What Can Transformers Learn In-Context? A Case Study of Simple Function Classes

Shivam Garg* **Dimitris Tsipras*** **Percy Liang** **Gregory Valiant**

Stanford University
{shivamg, tsipras, pliang, gvaliant}@cs.stanford.edu

Abstract

In-context learning refers to the ability of a model to condition on a prompt sequence consisting of in-context examples (input-output pairs from some task) along with a new query input, and generate the corresponding output. Crucially, in-context learning happens only at inference time without any parameter updates to the model. While large language models such as GPT-3 exhibit some ability to perform in-context learning, it is unclear what the relationship is between tasks on which this succeeds and what is present in the training data. To make progress towards understanding in-context learning, we consider the well-defined problem of training a model to in-context learn a function class (e.g., linear functions): given data derived from some functions in the class, can we train a model to in-context learn "most" functions from this class? We show empirically that standard Transformers can be trained from scratch to perform in-context learning of linear functions—that is, the trained model is able to learn unseen linear functions from in-context examples with performance comparable to the optimal least squares estimator. In fact, in-context learning is possible even under two forms of distribution shift: (i) between the training data of the model and inference-time prompts; (ii) between the in-context examples and the query input during inference. We also show that we can train Transformers to in-context learn more complex function classes—i.e., sparse linear functions, two-layer neural networks, and decision trees—with performance that matches or exceeds task-specific learning algorithms.[1]

1 Introduction

Large language models such as GPT-3 [10] are able to perform *in-context learning*: given a prompt containing examples from a task (input-output pairs) and a new query input, the language model can generate the corresponding output. For example, these models are able to produce English translations of French words after being prompted on a few such translations, e.g.:

$$\underbrace{\text{maison} \rightarrow \text{house},\ \text{chat} \rightarrow \text{cat},\ \text{chien} \rightarrow}_{\text{prompt}}\ \underbrace{\text{dog}}_{\text{completion}}\ .$$

This capability is quite intriguing as it allows models to adapt to a wide range of downstream tasks on-the-fly—without the need to update the model after training [10, 33, 53, 8]. However, it is unclear to what extent these models are able to learn *new tasks* from in-context examples alone as opposed to indexing into a vast set of known tasks from the training data (e.g., see Min et al. [39]). [2]

*Equal contribution.

[1] Our code and models are available at `https://github.com/dtsip/in-context-learning`.

[2] The term "in-context learning" may also refer to a more general notion of learning from a prompt [46]. In this work, we focus on the standard notion which refers to learning a task/function given in-context examples [10].

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

To make progress towards understanding in-context learning, we consider the well-defined problem of learning a *function class* from in-context examples. That is, we say that a model can in-context learn a function class $\mathcal{F}$ if, for "most" functions $f \in \mathcal{F}$, the model can approximate $f(x_{\text{query}})$ for a new query input x_{query} by conditioning on a prompt sequence $(x_1, f(x_1), \ldots, x_k, f(x_k), x_{\text{query}})$ containing in-context examples and the query input.

Formally, let $D_{\mathcal{X}}$ be a distribution over inputs and $D_{\mathcal{F}}$ be a distribution over functions in $\mathcal{F}$. A prompt P is a sequence $(x_1, f(x_1), \ldots, x_k, f(x_k), x_{\text{query}})$ where inputs (i.e., x_i and x_{query}) are drawn i.i.d. from $D_{\mathcal{X}}$ and f is drawn from $D_{\mathcal{F}}$. We say that a model M can in-context learn the function class $\mathcal{F}$ up to ϵ, with respect to $(D_{\mathcal{F}}, D_{\mathcal{X}})$, if it can predict $f(x_{\text{query}})$ with an average error

$$\mathbb{E}_P \left[\ell \left(M \left(P \right), f \left(x_{\text{query}} \right) \right) \right] \leq \epsilon, \tag{1}$$

where $\ell(\cdot, \cdot)$ is some appropriate loss function, such as the squared error. Note that, here, being able to in-context learn a function class is a property of model M alone, independent of how it was trained.

Within this framework, we can now concretely ask:

Can we train a model to in-context learn a certain function class?

We empirically study this question, focusing on Transformer models [66, 51]—the architecture behind recent large language models—trained from scratch to in-context learn simple, well-defined function classes (e.g. linear functions). Specifically, we sample prompts containing in-context examples (input-output pairs) generated using functions in a given class and train models to predict the function value at the corresponding query inputs. (see illustration in Figure 1). Our findings are as follows.

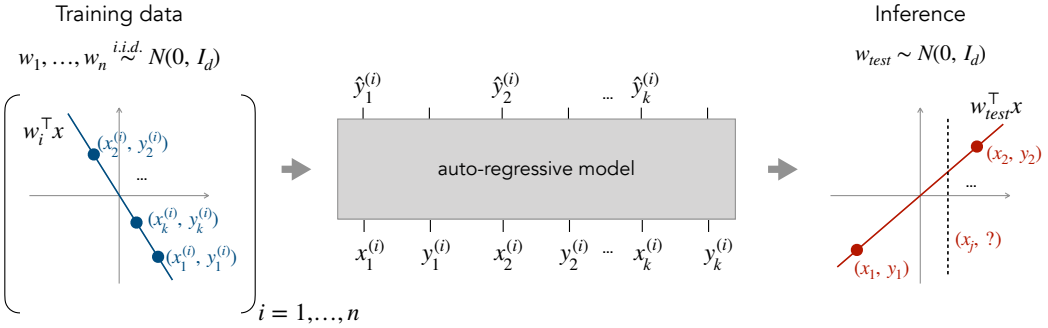


Figure 1: *Can we train a model that in-context learns a function class (here linear functions)?* We train Transformers by repeatedly sampling a random function f from that class, as well as random inputs $x_1, \ldots, x_k$ and training the model to predict each $f(x_i)$ given the prompt $x_1, f(x_1), \ldots, x_{i-1}, f(x_{i-1}), x_i$ (wrt squared loss). Then, during inference, we evaluate the model's ability to predict accurately on new, *unseen* functions.

Transformers can in-context learn linear functions. We show empirically that we can train a standard Transformer from scratch to in-context learn the class of linear functions, with respect to the input distribution $D_{\mathcal{X}}$ being an isotropic Gaussian in 20 dimensions, and $D_{\mathcal{F}}$ being the distribution over linear functions with weight vectors drawn from an isotropic Gaussian (the model was trained on prompts generated from the same distributions $D_{\mathcal{X}}$ and $D_{\mathcal{F}}$). Specifically, the trained model achieves error comparable to the optimal least squares estimator, suggesting that it encodes an effective learning algorithm, at least for the distribution used to generate the training prompts.

Generalization to out-of-distribution prompts. To understand the extent to which the trained model encodes an algorithm that works beyond the training distribution, we consider in-context learning under two types of distribution shifts: (a) a shift between the prompts encountered during training and inference (e.g., training on prompts without any noise in the in-context example outputs but testing with noisy outputs), (b) a shift between the in-context examples and the query input during inference (e.g., in-context examples lie in one orthant and the query input lies in another). We find that the performance of our model is quite robust to such shifts, indicating that it has learned to perform linear regression to some generality.

More complex function classes. We also consider the function classes of 3-sparse linear functions, two-layer ReLU neural networks with 100 hidden units, and decision trees of depth 4, all with

20 dimensional inputs. We show that we can train Transformer models that can in-context learn these classes (with respect to isotropic Gaussian inputs and appropriately defined distributions over functions). For sparse linear functions, the trained model exploits sparsity, obtaining performance better than least squares and comparable to Lasso. For neural networks, the trained model performs comparably to neural networks of the same architecture trained using gradient descent on in-context examples. Moreover, it is also able to in-context learn linear functions. For decision trees, the trained model learns unseen trees with as few as 100 in-context examples, whereas greedy learning and tree boosting algorithms are unable to achieve competitive performance. Note that learning these function classes requires involved algorithms (e.g., gradient descent with the Lasso objective), and our results suggest that Transformers can both represent and discover such algorithms.

Model capacity and problem dimension. Finally, we explore how the ability of Transformers to in-context learn linear functions scales with model capacity and problem dimensionality. We find that increasing model capacity improves performance significantly, and allows the model to in-context learn higher-dimensional functions. Also, increasing the capacity often improves performance under distribution shifts significantly, even when the absolute improvement in the standard error is small.

2 Training models for in-context learning

We now describe a general methodology for training a model that can in-context learn a function class $\mathcal{F}$ with respect to a distribution $D_{\mathcal{F}}$ over functions, and $D_{\mathcal{X}}$ over inputs. We start by constructing random training prompts as follows. We sample a random function f from the class according to $D_{\mathcal{F}}$, then create a set of random inputs $x_1, \ldots, x_{k+1}$ drawn independently from $D_{\mathcal{X}}$, and evaluate f on these inputs to produce the prompt $P = (x_1, f(x_1), \ldots, x_{k+1}, f(x_{k+1}))$. For example, in the case of linear functions, inputs could be drawn from the isotropic Gaussian distribution $N(0, I_d)$, and a random function chosen by sampling weight vector w from $N(0, I_d)$ and setting $f(x) = w^\top x$.

Now, given such prompts, we train a model to predict $f(x_i)$ for a given x_i based on a set of preceding in-context examples. Concretely, let P^i denote the prompt prefix containing i in-context examples (the first i input-output pairs) and the $(i+1)^{\text{th}}$ input: $P^i = (x_1, f(x_1), x_2, f(x_2), \ldots, x_i, f(x_i), x_{i+1})$. Then, we train a model M_θ parameterized by θ aiming to minimize the expected loss over all the prompt prefixes:

$$\min_\theta \ \mathbb{E}_P \left[\frac{1}{k+1} \sum_{i=0}^{k} \ell \left(M_\theta \left(P^i \right), f\left(x_{i+1} \right) \right) \right], \tag{2}$$

where $\ell(\cdot, \cdot)$ is an appropriately chosen loss function. Below, we describe how this general methodology can be implemented for a concrete model family (see Appendix A for additional details).

Model structure. We use a decoder-only Transformer architecture [66] from the GPT-2 family [52], with 12 layers, 8 attention heads, and a 256-dimensional embedding space (22.4M parameters). The model takes as input a sequence of vectors in its embedding space and predicts the next vector in the sequence within the same space (in language modeling, these vectors correspond to input tokens). We apply this model to our prompts $(x_1, f(x_1), \ldots, x_{k+1}, f(x_{k+1}))$ as follows. We map each prompt output $f(x_i)$ to the same dimension as prompt inputs x_i by appending zeros, and map the prompt inputs and outputs into the latent embedding space of the Transformer through a (learnable) linear transformation. We then use another (learnable) linear transformation to map the vector predicted by the model to a scalar. Note that the Transformer architecture allows us to compute the prediction $(M_\theta(P^i))$ for all prompt prefixes in a single forward pass.

Training. We train the model according to the training objective in (2) using squared error as the loss function. We sample a batch of random prompts at each training step and update the model through a gradient update. We use a batch size of 64 and train for 500k steps. This training is done from scratch, that is, we do *not* fine-tune a pre-trained language model, nor do we train on actual text.

Curriculum learning. Many function classes contain functions of varying complexity. We exploit this by training our model using a curriculum [5, 20, 58, 70], where we train on a simpler distribution of functions in the beginning (e.g., linear functions with weight vectors restricted to a low-dimensional subspace) and gradually increase the function complexity. This speeds up training drastically, often allowing us to train models that would be significantly more expensive to train without a curriculum (see Section 6 for details).

