

Transformers as Empirical Bayes Estimators The Poisson Model

by

Mark Jabbour

B.S. Mathematics and Computer Science and Engineering, Massachusetts Institute of
Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

© 2025 Mark Jabbour. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Mark Jabbour
Department of Electrical Engineering and Computer Science
Jan 25, 2025

Certified by: Yury Polyanskiy
Associate Professor, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Transformers as Empirical Bayes Estimators The Poisson Model

by

Mark Jabbour

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 25, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE

ABSTRACT

We study the ability of transformers to perform In Context Learning (ICL) in the setting of Empirical Bayes for the Poisson Model. On the theoretical side, we demonstrate the expressibility of transformers by formulating a way to approximate the Robbins estimator, the first empirical Bayes estimator for the Poisson model. On the empirical side, we show that transformers pre-trained on synthetic data can generalize to unseen prior and sequence lengths, outperforming existing methods like Robbins, NPMLE, and ERM monotone in efficiency and accuracy. By studying the internal behavior of the representations of the intermediate layers of these transformers, we found that the representation converges quickly and smoothly over the layers. We also demonstrate that it's unlikely transformers are implementing Robbin's or NPMLE estimators in context.

Thesis supervisor: Yury Polyanskiy

Title: Associate Professor

Acknowledgments

I have received a lot of help throughout the process of building my thesis. From navigating the vast literature on Empirical Bayes, Attention, and In Context Learning, to designing and interpreting experiments.

I am very grateful to my research supervisor Professor Yury Polyanskiy. His mentorship, guidance, and kindness were crucial throughout the process. I am also very thankful to Anzo Teh for collaborating closely with me and always being eager to answer questions and offer advice. This thesis came at a personally difficult time, and I am lucky to have been supported by great mentors.

I would also like to thank my family and friends for supporting me throughout my academic journey.

Contents

Title page	1
Abstract	3
Acknowledgments	5
List of Figures	9
List of Tables	11
1 Introduction	13
2 In Context Learning (ICL)	15
2.1 A Brief Introduction to ICL	15
2.2 Transformer Based Models	15
2.3 Literature Review of In Context Linear Regression	16
2.3.1 Theoretical Results	17
2.3.2 Empirical Observations	17
3 Empirical Bayes Literature Review	21
3.1 Definitions and Key Theoretical Results	21
3.1.1 Description and Famous application	21
3.1.2 Bayes Estimator for the Poison Model	21
3.1.3 Regret Lower Bound	22
3.1.4 Worst Priors and Golden Standard Estimators	22
3.2 Regret Optimal Poison Estimators	23
3.2.1 Robbins	23
3.2.2 NPMLE	24
3.2.3 ERM	24
4 Proposed Transformer Based Empirical Bayes Estimator	25
4.1 Problem Setting	25
4.2 Architecture Description	26
4.3 Training Data Distribution	27
4.3.1 Neural Prior	27
4.3.2 Dirichlet Distribution	27

4.4	Evaluation Metrics	28
4.4.1	MSE paired T-Test	28
4.4.2	Plackett-Luce Rank Model for Rank Aggregation	28
4.4.3	Regret estimates	29
4.5	Selected Models	29
4.5.1	Chosen Training Prior	29
4.5.2	Selected Hyper-parameters	29
5	Expressibility of Transformers	31
5.1	Setup	31
5.2	Dimension Inefficient Construction	31
5.3	More Efficient Use of Attention Dimensions	33
5.4	Transformers Can Get Arbitrary close to asymptotically Regret Optimal	34
6	Evaluation	35
6.1	Out of Distribution Synthetic Evaluation	35
6.1.1	Changing The Sequence Length	35
6.1.2	Unseen priors: Worst Prior	35
6.1.3	Cost of being robust to Support	36
6.1.4	Summary of Synthetic Evaluation	37
6.2	Runtime	38
6.3	Evaluation on Real Data	39
6.3.1	Sports datasets	39
6.3.2	Word frequency datasets	40
6.3.3	Result Summary	41
7	Model Internals	43
7.1	Activations Converge Smoothly and Rapidly	43
7.2	Activations of Our Layers Do not Encode Information Necessary Robbins or NPMLE	43
8	Concluding Remarks	45
8.1	Summary	45
8.2	Future Directions	45
8.2.1	Improve and Better Understand the Models from This Work	45
8.2.2	Working on Empirical Bayes for Multidimensional Poisson	46
8.2.3	Studying the effect of Changes to the Setup	46
A	Pattern in the Updates To Transformer’s Residual Steams	47
B	Code Snippets	51
B.1	Transformer	51
B.2	Plackett-Luce	54
	References	59

List of Figures

2.1	Transformer Based Models	16
2.2	In Context Linear Regression Problem Structure. The goal is to predict Y_n given $n - 1$ (x, y) pairs, as well as x_n	17
4.1	In Context Empirical Bayes Problem Structure. The goal is to predict $\theta_1 \dots \theta_n$ given $x_1 \dots x_n$ such that $x_i \sim \theta_i$	25
4.2	Chosen Transformer Model Architecture with layers $L = 2k$. The first half of the layers share all weights except for normalization, and so does the second half.	26
4.3	Top row shows the distribution of θ , the model's target, for different priors. The bottom row shows the Poisson mixtures, which are the model's inputs. We note that for a single random network, θ and x appear clustered. For Neural Priors θ exhibits a few separated clusters, but they are smeared in x . With Dirichlet θ it varies chaotically in a none-continuous manner, after applying Poisson it becomes smoother	27
4.4	Histogram of regret on neural prior for different estimators. We can see that the Dirichlet-only models performs poorly, though it still outperforms Robbins and worst prior. We also see that the performance drop from adding Dirichlet to the training data is small.	30
5.1	The Structure used for transformer based models that implement clipped Robbins estimator	32
6.1	Regret vs sequence length. The regret decreases for both transformers as the sequence length increases, showing that they do have the ability to generalize. We nevertheless note that NPMLE has a better generalization ability, as shown by the regret at sequence length 2048 as compared to smaller sequences. In comparison, the average regret for ERM monotone is 11.20, 8.19, 5.58, 3.66, 2.36 for the various sequence lengths, while the average regret for MLE and worst prior Bayes stays constant at 14.816 and 14.658, respectively	35

6.2	Worst Prior SeqLen Regret of various transformers on worst prior as compared to NPMLE. Again, transformers show the ability to generalize to longer sequence lengths, although for longer sequences NPMLE generalizes better. In comparison, this is already better than ERM-monotone’s regret at 12.79, 9.80, 6.99, 4.81, and 3.256421 across the 5 sequence lengths, while MLE’s regret stays at 11.73.	36
6.3	Training on a randomized θ_{\max} increases mean regret by 18.5% and 23.4% for the transformers with 18 and 24 layers, respectively, when compared against transformers that learn the true θ_{\max} during training. The corresponding t-stat are 32.99 and 36.16.	37
6.4	Time vs sequence length (labels in the form of (layers, model dimension, head), showing that the inference time of transformers is comparable with that of ERM. We nevertheless qualify this finding by noting that there seems to be a running time that scales super-linearly with sequence length. For comparison, the running time of NPMLE for sequence lengths 128, 256, 512, 1024, and 2048 are 41.69, 67.70, 109.81, 175.72, and 289.79 seconds, respectively, which indicates that transformers are 2 orders of magnitudes faster than NPMLE given the GPU.	39
7.1	Plot of R2 for the decoder output of a transformer model against θ and $\hat{\theta}$ activations.	44
7.2	We perform a linear probe for the representations of $N(x)$, and $f_{\pi}(x)$, we notice that they have a downward trend	44
A.1	Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added by the MLP in layer i	47
A.2	Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added in the MLP in layer i	48
A.3	Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added in the MLP in layer i	49

List of Tables

4.1	Table of regret difference; $A-B$ denotes the difference of regret of transformers trained on A vs trained on B	29
4.2	The hyperparameters for selected models	30
6.1	Plackett-Luce Rankings on Synthetic Experiments	37
6.2	T -stat of T18 against classical algorithms (positive = improvement)	38
6.3	T -stat of T24 against classical algorithms (positive = improvement)	38
6.4	Relative Improvement of each algorithm over MLE in MAE metric, 95% confidence interval.	42
6.5	Percentage Improvement of each algorithm over MLE in RMSE metric, 95% confidence interval	42
6.6	Plackett Luce ELO Ranking Coefficient based on RMSE. The coefficient of MLE is set to 0 throughout.	42

Chapter 1

Introduction

Transformers were introduced in 2017 [1] as a more efficient alternative to recurrence-based natural language processing structures like LSTMs. The efficiency comes from replacing the recursive structure, with an attention mechanism and positional encoding. Getting rid of the recursive structure allows for more parallelism in processing data, which in turn allows language models to grow into the Large Language Models (LLMs) we know today. It also helps improve the model's ability to connect tokens that are far apart.

One exciting fact about LLMs is their ability to combine knowledge from text they trained on with patterns extrapolated from a few examples. This is known as In Context Learning (ICL) [2]. ICL can allow us to use prompts to guide LLMs to do a specific task for us based on the knowledge they obtained, without having to retrain them. This is believed to be useful in two ways:

- it avoids the cost of curating many examples to train models on specific tasks like sentiment analysis, a few examples would suffice instead
- It allows the models to learn from richer data which could make them more powerful

This phenomenon is believed to be at the heart of what makes transformers so effective, and it has been a hot research topic. Many works attempt to understand in context learning by analyzing its behavior on linear regression. These include [3], [4] and [5]. We start a new line of research that studies this phenomenon by analyzing it in the setting of Empirical Bayes (EB).

Empirical Bayes is a classical statistical approach, where instead of assuming a specific prior, only the existence of a prior is used to derive new estimators. Especially in independent decision problems, where collective performance is the objective.

Empirical Bayes has also received increased attention lately. On the one hand, EB techniques have been increasingly used in economics [6]. Particularly in value-added studies, which tend to involve multiple parallel estimation problems for many similar units (such as the effect of different schools on a graduate's income). On the other hand, several theoretical breakthroughs were recently made in the EB problems of estimating the parameters of several Poisson distributions. Including proving the asymptotic optimality of certain estimators, theoretically optimal estimators with desirable practical properties, and more tractable methods for multidimensional data. [7] [8] [9]

We believe that EB is a good problem to study in context learning in transformers for the following reasons:

1. Empirical Bayes operates on multiple independent tokens; unlike the pairs in linear regression. This simplifies the model as it removes the need for positional encoding and helps us focus on how tokens interact with each other in the attention mechanism.
2. Empirical Bayes Estimators exhibit a shrinkage effect. Recent work has shown that the attention mechanism tends to cluster tokens [10]
3. Unlike linear regression, there is no known practical closed form. Hence, transformers could offer a favorable mix of efficiency and accuracy, especially in high-dimensional settings.
4. Previous work on in context linear regression suggests that transformer-based models appear to implement a Bayesian algorithm that depends on the distribution of the data [4]. Further suggesting that they might be good empirical Bayes estimators.

Chapter 2

In Context Learning (ICL)

2.1 A Brief Introduction to ICL

In Context, Learning refers to a model's ability to learn how to perform a task from a few examples in its input, without updating its weights. This phenomenon got a lot of interest in the machine learning community after [11] demonstrated that GPT-3, which was much larger than contemporary models in both the number of parameters and size of the training dataset, was able to outperform state of the art models fine-tuned or trained specifically for a certain task, by getting a small number of examples of the task added to the input.

Interest in ICL led to interest in crafting the right "context", by giving models a prompt to elicit the desired behaviors. They include trying to guide the LLM by asking it to "think step by step" [12], as well as embedding based techniques such as Retrieval Augmented Generation where indexed data is queried for relevant information to add to the context [13].

The Success of LLMs and ICL prompted many works that apply them to problems beyond NLP. For example [14] managed to use off-the-shelf text-only LLMs to directly predict robot actions through ICL without training, and [15] managed to use an LLM as a robot controller in a feedback loop. Additionally, many works such as [16] attempt to bring ICL to multi-modal problems.

2.2 Transformer Based Models

Transformer based models discussed for the rest of this document have the structure described in 2.1. They start with an encoder that independently maps each token into a vector and optionally adds a positional encoding to it. Then they have a series of transformer layers, and they finally end with a decoder that operates independently on the embeddings, mapping the embedding associated with each input to an output.

