Université Paris Dauphine - PSL

Numerical methods for deterministic and stochastic problems

Academic year 2023/2024

Adversarial examples in “pure Python”

Belo Nassauer David, Bonazza Arianna, Magnino Lorenzo

Contents

The aim of this project is to study the vulnerability of neural networks to adversarial examples and its implications for image classification systems.
This is done through the MNIST dataset (modified National Institute of Standards and Technology), a large database of 60,000 gray-scale train images (each of those is made of 28×28 pixels and represents one of the 10 digits) plus 10,000 additional labelled test images. Our work is based on neural networks' architecture and backpropagation for the computation of the gradient.

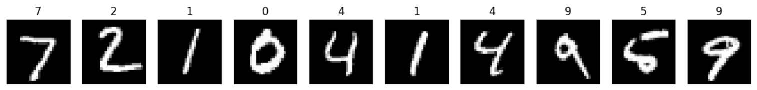


Figure 1: The first 10 images of the test database, with labels

1 Introduction

Goodfellow 2015 [1] showed that several machine learning models for classification, as state of art neural network suffer of very small perturbation of the input. This cases are called adversarial example: the neural network misclassifies images that are only slightly different from correctly classified examples.

This phenomenon highlights a big weakness for this types of neural network machines: they do not learn the real underline concept that determined the correct classification of an object but build a Potemkin village that works well on naturally occurring data, but is exposed as a fake when one visits points in space that do not have high probability in the data distribution. This is particularly dangerous because a popular approach in computer vision is to use convolutional network features as a space where Euclidean distance approximates perceptual distance. This tools is clearly flawed if images that have an immeasurably small perceptual distance correspond to completely different classes in the network's representation.

In the past this kind of behavior of neural networks was associated to extreme non-linearity, insufficient regularization and overfitting.

However Goodfellow 2015 [1] showed that, on the contrary, we can find big weakness on adversarial examples on linear model in high dimensions while non linear model as RBF network are adverse to this problem.

This led us to a trade off in building a neural network between the simplicity of its architecture (link to its linearity) and the used of non linear effect to prevent adversarial perturbation.

2 Understanding Adversarial Examples

2.1 Perturbation in Linear Models

Perturbation in linear models come straightforward and it is fundamental to understand the more complicated situations, as in the non linear case. Indeed in many problems the precision of the input is limited, like in image recognition where digital images often use 8 bit per pixel so they discard all the information below 1/255 of the dynamic range. The idea is that we expect that the network classifies in the same way an input $\mathbf{x}$ and an adversarial input $\tilde{\mathbf{x}} =: \mathbf{x} + \boldsymbol{\eta}$ when $\boldsymbol{\eta}$ is smaller than the precision of the features.

Let's formalize it: for problems of well-separated classes we expect that the network assign the same class to $\boldsymbol{x}$ and $\tilde{\boldsymbol{x}}$ so long as $||\boldsymbol{\eta}||_\infty \leq \epsilon$ where ϵ is small enough in order to to be discarded by the sensor associated to out problem.

Giving the linear structure of out network we can consider the dot product between the weights and the adversarial example:

$$\boldsymbol{w}^{\boldsymbol{T}}\tilde{\boldsymbol{x}} = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{\eta}$$

We can see that the error caused by the perturbation is measured by $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{\eta}$. So the purpose of our work is to maximize it in order to reach a misclassification. A smart way to do it is to use $\boldsymbol{\eta} = sign(\boldsymbol{w})$. Indeed if $\boldsymbol{w}$ has dimension n and each element of $\boldsymbol{w}$ has magnitude m then the activation will grow as ϵmn. The key concept that permit us to let the error grow is that the norm of η do not grow with the dimensionality of the problem while the perturbation cause by it grow linearly with n. So with high dimensional problems we can make infinitesimal changes to the input that add up to one large change to the output.

This explanation shows that a simple linear model can have adversarial examples if its input has sufficient dimensionality.

2.2 Perturbation in Non-Linear Models

When in our models are present non linear activation function as ReLU, LSTMs, maxout networks it doesn't change the idea of linear models since they are designed to behave in very linear ways, so that they are easier to optimize. Even totally non linear activation function as sigmoid are carefully tuned to spend most of their time in the non-saturating, more linear regime for the same reason.

