Workspace 'Workspace' in 'Neural Tangent Kernel Distillation LessWrong'

Page 1 (row 1, column 1)

https://distill.pub/2019/visual-exploration-gaussian-processes/

Excellent Resource

https://colab.research.google.com/drive/1fDp8qFLGB2UAbuN--zRAFge00S4_B7iy?usp=sharing#scrollTo=V3rl49_6HrSh

Nice colab with code.

Neural Tangent Kernel Distillation

by **Thomas Larsen, Jeremy Gillen** 5th Oct 2022

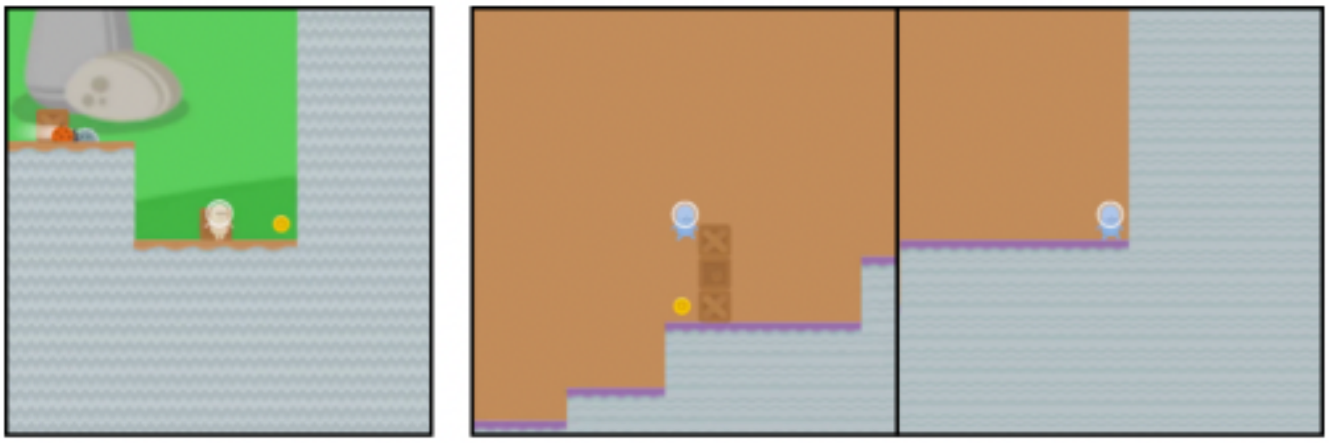
65

SERI MATS AI Frontpage

Produced As Part Of The SERI ML Alignment Theory Scholars Program 2022 Under John Wentworth.

Introduction

Consider the following example from Goal Misgeneralization: you train an RL agent to pursue a coin, which is always at the same location (at the end of the level) during training. At test time, if the coin position is randomized, does the RL agent pursue the coin, or does it go to the fixed location?


(a) Goal position fixed (b) Goal position randomized

Since the training data could not distinguish between the two goals ('go to the coin' and 'go to the location'), which one is chosen is purely based on the neural network prior. It would be nice to be able to use simplicity priors like the Solomonoff prior to think about this: the neural network might tend to choose the 'simplest' extrapolation. But what is the right notion of simplicity for neural networks? It's clearly not program length in any normal programming language, because the parity function[1] is very short but hard for neural networks to learn.

What does the NN learn?

"Simplest" is hard to define.

In this post, we will summarize some recent advances in DNN theory that have given us the ability to describe the prior of a deep learning network, and discuss the relevance to alignment. Our goal is to communicate quickly the insights that we found difficult to understand or slow to extract from the sources we were using.

Neural Tangent Kernel (NTK) theory compares neural networks to kernel methods. [2] The most interesting takeaways (to us) are:

- We can make a "linearized" neural network and prove that training this is equivalent to kernel regression.
- As we increase the width of a normal neural network, we can prove that it will behave more like the linearized network.
- By relating this to Gaussian Process inference, we can think of the neural network as doing Bayesian inference, and describe and visualize the prior distribution that it is using.
    - We can print out features/eigenmodes (functions from input vectors to output vectors), and think of the neural network as finding the best linear combination of these, $\underline{\text{with a preference for using the earlier eigenmodes}}$.

$\hookrightarrow$ Key

Prerequisites: Linear Algebra, Gaussian Processes (GPs), to the level explained here, a source we highly recommend playing with to gain GP intuition.

Background

Notation

Let f denote the network architecture, a function that takes in the parameters and a network input, and outputs a predicted label. $\theta \in \mathbb{R}^d$ denotes the parameters, $x \in \mathbb{R}^k$ is the input to the network, and $y = f(\theta, x) \in \mathbb{R}$ is the output of the network.[3] We will use $X \in \mathbb{R}^{n \times k}$ to denote the network inputs, and $Y \in \mathbb{R}^n$ to denote the network outputs, and denote $f(\theta, X)$ to be the network evaluated on each x in X. $\phi : \mathbb{R}^k \to \mathbb{R}^m$ denotes a feature map associated with a kernel.

We will use L2 loss: $L(\theta) = \frac{1}{2} \sum_i (f(\theta, x_i) - y_i)^2$.

Kernel Methods

Kernel Methods have been extensively studied in the pre-deep learning era. And it turns out that we can use insights from this area to better understand neural networks.

A kernel is a function that tells you how *a priori* similar any two data points are.

There are three intuitive ways I think about kernel methods:

1. A kernel method predicts a label by taking a weighted average of the labels of nearby data points, weighted by how close the kernel thinks the data points are.
2. From a Bayesian point of view, a kernel gives you the *a priori* covariance between data points, which we can use as a prior to do Bayesian inference.
3. A kernel method transforms data into a fixed feature space, then does linear regression on the data points in that space (with some prior over the linear regression parameters).

