Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

# Acceleration of ListNet for Ranking Using Reconfigurable Architecture

Qiang Li

Supervised by Dr David B. Thomas and Prof Peter Y.K. Cheung

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College, October 2019

# Abstract

Document ranking is used to order query results by relevance with ranking models. ListNet is a well-known ranking approach for constructing and training learning-to-rank models. Compared with traditional learning approaches, ListNet delivers better accuracy, but is computationally too expensive to learn models with large data sets due to the large number of permutations and documents involved in computing the gradients. Currently, the long training time limits the practicality of ListNet in ranking applications such as breaking news search and stock prediction, and this situation is getting worse with the increase in data-set size. In order to tackle the challenge of long training time, this thesis optimises the ListNet algorithm, and designs hardware accelerators for learning the ListNet algorithm using Field Programmable Gate Arrays (FPGAs), making the algorithm more practical for real-world application.

The contributions of this thesis include: 1) A novel computation method of the ListNet algorithm for ranking. The proposed computation method exposes more fine-grained parallelism for FPGA implementation. 2) A weighted sampling method that takes into account the ranking positions, along with an effective quantisation method based on FPGA devices. The proposed design achieves a 4.42x improvement over GPU implementation speed, while still guaranteeing the accuracy. 3) A full reconfigurable architecture for the ListNet training using multiple bitstream kernels. The proposed method achieves a higher model accuracy than pure fixed point training, and a better throughput than pure floating point training. This thesis has resulted in the acceleration of the ListNet algorithm for ranking using FPGAs by applying the above techniques. Significant improvements in speed have been achieved in this work against CPU and GPU implementations.

# Acknowledgements

I would like to thank my supervisor, Prof. Peter Cheung, without whom this thesis would not have been possible. He actively encouraged me to explore my own research interests. He also advised me to improve on my English language skills and helped me build my self-confidence. In addition, he provided funding for my research. I truly appreciate his support.

Thank you to my co-supervisor, Dr. David Thomas, for his guidance during our weekly meetings. Through our fruitful discussions, I gained a deeper understanding on how to conduct research. It is a skill I will treasure forever.

I would also like to thank Shane Fleming, who helped me immensely in my research and publications. To Erwei Wang, He Li, Ben Chua, and Jianyi Cheng, it has been an enjoyable journey together. To the rest of the CAS group, thank you for the fun and laughter.

Most importantly, I also thank my family for their continuous support in my life - my parents, who spent their precious time and energy to nurture me, and my siblings, who have been there for me through my ups and downs. I love you three thousand times.

# Declarations

## Declaration of Originality

I herewith certify that the work presented in this thesis is my own. All other related work are appropriately referenced.

## Declaration of Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

# Contents

**Abstract** **i**

**Acknowledgements** **iii**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.5 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Background** **9**

2.1 Introduction of Learning to Rank . . . . . . . . . . . . . . . . . . . . . 9

2.2 Major Ranking Approaches . . . . . . . . . . . . . . . . . . . . . . . . 13

2.2.1 The Pointwise Approach . . . . . . . . . . . . . . . . . . . . . . 13

2.2.2 The Pairwise Approach . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.3 The Listwise Approach . . . . . . . . . . . . . . . . . . . . . . . 16

2.2.4 Ranking Approach Overview . . . . . . . . . . 18

2.3 Evaluation Measures . . . . . . . . . . 19

2.4 Review of Top-$k$ ListNet . . . . . . . . . . 21

2.5 Related Work . . . . . . . . . . 26

2.5.1 Algorithm Optimisation . . . . . . . . . . 26

2.5.2 Hardware Acceleration . . . . . . . . . . 28

2.6 Design Flow of Hardware-Based Accelerator . . . . . . . . . . 32

2.7 Summary . . . . . . . . . . 33

**3 Accelerating Top-$k$ ListNet Training for Ranking Using FPGA 34**

3.1 Introduction . . . . . . . . . . 34

3.2 Analysis of Top-$k$ ListNet . . . . . . . . . . 36

3.2.1 Challenge of Top-$k$ ListNet . . . . . . . . . . 36

3.2.2 Redesign of Top-$k$ ListNet Computation . . . . . . . . . . 37

3.2.3 Benefit of Precomputation for Top-$k$ ListNet . . . . . . . . . . 40

3.3 Hardware Mapping . . . . . . . . . . 41

3.3.1 HW/SW Partitioning and Communication . . . . . . . . . . 41

3.3.2 Data Parallelism . . . . . . . . . . 45

3.3.3 Fine-grain Pipeline Parallelism . . . . . . . . . . 46

3.4 Experiments and Analysis . . . . . . . . . . 47

3.4.1 Analysis of the Speedup . . . . . . . . . . 48

3.4.2 Ranking Accuracy . . . . . . . . . . 50

3.4.3 Scalability of Speedup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

3.5 Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

3.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

**4 Accelerating Position-Aware Top-$k$ ListNet under Custom Precision Regimes 54**

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.2 Position-Aware Sampling Approach . . . . . . . . . . . . . . . . . . . . . . . . 56

4.2.1 Optimisation of Top-$k$ ListNet Scheduling . . . . . . . . . . . . . . . . . 56

4.2.2 Position-Aware Sampling Scheme . . . . . . . . . . . . . . . . . . . . . . 57

4.3 Mixed Precision Fixed Point Implementation . . . . . . . . . . . . . . . . . . . 60

4.3.1 Organisation of Computation Tasks . . . . . . . . . . . . . . . . . . . . . 60

4.3.2 Limitation of Traditional Fixed Point Implementation . . . . . . . . . . . 63

4.3.3 Batch Fixed Point Implementation . . . . . . . . . . . . . . . . . . . . . . 64

4.3.4 Case Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4.4 Experiment Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.4.1 Analysis of the Performance . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.5 Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

4.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**5 Novel Reconfigurable Implementation of Top-$k$ ListNet on FPGA 71**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

5.2 Training with Multi-Kernels under Different Data Representations . . . . . . . . 73

5.3 Experiment Results and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . 74

5.3.1 Performance of Swapping from Floating Point Kernel to Fixed Point Kernel 75

5.3.2 Performance of Swapping from Fixed Point Kernel to Floating Point Kernel 79

5.4 Performance Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

5.5 Expectation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

5.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**6 Conclusion** **85**

6.1 Summary of Thesis Achievements . . . . . . . . . . . . . . . . . . . . . . . 85

6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

**Bibliography** **90**

# List of Tables

2.1 The process of pointwise approach. The technology of pointwise approach is regression . . . . . 14

2.2 The process of pairwise approach. The technology of pairwise approach is classification . . . . . 16

2.3 The process of listwise approach. The output space of listwise approach is exactly the same as that of the task . . . . . 17

2.4 Summary of ranking approaches . . . . . 18

3.1 Size of communication data between software and hardware of one-task hardware implementation for Top-2 ListNet . . . . . 44

3.2 Size of communication data between software and hardware of two-task hardware implementation for Top-2 ListNet . . . . . 45

3.3 Utilization of hardware resource for Top-2 ListNet on Xilinx ZCU102 development board . . . . . 49

3.4 Time consumption of different tasks on CPU VS FPGA in one epoch . . . . . 49

3.5 The contribution of data parallelism with unrolling factor $U = 150$ (under fanout limitation) and pipeline with initiation interval $II = 11$ to speedup. Metric of computational efficiency is FLOP/s/DSP. . . . . . 50

4.1 Consumption of hardware resource and speedup over GPU implementation for different data types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4.2 Consumption of hardware resource under different numbers of bits for batch quantisation implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

4.3 The performance comparison of different data types implemented on FPGA. The standard time consumption on CPU is 420.05s per epoch, GPU is 45.8s . . . . . 69

4.4 The resource required and computation efficiency for different data types implemented on FPGA. Metric of computational efficiency is OP/s/DSP . . . . . . . 69

4.5 The contribution of data parallelism with unrolling factor $U = 150$ (under fanout limitation) and pipeline with initiation interval $II = 11$ to speedup. Metric of computational efficiency is Gop/s/DSP . . . . . . . . . . . . . . . . . . . . . . 69

4.6 The consumption of hardware resource under different strategies for 8-bit batch quantsation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

# List of Figures


2.1 Searching result for query "learning to rank" [1] . . . . . . . . . . . . . . . . . . . 10

2.2 Framework of learning to rank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3 Machine-learning-based search engine overview . . . . . . . . . . . . . . . . . . . 13

2.4 Computation process of Top-$k$ ListNet . . . . . . . . . . . . . . . . . . . . . . . . 22

2.5 FPGA-based Accelerator Design Flow for Listwise Approach . . . . . . . . . . . 32

3.1 An example of document representation in benchmark data set LETOR 4.0 . . 36

3.2 Time consumption on our Intel Xeon CPU for different tasks in Top-2 ListNet using the source code of RankLib . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.3 Time consumption on our Intel Xeon CPU for different probability computation methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.4 Communication data among different tasks in computation per training epoch . 41

3.5 The architecture of FPGA system integrated with Top-$k$ ListNet IP . . . . . . . 44

3.6 Data flow between different tasks on FPGA . . . . . . . . . . . . . . . . . . . . . 46

3.7 Structure of 2-layer neural network ranking model . . . . . . . . . . . . . . . . . . 47

3.8 The organisation of Computation Unit for hidden layer (hCU) . . . . . . . . . . 48

3.9 Accuracy comparison among different algorithms, NDCG@$k$ means it calculated at the first $k$ positions. . . . . 51

3.10 The complete execution speedup of Top-$k$ (k=2) ListNet with different $n_d$ . . . . 52

4.1 Stochastic Top-3 ListNet. Each box represents a selected permutation, which is selected randomly . . . . . 56

4.2 Position-aware Top-3 ListNet. Each box is one position, the number of samples in each box decreases from top to bottom . . . . . 58

4.3 NDCG@$k$ comparison among different sampling factor sets, NDCG@$k$ means it calculated at the first $k$ positions . . . . . 60

4.4 Performance comparison among different ranking algorithms using CPU platform over 300 epochs . . . . . 61

4.5 The organisation of computation tasks on FPGA . . . . . 62

4.6 Distribution of gradient values of loss function over 300 epochs . . . . . 63

4.7 The trend of mean and maximum gradient value of loss function over 300 epochs 64

4.8 The theory of batch implementation . . . . . 65

4.9 Accuracy loss with different fixed point bit-widths $bw_1$ compared to floating point implementation . . . . . 67

5.1 Accuracy loss for different representation types over 300 epochs compared to single floating point . . . . . 74

5.2 Accuracy for swapping once from floating point kernel to 8-bit fixed point kernel at different epochs . . . . . 76

5.3 Accuracy for swapping once from floating point kernel to 16-bit fixed point kernel at different epochs . . . . . 77

5.4 Accuracy with swapping once from floating point kernel to 8-bit fixed point kernel at different time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.5 Accuracy with swapping once from floating point kernel to 16-bit fixed point kernel at different time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.6 Accuracy for swapping once from 8-bit fixed point kernel to floating point kernel at different epochs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

5.7 Accuracy for swapping once from 16-bit fixed point kernel to floating point kernel at different epochs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

5.8 Accuracy for swapping from once 8-bit fixed point kernel to floating point kernel at different time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

5.9 Accuracy for swapping once from 16-bit fixed point kernel to floating point kernel at different time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

# Chapter 1

# Introduction

In recent years, with the increase of data-set size for document ranking, high-quality ranking models are required. It is promising to learn ranking models by leveraging machine learning technologies. However, when applying machine learning, it faces challenges in computational complexity for modern ranking algorithms. In this chapter, we display the difficulties to accelerate state-of-the-art ranking algorithms, and present several effective techniques to address the difficulties. These techniques will be explained in following chapters in details.

## 1.1 Motivation

Ranking is the process of ordering a set of documents according to relevance to a query [2, 3, 4]. A well-known example is in the ordering of links returned from a search engine such as Google: links near the top of the page should be those which are most likely to satisfy the user's query. Ranking relies on the definition and training of a ranking model $f(q, d)$ which accepts a query $q$ and a set of documents $d$, and returns a ranked list of documents. To improve ranking the use of machine-learning techniques has been explored to automatically construct and tune the ranking model, which has led to a general field called learning to rank, with three main approaches:

- pointwise approach;

- pairwise approach;
- listwise approach.

Pointwise and pairwise learning approaches are currently used in practice, as they have relatively low compute requirements. In the pointwise approach a ranking function is applied independently to each individual document to determine individual scores. These scores are then sorted to determine the overall ranking [5, 6]. In the pairwise approach pairs of documents are considered at a time and their relative rankings are determined [7, 8, 9]. The advantage of both these approaches is that existing methods, such as regression techniques for pointwise and classification techniques for pairwise, can be directly applied. However, because ranking is about learning a complete ordering on documents, it is difficult to accurately derive the relative position of the documents in the final ranked list using these two approaches [10].

The listwise approach is a newer method which addresses the issue of relative ranking accuracy by constructing the ranking function using lists of documents, rather than considering single document or pairs of documents [11, 12]. While this improves the accuracy of the ranking, it scales poorly: if the number of documents is $n_d$, then the naive listwise approach takes $O(n_d!)$ time. To address this scaling problem, Top-$k$ ListNet was proposed where instead of all permutations being considered, they are only clustered by the first $k$ documents [13, 14, 15].

Top-$k$ ListNet reduces the number of permutations that need to be considered from $n_d!$ to $\frac{n_d!}{(n_d-k)!}$. However, as the number of documents being ranked is usually large ($n_d > 1000$), $k$ is generally limited to 1, as for $k > 1$ the time complexity is still quite high. For many applications, the high time complexity can be a problem, for example, searching breaking news, where the ranking model needs to be updated and retrained frequently so that it can reflect the freshness of more recent documents [16, 17, 18, 19]. Wang et al, give a good example for the importance of freshness [20]: a user types the query “Apple Company” into a search engine, with the goal of finding breaking news relevant to this company. The documents which cover the release of iPhone 4S were quite relevant on October 4, 2011; however, one day later these documents became less relevant as the Apple founder Steve Jobs had just passed away. So a document

which was quite relevant to a certain query may become less relevant than newer documents. If we want to exploit the potential gains in accuracy from Top-$k$ ListNet approach, then we need to find a way to decrease its training time.

In this research, we investigate the training process of Top-$k$ ListNet by setting up a randomly initialised network, to reveal the bottleneck in computation. Then we offer local optimal solutions by redesigning the Top-$k$ ListNet algorithm and applying hardware acceleration to tackle the challenge of high training time. We hope to improve the training speed of Top-$k$ ListNet and maintain the model accuracy at the same time, so as to bring an alternative for commercial applications.

## 1.2 Challenges

As listwise approach, Top-$k$ LisNet is quite effective in the application of ranking, but the intensive computation restrains it in real applications. Like other neural network algorithms, the Top-$k$ ListNet algorithm consists of three basic steps: forward propagation, loss function and back propagation. Forward propagation is not expensive in computation, which just calculates the score for each candidate document. The loss function and back propagation are the computationally intensive steps because of the large number of permutations involved in computing the gradients. The loss function and back propagation necessarily need to access all of the possible permutations at each epoch in order to evaluate the distance between each possible permutation and ground truth permutation. As mentioned in Section 1.1, the number of possible permutations is $\frac{n_d!}{(n_d-k)!}$, where $n_d$ is the number of the documents. This makes the Top-$k$ ListNet algorithms time-consuming, especially when large data sets are targeted. One possible solution is parallel implementations of Top-$k$ ListNet, however, it is not straightforward. There are several challenges in the acceleration of Top-$k$ ListNet using hardware:

(1) The first challenge of accelerating the Top-$k$ ListNet algorithms in parallel computing devices is the computational dependence between different permutations. The evaluation of the divergence loss which needs to access every possible permutation is the dominant computational

bottleneck. Therefore, accelerating the permutation computation is a crucial task in order to allow the listwise approach to learn ranking models with large-scale training data sets. However, the probability of next permutation is calculated by changing the document order of last permutation, that means the permutation probability is calculated sequentially and there is dependence between different permutations. In order to achieve useful speedup using parallel computing platforms, we need to expose the parallelism first.

(2) The second challenge is how to select high-weighted samples from the whole data set which has a large population. As mentioned in Section 1.1, the full set of the permutation class is $\frac{n_d!}{(n_d-k)!}$, which is intractable in practice. Thus it is necessary to using sampling approaches to reduce the scale of computation in the model training. However, it is difficult to select high-weighted samples using traditional sampling methods since there are many more low-weighted samples.

(3) The third challenge of accelerating the Top-$k$ ListNet algorithms in parallel computing devices is how to maintain the model accuracy using low precision training. The quantisation implementation has already become an important technique in the acceleration of the Top-$k$ ListNet algorithm, as the low precision representation is able to achieve high computation speed. However, the quantisation method would strongly influence the model accuracy if the number of bits is not enough to represent the whole range of the variable faithfully. Thus it is quite necessary to explore effective quantisation methods on listwise algorithms in ranking applications, as Top-$k$ ListNet has variables with a large range.

(4) The fourth challenge is how to deal with the trade-off between model accuracy and training time. As we know, the fixed point operation consumes less resource and leads to a higher parallelism compared to floating point data paths. That means the fixed point computation can minimise training latency. However, apparently the fixed point computation sacrifices the model accuracy. Floating point training can guarantee the model accuracy, but it requires a high computation time. Therefore it is necessary to develop new methodologies to improve the trade-off.

In order to improve the practicality of information retrieval, multi-core computational plat-

forms, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs), have been widely used to accelerate ranking algorithms [21, 22, 23]. However, for current computational capability of CPUs and GPUs, it is difficult to keep up with the pace at which data is increasing dramatically. Thus we turn to the reconfigurable device: Field Programmable Gate Arrays (FPGAs) in this work. FPGAs can be used as customized computing engines for accelerating many machine learning algorithms due to its bit-level parallelism. By properly optimising the computation task so that it can be operated in parallel, and mapping the task into an FPGA, a competitive acceleration can be achieved compared to the respective CPU and GPU implementations [24, 25, 26]. Furthermore, FPGAs make it possible to explore the potential of any custom precision format for neural network parameters. Comparing to implementations with full precision (floating point) which is the default approach in the ranking algorithm, low precision (fixed point) operators consume less resource, and thus allow for more parallelism. Last but not least, reconfiguration is basically a feature in modern FPGAs that enables users to change their functions at running time according to the application requirement. This capability can be utilised to improve the training speed of floating point implementation and model accuracy of fixed point implementation. In this research, the FPGA platform used is Xilinx UltraScale+ ZCU 102 board, and the RTL code is transcompiled from high-level language C++.

## 1.3 Contributions

This research aims to accelerate the training process of the Top-$k$ ListNet algorithm on FPGAs by proposing new computation methods for algorithms and novel customized precision method, in order to largely utilize the advantages and characteristics of FPGAs. The new computation method of the Top-$k$ ListNet training exposes more fine-grained parallelisms, as such, parallel computation can be realized from node to node by applying loop unrolling directive using high-level synthesis, and pipelining computation can be implemented from permutation to permutation by applying pipelining directive using high-level synthesis. The novel customized precision method improves the training speed of Top-$k$ ListNet by implementing the back

propagation arithmetic operators in fixed point on FPGAs, but within acceptable accuracy loss. As such, more parallel operators can be instantiated for a given resource budget, and thus improving the computation speed. The thesis also investigates how to utilise the differentiating reconfiguration capability of FPGAs for applications with large data sets, in order to tackle the trade-off between long training time under floating point implementation and low model accuracy under fixed point implementation. This is achieved by switching between floating point kernel and fixed point kernel automatically. The results presented in the thesis denotes that significant improvements in speed can be attained with the proposed FPGA implementation techniques compared to the respective CPU and GPU implementations.

This research achieves several contributions on developing the hardware-based acceleration of the Top-$k$ ListNet algorithm for ranking. These contributions are summarized as follows (the details are given in each chapter):

- A new computation method, which exposes more parallelism, allowing the computation in parallel to address challenge (1). It is the first work that presents parallel FPGA architectures for the Top-$k$ ListNet algorithm, and shows a 3.21x speedup compared to that of the CPU implementation.

- A novel biased sampling method, which takes the importance of positions into account to address challenge (2). This method has a higher probability to select high-weighted samples, leading to significant accuracy gains.

- A novel quantisation method based on FPGA devices for variables with a large range to address challenge (3). This method organises the whole range of the variable to several batches, and associates each batch with a different fractional precision. It needs fewer bits to represent the whole range, fewer bits consume less resource and delivers a higher degree of parallelism for a fixed hardware resource. FPGA implementation based on the quantisation method shows a 4.42x speedup over the GPU implementation.

- A methodology that allows switching between floating point and fixed point kernels automatically via full reconfiguration, which improves the trade-off between model accuracy

and training time, to address challenge (4).

## 1.4 Thesis Outline

This thesis contains six chapters. In this chapter, the motivation, challenges and contributions of this work are presented.

Chapter 2 gives an introduction to the theory of ranking algorithms (the pointwise, pairwise, and listwise approaches) in details. It contains the concrete training process of these algorithms to follow the remaining chapters. Furthermore, the main advantages and shortcomings of ranking algorithms focusing on the model accuracy and the training complexity are analysed and concluded in details. Then the main evaluation measures for ranking model are provided. After that, the Top-$k$ ListNet approach is reviewed, and the main bottleneck of Top-$k$ ListNet is identified, especially for big data applications. Finally, a complete literature review is presented as a separate section, together with the acceleration of ranking algorithms using multi-CUPs, GPUs, and FPGAs in previous works.

Chapter 3 meets challenge (1), it investigates the acceleration of ListNet training using FPGAs, and improves the training speed by using hardware-oriented algorithmic optimisations, and by transforming algorithmic structures to expose parallelisms. The parallelism is ready to be exploited by implementing a randomly initialised the ListNet algorithm on FPGA devices. The performance is compared to existing ranking approaches, and also to the respective CPU implementation. Analysis on the results is also provided.

Chapter 4 meets challenges (2) and (3), an FPGA-based accelerator for Top-$k$ ListNet under custom precision regimes is proposed . This position-aware sampling approach takes the importance of ranking positions into account, and shows a better accuracy than previous sampling methods. An effective quantisation method based on FPGA devices for a randomly initialised the ListNet algorithm is proposed, which organises the gradient values to several batches, and associates each batch with a different fractional precision. We implemented our approach on a Xilinx UltraScale+ board and applied it to the benchmark data set for ranking. The ex-

periment results show a significant improvement in the training speed, and also computation efficiency (OP/s/DSP).

Chapter 5 meets challenge (4), an multiple kernel training method is proposed which is based on the differentiating reconfigurable feature of FPGAs. This new method deploys multiple functions, which are the fixed point function and floating point function. The function on board can be interchanged during runtime, which is decided according to the model accuracy and training time requirements. We implemented the methodology for randomly initialised position-aware Top-$k$ ListNet on FPGAs, the performance is compared to pure floating point training and fixed point training, great balance is attained.

Finally, Chapter 6 summarizes the current state of our work and discusses future research directions.

## 1.5 Publications

This thesis has led to the following published conference papers:

- Qiang Li, Shane T. Fleming, David B. Thomas, Peter Y. K. Cheung. Accelerating Top-$k$ ListNet Training for Ranking Using FPGA. IEEE International Conference on Field Programmable Technology (FPT), December, 2018.
- Qiang Li, Erwei Wang, Shane T. Fleming, David B. Thomas, Peter Y. K. Cheung. Accelerating Position-Aware Top-$k$ ListNet for Ranking under Custom Precision Regimes. International Conference on Field Programmable Logic and Applications (FPL), September, 2019.

# Chapter 2

# Background

With the fast development of learning to rank, different ranking approaches have been proposed. However, different approaches have their own advantages and shortcomings. In order to obtain high-quality ranking models with low training latency, existing ranking approaches and acceleration work are briefly depicted in this chapter.

## 2.1 Introduction of Learning to Rank

In order to improve the efficiency of information retrieval on FPGAs, we need to know how to order the relevant documents which match the queries first, and then consider how to utilize FPGAs to implement ranking algorithms. A typical application of ranking algorithms is search engine.

A search engine architecture generally contains six major components: crawler, parser, indexer, link analyzer, query processor, and ranker. When a user submits queries to a document retrieval system, the crawler collects the relevant documents from the data set, according to some prioritization strategies. The parser generates index terms and a hyperlink graph for these documents after analyzing them. The indexer creates the indexes or data structures which enable fast search of the documents. The link analyzer determines the importance of each document

**Query=learning to rank**

| | |
|---|---|
| a. http://web.mit.edu/shivani/www/Ranking-NIPS-05/ | 1 (Good) |
| b. http://www.learn-in-china.com/rank.htm | 0 (Bad) |
| c. http://research.Microsoft.com/~letor/ | 3 (Perfect) |
| … … | |

Figure 2.1: Searching result for query "learning to rank" [1]

on the basis of the document graph. The query processor is the interface between users and search engines, which transforms the input queries to index terms which can be understood by search engines. The ranker is the central component which is responsible to rank indexed documents using ranking models [1]. For example, when a user types query "learning to rank" in the search engine as Figure 2.1, the engine system processes the query as followings:

1) The indexer generates the indexes for documents to enable efficient document retrieval;

2) The link analyzer finds the candidate documents (a, b, c...) that satisfy the input query – "learning to rank";

3) The ranker calculates scores for the documents (a, b, c...) with the ranking model, which is usually the combination of many information retrieval features, such as frequency of the term (noted as $TF$), the length (noted as $LEN$), etc.. Then all these documents are sorted in descending order of their scores.

Due to its importance, machine learning technologies have been leveraged to build innovative and effective ranking models, which is named learning to rank. In machine learning, there are four key components, which are *input space*, *output space*, *hypothesis space*, and *loss function* [1].

**(1)** The *input space*, which contains the relevant documents for ranking. The input, or so-called training set, is structured with queries that are issued. As mentioned before, a data set has lots of queries, and every query has $n_d$ relevant documents. These documents are usually represented by feature vectors which are chosen based on the application. Each document

feature is real-valued data.

**(2)** The *output space*, which contains the learning results in regard to relevant documents. In machine learning, the *output space* is usually decided by the application. The *output space* is the set of real numbers in regression; the *output space* is discrete categories in classification, which is not associated with positions; while in ranking, it should be a permutation, which is position based.

**(3)** The *hypothesis space*, which defines the function mapping the input to the output. That is, the function takes the relevant documents as input, and generates predictions for each document.

**(4)** The *loss function*, which measures the compatibility between the prediction of the model and the ground truth. It is clear that *loss function* plays a key role in machine learning, because it reflects the distance between the prediction and the target. With the *loss function*, the differences between the predictions and the ground truths are collected, and the parameters of ranking model are optimised by minimising the differences.

In ranking, the ranking model takes documents as inputs and assigns a score to each document; then the documents are sorted in descending order of the scores, and a ranked list is returned back to user. The order of the ranked list should reflect how relevant each document is to the query, with the most relevant coming first.

