

Neighborhoods of Adversarial Examples

Brian Bell : University of Arizona

June 23, 2023

Contents

1	Introduction	3
I	Persistence	20
2	Introduction	20
3	Motivation and related work	21
4	Stability and Persistence	23
5	Experiments	27
6	Conclusion	33
7	Bracketing Algorithm	34
8	Convolutional neural networks used	34
9	Additional Figures	35
10	Concentration of measures	40
II	Model Geometry	40

11 Neural Networks are (Mostly) Kernel Machines	41
12 Introduction	41
13 Related Work	42
14 Discrete Path Kernels	43
15 Experimental Results	45
16 Discussion	48
17 Acknowledgements	51
18 Neural Networks are Gaussian Processes	61
19 prove path kernel result in context of differential flow of gradients on neural network. using forward euler approx of grad flow.	62
20 ** look for sample in weight space and look for gradients	62
21 training gradients are smooth	62
22 robust network types : regularized, michael's pca, Soft Nearest Neighbor Loss (SNNL) and adversarially trained.	62
23 high dimensional arcs are very similar to chords	62
24 linear interpolated model parameters from random to trained state yield robust models	62
25 define robustness in terms of skew versus orthogonal	63
26 define robustness in terms of attack perturbation magnitude	63
III Decision Boundaries	63
27 decision boundary definitions	63

28 The Argmax Function	63
29 Defining Decision Boundaries	64
30 Properties of Decision Boundaries	66
31 exploring boundary curvature with Random Walks	75
32 Re-Examining Persistence	83
33 define orthants	84
34 skewness	84
35 sampling decision boundaries and analyzing dimensionality with PCA	84
36 neural network attack gradients versus decision boundaries	84
37 skewed orthant recreates persistence picture (?).	84
38 measuring skewness of ANNs	84
39 Relate skewness with dimpled manifold and features not bugs papers	84

1 Introduction

Artificial Neural Networks and other optimization-based general function approximation models are the core of modern machine learning [Prakash et al.(2018)Prakash, Moran, Garber, DiLillo, and Storer]. These models have dominated competitions in image processing, optical character recognition, object detection, video classification, natural language processing, and many other fields [Schmidhuber(2015)]. All such modern models are trained via gradient-based optimization, e.g. Stochastic Gradient Descent (SGD) with gradients computed via back propagation. [Goodfellow et al.(2013)Goodfellow, Bulatov, Ibarz, Arnoud] Although the performance of these models is practically miraculous within the training and testing context for which they are designed, they have a few intriguing properties. It was discovered in 2013 [Szegedy et al.(2013)Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus] that images can be generated which apparently trick such models in a classification context in

difficult-to-control ways [Khoury and Hadfield-Menell(2018)]. The intent of this research is to investigate these *adversarial examples* in a mathematical context and use them to study pertinent properties of the learning models from which they arise.

1.1 Artificial Neural Networks (ANNs)

The history of Neural Networks begins in the field of Theoretical Neuropsychology with a much-cited paper by McCulloch and Pitts in which they describe the mechanics of cognition in the context of computation [McCulloch and Pitts(1943)]. This initial framework for computational cognition did not include a notion for learning, but the following decade brought the concept of learning (as optimization) and many simple NNs (linear regression models applied to computational cognition). The perceptron, the most granular element of a neural network, was proposed in another much-cited paper by Rosenblatt in 1958 [Rosenblatt(1958)], and multilevel (deep) networks were proposed by 1965 in a paper by Ivakhnenko and Lapa [Ivakhnenko and Lapa(1965)].

By the 1960s, these neural network models became disassociated from the cutting-edge of cognitive science, and interest had shifted to their application in modeling and industrial computation. The hardware limitations of the time served as a significant barrier to wider application and the concept of the "neural network" was sometimes treated as a solution looking for a problem. Compounding these limitations was a significant roadblock published by Minsky and Papert in 1969: A proof that basic perceptrons could not encode exclusive-or [Minsky and Papert(1969)]. As a result, interest in developing neural network theory waned. The next necessary step in the development of modern neural network models was an advance that would allow them to be trained efficiently with computing power available. Learning methods required a gradient, and the technique necessary for computing gradients of large-scale multi-parameter models was apparently proposed in a 1970 in a Finnish masters thesis [Linnainmaa(1970)]. Techniques from control theory were applied to develop a means of propagating error backward through models which could be described as directed graphs. The idea was applied to neural networks by a Harvard student Paul Werbos[Werbos(1974)] and refined in later publications.

The final essential puzzle piece for neural network models was to take advantage of their layered structure, which would allow backpropagation computations at a given layer to be done in parallel. This key insight, indeed the core of much of modern computing, was a description

of parallel and distributed processing in the context of cognition by Rumelhard and McClelland in 1986 [McClelland et al.(1986)McClelland, Rumelhart, Group, et al.] with an astonishing 22,453 citations (a number that grows nearly every day). With these pieces in place, the world was ready for someone to finally apply neural network models to a relevant problem. In 1989, Yann LeCun and a group at Bell Labs managed to do just that. LeCun refined backpropagation into the form it is used today [LeCun et al.(1989)LeCun, Boser, Denker, Henderson, Howard, Hubbard, and Jackel], invented Convolutional Neural Networks 1.2.1 [LeCun et al.(1995)LeCun, Bengio, et al.], and by 1998, he had worked with his team to implement what has become the industry standard for banks to recognize hand-written numbers on checks [LeCun et al.(1998)LeCun, Bottou, Bengio, Haffner, et al.].

Starting in the 2000s, neural networks have blown up in scale and application, so it's harder to keep track of the discrete historical developments. Most of the progress in terms of performance and application has come from contests (e.g: <http://image-net.org/challenges/LSVRC/>) in which a bounty (or publicity) is offered and labs around the world compete to build a network which solves the competition's problem. Schmidhuber's group [Schmidhuber(2015)], and a similar group at Google have dominated many of these competitions. The cutting-edge today is represented by networks like Inception v4 designed by Google for image classification which contains approximately 43 million parameters [Szegedy et al.(2013)Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus]. Early versions of this network took 1-2 million dollars worth of compute-time to train. ANNs now appear in nearly every industry from devices which use ANNs to intelligently adapt their performance, to the sciences which rely on ANNs to eliminate tedious sorting and identification of data that previously had to be relegated to humans.

1.2 Structure

In this subsection we give a mathematical description of artificial neural networks.

A *neuron* is a nonlinear operator that takes input in \mathbb{R}^n to \mathbb{R} , historically designed to emulate the activation characteristics of an organic neuron. A collection of neurons that are connected via a (usually directed) graph structure are known as an *Artificial Neural Network (ANN)*.

The fundamental building blocks of most ANNs are artificial neurons which we will refer to as *perceptrons*.

Definition 1.1. A Perceptron is a function $P_{\vec{w}} : \mathbb{R}^n \rightarrow \mathbb{R}$ which has weights $\vec{w} \in \mathbb{R}^n$ corresponding with each element of an input vector $\vec{x} \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$:

$$P_{\vec{w}}(\vec{x}) = f((\langle \vec{w}, \vec{x} \rangle + b))$$

$$P_{\vec{w}}(\vec{x}) = f \left(b + \sum_{i=1}^n w_i x_i \right)$$

where $f : \mathbb{R} \rightarrow \mathbb{R}$ is continuous. The function f is called the *activation function* for P .

The only nonlinearity in P_w is contained in f . If f is chosen to be linear, then P will be a linear operator. Although this has the advantage of simplicity, linear operators do not perform well on nonlinear problems like classification. For this reason, activation functions are generally chosen to be nonlinear. Historically, heaviside functions were used for activation, later replaced with sigmoids [Malik and Perona(1990)] for their smoothness, switching structure, and convenient compactification of the output from each perceptron. It was recently discovered that a simpler nonlinear function, the *Rectified Linear Unit (ReLU)* works as well or better in most neural-network-type applications [Glorot et al.(2011)Glorot, Bordes, and Bengio] and additionally training algorithms on ReLU activated networks converge faster [Nair and Hinton(2010)].

Definition 1.2. The Rectified Linear Unit (ReLU) function is

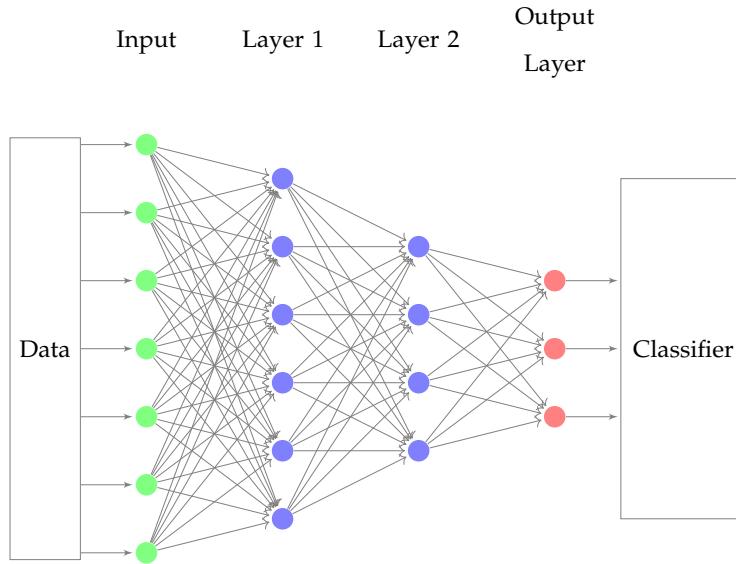
$$\text{ReLU}(x) = \begin{cases} 0, & x \leq 0; \\ x, & x > 0, \end{cases}$$

The single nonlinearity of this activation function at $x = 0$ is sufficient to guarantee existence of ϵ approximation of smooth functions by an ANN composed of sufficiently numerous perceptrons connected by ReLU [Petersen and Voigtlaender(2018)]. In addition, ReLU is convex, which enables efficient numerical approximation of smooth functions in shallow networks [Li and Yuan(2017)].

In general ANNs must not be cyclic and, for convenience, are often arranged into independent layers. An early roadblock for neural networks was a proof by Minsky [Minsky and Papert(1969)] that single layers of perceptrons could not encode exclusive-or. Depth, the number of layers in a neural network, is a key factor in its ability to approximate complicated functions including exclusive-or [Kak(1993)]. For this reason, modern ANNs are usually composed of many layers

(3-100). The most common instance of a neural network model is a fully connected *feed forward (FF)* configuration. In this configuration data enters as an input layer which is fed into each of the nodes in the first layer of neurons. Output of the first layer is fed into each of the nodes in the second layer, and so on until the output of the final layer is fed into an output filter which generates the final result of the neural network.

In this example of a FF network, an input vector in \mathbb{R}^7 is mapped to a an output in \mathbb{R}^3 which is fed into a classifier. Each blue circle represents a perceptron with the ReLU activation function.



The output of this ANN is fed into a classifier. To complete this example, we can define the most common classifier, Softmax:

Definition 1.3. *Softmax (or the normalized exponential) is the function given by*

$$s : \mathbb{R}^n \rightarrow [0, 1]^n$$

$$s_j(\vec{x}) = \frac{e^{x_j}}{\sum_{k=1}^n e^{x_k}}$$

Definition 1.4. *We can define a classifier which picks the class corresponding with the largest output element from Softmax:*

$$(Output\ Classification) c_s(\vec{x}) = \text{argmax}_i s_i(\vec{x})$$

During training, the output $y \in \mathbb{R}^n$ from a network can thus be compressed using softmax

into $[0, 1]^n$ as a surrogate for probability for each possible class or directly into the classes which we can represent as the simplex for the vertices of $[0, 1]^n$.

[Bishop(2006)]

1.2.1 Convolutional Neural Networks (CNNs)

Another common type of neural network which is a component in many modern applications including one in the experiments to follow are Convolutional Neural Networks (CNNs). CNNs are fundamentally composed of perceptrons, but each layer is not fully connected to the next. Instead, layers are arranged spatially and overlapping groups of perceptrons are independently connected to the nodes of the next layer, usually with a nonlinear filter that computes the maximum of all of the incoming nodes to a new node. This structure has been shown to be very effective on problems with spatial information [LeCun et al.(1995)LeCun, Bengio, et al.].

1.3 Training ANNs

Neural networks consist of a very large number of perceptrons with many parameters. Directly solving the system implied by these parameters and the empirical risk minimization problem defined below would be difficult, so we must use a modular approach which takes advantage of the simple and regular structure of ANNs.

A breakthrough came with the application of techniques derived from control theory to ANNs in the late 1980s [Rumelhart et al.(1986)Rumelhart, Hinton, and Williams], dubbed back-propagation. This technique was refined into its modern form in the thesis and continuing work of Yann LeCun [LeCun et al.(1988)LeCun, Touresky, Hinton, and Sejnowski]. In this method, error is propagated backward taking advantage of the directed structure of the network to compute a gradient for each parameter defining it. Because modern ANNs are usually separated into discrete layers, gradients can be computed in parallel for all perceptrons at the same depth of the network [Bishop(2006)]. Leveraging modern GPUs and parallel computing technologies, these gradients can be computed very quickly. There are a number of important considerations in training. We discuss a few in the following subsections.

1.3.1 Selection of the Training Set

The first step in training an ANN is the selection of a training set. ANNs fundamentally are universal function approximators: Given a set of input data and corresponding output data, they approximate a mapping from one to the other. Performance is dependent on how well the phenomenon we hope to model is represented by the training data. The training data must consist of a set of inputs (e.g., images) and a set of outputs (e.g., labels) which contain sufficient examples to characterize the intended model. In a way, this is how we pose a question to the neural network. One must always ask whether the question we wish to pose is well-expressed by the training data we have available.

The most important attributes of a training dataset are the number of samples it contains and its density near where the model will be making predictions. According to conventional wisdom, training a neural network with K parameters will be very challenging if there are fewer than K training samples available. The modular structure of ANNs can be combined with regularization of the weights to overcome these limitations [Liu and Deng(2015)]. In general, we will denote a training set by (X, Y) where X is an indexed set of inputs and Y is a corresponding indexed set of labels.

1.3.2 Selecting a Loss Function

Once we have selected a set of training data (both inputs and outputs), we must decide how we will evaluate the match between the ANNs output and the defined outputs from the training dataset – we will quantify the deviation of the ANN compared with the given correspondence as a Loss. In general *loss functions* are nonzero functions which compare an output y against a ground-truth \hat{y} . Generally they have the property that an ideal outcome would have a loss of 0.

One commonly used loss function for classification is known as Cross-Entropy Loss:

Definition 1.5. *The Cross-Entropy Loss comparing two possible outputs is $L(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i$.*

Other commonly used loss functions include L^1 loss (also referred to as Mean Absolute Error (MAE)), L^2 loss (often referred to as Mean-Squared-Error (MSE)), and Hinge Loss (also known as SVM loss).

To set up the optimization, the loss for each training example must be aggregated. Generally,

ANN training is conducted via Empirical Risk Minimization where Empirical Risk is defined for a given loss function L as follows:

Definition 1.6. *Given a loss function L , the Empirical Risk over a training dataset (X, Y) of size N is*

$$R_{\text{emp}}(P_{\vec{w}}(x)) = \frac{1}{N} \sum_{(x,y) \in (X,Y)} L(P_{\vec{w}}(x)), y).$$

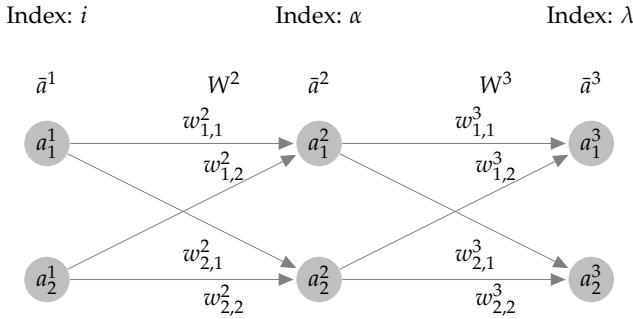
We seek parameters \vec{w} which will minimize $R_{\text{emp}}(P_{\vec{w}}(x))$. This will be done with gradient-based optimization.

1.3.3 Computation of Gradient via Backpropagation

Since it is relevant to the optimization being performed, we will briefly discuss the computation of gradients via backpropagation. For this discussion, we will introduce a small subset of a neural network in detail. In general, terms will be indexed as follows:

$$x_{[\text{node in layer}], [\text{node in previous layer}]}^{[\text{layer}]}$$

When the second subscript is omitted, the subscript will only index the node in the current layer to which this element belongs.



In this diagram, the W^l are matrices composed of the weights indexed as above. Given an activation function for layer n A^n and its element-wise application to a vector \bar{A}^n , we can now write the output \bar{a}_n for any layer of an arbitrary ANN in two ways [Krause(2020)]. Recursively, we can define

$$a_\lambda^n = A^n \left(\sum_\alpha w_{\alpha,\lambda}^n a_\alpha^{n-1} \right) \quad (1)$$

We can also write the matrix form of this recursion for every node in the layer:

$$\bar{a}^n = \bar{A}^n(W^n(\bar{a}^{n-1})) \quad (2)$$

The matrix form makes it easier to write out a closed form for the output of the neural network.

$$\bar{a}^n = \bar{A}^n(W^n(\bar{A}^{n-1}(W^{n-1}(\dots(\bar{A}^2(W^2\bar{a}^1)\dots)))) \quad (3)$$

Now, given a loss function $L = \sum_i \ell_i(a_i^n)$ where each ℓ_i is a loss function on the i^{th} element of the output, we wish to compute the derivatives $\frac{\partial L}{\partial w_{ij}^l}$ for every l, i , and j which compose the gradient ∇L . Using the diagram above, we can compute this directly for each weight using chain rule:

$$\frac{\partial L}{\partial w_{\lambda,\alpha}^3} = \frac{\partial L}{\partial a_\lambda^3} \frac{\partial a_\lambda^3}{\partial w_{\lambda,\alpha}^3} = \sum_{\lambda=1}^n \ell'_\lambda(a_\lambda^3) A'^3 \left(\sum_{\alpha=1}^n w_{\alpha,\lambda}^3 a_\alpha^2 \right) a_\alpha^2$$

Many of the terms of this gradient (e.g. the activations a_i^n and the sums $\sum_i w_{i,j}^n a_i$) are computed during forward propagation when using the network to generate output. We will store such values during the forward pass and use a backward pass to fill in the rest of the gradient. Furthermore, notice that ℓ'_λ and A'^m are well understood functions whose derivatives can be computed analytically almost everywhere. We can see that all of the partials will be of the form $\frac{\partial L}{\partial w_{n,i}^l} = \delta_n^l a_i^l$ where δ_n^l will contain terms which are either pre-computed or can be computed analytically. Conveniently, we can define this error signal recursively:

$$\delta_n^l = A'^l(a_n^l) \sum_{i=1}^n w_{i,n}^{l+1} \delta_i^{l+1}$$

In matrix form, we have

$$\delta^l = \bar{A}'^l(W^l \bar{a}^l) \odot ((W^{l+1})^T \delta^{l+1})$$

Where \odot signifies element-wise multiplication.

Then we can compute the gradient with respect to each layer's matrix W^l as an outer product:

$$\nabla_{W^l} L = \bar{\delta}^l \bar{a}^{(l-1)T}$$

Since this recursion for layer n only requires information from layer $n + 1$, this allows us to propagate the error signals that we compute backwards through the network.

1.3.4 Optimization of Weights

Given a set of training input data and a method for computing gradients, our ultimate goal is to iteratively run our training-data through the network, updating weights gradually according to the gradients computed by backpropagation. In general, we start with some default arrangement of the weights and choose a step size η for gradient descent. Then for each weight, in each iteration of the learning algorithm, we apply a correction so that

$$w'_{i',j',k'} = w_{i',j',k'} - \eta \frac{\partial E(Y, \hat{Y})}{\partial w_{i',j',k'}}$$

In this case, the step size (learning rate) η is fixed throughout training. Numerical computation of the gradient requires first evaluating the network forward by computing the output for a given input. The value of every node in the network is saved and these values are used to weight the error as it is propagated backward through the network. Once the gradient is computed, the weights are adjusted according to the step defined above. This process is repeated until convergence is attained to within a tolerance. It should be clear from the number of terms in this calculation that the initial guess and step size can have significant effect on the eventual trained weights. Due to lack of a guarantee for general convexity, poor guesses for such a large number of parameters can lead to gradients blowing up or down [Bishop(2006)]. . Due to nonlinearity and the plenitude of local minima in the loss function, classic gradient descent usually does not perform well during ANN training.

By far the most common technique for training the weights of neural networks adds noise in the form of random re-orderings of the training data to the general optimization process and is known as stochastic gradient descent.

Definition 1.7. *Stochastic Gradient Descent (SGD)*

Given an ANN $N : \mathbb{R}^n \rightarrow C$, an initial set of weights for this network \vec{w}_0 (usually a small random perturbation from 0), a set of training data X with labels Y , and a learning rate η , the algorithm is as follows:

Batch Stochastic Gradient Descent

```
w = w0
while E(Ŷ, P_w(X)) (cumulative loss) is still improving do      ▷ (the stopping condition may
    require that the weight change by less than ε for some number of iterations or could be a fixed
    number of steps)
    Randomly shuffle (X, Y)
    Draw a small batch (X̂, Ŷ) ⊂ (X, Y)
    w ← w - η (Σ_{(x,y) ∈ (X̂, Ŷ)} ∇L(P_w(x̂), ŷ))
end while
```

Stochastic gradient descent achieves a smoothing effect on the gradient optimization by only sampling a subset of the training data for each iteration. Miraculously, this smoothing effect not only often achieves faster convergence, the result also generalizes better than solutions using deterministic gradient methods [Hardt et al.(2015)[Hardt, Recht, and Singer](#)]. It is for this reason that SGD has been adopted as the de facto standard among ANN training applications.

