Imperial College London

Department of Computing

MEng Joint Maths and Computing

Interim Report

Author:
Aaraf Hossain
ah2620

CID:
018517167

Imperial College Supervisor:
Dr. Roberto Bondesan

June 19, 2024

Abstract

This work applies the concept of machine learning to improve the efficiency of MCMC methods. Markov chain Monte Carlo (MCMC) methods sample from probability distributions, crucial when direct sampling is impractical. The Metropolis-Hastings (MH) algorithm is fundamental in this field, it proposes new samples based on current ones, which are then accepted probabilistically. The efficiency and effectiveness of this algorithm is dependent on the proposal distribution used. Finding suitable proposal distributions, is manual and time-consuming, particularly in cluster algorithms. While the standard Ising model can be sampled from efficiently using either the Wolff or Swendsen-Wang algorithms, this is not the case for the Ising model with plaquette interactions.

In this paper I provide a generalisation of the Wolff algorithm incorporating MH to sample from the Ising model without and with plaquette interactions. The framework of this algorithm was first initialised by Roberto Bondesan and Nabil Iqbal from which I built upon. I then applied RL in order to learn a good proposal distribution to be used in the algorithm.

In the end I was able to replicate the original Wolff algorithm almost exactly when sampling from the Ising model and had decent success when adding plaquette interactions. The Effective sample size measures how many samples in a set are independent and is able to measure the quality of a sampler. High ESS values were achieved when K, the plaquette coupling term, was greater than zero and moderate values were achieved when $K < 0$ indicating decent success.

Contents

Chapter 1

Acknowledgments

This project would not have been possible without the initial work and continuous support of several individuals. I would like to extend my heartfelt appreciation to my supervisor, Roberto Bondesan (hereafter referred to as R.B) and Nabil Iqbal (hereafter referred to as N.I), who laid the groundwork for this research through their preliminary investigations. Their work on applying the concept of reinforcement learning to Monte carlo cluster algorithms has been a crucial stepping stone for my project.

Their expertise, guidance, and encouragement have been invaluable throughout this process. I am grateful for their willingness to share their knowledge and for providing me with the opportunity to build upon their initial research. Their support has been essential in navigating the challenges of this project and in achieving the objectives set forth.

Chapter 2

Introduction

Markov chain Monte Carlo (MCMC) methods comprise a class of algorithms used to sample from arbitrary probability distributions. They are particularly useful when it is difficult or impractical to directly sample from a distribution. By constructing a Markov chain that has the desired distribution at equilibrium, one can obtain a sample from said distribution by recording states from the chain. One of the fundamental MCMC algorithms is the Metropolis-Hastings (MH) algorithm [5]. It works by proposing a new state based on the current state, and then accepting or rejecting the proposed state probabilistically .

While the MH (and MCMC methods in general) are powerful and versatile, they have their problems which can impact their practicality. Since we only care about the samples obtained at equilibrium, a large number of iterations may be needed to achieve convergence. This can make them computationally intensive, especially for complex models or high-dimensional parameter spaces. In addition to this, at times it can be hard to distinguish whether the chain has reached equilibrium or just appears to have (explained in [1] and 3.1). Samples obtained from MCMC chains are often autocorrelated, meaning that consecutive samples are not independent and the effective sample size may be smaller than the amount of samples obtained [1]. However the aim of my project is to tackle a different problem with MCMC methods while keeping the aforementioned problems in mind.

MCMC methods often rely on something called a proposal distribution which is sampled from in place of the target distribution. These samples are accepted probabilistically based on if it is effectively coming from the target distribution itself [5]. Finding a suitable proposal distribution requires manual work that can be difficult and time consuming. This is even harder in the setting of cluster algorithms such as sampling from the Ising model with plaquette interactions where no good proposal distribution is known.

My project will frame this as a reinforcement learning problem, in hopes that a good proposal can be learned for the Ising model with and without plaquette interactions. By characterising the sampler as an agent within the environment described by the Ising model variant, it aims to learn how to maximise the effective sample size of the sets it produces. The concept of learning more efficient monte carlo updates using the effective sample size has been done before using Policy Guided Monte Carlo[6]. However this approach requires large high quality datasets to sample from complex models. My work will take this concept a step further by relying on the sampler alone to learn without the use of such datasets.

Chapter 3

Background

3.1 Autocorrelation time

Since successive states in markov chains used in MCMC methods are correlated, it may take upto 5000 iterations (5000 samples) or more just to obtain 2 truly independent samples. The degree of similarity between a chain and a time lagged version of itself is referred to as autocorrelation, [1] sheds more light on this topic in the context of markov chains and how to estimate these values.

The paper makes the distinguishment between the two types of autocorrelation time in Monte Carlo simulation, initialisation bias and autocorrelation in equilibrium. Initialisation bias refers to the initial distribution the chain finds itself given the start configuration. If this distribution is not the stationary distribution π (almost always the case) then there is an "initial transient" present in which the data does not reflect the desired equilibrium distribution. This means that the chain requires a certain amout of iterations for this transient to be removed, this is commonly referred to as the "burn in period" since these data points end up being thrown away. Throughout the paper, the upper bound for this autocorrelation time is referred to as τ_{exp}, if we know the upper bound then we have a sure fire way of knowing when the chain has converged. However in practice, τ_{exp} is almost never known for certain and can only be estimated theoretically and empirically and even if it was known it may be overly conservative, this can be the case when the state space of the chain is infinite making it possible that $\tau_{exp} = \infty$. While there is no sure fire way of finding τ_{exp}, it can be empirically estimated by measuring the autocorrelation function $C_{ff}(t)$ for a suitably large set of observables f (more on the autocorrelation function later). It is made clear that empirically estimating τ_{exp} has the potential danger of metastability.

Metastability is when the chain being used appears to have converged but instead has just settled down to a long-lived metastable region of configuration space that may be far from equilibrium. While a proof or concrete upper bound on τ_{exp} is evidence that metastability has been avoided, a heuristic arguement that τ_{exp} is not too large is more feasible. If metastability is an issue, it is useful to know what each of the metastable regions look like, so initial conditions in each of these regions can be trialed. Consistency between these runs does not guarantee metastability has been avoided but gives increased confidence.

Once equilibrium has been achieved the samples from the burn in period are discarded. Since this data contains the initial transient, throwing it away means we lose nothing and avoid a systematic error. Now the only concern is the autocorrelation in equilibrium. Throughout this paper, the integrated autocorrelation time is referred to as τ_{int} or $\tau_{int,f}$. Here f is a real valued function defined on the state space of the chain S. It is shown that the sample mean, $\bar{f}$, has a variance a factor of $2\tau_{int,f}$ higher than if independent sampling was used. In other words, a run length of n samples after reaching equilibrium effectively has $n/2\tau_{int,f}$ independent data points (Effective Sample Size). It is then stated that in the context of Monte Carlo methods, the *computational efficiency* of an algorithm is synonymous with $\tau_{int,f}$ when the time measured is units of cpu time. There maybe at times some complications between the number of iterations and the complexity per iteration, but in principle the smaller $\tau_{int,f}$ is the better.

The paper also gives us a way of estimating τ_{int} based on a finite but large sample. First of all let $f = \{f(x)\}_{x \in S}$ be a real valued function on the state space S. Then $\{f_t\} \equiv \{f(X_t)\}$ is a stationary stochastic process with mean

$$\mu \equiv \langle f_t \rangle = \sum_x \pi_x f(x) \tag{3.1}$$

where π is the equilibrium distribution of the markov chain X_t. μ is estimated by the sample mean (which is unbiased)$\bar{f}$

$$\bar{f} = \frac{1}{n} \sum_{i=1}^{n} f_i \tag{3.2}$$

The unnormalised autocorrelation function for a time lag t of this process is given by

$$C(t) \equiv \langle f_s f_{s+t} \rangle - \mu^2 \tag{3.3}$$

the normalised autocorrelation function

$$\rho(t) \equiv C(t)/C(0) \tag{3.4}$$

and the integrated autocorrelation function

$$\tau_{int} = \frac{1}{2} \sum_{t=-\infty}^{\infty} \rho(t) \tag{3.5}$$

the estimator of eq (3.3) is given by

$$\hat{C}(t) = \frac{1}{n-|t|} \sum_{i=1}^{n-|t|} (f_i - \bar{f})(f_{i+|t|} - \bar{f}) \tag{3.6}$$

if μ is unknown, if it is known then $\bar{f}$ can simply be replaced with it above. It follows that the "natural estimator" of $\rho(t)$ is given by substuting the above equation into eq (3.4). The natural estimator for τ_{int} is shown to not be a viable one since the variance does not go to zero as the sample size n tends to infinity. To circumvent this issue, a cutoff window of size M is used.

$$\lambda(t) = \begin{cases} 1 & \text{if } |t| \leq M \\ 0 & \text{if } |t| > M \end{cases} \tag{3.7}$$

$$\hat{\tau}_{int} = \frac{1}{2} \sum_{t=-(n-1)}^{n-1} \hat{\rho}(t)\lambda(t) \tag{3.8}$$

This gives us a better estimator for the autocorrelation time since the variance goes to zero as the sample size increases,

$$var(\hat{\tau}_{int}) \approx \frac{2(2M+1)}{n}\tau_{int}^2 \tag{3.9}$$

however it does introduce a bias

$$bias(\hat{\tau}_{int}) = -\frac{1}{2}\sum_{|t|>M} \rho(t) + o\left(\frac{1}{n}\right) \tag{3.10}$$

The choice of M is a tradeoff between bias and variance. A large M may give us a low variance in our estimate of the autocorrelation time but too large will result in a large bias and vice versa. I suspect this choice of M will possibly be a hyperparameter when constructing a reward function that utilises the autocorrelation time since better and worse estimates will allow us to explore different realisations of our end proposal. Reading this paper has made it clear to me that a good reward function should aim to minimise both $\tau_{exp} and \tau_{int}$ (τ_{int} being of more importance) while keeping metastability in mind and taking measures into avoiding it.

3.2 Effective Sample Size

The effective sample size (ESS) measures the amount by which autocorrelation within a chain increases uncertainty in estimates as exaplained by [2]. While I have briefly mentioned ESS in the previous section, this reference manual reaffirms some of the claims in [1] and provides an alternative way to estimate autocorrelation and ESS. Due to the correlation between samples in a chain, given N samples, the number of independent samples is the effective sample size, N_{eff}. [2] mentions that N_{eff} is related to the autocorrelation time in the same way as [1]

$$N_{eff} = \frac{N}{\sum_{t=-\infty}^{\infty} \rho(t)} \tag{3.11}$$

The denominator here is synonymous with $2\tau_{int}$ in eq (3.5). The manual explains that the autocorrelations and the ESS by extension cannot be calculated due to the intractability of integrating the probaility function associated with the chain (see section 16.4.2 in [2]). As a result, the autocorrelations are estimated by a different method from [1].

