Fast Adaptation with Linearized Neural Networks

Wesley J. Maddox*
New York University
wjm363@nyu.edu

Shuai Tang*
UC San Diego
shuaitang93@ucsd.edu

Pablo Garcia Moreno
Amazon, Cambridge
morepabl@amazon.com

Andrew Gordon Wilson
New York University
andrewgw@cims.nyu.edu

Andreas Damianou
Amazon, Cambridge
damianou@amazon.com

*Work partly completed during internship at Amazon.

Abstract

The inductive biases of trained neural networks are difficult to understand and, consequently, to adapt to new settings. We study the inductive biases of linearizations of neural networks, which we show to be surprisingly good summaries of the full network functions. Inspired by this finding, we propose a technique for embedding these inductive biases into Gaussian processes through a kernel designed from the Jacobian of the network. In this setting, domain adaptation takes the form of interpretable posterior inference, with accompanying uncertainty estimation. This inference is analytic and free of local optima issues found in standard techniques such as fine-tuning neural network weights to a new task. We develop significant computational speed-ups based on matrix multiplies, including a novel implementation for scalable Fisher vector products. Our experiments on both image classification and regression demonstrate the promise and convenience of this framework for transfer learning, compared to neural network fine-tuning. Code is available at `https://github.com/amzn/xfer/tree/master/finite_ntk`.

Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130.

1 INTRODUCTION

Deep neural networks (DNNs) trained on a source task can be used for predicting in a new (target) task through a process which we interchangeably refer to as transfer learning or domain adaptation (Montavon et al., 2012; Bengio, 2012; Yosinski et al., 2014; Sharif Razavian et al., 2014). One family of approaches to solve this problem adapts the parameters of the full source task network, through gradient descent or more elaborate methods such as meta-learning (Finn et al., 2017). However, adaptation of the network is computationally demanding and prone to getting trapped in local minima. Moreover, fine-tuning the full network in excess can lose a useful representation that was learned from the source domain.

A compelling alternative is to fine-tune only the last layer of the source network, which leads to a log-convex problem that avoids local optima issues and, at the same time, provides a more computationally affordable and less data demanding solution. However, the resulting solution can lead to a poor fit when dealing with complex transfer learning tasks due to the lack of flexibility. Moreover, it is not clear how many of the last layers one would need to fine-tune and what are the overall inductive biases that are transferred to the target task in each case after these layers' parameters have been moved from their original states.

In this paper we aim to address the limitations of fine-tuning with minimal computational overhead, while keeping the performance competitive with costly and involved solutions based on adapting the full network. We propose to structure the transfer learning problem in the following simpler way: firstly, we linearize the DNN with a first order Taylor expansion, giving rise to a linear model whose inductive biases we study here

empirically. Secondly, we embed these inductive biases into a probabilistic lightweight framework which takes the form of a Bayesian linear model with the DNN Jacobian matrix $\boldsymbol{J}$ as the features or, equivalently, a Gaussian process (GP) with $\boldsymbol{J}^\top \boldsymbol{J}$ as the kernel. The kernel matrix is the finite width counterpart of the neural tangent kernel (NTK) (Jacot et al., 2018; Lee et al., 2019) but, crucially, our Taylor expansion is performed around a *trained* model, rather than a randomly initialized one. We then achieve fast adaptation by computing the Jacobian matrix on the target task and invoking the GP predictive equations.

We therefore cast domain adaptation as posterior inference in function space. This solution is analytic (and closed-form, for regression), hence by-passing local optima issues. It is also interpretable: firstly, all of our prior assumptions are encoded in the kernel of the GP, whose inductive biases we study; secondly, our probabilistic model gives a calibrated estimate of the uncertainty of the transfer task and closed form predictive distributions for regression. Figure 1 sums up our approach to transfer learning. Given a trained model at a global optimum (purple star), we transfer in function space using the linearized model to another optimum (red star) avoiding local optima issues incurred by transferring in parameter space using fine-tuning.

To make our approach competitive with respect to non-probabilistic domain adaptation methods we need to tackle one remaining challenge: scalable inference. To this end, we consider a black box inference framework which uses implicit Jacobian vector products and, hence, avoids forming the kernel explicitly. This is complemented by a novel implementation of a directional derivative based on Fisher vector products to enable fast black-box conjugate gradients optimization and test-time caching for fast predictive variances (Gardner et al., 2018; Pleiss et al., 2018). Overall, we are thus able to combine the benefits of probabilistic and neural network models without sacrificing scalability.

We consider applications in both regression and classification. Our key contributions are as follows:

- We study empirically the inductive biases of the linearized DNN model, demonstrating that the linearized model still maintains the strong capabilities of the full neural network for transfer learning.
- We include neural networks in a probabilistic framework by considering the aforementioned linearized DNN model. We develop techniques for fast inference, including a novel implementation for scalable Fisher vector products.
- We apply this probabilistic framework to domain adaptation, casting transfer learning as posterior inference in function space. We demonstrate that this produces strong performance compared to fine-tuning.

The rest of the paper is organized as follows: in Section 2 we summarize related work using the Jacobian matrix of neural networks, in Section 3 we describe how we efficiently use the Jacobian matrix in either a Bayesian (generalized) linear model or a degenerate Gaussian process. In Section 4, we test the inductive biases of these Jacobian kernels. Finally, in Section 5, we test their domain adaptation abilities, demonstrating that they are a strong principled baseline for fast adaptation.

2 RELATED WORK

Using the gradients of a model (such as a DNN) as features for classification problems has a long history. The Fisher kernel used both the Jacobian and inverse Fisher matrices of a generative unsupervised model as the kernel for support vector machines (SVMs) Jaakkola and Haussler (1998). More recently, Zhai et al. (2019) constructed Fisher vectors from GANs and used them to produce highly accurate linear models on CIFAR10. Zinkevich et al. (2017) demonstrated that linear models can be constructed from the Jacobians of DNNs to match the predictions of the full DNN on the training set. Tosi et al. (2014) and Tosi (2014) interpreted the Jacobian matrices for different tasks as creating a probabilistic metric tensor whose expectation is the average Fisher information across several tasks. Finally, we note that model linearization using the Jacobian can be linked to Laplace approximation (MacKay, 2003).

Empirical evidence has shown that features across neural networks are transferable (Bengio, 2012). In particular, Yosinski et al. (2014) and Li et al. (2015) found that features learned by neural networks were transferable across both tasks and architectures. These empirical results motivate our work as the Jacobian is a transformation of the features of the network. More recently, (diagonal) Fisher matrices for classifiers have been shown to be informative for estimating similarity across tasks (Achille et al., 2019). Mu et al. (2020) argue that the Jacobian is useful for representation learning because it is a local linearization of fine-tuning.

Jacot et al. (2018) showed that infinitely wide DNNs behaved as their associated Taylor expansion around *initialization*, terming the resultng kernel in the infinite limit the neural tangent kernel (NTK). Lee et al. (2019) extended their analysis showing that the evolution of finitely wide DNNs over the course of training is similar to that of linear models. Lee et al. (2019) and Arora et al. (2019) gave implementations of infinitely wide DNNs, demonstrating strong performance on CIFAR-

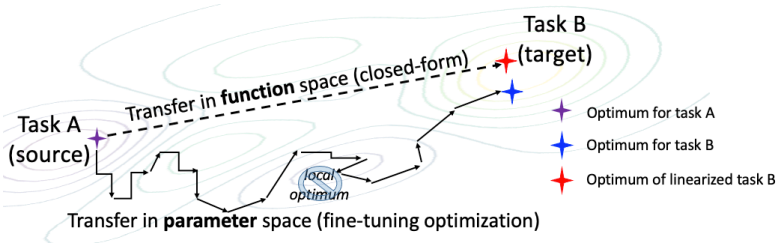


Figure 1: Schematic of our approach to probabilistic transfer learning. We show the loss surfaces for the model as we transfer from the source task (purple star) to the target task (red/blue stars). Linearizing the network allows for transferring in *function* space which gives a direct mapping to an optimum, while transferring in parameter space (e.g. fine-tuning) can get stuck in local optima.

10. Due to the analytic nature of these computations, the DNNs studied are restricted: allowing no batch normalization and a slim choice of layers, and use a different scaling than standard DNNs.

3 LINEARIZED NEURAL NETWORK MODEL

We first relate using the Jacobian matrix as features to degenerate Gaussian processes. Given an arbitrary neural network, f, with p parameters θ and n inputs $x = \{x_i\}_{i=1}^n$ with outputs in dimension o, we may Taylor expand f around θ in the following manner:

$$f(x;\theta') \approx f(x;\theta) + \boldsymbol{J}_\theta(x)^\top(\theta - \theta'), \tag{1}$$

where $\boldsymbol{J}_\theta(x)$ is the $p \times on$ Jacobian matrix of the model.[1] We will term Eq. 1 the *linearized NN* model and will consider additionally the first-order approximation as their own model, $g(x;\theta') \approx \boldsymbol{J}_\theta(x)^\top(\theta - \theta')$. Interestingly, this model can be seen as a version of the neural tangent kernel (Jacot et al., 2018) but at finite width, so we refer to it as the finite NTK. For regression, the two models (finite NTK and linearized NN) only differ in the choice of the mean function, which is somewhat unimportant in comparison to the covariance, so we will focus our experiments on the finite NTK.

