NEURAL TANGENTS:
FAST AND EASY INFINITE NEURAL NETWORKS IN PYTHON

Anonymous authors
Paper under double-blind review

ABSTRACT

NEURAL TANGENTS is a library for working with infinite-width neural networks. It provides a high-level API for specifying complex and hierarchical neural network architectures. These networks can then be trained and evaluated either at finite-width as usual, or in their infinite-width limit. For the infinite-width networks, NEURAL TANGENTS performs exact inference either via Bayes' rule or gradient descent, and generates the corresponding Neural Network Gaussian Process and Neural Tangent kernels. Additionally, NEURAL TANGENTS provides tools to study gradient descent training dynamics of wide but finite networks.

The entire library runs out-of-the-box on CPU, GPU, or TPU. All computations can be automatically distributed over multiple accelerators with near-linear scaling in the number of devices. NEURAL TANGENTS is available at

www.github.com/neural-tangents/neural-tangents

We also provide an accompanying interactive **Colab notebook**.[1]

1 INTRODUCTION

Deep neural networks (DNNs) owe their success in part to the broad availability of high-level, flexible, and efficient software libraries like Tensorflow (Abadi et al., 2015), Keras (Chollet et al., 2015), PyTorch.nn (Paszke et al., 2017), Chainer (Tokui et al., 2015; Akiba et al., 2017), JAX (Frostig et al., 2018a), and others. These libraries enable researchers to rapidly build complex models by constructing them out of smaller primitives. The success of new machine learning approaches will similarly depend on developing sophisticated software tools to support them.

1.1 INFINITE-WIDTH BAYESIAN NEURAL NETWORKS

Recently, a new class of machine learning models has attracted significant attention, namely, deep *infinitely wide* neural networks. In the infinite-width limit, a large class of Bayesian neural networks become Gaussian Procesesses (GPs) with a specific, architecture-dependent, compositional kernel; these models are called Neural Network Gaussian Processes (NNGPs). This correspondence was first established for shallow fully-connected networks by Neal (1994) and was extended to multi-layer setting in (Lee et al., 2018; Matthews et al., 2018b). Since then, this correspondence has been expanded to a wide range of nonlinearities (Matthews et al., 2018a; Novak et al., 2019) and architectures including those with convolutional layers (Garriga-Alonso et al., 2019; Novak et al., 2019), residual connections (Garriga-Alonso et al., 2019), and pooling (Novak et al., 2019). The results for individual architectures have subsequently been generalized, and it was shown that a GP correspondence holds for a general class of networks that can be mapped to so-called *tensor programs* in (Yang, 2019). The recurrence relationship defining the NNGP kernel has additionally been studied separately from the GP correspondence in (Cho & Saul, 2009; Daniely et al., 2016; Poole et al., 2016; Schoenholz et al., 2016; Yang & Schoenholz, 2017; Xiao et al., 2018).

[1] www.colab.sandbox.google.com/github/neural-tangents/neural-tangents/blob/master/notebooks/neural_tangents_cookbook.ipynb

1.2 Infinite-width neural networks trained by gradient descent

In addition to enabling a closed form description of Bayesian neural networks, the infinite-width limit has also very recently provided insights into neural networks trained by gradient descent. In the last year, several papers have shown that randomly initialized neural networks trained with gradient descent are characterized by a distribution that is related to the NNGP, and is described by the so-called Neural Tangent Kernel (NTK) (Jacot et al., 2018; Lee et al., 2019; Chizat et al., 2019), a kernel which was implicit in some earlier papers (Li & Liang, 2018; Allen-Zhu et al., 2018; Du et al., 2018a;b). In addition to this "function space" perspective, a dual, "weight space" view on the wide network limit was proposed in Lee et al. (2019) which showed that networks under gradient descent were well-described by the first-order Taylor series about their initial parameters.

1.3 Promise and practical barriers to working with infinite-width networks

Combined, these discoveries established infinite-width networks as useful theoretical tools to understand a wide range of phenomena in deep learning. Furthermore, the practical utility of these models has been proven by achieving state-of-the-art performance on image classification benchmarks among GPs without trainable kernels (Garriga-Alonso et al., 2019; Novak et al., 2019; Arora et al., 2019), and by their ability to match or exceed the performance of finite width networks in some situations, especially for fully- and locally-connected model families (Lee et al., 2018; Novak et al., 2019).

However, despite their utility, using NNGPs and NTK-GPs is arduous and can require weeks-to-months of work by seasoned practitioners. Kernels corresponding to neural networks must be derived by hand on a per-architecture basis. Moreover, to make the code performant, it may be necessary to introduce custom, specialized, CUDA kernels (Arora et al., 2019). Overall, this process is laborious and error prone, and is reminiscent of the state of neural networks before high quality Automatic Differentiation (AD) packages proliferated.

1.4 Summary of contributions

In this paper, we introduce a new open-source software library called Neural Tangents targeting JAX (Frostig et al., 2018a) to facilitate research on infinite limits of neural networks. The main features of Neural Tangents are:

- A high-level neural network API for specifying complex, hierarchical, models. Networks specified using this API can have their infinite-width NNGP kernel and NTK evaluated analytically.
- Functions to approximate infinite-width kernels by Monte Carlo sampling for networks whose kernels cannot be constructed analytically. These methods are agnostic to the neural network library used to build the network and are therefore quite versatile.
- An API to analytically perform inference using infinite-width networks either by computing the Bayesian posterior or by computing the result of continuous gradient descent with an MSE loss. The API additionally includes tools to perform inference by numerically solving the ODEs corresponding to: gradient descent, with-or-without momentum, on arbitrary loss functions, at finite or infinite time.
- Functions to compute arbitrary-order Taylor series approximations to neural networks about a given setting of parameters to explore the weight space perspective on the infinite-width limit.
- Leveraging XLA, our library runs out-of-the-box on CPU, GPU, or TPU. Kernel computations can automatically be distributed over multiple accelerators with near-perfect scaling (Figure 5, right).

