Neural Kernels without Tangents

Vaishaal Shankar [1] **Alex Fang** [1] **Wenshuo Guo** [1] **Sara Fridovich-Keil** [1] **Ludwig Schmidt** [1]
Jonathan Ragan-Kelley [2] **Benjamin Recht** [1]

Abstract

We investigate the connections between neural networks and simple building blocks in kernel space. In particular, using well established feature space tools such as direct sum, averaging, and moment lifting, we present an algebra for creating "compositional" kernels from bags of features. We show that these operations correspond to many of the building blocks of "neural tangent kernels" (NTK). Experimentally, we show a correlation in test error between neural network architectures and the associated kernels. We construct a simple neural network architecture using only 3×3 convolutions, 2×2 average pooling, ReLU, and optimized with SGD and MSE loss that achieves 96% accuracy on CIFAR10, and whose corresponding compositional kernel achieves 90% accuracy. We also use our constructions to investigate the relative performance of neural networks, NTKs, and compositional kernels in the small dataset regime. In particular, we find that compositional kernels outperform NTKs and neural networks outperform both kernel methods.

1. Introduction

Recent research has drawn exciting connections between neural networks and kernel methods, providing new insights into training dynamics, generalization, and expressibility (Daniely et al., 2016; Jacot et al., 2018; Du et al., 2019b;a; Ghorbani et al., 2019; Allen-Zhu et al., 2019; Lee et al., 2019). This line of work relates "infinitely wide" neural networks to particular kernel spaces, showing that infinite limits of random initializations of neural networks lead to particular kernels on the same input data. Since these initial investigations, some have proposed to use these kernels in prediction problems, finding promising results on many benchmark problems (Li et al., 2019; Arora et al., 2020). However, these kernels do not match the performance of neural networks on most tasks of interest, and the kernel constructions themselves are not only hard to compute, but their mathematical formulae are difficult to even write down (Arora et al., 2019).

In this paper, we aim to understand empirically if there are computationally tractable kernels that approach the expressive power of neural networks, and if there are any practical links between kernel and neural network architectures. We take inspiration from both the recent literature on "neural tangent kernels" (NTK) and the classical literature on compositional kernels, such as ANOVA kernels. We describe a set of three operations in feature space that allow us to turn data examples presented as collections of small feature vectors into a single expressive feature-vector representation. We then show how to compute these features directly on kernel matrices, obviating the need for explicit vector representations. We draw connections between these operations, the compositional kernels of Daniely et al. (2016), and the Neural Tangent Kernel limits of Jacot et al. (2018). These connections allow us to relate neural architectures to kernels in a transparent way, with appropriate simple analogues of convolution, pooling, and nonlinear rectification (Sec. 3).

Our main investigation, however, is not in establishing these connections. Our goal is to test whether the analogies between these operations hold in practice: is there a correlation between neural architecture performance and the performance of the associated kernel? Inspired by simple networks proposed by David Page (2018), we construct neural network architectures for computer vision tasks using only 3×3 convolutions, 2×2 average pooling, and ReLU nonlinearities. We show that the performance of these neural architectures on CIFAR-10 strongly predicts the performance of the associated kernel. The best architecture achieves 96% accuracy on CIFAR-10 when trained with SGD on a mean squared error (MSE) loss. The corresponding compositional kernel achieves 90% accuracy, which is, to our knowledge, the highest accuracy achieved thus far by a kernel machine on CIFAR-10. We emphasize here that we compute an *exact kernel* directly from pixels, and do not rely on random feature approximations often used in past work.

[1]University of California, Berkeley [2]Massachusetts Institute of Technology. Correspondence to: Vaishaal Shankar <vaishaal@berkeley.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

On CIFAR-10, we observe that compositional kernels provide dramatically better results than Neural Tangent Kernels. We also demonstrate that this trend holds in the "small data" regime (Arora et al., 2020). Here, we find that compositional kernels outperform NTKs and neural networks outperform both kernel methods when properly tuned and trained. On a benchmark of 90 UCI tabular datasets, we find that simple, properly tuned Gaussian kernels perform, on aggregate, slightly better than NTKs. Taken together, our results provide a promising starting point for designing practical, high performance, domain specific kernel functions. We suggest that while some notion of compositionality and hierarchy may be necessary to build kernel predictors that match the performance of neural networks, NTKs themselves may not actually provide particularly useful guides to the practice of kernel methods.

2. Related Work

We build upon many prior efforts to design specialized kernels that model specific types of data. In classical work on designing kernels for pattern analysis, Shawe-Taylor et al. (2004) establishes an algebra for constructing kernels on structured data. In particular, we recall the construction of the ANOVA kernels, which are defined recursively using a set of base kernels. Many ideas from ANOVA kernels transfer naturally to images, with operations that capture the similarities between different patches of data.

More recent work Mairal et al. (2014); Mairal (2016) proposes a multi-layer "convolutional kernel network" (CKN) for image classification that iterates convolutional and nonlinear operations in kernel space. However, CKNs *approximate* the iterative kernel map using monte-carlo and optimization based approaches. While our compositional kernels also perform similar convolutional operations in kernel space, we compute our kernel functions *exactly*, and the computational complexity of our kernel functions is worst-case linear in depth, enabling us to explore deep kernel compositions and achieve high test accuracy.

Another line of recent work investigates the connection between kernel methods and infinitely wide neural networks. Jacot et al. (2018) posits that least squares regression with respect to the neural tangent kernel (NTK) is equivalent to optimizing an infinitely wide neural network with gradient flow. Similarly, it has been shown that optimizing just the *last* layer of an infinitely wide neural network is equivalent to a Gaussian process (NNGP) based on the neural network architecture (Lee et al., 2018). Both of these equivalences extend to convolutional neural networks (CNNs) (Li et al., 2019; Novak et al., 2019). The compositional kernels we explore can be expressed as NNGPs.

