Master's Thesis

Hyperparameter Optimization using Ranking Loss Surrogates

Albert-Ludwigs-University Freiburg
Representation Learning Department

Abdus Salam Khazi

May 21, 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

Abstract goes here

Contents

Chapter 1

Introduction

The performance of any machine learning model is sensitive to the hyper-parameters used during the model training. Instead of using a new model type, it is more helpful to tune the hyper-parameters of an existing model to improve its performance. Learning the best hyper-parameter for an ML model is called, Hyperparameter optimization (HPO in short).

The true evaluation of a hyperparameter optimisation objective function is computationally very expensive. Researchers have tried to get around this problem by proposing model based HPO algorithms. In these methods, the true objective function is modelled by a cheap surrogate function of high representational capacity. For example, the Sequential Model Based Optimization (SMBO) [3] algorithm is a very important algorithm that iterates between learning a model given a few HP configuration evaluations and using the model to propose the next candidate to evaluate.

This thesis studies various existing approaches to HPO and proposes a new idea for the same using the concept of ranking. The proposed idea in this thesis is called **Hyperparamter Optimization using Ranking Loss Surrogates**. The results obtained using this model are compared against the state-of-the-art results obtained using models like FSBO, RGPE, TAF, and others.

1.1 Objective

The aim is to study ranking loss surrogates in the context of Hyperparameter optimization.

1.2 Overview

This sections contains the overview of the paper and how the thesis report is organised.

Chapter 2

Related Work

In this chapter, the hyper-parameter optimization problem is first defined concretely. Then, the various approaches already used to do the HP optimization are discussed. Subsequently, some important concepts that the thesis uses to build the proposed model are also discussed.

2.1 Hyper Parameter Optimization

To find out the best hyper-parameter for any machine learning model m, we must first quantify a given hyper-parameter 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 hyper-parameter search space.

This function $f(\mathbf{x}) \mapsto \mathbb{R}$ is evaluated in the following chronological steps:

1. Using a given hyper-parameter 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, E.g. learning the weights and biases of a Deep Neural Network. We use the training data to learn this model.

2. The validation data is passed through $m_{\mathbf{x}}^{trained}$ to obtain the required results. These results are evaluated based on an evaluation criterion 'eval'. This criterion is different for different problems, e.g. Regression,

Classification, etc. The result of this evaluation is a real-value that gives a score for the configuration $\mathbf{x}$.

Hence the function $f(\mathbf{x}) \mapsto \mathbb{R}$ can 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:

$$\underset{\mathbf{x}}{\text{argmin}} \;\; f(m_{\mathbf{x}}^{trained}(\text{Data}_{\text{val}})) \mapsto \mathbb{R} \quad \forall \mathbf{x} \in \mathbb{S} \tag{2.1}$$

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 [41].

HPO Constraints

Hyperparameter optimization is different from other optimization methods because it has different constraints [1]. It is because of the peculiar properties of the hyper-parameter search spaces. Finding out the correct hyper-parameter setting is generally not feasible using a brute-force approach (trying out all possible combinations of hyper-parameters) because the search space itself has many dimensions, and the search space may be continuous. More specifically, some of the important constraints of this optimization problem are:

1. The evaluation of a given HPO configuration is computationally expensive.
2. It is a non-convex optimization problem.
3. The process of getting $m_{\mathbf{x}}^{trained}$ from m is stochastic hence the value $v_{\mathbf{x}}$ is noisy.
4. 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 3 only makes sense if we have 3 or more layers in a neural network.
5. The search space is hybrid in terms of continuity. Some of the dimensions (variables) may be continuous, while others may be discrete. Using a gradient method is hence not trivial.

To deal with the constraints of HPO problems, researchers have used different strategies for developing HPO algorithms and models. Some of the more prominent approaches are Black-Box HPO, Multi-fidelity HPO and Online HPO.

2.1.1 Black-box HPO

In this approach the HPO objective function f in equation 2.1 is treated as a blackbox. The problem is thus generalised to finding a global optima of f. Some of the straightforward black-box HPO methods include Manual Search, Grid Search and Random Search. The optimization technique called Bayesian optimization gives us a more sophisticated mechanism to deal with this problem.

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 that 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 in which we divide the search space into grid points similar to ones 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.

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) shows 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 other HPO models.

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 configuration well. Moreover, these

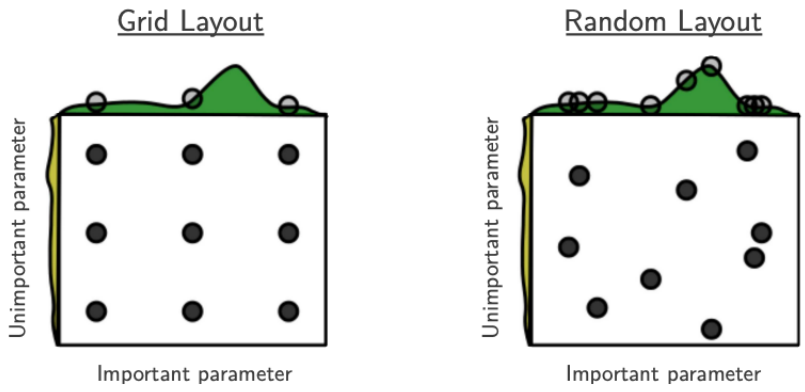

Figure 2.1: Illustrates Grid search and Random search in the case where 2 parameters are not equally important. Adpated from [4].

methods are computationally expensive. The reason is that they do not use surrogate evaluators and hence train and evaluate the whole model. Also, these search methods give us optimal HPO configurations only by chance.

Bayesian Optimization

Bayesian optimization tries to solve both computational costs and noisy evaluations of our objective function f. It does this by building a model of the HPO objective function. This model is called a surrogate function. Bayesian optimization uses known evaluations as its data to build the surrogate model. The data is of the form {x, f(x)} pairs. The surrogate model is a probabilistic model. Hence, it also learns about the noise in the evaluations of the objective function.

The core procedure of the optimization process is the following:

- 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
- Use the surrogate to sample the next best HPO configuration x' using a function known as acquisition function. Evaluate f(x').
- Append (x', f(x')) to D and repeat the process.

The above process repeats till the computational resources are finished (here time) or we find an acceptable HPO configuration. This procedure is

also called SMBO (Sequential model-based Optimization). The procedure alternates between collecting data and fitting the model with the collected data [15].

Hence, there are two essential components of Bayesian optimization:

- Probabilistic surrogate model of the objective function. Some surrogates are discussed in detail in the subsequent sections in this chapter.
- The acquisition function

Acquisition functions

The acquisition functions used in the bayesian optimization need to do balance exploitation of information from the known/observed data points and exploration of unknown data points in the domain. The following functions are some of the most prominent acquisition functions found in the literature [37]

- **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 gives the probability with which we can get 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}.

Expected Improvement acquisition function is used all around the thesis in order to do a fair comparison of the models and algorithms.

2.1.2 Online HPO

Traditionally to evaluate and select a new HP configuration, the objective function f is fully evaluated. In the most general sense this can be applied to both discrete and continuous HP search spaces. Some advanced methods have proposed even gradient based HP optimization.

For example, Maclaurin et al. [26] proposed a relatively cheap method to obtain hyper gradients i.e gradients of hyper parameters with respect to the whole objective function f. Further, Franceschi et al. [11] formulated the whole HPO problem as a bi level optimization problem [17].

In all these methods as the hyper parameters and the parameters of the model are being learnt disjointly, they can be referred to as offline HPO. The idea of online HPO is that it tries to evaluate and update the HP configuration during the training of the model itself.

Sometimes only a single hyper parameter is learnt online. For example, Baydin et al. [2], proposes the online learning of the learning rate. They target this hyper parameter because it is the single most important hyper parameter. In other times all parameters may be learnt. For example, Luketina el at. [25] proposes to interleave the updating of training parameters and hyper parameters.

2.1.3 Multi-fidelity HPO

If we treat our HPO objective function f as a black box function, we would need to evaluate equation 2.1 fully. This is prohibitively expensive as evaluating a single HP configuration may take days [16]. Multi-fidelity hyper parameter 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 a fidelity as it copies or reproduces the true objective function upto some degree. For example, a fidelity could be 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 etc. The idea is to trade off between the performance of the approximate function and its optimization accuracy such that we get best selection of HP using the least compute power. Since the true evaluation of HP configuration using f is not done, it is an approximate optimization technique.

The technique does not belong to the black box HPO domain. This can be understood very clearly if we take the example of the "few epochs" fidelity. Clearly, the optimization algorithm looks into the training process the learns the objective function to prematurely exit it if need be. Thus getting a feedback from within the "black box" of the HPO objective function.

The reason this technique is called multi-fidelity is because it may use different fidelities within the optimization process to get the best HP configuration. For example, while using successive having technique [18] for HPO one can start with a given "budget fidelity" for example - defined number of epochs or defined amount of training time. At each step of the optimization process, the budget is doubled and the worst performing HP configurations are eliminated for the next step. Hence it uses "multiple" fidelities during the optimization process.

2.2 Transfer-learning for HPO

The HPO methods discussed so far, evaluate each HP configuration from scratch. 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 mechanism that uses this concept is called a transfer learning method [38].

This concept of transfer learning can be utilized to accelerate HPO. For example, Gomes et al. [13] propose to meta learn a set of good HP configurations for the SVM. They propose that if we learn HP configurations that worked well previously, then there is high chance of finding that a new SVM model also works with these parameters well. This concept of warm starting the HP optimization is also proposed by Reif et al. [34] albeit for genetic algorithms.

There are 2 broad ways to transfer knowledge for doing HP optimization - By learning surrogates or by doing warm starting of initial configurations.

2.2.1 Warm starting

Before running any HPO method, normally experts study the data being used to train the given model. Based on their study 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. This is essentially automated by the warm starting method [10].

In the above mentioned papers [13], [34] the authors use the meta learnt HP configurations as starting points for finding the optima in the HP response surface. Additionally this idea was also proposed for the SMBO optimization algorithm by Feurer et al. [10].

2.2.2 Meta-learning of surrogates

HPO models that use surrogates like SMBO, have an additional gateway through which previous knowledge can be injected. The idea is to meta-learn the surrogate using previously stored meta-data from similar tasks. This is used by Schilling et al. [35] in which they learn a collection of Gaussian models for each previously similar dataset due to computational constraints. They then use the collection of GPs as a surrogate for the SMBO procedure. Another flavour of this approach was proposed by Feurer et al. [9] proposed

the use of rank weighted GP Ensembles (RGPE) surrogates meta-learnt from previous meta data.

This meta learnt surrogate can be used in 2 different ways during the HPO optimization procedure. For example in SMBO, one could use it without any modification (aka fine tuning) to suggest the next best candidate to evaluate from a given HP configuration list. On the other hand the surrogate can be further meta trained (aka fine tuned) using the available target task meta data.

Using meta learning of surrogates, this thesis proposes a new surrogate model.

2.3 Types of Surrogates for BO in HPO

Hyperparameter optimization using Sequential Model Based Optimization (SMBO) [3] is a convenient method proposed in the literature. However, the performance of this method is heavily reliant on how well the surrogate (model in SMBO) models the true HPO objective. In this section, we discuss in details some of the powerful surrogates that may be used with SMBO.

2.3.1 Gaussian Processes

Gaussian processes [37] 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 problems such as hyperparameter optimization, where uncertainty estimation is essential. In this section, we briefly explain the Gaussian process regression intuitively.

Before we proceed, we need to understand normal (Gaussian) distributions. Consider a scalar random variable X that is distributed normally (a.k.a Gaussian distribution) 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 [37]. 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 variance, represented by Σ, is in the $R^{d \times d}$ space. It is

because the variance of all components in any valued vector random variable $\mathbf{X}$ should contain the following two types of variance

- Variance of a vector component w.r.t itself. d diagonal values of the matrix Σ represent this variance.
- Variance of each vector component w.r.t all other components. These variances are represented by the upper/lower triangular values in the matrix Σ.

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

The probability density function of a vector valued variable $\mathbf{X} \in \mathbb{R}^d$ with a mean $\boldsymbol{\mu}$ and covariance matrix Σ is given by [27]:

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

This equation defines the PDF of a multivariate normal distribution.

The core idea used in the Gaussian processes is that functions can be considered as 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. A function, hence, is nothing but a sample from $\mathbb{R}^\infty$. Unfortunately, functions sampled from $\mathbb{R}^\infty$ are too general and not useful by themselves.