As our primary goal is to account for functional uncertainty in new tasks, we will consider the linearized model in a Bayesian setting, assigning a prior to the linearized parameters, θ'. We assume that $\theta' \sim \mathcal{N}(0, \boldsymbol{I}_p)$ for simplicity as in Jacot et al. (2018); however, more highly structured Gaussian priors could be used. Under the Gaussian prior assumption, the linearized model and the gradient model are degenerate Gaussian processes (so named because in function space, the resulting kernel is degenerate — i.e. it has a finite number of features) (Rasmussen and Williams, 2008). Both models have the same kernel, $k(x_i, x_j) = \boldsymbol{J}_\theta(x_i)^\top \boldsymbol{J}_\theta(x_j)$, but different mean functions. In contrast to Jacot et al., we take the Taylor expansion around a trained neural network rather than the network at initialization.

[1] We will drop the dependency on x and squeeze o outputs.

3.1 Function Space Updates for Regression

Following Rasmussen and Williams (2008), the predictive posterior over the function f^* on new inputs x^* is:

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

where $\boldsymbol{J}_\theta^*$ is the Jacobian matrix on the new data points.[2] 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* $(\boldsymbol{J}_\theta\boldsymbol{J}_\theta^\top + \sigma^2\boldsymbol{I}_p)x = b$, with the Gram matrix, $\boldsymbol{J}_\theta\boldsymbol{J}_\theta^\top$. Naively, this operation requires $\mathcal{O}(p^3)$ time as the linear system is solved in parameter space (the weight space view). Fortunately, we can flip the computations in the dual function space by interpreting $\boldsymbol{J}_\theta$ as producing a degenerate Gaussian process with kernel matrix $\boldsymbol{J}_\theta^\top\boldsymbol{J}_\theta$. Using Woodbury's matrix identity, the posterior predictive distribution can be re-written as:

$$f^*|x^*, \mathcal{D} \sim \mathcal{N}\left(\boldsymbol{J}_\theta^{*T}\boldsymbol{J}_\theta(\boldsymbol{J}_\theta^\top\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)^{-1}\boldsymbol{y}, \right. \\ \left. \sigma^2\boldsymbol{J}_\theta^{*T}(\boldsymbol{I}_p - \boldsymbol{J}_\theta(\boldsymbol{J}_\theta^\top\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)^{-1}\boldsymbol{J}_\theta^\top)\boldsymbol{J}_\theta^*\right) \tag{3}$$

Again, inference only requires solving the linear system $(\boldsymbol{J}_\theta^\top\boldsymbol{J}_\theta + \sigma^2\boldsymbol{I}_n)x = b$, naively requiring $\mathcal{O}(n^3)$ time as the linear system is solved in function space (the function space view). Popular neural networks are generally over-parameterized, containing many more parameters than training samples; therefore, the function-space view will typically be faster. In the following section, we will consider computational considerations.

[2] Dropping the dependency on x^* for a superscript and using $\boldsymbol{y}$ to include both the response and the mean terms $\mu_\theta(x) = \boldsymbol{J}_\theta^\top\theta + f_\theta(x)$.

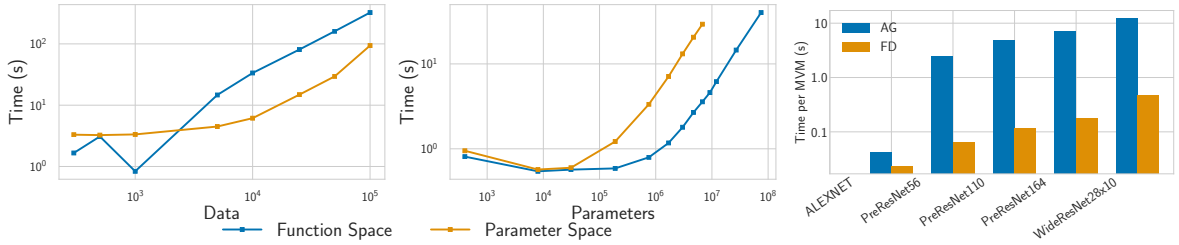


Figure 2: **Left:** Nearly linear scaling time of log probability calculation for an $\approx 800,000$ parameter MLP as a function of n. **Center:** Nearly linear scaling time of log probability calculation for $1,000$ data points as a function of model size. In both situations, both the function-space approach and the parameter space approach scale particularly well, reaching $100,000$ data points (and nearly 100 million parameters) unlike standard GP training procedures which are $\mathcal{O}(n^3)$. **Right:** Speedup of finite differences (FD) Fisher-vector products against autograd (AG) for AlexNet (Krizhevsky et al., 2012), PreResNets (He et al., 2016) of varying depth, and WideResNet (Huang et al., 2017) on CIFAR-10. Our method typically achieves a $30\times$ speedup over the standard implementation, demonstrating the practical utility of our approach to working with the Fisher matrix.

Extension to Non Gaussian Likelihoods: For non-Gaussian likelihoods, a degenerate Gaussian process on the latent functions is again produced, equivalent now to a Bayesian generalized linear model (GLM). Specifically, for multi-class classification the GLM contains the Jacobian within a categorical likelihood:

$$p_{\text{lin}}(y_i|x,\theta) = \text{Cat.}\left(y_i|\frac{\exp\{\boldsymbol{J}_\theta^\top\theta'\}}{\sum_{i=1}^{C}\exp\{\boldsymbol{J}_\theta^\top\theta'\}}\right), \qquad (4)$$

with appropriate reshaping to account for the *on* outputs of the model. Unfortunately, the posterior over θ' cannot be computed in closed-form. We consider both Laplace approximations and stochastic variational inference (Hoffman et al., 2013). See Appendix A for details.

3.2 Computational Speed-Ups

To consider large DNNs within our approach, we need to tackle two scalability issues: firstly the Jacobian matrix is not closed form for most NN models, and secondly, GP inference scales poorly. We resolve both issues simultaneously by using implicit Jacobian vector and vector Jacobian products as implemented in standard automatic differentiation software like Pytorch (Paszke et al., 2019). Implicit Jacobian vector products never form the full Jacobian matrix and use three backwards calls, one to compute $\boldsymbol{J}_\theta^\top v$ and two for $\boldsymbol{J}_\theta v$, rather than n. They are also extremely compatible with conjugate gradient (CG) approaches for GP inference (e.g. GPyTorch), as CG only requires matrix vector multiplications, resolving the poor scaling of GP inference. CG-enabled GP inference reduces the inference complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, while keeping the memory required constant by not forming dense matrices unnecessarily (Gardner et al., 2018). More implementation details are in Appendix B.

To demonstrate the efficiency of combining CG-enabled GP inference with implicit Jacobian vector products, we performed experiments with up to $100,000$ data points using five layer perceptrons with about $800,000$ parameters using only a single GPU. We show the timing in Figure 2, finding that predictive means and variances can be computed in under five minutes on even the largest models and datasets. We used similar architectures in Section 5.

For further computational improvements, we can perform inference in parameter space by leveraging the Jacobian's relationship to the Fisher information matrix, which is defined as. $\mathbb{F}(\theta) := \mathbb{E}_{p(x,y|\theta)}(\nabla_\theta \log p(y|x,\theta)\nabla_\theta \log p(y|x,\theta)^\top)$. For Gaussian likelihoods, the Fisher information matrix is proportional to the outer product of the Jacobian matrix with itself, e.g. $\boldsymbol{J}\boldsymbol{J}^\top \propto \mathbb{F}(\theta)$, across a dataset.

To exploit the relationhip, we derived a novel finite differences Fisher vector product (FVP) via the derivative of the Kullback-Leiber (KL) that gives matrix vector products with only one backwards call. The second order Taylor expansion of the KL divergence between two distributions, $p(y|\theta)$ and $p(y|\theta')$, with parameters θ and θ' is given by:

$$D_{\text{KL}}(p(y \mid \theta)||p(y \mid \theta')) = \frac{1}{2}(\theta-\theta')^\top\mathbb{F}(\theta)(\theta-\theta') + \mathcal{O}(\theta-\theta')^3.$$

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

$$\nabla_\theta D_{\text{KL}}(p(y \mid \theta)||p(y \mid \theta'))|_{\theta'=\theta+\epsilon v} = \epsilon\mathbb{F}(\theta)v + \mathcal{O}(\epsilon^2||v||). \qquad (5)$$

Therefore, to compute Fisher vector products, we merely need to compute a second forwards pass with θ', compute the KL divergence in likelihood space between the model with parameters θ and the model with

parameters θ', and then backpropagate with respect to θ'. The resulting gradient, up to error, is proportional to $\mathbb{F}(\theta)v$. To use these FVPs to increase the speed of our GP implementation, we use CG to solve systems of equations in parameter space (e.g. Eq. 2) replacing matrix vector multiplications of the form $\boldsymbol{J}_\theta \boldsymbol{J}_\theta^\top v$ with $a\mathbb{F}(\theta)v$, where a is the appropriate scaling constant. In Figure 2, we show the efficiency of finite NTK inference using these finite differences (FD) FVPs in comparison to kernel space inference. We also show the speedup against an exact FVP computed using autograd (AG), where the FD version is about 30× faster across a variety of modern architectures. Accuracy is not affected as relative error typically is on the order of 0.001. Exact implementation details are in Appendix C.

