AI Safety: A survey

Introduction

Neural networks are being recently deployed in safety critical domains, such as healthcare, automated driving, and more. Although these models have exhibited very high accuracy, there are many difficulties in understanding their behavior, due to the millions of parameters and their complicated structure (non – linear, non – convex). Therefore, a safe behavior (in an engineering sense, such as automatic control systems, etc.) cannot be guaranteed. Apart from that, neural networks show occasionally bizarre, unexpected failures, such as in the case of adversarial examples.

Despite these limitations, neural networks are as of today the most successful AI methodology, having solved a great number of problems, such as visual or speech recognition, that remained unsolvable for decades. Hence, researchers have been recently trying to address the safety limitations mentioned above, introducing the area of safe and robust AI. In these notes, we will try to analyze the main ideas and results in AI safety as of today.

Overview

The area of AI safety is divided into several categories. Some main categories are given below [1-4]:

- Adversarial training and defense.
- Formal methods on neural networks.
- Neural network robustness verification.
- System run-time monitoring.
- Testing.
- Bayesian learning.
- Interpretability - Explainability.
- Safety in Reinforcement Learning.

Adversarial Training

Adversarial Examples

A major source of concern about (deep) neural networks has arisen by the discovery of the so – called “adversarial examples” [5]. These are samples (usually images) that are altered by small modifications that look similar to noise. Although the altered sample seems identical to a human observer as before, a neural network miss-classifies it with high confidence. An example of an adversarial example can be seen below:

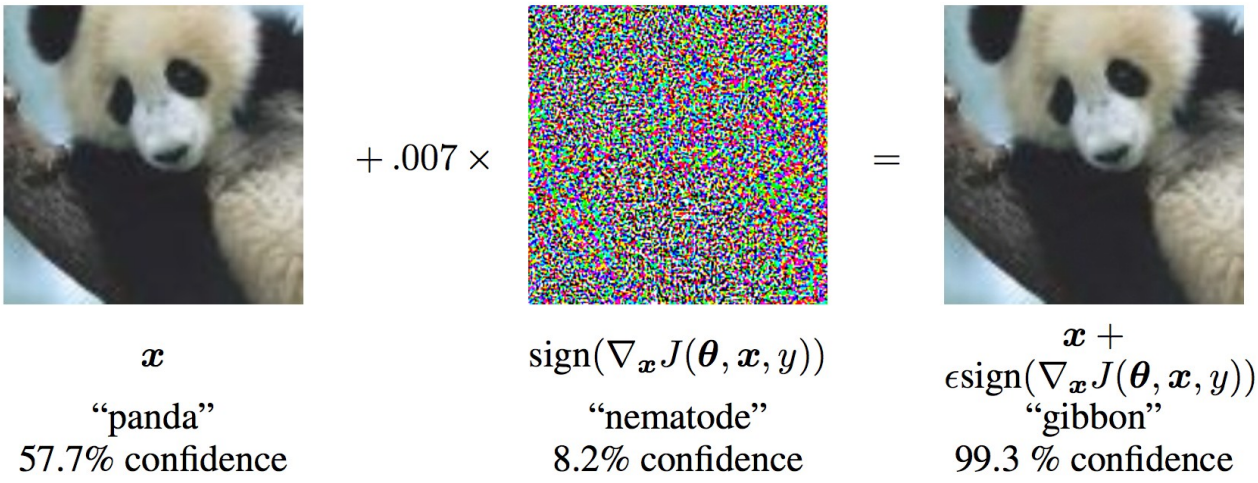

Fig 3.1: An adversarial example (from [5]).

Since then, many methods to produce adversarial examples have been discovered, which can be separated in two larger categories: white box attacks, where the targeted network architecture and parameters are known and used by the attacking algorithm, or black box attacks, where the underlying network is assumed unknown, and only its outputs are accessed. There have been even cases that changing one single pixel alters the network's decision [6].

Fig 3.2: Changing a single pixel changes the classification of the model (from [6]).

Similar concerns arise in other safety critical application domains of AI, such as automated medical diagnosis. Below, we can see an example where an adversarial perturbation changes the output of a standard medical diagnosis network from Benign to Malignant:

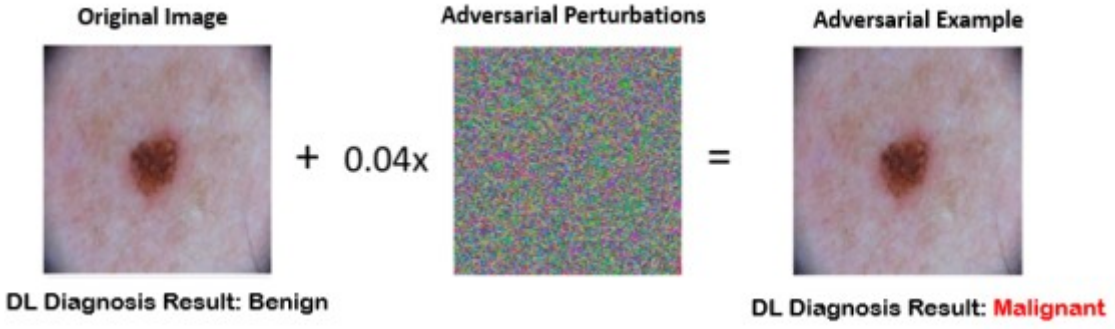

Fig. 3.3: An adversarial perturbation can change the diagnosis of a medical Deep Learning system (from [24]).

Apart from these, it should be noted that adversarial examples can also be produced by geometrical transformation, for example by rotating an image ([7]). Furthermore, it turns out that adversarial examples are to some degree universal across different deep learning models, meaning that samples designed against a given network are more likely to fool another ([5]).

We should also note that adversarial examples can occur on any type of data, not just images (although this is the usual case). We can see an example of adversarial examples in Natural Language Processing below:

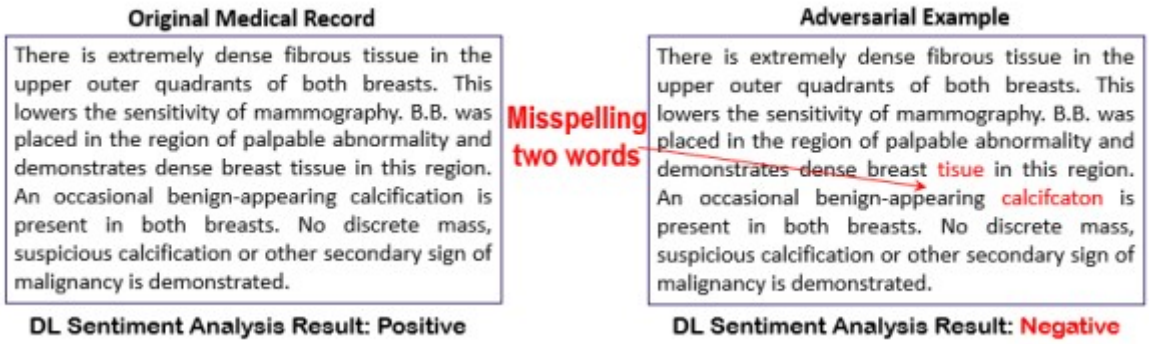

Fig. 3.4: An adversarial example in NLP: by miss-spelling two words, the sentiment analysis network changes its output (from [25]).

In general, let $\boldsymbol{x}$ a data point input to a neural network. Then, an adversarial example is a small perturbation of this input, $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \delta\boldsymbol{x}$, such that the network changes its classification output, while a human oracle will identify $\tilde{\boldsymbol{x}}$ as being in the same class as $\boldsymbol{x}$. $\boldsymbol{x}$ can be any type of data.

To tackle the problem of adversarial examples, methods of adversarial training of neural networks have been developed. The most standard consist of creating adversarial examples for a model, adding them into the training set, and retraining it. Repeating this process some times, we hope that the final trained model will be more resistant against adversarial examples. More advanced methods have also been developed, as for example in [8], where models are trained with a method called projected gradient descend, casting the problem into a robust optimization setting. We will review some of these methods next.

Adversarial Attacks

First, let us briefly summarize the various adversarial attacks in the literature, that is, the methods to create adversarial examples, given some neural network. These can be separated in two categories, **white box attacks** and **black box attacks**.

White box attacks assume full knowledge of the attacked network, namely its architecture and parameters, and perhaps also of the training method. The attack then utilizes this information to identify vulnerabilities of the network and produce adversarial samples.

On the other hand, black box attacks assume no knowledge of the network, and have access only to its outputs. They then examine the behavior of the network on some carefully crafted inputs, and from this they extract information used to create adversarial samples. These are called **adaptive** black box attacks. In contrary, **non – adaptive** attacks assume knowledge of the training data distribution used to produce the network, and are not able to query it at will. They then usually use these data to build a surrogate model and use some white box method to find adversarial samples on the new network. This process exploits the universality of adversarial examples across different

models, as observed in [5]. Another type is **strict** black box attacks, which have access to pairs $(\boldsymbol{x}, y)$ of network inputs and outputs, but they are not allowed to insert their own inputs into the network.

Adversarial attacks can be viewed as solving the following problem: Given an input $\boldsymbol{x}$ we want to find a small perturbation $\delta \boldsymbol{x}$ such that the network output changes. That it, we need to solve the problem:

$$\widetilde{\boldsymbol{x}} = \boldsymbol{x} + min_{\delta x}\{\|\delta \boldsymbol{x}\|_p : f(\boldsymbol{x} + \delta \boldsymbol{x}) \neq f(\boldsymbol{x})\} ,$$

where f is our neural network model, and $\|\cdot\|_p = L_p$ is a p-norm, measuring the size of the perturbation. By definition, we have:

$$L_p(\boldsymbol{x}) = (\sum_i x_i^p)^{1/p}$$

For example, for $p=2$ we have the familiar L_2 or Euclidean norm. Moreover, L_0 and L_∞ norms are also defined, in the following way:

$$L_0(\boldsymbol{x}) = \sum_i 1(x_i > 0)$$

$$L_\infty = max_i |x_i|$$

That is, L_0 is the number of non – zero components in $\boldsymbol{x}$, while L_∞ is the largest absolute value of $\boldsymbol{x}$'s components.

White – box methods solve the above problem by using optimization methods, exploiting their access to network gradients (since they posses full knowledge of the model and its architecture / parameters). Black box attacks try to solve the same problem by heuristic search methods. These are summarized in the figure below:

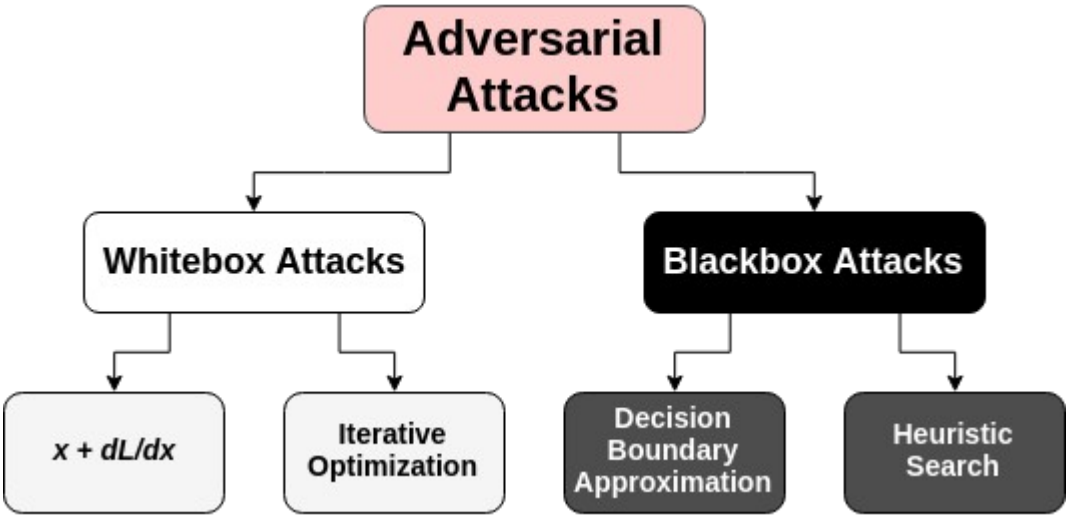

Fig. 3.5: Overview of white – box and black – box attacks.

White – box attacks: Below, we summarize some well - known white – box attacks from the recent literature. As suggested form the figure above, these can be divided in two categories.

Adversarial gradient – based attacks: These attacks perturb the input in such a way that the loss function of our network is changed to the maximum amount. That is, by taking the gradient of our loss function with respect to the input (and not with respect to weights as in the usual back – propagation), the attacker tries to determine the direction that maximally changes the loss and then perturbs the input a small amount towards that direction. Since that direction maximizes the change of loss, we hope that this perturbation will make the model miss-classify the input, while keeping the change made minimum. Some of these attacks include the following.

- **Fast Gradient Sign Method (FGSM)**: This method searches for a direction in which the loss function increases fastest, and then updates the input towards this direction. This is similar to a form of gradient descent on the input, rather than the network parameters. The attack formulation is as follows:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} + \epsilon \cdot sign(\nabla_x J(\boldsymbol{x}, y)),$$

  Here, ϵ is a small positive constant, and $J(\boldsymbol{x}, y)$ is the network loss function. This method does not guarantee that the samples found will be similar to their real counterparts. Practically, one needs to make a trade-off between small perturbations that are similar to the input, while the model still miss-classifies its input.

- **Basic Iterative Method (BIM)**: BIM (of Iterative FGSM, [28]) is an extension of FGSM, where we perform FGSM multiple times with a small step size, updating the input at every step. The method can be formulated as:

$$\tilde{\boldsymbol{x}}_0 = \boldsymbol{x}, \tilde{\boldsymbol{x}_{n+1}} = clip_{\boldsymbol{x},\epsilon}\{\tilde{\boldsymbol{x}}_n + a \cdot sign(\nabla_x J(\boldsymbol{x}, y))\},$$

  where n is the iteration step, a a small positive constant, and the clipping function ensures that the updates remain limited in size, that is, they stay in the $[\boldsymbol{x} - \epsilon \cdot \mathbf{1}, \boldsymbol{x} + \epsilon \cdot \mathbf{1}]$ ball (as well as in the $[0, 255]$ image pixel range in the case of images).

- **Random FGSM (R+FGSM):** This method is again similar to FGSM, enchanched by adding random perturbations, sampled from a Gaussian distribution, to the input. Specifically, the formula reads:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x}' + (\epsilon - \alpha) \cdot sign(\nabla_{x'} J(\boldsymbol{x}', y)),$$

  where $\boldsymbol{x}' = \boldsymbol{x} + \alpha \cdot sign(N(\mathbf{0}^d, \boldsymbol{I}^d))$ is the perturbed input (note that the gradient is computed at $\boldsymbol{x}'$). The motivation behind this perturbation is to counter the defensive technique of **gradient masking** [30], which we will analyze later. The idea behind gradient masking is to obscure or hide the model's gradients, so that an attacker cannot calculate the exact value. By adding random noise, this obscurification effect is countered, because the attack does not rely on the exact gradient value.

- **Jacobian-based Saliency Map Attack (JSMA)**: This attack is based on designing an saliency adversarial map, called Jabobian – based Saliency Map Attack. Given an input $\boldsymbol{x}$, this is the Jacobian matrix with respect to $\boldsymbol{x}$, namely:

$$J_F(\boldsymbol{x}) = \frac{\partial F(\boldsymbol{x})}{\partial \boldsymbol{x}} = \left[\frac{\partial F_j(x)}{\partial x_i}\right]_{i \times j},$$

where F devote the class logits (second-to-last layer before the output, i.e. the layer before the softmax). This map shows us the input components in $\boldsymbol{x}$ that cause the most significant changes to the output. They then designed a perturbation that targets these more influential features, thus managing to fool the network by changing only a small portion of the input. In their experiments, they managed to achieve 97% adversarial success rate by modifying only 4.02% of the input features per sample. A disadvantage of this approach though is that it comes with high computational cost, since we have to compute the Jacobian, namely a partial derivative of each class logit with respect to every input feature (e.g. pixel).

- **DeepFool:** The authors of DeepFool proposed a method to determine approximately the closest distance from the input to the decision boundary of a classier. Then we can add a perturbation corresponding to this minimum distance, the decision boundary will be crossed, and the classification output will change. This idea can be used then to generate adversarial examples by minimal perturbation.

  They first started with an affine binary classifier of the form $f(\boldsymbol{x})=\boldsymbol{w}^{\boldsymbol{T}}\cdot\boldsymbol{x}+b$. $f(\boldsymbol{x})>0$ represents the first class, and $f(\boldsymbol{x})<0$ the second, thus $f(\boldsymbol{x})=0$ is our decision boundary. Given an input $\boldsymbol{x_0}$, we can prove (by analytical geometry) that the minimal perturbation to reach the decision boundary is:

$$r_*(\boldsymbol{x_0})=\frac{-f(\boldsymbol{x_0})}{\|\boldsymbol{w}\|_2^2}\boldsymbol{w}$$

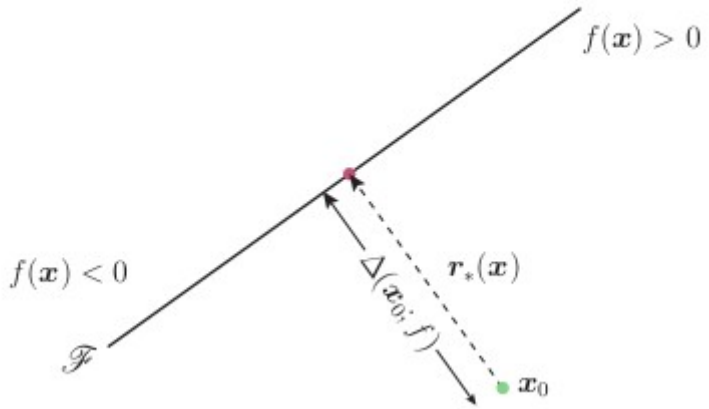

Fig. 3.6: Minimum perturbation to reach the decision boundary of a linear classifier.

Next, the authors considered the case of a binary non – linear classifier in the form of a neural network, $f:\mathbb{R}^n\rightarrow\mathbb{R}$. Their idea was to use the result of the linear case by replacing the network by it's linear approximation around $\boldsymbol{x_0}: f(\boldsymbol{x})\approx f(\boldsymbol{x_0})+\nabla f(\boldsymbol{x_0})^T\cdot(\boldsymbol{x}-\boldsymbol{x_0})$. Now, we can find the minimal perturbation to reach the boundary by using the linear case formula above, with $\boldsymbol{w}=\nabla f(\boldsymbol{x_0}), b=f(\boldsymbol{x_0})-\nabla f(\boldsymbol{x_0})^T\cdot\boldsymbol{x_0}$. The authors repeated this process in an iterative fashion, by repeatedly linearizing the model, perturbing the input, linearizing the model at the new point, etc., until the sign of $f(\boldsymbol{x})$ becomes different from the sign of $f(\boldsymbol{x_0})$, and the classification has changed.

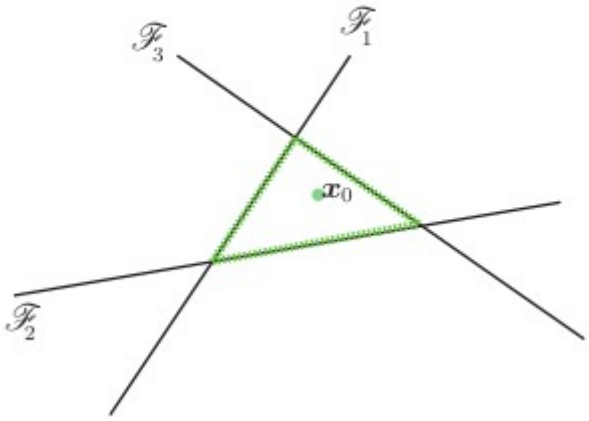

Fig. 3.7: Multiple decision boundaries in the case of a linear multi-class classifier.

Finally, they extended this approach to the multi-class case. They considered a multi-class linear classifier of the form $f(\boldsymbol{x})=W\boldsymbol{x}+\boldsymbol{b}$, where the selected class corresponds to the highest score, $\hat{k}(\boldsymbol{x_0})=argmax_k f(\boldsymbol{x_0})$. We want to determine the minimum perturbation that changes the class from $\hat{k}(\boldsymbol{x_0})$ to some other class. For each class k, its decision boundary over $\hat{k}(\boldsymbol{x_0})$, $\{\boldsymbol{x}, s.t: f(\boldsymbol{x})_k \geq f(\boldsymbol{x})_{\hat{k}(\boldsymbol{x_0})}\}$, is again a hyper-plane. So, we want to find the minimum distance of $\boldsymbol{x_0}$ to one of those hyper-planes. The authors gave one formula to do that, and they extended their iterative algorithm to this case as well, by replacing the network with its linear approximation, determining the perturbation, and repeating, until the class has changed.

DeepFool provided less perturbations than FGSM, and managed to reduce the insensitivity of the perturbations compared to JSMA (which reduces the number of features selected to perturb).

Adversarial attacks based on optimization of surrogate objective functions: This is the second class of white – box attacks, as suggested by the diagram. In contrary to the first category, these attacks define adversarial attacks as an optimization problem, that is, we define an optimization problem that an adversarial example should satisfy, and then solve it. This allows us to possibly add further constrains into this objective function, which may increase the quality of adversarial examples produced, or some other metric. This comes in contrary to the first category of methods, where we have no control of the examples produced. Some of those methods are the following.

- **L-BFGS Attack:** This attack method was suggested in [5], where adversarial examples on neural networks were observed. There, the authors defined adversarial attacks as perturbations being very close to the input (in the sense of some norm, usually the Euclidean L_2 norm), but cause the network to miss-classify them. That is, we want to solve the following problem:

$$min_{\boldsymbol{r}} \|\boldsymbol{r}\|_2, \text{ subject to:}$$
$$1.\ f(\boldsymbol{x}+\boldsymbol{r})=l$$
$$2.\ \boldsymbol{x}+\boldsymbol{r}\in[0,1]^m$$

This means, we want to find the perturbation r with the minimum (Euclidean) norm, which satisfies two conditions: First, the model classifies the perturbed input as belonging to class

l, and second, the distortion must remain in certain bounds (here $[0,1]$ for all dimensions, this was selected in the case of images).

