On Transfer Learning via Linearized Neural Networks

Wesley J. Maddox[*1] **Shuai Tang**[*2] **Pablo Garcia Moreno**[3]
Andrew Gordon Wilson[1] **Andreas Damianou**[3]

[1] New York University, New York, NY
[2] UCSD, San Diego, CA
[3] Amazon, Cambridge, UK

Abstract

We propose to linearize neural networks for transfer learning via a first order Taylor approximation. Making neural networks linear in this way allows the optimization to become convex (or even closed form) across several tasks. Not only does this vastly simplify the problem, but it allows us to rephrase transfer learning as sharing hyper-parameters across Gaussian processes, which can be solved using standard numerical linear algebra methods. Probabilistically, the framework is interpreted as a Gaussian process model with finite Neural Tangent Kernels. Our aproach is fast not only thanks to the linearization, but also because we leverage numerical results from relating the Fisher Information Matrix to the NTK.

1 Introduction

Typical deep transfer learning approaches implicitly assume that features learned by the source task model are relevant to the target tasks. Recent approaches collect such features by computing the Jacobian matrix of the data with respect to the model's parameters [11, 1]. Across several related tasks, we then assume that the Jacobian for each specific task will be similar to the Jacobian for any other task as each task is considered to be drawn from a distribution over tasks.

An interesting property of the Jacobian matrix is that it provides an approximate linearization to even very non-linear models. This allows us to write down an approximate version of the non-linear model as a (Bayesian) linear regression using the Jacobian matrix of the parameters as the features. We can then compute in closed form the optimal solution (along with predictive variances) to this problem by solving a system of equations. We show that this formulation is the dual parameter-space view of the recently proposed neural tangent kernel (NTK) [7, 9], which argues that under gradient descent training, many sufficiently wide neural networks evolve as if they were in fact linear models (e.g. despite their obvious nonlinearities, the functions induced by neural networks evolve like they are produced by linear models). We can then naturally pose the linearized problem as a degenerate Gaussian process (GP) which is used as the probabilistic model in our approach.

Under the light of the aforementioned connections, *our key idea is to enable fast, closed-form transfer learning among different models by: (a) mapping each model to a linear system through linearization with the Jacobian (b) embedding the approach into a probabilistic framework using degenerate GPs with NTK kernel*. In contrast to using similarities across features, solutions to the linearized systems are obtained through fast iterative solves of linear systems. Our linearization framework opens up the way for further speed-ups by leveraging numerical tricks. Specifically, we accelerate the required matrix-vector multiplications with the Jacobian by transforming them into Fisher-vector products.

[*]Work performed during an internship at Amazon. Correspondence to wjm363 AT nyu.edu

3rd Workshop on Meta-Learning at NeurIPS 2019, Vancouver, Canada.

The speed-up is obtained by approximating these products leveraging the connection of the Fisher information matrix to the gradient of KL divergence.

2 Methods

2.1 Gaussian Processes from Linearized DNNs

Deep neural networks are arbitrarily compositions of simple functions. As such, they are highly nonlinear. We define a deep neural network, f, as taking inputs, $\boldsymbol{X} \in \mathbb{R}^d$, and mapping it to an output, $\boldsymbol{y} \in \mathbb{R}^o$, with parameters $\theta \in \mathbb{R}^p$, letting $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$. Although f is arbitrarily non-linear, recent theoretical work, for example, [9] has argued that relatively wide neural networks evolve as *linear* models under gradient descent training dynamics. That is, the functions produced by wide neural networks change throughout training as if they were simply functions produced by linear models. Following [9], we can linearize f around its parameters using the Jacobian matrix, $\boldsymbol{J}_\theta(x) = (\nabla_\theta f(x))^T$, as $f_\theta(x) \approx \boldsymbol{J}_\theta(x)^T\theta$ (a standard first order Taylor expansion). For n data points of a training set, the Jacobian matrix is $p \times n \times o$. For regression $o = 1$.

To perform probabilistic modeling with the linearized model we then assume a likelihood, e.g. $y \sim \mathcal{N}(\boldsymbol{J}_\theta^T\theta', \sigma^2\boldsymbol{I}_p)$ for regression, which uses the Jacobian matrix as the features. A basis function linear model (in parameter space) is produced. Further, by assuming a Gaussian prior on the weights, $\theta' \sim N(0, \boldsymbol{I}_p)$, conjugacy can be exploited (see e.g. Chapter 2 of [16]) to give the posterior in parameter space as, $\theta' \sim \mathcal{N}((\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T + \sigma^2\boldsymbol{I}_p)^{-1}\boldsymbol{J}_\theta^T y, (\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T + \sigma^2\boldsymbol{I}_p)^{-1})$. The posterior predictive distribution is then simply:

$$f^*|x^*, \mathcal{D} \sim \mathcal{N}(\boldsymbol{J}_\theta^{*T}(\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T + \sigma^2\boldsymbol{I}_p)^{-1}\boldsymbol{J}_\theta^T y, \sigma^2\boldsymbol{J}_\theta^{*T}(\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T + \sigma^2\boldsymbol{I}_p)^{-1}\boldsymbol{J}_\theta^*).$$