M different chains are used with $\hat{\rho}_m(t)$ being the autocorrelation at lag t for chain $m \in \{1, .., M\}$, this is estimated using an FFT package(see [2] for more details). The combined autocorrelation is given by

$$\hat{\rho}(t) = 1 - \frac{\bar{\sigma}^2 - \frac{1}{M}\sum_{m=1}^{M} \sigma_m^2 \hat{\rho}_m(t)}{\hat{var}^+} \tag{3.12}$$

where σ_m^2 refers to the within chain variance estimator, $\hat{var}^+$ being the multi chain variance estimate and $\bar{\sigma}^2 = \frac{1}{M}\sum_{m=1}^{M} \sigma_m^2$ which is then used to estimate the ESS

$$\hat{N}_{eff} = \frac{M \cdot N}{1 + \sum_{t=0}^{2k+1} \hat{\rho}(t)} = \frac{M \cdot N}{1 + \sum_{t=0}^{k} \hat{P}(t)} \tag{3.13}$$

where $\hat{P}(t) = \hat{\rho}(2t) + \hat{\rho}(2t+1)$. Here k is the largest integer such that $\hat{P}(t') > 0$ for all $t' = 1, ..., k$.

Since one of the aims of my project is to find a practical way to use ESS and autocorrelation in a reward function, I am in a position to apply either of the two methods (or find others later) and put them into practice. An immediate downside I see with this method is that it relies on working with multiple different chains. While it may provide us in higher confidence in our estimates, it may be computationally inefficient or too intensive based on the sample size or the amount of chains used. What also remains unclear at this time is the use of the FFT package for the estimates of $\hat{\rho}_m(t)$. If this ends up being infeasible or not practical I can rely on estimating these autocorrelation using the method outlined in section 2.1.

3.3 Neural Simulated Annealing

Having a proposal distribution be learned via the means of a reinforcement learning algorithm is something that has been done before. An example of this is seen in the paper talking about Neural Simulated Annealing [3].

Simmulated annealing (SA) is a stochastic global optimisation metaheuristic that is applicable to a multitude of discrete and continous variable problems. This paper focuses on applying it to combinatorial optimisation (CO) problems. A CO is defined by a triplet, the set of problem instances, the set of feasible solutions and an energy function denoted by the symbols (Ψ, X, E) respectively. Simulated annealing constructs an inhomogeneous markov chain $x_0 \to x_1 \to x_2 \to ...$ for $x_k \in X$ which, in the context of a CO, converges asymptotically to a minimiser of the energy function. The transition from the current state, x_k, to the next state, x_{k+1}, depends on a proposal distribution, $\pi : X \to \mathbb{P}(X)$ and a temperature schedule. The proposal distribution is sampled from based on the current state (known as the Metropolis-Hastings step), perturbing the current solution in hopes of finding a lower energy solution. This new state, x', is either accepted with probability p setting $x_{k+1} = x'$ or is rejected and $x_{k+1} = x_k$ (see Algorithm 1 in [3] for more details.). The temperature schedule $T_1, T_2, ...$ is a parameter which is responsible for the balance of exploration and exploitation. This ensures that the perturbations smoothly approach an optimum of the energy function. It starts high and goes low with the limit $T_k \to 0$. Provided that the chain is long enough, the chain will visit the minimisers of E almost surely under certain regularity conditions. However, the convergence speed is ultimately determined by π and the temperature schedule, both of which are hard to tune.

The paper poses simulated annealing as a Markov decision process (MDP) which would allow for reinforcement learning to work in tandem with SA. This creates the opportunity to optimise the proposal distribution using RL while preserving the convergence of vanilla SA, this is dubbed as "Neural SA". Due to the nature of SA, it naturally fits into the MDP framework, an MDP $M = (S, A, R, P, \gamma)$ consists of state space S, a set of possible actions A, a reward function R, a transition kernel $P : S \times A \to \mathbb{P}(S)$ and a discount factor $\gamma \in [0, 1]$. A stochastic policy $\pi : S \to \mathbb{P}(A)$ is added on top of this, which, along with the transition kernel defines a length-K trajectory $\tau = (s_0, a_0, s_1, a_1, ..., s_K)$. The trajecotry can also be seen as a sample from the distribution $P(\tau|\pi) = \rho_0(s_0) \prod_{k=0}^{K-1} P(s_{k+1}|s_k, a_k)\pi(a_k|s_k)$, where where $s_0 \sim \rho_0$ is sampled from the start-state distribution ρ_0. An MDP has been solved if the a policy has been found that maximises the expected return $E_{\tau \sim P(\tau|\pi)}[R(\tau)]$ where $R(\tau) = \sum_{k=0}^{K-1} \gamma^{K-k} R(s_k, a_k, s_{k+1})$ is the discounted return.

SA is formalised as an MDP, with states $s = (x, \psi, T) \in S$ for ψ a parametric description of the problem instance and T the instantaneous temperature. An action $a \in A$ perturbs $(x, \psi, T) \to (x', \psi, T)$, where $x' \in N(x)$ is a solution in the neighborhood of x. The Metropolis-Hastings (MH) step in simulated annealing is perceived as a stochastic transition kernel, regulated by the system's temperature. The transition probabilities adhere to a Gibbs distribution, and the dynamics are expressed as

$$x_{k+1} = \begin{cases} x', & \text{with probability } p \\ x_k, & \text{with probability } 1 - p \end{cases}$$

where

$$p = min(1, e^{-\frac{1}{T_k}(E(x';\psi)-E(x_k;\psi))}).$$

This defines a transition kernel $P(s_{k+1}|s_k, a)$, where $s_{k+1} = (x_{k+1}, \psi, T)$. Proximal Policy Optimization (PPO) was used to learn the policy π, with rewards being either the immediate gain $r_k = E(x_k;\psi) - E(x_{k+1};\psi)$ or the primal reward $r_k = \min_{x \in x_{1:k+1}} E(x;\psi)$. The transition kernel incorporates the information in the current state and the energy function before making its decision to accept a given perturbation or not, effectively quantifying the likeliness of finding a lower energy solution.

The Policy network architecture was deliberately kept simple. A state s_k is mapped to a set of N feature vectors which were problem dependent and the paper notes that there should be a natural way to do this for a given CO problem (this was the case for all of their experiments). They are then fed into an MLP independently, embedded into a logit space and then mapped to probabilities by a softmax activation. A visualisation of this can be found in figure 2 of [3]. The architecture is adaptable to CO problems of any size as the complexity scales linearly with N. By keeping the architecture lightweight, any issues regarding the length of a SA chain is avoided since they typically require a large amount of iterations before converging. On the topic of convergence to an optimum, SA requires the markov chain associated with the proposal distribution to be irreducible. Irreducibility being that for any temperature, any two states are reachable through a sequence of transitions with positive conditional probability under π. The network policy fulfills this criterion, provided that the softmax layer does not assign zero probability to any state—a condition typically satisfied in practice. Consequently, Neural SA inherits convergence guarantees from the standard simulated annealing approach.

Neural SA was experimented on a variety of problems in section 4 of [3], including the TSP problem and Bin Packing problem. The results of which were compared to more complicated RL frameworks, Operations Research tools and other algorithms used to solve these problems and was competitive with them. Even with little to not fine-tuning of the models hyperparameters, vanilla SA was far outperformed and was within a few percentage points of the global minima. Neural SA only is shown to only require training data of problem instances to train without requiring solutions, this along with its scalability to any problem size and lightweight architecture the authors believe that Neural SA is a promising solver that can strike the right balance among solution quality, computing costs and development time.

While Neural SA is set in the context of CO problems, I can take some of the learnings and findings from this work and apply it to my project. For instance, I can see that using RL to have the proposal distribution be a learned policy is at least feasible and not out of the realm of possibility. What is important to consider is how the paper related SA to a Markov decision process since it is a key aspect in applying RL to an MCMC method. Since the aim of a CO problem is to minimise an energy function, this was incorporated in the dynamics of the MDP and reward of the agent, both areas where my work will differ. Due to the computationally intensive nature of MCMC methods, a lightweight architecture may also be something I need to replicate in the context of my work. On the flip side, the paper does not talk about tuning the temperature schedule. This can either be left as a hyperparameter or be something I look to optimise in some way shape or form.

3.4 Cluster Sampling Methods

In this section I will introduce the Ising/Potts model and a method to sample from it. The model is defined on a set of nodes organised on a lattice each with a label $l \in 1, ..., L$. Each node has 4 nearest neighbour connections with edges connecting them if they have the same label. An illustration of this is in the figure 1(b) below.

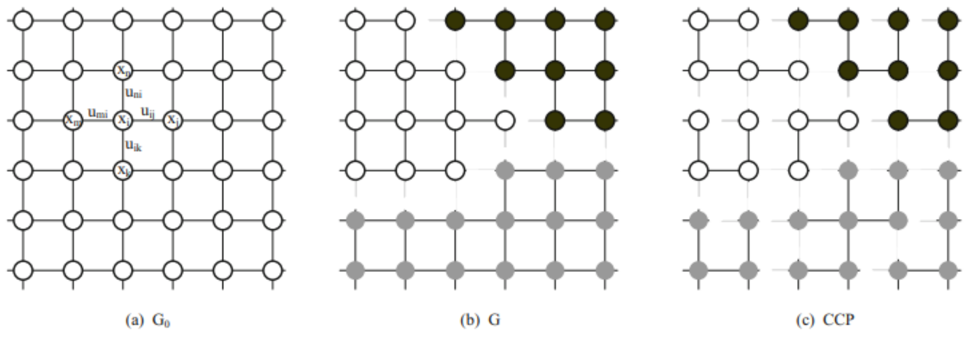

Figure 3.1: taken from [4]

Formally put, let $\mathbf{G} =< V, E >$ be an adjacency graph with 4 nearest neighbour connections with each vertex $v_i \in V$ having a corresponding state variable $x_i \in \{1, ..., L\}$(or colour/label). The total number of labels is predefined, if $L = 2$ then we have the Ising model, otherwise we have the Potts model. Let $\mathbf{X} = \{x_1, ..., x_L\}$ denote the labelling of the graph, then the Ising/Potts model is a markov random field,

$$\pi(\mathbf{X}) = \pi_{IS/PTS}(\mathbf{X}) = \frac{1}{Z} \exp\{- \sum_{<s,t> \in E} \beta_{st} \mathbf{1}(x_s \neq x_t)\} \tag{3.14}$$

where Z is a normalisation constant and $B_{st} > 0$. Such a model is typically used to study phase transitions in statistical mechanics and computational physics. The Swendsen-Wang(SW) algorithm, detailed in [4], is an MCMC algorithm that can efficiently simulate this model and its phase transitions. The algorithm introduces a set of auxiliary variables on the edges,

$$\mathbf{U} = \{\mu_e : \mu_e \in \{0,1\}, \forall e \in E\}$$

An edge is disconnected if and only if $\mu_e = 0$, these binary variables follow a Bernoulli distribution conditional on the vertices they correspond to,

$$\mu_e|(x_S, x_t) \sim Bernoulli(q_e \mathbf{1}(x_s = x_t) \tag{3.15}$$

$$q_e = 1 - e^{-\beta_{st}} \tag{3.16}$$

With this we have $\mu_e = 1$ with probability q_e if the nodes have the same label and $\mu_e = 0$ automatically if they do not share the same label. The SW algorithm is split into 2 steps which I will outline below.

