A GEOMETRIC FRAMEWORK FOR ADVERSARIAL VULNERABILITY IN MACHINE LEARNING

by

Brian Bell

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF MATHEMATICS MATH.ARIZONA.EDU

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY OF APPLIED MATHEMATICS

In the Graduate College

THE UNIVERSITY OF ARIZONA

2023

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Brian Bell entitled "A Geometric Framework for Adversarial Vulnerability in Machine Learning" and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy of Applied Mathematics.

Date: Type defense date here
David Glickenstein
(Chair)

Date: Type defense date here
Kevin Lin
(Member)

Date: Type defense date here
Marek Rychlik
(Member)

Date: Type defense date here
Hoshin Gupta

(Member)

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

____ Date: Type defense date here
Dissertation Director: David Glickenstein

ACKNOWLEDGEMENTS

This document allows you to type your acknowledgements to people, groups and organizations that have helped you along the way…

For my Supportive Friends and Confidants…

Dedicated to my humerous and wonderful late mother Carrie Bell….

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

Contents

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light	$c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

ABSTRACT

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Chapter 1 Kernel Neural Equialence

UNIVERSITY OF ARIZONA

Abstract

David Glickenstein

Department of Mathematics math.arizona.edu

Doctor of Philosophy of Applied Mathematics

A Geometric Framework for Adversarial Vulnerability in Machine Learning

by Brian Bell

We explore the equivalence between neural networks and kernel methods by deriving the first exact representation of any finite-size parametric classification model trained with gradient descent as a kernel machine. We compare our exact representation to the well-known Neural Tangent Kernel (NTK) and discuss approximation error relative to the NTK and other non-exact path kernel formulations. We experimentally demonstrate that the kernel can be computed for realistic networks up to machine precision. We use this exact kernel to show that our theoretical contribution can provide useful insights into the predictions made by neural networks, particularly the way in which they generalize.

1.1 Introduction

This study investigates the relationship between kernel methods and finite parametric models. To date, interpreting the predictions of complex models, like neural networks, has proven to be challenging. Prior work has shown that the inference-time predictions of a neural network can be exactly written as a sum of independent predictions computed with respect to each training point. We formally show that classification models trained with cross-entropy loss can be exactly formulated as a kernel machine. It is our hope that these new theoretical results will open new research directions in the interpretation of neural network behavior.

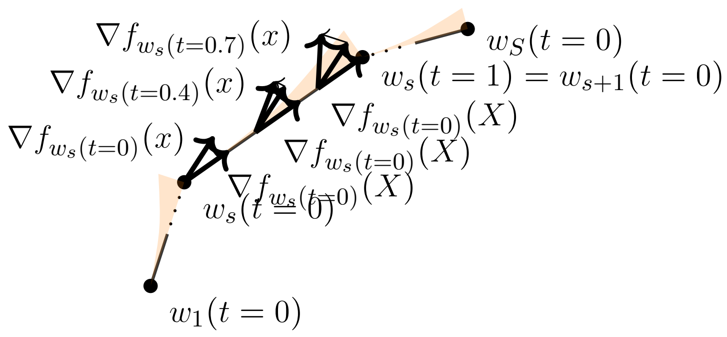


Figure 1.1: Comparison of test gradients used by Discrete Path Kernel (DPK) from prior work (Blue) and the Exact Path Kernel (EPK) proposed in this work (green) versus total training vectors (black) used for both kernel formulations along a discrete training path with S steps. Orange shading indicates cosine error of DPK test gradients versus EPK test gradients shown in practice in Fig. 1.2.

There has recently been a surge of interest in the connection between neural networks and kernel methods **bietti2019bias**; **du2019graphntk**; **tancik2020fourierfeatures**; **abdar2021uq**; **geifman2020similarity**; **chen2020generalized**; **alemohammad2021recurrent**
Much of this work has been motivated by the the neural tangent kernel (NTK), which describes the training dynamics of neural networks in the infinite limit of network

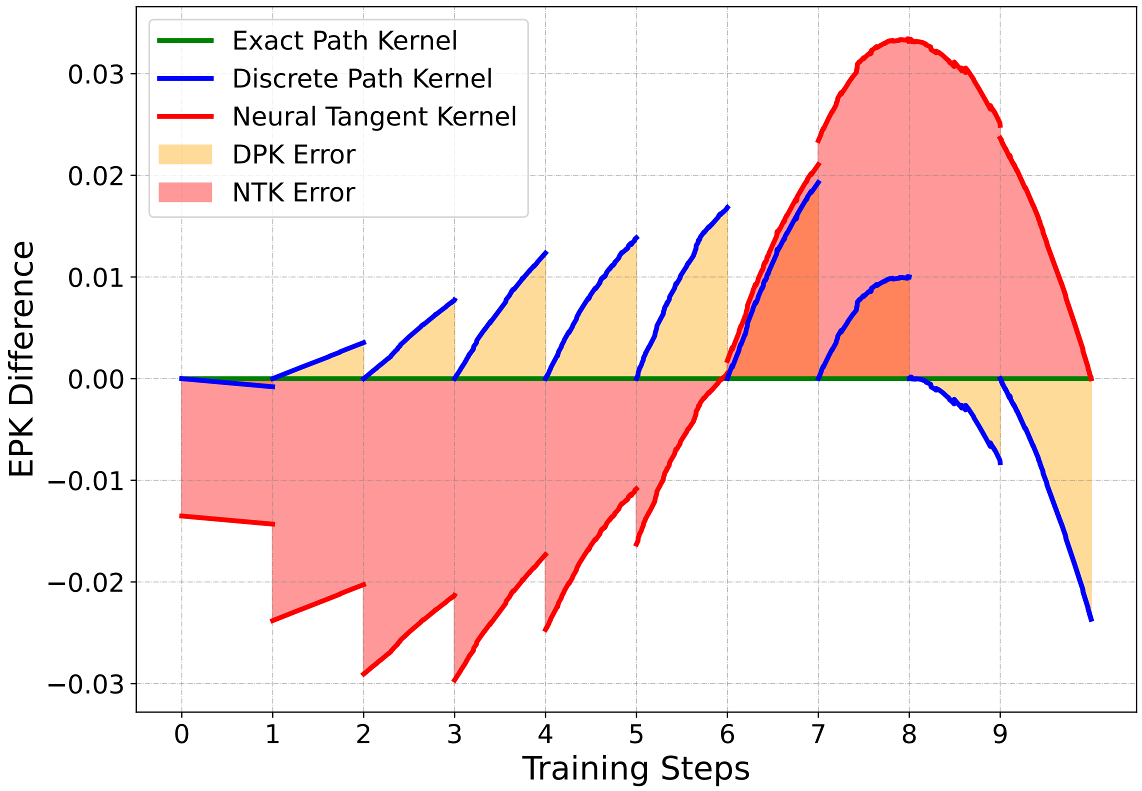


FIGURE 1.2: Measurement of gradient alignment on test points across the training path. The EPK is used as a frame of reference. The y-axis is exactly the difference between the EPK and other representations. For example $EPK - DPK = \langle \phi_{s,t}(X), \phi_{s,t}(x) - \phi_{s,0}(x) \rangle$ (See Definition 3.4). Shaded regions indicate total accumulated error. Note: this is measuring an angle of error in weight space; therefore, equivalent positive and negative error will not result in zero error.

width **jacot2018neural**. We argue that many intriguing behaviors arise in the *finite* parameter regime **DBLP:conf/nips/BubeckS21**. All prior works, to the best of our knowledge, appeal to discrete approximations of the kernel corresponding to a neural network. Specifically, prior approaches are derived under the assumption that training step size is small enough to guarantee close approximation of a gradient flow **ghojogh2021**; **shawe2004kernel**; **zhao2005extracting**.

In this work, we show that the simplifying assumptions used in prior works (i.e. infinite network width and infinitesimal gradient descent steps) are not necessary. Our **Exact Path Kernel (EPK)** provides the first, exact method to study the behavior of finite-sized neural networks used for classification. Previous results are limited in application **incudini2022quantum** due to dependence of the kernel on test data unless strong conditions are imposed on the training process as by **chen2021equivalence**. We show, however, that the training step sizes used in practice do not closely follow this gradient flow, introducing significant error into all prior approaches (Figure 1.2).

Our experimental results build on prior studies attempting to evaluate empirical properties of the kernels corresponding to finite neural networks **DBLP:conf/iclr/LeeBNSPS18**; **chen2021equivalence**. While the properties of infinite neural networks are fairly well understood **neal1996priors**, we find that the kernels learned by finite neural networks have non-intuitive properties that may explain the failures of modern neural networks on important tasks such as robust classification and calibration on out-of-distribution data.

This paper makes the following significant theoretical and experimental contributions:

1. We prove that finite-sized neural networks trained with finite-sized gradient descent steps and cross-entropy loss can be exactly represented as kernel machines using the EPK. Our derivation incorporates a previously-proposed path kernel, but extends this method to account for practical training procedures **domingos2020every**; **chen2021equivalence**.

2. We demonstrate that it is computationally tractable to estimate the kernel underlying a neural network classifier, including for small convolutional computer

vision models.

3. We compute Gram matrices using the EPK and use them to illuminate prior theory of neural networks and their understanding of uncertainty.

4. We employ Gaussian processes to compute the covariance of a neural network's logits and show that this reiterates previously observed shortcomings of neural network generalization.

