Meta Reinforcement Learning with Transformers

Exploring asymmetry bias in the multi-armed bandits

A THESIS PRESENTED
BY
VLADIMIR PETROV
TO
THE DEPARTMENT OF STATISTICS

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF ARTS (HONORS)
IN THE SUBJECT OF
STATISTICS

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
APRIL 2024

Meta Reinforcement Learning with Transformers

ABSTRACT

Over the past few years, Transformer architecture has resulted in major breakthroughs in numerous machine-learning domains. Based on the transformer's architecture, pre-trained Large Language Models like GPT3 demonstrate surprising generalization capabilities. One of these phenomena is known as in-context learning (ICL): when given a prompt of a few input-output examples from a natural language processing task, the model can generate contextually relevant and coherent responses across a wide range of tasks. Therefore, LLMs have exhibited the ability of meta-learning: generalize well across the distribution of tasks by learning to identify the task itself from a short context and quickly adapt to this task. In this work, we investigate the capabilities of the transformer's architecture for meta-learning in the field where fast adaptation is the essential part of the problem itself: Reinforcement Learning. In particular, we aim to measure the transformer's ability to approximate the best strategy in the classical Bayesian multi-armed bandit task, which is known as the Gittins index. Our experiments show that the original Transformer architecture can learn an efficient strategy that establishes exploration vs. exploitation trade-off and outperforms other competitive methods. However, we also find that transformers are prone to asymmetric biases that prevent them from converging to the exact global optimum. We also empirically show that in most experiments the original transformer is capable of approximating the global optimum very well when encouraged to be symmetric. Our results, therefore, suggest that the policy gradient-based optimization might be partially responsible for pushing the model to become greedy in Reinforcement Learning settings and, therefore, should be reconsidered in meta-RL problems, especially when the goal is to approximate complex dynamic algorithms such as the Gittins index.

Contents

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Associate Professor Lucas Janson, for his endless commitment to the project and investment in my personal growth as a researcher. His strong enthusiasm for the topic of the mathematics of deep learning, along with thoughtful guidance, allowed me to discover a novel academic and career passion. Second, I would also like to thank my co-advisor, Nikhil Vyas, whose practical expertise and knowledge of transformers illuminated novel directions, boosted the project's development, and allowed me to overcome computational difficulties. Finally, I would like to thank my family and closest friends, who have constantly inspired me to push academic and personal endeavors beyond my limits.

Chapter 1

Introduction

Generative pre-trained transformers (GPT, [24]), or simply transformers, have established a breakthrough in numerous machine learning domains over the past few years. Large language models (LLMs) like GPT-3 [2] have set new standards in AI-generated text that is at times indistinguishable from human-written content, while in computer vision, Vision Transformers [5] have surpassed previous state-of-the-art models. Given these advancements, some researchers hypothesize that transformers represent a significant stride toward the development of Artificial General Intelligence (AGI). When trained on sufficiently large datasets, these models have shown the first signs of the ability to generalize through a phenomenon called in-context learning (ICL). When given a prompt of a few input-output examples from an arbitrary task, the model can generate predictions for new inputs without any parameter updates. Therefore, Large language models have shown an ability to adapt quickly to new tasks and approximate desired solutions. This ability draws the connection with a field of meta-learning, where the objective for the model is to quickly identify *both* the task and the optimal solution for this specific task rather than excel at one singular task.

Concurrently, transformers have garnered increasing interest in Reinforcement Learning (RL). Many RL tasks are sequential by nature: the agent interacts with the environment through trial and error, while the optimal strategy is constantly updated depending on the current state and previous interaction history. Transformers excel at sequential tasks by efficiently processing sequential data and capturing long-range dependencies through the attention mechanism. This makes the transformer's architecture a strong candidate for partially observable RL problems, where episodes can span for hundreds or even an infinite number of steps, yet critical observations could span the whole history.

Motivated by LLMs' ability for quick adaptation (as demonstrated by in-context learning) and their increasing application in reinforcement learning settings, our work lies at the intersection of these fields: meta-reinforcement learning. In meta-RL, the model learns a meta-policy that can rapidly adapt its behavior across a range of RL tasks: the objective is to

identify and solve a new task efficiently rather than excelling at a single task.

This ability of models to adapt quickly towards different RL tasks has received particular attention recently with a premise of aligning LLMs towards human individual preferences[1], because the interaction with a human, inherently, has reinforcement nature. Hence, a successful performance in various Meta-RL problems can be seen as a crucial proof-of-concept for the transformer's ability to generalize well beyond the RL tasks seen in the training set.

In this work, we investigate the capabilities of transformers in Meta RL, as demonstrated by a classical problem in the field: multi-armed bandits. Our goal is to understand the shortcomings of the original transformer's architecture as measured by its ability to approximate the optimal strategy known as the Gittins index. At Every step throughout the episode, this solution recursively evaluates each arm based on its current and potential future rewards, factoring in the discount rate and the exploration vs. exploitation of the future. Our experiments show that the original Transformer architecture can learn efficient strategy in bandit problems that establish exploration vs. exploitation trade-off and even outperforms non-trivial, asymptotically optimal solutions (e.g., Thompson sampling) by a significant margin. However, we also find that transformers are prone to asymmetric biases that prevent them from converging to the exact optimum, i.e., the Gittins index. Moreover, this asymmetry is responsible for the gap from the optimum: the model approximates the optimum very well when enforced to respect the underlying problem's symmetry. Our results, therefore, suggest that either the original transformer's architecture or the gradient-based optimization methods should be reconsidered in meta-RL settings, especially when approximating complex dynamic algorithms like the Gittins index.

Our paper outlines as follows: chapter 2 provides the necessary background and discusses related work, chapter 3 formulates the methods used to train the models, chapter 4 presents experiments and results, and chapter 5 concludes.

[1] this framework is commonly referred to as Reinforcement Learning from Human Feedback (RLHF) [14]

Chapter 2

Background and Related Work

This chapter will discuss the related work in Deep and Meta Reinforcement Learning. We first introduce the necessary definitions and background (2.1), then motivate the development of the Deep RL field and its progression (2.2) - (2.4), and finally turn to an overview of recent applications of Transformers in the Meta RL domain and related fields (2.5) - (2.7).

2.1 Preliminaries

This section introduces the necessary background to formalize our goals. Here, we will provide the math notation necessary for the further discussion of Deep RL. Still, most of the concepts will be discussed at a high level, and a more detailed discussion can be found in the methodology chapter 3. We start with the basic definitions of Reinforcement Learning, then generalize it to the Meta RL framework, and describe a particular Meta RL problem investigated in our work: multi-armed bandits.

2.1.1 Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions by taking actions in an environment to achieve some goal or maximize some notion of cumulative reward. The standard RL framework is modeled as a Markov decision process (MDP) with a set of possible states S, actions A, reward distribution, and transition probability functions. The agent interacts with the environment over a discrete number of time steps (finite or infinite). At each time step t, the agent is in the state $s_t \in S$ and decides to take an action $a_t \in A$ (discrete or continuous control). In return, they receive the reward $r_t \sim P(\cdot|a_t, s_t)$ and transition to the next state s_{t+1} based on the environment's transition probability function $\Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$. Typically, we deal either with the infinite-horizon process, where each reward r_t would be discounted by γ^t with $\gamma \in [0, 1)$, or with the finite process that finishes

at some fixed time step T. At the end of the episode, the agent receives the cumulative discounted return along the overall episode trajectory $\tau = (s_0, a_0, r_0, ...)$ defined as $R = \sum_{t=0}^{\infty} r_t \cdot \gamma^t$, and then the process repeats again.

The agent aims to learn the policy $\pi : S \times A \to [0, 1]$, that would guide them at each state s to take an action a with probability $\pi(s, a) = \Pr(a_t = a \mid s_t = s)$. The *optimal* policy π for the current environment is the one that maximizes the expected reward:

$$\max_{\pi} \mathbb{E}_{\tau} R^{\pi}(\tau)$$

where the trajectories τ happen every time based on the policy π, and the expectation averages out the stochasticity coming from the state transition dynamics, reward distribution, and the policy's randomness.

There are two foundational concepts related to the MDP-based Reinforcement Learning framework that will be helpful further in our discussion:

– **Action-value** functions are given by the expected reward agent receives by taking action a_t from state s_t following policy π onwards:

$$Q_t^{\pi}(s, a) = E(r_t + \gamma \cdot r_{t+1} + ...|s_t = s, a_t = a),$$

and the optimal action-value is the the maximum achievable by any possible policy: $Q_t^*(s, a) = \max_{\pi} Q_t^{\pi}(s, a)$.

– **Value** function is just the expected reward from s_t following policy π:

$$V_t^{\pi}(s) = E(r_t + \gamma \cdot r_{t+1} + ...|s_t = s)$$

As described above, the optimal policy π for a single environment aims to maximize the expected cumulative reward R^{π} per episode of interactions with this environment. Hence, the environment is fixed, meaning all transition probabilities and reward distributions remain unchanged across different episodes. We will now break this assumption by suggesting that the environment can also be random. This changes the goal of learning the policy that would be optimal across several environments simultaneously rather than a single one, which directs us precisely to the Meta RL field.

2.1.2 Meta RL

As discussed in the introduction, the meta-learning algorithm aims to generalize well across the distribution of tasks (environment) rather than excel at a single task. In this way, we deal with some distribution of tasks $M \sim \mathbf{D}$, where M is an instance of reinforcement learning task (for example, it might be MDP described earlier). Each task M is episodic for the total length

of T (for now, fixed), and at each time step t model makes the decision based on all available history: $H_t = (s_0, a_0, r_0, ..., s_{t-1}, a_{t-1}, r_{t-1}, s_t)$ following $a_t \sim \pi(H_t)$. The meta-learner's objective is, thus, to maximize the expected reward across **all** tasks:

$$\max_{\pi} \mathbb{E}_{M \sim P(M)} \left(\mathbb{E}_{\tau \sim (M,\pi)} R(\tau) \right), \tag{2.1}$$

where episode trajectories depend on the environment characteristics M and policy π. In this way, a successful policy should not perfectly adapt towards one task (i.e., that would be "overfitting" to that specific task). Rather, it should learn to identify which task it deals with (by learning from the evolving history H_t) and quickly adapt to the current task. There are numerous Meta RL tasks, and we will describe one investigated in this work: the Bayesian multi-armed bandit.

2.1.3 Bandits

A classical Reinforcement learning task is the multi-armed bandit (MAB). In this problem, there are K arms with a set of associated arm means $\vec{\mu} = \{\mu_1, ..., \mu_K\} \in [0, 1]^K$ and the reward distributions are defined as $R_i \sim \text{Bernoulli}(\mu_i)$. The agent aims to effectively learn the most promising arm, i.e., the arm with the highest mean $\mu^* = \max_i \mu_i$. The *discounted* regret ρ is the measure of loss per episode of interactions with the bandit defined as the sum of $\rho = \sum_{t=0}^{T} (\mu^* - \mu_t) \cdot \gamma^t$. Thus, the goal of an effective algorithm is to minimize this discounted regret. Note that the bandit can be seen as an instance of a very simple MDP: there are no states s_t, and thus no action-to-state transition probabilities $\Pr(s_{t+1} = s' \mid a_t = s, a_t = a)$, and the reward distributions depend only on actions a_t of which arm to pull: $r_t \sim \text{Bernoulli}(\mu_{a_t})$. Most of the methods we will describe further in the work for MDPs in general (for instance, all the techniques of Deep RL) and therefore work for bandits too, but starting from a certain point, we switch directly to bandits. We hope that this exchange of notation (from MDPs to bandits) will be clear from the context, and we apologize to the reader for the ambiguous abuse of notation if it occasionally occurs in the paper.

Another critical thing to note about the bandits is the exploration-exploitation dilemma. There is always a trade off between exploring the arms that have not been pulled yet (or pulled way less than others), and exploting the ones that have already shown sucessful outomes $r = 1$. The optimal algorithm treats its choices of arms, respecting this trade-off. We can informally define that the algorithm is "greedy" if it chooses to exploit too much (for example, fully commit to whatever arm has shown successful outcome $r = 1$ before other arms) or "explorative" if it chooses to explore more (for instance, always pulling arms with equal probabilities no matter what the past outcomes are). We use these terms later when investigating the behavior of our model.

Going back to the discussion of meta RL objective (2.1), a single instance of task M in that objective corresponds to a single fixed vector $\vec{\mu}$ of arms means. When we assume randomness of this vector, i.e., $\vec{\mu} \sim \mathbf{D}$ follows some prior, we get the setting of the Bayesian multi-armed bandit. In this setting, the optimal strategy is known *exactly* as the Gittins index strategy [10], and we treat it as an oracle solution in our experiments. We discuss some essential properties of this algorithm in detail in chapter 3. Still, the full details of the algorithm are beyond the scope of this paper. For the remainder of this section, we suggest the reader treat the Gittins index strategy simply as a black-box oracle. Because this oracle represents the global optimum of the Bayesian MAB Meta RL objective, we can use it to measure whether the model has converged to the global optimum.

Another famous algorithm for bandits is known as the Thompson sampling (initially described in [29] and the convergence to the maximizing arm proved by [35]). The main idea behind this algorithm is to maintain a probability distribution over the possible arm means and select actions based on these distributions. We refer to the original paper for more details, but for the purpose of this work, we use Thompson sampling as a baseline measure for our problem. For instance, if the model has converged to the solution with the discounted regret higher than that of the Thompson sampling, we conclude that it significantly underperformed.

This concludes the necessary preliminaries for our work. Even though our work focuses on Meta Reinforcement Learning with deep networks like transformers, this discussion does not appear until section 2.7. Before that, it is crucial to understand why deep reinforcement learning appeared in the first place and what essential methods are needed when training networks for RL domains. We begin with the former by providing some motivation for applying modern deep learning models in the context of RL tasks. First, we need to understand the limitations of more classical (not neural network-based) algorithms in Reinforcement Learning, namely Tabular RL.

2.2 Tabular RL

In principle, any MDP-based Reinforcement Learning problem defined similarly to 2.1.1 can be solved *exactly* by value iteration algorithm or Q-value dynamic programming algorithm. For instance, the latter method calculates all values $Q_t(s,a) = Q(s,a,t)$ by solving the system of equations

$$Q(s,a,t) = r(s,a) + \mathbb{E}_{s_{t+1}\sim P(s,a)}\left(\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, t+1)\right) \tag{2.2}$$

and outputs the final policy decision as $\pi_t(s) = \arg\max_a Q(s,a,t)$. The solution to (2.2) can be found by the dynamic programming, where we would start with the Q values for the last horizon T and step by step progress up to horizon 0.

This class of methods is called **Tabular RL**: the state, action, and reward spaces are represented in a tabular form. In other words, the agent maintains and updates a table (or matrix) of values to decide which action to take in each state. Notably, these methods rely on the fact that we can calculate the right-hand side in the equation (2.2), in particular assuming that all $r(s, a)$ rewards are given to us. In this way, no "learning" is happening here; we simply use math-derived solutions for a system of equations. However, in real-world environments, we only get access to those rewards by gathering the data via simulations, i.e., playing with our MDP. Often, the chances of visiting a particular state and taking a particular action are very small, and we would never observe that reward. Several tabular algorithms address this problem by learning from the "offline" data of trajectories gathered in advance or by the "online" data generated by the algorithm itself. We will now describe one of these algorithms that appears useful in Deep RL.

2.2.1 Q-learning

In the online *Q-learning*, first suggested by [33], all Q values are initialized randomly with some fixed number. A single episode plays with the MDP in an *arbitrary* way and updates the Q function as follows:

$$Q_t(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \eta \cdot (r_t + \max_a Q_{t+1}(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{2.3}$$

where η is the learning rate. This "gathering the data" episode with MDP is repeated until convergence.

The expression $r_t + \max_a Q_{t+1}(s_{t+1}, a) - Q_t(s_t, a_t)$ is commonly referred to as temporal difference $TD(1)$ because it looks one step ahead and checks whether we can improve upon the current Q value. One drawback of this rule is that reward value r_t directly affects Q_t-value only for a single (s_t, a_t) pair, and indirectly other Q-values through the updated value $Q(s_t, a_t)$. The training might be very slow since many updates will be needed to propagate the reward r_t to the previous states. To mitigate this problem, the idea of a "further" look-ahead was introduced by [23], where the TD(n) difference is given by

$$r_t + \gamma r_{t+1} + \ldots + \gamma^n r_{t+n} + \max_a Q_{t+n+1}(s_{t+n+1}, a_{t+n+1}, \theta) - Q_t(s_t, a_t, \theta') \tag{2.4}$$

Even though we do not use Q-learning to train our model, the idea of temporal differences $TD(1)$ and further look ahead $TD(n)$ is very helpful for speeding up learning, for instance, when modeling the value function. As we will see further, the same approach appears in deriving the generalized advantages (section 2.4.2), leading to much faster convergence.

Going back to Q-learning updates (2.3), note that for the true solution $Q^*(s_t, a_t)$ all

temporal differences should be 0 because we cannot improve upon the best possible reward. Therefore, we can hope for convergence to the fixed point solution by constantly improving on the current values with the updates as in (2.3). In fact, [23] shows that Q-learning indeed converges to the optimum action values with probability 1 so long as all actions are repeatedly sampled in all states and the action values are represented discretely. Unfortunately, such an assumption might not be tractable when the space of states/actions is large or infinite, as in control tasks. For instance, in chess, the number of positions exceeds 10^{100}, while in the Bayesian multi-armed bandits, the number of possible input histories (i.e., history of past actions, rewards) for horizon $T = 100$ is at least K^{100}, none of these could be stored as $Q(s, a, t)$ table. The problem only worsens for longer and more complex games. Even when the environment is more compact, the question of the speed of convergence remains: $Q - learning$ requires exploration of all of the possible state-action pairs, but an intelligent algorithm should recognize that some moves are unpromising and never spend time nor memory on exploring those directions.

Notably, the above problem is not an intrinsic limitation of Q-learning specifically. Instead, it is the general drawback of all Tabular RL methods: they become impractical or infeasible for environments with large or continuous state and action spaces due to the vast number of entries in the Q-table. This motivates the development of function approximation techniques, bringing us to the discussion of Deep Reinforcement Learning.

2.3 Deep RL

To address the problem of extremely large or infinite action-state spaces, Q-learning and other RL algorithms for complex tasks are combined with function approximation with neural networks. In this paper, we focus on *model-free* approaches, which refer to methods that do not attempt to learn an explicit model of the environment, e.g., transition probabilities between the states of MDP. Broadly speaking, model-free approaches in Deep RL are split into value-based and policy-based methods. Though we follow the latter approach in our work, we start with discussing the former as it directly relates to the Q-learning that we just discussed: namely, deep learning models are used to approximate Q-values.