The Structure of the Transformer Layer

As noted in 2.1, each transformer layer starts with a Multihead Attention with a skip connection and a normalization layer. Followed by a Multilayer Perception (MLP), with another

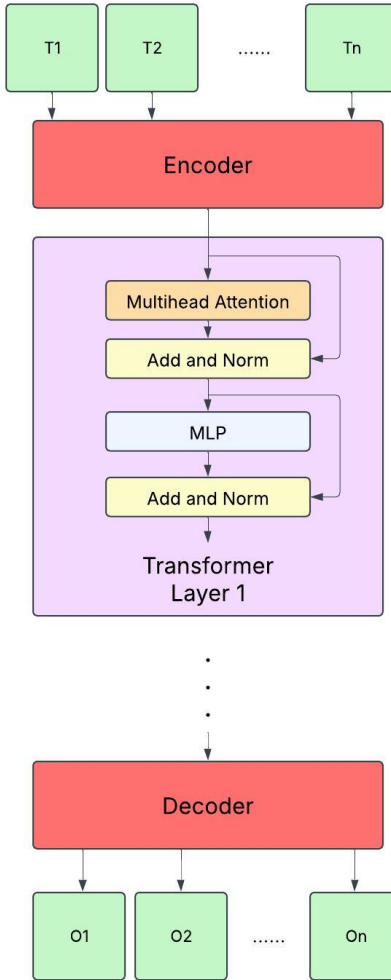


Figure 2.1: Transformer Based Models

skip connection and a normalization layer.

2.3 Literature Review of In Context Linear Regression

Many works try to improve our understanding of ICL by studying how transformer based models solve linear regression in context [3], [4], [5]. The structure of the problem is illustrated in 2.2. The parameters of the models θ are trained with the following objective for some distributions $p(x)$ over \mathcal{R}^n and $p(f)$ over linear functions of $\mathcal{R}^n \Rightarrow \mathcal{R}^1$:

$$\min_{\theta} \mathbb{E}_{\substack{x_1, \dots, x_n \sim p(x) \\ f \sim p(f)}} \left[\sum_{1 \leq i \leq n} (f(x_i) - T_{\theta}(x_1, f(x_1), x_2, f(x_2) \dots x_i))^2 \right] \quad (2.1)$$

In the following we highlight the different kinds of results these works have produced to guide our work on empirical Bayes.

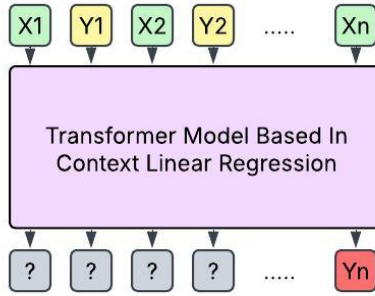


Figure 2.2: In Context Linear Regression Problem Structure. The goal is to predict Y_n given $n - 1$ (x, y) pairs, as well as x_n

2.3.1 Theoretical Results

It is known that transformer based models are universal learners. For example [17] shows that there’s a transformer based network that can learn any continuous permutation equivariant sequence-to-sequence functions with compact support up to any desired degree of precision. This implies that there’s a transformer based approximation of empirical Bayes. It also proves that if we allow the incorporation of a trainable positional encoding, transformers approximate sequence to sequence functions, which implies the transformer based models can learn linear regression. However, the constructions in these papers tend to depend on very deep models.

Works that focus on ICL for linear regression try to understand the number of layers needed. For example [3] provides a list of primitive operations that can each be implemented with a single layer, then proves that these operations can be used to implement a single iteration of SGD on linear regression with a constant number of layers, and $O(d)$ hidden space (where d is the dimension of the linear regression problem). It is then proved that transformers can implement a closed formula of linear regression with a constant number of layers and $O(d^2)$ hidden dimension. [5] Uses the same set of primitive operations to prove that transformer based models can implement k iterations of Newton’s algorithms in $k + 8$ layers $O(d)$ dimensions.

2.3.2 Empirical Observations

Some of the main experiments are classified in [3], [4], [5] and summarized below. Note that there are minor differences between them, for example in the kind of attention used (causal or not). However, I focus on the experiment designs and the results they yield to guide our experiments.

Distribution Shifts

To check the robustness of the algorithm the transformers learned in context, [4] trains a model on noise-free linear data, where weights are sampled from an Isotropic Gaussian. Then evaluate it on the following changes to the distribution:

- **Skewed Weights:** when the weights are sampled from a non-isotropic Gaussian, MSE seems to plateau even when there are enough samples to fully determine the problem. However, it still outperforms 3-nearest neighbors.
- **Adding Noise:** model closely tracks least-square estimators, even exhibiting a double descent behavior
- **Different Orthans:** Restricting the sign of each of the weights in the training data did not seem to significantly affect generalization.

Probing Intermediate Activations for Quantities Used in Regression Algorithms

[3] Uses probes to decide whether the activations of the transformer encode quantities that are used in well-known linear regression algorithms. Particularly, the weights implied by the input and the second moments.

- Two Layer MLP outperforms linear probe even outside the training sample, suggesting that the representation is non-linear.
- For the first few layers, the performance is poor even for the non-linear probe. However, in the deeper layer, the probe performs well. Suggesting that the model indeed tries to encode these quantities in its weights.

Correlating Predictions with different algorithms

The following three experiments were presented:

- In [3] small transformer based models (16 layers, 8 hidden dimensions) were trained and evaluated on noiseless data. The correlation of their predictions with different methods was evaluated, and they match OLS most closely.
- In [3] small transformer based models (16 layers, 8 hidden dimensions) were trained and evaluated on noisy datasets while controlling for noise and prior variance levels. The transformer’s predictions seemed to closely match ridge regression whose regularization parameters are consistent with the signal to noise ratio.
- In [5] a decoder is trained for each layer in the model. The decoder has the same architecture as the model’s decoder and is trained to predict the target u from the activations of the layer. The outputs of these decoders are viewed as the results of steps in an iterative algorithm. Their error is plotted as a function of the number of steps and is shown to have a trend that is more similar to Newton than Gradient Descent. They then proceed to correlate the outputs of each layer with the output of each step from Newton and SGD. They found a linear trend between Newton and transformer layers and their best matching Newton step, and exponential pattern for the matching Gradient Descent step.

Confirming that certain properties hold

Several works tried to match their results with known properties of the model they are studying.

- In [4] they check linearity by computing the gradient of the prediction with respect to the unlabeled x , as well as visualize changing the unlabeled x in a random direction.
- In [5] a The convergence of transformers on ill-conditioned priors is examined, and it's shown that it shows a logarithmic trend as a function of curvature. This stands in contrast with the conjecture that the convergence linear methods such as gradient descent exhibit at least polynomial dependence on curvature and further supports the claim that transformer layers behave like second-order optimization methods.

Chapter 3

Empirical Bayes Literature Review

3.1 Definitions and Key Theoretical Results

3.1.1 Description and Famous application

Empirical Bayes refers to techniques that use a Bayesian estimate for the desired quantity, that is it estimates a conditional mean. However, instead of the prior being assumed as is typical in Bayesian methods, relevant aspects of the prior are learned from the data. Hence combining Frequentist and Bayesian perspectives. We refer readers not familiar with it to [18] for a broader exploration of classical applications of this method.

More recently, empirical Bayes based methods were powerful tools for large-scale inference and hypothesis testing [19], estimating genetic effect sizes in genetic association studies [20] and applied econometrics [6].

3.1.2 Bayes Estimator for the Poison Model

In this work, we focus on empirical Bayes for the one-dimensional Poison setting. The Problem was first studied in [21]. In this case we are given $x_1 \dots x_n : x_i \sim Poi(\theta_i)$, that is $P(x = i|\theta) = e^{-\theta} \frac{\theta^i}{i!}$ and θ s are sampled from some unknown prior $\theta_i \sim \pi$. We wish to find $\hat{\theta}_1 \dots \hat{\theta}_n$ such that the expected risk $\mathbb{E}_\pi \left[\sum_{1 \leq i \leq n} (\theta_i - \hat{\theta}_i)^2 \right]$ is as small as possible. If we knew π , denoting $f_\pi(x) = P(X = x|\pi)$ the optimal estimator would be:

$$\hat{\theta}_\pi(x) = \mathbb{E}_{X \sim Poi(\theta), \theta \sim \pi} [\theta | X = x] = \frac{\int \theta e^{-\theta} \frac{\theta^i}{i!}}{\int e^{-\theta} \frac{\theta^x}{x!} d\theta} = (x + 1) \frac{f_\pi(x + 1)}{f_\pi(x)} \quad (3.1)$$

Based on this formula, two flavors of empirical Bayes are popular in the literature:

1. **f-modeling**: Approximate the marginal density f_π directly, then apply 3.1. For example 3.2.1
2. **G-modeling**: First obtain an estimate $\hat{\pi}$ of π from $x_1 \dots x_n$, then apply 3.1. For example 3.2.3 and For example 3.2.2

3.1.3 Regret Lower Bound

No Empirical Bayes method can outperform 3.1. As such, we are typically interested in how much extra error these methods have. This quantity is known as regret, for an estimator \hat{f} it is defined as:

$$\text{TotRegret}_n(\pi) = \mathbb{E} \left[\left\| \hat{f}(x_1 \dots x_n) - (\theta_1 \dots \theta_n) \right\|^2 \right] - n \mathbb{E} \left[\left(\hat{\theta}_\pi(x) - \theta \right)^2 \right] \quad (3.2)$$

In [7] it was shown that Regret for compact priors scales as $\theta \left(\left(\frac{\log n}{\log \log n} \right)^2 \right)$, and as $\Omega(\log^3 n)$ for sub-exponential priors. Based on this, we compare our transformers only to methods that are known to achieve these bounds.

3.1.4 Worst Priors and Golden Standard Estimators

Definition 3.1 (Worst-case prior). Let A be a compact subset of \mathcal{R} . Then the worst-case prior $\pi_{!,A}$ is defined as

$$\pi_{!,A} = \text{argmax}_{\pi \in \mathcal{A}} \text{mmse}(\pi)$$

A sample distribution of the worst-case prior on $[0, 50]$ is illustrated in Figure 1 of [8].

One motivation for using the worst case prior is that the Bayes estimator is that it is minimax-optimal for priors supported on A , making it a “gold standard” estimator.

A concrete statement can be found in the following lemma.

Lemma 3.2. *Let $\hat{\theta}_\pi$ be the Bayes estimator to a prior π . and let A be any compact subset of the reals. Then the least favorable prior $\pi_{!,A}$ of A satisfies the following:*

$$\text{MSE}_{\delta_\theta}(\hat{\theta}_{\pi_{!,A}}) \leq \text{mmse}(\pi_{!,A}), \forall \theta \in A$$

and equality holds whenever $\theta \in \text{Supp}(\pi_{!,A})$.

Proof of 3.2. We consider the prior $\pi_\epsilon \triangleq (1-\epsilon)\pi_! + \epsilon\delta_{\theta_0}$ for some $\theta_0 \in A$. Then $\frac{\partial}{\partial \epsilon} \text{mmse}(\pi_\epsilon)|_{\epsilon=0} \leq 0$ with equality if $\theta_0 \in \text{supp}(\pi_{!,A})$. Consider, now, the following form:

$$\begin{aligned} \text{mmse}(\pi) &= \mathbb{E}[\theta^2] - \mathbb{E}_X[\mathbb{E}[\theta|X]^2] = \mathbb{E}[\theta^2] - \mathbb{E}_X[\mathbb{E}[\theta|X]^2] \\ &= \mathbb{E}[\theta^2] - \sum_x \frac{e_\pi(x)^2}{m_\pi(x)} \end{aligned}$$

where $m_\pi(x) = \int p(x|\theta)d\pi(\theta)$ and $e_\pi(x) = \int \theta p(x|\theta)d\pi(\theta)$ are the PMF and posterior mean of x , respectively.

Now denote $e_{\theta_0}(x) = \theta_0 p(x|\theta_0)$ and $m_{\theta_0}(x) = p(x|\theta_0)$. Denote also the difference $d(x) \triangleq m_{\theta_0}(x) - m_{\pi_!}(x)$ and $k(x) \triangleq e_{\theta_0}(x) - e_{\pi_!}(x)$. Then

$$\text{mmse}(\pi_\epsilon) = \mathbb{E}_{\pi_!}[\theta^2] + \epsilon(\theta_0^2 - \mathbb{E}_{\pi_!}[\theta^2]) - \sum_x \frac{(e_{\pi_!}(x) + \epsilon k(x))^2}{m_{\pi_!}(x) + \epsilon d(x)}$$

which means the derivative when evaluated at 0:

$$\begin{aligned}
0 &\geq \frac{\partial}{\partial \epsilon} \text{mmse}(\pi_\epsilon)|_{\epsilon=0} \\
&= \theta_0^2 - \mathbb{E}_{\pi_1}[\theta^2] - \sum_x \frac{2m_{\pi_1}(x)e_{\pi_1}(x)k(x) - e_{\pi_1}(x)^2d(x)}{m_{\pi_1}(x)^2} \\
&= \theta_0^2 - \sum_x \frac{2m_{\pi_1}(x)e_{\pi_1}(x)e_{\theta_0}(x) - e_{\pi_1}(x)^2m_{\theta_0}(x)}{m_{\pi_1}(x)^2} \\
&\quad - \text{mmse}(\pi_1) \\
&= \theta_0^2 - 2 \sum_x e_{\theta_0}(x) \frac{e_{\pi_1}(x)}{m_{\pi_1}(x)} + \sum_x m_{\theta_0}(x) \left(\frac{e_{\pi_1}(x)}{m_{\pi_1}(x)} \right)^2 \\
&\quad - \text{mmse}(\pi_1) \\
&= \text{mse}_{\delta_\theta}(f_{\pi_1}) - \text{mmse}(\pi_1)
\end{aligned}$$

where the last equality follows from that $\frac{e_{\pi_1}(x)}{m_{\pi_1}(x)} = f_{\pi_1}(x)$. \square

Corollary 3.3. *For any compact subset A of the reals, we have*

$$\min_{\hat{\theta}} \max_{\pi \in \mathcal{P}(A)} \mathbb{E}[(\hat{\theta}(X) - \theta)^2] = \text{mmse}(\pi_{1,A})$$

achieved by the Bayes estimator $f_{\pi_{1,A}}$ of the least favourable prior, $\pi_{1,A}$.