3 In-context learning of linear functions

In the previous section, we describe a general methodology for training Transformer models to in-context learn a class of functions. Here, we focus on a simple function class—namely linear functions—and study how well models trained using our methodology can in-context learn this class.

Prompt distribution. We consider the class of linear functions $\mathcal{F} = \left\{ f \mid f(x) = w^\top x,\ w \in \mathbb{R}^d \right\}$, in d dimensions where $d = 20$. We sample $x_1, \ldots, x_k$, x_{query}, and w independently from the isotropic Gaussian distribution $N(0, I_d)$. We then compute each $y_i = w^\top x_i$ and construct the prompt as $P = (x_1,\ y_1,\ x_2,\ y_2, \ldots,\ x_k,\ y_k,\ x_{\text{query}})$.

Baselines. To contextualize the performance of our trained model, we compare it to other learning algorithms: (a) the least squares estimator, computing the minimum-norm linear fit to the in-context examples $(x_i,\ y_i)$, (b) n-Nearest Neighbors, averaging the y_i values for the n nearest neighbors of x_{query}, (c) averaging the values $y_i x_i$ to estimate w and compute the inner product of this estimate with x_{query}. Least squares is the optimal estimator for this problem and thus serves as a lower bound. The other baselines are consistent (but sub-optimal) estimators that are easier to compute and thus provide an estimate of the performance achieved by simple approaches. See Appendix A.3 for more details.

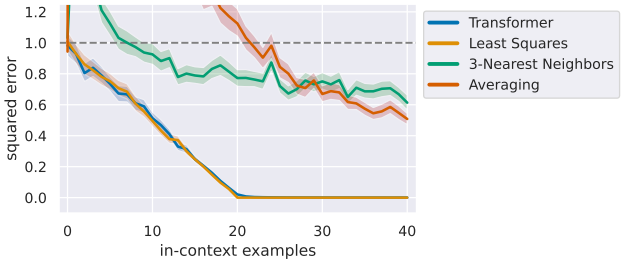


Figure 2: *Evaluating a trained Transformer on in-context learning linear functions.* We plot the normalized squared error of the model $((M(P) - w^\top x_{\text{query}})^2/d)$ and the relevant baselines, as a function of the number of in-context examples. The error of the Transformer decreases at a rate comparable to least squares. When the number of in-context examples reaches the problem dimension d (here 20), least squares achieves 0 error while our model achieves an error of 0.02, improving to 0.0006 at $2d$ examples. The simple baselines perform better than the zero estimator (dashed), but still rather poorly. (Error averaged over 1280 prompts. 95% confidence intervals, 1000 bootstrap trials.)

3.1 Transformers can in-context learn linear functions

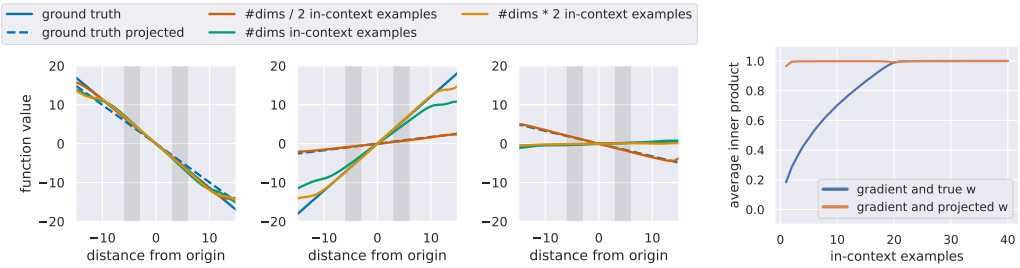
We show the in-context learning ability of the resulting model along with the relevant baselines in Figure 2. The trained model is able to in-context learn the class of linear functions with respect to the prompt distribution specified above, performing comparably to the optimal least squares estimator for any number of in-context examples considered. While the simpler baselines achieve non-trivial error, they are far from optimal, indicating that the trained model encodes a more complex algorithm. Moreover, in Appendix B.7, we show that the model cannot be relying on memorization of training prompts or weight vectors, and thus encodes an algorithm capable of in-context learning linear functions that are very different from those seen during training.

3.2 What functions is the model learning in-context?

Recall that the goal of our model is: given the prompt $P = (x_1,\ w^\top x_1, \ldots, x_k,\ w^\top x_k,\ x_{\text{query}})$, output $w^\top x_{\text{query}}$. Thus, if we fix the prefix given by the k in-context examples, we can view the output of the model as a function $\hat{f}_{w,x_{1:k}}(x_{\text{query}})$, that approximates $w^\top x_{\text{query}}$. When $k < d$ (fewer in-context examples than dimensions), the ground truth cannot be recovered perfectly and the ideal model should approximate $(\text{proj}_{x_{1:k}}(w))^\top x_{\text{query}}$, where $\text{proj}_{x_{1:k}}(w)$ is the projection of w onto the subspace spanned by $x_1, \ldots, x_k$. Here, we will evaluate how accurately the model approximates this.

Visualizing along a random direction. For a randomly sampled fixed prefix, we visualize $\hat{f}_{w,x_{1:k}}(x_{\text{query}})$ as we vary the query input along a random direction x (Figure 3a). That is, we

pick a random unit vector x, and evaluate $\hat{f}_{w,x_{1:k}}(\lambda x)$ as we vary λ, the distance of the query input from origin. We observe that $\hat{f}_{w,x_{1:d}}(\lambda x)$ and $\hat{f}_{w,x_{1:2d}}(\lambda x)$ closely match the ground truth and $\hat{f}_{w,x_{1:d/2}}(\lambda x)$ matches the projected ground truth, when the distance from origin is not too large compared to the norm of a typical randomly sampled input. In fact, in Appendix B.1, we show that the model is quite robust to scaling the query input: the error doesn't increase much as we scale up the query input by a factor of up to 2, or scale down by a factor of up to 16, and degrades slowly after.



(a) function visualizations (b) gradients

Figure 3: *Understanding the prefix-conditioned function.* (a) Model prediction as we fix the in-context examples and vary the query input along a random direction (three random prompts). The shaded region denotes where the norm of a randomly training input lies with probability 0.99. When the scale of the query input is close to this range, the model prediction is close to the true linear function (or its projection to the space of in-context inputs when $k < d$). (b) We compute the gradient of the model prediction with respect to the query input, and plot its (normalized) inner product with the true and projected w, averaged over 1280 prompts. The gradient aligns almost perfectly with w when $k \geq d$, and with projected w for all k, indicating that the model locally aligns with the ground truth.

Local correctness. So far, we have seen that the model is able to make predictions close to the ground truth for randomly drawn query inputs and in-context examples. We will now turn our attention to the local change of $\hat{f}$ around x_{query} by considering the gradient of the function $\hat{f}_{w,x_{1:k}}(x_{\text{query}})$ with respect to x_{query} (our model is fully differentiable so we can compute it directly). Since $\hat{f}$ computed by the model should ideally approximate $\text{proj}_{x_{1:k}}(w)^\top x$, this gradient should lie in the direction of the projected ground truth $\text{proj}_{x_{1:k}}(w)$. In Figure 3b, we show the inner product between the gradient and $\text{proj}_{x_{1:k}}(w)$ (both normalized), averaged over 1280 random prompts, and observe that they align almost perfectly. Since $\text{proj}_{x_{1:k}}(w) = w$ almost surely when $k \geq d$, we observe that the gradient also aligns with w perfectly in this regime. Thus the model is locally correct around the query input.

4 Extrapolating beyond the training distribution

In the previous section, we demonstrated that we can train a model to in-context learn linear functions with respect to the distribution of prompts encountered during training. That is, we evaluate the in-context learning ability of the model with respect to distributions $D_{\mathcal{X}}$ and $D_{\mathcal{F}}$ that were also used to train the model. Here, we evaluate the in-context learning performance of our model on prompt distributions different from the one used for training. Our overarching goal is to better understand the learning algorithm encoded by our model by analysing how it performs on such prompts.

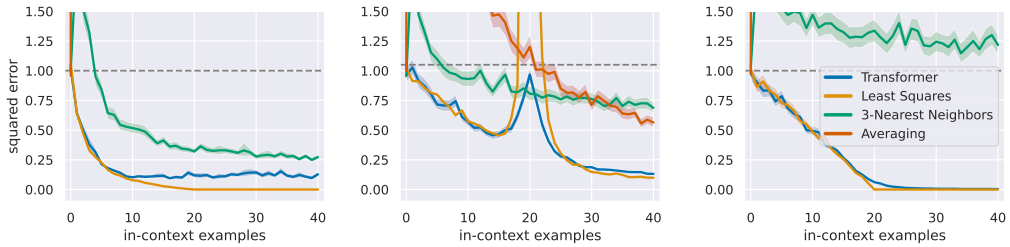
Formally, we refer to the distribution of functions used during training as $D_{\mathcal{F}}^{\text{train}}$ and the corresponding distribution of prompt inputs as $D_{\mathcal{X}}^{\text{train}}$. During inference, functions are sampled from a (potentially different) distribution $D_{\mathcal{F}}^{\text{test}}$, while prompt inputs from a distribution $D_{\mathcal{X}}^{\text{test}}$. Moreover, deviating again from our analysis so far, we also consider a separate distribution $D_{\text{query}}^{\text{test}}$, from which the query input is sampled, potentially dependent on the rest of the in-context inputs $x_1, \ldots, x_k$ (sampled from $D_{\mathcal{X}}^{\text{test}}$).

Within this framework, we consider the same model as last section, and evaluate its performance on prompts that deviate from those encountered during training, either by

1. sampling prompt inputs or functions from a different distribution, that is $D_*^{\text{train}} \neq D_*^{\text{test}}$ or
2. introducing a mismatch between in-context examples and query input, i.e., $D_{\text{query}}^{\text{test}} \neq D_{\mathcal{X}}^{\text{test}}$.

We describe three such prompt distributions below, along with the corresponding results in Figure 4, and provide the full results in Appendix B.2. Overall, the model performs reasonably accurate in-context learning with respect to these prompt distributions, indicating that it has indeed learnt to perform linear regression to some generality.

Recall that we generate a training prompt $P = (x_1, w^T x_1, \ldots, x_k, w^T x_k, x_{\text{query}})$ by drawing the prompt inputs (x_i and x_{query}), and the weight vector (w) i.i.d. from $N(0, I_d)$, with $d = 20$.



(a) Skewed covariance (b) Noisy linear regression (c) Different orthants

Figure 4: *In-context learning on out-of-distribution prompts.* We evaluate the model on prompts that deviate from those seen during training by: (a) sampling prompt inputs from a non-isotropic Gaussian, (b) adding label noise to in-context examples, (c) restricting in-context examples to a single (random) orthant. Model error degrades gracefully and remains close to that of the least squares estimator, indicating that its in-context learning ability extrapolates beyond the training distribution.

Skewed covariance. We sample prompt inputs from $N(0, \Sigma)$ where Σ is a skewed covariance matrix with eigenbasis chosen uniformly at random and i^{th} eigenvalue proportional to $1/i^2$. We normalize the inputs so that their expected squared norm is equal to that of inputs encountered during training, and study the effect of input scale separately in Appendix B.2. The model matches the performance of least squares until $k = 10$, mimicking the sharp drop in the error in this regime, but its error plateaus afterwards (see Figure 4a). Thus, it is not perfectly robust to this distribution mismatch but still does relatively well, achieving less than half the error of the nearest neighbor baseline in most cases.