1.4 Adversarial Attacks

In general *Adversarial Attacks* against models are small perturbations from natural data which significantly perturb model output. Szegedy et al. [[Szegedy et al.\(2013\)](#)[Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus](#)] realized that the same computational tools used to train ANNs could be used to generate attacks that would confuse them. Their approach was to define a loss function relating the output of the ANN for a given initial image to a target adversarial output plus the L^2 -norm of the input and use backpropagation to compute gradients – not on the weights of the neural network, but on just the input layer to the network. The solution to this optimization problem, efficiently approximated by a gradient-based optimizer, would be a slightly perturbed natural input with a highly perturbed output. Their experimental results are striking:

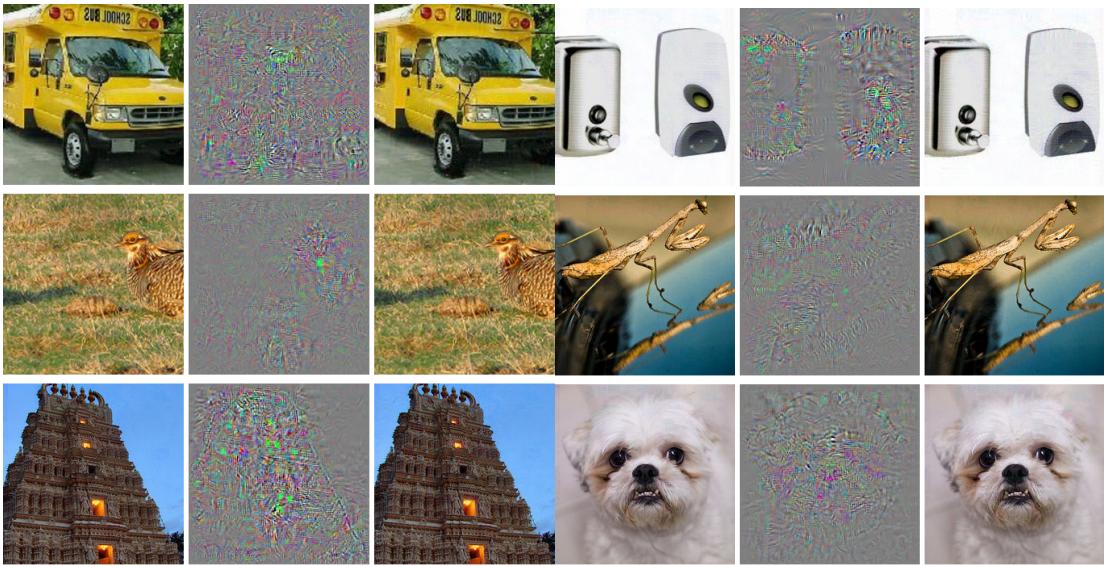


Figure 1: Natural Images are in columns 1 and 4, Adversarial images are in columns 3 and 6, and the difference between them (magnified by a factor of 10) is in columns 2 and 5. All images in columns 3 and 6 are classified by AlexNet as "Ostrich" [Szegedy et al.(2013)]

1.4.1 Data Sets

The Dataset used above is known as ImageNet – a large set of labeled images varying in size originally compiled for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). This dataset and its many subsets has become a standard for image classification and feature identification experiments. In the experiments that follow, ImageNet will be featured alongside the Modified National Institute of Standards and Technology (MNIST) dataset which is a database of hand written digits often used to develop image processing and character recognition systems. This dataset is much lower resolution than ImageNet and is therefore experiments run much more quickly on it and require less complex input/output.

1.4.2 L-BFGS minimizing distortion

Szegedy et al. took advantage of the tools they had on hand for training neural networks to set up a box-constrained optimization problem whose approximated solution generates these targeted misclassifications.

Let $f : \mathbb{R}^m \rightarrow \{1, \dots, k\}$ be a classifier and assume f has an associated continuous loss function

denoted by $\text{loss}_f : \mathbb{R}^m \times \{1, \dots, k\} \rightarrow \mathbb{R}^+$ and l a target adversarial .

Minimize $\|r\|_2$ subject to:

1. $f(x + r) = l$

2. $x + r \in [0, 1]^m$

The solution is approximated with L-BFGS (see Appendix ??) as implemented in Pytorch or Keras. This technique yields examples that are close to their original counterparts in the L^2 sense.

L-BFGS: Mnist The following examples are prepared by implementing the above technique via pytorch on images from the Mnist dataset with FC200-200-10, a neural network with 2 hidden layers with 200 nodes each:

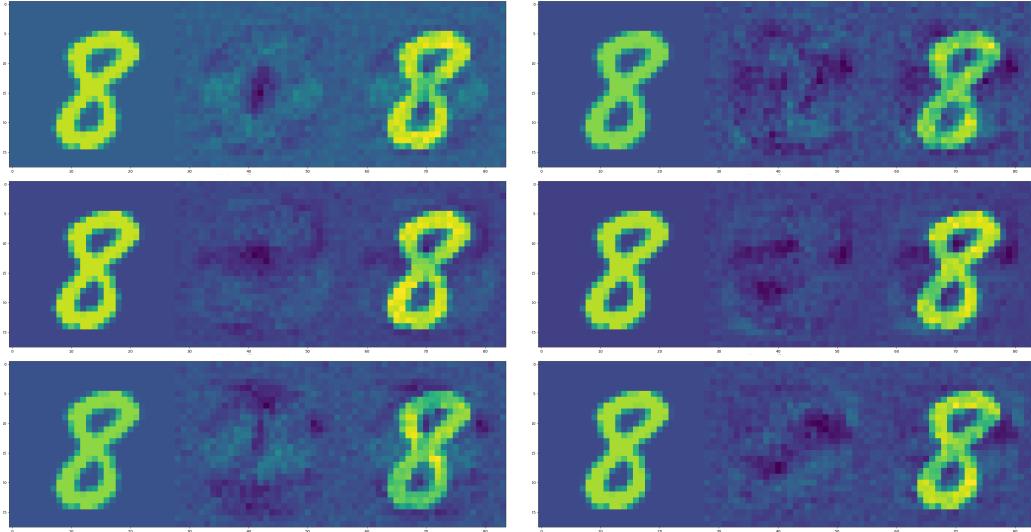


Figure 2: Original images on the left, Perturbation is in the middle, Adversarial Image (total of Original with Perturbation) is on the right. Column 1 shows an original 8 being perturbed to adversarial classes 0, 2, and 4. Column 2 shows adversarial classes 1, 3, and 5

Borrowing a metric from Szegedy et al to compare the magnitude of these distortions, we will define

Definition 1.8. *Distortion is the L^2 norm of the difference between an original image and a perturbed image,*

divided by the square root of the number of pixels in the image:

$$\sqrt{\frac{\sum_i (\hat{x}_i - x_i)^2}{n}}$$

Distortion is L^2 magnitude normalized by the square-root of the number of dimensions so that values can be compared for modeling problems with differing numbers of dimensions.

The 900 examples generated for the network above had an average distortion of 0.089 with the following distribution of distortions, given in figure 3.

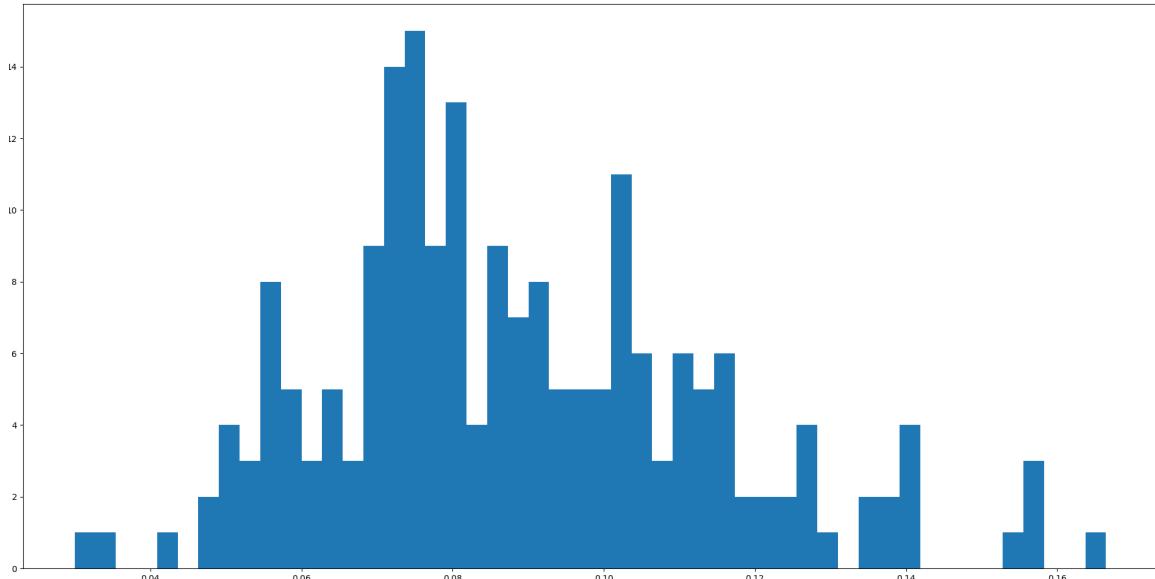


Figure 3: A histogram of the distortion measured for each of 900 adversarial examples generated using L-BFGS against the FC-200-200-10 network on Mnist. Mean distortion is 0.089.

L-BFGS: ImageNet We also tried to replicate [Szegedy et al.(2013)] results on ImageNet. Attacking VGG16, a well known model from the ILSVRC-2014 competition [Simonyan and Zisserman(2014)], on ImageNet images with the same technique generates the examples in figure 4:



Figure 4: Original images on the left, Perturbation (magnified by a factor of 100) by is in the middle, Adversarial Image (total of Original with Perturbation) is on the right.

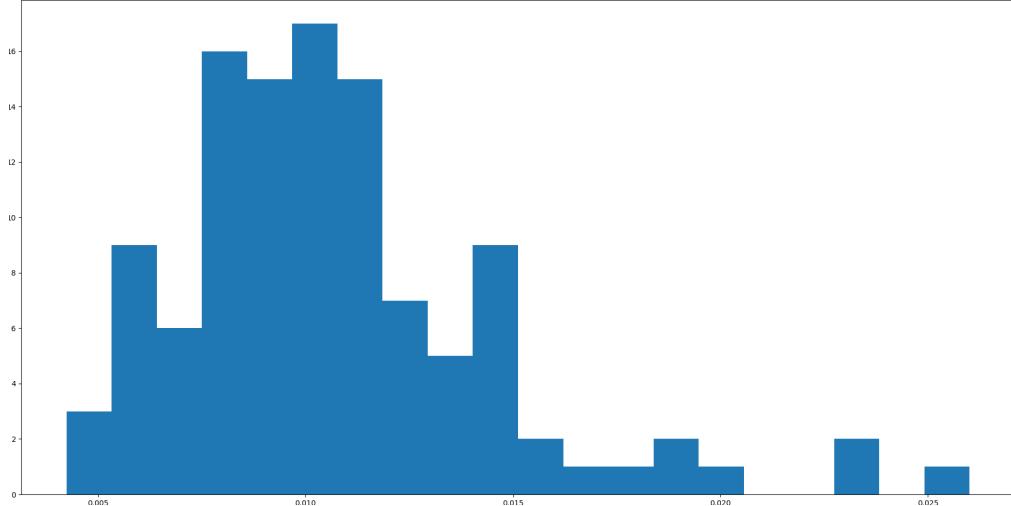


Figure 5: A histogram of the distortion measured for each of 112 adversarial examples generated using L-BFGS against the VGG16 network on ImageNet images with mean distortion 0.0107

Fast Gradient Sign Method (FGSM) We also implemented a single step attack process uses the gradient of the loss function L with respect to the image to find the adversarial perturbation [Goodfellow et al.(2014)[Goodfellow, Shlens, and Szegedy](#)]. for given ϵ , the modified image \hat{x} is computed as

$$\hat{x} = x + \epsilon \text{sign}(\nabla L(P_w(x), x)) \quad (4)$$

This method is simpler and much faster to compute than the L-BFGS technique described above, but produces adversarial examples less reliably and with generally larger distortion. Perfor-

mance was similar but inferior to the Iterative Gradient Sign Method summarized below.

Iterative Gradient Sign Method (IGSM) In [Kurakin et al.(2016)] an iterative application of FGSM was proposed. After each iteration, the image is clipped to a ϵL_∞ neighborhood of the original. Let $x'_0 = x$, then after m iterations, the adversarial image obtained is:

$$x'_{m+1} = \text{Clip}_{x,\epsilon} \left\{ x'_m + \alpha \times \text{sign}(\nabla \ell(F(x'_m), x'_m)) \right\} \quad (5)$$

This method is faster than L-BFGS and more reliable than FGSM but still produces examples with greater distortion than L-BFGS.

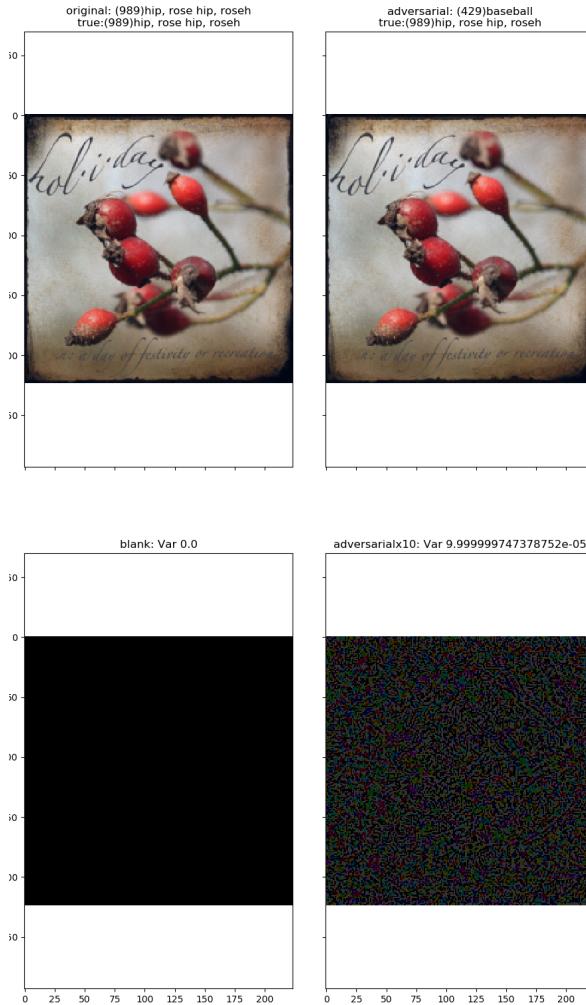


Figure 6: adversarial example generated against VGG16 (ImageNet) with IGSM. Original Image on the left, adversarial image and added noise (ratio of variance adversarial noise/original image: 0.0000999) on the right.

1.4.3 Other Attacks

The following attack techniques are also prevalent in the literature but have not been replicated in these experiments.

Jacobian-based Saliency Map Attack (JSMA) Another attack noted by [Papernot et al.(2015)[Papernot, McDaniel, Jha, Goodfellow, and Boneva\(2016\)](#)] estimates the *saliency map*, a rating for each of the input features (e.g. each pixel) on how influential it is for causing the model to predict a particular class with respect to the model output [Wiyatno and Xu(2018)]. This attack modifies the pixels that are most salient. This is a targeted

attack, and saliency is designed to find the pixel which increases the classifier's output for the target class while tending to decrease the output for other classes.

Deep Fool (DFool) A technique proposed by [Moosavi-Dezfooli et al.(2015)Moosavi-Dezfooli, Fawzi, and Frossard] to generate an un-targeted iterative attack. This method approximates the classifier as a linear decision boundary and then finds the smallest perturbation needed to cross that boundary. This attack minimizes L_2 norm with respect to the original image.

Carlini & Wagner (C&W) In [Carlini and Wagner(2016)] an adversarial attack is proposed which updates the loss function such that it jointly minimizes L_p and a custom differentiable loss function based on un-normalized outputs of the classifier (*logits*). Let Z_k denote the logits of a model for a given class k , and κ a margin parameter. Then C&W tries to minimize:

$$||x - \hat{x}||_p + c * \max(Z_k(\hat{x}_y) - \max\{Z_k(\hat{x}) : k \neq y\}, -\kappa) \quad (6)$$

Part I

Persistence

2 Introduction

Deep Neural Networks (DNNs) and their variants are core to the success of modern machine learning [Prakash et al.(2018)Prakash, Moran, Garber, DiLillo, and Storer], and have dominated competitions in image processing, optical character recognition, object detection, video classification, natural language processing, and many other fields [Schmidhuber(2015)]. Yet such classifiers are notoriously susceptible to manipulation via adversarial examples [Szegedy et al.(2013)Szegedy, Zaremba, Sutskever, Bruna]. Adversarial examples occur when natural data are close enough to the decision boundary that they can be imperceptibly perturbed to another class. Adversarial examples are not just a peculiarity, but seem to occur for most, if not all, DNN classifiers. For example, [?] used isoperimetric inequalities on high dimensional spheres and hypercubes to conclude that there is a reasonably high probability

that a correctly classified data point has a nearby adversarial example. [?] showed that adversarial examples can arise from features that are good for classification but not robust to perturbation.

There have been many attempts to identify adversarial examples using properties of the decision boundary. [?] found that decision boundaries tend to have highly curved regions, and these regions tend to favor negative curvature, indicating that regions that define classes are highly nonconvex. These were found for a variety of DNNs and classification tasks. A related idea is that adversarial examples often arise within cones, outside of which images are classified in the original class, as observed by [?]. Many theoretical models of adversarial examples, for instance the dimple model developed by [?], have high curvature and/or sharp corners as an essential piece of why adversarial examples can exist very close to natural examples.

Contributions. We propose two concepts, (γ, σ) -stability and γ -persistence. Both are defined with reference to a classifier and a given point (which can be either a natural or adversarial image, for example). The proposed notion of stability is more nuanced than simply measuring distance to the decision boundary, and is also capable of elucidating information about the curvature of the nearby decision boundary. Persistence is a statistic that measures how far away from a point one can go via Gaussian sampling and still consistently find points with the same classification. One advantage of this statistic is that it is easily estimated by sending a Monte Carlo sampling about the point through the classifier.

3 Motivation and related work

Our work is intended to shed light on the existence and prevalence of adversarial examples to DNN classifiers. It is closely related to other attempts to characterize robustness to adversarial perturbations, and here we give a detailed comparison.

Distance-based robustness. A typical approach to robustness of a classifier is to consider distances from the data manifold to the decision boundary. [Khoury and Hadfield-Menell(2018)] define a classifier to be robust if the class of each point in the data manifold is contained in a sufficiently large ball that is entirely contained in the same class. The larger the balls, the more robust the classifier. It is then shown that if training sets are sufficiently dense in relation to the reach of the decision axis, the classifier will be robust in the sense that it classifies nearby points

correctly. In practice, we do not know that the data is so well-positioned, and it is quite possible, especially in high dimensions, that the reach is extremely small, as evidenced by results on the prevalence of adversarial examples, e.g., [?].

[?] investigated robustness in terms of how small perturbations affect the average loss of a classifier. They define standard accuracy of a classifier in terms of how often it classifies correctly, and robust accuracy in terms of how often an adversarially perturbed example classifies correctly. It was shown that sometimes accuracy of a classifier can result in poor robust accuracy. [?] use the expected distance to the nearest different class (when drawing a data point from the data distribution) to capture robustness, and then show that an accurate classifier can result in a small distance to the nearest different class in high dimensions when the data is drawn from concentric spheres.

Adversarial detection via sampling. While adversarial examples often occur, they still may be rare in the sense that most perturbations do not produce adversarial examples. [?] used the observation that adversarial examples are both rare and close to the decision boundary to detect adversarial examples. They take a potential data point and look to see if nearby data points are classified differently than the original data point after only a few iterations of a gradient descent algorithm. If this is true, the data point is likely natural and if not, it is likely adversarial. We will draw on a similar idea by considering nearby points drawn from Gaussian distributions of varying standard deviations in order to better quantify this observation of rarity. In general, the results of [?] indicate that considering samples of nearby points, which approximate the computation of integrals, is likely to be more successful than methods that consider only distance to the decision boundary.

[?] proposed a statistical method to identify adversarial examples from natural data. Their main idea was to consider how the last layer in the neural network (the logit layer) would behave on small perturbations of a natural example. This is then compared to the behavior of a potential adversarial example. If it differs by a predetermined threshold, the example is flagged as adversarial. Successfully flagging adversarial examples in this way works best when adversarial examples tend to perturb toward the original class from which the adversarial example was perturbed. However, this is not always the case. It was shown by [?] that it is possible to produce adversarial examples, for instance using a logit mimicry attack, that instead of perturbing an adversarial example toward

the true class, actually perturb to some other background class. In fact, we will see in Section 5.1.1 that the emergence of a background class, which was observed as well by [?], is quite common.

Whereas [?] consider adding various types of noise to a given point and [?] consider small Gaussian perturbations of x sampled from $N(x, \varepsilon^2 I)$ for small ε , we specifically focus on tuning the standard deviation parameter to determine a statistic describing how a given data point is placed within its class. The γ -persistence then gives a measurement similar to distance to the boundary but that is drawn from sampling instead of distance. The sampling allows for a better description of the local geometry of the class and decision boundary, as we will see in Section 4. Our statistic is based on the fraction of a Gaussian sampling of the neighborhood of a point which receives the same classification; this is different from that of [?], which is the expected difference of the output of the logit layer of the original data point and the output of the logit layer for perturbed data points. Additionally, while their statistics are defined pairwise with reference to pre-chosen original and candidate classes, ours is not.

Summary. In Sections 4 and 5, we will investigate stability of both natural data and adversarial examples by considering sampling from Gaussian distributions centered at a data point with varying standard deviations. Using the standard deviation as a parameter, we are able to derive a statistic for each point that captures how entrenched it is in its class in a way that is less restrictive than the robustness described by [Khoury and Hadfield-Menell(2018)], takes into account the rareness of adversarial examples described by [?], builds on the idea of sampling described by [?] and [?], and represent curvatures in a sense related to [?].

4 Stability and Persistence

In this section we define a notion of stability of classification of a point under a given classification model. In the following, X represents the ambient space the data is drawn from (typically \mathbb{R}^n) even if the data lives on a submanifold of X , and L is a set of labels (often $\{1, \dots, \ell\}$). Note that points $x \in X$ can be natural or adversarial points.

Definition 4.1. Let $\mathcal{C} : X \rightarrow L$ be a classifier, $x \in X$, $\gamma \in (0, 1)$, and $\sigma > 0$. We say x is (γ, σ) -stable with respect to \mathcal{C} if $\mathbb{P}[\mathcal{C}(x') = \mathcal{C}(x)] \geq \gamma$ for $x' \sim \rho = N(x, \sigma^2 I)$; i.e. x' is drawn from a Gaussian with variance σ^2 and mean x .

In the common setting when $X = \mathbb{R}^n$, we have

$$\mathbb{P}[\mathcal{C}(x') = \mathcal{C}(x)] = \int_{\mathbb{R}^n} \mathbb{1}_{\mathcal{C}^{-1}(\mathcal{C}(x))}(x') d\rho(x') = \rho(\mathcal{C}^{-1}\mathcal{C}(x)).$$

Note here that \mathcal{C}^{-1} denotes preimage. One could substitute various probability measures ρ above with mean x and variance σ^2 to obtain different measures of stability corresponding to different ways of sampling the neighborhood of a point. Another natural choice would be sampling the uniform measure on balls of changing radius. Based on the concentration of measure for both of these families of measures (see Appendix 10), we do not anticipate significant qualitative differences in these two approaches. We propose Gaussian sampling because it is also a product measure, which makes it easier to sample and simplifies some other calculations below.

For the Gaussian measure, the probability above may be written more concretely as

$$\frac{1}{(\sqrt{2\pi}\sigma)^n} \int_{\mathbb{R}^n} \mathbb{1}_{\mathcal{C}^{-1}(\mathcal{C}(x))}(x') e^{-\frac{|x-x'|^2}{2\sigma^2}} dx'. \quad (7)$$

In this work, we will conduct experiments in which we estimate this stability for fixed (γ, σ) pairs via a Monte Carlo sampling, in which case the integral (7) is approximated by taking N i.i.d. samples $x_k \sim \rho$ and computing

$$\frac{|\{x_k : \mathcal{C}(x_k) = \mathcal{C}(x)\}|}{N}.$$

Note that this quantity converges to the integral (7) as $N \rightarrow \infty$ by the Law of Large Numbers.