We can now clearly see that inference (and predictive variance computation) only involves inverting a $p \times p$ matrix, e.g. *solving the linear system* $(\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T + \sigma^2\boldsymbol{I}_p)x = b$, with the Gram matrix, $\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T$. Naively, this requires $\mathcal{O}(p^3)$ time. Fortunately, we can flip to the dual function space [16], interpreting $\boldsymbol{J}_\theta$ as producing a degenerate Gaussian process with kernel matrix $\boldsymbol{J}_\theta^T\boldsymbol{J}_\theta$. This inner product is simply the Neural Tangent Kernel (NTK) of [7]. Using Woodbury's matrix identity, the posterior predictive distribution can be re-written as

$$f^*|x^*, \mathcal{D} \sim \mathcal{N}(\boldsymbol{J}_\theta^{*T}\boldsymbol{J}_\theta(\boldsymbol{J}_\theta^T\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)^{-1}y, \sigma^2\boldsymbol{J}_\theta^{*T}(\boldsymbol{I}_p - \boldsymbol{J}_\theta(\boldsymbol{J}_\theta^T\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)^{-1}\boldsymbol{J}_\theta^T)\boldsymbol{J}_\theta^*).$$

Again, inference only requires solving the linear system $(\boldsymbol{J}_\theta^T\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)x = b$, naively requiring $\mathcal{O}(n^3)$ time. Given the size of modern deep neural networks (i.e. number of parameters p is much greater than the number of data points, n), solving this system is preferable. To conclude, by linearization, we can either perform inference by solving linear systems in either function space ($n \times n$ matrices) or parameter space ($p \times p$ matrices).

For transfer learning tasks, we will distinguish between the pre-trained parameters, θ, (assumed fixed for now)[2] and the parameters, θ', arising from the linearization. Naturally, θ becomes the hyper-parameters for the linear model, and are shared across tasks.

Reverse mode automatic differentiation (see Appendix for further details) allows computing matrix vector products ($\boldsymbol{J}_\theta v$, $\boldsymbol{J}_\theta^T v$) in linear (in the cost of evaluating the function) time. We can then use iterative methods such as conjugate gradients and Lanczos to reduce the computational complexity to $\mathcal{O}(p^2 r)$, where r is the number of steps used, and avoid ever forming the matrices explicitly [17, 5, 15].

2.1.1 Why Linearize?

We next consider why we would wish to linearize non-linear models such as neural networks for transfer learning. First, we wish to have a kernel space representation as this allows us to naturally view transfer learning, for fixed parameters, in closed form. Using either representation, we get the optimal set of (transfer) parameters for linear models (and only have to solve a convex optimization problem for most other loss functions). This is an advantage of the kernel representation that is not present in many other meta-learning algorithms.

[2] It is possible to backpropagate through the GP regression, as hyper-parameter training. See [16] and [5] for the derivatives. However, we have not explored it yet.

2.2 Similarity of Jacobian Matrices Across Tasks

Empirical evidence has shown that features across neural networks are transferable - see e.g. [3] who focused on the efficacy of unsupervised pre-training for transfer learning. Similarly, [21, 10] found that features learned by neural networks, particularly in early convolutional layers, were transferable across both tasks and architectures. For a given architecure (which will have the same parameters), we may then expect that the Jacobian matrix, which is made up of the derivatives of features (a linear transformation), will then have similar structure across tasks.

More recently, (diagonal) Fisher matrices for classifiers have been shown to be informative for estimating similarity across tasks. In Task2vec, [1] proposed using the cosine similarity of different tasks' Fisher information matrices to measure the similarity between tasks. Note that Fisher information matrices can be approximated in explicit terms of the empirical distribution function (termed the *empirical Fisher information* [3]):

$$\begin{aligned}\mathbb{F}(\theta) &= \mathbb{E}_{p(x)p(y|x)}\left(\nabla_\theta \log p(y|x,\theta)\nabla_\theta \log p(y|x,\theta)^T\right) \\ &= \mathbb{E}_{p(x)}\left(\boldsymbol{J}_\theta(x)\boldsymbol{H}_\theta(\mathcal{X})\boldsymbol{J}_\theta(x)^T\right) \approx n\boldsymbol{J}_\theta(\mathcal{X})\boldsymbol{H}_\theta(\mathcal{X})\boldsymbol{J}_\theta(\mathcal{X})^T, \end{aligned} \tag{1}$$

where n is the number of data points and $\boldsymbol{H}_\theta(x) = \mathbb{E}_{p(y|x,\theta)}(\nabla_f \log p(y|f(x;\theta))\nabla_f \log p(y|f(x;\theta))^T)$. Note that $\boldsymbol{H}_\theta$ is an expectation over the derivatives over the last layer, by chain rule, and is both block-diagonal and can be computed in closed form. Thus, by assuming that $\mathbb{F}(\theta)$ is similar across different datasets, we are implicitly assuming that $\boldsymbol{J}_\theta$ is also similar across different tasks.

We focus on homoscedastic regression, where $y \sim \mathcal{N}(f(x), \sigma^2 I)$ so that $\boldsymbol{H}_\theta = \boldsymbol{I}_n$. Then, the Fisher information matrix is $n\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T$, while the neural tangent kernel (and the linearization) is $\boldsymbol{J}_\theta^T\boldsymbol{J}_\theta$. The two matrices are then *similar* and have the same eigenvalues (up to a constant factor). See further examples in the Appendix.

2.3 Probabilistic Model Over Tasks

Using the empirical evidence above, we can assume that transfer learning follows a hierarchical model over datasets themselves - namely that the singular values of Jacobian matrices from similar tasks will be closely related. First, we sample a task index, t, before sampling a dataset, $\mathcal{D}_t \sim p(\mathcal{D}|t)$, given the task index. We then sample the Jacobians given this task before defining a linear model (degenerate Gaussian process) for the specific response y_t. See Appendix B for further details.

3 Experiments

Transferring across Sinusoids: In this experiment, we transfer across learning tasks by directly using the neural tangent kernel as the prior for different regression settings. In Figure 1, we view each successive task as an independent draw from a Gaussian process with a neural tangent kernel. We replicate the generative process of [8], with different periods for the sine curves. We generate 10 datasets using this process and train only on the first dataset (the red points), using a three layer MLP with Tanh activations. In the Appendix, we show similar predictive means using the parameter space approximation (along with ReLU activations); however, it is slower for larger networks due to performing inverses in parameter space, rather than function space.