1.2 Related Work

Fundamentally, the neural tangent kernel (NTK) is rooted in the concept that all information necessary to represent a parametric model is stored in the Hilbert space occupied by the model's weight gradients up to a constant factor. This is very well supported in infinite width **jacot2018neural**. In this setting, it has been shown that neural networks are equivalent to support vector machines, drawing a connection to maximum margin classifiers **chen2021equivalence**; **chizat2020maxmargin**. Similarly, Shah et al. demonstrate that this maximum margin classifier exists in Wasserstien space; however, they also show that model gradients may not contain the required information to represent this **shah2021input**.

The correspondence between kernel machines and parametric models trained by gradient descent has been previously developed in the case of a continuous training path (i.e. the limit as gradient descent step size $\varepsilon \to 0$) **domingos2020**. We will refer to the previous finite approximation of this kernel as the Discrete Path Kernel (DPK). However, a limitation of this formulation is its reliance on a continuous integration over a gradient flow, which differs from the discrete forward

Euler steps employed in real-world model training. This discrepancy raises concerns regarding the applicability of the continuous path kernel to practical scenarios **incudini2022quantum**. Moreover, the formulation of the sample weights and bias term in the DPK depends on its test points. Chen et al. propose that this can be addressed, in part, by imposing restrictions on the loss function used for training, but did not entirely disentangle the kernel formulation from sample importance weights on training points **chen2021equivalence**.

We address the limitations of **domingos2020** and **chen2021equivalence** in Subsection 1.3.5. By default, their approach produces a system which can be viewed as an ensemble of kernel machines, but without a single aggregated kernel which can be analyzed directly. **chen2021equivalence** propose that the resulting sum over kernel machines can be formulated as a kernel machine so long as the sign of the gradient of the loss stays constant through training; however, we show that this is not necessarily a sufficient restriction. Instead, their formulation leads to one of several non-symmetric functions which can serve as a surrogate to replicate a given models behavior, but without retaining properties of a kernel.

1.3 Theoretical Results

Our goal is to show an equivalence between any given finite parametric model trained with gradient descent $f_w(x)$ (e.g. neural networks) and a kernel based prediction that we construct. We define this equivalence in terms of the output of the parametric model $f_w(x)$ and our kernel method in the sense that they form identical maps from input to output. In the specific case of neural network classification models, we consider the mapping $f_w(x)$ to include all layers of the neural network up to and including the log-softmax activation function. Formally:

Definition 1.3.1. *A kernel is a function of two variables which is symmetric and positive semi-definite.*

Definition 1.3.2. *Given a Hilbert space X, a test point $x \in X$, and a training set $X_T = \{x_1, x_2, ...x_n\} \subset X$ indexed by I, a* Kernel Machine *is a model characterized by*

$$K(x) = b + \sum_{i \in I} a_i k(x, x_i) \tag{1.1}$$

where the $a_i \in \mathbb{R}$ do not depend on x, $b \in \mathbb{R}$ is a constant, and k is a kernel. **rasmussen2006gaussian**

By Mercer's theorem **ghojogh2021** *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 use this property to construct a kernel machine of the following form.*

$$K(x) = b + \sum_{i \in I} a_i \langle \phi(x), \phi(x_i) \rangle \tag{1.2}$$

Where ϕ is a function mapping input data into the weight space via gradients. Our ϕ will additionally differentiate between test and training points to resolve a discontinuity that arises under discrete training.

1.3.1 Exact Path Kernels

We 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. Models trained by gradient descent can be characterized by a discrete set of intermediate states in the space of their parameters. These discrete states are often considered to be an estimation of the gradient flow, however in practical settings where $\epsilon \not\to 0$ these discrete states differ from the true gradient flow. Our primary theoretical contribution is an algorithm which accounts for this difference by observing the true path the model followed during training. Here we consider the training dynamics of practical gradient descent steps by integrating a discrete path for weights whose states differ from the gradient flow induced by the training set.

Gradient Along Training Path vs Gradient Field: In order to compute the EPK, gradients on training data must serve two purposes. First, they are the reference points for comparison (via inner product) with test points. Second, they determine the path of the model in weight space. In practice, the path followed during gradient descent does not match the gradient field exactly. Instead, the gradient used to move the state of the model forward during training is only computed for finitely many discrete weight states of the model. In order to produce a path kernel, we must *continuously* compare the model's gradient at test points with *fixed* training gradients along each discrete training step s whose weights we we interpolate linearly by $w_s(t) = w_s - t(w_s - w_{s+1})$. We will do this by integrating across the gradient field induced by test points, but holding each training gradient fixed along the entire discrete step taken. This creates an asymmetry, where test gradients are being measured continuously but the training gradients are being measured discretely (see Figure 1.1).

To account for this asymmetry in representation, we will redefine our data using an indicator to separate training points from all other points in the input space.

Definition 1.3.3. *Let X be two copies of a Hilbert space H with indices* 0 *and* 1 *so that $X = H \times \{0,1\}$. We will write $x \in H \times \{0,1\}$ so that $x = (x_H, x_I)$ (For brevity, we will omit writing $_H$ and assume each of the following functions defined on H will use x_H and x_I will be a hidden indicator). Let f_w be a differentiable function on H parameterized by $w \in \mathbb{R}^d$. Let $X_T = \{(x_i, 1)\}_{i=1}^M$ be a finite subset of X of size M with corresponding observations $Y_T = \{y_{x_i}\}_{i=1}^M$ with initial parameters w_0 so that there is a constant $b \in \mathbb{R}$ such that for all x, $f_{w_0}(x) = b$. Let L be a differentiable loss function of two values which maps $(f(x), y_x)$ into the positive real numbers. Starting with f_{w_0}, let $\{w_s\}$ be the sequence of points attained by N forward Euler steps of fixed size ε so that $w_{s+1} = w_s - \varepsilon \nabla L(f(X_T), Y_T)$. Let $x \in H \times \{0\}$ be arbitrary and within the domain of f_w for every w. Then $f_{w_s(t)}$ is a* finite parametric gradient model (FPGM).

Definition 1.3.4. *Let $f_{w_s(t)}$ be an FPGM with all corresponding assumptions. Then, for a given training step s, the* exact path kernel *(EPK) can be written*

$$K_{EPK}(x, x', s) = \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x') \rangle dt \tag{1.3}$$

where

$$\phi_{s,t}(x) = \nabla_w f_{w_s(t,x)}(x) \tag{1.4}$$

$$w_s(t) = w_s - t(w_s - w_{s+1}) \tag{1.5}$$

$$w_s(t,x) = \begin{cases} w_s(0), & \text{if } x_I = 1 \\ w_s(t), & \text{if } x_I = 0 \end{cases} \tag{1.6}$$

Note: *ϕ is deciding whether to select a continuously or discrete gradient based on whether the data is from the training or testing copy of the Hilbert space H. This is due to the inherent asymmetry that is apparent from the derivation of this kernel (see Appendix section .1.2). This choice avoids potential discontinuity in the kernel output when a test set happens to contain training points.*

Lemma 1.3.5. *The exact path kernel (EPK) is a kernel.*

Theorem 1.3.6 (Exact Kernel Ensemble Representation)**.** *A model f_{w_N} trained using discrete steps matching the conditions of the exact path kernel has the following exact representation as an ensemble of N kernel machines:*

$$f_{w_N} = KE(x) := \sum_{s=1}^{N} \sum_{i=1}^{M} a_{i,s} K_{EPK}(x, x', s) + b \tag{1.7}$$

where

$$a_{i,s} = -\varepsilon \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \tag{1.8}$$

$$b = f_{w_0}(x) \tag{1.9}$$

Assuming the theorem hypothesis, we'll measure the change in model output as we interpolate across each training step s by measuring the change in model state along a linear parametrization $w_s(t) = w_s - t(w_s - w_{s+1})$. We will let d denote the number of parameters of f_w. For brevity, we define $L(x_i, y_i) = l(f_{w_s(0)}(x_i), y_i)$ where l is the loss function used to train the model.

$$\frac{d\hat{y}}{dt} = \sum_{j=1}^{d} \frac{d\hat{y}}{\partial w_j}\frac{dw_j}{dt} \tag{1.10}$$

$$= \sum_{j=1}^{d} \frac{df_{w_s(t)}(x)}{\partial w_j}\left(-\varepsilon \sum_{i=1}^{M} \frac{\partial L(x_i, y_i)}{\partial f_{w_s(0)}(x_i)} \frac{\partial f_{w_s(0)}(x_i)}{\partial w_j}\right) \tag{1.11}$$

We use fundamental theorem of calculus to integrate this equation from step s to step $s+1$ and then add up across all steps. See Appendix .1.2 for the full proof.

Remark 1 Note that in this formulation, b depends on the test point x. In order to ensure information is not being leaked from the kernel into this bias term the model f must have constant output for all input. When relaxing this property, to allow for models that have a non-constant starting output, but still requiring b to remain constant, we note that this representation ceases to be exact for all x. The resulting approximate representation has logit error bounded by its initial bias which can be chosen as $b = \text{mean}(f_{w_0(0)}(X_T))$. Starting bias can be minimized by starting with small parameter values which will be out-weighed by contributions from training. In practice, we sidestep this issue by initializing all weights in the final layer to 0, resulting in $b = \log(\text{softmax}(0))$, thus removing b's dependence on x.

Remark 2 The exactness of this proof hinges on the *separate* measurement of how the model's parameters change. The gradients on training data, which are fixed from one step to the next, measure how the parameters are changing. This is opposed to the gradients on test data, which are *not* fixed and vary with time. These measure a continuous gradient field for a given point. We are using interpolation as a way to measure the difference between the step-wise linear training path and the continuous loss gradient field.