The ability to adjust the quantity γ is important because it is much weaker than a notion of stability that requires a ball that stays away from the decision boundary as in [Khoury and Hadfield-Menell(2018)]. By choosing γ closer to 1, we can require the samples to be more within the same class, and by adjusting γ to be smaller we can allow more overlap.

We also propose a related statistic, *persistence*, by fixing a particular γ and adjusting σ . For any $x \in X$ not on the decision boundary, for any choice of $0 < \gamma < 1$ there exists a σ_γ small enough such that if $\sigma < \sigma_\gamma$ then x is (γ, σ) -stable. We can now take the largest such σ_γ to define persistence.

Definition 4.2. Let $\mathcal{C} : X \rightarrow L$ be a classifier, $x \in X$, and $\gamma \in (0, 1)$. Let σ_γ^* be the maximum σ_γ such that x is (γ, σ) -stable with respect to \mathcal{C} for all $\sigma < \sigma_\gamma$. We say that x has γ -persistence σ_γ^* .

The γ -persistence quantity σ_γ^* measures the stability of the neighborhood of a given x with respect to the output classification. Small persistence indicates that the classifier is unstable in a small neighborhood of x , whereas large persistence indicates stability of the classifier in a small neighborhood of x . In the later experiments, we have generally taken $\gamma = 0.7$. This choice is arbitrary and chosen to fit the problems considered here. In our experiments, we did not see significant change in results with small changes in the choice of γ .

In order to better understand the notion of γ -persistence, consider a decision boundary that is the boundary of a conical wedge domain $W_k^n \subset \mathbb{R}^n$, where $0 < k \leq n$, defined by

$$W_k^n = \{(x_1, \dots, x_n) \in \mathbb{R}^n : x_1 \geq 0, \dots, x_k \geq 0\}.$$

One can consider this as a Cartesian product of a half space and a positive orthant (see Figure 7 for the three possibilities if $n = 3$). The boundary of W_k^n consists of parts of hyperplanes (in fact, W_ℓ^m for $m \leq n$ and $\ell \leq k$). It is clear that for larger k , there is more curvature (this can be made precise using the notion of curvature measures via tube formulas as in [?]). We now explore γ -persistence for data points inside wedge domains, which will serve as a local model for adversarial examples within their adversarial classes.

Proposition 4.3. *If the point $x \in W_k^n$ such that W_k^n describes the class of x and x is a distance d from each sheet of the boundary of W_k^n , then the integral (7) is equal to*

$$\left(\frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{d}{\sqrt{2}\sigma} \right) \right) \right)^k,$$

where $\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-y^2} dy$ is the error function. It follows that the γ -persistence of x is

$$\sigma_\gamma^* = \frac{d}{\sqrt{2} \operatorname{erf}^{-1} (2\gamma^{1/k} - 1)}.$$

Proof. First, consider a decision boundary consisting of a hyperplane in \mathbb{R}^n and suppose that the point x is a distance d from that decision boundary. If we rotate the situation so that the plane is

orthogonal to the x_n -axis, we can compute the integral (7) exactly:

$$\begin{aligned} & \frac{1}{(\sqrt{2\pi}\sigma)^n} \int_{-\infty}^{\infty} \cdots \left(\int_{-\infty}^{\infty} \left(\int_{-\infty}^{x_n+d} e^{-\frac{(x'_n-x_n)^2}{2\sigma^2}} dx'_n \right) e^{-\frac{(x'_{n-1}-x_{n-1})^2}{2\sigma^2}} dx'_{n-1} \right) \cdots e^{-\frac{(x'_1-x_1)^2}{2\sigma^2}} dx'_1 \\ &= \frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{d}{\sqrt{2}\sigma} \right) \right). \end{aligned}$$

Since the Gaussian factorizes, a similar calculation shows that for the point x of distance d away from the sheets of W_k^n , the integral (7) is the k -th power of the expression above. The fact that this is equal to $\mathbb{P}[\mathcal{C}(x') = \mathcal{C}(x)]$ for $x' \sim \rho = N(x, \sigma^2 I)$ completes the proof. \square

Note that these results are independent of the ambient dimension (n) and only depend on the number of hyperplanes (k) making up the cone piece. This fact is related to a similar observation about the Gaussian isoperimetric inequality (e.g., [?]).

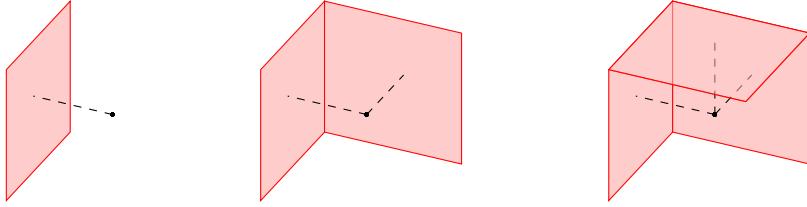


Figure 7: Views of different decision boundaries from Proposition 4.3 in the case $n = 3$ and $k = 1$ (left), $k = 2$ (center), and $k = 3$ (right). The point x of Proposition 4.3 is shown in black, and dashed lines indicate the shortest distances to the boundary.

Observe from Proposition 4.3 that if the distance d to the decision boundary is fixed, then the more planes that make up the decision boundary, the smaller the persistence will be. For illustration, Figure 8 shows the 0.7-persistence for the point x in Proposition 4.3 as a function of the number of planes k with $d = 1$. Observe that when $k = 1$, the 0.7-persistence is approximately 1.91 despite the distance from the decision boundary being equal to 1; this is due to the fact that one must sample from a Gaussian with larger variance to have 30% of the samples be on the other side of a single plane decision boundary. Additionally, as k , and hence curvature of the boundary, increases, the persistence value becomes less than the distance to the nearest decision boundary, which remains 1. This illustrates that persistence carries more information regarding the landscape of the decision boundary about x than merely the distance to the boundary. Note also that if the initial point is outside the cone, then the persistence will be significantly larger. This demonstrates how, even if

we consider points equally close to the decision boundary, the persistence can distinguish between being in a location that is inside a cone and outside a cone, as well as how sharp the boundary of those cones may be. The existence of such cones or curved decision boundaries is both seen experimentally [?, ?] and theorized [?]. It is expected that adversarial examples lie inside the cone, as modeled by the wedge domains, and the natural examples lie outside the cone or wedge domain.

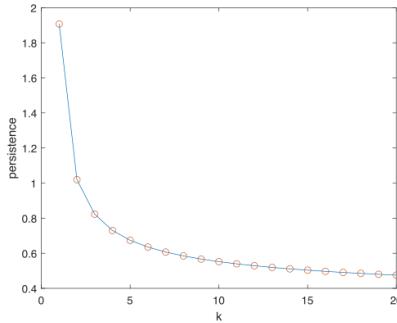


Figure 8: Plot of 0.7-persistence vs. k for a point x that is distance 1 from all sheets of the conic domain W_k^n , as in Proposition 4.3.

In our experiments, we numerically estimate γ -persistence via a bisection algorithm that we term the Bracketing Algorithm (Appendix 7). Briefly, the algorithm first chooses search space bounds σ_{\min} and σ_{\max} such that x is (γ, σ_{\min}) -stable but is not (γ, σ_{\max}) -stable with respect to \mathcal{C} , and then proceeds to evaluate stability by bisection until an approximation of σ_{γ}^* is obtained.

5 Experiments

In this section we investigate the stability and persistence behavior of natural and adversarial examples for MNIST [?] and ImageNet [?] using a variety of different classifiers. For each set of image samples generated for a particular dataset, model, and attack protocol, we study (γ, σ) -stability and γ -persistence of both natural and adversarial images, and also compute persistence along trajectories from natural to adversarial images. In general, we use $\gamma = 0.7$, and note that the observed behavior does not change significantly for small changes in γ . While most of the adversarial attacks considered here have a clear target class, the measurement of persistence does not require considering a particular candidate class. These experiments were performed using PyTorch on an HPC configured with 28 CPUs, 192gb of memory, and 1 nvidia p100 GPU

per node. Total compute time needed to generate the results presented was approximately 300 Teraflop/s-days.

5.1 MNIST Experiments

Since MNIST is relatively small compared to ImageNet, we trained several classifiers with various architectures and complexities and implemented the adversarial attacks directly. Adversarial examples were generated against each of these models using Iterative Gradient Sign Method (IGSM [Kurakin et al.(2016)Kurakin, Goodfellow, and Bengio]) and Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS [?]).

5.1.1 Investigation of (γ, σ) -stability on MNIST

We begin with a fully connected ReLU network with layers of size 784, 100, 20, and 10 and small regularization $\lambda = 10^{-7}$ which is trained on the standard MNIST training set. We then start with a randomly selected MNIST test image x_1 from the 1's class and generate adversarial examples x_0, x_2, \dots, x_9 using IGSM for each target class other than 1. The neighborhoods around each x_i are examined by generating 1000 i.i.d. samples from $N(x_i, \sigma^2 I)$ for each of 100 equally spaced standard deviations $\sigma \in (0, 1.6)$. Figure 9 shows the results of the Gaussian perturbations of a natural example x_1 of the class labeled 1 and the results of Gaussian perturbations of the adversarial example x_0 targeted at the class labeled 0. We provide other examples of x_2, \dots, x_9 in the supplementary materials. Note that the original image is very stable under perturbation, while the adversarial image is not.

5.1.2 Persistence of adversarial examples for MNIST

To study persistence of adversarial examples on MNIST, we take the same network architecture as in the previous subsection and randomly select 200 MNIST images. For each image, we used IGSM to generate 9 adversarial examples (one for each target class) yielding a total of 1800 adversarial examples. In addition, we randomly sampled 1800 natural MNIST images. For each of the 3600 images, we computed 0.7-persistence; the results are shown in Figure 10. One sees that 0.7-persistence of adversarial examples tends to be significantly smaller than that of natural

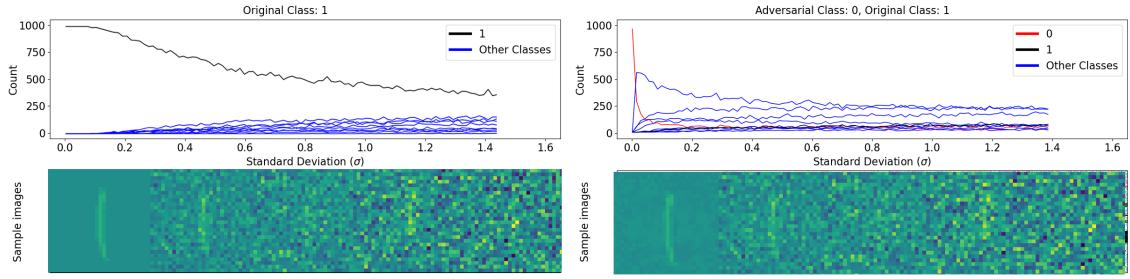


Figure 9: Frequency of each class in Gaussian samples with increasing variance around a natural image of class 1 (left) and around an adversarial attack of that image targeted at 0 generated using IGSM (right). The adversarial class (0) is shown as a red curve. The natural image class (1) is shown in black. Bottoms show example sample images at different standard deviations for natural (left) and adversarial (right) examples.

examples for this classifier, indicating that they are generally less stable than natural images. We will see subsequently that this behavior is typical.

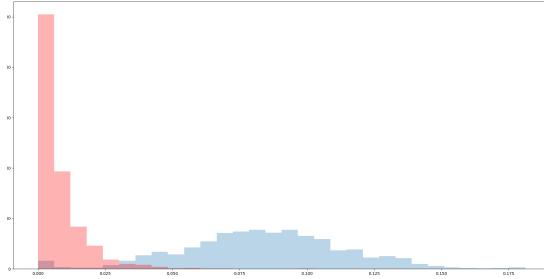


Figure 10: Histogram of 0.7-persistence of IGSM-based adversarial examples (red) and natural examples (blue) on MNIST.

Next, we investigate the relationship of network complexity and (γ, σ) -stability by revisiting the now classic work of [Szegedy et al.(2013)](Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus) on adversarial examples. Table 1 recreates and adds on to part of [Szegedy et al.(2013)](Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus) Table 1] in which networks of differing complexity are trained and attacked using L-BFGS. The table contains new columns showing the average 0.7-persistence for both natural and adversarial examples for each network, as well as the average distortion for the adversarial examples. The distortion is the ℓ^2 -norm divided by square root of the dimension n . The first networks listed are of the form FC10-k, and are fully connected single layer ReLU networks that map each input vector $x \in \mathbb{R}^{784}$ to an output vector $y \in \mathbb{R}^{10}$ with a regularization added to the objective function of the form $\lambda \|w\|_2 / N$, where $\lambda = 10^{-k}$ and N is the number of parameters in the weight vector

w defining the network. The higher values of λ indicate more regularization. FC100-100-10 and FC200-200-10 are networks with 2 hidden layers (with 100 and 200 nodes, respectively) with regularization added for each layer of perceptrons with the λ for each layer equal to 10^{-5} , 10^{-5} , and 10^{-6} . Training for these networks was conducted with a fixed number of epochs (typically 21). For the bottom half of Table 1, we also considered networks with four convolutional layers plus a max-pooling layer connected by ReLU to a fully connected hidden layer with increasing numbers of channels denoted as “C-Ch,” where C reflects that this is a CNN and Ch denotes the number of channels. A more detailed description of these networks can be found in Appendix 8.

Table 1: Recreation of [Szegedy et al.(2013)] for the MNIST dataset. For each network, we show Testing Accuracy (in %), Average Distortion ($\|x\|_2/\sqrt{n}$) of adversarial examples, and new columns show average 0.7-persistence values for natural (Nat) and adversarial (Adv) images. 300 natural and 300 adversarial examples generated with L-BFGS were used for each aggregation.

Network	Test Acc	Avg Dist	Persist (Nat)	Persist (Adv)
FC10-4	92.09	0.123	0.93	1.68
FC10-2	90.77	0.178	1.37	4.25
FC10-0	86.89	0.278	1.92	12.22
FC100-100-10	97.31	0.086	0.65	0.56
FC200-200-10	97.61	0.087	0.73	0.56
C-2	95.94	0.09	3.33	0.027
C-4	97.36	0.12	0.35	0.027
C-8	98.50	0.11	0.43	0.0517
C-16	98.90	0.11	0.53	0.0994
C-32	98.96	0.11	0.78	0.0836
C-64	99.00	0.10	0.81	0.0865
C-128	99.17	0.11	0.77	0.0883
C-256	99.09	0.11	0.83	0.0900
C-512	99.22	0.10	0.793	0.0929

The main observation from Table 1 is that for higher complexity networks, adversarial examples tend to have smaller persistence than natural examples. Histograms reflecting these observations can be found in the supplemental material. Another notable takeaway is that for models with fewer effective parameters, the attack distortion necessary to generate a successful attack is so great that the resulting image is often more stable than a natural image under that model, as seen particularly in the FC10 networks. Once there are sufficiently many parameters available in the neural network, we found that both the average distortion of the adversarial examples and the average 0.7-persistence of the adversarial examples tended to be smaller. This observation is

consistent with the idea that networks with more parameters are more likely to exhibit decision boundaries with more curvature.

5.2 Results on ImageNet

For ImageNet [?], we used pre-trained ImageNet classification models, including alexnet [?] and vgg16 [Simonyan and Zisserman(2014)]. We then generated attacks based on the ILSVRC 2015 [?] validation images for each of these networks using a variety of modern attack protocols, including Fast Gradient Sign Method (FGSM [Goodfellow et al.(2014)Goodfellow, Shlens, and Szegedy]), Momentum Iterative FGSM (MIFGSM [?]), Basic Iterative Method (BIM [Kurakin et al.(2016)Kurakin, Goodfellow, and Bengio]), Projected Gradient Descent (PGD [?]), Randomized FGSM (R+FGSM [?]), and Carlini-Wagner (CW [?]). These were all generated using the TorchAttacks [?] toolset.

5.2.1 Investigation of (γ, σ) -stability on ImageNet

In this section, we show the results of Gaussian neighborhood sampling in ImageNet. Figures 11 and 12 arise from vgg16 and adversarial examples created with BIM; results for other networks and attack strategies are similar, with additional figures in the supplementary material. Figure 11 (left) begins with an image x with label goldfinch. For each equally spaced $\sigma \in (0, 2)$, 100 i.i.d. samples were drawn from the Gaussian distribution $N(x, \sigma^2 I)$, and the counts of the vgg16 classification for each label are shown. In Figure 11 (right), we see the same plot, but for an adversarial example targeted at the class indigo_bunting, which is another type of bird, using the BIM attack protocol.

The key observation in Figure 11 is that the frequency of the class of the adversarial example (indigo_bunting, shown in red) falls off much quicker than the class for the natural example (goldfinch, shown in black). In this particular example, the original class appears again after the adversarial class becomes less prevalent, but only for a short period of σ , after which other classes begin to dominate. In some examples the original class does not dominate at all after the decline of the adversarial class. The adversarial class almost never dominates for a long period of σ .

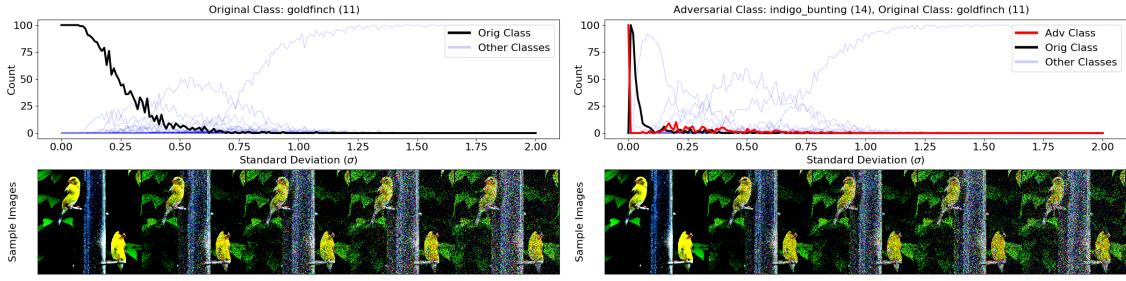


Figure 11: Frequency of each class in Gaussian samples with increasing variance around a goldfinch image (left) and an adversarial example of that image targeted at the `indigo_bunting` class and calculated using the BIM attack (right). Bottoms show example sample images at different standard deviations for natural (left) and adversarial (right) examples.

5.2.2 Persistence of adversarial examples on ImageNet

Figure 12 shows a plot of the 0.7-persistence along the straight-line path between a natural example and adversarial example as parametrized between 0 and 1. It can be seen that the dropoff of persistence occurs precisely around the decision boundary. This indicates some sort of curvature favoring the class of the natural example, since otherwise the persistence would be roughly the same as the decision boundary is crossed.

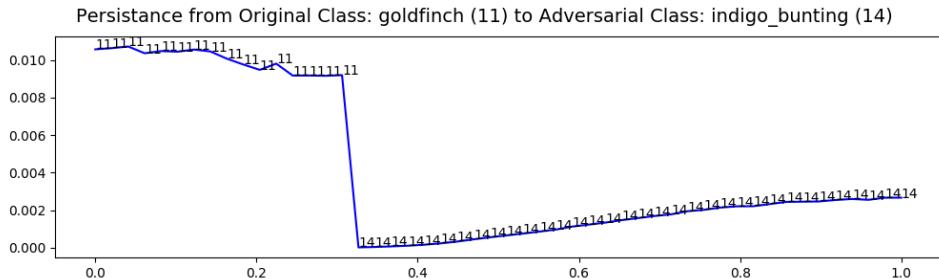


Figure 12: The 0.7-persistence of images along the straight line path from an image in class `goldfinch` (11) to an adversarial image generated with BIM in the class `indigo_bunting` (14) on a vgg16 classifier. The classification of each image on the straight line is listed as a number so that it is possible to see the transition from one class to another. The vertical axis is 0.7-persistence and the horizontal axis is progress towards the adversarial image.

An aggregation of persistence for many randomly selected images from the `goldfinch` class in the validation set for Imagenet are presented in Table 2. For each image of a `goldfinch` and for each network of alexnet and vgg16, attacks were prepared to a variety of 28 randomly selected targets using a BIM, MIFGSM, PGD, FGSM, R+FGSM, and CW attack strategies. The successful attacks were aggregated and their 0.7-persistences were computed using the Bracketing Algorithm along

with the 0.7-persistences of the original images from which each attack was generated. Each attack strategy had a slightly different mixture of which source image and attack target combinations resulted in successful attacks. The overall rates for each are listed, as well as particular results on the most successful attack strategies in our experiments, BIM, MIFGSM, and PGD. The results indicate that adversarial images generated for these networks (alexnet and vgg16) using these attacks were less persistent, and hence less stable, than natural images for the same models.

Table 2: The 0.7-persistence values for natural (Nat) and adversarial (Adv) images along with average distortion for adversarial images of alexnet and vgg16 for attacks generated with BIM, MIFGSM, and PGD on images from class `goldfinch` targeted toward other classes from the ILSVRC 2015 classification labels.

Network/Method	Avg Dist	Persist (Nat)	Persist (Adv)
alexnet (total)	0.0194	0.0155	0.0049
BIM	0.0188	0.0162	0.0050
MIFGSM	0.0240	0.0159	0.0053
PGD	0.0188	0.0162	0.0050
vgg16 (total)	0.0154	0.0146	0.0011
BIM	0.0181	0.0145	0.0012
MIFGSM	0.0238	0.0149	0.0018
PGD	0.0181	0.0145	0.0012

6 Conclusion

In order to better understand the observed tendency for points near natural data to be classified similarly and points near adversarial examples to be classified differently, we defined a notion of (γ, σ) -stability which is easily estimated by Monte Carlo sampling. For any data point x , we then define the γ -persistence to be the smallest σ_γ such that the probability of similarly classified data is at least γ when sampling from Gaussian distributions with mean x and standard deviation less than σ_γ . The persistence value can be quickly estimated by a Bracketing Algorithm. These two measures were considered with regard to both the MNIST and ImageNet datasets and with respect to a variety of classifiers and adversarial attacks. We found that adversarial examples were much less stable than natural examples in that the 0.7-persistence for natural data was usually significantly larger than the 0.7-persistence for adversarial examples. We also saw that the dropoff of the persistence tends to happen precisely near the decision boundary. Each of these observations

is strong evidence toward the hypothesis that adversarial examples arise inside cones or high curvature regions in the adversarial class, whereas natural images lie outside such regions.

We also found that often the most likely class for perturbations of an adversarial examples is a class other than the class of the original natural example used to generate the adversarial example; instead, some other background class is favored. In addition, we found that some adversarial examples may be more stable than others, and a more detailed probing using the concept of (γ, σ) -stability and the γ -persistence statistic may be able to help with a more nuanced understanding of the geometry and curvature of the decision boundary. Although not pursued here, the observations and statistics used in this paper could potentially be used to develop methods to detect adversarial examples as in [?, ?, ?, ?, ?, ?] and others. As with other methods of detection, this may be susceptible to adaptive attacks as discussed by [?].