The network's predictions are shown as the blue line across Figure 1 while the corresponding GP posterior mean and confidence region from the NTK is shown in green. Each of the panels additionally shows the 10 context points (in purple) and the training points (orange) from evaluating the observed function values as the new training set. No parameter updating was performed - only calls to the NTK. We can see that for related functions - here sinusoids with different periods, the features learned by a MLP are transferable across tasks, and that the NTK representation gives reasonable posterior predictive *distributions*.

Precipitation Experiments: To demonstrate the importance of including predictive variances in regression, we next perform few shot learning on a real world climate time series dataset of averaged

[3] Not to be confused with the observed Fisher information, which is defined directly as $\boldsymbol{J}_\theta\boldsymbol{J}_\theta^T$. In regression, the two matrices do correspond.

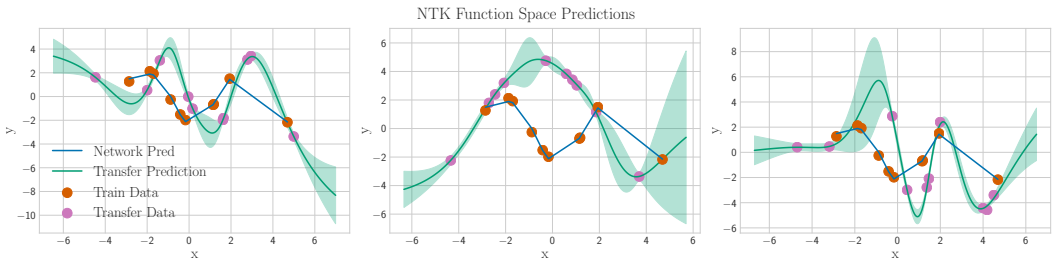

Figure 1: Posterior predictions on a few shot regression task produced with the NTK as the kernel.

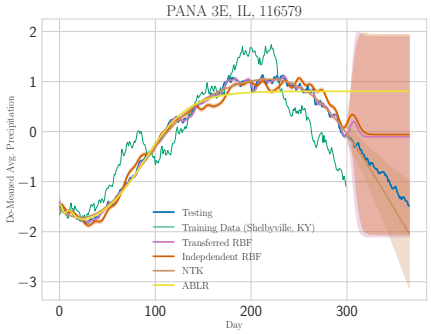

Figure 2: Posterior distributions for the NTK as compared to an RBF kernel trained on the source task, independently trained RBF GPs, and ABLR [14] on a selected transfer task (precipitation in Pana, IL). We additionally show the training data (on which the base neural network is trained).

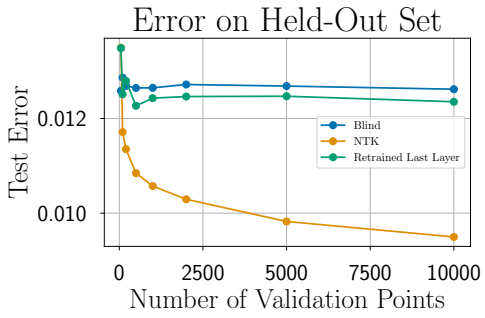

Figure 3: Error on held out set for infection rate of *Plasmodium falciparium* among African children. The NTK improves its accuracy on the held-out set as the number of validation data points from 2016 are given to it, while re-training the last layer stagnates.

daily precipitation from the publicly available US Historical Climatology Network [12]. We train on the first 10 months of single time series, before using the learned parameters as parameters for the prior Jacobian matrix for each successive time series, visualizing the results. Finally, for all time series, we test on the last 2 months of the year, showing visually the results (against independently trained models) in Figure 2. We show further plots in the Appendix, including the source task.

Malaria Dataset Inspired by Cutajar et al. [4], Balandat et al. [2], we used data describing the infection rate of *Plasmodium falciparium* (a parasite known to cause malaria) drawn from the Malaria Global Atlas[4] in a transfer learning set-up. We trained a heteroscedastic neural network with three layers on 2000 data points from the 2012 map, before testing on 5000 data points from the 2016 map. We considered no re-training (Blind), the finite neural tangent kernel (NTK), and retraining the final layer as function of the validation points given from the 2016 map in Figure 3. It is possible to see that the performance of the finite NTK as a transfer model improves as a number of data points, while the re-trained last layer stagnates.

4 Conclusion

We have embedded into a degenerate GP probabilistic model finite Neural Tangent Kernels and utilized the standard parameters of neural networks as the shared hyper-parameters across tasks. The whole framework is fast not only thanks to the linearization, but also because we leverage numerical results from relating the Fisher Information Matrix to the Neural Tangent Kernel. Future work will extend this method to classification via approximate likelihood approaches such as [13], alongside extending the method to be fully differentiable (enabling meta-learning) - as the GP derivatives are already closed form even when using conjugate gradients [5].

[4]Extracted from `https://map.ox.ac.uk`.

Acknowledgements

WJM was supported by an NSF Graduate Research Fellowship under Grant No. DGE-1650441. AGW was supported by an Amazon Research Award, Facebook Research, NSF IIS-1563887, and NSF IIS-1910266. We would like to thank Jacob Gardner and Alex Wang for GPytorch discussions and Tim Rudner for helpful discussions.

References

[1] A. Achille, M. Lam, R. Tewari, A. Ravichandran, S. Maji, C. Fowlkes, S. Soatto, and P. Perona. Task2vec: Task Embedding for Meta-Learning. *arXiv:1902.03545 [cs, stat]*, Feb. 2019. URL http://arxiv.org/abs/1902.03545.