To construct our compositional kernel functions, we rely on key results from Daniely et al. (2016), which explicitly studies the duality between neural network architectures and compositional kernels.

3. Compositional kernels for bags of features

A variety of data formats are naturally represented by collections of related vectors. For example, an image can be considered a spatially arranged collection of 3-dimensional vectors. A sentence can be represented as a sequence of word embeddings. Audio can be represented as temporally ordered short-time Fourier transforms. In this section, we propose a generalization of these sorts of data types, and a set of operations that allow us to compress these representations into vectors that can be fed into a downstream prediction task. We then show how these operations can be expressed as kernels and describe how to compute them. None of the operations described here are novel, but they form the basic building blocks that we use to build classifiers to compare to neural net architectures.

A *bag of features* is simply a generalization of a matrix or tensor: whereas a matrix is a list of vectors indexed by the natural numbers, a bag of features is a collection of elements in a Hilbert space $\mathcal{H}$ with a finite, structured index set $\mathcal{B}$. As a canonical example, we can consider an image to be a bag of features where the index set $\mathcal{B}$ is the pixel's row and column location and $\mathcal{H}$ is $\mathbb{R}^3$: at every pixel location, there is a corresponding vector in $\mathbb{R}^3$ encoding the color of that pixel. In this section we will denote a generic bag of features by a bold capital letter, e.g., $\boldsymbol{X}$, and the corresponding feature vectors by adding subscripts, e.g., $\boldsymbol{X}_b$. That is, for each index $b \in \mathcal{B}$, $\boldsymbol{X}_b \in \mathcal{H}$.

If our data is represented by a bag of features, we need to map it into a single Hilbert space to perform linear (or nonlinear) predictions. We describe three simple operations to compress a bag of features into a single feature vector.

Concatenation. Let $\mathcal{S}_1, \ldots, \mathcal{S}_L \subseteq \mathcal{B}$ be *ordered* subsets with the same cardinality, s. We write each subset as an ordered set of indices: $\mathcal{S}_j = \{i_{j1}, \ldots, i_{js}\}$. Then we can define a new bag of features $c(\boldsymbol{X})$ with index set $\{1, \ldots, L\}$ and Hilbert space $\mathcal{H}^s$ as follows. For each $j = 1, \ldots, L$, set

$$c(\boldsymbol{X})_j = (\boldsymbol{X}_{i_{j1}}, \boldsymbol{X}_{i_{j2}}, \ldots, \boldsymbol{X}_{i_{js}}) .$$

The simplest concatenation is setting $\mathcal{S}_1 = \mathcal{B}$, which corresponds to vectorizing the bag of features. As we will see, more complex concatenations have strong connections to convolutions in neural networks.

Downsampling. Again let $\mathcal{S}_1, \ldots, \mathcal{S}_L \subseteq \mathcal{B}$ be subsets, but now let them have arbitrary cardinality and order. We can define a new bag of features $p(\boldsymbol{X})$ with index set

$\{1, \ldots, L\}$ and Hilbert space $\mathcal{H}$. For each $j = 1, \ldots, L$ set

$$p(\boldsymbol{X})_j = \frac{1}{|\mathcal{S}_j|} \sum_{i \in \mathcal{S}_j} \boldsymbol{X}_i \,.$$

This is a useful operation for reducing the size of $\mathcal{B}$. Here we use the letter p for the operation as downsampling is commonly called "pooling" in machine learning.

Embedding. Embedding simply means a isomorphism of one Hilbert space to another. Let $\varphi : \mathcal{H} \to \mathcal{H}'$ be a map. Then we can define a new bag of features $\Phi(\boldsymbol{X})$ with index set $\mathcal{B}$ and Hilbert Space $\mathcal{H}'$ by setting

$$\Phi(\boldsymbol{X})_b = \varphi(\boldsymbol{X}_b) \,.$$

Embedding functions are useful for increasing the expressiveness of a feature space.

3.1. Kernels on bags of features

Each operation on a bag of features can be performed directly on the kernel matrix of all feature vectors. Given two bags of features with the same $(\mathcal{B}, \mathcal{H})$, we define the kernel function

$$k(\boldsymbol{X}, a, \boldsymbol{Z}, b) = \langle \boldsymbol{X}_a, \boldsymbol{Z}_b \rangle \,.$$

Note that this implicitly defines a *kernel matrix* between two bags of features: we compute the kernel function for each pair of indices in $\mathcal{B} \times \mathcal{B}$ to form a $|\mathcal{B}| \times |\mathcal{B}|$ matrix. Let us now describe how to implement each of the above operations introduced in Section 3.