Kernel Linear Regression

Classical linear regression works as follows: you want to find a parameter vector θ to predict data labels Y: you want $Y = X\theta$.[4] It's pretty easy to just solve for the θ that is closest:

$$\begin{aligned} Y &= X\theta \\ X^TY &= X^TX\theta \\ (X^TX)^{-1}X^TY &= \theta \end{aligned}$$

In order to get prediction for a new input, x, now that you have θ, you can simply multiply by theta:

$$\boxed{\begin{aligned} \hat{y} &= x\theta \\ &= x(X^TX)^{-1}X^TY \end{aligned}}$$

Kernel regression generalizes linear regression: instead of fitting a linear predictor from the feature space $\mathbb{R}^n$, we pick a kernel function $\phi : \mathbb{R}^n \to \mathbb{R}^m$ that picks out features of the input space, and then do linear regression in the higher dimensional space. We also assume that the parameters learned by the linear regression, θ, can be expressed as $\theta = \phi(X)^T w$, i.e., that it is a linear combination of features extracted by some feature function ϕ.[5] So:

$$\begin{aligned} Y &= \phi(X)\theta \\ &= \phi(X)\phi(X)^T w \end{aligned}$$

Solving for w gives:

$$w = (\phi(X)\phi(X)^T)^{-1}Y$$

$$\begin{aligned} \hat{y} &= \phi(x)\theta \\ &= \phi(x)\,\phi(X)^T w \end{aligned}$$

Now, we can substitute in:

$$\boxed{\begin{aligned} \hat{y} &= \phi(x)\phi(X)^T(\phi(X)\phi(X)^T)^{-1}Y \\ &= K(x,X)K^{-1}(X,X)Y \end{aligned}}$$

This is the equation for kernel regression, where we can understand the first two terms as being a weighted similarity vector of our test data point to each of the training data points, which is dotted with the training labels.

Neural Tangent Kernel

How are neural networks ≈ kernel methods?

Normally, you treat the neural network as $f(\theta, \cdot)$, fixing θ, so you get simply a map from inputs to outputs. But there's another way of thinking about it which is as a parameter to function map, given a fixed x. In particular, we can do a Taylor expansion of the parameter function map around θ_0, the initialization.

$$f(x,\theta) \approx f(x,\theta_0) + \nabla_\theta f(x,\theta_0) \cdot (\theta - \theta_0)$$

The error of this approximation is $o(\|\theta - \theta_0\|^2)$, so the less the parameters are updated during training, the better this approximation is. One of the key results behind NTK research is that as the width of a network increases toward infinity, the parameters change less during training.

But for a moment, let's keep the width finite. At finite w, we can make a new learning algorithm called a "linearized neural network", which is described by this equation:

$$f_{linear}(x, \theta) = f(x, \theta_0) + \nabla_\theta f(x, \theta_0) \cdot (\theta - \theta_0)$$

This equation describes (almost) linear regression on a particular feature space $\phi(x) = \nabla_\theta f(x, \theta_0)$:

$$\begin{aligned} f_{linear}(x, \theta) &= f(x, \theta_0) + \phi(x) \cdot (\theta - \theta_0) \\ &\approx \phi(x) \cdot \theta \end{aligned}$$

☆ Important

As we learned above in the Kernel Linear Regression section, linear regression on a feature space is equivalent to Kernel Regression with $K(X, X) = \phi(X)\phi(X)^T$!

Hence, training f_{linear} is equivalent to doing:

$$\hat{y} = K(x, X)K^{-1}(X, X)Y$$

where $K(x_1, x_2) = \nabla_\theta f(\theta_0, x_1) \cdot \nabla_\theta f(\theta_0, x_2)$.

The only major insight left is that $w \to \infty$ implies $\|\theta - \theta_0\| \to 0$, which means $f_{linear} \to f$. This is non-trivial to prove and depends on the initialization distribution.[6]

The NTK function

This Kernel regression motivates the definition of the NTK function:

$$\begin{aligned} NTK &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \\ & x_1, x_2 \mapsto \nabla_\theta f(\theta, x_1) \cdot \nabla_\theta f(\theta, x_2)|_{\theta=\theta_0} \end{aligned}$$

We can think of the NTK function as telling you the 'similarity' of two given data points according to the feature map at initialization. This is not to be confused with the NTK matrix: the matrix whose i, j-th component is $NTK(x_i, x_j)$, for a set of input datapoints X.

→ Proof of all this there in LLW blog.

The last result that we need to know is that the NTK stops depending on θ_0 when the width is ∞. We won't prove this here, but the sketch is that if we expand out $\nabla_\theta f(\theta, x_1) \cdot \nabla_\theta f(\theta, x_2)|_{\theta=\theta}$ for a particular neural network architecture, it ends up having a lot of sums over the weights. When the width is ∞, these sums become expectations.

The NTK in the infinite width limit can be written out analytically, e.g. the one for 2 layer ReLU network is:[7]

$$K(x, x') = \|x\|\|x'\|\kappa(\frac{x \cdot x'}{\|x\|\|x'\|})$$

where:

$$\kappa(u) := 2u\frac{1}{\pi}(\pi - \arccos(u)) + \frac{1}{\pi}\sqrt{1-u^2}$$

Prior over data

We can view any kernel method as giving us the posterior mean of a Gaussian Process (see the *Marginalization and Conditioning* section of this to see why, although beware that they are using different variable names which is confusing[8]).

So we can think of the NTK as giving us a prior over the labels of a given data distribution X, specifically:

Imp:

$$\boxed{Y \sim \mathcal{N}(0, H) = \frac{e^{-\frac{1}{2}Y^T H^{-1} Y}}{\sqrt{2\pi \det H}}}$$

Where H is the NTK matrix, which depends on our dataset: $H = \nabla_\theta f(\theta, X)^T \nabla_\theta f(\theta, X)$. We will abbreviate the constant denominator $C = \sqrt{2\pi \det H}$.

Kernel Eigenmodes

We can understand this prior via eigendecomposition. Since H is positive semidefinite, it is symmetric, and so the Spectral Theorem applies, allowing us to eigendecompose H into $H = PDP^{-1}$, where D is a diagonal matrix of the eigenvalues, and P is the matrix of eigenvectors of H.

$$H = PDP^{-1} = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \dots & v_n \\ | & | & & | \end{bmatrix} \cdot \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \cdot \begin{bmatrix} — & v_1^T & — \\ — & v_2^T & — \\ & \vdots & \\ — & v_n^T & — \end{bmatrix}$$

Here, $v_1 \dots v_n$ are the eigenvectors of H, with corresponding eigenvalues $\lambda_1 \geq \lambda_2 \dots \geq \lambda_n$.

