SEARCHING FOR A MORE MINIMAL INTRINSIC DIMENSION OF OBJECTIVE LANDSCAPES

Jonathan Lam & Richard Lee
Department of Electrical Engineering
The Cooper Union for the Advancement of Science and Art
New York, NY 10003, USA
{lam12,lee66}@cooper.edu

ABSTRACT

We present a series of experiments aimed towards finding a more minimal "intrinsic dimension" of an objective landscape, as first described in Li et al. (2018). The intrinsic dimension is the smallest number of free variables needed to solve a given learning problem in a given neural architecture (which together form the objective landscape). It can be roughly thought of as the minimal parameterization of the objective landscape. The original paper presented a method to estimate intrinsic dimension using a random linear projection from the set of intrinsic weights to the set of full network weights; we extend this procedure by first applying a nonlinear mapping on the intrinsic weights before applying the linear projection. We find that applying a power-term nonlinearity does not perform better or worse than the linear projection, and applying a random Fourier feature nonlinearity offers a small but notable gain in accuracy for a given intrinsic dimension.

1 INTRODUCTION

Neural networks have been at the forefront of machine learning applications, and their expressivity causes us to wonder how complex of a problem a given network architecture can solve; or, conversely, the minimum size network needed to describe some learning problem. While the former is mostly a theoretical question of the capacity of the the largest networks, such as GPT-3 (Brown et al. (2020)), the latter is of practical importance in model deployment on embedded systems with limited resources. Many models have been suggested for reducing the memory and computational requirements of neural networks to meet these requirements without overcompromising accuracy; Cheng et al. (2017) presents a survey of such methods.

A related question lies not in the capacity of a model, but rather the size constraint of a given learning problem. This may be useful when reducing the size of a network may be harmful its capacity (e.g., parameter pruning or otherwise reducing the depth, width, units in a dense layer, and/or filters or kernel size in a convolutional layer), and thus we focus on the problem size rather than the network size; or when the the weights in all possible solution sets are not totally independent of one another (and thus can be reparameterized by some smaller problem). Toward the former point, Urban et al. (2017) discovers that the deep and convolutional structure of common image classification problems is very efficient, such that changing or reducing the network structure such as in knowledge distillation may have counterproductive effects. Thus we focus on the question: given a neural network architecture and a specific learning problem, what is the minimum description length (MDL) of a given objective landscape (which depends both on the architecture and the learning problem), without changing the overall network architecture? Li et al. (2018) attempt to tackle this problem by measuring what they call the "intrinsic dimension" of a problem. The "intrinsic weights" are a set of d weights, which are then projected onto the network's set of D weights, where (presumably) $d < D$.

Li's formulation follows the form (following the notation in Li's paper):

$$\theta^{(D)} = \theta_0^{(D)} + P\theta^{(d)} \tag{1}$$

where $\theta^{(D)} \in \mathbb{R}^D$ is the set of all weights in the network, $\theta_0^{(D)} \in M_{D\times 1}(\mathbb{R})$ is a non-trainable randomly-initiated set of initial biases for $\theta^{(D)}$, $P \in M_{d\times D}(\mathbb{R})$ is a non-trainable randomly-initiated linear projection matrix, and $\theta^{(d)} \in \mathbb{R}^d$ is the set of trainable intrinsic weights. Each set of weights θ can be each thought of as a column vector in this matrix equation, although the resulting $\theta^{(D)}$ has to be reshaped into the form necessary for its operation (e.g., reshaped into a convolutional kernel for a convolutional layer). We denote $\theta_0^{(D)}$ the "initial weights" to distinguish it from "bias weights," which more typically refers to the the offset weights (e.g., the set of weights b in the dense layer calculation $W\vec{x} + \vec{b}$).

Another rough interpretation of this formulation is that the model's weights $\theta^{(D)}$ contain some large amount of information. However, it may be possible to find a function that manages to encode much of those redundancies of that information into a much smaller number of weights, $\theta^{(d)}$.

Li presents a toy problem to describe the intuition behind this solution, in which only ten simple linear constraints on the weights need to be met to constitute a correct (zero-loss) solution in a network with 1000 weights. Even if we randomly fix 990 of the weights, we can still solve the solution in a ten-dimensional space by ordinary gradient descent; in this contrived example, the intrinsic dimension of the problem is 10, and there are 990 degrees of freedom. If we randomly fix $\theta_0^{(D)}$, generate a random projection matrix P that will map the ten intrinsic dimensions to the full 1000 dimensions, and only train the smaller-dimensional vector $\theta^{(d)}$, we should be able to reach a solution.

This also gives a straightforward method to compress a model, namely by assuming a certain network architecture, and storing the seeds for the random initialization of $\theta_0^{(D)}$ and P, as well as the intrinsic weights. This has a memory savings of d/D for transmission or storage of the model. It is important to note that this method does not cause inference-time savings in either memory or computation, but the insight offered by the estimated intrinsic dimensionality may be used to engineer a more efficient network.

Our goal is to take the form of (1) and experiment with different forms of projection. While Li attempted to train in a random subspace $\mathbb{R}^d$ of $\mathbb{R}^D$, which is defined by a linear projection matrix P, we experiment with different techniques for projection. Most notably, we aim to look at nonlinear mappings, with a focus on random Fourier features and power function mappings. This involves a projection of the more general form:

$$\theta^{(D)} = \theta_0^{(D)} + f_{proj}\left(\theta^{(d)}\right) \tag{2}$$

In our paper, our projection function consists of "augmenting" the intrinsic weights with a set of nonlinear mappings performed on it, and then performing a projection from this augmented space onto the final output space. This is akin to adding nonlinear features or interaction terms before applying a machine learning model in order to capture more interesting relations between the input features (in this case, the "features" are $\theta^{(d)}$). This can be represented in the following form:

$$\theta^{(D)} = \theta_0^{(D)} + P \begin{bmatrix} \theta^{(d)} \\ f_1(\theta^{(d)}) \\ f_2(\theta^{(d)}) \\ \vdots \end{bmatrix} \tag{3}$$

The goal of these experiments is to try to reduce the intrinsic dimension even further; there may be some redundancies that may not be best captured in linear mappings, and thus allow us to create an even smaller MDL, or gain more insights about the weights of a trained network. Moreover, since Li's paper is the seminal work on this idea of "intrinsic dimension" and has not spawned any derivative papers to date, we experiment with various facets of the architecture that were unexplored in the original paper.[1]

[1]Relevant code is located at github.com/jlam55555/intrinsic-dimension-projections.