Let $\boldsymbol{\theta}$ denote the parameters of a model, x the input to the model, $\boldsymbol{y}$ the targets associated with $\boldsymbol{x}$ (for machine learning tasks that have targets), and $J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})$ be the cost function used to train the neural network. We can linearize the cost function around the current value of $\boldsymbol{\theta}$, obtaining an optimal max-norm constrained perturbation given by

$$\boldsymbol{\eta} = \epsilon \cdot \operatorname{sign}\left(\nabla_x J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right).$$

We refer to this approach as the "fast gradient sign method" for generating adversarial examples. It is worth noting that the required gradient can be efficiently computed using backpropagation.

3 Description of the model used

In this section we are going to described the architecture of the model we used for our numerical experiments. We built a Neural network composed by: an Input Layer, two hidden layer and an output Layer. In particular within the two hidden layers we used a ReLU activation function and then for the output a Softmax activation function. Finally we computed the Loss function using the categorical cross entropy:

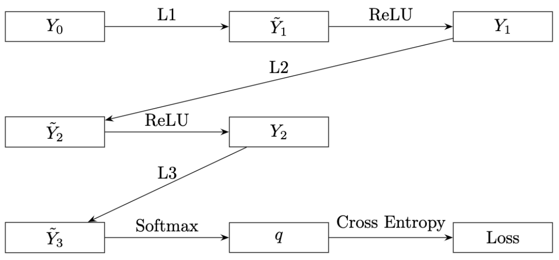


Figure 2: Simplified Diagram of our Neural Network

where L1, L2 and L3 are linear function with weights matrix W1, W2, W3 and biases b1, b2, b3. Let's described briefly these functions and the path from the input layer to the output one:

- **Input**: in the MNIST set an image is represented as e matrix 28x28. This can be a problem for our architecture since it works only with 1d vectors. For that reason we use the numpy reshape function.

- **Creation of Dense Layer**: A function is defined to initialize the weights and biases for a dense layer in the neural network. Weights are initialized with random values drawn from a normal distribution, scaled by the square root of the input size to prevent large weight updates. Biases are initialized to zeros.

- **Activation**: We can see that a ReLu function (returns the maximum of zero and the input value) it has been use for the hidden layers. In fact in the code we substituted it with its approximation: $x \to \frac{x+\sqrt{(x^2+a^2)}}{2}$. It permits to mitigates numerical instability issues for small input values

  The function $x \to \frac{x+\sqrt{(x^2+a^2)}}{2}$ for small values of a is a good approximation for the Rectified Linear Unit (ReLU) function because it behaves similarly. In fact, ReLU returns x if x is positive, and 0 otherwise. When a is small, the term a^2 becomes negligible compared to x^2.

  This approximation is a good approximation of the ReLU function since the following aspects:

  - Derivative: the derivative of this function is: $x \to \frac{1}{2}\left(1 + \frac{x}{\sqrt{x^2+a^2}}\right)$

  - Monocity: this function is always non-decreasing like ReLU and for $x < 0$ the derivative tends to 0 very fast and for x¿0 the derivative tends to 1

  - In x=0 ReLu = 0 and the Approx-function = $\frac{a}{2} \approx 0$ when $|a| \ll 1$.

- **Output Layer**: The softmax function takes a vector of real-valued scores as input and normalizes them into a probability distribution over multiple classes. It ensures that the output probabilities sum up to 1, making them interpretable as class probabilities. It is defined as follow:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}.$$

  In our case we have an output layer of dimension 10 since we have 10 classes (0,..,9).

- **Loss Function**: The loss function used is Categorical Cross-Entropy, which measures the dissimilarity between predicted probabilities and true labels. The loss is computed based on the predicted probabilities and the ground truth labels.