In learning, there is a training set consisting of $n_q$ queries, where each query has $n_d$ relevant documents and each document is represented by $n_f$ features. This results in an input data set $\boldsymbol{x}$ of size $n_q n_d n_f$. The ground truth labels $\boldsymbol{y}$ are given as a reference ordering of the documents, where an ordering is a permutation of $n_d$ documents for each query, and the size of $\boldsymbol{y}$ is $n_q n_d$. The goal of learning is to train the ranking model so that the output of the ranking model is as close to the ground truth as possible, as displayed in Figure 2.2. The learning process tunes the ranking model by three steps: first, applying the training data to the model; second, comparing the model output to the ground truth using a *loss function*; and finally updating the parameters of the model to minimise the error [1]. The overall structure of machine-learning-based search

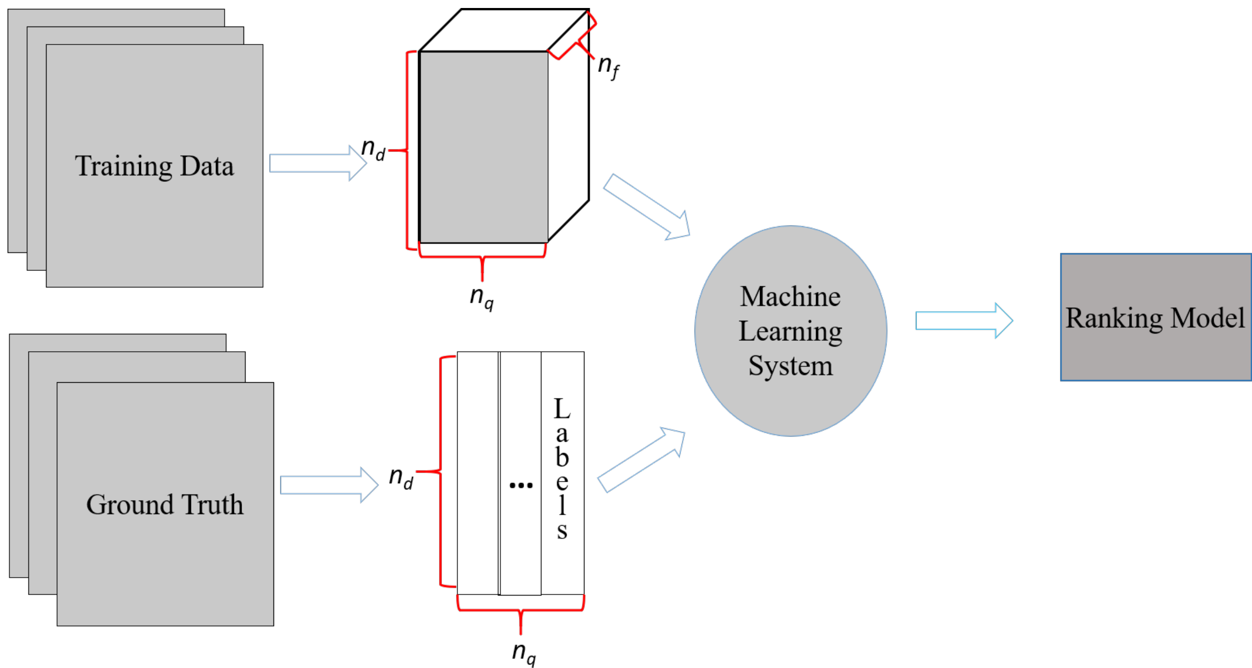


Figure 2.2: Framework of learning to rank

engine is shown in Figure 2.3.

In order to improve the accuracy of ranking model, a more sophisticated technique for hyper parameter tuning is widely used, which is called cross-validation. In cross-validation, a learning-to-rank algorithm is adopted to train ranking models, and then the model is adjusted using the validation fold. For example, in five-fold cross validation, each training set is split into five equal folds, which means the number of queries among folds is equal. Four training folds are used for learning the ranking model, the remaining validation fold is used for tuning the hyper parameters of the ranking algorithm. We would then iterate over which fold is the validation fold, the average performance across the different folds is used to measure the overall accuracy of a learning-to-rank algorithm.

Besides the model accuracy, the performance of learning-to-rank algorithms is also evaluated by training time. For the model accuracy, we prefer employing the standard metric "Normalized Discounted Cumulative Gain" to evaluate, which will be explained below. For the training time, the results are compared with that of implementation on general computing platforms, such as CPU and GPU. Normally we run 300 epochs on the benchmark data set. 300 epochs is a typical length of training to obtain a ranking model with good accuracy [27].

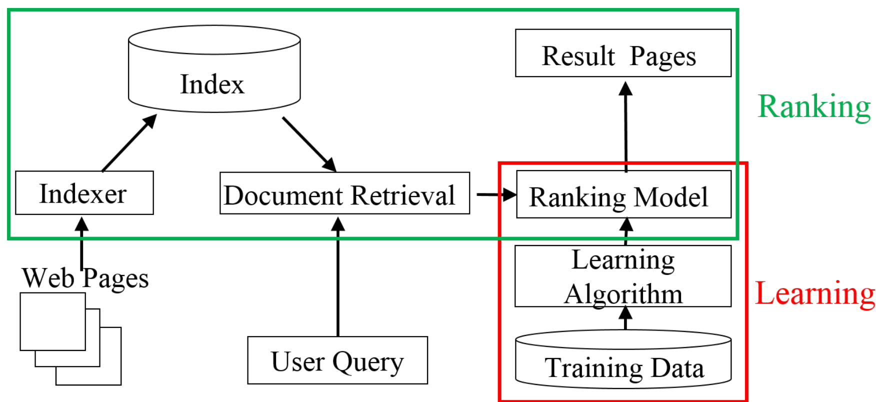


Figure 2.3: Machine-learning-based search engine overview

In order to obtain a deeper understanding of learning-to-rank algorithms, a categorization is performed on these algorithms. Based on the four components of machine learning, we categorise the algorithms and models for learning-to-rank into three groups: the pointwise, pairwise, and listwise approaches.

## 2.2 Major Ranking Approaches

As mentioned above, the major ranking approaches can be divided to three categories. In order to better understand learning to rank, we will explain the advantages and disadvantages of each category in the following subsections.

### 2.2.1 The Pointwise Approach

The pointwise approach assigns a score to each single document, and then the *loss function* examines the accurate prediction of the ground truth label for each document. The *input space* of pointwise approach is each single document, which is represented by a feature vector. The *output space* contains the accurate prediction for each document, which indicates the relevance degree to the query. The *hypothesis space* contains a function called scoring function $\boldsymbol{s}$ [1], which takes the feature vector as input and produces the accurate prediction for the document. Based on $\boldsymbol{s}$, we can sort all the documents and generate the predicted ranking list. When using the pointwise approach to do ranking, the *loss function* examines the accurate difference between

| Training Data Input: | $A$ $B$ $C$ $D$ $E$ |
|---|---|
| Regression | $s_A$<br>$s_B$<br>$s_C$<br>$s_D$<br>$s_E$ |
| Complexity | $O(n_d)$ |

Table 2.1: The process of pointwise approach. The technology of pointwise approach is regression

the predicted result and the ground truth label over each single document. So the computational complexity of training process is proportional to the number of associated documents, which is $n_d$.

Note that the pointwise approach is deployed for tasks of predicting real-valued quantities. In ranking, pointwise approach predicts the accurate relevance degree for each document, the position information of each document is not considered over the training process. Besides, the pointwise approach does not take into account that some documents are related to the same query. Because information retrieval is query level and position based, the pointwise approach has its limitations:

**(1)** In pointwise algorithms, the fact that the numbers of relevant documents for different queries are quite different is ignored. That means, when the number of relevant documents varies largely for different queries, the queries with a large number of documents will decide the final result of *loss function*.

**(2)** The pointwise approach calculates the accurate relevance degree of each document, so the position information of each document in the ranked list is not reflected. Therefore, the pointwise approach will be unconsciously dominated by those unimportant documents (the number of unimportant documents is much larger than that of important documents).

From above, we can see the pointwise approach is not the perfect approach to do ranking. In order to overcome the shortcomings of pointwise approach, people turn to take document pairs or a set of documents as the input rather than a single document. The approach which regards document pairs as input is called the pairwise approach, while the approach which takes the

whole set of documents as input is called the listwise approach. Using the pairwise approach, the relative relevance degree between documents can be visible. Using the listwise approach, the position of every document with the same query can be reflected.

### 2.2.2 The Pairwise Approach

The pairwise approach assigns a score to each relevant document using the current ranking model parameters, applies a pairwise differentiable *loss function* to the scores, and finally applies gradient descent to update the model parameters. The *input space* of the pairwise approach is pairs of documents, which are represented by feature vectors. The *output space* is the relative order between each pair of documents, which can be the value of -1 and +1. The *hypothesis space* takes a pair of documents as input and predicts the pairwise preference between them. In the pairwise approach, ranking is modeled as a pairwise classification, and the corresponding classification loss on a pair of documents is used as the *loss function*. When the scoring function $\boldsymbol{s}$ is used, the *loss function* measures the difference between $F(s_A, s_B)$ and the ground truth label $G(y_A, y_B)$.

Note that the *loss function* used in the pairwise approach only considers the relative order of document pairs. So the computational complexity of the training process is proportional to the number of document pairs, which is $n_d^2$. When only taking into account pairs of documents, however, it is difficult to derive the positions of the documents in the final permutation. Besides, pairwise approach does not leverage the fact that some pairs are related to the same query. Considering that ranking is query level and position based, we can predict a gap between this approach and ranking:

**(1)** When the relevance judgments are given in terms of multiple levels, however, it will lead to the loss of the finer granularity in relevant judgments as we assign the pairwise preference. For instance, there are two document pairs whose relevance degrees are different, the relevance judgments of one pair are *GOOD* and *BAD*, while the relevance judgments of the other pair are *GOOD* and *FAIR*, we can see these two pairs have different magnitudes of pairwise preferences.

| Training Data Input: | $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|---|
| Classification | $F(s_A, s_B)$ | | $F(s_A, s_C)$ | | $F(s_A, s_D)$ |
| | $F(s_A, s_E)$ | | $F(s_B, s_A)$ | | $F(s_B, s_C)$ |
| | $F(s_B, s_D)$ | | $F(s_B, s_E)$ | | $F(s_C, s_A)$ |
| | $F(s_C, s_B)$ | | $F(s_C, s_D)$ | | $F(s_C, s_E)$ |
| | $F(s_D, s_A)$ | | $F(s_D, s_B)$ | | $F(s_D, s_C)$ |
| | $F(s_D, s_E)$ | | $F(s_E, s_A)$ | | $F(s_E, s_B)$ |
| | $F(s_E, s_C)$ | | $F(s_E, s_D)$ | | |
| Complexity | | | $O(n_d^2)$ | | |

Table 2.2: The process of pairwise approach. The technology of pairwise approach is classification

However, in the *loss function* calculation of the pair approach, the ground truth labels of these two pairs are the same, which is not reasonable.

**(2)** For the pairwise approach, the number of pairs is in a quadratic order of the number of documents, that makes the imbalanced distribution across queries more serious than the pointwise approach.

**(3)** The position of documents in the final results is invisible from pairwise ranking algorithms, because the input of *loss functions* is document pairs. This means it only considers the relative order between two documents, the position of the documents in the final ranked list can hardly be derived. This approach is still not perfect for ranking.

From above analysis, we can see the pairwise approach has its advantages over the pointwise approach. It gives a relative order between two documents instead of the precise relevant value of single document. However, the problems mentioned above limits the effectiveness of the pairwise approach. To address the problems, new algorithms have been proposed.

### 2.2.3 The Listwise Approach

The listwise approach gives a score to each document and then these documents are sorted to produce permutations. Different from the pointwise and pairwise approaches, the *input space* of the listwise approach is the entire set of documents associated with a query, which are all represented by feature vectors. The *output space* of the listwise approach is a predicted

| Training Data Input: | $A$ $B$ | $C$ | $D$ $E$ |
|---|---|---|---|
| Classification | $P(s_A, s_B, ..., s_E)$ | $P(s_A, s_B, ..., s_D)$ | $P(s_A, s_B, ..., s_C)$ |
| | | ... | |
| | $P(s_B, s_A, ..., s_E)$ | $P(s_B, s_A, ..., s_D)$ | $P(s_B, s_A, ..., s_C)$ |
| | | ... | |
| | $P(s_C, s_A, ..., s_E)$ | $P(s_C, s_A, ..., s_D)$ | $P(s_C, s_A, ..., s_B)$ |
| | | ... | |
| | $P(s_D, s_A, ..., s_E)$ | $F(s_D, s_A, ..., s_C)$ | $P(s_D, s_A, ..., s_B)$ |
| | | ... | |
| | $P(s_E, s_A, ..., s_D)$ | $P(s_E, s_A, ..., s_C)$ | $P(s_E, s_A, ..., s_B)$ |
| | | ... | |
| Complexity | | $O(n_d!)$ | |

Table 2.3: The process of listwise approach. The output space of listwise approach is exactly the same as that of the task

permutation of the relevant documents, which is consistent with the task. The *hypothesis space* operates on the entire set of documents and outputs the predicted permutation, which is usually realised with the scoring function $\boldsymbol{s}$ in real implementations. $\boldsymbol{s}$ assigns a score to each document first, and then these documents are sorted in descending order to generate the ranked list based on the scores. The listwise *loss function* measures the inconsistency between the predicted permutations and the ground truth permutation.

Note that for the listwise approach, the *output space* that facilitates the learning process is exactly the same as the *output space* of the task. In this regard, the listwise approach models the raking problem in a more natural way than the other approaches where there are mismatches between the *output space* that facilitates learning and the real *output space* of the task. However, the computational complexity of training process of listwise approach is $O(n_d!)$, which is much higher than pointwise and pairwise approaches.

While the listwise approach improves the accuracy, the training complexity unfortunately scales quadraticaly with the number of documents, $n_d$. Comparatively speaking, the training complexity of pointwise and pairwise approaches is more acceptable. In particular, the complexity of listwise approach might be too high for real applications when $n_d$ is large. This situation would be changed by parallel computing, since there is a potential to execute the *loss function* of listwise approach in parallel, as will be shown later.

| Input Data: E, D, C, B, A | | | |
|---|---|---|---|
| Major Approaches | Pointwise | Pairwise | Listwise |
| | Regression | Classification | Ranking |
| | A → $S_A$ | A, B → $F(S_A,S_B)$ | A, B, C, D, E → $P(S_A,S_B,\ldots,S_E)$ |
| | B → $S_B$ | B, C → $F(S_B,S_C)$ | B, A, C, D, E → $P(S_B,S_A,\ldots,S_E)$ |
| | C → $S_C$ | ⋮ | ⋮ |
| | D → $S_D$ | C, D → $F(S_C,S_D)$ | D, A, B, C, E → $P(S_D,S_A,\ldots,S_E)$ |
| | E → $S_E$ | C, E → $F(S_C,S_E)$ | E, D, C, B, A → $P(S_E,S_D,\ldots,S_A)$ |
| | | D, E → $F(S_D,S_E)$ | |
| Complexity | $O(n_d)$ | $O(n_d^2)$ | $O(n_d!)$ |
| Ranking = A, B, C, D, E | | | |

Table 2.4: Summary of ranking approaches

### 2.2.4 Ranking Approach Overview

In the previous three subsections, we gave a comprehensive review on the pointwise, pairwise, and listwise approaches for learning to rank. We presented the basic framework, and discussed the advantages and disadvantages of each approach from the perspective of model accuracy and computational complexity. The key components for each approach are listed in Table 2.4. In the case of Figure 2.1, after calculating a score for each document, the pointwise approach compares the score with the ground truth label for each document (a, b, c...) directly; the pairwise approach measures the difference for relative order of each pair (ab, ac, bc...); the listwise approach evaluates the inconsistency with the ground truth permutation (c, a, b...). We can see that the major differences between these approaches lie in their *loss functions*, which guide the learning process. However, the evaluation of the learned ranking models is based on the evaluation measures, which will be discussed next.

## 2.3 Evaluation Measures

A standard evaluation mechanism plays an important role in selecting the most effective model. Because information retrieval is based on shareable document collections, queries, and relevance assessments, the corresponding evaluation process to proposed ranking models can be described as follows:

1) Collect queries submitted by surfers to form a test set.

2) For each query,

**(a)** Collect relevant documents of the query from the data set.

**(b)** Get the ground truth label for each document.

**(c)** Generate the desired permutation by sorting the relevant documents using a proposed model.

**(d)** Measure the difference between the permutation predicted by the proposed model and the permutation generated by the ground truth labels.

3) Calculate the average difference over all queries to evaluate the performance of the proposed model.

There are two most popular evaluation measures, one is called Mean Average Precision (MAP), the other is Normalized Discounted Cumulative Gain (NDCG). Usually the evaluation measures are defined query level and position based, as a function of the ordered documents given by the proposed model and the ground truth labels. The measured results are averaged over all the queries in the test set.

For MAP, the precision at position $k$ is defined as $(P@k)$. Suppose the binary judgment is used for the documents, 1 for relevant documents, while 0 for irrelevant documents. Then $P@k$ is defined by

$$P@k(\boldsymbol{y}, \boldsymbol{\pi}) = \frac{\sum_{j<=k} I_{\{y_{\pi(k)=1}\}}}{k} \tag{2.1}$$

Here $I_{\{\}}$ is the indicator function, and $\pi(k)$ means the document at position $k$ in the permutation $\pi$. Finally the Average Precision (AP) can be calculated as:

$$AP@k(\boldsymbol{y}, \boldsymbol{\pi}) = \frac{\sum_{j<=k}^{n_d} P@k \cdot I_{\{y_{\pi(k)=1}\}}}{n_d^1} \tag{2.2}$$

Here $n_d$ is the total number of documents relevant with query $q$, and $n_d^1$ is the total number of documents whose judgments are 1. The mean value of AP over all queries is mean average precision (MAP).

For NDCG, the precision at position $k$ is defined as (NDCG@$k$). This evaluation measure is special for multiple relevance judgments, and takes documents' explicit position into account. The DCG can be calculated by:

$$DCG@k(\boldsymbol{y}, \boldsymbol{\pi}) = \sum_{j=1}^{k} G(y_{\pi(j)})\eta(j) \tag{2.3}$$

Here $G()$ is the rating of a document, which usually sets $G(z) = (2^z - 1)$. $\eta(j)$ is a position discount factor, which is $\eta(j) = 1/log(j+1)$. By normalizing DCG@$k$ with its maximum possible value (denoted as $Z_k$), another measure called Normalized DCG (NDCG) is got as,

$$NDCG@k(\boldsymbol{y}, \boldsymbol{\pi}) = \frac{1}{Z_k} \sum_{j=1}^{k} G(y_{\pi(j)})\eta(j) \tag{2.4}$$

We can see that upper bound of NDCG is 1, while the bottom bound is 0. There are some similarities between these two evaluation measures:

1. Both MAP and NDCG are calculated at the query level. That means, we calculate the measure for each query first, and then average the results over all queries. Because the measure result is decided by the averaged value over all queries, and each query contributes similarly to the final result, a particular query will not dominate the evaluation process.

2. Both MAP and NDCG are position based. That is, rank position is explicitly considered in

computation. If and only if the score of one document passes another, the rank positions will change. And small changes in the scores will not influence the positions of documents. Note that both of the measures are usually position-based, that means they are discontinuous and non-differentiable with regards to the continuous scores.

MAP is assumed that there are two ground truth labels: relevant and irrelevant. So it is mainly used for the pointwise and pairwise approaches. While NDCG is mainly used to measure the ranking accuracy when there are more than two distinct ground truth levels. Using NDCG to compare the listwise with the other two approaches, the listwise approach delivers a better accuracy in general, because it emphasizes the concept of a ranked list and the position of the documents in the final ranking result are visible [1].

## 2.4 Review of Top-$k$ ListNet

ListNet, proposed by Cao [13], is the current state-of-the-art listwise approach receiving much attention. This method implements the ranking model as a fully-connected neural network (NN). For each query, the ranking model takes the feature vector of each relevant document as input and predicts the relevance degree for each document, the relevance degree is represented by the real-valued output; Then the divergence is collected between the permutation based on the outputs of ranking model and permutation based on ground truth; Finally a method called gradient descent is used to update the ranking model, so that the divergence loss diminishes.

Based on the fact that the calculation is intractable in practice, an alternative approach was proposed that only ranked the Top-$k$ elements in a listwise fashion. Given the scores of all documents, Top-$k$ ListNet clusters the permutations by the first $k$ documents.

The value of $k$ determines the computational complexity of *loss function* directly, as the metric between the corresponding Top-$k$ probability distributions is defined as the listwise *loss function*. For Top-$k$ ListNet, although the number of distinct probabilities reduces from $n_d!$ to $\frac{n_d!}{(n_d-k)!}$, the complexity is still a problem as it scales poorly as $k$ is increase, as it needs to calculate the probability of every permutation.

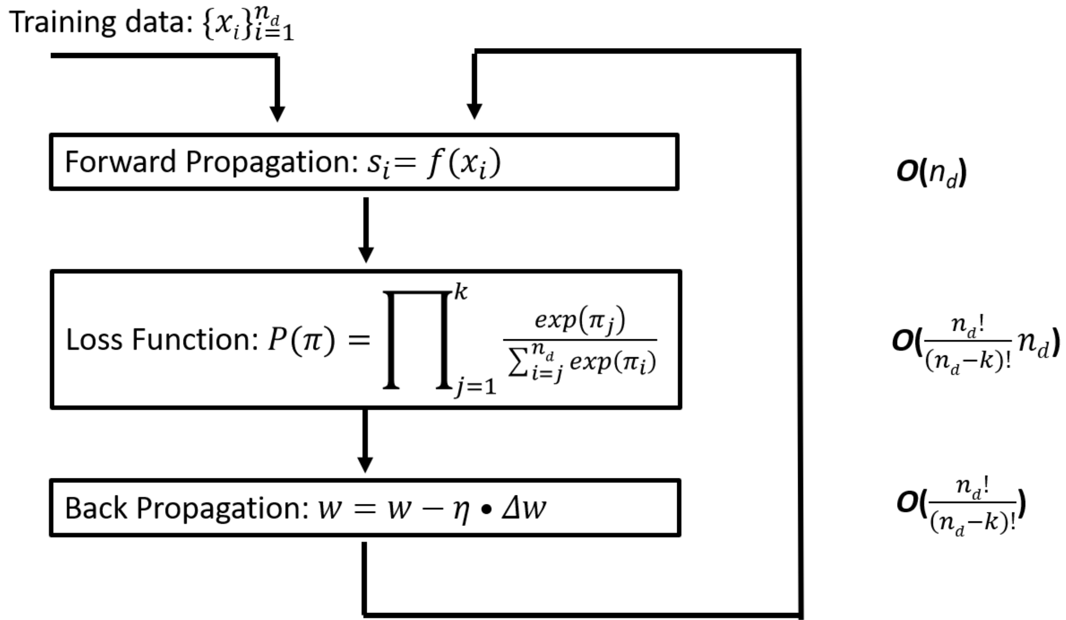


Figure 2.4: Computation process of Top-$k$ ListNet

In order to deal with the high computational complexity, $k = 1$ was selected in [13]. Choosing $k = 1$ reduces the amount of computation dramatically, but it largely restricts the capability of ListNet. In fact, setting $k = 1$ converts the probabilities distributed over document permutations to the probabilities over individual documents. As explained in Section 2.3, the evaluation measure NDCG contains a position discount factor. Therefore, it is quite important to learn the position information conveyed by the scores of other documents in the permutation. However, Top-1 ListNet places distributions over documents, and ignores the rank information of sequences, which will lead to lose the information about permutations. Thus the effectiveness of the ListNet algorithm cannot be guaranteed. We therefore seek to extend the Top-1 approximation to Top-$k$ ($k >= 2$) with an acceleration of computation.

We will introduce the Top-$k$ ListNet algorithm in detail below, which will be the focus of the acceleration work in this thesis. In the training process of Top-$k$ ListNet there are three steps:

**(1)** Forward Propagation;

**(2)** Loss Function;

**(3)** Back Propagation.

The first step is Forward Propagation (FP), as seen in Figure 2.4. In FP our ranking model uses a fully-connected 2-layer NN from [28], which has $n_f$ inputs, $n_h$ hidden units, and one output unit. The model parameters include the weights $\boldsymbol{w}^{21}$ between the input layer and the hidden layer, the weights $\boldsymbol{w}^{32}$ between hidden layer and output layer. For each query, the FP calculates the score $s_i$, $i = 1, ...n_d$ of each relevant document:

$$s_i = \text{Sigmoid}(\sum_{h=1}^{n_h} w_h \,\text{Sigmoid}(\sum_{f=1}^{n_f} w_{hf}^{21} x_{qif})) \tag{2.5}$$

Sigmoid function is historically deployed as the activation function, which takes a single number and squashes it into range between 0 and 1. Crucially, negative numbers saturate at 0 and positive numbers saturate at 1. Rectified Linear Unit (ReLU) is another option which drew wide attention recently as its linear computation, and it is cheap to implement on hardware. In order to keep consistent with previous ranking algorithms, we decide to choose Sigmoid function as the activation function.

The second step is the Loss Function calculation, as seen in Figure 2.4. The set of all possible permutations of $n_d$ documents for each query is denoted as $\boldsymbol{G}_{n_d}$, the number of permutations in $\boldsymbol{G}_{n_d}$ is $n_d!$. A permutation $\boldsymbol{\pi}$ of $\boldsymbol{G}_{n_d}$ on the $n_d$ documents is a bijection from $\{1, 2, ..., n_d\}$ to itself, which is written as $\boldsymbol{\pi} =< \pi(1), \pi(2), ...\pi(n_d) >$, where $\pi(j)$ denotes the document assigned to position $j$. Top-$k$ ListNet elaborates the permutations with $k$ documents, so the number of permutations decreases to $\frac{n_d!}{(n_d-k)!}$; we call the set of Top-$k$ permutations $\boldsymbol{G}_{n_d}^k$. The probability of permutation $\boldsymbol{\pi}$ is defined using the Top-$k$ *Plackett-Luce* model [29]:

$$P(\boldsymbol{s}, \boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{e^{s_{\pi(j)}}}{\sum_{i=j}^{n_d} e^{s_{\pi(i)}}} \tag{2.6}$$

where $s_{\pi(j)}$ is the score of the document at position $j$ ($j = 1, 2, ..., k$) in permutation $\boldsymbol{\pi}$. In Equation (2.6), exponentiating the scores produces the probabilities, and the division performs the normalization so that the summation of all permutation probabilities equals to 1. Based on the probability Equation (2.6), given a list of scores, we can calculate the probability of each possible permutation. In order to make the Loss Function differentiable cross entropy is

taken as the metric between two probability distributions over the document permutations, one derived from the ground truth permutation and the other derived from the model prediction, which means the Loss Function is calculated as:

$$L(\boldsymbol{y}, \boldsymbol{s}) = -\sum_{\forall \boldsymbol{\pi} \in \boldsymbol{G}_{\boldsymbol{n_d}}^{\boldsymbol{k}}} P(\boldsymbol{y}, \boldsymbol{\pi}) \log(P(\boldsymbol{s}, \boldsymbol{\pi})) \tag{2.7}$$

The final step is Back Propagation (BP), as seen in Figure 2.4. We get the gradient of $L$ with respect to the parameters $\boldsymbol{w}^{21}$ and $\boldsymbol{w}^{32}$ as:

$$\Delta w = \frac{\partial L(\boldsymbol{y}, \boldsymbol{s})}{\partial w} = -\sum_{\forall \boldsymbol{\pi} \in \boldsymbol{G}_{\boldsymbol{n_d}}^{\boldsymbol{k}}} \frac{\partial P(\boldsymbol{s}, \boldsymbol{\pi})}{\partial w} \frac{P(\boldsymbol{y}, \boldsymbol{\pi})}{P(\boldsymbol{s}, \boldsymbol{\pi})} \tag{2.8}$$

Then we update the parameters of the ranking model over epochs using gradient descent under learning rate $\eta$ [30]:

$$w^{'} = w - \eta \Delta w \tag{2.9}$$

Note that learning rate $\eta$ is the most important hyperparameter setting in the training process of NN algorithms. A small $\eta$ makes very tiny progress and is not efficient. Conversely, a large $\eta$ may not pay off. In practice, the learning rate is commonly decayed over time to improve the performance of NN training.