7 Bracketing Algorithm

The Bracketing Algorithm is a way to determine persistence of an image with respect to a given classifier, typically a DNN. The algorithm was implemented in Python for the experiments presented. The RANGEFINDER function is not strictly necessary, in that one could directly specify values of σ_{\min} and σ_{\max} , but we include it here so that the code could be automated by a user if so desired.

8 Convolutional neural networks used

In Table 1 we reported results on varying complexity convolutional neural networks. These networks consist of a composition of convolutional layers followed by a maxpool and fully connected layers. The details of the network layers are described in Table 3 where Ch is the number of channels in the convolutional components.

Algorithm 1 Bracketing algorithm for computing γ -persistence

```
function BRACKETING(image, classifier ( $\mathcal{C}$ ), numSamples,  $\gamma$ , maxSteps, precision)
     $[\sigma_{\min}, \sigma_{\max}] = \text{RANGEFINDER}(image, \mathcal{C}, \text{numSamples}, \gamma)$ 
    count = 1
    while count < maxSteps do
         $\sigma = \frac{\sigma_{\min} + \sigma_{\max}}{2}$ 
         $\gamma_{\text{new}} = \text{COMPUTE\_PERSISTENCE}(\sigma, \text{image}, \text{numSamples}, \mathcal{C})$ 
        if  $|\gamma_{\text{new}} - \gamma| < \text{precision}$  then
            return  $\sigma$ 
        else if  $\gamma_{\text{new}} > \gamma$  then
             $\sigma_{\min} = \sigma$ 
        else
             $\sigma_{\max} = \sigma$ 
        end if
        count = count + 1
    end while
    return  $\sigma$ 
end function

function RANGEFINDER(image,  $\mathcal{C}$ , numSamples,  $\gamma$ )
     $\sigma_{\min} = .5, \sigma_{\max} = 1.5$ 
     $\gamma_1 = \text{COMPUTE\_PERSISTENCE}(\sigma_{\min}, \text{image}, \text{numSamples}, \mathcal{C})$ 
     $\gamma_2 = \text{COMPUTE\_PERSISTENCE}(\sigma_{\max}, \text{image}, \text{numSamples}, \mathcal{C})$ 
    while  $\gamma_1 < \gamma$  or  $\gamma_2 > \gamma$  do
        if  $\gamma_1 < \gamma$  then
             $\sigma_{\min} = .5\sigma_{\min}$ 
             $\gamma_1 = \text{COMPUTE\_PERSISTENCE}(\sigma_{\min}, \text{image}, \text{numSamples}, \mathcal{C})$ 
        end if
        if  $\gamma_2 > \gamma$  then
             $\sigma_{\max} = 2\sigma_{\max}$ 
             $\gamma_2 = \text{COMPUTE\_PERSISTENCE}(\sigma_{\max}, \text{image}, \text{numSamples}, \mathcal{C})$ 
        end if
    end while
    return  $[\sigma_{\min}, \sigma_{\max}]$ 
end function

function COMPUTE_PERSISTENCE( $\sigma$ , image, numSamples,  $\mathcal{C}$ )
    sample =  $N(\text{image}, \sigma^2 I, \text{numSamples})$ 
     $\gamma_{\text{est}} = \frac{|\{\mathcal{C}(\text{sample}) = \mathcal{C}(\text{image})\}|}{\text{numSamples}}$ 
    return  $\gamma_{\text{est}}$ 
end function
```

9 Additional Figures

In this section we provide additional figures to demonstrate some of the experiments from the paper.

Table 3: Structure of the CNNs C-Ch used in Table 1

Layer	Type	Channels	Kernel	Stride	Output Shape
0	Image	1	NA	NA	(1, 28, 28)
1	Conv	Ch	(5, 5)	(1, 1)	(Ch, 24, 24)
2	Conv	Ch	(5, 5)	(1, 1)	(Ch, 20, 20)
3	Conv	Ch	(5, 5)	(1, 1)	(Ch, 16, 16)
4	Conv	Ch	(5, 5)	(1, 1)	(Ch, 12, 12)
5	Max Pool	Ch	(2, 2)	(2, 2)	(Ch, 6, 6)
7	FC	(Ch · 6 · 6, 256)	NA	NA	256
8	FC	(256, 10)	NA	NA	10

9.1 Additional figures from MNIST

In Figure 13 we begin with an image of a 1 and generate adversarial examples to the networks described in Section 5.1.1 via IGSM targeted at each class 2 through 9; plotted are the counts of output classifications by the DNN from samples from Gaussian distributions with increasing standard deviation; this complements Figure 9 in the main text. Note that the prevalence of the adversarial class falls off quickly in all cases, though the rate is different for different choices of target class.

We also show histograms corresponding to those in Figure 10 and the networks from Table 1. As before, for each image, we used IGSM to generate 9 adversarial examples (one for each target class) yielding a total of 1800 adversarial examples. In addition, we randomly sampled 1800 natural MNIST images. For each of the 3600 images, we computed 0.7-persistence. In Figure 14, we see histograms of these persistences for the small fully connected networks with increasing levels of regularization. In each case, the test accuracy is relatively low and distortion relatively high. It should be noted that these high-distortion attacks against models with few effective parameters were inherently very stable – resulting in most of the “adversarial” images in these sets having higher persistence than natural images. This suggests a lack of the sharp conical regions which appear to characterize adversarial examples generated against more complicated models. In Figure 15 we see the larger fully connected networks from Table 1 and in Figure 16 we see some of the convolutional neural networks from Table 1.

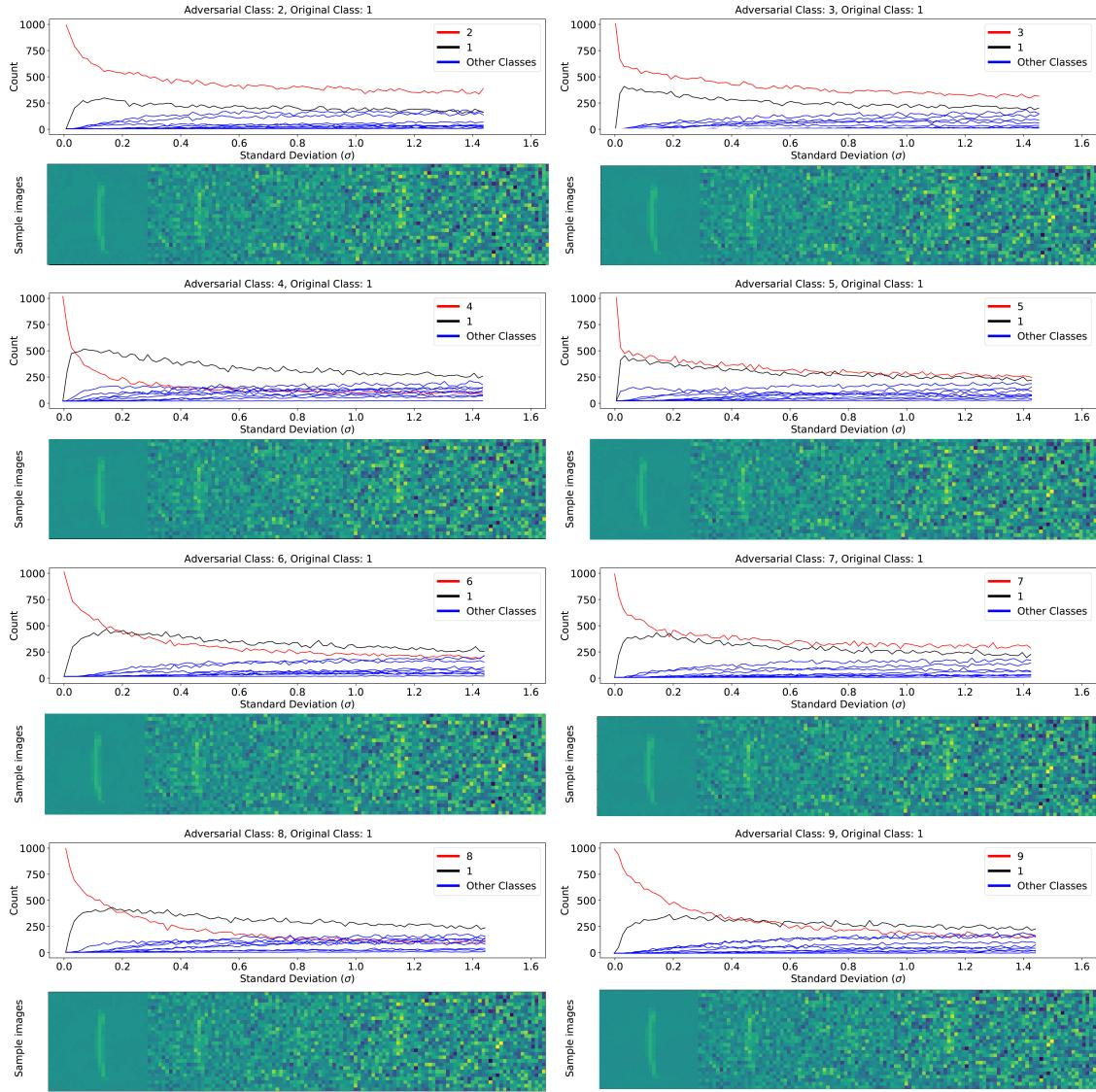


Figure 13: Frequency of each class in Gaussian samples with increasing standard deviations around adversarial attacks of an image of a 1 targeted at classes 2 through 9 on a DNN classifier generated using IGSM. The adversarial class is shown as a red curve. The natural image class (1) is shown in black. Bottoms show example sample images at different standard deviations.

9.2 Additional figures for ImageNet

In this section we show some additional figures of Gaussian sampling for ImageNet. In Figure 17 we see Gaussian sampling of an example of the class `indigo_bunting` and the frequency samplings for adversarial attacks of `goldfinch` toward `indigo_bunting` (classifier: alexnet, attack:

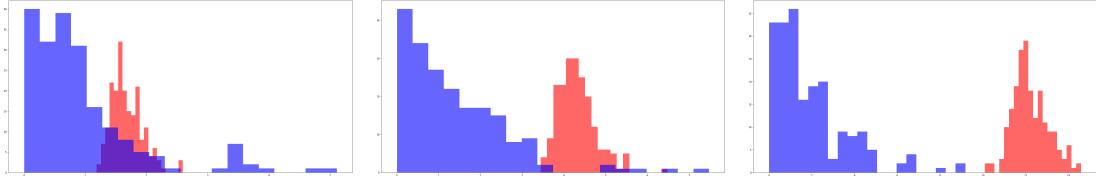


Figure 14: Histograms of 0.7-persistence for FC10-4 (smallest regularization, left), FC10-2 (middle), and FC10-0 (most regularization, right) from Table 1. Natural images are in blue, and adversarial images are in red. Note that these are plotted on different scales – higher regularization forces any “adversaries” to be very stable.

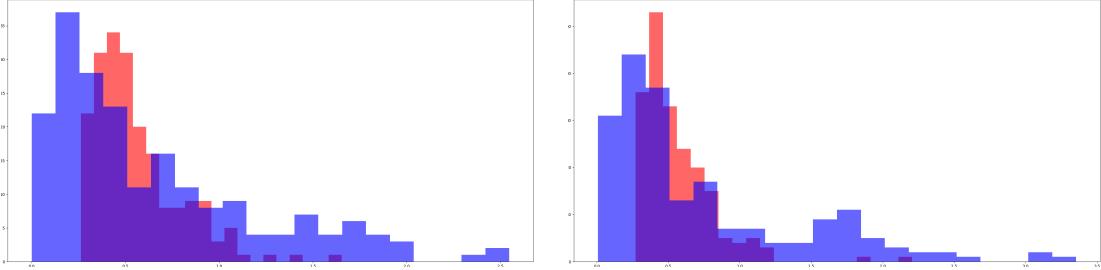


Figure 15: Histograms of 0.7-persistence for FC100-100-10 (left) and FC200-200-10 (right) from Table 1. Natural images are in blue, and adversarial images are in red.

PGD) and toward `alligator_lizard` (classifier: vgg16, attack: PGD). Compare the middle image to Figure 11, which is a similar adversarial attack but used the vgg16 network classifier and the BIM attack. Results are similar. Also note that in each of the cases in Figure 17 the label of the original natural image never becomes the most frequent classification when sampling neighborhoods of the adversarial example.

In Figure 18, we have plotted γ -persistence along a straight line from a natural image to an adversarial image to it with differing values of the parameter γ . The γ -persistence in each case seems to change primarily when crossing the decision boundary. Interestingly, while the choice of γ does not make too much of a difference in the left subplot, it leads to more varying persistence values in the right subplot of Figure 18. This suggests that one should be careful not to choose too small of a γ value, and that persistence does indeed depend on the landscape of the decision boundary described by the classifier.

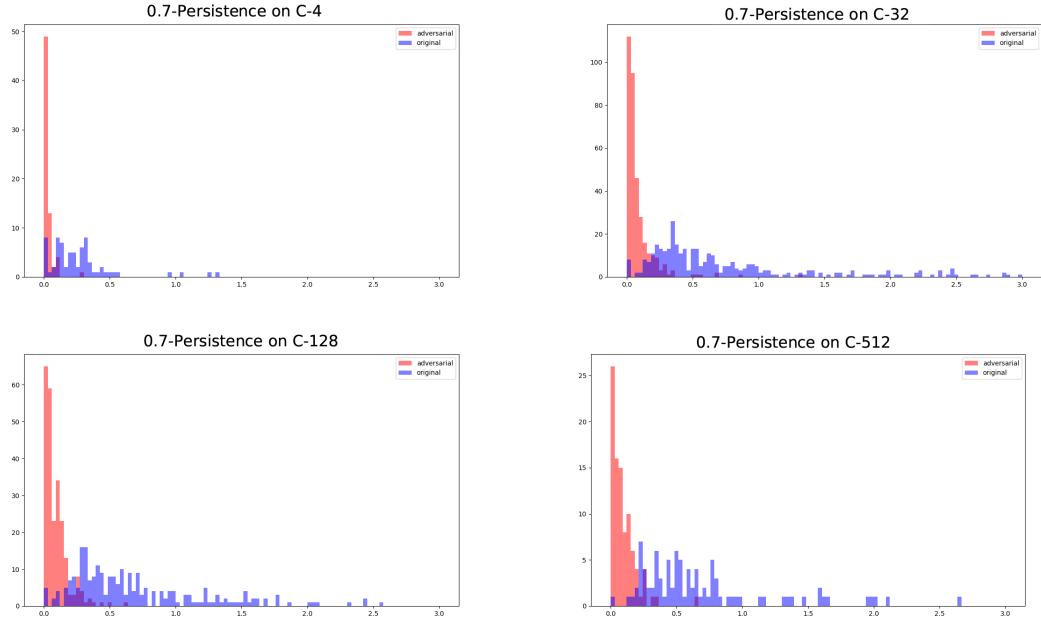


Figure 16: Histograms of 0.7-persistence for C-4 (top left), C-32 (top right), C-128 (bottom left), and C-512 (bottom right) from Table 1. Natural images are in blue and adversarial images are in red.

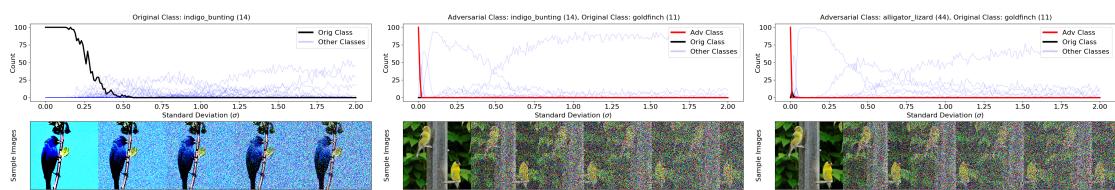


Figure 17: Frequency of each class in Gaussian samples with increasing variance around an *indigo_bunting* image (left), an adversarial example of the image in class *goldfinch* from Figure 11 targeted at the *indigo_bunting* class on a alexnet network attacked with PGD (middle), and an adversarial example of the *goldfinch* image targeted at the *alligator_lizard* class on a vgg16 network attacked with PGD (right). Bottoms show example sample images at different standard deviations.

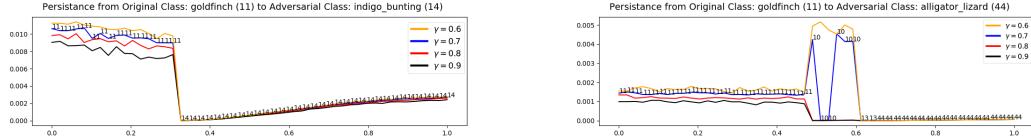


Figure 18: The γ -persistence of images along the straight line path from an image in class goldfinch (11) to an adversarial image generated with BIM in the class indigo_bunting (14) (left) and to an adversarial image generated with PGL in the class alligator_lizard (44) (right) on a vgg16 classifier with different values of γ . The classification of each image on the straight line is listed as a number so that it is possible to see the transition from one class to another. The vertical axis is γ -persistence and the horizontal axis is progress towards the adversarial image.

10 Concentration of measures

We use Gaussian sampling with varying standard deviation instead of sampling the uniform distributions of balls of varying radius, denoted $U(B_r(0))$ for radius r and center 0. This is for two reasons. The first is that Gaussian sampling is relatively easy to do. The second is that the concentration phenomenon is different. This can be seen in the following proposition.

Proposition 10.1. *Suppose $x \sim N(0, \sigma^2 I)$ and $y \sim U(B_r(0))$ where both points come from distributions on \mathbb{R}^n . For $\varepsilon < \sqrt{n}$ and for $\delta < r$ we find the following:*

$$\mathbb{P} \left[\left| \|x\| - \sigma\sqrt{n} \right| \leq \varepsilon \right] \geq 1 - 2e^{-\varepsilon^2/16} \quad (8)$$

$$\mathbb{P} \left[\left| \|y\| - r \right| \leq \delta \right] \geq 1 - e^{-\delta n/r} \quad (9)$$

Proof. This follows from [?, Theorems 4.7 and 3.7], which are the Gaussian Annulus Theorem and the concentration of measure for the unit ball, when taking account of varying the standard deviation σ and radius r , respectively. \square

The implication is that if we fix the dimension and let σ vary, the measures will always be concentrated near spheres of radius $\sigma\sqrt{n}$ and r , respectively, in a consistent way. In practice, Gaussians seem to have a bit more spread, as indicated in Figure 19, which shows the norms of 100,000 points sampled from dimension $n = 784$ (left, the dimension of MNIST) and 5,000 points sampled from dimension $n = 196,608$ (right, the dimension of ImageNet).

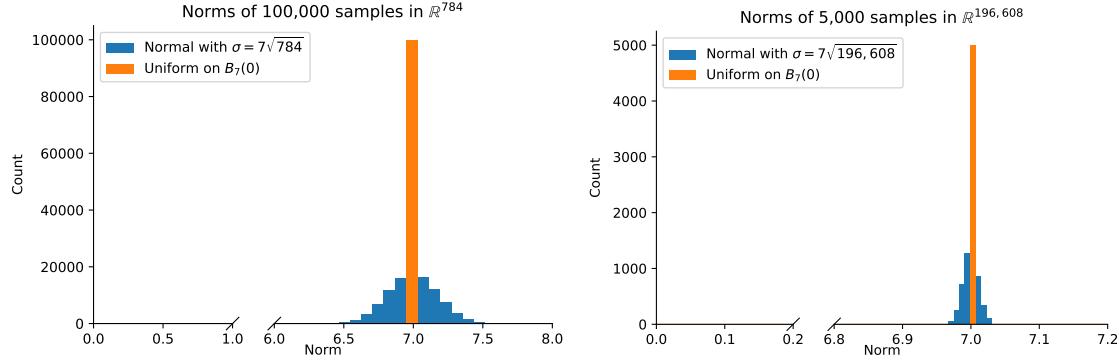


Figure 19: Comparison of the length of samples drawn from $U(B_7(0))$ and $N(0, 7\sqrt{n})$ for $n = 784$, the dimension of MNIST, (left) and $n = 196,608$, the dimension of ImageNet, (right).

Part II

Model Geometry

11 Neural Networks are (Mostly) Kernel Machines

Abstract

In this work, we write an exact representation for an arbitrary gradient trained model as a kernel based method extending the theory of the path kernel described in [?] and discussed by [?]. We discuss the conditions under which this representation is exact, measure approximation error, and compare to the well known Neural Tangent Kernel (NTK) [?]. We implement this representation for an artificial neural network and demonstrate that it is computationally tractable and accurate in practice. Using this kernel, we quantify uncertainty according to this kernel using Gaussian process regression and discuss the implicit limitations that this reveals about neural networks. In particular, we show that the kernel resulting from a typical neural network is non-stationary and has highly unusual spatial properties.

12 Introduction

TODO rewrite to focus on the second paper.

This study investigates the relationship between kernel methods and models whose parame-

ters are determined using gradient methods. The neural tangent kernel (NTK) is a well established method which represents training gradients as a kernel method [?] and compares trained models with this kernel in the case of models as their number of parameters approaches infinity. A theory of the neural path kernel (NPK) has been proposed which integrates finite tangent kernels [?]. The path kernel extends from the NTK by directly constructing a kernel method approximation of a model by integrating tangent kernels along the continuous gradient flow defined by the model’s gradient with respect to its training data and loss function. The case is made that this continuous path kernel is an exact representation of a continuous gradient trained model and that any practical model trained using discrete steps according to gradient descent is therefore approximately a kernel method. This opens up such models, including artificial neural networks (ANNs), to many theoretical tools available to kernel methods ([?, ?, ?]). However usage of such tools is highly dependent on the accuracy and dynamics of this approximation. Furthermore, the smooth path kernel requires difficult measurements of convergence and error in order to make practical comparisons with real discretely trained models. Although the convergence argued for likely holds as training step size converges to zero, these arguments do not help us understand the dynamics of this convergence and are very difficult to compute in practice.

In this work we propose an exact kernel representation for any gradient trained model satisfying very few conditions and demonstrate that this method is computable, practical, and exposes convergence and approximation properties to rigorous analysis. Our results demonstrate the potential for using the path kernel to study deep neural networks and provide a foundation for further research in this area. Furthermore, we will discuss relaxations of the required conditions on this kernel that allows very general functions to be represented with a small measurable approximation that can be bounded in practice. In addition, we study methods for approximating our discrete path kernel to reduce computation costs while maintaining many of these useful properties.

13 Related Work

[?] NTK

14 Discrete Path Kernels

Models trained by gradient descent can be characterized by a discrete set of intermediate states in the space of their parameters. These states are not bound to the gradient flow defined for such models, so we must consider how to integrate a discrete path for weights whose states differ from the gradient flow. In order to write this representation we must carefully define both kernel methods and kernels:

Definition 14.1. *A kernel is a function of two variables which is symmetric and positive definite.*

Definition 14.2. *Given a Hilbert space X , a query point $x \in X$, and a training set $X_T \subset X$, a Kernel Method is a model characterized by*

$$\hat{y}(x) = b + \sum_i a_i k(x, x_i) \quad (10)$$

where the $a_i \in \mathbb{R}$ do not depend on x , $b \in \mathbb{R}$ is a constant, and k is a kernel.