[2] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy. Botorch: Programmable bayesian optimization in pytorch. *arXiv preprint arXiv:1910.06403*, 2019.

[3] Y. Bengio. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Workshop on Unsupervised and Transfer Learning*, volume 27, page 21. PMLR W&CP, 2012.

[4] K. Cutajar, M. Pullin, A. Damianou, N. Lawrence, and J. González. Deep gaussian processes for multi-fidelity modeling. In *Bayesian Deep Learning Workshop at NeurIPS*, 2018. arXiv preprint arXiv:1903.07320.

[5] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*, volume arXiv:1809.11165 [cs, stat], Sept. 2018. URL http://arxiv.org/abs/1809.11165.

[6] G. H. Golub and V. Pereyra. The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems Whose Variables Separate. *SIAM Journal on Numerical Analysis*, 10(2):413–432, Apr. 1973. ISSN 0036-1429, 1095-7170. doi: 10.1137/0710036. URL http://epubs.siam.org/doi/10.1137/0710036.

[7] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.

[8] T. Kim, J. Yoon, O. Dia, S. Kim, Y. Bengio, and S. Ahn. Bayesian Model-Agnostic Meta-Learning. In *Advances in Neural Information Processing Systems*, volume arXiv:1806.03836 [cs, stat], June 2018. URL http://arxiv.org/abs/1806.03836.

[9] J. Lee, L. Xiao, S. S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington. Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent. *arXiv:1902.06720 [cs, stat]*, Feb. 2019. URL http://arxiv.org/abs/1902.06720.

[10] Y. Li, J. Yosinski, J. Clune, H. Lipson, and J. Hopcroft. Convergent Learning Do different neural networks learn the same representations? In *The 1st International Workshop on Feature Extraction: Modern Questions and Challenges*, volume 44, page 17. PMLR W&CP, 2015.

[11] T. Liang, T. Poggio, A. Rakhlin, and J. Stokes. Fisher-rao metric, geometry, and complexity of neural networks. In *Artificial Intelligence and Statistics*, volume arXiv preprint arXiv:1711.01530, 2017.

[12] M. Menne, C. Williams Jr, and R. Vose. *The United States Historical Climatology Network (USHCN) Main Page*. 2009. URL https://cdiac.ess-dive.lbl.gov/epubs/ndp/ushcn/ushcn.

[13] D. Milios, R. Camoriano, P. Michiardi, L. Rosasco, and M. Filippone. Dirichlet-based Gaussian Processes for Large-scale Calibrated Classification. In *Advances in Neural Information Processing Systems*, page 11, 2018. URL https://papers.nips.cc/paper/7840-dirichlet-based-gaussian-processes-for-large-scale-calibrated-classification.

[14] V. Perrone, R. Jenatton, M. W. Seeger, and C. Archambeau. Scalable Hyperparameter Transfer Learning. In *Advances in Neural Information Processing Systems*, page 11, 2018. URL https://papers.nips.cc/paper/7917-scalable-hyperparameter-transfer-learning.

[15] G. Pleiss, J. R. Gardner, K. Q. Weinberger, and A. G. Wilson. Constant-Time Predictive Distributions for Gaussian Processes. In *Artificial Intelligence and Statistics*, volume arXiv:1803.06058 [cs, stat], Mar. 2018. URL http://arxiv.org/abs/1803.06058.

[16] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass., 3. print edition, 2008. ISBN 978-0-262-18253-9.

[17] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, Philadelphia, Pa, 2. ed edition, 2003. ISBN 978-0-89871-534-7.

[18] A. Tosi. *Visualization and Interpretability in Probabilistic Dimensionality Reduction Models*. PhD Thesis, Universitat Politecnica de Catalunya, 2014.

[19] A. Tosi, S. Hauberg, A. Vellido, and N. D. Lawrence. Metrics for Probabilistic Geometries. In *Uncertainty in Artificial Intelligence*, Nov. 2014. URL http://arxiv.org/abs/1411.7432.

[20] J. Townsend. *A new trick for calculating Jacobian vector products*. June 2017. URL http://j-towns.github.io/2017/06/12/A-new-trick.html.

[21] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, page 9, 2014. URL http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf.

A Computational Considerations for Jacobian Matrices

Exact computation requires $\mathcal{O}(N)$ backwards passes (and it is not known in closed form); however, we can explicitly compute Jacobian vector products (e.g. Jv) (pearlmutter ref) with tape based autograd as implemented in most modern automatic differentiation software [5]

However, it is also necessary to compute vector products with the transpose of the Jacobian matrix, e.g. $J^T v$. It is in fact possible to do this with a second backwards call in forward mode autograd software [20]. Succinctly, this vector product consists of a first Jacobian vector product of the function with respect to ones, keeping the computation graph open, before a second backwards call against the output of the Jacobian vector product.

Thus, by stacking two Jacobian vector products (one against the Jacobian and one against the transpose), it is possible to efficiently compute $J^T Jv$ in just *three* backwards calls from the network. Given matrix-vector multiplies, we can now use efficient implementations of conjugate gradients and stochastic Lanczos quadrature as implemented in GPytorch [5] for efficient inference. **That is, we never have to explicitly form either $J^T J$ or JJ^T in order to perform inference in either weight or function space.** As demonstration of this capability, in Figure 4, we show the computation time for log probabilities with the NTK plotted against the number of data points for a four-layer MLP (5-200-2000-200-1) with $> 800,000$ parameters on a single GPU.