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 $f(g) \approx f(x)$. It is mathematically represented by the following equation:

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

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

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. Using this kernel function, the values in the

required covariance matrix are populated. The smoothness of the sampled function f is controlled by the kernel in the GP process. 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**
- **Periodic Kernel**

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

$$Pr\left(\begin{bmatrix}\mathbf{y}\\ y^*\end{bmatrix}\right) = \mathcal{N}\left(m(\mathbf{X}), \mathbf{\Sigma}\right) \tag{2.2}$$

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 [37]. Due to the robustness of the GP process, we use this as one of the baselines in our thesis.

2.3.2 Random Regression Forest

The core idea of this model is to train a Random Regression Forest, using the known data as in any SMBO procedure [15]. Random regression forests are an ensemble of regression trees. This property is used to our advantage to predict the mean and the variance. The mean of the prediction of all the trees is the mean of the surrogate model. The variance in the prediction of all trees is the variance of the surrogate model.

The advantages of this model are

- It can handle both continuous and discrete variables trivially without any modifications to the model. The data splitting during training is done using any variable be it discrete or continuous.

- It can handle conditional variables, unlike Gaussian processes, by making sure that data is not split based on a variable till it is guaranteed that no conditionality is broken by the split.

2.3.3 Bayesian Neural Networks

2.3.4 Neural Networks

they don't work great, because they cant model uncertainty). Uncertainty can be modeled directly, or through ensembles.

2.4 Types of Losses

2.4.1 BO with GP uses negative log likelihood

It is not clear whether pointwise losses (regression as in GP) are the correct way to model HPO responses, because we only care for the minima regions (best performing configurations), and not for estimating all observations accurately.

2.4.2 Ranking loss could be the answer in modeling hp optima better

Pointwise

Pairwise

Listwise

2.5 Set-modeling with Neural Networks

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

To model this concept, one has to learn to represent and pre-condition knowledge from known tasks in a model. This makes the model context aware. If we consider a 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 which includes

- Representation of a Set
- Conditioning of our model on the Set 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 [43] and Set Transformers by lee et. al [23] are two of the interesting researches that we found in the literature. Since the sophisticated attention mechanisms used in Set transformers were unnecessary for us, we chose to use the more simple architecture used in Deep Sets. The next section discusses about these Deep Sets.

2.5.1 Deep Sets

Traditionally deep learning or machine learning models learn functions of the following format:

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

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

which can be thought of transforming objects from the input space to the output space.

For our problem of set latent representation, the input space changes to a space containing sets (each instance in the space is a set by itself). The output space however remains similar to either the regression case for regression tasks and classification case for classification tasks. Consider $\mathbb{X} = \{a, b, c, ...\}$ be a set containing all possible elements within the sets of the input space. Then the set representation problem can be defined as:

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

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

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

2 very important 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.

Permutation equivariance is another type of constraint that the deep set paper deals with. In this constraint the latent space should be equivariant or symmetric to the permutation of the objects in the input set. This creates issues for our problem because we don't care if our latent space is some symmetric mapping of the input space. We want the whole set to be mapped definitively to a latent embedding.

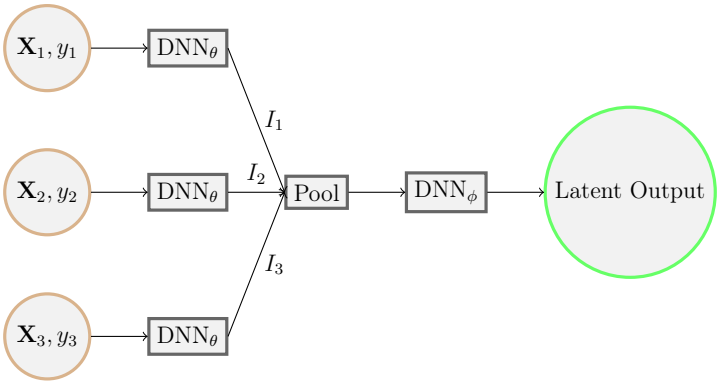

Figure 2.2: Skeletal of the architecture proposed in deep sets

Figure 2.2 illustrates the architecture proposed by the deep set paper. We use a set of only 3 objects in the Figure for simplicity. The architecture passes all the 3 inputs through the a deep neural network independently to get intermediate outputs I_1, I_2, I_3. To solve the permutation invariance constraint, any mathematical operation that is commutative and associative is applied on 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, max etc.

To solve the problem of cardinality constraint we can use the mean operator. Hence the pooling operation becomes:

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

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

Chapter 3

Background

In this chapter we discuss in detail the fundamental concepts one must grasp in order to understand the thesis work. First we talk about the problem of ranking and ranking losses. Thereafter, we discuss the modelling of uncertainty done using Deep Neural Network ensembles.

3.1 Rank Learning

Consider a set of objects $\mathbb{A} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ...\mathbf{x}_n\}$ where each $\mathbf{x}$ belongs to a domain $\mathbb{D}$. The problem of ranking is defined as finding an ordered list of objects in $\mathbb{A}$ such that an object $\mathbf{x}_i$ is ranked before $\mathbf{x}_j$ if $\mathbf{x}_i$ is more relevant/important than $\mathbf{x}_j$. To accomplish this objective, a ranking model needs to be learnt. In the most general case the cardinality of $\mathbb{A}$ is not fixed. For this reason, the ranking model, say f_r, can be thought of as a process that is divided in the following steps [33]

- Obtaining a relevance score of each object in set $\mathbb{A}$.
- Sorting the objects based on their relevance score.

We can learn the ranking model f_r by optimizing a criteria on the output of the model i.e the sorted list of object. This criteria in the jargon of machine learning is called a loss function. Hence, it can be referred to as a *Ranking Loss*.

Since the step of sorting is non differentiable, it cannot generally be learnt during the optimization of our Ranking Loss. Hence the ranking model, f_r, boils down to a relevance scoring function. After learning f_r, one can use it to rank newly given sets of objects by first finding their relevant scores and then sorting them accordingly.

Various types of ranking losses can be used in our optimization to learn the scoring function. These can be broadly classified into the following types [7]:

- Point-wise ranking losses
- Pair-wise ranking losses
- List-wise ranking losses

In point-wise ranking loss, the loss function views the problem of ranking as that of assigning a label to each of the input data points. Hence, for learning, each instance is a single object x_i within the set $\mathbb{A}$. For example, in the McRank paper [24], the authors reformulate the ranking problem as a multi-level classification problem where each data point is classified independently. They then calculate the score as the expected rank of the object based on its soft classification. Therefore, the complete scoring function comprises of a multi-level classifier and an external expectation calculation.

In pair-wise ranking loss, the loss function's input is a pair of objects. This loss function learns to model pair-wise preferences. The function tries to separate the input data points as much as possible in the output space by minimising the pair-wise classification error [8].

In list-wise ranking losses, the loss is defined on the complete set of objects. The 2 most important list-wise loss functions are

- ListNet [5].
- ListMLE [42].

The point wise and pair wise ranking models do not view the ranking problem as a problem to rank a set of objects. This is quite intuitive and is a fundamental advantage as compared with other methods. It has been shown in [5] that list wise approaches are superior in performance to point wise and pair wise losses. We hence use the list wise approach to ranking. We discuss and analyse the loss functions ListNet and ListMLE in detail in the next sections

3.2 Loss functions: Definition

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

Here let each data point a be a sample/element from the set $\mathbb{A}$. Each y represents the ground truth preference score of objects belonging to a set $\mathbb{Y}$. These preference scores of objects are relative to the objects within the input

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 format used to train the scoring function using list wise ranking loss

set. Let s be the scoring function to be learnt. Hence, the declaration of s is given by

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

As we can see from the table, one instance in our data consists of a set of objects as input and a set of corresponding ground truths to train from. To learn the function s we optimize our list wise loss function. This loss function takes as input the whole set of objects and their ground truth as one instance. If we take any set $\mathbb{P}$ such that $\mathbb{P} \subseteq \mathbb{A}$, the declaration of the list wise loss L is hence given by

$$L : s(\mathbb{P}) \times \mathbb{Y}^{|\mathbb{P}|} \mapsto \mathbb{R} \tag{3.1}$$

Where the scoring function s applied to the $\mathbb{P}$ gives us the set of corresponding scores of all objects in $\mathbb{P}$. We will consider the ground truth values to be $\mathbb{R}$ for our analysis as this is type of value we have in our data sets.

In the next 2 sections we analyse ListNet and ListMLE, the prominent listwise loss functions in the literature.

3.3 Loss function: ListNet

In this section we try to intuitively explain the ListNet idea proposed in [5]. Our objective is to learn the scoring function s such that it returns scores that are similar in relevance/order when compared to the ground truth scores. That is to say

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

This would make the ranking of the objects equal to the ranking obtained by using the ground truth values. Note that we do not need to get the exact ground truth scores. This increases the number of acceptable functions that

can be learnt by increasing the target function space. This makes it easier to learn the scoring function.

Ranking is obtained by sorting the objects based on their respective scores. Note 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 2 lists:

- List of scores given by scoring function s.
- List of scores given to us by ground truth.

The loss function must find some sort of a distance between the 2 given lists. It then can reduce the distance by changing the parameters of the scoring function.

In ListNet, a probabilistic approach is taken so as to account for any uncertainties in the ground truth values. Consider selecting an object from the input set with a probability

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

This make intuitive sense because the probability of selecting an object should be higher if it more relenvant and vise-versa. Note that the score of any object by the scoring function can negative as well. Therefore the score is passed through a strictly positive and increasing function ϕ. This changes the probability to

$$P = \frac{\phi(s(a))}{\Sigma_i \phi(s(a_i))} \quad \forall i \in \{1, 2, 3, ..., |\mathbb{P}|\} \tag{3.2}$$

Equation 3.2 is also referred to as top 1 probability of an object in [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 way to find the distance between 2 lists in ListNet is

- Find the top 1 probabilities of each object using the scores given by the scoring function.
- Using the ground truth values, find similar top 1 probabilities.
- The cross entropy between the 2 entities gives us the "distance" between the 2 lists.

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

$$L(\mathbf{y}, s(\mathbf{a})) = -\Sigma_i P_{s(a_i)} \log P_{y_i} \tag{3.3}$$

Where $\mathbf{y}$ and $s(\mathbf{a})$ represent the ground truth values and the scores given by the scoring function.

3.4 Loss function: ListMLE

ListMLE loss stands for, "List Maximum Likelihood Estimation" loss. It is another type of list loss function that is similar to listNet. As in ListNet, the probability of selecting an object from the list is taken the same as given in equation 3.2. However, the final loss used in MLE is not cross entropy. Rather 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 one document after another without replacement. In our case the permutation probability of selecting 1 permutation using the selection probabilities given by equation 3.2 is [5]

$$P_\pi = \prod_{j=1}^{k} \frac{\phi(s(\pi_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi_k))} \tag{3.4}$$

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

Applying log to the above equation gives us

$$\log P_\pi = \sum_{j=1}^{k} \log \frac{\phi(s(\pi_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi_k))} \tag{3.5}$$

However, the question remains which permutation to use? The best permutation for the given set of objects would be according to the true 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

$$\log P_{\pi^*} = \sum_{j=1}^{k} \log \frac{\phi(s(\pi^*_j))}{\sum\limits_{t=j}^{k} \phi(s(\pi^*_k))} \tag{3.6}$$

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

$$L_{mle} = -\log P_{\pi^*} \tag{3.7}$$

Expanding the right hand side of the equation gives us the final loss function that has to be minimised by any algorithm that uses ListMLE

$$L_{mle} = -\sum_{j=1}^{k} \log \frac{\phi(s(\pi_j^*))}{\sum_{t=j}^{k} \phi(s(\pi_k^*))} \tag{3.8}$$

Notice that in the calculation of the loss, the true score values of the objects are unused. Which means that the actual scores of the objects do not matter. The only constraint is that the scores must have the values that give the same permutation. Even uneven scaling of the actual scores does not affect the output as long as the constraint is maintained.

In the case of ListNet, however, the true scores do matter. The probability of selecting objects according to their true scores is used. Hence the result is invariant only to linear scaling.

This advantage makes listMLE loss function superior to the ListNet as it makes the target space of functions bigger and hence the convergence can be quicker. Because of this advantage we use ListMLE as a list loss function in our thesis.

3.5 Position Enhanced Ranking

In many problem domains that use the ranking concept, it may not be important that each object be placed at 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 to order the least important results correctly.