Proof. From 3.2, we have $\text{MSE}_\pi(\hat{\theta}_{\pi_{1,A}}) \leq \text{mmse}(\pi_1)$ for any $\pi \in \mathcal{P}(A)$. Therefore:

$$\min_{\hat{\theta}} \max_{\pi \in \mathcal{P}(A)} \mathbb{E}[(\hat{\theta}(X) - \theta)^2] \leq \text{mmse}(\pi_1)$$

by taking $\hat{\theta} = \hat{\theta}_{\pi_{1,A}}$. Now, for any $\hat{\theta}$, we have $\mathbb{E}_{\pi_{1,A}}[(\hat{\theta}(X) - \theta)^2] \geq \text{mmse}(\pi_1)$. Therefore the conclusion follows. \square

However, this estimator $f_{\pi_{1,A}}$ does not leverage the low-MMSE nature of some prior, leading to suboptimal regret produced by $f_{\pi_{1,A}}$.

3.2 Regret Optimal Poisson Estimators

3.2.1 Robbins

Robbin's is an intuitive f-modeling estimator, it depends on the fact that for a sufficiently large number of observations n . we have $N_\pi(x) \approx n f_\pi(x)$, where $N_\pi(x)$ is the number of times x was observed in the sample. Hence it approximates 3.1 as follows:

$$\hat{\theta}_{\text{robbins}}(x) = (x+1) \frac{N_\pi(x+1)}{N_\pi(x)} \tag{3.3}$$

While this meets the asymptotic regret lower bound stated in 3.1.2, it suffers from the following problems in practice:

- For large values of x , $N_\pi(x)$ tends to be noisy making the ratio unstable
- If $N_\pi(x + 1) = 0$ the prediction of 0 which is unreasonable, requiring some practical adjustments
- It does not retain the monotonicity of Bayes Estimators

3.2.2 NPMLE

Non-parametric maximum likelihood (NPMLE) estimates a none non-parametric prior that maximizes likelihood under the Poisson mixture.

$$\hat{\pi}_{npml e} = \underset{\pi}{\operatorname{argmax}} \prod_{1 \leq i \leq n} f_\pi(x_i) \quad (3.4)$$

$$\hat{\theta}_{npml e}(x) = (x + 1) \frac{f_{\hat{\pi}_{npml e}}(x + 1)}{f_{\hat{\pi}_{npml e}}(x)}$$

(3.5)

This method was proven regret optimal in [8], and it usually has a good empirical performance. It also retains the monotonicity of the Bayesian Method. However, its main drawback is that it tends to be slow in practice. Making it prohibitively expensive on high-dimensional or large datasets.

3.2.3 ERM

Unlike f-modeling or g-modeling, erm estimates $\hat{\theta}_{em}$ by solving a suitable empirical risk minimization problem subject to certain structural constraints satisfied by the Bayesian oracle. In the case of the Poisson model, it solves the following optimization over the class of monotone functions:

$$\hat{\theta}_{em} = \underset{f}{\operatorname{argmin}} \mathbb{E} [f^2(x) - 2xf(x - 1)] \quad (3.6)$$

The target of the optimization is based on the following property of the Bayesian Oracle.

$$\mathbb{E} \left[\left(\hat{\theta}_\pi(x) - \theta \right)^2 \right] = \mathbb{E} \left[\hat{\theta}_\pi^2(x) - 2\hat{\theta}_\pi(x)\theta \right] + \mathbb{E} [\theta^2] = \mathbb{E} \left[\hat{\theta}_\pi^2(x) - 2\hat{\theta}_\pi(x - 1)x \right] + \mathbb{E} [\theta^2] \quad (3.7)$$

In [9] it is proved that these properties uniquely define $\hat{\theta}_{em}$, and it is also proved that this estimator is regret optimal. An isotonic regression based algorithm for doing the optimization is provided, and it's shown to be more efficient than NPMLE while retaining its desirable practical properties.

Chapter 4

Proposed Transformer Based Empirical Bayes Estimator

4.1 Problem Setting

Let $\mathcal{P}([0, \theta_{\max}])$ be a set priors supported on $[0, \theta_{\max}]$, and let $f_{\mathcal{P}}$ be a distribution over those priors. As depicted in 4.1, each transformers model T is trained to predict $\theta_1 \dots \theta_n \sim f_{\mathcal{P}}$ from $x_1 \dots x_n$, and is trying to minimize the following loss function:

$$\mathbb{E}_{\substack{\pi \sim f_{\mathcal{P}} \\ \theta_1, \dots, \theta_n \sim \pi \\ x_i \sim \text{Poi}(\theta_i)}} [\|T(x) - \theta\|^2] \tag{4.1}$$

We use $f_{\mathcal{P}}$ instead of a single prior to prevent the model from overfitting to a specific prior. We chose $\theta_{\max} = 50$, and $n = 512$ (sequence length) in the training process. We discuss our choice of $f_{\mathcal{P}}$ in 4.3

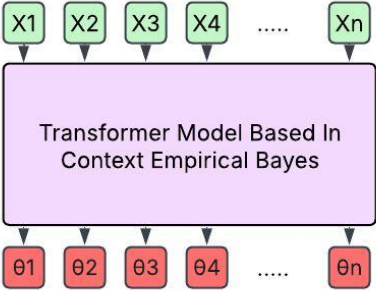


Figure 4.1: In Context Empirical Bayes Problem Structure. The goal is to predict $\theta_1 \dots \theta_n$ given $x_1 \dots x_n$ such that $x_i \sim \theta_i$

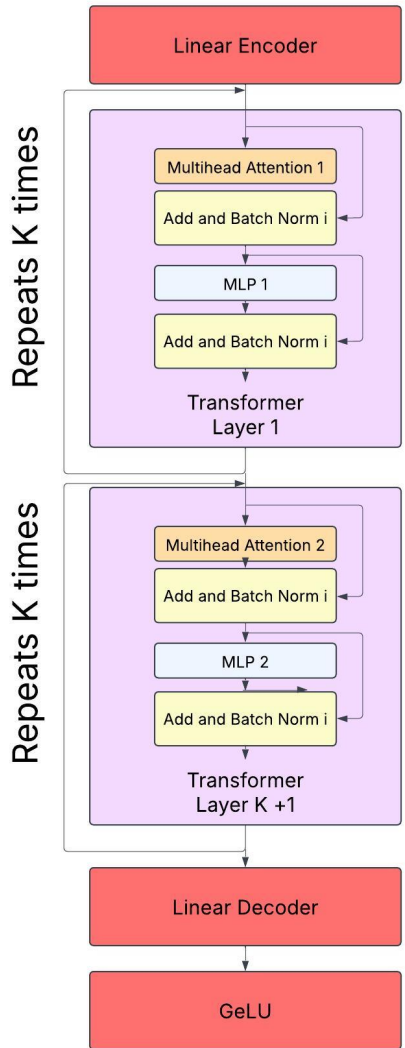


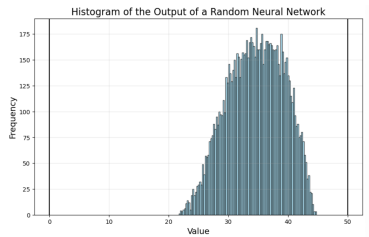
Figure 4.2: Chosen Transformer Model Architecture with layers $L = 2k$. The first half of the layers share all weights except for normalization, and so does the second half.

4.2 Architecture Description

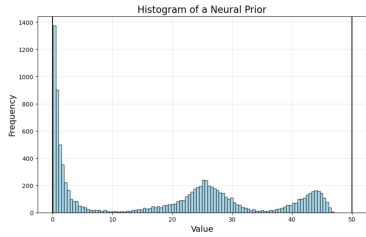
We chose a simple linear encoder, we also use a simple decoder comprised of a linear layer followed by *GeLU* activation. We add GeLU since Empirical Bayes output should be positive. Additionally, to avoid making our models too big we focus on models that have two distinct layers, each repeated $\frac{L}{2}$, these models share all their weights except for normalization layers. This keeps the number of parameters less than 30k. The intuition behind it is that one learns the encoding part (input) and the other the decoding part (output). Our intuition is corroborated in 4. These choices are depicted in 4.2.

Other choices, include our MLPs having two layers and either *GeLU* or *ReLU* activations. All our intermediate layers have input dimension d , except for the intermediate layer of the MLP which has input dimension $4d$.

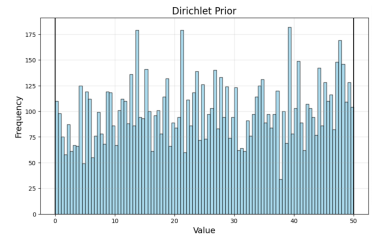
Prior Distributions, θ



(a) One Random Network

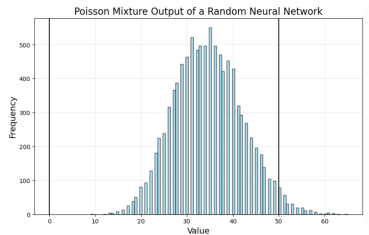


(b) Neural Prior

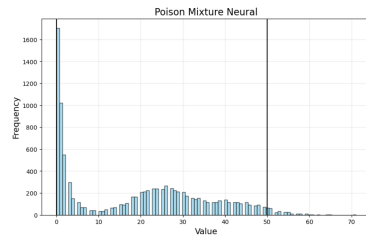


(c) Dirichlet

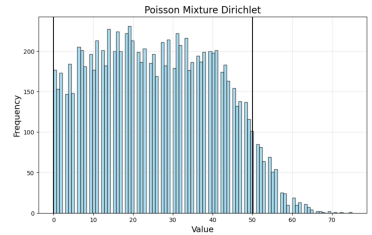
Poisson Mixtures, x



(d) One Random Network



(e) Neural Prior



(f) Dirichlet

Figure 4.3: Top row shows the distribution of θ , the model’s target, for different priors. The bottom row shows the Poisson mixtures, which are the model’s inputs. We note that for a single random network, θ and x appear clustered. For Neural Priors θ exhibits a few separated clusters, but they are smeared in x . With Dirichlet θ it varies chaotically in a none-continuous manner, after applying Poisson it becomes smoother

4.3 Training Data Distribution

4.3.1 Neural Prior

To sample $\pi \sim f_{\mathcal{P}}$, we generate four random neural networks, each with two layers. The input of the first is 4×4 dimensional, and the second is 4×1 dimensional. Weights are initialized according to PyTorch’s default, and the activation function is chosen at random from the list below, while the final activation is always a sigmoid.

$$GELU, ReLU, SELU, CELU, SiLU, Tanh, TanhShrink.$$

To sample $\theta \sim \pi$ we four random numbers in $[0, 1]$, and feed them, through one of the four random neural networks (chosen uniformly at random). A typical histogram of such a prior is shown in 4.3

4.3.2 Dirichlet Distribution

We consider a Dirichlet Process, which tends to produce repeated values that are not dependent on the structure of real numbers (not smooth). We take the uniform distribution as our

base distribution defined as $H_0 \triangleq \text{Unif}([0, \theta_{max}])$. Within each batch the elements $\theta_1, \dots, \theta_s$ are generated as follows:

$$\theta_j = \begin{cases} \theta_i & \text{w.p. } \frac{j-1}{\alpha+j-1}, \forall i = 1, \dots, j-1 \\ x \sim H_0 & \text{w.p. } \frac{\alpha}{\alpha+j-1} \end{cases}$$

Where α is a parameter that denotes how ‘close’ we are to iid generation ($\alpha = \infty$ essentially means we have iid). We use $\alpha = 50$ for a sequence length of 512. Note that the Dirichlet process implies that our data is not generated iid for each batch. Priors are compared in 4.3

4.4 Evaluation Metrics

As mentioned in 3.1.3, Empirical Bayes estimators are evaluated by their excess loss over the Bayes oracle, regret decays quickly for good estimators. Raising a concern, that if we simply pick the model with the lowest MSE, we might be losing the model’s regret in the mix due to noise. As such, we consider three alternatives when selecting model parameters. All three criteria were agreed in our experiments.