The authors replace the above objectives, which lead to a hard optimization problem, with the following easier, approximate problem:

$$min_{\boldsymbol{r}} c\|\boldsymbol{r}\|_2 + loss_f(\boldsymbol{x}+\boldsymbol{r}, l), s.t: \boldsymbol{x}+\boldsymbol{r} \in [0,1]^d$$

That is, they first solve the above problem for a certain $c>0$, and then they perform a line search to find the smallest c such that the minimizer $\boldsymbol{r}$ of the above problem still satisfies the condition $f(\boldsymbol{x}+\boldsymbol{r})=l$. In case of convex problems, one can show that the modified problem yields the same solution to the original one. In our case, neural networks are non – convex, so this problem gives us an approximate solution, which turns out to be good enough. The authors used the Limited – memory Broyden – Fletcher – Goldfarb – Shanno (L-BFGS) numerical optimization algorithm to solve the problem above, hence the attack's name.

The loss function indicated above was the usual cross – entropy loss in the paper, but it can be replaced with other more suitable surrogate loss functions, depending on the problem conditions.

- **Carlini & Wanger Attack (C&W):** Carlini and Wagner ([31]) extended the L-BFGS attack by modifying the loss function, by replacing the cross – entropy loss with the following loss function:

$$L(\tilde{\boldsymbol{x}}) = max(max(Z(\tilde{\boldsymbol{x}})_i : i \neq t) - Z(\tilde{\boldsymbol{x}})_t - \kappa)$$

Here $Z(\boldsymbol{x})$ denote the logits of the neural network, that is, the class scores before the final softmax layer, which turns them into proper probabilities. This function has the following interpretation: we want to maximize the distance between our target class t and the most likely class i. That is, given an adversarial input $\tilde{\boldsymbol{x}}$, we compute the class scores of the target class $Z(\tilde{\boldsymbol{x}})_t$, as well as the score of the most likely class, that is the highest logit value $Z(\tilde{\boldsymbol{x}})_i$, other than t. If we succeed in "fooling" the classifier, then $Z(\tilde{\boldsymbol{x}})_t$ will be larger that the $Z(\tilde{\boldsymbol{x}})_i$ of the second best class, and our objective function will attain it's minimum value (negative), and the problem is solved. Furthermore, κ is a positive constant controlling the confidence of adversarial examples produced. It does this by enforcing the scores difference between the target and the other classes to be larger.

Then, the authors replace the L-BFGS minimization problem with the following, in case of the L_2 attack:

$$\left\|\frac{1}{2}(\tanh(\boldsymbol{w})+1) - \boldsymbol{x}\right\|_2^2 + c \cdot f\left(\frac{1}{2}\tanh(\boldsymbol{w})+1\right),$$

where we minimize over variable $\boldsymbol{w}$. The idea here is to apply change of variables and set for the perturbation $\boldsymbol{r}$:

$$\boldsymbol{r} = \frac{1}{2}\tanh(\boldsymbol{w}) + \boldsymbol{1} - \boldsymbol{x}$$

Since $-1 \leq \tanh(w_i) \leq 1$ for every $w_i \in \mathbb{R}$, we have $r_i + x_i = \frac{1}{2}\tanh(w_i) + 1 \in [0,1]$, that is, the 2nd condition of L-BFGS is automatically satisfied, and this is helpful for the optimization process (making the problem easier, and allowing us to replace box – constrained optimization with standard gradient – based methods, which are generally faster).

Apart from that, Carlini and Wagner proposed attacks for the cases of L_0 and L_∞ norms. Specifically, for the L_0 norm, since it is not differentiable, the attack was performed iteratively. At each step, the importance of pixels is determined by using the gradient of the L_2 norm. Then, some of the least important pixels are removed, and the process repeats until the remaining pixels cannot generate an adversarial example (remember that the L_0 norm is just the number of pixels that are perturbed).

For the L_∞ norm, the authors replaced the function to be minimized with the following:

$$c \cdot g(\boldsymbol{x} + \boldsymbol{r}) + \sum_i [(r_i - \tau)^+]$$

For each iteration, they reduced τ by a factor of 0.9, if $r_i < \tau$ for all i. τ was considered as an estimation of the norm L_∞ (remember that the L_∞ norm is just the maximum absolute value of a vector's components).

- **Adversarial Transformation Network (ATN):** In this attack, authors use another neural network to either generate adversarial examples that are similar to the input data (Adversarial Auto-encoding), or adversarial perturbations that will create adversarial examples when added to the original input (perturbation ATN, P-ATN). This generator network achieves these objectives by trying to minimize the norm between generated and valid samples (similarity loss), as well as the classification loss between the model's prediction and the fake targets. This means that the generator can be used to create adversarial examples only for a certain target class, and we need to train different ATN's for different classes.

  Specifically, the loss being minimized is the following:

$$argmin_{\boldsymbol{\theta}} \sum_{\boldsymbol{x}_i \in X} \beta L_X(g_{f,\boldsymbol{\theta}}(\boldsymbol{x}_i)) + L_Y(f(g_{f,\boldsymbol{\theta}}(\boldsymbol{x}_i)), f(\boldsymbol{x}_i)),$$

  where $f : \boldsymbol{x}_i \in X \rightarrow y_i \in Y$ is a given neural network, and $g_{f,\boldsymbol{\theta}} : \boldsymbol{x} \in X \rightarrow \tilde{\boldsymbol{x}}$ is the generator network with parameters $\boldsymbol{\theta}$ that we want to learn. The loss indicated above consists of two terms: L_X is a similarity loss (for example the Euclidean norm) and L_Y is the loss of the class outputs, defined as $L_{Y,t}(\boldsymbol{y}', \boldsymbol{y}) = L_2(\boldsymbol{y}', r(\boldsymbol{y}, t))$, where $r(\cdot)$ is a re-ranking function that modifies $\boldsymbol{y}$ such that $y_k < y_t, \forall k \neq t$ (t being the target class), and β is a weight controlling the influence of the two terms.

  In this work, authors chose

$$r_a((\boldsymbol{y}), t) = norm\left(\left\{\begin{matrix} a \cdot max\, \boldsymbol{y}, if\, k = t \\ y_k, else \end{matrix}\right\}_{k \in \boldsymbol{y}}\right),$$

where $a>1$ is a parameter specifying how much larger y_t should be than the current max classification, and $norm(\cdot)$ is a normalization function that re-scales its inputs to be valid probabilities.

Essentially, what this ranking function does is to create a classification (logits) vector where the target class has the maximum value, which is what our generator must achieve. This re-ranking method used by the authors ensures besides that that $r(\boldsymbol{y},t)\sim\boldsymbol{y}$, that is, apart from the target class, the rest of the components retain their ranking, and the target vector produced is similar to the original losses. This helps the optimization method and avoids large distortions when computing $\widetilde{\boldsymbol{x}}=g_{f,t}(\boldsymbol{x})$.

At interference time, we can produce an adversarial example for the given target class by running $\widetilde{\boldsymbol{x}}=g_f(\boldsymbol{x})$. In the case of P-ATN, the authors structure the model as a residual block: $g_f(\boldsymbol{x})=\tanh(\boldsymbol{x}+G(\boldsymbol{x}))$, where $G(\boldsymbol{x})$ is the core function, which represents the added perturbation in this case.

- **Spatially Transformed Network (stAdv):** In this work [35], which applies in the case of images, the authors replace the L_2 norm as a similarity measure with the concept of optical flow from computer vision. Instead of changing the pixel values, they perturb their locations to generate adversarial examples. The optical flow controls the size of this perturbation, and minimizing it accounts for minimizing the pixel displacements. Using the optical flow, the authors tried to produce better quality adversarial images, with lower distortions.

  Specifically, their attack is an optimization attack similar to the L-BFGS and C-W attacks, but they minimize the following loss function:

$$L(\boldsymbol{x})=argmin_f L_{adv}(\boldsymbol{x},f)+\tau L_{flow}(f)$$

  The term $L_{adv}(\boldsymbol{x},f)=max(max_{i\neq t}g(\boldsymbol{x_{adv}})_i-g(\boldsymbol{x_{adv}})_t,\kappa)$ is the loss function of Carlini and Wagner, where $g(\cdot)$ are the class logits. The second term is the optical flow field of the image, defined as:

$$L_{flow}(f)=\sum_{p}^{all-pixels}\sum_{q\in N(p)}\sqrt{(\|\Delta u^{(p)}-\Delta u^{(q)}\|_2^2+\|\Delta v^{(p)}-\Delta v^{(q)}\|_2^2)}$$

  Here, (u,v) are the coordinates of a pixel p, $N(p)$ are the four neighboring pixels of p, and $f_i=(\Delta u^{(i)},\Delta v^{(i)})$ is the optical flow field, that is, the amount of displacement at every pixel. Minimizing the optical flow accounts for applying a similar amount of flow to a pixel and it's neighbors, thus enforcing a smoothness on the geometric transformation applied.

  The combined loss function can be optimized and the flow filed can be found. Then, since we know every pixel displacement applied, we can construct the adversarial example pixel by pixel. For the case of fractional displacements, bi-linear interpolation is applied. Finally, the adversarial example can be computed as:

$$\boldsymbol{x}_{\boldsymbol{adv}}^{(i)}=\sum_{q\in N(u^{(i)},v^{(i)})}\boldsymbol{x}^{\boldsymbol{q}}(1-|u^{(i)}-u^{(q)}|)(1-|v^{(i)}-v^{(q)}|)$$

  The authors tested this method in experiments, where it demonstrated significantly higher success rates against common defense methods (FGSM adversarial training, ensemble adversarial training, and projectile gradient) that FGSM or C&W. They also conducted

studies on the visual quality of the examples produced compared to the other attack methods, by using questionnaires, where they were chosen as more realistic in $47.01\% \pm 1.96\%$ of the cases, with 50% being the maximum possible score. This demonstrated the advantage of using the flow method (and more advanced computer vision similarity measures, rather than the L_2 norm), for producing realistic examples.

- **One Pixel Attack:** In this work [36], authors generated adversarial examples by modifying a single pixel. This amounts to the following optimization problem:

$$min_{\boldsymbol{r}} J(f(\boldsymbol{x}+\boldsymbol{r}), t), s.t. \|\boldsymbol{r}\|_0 \leq 1$$

The last constraint makes the optimization problem hard, so authors used an evolutionary algorithm, differential evolution (DE), to obtain the solution. DE does not require gradients, which makes it suitable in this case. They authors successfully demonstrated this method on standard model architectures and datasets.

- **Feature Adversary:** In this work ([38]), authors create adversarial examples by minimizing the distance between internal neural network layers instead of the output layer. Namely, they solve the problem:

$$min_{\tilde{\boldsymbol{x}}} \|\phi_k(\tilde{\boldsymbol{x}}) - \phi_k(\boldsymbol{x}_t)\|, s.t. \|\tilde{\boldsymbol{x}} - \boldsymbol{x}\|_\infty < \delta,$$

where ϕ_k is the output of a κ-th intermediate layer. Here, $\boldsymbol{x}$ is the input, $\tilde{\boldsymbol{x}}$ is the adversarial sample we are searching, while $\boldsymbol{x}_t$ a sample belonging to the target class (guiding image).

The motivation behind this approach is the hypothesis that internal network representations are more dense: two samples belonging to different classes (thus being far apart in the output space) are nearer in their internal representations, since these interpretations encode more general features.

The authors managed to generate adversarial examples using this approach, which belong to the sample class as the guiding image most of the times. The examples produced also look more natural than those of standard methods.

- **Hot / Cold Attack:** In this work ([39]), Rozsa et. al. introduced a method of finding multiple adversarial examples per image input, while replacing norms similarity with a new metric called Psychometric Perceptual Adversarial Similarity Score (PASS). PASS runs in two stages: first, the perturbed image is aligned to the initial image. Then, the similarity between the two images is measured.

To perform the first step, a homography transform $\psi(\boldsymbol{x}', \boldsymbol{x})$ between the two images must be estimated, with a 3×3 homography matrix H, which aligns image $\boldsymbol{x}'$ to $\boldsymbol{x}$. To compute H, we need to maximize the enchanched correlation coefficient (ECC, [40]), between the two images, which leads to the following optimization problem:

$$argmin_H \left\| \frac{\bar{\boldsymbol{x}}}{\|\bar{\boldsymbol{x}}\|} - \frac{\psi(\bar{\boldsymbol{x}}', \boldsymbol{x})}{\|\psi(\bar{\boldsymbol{x}}', \boldsymbol{x})\|} \right\|,$$

where $\bar{\cdot}$ devotes image zero – mean normalization.

After that, the image similarities are computed. These are based on the structural similarity index (SSIM, [41]), which better models the human visual system than norms. I.e. pass computes:

$$PASS(\widetilde{\boldsymbol{x}}, \boldsymbol{x}) = SSIM(\psi(\widetilde{\boldsymbol{x}}, \boldsymbol{x}), \boldsymbol{x})$$

With these, the authors define an adversarial image as:

$$\underset{d(\boldsymbol{x}, \widetilde{\boldsymbol{x}})}{argmin}\, \widetilde{\boldsymbol{x}} : f(\widetilde{\boldsymbol{x}}) \neq y \wedge PASS(\boldsymbol{x}, \widetilde{\boldsymbol{x}}) \geq \tau,$$

where $d(\boldsymbol{x}, \widetilde{\boldsymbol{x}})$ is a dissimilarity measure, for example $1 - PASS(\boldsymbol{x}, \widetilde{\boldsymbol{x}})$ or L_p, and τ is a similarity threshold between the input image and the adversarial examples produced.

The adversarial examples are generated by the hot / cold approach suggested in the tile. If l devotes the current class and t the target class, the hot and cold approach moves towards t and away from l at each step. This is done by constructing a feature vector from the logits, where they take the absolute value of the corresponding logit for the case of the target class, and the negative value for the case of the current class. Then, they compute the gradients of this vector with respect to the image, and update the image towards the gradient at each step (Fast Gradient Value method, FGV, similar to FGSM). The gradient so constructed points to the direction of the target class and away from the current class. With this and by varying the $PASS$ threshold, the authors managed to create a diverse set of natural – looking adversarial images.

Black-box attacks: In the case of black – box attacks, the attacker does not have any knowledge of the network architecture or it's parameters, and can only insert inputs and receive the corresponding outputs. Hence, the attacker doesn't have access to the model's gradient as well. Due to this limitation, black-box attacks rely on examining input – output pairs in order to extract information about the network, and then use it to craft adversarial examples.

Black-box attacks rely mainly on two methods. One of them is to approximate the decision boundary of the attacked model, or approximate it's gradient. Another method is by performing heuristic search. That means, we set a set of rules than an adversarial example must specify, and then we apply search methods to detect them. Some of the proposed black-box adversarial attacks in the literature are summarized below.

- **Substitute Black-box Attack:**This attacking method ([42]) exploits the property of adversarial examples to transfer to different networks. The main idea is to take a training set similar to the data for our case, pass them to the back-box model to obtain labels, and then train another network on this dataset. This second network is now, fully known to us, so we can apply a white-box adversarial attack and create adversarial examples to a given input. Thanks to the transferability property, many of these examples will fool the blck – box model as well. Note that it is important that the labels come from the black – box model, because this is the network that we want to approximate. Essentially we are training the other network to do the same mapping as the first.

  But there is a problem with this basic approach: our attack would need to run many thousand queries to the black-box model to create a large enough dataset to train on. Such an attack would be infeasible in practical cases, where the number of queries is finite. To remedy this, Papernot et. al. proposed a method called Jacobian-based dataset augmentation.

The heuristic behind this method is that we would like to capture information about the decision boundary of the unknown model. So, assuming we have only a limited number of queries to the unknown model, we would like to use them on data points close to decision boundaries, to obtain more information about them. The method of Jacobian – based data augmentation tries to do that. Starting from a small initial dataset, we train a first approximate model. Then, we compute the Jacobian of the class output logit with respect to the inputs of this model. This gives us the direction in which the classification output changes. Thus, taking some data points towards this direction we are likely to get new points closer to the decision boundary.

Specifically, at each iteration the dataset is augmented with new examples computed as:

$$S_{n+1}=\{\boldsymbol{x}+\lambda\, sign(J_F(O[\boldsymbol{x}])):\boldsymbol{x}\in S_n\}\cup S_n\,,$$

where S_n is our dataset at the n-th iteration, $O(\cdot)$ is the unknown model, and $J_F(O[\boldsymbol{x}])$ is the Jacobian of the unknown model's output class logit with respect to the input. So we construct new examples by taking a step towards the direction of the Jacobian. We get their labels, retrain the second network, and repeat until we reach the maximum allowed number of queries.

- **ZOO (Zeroth Order Optimization methods):** In this work, Chen et. al. [37] designed an attack which approximates gradients numerically, making it suitable for black – box scenarios. They defined a loss function similar to C&W, namely

$$g(\tilde{\boldsymbol{x}})=max(max_{i\neq t}(\log[f(\boldsymbol{x})]_i-\log[f(\boldsymbol{x})]_t)-\kappa)$$

This loss is similar to the C&W loss, since the $\log(\cdot)$ function is strictly increasing, and its included to help numerical gradient computations and usage, due to the fact that it's usually the case that trained models output probability distributions greatly skewed towards one class. This modification helps to mitigate this effect, and numerical gradients have more reasonable values.

Then, they start from the numerical evaluation of gradient and Hessian per coordinate using the symmetric difference quotient, via:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i}\approx\frac{f(\boldsymbol{x}+h\boldsymbol{e_i})-f(\boldsymbol{x}-h\boldsymbol{e_i})}{2h}\,,$$

$$\frac{\partial^2 f(\boldsymbol{x})}{\partial x_{ii}^2}\approx\frac{f(\boldsymbol{x}+h\boldsymbol{e_i})-2f(\boldsymbol{x})+f(\boldsymbol{x}-h\boldsymbol{e_i})}{h^2}\,,$$

where $\boldsymbol{e_i}$ is the standard basis vector and h a small value.

Due to the above computation, ZOO doesn't need access to the model's gradients to generate adversarial examples. Of course, the cost of obtaining gradients this way is prohibitive. To mitigate that, authors propose a coordinate – wise updating method, where one coordinate is updated at a time, and they use their method in combination with well known optimization methods, such as Newton's method (where the coordinate – wise Hessian is used) or ADAM.

- **Boundary Attack:** The boundary attack ([43]) is a form of heuristic search trying to detect the decision boundary of the unknown black – box model. It can be targeted, meaning that we want to create an adversarial example in a pre-specified class, or untargeted, meaning that we only want to change the classification. We will describe the first case, the other being similar.

  The search method works by performing a random walk through the image space, by alternatively making two kinds of steps. In the first step, the input is adjusted by a small step towards the target. In the second step, a small amount of Gaussian noise is added, which accounts for a random step (modified to be perpendicular to the first). The algorithm performs these steps alternatively, until the decision boundary is hit, where the classification output changes. Apart from that, the authors ensure that each applied perturbation will not exceed certain bounds (for example he pixel range $[0,255]$, etc.).

Other attack types: In this section, we will briefly review some attack types not falling into the above categories.

- **Universal Perturbation:** The goal of this attack ([44]) is to determine a universal perturbation that is likely to produce adversarial examples when applied to any input image in the dataset. That is, we want to find a perturbation $\boldsymbol{r}$ which satisfies the following:

$$\|\boldsymbol{r}\|_p \leq \epsilon,$$
$$P(f(\boldsymbol{x}+\boldsymbol{r}) \neq f(\boldsymbol{x})) \geq 1-\delta$$

  This means, the perturbation must be smaller than ϵ, and it must produce an adversarial example with success probability greater than $1-\delta$.

  The authors achieved that by an iterative algorithm. It loops through the data samples and for each data point $\boldsymbol{x}^{(i)}$ that is still not miss-classified by the universal perturbation, that is, $f(\boldsymbol{x}^{(i)}+\boldsymbol{r}) = f(\boldsymbol{x}^{(i)})$, the algorithm uses DeepFool to find the minimum additional perturbation to make it miss-classified, i.e., $f(\boldsymbol{x}^{(i)}+\boldsymbol{r}+\Delta\boldsymbol{r}^{(i)}) \neq f(\boldsymbol{x}^{(i)})$. The additional perturbation is added to $\boldsymbol{r}$, which is then projected on the L_p ball with radius ϵ, so that the first condition is satisfied. The process repeats until the fooling frequency exceeds the threshold set by the second condition. The authors successfully applied this method in several standard deep network architectures.

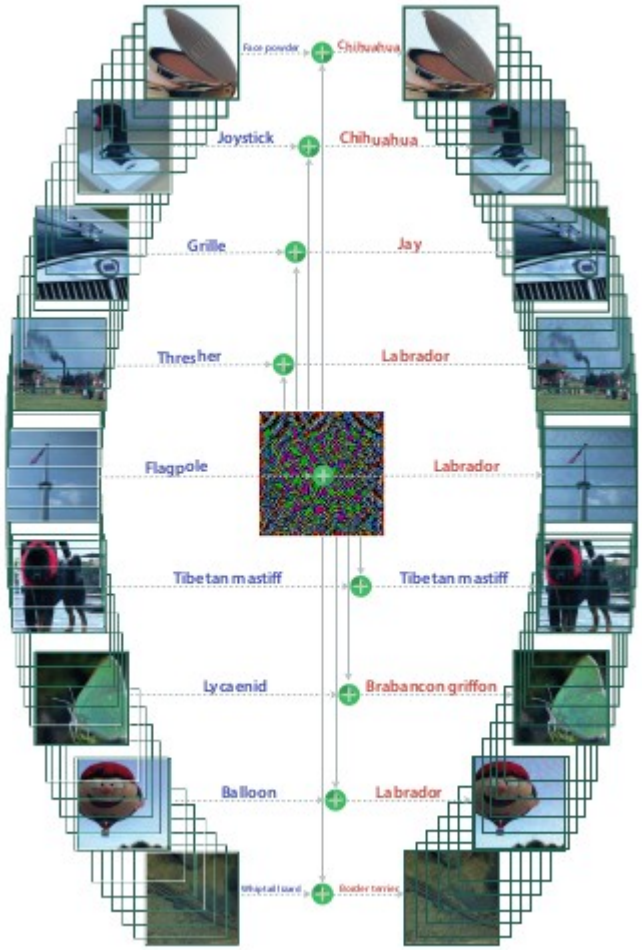

Fig. 3.8: Adversarial examples produced by the same universal perturbation in [44].