Noisy linear regression. We add noise to each prompt output: the ith output is equal to $w^T x_i + \epsilon_i$ where $\epsilon_i \sim N(0, 1)$. The trained model closely tracks the performance of least squares for most prompt sizes (see Figure 4b). Interestingly, the model also exhibits the double descent error curve [2] that is known to manifest for the least squares estimator [44]. Note that in this noisy setting, the optimal estimator corresponds to least squares with appropriate ℓ_2-regularization, which we cannot expect the model to learn since it was trained on noiseless data.

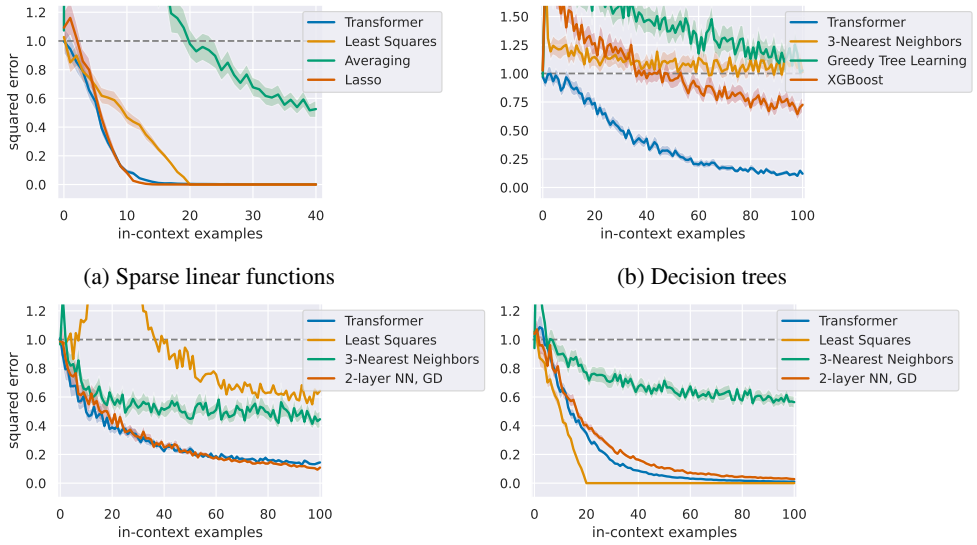
Different orthants for in-context and query inputs. We fix the sign of each coordinate to be positive or negative for all in-context inputs x_i (at random). As a result, all in-context inputs lie in the same orthant, while the query input lies in another orthant with high probability. The model is not affected by the mismatch between in-context and query inputs and closely match the performance of least squares. In this case, the model achieves errors 0.062 and 0.004 for 20 and 40 in-context examples respectively (see Figure 4c), whereas recall that it achieves errors 0.02 and 0.0006 on standard prompts. This indicates that the models is not relying on some variant of nearest neighbor search as in that case, its error would have been higher (see the 3-nearest neighbor baseline).

5 More complex function classes

We now turn our attention to in-context learning for more complex function classes, namely sparse linear functions, decision trees, and two-layer ReLU neural networks. Here, we are back in the setting where the distribution of prompts during inference is same as that during training. The overall methodology remains the same: we sample random functions from these families and train a model to approximate these functions given in-context examples. (See Appendix A.3 for details and baselines.)

Sparse linear functions. First, we consider functions of the form $f(x) = w^\top x$ where $w \in \mathbb{R}^d$ and has exactly s non-zero coordinates. To sample a prompt $P = (x_1, f(x_1), \ldots, x_k, f(x_k), x_{\text{query}})$, we draw prompt inputs x_i and x_{query}, and a weight vector w from $N(0, I_d)$, and then zero out all but s coordinates of w uniformly at random. In this setting, the least squares estimator is no longer optimal.

One can perform better by leveraging sparsity, e.g., using Lasso [65], which involves solving the least squares objective with an ℓ_1-norm weight regularizer. We plot the performance of our model trained for $d = 20$ and $s = 3$ in Figure 5a, and observe that it is also able to leverage sparsity, nearly matching the performance of Lasso. Note that, unlike least squares, Lasso does not have a closed form expression and involves iterative minimization of the regularized objective, yet the Transformer is able to achieve comparable performance in a single forward pass.



(a) Sparse linear functions

(b) Decision trees

(c) 2-layer NN

(d) 2-layer NN, eval on linear functions

Figure 5: *Training a Transformer to in-context learn more complex function classes.* (a) A Transformer trained on prompts generated using sparse linear functions can in-context learn this class, with error decreasing at a rate similar to Lasso, and significantly better than minimum norm least squares. (b) A Transformer trained on prompts generated using random decision trees can in-context learn this class, with much better performance than greedy tree learning or tree boosting. (c) A Transformer trained on prompts generated using random 2-layer ReLU neural networks can in-context learn this class. The error decreases at a rate similar to the baseline which involves training a neural network using a variant of gradient descent with in-context examples as the training data. (d) The same model (from (c)) can in-context learn the class of linear functions. The error decreases at a rate slower than least squares, but slightly better that a neural network trained using a variant of gradient descent. All errors are normalized so that the zero estimator achieves an error of 1 (dashed).

Decision trees. Next, we consider the class of depth 4 decision trees with 20 dimensional inputs. A function f in this class is represented by a full binary tree (with 16 leaf nodes) where each non-leaf node is associated with a coordinate, and each leaf node is associated with a target value. To evaluate f on an input x, we traverse the tree starting from the root node, and go to the right child if the coordinate associated with the current node is positive and go to the left child otherwise (that is, the threshold at each node is 0). $f(x)$ is given by the value associated with the leaf node reached at the end. To sample a random prompt $P = (x_1, f(x_1), \ldots, x_k, f(x_k), x_{\text{query}})$, we draw inputs x_is and x_{query} from $N(0, I_d)$, and f corresponds to a tree where the coordinates associated with the non-leaf nodes are drawn uniformly at random from $\{1, \ldots, d\}$ and the values associated with the leaf nodes are drawn from $N(0, 1)$. In Figure 5b, we show that Transformers can be trained to in-context learn this class much better than greedy tree learning and boosting (via XGBoost [13]).

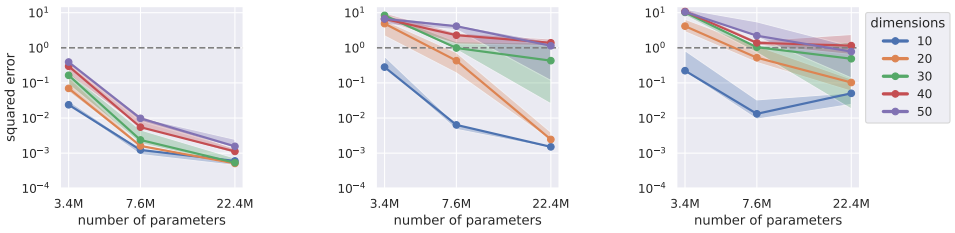
Two-layer neural networks. Finally, we consider the class of two layer ReLU neural networks containing functions of the form $f(x) = \sum_{i=1}^{r} \alpha_i \, \sigma(w_i^\top x)$, where $\alpha_i \in \mathbb{R}$, $w_i \in \mathbb{R}^d$ and $\sigma(\cdot) = \max(0, \cdot)$ is the ReLU activation function. To draw a random prompt $P = (x_1, f(x_1), \ldots, x_k, f(x_k), x_{\text{query}})$, we sample inputs x_is and x_{query} from $N(0, I_d)$, along with network parameters a_is and w_is from $N(0, 2/r)$ and $N(0, I_d)$ respectively. We set the input dimension d to 20 and the number of the hidden nodes r to 100. In Figure 5c, we show that Transformers can be trained to in-context learn this class of functions. In fact, the error of our model matches closely the performance of the baseline which trains a two-layer neural network of the same architecture on

in-context examples using (full-batch) Adam [30], a variant of gradient descent (see Appendix A.3 for details).Moreover, the model trained to in-context learn two-layer neural networks is also able to in-context learn linear functions (for which it is not explicitly trained), albeit at a rate slower than least squares, but slightly better than a neural network trained on in-context examples (see Figure 5d).

6 Investigating what matters for in-context learning

We now return to the setting of training models to in-context learn linear functions and explore different factors that lead to successful in-context learning.

Problem Dimension and Capacity. In Section 3 and 4, we saw that Transformer models can be trained to in-context learn 20-dimensional linear functions accurately and relatively robustly. To explore the interplay between problem dimensionality and capacity, we also consider models with fewer parameters (see Appendix A.1) and train each architecture on $\{10, 30, 40, 50\}$-dimensional problems. In Figure 6, we plot the model error with $2d$ in-context examples as we vary the problem dimension d and the model capacity. In the standard setting, i.e., when the training and inference time prompt distributions are the same, we observe that the error decreases as we increase the capacity or reduce the problem dimensionality (see Figure 6a)—i.e., model capacity helps. For out-of-distribution prompts, the settings with skewed covariance or with in-context example and query inputs lying in different orthants are challenging, especially for higher dimensional problems. However, the error decreases considerably (in most cases) as we increase the model capacity, even when absolute decrease in the standard error is small (see Figure 6b and 6c). See Appendix B.3 for additional plots.



(a) Standard (b) Different orthants (c) Skewed covariance

Figure 6: *Understanding the effect of model capacity and problem dimension on in-context learning performance for in-distribution (a) and out-of-distribution (b,c) prompts.* We train Transformers to in-context learn linear functions and plot the error with $2d$ in-context examples as we vary problem dimension d and model capacity. Capacity helps in most cases, especially on out-of-distribution prompts, even when the absolute gains in the in-distribution setting are small. We train 3 models in each case with different random seeds, and show the median error (solid lines), and the minimum and maximum errors (shaded region). (See Appendix B.4 for training variance analysis.)

Curriculum learning. When training our models, we initially draw the prompt inputs from a fixed 5 dimensional subspace (by setting some of the coordinates to 0) with prompt length 11 (number of input-output pairs), and increase the subspace dimension by 1 and prompt length by 2 every $2,000$ training steps, until the subspace dimension reaches the ambient dimension d and prompt length reaches $2d+1$ (see Appendix A.2 for details). This process can also be viewed as gradually increasing the complexity of the function class. This speeds up training drastically, especially for higher dimensional problems: for dimension 50, the loss barely decreases through the 500k training steps without curriculum but reaches close to the optimum with curriculum. For the 20 dimensional problem where we were able to train the model without curriculum within the training (step count) budget, we did not observe any qualitative difference in accuracy or robustness compared to the model trained with curriculum. We compare training with and without curriculum in Appendix B.5.

Notably, when training Transformers without curriculum, there is an initial—relatively long—period in training where the loss does not decrease, followed by a period of sharp decrease. The length of this period varies with training randomness and seems to increase on average with problem dimension. Understanding the model around this transition moment could give insights into the emergence of in-context learning. Interestingly, Olsson et al. [46] observe a similar jump in the in-context learning ability of a language model which they attribute to the formation of "induction heads".

