Master's Thesis

Hyperparameter Optimization using Ranking Loss Surrogates

Albert-Ludwigs-University Freiburg
Representation Learning Department

Abdus Salam Khazi

July 8, 2022

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof

Place, Date	Signature

Abstract

Hyperparameter optimization is vital in training any machine learning model. The hyperparameter configuration obtained from the optimization directly impacts the machine learning model's performance. Hence various hyperparameter optimization methods have been proposed in the past decade. Among these methods, Sequential Model-Based Optimization (SMBO) methods have successfully predicted suitable configurations at a relatively lesser computational cost. Moreover, using transfer learning has furthered their performance. However, the learning and design of surrogates used in SMBO is still an active area of research. This thesis proposes a different perspective on the learning and usage of surrogates in the SMBO algorithm. It formulates the selection of configurations to evaluate in the SMBO algorithm as a ranking problem. The surrogates are learned using ranking losses and work in the ranking space rather than the output space of the hyperparameter objective function. This thesis shows that using transfer learning surrogates in ranking space is more advantageous in a problem domain like hyperparameter optimization, where metadata is very scarce. The surrogate learning using listwise ranking losses gave the best performance among all ranking losses. Moreover, the thesis also proposes incorporating a weighting strategy that is position dependent in the listwise ranking loss to improve its performance. It also proposes making this surrogate context-aware by integrating Deep Sets into its architecture. It concludes that using a surrogate with Deep Sets learned with a weighted listwise ranking loss gave hyperparameter optimization results on par with state-of-the-art transfer hyperparameter methods on the HPO-B benchmark.

Contents

Chapter 1

Introduction

Machine learning has become a ubiquitous tool in the modern economy. Due to the availability of extensive data at a meager cost, applying machine learning methods has become very accessible. Computer vision, marketing, customer service, and cybersecurity are some of the many domains that use machine learning. A critical factor in its success is that it gives a general framework for learning the models. One only needs to use the data and apply the formulated optimization to train the models. In other words, complex models are created using this framework without needing to program them explicitly.

Even though most machine learning models can work out of the box, their performance does not come for free. Their performance is sensitive to the hyperparameters used during the model training. Some examples of hyperparameters are the learning rate, the type of optimizer, the architecture of the neural networks, and the regularization parameter. One needs to tune the model's hyperparameters to get the most performance. Tuning the hyperparameters to get the best performing machine learning model is called Hyperparameter Optimization (HPO in short).

A brute force method for an HPO would be to try out all possible configurations. This method is intractable because the space of hyperparameter configurations may be continuous. Even if the space is discrete, the number of combinations may be prohibitively large. Moreover, evaluating even one hyperparameter configuration may take many days. This task of evaluating an HP configuration is called the HPO objective function. Considering the cost of the HPO objective function, more sophisticated methods have been developed over the past decades. Sequential Model-Based Optimization (SMBO) [2] is one method that has been used successfully for HPO. This method uses cheap approximations of the actual HPO's objective function to predict good HP configurations. These cheap approximations are called surrogates which need to be learned using the available hyperparameters configurations and their evaluations. This thesis proposes using the concept of ranking to learn the surrogates in SMBO. It proves that using this ranking surrogate in SMBO is a viable alternative to other methods. Hence, the proposed idea in this thesis is called **Hyperparamter Optimization using Ranking Loss Surrogates**.

1.1 Objective

To define the objective and scope of this thesis, let us first understand how one does HPO. To find the best hyperparameter for any machine learning model m, we quantify a hyperparameter configuration $\mathbf{x}$ by a real-valued number $v \in \mathbb{R}$. If we define that

$$\mathbf{x}_1 \succ \mathbf{x}_2 \iff v_{\mathbf{x}_1} < v_{\mathbf{x}_2}$$

then HPO can be defined mathematically by an abstract function, say, $f(\mathbf{x}) \mapsto \mathbb{R}$ as

$$\underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{S}$$

where $\mathbb{S}$ is the space containing the set of all hyperparameters, also called hyperparameter search space. The function f is evaluated in the following chronological steps:

1. Using a given hyperparameter configuration $\mathbf{x}$, we train our model m to obtain the model $m_{\mathbf{x}}^{trained}$. It consists of learning the parameters of our model, for example, learning the weights and biases of a Deep Neural Network. We use any given training data to learn this model.

2. Validation split of the data is passed through $m_{\mathbf{x}}^{trained}$ to obtain the validation prediction results. These results are evaluated based on an evaluation criterion 'eval'. The result of this evaluation is a real-value that gives a score for the configuration $\mathbf{x}$.

The function f can hence be written as

$$f(m_{\mathbf{x}}^{trained}(\text{Data}_{\text{val}})) \mapsto \mathbb{R}$$

Finally, the HPO problem can be defined using the following equation:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \; f(m_{\mathbf{x}}^{trained}(\text{Data}_{\text{val}})) \mapsto \mathbb{R} \quad \forall \mathbf{x} \in \mathbb{S} \tag{1.1}$$

One way to view this objective function non-mathematically is that we are trying to select a hyper parameter setting of the given model to obtain the best (lowest) validation error [48]. Apart from being computationally expensive, HPO has the following constraints:

- It is a non-convex optimization problem.
- The process of getting $m_{\mathbf{x}}^{trained}$ from m is stochastic hence the value $v_{\mathbf{x}}$ is noisy.
- Some dimensions have conditional constraints. The values of some dimensions may depend on the values of others. For example, the number of neurons in layer three only makes sense if we have three or more layers in a neural network.
- The search space is hybrid in terms of continuity. Some dimensions (variables) may be continuous, while others may be discrete. Using a gradient method is hence not trivial.

This thesis aims to improve the HPO by learning good surrogates of the HPO objective function. It uses the concept of ranking to select the best HP configuration in the SMBO algorithm. Since the surrogate should behave like a ranking, ranking losses are used to learn the surrogate of the function f. The thesis only targets the discrete HPO problem. This is because ranking requires sets with discrete objects. In order to improve the performance of the learned surrogate, the thesis also tries to make it context-aware.

1.2 Overview

This report is divided into six chapters. The introductory chapter (i.e., this chapter) first introduces the concept of Hyperparameter optimization - this thesis's area of research. Then it describes the objective or primary aim of the thesis. In chapter 2, we discuss the basic concepts related to HPO optimization. We first discuss different types of HPO. We then discuss the Bayesian optimization and its subcomponents in detail, as understanding these concepts are a prerequisite to understanding the research presented in this report. We also talk about designing a context-aware surrogate. In chapter 3, we dive deep into the concept of ranking and ranking losses. We discuss in detail the mathematical workings behind the loss functions. We also discuss the weighting extension of the loss functions to make it better for the HPO problem.

In addition, we describe the primary baselines this thesis implements. Chapter 4 describes how the concepts of ranking and Bayesian Optimization are integrated to build the proposed ranking loss surrogate. We also talk about designing our surrogate to make it context-aware. In chapter 5, we do an ablation study of the implemented baselines and the proposed model's various components. Finally, in chapter 6, we conclude the report by describing some advantages and limitations. We also propose a few exciting future research topics.

Chapter 2

Related Work

This chapter discusses the concepts already found in previous literature relevant to the HPO problem. It can be divided into three parts. In the first part, we describe different classifications of HPO solutions. These include Blackbox HPO, Online HPO, Multi-fidelity HPO, and Transfer Learning HPO. Classifying the HPO solutions based on these criteria helps us quickly identify their advantages and drawbacks. However, it is essential to note that these classifications are not exclusive to one another. The second part is dedicated to Bayesian Optimization, a method that can be used for the HPO problem. We discuss in detail Bayesian optimization as this is what we use in the proposed model. We also discuss the various components of Bayesian Optimization, namely surrogates, acquisition functions, and losses. In the final part of this chapter, we discuss how to use sets to inject context into an ML model.

2.1 Black-box HPO

In a Black-Box HPO solution, the objective function f in equation 1.1 is treated as a blackbox. The problem is generalized to finding a global optimum of f. Some straightforward black-box HPO methods include Manual Search, Grid Search, and Random Search.

Manual Search in the HPO search space is feasible when we have expert knowledge of the problem and the given model. The idea is to select configurations step by step by observing the results obtained so we do not waste computation time evaluating similar configurations through intuition. This approach may be helpful for small models with lesser constraints. However, as the HPO search space becomes very large or conditional constraints become too complex, the selection of configurations becomes more and more difficult.

Hence a more systematic and automated approach is more practical.

Grid search is a more systematic approach to dividing the search space into grid points similar to those in a Euclidean graph. Let there be m dimensions in the search space $\mathbb{S}$. Let the selected number of values for each dimension be $n_1, n_2, ...n_m$. In the case of a discrete variable, the selected values will be a subset of the possible values, whereas, in the case of a continuous variable, we need to select the values based on a selected granularity. The cross-product of the selected subsets gives us the configurations to be evaluated. Hence, the number of configurations to evaluate will be $n_1 * n_2 * ...n_m$. The number of configurations we need to evaluate in this approach becomes a big issue for this method as the dimensions of the search spaces increase. Hence this approach becomes intractability for large search spaces.

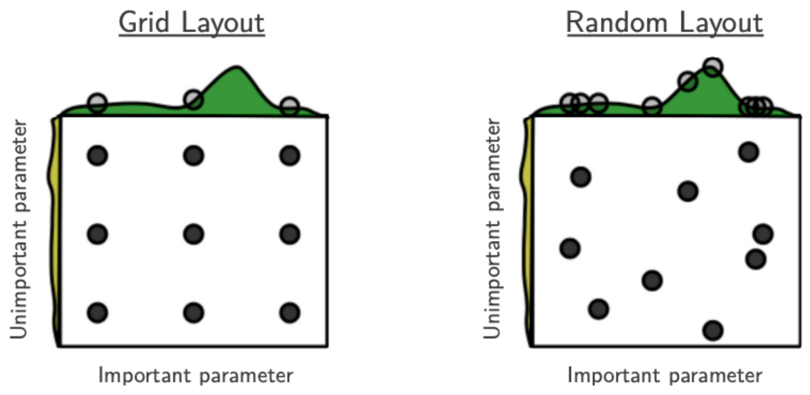

Figure 2.1: Illustrates Grid search and Random search in the case where 2 parameters are not equally important. Adpated from [3].

One issue with the Grid Search approach is that we assume that all dimensions in the HPO search space are equally important. It is not the case in many HPO problems. The Grid layout in Figure 2.1(left) illustrates this. For example, the learning rate in deep neural networks is much more important than many other parameters. If dimension p is the most important in the search space, then it makes sense to evaluate more values of p. Random Search helps us solve this problem. The Random layout in Figure 2.1(right) illustrates this. Hence Random Search can be used as a trivial baseline for comparing HPO solutions.

One advantage of these methods is that there are no restrictions on the HPO search spaces. Hence, they are suitable for any HPO problem at hand. On the other hand, these methods are non-probabilistic. Hence they cannot deal with noisy evaluations of the HPO configurations well. Moreover, these methods are computationally expensive. The reason is that they do not use surrogate evaluators and train and evaluate the whole model. Also, these search methods give us optimal HPO configurations only by chance.

2.2 Online HPO

The idea of an online HPO approach is that it tries to evaluate and update the HP configuration during the ML model's training. That is to say, the model's parameters and the hyper-parameters are updated jointly. If the hyper-parameters and the model parameters are learned disjointly in an HPO solution, such a solution is called an offline HPO.

Online HPO can be used to learn a single hyperparameter or all the hyperparameters. For example, Baydin et al. [1] propose the online learning of only the learning rate. They target this hyper parameter because it is the single most important one. On the other hand, Luketina et al. [28] propose the interleaved updating of all the hyperparameters with the ML model's parameters.

An example of offline HPO is the solution proposed by Franceschi et al. [12]. They formulate the whole HPO problem as a bi-level optimization problem [19]. A similar approach is used by Maclaurin et al. [29]. They use a relatively cheap method to obtain hyper gradients, i.e., gradients of hyper parameters for the whole objective function f. They use these gradients to update the hyperparameters, similar to any gradient-based optimization method. Notice that we must completely optimize the ML parameters in both these cases to update the hyperparameters once. This property of disjoint updating of the parameters and hyperparameters makes these approaches offline.

2.3 Multi-fidelity HPO

If we treat our HPO objective function f as a black box function, we would need to evaluate equation 1.1 fully. This evaluation is prohibitively expensive as evaluating a single HP configuration may take days [18]. Multi-fidelity hyperparameter optimization tries to solve this problem by evaluating the HP candidates on cheap functions f_{approx} that approximate the objective function f.

Here, f_{approx} is called fidelity as it reproduces the actual objective function to some degree. For example, a fidelity could be an evaluation of the HP configuration on only a subset of data, the evaluation of the HP on downscaled image data, or learning only for a few epochs. The idea is to trade-off between the performance of the approximate function and its optimization accuracy such that we get the best selection of HP using the least compute power. Since we do not do the actual evaluation of HP configuration using f, it is an approximate optimization technique.

The technique does not belong to the black box HPO domain. It can be understood if we consider the "few epochs" fidelity. The optimization algorithm looks into the training process and learns the objective function to exit it earlier. It thus gets feedback from within the "black box" of the HPO objective function.

This technique is named multi-fidelity because it may use different fidelities within the optimization process to get the best HP configuration. An example of the multi-fidelity technique is successive halving, described by Jamieson et al. [20]. While using successive halving, one can start with a given "budget fidelity," for example - a defined number of epochs or a defined amount of training time. The budget doubles at each step of the optimization process, and we eliminate the 50% of the worst-performing HP configurations for each step. Hence it uses "multiple" fidelities during the optimization process.

2.4 Transfer-learning for HPO

The HPO types discussed so far do every HP optimization from scratch. These types have no prior knowledge before starting any HP optimization. In addition to being computationally inefficient, it is contrary to how humans learn. Humans use prior experience that they have accumulated and condition their actions on this knowledge. Any machine learning method that uses this concept of conditioning its output on previously seen data is called a transfer learning method [45].

The concept of transfer learning can be utilized to accelerate HPO. For example, Gomes et al. [15] propose to meta-learn a set of exemplary HP configurations for the SVM. They propose that if we learn HP configurations that previously worked well, there is a high chance that a new SVM model also works well with these parameters. Reif et al. [38] also proposed a similar concept for the HP optimization of genetic algorithms. In this thesis, we compare the proposed model's performance with the performance of state-of-the-art transfer HPO methods like Two-Stage Transfer Surrogate (TST) [49], Ranking-Weighted Gaussian Process Ensemble (RGPE) [10], Transfer Acquisition Function Framework (TAF) [50], and Few Shot Bayesian Optimization

(FSBO) [48]. The TST method comprises two stages. In the first stage, multiple surrogates are learned for each metadata (or task). The outputs of the learned surrogates are combined in the second stage to rank the HP configurations. These two stages are used with the SMBO algorithm to do HP optimizations. We have discussed Bayesian Optimizations and SMBOs in detail in section 2.5. RGPE is very similar to TST because it also uses multiple models (Gaussian Processes in this case) learned for each task. One significant difference among many others between the two is that TST uses Nadaraya-Watson kernel-weighting to combine the surrogate outputs, whereas RGPE uses a linear combination of Gaussians instead. Wistuba et al. use a different method to do transfer HPO in their proposed model called TAF. They use the knowledge transfer mechanism during the acquisition phase of the SMBO process instead of using surrogates to transfer knowledge. FSBO is a transfer HPO solution that gives a state-of-the-art performance. Hence, we discuss this in detail in the following chapters.

On the other hand, any method that does not have prior knowledge before starting an HP optimization is called a non-transfer HPO method. We also compare our proposed method with some state-of-the-art non-transfer techniques in this thesis. These include Gaussian Processes (GP), Deep Networks for Global Optimization (DNGO) [40], Bayesian Optimization with Hamiltonian Monte Carlo Artificial Neural Networks (BOHAMIANN) [41], and Deep Kernel GP (DGP) [48]. GPs are standard models used in Bayesian Optimizations for the HPO problem. In addition to being used in a silo, they are also used as sub-components in many different models. We have discussed the working of GPs in detail in section 2.5.1. In DNGO, Snoek et al. propose using neural networks instead of GPs as surrogates in the Bayesian optimization. In BOHAMIANN, Springenberg et al. propose using neural networks as surrogates like in the case of DNGO. However, they employ the Bayesian Neural Network Regression Concept for doing the Bayesian optimization. DGP is a non-transfer version of FSBO where the Deep kernel surrogate is not meta-learned but directly used in the HP optimization. No meta-training or knowledge transfer is done in any of these methods. Hence they are classified as non-transfer HPO methods.

There are two broad ways to transfer knowledge for HP optimization: doing warm starting of initial configurations or learning surrogates.

2.4.1 Warm starting

Before running any HPO method, typically, experts study the data used to train the given model. They suggest a few initial HP configurations that have worked well for similar datasets, according to their experience. These initial configurations are evaluated for the given model, and the evaluated values act as a starting point (or hinge) for the HPO method. The idea of warm starting is to automate the selection of the initial configurations before the HPO optimization runs.

Warm starting was used by Gomes et al. [15], and Reif et al. [38]. In their respective papers, the authors used the meta-learned HP configurations as starting points for finding the optima in the HP response surface. This idea was also proposed for the SMBO optimization algorithm by Feurer et al. [11]

2.4.2 Meta-learning of surrogates

The idea of meta-learning a surrogate is to meta-train it using meta-data from previously optimized similar tasks. After its training, the surrogate can be used in an algorithm like SMBO to predict good-performing HP configurations. This surrogate's prediction of good-performing HP configurations is conditioned on the training meta-data. This pre-conditioning makes it a transfer HPO. The methods proposed by Schilling et al. [39] and by Feurer et al. [10] are a couple of examples among many others that use meta-learning of surrogates to do knowledge transfer.

The meta-learned surrogate can be used in two ways during the HPO optimization procedure. One could use it without modification during the HPO. On the other hand, the surrogate can be further meta-trained (finetuned) using the available metadata of the target task. This thesis proposes a transfer HPO method that uses meta-learning of surrogates for knowledge transfer. We study both the finetuned and non-finetuned versions of our model.

2.5 Bayesian Optimization

Bayesian Optimization (BO) is a technique used to find an optimum of any computationally costly and noisy objective function. It does this by building a model of the objective function that is cheap to evaluate computationally. This cheap model is called a surrogate function. When BO is used in the HPO domain, the objective function would be the HPO objective function f. The BO surrogate is referred to as an HPO surrogate.

BO uses known objective function evaluations as data to build the surrogate model. The data consists of a list of pairs of the form $(x, f(x))$. In the context of HPO, x refers to an HP configuration, and $f(x)$ refers to the true evaluations of the configuration. The surrogate model used is a probabilistic model. Hence, it also learns about the noise in the evaluations of the objective function. The core procedure of the BO in the HPO domain is the following:

1. From known data $D = (x_1, f(x_1)), (x_2, f(x_2)), (x_3, f(x_3))....$, build a probabilistic model that learns the mean and variance of the objective function

2. Use the surrogate to sample the next best HP configuration x' using a function known as acquisition function. Evaluate $f(x')$.

3. Append $(x', f(x'))$ to D and repeat the process.

The above process repeats till the computational resources are finished or we find an acceptable HP configuration. This procedure is also called SMBO (Sequential model-based Optimization). The procedure alternates between collecting data and fitting the surrogate model with the collected data [17]. Hence, there are two essential components of Bayesian Optimization:

- A probabilistic surrogate model of the objective function.

- An acquisition function.

Both these components are discussed in the following sections.

2.5.1 Surrogate models for Bayesian Optimization

HPO using Sequential Model Based Optimization (SMBO) [2] is a powerful and convenient method proposed in the literature. However, the performance of this method is heavily reliant on how well the surrogate models the true HPO objective function. In this section, we discuss some of the powerful models that may be used as surrogates.

Gaussian Processes

Gaussian processes [44] are predictive machine learning models that work well with few data points (or data pairs). They are inherently capable of modeling uncertainty. Hence, they are used widely in HPO problem settings, where uncertainty estimation is essential. Due to the robustness of the Gaussian processes, we use it as one of the baseline HPO surrogates in our thesis. In addition, many state-of-the-art HPO methods like FSBO, TST, DGP, and RGPE use Gaussian processes in one or more components. In this section, we explain the Gaussian process intuitively.

In order to completely grasp the concept of Gaussian processes, we need to understand normal (Gaussian) distributions. Consider a scalar random variable X distributed normally around a mean μ with a variance of σ^2. The following equation defines the probability density function (PDF) of X:

$$P_X(x) = \frac{1}{\sqrt[2]{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Here, X represents the random variable, and x represents an instance of the variable [44]. In this case, the mean μ, variance σ^2, and any sample x are all scalars.