1. **The clustering step**: Given the current labelling $\mathbf{X}$ sample resample the variables in the set $\mathbf{U}$ according to the distribution outlined in Eqs.(3.15) and (3.16). Note that the "sampling"" for the edges between nodes with different labels can be automatically set to 0 i.e for the edge $e =< s, t >$ its auxiliary variable $\mu_e = 0$ if $x_s \neq x_t$. This splits the edges into sets,

   $$E(\mathbf{X}) = E_{on}(\mathbf{X}) \cup E_{off}(\mathbf{X}) \tag{3.17}$$

   The remaining edges are then turned off with probability $1 - q_{st}$, further dividing $E_{on}(\mathbf{X})$

   $$E(\mathbf{X}) = E_{on}(\mathbf{U},\mathbf{X}) \cup E_{off}(\mathbf{U},\mathbf{X}) \tag{3.18}$$

   where $E_{on}(U, X)$ form a number of connected components which is denoted by

   $$CP(\mathbf{U},\mathbf{X}) = \{cp_i : i = 1, ..., K, with \cup_{i=1}^{K} cp_i = V\} \tag{3.19}$$

   Vertices that are in the same connected component cp_i are guaranteed to have the same label

2. **The flipping step**: Select one connected component $V_0 \in CP$ at random and assign the same label $l \in 1, 2, ..., L$ to all nodes in V_0. The label l follows a discrete uniform distribution

   $$x_s = l \,\forall s \in V_0, \;\; l \sim uniform\{1, 2, ..., L\} \tag{3.20}$$

   This step can be applied to more than just one or all of the connected components

For homogeneous models ($\beta_{ij} = \beta$), small values of β appear random whereas large values of β (close to 1) almost has the same label for all nodes. There is a value β_0 around 0.8 or 0.9 where a transition between the "random" and "uniform" labelling, with its inverse β_0^{-1} is known as the critical temperature. The time for convergence is known as the mixing time and when the SW markov chain has converged, exact sampling is reached. I will touch more on this below.

let the SW markov chain have kernel K and initial state X_0, after t steps the state follows probability $p_t = \delta(X - X_0)K^t$ where $\delta(X - X_0)$ is given by

$$\delta(X - X_0) = \begin{cases} 1 & \text{if } X = X_0 \\ 0 & otherwise \end{cases} \tag{3.21}$$

A way to measure the convergence of the chain is via the total variation

$$\|p_t - \pi\|_{tv} = \frac{1}{2}\sum_{\mathbf{X}} |p_t(X) - \pi(X)| \tag{3.22}$$

which is formally used to define the mixing time

$$\tau_{mix} = \max_{\mathbf{X}_0} \min\{t : \|p_t - \pi\|_{tv} \leq \epsilon\} \tag{3.23}$$

where τ_{mix} is a function of ϵ and the graph complexity $M = |G|$ in terms of the number of vertices and edges. When a chain is said to "mix rapidly", it means that it has a relatively small τ_{mix}. Formally, if $\tau_{mix}(M, \epsilon)$ is logarithmic or polynomial then the chain mixes rapidly. The paper states that analytical results have surfaced that shows that the SW chain mixes rapidly even on sparsely connected graphs

3.5 Ising model with Plaquette Interactions

For ease and simplicity I will refer to the Boltzmann weight of the Ising model rather than 3.14

$$\pi(\mathbf{S}) = \exp\left(BJ\sum_{l\in L}\prod_{i\in l}\sigma_i\right) \tag{3.24}$$

The 2 equations are analagous for $J > 0$ and constant β. L corresponds to the edges in the model and σ_i denotes the "spin" (label) at node i such that $\sigma_i \in \{-1, 1\}$.

The Ising model with plaquette interactions is an extension of the standard Ising model. This model adds terms that allow for interaction between corner sites on each "plaquette"" or square visible made by the lattice structure of the model, A visualisation of a plaquette can be seen in figure 3.2. Plaquette interactions allows for the consideration of more complex patterns of spin interactions beyond nearest neighbour interactions. The Boltzmann weight for this model is given by

$$\pi(\mathbf{S}) = \exp\left(BJ\sum_{l\in L}\prod_{i\in l}s_i + K\sum_{p\in P}\prod_{i\in p}s_i\right) \tag{3.25}$$

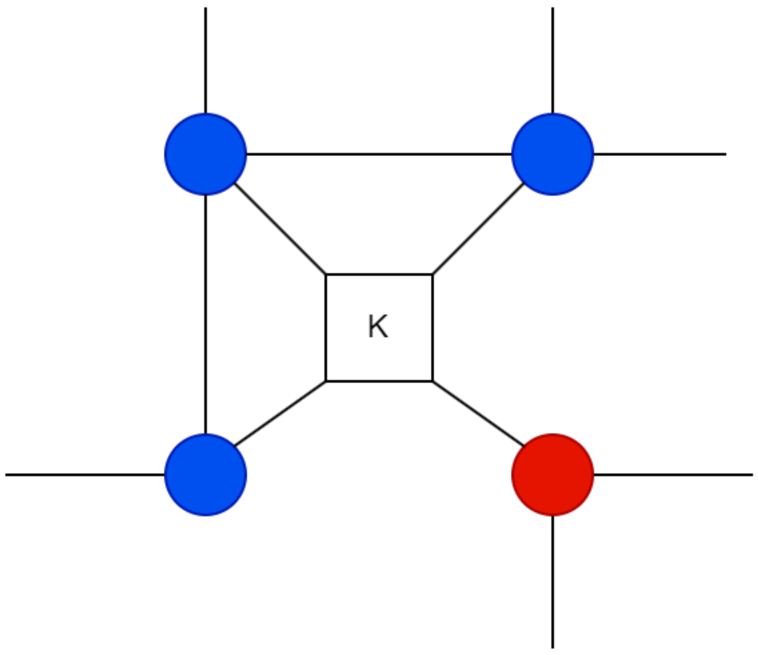

Figure 3.2: A visualisation of plaquette interaction where the different colours represent opposing spins and the lines representing connections or interaction terms. Plaquette interactions allow for corner sites of a plaquette to interact with one another through the plaquette

Due to the complex nature of the model, it is hard to sample from and as a result, there are no exact sampling methods like there is for the standard Ising model (SW and Wolf algorithms). Boltzmann machines were used in [7] and had varying success in trying to sample for this model. The major drawback established in the paper was that the acceptance rates decreases for large systems due to a mismatch between their surrogate model and the original physical model.

3.6 Wolff Algorithm

The Wolff algorithm is another sampling method for the Ising model. This sampling method was developed shortly after the SW algorithm and was essentially a single cluster variant of it. Starting from the Boltzmann weight in 3.24, It contains the following steps:

1. Choose a site x randomly as the first point of a cluster to be flipped and prepare an empty stack.
2. Go through the nearest neighbors y of x and add them to the cluster with probability

$$p(\sigma_x, \sigma_y) = 1 - \exp(\min(0, -2\beta J \sigma_x \sigma_y))\,. \tag{3.26}$$

3. For every site y that is added to the cluster also add it to the stack.
4. take an item off the stack and iteratively repeat steps 2 and 3 until the stack is empty
5. For every site in the cluster, flip its spin (1 to -1 and vice versa)

3.7 Policy Guided Monte Carlo

The next paper I will cover [6], introduces a framework called Policy Guided Monte Carlo (PGMC) that uses reinforcement learning to improve MCMC sampling. It addresses that the autocorrelation of the samples lead to long and computationally expensive simulations to achieve reliable results and that many machine learning ideas and methods have risen in order to combat this. One such method is explained in the paper and is called the *Effective model Monte Carlo* (EMMC).

Given a probability density (referred to a model in this paper) $w : S \to \mathbb{R}_{\geq 0}$ for a state space S, EMMC splits the task sampling from w into 2 tasks:

1. **A learning/training stage**. After a dataset following the distribution of w is obtained, an *effective* model $\tilde{w}_\theta : S \to \mathbb{R}_{\geq 0}, \tilde{S} \subset S$ is designed by a set of parameters $\theta = \{\theta_i\}$. This effective model is then fitted to the data set via some form of machine learned regression so that $\tilde{w}_\theta(s) \simeq w(s)$ for states of statistical significance. By construction, $\tilde{w}$ should be computationally advantageous over w, be it that it is cheaper to evaluate or allow for a better sampling method.
2. **$\tilde{w}_\theta$ is used as a proposal generator**. An MCMC sampling scheme should be applied onto $\tilde{w}_\theta$ to perform a number of updates $s \to s_1 \to s_2 \to ... \to s_n = s'$. The state s' is then proposed as a sample for the original model and is accepted with probability
$$\alpha(s \to s') = \min\left[1, \frac{\tilde{w}_\theta(s)w(s')}{\tilde{w}_\theta(s')w(s)})\right]$$
which ensures that the detail balance condition holds, given by
$$w(s)\pi(s \to s')\alpha(s \to s') = w(s')\pi(s' \to s)\alpha(s' \to s) \tag{3.27}$$
where $\pi(s \to s')$ is the probability of proposing state s' when at the state s. When this condition holds we know that the chain will asymptotically sample from the distribution of w.

EMMC has been shown to be a vast improvement shown by references [7,8,10-15] in [6] but has a few limitations. First of all the effective model has to be flexible and sophisticated enough to imitate the original model otherwise the result may actually be worse than a traditional MCMC method. Additionally, sufficiently good training data is required for the learning stage, with more complicated models requiring a

larger dataset. A lack of quality in the training data will be present in the proposal generator i.e. relevant parts of the state space will not be present or represented enough in the samples produced.
Much like in [3], the work mentions that a lot of the notions in reinforcement learning lines up well with MCMC and approaches it in a similar fashion:

- Actions - the action space at a given state A_s is defined to be all the possible updates $a := s \rightarrow s'$ with the inverse of this action being defined as $a^{-1} := s' \rightarrow s$
- Policy - the policy for taking an action $a \in A_s$ is quantified by the proposal probablity $\pi : A_s \times S \rightarrow [0, 1]$

The acceptance rate used for the MH step is given by

$$\alpha(s \rightarrow s') = \min\left[1, \frac{(w(s')\pi(s' \rightarrow s)}{w(s)\pi(s \rightarrow s')})\right] \tag{3.28}$$

This way the markov chain dynamics will be determined solely by the choice of π. If this was in the setting of the original Metropolis algorithm then the choice of π would be so that the detail balance equation (eq (3.27)) would hold making the equation in eq (3.28) simple. The paper mentions that this is not necessarily optimal and from my interpretation this would only limit the space finding an optimal policy to fit the original model. To find the optimal policy, the paper introduces something called a *performance factor*. The performance factor measures the efficiency of a policy by using the integrated autocorrelation time $\tau_O : C \rightarrow \left[\frac{1}{2}, \infty\right)$ and a cost factor $u : C \rightarrow \mathbb{R}$ where C is the set of all possible trajectories of states that the markov chain can follow through. Due to τ_O being measured in monte carlo updates in this work, to reflect real work performance of generating a trajectory it is multiplied by the cost factor that incorporates all important costs with obtaining a trajectory. The performance factor is given by

$$\epsilon_O(c) \equiv \frac{1}{2\tau_O(c)u(c)} \tag{3.29}$$

the larger the performance factor the better the sampling. A 0 performance factor implies that the samples are perfectly correlated whereas $\epsilon_O(c) = u(c)^{-1}$ means that each sample is independent. The performance factor plays the role of a reward function and an optimal policy is said to maximise the expected performance factor given by

$$\langle \epsilon_O \rangle_{c \sim w} = \sum_{c \in C} \epsilon_O(c) p(c) \tag{3.30}$$

where $p(c)$ denotes the probability of creating the trajectory c. Once this policy is found it is used to sample from the original model.