Eigendecomposing H gives $H = PDP^{-1}$, and the labels are sampled from a Gaussian Process, so the log probability density is:

$$\begin{aligned} \log p(Y) &= \log \left(\frac{e^{-\frac{1}{2} Y^T H^{-1} Y}}{C} \right) \\ &= -\frac{1}{2} \sum_{i=1}^{n} (v_i \cdot Y)^2 \lambda_i - \log(C) \end{aligned}$$

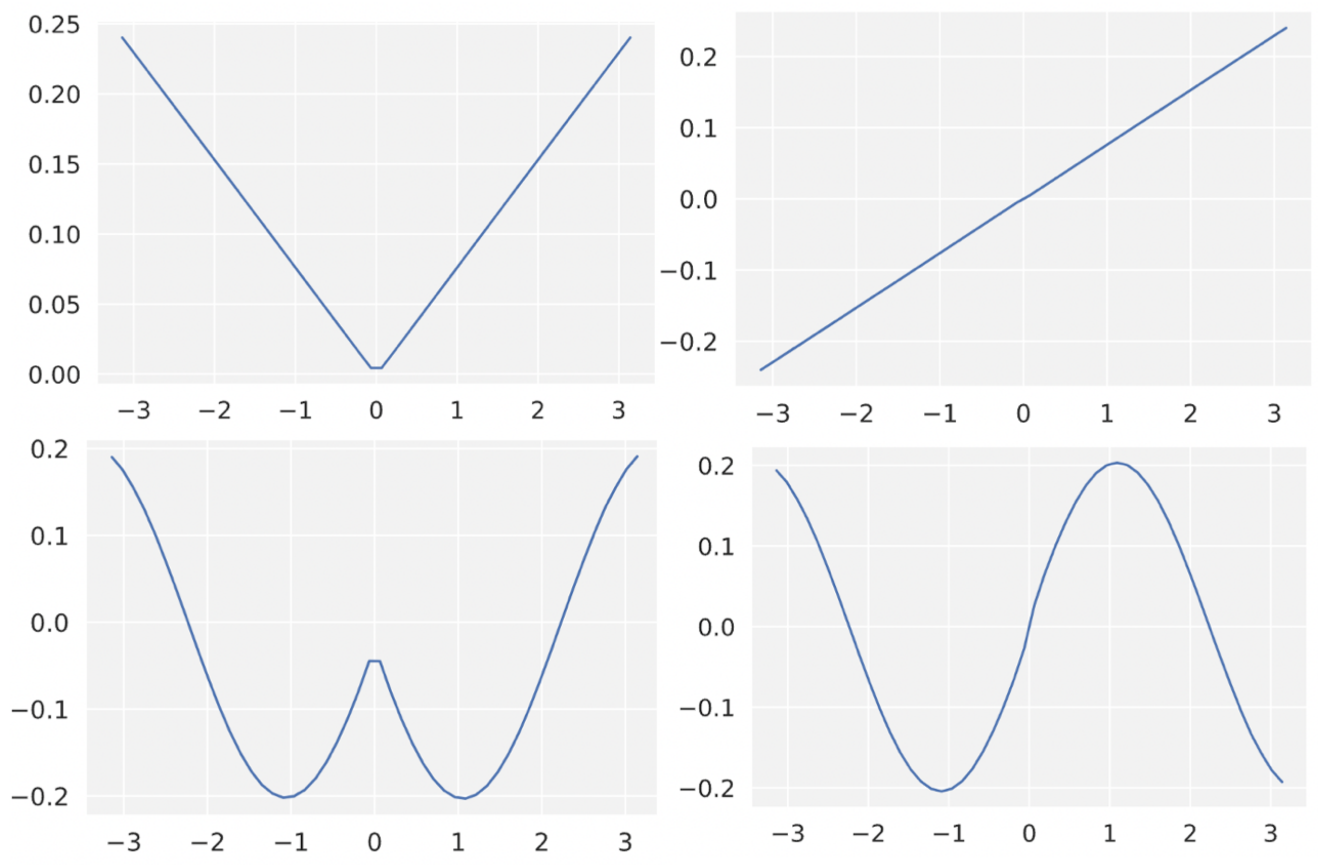
We can think of $(v_i \cdot Y)^2$ as the correlation between the dataset and each eigenvector. See footnote for the full derivation.[9]

This is an explicit prior for a neural network. We can predict how a neural network will generalize to certain test points X^{test}, when trained on the training data points X^{train} with labels Y^{train}. The way to do this is by computing the NTK with $X = [X^{test}, X^{train}]$, then calculating $\log p([Y^{test}, Y^{train}])$ for several different versions of Y^{test}. This version of the test labels that gives the highest prior probability is the generalization most likely to be chosen. This is analogous to conditioning this Gaussian on the labels Y^{train}.

$\longrightarrow$ Interesting perspective.

Visualizing Eigenvectors

We visualize these for a specific neural network below:



The first four eigenvectors for for a 2-layer fully connected neural network with training inputs ranging from 3 to -3.

When we get to the later eigenvectors, they turn out to be all sinusoidal. We can think of neural network training as finding a linear combination of these functions, where it prefers learning the functions with higher eigenvalues. It also learns the ones with higher eigenvalues earlier in training run on each of these, with update learning rate according to their eigenvalue (see appendix for why this is true).

Here is our Google Colab Notebook to generate these results.

Alignment relevance

So we have a mathematically precise notion of the simplicity prior! What does this tell us about alignment?

Unfortunately, not too much. The key problem is abstraction: it's really hard for us to express abstract concepts like 'is this network deceptive?' in the language of the kernel eigenfunctions sine wave decomposition. I am excited for future work to tackle this problem and use NTK theory to predict how neural networks will generalize. For example, could we prove something like "this neural network is very unlikely to learn an algorithm in the set of bounded tree search algorithms"?