As mentioned earlier, $n_d$ is above 1000 in general, so the computation of Loss Function and BP might be intractable with a large $k$ due to the large number of permutations. For example, when $k = 2$, the scale of computation is order of $10^9$, and when $k = 3$, the scale becomes order of $10^{12}$, thus the training time of Top-$k$ ListNet is rather slow when $k > 1$. For Top-$k$ ($k = 2$) ListNet, it normally takes about 24 hours or more to get acceptable results. For Top-$k$ ($k = 3$) ListNet, the calculation becomes intractable in practice. However, as the calculation is dominated by a summation over many permutations there is a potential for parallelism if we are able to restructure the algorithm correctly.

**Algorithm 1** Top-$k$ ListNet

1: Input: number of epochs $n_{epoch}$, number of hidden nodes $n_h$, document features $\boldsymbol{x}$, learning rate $\eta$
2: Output: $\boldsymbol{w}^{21}, \boldsymbol{w}^{32}$
3: Initialization: $\boldsymbol{w}^{21} = random(n_h, n_f)$, $\boldsymbol{w}^{32} = random(n_h)$
4: **for** $e = 1$ to $n_{epoch}$ **do**
5:     **for** $q = 1$ to $n_q$ **do**
6:         ▷ Forward Propagation
7:         Initialization: $\boldsymbol{s}^{32} = \boldsymbol{0}$, $\boldsymbol{s} = \boldsymbol{0}$
8:         **for** $i = 1$ to $n_d$ **do**
9:             **for** $h = 1$ to $n_h$ **do**
10:                 **for** $f = 1$ to $n_f$ **do**
11:                   $\boldsymbol{s}_{ih}^{32} = \boldsymbol{s}_{ih}^{32} + \boldsymbol{w}_{hf}^{21}\boldsymbol{x}_{qif}$
12:                 **end for**
13:                 $\boldsymbol{s}_{ih}^{32} = \text{Sigmoid}(\boldsymbol{s}_{ih}^{32})$
14:                 $\boldsymbol{s}_i = \boldsymbol{s}_i + \boldsymbol{w}_h^{32}\boldsymbol{s}_{ih}^{32}$
15:             **end for**
16:             $\boldsymbol{s}_i = \text{Sigmoid}(\boldsymbol{s}_i)$
17:         **end for**
18:         **while** $\boldsymbol{\pi} \in \boldsymbol{G}_{n_d}^k$ **do**
19:             ▷ Loss Function
20:             $P(\boldsymbol{y}, \boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{\exp(\boldsymbol{y}_{\pi(j)})}{\sum_{i=j}^{n_d} \exp(\boldsymbol{y}_{\pi(i)})}$
21:             $P(\boldsymbol{s}, \boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{\exp(\boldsymbol{s}_{\pi(j)})}{\sum_{i=j}^{n_d} \exp(\boldsymbol{s}_{\pi(i)})}$
22:             ▷ Back Propagation and Update Parameters
23:             **for** $h = 1$ to $n_h$ **do**
24:                 $\Delta\boldsymbol{w}_h^{32} = -\frac{\partial P(\boldsymbol{s},\boldsymbol{\pi})}{\partial \boldsymbol{w}_h^{32}} \frac{P(\boldsymbol{y},\boldsymbol{\pi})}{P(\boldsymbol{s},\boldsymbol{\pi})}$
25:                 $\boldsymbol{w}_h^{32} = \boldsymbol{w}_h^{32} - \eta\Delta\boldsymbol{w}_h^{32}$
26:                 **for** $f = 1$ to $n_f$ **do**
27:                   $\Delta\boldsymbol{w}_{hf}^{21} = -\frac{\partial P(\boldsymbol{s},\boldsymbol{\pi})}{\partial \boldsymbol{w}_{hf}^{21}} \frac{P(\boldsymbol{y},\boldsymbol{\pi})}{P(\boldsymbol{s},\boldsymbol{\pi})}$
28:                   $\boldsymbol{w}_{hf}^{21} = \boldsymbol{w}_{hf}^{21} - \eta\Delta\boldsymbol{w}_{hf}^{21}$
29:                 **end for**
30:             **end for**
31:         **end while**
32:     **end for**
33: **end for**

In this research, we use a single benchmark data set – LETOR 4.0. This is because LETOR 4.0 is the only training set where the ground truth is labeled as a permutation for a query. As the Loss Function of listwise approach is based the permutation probabilities, LETOR 4.0 allows to apply listwise algorithms directly. In addition, our target is developing techniques to accelerate the learning process rather than improving the model accuracy. One data set is enough to verify the effectiveness of proposed techniques, so our experiments are developed on

LETOR 4.0.

## 2.5 Related Work

From the analysis above, we can see the calculation of Top-$k$ ListNet algorithm is quite different from that of the pointwise and pairwise approaches. While considering just singles or pairs has much lower cost, it is difficult to derive the positions of documents in the final ranked list. Ultimately, ranking is about learning a complete ordering on documents, so we can expect a large accuracy gap between traditional approaches and the listwise approach. Top-$k$ ListNet uses permutations of documents as instances in learning, and the positions of documents in the final ranking result are visible. This makes the listwise approach more accordance with the ranking task than other two approaches. Note that there are in total $\frac{n_d!}{(n_d-k)!}$ elements in the permutation class for each query. In addition, the computational complexity of permutation probability for Top-$k$ ListNet is $O(n_d)$, which is much more complex than regressing one singe document into a concrete value in the pointwise approach or classifying document pairs into two categories in the pairwise approach. In order to address the high computational complexity of the Top-$k$ ListNet algorithm, we investigate optimisations on both software (reducing computational complexity) and hardware (accelerating computation speed).

### 2.5.1 Algorithm Optimisation

Several pieces of work have been proposed to overcome the computational complexity of ListNet by reducing the number of permutations in training process. Because every possible permutation has to be considered in conventional ListNet approach, the computation is intractable in practice.

ListMLE was devised to replace the objective of ListNet with biased sampling method [10, 31]. For each query $q$, it sorts the relevant documents based on the outputs of the ranking model to generate the predicted permutation, and then fully trains on the permutation. Since one

**Algorithm 2** Stochastic Top-$k$ ListNet scheduling

1: Input: number of epochs $n_{epoch}$, number of queries $n_q$, document features $\boldsymbol{x}$, learning rate $\eta$
2: Output: $\boldsymbol{w}^{21}, \boldsymbol{w}^{32}$
3: Initialization: $\boldsymbol{w}^{21} = random(n_h, n_f)$, $\boldsymbol{w}^{32} = random(n_h)$
4: **for** $e = 1$ to $n_{epoch}$ **do**
5:     **for** $q = 1$ to $n_q$ **do**
6:         ▷ Forward Propagation
7:         Initialization: $\boldsymbol{s}^{32} = \boldsymbol{0}$, $\boldsymbol{s} = \boldsymbol{0}$, $\Delta \boldsymbol{w} = \boldsymbol{0}$
8:         $\boldsymbol{s}^{32} = \text{Sigmoid}(\boldsymbol{w}^{21}\boldsymbol{x})$
9:         $\boldsymbol{s} = \text{Sigmoid}(\boldsymbol{w}^{32}\boldsymbol{s}^{32})$
10:         Sample from the permutation classes $\boldsymbol{G^k_{n_d}}\left(\frac{n_d!}{(n_d-k)!}\right)$ to $\boldsymbol{S^k_{n_d}}\left(m\frac{n_d!}{(n_d-k)!}\right)$
11:         **while** $\boldsymbol{\pi} \in \boldsymbol{S}^k_{n_d}$ **do**
12:             ▷ Loss Function
13:             $L_\pi = -P(\boldsymbol{y}, \boldsymbol{\pi}) \log(P(\boldsymbol{s}, \boldsymbol{\pi}))$
14:             ▷ Back Propagation
15:             $d\boldsymbol{w} = \frac{\partial L_\pi}{\partial \boldsymbol{w}} = \frac{\partial \boldsymbol{s}}{\partial \boldsymbol{w}}\frac{\partial L_\pi}{\partial \boldsymbol{s}}$
16:             $\Delta \boldsymbol{w} += d\boldsymbol{w}$
17:         **end while**
18:         ▷ Update Parameters
19:         $\boldsymbol{w} = \boldsymbol{w} - \eta \Delta \boldsymbol{w}$
20:     **end for**
21: **end for**

only needs to compute the probability of one single permutation in ListMLE, the training complexity has been greatly reduced as compared to conventional ListNet approach. In addition, a stochastic Top-$k$ ListNet variant has been proposed to address the complexity of Top-$k$ ListNet [32, 33]. It takes permutations as instances and selects a small number of permutations for training by uniform distribution sampling, fixed distribution sampling and adaptive distribution sampling. The computation process is illustrated in Algorithm 2, $m$ in line 10 represents complexity reduced after sampling, which is in the interval of [0, 1].

Both ListMLE and stochastic Top-$k$ ListNet take permutations as instances when they do sampling. The difference is the number of sampled permutations varies. ListMLE only pays attention to one single permutation but then fully trains on that permutation, while stochastic Top-$k$ ListNet selects more permutations but only trains the top $k$ positions within each permutation. Both methods select samples from the full set of $\frac{n_d!}{(n_d-k)!}$ permutation classes. However, it is difficult to select high-weighted samples since there are many more low-weighted samples. In addition, although both approaches can reduce the complexity successfully, neither approach

recognises that some positions are much more important than others. That means the existing sampling methods cannot guarantee the model accuracy. We therefore seek to develop new sampling methods to reflect the different importance of different positions.

Overall, both stochastic Top-$k$ ListNet and ListMLE provide feasible ways to reduce the computation complexity, but biased towards low-weighted samples. The key distinction of our work compared to previous efforts is that we introduce a new biased sampling approach for Top-$k$ ListNet which has a higher probability to select high-weighted samples and exploits the different importance between positions. We will describe the details in Chapter 4.

### 2.5.2 Hardware Acceleration

There have been some pieces of work which accelerated ranking algorithms such as Top-1 ListNet, LambdaRank using different hardware platforms [34, 35, 27], but there was no work about accelerating Top-$k$ ListNet ($k >= 2$).

Top-1 ListNet was accelerated by using Spark [34], which is a cluster computing system that supports reuse of data across parallel operations. By applying Top-1 ListNet to the benchmark data set LETOR 4.0 using Spark, the results show the training time reduces linearly as the number of parallel CPUs increases.

A parallel implementation of an on-demand ranking algorithm using GPUs was presented [35]. The association rule technique was used to derive a rule-based learning model, and the set of rules can be computed in parallel for each document using GPUs. A mean speedup of 127x in query processing time is reported compared to the sequential version. The speedup achieved at learning the model is not reported.

Several works have been proposed to accelerate pairwise algorithms using FPGAs. A deeply pipeline method was proposed to accelerate LambdaRank using an FPGA device [27], that achieved a 15.3x speedup (with PCIe4) over a software implementation. LambdaRank is a NN algorithm extended from RankNet models. As the Loss Function of RankNet is pair-based, LambdaRank optimizes the lost function indirectly by defining a smooth approximation to its

gradient to reflect the intent of ranking. However, no coarse-grained parallelism makes the training process of LambdaRank rather slow. [27] therefore presents an efficient FPGA-based accelerator for LambdaRank.

An FPGA-based accelerator was designed for RankBoost [36]. To build efficient accelerators, they optimised the RankBoost algorithm to reduce the computation complexity and to make it suitable for parallel implementation. The accelerator achieved a 27.63x speedup.

While the research above indicates promising approaches for accelerating Top-1 ListNet and Lambdarank, it is difficult to effectively accelerate Top-$k$ ListNet ($k >= 2$) using CPU clusters or GPUs for the following reasons.

- The training process of ListNet updates NN parameters after computing each query, so no query-level parallelism can be utilised by a CPU cluster.
- The number of cores in a CPU cluster is limited, so fine-grained weight-level parallelisms cannot be fully utilised.
- The power consumption for a CPU cluster or GPUs often limits how far the application can be cost effectively scaled [37].
- In Top-$k$ ListNet ($k >= 2$), the complex and multivariate BP cannot be efficiently computed by GPUs.

Using FPGAs to accelerate Top-$k$ ListNet has the potential to address the issues faced by CPUs and GPUs above, because the fine-grained and pipeline parallelism can be used by FPGA devices. LambdaRank was accelerated by deeply pipelining the problem and exploiting the fine-grained parallelism [27], and achieved a speed of 11.7 GFLOP/s. However, to the best of authors' knowledge, there is no existing work that accelerates Top-$k$ ListNet using FPGAs. One of the difficulties is the computation dependence. We will discuss how to optimise computation process and accelerate Top-$k$ ListNet using FPGAs in this thesis.

The work presented in this thesis also uses application-specific optimisation of arithmetic operators, in order to reduce resource utilisation with minimal reduction in accuracy. The trade-off

between floating-point and fixed-point in FPGAs is well-known, with floating-point providing a larger range and requiring less analysis, while fixed-point operators require less resource and energy per operation.

Low-precision multiplications were proposed for training deep neural networks [38]. By attempting floating-point, fixed-point and mixed precision fixed-point for the training of the Maxout network, they found 9 (10 with the sign) bits in mixed precision fixed-point is sufficient for training. 16-bit fixed-point representations have also been used for training, with the addition of stochastic rather than conventional rounding, resulting in no degradation in the classification accuracy [39]. Besides, FPGA-based reduced precision implementation for convolution networks also have been presented [40, 41, 42, 43], and a peak performance of 61.62 GFLOP/s is achieved under 100 MHz clock frequency in [43]. The fixed-point format used in previous works consists in a signed mantissa and a scaling factor, the scaling factor is a variant for different variables and layers, and it can be updated over epochs. For each scalar variable in each layer, the scaling factor is normally fixed during each epoch. However, it is difficult to faithfully represent the entire range of gradient values of the ListNet algorithm using the fixed-point format with a fixed scaling factor, because the gradient range is quite wide. The gradients are based on permutation probabilities, and the probabilities of different permutations are quite different. If we represent the gradient value using the traditional fixed-point format, the model accuracy will decrease seriously. To the best of our knowledge, none of the above works considered representing a single scalar variable in the same layer (as the gradient) using the fixed-point format with a dynamic scaling factor.

Finally, we also investigate the reconfigurable capability of FPGAs, in order to minimise latency with minimal reduction in accuracy. FPGA devices have the reprogram ability due to the unique composition, which are composed of the configuration memory layer and the hardware logic layer [44, 45, 46, 47]. The hardware logic layer mainly contains the computational resource, and the configuration memory layer stores the bitstream file which determines how to connect the hardware components in the hardware logic layer. Modern configuration memory layer is SRAM based, which is hence volatile. To change the function implemented in the FPGAs, a user can modify the contents of the configuration memory by loading a new bitstream. This

operation is called FPGA reconfiguration.

The technique of reconfiguration has been applied to deal with different issues in the literature, which can be classified as partial reconfiguration, full reconfiguration. Partial reconfiguration, where only part of the hardware layer is modified at runtime while the remaining part is not altered, is mainly used to reduce reconfiguration time, minimise area consumption. Full reconfiguration, where the whole hardware layer is modified during running, can also improve the resource utilisation. In addition, the terms partial reconfiguration and dynamic reconfiguration are used interchangeably, but they can be different. Since the partical reconfiguration operation can occur while the FPGA logic is in a reset state (static) or running (dynamic), partical reconfiguration can be static or dynamic. However, it does not mean all dynamic reconfigurations are partial in nature. For example, in context switching FPGAs, the full configuration is changed during reconfiguration, but the operation is dynamic [47].

Partial reconfiguration implementation of the support vector machine (SVM) classifier were provided in [48, 49, 50], where individual classifiers are selectively adapted to avoid multiplexed classifiers. Concurrent implementation of multiple classifiers achieved about 50x speedup compared with the implementation of general purpose processor. In addition, partial reconfiguration can also help to save areas by implementing tasks using the same region [51, 52, 53, 54]. Compared with partial reconfiguration, the reconfiguration time of full device reconfiguration is a bit larger. However, the magnitude of reconfiguration time is at the level of milliseconds, which is negligible compared to the time consumption of computation [55]. In [56, 57], the original synchronous dataflow graph is split into several subgraphs along its depth. Each subgraph is mapped to a distinct hardware architecture, The execution of each subgraph is realised by full reconfiguration of the FPGAs. In this thesis, we also utilise full reconfiguration to change the hardware functions, but we arrive at a method in which the reconfiguration happens over epochs.

To sum up, FPGAs are more promising to accelerate Top-$k$ ListNet, and the fixed-point computation on FPGA devices can significantly reduce the training time of NN algorithms, but with a fixed scaling factor is difficult to represent wide range variables. The key distinction of our

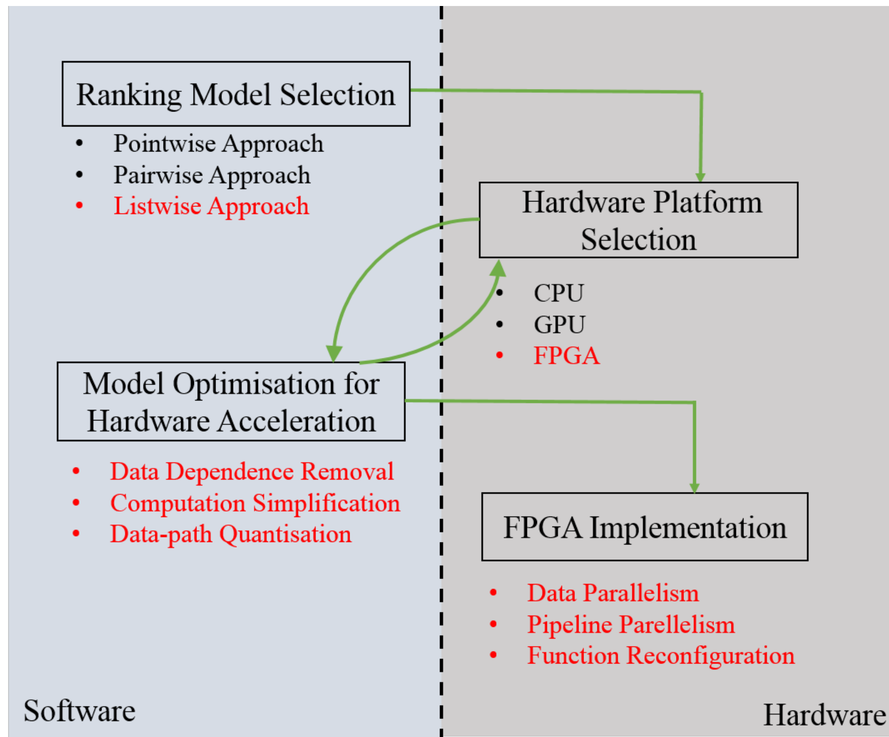


Figure 2.5: FPGA-based Accelerator Design Flow for Listwise Approach

work compared to previous efforts is that we present an effective fixed-point quantisation approach based on FPGA devices which represents the gradient values under different precision. Furthermore, we develop a training method with multiple kernels using the reconfiguration capability of FPGAs. We will explain the details in next several chapters.

## 2.6 Design Flow of Hardware-Based Accelerator

Based on the analysis of different ranking approaches and computing platforms, the design logic of hardware acceleration is summarised. As displayed in Figure 2.5, the appropriate ranking approach (software) which performs the highest accuracy is selected first. Then the highly flexible platform (hardware) – FPGA device is deployed to operate the computation, and the resource requirement is predicted. After that, the computation process of ranking algorithm is reorganised to fully utilise the advantages of FPGAs. In particular, the software and hardware designs are iteratively developed to achieve the best balance between model accuracy and resource consumption. Finally, a practical implementation which leverages the

unique capabilities of FPGAs should be delivered to fulfill expectations.

## 2.7 Summary

In this chapter, we introduced the background of this thesis. Specifically, we first introduced learning to rank for information retrieval by taking web search as an example. Second, the three different approaches for ranking were presented in details, the advantages and shortcomings of each approach were analysed, which demonstrate that the listwise approach is more promising. Third, the widely used evaluation measures for ranking were described. Then the typical listwise approach Top-$k$ ListNet was reviewed. After that, we summarized the existing work which is devoted to reduce the computational complexity of ranking algorithms, and also the work which focused on accelerating the training of raking algorithms on different platforms. Finally, the design flow of acceleration was displayed.

# Chapter 3

# Accelerating Top-$k$ ListNet Training for Ranking Using FPGA

## 3.1 Introduction

Document ranking is used to order query results by relevance, with different document ranking models providing trade-offs between ranking accuracy and training speed. ListNet is a well-known ranking approach which achieves high accuracy, but is infeasible in practice because the training time is quadratic in the number of training documents. This chapter considers accelerating ListNet training using FPGAs, and improves training speed by using hardware-oriented algorithmic optimisations, and by transforming algorithmic structures to expose parallelism.

As mentioned in Section 2.5.2, the nature of Top-$k$ ListNet makes it hard to be accelerated with commodity computing platforms. First, like LambdaRank, the training process of ListNet updates NN weights after computing each query, so there is no query-level parallelism that could be utilized by a cluster of computers. Second, the node-level parallelisms cannot be used efficiently by CPU clusters due to the limited number of CPU cores. Furthermore, the derivative computation of loss function is difficult to be efficiently computed by modern general-purpose computing on a Graphics Processing Unit (GPU), which requires relatively regular computations. Thus, it is natural to turn to a customized circuit implementation. Considering

reprogrammability, FPGAs offers a distinct advantage over ASICs [58, 59, 60].

FPGA implementation of pairwise approach has been studied. However, there is no existing work to accelerate Top-$k$ ListNet because the implementation has several new challenges. First, the loss function is on the basis of probability models for permutations, which is more complex than computation over single document or document pairs. Second, the acceleration rate should be scalable to the increasing size of the training data set, the training complexity of Top-$k$ ListNet is in the exponential order of the document number.

In this chapter we present an analysis and design for realising ListNet on FPGA devices. By exploiting the fine grain parallelism of the FPGA device, we demonstrate that it is possible to accelerate ListNet by 3.21x and show that we can increase the accuracy of the ranking function from 0.29 to 0.33 compared to LambdaRank, while still maintaining the same execution time of a pairwise approach on an Intel Xeon CPU. The main contributions of this chapter are:

- Redesigning the computation method of Top-$k$ ListNet that exposes more fine-grained parallelism.
- The first full implementation of listwise algorithm on FPGAs, which presents how to efficiently map a listwise algorithm onto an FPGA device.
- Results showing how our FPGA implementation of Top-$k$ ListNet can achieve a ranking accuracy of 0.33 using NDCG@10 and 3.21x speedup over an Intel Xeon CPU implementation on the MQ2008 data set of LETOR 4.0, which consists of 471 queries and 540679 documents.

The remainder of this chapter is organised as follows: Section 3.2 discusses the challenge of Top-$k$ ListNet, and presents optimisation of the computation. Section 3.3 explains the design of Top-$k$ ListNet algorithm for an FPGA device. Section 3.4 presents the experiments and analyses experimental results. Section 3.5 summarizes the chapter.

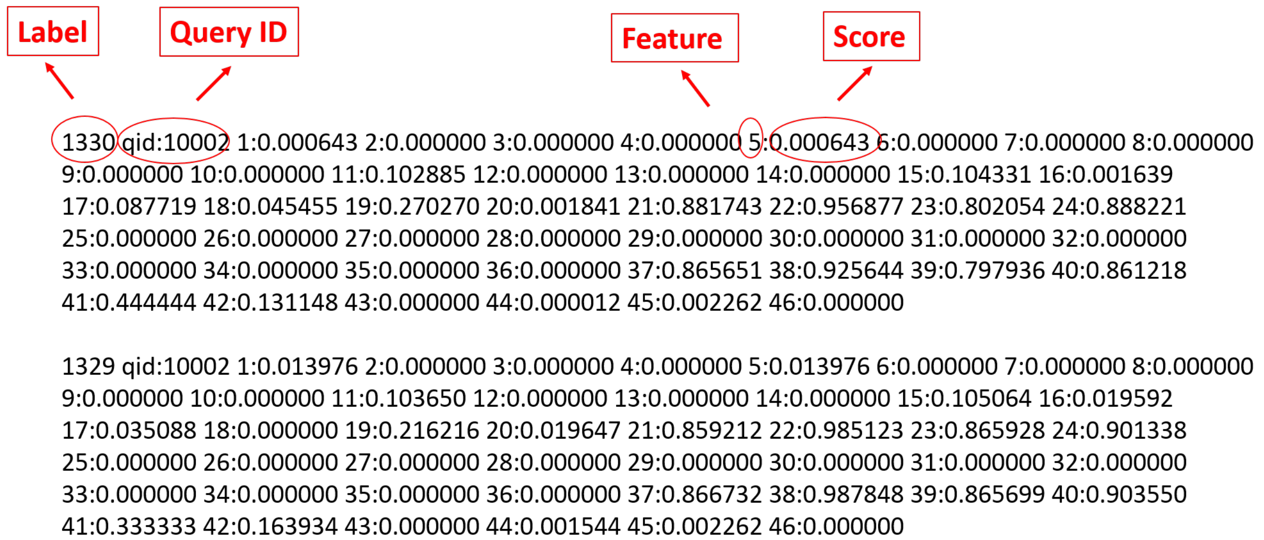


Figure 3.1: An example of document representation in benchmark data set LETOR 4.0

## 3.2 Analysis of Top-$k$ ListNet

In this section, we will discuss our optimisation of the Top-$k$ ListNet algorithm. A new computation method is proposed which reduces redundant computations in each permutation and also allows to compute the permutations in parallel.

### 3.2.1 Challenge of Top-$k$ ListNet

From Section 2.4, we can see the high computational complexity may constrain the power of Top-$k$ ListNet in real application. In this section we investigate the computational bottleneck of Top-$k$ ListNet where $k >= 2$ by adapting the open source "RankLib package" [61] to implement ListNet. We applied the ListNet implementation to the benchmark data set LETOR 4.0, which consists of 471 queries and 540679 documents. LETOR is a benchmark data set for ranking released by Microsoft. The 4.0 version is a new release. In version 4.0, the ground truth for each document is the position in the final list instead of multiple level relevance judgements, as shown in Figure 3.1. This version is more suitable for listwise approach training. For the older versions, ground truth label is given as 0 or 1, which is mainly for pointwise approach (regression technology) and pairwise approach (classification technology).

As seen in Figure 3.2, we find that for each epoch the FP calculation takes less than 0.1% of the total time, while Loss Function calculation and BP calculation are the most time consuming. As $n_d$ (the number of documents for each query) increases, the Loss Function takes a larger percentage of the time. This conclusion also holds for larger $k$ ($k > 2$).