Concatenation. Since

$$\langle c(\boldsymbol{X})_j, c(\boldsymbol{Z})_k \rangle = \sum_{\ell=1}^{s} \langle \boldsymbol{X}_{i_{j\ell}}, \boldsymbol{Z}_{i_{k\ell}} \rangle \,,$$

we have

$$k(c(\boldsymbol{X}), j, c(\boldsymbol{Z}), k) = \sum_{\ell=1}^{s} k(\boldsymbol{X}, i_{j\ell}, \boldsymbol{Z}, i_{k\ell}) \,.$$

Downsampling. Similarly, for downsampling, we have

$$k(c(\boldsymbol{X}), j, c(\boldsymbol{Z}), k) = \frac{1}{|\mathcal{S}_j||\mathcal{S}_k|} \sum_{i \in \mathcal{S}_j} \sum_{\ell \in \mathcal{S}_k} k(\boldsymbol{X}, i, \boldsymbol{Z}, \ell) \,.$$

Embedding. Note that the embedding function φ induces a kernel on $\mathcal{H}$. If $\boldsymbol{x}$ and $\boldsymbol{z}$ are elements of $\mathcal{H}$, define

$$k_\varphi(\boldsymbol{x}, \boldsymbol{z}) = \langle \varphi(\boldsymbol{x}), \varphi(\boldsymbol{z}) \rangle \,.$$

Then, we don't need to materialize the embedding function to compute the effect of embedding a bag of features. We only need to know k_φ:

$$k(\Phi(\boldsymbol{X}), j, \Phi(\boldsymbol{Z}), k) = k_\varphi(\boldsymbol{X}_j, \boldsymbol{Z}_k) \,. \tag{1}$$

We will restrict our attention to φ where we can compute $k_\varphi(\boldsymbol{x}, \boldsymbol{z})$ only from $\langle \boldsymbol{x}, \boldsymbol{z} \rangle$, $\|\boldsymbol{x}\|$ and $\|\boldsymbol{z}\|$. This will allow us to iteratively use Equation (1) in cascades of these primitive operations.

3.2. Kernel operations on images

In this section, we specialize kernel operations to operations on images. As described in Section 3, images are bags of three dimensional vectors indexed by two spatial coordinates. Assuming that our images have $D_1 \times D_2$ pixels, we create a sequence of kernels by composing the three operations described above.

Input kernel. The input kernel function k_0 relates all pixel vectors between all pairs of images in our dataset. Computationally, given N images, we can use an image tensor $\boldsymbol{T}$ of shape $N \times D_1 \times D_2 \times 3$ to represent the whole dataset of images, and map this into a kernel tensor $\boldsymbol{K}_{out}$ of shape $N \times D_1 \times D_2 \times N \times D_1 \times D_2$. The elements of $\boldsymbol{K}_{out} = k_0(\boldsymbol{T})$ can be written as:

$$K_{out}[i, j, k, \ell, m, n] = \langle T[i, j, k], T[\ell, m, n] \rangle \,.$$

All subsequent operations operate on 6-dimensional tensors with the same indexing scheme.

Convolution. The convolution operation c_w maps an input tensor $\boldsymbol{K}_{in}$ to an output tensor $\boldsymbol{K}_{out}$ of the same shape: $N \times D_1 \times D_2 \times N \times D_1 \times D_2$. w is an integer denoting the size of the convolution (e.g. $w = 1$ denotes a 3×3 convolution).

The elements of $\boldsymbol{K}_{out} = c_w(\boldsymbol{K}_{in})$ can be written as:

$$K_{out}[i, j, k, \ell, m, n] = \sum_{dx=-w}^{w} \sum_{dy=-w}^{w} K_{in}[i, j + dx, k + dy, \ell, m + dx, n + dy]$$

For out-of-bound location indexes, we simply zero pad the $\boldsymbol{K}_{in}$ so all out-of-bound accesses return zero.

Average pooling. The average pooling operation p_w downsamples the spatial dimension, mapping an input tensor $\boldsymbol{K}_{in}$ of shape $N \times D_1 \times D_2 \times N \times D_1 \times D_2$ to an output tensor $\boldsymbol{K}_{out}$ of shape $N \times (D_1/w) \times (D_2/w) \times N \times (D_1/w) \times (D_2/w)$. We assume D_1 and D_2 are divisible by w.

The elements of $\boldsymbol{K}_{out} = p_w(\boldsymbol{K}_{in})$ can be written as:

$$K_{out}[i, j, k, \ell, m, n] = \frac{1}{w^4} \sum_{a=1}^{w} \sum_{b=1}^{w} \sum_{c=1}^{w} \sum_{d=1}^{w} \Big(K_{in}[i, wj + a, wk + b, \ell, wm + c, wn + d] \Big)$$

Embedding. The nonlinearity layers add crucial nonlinearity to the kernel function, without which the entire map would be linear and much of the benefit of using a kernel method would be lost. We first consider the kernel counterpart of the ReLU activation.

The ReLU embedding, k_{relu}, is shape preserving, mapping an input tensor $\boldsymbol{K}_{in}$ of shape $N \times D_1 \times D_2 \times N \times D_1 \times D_2$ to an output tensor $\boldsymbol{K}_{out}$ of shape $N \times D_1 \times D_2 \times N \times D_1 \times D_2$. To ease the notation, we define two auxiliary tensors: $\boldsymbol{A}$ with shape $N \times D_1 \times D_2$ and $\boldsymbol{B}$ with shape $N \times D_1 \times D_2 \times N \times D_1 \times D_2$, where the elements of each are:

$$A[i,j,k] = \sqrt{K_{in}[i,j,k,i,j,k]}$$
$$B[i,j,k,\ell,m,n] = \arccos\left(\frac{K_{in}[i,j,k,\ell,m,n]}{A[i,j,k]A[\ell,m,n]}\right)$$

The elements of $\boldsymbol{K}_{out} = k_{relu}(\boldsymbol{K}_{in})$ can be written as:

$$K_{out}[i,j,k,\ell,m,n] = \frac{1}{\pi}\Big(A[i,j,k]A[\ell,m,n]\sin(B[i,j,k,\ell,m,n]) + (\pi - B[i,j,k,\ell,m,n])\cos(B[i,j,k,\ell,m,n])\Big)$$

The relationship between the ReLU operator and the ReLU kernel is covered in Subsection 3.3.