This is also the case in the problem of ranking HP configurations when the ranking model is used as a surrogate in an SMBO process. In this process, the ranking surrogate is only needed to obtain the most important HP configuration at each step in the optimization cycle.

Lan et al. discuss this problem in detail in their paper, "Position-Aware ListMLE: A Sequential Learning Process for Ranking" [22]. 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 decrease

function c [6]. This is possible because the listMLE loss function is in the form of a summation. Hence the weighted ListMLE function is given by:

$$L_{mle} = -\sum_{j=1}^{k} c(j) \log \frac{\phi(s(\pi_j^*))}{\sum_{t=j}^{k} \phi(s(\pi_k^*))} \tag{3.9}$$

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

3.6 Uncertainty modelling using Deep Ensembles

Deep Neural Networks (DNNs) are machine learning models with very high representational capacity [14]. Due to this property, one can use them as surrogates for HPO objective functions. But the issue is that DNNs do not quantify uncertainty trivially. In fact the results can be seen as overconfident. If used in any HPO optimization algorithm as surrogate, such overconfident wrong predictions cause a lot of computational overhead by predicting inefficient HP configurations.

We are discussing in detail this concept because we use deep neural networks as a scoring function in our proposed ranking loss surrogate model. Uncertainty estimation qualities of a surrogate have high importances especially if they are used in techniques like SBMO. As the proposed model is studied as a surrogate in the SMBO technique, we need to study how to model uncertainty efficiently using the underlying DNN architecture.

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

- Bayesian Neural networks [12].
- Ensemble Approach using monte carlo drop out [36].
- Ensemble approach using multiple neural networks.

In Bayesian neural network(BNN), a prior over weights and biases is specified during the initialization of the BNN. Given the data, a posterior predictive

distribution is calculated for all the parameters of the network (Weights and Biases). One issue with this approach is that BNNs are very complex and difficult to train.

Monte Carlo drop out is a regularization technique used during the training of neural networks. With a certain probability, connections between neurons are dropped. Using this technique one obtains possibly 2^N neural networks where N is the number of connections in the artificial neural network. We can get an ensemble of high capacity models for free. It is normally only used during training to obtain regularization.

However, if one uses Monte Carlo dropout during the evaluation, we can get multiple results from the same input using this approach. Given input x and output $y = \mathrm{NN}(x)$. If we have m neural networks obtained using Monte Carlo dropout, we get $\{y_1, y_2 ... y_m\}$ outputs, we can obtain the mean and variance of

$$y_{\text{mean}} = \frac{\Sigma y}{m} \qquad y_{\text{variance}} = \frac{\Sigma (y - y_{mean})^2}{m - 1} \tag{3.10}$$

Please note that the $m-1$ in the denominator is due to Bessel's Correction [39] to reduce the bias in estimation.

Lakshminarayana et al. [21] propose another method to predict uncertainty using deep neural networks. They propose that the uncertainty prediction can be done 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 2 outputs instead of one. One for the mean of the prediction, say μ, and the other for the variance, say σ^2 of the prediction. One important point to note is that the variance cannot really be negative. This is made sure by the authors to pass the output of the neural network through a strictly positive "softplus" function.

The authors propose to optimize the following loss function

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

Where both the outputs are some functions of the DNN parameters (θ) and the input ($\mathbf{x}$) i.e $\sigma^2 = f(\theta, \mathbf{x})$ and $\mu = g(\theta, \mathbf{x})$. Here, the back propagation finds and updates the parameters θ using the partial derivative:

$$\frac{\partial L}{\partial \theta}$$

We use this loss function to build deep ensemble surrogates for the HPO. The prediction of uncertainty using ranking losses is however done using the

simple ensemble approach given in equation 3.10. This is because the integration of the loss function which learns both the mean and variance with the ranking loss functions is non trivial.

One simple approach is to simply use a combination of losses like

$$L_{\text{total}} = L_{mse} + L_{de}$$

However such loss functions would need a thorough theoretical analysis which is out of the scope of this thesis. Hence we do not use this approach on our proposed model.

As there are are multiple neural networks, each predicting its own 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, the mean and variance are given by

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

3.7 Baselines

In this thesis, 2 HPO techniques were implemented before studying the proposed model - Deep Ensembles and Few Shot Bayesian Optimization (FSBO). Deep Ensembles were used as surrogates in the SMBO optimization. They were studied with with 2 main objectives in mind. First to study how uncertainty is estimated using Deep Neural Networks using the approach proposed by [21]. This was a pre-requisite to implement the uncertainty in our proposed model as we use deep neural networks as a scorer in our model. Second to understand how a non transfer technique like GP would work for our given problem. Note, it is also possible to make the Deep Ensemble surrogate a transfer technique by meta training it before using it in the optimization cycle.

The second technique implemented was FSBO. We choose this because as this gave the state of the art results on the HPO-B benchmark that we use. (The benchmark is discussed in the later chapters). Studying FSBO also gave us an idea on how to implement the transfer mechanism in our proposed model. In addition to this we used Random search and GP as standard baselines for result comparison.

Both these FSBO and DE have built in capability for uncertainty estimation. Hence we have to use an acquisition function during the optimization cycle of SMBO. Expected improvement was used in all models that deal with

uncertainty. This is to maintain consistency in results across all the models. Further more it also has advantages on other acquisition function [19].

The following 2 sections discuss the implementation details of the baselines methods used in our thesis.

3.7.1 Deep Ensemble

As previously mentioned Deep Ensembles were implemented according [21] as a non transfer surrogate. Hence, there was no meta-training done for the Deep Ensembles. Consequently the usage of DE as a surrogate was quite similar to that of Gaussian processes. SMBO with deep ensemble was implemented as shown in algorithm 1 [30].

Algorithm 1 SMBO with Deep Ensemble surrogate

$X_{known}, Y_{known} \leftarrow$ Initial HP configurations.
$X_{pending} \leftarrow$ HP configurations to evaluate.
$k \leftarrow$ Number of evaluation cycles
for $i < k$ **do**
    DE $\leftarrow$ Randomly initialize 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 FSOB implementation and the proposed Ranking Loss surrogate model. The only difference is that the surrogate and its training step differ with different methods.

A few points are worth noting here. First, we see that the algorithm evaluates a set of discrete HP configurations in the HP search space. This is the same approach we take when we apply our model because the ranking concept we use requires a set of defined 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 upto a required granularity based on our computational resources.

Secondly, we see that at each evaluation cycle, a new set of neural networks are trained. Old trained neural networks are discarded. This rationale behind this is discussed in chapter 4. We also use this sort of initialization in other models. Finally, as the neural networks are independent of each other, they can be trained in parallel. This makes the usage of this model scalable.

One question that remains to be answered is what sort of architecture is used by our neural networks. For this 2 architectures were analysed - undivided neural network as shown in Figure 3.1 and a neural network divided at its tail as shown in Figure 3.2

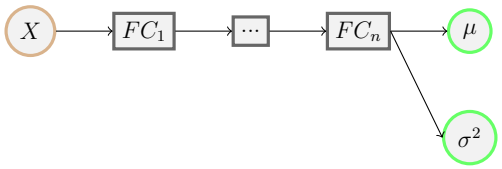

Figure 3.1: Example of an undivided Neural Network architecture

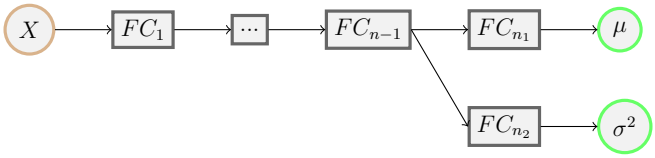

Figure 3.2: Example of a divided Neural Network architecture

In both these figures, FC stands for fully connected layers. As the undivided architecture is closer to the neural network used in the proposed ranking loss surrogate model, we used it for the sake of consistent comparison. 3 Fully connected layers of 32 neurons each were used for each neural network. All neural networks used the same architecture. The training was done using the the Adam optimizer with full batch gradients. This is because the number of data points in the evaluation cycle are very few. Each neural network was trained for a 1000 epochs with a learning rate of 0.02. We do not use adversarial examples as proposed in the deep ensemble paper because it was giving bad results.

3.7.2 FSBO

Few Shot Bayesian Optimization (FSBO) is a transfer learning HPO method that utilizes a meta learnt surrogate for knowledge transfer mechanism. It reformulates the problem into few shot learning task. In the context of our problem, this means meta learning thoroughly from the existing meta data and then adapting to the new task at evaluation cycle by fine tuning a few training epochs. Therefore the following 2 steps are required to be done in a chronological order

- Meta training - For knowledge transfer.
- Fine tuning - For few shot learning.

For the purpose of knowledge transfer, the authors make use of a deep kernel surrogate proposed in by wilson et al [40]. Here, a neural network is used to transform points in the HP search space to a latent space. Kernels are then applied to this latent space in a Guassinan process to obtain a probabilistic evaluation. The deep kernel kernel can be represented as [41]

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

Where $\mathbf{x}$ and $\mathbf{x'}$ are HP inputs in the original search space, θ and $\mathbf{w}$ are parameters of kernel k and the neural network ϕ. We used the implementation of deep kernels provided by Patacchiola et al. [29] in this thesis.

During the meta training step, we train our surrogate by learning the parameters θ and $\mathbf{w}$. A New FSBO model and consequently new surrogate parameters have to be learnt for every new search space. This is the case even if the input dimensions are of the search space domain are the same. This is because every HP search space represents different machine learning model and hence has a different HP response surface. During training we first used an RBF kernel and then used a matern $\frac{5}{2}$ kernel. A fully connected neural network was used to obtain the latent space representation.

If an assumption is made that the knowledge from the meta data is enough to predict the best HP configuration across all future dataset, the fine tuning step may be skipped. However, this is rarely the case as there are always variations in new dataset (Even though the model being optimised is the same). Therefore, the hyper parameter response surface would also be different.

The optimization algorithm in the case of FSBO is similar to Algorithm 1 with a minor change. The acquisition function used in this model is also expected improvement. The concept of restarting the fine tuning each time is also proposed in the FSBO paper [41]. The difference here is that at each

evaluation step, the stored model is loaded anew and fine tuned before using it for predicting the best model. The restart happens from the meta trained FSBO model and not from a randomly initialized model as in the case of Deep Ensembles.

In our implementation, Adam optimizer was used in both the training and fine tuning steps. Note, however, that the learning rate of the kernel parameters and the neural network parameters were identical. The learning rate used for meta-training was 0.0001 and that used for fine tuning was 0.03 different. We utilized cosine annealing during the fine tuning cycle. The rationale for using this is discussed in detailed in chapter 4.

We used early stopping mechanism for meta training because we wanted to avoid huge computation costs and over fitting of the model during training. We took advantage of the split of meta-validation data to do the early stopping. We saved the best model in our implementation and if the validation loss went greater than lowest validation loss. The training was stopped if the training error was consistently higher than the validation error for a set number of epochs. In our case it was 10 epochs but this is easily configurable.

Chapter 4

Method

The choice of surrogate to use in model based optimization is very crucial. Does the surrogate have enough representational capacity? Does it have the capability of representing uncertainty? How is the surrogate learnt? These are some of the questions that need to be answered before selecting a surrogate model. The selection of a surrogate model has a direct impact on the performance of the model based optimization algorithm.

In the quest to improve HPO surrogates, we propose and analyse a new type of surrogate model that is based on the concept of ranking. There are 2 components of our proposed idea

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

This chapter discusses in detail both these components. In this chapter, we first assume that a good surrogate model of sufficient representational capacity already exists. We use a simple Deep Neural network for this purpose. We then analyse and implement the proposed learning algorithm that uses the concept of ranking. We then do a simple case study of inverse mapping using the learnt model. Finally we build a ranking model surrogate to improve the performance of any model based HPO algorithm. We use SMBO as a reference for our study.

4.1 Basic scoring model : Deep Neural Network

In order to discuss the implementation details of ranking loss functions, we need to first have a reference scoring model. Since we also want representa-

tional capacity to be high a fully connected neural network is used as a scorer to begin with. This is depicted in Figure 4.1.

Figure 4.1: Basic scoring model

As we can see this model is almost similar to the model 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 to this there is a very crucial "Range Controller" component added to our scoring model.