- **Poisoning Attacks and Back-doors:** Another type of attacks against neural networks (and other machine leaning models) are the so – called poisoning attacks. These attacks aim to corrupt a machine learning system in various ways. One common of those is data poisoning, where an attacker manipulates a portion of the training data aiming to reduce the model's performance. For example, researchers in [45] show in experiments that a corruption in 3% of the training data can cause a drop in accuracy of 11%, even with best defenses. Poisoning attacks can take also other forms, as for example manipulating the architecture of the network, the learning process, etc.

  An interesting form of poisoning are the so – called Backdoor attacks. Here, the attacker modifies the model in a way so that it behaves normally as it should: but, when some certain inputs are presented to the network (known to the attacker), the network will behave erroneous. A simple way to do that is to add corrupt data containing some non – visible patterns, whereupon the network will associate these patterns with the corrupt labels of the attacker. This can then be triggered at run-time, when another input containing such a pattern is presented. An example from [46] can be seen below, where the network learned to associate the small checkerboard patterns on the bottom right corner with the output "bird" (the rest of the images are random with respect to the label, and so they don't affect the classification).

Fig. 3.9: A backdoor attack: the network learned to associate the small pattern at the bottom with the output "bird". This can then be triggered by the attacker. From [46].

Meanwhile, more advanced attacks have been presented in the literature. For example in [47], a face recognition network enters a backdoor when a person wears a special type of glasses, and identifies them wrongly! In general, poisoning and back-doors are interesting topics in AI safety, and various attacks and defenses have been proposed, but describing them here would be out of the topic of this survey.

Adversarial Defenses

Adversarial defenses are methods to defend models against adversarial examples. They are usually based in the following strategies: either making neural networks more robust against adversarial attacks, trying to detect them, or trying to hide information about the network and its gradients to make attacking attempts harder. In this section, we will present a brief review of some popular adversarial defenses suggested in the literature.

- **Adversarial Training:** Adversarial training was one of the first methods proposed against adversarial examples. In it's simplest form in [27], the method iterates in the following fashion: First, we train the initial model with our initial dataset. Then, we generate adversarial examples using an attack method. We add the new examples (with the correct labels they should have) into our training set (or better, we take some adversarial samples and some original in a fixed proportion, usually $1:1$), and retrain the model. We iterate in this way, until our model is robust enough against the attack.

  Moreover, we can also adjust the loss function used, by including a second term measuring the loss on adversarial examples (as for example in [48]), namely:

$$L(X,Y) \propto \sum_{i \in CLEAN} L(\boldsymbol{x_i}, y_i) + \lambda \sum_{i \in ADV} L(\widetilde{\boldsymbol{x}}_i, y_i),$$

  where λ is weighting the importance of classifying adversarial examples correctly, relative to the standard loss.

  Although simple, such naive forms of adversarial training can lead to false robustness, that means models being robust against certain types of attacks, but still weak against different attacking methods., as authors in [49] showed. They argue that the coupling of model training and adversarial sample generation (from the same model) can lead to a form of gradient masking, where local gradients point away from the global minimum. To correct this, they perform ensemble adversarial training, where adversarial perturbations are drawn from an ensemble of static models.

On the other hand, authors in [50] view the problem of adversarial training as an optimization problem. Namely, we train a model to minimize the following loss function:

$$min_{\boldsymbol{\theta}} E_{(\boldsymbol{x},y)\sim D}[\max_{\|\tilde{\boldsymbol{x}}-\boldsymbol{x}\|_p \leq \epsilon} L(f(\tilde{\boldsymbol{x}};\boldsymbol{\theta}),y)]$$

We can understand the above loss if we think in the following way: An adversarial example $\tilde{\boldsymbol{x}}$ is a small perturbation of the input that changes the classification, that is, increases the value of the loss function. Hence, in the above formulation, for an input $\boldsymbol{x}$, we find the adversarial perturbation in the allowed region that maximizes the loss. Then, we want that the expectation over those values is minimal. That is, any perturbation will change our model's loss only minimally, in expectation.

Danski's theorem from optimization (which doesn't strictly apply for neural networks though), enabled them to compute the gradient of this loss, by first finding the maximum in the inner expression, and then computing the gradient there. Furthermore, the inner problem is essentially the definition of adversarial attacks, so the inner problem can be solved (approximately) by some adversarial attack method.

This is an important idea in adversarial training, since if gives us a sound mathematical definition of adversarial robustness, and minimizing that loss can ensure a network with high average robustness against all attacks. This comes with higher computational cost though.

- **Adversarial Logit Pairing:** In this work ([51]), authors enforce similarity of logits between original and adversarial samples, by adding a corresponding term to the adversarial training objective function, which now becomes:

$$L(X,Y)=\frac{2}{n}\sum_{i=1}^{n/2}(aL(\boldsymbol{x_i},y_i)+(1-a)L(\tilde{\boldsymbol{x}}_{\boldsymbol{i}},y_i))+\lambda\frac{2}{n}\sum_{i=1}^{n/2}\|Z(\boldsymbol{x_i})-Z(\tilde{\boldsymbol{x}}_{\boldsymbol{i}})\|_2\,,$$

by taking pairs of clean and adversarial samples. Regularizing logit similarity, instead of looking only at the classification as before, enforces the model to learn better internal representations of the data.

Authors used this method to perform a state-of-the-art defense against adversarial attacks on imageNet. However, [52] managed subsequently to break it.

- **Input Transformations:** A simple and practical way to defend against adversarial examples is by performing input transformations to the image before feeding it to the model. Authors in [53] found out that performing standard transformations such as bit-depth reduction, JPEG compression, total variance minimization, or image quilting before passing the image to the network enabled them to improve the resistance against grey- and black-box attacks. This happens especially if these transformations are applied during training as well, since their randomness and non – differentiability hardens possible attacks.

- **PixelDefend:** In this work ([54]), authors use a network architecture called PixelCNN ([55]) to perform image "purification", before passing it to the model. PixelCNN is a recurrent generative model that learns to predict the next pixel in an image, given the previous ones. Namely, this model attempts to learn the distribution:

$$p_{CNN}(\boldsymbol{x})=\prod_i p_{CNN}(x_i|x_{1:(i-1)})$$

First, a pixelCNN is trained on the initial dataset, in order to learn its pixel distribution. Then, the trained pixelCNN is used to either detect adversarial examples, by performing statistical tests to the pixelCNN outputs and the actual pixel values. Or, the pixelCNN is used to "purify" the input image, by modifying pixels to their "most likely" value, given by the pixelCNN, within a given range.

The authors showed that the pixelCNN was robust against various attacks, such as FGSM, DeepFool, and C&W. But later work found it to be vulnerable to the Backward Pass Differentiable Approximation (BPDA, [56]) and the Simultaneous Perturbation Stochastic Approximation (SPCA, [57]) attacks.

- **Defensive Distillation:** Defensive distillation ([58]) is an adversarial defense method that uses the idea of knowledge distillation for neural networks, suggested by Hinton et. al. in [59] as a means to compress models or model ensembles. The method involves two models, a teacher and a student network, and works in the following way. First, the teacher network is trained on the dataset, where a "Temperature" constant is introduced in the final softmax layer, namely:

$$softmax(Z(\boldsymbol{x});T)=\frac{\exp(Z(\boldsymbol{x})_i/T)}{\sum_{j=1}^{n}\exp(Z(\boldsymbol{x})_j/T)},$$

where $Z(\boldsymbol{x})$ are class logits, and T is the temperature constant.

This temperature method has the effect of spreading the class probabilities more uniformly across the various classes, thus avoiding very small numbers, which helps optimization (while the probabilities order remain the same). Then, the softmax outputs of the first network are used as labels for the teacher network, which learns to predict them, again with the same temperature. Finally, at test time, the temperature is set back to 1, as usual.

The intuition behind this method is that probability output vectors capture more information about the dataset than hard labels (for example they can show which data look similar to other, in the sense that their probabilities are near, while hard labels do not contain such information). Thus, we hope that the the new model will be more robust, and thus more resistant to adversarial attacks. Moreover, the gradient of the softmax is inversely proportional to T, and thus choosing a large T can hinder gradient – based attacks, such as FGSM, but it turns out to be susceptible to C&W attacks ([31]).

- **Stochastic Activation Pruning (SAP):** Stochastic activation pruning ([60]) is a defense method, which works by trying to introduce randomness in the model, by pruning some of the nodes at test time. This is similar to dropout, but with the difference that the probability of a unit to be dropped is inversely proportional to the value of it's output (activation). Namely, for an input $\boldsymbol{x}$, the probability that neuron i in a layer is kept is:

$$p_i=\frac{|h_i|}{\sum_{j=1}^{n}|h_j|},$$

where h_i is the output (activation) of the neuron at input $\boldsymbol{x}$, and n is the total number of neurons in that layer. Then, similar to dropout, the outputs of the remaining neurons are re-

scaled, in order to retain the same expected output, and thus the pruning doesn't modify the classifications:

$$h_i = \frac{h_i}{1-(1-p_i)^r},$$

where r is the number of neurons selected from the layer in consideration. Due to this, SAP can be applied to pre-trained models without any additional work.

Authors showed in their experiments that SAP is robust against FGSM or BIM. However, [56] showed that the defense can be circumvented if the attacker knows that it is applied. Namely, the argue that SAP is a form of obfuscating gradients, in the form of stochastic gradients: the pruning of units changes model gradients in a stochastic way. But they showed that the true gradient can be then estimated by taking the expectation of stochastic gradients over multiple forward passes.

- **Adversary Detection Network:** Authors in [62] proposed a method to detect adversarial examples by using a second neural network, called the detector network. This would take the output of a certain layer of the original network as input, and then classify it as being clean or adversarial.

  To train the detector network, they created adversarial examples and included them in the dataset, by using the BIM method. They modified it in order to take account on the detector's loss as well (that is, take steps in the direction that increase either the standard loss, or the detection network loss). They then performed standard adversarial training.

  Authors found in experiments that the placing the detector network at different layers of the original network gives different properties against different attacks. They also argued that fooling this approach is harder, since one would need to fool both networks. However, Carlini and Wagner found in [63] that the model is hard to train, produces many false positive fates against their attack, and is weak against the substitute black – box attack ([30]).

- **Kernel Density Estimates (KDE):** Feinman et. al. in [63] proposed using the method of kernel density estimates in order to decide if a given sample is near the manifold of it's class. Namely, using the model's logits, they estimate the density of a sample $\boldsymbol{x}$ by the formula:

$$KDE(\boldsymbol{x}) = \frac{1}{|X_t|}\sum_{\boldsymbol{x_i}\in X_t} \exp\left(\frac{-\|Z(\boldsymbol{x})-Z(\boldsymbol{x_i})\|_2^2}{\sigma^2}\right),$$

  where X_t are the points belonging to the same class as $\boldsymbol{x}$, and $Z(\boldsymbol{x})$ are the class logits. So, they used the standard density estimation method with a Gaussian kernel, by using the class logits to perform the estimation, relying on the usual assumption that they are a more meaningful and "unwrapped" transformation of the data than their original form. An adversarial example being away enough from it's supposed class manifold should thus give a low density, and thus be detectable. However, authors note that there are cases that this assumption can fail, for example if the decision boundaries contain "pockets" or other irregularities.

Carlini and Wagner showed in [62] than this method can be defeated if the attacker is aware of it being used, by incorporating the KDE density into the optimization problem, and thus produce examples that fool both the network and the detection method. They performed that using the BIM attack. Moreover, in their experiments, the method performed poorly in the CIFAR dataset.

- **Bayesian Uncertainty Estimates:** Feinman et. al. in [63] suggested to use Bayesian uncertainty estimate methods in order to approximate the uncertainty of a given sample, and thus decide whether it is adversarial or benign. A practical way to do this at test time is to use dropout, since it has been showed in [64] than this is equivalent to a deep Gaussian process. We fun several forward passes of a sample using dropout, and then we can use the results to estimate the uncertainty of this sample, as:

$$U(\boldsymbol{x}) \approx \frac{1}{N}\sum_{i=1}^{n} f(\boldsymbol{x})_i^T f(\boldsymbol{x})_i - (\frac{1}{N}\sum_{i=1}^{n} f(\boldsymbol{x})_i)^T (\frac{1}{N}\sum_{i=1}^{n} f(\boldsymbol{x})_i),$$

where N is the number of forward passes, and $f(\cdot)$ is our model. This is an estimation of class output variances, and we can take the mean of the uncertainty vector as a scalar estimate of the input's uncertainty. Authors suggest that uncertainties of benign and adversarial examples should have different distributions, thus enabling the detection of the first. Carlini and Wagner bypassed this method as well, but claim that it was the hardest to attack against, compared to the other methods they tried.

Summary

In this section, we presented some background on adversarial examples, popular attacks and defenses, as well as methods of adversarial training. Of course, apart from the presented methods, there is a vast amount of work in this area, and we inevitably left out many of them. It should also be noted that many new attacks are presented, and many defense methods bypassed over time (as for example [62] shows us). The interested reader should follow the referenced researchers for further information.

Robustness Verification & Formal Methods

As we saw in the previous section, adversarial examples pose a serious threat against neural network security, and hinders the application of Deep Learning in safety critical domains, such as medical diagnosis, automated driving, etc. The defense methods we presented merely rely on useful heuristics suspected by the respective researchers, and they cannot offer any absolute safety or robustness guaranties. In fact, most defense methods have been found to be seriously insecure, as for example [62] has shown. Adding to that our lack of understanding neural networks makes this problem even more challenging.

In the recent years, researchers have been becoming aware of the above problems and limitations, which led to a more serious study of the area of AI safety. One of the challenges here is to offer better robustness guarantees and protection that the previous adversarial defenses did.

Verification is the effort to provide rigorous claims and proofs about the robustness or some other properties of neural networks in various cases. Given a neural network N and a property C, we need a proof that N will satisfy C, or we should provide a counterexample. In case that's impossible, some lower bound, approximation or strong statistical guarantee should be provided.

More specifically, following [4], we can split the guarantees given by verification techniques in the following categories:

- **Deterministic guarantees:** They guarantee the property exactly, using mathematical – formal methods (similar to a theorem).
- **One – sided guarantees:** They can provide some useful upper or lower bound for the property in question (for example, minimum perturbation needed to miss-classify an input). In some approaches, bounds might converge to some variable in the limit.
- **Statistical guarantees:** They approximate the probability of the property holding. They are not strictly secure, but are usually more feasible to implement in practice.

The properties under question that we want to guarantee are usually adversarial robustness, but there could be other cases as well. Validation uses methods from mathematics and computer science, such as formal verification, Satisfiability modulo theories (SMT), linear or mixed integer programming, abstract interpretation, optimization and more.

In the next paragraphs, we will give a brief overview of the properties we want to verify and the mathematical methods being used. Then, we will present some of the most promising results in the literature, as of today.

Properties

The most standard property we want to inspect is adversarial robustness. This can be separated in either local robustness (robustness around a specific point) or global robustness, which involves every possible input. Local robustness can be defined formally as follows:

Def. (Local Robustness): A neural network $f(\cdot)$ is locally robust at input $\boldsymbol{x_0}$ (with class output k) under norm p and threshold ρ, if for every perturbation $\boldsymbol{r}$ with $\|\boldsymbol{r}\|_p \leq \rho$ we have: $f(\boldsymbol{x}+\boldsymbol{r})_k \geq f(\boldsymbol{x}+\boldsymbol{r})_i, \forall\, i \neq k$. That is, the outputted class does not change. The maximum ρ that satisfies this property, $\rho_f(\boldsymbol{x_0}, p)$, is called the robustness radius at point $\boldsymbol{x_0}$ under norm p .

Similarly, we can define a global robustness for a model, at every possible input point.

Apart from that, other useful to verify properties of neural networks can include the following:

Def. (Reachability): Given a neural network $f(\cdot)$ and an input region R , the output reachable set is defined as: $Reach(f, R) = \{f(\boldsymbol{x}) | \boldsymbol{x} \in R\}$.

A property we usually want to verify if whether a specific region D lies in the reachable set.

Def. (Interval Property): Given a neural network $f(\cdot)$ and an input region R , the interval property of f and R is a convex set such that: $I(f, R) \supseteq Reach(f, R)$. Ideally, the interval could be the convex hull of a set of points.

A property we usually wish to specify is whether the neural network outputs will lie in a given interval for the inputs in R .

These are the most usual properties one attempts to verify, but other than those, other properties can be specified as well, depending on the problem, and on whether the verification method used is suitable for them. One should also keep in mind the problem of scaling to very large (deep)

networks with millions of parameters – actually, scaling is one of the main challenges of existing verification methods.

Tools

In this section, we turn to the tools employed in verification methods. These methods come form mathematics and computer science, and more specifically from formal verification – STM, linear or mixed – integer programming, optimization, and more.. We will give a very brief overview of each of these terms.

- **Boolean Satisfiability (SAT):** A Boolean formula is a formula involving Boolean variables (taking truth and false values), and some logical conjunctions between them. For example, $F(x_{1,}x_{2,}x3)=x1\wedge(x_2\vee\neg x_3)\vee(x_1\wedge x_3)$ is a Boolean formula (or function) in logical variables $x_{1,}x_{2,}x_3$.

  Boolean logic is the foundation of more advanced methods in verification. Many conditions or specifications in various applications (verification, AI, circuit design, etc.) can be expressed in the form of Boolean logic. Then, verifying some property of the system leads to questions of satisfiability of Boolean formulas (abbreviated SAT), that is whether there are variable assignments that satisfy it. This can be extended to other similar questions, such as the number of truth assignments etc.

  Generally, SAT is a NP complete problem, meaning that not efficient (polynomial) algorithm exists to resolve them. However, many algorithms and tools that are reasonably efficient in practical applications have been developed.

- **Linear Programming:** Linear programming is a branch of optimization, where the goal is to maximize a linear function, under a set of linear constraints. The general form of a linear program can be stated as follows:

  $$max_{x}\boldsymbol{c}^{T}\boldsymbol{x},s.t:A\boldsymbol{x}\leq\boldsymbol{0},\boldsymbol{x}\geq 0,$$

  where the inequalities are to be understood component-wise. Linear programming has numerous practical applications in logistics, planning, and many more.

  Many efficient algorithms to solve the linear programming problem have been developed, included in many software libraries. Apart from that, ideas and methods of linear programming (such as the simplex method or the introduction of slack variables – variables controlling the amount of violation of a constraint) have been extended to other domains.

- **Mixed – Integer Programming:** It is an extension of Linear Programming, where some of the variables involved are constrained to take only integer values. The general problem is NP – complete, but methods that are reasonably efficient for practical applications have been developed.

- **Satisfiability modulo theory (SMT):** Satisfiability modulo theory is an extension of SAT, where the variables and operations involved are not restricted to be logical. Instead, SMT clauses are allowed to represent first – order logic statements, which can involve numbers, as well as qualifiers like $\exists$ or $\forall$ (Boolean formulas on the other hand form a weaker form of logic, called propositional logic). The objects involved in a SMT formula come from an underlying theory, which specifies the objects and the operations involved (these could be integers, real numbers, or data structures such as byte vectors, arrays, etc.).

Practically, depending on the underlying theory used, a SMT solver will translate the higher – level statements into low – level statements involving SAT formulas, and also decision processes on other theories (for example, numerical statements). These are then passed to corresponding low – level solvers, and the answer is translated back to the original theory stated. SMT solvers for various theories have been developed, and SMT is a basic tool in formal verification.

- **Formal Verification:** Formal Verification is a branch of computer science, aiming to derive proofs and guarantees that programs, or systems will behave the way they intend. The first step towards this is to express the problem – statement about the program or system involved in a suitable modeling language. Then, formal verification has developed methods and tools that are able to operate at this abstract level and determine if some properties are satisfied or violated. Tools such as logic, SAT or SMT form the foundation of formal verification, along with modeling languages and other abstraction methods. Some Formal Verification methods have also been extended to handle statements about probabilistic systems.

- **Abstract Interpretation:** Abstract Interpretation is a theory of sound approximation of computer program semantics. It has been developed to be able to answer statements about computer programs, where a direct proof or analysis would be impossible. The main idea is to over – approximate variables into so – called abstract domains, and to perform operations on these abstract domains. Then, if the resulting domain satisfies the property we can guarantee that the original program satisfies the property as well. The abstract domains should be easily representable and computable, and a monotonicity property should hold, which means that operations performed should preserve ordering relationships across variables and domains ($\leq, \subseteq$ relations).

  Specifically, consider a (concrete) domain C, where our inputs lie ($\boldsymbol{x} \in C$), a series of operations f applied on elements in C, and some property our outputs should specify, which we will write in the form $\boldsymbol{y} = f(\boldsymbol{x}) \in O$, where O is some desired output set. Then we define an abstract set A, which over-approximates C, and it's easier to compute. Then, we determine the set $O' = f(A)$. If the operation f preserves ordering, that is, $A \subseteq B \Rightarrow f(A) \subseteq f(B)$ and if $O' \in O$, then we can guarantee that all elements in the original domain. $O' = f(A)$ is calculated by replacing the concrete operations with abstract operations applied on abstract domains.

  As an elementary example, consider that we want to verify the following property: $x \in [0.5, 0.7] \Rightarrow \ln(x^2) < 0$. We over – approximate x by the interval $A = [0,1]$ and perform the abstract operations: $A^2 = [0,1], \ln(A^2) = [-\infty, 0]$, so the property is verified.

  The abstract domains used in abstract interpretation should be easily representable and computable. The usual elements included in Euclidean spaces involve:

  Intervals: Each variable x_i is bounded in an interval: $x_i \in [l_i, u_i]$.
  Polyhedrons: A polyhedron can be represented as a set of linear constraints in the form $\sum_{i=1}^{n} a_i x_i \leq b$.

  Zonotopes: A zonotope can be expressed in the form $z_i = a_i + \sum_{j=1}^{m} b_{ij} \epsilon_j$, where $e_j \in [l_j, u_j], j = 1, .., m$. Zonotopes can express more complicated spaces than intervals, and are usually easier to compute with than polygons.