2 Related work

While model compression is not the ultimate goal of our research, it is perhaps the most straightforward application of attempting to find the smallest intrinsic dimension, as this leads to a smaller parameterization of the model. Cheng et al. (2017) present a survey of methods related to model compression and acceleration; four main techniques appear: parameter pruning and/or quantization, low-rank factorization, transferred or compact convolutional filters, and knowledge distillation. The method we use is most similar to the low-rank factorization method, as we attempt to factor every layer's weights into the product of a projection matrix and the same set of weights. The major difference is that one matrix in every factorization pair is fixed, and the other matrix is the same for all factorization pairs; i.e., there is only one set of intrinsic weights for the entire network, not for each trainable weight vector in the model. This allows our method to achieve a result closer to the MDL than low-rank factorization, albeit at the cost of accuracy.

Li et al. (2018) perform a number of experiments on the implications of intrinsic dimension, such as examining effect of model shape (depth and width), comparing the weight efficiency of convolutional versus fully-connected architectures for different objective landscapes (learning problems of different natures), and finding the intrinsic dimension of various common problems such as MNIST and CIFAR-10. However, there is still a largely-unexplored space in the formulation of the intrinsic dimension that this paper does not consider; we attempt to explore some of those orthogonal directions.

3 Methods and intuition

3.1 Designing the baseline architecture

There is a high-dimensional search space of hyperparameters for even simple problems such as the spiral classification problem and MNIST. We use the simple fully-connected model for MNIST as presented in Li's paper. This is a 784-200-200 FC network with a softmax final layer ($D \approx 2 \times 10^5$). We choose this simple model mainly for its simplicity and low training time. Using our custom projection layers significantly increases training time, so this was useful given our limited time and resources.

We conjecture that these results will extend to larger models. Li shows empirical evidence in the seminal work on intrinsic dimension that the concepts of intrinsic dimension were extensible to larger networks and were valid for both convolutional and fully-connected layers.

3.2 Linear projection

The linear projection matrix P may be simply generated by random initialization: this results in a (very large) dense matrix. Since the output dimensionality D is large, the resulting vectors in the output space (the columns of P) can be approximated as roughly orthogonal (https://math.stackexchange.com/users/6179/did). If these vectors are normalized (forming an orthonormal basis), then distance in the intrinsic and output dimensions is preserved.

In Li et al. (2018), the columns are normalized but not explicitly orthogonalized. Besides the simple random dense projection, two other random linear projection methods (a sparse method and the Fastfood method) are used. These methods decrease computational and memory cost but do not benefit compression, and we do not experiment with them in our research. We use the basic dense linear projection in our base model. We do not explicitly orthogonalize the vectors in the output space. We also experiment with normalizing these vectors.

We implement the formula (1) on a per-layer basis; i.e., while there is only one set of intrinsic dimension weights, there is a separate $\theta_0^{(D)}$ and P initialized for every layer. This offers some flexibility in case different initialization methods are desired; if all initialization methods are set to the same value, then this method is equivalent do agglomerating all of the weights of the entire model into a model-wide $\theta^{(D)}$.

This raises the question of initialization of P, and regularization. Intuitively, the initialization of the non-trainable variables P and $\theta_0^{(D)}$ may seem more critical to performance than in the case of trainable parameters. Following Li's example, we use a random normal initializer for P, the `he_normal` initializer initially proposed in He et al. (2015) for $\theta_0^{(D)}$, and perform regularization on $\theta^{(D)}$ rather than directly on the trainable weights $\theta^{(d)}$. Attempting different initialization and regularization schemes may be a direction for future research.

3.3 Power terms augmentation

The use of nonlinear mappings is based on the simple intuition that nonlinear functions can capture more complex relationships (and thus more information) than linear functions given the same number of parameters. A quadratic relationship that can be expressed in three parameters can take multiple linear approximations to estimate well. While the toy problem involved a linear constraint on the weights in the solution set, we conjecture that this is likely not always the case. Thus we might be able to design a better $f_{proj}(\theta^{(d)})$ than a simple linear projection (i.e., a matrix multiplication).

The simplest nonlinear mapping we can perform is a quadratic mapping. The most general quadratic form would be:

$$\theta_k^{(D)} = \sum_{1\le i\le d} a_{ik}\theta_i^{(d)} + \sum_{1\le i\le j\le d} b_{ijk}\theta_i^{(d)}\theta_j^{(d)} \tag{4}$$

This can be represented as the concatenation of $\theta^{(d)}$ with the quadratic terms $\theta_i^{(d)}\theta_j^{(d)}$ as one long column vector. This would form a $\mathbb{R}^{d+d^2/2}$ column vector, which could then be projected by an augmented projection matrix. (Consider this a form of augmenting the intrinsic weights with quadratic terms, and then performing a linear projection of these augmented weights onto the full network weights.)