Lets say $\{\mathbf{X}, \mathbf{y}\}$ be the training data used for training a generic machine learning model m. Most of the loss functions utilize the values present in $\mathbf{y}$ as a reference to train the model. After the training completes, the range of the model m is not vastly different from the the range of the observed ouputs $\mathbf{y}$. This is not guaranteed in the loss function L_{mle}. As mentioned in section 3.4 the only thing that matters is that their corresponding ground truth relevant scores always should yield the same order. Hence the scoring function s learnt using our ranking loss function can have an arbitrary range.

In our models, we use the strictly positive increasing function exp as proposed in [42]. Hence our L_{mle} becomes

$$L_{mle} = -\sum_{j=1}^{k} \log \frac{\exp(s(\pi_j^*))}{\sum_{t=j}^{k} \exp(s(\pi_k^*))} \tag{4.1}$$

Even though theoretically the range of our scoring model s does not need to be restricted, we do have practical limitations due implementation constraints of floating point numbers used in computer software. Due to the exponentiation in our loss function, if the range of s is too high, the values in equation 4.8 may overflow. If the range is too small, the values may underflow. In any case, we are bound to get nan (Not A Number) exceptions in our implementations.

For this reason we control the range of our scorer 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 of our scorer through the following function

$$k * \tanh(\alpha * s(\mathbf{X})) \tag{4.2}$$

Where α is the smoothness factor which is inversely proportional to the smoothness of the tanh graph [20]. Figure 4.2 shows how the to vary the smoothness and range of the output using α and k respectively.

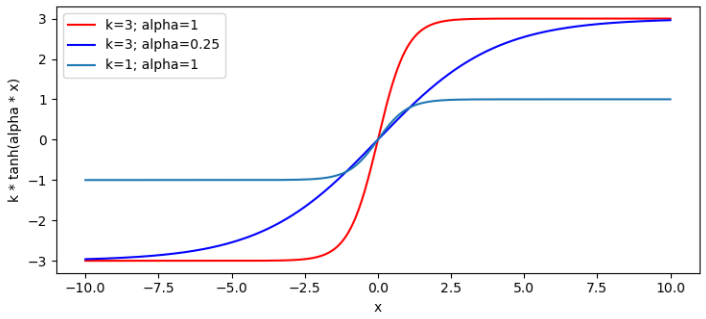

Figure 4.2: Effect of varying k and α in equation 4.2

We used $k = 2$ and $\alpha = 0.01$ for our scorer.

4.1.1 ListMLE Implementation

To train the scorer the loss function ListMLE was implemented in python using the PyTorch [28] deep learning library. Algorithm 2 shows the steps used to obtain the loss scalar before back propagating it using the autograd functionality of PyTorch. Please note that the actual implementation is a little more sophisticated due to the use of multi dimensional tensors.

For numerical safety and accuracy the values of the predicted score reduced by a constant factor. Here the constant factor is the biggest value of the list itself. This is possible because we are using exp as our strictly increasing function:

$$\frac{e^{a_1}}{\sum_i e^{a_i}} = \frac{e^{a_1+k}}{\sum_i e^{a_i+k}}$$

It is very evident from the implementation of Algorithm 2 that it calculates the permutation probability of the objects in the list as described in Section 3.4. However, the algorithm is not efficient. This is because it modifies and removes elements in the lists. if lists are implemented as contigous elements (like arrays) the time complexity of running would be $O(n^2)$ where n is the list size.

One way to get around this problem is to sort the lists before calculating the permutation probability. This implementation is done in the paper [32]. This makes the time complexity $O(n \log n)$ We use this implementation due

Algorithm 2 Loss ListMLE Algorithm

Input : $l_{predicted} \in \mathbb{R}^k$ ▷ Relevance scores predicted by the scorer
Input : $l_{actual} \in \mathbb{R}^k$ ▷ Actual relevance scores
Output : Loss $\in \mathbb{R}$

procedure ListMLE($l_{predicted}$, l_{actual})
  $l_{predicted} \leftarrow l_{predicted} - \max(l_{predicted})$
  $l_{predicted} \leftarrow \exp(l_{predicted})$
  $sum \leftarrow 0$
  for $i < k$ **do** ▷ k is list size here
    $sum \leftarrow sum$ + Top1LogProb($l_{predicted}$, l_{actual})
    $l_{predicted}, l_{actual} \leftarrow$ RemoveTop1($l_{predicted}$, l_{actual})
    $i \leftarrow i + 1$
  end for
  Return $-1 * sum$
end procedure
procedure Top1LogProb($l_{predicted}$, l_{actual})
  $j \leftarrow \text{argmax}(l_{actual})$
  $prob \leftarrow \frac{l_{predicted}[j]}{\sum l_{predicted}}$
  Return $\log(prob)$
end procedure
procedure RemoveTop1($l_{predicted}$, l_{actual})
  $j \leftarrow \text{argmax}(l_{actual})$
  $l_{predicted} \leftarrow$ RemoveElementAtIndex($l_{predicted}$, j)
  $l_{actual} \leftarrow$ RemoveElementAtIndex(l_{actual}, j)
  Return $l_{predicted}$, $l_{predicted}$
end procedure

to its efficiency. Algorithm 3 depicts this. More precisely the ListMLE is first expanded to the following equation.

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

Since the scores are sorted in the correct order we can re-write the equation as

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

Now $\sum_{t=j}^{k} \exp(s^*(k))$ is nothing but the reverse cumulative sum where the first element is the sum of all elements and the next element is the sum of all elements starting from position 1 and so on. Let this be represented by Q. $\log \exp(s^*(j))$ can be directly written as $s^*(j)$ if natural logarithm is taken. Hence, the equation that is implemented in Algorithm 3 is [32].

$$L_{mle} = \sum_{j=1}^{k} (\log Q(j) - s^*(j)) \tag{4.5}$$

Algorithm 3 Loss ListMLE Algorithm (sorted)

Input : $l_{predicted} \in \mathbb{R}^k$ ▷ Relevance scores predicted by the scorer
Input : $l_{actual} \in \mathbb{R}^k$ ▷ Actual relevance scores
Output : Loss $\in \mathbb{R}$

procedure ListMLESorted($l_{predicted}$, l_{actual})
  l_{actual}, IndexOrder $\leftarrow$ Sort(l_{actual})
  $l_{predicted} \leftarrow$ SortWithIndexOrder($l_{predicted}$, IndexOrder)
  $l_{predicted} \leftarrow l_{predicted} - \max(l_{predicted})$ ▷ Numerical Stability
  $prob \leftarrow 0$
  for $i < k$ **do**
    $prob \leftarrow prob + \log Q[i] - l_{predicted}[j]$
    $i \leftarrow i + 1$
  end for
  Return $prob$
end procedure

4.1.2 Weighted Loss

As discussed in Section 3.5, the ranking problem we are have in HPO 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 is evident from Algorithm 1 where at every optimization cycle we are only concerned about selecting the best available configuration for true evaluation. Keeping this constraint in mind, we need to have a weighting strategy such that

$$c(j) \propto \frac{1}{j}$$

where j is the ranking of the object and $c(i)$ specifies its weight.

In this section we discuss about some strategies to do this weighting for our problem. The concept of biased ranking loss is discussed in the paper, "Top-Rank Enhanced List wise Optimization for Statistical Machine Translation" by Chen et. al [6]. Here Chen et. al proposes a position based weighting of the ranking function such that:

$$w_j = \frac{k - j + 1}{\Sigma_{t=1}^{k} t} \tag{4.6}$$

where w_j represents weight of the object at position j in the ordered list. This strategy decreases the weights linearly across the ranks. The issue with this weighting is that it takes the weight of an object in the middle of the list to be 0.5 times the weight of the object at the top of the list. This however is not what we want for our case. We need a sharper decrease in weights across the ranks.

We could use 2 simple strategies of weighting for our problem

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

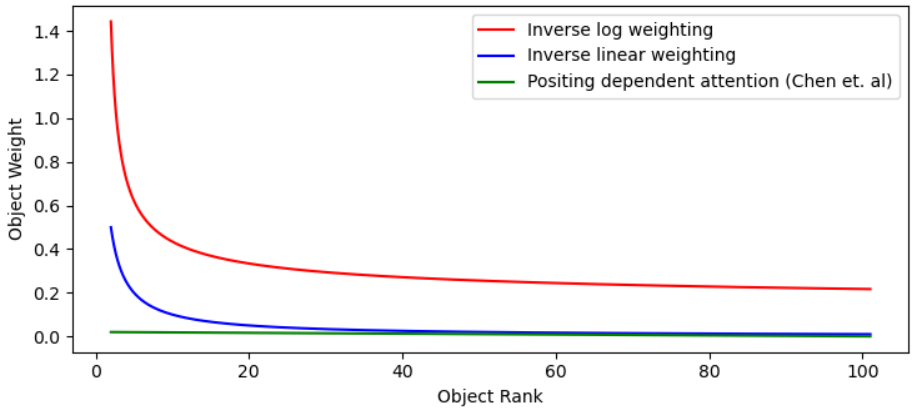

Figure 4.3: Different Weighting Functions

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 ranks in the list. This 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.

The inverse log weighting was found to be most suitable because it neither completely ignores the objects at the end of the list nor does it given them too high an importance. In fact, the weights of the rest of 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 our HPO during the optimization cycle. For example, at every step the top n configurations can be selected, and all of them can be tried for optimization. This overcomes any uncertainties that may occur while ranking the top part of the list. The parallelism is less effective when using the weighting proposed by Chen et al.

4.2 Case study with inverse mapping

In this case study, we study how the learnt ranking function behaves when we use different parameters to train. For this a toy example of sorting in the descending order is considered.

The main problem that we try to study here is - Is it possible to train a ranking function 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 of the scorer 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

$$\underset{\theta}{\operatorname{argmin}} \quad L_{mle}(s(l \mid \theta), -k * l) \tag{4.7}$$

where $k \in \mathbb{R}$. Note that second parameter of list wise loss function is scale invariant hence scaling the list has no effect on the loss output (Section 3.4).

The validation data taken from 3 different ranges

- Same range as the training data $[1, 100]$
- Completely different range as seen by the scorer during training i.e $[-100, -1]$
- Hybrid range i.e $[-50, 50]$

To evaluate our scorer, we first sample the validation data from the above ranges, We then check the percentage of the lists that are correctly sorted during our testing time. We used the same values of k and α as used in the basic scoring model. During training a batch size of 100 lists was used. We try sort 1000 lists according to the learnt scorer's results. The accuracy gives the fraction of the lists that were sorted in 1000 lists. We report the average of 5 runs in the Table 4.1.

Training Epochs	List size	Learning Rate	In-range Acc.	Out-range Acc.	Hybrid Acc.
1000	3	0.0001	0.99	0.27	0.40
100	3	0.0001	0.77	0.29	0.11
100	30	0.0001	0.80	0.79	0.46
100	100	0.0001	0.99	0.0	0.39
1000	100	0.0001	1.0	0.71	0.48

Table 4.1: Sorting Accuracies at test time

We find from the tabulated results that it is important to completely learn the function by running a higher number of epochs. To comprehensively learn the scoring function, it is crucial to have a larger list size. This will make our scorer model work well when the input is from the same distribution seen during training. Moreover, it also gives reasonable when the input comes from the distribution edge (i.e from a location close to the distribution). These are good properties to have in any machine learning model.

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

4.3 Method training and optimization

The proposed method in this thesis, "Ranking loss surrogate" is a transfer HPO model. For this reason its working principle is similar to that of FSBO model. Like FSBO we use surrogate learning as a knowledge transfer mechanism. Hence the 2 main parts of using our optimization process are

- Meta learning the ranking loss surrogate
- Using the trained surrogate in the evaluation cycle (with or without fine tuning).

4.3.1 Meta training the surrogate

The space of hyper parameter configurations in which we are trying to get an optimum during any HPO is called HP search space. This search space is different for different machine learning models. Hence we need to train different ranking loss surrogate models for different search spaces.

The same machine learning model, however, can be used to fit different data sets or tasks. This leads to different HP response surfaces in the same search space. Hence, we obtain different meta data when the same model is optimized on different datasets (or tasks). The goal of training our model is to use different meta-data sets (or tasks) from a single search space to train our model. The idea is to learn all the common characteristics of the tasks and transfer this knowledge to the new task.

We use stochastic gradient descent to train our model. For this we need to sample a batch of data from the given meta data. There are 2 ways to do this:

- Double sampling: First sample the task then sample the meta data within the task.
- Sample meta data from all tasks.