If the random variable $\mathbf{X}$ is a vector in $\mathbb{R}^d$ where $d \in I^+$, then each component of the vector can be considered as a random variable. In this case the mean $\boldsymbol{\mu} \in \mathbb{R}^d$ whereas the variance is a matrix $\Sigma \in R^{d\times d}$. The variance is a matrix because the variance of any vector-valued random variable $\mathbf{X}$ should contain the following two types of values:

- Variance of a vector component w.r.t itself. This is represented by the d diagonal values of the matrix.

- Variance of each vector component w.r.t all other components. These variances are represented by the upper/lower triangular values of the matrix.

This matrix Σ, also known as the covariance matrix, has all values necessary to represent the variance of any vector-valued random variable.

The Probability Density Function (PDF) of a vector valued variable $\mathbf{X} \in \mathbb{R}^d$ with a mean $\boldsymbol{\mu}$ and covariance matrix Σ is given by [30]:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma|^{\frac{d}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

This equation defines the PDF of a multivariate normal distribution.

The core idea in the Gaussian processes is that functions can be considered vectors of infinite dimensions. Consider any function f that has a domain $\mathbb{R}$. If f is considered to be a vector in $\mathbb{R}^\infty$, then each point $i \in \mathbb{R}$ can be represented by a component f_i of the function f. Hence, a function is nothing but a sample from $\mathbb{R}^\infty$.

The idea of Gaussian processes is to sample smooth functions from $\mathbb{R}^\infty$. In any smooth function f, if any point g is close to x in the domain of f, then $f(g) \approx f(x)$. The following equation mathematically represents this idea:

$$\lim_{\delta x \to 0} f_{x+\delta x} \approx f_x^+ \text{ and } \lim_{\delta x \to 0} f_{x-\delta x} \approx f_x^-$$

$$\text{where } \delta x > 0 \text{ and } x, \delta x \in \mathbb{R}$$

The above definition is nothing but the definition of a smooth function in terms of vector notation. Moreover, nearby components of f "vary" similarly w.r.t each other. These properties can be naturally encoded using a covariance matrix. Hence, we obtain smooth functions if we sample them from a multivariate normal distribution with the required covariance matrix. The Gaussian process restricts the function sample space to a multivariate normal distribution.

The similarity between 2 points in a domain is defined by a function called **kernel** in Gaussian processes. The values in the required covariance matrix are populated using this kernel function. The Gaussian process kernel controls the sampled function's smoothness f. Formally kernel k is defined as,

$$k(\mathbf{x}, \mathbf{x'}) \mapsto \mathbb{R}$$

Here, $\mathbf{x}, \mathbf{x'}$ belong to a domain in the most abstract sense. For example, when the input domain is a euclidean space, $\mathbf{x} \in \mathbb{R}^{\mathbb{I}^+}$. Some well known kernels are Radial Basis Function Kernel, Matern Kernel, and Periodic Kernel.

Finally, a Gaussian Process specifies that any new observation y^* for input $\mathbf{x}^*$, is jointly normally distributed with known observations $\mathbf{y}$ such that

$$Pr\left(\begin{bmatrix}\mathbf{y} \\ y^*\end{bmatrix}\right) = \mathcal{N}\left(m(\mathbf{X}), \mathbf{\Sigma}\right) \tag{2.1}$$

Here, $m(\mathbf{X})$ is the mean of the vectors which is commonly taken as $\mathbf{0}$. $\mathbf{\Sigma}$ is the covariance matrix defined as

$$\mathbf{\Sigma} = \begin{bmatrix}\mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**}\end{bmatrix}$$

Where $\mathbf{K} = k(\mathbf{X}, \mathbf{X})$, $\mathbf{K}_* = k(\mathbf{X}, \mathbf{x}_*)$ and $\mathbf{K}_{**} = k(\mathbf{x}_*, \mathbf{x}_*)$ for any given kernel k [44].

Random Regression Forest

Random regression forests are an ensemble of regression trees. When it is used as a surrogate in the SMBO procedure [17], its training is done using the known HP configurations and their evaluations. Since this model is an ensemble, predicting the mean and variance of the surrogate is trivial. The mean of the surrogate is the mean of the predictions of all the trained trees. Similarly, the variance of the surrogate is the variance of their predictions.

The following are the two main advantages of using this model.

- It can handle both continuous and discrete variables trivially without any modifications to the model. This means that the HP search space need not be restricted.

- It can handle conditional variables more efficiently as compared to the Gaussian surrogate.

Both these advantages are limitations of the Gaussian surrogates. Hence the random regression forests can be used as surrogates where the Gaussian process cannot.

Deep Neural Networks

Deep Neural Networks (DNNs) are machine learning models consisting of a network of multiple neuron layers. In its most basic form, the DNN network has every neuron of one layer connected to every neuron of the following layer. Hence each layer is called a fully connected layer (FC for short), and the neural network is called a fully connected deep neural network. Figure 2.2 illustrates an example of a fully connected deep neural network. DNNs can be viewed as models that pass inputs sequentially through multiple computation functions before getting the final output.

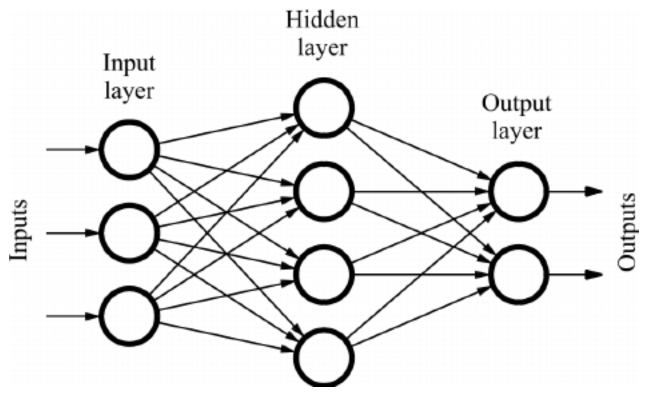

Figure 2.2: A sample of a fully connected deep neural network [37].

Let us denote I as the input to the fully connected DNN and O as its output. Let the interacting neuron layers, i.e., the input layer, the hidden layer, and the output layer, be represented by functions f_i, f_h, and f_o, respectively. Then the computation of this DNN can be abstractly represented by

$$O = f_o(f_h(f_i(I))) \tag{2.2}$$

And each function f can be represented by the equation

$$f(I) = \textsc{NonLinearFunction}(\mathbf{W} * I + \mathbf{b}) \tag{2.3}$$

where the matrix $\mathbf{W}$ holds the weights of the connections to the neurons of the current layer and $\mathbf{b}$ represents some bias that is added to the outputs of the neurons.

The use of NonLinearFunction makes it possible for the neural network to represent non-linear functions. The overall capacity of DNNs is very high. It can be further increased by increasing the depth of the network or by increasing the neurons at each layer. One disadvantage of DNNs is that they cannot model uncertainty trivially. One method to represent uncertainty in DNNs is to use an ensemble of neural networks or a specialized loss function with two outputs. Since uncertainty modeling is essential for this thesis, it is discussed in more detail in section 3.7. In addition to fully connected DNNs, there are neural networks with sophisticated architectures like convolutional neural networks and recurrent neural networks. However, we do not use these in our thesis; hence we do not describe them here.

Bayesian Neural Networks

Bayesian Neural Networks are a particular type of neural network in which the output of any layer in the neural network is stochastic and not a point estimate like in the case of classical neural networks. These types of neural networks are called Bayesian Neural Networks (BNN) because they are trained using Bayesian inference mechanism [22]. BNNs can be used to quantify uncertainty directly because of their stochastic property.

In BNNs, stochasticity is added either by using a stochastic activation function or by using stochastic weights and biases. Consider the case where we model a BNN using stochastic weights and biases. Let us represent all weights and biases by a parameter vector θ. First, we define a probability distribution of θ given by $p(\theta)$. This is the prior or a hypothesis in the Bayesian jargon. For example, one prior could be a multivariate Gaussian distribution with a defined mean and a covariance matrix. Given the training data D, we calculate the new probability distribution to get the posterior distribution $p(\theta|D)$ using Bayes theorem. Here D is also referred to as evidence in Bayesian terms. We do not mention the intricacies of the BNN training procedure as it is out of scope for this thesis.

Let a non-stochastic neural network be represented by f_θ for a definite parameter vector θ. The uncertainty calculation in BNNs is done using the following equations

$$\begin{aligned} \theta_i &\sim p(\theta|D) \\ y_i &= f_{\theta_i}(x) \\ where \quad i &\in \mathbb{I}^+ \end{aligned}$$

First, we sample a set of parameters from the posterior probability distribution $p(\theta|D)$. Using this set of parameters, we calculate multiple values of y for the given input, say x. Hence we obtain multiple output values for a single input x. One can use these multiple output values to calculate the output's mean and variance. Note that if the output of the Bayesian Neural Network is a vector, we would need to calculate the covariance matrix to get the multivariate variance.

Even though BNNs are theoretically very robust, they are practically difficult to train. Firstly, the exact estimation of the posterior is computationally intractable. Hence approximate methods like Monte Carlo Markov chains and variational inference methods are used to train the BNNs. Even these approximations are computationally more expensive than the classical artificial neural network. Moreover, it is unclear from the beginning what prior probability distribution should be used for the stochastic parameters. Because of these disadvantages, we do not use them as surrogates in our thesis.

2.5.2 Learning the surrogates

In section 2.5.1 we discussed the details of various surrogate models that can be used in the Bayesian Optimization process. These surrogates must be learned using available $(x, f(x))$ data pairs regardless of using them in a transfer or non-transfer HPO method. Surrogate learning is formulated as an optimization problem in which a loss function is minimized to get the trained surrogate. The type of loss function used will differ based on the problem formulation. For example, if we formulate the HPO problem as a regression problem in which the surrogate would learn the HPO objective function's output, we could use regression losses like RMSE (Root Mean Squared Error).

In this thesis, we study and compare the performance of different loss functions. We use the negative log-likelihood loss function to learn the Deep Ensemble surrogate [24]. This loss L_{de} is given by:

$$L_{de} = -\log p(y|x) \tag{2.4}$$

where x is the input HP configuration and y signifies the HPO objective function's output. In FSBO [48], the loss function is similar, with the only distinction being that it uses a negative log **marginal** likelihood. Our proposed method sees the surrogate as a ranker in the context of SMBO. Hence, we use ranking losses to learn the surrogate in our thesis. In the next section, we discuss the ranking losses in more detail.

Ranking Losses

Ranking losses can be broadly classified into the following types [7]:

- Pointwise ranking losses
- Pairwise ranking losses
- Listwise ranking losses

In pointwise ranking losses, the loss function views the ranking problem as a problem of assigning a label to each queried data point. Hence, each learning instance is a single object. For example, Li et al. [27] use pointwise ranking losses for their proposed model called McRank. They formulate the ranking problem as a multilevel classification problem. Each data point is classified independently using soft classification. The score of an object then is its expected rank. Similarly, Cossock et al. [8] use pointwise ranking losses for subset regression.

In pairwise ranking losses, the loss function's input is a pair of objects. The loss function tries to separate the input data points as much as possible in the output space by minimizing the pairwise classification error [9]. Examples of models built using pairwise loss functions are Rankboost [13] and RankNet [4].

In listwise ranking losses, a single learning instance is the whole input set, i.e., all the objects in the input set. This makes an intuitive sense because we cannot rank the objects independently as an object's rank is relative to the ranks of other objects in a set. This is the fundamental advantage of listwise ranking losses compared to the pointwise and pairwise losses. Moreover, it has been shown by Cao et al. [5] that training a model with listwise losses gives us a superior model compared to training with pointwise or pairwise losses. The 2 most prominent listwise loss functions are ListNet [5] and ListMLE [51]. We discuss and analyze these loss functions in detail in chapter 3.

2.5.3 Acquisition functions

Acquisition functions are used in each step of BO to "acquire" a good-performing HP configuration. During BO, they need to balance exploiting information from the evaluated HP configurations and exploring unknown configurations in the HP search space. The following functions are some of the most prominent acquisition functions found in the literature [44]

- **Upper Confidence Bound (UCB)**: It returns the best possible hyperparameter configuration using a linear combination of the mean and the standard deviation.
- **Probability of Improvement**: It returns the probability of getting a better hyperparameter configuration than the incumbent best configuration.
- **Expected Improvement**: Given a Gaussian distribution at a new input point, it finds the expectation of improvement i.e $(f(x) - f_{max})$ over the part of normal that is greater than f_{max}. This acquisition function is used with all the baselines in this thesis. However, it is not used with the proposed method. The reason for this decision is explained in more detail in section 4.1.2.

2.6 Set-Modelling with Neural Networks

In the previous sections, we discussed various types of surrogates that can transfer knowledge in transfer HPO methods. However, training on data from known tasks and using this trained model for unknown tasks has a conceptual shortcoming. Humans do not pre-condition their actions on new tasks only on previous experiences. Their pre-conditioning also includes the knowledge of the current task (however slight it may be).

To model this concept, one has to learn to represent and pre-condition knowledge from known tasks in a model. Doing this makes the model context-aware. If we consider an HP configuration and its evaluation $(\mathbf{x}, y)$ as a single object, then the group of all the known pairs can be represented as a set. Here, $\mathbf{x}$ is a feature vector, and y is a scalar. Now our problem is a 2 stage process that includes

- Representation of a set.
- Conditioning of our model on this representation.

We use Deep Neural networks to represent a set in this thesis because it is a model with high representational capacity and is easy to train. Deep Sets by Zaheer et al. [52] and Set Transformers by lee et al. [26] are two of the exciting researches that we found in the literature. Since the sophisticated attention mechanisms used in Set Transformers were unnecessary, we chose to use the simpler architecture used in Deep Sets. The following section discusses these Deep Sets.

2.6.1 Deep Sets

Machine learning models learn functions of the following format:

$$f : \mathbb{R}^d \mapsto \mathbb{R}^k \quad d, k \in \mathbb{I}^+ \quad \text{For Regression}$$

$$f : \mathbb{R}^d \mapsto \{c_1, c_2, ...c_n\} \quad \text{For Multi-Classification}$$

The function f transforms objects from an input space to an output space. For the problem of set latent representation, the input space is a space containing Sets (i.e., each object in the space is a Set by itself). The output space, however, remains similar to the regression case for regression tasks and the classification case for classification tasks. Let $\mathbb{X}$ be the union of all possible sets in the inputs space. Then the set representation problem can be defined as:

$$g : 2^{\mathbb{X}} \mapsto R^k \quad k \in \mathbb{I}^+ \quad \text{For Regression}$$

$$g : 2^{\mathbb{X}} \mapsto \{c_1, c_2, ...c_n\} \quad \text{For Multi-Classification}$$

Here, $2^{\mathbb{X}}$ is the power set of $\mathbb{X}$.

Two critical constraints of this problem are:

- Permutation-Invariance constraint: The permutation of the objects within an input set should be irrelevant for the model g.
- Set cardinality invariance constraint: The cardinality of the set can be variable. Hence our model g should be invariant to the number of elements in the input set.

Figure 2.3 illustrates the architecture proposed by the Zaheer et al. [52]. We use a set of only three objects in the Figure for simplicity. The architecture independently passes all the three inputs through the deep neural network to get intermediate outputs I_1, I_2, and I_3. To solve the permutation invariance constraint, any mathematical operation that is commutative and associative is applied to the intermediate outputs. As long as the pooling operator is permutation invariant, the proposed architecture is also permutation invariant. Examples of such operators are sum, mean, and max. The mean pooling operator is better in our case because it also meets the Set cardinality constraint. Hence the pooling operation becomes:

$$\text{Pool}(I_1, I_2, I_3) = \frac{I_1 + I_2 + I_3}{3}$$

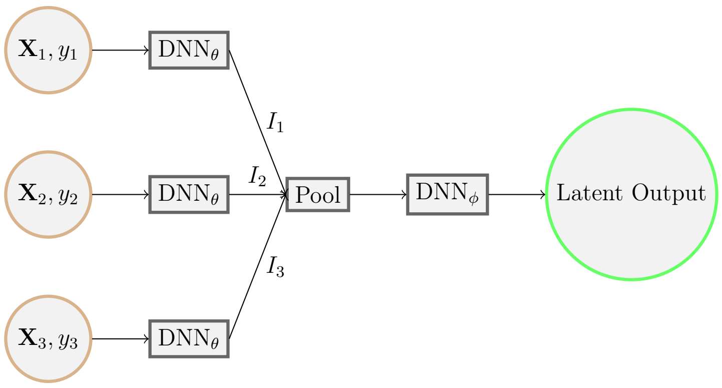

Figure 2.3: Skeletal of the architecture proposed in deep sets

The output of the pool operation is then passed through a different Deep Neural Network to get a latent output.

Chapter 3

Background

This chapter discusses the fundamental concepts one must grasp to understand the thesis work. We first talk about the concept of ranking. Then, we take the next step to define the ranking loss functions abstractly. After that, we discuss the concrete loss functions studied in this thesis. We also talk about how uncertainty is modeled using Deep Neural Network Ensembles. Lastly, we discuss the details of the baselines - Deep Ensembles and FSBO that we studied in this thesis.

3.1 Understanding Ranking

Consider a set $\mathbb{A} = \{a_1, a_2, a_3, ..., a_n\}$ where each object $a_i \in \mathbb{D}$ for some domain $\mathbb{D}$. Let us consider that an object with a lower rank is preferable to an object with a higher rank. Ranking the objects of set $\mathbb{A}$ is defined as ordering all objects $a_i \in \mathbb{A}$ such that

$$\texttt{Rank}(a_i) < \texttt{Rank}(a_j) \iff a_i \succ a_j$$

Here `Rank` of an object a_i is analogous to its position in the ordered list. An ML model can be used to model this task. In the most general case, the cardinality of the set $\mathbb{A}$ is not fixed. Practically building an ML model that takes a whole set of an unknown cardinality as input is not trivial. Hence, to keep the model practical, the task of ranking is divided into the following components: [36]

- Calculation of relevance scores of each $a_i \in \mathbb{A}$.
- Ordering or sorting of objects based on their relevance scores.

Doing a gradient-based optimization for an ML model that models both the above tasks is not possible. This is because sorting is non-differentiable, and we cannot get a gradient for the sorting functionality. Hence, to do a gradient-based optimization, we only model the calculation of relevance scores. We represent this model by s. We use the letter "s" to signify that we are calculating a "score" for an object. s is defined as

$$s : \mathbb{A} \mapsto \mathbb{R}$$

Given a set of objects to rank, first, the relevant scores of the objects are predicted by s. After that, using these relevant scores, the objects are sorted. Hence, one can use s to rank any newly given set of objects.

Learning a machine learning model requires optimizing criteria on the model's output. This output, in our case, is the sorted list of objects. The optimizing criteria in machine learning jargon are called a loss function. Hence, in our case, the loss function can be referred to as a *Ranking Loss*.

3.2 Understanding Ranking Loss functions

Consider data in the format shown in table 3.1 is given to us.

Instance	Object Set	Ground Truth
1	$\{a_1, a_2, a_3, ..., a_{10}\}$	$\{y_1, y_2, y_3, ..., y_{10}\}$
2	$\{a'_1, a'_2, a'_3, ..., a'_{15}\}$	$\{y'_1, y'_2, y'_3, ..., y'_{15}\}$
3	$\{a''_1, a''_2, a''_3, ..., a''_7\}$	$\{y''_1, y''_2, y''_3, ..., y''_7\}$
...	$\{...\}$	$\{...\}$

Table 3.1: Data used to train a scoring function.

Let each data point a_i be a sample/element from the set $\mathbb{A}$ and let y_i be their respective ground truth preferences. Normally the ground truth scores are given as real-valued numbers. Hence we assume that $y_i \in \mathbb{R}$. These preference scores are relative to the objects within $\mathbb{A}$. This means that the preference scores of two objects a_i and b_i belonging to 2 different sets $\mathbb{A}$ and $\mathbb{B}$ cannot be compared.

The goal is to learn the function s to predict the relevance score of any new object sampled from the set $\mathbb{A}$. If we parameterize function s with θ, we can use typical optimization techniques like gradient descent to obtain an optimum θ^* by minimizing the loss function. Hence, our goal is obtaining the function s_{θ^*}.

There are various types of ranking loss functions. The type of ranking loss defines its signature, i.e., what type of input it takes. This is because the learning instance (or an input instance) to a loss function is different for different ranking types. The output of any loss function studied in this thesis remains a scalar irrespective of its type.