3.3 Fast Adaptation Modelling

There is a long history of multi-task Gaussian process models, see Álvarez et al. (2012) for further discussion. We adopt the simplest multi-task model, which considers the parameters of the trained neural network to be shared across all tasks, and recomputing the Jacobian for each new task (equivalent to sharing the kernel hyper-parameters across tasks).

The generative process can be described as follows: first, the dataset is drawn from a dataset distribution $\mathcal{D}_t = (x_t, y_t) \sim p(\mathcal{D})$, so that, for each individual task and dataset, $\mathcal{D}_t$, we have the likelihood $p(\boldsymbol{y}_t|x_t) = p(\boldsymbol{y}_t|f_{\theta_t}(x_t))$. In our setting, we have the linearized neural network, $f_t \approx \boldsymbol{J}_\theta(x_t)\theta_t' + \mu(x_t)$, where $\boldsymbol{J}_\theta(x) = (\nabla_\theta f(x))^\top \in \mathbb{R}^{p\times on}$, and $\mu(x,t)$ is the GP mean function as described previously, and θ_t' are the parameters of the (Bayesian) linearized model. The kernel for each task is given by $\boldsymbol{J}_\theta(x_t)^\top \boldsymbol{J}_\theta(x_t')$ as in Section 3.1. Note that the Jacobian computation depends only on the parameters θ of the pre-trained network. We can then represent the functions f_t (for other tasks) in a way that does not involve a new non-convex optimization. Thus, our procedure only requires a pretrained neural network on an initial task. When a new task is presented, the Jacobian matrix of data samples in the given task w.r.t. the pretrained parameters is used to derive the predictive distribution of the GP, which only requires solving a linear system. We present the overall algorihtm in Algorithm 1. In Appendix D we further describe the domain adaptation model.

4 INDUCTIVE BIASES OF LINEARIZED NEURAL NETWORKS

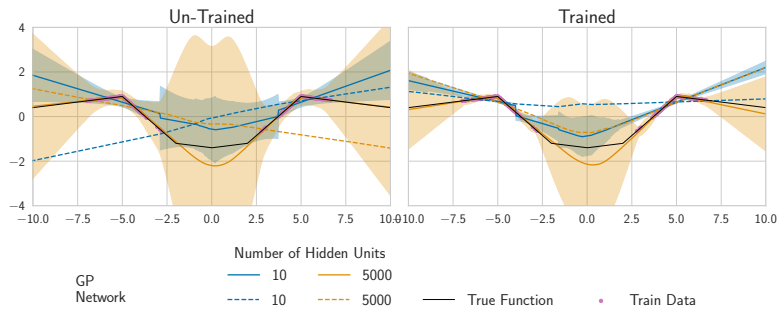
We first investigate the inductive biases of linearized neural networks on both regression and classification on CIFAR10, while showing that the Jacobians of similar tasks are related. Overall, we demonstrate i) that linearized neural networks retain the inductive biases of their nonlinear counterparts and ii) that the Jacobian matrix is a useful inductive bias for transfer learning.

Qualitative Regression Experiments: We begin with demonstrating qualitatively that linearized neural networks retain good inductive biases. See Appendix E for training and dataset details. First, in Figure 3a, we show the fit of the ReLU network trained on a synthetic dataset, alongside the posterior predictive mean and variance of the resulting Gaussian process. We do so for both trained and un-trained networks. Pleasingly, in both cases, having enough hidden units (in this case, 5000) gives reasonable predictions.

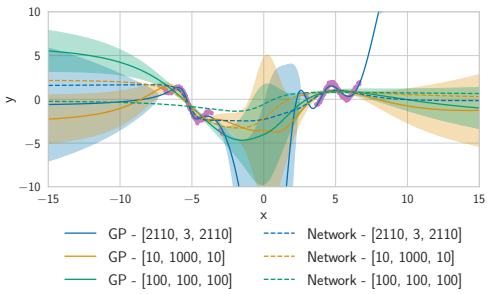
Second, in Figure 3b, we show the effect of different architectures on a different synthetic dataset. We can again qualitatively see that architectures that have good fits to the data (e.g. not over-fit) tend to also have reasonable predictive means and variances (e.g. the predictive mean fits the training data well and the predictive variance increases away from the data). We choose three separate architectures that all have about $21,000$ parameters, but differing numbers of hidden units at each layer to test the hypothesis that the performance of the linearized NN is tied to the over-parameterization effect rather than the architecture itself. Given that performance vastly differs even in this simple setting, we attribute the performance to the architecture itself, rather than over-parameterization.

Linearized PreResNets on CIFAR-10: We next tested the accuracy, test loglikelihood (NLL), and expected calibration error (ECE) (Guo et al., 2017) for PreResNets of varying depth on CIFAR10, displaying these results in Figure 4. Here, using the Jacobian matrices as features for classification leads to highly competitive performance for Bayesian generalized linear models — about 92% with our VI approximation for the standard 56 layer version. By comparison the top known kernel method (infinite un-trained neural network, but with a different architecture) on CIFAR-10 seems to be the result of Li et al. (2019), which is about 89%. Further results with MAP inference and Laplace approximations are in Appendix F. Intriguingly, we also find that the linearized networks here are more-overconfident (higher ECE) than the full model.

Transferability of Jacobian Features: Thus far, we have demonstrated that the Jacobian matrix is useful merely for a single task — e.g. a linearized model is a good inductive bias for both classification and regression. However, we next demonstrate that for a given architecture, the Jacobian matrix is similar across related tasks. We consider trained networks



(a) Trained and untrained ReLU networks with one hidden layer.



(b) Trained tanh networks of different widths.

Figure 3: Posterior means and confidence regions, $p(f^*|y)$, for MLPs of varying width and depth. Solid lines are the predictive means from the linearized neural network, with shading corresponding posterior confidence region; dashed lines are predictions from the full neural network. The observed data is shown as pink dots. **Left:** Linearization of un-trained single-layer ReLU networks of varying width. **Center:** Linearization for trained networks. Observable in both plots is the increase in the GP predictive variances as a function of width due to using standard initializations, while the trained networks have less predictive variance, especially around the training data (due to being fit on the data). **Right:** Three layer tanh architectures with different widths but similar numbers of parameters trained on a sinusoidal regression problem. Not only are the network's predictions qualitatively different, but their linearizations also are significantly different outside of the training data.

Algorithm 1 Domain Adaptation Procedure

Input: Data $(\boldsymbol{X}_1, \boldsymbol{y}_1)$, Initial parameters θ_0
Compute θ_{MLE} with data $(\boldsymbol{X}_1, \boldsymbol{y}_1)$.
Transfer inductive biases to GP:
$f \sim \mathcal{GP}(0, k = \mathbf{J}(\mathbf{x}_1; \theta_{MLE})^\top \mathbf{J}(\mathbf{x}_1; \theta_{MLE}))$
Compute Jacobian for task t: $\mathbf{J}_t = \mathbf{J}(\mathbf{x}_t; \theta_{MLE})$
Adapt f to domain t using Eq. 3 and $\mathbf{J}_t$:
$p(f_t^*|\mathcal{D}_t) = \int p(f^*|\theta_t')p(\theta_t'|\mathcal{D}_t)d\theta'$

on three different datasets — two halves of MNIST (MNIST1 and MNIST2) (LeCun et al., 1998), and FMNIST (Xiao et al., 2017). MNIST1 and MNIST2 are similar because they come from the same distribution, so we expect that classifiers trained on MNIST1 and MNIST2 will be similar due to being trained on similar data. By the same token, we expect these classifiers to have different representations than a classifier trained on FMNIST, which is images of clothing.

To test our hypothesis, we trained 25 LeNet-3 (LeCun et al., 1998) networks each on the three datasets, and then computed the full Jacobian matrix of each model on 5000 images from MNIST1[3]. Next, we computed the similarity of the matrices via their squared cosine similarity, e.g. $\text{sim}(A, B) = \frac{tr(A^\top BB^\top A)}{||AA^\top||_F||BB^\top||_F}$. We show histograms of the similarities across the 25 comparisons in Figure 5, finding that the Jacobians are vastly more similar between classifiers trained on MNIST1 and MNIST2 (shown in blue), while the classifiers trained on MNIST1 and MNIST2 have nearly zero similarity to those from FMNIST (shown in orange and green). This result supports our hypothesis, suggesting that the Jacobians of two models should be similar to each other if they were trained on similar enough tasks; full training details and the spectrum of the Jacobians for all of the models is shown in Appendix E.1.

[3]We fine-tuned the final layer of the FMNIST networks to remove distinctions in class labels and predictions.

5 TRANSFERABILITY EXPERIMENTS

Finally, we demonstrate that, on a variety of regression and classification tasks, our linearization framework enables transfer of the inductive biases from one task to others and enables good representations of uncertainty. Experimental details are in Appendix E, while further results are in Appendix F.