Algorithm 1 Exact Path Kernel: Given a training set (X, Y) with M data points, a testing point x and N weight states $\{w_0, w_1...w_N\}$, the kernel machine corresponding to the exact path kernel can be calculated for a model with W weights and K outputs. We estimate the integral across test points by calculating the Riemann sum with sufficient steps (T) to achieve machine precision. For loss functions that do not have constant gradient values throughout training, this algorithm produces an ensemble of kernel machines.

```
$b = f(w_0, x)$
**for** $s = 0$ **to** $N$ **do**
    $J^X = \nabla_w f_{w_s(0)}(X)$   {Jacobian of training point outputs w.r.t model weights $[M \times K \times W]$ }
    **for** t **from** 0 **to** 1 **with step** 1/T **do**
        $w_s(t) = w_s + t(w_{s+1} - w_s)$
        $J^x \mathrel{+}= \frac{1}{T}\nabla_w f_{w_s(t)}(x)$  {Jacobian of testing point output w.r.t model weights averaged across $T$ steps $[K \times W]$}
    **end for**
    $G_{ijk} = \sum_w J^X_{ijw} J^x_{kw}$  {Inner product on the weight space, this is the kernel value $[M \times K \times K]$}
    $L' = \nabla_f L(f_{w_s(0)}(X), Y)$ {Jacobian of loss w.r.t model output of training points $[M \times K]$}
    $P^s_{ik} = \sum_j L'_{ij} G_{ijk}$   {Inner product of kernel value scaled by loss gradients $[M \times K]$}
**end for**
$\mathcal{P}_{sik} = \{P^0, P^1, ..., P^N\}$   {Stack values across all training steps $[N \times M \times K]$}
$\hat{p} = -\varepsilon \frac{1}{M} \sum_s \sum_i \mathcal{P}_{sik} + b$  {Sum across training steps and average across training points for final prediction $[K]$} =0
```

Theorem 1.3.7 (Exact Kernel Machine Reduction)**.** *Let $\nabla L(f(w_s(x), y)$ be constant across steps s, $(a_{i,s}) = (a_{i,0})$. Let the kernel across all N steps be defined as*

$K_{NEPK}(x, x') = \sum_{s=1}^{N} a_{i,0} K_{EPK}(x, x', s)$ *Then the exact kernel ensemble representation for* f_{w_N} *can be reduced exactly to the kernel machine representation:*

$$f_{w_N}(x) = KM(x) := b + \sum_{i=1}^{M} a_{i,0} K_{NEPK}(x, x') \tag{1.12}$$

See Appendix .1.3 for full proof. By combining theorems 1.3.6 and 1.3.7, we can construct an exact kernel machine representation for any arbitrary parameterized model trained by gradient descent which satisfies the additional property of having constant loss across training steps (e.g. any ANN using catagorical cross-entropy loss (CCE) for classification). This representation will produce exactly identical output to the model across the model's entire domain. This establishes exact kernel-neural equivalence for classification ANNs. Furthermore, Theorem 1.3.6 establishes an exact kernel ensemble representation without limitation to models using loss functions with constant derivatives across steps. It remains an open problem to determine other conditions under which this ensemble may be reduced to a single kernel representation.

1.3.2 Discussion

$\phi_{s,t}(x)$ 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 exact path compared with the gradient flow for each step.

We can write another function k' which is continuous but not symmetric, yet still produces an exact representation:

$$k'(x, x') = \langle \nabla_w f_{w_s(t)}(x), \nabla_w f_{w_s(0)}(x') \rangle \tag{1.13}$$

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

$$\int_0^1 \nabla_w f_{w_s(t)}(x) dt = \nabla_w f_{w_s(0)}(x) \tag{1.14}$$

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

$$\frac{\partial w_s(t)}{dt} = -\varepsilon \nabla_w L(f_{w_s(0)}(x_i), y_i) \tag{1.15}$$

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

$$\int_0^1 \nabla_w f_{w_s(t)}(x) dt \approx \nabla_w f_{w_s(0)}(x) \tag{1.16}$$

In particular, given a model f that admits a Lipshitz constant K this approximation has error bounded by εK and a proof of this convergence is direct. This demonstrates that the asymmetry of this function is exactly measuring the disagreement between the discrete steps taken during training with the gradient field. This function is one of several subjects for further study, particularly in the context of Gaussian processes whereby the asymmetric Gram matrix corresponding with this function can stand in for a covariance matrix. It may be that the not-symmetric analogue of the covariance in this case has physical meaning relative to uncertainty.

1.3.3 Independence from Optimization Scheme

We can see that by changing equation 1.15 we can produce an exact representation for any first order discrete optimization scheme that can be written in terms of model gradients aggregated across subsets of training data. This could include backward Euler, leapfrog, and any variation of adaptive step sizes. This includes stochastic gradient descent, and other forms of subsampling (for which the training sums need only be taken over each sample). One caveat is adversarial training, whereby the a_i are now sampling a measure over the continuum of adversarial images. We can write this exactly, however computation will require approximation across the measure. Modification of this kernel for higher order optimization schemes remains an open problem.

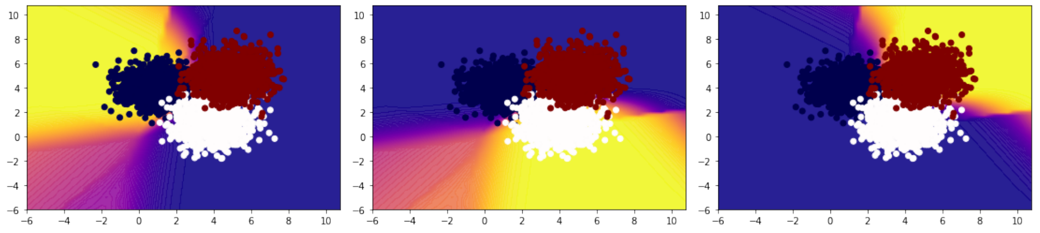


FIGURE 1.3: Updated predictions with kernel a_i updated via gradient descent with training data overlaid for classes 1 (left), 2 (middle), and 3 (right). The high prediction confidence in regions far from training points demonstrates that the learned kernel is non-stationary.

1.3.4 Ensemble Reduction

In order to reduce the ensemble representation of Equation (1.7) to the kernel representation of Equation (1.12), we require that the sum over steps still retain the properties of the kernel (symmetry and positive semi-definiteness). In particular we require that for every subset of the training data x_i and arbitrary α_i and α_j, we have

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{l=1}^{M}\sum_{s=1}^{N} \alpha_i \alpha_j a_{l,s} \int_0^1 K_{\text{EPK}}(x_i, x_j)dt \geq 0 \tag{1.17}$$

A sufficient condition for this reduction is that the gradient of the loss function does not change throughout training. This is the case for categorical cross-entropy where labels are in $\{0, 1\}$. In fact, in this specific context the gradient of the loss function does not depend on $f(x)$, and are fully determined by the ground truth label, making the gradient of the cross-entropy loss a constant value throughout training (See Appendix section .1.3). Showing the positive-definiteness of more general loss functions (e.g. mean squared error loss) will likely require additional regularity conditions on the training path, and is left as future work.

1.3.5 Prior Work

Constant sign loss functions have been previously studied by Chen et al. **chen2021equivalence**, however the kernel that they derive for a finite-width case is of the form

$$K(x, x_i) = \int_0^T |\nabla_f L(f_t(x_i), y_i)| \langle \nabla_w f_t(x), \nabla_w f_t(x_i) \rangle dt \tag{1.18}$$

The summation across these terms satisfies the positive semi-definite requirement of a kernel, however the weight $|\nabla L(f_t(x_i), y_i)|$ depends on x_i which is one of the two inputs. This makes the resulting function $K(x, x_i)$ asymmetric and therefore not a kernel.

1.3.6 Uniqueness

Uniqueness of this kernel is not guaranteed. The mapping from paths in gradient space to kernels is in fact a function, meaning that each finite continuous path has a unique exact kernel representation of the form described above. However, this function is not necessarily onto the set of all possible kernels. This is evident from the existence of kernels for which representation by a finite parametric function is impossible. Nor is this function necessarily one-to-one since there is a continuous manifold of equivalent parameter configurations for neural networks. For a given training path, we can pick another path of equivalent configurations whose gradients will be separated by some constant $\delta > 0$. The resulting kernel evaluation along this alternate path will be exactly equivalent to the first, despite being a unique path. We also note that the linear path l_2 interpolation is not the only valid path between two discrete points in weight space. Following the changes in model weights along a path defined by Manhattan Distance is equally valid and will produce a kernel machine with equivalent outputs. It remains an open problem to compute paths from two different starting points which both satisfy the constant bias condition from Definition (1.3.4) which both converge to the same final parameter configuration and define different kernels.

1.4 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.

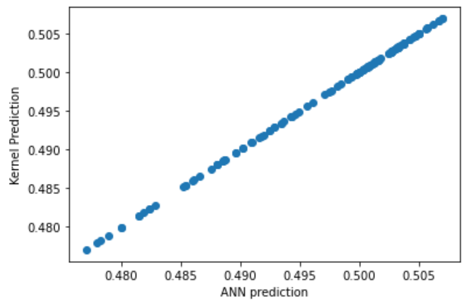


FIGURE 1.4: Class 1 EPK Kernel Prediction (Y) versus neural network prediction (X) for 100 test points, demonstrating extremely close agreement.