We already know from Section 2.5.2 that ranking losses can be either pointwise, pairwise, or listwise. For pointwise loss functions, each learning instance is a single object. Its format hence can be defined as:

$$L_{\texttt{pointwise}} : s(\mathbb{A}) \times \mathbb{Y} \mapsto \mathbb{R} \tag{3.1}$$

Similarly, the pairwise loss function has each instance as a pair of objects and their corresponding ground truths. Its format can hence be defined as:

$$L_{\texttt{pairwise}} : s_1(\mathbb{A}) \times s_2(\mathbb{A}) \times \mathbb{Y}_1 \times \mathbb{Y}_2 \mapsto \mathbb{R} \tag{3.2}$$

One learning instance for listwise losses is a set of objects with corresponding ground truths. If we take any positive integer $n \leq |\mathbb{A}|$, the declaration of the listwise loss is given by

$$L_{\texttt{listwise}} : s_1(\mathbb{A}) \times s_2(\mathbb{A}) ... s_n(\mathbb{A}) \times \mathbb{Y}_1 \times \mathbb{Y}_2 \times ... \mathbb{Y}_n \mapsto \mathbb{R} \tag{3.3}$$

In the following sections, we analyze the loss functions studied in this thesis. As listwise loss functions are more advantageous conceptually than others, we discuss them in more detail.

3.3 Pointwise Loss Function: Subset Regression

We take the subset regression loss function proposed by Cossock et al. [8] as a reference for analyzing pointwise loss functions. We reused the implementation written by Pobrotyn et al. [35] for our analysis.

The basic idea of subset regression is to learn the rank of the objects directly and not have an extra step of using the output scores to determine the rank of objects. The subset regression loss function is similar to the RMSE (Root Mean Squared Error) loss function. Given a batch of size N containing objects a_i and their corresponding ground truth values y_i such that $0 \leq i \leq N$, the loss function is of the form

$$L_{\texttt{SubsetRegression}} = \frac{1}{N} \sum_{i=1}^{N} (s(a_i) - \texttt{level}(y_i))^2 \tag{3.4}$$

The `level` is given by the rank of y_i in the current batch. For example, consider we have 3 ground truth values $\{y_1 = 0.8, y_2 = 0.9, y_3 = 0.1\}$. If we consider that a higher y value is better and a lower rank/level is better then $\texttt{level}(y_1) = 2, \texttt{level}(y_2) = 1, \texttt{level}(y_3) = 3$.

To keep the implementation such that the score value is in the same given range as the regression value, Pobrotyn et al. [35] use a slightly modified version of the loss function, which is conceptually the same. This loss function is given by

$$L_{\texttt{SubsetRegression}} = \frac{1}{N} \sum_{i=1}^{N} (\texttt{distinct\_levels} * s(a_i) - y_i)^2 \tag{3.5}$$

Where `distinct_levels` is the total number of distinct ground truth values in the given batch. As we can see, the loss function does not directly make the scoring function learn the output regression value. This property is shared across all ranking loss functions.

3.4 Pairwise Loss function : RankNet

To analyze how pairwise loss functions are helpful for the HPO problem, we study the RankNet loss function proposed by Burges et al [4]. We reused the implementation written by Pobrotyn et al. [35] for our analysis.

As we know from table 3.1 we are always given a set of values and their corresponding relevance scores to train. Hence, we have $\{a_1, a_2, a_3..., a_n\}$ as inputs and $\{y_1, y_2, y_3..., y_n\}$ as their corresponding relevance scores. As the $L_{\texttt{pairwise}}$ takes only pairs of values we obtain first the scores of all objects to get $\{s(a_1), s(a_2), s(a_3)..., s(a_n)\}$. Then, we form all possible pairs of inputs to the loss function from the given scores of objects. For example, after forming the pairs, one instance of the loss function is of the form $\{s(a_1), s(a_2), y_1, y_2\}$.

Let the actual probability of $a_1 \succ a_2$ is given by P^*, and the predicted value of the same is given by P. Then the pairwise loss is nothing but the cross entropy loss given by

$$L_{\texttt{RankNet}} = \texttt{C.E.Loss} = -P^* \log P - (1 - P^*) \log(1 - P) \tag{3.6}$$

Where,

$$P^*(a_1 \succ a_2) = \begin{cases} 1 & y_1 \geq y_2 \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

$$P(a_1 \succ a_2) = \frac{e^{s(a_1)-s(a_2)}}{1 + e^{s(a_1)-s(a_2)}} \tag{3.8}$$

The pairwise loss function only tries to classify the input objects. This means we only need to know from relevance scores if one object is better than the other. Hence a binary cross-entropy loss function is used instead of a general one in the implementation. For the loss function, the order of the objects in the pair is irrelevant. So, instead of taking two pairs containing the same objects (but in a different order), only one pair is taken such that $s(a_1) - s(a_2) > 0$. This makes the implementation easier.

3.5 Listwise Loss function: ListMLE

In this thesis, we used the listwise loss function called ListMLE for our analysis. To understand ListMLE in-depth, we first talk about a listwise loss function called ListNET, which is its predecessor. After this, we will define and analyze the ListMLE loss function itself.

3.5.1 ListNET

The ListNet idea was proposed by Cao et al. [5]. In this section, We intuitively explain the mathematical equations that define this loss function.

Our objective is to learn the scoring function s such that it returns scores that are similar in relevance (or) order when compared to the ground truth scores. Let us assume we learn the scoring function s such that

$$y_3 < y_{12} < y_1 \implies s(a_3) < s(a_{12}) < s(a_1)$$

The ranking of objects is obtained by sorting them based on their respective scores. Here, sorting the objects based on the scores returned by s would return the same ranking compared to the ranking obtained by sorting the objects using the ground truth values. Note that we do not need to get the exact ground truth scores. This increases the target function space, and more than one function gives the correct results. A bigger target function space makes learning a suitable scoring function easier. Another notable point is that the sort functionality is non-differentiable; hence it is not a part of the ranking loss function.

Our loss function needs to be constructed using the following two lists:

- List of scores given by scoring function s.

- List of scores given to us by the ground truth.

If the loss function returns a distance between the two given lists, learning the function s amounts to reducing this distance. Hence finding an optimum is to find the minimum distance between the two lists by changing the parameters of the scoring function s.

In ListNET, a probabilistic approach is adopted to calculate the distance between the two lists. This accounts for any uncertainties in the ground truth values. The probability of selecting an object from the input set is given by

$$P = \frac{s(a)}{\Sigma_i s(a_i)} \quad \forall i \in \{1, 2, 3, ..., n\} \tag{3.9}$$

where n is the list size. Equation 3.9 makes intuitive sense because the probability of selecting an object should be higher if it is more relevant and vise-versa. Note that the score of any object given by the scoring function can also be negative. Therefore the score is passed through a strictly positive and increasing function ϕ. This changes the probability to

$$P = \frac{\phi(s(a))}{\Sigma_i \phi(s(a_i))} \quad \forall i \in \{1, 2, 3, ..., n\} \tag{3.10}$$

Equation 3.10 is also referred to as top 1 probability of an object by Cao et al. [5]. This is because this gives the probability of ranking the object first when we are calculating the permutation probability of given list.

The proposed method to find the distance between 2 lists in ListNet is

1. Find the top 1 probabilities of each object using the scores given by the scoring function.

2. Using the ground truth values, find similar top 1 probabilities.

3. Since the list of top 1 probabilities forms a probability distribution, the cross-entropy between the two top 1 probability lists is taken as the "distance" metric.

Let $P_{s(a)}$ represent the top 1 probability of an object using the scores given by the scoring function. Similarly, let P_y represent the top 1 probability using its ground truth value. The cross entropy used as a loss in ListNet is given by

$$L(\mathbf{y}, s(\mathbf{a})) = -\Sigma_i P_{s(a_i)} \log P_{y_i} \tag{3.11}$$

Where $1 \leq i \leq n; i \in \mathbb{I}^+$ and $\mathbf{y}$ & $s(\mathbf{a})$ represent the ground truth values and the scores given by the scoring function respectively.

3.5.2 ListMLE

The ListMLE loss stands for "List Maximum Likelihood Estimation" loss. As in ListNet, the probability of selecting an object from the list is the same as given in equation 3.10. However, the final loss used in ListMLE is not cross-entropy. Instead, it maximizes a likelihood estimation, as the name suggests.

Let π define any permutation of a list. The probability of a permutation is nothing but the probability of selecting objects from the list without replacement. In our case, the permutation probability of selecting one permutation using the selection probabilities given by equation 3.10 is

$$P_\pi = \prod_{j=1}^{k} \frac{\phi(s(\pi_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi_k))} \tag{3.12}$$

where π_i is the object at position i in the permutation π.

Applying log to the above equation gives us

$$\log P_\pi = \sum_{j=1}^{k} \log \frac{\phi(s(\pi_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi_k))} \tag{3.13}$$

However, the question remains which permutation to use? The best permutation for the given set of objects would be according to the actual relevance scores of the objects. More precisely, it would be the objects ordered in the descending order of their relevance scores. Let this permutation be represented by π^*. Hence our probability equation becomes

$$\log P_{\pi^*} = \sum_{j=1}^{k} \log \frac{\phi(s(\pi^*_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi^*_k))} \tag{3.14}$$

The ListMLE loss function maximizes this probability. Since we generally minimize the objective function, the loss function is given by

$$L_{mle} = -\log P_{\pi^*} \tag{3.15}$$

Expanding the right-hand side of the equation gives us the final loss function that must be minimized. This is the general form of the loss function proposed by Xia et al. [51].

$$L_{mle} = -\sum_{j=1}^{k} \log \frac{\phi(s(\pi^*_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi^*_k))} \tag{3.16}$$

Notice that the objects' ground truth score values are unused in the loss calculation. This means that the absolute predicted scores of the objects do not matter. The only constraint is that the scores must have the correct relative values and that using these values to sort the list should result in the correct ordering of objects. Even non-linearly scaling the actual scores does not affect the output as long as the constraint is maintained.

In the case of ListNet, however, the actual scores do matter. The probability of selecting objects depends on their ground truth scores. Hence the result is invariant only to linear scaling. This advantage makes the ListMLE loss function superior to the ListNet loss. The ListMLE function has a larger target function space, making convergence easier during optimization. Because of this advantage, we use ListMLE as a list loss function in our thesis.

3.6 Position Enhanced Ranking

In many problem domains that use the ranking concept, it may not be essential that each object be placed at an exact location as induced by its relevance. For example, when a search engine ranks its search results, it is more important to find the most important results and rank them correctly than order the least important results correctly. This is also the case for the problem of ranking HP configurations in the SMBO process. During the SMBO process, the ranking surrogate is only needed to obtain the most important HP configuration at each step in the optimization.

Lan et al. [25] discuss this problem in their paper, "Position-Aware ListMLE: A Sequential Learning Process for Ranking". However, they reformulate the problem as a sequential learning process. A more accessible approach is to weight each object component in our ListMLE by any decreasing function c as proposed by Chen et al. [6]. This is possible because the ListMLE loss function is in the form of a summation.

Hence the weighted ListMLE function is given by:

$$L_{mle} = -\sum_{j=1}^{k} c(j) \log \frac{\phi(s(\pi_j^*))}{\sum\limits_{t=j}^{k} \phi(s(\pi_k^*))} \tag{3.17}$$

Where $c(j)$ gives the weight of the rank j in the ordered list. This is the approach used in our model to improve our ranking loss function. The type of decreasing function used is discussed in more detail in chapter 4. Note that it is also possible to use the weighting in the ListNet case as ListNET and ListMLE have similar forms.

3.7 Uncertainty modelling using Deep Ensembles

Deep Neural Networks (DNNs) are machine learning models with very high representational capacity [16]. Due to this property, one can use them as surrogates for HPO objective functions. However, the issue is that DNNs do not quantify uncertainty trivially. They predict results overconfidently, and the results may be wrong. If any HPO optimization algorithm used a DNN as a surrogate, its overconfident wrong predictions could cause much computational overhead by predicting inefficient HP configurations. This becomes even more crucial in an optimization technique like SMBO because one inefficient configuration selection might have a cascading effect of selecting further inefficient configurations. As DNNs are used as scoring functions in our proposed method, and we use the SMBO algorithm for the HP optimization, we must study how to model uncertainty efficiently using them.

In the current literature, uncertainty quantification methods using deep neural networks can be broadly classified into the following methods:

- Bayesian Neural networks [14].
- Ensemble Approach using Monte Carlo drop out [42].
- Ensemble approaches using multiple neural networks.

In a Bayesian neural network(BNN), a prior over the network parameters (Weights and Biases) is specified during the initialization of the BNN. Given the data, a posterior predictive distribution is calculated for all the network parameters. One issue with this approach is that BNNs are very complex and challenging to train.

Monte Carlo dropout is a regularization technique used during the training of DNNs. With a certain probability, connections between neurons are dropped. Using this dropping mechanism, one could obtain possibly 2^N neural networks where N is the number of connections in the neural network. We thus get an ensemble of high-capacity models for free. If we drop the connections during training, our model can be regularized well. However, using Monte Carlo dropout during the evaluation, we can get multiple results from the same input. Lets say we are given an input x, we can obtain an output $y = \text{DNN}(x)$. If we have m neural networks obtained using Monte Carlo dropout, we get $\{y_1, y_2 ... y_m\}$ outputs. We can calculate the mean and variance of the outputs by using the following equations

$$y_{\text{mean}} = \frac{\sum_{i=1}^{m} y_i}{m} \qquad y_{\text{variance}} = \frac{\sum_{i=1}^{m} (y_i - y_{mean})^2}{m-1} \tag{3.18}$$

Please note that the $m-1$ in the denominator is due to Bessel's Correction [46] to reduce the bias in estimation.

In an ensemble of multiple neural networks, we have separate neural networks instead of using a mechanism to get new neural networks from one network (like in the case of Monte Carlo). These neural networks are trained separately or together, depending on the architecture of the ensemble. If we have m separate neural networks, we would have m outputs similar to the Monte Carlo approach discussed above. Hence we could use the Equation 3.18 to calculate the ensemble's mean and variance for this approach.

Lakshminarayana et al. [24] propose another method to predict uncertainty using deep neural networks. They propose that the uncertainty prediction can be made directly using a single neural network. This is possible if we assume that the underlying uncertainty is a Gaussian distribution. With this assumption, the neural network would have two outputs instead of one. One output for the mean of the prediction, say μ, and the other for its variance, say σ^2. One important point to note is that the variance cannot be negative. The authors ensure this by passing the neural network's output through a strictly positive "soft plus" function.

The authors use the following loss function for the optimization.

$$L_{de} = \frac{\log \sigma^2}{2} + \frac{(y-\mu)^2}{2\sigma^2} + k$$

Where both the mean and the variance is given by $\mu = g(\theta, \mathbf{x})$ and $\sigma^2 = f(\theta, \mathbf{x})$ respectively. As there are multiple neural networks, each predicting its Gaussian distribution, there needs to be a mechanism to integrate the results. This is done using a mixture of Gaussian Distributions. If there are m neural networks in the ensemble (each with its own mean and variance), the final mean and variance are given by

$$\mu_{final} = \frac{\sum_{i=1}^{m} \mu_i}{m} \quad \text{and} \quad \sigma^2_{final} = \frac{\sum_{i=1}^{m} (\sigma_i^2 + \mu_i^2) - \mu^2_{final}}{m}$$

We use the loss function proposed by Lakshminarayana et al. [24] to build a deep ensemble baseline surrogate in this thesis. However, the prediction of uncertainty using the proposed ranking loss surrogates is made using the simple ensemble approach given in equation 3.18. This is because the integration of a loss function that learns both the mean and variance with the ranking loss functions is non-trivial. To integrate these two concepts, one must do a thorough theoretical analysis which is out of this thesis's scope.

3.8 Baselines

In this thesis, 2 HPO techniques were implemented before studying the proposed model - Deep Ensembles (proposed by Lakshminarayana et al. [24]) and Few Shot Bayesian Optimization (FSBO).

Deep Ensembles (DE) were studied with two main objectives in mind. First, to study how uncertainty is estimated using Deep Neural Networks. This was a prerequisite to understanding how to implement uncertainty in our proposed model as we use deep neural networks as a scorer in our model. The second objective was to understand how a non-transfer technique would work for the HP optimization problem. It is also possible to make the Deep Ensemble surrogate a transfer technique by meta-training it before using it in the HP optimization. However, we did not do this in our thesis.

The second technique implemented was FSBO. We chose this because this gave the state-of-the-art results on the HPO-B benchmark. The HPO-B benchmark is discussed later in Chapter 5. Studying FSBO also gave us an idea of implementing the transfer HPO mechanism in our proposed ranking loss surrogate model. In addition, we used Random search and GP as standard baselines for result comparison.

Both these FSBO and DE baselines have built-in capability for uncertainty estimation. Hence we use an acquisition function called Expected Improvement when using these baselines in the SMBO algorithm. This is because it uses uncertainty to its advantage to suggest the best HP configurations. Furthermore, it has advantages over other acquisition functions [21]. The following two sections discuss the details of the baselines methods used in our thesis.

3.8.1 Deep Ensemble

As previously mentioned, Deep Ensembles were implemented as non-transfer HPO surrogates. Hence, there was no meta-training done for them. Consequently, the usage of DE as a surrogate was quite similar to that of Gaussian processes. Algorithm 1 shows how they were used in the SMBO process.

Pineda et al. [33] implemented the infrastructure for manipulating HP configurations. We used this infrastructure to test the baseline and the proposed surrogates with their respective acquisition functions. In Algorithm 1 each time a new or best HP configuration is selected and evaluated, an HP optimization step (or optimization cycle) completes. In the rest of this report, we refer to this as an optimization step.

Algorithm 1 SMBO with Deep Ensemble surrogate

```
$X_{known}, Y_{known} \leftarrow$ Initial configurations and their evaluations.
$X_{pending} \leftarrow$ Configurations to evaluate.
$k \leftarrow$ Number of optimization steps.
for $i < k$ do
    DE $\leftarrow$ Randomly initialize a list Neural Networks.
    for $nn \in$ DE do                                  ▷ Can be trained in parallel
        train($nn$) with $X_{known}, Y_{known}$
    end for
    $EI_{scores} \leftarrow$ EI( $X_{pending}$ )                ▷ Expected Improvement scores
    $x^* \leftarrow$ best ($EI_{scores}$)
    $y^* \leftarrow f(x^*)$                            ▷ HP objective function evaluation
    $X_{known} \leftarrow X_{known} \cup x^*$
    $Y_{known} \leftarrow Y_{known} \cup y^*$
    $X_{pending} \leftarrow X_{pending} \setminus x^*$
    $i \leftarrow i + 1$
end for
```

We use similar procedures like Algorithm 1 for other models, i.e., FSBO and the proposed Ranking Loss surrogate model. The only difference is that the surrogate and its training differ with different methods.

A few points are worth noting here. First, the algorithm evaluates a set of discrete HP configurations in the HP search space. This is the same approach we take when applying this algorithm to our model because the ranking concept requires a set of discrete objects. The advantage of using this approach is that there is no restriction on the type of search space we optimize. It may be discrete or continuous. If it is continuous, we only have to discretize it to a required granularity based on our computational resources.

Secondly, neural networks are reinitialized before training at each optimization step. The old trained neural networks are discarded. The rationale behind this is discussed in Section 4.2.2. We also use this reinitialization in FSBO and the proposed ranking loss surrogate. Finally, as the neural networks are independent, they can be trained in parallel. This makes DE surrogates scalable.

One question that remains to be answered is what sort of architecture our neural networks use. For this, two architectures were analyzed - an undivided neural network as shown in Figure 3.1 and a neural network divided at its tail as shown in Figure 3.2.

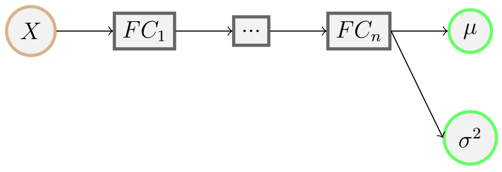

Figure 3.1: Example of an undivided Neural Network architecture.

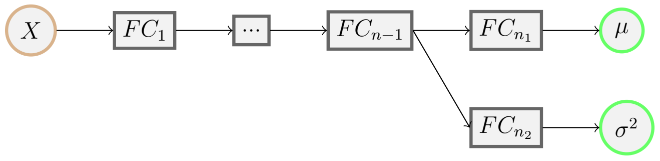

Figure 3.2: Example of a divided Neural Network architecture.

In both these figures, FC stands for fully connected layers. The ranking loss surrogate uses an undivided architecture similar to an architecture shown in Figure 3.1. Hence, we used this undivided architecture in the Deep Ensemble baseline implementation to compare the surrogates' results consistently.

3.8.2 FSBO

Few Shot Bayesian Optimization (FSBO) is a transfer HPO method that utilizes meta-learned surrogates for the knowledge transfer mechanism. It reformulates the HPO problem into a few-shot learning task. In FSBO, the surrogate meta-learns common characteristics of some given tasks by training on their respective metadata. After that, it adapts to the target task during HP optimization by finetuning for a few training epochs using the known HP configurations and their evaluations. Therefore, the following two steps are required to be done in chronological order:

1. Meta training - For knowledge transfer.

2. Finetuning - For few-shot learning.

The authors of the FSBO paper (Wistuba et al. [48]) make use of a deep kernel surrogate proposed by Wilson et al. [47]. Here, a neural network transforms points in the HP search space into a latent space. Kernels are then applied to this latent space in a Gaussian process to obtain a probabilistic evaluation. The deep kernel can is represented as: [48]

$$k(\phi(\mathbf{x}, \mathbf{w}), \phi(\mathbf{x'}, \mathbf{w})|\theta)$$

where $\mathbf{x}$ and $\mathbf{x'}$ are configurations in the original HP search space. θ and $\mathbf{w}$ are parameters of kernel k and the neural network ϕ respectively. We used the implementation of deep kernels provided by Patacchiola et al. [32] in this thesis.