We should be able to put any two data points into the kernel and get a measure of how similar they are. This should let us test whether the trained neural network going to treat an image of a lion in the snow° as more similar to a training data point of a husky in the snow or a lion in grass?

Appendix: Modeling training dynamics with gradient flow

A result that we thought was cool that didn't fit anywhere else in this is the proof that infinitely wide neural networks can always get to zero loss. Recall that we model the training dynamics of a neural network as having infinitely small step sizes, called a gradient flow. This allows us to model training as differential equation, where we are continuously updating the parameters θ over time according to how they perform on the loss L:

$$\begin{aligned}\dot{\theta}(t) &= -\nabla_\theta L(\theta(t)) \\ &= -\nabla_\theta \frac{1}{2}\sum_i (f(\theta, x_i) - y_i)^2 \\ &= -\sum_i \nabla_\theta f(\theta, x_i)(f(\theta, x_i) - y_i) \\ &= -\nabla_\theta f(\theta, X) \cdot (f(\theta, X) - Y)\end{aligned}$$

Where we sometimes abbreviate $\theta(t)$ as just θ. This was modeling the gradient flow in *parameter space*, but what we really care about is the dynamics in *function space*. In other words, we care about the changes in the function $f(\theta(t), \cdot)$ as t increases. Fortunately, we can simply compute this on the training set:

$$\begin{aligned}\frac{d}{dt} f(\theta, X) &= \nabla_\theta f(\theta(t), X) \cdot \dot{\theta}(t) \\ &= -\nabla_\theta f(\theta, X) \cdot \nabla_\theta f(\theta, X) \cdot (f(\theta, X) - Y)\end{aligned}$$

We have now found a crucial quantity: $\nabla_\theta f(\theta, X) \cdot \nabla_\theta f(\theta, X)$ is the NTK matrix. Let's call it $H(\theta)$. It turns out that in the limit of width, this quantity is constant over time. Thus:

$$\frac{d}{dt} f(\theta, X) = -H(\theta) \cdot (f(\theta, X) - Y)$$

$f(\theta, X) = Y$ is clearly an equilibrium of this ODE, because when this is satisfied, the RHS is 0. We can explicitly solve this ODE by making the substitution $U(t) = f(\theta, X) - Y$, so:

$$\frac{d}{dt} U(t) = -H(\theta) \cdot U(t)$$

This is a well-known ODE, with solution given by:

$$U(t) = e^{-H(\theta)t} \cdot U(0)$$

This gives us a proof of global convergence on the training data.

---

1. ^ The parity function is $f : \{0, 1\}^* \to \{0, 1\}$, and returns 1 if and only if the input has an odd number of ones.
2. ^ See Jacot et al., Lee et al.
3. ^ We will assume 1-dimensional outputs, because it makes the math much more manageable: the NTK becomes a 4-Tensor with output dimension more than 1.
4. ^ Assume for simplicity that there is no noise and we can perfectly fit the data with a linear function.
5. ^ There is a theorem (the Representer theorem) which says that the loss minimizing hypothesis (within the space of functions associated with this kernel) has a representation of this form.
6. ^ For the actual proof, start at the bottom of p14 of the original paper and work backwards. There's a simplified version here in the One hidden Layer Network proof.

7. ^ I haven't checked all of the derivation of this, I got it from On the Inductive Bias of Neural Tangent Kernels, who seem to have got it by combining the NTK definition in Appendix A of this with analytical evaluation of the integrals from here.

8. ^ See the equations for conditioning a Gaussian, and assume that the prior means μ_X and μ_Y are 0. Then, translating back into our variables, we get:

$$\begin{aligned} x|X &\sim \mathcal{N}\left(K(x,X)K(X,X)^{-1}Y, K(x,x) - K(x,X)K(X,X)^{-1}K(X,x)\right) \\ &\sim \mathcal{N}\left(\mu_{posterior}, \Sigma_{posterior}\right) \end{aligned}$$

9. ^

$$\begin{aligned} \log p(Y) &= \log\left(\frac{e^{-\frac{1}{2}Y^T H^{-1} Y}}{C}\right) \\ &= -\frac{1}{2}Y^T H^{-1} Y - \log(C) \\ &= -\frac{1}{2}Y^T (P^{-1}DP)^{-1} Y - \log(C) \\ &= -\frac{1}{2}Y^T PD^{-1}P^{-1} Y - \log(C) \\ &= -\frac{1}{2}(Y^T P)D^{-1}(P^T Y) - \log(C) \\ &= -\frac{1}{2}\sum_{i=1}^{n}(P^T Y)_i^2 \lambda_i - \log(C) \\ &= -\frac{1}{2}\sum_{i=1}^{n}(v_i \cdot Y)^2 \lambda_i - \log(C) \end{aligned}$$

10. ^ See Jacot et al., Lee et al.

SERI MATS 2 | AI 2 | Frontpage

Mentioned in

75 More Recent Progress in the Theory of Neural Networks

61 QAPR 4: Inductive biases

13 Cataloguing Priors in Theory and Practice

20 comments, sorted by top scoring

[-] **Quintin Pope** 1mo 10 2

I'm very glad to see this post! Academic understanding of deep learning theory has improved quite a lot recently, and I think alignment research should be more aware of such results.

Some related results you may be interested in:

- The actual NTKs of more realistic neural networks continuously change throughout their training process, which impacts learnability / generalization

    - https://arxiv.org/abs/2008.00938
    - https://arxiv.org/abs/2106.06770
- Discrete gradient descent leads to different training dynamics from the small step size / gradient flow regime
    - https://arxiv.org/abs/2009.11162
    - http://arxiv-export-lb.library.cornell.edu/abs/2107.09133

[-] **Ulisse Mini** 15d < 5 > × 4 ✓

I believe you've got a typo in the defn of K

$$K\left(x, x'\right) = \|x\|\|x\|\kappa\left(\frac{x \cdot x'}{\|x\|\|x\|}\right)$$

Shouldn't it be $\|x\|\|x'\|$?