With this method, however, the number of cross terms grows quadratically, which causes a blowup in memory usage. If we ignore cross terms, we can still have squared terms with double the number of weights. Similarly, we can have cubed terms, fourth-power terms, etc. with a linear memory requirement:

$$\theta_k^{(D)} = \sum_{i=1}^{d} a_{ik}\theta_i^{(d)} + b_{ik}\left(\theta_i^{(d)}\right)^2 + c_{ik}\left(\theta_i^{(d)}\right)^3 + \cdots \tag{5}$$

We conjecture that this might be able to have some of the benefit of nonlinear (polynomial) functions without an unreasonable number of parameters. In other words:

$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ \left(\theta^{(d)}\right)^2 \\ \left(\theta^{(d)}\right)^3 \\ \vdots \end{bmatrix} \tag{6}$$

where the exponents denote Hadamard (element-wise) exponentiation. In our experimentation we (arbitrarily) limit ourselves to squared and cubed terms.

Adding new function types also introduces many new hyperparameters, especially for the initialization of the random vectors. That is, if the weights of the projection matrix P were drawn from the same distribution, then for $\theta_i^{(d)} \gg 1$, squaring or cubing could result in a few dominant intrinsic weights, and for $\theta_i^{(d)} \ll 1$, squaring or cubing would result in a few intrinsic weights of negligible impact.

In other words, each $\theta_j^{(D)}$ is formed by a linear combination of terms from the augmented intrinsic weights matrix, which can be thought of as basis functions. Since the linear, squared, and cubed terms are not independent, we are forcing $\theta_j^{(D)}$ to be expressed as a sum of "basis functions" that are third-degree polynomials of a single variable ($\theta_i^{(d)}$) with random, fixed coefficients. Intuitively, polynomials with larger high-order (second- and third-order) coefficients are more nonlinear and more unstable. We were not exactly sure what the effect of this would be: would larger high-order coefficients increase performance by improving expressivity, or would it hurt convergence because of the extra constraints on each intrinsic weight? The results in the following sections show that forcing the basis functions to be more linear by attenuating the squared and cubed coefficients had surprisingly little effect on accuracy.

In order to check our intuition, we experiment with a "pre-training" session in which the projection matrix is trainable. More details follow in the experiments section. Following the recurring theme of this paper, there is a large unexplored space in initialization and regularization of the intrinsic weights and projection matrix.

3.4 Random Fourier features augmentation

The concept of Fourier features has been recognized for it success with scaling up the "kernel trick" and have been analyzed in great mathematical detail (see Li et al. (2019); Rahimi & Recht (2007)). However, we will focus on the more intuitive explanation given by Tancik et al. (2020), which focuses on the inverse rendering problem. In the inverse rendering problem, a network is trained on an image to output the pixel value given a set of coordinates (x, y). Historically, this problem learns only low-frequency functions (general patches of an image), and is not well-suited for high-frequency features (details and textures). Tancik et al. (2020) employs the mapping γ defined as follows to the input vector $\vec{v} = (x, y)$.

$$\gamma(\vec{v}) = \begin{bmatrix} a_1 \cos(2\pi \vec{b}_1^T \vec{v}) \\ a_1 \sin(2\pi \vec{b}_1^T \vec{v}) \\ a_2 \cos(2\pi \vec{b}_2^T \vec{v}) \\ a_2 \sin(2\pi \vec{b}_2^T \vec{v}) \\ \vdots \\ a_m \cos(2\pi \vec{b}_M^T \vec{v}) \\ a_m \sin(2\pi \vec{b}_M^T \vec{v}) \end{bmatrix} \tag{7}$$

The vectors $\vec{b}_m,\ m = 1, 2, \ldots, M$ are random vectors; they can be interpreted as a random frequency term in M-dimensions. In the inverse rendering case, this can be interpreted as "frequency-sampling" the image for patterns centered at $\vec{v}$ with M random frequencies (random in both magnitude and direction). This mapping is periodic and shift-invariant, and more amenable to high-frequency patterns. The range of frequencies that will be captured by this pattern are dependent on the distribution from which the elements of $\vec{b}_m$ are drawn; various distributions are discussed in Tancik's paper.

While we don't have the geometric interpretation of frequencies as in the inverse rendering problem, the random Fourier feature mapping can be thought of as a general way to extract more interesting interactions between features. Using the same framework as in the power terms, we augment the (linear) intrinsic weight terms with a set of random Fourier features.

$$\theta_j^{(D)} = \sum_i a_{ij} \theta_i^{(d)} + \sum_m^M \cos\left(b_m \theta^{(d)}\right) + \sin\left(b_m \theta^{(d)}\right) \tag{8}$$

The equivalent operation may be written as a matrix operation. Note that this still requires a projection from the Fourier feature-augmented intrinsic weights to the output dimension D.

$$\theta^{(D)} = P \begin{bmatrix} \theta^{(d)} \\ \cos\left(B\theta^{(d)}\right) \\ \sin\left(B\theta^{(d)}\right) \end{bmatrix} \tag{9}$$

Since the (real-valued) sine and cosine functions are bounded, we don't have to worry as much about a blowup of these terms as we did with the power terms.

3.5 Datasets and preprocessing

Note that, despite the ability of this method to compress a network for storage or transmission, this method increases computational and memory cost during training. Inference is not affected. Given that P is a $D \times d$ matrix, this has a much higher memory consumption than the original network containing only D weights. Using the sparse or Fastfood methods should partially alleviate this problem. Since we implement linear projections in multiple places, we elected to only use the simpler (but more expensive) random dense linear projection.

As mentioned previously, we perform most of our experiments on MNIST due to the increased cost. Some initial tests were performed on a simple generated spiral binary classification dataset. We did not possess the ability to train on larger datasets such as CIFAR-10 nor ImageNet.

As we are not looking to achieve the best performance, but rather compare the performance of various projection types against the non-projection models, we do not perform pre-processing (e.g., data augmentation) on any of the datasets. This results in our model having poorer performance than reported in Li's paper using the same architecture, dataset (MNIST), intrinsic dimension, and other hyperparameters (e.g., initialization and regularization parameters); as a result, we create and present our own baselines using the proposed architecture.

