2nd Symposium on Advances in Approximate Bayesian Inference, 2019 1–19

Neural Tangents: Fast and Easy Infinite Neural Networks in Python

Roman Novak* ROMANN@GOOGLE.COM
Lechao Xiao* XLC@GOOGLE.COM
Jiri Hron† JH2084@CAM.AC.UK
Jaehoon Lee JAEHLEE@GOOGLE.COM
Alexander A. Alemi ALEMI@GOOGLE.COM
Jascha Sohl-Dickstein JASCHASD@GOOGLE.COM
Samuel S. Schoenholz* SCHSAM@GOOGLE.COM
Google Brain, †*University of Cambridge*

Abstract

NEURAL TANGENTS is a library designed to enable research into 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. Infinite-width networks can be trained analytically using exact Bayesian inference or using gradient descent via the Neural Tangent Kernel. Additionally, NEURAL TANGENTS provides tools to study gradient descent training dynamics of wide but finite networks in either function space or weight space.

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 `github.com/google/neural-tangents`. We also provide an interactive **Colab notebook**.

1. Motivation

Deep neural networks (DNNs) owe their success in part to the broad availability of high-level, flexible, and efficient software libraries (Abadi et al., 2015; Chollet et al., 2015; Paszke et al., 2017; Tokui et al., 2015; Akiba et al., 2017; Bradbury et al., 2018a), enabling researchers to rapidly build complex models by constructing them out of smaller primitives.

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 and continuous gradient descent trained networks become Gaussian Procesesses, defined by architecture-specific kernels commonly referred to as NNGP and NTK respectively (§A). 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 SOTA 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 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. This process is laborious and error prone, and is reminiscent of the state of neural networks before high quality Automatic Differentiation (AD) packages proliferated.

* Equal contribution. †Work done during an internship at Google Brain.

In this note, we introduce a new open-source software library called Neural Tangents targeting JAX (Bradbury et al., 2018a) to facilitate research on infinite limits of neural networks. Neural Tangents provides a high-level neural network API for specifying and doing inference with complex, hierarchical, models. In §2 we demonstrate the ease, efficiency, and versatility of performing calculations with infinite networks using Neural Tangents on a very short example, and refer the reader to §C and §D for more details on functionality and performance.

2. Example: training and inference with infinite networks

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 on a toy task. Below we define a 3-layer Erf[1] model:

```
from neural_tangents import 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))
```

The above snippet simultaneously specifies the finite, 2048-wide network, as well as the infinite one. The finite network is defined by the `(init_fun, apply_fun)` function pair that initializes the trainable parameters and performs the forward pass respectively. The infinite network, as a GP, is specified by its kernel function `kernel_fn`,[2] and can be trained with gradient descent (computation amounting to a simple linear algebra expression in the infinite limit; see §A) as follows:

```
import neural_tangents as nt
predict_fn = nt.predict.gradient_descent_mse_gp(kernel_fn, x_train, y_train, x_test,
                                                get='ntk', diag_reg=1e-4, compute_var=True)
y_mean, y_var = predict_fn(t=100)  # Multivariate normal prediction at training time t = 100.
```

In Figure 1 we compare the result of infinite-width training 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.

We now demonstrate the flexibility of our library by specifying and doing inference with more complex infinite-width architectures, including a variant of a WideResNet (Zagoruyko and Komodakis, 2016), which we define in Listing 1. We present results in Figure 2, observing a familiar (from the finite-width world) hierarchy of performance in terms of architecture (FC $<$ ConvOnly $<$ WideResNet w/ pooling).

1. Error function, a nonlinearity similar to tanh; see §F for implemented nonlinearities, including ReLU.
2. JAX users will recognize that we closely mimick their default NN specification package `stax` (Bradbury et al., 2018b). In fact, our library module `nt.stax` is designed to serve as a drop-in replacement of `jax.experimental.stax`.