During meta-training, we train our surrogate by learning the parameters θ and $\mathbf{w}$. We used the Matern $\frac{5}{2}$ kernel in our implementation. A fully connected neural network was used to obtain the latent space representation. A New FSBO model and, consequently, new surrogate parameters must be learned for every new search space. This is the case even if the input dimensions of the HP search space are the same. This is because every HP search space represents a different ML model.

If an assumption is made that the knowledge from the metadata is enough to predict the best HP configuration for all future tasks, the finetuning step may be skipped. However, this is rarely the case. There are always variations in the new tasks. Therefore, finetuning was used with the FSBO model in every HP optimization procedure.

The optimization algorithm in the case of FSBO is similar to Algorithm 1. The acquisition function used in this model is also Expected Improvement. The FSBO implementation also uses restarting the finetuning before each HPO optimization step. However, the meta-trained FSBO surrogate is loaded anew instead of randomly initializing it, as in the case of DE surrogates.

We used an early stopping mechanism for meta-training to avoid substantial computation costs and overfitting of the surrogate. We took advantage of the split of meta-training and meta-validation data to do this early stopping. The training was stopped if the training error was consistently higher than the validation error for a set number of epochs (we used ten epochs in our case).

Chapter 4

Proposed Method: Deep Ranker

The choice of the surrogate used in a model-based optimization algorithm is crucial. This is because the selected surrogate directly impacts the optimization performance. Moreover, we also have to answer critical questions prior to its selection. Some of them are - Does the surrogate have enough representative capacity? Does it have the capability of representing uncertainty? How is the surrogate learned?

In the quest to improve HPO surrogates, we propose and analyze a new type of surrogate model based on the concept of ranking: There are two primary components of our proposed idea

- The learning mechanism of the surrogate model.
- The surrogate model itself.

This chapter discusses in detail both these components. We initially use a model with sufficient representational capacity and a capability to represent uncertainty. An excellent model with these properties is a Deep ensemble; hence, we use it as a model. We then analyze and implement the proposed learning algorithm that uses the concept of ranking. Finally, we make the proposed ranking loss surrogate context-aware by integrating deep sets into the Deep ensemble architecture. We use SMBO as a reference HPO algorithm for our study.

4.1 Deep Ranker

In order to discuss the implementation details of ranking loss functions, we need to understand how a basic ranker works. A basic ranker consists of a

Deep Neural Network that outputs real-valued scores **s** for a batch of inputs, say **X**. These scores are used to rank the batch of inputs, with the highest score getting the lowest rank. Our thesis assumes that a lower rank is better than a higher one. Figure 4.1 depicts the architecture of a basic ranker.

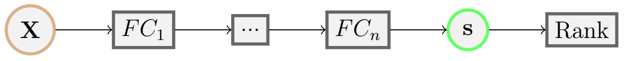

Figure 4.1: Basic ranker.

This DNN architecture is very similar to the DNNs used in the Deep Ensemble baseline implementation (Figure 3.1). The difference between the 2 is that Figure 3.1 had 2 outputs whereas Figure 4.1 has one output. In addition, there is one extra but crucial step of ranking in the basic ranker. Since we use a Deep Neural Network to get the scores before ranking the inputs, we may also refer to this architecture as a Deep Ranker.

Let $\{\mathbf{X}, \mathbf{y}\}$ be the training data used for training a generic machine learning model. Most loss functions utilize the values present in $\mathbf{y}$ as a reference for training the ML model. After the training completes, the range of the outputs of the learned ML model is not vastly different from the range of the actual output values. However, when we use a ranking loss function to train the ML model, we cannot guarantee this. This is because the ranking loss calculates the loss in the ranking space instead of the output space. The only thing that matters for ranking loss functions is that the objects should be ranked in the correct order. Hence the score range of the basic ranker can be arbitrary. The only exception is a point-loss function called Subset Regression. This is because subset regression directly learns the rank instead of using the output score s to rank the batch of inputs.

There may be occasions where the score range of the basic ranker needs to be controlled. For further discussion on this please refer Appendix A.

4.1.1 Uncertainty implementation using Deep Ensembles

We know from Section 1.1 that the evaluation of the target objective function in HPO is noisy. Hence it is essential for any surrogate that models this objective function to have the capability of estimating uncertainty. The uncertainty estimation becomes more critical in a sequential HPO algorithm like SMBO. In the first steps of SMBO, we do not have enough data to build or finetune the surrogate. Using a surrogate without uncertainty in the initial steps may give wrong results with high confidence. This is not good in a sequential process because selecting subsequent HP configurations depends on the previously selected HP configurations. For this reason, a model with uncertainty (if estimated correctly) is superior to other models.

We now integrate multiple basic rankers (or Deep Rankers) to build an ensemble of rankers. Using this ensemble, we can estimate uncertainty in the rank prediction. Figure 4.2 shows the architecture of the ensemble of basic rankers.

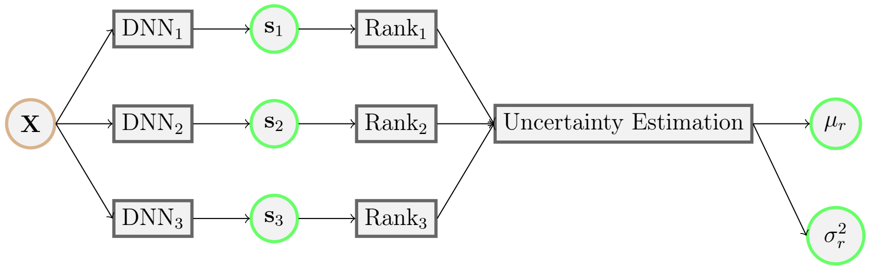

Figure 4.2: Ensemble of Basic Rankers.

In Section 3.7, uncertainty is estimated by calculating the mean and the variance of the DDN's output scores. This calculation of uncertainty is in the output space of the ensemble. In the case of ranking loss surrogates, uncertainty estimation is a little different. As we are primarily interested in ranking the HP configurations, individual scores and their distributions have less meaning for us. Instead, we are concerned about the uncertainty in the predicted rank. In other words, we want to calculate the uncertainty in the rank space.

We first calculate the rank of each HP configuration in the batch as per the rankers' output scores. Using these ranks, we calculate the mean rank μ_r and variance in the rank σ_r^2. Any acquisition function that uses a ranking model needs to use this "ranking distribution" instead of the output score distribution.

4.1.2 Acquisition function in the Ranking space

When doing HP optimization using ranking loss surrogates, it is essential to note that the ranking of an HP configuration is relative. A configuration can have different ranks based on the batch in which it is located. The expected rank of a configuration is not defined in absolute terms. It is always conditioned on other configurations in the ranking batch. Due to this property, using the predicted uncertainty in the ranking space is tricky.

Moreover, using well-defined acquisition functions like Expected Improvement (EI) is not trivial. EI requires us to maintain an incumbent evaluated value. The incumbent is the best-evaluated HP objective function's value in the output space. However, it is always 1 in the ranking space because the best possible rank in any ranking batch is always 1. Hence, if we used EI with ranking loss surrogates, it would be impossible to select the following HP configuration efficiently. It is also not possible to directly use the output scores of the ensemble. Each neural network in the ensemble gives separate scores for the input batch of configurations. These scores of each neural network may not be in the same range.

Due to these intricacies, we believe a detailed study of the different acquisition functions in the ranking space is necessary. Such a study is out of scope for this thesis. Therefore, we leave this study for future research. Instead, throughout our thesis, we use an elementary acquisition function with ranking loss surrogates. This acquisition function selects the HP configuration with the best mean rank predicted by the ensemble of rankers.

4.1.3 ListMLE Implementation

This thesis implemented the listwise loss function ListMLE to train the rankers. The deep learning library PyTorch [31] was used for the implementation. Algorithm 2 shows the steps to obtain the real-valued loss given the actual output and the predicted output. After calculating this loss, it is backpropagated through the ranker using the Autograd functionality of PyTorch. Please note that the implementation is a little more sophisticated than this due to the use of multi-dimensional tensors.

Before calculating the loss, the predicted scores are reduced by a constant factor for numerical safety and accuracy. Here we use the max of the predicted scores as a constant factor. This can be seen in Step 2 of ListMLE procedure in Algorithm 2. This reduction of the constant factor does not affect the loss because the strictly increasing positive function we use is exponentiation. This is illustrated in Equation 4.1 using sample numbers $a_i \in \mathbb{R}$. In this equation, k represents the total number of elements that are ranked.

$$\frac{e^{a_1}}{\sum\limits_i e^{a_i}} = \frac{e^{a_1+k}}{\sum\limits_i e^{a_i+k}} \tag{4.1}$$

Algorithm 2 calculates the permutation probability of the objects in the list as described in Section 3.5.2. However, this implementation is not efficient. The algorithm modifies and removes elements in predicted and actual lists.

Algorithm 2 ListMLE Algorithm

```
    Input : $l_{predicted} \in \mathbb{R}^k$                    ▷ Scores predicted by the ranker
    Input : $l_{actual} \in \mathbb{R}^k$                                  ▷ Actual scores
    Output : Loss $\in \mathbb{R}$
 1: procedure ListMLE($l_{predicted}$, $l_{actual}$)
 2:     $l_{predicted} \leftarrow l_{predicted} - \max(l_{predicted})$
 3:     $l_{predicted} \leftarrow \exp(l_{predicted})$                ▷ For Numerical Stability
 4:     $sum \leftarrow 0$
 5:     for $i < k$ do                                      ▷ $k$ is list size here
 6:         $sum \leftarrow sum +$ Top1LogProb($l_{predicted}$, $l_{actual}$)
 7:         $l_{predicted}, l_{actual} \leftarrow$ RemoveTop1($l_{predicted}$, $l_{actual}$)
 8:         $i \leftarrow i + 1$
 9:     end for
10:     Return $-1 * sum$     ▷ Negating loss as we are doing gradient descent
11: end procedure
12: procedure Top1LogProb($l_{predicted}$, $l_{actual}$)
13:     $j \leftarrow \text{argmax}(l_{actual})$
14:     $prob \leftarrow \frac{l_{predicted}[j]}{\sum l_{predicted}}$
15:     Return $\log(prob)$
16: end procedure
17: procedure RemoveTop1($l_{predicted}$, $l_{actual}$)
18:     $j \leftarrow \text{argmax}(l_{actual})$
19:     $l_{predicted} \leftarrow$ RemoveElementAtIndex($l_{predicted}$, $j$)
20:     $l_{actual} \leftarrow$ RemoveElementAtIndex($l_{actual}$, $j$)
21:     Return $l_{predicted}$, $l_{predicted}$
22: end procedure
```

Consider a case where the list elements are stored in a data structure that stores objects in contiguous memory locations (like an array). The time complexity of removing an element from an array is O(n), where n is the list size. The removal operation is done for all list elements one by one resulting in the worst-case time complexity of $O(n^2)$. Even if we store the list in a data structure that stores its elements in a non-contiguous memory location (for example, a linked list), accessing the elements in the list would have $O(n)$ time complexity. This would also result in time complexity of $O(n^2)$.

One way to solve this problem is to sort the lists before calculating the permutation probability. This approach is followed by Pobrotyn et. al [35] in their implementation. After sorting the lists, calculating the permutation probability can be done with linear time complexity. Since sorting has a time complexity of $O(n \log n)$, the worst-case complexity remains $O(n \log n)$. For this reason, we use the implementation given by Pobrotyn et. al [35] in this thesis. Algorithm 3 depicts this approach.

More precisely, the ListMLE loss is re-written in the following form:

$$L_{mle} = \sum_{j=1}^{k} \left(\log \sum_{t=j}^{k} \exp(s(\pi_k^*)) - \log \exp(s(\pi_j^*)) \right) \tag{4.2}$$

Since the list values are sorted in the correct order, we can further synthesize the equation as:

$$L_{mle} = \sum_{j=1}^{k} \left(\log \sum_{t=j}^{k} \exp(s^*(k)) - \log \exp(s^*(j)) \right) \tag{4.3}$$

Now, $\sum_{t=j}^{k} \exp(s^*(k))$ is nothing but the reverse cumulative sum which is the sum of all list values starting from position j. The reverse cumulative sum of all positions can be calculated and stored in linear time using iterative dynamic programming. The expression $\log \exp(s^*(j))$ can be directly written as $s^*(j)$ if we use a natural logorithm. Hence, if the reverse cumulative sum at position j is represented by $Q(j)$, the equation implemented in Algorithm 3 is given by:

$$L_{mle} = \sum_{j=1}^{k} (\log Q(j) - s^*(j)) \tag{4.4}$$

To test this implementation, we did a case study to learn the inverse mapping of real-valued numbers in a given range. For more details on the case study, please refer to Appendix B.

4.1.4 Weighted Loss

As discussed in Section 3.6, the ranking problem we have in the HPO domain is not the general ranking problem. It is more important for our ranking function to order the top part of the list than the bottom part. This constraint is evident from Algorithm 1 where we are only concerned about selecting the best available configuration for evaluating the HPO objective function at every optimization cycle. Keeping this constraint in mind, we need to use a weighting function $c(j)$ such that$c(j) \propto \frac{1}{j}$. Here, j is the ranking of the object.

Algorithm 3 ListMLE Algorithm (sorted)

```
    Input : $l_{predicted} \in \mathbb{R}^k$                          ▷ Scores predicted by ranker
    Input : $l_{actual} \in \mathbb{R}^k$                                        ▷ Actual scores
    Output : Loss $\in \mathbb{R}$
 1: procedure ListMLESorted($l_{predicted}$, $l_{actual}$)
 2:     $l_{actual}$, IndexOrder $\leftarrow$ Sort($l_{actual}$)
 3:     $l_{predicted} \leftarrow$ SortWithIndexOrder($l_{predicted}$, IndexOrder)
 4:     $l_{predicted} \leftarrow l_{predicted} - \max(l_{predicted})$     ▷ For Numerical Stability
 5:     $prob \leftarrow 0$
 6:     for $i < k$ do
 7:         $prob \leftarrow prob + \log Q[i] - l_{predicted}[j]$
 8:         $i \leftarrow i + 1$
 9:     end for
10:     Return $prob$
11: end procedure
```

This section discusses some strategies to do this weighting for our problem. The concept of biased ranking loss is discussed in "Top-Rank Enhanced Listwise Optimization for Statistical Machine Translation" by Chen et al. [6]. Chen et. al propose a position based weighting of the ranking function such that:

$$w_j = \frac{k - j + 1}{\Sigma_{t=1}^{k} t} \tag{4.5}$$

where w_j represents weight of the object at position j in the ordered list. k represents the total number of elements that are ranked. This strategy decreases the weights linearly across the ranks.

Linear weighting has an issue. It makes the weight of the middle object in the list 0.5 times the weight of the object at the top. This can be seen in the position dependent attention graph of figure 4.3 (The weights of the position dependent attention graph are scaled by a factor of 50 to make a fair comparison between the weights). This, however, is not what we want for our case. We need a sharper decrease in weights across the ranks. We could use 2 alternative strategies of weighting the ordered list

- Inverse linear weighting given by $w_j = \frac{1}{j}$.
- Inverse logarithmic weighting given by $w_j = \frac{1}{\log(j+1)}$.

Figure 4.3 shows all 3 weighting strategies. In the case of inverse linear strategy, we see that the weight becomes extremely small after only a few

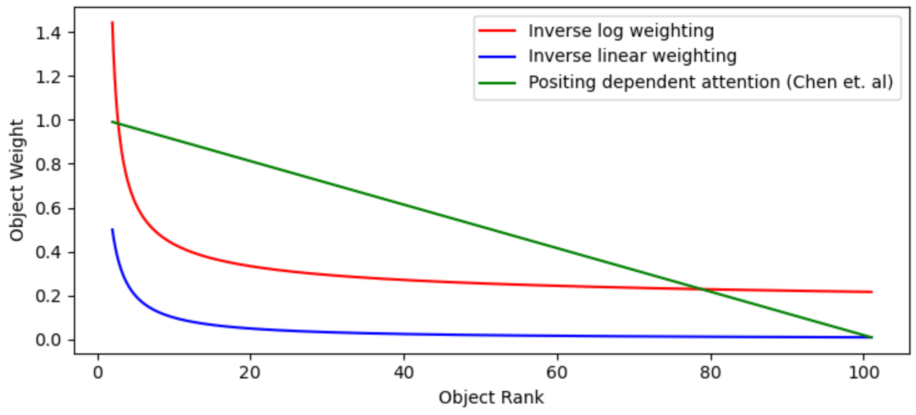

Figure 4.3: Different Weighting Functions.

ranks in the list. It has an unwanted consequence for our problem. Because the weights of objects at the end of the list are too small, the ranking function may not learn to rank optimally. On the other hand, inverse log weighting is the most suitable because it neither completely ignores the objects at the end of the list nor gives them too high an importance. The weights at the bottom of the list are very similar, which is what we want - all objects at the tail of the list are equally "unimportant" in our case.

Another advantage of using inverse weighting is that we can employ parallelism to enhance the HPO during the optimization cycle. For example, the top n configurations can be selected at every step, and they can be tried in parallel for optimization. This overcomes any uncertainties that may occur while ranking in the top part of the list. The parallelism is ineffective when using the inverse linear weighting or weighting proposed by Chen et al.

4.2 Method training and optimization

The proposed method in this thesis, "Ranking loss surrogate," is a transfer HPO solution. For this reason, its working principle is similar to that of the FSBO baseline model. Hence, the two main parts of using the optimization process are

- Meta-learning the ranking loss surrogate.
- Using the trained surrogate in the optimization cycle (with or without finetuning).

4.2.1 Meta training the surrogate

To understand the meta-training of the proposed surrogate, we need to understand the search space in question. The space of hyper parameter configurations in which we try to get an optimum during any HPO is called HP search space. This search space is different for different ML models. Hence we need to meta-train different ranking loss surrogates for different HP search spaces. However, the same ML model can be used to fit different data sets or tasks. We get different HP response surfaces in the same search space if we train the ML model for a different dataset. Hence we obtain different metadata for the same search space, each for a particular dataset used during the training. The goal of meta-training the surrogate is to use different metadata sets (or tasks) from a single search space to learn all the typical characteristics and transfer this knowledge to a new task.

After understanding the goal of meta-training, we need to define its mechanism. Since we use an ensemble of Deep Neural Networks as a surrogate in our proposed idea, we can employ two strategies for meta-training:

- Meta-train each network separately.
- Use the mean rank of the entire architecture to train the networks together.

A combined meta-training method is more efficient than meta-training each neural network separately. However, it is theoretically unnecessary to do this. Moreover, the combined meta-training also reduces the variance in the ensemble. This is because we must tie up each neural network's losses before backpropagating the loss through the networks. This reduces the independence of the networks in the ensemble, which is a disadvantage. Therefore we train each network separately.

We use stochastic gradient descent to meta-train every network. To do this, we must sample a batch of data from various metadata in a search space. There are two ways to accomplish this:

- Double sampling: Sample the task, then sample the metadata within the task.
- Sample metadata from all tasks together.

We use the first method of sampling. The reason for doing this is due to have a higher level of randomness in the sampling process. If we use a double sampling mechanism, there is also a good regularization and a possibility to use a lower learning rate due to a higher number of gradient descent steps in the meta-training process. We use Adam as our surrogate's optimizer with a learning rate of 0.001. We do the meta-training for 5000 epochs, and in each epoch, we take 100 steps in which each step does one double sampling. While sampling data points from within a metadata set, we sample without replacement. We use the number of lists (batch size) as 100 and the size of each list also as 100. Algorithm 4 illustrates to us a brief skeleton of the meta-training procedure. After the meta-training, every model is saved to a persistent location (e.g., hard disk) so that it can be loaded when necessary.

Algorithm 4 Ranking Loss surrogate Meta-training

```
    Input : $epochs \in \mathbb{I}$
    Input : $X_{train}$, $y_{train}$    ▷ Meta data used to train from all Search Spaces.
    Input : $m_\theta$                  ▷ Surrogate model to train (parameterized by $\theta$)
 1: procedure MetaTrain($m_\theta$, $X_{train}$, $y_{train}$, $epochs$)
 2:     for $i < epochs$ do
 3:         for $j < 100$ do
 4:             $B_X, B_y \leftarrow$ DoubleSample($X_{train}$)    ▷ Get the training batch
 5:             $y_{pred} \leftarrow m_\theta(B_X)$
 6:             loss $\leftarrow L_{mle}(y_{pred}, B_y)$
 7:             $g \leftarrow \frac{\partial \text{loss}}{\partial \theta}$    ▷ $g$ stands for gradient
 8:             Use $g$ to update $\theta$ using the Adam optimizer
 9:         end for
10:     end for
11: end procedure
```

4.2.2 Finetuning

Finetuning is the second major part of the optimization process. It may be considered optional for transfer HPO surrogates like ours. However, we found that it gave improved results.