During the forward pass, input data is passed through each layer using matrix multiplication with the corresponding weights and addition of biases. ReLU activation functions are applied after each hidden layer to introduce non-linearity. In order to train our neural network we use the Stochastic Gradient Descent (SGD) in order to update the weights and biases to minimize the loss. SGD is an optimization algorithm commonly used to train neural networks. It iteratively updates the parameters (weights and biases) of the neural network to minimize a predefined loss function. At each iteration, SGD computes the gradient of the loss function with respect to the parameters using a subset of the training data (mini-batch). It then updates the parameters in the direction opposite to the gradient, scaled by a learning rate α, to move towards the minimum of the loss function.

$$W^{t+1} = W^t - \alpha dW$$

$$b^{t+1} = b^t - \alpha db$$

In order to compute the gradient we use backpropagation. It is an algorithm used to compute the gradients of the loss function with respect to the parameters of a neural network. It works by applying the chain rule of calculus to propagate the error backward through the network. During the forward pass, the input data is passed through the network, and intermediate values are computed at each layer. During the backward pass, the gradients of the loss function with respect to the output of each layer are computed recursively using the chain rule. These gradients are then used to update the parameters of the network. To write it formally let denote $\frac{\partial L}{\partial Y}$ as dY.

So we have to compute the following gradient:

$$\begin{aligned}
d\tilde{Y_3} &= q - y \\
dW_3 &= d\tilde{Y_3} \cdot Y_2, \quad db_3 = d\tilde{Y_3} \\
dY_2 &= W_3 \cdot d\tilde{Y_3} \\
d\tilde{Y_2} &= dY_2 \cdot \frac{1}{2}\left(1 + \frac{\tilde{Y_2}}{\sqrt{\tilde{Y_2}^2 + a^2}}\right) \\
dW_2 &= d\tilde{Y_2} \cdot Y_1, \quad db_2 = d\tilde{Y_2} \\
dY_1 &= W_2 \cdot d\tilde{Y_2} \\
d\tilde{Y_1} &= dY_1 \cdot \frac{1}{2}\left(1 + \frac{\tilde{Y_1}}{\sqrt{\tilde{Y_1}^2 + a^2}}\right) \\
dW_1 &= d\tilde{Y_1} \cdot Y_0, \quad db_1 = d\tilde{Y_1}
\end{aligned}$$

where we computed $d\tilde{Y_3}$ as follow: recall that the loss function is

$$L(y) := -\sum_{k=1}^{k=10} y_k log(q_k)$$

and

$$q_k := \frac{e^{y_k}}{\sum_{l=1}^{l=10} e^{y_l}}$$

is the Softmax output. So for every component:

$$\begin{aligned}
(d\tilde{Y_3})_k = \frac{\partial L}{\partial(\tilde{Y_3})_k} &= -y_k \frac{1}{q_k}\left(\frac{e^{y_k}\sum_{l=1}^{l=10} e^{y_l} - e^{2y_k}}{\left(\sum_{l=1}^{l=10} e^{y_l}\right)^2}\right) \\
&= -y_k\left(\frac{\sum_{l=1}^{l=10} e^{y_l} - e^{y_k}}{\sum_{l=1}^{l=10} e^{y_l}}\right) = q_k - y_k
\end{aligned}$$

4 Adversarial Example Generation

As we already had a function that computes all the derivatives with respect to the weights, biases, and output of each layer (before and after activation), computing an adversarial example becomes quite easy.

Using the same notation as above, an adversarial example with respect to image x can be denoted by:

$$\text{ADV}_x = x + \epsilon \cdot \text{sign}\left(\nabla_x J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right).$$

We can also denote the right-hand side by an "ϵ-perturbation". And the gradient sign $(\nabla_x J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}))$ can be expressed as

$$\frac{\partial L}{\partial Y_0} = dY_0 = W_1 \cdot d\tilde{Y}_1.$$

As an example, we created a 0.01-perturbation of the 7th image in the test set, which results in the following:

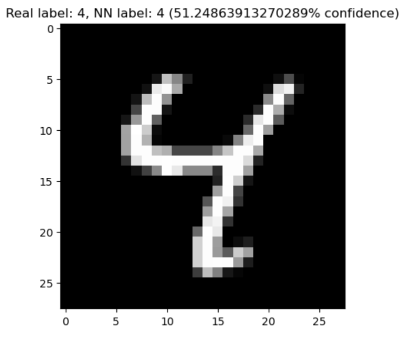


Figure 3: Default correct classification

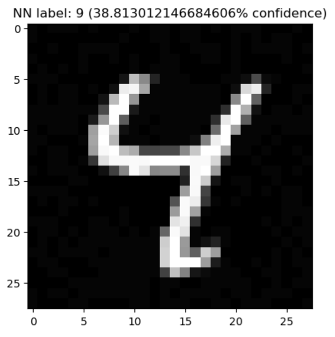


Figure 4: Perturbed misclassification

For a scheme of the generation of adversarial examples, refer to Figure 5.

