ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

MASTER THESIS

Electrical and Electronic Section

Major in Information Technologies

Measuring Robustness of Classifiers to Geometric Transformations

Student
Can Kanbak

Professor
Pascal Frossard

Supervisor
Seyed-Mohsen Moosavi-Dezfooli

EPFL LTS4 Laboratory

June 23, 2017

Abstract

For many classification tasks, the ideal classifier should be invariant to geometric transformations such as changing the view angle. However, this cannot be said decisively for the state-of-the-art image classifiers, such as convolutional neural networks. Mainly, this is because there is a lack of methods for measuring the transformation invariance in them, especially for transformations with higher dimensions. In this project, we are proposing two algorithms to do such measurement. The first one, Manifool, uses the structure of the image appearance manifold for finding small enough transformation examples and uses these to compute the invariance of the classifier. Second one, the iterative projection algorithm, uses adversarial perturbation methods in neural networks to find the fooling examples in the given transformation set. We compare these methods to similar algorithms in the areas of speed and validity, and use them to show that transformation invariance increases with the depth of the neural networks, even in reasonably deep networks. Overall, we believe that these two algorithms can be used for analysis of different architectures and can help to build more robust classifiers.

Acknowledgements

I would first like to thank Seyed-Mohsen Moosavi-Dezfooli for his enthusiastic supervision, for the weekly meetings in which we discussed the ideas and the results of the project. His help and guidance was invaluable for me and for this project. I also want to thank him for his review of this manuscript.

I would like to also thank Prof. Pascal Frossard for his assistance and for giving me the opportunity to work on this project and in his laboratory. Thanks to him and this project, I've learned a lot more than what I can write in this thesis.

Last and foremost, I would like to thank to my family, to whom this thesis is dedicated, for their unconditional and unending love, understanding, and support.

Contents

1 Introduction

In the past decade, convolutional neural networks have achieved state-of-the-art performance in computer vision tasks such as image classification [21] [42]. However, they have been shown to be vulnerable to certain changes in the input images. One such set of changes is geometric transformations, which are quite common as they can represent the change of the viewpoint of an image. Humans are shown to be mostly invariant to these transformations [10], but the same cannot be said about the classification systems. As these systems are now being used in various places such as self driving cars, the issue of correctly identifying the surrounding objects no matter their position are getting more and more important. Thus, one of the important areas of image classification research is finding classifiers that are robust against these changes in the image. However, to correctly identify methods that are actually robust against these transformations, a method for measuring this invariance is required.

To answer the problem of robustness in neural networks, many approaches has been taken during the last decades. For example, in 1991, Földiak [14] has tried to increase invariance by proposing a learning rule. More recently, Jaderberg et al. [22] has introduced spatial transformer modules that can be inserted into existing neural network architectures to increase their transform invariance. Others, like Dai et al. [9] tries to increase invariance by modifying the pooling layer of the network while Shen et al. [40] approach this by changing the convolutional layers. Other approaches in designing transformation invariant networks include scattering networks [40] and using transformed filter banks [23]. These methods are all trying to improve the invariance of networks, and thus their purpose is not measuring it. In fact, in the face of this many new method to increase invariance, a general method that can measure it becomes crucial to be able to compare their effects.

In another approach to this problem, Lenc and Vedaldi [29] and Soatto and Chiuso [43] analyze image and visual representations to find theoretical foundations of transform invariant features, while others such as Goodfellow et al. [17] do this analysis empirically. On the other hand, Bakry et al. [5] uses the information about the human visual system to understand transformation invariance. While these results show how the features that are used in convolutional neural networks should be, they either do not give ways to measure the invariance of a network, or give methods that can only measure the invariance in a single dimension(e.g. rotation). Similarly, some others approach the problem by creating sets of transformed images and measuring the accuracy of the classifier on this dataset [22], [25]. However, this is a laborious approach and does not let us effectively compare different kind of classifiers. Finally, Manitest [13] from Fawzi and Frossard measures the invariance of a classifier using the geodesic distances on the manifold of transformed images by using fast marching method. However, it is not feasible for larger dimensional transformations as its complexity increase exponentially with number of dimensions. This project can be considered a continuation of this method, as it starts from the same point as Manitest and improves the parts it is lacking, such as not being able to measure the invariance of complex classifiers, especially against transformations with higher dimensions.

Thus, to answer the problem of measuring invariance, we approach it similar to Manitest, i.e. by using the same metric to do this measurement. As this metric requires us to find small transformations that can 'fool' the classifiers, we then introduce two new methods to find such transformations. The said metric is used along with the outputs of these methods to calculate the invariance of a classifier. Our idea is to make a fast algorithm that can work on high dimensional geometric transformation sets while also finding fooling transformations that are close to being minimal. Currently, the proposed algorithms only work for some sets of transformations, but these sets include all common geometric transformations such as the projective transformations and its subsets.

The paper is organized as follows: In section 2 we give an overview of geometric transformations and structural properties of some transformation sets. We also introduce the metrics we use to measure the transformations. In section 3, we give information about convolutional neural networks and the concept of adversarial perturbations. Then, using the information on geometric transformations and neural networks, we formulate our problem in section 4. In section 5, we propose our first algorithm, Manifool, which can find fooling examples for certain sets of transformations called transform Lie groups and show some of the example outputs and experimental results using this algorithm. In section 6, we propose another approach, namely the iterative projection algorithm, which actively uses the adversarial perturbation methods to find fooling examples for certain sets that are not necessarily Lie groups. We conclude with section 7 and discuss possible future work.

2 Geometric Transformations

2.1 Introduction to geometric transformations of images

In the simplest sense, a geometric transformation is a function $\tau : \mathbb{R}^2 \to \mathbb{R}^2$. In the case of 2D images, it can be thought as a function which maps a pixel to another or in other words moves the positions of the pixels. More formally, if we consider a mathematical model where an image is represented as a square integrable function $I : \mathbb{R}^2 \to \mathbb{R}, I \in L^2$(or in the case of color images $I : \mathbb{R}^2 \to \mathbb{R}^c$ with $I_i \in L^2$ where c is the number of color channels) , an image transformed by τ can be represented as I_τ, or assuming the transformation is invertible, as $I_\tau(x, y) = I(\tau^{-1}(x, y))$. In this work, we are interested in invertible transformations and thus generally use the second representation. This is not a strict restriction as many common transformation examples such as rotations, affine transformations or diffeomorphisms are all invertible.

One of the most simple transformations is rotation of an image by an angle θ. For pixel coordinates (x, y), this transformation can be written as:

$$\tau_x(x, y) = x \cos\theta - y \sin\theta \tag{2.1}$$

$$\tau_y(x, y) = x \sin\theta + y \cos\theta \tag{2.2}$$

Another example of a projective transformation, which is represented as [20]:

$$\tau_x(x, y) = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + 1} \tag{2.3}$$

$$\tau_y(x, y) = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + 1} \tag{2.4}$$

Normally, this pair of coordinates, (τ_x, τ_y) are represented simply as a two dimensional vector. However, another way to represent them is to use a three dimensional vector $[a \quad b \quad c]$ and get the coordinates as $\tau_x = a/c, \tau_y = b/c$. This notation is called the homogeneous representation of a vector $\boldsymbol{x}$ and denoted as $\tilde{\boldsymbol{x}}$. By using this representation, the transformation can be represented as a matrix vector multiplication [20]:

$$\boldsymbol{H}\tilde{\boldsymbol{x}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \tag{2.5}$$

$$\tau_x = x'/w', \quad \tau_y = y'/w' \tag{2.6}$$

Thus, the whole transformation can be simply represented using the matrix $\boldsymbol{H}$ and the inverse transform can be represented using $\boldsymbol{H}^{-1}$. Some of the subsets of the projective transformations and their matrix representations are as follows:

- Rotations, which were previously mentioned, can be rewritten in matrix form as:

$$\boldsymbol{R}\tilde{\boldsymbol{x}} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2.7}$$

  They have only one parameter: the rotation angle.

- Translations, which shift the support of the image can be written as:

$$\boldsymbol{T}\tilde{\boldsymbol{x}} = \begin{bmatrix} 0 & 0 & t_x \\ 0 & 0 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2.8}$$

$$\tau_x = x + t_x, \quad \tau_y = y + t_y \tag{2.9}$$

  They have two parameters: amount of translation in x and y axes.

- Similarity transformations, which combine rotations, translations and scaling are written as:

$$\boldsymbol{S}\tilde{\boldsymbol{x}} = \begin{bmatrix} \alpha\cos\theta & -\alpha\sin\theta & t_x \\ \alpha\sin\theta & \alpha\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2.10}$$

$$\tau_x = \alpha x\cos\theta - \alpha y\sin\theta + t_x, \quad \tau_y = \alpha x\sin\theta + \alpha y\cos\theta + t_y \tag{2.11}$$

  They have four parameters: rotation angle, two translations and scaling.

- Affine transformations, which preserve the parallelisms of lines are written as:

$$\boldsymbol{A}\tilde{\boldsymbol{x}} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{2.12}$$

$$\tau_x = m_{11}x + m_{12}y + m_{13}, \quad \tau_y = m_{21}x + m_{22}y + m_{23} \tag{2.13}$$

  They have six parameters, which can be thought as translations, rotation, independent scaling for two dimensions, and shear.

2.1.1 Discrete images and interpolation

Although we model the images as continuous functions when analyzing the properties of transformations, in practice, they are in their sampled forms. A grayscale image can be though as a $w \times h$ matrix whereas a color image will be stacks of multiple $w \times h$ matrices. They can also be represented in their vectorized forms in $\mathbb{R}^{cwh}$. The sampled form of images means that the image is only defined on integer coordinates. Thus, the image transformations are not straightforward as $\tau(x, y)$(or conversely $\tau^{-1}(x, y)$) do not necessarily output integers. Because of this, the pixels of the transformed image, $I_\tau(x, y)$, has to be interpolated using the pixels of the original image.

Let source and destination domains denote the sets of coordinates where the original and transformed images are respectively defined. Then, the transformation for discrete images is done in two steps: First, the pixels in one domain are mapped to the other one. Second, the pixel values of the transformed image are computed using the nearby known values. When the source pixels (x, y) are mapped to the destination domain as $(u, v) = \tau(x, y)$ and interpolation is done there, this process is called forward mapping. This operation maps the regular grid of the source to an irregular one, and thus the interpolation has to be done using kernel methods such as Shepherd's method [41]. The other option, called backward mapping, maps destination pixels to the source domain and interpolates them using the regular grid of known values. The regularity of the grid means the neighboring pixels can be found easily and thus the interpolation is simpler to implement. The computation can be done using any interpolation scheme for 2D grids, such as nearest neighbor, bilinear or bicubic interpolation. The choice of interpolation depends on the preferred complexity and generated error, as less complex methods such as nearest neighbor will have higher error. In this project, bilinear interpolation is used for computing translations as it is both simple to implement and works well enough, especially on high dimensional images.

2.2 Geometric Transformations as Lie Groups

Although we have defined transformations simply as functions, various sets of transformations exhibit certain algebraic structures called Lie groups. Basically, these are groups which also have a smooth manifold structure. In this section we will explain the properties of these transform Lie groups and some of their connections with the transformed images. These connections will form the basis of how we can measure and compare transformations as well as the first algorithm, Manifool, that will be explained in Section 5.

2.2.1 Differentiable manifolds and Lie groups

Formally, a d-dimensional manifold, denoted by $\mathcal{M}$ is a topological space whose every point is homeomorphic to an open subset of $\mathbb{R}^d$ [6]. In other words, for every point $p \in \mathcal{M}$, we can find a continuous bijection $\varphi : \mathcal{U} \to S \subset \mathbb{R}^d$ with a continuous inverse where $\mathcal{U}$ is an open neighborhood of p. The map φ is called a chart and a set of charts whose domain covers the entire manifold is called an atlas. An example for an atlas of a circle in a 2-dimensional plane is seen on Figure 1. Similar to this case where the circle is a 1D subset of $\mathbb{R}^2$, a manifold can be a subset of another space, generally of a higher dimension. For the rest of this section, we will focus on embedded manifolds, since all of the manifolds in this project are embedded in some other vector space and restricting ourselves on this set will ease some of the definitions we will introduce.

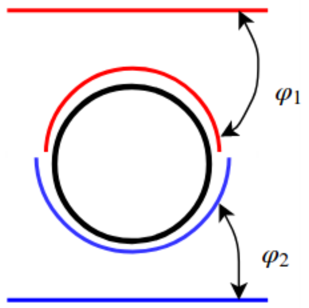


Figure 1: A circle is a 1-dimensional manifold embedded in $\mathbb{R}^2$. Here, it is represented using 2 charts which are homeomorphisms from the top and bottom half circles to the subsets of $\mathbb{R}$ (line segments). The set of these charts form an atlas as it covers the entirety of the circle.

A manifold is called differentiable if it admits an atlas where every chart in this atlas and their transition maps(maps between the ranges of two charts whose domains overlap) are differentiable [6]. A point p on a differentiable manifold $\mathcal{M}$ has a tangential space $T_p\mathcal{M}$, which, for a manifold embedded in a vector space, can be defined as the set of differentials of all paths on the manifold passing from this point [1]. The basis of $T_p\mathcal{M}$ can be acquired using the partial derivatives of the inverse of the local chart as

$$\partial_i \varphi^{-1}(0_p) = \lim_{t \to 0} \frac{\varphi^{-1}(0_p + te_i) - \varphi^{-1}(0_p)}{t} \tag{2.14}$$

where $0_p = \varphi(p)$ and e_i denotes the i^{th} canonical vector of $\mathbb{R}^d$. These partial derivatives are guaranteed to be linearly independent, as otherwise the chart will not be a bijection. As we have d linearly independent vectors, we can see that the tangential space also has d dimensions. The matrix whose columns are these basis vectors is called the Jacobian matrix and denoted as $\boldsymbol{J}_p$. Since the tangent space is the column space of $\boldsymbol{J}_p$, a d-dimensional vector $\boldsymbol{u} \in T_p\mathcal{M}$ can be mapped to the ambient space $X \supset \mathcal{M}$ as $\boldsymbol{J}_p\boldsymbol{u}$.

Lie groups are one important type of differentiable manifolds, which in addition to their manifold structure, also have properties of a mathematical group, i.e. has an associative and invertible group operation under which the manifold is closed. The number of dimensions for this manifold is equal to the number of parameters of the transform. The tangential space of the identity element is called the Lie algebra of the group and a bracket operation can be defined on it, giving it a algebraic structure [19]. The basis of this tangential space is called the generators, which algebraically, can be used to construct the set. Several sets of transformations can be modeled as Lie groups, with the function composition as the group operation. This means that if a set of transformations $\mathcal{T}$ is Lie group, then:

- If $\tau \in \mathcal{T}$, then τ is smooth.
- For all $\tau_i, \tau_j \in \mathcal{T}$, $\tau_i \circ \tau_j \in \mathcal{T}$
- For all $\tau_i, \tau_j, \tau_k \in \mathcal{T}$, $(\tau_i \circ \tau_j) \circ \tau_k = \tau_i \circ (\tau_j \circ \tau_k)$
- For all $\tau_i \in \mathcal{T}$, $\tau_i^{-1} \in \mathcal{T}$
- $\tau(x, y) = x, y \in \mathcal{T}$. This is the identity element of $\mathcal{T}$.

Although these are strong restrictions to add on a set of functions, they are applicable to many types of transformations. All of the previously mentioned transformations such as rotations, affine or projective transformations, as well as many other types such as piecewise affine transformations are Lie groups. In fact, the matrix representations of projective transformations also form Lie groups, where the group action is matrix multiplication and inverse action is matrix inversion. These groups are homeomorphic to their transform group counterparts, and this means certain operations on the transform groups can be done in the matrix versions using the better defined linear algebraic functions.

