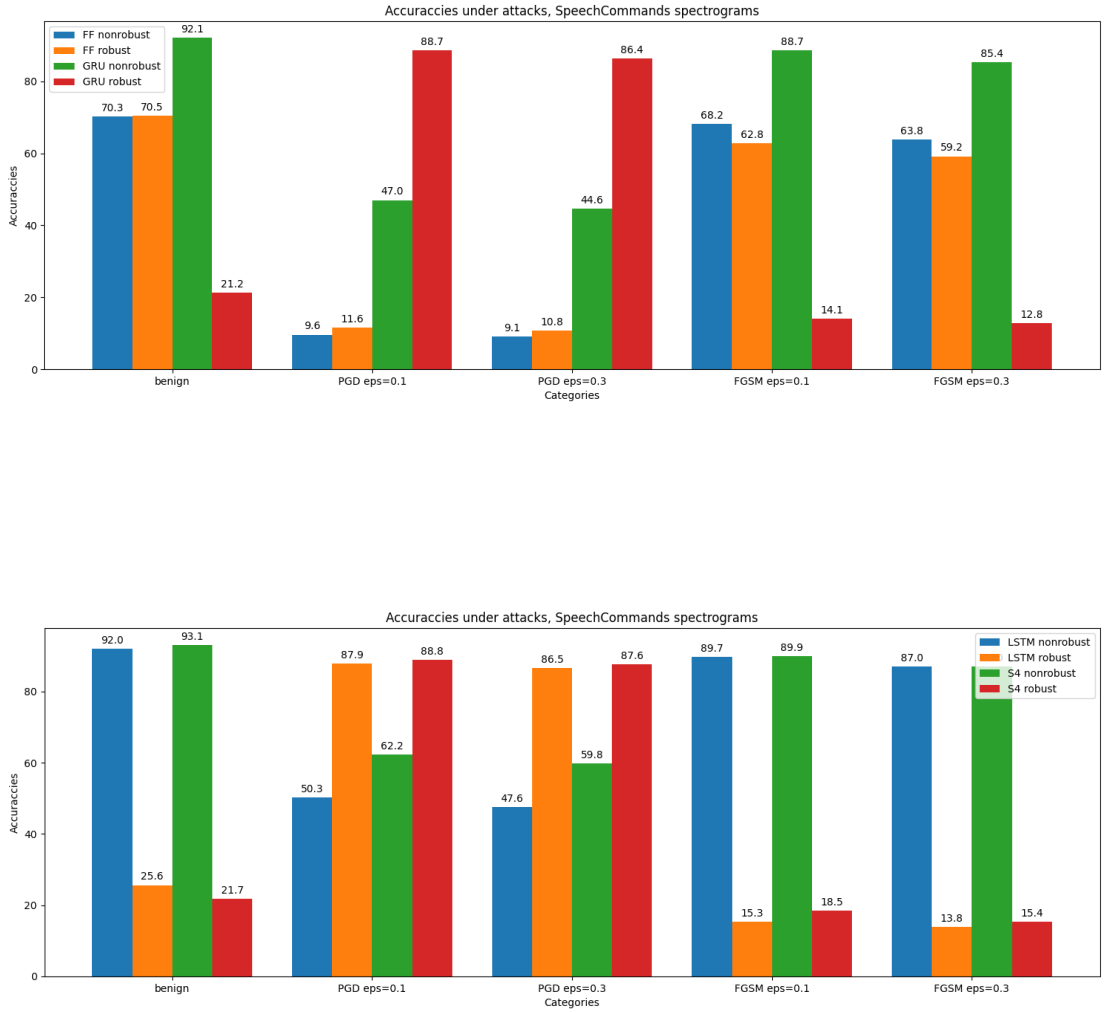


Figure A.3. Plot corresponding to Table 5.5., SpeechCommands classification, Mel spectrogram+AmplitudeToDB, Algorithm 1

Following is the plot of progress of the accuracy on the *test* datatest, and the loss on the *train* dataset, supporting the idea of the robust training as a form of regularization.