4 Experiments

4.1 Direct vs. linear projection models

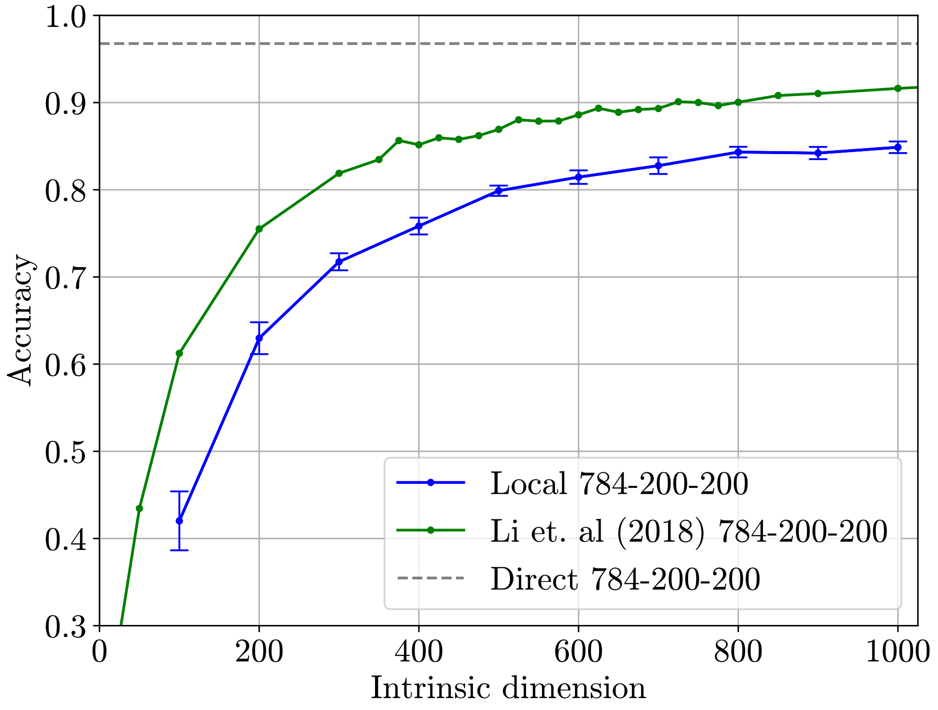


Figure 1: Comparison of our local MNIST implementation to Li et al. (2018). Both models are 784-200-200 FC models with default L2 regularization parameter $\lambda = 0.001$. Each dot for our local model represents the mean of eight trained models, and the error bars represent one standard deviation from the mean. We do not perform input data augmentation, which we believe is the most significant cause of the discrepancies in the accuracy. Li et al. data from `https://github.com/uber-research/intrinsic-dimension/blob/master/intrinsic_dim/plots/main_plots.ipynb`.

The 784-200-200 model was recreated directly using the standard Keras layers, and through projection using our own custom Keras layers. While we were able to achieve a similarly high result using the direct model (with test accuracy of 96.77%, which is not much lower than the reported test accuracy of 98.47% using the direct model in Li et al. (2018)), we were not able to recreate the 90% accuracy intrinsic dimension estimate of $d_{int,90} \approx 800$ for MNIST. For most of the intrinsic value dimensions we tested, we achieve roughly 10% lower accuracy for the same number of intrinsic dimensions. We speculate that this mostly is due to a lack of input image augmentation, but warrants further experimentation.

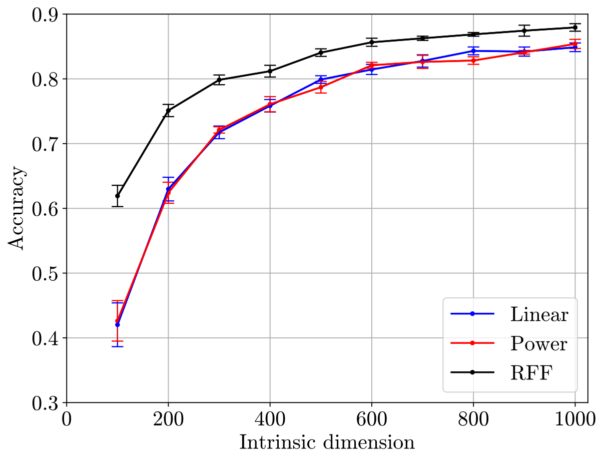
Our goal in these experiments is to achieve a (relatively) higher accuracy with a fixed intrinsic dimensionality, not to achieve the highest (absolute) accuracy or to recreate the results in Li et al. (2018) exactly. As a result, we use our results as-is and predict that our general results still hold in a relative sense. Given that accuracy as a function of intrinsic dimension is monotonically increasing, our result is equivalent to finding a lower intrinsic dimension given a fixed accuracy goal: i.e., attempting to minimize $d_{int,90}$.

For most of our models, we train with intrinsic dimensions $d = 100, 200, \ldots, 1000$, a default learning rate of $\alpha = 0.001$, and L2 regularization for both the FC layers' kernels and biases with $\lambda = 0.001$. The latter two parameters appeared to have a large impact: setting a higher learning rate (e.g., $\alpha = 0.01$) quickly causes a catastrophic drop in accuracy due failure to converge. In the case of augmenting the intrinsic weights with power terms, even $\alpha = 0.001$ sometimes does not converge for the larger d values – this is shown in some of the later results. We use the same regularization parameter value as Li; removing regularization empirically causes a slight decrease in test accuracy.

We follow Li's example and use `he_normal` to initialize $\theta_0^{(D)}$ and `random_normal` to initialize P. We initialize the intrinsic weights using `random_normal` rather than `zeros`. These choices are somewhat arbitrary and can be refined using a hyperparameter search.