In addition to the ReLU kernel, we also work with a normalized Gaussian kernel. The elements of $\boldsymbol{K}_{out} = k_{gauss}(\boldsymbol{K}_{in})$ can be written as:

$$K_{out}[i,j,k,\ell,m,n] = A[i,j,k]A[\ell,m,n]\exp(B[i,j,k,\ell,m,n] - 1)$$

The normalized Gaussian kernel has a similar output response to the ReLU kernel (shown in Figure 1). Experimentally, we find the Gaussian kernel to be marginally faster and more numerically stable.

3.3. Relating compositional kernels to neural network architectures

Each of these compositional kernel operations is closely related to neural net architectures, with close ties to the literature on random features (Rahimi & Recht, 2008). Consider two tensors: $\boldsymbol{U}$ of shape $N \times D_1 \times D_2 \times D_3$ and $\boldsymbol{W}$ of shape $(2w+1) \times (2w+1) \times D_3 \times D_4$. $\boldsymbol{U}$ is the input, which can be N images, w is an integer denoting the size of the convolution (e.g. $w = 1$ denotes a 3×3 convolution), and $\boldsymbol{W}$ is a tensor contains the "weights" of a convolution. Consider a simple convolutional layer followed by a ReLU layer in a neural network:

$$\Psi(\boldsymbol{U}) = \text{relu}(\boldsymbol{W} * \boldsymbol{U})$$

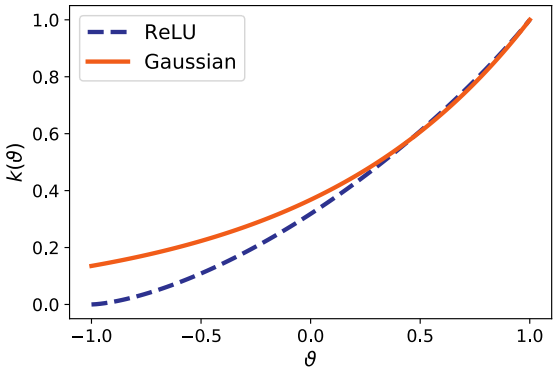

Figure 1. Comparison of the ReLU (arccosine) and Gaussian kernels ($\gamma = 1$), as a function of the angle ϑ between two examples.

where "$*$" denotes the convolution operation and relu denotes elementwise ReLU nonlinearity.

A convolution operation can be rewritten as a matrix multiplication with a reshaping of input tensors. We first flatten the weights tensor $\boldsymbol{W}$ to a matrix $\boldsymbol{W'}$ of D_4 rows and $D_3(2w+1)^2$ columns. For the input tensor $\boldsymbol{U}$, given the convolution size $(2w+1) \times (2w+1)$, we consider the "patch" of each entry $U[n, d_1, d_2, c]$, which includes the $(2w+1) \times (2w+1)$ entries $U[n, i, j, c]$, where $i \in [d_1 - w, d_1 + w]$, $j \in [d_2 - w, d_2 + w]$. Therefore, we can flatten the input tensor $\boldsymbol{U}$ to a matrix $\boldsymbol{U'}$ of size $D_3(2w+1)^2 \times D_1 D_2 N$ by padding all out-of-bounds entries in the patches to zero.

The ReLU operation is shape preserving, applying the ReLU nonlinearity $\varphi(x)$ elementwise to the tensor. Thus we can rewrite the above convolution and ReLU operations into

$$\Psi(\boldsymbol{U}) = \text{relu}(\boldsymbol{W'U'}) = \text{relu}(\boldsymbol{W} * \boldsymbol{U})$$

Therefore, a simple convolution layer and a ReLU layer give us an output tensor $\Psi(\boldsymbol{U})$ of shape $N \times D_1 \times D_2 \times D_4$.

With the help of random features, we are able to relate the above neural network architecture to kernel operations. Suppose the entries of $\boldsymbol{W}$ are appropriately scaled random Gaussian variables. We can evaluate the following expectation according to the calculation in Daniely et al. (2016), thereby relating our kernel construction to inner products between the outputs of *random* neural networks:

$$\mathbb{E}\left[\sum_{c=1}^{D_4} \Psi(\boldsymbol{U})[i,j,k,c]\Psi(\boldsymbol{U})[\ell,m,n,c]\right] = k_{relu}\Big(c_w\big(k_0(\boldsymbol{U})\big)\Big)[i,j,k,\ell,m,n] \tag{2}$$

where k_0 is the input kernel defined in Subsection 3.2. We include the proof for the above equality in the appendix.

Similar calculations can be made for the pooling operation, and for any choice of nonlinearity for which the above expectation can be computed. Moreover, since in Eq (2), the term inside the expectation only depends on inner products, this relation can be generalized to arbitrary depths.

3.4. Implementation

Now we actualize the above formulations into a procedure to generate a kernel matrix from the input data. Let $\mathcal{A}$ be a set of valid neural network operations. A given network architecture $\mathcal{N}$ is represented as an ordered list of operations from $\mathcal{A}$. Let $\mathcal{K}$ denote a mapping from elements of $\mathcal{A}$ to their corresponding operators as defined in Subsection 3.2.

Algorithm 1 defines the procedure for constructing a compositional kernel from a given architecture $\mathcal{N}$ and an input tensor $\boldsymbol{X}$ of N RGB images of shape $N \times D \times D \times 3$. We note that the output kernel is only a $N \times N$ matrix if there exist exactly $\log D$ pooling layers. We emphasize that this procedure is a deterministic function of the input images and network architecture.