We use the first method of sampling our data. This is because due to higher level of sampling, we have obtain good regularization, faster training and a possibility to use a lower learning rate. We use Adam optimizer with a learning rate of 0.001 for our model. We do our training for 5000 epochs and in each epoch we take 100 steps in which each step does 1 double sampling. While sampling data points from within a meta data set, we sample without replacement. We use the number of lists (batch size) as 100 and the size of each list also 100. Algorithm 4 illustrates us a brief skeleton of our meta training procedure. After the training, every model is saved on to a persistent location (e.g. hard disk) so that it can be load when necessary.

4.3.2 Fine tuning

Fine tuning is the second major part of the optimization process. This may be considered optional for transfer HPO models like ours. However, we found that it improved the HPO evaluation cycle performance.

We take the SMBO optimization of Deep Ensembles in Algorithm 1 as an example to understand the fine tuning process. In this algorithm, at every evaluation cycle the deep ensembles are retrained with the seen evaluations of

Algorithm 4 Ranking Loss surrogate meta training

Input : $epochs \in \mathbb{I}$
Input : X_{train}, y_{train} ▷ Meta data used to train
Input : s_θ ▷ Model to train

procedure MetaTrain(s_θ, X_{train}, y_{train}, $epochs$)
  for $i < epochs$ **do**
    for $j < 100$ **do**
      $B_X, B_y \leftarrow$ DoubleSample(X_{train}) ▷ Get the training batch
      $y_{pred} \leftarrow s_\theta(B_X)$
      loss $\leftarrow L_{mle}(y_{pred}, B_y)$
      $g \leftarrow \frac{\partial \text{loss}}{\partial \theta}$ ▷ g stands for gradient
      Use g to update θ using the Adam optimizer
    end for
  end for
end procedure

the true HPO objective function. We need to modify Algorithm 1 by replacing Deep Ensemble training with the fine tuning of our model. The fine tuning process is exactly similar to the MetaTrain procedure in Algorithm 4. The only difference is that we use seen evaluations instead of meta training data for training. To avoid duplication, we do not re-write the complete algorithm again. We do fine tuning for 300 epochs. The number of epochs can be varied based on the computation resources available.

Like FSBO, at every evaluation cycle, the model is reloaded from the saved state before being fine tuned. In the next section we discuss why this restart is required. Subsequently we talk about the use of cosine annealing used during fine tuning.

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 step (acquisition step). This is counter intuitive because an already trained model should converge to a local optima quickly. One of the reasons for this performance anomaly is that the model gets biased towards the points that are observed in the starting steps of the optimisation cycle. Lets say we have 2 models

- $\text{m}_{\text{restart}}$ which always restarts training at every acquisition step.

- m_{reuse} which trains the model trained in the previous optimization steps.

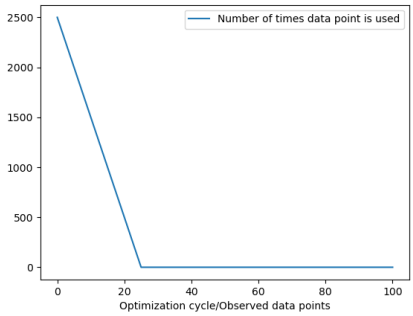

Figure 4.4: Bias at 25th optimization cycle

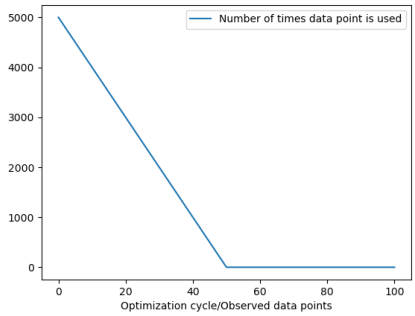

Figure 4.5: Bias at 50th optimization cycle

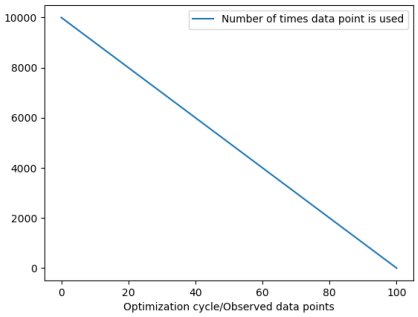

Figure 4.6: Bias at 100th optimization cycle

Let the models be fine tuned for 100 epochs at each step. Let there be just 1 seen HP configuration to start with. 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 cycle step by the m_{reuse} model.

Generally all the data is used during fine tuning 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 heavy bias that is present in the figures is actually not intended. This is because all observations should be treated equally in any training/fine-tuning step. When we use $m_{restart}$ every known HP configuration is used only 100 times at every evaluation step. Hence it is better to restart the model before fine tuning.

The second reason for restarting is that the model may get stuck at a stubborn local minima at any fine tuning step n where $1 <= n <= 100$ in

our case. Coming out of this local minima may require the response surface to change very drastically. The response will change like this only when the training data distribution significantly changes. This is not possible in the sequential process of SMBO because at every step only 1 new HP configuration is added to the known HP configurations.

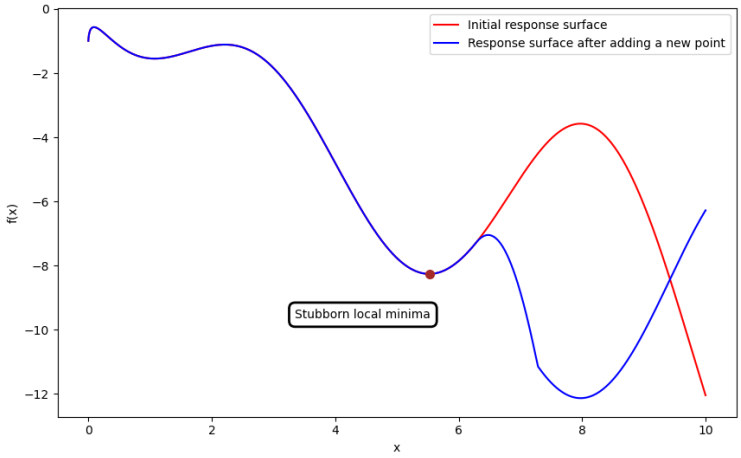

Figure 4.7: Figure showing changes in response surfaces and a stubborn local minima

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. This is represented by the blue response surface. If our model is already trained for the red curve, it becomes extremely difficult for it to come out of local minima because of the minimal changes in the response surface. We call this minima a stubborn local minima. This is depicted by the brown dot in the figure.

If however, we reload and retrain our model from the beginning there is more chance of it reaching the good local optima in the blue curve if it descends from the other direction. The number of ways to reach a good local minima increases exponentially with the increase in the HP search space dimensions. Hence, restarting the training (from the saved model in our case) should improve our fine tuned model.

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

Using cosine annealing

We plotted the fine tuning loss curves of our model We found that these loss curves were very jittery. We noticed the same behaviour when fine tuning FSBO model. Figure 4.8 shows the fine tuning loss for one of the search spaces.

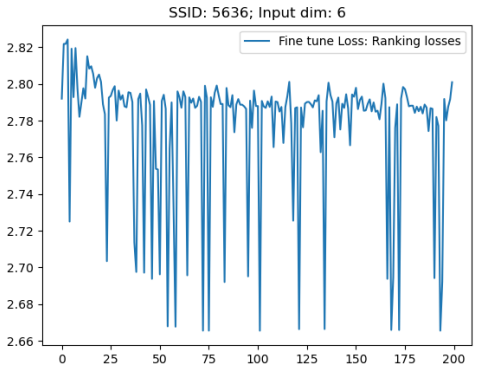

Figure 4.8: Figure showing jittery loss function curve

We hypothesise that these jittery losses are because the learning rate is not suitable with the response curve's curvature at the targeted local minima. One solution to this is to use a smaller learning rate. However, the fine tuning using smaller learning rate will take a long time. The solution we proposed and utilized was using cosine annealing.

There are couple of advantages of using cosine annealing. First the initial learning rate is kept high in order to give the optimizer time to get to an area close to the local minima if it has not done so. Thereafter, the learning rate declines quickly to the target small learning rate. This helps the model to get deep into the local minima and hence the possibility of the optimization to jump out of the local minima is very less.

Figure 4.9 shows a fine tuning loss curve obtained using cosine annealing. Even though this strategy may not be a perfect one, it reduces the possible problems that could prop us when using a constant learning rate. During training, however, we use a constant learning rate as can use the validation loss as a reference for over fitting or under fitting.

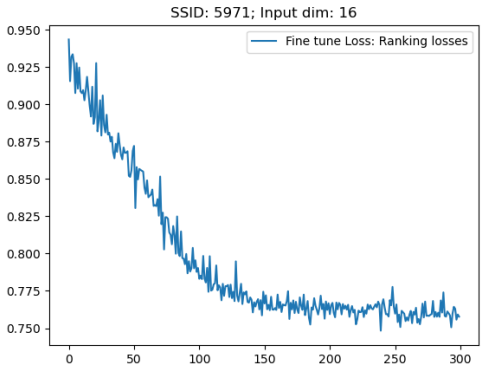

Figure 4.9: Figure showing a loss curve obtained by using cosine annealing

4.4 Ranking Surrogate Model

After explaining the learning mechanism, we now turn to 2 important components that are used in our method. The first is Deep Set to make our model context aware. The second is the use of uncertainty for improving the output of the model.

4.4.1 Using Deep Sets to build context aware models

We have already discussed the fundamental ideas of using deep sets in 2.5.1. In this section we discuss about how to add deep sets in the ranking loss surrogate model architecture and then how to train the same.

The basic idea is to precondition our scoring function s_θ on known evaluations of the target HP space. These known evaluations act as a support for the scoring function. Hence they are subscripted as "s" in the equation. We then query the scoring function to get the relevant scores of new $\mathbf{X}$ which are subscripted as "q" in the equation. The scoring function hence becomes:

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

The architecture of the scoring function s_θ with deep sets is given in Figure 4.10. For the complete architecture of the Deep Set node please refer 2.2. We keep the node abstract for simplicity.

First the data points in the query set $\mathbf{X}_q$, $\mathbf{y}_q$ are passed through the deep set to get the Latent Output. Each X, y are concatenated to get a single vector for input into the deep set. We obtain a latent output from the deep set. This latent output is then concatenated with the query data points. The

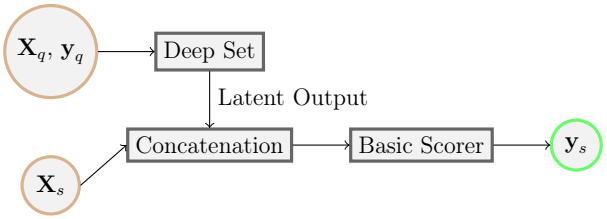

Figure 4.10: Skeleton of the proposed model with Deep Sets.

concatenated result is passed through a Deep Neural Network to finally get the desired relevance scores. 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

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

where ":" represents concatenation of 2 vectors.

Meta training

Using the above given scoring function during the evaluation cycle is very straightforward. We use the known data points as a support set and the target data points to evaluate as the query set. During meta training, however, we have to divide the training data into support set and query set.

In our data division we first select 20 (X, y) data points as a support set randomly without replacement. Thereafter, we select the query points from the remaining choices (again without replacement). The number of query points depends on the batch size (100 by default). During the meta training we do not sample the support and query points from all the given meta datasets. At each step within an epoch we select one meta data to sample these points as discussed in Section 4.3.1. The training of all the components in the model are carried out together (i.e. in the same back propagation step when using Pytorch). For the purpose of analysing the training accurately, we however sample data from all the training and validation task to report the training and validation loss respectively.

4.4.2 Uncertainty implementation

We know from Section 2.1 that the evaluation of the target objective function in HPO is noisy. Hence it is important for any surrogate that models this objective to have the capability of estimating uncertainty. The model proposed

till now has no such capability. Hence, in this section we propose the extension of the model to incorporate uncertainty.