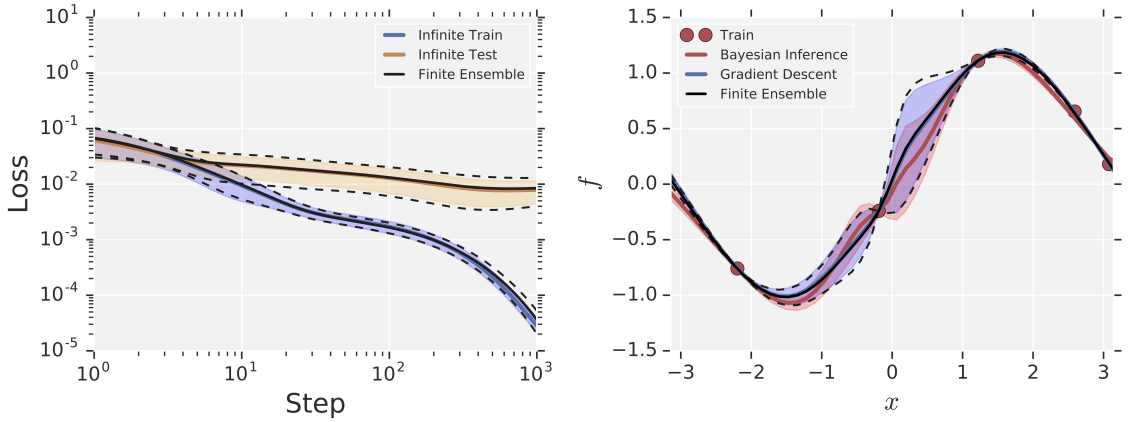


Figure 1: **Training dynamics of an infinite network and an ensemble of finite-width networks. Left:** Train and test MSE loss evolution throughout training. **Right:** Trained model output predictions of the trained infinite network and the respective ensemble of finite-width networks. The shaded region and the dashed lines denote (two, in the right plot) standard deviations of uncertainty in the loss or predictions for the infinite network and the ensemble respectively. Training data is drawn from the process $y_i = \sin(x_i) + \epsilon_i$ with $x_i \sim \text{Uniform}(-\pi, \pi)$ and $\epsilon_i \sim \mathcal{N}\left(0, \sigma^2\right)$.

3. How it works

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 3 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 §F for a list of all tensor operations for which translation rules are currently implemented, and §C and §D for a more detailed API and implementation description.

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 (§F) 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 applications and providing feedback!

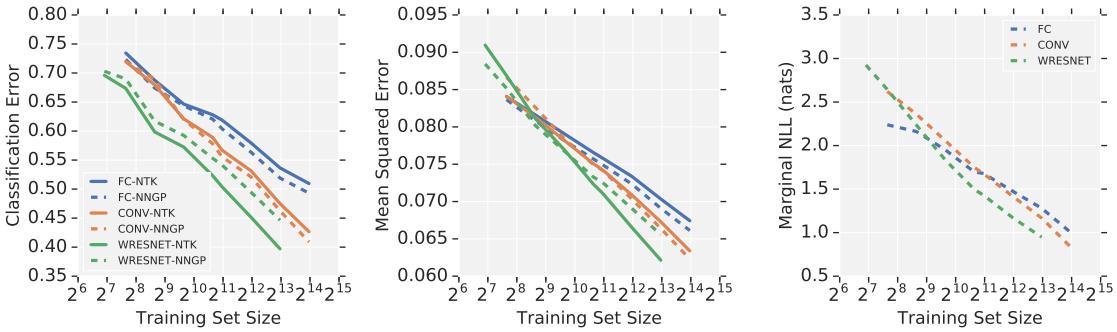
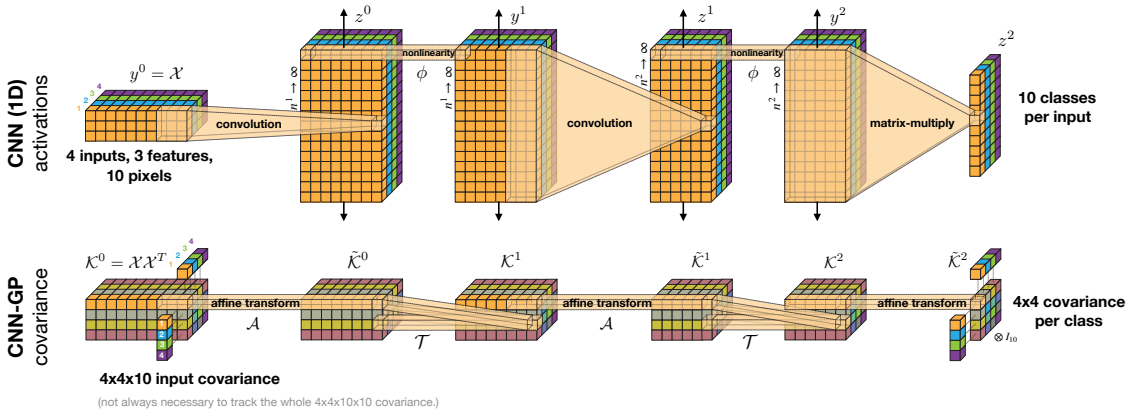