Also it appears like you can simplify

$$\kappa(u) := u\frac{1}{\pi}(\pi - \arccos(u)) + \frac{1}{\pi}\left(u(\pi - \arccos(u)) + \sqrt{1-u^2}\right)$$

Into

$$\kappa(u) := 2u\frac{1}{\pi}(\pi - \arccos(u)) + \frac{1}{\pi}\sqrt{1-u^2}$$

Not sure if this wasn't done for some reason or it was a typo.

[-] **Jeremy Gillen** 14d < 3 > × 0 ✓

Thanks, you are right on both. I don't know how I missed the simplification, I remember wanting to make the analytical form as simple as possible.

I really should have added the reference for this, since I just copied it from a paper, so I did that in a footnote. I just followed up the derivation a bit further and the parts I checked seem solid, but annoying that it's spread out over three papers.

[-] **interstice** 1mo < 5 > × 1 ✓

Nice survey, I think this line of research might be pretty important. Also worth noting that there are some significant aspects of network training that the tangent kernel model can't explain, like feature learning and transfer learning°.

[-] **Jeremy Gillen** 1mo < 2 > × 0 ✓

That makes sense to me about transfer learning, the NTK model simplifies away any path dependence during training, so we would have some difficulty using it directly as a model to understand fine-tuning and transfer learning.

But I've been a little confused about the feature learning thing since I read your post last year. I'm not sure what it means to "explain feature learning". Any Bayesian learning process is just multiplying the prior and likelihood, for each hypothesis. It seems that no feature learning is happening here? Solomonoff induction isn't doing feature learning, right?

Feature learning doesn't seem to be fundamental to interesting/useful learning algorithms, and it also seems plausible to me that a theoretical description of a learning algorithm can have no feature learning, while an efficient approximation to it with roughly the same data efficiency, could have feature learning.

[–] **interstice** 1mo 3 0

> Solomonoff induction isn't doing feature learning, right?

Sure, but neural network training isn't a Bayesian black-box, it has parts we can examine. In particular we see intermediate neurons which learn to represent task-relevant features, but we do *not* see this in the tangent kernel limit.

> it also seems plausible to me that a theoretical description of a learning algorithm can have no feature learning, while an efficient approximation to it with roughly the same data efficiency, could have feature learning

I wouldn't think of neural networks as an approximation to the NTK, rather the reverse. Feature learning makes SGD-trained neural networks more powerful° than their tangent-kernel counterparts.

[–] **Jeremy Gillen** 1mo 2 0

I don't understand why we want a theoretical explanation of neural network generalization to have the same "parts we can examine" as a neural network.
If we could describe a prior such that Bayesian updating on that prior gave the same generalization behavior as neural networks, then this would not "explain feature learning", right? But it would still be a perfectly useful theoretical account of NN generalization.

I agree that evidence seems to suggest that finite width neural networks seem to generalize a little better than infinite width NTK regression. But attributing this to feature learning doesn't follow. Couldn't neural networks just have a slightly better implicit prior, for which the NTK is just an approximation?

[–] **interstice** 1mo 3 0

> If we could describe a prior such that Bayesian updating on that prior gave the same generalization behavior

Sure, I just think that any such prior is likely to explicitly or implicitly explain feature learning, since feature learning is part of what makes SGD-trained networks work.

> But attributing this to feature learning doesn't follow

I think it's likely that dog-nose-detecting neurons play a part in helping to classify dogs, curve-detecting neurons play a part in classifying objects more generally, etc. This is all that is meant by 'feature learning' - intermediate neurons changing in some functionally-useful way. And feature learning is *required* for pre-training on a related task to be helpful, so it would be a weird coincidence if it was useless when training on a single task.

There's also a bunch of examples of interpretability work where they find intermediate neurons having changed in clearly functionally-useful ways. I haven't read it in detail but this article analyzes how a particular family of neurons comes together to implement a curve-detecting algorithm, it's clear that the intermediate neurons have to change substantially in order for this circuit to work.

[–] **johnswentworth** 1mo < 4 > × 0 ✓

Presumably the eigenfunctions are mostly sinusoidal because you're training against a sinusoid? So it's not really relevant that "it's really hard for us to express abstract concepts like 'is this network deceptive?' in the language of the kernel eigenfunctions sine wave decomposition"; presumably the eigenfunctions will be quite different for more realistic problems.

[–] **Thomas Larsen** 1mo < 4 > × 3 ✓

Hmm, the eigenfunctions just depend on the input training data distribution (which we call X), and in this experiment, they are distributed evenly on the interval $[-\pi, \pi)$. Given that the labels are independent of this, you'll get the same NTK eigendecomposition regardless of the target function.

I'll probably spin up some quick experiments in a multiple dimensional input space to see if it looks different, but I would be quite surprised if the eigenfunctions stopped being sinusoidal. Another thing to vary could be the distribution of input points.

[–] **johnswentworth** 1mo < 4 > × 1 ✓

Typically the property which induces sinusoidal eigenfunctions is some kind of permutation invariance - e.g. if you can rotate the system without changing the loss function, that should induce sinusoids.

The underlying reason for this:

- When two matrices commute, they share an eigenspace. In this case, the "commutation" is between the matrix whose eigenvectors we want, and the permutation.
- The eigendecomposition of a permutation matrix is, roughly speaking, a fourier transform, so its eigenvectors are sinusoids.

[illegible]

[–] **Jeremy Gillen** 1mo < 8 > × 1 ✓

We don't fully understand this comment.

Our current understanding is this:

- The kernel matrix K of shape $n \times n$ takes in takes in two label vectors and outputs a real number: $y^\top Ky$. The real number is roughly the negative log prior probability of that label set.
- We can make some orthogonal matrix that transforms the labels y, such that the real number output doesn't change. $(Ry)^\top K(Ry) = y^\top Ky$
    - This is a transformation that keeps the label prior probability the same, for any label vector.
- $(Ry)^\top K(Ry) = y^\top Ky$ for all $y \in \mathbb{R}^n$ iff $RK = KR$, which implies R and K share the same eigenvectors (with some additional assumption about K having different eigenvalues, which we think should be true in this case).
- Therefore we can just find the eigenvectors of R.