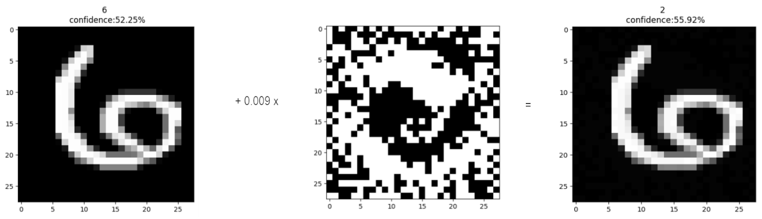


Figure 5: Demonstration with $\epsilon = 0.009$ of how adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input can change the classification

5 Testing

5.1 Evaluating the accuracy of our neural network

Across this section, the set of images we used for testing and analysing the behaviour of adversarial examples was the first 100 images of the MNIST's test set introduced above.

With one training epoch and 64 neurons in both the first and the second hidden layer, our model shows an accuracy of 87.32%.

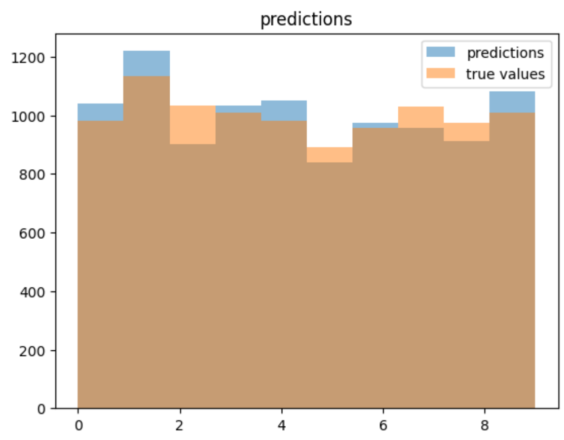


Figure 6: Predictions with 64, 64 neurons in the hidden layers

For example, with $\epsilon = 0.009$, we get the misclassified images in Figure 7. In the upper row one can see the original image with its assigned label and confidence in labelling; in the lower row the perturbed image with its assigned label and confidence in labelling.

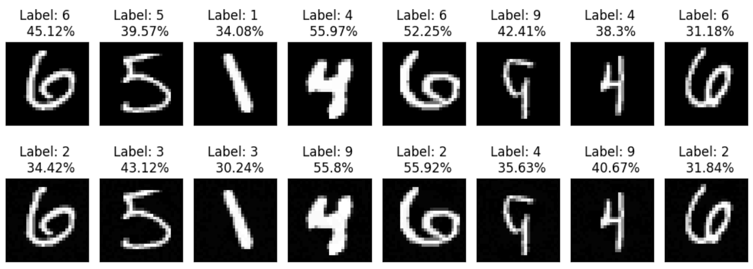
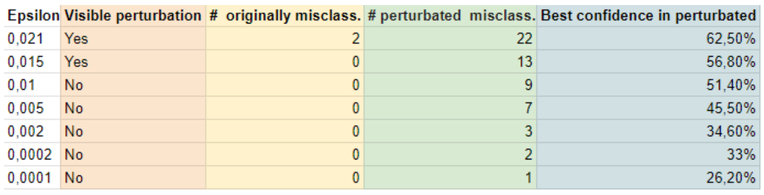


Figure 7: Misclassifications with $\epsilon = 0.009$

We make some experiments to see how the misclassified images vary when ϵ varies.



Epsilon	Visible perturbation	# originally misclass.	# perturbated misclass.	Best confidence in perturbated
0,021	Yes	2	22	62,50%
0,015	Yes	0	13	56,80%
0,01	No	0	9	51,40%
0,005	No	0	7	45,50%
0,002	No	0	3	34,60%
0,0002	No	0	2	33%
0,0001	No	0	1	26,20%

Figure 8: Misclassified images and confidence with 64, 64 neurons in the hidden layers

We notice that the perturbation in our adversarial examples is visible to human eye starting from epsilon above 0.01. Compare e.g., in Figure 5.1, an example of original image (center) with its 0.015-perturbed image (left) and its 0.01-perturbed image (right).