Number of distinct prompts or functions seen during training. To estimate the amount of training data required for in-context learning, we perform two ablation studies. In the first study, we limit the number of distinct prompts seen during training. That is, we create a set of n_p randomly generated prompts (as described in Section 2), and sample prompts from this set during training (here, we train without curriculum, as it would introduce additional prompts during the warmup phase). In the second study, we only limit the number of distinct functions used for training. That is we create a set of n_w randomly chosen vectors (corresponding to n_w linear functions) and sample weight vectors uniformly from that set to generate the training prompts (the inputs are still sampled from $N(0, I_d)$ for each training prompt). We find that the amount of training data required is relatively small: non-trivial in-context learning is possible with $n_p = 100$k or $n_w = 1$k, and the error drops close to that of the unrestricted model (discussed in Section 3) with $n_p = 1$M or $n_w = 10$k (details in Appendix B.6). For context, in Section 3, the model is trained on fresh prompts at each step and thus encounters 32M distinct linear functions and prompts (500k training steps with a batch size of 64).

7 Related work

In-context learning. Since Brown et al. [10] demonstrated the in-context learning ability of GPT-3, there has been a significant interest in improving and understanding this capability [34, 37, 76, 36, 57, 38, 14, 41, 31]. The works most relevant to ours are as follows. Xie et al. [71] propose a Bayesian inference framework explaining how in-context learning works despite formatting differences between training and inference distributions. Razeghi et al. [55] show that in-context learning for numerical reasoning tasks is better for instances whose terms are more prevalent in training data. Min et al. [37] demonstrate tasks where in-context learning works even when the prompt outputs are chosen randomly, questioning to what extent these models are truly learning new tasks on-the-fly, while Rong [56] gives examples of novel tasks on which these models demonstrate on-the-fly learning ability. Chan et al. [12] demonstrate that distributional properties such as long-tailedness are crucial for in-context learning on an image-based few-shot dataset. Olsson et al. [46] and Elhage et al. [19] consider a different framing of in-context learning, referring to any model behavior that utilizes information in a prompt to make predictions that improve with prompt size. They hypothesize the existence of special circuits inside Transformer models responsible for in-context learning, that copy similar patterns from the prompt sequence. Pesut [50] and Dinh et al. [16, Table 16] consider in-context learning for small tabular datasets and learning problems in one and two dimensions, and show that GPT-3 can obtain non-trivial accuracy. Our work contributes to this line of work, by posing in-context learning as a well-defined problem of learning function classes at inference time, and empirically investigating if we can train models that in-context learn simple function classes.

Transformers. There is a long line of work investigating the capabilities [66, 15, 74, 49, 73, 7, 75], limitations [25, 6], applications [35, 17, 47], and internal workings [19, 62, 68, 18, 46] of Transformer models. Most similar to our work, Müller et al. [42] and Nguyen and Grover [45] demonstrate the ability of Transformer models to solve prediction tasks using the input context, albeit in different settings. Müller et al. [42] introduce a "Prior-data fitted transformer network" that is trained to approximate Bayesian inference with priors such as Gaussian processes and Bayesian neural networks, and use it to perform downstream tasks such as tabular dataset classification and few-shot image classification. Nguyen and Grover [45] introduce Transformer neural processes, building on prior work on neural processes [24, 23, 29], and show that they achieve state-of-the art performance on tasks such as image completion and contextual multi-armed bandits. Our work complements these works, focusing on understanding the in-context learning ability of Transformers for various simple function classes and the extent to which this ability extrapolates beyond the training distribution.

Meta learning. In-context learning can be viewed as an instance of the more general learning-to-learn or meta-learning paradigm [64, 43]. Typical approaches from this extensive line of work (see [27] for a survey) include: training a meta-learner on how to update the parameters of a downstream learner [4, 32], learning parameter initializations from which one can quickly train for many downstream tasks [21, 54], learning latent embeddings that allow for effective similarity search [63]. Most relevant to our setting are approaches that directly take as input examples from a downstream task and a query input and produce the corresponding output [40, 59, 24, 23]. Our work contributes to this line of work, by investigating the learning-to-learn abilities of Transformer models in a well-defined setting.

Data-driven algorithm design. Another line of work aims to discover algorithms that perform well on a distribution of inputs [26, 72, 67, 3, 28, 61, 60] (as opposed to algorithms with guarantees

on their worst-case performance). See Balcan [1] for a survey on advancements on the theoretical foundations of such algorithms. Our work can be viewed as part of this line of work, as we train Transformer models to discover algorithms for different learning problems.

8 Discussion

In this work, we formalize and study the question: can we train models that learn different classes of functions in-context? We show that Transformer models trained from scratch can in-context learn the class of linear functions, with performance comparable to the optimal least squares estimator, even under distribution shifts. Moreover, we show that in-context learning is also possible for sparse linear functions, decision trees, and two-layer neural networks; learning problems which are solved in practice with involved iterative algorithms such as gradient descent.

At the same time, understanding the implications of our results for language models requires further investigation. A pertinent question regarding the in-context learning capabilities of language models is how they leverage in-context examples [39]. Our results demonstrate that Transformers can encode complex learning algorithms that utilize in-context examples in a far-from-trivial manner. In fact, this is the case for standard Transformer architectures trained with standard optimization procedures. The extent to which such non-trivial in-context learning behavior exists in large language models is still open, but we believe that our work takes a step towards formalizing and understanding this question.

Our work lays the groundwork for several future directions.

Complexity of in-context learning. We empirically show that model capacity helps in performing in-context learning accurately and robustly. How does the in-context learning loss (1) depend on the complexity of the function class $\mathcal{F}$, the capacity of model M, and data used to train M. Understanding this question for models explicitly trained to perform in-context learning may suggest an upper bound for the in-context learning performance of models such as GPT-3 that are not explicitly trained for it.

Curriculum learning. Within our framework, there is natural notion of curriculum learning where, during training, we gradually increase the complexity of the function class learned in-context. This leads to drastic training speed-ups. What is the reason behind such a speedup? Are similar speedups also possible for training large language models (thus significantly reducing training time and energy)?

Inductive bias of model families. Our framework presents an opportunity to understand and compare the inductive biases of different model families (e.g., Transformers vs. LSTMs) in a well-defined setting. For instance, a concrete question is: Are there function classes that are easier to in-context learn using Transformers but harder for LSTMs and vice-versa?

Understanding the learning algorithms encoded in Transformers. The models we train can perform in-context learning, and are thus themselves encoding learning algorithms. However, a limitation of our work is that we do not fully understand the encoded algorithms. A worthwhile research direction would be to investigate the internal workings of these models to get a better understanding of these algorithms. Moreover, for settings such as decision trees, we do not have a good understanding of what the optimal learning algorithms are or the conditions under which known heuristics work [9, 11]. Nevertheless, in Section 5 we found that Transformers are able to discover sample efficient algorithms, thus suggesting an intriguing possibility where we might be able to reverse engineer the Transformer to obtain better learning algorithms for such problems.

Acknowledgements

We thank Niladri Chatterji, Micah Goldblum, Rohith Kuditipudi, Shibani Santurkar, Carmen Strassle, Mirac Sugzun, Li-Yang Tan, and anonymous reviewers for helpful comments.

SG was funded by a Stanford Interdisciplinary Graduate Fellowship. DT was funded by Open Philanthropy, and partially supported by NSF Award CCF-1813049. GV was supported by NSF Awards CCF-1704417, CCF-1813049, Frontier Award 1804222 and DOE award DE-SC0019205. We performed our experiments on the Stanford NLP cluster.

References

[1] Maria-Florina Balcan. Data-driven algorithm design. In Tim Roughgarden, editor, *Beyond Worst Case Analysis of Algorithms*. Cambridge University Press, 2020.

[2] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 2019.

[3] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[4] Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. On the optimization of a synaptic learning rule. In *Preprints Conf. Optimality in Artificial and Biological Neural Networks*, 1995.

[5] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *International Conference on Machine Learning (ICML)*, pages 41–48, 2009.

[6] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

[7] Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of transformers and its implications in sequence modeling. *arXiv preprint arXiv:2006.09286*, 2020.

[8] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[9] Guy Blanc, Jane Lange, Mingda Qiao, and Li-Yang Tan. Decision tree heuristics can fail, even in the smoothed setting. *arXiv preprint arXiv:2107.00819*, 2021.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Neural Information Processing Systems (NeurIPS)*, 2020.

[11] Alon Brutzkus, Amit Daniely, and Eran Malach. Id3 learns juntas for smoothed product distributions. In *Conference on Learning Theory*, pages 902–915. PMLR, 2020.

[12] Stephanie CY Chan, Adam Santoro, Andrew K Lampinen, Jane X Wang, Aaditya Singh, Pierre H Richemond, Jay McClelland, and Felix Hill. Data distributional properties drive emergent few-shot learning in transformers. *arXiv preprint arXiv:2205.05055*, 2022.

[13] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *conference on knowledge discovery and data mining (KDD)*, 2016.

[14] Yanda Chen, Ruiqi Zhong, Sheng Zha, George Karypis, and He He. Meta-learning via language model in-context tuning. *arXiv preprint arXiv:2110.07814*, 2021.

[15] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

[16] Tuan Dinh, Yuchen Zeng, Ruisu Zhang, Ziqian Lin, Shashank Rajput, Michael Gira, Jy-yong Sohn, Dimitris Papailiopoulos, and Kangwook Lee. Lift: Language-interfaced fine-tuning for non-language machine learning tasks. *arXiv preprint arXiv:2206.06565*, 2022.

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[18] Benjamin L Edelman, Surbhi Goel, Sham Kakade, and Cyril Zhang. Inductive biases and variable creation in self-attention mechanisms. In *International Conference on Machine Learning (ICML)*, 2022.

[19] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

[20] Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 1993.

[21] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning (ICML)*, 2017.

[22] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001.

[23] Marta Garnelo, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Rezende, and SM Ali Eslami. Conditional neural processes. In *International Conference on Machine Learning*, pages 1704–1713. PMLR, 2018.

[24] Marta Garnelo, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. Neural processes. *arXiv preprint arXiv:1807.01622*, 2018.

[25] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 2020.

[26] Eric Horvitz, Yongshao Ruan, Carla Gomes, Henry Kautz, Bart Selman, and Max Chickering. A bayesian approach to tackling hard computational problems (preliminary report). *Electronic Notes in Discrete Mathematics*, 2001.

[27] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

[28] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Neural Information Processing Systems (NeurIPS)*, 2017.

[29] Hyunjik Kim, Andriy Mnih, Jonathan Schwarz, Marta Garnelo, Ali Eslami, Dan Rosenbaum, Oriol Vinyals, and Yee Whye Teh. Attentive neural processes. *arXiv preprint arXiv:1901.05761*, 2019.

[30] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[31] Andrew K Lampinen, Ishita Dasgupta, Stephanie CY Chan, Kory Matthewson, Michael Henry Tessler, Antonia Creswell, James L McClelland, Jane X Wang, and Felix Hill. Can language models learn from explanations in context? *arXiv preprint arXiv:2204.02329*, 2022.

[32] Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[33] Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. Jurassic-1: Technical details and evaluation. *White Paper. AI21 Labs*, 2021.

[34] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[35] Kevin Lu, Aditya Grover, Pieter Abbeel, and Igor Mordatch. Pretrained transformers as universal computation engines. *arXiv preprint arXiv:2103.05247*, 2021.

[36] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786*, 2021.

[37] Sewon Min, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Noisy channel language model prompting for few-shot text classification. *arXiv preprint arXiv:2108.04106*, 2021.

[38] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*, 2021.

[39] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

[40] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations (ICLR)*, 2018.

[41] Swaroop Mishra, Daniel Khashabi, Chitta Baral, Yejin Choi, and Hannaneh Hajishirzi. Reframing instructional prompts to gptk's language. *arXiv preprint arXiv:2109.07830*, 2021.