We begin with three short examples (§2) that demonstrate the ease, efficiency, and versatility of performing calculations with infinite networks using Neural Tangents. With a high level view of the library in hand, we then dive into a number of technical aspects of our library (§3).

1.5 BACKGROUND

We briefly describe the NNGP (§1.1) and NTK (§1.2). **NNGP.** Neural networks are often structured as linear transformations followed by pointwise applications of non-linearities. Let $z^l(x)$ describe the pre-activations following a linear transformation in l^{th} layer of a neural network. At initialization, the parameters of the network are randomly distributed and so central-limit theorem style arguments can be used to show that the pre-activations become Gaussian distributed with mean zero and are therefore described entirely by their covariance matrix $\mathcal{K}(x, x') = \mathbb{E}[z^l(x)z^l(x')]$. One can therefore use the NNGP to make Bayesian posterior predictions which are Gaussian distributed with mean $\mathcal{K}(x, \mathcal{X})\mathcal{K}(\mathcal{X}, \mathcal{X})^{-1}\mathcal{Y}$ and covariance $\mathcal{K}(x, x) - \mathcal{K}(x, \mathcal{X})\mathcal{K}(\mathcal{X}, \mathcal{X})^{-1}\mathcal{K}(\mathcal{X}, x)$, where $(\mathcal{X}, \mathcal{Y})$ is the training set of inputs and targets. **NTK.** When neural networks are optimized using continuous gradient descent with learning rate η on mean squared error (MSE) loss, the function evaluated on training points evolves as $\partial_t f_t(\mathcal{X}) = -\eta J_t(\mathcal{X})J_t(\mathcal{X})^T (f_t(\mathcal{X}) - \mathcal{Y})$ where $J_t(\mathcal{X})$ is the Jacobian of f evaluated at $\mathcal{X}$ and $\Theta_t(\mathcal{X}, \mathcal{X}) = J_t(\mathcal{X})J_t(\mathcal{X})^T$ is the NTK. In the infinite-width limit, the NTK remains constant ($\Theta_t = \Theta$) throughout training and the time-evolution of the outputs can be solved in closed form as a Gaussian with mean $f_t(x) = \Theta(x, \mathcal{X})\Theta(\mathcal{X}, \mathcal{X})^{-1} (I - \exp[-\eta\Theta(\mathcal{X}, \mathcal{X})t])\, \mathcal{Y}$.

2 EXAMPLES

We begin by applying NEURAL TANGENTS to several example tasks. While these tasks are designed for pedagogy rather than research novelty, they are nonetheless emblematic of problems regularly faced in research. We emphasize that without NEURAL TANGENTS, it would be necessary to derive the kernels for each architecture *by hand.*

2.1 INFERENCE WITH AN INFINITELY WIDE NEURAL NETWORK

We begin by training an infinitely wide neural network with gradient descent and comparing the result to training an ensemble of wide-but-finite networks. This example is worked through in detail in the **Colab notebook**.[2]

We train on a synthetic dataset with training data drawn from the process $y_i = \sin(x_i) + \epsilon_i$ with $x_i \sim \text{Uniform}(-\pi, \pi)$ and $\epsilon_i \sim \mathcal{N}(0, \sigma)$ independently and identically distributed. To train an infinite neural network with Erf[3] on this data using gradient descent and an MSE loss we write the following:

```
from neural_tangents import predict, stax

init_fn, apply_fn, kernel_fn = stax.serial(
    stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
    stax.Dense(2048, W_std=1.5, b_std=0.05), stax.Erf(),
    stax.Dense(1, W_std=1.5, b_std=0.05))

y_mean, y_var = predict.gp_inference(kernel_fn, x_train, y_train, x_test, 'ntk',
                                     diag_reg=1e-4, compute_var=True)
```

The above code analytically generates the predictions that would result from performing gradient descent for an infinite amount of time. However, it is often desirable to investigate finite-time learning dynamics of deep networks. This is also supported in NEURAL TANGENTS as illustrated in the following snippet:

```
predict_fn = predict.gradient_descent_mse_gp(kernel_fn, x_train, y_train, x_test, 'ntk',
                                             diag_reg=1e-4, compute_var=True)
y_mean, y_var = predict_fn(t=100)  # Predict the distribution at t = 100.
```

[2] www.colab.sandbox.google.com/github/neural-tangents/neural-tangents/blob/master/notebooks/neural_tangents_cookbook.ipynb

[3] Error function, a nonlinearity similar to $\tanh$; see §H for other implemented nonlinearities, including Relu.

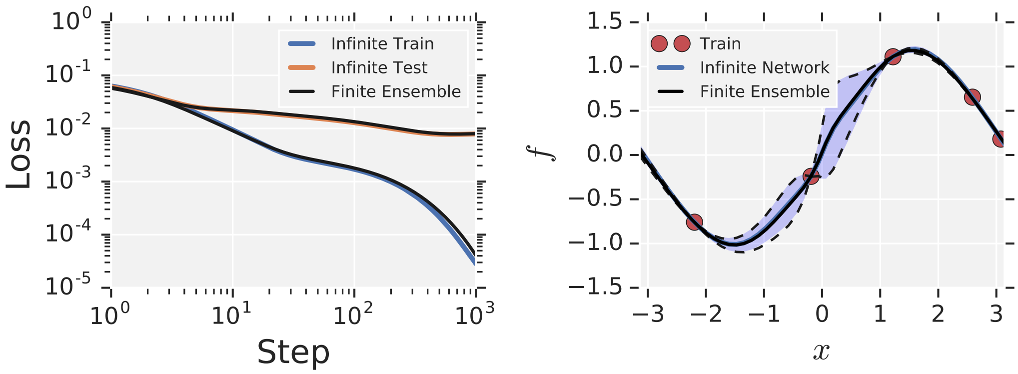


Figure 1: **Training dynamics for an ensemble of finite-width networks compared with an infinite network. Left:** Mean train and test MSE loss evolution throughout training. **Right:** Comparison between the predictions of the trained infinite network and the respective ensemble of finite-width networks. The shaded region and the dashed lines denote two standard deviations of uncertainty in the predictions for the infinite network and the ensemble respectively.