2.3.1 Value-based RL

Value-based approaches approximate Q functions with the artificial neural network. The network Q_θ, parametrized by θ, takes states and actions as input. The goal is to learn parameters θ that could approximate the true $Q^*(s, a)$ very well through "trial-and-error", i.e. aggregating the experience by playing with the environment. Notably, this does not require

visiting all action-states pairs. Such networks are commonly called deep Q-networks (DQNs) and were initially introduced in [20].
The learning happens by iteratively updating $\theta_{i-1} \to \theta_i$ with θ_i minimizing the loss function

$$L(\theta_i) = \mathbb{E}_{(s,a)\sim\rho(\cdot)}\left(r + \max_{a'} Q(s', a', \theta_{i-1}) - Q(s, a, \theta_i)\right)^2 \tag{2.5}$$

where s' is the next state after s, and $\rho(\cdot)$ is the transition distribution of the MDP. Comparing this to (2.3), the algorithm attempts to find θ_i that minimizes absolute values of TD(1) differences. The learning happens by the *gradient descent* on this objective. Rather than computing the full expectations in the above expression, it is often optimized by stochastic gradient descent (SGD): the expectations are replaced by several samples from the behavior distribution. In principle, SGD does not provide convergence guarantees. In section 2.4 we discuss further what adjustments can and *should* be made for the ordinary SGD to stabilize the training procedure.

The original DQN paper [20] combined this approach of deep learning modeling with a technique known as *experience replay*: they store the agents experiences at each time step in a data-set D, pooled over many episodes into a replay memory. During every Q-learning update as in (2.5), experiences are drawn randomly from D. As was found in previous works, learning directly from consecutive samples is inefficient due to the strong correlations between the samples; thus, randomly drawing the samples from D breaks these correlations and reduces the variance of the updates. They applied this idea to training deep CNNs, obtaining superhuman performance on the ATARI games benchmark.

Since then, several methods have been proposed to speed up the training of deep networks in RL settings from the engineering / computational perspective. For instance, the General Reinforcement Learning Architecture (Gorila) [21] presents a way to redistribute learning on different machines. Each process contains an actor that acts in its own copy of the environment, a separate replay memory, and a learner that samples data from the replay memory and computes gradients of the DQN loss. Those gradients are asynchronously sent back to the central model; the central server updates the corresponding local models at fixed intervals. Parallel computation on several machines allows for substantial speed-ups of training. In our procedure, we use a similar approach, but instead, parallelization happens across multi-GPUs on a single machine (described further).

2.3.2 Policy-based RL

In contrast to the value-based methods, policy-based methods for RL directly parameterize the policy actions as $\pi_\theta(a|s)$ and update the parameters θ by gradient ascent on $\mathbb{E}_{\tau\sim(M,\pi_\theta)}(R(\tau))$ (where M refers to the characteristics of the environment such as transition probabilities).

Policy gradient methods naturally extend to problems with continuous action spaces, whereas Q-learning typically requires discretization or function approximation techniques to handle continuous actions. In addition, policy gradient methods can learn stochastic policies, which are beneficial in uncertain environments or when exploring diverse actions.

Even when the optimal policy is **deterministic**, as with our problem of multi-armed bandits (Gittins index is deterministic), "forcing" the model to choose the maximizing action-value at each time step produces an "inductive" bias towards solving that particular RL problem. In other words, a good model should be able to learn this determinism itself, without any human guidance, just by observing the final $R(\tau)$. Therefore, a policy network $\pi_\theta(a|s)$ is, perhaps, a more appropriate choice for answering the fundamental question of interest: whether a particular architecture (e.g., transformer) can find optimal RL solution – with as little supervision from human as possible. This serves as a rationale for us to choose the policy-based method in our work (rather than value-based), though it is true one is no better than the other.

Policy-based models are typically trained in the "online" manner, i.e., the trajectories τ_i that the agent uses to learn from and back-propagate the loss depend themselves on the current policy π_θ and are being gathered anew at each time step. Therefore, the ordinary gradient ascent is not an appropriate strategy in this setting (we elaborate on this in 3.3).

The REINFORCE [34] family of algorithms deals with it by deriving an *unbiased* estimate for the gradient via the likelihood ration method:

$$\nabla_\theta \hat{R}_\pi(\tau) = \left(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \cdot R(\tau) \tag{2.6}$$

$$\nabla_\theta R_\pi(\tau) = \mathbb{E}_{\tau \sim \pi_\theta} \left(\hat{R}_\pi(\tau) \right)$$

It thus follows that during training, one can update parameters, making an incremental step in the direction of $\hat{R}_\pi(\tau)$ and, *on average*, that would be a correct direction to increase the total expected reward. The estimate $\mathbb{E}_{\tau \sim \pi_\theta}$ is typically gathered by Monte Carlo simulation of several trajectories τ_i.

Another difference between DQNs as in (2.5) and REINFORCE as in (2.6) is that the former belongs to *off-policy* RL algorithms, while the other to *on-policy* algorithms. In off-policy methods, the agent learns the value function or policy based on a different policy from which it collects data. For instance, the update used in (2.5) is based on the maximum Q-value for the next state, which is not the policy the agent is following. In contrast, in (2.6), the policy π_θ is both used to collect the experiences (trajectories τ) and updated based on these experiences. There is no free lunch between the two categories, i.e., one is not universally better, but on-policy methods are generally simpler to implement and more stable to train. In this work,

we start with the original on-policy REINFORCE approach but then modify it to the Off-Policy REINFORCE using baselines (see further) to reduce the variance of the gradient estimates.

In principle, at this moment, we have the necessary toolkit to train any model, e.g., GPT transformer, to solve any downstream task by performing gradient ascent using the expression in (2.6). However, as mentioned previously, SGD does not guarantee convergence to the optimum, and one of the reasons for that is the high variance of gradient direction estimates. The following section discusses different techniques for modifying this straightforward combination of gradient estimate and SGD to accelerate convergence and stabilize the training. In many RL tasks, including multi-armed bandits, these adjustments become crucial for successful convergence, i.e., without them, the networks often cannot find optimal or take a very long time to converge.

2.4 Optimizations in Deep RL

In this section, we will discuss various adjustments one can make for the original REINFORCE (2.6) approach to optimize the gradient descent. We organize these techniques as follows:

– Variance reduction techniques that keep the gradient estimate unbiased (2.4.1);

– Modifications that introduce bias-variance trade-off (2.4.2);

– Optimization to the gradient descent itself (2.4.3)

2.4.1 Unbiased variance reduction

The expression in (2.6) gives an unbiased estimate $\hat{R}_\pi(\tau)$ for the true gradient direction $\nabla_\theta R_\pi(\tau)$, but it typically has significant variance: it incorporates the sum of T stochastic rewards, randomness in the environment and the policy itself. A straightforward way to reduce this variance is to draw a large number B (batch size) of trajectories $\tau_1, ..., \tau_B \sim \pi_\theta$ and average out their estimates: $\frac{1}{B}\sum_{i=1}^{B}\hat{R}_\pi(\tau_i)$. This comes at the cost of time and computation. But even a large batch size of B might still be insufficient: in our problem, batch sizes, even up to $100k$, are not stable enough.

To further reduce the variance, a more refined mathematical expression can be derived that still gives an unbiased estimate for the gradient. In the same paper REINFORCE [34] authors suggest for each t to subtract any function of the current state s_t, called the "baseline", from $R(\tau)$:

$$\mathbb{E}_{\tau\sim\pi_\theta}\sum_{t=0}^{T}(\nabla_\theta \log\pi_\theta(a_t|s_t)\cdot(R(\tau)-b_t(s_t)).)$$

This baseline can also depend on anything up to the moment t so that it can be a function of "history" $H_t = \tau_{[0:t]}$. We later show in the methodology section 3 why that is the case.

Immediately it follows that we can subtract all the observed rewards $r_1, ..., r_{t-1}$ from $R(\tau)$, leaving us with $R(\tau[t:]) := \gamma^t r_t + \gamma^{t+1} r_{t+1} + ...\gamma^T r_T$, and then it also follows that the best baseline to use is the value function $b_t(s_t) \approx V_t^\pi(s_t)$ (we explain later why). In this way, $R_t(\tau) - b_t(s_t)$ can be seen as an estimate for the *Advantage* $A_t(a_t, s_t) = Q_t(a_t, s_t) - V_t(s_t)$ of taking action a_t at state s_t, since the $R_t(\tau)$ and $b_t(s_t)$ are the estimates for the corresponding $Q(a_t, s_t)$ and $V(s_t)$ values. This method is typically called the "actor-critic framework" [4], since the policy π acts as an actor (tries to improve upon the baseline), and the baseline b_t is the critic.

In reality, V values are unknown, so a common approach is to **learn** these baselines with the network as well: $V(s_t, \theta_v) \approx V_t(s_t)$, where θ_v are learnable parameters of the value network. To emphasize, when using $V(s_t, \theta_v)$ instead of the true $V_t(s_t)$, the gradient expression still remains unbiased because the baseline can be any function of the current state s_t, or even all the history H_t up to t, including a function that depends on parameters θ.

Combining these ideas together, we get one of the most commonly used approaches for training deep networks: "Advantage Actor-Critic." A new gradient expression is therefore given by

$$\widehat{\nabla_\theta} R_\pi(\tau) = \sum_{t=0}^{T} \left(\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (R_t(\tau) - V(s_t, \theta_v)) \right) \tag{2.7}$$

The original gradient descent is modified with an additional update by calculating the mean squared value loss of the residuals $R(\tau[t:]) - V(s_t, \theta_v)$ across the whole episode. We refer the reader to the final math details in the methodology section 3.3.

Valuable patterns learned by the network policy π_θ and by the value function $V(\theta_v)$, perhaps, should have a lot in common. Therefore, parameters θ and θ_v are typically shared, and in practice, they are almost the same: one softmax output is used for the policy $\pi(a_t|s_t; \theta)$ and one linear output for the value function $V(s_t; \theta_v)$, with all non-output layers shared.

To speed up the training process, the "Asynchronous Advantage Actor-Critic" (A3C) paper [19] develops a method similar to that of General Reinforcement Learning Architecture (Gorila, [21]) for DQNs, but instead of using separate machines and a parameter server, they use multiple CPU threads on a single machine (which reduces communication cost). They argue that multiple actors-learners running in parallel will likely explore different parts of the environment, alleviating the problem of highly correlated updates when done by a single agent. Additionally, authors suggest adding the entropy regularization term $\beta \cdot \nabla_\theta H(\pi(s_t, \theta))$ (where β controls the strength of regularization) to discourage the network from premature convergence to suboptimal deterministic policies. Overall, their method reduces the training time and allows using on-policy reinforcement learning methods such as Sarsa and actor-critic (since they no longer rely on experience replay). This method is commonly referred to as A3C ("Asynchronous Advantage Actor-Critic") and serves as the backbone of our training

procedure.

In addition, Mao et al. [16] propose a meta-learning approach to learn the baseline much more efficiently in applications where we can repeat the same input sequence multiple times during training. The idea is to use all (potentially infinitely many) input sequences to learn a "meta value network" model. Then, for each specific input sequence, they first **customize** the meta value network using a few example rollouts with that input sequence. This **customized** value network is used to calculate the baseline values for the policy network gradient update on parameters of this specific input sequence. In this way, more granular, "fine-tuned" value networks are used for each specific input, resulting in more precise baselines, further reducing the gradient estimate's variance. This design mirrors the generic Model-Agnostic Meta-Learning (MAML) algorithm [8], which we will discuss further in the Meta-Learning section 2.8.1.

2.4.2 Generalized Advantages

In equation (2.7), we still had an *unbiased* estimate for the gradient. The second optimization technique, known as Generalized Advantage Estimates [28], suggests breaking this assumption: introducing slight bias while significantly reducing the variance of this estimate. In short, this method modifies the original advantage formula by a weighted trade-off between one-step, biased estimate $r_t + \gamma V_\theta(H_{t+1}) - V_\theta(H_t)$ and until-the-finish, unbiased estimate $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... - V_\theta(H_t)$, because the former has lower variance than the latter. This method significantly reduces the noisiness from the rewards' stochasticity and helps stabilize the training in our problem. Further math details are discussed in the methodology section 3.4.2.

2.4.3 Other optimization techniques

The previous two optimization techniques addressed different adjustments that can be made to the original gradient estimate expression (2.6), yet keeping the overarching stochastic gradient descent (applied with this estimate) the same. This section briefly discusses other, more sophisticated procedures that revise the classical SGD to stabilize the training further. We did not use these advanced techniques during our training, but they are common in other related works. Therefore, it is an interesting direction to explore to see how much it could speed up the convergence of our models.

Trust Region Policy Optimization (TRPO, [27]) is a reinforcement learning algorithm used to train policy-based models, particularly for continuous action spaces. TRPO constrains the policy update to ensure that the new policy is not too different from the old policy, where the "difference" is usually measured by the KullbackLeibler divergence (link). This is achieved by

defining a "trust region," which limits the size of the policy update in each iteration. Then, TRPO performs a line search or a similar optimization procedure to find the maximum policy improvement within the trust region, which allows for more flexibility rather than the regular gradient descent.

IMPALA (Importance Weighted Actor-Learner Architectures, [7]) is another distributed reinforcement learning algorithm that combines advantages of both on-policy and off-policy methods IMPALA improves the standard Actor-Learner Architecture that of [19], prioritizing experience replay to sample more relevant experiences for policy updates. This prioritization is based on the estimated impact of each experience on the learning process, allowing for more effective learning from limited data.

Finally, the natural gradient policy [13] derives a way to perform natural gradient descent in the RL setting. It computes both the gradient and Fisher Information Matrix estimates and then performs an update by making a step in the direction of natural gradient estimate, which is, on expectation, *unbiased*. It, therefore, adjusts the parameters to consider the curvature of the objective function. However, in its original formulation, this strategy suffers from the curse of dimensionality of the networks because the Fisher Information Matrix is too computationally expensive for deep models with millions of parameters.

This concludes our discussion of Deep RL and brings us to an important checkpoint. So far, all introduced methods were model-agnostic, i.e., they can be applied to any non-linear approximation model. Some models will work better than others, e.g., deeper networks can express more complex functions. As the MDP-based framework suggests, most Reinforcement Learning tasks are sequential; therefore, neural models that can handle sequential inputs effectively are preferable. The architecture that has made recent breakthroughs in numerous sequential tasks is the Transformer.

2.5 Transformers

Transformers were proposed by Vaswani et al. [31] as an architecture to efficiently model sequences. They consist of stacked layers of self-attention with residual connections. The attention mechanism allows us to learn how different inputs should "attend" to each other by maximizing the self-attention dot-products. In addition, positional encoding is concatenated to inputs to indicate the index i of the sequence.

We refer to the original papers for more specific architectural details. Still, it is important to note how transformers differ from the previous models used for RL tasks, such as RNN (Recurrent Neural Network) and LSTM (Long short-term memory network). RNNs are based on recurrent connections that propagate information sequentially through time by maintaining an internal hidden state updated at each time step t. LSTM was proposed to improve the RNN

architecture designed to address the vanishing gradient problem encountered in RNNs. In addition, it includes recurrent connections with gating mechanisms (such as input, forget, and output gates) that allow the network to update and forget information over time selectively.

Naturally, Transformers are better at capturing long-range dependencies in sequences than RNNs and LSTMs because the self-attention mechanism allows tokens to attend to any other token in the sequence, regardless of distance. This ability is enhanced as one increases the number of attention heads in the model, allowing one to attend to multiple parts of the input simultaneously during training and inference. In other words, different heads learn different attention patterns between inputs. In addition, Transformers are more scalable than RNNs / LSTMs because they do not have recurrent connections that limit parallelization.

In our work, we use the GPT architecture [25], which modifies the transformer architecture with a *causal* self-attention mask to enable autoregressive generation: inputs at step t are only attending to the previous tokens $i \leq t$. This correctly matches the settings of RL tasks, i.e., at step t, we only have access to the past but not the future observations. At the same time, this also allows for speeding up computation: outputs at different positions can be computed in parallel (without any issues of "look-ahead" bias precisely because of the masked-attention nature of GPT).

We now turn to discussing how Transformers can be applied in Reinforcement Learning. We start with the general RL problems (2.6) and then expand to meta-reinforcement learning settings (2.7).

2.6 Transformers and RL

At the intersection of Reinforcement Learning and research about Transformers, most works focus on using Transformers to solve RL and meta-RL tasks. However, some papers illuminate other areas, such as analysis of Transformer's behavior *from* the RL perspective. For instance, Melo [17] argues that the two critical capabilities of transformers compose their central role as a *Meta-Reinforcement Learner*. First, transformers can handle long sequences and reason over long-term dependencies, which is essential to the meta-RL agent to identify the MDP from a sequence of trajectories. Second, transformers present context-dependent weights from self-attention. This mechanism is a fast adaptation strategy and provides adaptability to the meta-RL agent for new tasks. In this way, the transformer's decision-making $f_t(s_1, s_2, ..., s_t)$ serves as the "episodic memory," and architecture recursively refines this episodic memory using the "working memory" presented via self-attention weights. In short, this work poses the transformer's architecture as a meta-reinforcement learner and thus suggests that this model already has an inductive bias toward solving different Reinforcement Learning tasks. We will now see how a few works have applied transformers in RL via sequence modeling.

2.6.1 RL as sequence modelling

In Reinforcement Learning, the <states, actions, rewards> history can be seen as a sequence modeling task: at each time step t transformer has to output "most probable" action $P(a_t|H_t)$, where the measure of "most probable" can be trained with a goal of maximizing some desirable objective, such as immediate rewards or the log-probability of the trajectory itself (the latter relates to learning the environment rather than optimizing for rewards).

For instance, Trajectory Transformer [12] mirrors the language-based approach of generating sentences as a sequence of word-tokens into RL tasks: they treat states, actions, and rewards simply as a stream of tokens that are being fed into the Transformer. The model sees s_t, a_t, and r_t as all the same type of input, and the overall goal is to maximize the probability of observing the current sequence. They use supervised loss function $\log P_\theta(s_t|\tau_{[:t]}) + \log P_\theta(a_t|\tau_{[:t]}) + \log P_\theta(r_t|\tau_{[:t]})$ and train the Transformer in the teacher-forcing manner on the dataset of offline RL trajectories.

In the "Decision Transformer" paper [3], a very similar approach was explored except that they only predict actions a_t (but not states s_t or rewards r_t). The training still happens in a teacher-forced manner with mean squared loss (MSE) $(a_t - a_{true})^2$. This method encourages the model to mimic decision-making presented by other agents in the given offline dataset of trajectories (for instance, one of the training datasets is 500 thousand transitions observed by an online DQN agent playing Atari games).