One last thing to note is that not all transformation sets are Lie groups. For example, the set of integer translations is not a Lie group since it is not continuous and hence not smooth, even though it is still a group under composition. Similarly, the set of rotations between $-\pi/2$ and $\pi/2$ is smooth, but not a group since it is not closed under composition.

2.2.2 Lie group action on images

The set of transformations and transformed images are connected to each other by a function called group action. In group theory, this action of a group G on a manifold $\mathcal{M}$ is defined as a map

$G \times \mathcal{M} \to \mathcal{M}$. For the transform Lie groups and the image space, this action is defined as the transformation of the image, i.e. if $\tau \in \mathcal{T}$, the map is defined as $(\tau, I) \to I_\tau(x, y)$. If the transform is smooth; this map is also smooth, which is the case for all transformations represented as Lie groups. If we consider a single element of the image space, I, the orbit of I can be defined as the set of all images generated by the group action:

$$\mathcal{T} \cdot I = \{I_\tau : \tau \in \mathcal{T}\} \tag{2.15}$$

This set of transformed images also form a differentiable manifold, denoted as $\mathcal{M}(I)$ and called an image appearance manifold(IAM) following the works of [46] [24]. In the transform group, the stabilizers of I are defined as the set of transformations that do not change I:

$$\mathcal{T}_I = \{\tau \in \mathcal{T} : I_\tau = I\} \tag{2.16}$$

Lastly, using the definition of the orbit, we can define an orbit map as a function $\psi^{(I)} : \mathcal{T} \to \mathcal{T} \cdot I$ by

$$\psi^{(I)}(\tau) = I_\tau \tag{2.17}$$

If $\mathcal{T}_I = \{e\}$, i.e. if every transform but the identity changes the image in some way, then the orbit map is a smooth bijection and the orbit is an immersed submanifold of L^2 [28]. This means the orbit of the image and the Lie group is diffeomorphic, and thus, the dimensionality of this manifold is the same as the Lie group that generates it. So, the parametrized transformation can be used as chart for $\mathcal{M}(I)$ and the basis of the tangential spaces can be defined as

$$\frac{\partial I_{\tau(\theta)}(x, y)}{\partial \theta_i} = \frac{\partial \psi^{(I)}(\tau(\theta))}{\partial \theta_i} \tag{2.18}$$

where $\theta \in R^d$ represents the d parameters of the transform. Note that the existence of this diffeomorphism requires that the stabilizer of an image only includes the identity element, which might not be true in the general case. For example, for an image of a circle under rotation, the stabilizer is the whole Lie group and not only the identity element. However, for natural images that we are interested in, we assume that only the identity element is a stabilizer and IAM has the same dimensions as the transformations.

2.3 Metrics on Transformations, Exponential Maps and Retractions

To be able to measure the effect of transformations on classification results, we need a method to compute the impact of the transformation on an image, i.e. a metric $d(\tau_1, \tau_2) : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$. One idea might be to measure the distance between the parameter vectors, but since they have very different natures, the comparison would not be informative. As an example, in Figure 2, we can see that comparing a rotation of $\pi/2$ with a translation of 50 pixels in y axis is meaningless because of the different natures of the parameters. In addition, the metric should depend on the image as well as the transformations, since we are measuring how this transformation is affecting the image and the effect might change from one image to the next. One example of such a metric could be the squared L^2 distance between the transformed images which is defined as:

$$d_I(\tau_1, \tau_2) = \|I_{\tau_1} - I_{\tau_2}\|_{L^2}^2 = \iint_A |I_{\tau_1}(x, y) - I_{\tau_2}(x, y)|^2 dxdy \tag{2.19}$$

where A is the support of the image. For the discrete images, this metric simply becomes the squared Euclidean metric. However, for example, on the images seen on Figure 2, the L^2 distances

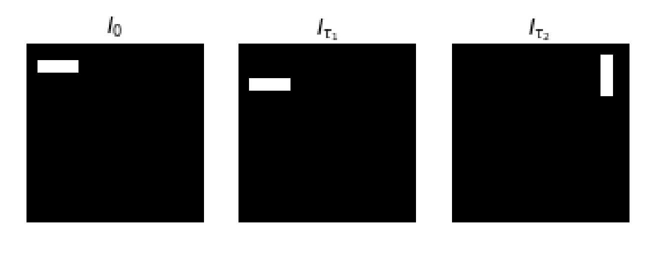


Figure 2: Representation for problem of using L^2 or transform parameters as metrics. τ_1 is translation in y axis by 50 pixels, while τ_2 is rotation around center by $\pi/2$. Black pixels have a value of zero, an thus both τ_1 and τ_2 have the same distance from τ_0 if L_2 metric is used. Adapted from [13]

are the same for two transformations while the rotation clearly had a greater effect on the image compared to the small translation. Thus, this metric also is not applicable for this problem.

A useful metric can be acquired from the IAM by using the approach of [13]. Let $\gamma(t) : [0, 1] \to \mathcal{T}$ be a differentiable curve on the Lie group $\mathcal{T}$ and $\gamma'(t) \in T_{\gamma(t)}\mathcal{T}$ be its derivative. Using the orbital map, the curve and the derivative can be mapped to IAM as $I_{\gamma(t)}$ and $dI_{\gamma(t)}/dt$. Using this map from the tangential space of $\gamma(t)$ to the tangential space of $I_{\gamma(t)}$, for $v_1, v_2 \in T_\tau\mathcal{T}$ we can define an inner product as

$$\langle v_1, v_2\rangle = \langle \frac{dI_{\gamma_1(t)}}{dt}, \frac{dI_{\gamma_2(t)}}{dt}\rangle \tag{2.20}$$

where $\gamma_1(t)$ and $\gamma_2(t)$ are two paths on $\mathcal{T}$ whose derivatives on τ are v_1 and v_2. The norm on the tangential space is similarly defined as $\|v_1\|_{\mathcal{T}} = \langle v_1, v_1\rangle$. By using this, we can define the length of the curve $\gamma(t)$, $L(\gamma)$, as

$$L(\gamma) = \int_0^1 \|\gamma'(t)\|_{\mathcal{T}} dt = \int_0^1 \left\|\frac{dI_{\gamma(t)}}{dt}\right\|_{L^2} dt \tag{2.21}$$

The inner product on equation (2.20) is called a Riemannian metric and any manifold that is equipped with a Riemannian metric is called a Riemannian manifold. As seen above, the Riemannian metric allows one to compute length of curves on the manifold. Let $\boldsymbol{v}_1, \boldsymbol{v}_2 \in T_\tau\mathcal{T}$ which are represented as d-dimensional vectors using a basis (such as the generators). Then, Riemannian metric can be written as

$$g_{\mathcal{T}}(\boldsymbol{v_1}, \boldsymbol{v_2}) = \boldsymbol{v}_1^T \boldsymbol{J}_\tau^T \boldsymbol{J}_\tau \boldsymbol{v}_2 = \boldsymbol{v}_1^T \boldsymbol{G}_\tau \boldsymbol{v}_2 \tag{2.22}$$

where $\boldsymbol{J}_\tau$ is the Jacobian matrix of $\mathcal{T}$ at τ. Using the matrix $\boldsymbol{G}_\tau$, the norm on the tangential space of τ is denoted as $\|\boldsymbol{v}\|_{\boldsymbol{G}_\tau}$.

Using this metric, one can also define a minimal geodesic between two given points on $\mathcal{M}$ as the curve $\gamma(t)$ that minimizes $L(\gamma)$. Geodesics are curves that extend the notion of lines on Euclidean spaces to the manifolds. On $\mathcal{T}$, the geodesic distance calculated using (2.20) fits our previous requests as it is both determined by the image and also captures the distance the transformation has to take to arrive at the final point. Thus, this is the metric we use to measure the distance between two transformations and hence will be referred as $d_I(x, y)$ for $x, y \in \mathcal{T}$. In general, to measure the effect of a transform, we will use the distance between itself and the identity element e. Transformations on different images can then be compared by normalizing this distance by the L^2 norm of the image

$$\tilde{d}_I(e, \tau) = \frac{d_I(e, \tau)}{\|I\|_{L^2}} \tag{2.23}$$

This will be our general metric to compare transformations if they can be represented as Lie groups. However, we also note that this is not always the case. If the set of transformations is not a Lie group, then the transformed images will not form a manifold and thus a geodesic distance cannot be measured[1]. In these cases, L^2 metric can be used as there is no simple alternative.

2.3.1 Exponential map

One aspect of Riemannian manifolds is the exponential map $\exp(\cdot) : T_p\mathcal{M} \to \mathcal{M}$ which connects the tangential spaces of a manifold to itself. For any point $p \in \mathcal{M}$ and $\boldsymbol{v} \in T_p\mathcal{M}$ where $\mathcal{M}$ is a Riemannian manifold, there exists a unique geodesic γ_v with initial point $\gamma_v(0) = p$ and derivative $\gamma_v'(0) = \boldsymbol{v}$ [45]. Using this geodesic, if $\gamma_v(1)$ is defined, the exponential map is simply defined as

$$\exp_p(\boldsymbol{v}) = \gamma_v(1) \tag{2.24}$$

Since $\gamma_v'(0) = \boldsymbol{v}$ and γ is a geodesic, the distance traveled in unit movement is equal to $\|\boldsymbol{v}\|_{G_p}$. Then, the geodesic distance between p and the mapped point can be stated as

$$d(p, \exp_p(\boldsymbol{v})) = \|\boldsymbol{v}\|_{G_p} \tag{2.25}$$

This means that by using exponential maps, we can guarantee that the movement on the tangential plane is equal to the movement on the manifold. Note that the existence of exponential map requires $\gamma_v(1)$ to be defined and thus it does not always exist. Its existence is guaranteed for an open ball $B(0, \varepsilon) \subset T_p\mathcal{M}$ where $\varepsilon > 0$ is a real number [45] but outside this ball, a different mapping must be used.

A Lie group G also holds a similar but different notion of an exponential map. In this case, the exponential map is defined for the entire Lie algebra $\mathfrak{g}$, and for a point $\boldsymbol{X} \in \mathfrak{g}$, it is defined as

$$\exp(\boldsymbol{X}) = c_{\boldsymbol{X}}(1) \tag{2.26}$$

where $c_{\boldsymbol{X}}$ is a one-parameter subgroup generated by $\boldsymbol{X}$ [28]. If G is a matrix Lie group, then this map is equivalent to the matrix exponential

$$\exp(\boldsymbol{X}) = \sum_{k=0}^{\infty} \frac{\boldsymbol{X}^k}{k!} = I + \boldsymbol{X} + \frac{1}{2}\boldsymbol{X}^2 + \frac{1}{6}\boldsymbol{X}^3 + \dots \tag{2.27}$$

which can be approximated using numerical algorithms such as [2].

Remark The exponential map in the Lie group sense is not same with the exponential map in the Riemannian sense. This is only the case if the Riemannian metric on the Lie group manifold is bi-invariant [45]. Unfortunately, it does not hold for matrix Lie groups we are interested in. For example, for $SE(2)$ group which combines translations and rotations, the non-existence of such a Riemannian metric can be proved [4].

[1]One exception, if possible, would be to measure the geodesic distance on a Lie group superset of the requested set of transformations, but depending on the problem, this might not be a good metric

2.3.2 Retractions

Riemannian exponential map is only a member of a larger set of smooth mappings from the tangential space to the manifold called retractions. A retraction, denoted for $p \in \mathcal{M}$ as $R_p : T_p\mathcal{M} \rightarrow \mathcal{M}$, has following properties [1]:

- $R_p(0_p) = p$ where 0_p denotes the zero element of $T_p\mathcal{M}$.
- With the identification that $T_{0_p}T_p\mathcal{M} \simeq T_p\mathcal{M}$, R_p satisfies

  $$DR_p(0_p) = id_{T_p\mathcal{M}} \tag{2.28}$$

  where $id_{T_p\mathcal{M}}$ is the identity mapping on $T_p\mathcal{M}$ and $DR_p(0_p)$ is the differential of the mapping. In other words, the derivative of a curve $\gamma(t)$ on $T_p\mathcal{M}$ at 0_p must be equal to the derivative of the mapped curve $R_p(\gamma(t))$ at point p.

Although the definition is very general, the retractions form the basis of line search (gradient descent like) methods on manifolds [1] and will be one of the key parts of Manifool.

2.3.3 Measuring geodesic distance

Although the geodesic distance is a good metric to compare different transformations, its computation is not straightforward. In fact, most methods find an approximate distance rather than computing it exactly. Two methods to measure this distance is examined.

Fast Marching Method The first method to examine uses the notion of a geodesic distance map to calculate the distance. For a fixed point $p \in \mathcal{T}$, the distance map is defined as a function $U_p(\tau) : \mathcal{T} \rightarrow \mathbb{R}$ which outputs the geodesic distance of τ to the point p, i.e. $U_p(\tau) = d(p, \tau)$. If $\tau \rightarrow G_\tau$ is a continuous, the geodesic distance map $U_p(\tau)$ is the unique viscosity solution to the Eikonal equation [34]

$$\forall \tau \in \mathcal{T}, \quad \|\nabla U_p(\tau)\|_{G_\tau^{-1}} = 1 \tag{2.29}$$

$$U_p(p) = 0 \tag{2.30}$$

Fast Marching Method(FMM) [38], numerically solves this equation by discretizing the manifold. It simulates a wavefront propagation to find this solution and as a dynamic programming successive approximation method it can be considered a continuous version of the Dijkstra's algorithm [11]. The algorithm initializes with $\hat{U}_p(p) = 0$ and $\hat{U}_p(x) = \infty$ for $x \neq p$ where p is the point to whom the distances are computed and $\hat{U}_p$ is the estimation of the distance map U_p. Then, it starts to iterate with q being the point considered in each iteration, it does three things:

1. Chooses q as $\arg\min_{x \notin K} \hat{U}_p(x)$, i.e. the point with the minimum estimated distance at this point.
2. Denotes q as a known point and adds it to set of known points K.
3. Updates $\hat{U}_p(x)$ for $x \in \mathcal{N}_q/K$ using the points in K, where $\mathcal{N}_q$ denotes the neighbors of q

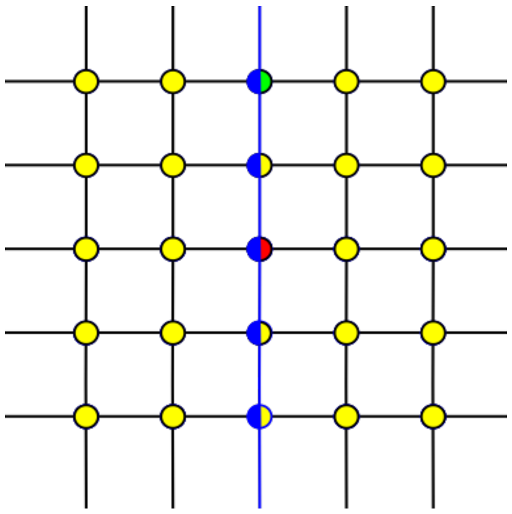


Figure 3: Schematic representation of a discretized manifold $\mathcal{T}$ with red circle showing the starting point of the algorithm and green circle showing the stopping point whose distance is being calculated. Assume the blue points are part of a 1D manifold which is embedded in a 2D manifold, shown here by yellow points. Also assume the distances between two neighboring nodes are the same for the 2D manifold. FMM must do at least 3 iterations for the 1D manifold and 9 iterations for 2D.

If the geodesic distance between $(p, \tau) \in \mathcal{T}$ is being calculated, the algorithm will stop when τ is chosen and will output $\hat{U}_p(\tau)$ as the distance. Further information on the algorithm can be found in [39] [34].