But what can R be? If K has multiple eigenvalues that were the same, then we could construct an R that works for all y. But empirically aren't the eigenvalues of K all different?
So we are confused about that.

Also we are confused about this: "without changing the loss function". We aren't sure how the loss function comes into it.

Also this: "training against a sinusoid" seems false? Or we really don't know what this means.

[-] **johnswentworth** 1mo < 3 > × 0 ✓

Ignore the part about training against a sinusoid. That was a more specific hypothesis, the symmetry thing is more general. Also ignore the part about "not changing the loss function", since you've got the right math.

I'm a bit confused that you're calling y a label vector; shouldn't it be shaped like a data pt? E.g. if I'm training an image classifier, that vector should be image-shaped. And then the typical symmetry we'd expect is that the kernel is (approximately) invariant to shifting the image left, right, up or down a pixel, and we could take any of those shifts to be R.

[-] **Jeremy Gillen** 1mo < 3 > × 0 ✓

The eigenfunctions we are calculating are solutions to:

$$\lambda\phi(x') = \int_{x\sim\mathcal{D}} k(x', x)\phi(x)dx$$

Where $\mathcal{D}$ is the data distribution, λ is an eigenvalue and $\phi(x)$ is an eigenfunction.

So the eigenfunction is a label function with input x, a datapoint. The discrete approximation to it is a label vector, which I called y above.

[-] **Jeremy Gillen** 1mo 2 × 1 ✓

I'd expect that as long as the prior favors smoother functions, the eigenfunctions would tend to look sinusoidal?

[-] **Lee Sharkey** 22d 2 × 0 ✓

> This equation describes (almost) linear regression on a particular feature space $\phi(x) = \nabla_\theta f(x, \theta_0)$:
>
> $$\begin{aligned} f_{linear}(x, \theta) &= f(x, \theta_0) + \phi(x) \cdot (\theta - \theta_0) \\ &\approx \phi(x) \cdot \theta \end{aligned}$$

This approximation isn't obvious to me. It holds if $ f(x, \theta_0) \approx 0 $ and $ \theta_0 \approx 0 $, but these aren't stated. Are they true?

[-] **Jeremy Gillen** 18d 4 × 0 ✓

Yeah good point, I should have put more detail here.

My understanding is that, for most common initialization distributions and architectures, $f(x, \theta_0) = 0$ and $\phi(x) \cdot \theta_0 = 0$ in the infinite width limit. This is because they both end up being expectations of random variables that are symmetrically distributed around 0.

However, in the finite width regime if we want to be precise, we can simply add those terms back onto the kernel regression.

So really, with finite width:

$$f_{linear}(x, \theta) = f(x, \theta_0) + K(x, X)K^{-1}(X, X)Y - \nabla_\theta f(x, \theta_0) \cdot \theta_0$$

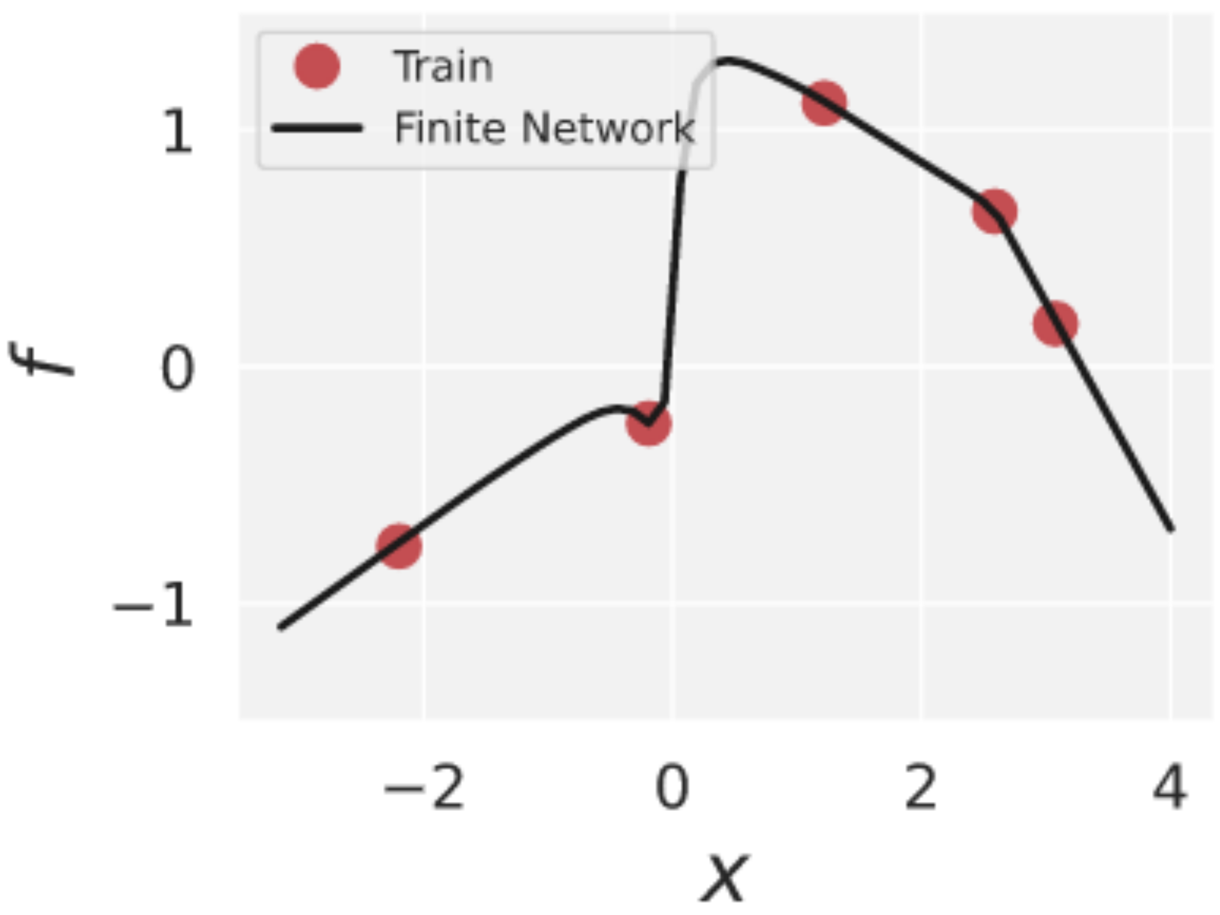
There are a few other very non-rigorous parts of our explanation. Another big one is that $\phi(x) \cdot \theta$ is underspecified by the data in the infinite width limit, so it could fit the data in lots of ways. Stuff about ridge regularized regression and bringing in details about gradient descent fixes this, I believe, but I'm not totally sure whether it changes anything at finite width.