These works differ from ours for several reasons. In principle, we attempt to answer a more theoretical question: How rich is the function space the transformer can express in RL environments, and is gradient-based optimization suitable for achieving the global optimum in this space? To answer this question, we want to train the model with as little supervision as possible to see how well it can approximate a non-trivial Gittins index strategy. In contrast, these works ([12] or [3]) use teacher-forced learning and thus encourage the model to mimic the patterns of expert trajectories. Of course, we could also use the Gittins index offline trajectories in our problem, but that would make the training objective so much easier. Also, that approach does not answer the question of whether the transformer can find the global minimum in settings where the optimal solutions are unknown (for instance, finite horizon MABs with high discount factor γ). In addition, both papers found that the design of returns-to-go $R_t = r_t + ... + r_T$ was necessary for decent performance. This implies that models struggled to learn exploration vs. exploitation heuristics and tended to be greedy without human supervision.

Last but not least, none of these works pose the objective of meta-learning. For instance, in [3], each training dataset consists only of the trajectories for a single fixed game, and therefore, the fitted model only learns to perform well for that specific game rather than generalize to instances of other RL tasks. Therefore, we now transition to discussing works that indeed

discover transformers in the meta-learning settings.

2.7 Meta Learning

Meta learning is a well-known field first established in the late 20th century [26]. The goal is to learn inductive biases to produce a solution that generalizes well across a distribution of tasks rather than excels at one single task. It is also commonly called "learning to learn" because meta-learning focuses on learning the learning process. Therefore, a successful meta-learner performs well in tasks that did not appear in the training set. For example, the model that can output OLS solution $\hat{\beta}$ from a few samples of $\mathbf{x_i}, y_i$ (where, $y_i \sim \beta \cdot \mathbf{x_i} + \epsilon$), even if that sequence of inputs $\mathbf{x_i}, y_i$ did not appear in the training distribution (for example, inputs were shifted) can be seen as a *meta-learner* that found optimal OLS solution for the linear regression task.

Recently, pretrained LLMs like GPT3 [2] have demonstrated the behavior of meta-learners known as in-context learning (ICL). We are now going to provide a detailed discussion of this phenomenon.

2.7.1 In-context Learning

Models like GPT3 can generalize from specific input-output examples to generate contextually relevant outputs. In this way, they learned how to analyze the preceding text to inform the generation of subsequent text in various tasks. For instance, when given a prompt for mathematical calculations, coding tasks, or Google search requests, GPT-3 can produce relevant and coherent outputs for each prompt. Crucially, this happens without any updates or fine-tuning to the model's parameters, with tasks and few-shot demonstrations specified purely via text interaction with the model. This trend becomes more profound as the model becomes larger (both wider and deeper), as highlighted in the original GPT3 paper [2].

In this way, *in-context learning* can be seen as meta-learning: the LLM can perform optimally for the whole distribution of inputs of a singular task and, more importantly, across different tasks. One might say that large LLMs have been able to learn how, by looking just at a few samples, to understand **what** is the task (i.e., objective), and **how** to output optimal solutions for new, unseen inputs based on the task. However, this last point has not been fully understood or proved. Since LLMs like GPT3 have been trained on data all over the internet (about 45TB of text data), one could argue that their ability to infer tasks in context can be explained by the fact that models have seen similar or even the same data during the training. Therefore, an ongoing discussion remains on the extent to which these models perform inference from a few samples rather than memorization.

Recently, several works started a closer investigation of the Transformers's in-context learning capabilities. For instance, Garg et al. [9] empirically show that Transformers can be trained **from scratch**[1] to perform in-context learning of linear functions with performance comparable to the optimal least squares estimator. During training, the model receives (k input-output pairs) and the (k + 1)-th input: $P_i = (x_1, f(x_1), x_2, f(x_2), ..., x_i, f(x_i), x_{i+1})$ and outputs $\widehat{f}(x_{i+1})$, where f is a linear transformation sampled from some distribution. As f is random itself (sampled from the distribution), the model is trained as a meta-learner. Notably, their solution $\hat{\beta}$ that the transformer learns in-context remains close to the optimal $\hat{\beta}^{OLS}$ even under out-of-distribution shift of covariates and β (i.e., task f) itself. This suggests that the model progressed toward the meta-learning objective of "learning the fitting OLS algorithm" of the least squares rather than simply "memorized" the training data well.

Zhang et al. [36] also investigate the dynamics of ICL in transformers but with a single layer and linear self-attention module. Their training procedure is the same as in [9], but with a different function space f. They show that the model converges to a global minimum despite the non-convexity of the underlying objective function (they use anistropic Gaussian data). However, their work also finds that "although a number of out-of-training distribution shifts can be tolerated, shifts in the covariate distribution of the features x_i can not." This does not directly contradicts with the results of [9] as the latter used deeper and wider model (12 layers and 4 attention heads), and also different training function space f. Yet, Zhang et al. perform further analysis with larger models and random covariate distributions during the training, which improve OOD performance but still fails to converge to $\hat{\beta}^{OLS}$. In general, this paper theoretically proves and empirically demonstrates that in specific settings, the ICL of a single-layer transformer does not perform the OLS solution *exactly*.

The goal of our work is to perform a similar analysis, i.e., how successful is a transformer trained from scratch, but for reinforcement-learning problems and without in-context learning training. This guides us to the final section of the literature review: meta Reinforcement Learning.

2.8 Meta Reinforcement Learning

Meta Reinforcement Learning is a subarea of meta-learning for RL tasks. As discussed earlier, in reinforcement learning, the model's ability to handle process inputs in a sequential manner makes a big difference since the agent almost always has to recall the experience to optimize actions for the future. That is why sequential models, like RNNs / LSTMs, and later Transformers, started to get attention in this area: they combine the rich function space (thanks

[1] This means that parameters are initialized randomly, without pre-training on any dataset

to the non-linearities inside the networks) with memory-based mechanisms (due to the sequential order of the hidden states).

Our work falls precisely into this Meta RL universe, as we are trying to teach transformers to perform the best way possible in different multi-armed bandit settings. In this area, works primarily split into more task-agnostic algorithms (2.8.1), which can be applied to all RL tasks and task-specific algorithms that examine more closely some specified meta-RL tasks and model architectures (2.8.2).

2.8.1 Task-agnostic Meta Learners

Task-agnostic meta-learners aim to generalize across a variety of tasks without task-specific training. For this reason, such algorithms conceptually can be seen as theoretical frameworks for meta-learning of any model towards any task. In the same way, gradient descent is the theoretical framework for optimizing any model towards any objective/loss.

For instance, Finn et al. [8] decompose the meta-learning algorithm into two parts: the traditional learners initial parameters θ are trained to be suitable for fast stochastic gradient descent adaptation to different tasks. In the reinforcement learning setting, they sample a collection of tasks T_i from the target distribution. They adapt the current parameters $\theta \rightarrow \theta_i$ using a few steps of the policy gradient via interacting with this specific task T_i and achieving task-adapted loss $L(\theta_i)$. Finally, the initial parameters θ are updated, averaging losses across **different** tasks T_i. In this way, one can say that the model was precisely trained to become a meta-learner: the resulting meta-network θ^* should be able to adapt to different tasks quickly via several steps of gradient descent. This algorithm is known as Model Agnostic Meta Learner or MAML.

Another paper by Laskin et al. [15] presents the method called "Algorithm Distillation" (AD) that performs in-context offline reinforcement learning. In this paper, the *training histories* H_t of a source RL algorithms (e.g., on-policy actor-critic or DQN) are used as the context for the transformer to output the next actions, and the training tries to perform behavioral cloning over the source's RL algorithm decisions. As the authors argue, if the context is sufficiently large (i.e., across-episodic), it will include moments of policy improvements. Hence, the AD model will be able to learn not only an optimal policy for a specific task but also a policy "improvement operator." During evaluation, the interaction with a new task instance happens in an Autoregressive manner (initialize empty context and start unrolling trajectories). A successfully fitted AD model can predict optimal actions entirely in-context, without updates to the network's parameter, which differs from the aforementioned MAML [8]. Algorithm Distillation, therefore, can be applied to any sequential model and any distribution of tasks for which offline data with trajectories is available.

These approaches benefit from broad generality: they are compatible with any model and

applicable to different RL problems. However, they explicitly follow a specific strategy during the adaptation stage (namely, gradient descent of MAML or in-context learning of AD). In a particular domain, better strategies may exist that exploit the meta-RL task's structure. For instance, in our problem, we find that MAML underperforms on the bandits' task, at times even when compared to the baseline Thompson sampling, and significantly loses to the transformer and the Gittins index. In contrast, our work aims to arrive at those optimal solutions *exactly*, therefore addressing whether the transformer has the capacity to learn an algorithm that exploits domain-specific task structure.

2.8.2 Task-specific Meta Learners

Finally, this section considers the works closest to our paper, where the meta-learner was trained toward the specific Meta-RL tasks.

In the "Learning to Reinforcement Learn" paper [32], Wang et al. trained RNNs to solve, among other RL tasks, multi-armed bandits. The hidden state of RNN serves as a memory mechanism, and for training, they closely follow the A3C algorithm, modeling both policy decisions and value functions with the network. Although their findings suggest that the model is capable of key RL trade-offs, such as exploration vs. exploitation trade-off, and even outperforms the baseline Thompson sampling, the model did not obtain the oracle solution (i.e., Gittins index). Simultaneously, Duan et al. [6] proposed to use LSTMs, which improved the performance and became quite close to the Gittins index but still slightly underperformed. Comparing these methods to our paper, the transformer seems to have significant computational benefits. That is because the transformer's architecture is easier to train than traditional RNNs / LSTMs: the entire sequence can be processed in a single forward pass, whereas the RNN model has to accumulate all information up to t (via its hidden state) in t forward passes because of the temporally-linear hidden state dependency. Duan et al. also used a significantly larger batch size (up to 250000 samples) for successful convergence.

Mishra et al. [18] hypothesize that the traditional RNN's architecture bottlenecks its capacity to perform sophisticated computation on a stream of inputs: the only way to access the past information is through the hidden state. They believe that soft attention (introduced by [31]) allows pinpointing a more specific piece of information from an infinite context. Thus, they proposed the combination of causal temporal convolutions[2](to aggregate past information) and soft-attention (to pinpoint specific information), which they applied to bandits with training mirroring the A3C approach [19] plus TRPO.

In this way, their model SNAIL ("simple attentive learner") becomes quite close to the GPT2 transformer's architecture, yet some differences remain. First, transformers also have

[2]initially, they took this block of architecture from [30]

positional encodings that help utilize time index t more efficiently. This becomes a crucial component for time-dependent sequential tasks like finite MDPs. Second, the temporal convolutions in SNAIL provide high-bandwidth access at the expense of only finite context size. In particular, to attend N steps back, the network should have at least $O(\log N)$ layers. This implies that their learner only attends to a few recent episodes, explaining why the model fails to scale for large T and under-performs compared to the oracle (i.e., Gittins index). In contrast, our study utilizes several multi-head self-attention heads to stabilize the entire transformer architecture, accommodating an arbitrarily large context window.

More importantly and surprisingly, in the same paper, Mishra et al. tried to use the transformer architecture as well and claimed that "its performance was no better than **random**." In this work, we show that this is not the case, and even though the original transformer struggles to converge to the global optimum, it still finds a solution quite close to this optimum and outperforms RNNs from [32]. Ultimately, in the experiments chapter 4, we compare our results against the SNAIL and MAML models. Note: we do not include RNNs or LSTMs because the SNAIL has been shown to outperform both in all bandit tasks.

This concludes the overview of related works, and we are now ready to proceed to a more detailed discussion of the methodology used in this work.

Chapter 3

Methodology

In this section, combining numerous ideas mentioned in the literature review, we will formalize the approach used for training and evaluating the GPT2 transformer on the multi-armed bandit task. To clarify, our reference to "GPT2" has nothing to do with the pre-trained parameter values for language-related tasks (or other domains); instead, we train the transformer entirely from scratch. However, we use OpenAI's GPT2 API, which provides the implemented architecture. Therefore, whenever we reference "GPT transformer" or "GPT2 transformer", we imply the application of this API.

The content is outlined as follows:

(3.1) Formal notation for Meta RL framework and our objective.

(3.2) Theoretically optimal solution in multi-armed bandits for finite and infinite horizons.

(3.3) Derivation of the gradient direction for the history-dependent model.

(3.4) Variance reduction techniques.

(3.5) Implementation of the policy gradient descent with a custom Loss function.

(3.6) Symmetry Regularization.

(3.7) Training procedure.

3.1 Meta RL formalism

Broadly speaking, our goal is to train the GPT meta-learner to solve multi-armed bandits. But first, we will remind the reader of our objective's different math components.

Let the number of arms $K \geq 2$, horizon T, and discount $\gamma \in (0, 1)$ be the characteristics of the problem chosen and fixed initially. Denote $M = M(\mu_1, \mu_2, ..., \mu_K)$ an instance of multi-armed bandit (MAB) task with arm means $\mu_1, \mu_2, ..., \mu_K$, and rewards for pulling arm i distributed as $r_i \sim \text{Bernoulli}(\mu_i)$. The agent never observes the true μ_i but can interact with the MAB for T steps, and at every time step t choose whatever arm a_t seems to be the most reasonable to pull at that step according to their strategy, and then receive reward

$r_t \sim \text{Bernoulli}(\mu_{a_t})$. The goal for this *single* bandit task is to maximize the expected discounted total reward $R = \sum_{t=0}^{T} \gamma^t r_t$, where the uncertainty of expectation comes both from the stochasticity of rewards r_t and the agent's strategy.[1]

Suppose we also have the distribution $\mathbf{D}$ over these bandit tasks: $M \sim \mathbf{D}$. To be more precise, this implies there is a distribution over arm means $(\mu_1, \mu_2, ..., \mu_K) \sim \mathbf{D}$, but we will primarily use the former notation for simplicity since by definition there is a bijection between bandit tasks and underlying vector of means $(\mu_1, \mu_2, ..., \mu_K) \in [0, 1]^K$. For instance, the main distribution of interest in our problem is $\mathbf{D} = \text{Unif}(0, 1)^K$, i.e., each of the K arm means μ_i is independently drawn from $\text{Unif}(0, 1)$.

Finally, suppose we can parameterize our RL agent with θ as $f_\theta : H \to \mathbb{R}^K$, which is a function that takes a history of past actions and rewards $H_t = (a_0, r_0, ..., a_{t-1}, r_{t-1})$ as its input and returns a vector $q_1, ..., q_K$ of K real values that correspond to the logits for pulling arms 1 to K. This guides RL agent to choose the next action a_t by calculating the probabilities p_i with the Softmax function

$$p_i = \frac{\exp(q_i)}{\sum_{j=1}^{K} \exp(q_j)}$$

and then take one sample $a_t = \text{Multinomial}(p_1, ..., p_K).\text{sample}()$. We always imply that the policy π_θ is probabilistic and outputs K values in a way we just described unless the opposite is claimed explicitly.

Further on, we will omit this link from logits to softmax for simplicity and refer to $a_t \sim \pi_\theta(H_t)$ to be the action-making distribution $\text{Multinomial}(p_1, ..., p_K)$ resulting from f_θ as described above. We introduce this new notation π_θ rather than f_θ because the former refers to the policy decision-making parameters, while the latter refers to all transformer parameters. Currently, they are the same set of parameters, but later, we also introduce value function V_θ, which will have an additional separate layer of parameters. f_θ can be seen as a general parametrization of any black box model f, but in our case specifically, we imply all weights and bias matrices of the transformer.

Hence, for any given MAB M we can see that the overall episode trajectory $\tau = (a_0, r_0, ..., a_{t-1}, r_{t-1})$ of interactions between the agent and the MAB depends on both the agent's parameterization f_θ and the bandit: $\tau \sim (M, f_\theta)$. Meta Reinforcement Learning objective is to maximize the expected reward $R(\tau)$ across the whole distribution $\mathbf{D}$:

$$\mathbb{E}_{M \sim D} \left(\mathbb{E}_{\tau \sim (M, f_\theta)} R(\tau) \right),$$

[1] To be more precise, the sum $\sum_{t=0}^{T} \gamma^t r_t$ involves T + 1 rewards, not T. Later, we explain that the first action a_0 is generated manually by random, not by the agent, so the agent indeed makes T = (T + 1) - 1 decision themselves. We elaborate on this moment in section 3.3

with $R(\tau) = \gamma^0 r_0 + \gamma r_1 + ...\gamma^T r_T$. Our objective minimizing function is thus defined as the

$$J(\theta) = (-1) \cdot \mathbb{E}_{M \sim D} \left[\mathbb{E}_{\tau \sim (M, f_\theta)} R(\tau) \right] \tag{3.1}$$

and we aim to find

$$\theta^* = \arg\min_\theta J(\theta).$$

In this way, we are doing *meta-learning* because we want to maximize expected reward across the distribution of all tasks $\mathbf{D}$, rather than for single MDP M_i. The motivating problem is the classical Bayesian MAB. It is called "Bayesian" because the policy (a rule for choosing arms) aims to maximize the Bayes return: averaging over both prior distribution $\mathbf{D}$ and realizations of the environment τ.

Before going into the details of solving this optimization problem (i.e., training our transformer), it is crucial to understand how one could evaluate the found solution θ^*. In principle, even if the optimization is solved correctly (i.e., θ^* is the actual global minimum among all functions f_θ), the resulting policy f_{θ^*} should not necessarily be the most optimal strategy: the function space of the transformer can simply be too constrained (which is probably indeed the case for a small transformer). Fortunately, for the Bayesian MAB problem, we know the most optimal strategy (i.e., the one that maximizes the Bayes return). Thus, its resulting regret can be a fair benchmark for assessing the transformer's success/failure.

3.2 Optimal solution in the multi-armed bandits

The most optimal strategy for the infinite horizon discounted Bayesian MAB is the Gittins index policy [10]. This strategy represents a non-trivial dynamic solution involving a high-calibrated recursion and binary search – we refer the detailed derivation to the original paper.

Nevertheless, there are several high-level moments we would like to note about the Gittins index. First, it is independent across arms and is based only on $2K$ numbers: the number of successes and failures in the past for each of the K arms. Precisely, the scalar value called "the index" is calculated independently for each arm (i.e., rewards from one arm do not influence the index for another arm) at every time step t. Then, the strategy pulls the arm with the maximizing index. The Gittins index, therefore, refers to both this scalar value per arm and the overall strategy, and the meaning should be inferred from the context. Second, it also considers the prior $\mathbf{D}$ for the arms μ_k and remains optimal for any pre-specified prior (not only Uniform). Hence, we use a different prior in some of our experiments as discussed in section 4.2.2.

The Gittins index is optimal only in the case of **infinite** horizon discounted Bayesian MAB. In reality, we can train the transformer only for finite horizon T, for which the *exact* solution

for MABs is unknown. However, for any given discount γ, starting from a particular horizon $T \geq T_0(\gamma)$, Gittins index with the same discount γ approximates the best solution for the finite horizon MAB$(\gamma, T_0(\gamma))$ very well. This is simply because starting from a certain point, γ^t becomes very close to 0, and therefore all rewards past the time step $t \geq T_0(\gamma)$ would constitute negligible impact on the decision-making because of discounting by γ^t. For instance, with $\gamma = 0.95$, $\gamma^{100} \approx 0.005$, so Gittins index with discount 0.95 and the best solution for bandits with $\gamma = 0.95$, T = 100 should be very close. Now that we can approximate the most optimal solution very well, we come back to solving the optimization 3.1.