One drawback of using this algorithm is that it does not scale well with the dimensionality of the manifold. An illustration for an example of this is shown on Figure 3 where a 1D manifold, that is using a blue/yellow nodes, is embedded in a 2D manifold that is shown using yellow nodes. Assume the distances between two nodes are the same for all neighboring nodes. Let us try to measure the distance between the green and red nodes by starting FMM from the red node. If we run the algorithm only on the 1D manifold, we need to iterate 3 or 4 times before we reach the green point, depending on which of the two points with same distance the algorithm choses first. However, on the 2D manifold, it would take 9-12 iterations to compute the same distance, since we need to compute the distances for all nodes in the center square because they all have smaller distances than the green node. If the 2D manifold is embedded in a 3D one, then the number of iterations increase to be at least 27 as we need to iterate over the central cube before reaching the final node. Furthermore, in each iteration,the surrounding simplices of neighboring nodes that also include the current node must be found to update their distances. In 1D, there is only one simplex that include the neighbor and the current node, while in 2D there are two of them and 3D there are four. Hence, the number of simplices surrounding one node also increase exponentially with dimension. Thus, as this example illustrates, the complexity of the algorithm (per iteration and in total) increases exponentially with the dimension of the manifold and thus while still being accurate, it is not very efficient in such circumstances.

Direct Distance Another, faster method is used which we call direct distance method whose idea lies in exponential maps. As stated in (2.25), the norm of the vector on the tangential space is

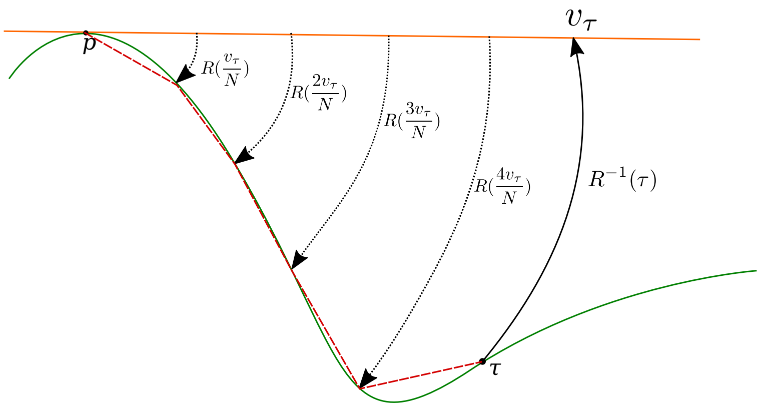


Figure 4: Illustration of the direct distance method on a 1D manifold(green curve). After v_τ is computed, the geodesic distance is estimated by mapping v_τ to the manifold part by part and summing the distances between these points. The length of the red curve is $d(p, \tau)$ from (2.33).

equal to the geodesic distance of the mapped point. So, given $p, \tau \in \mathcal{T}$, if one can compute

$$v_\tau = \exp_p^{-1}(\tau) = \log_p(\tau), \tag{2.31}$$

then the geodesic distance can be calculated as $\|v_\tau\|_{G_p}$. If the exponential map and its inverse are not readily available, a retraction can be used as an approximation of the exponential map. In this case, for the same p, τ, v_τ can be estimated as

$$\hat{v}_\tau = R_p^{-1}(\tau). \tag{2.32}$$

However, unlike the case with v_τ, $\|\hat{v}_\tau\|_{G_p}$ is not equal to the geodesic distance. Nevertheless, an approximation of the geodesic distance can be done by estimating the length of the curve $\gamma(t) = R(t\hat{v}_\tau)$. For a chosen step-size λ, the vector can be divided into $N = \left\lceil \frac{\|v_\tau\|_{G_p}}{\lambda} \right\rceil$ parts with each part having norm $\hat{\lambda} = \frac{\|v_\tau\|_{G_e}}{N}$. Then, the estimation of the distance is

$$d(p, \tau) = \sum_{t=1}^{N} \|I_{\exp(t\lambda)} - I_{\exp((t-1)\lambda)}\|_{L^2} \tag{2.33}$$

which is the sum of L^2 distances between the mapped points. An illustration of this is seen on Figure 4.

3 Convolutional Neural Networks and Their Vulnerabilities

3.1 Short Introduction to Convolutional Neural Networks

Neural networks(NNs) are powerful machine learning models which currently achieve state-of-the-art performance in various areas such as bioinformatics [7] [30], machine translation [47], recommender systems [8] and computer vision [21] [42]. Informally, they are networks composed of many nodes called neurons which are connected to each other in a layer by layer fashion. In a feed-forward network, the computation starts at the input layer and propagates forward until it reaches to the last layer, the output. These output neurons can represent different classes, probability distributions or extracted features of the input. This whole network can be represented as a single function $f : \mathbb{R}^n \to \mathbb{R}^m$, where n is the number of input nodes and m is the number of output nodes. If the network is used as a classifier, the chosen label can be found as the output with the largest value as

$$k(\boldsymbol{x}) = \arg \max_{i \in [1,m]} f(\boldsymbol{x}) \tag{3.1}$$

which will be our notation for rest of the thesis.

One important type of feed-forward networks is convolutional neural networks (CNNs) which include all of the current state-of-the-art image classification networks such as ResNet [21] or VGG [42]. In general, they are composed of three types of layers (see Figure 5): Convolutional, Pooling and ReLU. Convolutional layers, which gives the network its name, are constructed by a set of learnable filters whose outputs are stacked in a third dimension called depth. The filtering action is similar to performing a convolution on the previous layer. Pooling layers perform a downsampling operation along the spatial dimensions of the previous layer to reduce the number of parameters. This layer is usually implement using max pooling, which keeps the neuron with the maximum output and discards the rest, but other pooling functions such as average pooling exist [37]. The third type of layer, ReLU, applies an elementwise activation function. It gets its name from the current popular activation function, the Rectified Linear Unit ($\phi(x) = max(0, x)$), but other activation functions also exist. Overall a number of such layers are stacked together to construct the network. Further information on CNNs and neural networks in general can be found in [16].

Lastly, as with any other machine learning model, a NN must be trained before it is able to

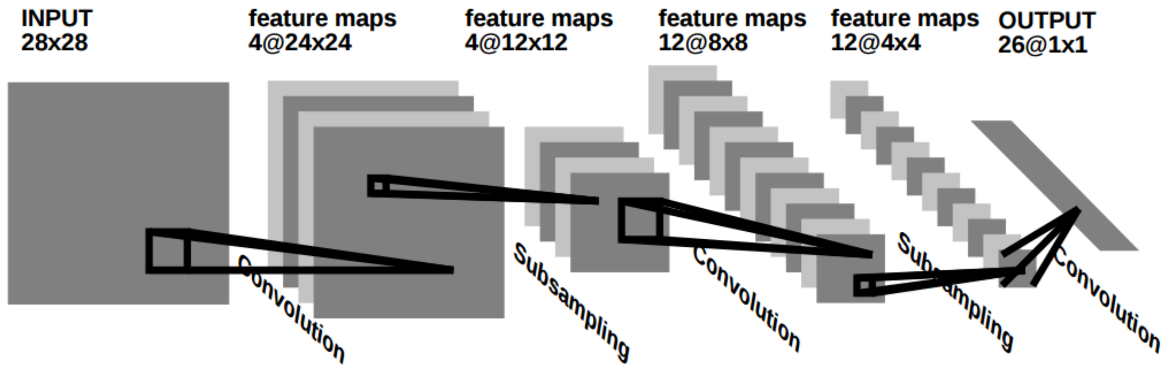


Figure 5: A convolutional neural network. The convolution and pooling layers are shown explicitly, while ReLU layers should follow the pooling layers. Image taken from [27].

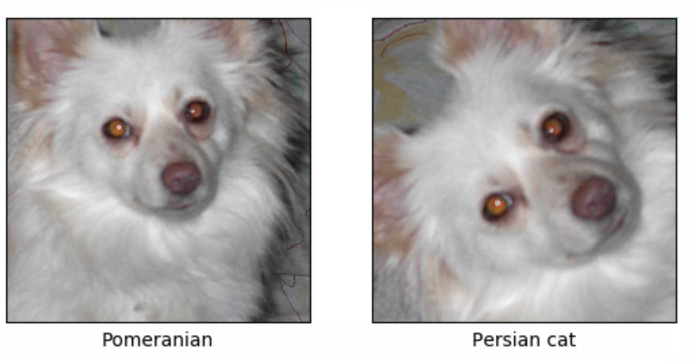


Figure 6: An image and its transformed version. A CNN (Resnet-50 in this case) can label the image on the left as a dog while labeling the image on the right as a cat, while the object in the image does not actually change.

do the required classification tasks. The training of a NN can be done using any of the major learning paradigms (supervised, unsupervised and reinforcement) [21] [12] [31]. In the pattern recognition applications which we are interested in, the training is mainly done in a supervised fashion. This method uses pre-labeled input examples to get correct values of network parameters (weights and biases) by minimizing a cost function which outputs high values for misclassifications. Mean squared error and cross entropy are two widely used examples of such cost functions. The minimization is generally done using stochastic gradient descent (SGD) where the gradients for SGD can be computed efficiently thanks to the backpropagation algorithm [35] which uses chain rule to compute the gradients starting from the output layer and moving backwards to the input layer. When the network is considered as a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, this algorithm can also be used for computing $\nabla_x f_i(x)$, the gradient of the i^{th} output of f with respect to the input.

3.2 Effect of Transformations in NNs

As discussed in Section 2.1, a geometric transformation will change the positions of the pixels on an image and thus the image itself. Therefore, the output values and perhaps the output label of any CNN which was fed by this image can also change when it is transformed. These transformations however, will not generally change the objects on an image. For example, as seen on Figure 6, an image transformed by a similarity transformation has still the same object and thus the output class should also not change. Human visual system possesses this invariance to geometric transformations to some extent and will perceive the same object for the transformed versions of an image [10], but the same claim cannot be made easily about CNNs or other classification systems. Thus, there have been numerous studies about invariance to geometric transformations on CNNs. Some, such as [40] [22] change parts of the network to improve the geometric transformations, while others, such as [29] and [43] do empirical or theoretical analysis to understand and promote invariance in classifiers. Some of these studies, like [17], do this analysis by proposing methods to measure this invariance property.

One of those methods is Manitest [13]. Given an input image $\boldsymbol{x}$, a set of transformations $\mathcal{T}$, and a classifier k, the algorithm tries to measure the invariance of k to transformation of $\boldsymbol{x}$ as the minimum geodesic distance between a transformation $\tau \in \mathcal{T}$ and the identity transformation. It does this by

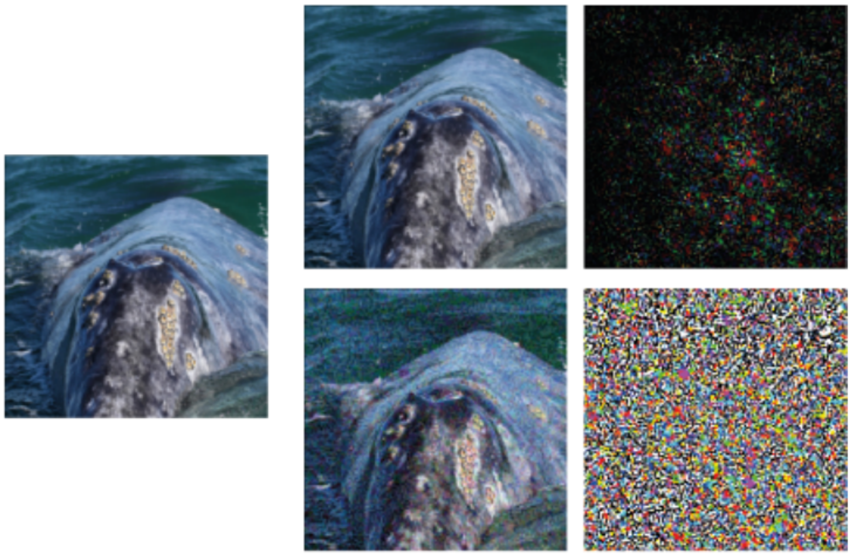


Figure 7: Examples of adversarial perturbations. First row, the original image classified as 'whale'. Second row, the perturbed versions of the image (using Deepfool in top, FGS in bottom) both classified as 'turtle'. Third row, the perturbations generated by Deepfool and FGS. The image taken from [32]

using fast marching method(FMM - explained in Section 2.3.3) with simple changes. Starting from the identity transformation, FMM is run on the IAM and $k(\boldsymbol{x}_q)$ is computed after choosing the minimum distance point $\boldsymbol{x}_q$ (i.e. after step 1 of FMM). However, unlike the case in Section 2.3.3, there is no predetermined stopping node and the iterations are stopped if $k(\boldsymbol{x}_q) \neq k(\boldsymbol{x})$, where $k(\boldsymbol{x})$ is the label of the input image. As FMM chooses the point with minimum distance in each iteration, the stopping point is guaranteed to be the transformation that minimizes the geodesic distance. Thus, it outputs the distance of the stopping point, $\hat{U}_e(\boldsymbol{x}_{q^*})$ as the invariance score of k for image $\boldsymbol{x}$.

3.3 Adversarial Perturbations

Another type of vulnerability of neural networks is adversarial perturbations, which were first discovered by Szegedy et al. [44]. These are small additive perturbations on correctly classified images which are imperceptible to the human eye but still change the result of the classification. They are called adversarial, because these perturbations are created specifically by methods that actively seek the misclassification of a given input by the neural network. Two examples of such methods are as follows:

Fast Gradient Sign(FGS) The first method, fast gradient sign [18], is a fast and simple method for finding an adversarial example. They take a linear view of adversarial examples, which state the linear behavior of neural networks as the cause of these perturbations and show that given the network parameters θ, the input $\boldsymbol{x}$, the original label y and the cost function that was used for training the network $C(\theta, x, y)$, an adversarial perturbation can be found as

$$r = \epsilon \,\text{sign}(\nabla_x C\,(\theta, \boldsymbol{x}, y)) \tag{3.2}$$

where ϵ is a variable that determines the magnitude of the perturbation. Computation of the gradient only requires a single backpropagation and thus the perturbation can be computed very quickly. It is however not guaranteed to change the label of the image and as expected, the misclassification percentage of the perturbed images is heavily related to ϵ and increases with it. Overall, the method is useful when the speed of the computation is more important than the magnitude of the perturbation. An example of this perturbation can be seen on Figure 7

Deepfool Deepfool [32] is an iterative algorithm (seen on Algorithm 1), which, unlike FGS, guarantees to find a small and misclassifying perturbation of the input image. At each iteration tarting from the input image, Deepfool acts as if the classifier is linear and moves towards the decision boundary. For a classifier $f : \mathbb{R}^n \to \mathbb{R}^m$, its linearized version can be written as

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b} \tag{3.3}$$

where $\boldsymbol{W}$ is $n \times m$ matrix whose i^{th} column is the gradients of the output value that corresponds to label i, i.e. $\boldsymbol{w}_i = \nabla f_i(\boldsymbol{x})$. The original label of the image is $l_{\boldsymbol{x}} = k(\boldsymbol{x})$ where k is the classification function defined in (3.1). In this linearized case, the label with the nearest boundary can be found as

$$l_n = \arg\min_{i \neq l_{\boldsymbol{x}}} \frac{|f_i(\boldsymbol{x}) - f_{l_{\boldsymbol{x}}}(\boldsymbol{x})|}{\|\boldsymbol{w}_i - \boldsymbol{w}_{l_{\boldsymbol{x}}}\|_2} \tag{3.4}$$