4.2 Augmenting linear projection with power and RFF terms

Figure 2 displays the accuracy of the various projection models, as described in the Methods section, for intrinsic dimensions 100 through 1000. We observe a considerable difference between the accuracy of the power-terms-augmented model and the ordinary linear projection model.



(a)

d	Linear	Accuracy Power	RFF
100	0.42 ± 0.03	0.43 ± 0.02	0.62 ± 0.02
200	0.63 ± 0.02	0.624 ± 0.009	0.751 ± 0.009
300	0.72 ± 0.01	0.721 ± 0.008	0.798 ± 0.008
400	0.758 ± 0.009	0.761 ± 0.009	0.812 ± 0.009
500	0.799 ± 0.006	0.787 ± 0.006	0.841 ± 0.006
600	0.815 ± 0.008	0.821 ± 0.006	0.857 ± 0.006
700	0.83 ± 0.01	0.826 ± 0.003	0.863 ± 0.003
800	0.843 ± 0.006	0.828 ± 0.003	0.869 ± 0.003
900	0.842 ± 0.007	0.841 ± 0.008	0.874 ± 0.008
1000	0.849 ± 0.007	0.854 ± 0.006	0.880 ± 0.006

(b)

Figure 2: Comparison between projection types and accuracy vs. intrinsic dimensions. The linear and power-terms-augmented models perform roughly equally for all intrinsic dimensionalities, and the RFF-augmented model clearly outperform the other two methods for all intrinsic dimensionalities. Each dot in 2a represents the mean of eight trained models, and the error bars represent one standard deviation from the mean. 2b represents the same data in tabular form.

It is clear from the figure that augmentation with power terms has no meaningful effect on test accuracy when compared to the linear projection model, and the RFF-augmented model has a small but notable gain in accuracy for every value of intrinsic dimension.

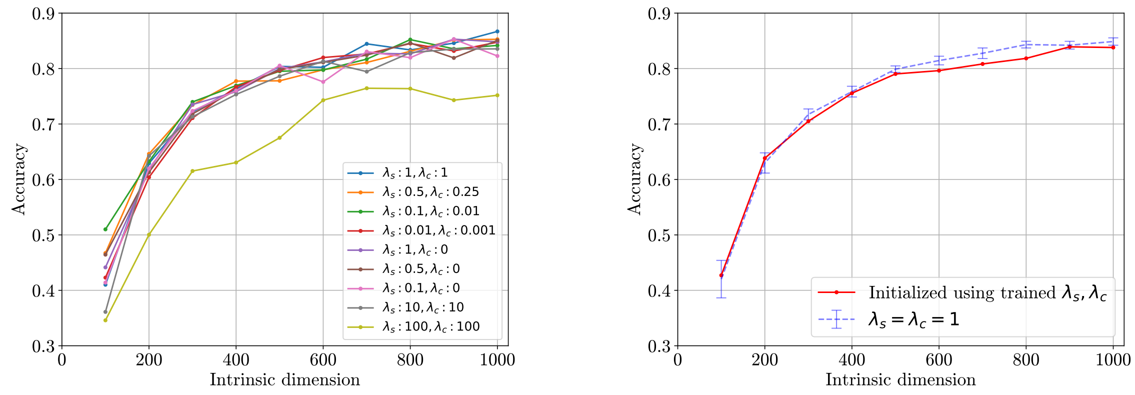
As stated in the methods section, we imagined that adding power terms might improve performance by being able to capture more complex relationships between intrinsic weights, and we were not sure what effect random Fourier feature mappings would have, having lost the intuitive geometric interpretation that can be found in the inverse rendering problem. In hindsight, after completing experiments that determine the distribution of the trained weights, we have a different hypothesis to explain our results. The results of those experiments are shown in the following section.

We found that the distribution of the intrinsic weights tend to be very narrow $\sigma_{\theta^{(d)}} \ll 1$; i.e. most of the trained intrinsic weights tend to converge towards very small numbers. As a result, the linear terms greatly dominate the squared and cubed terms when augmenting using power terms, and they provide almost no contribution. On the other hand, since the cosine function non-linearly maps small arguments (small intrinsic weights) to values near unity, there is a change in variance from the intrinsic weights matrix to its augmented form, which may allow for more expressivity in the resulting $\theta^{(D)}$. It would be interesting to follow up this research with a mathematical examination of this theory, which is something outside the knowledge scope of the current paper's authors.

4.3 Different coefficients for linear, squared, and cubed terms

After training an initial set of models as shown in the previous section, we attempt to improve the unremarkable performance of the models augmented by power terms. The results of this section give rise to the proposed explanation of the empirical results given in the previous section.

In our formulation of augmenting with power terms (5), we first began by choosing the projection weights $\{a_{ik}\}$, $\{b_{ik}\}$, and $\{c_{ik}\}$ from the same Normal distribution. Our intuition was that the squared and cubed terms might greatly dominate the linear terms or end up having negligible effect, so we experimented with setting the standard deviation of the weights in the projection matrix for the squared and cubed terms to be much smaller than those for the linear terms. We define λ_s and λ_c to be scaling factors for the standard deviation of the random Normal initializer used for the squared and cubed weights in the projection matrix (these are normalized to the standard deviation of the projection matrix weights for the linear, i.e., $\lambda_l = 1$). The results are shown in Figure 3a. There is no noticeable difference between any of these training runs, except for very large coefficients, which only harm accuracy.



(a) First attempt: Arbitrarily choosing coefficients for projection matrix weights. The accuracy is mostly roughly equal between the models, except when making the coefficients λ_s and λ_c very large.

(b) Second attempt: Using the parameters of the projection matrix weights from the networks trained with a trainable projection variable. This provides no accuracy gain over the models in 3a.