- **Optimization:** Optimization methods have also been used for verification purposes. Apart from Linear Programming or MILP, ideas based on Convex Optimization (optimization methods for convex functions, which are much better understood theoretically), and ideas in duality theory have been employed.

Robustness Verification

As we saw previously, adversarial defense methods do not offer any theoretical guarantees of their success. And not only that, but most do indeed fail in practice. To make progress towards neural network safety, researchers have tried to find methods that can guarantee if a given model is adversarialy robust, independent of the attack method used. Some of these methods will be presented below.

CLEVER (Cross Lipschitz Extreme Value for nEtwork Robustness)
In this work ([65]), Weng et. al. approached the robustness estimation problem using the concept of Lipschitz property of continuous functions. The Lipschitz constant can give us a bound of the change of a function when its input is varied. Using that, the authors managed to estimate the minimum perturbation needed to change the class of a given input.

For a single variable continuous function, we say that it has the Lipschitz property, if there exists a real number $L \geq 0$ such that: $|f(x)-f(x_0)| \leq L|x-x_0|, \forall x, x_0 \in D_f$. From mean value theorem, we know that $(f(x)-f(x_0))/(x-x_0) = f'(\xi), \xi \in (x_0, x)$ if f is differentiable, so we can see that the least L satisfying the Lipschitz property will be the maximum value of $|f'(x)|$ (supremum). This definition can be extended for multi-variate functions as follows:

Theorem (Lipschitz property): Let $S \subset \mathbb{R}^d$ be a convex bounded closed set and $f: S \to \mathbb{R}$ be a continuous differentiable function on an open set containing S. Then f is a Lipschitz function with Lipschitz constant L_q if $\forall \boldsymbol{x}, \boldsymbol{y} \in S$ we have:

$$|f(\boldsymbol{x})-f(\boldsymbol{y})| \leq L_q \|\boldsymbol{x}-\boldsymbol{y}\|_p ,$$

where $L_q = max\{\|\nabla f(\boldsymbol{x})\|_q : \boldsymbol{x} \in S\}$, and $p^{-1}+q^{-1}=1$, $1 \leq p, q \leq \infty$.

Technically, neural networks with ReLU activation functions are not differentiable in a mathematical sense due to the ReLUs, which are not derivable at $x=0$. However, the "problematic" set of points in a neural network is finite (with measure zero) and we define a Lipschitz property for neural networks as well.

The authors used this concept to bound the perturbations around a point $\boldsymbol{x_0}$, by creating a lower bound for them. Specifically, they proved the following:

Theorem: Let $\boldsymbol{x_0} \in \mathbb{R}^n$, and $f: \mathbb{R}^n \to \mathbb{R}^l$ be a neural network, and let $c = argmax_{1 \leq i \leq l} f_i(\boldsymbol{x_0})$ be the predicted class of $\boldsymbol{x_0}$. Then, if f's components are Lipschitz, with Lipschitz constant L_q^j for component j, then all perturbations around $\boldsymbol{x_0}$ with p-norm smaller than:

$$\|\boldsymbol{r}\|_p \leq min_{j \neq c} \frac{f_c(\boldsymbol{x_0}) - f_j(\boldsymbol{x_0})}{L_q^j} = \rho(\boldsymbol{x_0})$$

will not change the class output. That is, our network has an adversarial robustness $\rho(\boldsymbol{x_0})$ around $\boldsymbol{x_0}$.

Proof: An adversarial perturbation $\boldsymbol{r}$ will change the classification output of the network from class c to class i if $f_i(\boldsymbol{x_0}+\boldsymbol{r}) \geq f_c(\boldsymbol{x_0}+\boldsymbol{r})$, with equality for the minimal perturbation. So we will have, using the Lipschitz property for $h(\boldsymbol{x}) = f_i(\boldsymbol{x}) - f_c(\boldsymbol{x})$ (h satisfies the above requirements):

$$|h(\boldsymbol{x_0}+\boldsymbol{r}) - h(\boldsymbol{x_0})| \leq L_q^i \|(\boldsymbol{x_0}+\boldsymbol{r}) - \boldsymbol{x_0}\|_p \Leftrightarrow$$

$$|(f_i(\boldsymbol{x_0}+\boldsymbol{r}) - f_c(\boldsymbol{x_0}+\boldsymbol{r})) - (f_i(\boldsymbol{x_0}) - f_c(\boldsymbol{x_0}))| \leq L_q^i \|\boldsymbol{r}\|_p \Leftrightarrow$$

$$|0 - (f_i(\boldsymbol{x_0}) - f_c(\boldsymbol{x_0}))| \leq L_q^i \|\boldsymbol{r}\|_p \Leftrightarrow$$

$$\|\boldsymbol{r}\|_p \geq \frac{|f_i(\boldsymbol{x_0}) - f_c(\boldsymbol{x_0})|}{L_q^i}$$

So, if $\|\boldsymbol{r}\|_p$ is smaller than the above bound, no adversarial examples can be found (if it is equal, the two classes have the same score). For each of the possible classes we have to compute the same bound, and then the total bound will be the minimum among classes:

$$\|\boldsymbol{r}\|_p \leq \frac{|f_i(\boldsymbol{x_0}) - f_c(\boldsymbol{x_0})|}{L_q^i} = \rho(\boldsymbol{x_{0,}} p)$$

This finishes the proof. □

Now, the next step is to estimate the Lipschitz constant. It holds that $L_q = max\{\|\nabla h(\boldsymbol{x})\|_q : \boldsymbol{x} \in S\}$, so we need to estimate the maximum value of the gradient's norm in the Ball $B(\boldsymbol{x_0}, \rho(\boldsymbol{x_0}))$. To do that, the authors employ ideas from a field of mathematics called Extreme Value Theory.

We can view a sample of the gradient in B, $\|\nabla h(\boldsymbol{x})\|_q$ as a random variable Y with CDF $F_Y(y)$. Then we considered the maximum over samples, $Z = max(Y_{1,\ldots}, Y_n)$ with CDF $F_Z^n(z)$. A theorem in Extreme Value Theory tells us that if the limiting CDF $\lim_{n\to\infty} F_Z^n(z)$ can belong only to one of three classes of distributions: a Gumber, a Frechet, or a reverse Weibull distribution.
In particular, the authors found out in their experiments that gradient norm samples follow the reverse Weibull distribution given by:

$$G(y; a, b, c) = \begin{cases} \exp\left(-\left(\frac{a-y}{b}\right)^c\right), & y < a \\ 1, & y \geq a \end{cases}$$

We can see that the maximum value of this distribution is $y_{max} = a$. So the authors sampled gradients, performed a p-test to determine if the CDF is reverse Weibull, fitted the distribution to the data, and from the parameter a they determine L_q^i.

The authors performed experiments with various networks, and compared the robustness radius they determined with the results on adversarial attacks performed. The experiments supported their claims, and presented a general robustness estimation method, independent of the network used.

However, I. Goodfellow in [66] points out some potential problems with this approach. Since it relies on statistical methods, it can fail if "wrong" gradient samples are collected, and assure us that the model is very robust, when in fact it's not. A possibility for this to happen is through gradient masking (for example CLEVER might observe zero gradients almost anywhere), but also the finite precision of computer arithmetics can contribute to false estimations.

Measuring Neural Net Robustness with Constraints: In this work ([67]), Bastani et. al. propose to use linear programming to estimate the robustness radius $\rho_f(\boldsymbol{x_0}, \infty)$ around a point (it is defines a bit different that here in their work, but the two definitions can be used interchangeably). To do that, they approximate the optimization problem defining $\rho_f(\boldsymbol{x_0}, \infty)$ as a linear program, and present a method to efficiently solve it.

The authors define also additional useful metrics for the network robustness, such as the adversarial frequency ϕ, measuring how often a model f fails to be ϵ-robust, as well as the adversarial severity μ, which measures the expected $\rho_f(\boldsymbol{x_0}, \infty)$ at points where f is not ϵ-robust. These can be formally defined as follows:

$$\phi(f, \epsilon) = P_{\boldsymbol{x} \sim D}[\rho_f(\boldsymbol{x}, \infty) \geq \epsilon]$$

$$\mu(f, \epsilon) = E_{\boldsymbol{x} \sim D}[\rho_f(\boldsymbol{x}, \infty) | \rho_f(\boldsymbol{x}, \infty) \geq \epsilon]$$

These quantities can readily be estimated from the training set.

Then, they encode the model computations as a set of constraints. Setting $\boldsymbol{x}^{(i)}$ to be the output of the i-th layer $f^{(i)}$, we have:

Fully connected layer: For a linear FC layer we have $\boldsymbol{x^{(i)}} = f^{(i)}(\boldsymbol{x^{(i-1)}}) = W^{(i)}\boldsymbol{x^{(i-1)}} + \boldsymbol{b^{(i)}}$, which can be encoded by the equality constraints $C_i = \wedge_{j=1}^{n_i}\{x_j^{(i)} = W_j^{(i)}\boldsymbol{x^{(i-1)}} + b_j^{(i)}\}$, with $W_j^{(i)}$ being the j-th row of $W^{(i)}$.

Convolutional layer: A convolutional layer can be represented as a FC layer by choosing a suitable matrix, so it falls into the previous case.

ReLU layer: Here we have $x_j^{(i)} = max(x_j^{(i-1)}, 0), j = 1,..,n_i$, which can be encoded by the constraints $C_i = \wedge_{j=1}^{n_i} C_{ij}$, with $C_{ij} = (x_j^{(i-1)} < 0 \wedge x_j^{(i)} = 0) \vee (x_j^{(i-1)} \geq 0 \wedge x_j^{(i)} = x_j^{(i-1)})$.
Max pooling layer: It can be encoded similarly to the ReLU layer above.

Output: The condition for the output is that the classification does not change, which can be encoded as $C_{out}(l) = \wedge_{l' \neq l}(x_l^{(k)} \geq x_{l'}^{(k)})$.

The set of constraints $C_f(\boldsymbol{x}, l) = C_{out}(l) \wedge (\wedge_{i=1}^{k} C_i)$ encode the computations of f. So the problem we want to solve is:

$$max\, \epsilon, \quad s.t: C_f(\boldsymbol{x}, l) \wedge \|\boldsymbol{x} - \boldsymbol{x_0}\|_\infty \leq \epsilon$$

This is an optimization problem over $(\boldsymbol{x}, \epsilon)$, an the solution will give us the robustness radius around $\boldsymbol{x_0}$.

But now a problem arises: namely, that the above problem is not a linear program. Indeed , every single constraint is linear, and if we have two linear constraints, $\boldsymbol{c_1^T x \leq 0, c_2^T x \leq 0}$ then their conjunction $\boldsymbol{c_1^T x \leq 0 \wedge c_2^T x \leq 0}$ is still a convex set, and they can be readily Incorporated into the linear program, just by adding the corresponding vectors into the matrix A , e.g. $A=[.., \boldsymbol{c_1^T}, \boldsymbol{c_2^T}, ..]^T$. But the problem arises by disjunctive conditions in the form $\boldsymbol{c_1^T x \leq 0 \vee c_2^T x \leq 0}$, whose combination does not form a convex set, and thus linear programming cannot be applied. These arise in our case due to the ReLU lalyers of the network (conditions C_i of the ReLU layers contain disjunctions).

To remedy this, the authors constructed a convex subset $\hat{C}_f$ of $C_f(\boldsymbol{x}, l)$ and solved the problem in this subset. This is not a problem, because the solution found in the restricted set will be always smaller or equal to the solution in the full set, so the robustness guarantee offered is still exact (if ϵ' is the restricted solution, we always have $\epsilon' \leq \epsilon$).

To build $\hat{C}_f$, the authors noted that in each ReLU unit, only one of the two disjunctions $C_{ij} = (x_j^{(i-1)} < 0 \wedge x_j^{(i)} = 0) \vee (x_j^{(i-1)} \geq 0 \wedge x_j^{(i)} = x_j^{(i-1)}) = C_{ij}^{(1)} \vee C_{ij}^{(2)}$ can hold at any given time. So, for the given seed input $\boldsymbol{x_0}$, they compute the input of the respective ReLU unit, find out the condition which holds, and discard the other. The idea here is that if ϵ is small enough, any neighbor $\boldsymbol{x}$ with $\|\boldsymbol{x} - \boldsymbol{x_0}\| \leq \epsilon$ will activate the same ReLU units. The resulting set of conditions $D(\boldsymbol{x_0})$ forms a linear convex set, and any standard linear solver can be used to solve the optimization problem.

Apart from that, the authors employ a method to speed up the computation, by first including only the equality constraints in the conditions set. The, for the solution found, they inspect if it is feasible with respect to the complete set or not. If it is, the problem is solved, otherwise they add additional constraints C_i and repeat the process. They found out that this lazy evaluation methods offers a large speedup compared to directly solving the full problem. This enabled them to compute robustness bounds on given inputs, and then statistical robustness estimates for the entire input region.

Formal Methods on Neural Networks

Aiming to provide provable guarantees on various properties, researchers have been recently making efforts to apply formal methods on neural networks, using tools from formal verification we described earlier (the work in [67] we described falls also in this category). Although similar attempts have been made in the past too, the challenge here is to cope with the very large multi – million parameter networks used in Deep Learning, and scaling up previous approaches in these dimensions is not trivial. Below we will describe some recent work on this direction. Unfortunately, the details here are long and tedious, so we will present the general ideas and refer the reader to the original works for further details.

Reluplex: In this work ([68]), authors applied ideas of SMT in order to verify properties of a neural network that was intending to be used at an airport as aircraft collision analyzer. Their approach can verify network properties expressible as SMT statements / constraints.

In their approach, they started with the standard simplex algorithm used in linear programming. Simplex aims to optimize a linear function bounded in a convex set given by linear constrains, as we already saw. To do that, simplex introduces some concepts, namely basic and non – basic variables, and a pivoting rule that updates each of the variables at each step. The operation that this algorithm essentially performs is that it traverses the convex polytope formed by the constraints, in a direction that tries to increase the objective function – since it is known that the maximum of a

linear function will always occur on one of these corners. Simplex has also methods to tell if the process has succeeded or failed.

The simplex algorithm can be extended in the case of SMT's, in order to search whether some property expressed in the language of SMT's holds or not under the constraints. As we saw earlier though, constraints imposed by the computations of a neural network do not form a convex set – this happens due to the ReLU activation functions, which introduce disjunctions in the constraint set. To remedy this, the authors introduced a set of operations to handle the ReLU nodes. First, solutions for the linear constraints are searched. If the linear constraints are infeasible, the complete problem is infeasible and the search stops. Otherwise, they check if these solutions conflict with the conditions imposed by ReLU, and depending on that they resolve conflicts and decide how to continue the search (namely, the choose and correct one conflicting ReLU, potentially violating some linear constraints, and they solve the linear constraints again, etc. This process can potentially not terminate, so at some points they split the problem on one ReLU into two sub-problems. In the worst case, the problem might be split at every ReLU, leading to an exponential number of simple LPs).

Their approach was capable of verifying properties of the relatively small network they used, but it is infeasible for the large networks used in Deep Learning, and it cannot support Max Pooling layers (at least not without modifications). We should also mention Planet here in [70], which as another approach based on similar ideas as Reluplex.

Binarized Neural Networks: The work in [69] considers a special case of neural networks, binarized neural networks. In these models, all parameters take only two discrete binary values, typically $\{-1,1\}$. They can arise by compressing standard neural networks in some cases (there are approaches that try to compress networks and reduce computation times in constrained environments by reducing the bits used to represent network parameters – usually, the quantization performed is not that extreme, and usually 8 or 4 bits are used per parameter).

The authors encode these networks as a set of Boolean conditions, capable to be processed by a SAT solver. They also used some improvements in the encoding and in the search procedure (counterexample guided search) over the brute force approach, in order to make the problem tractable. One can also save computational time by taking the special layered form of neural networks into account. Using this setting, the authors manage to verify two network properties, namely adversarial robustness and network equivalence (meaning that two neural networks will always produce the same output for the same inputs) for medium sized binarized neural networks.

Mixed – Integer Linear Programming Approaches: Some researchers attempted to approach the verification problem by encoding neural network computations and properties with the help of MILP. For example, we saw earlier that there are difficulties to formulate a neural network as a linear program due to the ReLU activation functions. But the behavior of the ReLU's can be easily formulated in the setting of MILP, which is what motivated work in this direction.

As an example, let us consider a ReLU function, $y=ReLU(x)=max(x,0)$, and lets assume that x lies in the interval $x\in[l,u]$. In the two trivial cases where $u\leq 0$ or $l\geq 0$ we will always have $ReLU(x)=0$ or 1 respectively, so the unit can be dropped away and replaced by a single condition. Otherwise, the ReLU can be either active or inactive. In this case, we introduce an integer variable a to indicate this, namely $a=0$ indicates that the ReLU is in the inactive region ($x\leq 0$), or else it is active. Then, the ReLU unit is equivalent to the following set of linear – integer constraints:

$$(y\leq x-l(1-a))\wedge(y\geq x)\wedge(y\leq ua)\wedge(y\geq 0)\wedge(a\in\{0,1\})$$

Indeed, if $a=0$, the 3rd and 4th clauses give us $y \leq u \cdot 0 \wedge y \geq 0 \Rightarrow y=0$, and this also implies that $x \leq 0$ from the 2nd clause. Conversely, if $a=1$ we get $y \leq x$ from the 1st clause, and since $y \geq 0$, we get $x \geq 0$, and the unit is active. We can similarly encode the maximum operation in Max Pooling layers, and of course fully – connected and convolutional layers can be encoded by linear constraints alone. Hence, the computation of a neural network can be reduced to a set of MILP constraints. Thus, we can use MILP methods to verify properties of neural networks, by encoding the property we wish to verify into the optimization objective.

Naively encoding the entire network in a set of MILP conditions is very inefficient though. Hence, researchers tried to come up with meaningful heuristics, taking the special structure of neural networks and the problem at hand into account, in order to reduce search time.

For example, in [71], researchers wanted to determine the upper (or lower) bound of a neural network's output, when the input is bounded in a convex polygon. To achieve that, they employed two search procedures: a local search, performed by gradient ascent, where starting from given point a local maximum larger than that point is determined, and a global search, where they search if a still greater value can be obtained (increased by a tolerance δ). They encode this as an MILP feasibility problem. If the MILP solver fails, the point is declared to be global maximum, otherwise the new point determined by the solver is taken as the initial point and the process repeats.

In [72] on the other hand, the authors employ MILP in order to determine the robustness of neural networks. They reduce solving time by introducing a pre-solving method that tries to remove as many integer variables as possible and to reduce the possible variable bounds, called progressive bound tightening. Their approach is able to handle networks with up to some hundreds of neurons.

Branch and Bound: Realizing the connection of the above approaches with heuristic search methods, authors in [73] present a Branch-and-Bound framework, where most of the previous verification algorithms can be seen as special cases. Branch-and-Bound are heuristic search discrete optimization methods, specially suited for NP-hard problems. They split the search space into domains and subdomains, having a structure similar to a tree. Then, they transverse the tree nodes solving the respective sub-problems. They also keep track of an estimate of maximum value. If it turns out that a tree node cannot produce a higher value, the node is discarded and the search proceeds.

They formulate verification problems as a minimization problem, and present a general branch-and-bound framework for it. They show how the rules introduced in reluplex or planet can fit into this framework. Further, they propose improvements in the branching and bounding methods, which lead to a method with increased speed over these two approaches.

Abstract Interpretation: As we have seen in the previous approaches, coming up with provable guarantees for neural networks comes at a high computational cost. On the other hand, statistical methods cannot offer absolute safety guarantees. To overcome these, researchers have recently tried methods between these two extremes. Some of these use the idea of over – approximation. Namely, we over – approximate the hard to compute problem with a relaxed but easier to solve problem. Sometimes over – approximation methods might reject safe instances as unsafe; but if they declare an instance as safe, then it is guaranteed to be safe.

Following this reasoning, researchers at ETH have come up with a line of work that uses the methods of abstract interpretation to over – approximate safety regions of neural networks and test safety properties. For example, in [11], they transform fully – connected, ReLU, convolutional and max pooling layers in a set of abstract transformations, capable to be used in abstract interpretation

methods. The elements they use are called conditional affine transformations (CAT), which are affine functions $f:\mathbb{R}^n \to \mathbb{R}^m$ controlled by logical constraints. Compositions of CAT functions are also CAT functions. The general definition is the following:

Def. (CAT function): $f:\mathbb{R}^n \to \mathbb{R}^m$ is a CAT function if one of the following holds:

1. $f(\boldsymbol{x}) = W\boldsymbol{x} + \boldsymbol{b}$
2. f is the compositions of CAT functions $f_1 \circ f_2$
3. f can is case-wise affine: $E_1 : f_1(\boldsymbol{x}), \ldots, E_k : f_k(\boldsymbol{x})$
4. Cases E are in the form: $E \stackrel{\text{def}}{=} E \wedge E\,|x_i \geq x_j\,|x_i \geq 0|\,x_i < 0$

For example, a ReLU layer is composed of n ReLU activations, $ReLU = ReLU_n \circ \ldots \circ ReLU_1$, where each one processes the i-th entry of the input and can be readily expressed as a CAT function:

$$ReLU_i(\boldsymbol{x}) = \begin{Bmatrix} (x_i \geq 0) : \boldsymbol{x} \\ (x_i < 0) : I_{i \leftarrow 0}\boldsymbol{x} \end{Bmatrix},$$

where $I_{i \leftarrow 0}$ is the identity matrix with the i-th row replaced by zeros. Similarly, they find a method to express the max pooling operation with CAT function, while convolutional layers can be expressed as fully connected layers and then be also transformed into CAT functions.