The above specification set the hidden layer widths to 2048, which has no effect on the infinite width network inference, but the `init_fn` and `apply_fn` here correspond to ordinary finite width networks. In Figure 1 we compare the result of this exact inference with training an ensemble of one hundred of these finite-width networks by looking at the training curves and output predictions of both models. We see excellent agreement between exact inference using the infinite-width model and the result of training an ensemble using gradient descent.

2.2 An Infinitely WideResNet

The above example considers a relatively simple network on a synthetic task. In practice we may want to consider real-world architectures, and see how close they are to their infinite-width limit. For this task we study a variant of an infinite-channel Wide Residual Network (Zagoruyko & Komodakis, 2016) (WRN-28-∞). We first define both finite and infinite models within Listing 1.

We now study how quickly the kernel of the finite-channel WideResNet approaches it's infinite channel limit. We explore two different axes along which convergence takes place: first, as a function of the number of channels (as measured by the widening factor, k) and second as a function of the number of finite-network Monte Carlo samples we average over. Neural Tangents makes it easy to compute MC averages of finite kernels using the following snippet:

```
kernel_fn = nt.monte_carlo_kernel_fn(init_fn, apply_fn, rng_key, n_samples)
sampled_kernel = kernel_fn(x, x)
```

The convergence is shown in Figure 2. We see that as both the number of samples is increased or the network is made wider, the empirical kernel approaches the kernel of the infinite network. As remarked in Novak et al. (2019), for any finite widening factor the MC estimate is biased, but here the bias is small relative to the variance, and the distance to the empirical kernel decreases with the number of samples.

2.3 Comparison of Neural Network architectures and training set sizes

The above examples demonstrated the flexibility Neural Tangents provides in comparing various forms of inference for the same architecture. Now, we further leverage this flexibility to demonstrate training a range of architectures on CIFAR-10 and compare their performance as a function of dataset size. In particular, we compare a fully-connected network, a convolutional network whose penultimate layer vectorizes the image, and the wide-residual network described above. In each case, we perform exact infinite-time inference using the analytic infinite-width NTK-GP.

```python
from neural_tangents import stax
import neural_tangents as nt

def WideResNetBlock(channels, strides=(1, 1), channel_mismatch=False):
  Main = stax.serial(stax.Relu(), stax.Conv(channels, (3, 3), strides, padding='SAME'),
                     stax.Relu(), stax.Conv(channels, (3, 3), padding='SAME'))
  Shortcut = (stax.Identity() if not channel_mismatch else
              stax.Conv(channels, (3, 3), strides, padding='SAME'))
  return stax.serial(stax.FanOut(2), stax.parallel(Main, Shortcut), stax.FanInSum())

def WideResNetGroup(n, channels, strides=(1, 1)):
  blocks = [WideResNetBlock(channels, strides, channel_mismatch=True)]
  for _ in range(n - 1):
    blocks += [WideResNetBlock(channels, (1, 1))]
  return stax.serial(*blocks)

def WideResNet(block_size, k, num_classes):
  return stax.serial(stax.Conv(16, (3, 3), padding='SAME'),
                     WideResNetGroup(block_size, int(16 * k)),
                     WideResNetGroup(block_size, int(32 * k), (2, 2)),
                     WideResNetGroup(block_size, int(64 * k), (2, 2)),
                     stax.GlobalAvgPool(), stax.Dense(num_classes))

init_fn, apply_fn, kernel_fn = WideResNet(block_size=4, k=1, num_classes=10)
```

Listing 1: **Definition of an infinitely WideResNet**. This snippet simultaneously defines a finite (`init_fn, apply_fn`) and an infinite (`kernel_fn`) model. This model is used in Figures 2 and 3.

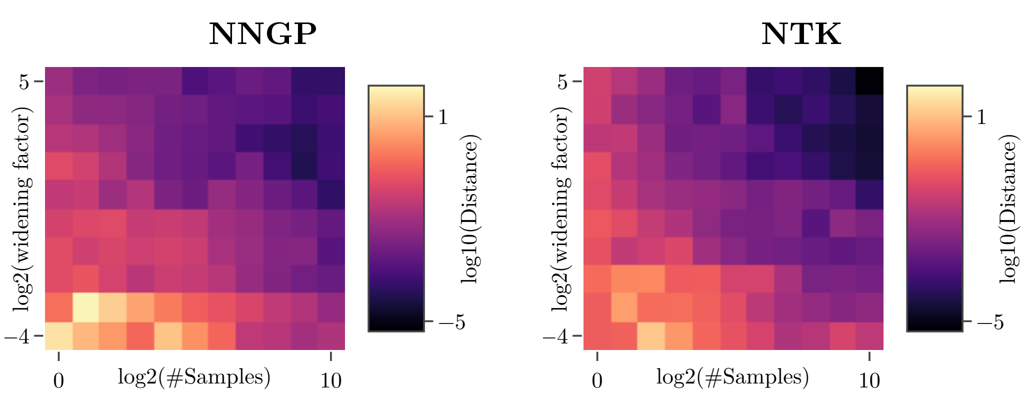


Figure 2: **Convergence of the Monte Carlo (MC) estimates** of the WideResNet WRN-28-k (where k is the widening factor) NNGP and NTK kernels (computed with `monte_carlo_kernel_fn`) to their analytic values (WRN-28-∞, computed with `kernel_fn`), as the network gets wider by increasing the widening factor (vertical axis) and as more random networks are averaged over (horizontal axis). **Experimental detail.** The kernel is computed in 32-bit precision on a 100×50 batch of 8×8-downsampled CIFAR10 (Krizhevsky, 2009) images. For sampling efficiency, for NNGP the output of the penultimate layer was used, and for NTK the output layer was assumed to be of dimension 1 (all logits are i.i.d. conditioned on a given input). The displayed distance is the relative Frobenius norm squared, i.e. $\|\mathcal{K} - \mathcal{K}_{k,n}\|_{\mathrm{F}}^2 \,/\, \|\mathcal{K}\|_{\mathrm{F}}^2$, where k is the widening factor and n is the number of samples.