4.4.1 MSE paired T-Test

We sample $\pi_1 \dots \pi_n \sim f_{\mathcal{P}}$, then for each prior we sample b different Empirical Bayes estimates. Compute the MSE of each of the two models on each of the batches. Then run a paired T-Test between the MSES. The Paired test has the advantage of subtracting away the Bayes Oracle part of MSE. However, it could still be sensitive to its scale, weighing different priors differently.

4.4.2 Plackett-Luce Rank Model for Rank Aggregation

We remedy the sensitivity to scale in the previous test by ranking the models on each test, then use the Plackett-Luce model to aggregate the rankings. The Plackett-Luce model is used to rank LLMs on subjective attributes in [Chatbot Arena](#). We estimate the coefficients of the Plackett-Luce model using gradient descent as suggested in [22], and provide our implementation in B.2 .

Recall that the Plackett-Luce model assigns each object x coefficients β_x . Then generates the order iteratively, selecting the next element from the set of elements that were not selected yet, S , according to the following equation:

$$P[a \text{ is next }] = \frac{e^{\beta_a}}{\sum_{x \in S} e^{\beta_x}} \tag{4.2}$$

Note that for this optimization to be convex, no algorithm can strongly dominate the other rankings. That is, if we draw an edge from algorithm A to algorithm B if there’s a ranking where A ranks higher than B , then the graph must be strongly connected for the problem to be convex. This was not always true in our experiments. Hence, our implementation first breaks this graph into strongly connected components, and then solves the convex

problem in each. It uses the additive invariance of the problem (can add a constant to all β s) to create a gap between the connected components. The gap is larger than twice any of the gaps that naturally occur to make the separation between components clear.

4.4.3 Regret estimates

We also compute the regret of some transformers. To compute the regret concerning a prior π , we need to subtract its Bayes Oracle loss from its MSE. Estimating the Bayes Oracle loss is computationally expensive, so we only do it for a few models depending on the results of the two metrics above.

4.5 Selected Models

4.5.1 Chosen Training Prior

We note that neural priors appear somewhat smooth. Since real-world data often has outliers, we look into training on a mixture of neural priors and the Dirichlet Priors. We train our transformers so that each of the 192 Empirical Bayes instances is sampled from Dirichlet Prior with probability .5 and from a fixed neural prior with probability .5. We believe that having both types of priors in each batch helps stabilize the training process.

In the table 4.1 we provide t-stats of networks trained on a mixture against the networks trained on each prior when evaluated on the prior. Naturally, each model is the best on its training dataset. However, the table below suggests that transformers trained on the mixture recover most of the difference in performance between the models if they were evaluated on each other priors. The effect is clearer in the regret plot below 4.4. This, and our belief that we need to be able to be robust outliers guided our selection of the mixture.

Table 4.1: Table of regret difference; $A - B$ denotes the difference of regret of transformers trained on A vs trained on B

	Evaluated on Neural		Evaluated on Dirichlet	
# lyr	mix - neu	dir - mix	mix - dir	neu - mix
12	0.0038	0.8645	0.0184	0.0379
18	0.0133	1.0647	0.0173	0.0469
24	0.0082	1.0021	0.0202	0.0388

4.5.2 Selected Hyper-parameters

We consider models of 6, 12, 18, 24, and 48 layers, embedding dimension `dmodel` either 32 or 64, and number of heads in $\{4, 8, 16, 32\}$. We fix the number of training epochs to 50k, the decay rate every 300 epochs to 0.9, and learning rate of 0.02. Among the trained models, the models are selected based on the mean-squared error evaluated on neural prior-on-prior and Dirichlet process during inference time. We then arrive at the four models described in

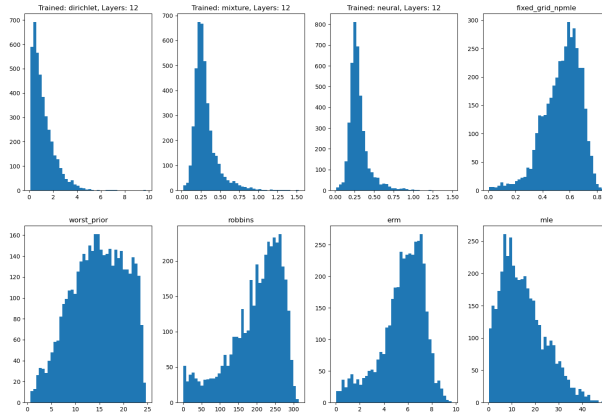


Figure 4.4: Histogram of regret on neural prior for different estimators. We can see that the Dirichlet-only models performs poorly, though it still outperforms Robbins and worst prior. We also see that the performance drop from adding Dirichlet to the training data is small.

4.2, which we will name T18, T24 T18r, and T24r depending on the number of layers, and there θ_{max} is random. We focus our evaluation on T18, T24.

Name	Layers	Dmodel	# Heads	θ_{max}
T18	18	32	16	50
T24	24	32	8	50
T18r	12	32	16	random [10,150]
T24r	24	32	8	random [10,150]

Table 4.2: The hyperparameters for selected models

Chapter 5

Expressibility of Transformers

5.1 Setup

In line with 2.3.1, we provide transformer based models that can approximate a variant of Robbin’s to arbitrary precision. Similar to previous work, we focus on minimizing the number of transformer layers needed, as there are universal approximation results. To simplify the exposition, we slightly modify the structure in 2.1 . Though it is possible to provide a similar result without doing so. Our modifications include removing the normalization layers and removing the skip connection after MLP. The modified structure is depicted in 5.1

We construct transformers that approximate a variant of Robbins’s estimator called Clipped Robbins’s defined by the equation below.

$$\hat{\theta}_{\text{Rob}(d,g)}(x) = \begin{cases} \min(\{\hat{\theta}_{\text{Rob}}(x), g\}) & 0 \leq x < d \\ g & x \geq d \end{cases} \quad (5.1)$$

5.2 Dimension Inefficient Construction

Theorem 5.1. *There is a single layer transformer based model with hidden dimension $d + 2$ that can approximate $\hat{\theta}_{\text{Rob},d,g}(x)$ to arbitrary precision.*

Proof. **Encoder**

We use the first d dimensions as a one-hot encoding of the input if it’s in $[0, d - 1]$, otherwise, we propagate its value in the last dimension, as described below.

$$\text{Encoder}(x) = \begin{cases} e_x & 0 \leq x < d \\ (0, \dots, 1, \frac{g}{1+g}) & x = d \\ (0, \dots, 0, \frac{g}{1+g}) & \text{otherwise} \end{cases} \quad (5.2)$$

Attention

Recall that attention is parametrized by matrices V, K, Q and produces the following output. (Where X is a matrix with the embeddings stacked)

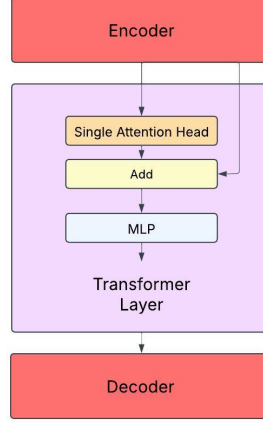


Figure 5.1: The Structure used for transformer based models that implement clipped Robbins estimator

$$Attention(X) = VX \text{Softmax}(X^T Q^T KX / \sqrt{d+2})$$

We set the parameters as follows, in terms of a large number $D \gg \text{MaxSequenceLength} \geq n$

$$V = \begin{bmatrix} -I_{d+1} & 0 \\ 0 & 0 \end{bmatrix}, \quad Q = I_{d+2}, \quad K_{i,j} = \begin{cases} D & i = j < d, \\ D + \sqrt{d+2} \log i & i = j + 1 \leq d, \\ 1 & i = j = d, \\ 0 & \text{otherwise.} \end{cases} \quad (5.3)$$

Let $M = \text{Softmax}(X^T Q^T KX / \sqrt{d+1})$ be the matrix of the attention weights.

$$M_{ij} \approx \begin{cases} \frac{1}{N(X_j) + (X_j+1)N(X_j+1)} & X_i = X_j < d \\ \frac{X_j+1}{N(X_j) + (X_j+1)N(X_j+1)} & X_i = X_j + 1 \leq d \\ \text{positive} & X_i \notin [0, d-1] \& X_j \notin [0, d-1]. \\ 0 & \text{otherwise} \end{cases}$$

This means that after Attention and the skip connections, The embedding of each token takes the following form:

$$Attention(Encode(X))_i \approx \begin{cases} \left(0, \dots, \overbrace{\frac{(X_i+1)N(X_i+1)}{N(X_i) + (X_i+1)N(X_i+1)}, \frac{-(X_i+1)N(X_i+1)}{N(X_i) + (X_i+1)N(X_i+1)}}^{\text{Indexes } x_i, x_{i+1}}, \dots, 0\right) & X_i = X_i < d \\ \left(0 \dots, 0, a, \frac{g}{1+g}\right) : a \geq 0 & \text{otherwise} \end{cases} \quad (5.4)$$

MLP

First, we project the embedding into the following basis $\{e_i - e_{i+1}\}_{i \leq d} \cup \{e_d, e_{d+1}\}$, then we drop the coordinate associated with e_d . At this point, the embedding should have coordinates all 0, except for one that is $\frac{\hat{\theta}_{\text{Rob}(d,g)}}{\hat{\theta}_{\text{Rob}(d,g)} + 1}$, Hence, if we add up its coordinates and subtract the

sum from one we should get $\frac{1}{\hat{\theta}_{\text{Rob}(d,g)}+1}$. Note that this is a linear transform and can be combined into one linear layer. .

To produce the output, we need to compute the continuous compact function $f : [0, 1] \rightarrow [0, g]$:

$$f(x) = \begin{cases} \frac{1}{x} - 1 & x \geq \frac{1}{1+g} \\ g & \text{otherwise} \end{cases}$$

Since it's continuous and compact, it can be approximated up to arbitrary precision by an MLP with classical activation functions. Alternatively, we could let the decoder compute this function, or use it as our activation function. □

5.3 More Efficient Use of Attention Dimensions

In the mechanistic interoperability literature for LLMs, it is believed that transformers exhibit superposition of features [23]. Where a set of nearly orthogonal vectors is used to express different concepts. We try to use this concept to use attention dimensions more efficiently.

Theorem 5.2. *There is a single-layer transformer based model with hidden attention dimension $O(\log d)$, that can approximate $\hat{\theta}_{\text{Rob},d,g}(x)$ to arbitrary precision.*

proof sketch. **New Encoder**

We replace the vectors $e_0 \dots e_d$ with nearly orthogonal vectors of $d_{\text{atten}} - 1$ dimensions, we call them $\tilde{e}_0, \dots \tilde{e}_d$, we replace e_{d+1} with $\tilde{e}_{d_{\text{atten}}}$ (which is orthogonal to all other vectors). The encoder assigns each number the same linear combination of corresponding vectors as before. Note the following:

$$\tilde{e}_i \cdot \tilde{e}_j = \begin{cases} \epsilon & i \neq j \\ 1 & i = j \end{cases}$$

$$\tilde{e}_i \cdot \tilde{e}_{d_{\text{atten}}-1} = 0 \forall i < d_{\text{atten}} - 1$$

Additionally, note that by the Johnson Linderstrauss lemma, we can set $d_{\text{atten}} = O(\frac{\log d}{\epsilon^2})$

New Attention Parameters

Then we change K in the attention mechanism to the following matrix.

$$\tilde{K} = \tilde{e}_{d_{\text{atten}}+1} \tilde{e}_{d_{\text{atten}}+1}^T + \sum_{0 \leq i \leq d-1} e_{i+1} \tilde{e}_i^T \left(D + i \sqrt{d_{\text{atten}} + 1} \right) + \sum_{0 \leq i < d} D \tilde{e}_i \tilde{e}_i^T \quad (5.5)$$

We can verify the products of the form $Encode(x_i)^T K Encode(x_j)$ are approximately the same as before. Hence, our input to the MLP is approximately the same linear combination of the encoding of different numbers, and if we can recover it with a few MLP layers the rest can stay the same.