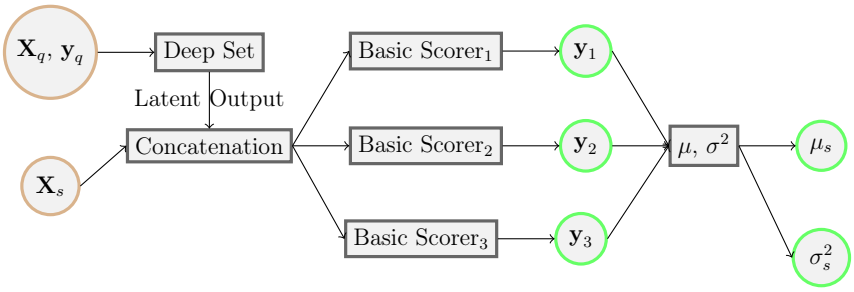

Figure 4.11: Full proposed scoring model with uncertainty

Figure 4.11 shows the architecture of the model with uncertainty. The only addition to the model is the usage of multiple deep neural networks. As discussed in Section 3.6, we use multiple neural networks to calculate the mean and the variance instead of using a single neural network to do so. Given the support set and a single query, the scoring function modelled in Figure 4.11 gives a mean and a variance assuming that the distribution of the possible relevance scores is a Gaussian.

We train all the neural networks together using the given meta data. For this, the loss function requires a definite value of the predicted output y. We use the mean μ_s for this purpose. Due to this combined training method the training is more efficient. Hence for example if we are using ListMLE, our loss function would be given by

$$L_{mle} = -\sum_{j=1}^{k} \log \frac{\exp(\mu(\pi_j^*))}{\sum_{t=j}^{k} \exp(\mu(\pi_k^*))} \tag{4.8}$$

One problem that may come up during the training of these DNNs together is that the ranges of the DNNs may be different. This is taken care by the fact that we use a range controller in our scoring function as shown in Figure 4.1. Hence, the ranges of all the DNNs are the same. The variance within the ensemble is still present due to the random initialization of the neural networks. By using the range controlling mechanism we both make the model numerically stable and make the training more efficient.

4.5 Different training mechanisms.

Independent training

Implementation yet to be done

Training with mean and restricted output

Implementation yet to be done

Chapter 5

Research Question

The format of this is the same as that of experiments and Results. Also known as Hypothesis.

Chapter 6

Experiments and Results

In this chapter we present the experiments that we conducted and the results we obtained in order to answer the research questions already posed. We first understand the structure of the (meta)data used for meta-training, meta-validation, and meta testing. In the subsequent sections we present the results obtained in detail.

Meta-Data

For comparing our proposed model with other HPO models we would have to run the Bayesian optimization used in our case (with our surrogate and with other surrogates) on a set of different machine learning models. Moreover, the optimum within the HP search space may be different when the ML model is trained on different data sets. Hence, we further need to do multiple HP optimizations of a model each time using a different dataset. In addition to this, due to the stochastic nature of ML models as well as 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 the HPO-B [31] benchmarking in this thesis. Using this benchmark, we do not need to train our ML models from scratch as the meta data contains evaluations of multiple HP configurations for different ML models and datasets. In the rest of the section we discuss about the organisation of the benchmarking meta data.

HPO-B is a benchmark that can be used for doing black box HPO. It can be used for both transfer models and non transfer models. The meta data consists of a list of (hyper-parameter) search spaces. These are hyper parameter search spaces of single models. It is organised in a json format with the structure illustrated in Figure 6.1.

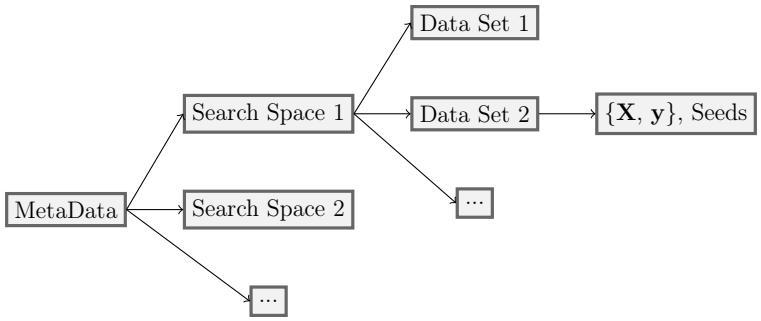

Figure 6.1: Structure of the meta data in the HPO-B benchmark

Where **X** represents the set of configurations for a evaluated for a model in a particular data set and **y** represents the evaluation results. The bayesian optimization used in our thesis can be started with different initial known HP configurations. One initial configuration, called a seed in the meta data, is provided by a set of values (or indices) in **X** and **y**. There are a total of 5 seeds provided for a search space and dataset combination.

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 distlilled the search spaces that have the most datasets and can be split is split into train, validation and test sets.

The meta data set in HPO-B is divided into test and train data. Using meta dataset, one can learn meta learn a model using the training split. Then the model is evaluated in the testing split. This data splitting approach is used both in the baselines and proposed idea models.

There are 2 types of HP optimization surrogates that are studied in this thesis - transfer learning surrogates and non-transfer learning surrogates. This benchmark can be used for analysing both types of models. However, in order cross compare transfer and non-transfer techniques, we only compare against **HPO-B-v3** test split as recommended in the HPO-B paper.

6.1 Baseline Results

To test our proposed model, we used 4 main baseline methods - Random Search, SMBO using Gaussian surrogates, SMBO using Deep Ensembles and FSBO. Library implementation of Random Search and GP surrogates were taken. Whereas the Deep Ensemble and FSBO papers were implemented in this thesis. The results of these baselines are discussed in this section.

In the given meta data if we take the set of all combinations in **HPO-B-**

v3 test split i.e {Search Spaces × Metadata sets × Seeds}, we get 403 HP optimizations. In each optimization, we start with 5 initial seeds given by HPO-B and run the evaluation cycle for 100 iterations. This evaluation cycle is discussed in Algorithm 1 albeit for deep ensembles.

In the whole evaluation cycle, an array of incumbent best (highest) HP evaluation based on the Metadata set is created. The incumbent array for every optimization cycle for a model is compared against other models to create a rank array for each model. For example, if at step 23 the incumbent of model a has a higher value than of model b, then rank of model a at step 23 is lower than model b (When we consider lower rank to be better). The rank arrays of every optimization cycle is averaged to get the rank graph of a particular model. We present this **rank graph** as a primary form of comparison between the models in question.

For example Figure 6.2 compares the rank graph of 2 baselines.

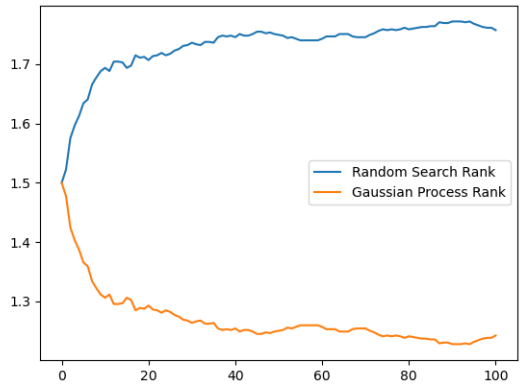

Figure 6.2: Rank Graph of RS and SMBO with GP surrogate

Deep Ensembles

The deep ensembles we used contained 5 neural networks by default. Each neural network contained 2 fully connected neural layers with 32 neurons each. We considered the non-transfer case for deep ensembles first. Hence we did not do any meta training for this model. Each neural network was trained for a 1000 epochs with a learning rate of 0.02 at evaluation cycle. We used Adam as our optimizer.

Figure 6.3 shows the performance of various Deep Ensemble models. We use the best results obtained from the previous baselines for comparing how

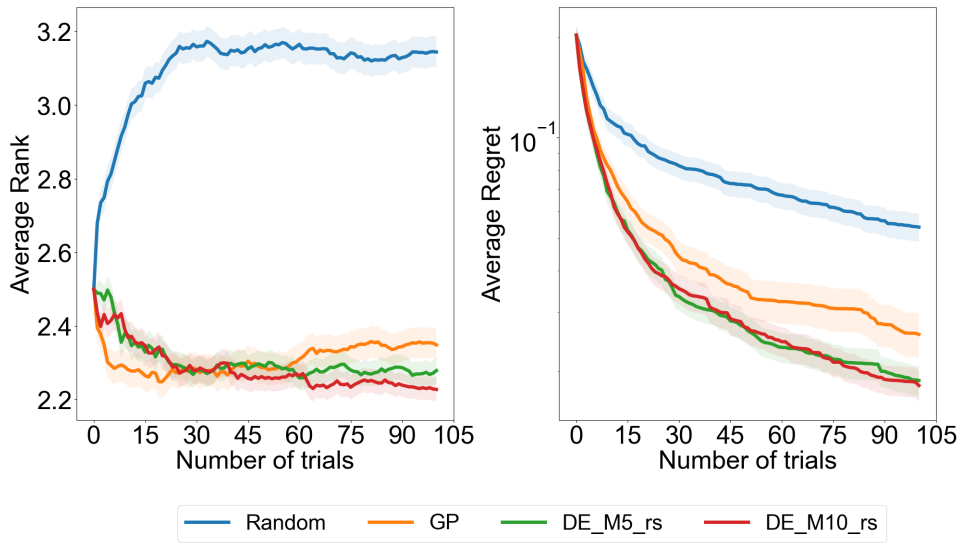

Figure 6.3: Graphs showing performance of Deep Ensembles

Deep ensembles fair in the HPO-B benchmark. In addition to plotting the rank graph, we also plot the average regret that is obtain if we use these models. The average regret on y-axis is plotted in a log scale.

Ablation Required for the raw part missing.

In our study, we studied the following main models of DE

- DE using restart at every evaluation cycle.
- Increasing the number of neural networks in DE to 10.

We observe that using deep ensembles does give us results which are comparable with GP. Increasing the number of neural network ensembles does not seem to give very high performance improvement. This is the case for both the ranking graph and the average regret. Since this ranking would be cluttered for a large ablation study, we also make use of critical rank graphs [30]. This is illustrated in Figure 6.4.

After our analysis, we a couple of advantages of the DE model. First, being a non transfer HPO model we found that it 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 we can scale up our ensemble based on the compute power available to us.

On the negative side, since we do not do any meta training we are bound to (and have to) over fit to the very less data available during the optimization cycle. In addition to this, it does not model discrete or semi discrete spaces

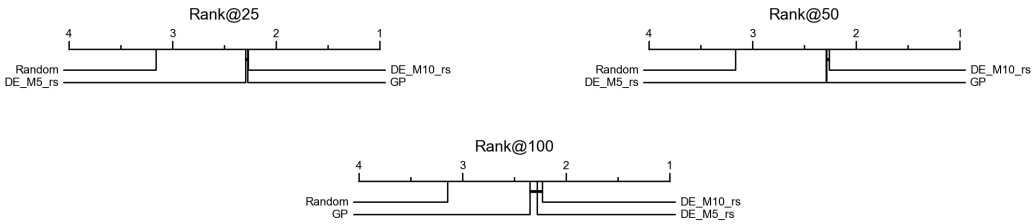

Figure 6.4: DE Ranking at various evaluation cycle steps

correctly. This is because during the learning, the input is assumed to be continuous.

Few Shot Bayesian Optimization (FSBO)

The reason for the selection of FSBO as a baseline was that it was the state of the art in transfer HPO. During our implementation of FSBO, due to the computational resources required for the optimization cycle, we tried to use this model with only single search spaces. We found that using a model with 4 neural network layers with the same width of 32 as used in the Deep Ensemble is the best for the performance of the model. We used early stopping for doing better training as mentioned before. During the fine tuning we ran our model for 500 epochs with a learning rate of 0.03. We used Adam optimiser for both training and fine tuning. However, we used cosine annealing on in the fine tuning phase.

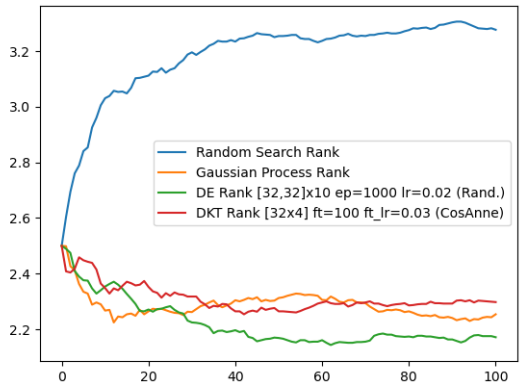

Figure 6.5: Rank graph of self implemented FSBO

We can see in Figure 6.5, our implementation could not give us the state

of the art results. Hence we used the FSBO results stored in the HPO-B benchmark instead for the comparison with the rest of the baselines.