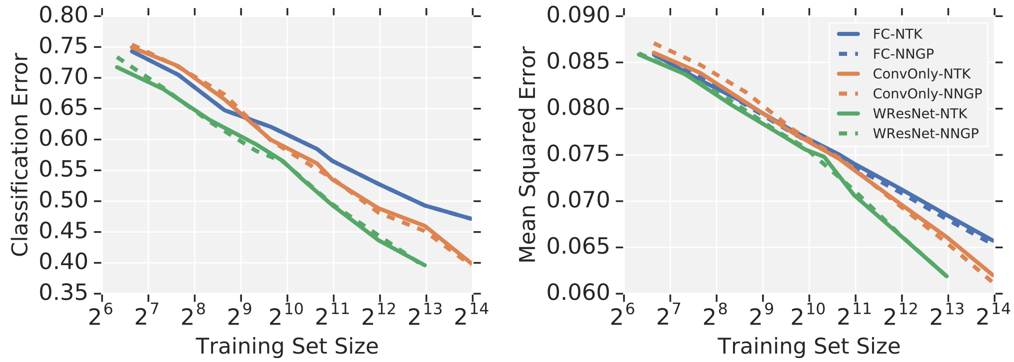


Figure 3: **CIFAR-10 classification with varying neural network architectures.** NEURAL TANGENTS simplify experimentation with architectures. Here we use infinite time NTK inference and full Bayesian NNGP inference for CIFAR-10 for Fully Connected (FC), Convolutional network without pooling (ConvOnly) and Wide Residual Network (WideResNet). See Listing 1 for WideResNet, 2 for ConvOnly, and 3 for FC model definitions.

The results are shown in Figure 3. We see that in each case the performance of the model increases approximately logarithmically in the size of the dataset. Moreover, we observe a clear hierarchy of performance, especially at large dataset size, in terms of architecture (FC < ConvOnly < WideResNet).

3 IMPLEMENTATION: TRANSFORMING TENSOR OPS TO KERNEL OPS

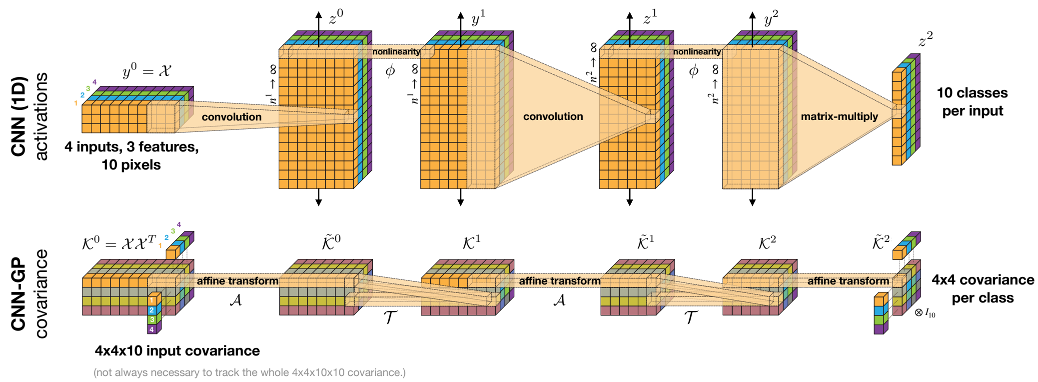
Neural networks are compositions of basic tensor operations such as: dense or convolutional affine transformations, application of pointwise nonlinearities, pooling, or normalization. For most networks without weight tying between layers, the kernel computation can be written compositionally, with a direct correspondence between each tensor operation and a kernel operation. The core logic of NEURAL TANGENTS is thus a set of *translation rules*, that sends each function acting on a finite-width layer to a function acting on the kernel for an infinite-width network. This is illustrated in Figure 4 for a simple convolutional architecture. The function applied to the data tensor is given in the second column, and the corresponding transformations of the NTK and NNGP kernel tensors are given in the third and fourth column. See §H for a list of all tensor operations for which translation rules are currently implemented.

One subtlety to consider when designing networks is that most infinite-width results require nonlinear transformations to be prefaced with affine transformations (either dense or convolutional). This is because infinite-width results often assume that the pre-activations of nonlinear layers are approximately Gaussian. Randomness in weights and biases causes the output of infinite affine layers to satisfy this Gaussian requirement. Fortunately, prefacing nonlinear operations with affine transformations is common practice when designing neural networks, and NEURAL TANGENTS will raise an error if this requirement is not satisfied.

3.1 PERFORMANCE

Our library does a number of automatic performance optimizations without sacrificing flexibility.

Leveraging block-diagonal covariance structure. A common computational challenge with GPs is inverting the training set covariance matrix. Naively, for a classification task with C classes and training set $\mathcal{X}$, NNGP and NTK covariances have the shape of $|\mathcal{X}|\,C \times |\mathcal{X}|\,C$. For CIFAR-10, this would be $500,000 \times 500,000$. However, we highlight that if a fully-connected readout layer is used (which is an extremely common design in classification architectures), the C logits are i.i.d. conditioned on the input x, resulting in outputs being normally distributed with a block-diagonal covariance matrix of form $\Sigma \otimes I_C$, where Σ has the shape $|\mathcal{X}| \times |\mathcal{X}|$, which makes closed-form exact inference feasible in these tasks.