We take the SMBO optimization of Deep Ensembles in Algorithm 1 as an example to understand the finetuning process. In this algorithm, the deep ensembles are retrained at every optimization step with the seen evaluations of the true HPO objective function. We need to modify Algorithm 1 by replacing Deep Ensemble training with the finetuning of our model. To avoid duplication, we do not rewrite the complete algorithm again.

The finetuning process is similar to the MetaTrain procedure in Algorithm 4. The difference is that we use seen evaluations instead of the train split of meta-data. Moreover, sampling is not required during the finetuning process as we use all of the available data. We do finetuning for 1000 epochs. The number of epochs can be varied based on the computation resources available.

We have to reload the saved surrogate at every optimization step to use the ranking loss surrogate as a transfer HPO method. This is similar to how FSBO is used. However, if we initialize it randomly instead of loading the saved surrogate, the surrogate can be used as a non-transfer HPO method. Nevertheless, there is a requirement to start the finetuning process afresh. In the next section, we discuss why this restart is required. Subsequently, we talk about the cosine annealing used during finetuning.

Requirement of restarting training

During the implementation of deep ensembles, we found that the model performs much better in the HPO evaluation cycle when we train it from scratch at every optimization step (acquisition step). This is shown in Section 5.1. This result is counterintuitive because an already trained model should quickly converge to a local optimum. One of the reasons for this performance anomaly is that the model gets biased towards the points observed in the starting steps of the optimization cycle. Lets say we have 2 models

- $\mathrm{m}_{\mathrm{restart}}$ which always restarts training at every acquisition step.
- $\mathrm{m}_{\mathrm{reuse}}$ which continues to train the model trained in the previous optimization steps.

Let the models be finetuned for 100 epochs at each step. Let there be just one seen HP configuration at the beginning of the optimization. The figures 4.4, 4.5, and 4.6 show the number of times each observed HP configuration is used for training the model at the 25th, 50th and 100th optimization step by the $\mathrm{m}_{\mathrm{reuse}}$ model.

We in figures 4.4, 4.5, and 4.6 that the number of times a data point is used during the training is biased with the first data point being used much more than the last. Generally, all the data is used during finetuning due to data scarcity. Hence, in our example, the number of times an HP configuration is used scales with the number of epochs trained. The usage bias is not intended because all observed HP configurations should be treated equally in any training or finetuning step. When we use $\mathrm{m}_{\mathrm{restart}}$ every known HP configuration is used only 100 times for finetuning at every evaluation step. Hence it is better to restart the model before finetuning.

The second reason for restarting is that the surrogate model may get stuck at a stubborn local minimum at any finetuning step n, where $1 <= n <= 100$

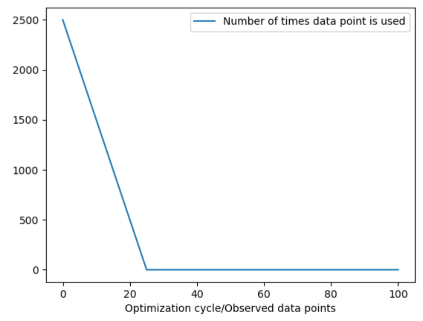

Figure 4.4: Bias at 25th optimization cycle.

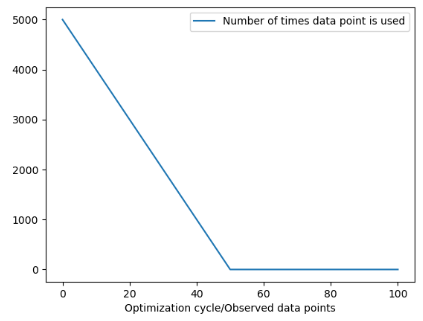

Figure 4.5: Bias at 50th optimization cycle.

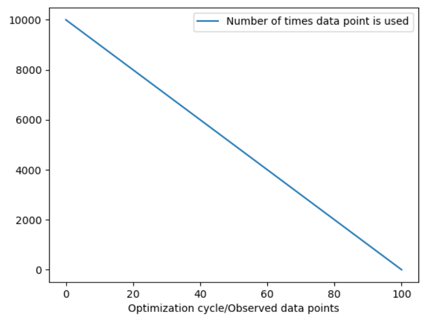

Figure 4.6: Bias at 100th optimization cycle.

in our case. Coming out of this local minimum may require the response surface to change drastically. The response surface will change drastically only when the training data distribution changes significantly. This is not possible in the sequential process of SMBO because, at every step, only one new HP configuration is added to the known HP configurations.

Figure 4.7 depicts this issue for a simple 1 dimensional case. Consider the red curve. It represents the HP response surface for k known data points. When we add $(k+1)^{th}$ data point, most of the response surface remains the same. Only part of it changes. The blue curve represents the response surface for $(k+1)$ data points. If our model is already trained for the red curve, it becomes challenging for it to come out of the local minimum because of the minimal changes in the response surface. We call this such a minimum, a stubborn local minimum. The brown dot in the figure depicts the stubborn local minimum.

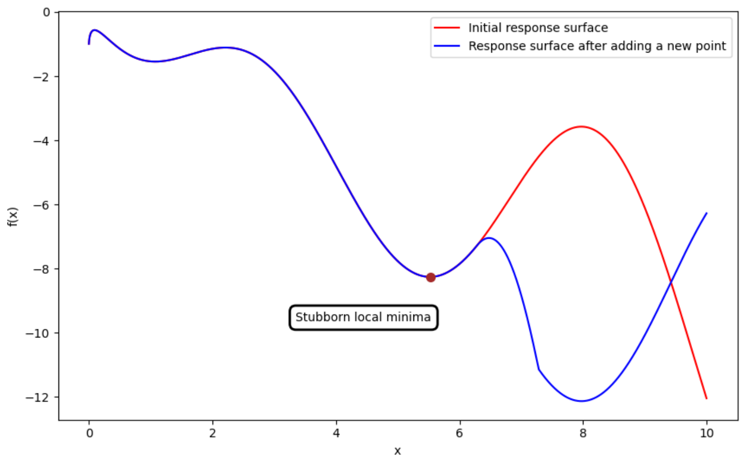

Figure 4.7: Figure showing changes in response surfaces and a stubborn local minimum.

If we reload and retrain our model afresh, the probability of reaching a good local minimum is higher. For example, during training for the blue curve, the surrogate model can start from a value $x > 10$. This would enable it to reach a better local minimum than a model that does not restart finetuning. The number of ways to reach a good local minimum would increase exponentially with the increase in the dimensions of the HP search space. Hence, restarting the training (from the saved surrogate model in our case) should improve our finetuned surrogate model.

Because of these reasons, we use the restart mechanism during the finetuning of our ranking loss surrogate (and baseline implementations). This makes the surrogate more robust.

Using cosine annealing

When we plotted the finetuning loss curves of our model, we found that these loss curves were very jittery. We noticed the same behavior when finetuning the FSBO model. Figure 4.8 shows the finetuning loss for one of the search spaces.

We hypothesize that these jittery losses are because the learning rate is unsuitable for the response curve's curvature at the targeted local minima. One solution to this is to use a lower learning rate. However, finetuning using a lower learning rate will take a long time to converge. The solution we

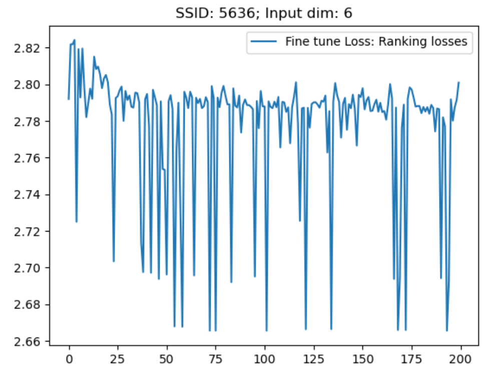

Figure 4.8: Figure showing jittery loss function curve.

proposed and utilized was using cosine annealing.

There are a couple of advantages of using cosine annealing. First, the initial learning rate is kept high to give the optimizer time to get to an area close to the local minima if it has not done so. After that, the learning rate declines quickly to a small value. This helps the model get deep into the local minimum; hence, the probability of the optimization jumping out of the local minimum is very low.

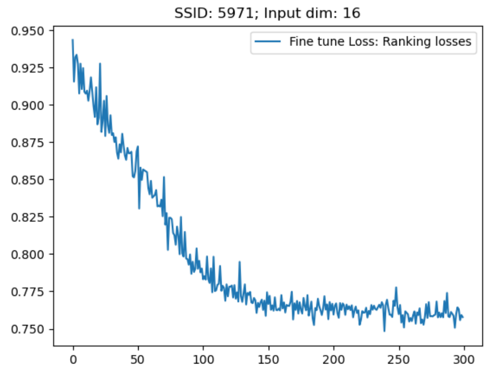

Figure 4.9: Figure showing a loss curve obtained by using cosine annealing.

Figure 4.9 shows a finetuning loss curve obtained using cosine annealing. This strategy reduces the problems that could prop us when using a constant learning rate. However, we realized at the later stage of the thesis that cosine annealing could not be used extensively while finetuning the proposed model due to the issue of negative transfer learning. During training, on the other hand, we use a constant learning rate as we can use the validation loss as a reference for overfitting or underfitting.

4.3 Ranking Surrogate Model

After explaining the learning mechanism, we now turn to the last critical component that is used in the proposed method. This is the addition of Deep Sets to the architecture to make our model context-aware. Context-aware means that the surrogate model conditions its outputs based on the available HP configurations (or support configurations). This helps the surrogate distinguish between different tasks. We first discuss the surrogate architecture after the addition of Deep Sets, and after that, we explain how meta-training is done for this surrogate.

4.3.1 Using Deep Sets to build context aware models

We have already discussed the fundamental ideas of using deep sets in 2.6.1. The basic idea is to precondition the surrogate function s_θ on known evaluations of the target HP response surface. These known evaluations act as a support for the scoring function. Hence they are subscripted as "s" in the equation. We then query the surrogate to get the relevant ranks of new $\mathbf{X}$ which are subscripted as "q" in the equation. The surrogate represents a function of the form:

$$s_\theta(\mathbf{X}_q|\mathbf{X}_s, \mathbf{y}_s)$$

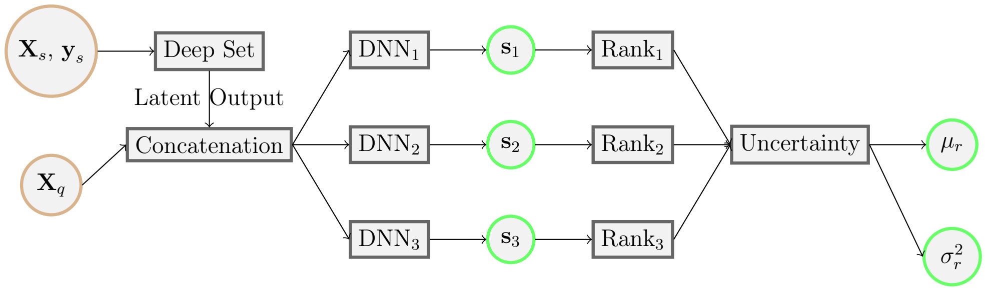

Figure 4.10: Skeleton of the proposed model with Deep Sets.

The architecture of the surrogate s_θ with deep sets is given in Figure 4.10. For the complete architecture of the Deep Set node, please refer 2.3. We keep the node abstract for simplicity.

To rank a batch of query HP configurations $\mathbf{X}_q$, first, the data points in the support set $\mathbf{X}_s$, $\mathbf{y}_s$ are passed through the deep-set. Each X_s^i, y_s^i are concatenated to get a single vector for input into the deep set to obtain a vector $X_s^i : y_s^i$. This concatenation is passed through the Deep-set, we obtain the latent output O_l. Please note that there is a single latent output from the deep set and all the support HP configuration points need to be input to it in one shot.

It can be argued that we could use the concatenation $X_s^i : rank(y_s^i)$ instead of directly using the actual value of the objective function. This is a way to concatenate a value from the ranking space instead of the output space. However there are two significant issues with this approach. First, the size of the support set is undefined; hence the range of the ranks can vary drastically. Second, we already know that values in the ranking space are relative. So even if we use the same number of support set elements, the same HP configuration can have different ranks in different support batches. Hence we believe a deeper study of the ranking space is required before it can be employed in the concatenation.

After obtaining the latent output O_l from the deep-set, it is concatenated with the query data points. The concatenated result is passed through a Deep Neural Network to get their respective scores and, subsequently, their ranks. For example if there is a batch of queries given by $\{X_{q_1}, X_{q_2}, X_{q_3}...\}$, and the latent output is given by O_l then the concatenation yields

$$\{O_l : X_{q_1}, O_l : X_{q_2}, O_l : X_{q_3}...\}$$

The whole batch of these concatenated vectors is then ranked using the rest of the architecture.

Meta training

Using the architecture with a deep-set during the HP optimization is very straightforward. We can use the known HP configurations as a support set, and the queried HP configurations as the query set. During meta-training, however, we have to divide the data into support and query sets.

In the meta-training, we randomly select 20% of the batch size as a support set. The sampling is done without replacement. For example, if the batch size is 100, we select 20 HP configurations as a support set. Afterward, we select the query points from the remaining choices (again without replacement). The number of query points depends on the batch size. If the batch size is 100, we will select 80 HP configurations randomly without replacement. We do not simultaneously sample the support and query points from all the given meta datasets. As already discussed in Section 4.2.1, we do a double sampling where we first select one meta-data in a search space. From the selected meta-data, we sample the support and query HP configurations. We sample data from all the training and validation tasks as separate batches to calculate the average training and validation loss.

Chapter 5

Experiments and Results

In this chapter, we present the experiments we conducted and the results we obtained to some of the research questions that arose during the thesis. We first understand the structure of the metadata used for meta-training, meta-validation, and meta-testing. In the subsequent sections, we present the results obtained in detail.

Meta-Data

To compare our proposed ranking loss surrogate with other HPO surrogates, we would have to run the Bayesian optimization (BO) on a set of different machine learning models. The BO would have to be run with the ranking loss surrogate and other surrogates. Moreover, the optimum within the HP search space is different when the ML model is trained on different data sets. In addition to this, due to the stochastic nature of ML models and the surrogate models, we would have to run the HPO multiple times for each dataset.

As one can see, this evaluation, if done right from the training of the ML models, is not feasible. To overcome this challenge, we use a meta dataset proposed by Pineda et al. called HPO-B [34] to benchmark our results in the thesis. Using this benchmark, we do not need to train our ML models from scratch, as the metadata contains evaluations of multiple HP configurations for different ML models and datasets. In the rest of the section, we discuss the organization of the benchmarking metadata.

HPO-B is a benchmark that is used for doing black box HPO. Both transfer and non-transfer HPO surrogates can be studied using this benchmark. The meta data in HPO-B comes in 3 versions namely **HPO-B-v1**, **HPO-B-v2**, and **HPO-B-v3**. Of this **HPO-B-v3** contains distilled search spaces that have the most datasets. **HPO-B-v3** can be split into meta-training, meta-validation, and meta-test sets. Using the meta-training dataset, one can meta-learn an HPO surrogate. Then the surrogate is evaluated and tested using the meta-validation and meta-test data, respectively. The metadata in any split (training, testing, or validation) is organized in a JSON format. Figure 5.1 shows an illustration of this format.

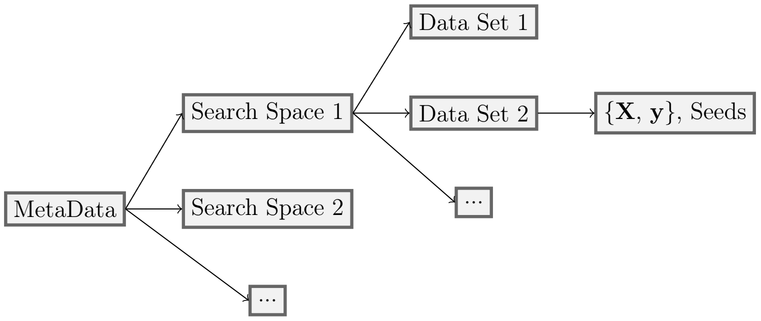

Figure 5.1: Structure of the metadata in the HPO-B benchmark.

As shown in Figure 5.1, the metadata consists of a list of HP search space ids. An HP search space id corresponds to a single ML model. Each search space further has multiple dataset ids. These dataset ids correspond to different training data used to train ML models. Each dataset id contains an **X**, **y** pair and 5 seeds. **X** represents a set of HP configurations and **y** represents their respective true evaluations. The Bayesian optimization used in our thesis starts with different initial known HP configurations. One initial configuration is called a seed. A seed gives a set of values (or indices in implementation) from the **X**, **y** pair.

We compare the results of ranking loss surrogates with transfer HPO surrogates and non-transfer HPO. HPO-B can be used for analyzing both types of surrogates. However, to cross-compare transfer and non-transfer HPO, we only compare against **HPO-B-v3** test split as recommended by Pineda et al. [34].

Experimental Protocol

To test the performance of ranking loss surrogates at different stages of development, we used four main baseline methods - Random Search, SMBO using a Gaussian surrogate, SMBO using Deep Ensembles, and FSBO. We reused the Gaussian and Random surrogate implementations of HPO-B in the thesis. However, the Deep Ensemble and FSBO surrogates were implemented from scratch.

In the given meta data if we take the set of all combinations in **HPO-B-v3** test split i.e {Search Spaces $\times$ Metadata sets $\times$ Seeds}, we get 430 HP optimizations. We start with five initially known HP configurations given by the HPO-B seed in each optimization and run the evaluation cycle for 100 iterations. This evaluation cycle is discussed in Algorithm 1 albeit for deep ensembles.

One optimization creates one array of incumbent best (highest) HP evaluations. This incumbent array for different models is compared against each other to create a rank array for each surrogate. For example, if at the evaluation step 23, the incumbent of surrogate a has a higher value than of surrogate b, then the rank of model a at step 23 is lower than model b (if we consider lower rank to be better). We get a list of rank arrays for every surrogate if we do this for all optimizations. This rank array list is averaged to get a single averaged rank array. Plotting this averaged rank array gives us a rank graph of a surrogate. We use this **rank graph** to compare the different surrogates in this thesis. For example, Figure 5.2 compares the rank graph of 2 baselines.

5.1 Baseline Results

The results of Deep Ensemble and FSBO baselines are discussed in this section.

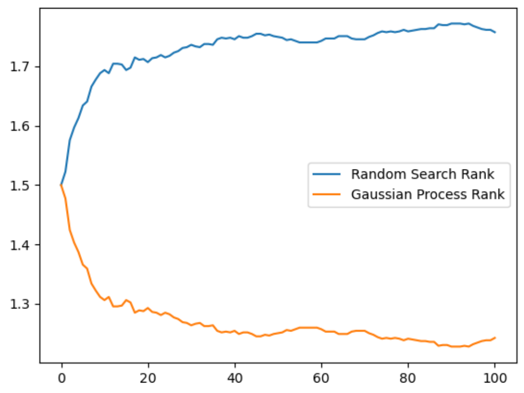

Figure 5.2: Sample rank graphs of Random and Gaussian surrogates.

Deep Ensembles

The Deep Ensembles (DE for short) we used contained five neural networks by default. Each neural network had two fully connected neural layers with 32 neurons each. All neural networks used the same architecture. We considered only the non-transfer case for DE. Hence we did not do any meta-training for them. The training was done using the Adam optimizer with a full batch gradient. This is because there are very few data points during any optimization step. We train each neural network for 1000 epochs with a learning rate of 0.02 at every evaluation step in an HPO optimization. We do not use adversarial examples as proposed by Lakshminarayanan et al. [24] as the input HP search space may not be continuous.