New MLP Parameters

Since the dot products are almost 0, we simply add a single layer whose weights project on each of the vectors $\tilde{e}_0, \dots, \tilde{e}_{d-1}$, as well as $e_{d_{attn}}$. This $d+1$ dimensional vector should look approximately equal to last time, and we can use the rest of the layer as is. (It is also possible to use a combination of ReLU and linear layers to reduce the noise level after projecting). \square

5.4 Transformers Can Get Arbitrary close to asymptotically Regret Optimal

Theorem 5.3. *For any $\epsilon > 0$, there exists N and a single layer transformer network Γ such that for all $n \geq N$, the minimax regret of $\Gamma(X_1, \dots, X_n)$ on prior $\pi \in \mathcal{P}([0, \theta_{\max}])$ satisfies*

$$\sup_{\pi \in \mathcal{P}([0, \theta_{\max}])} \text{Regret}(\Gamma(X_1, \dots, X_n)) \leq \epsilon$$

Proof. Choose d such that $\mathbb{P}[X > d] < \frac{\epsilon}{6 \cdot \theta_{\max}^2}$, and $g = \theta_{\max}$. Note that there exists an N such that for $n \geq N$, The Robbins estimator [24] enjoy a minimax regret bounded by $\frac{\epsilon}{6}$ over the class $\mathcal{P}([0, \theta_{\max}])$. Now, by the previous theorem, there exists a single layer transformers model Γ that can approximate either Robbins to $\sqrt{\frac{\epsilon}{6}}$ precision uniformly for inputs up to d . Then we have

$$\begin{aligned} \text{Regret}(\Gamma) &\leq 2(\text{Regret}(\hat{\theta}) + \mathbb{E}[(\hat{\theta} - \Gamma)^2]) \\ &\leq 2\left(\frac{\epsilon}{6} + \mathbb{E}[(\hat{\theta}(X) - \Gamma(X))^2 \mathbb{1}_{X \leq d}] + \mathbb{E}[\theta_{\max}^2 \mathbb{1}_{X > d}]\right) \\ &\leq 2\left(\frac{\epsilon}{6} + \frac{\epsilon}{6} + \frac{\epsilon}{6}\right) \\ &= \epsilon \end{aligned}$$

\square

Chapter 6

Evaluation

6.1 Out of Distribution Synthetic Evaluation

6.1.1 Changing The Sequence Length

In this experiment, we evaluate the ability of transformers to adapt to different sequence lengths, both fewer than and more than what is trained. For each sequence length in 128, 256, 512, 1024, and 2048 we generate 4092 random neural priors as described in 4.3.1, then generate batches of 192 instances of the desired sequence length from each. We report the average regret over the 4096 priors in 6.1.

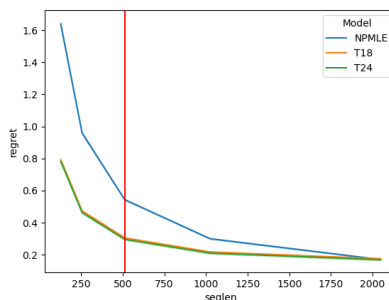


Figure 6.1: Regret vs sequence length. The regret decreases for both transformers as the sequence length increases, showing that they do have the ability to generalize. We nevertheless note that NPMLE has a better generalization ability, as shown by the regret at sequence length 2048 as compared to smaller sequences. In comparison, the average regret for ERM monotone is 11.20, 8.19, 5.58, 3.66, 2.36 for the various sequence lengths, while the average regret for MLE and worst prior Bayes stays constant at 14.816 and 14.658, respectively

6.1.2 Unseen priors: Worst Prior

Our transformers are trained on a mixture of neural and Dirichlet priors. Here, we consider their performances on the worst case prior in $\mathcal{P}([0, 50])$, explained in 3.1.4. Similar to the

previous section, we also vary the sequence length. The number of batches we use in this prior is 786k for sequence length $n = 128, 256, 512, 393k$ for $n = 1024$, and 197k for $n = 2048$. We report the estimated regret in 6.2

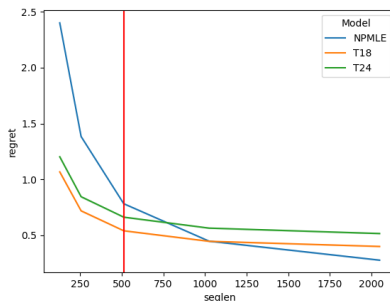


Figure 6.2: Worst Prior SeqLen Regret of various transformers on worst prior as compared to NPMLE. Again, transformers show the ability to generalize to longer sequence lengths, although for longer sequences NPMLE generalizes better. In comparison, this is already better than ERM-monotone’s regret at 12.79, 9.80, 6.99, 4.81, and 3.256421 across the 5 sequence lengths, while MLE’s regret stays at 11.73.

6.1.3 Cost of being robust to Support

To make our transformers more robust to changes in the support of the prior, we propose sampling θ_{max} from the following mixture.

$$\theta_{max} \sim \frac{3}{4}\mathcal{N}(0, 50, 10^2) + \frac{1}{8}\text{Exp}(50) + \frac{1}{8}\text{Cauchy}(50, 10)$$

and clamped at $[10, 150]$. Then, for the two sets of transformers, we evaluate them on 4096 neural prior-on-priors, using the default sequence length = 512 and 192 batches for each prior. We report the distribution of regrets in 6.3 that demonstrates that transformers trained with randomized θ_{max} see a small deterioration in regret, nonetheless still outperform NPMLE in regret minimization.

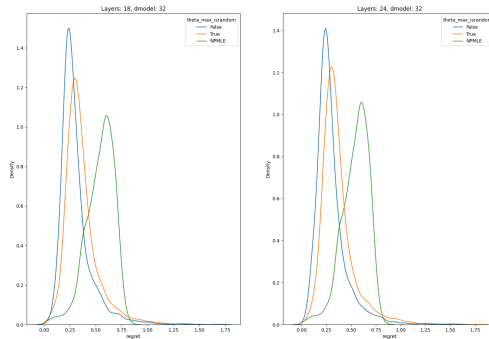


Figure 6.3: Training on a randomized θ_{\max} increases mean regret by 18.5% and 23.4% for the transformers with 18 and 24 layers, respectively, when compared against transformers that learn the true θ_{\max} during training. The corresponding t-stat are 32.99 and 36.16.

6.1.4 Summary of Synthetic Evaluation

Table 6.1: Plackett-Luce Rankings on Synthetic Experiments

EXPERIMENTS	GS	ROBBINS	ERM	NPMLE	T18	T24
NEURAL-128	-0.003	-3.196	0.965	4.310	7.497	7.696
NEURAL-256	-0.023	-3.090	1.678	4.885	7.624	8.002
NEURAL-512*	-0.044	-3.016	2.421	5.534	7.646	8.084
NEURAL-1024	-0.066	-2.930	3.032	6.197	7.579	7.983
NEURAL-2048	-0.092	-2.813	3.430	6.806	7.455	7.816
WP-128	-	-4.925	-2.434	2.476	7.416	4.945
WP-256	-	-2.470	2.470	4.943	9.878	7.412
WP-512	-	-2.466	2.463	4.924	9.842	7.385
WP-1024	-	-2.738	2.735	8.543	8.664	5.476
WP-2048	-	-2.468	2.470	9.863	7.405	4.938
MULTN-512	0.505	-2.664	2.239	4.877	9.686	7.339
MULTN-1024	0.463	-2.635	3.328	5.764	9.615	8.240
MULTN-2048	0.471	-2.728	3.432	5.963	8.971	8.811

Table 6.2: T -stat of T18 against classical algorithms (positive = improvement)

EXPERIMENTS	MLE	GS	ROBBINS	ERM	NPMLE
NEURAL-128	96.859	157.656	151.200	134.131	159.608
NEURAL-256	98.918	162.630	161.487	163.570	135.429
NEURAL-512*	100.009	165.198	179.252	188.449	88.431
NEURAL-1024	100.559	166.455	195.615	207.876	35.623
NEURAL-2048	100.819	167.038	206.403	218.789	-2.233
WP-128	2333.931	-	4497.921	2358.189	726.271
WP-256	2984.407	-	3628.320	2518.225	631.805
WP-512	3491.789	-	3401.316	1941.935	401.003
WP-1024	2582.811	-	2359.364	1918.689	3.193
WP-2048	1938.137	-	1690.330	1446.296	-316.804
MULTN-512	920.850	893.271	2272.100	2614.887	329.538
MULTN-1024	933.207	903.482	2524.041	3057.655	248.420
MULTN-2048	932.016	912.076	2186.958	2470.924	773.466

Table 6.3: T -stat of T24 against classical algorithms (positive = improvement)

EXPERIMENTS	MLE	GS	ROBBINS	ERM	NPMLE
NEURAL-128	96.979	157.212	151.218	134.445	171.786
NEURAL-256	99.013	162.393	161.503	163.939	145.136
NEURAL-512*	100.075	165.063	179.265	188.733	94.223
NEURAL-1024	100.604	166.374	195.619	207.913	39.106
NEURAL-2048	100.852	166.984	206.386	218.436	0.605
WP-128	2300.725	-	4497.084	2312.319	642.975
WP-256	2945.563	-	3627.207	2467.594	501.248
WP-512	3452.230	-	3399.734	1905.039	194.181
WP-1024	2557.278	-	2357.378	1859.728	-238.874
WP-2048	1918.384	-	1687.892	1381.589	-583.587
MULTN-512	920.237	893.098	2272.033	2613.866	327.396
MULTN-1024	933.200	903.982	2523.892	3055.446	247.394
MULTN-2048	932.490	912.951	2186.828	2471.843	796.633

6.2 Runtime

To illustrate the advantages of transformers in comparison with NPMLE, we evaluate their inference time. We do so using 4096 neural prior-on-priors, where for each prior we consider the time needed to estimate the hidden parameter of 192 batches and sequence length 128, 256, 512, 1024, and 2048. Each program is given 2 Nvidia Volta V100 GPUs and 40 CPUs for

computation. The results are tabulated at 6.4, where we see that the transformers’ runtime is comparable to that of ERM’s.

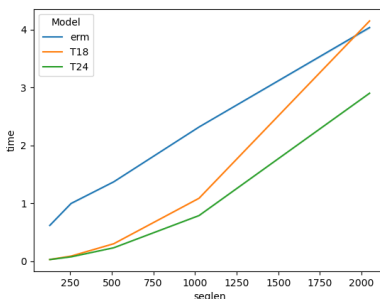


Figure 6.4: Time vs sequence length (labels in the form of (layers, model dimension, head), showing that the inference time of transformers is comparable with that of ERM. We nevertheless qualify this finding by noting that there seems to be a running time that scales super-linearly with sequence length. For comparison, the running time of NPMLE for sequence lengths 128, 256, 512, 1024, and 2048 are 41.69, 67.70, 109.81, 175.72, and 289.79 seconds, respectively, which indicates that transformers are 2 orders of magnitudes faster than NPMLE given the GPU.

6.3 Evaluation on Real Data

In this section, we demonstrate that transformers that are pre-trained only on synthetic data perform well on real datasets without re-training on any part of the dataset. Since we do not have access to the prior in-read data, We use the following experimental setup:

Given a countable attribute (eg: the number of goals by a specific soccer player). We let X be the count of the attribute in the initial section of the data, and Y be the count of the attribute in the remaining section. We evaluate how well different Empirical Bayes methods can predict Y , as we expect estimators that predict the correct conditional hidden parameter to better predict $\hat{Y} = \hat{\theta}(X)$.

We focus on two types of datasets: Sports and word frequency. We start by describing the datasets that we study and then tabulate the results at the end.

6.3.1 Sports datasets

Here, X and Y are the number of goals scored by a player within disjoint and consecutive time frames, and θ represents the innate ability of the given player. We will consider two datasets: National Hockey League (NHL) and Major League Baseball (MLB).

National Hockey League (NHL) dataset

. We proceed in the same spirit as [25], Section 5.2, and study the data on total number of goals scored by each player in the National Hockey League for 29 years: The first was

the 1989-1990 season and the last is the 2018-2019 season (2004-2005 season was canceled). The data is obtained from <https://www.hockey-reference.com/>, and we focus on the skater's statistics. Here, given the number of goals a player scored in season j , we wish to predict the same for season $j + 1$ (thus the input and label sets are the number of goals a player scored in consecutive seasons). Apart from fitting the statistics for *all* players at once, we also study what happens if we only fit the number of goals scored by players of a particular position. There are three positions of interest: defender, center, and winger.

Major League Baseball (MLB)

. The dataset is publicly available at <https://www.retrosheet.org/game.htm>, processed by <https://github.com/Zmalski/NHL-API-Reference>. Similar to the hockey dataset, we are interested in the hitting count of each player. However, rather than between-season prediction, we do in-season prediction. That is, we take X as the number of goals scored by a player in the beginning portion of the season and Y in the rest of the season. For batting and pitching players (which we fit separately), we use X as the goals in the first $\frac{1}{5}$ and first $\frac{1}{6}$ of the season, respectively. In other words: here we will predict $\hat{Y} = 4\hat{\theta}(X)$ and $\hat{Y} = 5\hat{\theta}(X)$ for each of the respective cases.