Layer	**Tensor Op**	**NNGP Op**	**NTK Op**
0 (input)	$y^0 = \mathcal{X}$	$\mathcal{K}^0 = \mathcal{X}\mathcal{X}^T$	$\Theta^0 = 0$
0 (pre-activations)	$z^0 = \text{Conv}\left(y^0\right)$	$\tilde{\mathcal{K}}^0 = \mathcal{A}\left(\mathcal{K}^0\right)$	$\tilde{\Theta}^0 = \tilde{\mathcal{K}}^0 + \mathcal{A}\left(\Theta^0\right)$
1 (activations)	$y^1 = \phi\left(z^0\right)$	$\mathcal{K}^1 = \mathcal{T}\left(\tilde{\mathcal{K}}^0\right)$	$\Theta^1 = \dot{\mathcal{T}}\left(\tilde{\mathcal{K}}^0\right) \odot \tilde{\Theta}^0$
1 (pre-activations)	$z^1 = \text{Conv}\left(y^1\right)$	$\tilde{\mathcal{K}}^1 = \mathcal{A}\left(\mathcal{K}^1\right)$	$\tilde{\Theta}^1 = \tilde{\mathcal{K}}^1 + \mathcal{A}\left(\Theta^1\right)$
2 (activations)	$y^2 = \phi\left(z^1\right)$	$\mathcal{K}^2 = \mathcal{T}\left(\tilde{\mathcal{K}}^1\right)$	$\Theta^2 = \dot{\mathcal{T}}\left(\tilde{\mathcal{K}}^1\right) \odot \tilde{\Theta}^1$
2 (readout)	$z^2 = \text{Dense} \circ \text{Flatten}\left(y^2\right)$	$\tilde{\mathcal{K}}^2 = \text{Tr}\left(\mathcal{K}^2\right)$	$\tilde{\Theta}^2 = \tilde{\mathcal{K}}^2 + \text{Tr}\left(\Theta^2\right)$

Figure 4: **An example of the translation of a convolutional neural network into a sequence of kernel operations.** We demonstrate how the compositional nature of a typical NN computation on its inputs induces a corresponding compositional computation on the NNGP and NT kernels. Presented is a 2-hidden-layer 1D CNN with nonlinearity ϕ, performing regression on the 10-dimensional outputs z^2 for each of the 4 (**1**, **2**, **3**, **4**) inputs x from the dataset $\mathcal{X}$. To declutter notation, unit weight and zero bias variances are assumed in all layers. **Top:** recursive output (z^2) computation in the CNN (top) induces a respective recursive NNGP kernel ($\tilde{\mathcal{K}}^2 \otimes I_{10}$) computation (NTK computation being similar, not shown). **Bottom:** explicit listing of tensor and corresponding kernel ops in each layer. See Table 1 for operation definitions. Illustration and description adapted from Figure 3 in Novak et al. (2019) with the authors' permission.

Automatically tracking only the smallest necessary subset of intermediary covariance entries. For most architectures, especially convolutional, the main computational burden lies in *constructing* the covariance (as opposed to inverting it), as construction of the $|\mathcal{X}| \times |\mathcal{X}|$ output covariance Σ involves computing intermediary layer l covariances Σ^l of size $|\mathcal{X}|\, d \times |\mathcal{X}|\, d$ (see Listing 1 for a model requiring this computation), where d is the total number of pixels in the intermediary layer outputs, which is 1024 in the case of CIFAR-10 and `SAME` padding. However, as Xiao et al. (2018); Novak et al. (2019); Garriga-Alonso et al. (2019) remarked, if no pooling is used in the network the output covariance Σ can be computed by only using the stack of d $|\mathcal{X}| \times |\mathcal{X}|$-blocks of Σ^l, bringing the time and memory cost from $\mathcal{O}(|\mathcal{X}|^2 d^2)$ down to $\mathcal{O}(|\mathcal{X}|^2 d)$ per layer (see Figure 4 and Listing 2 for models admitting this optimization). Finally, if the network has no convolutional layers, the cost further reduces to $\mathcal{O}(|\mathcal{X}|^2)$ (see Listing 3 for an example). These choices are performed automatically by NEURAL TANGENTS to achieve efficient computation and minimal memory footprint.

Expressing covariance computations as 2D convolutions with optimal layout. A key insight to high performance in convolutional models is that the covariance propagation operator for convolutional layers $\mathcal{A}$ can be expressed in terms of 2D convolutions when it operates on both the full $|\mathcal{X}|\, d \times |\mathcal{X}|\, d$ covariance matrix Σ, and on the d diagonal $|\mathcal{X}| \times |\mathcal{X}|$-blocks. This allows utilization of modern hardware accelerators, many of which target 2D convolutions as their primary machine learning application.

Simultaneous NNGP and NT kernel computations. As NTK computation requires the NNGP covariance as an intermediary computation, the NNGP covariance is computed together with the NTK at no extra cost. This is especially convenient for researchers looking to investigate similarities and differences between these two infinite-width NN limits.

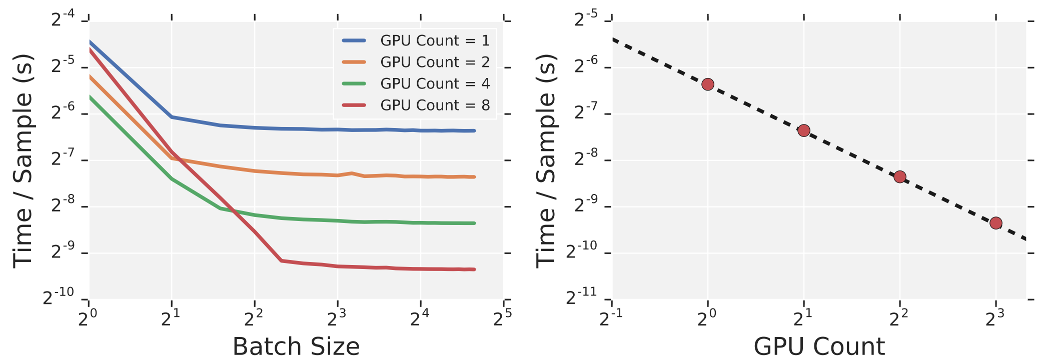


Figure 5: **Performance scaling with batch size (left) and number of GPUs (right).** Shows time per entry needed to compute the analytic NNGP and NTK covariance matrices (using `kernel_fn`) in a 21-layer ReLU network with global average pooling. **Left:** Increasing the batch size when computing the covariance matrix in blocks allows for a significant performance increase until a certain threshold when all cores in a single GPU are saturated. Simpler models are expected to have better scaling with batch size. **Right:** Time-per-sample scales linearly with the number of GPUs, demonstrating near-perfect hardware utilization.