1.4.1 Evaluating The Kernel

A small test data set within 100 dimensions is created by generating 1000 random samples with means $(1, 4, 0, ...)$, $(4, 1, 0, ...)$ and $(5, 5, 0, ...)$ and standard deviation 1.0. These points are labeled according to the mean of the Gaussian used to generate them, providing 1000 points each from 3 classes. A fully connected ReLU network with 1 hidden layer is trained using categorical cross-entropy (CCE) and gradient descent with gradients aggregated across the entire training set for each step. We then compute the EPK for this network, approximating the integral from Equation 1.3 with 100 steps which replicates the output from the ReLU network within machine precision. The EPK (Kernel) outputs are compared with neural network predictions in Fig. 1.4 for class 1. Having established this kernel, and its corresponding kernel machine, one natural extension is to allow the kernel weights a_i to be retrained. We perform this updating of the krenel weights using a SVM and present its predictions for each of three classes in Fig. 1.3.

1.4.2 Kernel Analysis

Having established the efficacy of this kernel for model representation, the next step is to analyze this kernel to understand how it may inform us about the properties of the corresponding model. In practice, it becomes immediately apparent that this kernel lacks typical properties preferred when humans select kernels. Fig. 1.3 show that the weights of this kernel are non-stationary on our toy problem, with very stable model predictions far away from training data. Next, we use this kernel to estimate uncertainty. Consistent with many other research works on Gaussian processes for classification **rasmussen2006gaussian** we use a GP to regress to logits. We then use Monte-Carlo to estimate posteriors with respect to probabilities (post-soft-max) for each prediction across a grid spanning the training points of our toy problem. The result is shown on the right-hand column of Fig. 1.5. We can see that the kernel values are more confident (lower standard deviation) and more stable (higher kernel values) the farther they get from the training data in most directions.

In order to further understand how these strange kernel properties come about, we exercise another advantage of a kernel by analyzing the points that are contributing to the kernel value for a variety of test points. In Fig. 1.6 we examine the kernel values for each of the training points during evaluation of three points chosen as the mean of the generating distribution for each class. The most striking property of these kernel point values is the fact that they are not proportional to the euclidean distance from the test point. This appears to indicate a set of basis vectors relative to each test point learned by the model based on the training data which are used to spatially transform the data in preparation for classification. This may relate to the correspondence between neural networks and maximum margin classifiers discussed in related work (**chizat2020maxmargin shah2021input**). Another more subtle property is that some individual data points, mostly close to decision boundaries are slightly over-weighted compared to the other points in their class. This latter property points to the fact that during the latter period of training, once the network has already achieved high accuracy, only the few points which continue to receive incorrect predictions, i.e. caught on the wrong side of a decision boundary, will continue contributing to the training gradient and therefore to the kernel value.

1.4.3 Extending To Image Data

We perform experiments on MNIST to demonstrate the applicability to image data. This kernel representation was generated for convolutional ReLU Network with the categorical cross-entropy loss function, using Pytorch **pytorch**. 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 for each training input to compute $\nabla_w f_{w_s(0)}(x_i)$, and then repeating this procedure for 200 t values between 0 and 1 in order to approximate $\int_0^1 f_{w_s(t)}(x)$. For MNIST, the resulting prediction is very sensitive to the accuracy of this integral approximation, as shown in Fig. 1.7. The top plot shows approximation of the above integral with only one step, which corresponds to the DPK from previous work (**chen2021equivalence**, **domingos2020**, **incudini2022quantum**) and as we can see, careful approximation of this integral is necessary to achieve an accurate match between the model and kernel.

1.5 Conclusion and Outlook

The implications of a practical and finite kernel representation for the study of neural networks are profound and yet importantly limited by the networks that they are built from. For most gradient trained models, there is a disconnect between the input space (e.g. images) and the parameter space of a network. Parameters are intrinsically difficult to interpret and much work has been spent building approximate mappings that convert model understanding back into the input space in order to interpret features, sample importance, and other details **simonyan2013deep**; **lundberg2017unified**; **Selvaraju_2019**. The EPK is composed of a direct mapping from the input space into parameter space. This mapping allows for a much deeper understanding of gradient trained models because the internal state of the method has an exact representation mapped from the input space. As we have shown in Fig. 1.6, kernel values derived from gradient methods tell an odd story. We have observed a kernel that picks inputs near decision boundaries to emphasize and derives a spatial transform whose basis vectors depend neither uniformly nor continuously on training points. Although kernel values are linked to sample importance, we have shown that most contributions to the kernel's prediction for a given point are measuring an overall change in the network's internal representation. This supports the notion that most of what a network is doing is fitting a spatial transform based on a wide aggregation of data, and only doing a trivial calculation to the data once this spatial transform has been determined **chizat2020maxmargin**. As stated in previous work **domingos2020**, 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, $a_{i,s} = -\varepsilon \frac{\partial L(f_{w_s(0)}(x_i), y_i)}{\partial f_i}$, 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 1.8 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 **hardle2004nonparametric**.

For kernel methods, our result also represents a new direction. Despite their firm mathematical foundations, kernel methods have lost ground since the early 2000s because the features implicitly learned by deep neural networks yield better accuracy than any known hand-crafted kernels for complex high-dimensional problems **NIPS2005_663772ea**. We're hopeful about the scalability of learned kernels based on recent results in scaling kernel methods **snelson2005sparse**. Exact kernel equivalence could allow the use of neural networks to implicitly construct a kernel. This could allow kernel based classifiers to approach the performance of neural networks on complex data. Kernels built in this way may be used with Gaussian processes to allow meaningful direct uncertainty measurement. This would allow for much more significant analysis for out-of-distribution samples including adversarial attacks **szegedy2013intriguing**; Ilyas et al., 2019. There is significant work to be done in improving the properties of the kernels learned by neural networks for these tools to be used in practice. We are confident that this direct connection between practical neural networks and kernels is a strong first step towards achieving this goal.

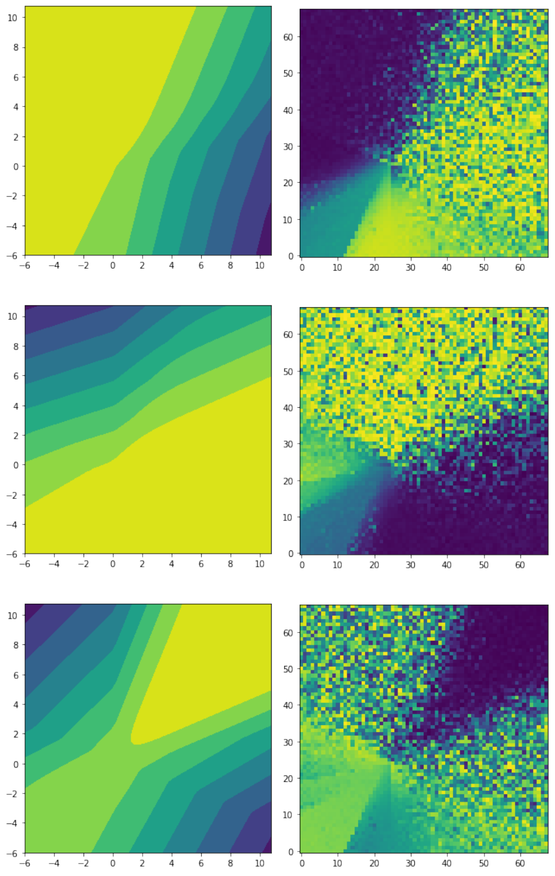


FIGURE 1.5: (left) Kernel values measured on a grid around the training set for our 2D problem. Bright yellow means high kernel value (right) Monte-Carlo estimated standard deviation based on gram matrices generated using our kernel for the same grid as the kernel values. Yellow means high standard deviation, blue means low standard deviation.