Sinusoids: In Figure 6, we generate 3 datasets and train a three-layer MLP with tanh activations to completion only on the first dataset (the orange points). We plot the fit from the finite NTK in blue (the mean and the confidence region), showing the validation (context) points in purple. We replicate the generative process of Kim et al. (2018), with various periods for the sinusoids. Here, we compare to a trained adaptive basis linear regression (ABLR) model (Perrone et al., 2018) is shown in grey and to transferred RBF Gaussian processes (yellow). ABLR requires many gradient steps to converge on the adaptation task as it continually re-trains the features of the last layer. Additionally, the

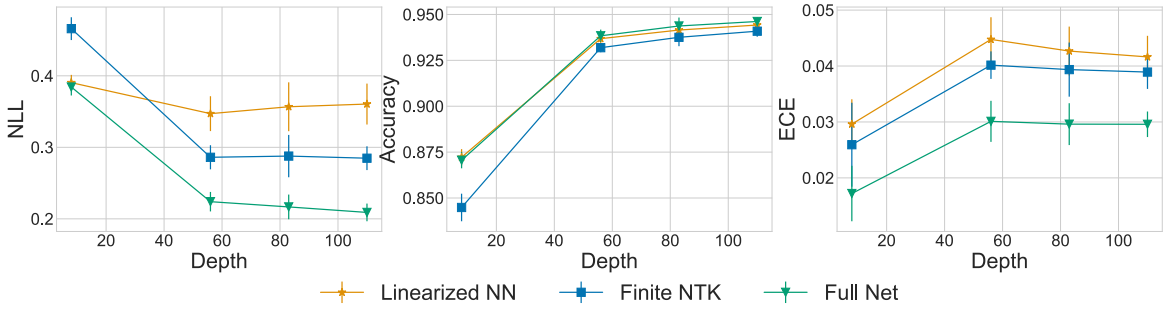


Figure 4: Test negative log-likelihood (NLL), accuracy, and expected calibration error (ECE) of PreResNets varying depth and their linearized counterparts as a function of depth on CIFAR-10, averaged over 10 seeds. For the linearization, we use a VI approach. Both the linearized NN (orange) and the finite NTK (blue) perform well in terms of accuracy, nearly comparable to the full network (green). However, they perform slightly worse relatively in terms of NLL and ECE, implying that they are more over-confident about their predictions.

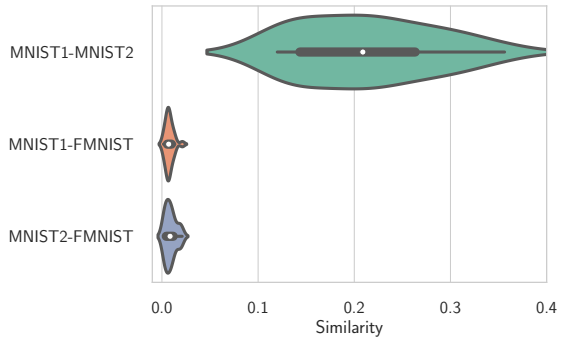


Figure 5: Cosine similarities of the Jacobians of trained LeNet-3s on the first half of MNIST (MNIST1). Models trained on FMNIST and evaluated on MNIST1 are less similar to models trained directly on MNIST1 than those trained on the second half of MNIST (MNIST2). The white dot in the middle is the average similarity across 25 random seeds, with the black lines showing the quartiles. The width of the violin for each comparison corresponds to the density estimate across these seeds, showcasing a population.

linearized model performs comparably to transferred hyper-parameter RBF GPs which are close to the gold standard on this task, because the functions are smooth. However, the linearized model gives reasonable posterior predictive *distributions* like the GP. In Appendix F.3, we additionally compare to transferring the deep kernel learning latent space, and adaptive deep kernel learning (Tossou et al., 2019), finding again that these alternative approaches underfit.

Malaria Prediction: Inspired by Balandat et al. (2020), we used data describing the infection rate of *Plasmodium falciparium* (a malaria causing parasite) drawn from the Malaria Global Atlas[4] in a domain adaptation task. We trained a heteroscedastic neural network with three layers on 2000 data points from the 2012 map, before testing on varying numbers of data points from the 2016 map. In Figure 7a, we show the test MSE for no re-training, the Jacobian as the kernel, and retraining the final layer versus the validation points given from the 2016 map. The performance of the Jacobian kernel as a transfer model improves as the number of data points given to it at validation time increases. For a fixed training budget (the same as the original training epochs), the re-trained last layer stagnates in performance. By contrast, re-training the last layer for $6x$ the training epochs will give comparable results to the finite NTK, but at a significantly increased computational expense; see Appendix E for further details.

Image Pose Prediction: We next convert the image rotation prediction task in Wilson et al. (2016) on the publicly available Olivetti Faces dataset.[5] We used a similar version of the LeNet-3 architecture for this task (LeCun et al., 1998). To construct this task, we used all 40 faces and rotated them such that $\theta \sim U(0, 30)$ where θ is the degree of rotation, cropping them to fit the 45×45 images, selected 20 faces to serve as the training set and the other 20 to serve as the adaptation set; the targets in all cases are the standardized degree of rotation. From the adaptation set, we randomly selected varying numbers of the data to serve as the adaptation points as we wish to have the methods quickly adapt to the new faces; we then evaluated on the MSE on the remainder of the adaptation points. We find, as displayed in Figure 7b, that the linearized NN performs best across the entirety of the number of adaptation points, although the gains

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

[5] `scikit-learn.org/0.19/datasets/olivetti_faces.html`.

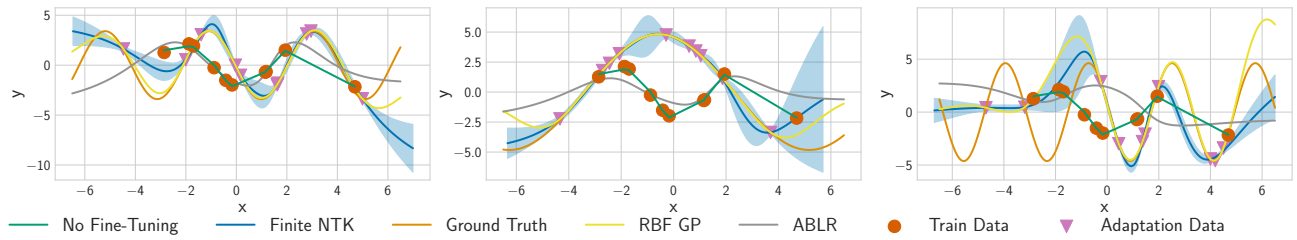
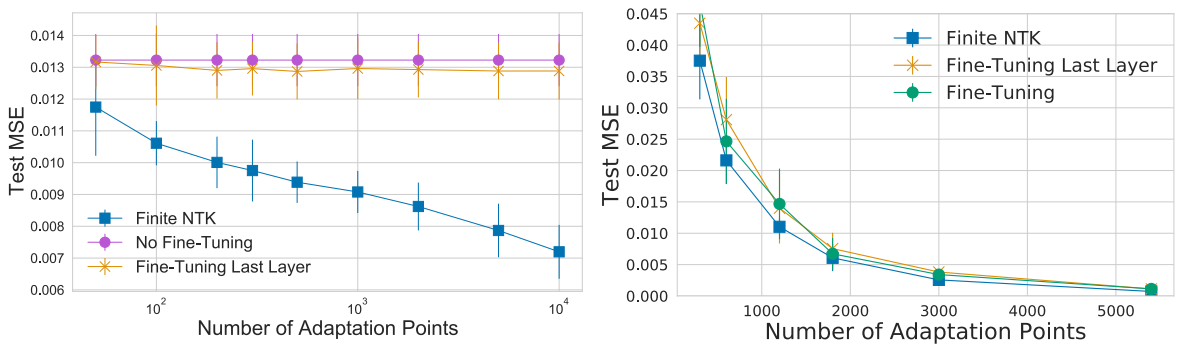


Figure 6: Posterior predictions on a few shot regression task using the finite NTK (blue). We performed training on the orange points, while adapting to the purple points (context), viewing each function as an independent draw from the GP described in Section 3.3. The comparisons here are RBF GPs fit on the source task with the hyper-parameters shared across tasks (yellow) and adaptive Bayesian linear regression (ABLR) (gray) (Perrone et al., 2018). Our finite NTK approach improves upon the RBF kernel, and significantly outperforms ABLR.



(a) Malaria dataset

(b) Olivetti faces dataset

Figure 7: **(a)** 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 and only has marginally better performance than no-retraining at all. **(b)** Error on held out set on the Olivetti fast adaptation task. In this setting, all methods improve with more adaptation points, and the NTK has the biggest advantage when there is a small amount of data for transfer.

Table 1: Test Accuracy on CIFAR10 for linearizing different layers of the BiGAN encoder. $*$ denotes numbers reproduced from Table 1 of Mu et al. (2020), which are included as baselines. Using all of the layers produces slightly better results than using only the top layers. All linearization settings are best compared to the activation baseline of 62.87% which consists of training a classifier on top of the feature extractor. Fine-tuning all of the layers is also better overall.

Layers	Top Layer	Top 2 Layers	Top 3 Layers	All Layers	All Layers (VI)
Fine-tuning	71.78*	73.18*	74.30*	*77.25*	-
Finite NTK	70.28*	71.08*	70.64*	***72.65***	*71.89*

are small when many points are considered.