After this is done, the authors express possible perturbations around an input as an abstract set, using intervals or zonotopes. Then, using an abstract interpretation solver, they propagate the abstract set through the network layers which they expressed as abstract transformations with CAT functions, and get the abstract domain of possible outputs as a result, using a solver for abstract interpretation. Then, if the over – approximated abstract set satisfies the robustness property (e.g. the class output is the same – this can be checked by passing it to an abstract output layer), then the original network does too. The process is schematically depicted below:

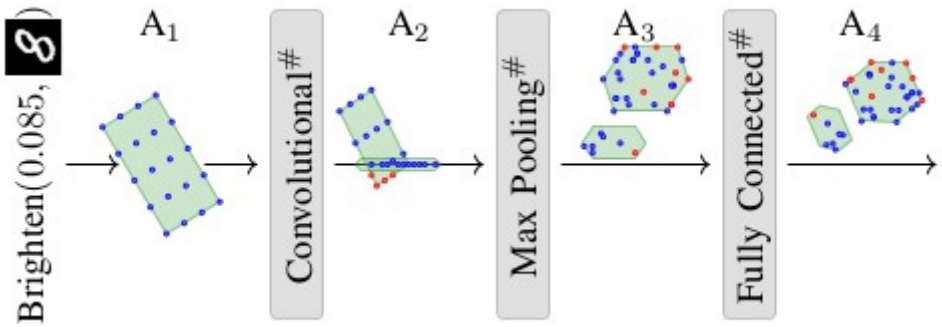

Fig. 3.10: Illustration of abstract interpretation based methods for model robustness.

By using these over-approximation methods, and employing existing solvers and advances in the field of abstract interpretation, the authors managed to perform formal verification on larger networks than before, and on convolutional networks consisting of about 50,000 neurons.

Moreover, as shown in [7], adversarial examples cannot only be produced by altering pixels, but also by performing geometric transformations, such as rotations and translations , on the input image. So, in subsequent work ([74], [75]), the authors try to use their abstract interpretation methods for the case of rotations, and other geometric transformations (typically translation). The

difficulty here is coming up with a meaningful abstract set to capture these possible transformations, that is not unreasonably wide (and thus useless). The authors used optimization (convex relaxation) methods to find reasonable sets, and showed how they can be expressed into abstract domains and processed by their solver. Thus, they achieved to verify geometric robustness for some cases for the first time.

Other: Although the above mentioned are the most promising approaches in the literature, other researchers have proposed additional ideas. For example, [76] uses symbolic interval propagation through layers, along with optimization methods to refine intervals (performing this approach naively would result in unreasonably wide intervals). They use this approach with a system called ReluVal to validate network properties. In [6], on the other hand, authors consider the problems of maximum safe radius and feature robustness. Using the Lipschitz property for all layers, they show that the problem can be approximated by a discrete optimization problem, which provides provable guarantees. Then, they reformulate the optimization problem as a two – player game, and present methods to solve it. Apart from these, other ideas have been proposed as well.

Training Provably Robust Networks

The approaches mentioned in the previous section do the following: given a neural network, they attempt to verify (prove or disprove) some given property. But, a number of researchers wondered if the reverse direction is also possible. Namely, can we build networks in such a way, that are robust to some property by construction (or by training them in a special way)?

This is another promising direction in AI safety. In this section, we will briefly present the main work in the literature towards this direction.

Convex relaxation: One of the first approaches towards this direction was the work of Wong et. al. in [77]. They managed to build ReLU – based neural networks that are provably robust against norm – bounded adversarial perturbations over the training data. To do that, they first considered a convex over – approximation of the activations, given all possible input perturbations. The starting point for this is to relax the ReLU activation functions into convex sets by replacing the ReLU with it's convex envelope. Specifically, the condition $z=max(\hat{z},0)$ imposed by the ReLU was replaced by the following convex conditions:

$$z\geq 0, z\geq \hat{z}, -u\hat{z}+(u-l)z\leq -ul,$$

assuming that the previous variable $\hat{z}$ is bounded by $\hat{z}\in[l,u]$. This amounts to replacing the ReLU curve by it's linear envelope, as depicted below:

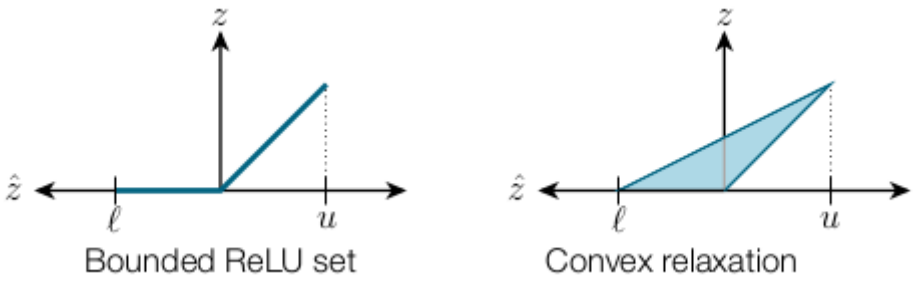

Fig. 3.11: Replacing the ReLU (bounded) condition by it's linear envelope.

Let's now consider an input $\boldsymbol{x_0}$ of the network with class y^*. The question of adversarially robustness ϵ around $\boldsymbol{x_0}$ is equivalent to the following optimization problem:

$$min Z(\boldsymbol{x})_{y^*} - Z(\boldsymbol{x})_{y^{targ}} = \boldsymbol{c}^T Z(\boldsymbol{x}),\ \ s.t. \|\boldsymbol{x} - \boldsymbol{x_0}\|_\infty \leq \epsilon \wedge C_f(\boldsymbol{x})$$

Here $Z(\boldsymbol{x})$ are the class logits, $C_f(\boldsymbol{x})$ are the conditions imposed by the network computation, and $\boldsymbol{c} = \boldsymbol{e}_{y^*} - \boldsymbol{e}_{y^{targ}}$ (with $\boldsymbol{e}_i$ being the i-th unit vector). If the minimum value found is negative then the class output has changed, and the network is not adversarially robust. Otherwise, if the minimum found is positive, then robustness is verified.

Now, the authors relax the ReLU activations as described above, and $C_f(\boldsymbol{x})$ becomes now a convex set. The overall conditions $Z_\epsilon(\boldsymbol{x}) = C_f^{conv}(\boldsymbol{x}) \wedge \|\boldsymbol{x} - \boldsymbol{x_0}\|_\infty \leq \epsilon$ is a convex set, and the above optimization problem is now a linear program. Since the relaxed set used contains the original set, if robustness is verified on the larger set, then it is also guaranteed for the restricted set. On the other hand, if we find a negative solution on the larger set, there might be the case that the optimum in the restricted set is still positive. So the method might characterize robust inputs as non – robust, but if an input is found non – robust, this is always true in the restricted set as well.

Still, solving such a program for an entire neural network is not very practical. For this, the authors employ a concept from optimization called duality. Given an optimization problem with constraints, duality uses the method of Lagrange multipliers to build a second problem (dual), the solution of which is an upper bound for the solution of the original problem. For example, in the case of a linear program, $min\ \ \boldsymbol{c}^T \boldsymbol{x},\ \ s.t: A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0$, the dual program is the following:

$$max\ \ \boldsymbol{b}^T \boldsymbol{y},\ \ s.t: A^T \boldsymbol{y} \geq \boldsymbol{c}, \boldsymbol{y} \geq 0$$

Here is the crucial idea of this work: As we saw, earlier, a neural network has been transformed to a linear program. Then the dual linear program was obtained. We can do the same process in reverse: namely, this dual program will correspond to another neural network! Since any solution of the dual program is an upper bound for the primal, we can just use gradient descend on the second network, and every solution that we find will be an upper bound. Finally, a point we omitted here is how to compute the bounds for the ReLU inputs described earlier. The trick is to compute an upper bound for the output using the dual network, and then propagate it back to the previous layers.

This is what the authors do: They construct the second network by determining it's parameters from the dual problem, and it's loss will be an upper bound for our adversarial robustness. Then, they train the second network on the dataset. If a low enough loss is achieved, then we have adversarial robustness over our training set. From the parameters of the second network, the parameters of the first network can be derived, and the problem is solved. Thus, a provably robust model has been trained. The second network can also be used at test time to determine whether the network is robust to an new, unseen input.

Using this approach, the authors managed to build provably robust medium – sized networks for MNIST in their experiments. A problem with this approach is that it cannot scale to large networks such as those for example used in ImageNet, at least not without substantial modifications. Moreover, as they show in their experiments, the upper bounds obtained might be too wide in some cases. Finally, we should also mention the work in [79] which also uses duality to provide upper bounds for robustness specification, in a slightly different way.

Interval Bound Propagation: Another approach towards building provably robust models is via the method of Interval Bound Propagation (IBP). In [78] and a subsequent series of works, Deep Mind researchers employ this method in training robust neural networks.

Similarly as before, the robustness property is formulated as an optimization problem in the form:

$$max_{\boldsymbol{x_0}\in X(\boldsymbol{x_0})} \boldsymbol{c}^T \boldsymbol{z_K}+\boldsymbol{d}\,,\ \ s.t: \boldsymbol{z_k}=h_k(\boldsymbol{z_{k-1}}), k=1,\dots,K\,,$$

where $h_k(\cdot)$ are the network layers, and $X(\boldsymbol{x_0})=\{\boldsymbol{x}:\|\boldsymbol{x}-\boldsymbol{x_0}\|_\infty \le \epsilon\}$. The goal of IBP is to find an upper bound for the previous problem. This is done by bounding the activation $\boldsymbol{z_k}$ of each layer by an axis – aligned bounding box, as: $\underline{\boldsymbol{z_k}}(\epsilon)\le \boldsymbol{z_k}\le \overline{\boldsymbol{z_k}}(\epsilon)$, where $\underline{\boldsymbol{z_k}}(\epsilon), \overline{\boldsymbol{z_k}}(\epsilon)$ are lower and upper bounds respectively. These bounds can be computed from the previous layer bounds as:

$$\underline{z_{k,i}}(\epsilon)= \min_{\underline{\boldsymbol{z_{k-1}}}(\epsilon)\le \boldsymbol{z_{k-1}}\le \overline{\boldsymbol{z_{k-1}}}(\epsilon)} \boldsymbol{e_i}^T h_k(\boldsymbol{z_{k-1}})$$

$$\overline{z_{k,i}}(\epsilon)= \max_{\underline{\boldsymbol{z_{k-1}}}(\epsilon)\le \boldsymbol{z_{k-1}}\le \overline{\boldsymbol{z_{k-1}}}(\epsilon)} \boldsymbol{e_i}^T h_k(\boldsymbol{z_{k-1}})$$

with $\underline{\boldsymbol{z_0}}=\boldsymbol{x_0}-\epsilon\boldsymbol{1}$ and $\overline{\boldsymbol{z_0}}=\boldsymbol{x_0}+\epsilon\boldsymbol{1}$ for L_∞ bounded perturbations.

This problem can be easily solved for all types of layers. For ReLU activation layers, and generally for any monotonic activation function we simply have $\underline{\boldsymbol{z_k}}=h_k(\underline{\boldsymbol{z_{k-1}}})$ and $\overline{\boldsymbol{z_k}}=h_k(\overline{\boldsymbol{z_{k-1}}})$. For affine layers $h_k(\boldsymbol{z_{k-1}})=W\boldsymbol{z_{k-1}}+\boldsymbol{b}$ we can easily show that the solution is:

$$\underline{\boldsymbol{z_k}}=\boldsymbol{\mu_k}-\boldsymbol{r_k}, \overline{\boldsymbol{z_k}}=\boldsymbol{\mu_k}+\boldsymbol{r_k},$$

$$\boldsymbol{\mu_k}=W\boldsymbol{\mu_{k-1}}+\boldsymbol{b},\ \ \boldsymbol{r}=|W|\boldsymbol{r_{k-1}},\ \ \boldsymbol{\mu_{k-1}}=\frac{1}{2}(\overline{\boldsymbol{z_{k-1}}}+\underline{\boldsymbol{z_{k-1}}}),\ \ \boldsymbol{r_{k-1}}=\frac{1}{2}(\overline{\boldsymbol{z_{k-1}}}-\underline{\boldsymbol{z_{k-1}}})$$

where $|W|$ is the element – wise absolute value. With these, we can construct an upper bound for the output as:

$$\max_{\underline{\boldsymbol{z_K}}(\epsilon)\le \boldsymbol{z_K}\le \overline{\boldsymbol{z_{k-1}}}(\epsilon)} \boldsymbol{c}^T \boldsymbol{z_K}+\boldsymbol{d}$$

We can now bound the adversarial specification by replacing the true class logit by its lower bound and the other logits by their upper values:

$$\hat{z_{K,y}}=\begin{cases}\overline{z_{K,y}}(\epsilon), & if\ y\neq y_{true}\\ \underline{z_{K,y}}(\epsilon), & else\end{cases}$$

Hence, for all $y\neq y_{true}$ we can express our robustness specification as:

$$(\boldsymbol{e_y}-\boldsymbol{e_{y_{true}}})^T \hat{\boldsymbol{z}}_{\boldsymbol{K}}(\epsilon)= \max_{\underline{\boldsymbol{z_K}}(\epsilon)\le \boldsymbol{z_K}\le \overline{\boldsymbol{z_{k-1}}}(\epsilon)} (\boldsymbol{e_y}-\boldsymbol{e_{y_{true}}})^T \hat{\boldsymbol{z}}_{\boldsymbol{K}}$$

and we can formulate the following loss function:

$$L=\kappa L_{entr}(\boldsymbol{z_K}, y_{true})+(1-\kappa)L_{spec}(\hat{\boldsymbol{z}}_{\boldsymbol{K}}(\epsilon), y_{true})\,,$$

where L_{entr} is the usual cross – entropy loss, and L_{spec} is the upper bound we determined above, while κ controls the relative weight of each term.

The authors use this method to train robust networks. The advantage of this approach is that due to the simplicity of computing the bounds, the method is easier to scale to larger models. The disadvantage is that the bounds obtained by IBP might be too wide (although they usually turn out to be tighter than one would expect). The overall method is schematically depicted below:

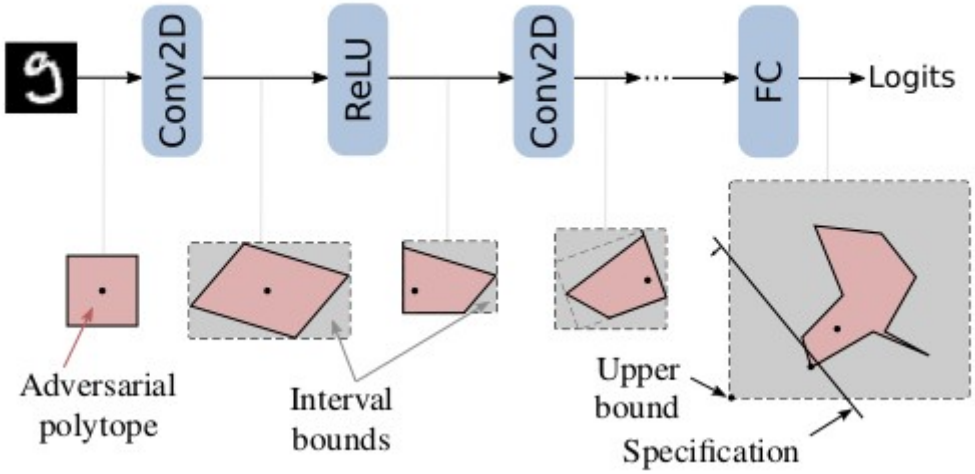

Fig 3.12: Illustration of Interval Bound Propagation.

Abstract Interpretation: In [80], the authors leverage their previous work on abstract interpretation for robustness verification, in order to use it for training provably robust neural networks.

Their approach works in the following way. First, they transform robustness into an equivalent optimization problem, as they other works. Then, for every input point, they construct the abstract set of possible perturbations. Using the methods they developed, they propagate this abstract set through the network (where layers are defined as abstract transformations in their setting), and they get the abstract output set. Then, they determine the maximum value of their loss over the abstract output. This value can be differentiated with respect to the network parameters and optimized. In this way, they manage to train provably robust models.

The authors managed to train CNN with about 80000 neurons with this approach. It is slower than IBP, but it can represent more exact bounds, and it can potentially scale as well.

Reduction to Gaussian Robustness: In a recent work in [81] (inspired by related ideas in [82]), authors make a step forward in scaling provably robust models towards state of the art deep learning models, by managing to build such a model for ImageNet. They accomplished that by finding a way to reduce the problem of L_2 certifiable robustness to statistical robustness under Gaussian (isotropic) noise.

Specifically, authors proved the following interesting theorem:

Theorem: Let $f:\mathbb{R}^d \to Y$ be any classifier, and let $\boldsymbol{\epsilon} \sim N(\mathbf{0}, \sigma^2 I)$. Let also g be another classifier, such that: $g(\boldsymbol{x}) = argmax_{c \in Y} Pr[f(\boldsymbol{x}+\boldsymbol{\epsilon}) = c]$. If $c_A \in Y$ and $\underline{p_A}, \overline{p_B} \in [0,1]$ satisfy:

$$Pr[f(\boldsymbol{x}+\boldsymbol{\epsilon})=c_A] \geq \underline{p_A} \geq \overline{p_B} \geq max_{c \neq c_A} Pr[f(\boldsymbol{x}+\boldsymbol{\epsilon})=c],$$

then $g(\boldsymbol{x}+\boldsymbol{\delta})=c_A$ for every $\|\boldsymbol{\delta}\|_2<R$, with $R=\frac{\sigma}{2}(\Phi^{-1}(\underline{p_A})-\Phi^{-1}(\overline{p_B}))$.

This means, that if we have a classifier that is robust under Gaussian noise, then we can construct a second classifier, $g(\boldsymbol{x})=argmax_{c \in Y} Pr[f(\boldsymbol{x}+\boldsymbol{\epsilon})=c]$ which is provably robust for perturbations $\|\boldsymbol{\delta}\|_2<R$ with R given by the above theorem. Hence, the authors first trained a neural network for ImageNet, using data augmentation with Gaussian noise, and then applied the above method to define a certifiably robust classifier for ImageNet.

Now, for a new sample $\boldsymbol{x}$ we have to find it's class and also to verify it's robustness radius. This can be done using statistical methods up to any probability close to 1, provided we perform enough sampling. The first step is to pass several copies of $\boldsymbol{x}$ corrupted under $N(\boldsymbol{x}, \sigma^2 I)$ to f, and use a statistical test to determine the most probable class with high confidence. This is the output of g. Then, the second step is to find the (certifiable) robustness radius of $\boldsymbol{x}$. For that, we perform again sampling under Gaussian noise, and use statistical tests to determine a valid lower bound $\underline{p_A}$ for f with high confidence. Then, we can use the theorem above to compute the robustness radius under L_2. This confidence can be as high as we need, provided we draw enough samples. If the required confidence cannot be attained, the algorithm will detect this, and won't certify $\boldsymbol{x}$.

The authors hope that such smoothing approaches is an important step forward towards certifying large state of the art Deep Learning models.

Neural Networks with Logical Rules

The works we described above aim to build neural networks that satisfy some given property. Another similar line of work is aiming to build models that satisfy a set of logical rules. That is, we know some rules that our data should satisfy, and we try to build models that satisfy these rules by construction, or that it has the ability to learn them from the data. In this section, we will present a brief overview of the work being done in this area.

One direction in this approach is to incorporate logical rules we know to hold for our dataset into the model. [83] is a recent work towards this goal, improving on older approaches such as [82].

At first, the authors show a method to convert the discrete logical constraints into continuous constrains, typically used in optimization. They first start with the set of logical atoms $t \leq t'$, $t \neq t'$ and encode them as a continuous loss, which is 0 only if the respective constraint is satisfied, and positive else:

$$L(t \leq t') = max(t - t', 0), \quad L(t \neq t') = \xi[t = t'], \quad \xi > 0$$

Note than we have $L(\phi)=0$ if and only if ϕ is satisfied, and $L(\phi)>0$ (also, the constraints $t=t', t<t'$ can be encoded using the above, along with logical conjunctions). Then, compound AND and OR constrains are encoded thus:

$$L(\phi' \wedge \phi'') = L(\phi') + L(\phi'')$$

$$L(\phi' \vee \phi'') = L(\phi') \cdot L(\phi'')$$

This encoding is valid, for example $L(\phi' \wedge \phi'')=0$ if and only if $L(\phi')=0$ and $L(\phi'')=0$. Finally, NOT conditions are transformed to their logical equivalents and then encoded, for example $L(\neg(t=t'))=L(t\neq t')$, etc. Thus, any Boolean formula can be encoded as a loss, and the loss is minimized ($L(\psi)=0$) if and only if the formula is satisfied.

Then, the training objective can be formulated as:

$$argmin_{\boldsymbol{\theta}} E_{\boldsymbol{x}\sim D} L(\phi)(\boldsymbol{x}, \boldsymbol{z}^*(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{\theta}) \text{, where}$$

$$\boldsymbol{z}^*(\boldsymbol{x}, \boldsymbol{\theta}) = argmin_{\boldsymbol{z}\in A} L(\neg\phi)(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta})$$

That is, we minimize the expected maximum violation of the constraints ($min\, L(\neg\phi)=max\, L(\phi)$). The algorithm employed is similar to Projected Gradient Descent used for adversarial training. Namely, for each mini-batch, the inner minimization problem is solved, and then a gradient descent step is performed with the found $\boldsymbol{z}^*$ from the inner problem. Apart from that, the authors employ some additional modifications in order to improve the optimization efficiency. Namely, since optimizing over a lot of conditions can be cumbersome, the authors extract a convex set from the conditions, and set the rest as optimization constraints.

Apart from that, the authors find a way to perform logical queries to the network. Namely, the user determines some network variables over which the query will be performed, and some conditions for these variables. Then, this condition is transformed into a loss function, and minimization is performed, until the loss reaches the value zero, or until time – out. If the loss becomes zero, we have found a valid answer, and the respective variable values (or some quantity of interest defined over them) is returned. Otherwise, false is returned. Using this method, authors queried for example if an adversarial example classified as “deer” exists at a near distance to the given image, under the constraint that some neuron is deactivated. They thus achieved to formulate a variety of interesting questions as logical queries over the network.

In the reverse direction, authors in [85] attempt to learn logical relations from data. More specifically, their aim is to tackle the max - SAT problem, and transform it into an approximate continuous version, which can then be employed in neural networks.