6.3.2 Word frequency datasets

In this setting, we model the alphabet of tokens as M categorical objects $A = \{A_1, \dots, A_M\}$. Given n samples from these objects, denote X_1, \dots, X_M the frequency of the samples. We want to predict the following: if we are to observe t more samples (t known), and let Y_1, \dots, Y_M be the frequency in the next t samples. We are interested in minimizing the mean square error of $\sum (\hat{Y}_i - Y_i)^2$ where \hat{Y}_i is our predicted next frequencies of A_i .

To do so, we model as follows: consider p_1, \dots, p_M as the "inherent" probability distribution over M (or proportion in a population), so $\sum_{i=1}^M p_i = 1$. In our empirical Bayes setting, suppose also that there is an unknown π such that $p_i \stackrel{\text{iid}}{\sim} \pi$ (in this case we won't have $\sum p_i = 1$, but should be "close enough" when M big; all we need is just $\mathbb{E}[\pi] = \frac{1}{M}$). Now the frequency $X_i \sim \text{Binom}(n, p_i)$. Given that n, M are big (hence each θ_i are likely to be small), we may instead approximate these as $X_i \sim \text{Poi}(np_i)$. Thus we may use empirical Bayes method to estimate $\hat{\theta}_i$ based on the frequencies X_1, \dots, X_M , and then predict $\hat{Y}_i = t\hat{\theta}_i$.

BookcorpusOpen

. This is a well-known large-scale text dataset, originally collected and analyzed by [26]. Here, we use the BookcorpusOpen, hosted on websites like <https://huggingface.co/datasets/lucadiliello/bookcorpusopen>. This version of the dataset contains 17868 books in English; we discard 3 of the books that are too short (< 2000 tokens) and 5 other books where NPMLE incurs out-of-memory error. To curate the dataset, we first tokenize the text using scikit-learn's CountVectorizer with English stopwords removed, which results in around 10000 tokens. For each book, the input set comprises the beginning section containing approximately 2000 tokens, while the label set is the remainder of the book. Then for each word, X and Y are the frequency of each word within the input and label set, respectively.

We will then use the prediction $\hat{Y} = c \cdot \hat{\theta}(X)$ where c is the ratio of several sentences in the label to input.

6.3.3 Result Summary

We use the RMSE of each dataset item as our main evaluation metric. Specifically, for each dataset, we will compute the RMSE incurred by each estimator. We then compare them using the following guidelines.

Comparison against MLE. We consider the ratio of RMSE of each estimator against the MLE, and ask, “how much improvement did we achieve against the MLE” by looking at the *average* of the ratio.

Relative ranking. We use the Plackett-Luce rank aggregation system described in [4](#)

Significance of improvement. We will also consider whether one improvement is *significant* by looking at the t -stats of the RMSE. Our findings are tabulated below.

Transformers significantly outperform other methods across the board. With deeper (24 layers) one performs better on all datasets except for NHL, where the 18-layer one performs better. See results summary below [6.4](#) [6.5](#) [6.6](#).

Table 6.4: Relative Improvement of each algorithm over MLE in MAE metric, 95% confidence interval.

Dataset	Robbins	ERM	NPMLE	Transformers18	Transformers24
Hockey (all)	-0.2014 \pm 0.0444	-0.0020 \pm 0.0060	0.0090 \pm 0.0058	0.0076 \pm 0.0058	0.0077 \pm 0.0059
Hockey (defender)	-0.1338 \pm 0.0424	0.0144 \pm 0.0116	0.0326 \pm 0.0107	0.0366 \pm 0.0123	0.0362 \pm 0.0122
Hockey (center)	-0.4143 \pm 0.0921	-0.0065 \pm 0.0086	0.0226 \pm 0.0082	0.0287 \pm 0.0086	0.0276 \pm 0.0086
Hockey (winger)	-0.3243 \pm 0.0723	-0.0012 \pm 0.0078	0.0143 \pm 0.0063	0.0177 \pm 0.0067	0.0168 \pm 0.0067
Baseball (batting)	-0.2597 \pm 0.0381	0.0350 \pm 0.0032	0.0522 \pm 0.0036	0.0560 \pm 0.0034	0.0553 \pm 0.0036
Baseball (pitching)	-0.1674 \pm 0.0219	0.0345 \pm 0.0046	0.0565 \pm 0.0048	0.0560 \pm 0.0047	0.0570 \pm 0.0044
Bookcorpus	0.2805 \pm 0.0014	0.2953 \pm 0.0010	0.2974 \pm 0.0010	0.3170 \pm 0.0015	0.2784 \pm 0.0015

Table 6.5: Percentage Improvement of each algorithm over MLE in RMSE metric, 95% confidence interval

DATASET	ROBBINS	ERM	NPMLE	TRANSFORMERS18	TRANSFORMERS24
NHL	-30.55 \pm 6.55	1.46 \pm 0.65	3.24 \pm 0.92	3.51 \pm 1.01	3.46 \pm 1.00
MLB (BATTING)	-32.80 \pm 5.67	2.50 \pm 0.36	4.30 \pm 0.41	4.45 \pm 0.37	4.58 \pm 0.39
MLB (PITCHING)	-21.71 \pm 2.45	2.51 \pm 0.31	4.70 \pm 0.41	4.89 \pm 0.42	4.95 \pm 0.38
BOOKCORPUSOPEN	-4.58 \pm 0.43	9.38 \pm 0.10	10.82 \pm 0.11	10.38 \pm 0.18	11.43 \pm 0.17

Table 6.6: Plackett Luce ELO Ranking Coefficient based on RMSE. The coefficient of MLE is set to 0 throughout.

DATASET	ROBBINS	ERM	NPMLE	TRANSF18	TRANSF24
NHL	-2.536	1.458	3.252	3.756	3.587
MLB (BATTING)	-2.981	2.991	5.145	6.221	8.575
MLB (PITCHING)	-3.217	3.225	5.916	6.696	7.689
BOOKCORPUSOPEN	0.024	1.547	2.341	2.012	2.698

Chapter 7

Model Internals

7.1 Activations Converge Smoothly and Rapidly

We aim to address two key questions in this experiment(7.1):

1. **When does the representation converge to something close to that of the final layer?** This seems to consistently occur at the output of the transformer layer of the first set of weights.
2. **When is an improved answer (over x) extractable?** This happens at least by the midpoint of the model, although different decoders may enable earlier extraction.

The Positive answer to 1, suggests that we might be able to train a single Sparse Auto Encoder (SAE) on the activations of the second layer [27], as they seem to live in similar representations. SAEs have proven very effective and identifying LLM modalities and might help us discern modalities in our transformers.

7.2 Activations of Our Layers Do not Encode Information Necessary Robbins or NPMLE

The downward trend followed by a plateau in A.3 suggests that these quantities are not key to the algorithm the model is implementing. Hence, it's unlikely the model is implementing Robbins or NPMLE. Especially if we contrast this with the results of Figure 4 in [4]. Where the prediction error of second moments relevant for computing linear regression goes down. (That is R_2 goes up)

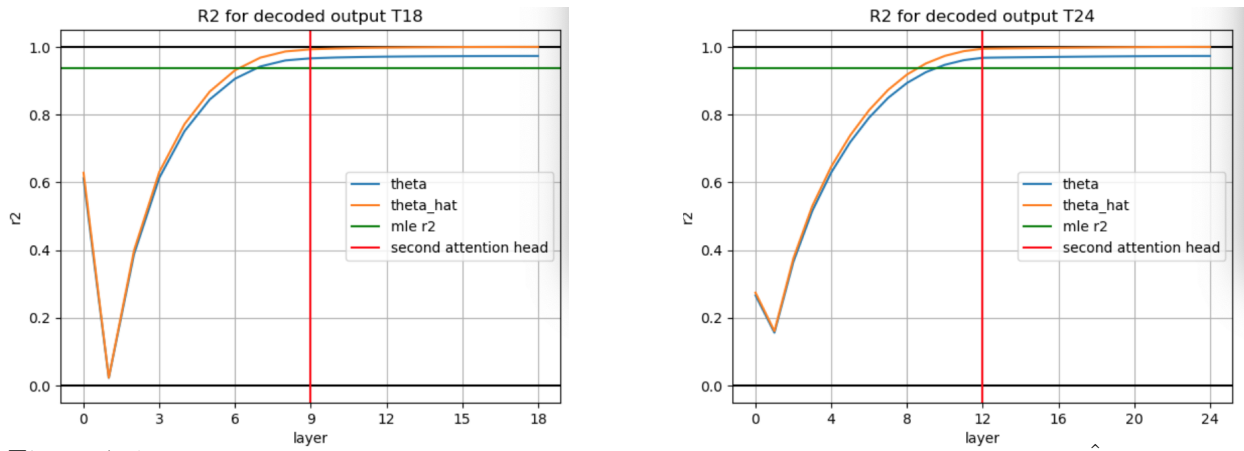


Figure 7.1: Plot of R2 for the decoder output of a transformer model against θ and $\hat{\theta}$ activations.

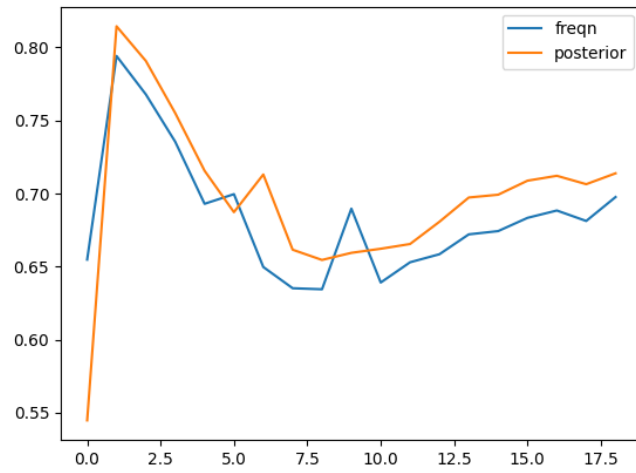


Figure 7.2: We perform a linear probe for the representations of $N(x)$, and $f_{\pi}(x)$, we notice that they have a downward trend

Chapter 8

Concluding Remarks

8.1 Summary

We started by providing an extensive literature review on in context linear regression. Categorizing the types of experiments done, and the conclusions drawn from them. 2. We then provided a brief literature review of the vast literature on empirical Bayes, highlighting known estimators that are regret optimal, as well as properties for each. 3

We then described a setup for training transformer based models. Highlighting a rigorous process in their evaluation. 4. We demonstrate that our small transformer models (around 25k parameters) consistently outperform state-of-the-art estimators. On both unseen synthetic and real data. For synthetic data, these include unseen priors with theoretical significance, as well as being able to take advantage of increases in sample size. For real-world data, they outperform other alternatives in sports data, and word frequency data 6

Additionally, we prove that a transformer based model, with a single transformer layer, can get arbitrarily close to being regret optimal on the interval $[0, \theta_{max}]$ while requiring only $O(\log(\theta_{max}))$ hidden dimension. 5. We finally attempt to understand the internals of our transformer models, demonstrating they converge quickly into the encoding of the output, and that they recover a good estimate before being halfway through the model (while only going one repeated layer due to weight-sharing). We also show that, unlike regression, it does not seem like their activations encode any of the celebrated algorithms. Particularly, we could not observe quantities related to Robbins or NPMLE in them. 7

Overall, we were able to provide matching experiments to many of the experiments highlighted in the literature review. Except for the ones in [5]

8.2 Future Directions

8.2.1 Improve and Better Understand the Models from This Work

Better understanding can be accomplished by investigating more parallels to in context linear regression. Particularly, we did not train decoders after each layer and analyze their properties similar to [5]. We also did not get to try Sparse Auto Encoders (SAEs) even though both 7.1 and A suggest they might yield interesting results. Additionally, deeper

probes on different targets can be used to attempt to understand the activations of the transformer.

In terms of improving the current transformers, one can tune additional parameters such as the Dirichlet Prior α , or its percentage per batch. Additionally, attempts to train the architecture for unbounded priors can be made.

Another way we might be able to add some of the desired properties of empirical Bayes estimators into the loss functions. Like Monotonicity for example.

8.2.2 Working on Empirical Bayes for Multidimensional Poisson

There are not many good-performing algorithms for Multi-Dimensional Poisson that scale well. Our Promising results in the single-dimensional case suggest that transformers could perform well in that setting. Especially, that the entire model architecture and selection setup described in 4 can be readily applied to higher dimensions. This is one of the advantages of our simple linear encoder.