Figure 2: **CIFAR-10 classification with varying neural network architectures.** Neural Tangents simplify experimentation with architectures. Here we use infinite time NTK inference and full Bayesian NNGP inference for CIFAR-10 for **Fully Connected (FC, Listing 3)**, **Convolutional network without pooling (CONV, Listing 2)**, and **Wide Residual Network (WRESNET, Listing 1)** with pooling. As is common in prior work (Lee et al., 2018; Novak et al., 2019), the classification task is treated as MSE regression on zero-mean targets like $(-0.1, \ldots, -0.1, 0.9, -0.1, \ldots, -0.1)$. For each training set size, the best model in the family is selected by minimizing the mean negative marginal log-likelihood (NLL, right) on the training set.

```
from neural_tangents import stax

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 4.



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)	$z^0 = \mathrm{Conv}\left(y^0\right)$	$\tilde{\mathcal{K}}^0 = \mathcal{A}\left(\mathcal{K}^0\right)$	$\tilde{\Theta}^0 = \tilde{\mathcal{K}}^0 + \mathcal{A}\left(\Theta^0\right)$
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)	$z^1 = \mathrm{Conv}\left(y^1\right)$	$\tilde{\mathcal{K}}^1 = \mathcal{A}\left(\mathcal{K}^1\right)$	$\tilde{\Theta}^1 = \tilde{\mathcal{K}}^1 + \mathcal{A}\left(\Theta^1\right)$
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)	$z^2 = \mathrm{Dense} \circ \mathrm{Flatten}\left(y^2\right)$	$\tilde{\mathcal{K}}^2 = \mathrm{Tr}\left(\mathcal{K}^2\right)$	$\tilde{\Theta}^2 = \tilde{\mathcal{K}}^2 + \mathrm{Tr}\left(\Theta^2\right)$

Figure 3: **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).

Acknowledgments

We thank Yasaman Bahri for significant code contributions, frequent discussion and useful feedback on the manuscript, Sergey Ioffe for feedback on the text, as well as Greg Yang, Ravid Ziv, and Jeffrey Pennington for discussion and feedback on early versions of the library.

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.

Anonymous. Infinite attention: Nngp and ntk for deep attention networks. In *International Conference on Machine Learning (ICML)*, 2020. submission under review.

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.

Anastasia Borovykh. A gaussian process perspective on convolutional neural networks. *arXiv preprint arXiv:1810.10798*, 2018.

James Bradbury, 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.

James Bradbury, 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.

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*, pages 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.

Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in Neural Information Processing Systems*, 2018.

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

GPy. GPy: A gaussian process framework in python. http://github.com/SheffieldML/GPy, 2012.

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*, pages 8157–8166, 2018.

Alexander G. de G. Matthews, Mark van der Wilk, Tom Nickson, Keisuke. Fujii, Alexis Boukouvalas, Pablo Le'on-Villagr'a, Zoubin Ghahramani, and James Hensman. GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18 (40):1–6, apr 2017. URL http://jmlr.org/papers/v18/16-537.html.

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.

Lechao Xiao, Jeffrey Pennington, and Samuel S Schoenholz. Disentangling trainability and generalization in deep learning. *arXiv preprint arXiv:1912.13053*, 2019.

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.

Difan Zou, Yuan Cao, Dongruo Zhou, and Quanquan Gu. Gradient descent optimizes over-parameterized deep relu networks. *Machine Learning*, Oct 2019. ISSN 1573-0565. doi: 10.1007/s10994-019-05839-6. URL https://doi.org/10.1007/s10994-019-05839-6.

Appendix

A. 5-minute background

We briefly describe the NNGP (§A.1) and NTK (§A.2). **NNGP.** Neural networks are often structured as affine transformations followed by pointwise applications of nonlinearities. 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 $\mu(x) = \mathcal{K}(x, \mathcal{X})\mathcal{K}(\mathcal{X}, \mathcal{X})^{-1}\mathcal{Y}$ and variance $\sigma^2(x) = \mathcal{K}(x, x) - \mathcal{K}(x, \mathcal{X})\mathcal{K}(\mathcal{X}, \mathcal{X})^{-1}\mathcal{K}(\mathcal{X}, x)$, 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 $\mu_t(x) = \Theta(x, \mathcal{X})\Theta(\mathcal{X}, \mathcal{X})^{-1} (I - \exp[-\eta\Theta(\mathcal{X}, \mathcal{X})t]) \mathcal{Y}$, which further simplifies to $\mu_\infty(x) = \Theta(x, \mathcal{X})\Theta(\mathcal{X}, \mathcal{X})^{-1}\mathcal{Y}$ at convergence (infinite training time).

A.1. Infinite Bayesian neural networks references (NNGP)

The NNGP 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 (Borovykh, 2018; 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 and Saul, 2009; Daniely et al., 2016; Poole et al., 2016; Schoenholz et al., 2016; Yang and Schoenholz, 2017; Xiao et al., 2018).

A.2. Infinite gradient descent trained neural networks references (NTK)

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 and Liang, 2018; Allen-Zhu et al., 2018; Du et al., 2018a,b; Zou et al., 2019). 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.

B. Neural Tangents and prior work

We briefly discuss the differences between Neural Tangents and the relevant prior work.

1. **Prior benchmarks in the domain of infinitely wide neural networks.** Various prior works have evaluated convolutional and fully-connected models on certain datasets (Lee et al., 2018; Matthews et al., 2018b,a; Novak et al., 2019; Garriga-Alonso et al., 2019; Arora et al., 2019). While these efforts must have required implementing certain parts of our library, to our knowledge such prior efforts were either not open-sourced or not comprehensive / user-friendly / scalable enough to be used as a user-facing library. In addition, all of the works above used their own separate implementation, which further highlights a need for a more general approach.

2. **Code released by Lee et al. (2019).** Lee et al. (2019) have released code along with their paper submission, which is a strict and minor subset of our library. More specifically, at the time of the submission, Lee et al. (2019) have released code equivalent to `nt.linearize`, `nt.empirical_ntk_fn`, `nt.predict.gradient_descent_mse`, `nt.predict.gradient_descent`, and `nt.predict.momentum`. Every other part of the library (most notably, `nt.stax`) is new in this submission and was not used by Lee et al. (2019) or any other prior work. At the time of writing, Neural Tangents differs from the code released by Lee et al. (2019) by about $+9,500/-2,500$ lines of code.

3. **GPy (2012), GPFlow (Matthews et al., 2017), GPytorch (Gardner et al., 2018), and other GP packages.** While various packages allowing for kernel construction, optimization, and inference with Gaussian Processes exist, none of them allow easy construction of the very specific kernels corresponding to infinite neural networks (NNGP/NTK; `nt.stax`), nor do they provide the tools and convenience for studying wide but finite networks and their training dynamics (`nt.taylor_expand`, `nt.predict`, `nt.monte_carlo_kernel_fn`). On the other hand, Neural Tangents does not provide any tools for approximate inference with these kernels.

C. Library description

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

C.1. 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)
```