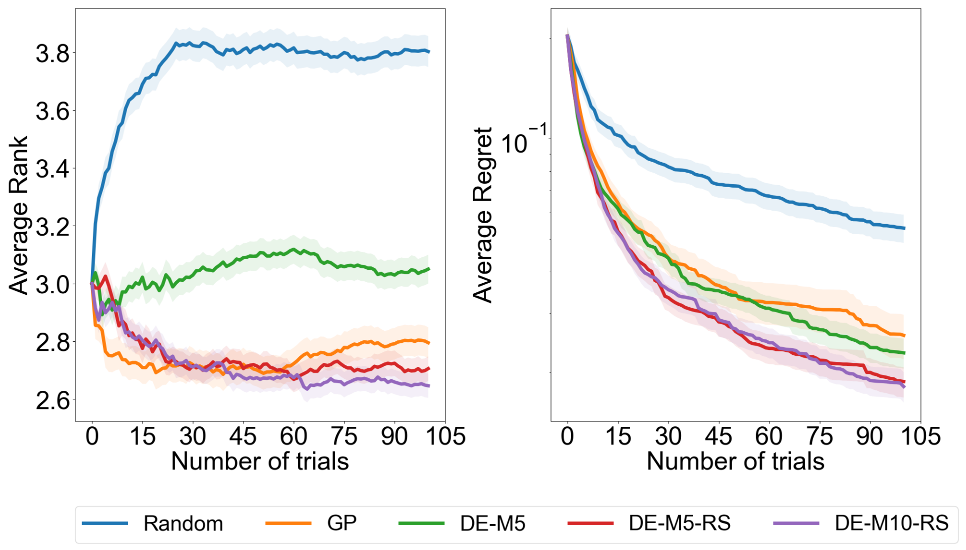

Figure 5.3: Graphs showing performance of Deep Ensembles.

Figure 5.3 shows the performance of various Deep Ensemble models. We used the best results of Random and Gaussian surrogate baselines to compare how Deep ensembles fair in the HPO-B benchmark. In addition to plotting the rank graph, we also plot the average regret for every model. In the average regret graph, the y-axis represents the regret on a log scale. In Figure 5.3, DE-M5 stands for Deep Ensembles with five neural networks, whereas DE-M10 stands for Deep Ensembles with ten neural networks. An "RS" suffix in the legend denotes that the surrogates are reinitialized before finetuning at every evaluation step.

In this thesis, we studied the following main models of Deep Ensembles

- Deep Ensembles that finetune with no restarting.

- Deep Ensembles that finetune with restart at every evaluation step.
- Deep Ensembles with 5 and 10 neural networks.

We observed in our study that Deep Ensembles with restarts (at every evaluation step) give us better results than Deep Ensembles that do not do a restart. This observation proves our hypothesis of the requirement of restarts as discussed in Section 4.2.2. Furthermore, we observe that using deep ensembles with restarts gives us results comparable with Gaussian surrogates. Increasing the number of neural network ensembles does not improve performance significantly. We see this illustrated both in the ranking graph and the average regret graph. We also use critical rank graphs developed by Pineda et al. [33]. Critical graphs represent ranks at particular evaluation steps averaged across all HPO optimizations. Figure 5.4 illustrates a critical rank graph for the Deep Ensemble study.

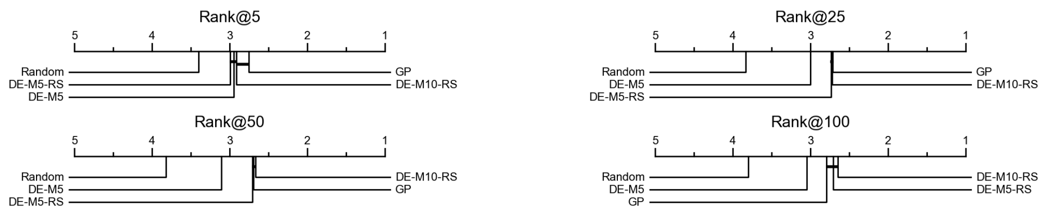

Figure 5.4: Critical rank graphs of Deep Ensembles at the 5^{th}, 25^{th}, 50^{th}, and 100^{th} evaluation step.

After the DE results analysis, several advantages and disadvantages of using the DE surrogate were found. First, being a non-transfer HPO surrogate, DE can be applied to any continuous HPO problem. Second, as each neural network is independent of the others, the neural networks can be trained in parallel. Hence, DE can be scaled quickly based on the available computing power. On the negative side, since no meta-training is done, we have to fit the DE on the very few data points available during the HPO optimization. This training on very few data points leads to overfitting DE surrogates because of their enormous representative capacity. In addition, DE is incapable of modeling objective functions with discrete or semi-discrete domains. The input data points to DE are assumed to lie in a continuous space. Hence DE will not be able to model discrete and semi-discrete HP search space trivially.

Few Shot Bayesian Optimization (FSBO)

FSBO was selected as a baseline because it was the state-of-the-art baseline in the transfer HPO problem domain. During our implementation of FSBO, we found that an FSBO model with four neural network layers of 32 neurons each gave us the best performance. This architecture is similar to the Deep Ensemble architecture. The difference between the architectures is that FSBO has four network layers instead of 2. We used early stopping to get better training loss, as mentioned in Section 3.8.2. The learning rate used for meta-training was 0.0001, and that used for finetuning was 0.03. We finetuned our model for 500 epochs. The learning rates of the kernel parameters and the neural network parameters were kept identical. We used Adam optimizer for both training and finetuning. However, during the finetuning phase, we used a cosine annealing schedular. The reason for using cosine annealing has been discussed in Section 4.2.2. Finally, we also do restarts for the FSBO domain with the restart mechanism discussed in Section 3.8.2.

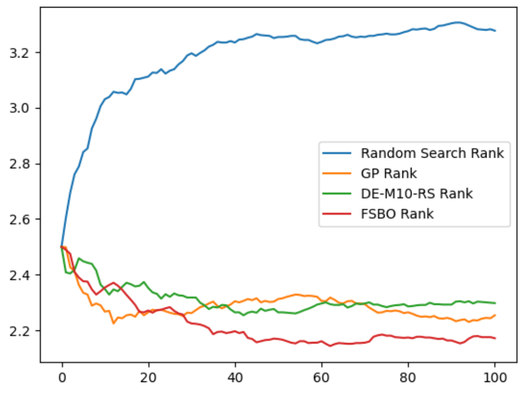

Figure 5.5: Rank graph showing performance of self implemented FSBO.

Figure 5.5 shows the rank graph of FSBO compared to the best performing models of random search, GP and DE surrogates. There is a slight difference in ranking graphs of previous surrogates because a few HPO optimizations were unsuccessful for the FSBO due to non-positive definite Matrix exceptions obtained during meta-training and finetuning (The GPytorch implementation threw this). As we can see, our implementation could not give us state-of-the-art results for FSBO. Since these results are a correct yardstick to measure the ranking loss surrogate model, we used the results obtained by the FSBO implementation provided by Pineda et al. [33] in the rest of the thesis.

Figure 5.6 compares the best performing FSBO surrogate against the performance of other baseline surrogates. As one can see from this figure, FSBO is an HPO model that outperforms the other HPO models both in the rank graph and the regret graph. Moreover, FSBO always has a better rank at every evaluation step. This can be seen in the critical rank graphs illustrated in Figure 5.7. Using these baseline results as a reference, we discuss how the ranking loss surrogate model compares to these methods in detail in the following sections.

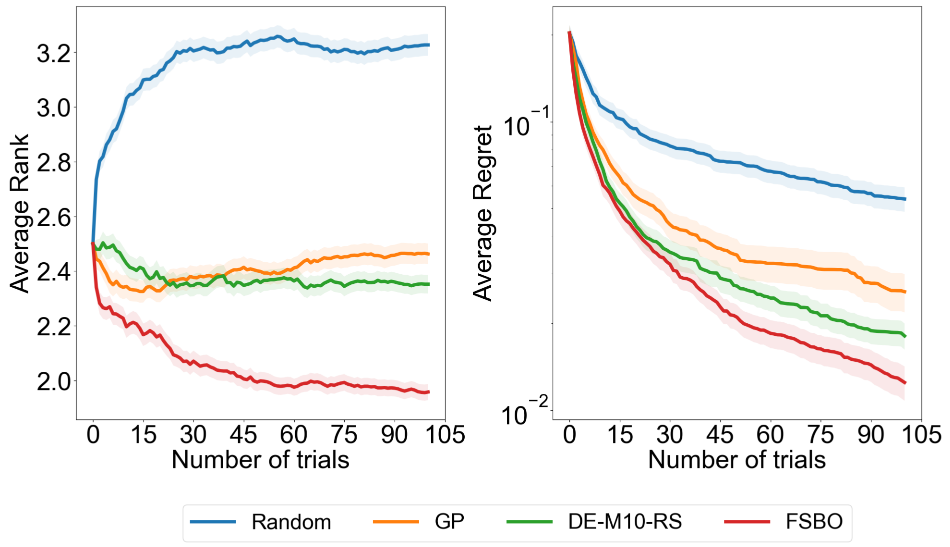

Figure 5.6: Rank and regret graph of best performing of FSBO.

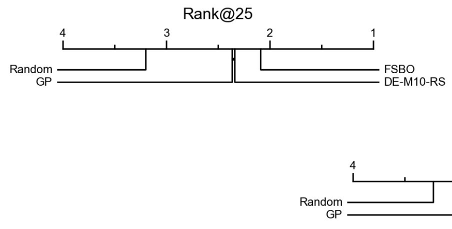

Figure 5.7: Critical rank graphs of baselines at different evaluation steps.

5.2 Ranking Loss model results: Research question results

As discussed in Chapter 4, our proposed idea has two components - a ranking loss and a surrogate model. In both these components, several research questions need to be studied. This section discusses the results or answers to these research questions in different sub-sections separately. In each section, we present an ablation study to highlight the essential components of our proposed idea.

For uniformity, we use the following strategy - We will use the same number of neural networks and the same neural network architecture in our study. We also try to keep the learning rate similar for similar components. For different components, however, we use different learning rates. For example, learning rates during meta-training and finetuning are different. Similarly, each loss function may require a different learning rate to perform the best as the HP response surfaces change with the used loss function. Like in the best case of Deep Ensembles, we use ten neural networks in our ensemble. Each neural network has two 32-neuron layers.

5.2.1 Meta training results

In this section, we discuss the observations we made during the meta-training of the proposed model. The effect of finetuning on the optimization results is discussed in detail in Section 5.2.5. Please note that this meta-training is required only for the transfer HPO case. For the non-transfer case, we directly do a finetuning with the evaluated HP configurations during the HP optimization. The two main architectures we study in this thesis are the Basic Scoring model and the Basic Scoring model with Deep Sets.

The training of the Basic Scoring Model (without Deep sets) obtained relatively smooth loss curves. This is illustrated using the loss curves of the search space with id "4796" in Figure 5.8. The "input dim = 3" in the image title specifies that the dimensionality of this HP search space is 3. After adding deep sets to the model, the loss curves become jumpy. This is illustrated in Figure 5.9 using the same search space. These jumpy loss curves are expected because adding deep sets increases the architecture's representation capacity (and the parameter space). A bigger parameter space results in more noisy gradients during the stochastic gradient descent.

The second thing we observed is that sometimes the validation loss gets significantly worse during the training. Figure 5.10 illustrates this sort of behavior. One reason for this behavior could be that the validation task is not similar to the training task. There is some negative knowledge transfer from the meta-training phase to the meta-validation phase. This type of learning is called Negative transfer learning and is discussed in detail in Section 6.2. Due to the negative transfer learning issue, we could not employ an early stopping mechanism like in the case of FSBO baseline implementation for the ranking loss surrogates.

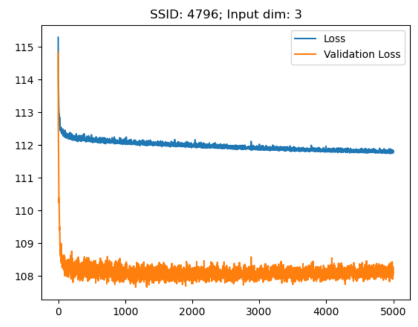

Figure 5.8: Loss curves for the basic scoring model.

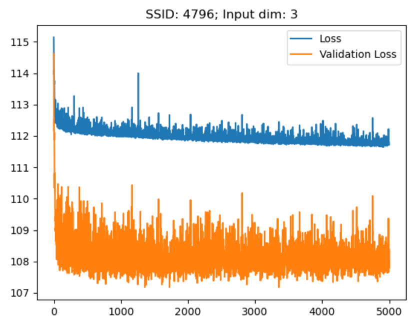

Figure 5.9: Loss curves for a model with deep sets.

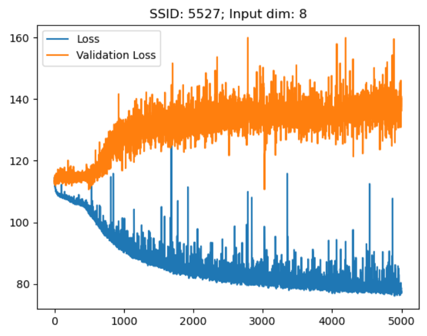

Figure 5.10: An instance of negative transfer learning.

5.2.2 Study of ranking losses

In this section, we discuss the following question - are ranking losses better for HPO than regression ones? Secondly, which ranking loss is better for HPO: pointwise, pairwise, or listwise loss? For the regression case, we use the Deep Ensemble baseline implementation proposed by Lakshminarayanan et al. [24] as a reference. We use subset regression, RankNet, and ListMLE loss functions for pointwise, pairwise, and listwise ranking losses, respectively.

Non-transfer HPO

We first test the non-transfer HPO case where the surrogate models are not meta-trained. We only do finetune (or training) during the evaluation phase. We also reinitialize the models before every evaluation step. We have discussed the reason for this in-depth in Section 4.2.2. We train each neural network for 1000 epochs at every evaluation step. The learning rate is 0.02, and we do not use cosine annealing. Using cosine annealing here did not make sense since there is no guarantee that the randomly initialized model is near a local optimum. Hence, we need a bigger learning rate to find quicker optimization. We try to keep all the parameters the same as the best case of DE results (Our reference method for the non Transfer HPO case)

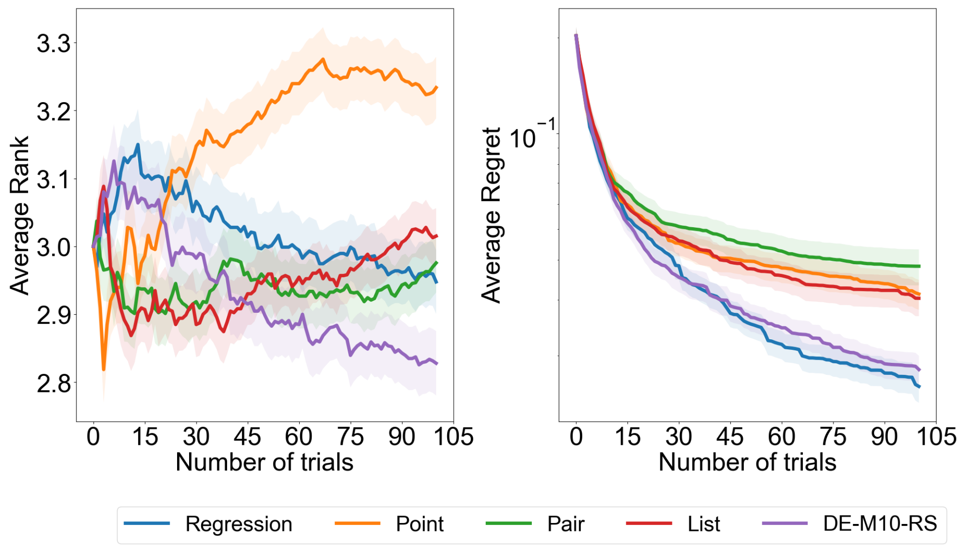

Figure 5.11: Comparison of Ranking losses with regression losses (Non-transfer HPO case).

Firstly, we see that the average regret for both the regression losses is similar. This result is expected because the only significant difference between the two models is how uncertainty is modeled. Secondly, we see from Figure 5.11 that the models trained with ranking losses are better at the beginning of the optimization steps cycle than the regression losses. This can be seen in the critical rank graphs of Figure 5.12 as well. However, the pointwise loss function very quickly deteriorates in performance in the long run. The pointwise loss function tries to learn the rank directly from the input. The learned ranks also depend on the other HP configurations in a given batch. Hence this loss function does not train the model to learn relative differences in the HP configurations.

Transfer HPO

In the transfer HPO case, we first train the models with the meta-training data given by the HPO-B data set and then store our models on persistent

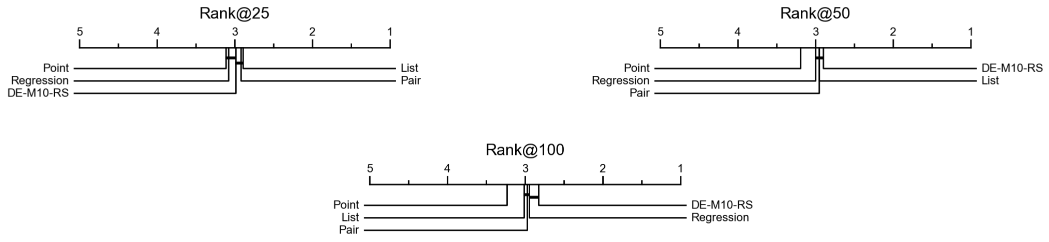

Figure 5.12: Critical rank graph of the non-transfer HPO methods.

storage. We train the models for 5000 epochs with a fixed learning rate of 0.001. We used a batch size of 100 for all models. We used a list size of 100 for the listwise loss functions. However, we use cosine annealing during finetuning for the reasons discussed in Section 4.2.2. It is because there is a very high chance that the learned model is close to any one of the local optima (the exception to this is the negative learning rate case discussed later).

The selection of batches of HP configuration for training is not trivial. For any given search space, we have data on multiple tasks. As the distribution of HP configurations may differ for different tasks, we need to learn to rank the HP configurations against other configurations within the same task. Hence, for each step within an epoch (i.e., for the number of batches within the epoch), we randomly select a task (i.e., meta-dataset) and then select the HP configurations within this task. This sampling mechanism is the double sampling mechanism we discussed in Section 4.2.1. Double sampling is not required during finetuning as we are finetuning only for a particular task.

Figure 5.13 compares the average regret and the ranking graph for the regression and ranking losses for the transfer case. The "-T" suffix in the legends signifies that the corresponding plot is for a transfer HPO case. As we can see here, the listwise ranking losses are far superior to other losses. It is also interesting to see that the average regret of the listwise losses is similar to the regression losses in the transfer case. To check the overall performance of the listwise loss models, we compare this model with the worst, medium, and best performing surrogates as sentinels for the study. The random selection strategy is the worst performing, whereas FSBO is the best performing SMBO surrogate. We use a GP surrogate to represent a surrogate with a medium performance. Figure 5.14 shows this ablation. The listwise loss function functions better than the state-of-the-art transfer HPO model "FSBO" in the first 15 to 20 evaluation steps.

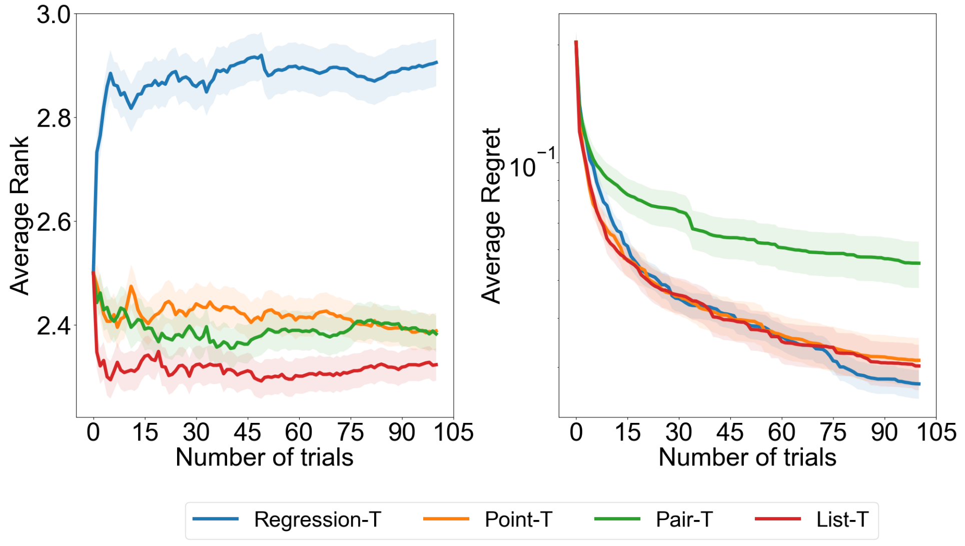

Figure 5.13: Comparison of Ranking losses with regression losses (Transfer case).

5.2.3 Study of weighted Listwise losses

In Section 3.6 we discussed the requirement for weighting for our problem. This section presents a short ablation to study its effect on our models. We do not separate the transfer and non-transfer cases here. We use inverse log weighting for the weighting in our case. From Figure 5.15, we see that both in the case of transfer HPO and non-transfer HPO, the weighting positively impacted the performance. Moreover, the models trained with weighted listwise losses are better at every optimization cycle step.

Even though the difference in the average regret is not significant in the 4 cases, the ranking graph differences are pretty significant. One thing to note here is that when we use listwise loss, we train and finetune with the same weighted loss. We do not use a different weighting (like the inverse linear weighting for training and inverse log weighting for the meta training). Moreover, we use the double sampling mechanism discussed above during meta-training.