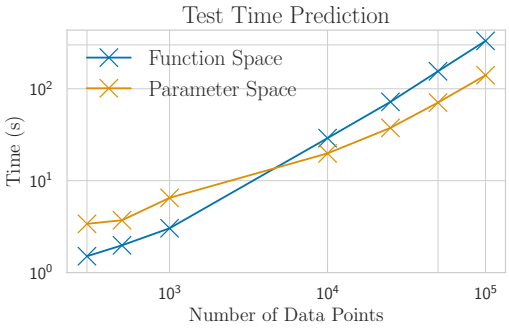

Figure 4: Scaling time of log probability calculation for an MLP using the NTK. Note the nearly linear computation time for this model. Memory issues in the forwards pass become the bottleneck to hold more data. Note that function space inference uses Jacobian vector products while parameter space inference uses Fisher vector products (see Appendix D).

The only limit here is the maximum size of the batch that can be used with the neural network. The Jacobian vector products can alternatively be batched, requiring $\mathcal{O}(B^2)$ backwards calls which can be done in parallel.

B Probabilistic Model over Tasks

In our probabilistic model over tasks we assume that tasks are drawn in the following way. First, we sample a task index, t, before sampling a dataset, $\mathcal{D}_t \sim p(\mathcal{D}|t)$, given the task index. For each individual task and dataset, $\mathcal{D}$, we have the likelihood $p(\boldsymbol{y}_t|\boldsymbol{X}_t) = p(\boldsymbol{y}_t|f_{\theta_t}(\boldsymbol{X}_t))$. Further, we linearize the neural network by defining a fixed basis function approximation, $f_t \approx \boldsymbol{J}_\theta(\boldsymbol{X}_t)\theta'_t + \mu(\boldsymbol{X}_t)$, where $\boldsymbol{J}_\theta(\boldsymbol{X}) = (\nabla_\theta f(\boldsymbol{X}))^T \in \mathbb{R}^{p\times on}$, and θ'_t are the parameters of the (Bayesian) linearized model. Notice that the Jacobian computation depends only on the parameters θ of the pre-trained network. This allows us to represent the functions f_t (for other tasks) in a way that does not involve a new non-convex optimization.

To complete the probabilistic model, for each task we further assume a prior distribution $\theta'_t \sim \mathcal{N}(0, \boldsymbol{I})$, resulting in the following hierarchical model:

$$\begin{aligned} \theta'_t &\sim p(\theta'_t) \\ f_t &= J_\theta(\boldsymbol{X}_t)^T \theta'_t + \mu_t \\ \boldsymbol{y}_t|f_t &\sim p(\boldsymbol{y}_t|f_t), \end{aligned} \tag{2}$$

indexing each task by t and including a bias term, μ, in the model. If we assume a Gaussian likelihood $\boldsymbol{y}_t \sim N(\boldsymbol{J}_\theta^T \theta'_t, \sigma_t^2 \boldsymbol{I}_n)$, where we have used the shorthand $\boldsymbol{J}_\theta = \boldsymbol{J}_\theta(\boldsymbol{X}_t)$, the posterior over θ'_t is also Gaussian

[5] e.g. Pytorch, Tensorflow.

Algorithm 1 Fast Adaptation Procedure

Input: Data (X_1, Y_1), Initial parameters θ_0
Compute θ_{MLE} with data $(\boldsymbol{X}_1, \boldsymbol{y}_1)$.
for $i = 1$ **to** N_{tasks} **do**
  Compute $p(\theta_t'|\mathcal{D}_t)$ using $\boldsymbol{J}_{\boldsymbol{X}_t}$, $\boldsymbol{y}_t$, and Eq. 3 .
  Compute $p(f_t^*|\mathcal{D}_t) = \int p(f^*|\theta_t')p(\theta_t'|\mathcal{D}_t)d\theta'$
end for

and can be computed in closed form:

$$\theta_t'|\mathcal{D}_t, \theta \sim \mathcal{N}((\boldsymbol{J}_\theta \boldsymbol{J}_\theta^T + \sigma_t^2 \boldsymbol{I}_p)^{-1} \boldsymbol{J}_\theta^T (y_t - \mu_t), \\ \sigma_t^2 (\boldsymbol{J}_\theta \boldsymbol{J}_\theta^T + \sigma_t^2 \boldsymbol{I}_p)^{-1}). \tag{3}$$

For other likelihoods, we need to resort to approximate inference.

Thus, our procedure will require training a network on an initial task to get parameters θ.[6] Then, across the successive tasks, we will compute the Jacobian matrix of the trained network on the new data and compute the posterior predictive distribution in closed form (since θ' is derived analytically from equation equation 3). This procedure is explained in Algorithm 1.

C Relationship of the FIM and the finite NTK for General Losses:

C.1 Relating the Fisher Information Matrix to the NTK

Regression: An interesting example is given by homoscedastic regression, where $y \sim \mathcal{N}(f(x), \sigma^2 \boldsymbol{I})$. In this case, $\boldsymbol{H}_\theta = \boldsymbol{I}_n$. Then, the empirical Fisher information matrix is $\frac{1}{n} \boldsymbol{J}_\theta \boldsymbol{J}_\theta^T$, while the neural tangent kernel (and the linearization) is $\boldsymbol{J}_\theta^T \boldsymbol{J}_\theta$. The Fisher information and the NTK then have the same eigenvalues (up to a constant factor of n) as they are similar matrices[7].