C.2. 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 Listing 1.

`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_cov=False)
```

C.3. 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 6, right).

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

Note that batching is often used to compute large covariance matrices that may not even fit on a GPU/TPU device, and require to be stored and used for inference using CPU RAM. This is easy to achieve by simply specifying `nt.batch(..., store_on_device=False)`. Once the matrix is stored in RAM, inference will be performed with a CPU when `nt.predict` methods are called. As mentioned in §D, for many (notably, convolutional, and especially pooling) architectures, inference cost can be small relative to kernel construction, even when running on CPU (for example, `jax.scipy.linalg.solve(..., sym_pos=True)` takes less than 3 minutes to execute on a $45,000 \times 45,000$ training covariance matrix and a $45,000 \times 10$ training target matrix).

C.4. 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`.

```
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)
```

C.5. Infinite networks of any architecture through sampling

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.

```
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 4.

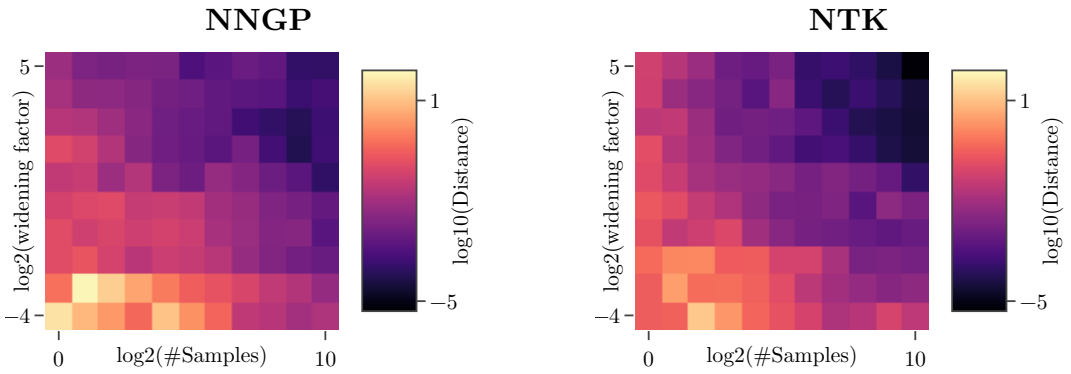


Figure 4: **Convergence of the Monte Carlo (MC) estimates** of the WideResNet WRN-28-k (where k is the widening factor) NNGP and NTK kernels (computed with `monte_carlo_kernel_fn`) to their analytic values (WRN-28-∞, computed with `kernel_fn`), as the network gets wider by increasing the widening factor (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.