By Mercer's theorem [?] a kernel can be produced by composing an inner product on a Hilbert space with a mapping ϕ from the space of data into the chosen Hilbert space. We will first derive a kernel which is an exact representation of the change in model output over one training step, and then compose our final representation by summing along the finitely many steps.

Definition 14.3. *Let y_w be a differentiable function parameterized by $w \in \mathbb{R}^d$ which is trained via N forward Euler steps of fixed size ϵ on a finite subset $X_T = \{x_i\}_{i=1}^M$ of a Hilbert space X of size M with labels $Y_T = \{y_i\}_{i=1}^M$, with initial parameters w_0 so that there is a constant $b \in \mathbb{R}$ such that for all x , $\hat{y}_{w_0}(x) = b$, and weights at each step $w_s : 0 \leq s \leq N$. Let $x \in X$ be arbitrary and within the domain of \hat{y}_w for every w . Then the discrete path kernel (DPK) can be written*

$$K_{DPK}(x, x') = \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x') \rangle dt \quad (11)$$

where

$$a_{i,s} = -\varepsilon \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \in \mathbb{R} \quad (12)$$

$$\phi_{s,t}(x) = \nabla_w \hat{y}_{w_s(t,x)}(x) \quad (13)$$

$$w_s(t, x) = \begin{cases} w_s, & x \in X_T \\ w_s(t), & x \notin X_T \end{cases} \quad (14)$$

Lemma 14.4. *The discrete path kernel (DPK) is a kernel.*

Theorem 14.5 (Exact Kernel Representation). *A model \hat{y}_{w_N} trained using discrete steps matching the conditions of the discrete path kernel has the following exact kernel method representation:*

$$\hat{y}_{w_N}(x) = b + \sum_{i=1}^M \sum_{s=1}^N a_{i,s} K_{DPK}(x, x') \quad (15)$$

Remark 0 We can see that by changing equation 15 we can produce an exact representation for any discrete optimization scheme that can be written in terms of model gradients. This could include backward Euler, leapfrog, higher order schemes (which are generally intractable for Artificial Neural Networks), and any variation of adaptive step sizes.

Remark 1 $\phi_{s,t}(x)$ depends on both s and t , which is non-standard but valid, however an important consequence of this mapping is that the output of this representation is not guaranteed to be continuous. This discontinuity is exactly measuring the error between the model along the discrete path compared with the gradient flow for each step. We can write another function k' which is continuous but not symmetric, but still produces an exact representation:

$$k'(x, x') = \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x') \rangle \quad (16)$$

The resulting function is a valid kernel if and only if for every s and every x ,

$$\int_0^1 \nabla_w \hat{y}_{w_s(t)}(x) dt = \nabla_w \hat{y}_{w_s(0)}(x) \quad (17)$$

The asymmetry of this function is exactly measuring the disagreement between the discrete steps

taken during training with the gradient field defined by the loss function composed with the model. This function is one of several subjects for further study, particularly in the context of gaussian processes whereby the asymmetric matrix corresponding with this function can stand in for a covariance matrix. It may be that gaussian thermostats can be used to repair they asymmetry while accounting for any loss accrued due to disagreement between the discrete steps and the gradient flow or it may be that the not-symmetric analogue of the covariance in this case has physical meaning relative to uncertainty.

Remark 2 In order to obtain an exact representation, one must start with a model that has constant output for all input, i.e. for every x and $\hat{y}_0(x) = b$ (e.g. an ANN with all weights in the final layer initialized to 0). When relaxing this property, to allow for models that have a non-constant starting output, we note that this representation ceases to be exact. The resulting approximate representation will still agree strongly with the ANN, and will converge quickly in output as the ratio of the length of the training path divided by the step size goes to infinity. In fact this convergence is very rapid and useful approximation will be achieved within typical training length. (TODO : lipshitz argument)

Remark 3 We note that since f is being trained using forward Euler, we can write:

$$\frac{\partial w_s(t)}{\partial t} = -\varepsilon \nabla_w L(\hat{y}_{w_s}(x_i), y_i) = -\varepsilon \sum_{j=1}^d \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial w_j} \quad (18)$$

In other words, our parameterize of this step depends on the step size ε and as $\varepsilon \rightarrow 0$, we have

$$\int_0^1 \nabla_w \hat{y}_{w_s, \varepsilon}(t)(x) dt \rightarrow \nabla_w \hat{y}_{w_s(0)}(x) \quad (19)$$

In particular, given a model \hat{y} that admits a Lipshitz constant K this approximation has error bounded by εK and a proof of this convergence is direct.

15 Experimental Results

Our first experiments test the kernel formulation on a dataset which can be visualized in 2d. These experiments serve as a sanity check and provide an interpretable representation of what the kernel is learning.

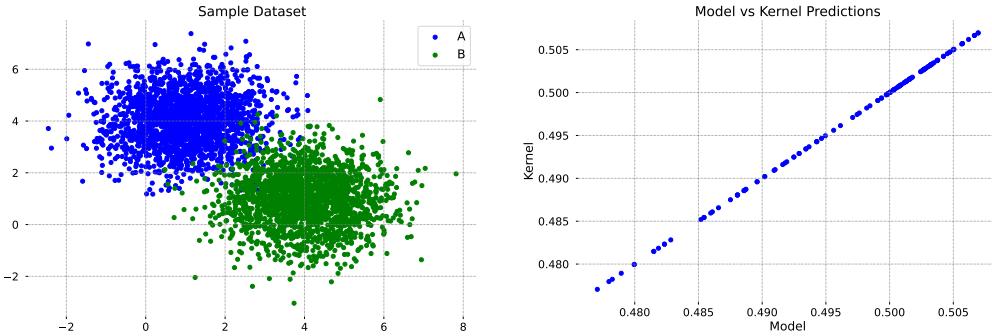


Figure 20: On the left is a 2d dataset of points sampled from gaussians with different means. Specifically, class A is normally distributed with $\mu = [1, 4]$ and $\sigma^2 = 1$ while class B is $\mu = [4, 1]$ and $\sigma^2 = 1$. 2000 data points were sampled for each class. These values were chosen arbitrarily to provide separation with a limited amount of overlap. On the right is the prediction similarity between the kernel and the original model. This demonstrates that our kernel formulation accurately represents the trained network.

15.1 Evaluating The Kernel

Examples of the kernel values across 4 test points are shown in Figure 21. We are interested in how the kernel is learning and whether this kernel will allow out-of-distribution (OOD) detection.

From these plots we see that the in-distribution samples have a significantly higher sum over kernel distances than the OOD examples. Of note is that the OOD detection is not perfect. For the test point (4.0, 15.0) it still identifies a large portion of class A samples as being relatively close in kernel space. Despite this, both OOD examples shown are significantly lower in total kernel distance than the in-distribution samples. Further experiments will be required to better understand why some OOD regions are closer than others

15.2 Extending To Image Data

We perform experiments on MNIST to demonstrate the applicability to image data. This kernel representation was generated for a two-layer fully connected ReLU Network with the cross-entropy loss-function, using Pytorch (citation). The model was trained using forward Euler (gradient descent) using gradients generated as a sum over all training data for each step. The state of the model was saved for every training step. In order to compute the per-training-point gradients needed for the kernel representation, the per-input jacobians are computed at execution time in the representation by loading the model for each training step i , computing the jacobians

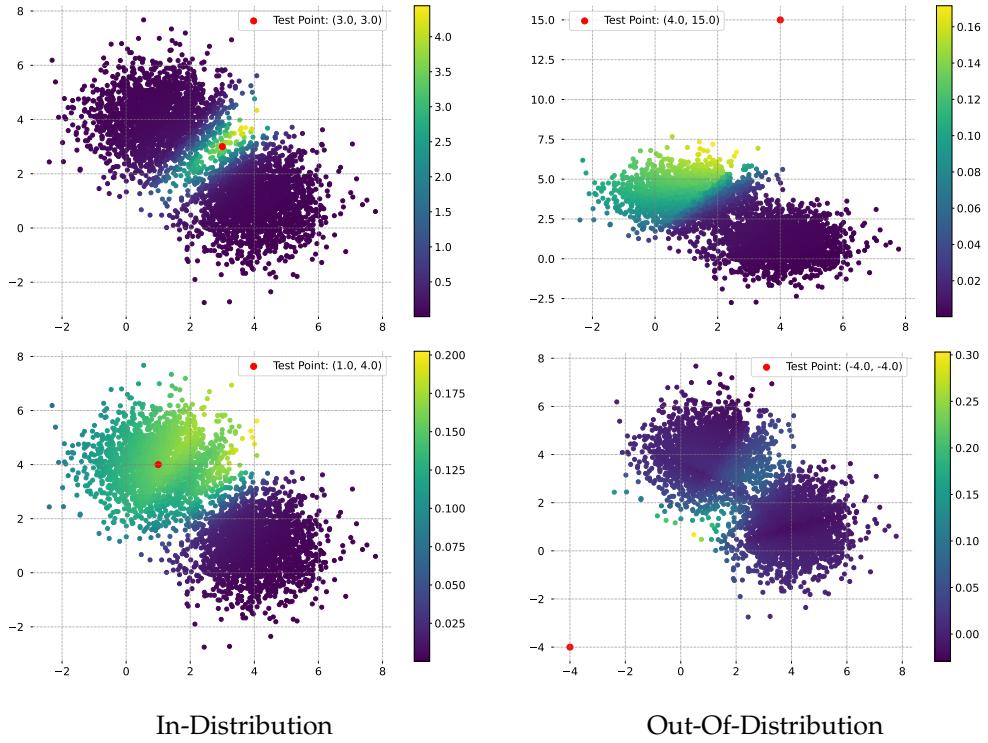


Figure 21: Example of the kernel values on in-distribution and out-of-distribution (OOD) data. Left column shows samples which are in-distribution for our dataset. Right column row shows OOD samples.

for each training input to compute $\nabla_w \hat{y}_{w_s(0)}(x_i)$, and then repeating this procedure for 100 t values between 0 and 1 in order to approximate $\int_0^1 \hat{y}_{w_s(t)}(x)$. Torch is not currently optimized to provide per-input jacobians or to provide jacobians for multiple weight states, so this procedure can be accelerated greatly by deeper integration with existing pytorch tools.

We are able to see the agreement between the neural network and its kernel representation in figure 23. In figure 22 we see that the function learned by the kernel does not directly mimic euclidean distance in the image space. Samples which are nearby in kernel space are not necessarily nearby in pixel space. The similarity metric learned is a direct explanation of how the neural network is making decisions.

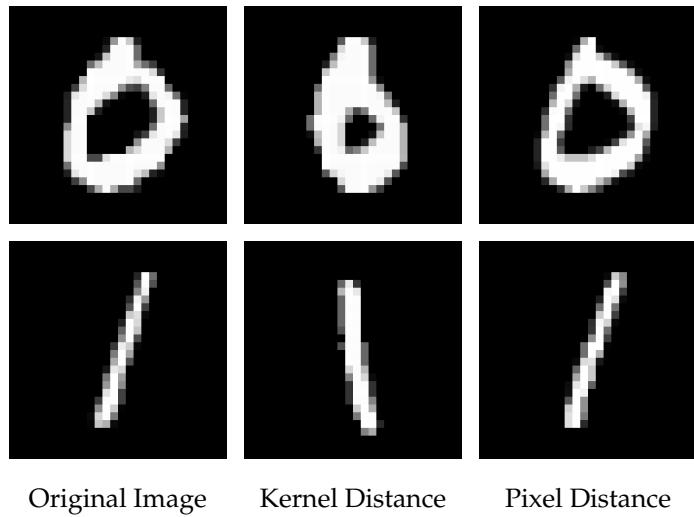


Figure 22: Comparison between the nearest samples in kernel space and pixel space. From left to right in each column: Test set point, nearest sample in kernel space, nearest sample in pixel space using euclidean distance.

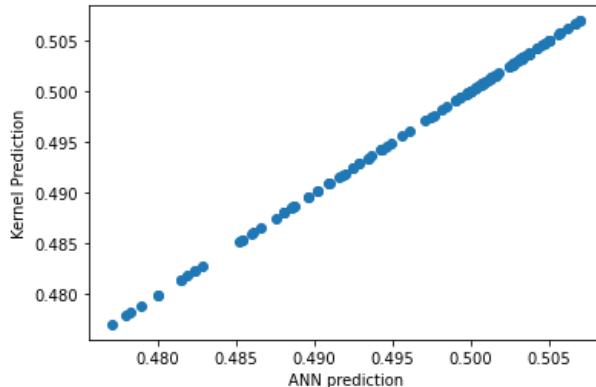


Figure 23: This plot shows output of the ANN versus output of the corresponding kernel representation for a set of test images from the MNIST dataset. We note the very strong agreement between the two outputs.

16 Discussion

The implications of a practical and finite kernel representation for neural networks are wide and profound. For most gradient trained models, there is a disconnect between the problem space (e.g. images) and the parameter space of a network. Parameters are intrinsically un-interpretable and much work has been spent building approximate mappings that convert model understanding back into the problem space in order to interpret features, sample importance, and other details ([?],

[?], and [?]). A kernel is composed of a direct mapping from the problem space into parameter space. This mapping allows much deeper understanding of gradient trained models because the internal state of the method has an exact representation mapped from the problem space. Sample importance is produced directly by looking at the kernel and its corresponding weights per training input.

As stated in previous work [?], this representation has strong implications about the structure of gradient trained models and how they can understand the problems that they solve. Since the kernel weights in this representation are fixed derivatives with respect to the loss function L , $a_{i,s} = -\varepsilon \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i}$ nearly all of the information used by the network is represented by the kernel mapping function and inner product. Inner products are not just measures of distance, they also measure angle. In fact, figure 24 shows that for a typical training example, the L_2 norm of the weights changes monotonically by only 20-30% during training. This means that the "learning" of a gradient trained model is dominated by change in angle, which is predicted for kernel methods in high dimensions [?].

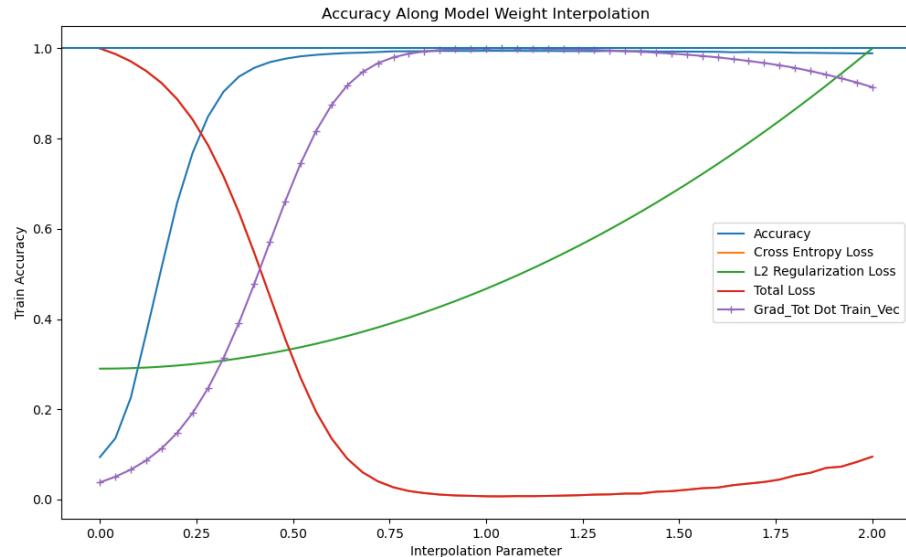


Figure 24: This plot shows a linear interpolation $w(t) = w_0 + t(w_1 - w_0)$ of model parameters w for a convolutional neural network \hat{y}_w from their starting random state w_0 to their ending trained state w_1 . The hatched blue line shows the dot product of the aggregated gradient over the training data X , $\langle \nabla_w \hat{y}_{w(t)}(X), (w_1 - w_0) / |w_1 - w_0| \rangle$. The other lines indicate accuracy (blue), total loss (red decreasing), and L2 Regularization (red increasing)

Perhaps the most significant advantage for gradient trained models of an exact kernel repre-

sentation is that the combination of kernel and kernel weights provides a spatial representation of the model’s understanding relative to the training data. In previous work ([?], [?]) it has been shown that image classification can be represented by projection onto the convex hull of training data. This projection is computationally infeasible, but it provides a geometric gold-standard classifier. Since kernel methods provide a spatial representation of their prediction, this representation can be directly compared with convex hull projections. It also provides data through which we can infer the gradient model’s understanding of the data spatially.

For kernel methods, this also represents a significant step. Despite their firm mathematical foundations, kernel methods have lost ground since the early 2000s due to the limitations of developing new kernels for complex high-dimensional problems [?]. This opens up many modern problems to the powerful tools available to kernel methods. Of these, Gaussian Processes (GPs) may be the most exciting. Given our kernel function, we can generate covariance matrices for GP which will allow direct uncertainty measurement. This will allow much more significant analysis for out-of-distribution samples including adversarial attacks ([?], [?]).

Gaussian processes are one of many research directions that naturally follow this work. It is worth noting that since the kernel from our representation can be either continuous or symmetric but not both (See remark 14), covariance matrices used in GPs will have slightly unusual properties that will reflect the divergence of the discrete training path from the smooth gradient flow. In the case of the asymmetric version of our representation, this asymmetry measures this divergence in a way that may be tractably explored using gaussian thermostats ([?], [?]). In addition to exploring GPs, it is natural to pursue more efficient computation of these representations by exploiting features of pytorch and incorporating the necessary integral computation with some codes that have been developed for NeuralODEs which require similar information ([?], [?])

Another implication from this representation is the increased importance of models following their gradient flow during training. Since this derived kernel is either discontinuous or asymmetric depending on the neural network’s training trajectory, developing training restrictions which satisfy equation 17 may produce more useful kernels and have implications about the accuracy and generalizability of ANN models. This will provide a new motivation for such research separate from just the question of efficiency of training. Approaches in this direction may be found in control theory ([?]) and the neural ODE approach ([?], [?]). Also in this vein is the precise formulation of

the divergence error from the discrete training path to the smooth gradient flow. Such a formulation should shed light on the dynamics of how such representations converge in performance under various step refinements.

17 Acknowledgements

This research was funded by Los Alamos National Lab LDRD-DR XX9C UQ4ML (help with how to acknowledge this LDRD funding juston?) Thanks to Yen Ting Lin, Philip Hoskins, Keenan Eikenberry, and Craig Thompson for feedback on early iterations of this paper.

ex

17.1 The DPK is a Kernel

Lemma 17.1. *The discrete path kernel (DPK) is a kernel.*

Proof. We must show that the associated kernel matrix $K_{\text{DPK}} \in \mathbb{R}^{n \times n}$ defined for an arbitrary subset of data $\{x_i\}_{i=1}^M \subset X$ as $K_{\text{DPK},i,j} = \int_0^1 \langle \phi_{s,t}(x_i), \phi_{s,t}(x_j) \rangle dt$ is both symmetric and positive semi-definite.

Since the inner product on a Hilbert space $\langle \cdot, \cdot \rangle$ is symmetric and since the same mapping φ is used on the left and right, K_{DPK} is **symmetric**.

To see that K_{DPK} is **Positive Semi-Definite**, let $f = (f_1, f_2, \dots, f_n)^\top \in \mathbb{R}^n$ be any vector. We need to show that $f^\top K_{\text{DPK}} f \geq 0$. We have

$$f^\top K_{DPK} f = \sum_{i=1}^n \sum_{j=1}^n f_i f_j \int_0^1 \langle \phi_{s,t}(x_i), \phi_{s,t}(x_j) \rangle dt \quad (20)$$

$$= \sum_{i=1}^n \sum_{j=1}^n f_i f_j \int_0^1 \langle \nabla_w \hat{y}_{w_s(t,x_i)}, \nabla_w \hat{y}_{w_s(t,x_j)} \rangle dt \quad (21)$$

$$= \int_0^1 \sum_{i=1}^n \sum_{j=1}^n f_i f_j \langle \nabla_w \hat{y}_{w_s(t,x_i)}, \nabla_w \hat{y}_{w_s(t,x_j)} \rangle dt \quad (22)$$

$$= \int_0^1 \sum_{i=1}^n \sum_{j=1}^n \langle f_i \nabla_w \hat{y}_{w_s(t,x_i)}, f_j \nabla_w \hat{y}_{w_s(t,x_j)} \rangle dt \quad (23)$$

$$= \int_0^1 \left\langle \sum_{i=1}^n f_i \nabla_w \hat{y}_{w_s(t,x_i)}, \sum_{j=1}^n f_j \nabla_w \hat{y}_{w_s(t,x_j)} \right\rangle dt \quad (24)$$

Re-ordering the sums so that their indices match, we have (25)

$$= \int_0^1 \left\| \sum_{i=1}^n f_i \nabla_w \hat{y}_{w_s(t,x_i)} \right\|^2 dt \quad (26)$$

$$\geq 0, \quad (27)$$

Note that this reordering does not depend on the continuity of our mapping function $\phi_{s,t}(x_i)$. \square

Remark In the case that our mapping function φ is not symmetric, after re-ordering, we still yield something of the form:

$$= \int_0^1 \left\| \sum_{i=1}^n f_i \nabla_w \hat{y}_{w_s(t,x_i)} \right\|^2 dt \quad (28)$$

$$(29)$$

The natural asymmetric φ is symmetric for every non-training point, so we can partition this sum. For the non-training points, we have symmetry, so for those points we yield exactly the L^2 metric. For the remaining points, if we can pick a Lipschitz constant E along the entire gradient field, then if training steps are enough, then the integral and the discrete step side of the asymmetric kernel will necessarily have positive inner product. In practice, this Lipschitz constant will change during training and for appropriately chosen step size (smaller early in training, larger later in training) we

can guarantee positive-definiteness. In particular this only needs to be checked for training points.

17.2 The DPK is an Exact Representation

Theorem 14.5 (Exact Kernel Representation). *A model \hat{y}_{w_N} trained using discrete steps matching the conditions of the discrete path kernel has the following exact kernel method representation:*

$$\hat{y}_{w_N}(x) = b + \sum_{i=1}^M \sum_{s=1}^N a_{i,s} K_{DPK}(x, x') \quad (15)$$

Proof. Let \hat{y}_w be a differentiable function parameterized by parameters w which is trained via N forward Euler steps of fixed step size ε on a training dataset X with labels Y , with initial parameters w_0 so that there is a constant b such that for every x , $\hat{y}_{w_0}(x) = b$, and weights at each step $w_s : 0 \leq s \leq N$. Let $x \in X$ be arbitrary and within the domain of \hat{y}_w for every w . For the final trained state of this model \hat{y}_{w_N} , let $y = \hat{y}_{w_N}(x)$.