General Losses: For general losses, it is still possible to relate the Fisher information matrix to either the Gram matrix or the NTK. Note that $\boldsymbol{H}_\theta(x)$ is block-diagonal and positive definite if the likelihood can be written to factorize across data points (e.g. the responses are i.i.d). We can parameterize the empirical Fisher in terms of the eigen-decomposition, $\mathbb{F}(\theta) \approx \boldsymbol{J}_\theta \boldsymbol{H}_\theta \boldsymbol{J}_\theta^T = S \Lambda S^T$, and noting that $\boldsymbol{H}_\theta = L L^T$ (its root or Cholesky decomposition). From before, we can follow the same trick, writing $\boldsymbol{J}_\theta L = S \Lambda^{1/2} Q^T$, where Q is another basis. Then $\boldsymbol{J}_\theta = S \Lambda^{1/2} \tilde{Q}^T$, with $\tilde{Q} = L^{-T} Q$, giving that

$$\boldsymbol{J}_\theta^T \boldsymbol{J}_\theta = \tilde{Q} \Lambda \tilde{Q}^T = L^T Q \Lambda Q^T L$$

and that

$$\boldsymbol{J}_\theta \boldsymbol{J}_\theta^T = S \Lambda^{1/2} Q^T \boldsymbol{H}_\theta^{-1} Q \Lambda^{1/2} S^T.$$

For practical usage, it may be difficult to compute the basis Q without requiring further Fisher vector or Jacobian vector products. One intriguing possibility is a randomized SVD on $\boldsymbol{J}_\theta L$, possibly computing L in closed form; however, we leave this for future work.

C.2 Structure of Fisher Information over Multiple Tasks

Naturally, the generative process over datasets given in Eq. 2 explicitly codifies our beliefs that multiple tasks are related to each other. While it would be possible to infer a posterior distribution over tasks via a generative modelling approach, this is not necessary (and could be quite intractable). Instead, assuming this probabilistic generative model, we can assume that parameters of f are shared across tasks, e.g. that $\theta_t = \theta$. This allows us to write down the Fisher information of the probabilistic model as an expectation across tasks, and so that the multi-task empirical Fisher information will simply be the average across all tasks, $\mathbb{F}(\theta) \approx \frac{1}{T} \sum_{t=1}^{T} \mathbb{F}_{emp,t}(\theta)$.

We can define this multi-task Fisher information as a probabilistic metric tensor (e.g. Tosi et al. [19], Tosi [18]) over several different tasks, which are simply random samples (random metric tensors) from a distribution over random positive semi definite matrices. Then, the local geometry of each task is driven by the components of this random metric tensor, the task specific Fisher information, $\mathbb{F}_{emp,t}(\theta)$.

[6] Note that in theory, there is no need to train the network at all. We found that it is practically useful to train the network to learn good representations.

[7] Note that similar connections between the Jacobian and the Fisher information matrix are utilised by Tosi et al. [19], and the connection seems to originate in the generalized Gauss-Newton decomposition of Golub and Pereyra [6].

D Fast Fisher Vector Products

We now describe how to compute Fisher matrix vector products calls in simply *two* backwards calls. We can use directional derivatives to approximate a Fisher vector product. The second order Taylor expansion between two distributions is given by:

$$D_{\mathrm{KL}}(p(y \mid \theta)||p(y \mid \theta')) = \frac{1}{2}(\theta - \theta')^T \mathbb{F}(\theta)(\theta - \theta') + \mathcal{O}(\theta - \theta')^3.$$

Evaluating the derivative at $\theta' = \theta + \epsilon v$ gives:

$$\nabla D_{\mathrm{KL}}(p(y \mid \theta)||p(y \mid \theta'))|_{\theta'=\theta+\epsilon v} = \epsilon \mathbb{F}(\theta)v + \mathcal{O}\{\epsilon^2||v||\}, \tag{4}$$

which can therefore be used to compute Fisher vector products.

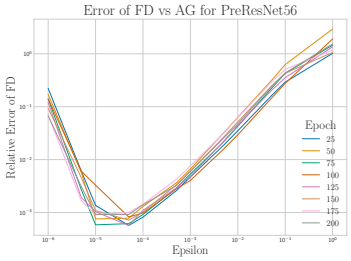

Figure 5: Accuracy of Fisher-vector products as a function of tuning parameter ϵ for Finite Differences (FD) versus AutoGrad (AG).

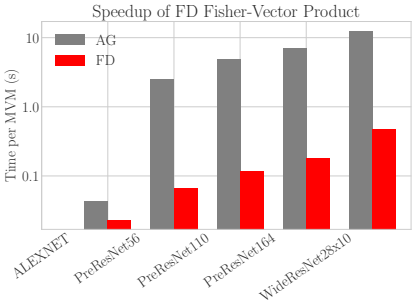

Figure 6: Speedup of finite differences (FD) Fisher-vector products through autograd (AG) for AlexNet, PreResNets of varying depth, and WideResNet on CIFAR10.

Illustrative Experiment: We demonstrate the approximation error between this directional derivative and the exact $F_i v$ computed using the standard second order autograd; this is shown in Figure 5. We used a modern neural network architecture, PreResNet56, on the benchmark CIFAR10 dataset and computed the relative error as a function of ϵ:

$$\frac{||(F_i \nabla f(x))_{AG} - (F_i \nabla f(x))_{FD(\epsilon)}||}{||(F_i \nabla f(x)_{AG}||},$$

through various stages of the standard training prodcedure with stochastic gradient descent. Crucially, we note that the relative error produced by this approximation is on the order of $1e-3$ and stays nearly constant throughout training, suggesting a simple procedure for tuning this hyper-parameter at the beginning of training. In Figure 6, we demonstrate that the directional derivative Fisher vector product is typically at least $25\times$ faster than the autograd Fisher vector product on modern DNN architectures including AlexNet, three types of PreResNets of varying depth, and a WideResNet on the same CIFAR10 dataset.