[42] Samuel Müller, Noah Hollmann, Sebastian Pineda Arango, Josif Grabocka, and Frank Hutter. Transformers can do bayesian inference. *arXiv preprint arXiv:2112.10510*, 2021.

[43] Devang K Naik and Richard J Mammone. Meta-neural networks that learn by learning. In *International Joint Conference on Neural Networks (IJCNN)*, 1992.

[44] Preetum Nakkiran. More data can hurt for linear regression: Sample-wise double descent. *arXiv preprint arXiv:1912.07242*, 2019.

[45] Tung Nguyen and Aditya Grover. Transformer neural processes: Uncertainty-aware meta learning via sequence modeling. *arXiv preprint arXiv:2207.04179*, 2022.

[46] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads. *Transformer Circuits Thread*, 2022. https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

[47] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning (ICML)*, 2018.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[49] Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*, 2019.

[50] Lovre Pesut. Who models the models that model models? an exploration of gpt-3's in-context model fitting ability, 2022. URL `https://www.alignmentforum.org/posts/c2RzFadrxkzyRAFXa/who-models-the-models-that-model-models-an-exploration-of`.

[51] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI blog*, 2018.

[52] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[53] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[54] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference for Learning Representations (ICLR)*, 2017.

[55] Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. Impact of pretraining term frequencies on few-shot reasoning. *arXiv preprint arXiv:2202.07206*, 2022.

[56] Frieda Rong. Extrapolating to unnatural language processing with gpt-3's in-context learning: The good, the bad, and the mysterious), 2021. URL `http://ai.stanford.edu/blog/in-context-learning/`.

[57] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633*, 2021.

[58] Terence D Sanger. Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE transactions on Robotics and Automation*, 1994.

[59] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning (ICML)*, 2016.

[60] Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. *Neural Information Processing Systems (NeurIPS)*, 2021.

[61] Daniel Selsam, Matthew Lamm, B Benedikt, Percy Liang, Leonardo de Moura, David L Dill, et al. Learning a sat solver from single-bit supervision. In *International Conference on Learning Representations (ICLR)*, 2018.

[62] Charlie Snell, Ruiqi Zhong, Dan Klein, and Jacob Steinhardt. Approximating how single head attention learns. *arXiv preprint arXiv:2103.07601*, 2021.

[63] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Neural Information Processing Systems (NeurIPS)*, 2017.

[64] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

[65] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 1996.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Neural Information Processing Systems (NeurIPS)*, 2017.

[67] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Neural Information Processing Systems (NeurIPS)*, 2015.

[68] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *International Conference on Machine Learning*, 2021.

[69] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics (ACL), 2020.

[70] Xiaoxia Wu, Ethan Dyer, and Behnam Neyshabur. When do curricula work? *arXiv preprint arXiv:2012.03107*, 2020.

[71] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In *International Conference on Learning Representations (ICLR)*, 2022.

[72] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 2008.

[73] Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. *arXiv preprint arXiv:2105.11115*, 2021.

[74] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077*, 2019.

[75] Yi Zhang, Arturs Backurs, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, and Tal Wagner. Unveiling transformers with lego: a synthetic reasoning task. *arXiv preprint arXiv:2206.04301*, 2022.

[76] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning (ICML)*, 2021.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] We don't perceive any negative impacts beyond those of building large-scale models in general.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Appendix A.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Appendix A.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] Appendix A.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Experimental setup

Here, we provide additional details on our experimental setup.

A.1 Model architecture

We use architectures from the GPT-2 family [51] as implemented by HuggingFace [69] [3] . Specifically, we consider the following set of configurations.

Model	Embedding size	#Layers	#Heads	(Total parameters)
Tiny	64	3	2	3.4M
Small	128	6	4	7.6M
Standard	256	12	8	22.4M

We use the Standard model for the bulk of our experiments and only consider the smaller models for the capacity explorations in Section 6 and Appendix B.3. Since we train on each input once (we sample new inputs at each training step), overfitting to the training data is not an issue. Therefore, we set the Dropout probability to 0.

Out of the box, these models take as input a sequence of vectors in embedding space and output a sequence of vectors in the same space. However, the tasks we study are functions from a lower dimensional vector space (e.g., 10-50 dimensions) to a scalar value. Thus, in order to use a prompt such as $x_1, f(x_1), x_2, f(x_2) \ldots$, we need to map x_is and $f(x_i)$s to vectors in embedding space. We do so by first turning the scalars $f(x_i)$ into vectors of the same dimension as x_i by appending 0s and then applying a learnable linear transformation to map all these vectors into the embedding space. Finally, we map the model output into a scalar value through a dot product with a learnable vector.

We treat the prediction of the model at the position corresponding to x_i (that is absolute position $2i-1$) as the prediction of $f(x_i)$. Due to the structure of these models, this prediction only depends on $(x_j, f(x_j))$ for $j < i$ and x_i. We ignore the model predictions at positions corresponding to $f(x_i)$.

A.2 Training

Each training prompt is produced by sampling a random function f from the function class we are training on, then sampling inputs x_i from the isotropic Gaussian distribution $N(0, I_d)$ and constructing a prompt as $(x_1, f(x_1), \ldots, x_k, f(x_k))$. Given a prompt, we obtain model predictions $\hat{y}_i$ (meant to approximate $f(x_i)$) for each input, and compute the loss

$$\frac{1}{k}\sum_{i=1}^{k}\left(\hat{y}_i - f(x_i)\right)^2 .$$

At each training step, we average the loss over a batch of randomly generated prompts (with different functions and prompt inputs), and perform an update step. We use the Adam optimizer [30], and train for 500,000 total steps with a batch size of 64. We use a learning rate of 10^{-4} for all function classes and models.

Curriculum learning. To accelerate training, we start by training on prompt inputs x_i lying in a smaller dimensional subspace, and with fewer inputs per prompt, and gradually increase the subspace dimension and number of prompt inputs. Specifically, we zero out all but the first d_{cur} coordinates of x_i, sample prompts of size k_{cur} and leave the rest of the training process the same. We use the same schedule for all training runs for the function classes of linear functions and sparse linear functions, starting with $d_{\text{cur}} = 5,\ k_{\text{cur}} = 11$, and increasing d_{cur} and k_{cur} by 1 and 2 respectively, every 2000 steps, until $d_{\text{cur}} = d$, $k_{\text{cur}} = 2d+1$. We use a slightly different schedule for 2 layer neural networks and decision trees as we want prompts with more inputs for these function classes. For these classes, we start with $d_{\text{cur}} = 5,\ k_{\text{cur}} = 26$, and increase d_{cur} and k_{cur} by 1 and 5 respectively, every 2000 steps, until $d_{\text{cur}} = d$, $k_{\text{cur}} = 5d+1$.

Overall, with curriculum, a training prompt $(x_1, f(x_1), \ldots, x_{k_{\text{cur}}}, f(x_{k_{\text{cur}}})$ is generated by sampling a random function f from the function class, drawing inputs x_i by sampling i.i.d. from $N(0, I_d)$ and

[3] `https://huggingface.co/docs/transformers/model_doc/gpt2`

zeroing out all but the first d_{cur} coordinates. Given model predictions $\hat{y}_i$, the loss is given by

$$\frac{1}{k_{\text{cur}}} \sum_{i=1}^{k_{\text{cur}}} \left(\hat{y}_i - f(x_i)\right)^2 .$$

Sampling random functions. For the class of linear functions, we sample random function $f(x) = w^\top x$ by drawing $w \sim N(0, I_d)$. For our main setting (Section 3 and 4), we set $d = 20$.

For the class of two-layer neural networks, we sample $f(x) = \sum_{i=1}^{r} \alpha_i \sigma(w_i^\top x)$, where α_is and w_is are drawn i.i.d. from $N(0, 2/r)$ and $N(0, I_d)$ respectively. We set $d = 20$ and $r = 100$.

For the class of k-sparse linear functions, we sample $f(x) = w^\top x$ by drawing $w \sim N(0, I_d)$ and zeroing out all but k coordinates of w chosen uniformly at random from the first d_{cur} coordinates (as defined in the curriculum learning description above). We set $d = 20$ and $k = 3$.

For the class of decision trees, the random function f is represented by a decision tree of depth 4 (with 16 leaf nodes), with 20 dimensional inputs. Each non-leaf node of the tree is associated with a coordinate selected uniformly at random from $\{1, 2, \ldots, d\}$, and each leaf node is associated with a value drawn randomly from $N(0, 1)$. To evaluate f on an input x, we traverse the tree starting from the root node, and go to the right child if the coordinate associated with the current node is positive and go to the left child otherwise. $f(x)$ is given by the value associated with the leaf node reached at the end.

Computational resources. We train using a single NVIDIA GeForce RTX 3090 GPU and most training runs take 5-20 hours depending on model size and context length. For instance, for the class of linear functions, training the standard model takes 17 hours for $d = 50$, 7 hours for $d = 20$ and 5.5 hours for $d = 10$. For decision trees, training the standard model takes 17 hours. The time it takes for decision trees and 50 dimensional linear functions is higher due to larger context lengths (we train for d dimensional linear functions with $2d + 1$ input-output pairs per prompt).

A.3 Baselines

Least squares. Minimum norm least squares is the optimal estimator for the linear regression problem. Given a prompt $P = (x_1, y_1, \ldots, x_k, y_k, x_{\text{query}})$, let X be a $k \times d$ matrix with row i given by x_i, and let y be a k dimensional vector with the ith entry y_i. Set $\hat{w}^T = X^+ y$, where X^+ denotes the Moore-Penrose pseudoinverse of X. The estimator predicts $M(P) = \hat{w}^T x_{\text{query}}$.

Averaging estimator. This corresponds to $M(P) = \hat{w}^T x_{\text{query}}$ where $\hat{w} = \frac{1}{k} \sum_{i=1}^{k} x_i y_i$. This estimator is consistent (yet sub-optimal) when x_is are drawn from $N(0, I_d)$. Unlike least squares, this estimator does not involve an inverse computation, and might be easier for a model to encode.

Nearest neighbors. This corresponds to setting $M(P) = \frac{1}{n} \sum_{i \in S} y_i$. Here, S is the set of indices of the n nearest neighbors of x_{query} among x_1 to x_k. For $k < n$, we average over all the y_is from 1 to k, and for $k = 0$, we set $M(P) = 0$. We consider the nearest neighbors baselines as it might be easier for a Transformer model to encode using self-attention compared to least squares.

Lasso. We use this baseline for sparse linear functions (Section 5). This corresponds to $M(P) = \hat{w}^T x_{\text{query}}$, where $\hat{w}$ minimizes the ℓ_1-norm regularized least squares objective:

$$\min_{\hat{w}} \frac{1}{2k} ||y - X\hat{w}||_2^2 \; + \; \alpha ||\hat{w}||_1 .$$

We try different values of $\alpha \in \{1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$, and report the best solution (achieving close to 0 error with minimum number of examples) corresponding to $\alpha = 10^{-2}$. To solve the optimization problem, we use the Lasso implementation from Scikit-learn [48] [4].

Greedy Tree Learning. We use this baseline for the class of decision trees. This corresponds to greedily learning a decision tree using the in-context examples, and using it to classify the query input. To construct the tree, at each node (starting from a root node), we choose a coordinate for partitioning the examples into two sets, so as to minimize the variance of y_is in each set, averaged across the two sets. The value associated with a leaf node is the average y_i value of the examples belonging to it. We use Scikit-learn's decision tree regressor [48] [5] implementation for this, with

[4] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

[5] https://scikit-learn.org/stable/modules/tree.html#regression

all the arguments set to their default value except the max_depth argument which is set to 4. We also tried setting max_depth argument to None, but did not find any significant improvement. This corresponds to increasing tree depth until all leaf nodes have a single training example. More details about the algorithm can be found at the webpage linked above.