For one step of training, we consider $y_s = \hat{y}_{w_s}(x)$ and $y_{s+1} = \hat{y}_{w_{s+1}}(x)$. We wish to account for the change $y_{s+1} - y_s$ in terms of a gradient flow, so we must compute $\frac{\partial \hat{y}}{\partial t}$ for a continuously varying parameter t . Since f is trained using forward Euler with a step size of $\varepsilon > 0$, this derivative is determined by a step of fixed size of the weights w_s to w_{s+1} . We will parameterize this step in terms of the weights:

$$\frac{\partial w_s(t)}{\partial t} = (w_{s+1} - w_s) \quad (30)$$

$$\int \frac{\partial w_s(t)}{\partial t} dt = \int (w_{s+1} - w_s) dt \quad (31)$$

$$w_s(t) = w_s + t(w_{s+1} - w_s) \quad (32)$$

$$(33)$$

Since f is being trained using forward Euler, we can write:

$$\frac{\partial w_s(t)}{\partial t} = -\varepsilon \nabla_w L(\hat{y}_{w_s}(x_i), y_i) = -\varepsilon \sum_{j=1}^d \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial w_j} \quad (34)$$

Applying chain rule and the above substitution, we can write

$$\frac{\partial \hat{y}}{\partial t} = \frac{d\hat{y}_{w_s(t)}}{dt} = \sum_{j=1}^d \frac{\partial \hat{y}}{\partial w_j} \frac{\partial w_j}{dt} \quad (35)$$

$$= \sum_{j=1}^d \frac{\partial \hat{y}_{w_s(t)}(x)}{\partial w_j} \left(-\varepsilon \frac{\partial L(\hat{y}_{w_s}(X_T), Y_T)}{\partial w_j} \right) \quad (36)$$

$$= \sum_{j=1}^d \frac{\partial \hat{y}_{w_s(t)}(x)}{\partial w_j} \left(-\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_{w_s}(x_i)}{\partial w_j} \right) \quad (37)$$

$$= -\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \sum_{j=1}^d \frac{\partial \hat{y}_{w_s(t)}(x)}{\partial w_j} \frac{\partial \hat{y}_{w_s}(x_i)}{\partial w_j} \quad (38)$$

$$= -\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \nabla_w \hat{y}_{w_s(t)}(x) \cdot \nabla_w \hat{y}_{w_s}(x_i) \quad (39)$$

$$(40)$$

Using the fundamental theorem of calculus, we can compute the change in the model's output over step s

$$y_{s+1} - y_s = \int_0^1 -\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \nabla_w \hat{y}_{w_s(t)}(x) \cdot \nabla_w \hat{y}_{w_s}(x_i) dt \quad (41)$$

$$= -\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \left(\int_0^1 \nabla_w \hat{y}_{w_s(t)}(x) dt \right) \cdot \nabla_w \hat{y}_{w_s}(x_i) \quad (42)$$

$$(43)$$

For all N training steps, we have

$$y_N = b + \sum_{s=1}^N y_{s+1} - y_s \quad (44)$$

$$y_N = b + \sum_{s=1}^N -\varepsilon \sum_{i=1}^M \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \left(\int_0^1 \nabla_w \hat{y}_{w_s(t)}(x) dt \right) \cdot \nabla_w \hat{y}_{w_s}(x_i) \quad (45)$$

$$= b + \sum_{i=1}^M \sum_{s=1}^N -\varepsilon \frac{\partial L(\hat{y}_{w_s}(x_i), y_i)}{\partial \hat{y}_i} \int_0^1 \langle \nabla_w \hat{y}_{w_s(t,x)}(x), \nabla_w \hat{y}_{w_s(t,x_i)}(x_i) \rangle dt \quad (46)$$

$$= b + \sum_{i=1}^M \sum_{s=1}^N a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (47)$$

Since an integral of a symmetric positive semi-definite function is still symmetric and positive-

definie and likewise for discrete sums, this representation is a kernel method.

□

17.3 When is an Ensemble of Kernel Machines itself a Kernel Machine?

Here we investigate when our derived ensemble of kernel machines composes to a single kernel machine. In order to show that a linear combination of kernels also equates to a kernel it is sufficient to show that $\text{sign}(a_{i,s}) = \text{sign}(a_{i,0})$ for all $a_{i,s}$. In this case it is possible to let the sample weights of our final kernel machine equal $\text{sign}(a_{i,0})$. In order to show this, we impose some structure on the loss function and network. Here we show this is the case for binary crossentropy on a network with sigmoid activations on the logits. (TODO: More argument here using mercer's theorem. All positive linear combinations of kernels are kernels. There are cases where some negative coefficients are allowed but that's going to take a lot more thought. How do we extend this to say $aKa > 0$ for all a ?)

Proof.

$$L(\hat{y}_i, y_i) = -y_i \ln(\hat{y}_i) - (1 - y_i) \ln(1 - \hat{y}_i) \quad (48)$$

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = \frac{y_i - \hat{y}_i}{(\hat{y}_i - 1)\hat{y}_i} \quad (49)$$

For a binary classification problem it is standard to have $y_i \in \{0, 1\}$ and using a sigmoid activation on the final layer we have $\hat{y}_i \in (0, 1)$.

Assume $y_i = 0$.

Assume $y_i = 1$.

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = \frac{0 - \hat{y}_i}{(\hat{y}_i - 1)\hat{y}_i} \quad (50)$$

$$= \frac{-1}{\hat{y}_i - 1} \quad (51)$$

$$= \frac{1}{|\hat{y}_i - 1|} \quad (52)$$

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = \frac{1 - \hat{y}_i}{(\hat{y}_i - 1)\hat{y}_i} \quad (54)$$

$$= \frac{1 - \hat{y}_i}{-(1 - \hat{y}_i)\hat{y}_i} \quad (55)$$

$$= -\frac{1}{\hat{y}_i} \quad (56)$$

The last equality relies on the fact that $\hat{y}_i < 1$. Because $\hat{y}_i > 0$.

$$y_i = 0 \implies \frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} > 0 \quad (53) \qquad y_i = 1 \implies \frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} < 0 \quad (57)$$

This shows that the sign of the gradient of the loss function depends only on the label y_i , not on the predicted value of our model \hat{y}_i and is constant through training. Therefore:

$$y_S = b - \varepsilon \sum_{i=1}^N \sum_{s=1}^S a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (58)$$

$$= b - \varepsilon \sum_{i=1}^N \text{sign}(a_{i,0}) \sum_{s=1}^S |a_{i,s}| \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (59)$$

This formulates a kernel machine where

$$a_{i,0} = \text{sign}\left(\frac{\partial L(\hat{y}_{w_0}(x_i), y_i)}{\partial \hat{y}_i}\right) \in \{-1, 1\} \quad (60)$$

$$K(x, x_i) = \sum_{s=1}^S |a_{i,s}| \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (61)$$

$$\phi_{s,t}(x) = \nabla_{w_s} \hat{y}_{w_s(t,x)}(x) \quad (62)$$

$$w_s(t, x) = \begin{cases} w_s, x \in X_T \\ w_s(t), x \notin X_T \end{cases} \quad (63)$$

$$b = 0 \quad (64)$$

□

This argument does not hold in the simple case of linear regression. Assume our loss is instead

squared error. Our labels are continuous on \mathbb{R} and our activation is the identity function.

$$L(\hat{y}_i, y_i) = (y_i - \hat{y}_{i,s})^2 \quad (65)$$

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = 2(y_i - \hat{y}_{i,s}) \quad (66)$$

This quantity is dependent on \hat{y}_i and its sign is changing throughout training. (TODO: Make this more formal and rigorous)

In order for

$$\sum_{s=1}^S a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (67)$$

to be a kernel on its own, we need it to be a positive (or negative) definite operator. In the specific case of our practical path kernel, i.e. that in $K(x, x')$ if x' happens to be equal to x_i , then:

$$= \sum_{s=1}^S 2(y_i - \hat{y}_{i,s}) \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \quad (68)$$

$$= \sum_{s=1}^S 2(y_i - \hat{y}_{i,s}) \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt \quad (69)$$

$$= \sum_{s=1}^S 2 \left(y_i \cdot \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt - \hat{y}_{i,s} \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt \right) \quad (70)$$

$$= \sum_{s=1}^S 2 \left(y_i \cdot \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt - \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \hat{y}_{i,s} \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt \right) \quad (71)$$

$$= \sum_{s=1}^S 2 \left(y_i \cdot \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \nabla_w \hat{y}_{w_s(0)}(x_i) \rangle dt - \int_0^1 \langle \nabla_w \hat{y}_{w_s(t)}(x), \frac{1}{2} \nabla_w (\hat{y}_{w_s(0)}(x_i))^2 \rangle dt \right) \quad (72)$$

$$(73)$$

Otherwise, we get the usual

$$= \sum_{s=1}^S 2(y_i - \hat{y}_{i,s}) \int_0^1 \langle \nabla_w \hat{y}_{w_s(t,x)}(x), \nabla_w \hat{y}_{w_s(t,x)}(x') \rangle dt \quad (74)$$

$$(75)$$

The question is two fold. One, in general theory (i.e. the lower example), can we contrive two pairs

(x_1, x'_1) and (x_2, x'_2) that don't necessarily need to be training or test images for which this sum is positive for 1 and negative for 2. Second, in the case that we are always comparing against training images, do we get something more predictable since there is greater dependence on x_i and we get the above way of re-writing using the gradient of the square of $\hat{y}(x_i)$.

17.4 Multi-Class Case

There are two ways of treating our loss function L for a number of classes (or number of output activations) K :

$$\text{Case 1: } L : \mathbb{R}^K \rightarrow \mathbb{R} \quad (76)$$

$$\text{Case 2: } L : \mathbb{R}^K \rightarrow \mathbb{R}^K \quad (77)$$

$$(78)$$

17.4.1 Case 1 Scalar Loss

Let $L : \mathbb{R}^K \rightarrow \mathbb{R}$. We will be using the chain rule $D(g \circ f)(x) = Dg(f(x))Df(x)$.

Let \hat{y} be a vector valued function so that $\hat{y} : \mathbb{R}^D \rightarrow \mathbb{R}^K$ satisfying the conditions from [representation theorem above] with $x \in \mathbb{R}^D$ and $y_i \in \mathbb{R}^K$ for every i . We note that $\frac{\partial \hat{y}}{\partial t}$ is a column and has shape $K \times 1$ and our first chain rule can be done the old fashioned way on each row of that

column:

$$\frac{\partial \hat{y}}{\partial t} = \sum_{j=1}^M \frac{\partial \hat{y}(x)}{\partial w_j} \frac{\partial w_j}{\partial t} \quad (79)$$

$$= -\varepsilon \sum_{j=1}^M \frac{\partial \hat{y}(x)}{\partial w_j} \sum_{i=1}^N \frac{\partial L(\hat{y}(x_i), y_i)}{\partial w_j} \quad (80)$$

Apply chain rule (81)

$$= -\varepsilon \sum_{j=1}^M \frac{\partial \hat{y}(x)}{\partial w_j} \sum_{i=1}^N \frac{\partial L(\hat{y}(x_i), y_i)}{\partial \hat{y}} \frac{\partial \hat{y}(x_i)}{\partial w_j} \quad (82)$$

Let (83)

$$A = \frac{\partial \hat{y}(x)}{\partial w_j} \in \mathbb{R}^{K \times 1} \quad (84)$$

$$B = \frac{\partial L(\hat{y}(x_i), y_i)}{\partial \hat{y}} \in \mathbb{R}^{1 \times K} \quad (85)$$

$$C = \frac{\partial \hat{y}(x_i)}{\partial w_j} \in \mathbb{R}^{K \times 1} \quad (86)$$

We have a matrix multiplication ABC and we wish to swap the order so somehow we can pull B out, leaving A and C to compose our product for the representation. Since $BC \in \mathbb{R}$, we have $(BC) = (BC)^T$ and we can write

$$(ABC)^T = (BC)^T A^T = BCA^T \quad (87)$$

$$ABC = (BCA^T)^T \quad (88)$$

Note: This condition needs to be checked carefully for other formulations so that we can re-order the product as follows:

$$= -\varepsilon \sum_{j=1}^M \sum_{i=1}^N \left(\frac{\partial L(\hat{y}(x_i), y_i)}{\partial \hat{y}} \frac{\partial \hat{y}(x_i)}{\partial w_j} \left(\frac{\partial \hat{y}(x)}{\partial w_j} \right)^T \right)^T \quad (89)$$

$$= -\varepsilon \sum_{i=1}^N \left(\frac{\partial L(\hat{y}(x_i), y_i)}{\partial \hat{y}} \sum_{j=1}^M \frac{\partial \hat{y}(x_i)}{\partial w_j} \left(\frac{\partial \hat{y}(x)}{\partial w_j} \right)^T \right)^T \quad (90)$$

(91)

Note, now that we are summing over j , so we can write this as an inner product on j with the ∇ operator which in this case is computing the jacobian of \hat{y} along the dimensions of class (index k) and weight (index j). We can define

$$(\nabla \hat{y}(x))_{k,j} = \frac{\partial \hat{y}_k(x)}{\partial w_j} \quad (92)$$

$$= -\varepsilon \sum_{i=1}^N \left(\frac{\partial L(\hat{y}(x_i), y_i)}{\partial \hat{y}} \nabla \hat{y}(x_i) (\nabla \hat{y}(x))^T \right)^T \quad (93)$$

(94)

We note that the dimensions of each of these matrices in order are $[1, K]$, $[K, M]$, and $[M, K]$ which will yield a matrix of dimension $[1, K]$ i.e. a row vector which we then transpose to get back a column of shape $[K, 1]$

17.5 Multi Class Case

In the case where \hat{y} is a vector we denote the data index by the superscript $y^{[i]}$ and the vector component by the subscript y_k .

query point may need to know about other class gradients than the target class.

k by k

yen ting's approach is needed because we are making some assumption about the model that allows us to measure the query change along a linear path.

The negative log likelihood function where y and \hat{y} are vectors.

$$NLL(y, \hat{y}) = \sum_k^K -y_k \ln(\hat{y}_k) \quad (95)$$

$$\frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial t} = \sum_{j=1}^M \frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial w_j} \frac{\partial w_j}{\partial t} \quad (96)$$

$$= \sum_{j=1}^M \frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial w_j} \left(-\varepsilon \sum_{i=1}^N \frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial w_j} \right) \quad (97)$$

$$= \sum_{j=1}^M \frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial w_j} \left(-\varepsilon \sum_{i=1}^N \frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial [\hat{y}_0^{[i]} \dots \hat{y}_k^{[i]}]} \frac{\partial [\hat{y}_0^{[i]} \dots \hat{y}_k^{[i]}]}{\partial w_j} \right) \quad (98)$$

$$= -\varepsilon \sum_{i=1}^N \frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial [\hat{y}_0^{[i]} \dots \hat{y}_k^{[i]}]} \sum_{j=1}^M \frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial w_j} \frac{\partial [\hat{y}_0^{[i]} \dots \hat{y}_k^{[i]}]}{\partial w_j} \quad (99)$$

Now we use the fact that L is CCE

$$\frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial y_k^{[i]}} = \begin{cases} 0, k \neq y^{[i]} \\ -1 \end{cases} \quad (100)$$

$$\frac{\partial \hat{y}_k}{\partial t} = -\varepsilon \sum_{i=1}^N 1 \sum_{j=1}^M \frac{\partial [\hat{y}_0 \dots \hat{y}_k]}{\partial w_j} \frac{\partial [\hat{y}_0^{[i]} \dots \hat{y}_k^{[i]}]}{\partial w_j} \quad (101)$$

$$y_0 + \left[\frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial y_0^{[i]}} \sum_{j=1}^M \frac{\partial f(x_i)_0}{\partial w_j} \cdot \frac{\partial f(x)_0}{\partial w_j}, \frac{\partial L(\hat{y}^{[i]}, y^{[i]})}{\partial y_1^{[i]}} \sum_{j=1}^M \frac{\partial f(x_i)_1}{\partial w_j} \cdot \frac{\partial f(x)_1}{\partial w_j} \right] \quad (102)$$

18 Neural Networks are Gaussian Processes

With Dropout on, the interpolant from one class to another will go into a variety of other classes. If you make a histogram of the locations where these boundary crossings occur, that will show a gaussian.

insert figure(s)

19 prove path kernel result in context of differential flow of gradients on neural network. using forward euler approx of grad flow.

20 ** look for sample in weight space and look for gradients

that are pointing toward the final point versus wanting a different direction. Then dot product those with the training direction.

21 training gradients are smooth

22 robust network types : regularized, michael's pca, Soft Nearest Neighbor Loss (SNNL) and adversarially trained.

23 high dimensional arcs are very similar to chords

24 linear interpolated model parameters from random to trained state yield robust models

24.1 For Mnist Inner Products in weight space matter more than distances

– does this generalize to ImNet?

25 define robustness in terms of skew versus orthogonal

26 define robustness in terms of attack perturbation magnitude

Part III

Decision Boundaries

27 decision boundary definitions

28 The Argmax Function

A central issue when writing classifiers is mapping from continuous outputs or probabilities to discrete sets of classes. Frequently argmax type functions are used to accomplish this mapping. To discuss decision boundaries, we must precisely define argmax and some of its properties.

In practice, argmax is not strictly a function, but rather a mapping from the set of outputs or activations from another model into the power set of a discrete set of classes:

$$\text{argmax} : \mathbb{R}^k \rightarrow \mathcal{P}(C) \quad (103)$$

Defined this way, we cannot necessarily consider arg max to be a function in general as the singleton outputs of argmax overlap in an undefined way with other sets from the power set. However, if we restrict our domain carefully, we can identify certain properties.

Restricting to only the pre-image of the singletons, it should be clear that argmax is continuous.

Indeed, restricted to the pre-image of any set in the power-set, argmax is continuous.

Further, we can directly prove that the pre-image of an individual singleton is open. Observe that for any point whose image is a singleton, one element of the domain vector must exceed the others by $\epsilon > 0$. We shall use the ℓ^1 metric for distance, and thus if we restrict ourselves to a ball of radius ϵ , then all elements inside this ball will have that element still larger than the rest and thus map to the same singleton under argmax. Since the union of infinitely many open sets is open

in \mathbb{R}^k , the union of all singleton pre-images is an open set. Conveniently this also provides proof that the union of all of the non-singleton sets in $\mathcal{P}(C)$ is a closed set. We will call this closed set the argmax Decision Boundary.

Note : there are ways argmax can be forced to break ties, i.e. by ordering the

Questions:

Is the decision boundary connected

29 Defining Decision Boundaries

29.1 Complement Definition

A point x is in the *decision interior* D'_f for a classifier $f : \mathbb{R}^N -> \mathcal{C}$ if there exists $\delta > 0$ such that $\forall \epsilon < \delta, |f(B_\epsilon(x))| = 1$.

The *decision boundary* of a classifier f is the closure of the complement of the decision interior $\overline{\{x : x \notin D'_f\}}$.

29.2 Constructive Definition

A point x is on the *decision boundary* D of a classifier f if $\forall \epsilon > 0, |f(B_\epsilon(x))| \geq 2$.

A point x is a *binary decision point* if $\exists \delta > 0$ such that $\forall \epsilon > 0$ if $\epsilon < \delta$, then $|f(B_\epsilon(x))| = 2$.

A point x is on the *K-decision boundary* D^K of a classifier f if $\exists \delta > 0$ such that $\forall \epsilon > 0$ if $\epsilon < \delta$, then $|f(B_\epsilon(x))| = K$.

29.3 Level Set Definition

The decision boundary D of a probability valued function F is a union of all level sets $L_{c_1, c_2, a}$ defined by two classes c_1 and c_2 and a constant a which satisfy two properties: First, given $x \in L_{c_1, c_2, a}$, $F(x)_{c_1} = F(x)_{c_2} = a$ where a is a constant also defining the level set. Second, for all $c \notin \{c_1, c_2\}$, we have $a > F(x)_c$.

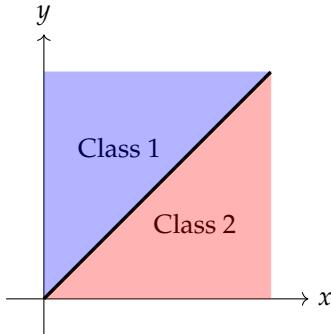
29.4 Images of Decision Boundaries

We can consider a classification workflow f from a data space X (e.g. \mathbb{R}^N) using a model F mapping the data space X to a probability space Y (e.g. $[0, 1]^K$ where K is the number of classes) and using argmax to convert continuous representations in Y to discrete classes in the set of classes \mathcal{C} .

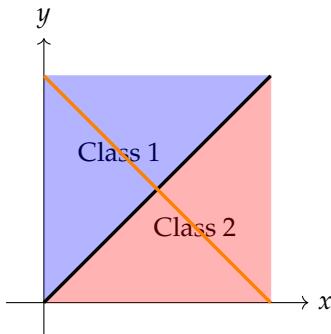
$$X \xrightarrow{F} Y \xrightarrow{\text{argmax}} \mathcal{C} \quad (104)$$

In this way we define a usual discrete classifier $f = F \circ \text{argmax}$

The decision boundary D is defined on X , however we may look at the image of the decision boundary under F . For example, if there are only two classes, then $Y = [0, 1]^2$ and the decision boundary can be nicely visualized by the black line below:



Furthermore, if the output of F are *probabilities* which add to one, then all points of x will map to the orange line:



We note that the point $(0.5, 0.5)$ is therefore the only point on the decision boundary for probability valued F . We may generalize to higher dimensions where all probability valued models

F will map into the plane $x + y + z + \dots = 1$ in Y and the decision boundary will be partitioned into $K - 1$ components, where the K -decision boundary is the intersection of this plane with the *centroid* line $x = y = z = \dots$ and the 2-decision boundaries become planes intersecting at the same line.

30 Properties of Decision Boundaries

In practice, decision boundaries have a variety of properties. The decision boundary is a pre-image under an onto function that is not 1-1, which allows decision boundaries to vary in up to as many dimensions as the data space. In practice, this will likely be many fewer. In particular, decision boundaries should correspond with the structure of the data. If the data is embedded within a k -manifold within the input space, the decision boundary can be thought of on the dual space of that manifold.

The dual space of a k -manifold embedded in an n -dimensional Euclidean space is the space of differential k -forms on the manifold. More precisely, for a k -manifold M embedded in an n -dimensional Euclidean space, the dual space is defined as the space of covariant k -tensors on M , denoted by $\Lambda^k(M)$, which consists of all linear maps from the tangent space of M at each point to the real numbers.

In particular, when M is a smooth k -dimensional manifold, $\Lambda^k(M)$ is the space of smooth k -forms on M , which can be locally expressed as linear combinations of differential forms of the form $dx_I = dx_{i_1} \wedge dx_{i_2} \wedge \dots \wedge dx_{i_k}$, where the indices $i_1 < i_2 < \dots < i_k$ denote the coordinates of a k -dimensional chart on M and dx_i denotes the differential of the i -th coordinate function.

Intuitively, a k -form on M assigns to each k -dimensional oriented subspace of the tangent space at a point a signed volume, and the space of all such k -forms forms the dual space of M . This dual space is important in various areas of mathematics and physics, including differential geometry, topology, and gauge theory.

30.1 Delaunay Decision Boundaries

The simplest special case within this dual space is a classifier which is defined using the Voronoi diagram for the dataset. We can use the voronoi diagram to define a classifier for an

arbitrary query point by simply assigning it the class of the training point whose voronoi cell contains it. This has several useful properties, though one in particular stands out.

Given two points x and y in the training set X which define a delaunay triangulation and its corresponding voronoi diagram (which is the dual of the delaunay triangulation) we can see that a line connecting x to y must orthogonally cross the decision boundary – which is exactly the voronoi diagram of this dataset.