The objective function of the minimization can be thought as the duration it would take to move to the boundary, $f_k(\boldsymbol{x}) = f_{l_{\boldsymbol{x}}}(\boldsymbol{x})$, with a speed of $\|\boldsymbol{w}_i - \boldsymbol{w}_{l_{\boldsymbol{x}}}\|_2$, which is the norm of the gradient vector for $f_k(\boldsymbol{x}) - f_{l_{\boldsymbol{x}}}(\boldsymbol{x})$. Using l_n, the minimum perturbation to change the label of the linearized classifier can be found as

$$\boldsymbol{r}^*(\boldsymbol{x}) = \frac{|f_{l_n}(\boldsymbol{x}) - f_{l_{\boldsymbol{x}}}(\boldsymbol{x})|}{\|\boldsymbol{w}_{l_n} - \boldsymbol{w}_{l_{\boldsymbol{x}}}\|_2^2}(\boldsymbol{w}_{l_n} - \boldsymbol{w}_{l_{\boldsymbol{x}}}) \tag{3.5}$$

Since it is found only for a linearized version of f, this perturbation does not necessarily cause misclassification. Therefore, starting from the original image, the algorithm iterates where in each iteration it linearizes the classifier around the perturbed image, $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \boldsymbol{r}^*(\boldsymbol{x}_{i-1})$, and computes a new perturbation for this image, $\boldsymbol{r}^*(\boldsymbol{x}_i)$. If $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{r}^*(\boldsymbol{x}_i)$ is misclassified, than the algorithm returns the total perturbation, $\sum_i \boldsymbol{r}^*(\boldsymbol{x}_i)$. If it is not misclassified, then it continues iterating. It

Algorithm 1 Deepfool [32]

```
1: Initialize with $\boldsymbol{x}_0 \leftarrow x$, $i \leftarrow 0$.
2: while $k(\boldsymbol{x}_i) = k(x_0)$ do:
3:     for $j \neq \hat{k}(x_0)$ do
4:         $\boldsymbol{w}_j \leftarrow \nabla f_i(\boldsymbol{x}_i) - \nabla f_{l_{\boldsymbol{x}}}(\boldsymbol{x}_i)$
5:         $f'_j \leftarrow f_j(\boldsymbol{x}_i) - f_{l_{\boldsymbol{x}}}(\boldsymbol{x}_i)$
6:     end for
7:     $l_n \leftarrow \arg\min_{j \neq k(x_0)} \frac{|f'_j|}{\|\boldsymbol{w}_j\|}$
8:     $r_i \leftarrow \frac{|f'_{l_n}|}{\|\boldsymbol{w}_{l_n}\|^2} \boldsymbol{w}_{l_n}$
9:     $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
10:    $i \leftarrow i + 1$
11: end while
12: return $\boldsymbol{r}^* = \sum_i \boldsymbol{r}_i$
```

can be seen from this description that there is a significant difference in complexity of FGS and Deepfool. Even with this case, the algorithm generally ends in a few iterations and on high end GPUs, it takes less than a second to compute adversarial examples for deep neural networks [32]. An example of this perturbation can be seen on Figure 7

As adversarial perturbations are additive noises, the perturbed image $\boldsymbol{x}+\boldsymbol{r}$ can be any point on the image space and it is not limited by the values of $\boldsymbol{x}$. A geometrically transformed image however, can only take values from the IAM. Thus, the adversarial perturbations are less limited and significantly different than the geometric transformations that this project focuses on. Nevertheless, the algorithms for finding either of them have similar methods since these algorithms both actively seek the misclassifying perturbed or transformed examples. In addition, the adversarial perturbations can be actively used in the algorithms for finding fooling transformations. Thus, although they are looking for different type of examples, the algorithms for finding misclassifying transformations have strong similarities with algorithms for finding adversarial perturbations, which can be seen in Sections 5 and 6

4 Problem Formulation

As outlined in Section 3.2, finding classifiers that are robust against geometric transformations is an important problem. To answer this problem, one has to compare different classifiers to understand which approaches are better in increasing the robustness against these transformations. Thus, we need a method to measure the invariance of a classifier to a given set of transformations $\mathcal{T}$. Similar to [13], we approach this problem by defining an invariance score $\Delta_{\mathcal{T}}(I, k)$ that captures the invariance of the classifier k against the transformations of the image I. We then propose an algorithm that can measure $\Delta_{\mathcal{T}}(I, k)$ given k, I and $\mathcal{T}$. Afterwards, we use this metric to compute the global invariance score of the classifier k to the transformations in $\mathcal{T}$.

Formally, our approach can be stated as follows: Let k be the given classifier, I be an image and $\mathcal{T}$ the set of transformations we are interested in. As with [13], we define the invariance metric as the minimal normalized distance from the identity transformation to a misclassifying one, which is denoted as

$$\Delta_{\mathcal{T}}(I, k) = \min_{\tau \in \mathcal{T}} \tilde{d}(e, \tau) \quad \text{subject to} \quad k(I) \neq k(I_\tau), \tag{4.1}$$

where $\tilde{d}$ is the normalized geodesic distance metric defined in (2.23). Unlike [13] however, our focus is to find a quick algorithm that can find a small enough transformation that changes the output label instead of finding the actual minimum.

Finally, for a probability distribution μ on the set of images, the global invariance score of the classifier k to transformations in $\mathcal{T}$ is defined as

$$\rho_{\mathcal{T}}(k) = \mathbb{E}_{I \sim \mu} \Delta_{\mathcal{T}}(I, k). \tag{4.2}$$

In practice, the underlying probability distribution is generally unknown. Because of this, the expectation is calculated using the empirical average over a set of training points:

$$\hat{\rho}_{\mathcal{T}}(k) = \frac{1}{m} \sum_{j=1}^{m} \Delta_{\mathcal{T}}(I_j, k). \tag{4.3}$$

In our case, this score is estimated using the output $\hat{\tau}$ of the constructed algorithm,

$$\hat{\rho}_{\mathcal{T}}(k) = \frac{1}{m} \sum_{j=1}^{m} \tilde{d}_{I_j}(e, \hat{\tau}). \tag{4.4}$$

In the following sections, we will describe two algorithms to find $\hat{\tau}$: First one, which we call Manifool, works on transformation Lie groups and uses the manifold structure to find $\hat{\tau}$. The second one, called iterative projection algorithm, uses the adversarially perturbed images to find a fooling transform.

5 Manifool

We propose an algorithm inspired from Deepfool in which we incorporate the manifold structure of the transformed images if the transformation is a Lie group. In our implementation, a convolutional neural network is used as a classifier whose inputs are defined as vectors in $\mathbb{R}^n$. As such, the discrete version of the image(from Section 2.1.1), denoted as $\boldsymbol{x}$, is used instead of the continuous version I.

5.1 Intuition

In the beginning of the algorithm, the input image lies on the IAM, $\mathcal{M}(\boldsymbol{x})$, which was defined in Section 2.2.2, as can be seen on Figure 5. Since we want to find a point $\boldsymbol{x}_\tau$ with $k(\boldsymbol{x}_\tau) \neq k(\boldsymbol{x})$, we need to move past the decision boundary(the green curve on the figure). Normally, as in the case of Deepfool algorithm(Section 3.3), the direction of this boundary can be found using the gradient of the classifier $\nabla f(\boldsymbol{x})$. However, we need to stay on the manifold $\mathcal{M}(\boldsymbol{x})$ and thus our movement directions are limited to the tangential space. So, the gradient is projected on the tangential space(as the red vector $\boldsymbol{u}$ in the figure) to find a direction which the image can move and which also gets us closer to the decision boundary. After this, the vector $\boldsymbol{u}$ on the tangential space is mapped on to $\mathcal{M}(\boldsymbol{x})$ to remain on the manifold. If the mapped point has passed the boundary, the algorithm can stop and return the transformation that corresponds to this point. If not, the same process with projecting the gradient onto the tangential space and afterwards mapping it onto the manifold is repeated until the algorithm passes the decision surface. A more formal examination of the algorithm will be done in next section.

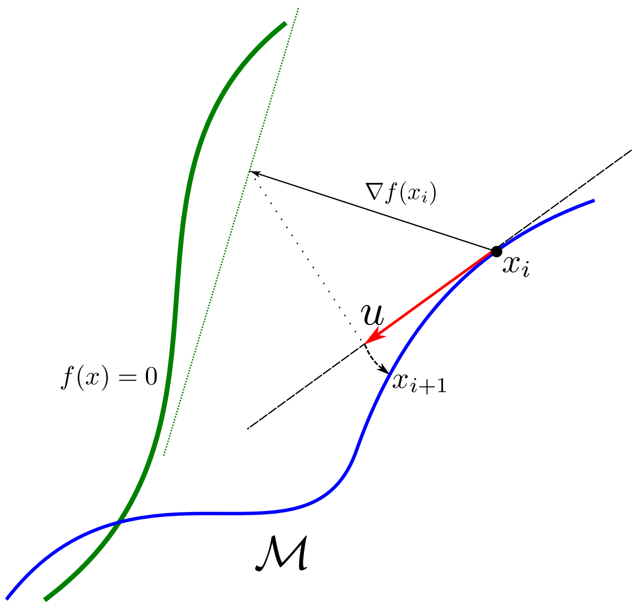


Figure 8: An illustration of the Manifool algorithm with a binary classifier on a 2D space. The manifold is shown in blue while the decision boundary ($f(\boldsymbol{x}) = 0$ in binary case) is shown in green. The algorithm uses the gradient of the classifier to find the direction of the boundary and moves in this direction by projecting the gradient to $T_{x_i}\mathcal{M}$ and mapping it to the manifold. This process repeats until the algorithm passes the decision boundary.

5.2 Description

Formally, the algorithm starts by setting the initial image as $\boldsymbol{x}_0 = \boldsymbol{x}$ where $\boldsymbol{x}$ denotes the input image. The label of this image is $l_{\boldsymbol{x}} = k(\boldsymbol{x})$, according to (3.1). The algorithm then begins iterating, starting with $i = 0$.

The iteration i starts with finding the movement direction for this step. In the case of Deepfool, this was done by using the gradients of the classifier for different labels with respect to the image $\boldsymbol{x}_i$ [32]. In our case, we are not interested in the function in the ambient space, and thus we restrict the classifier on the manifold as $f_{|\mathcal{M}}$ and use its gradients for determining the movement direction. These can be acquired simply by projecting the gradient onto the tangential space of the current point [1].

Therefore, formally, each iteration starts by calculating the basis of the tangential space, i.e. the columns of the Jacobian matrix $\boldsymbol{J}_{\boldsymbol{x_i}}$ as

$$(\boldsymbol{J}_{\boldsymbol{x}_i})_j = \frac{\partial \psi^{(\boldsymbol{x}_i)}(\tau(\theta))}{\partial \theta_i}, \tag{5.1}$$

where ψ is the orbit map defined in (2.17). The columns of $\boldsymbol{J}_{\boldsymbol{x_i}}$ are assumed to be linearly independent, but they are not orthonormal or even orthogonal. Thus, the the projection is done by solving the least squares problem $\arg\min_{\boldsymbol{u}\in\mathbb{R}^d} \|\boldsymbol{J}_{\boldsymbol{x}_i}\boldsymbol{u} - \boldsymbol{w}_j\|$. Here, similar to the Deepfool algorithm, $\boldsymbol{w}_j$ is defined as $\boldsymbol{w}_j = \nabla f_j(\boldsymbol{x}_i) - \nabla f_{l_{\boldsymbol{x}}}(\boldsymbol{x}_i)$ which are the directions that maximally increase $f_j(\boldsymbol{x}_i)$ and maximally decrease $f_{l_{\boldsymbol{x}}}(\boldsymbol{x}_i)$, thus getting us closest to the decision boundary $f_j(\boldsymbol{x}_i) = f_{l_{\boldsymbol{x}}}(\boldsymbol{x}_i)$. The least squares problem can be solved using the pseudoinverse of the Jacobian as

$$\boldsymbol{u}_j = \boldsymbol{J}^+_{\boldsymbol{x}_i}\boldsymbol{w}_j = (\boldsymbol{J}^T_{\boldsymbol{x}_i}\boldsymbol{J}_{\boldsymbol{x}_i})^{-1}\boldsymbol{J}^T_{\boldsymbol{x}_i}\boldsymbol{w}_j. \tag{5.2}$$

The label with the nearest boundary can be found as

$$l_n = \arg\min_{j\neq l_x} \frac{|f_j(\boldsymbol{x}_i) - f_{l_x}(\boldsymbol{x}_i)|}{\|\boldsymbol{u}_j\|_{G_{x_i}}}. \tag{5.3}$$

Note that since $\boldsymbol{u}$ is in the tangential space of the manifold, we are using the Riemannian metric from (2.22) instead of the Euclidean one. Before the mapping step, the movement vector on tangential space is scaled as

$$\boldsymbol{u} = \lambda_i \frac{\boldsymbol{u}_{l_n}}{\|\boldsymbol{u}_{l_n}\|} \tag{5.4}$$

where λ_i is a step size parameter that is chosen to maximize the decrease in $f_j(R(\boldsymbol{u})) - f_{l_{\boldsymbol{x}}}(R(\boldsymbol{u}))$ to get as close as possible to the decision boundary.

The second step in the iteration is the mapping of $\boldsymbol{u}_{l_n} \in T_{\boldsymbol{x}_i}\mathcal{M}(\boldsymbol{x})$ onto $\mathcal{M}(\boldsymbol{x})$. This step depends heavily on the transformation set. As we want to minimize the geodesic distance, the natural choice of mapping would be to use exponential map. If an exponential map is readily available for $\mathcal{M}$ and does not have high computational complexity, it can be used. However, for most transformation sets, this does not hold and a retraction is used instead. Here, we will talk about one such retraction for the set of projective transformations and its subsets.

The retraction in our implementation is done using the exponential map of the matrix Lie group counterparts of projective transformations, i.e. matrix exponentials of the respective Lie algebras.

Let us define as $h : \mathcal{T}_m \to \mathcal{T}$ the map between the matrix Lie group $\mathcal{T}_m$ and its transformation group counterpart $\mathcal{T}$. Also let $\boldsymbol{y} \in \mathcal{M}(\boldsymbol{x})$, $\boldsymbol{u} \in T_{\boldsymbol{y}}\mathcal{M}(\boldsymbol{x})$ and G_i be the generators of $\mathfrak{t}$, the Lie algebra of the matrix Lie group $\mathcal{T}_m$. Then, the retraction at the point $\boldsymbol{y}$, $R_{\boldsymbol{y}} : T_{\boldsymbol{y}}\mathcal{M} \to \mathcal{M}$ can be written as

$$R_{\boldsymbol{y}}(\boldsymbol{u}) = \psi_{\mathcal{T}}^{(\boldsymbol{y})}\left(\exp\left(\sum_i u_i G_i\right)\right) \tag{5.5}$$

where $\psi_{\mathcal{T}} = \psi \circ h$ represents the orbit map from the matrix Lie group $\mathcal{T}_m$ to $\mathcal{M}(\boldsymbol{x})$. This means that the chosen vector $\boldsymbol{u}_{l_n}$, is first mapped to the Lie algebra $\mathfrak{t}$ using the generators, then it is mapped to the matrix Lie group $\mathcal{T}$ using the exponential map and lastly, back to the manifold using the orbit map from this group to the image. The image for the next iteration is thus written as

$$\boldsymbol{x}_{i+1} = R_{\boldsymbol{x}_i}(\boldsymbol{u}) \tag{5.6}$$

Along the way, the transformation used for generating this image can also be found as

$$\tau_i = h\left(\exp\left(\sum_i u_i G_i\right)\right) \tag{5.7}$$

Lastly, the label of the generated image is checked. If $k(\boldsymbol{x}_{i+1}) = l_{\boldsymbol{x}}$, the algorithm continues with the next iteration, this time using $\boldsymbol{x}_{i+1}$. If $k(\boldsymbol{x}_{i+1}) \neq l_{\boldsymbol{x}}$, then the algorithm has finished successfully and the transformation that generated this image

$$\tau = \tau_0 \circ \tau_1 \circ \ldots \tau_i \tag{5.8}$$

is returned. If the intermediate transformations has not been found along with the retraction, the output transformation can be found using the inverse orbit map as

$$\tau = (\psi^{(\boldsymbol{x})})^{-1}(\boldsymbol{x}_{i+1}) \tag{5.9}$$

The algorithm is summarized on Algorithm 2. Overall, it should be noted that our algorithm is closely related to manifold optimization techniques, particularly to line-search methods. The convergence analysis such methods can be found for example in [1].