Tree boosting. For the class of decision trees, we also consider a tree boosting baseline that corresponds to learning an ensemble of decision trees (see Friedman [22] for a description of the general framework). Specifically, we use the XGBoost library [13] [6], an implementation commonly used for a wide range of real-world machine learning tasks. We performed a hyperpameter search by considering {10, 20, 50, 100, 200} estimators in the ensemble (equivalent to number of boosting rounds), a learning rate of {0.01, 0.05, 0.1, 0.2, 0.3, 0.5}, and a maximum depth of {4, 8, 10, 16}, but did not observe significant differences, so we report the results of the default choice of hyperparameters (100, 0.1, and 10, respectively).

Learning neural networks with gradient descent. We use this baseline for the class of two-layer neural networks (Section 5). This corresponds to training a two-layer neural network on the in-context examples, and outputting its prediction on the query point. That is, $M(P) = \hat{f}(x_{\text{query}})$, where

$$\hat{f}(x_{\text{query}}) = \sum_{i=1}^{r} \hat{\alpha}_i \, \sigma(\hat{w}_i^\top x_{\text{query}}).$$

Here, $\sigma(\cdot)$ is the ReLU activation. We find parameters $\hat{\alpha}_i, \hat{w}_i$ by minimizing the squared error of the prediction for the in-context examples

$$\sum_{i=1}^{k} \left(\hat{f}(x_i) - f(x_i) \right)^2,$$

using the Adam optimizer over the whole batch of size k. We use a learning rate of $5 \cdot 10^{-3}$ with 100 optimization steps, and set $r = 100$. We performed initial experiments with different values of these parameters (also considering SGD as an optimizer) and did not find a significant improvement.

[6] https://github.com/dmlc/xgboost

B Additional experimental results

B.1 Robustness to query scale

In Figure 7, we show that the trained model is quite robust to scaling the query input (while keeping the in-context examples fixed): the error does not increase much as we scale up the query input by a factor of up to 2, or scale down by a factor of up to 16, and degrades slowly after that.

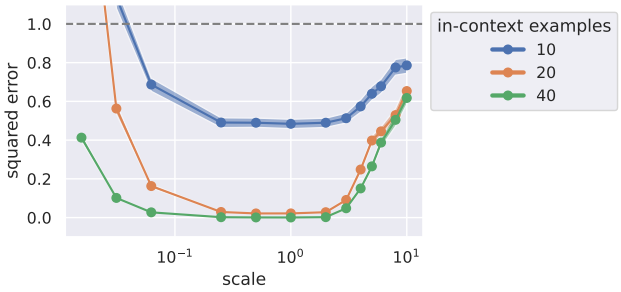


Figure 7: *Robustness to the scale of query input.* For a fixed set of in-context examples, we measure the model's error as we scale the query input by a scalar.

B.2 Out-of-distribution prompts

Here, we describe the structure of our out-of-distribution prompts (cf. Section 4), and show the corresponding plots (Figure 8). To avoid conflating factors, we normalize the prompt inputs such that their expected norm is equal to the expected norm of inputs during training and investigate the role of scaling these inputs separately. We summarize how these prompts deviate from those seen during training in the table below.

Prompting strategy	$D_{\mathcal{X}}^{\text{train}} \neq D_{\mathcal{X}}^{\text{test}}$	$D_{\mathcal{F}}^{\text{train}} \neq D_{\mathcal{F}}^{\text{test}}$	$D_{\text{query}}^{\text{test}} \neq D_{\mathcal{X}}^{\text{test}}$
Skewed covariance	✓		
$d/2$-dimensional subspace	✓		
Scale inputs	✓		
Noisy output		✓	
Scale weights		✓	
Different Orthants	✓		✓
Orthogonal query			✓
Query matches example			✓

Skewed covariance. (Figure 8a) We sample prompt inputs from $N(0, \Sigma)$ where Σ is a skewed covariance matrix with eigenbasis chosen uniformly at random and i^{th} eigenvalue proportional to $1/i^2$. The model matches the performance of least squares until $k = 10$, mimicking the sharp drop in the error in this regime, but its error plateaus afterwards (see Figure 4a). Thus, it is not perfectly robust to this distribution mismatch but still does relatively well, achieving less than half the error of the nearest neighbor baseline in most cases.

Low-dimensional subspace. (Figure 8b) We sample prompt inputs from a random $d/2$ dimensional subspace ($d = 20$ in Figure 8b). That is, we pick a random $d/2$ dimensional subspace, and draw the prompt inputs from an isotropic Gaussian distribution restricted to this subspace. In this case, the model achieves low error after 10 in-context examples, closely matching the behavior of the optimal least squares estimator (the model achieves an error of 0.036, 0.0014, and 0.00057 at 10, 20, and 40 in-context examples respectively). Crucially, unlike the training prompts, when k is between 10 and 20, the prompt inputs are linearly dependent, and a model achieving low error in this regime indicates that it encodes a valid orthogonalization procedure for these inputs.

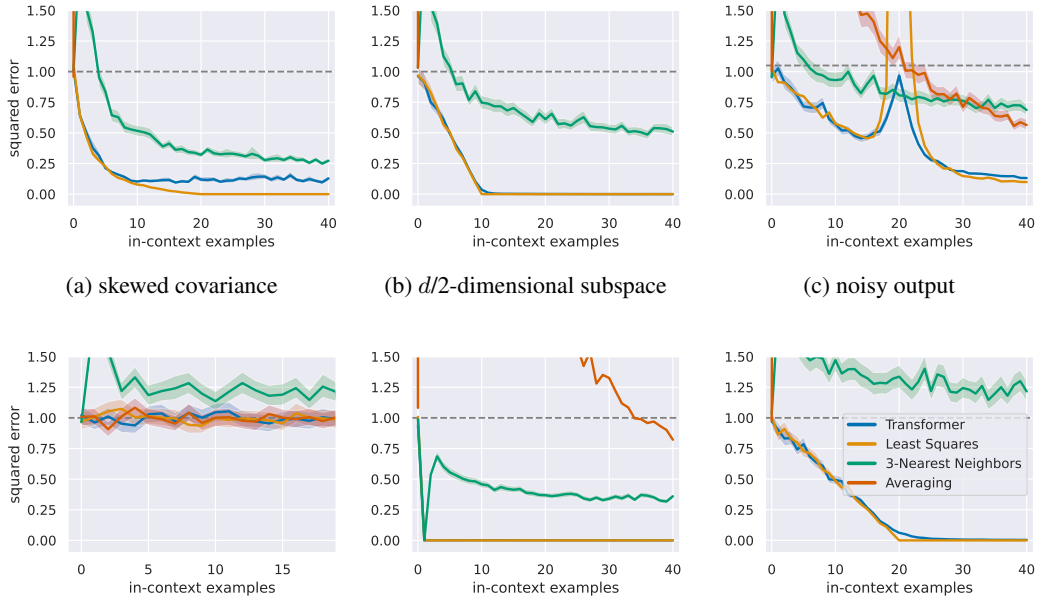
Prompt scale. (Figure 9) We consider the setting where the prompt scale between training and inference is different. We either scale the prompt inputs or the weight vectors, by a factor $\{1/3, 1/2, 2, 3\}$. The model is relatively robust when scaling the weight vector, but not as robust when scaling the prompt inputs, especially for the more extreme scales $1/3$ and 3. Specifically, for 40 in-context examples, the model achieves errors $0.0012, 0.0008, 0.0016, 0.0278$ when scaling the weights, and errors $0.30, 0.013, 0.043, 0.58$ while scaling the inputs, by factors $1/3, 1/2, 2$ and 3 respectively. For context, recall that with 40 in-context examples, the least squares estimator achieves an error of 0 whereas the model achieves an error of 0.0006 at the original scale.

Noisy linear regression. (Figure 8c) We add noise to each prompt output, that is, the ith output is equal to $w^T x_i + \epsilon_i$ where $\epsilon_i \sim N(0, d/20)$. The trained model closely tracks the performance of least squares for most prompt sizes. Interestingly, the model also exhibits the double descent error curve [2] that is known to manifest for the least squares estimator [44]. Note that in this noisy setting, the optimal estimator corresponds to least squares with appropriate ℓ_2-regularization, which we cannot expect the model to learn from noiseless data.

Different orthants for in-context and query inputs.(Figure 8f) We fix the sign of each coordinate to be positive or negative for all in-context inputs x_i (at random), and draw x_{query} (as before) i.i.d. from $N(0, I_d)$. As a result, all in-context inputs lie in the same orthant, while the query input lies in another orthant with high probability. The model is not affected by the mismatch between in-context and query inputs and closely match the performance of least squares. In this case, the model achieves errors 0.062 and 0.004 for 20 and 40 in-context examples respectively, whereas recall that it achieves errors 0.02 and 0.0006 on standard prompts. This indicates that the models is not relying on some variant of nearest neighbor search as in that case, its error would have been higher (see the 3-nearest neighbor baseline).

Query input orthogonal to in-context inputs.(Figure 8d) We choose the query input randomly in the space orthogonal to the space spanned by in-context example inputs (there can be at most $d-1$ in-context examples for an orthogonal query to exist). That is, we draw the query input from an isotropic Gaussian distribution restricted to the subspace orthogonal to the space spanned by the in-context examples. Here, there is no information relevant to the query input in the in-context examples and thus the model would ideally predict something close to 0 to minimize the error. Indeed, the model outputs such a prediction, achieving an error close to 1.

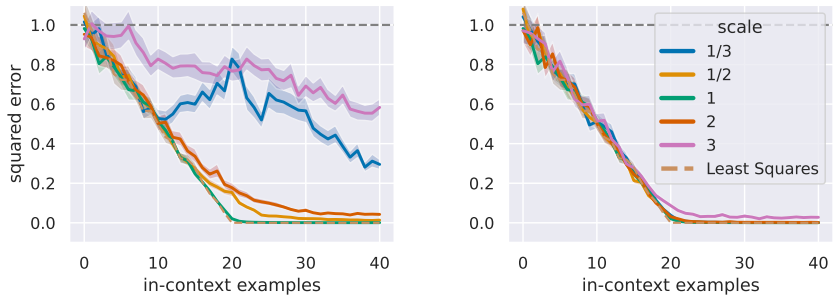
Query input matches an in-context example.(Figure 8e) We set the query input equal to one of the in-context examples chosen uniformly at random. Thus it's possible to achieve zero error since the in-context examples include the correct prediction for the query input already. In this case, the model achieves errors $0.001, 0.001, 0.0005$ for 10, 20, 40 examples respectively thus making close to the correct prediction, without being affected by the additional in-context examples present.



(a) skewed covariance (b) $d/2$-dimensional subspace (c) noisy output

(d) orthogonal query (e) query matches in-context example (f) different orthants

Figure 8: *In-context learning on out-of-distribution prompts.* We evaluate the model trained to in-context learn linear functions on prompt distribution that deviates from the training prompt distribution. In general, the model error degrades gracefully and closely tracks that of the least squares estimator.



