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 & 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 & 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

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

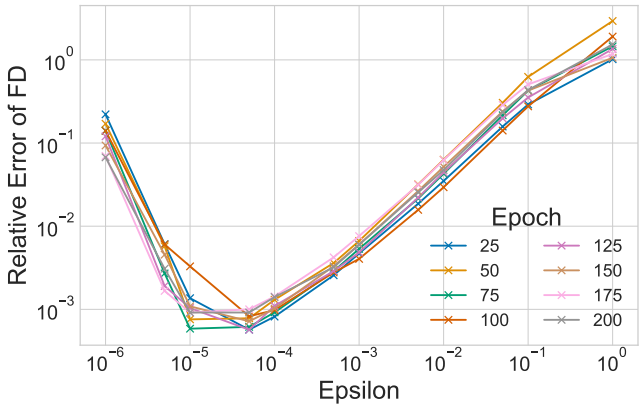

Figure A.1: Accuracy of Fisher-vector products as a function of tuning parameter ϵ for finite differences (FD, our approximate fast Fisher-vector product) versus autograd (AG), which is exact, across training of a PreResNet56 measured every 25 epochs. The finite differences approximation is accurate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

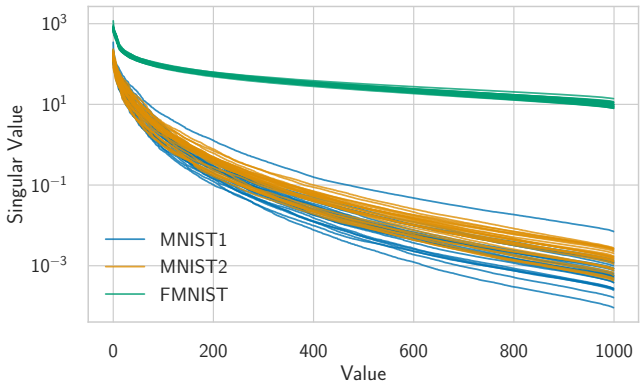

Figure A.2: Singular values of the Jacobian on 5000 images from the first half of MNIST (MNIST1) for25 models trained each on MNIST1, as well the second half of MNIST (MNIST2) and on Fashion MNIST (FMNIST). The singular values decay much more rapidly (and in a similar fashion) for models trained on MNIST1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

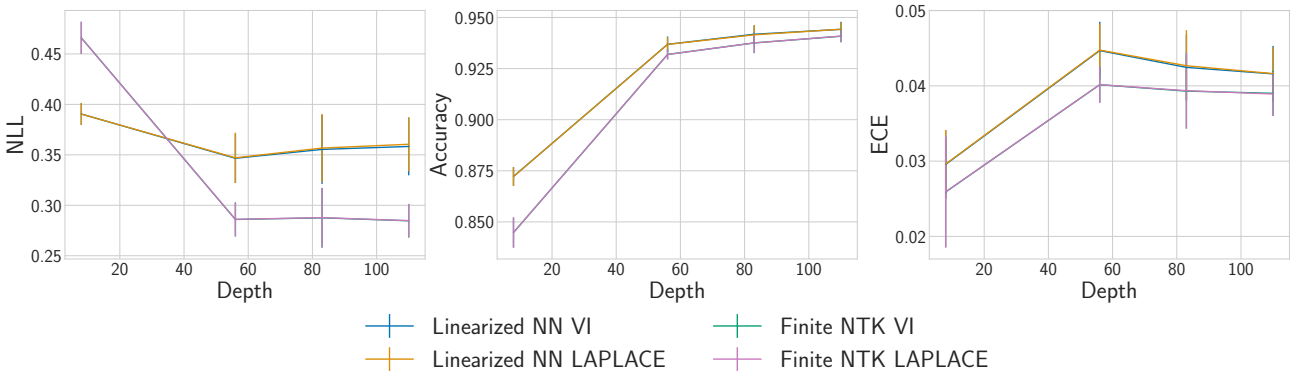

Figure A.3: Test Accuracy of PreResNets averaged over 10 random seeds. Both VI and Laplace are included here to see if the approximate inference method mattered much for either the linearized NN or the finite NTK. They perform similarly with some small differences.

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.