The computational bottleneck of Top-$k$ ListNet ($k >= 2$) is the Loss Function and BP, and as $n_d$ grows the bottleneck becomes larger. This is because the computation of FP is over individual documents, while the computation of Loss Function and BP is over permutations. As we saw in Equation 3.1, the *Luce model* is a stagewise model, which decomposes the probability computation of a permutation with $k$ documents into $k$ stages (line 20 and line 21 in Algorithm 1). It computes the permutation probability in this way: first, at the $i$-th stage, a document is selected and assigned to position $i$ according to a probability $\frac{e^{\pi_i}}{\sum_{m=i}^{n} e^{\pi_m}}$, the probability is based on the scores of unassigned documents $\sum_{m=i}^{n} e^{\pi_m}$; second, the product of the selection probability at all $k$ stages defines the probability of the permutation; and finally we calculate the probability of next permutation by changing the document order of last permutation, that means the permutation probability is calculated sequentially and there is dependence between different permutations. Because the number of unassigned documents is $(n_d - i)$ for the $i$-th stage, and there are in total $\frac{n_d!}{(n_d-k)!}$ permutations, the complexity of computation of Top-$k$ ListNet is caused by two factors, one is the number of documents in each permutation $n_d$; the other is the number of permutations $\frac{n_d!}{(n_d-k)!}$. The computational complexity of Top-$k$ ListNet is $O\left(\frac{n_d!}{(n_d-k)!} n_d\right)$.

### 3.2.2 Redesign of Top-$k$ ListNet Computation

When using the *Luce model* (Equation 2.7) to compute the probability $P(s, \pi)$ we must evaluate a fraction, where the numerator is the score of the assigned document, and the denominator is the summation of unassigned document scores. If we expand the *Plackett-Luce* model, we can see there are repeated computations for the probability of different stages:

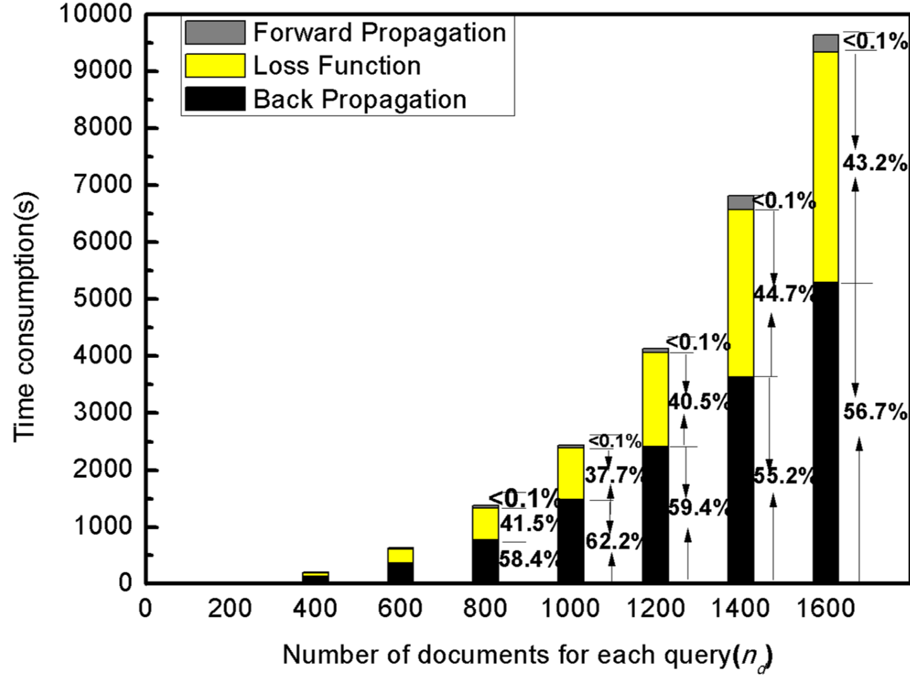


Figure 3.2: Time consumption on our Intel Xeon CPU for different tasks in Top-2 ListNet using the source code of RankLib

$$P(\boldsymbol{s},\boldsymbol{\pi}) = \frac{e^{s_{\pi(1)}}}{e^{s_{\pi(1)}} + e^{s_{\pi(2)}} + ... + \boxed{e^{s_{\pi(k)}} + ... + e^{s_{\pi(n_d)}}}} \times \frac{e^{s_{\pi(2)}}}{e^{s_{\pi(2)}} + ... + \boxed{e^{s_{\pi(k)}} + ... + e^{s_{\pi(n_d)}}}} ... \times \frac{e^{s_{\pi(k)}}}{\boxed{e^{s_{\pi(k)}} + ... + e^{s_{\pi(n_d)}}}} \tag{3.1}$$

In order to reduce the computational complexity, precomputation method is used to reduce the redundant computation. This method computes the summation of all document scores initially:

$$\lambda = \sum_{i=1}^{n_d} e^{s_{\pi(i)}} \tag{3.2}$$

where $\lambda$ is the sum of all document scores. Then through algebraic manipulation we can replace $O(n_d)$ additions with $O(k)$ subtractions:

$$P'(\boldsymbol{s},\boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{e^{s_{\pi(j)}}}{\lambda - \sum_{i=j}^{k-1} e^{s_{\pi(i)}}} \tag{3.3}$$

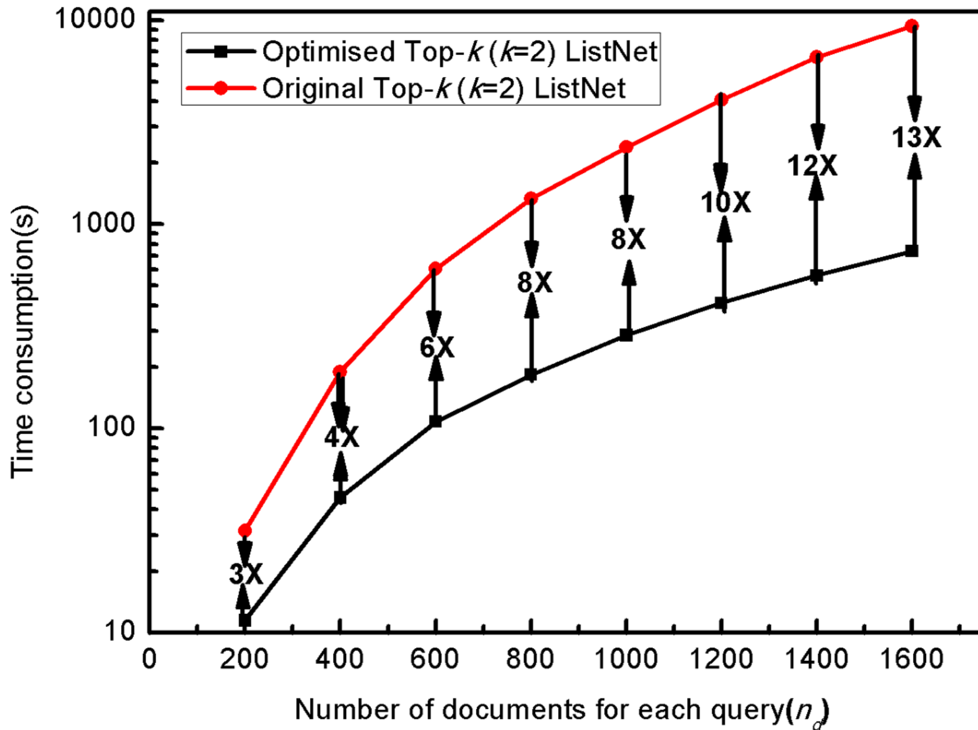


Figure 3.3: Time consumption on our Intel Xeon CPU for different probability computation methods

As $k << n_d$, the main contribution of the new method is to reduce the computational complexity of the *Luce model* calculation, which decreases from $O(kn_d)$ to $O(k^2)$ for each permutation. In order to verify the hypothesis, we implement Top-$k$ ($k = 2$) ListNet on our Intel Xeon CPU with the data set LETOR 4.0 mentioned in Section 3.2.1. The result is shown as Figure 3.3. The time decreased, especially when $n_d$ is large.

The precomputation method does not sacrifice the accuracy of the model, even though it simplifies the calculation. When employing gradient calculation for each permutation, we take derivatives of each variable with respect to the parameters. For the original version, all $n_d$ documents need to be considered in each permutation, including the assigned $k$ documents and the unassigned $(n_d - k)$ documents. While the new computation only considers the scores of assigned $k$ documents, the number of variables decreases from $n_d$ to $k$. Based on riffled independence [62, 63, 64, 65], our method, which only takes derivatives of the Top-$k$ variables does not sacrifice the accuracy of final model.

### 3.2.3 Benefit of Precomputation for Top-$k$ ListNet

The proposed optimisation reduces the complexity of the Loss Function, especially when $k$ is small. However, the method has other benefits as it enables permutations to be computed in parallel. For the original *Luce model* (2.7), we need to calculate both the assigned documents and unassigned documents when computing the probability of each stage, which means it is difficult to compute the probability of different permutations in parallel. The permutation probability is normally computed sequentially as shown in Algorithm 3.

**Algorithm 3** *Luce model* scheduling

```
1: while π=next_permutation(π) do
2:     Loss Function
3:     Back Propagation and Update Parameters
4: end while
```

After our optimisation, the variables for each stage only contain the assigned documents, so the number of variables for each stage decreases from $n_d$ to $k$, and it becomes easy to compute the probability of different permutations in parallel, which is showed in Algorithm 4. The parallelism makes it possible to accelerate the algorithm using various techniques, such as an FPGA-based accelerator, as described in the next section.

**Algorithm 4** Optimised model scheduling

```
1: for t_k = 1 to n_d do
2:     ...
3:     for t_2 = 1 to n_d do
4:         for t_1 = 1 to n_d do
5:             π=generate_top(t_1, t_2, ..., t_k)
6:             Loss Function Calculation
7:             Back Propagation Calculation
8:         end for
9:     end for
10:    ...
11: end for
```

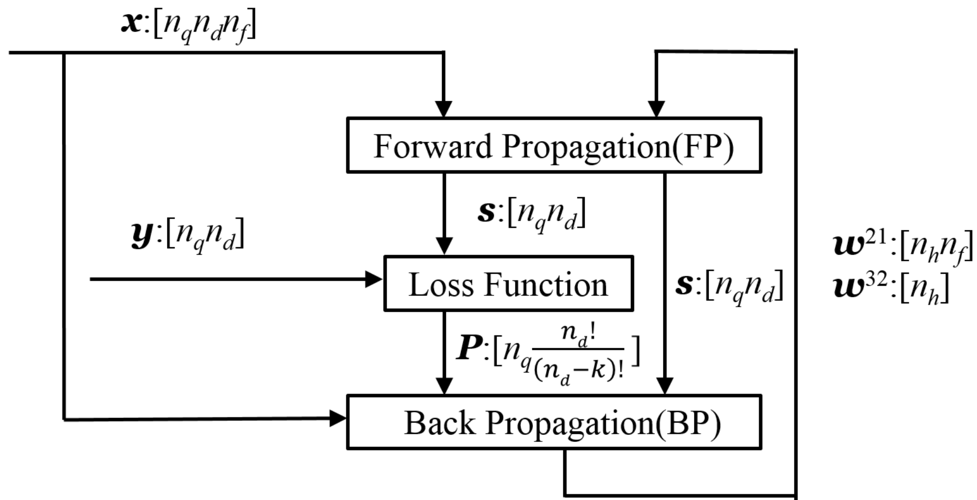


Figure 3.4: Communication data among different tasks in computation per training epoch

## 3.3 Hardware Mapping

Our main goal is to accelerate Top-k ListNet using an FPGA device, so we will now discuss how best to map the algorithm to hardware, starting with hardware and software partitioning, then explores how the data and instruction parallelism were exploited.

### 3.3.1 HW/SW Partitioning and Communication

As we saw in Section 2.4, Top-$k$ ListNet is split into three tasks: FP, Loss Function, and BP. These three tasks can be executed concurrently, but they also have dependencies which requires communication overheads which are summarised in Figure 3.4. In this chapter we do not consider number representations, and assume that all data is represented as single precision floating point numbers.

As in Figure 3.4, the FP task consumes the training data and NN parameters and sends document scores to the Loss Function task and the BP task. Then the Loss Function task takes the ground truth labels and document scores as inputs, and sends permutation probabilities to the BP task. Finally the BP task processes the results of the other two tasks and returns updated NN parameters back to the FP task. Based on the experiment results of software in Section 3.1, there are several different choices of how to map the application to an FPGA

device.

For one-task hardware implementation,

- [**FP-only**] only maps the FP task onto the FPGA device, the Loss Function and BP tasks remain in software.
- [**LF-only**] only maps the Loss Function task onto the FPGA device, the FP and BP tasks remain in software.
- [**BP-only**] only maps the BP task onto the FPGA device, the FP and Loss Function tasks remain in software.

For two-task hardware implementation,

- [**FP+LF**] maps the FP and Loss Function tasks onto the FPGA device, only the BP task remains in software.
- [**FP+BP**] maps the FP and BP tasks onto the FPGA device, only the Loss Function task remains in software.
- [**LF+BP**] maps the Loss Function and BP tasks onto the FPGA device, only the FP task remains in software.

For all-task hardware implementation,

- [**FP+LF+BP**] maps all three tasks onto the FPGA device.

According to the running process of Top-$k$ ListNet, all the implementations share the same high level time model:

$$t_{total} = t_{ini} + n_{epoch} t_{epoch} \tag{3.4}$$

where $t_{ini}$ is the program initialization time, $n_{epoch}$ is the total number of epochs, and $t_{epoch}$ is the executing time of each training epoch. The values of $t_{ini}$ and $t_{epoch}$ are dependent on the implementation. As $t_{ini}$ is relatively small, we only evaluate $t_{epoch}$ which includes the computation time and communication time:

$$t_{epoch} = t_{FP} + t_{LF} + t_{BP} + t_{comm1} + t_{comm2} \tag{3.5}$$

where $t_{FP}$ is computation time for FP task, $t_{LF}$ is computation time for Loss Function task, and $t_{BP}$ is computation time for BP task. The computation time relies on the implementation platform. $t_{comm1}$ refers to the communication time of transferring data from software (DDR) to hardware (BRAM), while $t_{comm2}$ denotes the time consumption of transferring data from hardware to software. Under certain transfer speed, the communication time is decided by the amount of transferred data. The system-level view of the computing architecture is shown as Figure 3.5, which consists of an ARM Cortex processor, a DMA controller, an AXI bus interface and the IP core (which represents tasks mapped on the FPGA). The ARM processor is responsible to initialise the parameters and execute the tasks remained in software. The function of DMA controller is for accessing the DDR memory by both the ARM processor and the IP core.

We evaluate the three methods by the time consumption and hardware resource required. Without parallelism, the resource required is decided by the amount of total work on board.

For the one-task method, although it consumes the least hardware resource, the time consumption is unacceptable. Since only one task is executed on accelerator board, the one-task choice requires more computation time than other two methods. For communication time, take BP-only as an example, besides the features of training documents, which is $4n_q n_d n_f$ bytes, the calculation of BP requires the document scores from the FP task and the Loss Function task, which are the inputs of BP. The BP task requires as inputs: training data, which is $4n_q n_d n_f$ bytes; along with the outputs from the FP task, which is $4n_q n_d$ bytes of data from the host to the FPGA; and the outputs from the Loss Function task, which is $4n_q \frac{n_d!}{(n_d-k)!}$ bytes of data

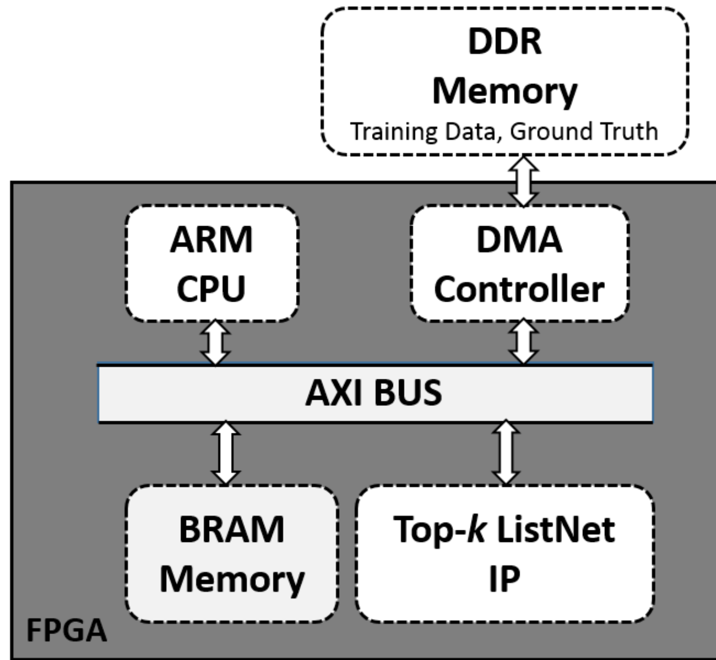


Figure 3.5: The architecture of FPGA system integrated with Top-$k$ ListNet IP

from the host to the FPGA device. Table 3.1 summarises the size of communication data for one-task implementation over one epoch using MQ2008 data set.

| Data Size | FP-only | LF-only | BP-only |
|---|---|---|---|
| Training Data ($4n_qn_dn_f$) | 97MB | 0MB | 97MB |
| Ground Truth ($4n_qn_d$) | 0MB | 2MB | 0MB |
| FB Results ($4n_qn_d$) | 2MB | 2MB | 2MB |
| Loss Results | 0MB | 2366MB | 2366MB |
| Total | 99MB | 2370MB | 2465MB |

Table 3.1: Size of communication data between software and hardware of one-task hardware implementation for Top-2 ListNet

For the two-task method, while it consumes more hardware resource to accelerate computation process compared to one-task method, the time consumption is less than one-task method. For the implementation of LF+BP, besides the features of training documents, the evaluation of the Loss Function and BP depends on the scores from FP calculation. So each time the FP routine transfers $4n_qn_d$ bytes data to FPGA, and DDR also transfers $4n_qn_d$ bytes data (ground truth labels) to FPGA. Table 3.2 summarises the size of communication data for two-task hardware implementation over one epoch using MQ2008 data set.

The all-task method requires the most hardware resource to implement the whole computation,

| Data Size | FP+LF | FP+BP | LF+BP |
|---|---|---|---|
| Training Data ($4n_qn_dn_f$) | 97MB | 97MB | 97MB |
| Ground Truth ($4n_qn_d$) | 2MB | 0MB | 2MB |
| FB Results ($4n_qn_d$) | 2MB | 2MB | 2MB |
| Loss Results | 2366MB | 2366MB | 0MB |
| Total | 2467MB | 2465MB | 101MB |

Table 3.2: Size of communication data between software and hardware of two-task hardware implementation for Top-2 ListNet

and the communication time is smallest. In each epoch of the training process the software sends features of training documents to the hardware and waits for the hardware to return. We can see the three tasks consume the features of training data from the DDR, then update parameters of the NN model back to to the FP task. The NN parameters and intermediate results can be kept on chip, but only the training date (97MB) and ground truth data (2MB) needs to be fetched from the DDR.

From above analysis, we can see there is a trade-off between hardware resource and time consumption. In order to achieve the largest possible speedup in our application, the all-task implementation (FP+LF+BP method) is chosen, which maps all the computation tasks to FPGA. With this partition, the software will be responsible for initialising the parameters of the NN model (the beginning of the Algorithm 1) and preparing the training data into arrays of floats. Then the software sends the training data and the ground truth labels to the hardware, and the computation starts in the hardware. The concrete data flow is as show in Figure 3.6.

We should keep in mind that the three tasks (FP, Loss Function, and BP) cannot run concurrently because of the data dependencies between them, but can be pipelined. When $k$ is increased, the computation time of Loss Function and BP increases exponentially, however, the communication time between software and hardware remains constant.

### 3.3.2 Data Parallelism

Top-$k$ ListNet is a NN-based algorithm, and NNs are known to have inherent parallelism which can be exploited to speed up computation [37]. Figure 3.7 shows the structure of the ListNet

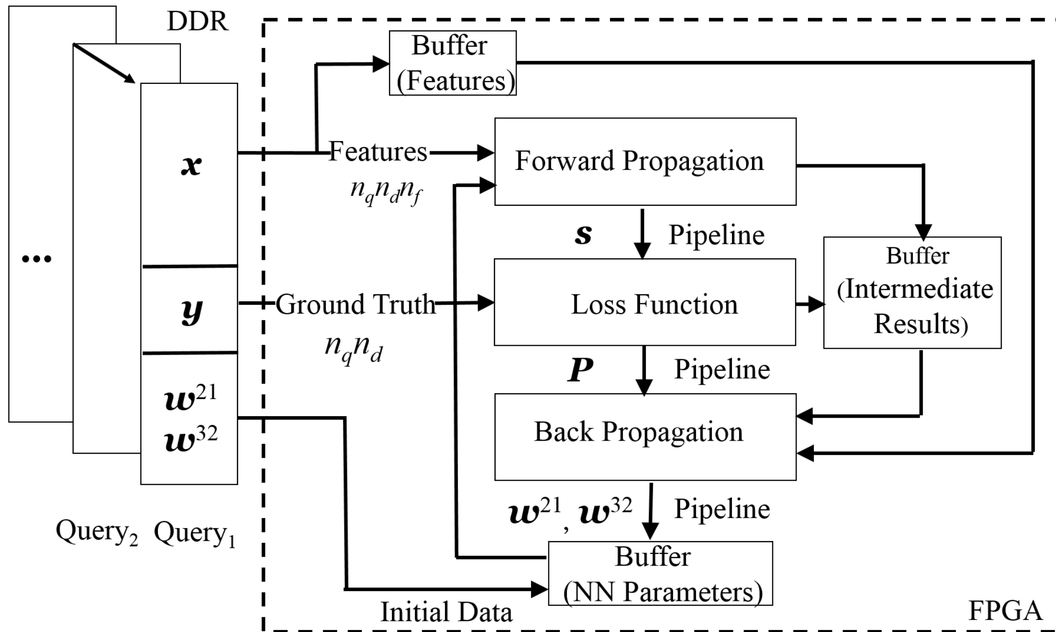


Figure 3.6: Data flow between different tasks on FPGA

NN, where it can be seen that each node of the same layer executes the same computation independently, but there is a data dependence between different layers. Because the input data of the NN is document features, which are independent from each other, the data parallelism can be utilised by an FPGA device to reduce design latency. Design latency is the number of cycle the computation system takes to output the result.

To accelerate execution within the same layer we used loop unrolling to instantiate multiple parallel copies of the loop body [43]. The hidden node loop (line 9 and line 24 in Algorithm 1) is unrolled to utilise the node-level parallelism, while the feature loop (line 10 and line 27 in Algorithm 1) is unrolled to utilise the feature-level data parallelism. The organisation of Computation Unit for hidden layer (hCU) can be found in Figure 3.8. The computation for each hidden node is implemented in the hCU.

### 3.3.3 Fine-grain Pipeline Parallelism

Pipeline parallelism is exploited through loop pipelining, which hides the loop latency by overlapping the execution of operations from different loop epochs. Pipelining is utilised to improve the design throughput, which is the number of cycles between new inputs. For the FP task,

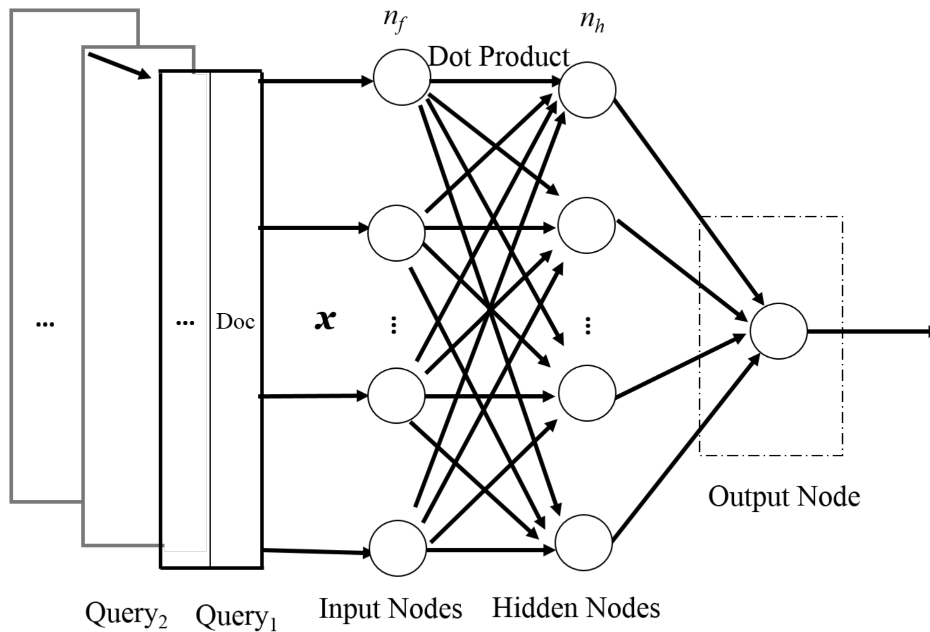


Figure 3.7: Structure of 2-layer neural network ranking model

the pipeline directive can be applied between different layers. The document loop (line 8 in Algorithm 1) was pipelined to fully utilise the hCUs in Figure 3.7. The Loss Function and the BP task are both based on the same permutations, so we performed loop-fusion to merge them together. As shown in Algorithm 3, our precomputation method generates these permutations using $k$ layers of nested loops, which can be pipelined (line 4 in Algorithm 3). If the loop is pipelined with an initiation interval ($II$), a new read operation can occur each $II$ cycles. We manage to achieve $II$ equals to 1, by removing the computational dependence between different permutations as discussed in Section 3.2.2.

## 3.4 Experiments and Analysis

We implemented our proposed enhanced version of ListNet using an FPGA device to demonstrate the potential acceleration and explore the accuracy of our approach. The target platform was a Xilinx ZCU102 development board and SDSoC was used to generate the RTL. SDSoC is a high-level synthesis (HLS) tool released by Xilinx, which converts C++ code to HDL, similar to Vivado HLS. The unrolling and pipelining techniques are controlled by applying pragma

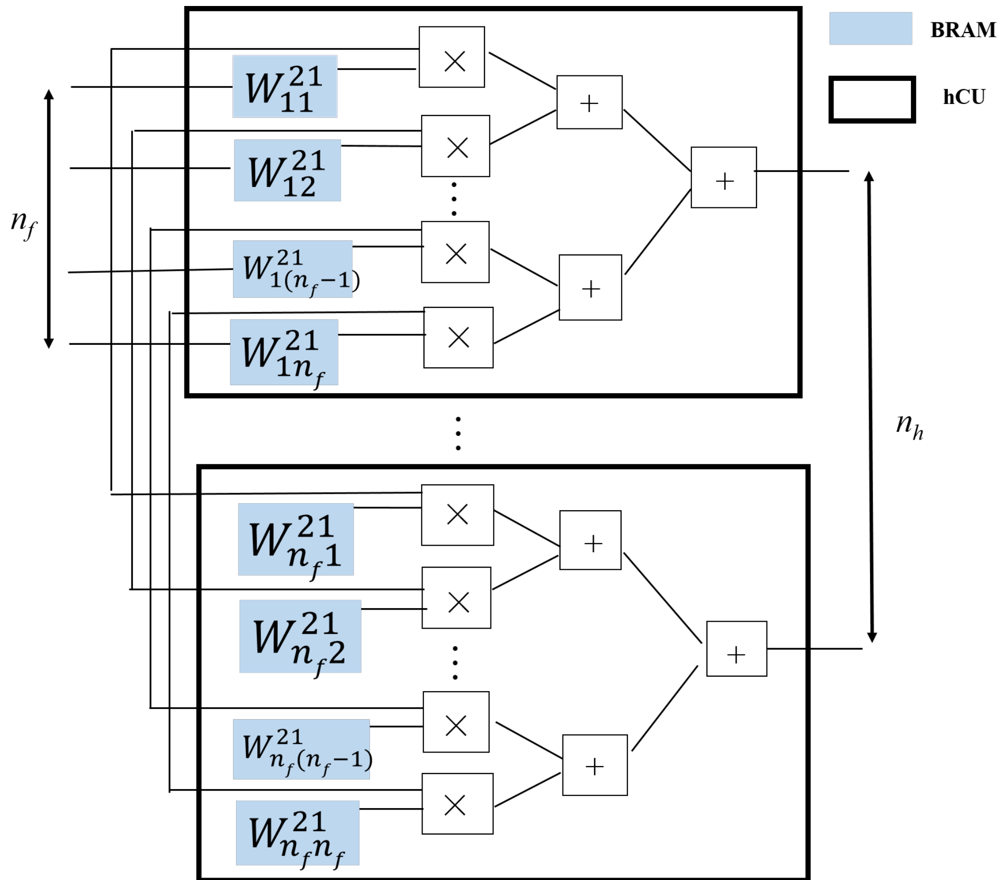


Figure 3.8: The organisation of Computation Unit for hidden layer (hCU)

directives in for loops. We compared the accuracy of Top-$k$ ($k = 2$) ListNet with that of Lambdarank and Top-1 ListNet, and explored the scalability of Top-$k$ ($k = 2$) ListNet by varying the document number. The data set we used is the standard learning-to-rank benchmark data set LETOR 4.0, which has 471 queries and 540679 documents, where each document is represented by 46 features. In order to explore the optimal hidden node number, we implemented the algorithm with varying numbers of hidden nodes from 3 to 40. Comparing the training results under different hidden nodes, the implementation with 15 hidden nodes suggests the best result.