Chapter 4

Implementation

4.1 RL Setup

Since my work is built upon the foundation set by R.B and N.I, I will first summarise the important set up and findings. They set out explore whether it was possible to use reinforcement learning to construct better cluster sampling algorithms. Specifically, an important milestone would be to be able to effectively sample from the Ising model with plaquette interactions since there is no good sampler known for it currently. They began by trying to use reinforcement learning to to construct a Wolff-type algorithm for the ordinary 2d Ising system before adding plaquette interactions.

Starting from the Ising model's Boltzmann weight in eq (3.24), we can set $J = 1$ since it always enters the combination BJ,

$$\pi(\mathbf{S}) = \exp\left(B \sum_{l \in L} \prod_{i \in l} \sigma_i\right) \tag{4.1}$$

Where are fixed number of sites/nodes ($\mathbf{S} := \sigma_i$) on a 2d lattice with spins $\sigma_i \in \{1, 1\}$ and L denoting the set of links/edges.

The initial RL framework containing the wolff algorithm .

1. Pick a site x randomly and add it to the cluster C. Let σ_0 be the value of this seed spin.
2. For each nearest neighbour y of x, add them to the cluster with some probability $\pi_\theta(y; \sigma; \sigma_0)$. The notation means that this probability sees:
   (a) The set of spins σ.
   (b) The site that we are considering y.
   (c) The value of the "seed spin" in the cluster σ_0.

This probability depends on some parameters etc. in θ, and is something that the RL algorithm should optimize. For practical reasons it should presumably only consider the spins in a window of a certain size around y.

3. Repeat this by now considering each of the neighbours of each of the added sites y until no more sites are added to the cluster.

At the end of it we will flip the spins in C. Let us denote the spin configurations of S before and after the above steps by σ and σ'. It is then important to consider the full transition amplitude of going from $\sigma \to \sigma'$ which we will denote with $W(\sigma \to \sigma')$,

$$\log W(\sigma \to \sigma') = \sum_{x \in C} \log \pi_\theta(x; \sigma, \sigma_0) + \sum_{x \in S \backslash C} \log(1 - \pi_\theta(x; \sigma, \sigma_0)) \ . \tag{4.2}$$

We now need to determine the probability of the reverse move, which, by the same logic, is

$$\log W(\sigma' \to \sigma) = \sum_{x \in C} \log \pi_\theta(x; \sigma', -\sigma_0) + \sum_{x \in S \backslash C} \log(1 - \pi_\theta(x; \sigma', -\sigma_0)) \ . \tag{4.3}$$

Now we have to determine the acceptance ratio for detailed balance to be upheld. If the acceptance probability is $A(\sigma \to \sigma')$, then the detailed balance equation is (see p95 of [9]):

$$\frac{W(\sigma \to \sigma')}{W(\sigma' \to \sigma)} \frac{A(\sigma \to \sigma')}{A(\sigma' \to \sigma)} = \exp\left(-\beta(E(\sigma') - E(\sigma))\right) \tag{4.4}$$

Where $E\sigma$ is the energy of the model for a given spin configuration

$$E(\sigma) = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j - h \sum_i \sigma_i \tag{4.5}$$

where h represents the external magnetic field (set to 0 in our case) This is solved using the following choice of $A(\sigma \to \sigma')$:

$$A(\sigma \to \sigma') = \min\left(1, \frac{W(\sigma' \to \sigma)}{W(\sigma \to \sigma')} \exp\left(-\beta \Delta E\right)\right) \qquad \Delta E = E(\sigma') - E(\sigma) \tag{4.6}$$

Rounding things out with a standard Metropolis Hastings step, the algorithm is completed. Now all that was left to do was to design a good policy and construct a suitable loss/reward function for it to learn. Finally, let's note that Wolff fits precisely into this framework: it is of the form above with the following choice for π_θ:

$$\pi_\theta(x; \sigma, \sigma_0) = 1 - \exp(\min(0, -2\beta\sigma_x\sigma_0)) \tag{4.7}$$

i.e. – if the spin disagrees with the seed, don't add it, and if it agrees with the seed, add it with probability $1 - e^{-2\beta}$. With this choice of probability density, one can analytically show that $A(\sigma \to \sigma') = 1$, which means that for this optimization scheme for the 2d Ising model, Wolff is the best that one can do.

4.2 RL Formulation

We follow notation of [3] to formalise this as a traditional RL problem. We consider the Markov chain $\sigma^{(1)} \to \sigma^{(2)} \to \cdots \to \sigma^{(T)}$ as a sequence of states in a Markov Decision Process with transition operator,

$$\Pi_\theta(\sigma \to \sigma') = W(\sigma \to \sigma')A(\sigma \to \sigma') \tag{4.8}$$

Where we combine the transition amplitude of going from $\sigma \to \sigma'$ with the metropolis hastings step where we accept the transition with probability $A(\sigma \to \sigma')$. The space of state is therefore the space of spin configurations (the space of all possible σ). The space of actions is the space of clusters (connected subset of vertices). Given a cluster C, the transition function simply flips the spins on C in σ to produce σ' – in this sense Π of (4.8) is our policy. The reward is then given by the negative of the loss function, $r(\sigma, \sigma') = -\mathcal{L}(\sigma, \sigma')$. Defined $\tau = (\sigma^{(1)}, \sigma^{(2)}, \cdots, \sigma^{(T)})$ and $R(\tau) = \sum_{t=1}^{T-1} \gamma^t r(\sigma_t, \sigma_{t+1})$, we want to maximise $\mathbb{E}_{\tau \sim P_\theta}(R_\theta(\tau))$ where P is the probability of the whole Markov chain. We have emphasised dependency of both P and R on θ.

4.3 Policy Formulation

The next step was to design a policy that was able to learn how to sample for a given model in eq (3.24) or eq (3.25) with the first goal being able to replicate the Wolff policy in eq (4.7) for the Ising model in (4.1) (the same as (3.24) with $J = 1$). As a result, I evaluated two distinct strategies for determining the probability of adding a spin to a cluster. Both policies aim to integrate a simple yet effective mechanism to enhance the clustering process, and their designs are as follows:

Policy 1: Simple Cluster Policy

This policy is built around a very simple two-parameter cluster algorithm, which serves as a gentle generalization of the Wolff algorithm. It is designed to be straightforward and efficient for a 2-spin $\mathbb{Z}_2$ preserving input.

Parameters:

- `p1` and `p2` are the two parameters initialised with random values and set to be trainable.
- `beta` is an additional parameter that can be used to control the coupling strength.

Window Size:

- A fixed window size of 1 is used.

Output:

- The final output represents a probability density of adding a spin to a cluster via a softmax activation.

```
class SimpleClusterPolicy(torch.nn.Module):

    def __init__(self):
        super().__init__()

        self.p1 = torch.nn.Parameter(torch.randn(1), requires_grad=True)
        self.p2 = torch.nn.Parameter(torch.randn(1), requires_grad=True)
        self.window_size = 1

    def forward(self, x):
        pDrop = torch.sigmoid(self.p1 + self.p2 * x[0] * x[1])
        return torch.cat((1 - pDrop.unsqueeze(0), pDrop.unsqueeze(0)))
```

Policy 2: Policy with Pairwise Products

The second policy, `PolicyPairwiseProducts`, is designed to take into account all pairwise products within a specified window size. This approach utilises the symmetry of the problem and the interactions between pairs of spins.

Window Size:

- The window size is a parameter that determines the scope of the pairwise products considered.

Row and Column Markers:

- These markers are used to extract the upper triangular indices for the pairwise products.

Network Structure:

- The input, consisting of a flattened $N \times N$ window and an additional seed spin value, is passed through a linear layer followed by a Softmax layer to produce the final probabilities.

Output:

- Similar to the first policy, the output consists of a probability density, indicating the probability of adding a spin to the cluster.

```
class PolicyPairwiseProducts(torch.nn.Module):
    def __init__(self, window_size):
        super().__init__()
        self.window_size = window_size
        self.rows, self.cols = torch.triu_indices(
            (self.window_size**2 + 1), (self.window_size**2 + 1), 1)
        self.s = torch.nn.Sequential(
            torch.nn.Linear(
                int((self.window_size**2 + 1) * (self.window_size**2) / 2), 2),
            torch.nn.Softmax(dim=-1)
        )

    def forward(self, x):
        x_pairs = torch.outer(x, x)
        return self.s(x_pairs[self.rows, self.cols])
```

Comparison and Consideration

Simplicity vs. Complexity: The simple cluster policy is simpler and potentially easier to train due to its limited parameter space. However, it may lack the expressiveness needed for more complex interactions.

Expressiveness: The pairwise product policy is more complex and considers interactions between all pairs of spins within the window, potentially capturing more nuanced behaviors but at the cost of increased computational complexity and training time.

Implementation: Both policies utilise the Softmax function to ensure that the output probabilities sum to 1, making the interpretation of the results straightforward.

Evaluating the performance and trainability of these two policies is crucial. This evaluation will reveal how effectively a simple policy can sample from complex models (if it is even possible), and how well a complex policy can generalise and learn to sample from a simple model.

4.4 Loss Functions

Before I made my own candidate for the loss function, R.B. and N.I. had decent success with using the pointwise covariance. This loss was calculated over an ensemble of M parallel chains and before we proceed let $I \in \{1, 2, \cdots, M\}$ distinguish the chains in the ensemble and $\sigma^I_{i,t}$ be the spin at site i of chain I at monte carlo time t. The pointwise covariance at each timestep is given by

$$\mathcal{C}^I_{t,t+1} := \langle \sigma^I_t \sigma^I_{t+1} \rangle_{\text{latt}} = \frac{1}{N^2} \sum_i \sigma^I_{i,t} \cdot \sigma^I_{i,t+1} \tag{4.9}$$

This is used to construct the loss function by taking the sum over the entire the entire ensemble

$$\mathcal{L}_{t,t+1} = \sum_I (\mathcal{C}^I_{t,t+1})^2 \tag{4.10}$$

This is then used to optimise the policy at each timestep. We can measure our success with this approach by directly plotting the movement of the probability density as shown in figures 4.1 and 4.2 with the aim to be to replicate wolff as in eq (4.7). Simply put, we would like for the orange star in these figures to be as close as possible to the grey dot. The models were trained for standard Ising model (4.1) $\beta = 0.4, J = 1$ on a 10×10 lattice.