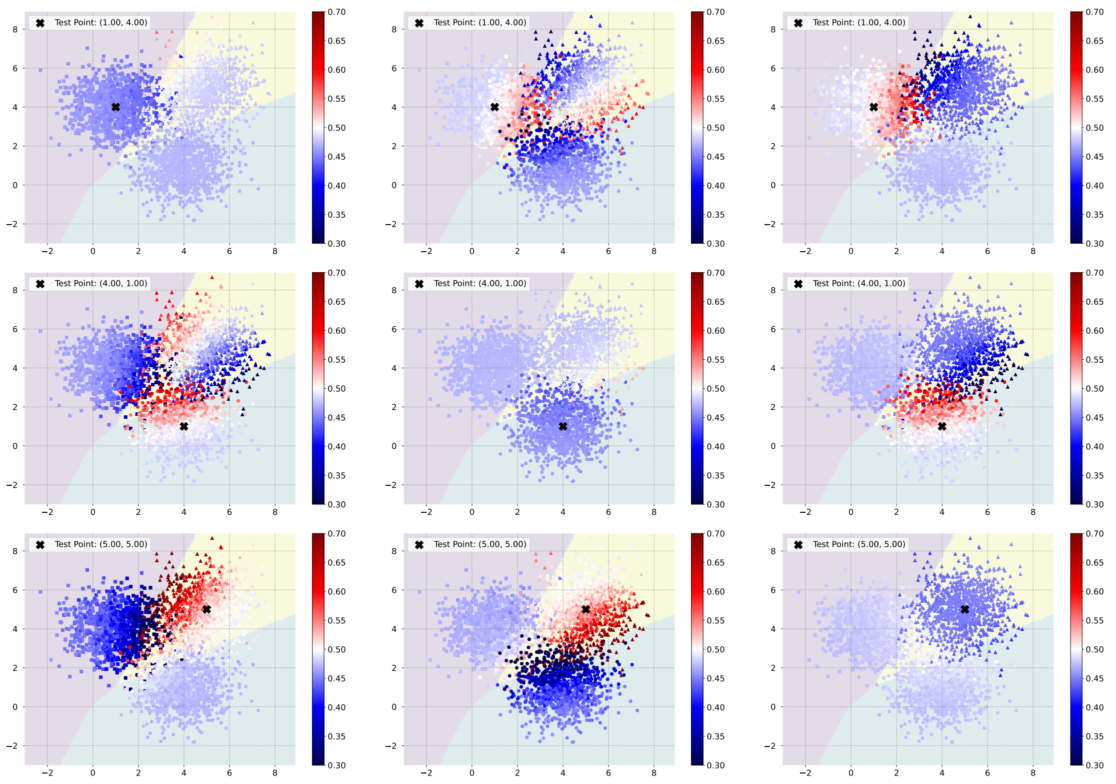
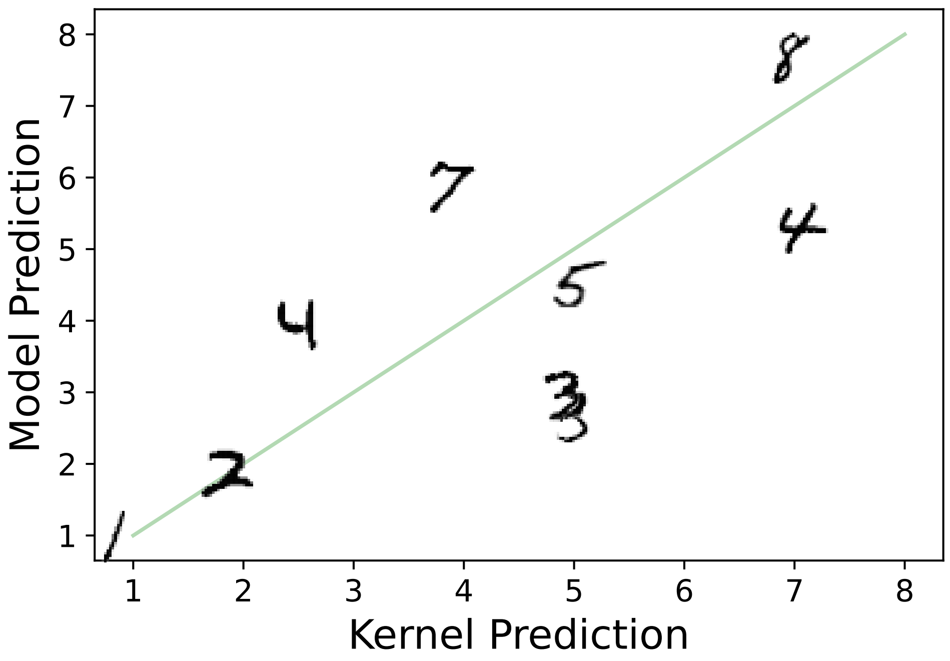
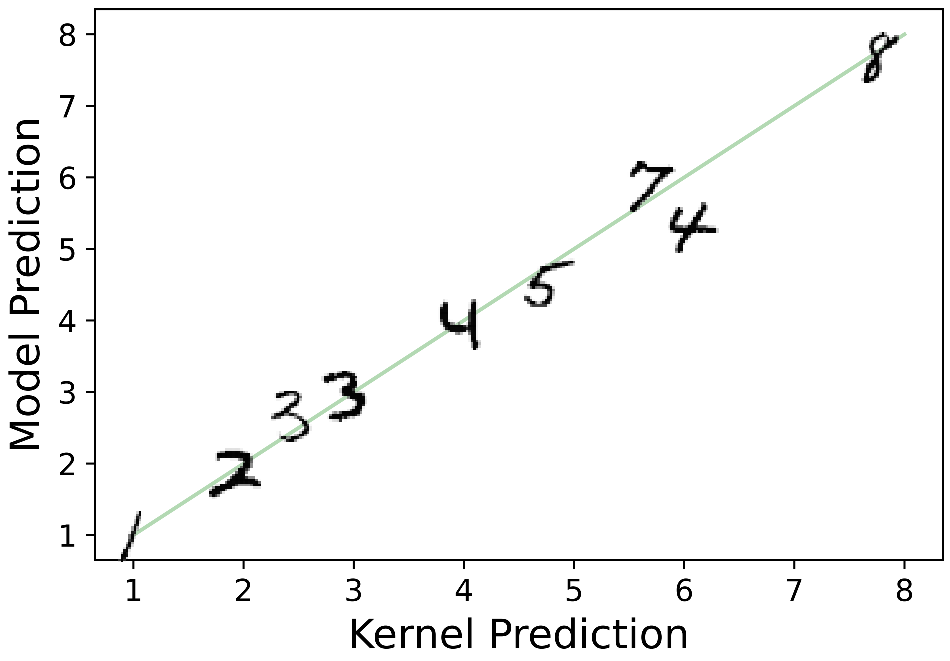
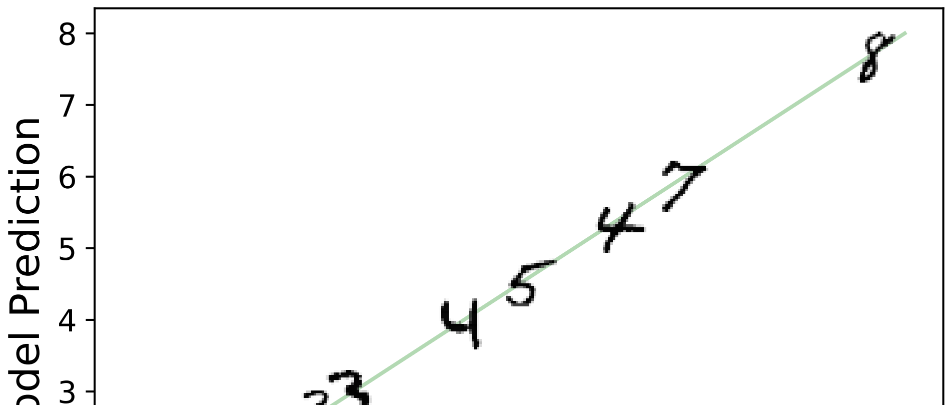


FIGURE 1.6: Plots showing kernel values for each training point relative to a test point. Because our kernel is replicating the output of a network, there are three kernel values per sample on a three class problem. This plot shows kernel values for all three classes across three different test points selected as the mean of the generating distribution. Figures on the diagonal show kernel values of the predicted class. Background shading is the neural network decision boundary.


Model Prediction
Kernel Prediction
1
2
3
4
5
6
7
8


Model Prediction
Kernel Prediction
1
2
3
4
5
6
7
8


Model Prediction
3
4
5
6
7
8

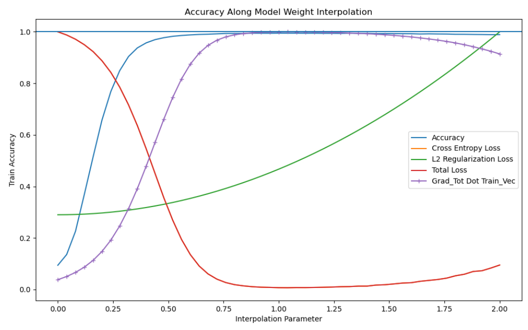


FIGURE 1.8: This plot shows a linear interpolation $w(t) = w_0 + t(w_1 - w_0)$ of model parameters w for a convolutional neural network f_w from their starting random state w_0 to their ending trained state w_1. The hatched purple line shows the dot product of the sum of the gradient over the training data X, $\langle \nabla_w f_{w(t)}(X), (w_1 - w_0)/|w_1 - w_0| \rangle$. The other lines indicate accuracy (blue), total loss (red decreasing), and L2 Regularization (green increasing)

.1 Appendix

.1.1 The EPK is a Kernel

Lemma 1.3.5. *The exact path kernel (EPK) is a kernel.*

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

To see that K_{EPK} is **Positive Semi-Definite**, let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)^\top \in \mathbb{R}^n$ be any vector. We need to show that $\alpha^\top K_{\text{EPK}} \alpha \geq 0$. We have

$$\begin{aligned}
\alpha^\top K_{\text{EPK}}\alpha &= \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j\int_0^1\langle\phi_{s,t}(x_i),\phi_{s,t}(x_j)\rangle dt && (19)\\
&= \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j\int_0^1\langle\nabla_w\hat{y}_{w_s(t,x_i)},\nabla_w\hat{y}_{w_s(t,x_j)}\rangle dt && (20)\\
&= \int_0^1\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j\langle\nabla_w\hat{y}_{w_s(t,x_i)},\nabla_w\hat{y}_{w_s(t,x_j)}\rangle dt && (21)\\
&= \int_0^1\sum_{i=1}^{n}\sum_{j=1}^{n}\langle\alpha_i\nabla_w\hat{y}_{w_s(t,x_i)},\alpha_j\nabla_w\hat{y}_{w_s(t,x_j)}\rangle dt && (22)\\
&= \int_0^1\langle\sum_{i=1}^{n}\alpha_i\nabla_w\hat{y}_{w_s(t,x_i)},\sum_{j=1}^{n}\alpha_j\nabla_w\hat{y}_{w_s(t,x_j)}\rangle dt && (23)\\
&\text{Re-ordering the sums so that their indices match, we have} && (24)\\
&= \int_0^1\left\|\sum_{i=1}^{n}\alpha_i\nabla_w\hat{y}_{w_s(t,x_i)}\right\|^2 dt && (25)\\
&\geq 0, && (26)
\end{aligned}$$

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

□

Remark 3 In the case that our mapping function φ is not symmetric, after re-ordering, we still yield something of the form:

$$= \int_0^1\left\|\sum_{i=1}^{n}\alpha_i\nabla_w\hat{y}_{w_s(t,x_i)}\right\|^2 dt \qquad (27)$$

$$(28)$$

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.