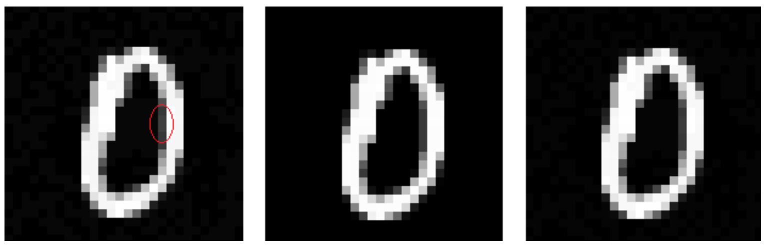
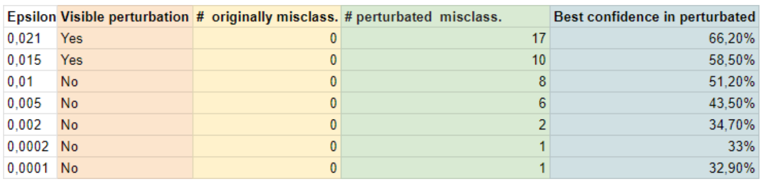


Figure 9: Comparison between original image (center) and its 0.015 and 0.01 perturbations

Moreover, the number of misclassified perturbed images and number of the ones also originally misclassified between them increase as epsilon increases. Also the best confidence in classification of our perturbed images increases as epsilon increases.

5.2 Classification's accuracy

We now want to improve the model's accuracy. We first try increasing the number of neurons in the hidden layers. See for example the case with 128, 128 neurons in Figure 10. We get an increase in accuracy: no original image is now misclassified and we observe less misclassified perturbed images and an overall better confidence in labeling the perturbed images.



Epsilon	Visible perturbation	# originally misclass.	# perturbated misclass.	Best confidence in perturbated
0,021	Yes	0	17	66,20%
0,015	Yes	0	10	58,50%
0,01	No	0	8	51,20%
0,005	No	0	6	43,50%
0,002	No	0	2	34,70%
0,0002	No	0	1	33%
0,0001	No	0	1	32,90%

Figure 10: Misclassified images and confidence with 128, 128 neurons in the hidden layers

It must be noticed, though, that the increase in accuracy is very small. For example we get accuracy 87.94% with 128, 128 neurons and accuracy 88.92% with 512, 512 neurons. Increasing the accuracy noticeably in this way would hence require a greater number of neurons and therefore a large computational time.

We then try adding epochs, to have bigger improvements in accuracy in a reasonable time. We look at the case with 64, 64 neurons and $\epsilon = 0.01$, i.e. the biggest ϵ for which the perturbations are imperceptible to human eye. With a single epoch we had 87.54% accuracy and 9 perturbed images misclassified. By increasing the number of epochs, instead, we have

- 3 epochs: accuracy 91.43% and 6 perturbed images misclassified,
- 7 epochs: accuracy 94.16% and 4 perturbed images misclassified,
- 12 epochs: accuracy 96.13% and 3 perturbed images misclassified
- 40 epochs: accuracy 97.27% and 4 perturbed images misclassified.

Having reached a much greater accuracy (96.13%), we showed that adding epochs leads to higher accuracy in less time. As we can observe in Figure 11 that as the increase becomes very small as the number of epochs becomes large, we start seeing an horizontal asymptote at 97% c.a.

5.2.1 Investigating perturbed images

We now return to the $784 \times 64 \times 64 \times 10$ architecture, 1 epoch, and we will now use all the 10000 images of the MNIST test set introduced above.

We can see that there are 1268 images (out of the 10000) in the test set that have a 0-perturbation misclassification. This is of course expected - it matches the 87.32% accuracy mentioned above.

We can now pose the following questions:

- For a fixed image, what is the smallest value of ϵ such that there's a ϵ-perturbation that is misclassified? We will denote this value by ϵ_{min}. In the case of an image that is misclassified by default, $\epsilon_{min} = 0$.