Transferring Unsupervised Models: We finally test the capability of the linearized model to work as a supervised model when the full model is trained on an un-supervised task, following the experiments of Mu et al. (2020), who used BiGANs (Donahue et al., 2016) trained on CIFAR10 and linearized only the top convolutional layers. We replicate their experimental setup, but consider inclusion of all of the features as well, by adding the Jacobian of the base feature extractor network into the model. Like Mu et al. (2020), we used a fixed projection from the output dimensionality of the network down to 10 output dimensions so that we can perform classification with it. The results are shown in Table 1, where we see that in this problem the top layers seem to be at least as useful as the full gradient projections. Here, the all layers column refers to solely MAP trained models for direct comparison with Mu et al. (2020), while all layers (VI) refers to our finite NTK model with variational inference. We hypothesize that the underperformance of the finite NTK is due to using approximate inference.

6 CONCLUSION

We have shown how it is possible to build scalable kernel methods with the inductive biases of neural networks, but with closed-form training and no local optima, through a linearization. We combine these kernels with Gaussian processes for a principled representation of uncertainty. We show how to make inference scalable by developing efficient techniques for Jacobian vector products and conjugate gradients. The techniques we propose are in fact generally applicable, and in the future could be leveraged in a wide range of other settings where one wishes to perform efficient operations with a Jacobian or Fisher matrix. In this paper, we apply these ideas primarily for fast, convenient, and analytic transfer learning with neural networks. Exciting future work could further develop this direction for meta-learning, where one wishes to efficiently capture the transfer of knowledge across several tasks and uncertainty plays an important role.

Acknowledgements

WJM, AGW are supported by an Amazon Research Award, NSF I-DISRE 193471, NIH R01 DA048764-01A1, NSF IIS-1910266, and NSF 1922658 NRT-HDR: FUTURE Foundations, Translation, and Responsibility for Data Science. WJM was additionally supported by an NSF Graduate Research Fellowship under Grant No. DGE-1839302. We would like to thank Jacob Gardner, Alex Wang, Marc Finzi, and Tim Rudner for helpful discussions, Jordan Massiah for help preparing the codebase, and the NYU Writing Center for writing guidance.

References

Achille, A., Lam, M., Tewari, R., Ravichandran, A., Maji, S., Fowlkes, C., Soatto, S., and Perona, P. (2019). Task2Vec: Task Embedding for Meta-Learning. In *ICCV*.

Álvarez, M. A., Rosasco, L., Lawrence, N. D., et al. (2012). Kernels for vector-valued functions: A review. *Foundations and Trends® in Machine Learning*, 4(3):195–266.

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

Balandat, M., Karrer, B., Jiang, D. R., Daulton, S., Letham, B., Wilson, A. G., and Bakshy, E. (2020). BoTorch: Programmable Bayesian Optimization in PyTorch. In *Advances in Neural Information Processing Systems*.

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

Donahue, J., Krähenbühl, P., and Darrell, T. (2016). Adversarial feature learning. In *International Conference on Learning Representations (ICLR)*.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *International Conference on Machine Learning*.

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. (2018). GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems*.

Golub, G. H. and Pereyra, V. (1973). The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems Whose Variables Separate. *SIAM Journal on Numerical Analysis*, 10(2):413–432.

Guo, C., Pleiss, G., Sun, Y., and Weinberger, K. Q. (2017). On Calibration of Modern Neural Networks. In *International Conference on Machine Learning*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *CVPR*. arXiv: 1512.03385.

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely Connected Convolutional Networks. In *CVPR*.

Jaakkola, T. and Haussler, D. (1998). Exploiting Generative Models in Discriminative Classifiers. In *Advances in Neural Information Processing Systems*.

Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*.

Kim, T., Yoon, J., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). Bayesian Model-Agnostic Meta-Learning. In *Advances in Neural Information Processing Systems*.

Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. page 60.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., and others (1998). Gradient-based learning applied to

document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lee, J., Xiao, L., Schoenholz, S. S., Bahri, Y., Novak, R., Sohl-Dickstein, J., and Pennington, J. (2019). Wide Neural Networks of Any Depth Evolve as Linear Models Under Gradient Descent. In *Advances in Neural Information Processing Systems*.

Li, Y., Yosinski, J., Clune, J., Lipson, H., and Hopcroft, J. (2015). 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.

Li, Z., Wang, R., Yu, D., Du, S. S., Hu, W., Salakhutdinov, R., and Arora, S. (2019). Enhanced Convolutional Neural Tangent Kernels. *arXiv:1911.00809 [cs, stat]*. arXiv: 1911.00809.

MacKay, D. J. C. (2003). *Information theory, inference, and learning algorithms*. Cambridge University Press, Cambridge, UK ; New York.

Montavon, G., Orr, G. B., and Müller, K., editors (2012). *Neural Networks: Tricks of the Trade - Second Edition*, volume 7700 of *Lecture Notes in Computer Science*. Springer.

Mu, F., Liang, Y., and Li, Y. (2020). Gradients as Features for Deep Representation Learning. In *International Conference on Learning Representations*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037.

Patacchiola, M., Turner, J., Crowley, E. J., O'Boyle, M., and Storkey, A. (2019). Deep kernel transfer in gaussian processes for few-shot learning. In *Advances in Neural Information Processing Systems*.

Perrone, V., Jenatton, R., Seeger, M. W., and Archambeau, C. (2018). Scalable Hyperparameter Transfer Learning. In *Advances in Neural Information Processing Systems*, page 11.

Pleiss, G., Gardner, J. R., Weinberger, K. Q., and Wilson, A. G. (2018). Constant-Time Predictive Distributions for Gaussian Processes. In *Artificial Intelligence and Statistics*.

Rasmussen, C. E. and Williams, C. K. I. (2008). *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass., 3. print edition.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. SIAM, Philadelphia, Pa, 2. ed edition.

Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. pages 806–813.

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

Tosi, A., Hauberg, S., Vellido, A., and Lawrence, N. D. (2014). Metrics for Probabilistic Geometries. In *Uncertainty in Artificial Intelligence*.

Tossou, P., Dura, B., Laviolette, F., Marchand, M., and Lacoste, A. (2019). Adaptive deep kernel learning.

Wilson, A. G., Hu, Z., Salakhutdinov, R., and Xing, E. P. (2016). Deep Kernel Learning. In *Artificial Intelligence and Statistics*. arXiv: 1511.02222.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

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

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*.

Zhai, S., Talbott, W., Guestrin, C., and Susskind, J. M. (2019). Adversarial Fisher Vectors for Unsupervised Representation Learning. In *Advances in Neural Information Processing Systems*.

Zinkevich, M. A., Davies, A., and Schuurmans, D. (2017). Holographic feature representations of deep networks. In *Uncertainty in Artificial Intelligence*.

Supplementary Materials for Fast Adaptation with Linearized Neural Networks

A APPROXIMATE INFERENCE

In this appendix, we discuss how we performed approximate inference in parameter space using the Jacobians as features. In general, function space approaches either require computation of the diagonal of $\boldsymbol{J}_\theta^\top \boldsymbol{J}_\theta$ (for variational versions) or many more solves (for Laplace approximations) which would slow down inference considerably in our framework.

Using the finite NTK (or even the Taylor expansion perspective), we can replace the network, $f_\theta(\cdot)$ with the linearized model in the likelihood. For example, for multi-class classification, $p(y_i|\boldsymbol{f}) = \text{Categorical}(y_i|\frac{\exp\{\boldsymbol{f}_i\}}{\sum_{j=1}^C \exp\{\boldsymbol{f}_j\}})$, so that the linearized model is then given as

$$p_{\text{lin}}(y_i|x,\theta) = \text{Cat.}\left(y_i|\frac{\exp\{\boldsymbol{J}_\theta(x)^\top\theta'\}}{\sum_{i=1}^C \exp\{\boldsymbol{J}_\theta(x)^\top\theta'\}}\right). \tag{A.1}$$

For classification models, we will also consider the full Taylor expansion in Eq. 1 as a sanity check.

Stochastic Variational Inference (SVI) We tried stochastic variational inference in parameter space by using the ELBO and assuming a factorized Gaussian posterior. Assuming the linearized loss in Equation A.1, the mini-batched ELBO (Hoffman et al., 2013) is simply

$$\log p(\boldsymbol{y}|\boldsymbol{X}) \geq \frac{N}{B}\sum_{i=1}^{B} \log p(y_i|x_i,\theta) - \text{KL}(q(\theta|\mu,\text{softplus}(v))||\mathcal{N}(\theta|0,\sigma^2 I)).$$

We initialized $\mu = 0$ and $v = -5$ before training for ten more epochs. If we include the predictions of the NN as well, we add in another mean term into the likelihood.

Laplace Approximation First, we can now utilize the Fisher information as a scalable Laplace approximation, so that (assuming a Gaussian prior with variance σ^2) the posterior is approximately $\mathcal{N}(\theta_i|\theta, (\mathbb{F}(\theta) + \sigma^2 I)^{-1})$. Approximating the posterior in this manner will give rise to a degenerate version of the multi-class Laplace approximation derived in Rasmussen and Williams (Chapter 3 2008). To compute the approximation, we trained for 10 further epochs to get the MAP estimate. At test time, we assumed conditional independence of the test points, computing the inverse of the Fisher information using the test batch via the approximate Fisher vector products that we derived previously upscaling this matrix by a term of N/B.