### 3.4.1 Analysis of the Speedup

Table 3.3 shows the hardware utilisation for our design. We were able to achieve a maximum frequency of 300MHz resulting in a speed of 13.6 GFLOP/s and a computational efficiency of 0.2 FLOP/s/DSP.

| Resource | LUTs | Regs | DSP | BRAM |
|---|---|---|---|---|
| Used | 79 690 | 111 846 | 217 | 353 |
| Available | 274 080 | 548 160 | 2520 | 912 |
| Percent | 29% | 20% | 9% | 39% |

Table 3.3: Utilization of hardware resource for Top-2 ListNet on Xilinx ZCU102 development board

The software implementation takes 395.53s to perform one training epoch over the LETOR 4.0 data set, running on a single CPU on an Intel Xeon 1.6 GHz processor for Top-$k$ ($k = 2$) ListNet. In comparison, our FPGA implementation takes 122.87s resulting in a 3.21x speedup over the software for Top-$k$ ($k = 2$) ListNet. There are two main factors affect speedup: (i) the FPGA device can exploit the data and instruction parallelism of the application, this is the main contributor to speedup (ii) the clock frequency also influences the time consumption, although it is much lower compared with that of the CPU. Table 3.4 shows the time consumption of the three tasks for each epoch. The FPGA implementation is able to achieve a 4.65x speedup for the Loss Function task, and 3.16x times speedup for the BP task, which is the most time consuming task. Although it does not result in a speedup for the FP task, it is still useful to map the task onto the FPGA device as it saves communication time. Overall, the system achieves a 3.21x speedup.

| Task | SW Time(s) | HW Time(s) | Speedup |
|---|---|---|---|
| FP | 0.57(0.5%) | 1.60(1.0%) | 0.36 |
| Loss | 34.20(8.5%) | 7.36(6.0%) | 4.65 |
| BP | 360.76(91.0%) | 113.91(93.0%) | 3.16 |
| Total | 395.53(100.0%) | 122.87(100.0%) | 3.21 |

Table 3.4: Time consumption of different tasks on CPU VS FPGA in one epoch

In order to quantify the contribution to speedup from data parallelism and pipeline, we modified the FPGA implementation to apply either loop unrolling or pipelining, with the results shown in Table 3.5. The data parallelism brings 62.30x speedup, while the pipelining achieves 6.06x speedup. The data parallelism has more effects than pipelining. The contribution of parallelism is to utilise more hardware resource, it does not have much impact on computation efficiency. The contribution of pipelining is to start the next execution as soon as the hardware resource becomes available, so it achieves speedup by improving computation efficiency.

| Optimisation | Time(s) | Speedup | Computational Efficiency |
|---|---|---|---|
| Original | 38 500.08 | 1.00 | 0.004 |
| Pipeline | 6354.99 | 6.06 | 0.020 |
| Parallelism | 618.91 | 62.30 | 0.007 |
| Parallelism+Pipeline | 122.87 | 313.34 | 0.200 |

Table 3.5: The contribution of data parallelism with unrolling factor $U = 150$ (under fanout limitation) and pipeline with initiation interval $II = 11$ to speedup. Metric of computational efficiency is FLOP/s/DSP.

### 3.4.2 Ranking Accuracy

To evaluate the quality of the generated model, we trained different learning to rank algorithms with 300 epochs. Then we compare the NDCG of Top-$k$ ListNet with that of RankNet and Top-1 ListNet. Using the NDCG evaluation measure, Figure 3.9 gives the accuracy for the selected learning-to-rank algorithms. Our experiments confirm that the listwise algorithm delivers better accuracy than pairwise algorithm. The experiment results show that the ranking accuracy of Top-$k$ ListNet is better than Top-1 ListNet, implying that to improve accuracy it is beneficial to extend from Top-1 model to Top-$k$ model.

### 3.4.3 Scalability of Speedup

The execution time of ListNet scales poorly (exponential) with the number of documents, $n_d$. However, in order to achieve high accuracy we must train models over large numbers of documents. To investigate scalability we conducted experiments where we increased the number of documents for each query. As for the all-task implementation (FP+LF+BP method), only the training date and ground truth are transferred. Since the size of training date is $n_q n_d n_f$, and ground truth is $n_q n_d$, the communication time ($T_{comm1}$ and $T_{comm2}$) scales linearly with $n_d$. For the computation time, it scales exponentially with $n_d$ as the number of permutations is $\frac{n_d!}{(n_d-k)!}$. Compared with the computation time, the communication time can be neglected. Assuming that a new input read can only be performed every $L$ cycles, and the computation is pipelined with an initiation interval of $II$, it will take $L + (N/F - 1)II$ cycles to complete all

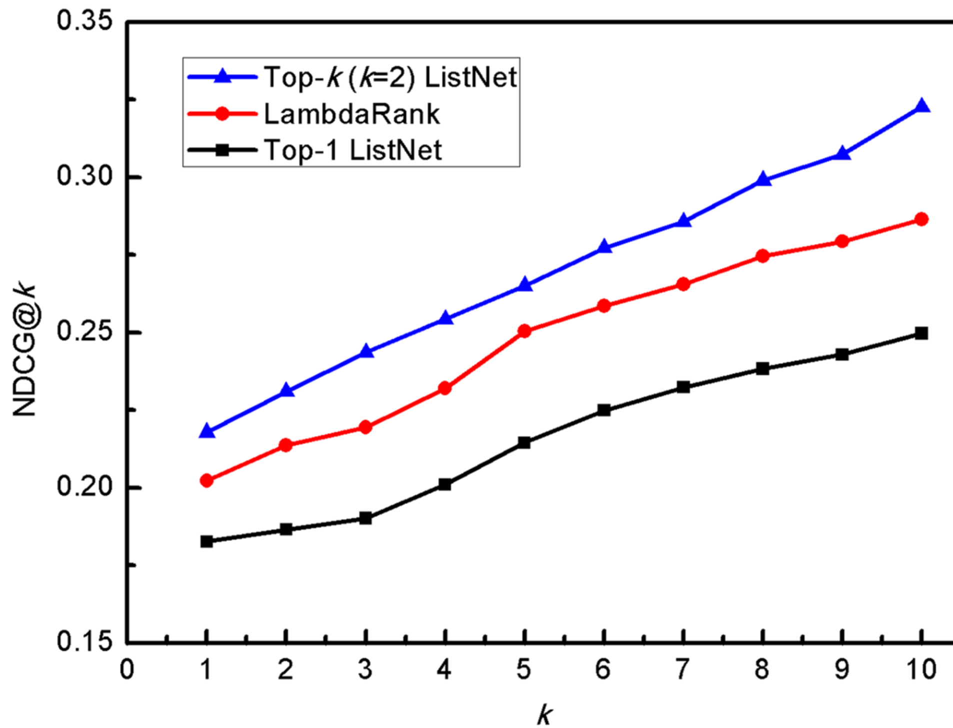


Figure 3.9: Accuracy comparison among different algorithms, NDCG@$k$ means it calculated at the first $k$ positions.

the computation. Here, $F$ is the unrolling factor, and $N$ is the amount of computation which equals to the number of permutations in Top-$k$ ListNet.

The ideal situation is the case where there are no communication overheads, the value of $F$ varies linearly with $n_d$, the speedup increases linearly with $n_d$. However, the results, displayed in Figure 3.10, show that as the number of documents for each query is increased the accelerator scales well in reality, but the speedup is not as high as the ideal scaling curve. We believe the explanation for this is because the communication time overheads increases dramatically in real execution, it can not take full advantage of the powerful computation capability of FPGA.

## 3.5 Expectation

The proposed techniques in this chapter are feasible for Top-$k$ ListNet, no matter what the value of $k$ is. The experiment results showed are only for $k = 2$, this is because the computational complexity is in exponential order of $k$, and it is intractable for large $k$ in practice. For

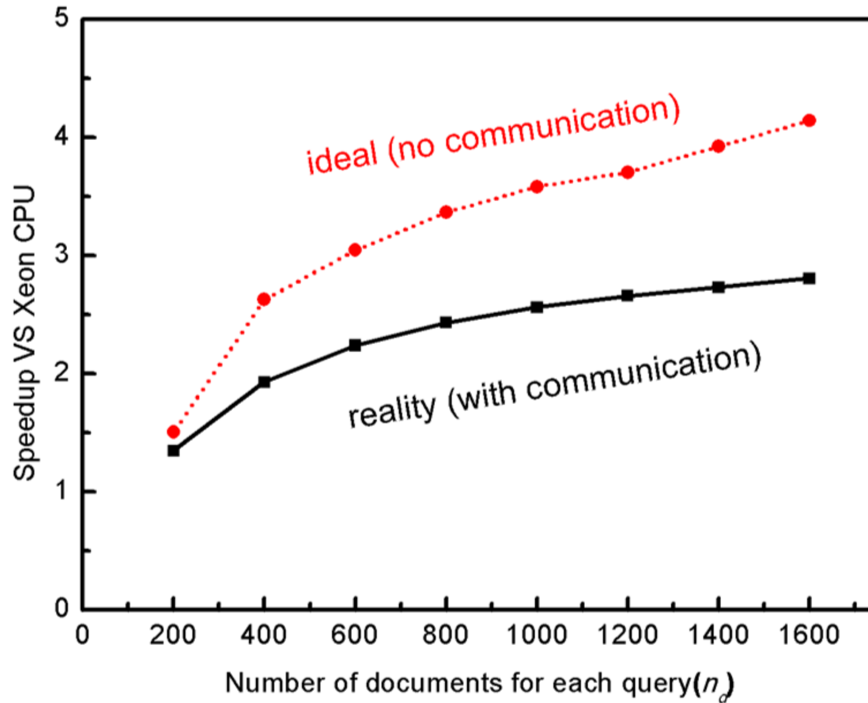


Figure 3.10: The complete execution speedup of Top-$k$ (k=2) ListNet with different $n_d$

example, it takes 5 days to run one epoch for Top-$k$ ($k$ = 3) ListNet. In the case of large $k$, the only difference is that there are more layers in the nested loop in Algorithm 4. Applying the pipelining technique in computation, the nested loop will be flattened, and the throughput will be improved. The $k$ is larger, the contribution of pipelining will be more significant, the speedup will be more apparent. So the precomputation methodology can be deployed for any Top-$k$ ListNet.

The proposed techniques are also applicable for different data sets, although in this chapter only one benchmark data set LETOR 4.0 is investigated. The number of queries, relevant documents (possible permutations) and features may vary for different data sets. As the resource is limited on the board, it is difficult to compute different permutations in parallel. For the permutation layer, pipeline directive is applied. For the layer of feature loop, parallel directive is applied, so the number of feature affects the degree of parallelism. Based on the discussion, we can see the techniques proposed in this chapter will benefit training speed even when the data set varies. In previous data sets, the ground truth labels are given in terms of relevant or irrelevant, which is more suitable for binary classification rather than ranking. That is the reason we paid full attention to LETOR data set. Since it is infeasible to access every possible permutation in the

case of large $k$, we will discuss how to compute with sampling methods later.

## 3.6 Summary

This chapter explores the potential of accelerating ListNet with FPGA devices, so that the calculation of Top-$k$ ListNet is practical for real world applications. We first redesign the computation process of Top-$k$ ListNet to reduce the computation complexity and to make it feasible for parallel execution on an FPGA device. Then we explaine our design and design decisions required to achieve the largest acceleration. Top-$k$ ListNet shows a better ranking accuracy and the acceleration can bring 3.21x speedup to Top-$k$ ($k = 2$) ListNet, which takes the communication between hardware and software into account, showing that FPGA devices can be utilised to accelerate listwise algorithms. Based on the achievements of this work, we explore the potential to accelerate optimised Top-$k$ ListNet with a large $k$ using FPGA devices.

# Chapter 4

# Accelerating Position-Aware Top-$k$ ListNet under Custom Precision Regimes

## 4.1 Introduction

ListNet has been intensively investigated in constructing and training learning to rank models. Compared with traditional learning approaches, such as RankNet, ListNet delivers better accuracy. However, ListNet is computationally too expensive to learn models with large data sets due to the large number of permutations involved in computing the gradients of the loss function. Previous solutions propose to compute with sampling approaches, which take permutations as instances and select a subset of the permutation classes. That means they collect same number of samples for each position in the permutation, which do not take the importance of positions into account; therefore they cannot guarantee accuracy for top positions, which is critical in many applications. Moreover, significant gains in speed and computational efficiency could be realised by training in fixed point formats for ListNet. In order to convert to fixed point faithfully, the variable should have a sufficiently narrow dynamic range so that the entire range of values can be represented by mantissa bits alone. However, the gradient

range of the loss function is quite wide because it is based on permutation probabilities, and the probabilities of different permutations are quite different in magnitude.

In this chapter, a new position-aware sampling approach is introduced, which significantly reduces the computation complexity without sacrificing accuracy of top position. In addition, an effective quantisation method based on FPGA devices for the ListNet algorithm is proposed, which organises the gradient values to several batches, and associates each batch with a different fractional factor. We make Top-$k$ ranking tractable in two ways: we use a weighted sampling technique, that accounts for the position of the documents, to reduce the total number of documents needed to be considered for a given accuracy; and a hardware-accelerated FPGA implementation that uses custom precision arithmetic optimisations and dynamic management that when coupled with our novel sampling technique achieves a significant improvement in training speed and computation efficiency. The major contributions of this chapter are:

- A new sampling approach, which takes the importance of positions into account and shows better accuracy than existing methods.
- A new fixed point quantisation approach for Top-$k$ ListNet, which splits the gradient values of loss function into several batches, and associates different batches with different fractional precision. This method allows to represent wide range variables with limited bits, which can effectively save hardware resource.
- A novel hardware architecture for Top-$k$ ListNet, that exploits the flexibility of FPGA devices by using custom precision datapaths.
- Results showing how our novel sampling optimisations and our FPGA implementation of Top-$k$ ($k$=3) ListNet can achieve 40.51x and 4.42x speedups over a single Intel Xeon CPU and Nvidia GTX 1080T GPU implementation respectively, with 2% accuracy loss on the MQ2008 data set of LETOR 4.0, which consists of 471 queries and 540679 documents.

The remainder of this chapter is organised as follows: Section 4.2 discusses the computational complexity of Top-$k$ ListNet. Section 4.3 provides the design of Top-$k$ ListNet for custom

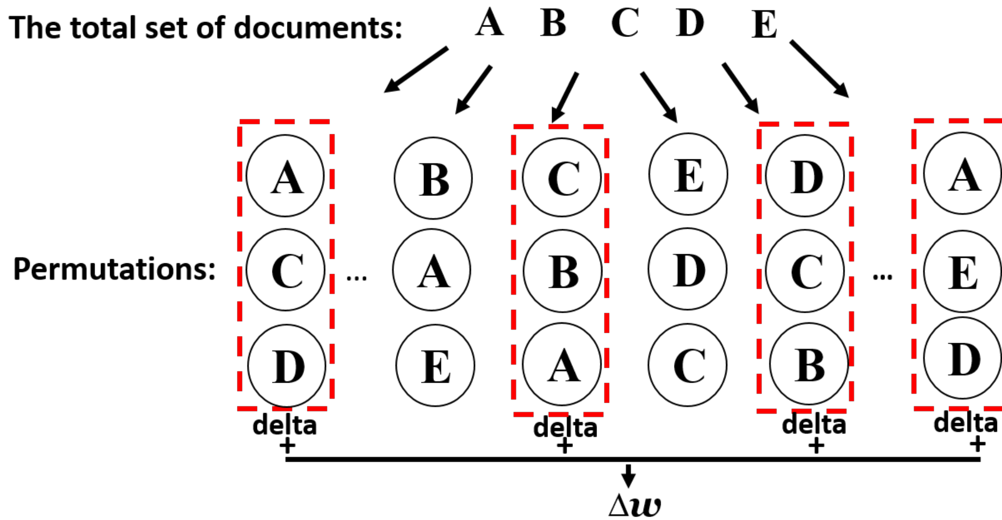


Figure 4.1: Stochastic Top-3 ListNet. Each box represents a selected permutation, which is selected randomly

precision accelerator. Section 4.4 presents the experiment and analyses the experimental results. Section 4.5 summarizes the chapter.

## 4.2 Position-Aware Sampling Approach

In this section, we will discuss our sampling approach for the Top-$k$ ListNet algorithm. A new computation approach is proposed which reduces computations for each query and also shows a better accuracy than previous approaches.

### 4.2.1 Optimisation of Top-$k$ ListNet Scheduling

When using stochastic Top-$k$ approach to compute the probability $P(\boldsymbol{s}, \boldsymbol{\pi})$, the probability of permutations is calculated by sampling the permutation class, and the permutations are normally generated sequentially. As shown in Figure 4.1, each vertical dashed box is a permutation.

The precomputation method proposed in Chapter 3 for the *Luce model* enables permutations to be computed in parallel. It becomes tractable to compute the probability of each position

**Algorithm 5** Optimised Top-$k$ ListNet scheduling

```
1: Input: number of epochs n_epoch, number of queries n_q, document features x, learning rate η
2: Output: w^21, w^32
3: Initialization: w^21 = random(n_h, n_f), w^32 = random(n_h)
4: for e = 1 to n_epoch do
5:     for q = 1 to n_q do
6:         ▷ Forward Propagation
7:         Initialization: s^32 = 0, s = 0, Δw = 0
8:         s^32 = Sigmoid(w^21 x)
9:         s = Sigmoid(w^32 s^32)
10:        for t_1 = 1 to n_d do
11:            ...
12:            for t_2 = 1 to n_d do
13:                for t_k = 1 to n_d do
14:                    π=generate_top(t_1, t_2, ..., t_k)
15:                    ▷ Loss Function
16:                    L_π = −P(y, π) log(P(s, π))
17:                    ▷ Back Propagation
18:                    dw = ∂L_π/∂w = (∂s/∂w)(∂L_π/∂s)
19:                    Δw+ = dw
20:                end for
21:            end for
22:            ...
23:        end for
24:        ▷ Update Parameters
25:        w = w − ηΔw
26:    end for
27: end for
```

for different permutations in parallel, which is showed in Algorithm 5. This allows us to sample from a controlled set of position levels. It should be noted the number of samples for different positions are the same in stochastic Top-$k$ approach (The boundaries from line 10 to 14 are the same in Algorithm 5).

### 4.2.2 Position-Aware Sampling Scheme

We propose a position-aware ListNet, which samples a subset of documents for each position. As shown in Figure 4.2, each horizontal red box represents one position. The concrete number of samples for each position is different, and is based on the importance of the position. The population of documents for each position is $n_d$, so it is $kn_d$ for $k$ positions. The small number of

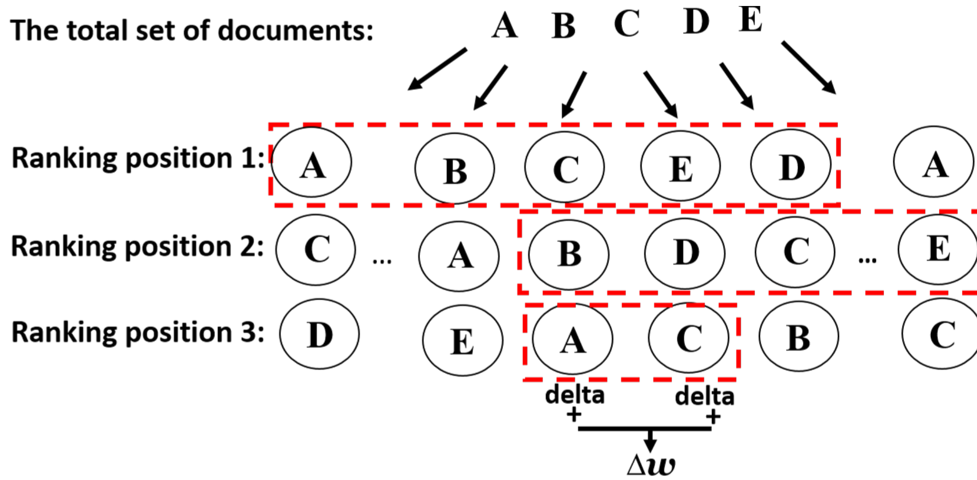


Figure 4.2: Position-aware Top-3 ListNet. Each box is one position, the number of samples in each box decreases from top to bottom

the population has a higher probability to select high-weighted samples. Thus the new sampling approach may achieve a higher accuracy for a given number of samples, compared to existing sampling approaches. In order to verify this hypothesis, we applied the position-aware Top-$k$ ListNet ($k$=3) implementation to a random subset of the benchmark data set LETOR 4.0, the subset consists of 40 queries and 46329 documents. The number of samples should decrease from position 1 to position $k$, which can be chosen by multiplying sampling factors $\boldsymbol{f}$ with the [0, 1] interval, as Algorithm 6. The values of $\boldsymbol{f}$ should fulfill $1 > f_1 > f_2 > ... > f_k > 0$.

As discussed in Chapter 3, the computational complexity of ListNet is decided by Loss Function calculation and BP calculation, as $n_d$ grows the bottleneck becomes larger. From Algorithm 6, we can see the complexity of position-aware ListNet ($k$=3) is $O\left((n_d)^3 f_1 f_2 f_3\right)$, where $f_1 f_2 f_3 << 1$, while the complexity of conventional Top-3 ListNet is $O\left((n_d)^3\right)$. The position-aware approach can reduce complexity by about $O(f_1 f_2 f_3)$ for any value of $n_d$.

To find an optimised set of values for the sampling factors of each position, we fixed the value of $f_1 f_2 f_3$ to $m$ (the parameter of line 10 in Algorithm 2) so that the complexity equals to that of the stochastic sampling approach, and trained position-aware ListNet with different sampling factor sets on the subset of LETOR 4.0 data set for 300 epochs. It is reasonable to fix $f_1$ to 1 because the position 1 is clearly the most important position. Then we varied the values of $f_2$ and $f_3$, compared the ranking accuracy with the stochastic approach. Before starting training,

**Algorithm 6** Position-aware Top-$k$ ListNet scheduling

1: Input: number of epochs $n_{epoch}$, number of queries $n_q$, document features $\boldsymbol{x}$, learning rate $\eta$, bit-width $bw_1$ $bw_2$, integer bits $iw$
2: Output: $\boldsymbol{w}^{21}, \boldsymbol{w}^{32}$
3: Initialization: $\boldsymbol{w}^{21} = random(n_h, n_f)$, $\boldsymbol{w}^{32} = random(n_h)$, $F = 2^{13}$, $\boldsymbol{B} = (1, 2, ..., M)$
4: **for** $e = 1$ to $n_{epoch}$ **do**
5: $\quad F = 2^{13+floor(e/30)}$
6: $\quad$ **for** $q = 1$ to $n_q$ **do**
7: $\quad\quad$ ▷ Forward Propagation
8: $\quad\quad$ Initialization: $\boldsymbol{s}^{32} = \boldsymbol{0}$, $\boldsymbol{s} = \boldsymbol{0}$, $\Delta\boldsymbol{w}^{32} = \boldsymbol{0}, \boldsymbol{w}^{21} = \boldsymbol{0}$
9: $\quad\quad$ $\boldsymbol{s}^{32} = \text{Sigmoid}(\boldsymbol{w}^{21}\boldsymbol{x})$
10: $\quad\quad$ $\boldsymbol{s} = \text{Sigmoid}(\boldsymbol{w}^{32}\boldsymbol{s}^{32})$
11: $\quad\quad$ **for** $t_1 = 1$ to $f_1 * n_d$ **do**
12: $\quad\quad\quad$ ...
13: $\quad\quad\quad$ **for** $t_2 = 1$ to $f_2 * n_d$ **do**
14: $\quad\quad\quad\quad$ **for** $t_k = 1$ to $f_k * n_d$ **do**
15: $\quad\quad\quad\quad\quad$ $\boldsymbol{\pi}$=generate_top$(t_1, t_2, ..., t_k)$
16: $\quad\quad\quad\quad\quad$ ▷ Loss Function
17: $\quad\quad\quad\quad\quad$ $P(\boldsymbol{y}, \boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{\exp(\boldsymbol{y}_{\pi(j)})}{\sum_{i=j}^{n_d} \exp(\boldsymbol{y}_{\pi(i)})}$
18: $\quad\quad\quad\quad\quad$ $P(\boldsymbol{s}, \boldsymbol{\pi}) = \prod_{j=1}^{k} \frac{\exp(\boldsymbol{s}_{\pi(j)})}{\sum_{i=j}^{n_d} \exp(\boldsymbol{s}_{\pi(j)})}$
19: $\quad\quad\quad\quad\quad$ $L_\pi = -P(\boldsymbol{y}, \boldsymbol{\pi}) \log(P(\boldsymbol{s}, \boldsymbol{\pi}))$
20: $\quad\quad\quad\quad\quad$ ▷ Back Propagation
21: $\quad\quad\quad\quad\quad$ Compute $\Delta\boldsymbol{w}$ according to Algorithm 7
22: $\quad\quad\quad\quad$ **end for**
23: $\quad\quad\quad$ **end for**
24: $\quad\quad\quad$ ...
25: $\quad\quad$ **end for**
26: $\quad\quad$ ▷ Update Parameters
27: $\quad\quad$ $\boldsymbol{w}^{32} = \boldsymbol{w}^{32} + \eta\Delta\boldsymbol{w}^{32}/F/2^{\boldsymbol{B}}$
28: $\quad\quad$ $\boldsymbol{w}^{21} = \boldsymbol{w}^{21} + \eta\Delta\boldsymbol{w}^{21}/F/2^{\boldsymbol{B}}$
29: $\quad$ **end for**
30: **end for**

the input data was preprocessed. By analysing the value distribution of each feature, we found the value of 6 features kept 0 without varying. This allowed us to apply the pruning technique to avoid redundant computation [66, 67]. In our application, the computation time was reduced by 13%, while still retaining the model accuracy.