Due to memory limitations, in practice we compute the compositional kernel in batches on a GPU. Implementation details are given in Section 4.

Algorithm 1 Compositional Kernel

Input
- $\mathcal{N}$ Input architecture of m layers from $\mathcal{A}$
- $\mathcal{K}$ Map from $\mathcal{A}$ to layerwise operators
- $\boldsymbol{X}$ Tensor of input images, shape $(N \times D \times D \times 3)$

Output
- $\boldsymbol{K}_m$ Compositional kernel matrix, shape $(N \times N)$

$\boldsymbol{K}_0 = k_0(\boldsymbol{X})$
for $i = 1$ **to** m **do**
  $k_i \leftarrow \mathcal{K}(\mathcal{N}_i)$
  $\boldsymbol{K}_i \leftarrow k_i(\boldsymbol{K}_{i-1})$
end for

4. Experiments

In this section, we first provide an overview of the architectures used in our experiments. We then present comparison results between neural networks, NTKs, and compositional kernels on a variety of datasets, including MNIST, CIFAR-10 (Krizhevsky (2009)), CIFAR-10.1 (Recht et al. (2019)), CIFAR-100 (Krizhevsky (2009)) and 90 UCI datasets (Fernández-Delgado et al. (2014)).

4.1. Architectures

We design our deep convolutional kernel based on the non-residual convolutional "Myrtle" networks introduced in Page (2018). We choose this particular network because of its rare combination of simplicity and high performance. Many components commonly used in neural networks, including residual connections, are intended to ease training but have little or unclear effect in terms of the function of the trained network. It is unclear how to model these neural network components in the corresponding kernels, but equally unclear what benefit this might offer. We further simplify the architecture by removing batch normalization and swapping out max pooling with average pooling, for similar reasons. The remaining components are exclusively 3×3 convolutions, 2×2 average pools, and ReLUs. More generally, we refer to all architectures that can be represented as a list of operations from the set $\{$`conv3`, `pool2`, `relu`$\}$ as the "Myrtle" family.

We work with 3 networks from this family: Myrtle5, Myrtle7 and Myrtle10, denoting the depth of each network. An example of the Myrtle5 architecture is shown in Figure 2. The deeper variants have more convolution and ReLU layers; we refer the reader to the appendix for an illustration of the exact architectures. Next we show convolutional neural networks from this family can indeed achieve high accuracy on CIFAR-10, as can their kernel counterparts.

4.2. Experimental setup.

We implemented all the convolutional kernels in the tensor comprehensions framework (Vasilache et al., 2018) and executed them on V100 GPUs using Amazon Web Services (AWS) P3.16xlarge instances. For image classification tasks (MNIST, CIFAR-10, CIFAR-10.1, and CIFAR-100), we used compositional kernels based on the Myrtle family described above. For tabular datasets (90 UCI datasets), we used simpler Gaussian kernels. All experiments on CIFAR-10, CIFAR-10.1 and CIFAR-100 used ZCA whitening as a preprocessing step, except for the comparison experiments explicitly studying preprocessing. We apply "flip" data augmentation to our kernel method by flipping every example in the training set across the vertical axis and constructing a kernel matrix on the concatenation of the flipped and standard datasets.

For all image classification experiments (MNIST, CIFAR-10, CIFAR-10.1, and CIFAR-100) we perform kernel ridge regression with respect to one-hot labels, and solve the optimization problem exactly using a Cholesky factorization. More details are provided in the appendix. For experiments on the UCI datasets, we minimize the hinge loss with libSVM to appropriately compare with prior work (Arora et al., 2020; Fernández-Delgado et al., 2014).

4.3. MNIST

As a "unit test," we evaluate the performance of the compositional kernels in comparison to several baseline methods, including the Gaussian kernel, on the MNIST dataset of

Figure 2. A 5 layer network from the "Myrtle" family (Myrtle5).

handwritten digits (LeCun et al., 1998b). Results are presented in Table 1. We observe that all convolutional methods show nearly identical performance, outperforming the three non-convolutional methods (NTK, arccosine kernel, and Gaussian kernel).

Table 1. Classification performance on MNIST. All methods with convolutional structure have essentially the same performance.

Method	MNIST Accuracy
NTK	98.6
ArcCosine Kernel	98.8
Gaussian Kernel	98.8
Gabor Filters + Gaussian Kernel	99.4
LeNet-5 (LeCun et al., 1998a)	99.0
CKN (Mairal et al., 2014)	99.6
Myrtle5 Kernel	99.5
Myrtle5 CNN	99.5

4.4. CIFAR-10

Table 3 compares the performance of neural networks with various depths and their corresponding compositional kernels on both the 10,000 test images from CIFAR-10 and the additional 2,000 "harder" test images from CIFAR-10.1[1] (Krizhevsky, 2009; Recht et al., 2019). We include the performance of the Gaussian kernel and a standard ResNet32 as baselines. We train all the Myrtle CNNs on CIFAR-10 using SGD and the mean squared error (MSE) loss with multi-step learning rate decay. The exact hyperparameters are provided in the appendix.

We observe that a simple neural network architecture built exclusively from 3×3 convolutions, 2×2 average pooling layers, and ReLU nonlinearities, and trained with only flip augmentation, achieves 93% accuracy on CIFAR-10. The corresponding fixed compositional kernel achieves 90% accuracy on the same dataset, outperforming all previous kernel methods. We note the previous best-performing kernel method from Li et al. (2019) heavily relies on a data dependent feature extraction before data is passed into the kernel function (Coates & Ng, 2012). When additional sources of augmentation are used, such as cutout and random crops, the accuracy of the neural network increases to 96%. Unfortunately due to the quadratic dependence on dataset size, it is currently intractable to augment the compositional kernel to the same extent. For all kernel results[2] on CIFAR-10, we gained a performance improvement of roughly 0.5% using two techniques: Leave-One-Out tilting and ZCA augmentation we detail these techniques in appendix **??**.