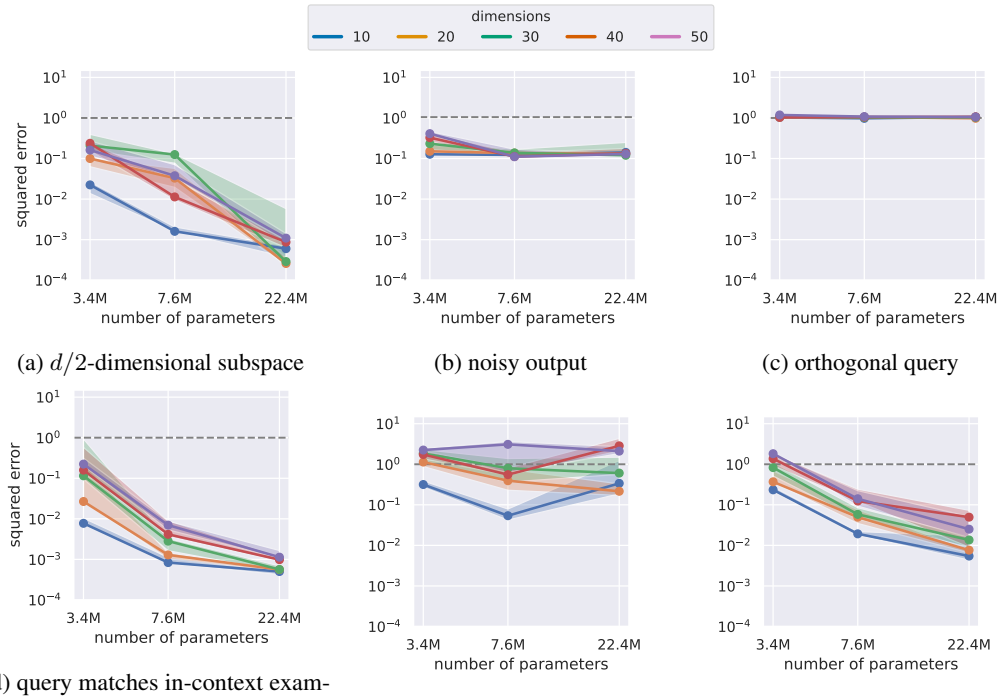
(a) scaled x, Transformer (b) scaled w, Transfomer

Figure 9: *In-context learning robustness to prompt scaling.* We evaluate the model trained to in-context learn linear functions when we scaled the prompt inputs x or the weight of the function class w. The model appear to be quite robust to scaling w but their performance degrades when scaling the inputs up or down by a factor of 3.

B.3 Effect of problem dimension and model capacity

We plot the model error for additional out-of-distribution prompts in Figure 10 for $2d$ in-context examples (with the exception of orthogonal queries where we use $d-1$ in-context examples).

Similar to the settings in Section 6 (skewed covariance and different orthants), accuracy improves with capacity in most cases. One exception is scaling x (Figure 10e), in which case we do not see any clear trend. In the case of noisy output (Figure 10b), the accuracy almost saturates at 7.6M parameters, close to the error of the least squares estimator. In the case of orthogonal query input (Figure 10c), the model achieves the optimum error of 1 even with the tiny model with 3.4M parameters.



(a) $d/2$-dimensional subspace

(b) noisy output

(c) orthogonal query

(d) query matches in-context examples

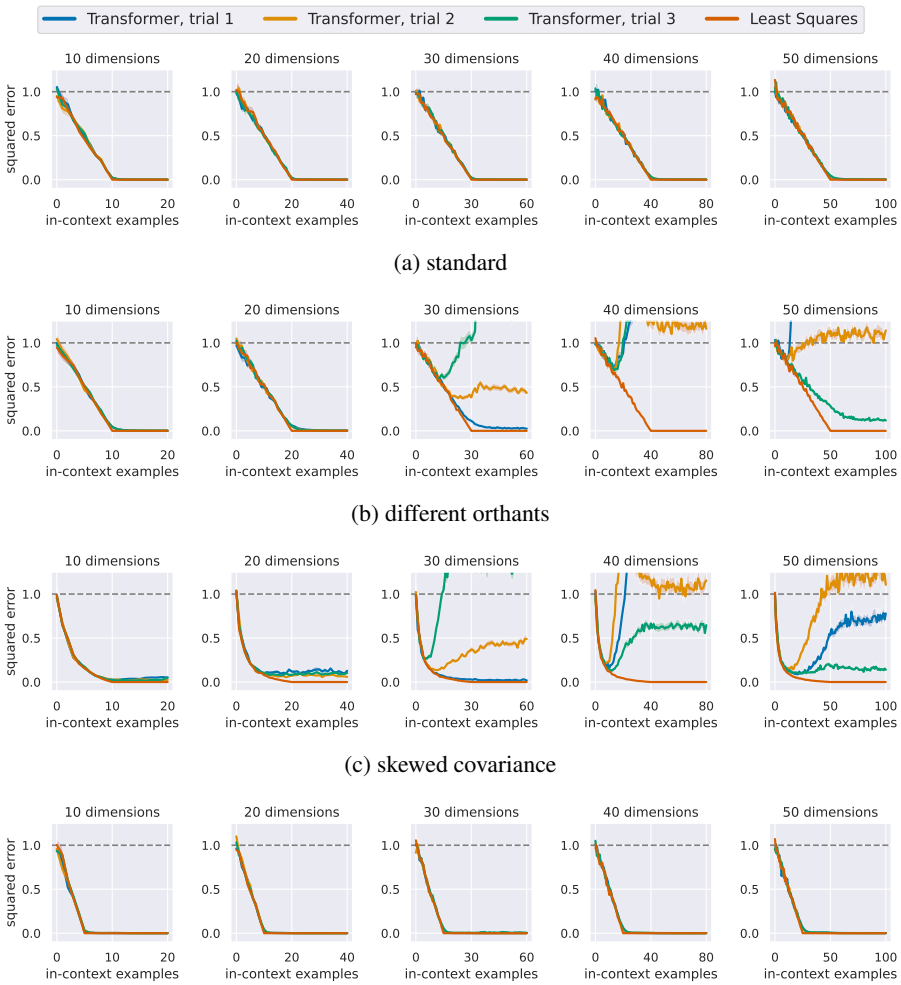
(e) scale x by a factor of 2

(f) scale w by a factor of 2

Figure 10: *The effect of model capacity and problem dimension for in-context learning performance on out-of-distribution prompts.* We train Transformer models of varying capacity to in-context learn linear function in varying dimensions d. We plot the error with $2d$ in-context examples (or $d-1$ for orthogonal queries). We find that capacity helps in most cases, with the exception of scaling x where we find no clear trend. For each setting, we train 3 models with different random seeds, and show the median error (solid lines), and the minimum and maximum errors (shaded region). (See Figures 6b, 6c in the main text for the corresponding plots on different-orthants and skewed-covariance.)

B.4 Training variance

In Figure 11, we show the variance in error across training runs for the standard Transformer model (22.4M parameters). We plot the squared error for 3 models (with different random seeds) each for $d \in \{10, 20, 30, 40, 50\}$, trained to in-context learn linear functions. The error is quite concentrated in the standard setting as well as for most out-of-distribution prompts. In the different-orthants and skewed-covariance settings, we observe a high variance for higher dimensional problems $(d \geq 30)$. However, in Section 6, we saw that the error in these settings usually decreases as we increase the model size. In the setting where we scale x, there is high variance even when $d = 10$.



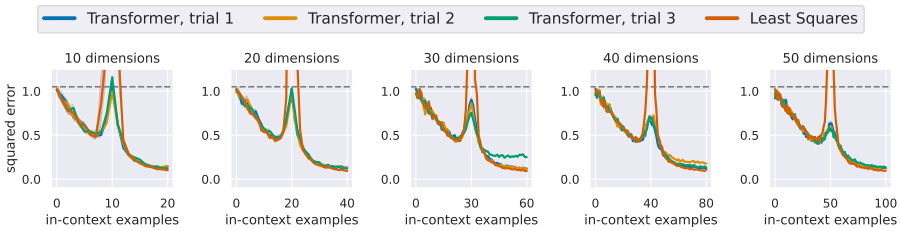
(a) standard

(b) different orthants

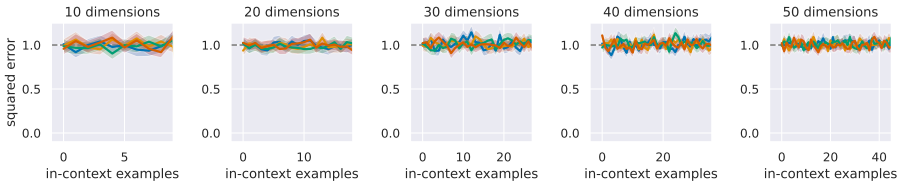
(c) skewed covariance

(d) $d/2$-dimensional subspace

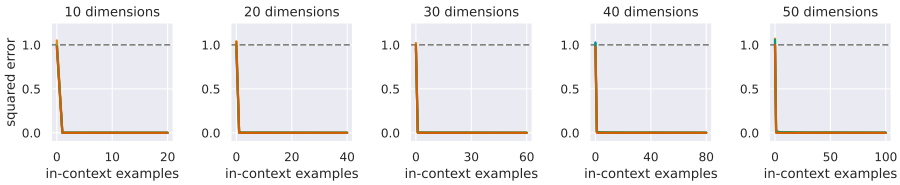
Figure 11: *Errors for models trained with different random seeds.* For each dimension, we train three models with different random seeds and show the corresponding error curves.



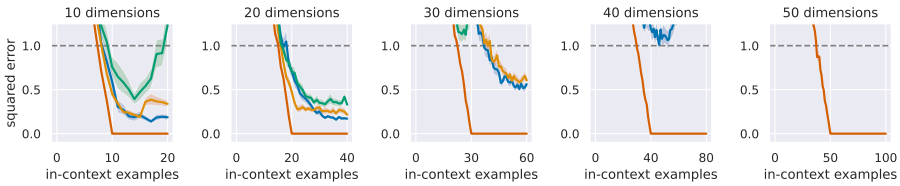
(e) noisy output



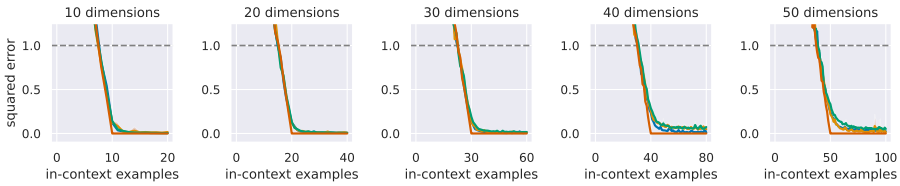
(f) orthogonal query



(g) query matches in-context example



(h) scaled x by a factor of 2



(i) scaled w by a factor of 2

Figure 11: (continued) *Errors for models trained with different random seeds.* For each dimension, we train three models with different random seeds and show the corresponding error curves.