5.2.4 Study of Surrogate Design: Addition of Deep Sets

We found in our research that our model could be improved by making it context-aware. We used an architecture with deep sets described in Section 4.3.1. As previously discussed, the latent result of the deep-set is used as

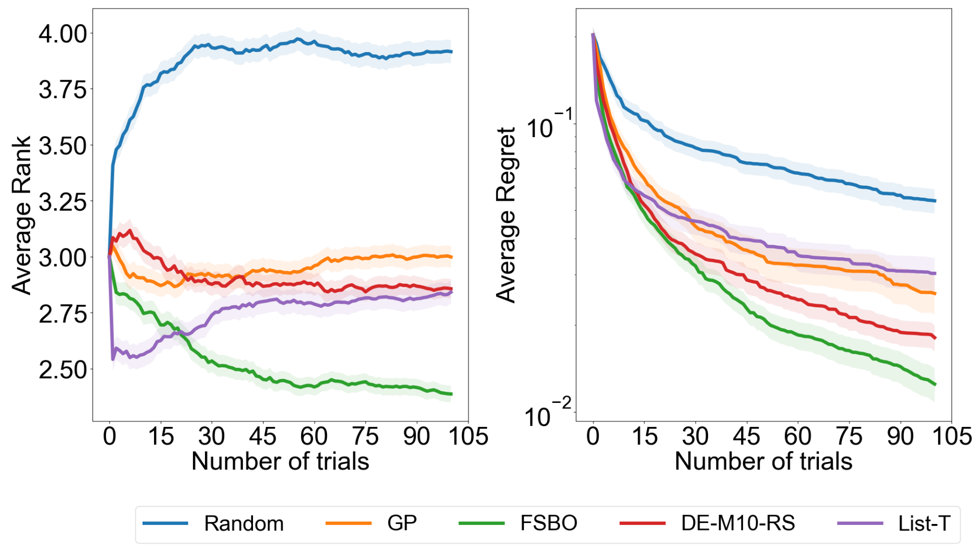

Figure 5.14: Comparison of listwise (transfer) loss with baseline HPO surrogates.

a context for the scoring model. To meta-train this architecture, we needed to change the training and sampling mechanisms.

For sampling, we used a support set of data points, i.e., $\{X_s, y_s\}$ that is 20% of the actual batch size. The concept of double sampling is also used in this case. For each double sampling, we sample first the task. Then the support points are sampled without replacement from this task. Subsequently, we sample the query points X_q based on the batch size and list size (By default, 100 for both). During meta tuning, we must also have a support and query set for training. We again use 20% of the points as support points and the rest as query points during the finetuning process.

One major issue when training with deep sets is that we cannot train the neural networks separately. It is because of the common backbone in ensemble architecture. In order to keep the training as separate as possible, we initialize the neural networks separately. After that, we calculate the loss of each neural network separately with the same batch data. Then we aggregate the losses and back-propagate the aggregated loss through the whole architecture. Note that we do not aggregate the scores of the neural networks but the ranking losses obtained from each scoring neural network.

Figure 5.16 shows the results of this study. In the non-transfer case, the model becomes terrible. The primary reason for this is that the complexity of the architecture is very high, and it needs a considerable amount of data to

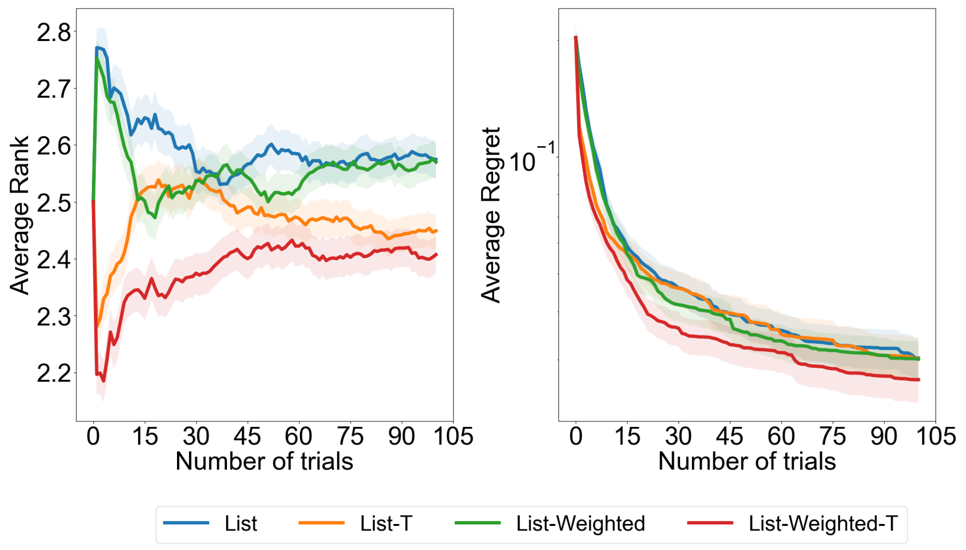

Figure 5.15: Comparison of listwise losses with theg weighting strategy.

train. This data is absent during the HPO optimization cycle. However, in the case of transfer learning, we see an enormous improvement when we use deep sets. It makes intuitive sense because each task is different from the other. The embedding of this information, or rather the difference in information, helps the model distinguish the tasks and pre-condition the learning process.

We compare the best results obtained so far with the sentinel performances of GP, FSBO, and Random search. Figure 5.17 illustrates this. Two interesting observations can be made from our results. First, our model performs significantly well in the first evaluation steps, and the performance later deteriorates w.r.t to state-of-the-art results. Hence, this model can be used in any transfer HPO case with limited evaluation steps due to the computation budget. Secondly, the average regret is also significantly better in the initial phase. Hence, we can unequivocally state that our proposed model is a good candidate to consider in any transfer HPO problem.

We also present in Figure 5.18 the critical rank graphs for reference.

5.2.5 Study of the effect of Finetuning

There are two ways of using the meta-trained ranking loss surrogate model. One way is to use it directly without changing it in the optimization process. The other is to finetune it at every evaluation step of the optimization process. We have always used the latter in our ablation studies (or answers to research questions) until now. In this section, we will see how the performance changes

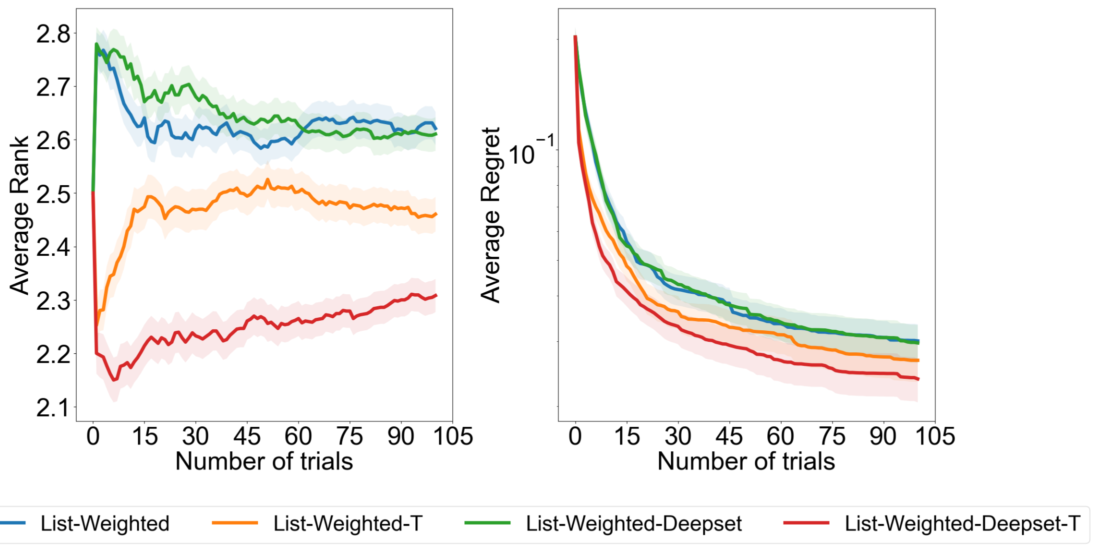

Figure 5.16: Comparing the effect of adding Deep Sets to the proposed model.

when no finetuning is done during the optimization process. We take the best model so far to study how finetuning affects performance. Then we run the optimization process with and without finetuning and plot the ranking and regret graphs.

If we do not use any finetuning, selecting the HP configurations for the evaluation process boils down to ranking the set of all unknown HP configurations. Then we select the ranked HP configurations one by one from this ranked list. Finetuning can be understood as a more sophisticated form of the optimization process in which the ranking loss surrogate tries to learn/understand the context from the known HP configurations. Figure 5.19 shows the graphs of models with and without finetuning.

From Figure 5.19 one can see that finetuning does help make the model get better results in the long run. The "-Raw" suffix signifies that no finetuning is performed using the trained model. One fascinating result we see is that without any finetuning, the ranking loss surrogate shows good performance in the first evaluation steps. This is because the surrogate model learns the best HP configuration found in the training metadata during meta-training. Hence it selects these HP configurations in the first evaluation steps. It makes intuitive sense because this behavior is shown by experts who try out the best HP configuration seen in their experience in the first evaluations. On average, this strategy works very well; hence the first evaluations were good for the non-finetuned surrogate model. However, as the optimization steps proceed,

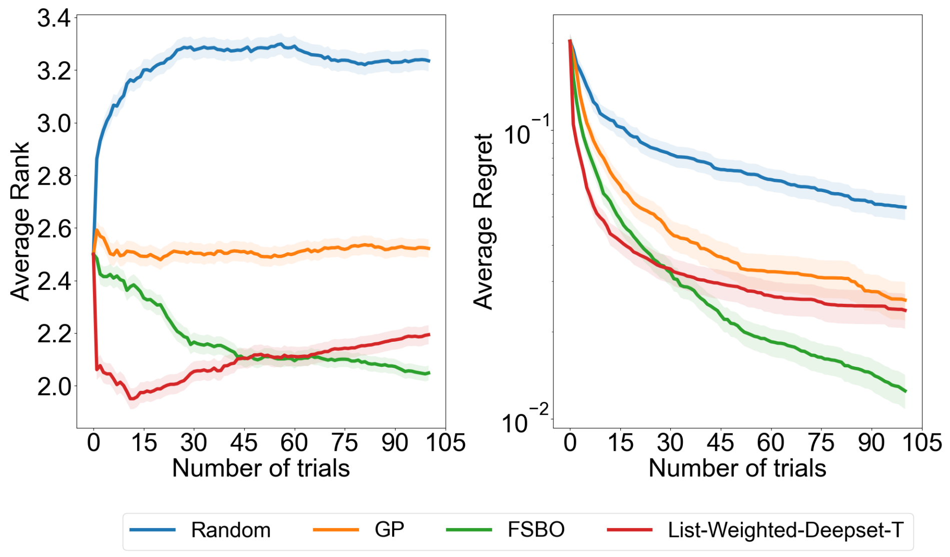

Figure 5.17: Comparison of the best proposed model with other baseline HPO surrogates.

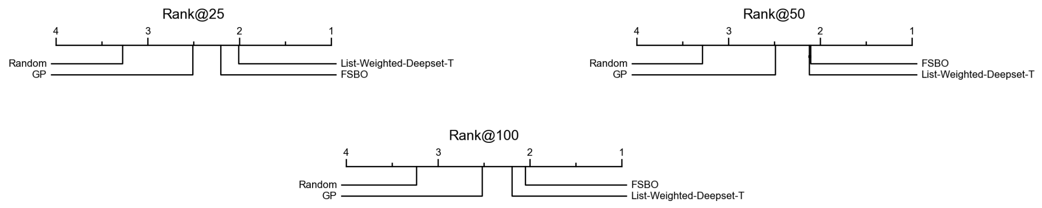

Figure 5.18: Critical rank graph of the listwise weighted transfer learning with deep sets with sentinel performances

the surrogate can condition its selection on more data. This conditioning is only done when the model is finetuned. Hence the finetuned surrogate works better in the long run.

5.2.6 Comparing results with SOTA surrogates

After analyzing our proposed model, one natural question is how it performs against the state-of-the-art HPO surrogates. We compare our results with the best-performing transfer and non-transfer HPO surrogates to answer this question.

Figure 5.20 compares the results with some prominent state-of-the-art non-transfer HPO surrogates. This figure shows that the non-transfer version of

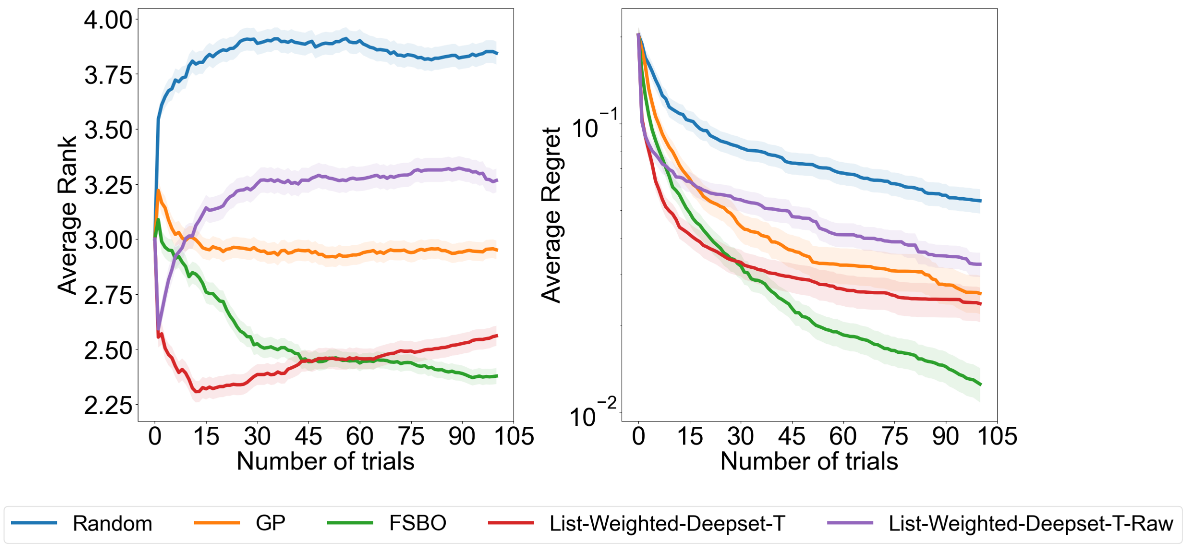

Figure 5.19: Effect of finetuning on surrogate performance.

the proposed model shows comparable performance to the Gaussian Process surrogate. This is the case both in the rank graph and the average regret. We also see that the proposed model performs very well in the initial few optimization steps. This can be seen in the critical graphs shown in figure 5.21. This property is also seen in its transfer variant.

Figure 5.22 compares the results with other state-of-the-art transfer HPO surrogates. As we see from this figure, the strength of using the proposed model is that it outperforms the state-of-the-art results in the first evaluations in the optimization procedure. Moreover, it outperforms RGPE, which also uses the concept of ranking making our model unequivocally better performing than other ranking methods. This can also be seen in the critical rank graphs shown in Figure 5.23.

5.2.7 Final ablation

Finally, in this section, we show the ablation of all the different ranking loss surrogates that were used in the previous ablations. Figure 5.24 shows this ablation. In this figure, we can see that making the model a transfer model, weighting the ranking loss, and adding deep sets all significantly improved the proposed model's performance. The biggest strength of the proposed model is that it is excellent in the initial 30-40 steps of the optimization process (compared to state-of-the-art HPO surrogates). This is crucial for a sequential process like SMBO because selecting a good HP configuration in a particular

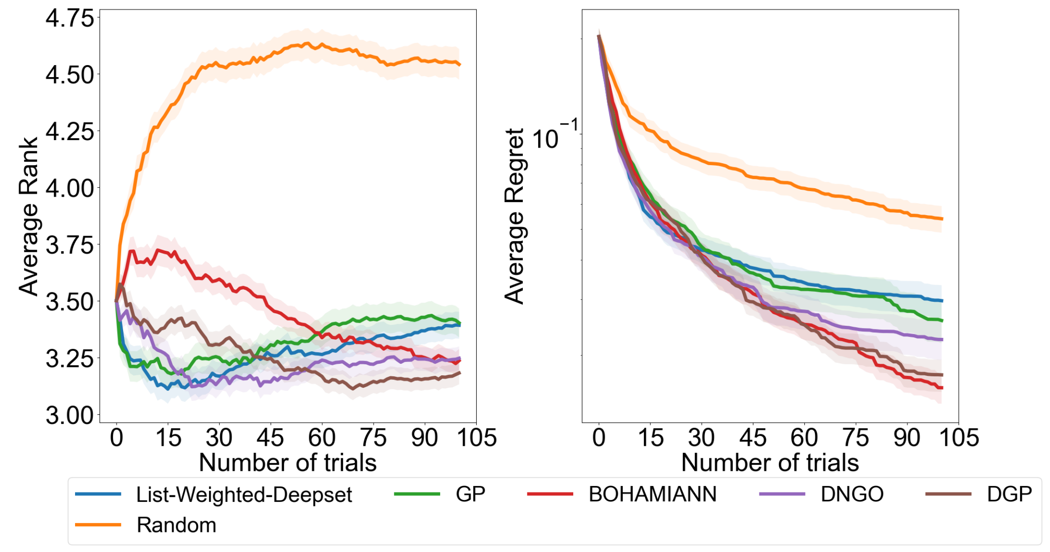

Figure 5.20: Comparing our results against SOTA non-transfer HPO surrogates.

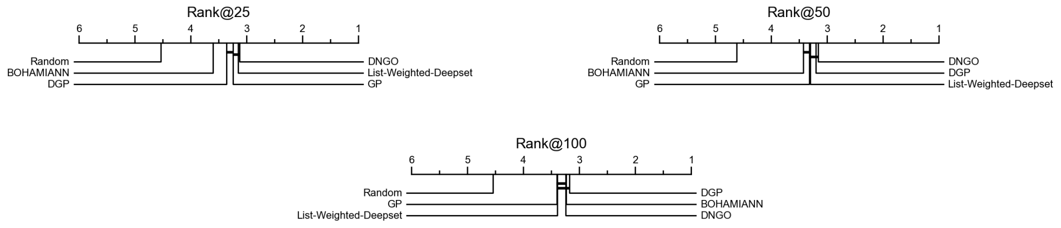

Figure 5.21: Critical rank graphs of the proposed model with SOTA non-transfer HPO surrogates.

step affects the selection of subsequent HP configurations. The second strength of the model is that there is a possibility of reducing negative learning from occurring because of the context encoded using the deep sets.

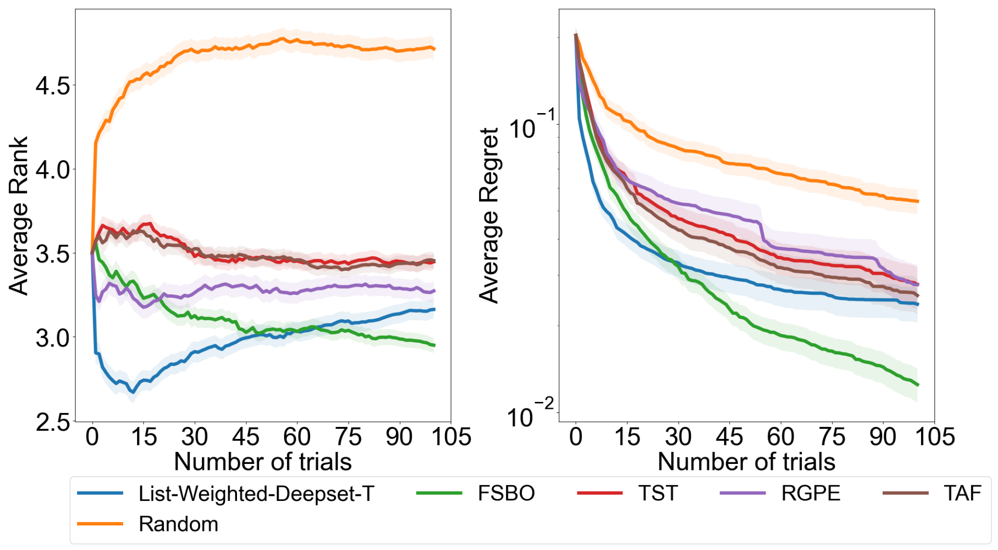

Figure 5.22: Comparing results against SOTA transfer HPO surrogates.