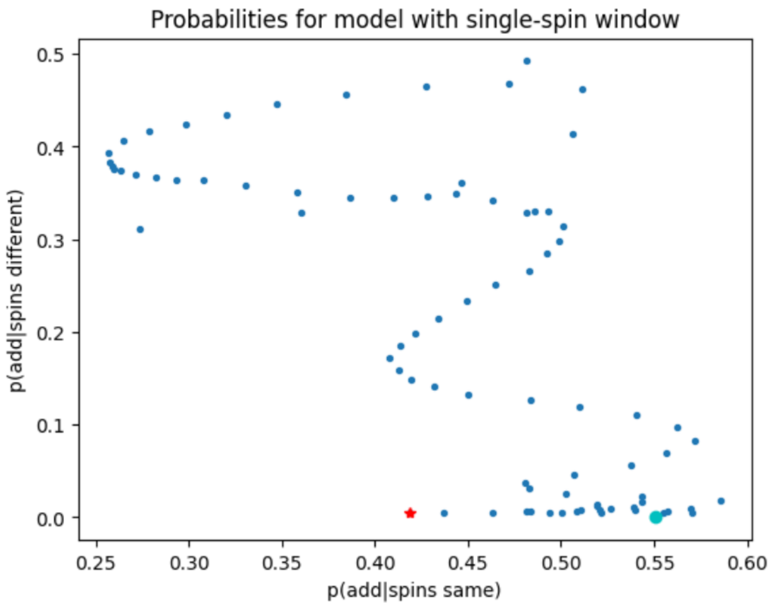

Figure 4.1: Movement of the probability density for the SimpleClusterPolicy using eq (4.10) as the loss function during training. Grey dot is the Wolff policy (4.7)

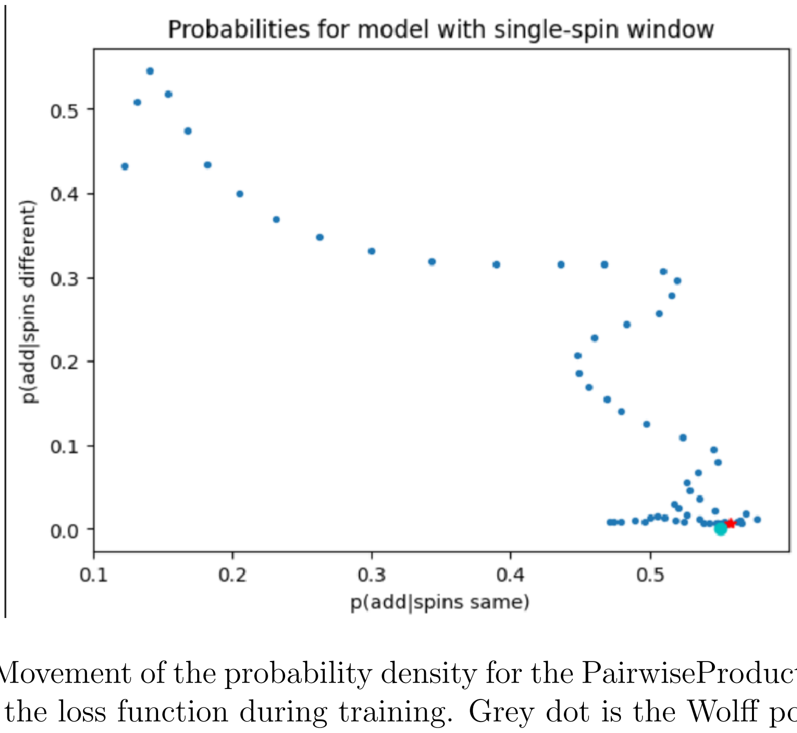

Figure 4.2: Movement of the probability density for the PairwiseProductPolicy using eq (4.10) as the loss function during training. Grey dot is the Wolff policy (4.7)

The results it suggests that this implementation of the loss performs well since it is able to replicate wolff to a certain degree of accuracy. In reality, It can be seen that the density moves in a horizontal window of the density without actually converging on the grey dot. In any case, the policies were able to effectively learn the parameters required to bring it close to wolf. It was hard to find any issues with this current loss but before taking a look at how it performs on the on the plaquette model, can we do better than this?

4.5 Autocorrelation time loss

The loss function used in the previous section came to be due to other implementations having very clear flaws. One such flaw being just adding spatial variance on the spin configurations suggesting a lower loss which was pointed out R.B and N.I.'s notes. This would be no better than just using a random number generator and that the quantity that we actually want to minimise was the integrated autocorrelation time. This was done indirectly using the pointwise covariance but only upto a certain degree. I set out to see if it was possible to do this directly.

The first step was to implement a way to calculate this statistic. I did this using the implementation in Sokal's notes [1]. Implementing eq (3.8) for n samples we can simplify the formula as follows,

$$\hat{\tau}_{int} = \frac{1}{2} \sum_{t=-(n-1)}^{n-1} \hat{\rho}(t)\lambda(t) = \frac{1}{2} \sum_{t=-M}^{M} \hat{\rho}(t) \tag{4.11}$$

Where a suitable window size M is determined as suggested in Sokal's notes [1]. The choice of the function f was the energy of the spin configuration, so at timestep t, f_t is given by

$$f_t = f(\sigma^t) = E\left(\sigma^t\right) = -J \sum_{\langle i,j \rangle} \sigma_i^t \sigma_j^t - h \sum_i \sigma_i^t \tag{4.12}$$

Which was then used to calculate the autocorrelations in order to calcluate τ_{int}. The next step was to somehow use this in the existing framework. Immediately, these questions came to mind:

1. How many samples shall I use for the estimate?
2. Is it possible to use earlier samples?
3. How do I incorporate the state transition probabilities with $\hat{\tau}_{int}$?
4. Will I have to scale/transform the estimates the estimates of $\hat{\tau}_{int}$ with some kind of function?

In the previous implementation of the loss, each training step only required generating one sample per chain in the ensemble. Training time was actually dominated by the time needed to generate samples, so by having a small number of samples generated per training step, training times were short. However, to get a reliable estimate of $\hat{\tau}_{int}$, one would need around 400 samples. Compromising on this with, say, 100 samples would result in the estimate of $\hat{\tau}_{int}$ having a very high variance for a fixed policy, which would make training very unstable. Even in the best case scenario of using around 400 samples, training times would increase hundredfold. Trying to remedy this, I tried to see if it was possible to use earlier samples. I quickly realised this would not work and would at best half the training times. For example, using 200 samples from the previous from the previous training step and generating 200 new ones. The problem of long training times was still there at the cost of worsening our estimate of $\hat{\tau}_{int}$ for training. As a result I decided to use a single chain and not an ensemble of them. This way I could get a way with using 400 samples per training step rather 4000 if I was using an ensemble of size 10. This was still really long but was actually feasible.

Now all that was left was to consider how to incorporate the state transition probability. The problem here is that it is not as simple as taking the probability of $\Pi_\theta(\sigma^t \to \sigma^{t+1})$ but rather $\Pi_\theta(\sigma^t \to \sigma^{t+1} \to \cdots \to \sigma^{t+n})$. I thought of just considering $\Pi_\theta(\sigma^t \to \sigma^{t+n})$ but that would miss the point entirely of considering all the state transition probabilities that resulted in the estimation of $\hat{\tau}_{int}$. As a result, I chose to change the transition operator for this by taking the average transition probability over the chain.

$$\Pi'_\theta(\sigma^t \to \sigma^{t+n}) := \frac{1}{n}\sum_{i=t}^{n} \Pi_\theta(\sigma^i \to \sigma^{i+1}) \tag{4.13}$$

I fear that this may wash out most of the probabilities but this is the best I can come up with. For now I will use $\hat{\tau}_{int}$ and look to experiment with applying a function or scaling it later. Like in the previous section, I try to see if both policies are able to replicate the wolff policy after training, the results can be seen in figures 4.3 and 4.4. The models were trained for standard Ising model (4.1) for $\beta = 0.4, J = 1$ on a 10×10 lattice estimating $\hat{\tau}_{int}$ using 400 samples.

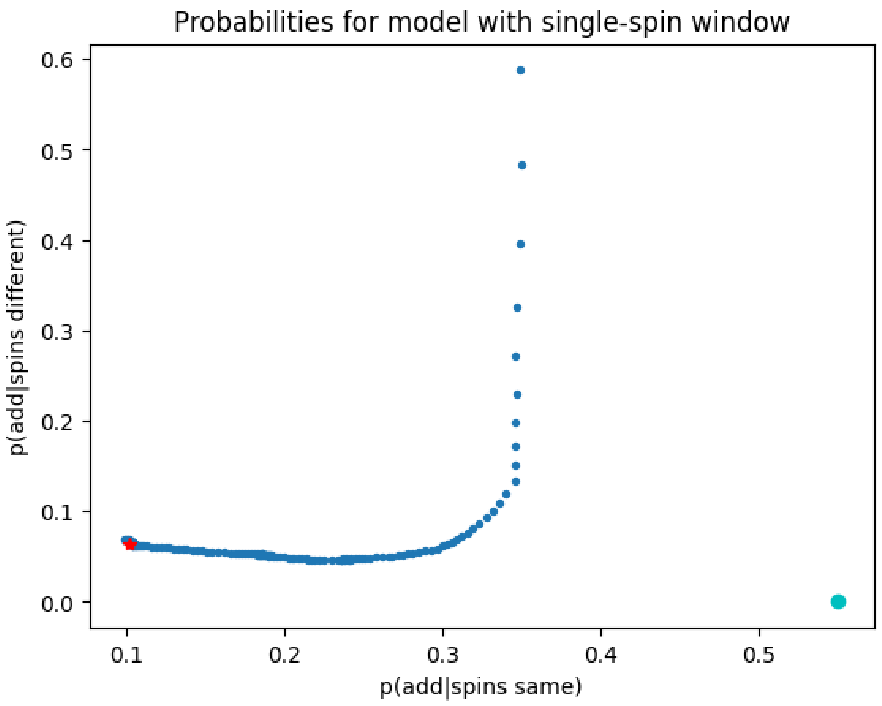

Figure 4.3: Movement of the probability density for the SimpleClusterPolicy using eq (4.11) as the loss function during training. Grey dot is the Wolff policy (4.7)