E Further Results with the NTK on Few Shot Regression

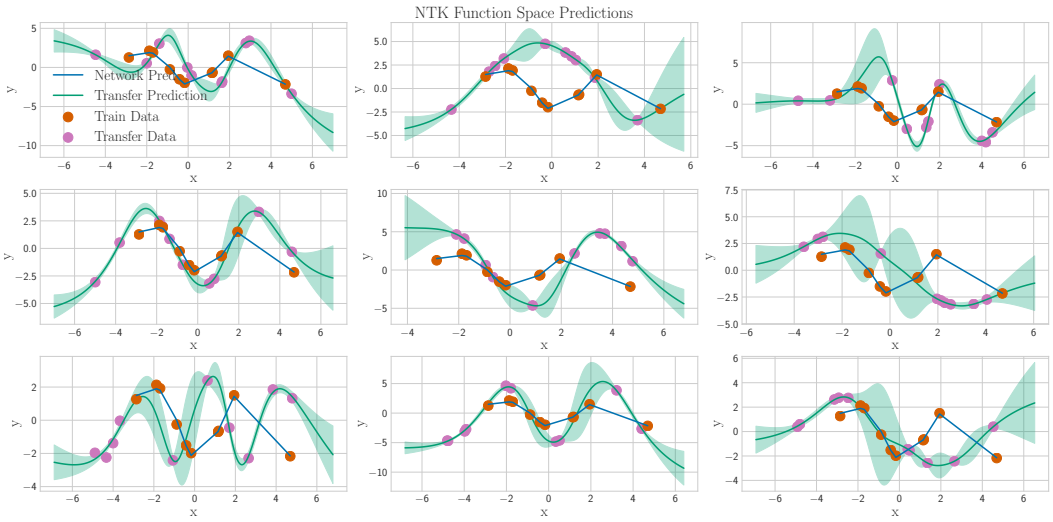

Figure 7: Posterior predictions on a few shot regression task produced with the NTK as the kernel.

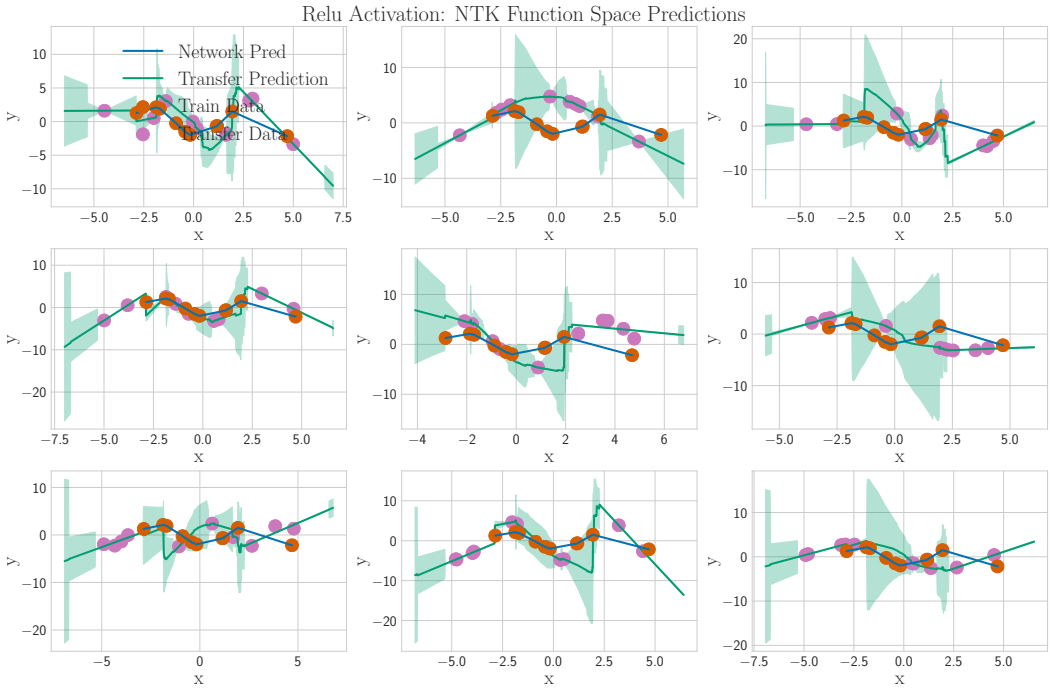

Figure 8: Posterior predictions on a few shot regression task produced with the NTK as the kernel using a network with ReLU activations.

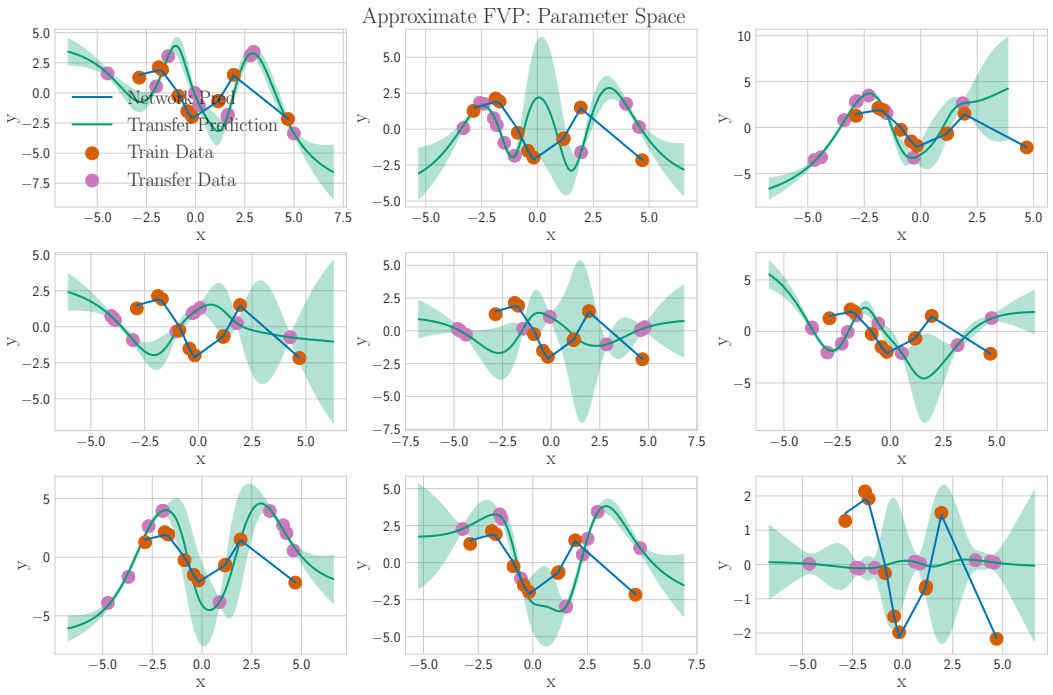

Figure 9: Posterior predictions on a few shot regression task produced with the NTK as the kernel using a network with ReLU activations.