Max – SAT is related to the SAT problem: given a Boolean formula, find truth assignments so that as many clauses are possible are satisfied. If we have n logical variables $v_i \in \{-1, 1\}$ and m clauses (here we let our logical variables to take values $-1,1$ instead of $0,1$, just for simplifying the formulation), then max – SAT can be written as:

$$max_{\boldsymbol{v}\in[-1,1]^n} \sum_{j=1}^{m} \vee_{i=1}^{n} 1[s_{ij} v_j > 0]$$

Here $S=s_{ij}$ is a matrix “deciding” the value that each variable must take in the respective clause (true, false, or indifferent). To understand this, consider an example clause $c=(A\vee B)$, which is satisfied if either A or B are true. If we let A, B take values in $\{-1,1\}$ instead, we could write the same conditions as $1[A>0]\vee 1[B>0]$, which is the formulation we have above. So, max – SAT just tries to maximize the sum of true clauses.

Now, the authors relaxed the max – SAT and the logical conditions into a continuous approximation (in a way that the approximate solutions obtained are good enough), and showed a way to learn some unknown variables given the others, so that a function approximating max – SAT is

maximized. They could thus use the approximate solver as a network layer, as seen in the figure below:

Fig. 3.13: The approximate max – SAT layer used in [85].

Crucially, all parameters of this layer are differentiable. That means, if we define a loss function, then we can learn either the variables $\boldsymbol{v}$ or the clause matrix S that maximize this loss function. Thus, we can integrate such layers into a network and learn logical relations from our data.

The authors derived the back – propagation rules through their solver, and could then use it to learn two logical problems from data: the max parity problem (which counts if the given inputs are odd or even), and Sudoku puzzles, where a network learned to fill the missing numbers. This represents a step forward towards incorporating logical rules into Deep Learning models.

Run – time Monitoring

Another new area of research in AI safety is the run-time monitoring of neural networks. Here, the goal is to test whether some specified safety properties hold at run-time, while deploying the system and raising a warning in case a problem occurs.

Activation patterns: An approach towards run-time monitoring is [17], where the authors utilize the activation patterns of layers close to the output of a network. The activation pattern of a ReLU layer is a binary string, indicating whether the respective ReLU unit is active (e.g. has a positive output), or not (outputs 0). Activation patterns have been employed in some works, for example in the detection of adversarial examples. In this case, the authors propose to use them to perform run – time monitoring of a model.

Specifically, they make two hypothesis: First, network layers close to the output capture more high – level, class specific features, while layers close to the input capture more general, class – independent features. Thus, the outputs of layers close to the output should be more related with the class of the input. Due to this, they assume that the activation patterns from these, of inputs belonging to the same class, should be more similar to each other than the activation patterns of inputs belonging to different categories.

Secondly, they assume that if the model receives inputs similar to the ones in the training set, it will output similar activation patterns. On the other hand, if a network receives some input not similar to the ones it was trained on, it will produce and activation pattern that is not similar to the activation patterns of the training data (of the same class).

With these hypotheses, they store the activation patterns of a close – to – output network layer during training, and at test time, they compare the activation pattern of a test input with the stored ones. If they are highly dissimilar, they assume the network output is not reliable, and they raise a warning. The similarity between activation patterns is measured by the Hamming distance. The Hamming distance between two binary numbers is just the number of different bits. For example:

$$Ham(101, 111)=1,$$

since the 2nd bit differs. They store recorded patterns using a Binary Decision Diagram, and they define a dissimilarity threshold over the Hamming distance in a validation set, so that their run – time monitor has meaningful detection rate and low false alarm rate (if the threshold is too low the monitor will reject almost everything, if it is too high errors might pass unnoticed – the correct threshold can be found over a validation set). In case of large layers, they propose to monitor the neurons with the largest gradients or weights. The overall idea is depicted below:

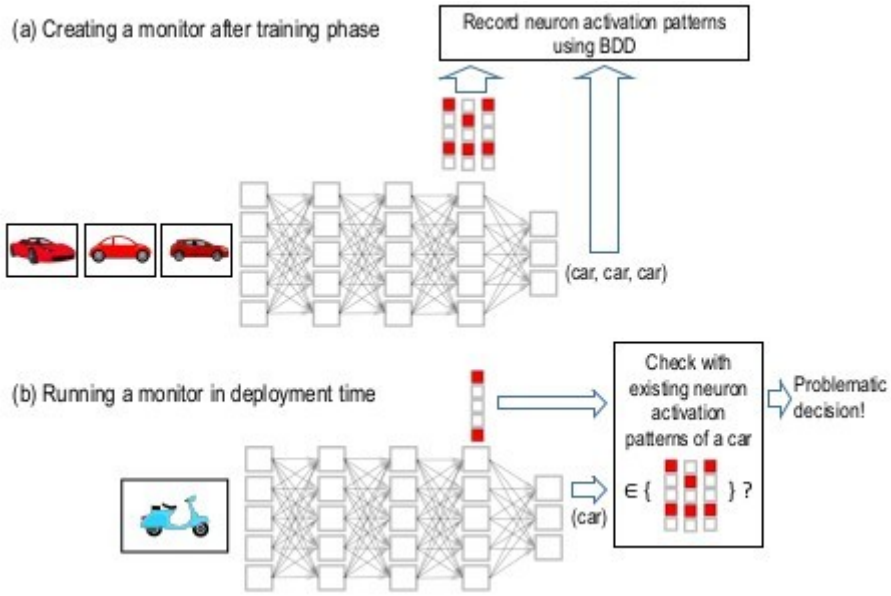

Fig. 3.14: Illustration of run-time monitoring in [17].

The authors performed experiments with the MNIST and GTSRB, and showed than their approach can detect some of the errors at test time. Unfortunately, many problematic test inputs produced patterns similar to the recorded ones, and were unnoticed. Apart from that, the method they use to calculate the Hamming distance to the training patterns is inefficient, since they store all patterns with Hamming distance up to the threshold into the decision diagram, which increase exponentially and can run the program out of memory.

Outlier Detection: Another part of AI safety, which can be considered as a part of run – time monitoring, is outlier detection. Specifically, given a model trained on a dataset with some given distribution, it may be the case that the data encountered at test time come from a different distribution. This phenomenon is called Concept Drift. Classifiers such as neural networks, trained with maximum likelihood, have no way of detecting this change, since they cannot model any uncertainties in the distribution. A model trained to recognize digits will classify a cat as a digit with high confidence. But for AI safety, such a behavior is of course problematic.

Due to this, a number of researchers proposed solutions for detecting out – of – distribution (OOD) data. Although Concept Drift has been investigated in the past, detection methods suitable for deep models with high – dimensional inputs have only recently been proposed.

Some intuitive approaches such as the following have been proposed ([86]):

- Using the entropy of the predicted class distribution, $H(y|\boldsymbol{x})=-\sum_k p(y=k|\boldsymbol{x})\log p(y=k|\boldsymbol{x})$. A high entropy indicates a more uniform, and thus more uncertain prediction. This might be an indicator of an OOD input.
- Using an ensemble of models that were trained with random initialization, and random shuffling of the training data. The ensemble outputs the class chosen by the majority of the models([89]).
- Learning a classifier that distinguishes in – distribution and out – of – distribution data, by training it on clean and perturbed data.
- Building a $K+1$ class classifier, where the K classes correspond to the dataset, and the last class is trained to detect perturbed in – distribution samples. The maximum class probability of this classifier might be indicative.

Authors in [87] proposed a method called ODIN. ODIN works as follows: first, it applied temperature scaling (as we saw in defensive distillation) to the outputted classes. Then, it applies a small perturbation to the input in the direction that increases the softmax scores. Finally, the input is fed to the network, and the maximum class score is compared to a threshold. It it is below the threshold, then the input is declared to be OOD. The rationale behind this method is that a well trained network should output a higher class score for in – distribution data, and a lower score for OOD. The temperature scaling and perturbation introduced have the effect of creating an imbalance in the softmax output, with high scores becoming higher, and low scores tending to lower. Then, the highest class score will be more indicative as in the baseline method.

Another idea was proposed in [88]. Lee et. al. calculated the mean output $\hat{\boldsymbol{\mu}}_{\boldsymbol{c}}=\frac{1}{N_c}\sum_{i:y_i=c} f(\boldsymbol{x}_{\boldsymbol{i}})$ per class and the common variance matrix, $\hat{\Sigma}=\frac{1}{N}\sum_c\sum_{i:y_i=c}(f(\boldsymbol{x}_{\boldsymbol{i}})-\hat{\boldsymbol{\mu}}_{\boldsymbol{c}})(f(\boldsymbol{x}_{\boldsymbol{i}})-\hat{\boldsymbol{\mu}}_{\boldsymbol{c}})^T$. With these, they performed Linear Discriminant Analysis (LDA), by computing the confidence score, measured with the Mahanalobis distance:

$$M(\boldsymbol{x})=max_c-(f(\boldsymbol{x})-\hat{\boldsymbol{\mu}}_{\boldsymbol{c}})^T\hat{\Sigma^{-1}}(f(\boldsymbol{x})-\hat{\boldsymbol{\mu}}_{\boldsymbol{c}})$$

This is just the exponent of the Gaussian distribution; the maximum $M(\boldsymbol{x})$ corresponds to the class giving maximum probability for the sample, so LDA will classify it as coming from class c. Then, they use the Mahanalobis distance as an indicator for OOD, by first extracting confidence bounds for it, and then using them to set a threshold. They also apply perturbations to increase class separation.

On the other hand, work [86] focuses on using generative models for OOD detection. Generative models can be used to model the distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ of the dataset. An idea proposed was to use such a model to compute the likelihood of a sample $\boldsymbol{x}$, and reject it if this likelihood is low. Surprisingly, it has been shown that generative models output high likelihoods for OOD data.

The authors in [86] believe that the explanation behind this is that models learn background information, which is common in in – distribution and OOD samples, hence simple likelihood cannot separate them. To mitigate that, they propose a method to remove this background information.
Specifically, they assume that a sample $\boldsymbol{x}$ is composed of two parts: a background part $\boldsymbol{x}_{\boldsymbol{B}}$, modeling the background information, and a semantics part $\boldsymbol{x}_{\boldsymbol{S}}$ modeling the semantic information. Assuming that they are independent, we can write:

$$p(\boldsymbol{x}) = p(\boldsymbol{x_B})\,p(\boldsymbol{x_S})$$

Assume now that we have two models, a model $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ capturing semantic information, and a model $p_{\boldsymbol{\theta_0}}(\boldsymbol{x})$ modeling background information. Assuming that the background content will be similar for both models, but the semantic information for the model capturing semantics will be higher, the log – likelihood ration will be:

$$LLR(\boldsymbol{x}) = \log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x})}{p_{\boldsymbol{\theta_0}}(\boldsymbol{x})} = \log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x_B})\,p_{\boldsymbol{\theta}}(\boldsymbol{x_S})}{p_{\boldsymbol{\theta_0}}(\boldsymbol{x_B})\,p_{\boldsymbol{\theta_0}}(\boldsymbol{x_S})} = \log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x_B})}{p_{\boldsymbol{\theta_0}}(\boldsymbol{x_B})}\cdot\log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x_S})}{p_{\boldsymbol{\theta_0}}(\boldsymbol{x_S})} \approx \log p_{\boldsymbol{\theta}}(\boldsymbol{x_S}) - \log p_{\boldsymbol{\theta_0}}(\boldsymbol{x_S})$$

They hypothesis is that *LLR* will be high for in – distribution data containing meaningful semantics, because the 1st model will output a high likelihood, while the background model will not be affected. On the other hand, an OOD will have low semantics information for the semantics model, and *LLR* will be low.

The authors train the semantics model $p_{\boldsymbol{\theta}}$ on the original dataset, and the background model $p_{\boldsymbol{\theta_0}}$ on the same dataset, with random permutations. They managed to achieve good results in identifying OOD DNA sequences, and showed also promising results on vision tasks.

Data Augmentation: A way usually employed in training Deep Learning models is data augmentation, where one increases the size of the dataset by introducing pre – processing steps such as rotations, translations, contrast – adjustment, etc., on the original images (augmentation applies more to vision tasks). The authors of [90] propose to use a data augmentation technique that manages to increase a model's robustness on data shifts or permutations. They call their method AugMix. The algorithm of this method can be seen below:

Algorithm AUGMIX Pseudocode

Input: Model $\hat{p}$, Classification Loss $\mathcal{L}$, Image x_{orig}, Operations $\mathcal{O} = \{\text{rotate}, \ldots, \text{posterize}\}$
function AugmentAndMix(x_{orig}, $k = 3, \alpha = 1$)
  Fill x_{aug} with zeros
  Sample mixing weights $(w_1, w_2, \ldots, w_k) \sim \text{Dirichlet}(\alpha, \alpha, \ldots, \alpha)$
  for $i = 1, \ldots, k$ **do**
    Sample operations $\text{op}_1, \text{op}_2, \text{op}_3 \sim \mathcal{O}$
    Compose operations with varying depth $\text{op}_{12} = \text{op}_2 \circ \text{op}_1$ and $\text{op}_{123} = \text{op}_3 \circ \text{op}_2 \circ \text{op}_1$
    Sample uniformly from one of these operations chain $\sim \{\text{op}_1, \text{op}_{12}, \text{op}_{123}\}$
    $x_{\text{aug}} \mathrel{+}= w_i \cdot \text{chain}(x_{\text{orig}})$ ▷ *Addition is elementwise*
  end for
  Sample weight $m \sim \text{Beta}(\alpha, \alpha)$
  Interpolate with rule $x_{\text{augmix}} = m x_{\text{orig}} + (1 - m) x_{\text{aug}}$
  return x_{augmix}
end function
$x_{\text{augmix1}} = \text{AugmentAndMix}(x_{\text{orig}})$ ▷ $x_{augmix1}$ *is stochastically generated*
$x_{\text{augmix2}} = \text{AugmentAndMix}(x_{\text{orig}})$ ▷ $x_{augmix1} \neq x_{augmix2}$
Loss Output: $\mathcal{L}(\hat{p}(y \mid x_{\text{orig}}), y) + \lambda\,\text{Jensen-Shannon}(\hat{p}(y \mid x_{\text{orig}}); \hat{p}(y|x_{\text{augmix1}}); \hat{p}(y|x_{\text{augmix2}}))$

Fig. 3.15: The AugMix algorithm.

The idea of this method is to explore permutations around a sample in a uniform way, by sampling operations at random, and chaining them in various lengths of 2 and 3. Then, similarly an interpolation rule with sampled weights is used. Finally, the network loss is modified, so that our network produces similar outputs for original and augmented data (the Jensen – Shannon term used

is the K-L divergence between the original and augmented output distributions). The authors show in experiments that their method improves network robustness under dataset shifts.

The above presented are some of the recent employed approaches in run – time monitoring and distribution shift detection. Run – time monitoring is a recent direction in AI safety, and we believe it to be important for the further development of the field.

Bayesian Deep Learning

Introduction

As we saw in chapter 2, Deep Learning models are trained using loss functions, which are based on the principle of maximum likelihood. Maximum likelihood does the following: out of possible distribution models $p_w(\boldsymbol{x})$, it chooses the model (parameters) that has the maximum probability of explaining our data $\boldsymbol{x} \sim D$:

$$\boldsymbol{w}^* = armax_w\, p(D|\boldsymbol{w}) = armax_w \prod_{\boldsymbol{x}^{(i)} \in D} p(\boldsymbol{x}^{(i)}|\boldsymbol{w})$$

But this approach can be problematic: First, it is unrealistic, since it seeks one single optimal parameter supposed to explain everything about the dataset, which can be far from the truth. Second, it cannot model any uncertainties we might have about the chosen model.

To illustrate this problem, let's consider the following example: suppose we have a coin with an unknown probability q of obtaining heads, and we want to estimate this probability from data. We throw the coin 3 times and obtain HHH (three heads). What does this tell us about q ?

According to Maximum Likelihood, $q = 1$! Indeed, this is the model that has the maximum probability to produce our data. We can see now better the problem of this approach: it chooses the most probable model (q) and considers it to be absolutely certain. Similarly, if we had obtained for example HHT , we would obtain $q = 2/3$ with certainty, which is again problematic. The mistake here is that there are many possible coins, and q will be a distribution; but Maximum Likelihood considers it as a fixed value.

The Bayesian approach tries to overcome this problem, by making more realistic assumptions. Namely, we assume that possible models have a prior distribution. Then, by obtaining data, we can refine this prior, and obtain a posterior distribution, which improves our estimates.

To formalize this, we suppose that possible models have a prior distribution: $\boldsymbol{w} \sim p(\boldsymbol{w})$. Then, by observing data $\boldsymbol{x} \sim D$, we can obtain a better estimate of the possible $\boldsymbol{w}$'s explaining our data, by invoking Bayes theorem:

$$p(\boldsymbol{w}|D) = \frac{p(D|\boldsymbol{w})\,p(\boldsymbol{w})}{p(D)} = \frac{p(D|\boldsymbol{w})\,p(\boldsymbol{w})}{\int p(D|\boldsymbol{w})\,p(\boldsymbol{w})\,d\boldsymbol{w}} \propto p(D|\boldsymbol{w})\,p(\boldsymbol{w})$$

To go back to our coin example, the Bayesian treatment is the following: there is some distribution $p(q)$ of fair and unfair coins in the world – we don't know which one we have, all we can say is that our coin has heads frequency q with distribution $p(q)$. Then, we can throw the coin some times and obtain some data D . Now, we can use this data to obtain a better distribution of the possible coins we have by using the above formula: $Pr[q|D] \propto p(D|q)\,p(q)$. The more data we obtain, the better estimates we get.

So, in the Bayesian setting, we do not learn a model with specific parameters, but a distribution of possible models. This distribution improves as we observe more data. Similarly, at interference, we do not use a single model to make predictions; rather, we calculate the most probable prediction given the distribution of possible models:

$$p(y|\boldsymbol{x},D)=E_{\boldsymbol{w}\sim p(\boldsymbol{w}|D)}[p(y|\boldsymbol{x};\boldsymbol{w})]=\int p(y|\boldsymbol{x};\boldsymbol{w})\,p(\boldsymbol{w}|D)\,d\boldsymbol{w}$$

This process is called Bayesian interference.

Using the Bayesian approach enables us to avoid over-fitting and estimate the uncertainties in our predictions. We can do that since we now have a distribution over possible models instead of a single one. The negative side is that performing the above computations is intractable, and even approximating them is challenging for models with millions of parameters. But due to the advantages of Bayesian learning, researchers have recently been trying to apply Bayesian ideas in Deep Learning.

Training methods

Variational Interference: The starting point to make the problem more tractable is called variational interference. Namely, we introduce an approximating variational distribution $q(\boldsymbol{w}|\theta)$ with parameters θ that approximates our model distribution $p(\boldsymbol{w}|D)$. Variational learning determines θ by minimizing the KL divergence between the two distributions:

$$\theta^{*}=argmin_{\theta}D_{KL}[q(\boldsymbol{w}|\theta)\|p(\boldsymbol{w}|D)]=argmin_{\theta}\int q(\boldsymbol{w}|\theta)\log\frac{q(\boldsymbol{w}|\theta)}{p(\boldsymbol{w}|D)}d\boldsymbol{w}$$

$$=argmin_{\theta}\int q(\boldsymbol{w}|\theta)\log\frac{q(\boldsymbol{w}|\theta)}{p(\boldsymbol{w})\,p(D|\boldsymbol{w})}d\boldsymbol{w}$$

$$=argmin_{\theta}\;D_{KL}[q(\boldsymbol{w}|\theta)|p(\boldsymbol{w})]-E_{q(\boldsymbol{w}|\theta)}[\log p(D|\boldsymbol{w})]$$

The last quantity, $F(D,\theta)=D_{KL}[q(\boldsymbol{w}|\theta)|p(\boldsymbol{w})]-E_{q(\boldsymbol{w}|\theta)}[\log p(D|\boldsymbol{w})]$, is called variational free energy. Bayesian variational learning amounts to minimizing this loss function. The prior distribution $p(\boldsymbol{w})$ is usually take to be some Gaussian. Calculating quantities involving integrals is generally not possible in closed form or intractable, so Monte Carlo sampling techniques are usually employed.

Bayes by Backprop: In [91], the authors propose a method to learn Bayesian model by extending the back – propagation algorithm. The idea is first to approximate $F(D,\theta)$ by Monte Carlo summation, as:

$$F(D,\theta)=\int q(\boldsymbol{w}|\theta)\log\frac{q(\boldsymbol{w}|\theta)}{p(\boldsymbol{w})\,p(D|\boldsymbol{w})}d\boldsymbol{w}\approx\frac{1}{n}\sum_{i=1}^{n}f(\boldsymbol{w}^{(i)},\theta),$$

where $f(\boldsymbol{w},\theta)=\log q(\boldsymbol{w}|\theta)-\log p(\boldsymbol{w})\,p(D|\boldsymbol{w})$, and samples $\boldsymbol{w}^{(i)}$ are drawn from the distribution $q(\boldsymbol{w}|\theta)$. Next, they model each weight as having a Gaussian distribution with some mean and weight, i.e. the parameters θ are: $\theta=(\boldsymbol{\mu},\boldsymbol{\sigma})$, where $\boldsymbol{\sigma}=\log(1+\exp(\boldsymbol{\rho}))$ are point-wise standard deviations. $q(\boldsymbol{w}|\theta)$ is modeled thus to be a Gaussian, $q(\boldsymbol{w}|\theta)=\boldsymbol{\mu}+\boldsymbol{\sigma}\odot\boldsymbol{\epsilon}$, with $\boldsymbol{\epsilon}\sim N(\boldsymbol{0},I)$, and $\odot$ devotes element-wise multiplication. Finally, they prove that the expectation over gradients of

f is equivalent to the gradient of f 's expectation, e.g. $F(D,\theta)$. Thus, they learn the parameters $\theta=(\boldsymbol{\mu},\boldsymbol{\sigma})$ using the following back – propagation algorithm:

1. Sample $\boldsymbol{\epsilon}\sim N(\mathbf{0},I)$
2. Compute $\boldsymbol{w}=\boldsymbol{\mu}+\boldsymbol{\sigma}\odot\boldsymbol{\epsilon}$ (weights sample under $q(\boldsymbol{w}|\theta)$, $\theta=(\boldsymbol{\mu},\boldsymbol{\sigma})$)
3. Compute gradients: $\Delta_{\boldsymbol{\mu}}=\frac{\partial f}{\partial \boldsymbol{\mu}}$, $\Delta_{\boldsymbol{\rho}}=\frac{\partial f}{\partial \boldsymbol{\rho}}$
4. Update: $\boldsymbol{\mu}\leftarrow\boldsymbol{\mu}-a\Delta_{\boldsymbol{\mu}}$, $\boldsymbol{\rho}\leftarrow\boldsymbol{\rho}-a\Delta_{\boldsymbol{\rho}}$

Other similar methods to train Bayesian network have been proposed as well in the literature, for example in [92], which was the starting point for [91].