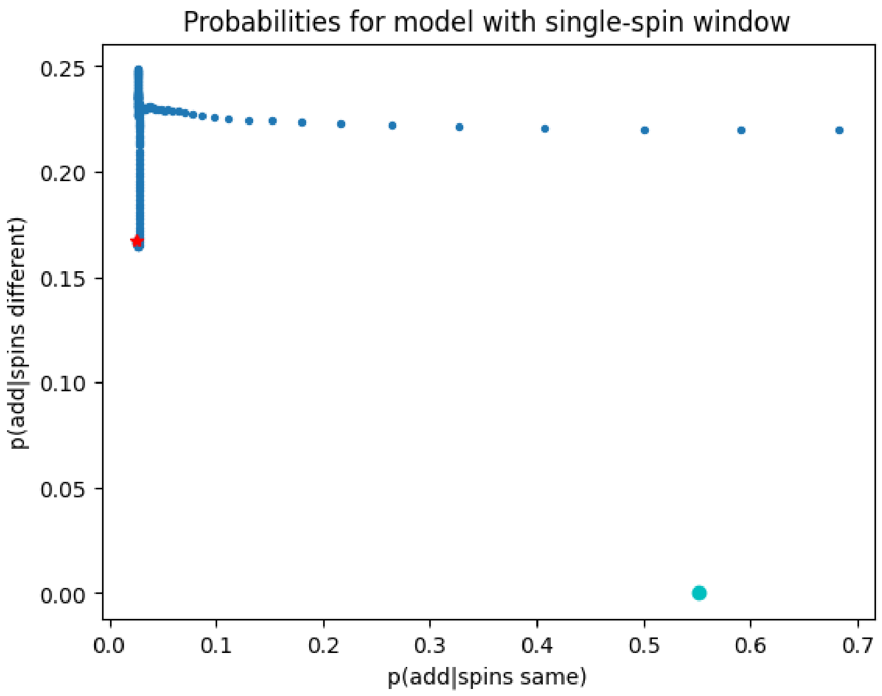

Figure 4.4: Movement of the probability density for the PairwiseProductPolicy using eq (4.11) as the loss function during training. Grey dot is the Wolff policy (4.7)

These results were not particularly the ones I desired. Both policies moved in the opposite direction from the wolff policy and then got stuck in their own respective regions with very little movement. After trying many different transformations, one being $-\log \hat{\tau}_{int}$, they all showed the same behaviour as figures 4.3 and 4.4 with no consistent movement towards wolff. It was clear to me that this approach was flawed and that I should try something else. My main suspicion as to why this approach did not work very well was due to the implementation of the transition operator. The concern of the amount of samples used to estimate $\hat{\tau}_{int}$ was still there but it was clear that the adjusted transition operator was fundamentally wrong. Fortunately, I had another approach in mind.

4.6 ESS as a reward

By minimising τ_{int} the effective sample size (ESS) would be increasing as a result. This is just a byproduct of the formula stated in eq (3.11). Putting more emphasis on the ESS rather than τ_{int} I was able to look at the problem from another perspective and try a different approach.

Speaking strictly in RL terms, I viewed the policy that was responsible for generating the samples as the "agent". Like before, the environment or state space would still be the set of all possible spin configurations (all possible σ). After generating a sample the "agent" would either accept or reject it. The key difference is that the action itself would be accepting or rejecting of the freshly generated sample rather than both generating the sample and accepting/rejecting the sample. This mean that our transition operator, $\to$, taking us from $\sigma^t \to \sigma'$ (the proposed sample) would be (4.6). Writing this explicitly,

$$\Pi'_\theta(\sigma^t \to \sigma') := A(\sigma^t \to \sigma') = \min\left(1, \frac{W(\sigma' \to \sigma^t)}{W(\sigma^t \to \sigma')} \exp\left(-\beta \Delta E\right)\right) \tag{4.14}$$

Then after a metropolis hastings step, we accept σ' with probability (4.14) and set $\sigma^{t+1} = \sigma'$ otherwise we reject it and set $\sigma^{t+1} = \sigma'$. For the reward function I would be considering the ESS for the previous m samples. I will use the following notation,

$$r_t(m) := ESS(-m) \equiv ESS(\{\sigma^{t-m+1}, \cdots, \sigma^t\}) \tag{4.15}$$

Previously I mentioned that this would not be a good idea since it would give us a bad estimate of τ_{int} and would not help us that much. While that was true for that approach, there was a way to make it work for me here. By considering discounted reward,

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{4.16}$$

for $\gamma \in (0,1)$ the samples from the policy used in the previous time step would slowly get washed out for the later rewards. An example would make this clear, if the policy at time t, π^t, is used to generate the samples s^{t+k} for $k > 0$ and the policy used at the previous time step, π^{t-1} generated the samples s^{t-k}, if we are considering the last 300 samples for the reward then the value r_{t+150} used to calcuate the cumulative reward would have 150 samples from π^t and 150 samples from π^{t-1}.Note, I use s instead of σ to denote the samples since they are not to be confused with the state of the markov chain since the purpose of the extra samples s^t are used to generate the cumulative reward by "looking ahead". The extra samples are stored and used in future iterations. The idea is to have a high discount factor (γ close to 1) so that future rewards are given high weight in hopes that the sampler is guided to a region of the parameter space where the ESS is maximised and given a chance to converge by considering the samples generated by the previous version of the sampler. By setting $m = 300$ I try to maximise the estimated cumulative reward,

$$\hat{G}_t = \sum_{k=0}^{m} \gamma^k r_{t+k} \tag{4.17}$$

This way I would be I would generate 300 new samples every training step and "looking back" to the newest 300 samples each time I calculate the ESS. I attempt this approach on the PairwiseProductPolicy on the same model used in the previous trial runs. While my first attempt was not perfect, it did show some promise. The results can be seen in figure 4.5

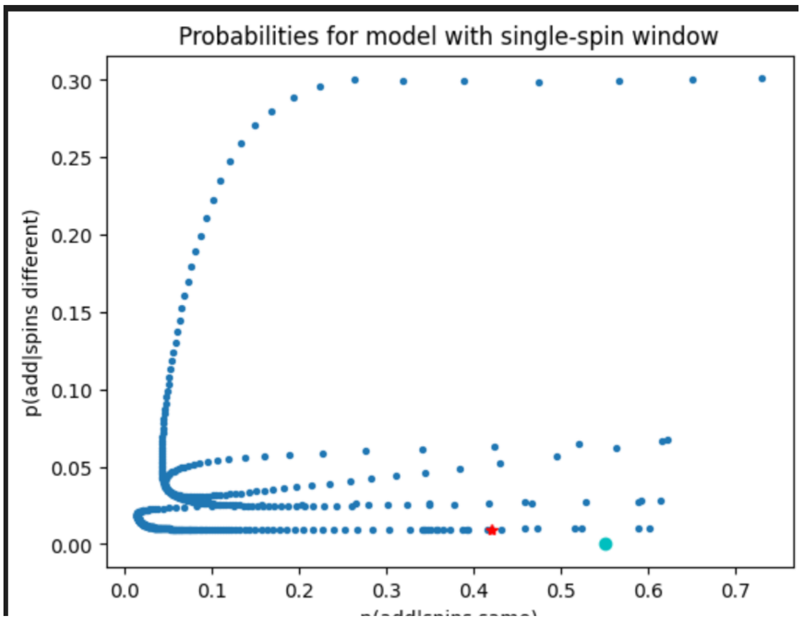

Figure 4.5: Movement of the probability density for the PolicyPairwise using eq (4.17) as the discounted reward during training. Grey dot is the Wolff policy (4.7)

This time around the policy was able to get close to wolff but then quickly move away, only to get close again and then move away again. I could clearly see that the model had clear issues converging since the discounted rewards were larger the close the policy was to wolff but it would quickly move away. I was hoping that by using the current (π^t) and previous version (π^{t-1}) of the policy to estimate the ESS for for the last m samples, the policy would not change much between iterations and would then converge in a region where the ESS reward is maximised. I suspected that the learning rate was too high when the policy got close wolff and so I decided to add a decaying learning rate. This on its own did not help that much and after experimenting with it, the training procedure exhibited results that were very similar to figure 4.5. It was at this point I realised that I forgot the notion of metastability or τ_{exp} (section 3.1). I was not allowing the chain to reach a metastable region before calculating the ESS. I decided to let the chain run for a certain amount of time between training steps to improve the estimates of the ESS. This meant that training times would increase but would allow for better accuracy. At the same time I found an R package [10] that was able to estimate the ESS. Comparing this with my implementation, it gave very similar results in calculating the ESS for large sample sets but had lower variance when estimating for smaller sample sets. I will provide my estimate and the R package estimate when evaluating the samplers but for training purposes I decided to use the R package since I would be estimating with smaller sample sets. In summary, I decided to:

1. Add a decaying learning rate. After every 20 training steps, the learning rate would decrease by a factor of $\alpha \in (0, 1)$
2. After optimising the policy at each training step, I would allow the chain to run for a certain amount of time before the next training step. This would be a hyper parameter to determine.
3. I would instead use an R package [10] to estimate the ESS during training.

I attempt this altered approach on both policies for the same model used in previous attempts. The results can be seen in figures 4.6 and 4.7.

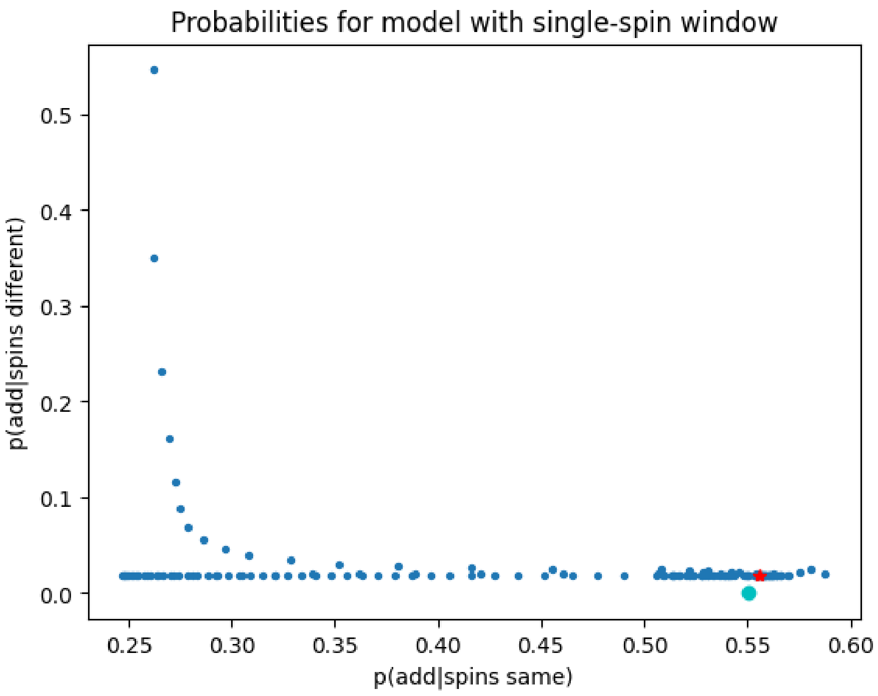

Figure 4.6: Movement of the probability density for the SimpleClusterPolicy usingeq (4.17) as the discounted reward during training. Grey dot is the Wolff policy (4.7)