Under both linearizations, note that if θ is used as both the parameters for the Jacobian and the parameters in the model, then we can additionally write down the Fisher information matrix of this model and see that the only difference is in the function value of the inner portion of the loss function, which under the linearization assumption is assumed to be close to the true model, suggesting that we can use the fast Fisher vector products plus a Lanczos decomposition to sample from the Gaussian posterior. We use a single sample at test time.

B SCALABLE EXACT GAUSSIAN PROCESSES

The chief bottleneck in Gaussian process computation is the cubic scaling as a number of data points due to having to invert the kernel matrix for posterior predictions, e.g. finding $K^{-1}z$. However, Gardner et al. (2018) provides a numerical linear algebra way around this issue via iterative methods; here, we give a more detailed explanation of their approach in the context of our use case. To reduce the time complexity of solving linear sytems, we can use either the Lanczos algorithm or conjugate gradients (Saad, 2003, Chapters 6-10); both of

which use Krylov subspaces of a symmetric matrix $A \in \mathbb{R}^{p\times p}$. The Krylov subspaces method takes as input a starting vector b, before computing successive matrix vector products:

$$\mathcal{K}(A, b) = \text{span}\left[b, Ab, A^2b, A^3b, \cdots A^m b\right].$$

Orthogonalization and normalization (e.g. Gram-Schmidt) are then used to produce a basis matrix, Q, while the resulting coefficients from the orthogonalization can be stored into a matrix T_m, producing the recursion $AQ_m = Q_{m+1}T_m$. As A is symmetric, T_m is tri-diagonal, producing the decomposition: $A \approx Q_m T_m Q_m^T$. Taking $m = p$, produces the decomposition, $A = QTQ^T$. Note that solves can be produced from Lanczos as $A^{-1}b \approx ||b||Q_m T_m^{-1} e_1$, while the conjugate gradients method directly returns a closely related solution. Finally, we can see that only a single (or possibly a constant number) of matrix vector multiplies with A, reducing the complexity to $\mathcal{O}(p^2 m)$. Conjugate gradients converges exactly when $r = p$, but that we can speed up the convergence rate by using a pre-conditioner, P, where $P \approx A^{-1}$. We use the standard pivoted Cholesky preconditioner (Gardner et al., 2018).

Finally, for predictive variance computations, we must store $(\boldsymbol{J}_\theta \boldsymbol{J}_\theta^T + \sigma^2 I)^{-1}$ or $(\boldsymbol{J}_\theta^T \boldsymbol{J}_\theta + \sigma^2 I)^{-1}$. However, we can approximate this by an approximate eigen-decomposition using the Lanczos decomposition (e.g. Pleiss et al., 2018), writing $(\boldsymbol{J}_\theta \boldsymbol{J}_\theta^T + \sigma^2 I)^{-1} \approx RR^T$, where $R = QT^{-1/2}$. This requires only $\mathcal{O}(pr)$ storage and again can be computed in quadratic time as T is tri-diagonal.

C FURTHER DISCUSSION OF THE FISHER INFORMATION MATRIX

C.1 Relating the Fisher Information Matrix to the NTK

Regression: An interesting example is given by homoscedastic regression with a Gaussian likelihood, where $y \sim \mathcal{N}(f(x), \sigma^2 \boldsymbol{I})$. There, 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 finite NTK as we use it then have the same eigenvalues (up to a constant factor of n) as they are similar matrices. Similar connections between the Jacobian and the Fisher information matrix are used by Tosi et al. (2014), and the connection seems to originate in the generalized Gauss-Newton decomposition of Golub and Pereyra (1973). Yang (2019) first noted the connection in the context of the neural tangent kernel at infinite width.

General Losses: For general losses, it is still possible to relate the Fisher information matrix to either the Gram matrix or the NTK.

$$\begin{aligned}
\mathbb{F}(\theta) :&= \mathbb{E}(\nabla_\theta \log p(y|x,\theta) \nabla_\theta \log p(y|x,\theta)^\top) \\
&= \mathbb{E}(\nabla_\theta f(x;\theta) \nabla_f \log p(y|f) \nabla_f \log p(y|f)^\top \nabla_\theta f(x;\theta)^\top) \\
&= \mathbb{E}_{p(x)}(\nabla_\theta f(x;\theta) \mathbb{E}_{p(y|f)}(\nabla_f \log p(y|f) \nabla_f \log p(y|f)^\top) \nabla_\theta f(x;\theta)^\top) \\
&\approx \boldsymbol{J}_\theta \boldsymbol{H}_\theta \boldsymbol{J}_\theta,
\end{aligned}$$

with the approximation coming as the Jacobian is computed over the observed dataset rather than the true data distribution — exact if we use the empirical Fisher information, taking the empirical data distribution to be the tue data distribution, $p(x)$. $\boldsymbol{H}_\theta$ is the matrix of the gradient covariance with respect to the inputted function $\boldsymbol{H}_\theta := \mathbb{E}_{p(y|f)}(\nabla_f \log p(y|f) \nabla_f \log p(y|f)^\top)$. Note that $\boldsymbol{H}_\theta(x)$ is block-diagonal and positive semi-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, $\mathbb{F}(\theta) \approx \boldsymbol{J}_\theta \boldsymbol{H}_\theta \boldsymbol{J}_\theta^T = S\Lambda S^T$. Future work is necessary to be able to efficiently exploit the decomposition beyond the simple parameter space Laplace approximation we used.

C.2 Fast Fisher Vector Products via Directional Derivatives

To implement the fast Fisher vector products as described in Section 3.2, we need to know the closed form of the KL divergence of the likelihood distribution to itself (e.g. the KL between two Gaussians for regression). For many probability distributions used in machine learning, the KL divergence is closed form and often pre-implemented (for PyTorch in `torch.distributions`). For fixed homoscedastic Gaussian noise, the KL divergence is

$$D_{\text{KL}}(p(y \mid \theta) || p(y \mid \theta')) = \frac{(f_\theta - f_{\theta'})^2}{2\sigma^2}.$$

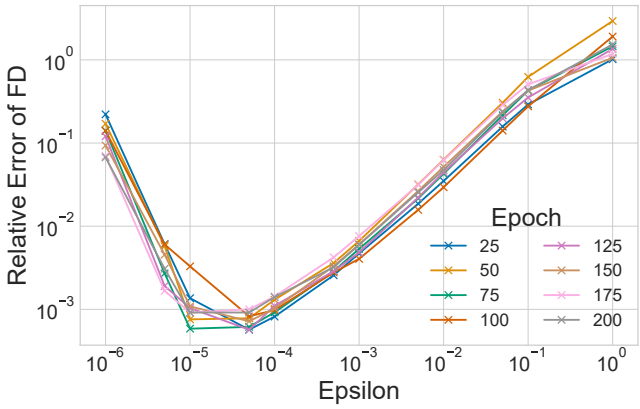


Figure A.1: Accuracy of Fisher-vector products as a function of tuning parameter ϵ for finite differences (FD, our approximate fast Fisher-vector product) versus autograd (AG), which is exact, across training of a PreResNet56 measured every 25 epochs. The finite differences approximation is accurate to a relative error of less than 1% for most choices of ϵ.

For multi-class classification with categorical likelihoods, the KL divergence reduces to a summation over all classes, giving

$$D_{\mathrm{KL}}(p(y \mid \theta)||p(y \mid \theta')) = \sum_{c=1}^{C} \mathrm{detach}(p(y_c \mid \theta)) \log\left(\frac{p(y_c \mid \theta)}{p(y_c \mid \theta')}\right),$$

where detach(.), refers to an operation that will not play a role in the computation graph.

Finally, we show the approximation error between the directional derivative as defined in Eq. 5 and the exact $F_i v$ computed using the standard second order autograd; this is shown in Figure A.1. We used a modern neural network architecture, PreResNet56, on the benchmark CIFAR10 dataset and computed the relative error as a function of ϵ:

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

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. Furthermore the approximations fail gracefully, only decaying to large error when ϵ is too small due to numerical precision issues or when ϵ is too large and the finite diferences approximation is not accurate. For regression, we observed similar qualitative results with higher stability.

D FAST ADAPTATION MODEL

We define a deep neural network, f, as taking an input, $x \in \mathbb{R}^d$, and mapping it to an output, $y \in \mathbb{R}^o$, with parameters $\theta \in \mathbb{R}^p$, letting $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$. We will additionally describe the full dataset as $\boldsymbol{X} = \{x_i\}_{i=1}^n$ and $\boldsymbol{y} = \{y_i\}_{i=1}^n$. For the purposes of fast adaptation, we consider an *initial* task $t = 0$ used to obtain parameters $\theta \triangleq \theta_{MLE}$ which are then re-used for learning the mappings f_t for the rest of the tasks $t = 1, \ldots, T$. In this way, we rapidly adapt f_0 to f_t. That is, we pre-train a network on the first task given the first set of data $(\boldsymbol{X}_0, \boldsymbol{y}_0)$ using any optimization procedure (e.g. SGD or Adam) to get out a set of parameters θ_{MLE}.