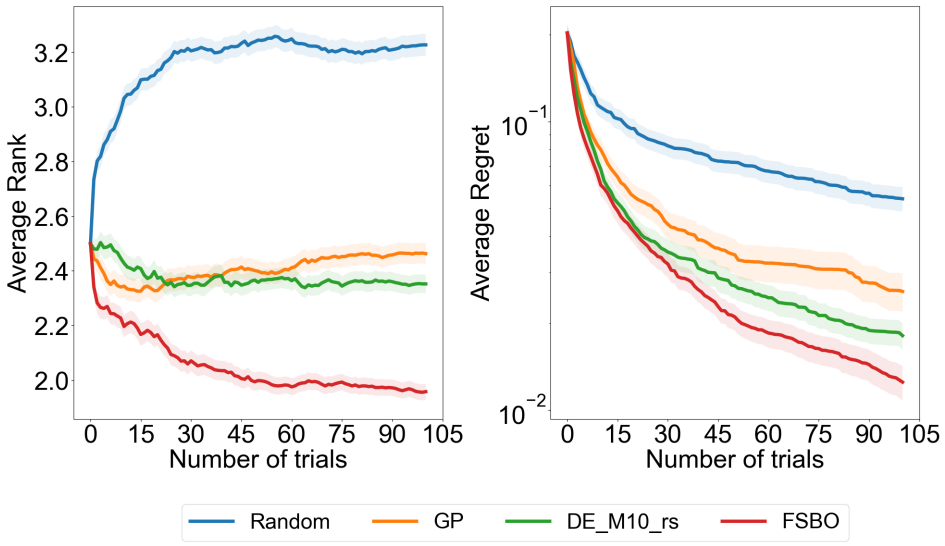

Figure 6.6: Graphs showing best performance of FSBO

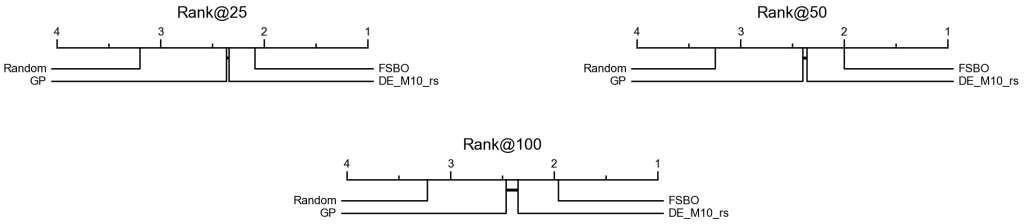

Figure 6.7: Ranking of all baselines at various evaluation steps

From Figure 6.6 it is evident that FSBO as a model vastly outperforms the other models both in the rank graph and the regret graph. Moreover, FSBO is always in a better rank at every evaluation step. This can be seen in Figure 6.7. We use the DE with 10 neural network ensemble because it was giving best results for DE. With the performance of the baselines set, we now move to discuss in detail how our model compares to these methods.

6.2 Ranking Loss model results

In this section we evaluate our built model chronologically first using basic scorer model, then making the model context aware and thereafter adding the

capability of uncertainty to the model.

ListMLE with Basic scoring model

Training of the basic scoring model with the meta data obtained relatively smooth loss curves. Figure 6.8 shows one such loss curve obtained for the search space ID 4796 in the HPO-B meta dataset. However, we did find search spaces that had bad loss curves. Figure 6.9 shows example of one such loss curves. We can deduce in this figure that the validation task is different from the training task. Since the testing/evaluation task may be similar to the training task, we did not do any early stopping of the model (during training) based on the validation losses.

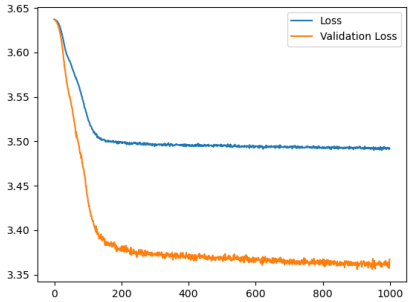

Figure 6.8: Good training loss curve (search space ID 4796)

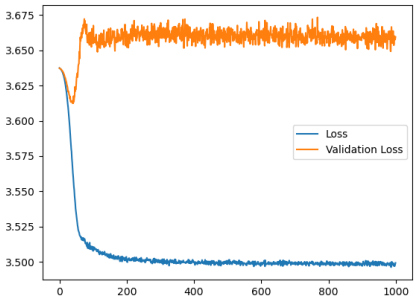

Figure 6.9: Bad training loss curve (search space ID 5896)

With the trained basic scoring model, the optimization runs can be done without any fine tuning. This is nothing but obtaining the relevance scores of the pending HP configurations using our basic scorer. Then evaluating the true HP objective function in the order given by their respective relevance scores. However, a more robust mechanism is to fine tune the scoring model based on the known HP evaluations. Fine tuning for every evaluation step is restarted as discussed in Section 4.3.2.

From Figure 6.10 one can see that the hypothesis regarding fine tune is correct. Fine tuning does help make the model get better results in the long run. One very interesting result we see is that without any fine tuning, the basic scoring model trained with listMLE is very good in the first 5 evaluation steps. We this more closely in the critical graph in Figure 6.11.

The reason for this is that the scorer model learns to rank the best HP configuration found in the meta data. This corresponds to a person trying

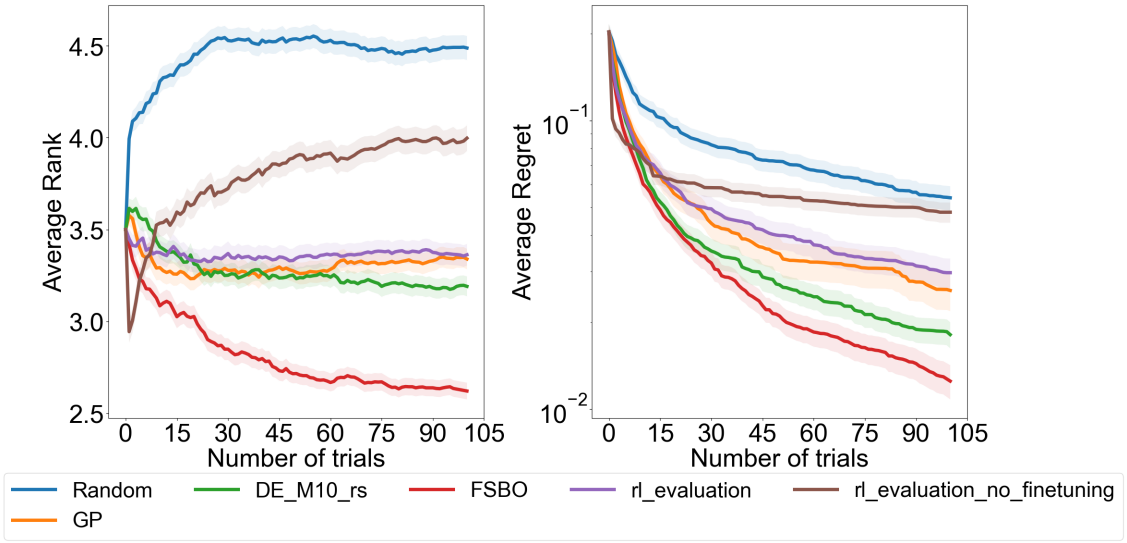

Figure 6.10: Benchmarking basic scoring model trained using ListMLE

out the best configuration seen in his experience. On average this works very well hence the first configurations were good for the non fine tuned model. In the long run however, there is more observed data that the model can rely on. This is only done when the model is fine tuned. This is the reason why the fine tuning scorer works better in the long run as compared to the non fine tuned model.

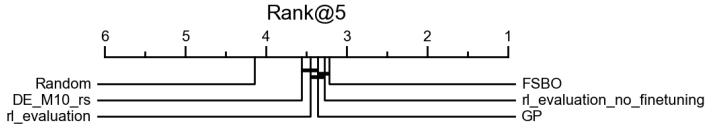

Figure 6.11: Critical rank graph at evaluation step 5

Deep Sets: Context aware scoring model

We found in our research that our model could be improved by making it context aware. For doing this we used an architecture with deep sets described Section 4.4.1. As previously discussed the latent result of the deep set can be used as a context for the scoring model. In order to meta-train this architecture, we needed to change the way we sample the data. We used a support set of 20 data points i.e $\{X_s, y_s\}$ pairs. For each double sampling we sample first the task. Then the support points are sampled without replacement. Then we

sample the query points X_q based on the batch size and list size (By default 100 for both).

During the training however, we found that the loss curves were very jumpy.

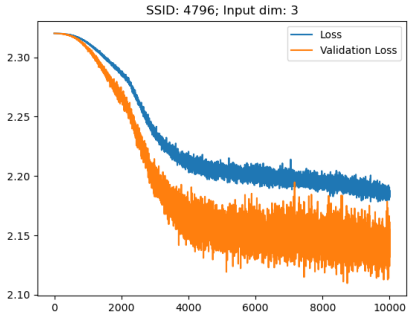

Figure 6.12: Loss curve for model with deep set (SSID: 4796)

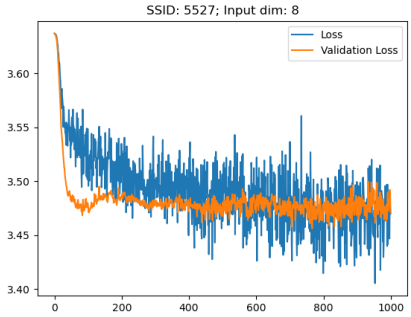

Figure 6.13: Loss curve for model with deep set (SSID: 5527)

Figure 6.12 and Figure 6.13 shows this clearly. We think that this is because of very high representation capacity which is a consequence of using the deep set architecture.

During meta tuning again we have to have a support set and query set for training. We cannot fix the support data points because the evaluation data is very less. For this reason, we use 20% of the points as support points and the rest as query points during the fine tuning process.

The results of this training and fine tuning our model with deep sets are shown in Figure 6.15. In the figure "raw" means that there was no fine tuning performed and the trained model was used as is in the the evaluation/optimization cycle. One advantage we see with this model is that whether fine tuning is done or not, the first few evaluations are better than the state of the art results. This can be seen in the Rank@5 graph critical graph given in Figure 6.14. This is because there is less relearning that happens in fine tuning due to addition of context to our model. In the later evaluation steps, naturally, using fine tuning makes gives better performance.

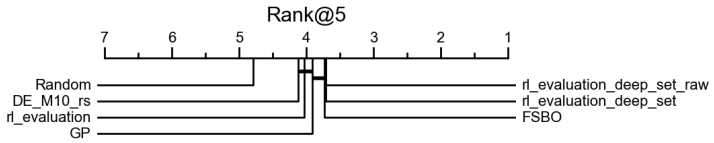

Figure 6.14: Critical rank graph for Model with Deep Set.

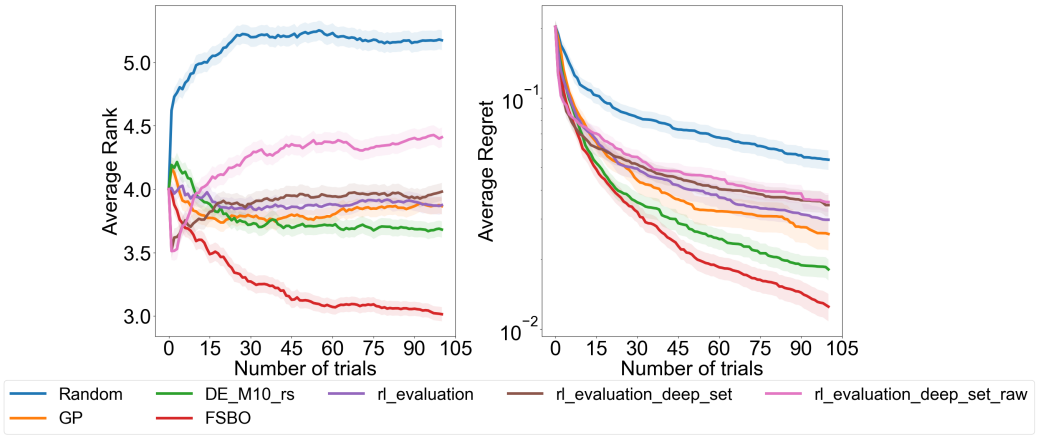

Figure 6.15: Benchmarking for the Deep set evaluation data

Using weighted Loss Function

We discussed that using a weighting loss function makes more sense for us as discussed in Section 3.5. The ablation of using inverse log weighting with and without fine tuning is shown in Figure 6.16. Here we see that there is a big improvement to our evaluation results. This proves the hypothesis that weighting is very essential for getting good results.