Using the NDCG evaluation measure, Figure 4.3 illustrates that the value of $f_2/f_3$ is larger, the accuracy of the model is better. However, we collected experiment results up to maximum ratio of $f_2/f_3$ equals to 250, as the computation becomes Top-2 ListNet when the ratio of $f_2/f_3$ is larger than 250. Figure 4.3 also verifies the hypothesis position-aware approach delivers

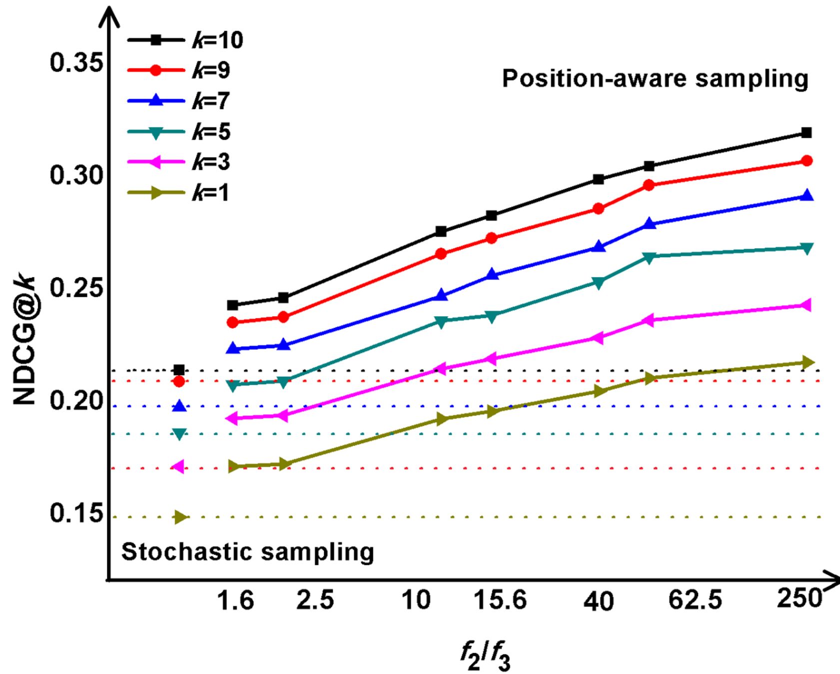


Figure 4.3: NDCG@$k$ comparison among different sampling factor sets, NDCG@$k$ means it calculated at the first $k$ positions

a better accuracy than traditional stochastic sampling method under the same computational complexity. We also compared the performance with other popular ranking algorithms based on CPU platform, the results in Figure 4.4 indicate position-aware Top-$k$ ($k$=3) ListNet achieves best accuracy over 300 epochs, while training time is a issue needs to solve.

## 4.3 Mixed Precision Fixed Point Implementation

Converting the position-aware ListNet algorithm to fixed point should lead to a large efficiency improvement [68, 69, 70, 71]. However, despite advances in fixed point implementation for NN algorithms in recent years, traditional fixed point quantisation method is not effective for the ListNet algorithm because it has variables with a large range.

### 4.3.1 Organisation of Computation Tasks

As seen in Chapter 2, Top-$k$ ListNet is split into three tasks: FP; Loss Function; and BP. As BP is the most compute and data intensive task of position-aware ListNet, we only converted

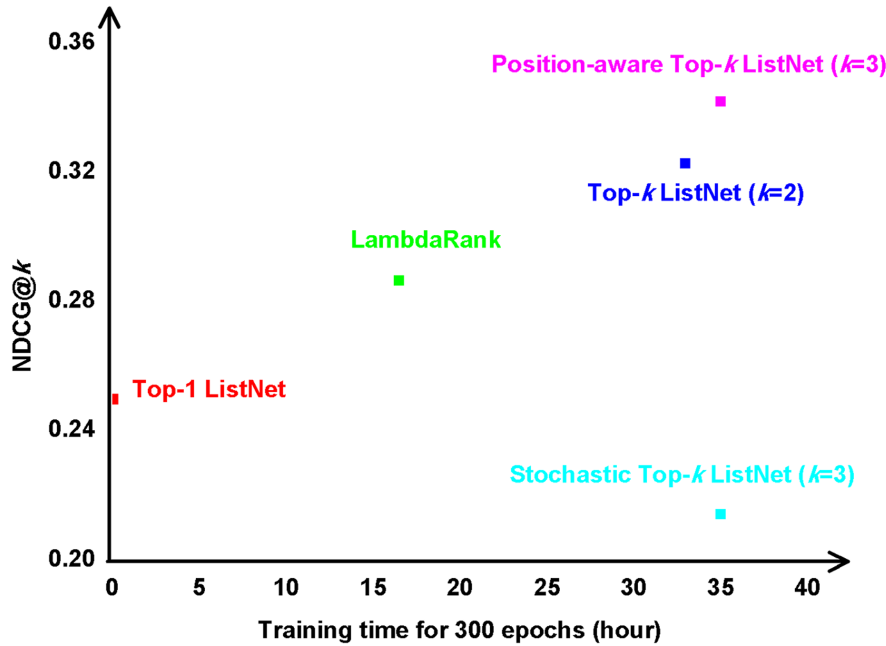


Figure 4.4: Performance comparison among different ranking algorithms using CPU platform over 300 epochs

the BP algorithm and computation process to fixed point. Because the input data $\boldsymbol{x}$ (feature vectors) are inherently unrelated features, which are independently and identically distributed, both activations $\boldsymbol{s}^{32}$ $\boldsymbol{s}$ (line 9 and 10 in Algorithm 6) and permutation probabilities $\boldsymbol{P}$ (line 17 and 18 in Algorithm 6) which are inputs of BP task will have slowly changing during training, such that exponents can be predicted with high accuracy based on exponent distributions seen in recent epochs. As a result, the fixed point format is well-suited to position-aware ListNet. Meanwhile, the ranges of these variables are in the [0, 1] interval, which means they can be represented as unsigned integers without much concern for scaling [72, 73].

The computation in BP is multiply-and-accumulate (MAC) operations, where the multiplication is to get the gradients for each permutation (line 6, 7, 8 in Algorithm 7), these tiny values are accumulated for updating (line 12 and 13 in Algorithm 7). We use less bits for the operands of the multipliers, and more bits for the accumulators. That means bit-width $bw_1$ (line 4 in Algorithm 7) is smaller than bit-width $bw_2$ (line 10 in Algorithm 7). The high precision in the accumulators ensures the accuracy of the final outputs, while low precision in the multiplication can improve the computational efficiency due to the low design complexity.

In order to achieve the largest possible performance in our application, we map all the com-

**Algorithm 7** The back propagation scheduling

1: Input: $\boldsymbol{s}^{32}$, $\boldsymbol{s}$, $P(\boldsymbol{y}, \boldsymbol{\pi})$, $P(\boldsymbol{s}, \boldsymbol{\pi})$, $F$, bit-width $bw_1$ $bw_2$
2: Initialization: $B_\pi = 0$
3: $B_\pi = M - floor(P(\boldsymbol{y}, \boldsymbol{\pi}) * M)$
4: Calculate_gradient[ap_uint$< bw_1 >$]$(\boldsymbol{x}, \boldsymbol{s}^{32}, \boldsymbol{s}, P(\boldsymbol{y}, \boldsymbol{\pi}), P(\boldsymbol{s}, \boldsymbol{\pi}))$→[ap_uint$< bw_1 >$]$(d\boldsymbol{w})$
5: {
6: var $u = \frac{\partial L_\pi}{\partial \boldsymbol{s}} = P(\boldsymbol{y}, \boldsymbol{\pi})(1 - P(\boldsymbol{s}, \boldsymbol{\pi})) * F * 2^{B_\pi}$
7: $d\boldsymbol{w}^{32} = u\boldsymbol{s}^{32}$
8: $d\boldsymbol{w}^{21} = u\boldsymbol{s}^{32}(1 - \boldsymbol{s}^{32})\boldsymbol{x}$
9: }
10: Accumulate_gradient[ap_uint$< bw_1 >$]$(d\boldsymbol{w}^{32}, d\boldsymbol{w}^{21})$→[ap_uint$< bw_2 >$]$(\Delta\boldsymbol{w})$
11: {
12: $\Delta\boldsymbol{w}^{32} = \Delta\boldsymbol{w}^{32} + d\boldsymbol{w}^{32}$
13: $\Delta\boldsymbol{w}^{21} = \Delta\boldsymbol{w}^{21} + d\boldsymbol{w}^{21}$
14: }

putation tasks (FP, Loss Function, and BP) to an FPGA device, rather than preserving some parts in software. FP and Loss Function are still computed in floating-point formats (32 bits), then convert the results to fixed point representations, and then BP computation starts in fixed point formats. The concrete organisation is shown in Figure 4.5.

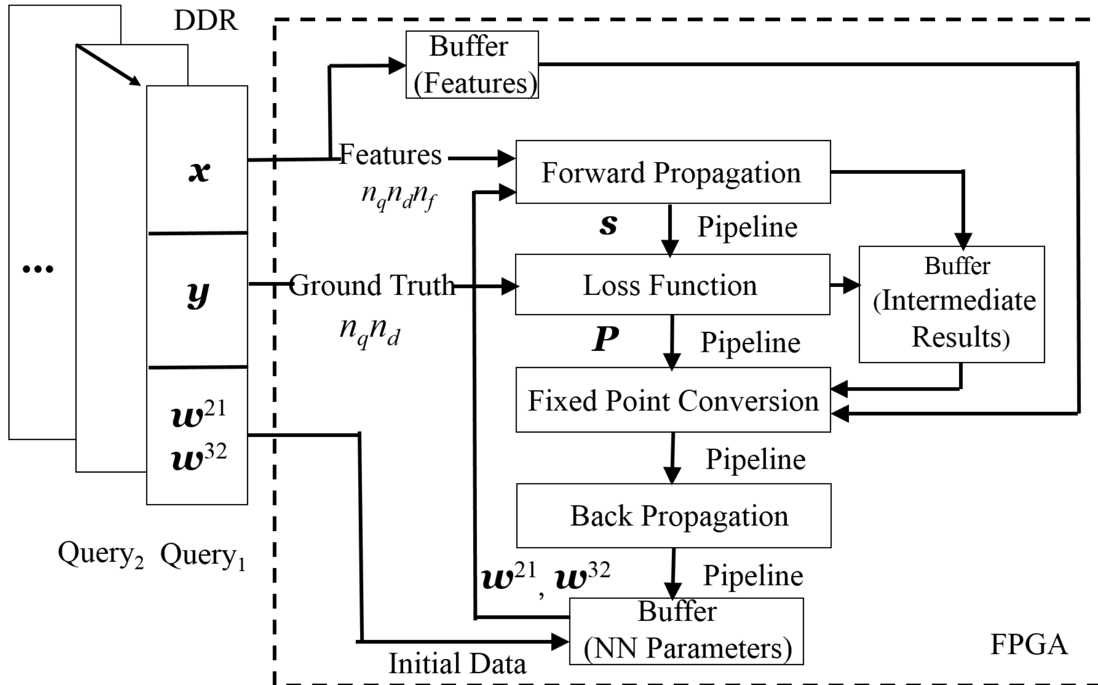


Figure 4.5: The organisation of computation tasks on FPGA

### 4.3.2 Limitation of Traditional Fixed Point Implementation

In order to convert to fixed point computation effectively, we estimate the trend of gradient values with respect to the parameters of ranking model ($d\boldsymbol{w}^{32}$ of line 7 and $d\boldsymbol{w}^{21}$ of line 8 in Algorithm 7) by tracking the value distribution of each epoch. Because the gradient values diminish slowly, the exponent can be updated to better utilize the available range of fixed point representation. As shown in Figure 4.6, we collected the distribution of gradients across all $\boldsymbol{\pi}$ of several random queries in floating-point training for 300 epochs. We find that in practise the gradient of loss function tends to be dominated by small magnitudes (negative exponents). We also investigated the trend of gradient values over epochs by analysing the change of the mean value and maximum value, as shown in Figure 4.7. If we convert to fixed point representation directly without considering the value change over epochs, much of the fixed point representable range was left unused, while many values were below the minimum representable range and rounded to zeros. That means we still need many bits to guarantee the quality of the model.

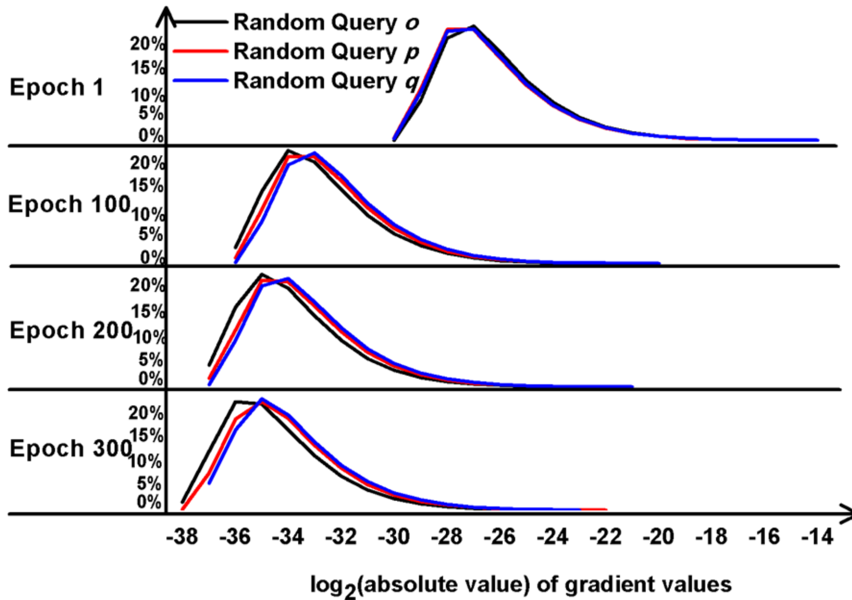


Figure 4.6: Distribution of gradient values of loss function over 300 epochs

One efficient way is to shift the gradient values of loss function by scaling them up (increase the exponent) so that these values occupy more of the representable range and preserve values that are otherwise lost to zeros [74, 75]. The scaling process should be done at the beginning of BP calculation (line 6 in Algorithm 7, $F$ is the scaling factor). This requires no extra operations

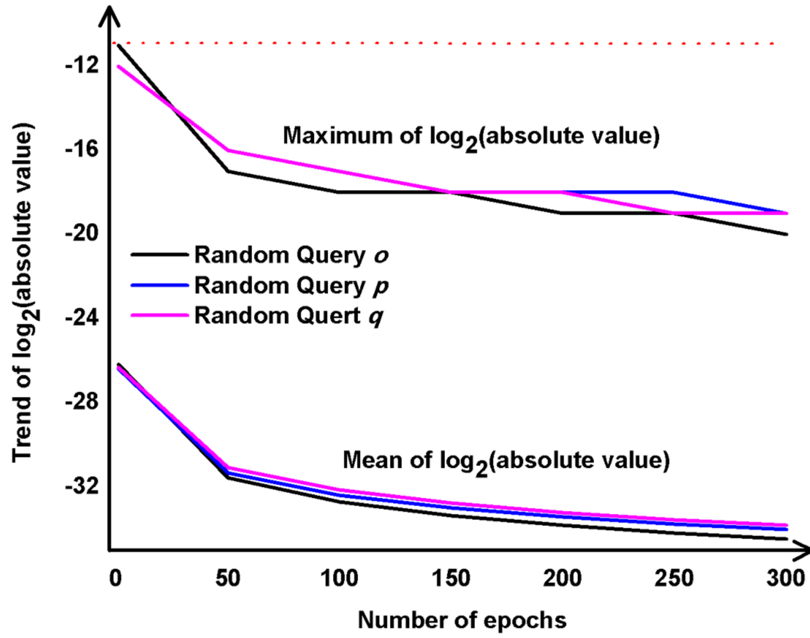


Figure 4.7: The trend of mean and maximum gradient value of loss function over 300 epochs

during BP calculation. Gradient values must be unscaled before the parameters of ranking model update to maintain the update magnitudes (line 27 and 28 in Algorithm 6).

There are several options for choosing the loss scaling factor $F$, the most effective one is to pick an adaptive loss scaling factor which is updated over epochs. With the loss scaling factor, the maximum gradient value should not overflow.

### 4.3.3 Batch Fixed Point Implementation

Different from the previous quantisation methods where scaling factor of each scalar variable is fixed during each epoch, we proposed a new quantisation method. Because the gradients of loss function are based on permutation probabilities, and the probabilities of different permutations are quite different in magnitudes, the whole range of the gradients is quite wide for each epoch. That means we need many bits to represent the whole range using the traditional quantisation method. If the gradient values are classified into $M$ batches, the fixed point format with limited bits is enough to represent the values of every batch by multiplying different batch scaling factors for each batch, as shown in Figure 4.8. Furthermore, the scaling factor varies for the same batch over epochs. The fixed point format with limited bits will consume less

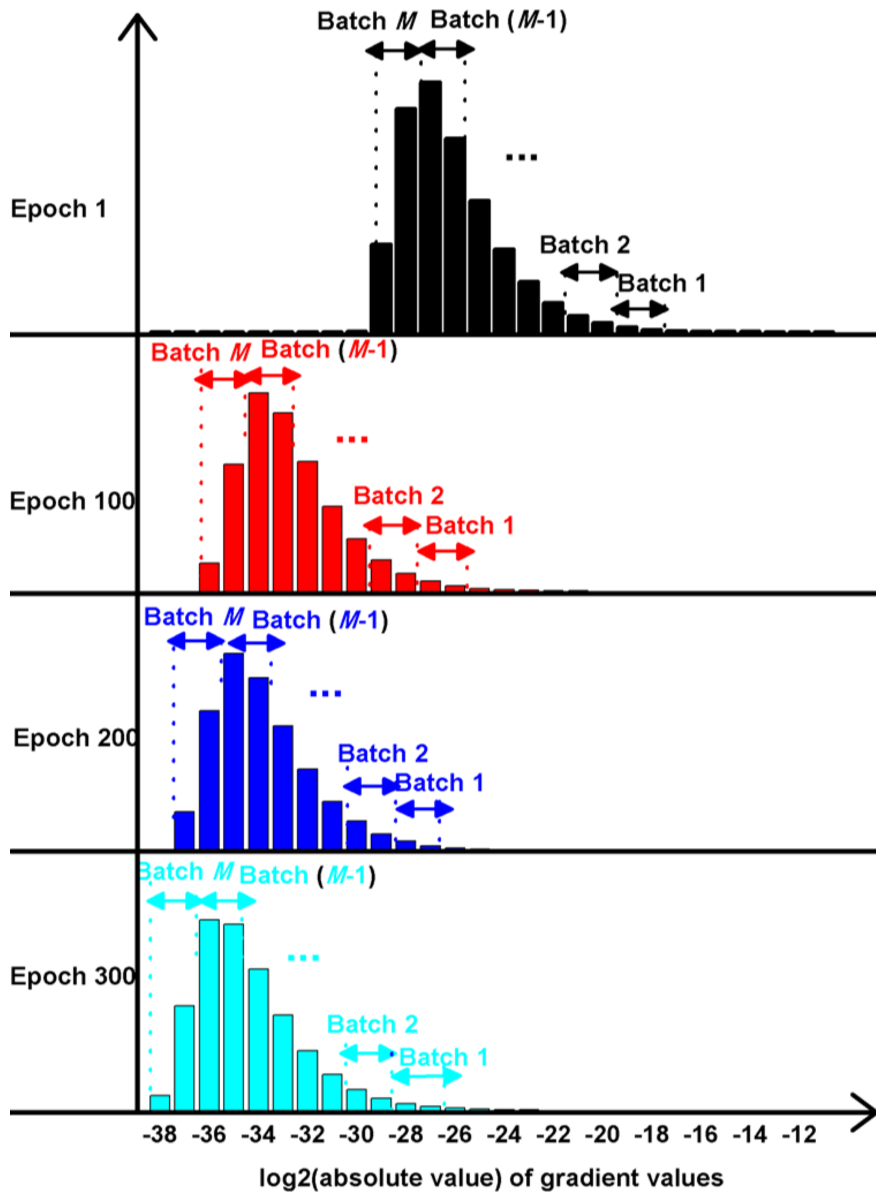


Figure 4.8: The theory of batch implementation

resource in computation. Because the gradient value of each permutation is mainly decided by $P(\boldsymbol{y}, \boldsymbol{\pi})$, we can sort the permutations based on the $P(\boldsymbol{y}, \boldsymbol{\pi})$ and assign a batch scaling factor to each permutation (line 3 in Algorithm 7). The value of batch scaling factor is from $2^1$ to $2^M$, where $M$ is a positive integer. For the permutations with large gradient values, we assign a small batch scaling factor to avoid overflow. For the permutations with small gradient values, we assign a large batch scaling factor so that the gradient values do not round to zero.

### 4.3.4 Case Study

In order to demonstrate the high resource utilisation of batch quantisation, we implemented position-aware Top-$k$ ListNet ($k$=3) on the subset of LETOR 4.0 data set. We chose to assign unsigned integers with the same bit-widths to all the inputs of BP, and applied to the BP computation (line 4 in Algorithm 7). When decreasing the bit-width, the accuracy loss increases. Figure 4.9(a) shows direct fixed point quantisation needs at least 27 bits for both $bw_1$ and $bw_2$ within 4% accuracy loss and 36 bits without accuracy loss. Under 300 MHz clock frequency, it does not bring much benefit compared to floating implementation on an FPGA as table 4.1. The reason we chose 4% loss as the metric (the dash line in Figure 4.9) is that the NDCG@10 of floating-point implementation on LETOR 4.0 is in the [0.34, 0.40] interval, 4% accuracy loss varies the interval to [0.30, 0.36], this does not cause position-swap to the documents. Here, the NDCG@10 results of floating-point implementation is collected from RankSVM, RankBoost, ListNet, and AdaRank [1].

| Data Type | LUTs | Regs | DSP | Speedup over GPU |
|---|---|---|---|---|
| Floating-Point | 71,962 | 129,526 | 654(100.0%) | 1.42 |
| Direct fixed point 27-bit | 56,768 | 93,178 | 613(93.7%) | 1.43 |
| Adaptive fixed point 10-bit | 53,244 | 86,289 | 231(35.3%) | 1.44 |
| Batch fixed point 8-bit | 57,735 | 67,183 | 102(15.6%) | 1.43 |

Table 4.1: Consumption of hardware resource and speedup over GPU implementation for different data types

Based on the direct fixed point quantisation, we applied the loss scaling factor $F$ as a power of 2, and initialized it by tuning the exponential $j$ ranging from 10 to 20 (line 3 in Algorithm 6) to choose a value so that it does not cause overflow, and 13 is sufficient to fulfill the requirement. And then we updated $j$ over epochs based on the trend of gradient values in Figure 4.7, increased the value $j$ linearly by 1 per 30 epochs for simplicity (line 5 in Algorithm 6). As shown in Figure 4.9(b), 10 bits for $bw_1$ are needed for the accuracy requirement and 22 bits are sufficient to match the accuracy achieved with floating-point training. $bw_2$ needs 10 more bits. As table 4.1 it only consumes 40% of DSP resource compared to direct quantisation.

Finally we implemented the algorithm with applying batch quantisation approach. We varied the value of $M$ to explore the optimal batch number. Comparing the training results under

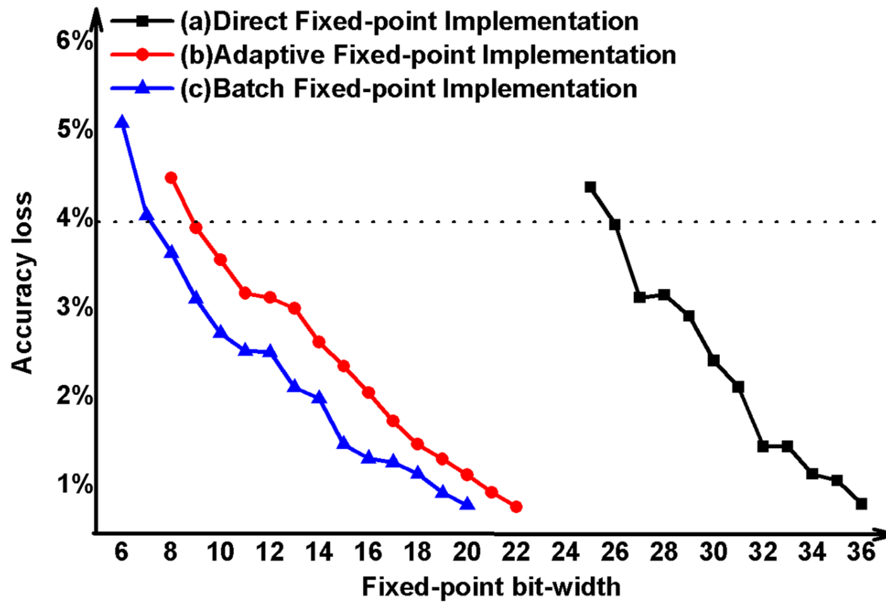


Figure 4.9: Accuracy loss with different fixed point bit-widths $bw_1$ compared to floating point implementation

| Number of Bits | LUTs | Regs | DSP |
|---|---|---|---|
| 22 | 52,351 | 78,125 | 360 |
| 20 | 61,917 | 80,011 | 231 |
| 16 | 50,160 | 72,202 | 231 |
| 10 | 48,609 | 68,287 | 231 |
| 8 | 57,735 | 67,183 | 102 |

Table 4.2: Consumption of hardware resource under different numbers of bits for batch quantisation implementation

different batches, the implementation with 10 batches shows the best result. Figure 4.9(c) illustrates it needs less bits (8 bits for $bw_1$) within the acceptable accuracy loss. $bw_2$ needs 10 more bits. As shown in Table 4.2, the decrease of required bits leads to less resource consumption. Comparing to the original adaptive quantisation method in Table 4.1, batch quantisation can save 50% DSP resource. The computational efficiency is 6.41x better than floating-point implementation due to a improvement in resource utilisation, and metric of computational efficiency is OP/s/DSP.

## 4.4 Experiment Results and Analysis

To evaluate the performance and accuracy of our batch quantisation approach discussed in Section 4.3.3, we implemented it on the whole LETOR 4.0 data set and compared the performance of our batch quantisation approach to that of a floating point GPU implementation. The target platform was a Xilinx ZCU102 development board and SDSoC was used to generate the RTL.

### 4.4.1 Analysis of the Performance

Table 4.3 shows the performance of our design – our batch quantisation method achieved a maximum frequency of 300 MHz resulting in 390.50 GOP/s. By implementing the computation in parallel as much as possible, we achieved 40.51x speedup over a single Intel Xeon 1.6 GHz CPU implementation, while 4.42x speedup over one Nvidia GTX 1080T GPU implementation using the same board, although overall accuracy is about 2% lower. The CPU implementation deployed the benchmark source code RankLib package to guarantee the computation speed [61]; the GPU implementation used another open source library called TensorFlow Ranking to achieve largest parallelism. For the CPU implementation, as general-purpose systems, homogeneity is highly prior, improvements in efficiency is difficult to obtain. For GPU implementation, although we unrolled the permutation computation beforehand, it is difficult to take full advantage of the computation ability of the GPU because of limitation of memory bandwidth. In Table 4.4, we also displayed the resource consumption and computation efficiency under two different data types. In order to quantify the contribution to speedup from data parallelism and pipeline, we modified the FPGA implementation to apply either loop unrolling or pipelining, with the results shown in Table 4.5. The data parallelism brings 31.95x speedup, while the pipelining achieves 2.72x speedup. The data parallelism has a larger effect than the pipeline. Table 4.6 displays the resource consumed under different optimisation strategies, which verifies the theory that that pipeline directive requires additional registers to improve the throughput, while the parallel directive reduces the latency by consuming more resource.