In the multi-task learning scenario, we have multiple related tasks $t = 1, \ldots, T$, where a task is defined as learning a neural network f_t, which depends on parameters θ_t, given the corresponding dataset $\mathcal{D}_t = \{\boldsymbol{X}_t, \boldsymbol{y}_t\}$. So for every subsequent task, we lazily compute the Jacobian for task t using the new training inputs (or context points), $\boldsymbol{x}_t$ and then compute the predictive mean cache of the Gaussian process (e.g. the terms dependent on the training data in Equations 2 and 3). For function space inference (Equation 3), following Gardner et al. (2018) and Pleiss et al. (2018), we compute $m = \boldsymbol{J}_\theta(\boldsymbol{J}_\theta^\top \boldsymbol{J}_\theta + \sigma^2 \boldsymbol{I}_n)^{-1}\boldsymbol{y}$ and $RR^\top \approx \boldsymbol{J}_\theta(\boldsymbol{J}_\theta^\top \boldsymbol{J}_\theta + \sigma^2 \boldsymbol{I}_n)^{-1}\boldsymbol{J}_\theta$. On the task's test set, we then only have to compute $\mu(x^*) = \boldsymbol{J}_\theta^{*T} m$ (ignoring the GP mean function) and $\sigma^2(x^*) = \boldsymbol{J}_\theta^{*T}\boldsymbol{J}_\theta^* - \boldsymbol{J}_\theta^{*T} RR^\top \boldsymbol{J}_\theta^*$. Parameter space inference uses the same mechanism, pre-computing $m = (\boldsymbol{J}_\theta \boldsymbol{J}_\theta^\top + \sigma^2 \boldsymbol{I}_p)^{-1}\boldsymbol{J}_\theta \boldsymbol{y}$ and $BB^\top \approx (\boldsymbol{J}_\theta \boldsymbol{J}_\theta^\top + \sigma^2 \boldsymbol{I}_p)^{-1}$, but the predictive variance becomes $\sigma^2(x^*) = \sigma^2 \boldsymbol{J}_\theta^{*\top} BB^\top \boldsymbol{J}_\theta^*$. In both cases, only a single Jacobian vector product on the test inputs $\boldsymbol{x}^*$ is required to get a test predictive mean and variance.

E EXPERIMENTAL DETAILS

E.1 Similarity of Jacobian across Tasks

We used the LeNet implementation from `https://github.com/activatedgeek/LeNet-5` and followed their training procedure, resizing the images to be 32×32, training for 20 epochs with a learning rate of $2e-3$ using

Adam and a batch size of 256.. For MNIST1 and MNIST2, we fixed the seed and then split the dataset using the first 30000 images. For FashionMNIST, we re-trained the final linear layer (so as to not disturb the internal representations) for 1000 steps using Adam with a learning rate of 0.01.

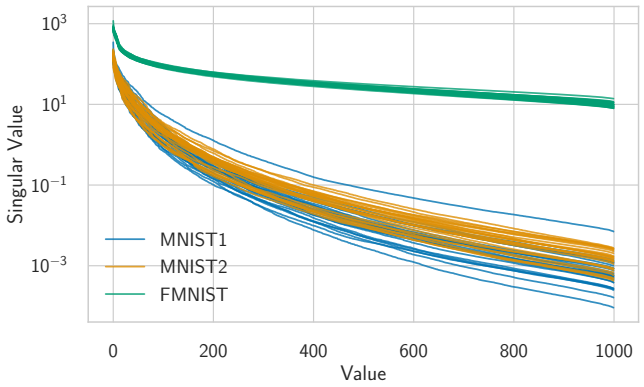


Figure A.2: Singular values of the Jacobian on 5000 images from the first half of MNIST (MNIST1) for25 models trained each on MNIST1, as well the second half of MNIST (MNIST2) and on Fashion MNIST (FMNIST). The singular values decay much more rapidly (and in a similar fashion) for models trained on MNIST1 as well as MNIST2, implying that their corresponding finite NTK kernels will have similar properties.

In Figure A.2, we show 1000 singular values[6] of the Jacobian for data coming from the MNIST1 task (first half of MNIST) from 25 models each that were pre-trained on MNIST1, the second half of MNIST (MNIST2), or Fashion MNIST. To ensure fair comparison across tasks, we re-trained the classifier networ of Fashion MNIST until it had small training error on MNIST1. Surprisingly, the Jacobians of the models trained on MNIST2 have similar singular values to the Jacobians of models trained on MNIST1, despite being trained on different data.

As our finite NTK kernel is given by $\boldsymbol{J}_\theta \boldsymbol{J}_\theta^\top$, the squared singular values of $\boldsymbol{J}_\theta$ are the eigenvalues of the kernel. The eigenvalues of kernel matrices help to determine the properties and inductive biases of the kernel, so that we expect two kernels with similar eigenvalues (and a similar decay rate for them) to have similar properties.

E.2 Regression and Classification Details

For all regression experiments, unless otherwise specified, we set σ^2 to be the training mean squared error on the original source task. We could alternatively have marginalized or optimized it.

Qualitative Regression Experiments For 3a, the true regression function is $y = 0.1x^2 + |x| + \epsilon_i$, with $\epsilon_i \sim \mathcal{N}(0, 0.9^2)$, and the 300 training data points are drawn from $\mathcal{N}(\pm 5, 0.8^2)$. The neural networks were trained with 250 epochs of SGD with momentum with learning rate $1e-4$ and momentum 0.9. For Figure 3b, we next fit three different architectures with tanh activations — $[2110, 3, 2110]$ (21104 parameters), $[10, 1000, 10]$ (21041 parameters) and $[100, 100, 100]$ (20501 parameters) — on 150 data points with the same x generation process as before, but now $y = \sin(3x) + 1.5\sin(0.5x) + 0.8|x| - 4 + 0.1 * \mathcal{N}(0, 0.1^2)$.

Sinusoidal Regression The generative process matches the generative process of Kim et al. (2018), 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 $\epsilon \sim \mathcal{N}(0, 0.01A)$. We follow the setup of Kim et al. (2018), and use neural networks with two hidden layers and 40 hidden units and tanh activations (or ReLU for Figure A.5). 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 compute the predictive variances using either function or parameter space. For Figure A.6, we perform parameter space inference with our novel implementation of Fisher vector products ($\epsilon = 1e-4$.).

Malaria 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 (using the dataset preprocessing from Balandat et al. (2020)), training with a heteroscedastic loss, so that the likelihood model was $\mathcal{N}(y|\mu_\theta(x), 1e-5+\text{softplus}(\sigma_\theta(x)))$. 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? We used inference in function space here as it was slightly more numerically stable (e.g. a smaller conjugate gradients residual norm) for small amounts of data.

[6] Using `torch.svd_lowrank`.

Olivetti For the olivetti faces dataset, we trained neural networks similar to the LeNet-3 architecture used in (Wilson et al., 2016); that is, after enforcing that the image was 45×45, a convolutional layer with kernel 5×5 and 20 channels, another with the same kernel size and 50 channels, then a max pooling layer, and linear layers of $3200 - 500 - 2$ with ReLU activations. We again trained with the same heteroscedastic loss, but here used Adam with a learning rate of $1e-3$, a batch size of 32, and for 150 epochs. In this setting, we found that re-training the *entire* neural network seemed practical due to the small size of the dataset, so we included it in the experiments. Again, we used inference in function space here as it was slightly more numerically stable, except for the largest data point size, where we used inference in parameter space for numerical stability.

Linearized PreResNets To train the PreResNets, we followed the training procedure and model definitions originally from `https://github.com/kuangliu/pytorch-cifar`, except that we varied depth, used random cropping and horizontal flips, training with SGD with momentum for 200 epochs with an initial learning rate of 0.1 and weight decay of $5e-4$. For the linearization experiments, we compared to accuracies at the end of training, training the approximate inference models with an initial learning rate of $1e-3$ and using Adam for 10 epochs. The prior was again set to $\mathcal{N}(0, 1)$; here, we found that the prior variance was important, as high variances performed poorly. We followed the same procedure for transferring the models to STL10, described in Appendix F.2. For fine-tuning those networks, we used the same learning rate, optimizer, and training time.