Monte Carlo Dropout (MCD): This is an approximate variational interference method based on the dropout method, as shown in [64]. Namely, samples from the distribution $p(\boldsymbol{w}|D)$ are extracted at test time, by applying dropout on the network and taking sample outputs (the network needs also to be trained by dropout and stochastic gradient descent). The idea of this equivalence is that the approximate distribution $q(\boldsymbol{w}|\theta)$ can be modelled by placing Bernoulli distributions over the weights.

Hamiltonian Monte Carlo (HMC): This method ([93], [23]) defines a Markov chain, which has an invariant distribution equal to $p(\boldsymbol{w}|D)$, and it uses methods of Hamiltonian dynamics to increase exploration efficiency. Contrary to the other methods above, HMC doesn't make any assumption on the form of the posterior distribution. The output is a set of samples $\boldsymbol{w_i}$ that approximate $p(\boldsymbol{w}|D)$ (and converge in the limit).

Modeling Uncertainties

The advantage of Bayesian learning is the ability to model uncertainties. Generally, we can separate two kinds of uncertainties:

- Epistemic uncertainty: This is our uncertainty regarding the model parameters. For example, if the weights distribution $p(\boldsymbol{w}|D)$ is nearly uniform, then our uncertainty regarding the correct model parameters is large. Epistemic uncertainty can generally be reduced by training on additional data.
- Aletoric uncertainty: Aletoric uncertainty is uncertainty inherit in the data, in the form of noise. It can be further separated in two types: Homoscedastic uncertainty, where the noise distribution is the same for all data points (for example, measurement noise), and Heteroscedastic uncertainty, where the noise distribution can be different on each data point.

Bayesian learning can be used to estimate epistemic uncertainty: since we have a distribution $p(\boldsymbol{w}|D)$ of possible models, we can sample possible output vectors from them, and them estimate the uncertainty using the variance of these predictions, or the entropy of the probability vectors. Furthermore, there are also works such as [95], that propose methods to estimate aletoric uncertainty as well, and jointly learn them with a common loss function.

Bayesian deep learning has been proposed as a possible solution for safety in automated driving as well. For example, in [96], authors propose an automatic driving pipeline based on Bayesian Deep Learning. By having the ability to estimate uncertainties, they claim that safety issues can be modeled and tackled better than conventional Deep Learning approaches. They show a scenario depicted in the figure below: Here, a Deep Learning system doesn't detects another car's indicator, which leads to a crash, since the system assumes that the other car will not turn. On the other hand,

the Bayesian system below also doesn't detect the indicator, but it also outputs a high uncertainty. Do to this, there is an uncertainty about the other car's trajectory, so the Bayesian – based system slows down, and the crash is avoided.

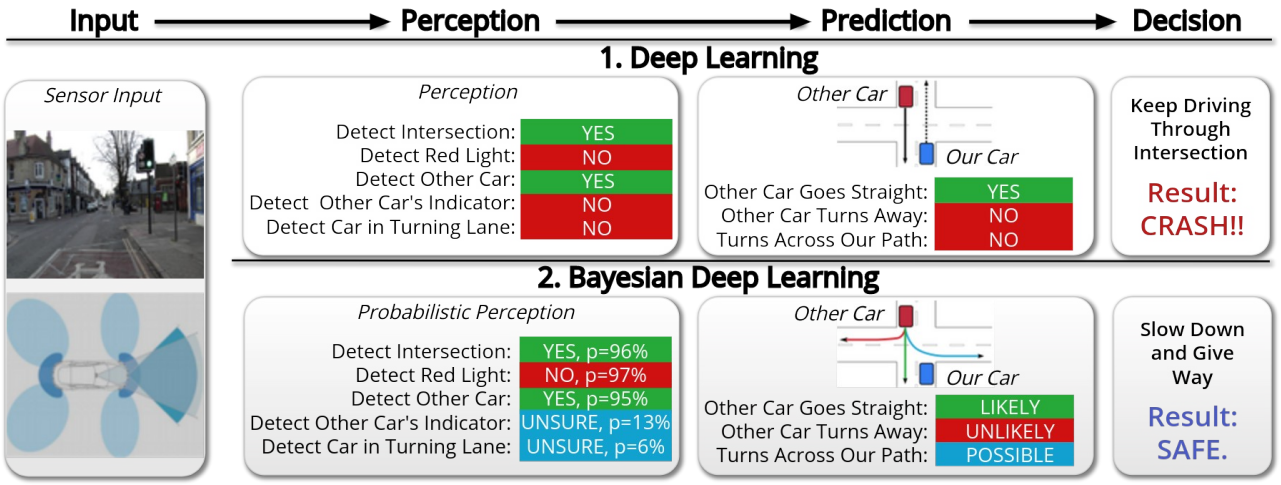

Fig. 3.16: A comparison between a conventional and a Bayesian Deep Learning system, in a hypothetical dangerous situation (from [96]).

In another work ([97]), authors employ Bayesian techniques to estimate the uncertainties in classification and bounding box regression outputs of a neural network intended to be used for automated driving. [23] proposes using Bayesian networks to construct end – to – end automated car controllers which can provide statistical guarantees.

The challenge in Bayesian Deep Learning is the difficulty in training and scaling – up, in comparison with standard approaches. But researchers are working on these, as it forms an important direction towards safer AI systems.

Testing

In many cases, formal verification methods as some presented previously are either inefficient or impossible. In this cases testing is used, which attempt to find erroneous or problematic inputs by efficient search methods. The idea is to try to extend testing methods in software for neural networks.

Coverage Criteria: In software testing, one of the goals of a test suit is that the test cases cover a set of criteria to a large extend. This increases the reliability of the testing method. Thus, methods to generate test cases covering the specifications. Similarly, in the case of neural networks, a set of plausible coverage criteria have been proposed.

Neuron Coverage: A neuron n in a network is covered by a test case $\boldsymbol{x}$ if the neuron's output is positive when $\boldsymbol{x}$ is given as input to the network. This applies especially for ReLU neurons. The intuition is that different neurons recognize different features in the input. Erroneous behaviors might be triggered by such neurons recognizing – being activated. Under normal circumstances, the probability of this to happen in a regular test dataset might be low, and thus errors might remain unnoticed. But a testing method should be able to detect those cases.

The concept of neuron coverage was introduced in [98], where authors proposed DeepXplore, a testing method to generate problematic test cases for neural networks. Specifically, DeepXplore takes a number of similar neural networks as an input (performing the same functionality) and tries

to detect test cases where network outputs differ. These samples will lie between the decision boundaries of the various networks, as shown below:

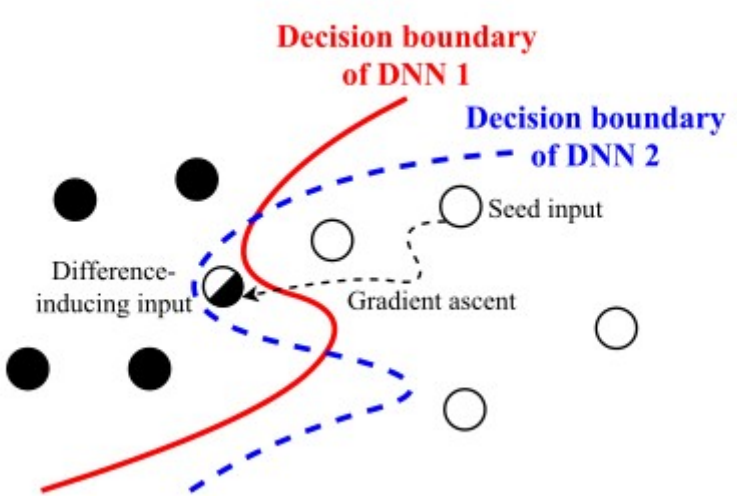

Fig. 3.17: Problematic inputs lying between decision boundaries (from [98]).

DeepXplore attempts to find these samples by performing gradient ascent to a loss function defining the prediction difference between networks. At the same time, is tires to find samples that increase neuron coverage. The combined optimization objective can be formulated as follows:

$$obj(\boldsymbol{x}, n_{j,i}) = (\sum_{k \neq j} f_c^k(\boldsymbol{x}) - \lambda_1 f_c^j(\boldsymbol{x})) + \lambda_2 u_{j,i}(\boldsymbol{x})$$

Here, $f_c^k(\boldsymbol{x})$ devotes the probability for class c of the k-th network, $u_{j,i}$ is the activation of some neuron i in the j-th network, while $\lambda_{1,2}$ are weights controlling the importance of each term. So, this loss function tries to increase the prediction disagreement $\sum_{k \neq j} f_c^k(\boldsymbol{x}) - \lambda_1 f_c^j(\boldsymbol{x})$, while also increasing the coverage, by increasing the neuron activation, $u_{j,i}$.

The authors used this method to find problematic test cases for various neural networks and datasets.

Neuron Coverage Extensions: In [99], extensions of the previous neuron coverage metric are proposed. They define various coverage criteria in the neuron and layer levels.

- k **- multi-section Neuron Coverage**: for a test set T and a neuron n, let $[l_n, h_n]$ be the interval of activations of n. Then, the authors divide this interval in k equal sections, with S_i^n devoting the i-th section, $1 \leq i \leq k$. Then, the k multi-section neuron coverage is defined as the ration of the intervals covered by the test set divided by k:

$$\frac{\left|\{S_i^n | \exists \boldsymbol{x} \in T : n(\boldsymbol{x}) \in S_i^n\}\right|}{k}$$

  Similarly, they define the k - multi-section neuron coverage of a neural network as:

$$KMNCov(T, k) = \frac{\sum_{n \in N} \left|\{S_i^n | \exists \boldsymbol{x} \in T : n(\boldsymbol{x}) \in S_i^n\}\right|}{k \cdot |N|}$$

Moreover, they define corner cases when for an input $\boldsymbol{x}$ we have $x \in (-\infty, l_n) \cup (h_n, +\infty)$. This means that the neuron activation lie outside their normal range. The total boundary coverage for the test set is defined as:

$$NBCov(T) = \frac{|UpperCornerNeuron| + |LowerCornerNeuron|}{2 \cdot |N|}$$

Similarly we can define upper boundary coverage, lower boundary coverage, etc. Next, authors consider coverage metrics for entire layers, and define the following metrics:

- **Top-k neuron coverage:** this measures the percentage of neurons that have been among the top-k neurons at some instance in their respective layer. This means that these neuron's activations were among the top-k for that layer. Formally:

$$TKNCov(T, k) = \frac{|\cup_{x \in T} (\cup_{1 \le i \le l} top_k(\boldsymbol{x}, i))|}{|N|}$$

- **Top-k neuron patterns:** Given a test input $\boldsymbol{x}$, the authors define the top-k neuron patterns as the sets of top-k active neurons in each layer. Then, for the entire test dataset T, they define the number of top-k neuron patterns as:

$$TKNPat(T, k) = |top_k(\boldsymbol{x}, 1), .., top_k(\boldsymbol{x}, l) | \boldsymbol{x} \in T|$$

The authors perform experiments using the above metrics, and find that adversarial attacks increase the coverage metrics on neural networks,, suggesting that adversarial attacks explore further regions in the network.

Modified Condition / Decision Coverage (MC / DC): This is a testing method for safety – critical software. The idea is that all possible conditions leading to a certain decision must be tested. Variations of this concept for neural networks have been proposed in [100-101], where relations between neurons are tested, e.g. the effect of some neurons on subsequent neurons. The intuition is that we should not only test for features, but also the effect of simple features on more complex features.

Let Ψ_k be a collection of neurons at layer k. Then, the authors define the following properties:

- **Sign Change:** Given two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$ and a feature $\psi_{k,l}$, they say that the sign change of $\psi_{k,l}$ is exploited by $\boldsymbol{x_1}, \boldsymbol{x_2}$ and devote it as $sc(\psi_{k,l}, \boldsymbol{x_1}, \boldsymbol{x_2})$ if

$$sign(n_{k,j}, \boldsymbol{x_1}) \neq sign(n_{k,j}, \boldsymbol{x_2}), \forall n_{k,j} \in \psi_{k,l}$$

The converge, $sign(n_{k,j}, \boldsymbol{x_1}) = sign(n_{k,j}, \boldsymbol{x_2}), \forall n_{k,j} \in \psi_{k,l}$, is devoted as $nsc(\psi_{k,l}, \boldsymbol{x_1}, \boldsymbol{x_2})$.

- **Value Change:** Given two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$, a feature $\psi_{k,l}$, and a value function g, that value change of $\psi_{k,l}$ is exploited by $\boldsymbol{x_1}, \boldsymbol{x_2}$ with respect to g, devoted as $vc(\psi_{k,l}, \boldsymbol{x_1}, \boldsymbol{x_2})$, if $g(\psi_{k,l}, \boldsymbol{x_1}, \boldsymbol{x_2}) = true$.

Then, based on these concepts, the authors present four different coverage criteria, defined as follows:

- **Sign – Sign Coverage (SS Cov.):** A feature pair $a=(\psi_{k,i}, \psi_{k+1,j})$ is said to be SS-covered by two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$, written as $SS(a, \boldsymbol{x_1}, \boldsymbol{x_2})$ if $sc(\psi_{k,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ and $nsc(P_k / \psi_{k,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ and $sc(\psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ are satisfied, where L_k is the set of neurons at layer k. That is, the sign change of $\psi_{k,l}$ (and only that) induces a sign change on $\psi_{k+1,l}$.

- **Sign – Value Coverage (SV Cov.):** A feature pair $a=(\psi_{k,i}, \psi_{k+1,j})$ is said to be SV-covered by two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$ and a value function g, written as $SV^g(a, \boldsymbol{x_1}, \boldsymbol{x_2})$ if $sc(\psi_{k,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$, $nsc(P_k / \psi_{k,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$, $vc(g, \psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ and $nsc(\psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ are all satisfied.

- **Value – Sign Coverage (VS Cov.):** A feature pair $a=(\psi_{k,i}, \psi_{k+1,j})$ is said to be VS-covered by two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$ and a value function g, written as $VS^g(a, \boldsymbol{x_1}, \boldsymbol{x_2})$ if $nsc(L_k, \boldsymbol{x_1}, \boldsymbol{x_2})$, $vc(g, \psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ and $sc(\psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ are all satisfied.

- **Value – Value Coverage (VV Cov.):** A feature pair $a=(\psi_{k,i}, \psi_{k+1,j})$ is said to be VV-covered by two test cases $\boldsymbol{x_1}, \boldsymbol{x_2}$ and value functions g_1, g_2, written as $VV^{g_1,g_2}(a, \boldsymbol{x_1}, \boldsymbol{x_2})$, if $vc(g_1, \psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$, $nsc(L_k, \boldsymbol{x_1}, \boldsymbol{x_2})$, $vc(g_2, \psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ and $nsc(\psi_{k+1,i}, \boldsymbol{x_1}, \boldsymbol{x_2})$ are all satisfied.

For the test case generation, authors employ a method based on Linear Programming, and a Concolic testing algorithm in their subsequent work.

Surprise Coverage: In [102], the authors introduce a coverage metric based on a measure of "surprise" that a given test input $\boldsymbol{x}$ creates. Specifically, they measure the difference between the activation patterns produced by $\boldsymbol{x}$ and the patterns produced by the training set. They do that by computing the following value:

$$sur(\boldsymbol{x}) = -\log\left\{\frac{1}{|X|}\sum_{\boldsymbol{x_i}\in X} K_H(\boldsymbol{u_k}(\boldsymbol{x_k}) - \boldsymbol{u_k}(\boldsymbol{x_i}))\right\}$$

Here $\boldsymbol{u_k}$ is a vector of activation values at layer k, X is the training set, and $K_H = (2\pi)^{-d/2}|H|^{-1/2}\exp(-\frac{1}{2}\boldsymbol{x}^T H^{-1}\boldsymbol{x})$ is a Gaussian kernel function, used in multivariate Kernel Density Estimation. Then, they divide the range of possible surprises U into n buckets, and measure the amount of surprise buckets that a test set covers, similar to [99]. The authors perform various experiments to test the soundness of the method, by showing that samples with higher surprise are harder to classify.

Quantitative Projection Coverage: In [21], authors assume that there are some weighted criteria that describe the operation conditions, such as for example weather, road conditions etc. for autonomous driving (which is their test case of interest). With these criteria, their method partitions the input data in such a way that each partition, based on some combination of the criteria, has data points at least equal to the corresponding weight. In this way, they build test sets that obey to the specified conditions. Their method is based on Binary Integer Linear Programming.

Fuzzing: In automated software testing, fuzzing refers to a set of methods that generate random input data for testing, by modifying an existing input set. The system is then tested against the generated data. In [103], Odena et. al. try to extend these ideas for neural networks, by introducing their TensorFuzz method. TensorFuzz tries to create and detect problematic inputs for a DNN, that create numerical errors or disagreements between networks, or some other constraint given by the

user. It achieves that by performing random mutations over selected test inputs, analyzing the outcome, and measure coverage by an approximate nearest neighbor algorithm, which takes network activations as input and checks their distance from previous activations. If it exceeds some threshold, we consider the network to be in a new “state”, and the coverage has increased.

Testing is an important research area in AI safety, since it can potentially offer practical solutions in cases where theoretical guarantees are difficult or impossible. Apart from the approaches mentioned above, other suggestions have also been proposed, for example using GANs for testing ([104]), a method called concolic testing in [101], which combines testing with symbolic execution methods to find test cases, and others.

Interpretability – Explainability

Interpretability – explainability is a field of AI safety aiming to interpret the decisions made by a DNN. It aims to “open the black box” and make Deep Learning systems more understandable for human users. Apart from that, insights from this area might be helpful in other fileds of AI safety, such as verification or testing.

Existing literature in interpretability can be classified in the following main categories: Instant-wise explanations, where we seek to explain the decision of a network on a particular input, model explanations, which attempt to understand the model, and explanation methods based on the training process. Below, we will present a brief review of these approaches.

Visualization – based methods
This set of methods attempts to interpret DNN decisions by producing diagrams and visualizations, which can provide insights on their behavior. They apply mostly on CNNs and vision related tasks.

A standard approach, first suggested in [106], is to determine inputs which cause high activation on a certain neuron at some layer. This can be done by conventional gradient descend, where now gradients are taken with respect to the input (instead of the weights) and the optimization objective is the output of the neuron under consideration. With this technique, we can get an idea on the input features that cause that neuron to activate.

Many further approaches based on visualization techniques have been proposed. For example, authors in [107] try to invert the representation function of a neural network, in order to inspect the transformations it performs. In [108], the gradient of a given class score with respect to the input, $\partial S_c(\boldsymbol{x})/\partial \boldsymbol{x}$ is calculated. Linearizing the model around $\boldsymbol{x}$, $S_c(\boldsymbol{x}) \approx \boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b}$, with $\partial S_c(\boldsymbol{x})/\partial \boldsymbol{x} = \boldsymbol{w}$, the gradient shows us how the change of each input pixel affects the change in classification. Sometimes, this gradient is multiplied element – wise with the input image, to overlay the contribution of each input pixel. In [109], authors use deconvolutional layers to detect the image region causing a given neuron to activate. Other ideas proposed is Gradient-weighted Class Activation Mapping (GradCAM, [110]), Deep Learning Important FeaTures (DeepLift, [111]), Integrated Gradients ([112]), and many others. A very good and interactive summary can be found at the digital publication of distill in [113].

Model explanation methods
These methods attempt to explain a complex model by various methods, for example by approximating it with simpler models, extracting rules from it, or with other methods.

For example, in [114], authors attempt to extract a decision tree representation of a neural network, obtaining rules from its parameters. Similarly, authors in [115] treat ReLU – based DNNs as a set of

locally linear classifiers, where each region is defined by the ReLU conditions. On the other hand, the authors in [116] try to extract local decision rules near the network inputs. They define the concept of anchor rules, which are rules that should explain the models behavior near some input $\boldsymbol{x}$ with high confidence.

Explanations based on Training
These methods try to provide model explanations by looking at the training process. An example of this is the work in [117], where authors attempt to explain the training process of a DNN using ideas from information theory.

Safety in Reinforcement Learning

Finally, another branch of AI safety is about safety in Reinforcement Learning. This concerns questions about safe exploration by agents, discovering faulty agent behaviors that have low probability (and thus cannot be detected with naive testing), and more. Naturally, RL uses DNNs, so a large part of the concepts in neural network safety still applies. We will not go further into this topic, since it is outside the scope of this survey. Interested readers are referred to [105] for further information.

Summary

In these notes we have tried to summarize the main ideas and approaches in the important field of AI safety, aiming to enable the further adoption of AI methods in safety critical systems, and also to improve and understand the science of AI in general. This is a dynamically evolving field with many ideas being proposed every day. One should look at current conferences and publications in order to keep up with new developments.