Effect of preprocessing. For all of our primary CIFAR-10 experiments, we begin with ZCA pre-processing (Goodfellow et al., 2013). Table 3 also shows the accuracy of our baseline CNN and its corresponding kernel when we replace ZCA with a simpler preprocessing of mean subtraction and standard deviation normalization. We find a substantial drop in accuracy for the compositional kernel without ZCA preprocessing, compared to a much more modest drop in accuracy for the CNN. This result underscores the importance of proper preprocessing for kernel methods; we leave improvements in this area for future work.

4.5. CIFAR-100

For further evaluation, we compute the compositional kernel with the best performance on CIFAR-10 on CIFAR-100. We report our results in Table 2. We find the compositional kernel to be modestly performant on CIFAR-100, matching the accuracy of a CNN of the same architecture when no augmentation is used. However we note this might be due to training instability as the network performed more favorably after flip augmentation was used. Accuracy further increased when batch normalization was added, lending credence to the training instability hypothesis. We also note cross entropy loss was used to achieve the accuracies in Table 2, as we had difficulty optimizing MSE loss on this dataset. We leave further investigations on the intricacies of achieving high accuracy on CIFAR-100 for future work.

4.6. Subsampled CIFAR-10

In this section, we present comparison results in the small dataset regime using subsamples of CIFAR-10, as investigated in Arora et al. (2020). Results are shown in Figure 3. Subsampled datasets are class balanced, and standard deviations are computed over 20 random subsamples, as in Arora et al. (2020). More details are provided in the appendix.

[1] As this dataset was only recently released, some works do not report accuracy on this dataset.

[2] with the exception of the experiment performed without ZCA processing

Table 2. Accuracy on CIFAR-100. All CNNs were trained with cross entropy loss.

Method	CIFAR-100 Accuracy
Myrtle10-Gaussian Kernel	65.3
Myrtle10-Gaussian Kernel + Flips	68.2
Myrtle10 CNN	64.7
Myrtle10 CNN + Flips	71.4
Myrtle10 CNN + BatchNorm	70.3
Myrtle10 CNN + Flips + BatchNorm	74.7

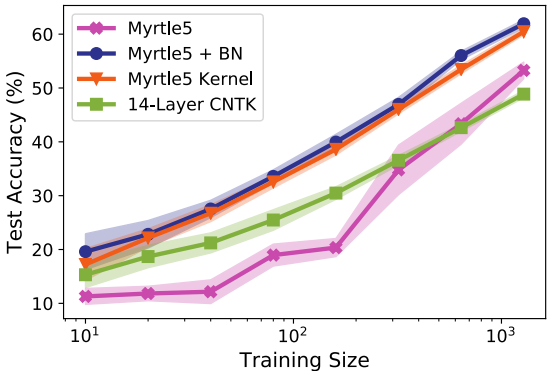

Figure 3. Accuracy results on random subsets of CIFAR-10, with standard deviations over 20 trials. The 14-layer CNTK results are from Arora et al. (2020).

Results. We demonstrate that in the small dataset regime explored in Arora et al. (2020), our convolutional kernels significantly outperform the NTK on subsampled training sets of CIFAR-10. We find a network with the same architecture as our kernel severely underperforms both the compositional kernel and NTK in the low data regime. As with CIFAR-100 we suspect this is a training issue as once we add batch normalization the network outperforms both our kernel and the NTK from Arora et al. (2020).

4.7. UCI datasets

In this section, we present comparison results between the Gaussian kernel and NTK evaluated on 90 UCI datasets, following the setup used in Arora et al. (2020). Arora et al. (2020) identifies that the NTK outperforms a variety of classifiers, including the Gaussian kernel, random forests (RF), and polynomial kernels, evaluated in Fernández-Delgado et al. (2014) on 90 UCI datasets.

Results. For appropriate comparison, we use the same set of 90 "small" UCI datasets (containing no more than 5000 data points) as in Arora et al. (2020) for the evaluations. For the tuning and evaluation procedure we make one crucial

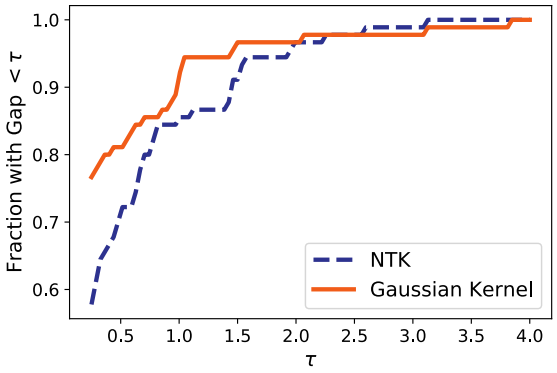

Figure 4. Performance profiles for NTK and tuned Gaussian kernel on 90 UCI datasets.

modification to the evaluation procedure posed in Arora et al. (2020) and Fernández-Delgado et al. (2014). We compute the optimal hyperparameters for each dataset (for both NTK and Gaussian kernel) by averaging performance over *four* cross-validation folds, while both Arora et al. (2020) and Fernández-Delgado et al. (2014) choose optimal hyper parameters on a *single* cross validation fold. Using a single cross validation fold can lead to high variance in final performance, especially when evaluation is done purely on small datasets. A single fold was used in the original experimental setup of Fernández-Delgado et al. (2014) for purely computational reasons, and the authors point out the issue of high variance hyperparameter optimization. Table 4 reports the average cross-validation accuracy over the 90 datasets for the NTK and Gaussian kernel. Compared to results in Arora et al. (2020), the modified evaluation protocol increases the performance of both methods, and the gap between the NTK and Gaussian kernel disappears.