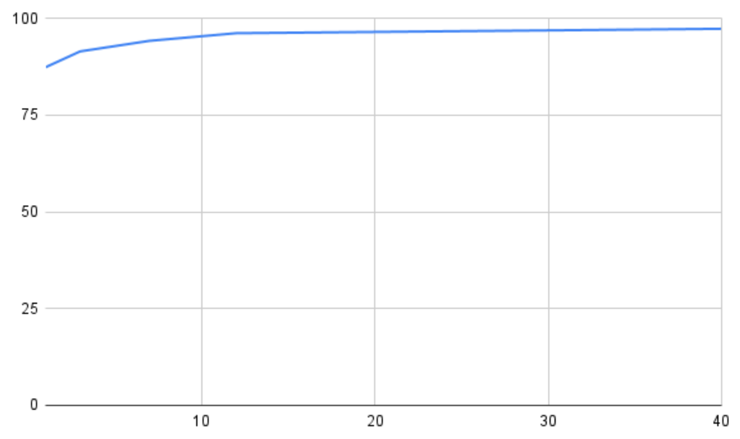


Figure 11: Change in accuracy during the learning

- For a fixed ϵ, how many of the images in the test set have a ϵ-perturbation that is misclassified?

As we'll see, the behaviour of ϵ-perturbations has a very nice property which will link these two questions into one.

First, we define a function that for each image computes its ϵ_{min} in a linear fashion, by testing an ϵ-perturbation for every value of ϵ from 0 to 1 in small increments of size *step*. We can then notice that every image in the test set is either misclassified by default (i.e. is a 0-perturbation), or has an adversarial example with $\epsilon < 1$ (in fact, the largest ϵ_{min} over the test set is approximately 0.165).

Among the images that are correctly classified by default, one natural hypothesis that we can pose is the following: *For a fixed image, is the set of* $\epsilon \in [0, 1]$ *such that an* ϵ*-perturbation of that image is misclassified a subinterval?*

This would mean that for $\epsilon \in [0, \epsilon_{min}]$ the corresponding ϵ-perturbation is correctly classified, and for $\epsilon \in]\epsilon_{min}, 1]$ it is incorrectly classified.

We tested this hypothesis by creating a function that for each image computes which values of $\epsilon \in [0, \text{step}, 2 \cdot \text{step}, \cdots, 1]$ produce an ϵ-perturbation. If the resulting values form a "discrete" interval (*i.e.* a set of the form $\{0, \text{step}, 2 \cdot \text{step}, \cdots, k \cdot \text{step}\}$) we say that the image confirms the hypothesis. Perhaps unsurprisingly, every image in the test set confirmed the hypothesis, so we'll assume it's true from now on.

The truthfulness of the hypothesis leads us to two interesting conclusions:

- We can answer the second question posed above by answering the first - if for each image we know its ϵ_{min}, for a fixed ϵ we can simply count how many images have a corresponding $\epsilon_{min} \leq \epsilon$.
- Instead of using a linear algorithm to find ϵ_{min} for each image (which runs in $O(\frac{1}{\text{step}})$), we can use a binary search algorithm that finds it in a much faster $O(\log \frac{1}{\text{precision}})$.

We then use this binary search method to compute ϵ_{min} for each image in the test set. This allows us to produce various interesting facts about ϵ-perturbations:

- Assuming 0.01 is the ϵ threshold after which an ϵ-perturbation starts being perceptible to the human eye, we can see that 7.82% of images in the test set have an imperceptible adversarial example.

- The distribution of all ϵ_{min} has a mean of 0.0479 and a standard deviation of 0.0286. The distribution looks like this:

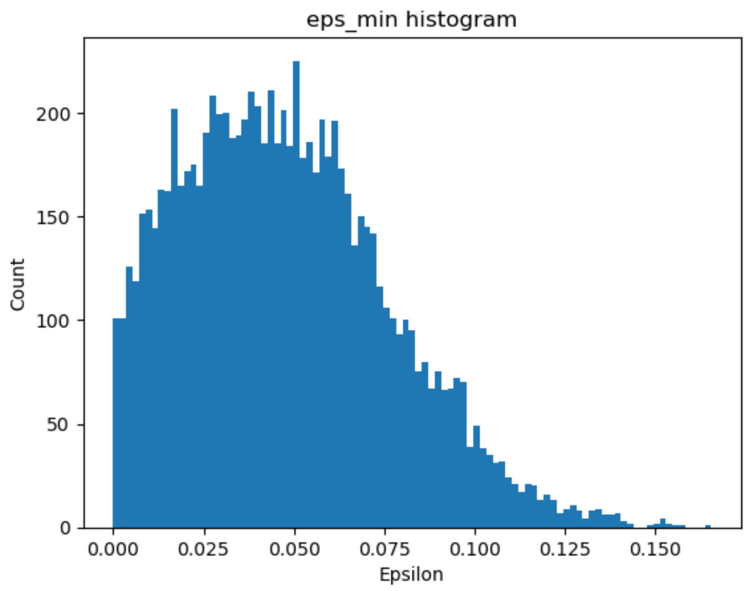


Figure 12: ϵ_{min} distribution

- For each threshold ϵ we can easily compute what the percentage of images with a misclassified ϵ-perturbation is, which is the same as computing the cumulative distribution function for ϵ_{min}. Here it is:

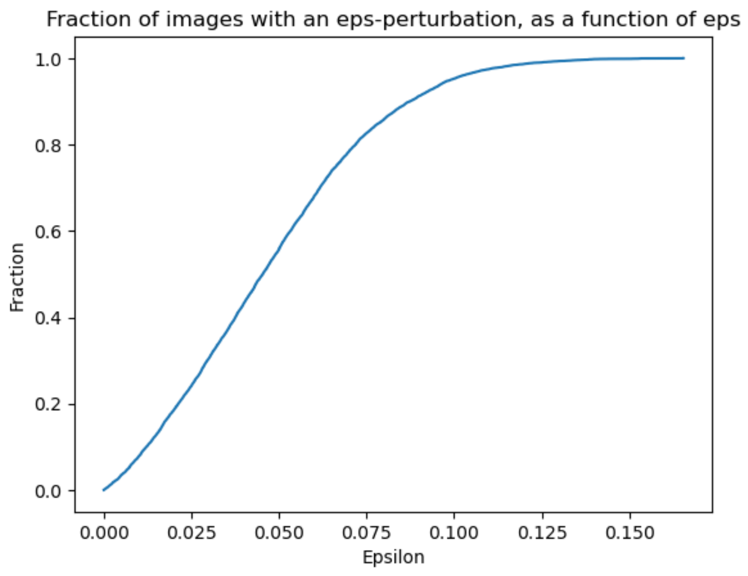


Figure 13: ϵ_{min} CDF

5.2.2 Addendum

We noticed that the ϵ_{min} distribution looks a bit like a truncated normal. Since it is non-trivial to fit a truncated normal, we experimented a little bit and found that the following parameters seemed pretty good: $a = 0$, $b = 0.166$, $\mu = 0.042$, $\sigma = 0.03$. The resulting PDF and CDF look like this:

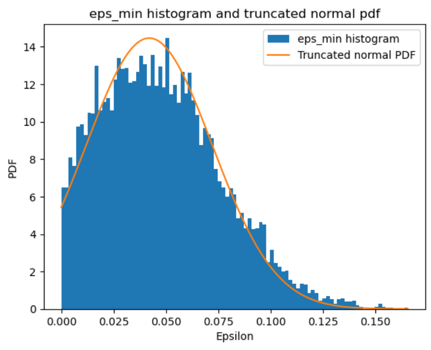


Figure 14: Truncated normal PDF and ϵ_{min} distribution

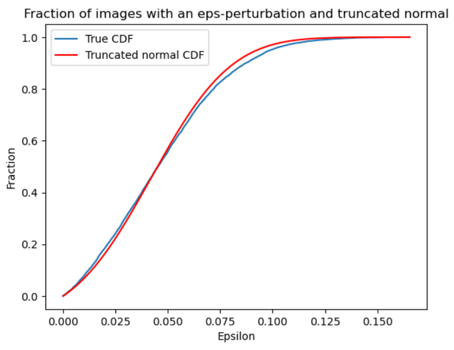


Figure 15: Truncated normal CDF and ϵ_{min} CDF

5.2.3 Confidence as a function of ϵ

To understand how different values of ϵ affect the classification confidence, we created a function that plots both the confidence of the network that an ϵ-perturbation gives the correct digit, and its confidence that it is an incorrect digit (we take the maximum confidence among all digits different from the correct one) as a function of ϵ. We include a couple of examples below:

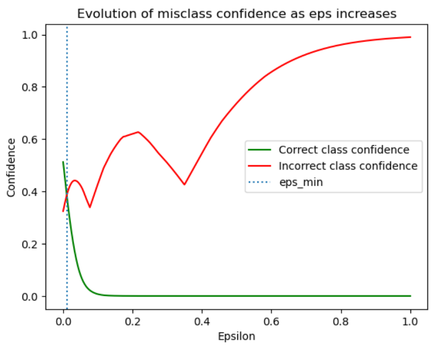


Figure 16: 7th image in the test set

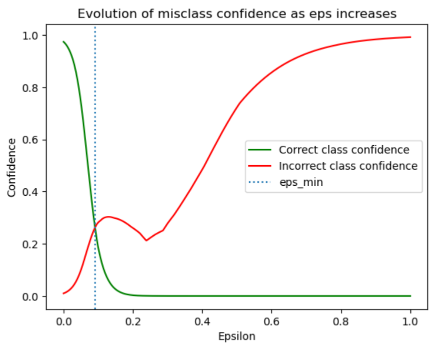


Figure 17: 315th image in the test set

5.3 Adversarial training

We created a function that for a given ϵ extracts from a data set all of its ϵ-perturbations which are misclassified. We then added all of these adversarial examples to the original training set and trained a new network on the new set.

The results were a bit lackluster: the network's accuracy stayed the same, while the number of adversarial examples produced from the test set (at the $\epsilon = 0.01$ level) reduced from 7.82% to 7.40%.

In the future it would be interesting to improve this analysis by trying different values of ϵ or increasing the number of epochs for example.

6 Conclusion

Our neural network suffers from adversarial examples with the MNIST dataset. This is due to linearity for how the adversarial examples have been built.

When epsilon increases, the number of misclassified images increases, as well as the confidence in classification of the perturbed images.

We important result we get is that for $\epsilon > 0.165$ every image is originally misclassified or is misclassified after perturbation. For ϵ above 0.01, though, the perturbation is visible to human eye.

Improving accuracy, the neural network fiercely resists the adversarial examples. Accuracy, rather than modifying the network's architecture, can be noticeably enhanced by adding training epochs. In this way, we enhanced accuracy of 10% c.a. and start seeing an horizontal asymptote at 97% c.a.

As for our investigation into how different values of ϵ affect adversarial examples, we see that there's a sharp increase in the number of images with an ϵ-adversarial example while $0 \leq \epsilon \leq 0.1$. After this threshold we can see that almost all (over 95%) of the images already have one such example.

We can also see that confidence in the correct class decreases very sharply as ϵ increases, and that after $\epsilon \geq 0.1$ it is very close to 0. This of course is expected as it matches the previous conclusion. We can also observe that the incorrect classification changes as ϵ increases. For example, in figure 16 there is a discontinuity in the derivative of the red curve in multiple points (the most noticeable at $\epsilon \approx 0.37$). This happens when there's a new (incorrect) class that takes over as the most confidently classified by the neural network.

7 Further Works

As first extension it is possible to analyse the structure of the most misclassified images during the experiment and try to find their common features. We saw that our Neural Network is exposed to adversarial examples. A further works is to implement Adversarial Training in a more robust way. Adversarial training is a technique used in the field of machine learning, particularly in the context of training neural networks, to improve the robustness of the model against adversarial attacks. The model is trained not only on the original training data but also on perturbed versions of the data.
Another aspect that could be taken into consideration is the implementation of other architecture of the neural network in order to see which model is more robust to adversarial example. We expect that RBF network could be better than the linear model presented in this paper under this point of view.

References

[1] I. J. Goodfellow, S. Jonathon, and C. Szegedy, *Explaining and Harnessing Adversarial Examples*, 2015.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going Deeper with Convolutions*. Cornell University Library, arXiv.org, 2015.