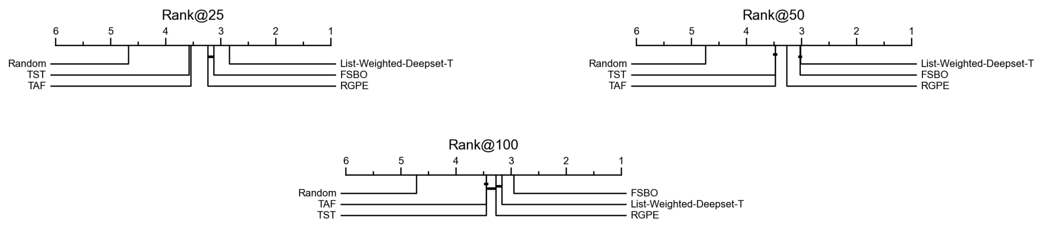

Figure 5.23: Comparing critical rank graphs against SOTA transfer HPO surrogates.

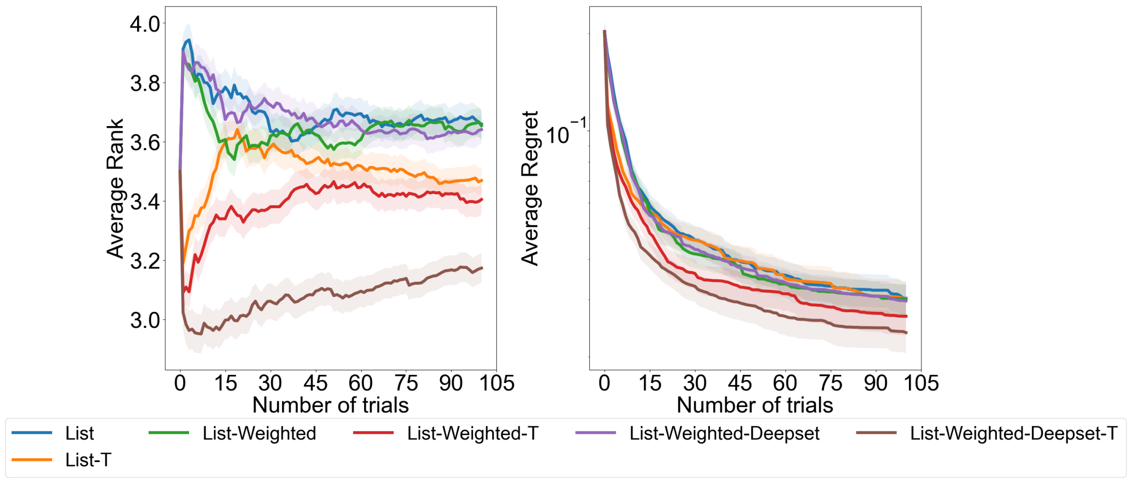

Figure 5.24: Ablation of all ranking loss surrogate models studied in this thesis.

Chapter 6

Conclusion

In this thesis, we studied how to use the concept of ranking with Sequential Model-Based Optimization (SMBO) to improve hyperparameter optimization. The HPO method we researched was a transfer learning method with two canonical parts: ranking loss functions and context-aware HPO surrogates. Firstly, we formulated the problem of selecting HP configurations within the SMBO algorithm as a ranking problem. Then, we researched the effects of using various ranking losses to train and finetune the HPO surrogate. We also studied the effect of adding weighting to the ranking loss. Finally, we integrated deep sets into the surrogate model architecture to make it context-aware.

We found in our study that the loss function used to learn an HPO surrogate significantly affects its performance. Among the ranking loss functions, listwise ranking loss showed the best performance during the HPO optimizations. After adding weighting to the listwise ranking loss function, the proposed model's performance further improved. Adding deep-sets to the surrogate made its performance on par with the performance of state-of-the-art transfer HPO methods. However, the results were the best only when we used the ranking loss surrogate in the transfer learning scenario.

One of the key takeaways of this thesis is that it is crucial to differentiate the ranking space from the output space. The ranking loss surrogates performed well because they learn and function in the ranking space rather than the output space. Hence we also conclude that working in the ranking space is better than working in the output space. Moreover, working in the ranking space is helpful in a problem domain like the HPO. The output ranges of the evaluated HPO objective functions may differ for different tasks. By working in the ranking space, we avoid normalizing meta-data values usually done by other transfer HPO methods.

We conclude that a context-aware model trained with a weighted listwise loss function has the potential to give excellent hyperparameter optimization results. We encourage the reader to use this model for any HPO. We also believe that doing more research on listwise losses in the HPO domain could give promising results. The concepts developed in this thesis and any further research on listwise losses could also aid other areas like Information Retrieval, which heavily uses ranking losses. In the rest of this chapter, we discuss the proposed model's advantages and limitations. Finally, we end this report by suggesting some interesting future research directions.

6.1 Advantages

First and foremost, one of the main advantages of the proposed surrogate model is that it is a transfer HPO surrogate. Training the surrogate on previously available metadata makes the surrogate perform significantly better than other non-transfer HPO surrogates.

Secondly, we also found that the sampling mechanism required for the listwise loss function had the upper hand compared to other loss functions. Because a set has a massive number of possible subsets, the number of data instances sampled for the listwise loss is very high. For example, if we have a metadata set of 100 HP observations, i.e., 100 ($\mathbf{x}$,y) pairs, and a training list size of 15, then a total of $\binom{100}{15}$ training lists can be sampled from the metadata. This is a significant advantage because, during an HPO optimization, the known HP configurations and their evaluations are very scarce. We can also increase the surrogate model's complexity without worrying much about the available data.

Another significant advantage of the proposed method is that training using ranking losses inherently dampens any uncertainty of the HPO objective function's output. This is because ranking losses work in the ranking space. The range of the actual scores in the metadata is irrelevant to the training procedure. As the order of the known evaluations is the only relevant thing, minor variations in the objective function's output do not affect the loss's training. This also allows us to manipulate the range of our surrogate's scores based on our requirements.

6.2 Limitations

Even though the proposed method gives state-of-the-art results, it has some limitations that need further study. In this section, we critically analyze the limitations of the proposed model.

We found in our research that for some search spaces, the ranking loss surrogate's validation errors did not reduce during its training. In fact, they increased. Figure 6.1 illustrates this for the search space id 5527. This deterioration of validation loss signifies that negative transfer learning is occuring [45]. Negative transfer learning occurs when the source tasks set differs from the target tasks. In this example, the meta-training task is different from the meta-validation task.

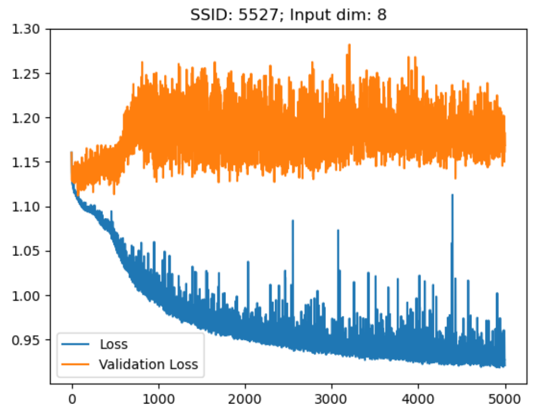

Figure 6.1: Training and validation loss curves obtained for the training of search space 5527.

This problem should occur only in the cases where the model is context-free. Even though our ranking loss surrogate is made context-aware by integrating deep sets into its architecture, we still have the problem of negative transfer learning. One remedy for getting around this problem is to finetune the model with more epochs. Nevertheless, we cannot guarantee that the finetuning will re-learn from the small number of observed HP configurations during the optimization cycle. Since the testing/evaluation task may be similar to the training task, we could also not do any early stopping during meta-training.

Another disadvantage of the proposed model is that we ignore the sorting functionality necessary to complete the ranking process during our surrogate training. This is because sorting is non-differentiable. Due to this issue, the complete ranking process is not learned with the listwise loss. This means that we do not solve the complete ranking problem at hand conceptually.

Our loss function uses the exponentiation function as the strictly increasing positive function. The usage of the exponentiation function puts a practical restriction on the range of the relevance scores. This is because standard floating-point implementations in computer languages only have a fixed range of values they can represent. These floating-point implementations may not have the range to map the actual HPO objective function's range completely. Hence, more research is required to find out better strictly increasing functions.

Another problem with our model is that the meta-trained model may be biased towards one task due to the larger availability of data from this task. Double sampling helps us eliminate this issue by sampling all tasks with equal probability. However, the consequence of double sampling is that the meta-training overfits the surrogate to the tasks with lesser data because their data is overused during training. More studies must be conducted to find a mechanism for fair metadata sampling across tasks.

6.3 Further research work

During our thesis, we found various topics that could be taken up for future research. For example, in section 4.1.2 we proposed that further research on acquisition functions is necessary due to the intricacies of the ranking space. Similarly, we already know from section 6.2 that the listwise loss function does not optimize the sorting function. Interestingly, the inclusion of sorting functionality in a loss function has been proposed and studied in Swezey et al. [43]. One research direction would be to incorporate the ideas of PiRank in order to see if there is any improvement in the proposed model.

Lakshminarayanan et al. [24] propose a new loss function that inherently learns an output's mean and variance. Our idea, on the other hand, predicts definite scores. After that, it predicts the mean and variance in the ranks induced by the scores. One exciting idea we could not explore due to the scope of the thesis is the integration of the loss function proposed by Lakshminarayanan et al. and ranking losses. Doing a theoretical and empirical analysis of such an integration could be one research direction. It would be interesting to see the effect of this integration on HPO problems with less metadata because Deep Ensembles by themselves perform relatively well in a silo.

This thesis has primarily focused on how to select HP configurations from a set of discrete HP configurations. Hence the model does not deal with continuous search spaces directly. For working with continuous HP search spaces, we can divide the space (in the Euclidean sense) into hyper-volumes and use 1 HP configuration to represent the whole hyper-volume. Then we may select the best region, subdivide the space, and continue the process until we reach a particular granularity. This method is just a workaround to the main problem, and we also cannot guarantee that an optimum will be found. Hence, further research must be done on formulating the ranking loss function for continuous search spaces.

Lastly, novel HPO surrogates that this thesis has not explored can be implemented. It would not only further the research empirically, but also ideas used in other baselines could be incorporated to improve the model.

Bibliography

[1] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. 2017.

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, feb 2012.

[4] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[5] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 129–136, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Huadong Chen, Shujian Huang, David Chiang, Xinyu Dai, and Jiajun Chen. Top-rank enhanced listwise optimization for statistical machine translation, 2017.

[7] Wei Chen, Tie-yan Liu, Yanyan Lan, Zhi-ming Ma, and Hang Li. Ranking measures and loss functions in learning to rank. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[8] David Cossock and Tong Zhang. Statistical analysis of bayes optimal subset ranking. *IEEE Transactions on Information Theory*, 54:5140–5154, 11 2008.

[9] David Cossock and Tong Zhang. Statistical analysis of bayes optimal subset ranking. *IEEE Transactions on Information Theory*, 54(11):5140–5154, Nov 2008.

[10] Matthias Feurer. Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles. 2018.

[11] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Using meta-learning to initialize bayesian optimization of hyperparameters. In *MetaSel@ECAI*, 2014.

[12] Luca Franceschi, Riccardo Grazzi, Massimiliano Pontil, Saverio Salzo, and Paolo Frasconi. Far-ho: A bilevel programming package for hyperparameter optimization and meta-learning, 2018.

[13] Yoav Freund, Raj Iyer, Robert Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 01 2003.

[14] Ethan Goan and Clinton Fookes. Bayesian neural networks: An introduction and survey. In *Case Studies in Applied Bayesian Data Science*, pages 45–87. Springer International Publishing, 2020.

[15] Taciana A. F. Gomes, Ricardo B. C. Prudêncio, Carlos Soares, André L. D. Rossi, and André Carvalho. Combining meta-learning and search techniques to select parameters for support vector machines. *Neurocomput.*, 75(1):3–13, jan 2012.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, page 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

[19] Frank Hutter and Joaquin Vanschoren, editors. *AutoML tutorial at NeurIPS 2018*. online, 2018.

[20] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization, 2015.

[21] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[22] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. Hands-on bayesian neural networks – a tutorial for deep learning users, 2020.

[23] Abdus Salam Khazi and ddaedalus. Are the q-values of dqn bounded at a single timestep? `https://ai.stackexchange.com/questions/31595/are-the-q-values-of-dqn-bounded-at-a-single-timestep/31648#31648`, 2021.

[24] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles, 2016.

[25] Yanyan Lan, Yadong Zhu, Jiafeng Guo, Shuzi Niu, and Xueqi Cheng. Position-aware listmle: A sequential learning process for ranking. 09 2014.

[26] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks, 2018.

[27] Ping Li, Qiang Wu, and Christopher Burges. Mcrank: Learning to rank using multiple classification and gradient boosting. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[28] Jelena Luketina, Mathias Berglund, and Tapani Raiko. Scalable gradient-based tuning of continuous regularization hyperparameters. 11 2015.

[29] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning, 2015.

[30] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[32] Massimiliano Patacchiola, Jack Turner, Elliot J. Crowley, and Amos Storkey. Bayesian meta-learning for the few-shot setting via deep kernels. In *Advances in Neural Information Processing Systems*, 2020.

[33] Sebastian Pineda-Arango, Hadi S. Jomaa, Martin Wistuba, and Josif Grabocka. HPO-B: A large-scale reproducible benchmark for black-box HPO based on openml. *Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2021.

[34] Sebastian Pineda-Arango, Hadi S. Jomaa, Martin Wistuba, and Josif Grabocka. HPO-B: A large-scale reproducible benchmark for black-box HPO based on openml. *CoRR*, abs/2106.06257, 2021.

[35] Przemysław Pobrotyn, Tomasz Bartczak, Mikołaj Synowiec, Radosław Białobrzeski, and Jarosław Bojar. Context-aware learning to rank with self-attention, 2020.

[36] Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. Letor: A benchmark collection for research on learning to rank for information retrieval. *Inf. Retr.*, 13:346–374, 08 2010.

[37] Ramon Quiza and J. Davim. *Computational Methods and Optimization*, pages 177–208. 01 2011.

[38] Matthias Reif, Faisal Shafait, and Andreas Dengel. Meta-learning for evolutionary parameter optimization of classifiers. *Machine Learning*, 87:357–380, 06 2012.

[39] Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. Scalable hyperparameter optimization with products of gaussian process experts. In *ECML/PKDD*, 2016.

[40] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams.

Scalable bayesian optimization using deep neural networks. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2171–2180, Lille, France, 07–09 Jul 2015. PMLR.

[41] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust bayesian neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 4141–4149, Red Hook, NY, USA, 2016. Curran Associates Inc.

[42] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[43] Robin Swezey, Aditya Grover, Bruno Charron, and Stefano Ermon. Pi-rank: Scalable learning to rank via differentiable sorting. 2020.

[44] Jie Wang. An intuitive tutorial to gaussian processes regression, 2020.

[45] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big Data*, 3(1):9, May 2016.

[46] Wikipedia contributors. Bessel's correction — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bessel%27s_correction&oldid=1073637562`, 2022. [Online; accessed 8-May-2022].

[47] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 370–378, Cadiz, Spain, 09–11 May 2016. PMLR.

[48] Martin Wistuba and Josif Grabocka. Few-shot bayesian optimization with deep kernel surrogates, 2021.

[49] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Two-stage transfer surrogate model forźautomatic hyperparameter optimization. In *European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 9851*, ECML PKDD 2016, page 199–214, Berlin, Heidelberg, 2016. Springer-Verlag.

[50] Martin Wistuba, Nicolas Schilling, and Lars Schmidt-Thieme. Scalable gaussian process-based transfer surrogates for hyperparameter optimization. *Machine Learning*, 107:43–78, 2017.

[51] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. Listwise approach to learning to rank: theory and algorithm. In *ICML '08*, 2008.

[52] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets, 2017.

Appendix A

Controlling the output range

Lets take the listwise loss function L_{mle} which is nothing but maximum log-likelihood estimation. In our surrogate models, we use the strictly positive increasing function exp as proposed by Fen et. al. [51]. Hence, our L_{mle} becomes

$$L_{mle} = -\sum_{j=1}^{k} \log \frac{\exp(s(\pi_j^*))}{\sum\limits_{t=j}^{k} \exp(s(\pi_k^*))} \tag{A.1}$$

Even though theoretically, the score range does not need to be restricted, we have practical limitations due to implementation constraints of floating-point numbers. Due to the exponentiation in our loss function, if the score range is too high, the values in equation A.1 may overflow. On the other hand, if it is too small, the values may underflow. In both these cases, we are bound to get NAN (Not A Number) exceptions when executing our implementations. We would also get similar issues for the pairwise loss function because we use exponentiation for calculating the probability, as shown in equation 3.8. One way to eliminate this issue is to use the log-sum-exp trick. This trick is used in the ListMLE implementation given by Pobrotyn et. al. [35] and we use this implementation in our thesis.

However, if the output score is used anywhere other than ranking, we would need to control its range. We can do this by passing the output of the deep neural network through a tanh function. If we would like to strictly limit the range of our function between $[-k, k]$, then we can pass the output score through the following function

$$k * \tanh(\alpha * s(\mathbf{X})) \tag{A.2}$$

Here α is the smoothness factor which is inversely proportional to the smoothness of the tanh graph [23]. Figure A.1 shows how to vary the smoothness and range of the output using α and k respectively.

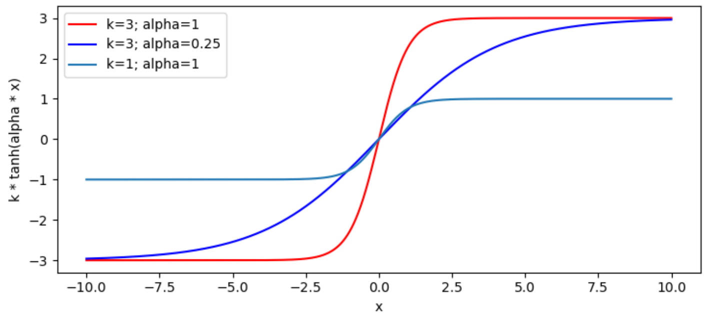

Figure A.1: Effect of varying k and α in equation A.2.

Appendix B

Case study: Inverse mapping

This case study examines how well an ML model performs if trained using the ListMLE loss function. For this, we consider a toy problem of sorting a list of real numbers in descending order. The main problem that we try to study here is - Is it possible to train a model using the ListMLE loss function such that it learns inverse mapping of points on a number line.

Consider numbers sampled from the range $[k, p]$ where $k, p > 0$ and $k, p \in \mathbb{R}$. If we take 2 numbers $x_1, x_2 \in [k, p]$, we need to learn a mapping $s : x \mapsto \mathbb{R}$ such that $s(x_1) \leq s(x_2)$ when $x_1 \geq x_2$. Consequently, we could sort the numbers based on the output score of the model to obtain a descending sorted order.

Consider a list $l = \{x_1, x_2, ..., x_n\}$ where $x_i \in [1, 100]$. Let $s(x \mid \theta) \mapsto \mathbb{R}$ be our scoring function parametrised by θ. Here, one list contains n data points sampled from $[1, 100]$. Then the loss function we would use to learn our scorer is

$$\underset{\theta}{\operatorname{argmin}} \quad L_{mle}(s(l \mid \theta), -k * l) \tag{B.1}$$

where $k \in \mathbb{R}$. Note that the second parameter of listwise loss function is scale-invariant; hence scaling the list does not affect the loss output (Section 3.5.2). To keep the output score in a defined range, we utilized the range controlling mechanism discussed in Appendex A.

We sample the testing data from the following ranges to evaluate our learned model:

- **In range**: The Same range as seen during training i.e $[1, 100]$.

- **Out of range**: A completely different range as seen during training i.e $[-100, -1]$.

- **Hybrid range**: A range that is partially seen and partially unseen i.e $[-50, 50]$.

During training, a batch size of 100 lists was used. The training was done with values sampled inside the $[1, 100]$ range. For testing, we sample data from inside, outside, and hybrid ranges. A batch of 1000 lists was sampled for testing. We sort these lists according to their respective scores output by our learned model. After that, we check the percentage of the correctly sorted lists. This accuracy of our model is the fraction of 1000 lists that were sorted. We report the average of 5 runs in Table B.1.

Training Epochs	**List size**	**In range Acc.**	**Out of range Acc.**	**Hybrid range Acc.**
1000	3	0.99	0.27	0.40
100	3	0.77	0.29	0.11
100	30	0.80	0.79	0.46
100	100	0.99	0.0	0.39
1000	100	1.0	0.71	0.48

Table B.1: Learned model Accuracies.

We find from the tabulated results that training the model by running a higher number of epochs is vital. Moreover, bigger list size is crucial to comprehensively learning the objective function. The larger list size will make our model work well when the input is from the same distribution seen during training. Moreover, it also performs well when the input comes from a hybrid range. These are good properties to have in any machine learning model.

The loss function is not weighted in this toy example because sorting requires each data point to be in its correct location. Therefore, our problem is not a direct generalization of this toy example. Moreover, the toy example's input domain is relatively simple compared to real-world problems. Nevertheless, we take these results into account to decide on the list size for training our proposed ranking surrogate model.