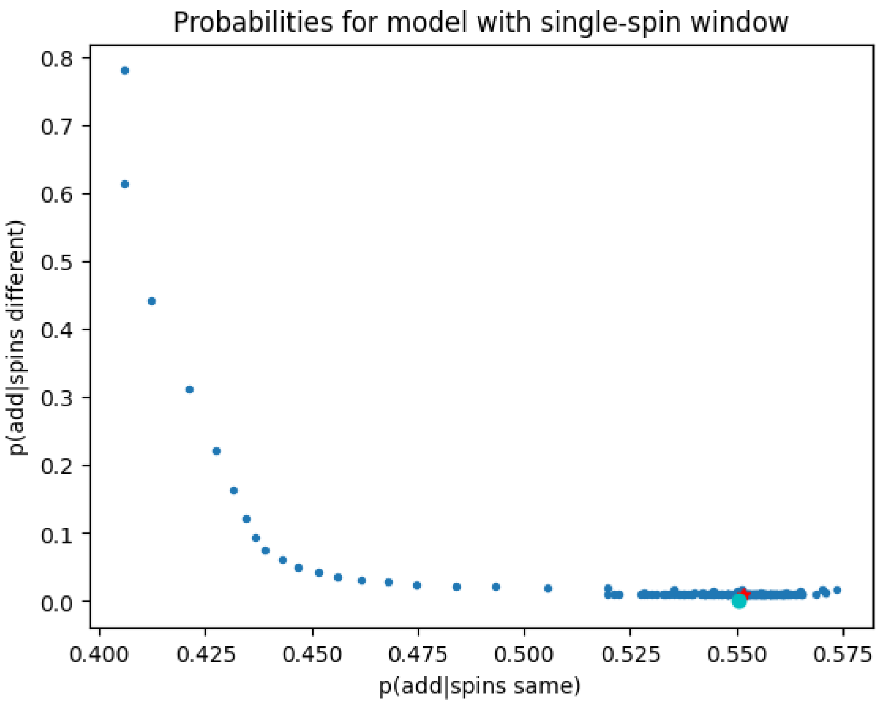

Figure 4.7: Movement of the probability density for the PairwiseProductPolicy using eq (4.17) as the discounted reward during training. Grey dot is the Wolff policy (4.7)

This approach worked alot better and was able to get much closer to wolff than when using the pointwise covariance (4.9) as the loss. For the SimpleClusterPolicy it was able to get close to wolff and then oscillated in a horizontal window around it. The PairwiseProductPolicy showed the same behaviour but with a much smaller window. With the help of the decaying learning rate, I was able to observe in both cases that the window of oscillation grew smaller around wolff as the learning rate decayed. With this I was able to show convergence towards wolff. Not only this, when I compared these policies to the ones trained using the pointwise covariance loss function (4.9), they peformed much better. I will elaborate on this further in the evaluation section but I compared the samplers by calculating the ESS on a large sample set. Now that I was able use the framework to learn a good sampler for the standard Ising model, I moved on to see how well the framework would perform for the Ising model with plaquette interactions (3.25).

4.7 Plaquette interactions

Now I will look to sample from the Ising model with plaquette interactions (3.25). Like before I will set $J = 1$ and will consider the model for specific values of K. Writing this explicitly, the models I will try to sample from will be,

$$\pi(\mathbf{S}) = \exp\left(\beta \sum_{l \in L} \prod_{i \in l} s_i + K \sum_{p \in P} \prod_{i \in p} s_i\right) \quad \begin{array}{l} \text{for } K \in \{-0.4, -0.2, 0.2, 0.4\}, \\ \beta \in \{0.4\} \end{array} \tag{4.18}$$

Unlike before this was uncharted territory. There was no known target density for the policy to reach for so the best I could do was show that the discounted rewards converged and that there was a large ESS for a large sample set. I experimented using the PolicyPairwiseProducts on a 10×10 model size using the procedure. The results can be seen below.

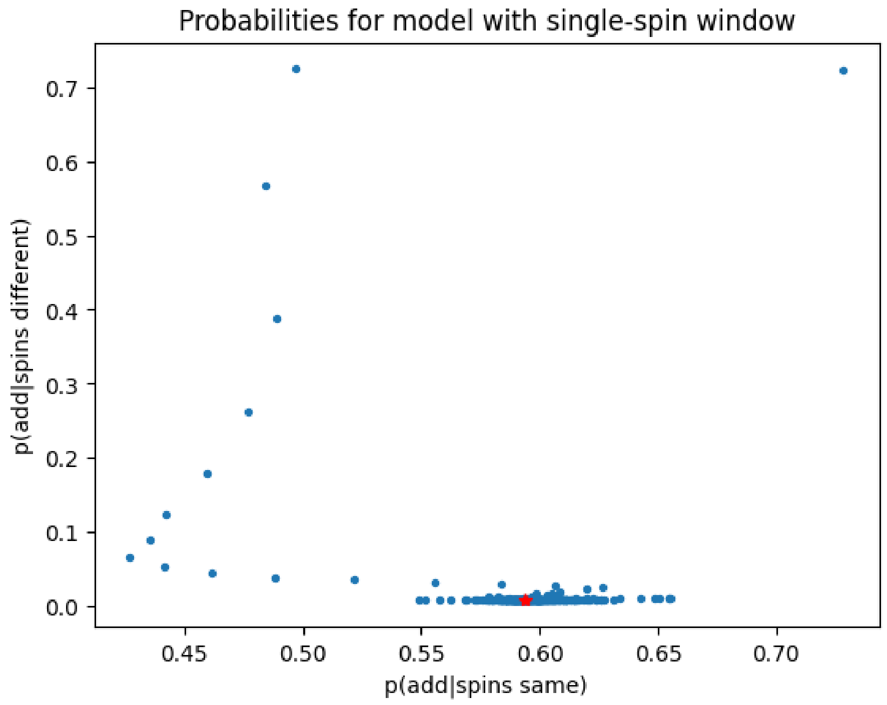

Figure 4.8: $K = 0.2$. Movement of the density for the model in (4.18)

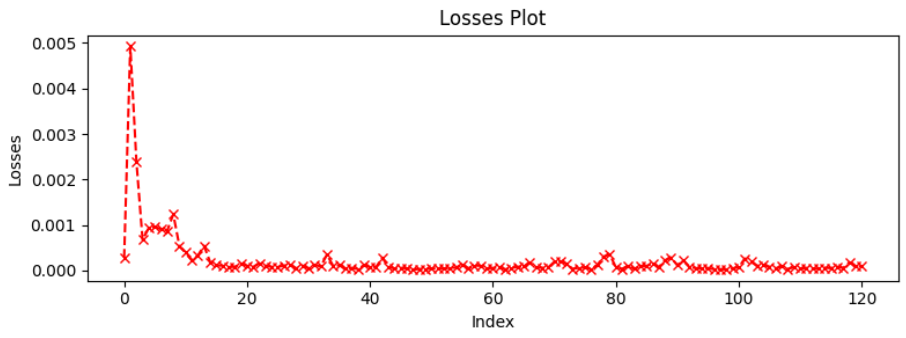

Figure 4.9: $K = 0.2$. Plot of the sum of the action probabilities divided by the discounted rewards

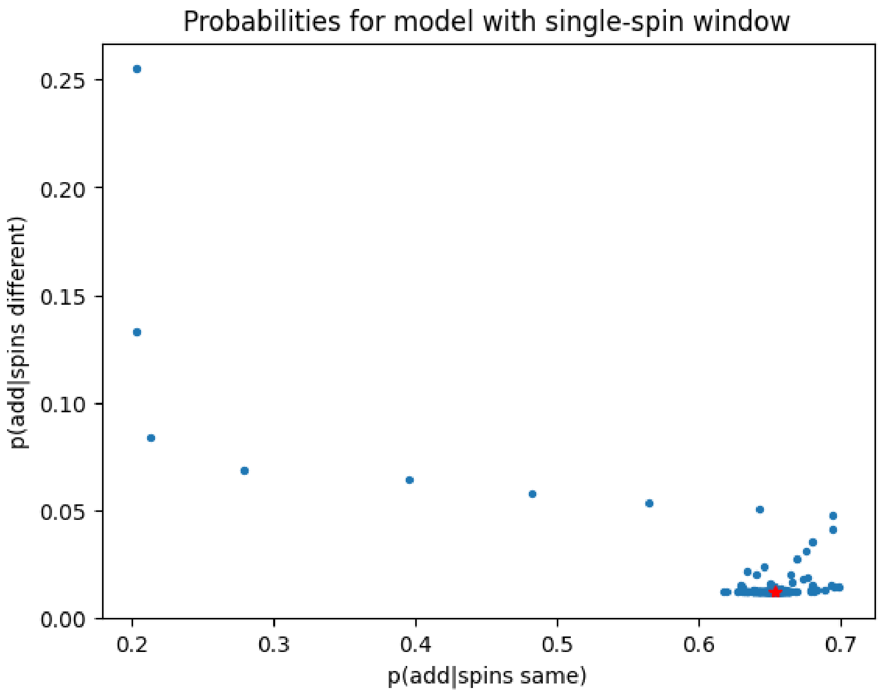

Figure 4.10: $K = 0.4$. Movement of the density for the model in (4.18)

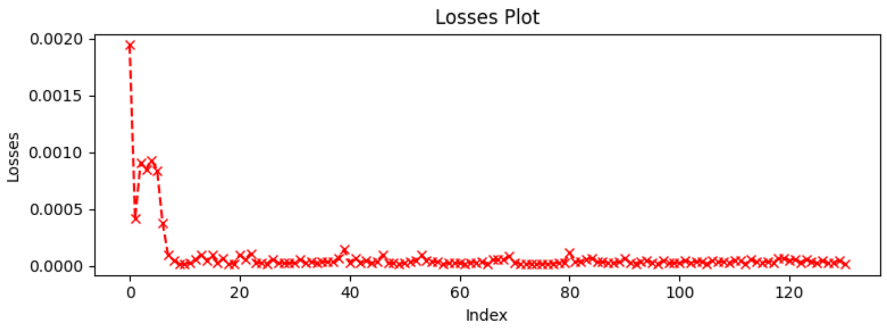

Figure 4.11: $K = 0.4$. Plot of the sum of the action probabilities divided by the discounted rewards

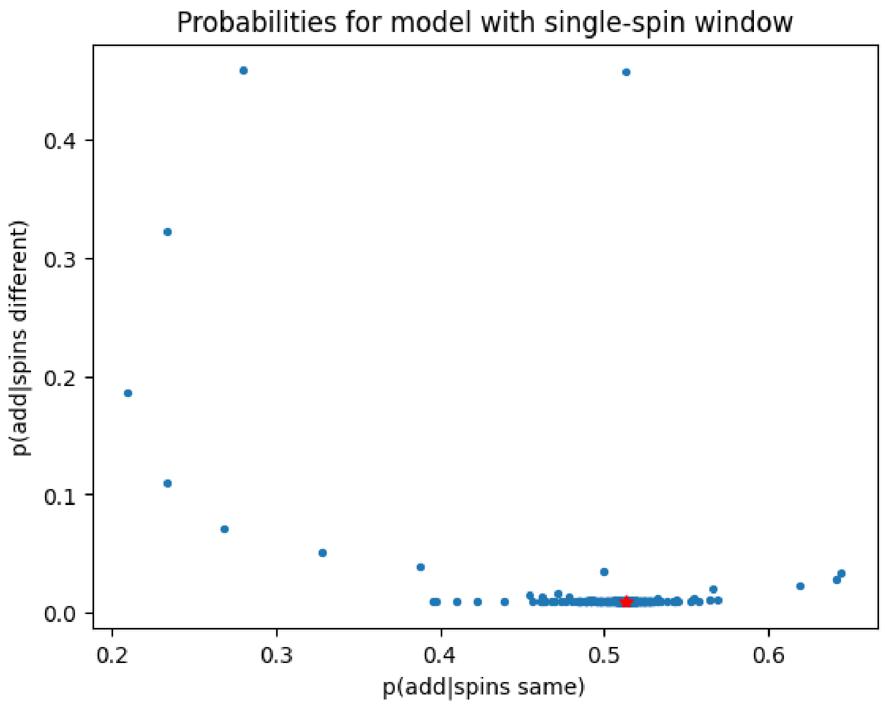

Figure 4.12: $K = -0.2$. Movement of the density for the model in (4.18)