5.3 Implementation

In this section, the various issues about the implementation of the algorithm is discussed. Our implementation is done in Python using PyTorch [33] for CNN implementations. As the implementation is very similar to an optimization algorithm,$|f_j(\boldsymbol{x}_i) - f_{l_x}(\boldsymbol{x}_i)|$ is labeled as the objective function throughout this section.

First issue is the step size parameter λ_i, which was not explained in the section. During the implementation, it has been seen that a constant step size does not perform well. If large, a constant step size can cause an increase in the objective function in some steps, i.e. it gets further away from the boundary. However, if the step size is low, then it causes the algorithm to iterate for a longer time. Because of this, in the current implementation, direct line-search is used for finding λ_i. This is done by computing $\boldsymbol{u}_t$ for a range of values λ_t where $t \in 0, 1, ...N$, computing $\boldsymbol{x}_t = R_{\boldsymbol{x}_i}(\boldsymbol{u}_t)$ for each of them and feeding them altogether to the network to get $|f_j(\boldsymbol{x}_t) - f_{l_x}(\boldsymbol{x}_t)|$. The λ_t that

Algorithm 2 Multiclass Manifool Algorithm

```
1: Initialize with $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
2: while $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ do
3:     $\boldsymbol{J} \leftarrow \boldsymbol{J}_{\boldsymbol{x}_i}$
4:     for $j \neq \hat{k}(x_0)$ do
5:         $\boldsymbol{w}_j \leftarrow \nabla f_j(x_i) - \nabla f_{l_{\boldsymbol{x}}}(x_i)$
6:         $f'_k \leftarrow f_k(x_i) - f_{\hat{k}(x_0)}(x_i)$
7:         $\boldsymbol{u}_j \leftarrow \boldsymbol{J}^+ \boldsymbol{w}_j$
8:     end for
9:     $l_n \leftarrow \arg\min_{j \neq l_x} \frac{|f_j(\boldsymbol{x}_i) - f_{l_x}(\boldsymbol{x}_i)|}{\|\boldsymbol{u}\|_{G_{x_i}}}$
10:    $\boldsymbol{u} \leftarrow \lambda_i \frac{\boldsymbol{u}_{l_n}}{\|\boldsymbol{u}_{l_n}\|}$
11:    $x_{i+1} \leftarrow R_{\boldsymbol{x}_i}(\boldsymbol{u})$
12:    $i \leftarrow i + 1$
13: end while
14: return $\tau = (\psi^{(\boldsymbol{x})})^{-1}(\boldsymbol{x}_{i+1})$
```

minimizes this function is chosen as the step size and $\boldsymbol{x}_{i+1}$ is set as $\boldsymbol{x}_t$ that was computed using λ_t. The result is a consistent and smooth output for the algorithm.

Another issue is that $|f_j(\boldsymbol{x}_t) - f_{l_x}(\boldsymbol{x}_t)|$ is not a convex function. Because of this, the algorithm can get stuck in a local minimum without ever finding a 'fooling' transformation. A similar phenomenon is caused by the step in 5.3 (step 9 on Algorithm 2), where the two different gradients with opposite directions can be chosen in consecutive steps. This can cause the algorithm to oscillate between two different labels without moving and thus never finding a transformation. As can be seen on Table 2, the issues with getting stuck on certain points are more apparent in very low dimensional transformations such as translations.

Third issue is the computational complexity of the algorithm. In general, for reasonably deep networks, the complexity of one iteration is dominated by the first step, i.e. choosing the movement direction. Let m be the number of outputs (labels) of the network. Then, the complexity of this step is mt_{bp} where t_{bp} is the complexity of the backpropagation for the network that is used for computing the gradient $\nabla f_j(x_i)$. If, instead of trying all m candidates, only n_c more probable labels are chosen, the duration for this step can be reduced to $n_c t_{bp}$. Since l_n is chosen as

$$l_n = \arg\min_{j \neq l_x} \frac{|f_j(\boldsymbol{x}_i) - f_{l_x}(\boldsymbol{x}_i)|}{\|\boldsymbol{u}_j\|_{G_{x_i}}}. \tag{5.10}$$

without knowing $\boldsymbol{u}_j$, the labels with lower $|f_j(\boldsymbol{x}_t) - f_{l_x}(\boldsymbol{x}_t)|$ values are expected to be the more probable than labels with higher values. Thus, the n_c labels with lowest $|f_j(\boldsymbol{x}_t) - f_{l_x}(\boldsymbol{x}_t)|$ values are chosen as the most probable and only the gradients for these labels are calculated. Thus, the complexity of the algorithm can be reduced from mt_{bp} to $n_c t_{bp}$.

For a constant classifier, as d, the dimensionality of the transformation set is increased, the complexity of one iteration dominated by step 7 of Algorithm 2, where $\boldsymbol{w_j}$ is projected on to the manifold by computing $\boldsymbol{J}^+_{x_i}\boldsymbol{w}_j$. As $\boldsymbol{J}_{x_i}$ is a matrix of size $n \times d$, the psuedoinverse requires $O(nd^2)$ operations [15]. Thus, for a constant classifier with set input size, the complexity of a single iteration increases quadratically with number of dimensions of the transformation.

Last issue is about interpolation. In the mapping step of the algorithm(step 11 in Algorithm 2), $R_{\boldsymbol{x}_i}$ might include transformation of the image $\boldsymbol{x}_i$. As explained in Section 2.1.1, transformation of a discrete image requires interpolation to realize. This effect distorts the image alongside the effect of transformation. As such, if $\boldsymbol{x}_{i+1} = R_{\boldsymbol{x}_i}(\boldsymbol{u})$ is computed by transforming the image $\boldsymbol{x}_i$, which in turn is a transformed version of image $\boldsymbol{x}_{i-1}$, the distortion from generating $\boldsymbol{x}_i$ will propagate to x_{i+1} and added to the distortion transforming $\boldsymbol{x}_i$. It can be seen that if R_{x_i} is calculated this way, error caused by interpolation will propagate between images and increase the total distortion in each step. Thus, if $R(x)$ requires transformation to be computed, this transformation should always be done using the input image $\boldsymbol{x}$ to prevent this propagation of the distortion caused by interpolation.

5.4 Experiments

We now test the Manifool algorithm on convolutional neural network architectures for MNIST [26] and ImageNet (ILSVRC 2012) [36] image classification datasets. We consider the projective transformations group and its subsets of translations, similarity and affine transformations. Output examples for these transformations using the images from these two datasets can be found on Figures 10 and 12 respectively. In all cases, the transformed images have the same size as the original one. Bilinear interpolation with zero-padding boundary conditions is used for transformations.

5.4.1 Validity of direct distance as a metric

As our first experiment, we wanted to be sure that direct distance method from Section 2.3.3, can be used as a metric in our experiments. To this end, this method is compared with FMM, again from Section 2.3.3, by calculating the normalized geodesic distance between the identity transformation e and 3 randomly generated translations in 200 different images in the MNIST dataset [26]. The

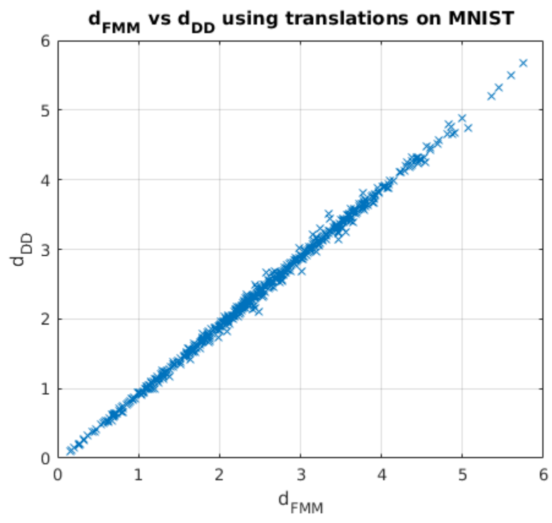


Figure 9: Comparison of distances found using FMM and direct distance method. Although some differences exist between two methods, the outputs show high correlation. The negative bias on the direct distance output is also visible as the samples are not exactly on the identity line

	Mean Value	Standard Dev.	Max. Value	Min. Value
d_{FMM}	2.5209	1.0408	5.7437	0.1591
$d_{DD} - d_{FMM}$	-0.1144	0.0530	0.1720	-0.3723
$\frac{d_{DD}-d_{FMM}}{d_{FMM}}$	0.0555	0.0404	0.0514	-0.4072

Table 1: The statics for the comparison of geodesic distances calculated using fast marching method(denoted as d_{FMM}) and distance calculated using the direct distance method(denoted as d_{DD})

translations in x and y directions are generated independently using a discrete uniform distribution between $[-6, 6]$ with 0.1 steps . For FMM, the manifold is discretized with 0.1 steps as well.

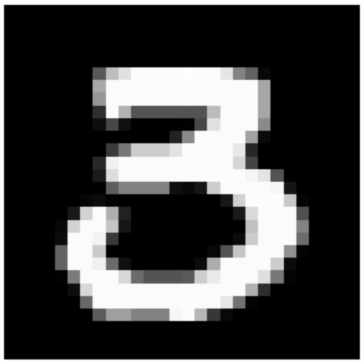
The resulting statistics can be seen on Table 1. The error has a negative bias which is expected as L^2 distance underestimates the length of the path between two points. The large minimum percent error(40%) is mainly caused by this bias at one of the low distance points. The bias of the estimation is not important to us because the ultimate goal of this metric is to compare different transformations. If $\tilde{d}(e, \tau_1) < \tilde{d}(e, tau_2)$, then for a fixed bias b, $\tilde{d}(e, \tau_1) - b < \tilde{d}(e, tau_2) - b$, i.e. the comparison between them are the same even if the metric is biased. As such, the standard deviation of the error is more important when evaluating the performance of the metric and 0.053 deviation in score or 4% deviation in percentage is low enough. Also, the correlation coefficient between FMM and this method is 0.9988, which shows they are highly correlated regardless of the error. This high correlation can be better seen on Figure 9 as the distance found using direct distance is basically linear with respect to the distance found using FMM. As it is quite accurate in addition to being fast, we will use this algorithm to calculate the geodesic distance when using it as a metric is possible.

5.4.2 Handwritten digits dataset

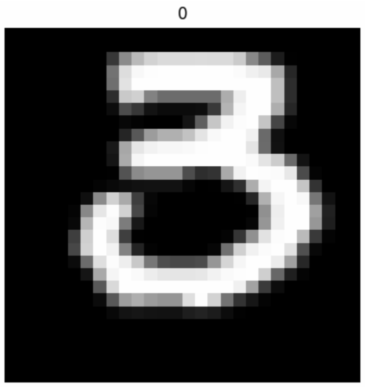
As the second experiment, we compare the proposed Manifool algorithm to the Manitest method that was introduced in 3.2. By using FMM, Manitest guarantees to find the transformation with (discretized) minimal distance and outputs the distance it has find thorough FMM. However, instead of using the output score of Manitest, we compute the distance introduced in (2.23) that uses direct distance. As it was validated in the previous experiment, we prefer using it as this metric for both methods since it is faster than using FMM for Manifool.

The comparison is done using a simple CNN architecture with two hidden convolutional layers with 5×5 filters and 32 and 64 feature maps respectively. ReLU is used as the nonlinear activation function with 2×2 max pooling with a stride of 2 to do subsampling. This is the exact same network used in the Manitest paper along with the parameters. The experiment is done using 1000 random images from the MNIST dataset, where the scores and the running time for both methods are only recorded if the Manifool method is successful. For low dimensional transformations ($d \leq 4$), the output has been computed both for Manifool and Manitest. However, only the score from Manifool is computed for affine and projective groups, since for these transformations, the average running time for Manitest becomes higher than what is feasible.

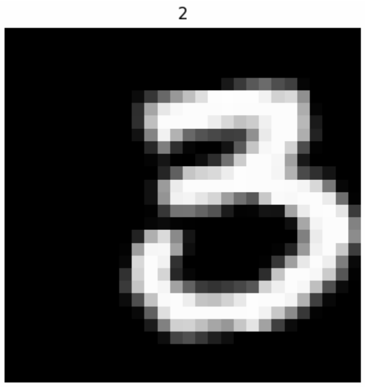
Table 2 reports these invariance scores $\hat{\rho}_{\mathcal{T}}(f)$ for both methods, as well as the success percentage of Manifool and the computation duration for both methods. On average, as expected, Manitest



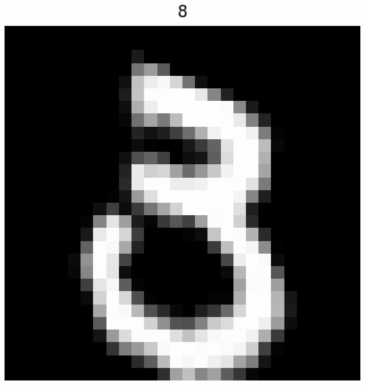
(a) Original image



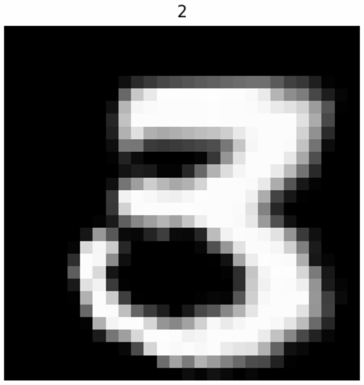
(b) Translation($\tilde{d}(e, \tau) = 1.757$)



(c) Similarity($\tilde{d}(e, \tau) = 1.697$)



(d) Affine($\tilde{d}(e, \tau) = 1.444$)



(e) Projective($\tilde{d}(e, \tau) = 1.139$)

Figure 10: Examples of fooling transformations for an image from MNIST dataset. The numbers on the top of the images show the output of the classifier, $k(x_\tau)$ for these images while the transformation set and corresponding score are written below the digits.

outputs a transformation with a lower score, as it guarantees to output the minimal transformation. As such, for low dimensional transformations such as translations, Manitest is inarguably better as it is both reasonably fast and successful, while Manifool has a tendency to get stuck in local minimums, as explained in 5.3, which causes it to have a low success percentage. On the other hand, it gets better as dimension is increased and the duration it requires does not change significantly.

In both cases, the invariance score is decreasing with the dimensionality of the transformation since the lower dimensional transformations are the subsets of the higher dimensional transformations. For example, the minimal distance for the similarity set($\mathcal{T}_{sim}$) cannot be higher than the minimal distance for translations($\mathcal{T}_{tr}$), because at its maximum, the minimal transformation for similarity must be the same with the translations since $\mathcal{T}_{tr} \subset \mathcal{T}_{sim}$ i.e. every transformation in $\mathcal{T}_{tr}$ is also in $\mathcal{T}_{sim}$.

As Manitest uses FMM, its complexity increases exponentially with the dimension of the transformation group, as explained in Section 2.3.3. However, as shown in Section 5.3, the per iteration complexity of Manifool is only quadratic with respect to the dimension. This is clearly seen on Table 2, where the running time of the algorithm increases drastically for Manitest, while it remains low for Manifool.

	Manifool			Manitest	
Transformation	Fooling Percentage	Avg. Norm. Direct Distance	Avg. Run Time per Image	Avg. Norm. Direct Distance	Avg. Run Time per Image
Translation	60.7%	1.633	337 ms	1.377	2.52 s
Similarity	96%	1.536	218 ms	1.228	27.5 s
Affine	99.1%	1.374	166 ms	-	-
Projective	99.2%	1.135	188 ms	-	-

Table 2: Results for using Manifool with various transformation sets and its comparison with Manitest.