F FURTHER EXPERIMENTS

F.1 PreResNets: Linearization on CIFAR10

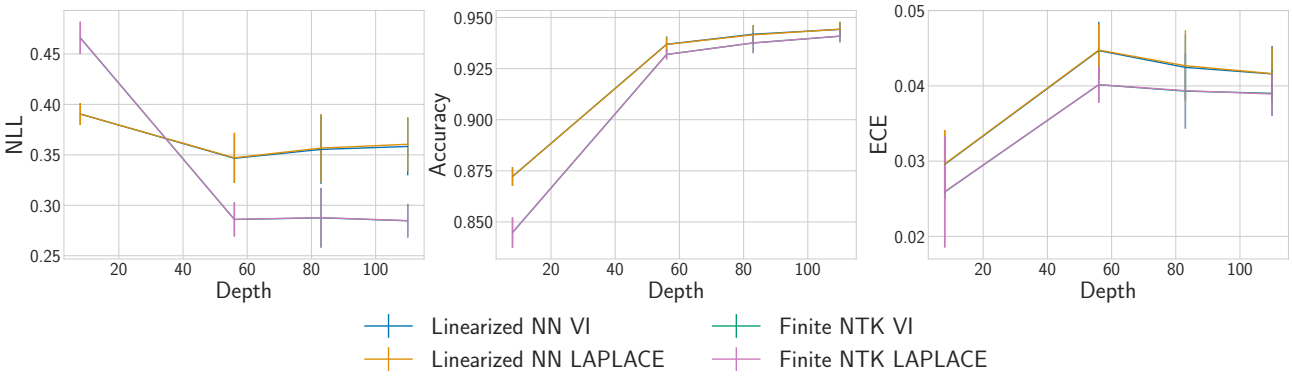


Figure A.3: Test Accuracy of PreResNets averaged over 10 random seeds. Both VI and Laplace are included here to see if the approximate inference method mattered much for either the linearized NN or the finite NTK. They perform similarly with some small differences.

In Figure A.3, we show the differences between using variational inference and the Laplace approximation for linearized NNs and the finite NTK on NLL, accuracy, and the expected classifcation error (ECE). Somewhwat surprisingly, we found that there was not that much difference; however, we attribute this to using the mean parameter for SVI at test time, and the fact that the Laplace approximation produces essentially the same mean parameter due to training with SGD before using a single sample at test time. MAP training produced similar results; except that it was faster than the Laplace approximation at test time.

F.2 PreResNets: CIFAR-10 to STL10

Finally, we perform domain adaptation by training PreResNets on CIFAR-10 Krizhevsky (2009), and transfer classification to STL10 Netzer et al. (2011). To do so, we downsample STL10 to utilize 32 x 32 images.[7] We show accuracy as a function of depth for linearized preactivation ResNets, comparing to fine-tuning and no fine-tuning, in Figure A.4. Here, approximate inference with Laplace approximations converged but again the VI results for the linearization did not. Interestingly, the deeper Laplace approximations that incorporated the function

[7]Nine of the ten CIFAR10 classes are represented in STL10. We map these classes to each other and the non-matching ones as well.

itself had higher variance, possibly due to over-fitting, while the Laplace approximation using the Jacobian as the features nearly matched the performance of the non-fine tuned model on this task. Specifically, we find that utilizing the full Taylor expansion and approximate inference is somewhat better than no adaptation, while using the Laplace approximation without the prediction of the model is a competitive baseline for a linear model.

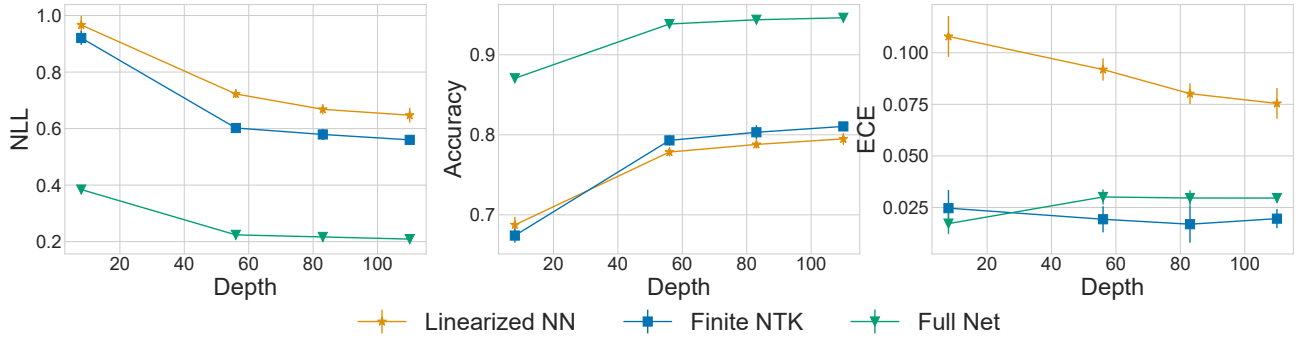


Figure A.4: Test Accuracy of PreResNets averaged over 10 random seeds with fine-tuning (green), the linearized NN (blue) and the finite NTK (orange) after training on CIFAR-10 and transferring to STL10. Linearization using solely the Jacobian as the features (yellow) is competitive with fine-tuning the final layer and no fine-tuning.

F.3 Further Sinusoids Plots

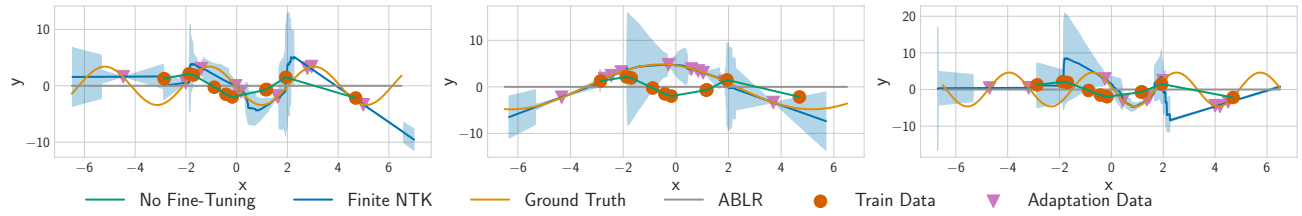


Figure A.5: Posterior predictions on a few shot regression task produced with the NTK as the kernel using a network with ReLU activations. Here, the ReLU network is a poor inductive bias for this task, as is reflected in the jagged predictive variances and in the ABLR comparison which is completely flat.

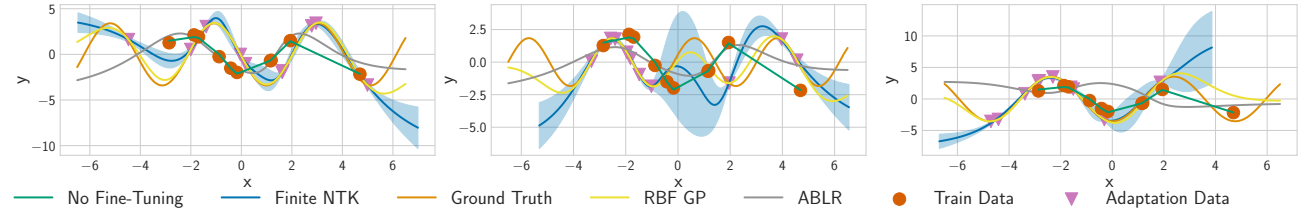


Figure A.6: Posterior predictions on a few shot regression task produced with the NTK as the kernel using Fisher vector products instead of Jacobian vector products. These results are nearly identical to the results with Jacobian vector products described in the main text.

In Figure A.5, we show transferring with ReLU activations and the accompanying sharp variance predictions (with the same width as described in Appendix E). In Figure A.6, we show the same networks as in the main text but with the inference performed using Fisher vector products (e.g. in parameter space) instead of using Jacobian vector products. The results are essentially identical to those displayed in the main text, as expected.

Finally, in Figure A.7, we compare to adaptive deep kernel learning (ADKL) (Tossou et al., 2019) and a deep kernel learning implementation that transfers the learned representation into a new Gaussian process model, with the same hyper parameters. The deep kernel learning implementation is a two task version of Patacchiola et al. (2019). Both ADKL and the transferred DKL significantly underfit on the second task for sinusoids, which

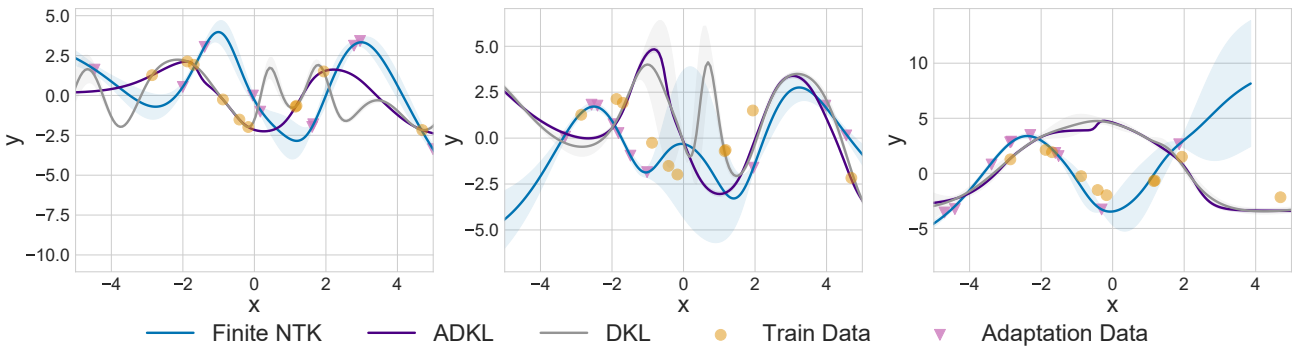


Figure A.7: Comparison to ADKL (Tossou et al., 2019) and transferred DKL, a two task version of Patacchiola et al. (2019) on the sinusoids problem. Both ADKL and transferred DKL underfit.

indicates their unsuitability for few shot transfer. Both methods are designed for meta-learning over many tasks instead.