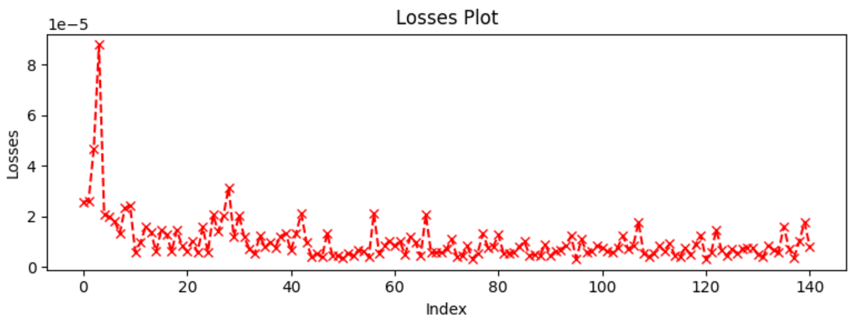

Figure 4.13: $K = -0.2$. Plot of the sum of the action probabilities divided by the discounted rewards

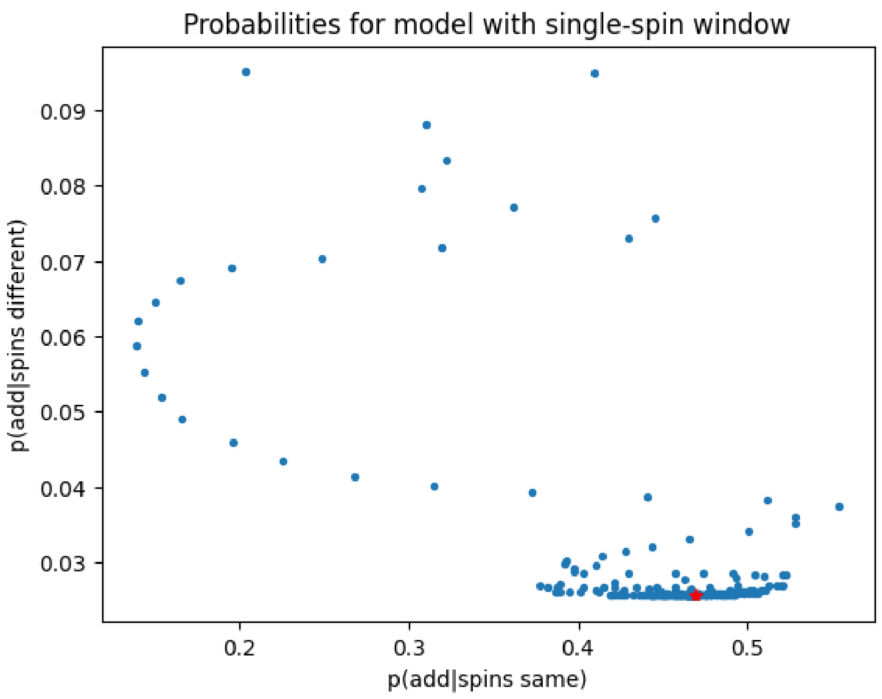

Figure 4.14: $K = -0.4$. Movement of the density for the model in (4.18)

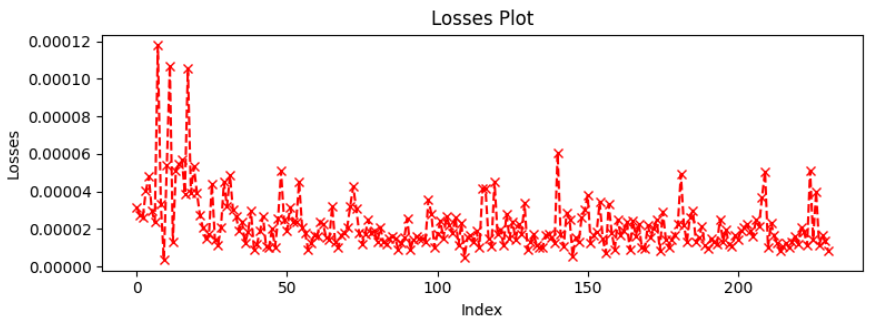

Figure 4.15: $K = -0.4$. Plot of the sum of the action probabilities divided by the discounted rewards

Training for the models with $K > 0$ was a success since both the probability density and the losses converged to a minimum and produced high values for the ESS (seen in table 5.2). For experiments where we had $K < 0$, the densities managed to converge whereas the losses did not. The ESS values it produced were also significantly smaller than when $K > 0$ and there was no clear indication as to why this was the case.

Chapter 5

Evaluation

The ESS estimates in the tables below were taken for 5000 samples and were averaged over 10 chains. I use the abreviations Simple for the SimpleClusterPolicy and Pairwise for the PairwiseProductsPolicy. It should be noted that if the estimated effective sample size is larger than $n = 5000$, this means that there are some negative autocorrelation estimates i.e. for some values of t in eq (3.6). What this means is that if there are negative correlation in the samples, the variance of the estimator from correlated samples can be smaller than the variance of the estimator from independent samples. This in turn leads to larger effective sample size.

5.1 ESS results table

Policy	Training method	My ESS estimate	R package - mcmcse estimate
Wolff	Fixed (no training)	7578.95	8656.32
Simple	Pointwise Covariance loss	4190.23	4462.45
Simple	ESS Reward	4607.86	5074.5
Pairwise	Pointwise Covariance loss	4640.50	4992.01
Pairwise	ESS Reward	5674.27	5713.43

Table 5.1: Standard Ising model

The purpose of the table above is to highlight the clear improvement made when using the ESS as a reward function (4.17) over the pointwise covariance loss (4.10) for the same policy. The ESS values increased for both policies on average when estimating it with both my implementation and mcmcse [10]. The PairwiseProductPolicy using the ESS reward function managed to get ESS values larger than n making it the best performing policy that was trained. While the trained policies were not able to get ESS values close to the original wolff algorithm, it is more important to get ESS values or n_{eff} to be close to or greater than n as that is enough to say that most of or all the samples produced were independent. This is evidence that this training method was clearly superior and has potential to be generalised to learn samplers for models other than the Ising model.

Policy	Training method	K	My ESS estimate	R package - mcmcse estimate
Pairwise	ESS Reward	0.2	6638.55	6726.38
Pairwise	ESS Reward	0.4	5436.24	4380.08
Pairwise	ESS Reward	-0.2	1476.15	1403.57
Pairwise	ESS Reward	-0.4	455.98	457.32

Table 5.2: Ising model with plaquette interactions

From this table we can see that the policies were able to learn to sample from the Ising model with plaquette interactions with varied success. They were able to sample particularly well for the models with $K > 0$ but not so well for the other case. It was not clear to me if this was an engineering problem on my end or something to do with the actual physics behind the model. This could also just be a limiting factor of the Policy architecture itself, not being able to characterise the model well enough for when $K < 0$.

Overall, my results indicate that using reinforcement learning to learn better samplers without the use of datasets was possible, as shown by my results. There was now also a way to effectively sample from the Ising model with plaquette interactions for when the plaquette coupling term $K > 0$. Both of which extends the state of the art in their own way and the concepts used for training can be generalised for other models.

5.2 Conclusions and future work

I touched upon the fact that the concepts used for training can be generalised for other models. To elaborate, all that was required was to design a suitable policy that characterises the model to be sampled for, a way to use said policy to generate samples and a well suited function f to map the samples to a real value. This choice of f should be based off the context of the model being sampled for. Then the training method used earlier can be employed to learn the correct parameters of the policy. The drawbacks of this approach mainly comes from the setting of hyperparameters. Both the training time and the learning rate was important to tune but what was more crucial was the learning rate decay. There was no good way of telling when the policy had converged during training since the ESS estimate calculated in each iteration of the training loop was based on a small sample set causing it to be subject to variance. Therefore the learning rate decay played a huge part in convergence as the policy will eventually settle in region where the ESS is maximised. The amount of samples used to estimate the ESS as well as the time for the chain to reach equilibrium τ_{exp} in each training step were also parameters to be set. The ESS estimate improved in accuracy just by adding more samples whereas τ_{exp} was hard to determine by nature. Again the problem raised by τ_{exp} was easily solved by just letting the chain run for a long period at the cost of training times being increased. The trade off between accuracy and training times was present in all of the hyperparameters listed so far. The problem raised by the learning rate decay can be mitigated by determining a better convergence criterion and if there was a way to estimate an appropriate τ_{exp} there would be less guesswork in setting that parameter. However the problem raised by estimating the ESS for small sample sets will always be there.

As for the future work for sampler I produced, all that was left to be able to sample for the Ising model with plaquette interactions for when the plaquette coupling term $K < 0$. This could be done by adjusting the policy architecture or by maybe altering the reward function.

Bibliography

[1] Sokal AD. Monte Carlo Methods in Statistical Mechanics. New York; 1996. Chapter 3.

[2] Stan Development Team. Stan Reference Manual [Internet]. 2011–2022 [cited 2024 Jun 19]. Section 16.4. Available from: `https://mc-stan.org/docs/reference-manual/effective-sample-size.html`

[3] Correia A, Worrall D, Bondesan R. Neural Simulated Annealing. arXiv e-prints [Internet]. 2022 Mar [cited 2024 Jun 18]. Available from: `https://arxiv.org/abs/2203.02201`

[4] Barbu A, Zhu SC. Monte Carlo Methods. Springer Nature; 2020.

[5] Robert CP. The Metropolis-Hastings Algorithm. arXiv [Internet]. 2015 Apr [cited 2024 Jun 18]. Available from: `https://arxiv.org/abs/1504.01896`

[6] Bojesen T. Policy-guided Monte Carlo: Reinforcement-learning Markov chain dynamics. Phys Rev E. 2018;98:063303.

[7] Wang L. Exploring cluster Monte Carlo updates with Boltzmann machines. Phys Rev E. 2017;96(5):051301. doi: `10.1103/PhysRevE.96.051301`.

[8] Luijten E. Introduction to Cluster Monte Carlo Algorithms. Department of Materials Science and Engineering, Frederick Seitz Materials Research Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, U.S.A. [Internet]. 2021 [cited 2024 Jun 18]. Available from: `https://csml.northwestern.edu/resources/Reprints/lnp_color.pdf`

[9] Newman MEJ, Barkema GT. Monte Carlo Methods in Statistical Physics. Oxford: Oxford University Press; 1999.

[10] Flegal JM, Hughes J, Vats D, Dai N. mcmcse: Monte Carlo Standard Errors for MCMC (version 1.4-8) [Internet]. 2022 [cited 2024 Jun 15]. Available from: `https://cran.r-project.org/web/packages/mcmcse/`