3.3 Gradient estimate

The following three sections (3.3 - 3.5) attempt to solve the optimization problem with one overarching idea: the gradient descent. However, as discussed in the literature review, many adjustments to the standard GD should be made to make it work, which we discuss in detail here.

We first proceed by deriving an unbiased estimate for the gradient $\nabla_\theta J(\theta)$ in (3.1) similarly to the REINFORCE method [34]. Typically, we would simply try to put differentiation sign ∇_θ inside the expectation $\mathbb{E}_{\tau\sim(M,f_\theta)} R(\tau) = \int_\tau (p_\theta(\tau)R(\tau))d\tau$, where $p_\theta(\tau)$ is the distribution of trajectories resulting from the model f_θ. Yet, the challenge is that this distribution $\tau \sim p_\theta(\tau)$ depends on parameters θ itself, so $\nabla_\theta \mathbb{E}_{\tau\sim(M,f_\theta)} R(\tau) \neq \mathbb{E}_{\tau\sim(M,f_\theta)} \nabla_\theta R(\tau)$ (in fact, this last part is just zero, because $R(\tau)$ does not depend on θ). Fortunately, we can handle this challenge by noticing that $\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \cdot \nabla_\theta \log p_\theta(\tau)$, which is usually called the *likelihood ratio* method. This allows to express the gradient as the expectation over both M and τ with the differentiation of log-probabilities $\log p_\theta(\tau)$ inside:

$$\nabla_\theta J(\theta) = \nabla_\theta \left[-E_{M\sim D}\, \mathbb{E}_{\tau\sim(M,f_\theta)} R(\tau)\right] = -E_{M\sim D}\left(\nabla_\theta \left[\mathbb{E}_{\tau\sim(M,f_\theta)} R(\tau)\right]\right) =$$

$$-E_{M\sim D}\left(\int_\tau \nabla_\theta \left[p_\theta(\tau)\cdot R(\tau)\right] d\tau\right) = -E_{M\sim D}\left(\int_\tau \left[p_\theta(\tau)\cdot \nabla_\theta \log(p_\theta(\tau))R(\tau)\right] d\tau\right) =$$

$$E_{M\sim D}\left[E_{\tau\sim(M,f_\theta)}\left[\nabla_\theta \log p_\theta(\tau) R(\tau)\right]\right].$$

In the case of bandits, $\log p_\theta(\tau)$ rewrites as

$$\nabla_\theta \log p_\theta(\tau) = \left[\nabla_\theta \log \pi_\theta(a_1|a_0, r_0) + \nabla_\theta \log \pi_\theta(r_1|a_0, r_0, a_1)\right] + ...$$

$$...+\left[\nabla_\theta \log \pi_\theta(a_t|a_0, r_0, ..., a_{t-1}, r_{t-1}) + \nabla_\theta \log \pi_\theta(r_t|a_0, r_0, ..., a_{t-1}, r_{t-1}, a_t)\right]+... = \sum_{t=1}^{T} \log \pi_\theta(a_t|H_t)$$

Parts $\pi_\theta(r_i|a_0, r_0, ..., a_{i-1}, r_{i-1}, a_i)$ cancel out because they do not depend on θ and only depend on the $\mathrm{Bernoulli}(\mathrm{a_i})$ distribution. In this equation, we have not said anything about $\log \pi_\theta(a_0)$.

In principle, we can ask the model to make the very first action a_0 too, and in the case of asymmetric priors (where some arms have apriori higher means than others), the model should learn which action is the most beneficial. In this work, however, we work only with symmetric priors, and we thus decided to generate the first actions by ourselves. This also allows us to compare the model and the Gittins index more directly on the same trajectories (which we do in the experiments section) when they both start with the same first pair a_0, r_0 (thus, we cancel out any difference that could come just from the randomness of this first action-reward observation).

Finally, putting things together, we get the gradient expression:

$$\nabla_\theta J(\theta) = -\mathbb{E}_{M\sim D}\left(\mathbb{E}_{\tau\sim(M,f_\theta)}(\sum_{t=1}^{T}\nabla_\theta \log\pi_\theta(a_t|H_t))\cdot R(\tau)\right) \tag{3.2}$$

3.4 Variance reduction

As discussed in the literature review, the equation in (3.2), though unbiased, is a very unstable estimate of the gradient (i.e., high variance). Works in the past have shown that variance reduction techniques are crucial for the successful convergence of Deep RL networks. In our work, we implement two of these techniques: baselines and generalized advantages.

3.4.1 Baselines

The general idea of baselines is to mathematically modify the inner part of expectation in (3.2) so that the gradient still remains unbiased, but the variance goes down. The original REINFORCE work [34] shows that any function b_t of the MDP's state s_t can be subtracted at step t from $R(\tau)$ resulting in $\sum_{t=1}^{T}(\log\pi_\theta(a_t|s_t, a_{t-1}, r_{t-1})\cdot(R(\tau) - b_t(s_t))$. We remind the reader that in our problem, we work with MAB instead of MDP, so there are no states for s_t. However, we can still apply baselines. The proof below shows that the baselines can depend on any history $H_t = (a_0, r_0, ..., a_{t-1}, r_{t-1})$, not just single s_t. The proof completely mirrors the one in [34] for the original baseline expression, so the reader should feel free to skip it if they are familiar with it. Yet, it is important to check that the same proof works for the history-dependent baselines $b_t(H_t)$ because this is how the transformer model makes its decision and leverages the attention mechanism (as opposed to RNNs, for instance, which would store everything in one hidden state at time t, and so $b_t(s_t)$ would be a reasonable choice then).

Proposition: In equation (3.2) we can subtract any function $b(H_t)$ (dependent only "on the

past" history) from $R(\tau)$ at time t and still get an unbiased estimate of the gradient:

$$\mathbb{E}_{\tau\sim(M,f_\theta)}\left((\sum_{t=1}^{T}\log\pi_\theta(a_t|H_t))\cdot R(\tau)\right)=\mathbb{E}_{\tau\sim(M,f_\theta)}\left(\sum_{t=1}^{T}[\nabla_\theta\log\pi_\theta(a_t|H_t)\cdot(R(\tau)-b_t(H_t))]\right) \tag{3.3}$$

Proof. It suffices to prove that for every MAB M and t:

$$\mathbb{E}_{\tau\sim(M,f_\theta)}\left[\nabla_\theta\log\pi_\theta(a_t|H_t)\cdot b_t(H_t)\right]=0$$

as the remaining part $\log\pi_\theta(a_t|H_t))R(\tau)$ is the same on both parts of the equation (3.3). Using Adam's Law and conditioning on the history H_t, we get:

$$\mathbb{E}_{\tau\sim f_\theta}\left[\nabla_\theta\log\pi_\theta(a_t|H_t)\cdot b_t(H_t)\right]=\mathbb{E}_{H_t}(\mathbb{E}_{\tau_{[t:]}}\left[\nabla_\theta\log\pi_\theta(a_t|H_t)\cdot b_t(H_t)\right]|H_t)=$$

$$\mathbb{E}_{H_t}(\mathbb{E}_{\tau_{[t:]}}\left[\nabla_\theta\log\pi_\theta(a_t|H_t)\right]\cdot b_t(H_t))$$

where $\tau_{[t:]}$ refers to the distribution of trajectories drawn from $f_\theta|H_t$, i.e. when the model observed H_t. It remains to notice that the inner part with the logarithm cancels out:

$$\mathbb{E}_{\tau_{[t:]}}\left[\nabla_\theta\log\pi_\theta(a_t|H_t)\right]=\nabla_\theta(\sum_{a_t}\pi_\theta(a_t|H_t))=\nabla_\theta(1)=0$$

The last modification is due to the likelihood ratio method. □

Using (3.3), we first choose to work with $R(\tau[t:])=\gamma^t r_t+...+\gamma^T r_T$, instead of the overall sum $R(\tau)=\gamma^0 r_0+...+\gamma^T r_T$, which immediately reduces the noisiness due to the variance of the first t rewards. This can be done because the baseline $\gamma^0 r_0+..\gamma^{t-1}r_{t-1}$ is indeed a function of H_t (NB: H_t does not contain r_t, which will be observed only after taking action a_t, and hence we have to leave it as a part of $R(\tau[t:])$.
Furthermore, in order to reduce variance of the remaining part $R(\tau[t:])-b_t(H_t)$, we aim to choose $b_t(H_t)=\mathbb{E}(R(\tau[t:])|H_t)=\mathbb{E}(\gamma^t r_t+\gamma^{t+1}r_{t+1}+...|H_t)$, i.e. expected future reward conditioning on seeing the current history. This is a well-known fact and can be shown by Eve's law: if we denote $Y=R(\tau[t:])=\gamma^t r_t+\gamma^{t+1}r_{t+1}+...+\gamma^T r_T)$, then

$$\begin{aligned}\mathrm{Var}(Y-b_t(H_t))=\mathbb{E}(\mathrm{Var}(Y-b_t(H_t)|H_t))+\mathrm{Var}(\mathbb{E}(Y-b_t(H_t)|H_t))=\\ \mathbb{E}(\mathrm{Var}(Y|H_t))+\mathrm{Var}(\mathbb{E}(Y|H_t)-b_t(H_t))\geq\mathbb{E}(\mathrm{Var}(Y|H_t)),\end{aligned} \tag{3.4}$$

and this lower bound is achieved when $b_t(H_t)=\mathbb{E}(Y|H_t)$.
The baselines $b_t(H_t)=\mathbb{E}(\gamma^t r_t+\gamma^{t+1}r_{t+1}+...|H_t)$ are unknown, and therefore following [19] we model them as part of the network: $V_{\theta_V}(H_t)\approx\mathbb{E}(r_t+\gamma^1 r_{t+1}+...|H_t)$, i.e.

$V_{\theta_V}(H_t) \approx b_t(H_t)/\gamma^t$. We take out the factor γ^t as we do not want the model to struggle to learn discount γ^t at each t. In other words, all learned values $V_{\theta_V}(H_t)$ will be on the same scale instead of going down in magnitude with every time step. These value functions give a modified (still unbiased!) expression for the policy gradient:

$$\nabla_\theta J(\theta) = -\mathbb{E}_M \mathbb{E}_{\tau\sim M, f_\theta}\left((\sum_{t=1}^{T} \nabla_\theta \log\pi_\theta(a_t|H_t)) \cdot (R_{[t:]}(\tau) - \gamma^t V_{\theta_V}(H_t))\right) \tag{3.5}$$

To encourage the model to learn this value function, we will penalize deviations between $(r_t + \gamma^1 r_{t+1} + ...\gamma^T r_T)$ and $V_{\theta_V}(H_t)$. We will discuss this further when defining the loss function (section 3.5). Also, as mentioned in the literature review, $R_{[t:]}(\tau) - \gamma^t V_{\theta_V}(H_t)$ is the estimate for the Advantage of taking action a_t with respect to the baseline $\gamma^t V_{\theta_V}(H_t)$, except that we usually do not discount the advantage by γ^t.

3.4.2 Generalized Advantages

In equation (3.5), we still had an *unbiased* estimate for the gradient. Generalized Advantages Estimation (GAE) paper [28] suggests breaking this assumption: introducing slight bias while significantly reducing the variance of this estimate. We will show a short derivation for GAEs in the case of history-dependent value functions $V(H_t)$, but again, it mirrors the state-based $V(s_t)$ approach in the original paper. First, let us provide a couple of mathematically different expressions for the advantage function estimate. Let $V(H_t) = V(H_t;\theta_v)$ be an approximate value function (for simplicity, we will omit θ_V), and let us define

$$\delta_t^V = r_t + \gamma V(H_{t+1}) - V(H_t), \tag{3.6}$$

i.e., the TD(1) residual of V with discount γ. This one-step look-ahead δ_t^V can be seen as an estimate for the advantage of the action a_t given the current history H_t. In fact, when V equals to the true value V^π, the delta becomes an *unbiased* :
$A^\pi(H_t, a_t) = \mathbb{E}_{r_t}\left(r_t + \gamma V^\pi(H_{t+1}) - V^\pi(H_t)\right)$.

Now, if we take a γ-discounted sum of these deltas until the last horizon T (or until ∞ in case of the infinite horizon), we get exactly the advantage estimate from (3.5):

$$\begin{aligned}\sum_{k=0}^{t+k=T} \gamma^k \delta_{t+k}^V = (r_t + \gamma V(H_{t+1}) - V(H_t)) + \gamma\cdot(r_{t+1} + \gamma V(H_{t+2}) - V(H_{t+1})) + ... = \\ (r_t - V(H_t)) + \sum_{k=1}^{t+k=T} \gamma^k r_{t+k} = \gamma^{-t} R_{[t:]}(\tau) - V(H_t),\end{aligned} \tag{3.7}$$

since all the intermediate terms V_{t+k} cancel out (the last one $V_T = 0$).

Observe how these expressions (3.6) and (3.7) relate to the temporal differences (2.3) and (2.4) introduced in the Q-learning. Every single delta δ_t^V serves as a one-step look-ahead advantage estimate of a current agent, similar to TD(1). Such an estimate is "greedy" and might be biased unless we have the precise value network V. Thus, an optimal agent should look ahead "further," ideally into infinity, calculating the sum of all deltas (because it is indeed unbiased) – similar to TD(n) calculations. However, δ_t^V has a benefit of much lower variance (only one reward term r_t), while the sum of all deltas has a way larger variance (it has a long sum of rewards up to the last time step T).

The Generalized Advantage Estimate compromises between the one-step and long-step estimates by computing the exponentially weighted sum of these delta values:

$$A^{GAE(\gamma,\lambda)} = \sum_{k=0}^{t+k=T} (\gamma\lambda)^k \cdot \delta_{t+k}^V \tag{3.8}$$

where $\lambda = \lambda_{GAE}$ is a pre-specified constant that controls the bias-variance tradeoff. There are two extreme cases with $GAE(\gamma, 1) = R_t(\tau) - V(H_t)$ and $GAE(\gamma, 0) = \delta_t^V$. Eventually, λ becomes a hyper-parameter that we should tune. Incorporating GAEs into the previous gradient expression (3.5) is quite simple: instead of $R_t(\tau) - V(s_t, \theta_v)$, we use the generalized advantage, which can be calculated recursively from $t = T$ to 1 (at each t, we discount the previous value for $t + 1$ by $\gamma\lambda$ and add TD residual δ_t^V). The optimal value of λ depends on the task, and for instance, in the original paper [28], they found $\lambda \approx 0.9$ to work well. In our setting, we start with 0.3 and then progressively increase it up to 1 as the training converges. This is further discussed in detail in the training procedure section 3.7.

Finally, substituting GAEs instead of the regular advantages in (3.5), we get the updated policy gradient:

$$\nabla_\theta \widehat{J}(\theta, \lambda_{GAE}) = -\mathbb{E}_M \mathbb{E}_{\tau\sim M, f_\theta} \left((\sum_{t=1}^{T} \nabla_\theta \log\pi_\theta(a_t|H_t)) \cdot A^{GAE(\gamma,\lambda)}(\tau) \right), \tag{3.9}$$

where $A^{GAE(\gamma,\lambda)}(\tau)$ depends on τ via rewards and value function $V(H_t) = V_{\theta_V}(H_t)$ as in the definition of δ_t^V (3.6).

3.5 Loss Function and modifications

Expression in (3.9) is the desired gradient, but implementing it by calculating the individual vectors $\nabla_\theta \log\pi_\theta(a_t|H_t)$, and then summing across all T horizons would be pretty tedious implementation-wise. Luckily, Pytorch (or Tensorflow) allows us to leverage the encapsulated implementation of function differentiation: all we need is to define the custom loss function, for which gradient would match the one in (3.9). Foreseeing that the reader could feel

confused and lost with a new series of math equations, we reassure that this is *almost* the same as simply removing the differentiation ∇_θ sign from the (3.9). Namely, we claim that such loss function can be defined as follows:

$$L_P(\theta, \lambda_{GAE}) = -\mathbb{E}_M \mathbb{E}_{\tau\sim M, f_\theta}\left(\sum_{t=1}^{T}(\log\pi_\theta(a_t|H_t)\cdot \widetilde{A}^{GAE(\gamma,\lambda)}(\tau)\right) \tag{3.10}$$

$$\Rightarrow \nabla_\theta L_P(\theta, \lambda_{GAE}) = \nabla_\theta \widehat{J}(\theta, \lambda_{GAE})$$

Note that the difference with (3.9) is only in dropping ∇_θ and adding "tilde" on top of $\widetilde{A}^{GAE(\gamma,\lambda)}(\tau)$, which stands for the "frozen" version of the original $A^{GAE(\gamma,\lambda)}(\tau)$. This means that the loss function should treat this advantage as **constant** with respect to parameters θ, even though it depends on θ (recall, GAE is the weighted sum of δ_t^V, which expands as a sum of rewards r_t and $V_\theta(H_t)$, so the latter indeed depends on θ). The "frozen" version can be easily achieved in PyTorch by the .detach() function (detaching this variable from the gradient dependency graph). Therefore, when taking the gradient ∇_θ of this loss, we move differentiation inside to get $\nabla_\theta\left(\log\pi_\theta(a_t|H_t)\cdot\widetilde{A}^{GAE(\gamma,\lambda)}(\tau)\right) = \left(\nabla_\theta\,\pi_\theta(a_t|H_t)\right)\cdot A^{GAE(\gamma,\lambda)}(\tau)$. The policy loss function $L_P(\theta, \lambda_{GAE})$ might be quite unintuitive from the traditional Deep Learning perspective: this loss is not guaranteed to go down during training and will not go down. This loss function has nothing to do with the actual loss we care about, which is the total regret averaged across all tasks and all trajectories. This loss should be treated purely as a surrogate to make the "desired" gradient descent possible simply via the loss differentiation implemented by Pytorch. We denote the above loss as $L_P(\theta, \lambda_{GAE})$ with subscript "P" because it corresponds to the "policy component" of our combined loss, and we will now cover the value $L_V(\theta)$ and entropy regularization $L_{entropy}(\theta)$ components.

As for the **value loss**, we add the mean squared error of the deviations $\Delta V_t = (r_t + \gamma^1 r_{t+1} + ...\gamma^{T-t} r_T) - V_{\theta_V}(H_t) = \gamma^{-t} R_{[t:]}(\tau) - V_{\theta_V}(H_t)$ between the observed rewards and the model's value. These deviations are summed up across all t, and averaged across all MABs:

$$L_V(\theta) = \mathbb{E}_{M\sim D, \tau\sim f_\theta}(\sum_{t=1}^{T}\gamma^t \Delta V_t^2) = \mathbb{E}_{M\sim D, \tau\sim f_\theta}(\sum_{t=1}^{T}\gamma^t\cdot\left[\gamma^{-t}R_{[t:]}(\tau) - V_{\theta_V}(H_t)\right]^2). \tag{3.11}$$

We discount with γ^t each of ΔV_t^2 because we want to value and policy losses aligned in scale for every t. Otherwise, the losses associated with value deviations would dominate the gradient estimates for policy actions in later horizons as t approaches T.