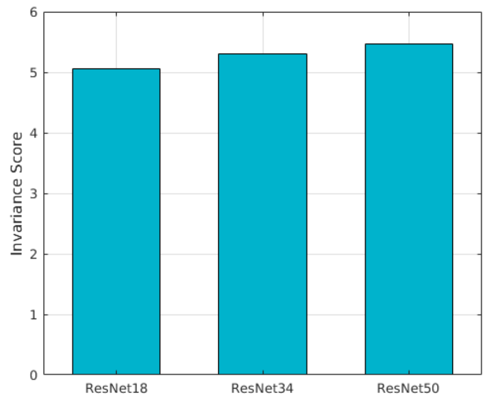
Lastly, the scores on Table 2 are on average and there are cases where Manifool finds a point with lower distance than Manitest. Generally this is caused by the grid structure Manitest uses as a fooling point is missed if it is not on the grid. However, Manifool does not have such restrictions and sometimes can find these points and thus output a lower score. Manitest would require a tighter grid and thus a longer duration to find these points, which, depending on the increase in duration, might not be feasible.

5.4.3 Natural images

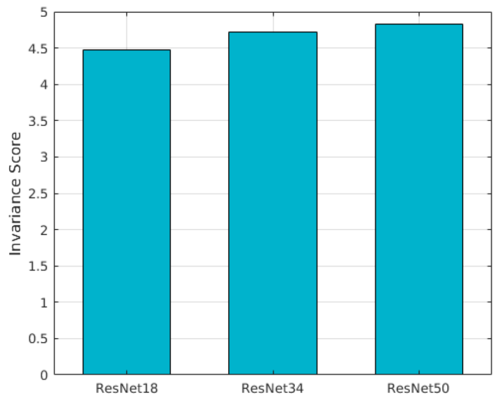
For the last experiment, ImageNet dataset is used for computing the invariance scores of three pre-trained networks: ResNet-18, ResNet-34 and ResNet-50 [21]. All of these networks have similar architectures and the numbers represent the depth of these networks. They accept 224×224 images, and thus before feeding the images into the network, the input images are resized to have 256 pixels on the shorter side and the center 224×224 is cropped. The transformations during the algorithm are done on the resized image to limit boundary issues and the same crop is applied afterwards. For this cropping method, top-1 error percentages for these networks are 30.24%, 26.7% and 23.85% respectively.

The experiment is done using 1000 random images from the ILSVRC2012 validation set. Only invariance scores for similarity, affine and projective transformations are computed, as the previous experiment has showed that the algorithm does not work well on low dimensional manifolds.

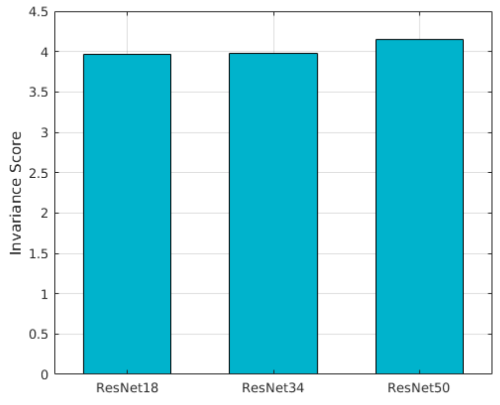
The results can be seen in Figure 11. The output scores show the increasing nature of the invariance with number of layers of the network against these three groups of transformations. This result is in agreement with the previous empirical studies such as [17], and the results from the Manitest paper [13]. These results however, either measure the invariance in one dimensional transformation groups (e.g. rotation or dilation) or can only be applied on low dimensional manifolds. Using Manifool, we have shown that these results can be reproduced for reasonably deep networks as well. Also, similar to the previous experiment, we observe that the invariance scores decrease as the dimension of the transformation set increases. This has the same reason with the previous experiment, i.e. the lower dimensional transformations are subsets of higher dimensional ones.



(a) Similarity



(b) Affine



(c) Projective

Figure 11: Invariance scores of CNNs with different depths on similarity, affine and projective transformation groups for ImageNet dataset.

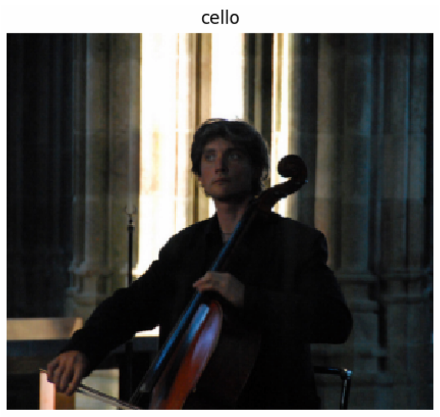
5.5 Conclusion

As we have shown, the Manifool method is an interesting algorithm for quickly measuring the invariance of a neural network to geometric transformations with a Lie group structure. When compared to Manitest, although it does not necessarily find the minimal transformation, it can still find small enough transformation examples from a set of transformations with $d \geq 4$. It can also compute these invariance scores for reasonably deep networks in a reasonable amount of time. These results are accurate enough to confirm our previous beliefs on the relation of invariance and number of layer, even for deep neural networks. However, it also has certain drawbacks:

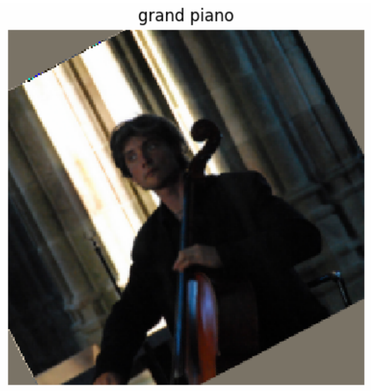
- The algorithm only works for Lie groups and even for them, it requires a retraction to be computed in an efficient fashion. This means there are many sets of transformations (e.g. integer translations) that the Manifool cannot be used efficiently.
- The algorithm does not work efficiently on very low dimensional transformation sets such as translations. The low degrees of freedom and the non-convexity of the score difference can

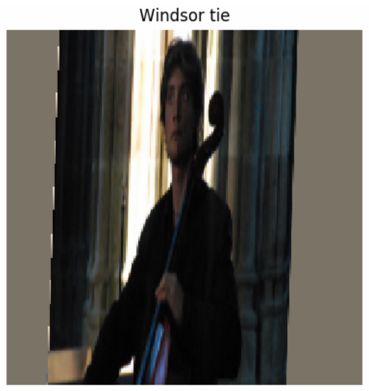
sometimes cause the output to get stuck in local minima. For these sets of transformations, Manitest might be a better choice to compute invariance score than Manifool as it is both more accurate and has a higher chance of finding a transformation.

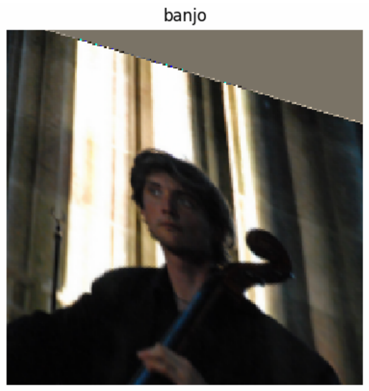
In the next section, we propose another algorithm which can solve the first issue.



(a) Original image







(b) Similarity($\tilde{d}(e, \tau) = 4.737$) (c) Affine($\tilde{d}(e, \tau) = 4.426$) (d) Projective($\tilde{d}(e, \tau) = 4.212$)

Figure 12: Examples of fooling transformations for an image from ImageNet dataset. The labels on the top of the images show the output of the classifier, $k(x_\tau)$ for these images, while the transformation set and corresponding score are written below the digits.

6 Iterative Projection Algorithm

In this section, we propose another algorithm that can work on sets of transformations that are not necessarily Lie groups. Similar to the previous algorithm, the classifier is a neural network, denoted as $f : \mathbb{R}^n \to \mathbb{R}^m$ and the discrete version of the image $\boldsymbol{x}$ is used instead of the function I. The transformations are defined using a finite number of parameters denoted as $\theta \in \mathbb{R}^d$.

6.1 Intuition

The existence of adversarial perturbation methods mean that for a neural network, one can find an image $\hat{\boldsymbol{x}}$ that is close to its original version $\boldsymbol{x}$ but also on the other side of the decision boundary, or at least, closer to the boundary than the original one. By projecting this point onto the set of transformed images, an example that is closer to the decision boundary can be found. If this transformed image fools the classifier, then we have our transformation as the one which generated this image. If not, we can do the same process over again, until we reach the other side of the decision boundary. Thus, the algorithm is composed of two alternating steps: finding an adversarial example and finding the transformation that creates the point closest to this example. An illustration of these two steps are seen on Figure 13.

6.2 Description

Formally, similar to Manifool, the algorithm starts by setting the initial image as $\boldsymbol{x}_0 = \boldsymbol{x}$ and finding the label of the original image $l_{\boldsymbol{x}} = k(\boldsymbol{x})$. Then the algorithm begins iterating, starting with $i = 0$.

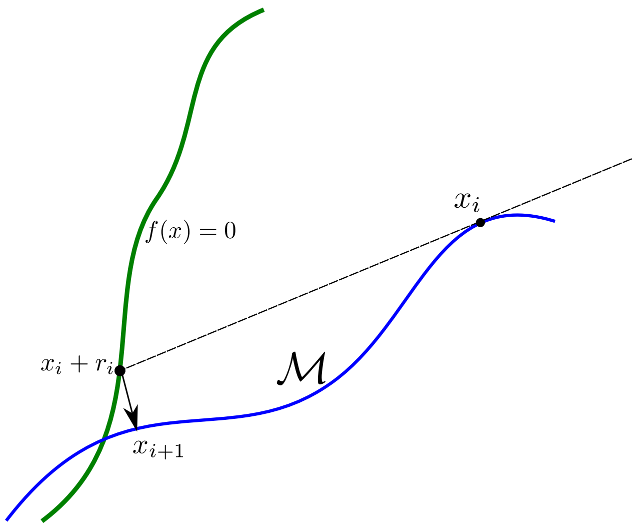


Figure 13: An illustration of the Iterative Projection algorithm with a binary classifier on a 2D space. The set of transformed images form a manifold that is shown in blue while the decision boundary ($f(\boldsymbol{x}) = 0$ in binary case) is shown in green. An adversarial example is found on the tangential space of the manifold to prevent the original point to be the closest to the adversarial example. Then, the closest point on the manifold to this example is found. This process repeats until the algorithm gets a point on the other side of the decision boundary.

The iteration i starts by finding an adversarial example using $\boldsymbol{x}_i$. This can be done using any adversarial perturbation method on neural networks, such as Deepfool or fast gradient sign(FGS) method. The perturbation is denoted as $\boldsymbol{r}_i$ and the perturbed example is written as

$$\hat{\boldsymbol{x}} = \boldsymbol{x}_i + \boldsymbol{r}_i. \tag{6.1}$$

As the next step finds the closest point to this adversarial example, if possible, the method should be modified in a way that prevents $\boldsymbol{x}_i$ to be the closest point in the set to $\hat{\boldsymbol{x}}$. For example, if the set transformed images form a manifold, Deepfool can be used while restricting the classifier to the tangential space of the current point. This way, $\boldsymbol{r}_i$ is found on the tangential space, which means $\boldsymbol{x}_i$ cannot be the closest point to $\hat{\boldsymbol{x}}$, as the perturbation must be perpendicular to the tangent space for $\boldsymbol{x}_i$ to be the closest.

The second step of the iteration is finding the minimizing transformation. Formally, this problem can be written as

$$\tau_{i+1} = \arg\min_{\tau \in \mathcal{T}} \frac{1}{2} \|\boldsymbol{x}_\tau - \hat{\boldsymbol{x}}\|_2^2. \tag{6.2}$$

Or, equivalently,

$$\theta_{i+1} = \arg\min_{\theta \in X} \frac{1}{2} \|\boldsymbol{x}_{\tau(\theta)} - \hat{\boldsymbol{x}}\|_2^2, \tag{6.3}$$

where θ are the parameters of the transformation τ.

This step is crucial to be solved efficiently in order to have a working algorithm. In our implementation, gradient descent is used for finding a local minimum to this problem. This means the transformation must be differentiable, as we need the gradient of the image with respect to the transformation. The descent is initialized with $\theta^{(0)} = \theta_i$ to start from the current transformation and the iterations start from $j = 0$. The parameters at iteration j is denoted as $\theta^{(j)}$. For this, the gradient of the objective function C can be written as

$$\nabla C(\theta^{(j)}) = (\boldsymbol{J}_{\boldsymbol{x}_{\tau(\theta^{(j)})}})^T (\boldsymbol{x}_{\tau(\theta^{(j)})} - \boldsymbol{x}) \tag{6.4}$$

where $\boldsymbol{J}^{(\boldsymbol{x}_{\tau(\theta)})}$ is the Jacobian matrix. The update state is then written simply as

$$\theta^{(j+1)} = \theta^{(j)} - \lambda_j \frac{\nabla C(\theta^{(j)})}{\|\nabla C(\theta^{(j)})\|} \tag{6.5}$$

where λ_j is the step size of the gradient that is found by backtracking line search [3]. The gradient descent is terminated when the amount of decrease is lower than threshold.

Furthermore, one can regularize the objective function to get an output with certain desirable features. For example, adding the term $\eta\|\theta\|_2^2$ to the objective function in (6.3) gives us an output with smaller parameters, or a quadratic term $\eta\theta^T L\theta$ where L is a Laplacian matrix can be added to get a smoother output. In both cases, η term is the regularization parameter, which would determine how much the regularization will affect the output. The effect of these regularizers can be seen in Figure 14.

Similar to the previous algorithm, the algorithm terminates $k(\boldsymbol{x}_{i+1}) \neq l_{\boldsymbol{x}}$. $\tau(\theta_{i+1})$ is then returned as the fooling transformation.

The algorithm including regularization is summarized in Algorithm 3.

Algorithm 3 Iterative Projection Algorithm

```
1: Initialize with $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
2: while $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ do
3:     $\boldsymbol{r}_i \leftarrow \text{ADVPERTURB}(\boldsymbol{x}_i)$
4:     $\hat{\boldsymbol{x}} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
5:     $\theta_i \leftarrow \arg\min_{\theta \in X} \left\{ \frac{1}{2} \|\boldsymbol{x}_{\tau(\theta)} - \hat{\boldsymbol{x}}\|^2 + \eta Reg(\theta) \right\}$
6:     $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_{\tau(\theta_i)}$
7:     $i \leftarrow i + 1$
8: end while
9: return $\tau = \tau(\theta_i)$
```

6.3 Experiments

We now test the second algorithm on MNIST and ImageNet datasets. This time, in addition to the projective transformations and its subsets, we also consider the set of 'general transformations'. We define this transformations by giving an independent movement vector to every pixel, i.e. for every (x, y) in the support of the image, there exists θ_x, θ_y such that the transformation is defined as

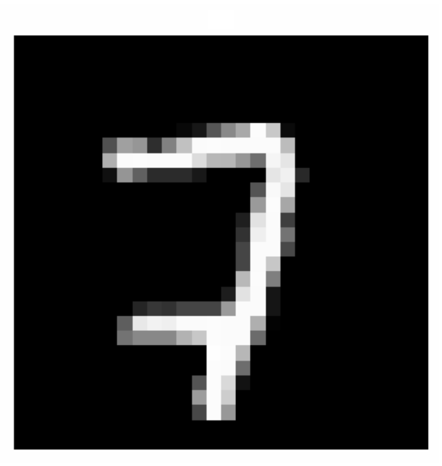
$$\tau(x, y) = (x + \theta_x, y + \theta_y) \tag{6.6}$$

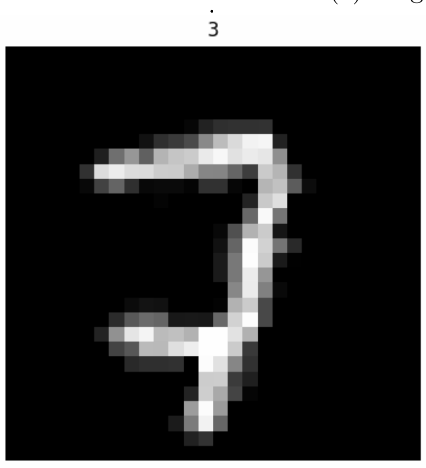
For grayscale images, this set of transformations has higher number of dimensions than the image itself and thus IAM is not embedded in the discrete images space. Thus, Manifool cannot be applied in this case. However, iterative projection algorithm allows us to compute fooling transformations in this set. Examples of such transformations can be found in Figure 14 and 19 respectively. In all cases, similar to the previous experiments, the transformed images have the same size as the original one. Bilinear interpolation with zero-padding boundary conditions is used for transformations.