We compute the same metrics used in Arora et al. (2020): Friedman rank, P90, P95 and PMA, where a better classifier is expected to have lower Friedman rank and higher P90, P95, and PMA. The average accuracy is reported together with its standard deviation. Friedman rank denotes the ranking metric introduced to compare classifiers across multiple datasets in Demšar (2006), and reports the average ranking of a given classifier compared to all other classifiers. P90/P95 denotes the percentage of datasets on which the classifier achieves at least $90\%/95\%$ of the maximum accuracy across all classifiers for this dataset. PMA denotes the average percentage of the maximum accuracy across all classifiers for each dataset.

On all metrics reported by Arora et al. (2020), the Gaussian kernel has comparable or better performance relative to the NTK. Figure 4 shows a performance profile to visually compare the two classifiers (Dolan & Moré, 2002). For a given τ, the y axis denotes the fraction of instances where

Table 3. Classification performance on CIFAR-10.

Method	CIFAR-10 Accuracy	CIFAR-10.1 Accuracy
Gaussian Kernel	57.4	-
CNTK + Flips (Li et al., 2019)	81.4	-
CNN-GP + Flips (Li et al., 2019)	82.2	-
CKN (Mairal, 2016)	85.8	-
Coates-NG + Flips (Recht et al., 2019)	85.6	73.1
Coates-NG + CNN-GP + Flips (Li et al., 2019)	88.9	-
ResNet32	92.5	84.4
Myrtle5 Kernel + No ZCA	77.7	62.2
Myrtle5 Kernel	85.8	71.6
Myrtle7 Kernel	86.6	73.1
Myrtle10 Kernel	87.5	74.5
Myrtle10-Gaussian Kernel	88.2	75.1
Myrtle10-Gaussian Kernel + Flips	89.8	78.3
Myrtle5 CNN + No ZCA	87.8	75.8
Myrtle5 CNN	89.8	79.0
Myrtle7 CNN	90.2	79.7
Myrtle10 CNN	91.2	79.9
Myrtle10 CNN + Flips	93.4	84.8
Myrtle10 CNN + Flips + CutOut + Crops	96.0	89.8

Table 4. Results on 90 UCI datasets for the NTK and Gaussian kernel (both tuned over 4 eval folds).

Classifier	Friedman Rank	Average Accuracy (%)	P90 (%)	P95 (%)	PMA (%)
SVM NTK	14.3	83.2 ± 13.5	96.7	83.3	97.3 ± 3.8
SVM Gaussian kernel	11.6	83.4 ± 13.4	95.6	83.3	97.5 ± 3.7

a classifier either has the highest accuracy or has accuracy within τ of the best accuracy. The performance profile reveals that the Gaussian kernel and NTK perform quite comparably on the 90 UCI datasets.

5. Limitations and Future Work

The compositional kernels proposed in this manuscript significantly advance the state of the art of kernel methods applied to pattern recognition tasks. However, these kernels still have significant limitations that must be addressed before they can be applied in practice.

Computational cost. The compositional kernels we study compare all pairs of input pixels for two images with D pixels each, so the cost of evaluating the kernel function on two data points is $\tilde{O}(D^2)$. In addition, $O(N^2)$ kernel evaluations must be computed to construct the full kernel matrix, creating a total complexity of $\tilde{O}(N^2D^2)$. Even with heavily optimized GPU code, this requires significant computation time. We therefore limited our scope to image datasets with a small pixel count and modest number of examples: CIFAR-10/CIFAR-100 consist of $60,000$ $32 \times 32 \times 3$ images and MNIST consists of $70,000$ 28×28 images. Even with this constraint, the largest compositional kernel matrices we study took approximately 1000 GPU hours to compute. Thus, we believe an imperative direction of future work is reducing the complexity of each kernel evaluation. Random feature methods or other compression schemes could play a significant role here.

Once a kernel matrix is constructed, exact minimization of empirical risk often scales as $O(N^3)$. For datasets with less than 100,000 examples, these calculations can be performed relatively quickly on standard workstations with sufficient RAM. However, even these solves are expensive for larger datasets. Fortunately, recent work on kernel optimization (Ma & Belkin, 2018; Dai et al., 2014; Shankar et al., 2018; Wang et al., 2019) paves a way to scale our approach to larger datasets.

Data augmentation. A major advantage of neural networks is that data augmentation can be added essentially for free. For kernel methods, data augmentation requires

treating each augmented example as if it was part of the data set, and hence computation scales superlinearly with the amount of augmentation: if one wants to perform 100 augmentations per example, then the final kernel matrix will be 10,000 times larger, and solving the prediction problem may be one million times slower. Finding new paths to cheaply augment kernels (Ratner et al., 2017; Dao et al., 2019) or to incorporate the symmetries implicit in data augmentation explicitly in kernels should dramatically improve the effectiveness of kernel methods on contemporary datasets. One promising avenue is augmentation via kernel ensembling, e.g. by forming many smaller kernels with augmented data and averaging their predictions appropriately.