As for the **entropy loss**, A3C paper [19] suggested adding the entropy of the policy to the objective to discourage the network from premature convergence to deterministic policies. In our problem, we indeed noticed that the model becomes "greedy" very fast (i.e., decides to

prioritize one of the arms over others, almost always pulling that arm) if we do not impose the entropy regularization. This additional component is:

$$L_{entropy}(\theta) = \mathbb{E}_{M\sim D,\tau\sim f_\theta}(\sum_{t=1}^{T}\gamma^t H(\pi_\theta(H_t))), \tag{3.12}$$

where $H(\pi_\theta(H_t)) = -\sum_{k=1}^{K} p_k \cdot \log p_k$ with $(p_1, ..., p_K)$ being a vector of probabilities for pulling arms 1 to K, and H is the entropy function. We discount by γ^t for the same reason as with the value loss, but also it is reasonable to assume that further in the episode (as $t \to T$), the policy can become more and more deterministic.

3.5.1 Combining all losses together

Finally, a weighted sum of the policy (3.10), value (3.11), and entropy (3.12) losses is our final total loss:

$$L(\theta) = L_P(\theta, \lambda_{GAE}) + c_V \cdot L_V(\theta) + c_{entropy} \cdot L_{entropy}(\theta) \tag{3.13}$$

where c_V and $c_{entropy}$ are the optimization hyperparameters to balance between the losses, which we discuss in section 3.7.

During training, we cannot compute either of the three losses $L_P(\theta, \lambda_{GAE}), L_V(\theta), L_{entropy}(\theta)$ exactly (that would require integrating over all MABs and trajectories τ), so this expression is instead replaced with the Monte-Carlo estimates. For instance, the estimate for the policy loss is:

$$\widehat{L_P}(\theta) = -\frac{1}{B}\sum_{\tau_i}\left((\sum_{t=1}^{T}\log\pi_\theta(a_t|H_t)) \cdot \widetilde{A}^{GAE(\gamma,\lambda)}(\tau_i)\right), \tag{3.14}$$

where the inside part is calculated with respect to the trajectory τ_i. The Monte-Carlo estimates $\widehat{L_V}(\theta), \widehat{L}_{entropy}(\theta)$ are defined similarly. Here, τ_i are generated by:
1) drawing MAB M_i from the distribution of arms $\mathbf{D}$ and generating first actions r_0^i by random;
2) rolling out the trajectory τ_i by iterating over $t = 1...T$ and interacting with bandit M_i. The algorithm's pseudocode 2 provides a detailed description.

This brings us to a critical checkpoint: everything that we introduced up to this moment (3.3-3.5) has recently become standard techniques in Deep RL that work well in many domains. We just modified these techniques so that they can be applied to the history-dependent policy $\pi_\theta(H_t)$ such as the GPT transformer. However, looking ahead to the results section, we discovered that these methods were insufficient to converge to the optimum, and the next section will demonstrate the last modification we applied to our procedure: symmetrization.

3.6 Symmetry regularization

The combination of policy, value, and entropy losses as established in (3.13) were not enough to achieve the optimal solution in our problem. Even though the transformer can find strategies better than random or even Thompson sampling (better in terms of expected reward), it always faces one unresolvable challenge: lack of *symmetry*. With this new concept, several questions arise immediately. What does the symmetry mean? Why should the optimal policy be symmetric or asymmetric? And how do we evaluate the model's symmetry? Let us address these questions one by one.

First, symmetry refers to treating any pair of two arms equally whenever we have the "same past data" for them, i.e., the same time steps t when the arm was pulled and the same result r_t. For example, at step $t = 0$, before any arms were pulled, we cannot favor any arm over another because our prior is the same for all of them: Unif(0,1). In general, for any pair of indexes $1 \le i < j \le K$, if we consider two histories H_t and augmented $\widetilde{H_t}$, where the latter was obtained from the former by simply swapping all actions $a_t = i$ to $a_t = j$ and vice versa, then symmetric policy π_θ should output the same probability vectors $\pi_\theta(H_t)$ and $\pi_\theta(\widetilde{H_t})$, except that the values at indexes i and j will be swapped.

We hope this definition of symmetry makes sense to the reader and that the optimal policy, intuitively, should be symmetric: when swapping the labels of arms (i with j), our beliefs about these arms should simply swap with each other. The optimal strategy, i.e., the Gittins index, is indeed symmetric because when calculating the index of every arm, it relies only on two numbers (Number of successes, Number of failures) for this specific arm. Therefore, when we swap two arms i and j, these stats will be swapped, and the same happens to Gittins indexes of the arms.

One way to evaluate the model's symmetry is to observe its performance on MAB M_k with mus = $(\mu_1, ..., \mu_K) = (0, ..., 1, ...0)$, i.e., all zeros except for arm k with $\mu_k = 1$. Symmetric performance implies that the expected reward $R_k = \mathbb{E}_{\tau \sim (M_k, f_\theta)} R(\tau)$ for all of these bandits $M_1, ..., M_K$ should be the same. During the training, we can track how the vector $(R_1, ..., R_K)$ "evolves," for example, by running an experiment every 10th training step: roll out 100 trajectories of length T with each of M_i and estimate $\hat{R}_i$. We usually find that the transformer learns in a "greedy" manner. Namely, at the beginning of training, the vector $(\hat{R}_1, ..., \hat{R}_K)$ is very similar to $(T/2, ..., T/2)$, which is the performance of a random policy (makes sense). Then, a quick spike in one of $\hat{R}_i$ happens, while others remain close to $T/2$. At this moment, the model becomes biased towards one of the arms. Later on, another arm "joins" in the group of favored ones, and so on, until all arms are learned to be pulled when they have high true mean $\mu = 1$. However, even though all of the arms eventually will be "recognized," the arm that the model recognizes the latest, denote it a_{last}, never "fully catches up" with the earliest

one, a_{first}. In other words, there is a clear gap between the two $R_{a_{first}} - R_{a_{last}} > 0$, which never vanishes no matter how long the model is trained (even when we decrease learning rate).

Interestingly, whenever such asymmetry happens, the model turns out to be sub-optimal overall, i.e., the overall expected reward across all MABs is lower than the optimal Gittins index. We therefore hypothesize that this asymmetry gap $R_{a_{first}} - R_{a_{last}} > 0$ is partially or fully responsible for the overall model's sub-optimality. To investigate this phenomenon, we developed two techniques to alleviate the asymmetry: (1) making the model symmetric by the definition or (2) penalizing the model for asymmetric behavior. To emphasize, our primary motivation for these techniques is to understand how much these techniques improve the transformer's performance and convergence stability rather than introduce a new architecture that "fixes" the asymmetry. This allows us to empirically evaluate how much the asymmetry bias of the original transformer is responsible for the gap from the optimal strategy (i.e., Gittins index). The following two subsections discuss how we encourage the model to be symmetric, whereas the resulting effect is presented in the experiments chapter.

3.6.1 Symmetrization

To encourage the model to be symmetric, we can make it symmetric *by definition*. Our first method achieves this by changing the model's forward pass. But first, let us formally define the symmetry property. In the problem of multi-armed bandits, model f_θ is considered to be symmetric if and only if for any permutation of arm indexes $\sigma : \{1, .., K\} \xrightarrow{\text{B}} \{1, .., K\}$ ("B" stands for bijection) and actions-rewards history $H_t = (a_0, r_0, ..., a_{t-1}, r_{t-1})$, model's forward pass operator and permutation σ on arm indexes are interchangeable:

$$\sigma(f_\theta(H_t)) = f_\theta(\sigma(H_t)). \tag{3.15}$$

In this equation, the left side applies permutation σ on the resulting vector $f_\theta(H_t)$ of K probabilities, while the right side applies σ on history H_t first (meaning it changes all actions a_i to $\sigma(a_i)$, keeping rewards the same: $\sigma(H_t) = \sigma(a_0), r_0, ..., \sigma(a_{t-1}), r_{t-1}$). In this way, the discussion of symmetry in the previous section is a particular case of σ that swaps i with j while keeping other indexes in their original places. Note that (3.15) is also equivalent to $f_\theta(H_t) = \sigma^{-1}(f_\theta(\sigma(H_t)))$, which can be also interpreted as operator $\sigma^{-1} \circ f_\theta \circ \sigma$ is the same as f_θ on bandit histories H_t. Now we are ready to define the "Symmetrization" transformation: for any model f_θ, define f^S_θ as follows:

$$f^S_\theta(H_t) := \text{Ave}_\sigma\{\sigma^{-1}(f_\theta(\sigma(H_t)))|\forall\sigma : \{1, .., K\} \xrightarrow{\text{B}} \{1, .., K\}\}, \tag{3.16}$$

i.e. we apply $\sigma^{-1}(f_\theta(\sigma(.)))$ operator on H_t for all possible permutations σ and average out results. It is easy to see that this definition makes the model f_θ^S symmetric: assume we have any permutation σ_0 and we want to check $f_\theta^S(H_t) \stackrel{?}{=} \sigma_0^{-1}(f_\theta^S(\sigma_0(H_t)))$. Substituting the definition of f_θ^S on the right side:

$$\sigma_0^{-1}(f_\theta^S(\sigma_0(H_t))) = \sigma_0^{-1}(\mathrm{Ave}_\sigma\left[\sigma^{-1}(f_\theta(\sigma(\sigma_0(H_t))))\right]) = \mathrm{Ave}_\sigma\left[\sigma_0^{-1}(\sigma^{-1}(f_\theta(\sigma(\sigma_0(H_t)))))\right]) =$$

$$\mathrm{Ave}_\sigma\left[(\sigma \circ \sigma_0)^{-1}(f_\theta((\sigma \circ \sigma_0)(H_t)))\right] = f_\theta^S(H_t),$$

where the second equation follows because Ave and σ_0^{-1} operators are interchangeable, the third equation simply combines σ and σ_0 into the composition of permutations, and last equation follows because $\sigma \circ \sigma_0$ still spans all permutations as σ did before.

There are several drawbacks to this method. From the computation perspective, it works perfectly with $K = 2$ arms, but for $K > 2$, we start to face a combinatorial explosion. There are $K!$ different permutations. Thus, every forward pass becomes either $K!$ more time-consuming, or we have to decrease the batch size by the same factor to fit those trajectories into memory (we discuss this point more in 3.7.1). This becomes very inefficient already for $K = 5$ with $K! = 120$ and impossible for $K = 10$ and 50, which we use in our experiments. Therefore, for $K > 2$ we suggest to loosen the assumption "for all" $\sigma : \{1, .., K\} \stackrel{\text{B}}{\to} \{1, .., K\}$ and instead work only with a selective subset of permutations G. One can see that if we define symmetrization f_θ^S as in (3.16), except averaging only across $\sigma \in G$, then the same conclusions about f_θ^S being symmetric hold to be true as long as $\forall \sigma, \sigma_0 \in G$, the composition $\sigma \circ \sigma_0$ still belongs to G. In other words, G should be a subgroup in the group of all permutations $\{1, .., K\} \stackrel{\text{B}}{\to} \{1, .., K\}$. A natural choice of such G_K, which works for all K, is the group of cyclic shifts, i.e. $\sigma_s(i) = i + s \mod K$ for $s = 0, ..., K - 1$. Even though such a choice does not make the model "completely" symmetric in the sense of (3.15), it "serves our goal" of eliminating bias towards one of the arms: none of the arms will be completely dominating the others now. This is because if, say, original model f_θ favors arm 1, then large output probabilities for this arm 1 will be transferred to all other arms $i > 1$ after applying cyclical shift σ_{i-1} on the output f_θ^S.

However, even if we use G_K instead of G, there are still some drawbacks associated with such symmetrization. First, it is still quite computationally expensive: every forward pass takes K times longer, which becomes a significant issue for larger values of K. Second and more importantly, this architecture "distances" us conceptually from whether the original transformer can represent the optimal solution. With symmetrization, we "engineer" our prior knowledge about the optimal solution and change its architecture accordingly. Therefore, it remains unclear whether the original transformer is just incapable[2] of representing the optimal,

[2]Remark: in theory, transformer model can approximate well any function if its architecture is big enough; here

symmetric solution, or it is capable of doing that, but it faces very non-convex optimization problem and gets stuck in a local minimum. Our second method addresses these concerns via **symmetry regularization**. An additional benefit of the regularization is the ability to control its strength (via the calibration of the penalizing λ constant) and its flexibility of what kind of beliefs can be reflected in our penalties (e.g., it is unclear how symmetrization can help when the optimal solution requires symmetry along only a subset of coordinates or even asymmetry).

3.6.2 Symmetry regularization

Instead of changing the model's forward pass, we can penalize the original model for asymmetric behavior during training. As we find in our experiments, symmetry regularization helps stabilize the training and push the model to achieve optimality in most experiments. At the same time, since regularization does NOT change the original architecture, we conclude that the original transformer can indeed represent the optimal solution.

In short, we penalize deviations between $f_\theta(H_t)$ and $\sigma^{-1}(f_\theta(\sigma(H_t)))$ for histories H_t encountered during the training, where we use the same group $\sigma \in G_K$ of cyclic shifts. This penalty on deviations is added as a new component to the combined loss in (3.13), so the model learns to be symmetric as part of the training. Below, you can find the formal definition of this symmetry regularization procedure. The reader should feel free to skip to the discussion

"incapable" refers to the abilities of a small-size transformer (details provided further)

of its computational benefits if the intuition of the algorithm is straightforward.

Algorithm 1: Symmetry Regularization

1 Perform the regular training procedure: roll out trajectories and calculate the policy, value, and entropy losses as previously. This step is unmodified and produces B final trajectories $\tau_i = a_0, r_0, a_1, r_1, ..., a_T, r_T$ (i = 1 ... B);

2 Choose a random subset τ_j of size [B / K] among τ_i ; `// K is the number of arms`

3 For each chosen τ_j, consider K augmented trajectories by applying all permutations from G_K:

$$\tau_{j,s} := \sigma_s(\tau_j) = ((a_0 + s) \mod K, r_0, (a_1 + s) \mod K, r_1, ..., (a_T + s) \mod K, r_T)$$

where $s = 0, ..., K - 1$ ("s" for shift). There will be $[B/K] \cdot K \leq B$ trajectories overall ; `// all the actions are shifted by some number` s `modulo K and rewards are kept the same`

4 Stack these trajectories together in a single batch of size B. Perform **one forward pass** and obtain the policy output P: the tensor with the shape (B, T, K) ; `// the third dimension is` K `because it corresponds to` K `logits of the policy output`

5 For simplicity, we assume the first dimension can be indexed via pair (j, s) so that $P[(j, s), t] = f_\theta(\sigma_s(\tau_j))[t]$ – the result of the original model on permuted $\tau_{j,s} = \sigma_s(\tau_j)$ at time step t. Recall that each of $P[(j, s), t]$ is still a K-dimensional vector, not a scalar.

6 For each index j and time step t, calculate the symmetry loss $L_{j,t}$ as the variance between the *permuted back* versions of the outputs:

$$L_{j,t} = \text{Ave}\left[\text{Var}_{s=0...K}\{\sigma_s^{-1}(P[(j, s), t])\}\right] = \text{Ave}\left[\text{Var}_{s=0...K}\{\sigma_s^{-1}(f_\theta(\sigma_s(\tau_j))[t])\}\right],$$

where the variance inside is taken coordinate-wise and then Averaged out across those K coordinates to obtain the final scalar value.

7 Finally, This loss is summed up across time horizons and averaged over the batch, where discounting happens for the same reason as with value and entropy losses:

$$L_{symmetry} := \lambda_{symmetry} \cdot \frac{1}{B} \sum_{j=1}^{B} \sum_{t=1}^{T} \gamma^t L_{j,t}. \tag{3.17}$$

8 Return this loss and add it to the other components of the total loss before the gradient update step. $\lambda_{symmetry}$ is a hyperparameter controlling the strength of regularization.

At first glance, this procedure seems similar to the symmetrization discussed in the previous

section. Also, for the symmetrized transformer, this additional loss $L_{symmetry}$ would be precisely zero: as one can see at step (6) of the algorithm, vectors $\sigma_s^{-1}(f_\theta(\sigma_s(\tau_j))[t])$ would be all the same, and therefore coordinate-wise variance would be simply a vector of K zeros.

An additional significant benefit of the regularization as opposed to mathematical symmetrization (3.16) is its computational efficiency: we perform **only one** additional forward pass. To understand why this matters, the reader should keep in mind that the time to do one training step[3] grows roughly **linearly** with the number of forward passes. We discuss why this is the case in more detail in 3.7.1, but let us assume this is true. Then, comparing all three procedures, the original algorithm performs T forward passes (step by step processing $a_0, r_0, ...a_t, r_t$ for each of $t = 1...T$). The symmetry regularization Algorithm 1 adds on top of that just one more forward pass at step (4), so the computational expense factors just by $\frac{T+1}{T}$, which is very small for $T = 100$ (our main experiment).

At the same time, the symmetrized transformer 3.16 needs to perform K times the larger batch size at each step $t = 0...T-1$ because it does symmetrization for any forward pass (not just the last one). This can be done by reducing the batch size K times to B/K (to fit into the memory all other $K-1$ permuted trajectories), or by repeating the same process K times. In other words, to make one learning step from batch size B of independent trajectories ("independent" means not the ones created by us via permutation), it needs K more forward passes, resulting in K times longer training! That becomes very expensive for $K \geq 10$.

To sum up, symmetrization and regularization aim to help alleviate the asymmetry bias. The primary motivation here is understanding how much that would improve the original transformer. However, one conceptual difference is that the regularization does not change the original model; it just changes the optimization procedure (from the regular SGD to the regularized one). Therefore, it can help us understand whether the original transformer could represent the optimal solution in the first place. If it did, it could signal that the asymmetry bias should have arrived from the gradient-flow optimization.

[3]here, "the training step" refers to rolling out trajectories until length T, calculating gradients, performing backward pass, and finally updating the parameters

3.7 Training procedure

Putting all parts together, the pseudocode for the overall algorithm is presented below.[4]

Algorithm 2: Final algorithm

Initialize θ_π and θ_V (all shared except the last layer)

for *step in range(total steps)* **do**

  Draw B bandits M_i from the distribution $\mathbf{D}$. Generate actions a_0 by random (tensor of $B \times 1$), get rewards $r_0[i]$ by pulling $a_0[i]$ in bandit M_i for each i.

  for t = *1...T* **do**

    Perform model's forward pass on $H_t = (a_0, r_0, ..., a_{t-1}, r_{t-1})$; `// this input tensor has shape` $B \times t \times 2$ `(because we group each action-reward pair together)`

    Use model's output logits to sample actions a_t ; `// tensor` $B \times 1$

    Obtain rewards r_t by pulling arms $a_t[i]$ for each bandit M_i.