6.3.1 Effect of different adversarial perturbations on the algorithm

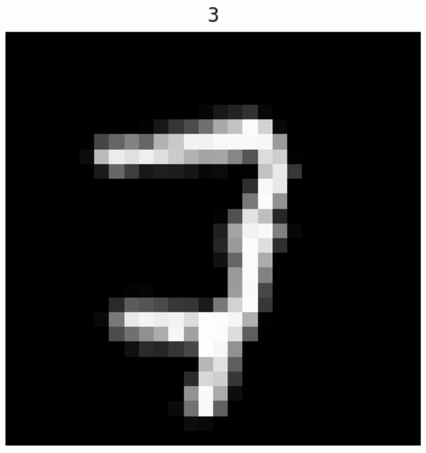
In the first experiment, the effect of the chosen adversarial example on the algorithm is examined by comparing two possible choices: fast gradient sign and Deepfool. This test is done on ImageNet dataset using the previously defined set of 'general transformations' and ResNet-50 as our classifier. Because this transformation set is not an embedded manifold, we cannot use our previous metric. Thus, we have to use the L^2 metric stated in (2.19) to compute the invariance score, which becomes the Euclidean distance between the transformed and original image for discrete images. The distance is normalized, similar to (2.23).

As explained in Section 3.3, although both Deepfool and FGS are trying to find an adversarial example, there are some differences. Deepfool tries to minimize the Euclidean distance between the perturbed example and the original image, while FGS uses a single step of gradient to generate the perturbation. Because of this, the output of Deepfool, $\boldsymbol{r}_{df}$ has a lower norm than its FGS counterpart, and it is more localized as can be seen on Figure 7. When these methods are used in the algorithm, their properties pass onto the output. Thus, using Deepfool will return a more localized transformation with a small norm while using FGS will return a transformation with a higher norm. The examples of this can be seen on Figure 15.

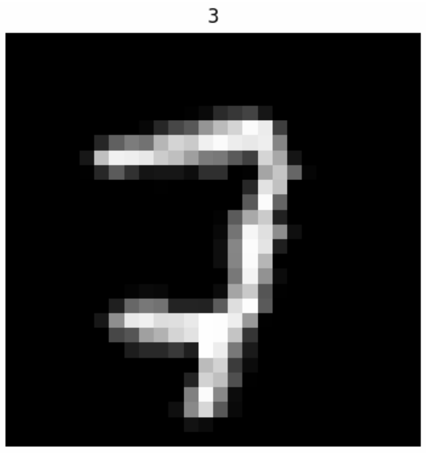

(a) Original image



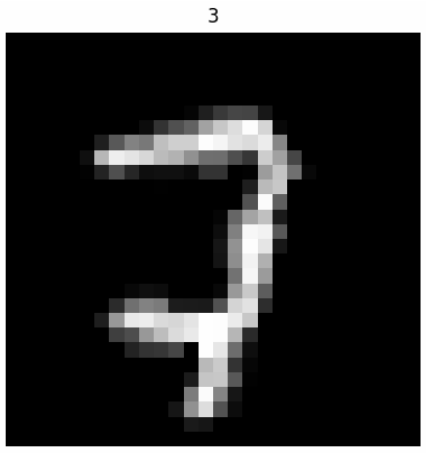
(b) Image generated using FGS (Normalized L^2 metric : 0.248)



(c) Image generated using Deepfool (Normalized L^2 metric : 0.185)



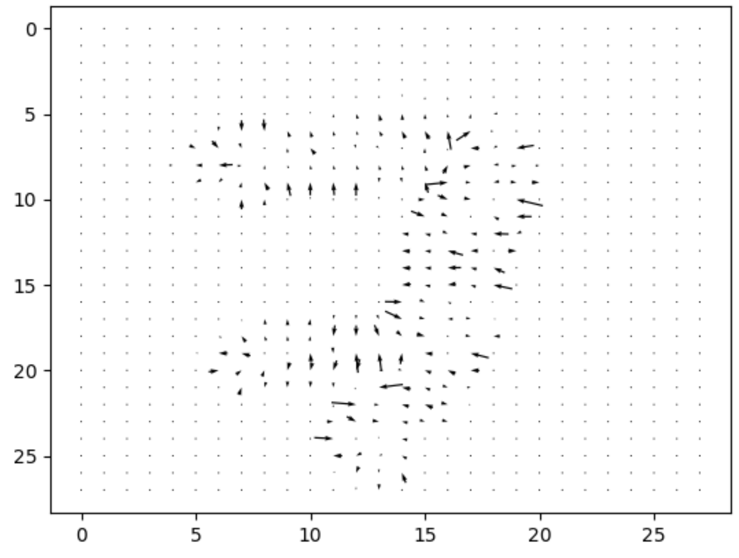
(d) Image generated using FGS and l_2 regularization (Normalized L^2 metric : 0.249)



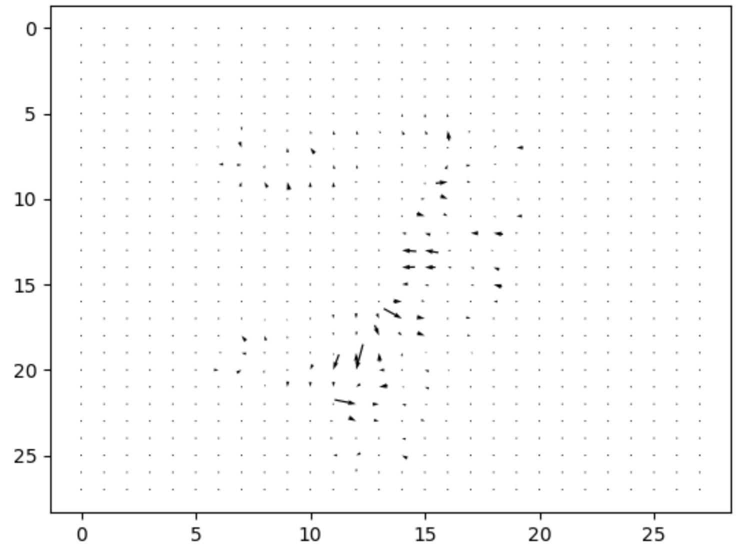
(e) Image generated using FGS and smoothing regularization (Normalized L^2 metric : 0.287)

Figure 14: Examples for fooling 'general transformations' for an image from MNIST database. It can be seen that image generated using FGS is distorted more than the image generated using Deepfool. The movement vectors for these transformations can be seen on Figure 15 and 16.

.



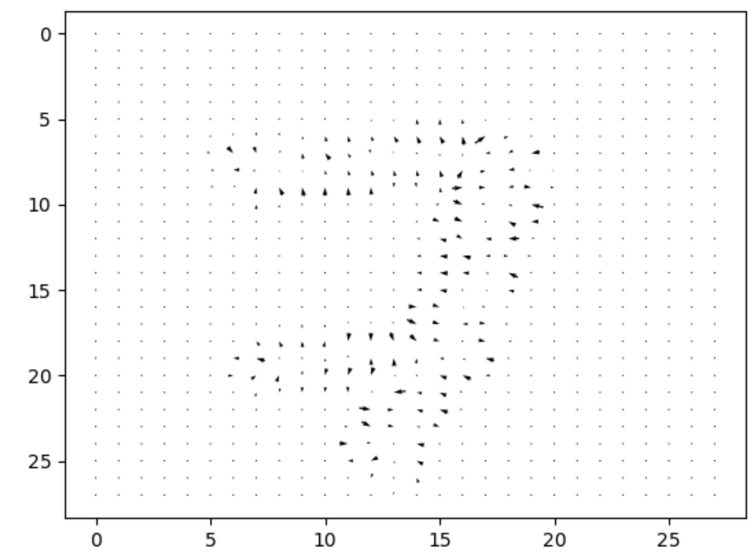
(a) Movement parameters of the image generated using FGS



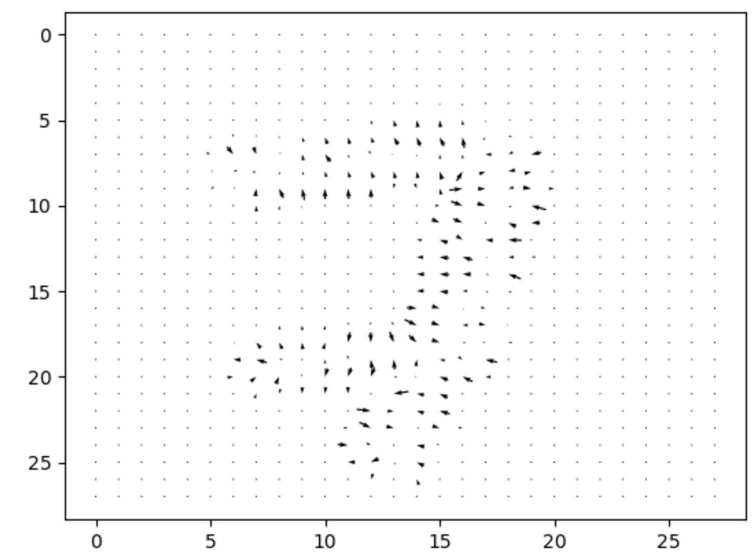
(b) Movement parameters of the image generated using Deepfool

Figure 15: Movement vectors of the transformations in Figure 14. It can be seen that, compared to FGS, using Deepfool generates much more localized vectors.

.



(a) Movement parameters of the image generated using FGS and l_2 regularization



(b) Movement parameters of the image generated using FGS and smoothing regularization

Figure 16: Movement vectors of the regularized transformations in Figure 14. The effects of regularization can be seen clearly where l^2 regularization decreases the norms of the movement in general while smoothing regularization outputs a smoother transformation

In addition to using different perturbation methods, changing the parameters of these methods can also affect the output. For example, FGS method has the parameter ϵ which determines the l_∞ norm of the returned perturbation. To see the effect of this parameter on our algorithm, we computed the score of the classifier for 1000 images from the ImageNet dataset for different values of epsilon. The resulting graph can be seen on Figure 17. It can be seen that the time it takes to compute the transformation decreases as the parameter is increased; while the resulting score increases approximately linearly. So, if the speed is more important than the accuracy of the invariance score, a larger ϵ can be used. In our tests, if FGS is used, we use $\epsilon = 0.014$ as a good compromise between duration and accuracy.

For the case of Deepfool, we realized that the algorithm starts to have numerical issues as it gets closer and closer to the boundary and thus as transformations in each step gets smaller and smaller. To combat this issue, instead of using the output of Deepfool directly as $\hat{\boldsymbol{x}} = \boldsymbol{x}_i + \boldsymbol{r}_{df}$, we can use an overshoot factor α and get a further point as $\hat{\boldsymbol{x}} = \boldsymbol{x}_i + (1+\alpha)\boldsymbol{r}_{df}$. This overshoot factor is another parameter and the same test with ϵ is done for it as well. The resulting graph is seen in Figure 18. Similar to the previous case, the duration is decreasing while the score is increasing semi-linearly. Again, if the speed is more important than the accuracy of the invariance score, a larger α can be used to compute the transformation. In our tests, to limit the duration of the computation, we used $\alpha = 1$ when we used Deepfool in the algorithm.

6.3.2 Comparison with Manifool

In this section, we compare the performance of the two proposed algorithms, i.e. Manifool and iterative projection. Deepfool is chosen as the adversarial perturbation method for the iterative projection algorithm. The invariance scores are computed using the metric from (2.23), which was previously used in the experiments for Manifool. Each score is calculated using 1000 random images

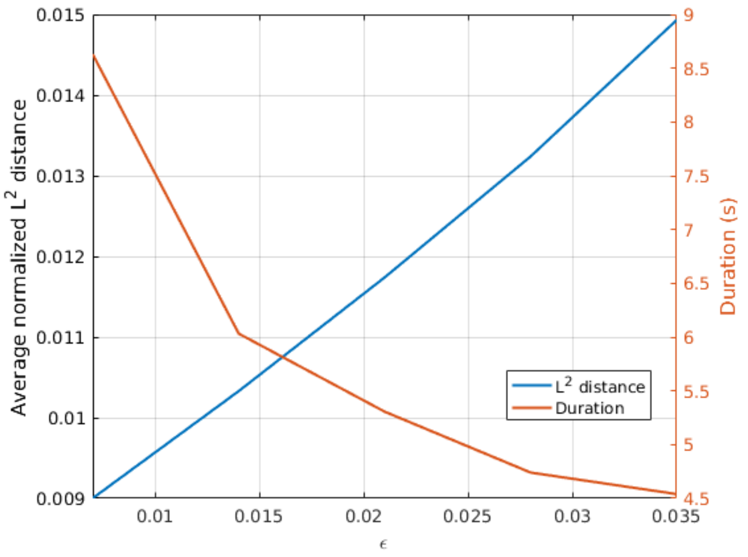


Figure 17: The average normalized L_2 distance between the transformed and original image and the computation time with respect to ϵ when FGS is used for creating the adversarial example in the algorithm.

	Manifool			Iterative Projection		
Transform.	Fooling Percentage	Avg. Norm. Direct Distance	Avg. Duration	Fooling Percentage	Avg. Norm. Direct Distance	Avg. Duration
Translation	60.7%	1.633	337 ms	99.9%	2.049	762 ms
Rotation and Translation	92.6%	1.651	400 ms	82.2%	1.632	888 ms
Similarity	96%	1.536	218 ms	83%	1.669	785 ms

Table 3: Results for comparison of Manifool and Iterative Projection Algorithm. The results for Iterative Projection is found using Deepfool with $\alpha = 1$.

from MNIST dataset, which is used alongside with the CNN that was explained in Section 5.4.2.