Figure 3: Attempts at improving the performance of power-terms-augmented models via different initialization of projection weights for linear, squared, and cubed terms.

However, we felt that this method of choosing the coefficients λ_s and λ_c was very arbitrary and not robust at all. Our second attempt involves allowing the matrix P to be trainable, and thus estimating the values λ_s and λ_c from the trained distributions in P. Sample distributions of the projection matrix weights are shown in Figure 4 for $d \in \{100, 500, 1000\}$. These generated λ_s and λ_c values were then used to initialize the P matrices, and all of the models were retrained. The result of this experiment is shown in Figure 3b. There are a number of noteworthy details. First,

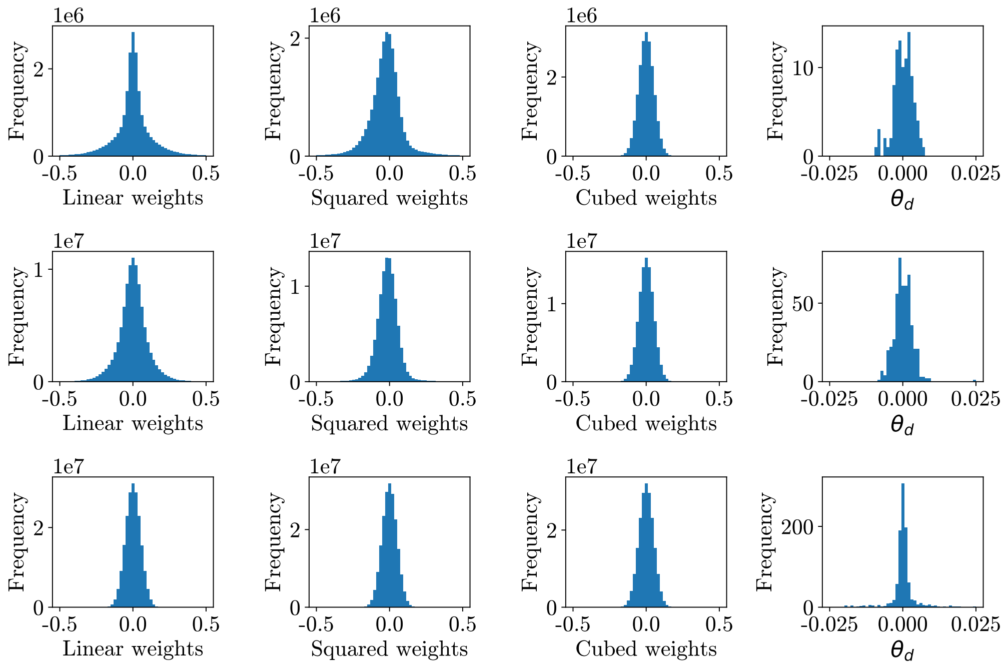


Figure 4: The distributions of weights in the projection matrix for linear, squared, and cubed weights, generated from training with a trainable projection matrix, as well as the distribution of intrinsic weights. The first row is for $d = 100$, the second for $d = 500$, and the last for $d = 1000$.

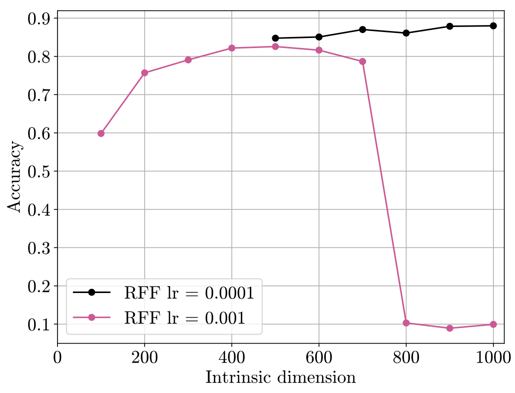
we notice that the distributions of P are indeed different between the linear, quadratic, and cubic terms; as expected, the coefficients of the cubic terms are generally more closely centered at zero ($\lambda_c < \lambda_l, \lambda_s$). However, what we failed to realize is that the distribution of trained intrinsic weights is also very close to zero, which makes the squared and cubic terms even less significant – this results in the contributions to $\theta^{(D)}$ coming largely from the linear terms in the linear combination. As a result of these two facts, the test accuracies resulting from the trained λ_s and λ_c initializers in Figure 3b show no notable difference from the other power-term-augmented models or the linear projection model.

Using this intuition, we tried increasing the initializer coefficients for the squared and cubed terms rather than decreasing them. With $\lambda_s = \lambda_c = 10$, the results are not considerably different from the other power-term-augmented models. With $\lambda_s = \lambda_c = 100$, then we start to see a breakdown of the accuracy.

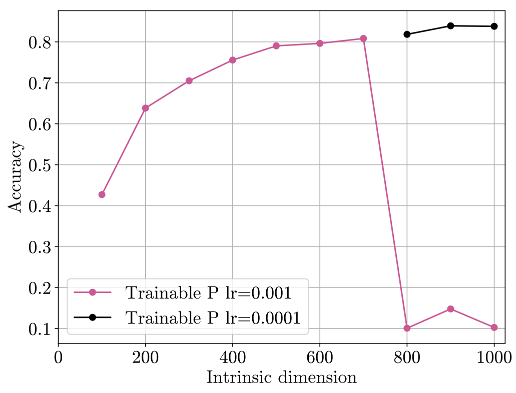
This also has an implication for the RFF results. Looking at the intrinsic weight distributions in Figure 4, we note that the variance of the intrinsic weights (the right-most column) is a decreasing function of d. If the variance to the (smooth) nonlinear mappings is small, then the nonlinear sine and cosine functions will be approximately linear. In other words, the smaller variance at high d will reduce the nonlinearity of the mapping; this may explain why the RFF was more advantageous for small d.