8.2.3 Studying the effect of Changes to the Setup

We briefly studied the effect of randomizing the support of the prior 6.1.3. However, it's interesting to investigate how changing other aspects of the setup can change our results. For example, would changing the encoder change 7.1 and A, so that the properties of the two layers are more similar? We can also experiment with other forms of weight sharing such as alternating layers instead of repeating layers in sequence. ($L_1L_2 \dots L_1L_2$ instead of $L_1 \dots L_1L_2 \dots K_2$)

Appendix A

Pattern in the Updates To Transformer’s Residual Steams

Most of our top-performing transformers, for their respective depth, exhibit an interesting pattern in pairwise cosine similarity of their update streams. We illustrate the pattern on a smaller model first.

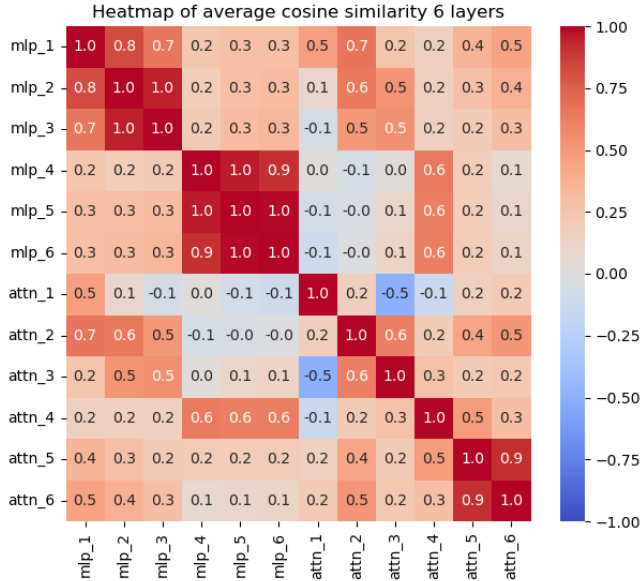


Figure A.1: Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added by the MLP in layer i

Observe the following, and note that it holds for the larger transformers below:

1. The updates of the second-half layers, which share the same weights, are highly correlated (shown as a red block in the bottom right of the top left quadrant, and bottom right quadrant).

- Most attention layer updates have a non-negative similarity with MLP updates. Except for the first few attention updates that tend to have negative similarities with further layers. (Especially those in the same shared-weights half)

My intuition is that these patterns might be related to to what we observed in 7.1. However, further analysis is needed to establish that.

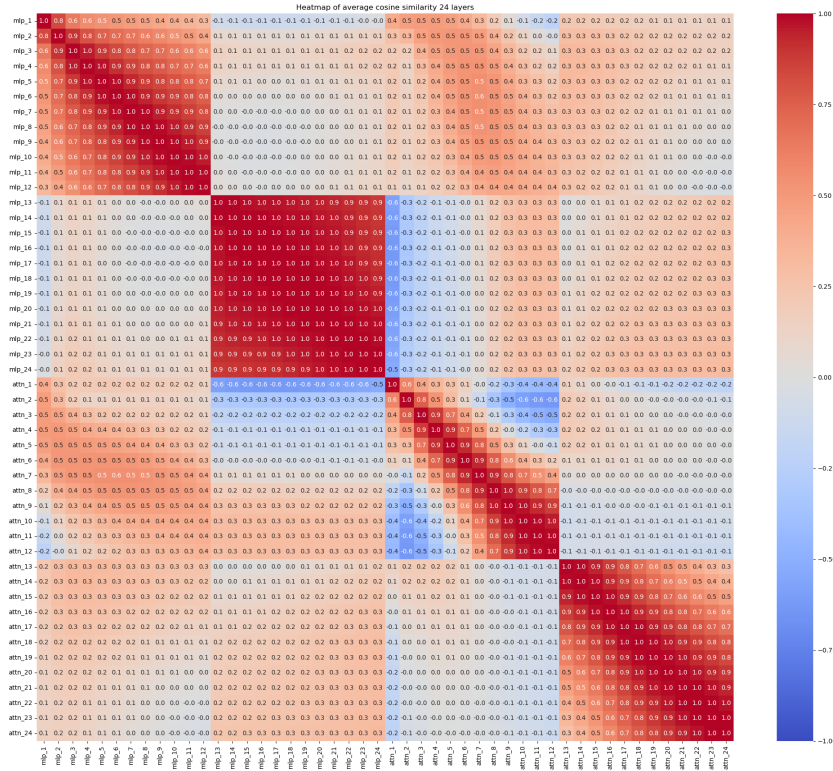


Figure A.2: Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added in the MLP in layer i

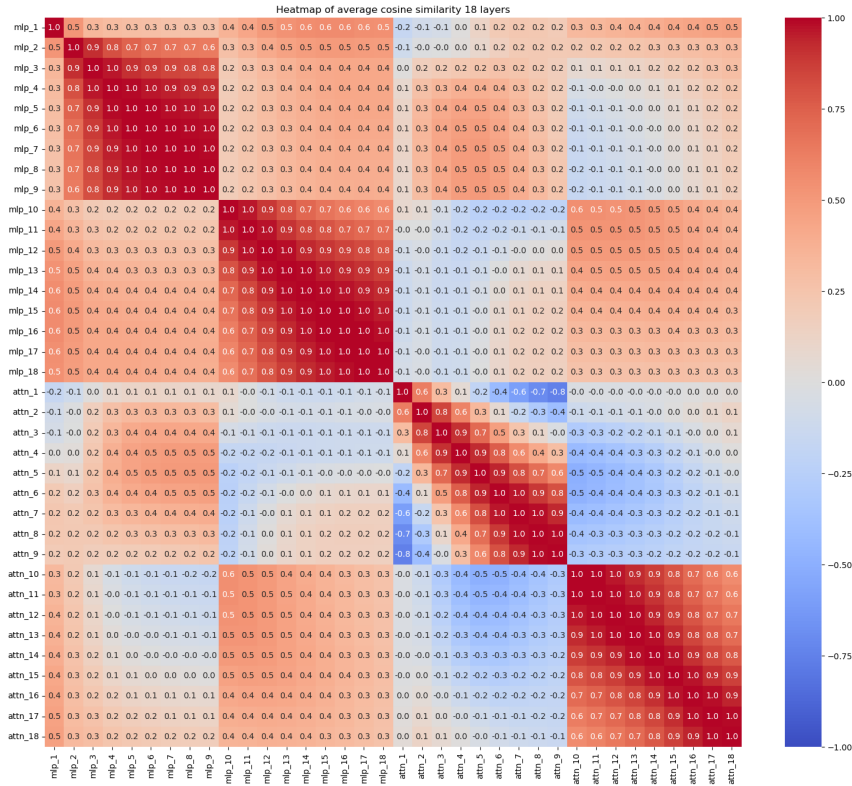


Figure A.3: Entries denote the average cosine similarity of additions to the residual stream. $attn_i$ denotes the vector added to the residual stream by the attention mechanism in layer i , and mlp_i denotes that added in the MLP in layer i

Appendix B

Code Snippets

B.1 Transformer

```
1 class EBTransformer(torch.nn.Module):
2 def __init__(self, args):
3     super().__init__()
4     self.args = args
5     factory_kwargs = {"device": args.device, "dtype": args.dtype}
6     # We augment each input with constant 1 dimension. Otherwise, for
7     # layernorm completely kills the
8     self.embed = torch.nn.Linear(args.dinput + 1, args.dmodel, **
9     factory_kwargs)
10    # Standard transformers have layernorms without weight sharing.
11    # So we want to incorporate that while making sure that previous
12    # transformers (layernorms w weight sharing) still works well.
13    if args.norm_share:
14        self.norm = torch.nn.LayerNorm(args.dmodel, **factory_kwargs)
15        self.norm2 = torch.nn.LayerNorm(args.dmodel, **factory_kwargs)
16    else:
17        num_norms = args.weight_share if args.weight_share > 0 else
18        args.layers
19        self.norm = torch.nn.ModuleList(
20            [
21                torch.nn.LayerNorm(args.dmodel, **factory_kwargs)
22                for _ in range(num_norms)
23            ]
24        )
25        self.norm2 = torch.nn.ModuleList(
26            [
27                torch.nn.LayerNorm(args.dmodel, **factory_kwargs)
28                for _ in range(num_norms)
29            ]
30        )
31
32    # Separating between weight sharing and no weight sharing.
33    # If weight sharing is N > 0, then we have N of the different
34    # weights.
```

```

31     if args.weight_share > 0:
32         self.linear1 = torch.nn.ModuleList(
33             [
34                 torch.nn.Linear(args.dmodel, 4 * args.dmodel, **
35                     factory_kwargs)
36                 for _ in range(args.weight_share)
37             ]
38         )
39         # self.dropout = torch.nn.Dropout(args.dropout);
40         self.linear2 = torch.nn.ModuleList(
41             [
42                 torch.nn.Linear(4 * args.dmodel, args.dmodel, **
43                     factory_kwargs)
44                 for _ in range(args.weight_share)
45             ]
46         )
47     else:
48         self.linear1 = torch.nn.ModuleList(
49             [
50                 torch.nn.Linear(args.dmodel, 4 * args.dmodel, **
51                     factory_kwargs)
52                 for _ in range(args.layers)
53             ]
54         )
55         # self.dropout = torch.nn.Dropout(args.dropout);
56         self.linear2 = torch.nn.ModuleList(
57             [
58                 torch.nn.Linear(4 * args.dmodel, args.dmodel, **
59                     factory_kwargs)
60                 for _ in range(args.layers)
61             ]
62         )
63     self.activation = (
64         torch.nn.modules.GELU()
65         if args.activation == "gelu"
66         else torch.nn.modules.ReLU()
67     )
68     # Here we distinguish between weight sharing or no.
69     if args.weight_share > 0:
70         self.self_attn = torch.nn.ModuleList(
71             [
72                 torch.nn.MultiheadAttention(
73                     args.dmodel,
74                     args.heads, # dropout=args.dropout,
75                     batch_first=True,
76                     **factory_kwargs,
77                 )
78                 for _ in range(args.weight_share)
79             ]
80         )
81     else:
82         self.self_attn = torch.nn.ModuleList(
83             [

```

```

81         torch.nn.MultiheadAttention(
82             args.dmodel,
83             args.heads, # dropout=args.dropout,
84             batch_first=True,
85             **factory_kwargs,
86         )
87         for _ in range(args.layers)
88     ]
89 )
90
91 # Enable causal processing
92 if False:
93     tmp = torch.full((args.seqlen, args.seqlen), 1, dtype=torch.
94                     uint8)
95     self.mask = torch.triu(tmp, diagonal=1).to(
96         args.device
97     ) # replace diagonal and below with zeros
98 else:
99     self.mask = None
100
101 if args.decoding_layer_norm:
102     self.norm3 = torch.nn.LayerNorm(args.dmodel, **factory_kwargs)
103
104 self.decoder = torch.nn.Linear(args.dmodel, args.dinput, **
105     factory_kwargs)
106 return None
107
108 def forward(self, inputs):
109     # Add dummy dimension to help avoid killing input's magnitude by
110     # the LayerNorm.
111     # inputs is BxTx d_in. inputs_aux is BxTx(d_in+1).
112     ones = torch.ones(inputs.shape[0], inputs.shape[1], 1).to(self.
113         args.device)
114     inputs_aux = torch.cat([inputs, ones], dim=2)
115     emb = self.embed(inputs_aux)
116     for i in range(self.args.layers):
117         idx = (
118             int(i * self.args.weight_share // self.args.layers)
119             if self.args.weight_share > 0
120             else i
121         )
122         if self.args.norm_share:
123             tmp = self.norm(emb)
124         else:
125             tmp = self.norm[idx](emb)
126         tmp, _ = self.self_attn[idx](
127             tmp, tmp, tmp, attn_mask=self.mask, need_weights=False
128         )
129
130         # tmp = self.dropout(tmp);
131         ## WARNING: do not use emb += .... since that op is inplace
132         emb = emb + tmp * self.args.step
133     # Implement FF module
134     if self.args.norm_share:

```

```

131         tmp = self.norm2(emb)
132     else:
133         tmp = self.norm2[idx](emb)
134         tmp_ff1 = self.linear1[idx](tmp)
135         tmp_ff2 = self.activation(tmp_ff1)
136         tmp_ff3 = self.linear2[idx](tmp_ff2)
137         emb = emb + tmp_ff3 * self.args.step
138
139     if self.args.decoding_layer_norm:
140         emb = self.norm3(emb)
141     out = self.decoder(emb)
142     # To ensure that we only prediction nonnegative values, we add a
143     # final relu step.
144     # Edit (AT) : ReLU can turn bad if your network start with all
145     # negative values (or get into that at one point),
146     # let's try GELU instead.
147     gelu = torch.nn.GELU()
148     out = gelu(out)
149     return out
150
151 def eval_loss(self, inputs, labels):
152     out = self.forward(inputs)
153     loss = ((out - labels) ** 2).sum() / inputs.numel()
154     return loss