  Using the last history $H_T = (a_0, r_0, ..., a_T, r_T)$, calculate Monte Carlo estimates for the policy, values, and entropy losses following formulas in (3.14) and symmetry loss in (3.17) in case of regulation.

  Perform backward pass on the combined sum

$$\widehat{L_P}(\theta, \lambda_{GAE}[t]) + c_V \cdot \widehat{L_V}(\theta) + c_{entropy}[\text{step}] \cdot \widehat{L}_{entropy}(\theta) + \lambda_{symmetry} \cdot \widehat{L}_{symmetry}(\theta).$$

  Update the parameters by performing one gradient step with learning rate = lr[step].

Note that the algorithm works the same way for the original and symmetrized transformer (because for the latter, we simply change the forward pass), and for the regularized, we only add one more component to the loss function. There are many hyperparameters in this algorithm: calibration of $c_V, \lambda_{GAE}, c_{entropy}, \lambda_{symmetry}$ coefficients in the loss calculation, learning rate scheduling and number of training steps. Let us shortly discuss each of them:

– c_V is the coefficient of the value function and is typically set between 0 and 0.5; in our case, we chose $c_V = 0.1$, being constant throughout the training.

– $\lambda_{symmetry}$ controls the symmetry regularization strength; we chose the values empirically depending on the convergence dynamics; details are provided in the experiments chapter 4.

– $c_{entropy}$ follows the dynamic annealing scheduling: it starts with 0.2. It decreases to 0 as an arithmetic progression for the first half of the training and stays at 0 for the remaining half. We remind the reader that higher $c_{entropy}$ penalizes the model for deterministic outputs $\pi_\theta(H_t)$, implying that the model is artificially encouraged for exploration. As the training progresses, we give the model the freedom to become more deterministic with a premise that, at a later

[4]The code for the algorithm, along with all other components of the work, can be found on GitHub

stage, it has lower chances of being stuck in sub-optimal deterministic policies. This is a common technique in Deep RL (for example, as used in [32]).

– λ_{GAE} follows the dynamically rising scheduling: it starts with 0.3. It increases to 1 as an arithmetic progression for the first half of training and stays at 1 for the remaining half. We remind the reader that $\lambda_{GAE} = 1$ corresponds to the unbiased estimates for the gradient, and $\lambda_{GAE} < 1$ implies biased estimates with a lower variance, which decreases the influence of future rewards. The idea is that when the model is only starting to learn at the beginning of the training, we prioritize the variance reduction of the gradient estimates as they are very noisy at that stage. As the training progresses, we decrease the bias, allowing the model to propagate future rewards more and more (as $\lambda_{GAE} \to 1$). The starting value of 0.3 was taken from [18] and [6], where they were used during the entire length of TRPO training. We find, however, that when keeping λ_{GAE} fixed during all the procedures, the network fails to converge, motivating the dynamic scheduling described above.

– The Learning rate in all experiments was tuned. It followed a warm-up and then a constant schedule with Adam Optimizer. In several experiments, we observed that the learning rate can affect the convergence outcome (i.e., converge / not converge to the optimum). Therefore, we did some basic learning rate exploration. We usually chose the lowest learning rate and the largest number of training steps when reporting the final findings, as those produced consistent results. In several settings, we also tried cosine learning rate decay (regime when the learning rate is adjusted following a cosine function decay, the explanation can be found here), but it did not change our results.

3.7.1 Computational considerations

Besides hyperparameter tuning, it is crucial to understand the computational constraints of the problem. For instance, the Monte Carlo estimate for the gradient direction (3.14), in theory, can be perfectly precise with a very large batch size B (by the law of large numbers), but in reality, we have limitations on how large the batch can be due to the memory constraints.

First, it is critical to note that one forward pass takes practically the same amount of time at any step t, no matter if it is only the beginning of the episode with just one action-reward (a_0, r_0) or the entire length T. This is because the transformer parallelizes computations across all t-s (which can be done without look-ahead due to the casual nature of GPT2), so the time to calculate outputs for all steps t is close to the time spent only on one t.

Naturally, during training, we choose the largest batch size B possible that fits into the memory of a single GPU. This depends on the experiment's context length $|C| = T$, as the most memory-heavy calculation happens at the last step $t = T$. This is because, during the forward pass, the transformer calculates attention scores between all input-input pairs (hence, the memory required for that grows as $O(|C|^2)$). The exact numbers are provided in the results

section. We also experimented with batch sizes larger than the memory capacity of a single GPU to stabilize the training. This can be done by repeating the algorithm 2 steps from line 3 to 9 multiple times N, i.e., accumulating the gradient without updating parameters yet. At the same time, we decrease the learning rate by also a factor of N. This is mathematically equivalent to increasing the batch size N times at the cost of N times longer training.

We hope that now the discussion of computational benefits of symmetry regularization vs. symmetrization in the previous section 1 becomes clearer. To recap, regularization just does one more forward pass at the end (we sample $\frac{1}{K}$-th among original trajectories for augmentation so that the batch size B for this additional forward pass remains the same and still fits into the memory). In contrast, symmetrization needs to perform K (numb er of arms) times more forward passes every time step, which becomes either time or memory expensive. Eventually, we decided to keep symmetrization limited: for every experiment with K arms, we generate only $\frac{B}{K}$ original trajectories for the symmetrized transformer, which then results in the total of B trajectories inside the forward pass after making K shifted cycles.

Chapter 4

Experiments

This section of the paper tries to evaluate the performance of the transformer and its symmetric modifications that were introduced in 3.6. Our goals include 1) understanding how well the transformer can approximate the Gittins index and how it compares to other meta-learners; 2) how much the symmetry adjustments improve its performance; and 3) to which extent approximation generalizes beyond the training distribution.

Before we go into the details of comparison, there are several things that remained unchanged for all the experiments. First, the architecture parameters. We used the GPT2 configuration with $N_{dim} = 96, N_{head} = 3, N_{layer} = 6$, meaning that each input was represented by $96/3 = 32$-dimensional vector, the transformer had 3 attention heads and had 6 layers of depth. This totals to $\approx 700k$ parameters (relatively small model). This choice was made because of the computational constraints, but more importantly, in order to have a relatively fair comparison with other SOTA meta-reinforcement learners, specifically the SNAIL model from [18]. There is no direct comparison between SNAIL and GPT because the former has only 4 separate self-attention blocks (whereas GPT has self-attention at every layer). Still, we keep the width the same (SNAIL also parameterizes every input with a 32-dim vector), and we keep the depth relatively small. As discussed in the literature review, the major limitation of SNAIL is that its depth has to scale logarithmically with the input length, so for $T = 100$, it had $2 \cdot \lceil \log_2(100) \rceil = 14$ convolution layers, but our depth is only 6.

Second, while the training settings differ across experiments, we hold them fixed for each experiment when comparing different models, e.g., original transformer vs. regularized vs. symmetrized. This implies the same scheduling for all hyperparameters in the training algorithm 2, the same learning rate, batch size, and the number of training steps. This gives us the ability to directly compare the models, factoring out all other components that can contribute to the training stability. The only difference is in the batch size of the symmetrized transformer as discussed in 3.7.1: its "independent" number of samples is K times smaller than for the original/regularized transformers in each experiment.

4.1 Comparison with other models

In this section, we focus on the evaluation based on the final expected reward per episode (scalar value) and training convergence dynamics. To remind the reader, the meta-learner's objective is to maximize the expected discounted reward across all possible bandits with uniformly distributed arm means.

We take the results of [18] as a benchmark as their "SNAIL" meta-learner has achieved state-of-the-art performance in most of the settings for bandits. Following their experiments, we tested all combinations of $T = 10, 100$ and $K = 5, 10, 50$. Their original work also had $T = 500$, but we did not include this length in the analysis because of the time constraints. As discussed in the training section 3.7.1, we use the largest batch size that fits into the memory: for $T = 10$, $B \approx 20000$ was more than enough (we could you up to $200k$ though), but for $T = 100$ we used $B = 5000$.

Table 4.1.1 presents the results. For comparison, we also report the results of one task-agnostic meta-learner, namely MAML from [8], and Thompson Sampling (TS), which should be treated as a baseline.

	Methods						
Experiment (T, K)	Gittins (optimal as $T \to \infty$)	TS	MAML	SNAIL	Transformer	Transformer regularized	Transformer symmetrized
10, 5	6.6	5.7	6.5 ± 0.1	**6.6 ± 0.1**	6.3 ± 0.1	**6.6 ± 0.1**	**6.6 ± 0.1**
10, 10	6.6	5.5	6.6 ± 0.1	**6.7 ± 0.1**	6.4 ± 0.1	**6.7 ± 0.1**	**6.7 ± 0.0**
10, 50	6.5	5.2	6.6 ± 0.1	**6.7 ± 0.1**	6.3 ± 0.0	6.4 ± 0.1	**6.7 ± 0.1**
100, 5	78.3	74.7	67.1 ± 1.1	**79.1 ± 1.0**	77.8 ± 0.2	**78.1 ± 0.1**	**78.4 ± 0.3**
100, 10	82.8	76.7	70.1 ± 0.6	**83.5 ± 0.8**	80.7 ± 0.1	**83.1 ± 0.1**	**83.5 ± 0.1**
100, 50	85.2	64.5	70.3 ± 0.4	**85.1 ± 0.6**	81.2 ± 0.1	81.0 ± 0.1	**84.5 ± 0.5**

Table 4.1.1: Total undiscounted reward on multi-arm bandits. Estimates are given with 95% confidence intervals. For each experiment, we highlight all models whose performance is not statistically significantly different from the best model (based on a one-sided t-test with p = 0.05)

NB: this table contains the expected total **undiscounted** reward after an episode of T interactions with the bandit. For instance, for the random policy we see $\mathbb{E}_{\mu \sim \text{Unif}[0,1]}(\mu) \cdot T = 0.5 \cdot 100 = 50$. It would be more reasonable to study total **discounted** reward instead because that is the training objective of the model, but since [18] as well as other papers in the domain (e.g., [6]) reported only undiscounted rewards results, that is the only way to directly compare the transformer with those models. It is also important to note that the Gittins index is not optimal for the settings in this table as $\gamma^{100} = 0.99^{100} \approx 0.37$ (hence,

for the Gittins index, rewards after horizon 100 still influence the decision making). For this reason, meta-learners can outperform the Gittins for smaller T by choosing to explore less. The next section 4.2 presents a comparison with Gittins index in terms of total **discounted** rewards with the lower discount rate $\gamma \leq 0.95$, where Gittins becomes very close to the exact optimum.

There are two major observations we would like to make from the table 4.1.1. First, even without symmetry adjustments, the original transformer achieves the reward close to the best possible, at times significantly outperforming both Thompson Sampling and MAML. This signals about its generally decent capability for meta-RL in bandit tasks. Second, in all experiments except for $K = 50$ (which we discuss further), there is a clear progression pattern in performance: Original Transformer < Regularized ≤ Symmetrized, where the latter two achieve the exact optimum in almost all and all settings accordingly. This second argument is supported by the convergence dynamics of all three models, which we present below for horizon $T = 100$ and different arms $K = 5, 10$. We remind the reader that the loss function used in training (3.14) is just a surrogate for the gradient derivation, but "the true loss" function is the discounted regret. To remind the reader, the discounted regret is $\sum_{t=0}^{T} \gamma^t (\mu_{t^*} - \mu_{a_t})$[1], where $\mu_{t^*} = \max_k\{\mu_k\}$ is the highest available of the arm means (note: the discounted regret also equals minus expected discounted reward up to an additive constant). Therefore, we output this undiscounted regret on the y-axis of the plots below because that is the true "measure of convergence."

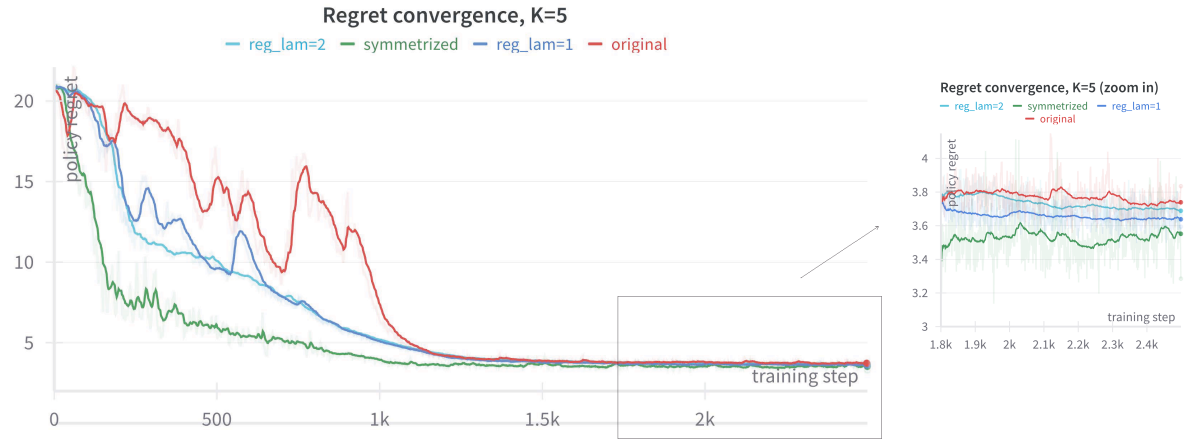

Figure 4.1.1: Training convergence for 5 arms, comparing 4 models: original transformer (blue), symmetrized (green), and two regularized ones (red and violet) with different penalties. The right figure zooms in on the later stage of training (highlighted with a black rectangle) for closer comparison

[1]Also note that we can either include or not the very first term $\gamma^0(\mu_{t^*} - \mu_{a_0})$ in all calculations, because if we generate first actions by random, then this value is the same on expectation, so it does not change comparison between the models. In our analysis, this 0-th term is included for all models and strategies

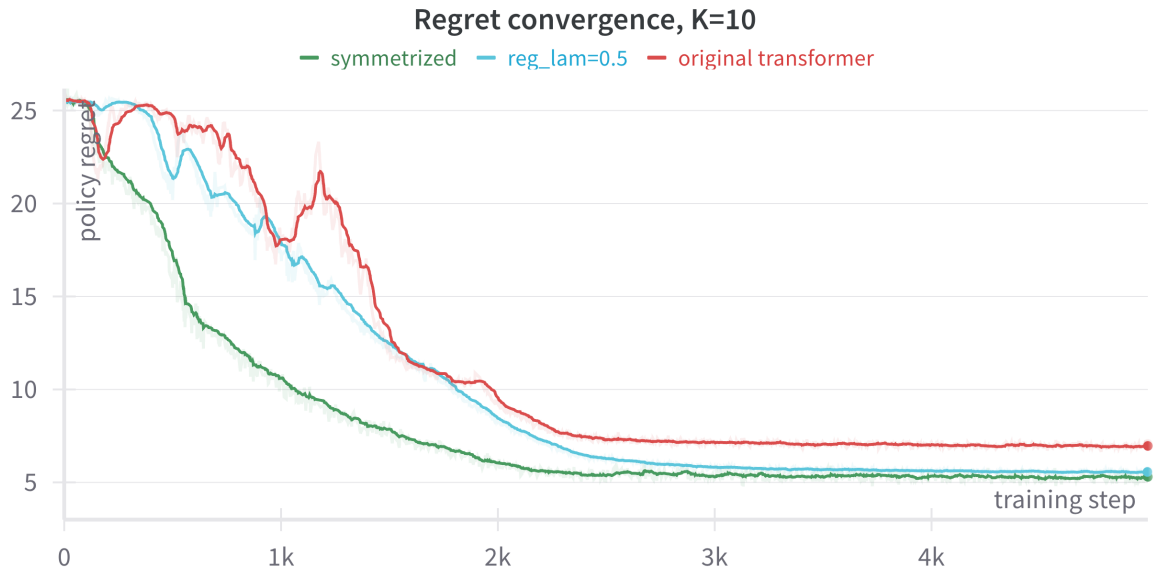

Figure 4.1.2: Training convergence for 10 arms

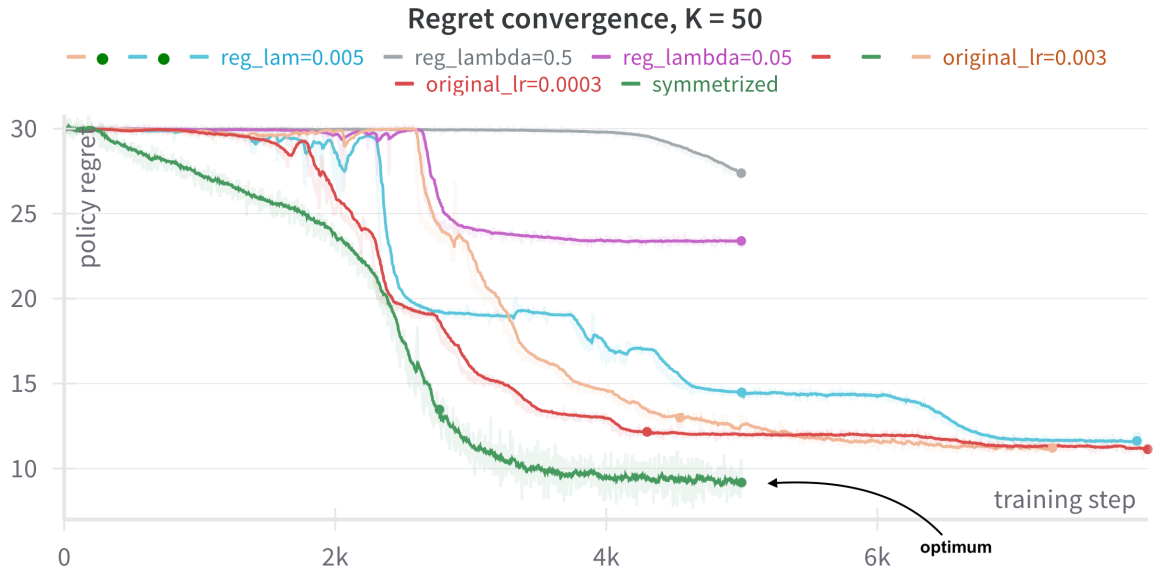

Figure 4.1.3: Training convergence for 50 arms. This problem was particularly hard for both the original and regularized transformers. Different curves show learning rate exploration and $\lambda_{symmetry}$ exploration. Symmetrized transformer still achieves the global optimum. (Note: the dots in the middle of some curves represent the runs that were interrupted and continued later)

It is clear that the symmetrized transformer has the fastest rate of convergence and also tends to be the most stable throughout training. We should say it is quite surprising that it has that faster convergence even though we effectively give it K times fewer "independent samples" to learn from (recall, for the symmetrized transformer, we generate only $\frac{B}{K}$

independent trajectories, and the shift each K times). For instance, when $K = 5$, a significant part of the learning (in terms of regret going down) happens just during the warmup (up to step 200), while the original transformer gets to that stage only after 1000+ steps.