Automatic batching and parallelism across multiple devices. In most cases as the dataset or model becomes large, it is impossible to perform the entire kernel computation at once. Additionally, in many cases it is desirable to parallelize the kernel computation across devices (CPUs, GPUs, or TPUs). NEURAL TANGENTS provides an easy way to perform both of these common tasks using a single `batch` decorator shown below:

```
batched_kernel_fn = nt.batch(kernel_fn, batch_size)
batched_kernel_fn(x, x) == kernel_fn(x, x)  # True!
```

This code works with either analytic kernels or empirical kernels. By default, it automatically shares the computation over all available devices. We plot the performance as a function of batch size and number of accelerators when computing the theoretical NTK of a 21-layer convolutional network in Figure 5, observing near-perfect scaling with the number of accelerators.

Op fusion. JAX and XLA allow end-to-end compilation of the whole kernel computation and/or inference. This enables the XLA compiler to fuse low-level ops into custom model-specific accelerator kernels, as well as eliminating overhead from op-by-op dispatch to an accelerator. In similar vein, we allow the covariance tensor to change it's order of dimensions from layer to layer, with the order tracked and parsed as additional metadata under the hood. This eliminates redundant transpositions[4] by adjusting the computation performed by each layer based on the input metadata.

4 CONCLUSION

We believe NEURAL TANGENTS will enable researchers to quickly and easily explore infinite-width networks. By democratizing this previously challenging model family, we hope that researchers will begin to use infinite neural networks, in addition to their finite counterparts, when faced with a new problem domain (especially in cases that are data-limited). In addition, we are excited to see novel uses of infinite networks as theoretical tools to gain insight and clarity into many of the hard theoretical problems in deep learning. Going forward, there are significant additions to NEURAL TANGENTS that we are exploring. There are more layers we would like to add in the future (§H) that will enable an even larger range of infinite network topologies. Additionally, there are further performance improvements we would like to implement, to allow experimenting with larger models and datasets. We invite the community to join our efforts by contributing new layers to the library, or by using it for research and providing feedback!

[4]These transpositions could not be automatically fused by the XLA compiler.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. ChainerMN: Scalable Distributed Deep Learning Framework. In *Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. URL http://learningsys.org/nips17/assets/papers/paper_25.pdf.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In *International Conference on Machine Learning*, 2018.

Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, Ruslan Salakhutdinov, and Ruosong Wang. On exact computation with an infinitely wide neural net. In *Advances In Neural Information Processing Systems*, 2019.

Lenaic Chizat, Edouard Oyallon, and Francis Bach. On lazy training in differentiable programming. *arXiv preprint arXiv:1812.07956*, 2019.

Youngmin Cho and Lawrence K Saul. Kernel methods for deep learning. In *Advances In Neural Information Processing Systems*, 2009.

François Chollet et al. Keras. https://keras.io, 2015.

Amit Daniely, Roy Frostig, and Yoram Singer. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. In *Advances In Neural Information Processing Systems*, pp. 2253–2261, 2016.

Simon S Du, Jason D Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. *arXiv preprint arXiv:1811.03804*, 2018a.

Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018b.

Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018a. URL http://github.com/google/jax.

Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. Stax, a flexible neural net specification library in jax, 2018b. URL https://github.com/google/jax/blob/master/jax/experimental/stax.py.

Adrià Garriga-Alonso, Carl Edward Rasmussen, and Laurence Aitchison. Deep convolutional networks as shallow gaussian processes. In *International Conference on Learning Representations*, 2019.

Jiri Hron, Roman Novak, Yasaman Bahri, and Jascha Sohl-Dickstein. Mean field theory of attention. personal communication, 2019.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, 2018.

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

Jaehoon Lee, Yasaman Bahri, Roman Novak, Sam Schoenholz, Jeffrey Pennington, and Jascha Sohl-dickstein. Deep neural networks as gaussian processes. In *International Conference on Learning Representations*, 2018.

Jaehoon Lee, Lechao Xiao, Samuel S. Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In *Advances in neural information processing systems*, 2019.

Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Advances in Neural Information Processing Systems*, pp. 8157–8166, 2018.

Alexander G de G Matthews, Mark Rowland, Jiri Hron, Richard E Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. *arXiv preprint arXiv:1804.11271*, 2018a.

Alexander G. de G. Matthews, Jiri Hron, Mark Rowland, Richard E. Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. In *International Conference on Learning Representations*, 2018b.

Radford M. Neal. Priors for infinite networks (tech. rep. no. crg-tr-94-1). *University of Toronto*, 1994.

Roman Novak, Lechao Xiao, Jaehoon Lee, Yasaman Bahri, Greg Yang, Jiri Hron, Daniel A. Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Bayesian deep convolutional networks with many channels are gaussian processes. In *International Conference on Learning Representations*, 2019.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Ben Poole, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli. Exponential expressivity in deep neural networks through transient chaos. In *Advances In Neural Information Processing Systems*, 2016.

Samuel S Schoenholz, Justin Gilmer, Surya Ganguli, and Jascha Sohl-Dickstein. Deep information propagation. *arXiv preprint arXiv:1611.01232*, 2016.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL `http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`.

Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of CNNs: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, 2018.

Ge Yang and Samuel Schoenholz. Mean field residual networks: On the edge of chaos. In *Advances In Neural Information Processing Systems*, 2017.

Greg Yang. Scaling limits of wide neural networks with weight sharing: Gaussian process behavior, gradient independence, and neural tangent kernel derivation. *arXiv preprint arXiv:1902.04760*, 2019.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2016.

APPENDIX

A LIBRARY DESCRIPTION

NEURAL TANGENTS provides a high-level interface for specifying analytic, infinite-width, Bayesian and gradient-flow trained neural networks as Gaussian Processes. This interface closely follows the `stax` API (Frostig et al., 2018b) in JAX.

A NEURAL NETWORKS WITH JAX

`stax` represents each component of a network as two functions: `init_fn` and `apply_fn`. These components can be composed in serial or in parallel to produce new network components with their own `init_fn` and `apply_fn`. In this way, complicated neural network architectures can be specified hierarchically.