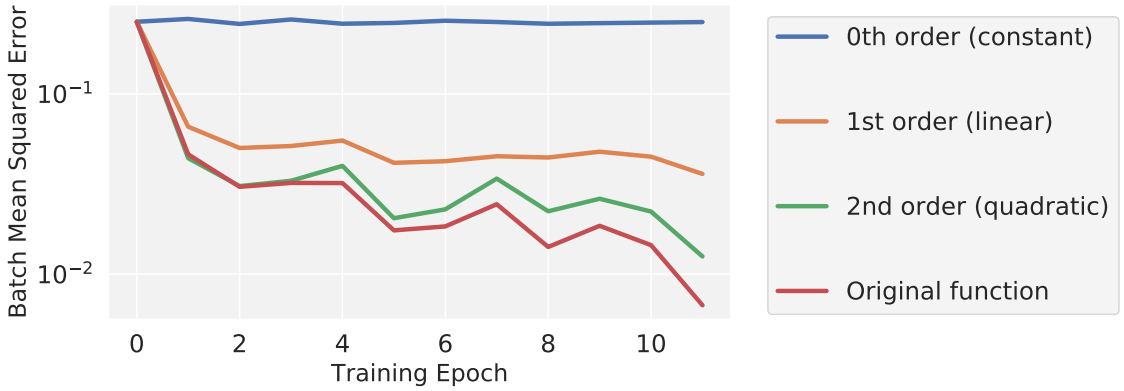


Figure 5: **Training a neural network and its various approximations using** `nt.taylor_expand`. Presented is a **5-layer Erf-neural network of width 512** trained on MNIST using SGD with momentum, along with its **constant (0^{th} order)**, **linear (1^{st} order)**, and **quadratic (2^{nd} order)** Taylor expansions about the initial parameters. As training progresses (left to right), lower-order expansions deviate from the original function faster than higher-order ones.

C.6. 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.

```
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)
```

These act exactly like normal JAX functions and, in particular, can be plugged into gradient descent, which we demonstrate in Figure 5.

C.7. Extending Neural Tangents

Many neural network layers admit a sensible infinite-width limit behavior in the Bayesian and continuous gradient descent regimes as long as the multivariate central limit theorem applies to their outputs conditioned on their inputs. To add such layer to Neural Tangents, one only has to implement it as a method in `nt.stax` with the following signature:

```
@_layer  # an internal decorator taking care of certain boilerplate.
NewLayer(layer_params: Any) -> (init_fn: function, apply_fn: function, kernel_fn: function)
```

Here `init_fn` and `apply_fn` are initialization and the forward pass methods of the finite-width layer implementation (see §C.1). If the layer of interest already exists in JAX, there is no need to implement these methods and the user can simply return the respective methods from `jax.experimental.stax` (see `nt.stax.Flatten` for an example; in fact the majority of `nt.stax` layers call the original `jax.experimental.stax` layers for finite-width layer methods). In this case what remains is to implement the `kernel_fn` method with signature

```
kernel_fn(input_kernel: nt.utils.Kernel) -> output_kernel: nt.utils.Kernel
```

Here both `input_kernel` and `output_kernel` are `namedtuple`s containing the NNGP and NTK covariance matrices, as well as additional metadata useful for computing the kernel propagation operation. The specific operation to be performed should be derived by the user in the context of the particular operation that the finite-width layer performs. This transformation could be as simple as an affine map on the kernel matrices, but could also be analytically intractable.

Once implemented, the correctness of the implementation can be very easily tested by extending the `nt.tests.stax_test` with the new layer, to test the agreement with large-widths empirical NNGP and NTK kernels.

D. Performance

Our library performs 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, 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. This results in outputs that are normally distributed with a block-diagonal covariance matrix of the form $\Sigma \otimes I_C$, where Σ has shape $|\mathcal{X}| \times |\mathcal{X}|$ and I_C is the $C \times C$ identity matrix. This reduces the computational complexity and storage in many common cases by an order of magnitude, which makes closed-form exact inference feasible in these cases.

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 matrix (as opposed to inverting it). Specifically for a convolutional network of depth l, constructing the $|\mathcal{X}| \times |\mathcal{X}|$ output covariance matrix, Σ, involves computing l intermediate layer covariance matrices, 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 intermediate layer outputs (e.g. $d = 1024$ in the case of CIFAR-10 with `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 3 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.

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:

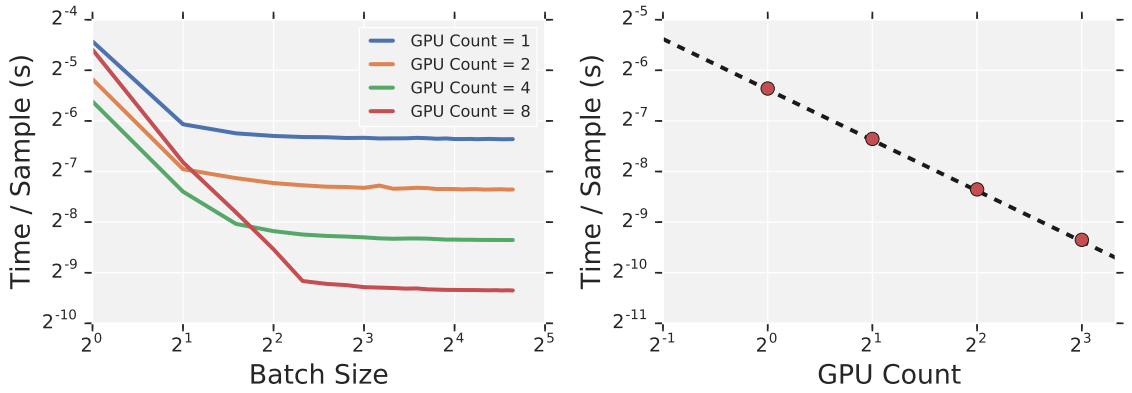


Figure 6: **Performance scaling with batch size (left) and number of GPUs (right).** Shows time per entry needed to compute the analytic NNGP and NTK covariance matrices (using `kernel_fn`) in a 21-layer ReLU network with global average pooling. **Left:** Increasing the batch size when computing the 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.

```
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 6, 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 its order of dimensions from layer to layer, with the order tracked and parsed as additional metadata under the hood. This eliminates redundant transpositions[3] by adjusting the computation performed by each layer based on the input metadata.

E. A taste of Tensor-to-Kernel Ops Translation

To get some intuition behind the translation rules, we consider the case of a nonlinearity followed by a dense layer. Let $z = z(\mathcal{X}, \theta) \in \mathbb{R}^{d \times n}$ be the preactivations resulting from d distinct inputs at a node in some hidden layer of a neural network. Suppose z has NNGP

3. These transpositions could not be automatically fused by the XLA compliler.

kernel and NTK given by

$$\mathcal{K}_z = \mathbb{E}_\theta \left[z_i z_i^T \right] , \quad \Theta_z = \mathbb{E}_\theta \left[\frac{\partial z_i}{\partial \theta} \left(\frac{\partial z_i}{\partial \theta} \right)^T \right] \tag{1}$$

where $z_i \in \mathbb{R}^d$ is the i^{th} neuron and θ are the parameters in the network up until z. Here d is the cardinality of the network inputs $\mathcal{X}$ and n is the number of neurons in the z node. We assume z is a mean zero multivariate Gaussian. We wish to compute the kernel corresponding to $h = \text{Dense}\left(\sigma_\omega, \sigma_b\right)\left(\phi(z)\right)$ by computing the kernels of $y = \phi(z)$ and $h = \text{Dense}\left(\sigma_\omega, \sigma_b\right)(y)$ separately. Here,

$$h = \text{Dense}(\sigma_\omega, \sigma_b)(y) \equiv \left(1/\sqrt{n}\right) \sigma_\omega W y + \sigma_b \beta, \tag{2}$$

and the variables W_{ij} and β_i are i.i.d. Gaussian $\mathcal{N}(0, 1)$. We will compute kernel operations - denoted ϕ^* and $\text{Dense}(\sigma_\omega, \sigma_b)^*$ - induced by the tensor operations ϕ and $\text{Dense}(\sigma_\omega, \sigma_b)$ [4]. Finally, we will compute the kernel operation associated with the composition $\left(\text{Dense}(\sigma_\omega, \sigma_b) \circ \phi\right)^* = \text{Dense}(\sigma_\omega, \sigma_b)^* \circ \phi^*$.

First we compute the NNGP and NT kernels for y. To compute $\mathcal{K}_y$ note that from its definition,

$$\mathcal{K}_y = \mathcal{K}_{\phi(z)} = \mathbb{E}_\theta \left[\phi(z)_i \, \phi(z)_i^T \right] = \mathbb{E}_\theta [\phi(z_i) \, \phi(z_i)^T] = \mathcal{T}(\mathcal{K}_z). \tag{3}$$

Since ϕ does not introduce any new variables Θ_y can be computed as,