In contrast, the original transformer has sharp spikes where the loss (regret) goes up, sometimes returning close to the original value of a random policy. As expected, we also find that these spikes usually correlate with asymmetric behavior: regret starts to go up when the model becomes biased towards some arms and discriminates against others. Regularization stabilizes the training well (some blue plots are smooth) for K = 5 and 10. However, spikes in regret remain for some of the experiments (observe K=5, dark blue curve). At times, we also saw large sudden spikes back to the utterly random policy just in a few training steps (excluded from the plots). We do not fully understand this phenomenon nor the proper rule for choosing $\lambda_{symmetry}$ yet, so further investigation is needed. One hypothesis of why these spikes could happen is that MSE symmetry loss (3.17) on the output logits can be pretty unstable for the extreme value of logits (i.e., when some of the probabilities are very small), causing significant changes to the parameter vector θ. This can be either fixed by using TRPO that would limit the magnitude of change $\Delta\theta$ in a single step or by rethinking the loss function in the first place.

In addition, Regularization failed to work for $K = 50$. In general, this experiment turned out to be the hardest for both the original and regularized transformers, as both the table of rewards and the convergence plots indicate. Here, the regularized transformer converged very close to the original one, but the convergence was way longer and more unstable (light blue curve on the figure 4.1). There are some interesting patterns that we do not understand about its convergence, for instance, a significant drop in regret next to the 2500th training step: observe that the light blue curve becomes close to red (original) and green (symmetrized). Further investigation is needed of what kind of learning happens in these moments. Nonetheless, we emphasize that for the scope of this paper, the primary motivation for the regularization is not to "fix" the original training but rather to understand whether the original transformer, without modifications to its architecture (like the symmetrization does), can represent the solution very close to the global optimum. As we see, this is true for $K = 5, 10$. It might that this would be true for $K = 50$ as well in case a better, more stable regularization loss was used.

In addition, the symmetrized transformer still achieved state of the art performance (as we can judge from the cumulative rewards table), and had a stable convergence curve. Again, this is quite surprising because, in this experiment, it only learned from $\frac{B}{K} = \frac{5000}{50} = 100$ "independent" trajectories as opposed to 5000 that of the original transformer.

4.1.1 Permutation invariance

Besides the symmetry-based invariance, our problem also has the permutation-based invariance: the optimal decision should not depend on the order in which we pulled arms in the

past, nor the order of their successes or failures. This makes sense intuitively, and it is valid for the Gittins index strategy as it is based only on $2K$ numbers: the number of successes and failures in the past for each of the arms. These types of input-based invariances, i.e., symmetry and permutation, relate to a well-known field of Geometric Deep Learning [1]. This field frames different Deep Learning models as methods that respect the structure of the domain tasks. In particular, the attention-based mechanism [31] can be posed as a Graph Neural Network that learns the complete graph relationship between the inputs within the gradient-based optimization towards some downstream task. However, the permutation-based invariance faces a contradiction with the transformer's architecture, which is designed for sequential tasks. First, the inputs should take into account positions, which are achieved via concatenated positional encodings. Second, the transformer is causual in nature, and thus during the training inputs at time t_1 would not attend to the future inputs at times $t_2 > t_1$, but the attention happens from t_2 to t_1. In theory, we could train a completely permutation-invariant transformer by eliminating positional encodings and by breaking the causal assumption, but that would significantly slow down the training and also distance us from the initial goal of the study of a very general and universally used original transformer's architecture.

For these reasons, we do not observe a complete permutation invariance even for the symmetrized transformer. However, we observe that the model becomes more and more permutation invariant as the training progresses, even when the regret is already very close to the optimum (last 20-30% of the training for the symmetrized transformer). This is an intriguing phenomenon that requires more investigation.

4.2 Similarity with Gittins index

This section will show a more thorough analysis of how close the transformer's solution is to the Gittins index strategy. As promised, we switch towards evaluating solutions with discounted regrets instead of undiscounted rewards. Regret analysis is a more traditional way to evaluate RL algorithms for bandits, and the Gittins index is known to minimize the expected discounted regret in the infinite horizon case. Following the discussion in 3.2, we study only the setting where the Gittins index can be considered very close to the optimal strategy of the finite horizon bandits. In particular, we use $T = 100$ and $\gamma = 0.9$ and 0.95 because $0.95^{100} \approx 0.006$, and so all the rewards after horizon 100 can be neglected. We restrict analysis for the case of two arms ($K = 2$) to visualize the model's decisions (see heatmaps further).

4.2.1 In distribution analysis

Here, we perform a very similar experiment to the previous section: $\mu_1, \mu_2 \overset{\text{iid}}{\sim} \text{Uniform}[0, 1]$. We refer to this as "in distribution" analysis because even though the model is evaluated on novel bandits (i.e., pair of mus that potentially have never occurred *exactly* in training), these bandits are still within the training distribution. Therefore a very similar bandit could appear during the training. In fact, in the case of $K = 2$, this is true almost surely: if we draw μ_1, μ_2 independently and uniformly $20k$ times (the batch size used in this experiment), the probability that all of these pairs are more than $\epsilon/2$ away along both dimension from a fixed point in a unit square is no more than $(1 - \epsilon^2)^{20000}$. For $\epsilon = \frac{1}{50}$ this value is less than 0.1%, meaning that at every training step, almost indeed there was a bandit with arm means close up to $\frac{1}{100}$ to the one that happens in the evaluation. This is, of course, not true for higher K, potentially explaining why the convergence was harder to achieve.

Similarly to the previous section, we present the table with the final expected discounted regret for different methods and strategies and the convergence dynamics of all three transformers.

Methods				
Gittins (optimal)	Thompson	Transformer	Transformer regularized	Transformer symmetrized
0.404 ± 0.01	0.66 ± 0.01	0.44 ± 0.03	0.406 ± 0.03	0.405 ± 0.04

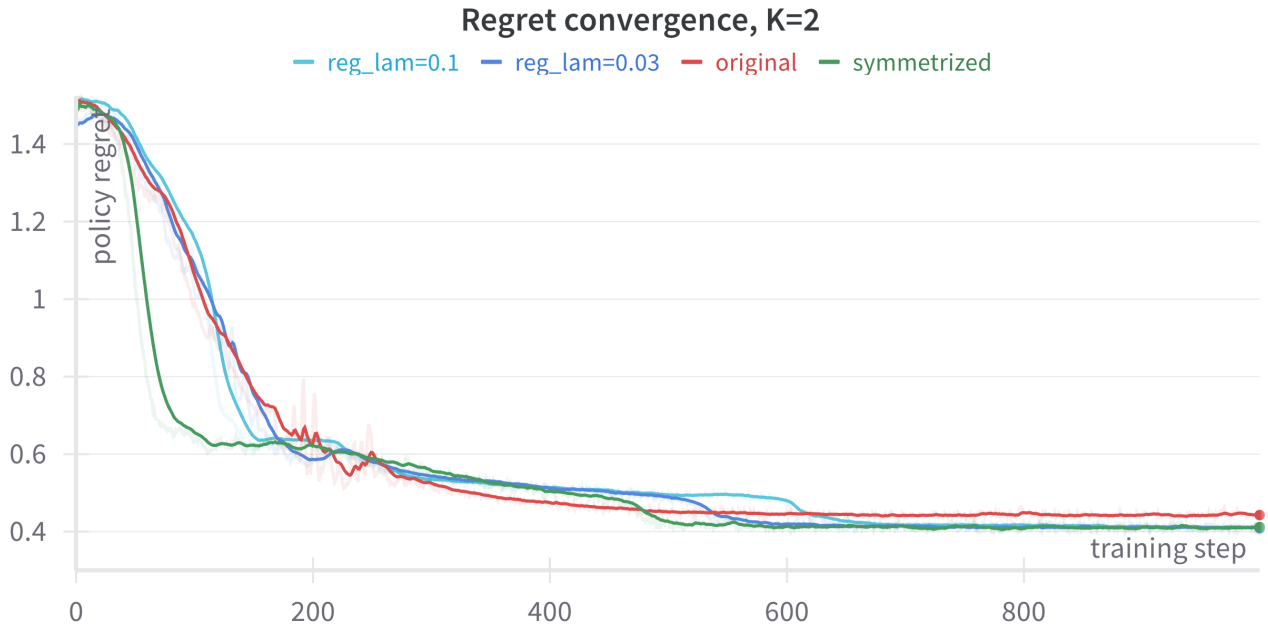

Figure 4.2.1: Training convergence for 4 different models: original transformer (red), symmetrized (green), and regularized with different lambda penalty (light and dark blue)

As we can see, both Regularized and Symmetrized transformers converge to the global minimum (their regrets are within 95% confidence intervals of the Gittins index). However, the Symmetrized one is still ahead in terms of convergence speed.

These results reconcile with what we have seen in the previous section: the original transformer fails to achieve the optimum, even for 2 arms. As we encourage it to be symmetric via regularization, it finds the optimum. The visualization below demonstrates this asymmetry phenomenon. In particular, we split unit square into 7 by 7 equally spaced grid (i.e., bounds are 0/7...1/7, ..., 6/7...7/7), and for each region $\{(x, y)|\frac{i}{7} \leq x < \frac{i+1}{7}, \frac{j}{7} \leq y < \frac{j+1}{7}\}$ we draw 250 instance of μ_1^k, μ_2^k (k = 1,..., 250) drawn i.i.d. from that region. Then, we roll out 250 trajectories τ_k using the Gittins index as a decision maker of which of the two arms to pull and interact with the corresponding bandit μ_1^k, μ_2^k. Simply speaking, this is our static dataset of "expert" Gittins index trajectories that are equally distributed across the unit square. To evaluate how close any particular model f is to the Gittins index, we ask this model to output decisions a_t for the same history $\tau_k[0:t]$ as the Gittins index has seen when rolling these trajectories τ_k. The final "difference score" for the single trajectory is $\sum_{t=0} T - 1\gamma^t \cdot \mathbb{1}(a_t \neq g_t)$, where a_t is the decision of model f and g_t is the decision of Gittins. We discount the difference by γ^t because the decisions at later stages are less important due to the nature of the discounted objective. We also neglect differences at steps t when both of the arms have equal Gittins scores, and thus, different decisions can be made simply because of randomness (for instance, when both arms have been pulled the same number of times, but none of them had a success outcome yet). This dataset of trajectories is fixed when evaluating different models, so it provides a direct fair comparison (the smaller the difference score, the more "similar" decisions are to the Gittins). These scores can be calculated in each region (averaging out differences across all 250 trajectories), after which we finally output all scores together as a heatmap over the unit square. These heat maps are presented below.

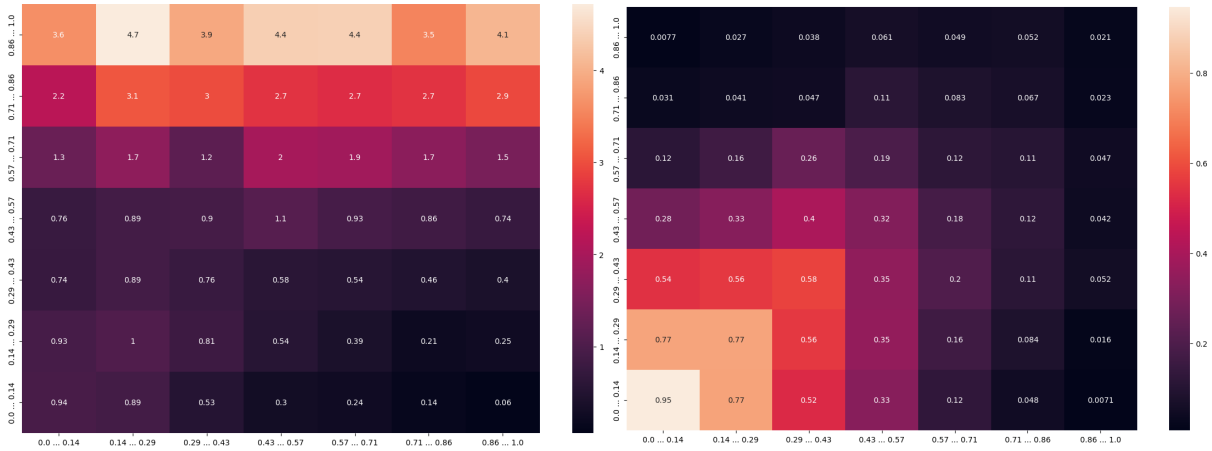

Figure 4.2.2: Heatmaps of difference scores with Gittins for the original (left) and symmetrized (right) transformers

Now asymmetry can be clearly observed: on the left heatmap, entrees above the diagonal $\mu_1 = \mu_2$ have way higher difference scores than those below the diagonal. Even the top left corner with $\mu_1 \leq \frac{1}{7}, \mu_2 \geq \frac{6}{7}$ has a score of 3.6, where seemingly the strategy should be very simple: if you pull arm 1, you observe (most probably) failure, so pull arm 2 instead; when arm 2 is pulled, you observe (most probably) success, so continue pulling it. In contrast, the symmetrized transformer produces a perfectly symmetric heatmap (on the right) with respect to the diagonal $\mu_1 = \mu_2$.

Another interesting pattern is the progression of difference scores from low μ-s to high μ-s (observe how the scores go from the bottom left corner to the top right). Seems like the model is almost identical to Gittins when arm means are high, but it struggles more with lower means. We believe that, to some extent, this is just the nature of the multi-armed bandit's problem: observing positive reward influences the model's actions "more deterministically" rather than observing negative reward. Namely, when both arm means are high (for instance, in the top right corner of the heatmap), Gittins pulls some arm, observes success, pulls it again, observes another success, etc. – continues pulling it until the end of the episode. It is reasonable to expect that the model will have the same behavior. However, the situation is different when both arm means are low (e.g., bottom left corner): in the beginning, we mostly see failures until the first couple of successes. One hypothesis could be that the model is not deterministic enough (recall, we do not force it to be so), i.e., maybe it has higher output chances for the correct arm, but it still gives some non-negligible chance for the incorrect one. When we change its decisions to be deterministic, the difference scores overall decrease, yet just slightly, and the diagonal pattern remains, so it does not explain the full picture.

Another way to measure this is to understand the difference scores not in relative but in absolute magnitude. Imagine we have a "broken" Gittins index that works precisely as the Gittins index, but with probability p, it returns incorrect action. Then, the expected difference score for such model is $\sum_{t=0}^{T-1} \gamma^t \cdot \mathbb{1}(a_t \neq g_t) = (\gamma \cdot 0 + \gamma \cdot p + \gamma^2 \cdot p + ...+) = \frac{\gamma}{1-\gamma} \cdot p = 9p$ (the probability to make mistake at step 0 is zero because we always feed in the same action-reward pair at step t to the Gittins and other models). This means that the score of 0.95 in the left bottom corner corresponds to the "broken" Gittins that make mistakes with probability $\frac{0.95}{9} > 10\%$, which intuitively seems quite large. This, therefore, suggests that the model is doing something different rather than *exactly* Gittins index, at least in the bottom left corner. To clarify, we do not expect the model to exactly "implement" the Gittins index; instead, we expect it to approximate it very well, which it does. At the same time, it is interesting to analyze how much it differs from the Gittins to understand the model's generalization abilities better. We further uncover this via out-of-distribution analysis, which is presented in the following section.

4.2.2 Out of distribution analysis

As discussed in the previous section, all experiments were held "in-distribution": *both* train and evaluate on bandits with Uniformly distributed means. In this final section, we conduct an experiment where the evaluation tasks include the ones outside of the range of tasks that the model has been trained on. Informally speaking, if the goal is to learn to implement the Gittins index, this is the proper evaluation of the unseen, out-of-distribution data (i.e., "data" refers to the range of tasks outside of training).

To do that, we train the model on the bandits with arm means drawn from the Beta(10, 10) distribution. The density for this distribution is presented below (figure 4.2.2). This is a symmetric distribution, which is very centered around the middle 0.5 point. In particular, the probability of being below $\frac{1}{7}$ or above $\frac{6}{7}$ is below 0.01%. Therefore, all regions on the boundary of the square in the previous heatmaps can be treated as OOD regions. The Gittins index, which changes its priors from Uniform to Beta, still remains optimal for the same goal of minimizing discounted regret when drawing bandits from this prior. Note: in this experiment, we also use $\gamma = 0.95$.[2]

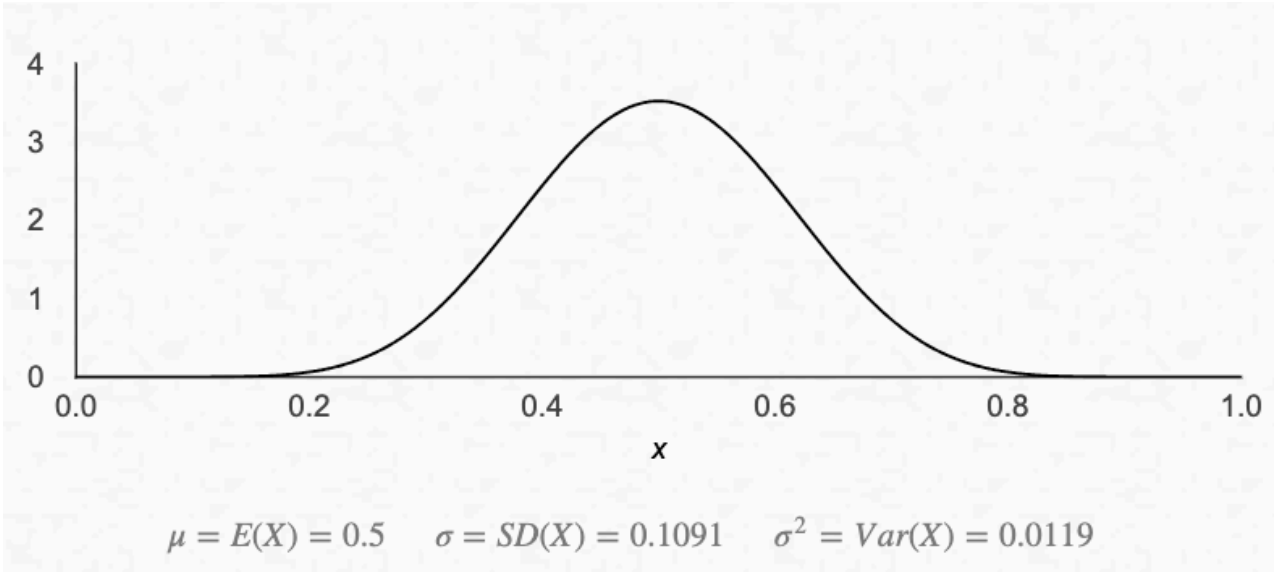

Figure 4.2.3: Beta(10, 10) density

Methods				
Gittins with Beta(10, 10) prior (optimal)	Thompson	Transformer	Transformer regularized	Transformer symmetrized
0.62 ± 0.10	0.84 ± 0.01	0.64 ± 0.01	0.638 ± 0.01	0.635 ± 0.01

[2]We apologize to the reader for the different choices of γ in Uniform and Beta experiments, which might be confusing. When experimenting with Uniform means, we tried to be very conservative (i.e., take smaller γ so that γ^{100} is extremely small), but then realized that the choice of $\gamma = 0.95$ suites this goal too.