It can be seen that the invariance scores are generally higher for the iterative projection algorithm. This result is expected as it does not actively try to minimize the geodesic distance, but in a way minimizes the Euclidean distance to the fooled transformed image instead. However, it is more successful in finding a fooling transformation for translations compared to Manifool as it does not have the local minima problem Manifool has. This is because iterative projection does not try to minimize $|f_j(\boldsymbol{x}_t) - f_{l_x}(\boldsymbol{x}_t)|$, but only the distance to a single point. On the other had, it can be seen that the success percentage is lower on similarity and rotation+translation groups. This is caused by the minimization step of the iterations, in particular because of the gradient descent. In these transformations, change induced by one of the parameters (rotation or scaling) is higher than the others. Thus, one of the rows of $\boldsymbol{J}_{x_{\tau(\theta)}}$ has a greater norm then the rest and on average, the gradient $(\boldsymbol{J}_{x_{\tau(\theta)}})^T(\boldsymbol{x}_{\tau(\theta)} - \boldsymbol{x})$ has a greater value for this parameter. This reflects on

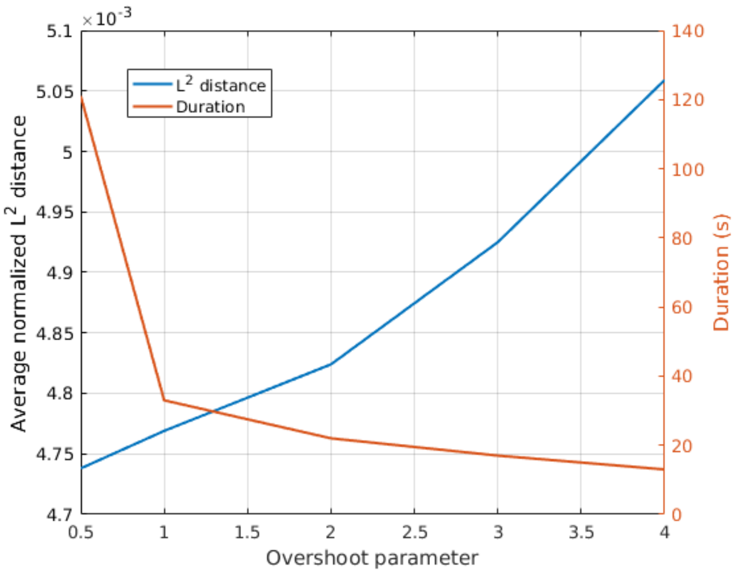


Figure 18: The average normalized L_2 distance between the transformed and original image and the computation time with respect to α when $(1+\alpha)\boldsymbol{r}_{df}$ is used for generating the adversarial example in the algorithm.

the update step on (6.5), and causes the greatest change to happen in this parameter. In the end, the output transformation will be dominated by this parameter, and sometimes it even stops the algorithm from converging. Thus, gradient descent should only be used for the minimization only if the transformation parameters change the image equivalently. For translation group, the changes are generally equivalent and thus the algorithm converges most of the time.

6.3.3 Invariance score for general transformations

In the last experiment, ImageNet dataset is used to measure the invariance score of four pre-trained networks (ResNet-16, ResNet-34, ResNet-50 and ResNet-101) to the set of 'general transformations' that was defined in (6.6). Again, similar to Section 6.3.1, the L^2 metric from (2.19) is used for computing the invariance score and the distance is normalized as well, similar to (2.23) . To reduce the computation time, we use FGS in step 3 of algorithm 3.

The results of the experiment are seen on Table 4. It shows that the invariance of the networks to this set of transformations also increase with the depth of the network like the other lower dimensional transformation sets. We also note that, the invariance scores tend to follow the Deepfool scores of the network, even while FGS is used for generating the transformations. As these transformations are generated using adversarial methods, the invariance score might be closely related to the adversarial perturbation invariance of the network. Thus, a further study on this can be made by measuring the change in the invariance score of a network fine-tuned using Deepfool examples, as this has been shown to increase the invariance of networks to adversarial perturbations [32].

6.4 Conclusion

In this section, the iterative projection algorithm is introduced, which can compute invariance scores for a wide range of transformations. Its complexity and running time depends on the methods that are used for the finding adversarial perturbation and finding the closest point, however in the inspected cases, it can find a small fooling transformation in less than a minute, even for a manifold with 100352 dimensions(general transformations for ImageNet). However, it has some important drawbacks as well. These can be listed as follows:

- Finding the minimizing transformation requires a minimization scheme to be readily available for the transformation set. Thus, the algorithm is not applicable for all sets of transformations.

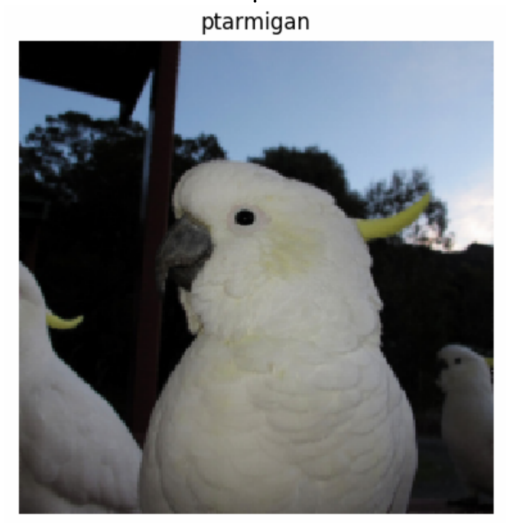
Network	Avg. Normalized L_2 Distance for Output Transform.	Avg. Normalized Deepfool Output Norm
ResNet-18	8.855×10^{-3}	1.594×10^{-3}
ResNet-34	1.001×10^{-2}	1.894×10^{-3}
ResNet-50	1.033×10^{-2}	1.829×10^{-3}
ResNet-101	1.073×10^{-2}	1.963×10^{-3}

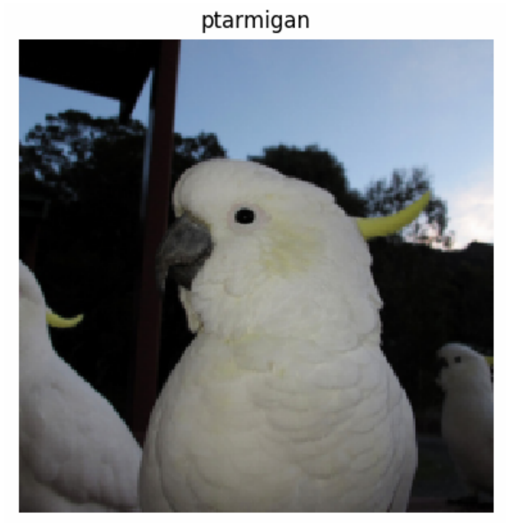
Table 4: Invariance scores of ResNet networks to general transformation set and Deepfool perturbation. The general transformation examples are found using FGS with $\epsilon = 0.014$.

- If a parameter has induces greater change on the image compared to the rest, it might cause the minimization step to focus mostly on this parameter and in some cases stop the algorithm from converging.
- The regularization parameter heavily depends on the image. A bad choice of η can prevent the convergence of the algorithm.

Nevertheless, the algorithm is a good choice for quickly finding perturbing examples in a set of transformations, even if the number of parameters of these transformations is high.

(a) Original image



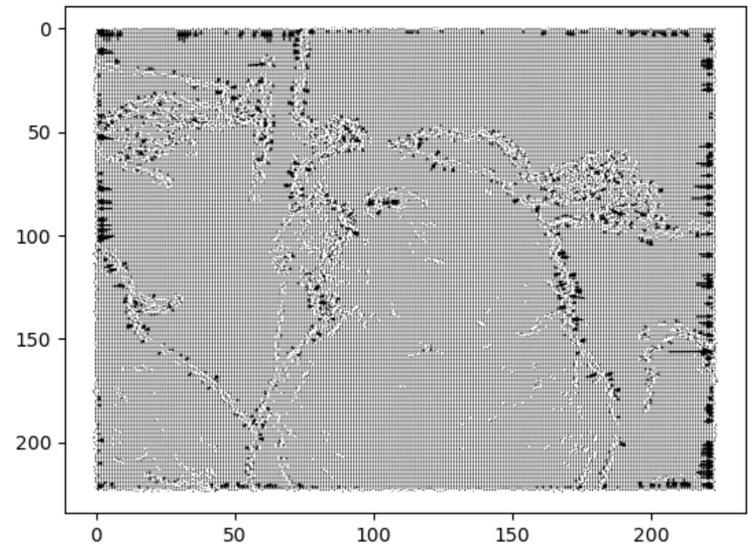


(b) Image generated using FGS (Normalized L^2 metric : 1.57×10^{-2})

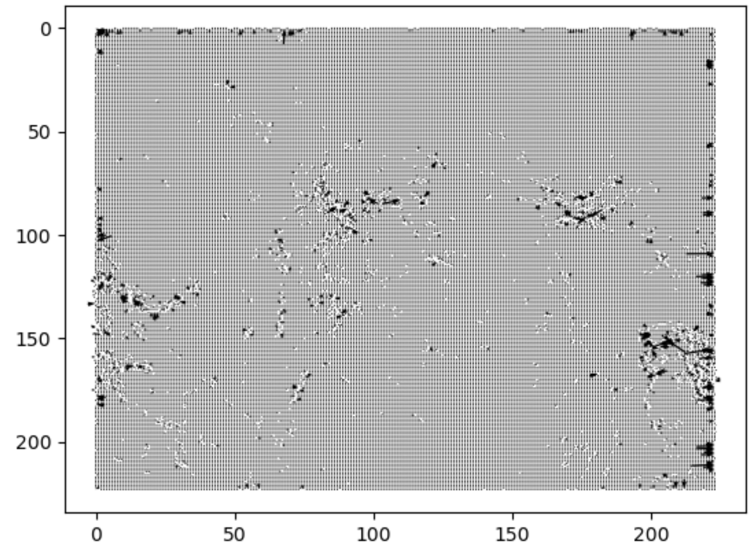
(c) Image generated using Deepfool (Normalized L^2 metric : 6.46×10^{-3})

Figure 19: Examples for fooling 'general transformations' for an image from ImageNet database. In both cases, similar to adversarial perturbations, the change in the image is too low to see the change. However, it is enough to change the output of the network. The movement vectors for these transformations can be seen on Figure 20.

.



(a) Movement parameters of the image generated using FGS



(b) Movement parameters of the image generated using Deepfool

Figure 20: Movement vectors of the transformations in Figure 19. It can be seen that, compared to FGS, using Deepfool generates much more localized vectors.

7 Conclusion and Future Work

In this work, we have introduced two algorithms to find examples in a chosen set of geometric transformations that change the output label of state-of-the-art classifiers. These methods are then used for computing the invariance scores of these classifiers to these chosen transformation sets. First algorithm, Manifool, uses the manifold structure of certain transformation sets along with linearizing the classifier restricted on this manifold to iteratively find the requested example. It is compared with Manitest method to show its validity and it is also shown that it can perform on problems that it cannot feasibly work on. Using Manifool, we have also quantified the relation between transformation invariance and depth of the network. The second algorithm, iterative projection, has less restrictions on the transformation sets it can work on and uses a combination of adversarial perturbations and Euclidean projection to compute the transformations. It has been shown that it can find a fooling examples from a set of transformations with very high dimensionality. Overall, we believe that these two algorithms can be used for analysis of different architectures and can also help to build more robust classifiers.

One of the areas we have not examined is the effect of using the outputs of these algorithms to fine tune the classifiers that they had worked on. The test error of these classifiers as well as the invariance scores should be tested if this fine tuning had made the classifier more robust to the set of transformations and more accurate in general. If this is the case, then the algorithms can easily be used to make better classifiers simply by using them for training.

References

[1] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds.* Princeton University Press, Princeton, N.J. ; Woodstock, 2008. OCLC: ocn174129993.

[2] A. Al-Mohy and N. Higham. A New Scaling and Squaring Algorithm for the Matrix Exponential. *SIAM. J. Matrix Anal. & Appl.*, 31(3):970–989, August 2009.

[3] Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific J. Math.*, 16(1):1–3, 1966.

[4] Vincent Arsigny, Pierre Fillard, Xavier Pennec, and Nicholas Ayache. Geometric Means in a Novel Vector Space Structure on Symmetric Positive-Definite Matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):328–347, January 2007.

[5] Amr Bakry, Mohamed Elhoseiny, Tarek El-Gaaly, and Ahmed Elgammal. Digging Deep into the layers of CNNs: In Search of How CNNs Achieve View Invariance. *arXiv:1508.01983 [cs]*, August 2015.

[6] Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. *Numerical Geometry of Non-Rigid Shapes.* Monographs in Computer Science. Springer New York, New York, NY, 2009.

[7] Davide Chicco, Peter Sadowski, and Pierre Baldi. Deep Autoencoder Neural Networks for Gene Ontology Annotation Predictions. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, BCB '14, pages 533–540, New York, NY, USA, 2014. ACM.

[8] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. pages 191–198. ACM Press, 2016.

[9] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable Convolutional Networks. *arXiv:1703.06211 [cs]*, March 2017.

[10] James J. DiCarlo and David D. Cox. Untangling invariant object recognition. *Trends in Cognitive Sciences*, 11(8):333–341, August 2007.

[11] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[12] Alexey Dosovitskiy, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative Unsupervised Feature Learning with Convolutional Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 766–774. Curran Associates, Inc., 2014.

[13] Alhussein Fawzi and Pascal Frossard. Manitest: Are classifiers really invariant? *arXiv preprint arXiv:1507.06535*, 2015.

[14] Peter Földiák. Learning Invariance from Transformation Sequences. *Neural Computation*, 3(2):194–200, June 1991.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* JHU Press, December 2012.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[17] Ian Goodfellow, Honglak Lee, Quoc V. Le, Andrew Saxe, and Andrew Y. Ng. Measuring invariances in deep networks. In *Advances in Neural Information Processing Systems*, pages 646–654, 2009.

[18] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[19] Brian C. Hall. *Lie Groups, Lie Algebras, and Representations*, volume 222 of *Graduate Texts in Mathematics.* Springer International Publishing, Cham, 2015.

[20] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision.* 2004. OCLC: 171123855.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015.

[22] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and others. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pages 2017–2025, 2015.

[23] Angjoo Kanazawa, Abhishek Sharma, and David Jacobs. Locally Scale-Invariant Convolutional Neural Networks. *arXiv:1412.5104 [cs]*, December 2014.

[24] E. Kokiopoulou and P. Frossard. Minimum Distance between Pattern Transformation Manifolds: Algorithm and Applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(7):1225–1238, July 2009.

[25] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10(Jan):1–40, 2009.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[27] Yann LeCun and Yoshua Bengio. Convolutional Neural Networks for Images, Speech, and Time Series. In Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.

[28] John M. Lee. *Introduction to Smooth Manifolds*, volume 218 of *Graduate Texts in Mathematics.* Springer New York, New York, NY, 2012.

[29] Karel Lenc and Andrea Vedaldi. Understanding image representations by measuring their equivariance and equivalence. *arXiv:1411.5908 [cs]*, November 2014.

[30] Seonwoo Min, Byunghan Lee, and Sungroh Yoon. Deep Learning in Bioinformatics. *arXiv:1603.06430 [cs, q-bio]*, March 2016.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[32] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[33] Adam Paszke, Sam Gross, and Soumith Chintala. PyTorch. `http://pytorch.org/`.

[34] Gabriel Peyré, Mickaël Péchaud, Renaud Keriven, and Laurent D. Cohen. Geodesic Methods in Computer Vision and Graphics. *Foundations and Trends in Computer Graphics and Vision*, 5(3-4):197–397, 2010.

[35] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int J Comput Vis*, 115(3):211–252, December 2015.

[37] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. *Artificial Neural Networks–ICANN 2010*, pages 92–101, 2010.

[38] James A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[39] James A. Sethian and Alexander Vladimirsky. Fast methods for the Eikonal and related Hamilton–Jacobi equations on unstructured meshes. *Proceedings of the National Academy of Sciences*, 97(11):5699–5703, 2000.

[40] Xu Shen, Xinmei Tian, Anfeng He, Shaoyan Sun, and Dacheng Tao. Transform-Invariant Convolutional Neural Networks for Image Classification and Search. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1345–1354, New York, NY, USA, 2016. ACM.

[41] Donald Shepard. A Two-dimensional Interpolation Function for Irregularly-spaced Data. In *Proceedings of the 1968 23rd ACM National Conference*, ACM '68, pages 517–524, New York, NY, USA, 1968. ACM.

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[43] Stefano Soatto and Alessandro Chiuso. Visual Representations: Defining Properties and Deep Approximations. *arXiv:1411.7676 [cs]*, February 2016.

[44] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv:1312.6199 [cs]*, February 2014.

[45] Loring W. Tu. *Differential Geometry*, volume 275 of *Graduate Texts in Mathematics*. Springer International Publishing, Cham, 2017.

[46] Michael B. Wakin, David L. Donoho, Hyeokho Choi, and Richard G. Baraniuk. The multiscale structure of non-differentiable image manifolds. In *Optics & Photonics 2005*, pages 59141B–59141B. International Society for Optics and Photonics, 2005.

[47] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [cs]*, September 2016.