When we investigated why our batch fixed point implementation, that has a wordlength of 8

| Data Type | Accuracy | Time(s) | Speedup over CPU | Speedup over GPU |
|---|---|---|---|---|
| Floating-Point | 1 | 32.11 | 13.08 | 1.43 |
| Batch fixed point 8-bit | 0.9399 | 10.37 | 40.51 | 4.42 |

Table 4.3: The performance comparison of different data types implemented on FPGA. The standard time consumption on CPU is 420.05s per epoch, GPU is 45.8s

| Data Type | LUTs | Regs | DSP | Computation Efficiency |
|---|---|---|---|---|
| Floating-Point | 71962 | 129526 | 654 | 0.06 |
| Batch fixed point 8-bit | 128680 | 123090 | 211 | 0.55 |

Table 4.4: The resource required and computation efficiency for different data types implemented on FPGA. Metric of computational efficiency is OP/s/DSP

bits, caused only a 2% drop in accuracy we came to the conclusion that the gradient noise caused by low precision implementation actually helps convergence. We also believe that this noise encourages faster exploration and annealing of optimisation space, which helps network generalisation performance.

| Optimisation | Time(s) | Speedup | Computation Efficiency |
|---|---|---|---|
| Original | 6947.92 | 1.00 | 0.02 |
| Pipeline | 2556.88 | 2.72 | 0.04 |
| Parallelism | 217.45 | 31.95 | 0.03 |
| Pipeline+Parallelism | 10.37 | 669.77 | 0.55 |

Table 4.5: The contribution of data parallelism with unrolling factor $U = 150$ (under fanout limitation) and pipeline with initiation interval $II = 11$ to speedup. Metric of computational efficiency is Gop/s/DSP

## 4.5 Expectation

The proposed techniques can be extended to higher $k$. The sampling technique is designed for a large sample population. When $k$ is larger, the full set of permutation class is larger, so the position-aware method is more competitive than previous sampling methods in selecting high-weighted samples. In addition, The fixed point technique is designed for variables with wide range, and when value of $k$ is larger, the number of permutations is larger. That means the range of gradient values is wider for large $k$, the proposed quantisation method will be more

| Optimisation | LUTs | Regs | DSP |
|---|---|---|---|
| Original | 31,249 | 41 878 | 26 |
| Pipeline | 34,079 | 46 407 | 41 |
| Parallelism | 70,631 | 88 898 | 223 |
| Pipeline+Parallelism | 138,680 | 123 090 | 211 |

Table 4.6: The consumption of hardware resource under different strategies for 8-bit batch quantsation

effective. $k$ is larger, the proposed techniques are more feasible. So we just tested for small $k$ ($k = 3$).

## 4.6 Summary

This chapter introduces a position-aware Top-$k$ ListNet algorithm and explores its acceleration on FPGA hardware using a custom precision fixed point implementation. Our novel algorithm reduce the complexity of Top-$k$ ListNet by sub-sampling the number of training documents, however, unlike previous approaches it was able to select high-weighted samples effectively and improve the accuracy with taking into consideration document position. We also explore how we can create an efficient FPGA implementation of our approach – our batch quantisation method allows to represent one single scalar variable with a wide range using different precision representations. Using a standard ranking benchmark, our position-aware Top-$k$ ListNet hardware implementation shows a better ranking accuracy than the previous stochastic approach. Compared to an Intel Xeon CPU and one Nvidia GTX 1080T GPU implementations, our batch quantisation achieves 40.51x and 4.42x speedups repectively, with a 2% accuracy loss. After the promising improvements demonstrated with this work, in the future we plan to explore the potential to accelerate the ListMLE, using multi-FPGA devices.

# Chapter 5

# Novel Reconfigurable Implementation of Top-$k$ ListNet on FPGA

## 5.1 Introduction

Ranking, which is to order documents based on the relevance degree, is widely used in question answering, video retrieval, and web search. Machine learning techniques called "learning to rank" is commonly used to solve ranking problems. Among different ranking approaches, listwise approach is the most promising. Take Top-$k$ ListNet as an example, it delivers a better accuracy than traditional tanking approaches. However, the main downside of Top-$k$ ListNet is the high computational complexity, especially for large data sets. In order to deal with this issue, quantisation method is proposed to compute with sampling approaches, which represents the variables approximately with limited number of bits. Although quantisation method delivers a higher training speed, it also affects the quality of ranking models.

Based on the results of Table 4.3, we can see the fixed point implementation leads to speedups over both CPU and GPU implementations using the floating point implementation. However, pure fixed point implementation inevitably sacrifices the model accuracy. Some information retrieval applications have a high accuracy requirement in the ranking result, for example, a navigational query. While applying pure floating point implementation, the training time

becomes an issue.

Due to the capabilities and sizes have increased, FPGAs attract more and more attention. The reconfigurable feature of FPGAs allows to change the functions at runtime in response to application requirements. A wide range of applications exploiting the reconfiguration have been discussed.

In this chapter, a new training methodology based on the reconfigurable feature of FPGAs is introduced, which effectively improves the trade-off between training time and model accuracy. This methodology defines two types of functions – floating point and fixed point functions, and modifies the function on board while the interface between FPGA and host remains active. We implemented the method as a way of kernel swap for position-aware Top-$k$ ListNet. The results show that swapping from fixed point kernel to floating point kernel achieves a higher accuracy against the other way round under the same training time. The major contributions of this chapter are:

- The first implementation changes the data representation of training process over epochs utilising the unique reconfigurable capability of FPGAs.
- Kernels with different data representations can be swapped automatically over epochs.
- An effective kernel swap mode is proposed, which shows a better accuracy under the same training time.
- Improvement in the model accuracy for fixed point implementation with acceptable training time sacrifice.

The remainder of this chapter is organised as follows: Section 5.2 discusses the limitation of low precision training for Top-k ListNet and presents potential solutions. Section 5.3 displays two kernel swapping modes. Section 5.4 compares the performance of the two swapping modes. Section 5.5 summarizes the chapter.

## 5.2 Training with Multi-Kernels under Different Data Representations

With the Top-$k$ ListNet model reaching new records in terms of computational complexity, dealing with the long training time becomes a crucial issue. One popular technique to tackle this problem is using the fixed point (low precision) training. This technique converts floating point operators to fixed point resulting in a higher degree of parallelism, as fixed point datapaths consume less resource. Despite the improvement in training speed using the technique, the model accuracy is a challenge in some applications.

Our proposed alternative is to use floating point training for some epochs and fixed point training for others by exploiting the reconfigurable capabilities of FPGAs. Nowadays, FPGA platforms accept synthesizable subsets of OpenCL through high level synthesis tools, such as SDAccel. Reconfiguration can be used as a way of implementing OpenCL kernels in the FPGA. In Top-$k$ ListNet training, we define a fixed point kernel and a floating point kernel, which are stored in different bitstream containers. These two kernels are compiled to the RTL language for implemention on a specific FPGA. At runtime, the host can swap the kernels according to the performance. when the training speed is slow, more epochs should use the fixed point kernel; while the model accuracy cannot fulfil the requirement, more epochs should employ the floating point kernel. Note that FPGA devices do not preserve the content of the configuration memory. Before loading a new kernel, the host has to migrate the contents from the FPGA to the host memory via PCI Express (PCIe). The time consumption for kernel swap is computed using:

$$\text{Configuration Time} = \frac{\text{Size of Bitstream File}}{\text{Bandwidth of the Configuration Mode}} \tag{5.1}$$

Training under multi-kernels delivers a higher model accuracy than fixed point training alone. In order to verify this hypothesis, we fixed the total number of epochs, trained Top-$k$ ListNet by swapping from the fixed point kernel to the floating point kernel, and also from the floating

point kernel to the fixed point kernel based on Amazon Web Services (AWS) F1 instances. The configuration time is 150 ms, which can be neglected compared to the computation time. As shown in Figure 5.1, training using different kernels with different representations is attractive, it reduces the accuracy loss compared to pure fixed point kernel implementations, particularly when the number of bits is limited. However, swapping from the fixed point kernel to the floating point kernel shows a different performance against the other way round, and the swapping point also contributes to the performance. Thus it is necessary to investigate the model accuracy and the training time under these two swapping modes.

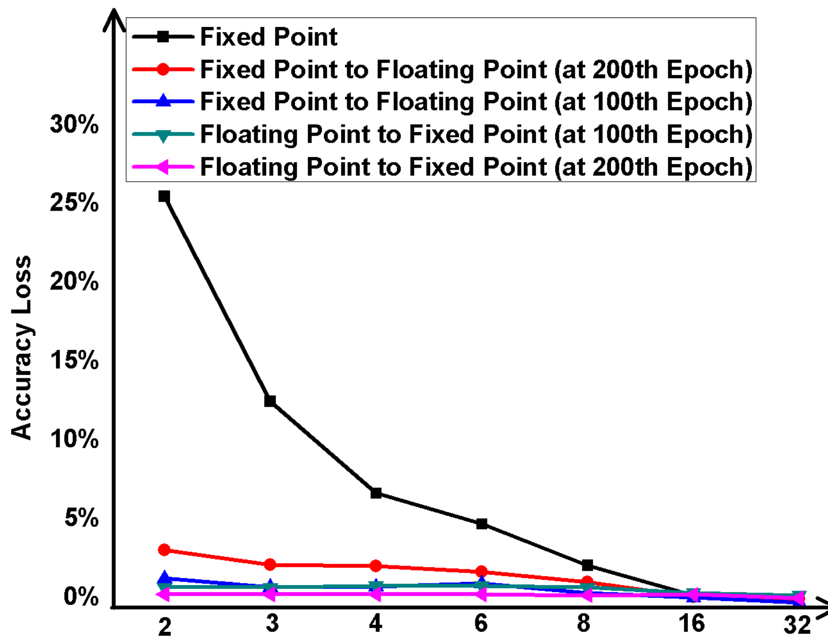


Figure 5.1: Accuracy loss for different representation types over 300 epochs compared to single floating point

## 5.3 Experiment Results and Analysis

As presented in Section 5.2, using multi-kernels in training leads to a large improvement in accuracy compared to the fixed point implementation. It achieves almost the same level of accuracy as the pure floating point implementation. However, further investigation is needed

to investigate the difference between the two swapping modes, so that we can provide a guideline for multi-kernel training.

### 5.3.1 Performance of Swapping from Floating Point Kernel to Fixed Point Kernel

One efficient way to achieve multi-kernel training is to swap from the floating point kernel to the fixed point kernel. As the gradient values diminish over epochs, swapping from floating point (full precision) kernel to fixed point (low precision) kernel guarantees the model accuracy. At the same time, it reduces the training time compared to pure floating point training.

In order to illustrate this methodology is technically sound, we fixed the total number of epochs to 300 so that it is consistent with that of the pure floating point or fixed point training, and swapped at different epochs as shown in Algorithm 8. We take 50 epochs as an interval to do swapping. If a smaller interval is applied, the experiment results of adjacent swapping points will be quite similar; while a larger interval is applied, it is difficult to collect enough experiment results to analyse the trend. We stored the two kernels – floating point and fixed point kernels in separate bitstream containers, and applied the floating point kernel first, and then swap to the fixed point kernel.

**Algorithm 8** Floating point to fixed point scheduling

1: Initialization: Swapping Point=sp
2: **for** $e = 1$ to $n_{epoch}$ **do**
3:     **if** $e < sp$ **then**
4:         Floating Point Computation
5:     **else**
6:         Fixed Point Computation
7:     **end if**
8: **end for**

As seen in Figure 5.2, the accuracy of both pure floating point and fixed point implementations increases significantly over initial epochs, and slows down. However, the accuracy gets saturated over epochs when swapping from the floating point kernel to the 8-bit fixed point kernel, although it is still higher than the results of pure fixed point training. As analysed in Chapter

4, the gradient values decrease over epochs quadratically, as shown in Figure 4.5. The gradients can be represented faithfully using the floating point format in initial epochs, thus the model accuracy is guaranteed. Using fixed point representation in latter epochs would round the smallest gradient values to zero, which causes the weights to never activate again, but it benefits the training time. In addition, it makes sense that the final accuracy is worse when swapping at earlier epochs. For instance, the final accuracy of swapping at 50th epoch is 0.331, while swapping at 250th epoch is 0.341.

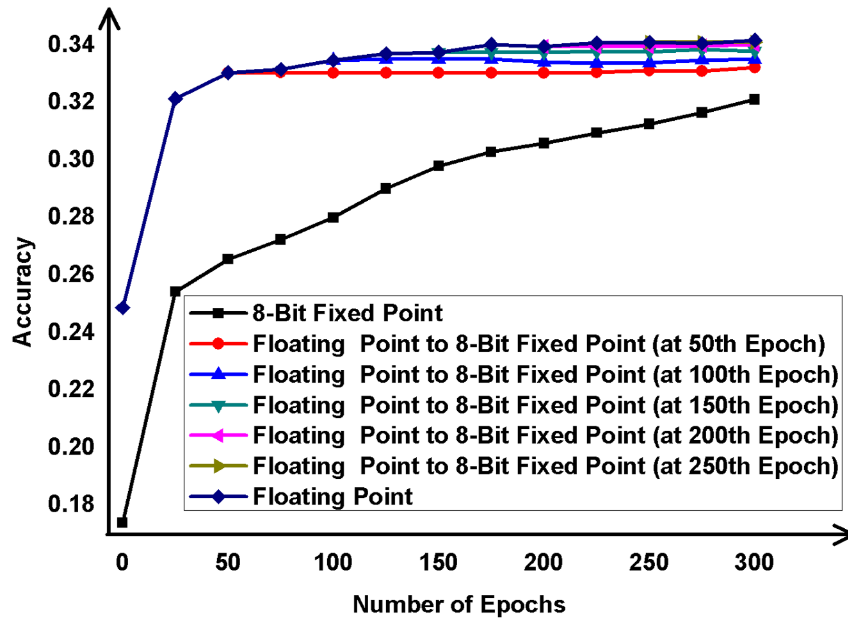


Figure 5.2: Accuracy for swapping once from floating point kernel to 8-bit fixed point kernel at different epochs

Comparing to the 8-bit fixed point kernel, Figure 5.3 illustrates using the 16-bit fixed point kernel delivers a better performance than 8-bit. The model accuracy (NDCG@10) under 16-bit fixed point implementation is 0.341 after 300 epochs, while the accuracy is 0.321 for 8-bit fixed point. This is reasonable because more bits are used to represent gradient values. Similar as 8-bit fixed point kernel, Figure 5.3 shows that swapping from the floating point kernel to the 16-bit fixed point kernel at different epochs shows similar accuracy. As displayed in Figure 5.3, 16-bit fixed point achieves the same level of accuracy as floating point implementation at the beginning epoch, and the increasing trends of the accuracy are also similar over epochs. The final accuracy of swapping approach is worse than the pure fixed point training. This is because the convergence step of floating point kernel is larger than fixed point implementation, which

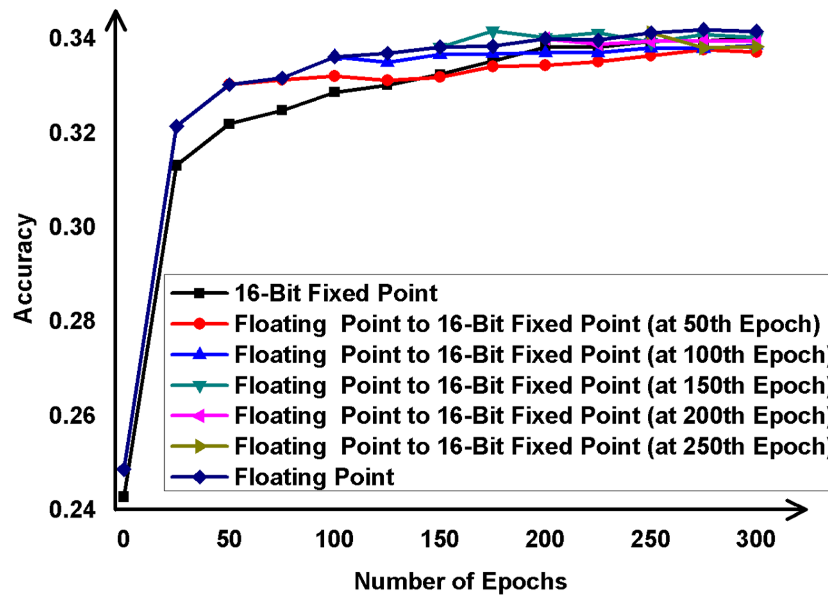


Figure 5.3: Accuracy for swapping once from floating point kernel to 16-bit fixed point kernel at different epochs

is easy to cause over-fitting. This result indicates using multi-kernels does not benefit every time.

In order to investigate the details of swapping further, we evaluated the model accuracy over the training time. We fixed the time budget to 9000 seconds, which is the time consumption of 300 epochs for the floating point implementation. 1500 seconds was chosen as the time interval to do swapping. Figure 5.4 indicates that the model accuracy for floating point over training time saturates earlier than 8-bit fixed point. In the case of 16-bit fixed point, Figure 5.5 denotes that fixed point implementation gets a higher increasing rate for the model accuracy.

The experiment results above verify that swapping from the floating point kernel to the fixed point kernel guarantees the model accuracy. It shows a better accuracy than pure fixed point implementation, when the number of bits is limited. Assuming the same number of epochs, swapping at earlier epochs sacrifices the accuracy more. On the other hand, the training time is reduced, as more epochs are under fixed point training.

Comparing the accuracy of floating point (dark blue line in Figure 5.2) and 8-bit fixed point point (black line in Figure 5.2) implementation, it seems that the increasing rate of accuracy for fixed point is larger than that of floating point, especially over 100 epochs. However, the final

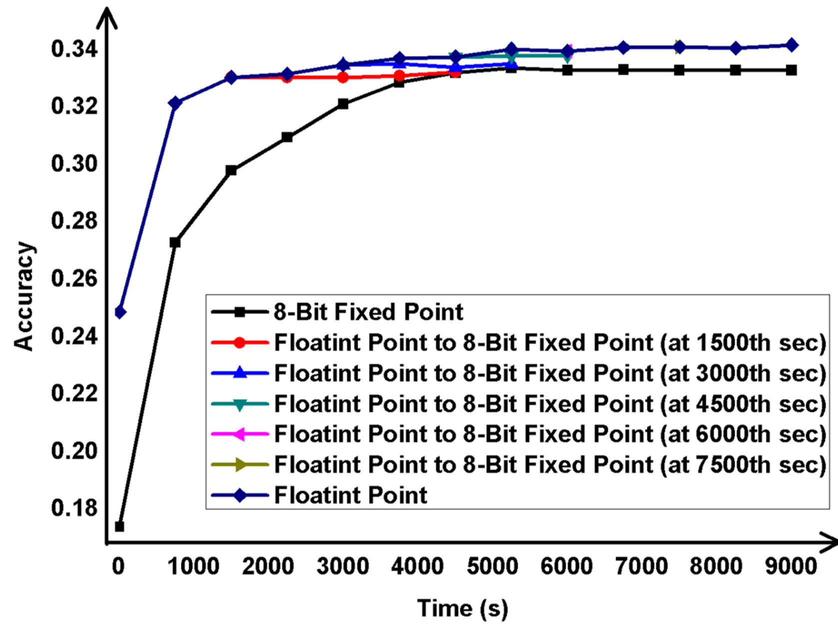


Figure 5.4: Accuracy with swapping once from floating point kernel to 8-bit fixed point kernel at different time

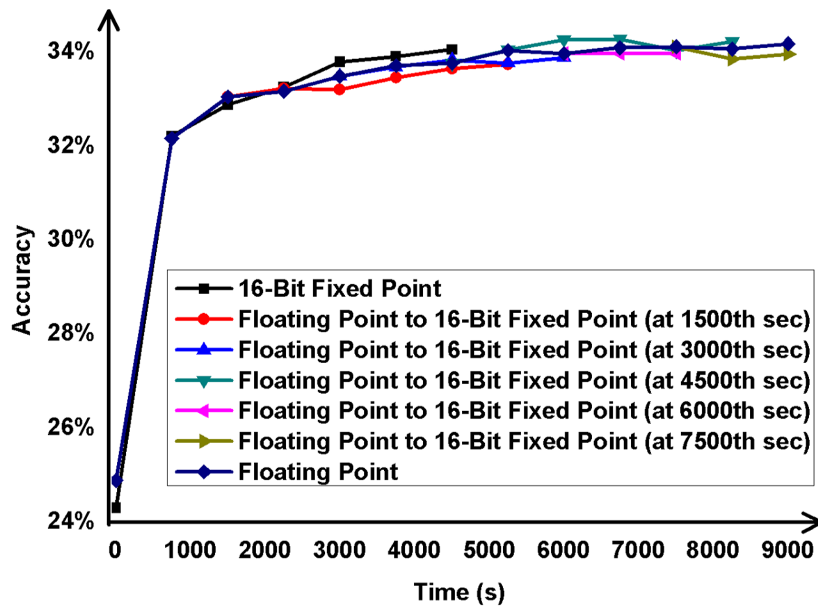


Figure 5.5: Accuracy with swapping once from floating point kernel to 16-bit fixed point kernel at different time

result of fixed point implementation is not as competitive as floating point implementation. The reason is that the accuracy of first epoch for floating point is much higher than that for fixed point. Therefore, we believe that, starting from the floating point kernel is more reasonable. In order to demonstrate floating point kernel should be deployed in the initial stage, we also test the other swapping mode – swapping from fixed point kernel to floating point kernel.

### 5.3.2 Performance of Swapping from Fixed Point Kernel to Floating Point Kernel

Another mode to realise multi-kernel training is to swap from the fixed point kernel to the floating point kernel. As the gradient values decay fast over epochs, fixed point format with limited bits may be not effective to represent gradient values at the initial stage. When using floating point format, the tiny values in latter epochs can still be represented to update the ranking model, but the contribution to model accuracy also follows the decayed trend of gradient values.

We also fixed the total number of epochs to 300, which is the same as swapping from floating point kernel to fixed point kernel, and set 50 epochs as an interval for swapping as Algorithm 9. These two kernels are exist in two different bit-stream containers, and this time the fixed point kernel is deployed first, and then swapped to the floating point kernel.

**Algorithm 9** Fixed point to floating point scheduling

```
1: Initialization: Swapping Point=sp
2: for e = 1 to n_epoch do
3:     if e < sp then
4:         Fixed Point Computation
5:     else
6:         Floating Point Computation
7:     end if
8: end for
```

Different from pure floating point and fixed point implementations, there are two significant increases for multi-kernel implementation as seen in Figure 5.6. The first jump happens at the initial stage as the other two implementations, the second jump occurs when the kernel swap happens. Swapping from the 8-bit fixed point kernel to the floating point kernel at different epochs shows different accuracy, since the epoch number of fixed point training and floating point training changes. It also shows that the final accuracy is better when swapping at earlier epochs. For instance, the final accuracy of swapping at 50th epoch is 0.341, while swapping at 250th epoch is 0.328. This is because, in Top-$k$ ListNet, gradient values indicate the difference between predicted permutations and ground truth permutations, these tiny values need many digits after radix point to represent, which is not a problem for floating point formats. The

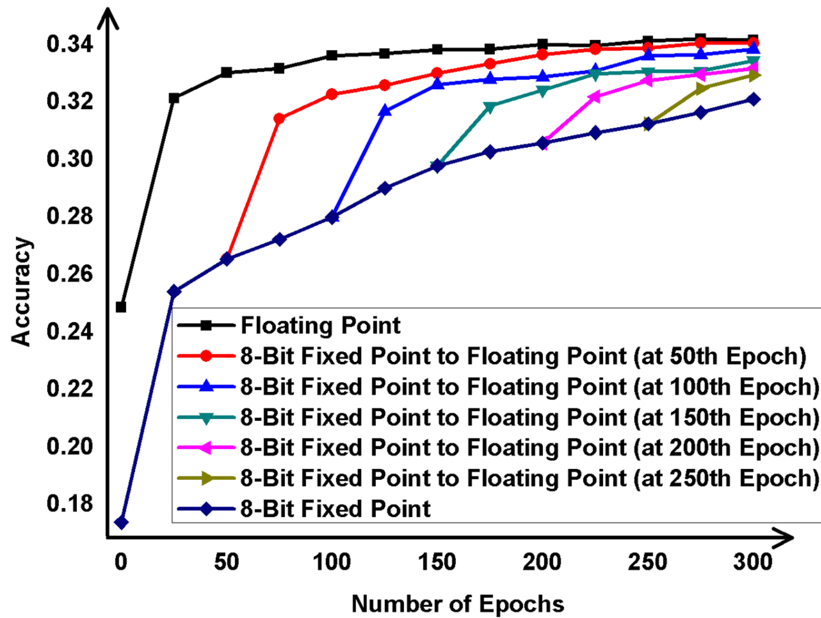


Figure 5.6: Accuracy for swapping once from 8-bit fixed point kernel to floating point kernel at different epochs

fixed point format would lead to lose some information represented by fractional bits. However, the training time will increase using floating point representations.

Compared to the results of the 8-bit kernel, Figure 5.7 denotes the contribution of the floating point kernel swapped from 16-bit kernel is not apparent. We believe the reason is that 16-bit kernel convergences faster than 8-bit kernel. When swapping at the same epoch, the gradient value for 16-bit kernel is much smaller than that of 8-bit kernel. In addition, swapping at different epochs from 16-bit kernel does not make a big change to the final accuracy.

For the model accuracy over the training time, we fixed the time budget to 3000 seconds, which is the time consumption of 300 epochs for the 8-bit fixed point implementation. 500 seconds was chosen as the time interval to do swapping. As seen in Figure 5.8, the slope of the curves indicates the increase rate of accuracy for fixed point implementation is higher than floating point implementation. One of the reasons is that the 8-bit fixed point consumes less times per epoch, as presented in Table 4.2. Assuming the same time consumption, the fixed point implementation takes 3 times more epochs than the floating point implementation.

For the case of 16-bit fixed point, we fixed the time budget to 4500 seconds, which is the time requirement of 300 epochs. 750 seconds was also chosen as the time interval to do swapping.