4.4 Learning rate problem

One of the general concerns with adding more operations to a network is that the added complexity may affect convergence. In particular, multiplication by a very large matrix P that multiplies the total weights in the network by a factor of d (during training) adds a large, undesirable complexity to the model; additionally, we are calculating the more unstable gradients of nonlinear functions. We



(a) RFF model



(b) Trainable projection matrix for learning initialization distributions for projection weights

Figure 5: Training accuracy of the two models with a higher ($\alpha = 0.001$) and lower learning rate ($\alpha = 0.0001$). Failure to converge at higher complexities (higher intrinsic dimensions) requires a smaller learning rate.

encountered convergence issues several times during training when the model for higher intrinsic dimensions, such as in the case of the random Fourier features (Figure 5a) and in the case of a trainable projection matrix (Figure 5b). In both cases, decreasing the default learning rate by a factor of 10 ($\alpha \leftarrow 0.0001$) fixed the convergence issue. This learning rate convergence issue does not occur for the direct model for similar learning rates.

A different interpretation for this effect is that by augmenting the intrinsic weights matrix with nonlinear mappings of itself, the gradient updates for each intrinsic weight $\theta_i^{(d)}$ become more complicated as it has more dependencies in the computational graph, which can be thought of loosely as “constraints” during backpropagation. Trying to meet three times as many contraints for either augmented model may be a cause for slower convergence and failure to converge at higher intrinsic dimensions.

Given that MNIST is a small dataset, this is a concerning issue that should be experimented on with larger datasets. We are unsure if this problem is still present if sparse or Fastfood linear projections are used in the place of dense linear projections.

4.5 Normalizing projection output basis vectors

The projection matrix P can be interpreted as the orthogonal basis of an output space. Li et al. describe normalizing the columns of P to form an orthonormal basis to make the linear projection isometric (i.e., distance-preserving).

We make the case that this distance preservation, and thus the normalization, will have little effect on accuracy. The reasoning for this is that a set of random basis vectors should have roughly equal length, and thus this normalization is roughly equivalent to scaling each column of P down by some constant average length. This will then only have the effect of scaling P’s variance down, which is inconsequential to a linear operation such as projection (it can be counteracted by learning the intrinsic weights to be larger by an equal factor).

In a departure from the procedure described in Li et al. (2018), we began by not normalizing the columns of the projection matrix, opting to run the linear, power, and RFF projections with the randomly generated data. We then tried normalizing the columns as suggested in the paper, leading to the results shown in (Figure 6). There is no noticeable difference between these results and the results when the projection matrix is not normalized, confirming our hypothesis that the normalization step is inconsequential.

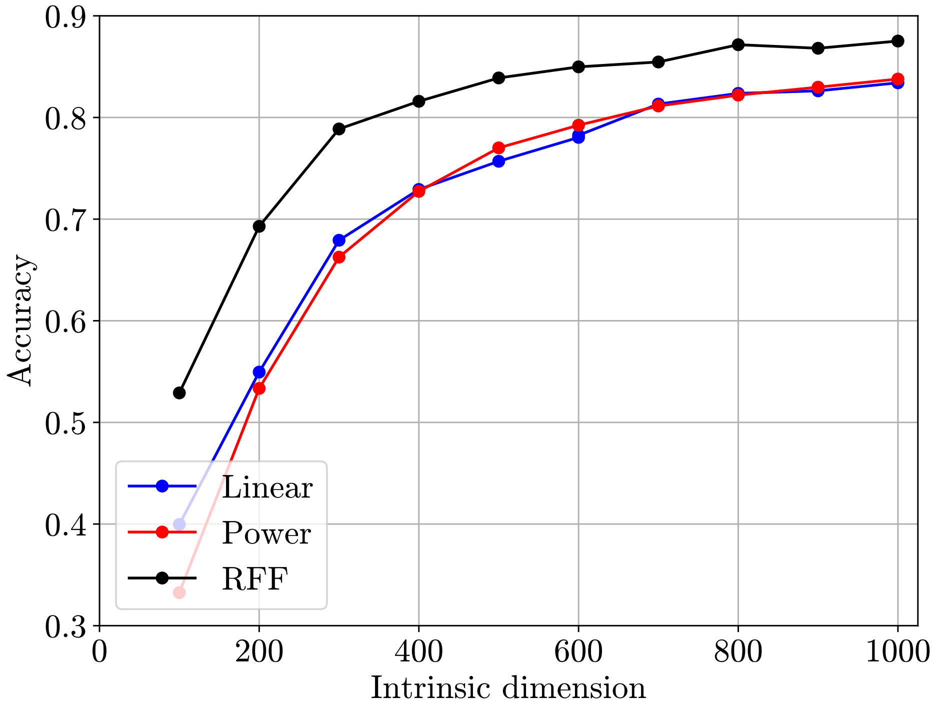


Figure 6: Accuracy vs. intrinsic dimensions for the projection types when the projection matrices are normalized. This is no appreciable difference from the results with a non-normalized projection matrix.

5 Conclusions and further research

We experimented with augmenting the method of intrinsic weights proposed in Li et al. (2018) by concatenating nonlinear-mapped features to the set of intrinsic weights. We use power terms (square and cubed terms) and a random Fourier feature mapping for the nonlinear mappings. Despite various attempts at improving the initialization of the projection matrix for the power terms, they did not considerably improve (or worsen) the accuracy as compared to the random linear projection. The RFF-augmented model does provide a noticeable improvement of 3-10% accuracy for all tested intrinsic dimensions. We attribute the improvement gain of the RFF to the fact that the sinusoids (especially the cosine function) increase variance in the small intrinsic weights, and the insignificant effect of the power terms to the fact that squaring or cubing the small weights results in inconsequential terms.