.1.2 The EPK gives an Exact Representation

Theorem 1.3.6 (Exact Kernel Ensemble Representation)**.** *A model f_{w_N} trained using discrete steps matching the conditions of the exact path kernel has the following exact representation as an ensemble of N kernel machines:*

$$f_{w_N} = KE(x) := \sum_{s=1}^{N} \sum_{i=1}^{M} a_{i,s} K_{EPK}(x, x', s) + b \tag{1.7}$$

where

$$a_{i,s} = -\varepsilon \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \tag{1.8}$$

$$b = f_{w_0}(x) \tag{1.9}$$

Proof. Let f_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, $f_{w_0}(x) = b$, and weights at each step $w_s : 0 \leq s \leq N$. Let $x \in X$ be arbitrary and within the domain of f_w for every w. For the final trained state of this model f_{w_N}, let $y = f_{w_N}(x)$.

For one step of training, we consider $y_s = f_{w_{s(0)}}(x)$ and $y_{s+1} = f_{w_{s+1}}(x)$. We wish to account for the change $y_{s+1} - y_s$ in terms of a gradient flow, so we must compute $\frac{\partial y}{dt}$ 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 parameterize this step in terms of the weights:

$$\frac{dw_s(t)}{dt} = (w_{s+1} - w_s) \tag{29}$$

$$\int \frac{dw_s(t)}{dt} dt = \int (w_{s+1} - w_s) dt \tag{30}$$

$$w_s(t) = w_s + t(w_{s+1} - w_s) \tag{31}$$

$$\tag{32}$$

Since f is being trained using forward Euler, across the entire training set X we can write:

$$\frac{dw_s(t)}{dt} = -\varepsilon \nabla_w L(f_{w_{s(0)}}(X), y_i) = -\varepsilon \sum_{j=1}^{d} \sum_{i=1}^{M} \frac{\partial L(f_{w_{s(0)}}(x_i), y_i)}{\partial w_j} \tag{33}$$

Applying chain rule and the above substitution, we can write

$$\frac{d\hat{y}}{dt} = \frac{df_{w_s(t)}}{dt} = \sum_{j=1}^{d} \frac{df}{\partial w_j}\frac{\partial w_j}{dt} \tag{34}$$

$$= \sum_{j=1}^{d} \frac{df_{w_s(t)}(x)}{\partial w_j}\left(-\varepsilon \frac{\partial L(f_{w_s(0)}(X_T), Y_T)}{\partial w_j}\right) \tag{35}$$

$$= \sum_{j=1}^{d} \frac{df_{w_s(t)}(x)}{\partial w_j}\left(-\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)}\frac{\partial f_{w_s(0)}(x_i)}{\partial w_j}\right) \tag{36}$$

$$= -\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \sum_{j=1}^{d} \frac{df_{w_s(t)}(x)}{\partial w_j}\frac{df_{w_s(0)}(x_i)}{\partial w_j} \tag{37}$$

$$= -\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \nabla_w f_{w_s(t)}(x) \cdot \nabla_w f_{w_s(0)}(x_i) \tag{38}$$

$$\tag{39}$$

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

$$y_{s+1} - y_s = \int_0^1 -\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \nabla_w f_{w_s(t)}(x) \cdot \nabla_w f_{w_s(0)}(x_i) dt \tag{40}$$

$$= -\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \left(\int_0^1 \nabla_w f_{w_s(t)}(x) dt\right) \cdot \nabla_w f_{w_s(0)}(x_i) \tag{41}$$

$$\tag{42}$$

For all N training steps, we have

$$\begin{aligned}
y_N &= b + \sum_{s=1}^{N} y_{s+1} - y_s \\
y_N &= b + \sum_{s=1}^{N} -\varepsilon \sum_{i=1}^{M} \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \left(\int_0^1 \nabla_w f_{w_s(t)}(x) dt \right) \cdot \nabla_w f_{w_s(0)}(x_i) \\
&= b + \sum_{i=1}^{M} \sum_{s=1}^{N} -\varepsilon \frac{dL(f_{w_s(0)}(x_i), y_i)}{df_{w_s(0)}(x_i)} \int_0^1 \left\langle \nabla_w f_{w_s(t,x)}(x), \nabla_w f_{w_s(t,x_i)}(x_i) \right\rangle dt \\
&= b + \sum_{i=1}^{M} \sum_{s=1}^{N} a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle \, dt
\end{aligned}$$

Since an integral of a symmetric positive semi-definite function is still symmetric and positive-definite, each step is thus represented by a kernel machine.

□

.1.3 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 $a_{i,s} = a_{i,0}$ for all $a_{i,s}$. The a_i terms in our kernel machine are determined by the gradient the training loss function. This statement then implies that the gradient of the loss term must be constant throughout training in order to form a kernel. Here we show that this is the case when we consider a log softmax activation on the final layer and a negative log likelihood loss function.

Proof. Assume a two class problem. In the case of a function with multiple outputs, we consider each output to be a kernel. We define our network output $\hat{y}_i$ as all layers up to and including the log softmax and y_i is a one-hot encoded vector.

$$L(\hat{y}_i, y_i) = \sum_{k=1}^{K} -y_i^k(\hat{y}_i^k) \tag{43}$$

$$\tag{44}$$

For a given output indexed by k, if $y_i^k = 1$ then we have

$$L(\hat{y}_i, y_i) = -1(\hat{y}_i^k) \tag{45}$$

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = -1 \tag{46}$$

$$\tag{47}$$

If $y_i^k = 0$ then we have

$$L(\hat{y}_i, y_i) = 0(\hat{y}_i^k) \tag{48}$$

$$\frac{\partial L(\hat{y}_i, y_i)}{\partial \hat{y}_i} = 0 \tag{49}$$

$$\tag{50}$$

In this case, since the loss is scaled directly by the output, and the only other term is an indicator function deciding which class label to take, we get a constant gradient.

This shows that the gradient of the loss function does not depend on $\hat{y}_i$. Therefore:

$$y = b - \varepsilon \sum_{i=1}^{N} \sum_{s=1}^{S} a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle \, dt \tag{51}$$

$$= b - \varepsilon \sum_{i=1}^{N} a_{i,0} \sum_{s=1}^{S} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle \, dt \tag{52}$$

This formulates a kernel machine where

$$a_{i,0} = \frac{\partial L(f_{w_0}(x_i), y_i)}{\partial f_i} \tag{53}$$

$$K(x, x_i) = \sum_{s=1}^{S} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle \, dt \tag{54}$$

$$\phi_{s,t}(x) = \nabla_w f_{w_s(t,x)}(x) \tag{55}$$

$$w_s(t,x) = \begin{cases} w_s, x \in X_T \\ w_s(t), x \notin X_T \end{cases} \tag{56}$$

$$b = 0 \tag{57}$$

□

It is important to note that this does not hold up if we consider the log softmax function to be part of the loss instead of the network. In addition, there are loss structures which can not be rearranged to allow this property. In the simple case of linear regression, we can not disentangle the loss gradients from the kernel formulation, preventing the construction of a valid kernel. For example assume our loss is instead squared error. Our labels are continuous on $\mathbb{R}$ and our activation is the identity function.

$$L(f_i, y_i) = (y_i - f_{i,s})^2 \tag{58}$$

$$\frac{\partial L(f_i, y_i)}{\partial f_i} = 2(y_i - f_{i,s}) \tag{59}$$

This quantity is dependent on f_i and its value is changing throughout training.

In order for

$$\sum_{s=1}^{S} a_{i,s} \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \tag{60}$$

to be a kernel on its own, we need it to be a positive (or negative) definite operator and symetric. 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 positive semi-definiteness can be accounted for:

$$= \sum_{s=1}^{S} 2(y_i - f_{i,s}) \int_0^1 \langle \phi_{s,t}(x), \phi_{s,t}(x_i) \rangle dt \tag{61}$$

$$= \sum_{s=1}^{S} 2(y_i - f_{i,s}) \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \nabla_w f_{w_s(0)}(x_i) \rangle dt \tag{62}$$

$$= \sum_{s=1}^{S} 2 \left(y_i \cdot \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \nabla_w f_{w_s(0)}(x_i) \rangle dt - f_{i,s} \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \nabla_w f_{w_s(0)}(x_i) \rangle dt \right) \tag{63}$$

$$= \sum_{s=1}^{S} 2 \left(y_i \cdot \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \nabla_w f_{w_s(0)}(x_i) \rangle dt - \int_0^1 \langle \nabla_w f_{w_s(t))}(x), f_{i,s} \nabla_w f_{w_s(0)}(x_i) \rangle dt \right) \tag{64}$$

$$= \sum_{s=1}^{S} 2 \left(y_i \cdot \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \nabla_w f_{w_s(0)}(x_i) \rangle dt - \int_0^1 \langle \nabla_w f_{w_s(t))}(x), \frac{1}{2} \nabla_w (f_{w_s(0)}(x_i))^2 \rangle dt \right) \tag{65}$$

$$\tag{66}$$

Otherwise, we get the usual

$$= \sum_{s=1}^{S} 2(y_i - f_{i,s}) \int_0^1 \langle \nabla_w f_{w_s(t,x))}(x), \nabla_w f_{w_s(t,x)}(x') \rangle dt \tag{67}$$

$$\tag{68}$$

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 $f(x_i)$.