References

[1] P. Ortega, V. Maini. “Building safe artificial intelligence: specification, robustness, and assurance”. Deep Mind blog, 09/2018. Available at: https://medium.com/@deepmindsafetyresearch/building-safe-artificial-intelligence-52f5f75058f1

[2] V. Krakovna. “AI safety resources”. Blog post, 2019. Available at: https://vkrakovna.wordpress.com/ai-safety-resources/

[3] M. Brundage et al.. “The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation”. Executive summary, 02/2018. Available at: https://arxiv.org/pdf/1802.07228.pdf

[4] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, X. Yi. “A Survey of Safety and Trustworthiness of Deep Neural Networks”. 2018 – 19. Available at: https://arxiv.org/abs/1812.08342

[5] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, R. Fergus. “Intriguing properties of neural networks”. ICLR, 2014. Available at: https://arxiv.org/abs/1312.6199

[6] M. Wu, M. Wicker, W. Ruan, X. Huang, M. Kwiatkowska. “A game-based approximate verification of deep neural networks with provable guarantees”. Th. CS, 2020. Available at: https://arxiv.org/pdf/1807.03571.pdf

[7] L. Engstrom, D. Tsipras, L. Schmidt, A. Madry. “A rotation and a translation suffice: Fooling cnns with simple transformations”, 2017. Available at: https://arxiv.org/pdf/1712.02779.pdf

[8] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, A. Vladu. “Towards deep learning models resistant to adversarial attacks”. ICLR, 2018. Available at: https://arxiv.org/pdf/1706.06083.pdf

[9] G. Katz, C. Barrett, D. L Dill, K. Julian, M. J. Kochenderfer. “Reluplex: An efficient SMT solver for verifying deep neural networks”. ICAV, 2017. Available at: https://arxiv.org/abs/1702.01135

[10] V. Tjeng, K. Y. Xiao, R. Tedrake. “Evaluating robustness of neural networks with mixed integer programming”. ICLR, 2019. Available at: https://arxiv.org/abs/1711.07356

[11] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. “Ai2: Safety and robustness certification of neural networks with abstract interpretation”. IEEE Symp. on Sec. and Priv., 2018. Available at: https://files.sri.inf.ethz.ch/website/papers/sp2018.pdf

[12] G. Singh, T. Gehr, M. Puschel, M. Vechev. “An abstract domain for certifying neural networks”. POPL, 2019. Available at: https://files.sri.inf.ethz.ch/website/papers/DeepPoly.pdf

[13] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, T. Mann, P. Kohli. “On the effectiveness of interval bound propagation for training verifiably robust models”. 2018. Available at: https://arxiv.org/abs/1810.12715.

[14] A. Raghunathan, J. Steinhardt, P. Liang. “Certified defenses against adversarial examples”. ICLR, 2018. Availabe at: https://arxiv.org/abs/1801.09344

[15] K. Dvijotham, S. Gowal, R. Stanforth, R. Arandjelovic, B. O'Donoghue, J. Uesato, and P. Kohli. "Training verified learners with learned verifiers". 2018. Available at: https://arxiv.org/abs/1805.10265.

[16] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, M. Vechev. "DL2: Training and Querying Neural Networks with Logic". ICML 2019. Available at: https://files.sri.inf.ethz.ch/website/papers/icml19-dl2.pdf

[17] C.-H. Cheng, G. Nührenberg, H. Yasuoka. "Runtime Monitoring Neuron Activation Patterns". DATE 2019. Available at: https://arxiv.org/abs/1809.06573

[18] R. Ashmore, R. M. Hill. "Boxing clever: Practical techniques for gaining insights into training data and monitoring distribution shift". AISE, 2018. Available at: https://link.springer.com/chapter/10.1007%2F978-3-319-99229-7_33

[19] M. Abbasi, A. Rajabi, A. S. Mozafari, R. B. Bobba, C Gagne. "Controlling Over-generalization and its Effect on Adversarial Examples Generation and Detection". CoRR, 2018. Available at: https://arxiv.org/abs/1808.08282

[20] J. G. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. V. Chawla, F. Herrera. "A unifying view on dataset shift in classification". Pattern Recogn., 45(1):521–530, 2012. Available at: https://www3.nd.edu/~dial/publications/moreno2012unifying.pdf

[21] C.-H. Cheng, C.-H. Huang, H. Yasuoka, "Quantitative Projection Coverage for Testing ML-enabled Autonomous Systems". CoRR, 2018. Available at: https://arxiv.org/abs/1805.04333

[22] H. Wang, D.-Y. Yeung. "Towards Bayesian Deep Learning: A Survey". CoRR, 2016. Available at: https://arxiv.org/pdf/1604.01662.pdf

[23] R. Michelmore, M. Wicker, L. Laurenti, L. Cardelli, Y. Gal, M. Kwiatkowska, "Uncertainty Quantification with Statistical Guarantees in End-to-End Autonomous Driving Control". ICRA, 2020. Available at: https://arxiv.org/pdf/1909.09884.pdf

[24] S. G. Finlayson, H. W. Chung, I. S. Kohane, A. L. Beam. "Adversarial Attacks Against Medical Deep Learning Systems". CORR, 2018. Available at: https://arxiv.org/abs/1804.05296

[25] J. Ebrahimi, A. Rao, D. Lowd, D. Dou. "HotFlip: White-Box Adversarial Examples for Text Classification". ACL, 2018. Available at: https://arxiv.org/abs/1712.06751

[26] A. Chakraborty, M. Alam, Vi. Dey, A. Chattopadhyay, D. Mukhopadhyay. "Adversarial Attacks and Defences: A Survey". CORR, 2018. Available at: https://arxiv.org/abs/1810.00069

[27] I. J. Goodfellow, J. Shlens, C. Szegedy. "Explaining and Harnessing Adversarial Examples". ICLR, 2015. Available at: https://arxiv.org/abs/1412.6572

[28] A. Kurakin, I. Goodfellow, S. Bengio. "Adversarial examples in the physical world". ICLR, 2017. Available at: https://arxiv.org/abs/1607.02533

[29] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, P. McDaniel. "Ensemble Adversarial Training: Attacks and Defenses". ICLR, 2018. Available at: https://arxiv.org/abs/1705.07204

[30] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, A. Swami. “Practical Black-Box Attacks against Machine Learning”. AsiaCCS, 2017. Available at: https://arxiv.org/abs/1602.02697

[31] N. Carlini, D. Wagner. “Towards Evaluating the Robustness of Neural Networks”. IEEE Symp. on Sec. and Priv., 2017. Available at: https://arxiv.org/abs/1608.04644

[32] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, A. Swami. “The limitations of deep learning in adversarial settings”. EuroS&P, 2016. Available at:

[33] S. Baluja, I. Fischer. “Adversarial Transformation Networks: Learning to Generate Adversarial Examples”. Available at: https://arxiv.org/abs/1703.09387

[34] S.-M. M.-Dezfooli, A. Fawzi, P. Frossard. “DeepFool: a simple and accurate method to fool deep neural networks”. CVPR, 2016. Available at: https://arxiv.org/abs/1511.04599

[35] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, D. Song. “Spatially Transformed Adversarial Examples”. ICLR, 2018. Available at: https://arxiv.org/abs/1801.02612

[36] J. Su, D. V. Vargas, S. Kouichi. “One pixel attack for fooling deep neural networks”. IEEE Trans. Evolutionary Computation , 2019. Available at: https://arxiv.org/abs/1710.08864

[37] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, C.-J. Hsieh. “ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models”. AISec, 2017. Available at: https://arxiv.org/abs/1708.03999

[38] S. Sabour, Y. Cao, F. Faghri, D. J. Fleet. “Adversarial Manipulation of Deep Representations”. ICLR, 2016. Available at: https://arxiv.org/abs/1511.05122

[39] A. Rozsa, E. M. Rudd, and T. E. Boult. “Adversarial diversity and hard positive generation”. CVPR, 2016. Available at: https://arxiv.org/abs/1605.01775

[40] G. D. Evangelidis, E. Z. Psarakis, “Parametric image alignment using enhanced correlation coefficient maximization”. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2008.

[41] J. R. Flynn, S. Ward, J. Abich, D. Poole. “Image quality assessment using the ssim and the just noticeable difference paradigm”. Intern. Conf. on Engineering Psychology and Cognitive Ergonomics, 2013.

[42] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, A. Swami. “Practical Black-Box Attacks against Machine Learning”. AsiaCCS, 2017. Available at: https://arxiv.org/abs/1602.02697

[43] W. Brendel, J. Rauber, M. Bethge. “Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models”. ICLR, 2018. Available at: https://arxiv.org/abs/1712.04248

[44] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, P. Frossard. “Universal adversarial perturbations”. CVPR, 2017. Available at: https://arxiv.org/abs/1610.08401

[45] J. Steinhardt, P. W. Koh, P. Liang. “Certified Defenses for Data Poisoning Attacks”. NIPS, 2017. Available at: https://arxiv.org/abs/1706.03691

[46] T. Gu, B. Dolan-Gavitt, S. Garg. “Badnets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain”. Arxiv preprint, 2017. Available at: https://arxiv.org/abs/1708.06733

[47] X. Chen, C. Liu, B. Li, K. Lu, D. Song. “Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning”. Available at: https://arxiv.org/abs/1712.05526

[48] A. Kurakin, I. Goodfellow, S. Bengio. “Adversarial Machine Learning at Scale”. ICLR, 2017. Available at: https://arxiv.org/abs/1611.01236

[49] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, P. McDaniel. “Ensemble Adversarial Training: Attacks and Defenses”. ICLR, 2018. Available at: https://arxiv.org/abs/1705.07204

[50] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, A. Vladu. “Towards Deep Learning Models Resistant to Adversarial Attacks”. ICLR, 2018. Available at: https://arxiv.org/abs/1706.06083

[51] H. Kannan, A. Kurakin, I. Goodfellow. “Adversarial Logit Pairing”. Arxiv preprint. Available at: https://arxiv.org/abs/1803.06373

[52] L. Engstrom, A. Ilyas, A. Athalye. “Evaluating and Understanding the Robustness of Adversarial Logit Pairing”. Arxiv preprint. Available at: https://arxiv.org/abs/1807.10272

[53] C. Guo, M. Rana, M. Cisse, L. van der Maaten. “Countering Adversarial Images using Input Transformations”. ICLR, 2018. Available at: https://arxiv.org/abs/1711.00117

[54] Y. Song, T. Kim, S. Nowozin, S. Ermon, N. Kushman. “PixelDefend: Leveraging Generative Models to Understand and Defend against Adversarial Examples”. ICLR, 2018. Available at: https://arxiv.org/abs/1710.10766

[55] A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, K. Kavukcuoglu. “Conditional Image Generation with PixelCNN Decoders”. NIPS, 2016. Available at: https://arxiv.org/abs/1606.05328\

[56] A. Athalye, N. Carlini, D. Wagner. “Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples”. ICML, 2018. Available at: https://arxiv.org/abs/1802.00420

[57] J. Uesato, B. O'Donoghue, A. van den Oord, P. Kohli. “Adversarial Risk and the Dangers of Evaluating Against Weak Attacks”. ICML, 2018. Available at: https://arxiv.org/abs/1802.05666

[58] N. Papernot, P. McDaniel, X. Wu, S. Jha, A. Swami. “Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks”. IEEE symp. On Sec. and Priv., 2016. Available at: https://arxiv.org/abs/1511.04508

[59] G. Hinton, O. Vinyals, J. Dean. “Distilling the Knowledge in a Neural Network”. Arxiv preprint, 2015. Available at: https://arxiv.org/abs/1503.02531

[60] G. S. Dhillon, K. Azizzadenesheli, Z. C. Lipton, J. Bernstein, J. Kossaifi, A. Khanna, A. Anandkumar. “Stochastic Activation Pruning for Robust Adversarial Defense”. ICLR, 2018. Available at: https://arxiv.org/abs/1803.01442

[61] J. H. Metzen, T. Genewein, V. Fischer, B. Bischoff. “On Detecting Adversarial Perturbations”. ICLR, 2017. Available at: https://arxiv.org/abs/1702.04267

[62] N. Carlini, D. Wagner. “Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods”. AISec@CCS, 2017. Available at: https://arxiv.org/abs/1705.07263

[63] R. Feinman, R. R. Curtin, S. Shintre, A. B. Gardner. “Detecting Adversarial Samples from Artifacts”. Arxiv preprint, 2017. Available at: https://arxiv.org/abs/1703.00410

[64] Y. Gal, Z. Ghahramani. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. ICML, 2016. Available at: https://arxiv.org/abs/1506.02142

[65] T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, L. Daniel. “Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach”. ICLR, 2018. Available at: https://arxiv.org/abs/1801.10578

[66] I. Goodfellow. “Gradient Masking Causes CLEVER to Overestimate Adversarial Perturbation Size”. CoRR, 2018. Available at: https://arxiv.org/abs/1804.07870

[67] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, A. Criminisi. “Measuring Neural Net Robustness with Constraints”. NIPS, 2016. Available at: https://arxiv.org/abs/1605.07262

[68] G. Katz, C. Barrett, D. Dill, K. Julian, M. Kochenderfer. “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks”. CAV, 2017. Available at: https://arxiv.org/abs/1702.01135

[69] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, T. Walsh. “Verifying Properties of Binarized Deep Neural Networks”. IAAI, 2018. Available at: https://arxiv.org/abs/1709.06662

[70] Ehlers, Ruediger. “Formal verification of piece-wise linear feed-forward neural networks”. Au-
tomated Technology for Verification and Analysis, 2017. Available at: https://arxiv.org/abs/1705.01320

[71] S. Dutta, S. Jha, S. Sanakaranarayanan, A. Tiwari. “Output Range Analysis for Deep Neural Networks”. NFM, 2018. Available at: https://arxiv.org/abs/1711.00455

[72] V. Tjeng, K. Xiao, R. Tedrake. “Evaluating Robustness of Neural Networks with Mixed – Integer Programming”. ICLR, 2019. Available at: https://arxiv.org/abs/1711.07356

[73] R. Bunel, I. Turkaslan, P. H.S. Torr, P. Kohli, M. P. Kumar. “A Unified View of Piecewise Linear Neural Network Verification”. NIPS, 2018. Available at: https://arxiv.org/abs/1711.00455

[74] Ga. Singh, T. Gehr, M. Püschel, M. Vechev. “An Abstract Domain for Certifying Neural Networks”. POPL, 2019. Available at: https://www.sri.inf.ethz.ch/publications/singh2019domain

[75] M. Balunović, M. Baader, G. Singh, T. Gehr, M. Vechev. “Certifying Geometric Robustness of Neural Networks”. NIPS, 2019. Available at:
https://www.sri.inf.ethz.ch/publications/balunovic2019geometric

[76] S. Wang, K. Pei, J. Whitehouse, J. Yang, S. Jana. “Formal security analysis of neural networks using symbolic intervals”. USENIX Sec., 2018. Available at: https://arxiv.org/abs/1804.10829

[77] E. Wong, Z. Kolter. “Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope”. ICML, 2018. Available at: https://arxiv.org/abs/1711.00851

[78] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. Mann, P. Kohli. “On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models”. NIPS, 2018. Available at: https://arxiv.org/abs/1810.12715

[79] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, P. Kohli. “A Dual Approach to Scalable Verification of Deep Networks”. UAI, 2018. Available at: https://arxiv.org/abs/1803.06567

[80] M. Mirman, T. Gehr, M. Vechev. “Differentiable Abstract Interpretation for Provably Robust Neural Networks”. ICML, 2018. Available at: https://files.sri.inf.ethz.ch/website/papers/icml18-diffai.pdf

[81] J. M. Cohen, E. Rosenfeld, J. Z. Kolter. “Certified Adversarial Robustness via Randomized Smoothing”. ICML, 2019. Available at: https://arxiv.org/abs/1902.02918

[82] M. Lecuyer, V. Atlidakis, R. Geambasu, D. Hsu, S. Jana. “Certified Robustness to Adversarial Examples with Differential Privacy”. IEEE Sec. and Priv., 2019. Available at: https://arxiv.org/abs/1802.03471

[83] Z. Hu, X. Ma, Z. Liu, E. H. Hovy, E. P. Xing. “Harnessing deep neural networks with logic rules”. ACL, 2016. Available at: https://arxiv.org/abs/1603.06318

[84] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, M. Vechev. “DL2: Training and Querying Neural Networks with Logic”. ICML, 2019. Available at: https://www.sri.inf.ethz.ch/publications/fischer2019dl2

[85] P.-W. Wang, P. L. Donti, B. Wilder, Z. Kolter. “SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver”. ICML, 2019. Available at: https://arxiv.org/abs/1905.12149

[86] J. Ren, P. J. Liu, E. Fertig, J. Snoek, R. Poplin, M. A. DePristo, J. V. Dillon, B. Lakshminarayanan. “Likelihood Ratios for Out-of-Distribution Detection”. NIPS, 2019. Availalbe at: https://arxiv.org/abs/1906.02845

[87] S. Liang, Y. Li, R. Srikant. “Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks”. ICLR, 2018. Available at: https://arxiv.org/abs/1706.02690

[88] K. Lee, K. Lee, H. Lee, J. Shin. “A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks”. NIPS, 2018. Available at: https://arxiv.org/abs/1807.03888

[89] B. Lakshminarayanan, A. Pritzel, C. Blundell. “Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles”. NIPS, 2017. Available at: https://arxiv.org/abs/1612.01474

[90] D. Hendrycks, N. Mu, E. D. Cubuk, B. Zoph, J. Gilmer, B. Lakshminarayanan. “AugMix: A Simple Data Processing Method to Improve Robustness and Uncertainty”. ICLR, 2020. Available at: https://arxiv.org/abs/1912.02781

[91] C. Blundell, J. Cornebise, K. Kavukcuoglu, D. Wierstra. “Weight Uncertainty in Neural Networks”. ICML, 2015. Available at: https://arxiv.org/abs/1505https://arxiv.org/abs/1703.00810.05424

[92] A. Graves. “Practical variational inference for neural networks”. NIPS, 2011. Available at: https://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks

[93] R. M. Neal. “MCMC using Hamiltonian dynamics”. In Handbook of Markov Chain Monte Carlo, CRC Press, 2011. Available at: https://arxiv.org/pdf/1206.1901.pdf

[94] Y. Gal. “Uncertainty in Deep Learning”. PhD thesis, Cambridge University, 2016. Available at: http://mlg.eng.cam.ac.uk/yarin/thesis/thesis.pdf

[95] A. Kendall, Y. Gal. “What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?”. NIPS, 2017. Available at: https://arxiv.org/abs/1703.04977

[96] R. McAllister, Y. Gal, A. Kendall, M. van der Wilk, A. Shah, R. Cipolla, A. Weller. “Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning”. IJCAI, 2017. Available at: https://www.ijcai.org/Proceedings/2017/661

[97] D. Feng, L. Rosenbaum, K. Dietmayer. “Towards Safe Autonomous Driving: Capture Uncertainty in the Deep Neural Network For Lidar 3D Vehicle Detection”. ITCS, 2018. Available at: https://arxiv.org/abs/1804.05132

[98] K. Pei, Y. Cao, J. Yang, S. Jana. “DeepXplore: Automated White-box Testing of Deep Learning Systems”. SOSP, 2017. Available at: https://arxiv.org/abs/1705.06640

[99] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, Y. Wang. “DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems”. ASE, 2018. Available at: https://arxiv.org/abs/1803.07519

[100] Y. Sun, X. Huang, D. Kroening. “Testing deep neural networks”. CoRR, 2018. Available at: https://arxiv.org/abs/1803.04792

[101] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, D. Kroening. “Concolic Testing for Deep Neural Networks”. ASE, 2018. Available at: https://arxiv.org/abs/1805.00089

[102] J. Kim, R. Feldt, S. Yoo. “Guiding Deep Learning System Testing using Surprise Adequacy”. ICSE, 2019. Available at: https://arxiv.org/pdf/1808.08444.pdf

[103] A. Odena, C. Olsson, D. Andersen, I. Goodfellow:. “TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing”. ICML, 2019. Available at: http://proceedings.mlr.press/v97/odena19a.html

[104] M. Zhang, Y. Zhang, L. Zhang, C. Liu, S. Khurshid. “DeepRoad: GAN-based Metamorphic Autonomous Driving System Testing”. ASE, 2018. Available at: https://arxiv.org/abs/1802.02295

[105] J. Garcia, F. Fernandez. “A Comprehensive Survey on Safe Reinforcement Learning”. JMLR, 2015. Available at: http://jmlr.org/papers/volume16/garcia15a/garcia15a.pdf

[106] D. Erhan, Y. Bengio, A. Courville, P. Vincent. “Visualizing Higher-Layer Features of a Deep Network”. Technical report, Univ. of Montreal. Available at: https://www.researchgate.net/publication/265022827_Visualizing_Higher-Layer_Features_of_a_Deep_Network

[107] A. Mahendran, A. Vedaldi. “Understanding deep image representations by inverting them”. CVPR, 2015. Available at: https://arxiv.org/abs/1412.0035

[108] K. Simonyan, A. Vedaldi, A. Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”. ICLR, 2014. Available at: https://arxiv.org/abs/1312.6034

[109] M. D. Zeiler, R. Fergus. “Visualizing and Understanding Convolutional Networks”. ECCV, 2014. Available at: https://arxiv.org/abs/1311.2901

[110] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization”. ICCV, 2017. Available at: https://arxiv.org/abs/1610.02391

[111] A. Shrikumar, P. Greenside, A. Kundaje. “Learning Important Features Through Propagating Activation Differences”. ICML, 2017. Available at: https://arxiv.org/abs/1704.02685

[112] M. Sundararajan, A. Taly, Q. Yan. “Axiomatic Attribution for Deep Networks”. ICML, 2017. Available at: https://arxiv.org/abs/1703.01365

[113] https://distill.pub/2018/building-blocks

[114] J. R. Zilke, E. L. Mencía, F. Janssen. “DeepRED –Rule Extraction from Deep Neural Networks”. DS, 2016. Available at: https://www.ke.tu-darmstadt.de/publications/papers/DS16DeepRED.pdf

[115] L. Chu, X. Hu, J. Hu, L. Wang, J. Pei. “Exact and Consistent Interpretation for Piecewise Linear Neural Networks: A Closed Form Solution”. KDD, 2018. Available at: https://arxiv.org/abs/1802.06259

[116] M. T. Ribeiro, S. Singh, C. Guestrin. “Model-Agnostic Explanations By Identifying Prediction Invariance”. AAAI, 2018. Available at: https://arxiv.org/abs/1611.05817

[117] R. Shwartz-Ziv, N. Tishby. “Opening the Black Box of Deep Neural Networks via Information”. CoRR, 2017. Available at: https://arxiv.org/abs/1703.00810

Image sources

3.1: [27]
3.2: [4]
3.3: [4]
3.4: [4]
3.5: https://medium.com/element-ai-research-lab/tricking-a-machine-into-thinking-youre-milla-jovovich-b19bf322d55c
3.6: [34]
3.7: [34]
3.8: [44]
3.9: [46]
3.10: [11]
3.11: [77]
3.12: [79]
3.13: [85]
3.14: [17]
3.15: [90]
3.16: [96]
3.17: [98]