Calling `init_fn` on a random seed and an input shape generates a random draw of trainable parameters for a neural network. Calling `apply_fn` on these parameters and a batch of inputs returns the outputs of the given finite neural network.

```
from jax.experimental import stax
init_fn, apply_fn = stax.serial(stax.Dense(512), stax.Relu, stax.Dense(10))
_, params = init_fn(key, (-1, 32 * 32 * 3))
fx_train, fx_test = apply_fn(params, x_train),  apply_fn(params, x_test)
```

B INFINITE NEURAL NETWORKS WITH NEURAL TANGENTS

We extend `stax` layers to return a third function `kernel_fn`, which represents the covariance functions of the infinite NNGP and NTK networks of the given architecture (recall that since infinite networks are GPs, they are fully defined by their covariance functions, assuming 0 mean as is common in the literature).

```
from neural_tangents import stax
init_fn, apply_fn, kernel_fn = stax.serial(stax.Dense(512), stax.Relu(), stax.Dense(10))
```

We demonstrate a specification of a more complicated architecture (WideResNet) in §2.2.

`kernel_fn` accepts two batches of inputs `x1` and `x2` and returns their NNGP covariance and NTK matrices as `kernel_fn(x1, x2).nngp` and `kernel_fn(x1, x2).ntk` respectively, which can then be used to make posterior test set predictions as the mean of a conditional multivariate normal:

```
from jax.numpy.linalg import inv
y_test = kernel_fn(x_test, x_train).ntk @ inv(kernel_fn(x_train, x_train).ntk) @ y_train
```

Note that the above code does not do Cholesky decomposition and is presented merely to show the mathematical expression. We provide efficient GP inference method in the `predict` submodule:

```
import neural_tangents as nt
y_test = nt.predict.gp_inference(kernel_fn, x_train, y_train, x_test,
                                 get='NTK', diag_reg=1e-4, compute_var=False)
```

C COMPUTING INFINITE NETWORK KERNELS IN BATCHES AND IN PARALLEL

Naively, the `kernel_fn` will compute the whole kernel in a single call on one device. However, for large datasets or complicated architectures, it is often necessary to distribute the calculation in some way. To do this, we introduce a `batch` decorator that takes a `kernel_fn` and returns a new `kernel_fn` with the exact same signature. The new function computes the kernel in batches and automatically parallelizes the calculation over however many devices are available, with near-perfect speedup scaling with the number of devices (Figure 5, right).

```python
import neural_tangents as nt
kernel_fn = nt.batch(kernel_fn, batch_size=32)
```

D TRAINING DYNAMICS OF INFINITE NETWORKS

In addition to closed form multivariate Gaussian posterior prediction, it is also interesting to consider network predictions following continuous gradient descent. To facilitate this we provide several functions to compute predictions following gradient descent with an MSE loss, for gradient descent with arbitrary loss, or for momentum with arbitrary loss. The first case is handled analytically, while the latter two are computed by numerically integrating the differential equation. For example, the following code will compute the function evaluation on train and test points following gradient descent for some time `training_time`.

```python
import neural_tangents as nt
predictor = nt.predict.gradient_descent_mse(kernel_fn(x_train, x_train), y_train,
fx_train, fx_test = predictor(training_time, fx_train, fx_test)
```

E INFINITE NETWORKS OF ANY ARCHITECTURE THROUGH SAMPLING

Of course, there are cases where the analytic kernel cannot be computed. To support these situations, we provide utility functions to efficiently compute Monte Carlo estimates of the NNGP covariance and NTK. These functions work with neural networks constructed using *any* neural network library.

```python
from jax import random
from jax.experimental import stax
import neural_tangents as nt

init_fn, apply_fn = stax.serial(stax.Dense(64), stax.BatchNorm(), stax.Sigmoid, stax.Dense(1))
kernel_fn = nt.monte_carlo_kernel_fn(init_fn, apply_fn, key=random.PRNGKey(1), n_samples=128)
kernel = kernel_fn(x_train, x_train)
```

We demonstrate convergence of the Monte Carlo kernel estimates to the closed-form analytic kernels in the case of a WideResNet in Figure 2.

F WEIGHTS OF WIDE BUT FINITE NETWORKS

While most of NEURAL TANGENTS is devoted to a function-space perspective—describing the distribution of function values on finite collections of training and testing points—we also provide tools to investigate a dual weight space perspective described in Lee et al. (2019). Convergence of dynamics to NTK dynamics coincide with networks being described by a linear approximation about their initial set of parameters. We provide decorators `linearize` and `taylor_expand` to approximate functions to linear order and to arbitrary order respectively. Both functions take an `apply_fn` and returns a new `apply_fn` that computes the series approximation. These act exactly like normal JAX functions and, in particular, can be plugged into gradient descent.

```python
import neural_tangents as nt
taylor_apply_fn = nt.taylor_expand(apply_fn, params, order)
fx_train_approx = taylor_apply_fn(new_params, x_train)
```

G ARCHITECTURE SPECIFICATIONS

```python
from neural_tangents import stax

def ConvolutionalNetwork(depth, W_std=1.0, b_std=0.0):
  collect_layers = []
  for _ in range(depth):
    collect_layers += [stax.Conv(1, (3, 3), W_std=W_std, b_std=b_std, padding='SAME'),
                       stax.Relu()]
  collect_layers += [stax.Flatten(), stax.Dense(1, W_std, b_std)]
  return stax.serial(*collect_layers)
```

Listing 2: **All-convolutional model (ConvOnly) definition used in Figure 3.**

```python
from neural_tangents import stax

def FullyConnectedNetwork(depth, W_std=1.0, b_std=0.0):
  collect_layers = [stax.Flatten()]
  for _ in range(depth):
     collect_layers += [stax.Dense(1, W_std, b_std), stax.Relu()]
  collect_layers += [stax.Dense(1, W_std, b_std)]
  return stax.serial(*collect_layers)
```

Listing 3: **Fully-connected (FC) model definition used in Figure 3.**

H IMPLEMENTED AND COMING SOON FUNCTIONALITY

The following layers[5] are currently implemented, with translation rules given in Table 1:

- `serial`, `parallel`
- `FanOut`, `FanInSum`
- `Dense`, `Conv`[6] with arbitrary filter shapes, strides, and padding[7]
- `Relu`, `LeakyRelu`, `Abs`, `ABRelu`[8], `Erf`, `Identity`
- `Flatten`, `AvgPool`, `GlobalAvgPool`, `GlobalSelfAttention` (Hron et al., 2019)

The following is in our near-term plans:

- `SumPool`, `Exp`, `Elu`, `Selu`, `Gelu`
- `LayerNorm`
- Apache Beam support.