While most of Li et al.'s paper and this paper focus on the theoretical aspects of intrinsic dimension, some practical concerns should be raised. First of all, this method increases computational and memory complexity, especially in the case of dense projection matrices (as is the case of our experiments); while this may be mitigated using the Fastfood transform or sparse projection matrices, there is still a significant overhead over the underlying network architecture. Additionally, there is the issue of bad convergence on higher intrinsic dimensions. These factors make it a poor choice of a methodology for any practical deployment on resource-limited systems, and more of a theoretical toy aimed at comparing the difficulties of problems (e.g., comparing CIFAR-10 to MNIST) and architecture types (e.g., FC versus convolutional networks for image classification). However, it may be interesting to attempt to combine this with some existing compression technique, i.e., to find a way in which the intrinsic dimension can directly influence some aspect of model design.

There are many additional directions in which one could explore. For example, we limited ourselves to a single dataset and network architecture for the sake of simplicity, reproducibility, and practically completing the research in the given timeframe. It may be informative to test if these results extend to similar problems (e.g., CIFAR-10), different problems (e.g., generative models), much larger architectures, convolutional architectures, recurrent networks, residual networks (i.e., with skip connections), etc. It may also be interesting to use this method to "probe" network and layer architectures: one might intuit that the "factored convolution" used in the MobileNets architecture from Howard et al. (2017) might contain much less redundant layer than a corresponding ordinary convolution, and thus only able to represent problems with a lower intrinsic dimension; using this intrinsic dimension method may be a way to confirm this suspicion.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017. URL `http://arxiv.org/abs/1710.09282`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL `http://arxiv.org/abs/1502.01852`.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

Did (https://math.stackexchange.com/users/6179/did). Why are randomly drawn vectors nearly perpendicular in high dimensions. Mathematics Stack Exchange, 2018. URL `https://math.stackexchange.com/q/995678`. URL:https://math.stackexchange.com/q/995678 (version: 2018-05-15).

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.

Zhu Li, Jean-Francois Ton, Dino Oglic, and Dino Sejdinovic. Towards a unified analysis of random fourier features, 2019.

Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pp. 1177–1184, Red Hook, NY, USA, 2007. Curran Associates Inc. ISBN 9781605603520.

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains, 2020.

Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Ozlem Aslan, Shengjie Wang, Rich Caruana, Abdelrahman Mohamed, Matthai Philipose, and Matt Richardson. Do deep convolutional nets really need to be deep and convolutional?, 2017.

A Appendix: Notes on implementation

A.1 Libraries, hardware, and source code

The Tensorflow Keras library is used for general deep learning utilities. For each model, we create an instance of the `IntrinsicWeights` class, which is a semantic wrapper around a trainable `tf.Variable` instance. We then implement a `WeightCreator` class, which has utilities to return the calculated weights $\theta^{(D)}$ given a set of intrinsic weights and a projection type. This in turn is used in custom Keras layers, which are implemented very similarly to the ordinary non-projection ("direct") layers in the `tf.keras.layers` package, with the exception that the `add_weight` method is overloaded to use generate weights using a `WeightCreator` instance.

Experiments were performed on the Cooper Union Kahan deep learning server (primarily using GTX Titan X's). Roughly 200 GPU-hours of training were used for the experiments in this paper. The source code for our experiments can be found at `https://github.com/jlam55555/intrinsic-dimension-projections`, which was in turn heavily based on the source code from Li's paper, which can be found at `https://github.com/uber-research/intrinsic-dimension`.

A.2 Difficulties with the Tensorflow library

The first step in experimentation was updating the software implementation used in Li et al. (2018) to a more modern stack. (Their implementation uses Tensorflow 1 and Python 2.7; we attempt to upgrade the stack to Tensorflow 2 and Python 3.7.) The main reason for reimplementing the entire research was to gain a better understanding of the methods, and to try to independently reproduce the original results; a secondary reason is that Python 2.7 is soon to be deprecated, and Tensorflow 2 is more modern and offers a very different paradigm to similar tasks than Tensorflow 1.

There were a number of issues regarding the upgrade. In particular, TF2 uses eager evaluation by default, which is not the case with TF1. Since the weights $\theta^{(D)}$ is based on a calculation (and thus $\theta^{(D)}$ is a computational graph node rather than a leaf `tf.Variable` instance), this is actually easier to perform in the older version of Tensorflow. We had to explicitly declare the delayed computation of $\theta^{(D)}$ using `tf.Variable` instances, which is bulkier than passing around the computation tensor itself.

Another issue we encountered was that it appears that the TF2 Keras API requires that all trainable weights (all trainable `tf.Variable` instances) be instance members of the layer class; otherwise, even if they are part of the computational graph for the loss, they do not receive the gradient update rule. This results in some messier code.

We encountered an out-of-memory (OOM) error when running any model in a training loop, which appears to be an unfixed error in the Tensorflow library due to model VRAM not being garbage collected. A workaround that worked for us[2] was to run models in separate processes; the process clean-up sequence appears to successfully handle the memory freeing.

A.3 Similarities to and differences from the original research

Following the conventions of the original paper (and of linear algebra), we denoted the projection matrix as a $D \times d$ matrix, and the intrinsic weights $\theta^{(d)}$ and full network weights $\theta^{(D)}$ as length D and length d column vectors, respectively. However, following the implementation in their codebase, the projection is a $d \times D$ matrix, the weights are row vectors, and the order of the matrix multiplication is swapped. There is no functional difference here, but it may be confusing at first sight.

We had difficulty implementing the sparse and Fastfood implementations that were used in the original paper, and thus chose to omit them from any of our final results. With more time, we would hope to be able to include these in our analysis.

[2] `https://github.com/tensorflow/tensorflow/issues/36465#issuecomment-582749350`