Architectural modifications. We consider a simple set of architectural building blocks (convolution, average pool, and ReLU) in this work, but there exist several commonly used primitives in deep networks that have no clear analogues for kernel machines (e.g residual connections, max pool, batch normalization, etc.). While it is unclear whether these primitives are necessary, the question remains open whether the performance gap between kernels and neural networks indicates a fundamental limitation of kernel methods or merely an engineering hurdle that can be overcome (e.g. with improved architectures or by additional subunits).

6. Acknowledgements

We would like to thank Achal Dave for his insights on accelerating kernel operations for the GPU and Eric Jonas for his guidance on parallelizing our kernel operations with AWS Batch. We would additionally like to thank Rebecca Roelofs, Stephen Tu, Horia Mania, Scott Shenker, and Shivaram Venkataraman for helpful discussions and comments on this work.

This research was generously supported in part by ONR awards N00014-17-1-2191, N00014-17-1-2401, and N00014-18-1-2833, the DARPA Assured Autonomy (FA8750-18-C-0101) and Lagrange (W911NF-16-1-0552) programs, a Siemens Futuremakers Fellowship, an Amazon AWS AI Research Award. SFK was supported by an NSF graduate student fellowship.

References

Allen-Zhu, Z., Li, Y., and Liang, Y. Learning and generalization in overparameterized neural networks, going beyond two layers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Arora, S., Du, S. S., Hu, W., Li, Z., Salakhutdinov, R., and Wang, R. On exact computation with an infinitely wide neural net. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Arora, S., Du, S. S., Li, Z., Salakhutdinov, R., Wang, R., and Yu, D. Harnessing the power of infinitely wide deep nets on small-data tasks. In *International Conference on Learning Representations (ICLR)*, 2020.

Coates, A. and Ng, A. Y. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pp. 561–580. Springer, 2012.

Dai, B., Xie, B., He, N., Liang, Y., Raj, A., Balcan, M., and Song, L. Scalable kernel methods via doubly stochastic gradients. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Daniely, A., Frostig, R., and Singer, Y. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Dao, T., Gu, A., Ratner, A. J., Smith, V., Sa, C. D., and Re, C. A kernel theory of modern data augmentation. In *International Conference on Machine Learning (ICML)*, 2019.

Demšar, J. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research (JMLR)*, 7(Jan):1–30, 2006.

Dolan, E. D. and Moré, J. J. Benchmarking optimization software with performance profiles. *Mathematical Programming, Series A*, 91:201–213, 2002.

Du, S. S., Lee, J. D., Li, H., Wang, L., and Zhai, X. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning (ICML)*, 2019a.

Du, S. S., Zhai, X., Poczos, B., and Singh, A. Gradient descent provably optimizes over-parameterized neural networks. In *International Conference on Learning Representations (ICLR)*, 2019b.

Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research (JMLR)*, 15(1):3133–3181, 2014.

Ghorbani, B., Mei, S., Misiakiewicz, T., and Montanari, A. Linearized two-layers neural networks in high dimension. *arXiv preprint arXiv:1904.12191*, 2019. URL `http://arxiv.org/abs/1904.12191`.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A. C., and Bengio, Y. Maxout networks. In *International Conference on Machine Learning (ICML)*, 2013.

Jacot, A., Hongler, C., and Gabriel, F. Neural tangent kernel: Convergence and generalization in neural networks.

In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009.

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

LeCun, Y., Cortes, C., and Burges, C. The mnist dataset of handwritten digits, 1998b.

Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-Dickstein, J. Deep neural networks as gaussian processes. In *International Conference on Learning Representations (ICLR)*, 2018.

Lee, J., Xiao, L., Schoenholz, S., Bahri, Y., Novak, R., Sohl-Dickstein, J., and Pennington, J. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Li, Z., Wang, R., Yu, D., Du, S. S., Hu, W., Salakhutdinov, R., and Arora, S. Enhanced convolutional neural tangent kernels. *arXiv preprint arXiv:1911.00809*, 2019. URL https://arxiv.org/abs/1911.00809.

Ma, S. and Belkin, M. Kernel machines that adapt to gpus for effective large batch training. *arXiv preprint arXiv:1806.06144*, 2018. URL https://arxiv.org/abs/1806.06144.

Mairal, J. End-to-end kernel learning with supervised convolutional kernel networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1399–1407. Curran Associates, Inc., 2016.

Mairal, J., Koniusz, P., Harchaoui, Z., and Schmid, C. Convolutional kernel networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Novak, R., Xiao, L., Bahri, Y., Lee, J., Yang, G., Hron, J., Abolafia, D. A., Pennington, J., and Sohl-Dickstein, J. Bayesian deep convolutional networks with many channels are gaussian processes. In *International Conference on Learning Representations (ICLR)*, 2019.

Page, D. myrtle.ai, 2018. URL https://myrtle.ai/how-to-train-your-resnet-4-architecture/.

Rahimi, A. and Recht, B. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1177–1184, 2008.

Ratner, A., Ehrenberg, H., Hussain, Z., Dunnmon, J., and Re, C. Learning to compose domain-specific transformations for data augmentation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Recht, B., Roelofs, R., Schmidt, L., and Shankar, V. Do imagenet classifiers generalize to imagenet? In *International Conference on Machine Learning (ICML)*, 2019.

Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., and Ragan-Kelley, J. numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018. URL http://arxiv.org/abs/1810.09679.

Shawe-Taylor, J., Cristianini, N., et al. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018. URL https://arxiv.org/abs/1802.04730.

Wang, K. A., Pleiss, G., Gardner, J., Tyree, S., Weinberger, K., and Wilson, A. G. Exact gaussian processes on a million data points. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.