The fine tuned results is better than all the baselines except FSBO. In fact, it is better than FSBO in the first few evaluation steps. This can be seen in Figure 6.17.

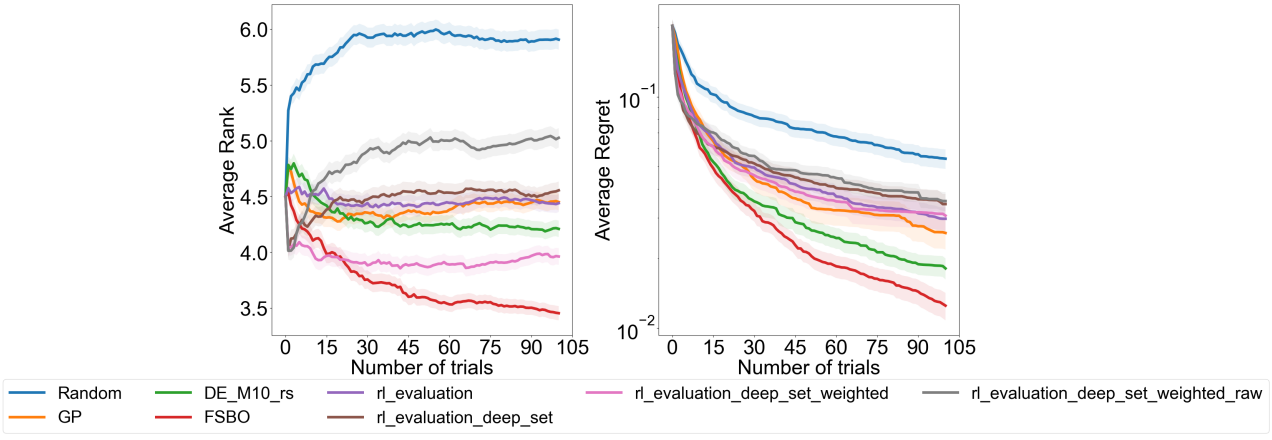

Figure 6.16: Ablation for weighted loss

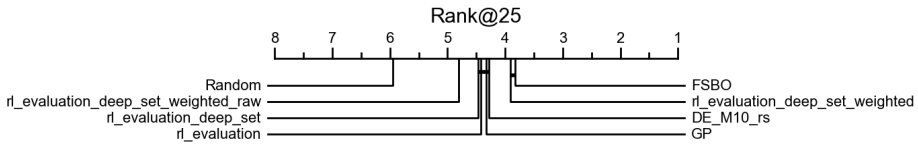

Figure 6.17: Critical rank graph for weighting ablation.

Uncertainty

Uncertainty estimation is very important for every surrogate model that is used in SMBO. This is implemented using the ensemble method described in Section 4.4.2. We see from Figure 6.18 that there is massive improvement of performance in our model after uncertainty is implemented in it. We can also see that it is on par with the state of the art FSBO results.

In the first steps of the optimization cycle, using uncertainty is very important. This is because we do not have enough data to build or fine tune our surrogate model. Using a surrogate without uncertainty in the first steps may lead to an overconfident model that gives bad results. This especially not good because the evaluation cycle is a sequential process. The selection of the next point depends on the surrogate which has been fine-tuned with previously explored points. This is what makes the model with uncertainty (if estimated correctly) superior to other models.

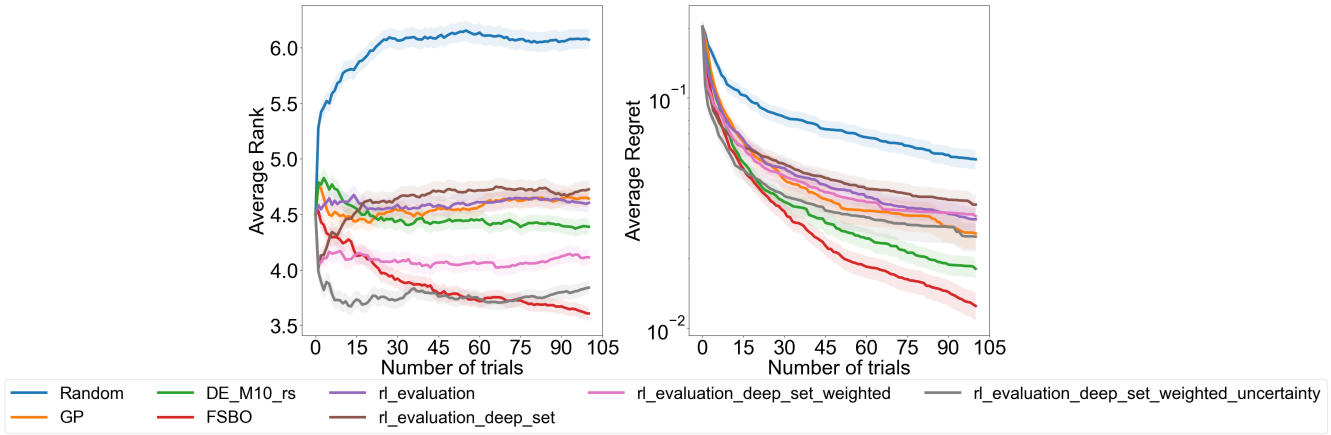

Figure 6.18: Ablation for uncertainty implementation

One thing interesting is that the average regret of the best model we propose is not very good as compared to even the Deep Ensemble baseline. Since the average regret gives a biased estimation of the results and the rank graph

is a more accurate picture of the results, we conclude here that the proposed ranking loss surrogate model is better than the Deep Ensemble baseline.

Finally, in Figure 6.19 we show how each of models built chronologically compare against each other. In this figure we see that the 2 main factors that improved our model were the use of a weighted loss and the use of uncertainty. One anomoly we see is that the addition of deep set itself seemed to give bad results. Nevertheless, we accept it to be better than just using the basic scoring model. Firstly because addition of the deep set improved the performance of the model in the first approximately 30 steps. This is crucial for a sequential process. Secondly, because there is a possibility to reduce the probability of negative learning to occur because of the context encoded using the deep sets. Also one can see that one standard deviation of both the basic scoring model and the deep set model are overlapping. This signifies that their performance difference in the last ranks is not very huge.

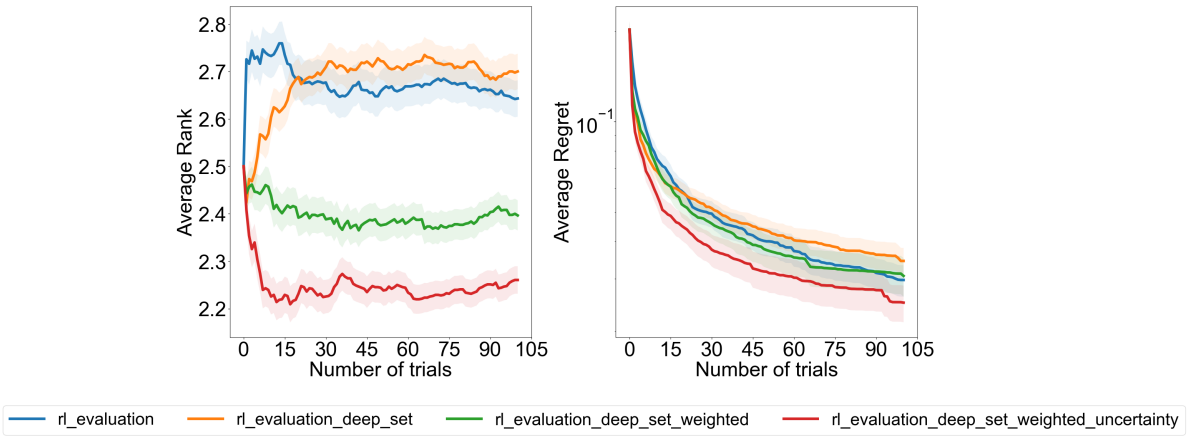

Figure 6.19: Ablation of all RL models

6.3 Advantages and Limitations

Compare this loss function and method with other base lines.. Advantages of the proposed Idea: The amount of data instances for training is exponential in number. Which is very good for a deep learning model For example if we have 100 observation set and we use a list size of 15 to train our model, we will have 100C15 unique instances to train.

Observed disadvantages:

- In our model, a scorer is first learnt and then using the scorer, we rank

the set of objects in question. When optimising the scorer, we ignore the sorting functionality necessary to complete the process of ranking. This is because sorting is non differentiable. This means that the true evaluation of the ranked list is not optimized. This needs to be improved which is done in Pi-Rank paper.

- The learnt model is extremely sensitive to the learning rate and the number of epochs.

- It is assumed that the target task and the training task have the same output range. They must be normalized in order to get the correct results if they are not in the same range.

- Using tanh function restricts the output very much. It may not have the latent space to completely the output.

- Due to disadvantages of exp increasing positive function - Perhaps using other increasing positive function helps? (More reading/research required on this topic

- The search spaces that have less data need more uncertainty The search spaces having more data need less uncertainty.

Negative transfer learning

For some search spaces, the validation errors of the ranking loss model did not reduce at all during its training. In fact the validation loss became worse. Figure 6.20 shows an illustration of this for the search space id 5527. This is a classic example of negative transfer learning [38]. Negative transfer learning occurs when the set of source tasks (here, training datasets) is very different from the target tasks (here, validation datasets).

This problem should occur only in the cases where the model is context free. In our case, the ranking loss model with deep set is context aware and results like these were unexpected. Hence the issues with negative transfer learning remains unsolved when using our model. One remedy for getting around this problem is to fine tune the model with a larger number of epochs. Nevertheless, we cannot guarantee that the fine tuning will re-learn from the small amount observed data points during the optimization cycle.

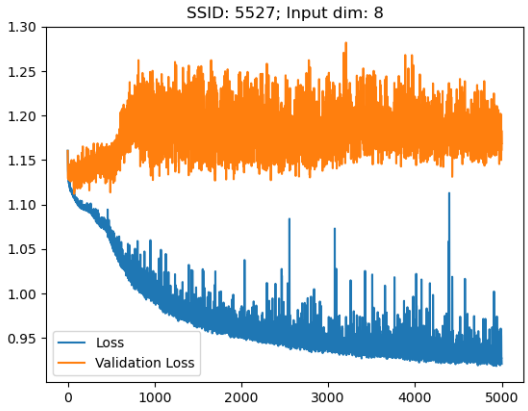

Figure 6.20: Training loss curve of ranking losses

6.4 Evaluation

6.4.1 Testing

explain how a ranking graph works ar implemented Explain the regret rank@ some location.

6.4.2 Ablation

Result tabulation of case study: sorting: 1. Within range 2. Outside range mean of 3 times should be written.

show the results of raw without deep set.

Next show different strategies used for building the ranking loss model one step at a time. First with only scorer. then with deep set. Then with raw deep set fine tuning and deep set adding uncertainty

Checking the early stop and hypothesing why is was wrong.

box plot variation of each of the scorers... for 1 or more data sets?

show results of independent training and training with output restriction

what about training independently, this requires normalization. as explained by sebastian.

Chapter 7

Conclusion

7.1 Further work

Further study required with other baselines that deal with ranking loss.

In our methods we used the ensemble of DNNs which output a list of results. We use this results to calculate the mean and the variance of our prediction. However, the deep ensemble paper proposes a method to directly calculate the mean and variance. However, the integration of this idea with ranking losses is non-trivial. Hence the usage of this type of loss with the ranking loss function can be taken up in further research in order to check whether their is improvement in the results or not.

Working with continuous HP spaces. How about dividing the space into areas and use 1 HP configuration as a representative of the space (in the euclidean sense) Then select the best region and subdivide the space and continue the process. There are limitation, cannot really guarantee the optima will be found like the gradient methods. Hence this method is suboptimal to the gradient based HP methods for continuous search spaces.

7.2 Conclusion

This is the conclusion

Bibliography

[1] M. O. Ahmed and Simon Prince. Tutorial 8 - bayesian optimization. *BorealisAI*, 06 2020.

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

[3] 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.

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

[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(11):5140–5154, Nov 2008.

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

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

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

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

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

[36] 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.

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

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

[39] 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].

[40] 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.

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

[42] 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.

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

Appendix A

More information

This thesis was completed in the representation learning lab of Albert-Ludwig-Universität Freiburg. (Figure A.1)

Figure A.1: Logo: Albert-Ludwig-Universität Freiburg