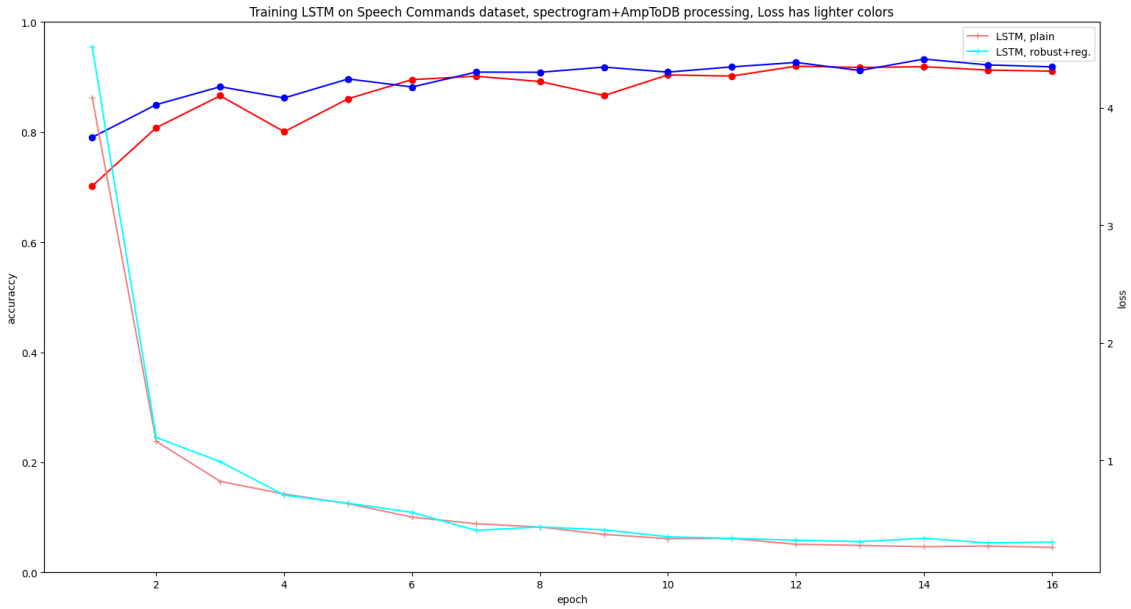


Figure A.4. Higher Increase in accuracy on the *test* data when the *training* loss is increased. See Table 5.6.

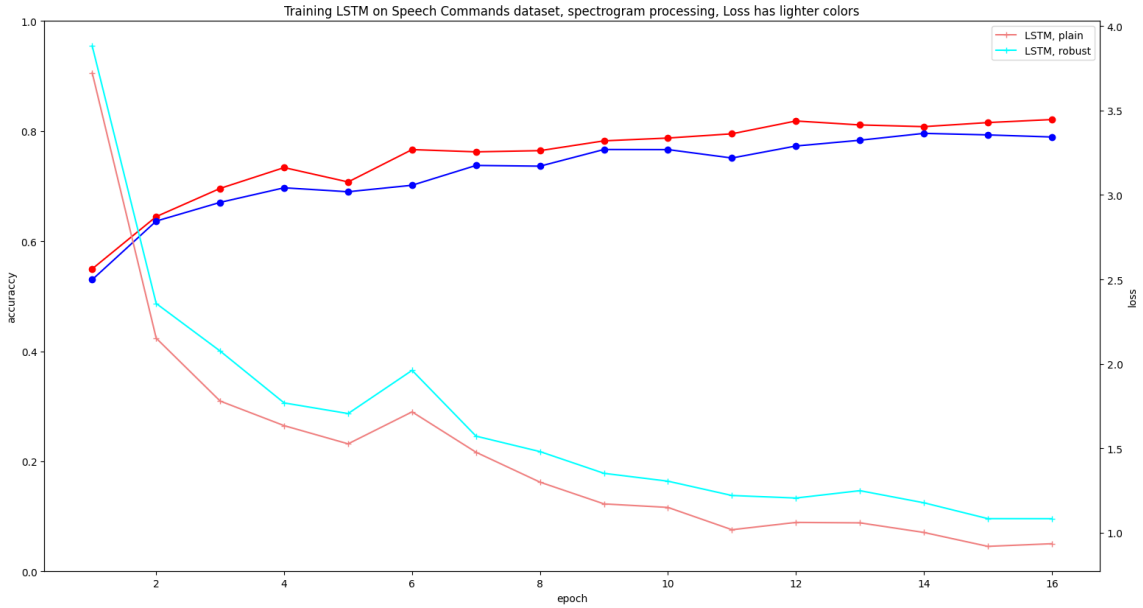


Figure A.5. Higher Increase in accuracy on the *test* data when the *training* loss is increased. See Table 5.7.