[-] **Lee Sharkey** 17d 1 × 0 ✓

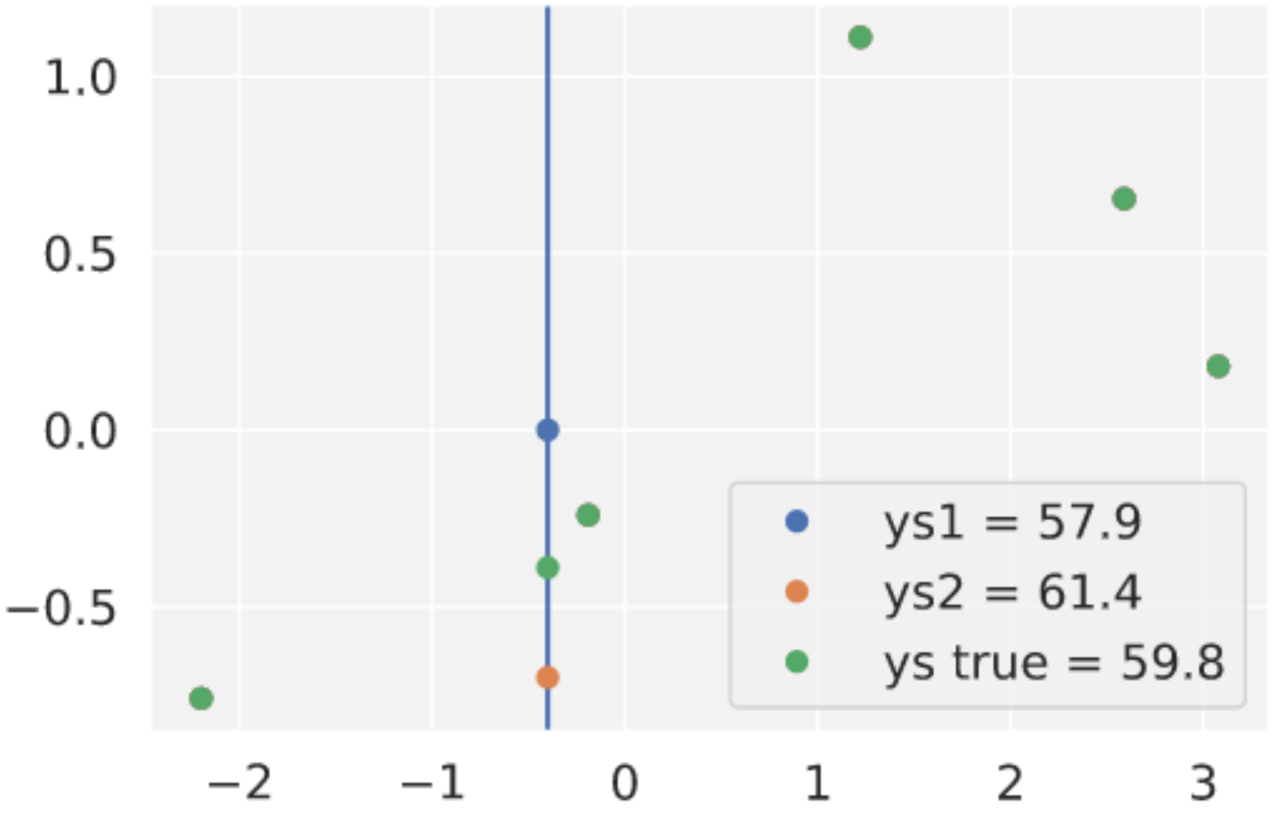
Thanks!

[-] **Jeremy Gillen** 1mo 2 × 1 ✓

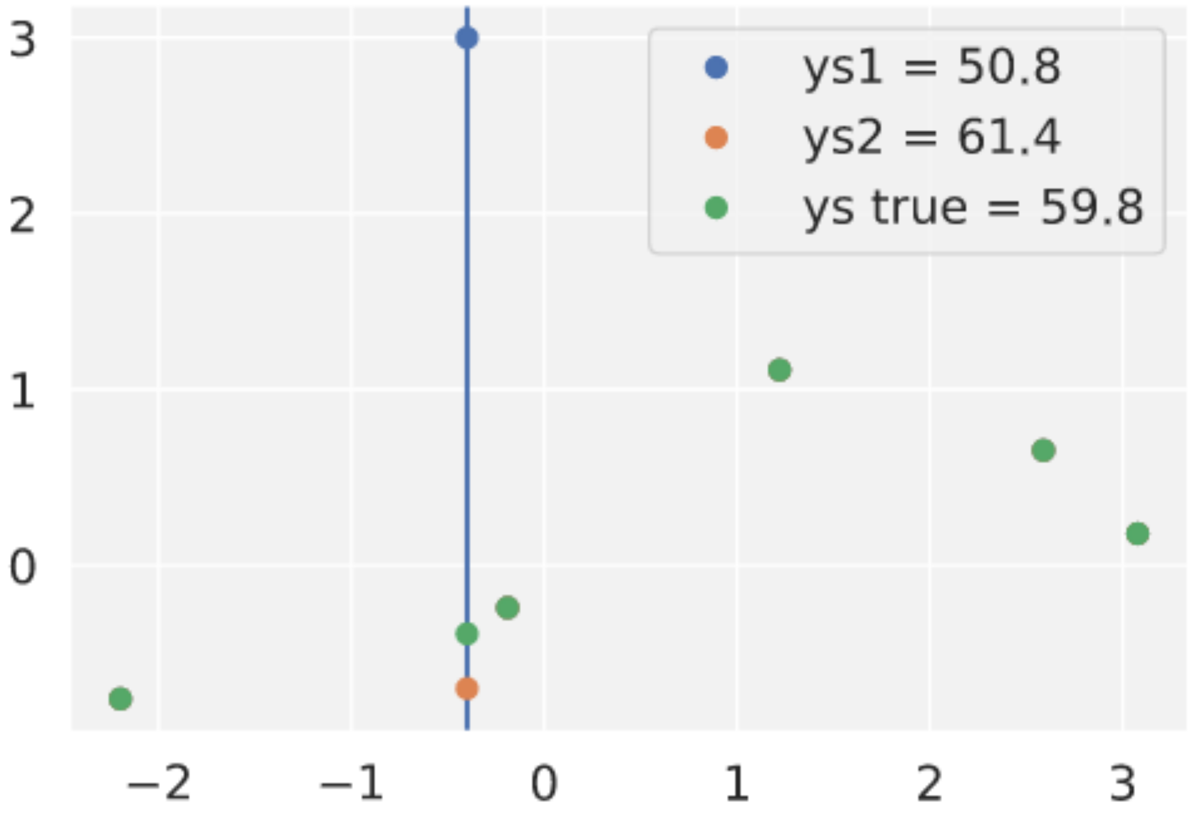
From our implementation in the notebook, we can train a 3 layer ReLU network on 5 datapoints, and it tends to land on a function that looks something like this:



I was curious if the NTK prior would predict the two slightly odd bumps on either side of 0 on x-axis. I usually think of neural networks as doing linear interpolation when trained on tiny 1-d datasets like this, and this shape doesn't really fit that story.
I tested this by adding three possible test datapoints and finding the log-prob of each. As you can see, the blue one has the lowest negative log-prob, so the NTK does predict that a higher data point is more likely at that location:

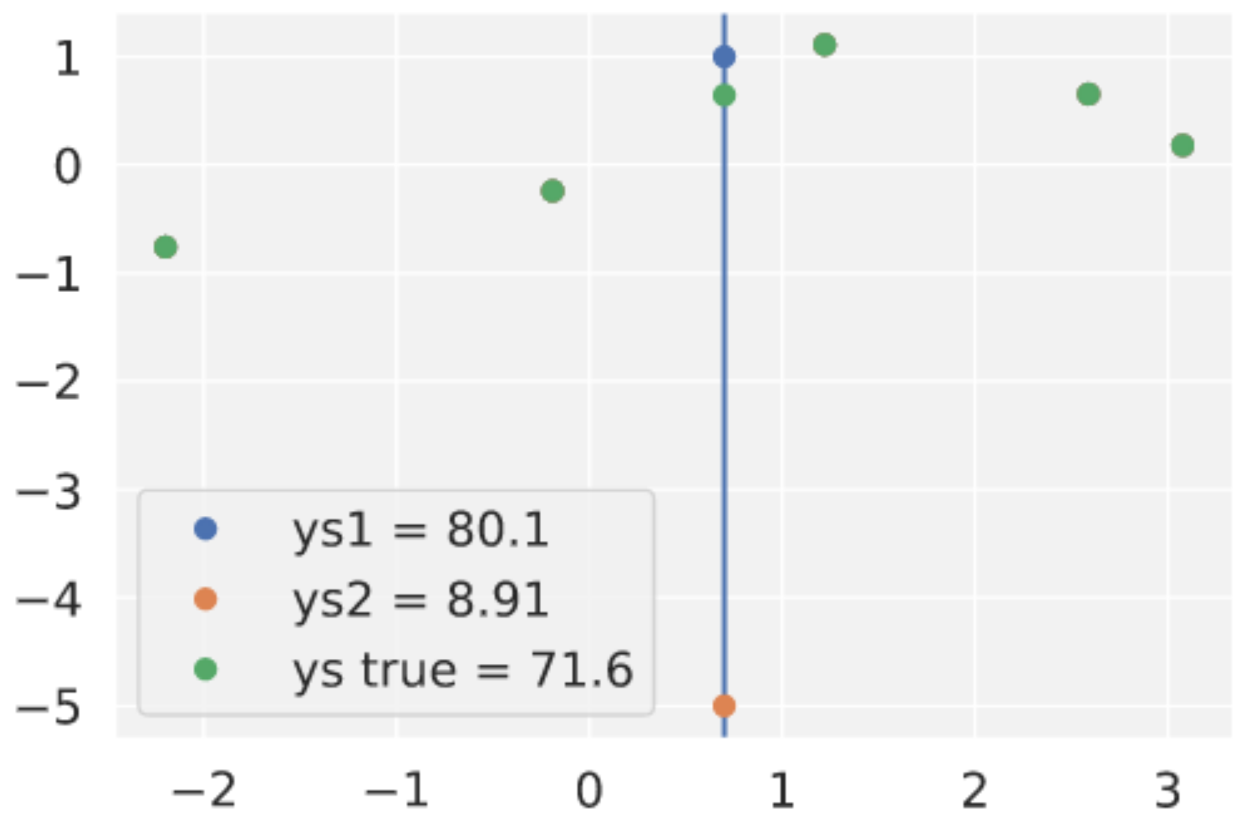


Unfortunately, if I push the blue test point even higher, it gets even more probable, until around y=3:



I'm confused by this. If the NTK predicts y=3 as the most likely, why doesn't the trained neural network have a big spike there?

Another test at y=0.7 to see if the other bump is predicted gives us a really weird result:



The yellow test point is by far the most a priori likely, which just seems wrong, considering the bump in the nn function is in the other direction.

Before publishing this post I'd only tested a few of these, and the results seemed to fit well with what I expected (classic). Now that I've tested more test points, I'm deeply confused by these results, and they make me think I've misunderstood something in the theory or implementation.

[-] **Charlie Steiner** 1mo < 2 > × 0 ✓

There was some Yannic video that I remember thinking did a good job of motivating kernels...

Aha: https://www.youtube.com/watch?v=hAooAOFRsYc

The gist being that they're nonlinearities with a "key" that lets you easily do linear operations on them.