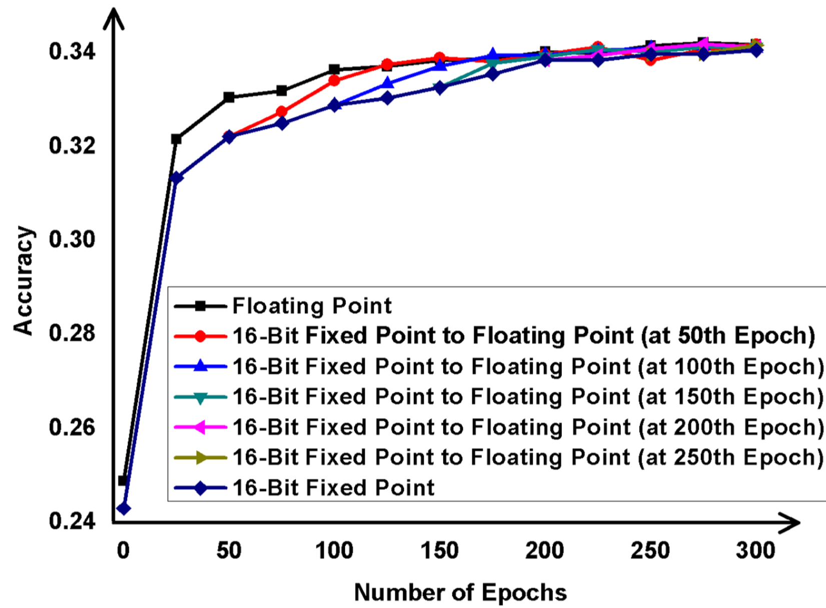


Figure 5.7: Accuracy for swapping once from 16-bit fixed point kernel to floating point kernel at different epochs

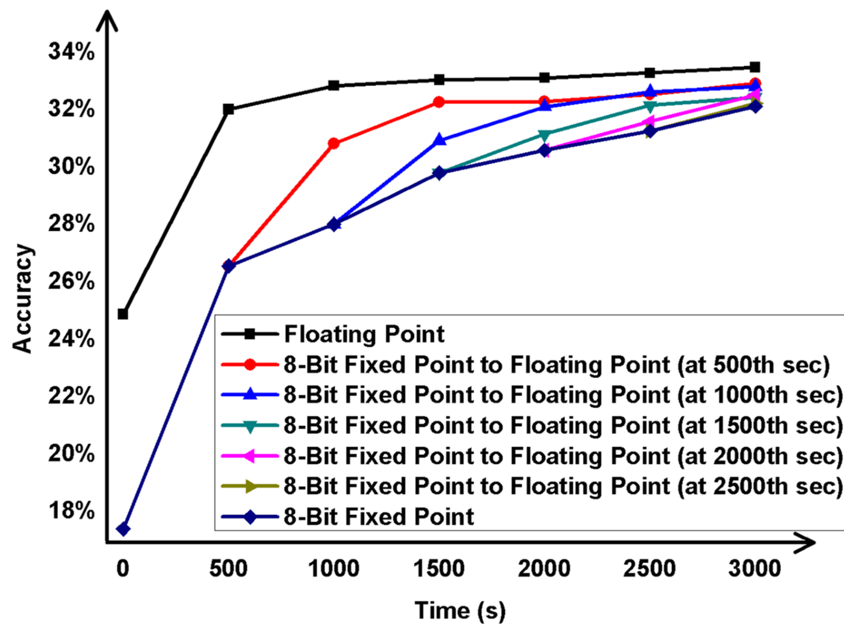


Figure 5.8: Accuracy for swapping from once 8-bit fixed point kernel to floating point kernel at different time

As seen in Figure 5.9, the swapping approach sometimes even shows a better accuracy than the floating point implementation. As the results at 2250 seconds, the accuracy of floating point implementation is 0.332, while the swapping approach can achieve 0.338. Because the training speed of 16-bit fixed point is faster than floating point implementation, the increasing rate of 16-bit fixed point is greater than floating point implementation over times.

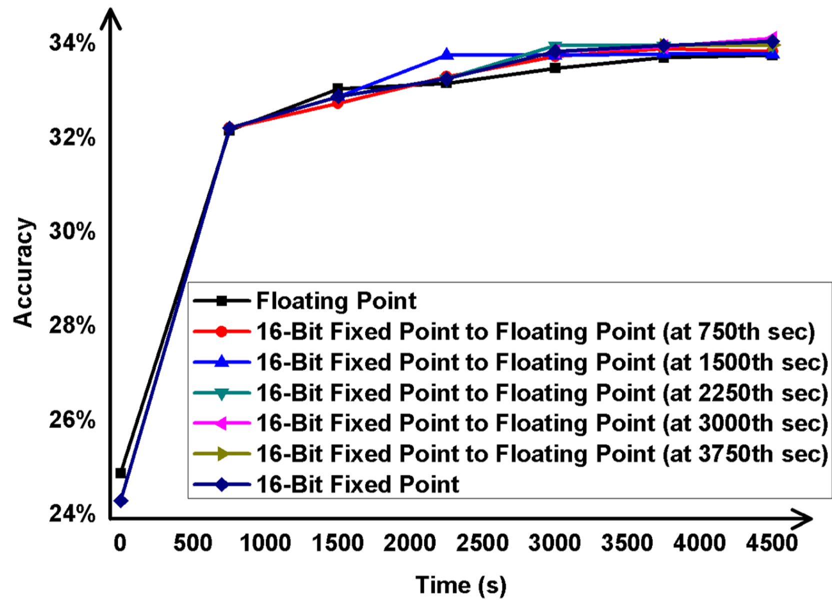


Figure 5.9: Accuracy for swapping once from 16-bit fixed point kernel to floating point kernel at different time

Based on the results above, we can see, when the number of bits is less, swapping from the fixed point kernel to floating point kernel has a more significant improvement in accuracy. Swapping at early epochs achieves a better accuracy with the same number of epochs. Using multi-kernels improves the model accuracy over the pure fixed point implementation, and reduces the training time over floating point implementation.

## 5.4 Performance Comparison

In order to evaluate the performance of the above two swapping modes, we compare the final accuracy of the model under the same amount of training time. Here the same amount of time consumption means the number of epochs for fixed point kernel should be the equal in the two modes, and also for floating point kernel. For example, the experiment results of swapping from fixed point to floating point at 250th epoch in Figure 5.2 (final accuracy is 0.329), should be compared with that of swapping from floating point to fixed point at 50th epoch in Figure 5.6 (final accuracy is 0.332), as both run 250 epochs of fixed point training plus 50 epochs of floating point training. The experiment results indicate that begins with floating point training, and then switches to fixed point training can achieve better accuracy. Swapping from floating

point kernel to fixed point kernel is more promising. It should be noted that the model accuracy is mainly decided by two factors, the initial learning result and increasing rate over learning. Floating point training achieves a better initial learning result, and less bit fixed point kernel shows a better increasing rate.

Based on the experiment results, if the time budget is limited for floating point training, swapping from floating point training to fixed point training is effective to save time. In addition, if the model accuracy needs to be guaranteed for fixed point training, swapping from fixed point training to floating point training is a valid methodology. However, under the same time budget, swapping from floating point training to fixed point training should be employed if it is allowed.

In order to achieve the best trade-off between model accuracy and training time, we need to develop a methodology to guide the selection of swapping point between different number representations. For the approach of swapping from fixed point to floating point, the training time would be too long if swapping too early, while it cannot meet the requirement of model accuracy if swapping too late. However, it is a bit different for the approach of swapping from floating point to fixed point, the problem for swapping too early is the model accuracy, while for swapping too late is the training time.

For the fixed point kernel, the bit number of representation formats should be selected carefully. We hope the value of bit number is as small as possible, since less bits consume less resource and lead to a higher degree of parallelism on FPGA boards. However, if the bit number is too small, the accuracy cannot be guaranteed, so we recommend to choose the bit number based on the results of pure fixed point implementation.

## 5.5 Expectation

Compared with the work of Chapter 3 and 4, this piece of work is more empirical. This technique can be applied to other machine learning algorithms, as long as the distribution of gradients obeys the same decreasing trend. Furthermore, the latency brought by reconfiguration

is tiny compared to the computation time, especially for training. Finally, in order to maintain consistency with Chapter 4, we take 8 bits (the acceptable bit number of batch quantisation) for fixed point implementation in this chapter, and also 16 bits (the bit number of half precision floating point) is used as a comparison. However, it is interesting to investigate the performance using fixed point with other "odd" bits.

## 5.6 Summary

This chapter introduces a new training method for position-aware Top-$k$ ListNet algorithm by exploring the unique reconfiguration capability of FPGA devices. Our novel training method which uses multi-kernels reduces the accuracy loss of low precision (fixed point) training. However, unlike pure full precision (floating point) training, it is able to balance the trade-off between the training time and the model accuracy. We also explore the swap modes to achieve higher accuracy – starting with floating point kernel and swapping to fixed point kernel is a better choice. After the promising improvements demonstrated with this work, in future we plan to apply the technique of multi-kernel training in other listwise algorithms.

# Chapter 6

# Conclusion

The previous chapters propose precomputation methods, low precision quantisation for the Top-$k$ ListNet algorithm on FPGAs to speed up training process, and kernels under different representations to improve the trade-off between the model accuracy and the training latency. This chapter starts with the summary of the current achievements of this thesis, followed by discussion on the future work.

## 6.1 Summary of Thesis Achievements

Ranking algorithms have been well-known in modern machine learning field, and they have been widely used in commercial information retrieval products [76]. However, there was no breakthrough in ranking approaches over recent years because of the high computational complexity of new ranking algorithms.

In particular, with the increasing data-set sizes and the constant demand for high computation speed, significant computational challenges have been presented to ranking algorithms. Currently, the approaches based on multi-core CPUs, GPUs, and FPGAs, have become more and more popular to accelerate ranking algorithms. However, as described in Section 2.5.2, most of previous work focused on the pointwise and pairwise approaches. In this thesis, it shows how

FPGA devices can be utilised to accelerate Top-$k$ ListNet by parallelism, word-length optimisations and reconfiguration. Great speedups against the respective state-of-the-art CPU and GPU implementations have been attained in this work.

In Chapter 3, a new computation method for the Top-$k$ ListNet algorithm is proposed, which makes it feasible to accelerate the training process using FPGA devices. Although Top-$k$ ListNet is a NN based algorithm, accelerating the dot product computation does not bring much speedup as the computational bottleneck is from the computation of permutation probabilities. Our proposed precomputation method improves the calculation speed by reducing the redundant computation. Furthermore, this method allows permutations to be computed in parallel. Speedups of 3.21x over the respective CPU implementations were achieved using the proposed design on FPGAs.

When FPGA devices are utilised for acceleration, we should keep in mind that different techniques should be used in cooperation to improve the performance. The latency can be minimised by allowing functions or loops to operate in parallel, and the throughput can be improved by pipelining functions or loops. As the results in Table 3.5 shown, both data parallelism and pipeline parallelism can attain speedup compared to original FPGA implementation without any optimisation. However, using one single technique is difficult to achieve competitive computation speed comparing with the respective CPU results.

The remaining challenge is that the maximum fanout of each variable may limit the advantage of the proposed architecture design. For NN structure with a large number of nodes, it limits the maximum degree of parallelism we can achieve on FPGA devices. To tackle this challenge, it is beneficial to store the same input data using different variables to avoid routing congestion.

Although the precomputation method can reduce the computational complexity caused by the large number of documents in each permutation, the time consumption of training is still too high with the large number of permutations. Chapter 4 therefore provides an effective sampling method, which can significantly reduce complexity and guarantee the quality of the ranking model at the same time. Furthermore, an effective quantisation method which is special for variables with wide ranges is presented in the computation of gradients. Compared to CPU-

based and GPU-based work which utilises floating point arithmetic, speedups of 40.51x and 4.42x speedups have been achieved respectively, and low precision representations also save hardware resources.

The main novelty of our proposed sampling method compared to existing sampling methods is that position-aware sampling method takes the evaluation measure into account, as the evaluation measure NDCG has an explicit position discount factor in its definition. For the batch quantisation method, it uses fixed point formats with different scaling factors to represent gradient values. Unfortunately, it is not straightforward how many batches should be classified for other applications.

The remaining challenge for the sampling technique proposed in Chapter 4 is lack of sufficient verification. This technique is verified only by Top-$k$ ListNet ($k = 3$), where the computational complexity is order of $10^9$. However, for larger values of $k$, it is difficult to demonstrate the effectiveness of proposed techniques as the high complexity makes it impractical to have an implementation. In this case, we can only implement Top-$k$ ListNet ($k > 3$) on a data set with limited number of documents.

The main limitation for the proposed quantisation method proposed in Chapter 4 is that both floating point implementation and fixed point implementation cannot balance the trade-off between the model quality and the training time, which is unacceptable for some applications. This motivates the work in Chapter 5 that aims to improve the accuracy of low precision training and minimise the training time of floating point implementation.

In Chapter 5, a new training method that using multiply kernels is proposed which improves the trade-off between the training time and the model accuracy by modifying the hardware during runtime. This method utilises the distinct reconfigurable feature of FPGA devices, whereby the function on board is changed at runtime. By implementing the proposed method for position-aware Top-$k$ ListNet on FPGA, a higher model accuracy was achieved over the respective fixed point training, and a better latency is attained over the traditional floating point training.

Related work focused on full reconfiguration application is quite limited in academic literature, although a wide range of applications exploited partial reconfiguration implementation. For full reconfiguration, the reconfiguration time was the key problem in the past. However, the high level synthesis tools for OpenCL abstract the integration of user-designed accelerators with the host system, which increases the reconfiguration speed. The emerging interest in using FPGAs will bring the reconfiguration being widely used.

The main shortcoming of the proposed reconfigurable method is that it only supports swapping once, which cannot adjust the data representation flexibly based on the trend of the accuracy over epochs. Thus it limits the improvement we can achieve on FPGA even the reconfiguration time can be neglected. Further optimisation can be made by proposing more effective ways, which support to swap between different kernels for multiple times. For example, we can define a threshold value to trigger swapping automatically, to achieve better balance between the training latency and the model accuracy.

To summarize, the core breakthrough of this research is that we deal with the bottleneck of computational complexity for the Top-$k$ ListNet algorithm in big data applications by using FPGA platforms. By taking advantage of the differentiating feature of FPGAs, such as fine-grained parallelism, custom precision support and reconfiguration capability, a considerable amount of speedup is achieved. Under the FPGA-based accelerators, the techniques proposed in this thesis can be applied to bring the listwise algorithm closer to real-world applications. For example, with the above techniques, the training time of Top-$k$ ListNet ($k = 3$) is at the same level as commercial ranking approaches, which makes it feasible to implement Top-$k$ ListNet ($k = 3$) in real information retrieval applications.

It should be noted that the proposed techniques are developed for accelerating listwise approaches using FPGAs, they may be not generic enough for applying in other machine learning algorithms, and also other hardware platforms.

## 6.2 Future Work

At the end of this thesis, it is a good time to discuss several potential directions we can extended in, including developing new ranking algorithms, new sampling approaches, and using the high bandwidth of FPGA devices to further improve the performance. In addition, power consumption is an important topic.

- There are some other listwise algorithms that are quite popular in ranking besides Top-$k$ ListNet presented in this thesis. For example, the ListMLE is a family of listwise approach based on the *Plackett-Luce* model, which uses the negative log likelihood of the ground truth permutation as the Loss Function. The complexity of ListMLE is $O(n_d)$, which equals to the number of documents for each query. An interesting research direction is to involve more permutations in gradient decent computation to improve the accuracy of final model. The main disadvantage of implementing more permutations is increasing the computational complexity of ListMLE. However, parallel computation platforms such as FPGAs can be utilised to solve this challenge.

- With respect to the work presented in Chapter 4, the position-aware sampling method proposed is designed for Top-$k$ ListNet. Future work can focus on the implementation of proposed sampling methods in different listwise approaches with comparisons of position-aware Top-$k$ ListNet.

- The newest GPUs also support low precision computation, therefore it would be an interesting research direction to implement our proposed quantisation approach in Chapter 4 in GPUs to compare its performance to that of FPGAs in this work.

- Another important research direction is to investigate custom precision representation for Forward Propagation and Loss Function. As shown in Chapter 4, it is beneficial to use low precision representation to reduce resource consumption. It would be interesting to see how the accuracy will change if we extend low precision representation to the Forward Propagation and Loss Function.

- A reconfiguration method is proposed to improve the trade-off between the model accuracy and the training latency in Chapter 5. However, this methodology still has space to be improved. The times of kernel swap and threshold value of swapping point, taking into account the model accuracy and the training latency, can be explored to achieve the best performance.

- Since we have achieved speed improvement in learning the Top-$k$ ListNet algorithm, by using the parallel computation capability, custom precision representation and reconfiguration of FPGA devives. It is interesting to explore how to improve the performance by utilising the high bandwidth of on-chip memory of FPGA devices. The main emphasis then should be in exploiting how to compress and store the training data in on-chip memory effectively.

- Both software and hardware techniques have been represented to improve the speed of the Top-$k$ algorithm in previous chapters. In addition, it is quite interesting to investigate how the power consumption is affected by applying these techniques, since the proposed techniques improve the computational efficiency, where the metric is FLOP/s/DSP. The power monitor can be utilised to analysis the power consumption under different computation designs.

- An important direction to accelerate Top-$k$ ListNet is to utilize multiple FPGAs, since a single device is difficult to properly process the whole problem with the limited resources. By appropriately splitting the problem into different tasks that can be executed in parallel and mapping them into different FPGAs, an impressive speedup can be attained. The memory bandwidth needs to be followed for the data transfer between different devices, in order to reduce the communication time.

- Finally, as the biggest weakness of this work is associated with the data set. In our work, only one data set – LETOR 4.0 is used, as mentioned before, the reason is that only in this data set the ground truth is given as a permutation for a query, instead of multiple level relevance judgements. However, it is still necessary to evaluate the performance of proposed techniques on multiple data sets.

# Bibliography


[1] T.-Y. Liu *et al.*, “Learning to Rank for Information Retrieval,” *Foundations and Trends in Information Retrieval*, 2009.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Stanford InfoLab, Tech. Rep., 1999.

[3] S. Barberà, W. Bossert, and P. K. Pattanaik, “Ranking sets of objects,” in *Handbook of utility theory.* Springer, 2004, pp. 893–977.

[4] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme, “Information retrieval in folksonomies: Search and ranking,” in *European semantic web conference.* Springer, 2006, pp. 411–426.

[5] G. Bouma, “Normalized (pointwise) mutual information in collocation extraction,” *Proceedings of GSCL*, pp. 31–40, 2009.

[6] H. H. Pareek and P. K. Ravikumar, “A representation theory for ranking functions,” in *Advances in Neural Information Processing Systems*, 2014, pp. 361–369.

[7] E. Hüllermeier, J. Fürnkranz, W. Cheng, and K. Brinker, “Label ranking by learning pairwise preferences,” *Artificial Intelligence*, vol. 172, no. 16-17, pp. 1897–1916, 2008.

[8] X. Chen, P. N. Bennett, K. Collins-Thompson, and E. Horvitz, “Pairwise ranking aggregation in a crowdsourced setting,” in *Proceedings of the sixth ACM international conference on Web search and data mining.* ACM, 2013, pp. 193–202.

[9] V. Melnikov, P. Gupta, B. Frick, D. Kaimann, and E. Hüllermeier, “Pairwise versus pointwise ranking: A case study,” *Schedae Informaticae*, vol. 25, pp. 73–83, 2016.

[10] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, and H. Li, "Listwise Approach to Learning to Rank: Theory and Algorithm," in *Proceedings of the 25th international conference on Machine learning*, 2008.

[11] R. Jagerman, J. Kiseleva, and M. de Rijke, "Modeling label ambiguity for neural list-wise learning to rank," *arXiv preprint arXiv:1707.07493*, 2017.

[12] Y. Shi, M. Larson, and A. Hanjalic, "List-wise learning to rank with matrix factorization for collaborative filtering," in *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 2010, pp. 269–272.

[13] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: from pairwise approach to listwise approach," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 129–136.

[14] Y. Lan, S. Niu, J. Guo, and X. Cheng, "Is top-k sufficient for ranking?" in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 1261–1270.

[15] Y. Liu, X. Zhang, X. Zhu, Q. Guan, and X. Zhao, "Listnet-based Object Proposals Ranking," *Neurocomputing*, 2017.

[16] N. Dai and B. D. Davison, "Capturing page freshness for web search," in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2010, pp. 871–872.

[17] R. Zhang, Y. Konda, A. Dong, P. Kolari, Y. Chang, and Z. Zheng, "Learning recurrent event queries for web search," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2010, pp. 1129–1139.

[18] T. Moon, L. Li, W. Chu, C. Liao, Z. Zheng, and Y. Chang, "Online learning for recency search ranking using real-time user feedback," in *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM, 2010, pp. 1501–1504.

[19] F. Diaz, "Integration of news content into web results," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining.* ACM, 2009, pp. 182–191.

[20] H. Wang, A. Dong, L. Li, Y. Chang, and E. Gabrilovich, "Joint Relevance and Freshness Learning from Clickthroughs for News Search," in *Proceedings of the 21st international conference on World Wide Web*, 2012.

[21] M. S. Rehman, K. Kothapalli, and P. Narayanan, "Fast and scalable list ranking on the GPU," in *Proceedings of the 23rd international conference on supercomputing.* ACM, 2009, pp. 235–243.

[22] D. C. G. Pedronette, E. Borin, M. Breternitz *et al.*, "Efficient image re-ranking computation on GPUs," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications.* IEEE, 2012, pp. 95–102.

[23] G. Ortega, E. Filatovas, E. Garzón, and L. G. Casado, "Non-dominated sorting procedure for Pareto dominance ranking on multicore CPU and/or GPU," *Journal of Global Optimization*, vol. 69, no. 3, pp. 607–627, 2017.

[24] B. Cope *et al.*, "Implementation of 2D Convolution on FPGA, GPU and CPU," *Imperial College Report*, pp. 2–5, 2006.

[25] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *2009 international conference on field programmable logic and applications.* IEEE, 2009, pp. 126–131.

[26] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI.* IEEE, 2010, pp. 288–293.

[27] J. Yan, N.-Y. Xu, X.-F. Cai, R. Gao, Y. Wang, R. Luo, and F.-H. Hsu, "An FPGA-based accelerator for LambdaRank in Web search engines," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2011.

[28] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to Rank Using Gradient Descent," in *Proceedings of the 22nd international conference on Machine learning*, 2005.

[29] T. Qin, X. Geng, and T.-Y. Liu, "A New Probabilistic Model for Rank Aggregation," in *Advances in neural information processing systems*, 2010.

[30] C. J. Burges, "From Ranknet to Lambdarank: An overview," *Learning*, 2010.

[31] Y. Lan, Y. Zhu, J. Guo, S. Niu, and X. Cheng, "Position-Aware ListMLE: A Sequential Learning Process for Ranking," in *UAI*, 2014, pp. 449–458.

[32] T. Luo, D. Wang, R. Liu, and Y. Pan, "Stochastic top-k listnet," *arXiv preprint arXiv:1511.00271*, 2015.

[33] F. Xia, T.-Y. Liu, and H. Li, "Statistical consistency of top-k ranking," in *Advances in Neural Information Processing Systems*, 2009, pp. 2098–2106.

[34] S. Shukla, M. Lease, and A. Tewari, "Parallelizing ListNet training using spark," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, 2012.

[35] D. X. De Sousa, T. C. Rosa, W. S. Martins, R. Silva, and M. A. Gonçalves, "Improving on-demand learning to rank through parallelism," in *International Conference on Web Information Systems Engineering*, 2012.

[36] N.-Y. Xu, X.-F. Cai, R. Gao, L. Zhang, and F.-H. Hsu, "FPGA acceleration of rankboost in web search engines," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2009.

[37] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," *ACM SIGARCH Computer Architecture News*, 2014.

[38] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *International Conference on Learning Representations*, 2015.

[39] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015.

[40] X. Han, D. Zhou, S. Wang, and S. Kimura, "CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 320–327.

[41] Z. Zhang, D. Zhou, S. Wang, and S. Kimura, "Quad-multiplier packing based on customized floating point for convolutional neural networks on FPGA," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*, 2018.

[42] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[43] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[44] S. Donthi and R. L. Haggard, "A survey of dynamically reconfigurable FPGA devices," in *Proceedings of the 35th Southeastern Symposium on System Theory, 2003.* IEEE, 2003, pp. 422–426.

[45] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in virtex fpgas," *IEE Proceedings-Computers and Digital Techniques*, vol. 153, no. 3, pp. 157–164, 2006.

[46] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, p. 36, 2011.

[47] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 72, 2018.

[48] H. M. Hussain, K. Benkrid, A. Ebrahim, A. T. Erdogan, and H. Seker, "Novel dynamic partial reconfiguration implementation of k-means clustering on FPGAs: Comparative results with GPPs and GPUs," *International Journal of Reconfigurable Computing*, vol. 2012, p. 1, 2012.

[49] H. M. Hussain, K. Benkrid, and H. Seker, "Reconfiguration-based implementation of SVM classifier on FPGA for classifying microarray data," in *2013 35th Annual international conference of the IEEE engineering in medicine and biology society (EMBC)*. IEEE, 2013, pp. 3058–3061.

[50] H. Hussain, K. Benkrid, and H. ŞEKER, "Novel dynamic partial reconfiguration implementations of the support vector machine classifier on FPGA," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 24, no. 5, pp. 3371–3387, 2016.

[51] C. Patterson, "High performance DES encryption in Virtex/sup TM/FPGAs using JBits/sup TM," in *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00871)*. IEEE, 2000, pp. 113–121.

[52] J. Noguera and I. O. Kennedy, "Power Reduction in Network Equipment through Adaptive Partial Reconfiguration," in *FPL*, 2007, pp. 240–245.

[53] S. U. Bhandari, S. Subbaraman, S. Pujari, and R. Mahajan, "Real time video processing on FPGA using on the fly partial reconfiguration," in *2009 International Conference on Signal Processing Systems*. IEEE, 2009, pp. 244–247.

[54] B. Krill, A. Amira, A. Ahmad, and H. Rabah, "A new FPGA-based dynamic partial reconfiguration design flow and environment for image processing applications," in *2010 2nd European Workshop on Visual Information Processing (EUVIP)*. IEEE, 2010, pp. 226–231.

[55] S. Venieris, "Automated methodologies for mapping convolutional neural networks on reconfigurable hardware," 2018.

[56] S. I. Venieris and C.-S. Bouganis, "Latency-driven design for FPGA-based convolutional neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[57] ——, "fpgaConvNet: A toolflow for mapping diverse convolutional neural networks on embedded FPGAs," *arXiv preprint arXiv:1711.08740*, 2017.

[58] Y. Liu, K. Benkrid, A. Benkrid, and S. Kasap, "An fpga-based web server for high performance biological sequence alignment," in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE, 2009, pp. 361–368.

[59] A. Nikitakis and L. Papaefstathiou, "A memory-efficient FPGA-based classification engine," in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2008, pp. 53–62.

[60] J. Yan, Z.-X. Zhao, N.-Y. Xu, X. Jin, L.-T. Zhang, and F.-H. Hsu, "Efficient query processing for web search engine with FPGAs," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 97–100.

[61] Vdang, "https://sourceforge.net/p/lemur/wiki/RankLib/," in *RankLib*, 2012.

[62] J. Huang and C. Guestrin, "Riffled Independence for Ranked Data," in *Advances in Neural Information Processing Systems*, 2009.

[63] J. Huang, A. Kapoor, and C. Guestrin, "Riffled independence for efficient inference with partial rankings," *Journal of Artificial Intelligence Research*, vol. 44, pp. 491–532, 2012.

[64] J. Huang, C. Guestrin *et al.*, "Uncovering the riffled independence structure of ranked data," *Electronic Journal of Statistics*, vol. 6, pp. 199–230, 2012.

[65] T. Lan and G. Mori, "A max-margin riffled independence model for image tag ranking," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 3103–3110.

[66] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[67] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[68] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proceedings of Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[69] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *arXiv preprint arXiv:1703.03073*, 2017.

[70] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[71] Y. Lee, Y. Choi, S.-B. Ko, and M. H. Lee, "Performance analysis of bit-width reduced floating-point arithmetic units in FPGAs: a case study of neural network-based face detector," *EURASIP Journal on Embedded Systems*, vol. 2009, p. 4, 2009.

[72] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in neural information processing systems*, 2017, pp. 1742–1752.

[73] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[74] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[75] X. Chen, X. Hu, H. Zhou, and N. Xu, “Fxpnet: Training a deep convolutional neural network in fixed-point representation,” in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2494–2501.

[76] B. N. Bullock, A. Hotho, and G. Stumme, “Accessing information with tags: search and ranking,” in *Social Information Access*. Springer, 2018, pp. 310–343.