However, even accounting for this by removing the sign of the loss will still produce a non-symmetric function. This limitation is more difficult to overcome.

.1.4 Multi-Class Case

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

$$\text{Case 1: } L : \mathbb{R}^K \rightarrow \mathbb{R} \tag{69}$$

$$\text{Case 2: } L : \mathbb{R}^K \rightarrow \mathbb{R}^K \tag{70}$$

$$\tag{71}$$

Case 1 Scalar Loss

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

Let f be a vector valued function so that $f : \mathbb{R}^D \rightarrow \mathbb{R}^K$ 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 $\dfrac{\partial f}{\partial t}$ is a column and has shape $Kx1$ and our first chain rule can be done the old fashioned way on each row of that column:

$$\frac{df}{dt} = \sum_{j=1}^{M} \frac{df(x)}{\partial w_j} \frac{dw_j}{dt} \tag{72}$$

$$= -\varepsilon \sum_{j=1}^{M} \frac{df(x)}{\partial w_j} \sum_{i=1}^{N} \frac{\partial L(f(x_i), y_i)}{\partial w_j} \tag{73}$$

$$\text{Apply chain rule} \tag{74}$$

$$= -\varepsilon \sum_{j=1}^{M} \frac{df(x)}{\partial w_j} \sum_{i=1}^{N} \frac{\partial L(f(x_i), y_i)}{\partial f} \frac{df(x_i)}{\partial w_j} \tag{75}$$

$$\text{Let} \tag{76}$$

$$A = \frac{df(x)}{\partial w_j} \in \mathbb{R}^{K \times 1} \tag{77}$$

$$B = \frac{dL(f(x_i), y_i)}{df} \in \mathbb{R}^{1 \times K} \tag{78}$$

$$C = \frac{df(x_i)}{\partial w_j} \in \mathbb{R}^{K \times 1} \tag{79}$$

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

$$(ABC)^T = (BC)^T A^T = BCA^T \tag{80}$$

$$ABC = (BCA^T)^T \tag{81}$$

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

$$= -\varepsilon \sum_{j=1}^{M} \sum_{i=1}^{N} \left(\frac{dL(f(x_i), y_i)}{df} \frac{df(x_i)}{\partial w_j} \left(\frac{df(x)}{\partial w_j} \right)^T \right)^T \tag{82}$$

$$= -\varepsilon \sum_{i=1}^{N} \left(\frac{dL(f(x_i), y_i)}{df} \sum_{j=1}^{M} \frac{df(x_i)}{\partial w_j} \left(\frac{df(x)}{\partial w_j} \right)^T \right)^T \tag{83}$$

$$\tag{84}$$

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 f along the dimensions of class (index k) and weight (index j). We can define

$$(\nabla f(x))_{k,j} = \frac{df_k(x)}{\partial w_j} \tag{85}$$

$$= -\varepsilon \sum_{i=1}^{N} \left(\frac{dL(f(x_i), y_i)}{df} \nabla f(x_i) (\nabla f(x))^T \right)^T \tag{86}$$

$$\tag{87}$$

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 $[K, 1]$. Also, we note that our kernel inner product now has shape $[K, K]$.

.1.5 Schemes Other than Forward Euler (SGD)

Variable Step Size: Suppose f is being trained using Variable step sizes so that across the training set X:

$$\frac{dw_s(t)}{dt} = -\varepsilon_s \nabla_w L(f_{w_s(0)}(X), y_i) = -\varepsilon \sum_{j=1}^{d} \sum_{i=1}^{M} \frac{\partial L(f_{w_s(0)}(X), y_i)}{\partial w_j} \tag{88}$$

This additional dependence of ε on s simply forces us to keep ε inside the summation in equation 1.11.

Other Numerical Schemes: Suppose f is being trained using another numerical scheme so that:

$$\frac{dw_s(t)}{dt} = \varepsilon_{s,l} \nabla_w L(f_{w_s(0)}(x_i), y_i) + \varepsilon_{s-1,l} \nabla_w L(f_{w_{s-1}}(x_i), y_i) + \cdots \tag{89}$$

$$= \varepsilon_{s,l} \sum_{j=1}^{d} \sum_{i=1}^{M} \frac{\partial L(f_{w_s(0)}(x_i), y_i)}{\partial w_j} + \varepsilon_{s-1,l} \sum_{j=1}^{d} \sum_{i=1}^{M} \frac{\partial L(f_{w_{s-1}(0)}(x_i), y_i)}{\partial w_j} + \cdots \tag{90}$$

This additional dependence of ε on s and l simply results in an additional summation in equation 1.11. Since addition commutes through kernels, this allows separation into a separate kernel for each step contribution. Leapfrog and other first order schemes will fit this category.

Higher Order Schemes: Luckily these are intractable for for most machine-learning models because they would require introducing dependence of the kernel on input data or require drastic changes. It is an open but intractable problem to derive kernels corresponding to higher order methods.

.1.6 Variance Estimation

In order to estimate variance we treat our derived kernel function K as the covariance function for Gaussian process regression. Given training data X and test data X', we can use the Kriging to write the mean prediction and it variance from true $\mu(x)$ as

$$\bar{\mu}(X') = \left[K(X', X)\right] [K(X, X)]^{-1} [Y] \tag{91}$$

$$\mathrm{Var}(\bar{\mu}(X') - \mu(X')) = \left[K(X', X')\right] - \left[K(X', X)\right] [K(X, X)]^{-1} \left[K(X, X')\right] \tag{92}$$

$$\tag{93}$$

Where

$$[K(A, B)] = \begin{bmatrix} K(A_1, B_1) & K(A_1, B_2) & \cdots \\ K(A_2, B_1) & K(A_2, B_2) & \\ \vdots & & \ddots \end{bmatrix} \tag{94}$$

To finish our Gaussian estimation, we note that each $K(A_i, B_i)$ will me a k by k matrix where k is the number of classes. We take $\mathrm{tr}(K(A_i, B_i)$ to determine total variance for each prediction.

Acknowledgements

This material is based upon work supported by the Department of Energy (National Nuclear Security Administration Minority Serving Institution Partnership Program's CONNECT - the COnsortium on Nuclear sECurity Technologies) DE-NA0004107.

This material is based upon work supported by the National Science Foundation under Grant No. 2134237. We would like to additionally acknowledge funding from NSF TRIPODS Award Number 1740858 and NSF RTG Applied Mathematics and Statistics for Data-Driven Discovery Award Number 1937229. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

.1 Appendix: L-BFGS

Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) is a quasi-newton gradient based optimization algorithm which stores a history of gradients and positions from each previous optimization step Liu and Nocedal, 1989. The algorithm as implemented to optimize a function f with gradient at step k of g_k is as follows

L-BFGS

```
0: Choose $x_0, m, 0 < \beta' < 1/2, \beta' < \beta < 1$, and a symmetric positive definite starting
   matrix $H_0$.
0: **for** $k = 0$ to $k =$ (the number of iterations so far) **do**
0:     $d_k = -H_k g_k$,
0:     $x_{k+1} = x_k + \alpha_k d_k$, Where $\alpha_k$ satisfies
```

$$f(x_k + \alpha_k d_k) \leq f(x_k) + \beta' \alpha_k g_k^T d_k,$$
$$g(x_k + \alpha_k d_k)^T d_k \geq \beta g_k^T d_k.$$

```
   {Trying steplength $\alpha_k = 1$ first.}
0:     Let $\hat{m} = \min(k, m-1)$.
0:     **for** $i$ from 0 to $\hat{m}+1$ **do** {Update $H_0$ $\hat{m}+1$ times using pairs $\{y_j, s_j\}_{j=k-\hat{m}}^{k}$,}
```

$$\begin{aligned} H_{k+1} = {} & (V_k^T \cdot V_{k-\hat{m}}^T) H_0 (V_{k-\hat{m}} \cdots V_k) \\ & + \rho_{k-\hat{m}} (V_k^T \cdots V_{k-\hat{m}+1}^T) s_{k-\hat{m}} s_{k-\hat{m}}^T (V_{k-\hat{m}+1} \cdots V_k) \\ & + \rho_{k-\hat{m}+1} (V_k^T \cdots V_{k-\hat{m}+2}^T) s_{k-\hat{m}+1} s_{k-\hat{m}+1}^T (V_{k-\hat{m}+2} \cdots V_k) \\ & \vdots \\ & + \rho_k s_k s_k^T \end{aligned}$$

```
0:     **end for**
0: **end for**=0
```