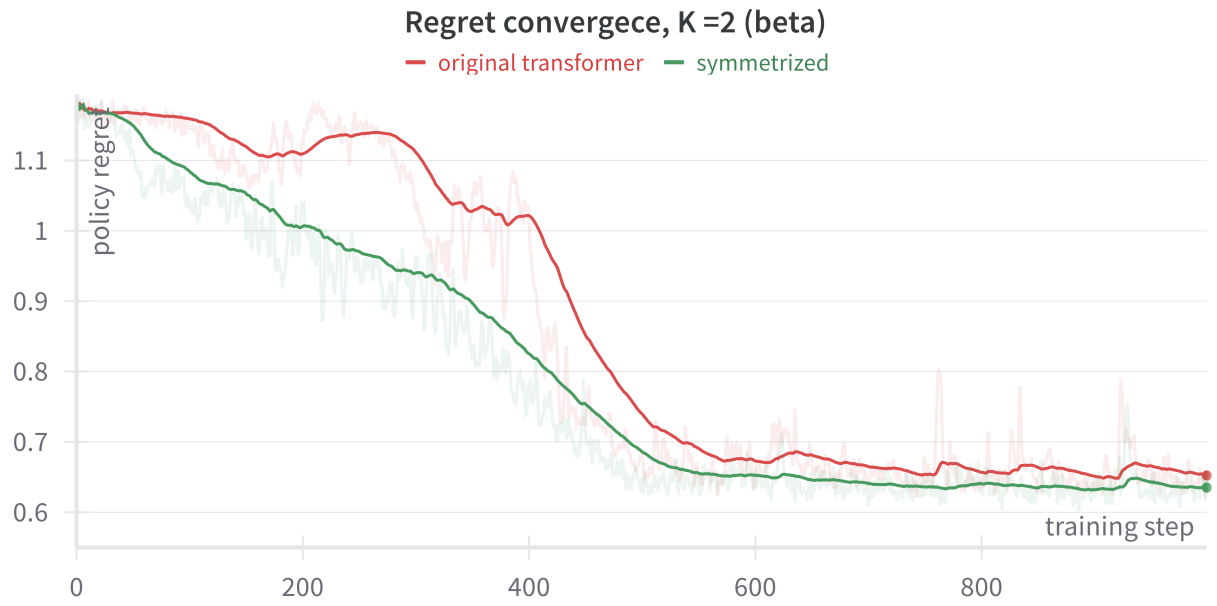

Figure 4.2.4: Training convergence with Beta(10, 10) priors: the original transformer (red) and symmetrized (green)

Similarly to the previous section, we train the original, regularized, and symmetrized transformers. Comparison table 4.2.2 and regret convergence plots 4.2.2 are presented above. Again, the transformer can approximate the Gittins index quite well. This time, we also noticed that the gap between the original and symmetrized transformers was much smaller. In fact, in several examples of initialization, we observed that the two models converged to practically the same final regret. This suggests that during regular training with Uniform priors, the original transformer is more prone to be greedy/biased towards one of the arms because of observing arms with high means. The lack of asymmetry is also demonstrated in the heat maps below (similar to the previous section). The asymmetry of the original transformer (on the left) is barely noticeable as the heatmap seems to be relatively symmetric with respect to the $\mu_1 = \mu_2$ diagonal (it is not even clear whether there is an asymmetry or simply imprecision due to the Monte Carlo of 250 trajectories per region).

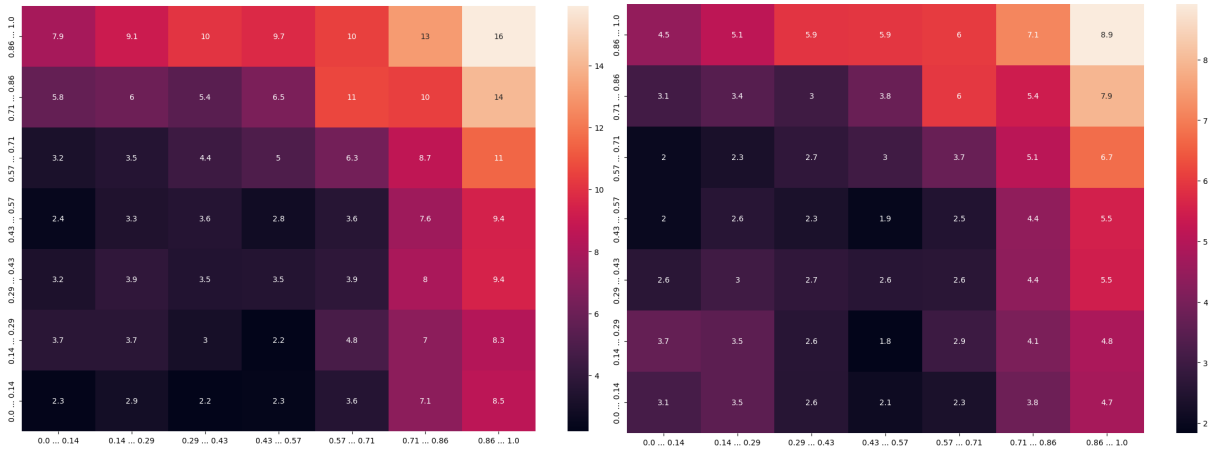

Figure 4.2.5: Heatmaps of difference scores with Gittins and Beta(10, 10) prior; the original (left) and symmetrized (right) transformers

Perhaps another surprising phenomenon is that the pattern on the diagonal "flipped". Now, the difference scores go up as the arm means become higher. In particular, the greatest difference happens in the top right region, which previously was the area of closest agreement between the model and Gittins. Notably, this does not imply that the model underperforms in this top-right region compared to Gittins, and seemingly the opposite happens.[3] Monte Carlo estimates show that Gittins' (10, 10) regret in this region is about 0.37, while the model's is 0.33. This motivates us to look at one more (final) heatmap: simply the difference between the model's discounted regret and Gittins' discounted regret across different regions (presented below).

[3]To clarify, it is possible for the model to outperform Gittins(10, 10) in that specific, because Gittins(10, 10) is *exactly* optimal when averaged for the whole unit square and under the assumption of arm means drawn from Beta(10,10), but heatmaps draw them uniformly

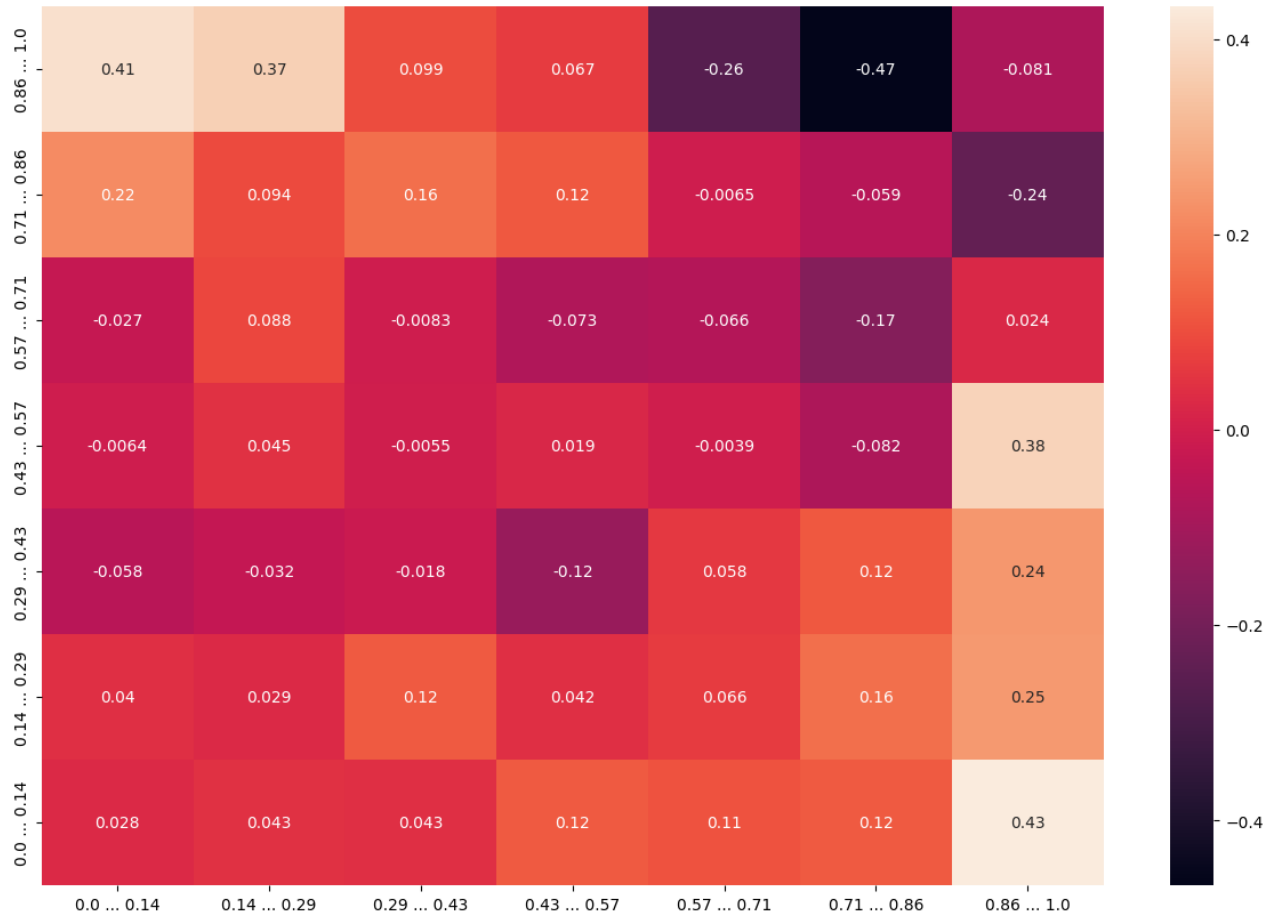

Figure 4.2.6: Heatmap of regret differences between symmetrized transformer and Gittins Beta(10,10)

Because this heatmap depicts [model regret minus Gitins regret], positive values represent the regions where the model does worse than Gittins, and negative is better. We remind the reader that the unit square's boundaries are out of sample, and here we observe an interesting pattern. The model does better when the arm means are high (top right region) but worse when one arm is high, and the other is low (bottom right or top left corners). One potential interpretation of this pattern is that the model is "less greedy" than the Gittins and explores more. In other words, it does not fully commit to the arm when it sees "1", explaining why it is doing worse in the top left/bottom right but also why regret is lower for the top right area (exploring other arms might be profitable for that region, because the chances of getting 1 are high, but you still gather more data to distinguish better which mean is higher). Intuitively, this also reconciles with asymmetry practically disappearing for the original transformer.

This trend with Beta(10, 10) prior holds after we perform the training several times, with different initializations. One hypothesis is that this phenomenon might relate to the PAC Bayes Learning Theory [11]. This theory frames the process of learning from the data not as seeking for a single "best" solution (e.g., transformer's weights that result in the optimal policy), but

rather to find a *distribution* over different solutions that performs well on the training data. Before training, we start with some prior over all possible functions that the model can represent (here, "prior" means the tuple < architecture, initializer of parameters >). Then, the training data serves as an instrument that reshapes our before the posterior of functions that fit the training data very well (hence the connection with the Bayesian theory). In this way, the training data from the Uniform experiment seemingly pushes the transformer to the posterior, which has way higher chances for greedy, asymmetric policies. In contrast, the more balanced Beta(10,10) training data does not. It would be interesting to see whether our results about the original transformer change if, during the training, we progressively shift the training data from Beta(10,10) towards Beta(1,1) (which is Uniform).

Above all, we recognize that in this experiment, as opposed to the Uniform arm means, our models are still a little bit off from the optimum (consider the table 4.2.2, Gittins has regret 0.62 ± 0.10, while symmetrized transformer 0.635 ± 0.01). We did not manage to push them entirely to the optimum due to time constraints and, therefore, understand that current observations might be premature. In addition, it will be interesting to investigate how this Out-of-distribution trend changes as we continue training even after the policy is very close to the Gittins(10,10) in terms of regret.

To summarize all results, we observed that the original transformer can approximate the Gittins quite well but also establishes the asymmetry bias, which is fully responsible for the gap from the optimum. This bias can be alleviated by symmetrization/regularization, but it is equally important to understand why the bias appears in the first place. Experiments suggest that shifting the training distribution towards the one that encourages more exploration, e.g., Beta(10,10), helps the original model to practically eliminate the asymmetry bias and achieve an optimum without adjustments. One implication is that more deterministic bandits in the training data (arm means close to 0 or 1) might have an unstable impact on the model's updates during the gradient-based optimization, which pushes the model to become greedy. In addition, we also observed that the model shows intriguing performance OOD, outperforming Gittins in certain regions. This might be considered a partially successful outcome for our meta-learning objective: the model generalizes well for OOD tasks but, at the same time, converges to something different rather than *exactly* the Gittins index.

Chapter 5

Discussion & Conclusion

In this work, we discussed the capabilities of the transformer architecture to perform Meta Reinforcement Learning. In particular, we studied how well the original model can approximate the optimal strategy known as the Gittins index in the problem of multi-armed bandits. We show that despite the previous results [18], transformers are able to learn efficient policies that are quite close to the optimum in terms of expected total reward per episode. However, we also discovered an inductive asymmetry bias that appears in all experiments when using the regular policy-based gradient descent for training and causes the model to result in a globally sub-optimal solution.

Transformer symmetrization as proposed in 3.6.1 has shown to cancel out the asymmetry bias and push the model towards the global optimum, i.e., Gittins index and other state-of-the-art meta-learners. In this way, multi-armed bandit problems (with symmetric priors over the arms) seem to favor models that respect invariances over the inputs, such as symmetry or permutation invariance, as a part of either their architecture or the optimization procedure.

Permutation-based invariance, as discussed in 4.1.1, is in apparent conflict with the Transformer being initially designed to address sequential tasks and encode different positions as part of the input. However, there is nothing in the transformer's architecture that hinders them from respecting symmetry over the input labels. One explanation for this phenomenon could come from the optimization perspective, i.e., the policy gradient descent appears to be stuck in the local minimum of asymmetric solutions along the way. As evidence supporting this direction, in this work, we have shown that symmetry regularization (Algorithm 1) can push the transformer towards the optimal solution (at times, achieving it exactly), without any modification to the model's initial architecture. Inter alia, the proposed regularization algorithm can be also used in other RL tasks to help transformers respect the input-based invariances (when we have prior belief about such invariances) with little to no additional computational expenses. In addition, the fact that the original model struggles even in the

setting of the short horizon (T = 10) as indicated by table 4.1.1 also suggests that the challenge is coming from the training instability because for this smaller problem, the medium-sized transformer, intuitively, should have the ability to express the optimal solution precisely.

At the same time, past works suggested that the transformer's inferior performance in RL tasks and difficult optimization can be addressed by minor architecture changes. For instance, [22] have proposed layer normalization and a new gating mechanism, which result in stabilization of training and significantly better final performance on various RL tasks, including control and navigation environments, though not bandits or other symmetry-related tasks.

Nonetheless, further investigation is needed to understand better the interplay between the original transformer's inductive biases and gradient-based optimization in RL domains. Such biases (e.g., found in our work asymmetry) have further implications in the domains where the model's decision aims to serve human preferences, such as in RLHF [14]. The issue is not that the transformer fails to achieve exact optimality, but rather the emergence of a strange biased behavior in the first place where the data does not possess such bias. In other words, it is acceptable that LLMs are imperfect, but it becomes much more dangerous if they possess unexplained discrimination against certain input tokens, especially if these tokens have an empathic nature and target humans, as in language-related RLHF.

There are a number of promising directions for the future research. First, multi-armed bandits is a relatively simple RL task, and it is a natural question whether similar results would hold for other RL domains that require symmetric meta-solutions, especially with continuous control problems. Second, it is interesting how much this phenomenon develops with the model and data scale, which we did not have a chance to explore fully due to time constraints. In particular, we would like to know how much deeper and wider the network should be to find the global optimum without any modifications and whether the scale could help to generalize beyond the training distribution of tasks (i.e., avoid the "beta failure" found in 4.2.2). From the data perspective, increased batch size could also help the model to get out of asymmetric local minimum. Ultimately, the idea of developing an architecture or optimization technique that would allow the model to automatically identify input-based invariances and adjust the policy accordingly could be a profound advancement for the future of transformer at the intersection of Reinforcement Learning and Geometric Deep Learning.

References

[1] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[2] 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. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[4] Thomas Degris, Patrick M Pilarski, and Richard S Sutton. Model-free reinforcement learning with continuous action in practice. In *2012 American control conference (ACC)*, pages 2177–2182. IEEE, 2012.

[5] 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.

[6] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[7] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[8] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[9] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.

[10] John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 41(2):148–164, 1979.

[11] Benjamin Guedj. A primer on pac-bayesian learning. *arXiv preprint arXiv:1901.05353*, 2019.

[12] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34:1273–1286, 2021.

[13] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.

[14] Nathan Lambert, Louis Castricato, Leandro von Werra, and Alex Havrilla. Illustrating reinforcement learning from human feedback (rlhf). *Hugging Face Blog*, 2022. https://huggingface.co/blog/rlhf.

[15] Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, et al. In-context reinforcement learning with algorithm distillation. *arXiv preprint arXiv:2210.14215*, 2022.

[16] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018.

[17] Luckeciano C Melo. Transformers are meta-reinforcement learners. In *International Conference on Machine Learning*, pages 15340–15359. PMLR, 2022.

[18] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017.

[19] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2015.

[21] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[22] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[23] Jing Peng and Ronald J Williams. Incremental multi-step q-learning. In *Machine Learning Proceedings 1994*, pages 226–232. Elsevier, 1994.

[24] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[26] Juergen Schmidhuber, Jieyu Zhao, and Marco Wiering. Simple principles of metalearning, 1996.

[27] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[28] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[29] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[30] Aaron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu, et al. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 12, 2016.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. pages 6000–6010, 2017.

[32] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[33] Christopher JCH Watkins. Learning from delayed rewards. *University of Cambridge*, 1989.

[34] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[35] Jeremy Wyatt. Exploration and inference in learning from reinforcement. 1998.

[36] Ruiqi Zhang, Spencer Frei, and Peter L Bartlett. Trained transformers learn linear models in-context. *arXiv preprint arXiv:2306.09927*, 2023.