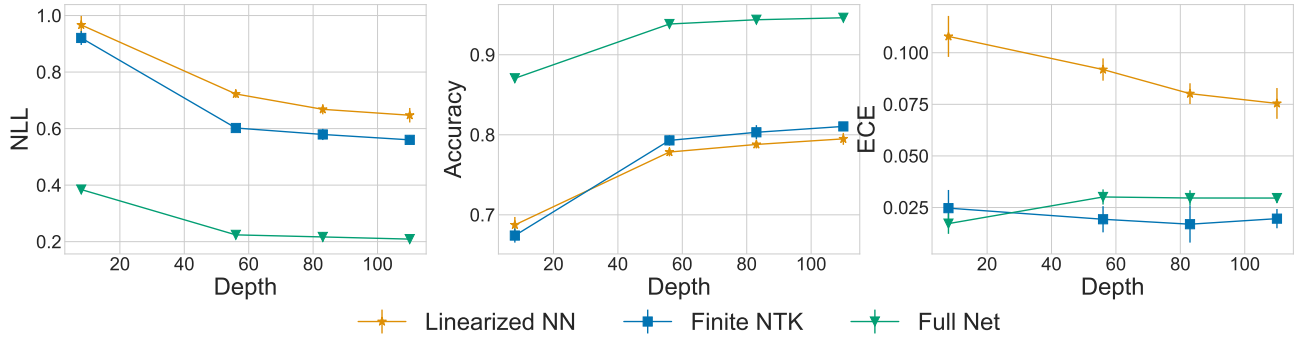

Figure A.4: Test Accuracy of PreResNets averaged over 10 random seeds with fine-tuning (green), the linearized NN (blue) and the finite NTK (orange) after training on CIFAR-10 and transferring to STL10. Linearization using solely the Jacobian as the features (yellow) is competitive with fine-tuning the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

F.3 Further Sinusoids Plots

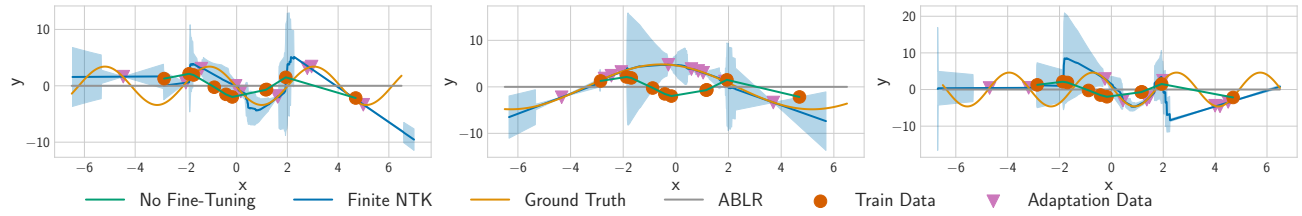

Figure A.5: Posterior predictions on a few shot regression task produced with the NTK as the kernel using a network with ReLU activations. Here, the ReLU network is a poor inductive bias for this task, as is reflected in the jagged predictive variances and in the ABLR comparison which is completely flat.

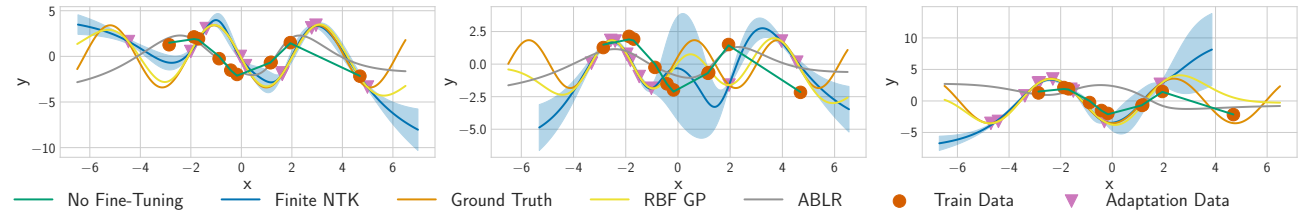

Figure A.6: Posterior predictions on a few shot regression task produced with the NTK as the kernel using Fisher vector products instead of Jacobian vector products. These results are nearly identical to the results with Jacobian vector products described in the main text.

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

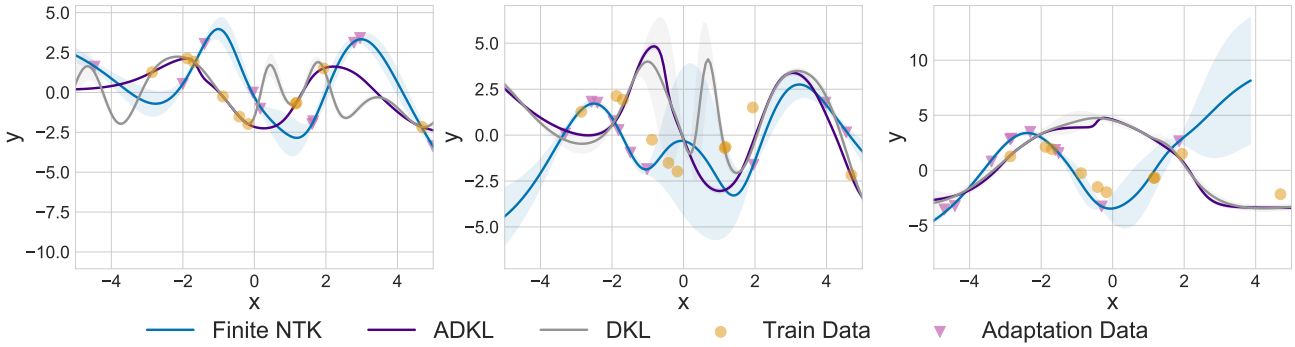

Figure A.7: Comparison to ADKL (Tossou et al., 2019) and transferred DKL, a two task version of Patacchiola et al. (2019) on the sinusoids problem. Both ADKL and transferred DKL underfit.

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