$$\Theta_y = \mathbb{E}_\theta \left[\frac{\partial \phi(z_i)}{\partial \theta} \left(\frac{\partial \phi(z_i)}{\partial \theta} \right)^T \right] = \mathbb{E}_\theta \left[\text{diag}(\dot{\phi}(z_i)) \frac{\partial z_i}{\partial \theta} \left(\frac{\partial z_i}{\partial \theta} \right)^T \text{diag}(\dot{\phi}(z_i)) \right] = \dot{\mathcal{T}}(\mathcal{K}_z) \odot \Theta_z.$$

Taken together these equations imply that,

$$\left(\mathcal{K}_y, \, \Theta_y\right) = \phi^* \left(\mathcal{K}_z, \, \Theta_z\right) \equiv \left(\mathcal{T}(\mathcal{K}_z), \, \dot{\mathcal{T}}(\mathcal{K}_z) \odot \Theta_z \right) \tag{4}$$

will be the translation rule for a pointwise nonlinearity. Note that Equation Equation 4 only has an analytic expression for a small set of activation functions ϕ.

Next we consider the case of a dense operation. Using the independence between the weights, the biases, and h it follows that,

$$\mathcal{K}_h = \mathbb{E}_{W,\beta,\theta}[h_i h_i^T] = \sigma_\omega^2 \mathbb{E}_\theta[y_i y_i^T] + \sigma_b^2 = \sigma_\omega^2 \mathcal{K}_y + \sigma_b^2. \tag{5}$$

Finally, the NTK of h can be computed as a sum of two terms:

$$\Theta_h = \mathbb{E}_{W,\beta,\theta} \left[\frac{\partial h_i}{\partial (W, \beta)} \left(\frac{\partial h_i}{\partial (W, \beta)} \right)^T \right] + \mathbb{E}_{W,\beta,\theta} \left[\frac{\partial h_i}{\partial \theta} \left(\frac{\partial h_i}{\partial \theta} \right)^T \right] = \sigma_\omega^2 \mathcal{K}_y + \sigma_b^2 + \sigma_\omega^2 \Theta_y \, . \tag{6}$$

This gives the translation rule for the dense layer in terms of $\mathcal{K}_y$ and Θ_y as,

$$\left(\mathcal{K}_h, \Theta_h\right) = \text{Dense}(\sigma_\omega, \sigma_b)^* \left(\mathcal{K}_y, \Theta_y\right) \equiv \left(\sigma_\omega^2 \mathcal{K}_y + \sigma_b^2, \, \sigma_\omega^2 \mathcal{K}_y + \sigma_b^2 + \sigma_\omega^2 \Theta_y \right) . \tag{7}$$

4. $\mathcal{T}\left(\Sigma\right) \equiv \mathbb{E}\left[\phi(u)\phi(u)^T\right], \dot{\mathcal{T}}\left(\Sigma\right) \equiv \mathbb{E}\left[\phi'(u)\phi'(u)^T\right], u \sim \mathcal{N}\left(0, \Sigma\right)$, as in (Lee et al., 2019).

F. Implemented and coming soon functionality

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

- `serial`
- `parallel`
- `FanOut`
- `FanInSum`
- `FanInConcat`
- `Dense`
- `Conv`[6] with arbitrary filter shapes, strides, and padding[7]
- `Relu`, `LeakyRelu`, `Abs`, `ABRelu`[8], `Erf`, `Identity`
- `Flatten`
- `AvgPool`, `GlobalAvgPool`
- `SumPool`, `GlobalSumPool`
- `Dropout`
- `GlobalSelfAttention` (Anonymous, 2020)
- `LayerNorm`

The following is in our near-term plans:

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

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`.

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. $a \min(x, 0) + b \max(x, 0)$.
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`.

G. Architecture specifications