Proof:

Proof. Let x and y be two points in a dataset X that share an edge in the Delaunay triangulation for X , and let e be the edge they share in the Delaunay graph. Suppose that e is not orthogonal to the face of the Voronoi diagram that x and y share. Then there exists a point z in the interior of the Voronoi cell for x and y such that the line through z and the midpoint of e intersects the face of the Voronoi diagram at a point p .

Since z is in the Voronoi cell for x and y , it follows that $d(z, x) < d(z, y)$ (where d is the Euclidean distance). But this contradicts the fact that x and y share an edge in the Delaunay triangulation, which implies that the circumcircle of the triangle formed by x , y , and z does not contain any other points in X . In particular, this means that the circumcenter of this triangle lies on the perpendicular bisector of e , which implies that e is orthogonal to the face of the Voronoi diagram that x and y share.

Therefore, we have shown that if x and y share an edge in the Delaunay triangulation for X , then the edge they share in the Delaunay graph must be orthogonal to the face of the Voronoi diagram that they share. \square

It is immediately clear that for points not in the training set, the line connecting them with eachother or with a point in the training set need not cross orthogonal to the voronoi cell boundaries. In order to talk clearly about such non-orthogonal crossings, we will define

Definition 30.1. *In arbitrary dimensional space, the angle of incidence between a line and a plane is defined as the angle between the line and its projection onto the plane, measured in the plane.*

More precisely, let \mathbb{R}^n be an n -dimensional Euclidean space, let ℓ be a line in \mathbb{R}^n with direction vector \vec{v} , and let P be a plane in \mathbb{R}^n with normal vector \vec{n} . Suppose that ℓ intersects P at a point Q . Let H be the orthogonal projection of ℓ onto P , and let θ be the angle between ℓ and H .

Then, the angle of incidence between ℓ and P is defined as $\alpha = \frac{\pi}{2} - \theta$, where π is the constant representing the ratio of the circumference of a circle to its diameter.

Note that when $n = 3$, this definition reduces to the familiar definition of the angle of incidence between a line and a plane in three-dimensional space. However, the definition is valid for arbitrary dimensional space.

and

Definition 30.2. The

30.2 Level Set Definition

Definition 30.3. The decision boundary D of a probability valued function F is a union of all level sets $L_{c_1, c_2, a}$ defined by two classes c_1 and c_2 and a constant a which satisfy two properties: First, given $x \in L_{c_1, c_2, a}$, $F(x)_{c_1} = F(x)_{c_2} = a$ where a is a constant also defining the level set. Second, for all $c \notin \{c_1, c_2\}$, we have $a > F(x)_c$. This is the decision level set for level a . For a given point x on the decision boundary of a function, the decision level is the value a defining the level set of the decision boundary of which x is a member.

Note: Any point x can be in at most one level set of the decision boundary.

Definition 30.4. The dimension of the decision boundary at a point x is the number of dimensions needed to span $\{z : (x + z) \text{ is in the decision level } a \text{ of the level set containing } x\}$

A primary property of interest in adversarial attacks is “sharpness” i.e. the property that a partition of a set which is attributed to one class might stab needle-like into a partition of another set. We will investigate this “sharpness” concept by examining both the incident angle (the acuteness of angles in the decision boundary) and the dimension of the decision boundary, since both properties behave like one another.

There is one conditional bound which is specifically for voronoi cells which are not exterior to the voronoi diagram. Such cells are bounded. The line connecting a point within a bounded cell to the point defining any adjacent voronoi cell.

30.3 Weighted Delaunay Decision Boundaries

This simple voronoi classifier has the limitation that it only cares about which individual training datapoint is closest to the query point, but is not sensitive to the distribution of this data. If

we suppose that this data is distributed on a manifold with significant co-dimension with respect to the ambient data space, then it would be best to use a metric based on this manifold, so that distance among the data, and hence the delaunay triangulation and the comensurate voronoi cells would be determined by this metric and thus properly sensitive to the underlying geometry of this distribution. Alas, this is akin to having the generating function for nature. It cannot be computed. So instead, we can take advantage of the Delaunay triangulation with the usual euclidean metric as an approximation of the lower dimensional manifold in this case.

This method is limited by the quality of the training dataset as a sampling of the distribution in question. The Delaunay triangulation is very sensitive to outliers from a distribution, meaning that our samples must have relatively little noise in order to be treated as appropriate samples from the low-dimensional manifold. Second, Delaunay triangulations are unstable in high-dimensions where very small triangles are more likely. Both of these problems are lessened by using data which are sufficiently plentiful relative to the dimension of the lower dimensional manifold and by requiring that they do not have more than a small amount of noise to perturb them from the low dimensional manifold.

Suppose the lower dimensional manifold M could be defined, and the training dataset X could be projected onto M . The decision boundary partitioning M would be a set of slices orthogonal to M which would partition any training dataset X into its appropriate parts. In particular, for a sufficiently dense sampling x , we could define this decision boundary by examining the data points immediately on either side the decision boundary wherever they meet. We can define an approximation of the decision boundary as the orthogonal partition that we expect to be equidistant between adjacent points of different classes for all samplings of training points. In fact, we can define this boundary in the limit using delaunay triangulation:

Definition 30.5. *Let M be a low dimensional manifold and let X_ε be a sampling of points on M for which the maximal edge length along its Delaunay triangulation D_ε is ε . Given a query point z , the ε -Delaunay weight for class i , $w(z)_i$, is the sum of the barycentric coordinates of z relative to the vertices with label i of the simplex of D_ε which contains z . The weight vector ε boundary level set for factor a , $B_{\varepsilon,a}$ is the level set $\{z \in M : \text{for all } i, \text{we have } w(z)_i \leq a \text{ and for some } i, j, w(z)_i = w(z)_j = a\}$. The union of all such level sets over a is the ε -Delaunay-Boundary B_ε .*

Remark, we can extend the decision boundary outside of M by simply extending the Delaunay-

Boundary using the usual euclidean metric.

Theorem 30.6. *Given a low dimensional manifold M ,*

$$\lim_{\varepsilon \rightarrow 0} B_\varepsilon(M) = B(M) \quad (105)$$

$B(M)$ is the decision boundary restricted to M .

In this way, a particular training set X defines an approximation of the general Delaunay Decision Boundary which is well defined outside of M . If the decision boundary is smooth, then for a sufficiently dense training set X which is sufficiently close to M , this approximation can be bounded by ε and a lipschitz constant coming from the mean value theorem on the decision boundary.

It is this approximate form of the decision boundary which we may in practice evaluate.

30.4 Orthant and Wedge

In order to study decision boundaries, we must talk about properties related to how decision boundaries interact. We desire a test-object whose dimensionality and acuteness of angle we can control. To this end we will use the k -dimensional orthant:

Definition 30.7. *Given an integer k , the k -orthant in n dimensions is the set $\{x \in \mathbb{R}^n \text{ such that for every natural number } i \leq k, x_i > 0\}$*

This concept for an orthant can be extended to be acute in arbitrarily many angles by tipping toward x to form a wedge.

Definition 30.8. *Given integer k and a set of $k - 1$ angles Θ , the $k - \Theta$ -wedge in n dimensions is the set $\{x \in \mathbb{R}^n \text{ such that } x_1 > 0 \text{ and for every natural number } 1 < i \leq k, x_i > x_1 \tan(\theta_{i-1})\}$.*

These objects allow us to control the dimensionality and acuteness of a decision boundary to test decision boundary metrics. Another recipe for analysis is standardizing the paths along which we will cross decision boundaries. We will use two general paths relative to the wedge for comparison: centerline and x -parallel:

Definition 30.9. *The centerline $k - \Theta$ -wedge crossing is defined along the line $x_1 = t$, for $1 < i \leq k$ $x_i = t \tan(\theta_{i-1}/2)$, and for $i > k$ $x_i = 0$.*

The $x - \varepsilon$ -parallel $k - \Theta$ -wedge crossing is defined along the line $x_1 = t$, for $i > 1$ $x_i = \varepsilon$

Although we can easily determine whether a point is inside or outside the wedge simply by checking whether it satisfies the conditions defining the wedge, we also wish to compute the projection or nearest point on the wedge for an arbitrary query point. This involves solving for the closest point to a convex object, which can be accomplished by projection onto convex sets.

Computing this projection requires we determine which inequalities have been violated. Each inequality corresponds with a hyper-plane.

Definition 30.10. *Given a finite set of half-planes Y_i in \mathbb{R}^n and a point x , let I contain all i such that $x \notin Y_i$. Then since the intersection of any number of convex sets is convex, $Y_x = \cap_{i \in I} Y_i$ is a convex set. Furthermore, there is exactly one point z with $|x - z| \leq |x - y|$ for every y in Y_x . This is the projection of x onto Y_x .*

In order to compute this point in practice, we must first determine all half-planes that do not contain x and then perform orthogonal projection in sequence along these half-planes using Projection onto Convex Sets / cite vonneuman-pocs.

Lemma 30.11. *If all half-planes Y_i with corresponding normal vector v_i have boundaries $B(Y_i)$ which are mutually orthogonal, i.e. if $\langle v_i, v_j \rangle = 0$ for every combination of i and j , then the nearest point can be solved by iterative projection*

$$\text{proj}_{v_{i_1}}(\text{proj}_{v_{i_2}}(\text{proj}_{v_{i_3}} \dots (\mathbf{x}))) \quad (106)$$

([?])

In the case of the upper wedge, the boundaries are not mutually orthogonal, so simple orthogonal projection is not sufficient. We must explore this projection in more detail. Really, what we wish to do is project the query point x onto the intersection of all of its violated boundaries. Iteratively, we can orthogonally project onto the first hyperplane. We will conduct our projection by subtracting a projection onto a normal vector for each hyperplane. Given a point x , a hyperplane p_i

and its normal vector v_i , we define

$$\text{proj}_{\mathbf{p}_i}(\mathbf{x}) = \mathbf{x} - \mathbf{x} \cdot v_i \quad (107)$$

The normal vectors for our wedge come in two forms. For the orthant, all of the normals are of the form $v_i = e_i$. For the angled wedge planes, they are all of the form $w_i = e_i \cos(\theta_i) - e_1 \sin(\theta_i)$ where $i > 1$. We immediately note that $v_i \cdot v_j = 0$ for every natural number i and j less than the number of planes k . However,

$$w_i \cdot w_j = \langle e_i \cos(\theta_i) - e_1 \sin(\theta_i), e_j \cos(\theta_j) - e_1 \sin(\theta_j) \rangle \quad (108)$$

$$= e_i e_j \cos(\theta_i) \cos(\theta_j) - e_1 e_j \cos(\theta_j) \sin(\theta_i) - e_1 e_i \cos(\theta_i) \sin(\theta_j) + e_1^2 \sin(\theta_i) \sin(\theta_j) \quad (109)$$

$$= e_1 \sin(\theta_i) \sin(\theta_j) \quad (110)$$

$$(111)$$

This means that none of the w_j are orthogonal to each-other and furthermore

$$v_1 \cdot w_i = \langle e_1, -e_1 \sin(\theta_i) + e_i \cos(\theta_i) \rangle \quad (112)$$

$$= -\sin(\theta_i) \quad (113)$$

We will leave this latter lack of orthogonality for last, and start by determining how to sequentially orthogonally project onto the w_j . Critically, we need only project onto the intersection of each w_i with all previously projected w_j . We begin by projecting from within the plane defined by w_i onto the plane defined by w_j . These vectors are not orthogonal, so we must derive a new plane and its normal vector w_{ij} which is still orthogonal to w_i but shares the same intersection with the plane defined by w_j along the same edge. The conditions that must be satisfied are:

$$w_{ij} \cdot w_j = 0 \quad (114)$$

$$w_{ij} \perp \cap(w_i, w_j) \quad (115)$$

$$w_{ij} \cdot (e_1 + e_i \tan \theta_i + e_j \tan \theta_j) = 0 \quad (116)$$

This third condition is the requirement that w_{ij} is orthogonal to all vectors in the intersection of planes i and j .

We'll attempt to solve these equations by using a variable a to account for the discrepancy needed to solve these equations.

$$w_{ij} = w_j + a = e_j \cos \theta_j - e_1 \sin \theta_j + a \quad (117)$$

$$a = a_1 e_1 + a_2 e_2 + \cdots + a_n e_n \quad (118)$$

$$w_{ij} \cdot w_i = \sin \theta_j \sin \theta_i - a_1 \sin \theta_j + a_i \cos \theta_i = 0 \quad (119)$$

$$w_{ij} \cdot (e_1 + e_i \tan \theta_i + e_j \tan \theta_j) = a_1 - a_i \tan \theta_i + a_j \tan \theta_j \quad (120)$$

Letting $a_1 \neq 0$ requires that $a = e_1 \sin \theta_j - e_j \cos \theta_j$ which is a degenerate case, so let $a_1 = 0$. For every a_k not involved in the intersection of planes, they will have zero dot product, so we can let them all be 0. Then

$$-a_i \cos \theta_i = \sin \theta_j \sin \theta_i \quad (121)$$

$$a_i = -\sin \theta_j \tan \theta_i \quad (122)$$

$$a_j \tan \theta_j = \sin \theta_j \tan^2 \theta_i \quad (123)$$

$$a_j = \cos \theta_j \tan^2 \theta_i \quad (124)$$

$$w_{ij} = w_j - e_i \sin \theta_j \tan \theta_i + e_j \cos \theta_j \tan^2 \theta_i \quad (125)$$

Projecting once more onto w_k which also shares an intersection with w_i and w_j , we gain one parameter and one degree of freedom.

$$w_{ijk} = w_k + a = e_k \cos \theta_k - e_1 \sin \theta_k + a \quad (126)$$

$$a = a_1 e_1 + a_2 e_2 + \cdots + a_n e_n \quad (127)$$

$$w_{ijk} \cdot w_i = \sin \theta_k \sin \theta_i - a_1 \sin \theta_k + a_i \cos \theta_i = 0 \quad (128)$$

$$w_{ijk} \cdot w_j = \sin \theta_k \sin \theta_j - a_1 \sin \theta_j + a_j \cos \theta_j = 0 \quad (129)$$

$$w_{ijk} \cdot (e_1 + e_i \tan \theta_i + e_j \tan \theta_j) = a_1 + a_i \tan \theta_i + a_j \tan \theta_j + a_k \tan \theta_k \quad (130)$$

Now, again letting $a_1 = 0$ to avoid a degenerate case and letting all other components of a be zero, we have:

$$-a_i \cos \theta_i = \sin \theta_k \sin \theta_i \quad (131)$$

$$a_i = -\sin \theta_k \tan \theta_i \quad (132)$$

$$-a_j \cos \theta_j = \sin \theta_k \sin \theta_j \quad (133)$$

$$a_j = -\sin \theta_k \tan \theta_j \quad (134)$$

$$a_k \tan \theta_k = \sin \theta_k \tan^2 \theta_i + \sin \theta_k \tan^2 \theta_j \quad (135)$$

$$a_k = \cos \theta_k (\tan^2 \theta_i + \tan^2 \theta_j) \quad (136)$$

$$w_{ij} = w_j - e_i \sin \theta_k \tan \theta_i \quad (137)$$

$$-e_j \sin \theta_k \tan \theta_j \quad (138)$$

$$+ e_k \cos \theta_k (\tan^2 \theta_i + \tan^2 \theta_j) \quad (139)$$

We see now that we can project an arbitrary sequence of such intersecting surfaces with each $w_{i..k}$ with $a_i = -\sin \theta_k \tan \theta_i$ for i less than k and $a_k = \cos \theta_k (\sum_{i=1}^k \tan^2 \theta_i)$. Now for our projection algorithm, we first determine if all planes are satisfied ($x \cdot v_i \geq 0$ and $x \cdot w_i \leq 0$ for every i .) If they are, then we will project onto each plane and pick the nearest point. If they are not all satisfied, now we'll check each pair of planes w_i and v_i . If both are satisfied, we will skip them, if both are violated, we will set the coordinate in their native direction e_i to 0. When only one of the two planes for all compute all planes that are violated by the position of each query point. For each pair of related planes w_i and v_i , there are three cases. Both satisfied, Both violated, or either or. In the case that both are violated, i.e. $x \cdot v_i < 0$, then $x_1 < 0$, which is to say the nearest point within the wedge in these coordinates will be on the origin. When both conditions are satisfied, we will ignore this plane except

Now, performing orthogonal projection, we can orthogonally project in sequence for all half-planes not containing x except for v_1 since e_1 is not orthogonal to any of the wedge planes. Once we have projected onto all other violated planes, if the projection onto e_1 is negative, this will mean that the nearest point to x is the origin. For $i \neq 1$, all these projections are of the first form. $x - \sum_i x \cdot e_i$. The first projection of the second form can be done by simple orthogonal projection, however in

order to perform the next projection, we must project onto the intersection of the two half-planes without leaving the first plane. We'll need a new plane which is orthogonal to the first plane, but has the same intersection with the second plane. We can compute what is missing to make this projection orthogonal. We have

$$w_i \cdot w_j = \langle e_i \sin(\theta_i) - e_1 \cos(\theta_i), e_j \sin(\theta_j) - e_1 \cos(\theta_j) \rangle \quad (140)$$

$$= e_i e_j \sin(\theta_i) \sin(\theta_j) - e_1 e_j \sin(\theta_j) \cos(\theta_j) - e_1 e_i \sin(\theta_i) \cos(\theta_j) + e_1^2 \cos(\theta_j) \cos(\theta_i) \quad (141)$$

$$= e_1 \cos(\theta_j) \cos(\theta_i) \quad (142)$$

$$(143)$$

We can solve for a perturbation in the e_1 direction which will make these two normal vectors orthogonal but due to preserving

31 exploring boundary curvature with Random Walks

To analyze decision boundary curvature, we will project samples of points onto the decision boundary and then use Singular Value Decomposition to analyze the *projected points*. In general, this process will involve first selecting two images x_1 and x_2 for which $C(x_1) \neq C(x_2)$. A point x_b for which $C(x_b)$ is ambiguous between $C(x_1)$ and $C(x_2)$. The resulting sample X will be projected to the decision boundary by either computing a loss function that is minimized when each of the classes $C(X)$ flip from $C(x_1)$ to $C(x_2)$ or vice versa respectively and performing gradient descent, or by interpolating to the parent point of opposite class (so for $x \in X$, if $C(x) = C(x_1)$, we will interpolate from x to x_2).

Once a projection has been found, we will take the singular value decomposition (SVD) of this sample and examine it for degeneracy.

Initial comparison of SVDs of decision boundary points yields a single dominant singular value which corresponds with the content of the original image on the boundary. All random noise appears to be mostly orthogonal to this.

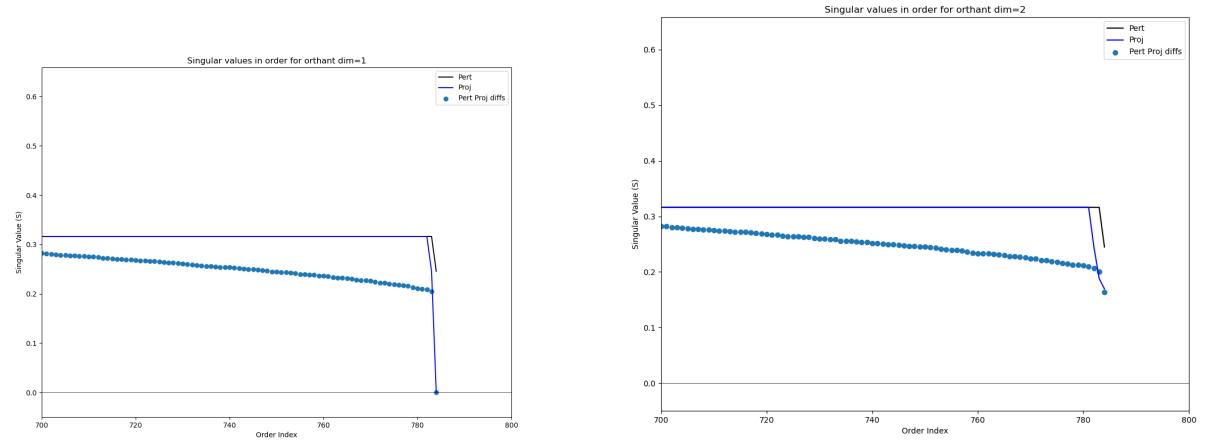
Once this vector is removed, the remaining signal is simply the adversarial attack information.