```

B.2 Plackett-Luce

```

1 import torch
2 import torch.optim.lr_scheduler as lr_scheduler
3 from collections import defaultdict
4 from itertools import chain
5 #NOTE: RANKING HERE IS NOT REALLY REALLY RANKING, IT'S ARGSORT
6 def get_log_outcome_probability(weights, ranking):
7     exp_weights = torch.exp(weights)
8     ranked_exp_weights = exp_weights[ranking]
9     cumsum = torch.cumsum(ranked_exp_weights, dim=1)
10    rev_cumsum = torch.sum(ranked_exp_weights, dim = 1)[:,None] - cumsum +
11    ranked_exp_weights
12    log_probs = torch.log(rev_cumsum + 1e-6)
13    return torch.sum(weights) - torch.sum(log_probs, dim = 1)
14
15 def sgd_plackett_luce(rankings, lr = 1e-1, thresh = 1e-6, max_iter= 100, c
16    =.01, debug=True, debug_out = ""):
17    weights = torch.randn(rankings.shape[1], requires_grad=True)
18    print(weights)
19    optimizer = torch.optim.Adam([weights], lr=lr)
20    optimizer.zero_grad()
21    scheduler = lr_scheduler.LinearLR(optimizer, start_factor=.5,
22    end_factor=1e-3, total_iters=max_iter)
23    for i in range(1, int(max_iter)):

```

```

22     if weights.grad is not None:
23         if torch.norm(weights.grad) < thresh:
24             break
25         if i % 100 == 0 or i < 10:
26             print(torch.norm(weights.grad))
27     else:
28         if i % 100 == 0 or i < 10:
29             print(i, " bad iteration")
30     optimizer.zero_grad()
31     loss = -torch.mean(get_log_outcome_probability(weights, rankings))
32             + c * (torch.sum(torch.exp(weights)) - 1) ** 2
33     loss.backward()
34     optimizer.step()
35     if i % 100 == 0:
36         scheduler.step()
37     return weights
38
39 def get_adjacency_lists(rankings):
40     actual_rankings = rankings.argsort(axis = 1)
41     adjacency_list = defaultdict(lambda: [])
42     adjacency_list_rev = defaultdict(lambda: [])
43
44     for i in range(rankings.shape[1]):
45         for j in range(rankings.shape[1]):
46             if torch.any(actual_rankings[:, i] < actual_rankings[:, j]):
47                 adjacency_list[i].append(j)
48                 adjacency_list_rev[j].append(i)
49     return adjacency_list, adjacency_list_rev
50
51 def dfs(adjacency_list, l, i, visited):
52     if i not in visited:
53         visited.add(i)
54         for j in adjacency_list[i]:
55             dfs(adjacency_list, l, j, visited)
56     l.append(i)
57
58
59 def get_strongly_connected_component(adjacency_list, adjacency_list_rev):
60     #Source: https://cp-algorithms.com/graph/strongly-connected-components.html
61     #STEP 1: make post order
62     visited = set()
63     post_order = []
64     for i in range(len(adjacency_list)):
65         dfs(adjacency_list, post_order, i, visited)
66     #reverse post order
67     post_order.reverse()
68     #run dfs on reversed post order to get connected components
69     components = []
70     visited = set()
71     for i in post_order:
72         component = []
73         dfs(adjacency_list_rev, component, i, visited)

```

```

74         if len(component) > 0:
75             components.append(component)
76     return components
77
78 def convex_placketluce(rankings, lr = 1e-1, thresh = 1e-7, max_iter=
1000000, c =.01, gap_factor = 2.5, debug=True, debug_out = ""):
79     adjacency_list, adjacency_list_rev = get_adjacency_lists(rankings)
80
81     components = get_strongly_connected_component(adjacency_list,
82           adjacency_list_rev)
83     #NOTE: In our case the condensed graph must be a line. In a general
84     #setting we might want to return the condensed graph as well to
85     #get the full picture
86     actual_rankings = rankings.argsort(axis = 1)
87     weights = []
88     for i, component in enumerate(components):
89         print(f"handling component {i} with {len(component)} models")
90         if len(component) == 1:
91             weights.append(torch.tensor([0.0]))
92             continue
93         relevant_actual_ranking = actual_rankings[ : , torch.tensor(
94             component)]
95         relevant_ranking = relevant_actual_ranking.argsort(axis = 1)
96         component_weights = sgd_placket_luce(relevant_ranking, lr , thresh
97             , max_iter, c, debug=debug, debug_out = debug_out + f"_{i}")
98         weights.append(component_weights)
99         temp_concatinated_weights = torch.concat(weights)
100         gap = gap_factor * (torch.max(temp_concatinated_weights) - torch.min(
101             temp_concatinated_weights))
102         output_weights = [weights[0]]
103         for comp_weight in weights[1:]:
104             upper_bound = torch.min(output_weights[-1]) - gap
105             difference = upper_bound - torch.max(comp_weight)
106             output_weights.append(difference + comp_weight )
107         output_weights = torch.concat(output_weights)
108         output_weights[ torch.tensor(list(chain.from_iterable(components))) ]
109             = output_weights.clone()
110     return output_weights
111
112 if __name__ == "__main__":
113     losses = torch.tensor([[1, 2, 3], [2, 1, 3], [1, 2, 3]])
114     ranking = losses.argsort(axis = 1)
115     print(ranking)
116     num_iter = 10000
117     weights_convex = convex_placketluce(ranking, max_iter=num_iter)
118     weights_none_convex = sgd_placket_luce(ranking, max_iter=num_iter)
119
120     print("final weights")
121     print("weights_convex", weights_convex)
122     print("weights_none_convex", weights_none_convex)
123
124     #This case demonstaints that the gradient is much smaller when we
125     #break into connected components, which means we are more likely

```

```
120 # to detect convergence issues
121
122 #After switching to Adam, we can further decrease our threshold to 1e
    -9, and only the convex one will properly converge.
```


References

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2023. arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [2] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, *A survey for in-context learning*, Dec. 2022. DOI: [10.48550/arXiv.2301.00234](https://doi.org/10.48550/arXiv.2301.00234).
- [3] E. Akyürek, D. Schuurmans, J. Andreas, T. Ma, and D. Zhou, *What learning algorithm is in-context learning? investigations with linear models*, 2023. arXiv: [2211.15661](https://arxiv.org/abs/2211.15661) [cs.LG]. URL: <https://arxiv.org/abs/2211.15661>.
- [4] S. Garg, D. Tsipras, P. Liang, and G. Valiant, “What can transformers learn in-context? a case study of simple function classes,” in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS ’22, New Orleans, LA, USA: Curran Associates Inc., 2022, ISBN: 9781713871088.
- [5] D. Fu, T.-Q. Chen, R. Jia, and V. Sharan, *Transformers learn to achieve second-order convergence rates for in-context linear regression*, 2024. arXiv: [2310.17086](https://arxiv.org/abs/2310.17086) [cs.LG]. URL: <https://arxiv.org/abs/2310.17086>.
- [6] C. R. Walters, “Empirical bayes methods in labor economics,” National Bureau of Economic Research, Working Paper 33091, Oct. 2024. DOI: [10.3386/w33091](https://doi.org/10.3386/w33091). URL: <http://www.nber.org/papers/w33091>.
- [7] Y. Polyanskiy and Y. Wu, *Sharp regret bounds for empirical bayes and compound decision problems*, 2021. arXiv: [2109.03943](https://arxiv.org/abs/2109.03943) [math.ST]. URL: <https://arxiv.org/abs/2109.03943>.
- [8] S. Jana, Y. Polyanskiy, and Y. Wu, *Optimal empirical bayes estimation for the poisson model via minimum-distance methods*, 2024. arXiv: [2209.01328](https://arxiv.org/abs/2209.01328) [math.ST]. URL: <https://arxiv.org/abs/2209.01328>.
- [9] S. Jana, Y. Polyanskiy, A. Teh, and Y. Wu, *Empirical bayes via erm and rademacher complexities: The poisson model*, 2023. arXiv: [2307.02070](https://arxiv.org/abs/2307.02070) [math.ST]. URL: <https://arxiv.org/abs/2307.02070>.
- [10] B. Geshkovski, C. Letrouit, Y. Polyanskiy, and P. Rigollet, *The emergence of clusters in self-attention dynamics*, 2024. arXiv: [2305.05465](https://arxiv.org/abs/2305.05465) [cs.LG]. URL: <https://arxiv.org/abs/2305.05465>.
- [11] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020. arXiv: [2005.14165](https://arxiv.org/abs/2005.14165) [cs.CL]. URL: <https://arxiv.org/abs/2005.14165>.

- [12] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, *Large language models are zero-shot reasoners*, 2023. arXiv: [2205.11916](https://arxiv.org/abs/2205.11916) [cs.CL]. URL: <https://arxiv.org/abs/2205.11916>.
- [13] S. Borgeaud, A. Mensch, J. Hoffmann, *et al.*, *Improving language models by retrieving from trillions of tokens*, 2022. arXiv: [2112.04426](https://arxiv.org/abs/2112.04426) [cs.CL]. URL: <https://arxiv.org/abs/2112.04426>.
- [14] Y. Yin, Z. Wang, Y. Sharma, D. Niu, T. Darrell, and R. Herzig, *In-context learning enables robot action prediction in llms*, 2024. arXiv: [2410.12782](https://arxiv.org/abs/2410.12782) [cs.R0]. URL: <https://arxiv.org/abs/2410.12782>.
- [15] J. Y. Zhu, C. G. Cano, D. V. Bermudez, and M. Drozdal, *Incoro: In-context learning for robotics control with feedback loops*, 2024. arXiv: [2402.05188](https://arxiv.org/abs/2402.05188) [cs.R0]. URL: <https://arxiv.org/abs/2402.05188>.
- [16] Y. Zhou, X. Li, Q. Wang, and J. Shen, *Visual in-context learning for large vision-language models*, 2024. arXiv: [2402.11574](https://arxiv.org/abs/2402.11574) [cs.CV]. URL: <https://arxiv.org/abs/2402.11574>.
- [17] C. Yun, S. Bhojanapalli, A. S. Rawat, S. J. Reddi, and S. Kumar, *Are transformers universal approximators of sequence-to-sequence functions?* 2020. arXiv: [1912.10077](https://arxiv.org/abs/1912.10077) [cs.LG]. URL: <https://arxiv.org/abs/1912.10077>.
- [18] B. Efron, “Empirical bayes: Concepts and methods,” in *Handbook of Bayesian, Fiducial, and Frequentist Inference*, Chapman and Hall/CRC, 2024, pp. 8–34.
- [19] B. Efron, *Large-Scale Inference: Empirical Bayes Methods for Estimation, Testing, and Prediction* (Institute of Mathematical Statistics Monographs). Cambridge University Press, 2010.
- [20] W. K. Thompson, Y. Wang, A. J. Schork, A. Witoelar, V. Zuber, S. Xu, T. Werge, D. Holland, S. W. G. of the Psychiatric Genomics Consortium, O. A. Andreassen, *et al.*, “An empirical bayes mixture model for effect size distributions in genome-wide association studies,” *PLoS Genetics*, vol. 11, no. 12, e1005717, 2015.
- [21] H. E. Robbins, “An empirical bayes approach to statistics,” in *Breakthroughs in Statistics: Foundations and basic theory*, Springer, 1992, pp. 388–394.
- [22] A. Gadetsky, K. Struminsky, C. Robinson, N. Quadrianto, and D. Vetrov, *Low-variance black-box gradient estimates for the plackett-luce distribution*, 2019. arXiv: [1911.10036](https://arxiv.org/abs/1911.10036) [cs.LG]. URL: <https://arxiv.org/abs/1911.10036>.
- [23] N. Elhage, T. Hume, C. Olsson, *et al.*, *Toy models of superposition*, 2022. arXiv: [2209.10652](https://arxiv.org/abs/2209.10652) [cs.LG]. URL: <https://arxiv.org/abs/2209.10652>.
- [24] L. D. Brown, E. Greenshtein, and Y. Ritov, “The poisson compound decision problem revisited,” *Journal of the American Statistical Association*, vol. 108, no. 502, pp. 741–749, 2013.
- [25] S. Jana, Y. Polyanskiy, and Y. Wu, “Optimal empirical bayes estimation for the poisson model via minimum-distance methods,” *arXiv preprint arXiv:2209.01328*, 2022.

- [26] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books,” *arXiv preprint arXiv:1506.06724*, 2015.
- [27] H. Cunningham, A. Ewart, L. Riggs, R. Huben, and L. Sharkey, *Sparse autoencoders find highly interpretable features in language models*, 2023. arXiv: [2309.08600](https://arxiv.org/abs/2309.08600) [cs.LG]. URL: <https://arxiv.org/abs/2309.08600>.