```
from neural_tangents import stax

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

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

```
from neural_tangents import stax

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

Listing 3: **Fully-connected (FC) model definition used in Figure 2.**

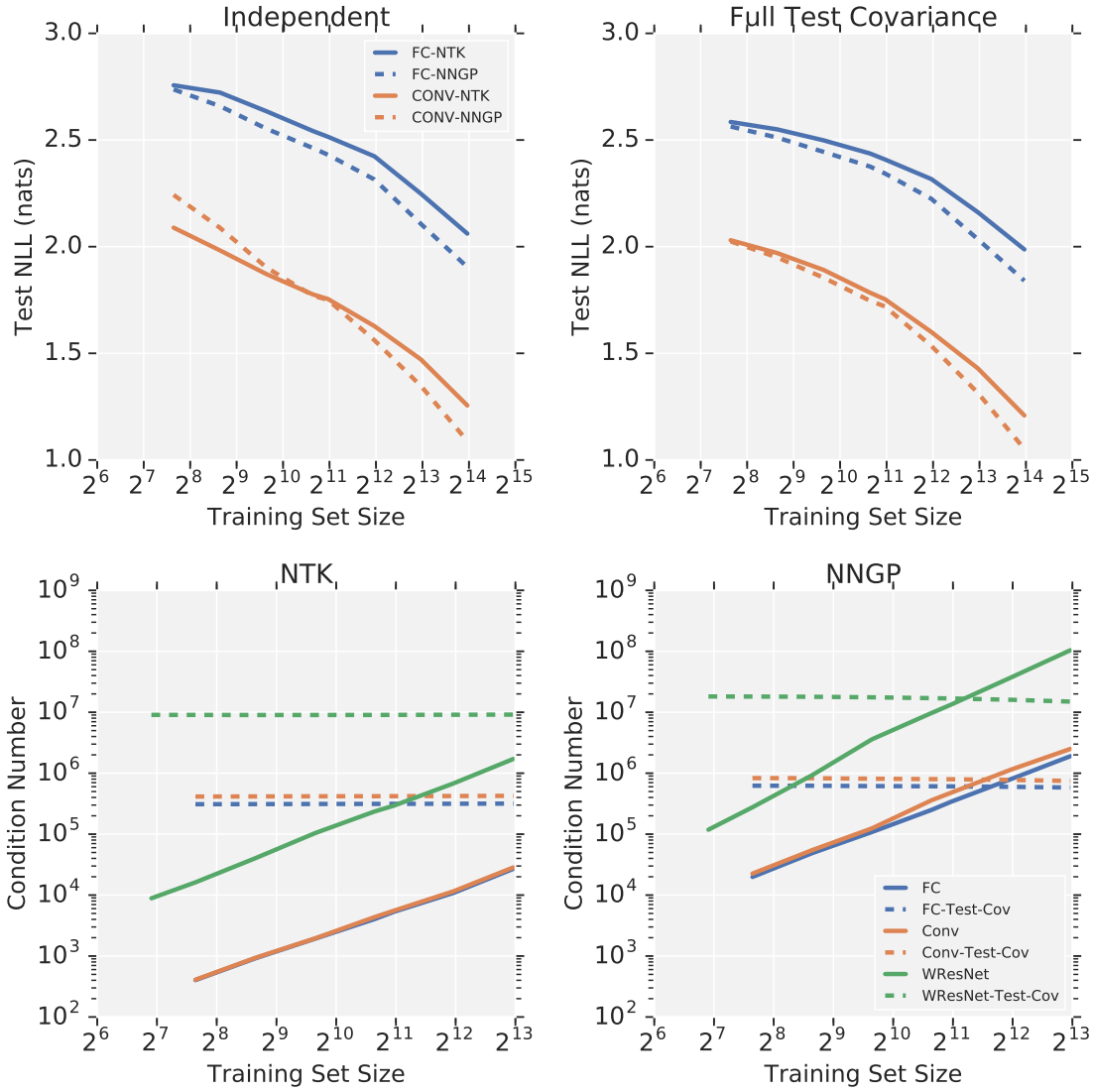


Figure 7: **Predictive negative log-likelihoods and condition numbers. Top.** Test negative log-likelihoods for NNGP posterior and Gaussian predictive distribution for NTK at infinite training time for CIFAR-10 (test set of 2000 points). **Fully Connected (FC, Listing 3)** and **Convolutional network without pooling (CONV, Listing 2)** models are selected based on train marginal negative log-likelihoods in Figure 2. **Bottom.** Condition numbers for covariance matrices corresponding to NTK/NNGP as well as respective predictive covaraince on the test set. Ill-conditioning of **Wide Residual Network** kernels due to pooling layers (Xiao et al., 2019) could be the cause of numerical issues when evaluating predictive NLL for this kernels.

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	$\mathcal{T}(\mathcal{K})$	$\dot{\mathcal{T}}(\mathcal{K}) \odot \Theta$
$\text{Dropout}(\rho)$	$\mathcal{K} + \left(\frac{1}{\rho} - 1\right) \text{Diag}(\mathcal{K})$	$\Theta + \left(\frac{1}{\rho} - 1\right) \text{Diag}(\Theta)$
$\text{Conv}(\sigma_w, \sigma_b)$	$\sigma_w^2 \mathcal{A}(\mathcal{K}) + \sigma_b^2$	$\sigma_w^2 \mathcal{A}(\mathcal{K}) + \sigma_b^2 + \sigma_w^2 \mathcal{A}(\Theta)$
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	$\text{GlobalAvgPool}(\mathcal{K})$	$\text{GlobalAvgPool}(\mathcal{K} + \Theta)$
$\text{Attn}(\sigma_{QK}, \sigma_{OV})$ (Anonymous, 2020)	$\text{Attn}(\sigma_{QK}, \sigma_{OV})(\mathcal{K})$	$2\text{Attn}(\sigma_{QK}, \sigma_{OV})(\mathcal{K}) +$ $\text{Attn}(\sigma_{QK}, \sigma_{OV})(\Theta)$
$\text{FanInSum}(\mathcal{X}_1, \ldots, \mathcal{X}_n)$	$\sum_{j=1}^n \mathcal{K}_j$	$\sum_{j=1}^n \Theta_j$
$\text{FanOut}(n)$	$[\mathcal{K}] * n$	$[\Theta] * n$

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 §D 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 $[\mathcal{A}(\Sigma)]^{w,w'}_{h,h'}(x, x') = \sum_{dh,dw} [\Sigma]^{w+dw,w'+dw}_{h+dh,h'+dh}(x, x')/q^2$, 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). $[\Sigma] * n = [\Sigma, \ldots, \Sigma]$ (n-fold replication). See Figure 3 for an example of applying the translation rules to a specific model, and §E for deriving a sample translation rule.