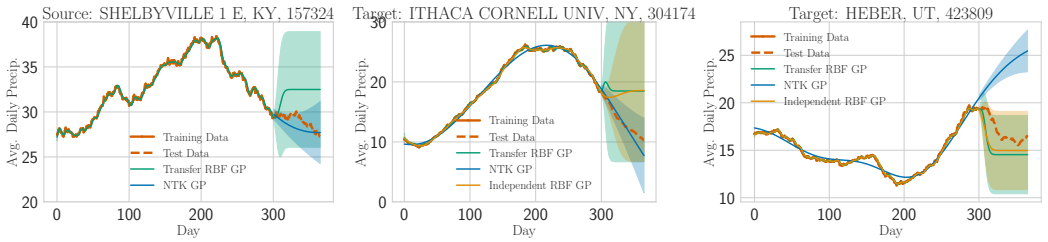

Figure 10: Posterior distributions for the NTK as compared to an RBF kernel trained on the source task, and independently trained RBF GPs. The NTK markedly improves the fit of the neural network on the source task, while additionally providing both uncertainty on this validation set and on one of the two randomly chosen transfer tasks, failing when the climate pattern is not related.

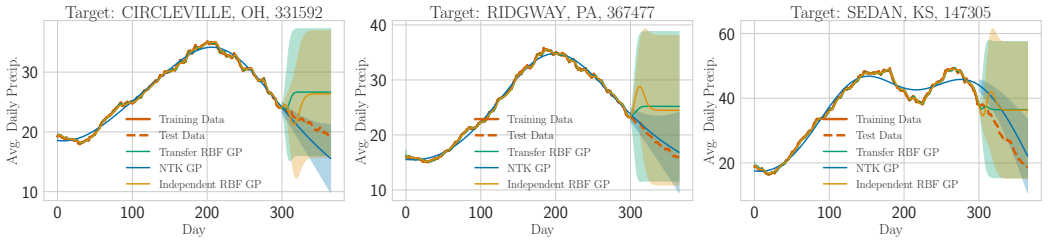

Figure 11: Posterior distributions for the NTK as compared to an RBF kernel trained on the source task, and independently trained RBF GPs. Here, we can see the effectiveness of sharing the NTK across multiple related tasks.

F Experimental Details

Sinusoidal Regression The generative process matches the generative process of [8], where we generate $x \sim \mathcal{N}(0, \boldsymbol{I}_{10})$ and then $y_i = A\sin(wx_i + b) + \epsilon_i$, where $A \sim U(0.1, 5.)$, $w \sim U(0, 2\pi)$ and $\epsilon \sim \mathcal{N}(0, 0.01A)$.

For the sinusoidal regression tasks, we follow the setup of [8], and utilize neural networks with two hidden layers and 40 hidden units and tanh activations (or ReLU for Figure 8). On the first task, we train the network with batch sizes of 3 for 2500 epochs using stochastic gradient descent with a learning rate of 1e-3 and momentum = 0.9. We then incorporate this into a NTK model and either solve the system and cache the predictive variances (as in [5]) using either function or parameter space.

For Figure 9, we use the approximate Fisher vector product describe in Appendix D on the regression loss with $\epsilon = 1e-4$. We found that ϵ was very stable for regression losses.

Precipitation Experiments For the precipitation experiments, we randomly chose a location of the 1209 locations (for a total of 362,700 data points seen as validation and 78585 data points seen as testing), trained using a single layer neural network with Tanh activations and 400 hidden units to approximate RBF kernel Gaussian processes. We standardized both the inputs and features to have zero mean and small variance for ease of training. To train the neural network, we ran SGD with momentum with batch sizes of 100 for 1000 epochs with a learning rate of 1e-4 and momentum = 0.9.

To train the RBF Gaussian processes, we fit using BoTorch's [2] scipy minimizer [8], again using fast predictive variances in GPyTorch for prediction. For the RBF GP shared across all tasks, we reset the training data for each successive task.

These experiments were performed on a single Nvidia Tesla V100 GPU.

Malaria Experiment For the Malaria global atlas experiment, we trained a single neural network with three hidden layers (2 - 500 - 500 -2) and tanh activations on 2000 randomly selected datapoints of the 2012 map in Nigeria (similar to Balandat et al. [2]), training with a heteroscedastic loss, so that the likelihood model was $(y|\mu_\theta(x), 1e-5 + \text{softplus}(\sigma_\theta(x)))$. We trained using SGD with momentum with batch sizes of 200 for 500 epochs decaying the initial learning rate from 1e-3 by a factor of 10 every 100 epochs. For the fine-tuned final layer, we continued training for 100 more epochs using the same procedure; we note that slightly better performance was acheived by training for 1000 epochs; however, this rapidly becomes an unfair comparison due to the increased training time. After all, if training for 1000 epochs, why not just train a full model?

This experiment was performed on a single Nvidia Tesla 1080 GPU, along with all other timing related experiments.

[8] `https://botorch.org/docs/optimization`