A.4 UrbanSound8K results

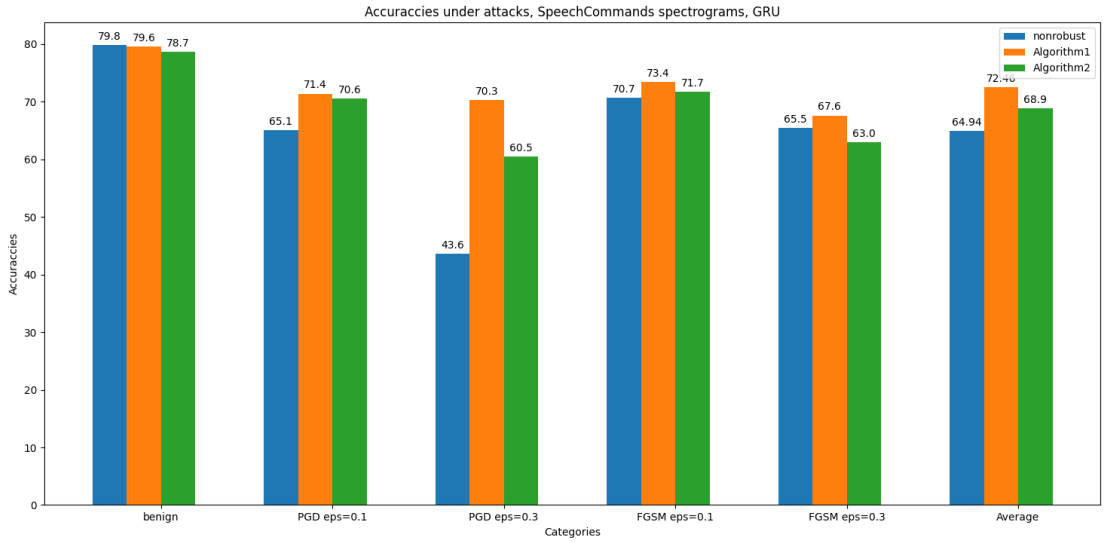


Figure A.6. Comparison of performance of Algorithm 1 and Algorithm 2 (see Table 5.9. and 5.10.) on UrbanSound8k classification

And finally, a comparison of Algorithm 1 and Algorithm 2 on the UrbanSound8K dataset, with the Mel spectrogram+AmplitudeToDB transformation. The outcomes are, in this setting, very similar.

Model	Plain	PGD0.1	PGD0.3	FGSM0.1	FGSM0.3
FF	84.5	48.2	42.3	78.1	69.9
FF Alg.1	84.1	48.8	43.9	79.1	71.6
FF Alg.2	84.7	43.1	36.2	77.0	71.1
GRU	90.5	39.8	39.1	79.8	75.8
GRU Alg.1	90.4	40.9	39.9	84.6	82.1
GRU Alg.2	90.1	43.1	41.1	86.1	84.7
LSTM	89.1	41.8	41.6	81.6	77.1
LSTM Alg.1	87.6	43.7	42.4	78.6	74.5
LSTM Alg.2	88.1	43.4	42.6	72.4	67.8
S4	88.3	40.3	38.5	85.5	79.6
S4 Alg.1	89.2	42.5	40.9	87.2	81.1
S4 Alg.2	89.2	41.3	39.6	87.2	81.4

Table A.2. Accuracy reached for the UrbanSound8K dataset, Mel spectrogram+AmplitudeToDB processing

Appendix B
Assignement

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kasl** Jméno: **Tomáš** Osobní číslo: **474747**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Metody min-max optimalizace v adversariálním učení

Název diplomové práce anglicky:

Min-max optimization methods in adversarial learning

Pokyny pro vypracování:

Min-max problems in adversarial and robust ML involve optimizing models against worst-case scenarios to enhance security and reliability. Adversarial ML focuses on identifying and mitigating vulnerabilities through min-max frameworks, where attackers maximize model errors while defenders minimize them. Robust ML aims to develop algorithms resilient to such attacks, ensuring model performance remains stable even under adversarial conditions. This dynamic interplay enhances model robustness, making ML systems more secure and dependable in real-world applications. The goals of the thesis are the following.
1. To implement the robust version of the model of classifier based on [3] or [2] and the methods of min-max optimization (GDA and its variants).
2. To analyze the properties of found solutions from the viewpoint of optimization and game theory.
3. (Optional) If possible, to build the classifier model on the new S4 architecture [1].

Seznam doporučené literatury:

[1] Gu, Albert, Karan Goel, and Christopher Ré. "Efficiently modeling long sequences with structured state spaces." arXiv preprint arXiv:2111.00396 (2021).
[2] M. Nouiehed, M. Sanjabi, T. Huang, J. D. Lee, and M. Razaviyayn, “Solving a class of non-convex min-max games using iterative first order methods,” in Advances in Neural Information Processing Systems 32, 2019, pp. 14 905-14 916.
[3] M. Razaviyayn et al. Nonconvex min-max optimization: Applications, challenges, and recent theoretical advances. IEEE Signal Processing Magazine, 2020, 37.5: 55-66.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Tomáš Kroupa, Ph.D. centrum umělé inteligence FEL

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

Datum zadání diplomové práce: **09.02.2024** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **21.09.2025**

doc. Ing. Tomáš Kroupa, Ph.D. 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