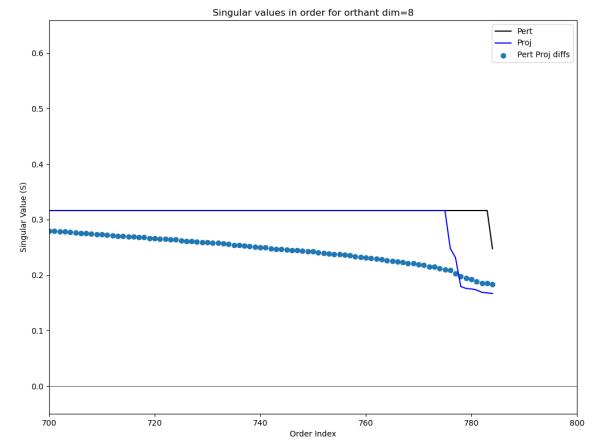
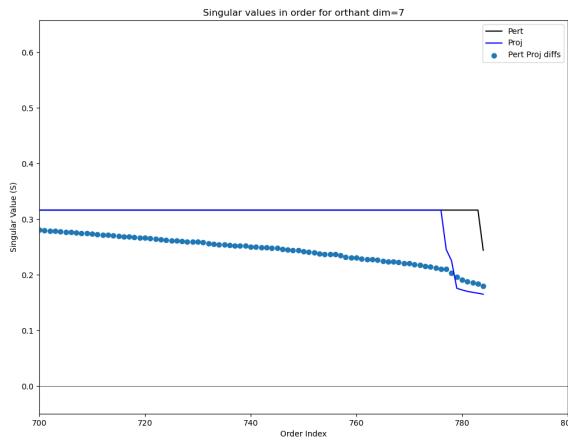
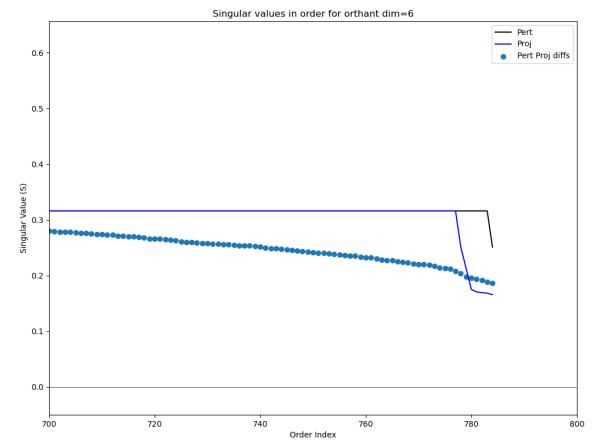
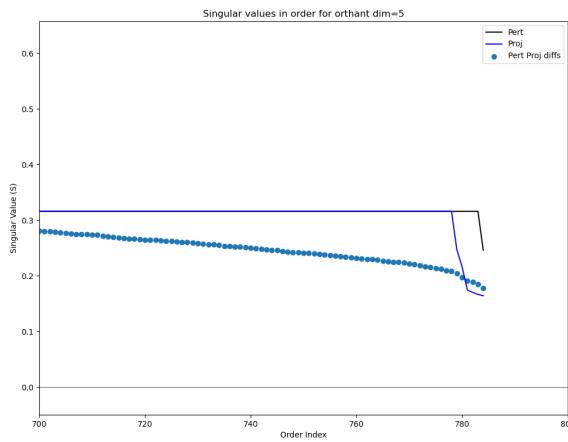
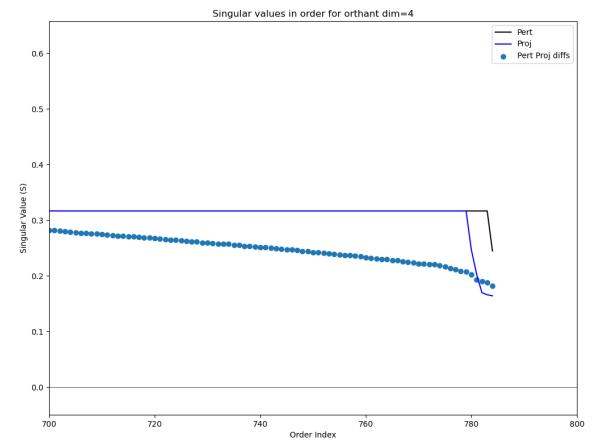
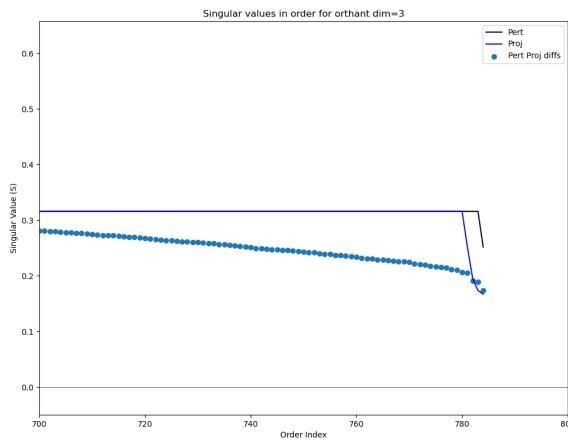
To get a set of singular vectors which do not emphasize the image content, we take random

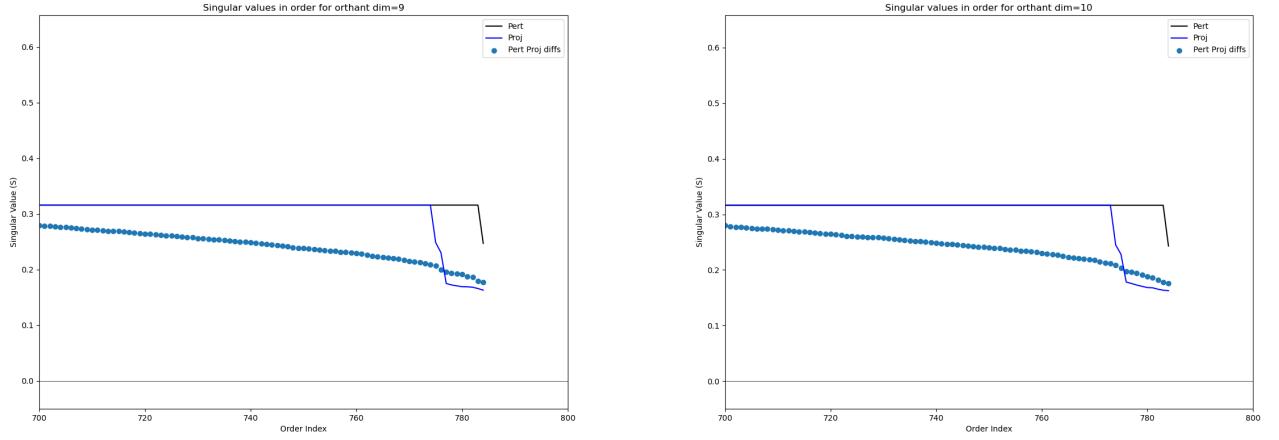
differences among the sampled images and take the SVD of those random differences.

In these first 10 images, this procedure is carried out on the orthant, where samples are generated with mean 0 and are projected onto orthants with increasing numbers of dimensions. We can see a very clear dropped set of singular values smaller than the rest, which indicate the number of degenerate dimensions. In each of these cases, the dropped singular values match the dimension of the orthant.

Experiment : A valid experiment here is to measure the difference between the images and the projection to get – in this case – normal vectors to the orthant for each image sampled. The SVD of these normals can be computed to determine the number of dimensions in the projecetion operation. This same procedure can be caried out later in the real practical image projections, although in that case orthogonality is not guaranteed.



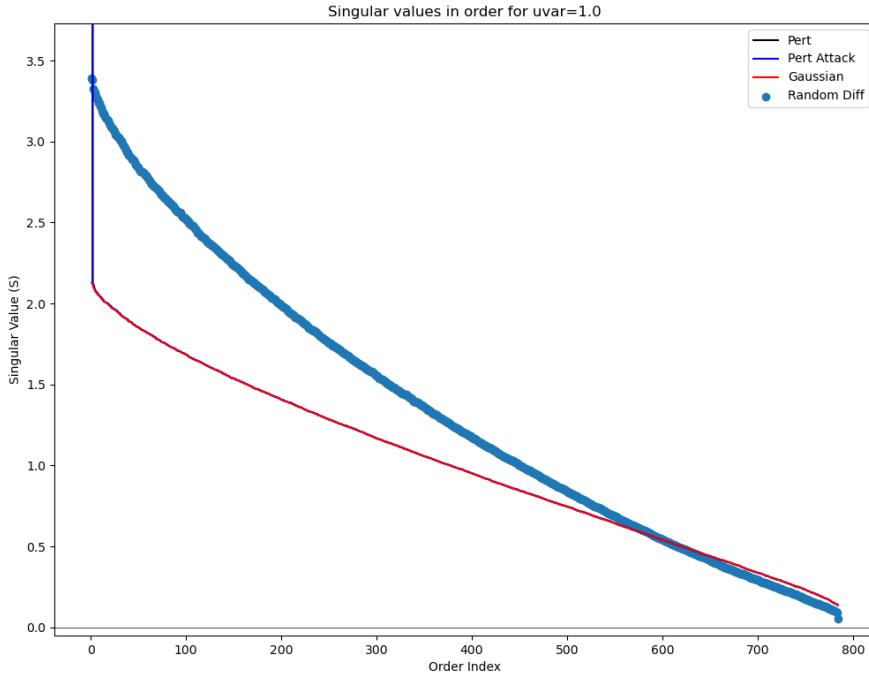




There is a question of how best to measure "normal" vectors for each projection. The most direct computation is to take a sample with small variance around each point on the decision boundary and solve least squares of each of these samples. We have previously observed that *most* of the decision boundary is locally planar.

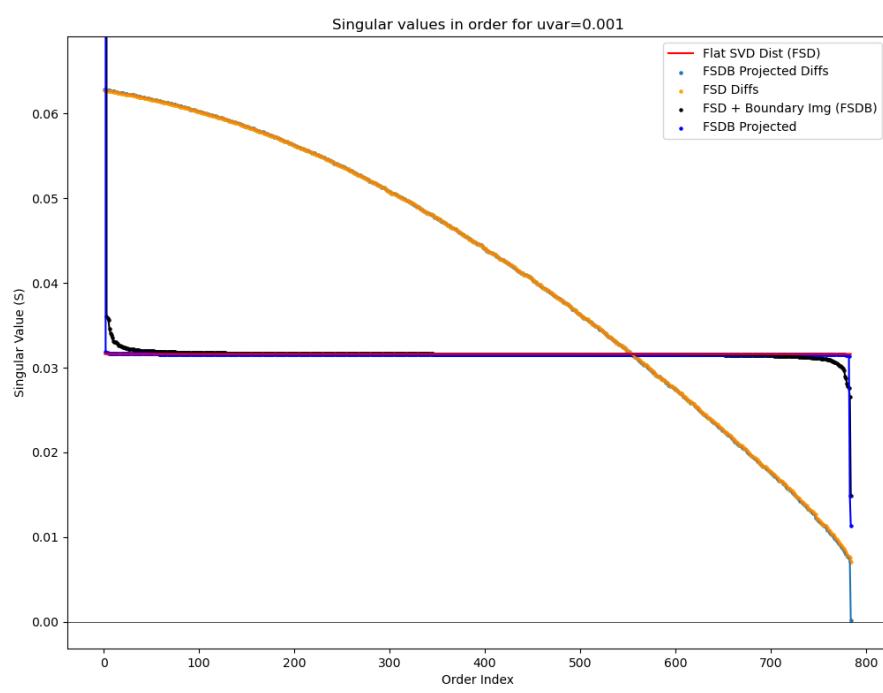
It remains to compare these computed normals with other known quantities, e.g. the gradient computed with respect to adversarial loss functions and the difference between each sampled image and its resultant projection. In addition, it remains to compare the normals of multiple nearby images to determine at what radius of sample the decision boundary begins to show curvature.

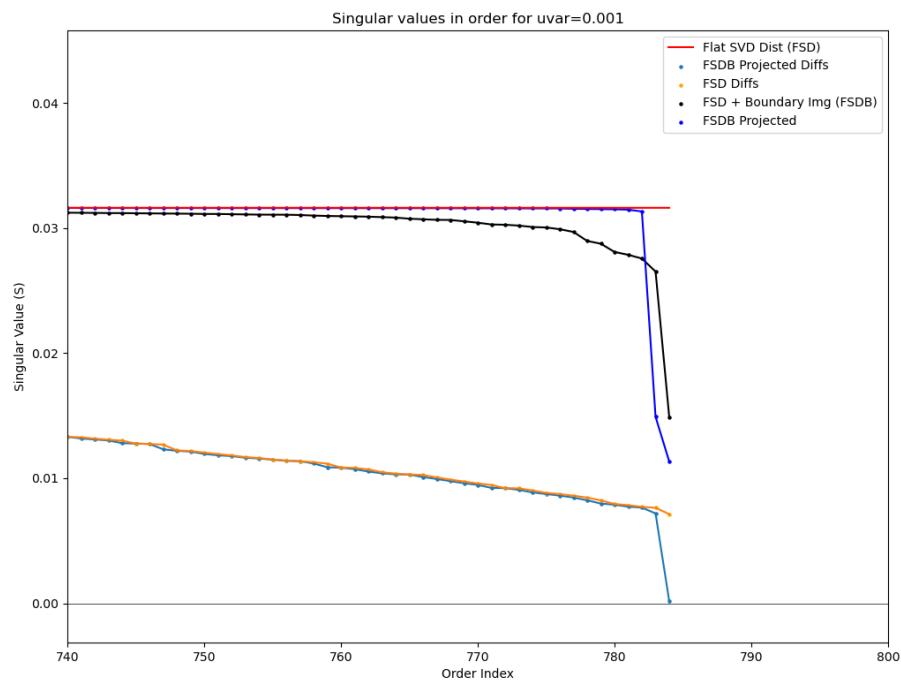
The following image shows singular values in order for a sample around a decision boundary image in MNIST generated by interpolating from a test image to an adversarial image generated from it. This plot is dominated by the natural distribution of singular values for a gaussian and also by the original image information which can be seen as a huge singular value on the far left. We will address both of these factors in following plots.



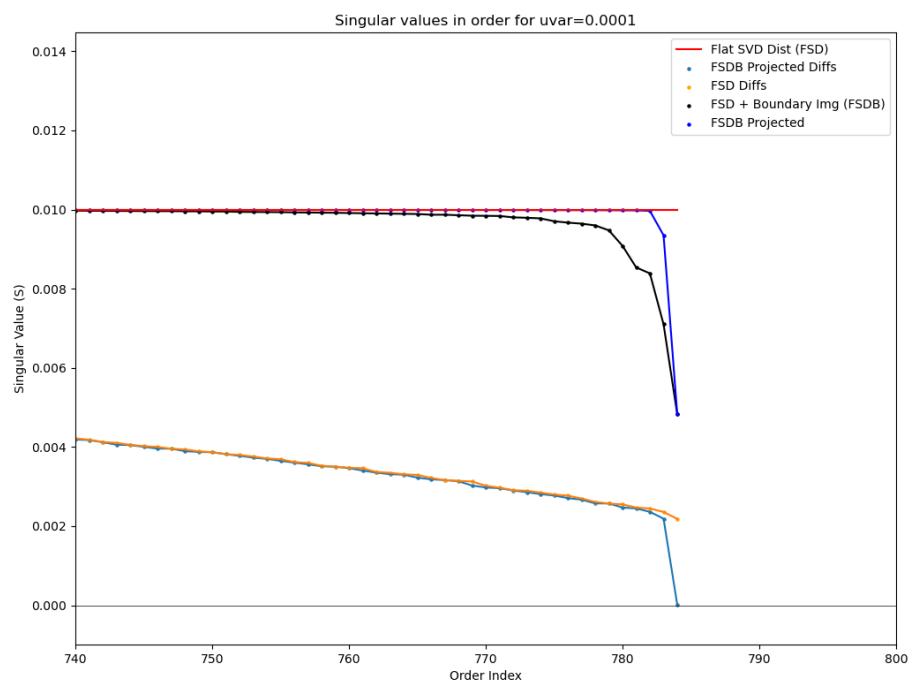
The following plots are generated with a new distribution to replace the gaussian. This "Flat SVD" distribution is generated by first taking a gaussian sample representable by a matrix X , then taking the SVD of this gaussian sample $U\Sigma V = X$, replacing Σ by $|\Sigma|_2 I$. This distribution has the property that its SVD is flat, while maintaining the Frobenius norm of X . This helps to highlight degeneracy in singular values.

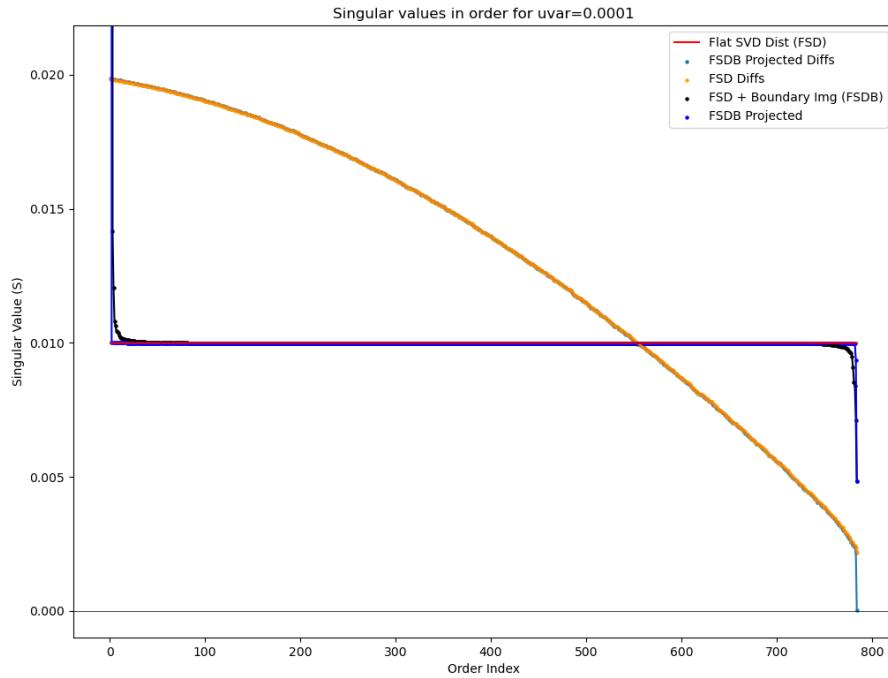
These first two plots are generated by sampling around a decision boundary image found by interpolating between two test images (rather than a test image and an adversarial example generated from it). We note that the SVD contains two degenerate values. These seem to correspond with the two original images of the interpolation.





These last two images are generated with the flattened distribution

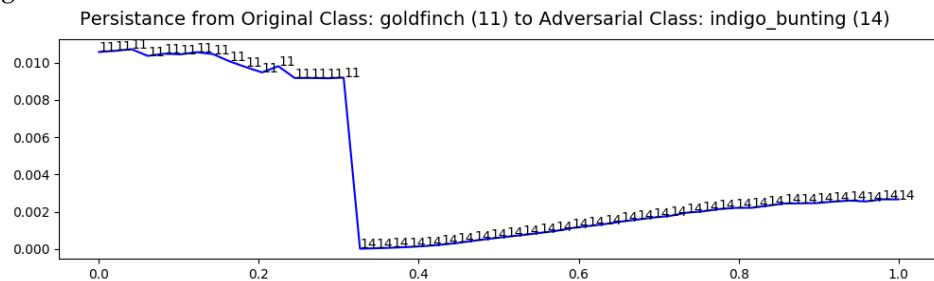




What these experiments tell us is that most of the time when we cross a decision boundary in the MNIST Dataset, we are crossing at a plane, however if we examine a slightly larger ball around a given point, it will intersect the decision boundary at many other places.

32 Re-Examining Persistence

A motivating picture in this research is an image generated while interpolating persistence in the beginning of this research.



The next objective is to examine this particular case with our random walk and projection

Tools.

We also wish to augment these tools to include

32.1 in probability space

32.2 in image space

33 define orthants

34 skewness

**35 sampling decision boundaries and analyzing dimensionality
with PCA**

36 neural network attack gradients versus decision boundaries

37 skewed orthant recreates persistence picture (?).

38 measuring skewness of ANNs

**39 Relate skewness with dimpled manifold and features not bugs
papers**

turn observations into are they a definition, a theorem, or a discussion
decision_boundary crossing

References

[Bishop(2006)] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

- [Carlini and Wagner(2016)] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. *arXiv:1608.04644 [cs]*, Aug. 2016. URL <http://arxiv.org/abs/1608.04644>. arXiv: 1608.04644.
- [Glorot et al.(2011)Glorot, Bordes, and Bengio] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.
- [Goodfellow et al.(2013)Goodfellow, Bulatov, Ibarz, Arnoud, and Shet] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks, 2013.
- [Goodfellow et al.(2014)Goodfellow, Shlens, and Szegedy] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *arXiv:1412.6572 [cs, stat]*, Dec. 2014. URL <http://arxiv.org/abs/1412.6572>. arXiv: 1412.6572.
- [Hardt et al.(2015)Hardt, Recht, and Singer] M. Hardt, B. Recht, and Y. Singer. Train faster, generalize better: Stability of stochastic gradient descent. *CoRR*, abs/1509.01240, 2015. URL <http://arxiv.org/abs/1509.01240>.
- [Ivakhnenko and Lapa(1965)] A. G. Ivakhnenko and V. G. Lapa. *Cybernetic predicting devices*. CCM Information Corporation, 1965.
- [Kak(1993)] S. Kak. On training feedforward neural networks. *Pramana*, 40(1):35–42, 1993.
- [Khoury and Hadfield-Menell(2018)] M. Khoury and D. Hadfield-Menell. On the geometry of adversarial examples. *CoRR*, abs/1811.00525, 2018. URL <http://arxiv.org/abs/1811.00525>.
- [Krause(2020)] A. Krause. Introduction to machine learning, August 2020.
- [Kurakin et al.(2016)Kurakin, Goodfellow, and Bengio] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv:1607.02533 [cs, stat]*, July 2016. URL <http://arxiv.org/abs/1607.02533>. arXiv: 1607.02533.
- [LeCun et al.(1988)LeCun, Touresky, Hinton, and Sejnowski] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski. A theoretical framework for back-propagation. In *Proceedings of the 1988*

connectionist models summer school, volume 1, pages 21–28. CMU, Pittsburgh, Pa: Morgan Kaufmann, 1988.

[LeCun et al.(1989)] LeCun, Boser, Denker, Henderson, Howard, Hubbard, and Jackel] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Back-propagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LeCun et al.(1995)] LeCun, Bengio, et al.] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

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

[Li and Yuan(2017)] Y. Li and Y. Yuan. Convergence analysis of two-layer neural networks with relu activation. In *Advances in Neural Information Processing Systems*, pages 597–607, 2017.

[Linnainmaa(1970)] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[Liu and Deng(2015)] S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian conference on pattern recognition (ACPR)*, pages 730–734. IEEE, 2015.

[Malik and Perona(1990)] J. Malik and P. Perona. Preattentive texture discrimination with early vision mechanisms. *JOSA A*, 7(5):923–932, 1990.

[McClelland et al.(1986)] McClelland, Rumelhart, Group, et al.] J. L. McClelland, D. E. Rumelhart, P. R. Group, et al. Parallel distributed processing. *Explorations in the Microstructure of Cognition*, 2:216–271, 1986.

[McCulloch and Pitts(1943)] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Minsky and Papert(1969)] M. Minsky and S. Papert. Perceptrons: Anlntrduction to computational geometry. *MITPress, Cambridge, Massachusetts*, 1969.

[Moosavi-Dezfooli et al.(2015)Moosavi-Dezfooli, Fawzi, and Frossard] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: a simple and accurate method to fool deep neural networks. *arXiv:1511.04599 [cs]*, Nov. 2015. URL <http://arxiv.org/abs/1511.04599>. arXiv: 1511.04599.

[Nair and Hinton(2010)] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. pages 807–814, 2010.

[Papernot et al.(2015)Papernot, McDaniel, Jha, Fredrikson, Celik, and Swami] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. *arXiv:1511.07528 [cs, stat]*, Nov. 2015. URL <http://arxiv.org/abs/1511.07528>. arXiv: 1511.07528.

[Petersen and Voigtlaender(2018)] P. Petersen and F. Voigtlaender. Optimal approximation of piecewise smooth functions using deep relu neural networks. *Neural Networks*, 108:296–330, 2018.

[Prakash et al.(2018)Prakash, Moran, Garber, DiLillo, and Storer] A. Prakash, N. Moran, S. Garber, A. DiLillo, and J. A. Storer. Deflecting adversarial attacks with pixel deflection. *CoRR*, abs/1801.08926, 2018. URL <http://arxiv.org/abs/1801.08926>.

[Rosenblatt(1958)] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rumelhart et al.(1986)Rumelhart, Hinton, and Williams] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[Schmidhuber(2015)] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015. ISSN 0893-6080. doi: <https://doi.org/10.1016/j.neunet.2014.09.003>. URL <http://www.sciencedirect.com/science/article/pii/S0893608014002135>.

[Simonyan and Zisserman(2014)] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[Szegedy et al.(2013)] Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow, and Fergus] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. URL <http://arxiv.org/abs/1312.6199>.

[Werbos(1974)] P. J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences/'phd diss., harvard university. werbos, paul j. 1988. *Generalization of back propagation with application to a recurrent gas market method," Neural Networks*, 1(4):339–356, 1974.

[Wiyatno and Xu(2018)] R. Wiyatno and A. Xu. Maximal jacobian-based saliency map attack. *CoRR*, abs/1808.07945, 2018. URL <http://arxiv.org/abs/1808.07945>.