.2 Appendix: Bracketing Algorithm

This algorithm was implemented in Python for the experiments presented.

```
0: function BRACKETING(image, ANN, n, tol, n_real, c_i) { start with same
   magnitude noise as image}
0:   u_tol, l_tol = 1.01, 0.99
0:   a_var = Variance(image)/4 {Running Variance}
0:   l_var, u_var = 0, a_var*2{ Upper and Lower Variance of search space} {
   Adversarial image plus noise counts}
0:   a_counts = zeros(n)
0:   n_sz = image.shape[0]
0:   mean = Zeros(n_sz)
0:   I = Identity(n_sz)
0:   count = 0 {grab the classification of the image under the network}
0:   y_a = argmax(ANN.forward(image))
0:   samp = N(0, u_var*I, n_real)
0:   image_as = argmax(ANN.forward(image + samp)) {Expand search window}
0:   while Sum(image_as == y_a) > n_real*tol/2 do
0:     u_var = u_var*2
0:     samp = N(0, u_var*I, n_real)
0:     image_as = argmax(ANN.forward(image + samp))
0:   end while{ perform the bracketing }
0:   for i in range(0,n) do
0:     count+=1 {compute sample and its torch tensor}
0:     samp = N(0, a_var*I, n_real)
0:     image_as = argmax(ANN.forward(image + samp))
0:     a_counts[i] = Sum(image_as == y_a)
  {floor and ceiling surround number}
0:     if ((a_counts[i] ≤ Ceil(n_real*(tol*u_tol))) & (a_counts[i] >
   Floor(n_real*(tol*l_tol)))) then
   return a_var
0:     else if (a_counts[i] < n_real*tol) then {we're too high}
0:       u_var = a_var
0:       a_var = (a_var + l_var)/2
0:     else if (a_counts[i] ≥ n_real*tol) then {we're too low}
0:       l_var = a_var
0:       a_var = (u_var + a_var)/2
0:     end if
0:   end for
   return a_var
0: end function=0
```

Bibliography

Bengio, Y et al. (2007). *Greedy layer-wise training of deep networks. NIPS 19 (pp. 153–160).*

Bengio, Yoshua et al. (2009). "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127.

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

Boureau, Y-Lan et al. (2010). "Learning mid-level features for recognition". In: *2010 IEEE computer society conference on computer vision and pattern recognition.* IEEE, pp. 2559–2566.

Carlini, Nicholas and David Wagner (Aug. 2016). "Towards Evaluating the Robustness of Neural Networks". In: *arXiv:1608.04644 [cs].* arXiv: 1608.04644. URL: `http://arxiv.org/abs/1608.04644` (visited on 04/25/2018).

Coates, Adam, Andrew Ng, and Honglak Lee (2011). "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings, pp. 215–223.

Collobert, Ronan, Samy Bengio, and Johnny Mariéthoz (2002). "Torch: a modular machine learning software library". In:

Collobert, Ronan et al. (2011). "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.ARTICLE, pp. 2493–2537.

Erhan, Dumitru et al. (2010). “Why does unsupervised pre-training help deep learning?” In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 201–208.

Frosst, Nicholas, Sara Sabour, and Geoffrey E. Hinton (2018). “DARCCC: Detecting Adversaries by Reconstruction from Class Conditional Capsules”. In: *CoRR* abs/1811.06969. arXiv: `1811.06969`. URL: `http://arxiv.org/abs/1811.06969`.

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). “Deep sparse rectifier neural networks”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 315–323.

Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy (Dec. 2014). “Explaining and Harnessing Adversarial Examples”. In: *arXiv:1412.6572 [cs, stat]*. arXiv: 1412.6572. URL: `http://arxiv.org/abs/1412.6572` (visited on 04/25/2018).

Goodfellow, Ian J. et al. (2013). *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks*. arXiv: `1312.6082 [cs.CV]`.

Hardt, Moritz, Benjamin Recht, and Yoram Singer (2015). “Train faster, generalize better: Stability of stochastic gradient descent”. In: *CoRR* abs/1509.01240. arXiv: `1509.01240`. URL: `http://arxiv.org/abs/1509.01240`.

Hinton, Geoffrey (2010). “A practical guide to training restricted Boltzmann machines”. In: *Momentum* 9.1, p. 926.

Hinton, Geoffrey E, Simon Osindero, and Yee-Whye Teh (2006). “A fast learning algorithm for deep belief nets”. In: *Neural computation* 18.7, pp. 1527–1554.

Hinton, Geoffrey E and Ruslan R Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *science* 313.5786, pp. 504–507.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Ilyas, Andrew et al. (2019). “Adversarial examples are not bugs, they are features”. In: *Advances in neural information processing systems* 32.

Ivakhnenko, Alekse Grigorevich and Valentin Grigorévich Lapa (1965). *Cybernetic predicting devices*. CCM Information Corporation.

Kak, Subhash (1993). “On training feedforward neural networks”. In: *Pramana* 40.1, pp. 35–42.

Khoury, Marc and Dylan Hadfield-Menell (2018). “On the Geometry of Adversarial Examples”. In: *CoRR* abs/1811.00525. arXiv: `1811.00525`. URL: `http://arxiv.org/abs/1811.00525`.

Krause, Andreas (2020). *Introduction to Machine Learning*.

Kurakin, Alexey, Ian Goodfellow, and Samy Bengio (July 2016). “Adversarial examples in the physical world”. In: *arXiv:1607.02533 [cs, stat]*. arXiv: 1607.02533. URL: `http://arxiv.org/abs/1607.02533` (visited on 04/25/2018).

LeCun, Yann, Yoshua Bengio, et al. (1995). “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10, p. 1995.

LeCun, Yann et al. (1988). “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann, pp. 21–28.

LeCun, Yann et al. (1989). “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4, pp. 541–551.

LeCun, Yann et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Lee, Honglak et al. (2009). “Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations”. In: *Proceedings of the 26th annual international conference on machine learning*, pp. 609–616.

Lee, Kimin et al. (2018). “A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks”. In: *NeurIPS*.

Li, Yuanzhi and Yang Yuan (2017). “Convergence Analysis of Two-layer Neural Networks with ReLU Activation”. In: *Advances in Neural Information Processing Systems*, pp. 597–607.

Linnainmaa, Seppo (1970). “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: *Master’s Thesis (in Finnish), Univ. Helsinki*, pp. 6–7.

Liu, Dong C and Jorge Nocedal (1989). “On the limited memory BFGS method for large scale optimization”. In: *Mathematical programming* 45.1-3, pp. 503–528.

Liu, Shuying and Weihong Deng (2015). “Very deep convolutional neural network based image classification using small training sample size”. In: *2015 3rd IAPR Asian conference on pattern recognition (ACPR)*. IEEE, pp. 730–734.

Madry, Aleksander et al. (June 2017). "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *arXiv:1706.06083 [cs, stat]*. arXiv: 1706.06083. URL: `http://arxiv.org/abs/1706.06083` (visited on 04/25/2018).

Malik, Jitendra and Pietro Perona (1990). "Preattentive texture discrimination with early vision mechanisms". In: *JOSA A* 7.5, pp. 923–932.

McClelland, James L, David E Rumelhart, PDP Research Group, et al. (1986). "Parallel distributed processing". In: *Explorations in the Microstructure of Cognition* 2, pp. 216–271.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Mikolov, Tomas et al. (2010). "Recurrent neural network based language model." In: *Interspeech*. Vol. 2. 3. Makuhari, pp. 1045–1048.

Minsky, Marvin and Seymour Papert (1969). "Perceptrons: AnIntroduction to computational geometry". In: *MITPress, Cambridge, Massachusetts*.

Moosavi-Dezfooli, Seyed-Mohsen, Alhussein Fawzi, and Pascal Frossard (Nov. 2015). "DeepFool: a simple and accurate method to fool deep neural networks". In: *arXiv:1511.04599 [cs]*. arXiv: 1511.04599. URL: `http://arxiv.org/abs/1511.04599` (visited on 04/25/2018).

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". en. In: pp. 807–814.

Papernot, Nicolas et al. (Nov. 2015). "The Limitations of Deep Learning in Adversarial Settings". In: *arXiv:1511.07528 [cs, stat]*. arXiv: 1511.07528. URL: `http://arxiv.org/abs/1511.07528` (visited on 04/25/2018).

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Petersen, Philipp and Felix Voigtlaender (2018). "Optimal approximation of piecewise smooth functions using deep ReLU neural networks". In: *Neural Networks* 108, pp. 296–330.

Prakash, Aaditya et al. (2018). "Deflecting Adversarial Attacks with Pixel Deflection". In: *CoRR* abs/1801.08926. arXiv: `1801.08926`. URL: `http://arxiv.org/abs/1801.08926`.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *nature* 323.6088, pp. 533–536.

Schmidhuber, JÃ¼rgen (2015). "Deep learning in neural networks: An overview". In: *Neural Networks* 61, pp. 85 –117. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2014.09.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608014002135`.

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556*.

Szegedy, Christian et al. (2013). "Intriguing properties of neural networks". In: *CoRR* abs/1312.6199. arXiv: `1312.6199`. URL: `http://arxiv.org/abs/1312.6199`.

Tsipras, Dimitris et al. (2018). “Robustness may be at odds with accuracy”. In: *stat* 1050, p. 11.

Vaswani, Ashish et al. (2017). *Attention Is All You Need.* arXiv: `1706.03762 [cs.CL]`.

Vincent, Pascal et al. (2010). “Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion.” In: *Journal of machine learning research* 11.12.

Werbos, Paul J (1974). “Beyond regression: New tools for prediction and analysis in the be havioral sciences/’PhD diss., Harvard Uni versity. Werbos, Paul J. 1988”. In: *Generalization of back propagation with application to a recurrent gas market method,″ Neural Networks* 1.4, pp. 339–356.

Wiyatno, Rey and Anqi Xu (2018). “Maximal Jacobian-based Saliency Map Attack”. In: *CoRR* abs/1808.07945. arXiv: `1808.07945`. URL: `http://arxiv.org/abs/1808.07945`.