B.5 Curriculum

In Figure 12, we show the training loss of the Transformer model trained to in-context learn linear functions, with and without a curriculum. Specifically, given a random training prompt sequence $(x_1, f(x_1), x_2, f(x_2), \ldots, x_{k_{\text{cur}}}, f(x_{k_{\text{cur}}}))$, let $\hat{y}_i$ be the model's prediction for the ith input (meant to approximate $f(x_i)$). For each such prompt, we consider the loss given by the normalized squared error averaged over all prompt prefixes

$$\frac{1}{k_{\text{cur}}} \sum_{i=1}^{k_{\text{cur}}} \frac{(\hat{y}_i - f(x_i))^2}{d}.$$

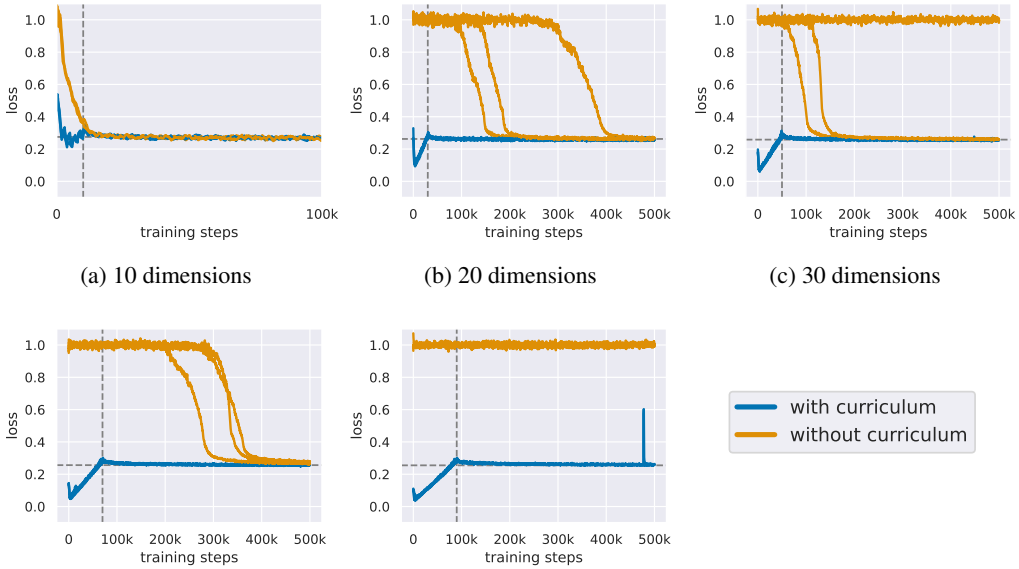
At each training step, we plot the loss averaged over a batch of 64 random prompts. For training with curriculum, k_{cur} is gradually increased to $2d + 1$ as described in Section A.2. For training without curriculum $k_{\text{cur}} = 2d + 1$ at all times.

Note that the loss often increases in the beginning as we train the model with curriculum. This is due to a sharp increase in the loss at steps where we increase the effective dimensionality (d_{cur}) of prompt inputs (x_i). There are two reasons for this increase: (i) variance of the target output ($f(x_i) = w^\top x_i$) increases, so even the optimum loss is larger, (ii) the model performance is worse for the prompt inputs with increased effective dimension. After each such step where we increment d_{cur}, the loss starts to decrease again until the next increment. The overall trend in the loss looks upward when the sharp increase dominates the decrease that follows. Some observations worth highlighting are as follows.

Curriculum drastically speed-ups training. For functions in 20 or more dimensions, curriculum allows us to train a low-error model often 4 times faster. Moreover, training without curriculum does not always succeed within our training budget (500k steps), e.g., for one run with $d = 30$ and *all* runs with $d = 50$, the loss does not decrease at all without curriculum.

Initial lull without curriculum. For training without curriculum, we observe that the loss does not decrease for relatively a long period in the beginning, and starts to decrease sharply thereafter. There is a large variance in the length of this period for any fixed dimension, and the average length seems to increase with dimension. This period is almost non-existent for smaller dimensions (e.g., see the plot for $d = 10$), and therefore we do not observe such a period while training with curriculum where we start training with inputs lying in a 5 dimensional subspace.

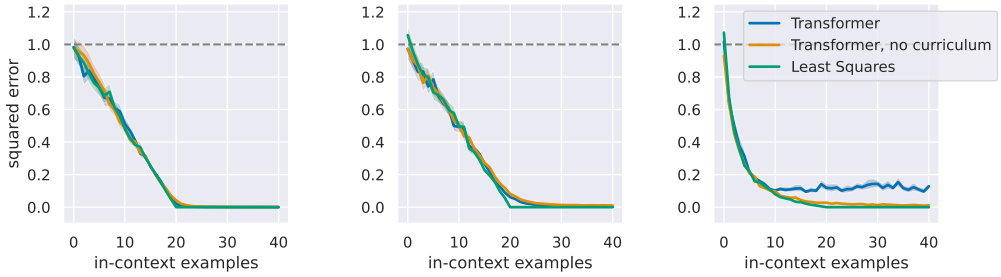
Curriculum does not affect final performance significantly. For our core setting ($d = 20$), where we are able to train the model to low error even without curriculum, we do not observe any qualitative differences in the error in most cases (both with and without distribution shifts). One exception is the case with skewed covariance, where the model trained without curriculum seems to do slightly better. We plot the error curves for the standard, different orthants and skewed covariance cases in Figure 13.



(a) 10 dimensions (b) 20 dimensions (c) 30 dimensions

(d) 40 dimensions (e) 50 dimensions

Figure 12: *Loss progression during training with and without curriculum.* For each dimension, we show the loss progression with 3 random seeds each for training with and without curriculum. The vertical dashed line shows the point at which the effective dimension of prompt inputs d_{cur} reaches the actual dimension d, after which training with and without curriculum have the same prompt distribution. The horizontal dashed line shows the optimum expected loss. There is a drastic speedup in training with curriculum. Without curriculum, there is an initial relatively long period where the loss does not decrease. For each dimension, there is a large variance in the length of this period, and the average length seems to increase with dimension.



(a) standard (b) different orthants (c) skewed

Figure 13: *In-context learning performance for models trained with and without curriculum.* We show the performance for models trained with and without curriculum for in-context learning linear functions ($d = 20$). We did not observe any major qualitative difference in performance between the two settings in most cases. One exception is the case with skewed covariance where the model trained without curriculum does better.

B.6 Effect of number of distinct prompts/functions seen during training

Here, we investigate the effect of amount of training data required for in-context learning linear functions.

First, we consider the effect of number of distinct prompts encountered during training. For this, we create a set S_p of n_p randomly generated prompts, where each prompt in S_p is generated by sampling a weight vector and prompt inputs from $N(0, I_d)$. We generate random prompts during training by sampling uniformly from this set. As before, we train the model for $500k$ steps with a batch size of 64. We observe (see Figure 14) that a model trained with $n_p = 100k$ is able to achieve non-trivial error and a model trained with $n_p = 1M$ achieves error close to that of the unrestricted model (trained with $32M$ distinct prompts). Recall that with curriculum learning, we zero out some of the coordinates of prompt inputs in the beginning of training, which will increase the total number of prompts the model sees during training. Therefore we do not use curriculum learning for this study to avoid inflating the number of distinct prompts seen during training.

Second, we consider the effect of number of distinct weight vectors (equivalently, distinct functions) encountered during training. For this, we create a set S_w of n_w weight vectors where each weight w is drawn i.i.d. from $N(0, I_d)$. To generate a training prompt, $(x_1, w^\top x_1, \ldots, x_k, w^\top x_k)$, we draw prompt inputs (x_is) i.i.d. from $N(0, I_d)$ as in the unrestricted setting, and sample w uniformly at random from S_w. Thus while we sample from a finite set of weight vectors, we sample fresh inputs at each step. As before, we train the model for $500k$ steps with a batch size of 64. Here, we observe (see Figure 14) that the model trained with as few as $10k$ distinct weight vectors achieves error close to the unrestricted model (trained with $32M$ distinct functions). We use curriculum learning for this study as in our standard setting. Recall that with curriculum learning, we only zero out some coordinates of prompt inputs in the beginning, so this does not change the number of distinct weight vectors seen by the model during training.

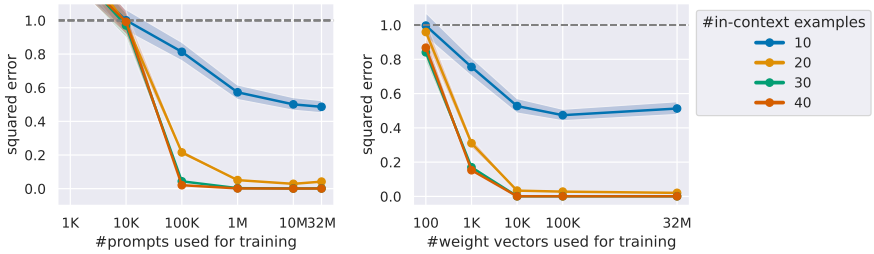


Figure 14: *Effect of number of distinct prompts/functions seen during training.* We plot the squared error for models trained to in-context linear functions, as we increase the number of distinct prompts and distinct weight vectors (equivalently, distinct functions) seen during training. (Note that 32M corresponds to the unrestricted model where we sample fresh prompts at each training step.) The models are able to achieve error close to that of the unrestricted model with $1M$ distinct prompts or $10k$ distinct weight vectors.

B.7 Can memorization explain model performance?

Can memorization of training prompts or functions seen during training explain the model performance? Consider the setting of linear functions from Section 3. Note that the probability of the model encountering a training prompt similar to the one used for testing is astronomically low—the prompt inputs alone lie in a 800-dimensional space when predicting with $2d$ in-context examples ($d = 20$). Moreover, even considering the possibility that the model encountered a similar *weight vector* during training cannot explain its performance. That is, the model encounters 32 million random weight vectors during training and even using the best of these vectors would lead to an expected error of around 0.2 (computed empirically, details below). However, the model is able to achieve an error of less than 0.001 for a prompt with $2d$ in-context examples. Further, in Section 6, we show that the model is able to obtain a similar error even when trained on prompts generated using only $10,000$ distinct weight vectors, in which case the best weight vector seen during training would yield an even worse error of around 0.5. Thus, the model cannot be relying on memorization of training prompts or weight vectors, and instead encodes an algorithm capable of in-context learning linear functions that are very different from those seen during training.

Error for the algorithm memorizing the weight vectors. Let S_w be the set of weight vectors used to generate training prompts. At inference time, given a prompt with in-context examples generated using a weight vector $w_\star$, suppose the model is somehow able to find the best weight vector $\hat{w}$ in S_w minimizing the normalized squared error on query inputs:

$$\begin{aligned}\hat{w} &= \underset{w \in S_w}{\arg\min}\ \mathbb{E}_{x_{\text{query}} \sim N(0, I_d)} \left[\frac{(w^\top x_{\text{query}} - {w_\star}^\top x_{\text{query}})^2}{d} \right] \\ &= \underset{w \in S_w}{\arg\min}\ \frac{\|w - w*\|_2^2}{d}\end{aligned}$$

Taking expectation over the weight vector $w_\star$, we get the expected normalized squared error of the model (with respect to randomly drawn in-context examples and query inputs):

$$\mathbb{E}_{w_\star \sim N(0, I_d)} \left[\min_{w \in S_w} \frac{\|w - w_\star\|_2^2}{d} \right].$$

To empirically estimate this quantity, we sample n_w weight vectors from $N(0, I_d)$ (with $d = 20$) that form the set S_w, and 500 weight vectors from $N(0, I_d)$ to estimate the outer expectation. We do this 20 times, freshly sampling the 500 weight vectors and the vectors comprising S_w each time, and compute the mean of the 20 estimates obtained. When $n_w = 32M$ (number of weight vectors encountered in our standard training setup), we get a mean of 0.216 (standard deviation 0.004). However, our model is able to achieve an expected error of less than 0.001 for prompts with $2d$ in-context examples. Similarly, when $n_w = 10,000$, we get a mean of 0.505 (standard deviation 0.006), while a model trained on prompts generated using $10,000$ distinct weight vectors is able to achieve a much smaller error (see Figure 14).