[5] `Abs`, `ABRelu`, `GlobalAvgPool`, `GlobalSelfAttention` are only available in our library `nt.stax` and not in `jax.experimental.stax`.

[6] Only `NHWC` data format is currently supported, but extension to other formats is trivial and will be done shortly.

[7] Note that in addition to `SAME` and `VALID`, we support `CIRCULAR` padding, which is especially handy for theoretical analysis and was used by Xiao et al. (2018) and Novak et al. (2019).

[8] $a \min(x, 0) + b \max(x, 0)$.

The following layers do *not* have a known closed-form solution for infinite network covariances, and networks with them have to be estimated empirically (provided with out implementation via `nt.monte_carlo_kernel_fn`) or using other approximations (not currently implemented):

- `Sigmoid`, `Tanh`[9], `Swish`[10], `Softmax`, `LogSoftMax`, `Softplus`, `MaxPool`.

Tensor Op	**NNGP Op**	**NTK Op**
$\mathcal{X}$	$\mathcal{K}$	Θ
$\text{Dense}(\sigma_w, \sigma_b)$	$\sigma_w^2 \mathcal{K} + \sigma_b^2$	$(\sigma_w^2 \mathcal{K} + \sigma_b^2) + \sigma_w^2 \Theta$
ϕ	$\mathcal{T}(\mathcal{K})$	$\dot{\mathcal{T}}(\mathcal{K}) \odot \Theta$
$\text{Conv}(\sigma_w, \sigma_b)$	$\sigma_w^2 \mathcal{A}(\mathcal{K}) + \sigma_b^2$	$\sigma_w^2 \mathcal{A}(\mathcal{K}) + \sigma_b^2 + \sigma_w^2 \mathcal{A}(\Theta)$
Flatten	$\text{Tr}(\mathcal{K})$	$\text{Tr}(\mathcal{K} + \Theta)$
$\text{AvgPool}(s, q, p)$	$\text{AvgPool}(s, q, p)(\mathcal{K})$	$\text{AvgPool}(s, q, p)(\mathcal{K} + \Theta)$
GlobalAvgPool	$\text{GlobalAvgPool}(\mathcal{K})$	$\text{GlobalAvgPool}(\mathcal{K} + \Theta)$
$\text{Attn}(\sigma_{QK}, \sigma_{OV})$ (Hron et al., 2019)	$\text{Attn}(\sigma_{QK}, \sigma_{OV})(\mathcal{K})$	$2\text{Attn}(\sigma_{QK}, \sigma_{OV})(\mathcal{K}) +$ $\text{Attn}(\sigma_{QK}, \sigma_{OV})(\Theta)$
$\text{FanInSum}(\mathcal{X}_1, \dots, \mathcal{X}_n)$	$\sum_{j=1}^n \mathcal{K}_j$	$\sum_{j=1}^n \Theta_j$
$\text{FanOut}(n)$	$[\mathcal{K}] * n$	$[\Theta] * n$

Table 1: **Translation rules (§3) converting tensor operations into operations on NNGP and NTK kernels.** Here the input tensor $\mathcal{X}$ is assumed to have shape $|\mathcal{X}| \times H \times W \times C$ (dataset size, height, width, number of channels), and the full NNGP and NT kernels $\mathcal{K}$ and $\mathcal{T}$ are considered to be of shape $(|\mathcal{X}| \times H \times W)^{\times 2}$ (in practice shapes of $|\mathcal{X}|^{\times 2} \times H \times W$ and $|\mathcal{X}|^{\times 2}$ are also possible, depending on which optimizations in §3.1 are applicable). **Notation details.** The Tr and GlobalAvgPool ops are assumed to act on all spatial axes (with sizes H and W in this example), producing a $|\mathcal{X}|^{\times 2}$-kernel. Similarly, the AvgPool op is assumed to act on all spatial axes as well, applying the specified strides s, pooling window sizes p and padding strategy p to the respective axes pairs in $\mathcal{K}$ and $\mathcal{T}$ (acting as 4D pooling with replicated parameters of the 2D version). $\mathcal{T}$ and $\dot{\mathcal{T}}$ are defined identically to Lee et al. (2019) as $\mathcal{T}(\Sigma) = \mathbb{E}\left[\phi(u)\phi(u)^T\right], \dot{\mathcal{T}}(\Sigma) = \mathbb{E}\left[\phi'(u)\phi'(u)^T\right], u \sim \mathcal{N}(0, \Sigma)$. These expressions can be evaluated in closed form for many nonlinearities, and preserve the shape of the kernel. The $\mathcal{A}$ op is defined similarly to Novak et al. (2019); Xiao et al. (2018) as $[\mathcal{A}(\Sigma)]_{h,h'}^{w,w'}(x, x') = \sum_{dh,dw} [\Sigma]_{h+dh,h'+dh}^{w+dw,w'+dw}(x, x')/q^2$, where the summation is performed over the convolutional filter receptive field with q pixels (we assume unit strides and circular padding in this expression, but generalization to other settings is trivial and supported by the library). $[\Sigma] * n = [\Sigma, \dots, \Sigma]$ (n-fold replication). See Figure 4 for an example of applying the translation rules to a specific model.

[9] Note that these nonlinearities are similar to `Erf` which does have a solution and is implemented.

[10] Note that this nonlinearity is similar to `Gelu`.